Chapter 1: Introduction to Data Structures

What are we studying in this chapter?

- Introduction to data structures
- Classification of data structures: Primitive and non-primitive data structures
- Data structure operations
  - Inserting, deleting
  - Traversing, searching, sorting and merging

1.1 Introduction

In this chapter, let us discuss about the definition of data structures and see how the data structures are classified. Basic terminology and concepts are defined along with examples. The way the data is represented, organized, managed and structured is crucial part of the data structures. The data to be manipulated can be used individually or in a group under one common name and data can be of different types.

1.2 Basic terminology

First, let us see "*What is data? What is information?*"

Definition: The word *data* is the plural of *datum*. *Data* is a representation of facts or concepts in an organized manner. The data may be stored, communicated, interpreted, or processed. In short, we can say that *data* is a piece of information or simply set of values and data as such may not convey any meaning.

For example, numbers, names, marks etc are the data. Consider the two data items "Rama" and 10. The data "Rama" and 10 does not convey any meaning to the reader.

Definition: *Information* is defined as a collection of data from which conclusions may be drawn. So, information is subset of *data* which when interpreted conveys meaning so that the people can understand. Hence, Information can be considered as a message that is received and understood.

Ex 1: The output of the computer may be "Rama scored 10 marks". This sentence is an information. Because, it conveys meaning to the reader.

Ex 2: The instructions given to a person or commands issued to the computer are considered as information.

1.2 💻 Introduction to data structures and arrays

Now, let us see *"What are the differences between data and information?"*

Data	Information
1. Data is collection of facts and figures (may be numbers). Facts are the things done or things existing.	1. Information is the result of processing the data. So, information is collection of data from which we can draw conclusions.
2. Data do not convey any meaning	2. Information conveys meaning
3. Computers work with data	3. Computers do not work with information
4. Data is unstructured, lacks context and may not be relevant to the recipient.	4. When data is correctly organized, processed, filtered and presented with context, it can become information because it then has meaning to the recipient
5. Data is the representation of information. So, data representation may change	5. Information is the things that we know. So, even though data representation changes, information do not change

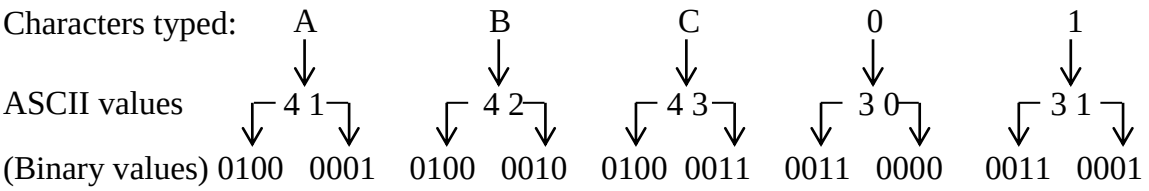
Note: Let us take an example relating to the last difference. Consider the statement "My daughter is seven years old". I can represent may daughter's age as "seven in words" or "7 in figures" or "VII in roman" or "111 in binary" or "1111111 in unary" and so on. All these representations we call as *data*. Even though representation changes, my daughter's age has not changed. So, we say that *information* does not change and *data* representation may change.

Note: Even though normally we use *data* and *information* interchangeably, they are totally different and they are not synonyms.

Now, let us see "How computers represent data?"

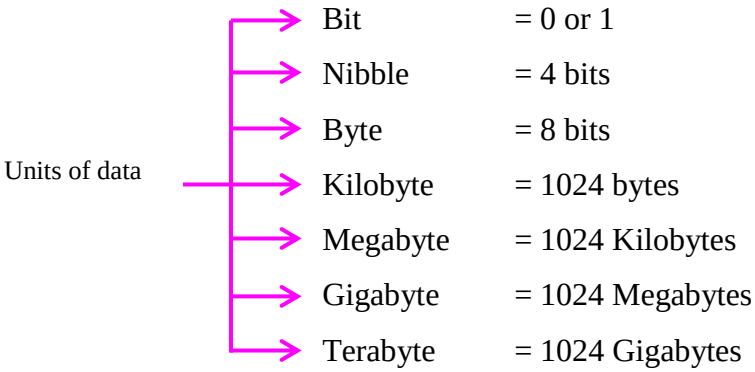
The data are represented by the state of electronic switches. A switch with ON state represents 1 and a switch with OFF state represents 0 (zero). Thus, all digital computers use *binary number system* to represent the data. So, the data that we input to the computer is converted into 0's and 1's. The letters, digits, punctuation marks, sound and even pictures are represented using 1's and 0's. Thus, *computers represent data using binary digits 0 and 1 i.e., in binary number system.*

For example, consider the string "ABC01" that we type from the keyboard. This data can be represented in the computer as shown below.



Note: In general, any data whether numerical or non-numerical is represented using 1's and 0's.

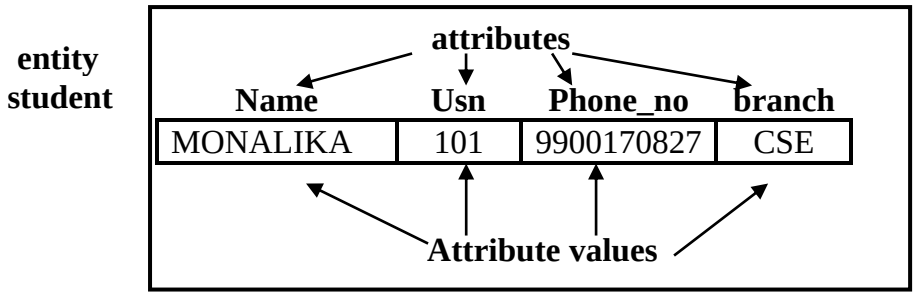
All the quantities are measured in some units. For example, length may be measured in meters or feet. On similar lines, to measure computer memory, we require units. Now, let us see *"How does the data represented using 1's and 0's can be grouped or measured?"* The data represented can be grouped or measured using following units:



Now, let us see "What is an entity and what is an attribute?"

Definition: An *entity* can be defined as a person, place, thing or concept about which data can be collected and stored. An entity is made up of a number of attributes which represent that entity. An *attribute* is the one which describes the facts, details or characteristics of an entity.

For example, if we collect *name*, *usn*, *phone number*, *branch of engineering* of a particular student, then *student* is treated as an entity whereas *name, usn, phone number, branch* are the attributes. The entity *student* and its attributes are pictorially represented as shown below:



Now, let us see "What is an entity set?"

Definition: A collection of entities with similar attributes is an *entity set*. For example, all the students in a college represent entity set. The entity set of students can be pictorially represented as shown below:

Entity set

Name	Usn	Phone_no	branch
MONALIKA	101	9900170827	CSE
KRISHNA	102	999999999	MECH
NARENDRA	103	9845070827	CIVIL
MITHIL	104	9844010200	ISE
PADMA	105	9844012222	EC

Now, let us see "In what way the attributes, entities and entity sets are related to *fields, records* and *files*?" The way the data are organized into the hierarchy of *fields*, *records* and *files* reflect the relationship between *attributes, entities* and *entity sets*.

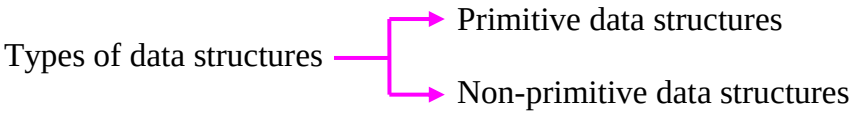
Now, let us see "What is a field? What is a record? What is a file"

Definition: A field is a single elementary unit of information representing an attribute of an entity. A record is a collection of field values of a given entity. A file is a collection of records of the entities in a given entity set.

1.3 Data Structures

In this section, we shall "Define data structures?"

Definition: The study of how the data is collected and stored in the memory, how efficiently the data is organized in the memory, how efficiently the data can be retrieved and manipulated, and the possible ways in which different data items are logically related is called data structures. The data structures are mainly classified as shown below:



1.3.1 Primitive Data Structures

Now, let us see *"What are primitive data structures?"*

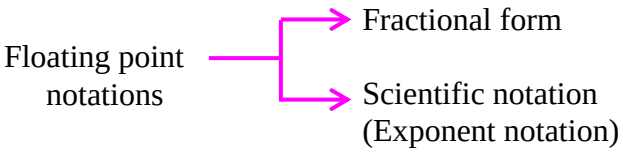
Definition: The data structures that can be manipulated directly by machine instructions are called *primitive data structures.* Thus, the primitive data structures are fundamental data types that are supported by a programming language. The primitive data structures are also called *simple data structures.*

For example, in C language, the different primitive data structures are defined using the data types such as **int, float, char, double** and **pointers.**

Integers: An integer is a whole number without any decimal point or a fraction part. No extra characters are allowed other than '+' and '–' sign. If '+' and '–' are present, they should precede the number. The integers are normally represented in binary or hexadecimal. *All negative numbers are represented using 2's complement.* Based on the sign, the integers are classified into:

- Unsigned integer
- Signed integer

Floating point number: The floating point constants are base 10 numbers with fraction part such as 10.5. All negative numbers should have a prefix '–'. A positive number can have an optional '+' sign. No other extra characters are allowed. The floating point constants can be represented using two forms as shown below:



Fractional form: A floating point number represented using *fractional form* has an integer part followed by a dot and a fractional part. We can omit the digits before the decimal point or after the decimal point. For example, 0.5, -0.99, -.6, -9., +.9 etc are all valid floating point numbers.

Exponent form (Scientific notation): The floating point number represented using *scientific notation* (also called *exponential notation*) has three parts namely:

mantissa e/E exponent.

Ex1:	9.86	E	3	imply 9.86×10^{3}
Ex2:	9.86	e	-3	imply 9.86×10^{-3}

where

- The mantissa can be an integer or a floating point number represented using fractional notation.
- The letter *e* or *E* should follow mantissa.
- The exponent should follow *e* or *E*. The exponent has to be an integer with optional '+' or '–' sign.

For example,

6.698274e2 means 6.698274×10^{2}
-0.36e-54 means -0.36×10^{-54}

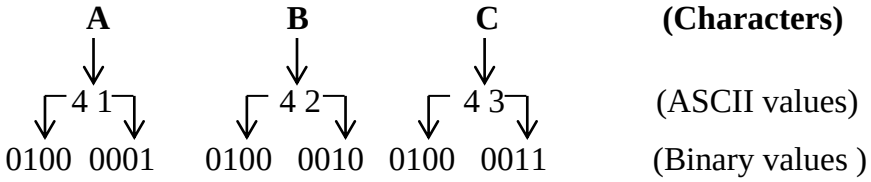
Character: A character data is a primitive data structure. A wide variety of character sets (also called alphabets) are handled by computers. The two widely used character sets are:

- **ASCII** (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange)
- **EBCDIC**(**E**xtended **B**inary **C**oded **D**ecimal **I**nterchange **C**ode)

Note: **ASCII** (pronounced as *as-key*) code is a 7-bit code to represent various characters such as letters, digits, punctuation marks and other symbols. **ASCII** stands for **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange. The ASCII Table is shown below.

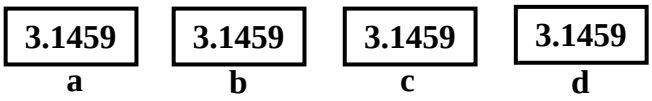
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Observe from the table that character 'A' has an ASCII value 41, the character 'B' has an ASCII value 42 and character 'C' has an ASCII value 43. So, if we type the text "ABC" from the keyboard, to the computer, it appears as shown below:

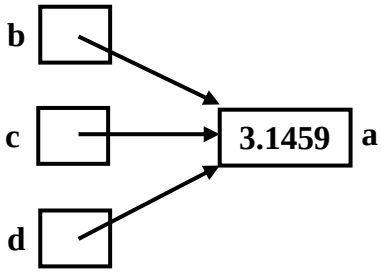


Pointer: A pointer is a special variable which contains address of a memory location. Using this pointer, the data can be accessed.

For example, assume that a program contains four occurrences of a constant 3.1459. During the compilation process, four copies of 3.1459 can be created as shown below:



However, it is more efficient to use one copy of 3.1459 and three pointers referencing a single copy, since less space is required for a pointer when compared to floating point number. This can be represented pictorially as shown below:



The variables *b, c* and *d* are called pointers since they contain address of variable *a*.

1.3.2 Non-primitive Data Structures

Now, let us see *"What are non-primitive data structures? What are the different types of non-primitive data structures?"*

Definition: The data structures that cannot be manipulated directly by machine instructions are called *non-primitive data structures.* Thus, the non-primitive data structures are created or constructed using primitive data structures. The non-primitive data structures are also called *compound data structures.*

For example, arrays, structures, stacks, queues, linked lists, trees, graphs, files etc., are some of the non- primitive data structures.
Now, let us see "What are the different types of non-primitive data structures?" The non-primitive data structures are classified as shown below:

- Linear data structures **Ex:** Arrays, stacks, queues, linked lists
- Non-linear data structures **Ex:** Trees, graphs, files etc.

1.8 💻 Introduction to data structures and arrays

Now, let us see "What are linear data structures?"

Definition: The data structure where its values or elements are stored in a *sequential* or *linear order* is called *linear data structure.* The elements can be accessed sequentially one after the other in a linear order and hence the name. The linear relationship between the elements is maintained:

- by means of sequential memory locations as in arrays: Here, one or more elements are stored one after the other sequentially.
- by means of links as in linked list: Here, each element is stored in a node. One or more nodes are logically adjacent and the logical adjacency is maintained using links. In linked lists, even though the nodes are not physically adjacent, they seem to be adjacent logically because of links. The logical adjacency is maintained using pointers.

For example, arrays, stacks, queues, linked lists etc., are all linear data structures.

Now, let us see "What are non-linear data structures?"

Definition: The data structure where its values or elements are not stored in a *sequential* or *linear order* is called *non-linear data structures.* Unlike linear data structures, here, the logical adjacency between the elements is not maintained and hence elements cannot be accessed if we go in sequential order. In non-linear data structures, a data item (or an element) could be attached to several other data items.

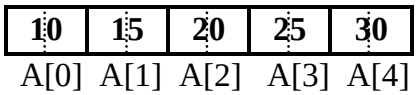
For example, graphs, trees, files are all non-linear data structures.

Now, let us "Explain non-linear data structures?" All the non-linear data structures such as arrays, stacks, queues, linked lists, graphs, trees and files are discussed below:

- **Arrays: Definition:** An array is a special and very powerful data structure in C language. An array is a collection of similar data items. All elements of the array share a common name. Each element in the array can be accessed by the subscript (or index). Array is used to store, process and print large amount of data using a single variable.

  Ex 1: Set of integers, set of characters, set of students, set of pens etc. are examples of various arrays.

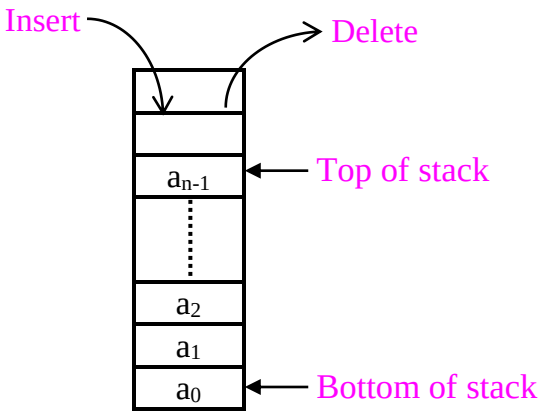
  Ex 2: An array of 5 integers is pictorially represented as shown below:



Ex 3: An array of 5 characters is pictorially represented as shown below:

'A'	'B'	'C'	'D'	'E'
A[0]	A[1]	A[2]	A[3]	A[4]

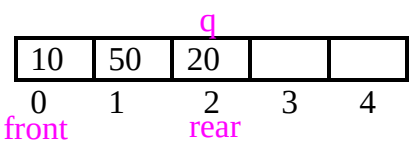
- **Stack:** A stack is a special type of data structure (linear data structure) where elements are inserted from one end and elements are deleted from the same end. Using this approach, the **L**ast element **I**nserted is the **F**irst element to be deleted **O**ut, and hence, stack is also called **L**ast **I**n **F**irst **O**ut **(LIFO)** data structure. The stack s={a_0, a_1, a_2,……a_{n-1}) is pictorially represented as shown below:



The elements are inserted into the stack in the order a_0, a_1, a_2,……a_{n-1}. That is, we insert a_0 first, a_1 next and so on. The item a_{n-1} is inserted at the end. Since, it is on top of the stack, it is the first item to be delted. The various operations performed on stack are:

- **Insert:** An element is inserted from top end. Insertion operation is called push operation
- **Delete:** An element is deleted from top end only. Deletion operation is called pop operation.
- **Overflow:** Check whether the stack is full or not.
- **Underflow:** Check whether the stack is empty or not.

- **Queue:** A queue is a special type of data structure (linear data structure) where elements are inserted from one end and elements are deleted from the other end. The end at which new elements are added is called the *rear* and the end from which elements are deleted is called the *front.* Using this approach, the **F**irst element **I**nserted is the **F**irst element to be deleted **O**ut, and hence, queue is also called **F**irst **I**n **F**irst **O**ut **(FIFO)** data structure.
  For example, consider the queue shown below having the elements 10, 50 and 20:



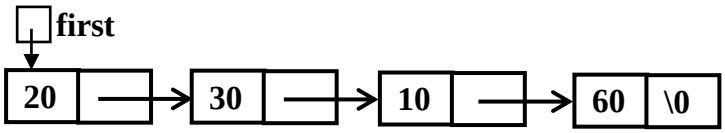
- The items are inserted into queue in the order 10, 50 and 20. The variable *q* is used as an array to hold these elements

- Item 10 is the first element inserted. So, the variable *front* is used as index to the first element
- Item 20 is the last element inserted. So, the variable *rear* is used as index to the last element

♦ **Linked lists:** A linked list is a data structure which is collection of zero or more nodes where each node is connected to the next node. If each node in the list has only one link, it is called singly linked list. If it has two links one containing the address of the next node and other link containing the address of the previous node it is called doubly linked list. Each node in the singly list has two fields namely:

- **info** – This field is used to store the data or information to be manipulated
- **link** – This field contains address of the next node.

The pictorial representation of a singly linked list where each node is connected to the next node is shown below:

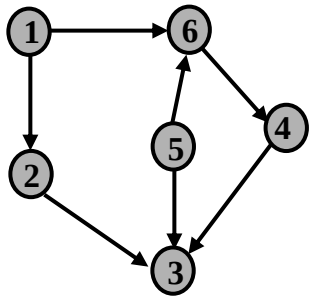


The above list consists of four nodes with info fields containing the data items 20, 30, 10 and 60.

♦ **Graphs:** A graph is a non-linear data structure which is a collection of vertices called nodes and the edges that connect these vertices. Formally, a graph **G** is defined as a pair of two sets V and E denoted by

G = (V, E)

where V is set of vertices and E is set of edges. For example, consider the graph shown below:



Here, graph G = (V, E) where

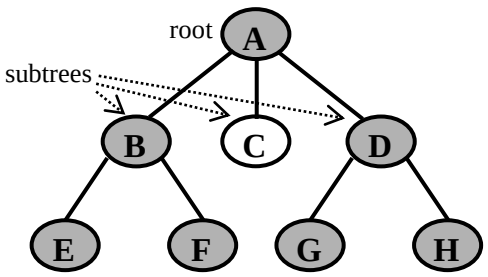
- ♦ V = {1, 2, 3, 4, 5, 6} is set of vertices
- ♦ E = { (1, 6), (1, 2), (2, 3), (4, 3), (5, 3), (5, 6), (6, 4) } is set of edges

Note:|V| = |{1, 2, 3, 4, 5, 6}| = 6 represent the number of vertices in the graph.

- |E| = |{(1, 6), (1, 2), (2, 3), (4, 3), (5, 3), (5, 6), (6, 4) }| = 7 represent the number of edges in the graph.

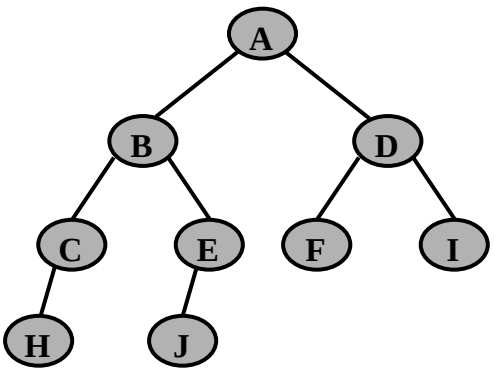
- **Trees:** A tree is a set of finite set of one or more nodes that shows parent-child relation such that:
  - There is a special node called the root node
  - The remaining nodes are partitioned into disjoint subsets $T_1, T_2, \ldots\ldots T_n$, $n \geq 0$ where $T_1, T_2, T_3 \ldots\ldots T_n$ which are all children of root node are themselves trees called *subtrees*.

Ex1 : Consider the following tree. Let us identify the root node and various subtrees:



- The tree has 8 nodes : A, B, C, D, E, F, G and H.
- The node A is the *root* of the tree
- We normally draw the trees with root at the top
- The node B, C and D are the children of node A and hence there are 3 subtrees identified by B, C and D
- The node A is the parent of B, C and D whereas D is the parent of G and H

- **Binary tree:** A binary tree is a tree which is collection of zero or more nodes and finite set of directed lines called branches that connect the nodes. A tree can be empty or partitioned into three subgroups namely root, left subtree and right subtree.
  - Root – If tree is not empty, the first node is called root node.
  - left subtree – It is a tree which is connected to the left of root. Since this tree comes under root, it is called left subtree.
  - right subtree – It is a tree which is connected to the right of root. Since this tree comes under the root, it is called right subtree.



1.4 Data structure operations

In this section, let us see "What are the operations performed on various data structures?" The operations performed on various data structures are:

- **Creating:** The process of repatedly adding various data items into the list is called creating a list. For example, we can create a student records consisting of names,

marks, address, phone number etc., by repeatedly adding or inserting data items into the list.

- **Inserting:** The process of adding a data item into the list is called inserting. For example, a student record consisting of name, marks and address can be added whenever a student joins a course. It is called inserting student record into the list.
- **Deleting:** The process of removing a data item from a list is called deleting. For example, removing a student record from the list whenever the student leaves the course in the middle.
- **Searching:** The process of informing whether a particular item is present in a list or it is not present in a list is called searching. We can also find a set of elements based on some criteria. For example, display student names who have passed in all subjects or who have scored more than 90%.
- **Sorting:** The process of arranging various data items either in ascending order or descending order is called sorting. For example, in a student record consisting of names, marks, phone number etc., we can arrange student records based on alphabetical order of names or based on ascending order of marks etc.
- **Merging:** Given two sorted lists we can combine those two lists into a single sorted list. This process is called merging. For example, two sorted arrays can be combined into a single sorted array. This process is called simple merge.
- **Traversing:** The process of accessing each data item exactly once so that it can be processed and manipulated is called traversal. For example, to print all array elements, to display all student names in a class, to display each item in a list etc.

Each of the operations can be performed one after the other. For example, given an item, we may have to traverse the list and during this process we can search for the item in the list. If the item is present in the list, we may delete that item and insert another item in that place. Then we may have to sort the resulting list and display the sorted list.

Exercises

1) What is data? What is information?
2) What are the differences between data and information?
3) Define the terms: entity, attribute, entity set, field, record, file
4) Define data structures and types of data structures
5) What are primitive data structures? Explain
6) What are non-primitive data structures? Explain
7) Explain the different types of non-primitive data structures
8) What are the different types of number systems that are commonly used?
9) What are linear data structures? What are non-linear data structures? Explain
10) What are the operations performed on various data structures?

Chapter 2: Arrays

What are we studying in this chapter?

- Arrays: Definition and representation of linear arrays in memory
- Dynamically allocated arrays (Discussed in chapter 3)
- Array operations
  - Traversing, Inserting and deleting
  - Searching and sorting
- Multi-dimensional arrays
- Application of arrays:
  - Polynomials (discussed in structures and unions – chapter 5)
  - sparse matrices (discussed in structures and unions – chapter 5)

2.1 Introduction

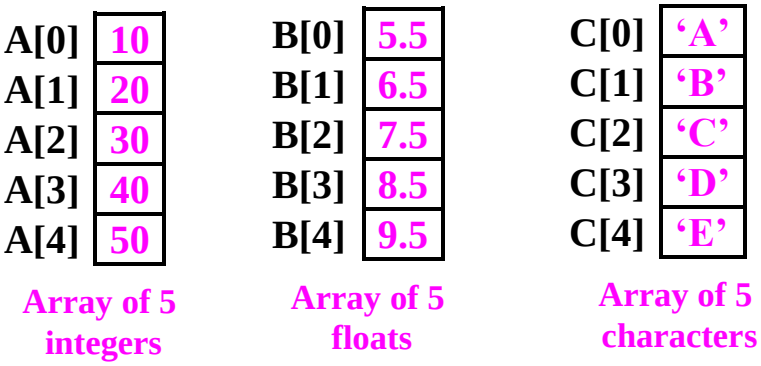
In this section, let us see array concepts in detail. First, let us see "What is an array?"

Definition: An array is a special and very powerful data structure in C language. An array is a collection of similar data items. All elements of the array share a common name. Each element in the array can be accessed by the subscript (or index). Array is used to store, process and print large amount of data using a single variable.

Ex 1: Set of integers, set of characters, set of students, set of pens etc. are examples of various arrays.

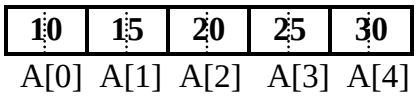
Ex 2: Marks of all students in a class is *an array of marks*

The pictorial representation of an array of 5 integers, an array of 5 floating point numbers and an array of 5 characters is shown below:



2.2 💻 Arrays

Now, let us see "What is a single-dimensional array?" A *single-dimensional* array (also called one-dimensional array) is a *linear list* consisting of related data items of same type. In memory, all the data items are stored in contiguous memory locations one after the other. For example, a single dimensional array consisting of 5 integer elements is shown below:



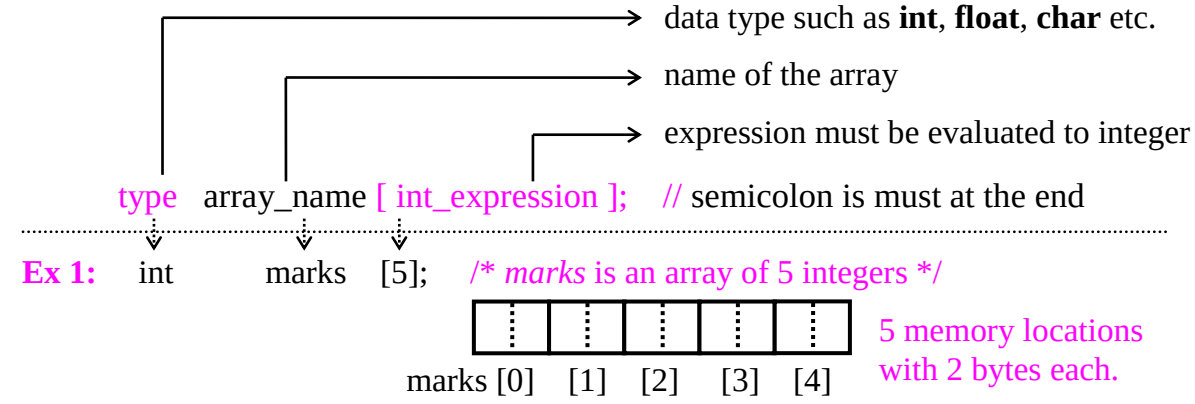
Now, the question is "How to access these elements?" Since an array is identified by a common name, any element in the array can be accessed by specifying the subscript (or an index). For example,

- 0^{th} item 10 can be accessed by specifying A[0]
- 1^{st} item 20 can be accessed by specifying A[1]
- 2^{nd} item 30 can be accessed by specifying A[2]
- 3^{rd} item 40 can be accessed by specifying A[3]
- 4^{th} item 50 can be accessed by specifying A[4]

Now, let us see "How to declare and define a single dimensional array?" As we declare and define variables before they are used in a program, an array also must be declared and defined before it is used. The declaration and definition informs the compiler about the:

- Type of each element of the array
- Name of the array
- Number of elements (i.e., size of the array)

The compiler uses this size to reserve the appropriate number of memory locations so that data can be stored, accessed and manipulated when the program is executed. A single dimensional array can be declared and defined using the following syntax:



2.2 Representation of Linear arrays in memory

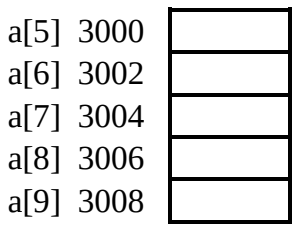
Consider the following declaration:

C Syntax	Pascal Syntax	Pictorial representation
int a[5];	a : array[0..4] of integer lower bound = 0 upper bound = 4	 a[0] a[1] a[2] a[3] a[4] Number of items = 5 = 4 – 0 + 1 = ub – lb + 1 Where *ub* is upper bound = 4 *lb* is lower bound = 0
not possible	a : array[5..9] of integer	 a[5] a[6] a[7] a[8] a[9] Number of items = ub – lb + 1 = 9 - 5 + 1 = 5

Observe that in C language, array index always starts from 0 whereas in Pascal language, array index can start from any integer (even negative indexing is possible in Pascal). Now, let us see "How to obtain the location of a[j] in a single dimensional array?" Let us take the following array declaration in Pascal language:

```
a : array[5..9] of integer;      // Here, lower bound : LB = 5
                                 //       upper bound : UB = 9
```

The storage representation for the above single dimensional matrix assuming base address as 3000 can be written as shown in figure on the right hand side.



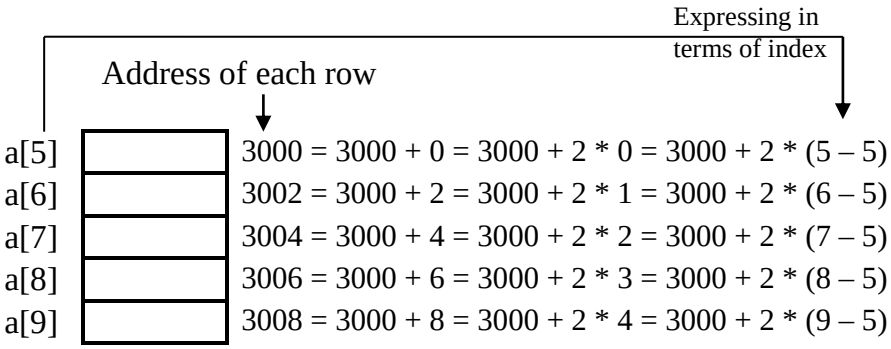
The location of a[j] is given by:

$$\text{Loc A[j]} = \text{base address} + x \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots(1)$$

where

- x is the distance from base address upto jth row

Calculation of x: Given the base address, the distance from base address can be obtained as shown below:



So, distance from base address 3000 = 2 * (j – 5)

$$x = w * (j - lb)$$

where

- 2 represent size of integer i.e., w in general
- 5 represent the lower bound *lb* of the array

Substituting *x* in relation (1) we get:

$$Loc\ A[j] = base\ address(a) + w * (j - lb)$$

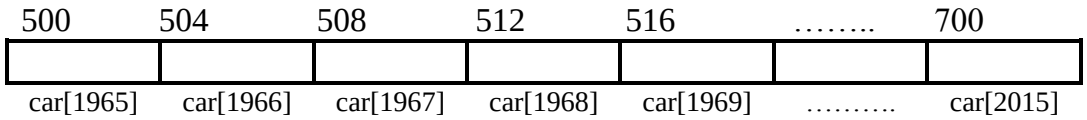
where

- w is the word length. w = 2 for integer values
  w = 4 for floating point values
  w = 8 for double values
- j is the index of the array
- lb is the lower bound of the array

Note: Observe the following points:

- Given any element a[j], its address can be calculated using the above relation and the time taken to calculate location is same. So, the time taken to access a[5], a[6], a[7], a[8] and a[9] remains same.
- The time taken to locate any element a[j] is independent of *j*. That is, irrespective of *j*, the time taken to locate an array element a[j] remains same.
- This is very important property of linear arrays. (Linked lists discussed in chapters 8 and 9 do not have this property.

Example 2.1: A car manufacturing company uses an array *car* to record number of cars sold each year starting from 1965 to 2015. Rather than beginning the array index from 0 or 1, it is more useful to begin the array index from 1965 as shown below:



(a) Find the total number of years
(b) Suppose base address = 500, word length w = 4, find address of car[1967], car[1969] and car[2015]

Solution:

(a) It is given that lower bound = *lb* = 1965
Upper bound = *ub* = 2015
So, total number of years = *ub* – *lb* + 1
= 2015 – 1965 + 1
= 51 years

(b) We know that Loc(a[j]) = Base(a) + w *(j – *lb*)
Loc(car[1967]) = 500 + 4 * (1967 – 1965] = 508
Loc(car[1969]) = 500 + 4 * (1969 – 1965] = 516
Loc(car[2015]) = 500 + 4 * (2015 – 1965] = 700

Example 2.2: Consider the linear arrays AAA(5:50), BBB(-5:10) and CCC(1:18)
(a) Find the number of elements in each array
(b) Suppose Base(AAA) = 300 and w = 4 words per memory cell for AAA. Find the address of AAA[15], AAA[35] and AAA[55]

Solution:

(a) The number of elements in each array can be calculated using the following relation:
Number of elements = ub – lb + 1
So, number of elements of array AAA = 50 – 5 + 1 = 46
Number of elements of array BBB = 10 – (-5) + 1 = 16
Number of elements of array CCC = 18 – 1 + 1 = 18

(b) It is given that Base(AAA) = 300, w = 4, *lb* = 5
We know that Loc(a[i]) = Base(a) + w *(i – *lb*)
Loc(AAA[15]) = 300 + 4 * (15 – 5) = 340
Loc(AAA[35]) = 300 + 4 * (35 – 5) = 420
Loc(AAA[55]) cannot be computed since 55 exceeds ub = 50

2.3 Operations on arrays

Now, let us see "What are the various operations that can be performed on arrays?"

The various operations that can be performed on arrays are shown below:

- Traversing
- Inserting
- Deleting
- searching
- sorting

2.3.1 Traversing

Now, let us see "What is traversing an array?" Visiting or accessing each item in the array is called traversing the array. Here, each element is accessed in linear order either from left to right or from right to left.

In this section, let us see "How to read the data from the keyboard and how to display data items stored in the array?"

We can easily read, write or process the array items using appropriate programming constructs such as for-loop, while-loop, do-while, if-statement, switch-statement etc. Consider the declaration shown below:

```
int   a[5];
```

Here, memory for 5 integers is reserved and each item in the array can be accessed by specifying the index as shown below:

Using a[0] through a[4] we can access 5 integers.

Note: *In general, Using a[0] through a[n-1] we can access n data items.*

Once we know how to access each location in the memory, next question is "*How to store the data items in these locations which are read from the keyboard*?"

This is achieved by reading *n* data items from the keyboard using *scanf()* function which is available in C library as shown below:

```
scanf("%d",  &a[0]);
scanf("%d",  &a[1]);
scanf("%d",  &a[2]);
……
……
scanf("%d",&a[n-1]);
```

In general, **scanf("%d",&a[i])** where i = 0, 1, 2, 3,……n-1.

So, in C language, if we want to read *n* data items from the keyboard, the following statement can be used:

```
for (i = 0; i <= n-1; i++)
{
        scanf("%d", &a[i]);
}
```

or

```
for (i = 0; i < n; i++)
{
        scanf("%d", &a[i]);
}
```

Similarly to display *n* data items stored in the array, replace *scanf()* by *printf()* statement as shown below:

```
for (i = 0; i < n; i++)
{
        printf("%d", a[i]);
}
```

Now, the function to create an array by reading *n* elements can be written as shown below:

Example 2.3: Functions to read *n* items into an array

```
void create_array(int a[], int n)
{
        int     i;

        for (i = 0; i < n; i++)
        {
                scanf("%d", &a[i]);
        }
}
```

i = 0	1	2	3	4
a[0]	a[1]	a[2]	a[3]	a[4]
10	20	30	40	50

Now, the function to display n elements of the array can be written as shown below:

Example 2.4: Functions to display n elements of array

```
void display_array(int a[], int n)
{
        int     i;

        for (i = 0; i < n; i++)                 i = 0    1    2    3    4
        {                                         a[0] a[1] a[2] a[3] a[4]
                printf("%d   ", a[i]);           10   20   30   40   50
        }
}
```

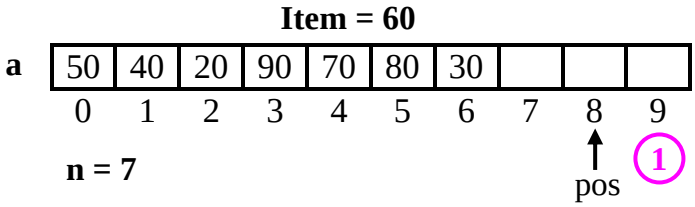
2.3.2 Inserting an item into an array (based on the position)

Now, let us see "How to insert an item into an unsorted array based on the position?"

Problem statement: Given an array *a* consisting of *n* elements, it is required to insert an *item* at the specified position say *pos*.

Design: An item can be inserted into the array by considering various situations as shown below:

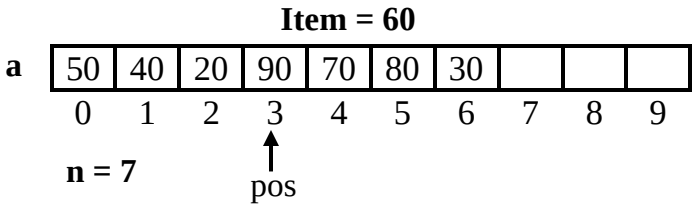
Step 1: Elements are present (Invalid position): This case can be pictorially represented as shown below:



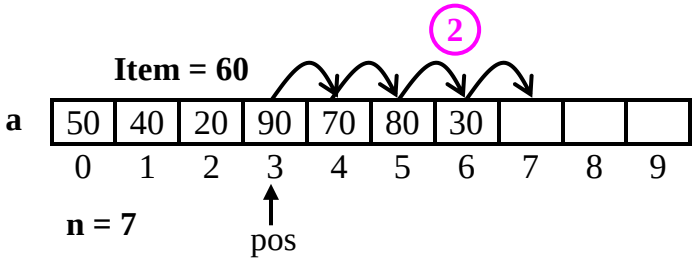
Can you insert an item at 8th position onwards in the above array? No, we cannot insert since, it is invalid position. That is, if *pos* is greater than 7 or if *pos* is less than 0, the position is invalid. The code for this case can be written as shown below:

```
        if (pos > n || pos < 0)              // When pos is greater than number of items
        {
1               printf ("Invalid position\n");
                return n;                    // No insertion and return number of items
        }
```

Step 2: Make room for the item to be inserted at the specified position: Consider the following list with 7 elements and item 60 to be inserted at position 3.



We have to make room for the item to be inserted at position 3. This can be done by moving all the elements 30, 80, 70 and 90 from positions 6, 5, 4, 3 into new positons 7, 6, 5, 4 respectively towards right by one position as shown below:



This is possible using the following assignment statements in the order specified:

```
a[7]  = a[6];
a[6]  = a[5];
a[5]  = a[4];
a[4]  = a[3]
```

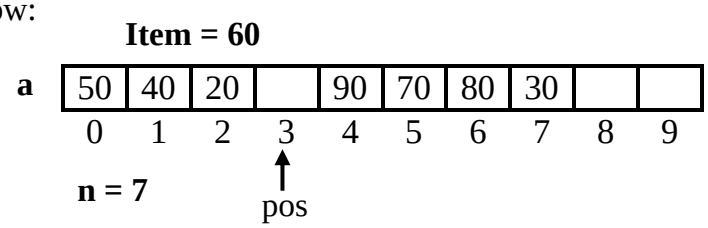
In general, a[i+1] = a[i] for i = 6 down to 3
for i = n-1 down to *pos*

Now, the code for above activity can be written as shown below:

```
for (i = n-1; i >= pos; i--)
{
        a[i+1] = a[i];
}
```

2

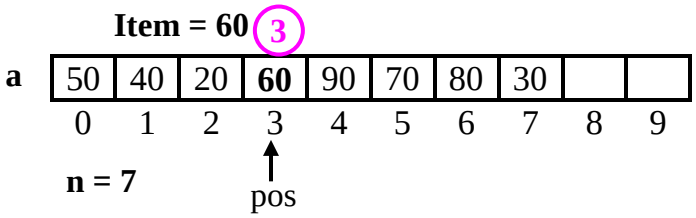
After executing, the above statement, the array contents can be pictorially represented as shown below:



Step 3: Insert item at the specified position: The code for this case can be written as shown below:

```
(3)  a[pos] = item;
```

Now, the contents of array can be written as shown below:



Thus, item 60 is inserted at position 3.

Step 4: Update number of elements in above array: The code for this case can be written as shown below:

```
(4)  return n + 1;
```

Now, the complete function to insert an item at the specified position can be written as shown below:

Example 2.5: Function to insert an item at the specified position in the array

```
int insert_at_pos (int item, int a[], int n, int pos)
{
        int     i;

        if (pos > n || pos < 0)                 // When pos is greater than number of items
        {
(1)             printf ("Invalid position\n");
                return n;                       // No insertion and return number of items
        }

        for (i = n-1; i >= pos; i--)            /* Create space for item to insert  at pos*/
        {
(2)             a[i+1] = a[i];
        }

(3)     a[pos] = item;                          /* Insert item at the specified position */

(4)     return n + 1;                           /* Update number of items in the array */
}
```

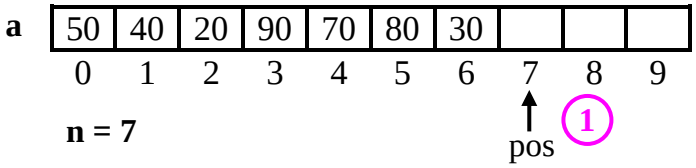
2.3.3 Deleting an item from an array (based on the position)

Now, let us see **"How to delete an item from an unsorted array based on the position?"**

Problem statement: Given an array *a* consisting of *n* elements, it is required to delete an *item* at the specified position say *pos*.

Design: An item can be deleted from the array by considering various situations as shown below:

Step 1: Elements are present (Invalid position): This case can be pictorially represented as shown below:

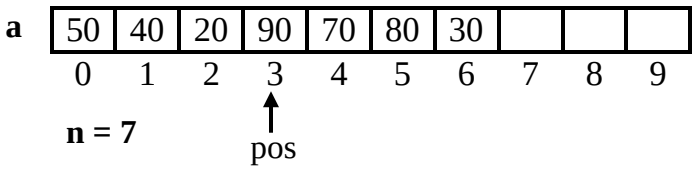


Can you delete an item from 7th position onwards in the above array? No, we cannot delete since, it is invalid position. That is, if *pos* is greater than or equal to 7 or if *pos* is less than 0, the position is invalid. The code for this case can be written as shown below:

```
if (pos >= n || pos < 0 )                    // When pos >= number of items
{
        printf ("Invalid position\n");
        return n;                            // No deletion and return number of items
}
```

(1)

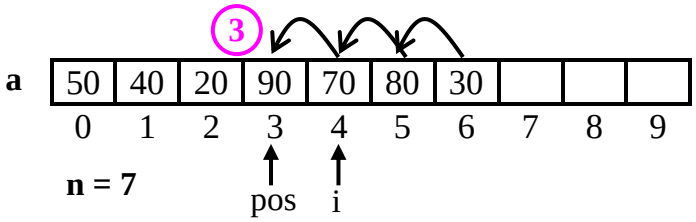
Step 2: Display the item to be deleted: Consider the following list with 7 elements and let the position *pos* is 3.



The item at position *pos* can be accessed by writing a[pos] and it can be displayed using the printf() function as as shown below:

(2) printf("Item deleted = %d\n", a[pos]);

Step 3: Remove the item from the array: Removing an element at the given position can be illustrated using the following figure:



As shown in above figure, move all the elements from 4th position onwards towards left by one position using the following statements:

```
a[3]  = a[4];
a[4]  = a[5];
a[5]  = a[6];
```

(3)

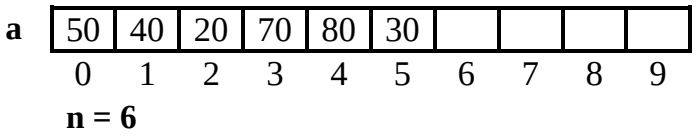
In general, a[i - 1] = a[i] for i = 4 to 6
for i = pos+1 to n-1

Now, the code for above activity can be written as shown below:

```
for (i = pos + 1;  i < n; i++)
{
        a[i - 1] = a[i];
}
```

(3)

After executing the above statement, the array contents can be pictorially represented as shown below:



Step 4: Update number of elements in above array: After deleting an item, the number of items in the array should be decremented by 1. It can be done using the following statement:

```
return n - 1;
```

(4)

Now, the complete function to delete an item from the specified position can be written as shown below:

Example 2.6: Function to delete an item from the specified position in the array

```
int delete_at_pos (int a[], int n, int pos)
{
        int     i;

        if (pos >= n || pos < 0)                // When pos >= number of items
        {
(1)             printf ("Invalid position\n");
                return n;                       // No deletion and return number of items
        }
(2)     printf("Item deleted = %d\n", a[pos]);

        for (i = pos + 1;  i < n; i++)          // Move elements towards left
        {
(3)             a[i - 1] = a[i];
        }
(4)     return n - 1;                           // Decrement the number of items in array
}
```

Example 2.7: Design, Develop and Implement a menu driven Program in C for the following **Array** operations

1. Creating an Array of **N** Integer Elements
2. Display of Array Elements with Suitable Headings
3. Inserting an Element (**ELEM**) at a given valid Position (**POS**)
4. Deleting an Element at a given valid Position(**POS**)
5. Exit.

Support the program with functions for each of the above operations.

```
#include <stdio.h>
#include <process.h>

/* Insert: Example 2.3: function to create an array of N elements */

/* Insert: Example 2.4: function display an array of N elements */

/* Insert: Example 2.5: function to insert an item at a given position */

/* Insert: Example 2.6: function to delete an item at a given position */

void main()
{
        int     choice, a[10], n, item, pos;
```

```
        for (;;)
        {
                printf("1:Creat an array       2: Display \n");
                printf("3:Insert at position   4: Delete at position \n");
                printf("5:Exit\n");
                printf("Enter the choice\n");

                scanf("%d", &choice);

                switch(choice)
                {
                        case 1:
                                printf("Enter the number of elements\n");
                                scanf("%d", &n);
                                printf("Enter %d elements\n", n);
                                create_array(a, n);
                                break;

                        case 2:
                                printf("The contents of the array are\n");
                                display_array(a, n);
                                break;

                        case 3:printf("Enter the item to be inserted : ");
                                scanf("%d", &item);
                                printf("Enter the position : ");
                                scanf("%d", &pos);
                                n = insert_at_pos (item, a, n, pos);
                                break;

                        case 4:printf("Enter the position : ");
                                scanf("%d", &pos);
                                n = delete_at_pos (a, n, pos);
                                break;
                        default:
                                exit(0);
                }
        }
}
```

2.3.4 Sorting array elements

First, we shall see "*What is sorting?*"

Definition: More often programmers will be working with large amount of data and it may be necessary to arrange them in ascending or descending order. This process of arranging the given elements so that they are in ascending order or descending order is called *sorting*. For example, consider the unsorted elements:

10, 50, 25, 20, 15

After sorting them in ascending order, we get the following list:

10, 15, 20, 25, 50

After sorting them in descending order, we get the following list:

50, 25, 20, 15, 10

Before writing the algorithm or the program let us know the answer for "*What is the concept used in bubble sort (Why it is also called sinking sort)?*"

Design: Once we know what is sorting, the next question is "*How to sort the elements using bubble sort?"*

Step 1: Identify parameters to function: Given an array *a* consisting of *n* elements we have to sort them in ascending or descending order. So,

parameters are : **int a[], int n**

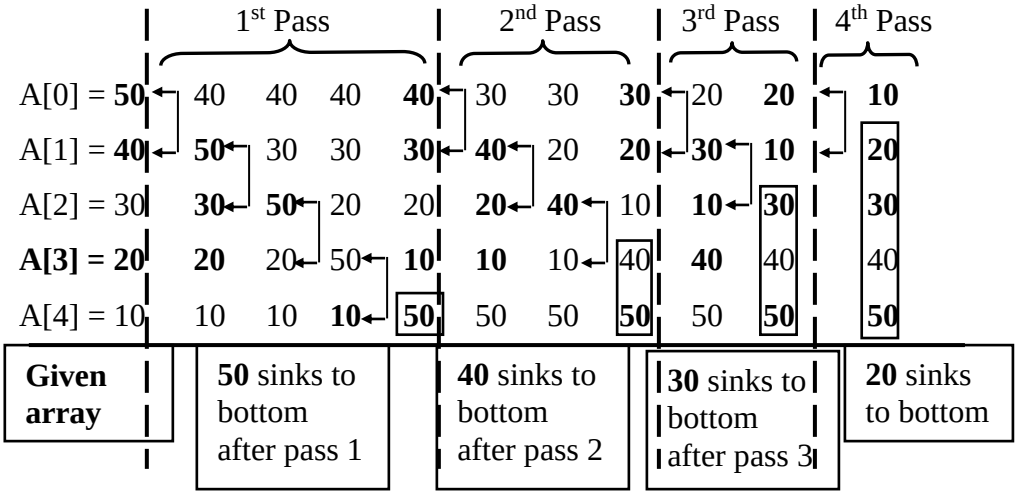
Step 2: Return type: Our intention is only to sort the numbers. Hence, we are not returning any value. So,

return type : **void**

Step 3: Designing body of the function: This is the simplest and easiest sorting technique. In this technique, the two successive items A[i] and A[i+1] are exchanged whenever A[i] >= A[i+1]. For example, consider the elements shown below:

50, 40, 30, 20, 10

The elements can be sorted as shown below:



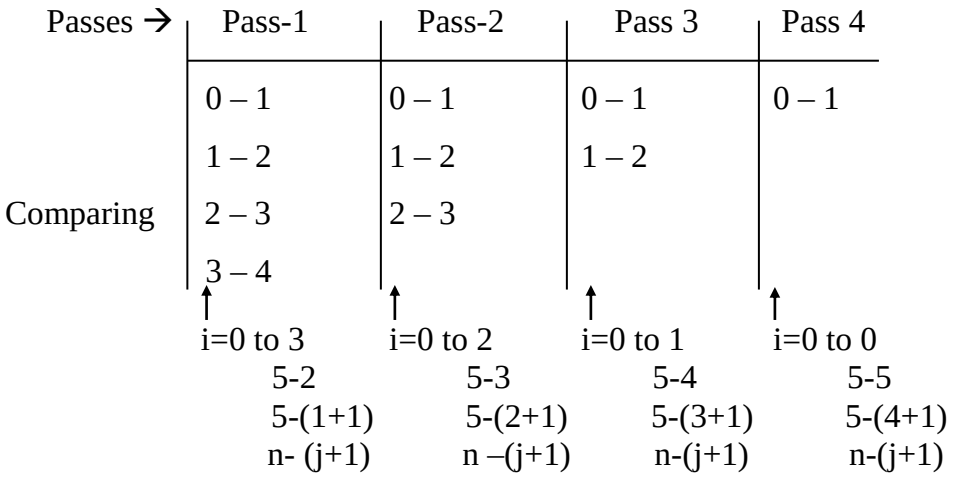
- In the first pass 50 is compared with 40 and they are exchanged since 50 is greater than 40.
- Next 50 is compared with 30 and they are exchanged since 50 is greater than 30. If we proceed in the same manner, at the end of the first pass the largest item occupies the last position.
- On each successive pass, the items with the next largest value will be moved to the bottom and thus elements are arranged in ascending order.

Note: Observe that after each pass, the larger values sinks to the bottom of the array and hence it is called *sinking sort*. The following figure below shows the output of each pass.

Original Array	Out put of each pass 1st	2nd	3rd	4th
A[0] = 50	40	30	20	10
A[1] = 40	30	20	10	20
A[2] = 30	20	10	30	30
A[3] = 20	**10**	**40**	**40**	**40**
A[4] = 10	50	50	50	50

Note: Observe that at the end of each pass smaller values gradually "bubble" their way upward to the top (like air bubbles moving to surface of water) and hence called ***bubble sort***.

Now we concentrate on the designing aspects of this sorting technique. The comparisons that are performed in each pass are shown below:



- In general we can say i = 0 to n-(j+1) or i = 0 to n – j – 1.
- Here, j = 1 to 4 represent pass number. In general j = 1 to n-1.
- So, the partial code can be written as:

```
for  j = 1 to n-1
        for i = 0 to n-j-1
                if  ( A[i] > A[i +1] )
                        exchange ( A[i] , A[i+1] )
                end if
        end if
end if
```

Now, the complete bubble sort algorithm can be written as shown below:

Example 2.8: Algorithm for bubble sort

```
Algorithm BubbleSort(a[], n)
        for j ← 1 to n – 1 do                   // Perform n-1 passes
                for i ← 0 to n-j-1do            // To compare items in each pass
                        if ( a[i] > a[i+1] )    // If out of order exchange
                                temp ← a[i]
                                a[i] ← a[i+1]
                                a[i+1] ←temp
                        end if
                end for
        end for
```

Note: All variables other than parameters should be declared as local variables i.e., the variables *i*, *j* and *temp* should be declared as **int** data type in function.

The C equivalent for the above algorithm can be written as shown below:

Example 2.9: C function to sort the array elements in ascending order

```
void bubble_sort ( int   a[],  int  n )
{
        int     j;                          /* Represent pass number */
        int     i;                          /* To access elements in each pass */
        int     temp;                       /* Temporary variable used to exchange*/
        for ( j = 1; j < n;  j++)           /* Number of passes required: n-1*/
        {
                for ( i = 0; i < n - j; i++)    /* To access elements in each pass */
                {
                        if ( a[i] >= a[i+1] )   /* Exchange if out of order */
                        {
                            temp = a[i];
                            a[i] = a[i+1];
                            a[i+1] = temp;
                        }
                }
         }
}
```

The C program that uses the above function to sort the elements is shown below:

Example 2.10: C program to read *n* numbers, sort them using bubble sort

```
#include <stdio.h>

/* Include: Example  2.9:  bubble sort  */

void main()
{
        int     a[20];              /* Array of integers to be sorted */
        int     n;                  /* Number of elements in array a and b  */
        int     i;                  /* Index used to access elements in a and b */
```

	Input
printf("Enter the number of items"); **scanf**("%d",&n);	Enter the number of items 5

printf("Enter the items to sort \n");
for (i = 0; i < n; i++)
{
scanf("%d",&a[i]);
}
	Enter the items to search
	i = 0 1 2 3 4
	a[0] [1] [2] [3] [4]
	40 50 30 10 20

bubble_sort (a, n);	10 20 30 40 50 a[0] [1] [2] [3] [4]
printf("The sorted elements are"); **for** (i = 0; i < n; i++) **{** **printf**("%d\n",a[i]); **}**	The sorted elements are 10 20 30 40 50

}

Advantages of bubble sort

- Very simple and easy to program
- Straight forward approach

Disadvantages of bubble sort

- It runs slowly and hence it is not efficient. More efficient sorting techniques are present
- Even if the elements are sorted, n-1 passes are required to sort.

2.3.5 Searching an element in an array

First, let us see *"What is searching? What are the various searching techniques?"*

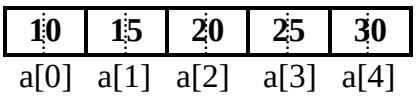
Definition: More often we will be working with large amount of data. It may be necessary to determine whether a particular item is present in the large amount of data. *This process of finding a particular item in the large amount of data is called searching*. The two important and simple searching techniques are shown below:

- Linear search
- Binary search

2.3.5.1 Linear search (Sequential search)

Now, let us see "*What is linear search?*"

Definition: A *linear search* also called *sequential search* is a simple searching technique. In this technique, we search for a given *key* in the list in linear order (sequential order) i.e., one after the other from first element to last element or vice versa. The search may be successful or unsuccessful. If key is present, we say search is successful, otherwise, search is unsuccessful. For example, consider the following array:



- Successful search: If search key is 25, it is present in the above list and we return its position 3 indicating "Successful search".

- Unsuccessful search: If search key is 50, it is not present in the above list and we return -1 indicating "Unsuccessful search"

Design: Now, let us see how to search for an item.

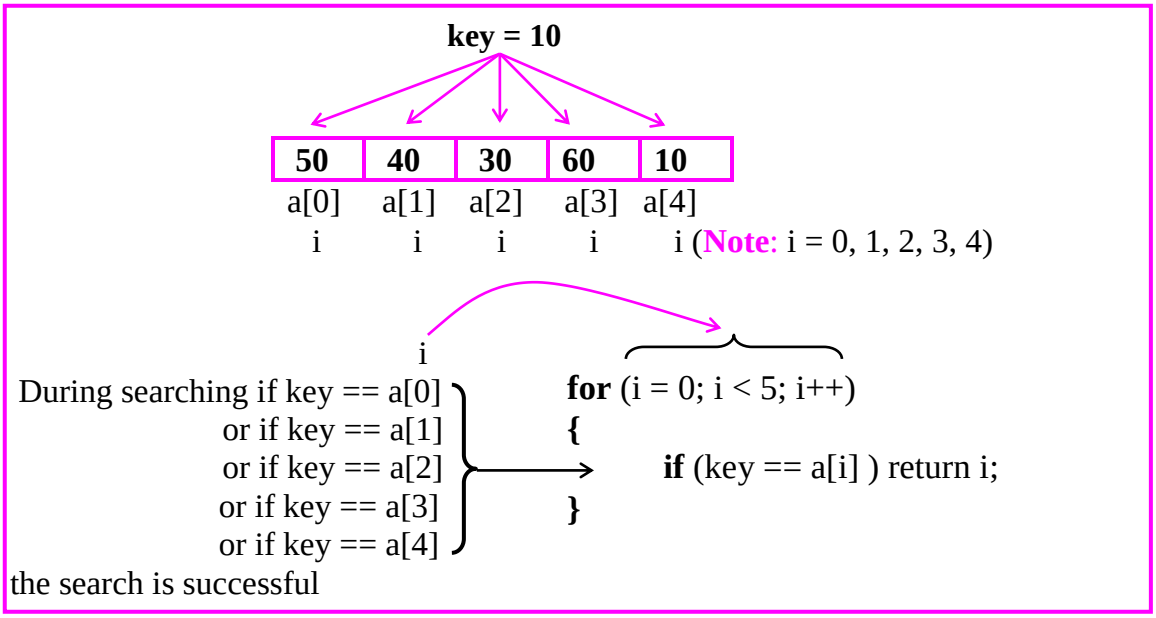
Step 1: Identify parameters to function: We have to search for *key* item in an array *a* consisting of *n* elements. So, input must be *key*, *array a* and n. So,

parameters are : **int** key, **int a[], int n**

Step 2: Return type: We are returning position of *key* item if found, otherwise, we return -1. Note that the position is integer and -1 is also integer. So,

return type : **int**

Step 3: Designing function body: Let a[0], a[1],…..etc. be an array of *n* elements. Let us take an example. Assume 10 is the *key* item to be searched in the list shown in figure. The list consists of items 50, 40, 30, 60 and 10 in order. In the worst scenario, *key* item may have to be compared with all the elements in the array. Observe from following figure that, *key* item 10 has to be compared with a[0], a[1], a[2], a[3] and a[4] as shown below:



But, once the value of *i* is greater than or equal to 5, it is an indication that *item* is not present and we return -1. The code for this can be written as:

```
return -1;        /*Unsuccessful search  */
```

Note: The terminal condition in the *for* loop i.e., i < 5 can be replaced by i < n for a general case. Now, the code can be written as:

```
for (i = 0; i < n; i++)
{
        if (key == a[i] ) return i;      /* Successful search */
}

return – 1;     /* Unsuccessful search */
```

Note: All variables other than parameters should be declared as local variables i.e., the variable *i* which is of type **int** must be declared inside the function definition.

The C function to search for a *key* item is shown below:

Example 2.11: C function to implement linear search.

```
int   linear ( int   key,  int  a[],  int n)
{
        int   i;
        .....
        for (i = 0; i < n; i++)
        {
                if (key == a[i] ) return i;      /* Successful search */
        }
        .....
        return – 1;     /* Unsuccessful search */
}
```

The C function that calls above function is shown below:

Example 2.12: C Program to implement linear search

```
#include <stdio.h>
/* Include: Example 2.11:  Function to implement linear search  */

void main()
{
        int    n, key, a[20], i, pos;
        .....
        printf("Enter the value of n\n");
        scanf("%d",&n);
        .....
        printf("Enter n values\n");
        for ( i = 0; i < n; i++) scanf("%d",&a[i]);
        .....
        printf("Enter the item to be searched\n");
        scanf("%d",&key);
        .....
        /* Search for an element */
        pos = linear (key, a , n);
        .....
        if (pos == -1)
                printf("Item not found\n");
        else
                printf("Item found\n");
}
```

TRACING1	TRACING2
Enter value of n 5	Enter value of n 5
Enter n values 10 20 50 40 30	Enter n values 10 20 50 40 20
Enter item to search 30	Enter item to srch 60
Output pos = 4	**Output** pos = -1
	Item not found
Item found	

Advantages of linear search

- Very simple approach
- Works well for small arrays
- Used to search when the elements are not sorted

Disadvantages of linear search

- Less efficient if the array size is large
- If the elements are already sorted, linear search is not efficient.

2.3.5.2 Binary search

To overcome the disadvantages of *linear search*, we use *binary search*. Now, let us see "*What is binary search? What is the concept used in binary search?*"

Definition: A *binary search* is a simple and very efficient searching technique which can be applied if the items to be compared are either in ascending order or descending order.

The general idea used in binary search is similar to the way we search for the telephone number of a person in the telephone directory. Obviously, we do not use *linear search*. Instead, we open the book from the middle and the *name* is compared with the element at the middle of the book. If the name is found, the corresponding telephone number is retrieved and the searching has to be stopped. If the *name* to be searched is less than the middle element, search towards left otherwise, search towards right. The procedure is repeated till *key* item is found or the *key* item is not found.

Design: Once we know the concept of *binary search*, the next question is "*How to search for key in a list of elements?*"

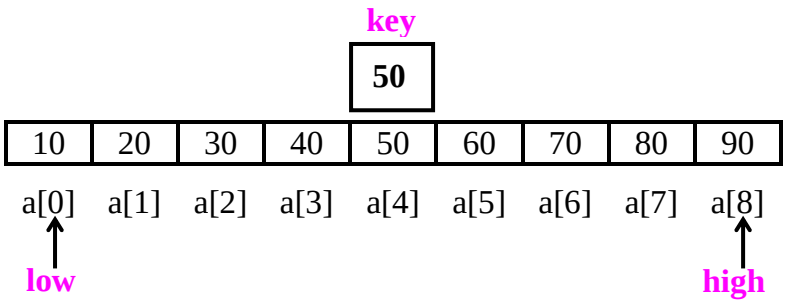
Step 1: Identify parameters to function: We have to search for *key* item in an array *a* consisting of *n* elements. So, input must be *key*, *array a* and n. So,

parameters are : **int** key, **int a[], int n**

Step 2: Return type: We are returning position of *key* item if found, otherwise, we return -1. Note that the position is integer and -1 is also integer. So,

return type : **int**

Step 3: Designing function body: Let *key* is the element to be searched in the array *a* consisting of *n* elements. In the array *a*, element 10 in position 0 is the first element and element 90 in position 8 is the last element as shown below:



So, initial values are: low = 0 and high = 8
= 9 – 1
= n – 1 (general)

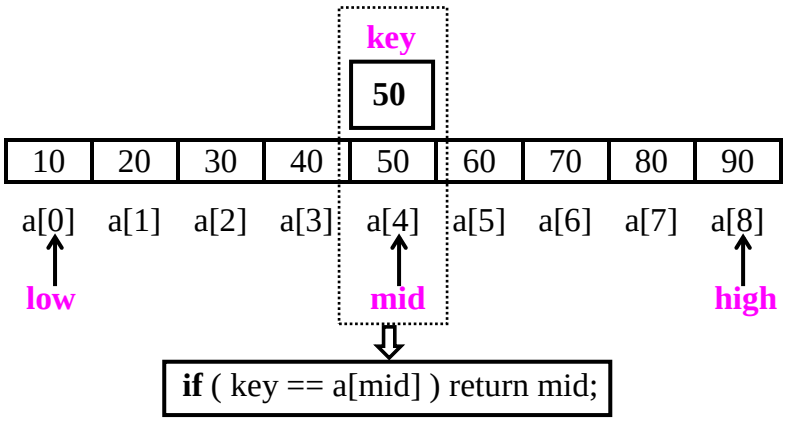
So, in general, initial values are:

```
low  = 0
high = n – 1
```

Step 4: If *low* is the position of the first element and *high* is the position of the last element, the position of the middle element can be obtained using the statement:

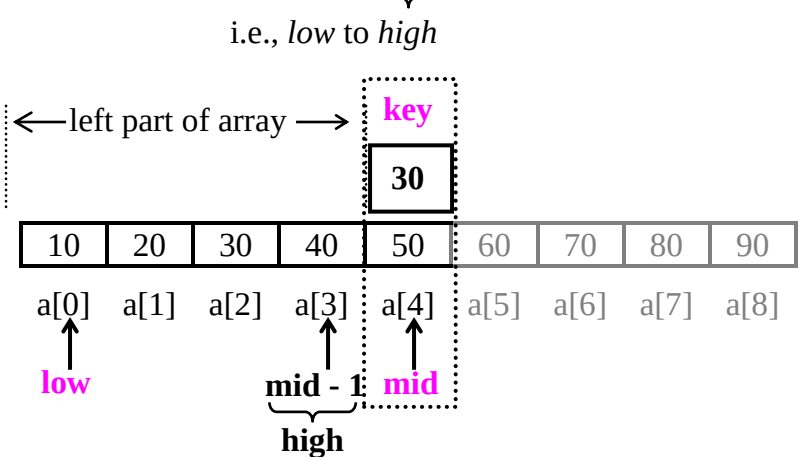
```
mid = ( low + high ) /2
```

The *key* to be searched is compared with middle element. The pictorial representation and equivalent code can be written as shown below:



After executing the statement, if *key* is same as *a[mid]*, we return the position of *key*. If *key* is not same as *a[mid]*, it may be present either in the left part of the array or right part of the array and leads to following 2 cases:

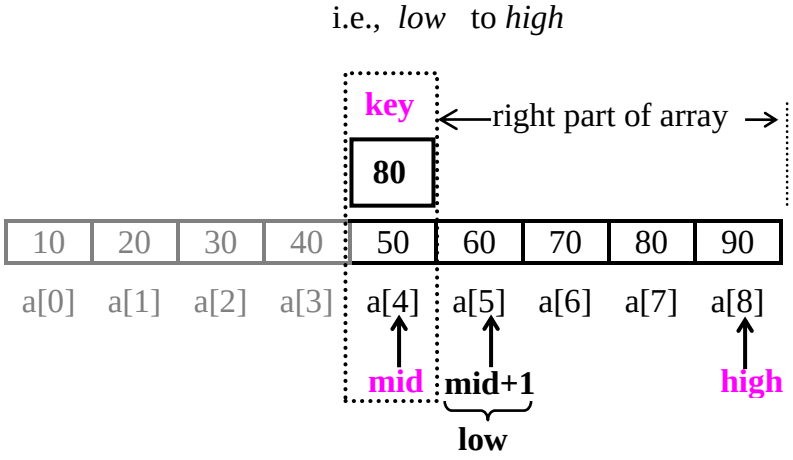
Case 1: *key* towards left of *mid*: If *key* is less than the middle element, the left part of array has to be compared from *low* to *mid*-1 as shown in figure below:



Note that position of *low* has not been changed. But, position of *high* is changed to mid–1. The equivalent code can be written as:

```
if ( key < a[mid] ) high = mid – 1;
```

Case 2: *key* towards right of *mid*: If *key* is greater than the middle element, the right part of array has to be compared from *mid* + 1 to *high* as shown in figure below:



Note that position of *high* has not been changed. But, position of *low* is changed to mid+1. The equivalent code can be written as:

```
if ( key > a[mid] ) low = mid + 1;
```

Note that entire step 4 has to be repeatedly executed as long as long as *low* is less than or equal to *high*. The code for this can be written as shown below:

```
while ( low <= high )
{
        mid = ( low + high ) / 2;              /* Find the mid point */

        if ( key == a[mid] ) return mid;       /* Item found */

        if ( key < a[mid] )  high = mid – 1;   /* To search left part */

        if ( key > a[mid] ) low = mid + 1;     /* To search right part */
}
```

Finally, when the value of *low* exceeds the value of *high* indicates that *key* is not present in the array and return -1 using the following statement:

```
return -1;              /* Search unsuccessful */
```

The complete C function can be written as shown below:

Example 2.13: C function returning position of item found. Otherwise return -1

```
int binary_search ( int key, int a[], int n)
{
        int     low,  high,  mid;

        low = 0;                                /* Initialization */
        high = n-1;

        while ( low <= high )
        {
                mid = ( low + high ) / 2;              /* Find the mid point */

                if ( key == a[mid] ) return mid;       /* Item found */

                if ( key < a[mid] ) high = mid – 1;    /* To search left part */

                if ( key > a[mid] ) low = mid + 1;     /* To search right part */
        }

        return -1;      /* Unsuccessful search */
}
```

The C program that uses the above function is shown below:

Example 2.14: C program search for an item using binary search

```
#include <stdio.h>

/* Include: Example 2.13: Function binary search */
void main()
{
        int   n,  a[10];          /* Variables associated with array     */
        int   key;                /* Element to be searched              */
        int   position;           /* Contains position of key if found:
                                     -1 if not found                     */

        printf ("Enter the number of elements\n");
        scanf ("%d", &n);
```

Enter no. of items
5

```
        printf ("Input %d elements\n", n);
        for (i = 0; i < n;  i++) scanf("%d", &a[i]);
```

Elements in the list

10	20	30	**40**	50	
a [0]	[1]	[2]	[3]	[4]	

```
        printf ("Enter the element to be searched\n");
        scanf ("%d", &key);
```

Input1	**Input 2**
55	**40**

```
        pos = binary_search (key, a, n);
```

pos = -1 pos = 3

```
        if (pos == -1)
                printf ("Item not found \n");
        else
                printf ("Item found at pos: %d\n", position);
}
```

Output1
Item not found

Output2
Item found at pos: 3

Advantages of binary search

- Simple technique
- Very efficient searching technique

Disadvantages of binary search

- The list of elements to be searched should be sorted.

- It is necessary to obtain the middle element which is possible only if the elements are stored in the array. If the elements are stored in linked list, this method cannot be used.

2.4 Multidimensional arrays

So far we have discussed single dimensional arrays where each element in the array is referenced by a single subscript. Most programming languages allow two and three dimensional arrays where array elements are referenced using two subscripts and three subscripts respectively. In this section, let us concentrate on two-dimensional arrays. The term *dimension* represents number of indices (plural of index) used to access a particular item in an array. Now, the question is *"What are multi-dimensional arrays?"*

Definition: Arrays with one set of square brackets [] are single dimensional arrays. Arrays with two sets of square brackets [][] are called 2-dimensional arrays and so on. Arrays with two or more dimensions are called *multi-dimensional arrays.* It is the responsibility of the programmer to specify the number of elements to be processed within square brackets. For example,

```
int a[10];          /* a is declared as single dimensional array  */
int b[10][10];      /* b is declared as two dimensional array   */
int c[3][4][5];     /* c is declared as three dimensional array */
```

Arrays with more than 2-dimensions are not the scope of this book.

Now, we discuss 2-dimensional arrays. Before proceeding further, let us answer the question *"What are 2-dimensional arrays? When are 2-dimensional arrays used?"*

Definition: Arrays with two sets of square brackets [][] are called 2-dimensional arrays. A two-dimensional array is used when data items are arranged in row-wise and column wise in a tabular fashion. Here, to identify a particular item we have to specify two indices (also called subscripts): the first index identifies the row number and second index identify the column number of the item.

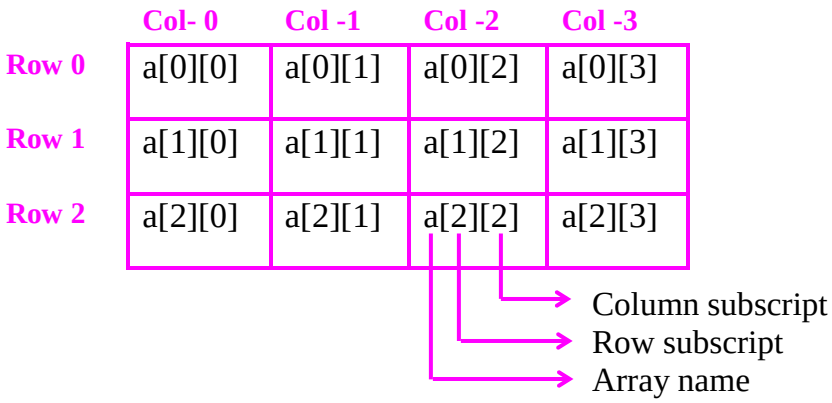
2.4.1 Declaring and initializing two dimensional arrays

For example, consider the following declaration

```
int     a[3][4];
```

The array *a* has two sets of square brackets [][] and hence it is a 2-dimensional array with 3 rows and 4 columns. This declaration informs the compiler to reserve 12

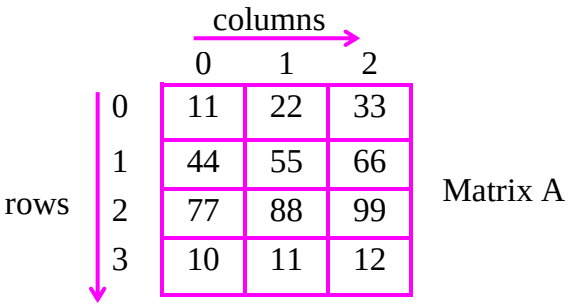
locations (3 * 4 = 12) contiguously one after the other. The pictorial representation of this array *a* is shown below:



Now, let us see "How a 2-dimensional array can be initialized?" Consider the initialization statement shown below:

```
int a[4][3] =  {
                    {11,  22,  33},
                    {44,  55,  66},
                    {77,  88,  99},
                    {10,  11,  12}
               };
```

The declaration indicates that array *a* has 4 rows and 3 columns. The pictorial representation of this 2-dimensional array is shown below:



2.4.2 Storage representation of 2-dimensional arrays

Even though we represent a matrix pictorially as a 2-dimensional array, in memory the elements are stored contiguously one after the other.

2.30 Arrays

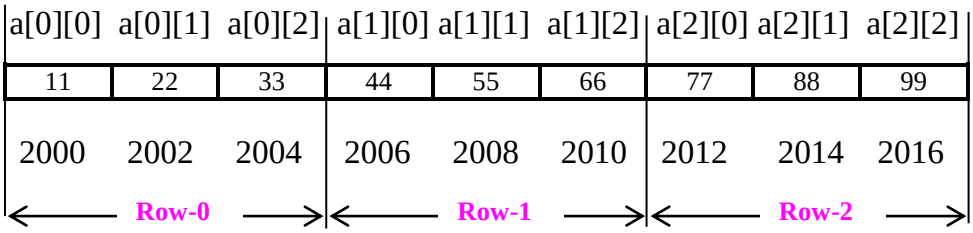
Now, let us see "What are the two different ways in which elements can be stored in a 2-dimensional array?" The elements in a 2-dimensional array can be stored using:

- Row major order
- Column major order

Now, let us see "What is row major order?" In row major order, the elements are stored row by row one row at a time. For example, consider the following 2-dimensional matrix:

```
int a[4][3] =   {
        {11,  22,  33},
        {43,  55,  66},
        {77,  88,  99},
};
```

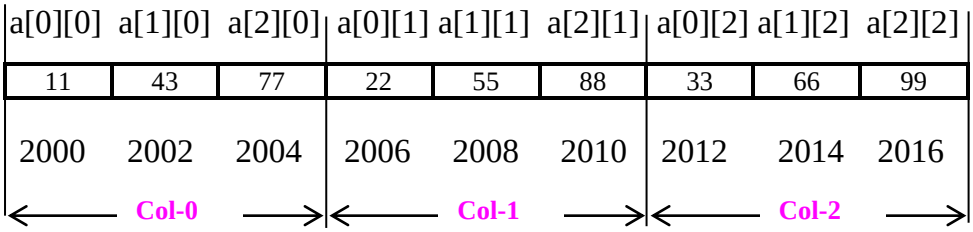
Assume that address of first byte of *a* (usually called base address) is 2000 and size of integer is 2 bytes. The above matrix can be stored in memory using row-major order as shown below:



Now, let us see "What is column major order?" In column major order, the elements are stored column by column one column at a time. For example, consider the following 2-dimensional matrix:

```
int a[4][3] =   {
        {11,  22,  33},
        {43,  55,  66},
        {77,  88,  99},
};
```

Assume that address of first byte of *a* (usually called base address) is 2000 and size of integer is 2 bytes. The above matrix can be stored in memory using column-major order as shown below:



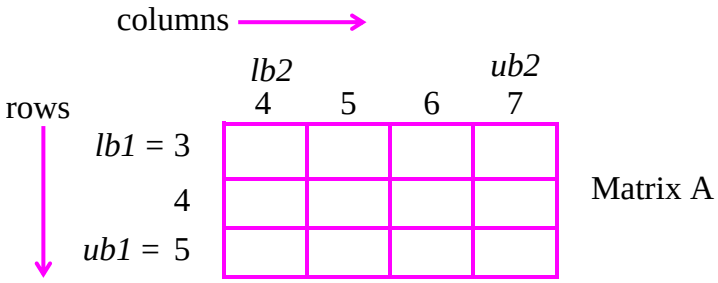
Observe that in C language, array index always starts from 0 whereas in Pascal language, array index can start from any integer (even negative indexing is possible in Pascal).

2.4.2.1 Storage representation of 2-dimensional arrays (Row-major order)

Now, let us see "How to obtain the address of a[i][j] when array elements are stored in row-major order?" Let us take the following array declaration in Pascal language:

```
a : array[3..5, 4..7] of integer;          // size of array is 3 x 4
```

The pictorial representation of above array can be written as shown below:



In the above matrix representation observe that:

- There are three rows ranging from 3 to 5 and so lower bound and upper bound for rows is given by *lb1* = 3 and *ub1*= 5
- There are four columns ranging from 4 to 7 and so lower bound and upper bound for columns is given by *lb2* = 4 and *ub2* = 7

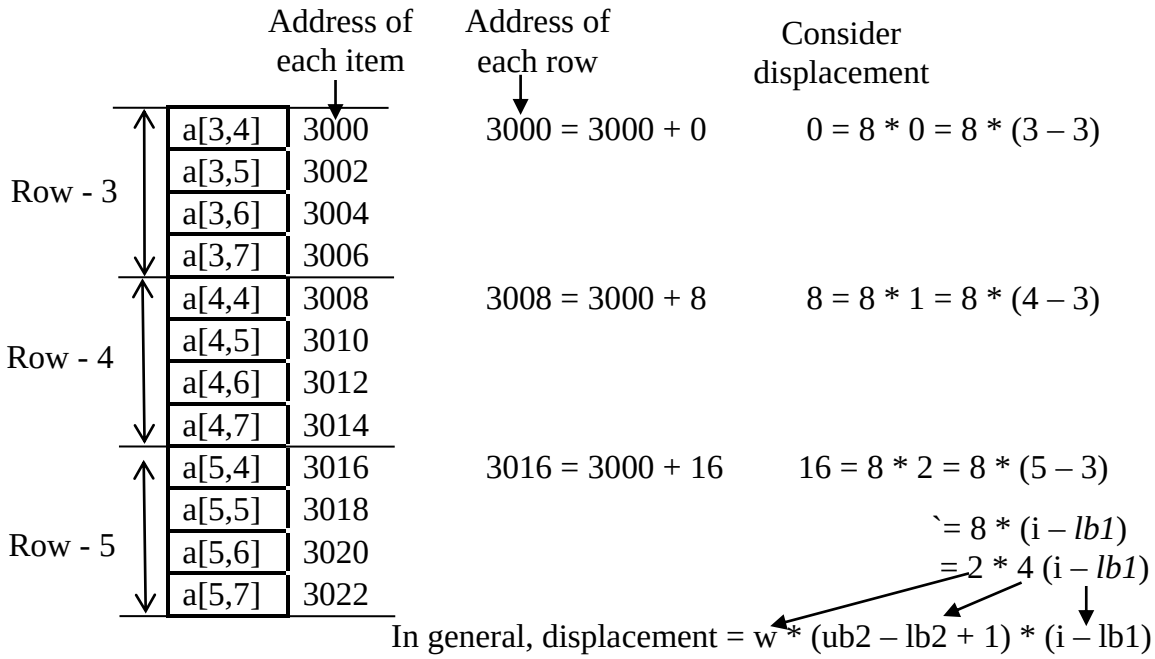
The location of a[i][j] is given by:

Loc A[i][j] = base address + x + y ……(1)

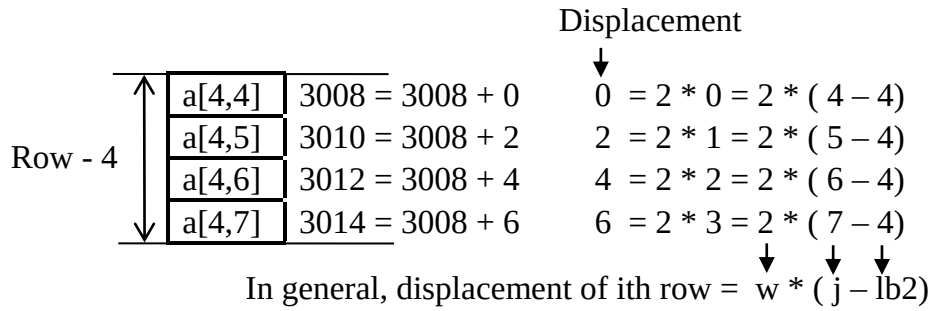
where

- x is the displacement of ith row from base address
- y is the displacement of jth column from ith row

Finding x (displacement of ith row from base address): The row-major order representation for the above matrix assuming base address as 3000 can be written as shown below:



Finding y (displacement of jth column from ith row): Consider any row and corresponding addresses. Let us consider row 4 elements as shown below:



Substituting x and y in relation (1) we get:

The location of a[i][j] is given by:

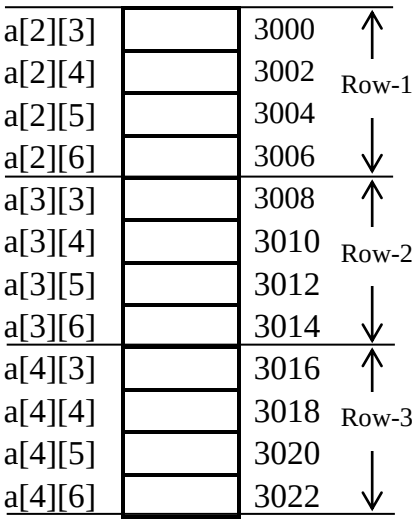
Loc A[i][j] = base address(a) + w * (ub2 – lb2 + 1) * (i – lb1) + w * (j – lb2)

Example 2.15: Consider the array declaration in Pascal language:

a : array[2..4, 3..6] of integer; // size of array is 3 x 4

Find the location a[3][5] when elements are stored in row-major order

The 2-dimensional array elements stored using row major order assuming **base address = 3000** is pictorially represented as shown below:



It is given that:

a : array[2..4,3..6] of integer;

So, *lb1* = 2, *ub1* = 4

lb2 = 3, *ub2* = 6

w = 2 (size of integer)

We need to find location of a[3][5]

So, i = 3, j = 5

Loc A[i][j] = base address + (ub2 – lb2 + 1) * w * (i – *lb1*) + w * (j – lb2)

Loc a[3][5] = 3000 + (6 – 3 + 1) * 2 * (3 – 2) + 2 * (5 – 3)

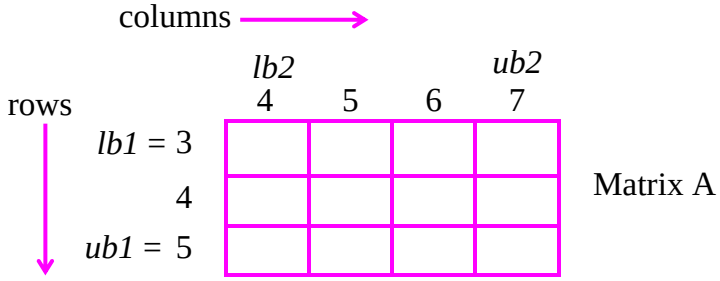
Loc a[3][5] = 3012 (observe from the figure that location of a[3][5] is 3012

2.4.2.2 Storage representation of 2-dimensional arrays (column-major order)

Now, let us see "How to obtain the address of a[i][j] when array elements are stored in column-major order?" Let us take the following array declaration in Pascal language:

a : array[3..5, 4..7] of integer; // size of array is 3 x 4

The pictorial representation of above array can be written as shown below:



The location of a[i][j] is given by:

Loc A[i][j] = base address + x + y ……(1)

Finding x (displacement of jth column from base address): The column-major order representation for the above matrix assuming base address as 3000 can be written as shown below:

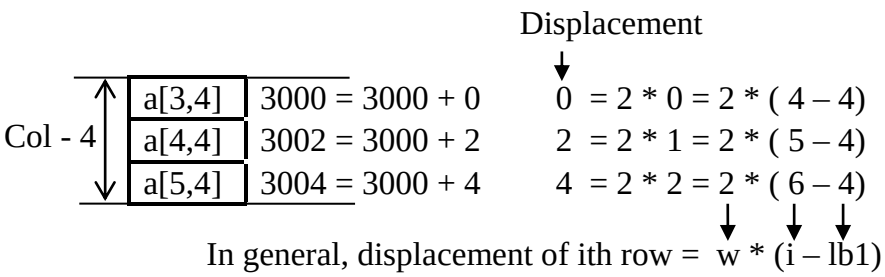
		Address of each item	Address of each column	Consider displacement
Col - 4	a[3,4]	3000	3000 = 3000 + 0	0 = 6 * 0 = 6 * (4 – 4)
	a[4,4]	3002		
	a[5,4]	3004		
Col - 5	a[3,5]	3006	3006 = 3000 + 6	6 = 6 * 1 = 6 * (5 – 4)
	a[4,5]	3008		
	a[5,5]	3010		
Col - 6	a[3,6]	3012	3012 = 3000 + 12	12 = 6* 2 = 6 * (6 – 4)
	a[4,6]	3014		
	a[5,6]	3016		
Col - 7	a[3,7]	3018	3018 = 3000 + 18	18 = 6* 3 = 6 * (7 – 4)
	a[4,7]	3020		`= 6 * (j – *lb2*)
	a[5,7]	3022		= 2*3(j – *lb2*)
	a[5,7]	3022		

In general, displacement = w * (ub1 – lb1 + 1) * (j – lb2)

where

w = 2 (size of integer), w = 4 (size of float), w = 8 (size of double)
ub1 – lb1 + 1 represent number of items in each column
j is the column index
lb2 is the lower bound of column

Finding y (displacement of ith item from jth column): Consider any column and corresponding addresses. Let us consider column 4 elements as shown below:



Substituting x and y in relation (1) we get:

The location of a[i][j] is given by:

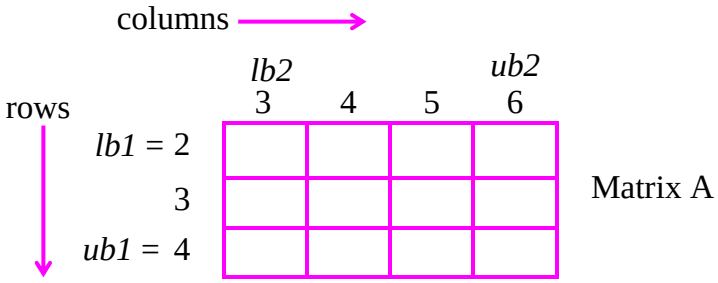
$$\text{Loc A[i][j]} = \text{base address(a)} + w * (ub1 - lb1 + 1) * (j - lb2) + w * (i - lb1)$$

Example 2.16: Consider the array declaration in Pascal language:

a : array[2..4, 3..6] of integer; // size of array is 3 x 4

Find the location a[3][5] when elements are stored in column-major order

The pictorial representation of above array can be written as shown below:



It is given that: a : array[2..4,3..6] of integer;

So, *lb1* = 2, *ub1* = 4, *lb2* = 3, *ub2* = 6, w = 2 (size of integer)

We need to find location of a[3][5]

So, i = 3, j = 5, Let Base address = 3000

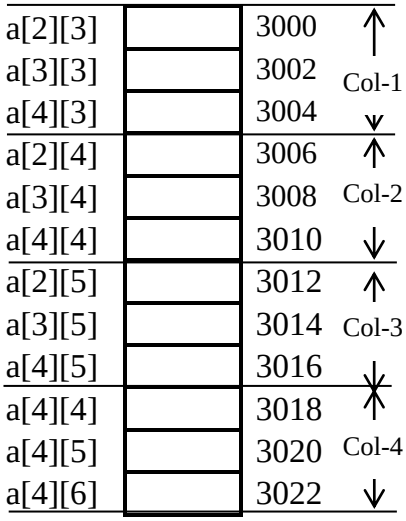
Loc A[i][j] = base address(a) + w * (ub1 – lb1 + 1) * (j – lb2) + w * (i – lb1)

Loc A[3][5]= 3000 + 2 * (4 – 2 + 1) * (5 – 3) + 2 * (3 – 2)

= 3000 + 12 + 2

= 3014

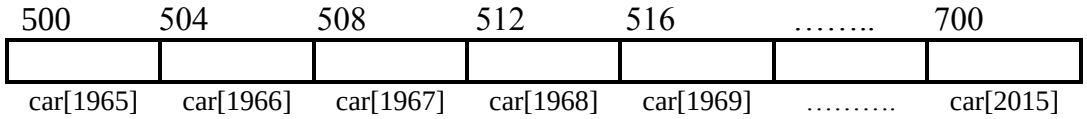
Observe from the following figure that location a[3][5] is indeed 3014 where the 2-dimensional array elements are stored using column major order:



Exercises

1) What is an array? How to obtain the location of a[j] in a single dimensional array?

2) A car manufacturing company uses an array *car* to record number of cars sold each year starting from 1965 to 2015. Rather than beginning the array index from 0 or 1, it is more useful to begin the array index from 1965 as shown below:



   a) Find the total number of years
   b) Suppose base address = 500, word length w = 4, find address of car[1967], car[1969] and car[2015]

3) Consider the linear arrays AAA(5:50), BBB(-5:10) and CCC(1:18)
   a) Find the number of elements in each array
   b) Suppose Base(AAA) = 300 and w = 4 words per memory cell for AAA. Find the address of AAA[15], AAA[35] and AAA[55]

4) What are the various operations that can be performed on arrays?

5) What is traversing an array? Write C functions to read and print n numbers

6) How to insert an item into an unsorted array based on the position? Write the function for the same

7) How to delete an item from an unsorted array based on the position? Write the function for the same.

8) What is sorting? Write a function to sort the elements using bubble sort?

9) What is searching? What are the various searching techniques?

10) What is linear search? Write a function to search for an element using linear search

11) What is binary search? What is the necessary condition for binary search? Write a function to search for an element using linear search

2.38 Arrays

12) What are 2-dimensional arrays? Write a function to read and print two-dimensional array

13) What is row major order? Show how 2 dimensional array is represented in memory using row major order. Obtain the location of a[i][j] using row major order.

14) What is column major order? How to obtain the address of a[i][j] when array elements are stored in column-major order?

15) Consider the array declaration in Pascal language:

```
a : array[2..4, 3..6] of integer;          // size of array is 3 x 4
```

Find the location a[3][5] when elements are stored in row-major order

16) Consider the array declaration in Pascal language:

```
a : array[2..4, 3..6] of integer;          // size of array is 3 x 4
```

Find the location a[3][5] when elements are stored in column-major order

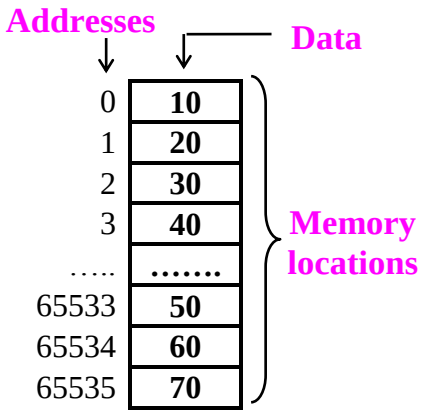
Chapter 3: Pointers

What are we studying in this chapter?

- Pointers
- Dynamic memory allocation functions
- Dynamically allocated arrays (dynamic arrays)

3.1 Pointers

As we store any data or information in our memory, the computer stores data in computer memory as shown below:



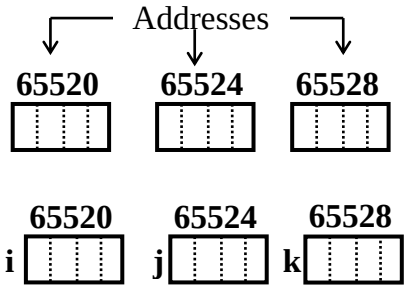
- The computer memory is divided into number of cells called memory locations. Each location can hold only one byte of data.
- Each location is associated with address. In the figure, addresses of memory locations ranges from 0 to 65535.
- We cannot change these addresses assigned by the system and hence they are constants; But, we can only use them to store data. These addresses are called pointer constants. In these memory locations, the data can be stored. Here,10, 20, 30, 40, …..70 represent the data.

Consider the following declaration:

```
int     i = 100,  j = 200, k = 300;
```

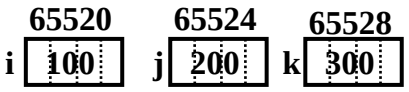
This declaration tells the compiler to perform the following activities:

- Allocate the memory for the variables *i*, *j* and k. Assume 4 bytes are reserved for each variable starting from address 65520. Only starting address of each memory location is shown:
- Associate the variable names *i*, *j* and *k* with these memory locations



3.2 Pointers

- Store 100, 200 and 300 at the locations *i*, *j* and *k* respectively as shown in the figure.



Note: Address of variable can be accessed by prefixing the address operator & with the variable. For example, address of *i* denoted by &i is 65520, address of *j* denoted by &j is 65524, address of *k* denoted by &k is 65528

Now, let us see "What are pointer variables?"

Definition: A variable which holds address of another variable or a memory location is called a pointer variable. For example, if *p* is a pointer variable, it can hold the address of variable *i*. The physical memory representation and logical memory representation is shown below:

Since *p* contains address of a variable *i*, the variable *p* is called a *pointer variable.* Now, let us see "How to declare a pointer variable?" Declare the variable in usual manner and insert * operator between *type* and *variable* as shown below:

Normal declaration **insert *** ⇨ **Pointer declaration**

```
Ex1:  int    a;    //integer variable        int    *a;    // a is pointer to integer
Ex2:  float  b;    //float variable          float  *b;    // b is pointer to float
```

Now, let us see "What are the steps to be followed to use pointers?" The following sequences of operations have to be performed by the programmer:

Steps to be followed while using pointers

Declare a data variable	**Ex: int a;**
Declare a pointer variable	**Ex: int *p;**
Initialize a pointer variable	**Ex: p = &a;**
Access data using pointer variable	**Ex: printf("%d",*p);**

Now, let us see "How to access the value of a variable using pointer?" The value of a variable can be accessed as shown below:

- Using primitive data type: The data stored using primitive data types can be accessed using the variable names. For example,

```
int     a = 10;                    a
                                  10
printf("%d", a);                  // Output: 10
```

- Using pointer type: If a pointer variable contains address of a memory location, then the data can be accessed using dereferencing operator *. For example,

Program segment	Logical memory representation

int *pa; // pointer variable
int a = 10; // data variable
	pa [ ] a: **10**

pa = &a;	**pa** [ ] → a: **10**
printf("%d %d\n", *pa, a);	// **Output:** 10 10

Now, let us see "What is a NULL pointer?" A NULL pointer is defined as a special pointer value that points to '\0' (nowhere) in the memory. If it is too early in the code to assign a value to the pointer, then it is better to assign NULL (i.e., \0 or 0) to the pointer. For example, consider the following code:

```
#include <stdio.h>
int     *p  = NULL;
```

Pictorial representation: p [NULL]

Here, the pointer variable **p** is a NULL pointer. *This indicates that the pointer variable p does not point to any part of the memory.* The value for NULL is defined in the header file "stdio.h". The programmer can access the data using the pointer variable *p* if and only it does not contain NULL. The error condition can be checked using the following statement:

```
if (p == NULL)
        printf("p does not point to any memory\n");
else {
        printf("Access the value of p\n");
        ……..
}
```

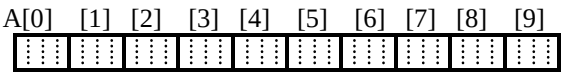
3.2 Memory allocation functions

In this section, let us see "What are the various memory allocation techniques?" Memory can be allocated for variables using two different techniques:

Memory allocation techniques → **Static allocation** (User declarations and definitions)
Memory allocation techniques → **Dynamic allocation** (Using predefined functions)

Now, let us see "What is static memory allocation?" If memory space to be allocated for various variables is decided during compilation time itself, then the memory space cannot be expanded to accommodate more data or cannot be reduced to accommodate less data. In this technique, once the size of the memory space to be allocated is fixed, it cannot be altered during execution time. This is called "*static memory allocation*". For example, consider the following declaration:

int a[10];



During compilation, the compiler will allocate 10 locations (each location consisting of 4 bytes say) for the variable ***a***. In the worst case, 10 elements can be inserted. Inserting less than 10 elements leads to underutilization of allocated space and more than 10 elements cannot be inserted.

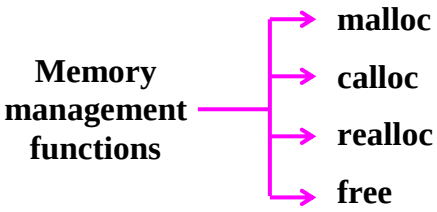
Disadvantages of static memory allocation

- The memory space to be allocated is fixed during compilation. Hence, the memory space cannot be altered during execution time
- Leads to underutilization if more memory space is allocated
- Leads to overflow if less memory is allocated
- The static nature imposes certain limitations and can find their applications only when the data is fixed and known before processing.

Note: All the above disadvantages can be overcome using dynamic memory allocation. If there is an unpredictable storage requirement, then static allocation technique is not used. This is the point where the concept of dynamic allocation comes into picture.

Now, the question is "What is dynamic memory allocation?" Dynamic memory allocation is the process of allocating memory space during execution time (i.e., run time). This allocation technique uses predefined functions to allocate and release memory for data during execution time. So, if there is an unpredictable storage requirement, then the dynamic allocation technique is used. Dynamic allocation will be used when we create dynamic arrays, linked lists, trees

Now, let us see "What are the various memory management functions in C?" The various memory management functions that are available in C are shown below:



3.2.1 malloc(size)

Now, let us see "What is the purpose of using malloc?" This function allows the program to allocate a block of memory space as and when required and the exact amount needed during execution. The size of the block is the number of bytes specified in the parameter. The syntax is shown below:

```
#include <stdlib.h>          /* Prototype definition of malloc() is available */
……..
ptr = (data_type *) malloc(size);
…….
```

where

- *ptr* is a pointer variable of type **data_type**
- **data_type** can be any of the basic data type or user defined data type
- *size* is the number of bytes required

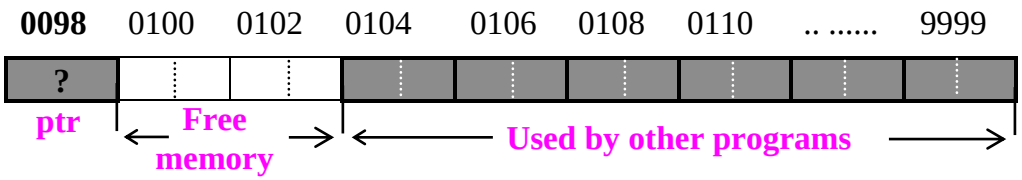
- On successful allocation, the function returns the address of first byte of allocated memory. Since address is returned, the return type is a **void** pointer. By ***type casting*** appropriately we can use it to store integer, float etc.
- If specified size of memory space is not available, the condition is called **"overflow of memory".** In such case, the function returns NULL.

Example 3.1: Now, let us see "What will happen if the following program segment is executed?

```
int *ptr;
ptr = (int *) malloc (10);
```

The compiler reserves the space for the variable **ptr** in the memory. No initialization is done if **ptr** is a local variable (It is initialized to NULL if it is global variable). Now, using the function malloc() we can request a block of memory to be reserved. A block of memory may be allocated or may not be allocated. Now, let us discuss both the situations.

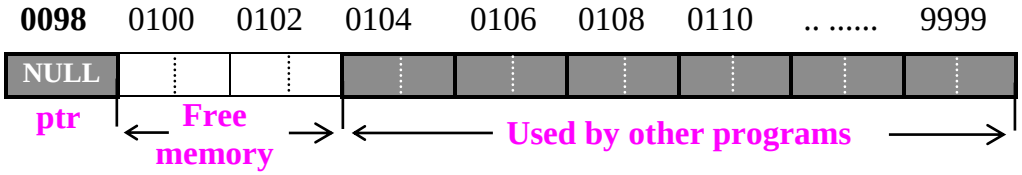
Case 1: Un successful allocation of memory: Consider the following memory status having 4 bytes of free memory space:



Now, let us execute the statement:

```
ptr = (int *) malloc (10);
```

Since 10 bytes of free memory space is not available, the function malloc(), cannot allocate memory and hence the function returns NULL and is copied into **ptr**. The memory status after execution of above statement is shown below:

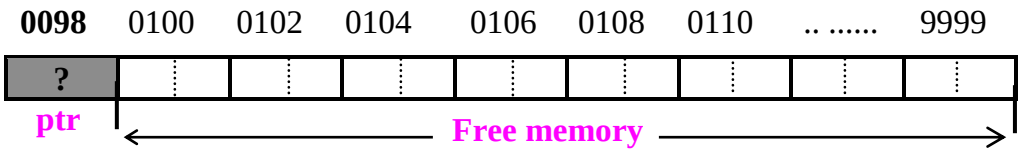


Since **ptr** contains NULL, it indicates that memory has not been allocated and the user should not use **ptr** to store some data. Instead, if **ptr** is NULL appropriate message has to be displayed using the following statement:

```
if (ptr == NULL)
{
        printf("Insufficient memory\n");
        return;
}
```

If **ptr** is not NULL, it indicates that memory has been allocated successfully and allocated memory can be used to store the data. This situation is discussed below:

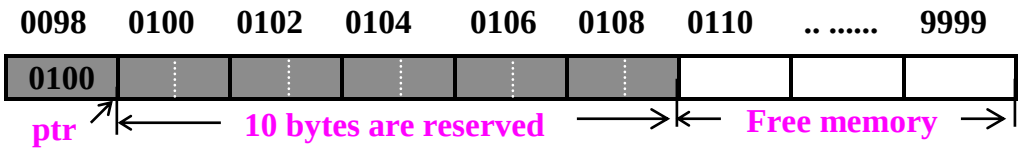
Case 2: Successful allocation of memory: Consider the following memory status:



Now, let us execute the statement:

```
ptr = (int *) malloc (10);
```

Since 10 bytes of free memory space is available, the function malloc(), allocates a block of 10 bytes and returns address of the first byte. This returned address is stored in pointer variable **ptr** as shown below:

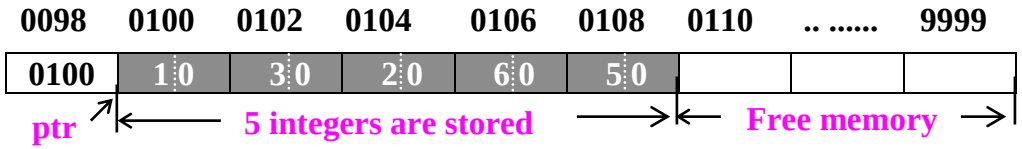


Note: **ptr** points only to the first byte of memory block of 10 bytes reserved using malloc() function. But, the allocated memory is not initialized.

Now, let us see "How to read the data into allocated memory?" Because of type cast **(int *)**, every two bytes are used to store an integer. So, totally 5 integers can be read and stored in the allocated memory of 10 bytes using the following statement:

```
for (i = 0; i < 5; i++)
        scanf("%d", ptr + i);          /* Input: 10  30  20  60  50  */
```

If we input 10, 30, 20, 60 and 50 after executing the above statement, these numbers are stored in memory as shown below:



Note: The expression **ptr + i** can be written as **&ptr[i]** or **&i[ptr].**

Now, let us see "How to access the data from allocated memory?" The contents of the above memory locations can be accessed using ***(ptr+i)** with the help of de-reference operator or using **ptr[i]** or **i[ptr]** using array notation. The equivalent statements are:

```
for (i = 0; i < 5; i++)
        printf("%d  ", *(ptr+i) );
```

Example 3.2: Program showing the usage of malloc() function (dynamic arrays)

```
#include <stdio.h>
#include <stdlib.h>

void main()
{
        int     i,n;
        int     *ptr;

        printf("Enter the number of elements\n");
        scanf("%d",&n);

        ptr = (int *) malloc (sizeof(int)* n);

        /* If sufficient memory is not allocated */
        if (ptr == NULL)
        {
                printf("Insuffient memory\n");
                return;
        }

        /* Read N elements */
        printf("Enter N elements\n");
        for (i = 0; i < n; i++)
                scanf("%d", ptr+i);

        printf("The given elements are\n");
        for (i = 0; i < n; i++)
                printf("%d  ", *(ptr+i));
}
```

TRACING

Execution starts from here

Inputs

Enter the number of elements

5

Enter N elements

10 20 30 40 50

The given elements are

10 20 30 40 50

3.2.2 calloc(n, size)

Now, let us see "What is the purpose of using calloc?"

This function is used to allocate multiple blocks of memory. Here, **calloc** – stands for contiguous allocation of multiple blocks and is mainly used to allocate memory for arrays. The number of blocks is determined by the first parameter **n.** The size of each block is equal to the number of bytes specified in the parameter i.e., **size.** Thus, total number of bytes allocated is n*size and all bytes will be initialized to 0. The syntax is shown below:

```
#include <stdlib.h>          /* Prototype definition of calloc() is available */
```

```
……..
……..
ptr = (data_type *) calloc(n, size);
……..
……..
```

where

- *ptr* is a pointer variable of type **data_type**
- **data_type** can be any of the basic data type or user defined data type
- **n** is the number of blocks to be allocated
- *size* is the number of bytes in each block

Now, let us see "What does this function return?" The function returns the following values:

- On successful allocation, the function returns the address of first byte of allocated memory. Since address is returned, the return type is a **void** pointer. By **type casting** appropriately we can use it to store integer, float etc.
- If specified size of memory is not available, the condition is called **"overflow of memory".** In such case, the function returns NULL.

Note: It is the responsibility of the programmer to check whether the sufficient memory is allocated or not as shown below:

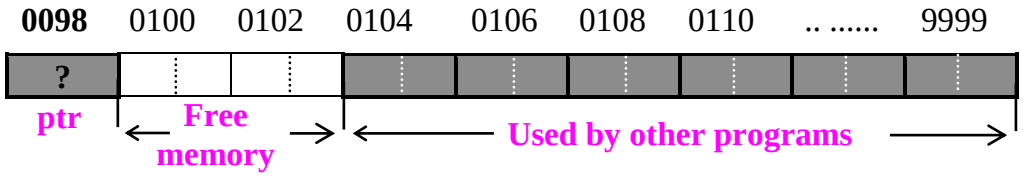
```
void function_name()
{
        ……..
        ptr = (data_type *) calloc(n, size);
        if (ptr == NULL)
        {
                printf("Insufficient memory\n");
                exit(0);
        }
        ……..
        ……..
}
```

Example 3.3: What will happen if the following program segment is executed?

```
int *ptr;
ptr = (int *) calloc (5, sizeof(int));
```

Solution: The compiler reserves the space for the variable **ptr** in the memory. No initialization is done if **ptr** is a local variable (It is initialized to NULL if it is global variable). Now, using the function calloc() we can request specified number of blocks of memory to be reserved. The specified blocks of memory may be allocated or may not be allocated. Now, let us discuss both the situations.

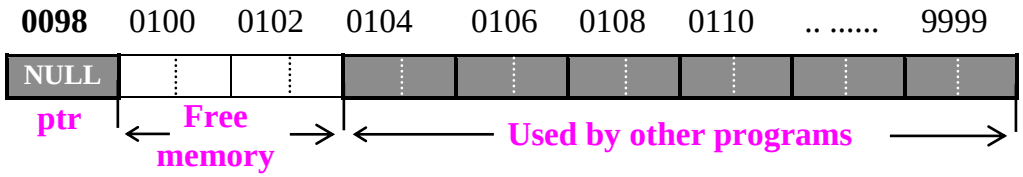
Case 1: Unsuccessful allocation of memory: Consider the following memory status where 4 bytes of free space is available:



Now, let us execute the statement:

```
ptr = (int *) calloc (5, sizeof(int));
```

Since 5*sizeof(int) = 5*2 = 10 bytes of free memory space is not available, the function calloc(), cannot allocate memory and hence the function returns NULL and is copied into **ptr**. The memory status after execution of above statement is shown below:

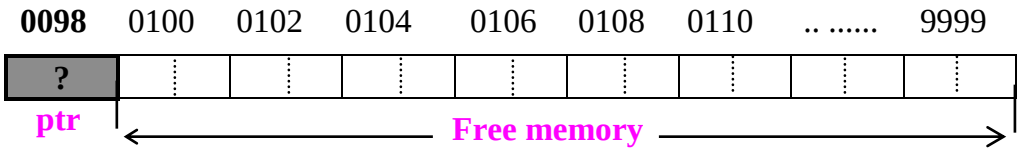


Since **ptr** contains NULL, it indicates that memory has not been allocated and the user should not use **ptr** to access the data. Instead, if **ptr** is NULL appropriate message has to be displayed using the following statement:

```
if (ptr == NULL)
{
        printf("Insufficient memory\n");
        return;
}
```

If **ptr** is not NULL, it indicates that memory has been allocated successfully and allocated memory can be used to store the data. This situation is discussed below:

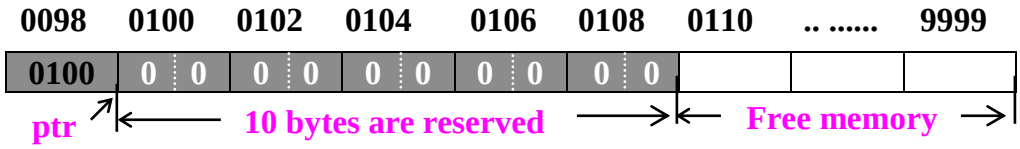
Case 2: Successful allocation of memory: Consider the following memory status:



Now, let us execute the statement:

```
ptr = (int *) calloc (5, sizeof(int));
```

Since 5*sizeof(int) = 5*2 = 10 bytes of free memory space is available, the function calloc(), allocates 5 blocks of 2 bytes each, initializes each byte to 0 and returns the address of the first byte. This returned address is stored in pointer variable **ptr** as shown below:

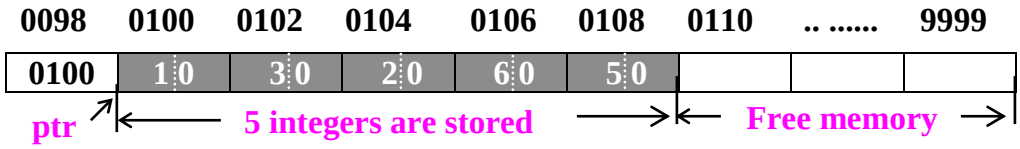


Note: **ptr** points only to the first byte and the allocated memory is initialized to 0's.

Now, let us see "How to read the data into allocated memory?" Because of type cast **(int *)**, every two bytes are used to store an integer. So, totally 5 integers can be read and stored in the allocated memory of 10 bytes using the following statement:

```
for (i = 0; i < 5; i++)
        scanf("%d", ptr+i);          /* Input: 10  30  20  60  50  */
```

If we input 10, 30, 20, 60 and 50 after executing the above statement, these numbers are stored in memory as shown below:



Note: The expression **ptr + i** can be written as **&ptr[i]** or **&i[ptr].**

Now, let us see "How to access the data from allocated memory?"

The contents of the above memory locations can be accessed using ***(ptr+i)** with the help of de-reference operator or using **ptr[i]** or **i[ptr]** using array notation. The equivalent statements are:

```
for (i = 0; i < 5; i++)
        printf("%d  ", *(ptr+i) );
```

The program below shows how to find maximum of *n* numbers which uses calloc() function. This program prints the largest of *n* elements along with its position. Note that the value of *n* is supplied only during execution time. This is straightforward program and it is self-explanatory.

Example 3.4: Program to find maximum of n numbers using dynamic arrays

```
#include <stdio.h>
#include <stdlib.h>

void main()
{
        int *a, i, j, n;

        printf("Enter the no. of elements\n");
        scanf("%d",&n);

        /* Allocate the required number of memory locations dynamically */
        a = (int *) calloc( n, sizeof(int) );

        if (a == NULL)                          /* If required amount of memory  */
        {                                       /* is not allocated */
                printf("Insufficient memory\n");
                return;
        }

        printf("Enter %d elements\n", n);   /* Read all elements */
        for ( i = 0; i < n; i++)
        {
                scanf("%d", &a[i]);
        }
```

```
        j = 0;                          /* Initial position of the largest number */

        for ( i = 1; i < n; i++)
        {
                if ( a[i] > a[j] ) j = i;       /* obtain position of the largest element*/
        }

        printf("The biggest = %d is found in pos = %d\n",a[j], j+1);

        /* free the memory allocated to n numbers */
        free(a);
}
```

Note: In the above program **&a[i]** or **&a[j]** can be replaced by **a + i** or **a + j.** At the same time **a[i]** and **a[j]** can be replaced by ***(a + i)** and ***(a + j)** respectively.

Now, let us see "What is the difference between malloc() and calloc()?" Even though malloc() and calloc() are used to allocate memory, there is significant difference between these two techniques. The difference between these two lies in the way the memory is allocated and initialized along with the number of parameters passed to them.

malloc	**calloc**
1. The syntax of malloc() is: ptr = (**data_type***) malloc(**size**); The required number of bytes to be allocated is specifies as argument i.e., **size** in bytes	**1.** The syntax of calloc is: ptr = (**data_type** *) calloc(n, **size**); Takes two arguments: **n** number of blocks to be allocated **size** is number of bytes to be allocated for each block.
2. Allocates a block of memory of **size** bytes	2. Allocates multiple blocks of memory, each block with the same **size**
3. Allocated space will not be initialized	3. Each byte of allocated space is initialized to zero
4. Since no initialization takes place, time efficiency is higher than calloc().	4. calloc() is slightly more computationally expensive because of zero filling but, occasionally, more convenient than malloc()

5. This function can allocate the required size of memory even if the memory is not available contiguously but available at different locations.	5. This function can allocate the required number of blocks contiguously. If required memory cannot be allocated contiguously, it returns NULL.
6. Allocation and initialization of allocated memory with 0's can be done using the following statements: p = **malloc**(sizeof(int) * n); memset(p, 0, sizeof(int) * n)	6. Allocation and initialization of allocated memory with 0's can be done using the following statements: p = **calloc**(sizeof(int) * n);

3.2.3 realloc(ptr, size)

Now, let us see "What is the purpose of using realloc?"

Before using this function, the memory should have been allocated using malloc() or calloc(). Sometimes, the allocated memory may not be sufficient and we may require additional memory space. Sometimes, the allocated memory may be much larger and we want to reduce the size of allocated memory. In both situations, the size of allocated memory can be changed using realloc() and the process is called reallocation of memory. The reallocation is done as shown below:

- realloc() changes the size of the block by extending or deleting the memory at the end of the block.
- If the existing memory can be extended, **ptr** value will not be changed
- If the memory cannot be extended, this function allocates a completely new block and copies the contents of existing memory block into new memory block and then deletes the old memory block. The syntax is shown below:

```
#include <stdlib.h>          /* Prototype definition of realloc() is available */
……..
……..
ptr = (data_type *) realloc(ptr, size);
…….
…….
```

where

- ptr is a pointer to a block of previously allocated memory either using malloc() or calloc().
- **size** is new size of the block

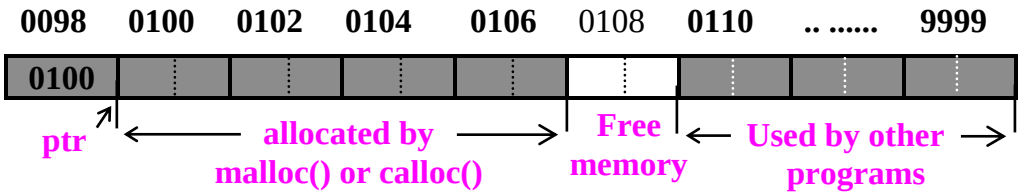
```
if (ptr == NULL)
{                                        /* Memory is not allocated */
        printf("Insufficient memory\n");
        return;
}
```

Now, let us see "What does this function return?" The function returns the following values:

- On successful allocation, the function returns the address of first byte of allocated memory.
- If specified size of memory cannot be allocated, the condition is called **"overflow of memory".** In such case, the function returns NULL.

Now, let us discuss each of the above case one by one.

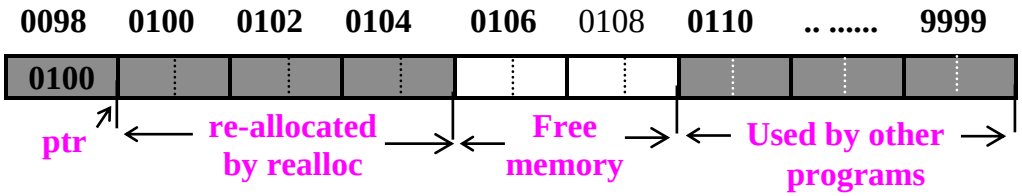
Case 1: Reducing the size of the allocated memory. Consider the following memory status



Assume, **ptr** points to the address of the first byte of memory block. After executing the statement:

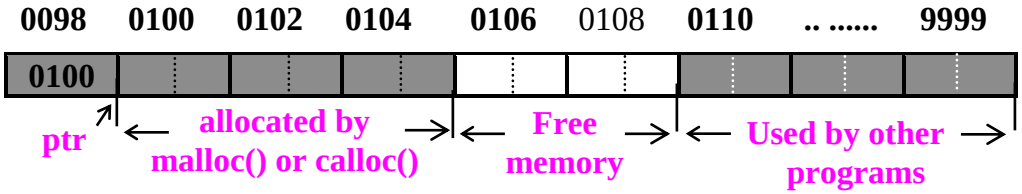
```
ptr = (int *) realloc(ptr, 3*sizeof(int) )
```

only 3*2 = 6 bytes are re-allocated starting from 0100 and last block of memory starting from 0106 is de-allocated and the resulting memory status is shown below:



Observe from the above memory status that memory is deleted from the end of block there by free memory space is increased.

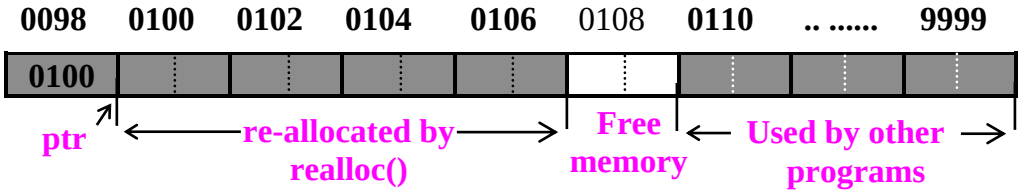
Case 2: Extend the allocated memory without changing the starting address.
Consider the following memory status



Assume, **ptr** points to the address of the first byte of memory block. After executing the statement:

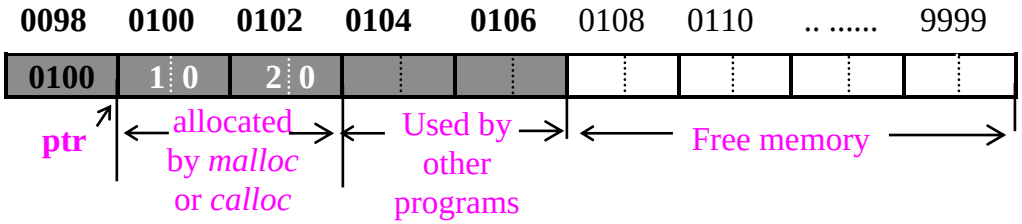
ptr = (**int** *) realloc(ptr, 4*sizeof(**int**))

only 4*2 = 8 bytes are re-allocated starting from 0100 and the resulting memory status is shown below:



Observe from the above memory status that free memory space is reduced by re-allocating the extra memory.

Case 3: Extend the allocated memory by changing the starting address. Consider the following memory status.

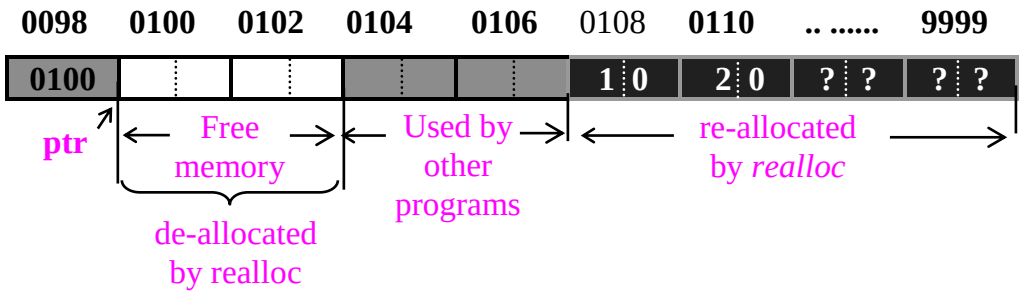


Assume, **ptr** points to the address of the first byte of memory block and integers **10** and **20** are stored in that memory block. After executing the statement:

ptr = (**int** *) realloc(ptr, 4*sizeof(**int**))

4*2 = 8 bytes should be re-allocated. But, the existing memory block cannot be extended because there is no free space at the end of the current block. So, realloc

allocates a completely new block, copies the existing memory contents to the new block and deletes the old allocation as shown below:



After copying the data from old block to new block, the remaining locations of new block are not initialized and hence are represented using **??**. Thus, the function realloc() guarantees that re-allocating the memory will not destroy the original contents of memory.

Example: 3.5: C program showing the usage of realloc() function.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void main()
{
        char *str;

        /* allocate memory for the string */
        str = (char *) malloc(10);
        strcpy(str, "Computer");

        str = (char *) realloc(str, 40);
        strcpy(str, "Computer Science and Engineering");
}
```

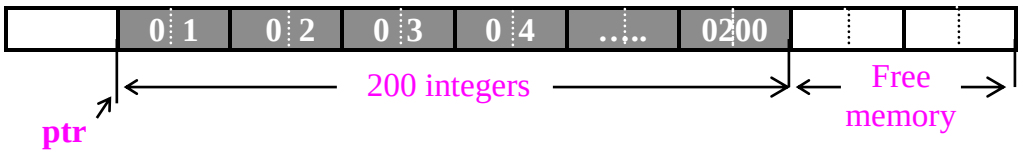
3.2.4 free(ptr)

Now, let us see "What is the purpose of using free()?"

This function is used to de-allocate (or free) the allocated block of memory which is allocated using the functions calloc(), malloc() or realloc(). It is the responsibility of a programmer to de-allocate memory whenever it is not required by the program and initialize **ptr** to NULL. The syntax is shown below:

```
#include <stdlib.h>          /* Prototype definition of free() is available */
……..
free(ptr);
ptr = NULL;
…….
…….
```

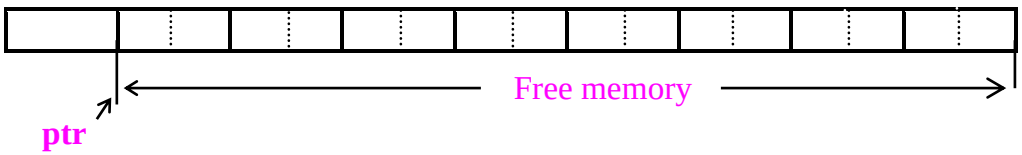
Here, ***ptr*** is a pointer to a memory block. Now, let us see how this function works. For example, consider the following memory configuration where **ptr** points to a memory block containing 200 integers ranging from **01** to **0200**.



After executing the statement:

```
free(ptr);
```

the following memory configuration is obtained.



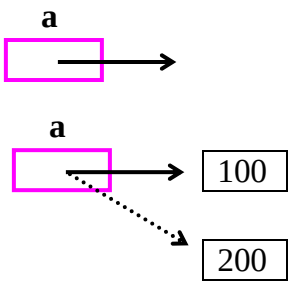
This shows the memory allocated for 200 integers is de-allocated and is available as part of the free memory (heap area). Observe from the above figure that even after freeing the memory, the pointer value stored in **ptr** is not changed. It still contains the address of the first byte of the memory block allocated earlier. Using the pointer **ptr** even after the memory has been released is a logical error.

Note: These logical errors are very difficult to debug and correct. So, immediately after freeing the memory, it is better to initialize to NULL as shown below:

```
……..
/* allocate the memory */
……..
/* de-allocate the memory */
free (ptr);
ptr = NULL;
```

Example: 3.6: Program to shown the problems that occur when free() is not used.

```
1. #include <stdlib.h>
2.
3. void main()
4. {
5.      int *a;
6.
7.      a = (int *) malloc(sizeof(int));
8.      *a = 100;
9.
10.     a = (int *) malloc(sizeof(int));
11.    *a = 200;
12. }
```



Now, let us see "What will happen if the above program is executed?" The various activities done during execution are shown below:

- When control enters into the function main, memory for the variable **a** will be allocated and will not be initialized.
- When memory is allocated successfully by malloc (line 7), the address of the first byte is stored in the pointer **a** and integer **100** is stored in the allocated memory (line 8).
- But, when the memory is allocated successfully by using the function malloc in line 10, address of the first byte of new memory block is copied into **a (**shown using dotted lines.)

Observe that the pointer **a** points to the most recently allocated memory, thereby making the earlier allocated memory inaccessible. So, memory location where the value **100** is stored is inaccessible to any of the program and is not possible to free so that it can be reused. This problem where in memory is reserved dynamically but not accessible to any of the program is called **memory leakage.**

So, care should be taken while allocating and de-allocating the memory. It is the responsibility of the programmer to allocate the memory and de-allocate the memory when no longer required.

Note: Observe the following points:

- It is an error to free memory with a NULL pointer
- It is an error to free a memory pointing to other than the first element of an allocated block
- It is an error to refer to memory after it has been de-allocated

Note: Be careful, if we dynamically allocate memory in a function. We know that local variables vanish when the function is terminated. If **ptr** is a pointer variable used in a function, then the memory allocated for **ptr** is de-allocated automatically. But, the space allocated dynamically using malloc, calloc or realloc will not be de-allocated automatically when the control comes out of the function. But, the allocated memory cannot be accessed and hence cannot be used. This unused un-accessible memory results in memory leakage.

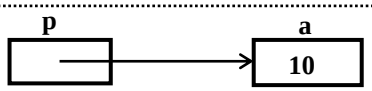
3.3 Pointers can be dangerous

"Pointers can be dangerous: Justify your answer?" The pointers are dangerous in following situations:

- We may attempt to access an area of memory that is out of range of our program. For example, consider the statements:

```
int     *p;
int     a = 10;
```


```
p = &a;
```

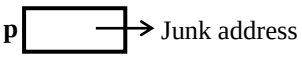



```
printf("%d", *p)        // Output: 10
printf("%d", *(p-1)); // accessing memory which is out of range
```

Observe that *p* contains address of *a*. So, *p refers to value of *a*. But, *(p-1) refers to the value in previous location which does not exist. So, we are accessing memory location which is outside our program area which eventually may crash the program.

- The pointer may not contain address of legitimate variable (memory location). For example, consider the following program segment:

```
int     *p;
```



The pointer *p* contains invalid address. So, de-referencing *p* results in unpredictable output and may even crash the program.

- We may de-reference a NULL pointer. When a NULL pointer is de-referenced, some computers may return 0 and some computers may return data in location zero producing serious error. So, we should never try to dereference NULL pointer.

- In many computers, the size of **int** data type and size of pointer is same. So, a valid integer value can be interpreted as a pointer which is very serious error. For example, when a function returns an integer value, the returned value can be interpreted as a pointer. This is very serious error.

3.4 Dynamically allocated arrays

3.4.1 One dimensional arrays

In the previous section, we have seen how to allocate memory for single dimensional array dynamically.

3.4.2 Pointers to Pointers

Before proceeding further, let us see "What is pointer to a pointer?"

Definition: A variable which contains address of a pointer variable is called *pointer to a pointer*. So, it is possible to make a pointer to point to another pointer variable.

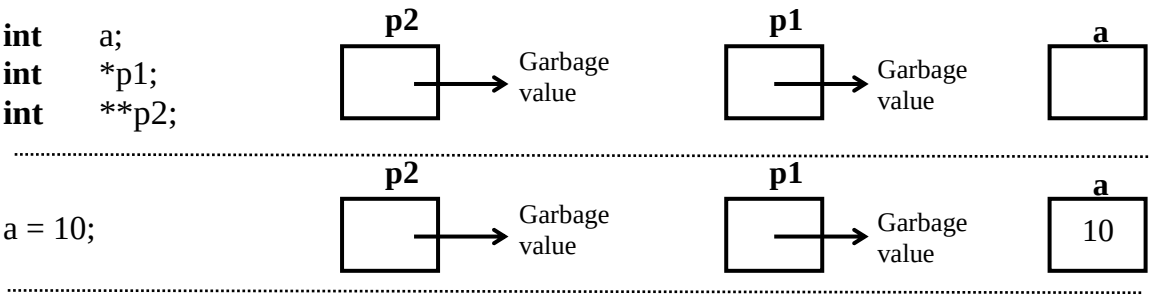
For example, consider the following declarations:

```
int     a;          // Used to store an integer
int     *p1;        // Used to store address of integer variable
int     **p2;       // Used to store address of pointer to integer variable
```

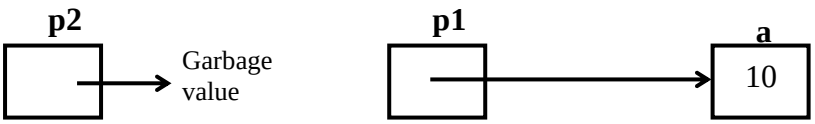
Example 3.7: Program to demonstrate a pointer to a pointer

```
int     a;
int     *p1;
int     **p2;
a = 10;
p1 = &a;
p2 = &p1;
```

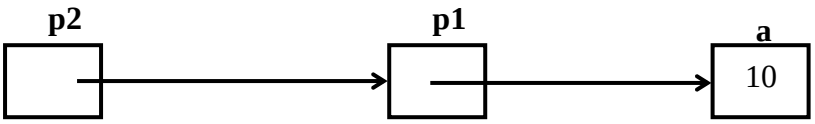
The memory representation after executing each of the statement is shown below:



p1 = &a;



p2 = &p1;




```
printf ("%d\n", a);          // 10
printf("%d\n", *p1);         // 10
printf("%d\n", **p2);        // 10
```

Observe the following points in the above program segment:

a refers to the data item 10

*p1 also refers to the data item 10. Using *p1* and de-referencing operator, the data 10 can be accessed

**p2 also refers to the data item 10. Here, using *p2* and two indirection operators, the data item 10 can be accessed.

3.4.3 Generalized macro to allocate memory using malloc function

If we frequently allocate the memory space, then it is better to use functions as shown below where *n* is number of items.

Example 3.8: Functions to allocate memory for integers, floats, chars and doubles.

Allocating memory for integers

```
int * MALLOC_INT(int n)
{
    int *x;

    x = (int *) malloc (n*sizeof(int));

    if ( x == NULL)
    {
        printf ("Insufficient memory\n");
        exit(0);
    }

    return x;
}
```

Allocating memory for floats

```
float * MALLOC_FLOAT(int n)
{
    float *x;

    x = (float *) malloc (n*sizeof(float));

    if ( x == NULL)
    {
        printf ("Insufficient memory\n");
        exit(0);
    }

    return x;
}
```

Allocating memory for characters

```
char * MALLOC_CHAR(int n)
{
    char *x;

    x = (char *) malloc (n*sizeof(char))

    if ( x == NULL)
    {
        printf ("Insufficient memory\n");
        exit(0);
    }

    return x;
}
```

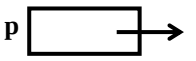
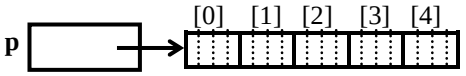
Allocating memory for double values

```
double * MALLOC_DOUBLE(int n)
{
    double *x;

    x = (double *) malloc (n*sizeof(double))

    if ( x == NULL)
    {
        printf ("Insufficient memory\n");
        exit(0);
    }

    return x;
}
```

The above functions have to be invoked appropriately as shown below:

Case 1: To allocate memory for n integers call the function:

`p = MALLOC_INT(n)` where *n* is number of elements in array

Program segment	Logical representation
Step1: int *p;	
Step 2: p = MALLOC_INT(5)	

- In step1, memory is allocated for variable *p* by the compiler and this memory location contains junk value. Hence, *p* is dangling pointer.
- In step2, the function MALLOC_INT(5) is invoked and allocates 5 locations where each location consists of 4 bytes (total 5*4 = 20 bytes). Here, 4 is sizeof(int). The address of the first byte of allocated space is copied into variable *p*. Now, *p* contains valid address and hence it is not a dangling pointer.

Case 2: To allocate memory for *n* floating point numbers, call the function:

`p = MALLOC_FLOAT(n)` where, *n* is number of elements in array

Case 3: To allocate memory for *n* characters, call the function:

p = MALLOC_CHAR(n) where, *n* is number of elements in array

Now, the question is "Can we write a generalized single function to allocate memory for *n* integers or *n* characters etc.?" Answer is no. It is not possible to write a generalized function to allocate memory for n elements of any data type in C. But, we can write a generalized macro in place of a generalized function. The macro definition to allocate memory for *n* elements of any data type can be written as shown below:

Example 3.9: Macro to allocate memory for one or more items of any data type

```
#define MALLOC(p, n, type)                  \
        p = (type*) malloc ( n * sizeof(type) );    \
        if (p == NULL)                      \
        {                                   \
                printf("Insufficient memory\n");    \
                exit(0);                    \
        }
```

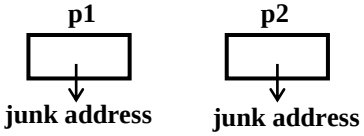
Observe the following points:

- Macros are normally written in a single line.
- Since, the body of the macro consists of more number of statements, for readability purpose we may write the body of macro in multiple lines. ***The symbol backslash (denoted by \) is must at the end of each line.***
- The symbol \ is an instruction given to the pre-processor that all the lines following \ are continuation of the first line of that macro definition.

Example 3.10: Program showing the usage of macro MALLOC

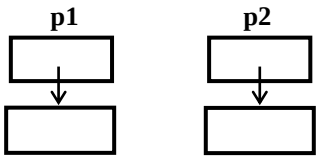
```
#include <stdio.h>
#include <stdlib.h>

// Include : Example 3.9: MACRO to allocate memory for any number of items

void main()
{
        int     *p1, *p2;
        int     sum;
```

Memory representation and tracing

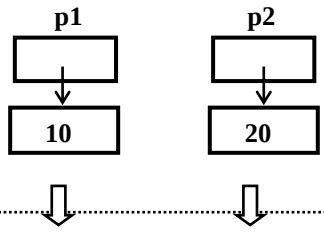


```
MALLOC(p1, 1, int);
MALLOC(p2, 1, int);
```

Note: Allocate memory for 1 integer




```
*p1 = 10;
*p2 = 20;
```




```
sum = *p1 + *p2;
```

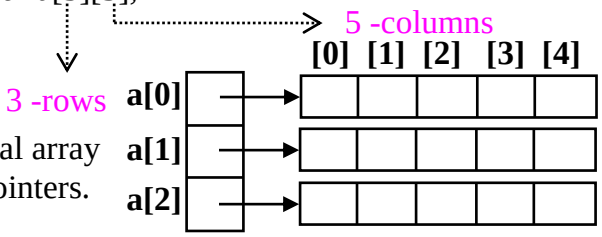
sum = 10 + 20
sum = 30

```
printf("%d + %d = %d\n", *p1, *p2, sum);     // Output: 10 + 20 = 30
}
```

3.4.4 Two dimensional arrays using pointer to pointer

Now, let us see "How two dimensional arrays can be represented in C?" A two-dimensional array is represented as one-dimensional array of pointers where each pointer contains address of one-dimensional array. For example, consider the following declaration:

```
int a[3][5];
```

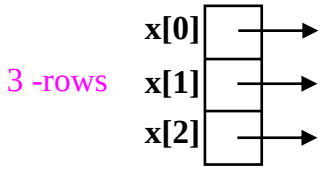


It is a single-dimensional array which can hold three pointers.

Each of the three pointer points to a one-dimensional array consisting of 5 elements

Now, let us see "How to create a 2-dimensional array dynamically?" A two-dimensional array with 3 rows and 5 columns can be created by allocating the memory dynamically in two stages:

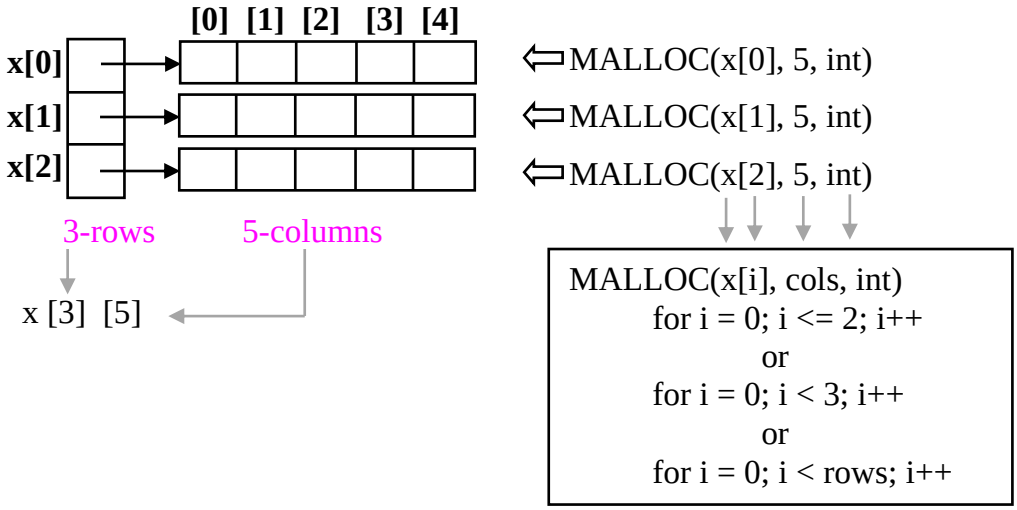
Stage 1: Allocate memory for specified number of row pointers. We can allocate memory for 3 row pointers using: MALLOC(x, 3, int *) whose pictorial representation is shown below:



In general, we use

```
MALLOC(x, rows , int *)     // rows – represent number of rows in
                            // a two-dimensional array
```

Stage 2: Allocate memory for each row with specified number of columns. We can allocate memory for 5 columns with respect to row x[0], x[1] and x[2] using MALLOC() three times as shown below.



By using both stages, we can write the code (that is, program statements) in general, as shown below:

```
MALLOC(x, rows, int *)          // Allocate memory for row pointers
for (i = 0; i < rows, i++)
{                               // Allocate memory for specified number
        MALLOC(x[i], cols, int);//       of columns
}
return x;                       // Return address of 0th row and 0th column
```

Now, the complete function to allocate memory for x[row][col] can be written as shown below:

Example 3.11: Allocating memory for 2-dimensional array dynamically

```
int ** make2dArray(int rows,  int cols)
{
        int     **x;            // Two stars for two dimensional array
        int     i;

        MALLOC(x, rows, int *);         // allocate memory for row pointers

        for (i = 0; i < rows; i++)
        {
            MALLOC(x[i], cols, int);    // allocate memory for each row
        }

        return x;                       // Return address of 0th row and 0th column
}
```

Note: Observe the following points:

*a with one star is used to store address of one-dimensional array dynamically

**a with two stars is used to store address of two-dimensional array dynamically

***a with three stars is used to store address of three-dimensional array dynamically

Declaration	Stores address of	Access address	Data access
*a	one-dimensional array	&a[i] or (a+i) or (i + a) or &i[a]	a[i] or *(a+i) or *(i+a) or i[a]
**a	two-dimensional array	&a[i][j]	a[i][j]
***a	three-dimensional array	&a[i][j][k]	a[i][j][k]

Note: Allocation of memory for arrays can be done using static allocation and dynamic allocation as shown below:

Static allocation	Dynamic allocation
int a[5];	**int** *a; MALLOC(a, 5, int);
float b[10];	**float** *b; MALLOC(b, 10, float);

int a[5][10];
2-d array. Hence, 2 stars
	int ** make2dArray(**int** row, **int** col)
	{
	int **x, i;
	MALLOC(x, row, **int** *);
	for (i = 0; i < row; i++)
	MALLOC(x[i], col, **int**);
	return x;
	}

3.4.4.1 Read/Write a matrix for dynamically allocated arrays

The display function is similar to the way we display a two-dimensional array using arrays. This function can be written as shown below:

Example 3.12: Function to display 2-dimensional array created dynamically

```
                                  can also be written as int [][MAX_COLS]
void print_array(int  **a,  int rows,  int cols)
{
        int     i;              // Row index
        int     j;              // Column index

        for (i = 0; i < rows; i++)
        {
                for (j = 0; j < cols; j++)
                        printf("%4d", a[i][j]);
                                          can also be written as   *(a[i]+j)
                printf("\n");
        }
}
```

Note: MAX_COLS is a symbolic constant defined using preprocessor directive as
#define MAX_COLS 10

The function to read elements into 2-d array created dynamically is similar to the way we read a 2-dimensional array using arrays. This function can be written as shown below:

Example 3.13 : Function to read elements into 2-d array created dynamically

```
                        can also be written as int [][MAX_COLS]
void read_array(int **a,   int rows,   int cols)
{
        int     i;              // row index
        int     j;              // column index

        for (i = 0; i < rows; i++)
        {
                for (j = 0; j < cols; j++)
                        scanf("%d", &a[i][j]);
                                        can also be written as scanf("%d", a[i]+j );
        }
}
```

3.4.4.2 Initialization using malloc() system call

Now, let us "Make fewest number of changes to the function shown in example 3.11 and write a function that creates a 2-d array all of whose elements are initialized to 0."

Example 3.14: Function to initialize allocated memory locations to 0 using malloc()

```
int ** make2dArray(int rows, int cols)
{
        int     **x;                    // To hold address of 0th row and 0th colums
        int     i, j;

        MALLOC(x, rows, int*);                  // allocate memory for row pointers

        for(i = 0; i < rows; i++)
        {
                MALLOC(x[i], cols, int);        // allocate memory for each row
                for (j = 0; j < cols; j++) x[i][j] = 0;  // Initialize all locations to 0's
        }

         return x;                              // return base address
}
```

Note: It is same as example 3.11. But, using for loop, we initialize each location to 0

The C program to create a two-dimensional array and initializes memory locations to 0 can be written as shown below:

Example 3.15: Program to initialize allocated memory locations to 0 using malloc()

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_ROWS  5
#define MAX_COLS  10

// Insert Example 3.9: A macro MALLOC to allocate the memory

// Insert Example 3.14: Function to create 2-dimeinsional array and initialize to 0

// Insert Example 3.12: To display two-dimensional array

void main()
{
        int     **a;            // Holds address of 0th row and 0th column of matrix A
        int     rows, cols;     // Number of rows and columns

        printf("Enter the size of the matrix\n");
        scanf("%d %d", &rows, &cols);

        // check for invalid rows and columns
        if (rows < 0 || cols < 0)
        {
                printf("Invalid matrix size\n");
                exit(0);
        }

        a = make2dArray(rows, cols);            // create matrix A dynamically

        printf("The initialized matrix\n");     // The initialized matrix
        print_array(a, rows, cols);             //      0  0  0
}                                               //      0  0  0
```

3.4.4.3 Matrix Addition using dynamic allocation

Now, let us "Write a program to add two matrices using dynamically allocated arrays" The header for the function should be:

```
void add(int **a, int **b, int **c, int rows, int cols);
```

The function is similar to the way to add two matrices using arrays. The complete function can be written as shown below:

Example 3.16: Function to add two matrices that are created dynamically

```
void add(int **a, int **b, int **c, int rows, int cols)
{
        int     i;              // row index
        int     j;              // column index
        for (i = 0; i < rows; i++)
        {
                for (j = 0; j < cols; j++)
                {
                        c[i][j] = a[i][j] + b[i][j] ;   // can also be written as
                }                                       // *(c[i]+ j) = *(a[i] + j) + *(b[i] + j)
        }
}
```

The complete C program to add two matrices created dynamically is shown below:

Example 3.17: Program to add two matrices that are created dynamically

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_ROWS  5
#define MAX_COLS  10
```

Include : **Example 3.9 :** Macro to allocate memory dynamically

Include : **Example 3.11 :** To create a 2-d array dynamically

Include : **Example 3.12 :** To display elements of 2-d array created dynamically

Include : **Example 3.13 :** To read the elements into 2-d array created dynamically

Include : **Example 3.16:** To add the elements of 2 matrices created dynamically

```
void main()
{
        int     **a;            // Holds address of 0th row and 0th column of matrix A
        int     **b;            // Holds address of 0th row and 0th column of matrix B
        int     **c;            // Holds address of 0th row and 0th column of matrix C
        int     rows, cols;     // Number of rows and columns

        printf("Enter the size of the matrix\n");
        scanf("%d %d", &rows, &cols);

        if (rows < 0 || cols < 0)
        {
                printf("Invalid matrix size\n");
                exit(0);
        }

        a = make2dArray(rows, cols);                    // create matrix A
        printf("Enter the elements of matrix A\n");     // read matrix A

        read_array(a, rows, cols);

        b = make2dArray(rows, cols);                    // create matrix B
        printf("Enter the elements of matrix B\n");     // read matrix B

        read_array(b, rows, cols);

        c = make2dArray(rows, cols);                    // create matrix C

        add(a, b, c, rows, cols);                       // peform C = A + B

        printf("Sum of two matrices\n");                // output the result
        print_array(c, rows, cols);
}
```

3.4.5 A macro to allocate memory using calloc() function

If we frequently allocate the memory space using various data types, then we can write a macro as shown below:

Example 3.18: Macro to allocate memory for one or more items of any data type

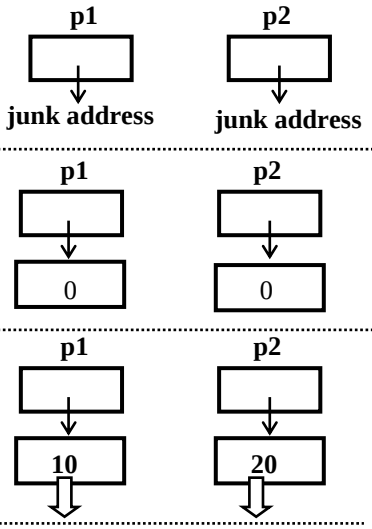
```
#define CALLOC(p, n, type)                          \
        p = (type*) calloc ( n, sizeof(type) );     \
        if (p == NULL)                              \
        {                                           \
                printf("Insufficient memory\n");    \
                exit(0);                            \
        }
```

The program to use the above macro can be written as shown below:

Example 3.19: Program showing the usage of macro CALLOC

```
#include <stdio.h>
#include <stdlib.h>

// Include : Example 3.18: MACRO to allocate memory for any number of items

void main()
{
        int     *p1, *p2;
        int     sum;

        CALLOC(p1,1,int);
        CALLOC(p2,1,int);

        Note: Allocate memory for 1 integer

        *p1 = 10;

        *p2 = 20;

        sum = *p1 + *p2;

        printf("%d + %d = %d\n", *p1, *p2, sum);    // Output: 10 + 20 = 30
}
```

Memory representation and tracing



3.4.6 Macro to allocate memory using realloc() function

If we frequently re-allocate the memory space using various data types, then we can write a macro as shown below:

Example 3.20: Macro to re-allocate memory for one or more items of any data type

```
#define REALLOC(p, n, type)                          \
        p = (type*) realloc (p, n*sizeof(type) );    \
        if (p == NULL)                               \
        {                                            \
                printf("Insufficient memory\n");     \
                exit(0);                             \
        }
```

The program to use the above macro can be written as shown below:

Example 3.21: Program showing the usage of macro REALLOC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Include : Example 3.20: MACRO to reallocate memory for any number of items

void main()
{
        char *str;

        str = (char *) malloc(10);
        strcpy(str, "Computer");
        printf("String = %s\n", str);

        REALLOC(str, 40, char);

        strcpy(str, "Computer Science and Engineering");

        printf("String =%s\n", str);
}
```

Exercises

1) What are pointer variables? How to declare a pointer variable?
2) What are the steps to be followed to use pointers? How to access the value of a variable using pointer?
3) What is a NULL pointer? What are the various memory allocation techniques?
4) What is static memory allocation? What are the disadvantages of static memory allocation?
5) What is dynamic memory allocation?
6) What are the various memory management functions in C?
7) What is the purpose of using malloc?
8) What will happen if the following program segment is executed?

```
int *ptr;
ptr = (int *) malloc (10);
```

9) What is the purpose of using calloc? What does this function return?
10) What will happen if the following program segment is executed?

```
int *ptr;
ptr = (int *) calloc (5, sizeof(int));
```

11) Write a program to find maximum of n numbers using dynamic arrays
12) What are the difference between malloc() and calloc()?
13) What is the purpose of using realloc? What does this function return?
14) What is the purpose of using free()?
15) Pointers can be dangerous: Justify your answer?
16) What is pointer to a pointer?
17) Write a macro to allocate memory for one or more items of any data type
18) How to create a 2-dimensional array dynamically?
19) Write a function to display 2-dimensional array created dynamically
20) Write a function to read elements into 2-d array created dynamically

3.36 💻 Pointers

21) Write a function to initialize allocated memory locations to 0 using malloc()

22) Write a function to add two matrices that are created dynamically

23) Write a function to multiply two matrices created dynamically

24) Write a program to find transpose of a matrix using dynamically allocated arrays

25) Write a macro to allocate memory for one or more items of any data type using calloc

26) Write a macro to re-allocate memory for one or more items of any data type

Chapter 4: Strings

What are we studying in this chapter?

- **Basic terminology**
- **Storing**
- **Pattern matching algorithms**
- **Programming examples**

4.1 Basic terminology

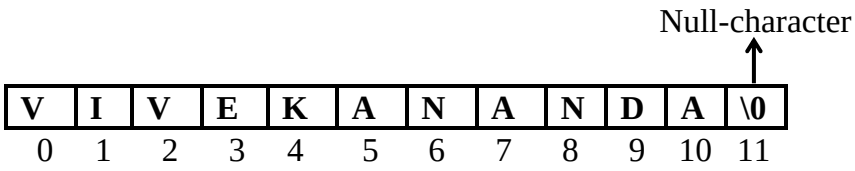
Each programming language contains a set of characters:

- **Letters:** A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
- **Digits:** 0 1 2 3 4 5 6 7 8 9
- **Special characters:** + 1 * / ( ) [ ] { } # $ and so on.

All the above character sets are used to communicate with the computer. Now, let us see "What is a string?"

Definition: A string is a sequence of characters that are obtained by concatenating the various characters in a character set. In C language, a sequence of characters enclosed within double quotes is called a string. In C, string is implemented as an array of characters. A string constant in C is always terminated by a NULL character. A null-terminated string is the only type of string defined by C language.

For example, the string **"VIVEKANANDA"** is stored in memory as shown below:



Note that array starts from zero and each location holds one character starting from index 0 to 10. The string always ends with NULL character denoted by '\0'. The string can be displayed on the screen using the following program:

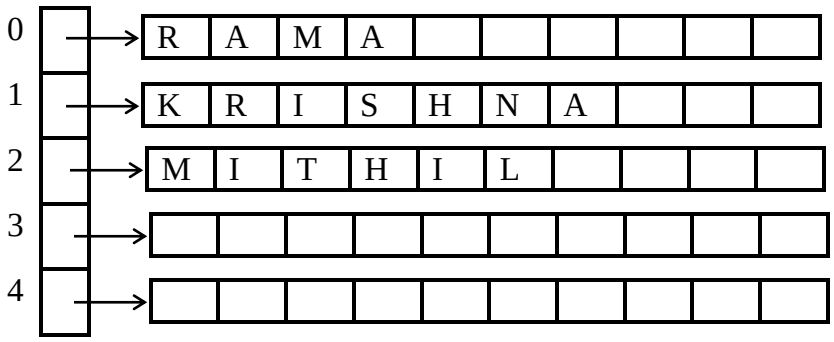
```
#include <stdio.h>
void main()
{
        printf("VIVEKANANDA\n");
}
```

4.2 Storing strings

In this section, let us see "How strings are stored in memory?" The strings are stored in three types of structures:

- Fixed length storage structure
- Variable length structure
- Linked storage structure

- **Fixed length structures:** In this storage structure, each line of text to be manipulated is viewed as a record where all records have same length and same number of characters. Assuming maximum of 15 characters per record, the three strings "RAMA", "KRISHNA", "MITHIL" are stored using fixed length structures as shown below:

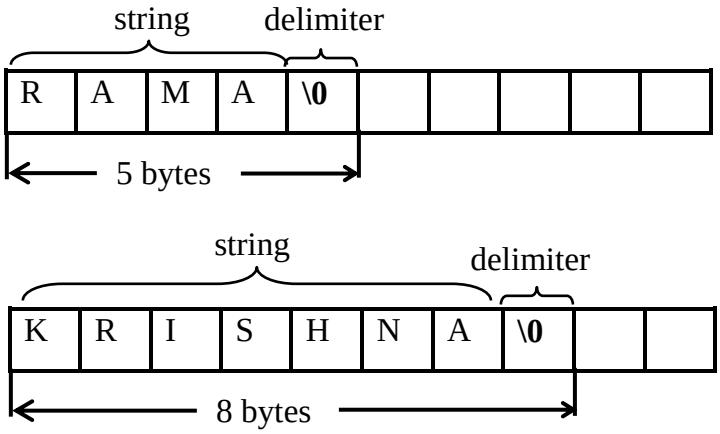


The above records are accessed pointers. Some of the disadvantages using this approach are shown below:

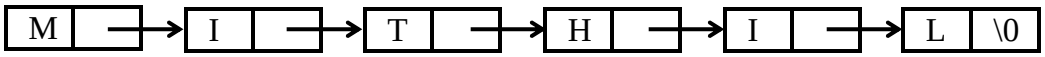
1) Time is wasted in reading the entire record even if more spaces are present
2) Certain records may require more space than available to store a string
3) If the length reserved for string is too small, it is not possible to store the larger data. (see the above figure).
4) If the length reserved for string is too large, too much memory is wasted.
5) Once the string is defined, the length of a string cannot be changed. It is "fixed" for the duration of program execution.

- **Variable length storage structure:** As the name implies, variable-length strings don't have a pre-defined length. In this string, neither the precise length nor maximum length is known at creation time. The storage structure for a string can expand or shrink to accommodate any size of data. But, there should be a mechanism to indicate the end of the data. In a variable-length string, the string ends with a delimiter such as $. In C language, the string ends with NULL (denoted by \0) character.

For example, the strings "**RAMA**"and "**KRISHNA**" are stored using in C language as shown below:



- **Linked storage structure:** In most of the word processing applications, the strings are represented using linked lists. Using linked lists (discussed in chapter 8 and 9), inserting/deleting a character/word is much easier. The string "MITHIL" can be represented using linked list as shown below:



4.3 String operations

Now, let us see "What are operations that can be performed on strings?" The normal operations that can be performed on strings are:

- Substring
- Indexing
- Concatenation
- Length

- **Substring:** Substring is a string obtained by extracting a part of a given string given the position and length of the substring. So, accessing a substring from a given string requires three pieces of information:
  1) The given string
  2) The position from where the string has to be extracted
  3) The number of characters to be extracted.

  For example, SUBSTRING ("TO BE OR NOT TO BE", 4, 5) = "BE OR"

- **Indexing:** The process of finding the position of *pattern* string in a given text *t* is called *indexing*. It is also called *pattern matching*. If the *paatern* string is present in text *t*, the position of the first occurrence of the *pattern* string is returned otherwise, 0 is returned.

  Let, text *t* = "RAMA IS THE KING OF AYODHYA", the pattern string *pattern* is "KING". Then,

  INDEX(t, pattern) = 13

- **Concatenation:** The process of appending the second string to the end of first string is called concatenation. We can denote the concatenation symbol by "+".

  For example, let first string is s1="SEETA" and the second string s2 = " RAMA". Then,

  s1+ s2 = "SEETA RAMA"

- **Length:** The number of characters in a string is called length of the string. For example, Length("RAMA") = 4

4.4 String handling functions in C

Now, let us see "What is the need for string handling functions?" String is not a standard type in C, but it can be treated as a derived data type. So, we cannot manipulate the strings directly using C operators. For this reason, C provides a rich set of string handling functions. Now, let us see "What are the various operations that can be performed on strings?" Some of the operations that can be performed on strings are shown below:

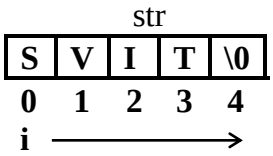
String functions	Description of each function
strlen(str)	Returns length of the string ***str***
strcpy(dest, src)	Copies the source string ***src*** to destination string ***dest***
strcat(str1,str2)	Append string ***str2*** to string ***str1***
strcmp(str1,str2)	Compare two strings ***str1*** and ***str2***
strrev(str)	Reverse the contents of string stored in *str*

4.4.1 strlen(str) – String Length

Syntax: It is defined in header file "string.h". The syntax to use strlen() is shown below:

```
int strlen (char str[]);
```

Working: This function returns the length of the string *str* i.e., it counts all the characters up to '\0' except '\0'. So, an empty string has length zero. Now, let us see "How to write user defined function?" Consider the string "SVIT"



To start with, the index variable ***i*** is zero. Now, keep incrementing value of index variable ***i*** as long as str[i] is not equal to '\0' using the following statement:

```
i = 0;
while (str[i] != '\0')
        i++;
```

Once control comes out the loop, the index variable ***i*** gives the length of the string. The complete C function is shown below:

Example 4.1: Function returning the string length

```
int my_strlen(char str[])
{
        int  i = 0;

        while (str[i] != '\0') i++;

        return   i;
}
```

4.4.2 strcpy(dest, src) – string copy

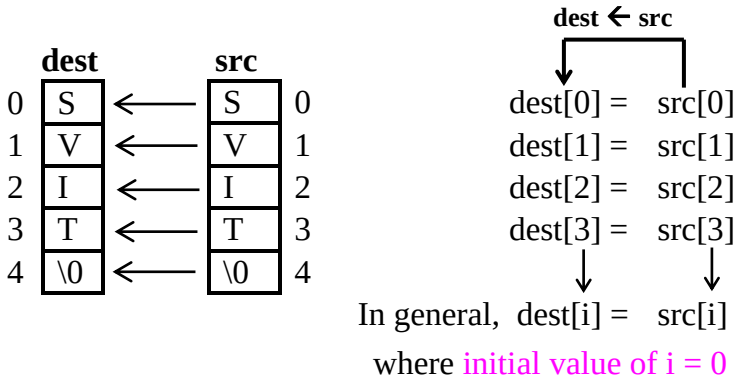
Syntax: It is defined in header file "string.h". The syntax to use strcpy() is shown below:

```
strcpy (char dest[], char src[]);
```

Working: The function ***strpcy*** copies the contents of source string **src** to destination string **dest** including '\0'. So, the size of destination string **dest** should be greater or equal to the size of source string **src**. After copying, the original contents of destination string **dest** are lost.

Now, let us see "How to implement strcpy() function?"

Design: Copy each character of source string **src** to destination string **dest** as long as character is not **'\0'.** This is pictorially represented as shown below:



Observe from above figure that as long as **src[i] != '\0'** it is required to copy **src[i]** to **dest[i]** and increment **i** after copying each character. The partial code can be written as shown below:

```
i = 0;
while (src[i] != '\0') /* Copy as long as '\0' is not encountered */
{
        dest[i] = src[i];
        i++;
}
```

It is the responsibility of the programmer to attach null character '\0' at the end of the string. The equivalent code for this is shown below:

```
dest[i] = '\0';
```

So, the final function can be written as shown below:

Example 4.2: Function to copy string **src** to string **dest** is shown below:

```
void my_strcpy(char dest[], char src[])
{
        int     i = 0;

        while (src[i] != '\0')          /* Copy the string  till we get NULL character */
        {
                dest[i] = src[i];
                i++;
        }

        dest[i] = '\0';                 /* Attach null character at the end */
}
```

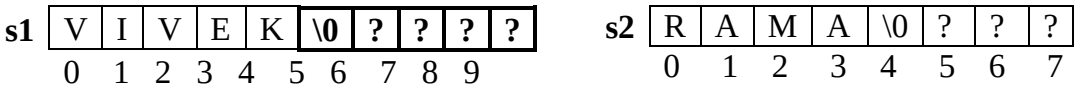
4.4.3 strcat(s1, s2) – string concatenate

Syntax: It is defined in "string.h" as: strcat (**char** s1[], **char** s2[]);
where

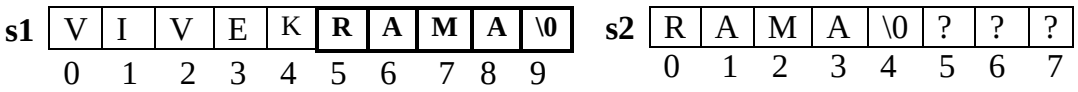
- **s1** is the first string
- **s2** is the second string

Working: The function copies all the characters of string **s2** to the end of string **s1**. The delimiter of string **s1** is replaced by the first character of **s2.** Only condition is that the size of **s1** should be sufficiently large enough to store a string whose length is s1 + s2.

For example, the memory representation for the variables **s1** and **s2** before executing ***strcat*** is shown below:



After **strcat(s1, s2);**



Now, let us see "How to implement strcat() function?" Let us design our own function ***strcat*** and call this function as ***my_strcat***.

Design: This is very simple and straightforward approach. Given two strings **s1** and **s2** perform the following activities:

- Obtain the position of the null character of **str1.** This can be done using the following statements:

```
i = 0;
while (str1[i] != '\0')   i++;
```

- Copy the second string to the end of the first string

```
j = 0;
while ( str2[j] != '\0')
{
        str1[i++] = str2[j++];
}
```

- Attach null character at the end using the statements:

```
str1[i++] = '\0';
```

So, the complete C function to implement ***my_strcat*** is below:

Example 4.3: C function to implement ***my_strcat*** using arrays

```
void my_strcat(char str1[], char str2[])
{
        int  i, j;

        i = 0;

        /* obtain position of null character of str1 */
        while (str1[i] != '\0')   i++;

        /* Copy second string to destination string */
        j = 0;
        while ( str2[j] != '\0')
        {
                str1[i++] = str2[j++];
        }
        str1[i++] = '\0'; // attach '\0'
}
```

```
while ( (str1[i++] = str2[j++]) != '\0')
        ;
```

4.4.4 strcmp(s1, s2) – string compare

Working: This function is used to compare two strings. The comparison starts with first character of each string. The comparison continues till the corresponding characters differ or until the end of the character is reached. The following values are returned after comparison:

- If the two strings are equal, the function returns 0
- If **s1** is greater than **s2**, a positive value is returned
- If **s1** is less than **s2**, the function returns a negative value.

Now, let us see "How to implement ***strcmp*** without using built in function?"

Design: Consider the following figure that shows two strings *s1* and *s2*.

	s1		s2	
0	S	==	S	0
1	V	==	V	1
2	I	==	I	2
3	T	==	T	3
4	\0	==	\0	4

s1[0] == s2[0]
s1[1] == s2[1]
s1[2] == s2[2]
s1[3] == s2[3]
↓ ↓
In general, s1[i] == s2[i]
where initial value of i = 0

Given the two strings **s1** and **s2,** the comparison starts with first character of each string and continues as long as they are equal and increment the index *i* by 1 to point to the next character. During this process, if end of the character is encountered, with respect to **s1,** control comes out of the loop. The equivalent statement for this can be written as shown below:

```
i = 0;
while (s1[i] == s2[i])
{
        if (s1[i] == '\0') break;      / * Check if it is end of string */

        i++;                           /* Point to next character of each string */
}
```

The control comes out of the loop if corresponding two characters differ or **null character** '\0' is encountered. Since the characters are represented by numerical values (ASCII), let us take the difference of corresponding characters which results in zero, positive or negative numbers using the following statement:

```
return s1[i] – s2[i];
```

The complete function is shown below:

Example 4.4: C function to implement *my_strcmp*

```
int my_strcmp(char  s1[], char  s2[])
{
        int  i;

        i = 0;
        while (s1[i] == s2[i])                  /* As long as two strings are equal */
        {
                if (s1[i] == '\0') break;       /* Check if end of string */

                i++;                            /* Point to next character of each string */
        }

        return s1[i] – s2[i];                   /* Return the difference between chars */
}
```

The above function returns one of the following values:

- zero if **s1** = **s2**
- positive if **s1** > **s2**
- negative if **s1** < **s2**

4.4.5 strrev(str) – string reverse

In this section, we see the built in function strrev() and we write one user defined function without using any of the built-in functions.

Syntax: It is defined in "string.h".

void strrev (**char** str[])

where

- **str** is the given string

Working: This function reverses all characters in the string **str** except the terminating NULL character '\0'. The original string is lost. Now, let us see "How to design user defined function for string reversal?"

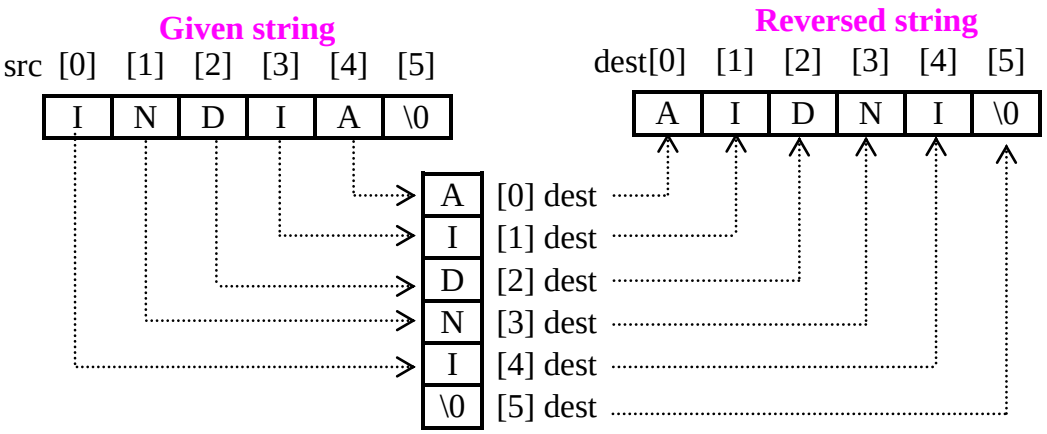
Step 1: Identify parameters to function: We have to reverse the given string say *src* and assume the reversed string is in *dest*. So,

parameters are : **char src[], char dest[]**

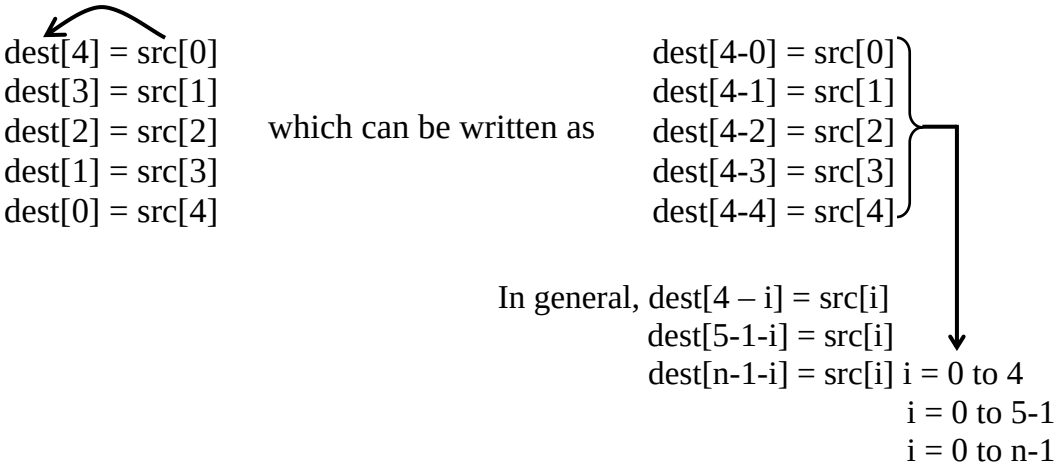
Step 2: Return type : We are not returning any value. So,

return type : **void**

Step 3: Designing function body: The source string characters have to be accessed and should be copied in reverse order in the destination string. This can be done as shown below:



Copying of characters from *src* to *dest* can be performed as shown below:



The above statement can be written as shown below:

```
for ( i = 0; i < n; i++)
{
        dest[n – 1 – i] = src [i];
}
```

where **n** is the string length of given string. Thus, in general the string can be reversed using the following code:

```
n = strlen(src);                /* Compute the length of the string */

for (i = 0; i < n; i++)         /* Reverse the given string */
{
        dest[n-1-i] = src[i];
}

dest[n] = '\0';                 /* Null terminate the string */
```

Note: Other than parameters *src* and *dest* other variables used should be declared as local variables inside the function as shown below:

n, i : declare as int

The complete function is shown below:

Example 4.5: C function that shows implementation of strrev()

```
void my_strrev (char src[],   char dest[])
{
        int  i;          /* Used as index to read characters from src  */
        int  n;          /* Holds the string length */
```

```
        n = strlen(src);                /* Compute the length of the string */

        for (i = 0; i < n; i++)         /* Reverse the given string */
        {
                dest[n-1-i] = src[i];
        }

        dest[n] = '\0';                 /* NULL terminate the string */
}
```

4.5 Pattern matching algorithms

First, let us see "What is pattern matching? What are different pattern matching techniques?"

Definition: Given a string called pattern *p* with *n* characters and another string called *text* with *m* characters where $n \leq m$. It is required to search for the pattern string *p* in the text string *t*. If search is successful return the position of the first occurrence of pattern string *p* in the text string *t*. Otherwise, return -1. This process of searching for a pattern string in a given text string is called *pattern matching.*

The pattern matching can be achieved using the following methods:

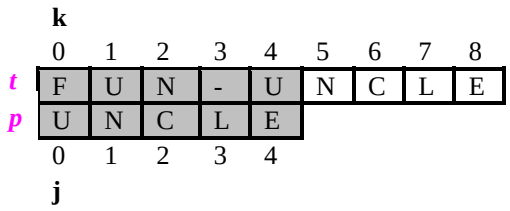
- Brute force pattern matching algorithm
- Checking end indices first and proceed using brute force method
- Using Finite State Machine (Deterministic finite automaton or DFA) called pattern matching table
- Knuth Morris Pratt (KMP) pattern matching algorithm

4.5.1 Brute force pattern matching algorithm

Now, let us see "What is brute force pattern matching algorithm?" In this pattern matching problem, we align *pattern* string to the beginning of the *text* string. If there is no match, we shift the pattern string towards right by one position and the procedure is repeated till we get the end of the *text* string. During this procedure, if pattern string is present, we return the position of pattern string in the text string.

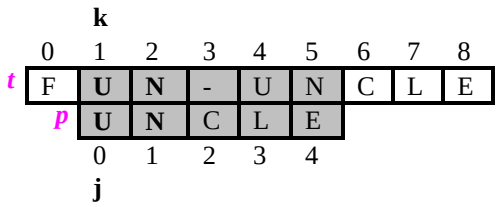
Now, let us see how to "Design the algorithm for pattern matching using brute force method"

Design: This procedure can be interpreted as sliding a *template* which contains the *pattern string* and slide over *text string* as long as the string shown in shaded region matches. The sequence of steps to be followed while searching for a *pattern* string "UNCLE" in the *text* string "FUN-UNCLE" is shown below:



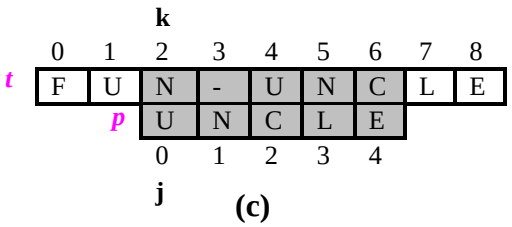
(a)

Note: The characters 'F' and 'U' are compared and they are not equal. So, slide the pattern string towards right as shown below:



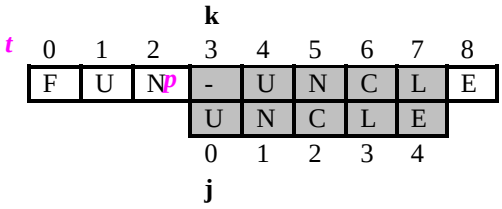
(b)

Note: The characters 'U' and 'U', 'N' and 'N' are compared and found equal. But, there is a mismatch when '–' and 'C' are compared. So, slide the pattern string towards right as shown below:



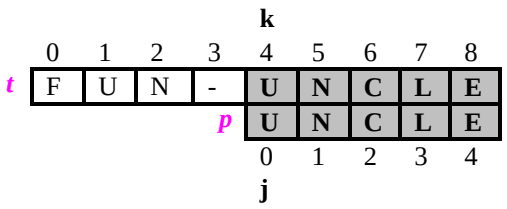
(c)

Note: The characters 'N' and 'U' are compared and they are not equal. So, slide the pattern string towards right as shown below:



(d)

Note: The characters '–' and 'U' are compared and they are not equal. So, slide the pattern string towards right as shown below:



(e)

Note: The pattern string found and position of *k* has to be returned

In figure (e), each character of pattern string has to be compared with corresponding character in the text string. If there is a mismatch we return the position from where the pattern string is matched. Otherwise, we return -1 indicating there is no match. The code can be written as shown below:

The pattern string can be compared with text string from position *k* onwards using the following code:

Example 4.6: C function to search for pattern in text string from specific position

```
int strcmp_from_spcefic_pos(char p[], char t[], int k)
{
        int n, i, j;

        n = strlen(p);
        for (j = 0, i = k; j < n; j++)   // search for pattern from position k
        {
                if (t[i] == p[j])
                {
                        i++;
                        continue;       // partial pattern found and continue
                }
                break;                  // pattern not present from position k
        }

        if (j == n) return k;           // Pattern found

        return -1;                      // Pattern not found
}
```

Observe from figures (a) through (e) that "sliding the pattern string towards right is nothing but incrementing the index *k* of the text string by 1". Also observe that, the index value *k* = 4 is the last position in *text* string that can match a *pattern* string. Beyond this position, there are not enough number of characters in *text* string to compare with *pattern* string. So, *k* range from 0 to 4. This can be written as shown below:

k = 0 to 4
k = 0 to 9 – 5 (Expressing 4 in terms of length of text and pattern string)

In general, k = 0 to m – n where *m* is length(t) and *n* is length(p). Now, the code for pattern search can be written as shown below:

```
m = strlen(t);
n = strlen(p);
for (k = 0; k <= m – n; k++)     // slide the pattern towards right by 1 position
{
        Insert example 4.6 string comparison from specified position
}

return -1;                              // pattern string not found in text string
```

Now, the complete function can be written as shown below:

Example 4.7: Function to search for pattern string in the text string

```
int pattern_match(char p[], char t[])
{
        int m, n, i, j, k, flag;

        m = strlen(t);
        n = strlen(p);

        for (k = 0; k <= m - n; k++)          // shift the pattern towards right
        {
                flag = strcmp_from_spcefic_pos (p, t, k);

                if (flag != -1) return k;     // pattern found at position k
        }

        return -1;                            // Pattern not found
}
```

Now, the complete program to read a text string *t*, pattern string *p* and search for the pattern string in text string can be written as shown below:

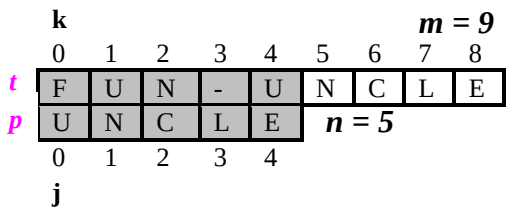
Example 4.8: C program to search for pattern string in the text string

```
#include <stdio.h>
#include <string.h>

// Include: Example 4.6: Function to search for pattern p in text string t at position k
// Include: Example 4.7: Function to search for pattern p in text string

void main()
{
        char t[20], p[10];
        int  pos;

        printf("Enter the text string : ");     gets(t);
        printf("Enter pattern string  : ");     gets(p);

        pos = pattern_match(p, t);

        if (pos == -1)
                printf("Pattern string not found\n");
        else
                printf("Pattern string found at pos = %d\n", pos);
}
```

4.5.2 Brute force pattern matching algorithm by checking end indices first

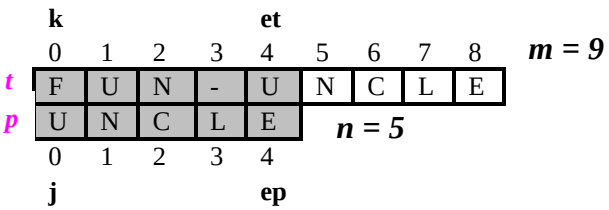
Now, let us see "What is brute force pattern matching algorithm by checking end indices first?" In this pattern matching problem, we align *pattern* string to the beginning of the *text* string. First we compare the last character of the pattern string with the corresponding character in the *text* string. If there is a match, we proceed to match the remaining characters of the pattern string with that of *text* string using the brute force technique from the beginning of the pattern string.

Design: First, align the pattern string to the beginning of the text string as shown below:



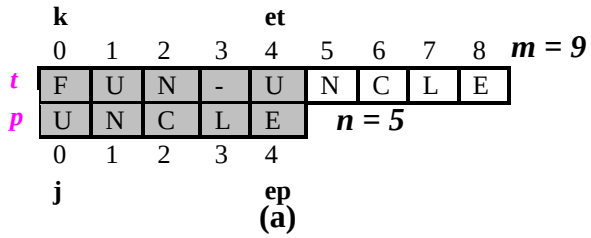
- *k* is the position in text string from where the comparison starts
- *j* is the positon in the pattern string from where the comparison starts.

Apart from the index variables, we require the end position in the text string and end position in the pattern string. Now, the pattern string and text string along with various index values are shown below:



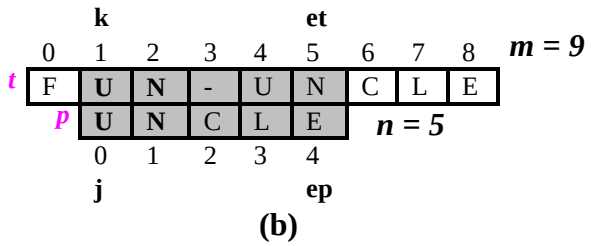
So, initial values are:
m = strlen(t);
n = strlen(p);
et = ep = n – 1;
k = 0

The sequence of steps to be followed while searching for a *pattern* string "UNCLE" in the *text* string "FUN-UNCLE" is shown below:



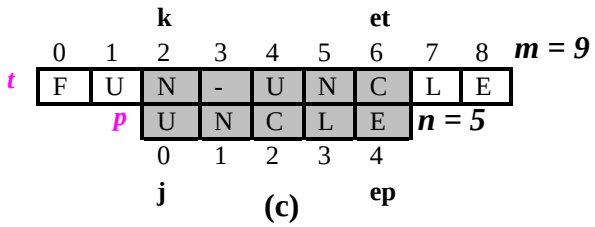
(a)

Note: The characters 'U' and 'E' at positon 4 in text string and at positon 4 in pattern string are not equal. So, slide the pattern string towards right as shown below:



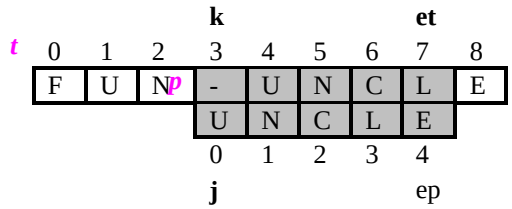
(b)

Note: The characters 'N' and 'E' at positon 5 in text string and at positon 4 in pattern string are not equal. So, slide the pattern string towards right as shown below:



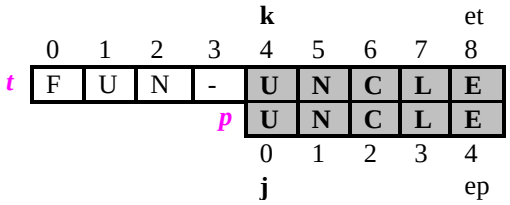
(c)

Note: The characters 'C' and 'E' at positon 6 in text string and at positon 4 in pattern string are not equal. So, slide the pattern string towards right as shown below:



(d)

Note: The characters 'L' and 'E' at positon 7 in text string and at positon 4 in pattern string are not equal. So, slide the pattern string towards right as shown below:



(e)

Note: The pattern string found and position of *k* has to be returned

Observe from figures (a) through (e) that "sliding the pattern string towards right is nothing but incrementing the index values *k* and *et* of the text string by 1".

Also observe that, the index value *et* = 8 is the last position in *text* string that can match a *pattern* string. Beyond this position, there are not enough number of characters in *text* string to compare with *pattern* string. So, *final value of et must be less than the length of the text string*. That is,

et < m;

and each time *et* and *k* are incremented by 1. Now, the code for pattern search can be written as shown below:

```
m = strlen(t);
n = strlen(p);
et = ep = n – 1;
k = 0
```

```
        for (k = 0; et < m; k++, et++)
        {
                if (t[et] != p[ep]) continue;   // compare last character of pattern with
                                                // corresponding character in text string
                Insert example 4.6 string comparison from position k
        }

        return -1;                              // pattern string not found in text string
```

Now, the complete function can be written as shown below:

Example 4.9: Function to search for pattern string in the text string

```
int pattern_match(char p[], char t[])
{
        int m, n, flag, i, j, k, et, ep;

        m = strlen(t);
        n = strlen(p);
        et = ep = n – 1;                        // To compare last character of pattern

        for (k = 0; et < m; k++, et++)          // move the pattern string towards right
        {
                if (t[et] != p[ep]) continue;   // compare last character of pattern with
                                                // corresponding character in text string
                flag = strcmp_from_spcefic_pos (p, t, k);

                if (flag != -1) return k;       // pattern found at position k
        }

        return -1;                              // pattern string not found in text string
}
```

Now, the complete program can be written as shown below:

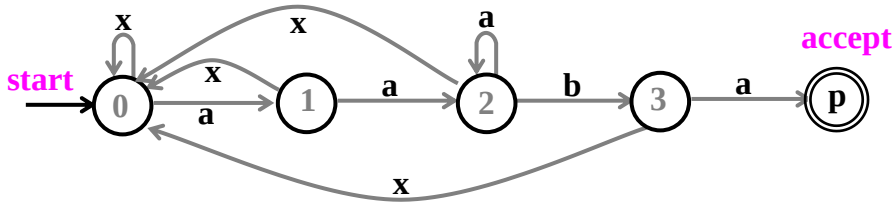
Example 4.10: C program to search for pattern string in the text string

```
#include <stdio.h>
#include <string.h>

// Include: Example 4.6: Function to search for pattern from the specified position

// Include: Example 4.9: Function to search for pattern p in text string t
```

```
void main()
{
        char t[20], p[10];
        int   pos;

        printf("Enter the text string : ");      gets(t);
        printf("Enter pattern string  : ");      gets(p);

        pos = pattern_match(p, t);

        if (pos == -1)
                printf("Pattern string not found\n");
        else
                printf("Pattern string found at pos = %d\n", pos);
}
```

4.5.3 Pattern matching algorithm (Pattern matching table)

In this section, we write a more efficient program with the help of pattern matching table. The pattern matching table can be easily created using the following finite automaton.



Note: In the above diagram, x represent any other scanned symbol.

The above finite automaton which accepts any string consisting of the substring "aaba". To accept any specific pattern we have to construct a finite automaton and then we have to write the program.

State 0: If the scanned character is 'a' go to state 1. Otherwise be in state 0
State 1: If the scanned character is 'a' go to state 2. Otherwise go to state 0
State 2: If the scanned character is 'a' be in state 2. If the scanned character is 'b' goto state 3. Otherwise goto state 0
State 3: If the scanned character is 'a' accept the string and return its position. Otherwise goto state 0.

All the above cases are included inside the switch statement. The complete program can be written as shown below:

Example 4.10: Pattern matching program using pattern matching graph.

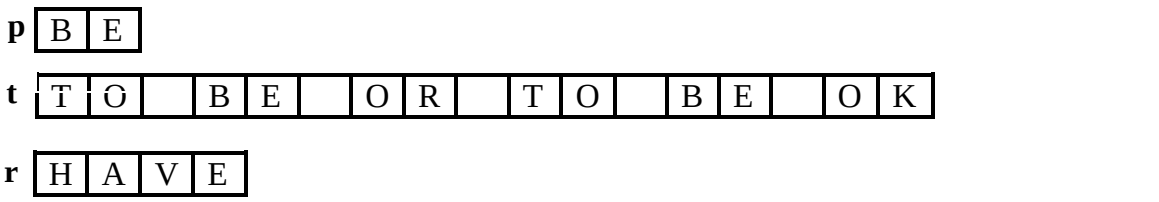
```
#include <stdio.h>
#include <string.h>

void main()
{
        char text[20];
        int flag = 0;
        int i, state = 0, pattern = 4;

        printf("Enter the text string: ");
        scanf("%s", text);

        for ( i = 0; i < strlen(text) && flag == 0; i++)
        {
                switch (state)
                {
                        case 0: if (text[i] == 'a')
                                        state = 1;
                                else
                                        state = 0;

                                break;

                        case 1:
                                if (text[i] == 'a')
                                        state = 2;
                                else
                                        state = 0;

                                break;

                        case 2:
                                if (text[i] == 'a')
                                        state = 2;
                                else if (text[i] == 'b')
                                        state = 3;
                                else
                                        state = 0;
                                break;
```

```
                    case 3:
                        if (text[i] == 'a')
                        {
                            flag = 1;
                        }
                        else
                        {
                            state = 0;
                        }
                        break;
                } // end switch
        } // end for

        if (flag)
                printf("Pattern string found at: %d " , i - pattern);
        else
                printf("Pattern string not found");
}
```

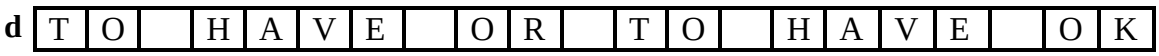
4.5.4 Search and replace a pattern

In this section, let us see "What is the statement problem?" and see how to "Design an algorithm to search for a pattern in a text string and replace with replace string"

Statement problem: Given a text string *t*, pattern string *p* and replace string *r*, it is required to search for every occurrence of pattern string *p* in a text string *t* with replace string *r*. Consider the following strings where *t* is the text string, *p* is the pattern string and *r* is the replace string:

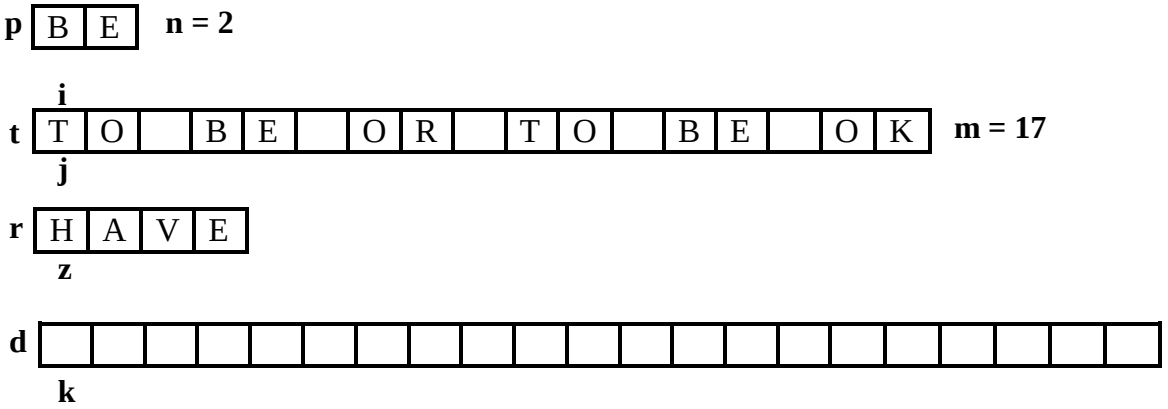


It is required to replace every occurrence of "BE" in the text string *t* with replace string "HAVE" and the resultant string to be obtained is shown below:



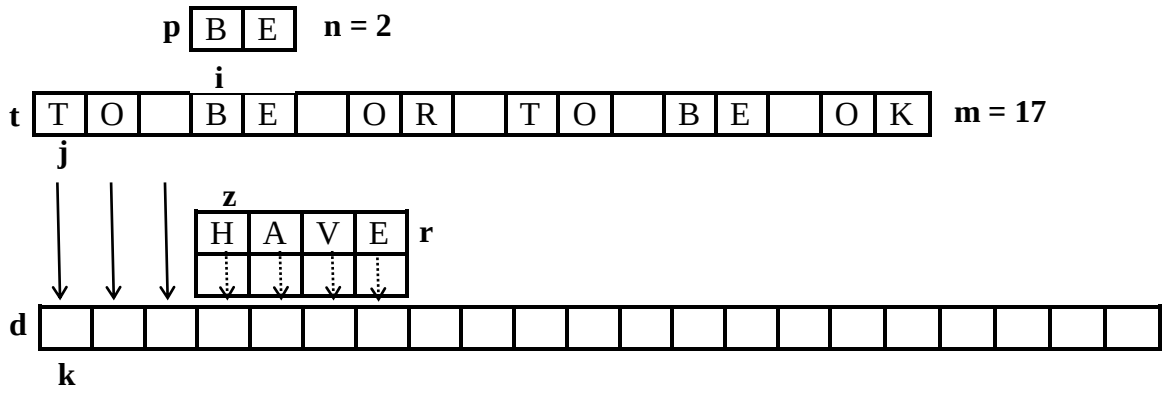
Design: The steps to replace the pattern string in the text string with replace string are shown below:

Step 1: Align the pattern to the beginning of the text string.



i – it is used to shift the pattern string towards right
j – it is used to copy the text string into destination string
k – it is used to access items in the destination string
z – it is used to access the items in the resultant string. It is used only when pattern string has to be replaced.

Step 2: Search for the pattern *p* in the text string and results in the following scenario:



Step3: Copy the characters from text and replace string into destination string: Observe that all the items in the text string *t* from position *j* till (i – 1) have to be copied into destination string *d*. This can be done using the following code:

```
while (j < i) d[k++] = t[j++];          // shown as thick lines in above figure
```

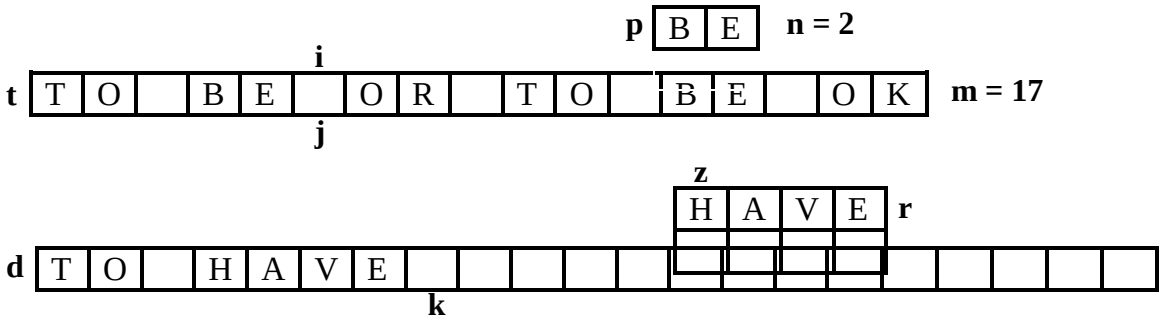
Now, copy all the characters from replace string *r* into destination string using the following code:

```
z = 0;
while (z < strlen(r) ) d[k++] = r[z++];   // shown as dotted lines in above figure
```

update *i* to point to immediate right of pattern string in the text string and assign *i* to *j* using the following code:

```
i += strlen(p);
j = i;
```

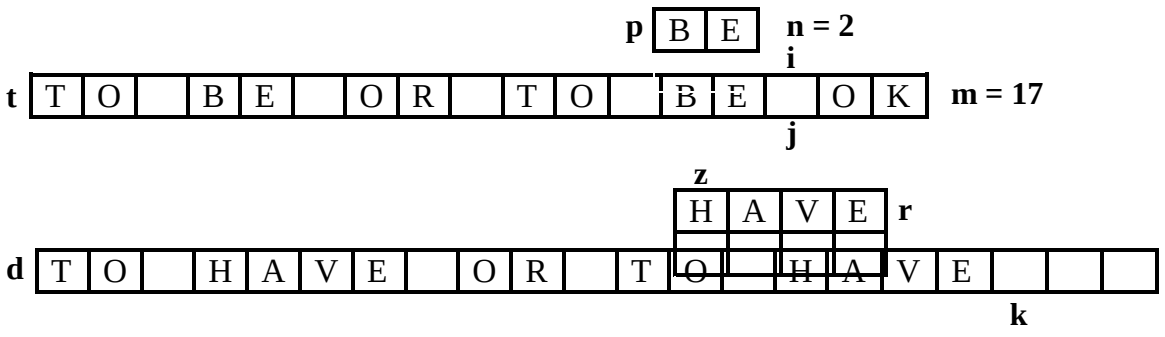
Now the resulting scenario is shown below:



The final code for this step can be written as shown below:

```
if ( strcmp_from_spcefic_pos (p, t, i) == -1)
{
        /* copy test string till the position of pattern string */
        while (j < i) d[k++] = t[j++];

        /* copy replace string into destination */
        z = 0;
        while(z < strlen(r)) d[k++]= r[z++];

        i = i + strlen(p);              // update i to point beyond pattern
        j = i;                          // save the index value
}
```

By executing the above code, we will have the following scenario:



Step 4: Since pattern string *p* is not present in i^{th} position in text string *t*, we increment *i* by 1 so that pattern string is shifted towards right by 1 position. As we increment *i* by 1 we should see that it should not exceed m-n. Now, the code can be written as shown below:

```
while (i <= m - n)
{
        if ( strcmp_from_spcefic_pos (p, t, i) != -1)
        {
                /* copy test string till the position of pattern string */
                while (j < i) d[k++] = t[j++];

                /* copy replace string into destination */
                z = 0;
                while(z < strlen(r)) d[k++]= r[z++];

                i = i + strlen(p);              // update i to point beyond pattern

                j = i;                          // save the index value
        }
        else
                i++;                    // shift the pattern towards right
}
```

Step 5: Copy the remaining characters from string *t* from positon *j* till the end into destination string. The code can be written as shown below:

```
while (j < m) d[k++] = t[j++];
```

Now, the final code can be written as shown below:

Example 4.11: Search for pattern and replace with replace string in a text string

```
void  my_search_replace(char p[], char t[], char r[], char d[])
{
        int   i, j, k, m, n, z;

        i = j = k = 0;

        m = strlen(t);
        n = strlen(p);
```

```
        while (i <= m - n)
        {
                if ( strcmp_from_spcefic_pos (p, t, i) == -1)
                {
                        /* copy test string till the position of pattern string */
                        while (j < i) d[k++] = t[j++];

                        /* copy replace string into destination */
                        z = 0;
                        while(z < strlen(r)) d[k++]= r[z++];

                        i = i + strlen(p);              // update i to point beyond pattern
                        j = i;                          // save the index value
                }
                else
                        i++;                    // shift the pattern towards right
        }
        // copy the remaining characters from string t from position j till the end
        while (j < m) d[k++] = t[j++];

        d[k] = '\0';                            // Null terminate the string
}
```

Now, the complete program in C to search for the pattern in text string *t* and replace with replace string *r* can be written as shown below:

Example 4.12: C program for search and replace

```
#include <stdio.h>
#include <string.h>

// Include: Example 4.6: Function to search for pattern p from the specific position
// Include: Example 4.11: Function my_search_replace()

void main()
{
        char t[40], p[10], r[10], d[40];

        printf("Enter the text string t: ");            gets(t);
        printf("Enter the pattern string p: ");         gets(p);
        printf("Enter the replace string r: ");         gets(r);

        my_search_replace( p, t, r, d);

        printf("The final string after replacing : %s\n", d);
}
```

4.5.4 KMP Algorithm

Before learning this algorithm, let us find out various prefixes and suffixes along with proper prefixes and proper suffixes for a given string. Consider the following table and understand the meaning of prefix and proper prefix as well as suffix and proper suffixes:

String	Prefixes	Proper prefixes	Suffixes	Proper suffixes
"A"	A	\0	A	\0
"AB"	A AB	A	B AB	B
"ABA"	A AB ABA	A AB	A BA ABA	A BA
"ABAB"	A AB ABA ABAB	A AB ABA	B AB BAB ABAB	B AB BAB
"ABABD"	A AB ABA ABAB ABABD	A AB ABA ABAB	D BD ABD BABD ABABD	D BD ABD BABD

Now, let us see how to "Design the algorithm for pattern matching using KMP method?" The two steps to be followed using this technique are:

Step 1: Create a failure function for the pattern to be searched

Step 2: Search for the pattern using the above failure function

Create a failure function for the pattern to be searched: Now, let us obtain the failure function for the pattern "ABABD"

Step 1: Given the pattern string "ABABD" obtain the proper prefixes and proper suffixes for the strings "A", "AB", "ABA", "ABAB" and "ABABD". Refer columns 1, 2 and 3 in the table in the next page.

Step 2: For every substring in step 1, obtain a longest proper prefix which is same as the longest proper suffix.

Step 3: Find the length of the longest proper prefix which is same as the longest proper suffix

Table to find failure function for a given pattern

String	Proper prefixes	Proper suffixes	Longest prefix = Longest suffix	Length	Value
"A"	\0	\0	\0	0	f[0] = 0
"AB"	A	B	\0	0	f[1] = 0
"ABA"	**A** AB	**A** BA	**A**	1	f[2] = 1
"ABAB"	A **AB** ABA	B **AB** BAB	**AB**	2	f[3] = 2
"ABABD"	A AB ABA ABAB	D BD ABD BABD	\0	0	f[4] = 0

So, the failure function for the corresponding pattern string can be written as shown below:

	0	1	2	3	4	
p	A	B	A	B	D	**pattern string**
f	0	0	1	2	0	**failure function**
	0	1	2	3	4	

Now, let us see "How to construct failure function for a given pattern?" This can be done using the following relations:

- Let *i* and *j* are two index variables with 0 and 1 values initially. Let f[0] = 0 where *f* s a failure function.
- If p[i] is same as p[j] then assign i + 1 to f[j]. The code for this case can be written as shown below:

```
if (p[i] == p[j])
{
        f[j] = i + 1
        i++, j++
}
```

- If p[i] is not equal to p[j] and if i is zero, then assign *i* to f[j] and increment *j* by 1. The code for this can be written as shown below:

```
if (i == 0)
{
        f[j] = i
        j++
}
```

- If p[i] is not equal to p[j] and if *i* is not zero, then assign f[i-1] to *i*. The code for this case can be written as shown below:

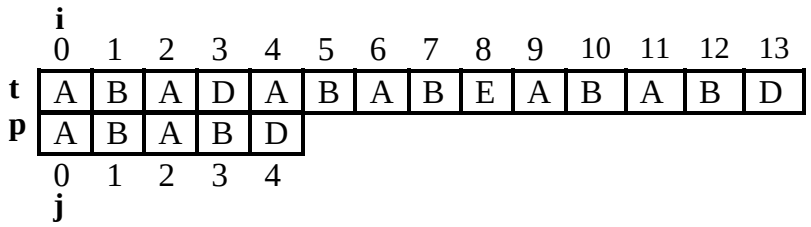
```
i = f[i-1]
```

Now, the complete code for the failure function can be written as shown below:

Example 4.13: Failure function for the pattern string

```
void failure(int f[], char p[])
{
        int i = 0, j = 1;

        f[i] = 0;
        while ( j < strlen(p))
        {
                if (p[i] == p[j])
                        f[j] = i+1, i++, j++;
                else if (i == 0)
                        f[j++] = i;
                else
                        i = f[i-1];
        }
}
```

Search for the pattern using failure function: For searching the pattern string in the text string, first we align patter to the beginning of text string as shown below:



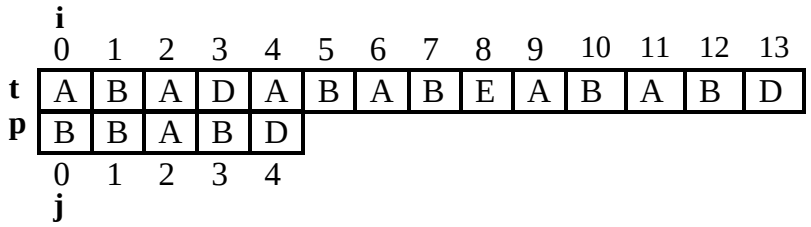
Initialization: Since we have to compare the first character of text string and first character of pattern string we have to initialize *i* to 0 and *j* to 0 as shown below:

```
i = 0
j = 0
```

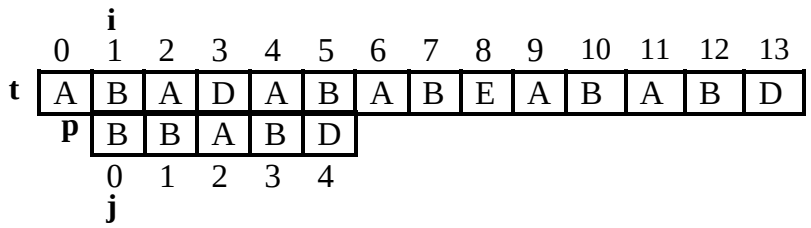
Case 1: When corresponding characters are equal: If t[i] is same as p[j] then we increment *i* by 1 and *j* by 1 so as to compare the next characters. The code for this case can be written as shown below:

```
if (t[i] == p[j])
{
        i++;
        j++;
}
```

Case 2: When corresponding characters are not equal: Consider the following scenario:



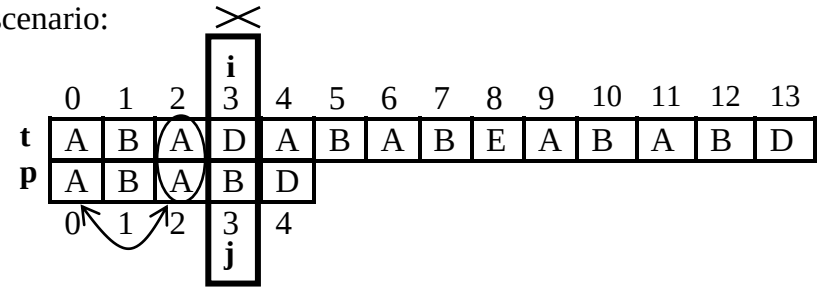
Note that t[i] is not equal to p[j] in the above scenario. So, we have to slide the pattern string towards right as shown below:



Observe that the value of *j* has not been changed. But, the value of *i* is incremented by one. It indicates that incrementing the value of *i* by 1 implies the pattern string is moved towards right by one position. The code corresponding to this can be written as shown below:

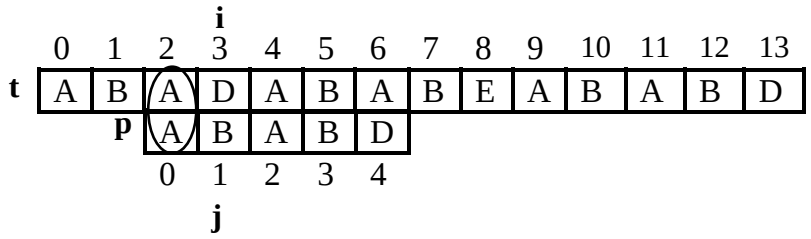
```
if (j == 0) i++;
```

Case 3: When corresponding characters are not equal: Consider the following scenario:



Observe that the A and A enclosed within the oval in the above figure are matched. But, A is the proper prefix and A is the proper suffix (shown using the arrow mark). It

means that the proper prefix A in the pattern and A in the 2nd position of text string are one and the same. So, shift the pattern string towards right so that A in the 2nd position of text string and the proper prefix A in the 0th position of pattern string are aligned as shown below:



Now, there is no need to compare two A's shown in oval shape in the above figure. Observe that the value of *j* should be 1. This can be done very easily by looking at the scenario shown in the first figure of case 3. In that figure mismatch occurs when *j* is 3 (see the figure in the previous page). Now, take the previous character i.e., A in position 2. Now, take the failure value in position 2 in failure function *f*. That is f[2] whose value is 1 has to be copied into *j*. The code for this case can be written as shown below:

```
if (j != 0) j = f [j-1];
```

All the statements in three cases should be repeatedly executed as long as *i* is less than *string length* of text string and *j* is less than *string length of* pattern string. Now, the code can be written as shown below:

```
while ( i < strlen(t) && j < strlen(p))
{
        if (t[i] == p[j)
        {
                i++;
                j++;
        }
        else if (j == 0)
                i++;
        else if (j != 0)
                j = f [j-1];
}
```

Whenever control comes out of the loop, if *j* is equal to the string length of pattern string, there is a match and return the position of the pattern string in the text string. Otherwise, return -1 indicating pattern not found. The code for this case can be written as shown below:

```
        if ( j == strlen(p))
                return i – strlen(p);          // Pattern string found. Return its position
        else
                return -1;                     // Pattern string not found
```

Now, the complete function to search for the pattern string in a text string using KMP method can be written as shown below:

Example 4.14: Pattern match using Knuth, Moris and Pratt method

```
int pattern_match(char p[], char t[], int f[])
{
        int i = 0, j = 0;

        while (i < strlen(t) && j < strlen(p))
        {
                if (t[i] == p[j])
                        i++, j++;       // compare successive characters
                else if (j == 0)
                        i++;            // Move the pattern string towards right by 1
                else
                        j = f[j-1];     // Move the pattern string towards right so that i = j
        }

        if (j == strlen(p))                    // Pattern string found
                return i - strlen(p);
        else                                   // Pattern string not found
                return -1;
}
```

Now, the complete program to search for the pattern string in a text string using KMP algorithm is shown below:

Example 4.15: C Program to search for the pattern in a given text using KMP method

```
#include <stdio.h>
#include <string.h>

// Include: Example 4.13: Failure function for the pattern string
// Include: Example 4.14: Pattern matching using KMP method

void main()
{
```

```
        int i, pos;
        char t[40], p[20];
        int f[20];

        printf("Enter the text string:");
        scanf("%s", t);
        printf("Enter the pattern string:");
        scanf("%s", p);

        failure(f, p);

        pos = pattern_match(p, t, f);

        if (pos == -1)
                printf("Pattern string not found\n");
        else
                printf("Pattern string found at pos: %d", pos);
}
```

Exercises

1) What is a string? How strings are stored in memory?
2) What are operations that can be performed on strings?
3) What is pattern matching? Design an algorithm to search for pattern string *p* in text string *t* from position *i* using straight forward method (brute force method)
4) Design a function to search for the string in a pattern string using pattern matching table.
5) Design the function to search for a pattern string *p* in the text string *t* and replace it with replace string *r*
6) Design a KMP algorithm for pattern matching
7) Design brute force pattern matching algorithm by checking end indices first
8) Design functions to implement following C string functions:
   a) strlen b) strcpy c) strcat d) strrev e)strcmp

Chapter 5: Structures and unions

What are we studying in this chapter?

- Structures and self-referential structures
- Unions
- Polynomials
- Sparse matrices

5.1 Structures

We know that array is collection of same type of data. But, in real world we often encounter a situation where we need to have collection of data of different types. In such situations, we use structures in C. Now, we shall see "*What is a structure*?"

Definition: A *structure* is defined as a collection of data of same/different data types. All data items thus grouped are logically related and can be accessed using variables. Thus, structure can also be defined as a group of variables of same or different data types. The variables that are used to store the data are called *members of the structure* or *fields of the structure*. In C, the structure is identified by the keyword **struct.**

Ex: The structure definition to hold the student information such as *name, roll_number* and *average_marks* can be written as shown below:

```
struct student
{
        char    name[10];
        int     roll_number;
        float   average_marks;
};
```

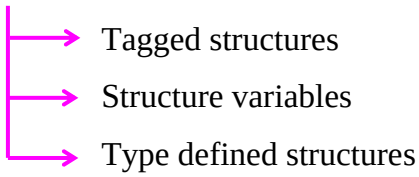
members of structure

Size of each member	Bytes
name[10]	10
roll_number	4
average_marks	8
Total size =	22

- In the above structure, *name, roll_number* and *average_marks* are the fields of the structure. They are also called members of the structure.
- Even though the size of structure is 22 bytes, no space is reserved for the above structure.
- Memory is reserved only if the above definition is associated with variable. That is, once the structures are defined, they have to be declared. Then only, 22 bytes of memory space is reserved.

5.2 Structure declaration

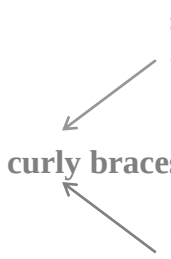
Once we know how to define the structure, the next question is "How to declare a structure?" As variables are declared before they are used in the function, the structures are also should be declared before they are used. A structure can be declared using three different ways as shown below:



5.2.1 Tagged Structure

Now, let us see "What is a tagged structure? Explain with example"

Definition: The structure definition with tag name is called tagged structure. The tag name is the name of the structure. The syntax of tagged structure is shown below:



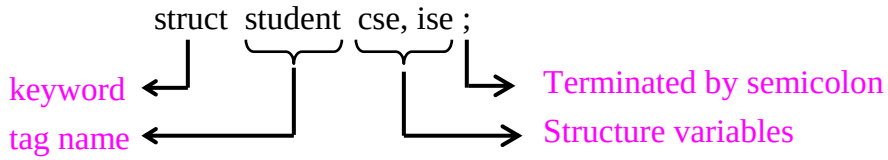

```
struct tag_name
{
        type1    member1;
        type2    member2;
        ……       ……
        ……       ……
} ;
```

Note: semicolon is must at the end

For example, consider the following structure definition:

```
struct student
{
        char   name[10];              10 bytes
        int    roll_number;            4 bytes
        float  average_marks;          8 bytes
};                              Total: 22 bytes
```

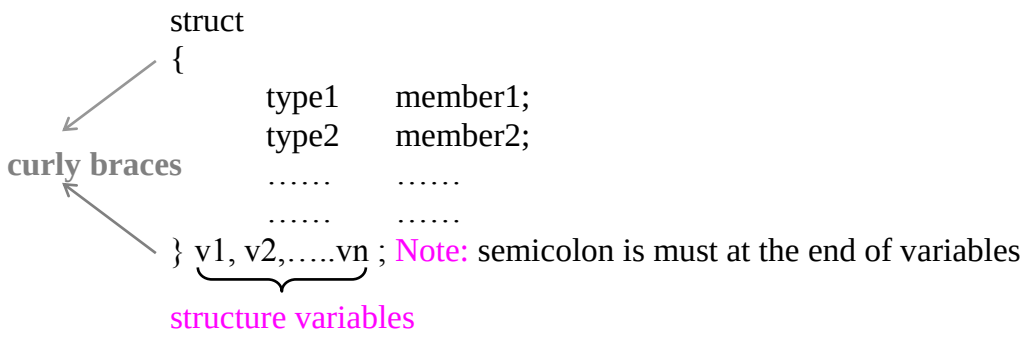
Here, *student* is the tag name. By defining the above structure, memory will not be reserved for any of the members. To allocate the memory for the structure, we have to declare the variable as shown below:



Once the structure variables are declared, the compiler allocates memory for the structure variables. So, 22 bytes are reserved for the variable *cse* and another 22 bytes are reserved for the variable *ise*. The size of the memory allocated is the sum of size of individual members.

5.2.2 Structure variables

Let us see "How to define the structure along with structure variables?" The syntax of structure definition and declaration using structure variables is shown below:



For example, consider the following declaration:

```
struct
{
        char    name[10];               10 bytes
        int     roll_number;            4 bytes
        float   average_marks;          8 bytes
} cse, ise;                     Total:  22 bytes
```

Observe that 22 bytes of memory is allocated for the variable *cse* and another 22 bytes of memory is allocated for the variable *ise*.

Note: We avoid this way of defining and declaring the structure variables because of the following reasons:

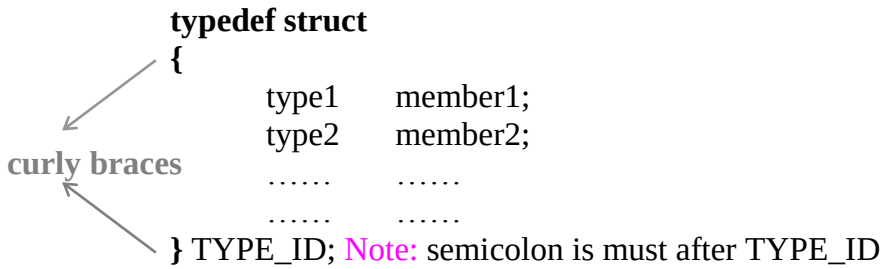
- Without a tag, it is not possible to declare variables in later stages inside the functions.
- It is not possible to use them as parameters in the function, because all the parameters have to be declared.
- We can define them only in the beginning of the program. In such situations, they are treated as global variables. In the structured programming it is better to avoid the usage of global variables.

5.2.3 Type-Defined Structure

Now, let us see "What is a type-defined structure? Explain with example"

Definition: The structure definition associated with keyword **typedef** is called type-defined structure. This is the most powerful way of defining the structure. This can be done using two methods:

Method 1: The syntax of type-defined structure is shown below:



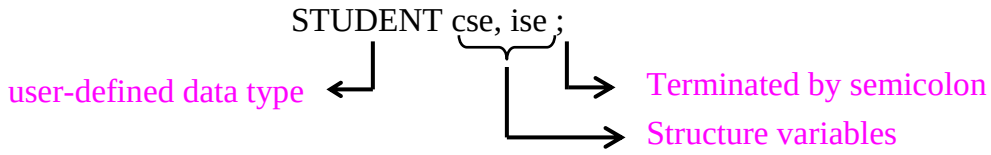
where

- typedef is the keyword added to the beginning of the definition
- struct is the keyword which tells the compiler that a structure is being defined.
- member1, member2,….. are called members of the structure. They are also called fields of the structure.
- The members are declared within curly braces.
- The closing brace must end with type definition name (TYPE_ID in the syntax shown) which in turn ends with semicolon. Note that TYPE_ID is not a variable, instead it is a user-defined data type.

For example, the type-defined structure definition is shown below:

```
typedef struct
{
        char    name[10];
        int     roll_number;
        float   average_marks;
} STUDENT;
```

Since STUDENT is the type created by the user using the keyword **typedef**, it can be called as user-defined data type. From this point onwards we can use **STUDENT** as data type and declare the variables. For example, consider the following declaration:



This statement declares that the variables *cse* and *ise* are variables of type STUDENT.

Method 2: Here, we use the *tag* for the structure and then we obtain the user-defined data type using the keyword **typdef**. For example, consider the structure definition:

```
/* Structure definition */
struct student                  Note: student is the tag name
{
        char    name[10];
        int     roll_number;
        float   average_marks;
};                                      /* No memory is allocated for structure */
```

The user-defined data type can be obtained using the keyword **typedef** as shown below:

```
typedef struct student STUDENT;/* STUDENT is user-defined data type */
```

Using the user-defined data type **STUDENT**, we can declare the variables as shown below:

```
/* Structure declaration */
STUDENT cse, ise;               /* Memory is allocated for the variables */
```

Note: The word student which is written using fully lowercase letters is the tag name of the structure whereas STUDENT which is written using fully capital letters is the user-defined data type.

5.2.4 Structure initialization

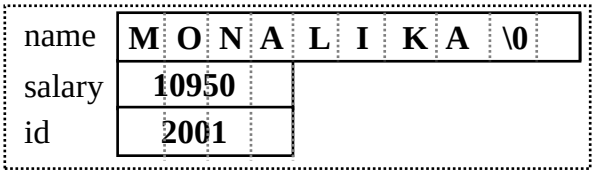
Now, let us see "How are structures initialized?" The structures can be initialized various ways:

Method 1: Specify the initializers within the braces and separated by commas when the variables are declared as shown below:

```
struct employee
{
        char    name[20];
        int     salary;
        int     id;
} a = {"MONALIKA", 10950, 2001};
```

initializers

Memory representation



a

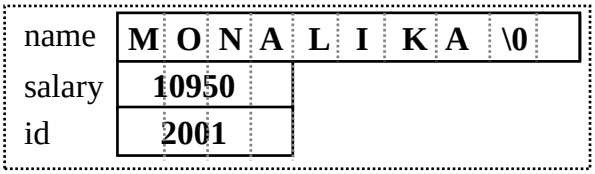
Method 2: Specify the initializers within the braces and separated by commas when the variables are declared as shown below:

```
/* structure definition */
struct employee
{
        char    name[20];
        int     salary;
        int     id;
};
```

Note: The compiler will not reserve memory for structure definition

```
/* structure declaration and initialization */
struct employee a = {"MONALIKA", 10950, 2001};
```

Note: By looking at the declaration for variable *a*, the compiler reserves the space for the variable *a*, whose size is equal to the sizes of the individual data members of the structure. Then, the data is copied into appropriate fields as shown below:



a

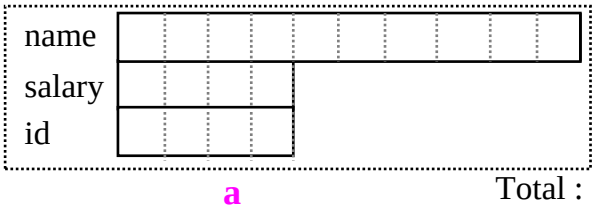
10 bytes	
4 bytes	
4 bytes	
Total :	18 bytes

Example 5.1: Show the memory representation for the following structure declaration:

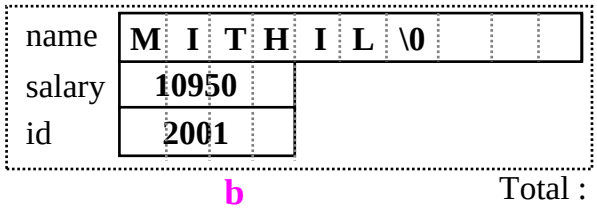
```
struct employee
{
        char   name[20];
        int    salary, id;
} a, b = {"MITHIL", 10950, 2001};
```

Solution: The compiler reserves the space for both variables *a* and *b*. Observe that the variable *a* is not initialized whereas the variable *b* is initialized. The size of each variable is the sum of sizes of individual data members.

Memory representation for structure variable *a*:



Memory representation for structure variable *b*:



Note: Observe that variable *a* is not initialized. But, only the variable *b* is initialized.

Now, let us see some important points to remember when we initialize the structures.

- The members of the structure cannot be initialized in the definition. For example,

```
struct employee
{
        char name[20] = "RAMA";
        int  salary = 20000;
        int  id = 25;
};
```

Note: Initialization within structure as shown results in syntax error

- The initial values are assigned to members of the structure on one-to-one basis. The values are assigned to various members in the order specified from the first member. For example, the following statement:

```
struct employee a = {"RAMA", 10950, 2001};
```

is valid. Here, the string "RAMA" will be assigned to the first member, 10950 is assigned to the second member and 2001 will be assigned to the third member.

- During partial initialization the values are assigned to members in the order specified and the remaining members are initialized with default values. For example, in the initialization statement

  struct employee a = {"RAMA"};

  observe that the string "RAMA" will be assigned to structure member *name*. The other members of the structure namely *salary* and *id* are initialized to zero by default.

- During initialization, the number of initializers should not exceed the number of members. It leads to syntax error. For example, for the following statement

  struct employee a = {"Rama",10950, 2001,10.2};

  the compiler issues a syntax error saying "too many initializers".
- During initialization, there is no way to initialize members in the middle of a structure without initializing the previous members. For example,

  struct employee a = {10950, 2001};

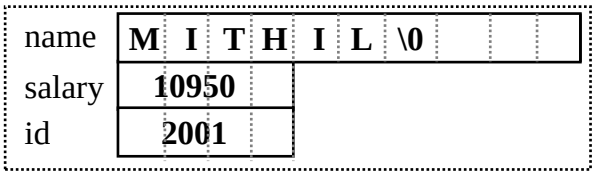
  is invalid. Because, without initializing the first member namely *name*, we are trying to initialize last two members. In this case, the number 10950 will be copied into *name* field, the number 2001 will be copied into *salary* field and the result is unpredictable.

5.2.5 Accessing structures

Now, let us see "How structure members are accessed?" The members of a structure can be accessed by specifying the variable followed by dot operator followed by the name of the member. For example, consider the structure definition and initialization along with memory representation as shown below:

```
//Structure initialization
struct employee
{
        char    name[20];
        int     salary;
        int     id;
} a = {"MITHIL", 10950, 2001};
```

Memory representation



a

The various members can be accessed using the variable *a* as shown below:

- By specifying *a.name* we can access the name "MITHIL".
- By specifying *a.salary* we can access the value 10950
- By specifying *a.id* we can access the value of 2001

Now, the question is "How to display the various members of a structure?" The various values can be accessed and printed as shown below:

Programming statements	Output
printf("%s\n", a.name);	MITHIL
printf("%d\n", a.salary);	10950
printf("%d\n", a.id);	2001

Once we know how to display the members of a structure, let us see "How to read the values for various members of a structure?" We know that format specifications such as %s %d %d are used to read a string, an integer and a float. The same format specifications can be used to read the members of a structure. For example, we can read the name of an employee, the salary and id as shown below:

```
gets(a.name);
scanf("%d", &a.salary);
scanf("%d", &a.id);
```

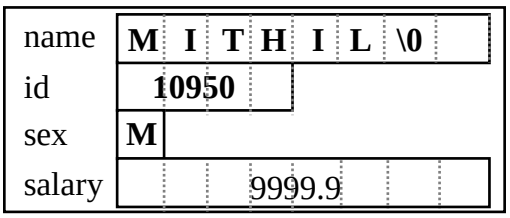
5.2.6 Internal implementation of structures

Now, let us see "What is the size of the structure?"

Definition: The size of a structure is defined as the sum of sizes of individual member of the structure. For example, the structure declaration along with sizes of individual data members is shown below:

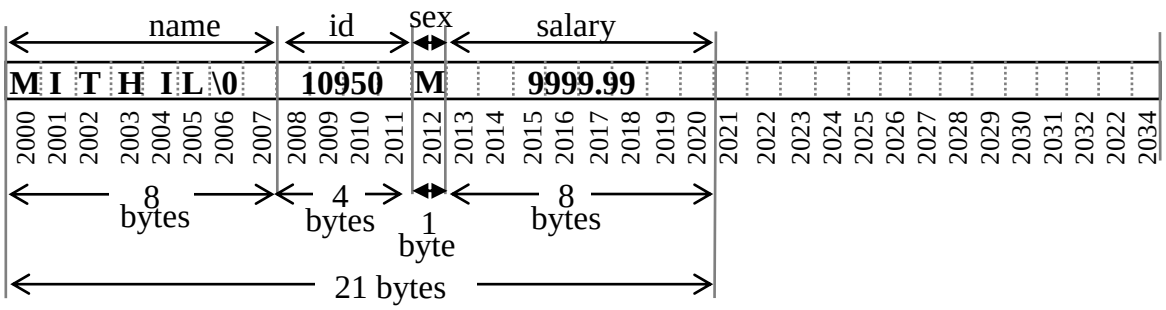
```
// Structure definition
struct employee
{
        char    name[8];
        int     id;
        char    sex;
        double salary;
} a = {"MITHIL",109,'M',9999.9};
```

Pictorial representation



Bytes	
8	
4	
1	
8	
Total :	21

Observe that total size of the structure = 21 bytes. If 2000 is the starting address of the first member, then the starting address of each member depends on size of previous member. The complete memory map along with addresses is shown below:



Observe that the values of the members are stored in increasing address locations in the order specified in the structure definition.

That is, address of *name* < address of *id* < address of *salary* and the address of each member is obtained using the following relation:

address of member = starting address of previous member + size of previous member

So, address of member *id* = starting address of member *name* + size of member *name*
= 2000 + 8
= 2008

Address of member *sex* = starting address of member *id* + size of member *id*
= 2008 + 4
= 2012

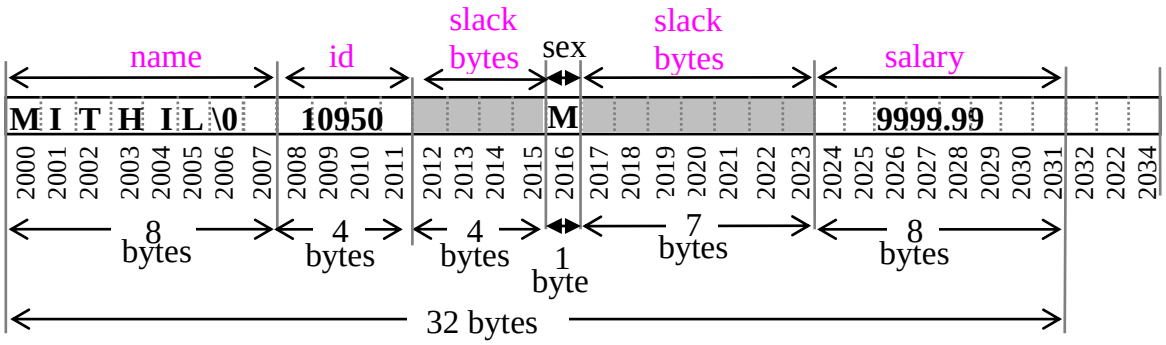
Address of member *salary* = starting address of member *sex* + size of member *sex*
= 2012 + 1
= 2013

Note: Observe that some members have odd addresses and some members have even addresses. For example, members such as *name*, *id* and *sex* have even addresses whereas the member *salary* has odd address. In such situation, microprocessor accesses the data stored in even addresses faster than the data stored in odd addresses. So, it is the responsibility of the compiler to allocate the memory for members such that they have even addresses so that the data can be accessed very fast. This leads to slack bytes.

Now, let us see "What are slack bytes?"

Definition: The optimized compilers will always assign even addresses to the members of the structure so that the data can be accessed very fast. The even addresses may be multiples of 2, 4, 8 or 16. This introduces some unused bytes or holes between boundaries of some members. These unused bytes or holes between boundaries of members of the structure are called slack bytes. The slack bytes do not contain any valid information and are useless and waste the memory. But, it results in faster accessing of data stored in members.

Some optimized compilers assign the addresses such that the addresses of each member will be multiple of the size of largest data type of that structure. For example, if the data types of members of a structure are **char**, **int**, **float** and **double**, then a member whose type is double occupy more space. Assuming sizeof(double) is 8 bytes, then the address of each member is multiple of 8 as shown below:



Observe that the starting address of each member is divisible by 8. But, we can access only the specified number of bytes of a member. The extra bytes that are not used are slack bytes. The shaded region represents the slack bytes.

Note: The size of the structure is 32 bytes which is greater than the sum of sizes of each member. So, when slack bytes are used, the size of a structure is greater than or equal to the sizes of its individual members.

5.2.7 Uses of structures

Now, let us see "What is the use of structures?"

The structures can be used for the following reasons:

- Structures are used to represent more complex data structures

5.12 Structures and unions

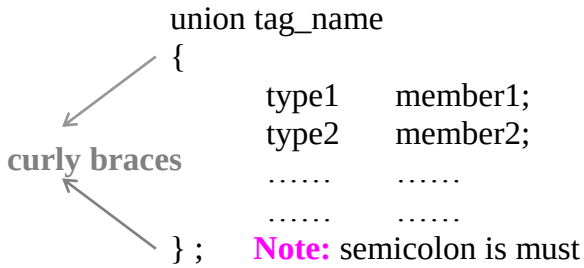
- Related data items of dissimilar data types can be logically grouped under a common name and all the items can be accessed using a common name.
- Can be used to pass arguments so as to minimize the number of function arguments.
- When more than one data has to be returned from the function, then structures can be used.
- Extensively used in applications involving database management
- To make the program more readable.

5.3 Union and its definition

In this section, we shall see another concept called union which is similar to structures. Let us see "What is a union?"

Definition: A *union* is similar to a structure which is also collection of data items of similar or dissimilar data types which are identified by unique names using identifiers. Each identifier is called a field or a member. All the members of the union share the same memory space. Thus, all the identifiers of **union** have the same addresses. At any given point during execution, only one member is active.

Now, let us see "How union is declared and used in C?" The general format (syntax) of a *union* definition is shown below:



Ex: The union definition to hold the various informations such as *integer*, *char* and *double* values can be written as shown below:

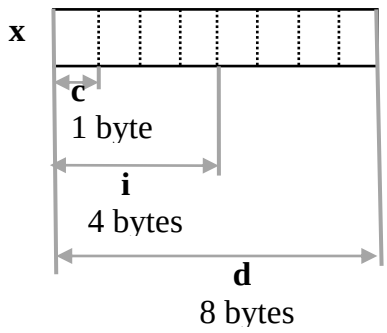
```
// union definition
typedef  union
{
        int          i;
        double     d;
        char        c;
} item;
```

members of union

- Here, *i*, *d* and *c* are the fields of the union. They are also called members of the union.
- No space is reserved for the above union.

Now, consider the following declaration along with memory representation:

```
item    x;
```



- By looking at the above declaration, the compiler will reserve the memory whose size is that of the largest member. Since **double** is the largest data type of the member of the union, sizeof(double) = 8 bytes of memory is reserved for the variable *x*.
- Observe that all the members have the same starting address and hence they share the same allocated space

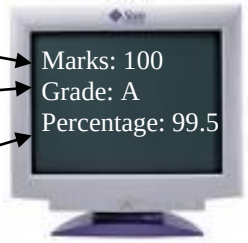
Now, let us consider two programs to show that union and structures behave differently.

Example 5.2: Program to show how union behaves

```
#include <stdio.h>

void main()
{
        typedef union
        {
                int     marks;
                char    grade;
                float   percentage;
        } STUDENT;

        STUDENT x;
```

```
        x.marks = 100;
        printf("Marks : %d\n",x.marks);

        x.grade = 'A';
        printf("Grade : %c\n",x.grade);

        x.percentage = 99.5;
        printf("Percentage: %f\n", x.percentage);
}
```

Output



Observe the following points during execution of the above program:

- After executing the statement x.marks = 100, the member *marks* will hold the value 100 whereas other members should not access the data. If accessed, it will be treated as garbage value.
- After executing the statement x.grade = 'A', the member *grade* will hold the value 'A' whereas other members should not access the data. If accessed, it will be treated as garbage value.
- After executing the statement x.percentage = 99.5, the member *percentage* will hold the value 99.5 whereas other members should not access the data. If accessed, it will be treated as garbage value.

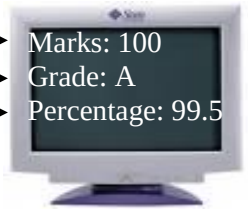
Note: It is observed from the above output that only one member of union can hold a value at a time. It is not possible to access all the members simultaneously. So, the variable of type STUDENT can be treated as *integer* variable or *char* variable or *float* variable. Now, consider the same program with structure.

Example 5.3: Program to show how structure behaves

```
#include <stdio.h>

void main()
{
        typedef struct
        {
                int     marks;
                char    grade;
                float   percentage;
        } STUDENT;
```

```
        STUDENT x;

        x.marks = 100;
        x.grade = 'A';
        x.percentage = 99.5;

        printf("Grade : %c\n",x.grade);
        printf("Marks : %d\n",x.marks);
        printf("Percentage: %f\n", x.percentage);
}
```

Output



Note: It is observed from the above output that the all the members of a structure can hold individual values at a time. It is possible to access all the members of a structure simultaneously. Now to answer the question "What is the difference between a structure and union?"

Structure	Union
1. The keyword *struct* is used to define a structure	1. The keyword *union* is used to define a union.
2. When a variable is associated with a structure, the compiler allocates the memory for each member. *The sizeof structure is greater than or equal to the sum of sizes of its members.* The smaller members may end with unused *slack bytes* (see section 2.4.5)	2. When a variable is associated with a union, the compiler allocates the memory by considering the size of the largest member. So, *size of union is equal to the size of largest member*
3. Altering the value of a member will not affect other members of the structure	3. Altering the value of any of the member will alter other member values.
4. The address of each member will be in ascending order This indicates that memory for each member will start at different *offset* values	4. The address is same for all the members of a union. This indicates that every member begins at *offset* zero.
5. Individual members can be accessed at any time since separate memory is reserved for each member	5. Only one member can be accessed at a time since memory is shared by each member.

5.4 Self-referential structures

Discussed in chapters 8 and 9

5.5 Polynomials with one variable

Now, let us see "What is a polynomial?"

Definition: A polynomial is a mathematical expression consisting of sum of terms where each term is made up of a coefficient multiplied by a variables raised to a power.

Ex: A polynomial consisting of only one variable is shown below:

$$x^4 + 10x^3 + 3x^2 + 1$$

Now, let us consider a polynomial with only one variable and see "How to represent each term of the polynomial?" Each term consists of a co-efficient multiplied by a variable raised to a power. So, each term can be represented by a structure consisting of 2 fields namely:

- cf (representing coefficient)
- px (power of x)

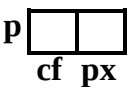
The structure definition for a term of a polynomial can be written as shown below:

```
typedef struct
{
        int     cf;         // used to hold the co-efficient
        int     px;         // used to hold power of x
} POLY;
```

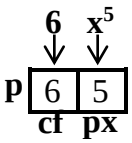
Now, consider the following declaration along with its memory representation:

```
/* declaration */
POLY        p;
```

/* memory representation */



Once the memory is allocated as shown above, the term $\mathbf{6x^5}$ can be stored using the variable *p* as shown below:



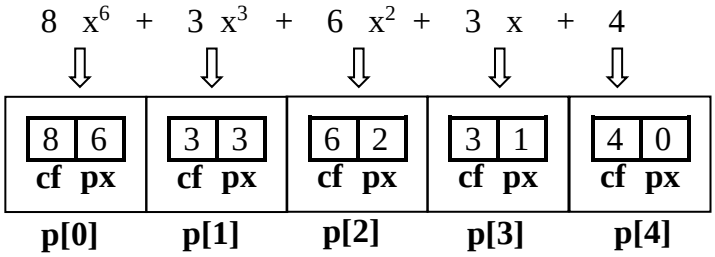
We have represented only one term. Now the question is "How to represent a polynomial with more than one node?" This can be done using array of structures with the following declaration:

```
typedef struct
{
        int     cf;             // used to hold the co-efficient
        int     px;             // used to hold power of x
} POLY;

POLY p[10];
```

Using the above declaration, we have an array of 10 terms where each term has four fields. Consider the following polynomial:

$$8x^6 + 3x^3 + 6x^2 + 3x + 4$$

It can be stored as an array of 5 terms as shown below:



Now, let us see "How to write a function to read a polynomial consisting of *n* terms" Consider the following polynomial with 5 terms:

$$8x^6 + 3x^3 + 6x^2 + 3x + 4$$

Read the coefficient and power of x of each term and copy into *cf* and *px* field as shown below:

```
scanf("%d %d", &cf, &px);
p[i].cf  = cf;
p[i].px = px
```

for i = 0 to 4
i = 0 to 5 – 1
i = 0 to n – 1

Now, the partial code can be written as shown below:

```
        for (i = 0; i < n; i++)
        {
                scanf(“%d %d”, &cf, &px);

                p[i].cf  = cf;
                p[i].px = px;
        }
```

Now, the complete function to read a polynomial with *n* terms is shown below:

Example 5.4: Function to read a polynomial with n terms

```
void read_poly(POLY p[], int n)
{
        int i, cf, px;

        for (i = 0; i < n; i++)
        {
                printf("cf, px:");        scanf("%d %d", &cf, &px);

                p[i].cf  = cf;
                p[i].px = px;
        }
}
```

On similar lines we can print a polynomial with n terms as shown below:

Example 5.5: Function to display a polynomial with n terms

```
void print_poly(POLY p[], int n)
{
        int i;

        for (i = 0; i < n; i++)
        {
                if (p[i].cf < 0)
                        printf("%d", p[i].cf);
                else
                        printf("+ %d", p[i].cf);

                if(p[i].px != 0) printf("x^%d", p[i].px);
        }
        printf("\n");
}
```

Now, the question is "How to add two polynomials" If we want to add two polynomials, first we have to search for power of polynomial 1 in polynomial 2. This can be done using linear search as shown below:

Example 5.6: Function to search for term of poly 1 in poly 2

```
int search (int px1, POLY p2[], int n)
{
        int     j, px2;

        for ( j = 0; j < n; j++)
        {
                px2 = p2[j].px;

                if (px1 == px2) return j;
        }

        return -1;
}
```

Once we know how to search for a term of polynomial 1 in polynomial 2, the general procedure to add two polynomials is shown below:

```
for each term of polynomial 1
        Step 1: access each term of poly 1

        Step 2: search for power of above term in poly2

        Step 3: if found in poly 2
                        Add the coefficients and add sum to poly 3
                else
                        Add the term of poly1 to poly 3
end for

Add remaining terms of poly2 to poly3
```

Now, the above algorithm can be written using C statements as shown below:

Example 5.7: Function to add two polynomials

```
int add_poly (POLY p1[], int m, POLY p2[], int n, POLY p3[])
{
        int     i, k, cf1, px1, pos, sum;

        k = 0;
        for ( i = 0; i < m; i++)
        {
                cf1  = p1[i].cf;
                px1 = p1[i].px;                         /* Access each item of poly 1 */

                pos = search(px1, p2, n);               /* get position of px1 in poly 2 */

                if (pos >= 0)                           /* px1 found in poly 2 */
                {
                        sum = cf1 + p2[pos].cf;         /* Add the coefficients */
                        if (sum != 0)p3[k].cf = sum; /* Insert the sum into poly 3*/

                        p2[pos].cf = -999;              /* Delete the term of poly2 */
                }
                else
                        p3[k].cf = cf1;                 /* Insert co-efficient of poly 1 */

                p3[k].px = px1;                         /* Insert power of x into poly 3 */
                k++;
        }

        k = copy_poly(p3, k,  p2, n);                   /* Copy remaining terms of poly 2*/
                                                        /* return total terms in poly 3 */
        return k;
}
```

Now, the function copy_poly() can be implemented as follows. Immediately after adding the terms of both polynomials, the term of polynomial 2 should be removed. This can be done by assigning -999 to co-efficient field of the corresponding term in poly 2. So, when all terms of polynomial 1 are scanned, we have to copy the remaining terms of polynomial 2 into poly 3. The remaining terms are the terms whose co-efficient field is not -999. Now, the function copy_poly() can be written as shown below:

Example 5.8: Function to copy remaining terms of polynomial 2 into polynomial 3

```
int copy_poly ( POLY p3[], int k, POLY p2[], int n)
{
        int     j;

        for (j = 0; j < n; j++)
        {
                if (p2[j].cf != -999)
                {
                        p3[k].cf  = p2[j].cf;
                        p3[k].px = p2[j].px;
                        k++;
                }
        }
        return k;
}
```

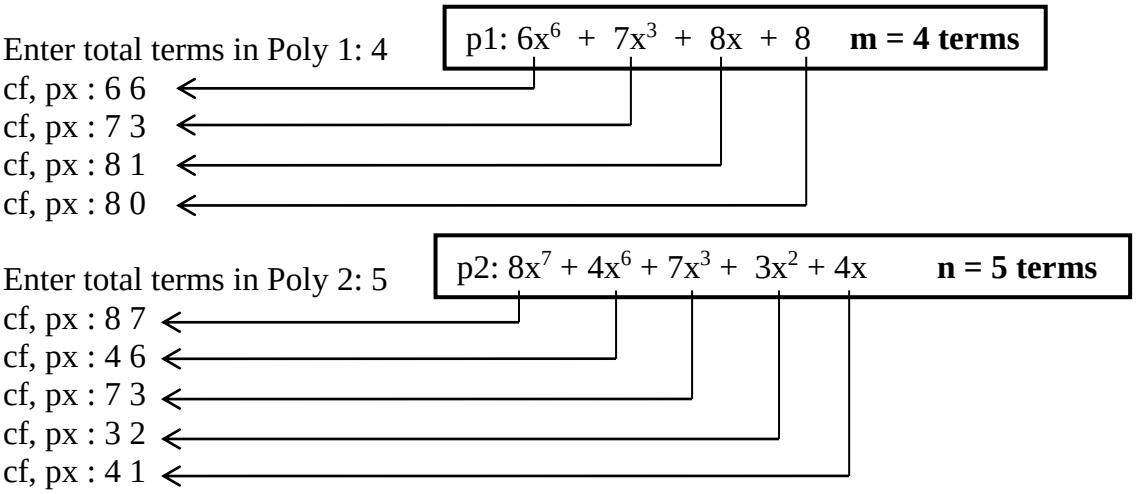
Now, the complete program to add two polynomials can be written as shown below:

Example 5.9: Program to add two add polynomials

```
#include <stdio.h>
typedef struct
{
        int     cf;
        int     px;
} POLY;

// Include : Example 5.4: To read a polynomial
// Include : Example 5.5: To display a polynomial
// Include : Example 5.6: To search for term of poly 1 in poly 2
// Include : Example 5.7: To add two polynomials
// Include : Example 5.8: To copy remaining terms in second polynomial

void main()
{
        POLY p1[20], p2[20], p3[40];
        int     m,      n,      k;

        printf("Enter total terms in Poly 1:");         scanf("%d", &m);
        read_poly(p1, m);
```

```
        printf("Enter total terms in Poly 2:");        scanf("%d", &n);
        read_poly(p2, n);

        printf("Poly 1: ");     print_poly(p1, m);
        printf("Poly 2: ");     print_poly(p2, n);
        printf("-------------------------------------------------------\n");

        k = add_poly (p1, m, p2, n, p3);

        printf("Poly 3: ");     print_poly(p3, k);
}
```

Input



Output:

```
Poly 1: 6x^6 + 7x^3 + 8x^1 + 8
Poly 2: 8x^7 + 4x^6 + 7x^3 + 3x^2 + 4x
-------------------------------------------------------
Poly 3: 10x^6 + 14x^3 + 12x^1 + 8 + 8x^7 + 3x^2
```

5.6 Polynomials with three variables

The polynomial with three variables is shown below:

$$6x^6y^4z^3 + 7x^3yz^2 + 8x^3 + 8$$

The program shown in example 5.9 can be modified to include variables *y* and *z* as shown below:

Example 5.10: Program to add two add polynomials with three variables

```
#include <stdio.h>

typedef struct
{
        int     cf;
        int     px;
        int     py;
        int     pz;
} POLY;

/* Function to read a polynomial with three variables */
void read_poly(POLY p[], int n)
{
        int i, cf, px, py, pz;

        for (i = 0; i < n; i++)
        {
                printf("cf, px, py, pz:");
                scanf("%d %d %d %d", &cf, &px, &py, &pz);

                p[i].cf  = cf;
                p[i].px = px;
                p[i].py = py;
                p[i].pz = pz;
        }
}

/* Function to print a polynomial with three variables */
void print_poly(POLY p[], int n)
{
        int i;
```

```
        for (i = 0; i < n; i++)
        {
                if (p[i].cf < 0)
                        printf("%d", p[i].cf);
                else
                        printf("+ %d", p[i].cf);

                if(p[i].px != 0) printf("x^%d", p[i].px);

                if(p[i].py != 0) printf("y^%d", p[i].py);

                if(p[i].pz != 0) printf("z^%d", p[i].pz);

        }
        printf("\n");
}

/* Function to search for powers of poly 1 in poly 2 */
int search (int px1, int py1, int pz1, POLY p2[], int n)
{
        int     j, px2, py2, pz2;

        for ( j = 0; j < n; j++)
        {
                px2 = p2[j].px;
                py2 = p2[j].py;
                pz2 = p2[j].pz;

                if (px1 == px2 && py1 == py2 && pz1 == pz2) return j;
        }

        return -1;
}

/* Function to add two polynomials with three variables */
int add_poly (POLY p1[], int m, POLY p2[], int n, POLY p3[])
{
        int     i, k, cf1, px1, py1, pz1, pos, sum;

        k = 0;
```

```
        for ( i = 0; i < m; i++)
        {
                cf1  = p1[i].cf;
                px1 = p1[i].px;                 /* Access each item of poly 1 */
                py1 = p1[i].py;
                pz1 = p1[i].pz;

                pos = search(px1, py1, pz1, p2, n);  /* get position of px1 in poly 2 */
                if (pos > 0)                    /* px1 found in poly 2 */
                {
                        sum = cf1 + p2[pos].cf;     /* Add the coefficients */
                        p3[k].cf = sum;             /* Insert the sum into poly 3*/
                        p2[pos].cf = -999;          /* Delete the term of poly2 */
                }
                else
                        p3[k].cf = cf1;             /* Insert co-efficient of poly 1 */
                p3[k].px = px1;                 /* Insert power of x into poly 3 */
                p3[k].py = py1;
                p3[k].pz = pz1;

                k++;
        }

        k = copy_poly(p3, k,  p2, n);           /* Copy remaining terms of poly 3*/
                                                /* return total terms in poly 3 */
        return k;
}

/* Function to copy remaining terms of polynomial 2 into polynomial 3 */
int copy_poly ( POLY p3[], int k, POLY p2[], int n)
{
        int     j;
```

```
        for (j = 0; j < n; j++)
        {
                if (p2[j].cf != -999)
                {
                        p3[k].cf  = p2[j].cf;
                        p3[k].px = p2[j].px;
                        p3[k].py = p2[j].py;
                        p3[k].pz = p2[j].pz;
                        k++;
                }
        }
        return k;
}

void main()
{
        POLY p1[20], p2[20], p3[40];
        int     m,      n,      k;

        printf("Enter total terms in Poly 1:");         scanf("%d", &m);
        read_poly(p1, m);
        printf("Enter total terms in Poly 2:");         scanf("%d", &n);
        read_poly(p2, n);

        printf("Poly 1: ");     print_poly(p1, m);
        printf("Poly 2: ");     print_poly(p2, n);
        printf("-------------------------------------------------------\n");

        k = add_poly (p1, m, p2, n, p3);

        printf("Poly 3: ");     print_poly(p3, k);
}
```

Output

```
Enter total terms in Poly 1:4
cf, px, py, pz:6 3 4 5
cf, px, py, pz:2 2 0 2
cf, px, py, pz:3 0 2 1
cf, px, py, pz:4 0 0 0
Enter total terms in Poly 2:5
```

cf, px, py, pz:6 4 4 5
cf, px, py, pz:2 3 4 5
cf, px, py, pz:6 0 0 3
cf, px, py, pz:7 0 3 0
cf, px, py, pz:5 0 0 0

Poly 1: + 6x^3y^4z^5+ 2x^2z^2+ 3y^2z^1+ 4
Poly 2: + 6x^4y^4z^5+ 2x^3y^4z^5+ 6z^3+ 7y^3+ 5

Poly 3: + 8x^3y^4z^5+ 2x^2z^2+ 3y^2z^1+ 9+ 6x^4y^4z^5+ 6z^3+ 7y^3

5.7 SPARSE MATRICES

Now, let us see "What is a sparse matrix?"

Definition: A sparse matrix is a matrix that has very few non-zero elements spread out thinly. Conversely, a matrix which has more number of zero elements (or high proportion of zeros elements is called sparse matrix. The sparse matrix can be single dimensional or multidimensional such as 2-dimensional, 3-dimensional and so on.

Ex 1: Consider the following single-dimensional matrices:

10	0	0	0	20	0

The matrix has less number of non-zero values (or more 0's). So, it is sparse matrix

10	0	30	40	20	50

The matrix has more number of non-zero values (or less 0's values). So, it is not a sparse matrix

Ex 2: Consider the following two-dimensional matrices:

	col0]	col[1]	col[2]
row[0]	**10**	**20**	**30**
row[1]	**11**	**0**	**31**
row [2]	**12**	**22**	**32**
row [3]	**13**	**23**	**0**

Not a sparse matrix

	col0]	col[1]	col[2]	col[3]
row[0]	**10**	**0**	**0**	**40**
row[1]	**11**	**0**	**22**	**0**
row [2]	**0**	**0**	**0**	**0**
row [3]	**20**	**0**	**0**	**50**
row [4]	**0**	**15**	**0**	**25**

Sparse matrix

Note the following points with respect to above two matrices:

- In the first matrix, 10 non-zero elements are present and 2 zero elements. The number of non-zero elements are more than the number of zero elements. So, it is not a sparse matrix.
- In the second matrix only 8 non-zero elements are present and 12 zero elements are present. So, the number of non-zero elements are less than the number of zero elements. So, it is a sparse matrix.

Now, let us see "What is the disadvantage of a sparse matrix?" The sparse matrix contains many zeroes. If we are manipulating only non-zero values, then we are wasting the memory space by storing unnecessary zero values. This disadvantage can be overcome by storing only non-zero values thus saving the space.

For example, consider a matrix of size 1000 x 1000.

- Assume it has 2000 non-zero elements and remaining are zero elements.
- The corresponding two-dimensional array occupy 1,000,000 memory locations.
- Instead of storing both zero and non-zero elements, if we store only non-zero elements, the space can be reduced.
- This can be done using sparse matrix representation as shown below:

5.7.1 Sparse Matrix Representation

Now, let us see "How sparse matrix can be represented by storing only non-zero values?"

We know that any element in the matrix can be uniquely defined using the triples <row, col, val>. Thus, a sparse matrix can be created using the array of triples as shown below:

```
#define MAX_TERMS 101          /* Maximum number of terms */

typdef struct
{
        int     row;
        int     col;
        int     val;
} TERM;

/* 1- dimensional array representing array of triples <row, col, val> */
TERM        a[MAX_TERMS];
```

Example 2.32: How do you represent the following spare matrix using triples in a single dimensional array?

	col[0]	col[1]	col[2]	col[3]
row[0]	**10**	**0**	**0**	**40**
row[1]	**11**	**0**	**22**	**0**
row [2]	**0**	**0**	**0**	**0**
row [3]	**20**	**0**	**0**	**50**
row [4]	**0**	**15**	**0**	**25**

Solution: The declaration to represent the above sparse matrix can be written as:

```
#define MAX_TERMS 101          /* Maximum number of terms */

typdef struct
{
        int     row;
        int     col;
        int     val;
} TERM;

/* 1- dimensional array representing array of triples <row, col, val> */
TERM        a[MAX_TERMS];
```

The non-zero elements in the above matrix along with *row* and *col* position can be represented using a single dimensional array starting from a[1] as shown below:

	row	col	val	
a[0]	5	4	8	5 x 4 is size and 8 non-zero values in given matrix
a[1]	0	0	10	row 0
a[2]	0	3	40	
a[3]	1	0	11	row 1
a[4]	1	2	22	
a[5]	3	0	20	row 3
a[6]	3	3	50	
a[7]	4	1	15	row 4
a[8]	4	3	25	

Note: Since row 2 in given matrix has all 0 entries, they are not considered. Hence, in this representation row 2 is missing.

The various information can be accessed using as shown below:

- The size of the matrix using : a[0].row, a[0].col
- The number of non-zero elements using : a[0].val
- The row index of a non-zero element : a[j].row
- The column index of a non-zero element : a[j].col
- The index of non-zero element : a[j].val

} for j = 1 to a[0].col

Now, let us see "How to read the elements of sparse matrix?" This can be done using the following code:

```
for (i = 0; i < m; i++)
        for (j = 0; j < n; j++)
                scanf("%d", &a[i][j]);
```

The above code represent the normal way of reading a 2-dimensional array. But, if we represent the sparse matrix using triples, the above code can be written as shown below:

Example 2.33: Function to read the sparse matrix as a triple

```
void read_sparse_matrix(TERM a[], int m, int n)
{
        int     i, j, k, item;

        a[0].row = m, a[0].col = n,   k = 1;

        for (i = 0; i < m; i++)
        {
                for (j = 0; j < n; j++)
                {
                        scanf("%d", &item);
                        if (item == 0) continue;
                        a[k].row = i,   a[k].col = j, a[k].val = item;
                        k++;
                }
        }
        a[0].val = k – 1;
}
```

5.7.2 Transpose of a matrix

Now, let us see "What is the transpose of a matrix?"

Definition: A matrix which is obtained by changing row elements into column elements and column elements into row elements is called transpose of a matrix.

Example 2.34: How do you represent the following sparse matrix using triples in a single dimensional array? Also show the transpose of the given matrix represented as a triple

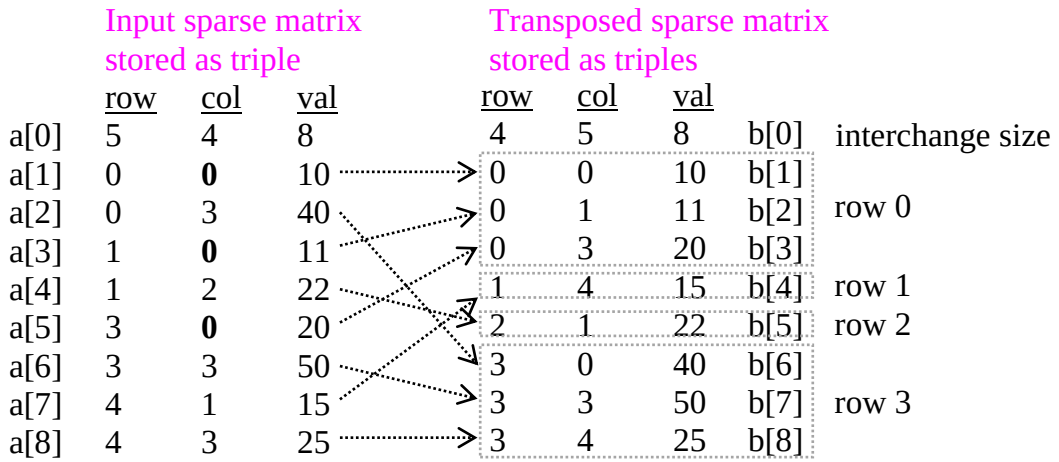
	col[0]	col[1]	col[2]	col[3]
row[0]	**10**	**0**	**0**	**40**
row[1]	**11**	**0**	**22**	**0**
row [2]	**0**	**0**	**0**	**0**
row [3]	**20**	**0**	**0**	**50**
row [4]	**0**	**15**	**0**	**25**

Solution: All non-zero elements represented in the form of a triple can be represented as shown below:

	row	col	val	
a[0]	5	4	8	5 x 4 is size and 8 non-zero values in given matrix
a[1]	0	0	10	row 0
a[2]	0	3	40	
a[3]	1	0	11	row 1
a[4]	1	2	22	
a[5]	3	0	20	row 3
a[6]	3	3	50	
a[7]	4	1	15	row 4
a[8]	4	3	25	

Note: Since row 2 in given matrix has all 0 entries, they are not considered. Hence, in this representation row 2 is missing.

To get the transpose we exchange *row*, *col* indices along with *val* as shown below:



Now, let us "Design the algorithm to transpose a given matrix represented as triples in a single dimensional array" Initially, the size of the matrix must be reversed. This can be done by interchanging rows and columns as shown below:

```
b[0].row = a[0].col
b[0].col  = a[0].row
b[0].val  = a[0].val
```

We find the transpose of given matrix by accessing *col* index of each non-zero element in array *a* and store in array *b* as shown below:

```
b[k].row = a[j].col    // make column as row
b[k].col  = a[j].row   // make row as column
b[k].val  = a[j].val
k++;                   // to add the next entry
```

To obtain the index *j*, we start searching for 0^{th} column, 1^{st} column, 2^{nd} column, 3^{rd} column and 4^{th} column from a[1].col to a[8].col. The corresponding code can be written as shown below:

```
for j = 1 to 8
{
   if (a[j]. col == 0)
   {
      b[k].row = a[j].col
      b[k].col  = a[j].row
      b[k].val  = a[j].val
      k++;
   }
}
```

Search for 0^{th} column

```
for j = 1 to 8
{
   if (a[j]. col == i)
   {
      b[k].row = a[j].col
      b[k].col  = a[j].row
      b[k].val  = a[j].val
      k++;
   }
}
```

Search for 1^{st} column

```
for j = 1 to 8
{
   if (a[j]. col == i)
   {
      b[k].row = a[j].col
      b[k].col  = a[j].row
      b[k].val  = a[j].val
      k++;
   }
}
```

Search for 2^{nd} column

```
for j = 1 to 8
{
   if (a[j]. col == i)
   {
      b[k].row = a[j].col
      b[k].col  = a[j].row
      b[k].val  = a[j].val
      k++;
   }
}
```

Search for 3^{rd} column

In general, we search for i^{th} column as shown below:

```
for (j = 1; j <= 8;  j++)                    // where 8 = a[0].val
{
        if (a[j]. col == i)                  for i = 0, 1, 2, 3
        {
                b[k].row = a[j].col
                b[k].col  = a[j].row
                b[k].val  = a[j].val
                k++;
        }
}
```

The index value i = 0, 1, 2,3 can be written as i = 0 to 3
i = 0 to 4 – 1
i = 0 to a[0].col – 1

Now, the above code can be translated into C as shown below:

Example 2.35: Function to find the transpose of given sparse matrix

```
void transpose(TERM a[], TERM b[])
{
        int      i, j, k;

        b[0].row = a[0].col;
        b[0].col  = a[0].row;
        b[0].val  = a[0].val;

        k = 1;                              // Position of the first non-zero element
        for ( i = 0; i < a[0].col; i++)     // search for columns 0, 1, ….a[0].col
        {
            for (j = 1; j <= a[0].val;  j++)
            {
                if (a[j]. col == i)         // If we get the appropriate column position
                {                           // interchange row and column position
                    b[k].row = a[j].col
                    b[k].col  = a[j].row
                    b[k].val  = a[j].val
                    k++;
                }
           }
        }
}
```

5.7.3 Triangular matrix

Now, let us see "What is a triangular matrix? Is it a sparse matrix?"

Definition: A matrix where all non-zero entries can occur on or below the main diagonal (lower triangular matrix) or non-zero entries can occur on or above the main diagonal (upper triangular matrix) is called triangular matrix. Since only zero elements are present either on upper part of the diagonal or on the lower part of the diagonal, the triangular matrix is a sparse matrix.

For example, the following matrix is a lower triangular matrix.

$$\begin{bmatrix} 10 & & & \\ 11 & 10 & & \\ 6 & 5 & -5 & \\ 20 & 24 & 15 & 8 \end{bmatrix}$$

Now, the question is "How to represent lower triangular matrix?" The lower triangular matrix can be represented using:

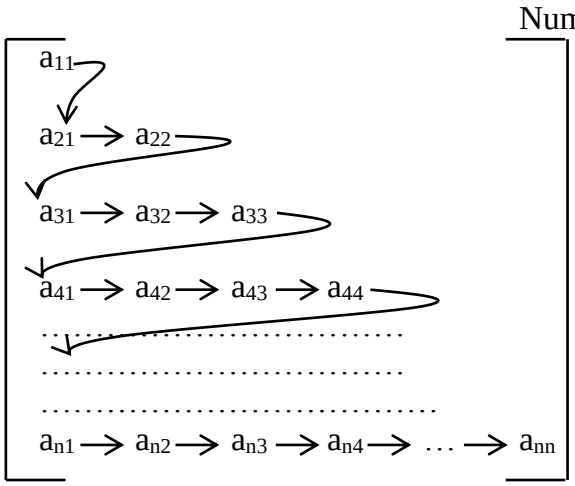
- 2-dimensional array
- 1-dimensional array

2-d representation: We can represent the lower triangular matrix as a two-dimensional array as shown below:

$$\begin{bmatrix} 10 & 0 & 0 & 0 \\ 11 & 10 & 0 & 0 \\ 6 & 5 & -5 & 0 \\ 20 & 24 & 15 & 8 \end{bmatrix}$$

Since all elements above diagonal are 0's, and we are not manipulating 0's, it is waste to store 0's in the memory. It results in wastage of memory space.

1-d representation: Consider the lower triangular matrix shown below:



So, total number of elements = 1 + 2 + 3 + ……n = n (n + 1) /2 elements

All the above items can be stored in a single dimensional array B one row at a time as shown below:

B[1]	a[1,1]	row1
B[2]	a[2,1]	row2
B[3]	a[2,2]	
B[4]	a[3,1]	row3
B[5]	a[3,2]	
B[6]	a[3,3]	
B[7]	a[4,1]	Row4
B[8]	a[4,2]	
B[9]	a[4,3]	
B[10]	a[4,4]	

No. of items up to beginning of row 2 = 1

No. of items up to beginning of row 3 = 1 + 2

No. of items up to beginning of row 4 = 1 + 2 + 3

No. of items up to beginning of row 5 = 1 + 2 + 3 + 4

In general, number of items up to beginning of row i = 1 + 2 + 3 + 4 + ……..(i – 1)

= (i – 1) * i / 2 ……(1)

Once we reach (i-1)th row using the above relation, the location of *k* in array *b* is obtained by simply specifying: **j**……...(2)

So, in array *b*, location of a[i,j] i.e., *k* = eq(1) + eq(2) = i * (i – 1) 2 + j

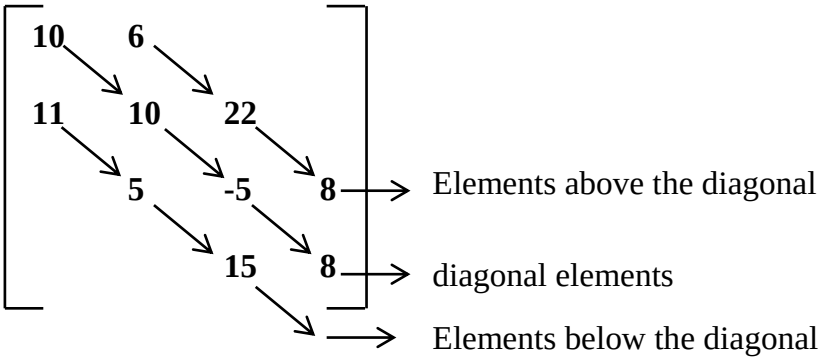
i.e., index of array *b* denoted by k = i * (i-1) / 2 + j

Ex: Loc(a[4,3]) in b i.e., k = i * (i – 1) / 2 + j
= 4 * (4 – 1) / 2 + 3
= 6 + 3
= 9
i.e, b[9] gives a[4, 3] (see contents of array b)

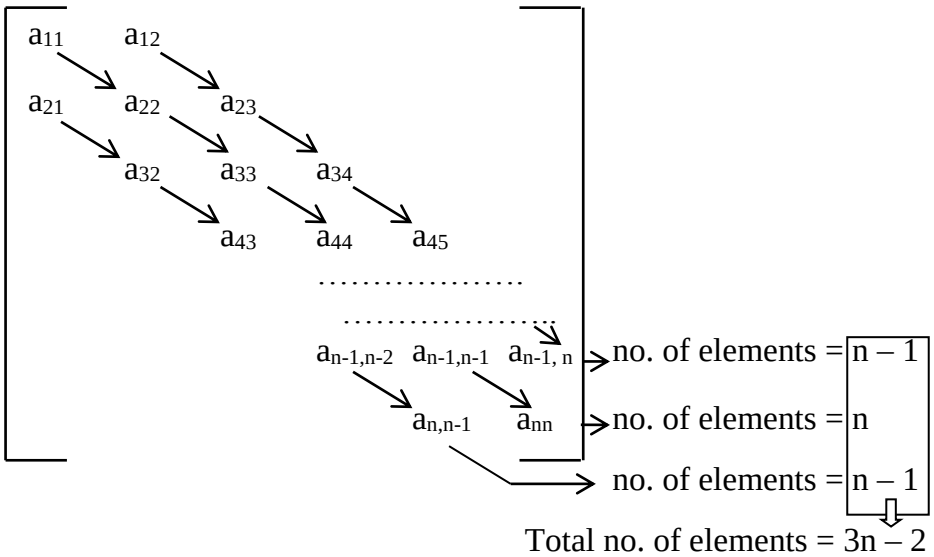
5.7.4 Tridiagonal matrix

Now, let us see "What is a tridiagonal matrix?"

Definition: A matrix where all non-zero entries can occur on the diagonal and just below and above the diagonal is called a tridiagonal matrix. For example, the following matrix is a lower tridiagonal matrix.



Now, the question is "How to represent tridiagonal matrix using one-dimensional array?" The n x n matrix tridiagonal array can be written as shown below:



`The above tridiagonal elements can be stored in single dimensional array b as shown below:

B[1]	a[1,1]	row1
B[2]	a[1,2]	
B[3]	a[2,1]	row2
B[4]	a[2,2]	
B[5]	a[2,3]	
B[6]	a[3,2]	row3
B[7]	a[3,3]	
B[8]	a[3,4]	
B[9]	a[4,3]	row4
B[10]	a[4,4]	
B[11]	a[4,5]	

…..

…..

Now, let us find the formal that will give us the value of *k* in terms of i and j so that we can access the value of a[i, j] from single dimensional array i.e, b[k].

Number of elements above row 2 = 0 + 2 = 3 * 0 + 2 = 3 * (2 – 2) + 2

Number of elements above row 3 = 3 + 2 = 3 * 1 + 2 = 3 * (3 – 2) + 2

Number of elements above row 4 = 6 + 2 = 3 * 2 + 2 = 3 * (4 – 2) + 2

Number of elements above row 5 = 9 + 2 = 3 * 3 + 2 = 3 * (5 – 2) + 2

In general, number of elements above row i = 3 * (i – 2) + 2 **……(A)**

Consider the following part of the above figure:

				displacement
		……..		
		……..		
row i = 4	**B[9]**	a[4,3]	Loc (B[9]) = Loc(a[4,3]) = **(A)** +	1 = 3 – 4 + 2
	B[10]	a[4,4]	Loc (B[10]) = Loc(a[4,4]) = (A) +	2 = 4 – 4 + 2
	B[11]	a[4,5]	Loc (B[11]) = Loc(a[4,5]) = (A) +	3 = 5 – 4 + 2

In general, Loc (B [k]) = Loc(a[i , j]) = (A) + j – i + 2

Substituting relation (A) in above equation, we get

$$Loc(B[k]) = 3*(i-2) + 2 + j - i + 2$$

$$So,\ k = 3i - 6 + 4 + j - i$$

$$\boxed{k = 2i + j - 2}$$

Ex: Loc(a[4,3]) in array B is given by $k = 2*i + j - 2$

$= 2*4 + 3 - 2$

$= 9$

Thus, it is clear from the above diagram that B[9] is indeed a[4,3]

Exercises

1) What is a structure? How to declare a structure?
2) What is a tagged structure? Explain with example
3) How to define the structure along with structure variables?
4) What is a type-defined structure? Explain with example
5) How are structures initialized? How structure members are accessed?
6) How to display the various members of a structure?
7) What is the size of the structure? What are slack bytes?
8) What are the use of structures?
9) What is a union? How union is declared and used in C?
10) What is the difference between a structure and union?
11) What is a polynomial? How you represent a polynomial using arrays?
12) What is a sparse matrix? What is the disadvantage of a sparse matrix?
13) How sparse matrix can be represented by storing only non-zero values?
14) Write a function to read the sparse matrix as a triple?
15) Write a function to find the transpose of a matrix which is stored as a triple
16) Design the algorithm to transpose a given matrix represented as triples in a single dimensional array
17) What is a triangular matrix? Is it a sparse matrix?
18) How to represent lower triangular matrix using single-dimensional array? Obtain the location of a[k] for given item a[i, j] or express *k* in terms of i, j

5.40 Structures and unions

19) What is a tridiagonal matrix? How to represent tridiagonal matrix using one-dimensional array and obtain the relation to get a[k] in terms of i, j given a[i, j]. That is express *k* in terms of i and j

20) How do you represent the following sparse matrix using triples in a single dimensional array? Also show the transpose of the given matrix represented as a triple

	col[0]	col[1]	col[2]	col[3]
row[0]	**10**	**0**	**0**	**40**
row[1]	**11**	**0**	**22**	**0**
row [2]	**0**	**0**	**0**	**0**
row [3]	**20**	**0**	**0**	**50**
row [4]	**0**	**15**	**0**	**25**

Chapter 6: STACKS

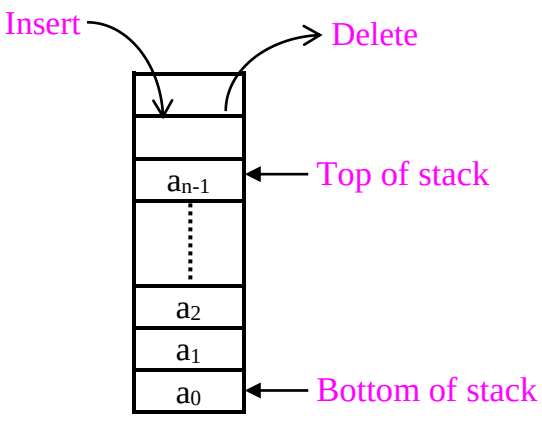
What are we studying in this chapter?

- Stack definition, operations
- Array representation of stacks
- Stacks using dynamic arrays
- Applications:
  - Polish notations
  - Infix to postfix conversion (Reverse polish notation)
  - evaluation of postfix expression
- Recursion
  - Factorial, GCD, Fibonacci Sequence, Tower of Hanoi
  - Ackerman's Recursive function, maze problem
- Implementation of multiple stacks

6.1 Introduction

In this section, let us see "What is a stack?"

Definition: A stack is a special type of data structure (linear data structure) where elements are inserted from one end and elements are deleted from the same end. Using this approach, the **L**ast element **I**nserted is the **F**irst element to be deleted **O**ut, and hence, stack is also called **L**ast **I**n **F**irst **O**ut **(LIFO)** data structure. The stack s={a_0, a_1, a_2,……a_{n-1}) is pictorially represented as shown below:



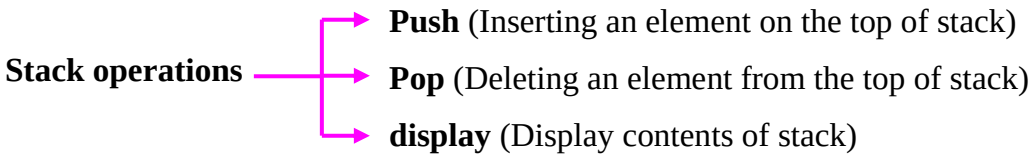
The elements are inserted into the stack in the order a_0, a_1, a_2,……a_{n-1}. That is, we insert a_0 first, a_1 next and so on. The item a_{n-1} is inserted at the end. Since, it is on top of the stack, it is the first item to be delted. The various operations performed on stack are:

- **Insert :** An element is inserted from top end. Insertion operation is called push operation
- **Delete:** An element is deleted from top end only. Deletion operation is called pop operation.

- **Overflow:** Check whether the stack is full or not.
- **Underflow:** Check whether the stack is empty or not.

Note: We have seen that, normally all the plates in a hotel/cafeteria are placed one above the other. The plates are inserted from the top only. If a plate is required, the top most plate is selected. This process of inserting the plates at the top and removing the plates from the top is called stacking of plates.

Let us see how stacks are used in the field of computer science. Before proceeding further, let us see "What are the various operations that can be performed on stacks?" The various operations that can be performed on stacks are shown below:



6.1.1 Push operation

Now, let us see "What is push operation?"

Definition: Inserting an element into the stack is called push operation. Only one item is inserted at a time and item has to be inserted only from top of the stack.

Example 6.1: Stack contents after inserting 4 items 30, 20, 25 and 10 one after the other with a STACK_SIZE 4.

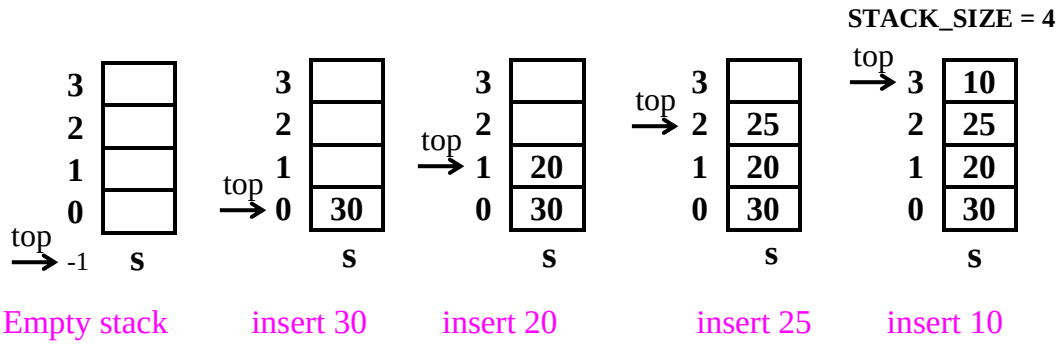


Figure 6.1 Sequence of insertion operations

When items are being inserted into the stack, we may get stack overflow. Now, let us see "What is stack overflow?"

Definition: When elements are being inserted, there is a possibility of stack being full. Once the stack is full, it is not possible to insert any element. Trying to insert an element, even when the stack is full results in overflow of stack.

For example, consider the stack shown in figure 6.1 with STACK_SIZE 4. After inserting 30, 20, 25 and 10 there is no space to insert any item. Then we say that stack is full. Trying to insert an element into the stack when the stack is full is called overflow of stack.

Now, let us see "How to implement push operation using arrays (static allocation technique)?"

Design: Before inserting any element, we should ask the question "Where and how the item has to be inserted?" If we know the answer for this question we have the **push** function ready. So, let us consider the situation where two elements 30 and 20 are already inserted into the stack as shown in figure 6.2.a. with top = 1.

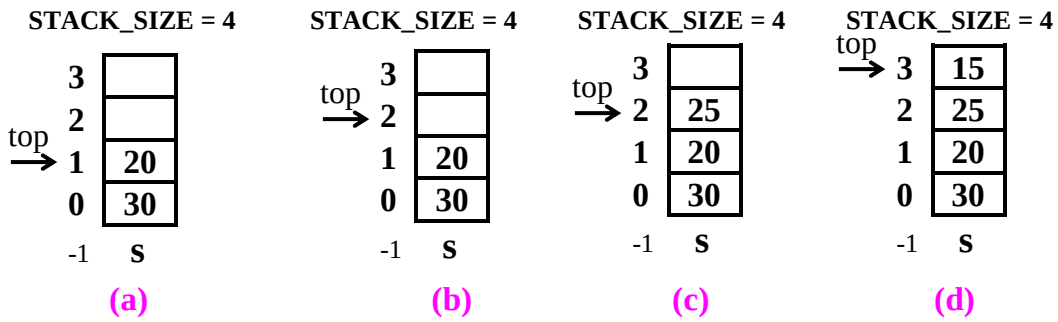


Fig. 6.2 Showing the design of push operation

Suppose, we have to insert an element say **item**. Now, let us ask "Where this item has to be inserted?"

We know that it has to be inserted at top = 2. For this to happen, we have to increment **top** by 1 (Fig. 6.2.b). This is achieved using the statement:

```
top = top + 1;          /* Increment top by 1 */
```

Then we ask "How item has to be inserted at **top** position?" This is achieved by copying **item** to s[top] (Fig 6.2.c) using the following statement:

```
s[top] = item;          /* Insert into stack */
```

But, as we insert an item, we must ask "When we cannot insert item into the stack?" We cannot insert any item into the stack when **top** has already reached STACK_SIZE–1 (Fig. 6.2.d). In such situation, we have to display appropriate message as shown below using the following code:

```
/* Check for overflow of stack */
if (top == STACK_SIZE – 1)
{
        printf("Stack overflow\n");
        return;
}
```

Here, STACK_SIZE should be *#define*d (preprocessor directive) and is called symbolic constant. If the above condition fails, the item has to be inserted. Now, the complete C function for this can be written as shown below:

Example 6.2: C function to insert an integer item (Using global variables)

```
void push()
{
        /* Check for overflow of stack */
        if (top == STACK_SIZE – 1)
        {
                printf("Stack overflow\n");
                return;
        }

        top = top + 1;          /* Increment top by 1 */
        s[top] = item;          /* Insert into stack */     } s[++top] = item;
}
```

Note: The array **s**, the variable **top** and the variable **item** are global variables and should be declared before all the functions. As far as possible let us avoid the usage of global variables in this book.

Let us rewrite the above code by passing parameters. It is clear from the above code that as the **item** is inserted, the contents of the stack identified by **s** and the index **top** pointing to topmost element are changed and so they should be passed as parameters (pass by reference) as shown below:

Example 6.3: C function to insert an integer item (by passing parameters)

```
void push(int item, int *top, int s[])
{
        /* Check for overflow of stack */
        if (*top == STACK_SIZE – 1)
        {
                printf("Stack overflow\n");
                return;
        }

        *top = *top + 1;        /* Increment top by 1 */
        s[*top] = item;         /* Insert into stack */
}
```

} s[++(*top)] = item;

Note: The identifier STACK_SIZE should be *#define*d (preprocessor directive) and is called symbolic constant. The following function inserts a character item into the stack.

Example 6.4: C function to insert a character item (by passing parameters)

```
void push(char item, int *top, char s[])
{
        /* Check for overflow of stack */
        if (*top == STACK_SIZE – 1)
        {
                printf("Stack overflow\n");
                return;
        }

        *top = *top + 1;                /* Increment top by 1 */
        s[*top] = item;                 /* Insert into stack */
}
```

} s[++(*top)] = item;

Note: By comparing functions shown in 6.3 and 6.4 note that body of the function has not been changed. But, the type of formal parameters changes.

6.1.2 Pop operation

Let us see "What is pop operation?"

6.6 Stacks

Definition: Deleting an element from the stack is called pop operation. Only one item can be deleted at a time and item has to be deleted only from top of the stack.

Example 6.5: Perform pop operation when stack already contains 30, 20, 25, and 10.

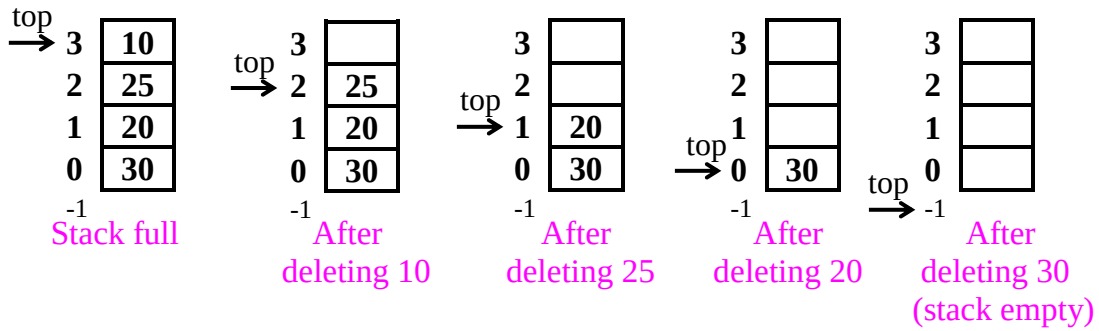


Figure 6.3 Sequence of Deletion operations

The items can be deleted one by one as shown in figure 6.3. The contents of stack and the **top** which contains the position of topmost elements are also shown. When items are being deleted, we may get stack underflow. Now, let us see "What is stack underflow?"

Definition: When elements are being deleted, there is a possibility of stack being empty. When stack is empty, it is not possible to delete any item. Trying to delete an element from an empty stack results in stack underflow.

For example, in the figure 6.3, after deleting 10, 25, 20 and 30 there are no elements in the stack and stack is empty. Deleting an element from the empty stack results in stack underflow.

Now, let us see "How to implement pop operation using arrays (static allocation technique)?"

Design: Let us access an item from the top of the stack. This can be achieved by using the following statement:

```
item_deleted = s[top];          /* Access the topmost item */
```

and then decrementing **top** by one as shown below:

```
top  = top – 1;                 /* Update the position of topmost item */
```

The above two statements can also be written using single statement as shown below:

```
item_deleted = s[top--];
```

Note: item_deleted = s[--top]; **is wrong.**

Each time, the item is deleted, **top** is decremented and finally, when the stack is empty the value of **top** will be –1. In this situation, it is not possible to delete any item from the stack. The code to delete an item from stack can be written as shown below:

Example 6.6: C function to delete an integer item (using global variables)

```
int pop()
{
        int     item_deleted;

        if (top == -1) return 0;        /* Stack underflow? */

        item_deleted = s[top--];        /* Access the item and delete */

        return item_deleted;            /* Send the item deleted to the calling function */
}
```

Now, let us write the above function without using global variables and by passing appropriate parameters. We know that to access the top item, we need the index **top** and the stack **s**. So, we have to pass **top** and **s** as the parameters. As the value of **top** changes every time the item is deleted, **top** can be used as a pointer variable. The complete C function (by passing parameters) is shown below:

Example 6.7: C function to delete an integer item (by passing parameters)

```
int pop(int *top, int s[])
{
        int     item_deleted;           /* To hold the top most item of stack */

        /* Stack underflow? */
        if (*top == -1) return 0;

        /* Obtain the top most element and change the position of top item */
        item_deleted = s[(*top)--];

        /* Send the item deleted to the calling function */
        return item_deleted;
}
```

The following function shows how to delete a character item from the stack.

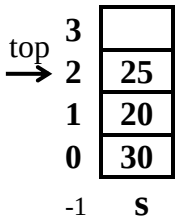
Example 6.8: C function to delete a character item (by passing parameters)

```
char pop(int *top, char s[])
{
        char    item_deleted;           /* Holds the top most item of stack */

        /* Stack underflow? */
        if (*top == -1) return 0;

        /* Obtain the top most element and change the position of top item */
        item_deleted = s[(*top)--];

        /* Send the item deleted to the calling function */
        return item_deleted;
}
```

6.1.3 Display stack items

In the display procedure, if the stack already has some items, all those items are displayed one after the other. If no items are present, the appropriate error message is displayed. Let us design the display function.

Design: Assume that the stack contains three elements as shown below:



Usually, the contents of the stack are displayed from the bottom to top. So, first item to be displayed is 30, next item to be displayed is 20 and final item to be displayed is 25. This can be achieved using the following statements:

Design

```
printf("%d\n", s[0]);
printf("%d\n", s[1]);
printf("%d\n", s[2]);
```

Output

30
20
25

In general, we can use **printf("%d\n", s [i]);**

Note: i = 0 to 2 (i.e., **top**).

Note: Please note how the value of **i** change from **0** to **top.** Now, the code takes the following form:

```
for (i = 0; i <= top; i++) printf("%d\n", s[i]);
```

But, the above statement should not be executed when stack is empty i.e., when *top* takes the value –1. So, the above code can be modified to include this error condition and can be written as shown below:

Example 6.9: C function to display the contents of stack (using global variables)

```
void display()
{
        int i;

        if (top == -1)                          /* Is stack empty */
        {
                printf("Stack is empty\n");
                return;
        }

        /* Display contents of stack */
        printf("Contents of the stack\n");

        for (i = 0; i <= top; i++) printf("%d\n", s[i]);
}
```

By passing parameters, the above function can be written as shown below:

Example 6.10: C function to display contents of the stack (by passing parameters)

```
void display(int top, int s[])
{
        int i;

        if (top == -1)                          /* Is stack empty */
        {
                printf("Stack is empty\n");
                return;
        }
```

```
        /* Display contents of stack*/
        printf("Contents of the stack\n");
        for (i = 0; i <= top; i++)
        {
                printf("%d\n", s[i]);
        }
}
```

6.1.4 Stack implementation using arrays (static implementation of stacks)

In the previous sections we have seen how the stacks can be implemented using global variables and by passing parameters. The program to implement stack operations such as push, pop and display can be written as shown below:

Example 6.11: C function to display contents of the stack (by passing parameters)

```
#include <stdio.h>
#include <process.h>

#define STACK_SIZE 5

/* Global variables */

int     top;            /* index to hold the position of the top most item */
int     s[10];          /* Holds the stack items */
int     item;           /* Item to be inserted into the stack */

/* Insert: Example 6.2 : function push */

/* Insert: Example 6.6 : function pop */

/* Insert: Example 6.9: function display */

void main()
{
        int     choice;                 /* user choice for push, pop and display */

        top = -1;                       /* Indicates stack is empty */
```

```
        for (;;)
        {
                printf("1:Push    2:Pop\n");
                printf("3:Display 4:Exit\n”);
                printf("Enter the choice\n");
                scanf("%d", &choice);

                switch(choice)
                {
                        case 1:
                                printf("Enter the item to be inserted\n");
                                scanf("%d", &item);
                                push();
                                break;

                        case 2:
                                item = pop();

                                if (item == 0)
                                        printf("Stack is empty\n");
                                else
                                        printf("Item deleted = %d\n",item);

                                break;

                        case 3:
                                display();
                                break;

                        default:
                                exit(0);
                }
        }
}
```

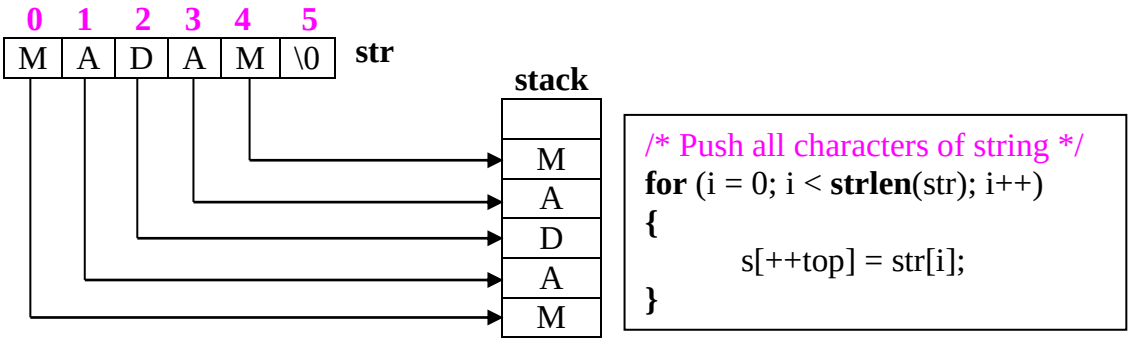
6.1.5 Check for a palindrome

First, let us see “What is a palindrome?”

6.12 Stacks

Definition: A palindrome is a string which happens to be the same if we read from left to right and right to left. For example, the strings "LIRIL", "RADAR", "MADAM", "MALAYALAM" etc. are strings of palindromes.

Design: Given the string say "MADAM", access each element of the given string and push it onto the stack as shown below:



Now, compare each character of the given string with corresponding character from top of the stack. If there is any mismatch we return 0 indicating the string is not a palindrome. Otherwise, we return 1 indicating the given string is a palindrome. The code for this can be written as:

```
/* Check whether the string is a palindrome or not */
for (i = 0; i < strlen(str); i++)
{
        stk_item = s[top--];                    /* Pop an item from stack */

        if (str[i] != stk_item) return 0;       /* i/p not equal to stack symbol */
}

return 1;                                       /* The string is a palindrome */
```

Now, the complete function can be written as shown below:

Example 6.12: C function to check for a palindrome using stack

```
int is_palindrome(char str[])
{
        int i;

        int     top;                    /* Initial top of the stack */
        char    s[20], stk_item;                /* stack variables */
```

```
        top = -1;

        /* Push all characters of string */
        for (i = 0; i < strlen(str); i++) s[++top] = str[i];

        for (i = 0; i < strlen(str); i++)          /* Check for palindrome */
        {
                stk_item = s[top--];              /* Pop an item from stack */

                if (str[i] != stk_item) return 0; /* i/p not equal to stack symbol */
        }
        return 1;                                 /* The string is a palindrome */
}
```

Example 6.13: Design, Develop and Implement a menu driven Program in C for the following operations on **STACK** of Integers (Array Implementation of Stack with maximum size **MAX**)

a. ***Push*** an element on to Stack
b. ***Pop*** an element from Stack
c. Demonstrate how Stack can be used to check ***Palindrome***
d. Demonstrate ***Overflow*** and ***Underflow*** situations on Stack
e. Display the status of Stack
f. Exit

```
#include <stdio.h>
#include <process.h>
#include <string.h>

#define STACK_SIZE 5

/* Insert: Example 6.3 : function push */

/* Insert: Example 6.7 : function pop */

/* Insert: Example 6.10 : function display */

/* Insert: Example 6.12 : function is_palindrome */

void main()
{
        int     item, top, s[10];       /* stack variables */
        char    str[20];                /* Used to check for palindrome */
        int     choice, flag;
```

```
top = -1;                              /* Indicates stack is empty to start with */

for (;;)
{
        printf("1:Push      2:Pop\n");
        printf("3:Display 4:Palindrome\n");
        printf("5:Exit\n");

        printf("Enter the choice\n");
        scanf("%d", &choice);

        switch(choice)
        {
                case 1:
                    printf("Enter the item to be inserted\n");
                    scanf("%d",&item);

                    push(item, &top, s);

                    break;

                case 2:
                    item = pop(&top, s);

                    if (item == 0)
                        printf("Stack is empty\n");
                    else
                        printf("Item deleted = %d\n", item);

                    break;

                case 3:
                    display(top, s);
                    break;

                case 4:
                      printf("Enter the string\n");
                      scanf("%s", str);

                      flag = is_palindrome(str);
```

```
                        if (flag == 0)
                                printf("The string is not a palindrome\n");
                        else
                                printf("The string is a palindrome\n");

                        break;

                default:
                    exit(0);
            }
        }
}
```

6.2 Applications of stack

Once we know the definition of a stack and how stack is implemented in C, let us concentrate on "What are the applications of stack?" The various applications in which stacks are used are:

- **Conversion of expressions:** When we write mathematical expressions in a program, we use infix expressions. These expressions will be converted into equivalent machine instructions by the compiler using stacks. Using stacks, we can efficiently convert the expressions from infix to postfix, infix to prefix etc.
- **Evaluation of expressions:** An arithmetic expression represented in the form of either postfix or prefix can be easily evaluated.
- **Recursion**: A function which calls itself is called recursive function. Some of the problems such as tower of Hanoi, problems involving tree manipulations etc., can be implemented very efficiently using recursion. It is a very important facility available in variety of programming languages such as C, C++ etc.,
- **Other applications:** There are so many other applications where stacks can be used: For example, to find whether the string is a palindrome, to check whether a given expression is valid or not and so on.

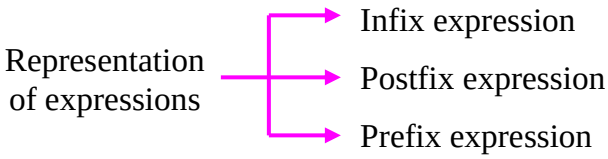
Now, we shall discuss these applications in detail in the subsequent sections.

6.3 Conversion of expressions

First, let us see "What is an expression? What are the various types of expressions?"

6.16 Stacks

Definition: The sequence of operators and operands that reduces to a single value after evaluation is called an expression. The operands consist of constants and variables whereas the operators consist of symbols such as +, -, *, / and so on. The operators indicate the operation to be performed on the operands specified. The expressions can be represented as shown below:

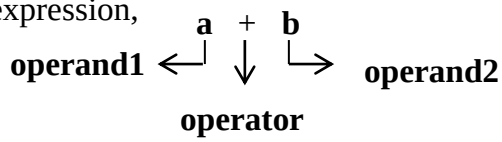


Now, let us see "What is an infix expression?"

Definition: In an expression, if an operator is in between two operands, the expression is called an infix expression. The infix expression can be parenthesized or un-parenthesized. For example,

a + b is an un-parenthesized infix expression
(**a + b**) is a parenthesized infix expression

In the expression,



a and *b* are operands and the symbol + is an operator.

Now, let us see "What is postfix expression?"

Definition: In an expression, if an operator follows the two operands (i.e., operator comes after the two operands), the expression is called postfix expression. No, parenthesis is allowed in postfix expressions. The postfix expression is always un-parenthesized. It is also called suffix expression or reverse polish expression. For example,

a b +

Now, let us see "What is prefix expression?"

Definition: In an expression, if an operator precedes the two operands (i.e., operator comes before the two operands), the expression is called prefix expression. No, parenthesis is allowed in prefix expressions. The prefix expression is always un-parenthesized. It is also called polish expression. For example,

+ a b

6.3.1 Precedence and associativity of the operators

While evaluating or converting an expression into other forms, we should know the precedence of operators and the associativity of operators. So, let us see "What is precedence of operators?"

Definition: The rules that determine the order in which different operators are evaluated are called precedence rules or precedence of operators.

Normally we associate values to determine the order in which the operators have to be evaluated. Highest precedence operators will have the highest value and lowest precedence operators will have the least value. The table below shows arithmetic operators along with priority values.

	Description	Operator	Priority	Associativity
Arithmetic Operators	Exponentiation	($, ^)	6	Right to Left
	Multiplication	(*)	4	Left to Right
	Division	(/)	4	Left to Right
	Mod	(%)	4	Left to Right
	Addition	(+)	2	Left to Right
	Subtraction	(–)	2	Left to Right

Note: The symbol '$' or '^' is considered as the operator to perform exponentiation and should be given first precedence since it has higher priority value. The addition or subtraction is given the least precedence since it has least priority value as shown in above table.

Now, "What if different operators have the same precedence?" During evaluation, if two or more operators have the same precedence, then precedence rules are not applicable. Instead, we go for **associativity** of the operators. Now, we shall see "What is associativity?

Definition: The order in which the operators with same precedence are evaluated in an expression is called associativity of the operator. In such case, precedence rules are not considered.

Example 6.14: What is the result of **8 + 4 + 3?**

All of us evaluate the expression as shown below:

$$\underbrace{8 + 4}_{12} + 3 \quad \textbf{[First add 8 and 4 to get 12]}$$

$$\underbrace{12 + 3}_{15} \quad \textbf{[Next add 12 and 3 to get 15]}$$

Observation: In the above expression, same operator "+" is used twice and hence both operators have the same priority and the precedence rules are not applicable. Then the question is "How to evaluate?" Normally, we do the evaluation as shown below:

First we compute 8 + 4 to get 12
Next we compute 12 + 3 to get 15 thus moving from Left to Right.

Since evaluation is done from left to right one after the other, we say that the operator "+" is left associative.

Now, let us "Define left associative"

Definition: In an expression, if two or more operators have the same priority and are evaluated from left-to-right, then the operators are called left associative operators. We normally denote using L → R. The process of evaluating from left to right is called left associativity.

Example 6.15: What is the result of **2$3$2?** if '$' represent exponentiation operator?

Note: Instead of using symbol '$' we can use symbol '^' to denote exponentiation.

Note: In mathematics **2$3$2** can be represented as 2^{3^2}. We know that $2^{3^2} = 512$. It is not 64. So, to get the answer 512, the computer has to evaluate the expression 2$3$2 as shown below:

$$2\ \$\ \underbrace{3\ \$\ 2}_{9} \qquad \text{First evaluate } 3\ \$\ 2 \text{ (i.e., } 3^2 = 9)$$

$$\underbrace{2\ \$\ \ 9}_{512} \qquad \text{Then evaluate } 2\$9 \text{ (i.e., } 2^9 = 512)$$

Observation: In the above expression, same operator "$" is used twice and both operators have the same priority and the precedence rules are not applicable. Then the question is "How to evaluate?" Normally, we do the evaluation as shown below:

First we compute 3$2 to get 9
Next we compute 2$9 to get 512 thus moving from right to left.

Since evaluation is done from right to left one after the other, we say that the operator "$" is right-to-left associative (also called right associative).

Now, let us "Define right associative (right associative).

Definition: In an expression, if two or more operators have the same priority and if they are evaluated from right-to-left, then the operators are called right associative operators.

6.3.2 Conversion from infix to postfix

Example 6.16: Obtain the postfix expression for **((A + (B – C) * D) ^ E + F)**

Solution: We can convert into postfix expression based on precedence and associativity as shown below:

((A + $\underbrace{(\ B - C\ )}_{T_1}$ * D) ^ E + F)	**(B-C) has highest precedence** **T_1 = B C –**
((A + $\underbrace{T_1 * D}_{T_2}$) ^ E + F)	**(T_1 * D) has highest precedence** **T_2 = T_1 D ***
(($\underbrace{A + T_2}_{T_3}$) ^ E + F)	**(A + T_2) has highest precedence** **T_3 = A T_2 +**
($\underbrace{T_3 \wedge E}_{T_4}$ + F)	**(T_3 ^ E) has highest precedence** **T_4 = T_3 E ^**

$\underbrace{(T_4 + F)}$ **Converting into postfix by repeated substitution**

T_4 F +

T_3 E ^ F + (Replacing T_4 by T_3 E ^)

A T_2 + E ^ F + (Replacing T_3 by A T_2 +)

A T_1 D * + E ^ F + (Replacing T_2 by T_1 D *)

A B C – D * + E ^ F + (Replacing T_1 by B C –)

Hence, equivalent postfix expression is **A B C – D * + E ^ F +**

Example 6.17: Obtain the postfix expression for **X ^ Y ^ Z – M + N + P / Q**

Solution: We can convert into postfix expression based on precedence and associativity as shown below:

X $ $\underbrace{Y\ \$\ Z}_{T_1}$ – M + N + P / Q	Y $ Z has highest precedence and right associative **T_1 = Y Z $**
$\underbrace{X\ \$\ T_1}_{T_2}$ – M + N + P / Q	X $ T_1 has highest precedence **T_2 = X T_1 $**
T_2 – M + N + $\underbrace{P\ /\ Q}_{T_3}$	P / Q has the highest precedence **T_3 = P Q /**
$\underbrace{T_2 – M}_{T_4}$ + N + T_3	**Note:** All have the same precedence and hence associativity comes into picture. Here, they are left associative **T_4 = T_2 M –**
$\underbrace{T_4 + N}_{T_5}$ + T_3	T_4 + N is considered because of left associative **T_5 = T_4 N +**

$\underbrace{T_5 + T_3}$ **Converting into postfix by repeated substitution**

T_5 T_3 +

T_4 N + P Q / (Replacing T_5 by T_4 N + and T_3 by P Q /

T_2 M – N + P Q / (Replacing T_4 by T_2 M –)

X T_1 $ M – N + P Q / (Replacing T_2 by X T_1 $)

X Y Z $ $ M – N + P Q / (Replacing T_1 by Y Z $)

Hence, equivalent postfix expression is **X Y Z $ $ M – N + P Q /**

Now, let us discuss "How to write a program to convert a given infix expression to its equivalent postfix expression?"

Design: The following precedence table is used while converting from infix to postfix.

Symbols	Stack precedence function F	Input precedence function G	Associativity
+,-	2	1	Left
*, /	4	3	Left
$ or ^	5	6	Right
operands	8	7	Left
(	0	9	Left
)	-	0	-
#	-1		-

The two functions F and G using the above precedence table can be written as shown below:

Example 6.18: C function which returns stack and input precedence values

```
int F(char symbol)
{
        switch(symbol)
        {
                case '+':
                case '-': return 2;

                case '*':
                case '/': return 4;

                case '^':
                case '$': return 5;

                case '(': return 0;

                case '#': return -1;

                default: return 8;
        }
}
```

```
int G(char symbol)
{
        switch(symbol)
        {
                case '+':
                case '-': return 1;

                case '*':
                case '/': return 3;

                case '^':
                case '$': return 6;

                case '(': return 9;

                case ')': return 0;

                default: return 7;
        }
}
```

Note: Observe the following points from the above table/function with respect to precedence:

- The operators '+' and '–' have the same precedence and are least precedence operators.
- The operators '*' and '/' also have the same precedence and have the precedence higher than '+' and '-'.
- The operator '$' or '^' indicates exponential operator and has a precedence higher than '*' and '\' but it is right associative operator.

Note: Observe the following points from the above table/function with respect to associativity.

- If an operator is left associative, stack precedence value is greater than input precedence value.
- If an operator is right associative, stack precedence value is less than the input precedence value.

Procedure: General procedure to convert from infix to postfix form is shown below:

- As long as the precedence of symbol on top of the stack is greater than the precedence of input symbol, pop an item from the stack and place it in the postfix expression. The code for this statement can be of the form:

```
while ( F(s[top]) > G(symbol) )
{
        postfix[j++] = s[top--]        /* Pop from stack and place into postfix */
}
```

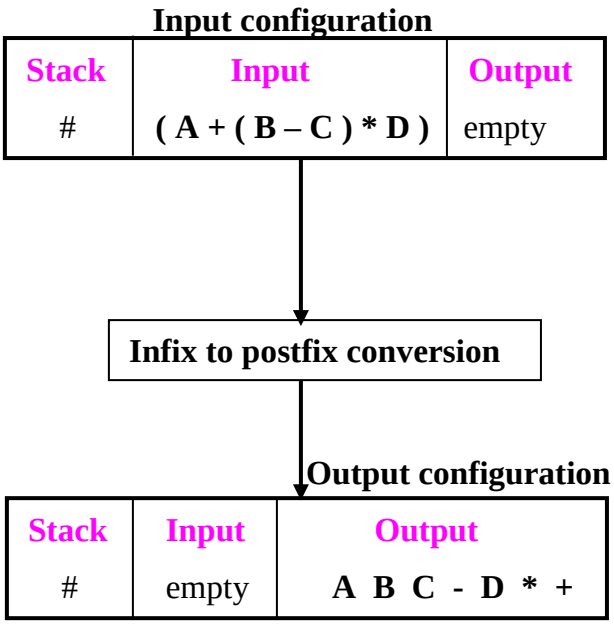
- Once control comes out of the above loop, if the precedence of the symbol on top of the stack is not equal to the precedence of the current input symbol, push the current symbol on to the stack. Otherwise, delete an item from the stack. The code for this statement can be of the form

```
if ( F(s[top]) != G(symbol) )
        s[++top] = symbol;             /* Push symbol on the stack */
else
        top--;                         /* Drop an element from stack */
```

Note: The above steps have to be performed for each input symbol in the infix expression.

Note: Before conversion starts, stack is empty denoted by symbol '#' on the stack and output is empty. The formal version of the algorithm can be written as:

```
top = -1;
s[++top] = '#';     /* stack empty */
j = 0;              /* Output is empty */

for each symbol in infix expression
{
        while ( F(s[top]) > G(symbol) )
                postfix[j++] = s[top--];

        if ( F(s[top]) !=  G(symbol) )
                s[++top] = symbol;
        else
                top--;
}
```



The complete C function to convert from infix expression to postfix expression is shown below:

Example 6.19: C function to convert from infix expression into postfix expression

```
void infix_postfix(char infix[], char postfix[])
{
        int     top;            /* points to top of the stack */
        int     j;              /* Index for postfix expression */
        int     i;              /* Index to access infix expression */

        char    s[30];          /* Acts as storage for stack elements */
        char    symbol;         /* Holds scanned char from infix expression */

        top = -1;               /* Stack is empty */
        s[++top] = '#';         /* Initialize stack to # */

        j = 0;                  /* Output is empty. So, j = 0 */
```

```
        for( i = 0; i < strlen(infix); i++)
        {
                symbol = infix[i];                      /* Scan the next symbol */

                while ( F(s[top]) > G(symbol) )         /* if stack precedence is greater */
                        postfix[j++] = s[top--];        /* Pop and place into postfix*/

                if ( F(s[top]) != G(symbol) )
                        s[++top] = symbol;              /* push the input symbol */
                else
                        top--;                          /* discard '(' from stack */
        }

        while ( s[top] != '#') postfix [j++] = s[top--]; /*pop and place in postfix */

        postfix[j] = '\0';                              /* NULL terminate */
}
```

The complete C program to convert an infix expression to postfix expression is shown below:

Example 6.20: C Program to convert an infix expression to postfix

```
#include <stdio.h>
#include <string.h>

/* Include: Example 6.18: Precedence functions F and G */

/* Include: Example 6.19: Function to convert from infix to postfix */
void main()
{
        char infix[20];
        char postfix[20];

        printf("Enter a valid infix expression\n");
        scanf("%s",infix);

        /* Convert infix to postfix expression */
        infix_postfix(infix, postfix);

        printf("The postfix expression is\n");
        printf("%s\n",postfix);
}
```

Input and output

```
Enter a valid infix expression
    ((a+b)*c-(d-e))^(f+g)
The postfix expression is
    ab+c*de--fg+^

Enter a valid infix expression
    a^b*c-d+e/f/(g+h)
The postfix expression is
    ab^c*d-ef/gh+/+
```

Let us trace the above program for the given expression: (A + (B – C) * D)

Example 6.21: The complete trace of the program for the expression:

(A + (B – C) * D)

Stack	s[top]	Symbol	F(s[top]) > G(symbol)	Postfix
#	#	(	(-1 < 9) Push (	
# (	(	A	(0 < 7) Push A	
# (A # (	A (	+	(8 > 1) Pop A (0 < 1) Push +	A
# (+	+	(	(2 < 9) Push (	A
# (+ (	(	B	(0 < 7) Push B	A
# (+ (B # (+ (	B (	-	(8 > 1) Pop B (0 < 1) Push -	A B
# (+ (-	-	C	(2 < 7) Push C	A B
# (+ (- C # (+ (- # (+ (	C - (	)	(8 > 0) Pop C (2 > 0) Pop - (0 = 0) Pop (don't place in postfix expr.	A B C A B C -
# (+	+	*	(2 < 3) Push *	A B C -
# (+ *	*	D	(4 < 7) Push D	
# (+ * D # (+ * # (+ # (	D * + (	)	(8 > 0) Pop D (4 > 0) Pop * (2 > 0) Pop + (0 = 0) Pop (don't place in postfix exp.)	A B C – D A B C – D * A B C – D * +
#	#	-	-	**A B C – D * +**

6.3.3 Conversion from infix to prefix

Example 6.22: Obtain the prefix expression for **((A + (B – C) * D) ^ E + F)**

Solution: We can convert into prefix expression based on precedence and associativity as shown below:

((A + $\underbrace{(B – C)}_{T_1}$) * D) ^ E + F)	**(B-C) has highest precedence** **T_1 = – B C**
((A + $\underbrace{T_1 * D}_{T_2}$) ^ E + F)	**(T_1 * D) has highest precedence** **T_2 = * T_1 D**

(($\underbrace{A + T_2}_{T_3}$) ^ E + F)	**($A + T_2$) has highest precedence** **$T_3 = + A\ T_2$**
($\underbrace{T_3 \wedge E}_{T_4}$ + F)	**(T_3 ^ E) has highest precedence** **$T_4 = \wedge\ T_3\ E$**

$\underbrace{(T_4 + F)}$ **Converting into postfix by repeated substitution**

+ T_4 F	
+ ^ T_3 E F	(Replacing T_4 by ^ T_3 E)
+ ^ + A T_2 E F	(Replacing T_3 by + A T_2)
+ ^ + A * T_1 D E F	(Replacing T_2 by * T_1 D)
+ ^ + A * – B C D E F	(Replacing T_1 by – B C)

Hence, equivalent postfix expression is **+ ^ + A * – B C D E F**

Example 6.23: Obtain the prefix expression for **X ^ Y ^ Z – M + N + P / Q**

Solution: We can convert into prefix expression based on precedence and associativity as shown below:

X \$ $\underbrace{Y \$ Z}_{T_1}$ – M + N + P / Q	Y \$ Z has highest precedence and right associative **T_1 = \$ Y Z**
$\underbrace{X \$ T_1}_{T_2}$ – M + N + P / Q	X \$ T_1 has highest precedence **T_2 = \$ X T_1**
T_2 – M + N + $\underbrace{P / Q}_{T_3}$	P / Q has the highest precedence **T_3 = / P Q**
$\underbrace{T_2 – M}_{T_4}$ + N + T_3	**Note:** All have the same precedence and hence associativity comes into picture. Here, they are left associative **T_4 = – T_2 M**
$\underbrace{T_4 + N}_{T_5}$ + T_3	T_4 + N is considered because of left associative **T_5 = + T_4 N**

$\underbrace{T_5 + T_3}$ **Converting into postfix by repeated substitution**

$+ T_5\ T_3$

$+ + T_4\ N\ /\ P\ Q$ (Replacing **T_5** by **$+ T_4\ N$** and **T_3** by **$/\ P\ Q$** **)**

$+ + - T_2\ M\ N\ /\ P\ Q$ (Replacing **T_4** by **$- T_2\ M$**)

$+ + - \$\ X\ T_1\ M\ N\ /\ P\ Q$ (Replacing **T_2** by **$\$\ X\ T_1$**)

$+ + - \$\ X\ \$\ Y\ Z\ M\ N\ /\ P\ Q$ (Replacing **T_1** by **$\$\ Y\ Z$**)

Hence, equivalent postfix expression is **+ + – $ X $ Y Z M N / P Q**

On similar lines, we can convert all infix expressions into their equivalent prefix expressions. Now, let us discuss "How to write a program to convert a given infix expression to its equivalent prefix expression?"

Design: To obtain a prefix expression from infix expression, the procedure that is followed while converting from infix to postfix expression is used with the following modifications:

- The precedence functions are different from that of the precedence functions shown earlier (given in infix to postfix conversion) The precedence functions used to obtain a prefix expression are shown below:

Symbols	Stack precedence function F	Input precedence function G	Associativity
+,-	1	2	Left
*, /	3	4	Left
$ or ^	6	5	Right
operands	8	7	Left
(	-	0	Left
)	0	9	-
#	-1	-	-

Table 6.2 Precedence values of symbols in the stack and input

- Reverse the infix expression and follow the same procedure that we have used to convert from infix to postfix
- The prefix expression is obtained by reversing the result.

Example 6.24: C function which returns stack and input precedence values

```
/* Stack precedence function F */
int F(char symbol)
{
        switch(symbol)
        {
                case '+':
                case '-': return 1;

                case '*':
                case '/': return 3;

                case '^':
                case '$': return 6;

                case ')': return 0;

                case '#': return -1;

                default: return 8;
        }
}
```

```
/* Input precedence function G */
int G(char symbol)
{
        switch(symbol)
        {
                case '+':
                case '-': return 2;

                case '*':
                case '/': return 4;

                case '^':
                case '$': return 5;

                case '(': return 0;

                case ')': return 9;

                default: return 7;
        }
}
```

Note: Observe the following points from the above table/function with respect to precedence:

- The operators '+' and '–' have the same precedence and are least precedence operators.
- The operators '*' and '/' also have the same precedence and have the precedence higher than '+' and '-'.
- The operator '$' or '^' indicates exponential operator and has a precedence higher than '*' and '\' but it is right associative operator.

Note: Observe the following points from the above table/function with respect to associativity.

- If an operator is left associative, stack precedence value is less than input precedence value.
- If an operator is right associative, stack precedence value is greater than the input precedence value.

Procedure: General procedure to convert from infix to prefix form is shown below:

- As long as the precedence value of the symbol on top of the stack is greater than the precedence value of the current input symbol, pop an item from the stack and place it in the prefix expression. The code for this statement can be of the form:

```
while ( F(s[top]) > G(symbol) )
{
        prefix[j++] = s[top--];        /* Pop from stack and place into prefix */
}
```

- Once the condition in while-loop is failed, if the precedence of the symbol on top of the stack is not equal to the precedence value of the current input symbol, push the current symbol on to the stack. Otherwise, delete an item from the stack but do not place it in the prefix expression. The code for this statement can be of the form

```
if ( F(s[top]) != G(symbol) )
        s[++top] = symbol;             /* Push symbol on the stack */
else
        top--;                         /* Drop an element from stack */
```

Note: These two steps have to be performed for each input symbol in the infix expression.

The complete C function to convert from infix expression to postfix expression is shown below:

Example 6.25: C function to convert from infix expression into prefix expression

```
void infix_prefix(char infix[], char prefix[])
{
        int     top;            /* Points to top of the stack */
        char    s[30];          /* Acts as storage for stack elements */
        int     j;              /* Index for prefix expression */
        int     i;              /* Index to access infix expression */
        char    symbol;         /* Holds scanned char from infix expression */

        top = -1;               /* Stack is empty */
        s[++top] = '#';         /* Initialize stack to # */
        j = 0;                  /* Points to first char of prefix expression */
```

```
        strrev(infix);          /* Reverse the infix expression */

        for( i = 0; i < strlen(infix); i++)
        {
                symbol = infix[i];      /* Scan the next symbol */

                while ( F(s[top]) > G(symbol) )
                {
                        prefix[j] = s[top--];           /* Pop from stack and place */
                        j++;                            /* into prefix */
                }

                if ( F(s[top]) != G(symbol) )
                        s[++top] = symbol;              /* push the input symbol */
                else
                        top--;                          /* discard '(' from stack */
        }

        while ( s[top] != '#')
        {
                prefix [j++] = s[top--];
        }

        prefix[j] = '\0';               /* Attach NULL char at the end of string */

        strrev(prefix);                 /* To obtain the actual prefix expression */
}
```

The complete C program to convert a valid infix expression to postfix expression is shown below:

Example 6.26: C program to convert from infix expression into prefix expression

```
#include <stdio.h>
#include <string.h>

/* Include: Example 6.24: Two functions F and G */

/* Include: Example 6.25: Function to convert from infix to prefix  */
```

```
void main()
{
        char infix[20];
        char prefix[20];

        printf("Enter a valid infix expression\n");
        scanf("%s",infix);

        infix_prefix(infix, prefix);

        printf("The prefix expression is\n");
        printf("%s\n",prefix);
}
```

Input and Outputs

```
Enter a valid infix expression
((a+b)*c-(d-e))^(f+g)
The prefix expression is
^-*+abc-de+fg

Enter a valid infix expression
a^b*c-d+e/f/(g+h)
The prefix expression is
+-*^abcd//ef+gh
```

Note: The precedence values of the operands in functions G and F in example 6.22 are 7 and 8. The same values are used in example 6.28. This is because the order of the operands in the prefix, postfix and infix expressions are same. Only the order of the operators changes. So, care should be taken while taking the precedence values for the operators. For example, consider the following expressions:

a+b*c	(Infix expression)
abc*+	(Postfix expression)
+a*bc	(Prefix expression)

If we look at above expressions, the operands a, b and c appear in the same sequence whereas the operators are not in same sequence. So, the precedence value of the operands is same, but the precedence values of operators are different while converting from infix to postfix and from infix to prefix.

6.3.4 Evaluation of Postfix expression

Let us see "What is the problem in evaluating the infix expressions? What is the need for evaluating postfix/prefix expressions?" The evaluation of infix expression is not recommended because of the following reasons:

- Evaluation of infix expression requires the knowledge of precedence of operators and the associativity of operators.
- The problem becomes complex, if there are parentheses in the expression because they change the order of precedence.
- During evaluation, we may have to scan from left to right and right to left repeatedly thereby complexity of the program increases.

All these problems can be avoided if the infix expression is converted to its corresponding postfix or prefix expression and then evaluate. Evaluation of a postfix expression or prefix expression is very simple.

Now, let us see "How to evaluate the postfix expression?" The postfix expression can be evaluated using the following procedure:

- Scan the symbol from left to right
- If the scanned symbol is an operand, push it on to the stack
- If the scanned symbol is an operator, pop two elements from the stack. The first popped element is operand2 and the second popped element is operand1. This can be achieved using the statements

```
op2 = s[top--];          /* First popped element is operand2 */
op1 = s[top--];          /* Second popped element is operand1 */
```

- Perform the indicated operation

```
res = op1 op op2         /* op is an operator such +, -, /, * etc. */
```

- Push the result on to the stack.
- Repeat the above procedure till the end of input is encountered.

The algorithm or pseudocode to evaluate the postfix expression is shown below:

Example 6.27: Algorithm to evaluate the postfix expression

```
while end of input is not reached do
{
    symbol =  nextchar()
    If ( symbol is an operand )
        push(symbol, top, s);        /* Push the operand */
    else
        op2 = pop(top, s);           /* Pop into second operand */
        op1 = pop(top, s);           /* Pop into first operand */
        res = op1 op op2;            /* Perform the operation */
        push(res, top, s);           /* Push the result */
    endif
}
```

The complete program to evaluate the postfix expression is shown below:

Example 6.28: C program to evaluate the postfix expression

```
#include <stdio.h>
#include <math.h>
#include <string.h>

/* Function to evaluate */
double compute(char symbol, double op1, double op2)
{
        switch(symbol)
        {
                case '+': return op1 + op2;          /* Perform addition operation */

                case '-': return op1 - op2;          /* Perform subtraction operation */

                case '*': return op1 * op2;          /* Multiply two operands */

                case '/': return op1 / op2;          /* Perform division */

                case '$':
                case '^': return pow(op1, op2);      /* Compute power */
        }
}

void main()
{
        double      s[20];      /* Place for stack elements */
        double      res;        /* Holds the result of partial or final result */
        double      op1;        /* First operand */
        double      op2;        /* Second operand */

        int     top;            /* Points to the topmost element */
        int     i;              /* Index to obtain each symbol from the postfix */

        char    postfix[20];    /* Valid input expression */
        char    symbol;         /* Scanned symbol of postfix */

        printf("Enter the postfix expression\n");
        scanf("%s",postfix);
```

```
        top = -1;                                   /* Stack is empty */

        for(i = 0; i < strlen(postfix); i++)
        {
                symbol = postfix[i];                /* Obtain the next character */

                if ( isdigit(symbol) )              /* If operand insert into the stack */
                        s[++top] = symbol –'0';
                else
                {
                        op2 = s[top--];         /* Obtain second operand from stack */
                        op1 = s[top--];         /* Obtain first operand from stack */

                        /* Perform the specified operation */
                        res = compute(symbol, op1, op2);

                        s[++top] = res;         /* Push the partial result on the stack */
                }
        }

        res = s[top--];                         /* Obtain result from the stack */

        printf("The result is %f\n",res);
}
```

Output

```
Enter the postfix expression
23/
The result is 0.666667
Enter the postfix expression
23+
The result is 5.00000
```

Example 6.29: Give the tracing to evaluate the following postfix expression

A B C – D * + E $ F +

corresponding to the infix expression ((A + (B – C) * D) $ E + F) with following values assigned: A = 6, B = 3, C= 2, D = 5, E = 1, F = 7

Solution: After substituting the given values, the resulting postfix expression is:

6 3 2 – 5 * + 1 $ 7 +

The tracing of the above algorithm is shown below:

Postfix Expression	Symbol Scanned	Op2	Op1	Result = Op1 **op** op2	Stack Contents
6 3 2 – 5 * + 1 ^ 7 +	**6**				6
3 2 – 5 * + 1 ^ 7 +	**3**				6 3
2 – 5 * + 1 ^ 7 +	**2**				6 3 2
- 5 * + 1 ^ 7 +	**-**	2	3	3 – 2 =1	6 1
5 * + 1 ^ 7 +	**5**				6 1 5
***** + 1 ^ 7 +	*****	5	1	1 * 5 = 5	6 5
+ 1 ^ 7 +	**+**	5	6	6 + 5 = 11	11
1 ^ 7 +	**1**				11 1
^ 7 +	^	1	11	11 ^ 1 = 11	11
7 +	**7**				11 7
+	**+**	7	11	11 + 7 = 18	18

The postfix expression is a sequence of characters. So, when an operand is pushed on to the stack, its ASCII value will be pushed. But, we want to push the corresponding integer value. This is achieved by subtracting '0' which is 48 in decimal from an operand

Note: If '6' is an operand its integer value is '6'-'0' i.e., 6. This integer value will be pushed on to the stack.

Note: The assumption is that the input consists of only non-negative, single digit integer numbers and the expression is valid.

Example 6.30: Apply the evaluation algorithm and trace for the valid postfix expression

A B C + * C B A - + *

for given value A = 1, B = 2, C = 3 is shown below.

Solution: Substituting the values for the variables we have the following postfix expression:

1 2 3 + * 3 2 1 - + *

The complete tracing is shown below:

Postfix Expression	Symbol Scanned	Op2	Op1	Result = Op1 **op** op2	Stack Contents
1 2 3 + * 3 2 1 – + *	1				1
2 3 + * 3 2 1 – + *	2				1 2
3 + * 3 2 1 –+ *	3				1 2 3
+ * 3 2 1 – + *	+	3	2	2 + 3 = 5	1 5
***** 3 2 1 – + *	*	5	1	1 * 5 = 5	5
3 2 1 – + *	3				5 3
2 1 – + *	2				5 3 2
1 – + *	1				5 3 2 1
– + *	–	1	2	2 – 1 = 1	5 3 1
+ *	+	1	3	3 + 1 = 4	5 4
*****	*	5	4	4 * 5 = 20	20

So, after evaluation of the postfix expression
A B C + * C B A - + * for given value A = 1, B = 2, C = 3,

result = 20

Example 6.31: Apply the evaluation algorithm discussed in section 6.3.13 and trace for the valid postfix expression

A B + C – B A + C $ -

for given value A = 1, B = 2, C = 3 is shown below.

Solution: Substituting the values for the variables we have the following postfix expression:

1 2 + 3 – 2 1 + 3 $ –

and the complete tracing is shown in table below:

Postfix Expression	Symbol Scanned	Op2	Op1	Result = Op1 **op** op2	Stack Contents
1 2 + 3 – 2 1 + 3 $ –	1				1
2 + 3 – 2 1 + 3 $ –	2				1 2
+ 3 – 2 1 + 3 $ –	+	2	1	1 + 2 = 3	3
3 – 2 1 + 3 $ –	3				3 3
– 2 1 + 3 $ –	–	3	3	3 – 3 = 0	0
2 1 + 3 $ –	2				0 2
1 + 3 $ –	1				0 2 1
+ 3 $ –	+	1	2	2 + 1 = 3	0 3
3 $ –	3				0 3 3
$ –	$	3	3	3 $ 3 = 27	0 27
–	–	27	0	0 – 27 = -27	-27

So, after evaluation of the postfix expression A B + C – B A + C $ – for given value A = 1, B = 2, C = 3,

result = -27

6.4 Stacks using Dynamic arrays

Now, let us see "How to insert an element into a stack using dynamic array?" Assume *s* is pointer to an integer, *top* is an index to top of the stack and initial size of the stack denoted by SIZE is 1.

Design: Before inserting, we check whether sufficient space is available in the stack. We know that if *top* value is same as SIZE – 1, then stack is full. The code for this condition can be written as shown below:

```
if (top == SIZE – 1)
{
        printf("Stack overflow\n");
        return;
}
```

But, once the stack is full, instead of returning the control, we can increase the size of array using realloc() function and the above code can be modified as shown below:

```
if (top == SIZE – 1)
{
        printf("Stack Full: update size, insert item\n ");
        SIZE++;
        s = (int *) realloc(s, SIZE*sizeof(int) );
}
```

When above condition fails, it means we can insert an item. If the above condition is true, space is not sufficient. So, using realloc() we increase the size by 1 and we can insert an item. To insert an *item*, we have to increment *top* by 1 as shown below:

```
top = top + 1;          /* Increment top by 1 */
```

Then, the *item* can be inserted on top of the stack using:

```
s[top] = item;          /* Insert into stack */
```

Now, the complete function can be written as shown below:

Example 6.32: Push() using global variables and by passing parameters

```
void push()
{
  /* Check for overflow of stack */
  if (top == SIZE – 1)
  {
    printf("Stack Full: Increase size by 1");

    SIZE++;

    s = (int *) realloc(s, SIZE*sizeof(int) );
  }

  top = top + 1;
  s[top] = item;      } s[++top] = item;
}
```

```
void push(int item, int *top, int *s)
{
  /* Check for overflow of stack */
  if (*top == SIZE - 1)
  {
      printf("Stack Full: update size, item insert\n");

      SIZE++;

      s = (int *) realloc(s, SIZE*sizeof(int) );
  }

  *top = *top + 1;
  [*top] = item;
}
```

Note: The functions pop() and display() are same as we discussed earlier.

Now, the complete C program to implement stack using dynamic arrays is shown below:

Example 6.33: C program to implement stacks using dynamic arrays

```
#include <stdio.h>
#include <stdlib.h>

int SIZE = 1;  // Global variable

// Include : Example 6.32: To insert an item into stack
// Include : Example 6.7:  To delete an item from stack*/
// Include : Example 6.10:  To display contents of stack*/

void main()
{
        int item, choice;
        int top = -1, *a;

        a = (int *) malloc (sizeof (int) );

        for(; ;)
        {
                printf("1:Push  2:pop \n");
                printf("3:Display  4:Exit\n");
                scanf("%d", &choice);
                switch (choice)
                {
                        case 1:
                                printf("Enter the item\n");
                                scanf("%d", &item);
                                push(item,&top, a);
                                break;
                        case 2:
                                item = pop(&top, a);

                                if (item == -1)
                                        printf("Stack is empty\n");
                                else
                                        printf("Item deleted = %d\n", item);

                                break;

                        case 3: display(top, a);
                                break;
```

```
                    default: exit(0);
                }
        }
}
```

6.5 Multiple Stacks

Multiple stacks can be easily implemented using array of linked lists (discussed in next chapter in section 8.9.5). Now, let us see "How to implement multiple stacks using a single dimensional array?" We can easily implement multiple stacks by knowing:

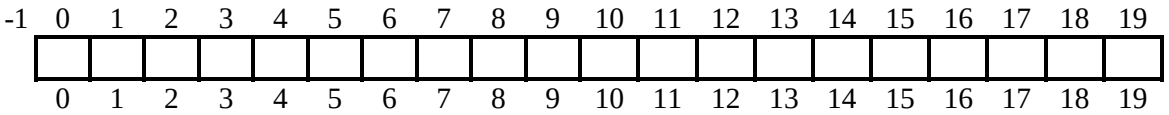
- How to initialize multiple stacks?
- How to find overflow of multiple stacks?
- How to implement push, pop and display operations?

6.5.1 Initializing multiple stacks

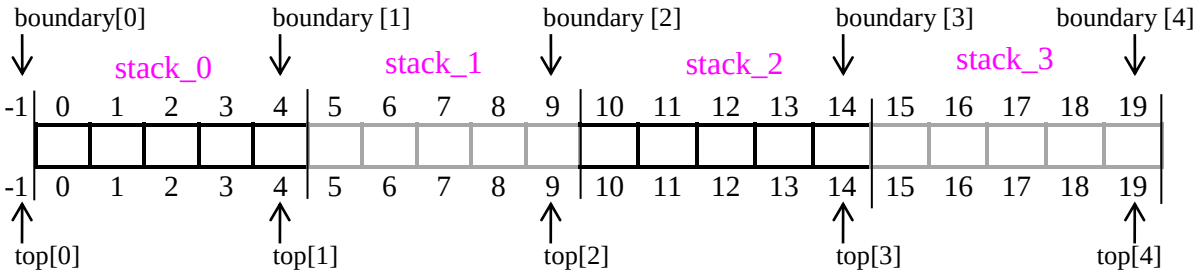
Let us assume *s* is a single dimensional array whose maximum size is defined as a symbolic constant MAX_SIZE. The corresponding declarations can be written as:

```
#define     MAX_SIZE   20
int         s[MAX_SIZE];      // Maximum of 20 items can be inserted
```

The pictorial representation of entire array *s* is shown below:



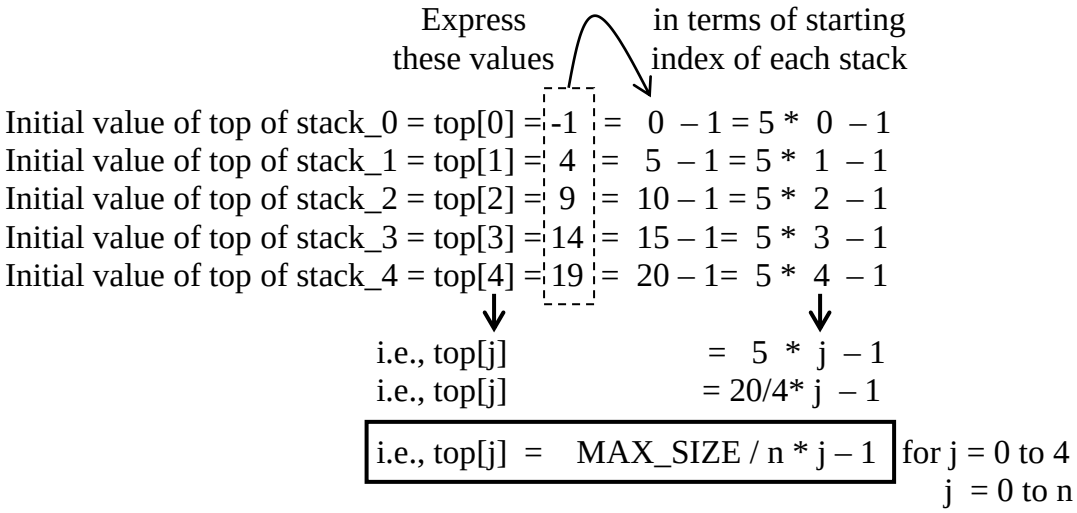
Let us divide the array into *n* stacks where *n* is the number entered by the user. In our example, let us say *n* is 4. Now, 4 empty stacks can be pictorially represented as shown below:



Observe the following points from above 4 stacks:

- Stack_0 starts from index 0
- Stack_1 starts from index 5
- Stack_2 starts from index 10
- Stack_3 starts from index 15

Now, let us see, "How to find the initial *top* value for each stack?" Note the initial values of each stack:



Now, the code to find the initial *top* value of each stack can be written as shown below:

```
for ( j = 0; j <=n ; j++)
{
        top[j] = MAX_SIZE / n * j  –  1;
}
```

Now, the question is "How to find the initial value of boundary of each stack?" Observe from the 4 stacks given in the previous page that initial *top* value of each stack is same as initial *boundary* value. The code for this case can be written as shown below:

```
for ( j = 0; j <=n ; j++)
{
        boundary[j] = top[j];
}
```

Now, the above two for loops can be combined into a single for-loop as shown below:

```
for (j = 0; j <= n ; j++)
{
        boundary [j] = top[j] = MAX_SIZE / n * j – 1;
}
```

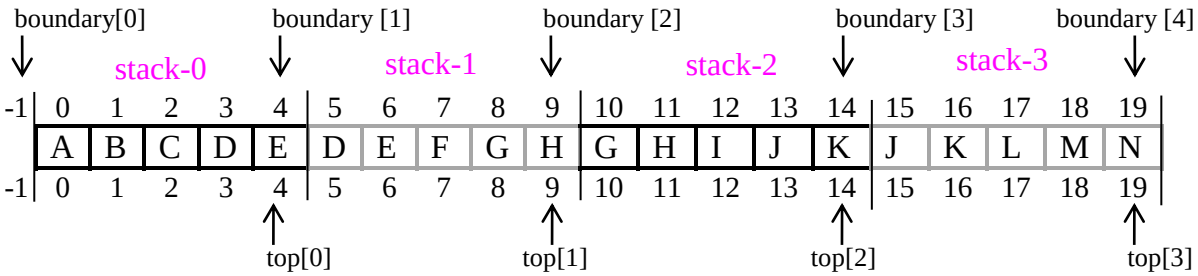
Now, the relevant declarations can be written as shown below:

```
#define     MAX_SIZE        20
int         s[MAX_SIZE];

#define     MAX_STACKS      5       // Maximum number of stacks
int         boundary[MAX_STACKS];
int         top[MAX_STACKS];

int         n;                      // Number of stacks entered by the user
int         item;                   // item to be inserted
int         i;                      // Stack number
```

6.5.2 Finding overflow for each stack

Now, the question is "How to find overflow for each stack?"

Design: Let us consider an array with 4 stacks where five elements are inserted into each stack. This can be pictorially represented as shown below:



Observe the following facts from above figure:

- Stack_0 is full when top[0] is same as boundary[1]
- Stack_1 is full when top[1] is same as boundary[2]
- Stack_2 is full when top[2] is same as boundary[3]
- Stack_3 is full when top[3] is same as boundary[4]

In general, stack_i is full when top[i] is same as boundary[i+1]. This can be coded as

```
if (top[i] == boundary[i+1]
{
        printf("Stack %d is full\n", i);
        return;
}
```

6.5.3 Insert an item into the stack

Now, the question is "How to insert an item into multiple stacks?" Now, all other operations such as push(), pop() and display remains same as the normal stack. The code for normal push operation is:

```
top++;
s[top] = item;
```

Now, *top* is replaced by *top[i]*. Now, the above set of activities can be written by replacing *top* by *top[i]* as shown below:

```
top[i]++;                /* Increment top of stack i by 1 */
s[top[i]] = item;        /* Insert item into stack i*/
```

Now, the complete function to insert an item into the stack can be written as shown below:

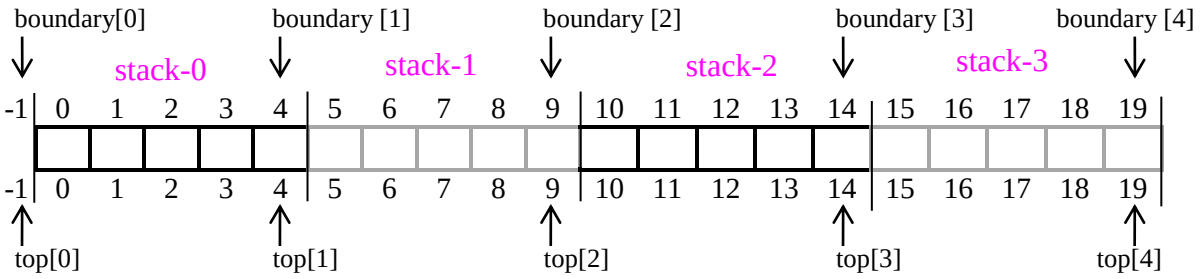
Example 6.34: Push() using global variables and by passing parameters

```
void push()
{
    /* Check for overflow of stack */
   if (top[i] == boundary[i+1])
   {
        printf("Stack %d is full\n", i);
        return;
   }

    top[i]++;
    s[top[i]] = item;      } s[++top[i]] = item;
}
```

```
void push(int item, int s[], int top[],
                    int boundary[], int i)
{
   /* Check for overflow of stack */
   if (top[i] == boundary[i+1])
   {
        printf("Stack %d is full\n", i);
        return;
   }

   top[i]++;
   s[top[i]] = item;       } s[++top[i]] = item;
}
```

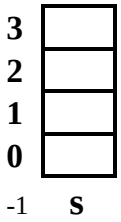
6.5.4 check for underflow of stack

Now, let us see "How to check for underflow in multiple stacks?" Consider the following figure where 4 stacks are used using only one single-dimensional array:



Since no elements are present in any of the stack, all stacks are empty. Observe the following facts:

- When top[0] is same as boundary[0] stack_0 is empty
- When top[1] is same as boundary[1] stack_1 is empty
- When top[2] is same as boundary[2] stack_2 is empty
- When top[3] is same as boundary[3] stack_3 is empty



In general, when top[i] is same as boundary[i] stack_i is empty. This can be written as shown below:

```
if (top[i] == boundary[i]) return -1;          // indicates stack is empty
```

6.5.5 Delete an item from the stack

Now, let us see "How to delete an element from multiple stack" In a normal stack, the topmost element is identified by s[top]. In multiple stack, the topmost element is identified by s[top[i]] and it can be returned using the following statement:

```
return s[top[i]--];
```

So, the final code can be written as shown below:

```
if (top[i] == boundary[i])   return -1;        // indicates stack is empty

return s[top[i]--];                            // remove from the stack
```

Now, the complete function can be written as shown below:

Example 6.35: Pop() using global variables and by passing parameters

```
int pop()
{
    if (top[i] == boundary[i])
        return -1;      // stack is empty

    return s[top[i]--];
}
```

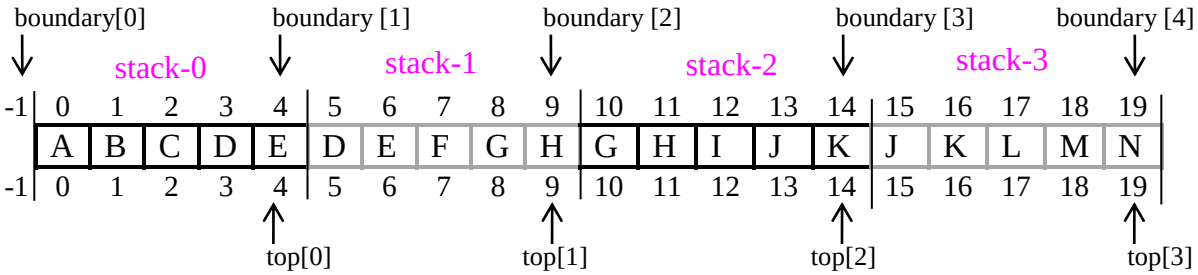
```
int pop(int s[], int top[], int boundary[], int i)
{
    if (top[i] == boundary[i])
        return -1;      // stack is empty

    return s[top[i]--];
}
```

6.5.6 Display stack items

Let us see "How to display stack elements from multiple stacks?" In section 6.5.4 we have seen that if top[i] is same as boundary[i] then stack is empty. The code for this can be written as:

```
if (top[i] == boundary[i])
{
        printf("Stack %d is empty\n",i);
        return;
}
```

When control comes out of the above if-statement, it means that stack is not empty. The contents of multiple stacks can be written as shown below:



Observe from above figure that contents of *stack[i]* starts from "*boundary[i] + 1*" and goes up to *top[i]*. So, the contents of the stack_i can be displayed using the following statements:

```
for (j = boundary[i]+1; j <= top[i]; j++)
        printf("%d\n", s[j]);
```

Now, the complete function to display stack_i can be written as shown below:

Example 6.36: Display contents of stack using global variables and by passing parameters

```
void display()
{
    int j;

    /* If stack is empty */
    if (top[i] == boundary[i])
    {
        printf("Stack %d empty\n",i);
        return;
    }

    /* Display contents of stack */
    printf("Contents of the stack\n");
    for (j=boundary[i]+1; j<=top[i]; j++)
    {
        printf("%d\n", s[j]);
    }
}
```

```
void display(int top[], int s[], int i,
                                int boundary[])
{
    int j;

    /* If stack is empty */
    if (top[i] == boundary[i])
    {
        printf("Stack %d is empty\n", i);
        return;
    }

    /* Display contents of stack */
    for (j=boundary[i]+1; j<=top[i]; j++)
    {
        printf("%d\n", s[j]);
    }
}
```

6.5.7 Implementing multiple stacks using global variables

The complete program to perform operations such as ***push***, ***pop*** and ***display*** using global variables is shown below:

Example 6.37: C Program to implement multiple stacks (Using global variables)

```
#include <stdio.h>
#include <stdlib.h>

#define     MAX_SIZE        20
int         s[MAX_SIZE];

#define     MAX_STACKS      10      // Maximum number of stacks
int         boundary[MAX_STACKS];
int         top[MAX_STACKS];
int         n;                      // Number of stacks entered by the user
int         i, item;                // stack number and element to be inserted
```

// Include : **Example 6.34:** To insert an item into a multiple stack

// Include : **Example 6.35:** To delete an item from multiple stack

// Include : **Example 6.36:** To display contents of multiple stack*/

```
void main()
{
        int     choice, j;

        printf("Enter number of queues (say 4 or 5)\n");
        scanf("%d", &n);

        for (j = 0; j <= n ; j++)
              boundary [j] = top[j] = MAX_SIZE / n * j – 1 ; /* stack is empty */

        for (;;)
        {
                printf("Stack number: ");
                for (j = 0; j < n; j++) printf("%d ", j);

                printf("Enter stack number: ");
                scanf("%d", &i);

                printf("1:Push    2:Pop\n");          printf("3:Display 4:Exit\n");
                printf("Enter the choice\n");         scanf("%d",&choice);

                switch(choice)
                {
                        case 1:
                                printf("Enter the item to be inserted\n");
                                scanf("%d",&item);
                                push();
                                break;

                        case 2:
                                item = pop();
                                if (item == -1)
                                        printf("Stack is empty\n");
                                else
                                        printf("Item deleted = %d\n", item);

                                break;
```

```
                    case 3:
                            display();
                             break;

                    default:
                            exit(0);
                }
        }
}
```

6.5.8 Implementing multiple stacks by passing parameters

The complete program to perform operations such as ***push***, ***pop*** and ***display*** by passing parameters is shown below:

Example 6.38: C Program to implement multiple stacks (by passing parameters)

```
#include <stdio.h>
#include <stdlib.h>

#define     MAX_SIZE        20
#define     MAX_STACKS      10      // Maximum number of stacks

// Include : Example 6.34: To insert an item into a multiple stack

// Include : Example 6.35:  To delete an item from multiple stack

// Include : Example 6.36:  To display contents of multiple stack

void main()
{
        int     choice;             /* user choice for push, pop and display */
        int     item;
        int     s[MAX_SIZE];
        int     boundary[MAX_STACKS];
        int     top[MAX_STACKS];
        int     n;                  // Number of stacks entered by the user
        int     i, j;

        printf("Enter number of queues (say 4 or 5)\n");
        scanf("%d", &n);
```

```
        for (j = 0; j <= n ; j++)
              boundary [j] = top[j] = MAX_SIZE / n * j – 1 ; /* stack is empty */

        for (;;)
        {
                printf(“Stack number: “);
                for (j = 0; j < n; j++) printf(“%d “, j);

                printf(“Enter stack number: ”);    scanf(“%d”, &i);

                printf("1:Push    2:Pop\n");
                printf("3:Display 4:Exit\n");
                printf("Enter the choice\n");       scanf("%d",&choice);

                switch(choice)
                {
                        case 1: printf("Enter the item to be inserted\n");
                                scanf("%d",&item);

                                push(item, s, top, boundary, i);

                                break;

                        case 2:  item = pop(s,  top, boundary,i);

                                 if (item == -1)
                                        printf("Stack is empty\n");
                                 else
                                        printf("Item deleted = %d\n",item);

                                break;

                        case 3:
                                display(top, s, i, boundary);
                                break;

                        default:
                                exit(0);
                }
        }
}
```

6.6 Recursion

Recursion is a powerful tool but least understood by most novice students. Programming languages such as Pascal, C, C++ etc support recursion. Now, let us see "What is recursion? What are the various types of recursion?"

Definition: A recursion is a method of solving the problem where the solution to a problem depends on solutions to smaller instances of the same problem. Thus, a recursive function is a function that calls itself during execution. This enables the function to repeat itself several times to solve a given problem. The various types of recursion are shown below:

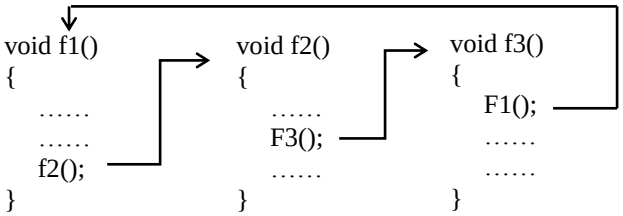
- Direct recursion
- Indirect recursion

Direct recursion: A recursive function that invokes itself is said to have direct recursion. For example, the factorial function calls itself (detailed explanation is given later) and hence the function is said to have direct recursion.

```
int fact (int  n)
{
    if ( n == 0 ) return 1;

    return n*fact(n-1);
}
```

Direct recursion

- **Indirect recursion:** A function which contains a call to another function which in turn calls another function which in turn calls another function and so on and eventually calls the first function is called indirect recursion. It is very difficult to read, understand and find any logical errors in a function that has indirect recursion. For example, a function f1 invokes f2 which in turn invokes f3 which in turn invokes f1 is said to have indirect recursion. This is pictorially represented as shown below:



Indirect recursion

Now, the question is "How to design recursive functions?" Every recursive call must solve one part of the problem using base case or reduce the size (or instance) of the problem using general case. Now, let us see "What is base case? What is a general case?"

Definition: A base case is a special case where solution can be obtained without using recursion. This is also called base/terminal condition. Each recursive function must have a base case. A base case serves two purposes:

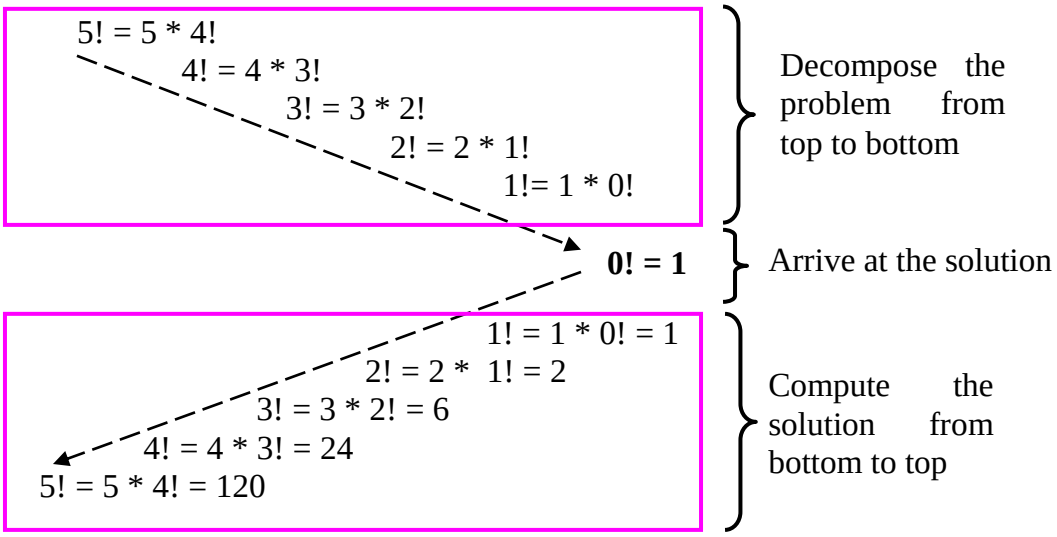
1) It acts as terminating condition.
2) The recursive function obtains the solution from the base case it reaches.

Definition: In any recursive function, the part of the function except base case is called general case. This portion of the code contains the logic required to reduce the size (or instance) of the problem so as to move towards the base case or terminal condition. Here, each time the function is called, the size (or instance) of the problem is reduced.

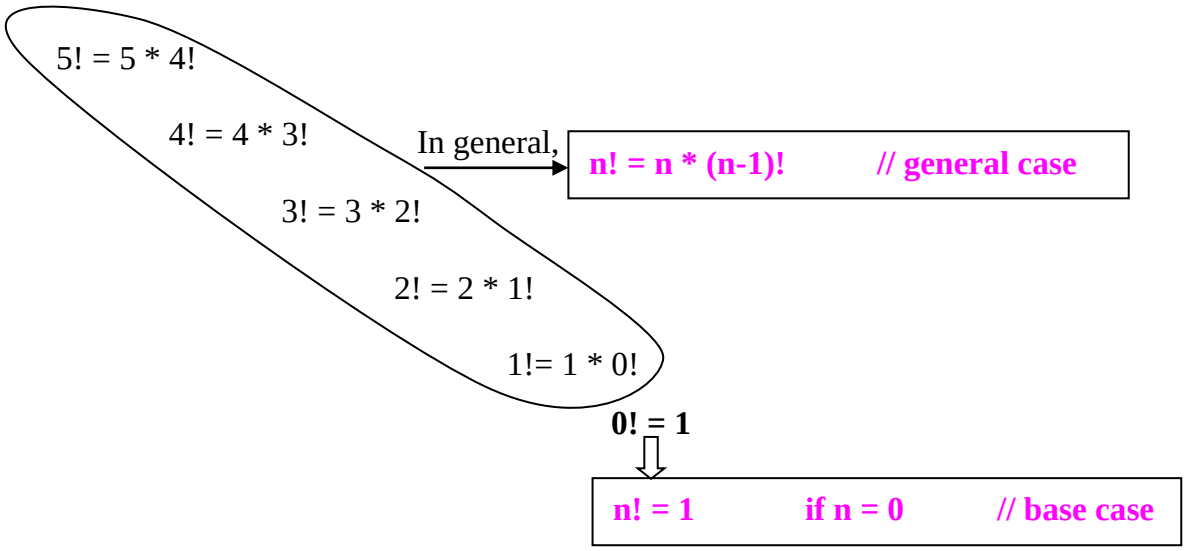
Note: A recursive function should never generate infinite sequence of calls on itself. An algorithm exhibiting this sequence of calls will never terminate and hence it is called infinite recursion and will crash the system.

6.6.1 Compute factorial of n

We all know the function to compute factorial of a given number using iteration. Now, let us see "How to compute factorial of 5 using recursion?" We can compute 5! as shown below:



Observe that computation of 5! is postponed and it is decomposed into computing factorials of 4, 3, 2, 1 as shown below till we get 0! which is 1.



Thus, by looking at the above base case and general case, the recursive definition to find factorial of *n* can be written as shown below:

n!= 1	if n == 0
n! = n * (n-1)!	otherwise

or

$$n! = \begin{cases} 1 & \text{if } n == 0 \\ n * (n-1)! & \text{otherwise} \end{cases}$$

The above definition can also be written as shown below:

$$F(n) = \begin{cases} 1 & \text{if } n == 0 \\ n * F(n-1) & \text{otherwise} \end{cases}$$

Output: 5 * 4 * 3 *2 * 1

Exchanging the operands, Output: 1 * 2 * 3 * 4 * 5

Using the above recursive definition, we can write the function as shown below:

Example 6.39: Recursive C function to find the factorial of N

```
int fact(int n)
{
        if ( n == 0 )    return 1;          /* factorial of n when n = 0 */
        return           n*fact(n-1);       /* factorial of n when n > 0 */
}
```

The above function can be invoked as shown below:

Example 6.40: C program to compute factorial of n

```
#include <stdio.h>

/* Include: Example 6.39: Function to compute factorial of n */

void main()
{
        int     n;
        float   res;

        printf("Enter n         \n");
        scanf("%d %d",&n, &r);

        res = fact(n);
        printf("%d! = %d\n", n, res);
}
```

Input

```
Enter n
    5
```

Output

```
5! = 120
```

So, the general rules that we are supposed to follow while designing any recursive algorithm are:

- Determine the base case. Careful attention should be given here, because: when base case is reached, the function must execute a return statement without a call to recursive function.
- Determine the general case. Here also careful attention should be given and see that each call must reduce the size of the problem and moves towards base case and must reach base case
- Combine the base case and general case into a function.

6.6.2 How recursion works

Let us visualize how the same function is called repeatedly using recursion. The figure shows exactly what happens when recursive function **fact()** gets called. The execution sequence for the above factorial function is shown in figure 6.4.

a. The first time when function **fact()** is called, 3 is passed to **n**.

b. Since **n** is not 0, the if-condition fails and **fact()** is called with argument **n-1** i.e., 2.

c. Since **n** is not 0, the if-condition fails and **fact()** is called again with argument **n-1** i.e., 1.

d. Since **n** is not 0, the if-condition fails and **fact()** is called again with argument **n-1** i.e., 0.

e. Since **n** is 0, control returns to previous call with value 1 and 1*1 = 1 is computed and sent to the previous point of invocation.

f. The resulting value **1** is passed to the point of previous invocation and 2*1 = 2 is computed and sent to the previous point of invocation.

g. The resulting value **2** is passed to the point of previous invocation and 3*2 = 6 is computed and sent to the previous point of invocation.

h. The resulting value **6** is passed to the point of previous invocation and **6** is copied into variable **n** in the main program.

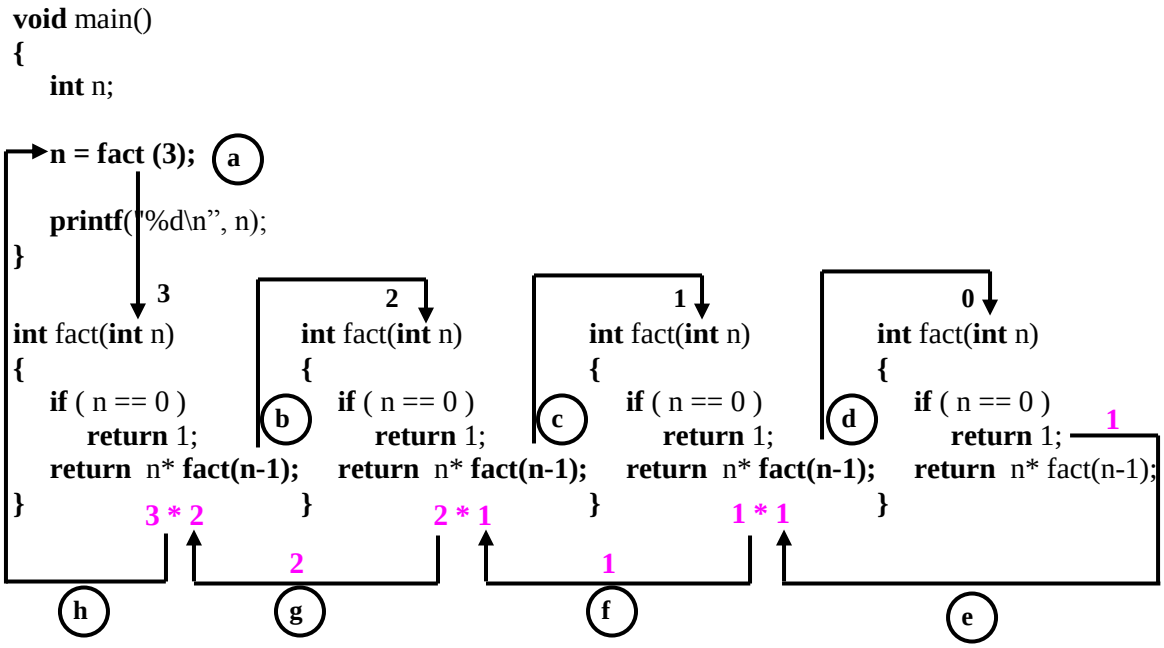


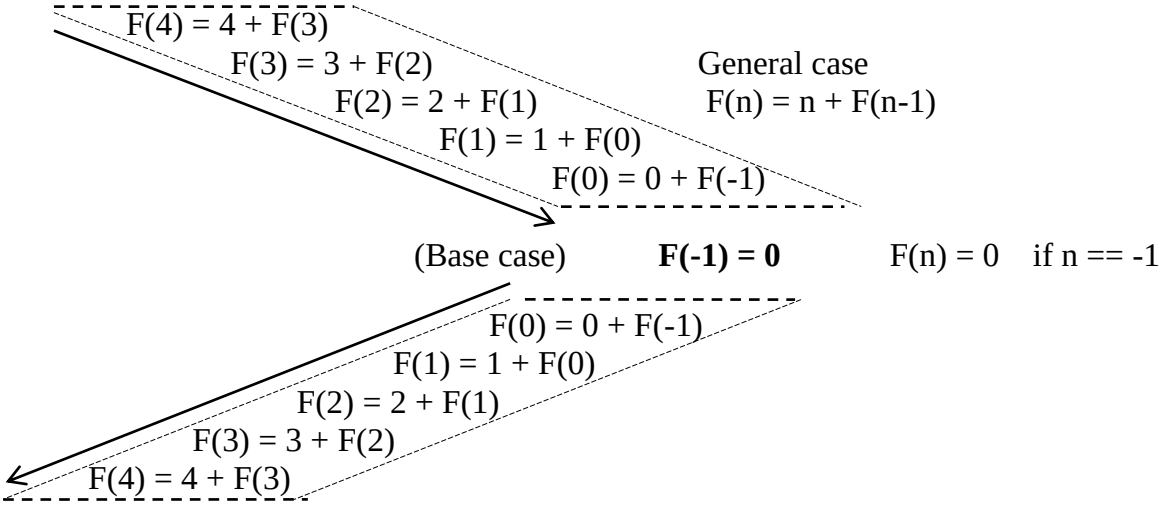
Fig 6.4 Steps in executing a recursive function

Note: A recursive function should never generate infinite sequence of calls on itself. An algorithm exhibiting this sequence of calls will never terminate.

6.6.3 Sum of natural numbers

Consider the sum of series: 4 + 3 + 2 + 1 + 0

The sum of above series can be computed recursively as shown below:



So, the recursive definition can be written as shown below:

$$F(n) = \begin{cases} 0 & \text{if } n == -1 \\ n + F(n-1) & \text{otherwise} \end{cases}$$

Example 6.41: Recursive C function to find the sum of series

```
// 4 + 3+ 2 + 1 + 0 where n = 4
int F(int n)
{
        if ( n == -1 )  return 0;
        return          n + F(n-1);
}
```

OR

```
// 0 + 1 + 2 + 3 + 4 where n = 4
int F(int n)
{
        if ( n == -1 )  return 0;
        return          F(n-1) + n;
}
```

Note: If we exchange the terms in the expression n + F(n-1) shown using dotted rectangular box as: F(n-1) + n, we get the sum of the series: 0 + 1 + 2 + 3 + 4.

6.6.4 Sum of array elements

Note: By inserting an array *a* as the parameter and changing *n* to a[n] for the above recursive relation, we get sum of following series:

$$F(a, n) = a[4] + a[3] + a[2] + a[1] + a[0]$$

So, the recursive definition can be written as shown below:

$$F(a,n) = \begin{cases} 0 & \text{if } n == -1 \\ a[n] + F(a, n-1) & \text{otherwise} \end{cases}$$

Example 6.42: Recursive function to find sum of array elements from a[n-1] to a[0]

```
// sum = a[4] + a[3] + a[2] + a[1] + a[0]
float F(float a[], int n)
{
        if ( n == -1 )   return 0;

        return           a[n] + F(a, n-1);
}
```

```
// sum = a[0] + a[1] + a[2] + a[3] + a[4]
float F(float a[], int n)
{
        if ( n == -1 )   return 0;

        return           F(a,n-1) + a[n];
}
```

Example 6.43: Program to find the sum of array elements

```
#include <stdio.h>

// Insert: Example 6.42 : Function to find sum of a[4] + a[3] + a[2] + a[1] + a[0]
                                    or
// Insert: Example 6.42: Function to find sum of a[0] + a[1] + a[2] + a[3] + a[4]

void main()
{
        int      n,  i;
        float    a[10], sum;

        printf ("Enter the number of elements\n");          // read number of items
        scanf (%d, &n);

        printf (“Enter %d items\n”, n);                     // read n items
        for (i = 0; i < n; i++) scanf(“%d”, &a[i]);

        sum = F(a, n-1);                                    // find sum of n items

        printf(“Result = %d\n”, sum);                       // print the result
}
```

6.6.5 Print array elements

Consider the following array elements:

a[0]	[1]	[2]	[3]	[4]
10	20	30	40	50

Note: Changing a[n] to "print a[n]" and removing the + symbol in the above program, we can print array elements. So, the recursive definition to print array elements in reverse order can be written as shown below:

$$F(a,n) = \begin{cases} 0 & \text{if } n == -1 \\ \text{print}(a[n]) \text{ , } F(a, n-1) & \text{otherwise} \end{cases}$$

Output: 50 40 30 20 10

Exchanging the operands, Output: 10 20 30 40 50

Note: Replacing name of function F() to PrintArray(), we can write the function as shown below:

Example 6.44: Recursive function to print array elements from a[0] to a[n-1]

```
void PrintArray (float a[], int n)
{
        if ( n == -1 )   return;

        PrintArray (a, n-1);
        printf ("%d\n", a[n]);
}
```

The complete program can be written as shown below:

Example 6.45: Program to print array elements

```
#include <stdio.h>

// Insert: Example 6.44: Function to print array elements

void main()
{
        int     n, i;   float   a[10];

        printf("Enter the number of elements\n");          // read number of items
        scanf (%d", &n);

        printf("Enter %d elements\n");                     // read n items
        for (i = 0; i < n; i++) scanf("%d", &a[i]);

        PrintArray(a, n-1);                                // print array elements
}
```

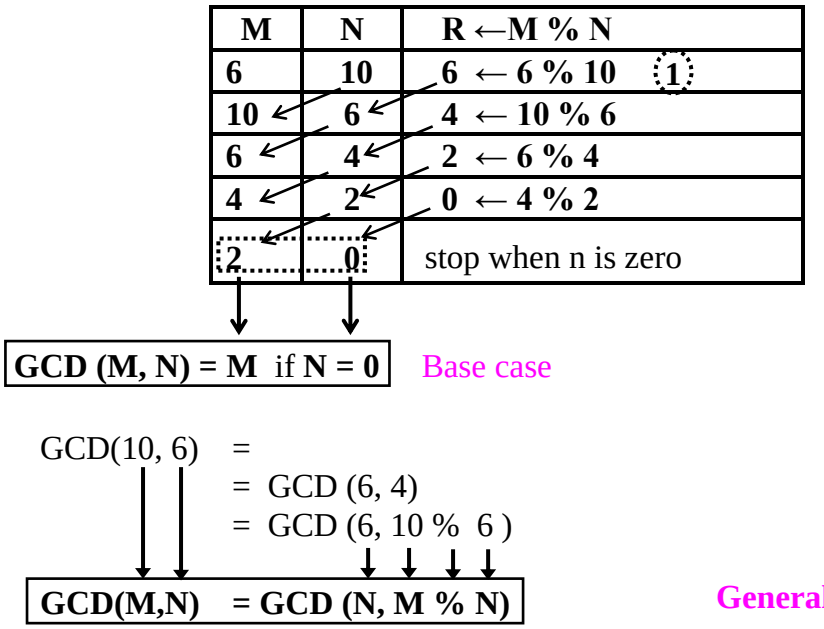
6.6.6 GCD – Greatest Common Divisor of two numbers (Euclid's algorithm)

Now, let us see "What is GCD of two numbers?" What is the recursive definition to compute GCD of two numbers?"

Definition: The GCD of two given numbers is the largest integer that divides both of them. GCD of two numbers is defined only for positive integers but, not defined for negative integers and floating point numbers.

Now, let us "Write a recursive function to find GCD of two numbers using Euclid's algorithm" The recursive solution for Euclid's algorithm can be obtained as shown below:

Design recursive algorithm: The procedure to obtain GCD(6, 10) using Euclid's algorithm is shown below:



Now, the recursive definition to compute GCD of two numbers M and N using EUCLID's algorithm can be written as shown below:

$$GCD(M, N) = \begin{cases} M & \text{if } N = 0 \\ GCD(N, M \% N) & \text{otherwise} \end{cases}$$

Base case

General case

Using the above recursive definition, we can write the recursive function as shown below:

Example 6.46: Recursive function to compute GCD of two numbers

```
int gcd(int m, int n)
{
        if (n == 0) return m;

        return gcd(n, m % n);
}
```

The main program to compute GCD of two numbers is shown below:

Example 6.47: C Program to compute GCD of two numbers (Euclid's algorithm)

```
#include <stdio.h>

/* Include: Example 6.46: To compute gcd of two numbers */

void main()
{
        int m, n, res;

        printf("Enter the value of m and n\n");
        scanf("%d %d",&m, &n);

        res = gcd(m, n);

        printf("gcd(%d, %d) = %d\n", m, n, res);
}
```

Input

```
Enter the value of m and n
10 6

res = 2
```

Output

```
gcd(10, 6) = 2
```

6.6.7 Fibonacci series

Now, let us see "What are Fibonacci numbers?"

Definition: The Fibonacci numbers are a series of numbers such that each number is the sum of the previous two numbers except the first and second number. These numbers are named after an Italian mathematician "Leonardo Fibonacci" who lived in early thirteenth century. The Fibonacci numbers are shown below:

0, 1, 1, 2, 3, 5, 8, 13, ……

Now, let us see "What is the recursive definition to compute a Fibonacci number?"

To write a recursive definition, we should know the base case and general case which can be computed as shown below:

fibonacci numbers

F[0] = 0 ⇨ F[n] = 0 if n = 0
F[1] = 1 ⇨ F[n] = 1 if n = 1

Base case

F[2] = F[1] + F[0] = 1 + 0 = 1
F[3] = F[2] + F[1] = 1 + 1 = 2
F[4] = F[3] + F[2] = 2 + 1 = 3

General case

In general, F[n] = F[n-1]+F(n-2) ⇨ F(n) = F(n-1)+F(n-2) otherwise

Now, using the above base case and general case, the recursive definition to find n^{th} Fibonacci number can be written shown below:

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

The algorithm to find the n^{th} Fibonacci number can be written using recursive C function as shown below:

Example 6.48: C Function to find nth Fibonacci number

```
int F(int n)
{
        if ( n == 0 ) return 0;          /* Base case: 1st number */

        if ( n == 1 ) return 1;          /* Base case: 2nd number */

        return F(n-1) + F (n-2);         /* General case: Computation of */
}                                        /* subsequent Fibonacci numbers */
```

The C program to display nth Fibonacci number is shown below:

Example 6.49: C program to display nth Fibonacci number

```
#include <stdio.h>

/* Include: Example 6.48: To compute nth Fibonacci number */

void main()
{
        int n;

        printf("Enter n\n");
        scanf("%d",&n);

        printf("fib(%d) = %d\n",n,F(n));
}
```

Input

```
Enter n
6
```

Output

```
Fib(6) = 8
```

The function F() can also be used to generate *n* Fibonacci numbers as shown below:

Example 6.50: C program to display n Fibonacci numbers

```
#include <stdio.h>

/* Include: Example 6.48: To compute nth Fibonacci number */

void main()
{
        int i, n;

        printf("Enter n\n");
        scanf("%d",&n);

        printf("%d Fibonacci numbers are\n", n);
        for (i = 0; i < n; i++)
        {
                printf("fib(%d) = %d\n",i,fib(i));
        }
}
```

Input

```
Enter n
6
```

Output

```
6 Fibonacci numbers are
fib(0) = 0
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
```

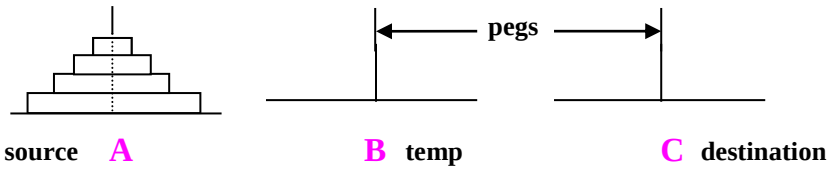
6.6.8 Tower of Hanoi

Let us see, "What is tower of Hanoi problem?" In this problem, there are three pegs say **A**, **B** and **C**. The different diameters of **n** discs are placed one above the other

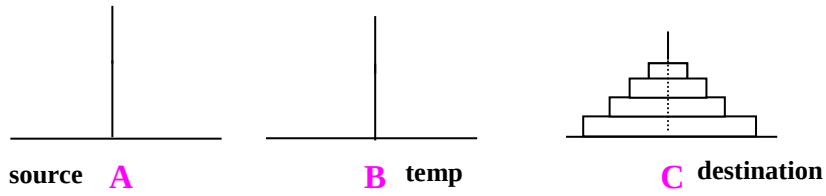
through the peg **A** and the discs are placed such that always a smaller disc is placed above the larger disc. The two pegs B and C are empty. All the discs from peg A are to be transferred to peg C using peg B as temporary storage. The rules to be followed while transferring the discs are

- Only one disc is moved at a time from one peg to another peg
- Smaller disc is on top of the larger disc at any time.
- Only one peg can be used to for storing intermediate discs

The initial set up of the problem is shown below.



After transferring all discs from **A** to **C** we get the following setup.



Design: Let us assume we have to transfer *n* discs from *source s* to *destination d* using *temp t* as temporary peg. The function header can be written as follows:

Function header: `void transfer (int n, char s, char t, char d)`
where

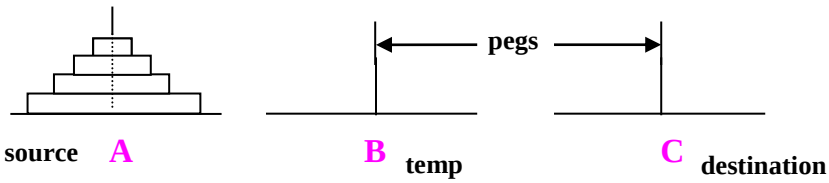
n: number of discs to be transferred
s: source peg
t : temporary peg
d: destination peg

The various actions to be taken during this transfer of discs results in following cases:

Case 1: When there are no discs: This situation occurs when *n* is 0. When there are no discs i.e., when *n* is 0, no action takes place and we simply return. The code for this can be written as shown below:

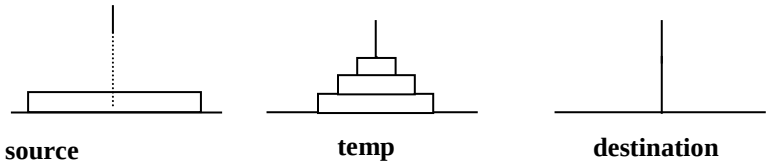
```
if ( n == 0) return;
```

Case 2: When there are some discs: Suppose there are 4 discs in the source peg. The initial set up when *n* is 4 is shown below:



Now, it is required to transfer all *n* discs from *source* to *destination* recursively. This can be done using following sequence of operations:

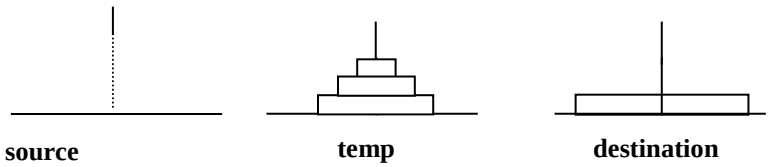
Step 1: Move n-1 discs recursively from *source* to *temp*. In our example we have to move 3 discs from *source* to *temp*. Observe that *temp* has to be 4th parameter when the function is called since *temp* is the destination. This results in following scenario:



This activity can be achieved by calling the function *transfer()* recursively as shown below:

```
transfer (n-1, s, d, t);          // transfer n-1 discs from source to destination
```

Step 2: Move nth disc from *source* to *destination*. This results in following scenario:



This activity can be achieved by moving the nth disc from source *s* to destination *d*. This can be achieved using the following statement:

```
printf("Move disc %d from %c to %c\n", n, s, d); // Move nth disc from source to destination
```

Step 3: Move n-1 discs recursively from *temp* to *destination*. In our example we have to move 3 discs from *temp* to *destination* recursively. Observe that *temp* has to be 2nd parameter when the function is called since *temp* is the source peg. This results in following scenario:

This activity can be achieved by calling the function *transfer()* recursively as shown below:

```
transfer (n-1, t, s, d);          // transfer n-1 discs from temp to destination
```

The C function to implement tower of Hanoi problem can be written as shown below:

Example 6.51: C function to implement tower of Hanoi problem

```
void transfer (int n, char s, char t, char d)
{
    if ( n == 0 )  return;                                  // Base case

    transfer (n-1, s, d, t);                                // Move n-1 discs from source to temp

    printf("Move disc %d from %c to %c\n", n, s, d); // Move nth disc from source to destination

    transfer (n-1, t, s, d);                                // Move n-1 discs from temp to destination
}
```

The complete program to implement tower of Hanoi problem is shown below:

Example 6.52: C Program to implement tower of Hanoi problem

```
#include <stdio.h>

/* Include: Example 6.51: C function to transfer disks from source to destination */

void main()
{
    int n;

    printf("Enter the number of discs\n");
    scanf("%d",&n);

    /* Transfer n disks from peg A to peg C */
    tower(n, 'A', 'B', 'C');
}
```

Output

```
Move disc 1 from A to C
Move disc 2 from A to B
Move disc 1 from C to B
Move disc 3 from A to C
Move disc 1 from B to A
Move disc 2 from B to C
Move disc 1 from A to C
Move disc 3 from A to C
Move disc 1 from B to A
Move disc 2 from B to C
Move disc 1 from A to C
```

6.6.9 Binomial co-efficient

We know that a binomial formula is given by

$$(a+b)^n = {}^nc_0a^nb^0 + {}^nc_1a^{n-1}b^1 + {}^nc_2a^{n-2}b^2 + \ldots\ldots {}^nc_ia^{n-i}b^i + \ldots\ldots {}^nc_na^{n-n}b^n$$

Here, ${}^nc_0$, ${}^nc_1$, ${}^nc_2$, $\ldots\ldots{}^nc_i, \ldots\ldots$ ${}^nc_n$ are called binomial coefficients. The two useful properties of binomial coefficients are:

$${}^nC_k = \begin{cases} 1 & \text{if } k = 0 \text{ or } n = k \\ {}^{n-1}C_{k-1} + {}^{n-1}C_k & \text{otherwise} \end{cases}$$

Note: The value of *n* cannot be 0. The above recurrence relation can also be written as

$$C(n, k) = \begin{cases} 1 & \text{if } k = 0 \text{ or } n = k \\ C(n-1, k-1) + C(n-1, k) & \text{for } n > k > 0 \end{cases}$$

For the above recursive relation, we can write the recursive function as shown below:

Example 6.53: C function to compute binomial coefficient

```
int C (int n, int k)
{
        if (k == 0 || k == n)    return 1;

        return C(n – 1, k – 1) + C (n – 1, k );
}
```

The C program to compute binomial co-efficient can be written as shown below:

Example 6.54: C Program to compute binomial co-efficient

```
#include <stdio.h>

/* Include: Example 6.53: C function to compute binomial co-efficient */

void main()
{
        int n, k, res;

        printf("Enter N and K\n");              // Enter N and K
        scanf("%d %d", &n, &k);                 // 6   3

        res = C(n, k);                          //  res = 20

        printf("%d C %d = %d\n", n, k, res);    // 6C3 = 20
}
```

6.6.10 Ackermann's function

The Ackermann's function is recursively defined as shown below:

$$A(M, N) = \begin{cases} N+1 & \text{if } M = 0 \quad \text{Case 1} \\ A(M-1, 1) & \text{if } N = 0 \quad \text{Case 2} \\ A(M-1, A(M, N-1)) & \text{otherwise} \quad \text{Case 3} \end{cases}$$

In the above recursive relation, Case 1 is the base case and Case 2 and Case 3 are general cases. Using Ackermann's function, A(1,2) can be evaluated as shown below:

```
A(1, 2)= A(0, A(1, 1))          (Case 3)
       = A(0, A(0, A(1, 0)))    (Case 3)
       = A(0, A(0, A(0, 1)))    (Case 2)
       = A(0, A(0, 2))          (Case 1)
       = A(0, 3)                (Case 1)
       = 4                      (Case 1)
```

For the above recursive relation, we can write the recursive function as shown below:

Example 6.55: C function to compute value of Ackermann's function

```
int A (int m, int n)
{
        if (m == 0) return n + 1;                    // Case 1

        if (n == 0) return A (m – 1, 1);             // Case 2

        return A( m – 1, A (m, n – 1));              // Case 3
}
```

The C program to invoke the above function to compute and print the value of Ackermann's function can be written as shown below:

Example 6.56: C Program to compute and print the value of Ackermann's function

```
#include <stdio.h>

/* Include: Example 6.55: C function to compute value of Ackermann's function */

void main()
{
        int m, n, res;

        printf("Enter M and N\n");                   // Enter N and K
        scanf("%d %d", &m, &n);                      // 1   2

        res = A(m, n);                               //  res = 4

        printf("A(%d, %d) = %d\n", m, n, res);       // A(1, 2) = 4
}
```

6.7 System stack

Now, let us see "What is system stack? How the system stack is used to process a function call?" A stack used by a program at run-time (that is when the program is being executed) is called system stack. The system stack is used when functions are invoked and executed:

- When a function is called, a structure called an activation record is created on top of the stack
- When a function is terminated, the activation record is deleted from the top of the system stack.

6.68 Stacks

Now, let us see "What is an activation record?"

Definition: An activation record also called stack frame is a structure consisting of various fields to store: *local variables (if any), parameters (if any), return address* and *pointer to the previous stack frame.* Using the activation record, the functions can communicate each other.

Now, let us see "How the control is transferred to or from the function with the help of activation record?"

The user gives the command to execute the program. Then, the OS calls the function *main()* and the body of the function *main()* is executed. After executing the body of the function *main()*, the control is returned to the operating system. The various actions that take place when we execute the program are shown below:

Step 1: Operating system transfers the control to function main() along with return address so that after executing the body of the function main(), the control is returned to OS.

Step 2: A stack frame or activation record is created consisting of:

- return address of OS
- local variables of main()
- previous frame pointer which is NULL (denoted by 0)

This stack frame is pushed onto the stack.

Step 3: Now, main() is executed.

Step 4: When return statement is executed in the main(), the activation record is removed from the stack. Using the removed activation record, we can get the return address of OS using which the control is transferred to OS.

Analogy: The above facts can be explained using the following analogy: We post the letter by writing the sender's address to whom it is to be delivered and we also write our address (also called return address). The receiver stores the return address written on the letter and uses this address to communicate back. On similar lines, control is transferred to or from the function.

Step 5: When main is being executed, if the function a1() is invoked, a new stack frame or activation record is created consisting of:

- return address of main()
- local variables of a1()
- current frame pointer which is on top of the stack

This new stack frame is pushed onto the stack.

Note: Observe that previous frame pointer of activation record which is on top of the stack points to the activation record of main().

Step 6: Control is transferred to function a1().

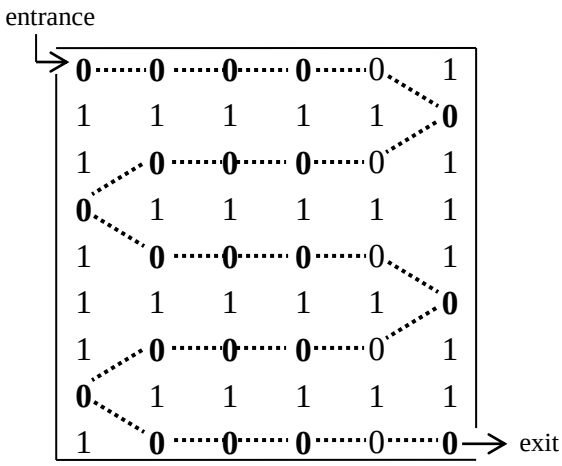
Step 7: When return statement in a1() is executed, the activation record on top of the stack is removed. Using the removed activation record, we can get the return address of main using which the control is transferred to main.

Step 8: Finally, when body of the function main is completely executed, the return address of OS is obtained from the activation record and control is transferred to OS.

6.8 Maze Problem

Mazes have been very interesting subject for many years. In this section, let us develop a program that runs a maze. Now, let us see "What is a maze?"

Definition: A maze is a confusing network of paths through which it is very difficult to find one's way. The experimental psychologists train the rats to search mazes for food. A maze problem is a nice application of stack.



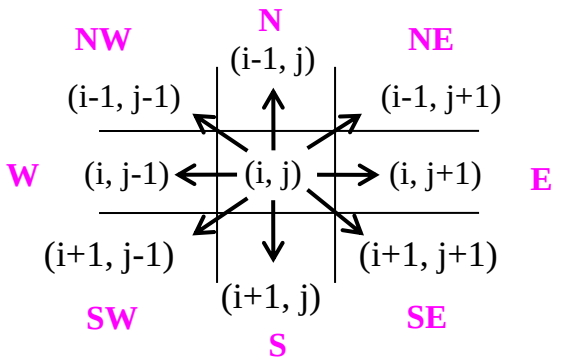
Before developing a program let us see "How maze is represented?" A maze is represented as a two dimensional array in which

- zeros represent the open paths
- ones represent barriers.

An example for a simple maze with long path is shown in above figure.

Note: Observe from the above maze that, the rat starts at the top left corner of the maze and leaves at bottom right of the maze.

Now, let us see "What is the disadvantage of representing a maze of size is m x n?" If (i, j) is the position of the rat in a maze, the following figure shows the possible moves along with new positions:



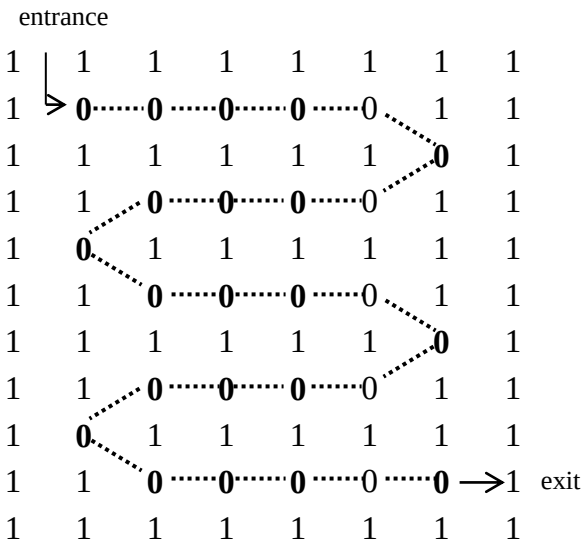
N – **N**orth
NW – **N**orth **W**est
NE – **N**orth **E**ast
W – **W**est
E – **E**ast
SW – **S**outh **W**est
S – **S**outh
SE – **S**outh **E**ast

Let (i, j) is the current position of rat in the maze where *i* is the row index and *j* is the column index. Now, the rat moves in 8 directions as shown in above figure. The new position of rat in the maze can be obtained as shown below:

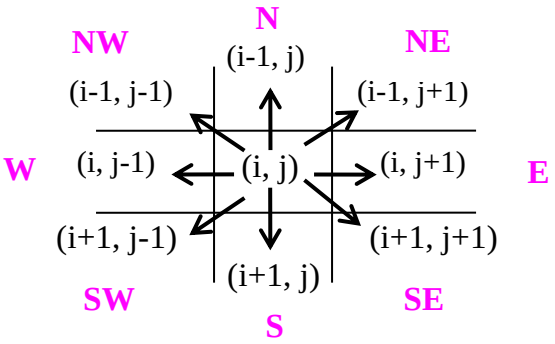
- North (vertical upwards) (i –1, j)
- South (vertical downwards) (i+1, j)
- West (horizontal right) (i, j + 1)
- East (horizontal left) (i, j – 1)
- North east (vertical upwards and horizontal right) (i – 1, j + 1)
- North west(vertical upwards and horizontal left) (i – 1, j – 1)
- South east (vertical downwards and horizontal right) (i + 1, j + 1)
- South west (vertical downwards and horizontal left) (i + 1, j – 1)

The rat can be moved in all 8 directions. But, this is not true all the time. If the position of rat is on any of the border, then it cannot move in all 8 directions. The checking for border conditions can be avoided as shown below:

Surround the maze by a border of ones. Thus a maze of size m x n requires (m + 2) x (n + 2) size. The maze shown earlier now can be surrounded by a border of ones as shown in the figure.



Note: The rat starts at the top left corner of the maze from the position (1, 1) and leaves at bottom right of the maze from the position (9, 6)
Now, let us obtain the next move of the rat. Given the position (i, j) the various position in which rat can move are shown below:



N – **N**orth
NW – **N**orth **W**est
NE – **N**orth **E**ast
W – **W**est
E – **E**ast
SW – **S**outh **W**est
S – **S**outh
SE – **S**outh **E**ast

Observe from above figure that either we add -1 or 0 or 1 to (i, j) to get the new position. These values that we add to find the new position are called offset values. The offset values of various moves can be obtained as shown below:

- (N) North position = (i-1, j) = (i-1, j+0) Offset value = -1, 0
- (NE)North east position = (i-1, j+1) Offset value = -1, 1
- (E) East position = (i, j+1) = (i+0, j+1) Offset value = 0, 1
- (SE)South east position = (i+1, j+1) Offset value = 1, 1
- (S)South position = (i+1, j) = (i+1, j+0) Offset value = 1, 0
- (SW)South west position = (i+1, j-1) Offset value = 1, -1
- (W)West position = (i, j-1) = (i+0, j-1) Offset value = 0, -1
- (NW)North west position= (i-1, j-1) Offset value = -1, -1

The above 8 directions can be represented using the numbers from 0 to 7 along with above offset values in the form of a table as shown below:

Name	dir	move[dir].vert	move[dir].horiz
N	0	-1	0
NE	1	-1	1
E	2	0	1
SE	3	1	1
S	4	1	0
SW	5	1	-1
W	6	0	-1
NW	7	-1	-1

The above table can be initialized using array of structures as shown below:

```
typedef struct
{
        int     vert;   /* 0: No vertical movement, 1:Vertical down, -1:Vertical up */
        int     horiz;  /* 0: No horizontal movement, 1:Horizontal right */
} offsets;              /* -1: Horizontal left */

//Offset value to move:N      NE      E       SE      S       SW      W       NW
offsets move[8] = {{-1,  0}, {-1,  1},{0,  1},{1,  1}, {1,  0},{1,  -1},{0,  -1},{-1, -1}};
```

Initialization: There is always an entrance for the rat at position (1, 1). So, mark this initial position of rat and save it on the stack using the following statements as shown below:

```
mark[1][1] = 1;                 /* Mark the path through (1, 1) */
pos.row = 1, pos.col = 1;       /* Rat starts from (1, 1) */
pos.dir = 0;                    /* dir : 0 – Movement is north */

s[top=0] = pos;                 /* Push the current position of rat */
```

Now, take the valid position of rat in the maze which is on top of the stack using the statements:

```
pos     = s[top--];             /* Obtain the position of rat from stack*/

row     = pos.row;              /* Make it the current position of rat */
col     = pos.col;
dir     = pos.dir;
```

To find more moves from the current position:

Now, from the current position (row, col) we can find the next valid position of rat in the maze at a particular direction using the following code:

```
nextRow  = row + move[dir].vert;
nextCol  = col  + move[dir].horiz;
```

If the above position is same as exit point, then we have the path and we stop searching. The code for this can be:

```
if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
    found = TRUE;
```

If the above condition is false, we check whether the new position is not a barrier. If there is no barrier and if the new position is not reached earlier, then mark that position as save it on the stack using the code:

```
if (maze[nextRow][nextCol]==0 && mark[nextRow][nextCol] == 0)
{
    mark[nextRow][nextCol] = 1;
    pos.row = row, pos.col = col, pos.dir = ++dir;

    s[++top] = pos;
    row = nextRow, col = nextCol, dir = 0;
}
```

If above condition fails, then we have to go in the next direction by incrementing the value of *dir* as shown below:

```
++dir;
```

The complete maze program along with output is shown below:

Example 6.57: C program for maze problem

```
#include <stdio.h>

typedef struct
{
        int     vert;
        int     horiz;
} offsets;

offsets move[8] = {{-1,  0}, {-1,  1},{0,  1},{1,  1}, {1,  0},{1,  -1},{0,  -1},{-1, -1}};

typedef struct
{
        int     row;
        int     col;
        int     dir;
} element;

enum  {FALSE, TRUE};

#define EXIT_ROW 12
#define EXIT_COL  15
```

```
int maze[20][20]= {          // Maze for the rat move */
                    {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
                    {1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1},
                    {1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1},
                    {1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1},
                    {1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1},
                    {1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
                    {1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1},
                    {1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1},
                    {1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
                    {1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1},
                    {1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1},
                    {1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1},
                    {1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1},
                    {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
                  };

void main()
{
    int         row, col, nextRow, nextCol, dir, found = FALSE, top, i;
    int         mark[20][20] = {{0}};
    element     s[200],  pos;

    mark[1][1] = 1;
    pos.row = 1, pos.col = 1;                   /* Rat starts from (1, 1) */
    pos.dir = 0;                                /* dir : 0 – Movement is north */

    s[top=0] = pos;                             /* Push the current position of rat */

    while (top != -1 && !found)
    {
        pos     = s[top--];                     /* Obtain the position of rat from stack*/
        row     = pos.row;                      /* Make it the current position of rat */
        col     = pos.col;
        dir     = pos.dir;

        while (dir < 8 && !found)
        {
            nextRow   = row + move[dir].vert;
            nextCol    = col  + move[dir].horiz;
```

```
            if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
                found = TRUE;
            else if (maze[nextRow][nextCol]==0 && mark[nextRow][nextCol] == 0)
            {
                mark[nextRow][nextCol] = 1;
                pos.row = row, pos.col = col, pos.dir = ++dir;

                s[++top] = pos;
                row = nextRow, col = nextCol, dir = 0;
            }
            else
                 ++dir;
        }
    }

    if (found)
    {
        printf("Path is\n");
        printf("Row   Col\n");
        for (i = 0; i <= top; i++)
        {
            printf("%3d %3d\n",s[i].row, s[i].col);
        }
        printf("%3d %3d\n", row, col);
        printf("%3d %3d\n",EXIT_ROW, EXIT_COL);
    }
    else
        printf("Maze does not have a path\n");
}
```

Output

Path is:

(1, 1)	(2, 2)	(1, 3)	(1, 4)	(1, 5)	(2, 4)
(3, 5)	(3, 4)	(4, 3)	(5, 3)	(6, 2)	(7, 2)
(8, 1)	(9, 2)	(10, 3)	(10, 4)	(9, 5)	(8, 6)
(8, 7)	(9, 8)	(10, 8)	(11, 9)	(11, 10)	(10,11)
(10,12)	(10,13)	(9, 14)	(10,15)	(11,15)	(12,15)

By looking at the above positions, we can find the path from entry to exist and is shown using dotted lines in the following maze.

Maze 2: Initialize the matrix *maze* in the above program change symbolic constants EXIT_ROW and EXIT_COL as shown below:

```
int maze[20][20] =
{
        {1, 1, 1, 1, 1, 1, 1, 1},
        {1, 0, 0, 0, 0, 0, 1, 1},
        {1, 1, 1, 1, 1, 1, 0, 1},
        {1, 1, 0, 0, 0, 0, 1, 1},
        {1, 0, 1, 1, 1, 1, 1, 1},
        {1, 1, 0, 0, 0, 0, 1, 1},
        {1, 1, 1, 1, 1, 1, 0, 1},
        {1, 1, 0, 0, 0, 0, 1, 1},
        {1, 0, 1, 1, 1, 1, 1, 1},
        {1, 1, 0, 0, 0, 0, 0, 1},
        {1, 1, 1, 1, 1, 1, 1, 1}
};
```

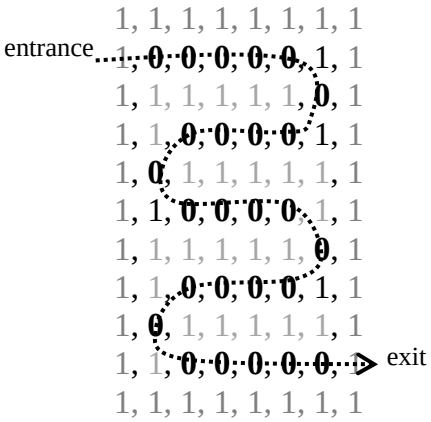
9 rows

6 columns

```
#define EXIT_COL 6
#define EXIT_ROW 9
```

After executing, the path is:
(1, 1) (1, 2) (1, 3) (1, 4) (1, 5) (2, 6) (3, 5) (3, 4) (3, 3) (3, 2) (4, 1) (5, 2) (5, 3) (5, 4) (5, 5) (6, 6) (7, 5) (7, 4) (7, 3) (7, 2) (8, 1) (9, 2) (9, 3) (9,4) (9, 5) (9, 6)

By looking at the above positions, we can find the path from entry to exist and is shown using dotted lines in the following maze.



Exercises

1) What is a stack? What are the various operations that can be performed on stacks?
2) What is push operation? What is stack overflow?
3) How to implement push operation using arrays (static allocation technique)?
4) What is pop operation? What is stack underflow?"
5) How to implement pop operation using arrays (static allocation technique)?
6) Write a C function to display the contents of stack (using global variables)
7) Write a C function to display contents of the stack (by passing parameters)
8) What is a palindrome? Write a C function to check for a palindrome using stack
9) What are the applications of stack?
10) What is an expression? What are the various types of expressions?
11) What is postfix expression? What is prefix expression?
12) How to write a program to convert a given infix expression to its equivalent prefix expression?
13) What is the problem in evaluating the infix expressions? What is the need for evaluating postfix/prefix expressions?
14) How to evaluate the postfix expression?
15) Given the following expressions give their postfix and prefix forms
    i. (A+B)*(D-C)
    ii. X$Y * Z – M + N + P | Q | (R+S) **(VTU-July/Aug 2004)**
16) Obtain the prefix expressions and postfix expressions for the following.

a) A+B-C*D
b) (A+B)*(C+D)$(A+B)

17) Convert the following prefix expressions to corresponding infix expressions.
a) –A+*$CDB/-FE*GHI
b) ^-*+ABC-DE+FG

18) Convert the following postfix expressions to corresponding infix expressions.
a) AB^C*D-EF/GH+/+
b) AB+C*DE—FG+^

19) Write an algorithm for evaluating a valid postfix expression. Trace the same on
i. A B + C – B A + C $ -
ii. A B C + * C B A - + *
for given value A=1, B=2, C=3

20) How to implement stacks using dynamic arrays?
21) How to implement multiple stacks using a single dimensional array?
22) What is recursion? What are the various types of recursion?
23) How to design recursive functions? What is base case? What is a general case?
24) Write a recursive C function to find factorial of a number
25) Write a recursive C function to find the sum of series
26) Write a recursive function to find sum of array elements from a[n-1] to a[0]
27) Write a recursive function to print array elements from a[0] to a[n-1]
28) What is GCD of two numbers? What is the recursive definition to compute GCD of two numbers?
29) What are Fibonacci numbers? What is the recursive definition to compute a Fibonacci number?
30) What is tower of Hanoi problem? Write a recursive function for the same
31) Write a C function to compute value of Ackermann's function
32) What is system stack? How the system stack is used to process a function call?
33) What is an activation record? How the control is transferred to or from the function with the help of activation record?
34) What is a maze problem? How maze is represented?
35) What is the disadvantage of representing a maze of size is m x n
36) Write a C program for maze problem

Chapter 7: QUEUES

What are we studying in this chapter?

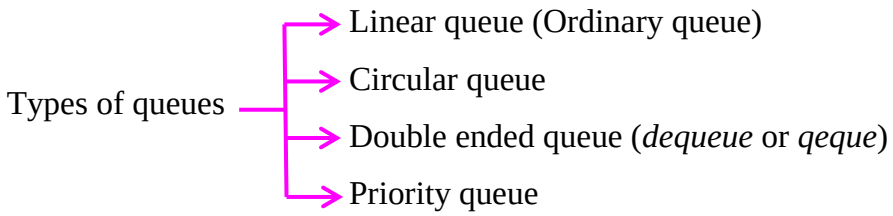
- Definition, array representation
- Queue operations
- Queue variants: Circular queue, priority queue, double ended queue
- Circular queues using dynamic arrays, multiple queues
- Programming examples **- 7 hours**

7.1 Definition and representation

This chapter deals with another important data structure namely queue. The term queue is very familiar to us in day to day life, as we see people standing in a queue to board the bus, or we see people standing in a queue near cinema hall to purchase the tickets etc., In any situation a person who just arrives will stand at the end of the queue and the person who is at the front of the queue is the first person to board the bus or to get the ticket etc., The same concept of queue is used in the field of computer science also. Now, let us see "What is a queue?"

Definition: A queue is a special type of data structure (an ordered collection of items) where elements are inserted from one end and elements are deleted from the other end. The end at which new elements are added is called the *rear* and the end from which elements are deleted is called the *front.* Using this approach, the **F**irst element **I**nserted is the **F**irst element to be deleted **O**ut, and hence, queue is also called **F**irst **I**n **F**irst **O**ut **(FIFO)** data structure.

Now, let us see "What are the different types of queues?" Since a queue elements are stored in linear order, it can be implemented using arrays and linked lists. Based on the method of insertion and deletion the queues are classified as shown below:



7.2 💻 Queues

Once we know what is a queue and what are the different types of queues, the next question is "How queues can be represented?" The queues can be represented using following two data structures:

- Arrays
- Linked lists

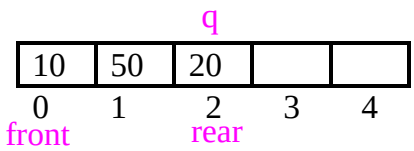
In this chapter we concentrate on how each type of queue can be represented using **arrays**. In the next chapter we see how a queue can be represented using **linked lists.**

7.2 Linear queue (Ordinary queue)

The linear queue or ordinary queue or simply a queue are one and the same. Now, let us see "What is linear queue or queue?"

Definition: A queue (linear queue or ordinary queue) is a special type of data structure (an ordered collection of items) where elements are inserted from one end and elements are deleted from the other end. The end at which new elements are added is called the *rear* and the end from which elements are deleted is called the *front.* Using this approach, the **F**irst element **I**nserted is the **F**irst element to be deleted **O**ut, and hence, queue is also called **F**irst **I**n **F**irst **O**ut **(FIFO)** data structure.

For example, consider the queue shown below having the elements 10, 50 and 20:



- The items are inserted into queue in the order 10, 50 and 20. The variable *q* is used as an array to hold these elements
- Item 10 is the first element inserted. So, the variable *first* is used as index to the first element
- Item 20 is the last element inserted. So, the variable *rear* is used as index to the last element
- Two more items can be inserted into above queue.

Now, let us see "what are the various operations that can be performed on a queue?" The various operations that can be performed on stacks are shown below:

Queue operations →
- **Insert** (An element is inserted from the rear end)
- **Delete** (An element is deleted from the front end)
- **Display** (Display the status of queue and queue contents)

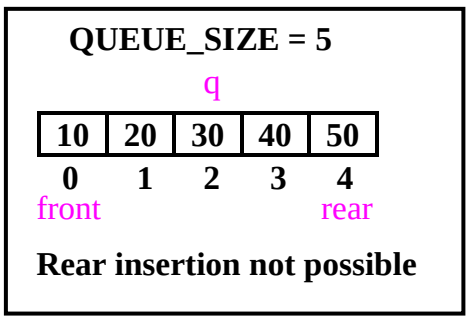
7.2.1 Insert into queue

In a queue, an item is always inserted at the rear end. Now, let us see "How to insert an item into the queue?"

Design: The various steps to be followed while inserting the elements into queue are shown below:

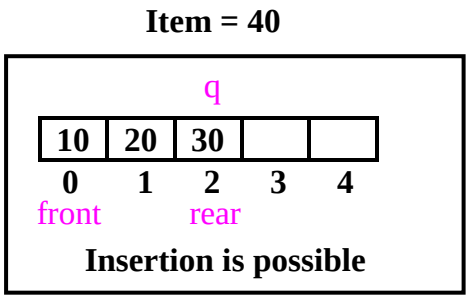
Step 1: Consider the queue shown below. Can we insert any element into the queue? No, it is not possible because queue is full.
Observe that whenever *rear* value is equal to "QUEUE_SIZE – 1" insertion is not possible. The code for this can be written as shown below:

```
if (rear == QUEUE_SIZE - 1)
{
        printf("Queue is full\n");
        return;
}
```



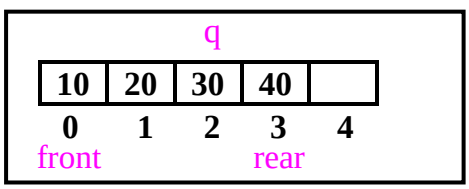
Step 2: If the above condition is not satisfied, it means the queue is not full and an element can be inserted at the *rear* end. Observe that *item* has to be inserted after 30 at position 3. That is, before inserting an item, we have to increment *rear* by one. This can be achieved using the statement:

```
rear = rear + 1;
```



Step 3: Now the *item* can be inserted at *rear* position. This can be achieved by copying *item* into q[rear] as shown below:

```
q[rear] = item;
```



Now, the complete function to insert an item into que can be written as shown below:

Example 7.1: Function to insert an item at the rear end of queue

Using global variables

```
void Insert_Rear()
{
    /* Check for overflow of queue */
    if (rear == QUE_SIZE – 1)
    {
        printf("Queue overflow\n");
        return;
    }

    /* Insert the item */
    rear = rear + 1;
    q[rear] = item;          } q[++rear] = item;
}
```

By passing parameters or int *q

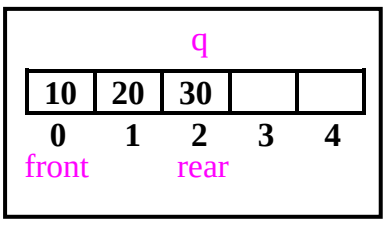
```
void Insert_Rear(int item, int *rear, int q[])
{
    /* Check for overflow of stack */
    if (*rear == QUE_SIZE – 1)
    {
        printf("queue overflow\n");
        return;
    }

    /* Insert the item */
    *rear = *rear + 1;
    q[*rear] = item;         } q[++(*rear)] = item;
}
```

7.2.2 Delete from queue

In a queue, an item is always removed from the front end. Now, let us see "How to delete an item from the queue?"

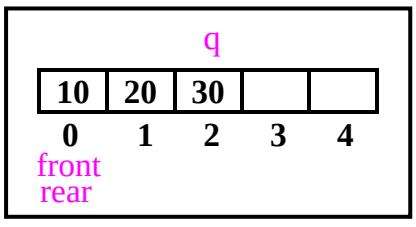
Design: The various steps to be followed while deleting an element from queue are shown below:

Step 1: Now, let us see "How to check whether queue is empty or not?"



3 items are present in queue

front < rear



only one item is present

front == rear

So, it is observed from above two figures that if *front* is less than or equal to *rear* some elements are present. Otherwise, that is, if *front* is greater than *rear* then queue is empty. We can check for empty queue using the following statement:

```
if (front > rear) return -1;      // Que is empty
```

Step 2: When above condition fails, we can delete an *item* from *front* end of queue. For this to happen, we have to access and return the first element and increment value of *front* by 1 as shown below:

```
return q[front++];
```

Now, the complete function to delete an element from the front end of the queue can be written as shown below:

Example 7.2: Function to delete an element from the front end of queue

```
int Delete_Front()
{
        if (front > rear) return -1;

        return q[front++];
}
```

Note: The variables *front*, *rear* and *q* are global. So, they can be accessed in all the functions.

or int *q

```
int Delete_Front(int *front,int *rear,int q[])
{
        if ( *front > *rear) return -1;

        return q[(*front)++];
}
```

Note: The variables *front*, *rear* and *q* have to be passed as parameters. Since the contents of *front*, *rear* and *q* are changed they should be treated as pointers.

7.2.3 Display queue items

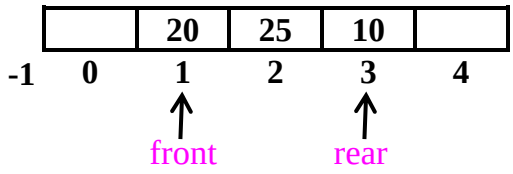
Now, let us see "How to display the elements present in queue?"

Design: The various steps to be followed while displaying the elements of from queue are shown below:

Step 1: Check for empty queue. This can be done using the following code:

```
if (front > rear)
{
        printf("Que is empty\n");
        return;
}
```

Step 2: If elements are present in queue control comes out of the above if statement. Assume that the queue contains three elements as shown in figure.



The contents of queue can be displayed as shown below:

```
printf("%d\n", s [0] );
printf("%d\n", s [1] );
printf("%d\n", s [2] );
```

Output

20
25
10

In general, we can use **printf("%d\n", s [i]);** **Note:** i = front to rear

Now, the code takes the following form:

```
for (i = front; i <= rear; i++)
{
        printf("%d\n", s[i]);
}
```

Now, the complete function can be written as shown below:

Example 7.3: Display queue elements

```
void Display()
{
    int i;

    /* If queue is empty */
    if (front > rear)
    {
        printf("Queue is empty\n");
        return;
    }
```

```
void Display(int front, int rear, int q[])
{
    int i;

    /* If queue is empty */
    if (front > rear)
    {
        printf("Queue is empty\n");
        return;
    }
```

```
        /* Display contents of queue */
        printf("Contents of the queue\n");
        for (i = front; i <= rear; i++)
        {
                printf("%d\n", q[i]);
        }
}
```

```
        /* Display contents of queue */
        printf("Contents of the queue\n");
        for (i = front; i <= rear; i++)
        {
                printf("%d\n", q[i]);
        }
}
```

7.2.4 Queue implementation using arrays (static implementation of queues)

The complete C program to implement different operations on a queue is shown below:

Example 7.4: C program to implement queue operations using global variables

```
#include <stdio.h>
#include <process.h>

#define QUE_SIZE 5

int choice, item, front, rear, q[10];          /* Global variables */

/* Include: Example 7.1: Function to insert an item (Using global variables) */

/* Include: Example 7.2: Function to delete an item (using global variables) */

/* Include: Example 7.3: To display contents of queue (global variables) */

void main()
{
        /* Initially queue is empty */
        front = 0;                                  /* Front end of queue */
        rear = -1;                                  /* Rear end of queue*/

        for (;;)
        {
                printf("1:Insert   2:Delete\n");
                printf("3:Display  4:Exit\n");
                printf("Enter the choice\n");
                scanf("%d", &choice);
```

```
            switch ( choice )
            {
                  case 1:
                        printf("Enter the item to be inserted\n");
                        scanf("%d", &item);

                        Insert_Rear();

                        break;

                  case 2:
                        item = Delete_Front();

                        if (item == -1)
                              printf("Queue is empty\n");
                        else
                              printf("Item deleted = %d\n", item);

                        break;

                  case 3:
                        Display();
                        break;

                  default:
                        exit(0);
            }
      }
}
```

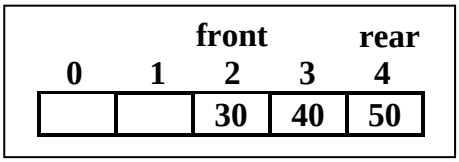
Example 7.5: C program to implement queue operations by passing parameters

```
#include <stdio.h>
#include <process.h>

#define QUE_SIZE 5

/* Include: Example 7.1: Function to insert an integer (passing parameters) */

/* Include: Example 7.2: Function to delete an item (by passing parameters) */

/* Include: Example 7.3: Function to display the contents of queue */
```

```
void main()
{
        int choice, item, front, rear, q[10];

        /* Initially queue is empty */
        front = 0;                                  /* Front end of queue */
        rear = -1;                                  /* Rear end of queue*/

        for (;;)
        {
                printf("1:Insert   2:Delete\n");
                printf("3:Display  4:Exit\n");
                printf("Enter the choice\n");
                scanf("%d", &choice);
                switch ( choice )
                {
                        case 1:
                                printf("Enter the item to be inserted\n");
                                scanf("%d", &item);
                                Insert_Rear(item, &rear, q);
                                break;

                        case 2:
                                item = Delete_Front(&front, &rear, q);
                                if (item == -1)
                                        printf("Queue is empty\n");
                                else
                                        printf("Item deleted = %d\n", item);

                                break;

                        case 3:
                                Display(front, rear, q);
                                break;

                        default:
                                exit(0);
                }
        }
}
```

7.3 Disadvantage of queue

Now, let us see "What is the disadvantage of queue which is represented using an array?" Consider the queue shown below:

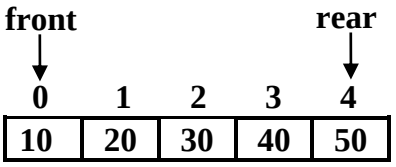


The above situation arises when 5 elements say 10, 20, 30, 40 and 50 are inserted and then deleting first two items 10 and 20. Now, if we try to insert an item we get the message

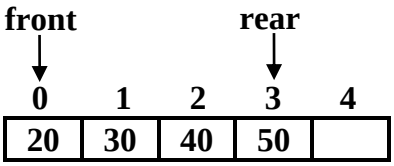
"Queue overflow"

Note: In the above situation, rear insertion is denied even if space is available at the front end. This is because in our function InsertQ() (see example **7.1)** before inserting an element, we test whether *rear* is equal to **QUEUE_SIZE – 1**. If so, we say "Queue is full and cannot insert". This is a disadvantage. This disadvantage can be overcome using two methods:

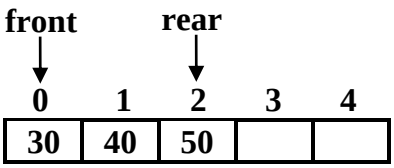
Method 1: Shift left: After deleting the element from the front, shift all remaining elements to the left. This can be pictorially represented as shown below:



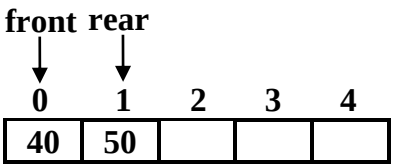
After inserting 10, 20, 30, 40, 50



After deleting 10, shift remaining elements to the left



After deleting 20 shift remaining items towards left



After deleting 30 shift remaining items towards left

When an item is deleted, all the items towards right are moved to left by one position and *rear* is decremented by 1. It is costly method and hence not recommended.

Method 2: Using circular representation: In a queue, we increment *rear* by 1 and then insert the item as shown below:

```
rear = rear + 1;
q[rear] = item;
```

In circular representation, we increment *rear* by 1 as usual and then perform the modulus operation using operator '%' as shown below:

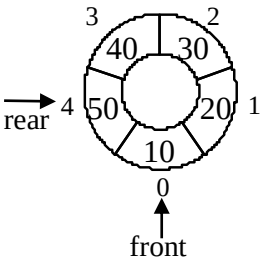
```
rear = (rear + 1) % QUE_SIZE;
```

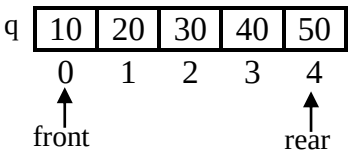
where QUE_SIZE is symbolic constant which represent maximum number of elements in the queue and it can be defined as shown below:

```
#define QUE_SIZE  5
```

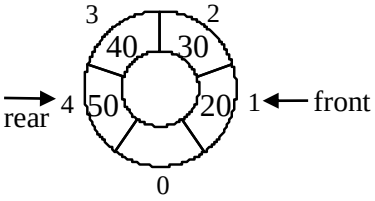
The significance of % operator is clear from the following example.

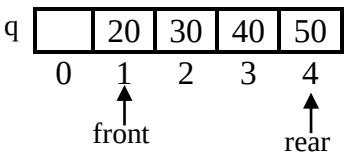
Insert: Assume QUE_SIZE is 5 and 5 elements are inserted into queue. The circular representation along with linear array are shown below:





Delete: In the above queue, item 10 is the first element. So, during deletion, the item 10 has to be deleted. This is achieved by incrementing *front* by 1 so that *front* contains the index of the second element. This is pictorially represented as shown below:

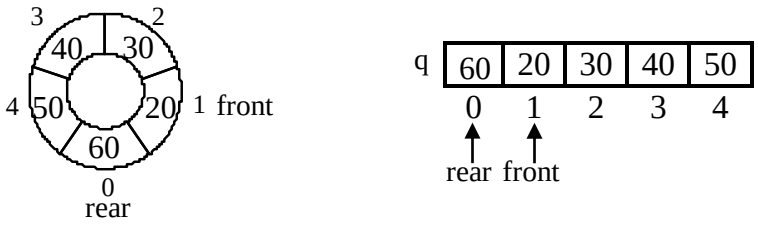




Insert: Now if we want to insert an item 60, we have to increment *rear* by 1. In the above figure, the value of *rear* is 4. If we increment *rear* by 1, its value will be 5. But, we have assumed that queue is circular. So, after incrementing by 1, it should be 0 instead of 5. This is achieved by taking modulus as shown below:

```
rear = (rear + 1) % QUE_SIZE
```

After executing the above statement, the value of *rear* will be 0 so that item 60 can be inserted at 0^{th} position as shown below:



Now, if we display the contents of queue, the output will be 20, 30, 40, 50, 60. This is because 20 is the first element in the queue and 60 is the last element in the queue.

Note: As we increment *rear* by 1 using the statement:

rear = (rear + 1) % QUE_SIZE

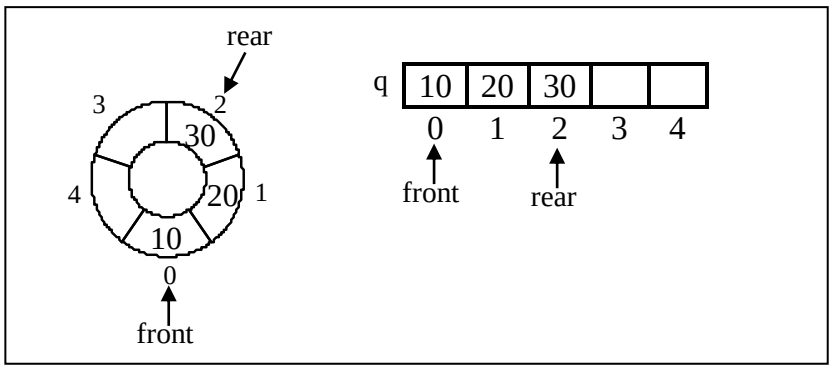
we increment *front* by 1 each time an item is deleted using the statement:

front = (front + 1) % QUE_SIZE

7.4 Circular queue

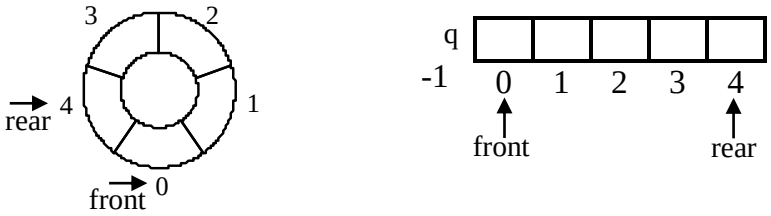
Now, we shall see "What is a circular queue?"

Definition: In circular queue, the elements of a given queue can be stored efficiently in an array so as to "wrap around" so that rear end of the queue is followed by the front of queue. The pictorial representation of a circular queue and its equivalent representation using an array are given side by side in figure below:



This circular representation allows the entire array to store the elements without shifting any data within the queue. This is an efficient way of implementing queues.

Empty queue: The circular representation and its equivalent array representation when queue is empty is shown below:



When queue is empty, *front* = 0 and *rear* = -1. But, in circular queue, just before index 0 we have index 4. So, instead of *rear* = -1, we can write *rear* = 4 also. Thus, empty queue is represented by following initialization statements:

```
front = 0
rear  = -1;
```

The above statements indicating empty queue can also be represented as shown below:

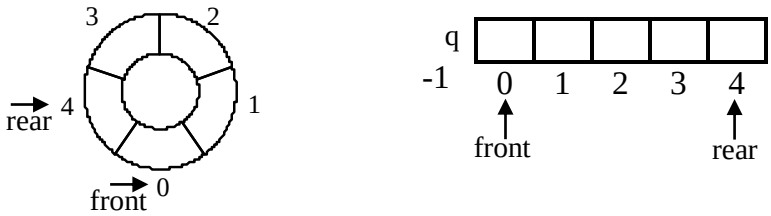
```
front = 0
rear  = 4;          // rear = 4. In general, rear = QUE_SIZE – 1
```

Example 7.6: Show the contents of circular queue after performing each of the following operations"

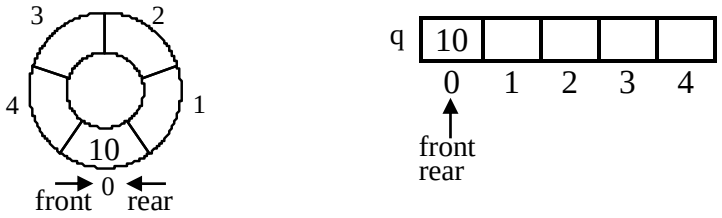
a) Empty queue
b) Insert 10
c) Insert 20 and 30
d) Insert 40 and 50
e) Insert 60
f) Delete two items
g) Insert 60 and 70
h) Insert 80

Solution: The queue and its representation after performing each of the above operations can be written as shown below:

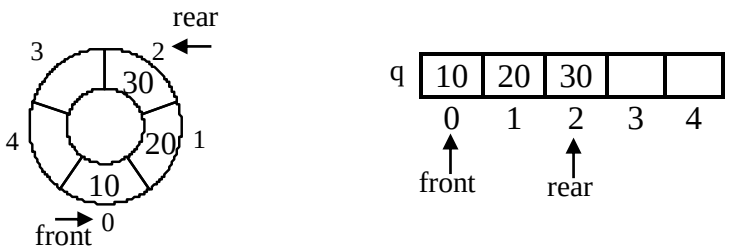
Step 1: Empty queue: Whenever *front* is 0 and *rear* is either -1 or QUE_SIZE – 1, queue is empty. An empty queue can be represented as shown below:



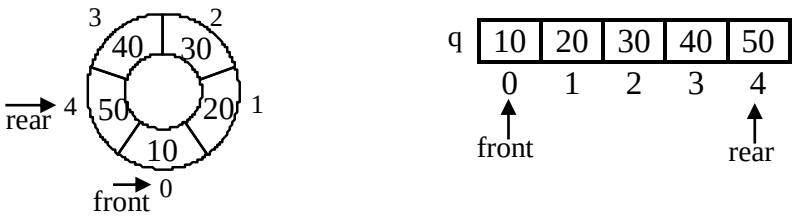
Step2: Inserting 10: After incrementing *rear* by 1, 10 is inserted as shown below:



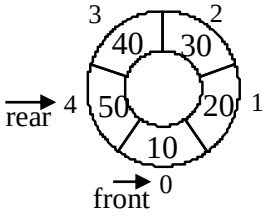
Step 3: Inserting 20 and 30: Increment *rear* by 1 and insert 20. Again increment *rear* by 1 and insert 30 as shown below:

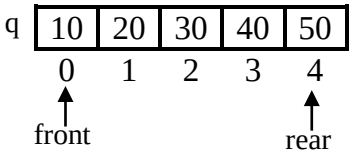


Step 4: Inserting 40 and 50: Increment *rear* by 1 and insert 40. Again increment *rear* by 1 and insert 50 as shown below:



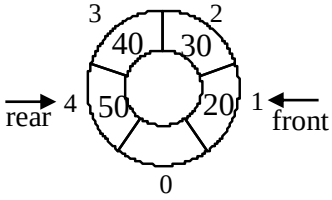
Step 5: Inserting 60: Queue is full. It is not possible to insert any element into queue. So, contents of queue have not been changed.

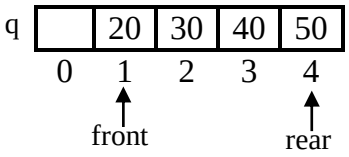




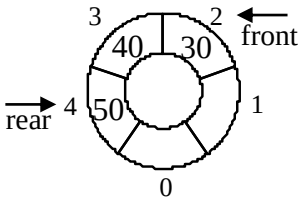
Queue is full

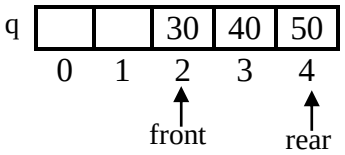
Step 6: Delete: An item has to be deleted always from the front end. So, 10 is deleted and contents of queue after deleting 10 is shown below:



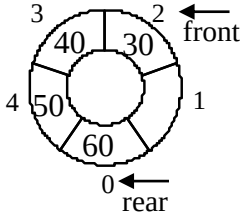


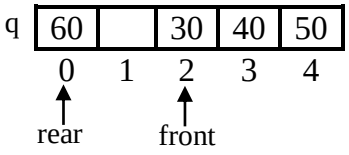
Step 7: Delete: An item has to be deleted always from the front end. So, 20 is deleted and contents of queue after deleting 20 is shown below:



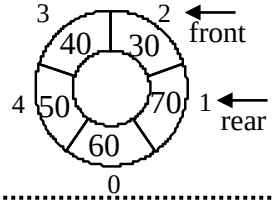


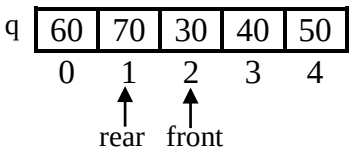
Step 8: Inserting 60: Incrementing *rear* by 1, its value is 0 (because of its circular representation) and insert 60 at 0^{th} location as shown below:



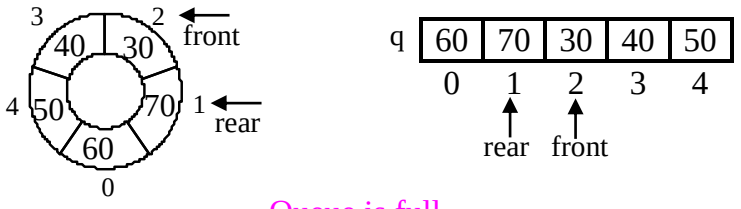


Step 9: Inserting 70: Increment *rear* by 1 and insert 70 as shown below:





Step 10: Inserting 80: Queue is full. It is not possible to insert any element into queue. So, contents of queue has not been changed.



Queue is full

7.4.1 InsertQ()

Now, let us see "How to implement insert function using arrays (static allocation technique)?"

Design: The various steps to be followed while inserting the elements into queue are shown below:

Step 1: Check for overflow: Before inserting, we check whether sufficient space is available in the queue. This can be achieved using the following code:

```
if (count == QUEUE_SIZE)
{
        printf("Queue is full\n");
        return;
}
```

Step 2: Insert item: Increment *rear* by 1 and then take the mod operation and then insert the item as shown below:

```
rear = (rear + 1) % QUEUE_SIZE;
q[rear] = item;
```

Step 3: Update count: As we insert an element into queue, we update **count** by 1. This is achieved using the following statement:

```
count++;
```

Note: Observe that as we insert an item, the *count* is incremented by 1. This indicates at any point of time, the variable *count* contains the total number of items present in the queue.

Now, the complete function using global variables and by passing as parameters can be written as shown below:

Example 7.7: InsertQ() using global variables and by passing parameters

```
void InsertQ()
{
    /* Check for overflow of queue */
    if (count == QUE_SIZE)
    {
        printf("Queue overflow\n");
        return;
    }

    rear = (rear + 1) % QUE_SIZE;
    q[rear] = item;
    count++;
}
```

```
void InsertQ(int item, int *rear, int *q,
                                  int *count)
{
    /* Check for overflow of stack */
    if (*count == QUE_SIZE)
    {
        printf("queue overflow\n");
        return;
    }

    *rear = (*rear + 1) % QUE_SIZE;
    q[*rear] = item;
    (*count)++;
}
```

7.4.2 DeleteQ()

Now, let us see "What are the various steps to be followed while deleting an element?" The following steps are followed:

Step 1: Check for underflow: Before deleting an element from queue, we check whether sufficient queue is empty or not. This can be achieved using the statement:

```
if (count == 0)  return -1;
```

When above condition fails, it means queue is not empty and return the element present at the front end of queue as shown in step2.

Step 2: Access the first item: This is achieved by accessing the element using index *front* and then updating *front* by adding 1 to it and then take mod value. The equivalent statements can be written as shown below:

```
item = q[front];                    // access the item
front = (front + 1) % QUE_SIZE;     // update front so that it contains
                                    // index of next element.
```

Step 3: Update count: As we delete an element from queue, decrement **count** by 1. This is achieved using the following statement:

```
count--;
```

Step 4: Return the element which was at the front end using the statement:

```
return item;
```

The function to delete an element using global variables/parameters is shown below:

Example 7.8: Function to delete an item from the front end of circular queue

```
int DeleteQ()
{
    int item;

    if (count == 0 ) return -1;

    item = q[front];
    front = (front + 1) % QUE_SIZE;
    count -= 1;

    return item;
}
```

```
int DeleteQ(int *front, int q[], int *count)
{
    int item;

    if ( *count == 0 ) return -1;

    item = q[*front];
    *front = (*front + 1) % QUE_SIZE;
    *count -= 1;

    return item;
}
```

7.4.3 DisplayQ()

Now, let us see "What are the various steps to be followed while displaying the elements of circular queue?" The following steps are followed:

Step 1: Check for underflow: This is achieved using the following statement:

```
if (count == 0)
{
        printf("Queue is empty\n");
        return;
}
```

Step 2: Display: Display starts from the *front* index. After displaying q[front] we have to update *front* by 1 (That is by incrementing *front* by 1 and then taking the modulus). The procedure is repeated for *count* number of times. This is because, *count* contains the number of items in queue. The code for this can be written as:

```
for (i = 1, f = front; i < = count; i++)
{
        printf("%d\n", q[f]);
        f = (f + 1) % QUE_SIZE;
}
```

So, the complete function to display the contents of queue is shown below:

Example 7.9: Function to display the contents of circular queue

```
void display()
{
    int i, f;

    if ( count == 0 )
    {
        printf("Q is empty\n");
        return;
    }
    printf("Contents of queue is\n");
    for ( i = 1, f = front; i <= count; i++)
    {
        printf("%d\n",q[f]);
        f = (f + 1) % QUE_SIZE;
    }
}
```

```
void display(int front, int q[], int count)
{
    int i, f;

    if ( count == 0 )
    {
        printf("Q is empty\n");
        return;
    }
    printf("Contents of queue is\n");
    for ( i = 1, f = front; i <= count; i++)
    {
        printf("%d\n",q[f]);
        f = (f + 1) % QUE_SIZE;
    }
}
```

The complete C program to implement circular queue by passing parameters is shown below:

Example 7.10: C program to implement circular queue using global variables

```
#include <stdio.h>
#include <process.h>

#define QUE_SIZE 5

int item, front, rear, count, q[QUE_SIZE];

/* Include: Example 7.7: Function to insert an item at the rear end */
/* Include: Example 7.8: Function to delete an item from the front end */
/* Include: Example 7.9: Function to display the contents of circular queue */
```

```
void main()
{
        int choice;

        front = 0;
        rear = -1;
        count = 0;      /* queue is empty */

        for (;;)
        {
                printf("1:Insert   2:Delete\n");
                printf("3:Display  4:Exit\n");
                printf("Enter the choice\n");
                scanf("%d",&choice);

                switch ( choice )
                {
                        case 1:
                                printf("Enter the item to be inserted\n");
                                scanf("%d",&item);
                                InsertQ();
                                break;

                        case 2:
                                item = DeleteQ();
                                if (item == -1)
                                {
                                        printf("Queue is empty\n");
                                        break;
                                }
                                printf("Item deleted = %d\n", item);
                                break;

                        case 3:
                                display();
                                break;

                        default:
                                exit(0);
                }
        }
}
```

The C program to implement circular queue by passing parameters is shown below:

Example 7.11: C program to implement circular queue by passing parameters

```
#include <stdio.h>
#include <process.h>

#define QUE_SIZE 5

/* Include: Example 7.7: Function to  insert an item at the rear end */

/* Include: Example 7.8: Function to delete an item from the front end */

/* Include: Example 7.9: Function to  display the contents of circular queue */

void main()
{
        int choice, item, front, rear, count, q[QUE_SIZE];

        front = 0;
        rear = -1;
        count = 0;      /* queue is empty */

        for (;;)
        {
                printf("1:Insert   2:Delete\n");
                printf("3:Display  4:Exit\n");
                printf("Enter the choice\n");
                scanf("%d", &choice);

                switch ( choice )
                {
                        case 1:
                                printf("Enter the item to be inserted\n");
                                scanf("%d",&item);

                                InsertQ(item, &rear, q, &count);

                                break;

                        case 2:
                                item = DeleteQ(&front, q, &count);
```

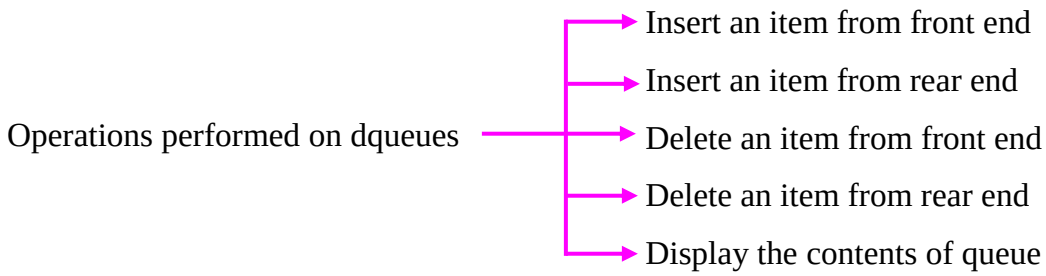
```
                        if (item == -1)
                        {
                                printf("Queue is empty\n");
                                break;
                        }

                        printf("Item deleted = %d\n", item);
                        break;
                case 3:
                        display(front, q, count);
                        break;
                default:
                        exit(0);
            }
        }
}
```

7.5 Double ended queue (Dequeue)

In this section, let us concentrate on another variant of queue called double ended queue. In short, it is also called dequeuer or qeque. The deque is pronounced as deck.

Now, let us see "What is a double ended queue (or dequeue)? and "What are the various operations that can be performed on dequeues?"

Definition: A Dequeue is a special type of data structure in which insertions are done from both ends and deletions are done at both ends. The operations that can be performed on deques are shown below:



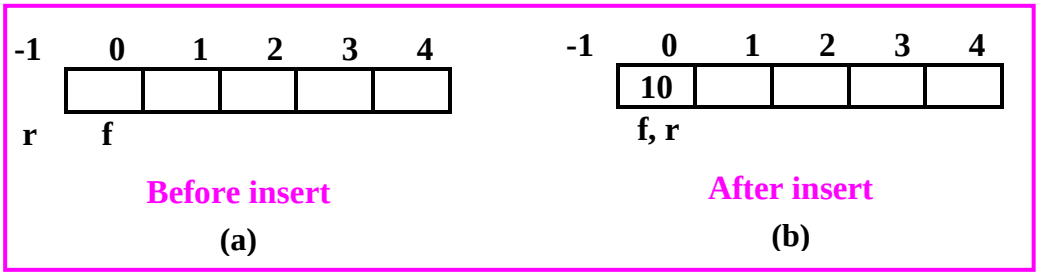
Note: The three operations Insert_Rear, Delete_Front and display operations have already been discussed in section 7.2. In this section, other two operations i.e., insert an item at the front end and delete an item from the rear end are discussed.

7.5.1 Insert at the front end

Now, let us see "How to implement insert front function using arrays (static allocation technique)?"

Design: Before inserting any element, we should ask the question "Where and how an item has to be inserted?" If we know the answer for this question we have the **insert front** function ready. So, let us consider various situations shown in figure below:

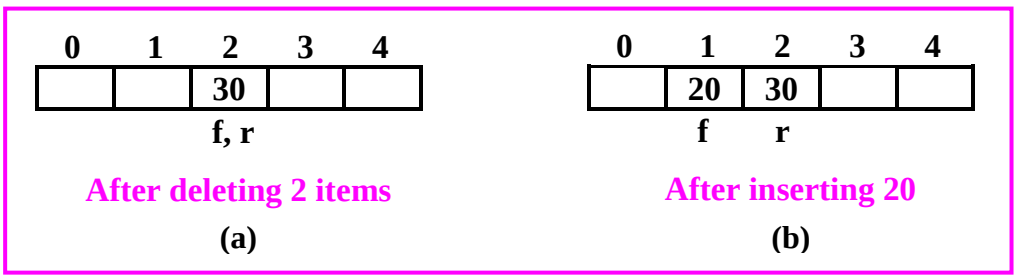
Case 1: Queue empty: When queue is empty, an item can be inserted at the front end first by incrementing **r** by 1 and then insert an item.



The equivalent code for this can be written as shown below:

```
/* Insert at front end if queue is empty: Case 1 (Fig a) */
if ( f == 0 && r == -1 )
{
        q[++r] = item;
         return;
}
```

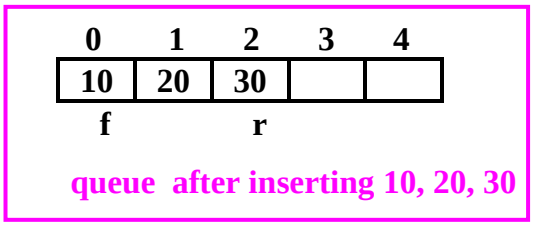
Case 2: Some items are deleted: Consider the following situation where 10, 20 and 30 were inserted earlier and 10 and 20 have been deleted from the front end. Now, there is only one item in queue. Here, an item can be inserted by decrementing the front index **f** by 1 and then inserting an item at that position as shown below:



The equivalent code for this can be written as shown below:

```
/* Insert at the front end if possible: Case 2 (Fig a)*/
if ( f != 0 )
{
        q[--f] = item;
        return;
}
```

Case 3: Some items are inserted (not deleted): Consider the following situation where 10, 20 and 30 were inserted into queue.



queue after inserting 10, 20, 30

Now, Observe that, in the above situation it is not possible to insert an any item at the front end and we should display the appropriate message. This is achieved using the following statement:

```
printf("Front insertion not possible\n");
```

The complete C function to insert an item at the front end is shown below:

Example 7.12: Function front insert(replacing f by front, r by rear (global variables)

```
void Insert_Front()
{
        if ( front == 0 && rear == -1 )        /* Case 1: Insert when Q empty */
        {
                q[++rear] = item;
                return;
        }

        if ( front != 0 )                      /* Case 2: Insert when items are present */
        {
                q[--front] = item;
                return;
        }

        printf("Front insertion not possible\n");
}                                       /* Case 3: Insertion not possible at front end */
```

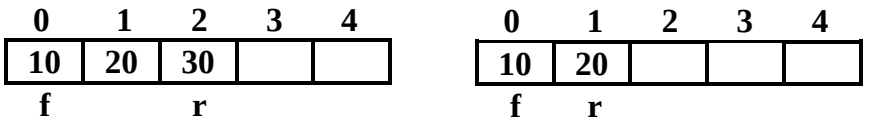
By passing parameters, the above function can be written as shown below:

Example 7.13: Function to insert an item at the front end (by passing parameters)

```
void Insert_Front(int item, int q[], int *front, int *rear)
{
        if ( *front == 0 && *rear == -1 )     /* Case 1: Insert when Q empty */
        {
                q[++(*rear)] = item;
                return;
        }

        if ( *front != 0 )                    /* Case 2: Insert when items are present */
        {
                q[--(*front)] = item;
                return;
        }

        printf("Front insertion not possible\n");
}                                             /* Case 3: Insertion not possible at front end */
```

7.5.2 Delete from the rear end

Now, let us see "How to implement delete rear operation using arrays (static allocation technique)?" Consider the following situation where 3 items are already inserted into queue and one item is deleted.



After inserting 3 items **(a)**

After deleting from rear **(b)**

Design: Item has to be deleted from rear end. This can be achieved by accessing the rear element q[r] as shown below:

```
printf("Item deleted = %d\n", q[r]);          /* Access and print rear item */
```

and then decrementing **r** by one as shown below:

```
r = r – 1;                                    /* Update position of rear item */
```

The above two statements can also be written using single statement as shown below:

```
printf("Item deleted = %d\n", q[r--]);        /* Access and update queue */
```

Observe that as each item is deleted, the index variable **r** is decremented so that it always contains the position of last item (see figure). Finally, when the queue is empty the value of **f** will be greater than **r.** Once **f** is greater than **r,** it is not possible to delete any item because queue is empty. This condition is called underflow of queue. Hence, the above statement has to be executed only if queue is not empty and the code to delete an item from rear end of queue can be written as shown below:

Example 7.14: Function to delete an item from the rear end (Using global variables)

```
int Delete_Rear()
{
        int item;
        if (front  > rear) return – 1;          /* Queue is empty */
        item = q[rear--];
        if (front > rear) front = 0, rear = -1; /* Reset to initial state of empty queue */
        return item;
}
```

The above function can be written by passing the parameters as shown below:

Example 7.15: Function delete rear (replace f by front, r by rear (passing parameters)

```
int Delete_Rear(int q[],int *front, int *rear)
{
        int item;
        if (*front  > *rear) return – 1;            /* Queue is empty */
        item = q[(*rear)--];                        /* Delete item from rear end */
        if (*front > *rear) *front = 0, *rear = -1; /* Reset to empty queue */
        return item;
}
```

The complete C code to implement double-ended queue using global variables is shown below

Example 7.16: C program to implement double-ended queue using global variables

```
#include <stdio.h>
#include <process.h>

#define QUE_SIZE 5

int choice, item, front, rear, q[10];          /* Global variables */

/* Include: Example 7.1: Function to insert an item at the rear end  */

/* Include: Example 7.2: Function to delete an item  at the front end */

/* Include: Example 7.12: Function to insert an item at the front end  */

/* Include: Example 7.14: Function to delete an item  at the rear end */

/* Include: Example 7.3: To display contents of queue   */

void main()
{
        /* Initially queue is empty */
        front = 0;                              /* Front end of queue */
        rear = -1;                              /* Rear end of queue*/

        for (;;)
        {
                printf("1:Insert_front  2:Insert_rear\n");
                printf("3:Delete_front  4:Delete_rear\n");
                printf("5:Display       6:Exit\n");
                printf("Enter the choice\n");
                scanf("%d", &choice);

                switch ( choice )
                {
                        case 1:
                                printf("Enter the item to be inserted\n");
                                scanf("%d",&item);
                                Insert_Front();
                                break;
```

```
                    case 2:
                        printf("Enter the item to be inserted\n");
                        scanf("%d",&item);
                        Insert_Rear();
                        break;
                    case 3:
                        item = Delete_Front();
                        if (item == -1)
                            printf("Queue is empty\n");
                        else
                            printf("Item deleted from front = %d\n", item);

                        break;
                    case 4:
                        item = Delete_Rear();
                        if (item == -1)
                            printf("Queue is empty\n");
                        else
                            printf("Item deleted from rear = %d\n", item);

                        break;

                    case 5:
                        Display();
                        break;

                    default:
                        exit(0);
                }
            }
}
```

Example 7.17: C program to implement dequeue by passing parameters.

```
#include <stdio.h>
#include <process.h>

#define QUE_SIZE 5

/* Include: Example 7.1: Function to insert an item at the rear end  */
```

```
/* Include: Example 7.2: Function to delete an item  at the front end */

/* Include: Example 7.13: Function to insert an item at the front end  */

/* Include: Example 7.15: Function to delete an item  at the rear end */

/* Include: Example 7.3: To display contents of queue   */

void main()
{
        int choice, item, front, rear, q[10];

        front = 0;
        rear = -1;

        for (;;)
        {
                printf("1:Insert_front  2:Insert_rear\n");
                printf("3:Delete_front  4:Delete_rear\n");
                printf("5:Display       6:Exit\n");

                printf("Enter the choice\n");
                scanf("%d",&choice);

                switch ( choice )
                {
                        case 1:
                                printf("Enter the item to be inserted\n");
                                scanf("%d",&item);

                                Insert_Front(item, q, &front, &rear);

                                break;

                        case 2:
                                printf("Enter the item to be inserted\n");
                                scanf("%d",&item);

                                Insert_Rear(item, &rear, q);

                                break;
                        case 3:
                                item = Delete_Front(&front, &rear, q);
```

```
                        if (item == -1)
                                printf("Queue is empty\n");
                        else
                                printf("Item deleted from front = %d\n", item);

                        break;

                case 4:
                        item = Delete_Rear(q, &front, &rear);

                        if (item == -1)
                                printf("Queue is empty\n");
                        else
                                printf("Item deleted from rear = %d\n", item);

                        break;

                case 5:
                        Display(front, rear, q);
                        break;

                default:
                        exit(0);
            }
        }
}
```

7.6 Priority queue

In this section, let us see another variation of queue called priority queue. Let us see "What is priority queue? What are the various types of priority queues?"

Definition: A queue in which we are able to insert items or remove items from any position based on some priority is often referred to as a priority queue. Always an element with highest priority is processed before processing any of the lower priority elements. If the elements in the queue are of same priority, then the element, which is inserted first into the queue, is processed. Priority queues are used in job scheduling algorithms in the design of operating system where the jobs with highest priorities have to be processed first.

The priority queues are classified into two groups:

- Ascending priority queue: In an ascending priority queue elements can be inserted in any order. But, while deleting an element from the queue, only the smallest element is removed first.

- Descending priority queue: In descending priority also elements can be inserted in any order. But, while deleting an element from the queue, only the largest element is deleted first.

Now, let us see "How to implement priority queues?" There are various methods of implementing priority queues using arrays.

Design 1: One of the methods is to implement an ascending priority queue where elements can be inserted in any fashion and only the smallest element is removed. Here, an element is inserted from the rear end of the queue but an element with least value should be deleted. After deleting the smallest number, store a very large value in that location, indicating the absence of an item. The variable count can be used to keep track of number of elements in the array. The three functions useful for this purpose are:

- insert_rear() - which inserts the item at the end of the queue.
- remove_small() – which returns the smallest item from the queue and at the same time store maximum number in that location indicating an item has been deleted.
- display() – which displays the contents of queue.

It is left as an exercise to the reader to implement this. Now, let us implement priority queues using another technique.

Design 2: The second technique is to insert the items based on the priority. In this technique, we assume the item to be inserted itself denotes the priority. So, the items with least value can be considered as the items with highest priority and items with highest value can be considered as the items with least priority. So, to implement priority queue, we insert the elements into queue in such a way that they are always ordered in increasing order. With this technique the highest priority elements are at the front end of the queue and lowest priority elements are at the rear end of queue. Hence, while deleting an item, always delete from the front end so that highest priority element is deleted first. The function to insert an item at the appropriate place is shown below:

Note: To insert an element into appropriate place so that elements are arranged in ascending order, we use the insertion sort technique.

Example 7.18: Function to insert an item at the correct place in priority queue

```
void Insert_Item(int item, int q[], int *r)
{
        int     j;

        if ( *r == QUE_SIZE-1 )      /* Check for overflow */
        {
                printf("Q is full\n");
                return ;
        }

        j = *r ;                     /*Compare from this initial rear point */

         /* Find the appropriate position to make room for inserting
         an item based on the priority */
        while ( j >= 0 && item < q[j])
        {
                q[j+1] = q[j];       /* Move the item at q[j] to its next position */
                j--;
        }

        q[j+1] = item;               /* Insert an item at the appropriate position */

        *r = *r + 1;                 /* Update the rear pointer */
}
```

Note: To implement priority queue, insert an item such that items are arranged in ascending order. But, always delete an item from the front end. The functions delete_front() and display() remains unaltered. The complete program to implement priority queue is shown below:

Example 7.19: C Program to implement priority queues

```
#include <stdio.h>
#include <process.h>

#define QUE_SIZE 5

/* Include: Example 7.2: C function to delete an integer item */

/* Include: Example 7.3: Function to display the contents of queue */
```

```
/* Include: Example 7.18: Function to insert item at the correct place in queue */

void main()
{
        int choice, item, front, rear, q[10];

        front = 0;
        rear = -1;

        for (;;)
        {
                printf("1:Insert  2:Delete\n");
                printf("3:Display 4:Exit\n");
                printf("Enter the choice\n");
                scanf("%d", &choice);

                switch (choice)
                {
                        case 1:
                                printf("Enter the item to be inserted\n");
                                scanf("%d",&item);
                                Insert_Item(item, q, &rear);
                                break;

                        case 2:
                                item = Delete_Front(&front, &rear, q);

                                if (item == -1)
                                        printf ("Queue is empty\n");
                                else
                                        printf("Item deleted = %d\n", item);

                                break;
                        case 3:
                                Display(, front, rear, q);
                                break;

                        default:
                                exit(0);
                }
        }
}
```

Note: We can input in any order. But, the items are inserted at the appropriate position and are arranged in ascending order. The item at 0^{th} position is having highest priority; the item at 1^{st} position is having next highest priority and so on. Finally, the item at the end of the queue is having the least priority. So, if we remove from front, an item with highest priority is removed.

Note: The priority queue need not have the numbers or characters. They can represent complex structures and elements can be ordered based on some common property. In this sense, a stack can be considered as a priority queue. If we take the time as an ordering on the elements, an element that is inserted first has to spend more time in the stack whereas the element which is inserted recently will spend less time in the stack. This is because stack is a last in first out data structure. So, while deleting, the element that has spent less time is removed first.

Even an ordinary queue can also be considered as a priority queue where the priority is based on the order in which they are inserted. Here, the first element inserted into queue is the first element to go out of the queue. So, an item that is inserted first can be considered as the item with highest priority and so it is removed first from the queue.

7.7 Applications of queue

We know that queues are special type of data structures where the items are deleted in the order in which they are inserted. So, queues are used for a situation where we have to maintain a FIFO order efficiently. These situations occur in every type of software development. The real applications of queues are shown below:

- In a multitasking operating system, the CPU cannot run all jobs at once. So, all the jobs are arranged in batches and are scheduled based on FIFO which is an application queue.
- Operating System maintains a queue of processes that are ready to execute
- Operating System maintains a queue of processes that are waiting for a particular event to occur
- When multiple users use a printer in a networked computer system, all the files to printed will be sent to a queue. All the files will be printed by the printer in the order in which they are queued in a print queue. This ensures that only one person has access to the printer at a given time.

7.8 Multiple queues

Multiple queues can be easily implemented using array of linked list (see section 8.9.4).

7.9 Circular Queue using Dynamic Arrays

Now, let us see "What is the disadvantage of using arrays to implement queue operations?" The disadvantage of using arrays to implement queue are shown below:

- If arrays are used to implement queues, then maximum size of the queue (QUE_SIZE) should be known during compilation.
- Once the size of the queue is fixed during compilation, it is not possible to alter the size of the queue during execution. This disadvantage we can overcome using dynamic arrays.

Now, the various operations that can be performed on circular queue using dynamic arrays can be implemented as shown below:

Now, let us see "How to insert an element into a circular queue using dynamic array?" Assume *q* is pointer to an integer, two index variables *front* and *rear* are initialized to 0 and -1 respectively. Another variable *count* is initialized to 0 which indicate queue is empty. Initial size of the queue denoted by SIZE is 1.

Design: Before inserting, we check whether sufficient space is available in the circular queue. We know that if *count* value is same as SIZE, then circular queue is full. The code for this condition can be written as shown below:

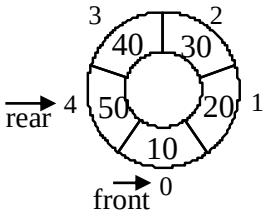
```
if (count == SIZE)
{
        printf("Q full:\n");
        return;
}
```

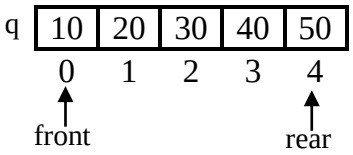
But, once the queue is full, instead of returning the control, we can increase the size of array using realloc() function and the above code can be modified as shown below:

```
if (count == SIZE )
{
        printf("Q Full: update size, insert item\n ");

        SIZE++;

        q = (int *) realloc(q, SIZE*sizeof(int) );
}
```

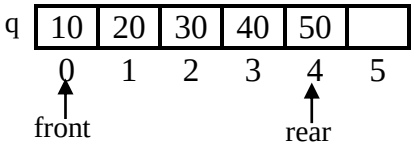
When above condition fails, it means we can insert an item. If the above condition is true, space is not sufficient. So, using realloc() we increase the size by 1. However, it is not sufficient to simply increase the size using realloc(). The queue contents have to be re-adjusted:

Case 1: The *front* index is less than the *rear* index: Consider the following circular queue with five elements whose capacity SIZE is 5:



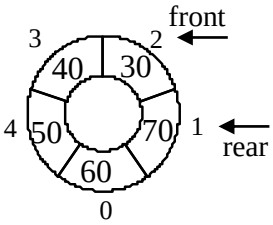


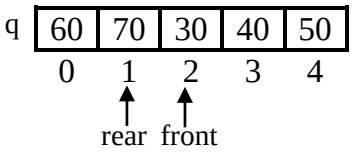
After increasing SIZE by 1 using realloc(), the contents of circular queue is shown below:



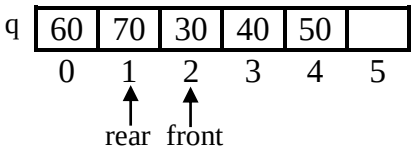
Now, an item can be inserted as usual into circular queue in usual manner by incrementing *rear* by 1, insert the item and then update *count* by 1.

Case 2: The *front* index is greater than the *rear* index: Consider the following circular queue with five elements whose capacity SIZE is 5:

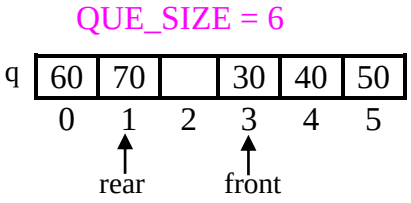




After increasing SIZE by 1 using realloc(), the contents of circular queue is shown below:



Now, to get the proper circular queue configuration, slide the elements 30, 40, 50 in above queue, to the right by one position using the following statements so that queue looks as shown below:




```
q[5] = q[4]
q[4] = q[3]
q[3] = q[2]
```

In general, q[i+1] = q[i] for i = 4 down to 2
= 6 - 2 down to 2
= SIZE – 2 down to *front*

Now, shifting towards right by 1 position can be coded as shown below:

```
for (i = SIZE – 2; i  >= front; i--)
{
        q[i+1] = q[i];
}
```

Observe that *front* should be incremented by 1. This can be done using the statement:

```
front++;
```

This re-adjusting is done only after re-allocating the memory. Now, the code can be written as shown below:

```
if (count == SIZE )
{
        printf("Q Full: update size, insert item\n ");
        SIZE++;
        q = (int *) realloc(q, SIZE*sizeof(int) );

        if (front > rear)
        {
                for (i = QUE_SIZE – 2; i >= front; i--)
                {
                        q[i+1] = q[i];
                }
                front++;
        }
}
```

All the above statements have been written by assuming *front*, *rear*, *q* and *count* as global variables. If they are passed as parameters and because their values are being changed and they should declared as pointers. Now, the complete function using global variables can be written as shown below:

Example 7.20: C function to InsertQ() using global variables

```
void InsertQ()
{
        int     i;

        if (count == QUE_SIZE)
        {
                printf("Q Full: update size, item insert\n");

                QUE_SIZE++;

                q = (int *) realloc(q, QUE_SIZE*sizeof(int) );

                if (front > rear)
                {
                        for (i = QUE_SIZE - 2; i >= front; i--)
                        {
                                q[i+1] = q[i];
                        }
                        front++;
                }
        }
        rear = (rear + 1) % QUE_SIZE;
        q[rear] = item;
        count++;
}
```

The complete C program to implement circular queue using dynamic arrays by using global variables is shown below:

Example 7.21: C program to implement circular queue using global variables

```
#include <stdio.h>
#include <stdlib.h>
```

```
int QUE_SIZE = 1;   // Global variable

int item, front, rear, count, *q;

/* Include: Example 7.20: Function to  insert an item at the rear end */

/* Include: Example 7.8: Function to delete an item from the front end */

/* Include: Example 7.9: Function to  display the contents of circular queue */

void main()
{
        int choice;
        front = 0, rear = -1, count = 0;        /* queue is empty */

        q = (int *) malloc (sizeof (int) );

        for (;;)
        {
                printf("1:Insert   2:Delete\n");
                printf("3:Display  4:Exit\n");

                printf("Enter the choice\n");
                scanf("%d",&choice);

                switch ( choice )
                {
                        case 1:
                                printf("Enter the item to be inserted\n");
                                scanf("%d",&item);
                                InsertQ();
                                break;

                        case 2:
                                item = DeleteQ();

                                if (item == -1)
                                {
                                        printf("Queue is empty\n");
                                        break;
                                }
                                printf("Item deleted = %d\n", item);
                                break;
```

```
                    case 3:
                        display();
                        break;

                    default:
                        exit(0);
                }
        }
}
```

The function written in example 7.20 ccan be written by passing parameters as shown below:

Example 7.22: Function to delete an item from the front end of circular queue

```
void InsertQ(int item, int *front, int *rear, int *q,  int *count)
{
        int i;

        if (*count == QUE_SIZE)
        {
                printf("q Full:update size,insert item\n");

                QUE_SIZE ++;

                q = (int *) realloc (q, QUE_SIZE * sizeof(int) );

                if (*front > *rear)
                {
                        for (i = QUE_SIZE – 2;  i  >=  *front;  i--)
                        {
                              q[i+1] = q[i];
                        }
                        (*front)++;
                }
        }

        *rear = (*rear + 1) % QUE_SIZE;

        q[*rear] = item;

        (*count)++;

}
```

The C program to implement circular queue by passing parameters is shown below:

Example 7.23: C program to implement circular queue by passing parameters

```
#include <stdio.h>
#include <stdlib.h>

int QUE_SIZE = 1;   // Global variable

/* Include: Example 7.22: Function to  insert an item at the rear end */

/* Include: Example 7.8: Function to delete an item from the front end */

/* Include: Example 7.9: Function to  display the contents of circular queue */

void main()
{
        int choice, item, front, rear, count, *q;

        front = 0;
        rear = -1;
        count = 0;      /* queue is empty */

        q = (int *) malloc (sizeof (int) );

        for (;;)
        {
                printf("1:Insert   2:Delete\n");
                printf("3:Display  4:Exit\n");
                printf("Enter the choice\n");
                scanf("%d", &choice);

                switch ( choice )
                {
                        case 1:
                                printf("Enter the item to be inserted\n");
                                scanf("%d",&item);

                                InsertQ(item, &front, &rear, q, &count);

                                break;
```

```
                    case 2:
                        item = DeleteQ(&front, q, &count);
                        if (item == -1)
                        {
                            printf("Queue is empty\n");
                            break;
                        }

                        printf("Item deleted = %d\n", item);
                        break;
                    case 3:
                        display(front, q, count);
                        break;
                    default:
                        exit(0);
                }
            }
}
```

EXERCISES

1. What is a queue? What are the various operations that can be performed on queues?

2. What are the different types of queues?

3. Explain an ordinary queue and write a C program to implement the same

4. What is the disadvantage of an ordinary queue? How you can overcome this disadvantage?

5. What is a circular queue? How it is different from an ordinary queue?

6. Write a C program to implement circular queues using arrays

7. What is a double ended queue or deque? What are the operations that can be performed on double-ended queues?

8. Write a C program to implement deques using arrays

9. What is a priority queue? How, it is different from an ordinary queue?

10. Explain how a priority queue can be implemented?

11. Write a C program to implement priority queues using arrays

12. A Circular queue the size of which is 5 has 3 elements 10,40,25, where F = 2 and R = 4. After inserting 50,60, what is the value of F and R? Trying to insert an element 30 at this stage what will happen? Delete 2 elements from the queue and insert 100. Show the sequence of steps with necessary diagrams with the value of F and R.

13. What is an ascending priority queue and descending priority queue?

14. Implement an ascending priority queue using arrays

15. Explain how elements can be arranged in ascending order in priority queue while inserting? After arranging them in ascending order, what other functions are necessary to implement priority queues?

16. Show the contents of circular queue after performing each of the following operations
    a) Empty queue
    b) Insert 10
    c) Insert 20 and 30
    d) Insert 40 and 50
    e) Insert 60
    f) Delete two items
    g) Insert 60 and 70
    h) Insert 80

Chapter 8: Linked Lists

What are we studying in this chapter?

- Definition, Representation of linked lists in Memory
- Linked list operations: Traversing, Searching, Insertion, Deletion.
- Linked Stacks and Queues
- Doubly Linked lists
- Circular linked lists
- header linked lists.
- Applications of Linked lists, Polynomials, Sparse matrix representation.
- Memory allocation; Garbage Collection.
- Programming Examples

8.1 Introduction

In the previous chapters, we have seen various types of linear data structures such as stacks, queues and their representations and implementations using sequential allocation technique i.e., using arrays. Let us see, "What are various advantages of using arrays?" The advantages of using arrays in implementing various data structures are shown below:

- Data accessing is faster: Using arrays, the data can be accessed very efficiently just by specifying the array name and the corresponding index. For example, we can access $\mathbf{i}^{th}$ item in the array A by specifying A[i]. The time taken to access a[0] is same as the time taken to access a[10000].
- Simple: Arrays are simple to understand and use.

Now, let us see "What are the disadvantages of arrays?" Even though arrays are very useful in implementing various data structures, there are some disadvantages:

- The size of the array is fixed: The size of the array is fixed during compilation time only.
- Insertion and deletion operations involving arrays is tedious job: Consider an array consisting of 200 elements. If item has to be inserted at 1^{st} position, all 200 elements must be moved to their next immediate positions, making room for the new item. Then item has to be inserted at the 1^{st} position. Similarly, deleting an item at a given position consumes time.

The above disadvantages can be overcome using linked lists.

8.2 💻 Linked Lists

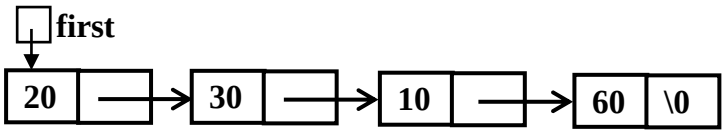
8.1.1 Linked list

Now, let us see "What is a linked list?"

Definition: A linked list is a data structure which is collection of zero or more nodes where each node is connected to one or more nodes. If each node in the list has only one link, it is called singly linked list. If it has two links one containing the address of the next node and other link containing the address of the previous node it is called doubly linked list. Each node in the singly list has two fields namely:

- **info –** This field is used to store the data or information to be manipulated
- **link –** This field contains address of the next node.

The pictorial representation of a singly linked list where each node is connected to the next node is shown below:



The above list consists of four nodes with info fields containing the data items 20, 30, 10 and 60.

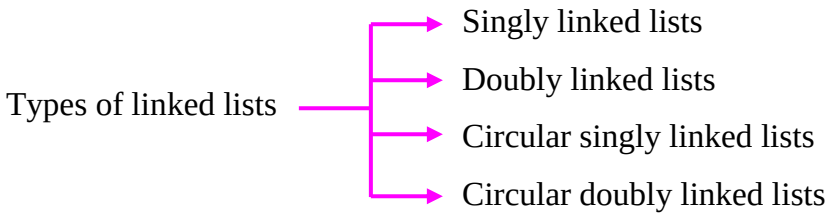
Note: Given the address of a node, we can easily obtain addresses of subsequent nodes using *link* fields. Thus, the adjacency between the nodes is maintained by means of links.

Analogy: A linked list can be compared to train having zero or more coaches.

- The people sitting inside each coach represent the data part of the linked list
- The connection between the coaches using iron links represent the address field of linked list. The link field contains address of the next node.
- As we can move from one coach to other coach, we can go from one node to other node.
- As coaches can be attached or detached easily, in linked list also a node can be inserted or a node can be deleted.

8.1.2 Types of Linked list

Now, let us see "What are the different types of linked list?" The linked lists are classified as shown below:

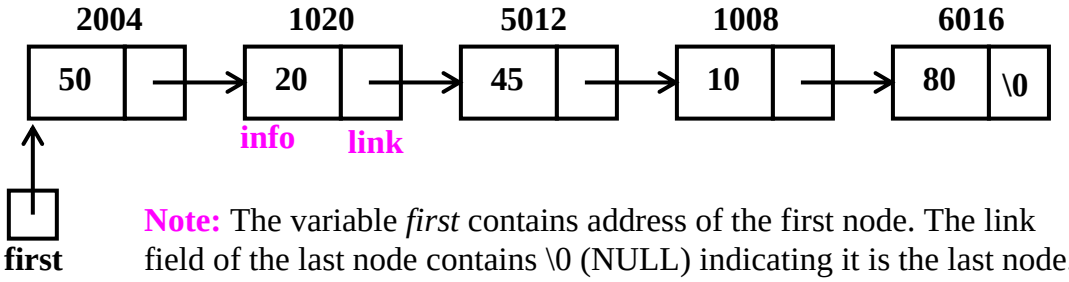


8.1.3 Singly linked lists

Now, let us see "What is a singly linked list?"

Definition: A singly linked list is a collection of zero or more nodes where each node has two or more fields but only one link field which contains address of the next node. Each node in the list can be accessed using the link field which contains address of the next node.

For example, a singly linked list consisting of the items 50, 20, 45, 10 and 80 is shown below:



Note: The variable *first* contains address of the first node. The link field of the last node contains \0 (NULL) indicating it is the last node.

Observe from above figure that:

- This list contains 5 nodes and each node consists of two fields, *info* and *link*.
- The *info* field of each node contains the data. In this list, the items **50**, **20**, **45**, **10** and **80** represent the data.
- The numbers on top of each node i.e., **2004**, **1020**, **5012**, **1008** and **6016** represent the address of each node in the memory.

Note: Observing the addresses of nodes i.e., **2004**, **1020**, **5012**, **1008** and **6016**, we know that they are physically far apart (they are not adjacent. One node starts at 2004, other starts at 1020 and so on). But, using *link* field we can obtain each node in the list in the order specified and hence we say they are logically adjacent.

- The *link* field of each node contains address of the next node. For example, consider the second node. The link field of second node contains 5012 which is address of the next node. It is denoted by an arrow originating at the link field and ending at next node.
- The variable *first* contains the address of the first node of the list. So, the entire list can be identified by the name *first*.
- Using the variable *first*, we can access any node in the list.
- The *link* field of a last node contains \0 (null).

Now, let us "Define an empty linked list"

Definition: An empty linked list is a pointer variable which contains NULL (\0). This indicates that linked list does not exist. For example, an empty list can be written as shown below:

8.1.4 Representing a node of linked list

The nodes in a linked list are self-referential structures. So, let us see "What are self-referential structures? How to define a node in C language?"

Definition: A structure is a collection of one or more fields which are of same or different types. A self-referential structure is a structure which has at least one field which is a pointer to itself. For example, consider the following structure definition:

```
struct node
{
    int          info;
    struct node  *link;
};
```

Observe from the above structure definition that a *node* has two fields:

- *info* – It is an integer field which contains the information.
- *link* – It is a pointer field and hence it should contain the address. The link field contains the address of the next node in the list whose type is same as the link field.

Now, let us see "How to define a self-referential structure?" The self-referential structure can be defined as shown below:

Example 8.1: Structure definition of a node along with declaration

```
/* Structure definition for a node */
struct node
{
        int             info;
        struct node     *link
} ;

typedef struct node *NODE;
```

Observe that the above structure is a tagged structure. Using the keyword typedef the type "struct node *" is given the alternate name as NODE. So, wherever we use "struct node *" we can replace with NODE.

A pointer variable *first* can be declared as shown below:

Method 1: **NODE** first;
or
Method 2: **struct node *** first;

Note: Both the declarations are same, since **NODE** and **struct node *** can be interchangeably used because of above **typedef**.

8.1.5 Create an empty list

Now, let us see "How to create an empty list?" An empty list is can be created by assigning NULL to a self-referential structure variable.

For example, consider the following code:

```
struct node
{
        int             info;
        struct node     *link
} ;

typedef struct node *NODE;

NODE first;                 // first is self-referential structure variable.

first = NULL;               // empty list by name first is created
```

An empty list identified by the variable *first* is pictorially represented as shown below:

8.1.6 Create a node

Now, let us see "How to create a node of the linked list?"

In C language, we can use malloc() function to allocate memory explicitly as and when required and exact amount of memory space needed during execution. This can be done using the statement:

x = (data_type *) malloc(size);

- On successful allocation, the function returns the address of first byte of allocated memory. Since address is returned, the return type is a **void** pointer. By ***type casting*** appropriately we can use it to point to any desired data type.

- If specified size of memory is not available, the condition is called **"overflow of memory".** In such case, the function returns NULL. It is the responsibility of the programmer to check whether the sufficient memory is allocated or not as shown below:

```
if (x == NULL)
{
        printf("Insufficient memory\n");
        exit(0);
}
```

It *x* is not NULL it means, a node is successfully created and we can return the node using the statement:

```
return x;
```

Thus, the C function to get a node can be written as shown below:

Example 8.2: C Function to get a new node from the availability list

```
NODE getnode()
{
        NODE x;
```

```
        x = ( NODE ) malloc(sizeof(struct node));  /* allocate the memory space */

        if ( x == NULL )                           /* Free nodes don't exist  */
        {
                printf("Out of memory\n");         /* Allocation failed */
                exit(0);                           /* Terminate the program */
        }

        return x;                                  /* allocation successful */
}
```

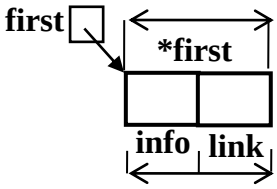
Note: The return type of getnode() is **NODE** and it is defined in previous section.

8.1.7 Create a node with specified item

Now, using the function getnode(), let us see "How to create a node with specified item?" This can be done using the following steps.

Step 1: get a node: A node which is identified by variable *first* with two fields: *info* and *link* fields can be created using getnode() function (described in previous section) as shown below:

```
first = getnode();
```

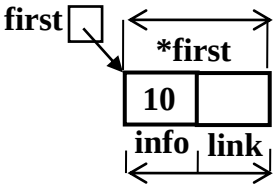


Note: Observe from the above figure that:

- Using the variable *first* we can access the address of the node
- Using **first* we can access entire contents of node
- Using (**first*).*info* or *first->info*, we can access the data stored in *info* field
- Using (**first*).*link* or *first->link*, we can access the *link* field.

Step 2: Store the item: The data item 10 can be stored in the *info* field using the following statement:

```
first->info = 10;
        or
(*first).info = 10;
```



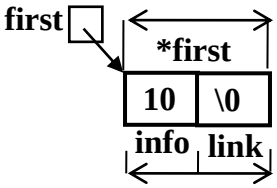
After executing the above statement, the data item 10 is stored in *info* field of *first* as shown in above figure.

Step 3: Store NULL character: After creating the above node, if we do not want *link* field to contain address of any other node, we can store '\0' (NULL) in *link* field as shown below:

first->link = NULL;

or

(*first).link = NULL;



Thus, using above three steps we can create a node with specified data item as shown in above figure.

8.1.8 Delete a node

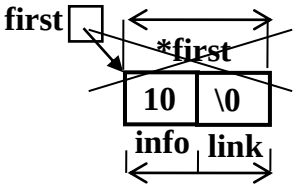
A node which is no longer required can be deleted using free() function in C as shown below:

free (variable);

For example, when above statement is executed, the memory space allocated for the node is de-allocated and returned to operating system so that it can be used by some other program. The memory de-allocated after executing:

free(first);

can be pictorially represented on the right hand side.



8.2 Operations on singly linked lists

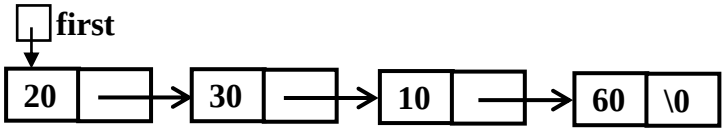
Now, let us see "What are the various operations that can be performed on singly linked lists?" The basic operations that can be performed on a linked list are shown below:

Various Operations of linked lists
- Inserting a node into the list
- Deleting a node from the list
- Search in a list
- Display the contents of list

8.2.1 Insert a node at the front end

In this section, let us see "How to insert a node at the front end of the list?"

Design: To design the function easily, let us consider a linked list with 4 nodes. Here, pointer variable ***first*** contains address of the first node of the list as shown below:

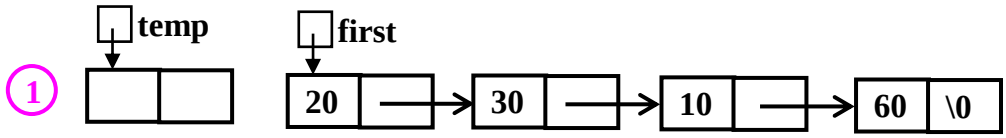


Let us try to insert the *item* **50** at the front end of the above list. The sequence of steps to be followed are shown below:

Step 1: Create a node using getnode() function as shown below:

(1) temp = getnode()

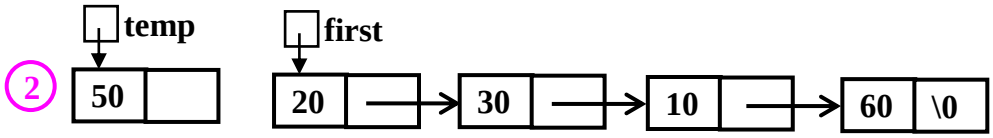
The pictorial representation after executing above statement is shown below:



Step 2: Copy the ***item*** **50** into *info* field of *temp* using the following statement:

(2) temp->info = item;

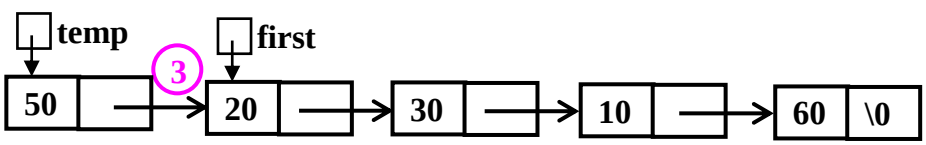
The pictorial representation after executing above statement is shown below:



Step 3: Copy address of the first node of the list stored in pointer variable *first* into *link* field of *temp* using the statement:

(3) temp->link = first;

The pictorial representation after executing above statement is shown below:



Step 4: Now, a node *temp* has been inserted and observe from the figure that *temp* is the first node. Let us always return address of the first node using the statement:

④ return temp;

The complete function is shown below:

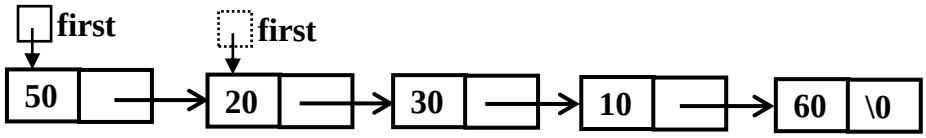
Example 8.3: C Function to insert an item at the front end of the list

```
NODE insert_front(int item, NODE first)
{
        NODE temp;
        temp = getnode();              /*obtain a node from the available list*/
        temp->info = item;             /* Insert an item into new node */
        temp->link = first;            /* insert new node at the front of list */
        return temp;                   /* return the new first node */
}
```

The above function should be called in the calling function as shown below:

```
first = insert_front(item, first);
```

After executing the above statement, the variable *first* in the calling function contains address of the first of the list as shown below:



The variable *first* shown using dotted lines contains the address of the first node of the list before insertion whereas the variable *first* shown using thick lines contains the address of the first node of the list after insertion.

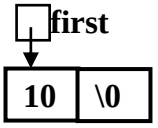
8.2.2 Create a linked list

Now, the question is "How to create a linked list?"

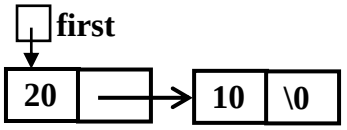
The answer is very simple. Call the insert_front() function described in previous section repeatedly as shown below:

first = insert_front(item, first);

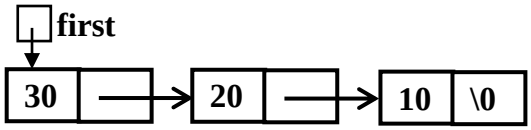
- If ***first*** is NULL and *item* is 10, when above statement is executed, a linked list with only one node is created as shown in figure on right hand side:



- If above statement is executed for the second time with *item* 20, a new node is inserted at the front end and there by number of nodes in the list is 2 as shown in figure:



- If above statement is executed for the third time with *item* 30, a new node is inserted at the front end and there by number of nodes in the list is 3 as shown in figure:

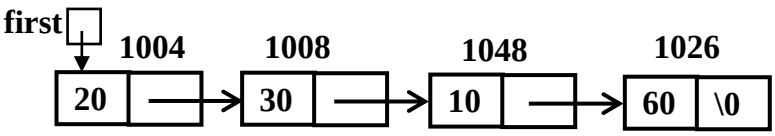


- Thus, if the function insert_front() is called ***n*** times, we have a list with *n* nodes.

Note: Thus, repeatedly inserting an element at the front end of the list we can create a linked list.

8.2.3 How to find address of last node in the list

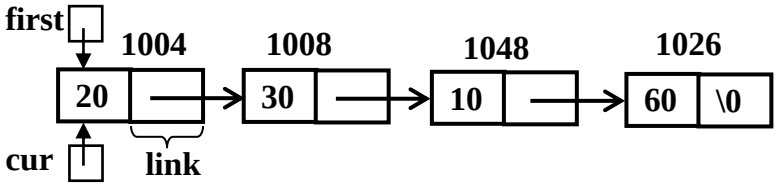
Consider the following singly linked list where the variable *first* contains address of the first node of the list.



We need to find the address of the last node. For this to happen, we have to start from the first node. Instead of updating *first*, let us use another variable say *cur*. To start with, the variable *cur* should contain the address of the first node. This is achieved using the statement:

cur = first;

Now, the list can be represented as shown below:

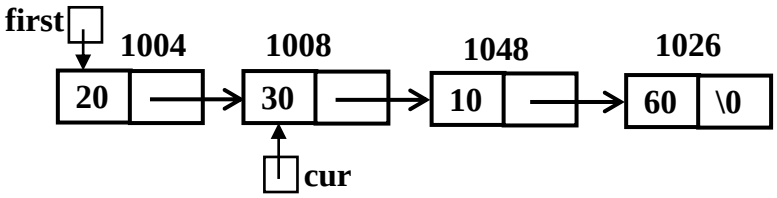


Note: What is first->info and cur->info? It is 20. What is first->link and cur->link? It is 1008 (which is the address of the next node)

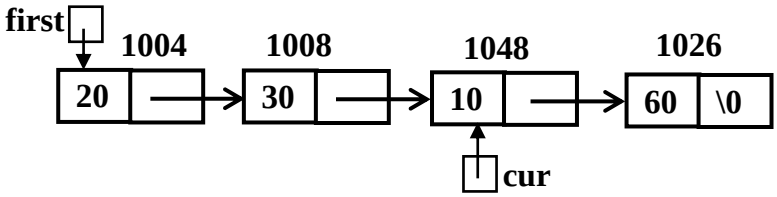
Let us update *cur* so that it contains address of the next node. This can be done by copying cur->link (i.e, 1008) to *cur* using the following code:

```
cur = cur->link;
```

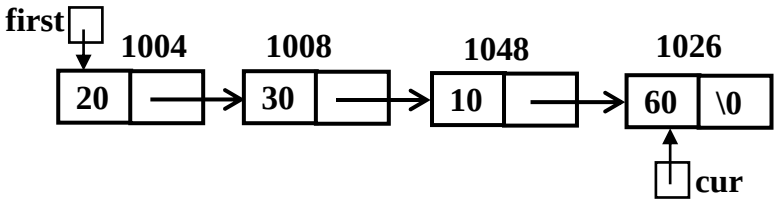
After executing the above instruction, *cur* contains address of the next node as shown below:



If we execute the instruction "cur = cur->link" again, *cur* contains address of the next node as shown below:



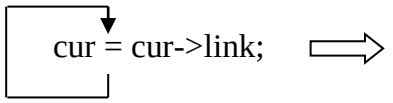
If we execute the instruction "cur = cur->link" again, *cur* contains address of the next node as shown below:



Note: Observe that, now the variable *cur* contains address of the last node of the list.

Now the question is "When we say that given node is the last node of the list?" If link field of a node contains NULL, then that node is the last node of that list. In the above list, cur->link is NULL. So, *cur* is the last node of the above list.

So, if *cur* contains the address of the first node, keep updating *cur* as long as link field of *cur* is not NULL as shown in figure below:



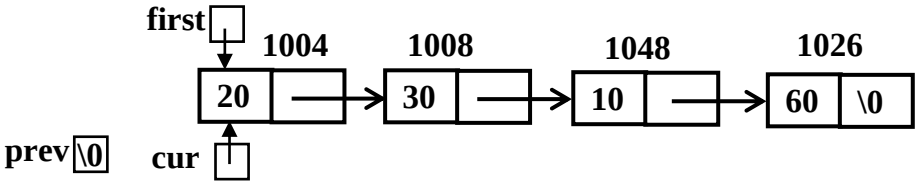

```
while ( cur->link != NULL)
{
        cur = cur->link
}
```

Now, given the variable *first* which contains address of the first node, we can find address of the last node as shown below:

```
// find the address of last node of the list
cur = first;
while (cur->link != NULL)
{
        cur = cur->link;
}
```

8.2.4 How to find last node and last but one in the list

Consider the following list:



If *cur* contains address of the first node of the list, what is the previous node? The previous node does not exit and so we say *prev* is NULL. The code for the above situation can be written as shown below:

```
prev = NULL;
cur = first;
```

Now, we know how to update *cur* to get the address of the last node. The code for this can be written as shown below: (see previous section)

```
while (cur->link != NULL)
{
        cur = cur->link;
}
```

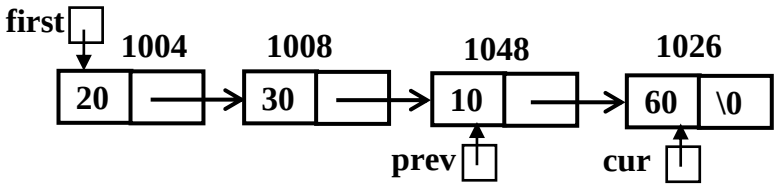
Now, before updating *cur* inside the loop using "cur = cur->link", let us copy *cur* to *prev*. The above code can be modifies as shown below:

```
while (cur->link != NULL)
{
        prev = cur;
        cur = cur->link;
}
```

After the loop, the variable *cur* contains address of the last node and the variable *prev* contains address of the *prev* node. Now, the complete code to find the address of the last node and last but one node is shown below:

```
/* Find address of last node and last but one node */
prev = NULL;
cur = first;
while (cur->link != NULL)
{
        prev = cur;
        cur = cur->link;
}
```

The given linked list after executing the above code can be written as shown below:



Note: The variable *first* contains address of the first node, the variable *cur* contains address of the last node, the variable *prev* contains address of last but one node.

8.2.5 Display singly linked list

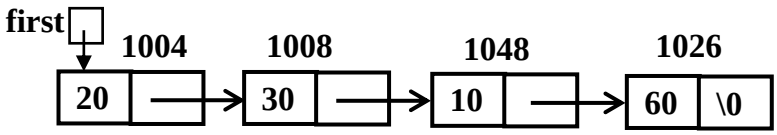
Now, let us see "How to display the contents of the list?"

Design: The C function for this can be designed by considering two cases as shown below:

Case 1: List is empty: If the list is empty, it is not possible to display the contents of the list. The code for this can be written as shown below:

```
if (first == NULL)
{
        printf("List is empty\n");
        return;
}
```

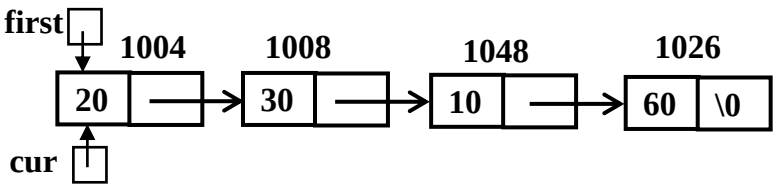
Case 2: List is exiting: Consider the linked list shown below with four nodes where the variable *first* contains address of the first node of the list.



Initialization: Use another variable say *cur* to point to the beginning of the list. This can be done by copying *first* to *cur* as shown below:

```
cur = first;
```

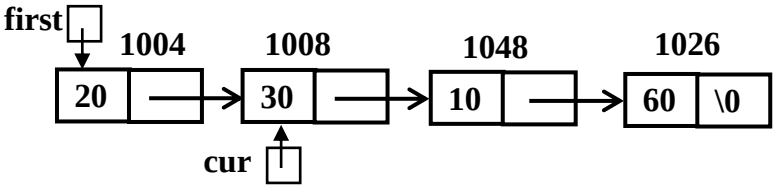
The resulting list can be written as shown below:



Display: Now, display *info* field of *cur* node and update *cur* as shown below:

```
printf("%d  ", cur->info);                    // Output: 20
cur = cur->link;                              //  cur = 1008
```

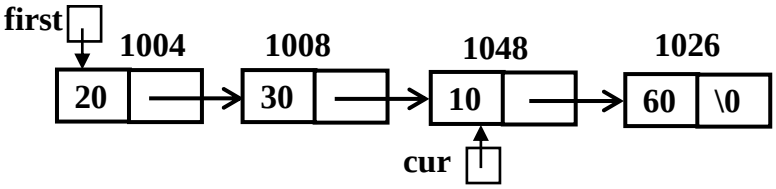
The resulting list is shown below:



Now, display *info* field of *cur* node and update *cur* as shown below:

```
printf("%d  ", cur->info);                    // Output: 30
cur = cur->link;                              //  cur = 1048
```

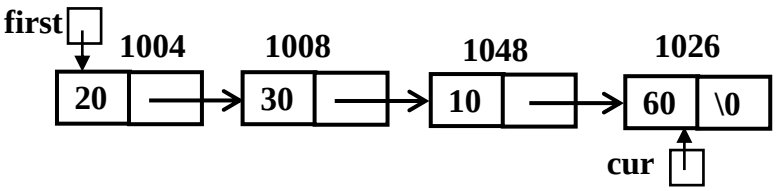
The resulting list is shown below:



Now, display *info* field of *cur* node and update *cur* as shown below:

```
printf("%d  ", cur->info);                    // Output: 10
cur = cur->link;                              // cur = 1026
```

The resulting list is shown below:



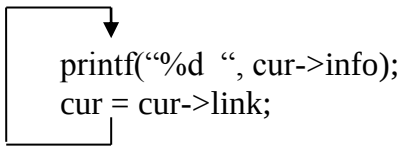
Now, display *info* field of *cur* node and update *cur* as shown below:

```
printf("%d  ", cur->info);                    // Output: 60
cur = cur->link;                              // cur = NULL
```

Finally, observe that *cur* is NULL indicating no more nodes are there to display. Observe that the following statements are repeatedly executed:

```
printf("%d  ", cur->info);
cur = cur->link;
```

as long as *cur* is not NULL. Once *cur* is NULL, the displaying of all the nodes is over. The code for this can be written as shown below:




```
while (cur != NULL)
{
        printf("%d ", cur->info);
        cur = cur->link;
}
```

Now, the complete C function to display the contents of the list is shown below:

Example 8.4: C function to display the contents of linked list

```
void display(NODE first)
{
        NODE cur;
        ..........................................................................
        if ( first == NULL )                    /* Check for empty list */
        {                                                    Case 1
                printf("List is empty\n");
                return;
        }
        ..........................................................................
        printf("The contents of  singly linked list\n");     Case 2
        cur = first;                            /* Holds address of the first node */
        while ( cur != NULL )                   /* As long as no end of list */
        {
                printf("%d  ",cur->info);       /* Display the info field of node */
                cur = cur->link;                /* Point to the next node */
        }
        printf("\n");
        ..........................................................................
}
```

8.2.6 Delete a node from the front end

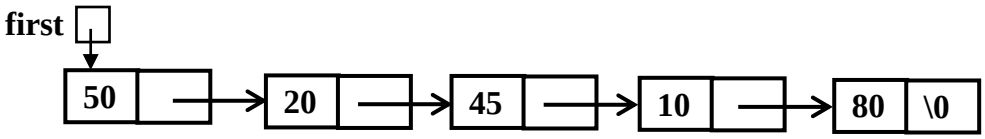
Now, let us see "How to delete a node from the front end of the list?"

Design: A node from the front end of the list can be deleted by considering various cases as shown below:

Case 1: List is empty: If the list is empty, it is not possible to delete a node from the list. In such case, we display "List is empty" and **return** NULL. The code for this can be written as shown below:

```
if (first == NULL)
{
        printf ("List is empty\n");
        return NULL;
}
```

Case 2: List is exiting: Consider the list with five nodes where the variable *first* contains address of the first node of the list.

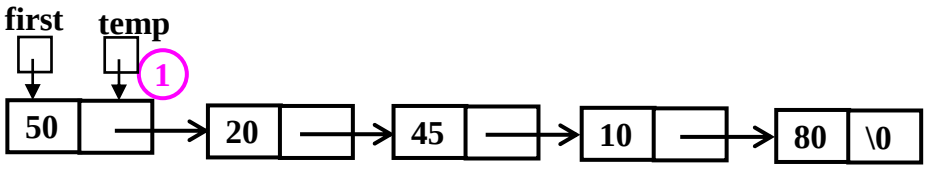


Given the address of the first node of the list, we need to know the address of the second node of the list. This is because, after deleting the first node, the second node will be the first node of the list. The sequences of steps to be followed while deleting an element are shown below:

Step 1: Use a pointer variable *temp* and store the address of the first node of the list as shown in figure below. This is achieved using the following statement:

(1) temp = first;

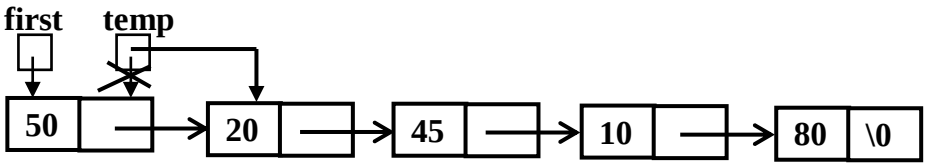
Now, the list can be written as shown below where *temp* and *first* points to first node.



Step 2: Update the pointer *temp* so that the variable *temp* contains address of the second node. This is achieved using the statement:

(2) temp = temp->link;

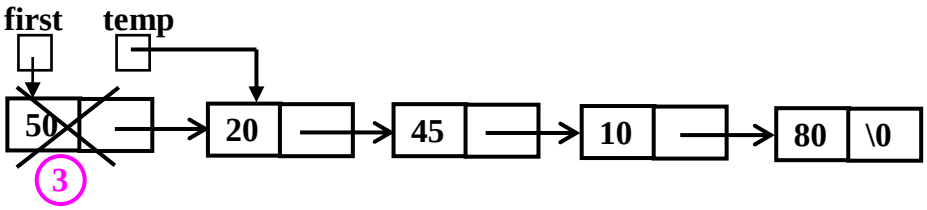
Now, the list can be written as shown below where *temp* points to second node.



Step 3: Note that variable *first* points to first node of the list and *temp* points to the second node of the list. Now, display *info* field of the node first node to be deleted and de-allocate the memory as shown below:

```
        printf("Item deleted = %d\n", first->info);
3       free (first);
```

After executing the above statement, the node pointed to by *first* is deleted and is returned to operating system. The resulting linked list is shown in fig. below:



Step 4: Once the node *first* is deleted, observe that node *temp* is the first node (see above list). So, return *temp* as the first node to the calling function using the statement:

```
return temp;
```

Now, the complete algorithm in C can be written as shown below:

Example 8.5: C function to delete an item from the front end of the list

```
NODE delete_front(NODE first)
{
    NODE temp;

    if ( first == NULL )          /* Check for empty list */
    {
        printf("List is empty cannot delete\n");
        return NULL;              // We can replace NULL with first also
    }

    temp = first;                 /* Retain address of the node to be deleted */
```

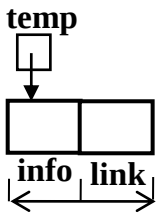
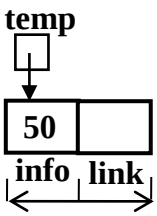
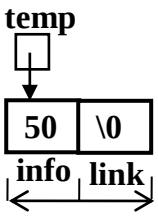
```
        temp = temp->link;          /* Obtain address of the second node */
        printf("Item deleted = %d\n",first->info);  /* access first node */
        free(first);                /* delete the front node */
        return temp;                /* return address of the first node */
}
```

8.2.7 Insert a node at the rear end

Now, let us see "How to insert a node from the rear end of the list?"

Design: Let us try to insert the *item* **50** at the rear end of the list. The sequence of steps to be followed are shown below:

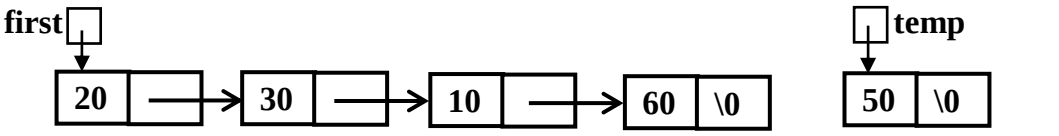
Step 1: Create a node using getnode() function (see example 8.2) then insert the *item* say 50 using the following statements:

temp = getnode();	temp->info = item;	temp->link = NULL;
		

Step 2: If list is empty, the above node can be returned as the first node of the list. This can be done using the statement:

```
        if (first == NULL) return temp;
```

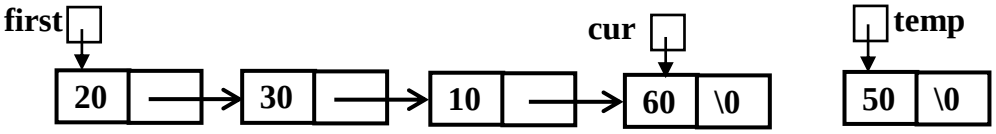
Step 3: If the list is existing, we have to insert *temp* at the end of the list



To insert at the end, we have to find address of the last node. The code to find the address of the last node can be written (for details see section 8.2.3) as shown below:

```
/* Find the address of the last node */
cur = first;
while (cur->link != NULL)
{
        cur = cur->link;
}
```

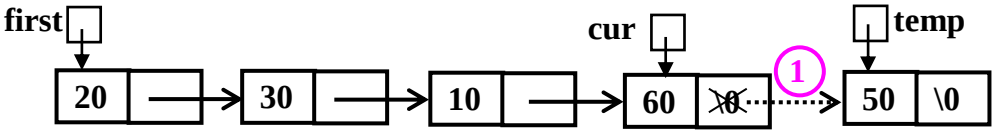
The pictorial representation of the list after executing the above code can be written as shown below:



Step 4: Insert node at the end: Looking at the above list, we can easily insert *temp* at the end of *cur*. This is achieved by copying *temp* to cur->link as shown below:

```
cur->link = temp;
```
(1)

The list obtained after executing the above code can be written as shown below:

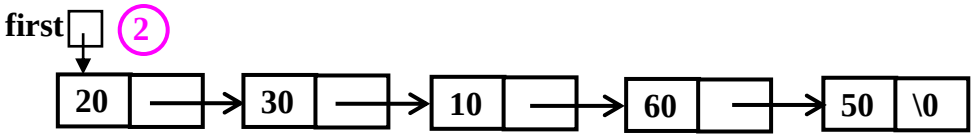


Note: The \0 (NULL character) is replaced by node pointed to by *temp*.

Step 5: Observe from the above list that *first* contains address of the first node of the list. So, we return *first*

```
return first;
```
(2)

Now, the complete list after inserting at the rear end is shown below:



The complete C function to insert an item at the rear end of the list is below:

Example 8.6: Function to insert an item at the rear end of the list

```
NODE insert_rear(int item, NODE first)
{
        NODE    temp;       /* Points to newly created node */
        NODE    cur;        /* To hold the address of the last node */
        ..........................................................................
        temp = getnode();   /* Obtain a new node and copy the item */
        temp->info = item;                                          Step 1
        temp->link = NULL;
        ..........................................................................
        /* If list is empty return new node as the first node*/
        if ( first == NULL ) return temp;                           Step 2
        ..........................................................................
        /* If list exists, obtain address of the last node */      Step 3
        cur = first;
        while ( cur->link != NULL )
        {
                cur = cur->link;
        }
        ..........................................................................
        cur->link = temp;           /* Insert the node at the end */    Step 4
        ..........................................................................
        return first;               /* return address of the first node */ Step 5
}
```

8.2.8 Delete a node from the rear end

Now, let us see "How to delete a node from the rear end of the list?"

Design: A node from the rear end of the list can be deleted by considering various cases as shown below:

Case 1: List is empty: If the list is empty, it is not possible to delete the contents of the list. In such case we display appropriate message and return. The code for this can be written as shown below:

```
if (first == NULL)
{
        printf("List is empty\n");
        return NULL;
}
```

Case 2: List contains only one node: Consider a list with single node shown in figure below:

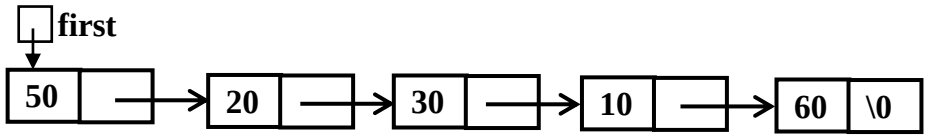
Before deleting After deleting

Note: If *link* field of *first* contains NULL, it indicates that there is only one node.

If only one node is present, it can be deleted using free() function. Then, we return NULL indicating list is empty. The code for this case is shown below.

```
/*If only one node is present, delete it */
if ( first ->link == NULL )
{
        printf ("The item to be deleted is %d\n", first->info);
        free(first);                    /* Delete and return to OS */
        return NULL;                    /* return empty list */
}
```

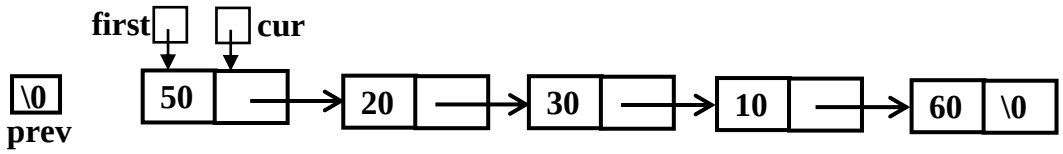
Case 3: List contains more than one node: Consider a list with five nodes shown in figure below:



Step 1: To delete the last node, we should know the address of the last node and last but one node. For this reason, we use two pointer variables: *cur* and *prev***.** Initially, *cur* points to the first node and *prev* points to \0 (null). This is achieved using the following statements:

```
prev = NULL;
cur = first;
```

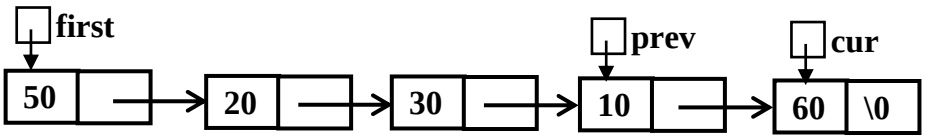
Now, the linked list looks as shown below:



Step 2: Now, update *cur* and *prev* so that *cur* contains address of the last node and *prev* contains address of the last but one node. This can be achieved by using the following statements (see section 8.2.4 for detailed explanation).

```
while( cur->link != NULL )
{
        prev = cur;
        cur = cur->link;
}
```

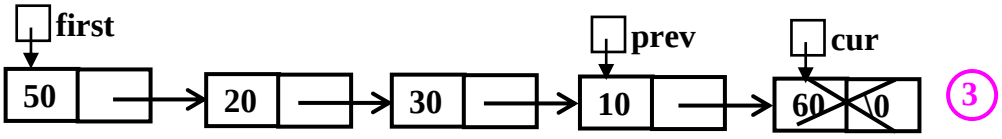
After executing the above loop, the variable *cur* contains address of the last node and *prev* contains address of last but one node as shown in figure below:



Step 3: To delete the last node pointed to by *cur*, the function free() is used as shown below:

```
(3) printf("Item deleted = %d\n", cur->info);        // Item deleted = 60
    free(cur);
```

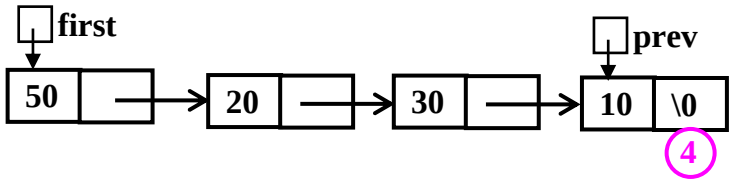
After executing the above statements, the last node is deleted and the list is shown below:



Step 4: Once the last node is deleted (the node shown using cross symbol in above figure), the node pointed to by *prev* should be the last node. This is achieved by copying NULL to *link* field of *prev* as shown below:

```
(4) prev->link = NULL;        /* Node pointed to by prev is the last node */
```

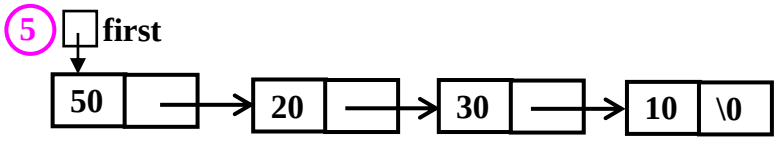
After executing the above statement, the list shown in previous step can be written as shown below:



Step 5: Finally return address of the first node.

```
return first;   (5)          /* return address of the first node */
```

Now, the linked list as seen from the calling function is shown below:



The complete C function to delete a node from the rear end of the list is shown below:

Example 8.7: Function to delete an item from the rear end of the list

```
NODE delete_rear(NODE first)
{
        NODE cur, prev;
        .....
        if (first == NULL)              /* Check for empty list */
        {                                                              Case 1
                printf("List is empty cannot delete\n");
                return first;
        }
        .....
        if ( first ->link == NULL )     /*Only one node is present and delete it */
        {
                printf ("The item to deleted is %d\n",first->info);    Case 2

                free(first);            /* return to availability list */

                return NULL;            /* List is empty so return NULL */
        }
```

Case 3

```
        /* Obtain address of the last node and just previous to that */
        prev = NULL;
        cur = first;
        while( cur->link != NULL )
        {
                prev = cur;
                cur = cur->link;
        }

        printf("The item deleted is %d\n",cur->info);
        free(cur);                      /* delete the last node */

        prev->link = NULL;              /* Make last but one node as the last node */

        return first;                   /* return address of the first node */
}
```

8.3 Stacks using linked lists

We know that stack is a special type of data structure where elements are inserted at one end and elements are deleted from the same end. That is, if an element is inserted at front end, an element has to be deleted from front end. If an element is inserted at rear end, an element has to be deleted from the rear end. Thus, a stack can be implemented using the following functions:

- insert_front()
- delete_front()
- display()

OR

- insert_rear()
- delete_rear()
- display()

The C program to implement stacks using singly linked list with the help of insert_front(), delete_front() and display() functions is below:

Example 8.8: Program to implement stacks using singly linked list

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
struct node
{
        int             info;
        struct node     *link;
};
```

```
typedef struct node* NODE;

/* Include: Example 8.2: Function to get a new node from the operating system */
/* Include: Example 8.3: Function to insert an item at the front end of the list*/
/* Include: Example 8.4: Function to display the contents of the list */
/* Include: Example 8.5: Function to delete an item from the front end of the list */

void main()
{
        NODE        first;
        int         choice, item;

        first  =  NULL;

        for (;;)
        {
                printf("1:Insert_Front    2:Delete_Front\n");
                printf("3:Display         4:Exit\n");
                printf("Enter the choice\n");
                scanf("%d", &choice);

                switch(choice)
                {
                        case 1:
                                printf("Enter the item to be inserted\n");
                                scanf("%d", &item);
                                first = insert_front (item, first);
                                break;
                        case 2:
                                first = delete_front(first);
                                break;
                        case 3:
                                display(first);
                                break;
                        default:
                                exit(0);
                }
        }
}
```

Note: In the function main(), we can use insert_rear() and delete_rear() functions and still implement stack operations using linked list.

8.4 Queue using linked lists

We know that queue is a special type of data structure where elements are inserted at one end and elements are deleted at the other end. It is a FIFO data structure as we have already seen in previous chapter. That is, if an element is inserted at front end, an element has to be deleted from rear end. If an element is inserted at rear end, an element has to be deleted from the front end. Thus, a queue can be implemented using the following functions:

- insert_front()
- delete_rear()
- display()

OR

- insert_rear()
- delete_front()
- display()

The C program to implement queues using singly linked list with the help of insert_rear(), delete_front() and display() functions is below:

Example 8.9: Program to implement queues using singly linked list

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
struct node
{
        int             info;
        struct node     *link;
};

typedef struct node* NODE;

/* Include: Example 8.2: Function to get a new node from the operating system */
/* Include: Example 8.4: Function to display the contents of the list */
/* Include: Example 8.5: Function to delete an item from the front end of the list */
/* Include: Example 8.6: Function to insert an item at the rear end of the list*/

void main()
{
        NODE            first;
        int             choice, item;
```

```
        first = NULL;

        for (;;)
        {
                printf("1:Insert_Front    2:Delete_Front\n");
                printf("3:Display         4:Exit\n");
                printf("Enter the choice\n");
                scanf("%d", &choice);

                switch(choice)
                {
                        case 1:
                                printf("Enter the item to be inserted\n");
                                scanf("%d", &item);
                                first = insert_rear (item, first);
                                break;
                        case 2:
                                first = delete_front(first);
                                break;
                        case 3:
                                display(first);
                                break;

                        default:
                                exit(0);
                }
        }
}
```

Note: In the above function main(), we can use insert_front() and delete_rear() functions and still implement queue operations using linked list.

Note: Let us see "What is enqueuer operation? What is dequeuer operation?" Inserting an element into queue is called *enqueue operation* and deleting an element from queue is called *dequeue operation*. But, there is a difference between *dequeue operation* and *dequeue*. Simply *dequeue* means is a *double ended queue*.

8.5 Double ended queue using linked lists

We know that double ended queue is a special type of data structure where elements can be inserted either from front end or rear end and elements can be deleted from front end and rear end. That is, insertions and deletions are possible from both ends.

So, to implement double ended queue, we have to use insert_front(), insert_rear(), delete_front() and delete_rear() functions.

The C program to implement double ended queue is shown below:

Example 8.10: Program to implement queues using singly linked list

```
#include <stdio.h>
#include <stdlib.h>


struct node
{
        int             info;
        struct node     *link;
};

typedef struct node* NODE;

/* Include: Example 8.2: Function to get a new node from the operating system */
/* Include: Example 8.3: Function to insert an item at the front end of the list*/
/* Include: Example 8.4: Function to display the contents of the list */
/* Include: Example 8.5: Function to delete an item from the front end of the list */
/* Include: Example 8.6: Function to insert an item at the rear end of the list*/
/* Include: Example 8.7: Function to delete an item from the rear end of the list*/

void main()
{
        NODE            first;
        int             choice, item;

        first  =  NULL;

        for (;;)
        {
                printf("1:Insert_Front    2:Insert_Rear\n");
                printf("3:Delete_Front   4:Delete_Rear\n");
                printf("5:Display          6:Exit\n");
                printf("Enter the choice\n");
                scanf("%d", &choice);
```

```
            switch(choice)
            {
                  case 1:
                        printf("Enter the item to be inserted\n");
                        scanf("%d", &item);
                        first = insert_front (item, first);
                        break;

                  case 2:
                        printf("Enter the item to be inserted\n");
                        scanf("%d", &item);
                        first = insert_rear (item, first);
                        break;

                  case 3:
                        first = delete_front(first);
                        break;

                  case 4:
                        first = delete_rear(first);
                        break;

                  case 5:
                        display(first);
                        break;

                  default:
                        exit(0);
            }
      }
}
```

8.6 Additional list operations

In this section, we discuss other list operations.

8.6.1 Find the length of the list

Now, let us see "How to find the length of linked list or number of nodes present in the linked list?" The solution is very simple and straight forward technique. Now, tell me what the following code does?

```
cur = first;
while (cur != NULL)                 /* As long as no end of list */
{
        printf("%d\n", cur->info);  /* Display the info of a node */
        cur = cur->link;            /* point cur to the next node */
}
```

The above code displays the *info* field of each node in the list. But, we are supposed to find the number of nodes in the list. This can be achieved very easily by replacing ***printf*** statement by *count++* and initialize *count* to 0 before the loop. Now, the complete function can be written as shown below:

Example 8.11: Function to find the length of the linked list

```
int Length(NODE first)
{
        NODE cur;
        int      count = 0;

        if ( first == NULL )   return 0;      /* Check for empty list. If so, return 0 */

        cur = first;
        while (cur != NULL)                   /* As long as no end of list */
        {
                count++;                      /* Update count by 1 */
                cur = cur->link;              /* point cur to the next node */
        }

        return count;                         /* Return the length of list */
}
```

8.6.2 Search for an item in a list

Now, let us see "How to search for an item in a linked list?" Searching for a *key* item is very simple and straight forward technique. We know how to display *info* of each node. The partial code can be written as:

```
cur = first;
while (cur != NULL)                 /* As long as no end of list */
{
        printf("%d\n", cur->info);  /* Display the info of a node */
        cur = cur->link;            /* point cur to the next node */
}
```

Replace the printf() by the following statement:

```
if (key == cur->info) break;
```

When control comes out of the loop, if *cur* is still NULL then key not found. Otherwise, key found. The partial code to check for an item in the list can be written as shown below:

```
cur = first;
while (cur != NULL)                    /* As long as no end of list */
{
        if (key == cur->info) break;   /* key found */
        cur = cur->link;               /* point cur to the next node */
}

if (cur == NULL)                       /* If end of list, key not found */
{
        printf("Search is unsuccessful\n");
        return;
}

printf("Serach is successful\n");      /* Yes, key found */
```

The complete C function to search for a **key** item is shown below:

Example 8.12: Function to search for key item in the list

```
void search(int key, NODE first)
{
        NODE cur;

        if ( first == NULL )           /* check for empty list */
        {
                printf("List is empty\n");
                return;
        }

        cur = first;
        while (cur != NULL)            /* As long as no end of list */
        {
                if (key == cur->info) break;   /* If found go out of the loop */
                cur = cur->link;               /* point cur to the next node */
        }
```

```
        if (cur == NULL)                    /* If end of list, key not found */
        {
                printf("Search is unsuccessful\n");
                return;
        }

        printf("Serach is successful\n");   /* Yes, key found */
}
```

"Assume that we have a list of integers. Create a function that searches for an integer number *item*. If *item* is present in the list, the function should return a pointer to the node that contains *item*. Otherwise, it should return NULL".

The above function should be modified as shown below:

- If *item* found, then return the pointer to the node. This can be done by replacing the keyword **break** by:

  return cur;

- If *item* not found, instead of displaying the message "Item not found" return NULL.

The complete C function can be written as shown below:

Example 8.13: Function to search for key item in the list

```
NODE search(int key, NODE first)
{
        NODE cur;

        if ( first == NULL )   return NULL;     /* Search for empty list */

        /* Compare one after the other */
        cur = first;
        while (cur != NULL)                     /* As long as no end of list */
        {
                if (key == cur->info) return cur;   /* If found go out of the loop */
                cur = cur->link;                    /* point cur to the next node */
        }

        return NULL;                            /* key not found */
}
```

8.6.3 Delete a node whose information field is specified

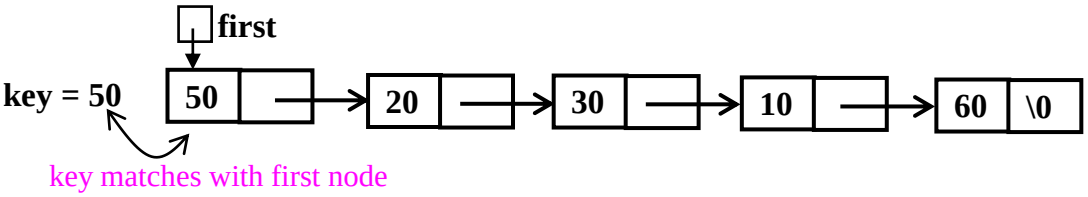
Now, let us see "How to delete a node whose **info** field is specified?"

Design: To design the function easily, let us follow the sequence of steps shown below:

Step 1: Check for empty list. The equivalent code can be written as shown below:

```
if (first == NULL)
{
        printf("List empty, Search fails\n");
        return NULL;
}
```

Step 2: Check for *key* in the first node in the list. If *key* matches with the first node of the list, the node at the front end of the list has to be deleted as shown below:



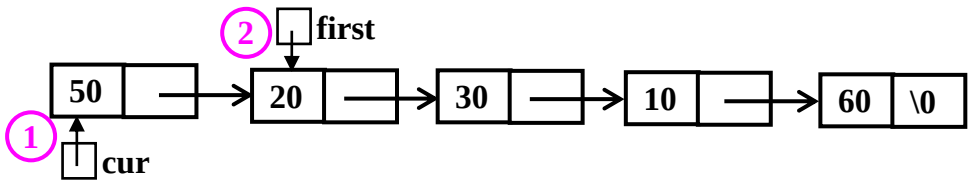
Now, save the address of *first* node into *cur* using the statement:

```
cur = first;   (1)
```

and update *first* to point to the next node using the statement:

```
first = first -> link;   (2)
```

Now, the pictorial representation of the list is shown below:



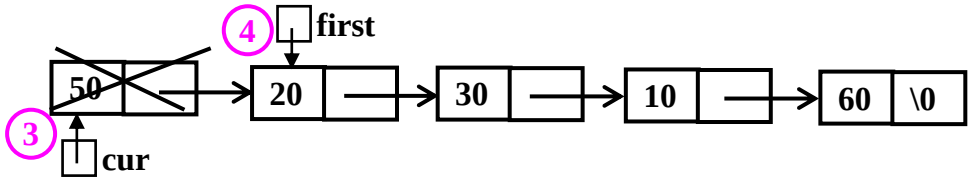
Now, delete the first node pointed to by *cur*. This can be done using the statement:

free (cur); (3)

and return the address of the *first* using the statement:

return first; (4)

After executing the above two statements, the list can be pictorially represented as shown below:



Thus, if key matches with first node of the list, the first node can be deleted and the corresponding code can be written as shown below:

```
if ( key == first->info )
{
        cur = first;            /* Save the address of the first node */
        first = first->link;    /* Point first to second node in the list */
        free(cur);              /* Delete the first node */
        return first;           /* Return second node as the first node */
}
```

Step 3: Search for the key: Control comes to step 3, if *key* is not present in the first node of the list. It may be present or may not be present in subsequent nodes. So, we have to search for *key* in the list. The search code given in previous section is repeated for convenience.

```
cur = first;
while (cur != NULL)                 /* As long as no end of list */
{
        if (key == cur->info) break;    /* If found go out of the loop */

        prev = cur;                 /* Save the address of cur node */
        cur = cur->link;            /* point cur to the next node */
}
```

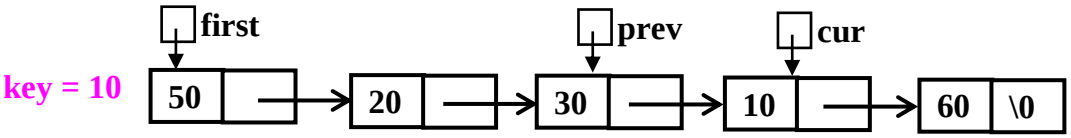
```
if (cur == NULL)                    /* If end of list, key not found */
{
        printf("Search is unsuccessful\n");
        return first;
}
printf("Serach is successful\n");   /* Yes, key found */
```

Note: Observe that the statement:

```
prev = cur;
```

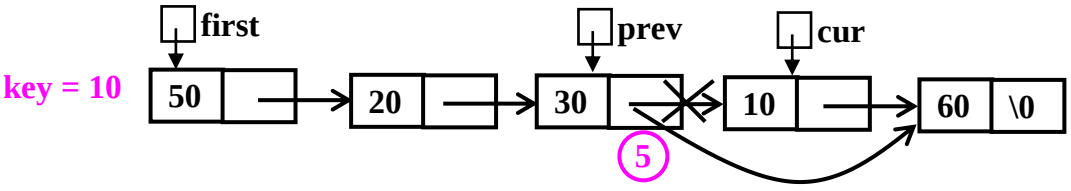
is added inside the while loop, just before updating *cur* to the next node. This is necessary to delete the *cur* node.

Now, if *10* is the item to be deleted, after executing the above statements, we get the message "Search is successful" and at this instant, the linked list looks as shown below:



Observe that *cur* contains address of the node to be deleted and *prev* contains address of predecessor of the node to be deleted.

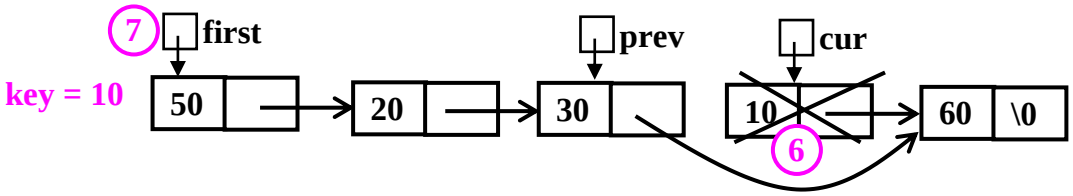
Step 4: Delete the node: Before deleting the *cur* node we need to adjust the pointers. That is, copy "*cur->link*" to "prev->link" as shown below:



The code corresponding to above activity can be written as shown below:

```
prev->link = cur->link;     (5)
```

After executing the above statement, prev->link points to node containing 60 and thus, *cur* node is isolated from the list and the linked list can be pictorially represented as shown below:



Now, the node *cur* can be removed using free() function as shown below:

```
free (cur);    6
```

Step 5: Finally return the address of the first node. The code for this is shown below.

```
return first;  7
```

The complete C function to delete a node whose information field is specified is shown below:

Example 8.14: Function to delete a node whose info field is specified

```
NODE delete_info(int key, NODE first)
{
        NODE prev, cur;
        if ( first == NULL )            /* Check for empty list */
        {
                printf("List is empty\n");
                return NULL;
        }
        if ( key == first->info )    /* If key is present in first node, delete that node */
        {
            1 cur = first;              /* Save the address of the first node */
            2 first = first->link;      /* Point first to second node in the list */
            3 free(cur);                /* Delete the first node */
            4 return first;             /* Return second node as the first node */
        }
```

```
        /* Search for the node to be deleted */
        prev = NULL;
        cur = first;
        while (cur != NULL)                    /* As long as no end of list */
        {
                if (key == cur->info) break;   /* If found go out of the loop */

                prev = cur;                    /* Save the address of cur node */
                cur = cur->link;               /* point cur to the next node */
        }

        if (cur == NULL)                       /* If end of list, key not found */
        {
                printf("Search is unsuccessful\n");
                return first;
        }

        /* Search successful. So, delete the node */
    5   prev->link = cur->link;                /* Establish link between the
                                                  predecessor and successor */

    6   free(cur);                             /* Delete the node with info key */

    7   return first;                          /* Return address of first node */
}
```

8.6.4 Concatenate two lists

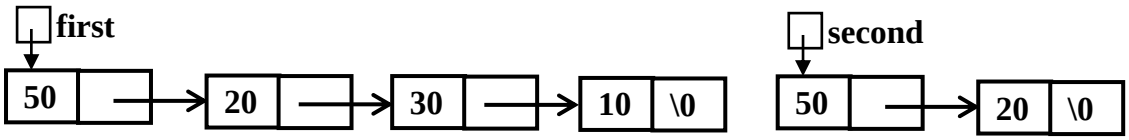
Now, let us see "What is concatenation of two lists?"

Definition: Concatenation of two lists is nothing but joining the second list at the end of the first list.

Design: Concatenation of two lists is possible if two lists exist. If one of the lists is empty, return the address of the first node of the non-empty list as shown below:

```
if (first == NULL)     return second;

if (sec == NULL)       return first;
```

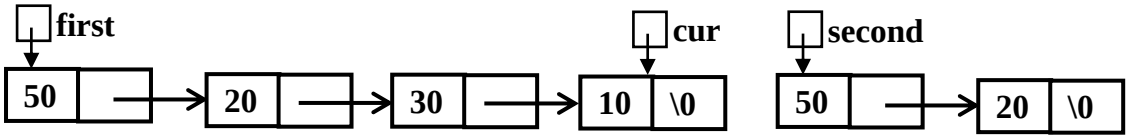
Once, control comes out of second *if-statement* it means both lists are existing. The pictorial representation of both lists are shown below:



If both lists exist, obtain the address of the last node of the first list. The code to obtain address of the last node of the first list is shown below:

```
cur = first;
while (cur->link != NULL)  /*Traverse till the end */
        cur = cur->link;
```

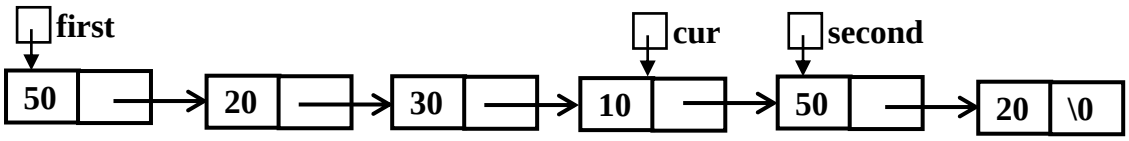
Once control comes out of the loop, pointer **cur** contains address of the last node of the first list (see the figure below).



Now, attach address of the first node of the second list identified by *second* to *cur* node using the following code:

```
cur->link = second;
```

The resulting list after executing the above statement is shown below:



Now, *first* contains the address of the first node of the concatenated list and return this node to the calling function using the statement:

```
return first;
```

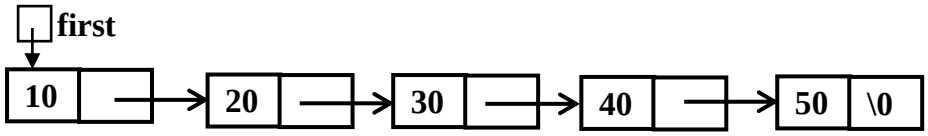
The corresponding C function is shown below:

Example 8.15: Function to concatenate two lists

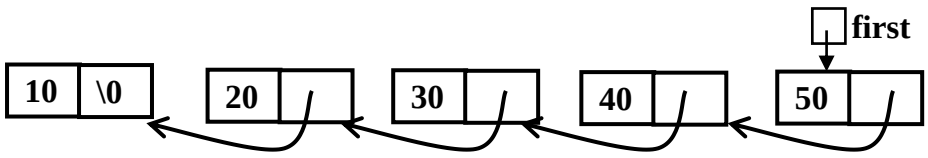
```
NODE concat (NODE first, NODE sec)
{
        NODE        cur;            /* holds the address of the last node of first list*/

        if (first == NULL)    return second;

        if (sec == NULL)      return first;

        /* Obtain address of the last node of first list*/
        cur = first;
        while ( cur->link != NULL)
                cur = cur->link;

        cur->link = sec;        /* Attach first node of second list to end of first list */

        return first;           /* Return the first node of the concatenated list */
}
```

8.6.5 Reverse (Invert) a list without creating new nodes

Now, let us see "How to reverse a given singly linked list?" Consider the list shown below:

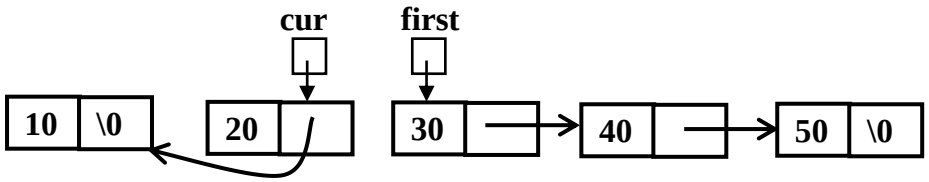


If we display the above list, the output is: 10 20 30 40 50. After reversing the list, the list is shown below:



Now, if we display the above list, the output is: 50 40 30 20 10

Design: Assume that the given list is divided into two sub lists such that the first list with two nodes is reversed and the second list with three nodes is yet to be reversed as shown below:



Now, let us assume two things:

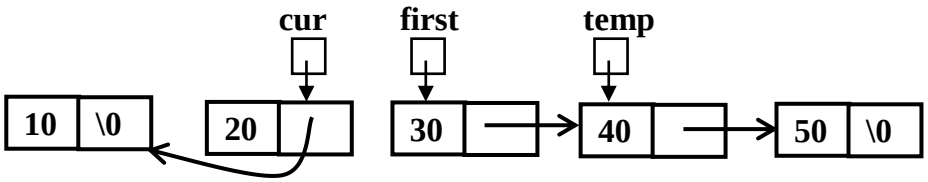
- The pointer variable *cur* always contains address of the first node of the partially reversed list (see above figure)
- The pointer variable *first* always contains address of the first node of the list to be reversed (see above figure)

With this situation in mind, let us try to reverse the list to be processed.

Step 1: Obtain the address of the second node of the list to be reversed. This can be achieved using:

```
temp = first->link;
```

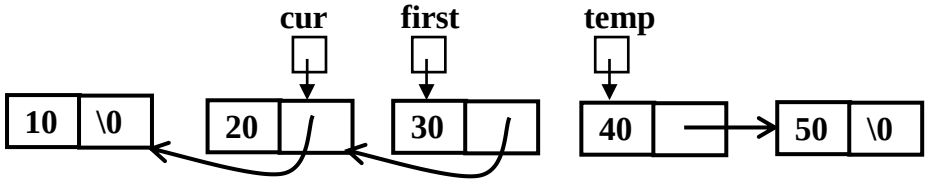
After executing the above statement, the list can be written as shown below:



Step 2: Attach the first node of the list to be reversed, to the front end of the partially reversed list. This can be achieved using:

```
first->link = cur;
```

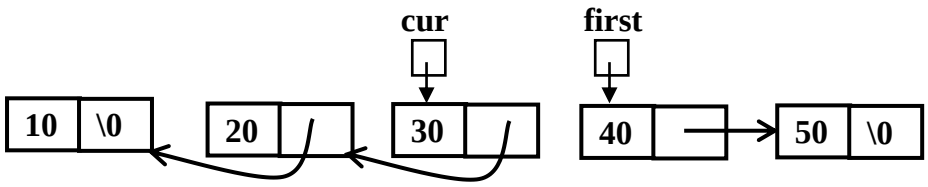
After executing the above statement, the list can be written as shown below:



Step 3: The pointer variable *cur* always should contain address of the first node of the reversed list and *first* should contain address of first node of the list to be reversed. This can be achieved using the following statements:

```
cur = first;
first = temp;
```

After executing the above statement, the list can be written as shown below:



The steps 1 through 3 can be repeatedly performed as long as list to be reversed is existing. That is, as long as the pointer variable *first* is not NULL keep executing statements from step 1 to step 3. Now, the code can be written as:

```
cur  = NULL;                    /* Initial reversed list */

while (first != NULL)
{
        temp = first->link;
        first->link = cur;
        cur = first;
        first = temp;
}
```

The C function to reverse a linked list is shown below:

Example 8.16: Function to reverse a list

```
NODE reverse (NODE first)
{
        NODE cur, temp;
        cur = NULL;                    /* Initial reversed list */
```

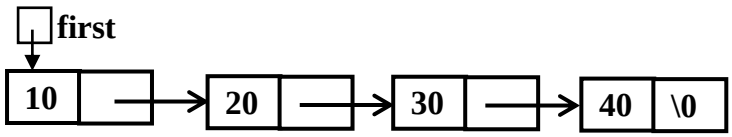
```
        while (first != NULL)
        {
                temp = first->link;     // Obtain the address of second node
                                        // of list to be reversed
                first->link = cur;      // attach first node of the list to be reversed
                                        // at the beginning of the partially reversed list
                cur = first;            // Point cur to point to newly partially reversed list
                first = temp;           // Point first to point to the list to be reversed
        }
        return cur;                     /* Contains address of the reversed list */
}
```

The complete C function to insert an item into an ordered linked list is below:

8.6.6 Creating an ordered linked list

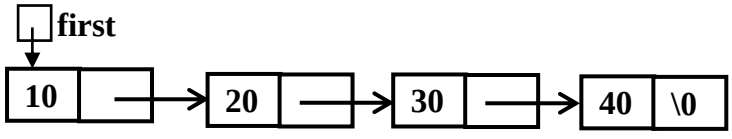
Now, let us see "What is an ordered linked list?"

Definition: A linked list in which the items are stored in some specific order is an ordered linked list. The elements in an ordered linked list can be in ascending or descending order or based on **key** information. A **key** is one or more fields within a structure that are used to identify the data. For example, an ordered linked list shown below:



Now, let us see "How to create an ordered linked list?"

Design: Assume that the ordered linked list is already existing with four nodes as shown below:



Here, the variable *first* contains address of the first node of the list.

After inserting any item into the above list, the order of list should be maintained i.e., the items in the list should be arranged in ascending/descending order only. We should see that the variable *first* always contains the address of the first node of the list. To insert an item into the list, the sequence of steps to be followed are:

Step 1: Create a node to be inserted and insert the item using the following statements:

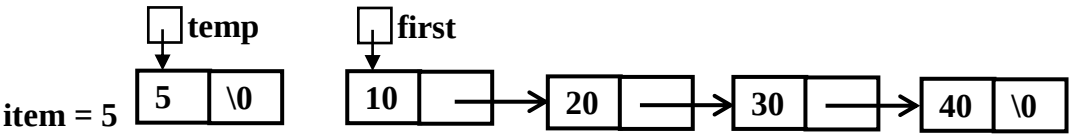
```
temp = getnode();
temp->info = item;
temp->link = NULL;
```

Step 2: If the node **temp** is inserted into the list for the first time (i.e., into the empty llist), then return **temp** itself as the first node using the following code.

```
if (first == NULL) return temp;
```

When control comes out of the above if statement, it means that the list is already existing and node *temp* has to be inserted at the appropriate place so that after inserting *temp*, the list must be arranged in ascending order. This results in following two cases: item has to be inserted at the front end or in the middle.

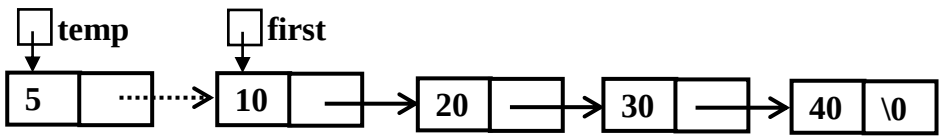
Step 3: Inserting an item at the front end of the list: This case occurs, when the node to be inserted is less than the first node of the list as shown below:



Suppose, *item* is 5. Can you tell me where it has to be inserted? It has to be inserted at the front end of the list so that the list is arranged in ascending order. This can be done by copying *first* into link field of *temp* as shown below:

```
temp->link = first.
```

This can be pictorially represented as shown below:



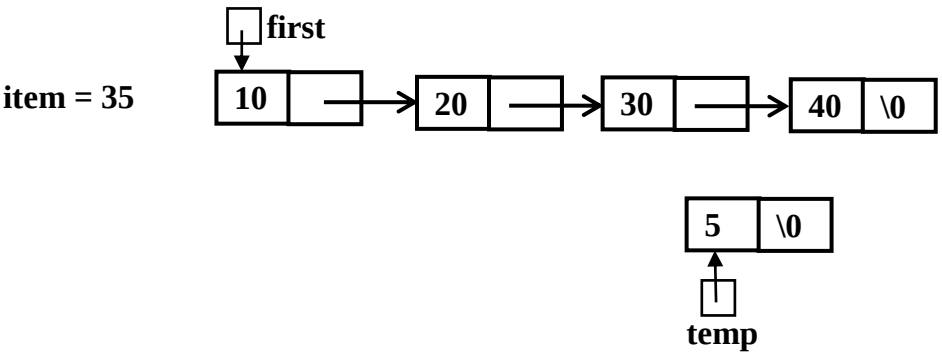
After inserting as shown in figure, we return *temp* since *temp* is the new first node of the list as shown below:

```
return temp;
```

The above two statement should be executed if and only if *item* is less than the first node of the list. Now, the code can be written as shown below:

```
if ( item <= first->info)        /* Insert at the front end of the list */
{
        temp->link = first;
        return temp;
}
```

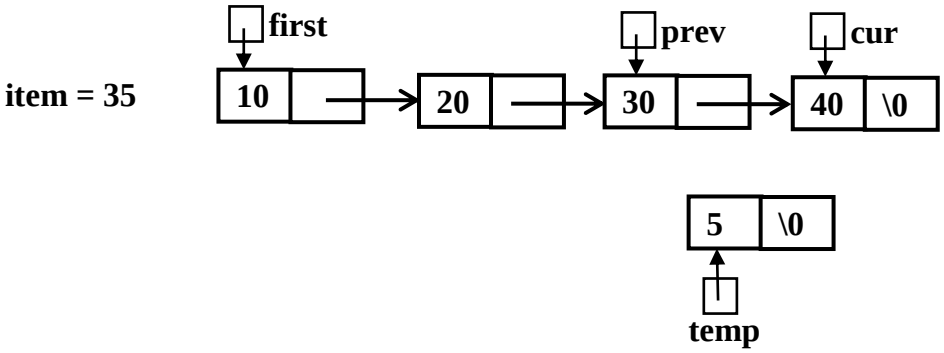
Step 4: Inserting somewhere in the middle of the list: Suppose item 35 has to be inserted into the list shown below:



To create an ordered list, node *temp* has to be inserted in between 30 and 40. So, let us find the appropriate place so that the list remains in order. The code to find the appropriate place can be written as shown below:

```
prev = NULL;
cur = first;
while (cur != NULL && item > cur->info)
{
        prev = cur;
        cur = cur=>link
}
```

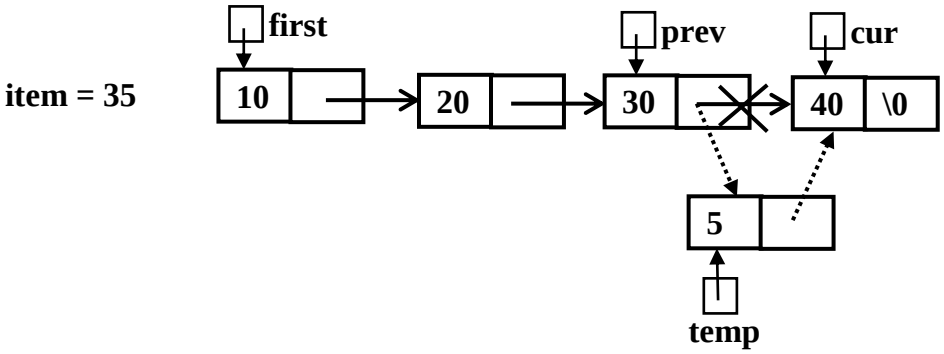
Now, after executing the above code, the linked list can be written as shown below:



Step 5: Insert at middle/end: By looking at the above list, observe that *temp* has to be inserted between *prev* and *cur*. The corresponding code is shown below:

```
prev->link = temp;     /* connect node prev with node temp */
temp->link = cur;      /* connect new node temp with next node cur */
```

After executing the above statements, the linked list can be written as shown below:



Step 6: Always we return the address of the first node as shown below:

```
return first;    /* return address of the first node */
```

Now, the complete function to create an ordered linked list can be written as shown below:

Example 8.17: C function to create an ordered linked list

```
NODE insert(int item, NODE first)
{
        NODE temp, prev, cur;

        temp = getnode();      /* obtain a node to be inserted */      STEP 1
        temp->info = item;
        temp->link = NULL;

        /* Inserting the node for the first time */                    STEP 2
        if ( first == NULL ) return temp;

        /* Inserting the node in the beginning of the list */          STEP 3
        if ( item <= first->info )
        {
                temp->link = first;
                return temp;
        }

        /* find prev and cur locations so that node temp has to be inserted */
        prev = NULL;                                                   STEP 4
        cur = first;

        while ( cur != NULL && item > cur->info )
        {
                prev = cur;
                cur = cur->link;
        }

        /* Insert the node between prev and cur */                     STEP 5
        prev->link = temp;
        temp->link = cur;

        return first;          /* return the first node */             STEP 6
}
```

8.6.7 Program to perform list operations

The complete program to implement various operations listed in previous sections is shown below:

Example 8.18: Program to perform various operations

```
#include <stdio.h>
#include <stdlib.h>

struct node
{
        int             info;
        struct node     *link;
};

typedef struct node* NODE;

/* Include: Example 8.2: Function to get a new node from the operating system */
/* Include: Example 8.3: Function to insert an item at the front end of the list*/
/* Include: Example 8.4: Function to display the contents of the list */
/* Include: Example 8.11: Function to find the length of the linked list */
/* Include: Example 8.12: Function to search for key item in the list */
/* Include: Example 8.14: Function to delete a node whose info field is specified */
/* Include: Example 8.16: Function to reverse a list */

void main()
{
        NODE            first;
        int             choice, item, key;

        first  =  NULL;
        for (;;)
        {
                printf("1:Insert_Front    2:Length of list\n");
                printf("3:Serach          4:Delete item\n");
                printf("5:Reverse         6:Display\n");
                printf("7: Exit\n");
                printf("Enter the choice\n");
                scanf("%d", &choice);

                switch(choice)
                {
                        case 1:
                                printf("Enter the item to be inserted\n");
                                scanf("%d", &item);
                                first = insert_front (item, first);
                                break;
```

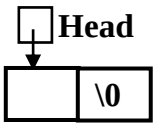
```
                    case 2:
                        printf("Length of list = %d\n", Length(first));
                        break;

                    case 3:
                        printf("Enter the item to be searched\n");
                        scanf("%d", &key);
                        search(key, first);
                        break;

                    case 4:
                        printf("Enter the item to be deleted\n");
                        scanf("%d", &key);
                        first = delete_info(key, first);
                        break;

                    case 5:
                        first = reverse(first);
                        break;

                    case 6:
                        display(first);
                        break;

                    default:
                        exit(0);
                }
        }
}
```

8.7 Header Node

Now, let us see "What is a header node?"

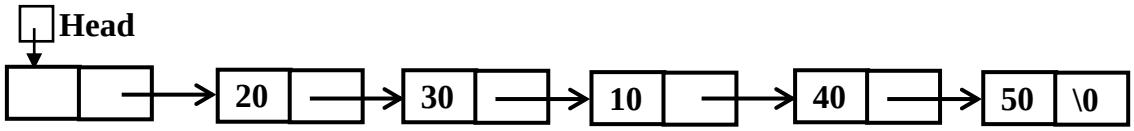
Definition: A header node in a linked list is a special node whose *link* field always contains address of the first node of the list. Using header node, any node in the list can be accessed. The *info* field of header node usually does not contain any information and such a node does not represent an item in the list. Sometimes, useful information such as, number of nodes in the list can be stored in the info field. If the list is empty, then *link* field of header node contains \0 (null).

Example 1: An empty list using header is represented as shown below:



If we display the above list, list is empty.

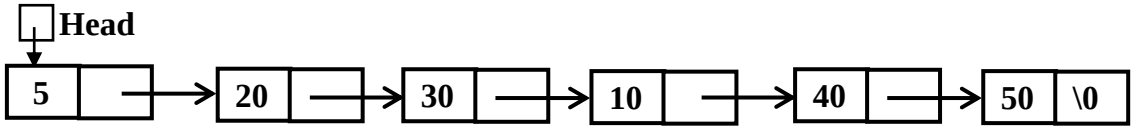
Example 2: The non-empty list using header node is represented as shown below:



In the above list

- The *info* field of header node does not contain any information.
- The first node starts from *link* field of header node i.e., the node containing the *info* 20 is the first node
- If we display the contents of above list, the output will be: 20, 30, 10, 40, 50

Example 3: The non-empty list using header node with valid information in header node is represented as shown below:



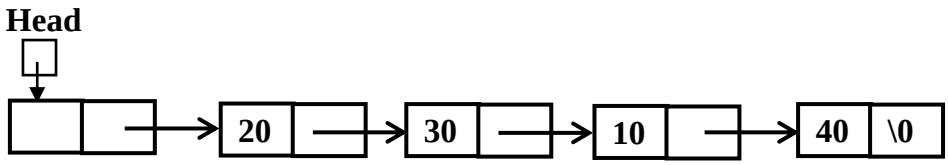
In the above list, the *info* field of the header node contains the number of nodes in the list.

Now, let us see "What are the advantages of a list with a header node?" The advantages of a header node are shown below:

- Simplifies Insertion and deletion operations
- Avoid the usage of various cases such as "if only one node is present what to do"
- Designing of program will be very simple
- Circular lists with header node are frequently used instead of ordinary linked lists because many operations can be easily implemented

8.7.1 Insert a node at the front end

Now, let us see "How to insert an item at the front end?" Consider the circular list with a header node shown in figure below.

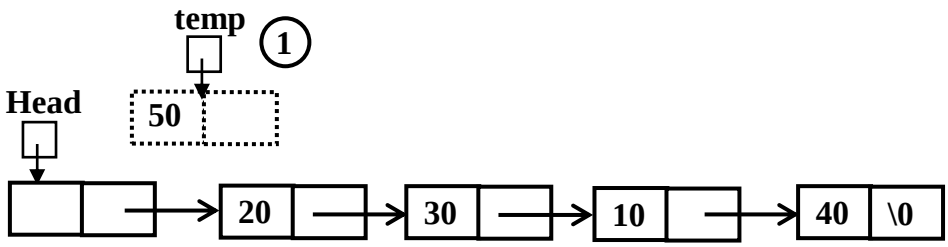


An *item* 50 can be inserted at the front end of the list using the following sequence of steps:

Step 1: Create a node: This can be done by using getnode() function and copying *item* into the *info* field as shown below:

```
(1)  temp = getnode()
     temp->info = item;
```

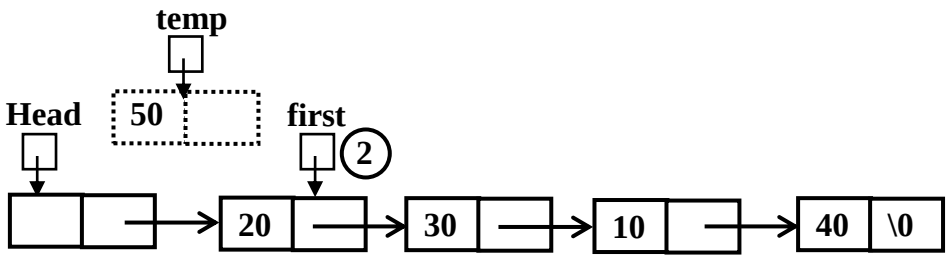
The list after executing above statements can be written as shown below:



Step 2: Obtain the address of the first node: The first node can be accessed using "head->link" and copy this into *first* as shown below:

```
(2)  first = head->link;
```

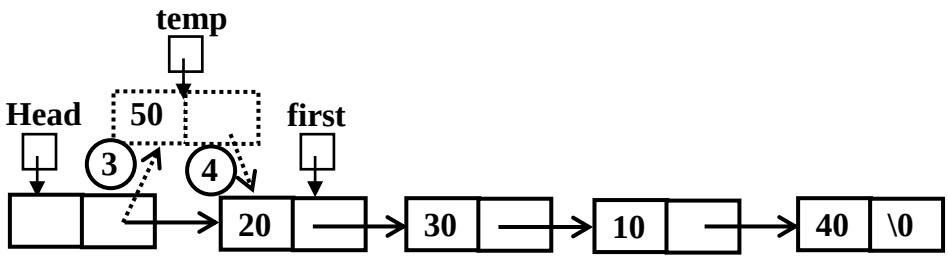
Now, the variable *first* contains address of the first node and the linked list can be written as shown below:



Step 3: Make the new node created as the first node: This can be done by inserting the node *temp* between *head* and *first*. This can be achieved using the following code:

```
(3) head->link = temp;
(4) temp->link = first;
```

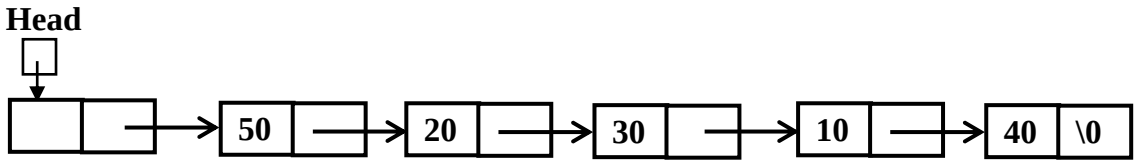
The list obtained after executing the above statements can be written as shown below:



Step 4: Finally return the address of the header node using the following statement:

```
return head;
```

The list as seen from the calling function can be written as shown below:



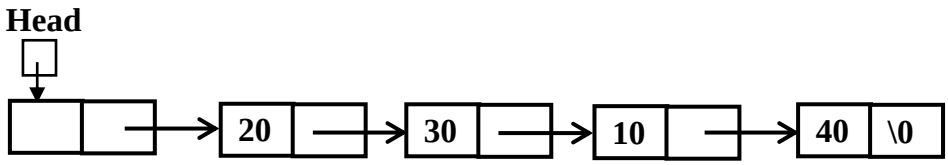
The complete C function is shown below:

Example 8.19: Function to insert at the front end of the list

```
NODE insert_front(int item, NODE head)
{
        NODE temp;
(1)     temp = getnode();                    /* Create a node, insert the item */
        temp->info = item;
(2)     first  = head->link;                 /* Obtain the address of the first node */
(3)     head->link = temp;                   /* Insert at the front end */
(4)     temp->link = first;
        return head;                         /* Return the header node */
}
```

8.7.2 Insert a node at the rear end

Now, let us see "How to insert an item at the rear end?" Consider the circular list with a header node shown in figure below.

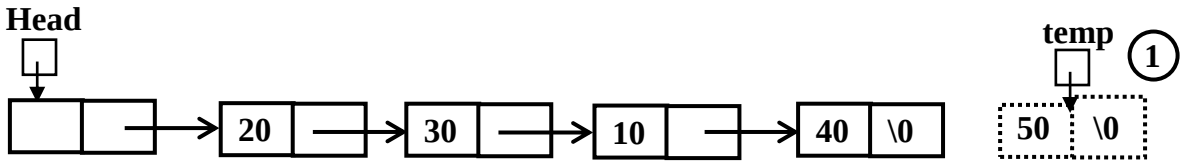


An *item* 50 can be inserted at the rear end of the list using the following sequence of steps:

Step 1: Create a node: This can be done by using getnode() function and copying *item* into the *info* field as shown below:

```
(1)  temp = getnode()
     temp->info = item;
     temp->link = NULL
```

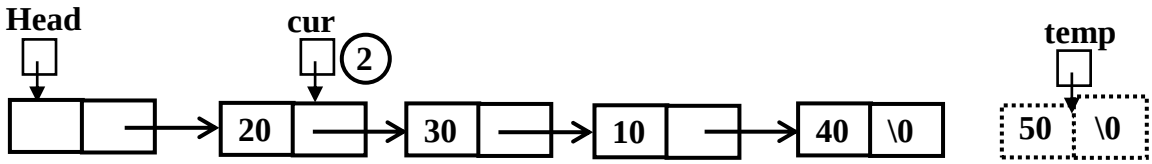
The list after executing above statements can be written as shown below:



Step 2: Obtain the address of the first node: The first node can be accessed using "head->link" and copy this into *cur* as shown below:

```
(2)  cur = head->link;
```

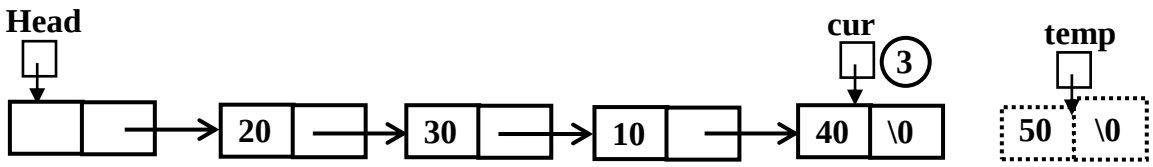
Now, the variable *cur* contains address of the first node and the linked list can be written as shown below:



Step 4: Obtain the address of the last node: This can be done by updating *cur* repeatedly to contain address of the next node till cur->link is not NULL. This can be achieved using the following code:

```
while (cur->link != NULL)
{
        cur = cur->link;
}
```
③

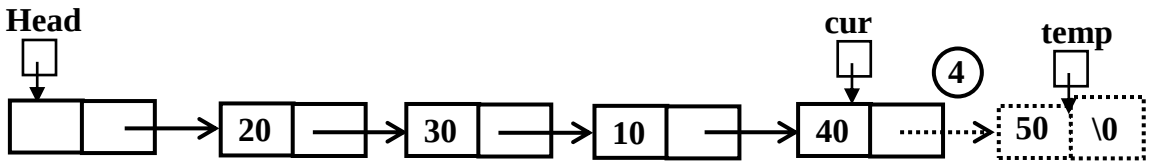
Once control comes out of the loop, the variable *cur* contains address of the last node. Now, the linked list obtained can be written as shown below:



Step 4: Make the new node created as the last node: This can be done by inserting the node *temp* after *cur*. This can be done by copying *temp* to *link* field of *cur* node as shown below:

④ cur->link = temp;

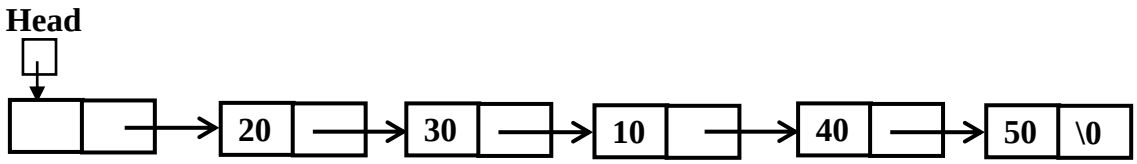
The list obtained after executing the above statements can be written as shown below:



Step 4: Finally return the address of the header node using the following statement:

return head;

The list as seen from the calling function can be written as shown below:



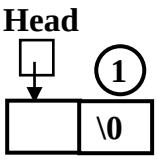
Now, the complete code to insert an element at the rear end of the list can be written as shown below:

Example 8.20: Function to insert at the front end of the list

```
NODE insert_rear(int item, NODE head)
{
        NODE temp, cur;

        temp = getnode();                   /* node to be inserted */
  (1)   temp->info = item;
        temp->link = NULL;

  (2)   cur = head->link;                   /* obtain address of the first node */

        while ( cur->link != NULL )         /* Obtain address of the last node */
        {
  (3)           cur = cur->link;
        }

  (4)   cur->link = temp;                   /* insert node at the end */

        return head;
}
```

8.7.3 Delete a node from the rear end

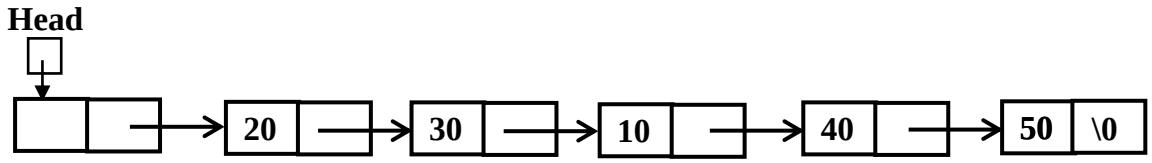
Step 1: Check for an empty list: An empty list is pictorially represented as shown below:



The code for the above case can be written as shown below:

```
        if (head->link == NULL)
        {
  (1)           printf("List is empty\n");
                return head;
        }
```

When control comes out of the above if-statement, it means list is already existing and it can be pictorially represented as shown below:

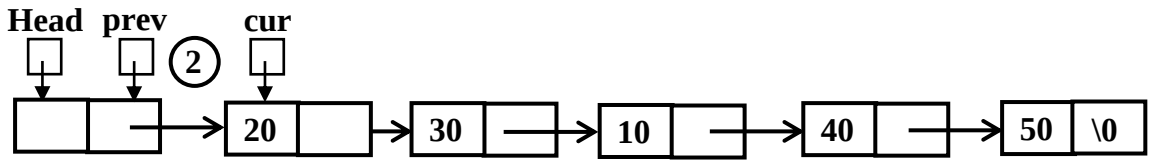


Step 2: Get the address of the first node and its previous:

The *link* field of node *head* is the address of the first node and *head* is the previous node. The first node and its predecessor can be obtained using the following statements:

```
(2)  prev = head;          // Previous to first node
     cur = head->link;     // Obtain the first node
```

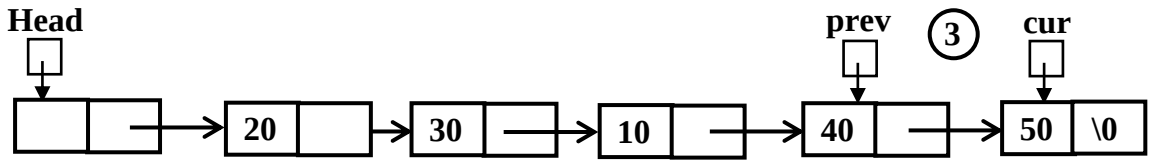
After executing the above statements, the linked list looks as shown below:



Step 3: Get the address of last node and its previous node: Keep updating *prev* and *cur* as long as *link* field of *cur* is not NULL. The code corresponding to this can be written as shown below:

```
while (cur->link != NULL)
{
(3)     prev = cur;
        cur = cur->link;
}
```

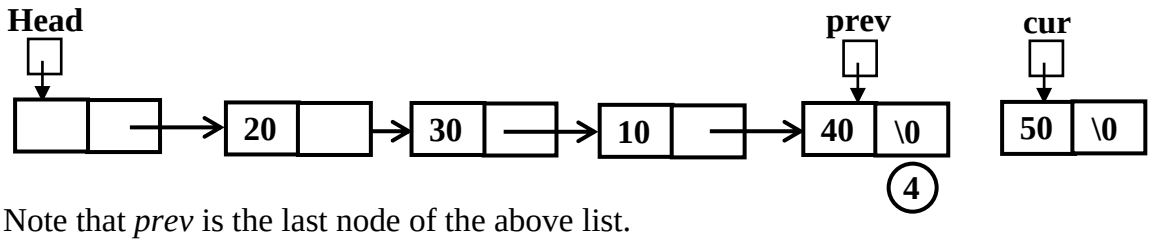
After executing the above part of the code, the linked list can be written as shown below:



Step 4: Make last but one node as the last node: This is achieved by copying NULL to *link* field of *prev* node. This can be done using the statement:

```
4  prev->link = NULL;
```

After executing the above statement, the last node *cur* is isolated and the linked list looks as shown below:

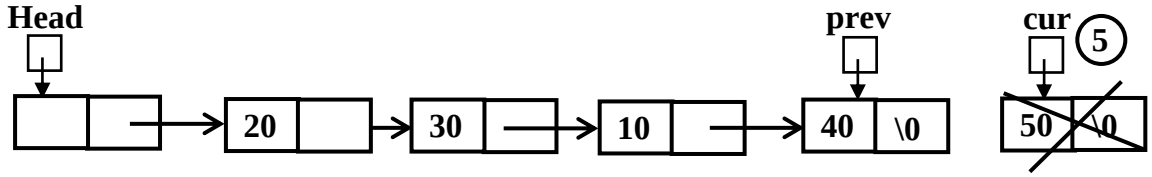


Note that *prev* is the last node of the above list.

Step 5: Delete the last node: First, display the *info* field of last node and then delete it. This can be done using the statement:

```
5  printf("Item deleted = %d\n", cur->info);   /* Item deleted = 50 */
   free (cur);
```

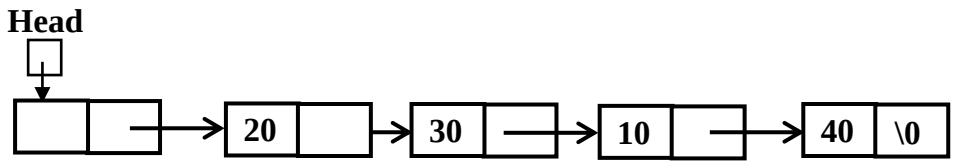
After executing the above statements, the linked list looks as shown below:



Step 6: Return the header node: This can be done using the statement:

```
return head;
```

The final list as seen from the calling function is shown below:



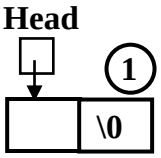
Thus, the last node of the list can be removed. The code to delete an element from the rear end can be written as shown below:

Example 8.21: Function to delete an element from the rear end

```
NODE delete_rear(NODE head)
{
        NODE prev, cur;

        if (head->link == NULL)
        {
(1)             printf("List is empty\n");
                return head;
        }

(2)     prev = head;                    // Previous to first node
        cur = head->link;               // Obtain the first node

        while (cur->link != NULL)  // Obtain address of last node and last but one */
        {
(3)             prev = cur;
                cur = cur->link;
        }

(4)     prev->link = NULL;                      // Make last but one node as the last node

(5)     printf("Item deleted = %d\n", cur->info);   /* Item deleted = 50 */
        free (cur);                             // Remove the last node

        return head;
}
```

8.7.4 Delete a node from the front end

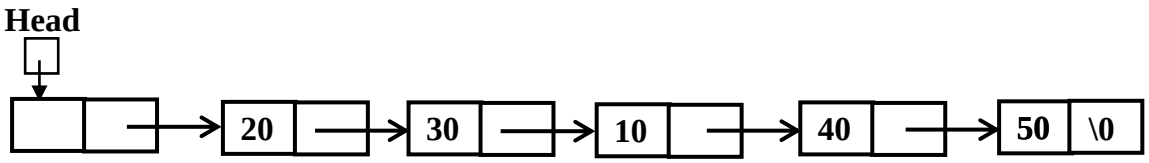
Step 1: Check for an empty list: An empty list is pictorially represented as shown below:



The code for the above case can be written as shown below:

```
        if (head->link == NULL)
        {
(1)             printf("List is empty\n");
                return head;
        }
```

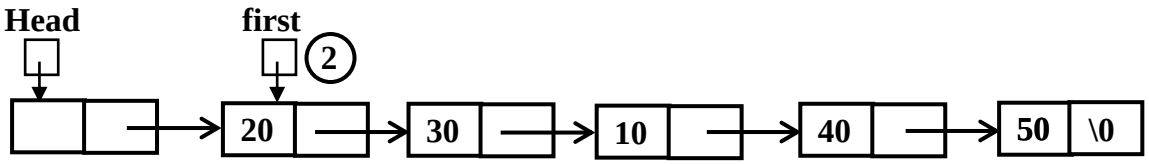
When control comes out of the above if-statement, it means list is already existing and it can be pictorially represented as shown below:



Step 2: Get the address of the first node: The *link* field of node *head* is the address of the first node and it can be obtained using the following statement:

```
(2) first = head->link;          // Obtain the first node
```

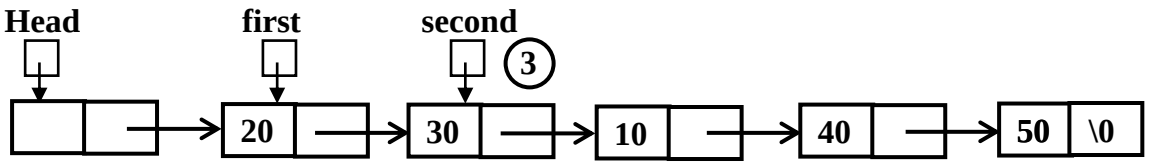
Now, the linked list looks as shown below:



Step 3: Get the address of the second node: The *link* field of node *first* is the address of the second node and it can be obtained using the following statement:

```
(3) second = first->link;        // Obtain the second node
```

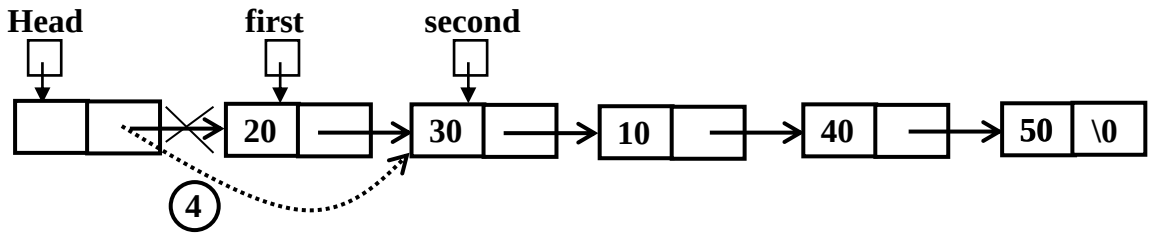
After executing the above instruction, the linked list looks as shown below:



Step 5: Make second node as the first node: This can be done by copying *second* into *link* field of *head* using the statement:

④ head->link = second; // Make 2nd node as the first node

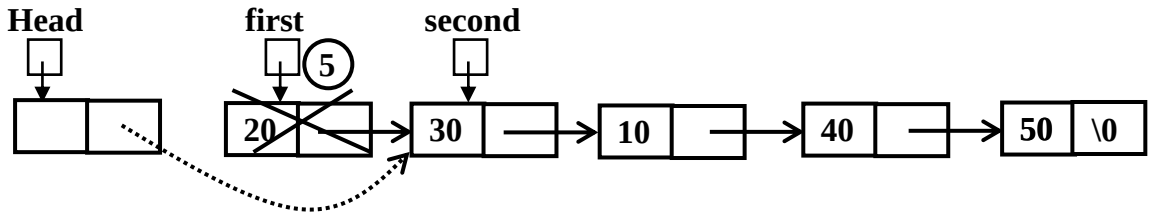
After executing the above instruction, the linked list looks as shown below:



Step 6: Remove the first node: The node *first* can be removed using the following code:

⑤ printf("Item deleted = %d\n", first->info); // Item deleted = 20
free(first);

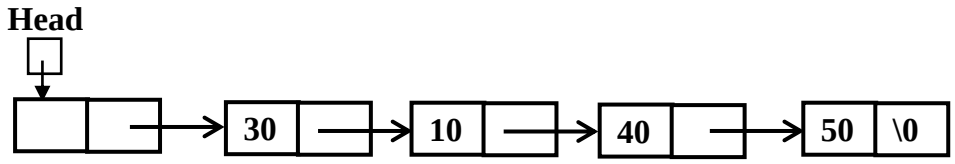
After executing the above instruction, the linked list looks as shown below:



Step 7: Return the header node: This can be done using the following code:

return head;

Now, the linked list as seen from the calling function is shown below:



The final function to delete an element from the front end can be written as shown below:

Example 8.22: Function to delete an element from the front end

```
NODE delete_front(NODE head)
{
        NODE first, second;

        if (head->link == NULL)
        {
(1)             printf("List is empty\n");
                return head;
        }

(2)     first = head->link;             // Obtain the first node

(3)     second = first->link;           // Obtain the second node

(4)     head->link = second;            // Make 2nd node as the first node

(5)     printf("Item deleted = %d\n", first->info);          // Item deleted = 20
        free(first);

        return head;
}
```

8.7.5 Implementation of double ended queue

The double ended queue can be implemented using singly linked list with a header node using five functions: insert_front(), insert_rear(), delete_front(), delete_rear() and display() functions. The C program to implement deque is shown below:

Example 8.23: Function to delete an element from the front end

```
#include <stdio.h>
#include <stdlib.h>

struct node
{
        int             info;
        struct node     *link;
};

typedef struct node* NODE;
```

```
/* Include: Example 8.2: Function to get a new node from the operating system */
/* Include: Example 8.19: Function to insert at the front end of the list */
/* Include: Example 8.20: Function to insert at the front end of the list */
/* Include: Example 8.21: Function to delete an element from the rear end */
/* Include: Example 8.22: Function to delete an element from the front end */
/* Include: Example 8.4: C function to display the contents of linked list */

void main()
{
        NODE        head;
        int         choice, item;

        head = getnode();
        head->link  =  NULL;
        for (;;)
        {
                printf("1:Insert_Front    2:Insert_Rear\n");
                printf("3:Delete_Front    4:Delete_Rear\n");
                printf("5:Display         6:Exit\n");
                printf("Enter the choice\n");
                scanf("%d", &choice);

                switch(choice)
                {
                        case 1:
                                printf("Enter the item to be inserted\n");
                                scanf("%d", &item);
                                head = insert_front (item, head);
                                break;

                        case 2:
                                printf("Enter the item to be inserted\n");
                                scanf("%d", &item);
                                head = insert_rear (item, head);
                                break;

                        case 3:
                                head = delete_front (item, head);
                                break;
```

```
                    case 4:
                            head = delete_rear (item, head);
                            break;
                    case 5:
                            display(head->link);
                            break;

                    default:
                            exit(0);
            }
        }
}
```

8.8 Representing student data using linked list

Now, let us "Design, Develop and Implement a menu driven Program in C for the following operations on **Singly Linked List (SLL)** of Student Data with the fields: ***USN, Name, Branch, Sem, PhNo***"

1. Create a **SLL** of **N** Students Data by using ***front insertion***.
2. Display the status of **SLL** and count the number of nodes in it
3. Perform Insertion and Deletion at End of **SLL**
4. Perform Insertion and Deletion at Front of **SLL**
5. Demonstrate how this **SLL** can be used as **STACK** and **QUEUE**
6. Exit

A structure to hold the student details can be written as shown below:

```
typedef struct
{
        int         usn;
        char        name[20];
        char        branch[20];
        int         sem;
        char        phone[20];
        struct
} STUDENT;
```

The design details are given in section 8.2. Creating a list is nothing but repeated insertion of items into the list. The insert front function (example 8.3) and insert rear function (example 8.6) are modified to incorporate the above details.

The C function to insert a node at the front end can be written as shown below:

Example 8.24: **C** Function to insert an item at the front end of the list

```
NODE insert_front(STUDENT item, NODE first)
{
        NODE temp;

        temp = getnode();                   /*obtain a node from OS */
        temp->usn = item.usn;               /* Insert various items into new node */
        strcpy(temp->name, item.name);
        strcpy(temp->branch, item.branch);
        temp->sem = item.sem;
        strcpy(temp->phone, item.phone);
        temp->link = NULL;

        if (first == NULL)  return temp;    /* Insert a node for the first time */

        temp->link = first;                 /* Insert at the beginning of existing list */

        return temp;                        /* return address of new first node */
}
```

The C function to insert a node at the rear end can be written as shown below:

Example 8.25: **C** Function to insert various items at the rear end of the list

```
NODE insert_rear(STUDENT item, NODE first)
{
        NODE temp, cur;

        temp = getnode();                   /*obtain a node from OS */
        temp->usn = item.usn;               /* Insert various items into new node */
        strcpy(temp->name, item.name);
        strcpy(temp->branch, item.branch);
        temp->sem = item.sem;
        strcpy(temp->phone, item.phone);
        temp->link = NULL;

        if (first == NULL)  return temp;    /* Insert a node for the first time */
```

```
        /* Get the address of the first node */
        cur = first;

        /* Find the address of the last node */
        while (cur->link != NULL)
        {
                cur = cur->link;
        }

        /* Insert the node at the end */
        cur->link = temp;

        /* return address of the first node */
        return first;
}
```

The function to delete an element from the front end given in example 8.5 can be modified as shown below:

Example 8.26: C function to delete an item from the front end of the list

```
NODE delete_front(NODE first)
{
        NODE temp;

        if ( first == NULL )            /* Check for empty list */
        {
                printf("student list is empty cannot delete\n");
                return NULL;                    // We can replace NULL with first also
        }

        temp = first;                   /* Retain address of the node to be deleted */

        temp = temp->link;              /* Obtain address of the second node */

        printf("Delete student record: USN = %d\n", first->usn);
        free(first);                    /* delete the front node */

        return temp;                    /* return address of the first node */
}
```

The function to delete an element from the rear end given in example 8.7 can be modified as shown below:

Example 8.27: Function to delete an item from the rear end of the list

```
NODE delete_rear(NODE first)
{
        NODE cur, prev;

        if (first == NULL)              /* Check for empty list */
        {
                printf("student list is empty cannot delete\n");
                return first;
        }

        if ( first->link == NULL )      /*Only one node is present and delete it */
        {
                printf("Delete student record: USN = %d\n", first->usn);

                free(first);            /* return to availability list */

                return NULL;            /* List is empty so return NULL */
        }

        /* Obtain address of the last node and just previous to that */
        prev = NULL;
        cur = first;
        while( cur->link != NULL )
        {
                prev = cur;
                cur = cur->link;
        }

        printf("Delete student record: USN = %d\n", first->usn);
        free(cur);                      /* delete the last node */

        prev->link = NULL;              /* Make last but one node as the last node */

        return first;                   /* return address of the first node */
}
```

The function to display the student details are shown below:

Example 8.28: Function to display student details

```
void display(NODE first)
{
        NODE cur;    int count = 0;

        if (first == NULL)                    /* List is empty */
        {
                printf("student list is empty\n");
                return;
        }

        cur = first;              /* Display employee details */
        while (cur != NULL)
        {
                printf("%d %s %s %d %s \n", cur->usn, cur->name, cur->branch,
                                                        cur->sem, cur->phone);
                cur = cur->link;   count++;
        }
        printf("Number of students = %d\n", count);
}
```

The program to implement dequeuer operations can be written as shown below:

Example 8.27: Program to implement dequeues using singly linked list (student)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct
{
        int             usn;
        char            name[20];
        char            branch[20];
        int             sem;
        char            phone[20];
} STUDENT;
```

```
struct node
{
        int             usn;
        char            name[20];
        char            branch[20];
        int             sem;
        char            phone[20];
        struct node     *link;
};

typedef struct node* NODE;

/* Include: Example 8.2: Function to get a new node from the operating system */

/* Include: Example 8.24: Function to insert an item at the front end of the list */

/* Include: Example 8.25: Function to insert an item at the rear end of the list */

/* Include: Example 8.26: Function to delete an item from the front end of the list */

/* Include: Example 8.27: Function to delete an item from the rear end of the list */

/* Include: Example 8.28: Function to display the contents of linked list */


void main()
{
        NODE        first;
        int         choice;
        STUDENT     item;

        first  =  NULL;

        for (;;)
        {
                printf("1:Insert_Front    2: Insert_Rear\n");
                printf("3:Delete_Front    4: Delete_Rear\n");
                printf("5:Display         6: Exit\n");

                printf("Enter the choice\n");
                scanf("%d", &choice);
```

```
            switch(choice)
            {
                case 1:
                    printf("usn          :"); scanf("%d",&item.usn);
                    printf("name         :"); scanf("%s",item.name);
                    printf("branch       :"); scanf("%s",item.branch);
                    printf("semester     :"); scanf("%d",&item.sem);
                    printf("phone        :"); scanf("%s",item.phone);
                    first = insert_front (item, first);
                    break;
                case 2:
                    printf("usn          :"); scanf("%d",&item.usn);
                    printf("name         :"); scanf("%s",item.name);
                    printf("branch       :"); scanf("%s",item.branch);
                    printf("semester     :"); scanf("%d",&item.sem);
                    printf("phone        :"); scanf("%s",item.phone);
                    first = insert_rear (item, first);
                    break;
                case 3:
                    first = delete_front(first);
                    break;
                case 4:
                    first = delete_rear(first);
                    break;
                case 5:
                    display(first);
                    break;
                default:
                    exit(0);
            }
        }
}
```

8.9 Other operations on linked list

In this section, let us see how to implement the following operations on linked lists:

- Remove duplicate elements in the list
- Union and intersection operation on two lists
- Multiple stacks
- Multiple queues

The first two operations require that an *item* has to be searched in the list. If item is present in the list, we return 1, otherwise, we return 0. The code already designed in section 8.6.2 (i.e., example 8.12) can be modified only in returning the values using the **return** statement. The modified code can be written as shown below:

Example 8.28: Function to search for key item in the list

```
int search(int key, NODE first)
{
        NODE cur;

        if ( first == NULL )   return 0;              /* Search unsuccessful */

        cur = first;
        while (cur != NULL)                           /* As long as no end of list */
        {
                if (key == cur->info) return 1;       /* Search successful */
                cur = cur->link;                      /* point cur to the next node */
        }

        return 0;                                     /* Search not successful */
}
```

Now, let us see, how the above function is helpful in designing the solution for next three operations discussed.

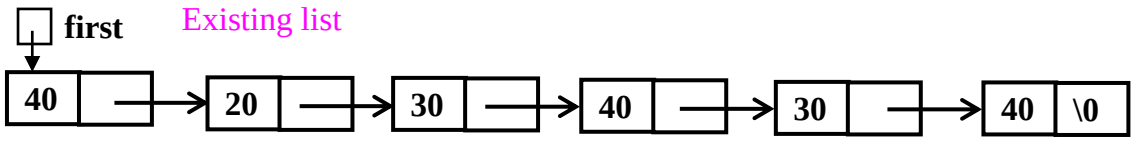
8.9.1 Remove duplicate elements

In this section, let us remove duplicate elements in the given linked list. This can be done as shown below:

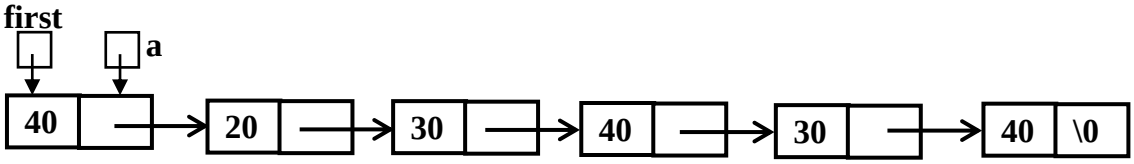
Step 1: Empty list: If list is empty return NULL indicating the resulting list is also empty. The code for this case can be written as shown below:

```
if (first == NULL) return NULL;
```

Step 2: List is existing: If the list is existing then duplicate elements have to be removed. This can be explained by considering the following list:



Instead of using the variable *first*, let us use another variable *a* which points to the beginning of the list as shown below:



The above situation can be achieved by executing the following statement:

```
a = first;
```

Each item in the above list has to be accessed and a new list without duplicates has to be created. Each item in the list can be accessed using the following code:

```
a = first;
while (a != NULL)
{
        a->info;            // access each item in the list
        a = a->link;        // get the address of the next node
}
```

Now, as you access using *a->info*, search for *a->info* in the second list *b*. If not found, insert *a->info* at the end of the list *b*. Now, the above partial code can be written as shown below:

```
a = first;
while (a != NULL)
{
        /* search for a->info in list b */
        found = search(a->info, b) ;

        /* If a->info not found in b, insert a->info at the end of list b */
        if (found == 0) b = insert_rear(a->info, b);

        a = a->link;
}
```

Step 3: Return final list: This can be done by returning *b* which contains address of the list. This can be done using the statement:

```
return b;
```

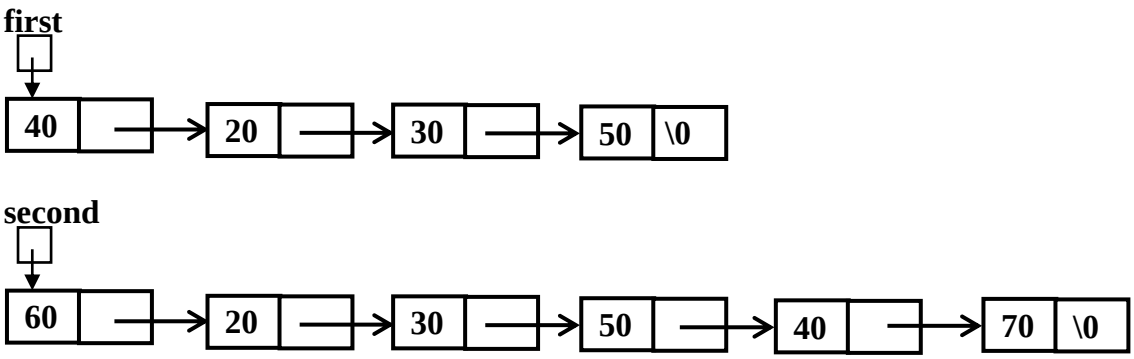
Now, the final code can be written as shown below:

Example 8.29: Function to remove duplicate elements from the list

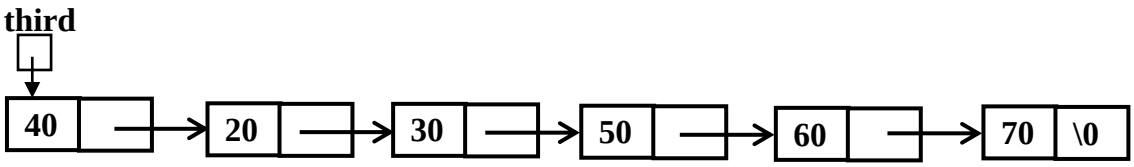
```
NODE remove_duplicate (NODE first)
{
        NODE        a, b;
        int         flag;

        if (first == NULL)   return NULL;        /* empty list */

        b = NULL;                                /* existing new list b*/

        a = first;                               /* existing list a */

        while (a != NULL)                        /* Traverse the entire list a */
        {
                flag = search(a->info, b);       /* search for item in new list b */

                /* if not found insert into new list b*/
                if (flag == 0) b = insert_rear(a->info, b);

                a = a->link;                     /* access next item in the list a */
        }

        return b;                                /* new list without duplicates */
}
```

8.9.2 Union of two lists

Now, let us see how to find union of two lists. It can be explained by looking into the following two lists:

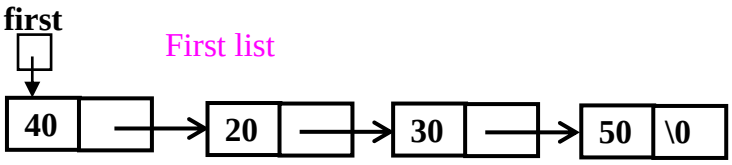


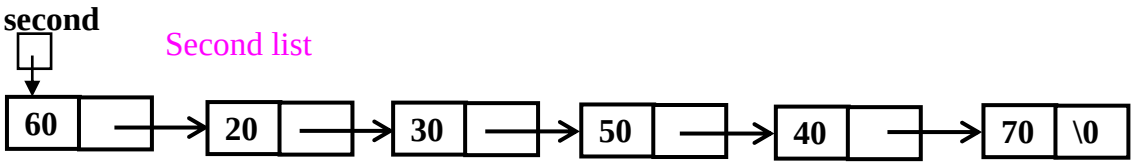
The union of above two lists can be written as shown below:

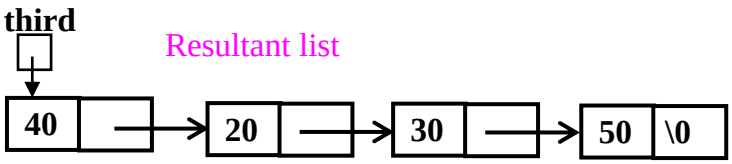


The above activity can be achieved using the following three steps:

Step 1: Add all elements of first list to the resultant list: This can be done by accessing each item from first list and adding at the end of the resultant list identified by variable *third*. The pictorial representation of three lists now can be written as shown below:



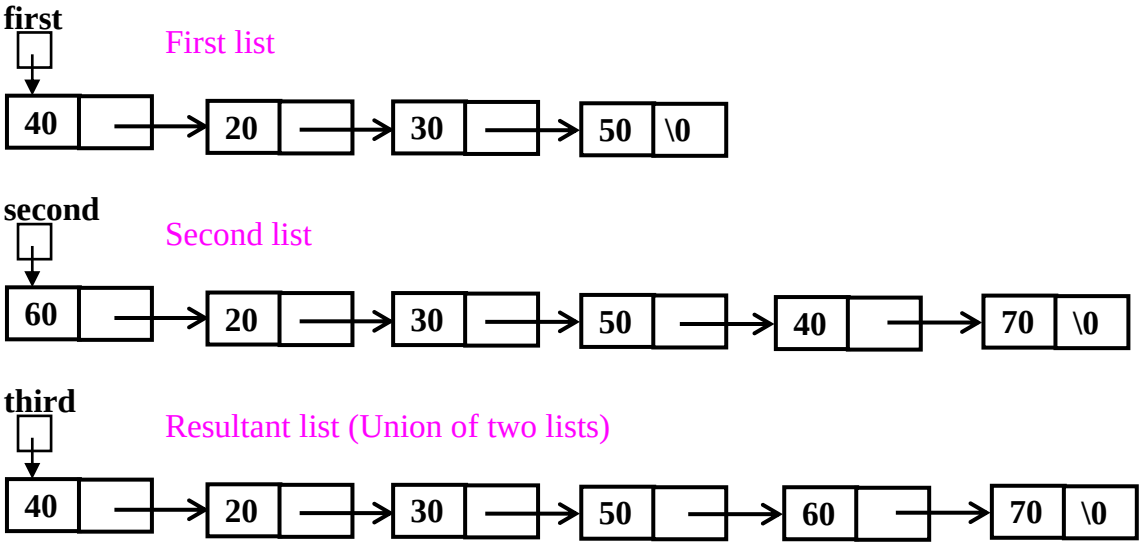




The code for this activity can be written as shown below:

```
a = first;                                  // access first list using variable a

/* Add all elements of first list to the end of resultant list */
third = NULL;                               // Initial resultant list
while (a != NULL)                           // Traverse till the end
{
        third = insert_rear (a->info, third);  // Insert item at the rear end of list
        a = a->link;                        // Move to the next node in the list
}
```

Step 2: Insert elements of second list into resultant list: This can be done by accessing each item from the first list and search in the resultant list. If *item* of first list is not present in the resultant list then add the *item* of the first list to the end of the resulting list. The above activity is pictorially represented as shown below:



This can be done using the statements shown in dotted lines in example 8.29 in previous section. But, replace the variable *first* by *second*. The code is re-written as shown below:

```
a = second;                          /* existing list */

while (a != NULL)                    /* Traverse the entire second list */
{
        flag = search(a->info, third);       /* search for item in new list b */

        /* if not found insert into new list b*/
        if (flag == 0) third = insert_rear(a->info, third);

        a = a->link;                 /* access next item in the list a */
}
```

Step 3: Return the resultant list: This can be done using the statement:

```
return third;
```

Now, the final code can be written as shown below:

Example 8.30: Function to find union of two lists

```
NODE union_of_list (NODE first, NODE second)
{
        NODE    a, third;
        int     flag;

        a = first;                                  // access first list using variable a

        /* Add all elements of first list to the end of resultant list */
        third = NULL;                               // Initial resultant list
        while (a != NULL)                           // Traverse till the end
        {
                third = insert_rear (a->info, third);  // Insert item at the rear end of list

                a = a->link;                        // Move to the next node in the list
        }

        /* search for each item of 2nd list in 3rd list. If not found add into 3rd list */
        a = second;                                 /* existing list */
        while (a != NULL)                           /* Traverse entire second list */
        {
                flag = search(a->info, third);      /* search for item in new list b */

                /* if not found insert into resultant list*/
                if (flag == 0) third = insert_rear(a->info, third);

                a = a->link;                        /* access next item in the list a */
        }

        return third;                               /* return the resultant result */
}
```

8.9.3 Intersection of two lists

The intersection of two linked lists is much easier. To start with initialize resultant list to NULL. This can be done as shown below:

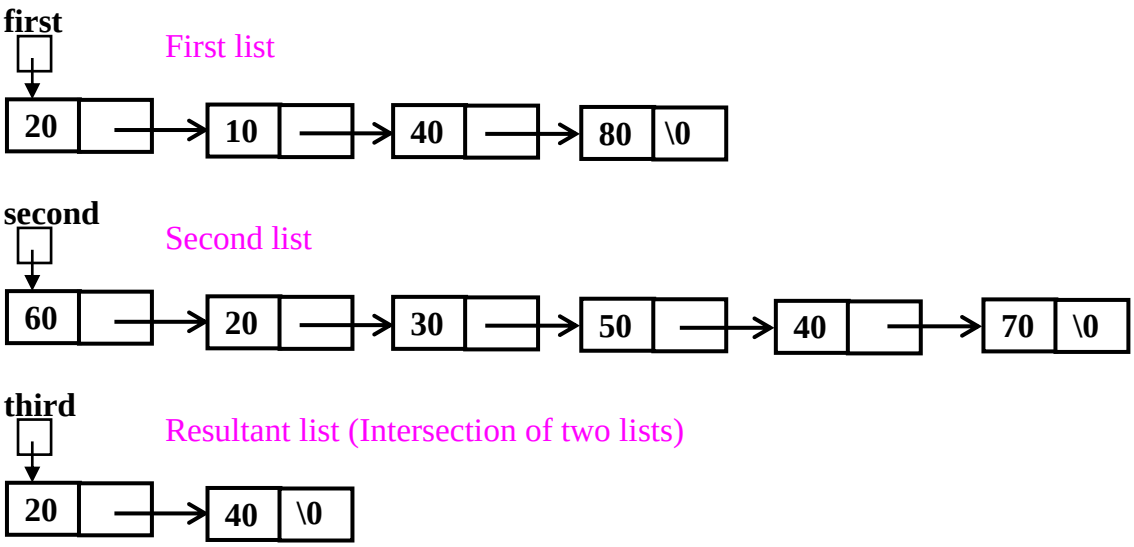
```
third = NULL;
```

Traverse the first list till the end. During this process search for each *item* of the first list in the second list. If the element is present then add the element to the resultant

list. This can be done by changing the if-statement inside the loop given in example 8.29 shown using dotted lines (section 8.9.1). In the if-statement check for *flag* to 1. The code is re-written with modification as shown below:

```
a = first;                          /* existing list a */
b = second;                         /* existing list b */
third = NULL;                       /* resultant list is empty */

while (a != NULL)                   /* Traverse the entire list a */
{
        flag = search(a->info, b);  /* search for item in new list b */

        /* if found insert into resultant list*/
        if (flag == 1) third = insert_rear(a->info, third);

        a = a->link;                /* access next item in the list a */
}
```

Now, given the two lists we can get the third list which is intersection of first two lists using the above code as shown below:



Now, the complete code can be written as shown below:

Example 8.31: Function to find intersection of two lists

```
NODE intersection_of_list (NODE first, NODE second)
{
        NODE    a, b, third;
        int     flag;

        a = first;                              /* a points to first list */
        b = second;                             /* b points to second list */
        third = NULL;                           /* resultant list is empty */

        while (a != NULL)                       /* Traverse entire second list */
        {
                flag = search(a->info, b);      /* search for item in new list b */
                /* if not found insert into new list b*/
                if (flag == 1) third = insert_rear(a->info, third);

                a = a->link;                    /* access next item in the list a */
        }

        return third;                           /* return the resultant result */
}
```

Now, the functions to remove duplicate elements, to find union and intersection of two lists can be invoked using function *main* which can be written as shown below:

Example 8.32: Program to remove duplicate items, find union and intersection of 2 lists

```
#include <stdio.h>
#include <stdlib.h>

struct node
{
        int             info;
        struct node     *link;
};

typedef struct node* NODE;

/* Include: Example 8.2: Function to get a new node from the operating system */
/* Include: Example 8.4: Function to display the contents of the list */
/* Include: Example 8.6: Function to insert an item at the rear end of the list*/
/* Include: Example 8.28: Function to search for key in the list*/
```

```
/* Include: Example 8.29: Function to remove duplicates in the list*/
/* Include: Example 8.30: Function to find union of two lists */
/* Include: Example 8.31: Function to find intersection of two lists */

void main()
{
        NODE        first, second, third;
        int         choice, item, i, n;

        for (;;)
        {
                printf("1:Create first list                2:Create second list\n");
                printf("3:Remov duplicates of list 1    4:Remov duplicates of list 2\n");
                printf("5:Union of two lists               6:Intersection of two lists\n");
                printf("7:Exit\n");

                printf("Enter the choice\n");
                scanf("%d", &choice);
                switch(choice)
                {
                        case 1:
                                printf("Enter the number of nodes in first list\n");
                                scanf("%d", &n);

                                first = NULL;

                                for (i = 1; i <= n; i++)
                                {
                                        printf("Enter the item\n");
                                        scanf("%d", &item);

                                        first = insert_rear (item, first);
                                }

                                break;

                        case 2:
                                printf("Enter the number of nodes in second list\n");
                                scanf("%d", &n);

                                second = NULL;
```

```
            for (i = 1; i <= n; i++)
            {
                  printf("Enter the item\n");
                  scanf("%d", &item);

                  second = insert_rear (item, second);
            }
            break;

      case 3:
            printf("The first list before removing duplicates\n");
            display(first);

            first = remove_duplicate (first);

            printf("The first list after removing duplicates\n");
            display(first);

            break;
      case 4:
            printf("The second  list before removing duplicates\n");
            display(second);

            second = remove_duplicate (second);

            printf("The second list after removing duplicates\n");
            display(second);

            break;
      case 5:
            printf("The first  list \n");
            display(first);

            printf("The second  list \n");
            display(second);

            third = union_of_list (first, second);

            printf("The union of two lists \n");
            display(third);

            break;
      case 6:
            printf("The first  list \n");
            display(first);
```

```
                    printf("The second  list \n");
                    display(second);

                    third = intersection_of_list (first, second);

                    printf("The intersection of two lists \n");
                    display(third);

                    break;
                default:
                    exit(0);
            }
        }
}
```

8.9.4 Implementation of multiple queues

Now, let us "Implement multiple queues using array of queues with the help of singly linked lists"

Design: Multiple queues can be implemented using array of linked lists as shown below:

- Let us fix the number of queues to 5. This can be done using the statement:

```
#define     MAX_QUES    5
```

- A queue can be implemented using linked list as shown in previous section. The structure definition for a node can be written as:

```
struct node
{
    int             info;
    struct node     *link;
};

typedef struct node * NODE;
```

- To implement number of queues specified using symbolic constant MAX_QUES, we use the following declarations:

```
NODE        a[MAX_QUES];
```

- Initially, all queues are empty. This can be done by assigning NULL to each of the queue stored in the array as shown below:

```
for (i = 0; i < MAX_QUES; i++)
{
        a[i] = NULL;
}
```

The pictorial representation of initial multiple queues are shown below:

a[0]	\0	0th queue is empty
a[1]	\0	1st queue is empty
a[2]	\0	2nd queue is empty
a[3]	\0	3rd queue is empty
a[4]	\0	4th queue is empty.

To insert an *item* into ith queue: Invoke the function insert_rear() (example 8.6) using *item* and *a[i]* as shown below:

```
a[i] = insert_front(item, a[i]);
```

To delete an *item* from ith queue: Invoke the function delete_front() (example 8.5) using the following statement:

```
a[i] = delete_front( a[i] );
```

To display contents of ith queue: Invoke the function display() (example 8.4) using the following statement:

```
display(a[i]);
```

Now, the complete algorithm in C can be written as shown below:

Example 8.33: Program to implement multiple queues using singly linked lists

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_QUES 5

struct node
{
      int               info;
      struct node       *link;
};

typedef struct node* NODE;

/* Include: Example 8.2: Function to get a new node from the operating system */
/* Include: Example 8.6: Function to insert an item at the rear end of the list */
/* Include: Example 8.5: Function to delete an item from the front end of the list */
/* Include: Example 8.4: Function to display the contents of the list */

void main()
{
        NODE        a[MAX_QUES];
        int         choice, item, i;

        for (i = 0; i < MAX_QUES; i++)    // Initialize multiple queues to NULL
        {
                a[i] = NULL;
        }

        for (;;)
        {
                printf("Enter queue number: 0, 1, 2, 3, 4 : ");
                scanf("%d", &i);

                printf("1:Insert rear    2:Delete front \n");
                printf("3:Display        4:Exit\n");

                printf("Enter the choice\n");
                scanf("%d",&choice);
```

```
                switch ( choice )
                {
                        case 1:
                                printf("Enter the item to be inserted\n");
                                scanf("%d",&item);
                                a[i] = insert_rear(item, a[i]);
                                break;
                        case 2:
                                a[i] = delete_front(a[i]);
                                break;
                        case 3:
                                display(a[i]);
                                break;
                        default:
                                exit(0);
                }
        }
}
```

8.9.5 Implementation of multiple stacks

Now, let us "Implement multiple stacks using array of singly linked lists"

Design: Multiple stacks can be implemented using array of linked lists as shown below:

- Let us fix the number of stacks to 5. This can be done using the statement:

```
#define         MAX_STACKS    5
```

- A stack can be implemented using linked list as shown in previous section. The structure definition for a node can be written as:

```
struct node
{
        int             info;
        struct node     *link;
};

typedef struct node * NODE;
```

- To implement number of stacks specified using symbolic constant MAX_STACKS, we use the following declarations:

```
NODE        a[MAX_STACKS];
```

- Initially, all stacks are empty. This can be done by assigning NULL to each of the stack stored in the array as shown below:

```
for (i = 0; i < MAX_STACKS; i++)
{
        a[i] = NULL;
}
```

The pictorial representation of initial multiple stacks are shown below:

a[0]	\0	0^{th} stack is empty
a[1]	\0	1^{st} stack is empty
a[2]	\0	2^{nd} stack is empty
a[3]	\0	3^{rd} stack is empty
a[4]	\0	4^{th} stack is empty.

To insert an *item* into i^{th} stack: Invoke the function insert_front() (example 8.3) using *item* and *a[i]* as shown below:

```
a[i] = insert_rear(item, a[i]);
```

To delete an *item* from i^{th} stack: Invoke the function delete_front() (example 8.5) using the following statement:

```
a[i] = delete_front( a[i] );
```

To display contents of i^{th} stack: Invoke the function display() (example 8.4) using the following statement:

```
display(a[i]);
```

Now, the complete algorithm in C can be written as shown below:

Example 8.34: Program for multiple stacks using array of singly linked lists

```
#include <stdio.h>
#include <stdlib.h>

struct node
{
        int             info;
        struct node     *link;
};

typedef struct node* NODE;

#define MAX_STACKS  5

/* Include: Example 8.2: Function to get a new node from the operating system */
/* Include: Example 8.3: Function to insert an item at the front end of the list*/
/* Include: Example 8.5: Function to delete an item from the front end of the list */
/* Include: Example 8.4: Function to display the contents of the list */

void main()
{
        NODE        a[MAX_STACKS];
        int         choice, item, i;

        for (i = 0; i < MAX_STACKS; i++) // Initialize multiple stacks to NULL
        {
                a[i] = NULL;
        }

        for (;;)
        {
                printf("Enter stack  number: 0, 1, 2, 3, 4 : ");
                scanf("%d", &i);

                printf("1:Push          2:Pop \n");
                printf("3:Display       4:Exit\n");

                printf("Enter the choice\n");
                scanf("%d",&choice);
```

```
                switch ( choice )
                {
                        case 1:
                                printf("Enter the item to be inserted\n");
                                scanf("%d",&item);
                                a[i] = insert_front(item, a[i]);
                                break;

                        case 2:
                                a[i] = delete_front(a[i]);
                                break;


                        case 3:
                                display(a[i]);
                                break;

                        default:
                                exit(0);
                }
        }
}
```

8.10 Memory allocation and Garbage collection

Now, let us see "What is memory allocation?"

Definition: The process of reserving the memory space to store the data is called memory allocation. The memory space is allocated for variables in the following situations:

- For all global variables and static variables, memory is allocated during program startup. *The BSS segment in the memory contains all global variables and static variables* that are initialized to zero or do not have explicit initialization in source code. When the program is terminated, the memory is freed and returned to the free space available.
- For all local variables, memory is allocated as and when the control enters into the function. *The memory space for all local variables is reserved on the stack area in memory.* Before control goes out of the function, the memory reserved for all local variables is freed and returned to the free space available.
- Memory can be allocated during run time (execution time) dynamically using memory allocation functions such as malloc(), calloc() and realloc() in C

language. *The memory space for all dynamically variables is reserved in the heap area.* Once, memory space is used and no longer required, we can free the memory space using free() function in C language.

When the memory is allocated dynamically and when it is not freed when not required, this memory space cannot be used by any program. Thus memory space is wasted. If too much memory space is wasted, soon the system may crash. So, it is necessary to identify the unused space and return this unused space to the operating system so that it can be allocated other programs which require memory space to store the data. For this reason, we require garbage collector. Now, let us see "What is garbage collector?"

Definition: Garbage collection is form of memory management technique which attempts to reclaim the unused memory space (called garbage) and return it to operating system so that the memory space thus collected can be assigned to other programs for storing the data. The garbage collections avoids the need for the programmer to deallocate the memory space.

Garbage collection can be using a method called *reference counting.* In this technique, each allocated space is given a *counter*. This *counter* always contains the number of pointers associated with the allocated space. It is automatically incremented whenever a new variable references this allocated space. It is decremented whenever a pointer to the allocated space is de-referenced. When the counter becomes zero, the allocated memory space is collected and returned to the operating system.

EXERCISES

1. Write C functions to perform the following operations on singly linked list
   a) Insert front b)Insert rear c) display
   d) Delete front e) Delete rear
2. Write a C program to implement stack operations using singly linked lists.
3. Write a C program to implement queue operations using singly linked lists.
4. Write a C program to implement dequeuer operations using singly linked lists.
5. Write C functions to perform the following operations on singly linked list with a header node
   b) Insert front b)Insert rear c) display
   d) Delete front e) Delete rear
6. Write a C program to implement stack operations using singly linked lists with a header node
7. Write a C program to implement queue operations using singly linked lists with a header node
8. Write a C program to implement dequeuer operations using singly linked lists with a header node.
9. Write a C function which will perform an insertion to the immediate left of the k^{th} node in the singly linked list.
10. Write a C function to count the number of nodes in a singly linked list.
11. Write a C function to change the info field of the k^{th} node to the value given by X.
12. Write a C function to concatenate two lists into a single list.
13. Write a C function to reverse a given singly linked list without creating new nodes.
14. Write a C function search(p,x) that accepts a pointer p to a list of integers and an integer x and returns a pointer to a node containing x, if it exists, and the NULL pointer otherwise.
15. Write a C function srchinsrt(p,x) that adds an item x to the list pointed to by p, provided x is not in the list. Insertion can be done at the front end or rear end.
16. Write a C function to delete a node from the front end and insert it at the end of the list.
17. Write a C function to check whether a given string is a palindrome using a doubly linked list.
18. Mention few advantages of a header node.
19. What are the advantages and disadvantages of dynamic data structures?
20. Compare and contrast static data structures with dynamic data structures.
21. Explain the advantages and disadvantages of representing a groups of items as an array versus a linear linked list.

8.90 💻 Linked Lists

22. What are the shortcomings of a singly linked list? How these are eliminated in circular list?
23. Write a C function to find the length of the linked list
24. Write a C function to search for key item in the list
25. Write a C function to delete a node whose info field is specified
26. Write a C function to create an ordered linked list?
27. What is a header node? What are the advantages of a list with a header node?”
28. How to implement dequeue operations using C functions.
29. Write a C function to remove duplicate elements from the list
30. Write a C function to find union of two lists
31. Write a C function to find intersection of two lists
32. Write a C function to implement multiple queues using array of queues with the help of singly linked lists
33. Write a C function to implement multiple stacks using array of singly linked lists
34. What is garbage collection? How it is done?

Chapter 9: Circular and doubly linked lists

In a singly linked list that we have discussed earlier, note that *link* field of last node contains \0 (null) indicating there are no more nodes from this point onwards.

Now, let us see "What are the disadvantages of singly linked lists?" The various disadvantages of singly linked lists are listed below:

- In a singly linked list, there is only one *link* field and hence traversing (visiting each node) is done only in one direction. So, given the address of a node *x*, only those nodes which follow *x* are reachable but, the nodes that precede *x* are not reachable.

  Note: Since traversing is only in one direction, singly linked lists are also called **one way lists**.

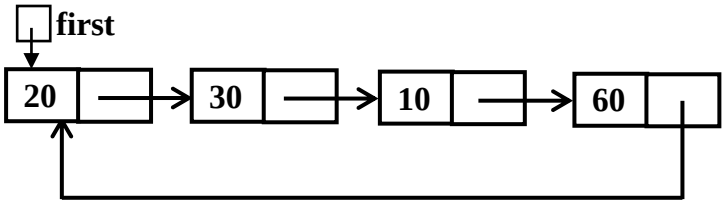
- To delete a designated node *cur*, address of the first node of the list should be provided. This is necessary because, to delete node *cur*, the predecessor of this node has to be found. For this, search has to begin from the first node of the list.

These disadvantages can be overcome using special type of list called **circular list.**

9.1 Circular list

Now, let us see "What is circular list?"

Definition: A circular singly linked list is a singly linked list where the link field of last node of the list contains address of the first node. The pictorial representation of a circular list is shown below:



Now, let us see "What are the advantages of circular lists?" The various advantages of circular list are shown below:

- Every node is accessible from a given node by traversing successively using the *link* field
- To delete a node *cur***,** the address of the first node is not necessary. Search for the predecessor of node *cur***,** can be initiated from *cur* itself.

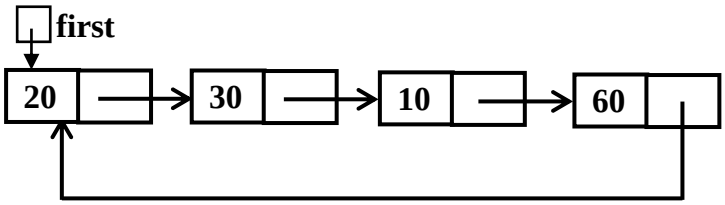
9.2 💻 Circular and doubly linked Lists

- Certain operations on circular lists such as concatenation and splitting of lists etc. will be more efficient.

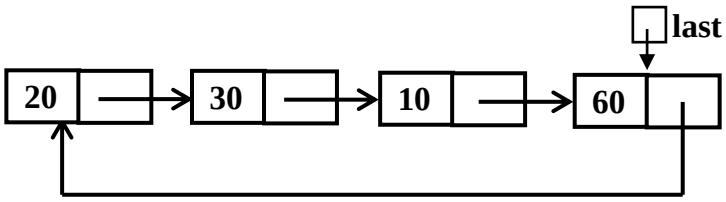
Note: Since *link* field of last node contains address of first node, there is no special indication to find which is the last node. So, some care has to be taken during processing. In circular list, it is very important to detect the end of the list.

Since the list is circular, any node can be considered as first node and its predecessor is considered as last node. The following two conventions can be used:

- **Approach 1:** A pointer variable *first* can be used to designate the starting point of the list. Using this approach, to get the address of the last node, the entire list has to be traversed from the first node. The pictorial representation of the circular list using this approach is shown below:



- **Approach 2:** In the second technique, a pointer variable *last* can be used to designate the last node and the node that follow *last*, can be designated as the first node of the list. The pictorial representation of the circular list using this approach is shown below:



Note: Observe from the above figure that, the variable *last* contains address of the last node. Using *link* field of last node, i.e, **last->link,** we can get the address of the first node.

In all our solutions to various problems, let us use the second approach. Using this approach, now, let us see "How to create an empty list?" An empty list is can be created by assigning NULL to a self-referential structure variable.

For example, consider the following code:

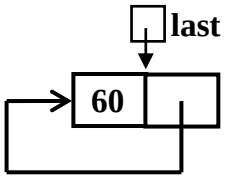
```
struct node
{
        int             info;
        struct node     *link
} ;

typedef struct node *NODE;

NODE last;                  // last is self-referential structure variable.

last = NULL;                // empty list by name last is created
```

An empty list identified by the variable *last* is pictorially represented as shown below:

Now, the question is "When we say list is empty?" Whenever *last* is NULL we say list is empty. The corresponding code can be written as shown below:

```
if (last == NULL)
{
        printf("List is empty\n");
        return;
}
```

A list with only node can be written as shown below:



Using the above list, "Can you tell when we say there is only one node in circular list?" Observe that if there is only one node, then *link* field of *last* is *last*. The code to check for only one node can be written as:

```
if (last->link == last)          // There is only one node in the circular list
```

9.4 💻 Circular and doubly linked Lists

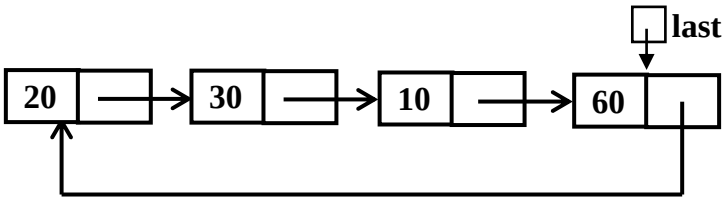
A circular list can be used as a **stack** or a **queue**. To implement these data structures, we require the following functions:

- **insert_front** – To insert an element at the front end of the list
- **insert_rear** – To insert an element at the rear end of the list
- **delete_front** – To delete an element from the front end of the list
- **delete_rear** - To delete an element from the rear end of the list
- **display** – To display the contents of the list

9.1.1 Insert a node at the front end

Now, let us see "How to insert a node at the front end of the list?"

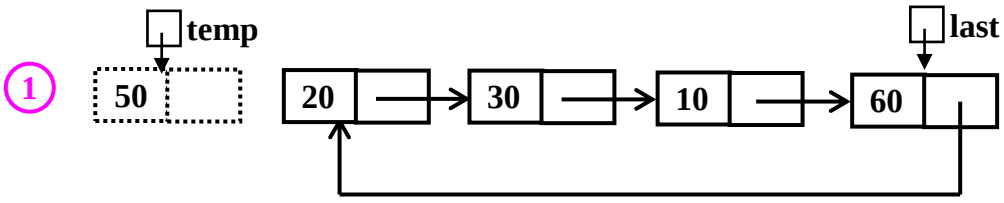
Design: To design the function easily, let us consider a list with 4 nodes. Here, variable ***last*** contains address of the last node of the list.



Step 1: Obtain a node using getnode() function and copy *item* into *info* field as shown below:

```
1   temp = getnode();
    temp->info = item;
```

After executing the above statements, assuming the *item* is 50, the linked list can be written as shown below:



Step 2: Copy the address of the first node (i.e., last->link) into *link* field of *temp*. This is possible if list is existing (i.e., *last* is not NULL). The code for this can be written as:

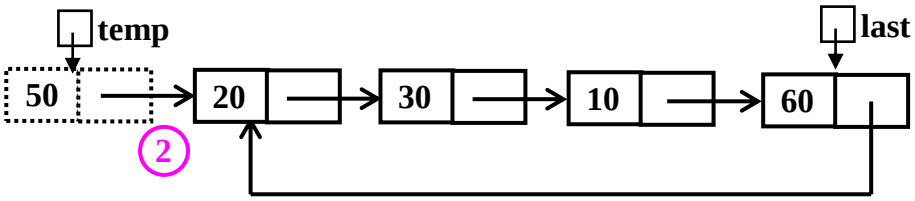
```
if (last != NULL)
        temp->link = last->link;
```

If *last* is NULL, make *temp* itself as the last node. Now, the above code can be modified as shown below:

```
if ( last != NULL)              // List is existing
        temp->link = last->link;
else
        temp = last;
```

(2)

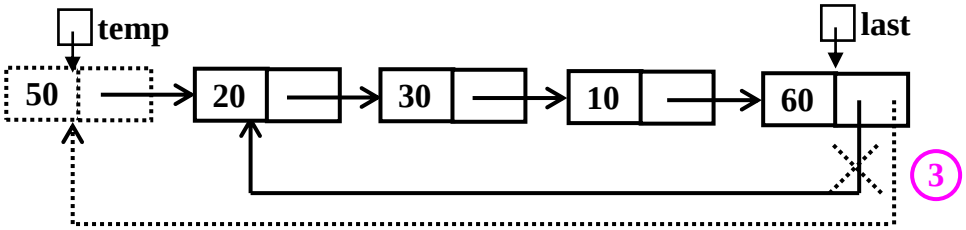
After executing the above code, the list can be written as shown below:



Step 3: Make *temp* as the first node. This is possible by copying *temp* into *link* field of *last* node as shown below:

(3) `last->link = temp;`

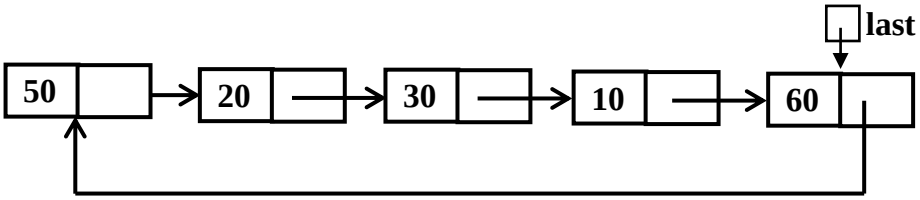
Now, the list can be written as shown below:



Step 4: Finally, we return address of the last node using the statement:

return last;

The list as seen from the calling function is shown below:



9.6 💻 Circular and doubly linked Lists

The C function to insert an item at the front of the circular linked list is shown below:

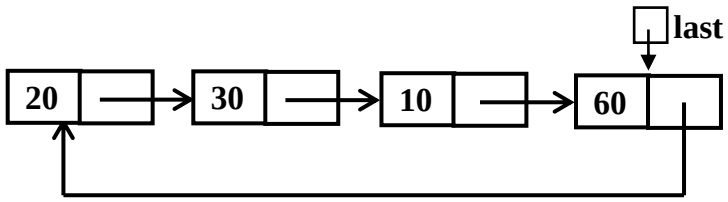
Example 9.1: Function to insert an item at the front end of the list

```
NODE insert_front(int item, NODE last)
{
        NODE temp;

(1)     temp = getnode();                   /* Create a new node to be inserted */
        temp->info = item;

        if ( last == NULL )                 /* Make temp as the first node */
(2)             last = temp;
        else
                temp->link = last->link;    /* Insert at the front end */

(3)     last->link = temp;                  /* link last  node to first node */

        return last;                        /* Return the last node */
}
```

9.1.2 Insert a node at the rear end

Now, let us see "How to insert a node at the rear end of the list?"

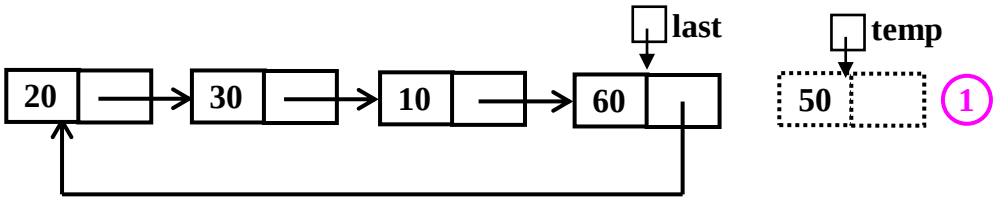
Design: To design the function easily, let us consider a list with 4 nodes. Here, the variable ***last*** contains address of the last node of the list.



Step 1: Obtain a node using getnode() function and copy *item* into *info* field as shown below:

```
(1)     temp = getnode();
        temp->info = item;
```

After executing the above statements, assuming the *item* is 50, the linked list can be written as shown below:



Step 2: Copy the address of the first node (i.e., last->link) into *link* field of *temp*. This is possible if list is existing (i.e., *last* is not NULL). The code for this can be written as:

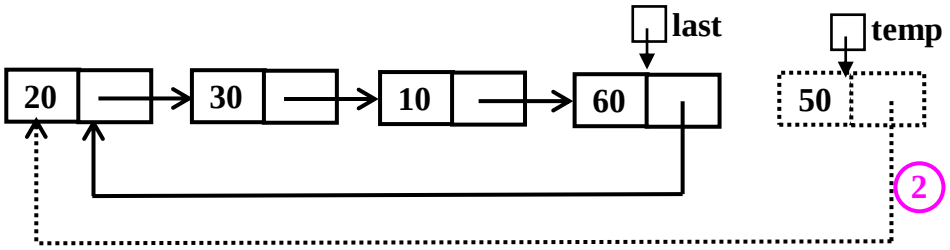
```
if (last != NULL)
        temp->link = last->link;
```

If *last* is NULL, make *temp* itself as the last node. Now, the above code can be modified as shown below:

```
if ( last != NULL)              // List is existing
        temp->link = last->link;
else
        temp = last;
```

(2)

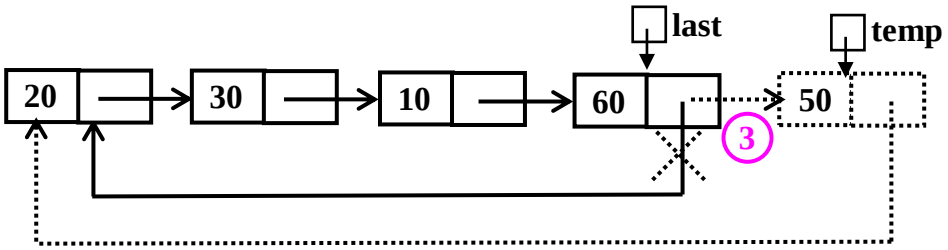
After executing the above code, the list can be written as shown below:



Step 3: Make *temp* as the last node. This is possible by copying *temp* into *link* field of *last* node as shown below:

(3) last->link = temp;

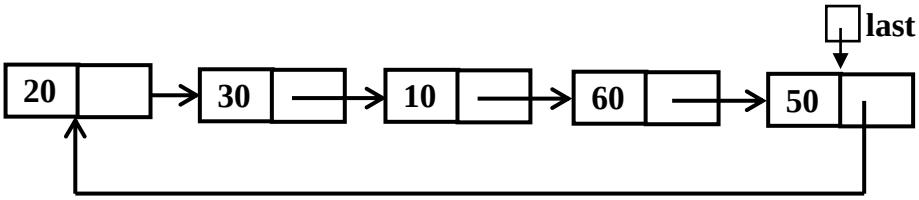
Now, the list can be written as shown below:



Step 4: Finally, we return address of the last node using the statement:

return temp;

The list as seen from the calling function is shown below:



The C function to insert an item at the rear end of the list is shown below:

Example 9.2: Function to insert an item at the rear end of the list

```
NODE insert_rear(int item, NODE last)
{
        NODE temp;

(1)     temp = getnode();                   /* Create a new node to be inserted */
        temp->info = item;

        if ( last == NULL )                 /* Make temp as the first node */
(2)             last = temp;
        else
                temp->link = last->link;    /* Insert at the rear end */

(3)     last->link = temp;                  /* link last  node to first node */

        return temp;                        /* Make the new node as the last node.*/
}
```

9.1.3 Delete a node from the front end

Now, let us see "How to delete a node from the front end of the list?"

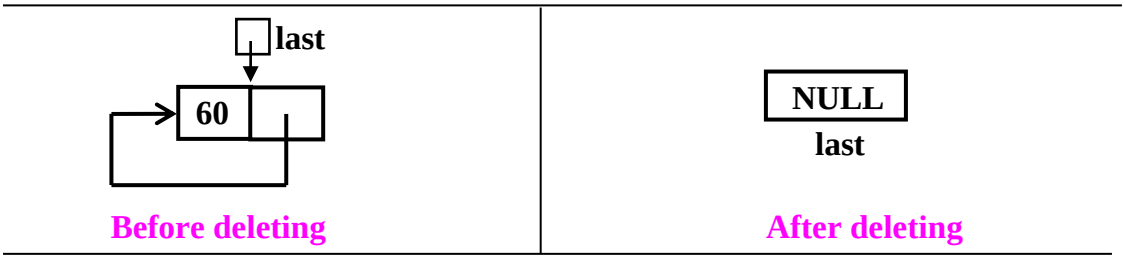
Design: When an item is deleted from the list, the various cases to be considered are:

- List is empty
- List containing one element
- List containing more than one element.

Case 1: List is empty: It is not possible to delete an element from the list whenever list is empty. In such situation, we display the appropriate message as shown below:

```
if ( last == NULL )                    /* Check for empty list */
{
(1)     printf("List is empty\n");
        return NULL;
}
```

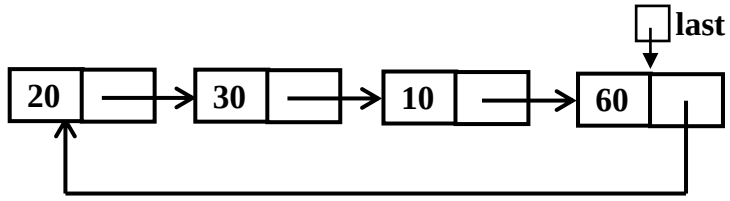
Case 2: List with one node: A list with only node can be written as shown below:



Observe that after deleting the only node in the list, the list will be empty and we have to return NULL. The code for this can be written as shown below:

```
if (last->link == last)          /* There is only one node in the list */
{
        printf("Item deleted = %d\n", last->info);      // Item deleted = 60
(2)     free(last);                                     // delete the node

        return NULL;                                    // return empty list
}
```

Case 3: List with more than one node: Consider the list with more than one node.

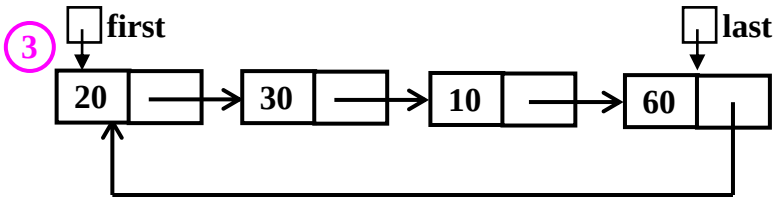


The sequence of steps to be followed to delete an element from the front end of the list can be written as shown below:

Step 1: Obtain the address of the first node. The code for this can be written as shown below:

```
3   first = last->link;
```

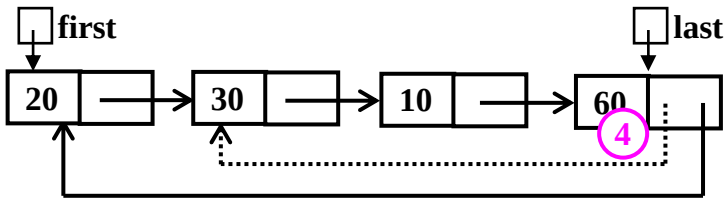
The list after executing above statement is shown below:



Step 2: Make second node as first node: This is obtained by copying first->link to last->link. The code for this case can be written as shown below:

```
4   last->link = first->link;
```

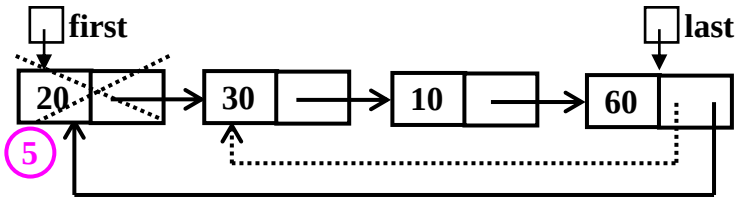
Thus, the first node is isolated and second node becomes the first node. The list after executing above statement is shown below:



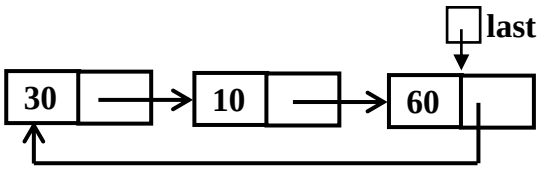
Step 3: Remove the first node: The first node in the list can be removed using free() function. But, before removing the node, display the appropriate message. The code for this case can be written as shown below:

5 **printf** (“The item deleted is %d\n”, first->info);
free(first);

After executing the above code, the list can be written as shown below:



The final list as seen from the calling function can be written as shown below:



Thus, a node at the front end can removed. The complete code to delete an item from the front end is shown below:

Example 9.3: Function to delete an item from the front end

```
NODE delete_front(NODE last)
{
        NODE first;

        if ( last == NULL )                          /* Check for empty list */
        {
1               printf("List is empty\n");
                return NULL;
        }

        if ( last->link == last )                    /* Delete if only one node */
        {
                printf("The item deleted is %d\n", last->info);
2               free(last);
                return NULL;
        }
```

```
        /* List contains more than one node */
(3)     first = last->link;                 /* obtain node to be deleted */

(4)     last->link = first->link;           /*Store new first node in link of last */

(5)     printf("Item deleted is %d\n", first->info);
        free(first);                        /* delete the old first node */

        return last;                        /* return always address of last node */
}
```

9.1.4 Delete a node from the rear end

Now, let us see "How to delete a node from the rear end of the list?"

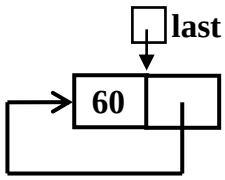
Design: Similar to the previous section, if we want to delete an item from the rear end of the list, it results in following three cases:

- List is empty
- List containing one element
- List containing more than one element.

Case 1: List is empty: It is not possible to delete an element from the list whenever list is empty. In such situation, we display the appropriate message as shown below:

```
        if ( last == NULL )                 /* Check for empty list */
        {
(1)             printf("List is empty\n");
                return NULL;
        }
```

Case 2: List with one node: A list with only node can be written as shown below:



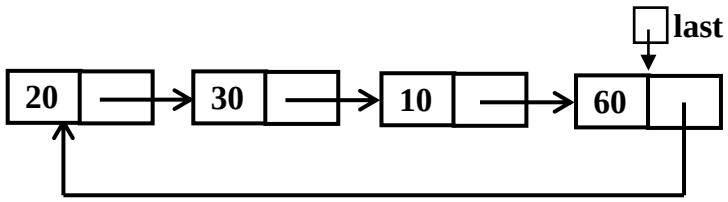
Before deleting

After deleting

Observe that after deleting the only node in the list, the list will be empty and we have to return NULL. The code for this can be written as shown below:

```
if (last->link == last)          /* There is only one node in the list */
{
        printf("Item deleted = %d\n", last->info);      // Item deleted = 60
(2)     free(last);                                     // delete the node
        return NULL;                                    // return empty list
}
```

Case 2: List with more than one node: Consider the list with more than one node.

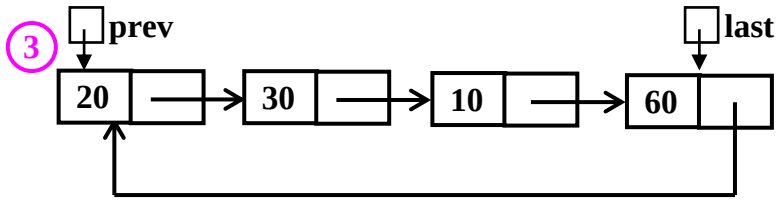


The sequence of steps to be followed to delete an element from the rear end of the list can be written as shown below:

Step 1: Obtain the address of the first node. The code for this can be written as shown below:

```
(3)  prev = last->link;
```

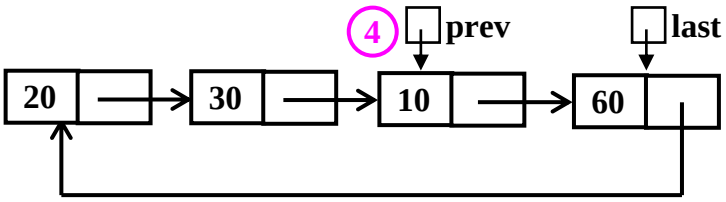
The list after executing above statement is shown below:



Step 2: Find the address of the last but one node: This can be done by updating *prev* as long as prev->link is not *last.* The code for this case can be written as shown below:

```
        while (prev->link != last)
(4)     {
                prev = prev->link;
        }
```

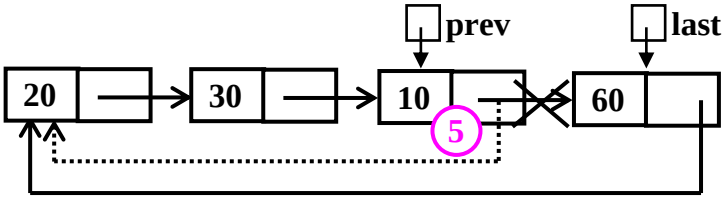
After executing the above loop, the variable *prev* contains address of the last but one node as shown below:



Step3: Make last but one node as the last node: This can be done by copying *first node* (that is, last->link) into prev->link. The code for this case can be written as shown

```
5  prev->link = last->link;
```

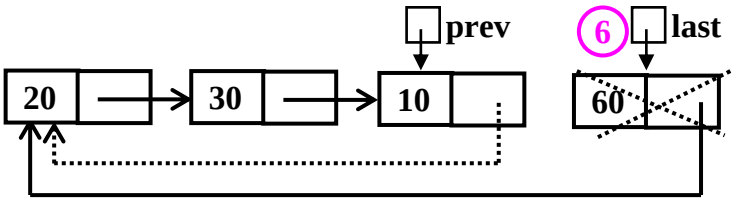
Thus, the last node is isolated and last but one node becomes the last node. The list after executing above statement is shown below:



Step 4: Remove the last node: The last node in the list can be removed using free() function. But, before removing the node, display the appropriate message. The code for this case can be written as shown below:

```
6  printf ("The item deleted is %d\n", last->info);
   free(last);
```

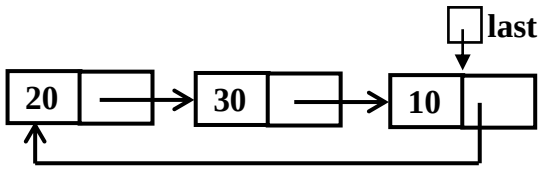
After executing the above code, the list can be written as shown below:



The new last node *prev* can be returned using the following statement:

```
return prev;
```

The list as seen from the calling function is shown below:



Thus, a node at the rear end can removed. The complete code to delete an item from the rear end is shown below:

Example 9.4: Function to delete an item from the rear end

```
NODE delete_rear (NODE last)
{
        NODE prev;

        if ( last == NULL )                    /* Check if list is empty */
        {
                printf("List is empty\n");
                return NULL;
        }

        if ( last->link == last )              /* Delete if only one node */
        {
                printf("The item deleted is %d\n", last->info);
                free(last);
                return NULL;
        }

        prev = last->link;                     /* Obtain address of previous node */
        while( prev->link != last )
        {
                prev = prev->link;
        }

        prev->link = last->link;               /* prev node is made the last node */

        printf("The item deleted is %d\n", last->info);

        free(last);                            /* delete the old last node */

        return prev;                           /* return the new last node */
}
```

9.1.5 Display circular singly linked list

The C function to display the contents of circular list is shown below: The reader is required to understand how the function is working.

Example 9.5: Function to display the contents of the circular queue

```
void display(NODE last)
{
        NODE temp;

        if ( last == NULL )                     /* Check for empty list */
        {
                printf("List is empty\n");
                return;
        }

        printf("Contents of the list is\n");    /* Display till we get last node */

        temp = last->link;                      /* Get the address of first node */
        while (temp != last)                    /* Traverse till the end */
        {
                printf("%d   ",temp->info);
                temp = temp->link;
        }

        printf("%d\n", temp->info);             /* Display last node */
}
```

The C program to implement double ended queue using circular linked list is below:

Example 9.6: Program to implement dequeue using circular list

```
#include <stdio.h>
#include <stdlib.h>
#incude <alloc.h>
struct node
{
        int             info;
        struct node     *link;
};
```

```
typedef struct node* NODE;
/* Include: Example 8.2: Function to get a node from OS */
/* Include: Example 9.1: Function to insert item at front end of the list */
/* Include: Example 9.2: Function to insert item at rear end of the list */
/* Include: Example 9.3: Function to delete an item from the front end */
/* Include: Example 9.4: Function to delete an item from the rear end */
/* Include: Example 9.5: Function to display the contents of the circular queue */

void main()
{
        NODE        last;
        int         choice, item;

        last = NULL;
        for (;;)
        {
                printf("1:Insert_Front   2:Insert_Rear\n");
                printf("3:Delete_Front   4:Delete_Rear\n");
                printf("5:Display        6:Exit\n");
                printf("Enter the choice\n");
                scanf("%d", &choice);

                switch(choice)
                {
                        case 1:
                                printf("Enter the item to be inserted\n");
                                scanf("%d", &item);
                                last = insert_front (item, last);
                                break;
                        case 2:
                                printf("Enter the item to be inserted\n");
                                scanf("%d", &item);
                                last = insert_rear (item, last);
                                break;
                        case 3:
                                last = delete_front(last);
                                break;
                        case 4:
                                last = delete_rear(last);
                                break;
```

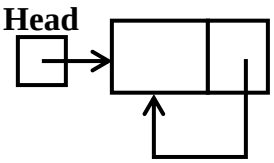
```
                    case 5:
                        display(last);
                        break;
                    default:
                        exit(0);
                }
        }
}
```

9.2 Circular list with a header node

Now, let us see "What is circular singly linked list with header node?"

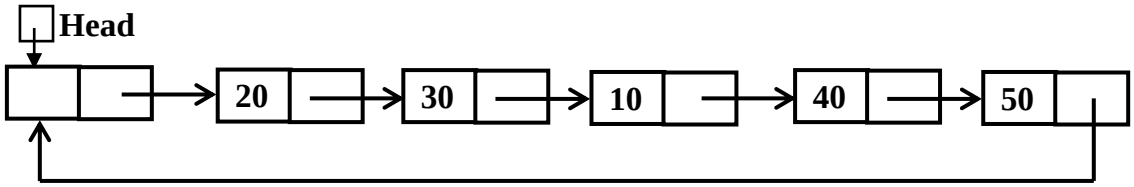
Definition: In a circular list with a header node, the *link* field of the last node contains address of the header node and the *link* field of the header node contains the address of the first node. Usually the info field of a header node does not contain any information. Sometimes, some useful information such as number of nodes in the list can be stored.

Example 1: An empty circular list using header is represented as shown below:



So, whenever *link* field of header node contains address of itself, then we say list is empty.

Example 2: The non-empty list using header node is represented as shown below:

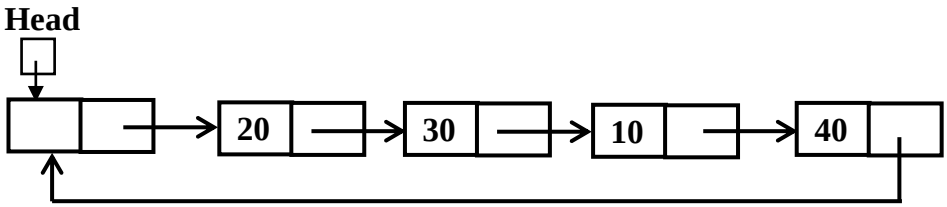


Note: In the above list, *link* field of last node contains the address of header node. While inserting or deleting elements, we can assume that the list is already present and we can write the code. The code thus written works for all other cases such as *list empty*, *a list having only one node* etc.

Now, let us see "What are the various operations that can be performed on circular singly linked list with header node?"

9.2.1 Insert a node at the front end

Now, let us see "How to insert an item at the front end?" Consider the circular list with a header node shown in figure below.

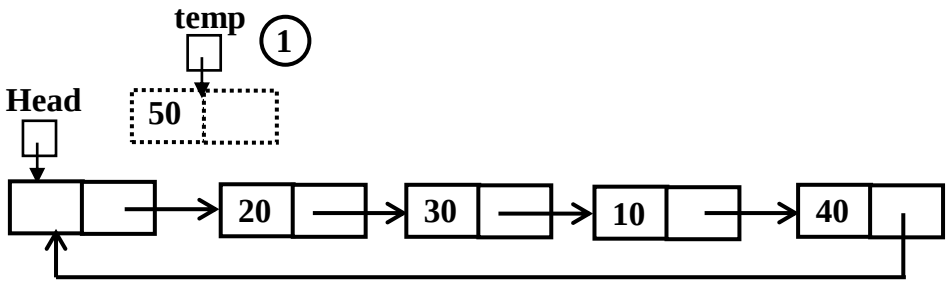


An *item* 50 can be inserted at the front end of the list using the following sequence of steps:

Step 1: Create a node: This can be done by using getnode() function and copying *item* into the *info* field as shown below:

```
(1)  temp = getnode()
     temp->info = item;
```

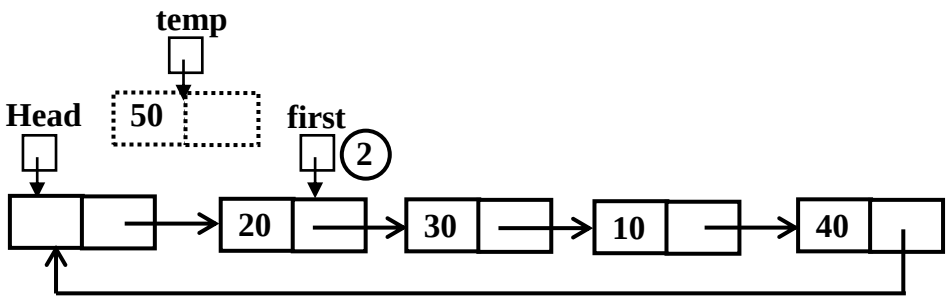
The list after executing above statements can be written as shown below:



Step 2: Obtain the address of the first node: The first node can be accessed using "head->link" and copy this into *first* as shown below:

```
(2)  first = head->link;
```

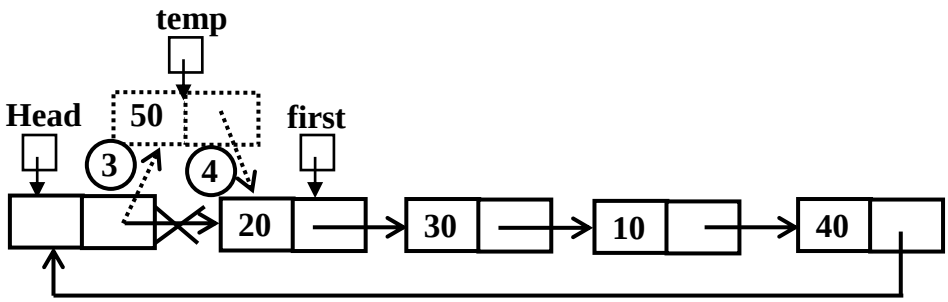
Now, the variable *first* contains address of the first node and the linked list can be written as shown below:



Step 3: Make the new node created as the first node: This can be done by inserting the node *temp* between *head* and *first*. This can be achieved using the following code:

```
3   head->link = temp;
4   temp->link = first;
```

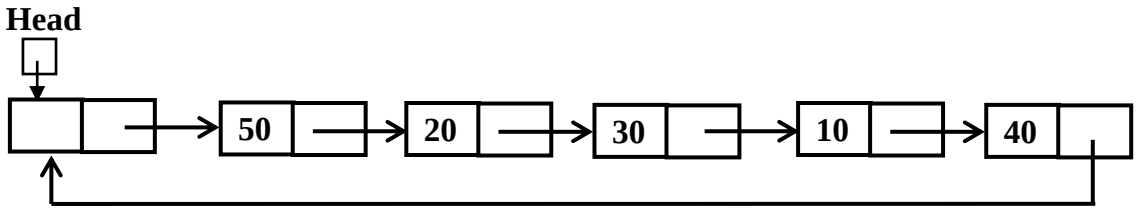
The list obtained after executing the above statements can be written as shown below:



Step 4: Finally return the address of the header node using the following statement:

```
return head;
```

The list as seen from the calling function can be written as shown below:



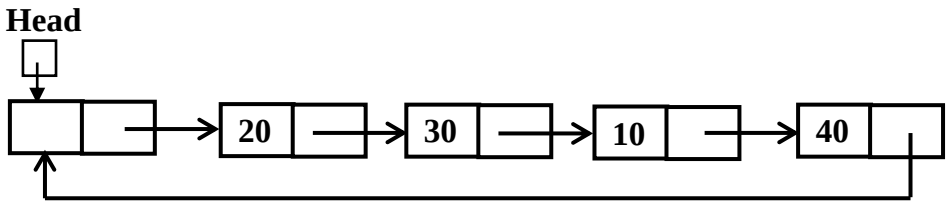
The complete C function is shown below:

Example 9.7: Function to insert at the front end of the list

```
NODE insert_front(int item, NODE head)
{
        NODE temp;
    1   temp = getnode();                  /* Create a node, insert the item */
        temp->info = item;
    2   first  = head->link;               /* Obtain the address of the first node */
    3   head->link = temp;                 /* Insert at the front end */
    4   temp->link = first;
        return head;                       /* Return the header node */
}
```

9.2.2 Insert a node at the rear end

Now, let us see "How to insert an item at the rear end?" Consider the circular list with a header node shown in figure below.

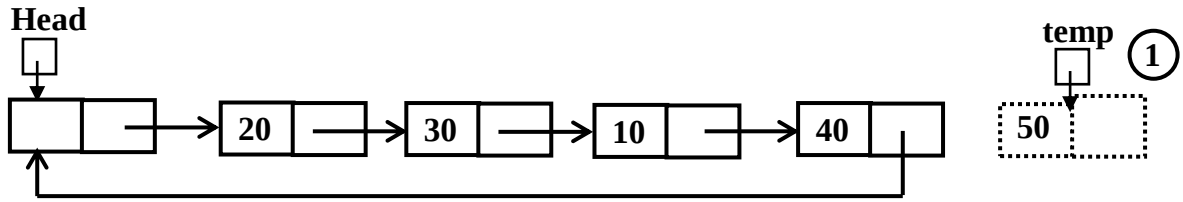


An *item* 50 can be inserted at the rear end of the list using the following sequence of steps:

Step 1: Create a node: This can be done by using getnode() function and copying *item* into the *info* field as shown below:

```
    1   temp = getnode()
        temp->info = item;
```

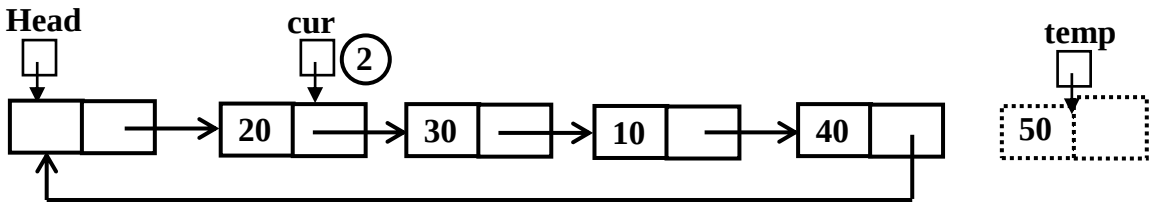
The list after executing above statements can be written as shown below:



Step 2: Obtain the address of the first node: The first node can be accessed using "head->link" and copy this into *cur* as shown below:

```
2   cur = head->link;
```

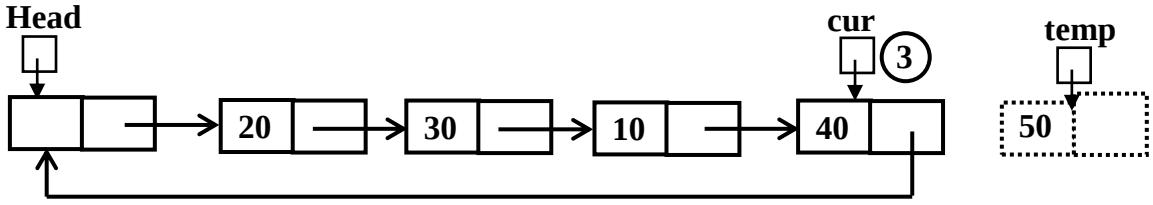
Now, the variable *cur* contains address of the first node and the linked list can be written as shown below:



Step 3: Obtain the address of the last node: This can be done by updating *cur* repeatedly to contain address of the next node till cur->link is not *head*. This can be achieved using the following code:

```
3   while (cur->link != head)
    {
            cur = cur->link;
    }
```

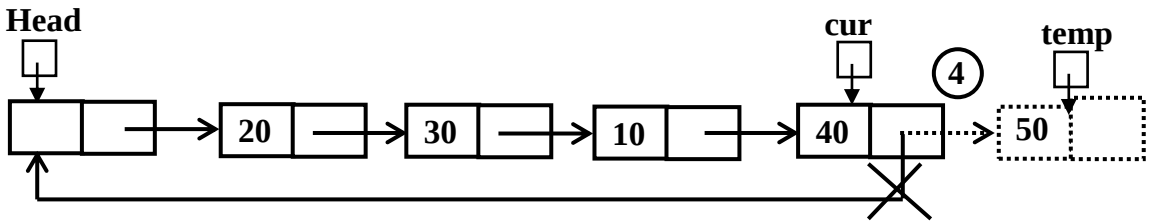
Once control comes out of the loop, the variable *cur* contains address of the last node. Now, the linked list obtained can be written as shown below:



Step 4: Make the new node created as the last node: This can be done by inserting the node *temp* after *cur*. This can be done by copying *temp* to *link* field of *cur* node as shown below:

④ cur->link = temp;

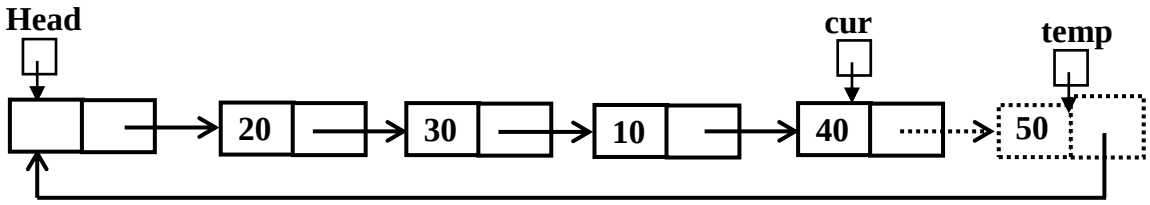
The list obtained after executing the above statements can be written as shown below:



Since *temp* is the last node, the *link* field of *temp* should contain address of the header node *head*. This can be done using the statement:

⑤ temp->link = head;

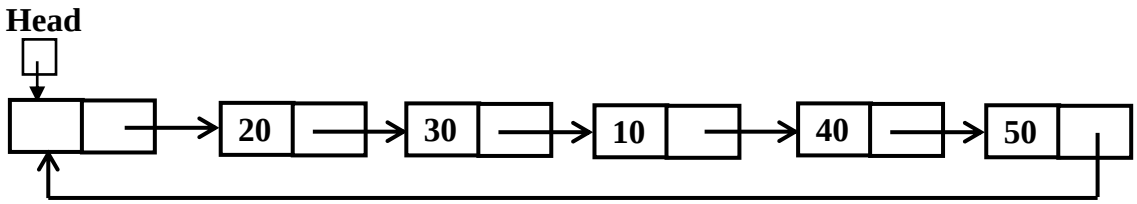
After executing the above instruction, the linked list can be written as shown below:



Step 5: Finally return the address of the header node using the following statement:

return head;

The list as seen from the calling function can be written as shown below:



The complete C function is shown below:

Example 9.8: Function to insert at the rear end of the list

```
NODE insert_rear(int item, NODE head)
{
        NODE temp, cur;

(1)     temp = getnode()                    /* create a node */
        temp->info = item;

(2)     cur = head->link;                   /* obtain the address of the first node */

(3)     while (cur->link != head)           /* obtain the address of last node */
        {
                cur = cur->link;
        }

(4)     cur->link = temp;                   /* insert the node at the end */
(5)     temp->link = head;

        return head;
}
```

9.2.3 Delete a node from the rear end

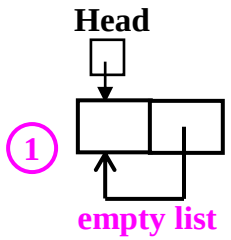
Now, let us see "How to delete a node from the rear end of the list?"

Design: If we want to delete an item from the rear end of the list, it results in following two cases:

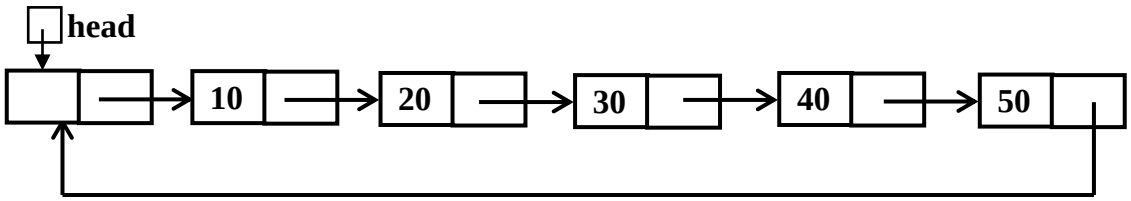
- List is empty
- List containing one or more elements

Case 1: List is empty: The pictorial representation of an empty circular list with a header node and the equivalent code to check for empty list can be written as shown below:

```
        if (head->link == head)
        {
(1)             printf("List is empty\n");
                return head;
        }
```



Case 2: List with more than one node: Consider the list with more than one node.

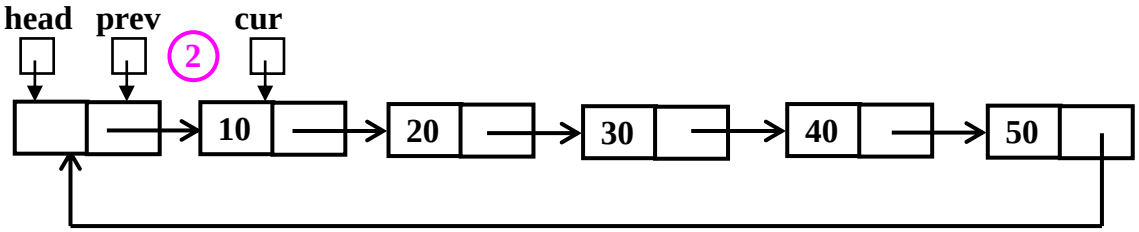


The sequence of steps to be followed to delete an element from the rear end of the list can be written as shown below:

Step 1: Get the first node and its predecessor. The code for this can be written as shown below:

```
(2) cur = head->link;          // first node of the list
    prev = head;               // previous to the first node
```

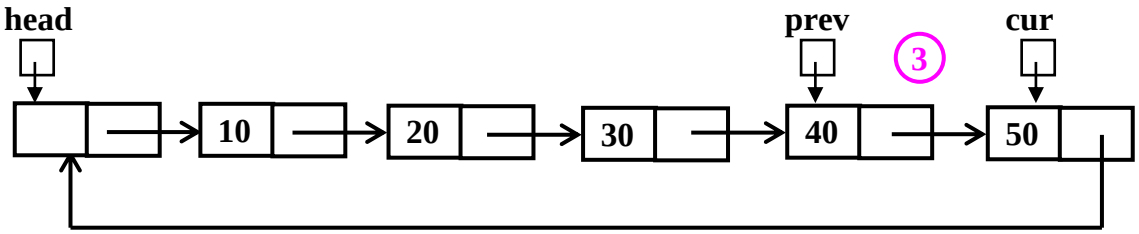
The list after executing above statements can be written as shown below:



Step 2: Get the address of last node and its previous node. This can be done by repeatedly updating *cur* and *prev* as long as *cur->link* is not equal to *head*. The code for this can be written as shown below:

```
while (cur->link != head)
{
(3)     prev = cur;
        cur = cur->link;
}
```

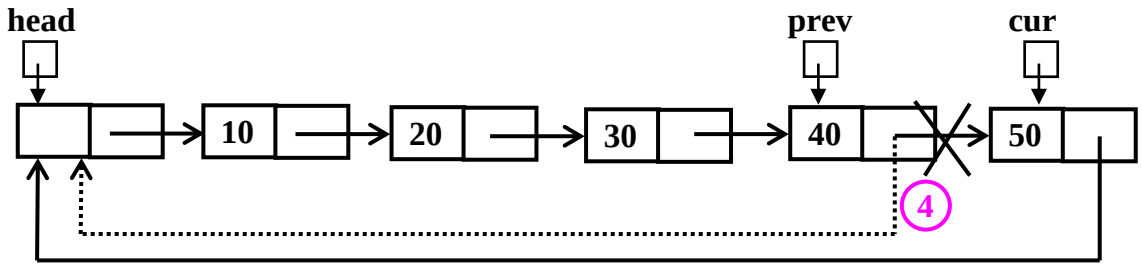
The list obtained after executing the above statements can be written as shown below:



Step3: Make last but one node as the last node: This can be done by copying *head* into prev->link. The code for this case can be written as shown below:

```
4  prev->link = head;
```

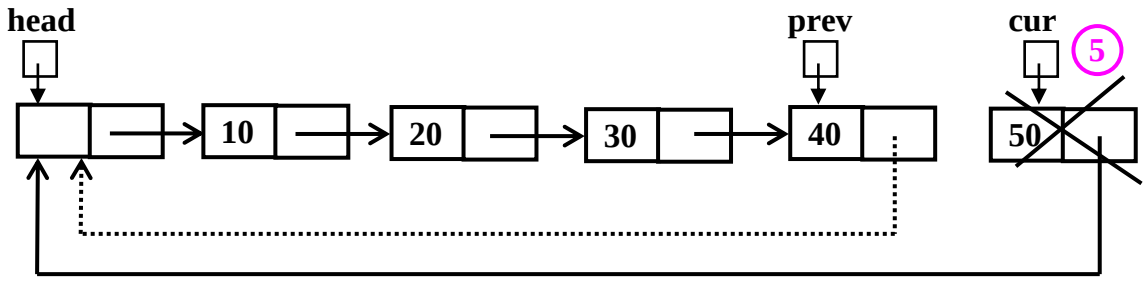
Thus, the last node is isolated and last but one node becomes the last node. The list after executing above statement is shown below:



Step 4: Remove the last node: The last node in the list can be removed using free() function. But, before removing the node, display the appropriate message. The code for this case can be written as shown below:

```
5  printf ("The item deleted is %d\n", cur->info);
   free(cur);
```

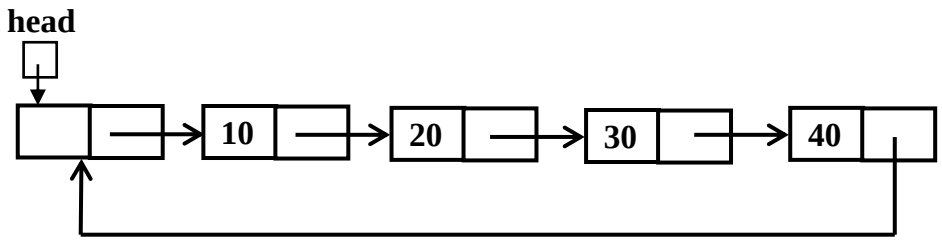
After executing the above code, the list can be written as shown below:



Step 5: return the header node: This can be done by using the following code:

```
return head;
```

The list as seen from the calling function can be written as shown below:



Thus, a node at the rear end can be removed. The complete code to delete an item from the rear end is shown below:

Example 9.9: Function to delete an item from the rear end

```
NODE delete_rear (NODE head)
{
        NODE prev, cur;

        if (head->link == head)                 // check for empty list
        {
(1)             printf("List is empty\n");
                return head;
        }

(2)     cur = head->link;                       // first node of the list
        prev = head;                            // previous to the first node

        while (cur->link != head)               // Obtain last node and its previous node
        {
(3)             prev = cur;
                cur = cur->link;
        }

(4)     prev->link = head;                      // Make last but one node as last node

(5)     printf ("The item deleted is %d\n", cur->info);     // delete the earlier last node
        free(cur);

        return head;
}
```

9.2.4 Delete a node from the front end

Now, let us see "How to delete a node from the front end of the list?"

Design: If we want to delete an item from the front end of the list, it results in following two cases:

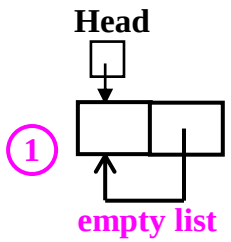
- List is empty
- List containing one or more elements

9.28 Circular and doubly linked Lists

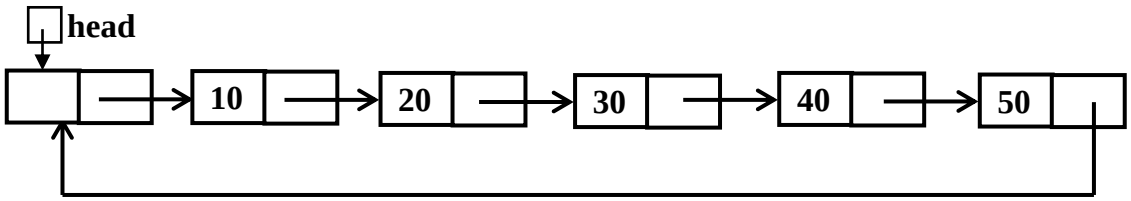
Case 1: List is empty: The pictorial representation of an empty circular list with a header node and the equivalent code to check for empty list can be written as shown below:

```
if (head->link == head)
{
        printf("List is empty\n");
        return head;
}
```

(1)



Case 2: List with more than one node: Consider the list with more than one node.

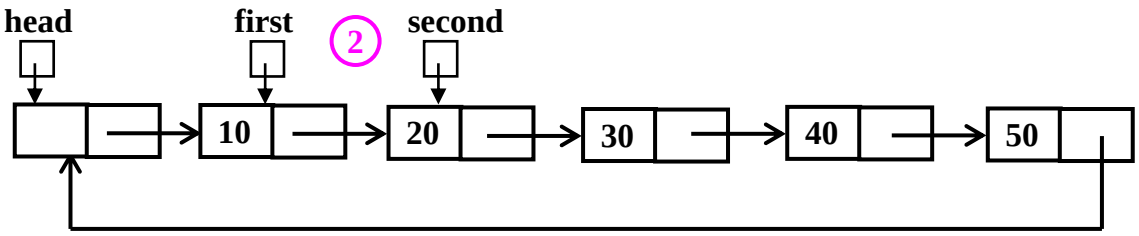


The sequence of steps to be followed to delete an element from the rear end of the list can be written as shown below:

Step 1: Get the first node and its successor. The code for this can be written as shown below:

```
(2) first = head->link;          // first node of the list
    second = first->link;        // second node of the list
```

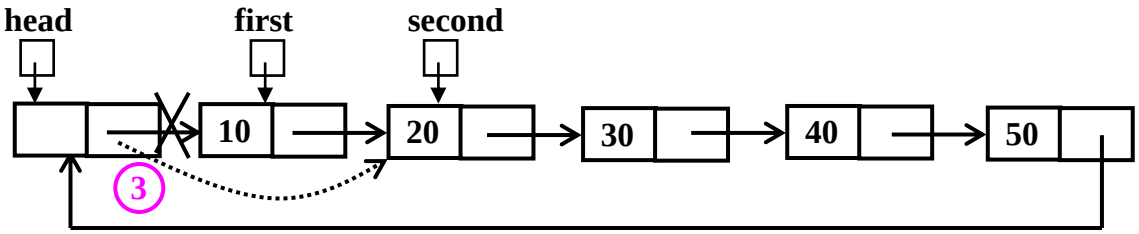
The list after executing above statements can be written as shown below:



Step 2: Make second node as first node: This can be done by copying *second* into head->link. The code for this case can be written as shown below:

```
(3) head->link = second;
```

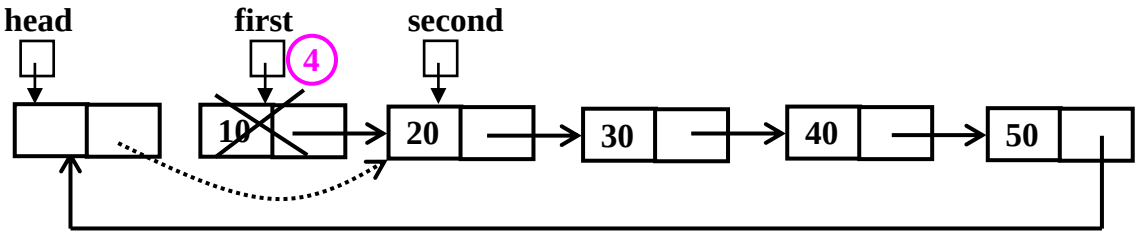
Thus, the first node is isolated and second node becomes the first node. The list after executing above statement is shown below:



Step 4: Remove the first node: The first node in the list can be removed using free() function. But, before removing the node, display the appropriate message. The code for this case can be written as shown below:

```
4  printf ("The item deleted is %d\n", first->info);
   free(first);
```

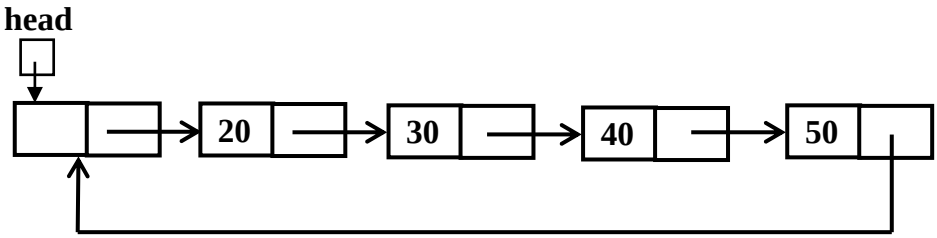
After executing the above code, the list can be written as shown below:



Step 5: return the header node: This can be done by using the following code:

```
return head;
```

The list as seen from the calling function can be written as shown below:



Thus, a node at the front end can be removed. The complete code to delete an item from the front end is shown below:

Example 9.10: Function to delete an item from the front end

```
NODE delete_front (NODE head)
{
        NODE first, second;

        if (head->link == head)              // check for empty list
        {
(1)             printf("List is empty\n");
                return head;
        }

(2)     first = head->link;                  // first node of the list
        second = first->link;                // second node of the list

(3)     head->link = second;                 // Make second node as first node

(4)     printf ("The item deleted is %d\n", first->info);    // delete earlier first node
        free(first);

        return head;
}
```

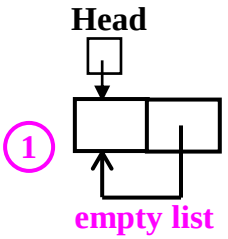
9.2.5 Display circular list with a header node

Now, let us see "How to display the contents of list?"

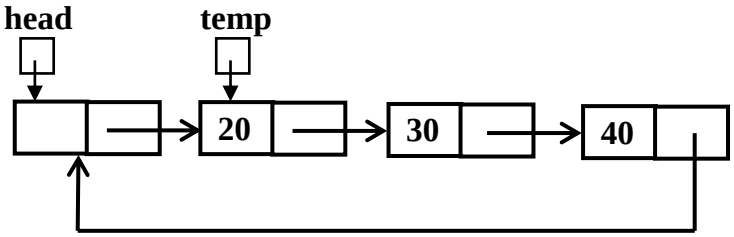
Design: The contents of the list can be displayed by considering various cases as shown below:

Case 1: List is empty: The pictorial representation of an empty circular list with a header node and the equivalent code to check for empty list can be written as shown below:

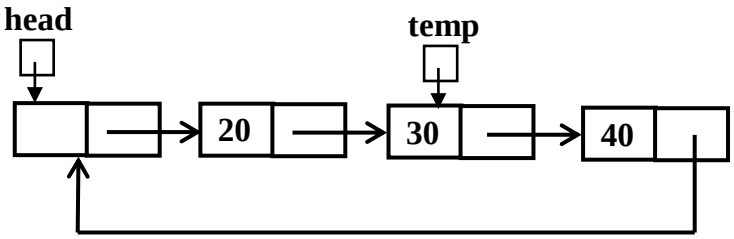
```
        if (head->link == head)
        {
(1)             printf("List is empty\n");
                return;
        }
```



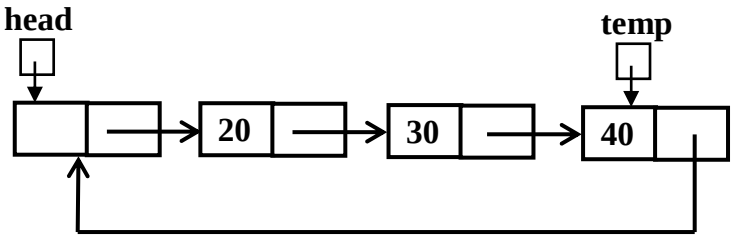
Case 2: List is exiting: Consider a list with three nodes. The variable *head* always contains address of the header node. The list starts from head->link onwards. Let *temp* is a variable that contains the address of the first node as shown in figure below:



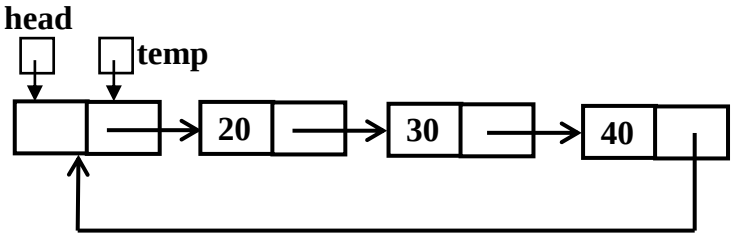
Display 20 and update *temp* using the statement: temp = temp ->link. Now, the list can be written as shown below:



Display 30 and update *temp* using the statement: temp = temp ->link. Now, the list can be written as shown below:



Display 40 and update *temp* using the statement: temp = temp ->link. Now, the list can be written as shown below:



Once *temp* is equal to *head* stop displaying and updating *temp*. It means display *info* of *temp* and update *temp* as long as *temp* is not equal to *head*. Now, the code can be written as shown below:

```
        while (temp != head)
        {
                printf("%d\n", temp ->info);
                temp = temp ->link;
        }
```

Now, the complete program can be written as shown below:

Example 9.11: C function to display the contents of linked list

```
void display(NODE head)
{
        NODE temp;

        if ( head->link == head)                    /* Check for empty list */
        {                                                    Case 1
                printf("List is empty\n");
                return;
        }

        printf("The contents of  singly linked list\n");     Case 2
        temp = head->link;                  /* Holds address of the first node */
        while ( temp != head )              /* As long as no end of list */
        {
                printf("%d  ",temp->info);  /* Display the info field of node */
                temp = temp->link;          /* Point to the next node */
        }
}
```

The program to implement stacks using circular singly linked list is shown below:

Example 9.12: Program for stacks (circular singly linked list with header node)

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
struct node
{
        int             info;
        struct node     *link;
};
```

```
typedef struct node* NODE;

/* Include: Example 8.2: Function to get a new node from the operating system */
/* Include: Example 9.7: Function to insert at the front end of the list */
/* Include:  Example 9.10: Function to delete an item from the front end */
/* Include: Example 9.11: C function to display the contents of linked list */

void main()
{
        NODE        head;
        int         choice, item;

        head  =  getnode();
        head->link = head;

        for (;;)
        {
                printf("1:Insert_Front    2:Delete_Front\n");
                printf("3:Display         4:Exit\n");
                printf("Enter the choice\n");
                scanf("%d", &choice);

                switch(choice)
                {
                        case 1:
                                printf("Enter the item to be inserted\n");
                                scanf("%d", &item);
                                head = insert_front (item, head);
                                break;
                        case 2:
                                head = delete_front(head);
                                break;
                        case 3:
                                display(head);
                                break;
                        default:
                                exit(0);
                }
        }
}
```

9.34 Circular and doubly linked Lists

Now, let us see "What are the disadvantages of singly linked lists whether circular or with or without the header?" Some of the disadvantages of singly linked lists/circular lists are:

- Using singly linked lists and circular lists it is not possible to traverse the list backwards. Two-way traversing is not possible.
- Insertion/Deletion to the left of a designated node *x* is difficult. This requires finding the predecessor of *x* which takes more time.

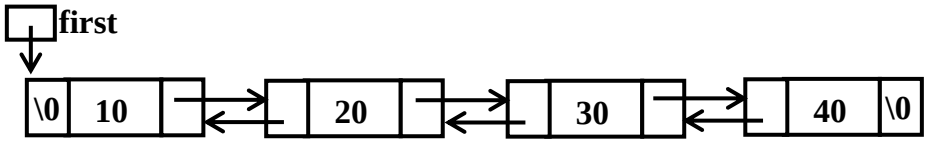
9.3 Doubly linked lists

Now, let us see "What is a doubly linked list?"

Definition: A doubly-linked list is a linear collection of nodes where each node is divided into three parts:

- *info* – This is a field where the information has to be stored
- *llink* – This is a pointer field which contains address of the left node or previous node in the list
- *rlink* – This is a pointer field which contains address of the right node or next node in the list

The pictorial representation of a doubly linked list is shown in figure below:



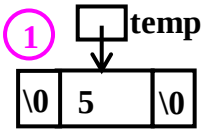
Using such lists, it is possible to traverse the list in forward and backward directions. Such a list where each node has two links is also called a two-way list.

9.3.1 Insert a node at the front end

In this section, let us see "How to insert a node at the front end of the list?"

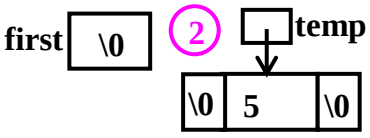
Step 1: Create a node: This can be done using getnode() function and copying *item* 5 as shown below:

```
1   temp = getnode()
    temp->info = item;
    temp->link = temp->rlink = NULL;
```



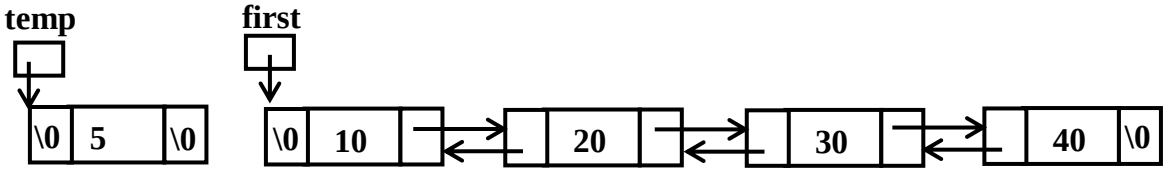
Step 2: Insert into empty list: If the list is empty, the above created node itself should be returned as the first node. The code for this case can be written as shown below:

```
(2) if (first == NULL)  return temp;
```



The list as seen from the calling function can be written as:

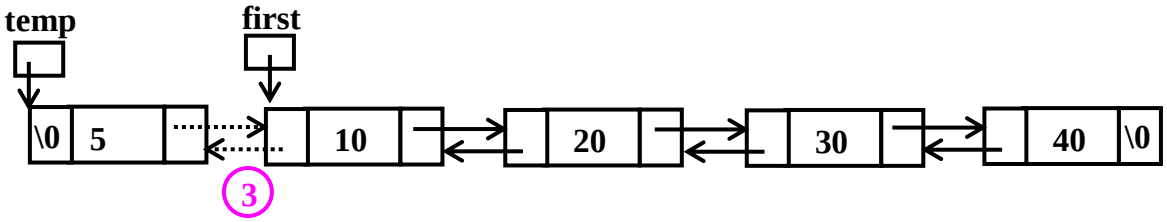
Step 2: Insert into existing list: Consider the following list and see how a node *temp* can be inserted at the front end:



To insert *temp* at the front end of the list, copy *first* into *rlink* of *temp* and copy *temp* into *llink* of *first* node using the following code:

```
(3) temp->rlink = first;
    first->llink = temp;
```

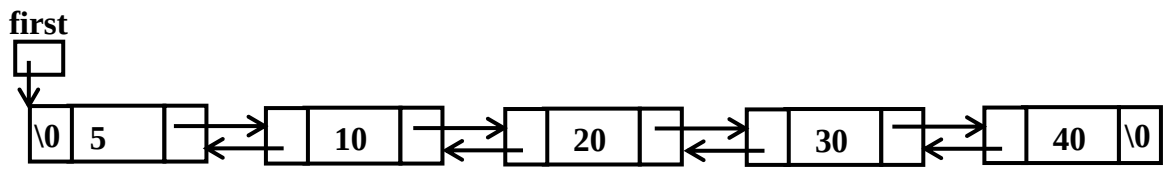
The list after executing the above statements can be written as shown below:



Step 3: Return the first node: After modification, always we have to return the address of the first node. In the above list, *temp* is the first node and it can be returned using the following statement:

```
return temp;
```

Now, the list as seen from calling function can be written as shown below:



9.36 💻 Circular and doubly linked Lists

The C function to insert at the front end can be written as shown below:

Example 9.13: C Function to insert an item at the front end of the list

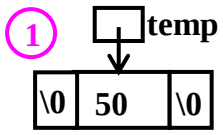
```
NODE insert_front(int item, NODE first)
{
        NODE temp;

        temp = getnode();                       /*obtain a node from OS */

(1)     temp->info = item;                      /* Insert an item into new node */
        temp->link = temp->rlink = NULL;

(2)     if (first == NULL)  return temp;        /* Insert a node for the first time */

(3)     temp->rlink = first;                    /* Insert at the beginning of existing list */
        first->llink = temp;

        return temp;                            /* return address of new first node */
}
```

9.3.2 Insert a node at the rear end

In this section, let us see "How to insert a node at the rear end of the list?"

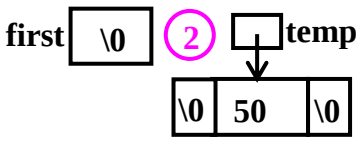
Step 1: Create a node: This can be done using getnode() function and copying *item* 50 as shown below:

```
(1)     temp = getnode()
        temp->info = item;
        temp->link = temp->rlink = NULL;
```



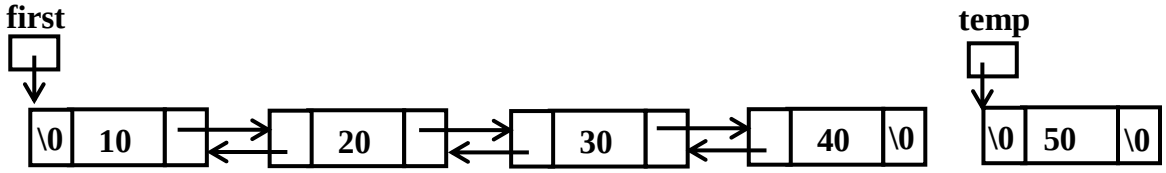
Step 2: Insert into empty list: If the list is empty, the above created node itself should be returned as the first node. The code for this case can be written as shown below:

```
(2)     if (first == NULL)  return temp;
```



The list as seen from the calling function can be written as:

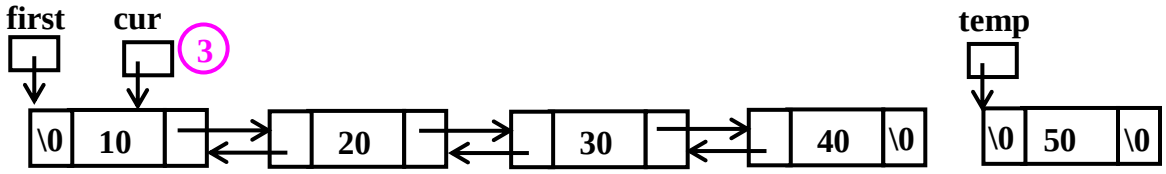
Step 2: Find the address of the last node: Consider the following list and see how a node *temp* can be inserted at the rear end:



Let the variable *first* always contain address of the first node. Let use another variable *cur* to point to the first node. This can be done using the following statement:

③ cur = first;

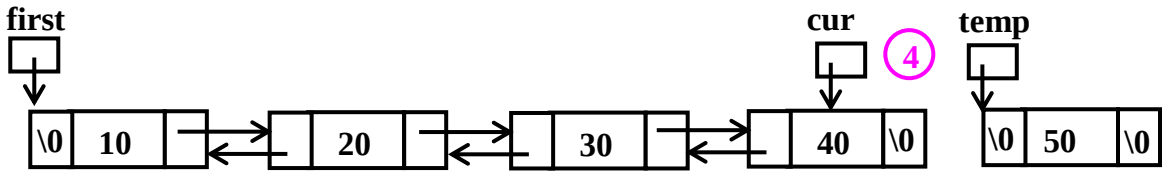
Now, the linked list looks as shown below:



Now, keep updating *cur* to point to the next node as long as *rlink* field of *cur* is not NULL. This can be done using the following code:

④
```
while (cur->rlink != NULL)
{
        cur = cur->rlink;
}
```

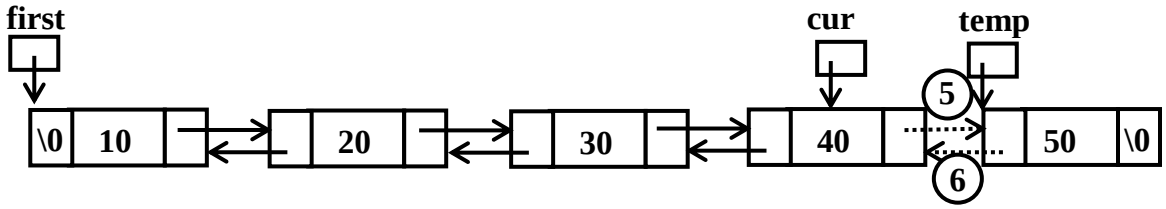
Now, after executing the above while loop, the variable *cur* contains address of the last node of the list as shown below:



Step 3: Insert node at the end: Using the above list, can you tell me how to insert the node at the end. It is very clear from above figure that we have to manipulate two links:

- *rlink* of *cur* should contain address of *temp*. This can be done using the statement:
  (5) cur->rlink = temp;
- *llink* of *temp* should contain address of *cur*. This can be done using the statement:
  (6) temp->llink = cur;

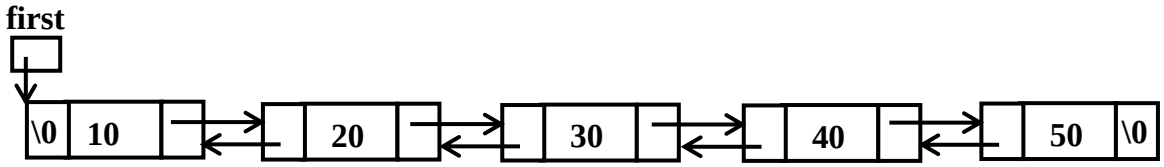
After executing the above two statements, the linked list can be written as shown below:



Step 4: Return address of the first node: This can be done using the statement:

return first;

Now, the list as seen from calling function can be written as shown below:



The C function to insert a node at the rear end can be written as shown below:

Example 9.14: C Function to insert an item at the rear end of the list

```
NODE insert_rear(int item, NODE first)
{
        NODE temp, cur;

        temp = getnode();                    /*obtain a node from OS */

(1)     temp->info = item;                   /* Insert an item into new node */
        temp->link = temp->rlink = NULL;
```

```
(2)  if (first == NULL)  return temp;     /* Insert a node for the first time */

     /* Get the address of the first node */
(3)  cur = first;

     /* Find the address of the last node */
     while (cur->rlink != NULL)
(4)  {
            cur = cur->rlink;
     }

     /* Insert the node at the end */
(5)  cur->rlink = temp;
(6)  temp->llink = cur;

     /* return address of the first node */
     return first;
}
```

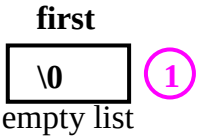
9.3.3 Delete a node from the front end

Now, let us see "How to delete a node from the front end of the list?"

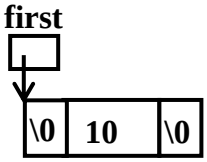
Design: A node from the front end of the list can be deleted by considering various cases as shown below:

Step 1: List is empty: If the list is empty, it is not possible to delete a node from the list. In such case, we display the message "List is empty" and **return** NULL. The code for this can be written as shown below:

```
     if (first == NULL)
     {
(1)        printf ("List is empty\n");
           return NULL;
     }
```

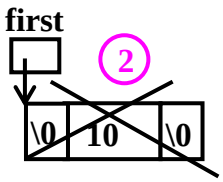


Step 2: Delete if there is only one node: A list having one node can be written as shown below:

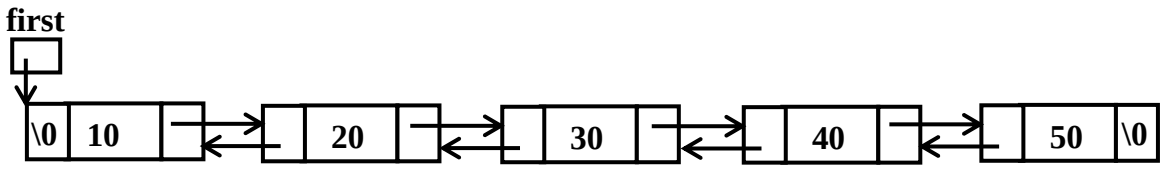


The code to delete the above node can be written as shown below:

```
if (first->rlink == NULL)
{
        printf("Item deleted = %d\n", first->info);
(2)     free(first);

        return NULL;
}
```



When control comes out of the above if-statement, it means that the list has more than one node and the list can be pictorially represented as shown below:

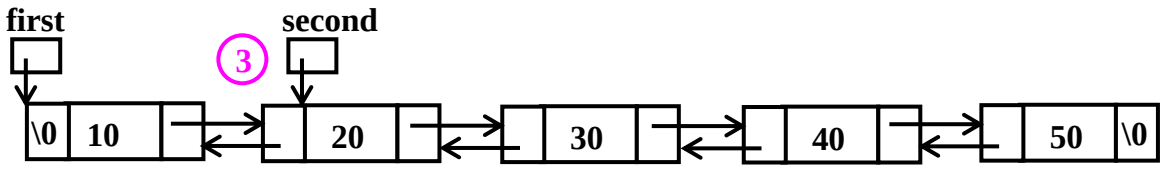


Now, the first node in the above list can be deleted as shown below:

Step 3: Obtain the address of the second node: The address of the second node can be obtained using the following statement:

```
(3)  second = first->rlink;
```

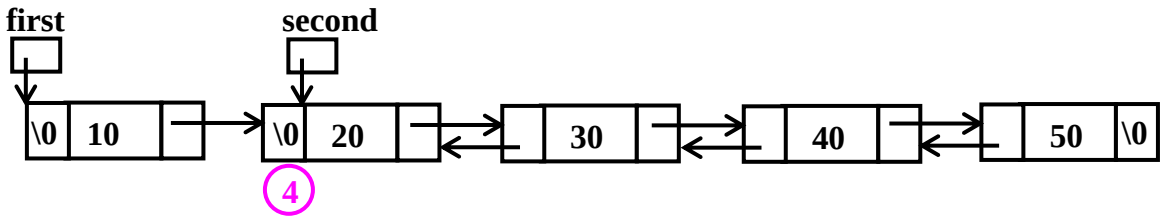
Now, the list can be written as shown below:



Step 4:Make second node as first node: It is achieved by copying NULL to left link of *second* node. It can be done using the following code:

```
(4)  second->llink = NULL;
```

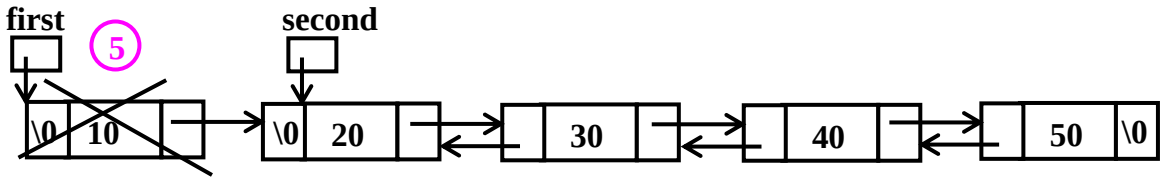
Now, *first* node is isolated and the resulting linked list is shown below:



Step 5: Delete the front node: It is achieved using free() function. The code can be written as shown below:

```
5   printf("Item deleted = %d\n", first->info);
    free(first);
```

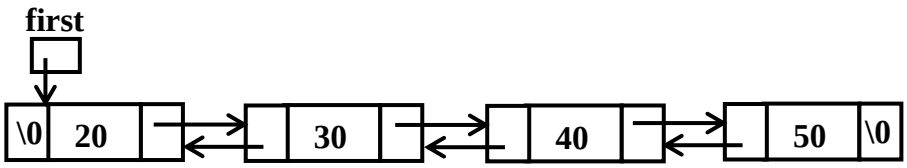
After executing the above code, the list can be written as shown below:



Step 6: Return address of the new first node: Note that the variable *second* now contains address of the new first node and it can be returned using the following statement:

```
return second;
```

Now, the linked list as seen from the calling function can be written as shown below:



Now, the complete C function to delete the first node can be written as shown below:

Example 9.15: C function to delete an item from the front end of the list

```
NODE delete_front(NODE first)
{
        NODE second;
```

```
        if ( first == NULL )                    /* Check for empty list */
        {
(1)             printf("List is empty cannot delete\n");
                return NULL;                    // We can replace NULL with first also
        }

        if (first ->rlink == NULL)              /* Delete if there is only one node */
        {
                printf("Item deleted = %d\n", first->info);
(2)             free(first);

                return NULL;
        }

(3)     second = first->rlink;                  /* Get the address of second node */

(4)     second->llink = NULL;                   /* Make second node as the first node */

(5)     printf("Item deleted = %d\n", first->info);
        free(first);                            /* Delete the first node */

        return second;
}
```

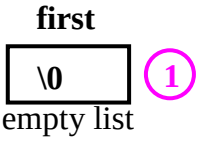
9.3.4 Delete a node from the rear end

Now, let us see "How to delete a node from the rear end of the list?"

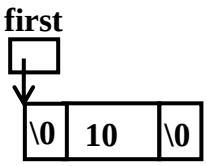
Design: A node from the rear end of the list can be deleted by considering various cases as shown below:

Step 1: List is empty: If the list is empty, it is not possible to delete a node from the list. In such case, we display the message "List is empty" and **return** NULL. The code for this can be written as shown below:

```
        if (first == NULL)
        {
(1)             printf ("List is empty\n");
                return NULL;
        }
```



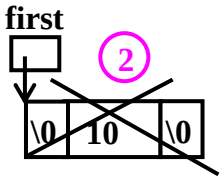
Step 2: Delete if there is only one node: A list having one node can be written as shown below:



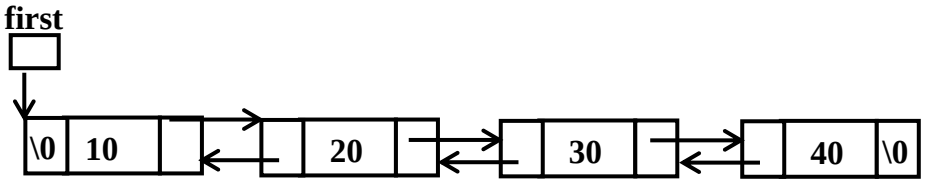
The code to delete the above node can be written as shown below:

```
if (first->rlink == NULL)
{
        printf("Item deleted = %d\n", first->info);
        free(first);

        return NULL;
}
```

2



When control comes out of the above if-statement, it means that the list has more than one node and the list can be pictorially represented as shown below:



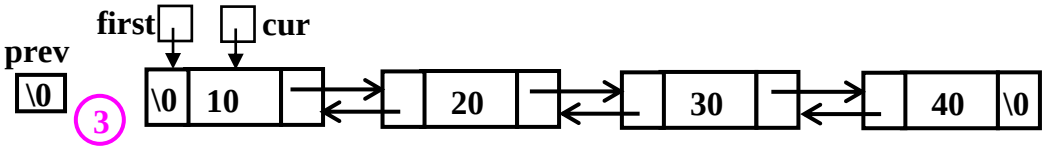
To delete the last node, we should know the address of the last node and last but one node. Let us find the address of last node and last but one node as shown in coming steps.

Step 3: Obtain the address of first node and its predecessor: This can be done using two pointer variables: *cur* and *prev*. Initially, *cur* points to the first node and *prev* points to \0 (null). This is achieved using the following statements:

3

```
prev = NULL;
cur = first;
```

Now, the linked list looks as shown below:

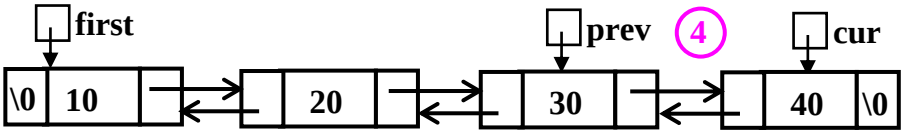


Step 4: Find the address of last node and last but one node: This can be done by updating *cur* and *prev* till *cur* contains address of the last node and *prev* contains address of the last but one node. This can be achieved by using the following statements (see section 8.2.4 for detailed explanation).

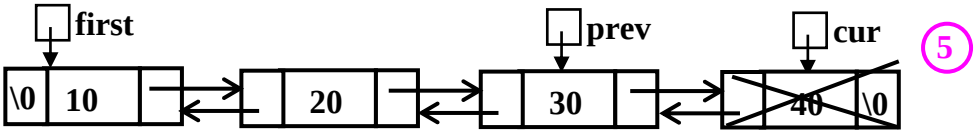
```
while( cur->link != NULL )
{
        prev = cur;
        cur = cur->link;
}
```

4

After executing the above loop, the variable *cur* contains address of the last node and *prev* contains address of last but one node as shown in figure below:



Step 5: The last node pointed to by *cur* can be deleted as shown below:

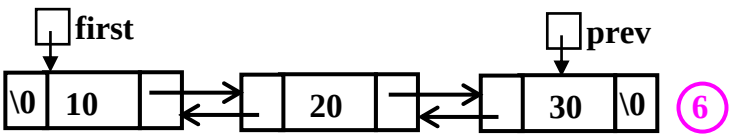


The above activity can be achieved using the following code:

```
printf("Item deleted = %d\n", cur->info);          // Item deleted = 40
free(cur);
```

5

Step 4: Once the last node is deleted (the node shown using cross symbol in above figure), the node pointed to by *prev* should be the last node. This is achieved by copying NULL to *rlink* field of *prev* as shown below:



The code for this case can be written as shown below:

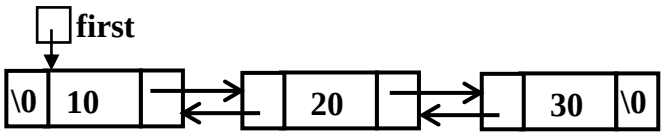
```
prev->rlink = NULL;          /* Node pointed to by prev is the last node */
```

6

Step 5: Finally return address of the first node.

```
return first;                    /* return address of the first node */
```

Now, the linked list as seen from the calling function is shown below:



The complete C function to delete a node from the rear end of the list is shown below:

Example 9.16: Function to delete an item from the rear end of the list

```
NODE delete_rear(NODE first)
{
        NODE cur, prev;
        .....
        if (first == NULL)              /* Check for empty list */
        {
                printf("List is empty cannot delete\n");
(1)             return first;
        }

        .....
        if ( first ->link == NULL )     /*Only one node is present and delete it */
        {
                printf ("The item to deleted is %d\n",first->info);

(2)             free(first);            /* return to availability list */

                return NULL;            /* List is empty so return NULL */
        }

        /* Obtain address of the last node and just previous to that */
(3)     prev = NULL;
        cur = first;
```

```
        while( cur->link != NULL )
        {
(4)             prev = cur;
                cur = cur->link;
        }

        printf("The item deleted is %d\n", cur->info);
(5)     free(cur);                      /* delete the last node */

(6)     prev->rlink = NULL;             /* Make last but one node as the last node */

        return first;                   /* return address of the first node */
}
```

9.3.5 Display doubly linked list

The function that is used to display normal linked list can be used to display the contents of doubly linked list. Only change is that *link* field should be replaced by *rlink*. The C function is shown below:

Example 9.17: C function to display the contents of linked list

```
void display(NODE first)
{
        NODE cur;    int count = 0;

        if ( first == NULL )                    /* Check for empty list */
        {
                printf("List is empty\n");
                return;
        }

        printf("The contents of  singly linked list\n");

        cur = first;                            /* Holds address of the first node */
        while ( cur != NULL )                   /* As long as no end of list */
        {
                printf("%d  ",cur->info);       count++;
                cur = cur->rlink;               /* Point to the next node */
        }
        printf("Number of nodes = %d\n", count);
}
```

9.3.6 Double ended queue using doubly linked lists

The C program to implement dequeue with the help of above functions can be written as shown below:

Example 9.18: Program to implement dequeues using doubly linked list

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>

struct node
{
        int             info;
        struct node     *llink;
        struct node     *rlink;
};
typedef struct node* NODE;

/* Include: Example 8.2: Function to get a new node from the operating system */
/* Include: Example 9.13: Function to insert an item at the front end of the list */
/* Include: Example 9.14: Function to insert an item at the rear end of the list */
/* Include: Example 9.15: Function to delete an item from the front end of the list */
/* Include: Example 9.16: Function to delete an item from the rear end of the list */
/* Include: Example 9.17: Function to display the contents of linked list */

void main()
{
        NODE            first;
        int             choice, item;

        first  =  NULL;

        for (;;)
        {
                printf("1:Insert_Front    2: Insert_Rear\n");
                printf("3:Delete_Front   4: Delete_Rear\n");
                printf("5:Display          6: Exit\n");
                printf("Enter the choice\n");
                scanf("%d", &choice);
```

```
            switch(choice)
            {
                  case 1:
                        printf("Enter the item to be inserted\n");
                        scanf("%d", &item);
                        first = insert_front (item, first);
                        break;

                  case 2:
                        printf("Enter the item to be inserted\n");
                        scanf("%d", &item);
                        first = insert_rear (item, first);
                        break;

                  case 3:
                        first = delete_front(first);
                        break;

                  case 4:
                        first = delete_rear(first);
                        break;

                  case 5:
                        display(first);
                        break;
                  default:
                        exit(0);
            }
      }
}
```

9.4 Circular doubly linked list

In a doubly linked list (discussed in previous section), observe that the *leftlink* of the leftmost node and *rightlink* of rightmost node points to NULL. The two variations of doubly linked lists are:

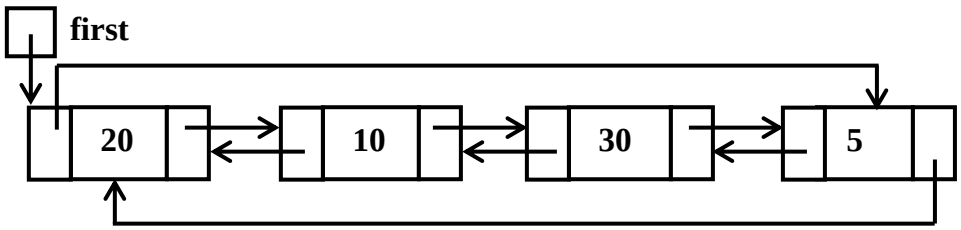
- Circular doubly linked list
- Circular doubly linked list with a header node

Let us see "What is circular doubly linked list?"
Definition: A circular doubly linked list is a variation of doubly linked list in which every node in the list has three fields:

- *info* – This is a field where the information has to be stored
- *llink* – This is a pointer field which contains address of the left node or previous node in the list
- *rlink* – This is a pointer field which contains address of the right node or next node in the list

and the *llink* of the first node contains address of the last node whereas *rlink* of the last node contains address of the first node.

The pictorial representation of circular doubly linked list is shown in figure below:



Using the above list, can you tell "What is the advantage of circular doubly linked list?" Observe the following points:

- As in normal doubly linked list, we can traverse the doubly linked circular list in both directions.
- In normal doubly linked list, given the address of the first node we have to traverse till the end to get the address of last node. But, in circular doubly linked list, we can easily find the address of the last node. The left link of the first node gives the address of the last node.

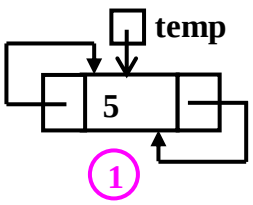
9.4.1 Insert a node at the front end

In this section, let us see "How to insert a node at the front end of the list?"

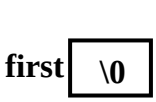
Step 1: Create a node: This can be done using getnode() function and copying *item* 5 as shown below:

```
temp = getnode()
temp->info = item;
temp->link = temp->rlink = temp;
```

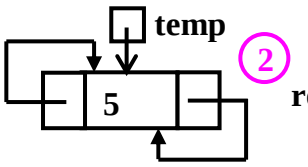
(1)



Step 2: Insert into empty list: If the list is empty, the above created node itself should be returned as the first node.



Initial list

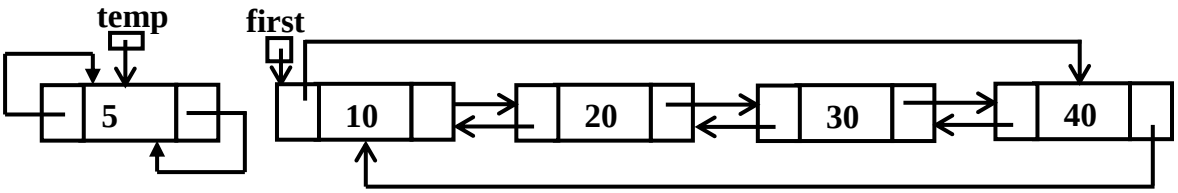


return temp;

The code for this case can be written as shown below:

```
2   if (first == NULL)  return temp;
```

If the above condition fails, it means that list is already existing. The new node *temp* (created in step 1) which has to be inserted at the front end and the existing list can be pictorially represented as shown below:

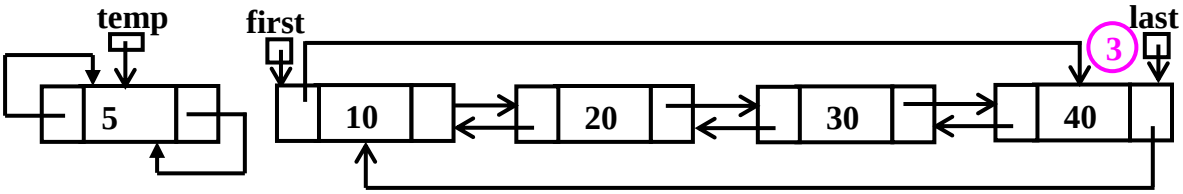


Now, the node *temp* can be inserted at the front end using the following steps:

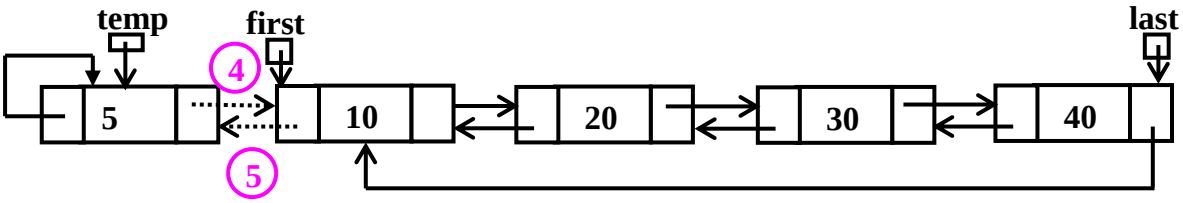
Step 3: Obtain the address of the last node: The last node of the list can be obtained using *llink* of the first node. The code can be written as:

```
3   last = first->llink;          /* Get the address of last node */
```

Now, the linked list can be written as shown below:



Step 4: Link the new node created with first node: This can be done by copying *first* into *rlink* of *temp* and copying *temp* into *llink* of *first* node as shown below:

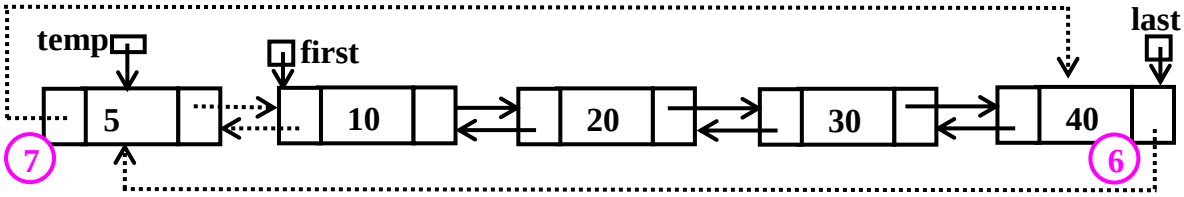


The code can be written as shown below:

4 temp->rlink = first;

5 first->llink = temp;

Step 4: Make new node created as the first node: This can be done by copying *temp* to *rlink* of last node and copying last node into *llink* of new first node (i.e., *temp->llink*) as shown below:



The above activity can be achieved using the following code:

6 last->rlink = temp;

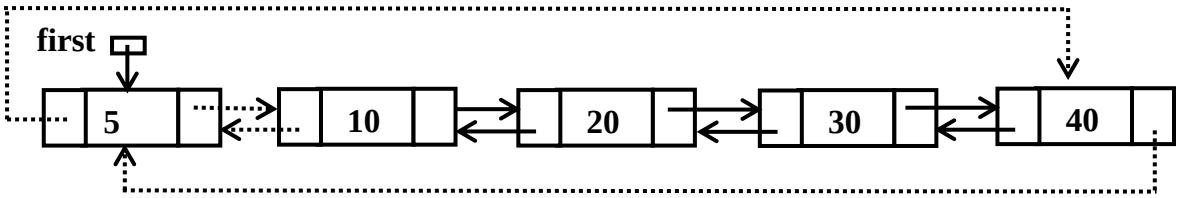
7 temp->llink = last;

Thus, a node is inserted at the front end of the list.

Step 5: Return the first node: After modifying the list, always we have to return the address of the first node. In the above list, *temp* is the first node and it can be returned using the following statement:

return temp;

Now, the list as seen from calling function can be written as shown below:



The C function to insert at the front end can be written as shown below:

Example 9.19: C Function to insert an item at the front end of the list

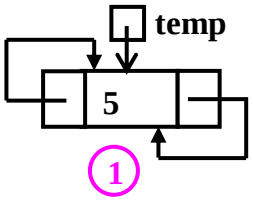
```
NODE insert_front(int item, NODE first)
{
        NODE temp, last;

        temp = getnode()
(1)     temp->info = item;
        temp->link = temp->rlink = temp;

(2)     if (first == NULL)  return temp;     /* create the node first time */

(3)     last = first->llink;                 /* Get the address of last node */

(4)     temp->rlink = first;                 /* Link the first node with new node*/
(5)     first->llink = temp;

(6)     last->rlink = temp;                  /* Link the last node with new node*/
(7)     temp->llink = last;

        return temp;
}
```

9.4.2 Insert a node at the rear end

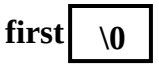
In this section, let us see "How to insert a node at the rear end of the list?"

Step 1: Create a node: This can be done using getnode() function and copying *item* 5 as shown below:

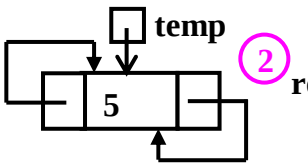
```
(1)     temp = getnode()
        temp->info = item;
        temp->link = temp->rlink = temp;
```



Step 2: Insert into empty list: If the list is empty, the above created node itself should be returned as the first node as shown below:



Initial list

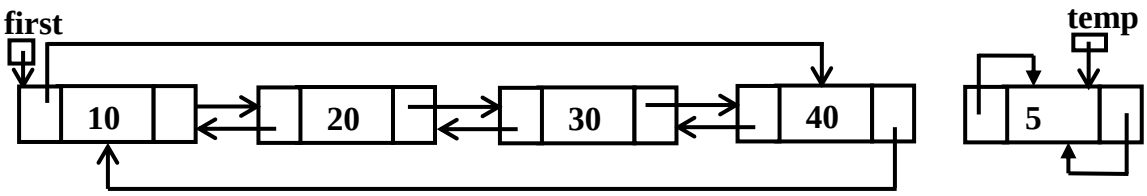


(2) **return** temp;

The code for this case can be written as shown below:

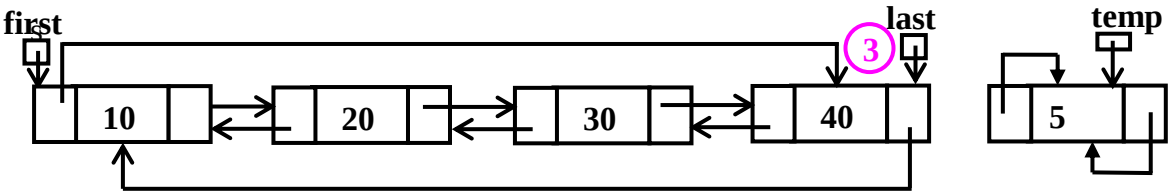
```
(2)  if (first == NULL)  return temp;
```

If the above condition fails, it means that list is already existing. The new node *temp* (created in step 1) which has to be inserted at the rear end and the existing list can be pictorially represented as shown below:



Now, the node *temp* can be inserted at the rear end using the following steps:

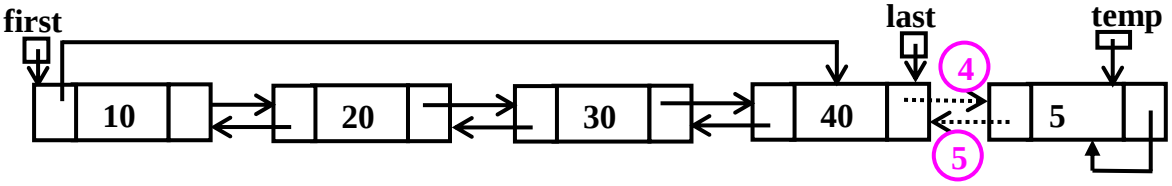
Step 3: Obtain the address of the last node: The last node of the list can be obtained using *llink* of the first node as shown below:



The code can be written as:

```
(3)  last = first->llink;          /* Get the address of last node */
```

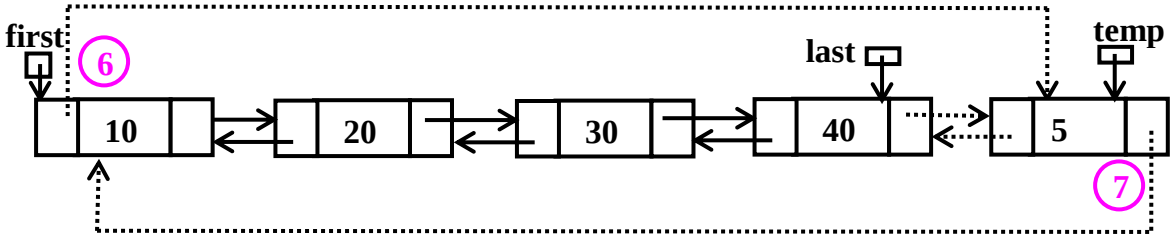
Step 4: Link the new node created with last node: This can be done by copying *temp* into *rlink* of *last* and copying *last* into *llink* of *temp* node as shown below:



The code can be written as shown below:

```
(4)  last->rlink = temp;

(5)  temp->llink = last;
```

Step 4: Make new node created as the last node: This can be done by copying *temp* to *llink* of first node and copying first node into *rlink* of *temp* node (i.e., *temp->rlink*) as shown below:



The above activity can be achieved using the following code:

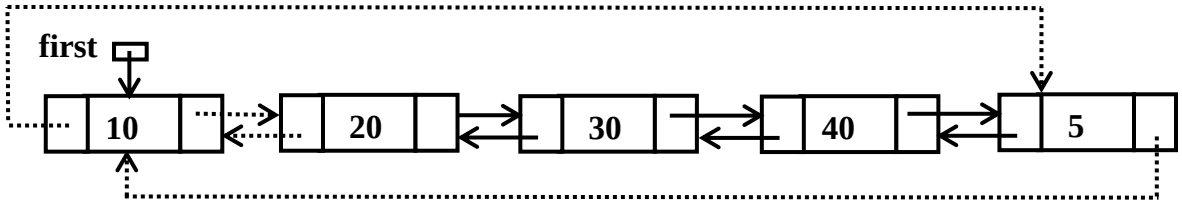
```
6   first->llink = temp;
7   temp->rlink = first;
```

Thus, a node is inserted at the rear end of the list.

Step 5: Return the first node: After modifying the list, always we have to return the address of the first node. It can be done using the following statement:

```
        return temp;
```

Now, the list as seen from calling function can be written as shown below:



The C function to insert at the rear end can be written as shown below:

Example 9.20: C Function to insert an item at the rear end of the list

```
NODE insert_rear(int item, NODE first)
{
        NODE temp, last;

        temp = getnode()
1       temp->info = item;
        temp->link = temp->rlink = temp;
```

```
(2)  if (first == NULL)  return temp;     /* Insert the node for the first time */
(3)  last = first->llink;                 /* Get the address of last node */
(4)  last->rlink = temp;                  /* Link the last node with new node*/
(5)  temp->llink = last;
(6)  first->llink = temp;                 /* Link the first node with new node*/
(7)  temp->rlink = first;

     return temp;
}
```

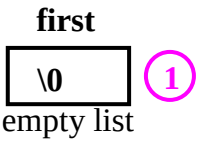
9.4.3 Delete a node from the front end

Now, let us see "How to delete a node from the front end of the list?"

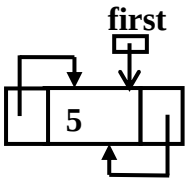
Design: A node from the front end of the list can be deleted by considering various cases as shown below:

Step 1: List is empty: If the list is empty, it is not possible to delete a node from the list. In such case, we display the message "List is empty" and **return** NULL. The code for this can be written as shown below:

```
(1)  if (first == NULL)
     {
          printf ("List is empty\n");
          return NULL;
     }
```

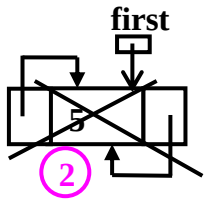


Step 2: Delete if there is only one node: A list having one node can be written as shown below:

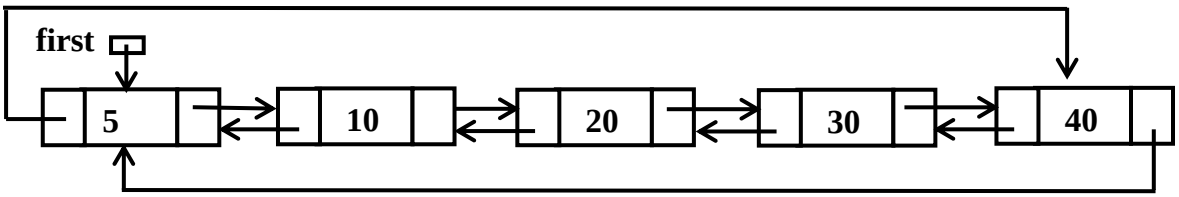


The code to delete the above node can be written as shown below:

```
if (first->rlink == first)
{
        printf("Item deleted = %d\n", first->info);
(2)     free(first);

        return NULL;
}
```

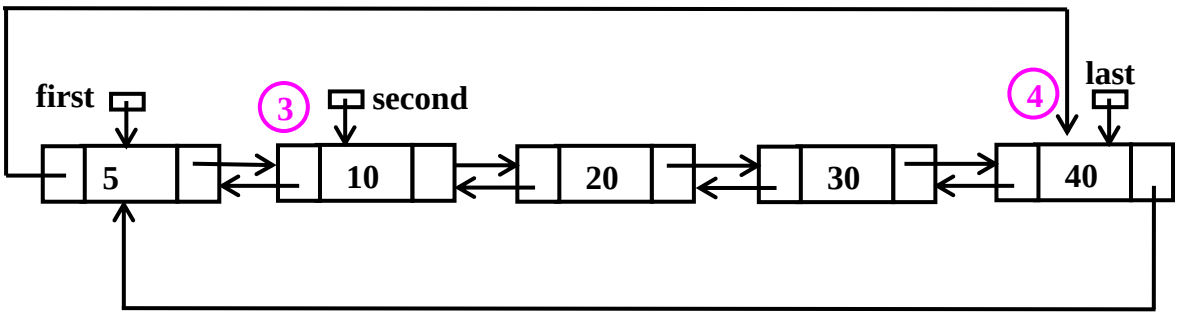


When control comes out of the above if-statement, it means that the list has more than one node and the list can be pictorially represented as shown below:



Now, the first node in the above list can be deleted as shown below:

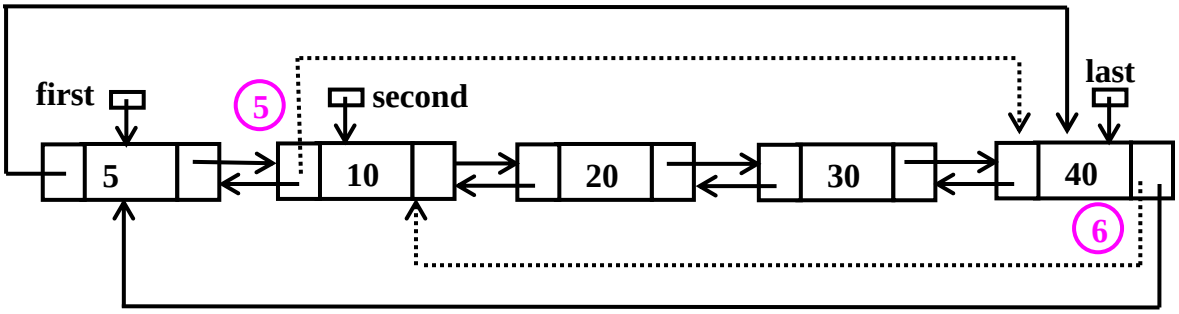
Step 3: Obtain the address of the second node and last node: The *rlink* of first node gives the second node and *llink* of first node gives the last node as shown below:



The corresponding code to find the address of the second node and last node can be written as shown below:

```
(3)  second = first->rlink;

(4)  last = first->llink;
```

Step 4:Make second node as first node: It is achieved by copying *last* to left link of *second* node and copying *second* to *rlink* of last node (dotted lines) as shown below:

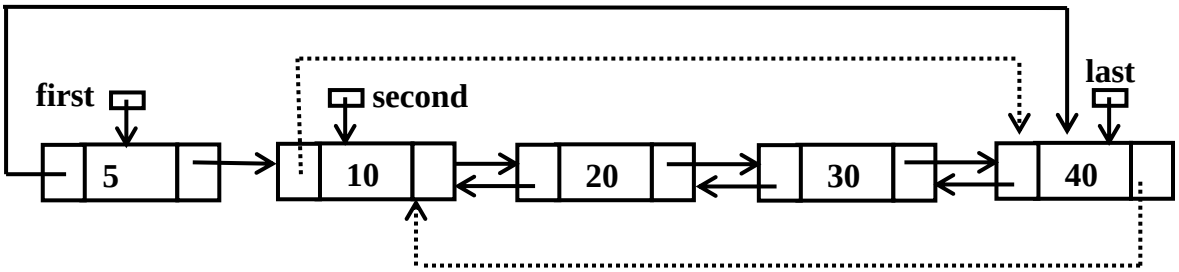


It can be done using the following code:

5 second->llink = last;

6 last->rlink = second;

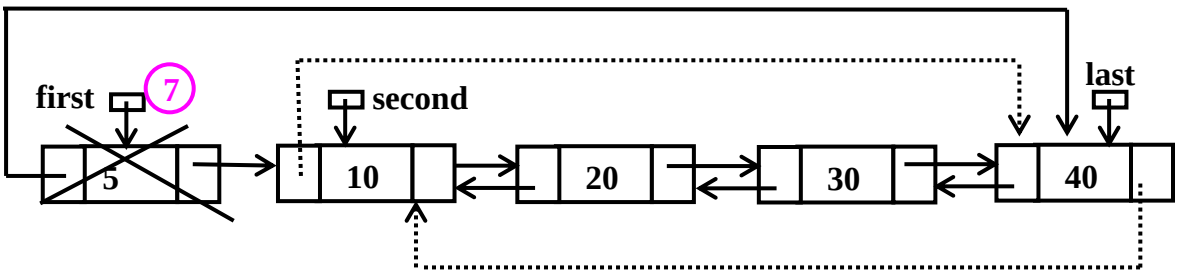
Now, *first* node is isolated and the resulting linked list is shown below:



Step 5: Delete the front node: Front node can be deleted using free() function. The code can be written as shown below:

```
7  printf("Item deleted = %d\n", first->info);
   free(first);
```

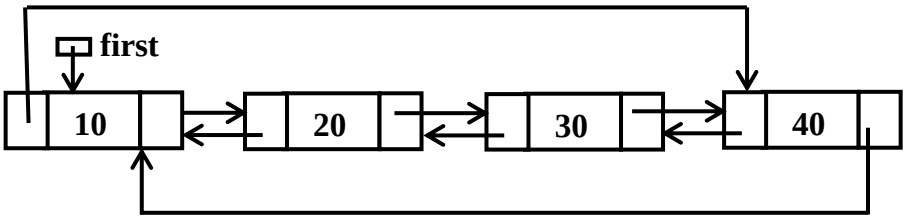
After executing the above code, the list can be written as shown below:



Step 6: Return address of the new first node: Note that the variable *second* now contains address of the new first node and it can be returned using the following statement:

return second;

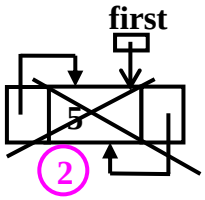
Now, the linked list as seen from the calling function can be written as shown below:



Now, the complete C function to delete the first node can be written as shown below:

Example 9.21: C function to delete an item from the front end of the list

```
NODE delete_front(NODE first)
{
        NODE second, last;

        if ( first == NULL )                    /* Check for empty list */
        {
(1)             printf("List is empty cannot delete\n");
                return NULL;                    // We can replace NULL with first also
        }

        if (first->rlink == first)              /* Delete if there is only one node */
        {
(2)             printf("Item deleted = %d\n", first->info);
                free(first);

                return NULL;
        }
(3)     second = first->rlink;                  /* obtain the address of second node */
(4)     last = first->llink;                    /* obtain the address of last node */
(5)     second->llink = last;                   /* Make second node as new first node */
(6)     last->rlink = second;
```



```
        /* Delete the old first node */
(7)     printf("Item deleted = %d\n", first->info);
        free(first);

        return second;                  /* Return second node as first node */
}
```

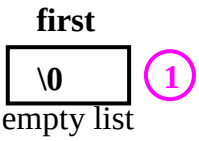
9.4.4 Delete a node from the rear end

Now, let us see "How to delete a node from the rear end of the list?"

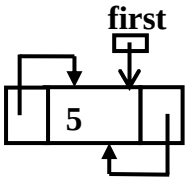
Design: A node from the rear end of the list can be deleted by considering various cases as shown below:

Step 1: List is empty: If the list is empty, it is not possible to delete a node from the list. In such case, we display the message "List is empty" and **return** NULL. The code for this can be written as shown below:

```
(1)     if (first == NULL)
        {
                printf ("List is empty\n");
                return NULL;
        }
```

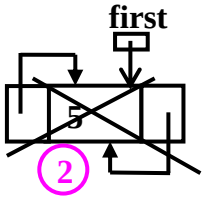


Step 2: Delete if there is only one node: A list having one node can be written as shown below:

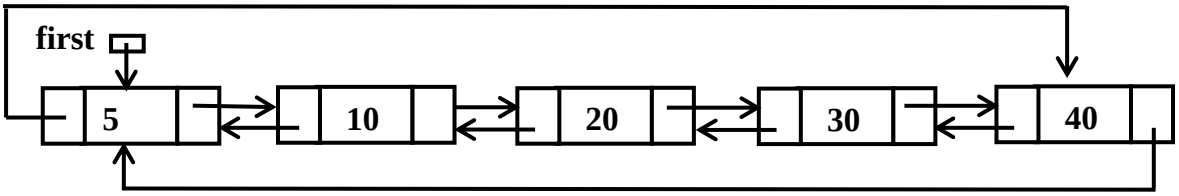


The code to delete the above node can be written as shown below:

```
        if (first->rlink == first)
        {
(2)             printf("Item deleted = %d\n", first->info);
                free(first);

                return NULL;
        }
```

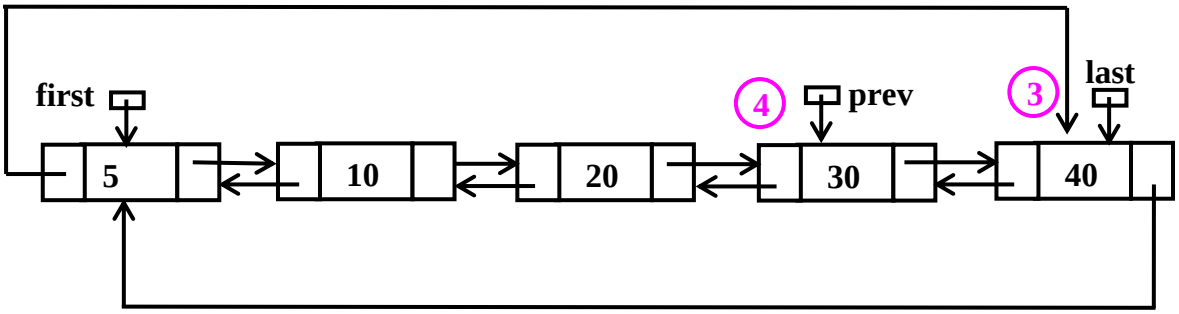


When control comes out of the above if-statement, it means that the list has more than one node and the list can be pictorially represented as shown below:



Now, the last node in the above list can be deleted as shown below:

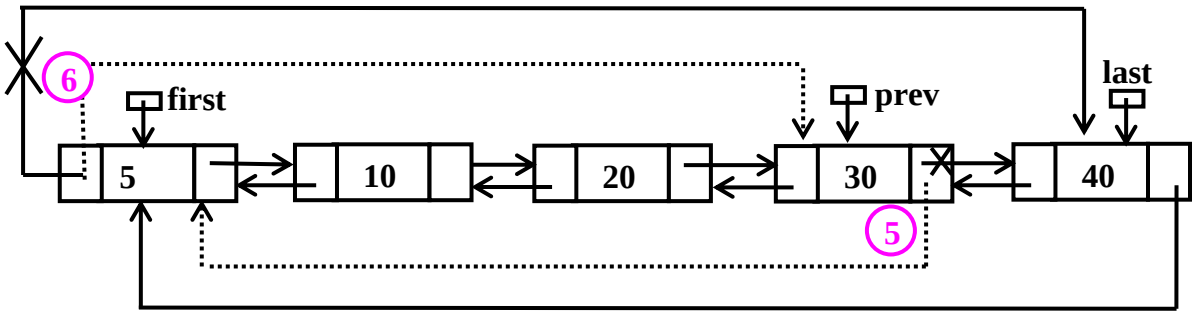
Step 3: Obtain the address of the last node and its predessor: The *llink* of first node gives the last node and *llink* of *last* gives its predessor as shown below:



The corresponding code to find the address of the last node and its predecessor can be written as shown below:

```
3  last = first->llink;
4  prev = last->llink;
```

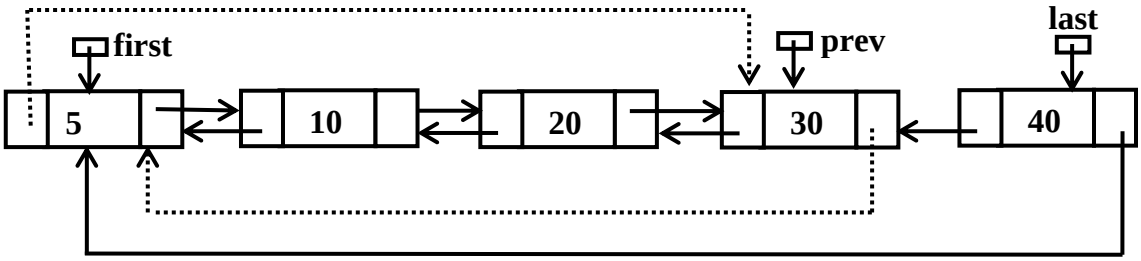
Step 4:Make last but one node as last node: It is achieved by copying *first* to right link of *prev* and copying *prev* to *llink* of first node (dotted lines) as shown below:



It can be done using the following code:

```
5  prev->rlink = first;
6  first->llink = prev;
```

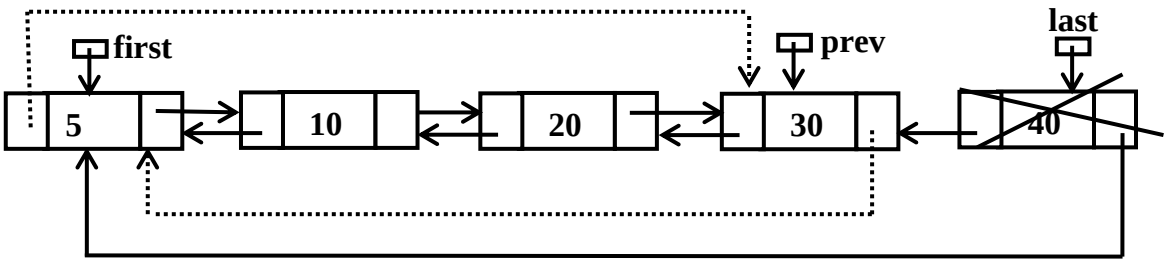
Now, *last* node is isolated and the resulting linked list is shown below:



Step 5: Delete the last node: Last node can be deleted using free() function. The code can be written as shown below:

```
7  printf("Item deleted = %d\n", last->info);
   free(last);
```

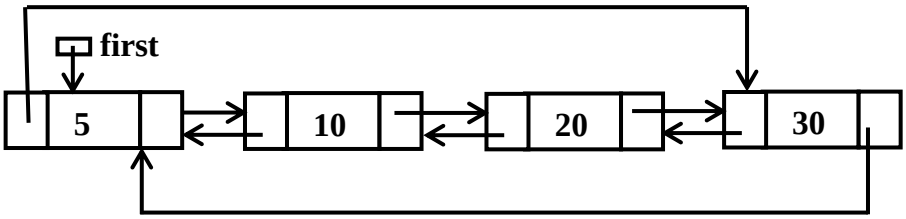
After executing the above code, the list can be written as shown below:



Step 6: Return address of the first node: The variable *first* contains address of the first node and it can be returned using the following statement:

```
return first;
```

Now, the linked list as seen from the calling function can be written as shown below:



Now, the complete C function to delete the last node can be written as shown below:

Example 9.22: Function to delete an item from the rear end of the list

```
NODE delete_rear(NODE first)
{
        NODE last, prev;
        .....
        if (first == NULL)              /* Check for empty list */
        {
                printf("List is empty cannot delete\n");
(1)             return first;
        }

        if ( first ->link == first )    /*Only one node is present and delete it */
        {
                printf ("The item to deleted is %d\n",first->info);
(2)             free(first);            /* return to availability list */

                return NULL;            /* List is empty so return NULL */
        }

(3)     last = first->llink;            /* obtain address of the last node */
(4)     prev = last->llink;             /* obtain address of last but one node */
(5)     prev->rlink = first;            /* Adjust pointers such that new last node */
(6)     first->llink = prev;            /* and first node are linked */

(7)     printf("Item deleted = %d\n", last->info);
        free(last);                     /* Delete the old last node */

        return first;                   /* return address of the first node */
}
```

9.4.5 Display circular doubly linked list

Keep traversing till we get the last node. Once we reach the last node come out of the loop and display the last node.

Example 9.23: C function to display the contents of linked list

```
void display(NODE first)
{
        NODE cur, last;

        if ( first == NULL )                    /* Check for empty list */
        {
                printf("List is empty\n");
                return;
        }

        printf("The contents of  singly linked list\n");
        cur = first;                            /* Holds address of the first node */
        last = first->llink;                    /* Obtain address of the last node */
        while ( cur != last )                   /* As long as no end of list */
        {
                printf("%d  ",cur->info);       /* Display the info field of node */
                cur = cur->rlink;               /* Point to the next node */
        }
        printf("%d  ",cur->info);       /* Display the info field of node */
}
```

9.4.6 Double ended queue using doubly linked lists

The C program to implement dequeue with the help of above functions can be written as shown below:

Example 9.24: Program to implement dequeues using circular doubly linked list

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
struct node
{
        int             info;
        struct node     *llink;
        struct node     *rlink;
};
typedef struct node* NODE;
```

```
/* Include: Example 8.2: Function to get a new node from the operating system */
/* Include: Example 9.19: Function to insert an item at the front end of the list */
/* Include: Example 9.20: Function to insert an item at the rear end of the list */
/* Include: Example 9.21: Function to delete an item from the front end of the list */
/* Include: Example 9.22: Function to delete an item from the rear end of the list */
/* Include: Example 9.23: Function to display the contents of linked list */

void main()
{
        NODE        first;
        int         choice, item;

        first  =  NULL;

        for (;;)
        {
                printf("1:Insert_Front    2: Insert_Rear\n");
                printf("3:Delete_Front    4: Delete_Rear\n");
                printf("5:Display         6: Exit\n");
                printf("Enter the choice\n");
                scanf("%d", &choice);

                switch(choice)
                {
                        case 1:
                                printf("Enter the item to be inserted\n");
                                scanf("%d", &item);
                                first = insert_front (item, first);
                                break;

                        case 2:
                                printf("Enter the item to be inserted\n");
                                scanf("%d", &item);
                                first = insert_rear (item, first);
                                break;

                        case 3:
                                first = delete_front(first);
                                break;
```

```
                    case 4:
                            first = delete_rear(first);
                            break;
                    case 5:
                            display(first);
                            break;
                    default:
                            exit(0);
                }
        }
}
```

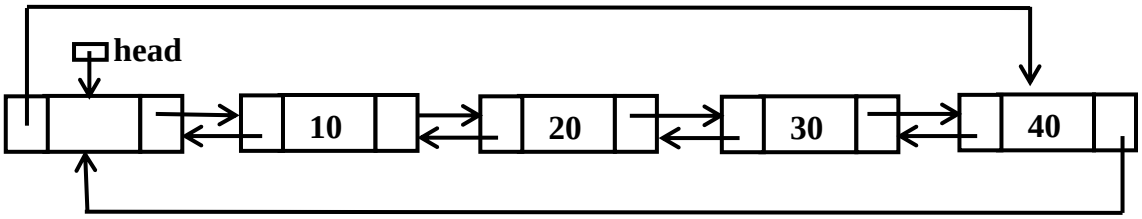
9.5 Circular doubly linked list with header node

Now, let us see "What is circular doubly linked list with a header?"

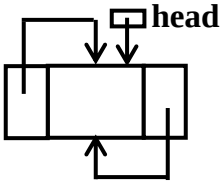
Definition: A circular doubly linked list (can also be called doubly linked circular list) is a variation of doubly linked with a header node in which:

- *llink* of header contains address of the last node of the list
- *rlink* of header contains address of the first node of the list.
- *llink* of last node contains the address of last but one node of the list
- *rlink* of last node contains the address of the header node of the list

The pictorial representation of circular doubly linked list with a header node is shown below.



This list is primarily used in structures that allow access to nodes in both the directions. An empty circular doubly linked list with a header node can be represented as shown in figure below where *llink* and *rlink* of a header node points to itself.

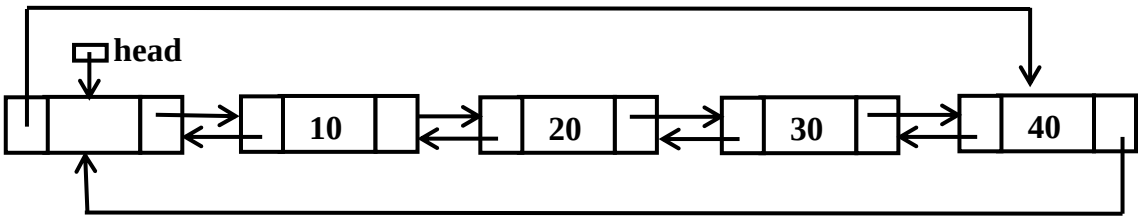


Note: Given any problem let us implement them using doubly linked lists and with a header node. Using a header node with circular doubly linked lists all the problems can be solved very easily and efficiently.

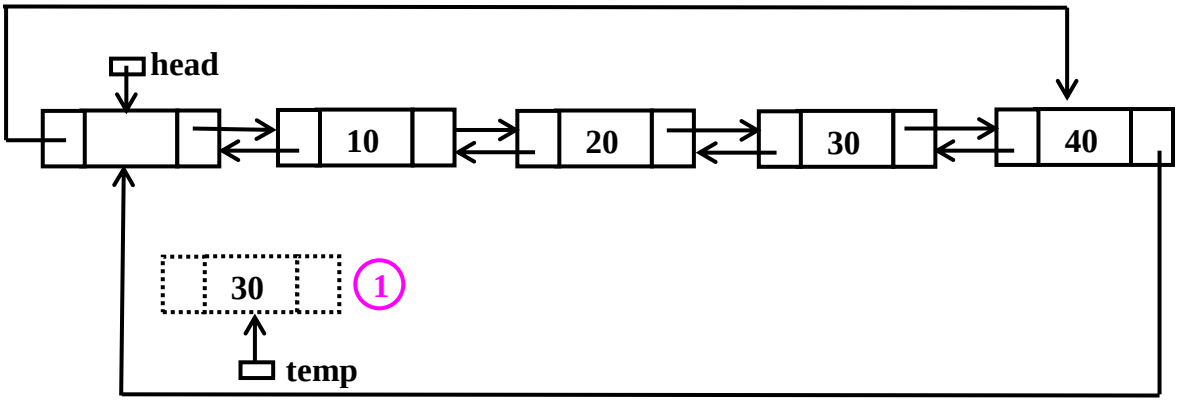
Note: Main advantage of using doubly linked circular list with a header node is that while designing a function we need not consider any extreme cases. Assume that list is existing and write a function to insert an item at the front end. The function works for all other cases (i.e., even if list is empty or if list has only one node or more than one node).

9.5.1 Insert a node at the front end

Now, let us see "How to insert an item at the front end of the list" Consider the list shown in figure below (represented using thick lines).



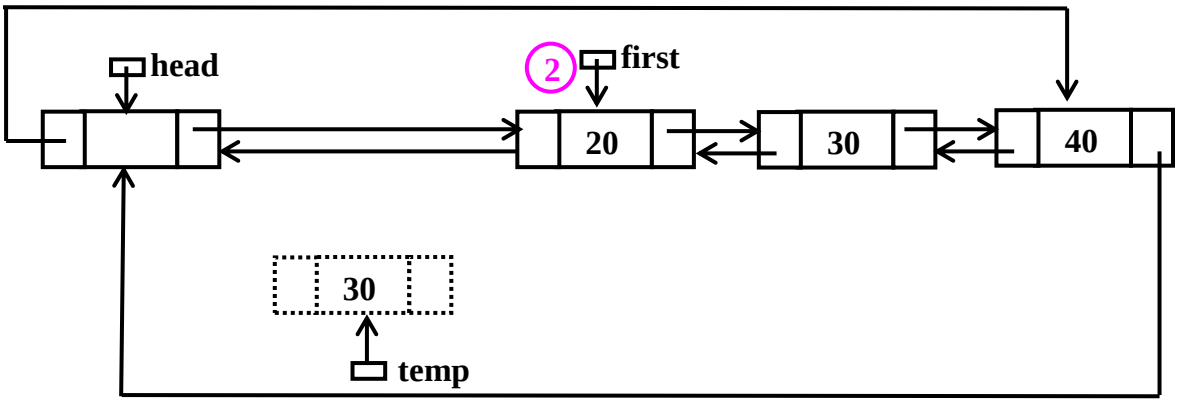
Step 1: Create a node and copy the item to be inserted: This can be pictorially represented as shown below:



The code for the above activity can be written as shown below:

```
1   temp = getnode();
    temp->info = item
```

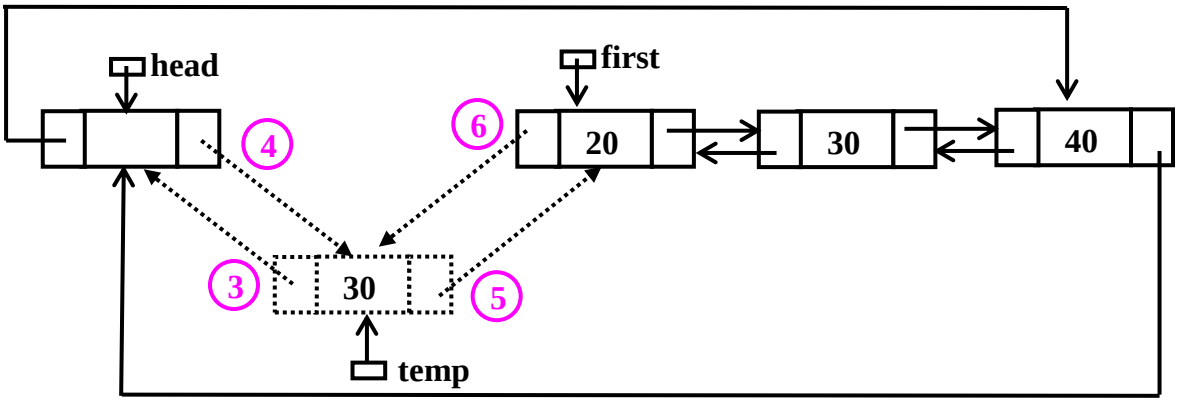
Step 2: Get the address of the first node: The *right link* of header node gives the address of the first node. Copying *rlink* of *head* to *first* as shown below:



The code corresponding to above figure can be written as:

```
2  first = head->rlink;          /* Get the address of the first node */
```

Step 3: Insert at the front end: The node *temp* has to be inserted between *head* and *first* as shown below:



The code corresponding to above figure can be written as:

```
3  temp->llink = head;           /* Insert temp after head */
4  head->rlink = temp;
5  temp->rlink = first;          /* Insert temp before first */
6  first->llink = temp;
```

Step 3: Return the header node: Using the header node, any node can be accessed. So, always we return the address of header node. This can be done using the statement:

```
return head;
```

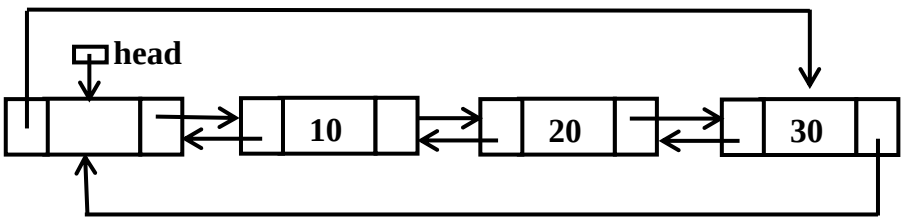
The complete function can be written as shown below:

Example 9.25: Function to insert a node at the front end of the list

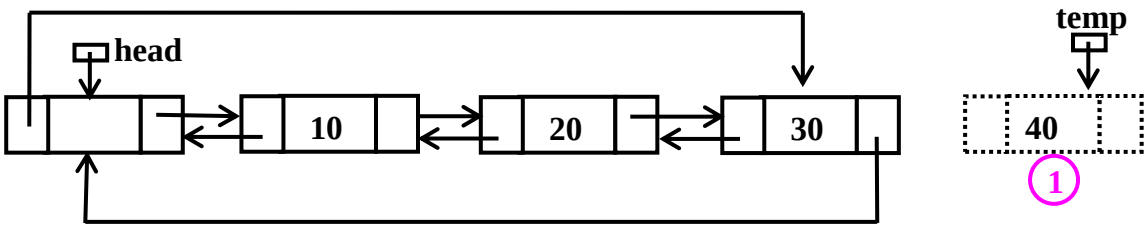
```
NODE insert_front(int item, NODE head)
{
        NODE temp, first;

(1)     temp = getnode();                   /* create a node temp */
        tem->info = item;

(2)     first = head->rlink;                /* Get the address of the first node */

        temp->llink = head;                 /* Insert temp after head */

(4)     head->rlink = temp;

(5)     temp->rlink = first;                /* Insert temp before first */

(6)     first->llink = temp;

        return head;                        /* return the header node */
}
```

9.5.2 Insert a node at the rear end

Now, let us see "How to insert an item at the rear end of the list" Consider the list shown in figure below:



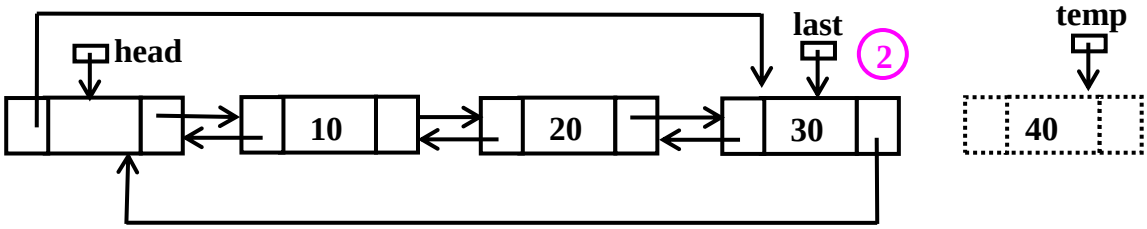
Step 1: Create a node and copy the item to be inserted: This can be pictorially represented as shown below:



The code for the above activity can be written as shown below:

```
1  temp = getnode();
   temp->info = item;
```

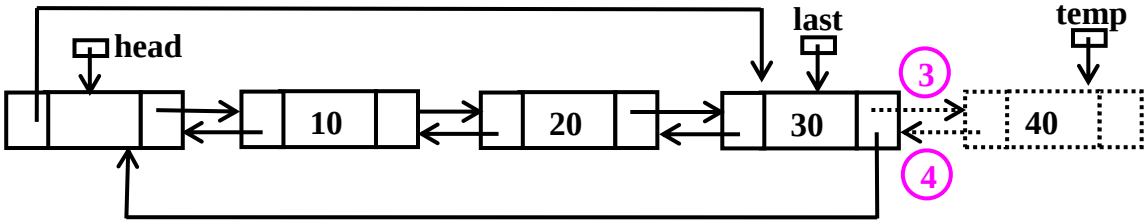
Step 2: Get the address of the last node: The *left link* of header node gives the address of the last node. Copying *llink* of *head* to *last* we get the list as shown below:



The code corresponding to above figure can be written as:

```
2  last = head->llink;          /* Get the address of the last node */
```

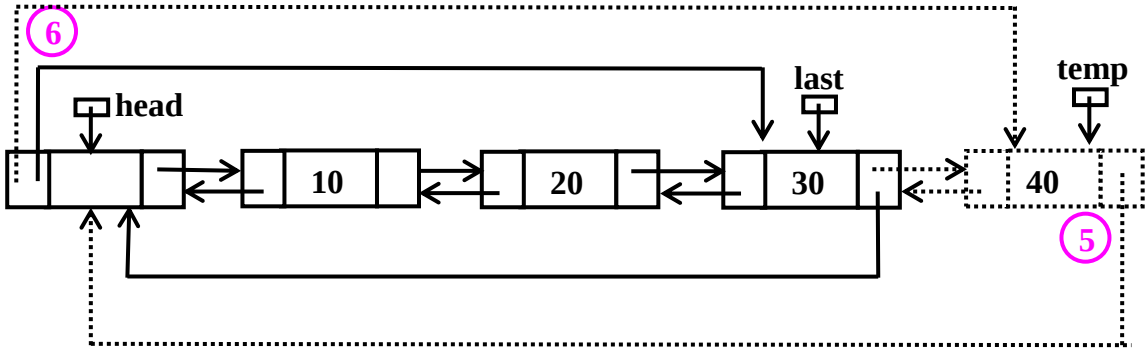
Step 3: Insert at the rear end: The node *temp* has to be inserted after *last* as shown below:



The code for the above situation can be written as shown below:

```
3  last->rlink = temp;
4  temp->llink = last;
```

Since *temp* is the last node, *rlink* of *temp* should contain address of header node and *llink* of *header* node should contain the address of last node i.e., *temp*. This can be pictorially represented as shown below:



The code for the above situation can be written as shown below:

5 temp->rlink = head;

6 head->llink = temp;

Step 4: Return the header node: Using the header node, any node can be accessed. So, always we return the address of header node. This can be done using the statement:

return head;

Now, the complete function can be written as shown below:

Example 9.26: Function to insert a node at the rear end of the list

```
NODE insert_rear(int item, NODE head)
{
        NODE temp, last;
1       temp = getnode();
        temp->info = item;
2       last = head->llink;              /* Get the address of the last node */
3       last->rlink = temp;              /* Insert temp at the end */
4       temp->llink = last;
5       temp->rlink = head;              /* Make temp as the last node */
6       head->llink = temp;
        return head;
}
```

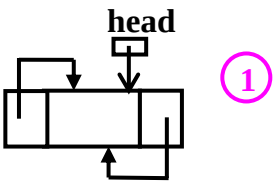
9.5.3 Delete a node from the front end

Now, let us see "How to delete a node from the front end of the list?"

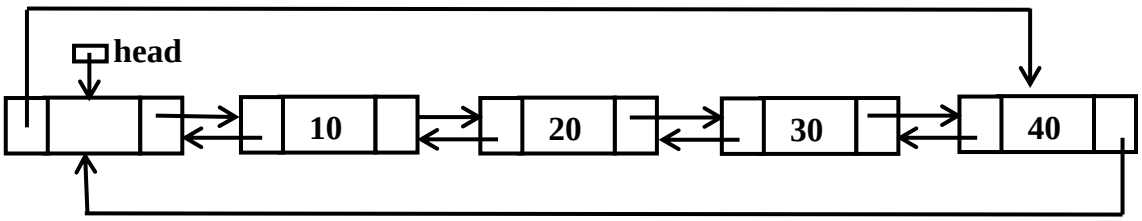
Design: A node from the front end of the list can be deleted by considering various cases as shown below:

Step 1: List is empty: If the list is empty, it is not possible to delete a node from the list. In such case, we display the message "List is empty" and **return** *head*. The code for this can be written as shown below:

```
(1)  if (head->rlink == head)
     {
          printf ("List is empty\n");
          return head;
     }
```

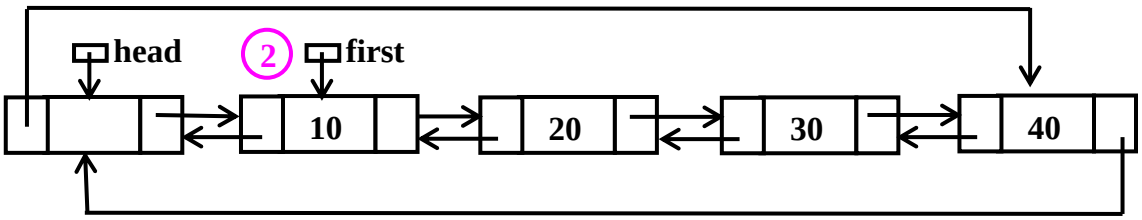


If the above condition fails, it means that list is existing. Consider the following list shown below:



We can delete an element from the front end using the following steps:

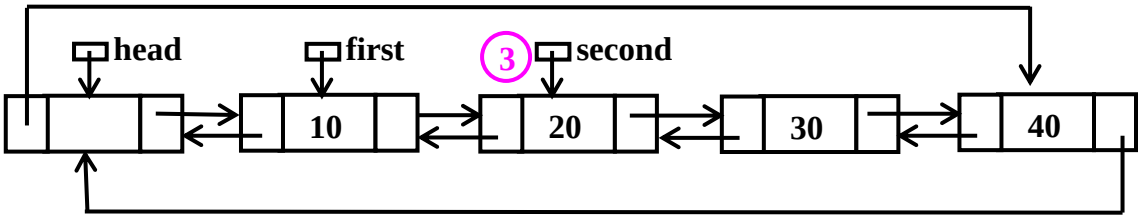
Step 2: Get the first node: The *right link* of *head* contains address of the first node and copy it into *first* as shown in figure below:



The code to obtain address of the first node can be written as shown below:

```
(2)  first = head->rlink;
```

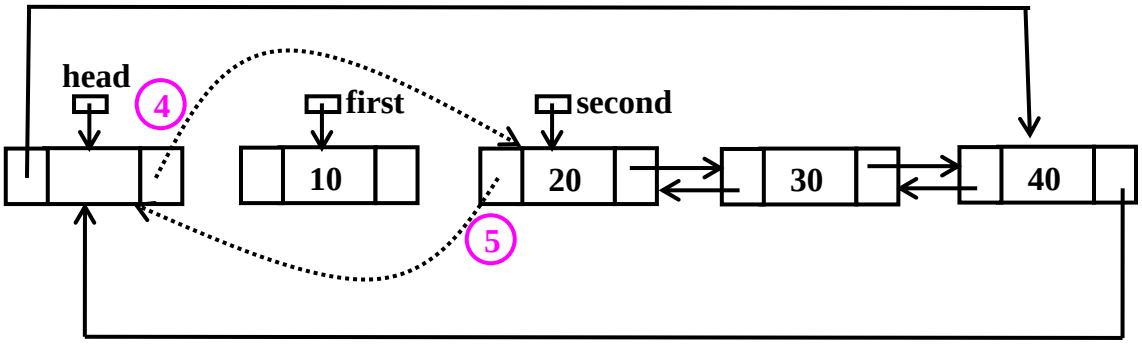
Step 3: Get the second node: The *right link* of *first* contains address of the second node and copy it into *second* as shown in figure below:



The code to obtain address of the second node can be written as shown below:

```
3  second = first->rlink;
```

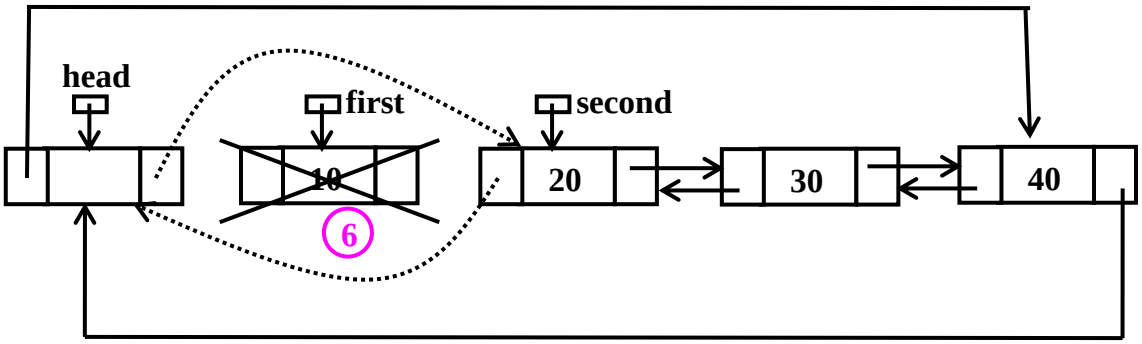
Step 4: Isolate the first node: This can be done by connecting *head* and *second* node as shown in figure below:



The corresponding code can be written as shown below:

```
4  head->rlink = second;
5  second->llink = head;
```

Step 5: Remove the first node: This first node can be removed as shown below:



The corresponding code can be written as shown below:

```
6   printf("Item deleted = %d\n", first->info);
    free(first);
```

Step 6: Return the header node: Using the header node, any node can be accessed. So, always we return the address of header node. This can be done using the statement:

```
return head;
```

Now, the complete function to delete an element from the front end can be written as shown below:

Example 9.27: Function to delete a node from the front end

```
NODE delete_front(NODE head)
{
        NODE first, second;

        if ( head->rlink == head )          /* Check for empty list */
        {
1           printf("Deque is empty\n");
            return head;
        }

2       first = head->rlink;                /* obtain the address of first node */

3       second = first->rlink;              /* obtain the address of second node */

4       head->rlink = second;               /* Link header  node with second node */

5       second->llink = head;

6       printf("Item deleted = %d\n", first->info);
        free(first);                        /* Delete the first node */

        return head;                        /* Return the address of the last node */
}
```

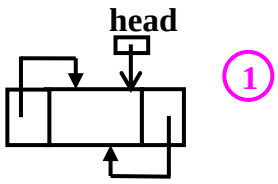
9.5.4 Delete a node from the rear end

Now, let us see "How to delete a node from the rear end of the list?"

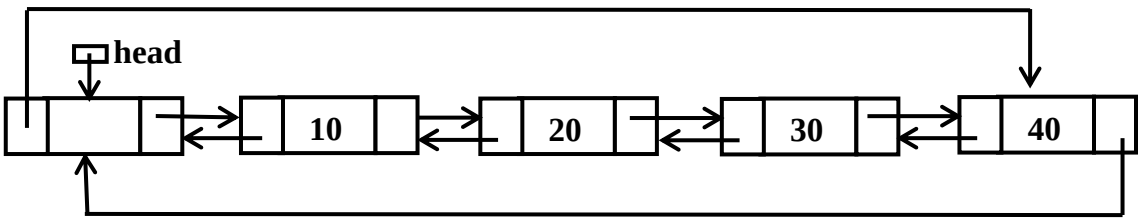
Design: A node from the rear end of the list can be deleted by considering various cases as shown below:

Step 1: List is empty: If the list is empty, it is not possible to delete a node from the list. In such case, we display the message "List is empty" and **return** *head*. The code for this can be written as shown below:

```
if (head->rlink == head)
{
        printf ("List is empty\n");
        return head;
}
```

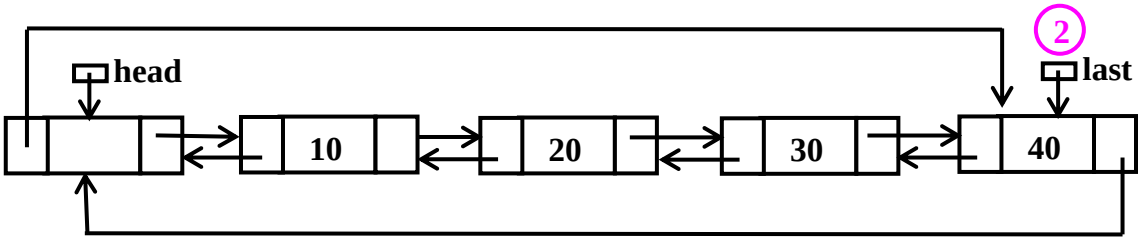


If the above condition fails, it means that list is existing. Consider the following list shown below:



We can delete an element from the rear end using the following steps:

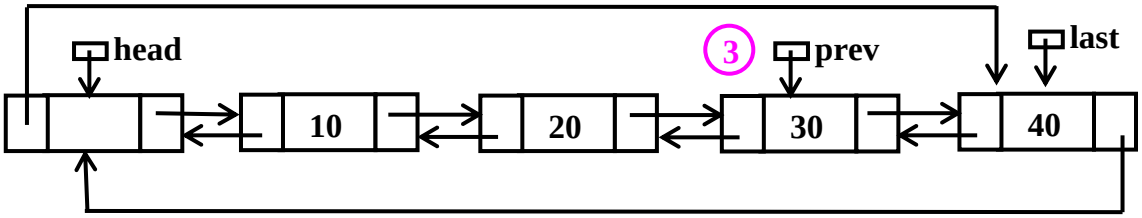
Step 2: Get the last node: The *llink* of *head* contains address of the last node and copy it into *last* as shown in figure below:



The code to obtain address of the last node can be written as shown below:

```
2  last = head->llink;
```

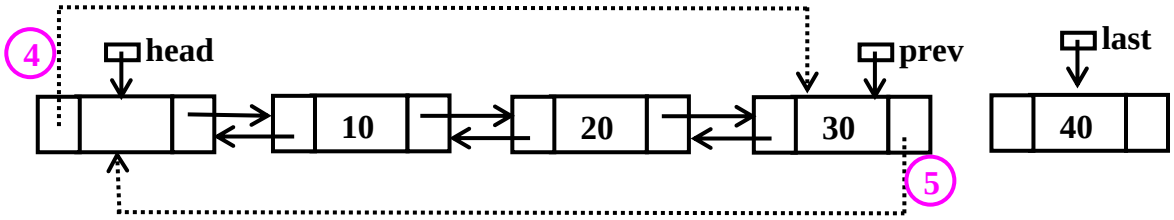
Step 3: Get the last but one node: The *left link* of *last* contains address of the last but one node and copy it into *prev* as shown in figure below:



The code to obtain address of the last but one node can be written as shown below:

```
3  prev = last->llink;
```

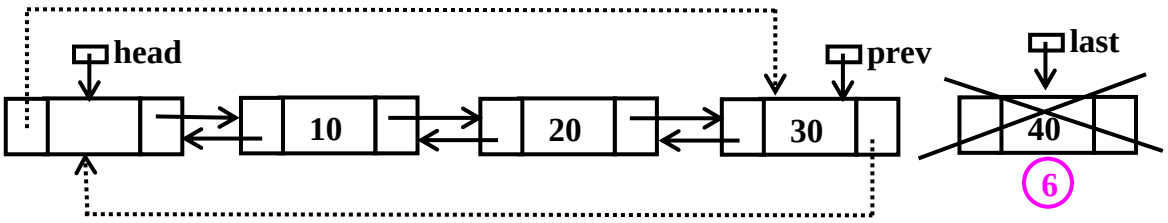
Step 4: Isolate the last node: This can be done by connecting *head* and *prev* node as shown in figure below:



The corresponding code can be written as shown below:

```
4  head->llink = prev;
5  prev->rlink = head;
```

Step 5: Remove the last node: This last node can be removed as shown below:



The corresponding code can be written as shown below:

```
6  printf("Item deleted = %d\n", last->info);
   free(last);
```

Step 6: Return the header node: Using the header node, any node can be accessed. So, always we return the address of header node. This can be done using the statement:

```
return head;
```

Now, the complete function to delete an element from the rear end can be written as shown below:

Example 9.28: Function to delete a node from the rear end

```
NODE delete_rear(NODE head)
{
        NODE last, prev;

        if ( head->rlink == head )          /* Check for empty list */
        {
(1)             printf("Deque is empty\n");
                return head;
        }

(2)     last = head->llink;                 /* obtain address of last node */

(3)     prev = last->llink;                 /* obtain address of last but one node */

(4)     head->llink = prev;                 /* Isolate the last node */

(5)     prev->rlink = head;

(6)     printf("Item deleted = %d\n", last->info);
        free(last);                         /* Delete the last node */

        return head;                        /* Return the address of the last node */
}
```

9.5.5 Display circular doubly linked list with header node

Start displaying from the first node till we get the header node. The code for this can be written as shown below:

Example 9.29: Display the contents of circular doubly linked list with header node

```
void display(NODE head)
{
        NODE cur;

        if ( head->rlink == head )              /* Check for empty list */
        {
                printf("Deque is empty\n");
                return head;
        }
        printf("Contents of doubly linked list\n");
        cur = head->rlink;

        while (cur != head)
        {
                printf("%d\n", cur->info);
                cur = cur->rlink;
        }
}
```

9.5.6 Double ended queue using circular doubly linked lists with header node

The C program to implement dequeue with the help of above functions can be written as shown below:

Example 9.30: Program to implement dequeues using circular doubly linked list

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
struct node
{
        int             info;
        struct node     *llink;
        struct node     *rlink;
};

typedef struct node* NODE;
```

```
/* Include: Example 8.2: Function to get a new node from the operating system */
/* Include: Example 9.25: Function to insert an item at the front end of the list */
/* Include: Example 9.26: Function to insert an item at the rear end of the list */
/* Include: Example 9.27: Function to delete an item from the front end of the list */
/* Include: Example 9.28: Function to delete an item from the rear end of the list */
/* Include: Example 9.29: Function to display the contents of linked list */

void main()
{
        NODE        first;
        int         choice, item;

        head  =  getnode();
        head->rlink = head->llink = head;

        for (;;)
        {
                printf("1:Insert_Front    2: Insert_Rear\n");
                printf("3:Delete_Front    4: Delete_Rear\n");
                printf("5:Display         6: Exit\n");
                printf("Enter the choice\n");
                scanf("%d", &choice);

                switch(choice)
                {
                        case 1:
                                printf("Enter the item to be inserted\n");
                                scanf("%d", &item);
                                head = insert_front (item, head);
                                break;
                        case 2:
                                printf("Enter the item to be inserted\n");
                                scanf("%d", &item);
                                head = insert_rear (item, head);
                                break;
                        case 3:
                                head = delete_front(head);
                                break;
                        case 4:
                                head = delete_rear(head);
                                break;
```

```
                    case 5:
                            display(head);
                            break;
                    default:
                            exit(0);
                }
        }
}
```

9.5.7 Storing and deleting employee details

Let us "Design, Develop and Implement a menu driven Program in C for the following operations on **Doubly Linked List (DLL)** of Employee Data with the fields: ***SSN, Name, Dept, Designation, Sal, PhNo*****"**

1. Create a **DLL** of **N** Employees Data by using ***end insertion***.
2. Display the status of **DLL** and count the number of nodes in it
3. Perform Insertion and Deletion at End of **DLL**
4. Perform Insertion and Deletion at Front of **DLL**
5. Demonstrate how this **DLL** can be used as **Double Ended Queue**

A structure to hold the employee details can be written as shown below:

```
typedef struct
{
        int         ssn;
        char        name[20];
        char        department[20];
        char        designation[20];
        float       salary;
        char        phone[20];
} EMPLOYEE;
```

The design details are given in section 9.3. Creating a list is nothing but repeated insertion of items into the list. The insert front function (example 9.13) and insert rear function (example 9.14) are modified to incorporate the above details.

The C function to insert a node at the front end can be written as shown below:

Example 9.31: C Function to insert an item at the front end of the list

```
NODE insert_front(EMPLOYEE item, NODE first)
{
        NODE temp;

        temp = getnode();                   /*obtain a node from OS */
        temp->ssn = item.ssn;               /* Insert various items into new node */
        strcpy(temp->name, item.name);
        strcpy(temp->department, item.department);
        strcpy(temp->designation, item.designation);
        temp->salary = item.salary;
        strcpy(temp->phone, item.phone);
        temp->llink = temp->rlink = NULL;

        if (first == NULL)  return temp;    /* Insert a node for the first time */

        temp->rlink = first;                /* Insert at the beginning of existing list */
        first->llink = temp;

        return temp;                        /* return address of new first node */
}
```

The C function to insert a node at the rear end can be written as shown below:

Example 9.32: C Function to insert various items at the rear end of the list

```
NODE insert_rear(int item, NODE first)
{
        NODE temp, cur;

        temp = getnode();                   /*obtain a node from OS */
        temp->ssn = item.ssn;               /* Insert various items into new node */
        strcpy(temp->name, item.name);
        strcpy(temp->department, item.department);
        strcpy(temp->designation, item.designation);
        temp->salary = item.salary;
        strcpy(temp->phone, item.phone);
        temp->llink = temp->rlink = NULL;

        if (first == NULL)  return temp;    /* Insert a node for the first time */
```

```
        /* Get the address of the first node */
        cur = first;

        /* Find the address of the last node */
        while (cur->rlink != NULL)
        {
                cur = cur->rlink;
        }

        /* Insert the node at the end */
        cur->rlink = temp;
        temp->llink = cur;

        /* return address of the first node */
        return first;
}
```

The C function given in example 9.15 to delete an item from the front end of the list can be modified as shown below:

Example 9.33: C function to delete an item from the front end of the list

```
NODE delete_front(NODE first)
{
        NODE second;
        if ( first == NULL )                    /* Check for empty list */
        {
                printf("employee list is empty \n");
                return NULL;                    // We can replace NULL with first also
        }

        if (first ->rlink == NULL)              /* Delete if there is only one node */
        {
                printf("Employee details deleted: ssn: %d = %d\n", first->ssn);
                free(first);

                return NULL;
        }

        second = first->rlink;                  /* Get the address of second node */
        second->llink = NULL;                   /* Make second node as the first node */
```

```
        printf("Employee details deleted: ssn: %d = %d\n", first->ssn);
        free(first);                            /* Delete the first node */

        return second;
}
```

The C function given in example 9.16 to delete an item from the front end of the list can be modified as shown below:

Example 9.34: C function to delete an item from the rear end of the list

```
NODE delete_rear(NODE first)
{
        NODE cur, prev;

        if (first == NULL)              /* Check for empty list */
        {
                printf("List is empty cannot delete\n");
                return first;
        }


        if ( first ->link == NULL )    /*Only one node is present and delete it */
        {
                printf("Employee details deleted: ssn: %d = %d\n", first->ssn);
                free(first);            /* return to availability list */
                return NULL;            /* List is empty so return NULL */
        }

        /* Obtain address of the last node and just previous to that */
        prev = NULL;
        cur = first;

        while( cur->link != NULL )
        {
                prev = cur;
                cur = cur->link;
        }
```

```
        printf("Employee details deleted: ssn: %d = %d\n", cur->ssn);
        free(cur);                      /* delete the last node */

        prev->rlink = NULL;             /* Make last but one node as the last node */

        return first;                   /* return address of the first node */
}
```

The function to display the employee details are shown below:

Example 9.35: Function to display employee details

```
void display(NODE first)
{
        NODE temp;  int count = 0;

        if (first == NULL)                      /* List is empty */
        {
                printf("employee list is empty\n");
                return;
        }

        /* Display employee details */
        cur = first;
        while (cur != NULL)
        {
                printf("%d %f %s %s %s %s\n", cur->ssn, cur->salary, cur->name,
                        cur->department, cur->designation, cur->phone);
                cur = cur->rlink;   count++;
        }
        printf("Number of employees = %d\n", count);
}
```

The complete program can be written as shown below:

Example 9.36: Program to implement dequeue using doubly linked list (employee)

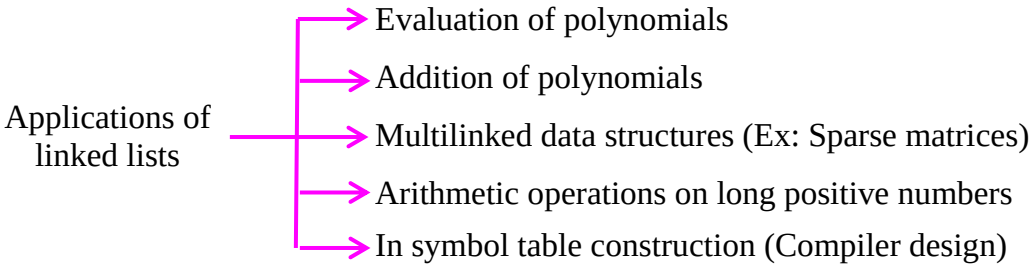
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <alloc.h>
```

```
typedef struct
{
        int             ssn;
        char            name[20];
        char            department[20];
        char            designation[20];
        float           salary;
        char            phone[20];
} EMPLOYEE;

struct node
{
        int             ssn;
        char            name[20];
        char            department[20];
        char            designation[20];
        float           salary;
        char            phone[20];
        struct node     *llink;
        struct node     *rlink;
};

typedef struct node* NODE;

/* Include: Example 8.2: Function to get a new node from the operating system */
/* Include: Example 9.31: Function to insert an item at the front end of the list */
/* Include: Example 9.32: Function to insert an item at the rear end of the list */
/* Include: Example 9.33: Function to delete an item from the front end of the list */
/* Include: Example 9.34: Function to delete an item from the rear end of the list */
/* Include: Example 9.35: Function to display the contents of linked list */

void main()
{
        NODE            first;
        int             choice;
        EMPLOYEE        item;

        first = NULL;
```

```
        for (;;)
        {
                printf("1:Insert_Front    2: Insert_Rear\n");
                printf("3:Delete_Front   4: Delete_Rear\n");
                printf("5:Display          6: Exit\n");
                printf("Enter the choice\n");
                scanf("%d", &choice);
                switch(choice)
                {
                        case 1:
                                printf("ssn             :"); scanf("%d",&item.ssn);
                                printf("name            :"); scanf("%s",item.name);
                                printf("department      :"); scanf("%s",item.department);
                                printf("designation     :"); scanf("%s",item.designation);
                                printf("salary          :"); scanf("%f",&item.salary);
                                printf("phone           :"); scanf("%s",item.phone);
                                first = insert_front (item, first);
                                break;
                        case 2:
                                printf("ssn             :"); scanf("%d",&item.ssn);
                                printf("name            :"); scanf("%s",item.name);
                                printf("department      :"); scanf("%s",item.department);
                                printf("designation     :"); scanf("%s",item.designation);
                                printf("salary          :"); scanf("%f",&item.salary);
                                printf("phone           :"); scanf("%s",item.phone);
                                first = insert_rear (item, first);
                                break;
                        case 3:
                                first = delete_front(first);
                                break;
                        case 4:
                                first = delete_rear(first);
                                break;
                        case 5: display(first);
                                break;
                        default: exit(0);
                }
        }
}
```

9.6 Application of linked lists

In this section, let us see "What are the applications of linked lists?" The various applications of linked lists are shown below:



9.6.1 Polynomial representation and evaluation

Before evaluating, let us see "How to represent a polynomial using linked list?" Let us assume we have a polynomial consisting of three variables **x, y** and **z** as shown below:

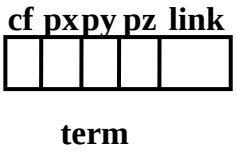
$$5x^3y^2y^2 + 3y^2 + 4x^3z - 5$$

where 5, 3, 4 and -5 are coefficients. The above polynomial can be easily represented using a linked list where each term can be considered as a node consisting of five fields as shown below:

- **cf** - to store the coefficient
- **px** – to store the power of x
- **py** – to store the power of y
- **pz** – sto store the power of z
- **link** – which contains the address of the next term(node).

The declaration for a **node** representing a term can be written as shown below:

```
struct node
{
        int             cf, px, py, pz;
        struct node     *link;
};
```



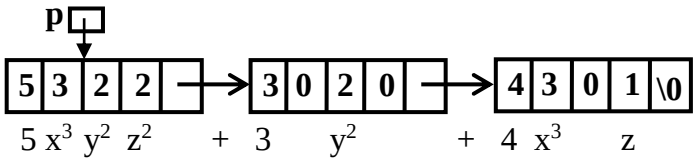
term

typedef struct node* NODE;

Using the above structure declaration, the polynomial:

$$5x^3y^2z^2 + 3y^2 + 4x^3z$$

can be represented using singly linked list as shown below:

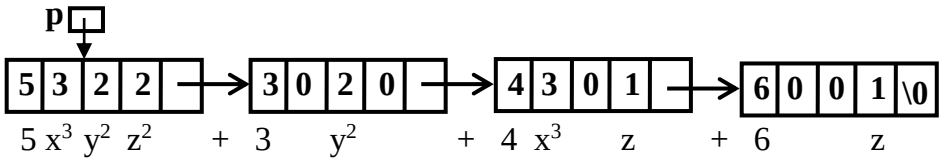


Since there are three terms in the above polynomial, the above polynomial is represented as a linked list consisting of three nodes. To design the algorithm, for the sake of simplicity we assume each term in the polynomial is unique i.e., each term has different powers of x, y and z. Each term in the polynomial can have the same or different coefficient.

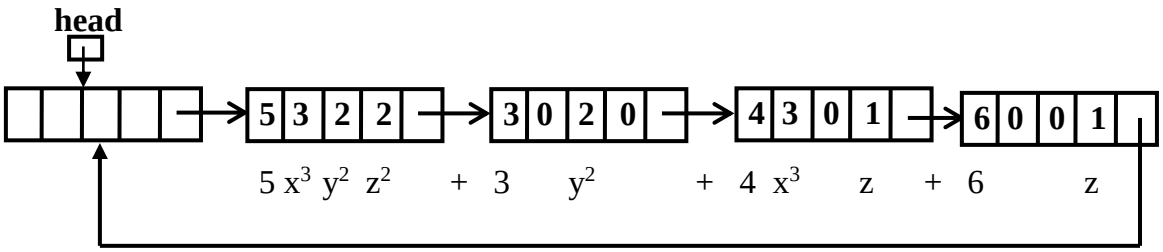
Now, to get the following polynomial:

$$5x^3y^2z^2 + 3y^2 + 4x^3z + 6z$$

we need to *add the node representing the term 6z at the end of the list above list.* The resulting list representing the above polynomial can be written as shown below:



The above list can be represented using circular singly linked list with a header node as shown below:



Thus the polynomial can be represented using circular singly linked list with a header node by repeatedly inserting each term at the end of the polynomial.

So, let us write the function to insert at the rear end of the list (for the detailed design see section 9.2.2). The function given in example 9.8 can be modified to incorporate *coefficient*, power of *x*, power of *y* and power of *z* as shown below:

Example 9.41: Function to insert at the rear end of the list

```
NODE insert_rear(int cf, int px, int py, int pz, NODE head)
{
        NODE temp, cur;

        temp = getnode();                   /* create a node */

        temp->cf = cf;                      /* insert coefficient */
        temp->px = px;                       /* insert power of x */
        temp->py = py;                       /* insert power of y */
        temp->pz = pz;                       /* insert power of z */

        cur = head->link;                   /* obtain the address of the first node */

        while (cur->link != head)           /* obtain the address of last node */
        {
                cur = cur->link;
        }

        cur->link = temp;                   /* insert the node at the end */
        temp->link = head;

        return head;
}
```

Now, let us see "How to read a polynomial consisting of *n* terms?" We can enter *n* terms using the scanf() statement. As we read each term representing *coefficient*, power of *x*, power of *y* and power of *z*, we insert at the rear end of the list as shown below:

Example 9.42: Function to read a polynomial

```
NODE read_poly(NODE head)
{
        int     i, n;
        int     cf, px, py, pz;         /* To hold term of a polynomial */

        printf("Enter the number of terms in the polynomial:");
        scanf("%d", &n);
```

```
        for(i = 1; i <= n; i++)
        {
                printf("Enter term: %d\n", i);   printf("Cf px py pz = ");
                scanf("%d %d %d %d", &cf, &px, &py, &pz);     /* Enter each term */

                head = insert_rear (cf, px, py, pz, head);          /* insert at the end */
        }
        return head;
}
```

Now, let us see "How to display a polynomial represented as a linked list?" The function to display the polynomial represented as a circular linked list with a header node is (for detailed design see section 9.2.5) shown below:

Example 9.43: Function to display a polynomial

```
void display(NODE head)
{
        NODE temp;

        if ( head->link == head )
        {
                printf("Polynomial does not exist\n");
                return;
        }

        temp = head->link;
        while (temp != head)
        {
                if (temp->cf < 0)       /* Print –ve coefficient */
                        printf("%d", temp->cf);
                else                    /* Print +ve coefficient */
                        printf("+%d", temp->cf);

                if (temp->px != 0) printf("x^%d", temp->px);
                if (temp->py != 0) printf("y^%d", temp->py);
                if (temp->pz != 0) printf("z^%d", temp->pz);

                temp = temp->link;
        }
        printf("\n");
}
```

Now, let us see "How to evaluate a polynomial?" To evaluate a polynomial, we should know the values of the variables **x, y** and **z.** Once these values are known, substitute these values for the corresponding variables in each term and by adding all the terms, the result is obtained. The function to evaluate the polynomial is shown below:

Example 9.44: Function to evaluate a polynomial with two variables

```
float evaluate(NODE head)
{
        int         x, y, z;
        float       sum = 0;
        NODE        p;

        printf("Enter the value of x, y and z\n");
        scanf("%d %d %d", &x, &y, &z);

        p = head->link;              /* Access each term, substitute x, y and z */
        while (p != head )
        {
                sum += p->cf * pow(x, p->px) * pow(y, p->py) * pow(z, p->pz);
                p = p->link;
        }
        return sum;
}
```

The main program to evaluate the polynomial is shown below:

Example 9.45: C Program to evaluate a given polynomial

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

struct node
{
        int         cf;
        int         px,  py,  pz;
        struct node *link;
};

typedef struct node* NODE;
```

```
/* Include: Example 8.2: Function to get new node from the availability list */
/* Include: Example 9.41: Function to insert a term at the rear end */
/* Include: Example 9.42: Function to read a polynomial */
/* Include: Example 9.43: Function to display a polynomial  */
/* Include: Example 9.44: Function to evaluate a polynomial with three variables  */

void main()
{
        NODE head;
        float res;

        head = getnode();
        head->link = head;

        printf("Enter the polynomial\n");
        head = read_poly(head);

        res = evaluate(head);

        printf("The given polynomial is\n");
        display(head);

        printf("The result = %f\n", res);
}
```

Now, let us evaluate the polynomial: $\mathbf{6x^2y^2z - 4yz^5 + 3x^3yz + 2xy^5z - 2xyz^3}$
where **x = 1, y = 2, z = 3**

Output

```
Enter the number of terms in the polynomial: 5

Enter term: 1
Cf px py pz = 6 2 2 1

Enter term: 2
Cf px py pz =  -4  0 1  5

Enter term: 3
Cf px py pz =  3 3 1 1

Enter term: 4
Cf px py pz =  2 1 5 1

Enter term: 5
Cf px py pz =  -2 1 1 3
```

Enter the value of x, y and z
1 2 3

The given polynomial is
+6 x^2 y^2 z^1 – 4 y^1 z^5 + 3 x^3 y^1 z^1 + 2 x^1 y^5 z^1 – 2 x^1 y^1 z^3

The result = -1770

9.6.2 Addition of two polynomials

Once we know how to read, display and evaluate a polynomial, now the next question is "How to add two polynomials?"

Design: Let use circular list with a header node to represent a polynomial as discussed in previous section.

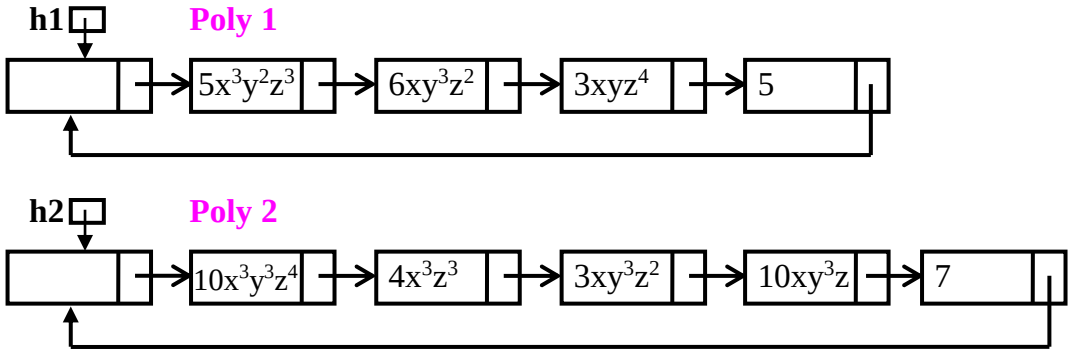
- The variable **h1** contains address of the header node of the first polynomial.
- The variable h**2** contains address of the header node of the second polynomial

Consider the following two polynomials:

Poly 1: $5x^3y^2z^3 + 6xy^3z^2 + 3xyz^4 + 5$ // h1 points to the first polynomial

Poly 2: $10x^3y^3z^4 + 4x^3z^3 + 3xy^3z^2 + 10xy^3z + 7$ // h2 points to the second polynomial

Pictorially, the above two lists can be represented using circular singly linked list with header node as shown below:



If we want to add two polynomials, first we have to search for power of polynomial 1 in polynomial 2. This can be done using linear search as shown below:

Example 9.46: Function to search for term of poly 1 in poly 2

```
NODE search(NODE p1, NODE h2)
{
        int     cf1, px1, py1, pz1, cf2, px2, py2, pz2;
        NODE p2;

        /* coefficient    power of x        power of y        power of z */
        cf1 = p1->cf;   px1 = p1->px;   py1 = p1->py;   pz1 = p1->pz;

        p2 = h2->link;
        while ( p2 != h2 )
        {
                /* coefficient    power of x        power of y        power of z */
                cf2 = p2->cf;   px2 = p2->px;   py2 = p2->py;   pz2 = p2->pz;

                if ( px1 == px2 && py1 == py2 && pz1 == pz2) break;

                p2 = p2->link;                  // obtain the next term of polynomial 2
        }

        return p2;
}
```

Once we know how to search for a term of polynomial 1 in polynomial 2, the general procedure to add two polynomials is shown below:

for each term of polynomial 1

Step 1: access each term of poly 1

Step 2: search for power of above term in poly2

Step 3: if found in poly 2

Add the coefficients and add sum to poly 3

else

Add the term of poly1 to poly 3

end for

Add remaining terms of poly2 to poly3

Now, the above algorithm can be written using C statements as shown below:

Example 9.47: Function to add two polynomials

```
NODE add_poly (NODE h1, NODE h2, NODE h3)
{
        NODE        p1, p2;
        int         cf1, px1, py1, pz1, sum;
```

```
        p1 = h1->link;

        while (p1 != h1)        /* As long as term of polynomial 1 exists */
        {
                /* coeeficent     power of x       power of y       power of z of poly1 */
                cf1  = p1->cf,   px1 = p1->px,   py1 = p1->py,   pz1 = p1->pz;

                p2 = search(p1, h2);                    /* search power of p1 in p2 */

                if (p2 != h2)                           /* powers of poly1 found in poly 2 */
                {
                        sum = cf1 + p2->cf;             /*Add coefficients, insert to poly3*/

                        h3 = insert_rear (sum, px1, py1, pz1, h3);

                        p2->cf = -999;                  /* Delete the term of poly2 */

                }
                else    /* If not found, insert term of poly 1  to poly 3*/
                        h3 = insert_rear (cf1, px1, py1, pz1, h3);

                p1 = p1->link;          /* Get the next term of polynomial 1 */
        }

        h3 = copy_poly(h2, h3);         /* Copy remaining terms of poly 2 into poly 3*/

        return h3;                      /* return total terms in poly 3 */
}
```

Now, the function copy_poly() can be implemented as follows. Immediately after adding the terms of both polynomials, the term of polynomial 2 should be removed. This can be done by assigning -999 to co-efficient field of the corresponding term in poly 2. So, when all terms of polynomial 1 are scanned, we have to copy the remaining terms of polynomial 2 into poly 3. The remaining terms are the terms whose co-efficient field is not -999. Now, the function copy_poly() can be written as shown below:

Example 9.48: Function to copy remaining terms of polynomial 2 into polynomial 3

```
NODE copy_poly ( NODE h2, NODE h3)
{
        NODE  p2;
```

```
        int     cf2, px2, py2, pz2;

        p2 = h2->link;

        while ( p2 != h2 )
        {
                /* Add remaining terms of poly 2 into poly 3);
                if (p2->cf != -999)
                {
                        cf2 = p2->cf,   px2 = p2->px,   py2 = p2->py,   pz2 = p2->pz;

                        h3 = insert_rear (cf2, px2, py2, pz2, h3);
                }

                p2 = p2->link;          /* Get the next term of polynomial 2 */
        }

        return h3;
}
```

The complete C program to add two polynomials is shown below:

Example 9.49: Program to add two polynomials

```
#include <stdio.h>

#include <stdlib.h>

struct node
{
        int     cf;
        int     px;
        int     py;
        int     pz;
        struct node *link;
};

typedef struct node* NODE;

/* Include: Example 8.2: Function to get new node from the availability list */
/* Include: Example 9.41: Function to insert a term at the rear end */
/* Include: Example 9.42: Function to read a polynomial */
```

```
/* Include: Example 9.43: Function to display a polynomial  */
/* Include: Example 9.46: Function to search for term of poly 1 in poly 2 */
/* Include: Example 9.48: Function to copy remaining terms of poly2 to poly 3 */
/* Include: Example 9.47: Function to add two polynomials */

void main()
{
        NODE h1, h2, h3;

        h1 = getnode();
        h2 = getnode();
        h3 = getnode();

        h1->link = h1;  h2->link = h2;  h3->link = h3;

        printf("Enter the first polynomial\n");
        h1 = read_poly(h1);

        printf("Enter the second polynomial\n");
        h2 = read_poly(h2);

        printf("Poly 1:");      display(h1);
        printf("Poly 2:");      display(h2);
        printf("----------------------------------------------------------------------------\n");

        h3 = add_poly(h1, h2, h3);

        printf("Poly 3:");      display(h3);
        printf("----------------------------------------------------------------------------\n");
}
```

Note: Let us add the following polynomials:

Poly 1: $5x^3y^2z^3 + 6xy^3z^2 + 3xyz^4 + 5$

Poly 2: $10x^3y^3z^4 + 4x^3z^3 + 3xy^3z^2 + 10xy^3z + 7$. We should get the following result.

Poly 3: $5x^3y^2z^3 + 9xy^3z^2 + 3xyz^4 + 12 + 10x^3y^3z^4 + 4x^3z^3 + 10xy^3z$

The first two polynomials should be entered as shown below:

Input

Enter the first polynomial

Enter the number of terms in the polynomial: 4

Enter term: 1
Coeff = 5 pow x = 3 pow y = 2 pow z = 3

Enter term: 2
Coeff = 6 pow x = 1 pow y = 3 pow z = 2

Enter term: 3
Coeff = 3 pow x = 1 pow y = 1 pow z = 4

Enter term: 4
Coeff = 5 pow x = 0 pow y = 0 pow z = 0

Enter the second polynomial
Enter the number of terms in the polynomial: 5

Enter term: 1
Coeff = 10 pow x = 3 pow y = 3 pow z = 4

Enter term: 2
Coeff = 4 pow x = 3 pow y = 0 pow z = 3
Enter term: 3
Coeff = 3 pow x = 1 pow y = 3 pow z = 2

Enter term: 4
Coeff = 10 pow x = 1 pow y = 3 pow z = 1

Enter term: 5
Coeff = 7 pow x = 0 pow y = 0 pow z = 0

Output: The following output is obtained

Poly 1: +5 x^3 y^2 z^3 + 6 x^1 y^3 z^2 + 3 x^1 y^1 z^4 + 5

Poly 2: +10x^3 y^3 z^4 + 4 x^3 z^3 + 3 x^1 y^3 z^2 + 10 x^1 y^3 z^1 + 7

Poly 3: +5 x^3 y^2 z^3 + 9 x^1 y^3 z^2 + 3 x^1 y^1 z^4 + 12 + 10 x^3 y^3 z^4 + 4 x^3 z^3 + 10 x^1y^3 z^1

9.7 Sparse matrix representation using multilinked data structure

First, let us see "What is a sparse matrix?"

9.98 Circular and doubly linked Lists

Definition: A sparse matrix is a matrix that has very few non-zero elements spread out thinly. In other words, a matrix in which most of the elements are zeroes is called a sparse matrix.

Ex 1: Consider the following two-dimensional matrices:

	col0]	col[1]	col[2]
row[0]	**10**	**20**	**30**
row[1]	**11**	**0**	**31**
row [2]	**12**	**22**	**32**
row [3]	**13**	**23**	**0**

Not a sparse matrix

	col0]	col[1]	col[2]	col[3]
row[0]	**10**	**0**	**0**	**40**
row[1]	**11**	**0**	**0**	**0**
row [2]	**0**	**0**	**0**	**0**
row [3]	**0**	**0**	**0**	**50**
row [4]	**0**	**15**	**0**	**25**

Sparse matrix

Note the following points with respect to above two matrices:

- In the first matrix, 10 non-zero elements are present and 2 zero elements. The number of non-zero elements are more than the number of zero elements. So, it is not a sparse matrix.
- In the second matrix only 6 non-zero elements are present and 14 zero elements are present. So, the number of non-zero elements are less than the number of zero elements. So, it is a sparse matrix.

Now, the question is, "How can we store a value in a two-dimensional array?" A *value* can be stored in the matrix A at specified *row* and *column* as shown below:

```
A[row][col] = value;
```

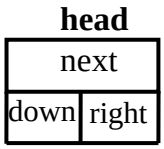
So, any element in a matrix *A* can be accessed using triples <*row, col val*>

Now, let us see "What is the disadvantage of a sparse matrix?" The sparse matrix contains many zeroes. If we are manipulating only non-zero values, then we are wasting the memory space by storing unnecessary zero values.

The above disadvantage can be overcome by storing only non-zero values thus saving the space as shown in subsequent sections.

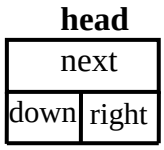
The sparse matrix can be more efficiently represented using linked list. In the linked representation:

- Each column of a sparse matrix is represented as a circularly linked list with a header node having three fields: *down*, *right* and *next* fields as shown below:



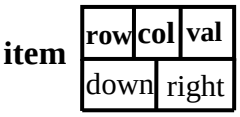
next – is the header of next column
right –not used
down – points to next item in the column

- Each row of a sparse matrix is represented as a circularly linked with a header node having three fields: *down*, *right* and *next* fields as shown below:



next – not used
right – points to next item in the row
down – not used

- Each item is represented as a node having 5 fields as shown below:

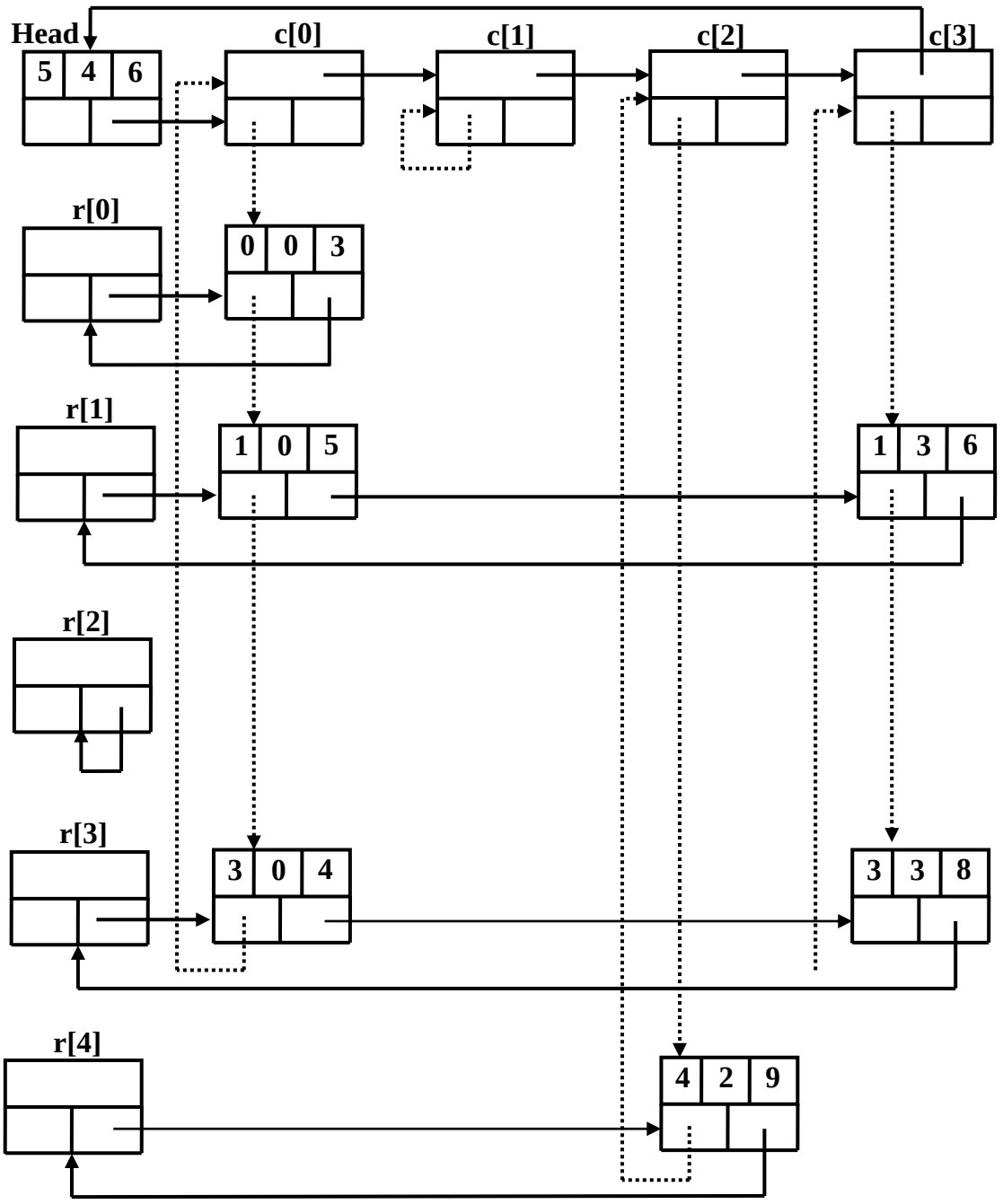


right – points to next item in the row
down – points to next item in the column

Now, consider the following 5 x 4 matrix:

	col[0]	col[1]	col[2]	col[3]
row [0]	**3**	**0**	**0**	**0**
row [1]	**5**	**0**	**0**	**6**
row [2]	**0**	**0**	**0**	**0**
row [3]	**4**	**0**	**0**	**8**
row [4]	**0**	**0**	**9**	**0**

The above matrix can be represented using linked list as shown below:



Note: The first node in the above list identified by the variable *head* contains the size of the matrix where 5 represents the number of rows, 4 represent the number of columns and 6 represent non-zero elements to be manipulated.

Exercises

1) What are the disadvantages of singly linked lists?

2) What is circular list? What are the advantages of circular lists?

3) Write C functions to perform following operations on circular singly linked lists
   a) Insert front b) Insert rear c) delete front d) delete rear e) display

4) What is circular singly linked list with header node?

5) Write C functions to perform following operations on circular singly linked list with a header node
   a) Insert front b) Insert rear c) delete front d) delete rear e) display

6) What are the disadvantages of singly linked lists?

7) What is a doubly linked list? What are the advantages of doubly linked lists?

8) Write C functions to perform following operations using doubly linked lists
   a) Insert front b) Insert rear c) delete front d) delete rear e) display

9) Write C program to implement dequeue using doubly linked list

10) What is circular doubly linked list? What is the advantage of circular doubly linked list?

11) Write C functions to perform the following operations using circular doubly linked lists
   a) Insert front b) Insert rear c) delete front d) delete rear e) display

12) What is circular doubly linked list with a header?

13) Write C functions to perform the following operations using circular doubly linked lists with a header node
   a) Insert front b) Insert rear c) delete front d) delete rear e) display

14) Write a program to implement dequeue using circular doubly linked list

15) Design, Develop and Implement a menu driven Program in C for the following operations on **Doubly Linked List (DLL)** of Employee Data with the fields: **SSN, Name, Dept, Designation, Sal, PhNo"**
   a) Create a **DLL** of **N** Employees Data by using **end insertion**.
   b) Display the status of **DLL** and count the number of nodes in it
   c) Perform Insertion and Deletion at End of **DLL**
   d) Perform Insertion and Deletion at Front of **DLL**
   e) Demonstrate how this **DLL** can be used as **Double Ended Queue**

16) What are the applications of linked lists?

17) Write a C function to read a polynomial and to display a polynomial

18) Write a C function to evaluate a polynomial with two variables

19) Write a C function to add two polynomials

20) What is a sparse matrix?" How to represent sparse matrix using linked list?

9.6.1 Addition of two long positive numbers

Now, let us see "Why very large numbers cannot be added using + operator?"

The range of a signed number is -2^{n-1} to $+2^{n-1}-1$

- For 16-bit machine, n = 16. So, range is -2^{16-1} to $+2^{16-1} - 1$ (i.e., -32768 to +32767)
- For 32-bit machine, n = 32. So, range is -2^{32-1} to $+2^{32-1} - 1$(i.e., -2147483648 to +2147483647)

Some applications may require larger than these numbers. But, all the computers have a certain maximum number of bits required for representing an integer, float etc. If the size of the numbers exceeds this limit, ordinary arithmetic operations cannot be performed. Using linked lists, we can represent these large numbers and any arithmetic operation can be performed on these large numbers.

Now, let us see "How to write algorithm/function to add two long positive numbers?" This achieved by splitting the program into various modules based on activity performed as shown below:

- Reading a long positive number
- Writing a long positive number
- Adding two long positive numbers

Now, let us see "How to read a long positive number using linked list?"

Design: Let us design by taking a small number 6698274. This number can be split into groups of one digit each as shown below from the most significant digit to least significant digit.

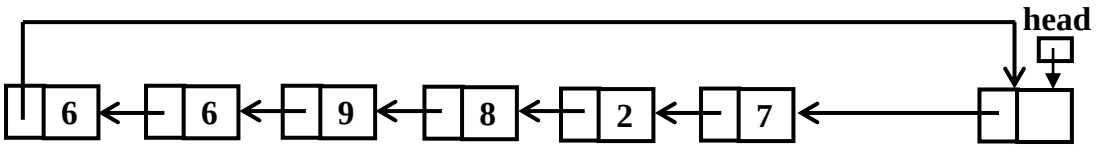
6, 6, 9, 8, 2, 7, 4

Since there are 7 digits, a singly linked list with seven nodes can be used to store each digit in each node. Consider the way the numbers are displayed on the display portion of the calculator. If the number 6698274 is typed, display portion of the calculator may look like as shown below.

6
6 6
6 6 9
6 6 9 8
6 6 9 8 2
6 6 9 8 2 7
6 6 9 8 2 7 4

Observe the following points:

- As the new digits are typed, old digits will be shifted one digit left thereby making room for the most recently typed digit.
- Thus the first digit typed will be the leftmost digit of the number and the last digit typed will be the right most digit of the number.
- If the position of the least significant digit i.e., right most digit of a number is considered as the front end, the digits that are typed will be inserted at the front end.
- The linked representation for the number typed i.e., **669827** using circular singly linked list with a header node is shown in figure below:



The next digit typed i.e., **4** will be the new least significant digit and has to be inserted immediately after the header node i.e., at the front end of the list. This is pictorially represented as shown below (see the dotted lines):

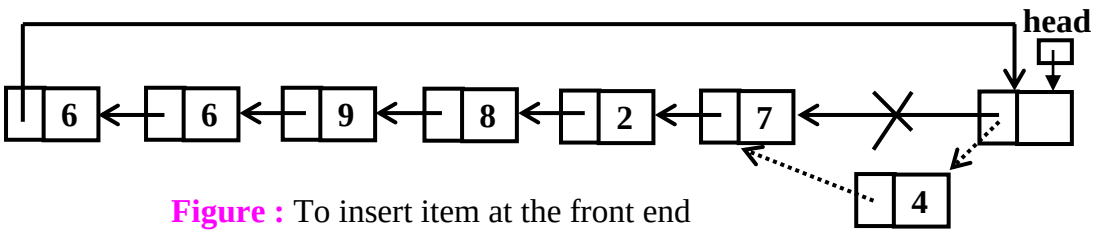


Figure : To insert item at the front end

Now, let us see "How to store the number entered from the keyboard using circular list with header node?"

The procedure can be written as shown below:

- Keep entering the digits till we press Enter key: This can be done using the following while loop:

```
while ( ( c = getche() ) != '\r' )
{
        // Insert the digit at the front end of the list
}
```

- Insert the digits at the front end of the list: As the digits are typed, they can be inserted at the front end of the list. The above while loop can be modified as shown below:

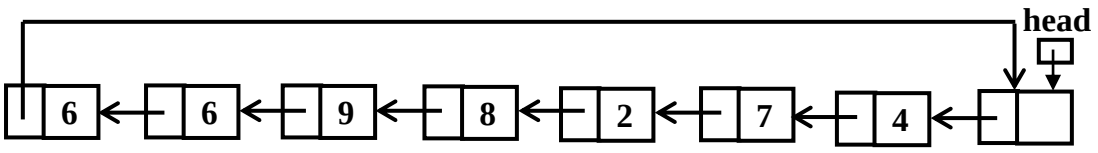
```
while ( ( c = getche() ) != '\r' )        // digit is a character. Convert into digit
{
        head = insert_front( c - '0',  head);   // and insert at the front end
}
```

Now, the function to read a number and store it in circular list can be written as shown below:

Example 9.37: Function to read a long positive number

```
NODE read_number(NODE head)
{
        char c;

        while ( ( c = getche() ) != '\r' )
        {
                head = insert_front( c - '0', head);
        }
        printf("\n");
        return head;
}
```

Now, if we call the above function and type the number 6698274 from the keyboard, we get the following circular list:



Now, the next question is "How to display the number stored in a linked list?" If we traverse the list from the first node

- the output will be: **4728966.**

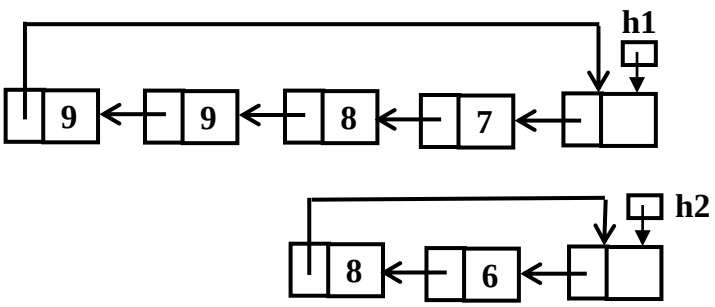
- But, we want the output to be: **6698274.** To get this output, traverse the list from front to last and copy each digit into an array. While printing, print the array elements in reverse order. Now, we get the output: **6698274**

The corresponding code can be written as shown below:

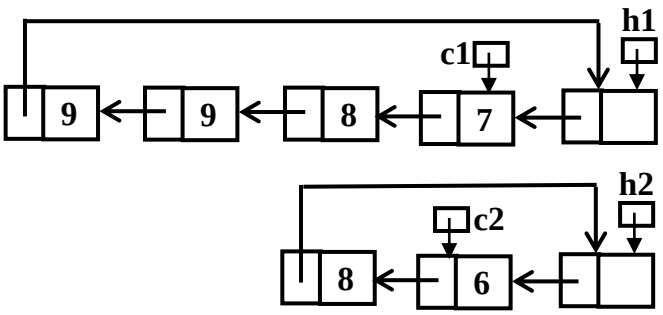
Example 9.38: Function to display a long positive integer

```
void display_number(NODE head)
{
        int k, a[20];
        NODE cur;

        cur = head->link,  k = 0;

        /* Traverse the list and copy into array */
        while (cur != head)
        {
                a[k++] = cur->info;          // Instead of displaying the digit, store it
                cur = cur->link;
        }

        /* Display array elements in reverse order */
        while (--k != -1 ) printf("%d", a[k]);

        printf("\n");
}
```

Now, we know how to read a number and display a number.
Now, let us see "How to add two numbers using linked list?" Suppose the two numbers to be added are **987** and **86**. These two numbers can be represented using circular linked list with a header node as shown below:



First, we have to add the digits 7 and 6. Let use *c1* to access the digits of first list and *c2* to access the digits of second list. This can be done by pointing *c1* to first node of first list and pointing *c2* to first node of the second list as shown below:



The corresponding code can be written as shown below:

```
carry = 0;
c1 = h1->link;
c2 = h2->link;
```

The various steps to be followed for adding two numbers are shown below:

Step 1: Add the two digits: It can be done using the statement:

```
sum = c1->info + c2->info + carry;          // sum = 7 + 6 = 13
```

Observe that initial *carry* is 0.

Step 2: Separate the answer and carry: In the above example, actual answer is 3 with carry 1. The actual answer and *carry* can be obtained using the following statements:

```
ans =  sum % 10;            // ans = 3
carry = sum / 10;           // carry = 1
```

Step 3: Insert answer to the end of resulting list: This can be done by inserting *ans* at the end of list *h3* using the statement:

```
h3 = insert_rear(ans, h3);
```

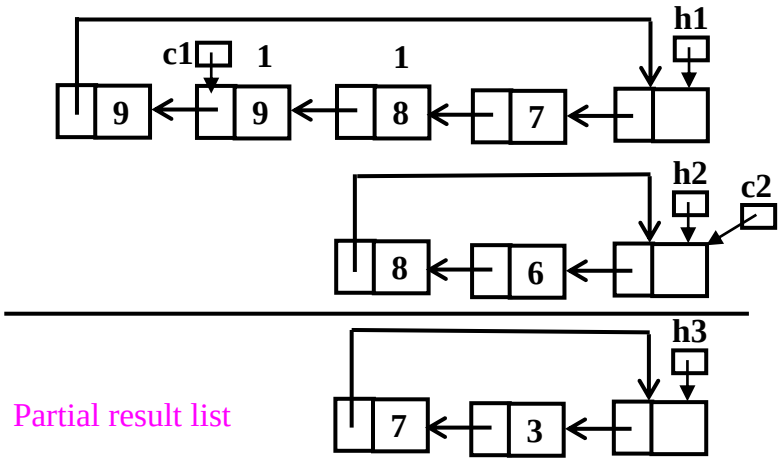
Step 4: Update pointers to point to the next digits: This can be done using the statements:

```
c1 = c1->link;
c2 = c2->link;
```

The above steps have to be repeatedly executed as long as one of *c1* or *c2* reaches the corresponding *header* node. The code for this case can be written as shown below:

```
while (c1 != h1 && c2 != h2)
{
        sum = c1->info + c2->info + carry;      // add the two digits
        ans =  sum % 10;                        // Separate the answer
        carry = sum / 10;                       // Separate the carry
        h3 = insert_rear(ans, h3);              // Insert answer at the end
        c1 = c1->link;                          // Point to next two digits
        c2 = c2->link;
}
```

Once control comes out of the loop, we get the following situation:



Step 5: Obtain the address of digit of existing number: Once the control comes out of the above loop, it means that one number still has digits whereas no more digits exists in another number. The address of the digit whose digits are still to be considered can be obtained using the following code:

```
if (c1 != h1)
        c = c1, h = h1;
else
        c = c2, h = h2;
```

Step 6: Add carry to the remaining digits of a number: This can be done using the following code:

```
while (c != h)
{
        sum = c->info + carry;          // add carry to remaining digits
        ans =  sum % 10;                // Separate the answer
        carry = sum / 10;               // Separate the carry
        h3 = insert_rear(ans, h3);      // Insert answer at the end
        c = c->link;                    // Point to next two digit
}
```

Step 7: Add carry to the result: If carry exists after adding all the digits, add that carry to the result. This can be done using the following code

```
if (carry == 1) h3 = insert_rear(carry, h3);
```

Step 8: Return the result: This can be done by executing the following code:

```
return h3;
```

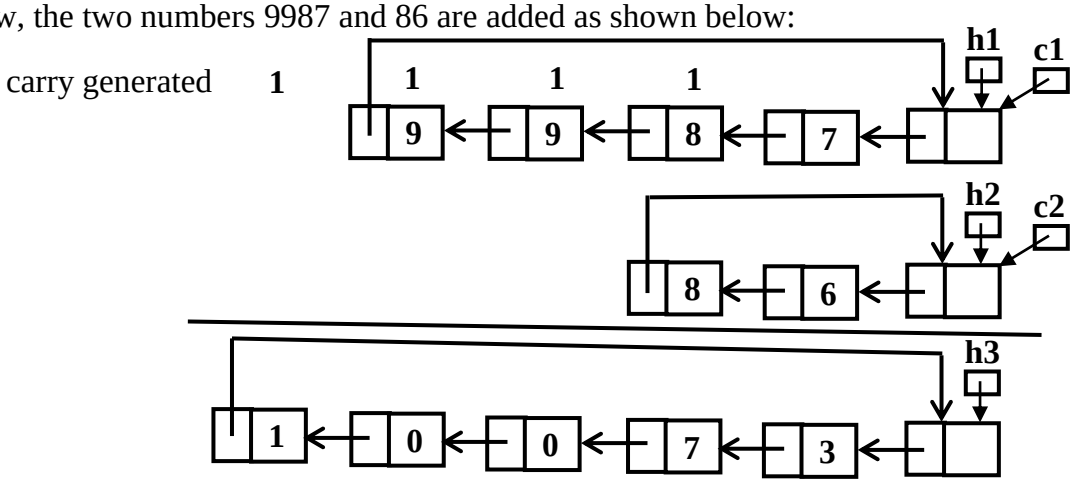
The final function to add two long positive integers can be written as shown below:

Example 9.39: Function to add two long positive numbers

```
NODE add_long(NODE h1, NODE h2, NODE h3)
{
        NODE c, c1, c2, h;
        int sum, carry, ans;

        carry = 0;              /*Initial carry is zero */
        c1 = h1->link;          /* Points to the first digit of the first number */
        c2 = h2->link;          /* Points to the first digit of the second number */

        /* add the two numbers digit by digit along with carry */
        while ( c1 != h1 && c2 != h2 )
        {
                sum = c1->info + c2->info + carry;
                ans = sum % 10;             /* Extract the result */
```

```
            carry = sum / 10;           /* Extract the carry  */

            h3 = insert_rear(ans, h3);  /* add the result to the partial output */

            c1 = c1->link;              /* Point to the next digit of 1st number */
            c2 = c2->link;              /* Point to the next digit of 2nd  number */
        }

        /* Obtain the address of the remaining digits of bigger number */
        if ( c1 != h1 )
            c = c1, h = h1;
        else
            c = c2, h = h2;

        /* Add the carry to the remaining digits of bigger number */
        while ( c != h )
        {
            sum = c->info + carry;
            ans = sum % 10;
            carry = sum / 10;
            h3 = insert_rear(ans, h3);
            c = c->link;
        }

        /* If carry add at the end of the result */
        if ( carry == 1 ) h3 = insert_rear(carry, h3);

        return h3;
}
```

Now, the two numbers 9987 and 86 are added as shown below:



The complete program to add two long positive numbers is shown below:

Example 9.40: C program to add two long positive numbers

```
#include <stdio.h>
#include <process.h>
#include <conio.h>
#include <alloc.h>
struct node
{
        int             info;
        struct node     *link;
};

typedef struct node* NODE;

/* Include: Example 8.2: Function to get new node from the availability list */
/* Include: Example 9.7: Function to insert at the front end of the list */
/* Include: Example 9.8: Function to insert at the rear end of the list */
/* Include: Example 9.37: Function to read a long positive number */
/* Include: Example 9.38: Function to display a long positive integer */
/* Include: Example 9.39: Function to add two long positive numbers */

void main()
{
        NODE h1, h2, h3;

        h1 = getnode();
        h2 = getnode();
        h3 = getnode();
        h1->link = h1;
        h2->link = h2;
        h3->link = h3;

        printf("Enter the first number\n");
        h1 = read_number(h1);

        printf("Enter the second number\n");
        h2 = read_number(h2);
```

Input

```
Enter 1st number
99999

Enter 2nd number
99999
```

```
        h3 = add_long(h1, h2, h3);
                                                        Output
        printf("Num1 = ");          display_number(h1);     Num1 =  99999
        printf("Num2 = ");          display_number(h2);     Num2 =  99999
        printf("Sum  = ");          display_number(h3);     Sum   = 199998
}
```

Chapter 10: Trees

What are we studying in this chapter?

- Terminology
- Binary Trees, Properties of Binary trees
- Array and linked Representation of Binary Trees,
- Binary Tree Traversals - Inorder, postorder, preorder
- Additional Binary tree operations.
- Threaded binary trees
- Binary Search Trees:
- Definition, Insertion, Deletion
- Traversal, Searching
- Application of Trees: Evaluation of Expression
- Programming Examples

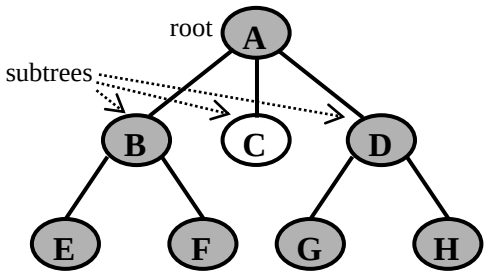
10.1 Introduction

Now, let us see "What is a tree?" A tree is recursively defined as follows.

Definition: A tree is a set of finite set of one or more nodes that shows parent-child relation such that:

- There is a special node called the root node
- The remaining nodes are partitioned into disjoint subsets $T_1, T_2, T_3 \ldots\ldots T_n$, $n \geq 0$ where $T_1, T_2, T_3 \ldots\ldots T_n$ which are all children of root node are themselves trees called *subtrees*.

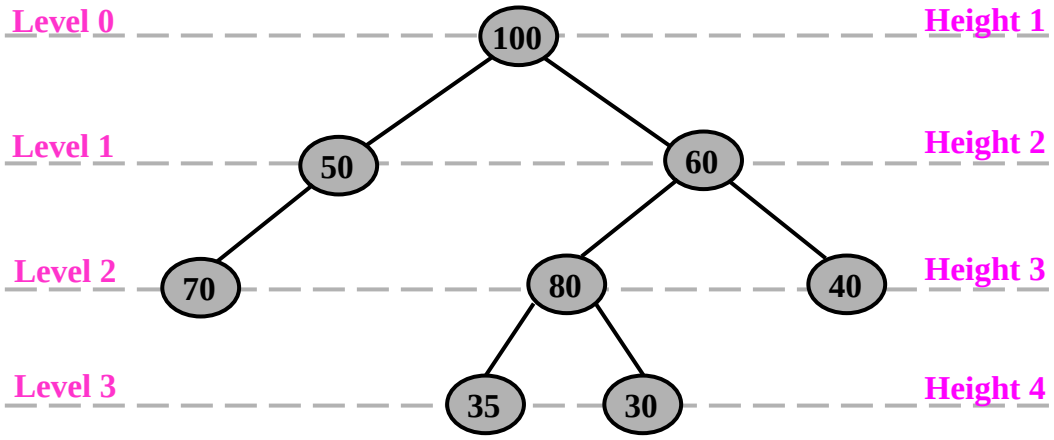
Ex1 : Consider the following tree. Let us identify the root node and various subtrees:



- The tree has 8 nodes : A, B, C, D, E, F, G and H.
- The node A is the *root* of the tree
- We normally draw the trees with root at the top
- The node B, C and D are the children of node A and hence there are 3 subtrees identified by B, C and D
- The node A is the parent of B,C and D whereas D is the parent of G and H

Now, let us see the terminologies used in a tree. Consider the following tree for explanation:

10.2 Trees



Number of levels = 4 Maximum height/depth of the tree = 4

Fig. 10.1: A tree showing the levels and height

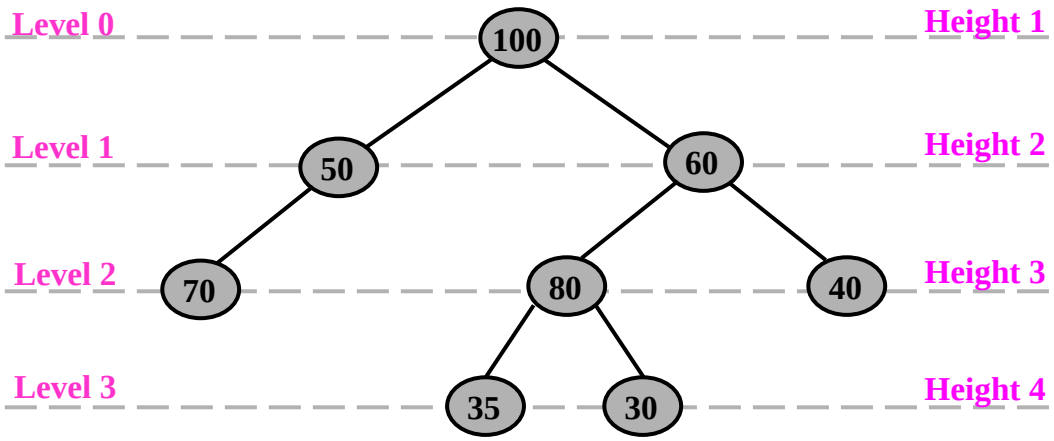
- **Root node:** A first node written at the top is root node. The root node does not have the parent. The node 100 is the root node in above tree.
- **Child:** The node obtained from the parent node is called child node. A parent node can have zero or more child nodes.
  - 50 and 60 are children of 100
  - 80 and 40 are children of 60
  - 70 is child of 50
  - 35 and 30 are children of 80
- **Siblings:** Two or more nodes having the same parent are called siblings. For example, for the tree shown in Fig. 10.1
  - 50 and 60 are siblings since they have same parent 100
  - 80 and 40 are siblings since they have same parent 60
  - 35 and 30 are siblings since they have same parent 80
- **Ancestors:** The nodes obtained in the path from the specified node **x** while moving upwards towards the root node are called ancestors. For example, for the tree shown in Fig. 10.1
  - 100 is the ancestor of 50 and 60
  - 60 is the ancestor of 35, 30, 80 and 40
  - 50 and 100 are the ancestors of 70
  - 60 and 100 are the ancestors of 80, 40, 35, 30
  - 80, 60 and 100 are the ancestors of 35 and 30

- **Descendants:** The nodes in the path below the parent are called descendants. In other words, the nodes that are all reachable from a node **x** while moving downwards are all called **descendants** of **x**. For example, for the tree shown in Fig. 10.1:
  - All the nodes below 100 are descendants of 100
  - All the nodes below 50 are descendants of 50 and so on.
- **Left descendants:** The nodes that lie towards left subtree of node **x** are called left descendants. For example, for the tree shown in Fig. 10.1
  - 50 and 70 are left descendants of 100
  - 80, 35 and 30 are the left descendants of 60
  - 35 and 80 are left descendants of 60
- **Right descendants:** The nodes that lie towards right subtree of node *x* are called right descendants. For example, for the above tree
  - The right descendants of 100 are 60, 80, 40, 35 and 30
  - The right descendent of 80 is 30
  - The right descendent of 60 is 40
- **Leftsubtree:** All the nodes that are all left descendants of a node *x* form the **left subtree** of *x***.** For example, for the above tree
  - The left subtree of 100 are 50 and 70
  - The left subtree of 60 are 80, 35 and 30
- **Right subtree:** All the nodes that are all right descendants of a node *x* form the **right subtree** of *x***.** For example, for the above tree
  - The are right descendants of 100 are 60, 80, 40, 35 and 30
  - The right descendent of 80 is 30
  - The right descendent of 60 is 40
- **Parent:** A node having left subtree or right subtree or both is said to be a parent node for the left subtree and/or right subtree. The word father is also used in place of parent. [Let us use the word parent, otherwise mothers may feel bad]. For example, for the above tree
  - The parent for 50 and 60 is 100
  - The parent for 70 is 50
  - The parent for 80 and 40 is 60
  - the parent for 35 and 30 is 80
- **Degree:** The number of subtrees of a node is called its *degree*. For example,
  - The node 100 has two subtrees. So, degree of node 100 is 2
  - The node 50 has one subtree. So, degree of node 50 is 1
  - The node 70 has no subtree. So, degree of node 70 is 0

- **Leaf:** A node in a tree that has a degree of zero is called a **leaf node.** In other words, a node with an empty left child and an empty right child is called a leaf node. It is also called a terminal node. For example,
  - 70, 35, 30 and 40 are the leaf nodes

- **Internal nodes**: The nodes except leaf nodes in a tree are called internal nodes. For example,
  - 100, 50, 60 and 80 are the internal nodes

- **External nodes:** The NULL link of any node in a tree is an external node. For example, rlink of 50, rlink and llink of nodes 70, 35, 30 and 40 are all external nodes.



- **Level:** The distance of a node from the root is called *level* of the node.
  - The distance from *root* to itself is 0. So, level of root node is 0.
  - The node 50 is at a distance of 1 from root node. So, its level is 1.
  - The node 70 is at a distance of two nodes from root node. So, its level is 2.
  - The node 35 and 30 are at a distance of 3 nodes from the root node. So, their levels are 3.
  - The level of each node is shown in above tree:

- **Height (Depth):** The height of the tree is defined as the maximum level of any leaf in the tree. For example,
  - height of above tree is 4

10.2 Representation of Trees

Now, let us see "What are the different ways of representing a tree?" A tree can be represented in three different ways.

- List representation
- Left child – right sibling representation
- Binary tree representation (Degree-2 representation)

10.2.1 List representation

Now, let us see "How a tree is represented using lists?" A tree can be represented using list as shown below:

- The root node comes first.
- It is immediately followed by a list of subtrees of that node.
- It is recursively repeated for each subtree. For example, consider the tree shown below:

`

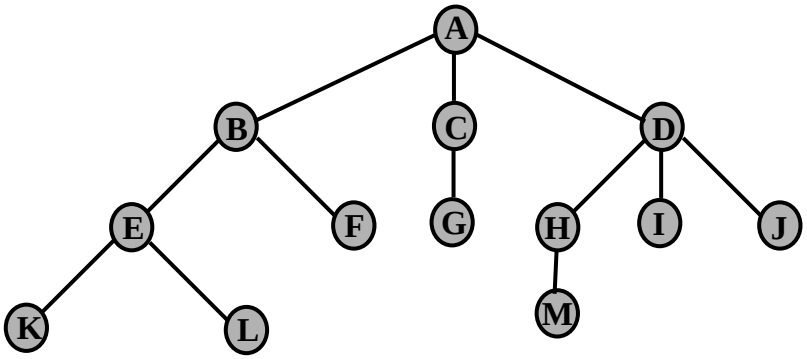
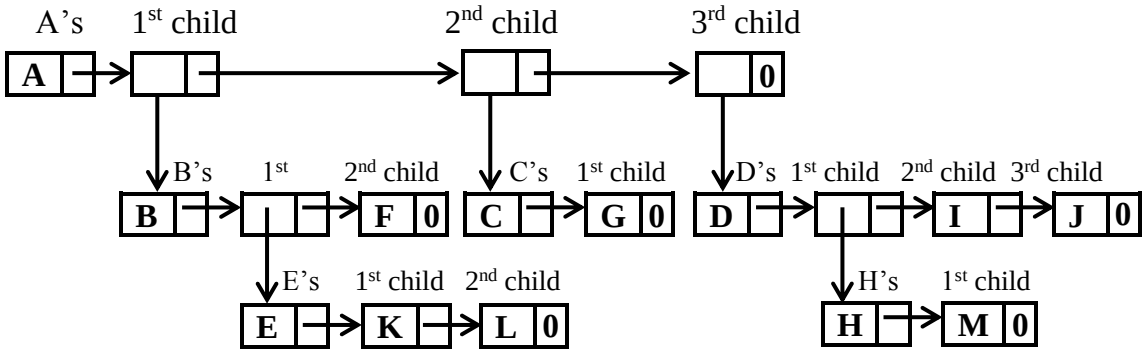


Fig. 10.2: A tree

The above tree can be represented using lists as shown below:



Observe the following points from above list representation:

- Since there are three children for node A in the tree, there are three nodes to the right of A in the list representation
- A's first child is B, 2nd child is C and 3rd child is D and they are shown using down links in list representation
- Since there are two children for node B in the tree, there are two nodes to the right of B in the list representation.
- Since there is only one child for C in the tree, there is only one node to the right of C in the list representation
- Since there are three children for node D in the tree, there are three nodes to the right of D in the list representation

10.2.2 Left-child Right-sibling representation

Now, let us see "What is left child-right sibling representation?" A left child-right sibling representation of a given tree can be obtained as shown below:

- The left pointer of a node in the tree will be the left child in this representation
- The remaining children of a node in the tree are inserted horizontally to the left child in the representation. For example, consider the tree shown below:

`

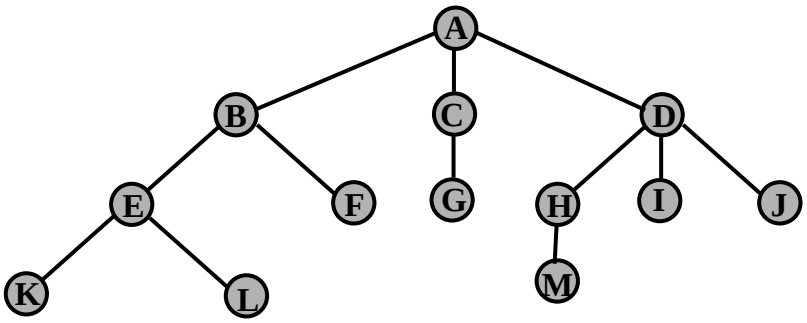
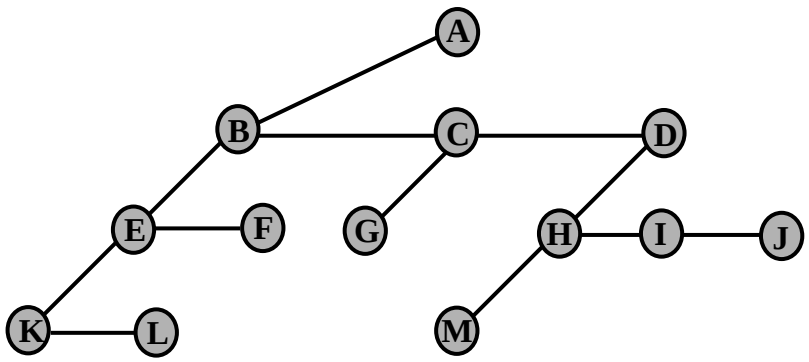


Fig. 10.3: A tree

The left child-right sibling representation of above tree can be written as shown below:



Observe the following points from above representation:

- A's left child is B in the tree. So, A's left child is B in the representation
- A's remaining children such as C and D in the tree are inserted horizontally to node B in the representation
- B's left child is E in the tree. So, B's left child is E in the representation
- B's remaining children such as F in the tree are inserted horizontally to node E in the representation.
- Similarly, D's left child is H in the tree. So, D's left child is H in the representation
- D's remaining children such as I and J in the tree are inserted horizontally to H in the representation.

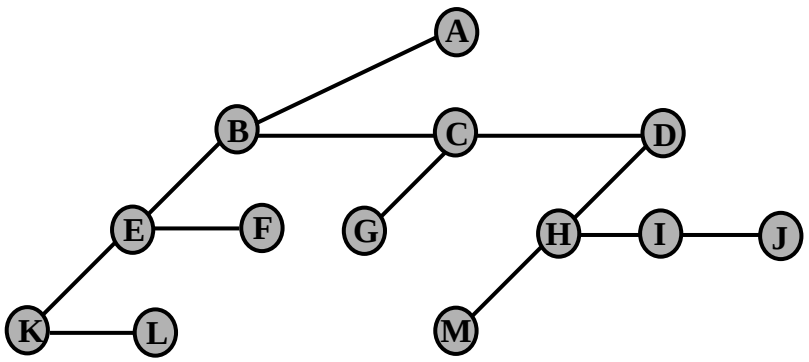
10.2.3 Binary tree (Degree two tree) representation

Now, let us see "What is binary tree representation?"

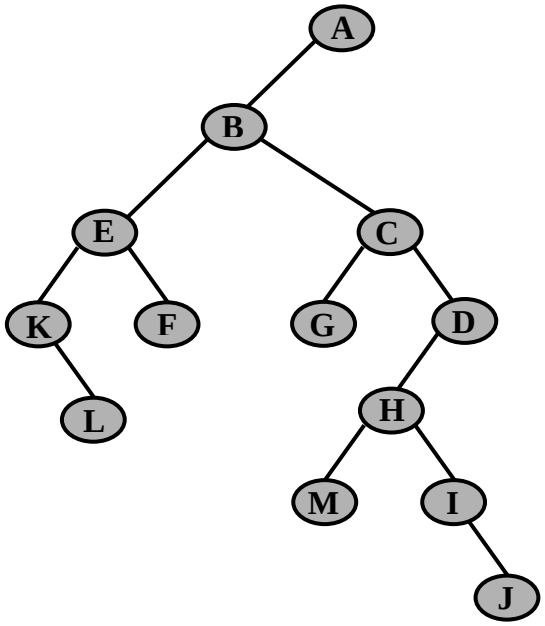
A binary tree representation is also called *left child-right child* tree representation or degree two representation. A binary tree representation can be obtained as shown below:

- Obtain left child-right sibling representation as shown in previous example.
- Rotate the horizontal lines clockwise by 45 degrees

For example, consider the left child-right sibling representation discussed in previous example. It is shown again for quick reference.



By rotating horizontal lines 45 degrees clockwise, we get the following binary tree:



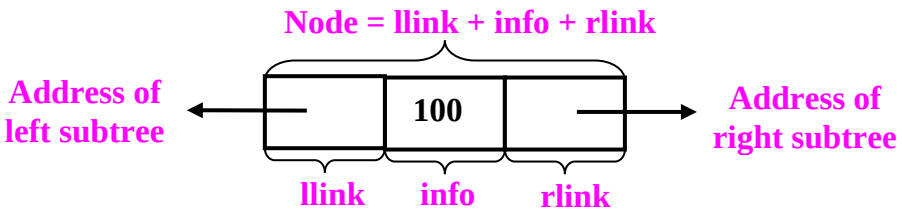
10.3 Binary trees

Now, let us see "What is a binary tree?"

Definition: A binary tree is a tree which has finite set of nodes that is either empty or consist of a root and two subtrees called left subtree and right subtree. A binary tree can be partitioned into three subgroups namely root, left subtree and right subtree.

- **Root** – If tree is not empty, the first node in the tree is called root node.
- **left subtree** – It is a tree which is connected to the left of root. Since this tree comes towards left of root, it is called left subtree.
- **right subtree** – It is a tree which is connected to the right of root. Since this tree comes towards right of root, it is called right subtree.

Note: In general, a tree in which each node has either zero, one or two subtrees is called a binary tree. The pictorial representation of a typical node in a binary tree is shown below:

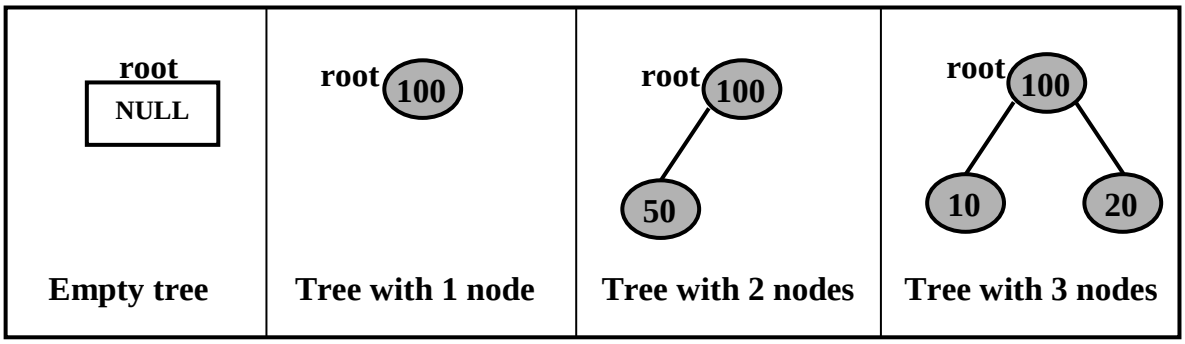


Note that a node in a tree consists of three fields namely llink, info and rlink:

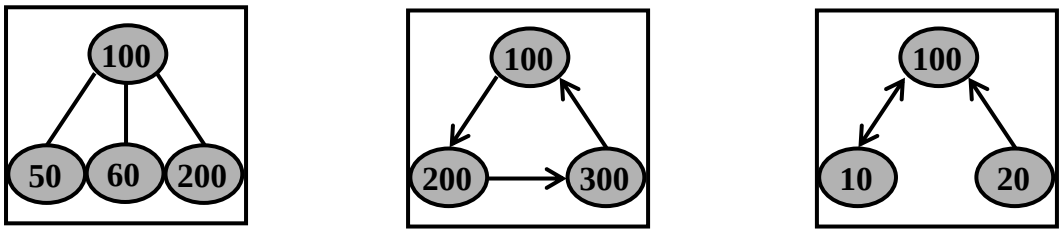
- **llink** –contains address of left subtree
- **info** – This field is used to store the actual data or information to be manipulated
- **rlink** –contains address of right subtree.

The pictorial representation of above node can also be written as shown below:

Note: In the text book, we show without directions. But, it is implied that directions are present moving away from the node. The following figure shows some of the binary trees:



Note: An empty tree is also a binary tree. Binary here means at most two i.e., zero, one or two subtrees are possible. But, more than two subtrees are not permitted. Consider the trees shown below: Are they binary trees? No.



Let us see "Why the above trees are not binary trees?"

- The tree shown in Fig (a) is not binary tree. This is because, it has more than two subtrees.
- The tree shown in Fig (b) is not a binary tree. There is a cycle from 100 to 200, 200 to 300 and 300 back to 100. The tree should not have cycles.
- The node 100 has subtree 10 and 10 has the subtree 100. If 10 is subtree of 100, then 100 cannot be the subtree of 10. So, it is not a binary tree.

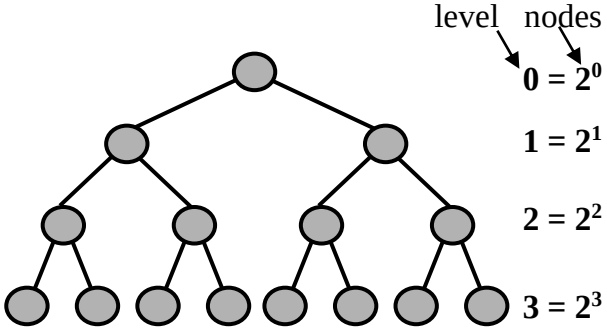
10.4 Properties of binary trees

In this section, let us see various properties of binary trees.

Lemma: a) The maximum number of nodes on level *i* of a binary tree = 2^i for $i \geq 0$
b) The maximum number of nodes in a binary tree of depth *k* = $2^k - 1$

Proof: Consider the following complete binary tree and observe the following factors from the complete binary tree:

Number of nodes at level 0 = 1 = 2^0
Number of nodes at level 1 = 2 = 2^1
Number of nodes at level 2 = 4 = 2^2
Number of nodes at level 3 = 8 = 2^3
……
……
Number of nodes at level ***i*** = 2^i.



Total number of nodes in the full binary tree of level *i* = $2^0 + 2^1 + 2^2 + \ldots\ldots 2^i$.

The above series is a geometric progression whose sum is given by

$S = a (r^n - 1) / (r - 1)$
where a = 1, n = i+1 and r = 2

So, total number of nodes $n_t = a (r^n - 1) / (r - 1)$
$= 1 (2^{i+1} - 1)/ (2 - 1)$
$= 2^{i+1} - 1$

Therefore, total number of nodes $\boxed{n_t = 2^{i+1} - 1}$ Substituting i = 3 (see above tree)

we get n_t = 15 which is total number of nodes in a fully binary tree

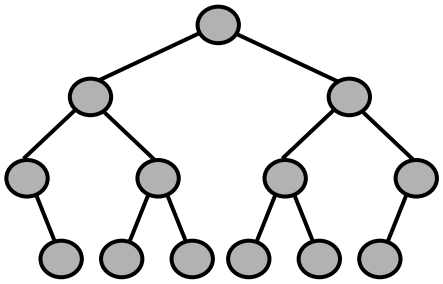
The depth of the tree k = maximum level + 1.
= i + 1

Substituting this value in above equation we get $\boxed{n_t = 2^k - 1}$

Lemma: The number of leaf nodes is equal to number of nodes of degree 2

Now, let us see "What is the relation between the number of leaf nodes and degree-2 nodes?"

Solution: Consider a binary tree of degree-2 nodes. That is, each node in the tree should have maximum of two children. A node need not have any children, or it can have only one child or it can have two children. But, a node cannot have more than two children.



Let number of nodes of degree-0 $= n_0$
Let number of nodes of degree-1 $= n_1$
Let number of nodes of degree-2 $= n_2$
Let total number of nodes in the tree $= n = n_0 + n_1 + n_2$ ----------(1)

Observe from the tree that total number of nodes is equal to the total number of branches (B) plus one. That is, $n = B + 1$ ----------(2)

If there is node with degree 1, number of branches = 1

So, for n_1 number of nodes of degree 1, number of branches = n_1 ----------(3)

If there is node with degree 2, number of branches = 2

So, for n_2 number of nodes of degree 2, number of branches = $2n_2$ ----------(4)

By adding (3) and (4) we get total number of branches $B = n_1 + 2n_2$ ----------(5)

Substituting (5) in (2) we get, $n = n_1 + 2n_2 + 1$ ----------(6)

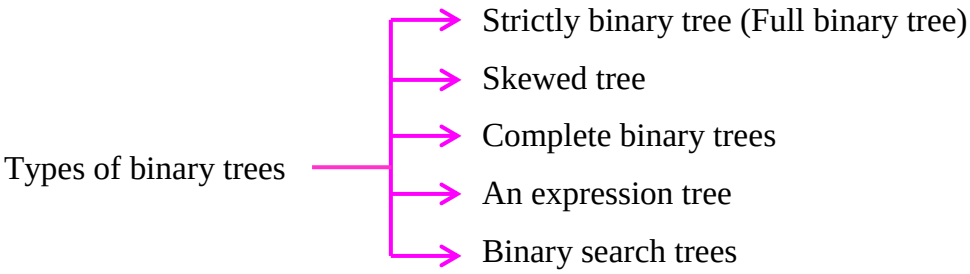
The relation between number of leaf nodes and number of nodes of degree-2 can be obtained by subtracting (6) from (1). That is,

$$\begin{array}{rl} & n = n_0 + n_1 + n_2 \\ - & n = n_1 + 2n_2 + 1 \\ \hline & 0 = n_0 - n_2 - 1 \end{array}$$

By re arranging we get, $\boxed{n_0 = n_2 + 1}$

10.5 Types of binary trees

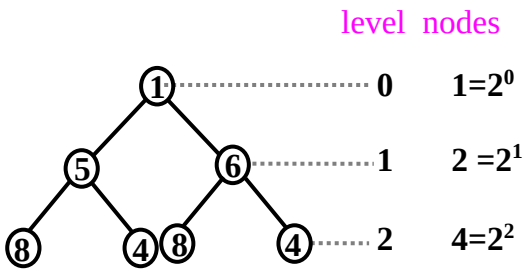
Once we know the terminologies or nomenclature used for trees, now let us see "What are the different types of binary trees?" The binary trees are classified as shown below:



10.5.1 Strictly(Full) binary tree

Now, let us see "What is strictly binary tree?"

Definition: A binary tree having 2^i nodes in any given level *i* is called strictly binary tree. Here, every node other than the leaf node has two children. A strictly binary tree is also called *full binary tree* or *proper binary tree*.



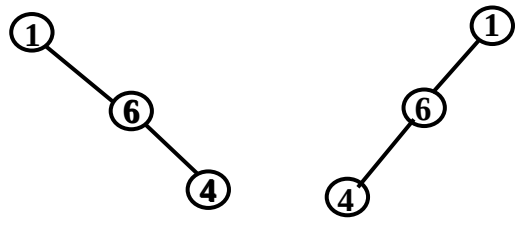
For example, in full binary tree shown,

- The number of nodes at level 0 = $2^0 = 1$
- The number of nodes at level 1 = $2^1 = 2$
- The number of nodes at level 2 = $2^2 = 4$

10.5.2 Skewed tree

Now, let us see "What is skewed tree?"

Definition: A skewed tree is a tree consisting of only left subtree or only right subtree. A tree with only left subtrees is called left skewed binary tree and a tree with only right subtree is called right skewed binary tree.

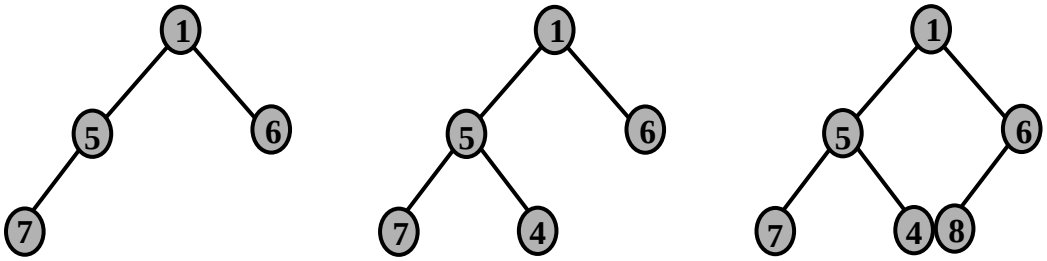


Right skewed tree Left skewed tree

10.5.3 Complete binary tree

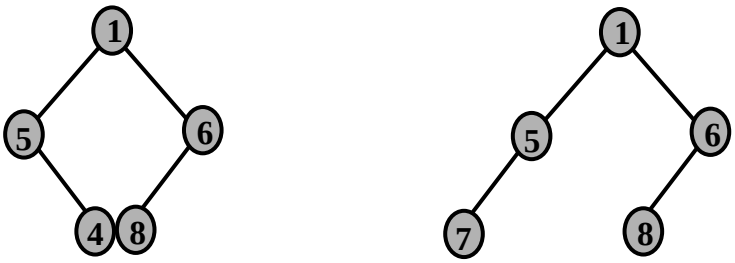
Now, let us see "What is a complete binary tree?"

Definition: A complete binary tree is a binary tree in which every level, *except possibly the last* level is completely filled. If the nodes in the last level are not completely filled, then all the nodes in that level should be filled only from left to right. For example, all the trees shown below are complete binary trees.



(a) **(b)** **(c)**

The tree shown below is not complete binary tree since the nodes in the last level are not filled from left to right. There is an empty left child for node 5 in figure (d) and there is an empty right child for node 5 in figure (e). All the nodes should be as left as possible.



(d) **(e)**

10.6 Binary tree representation

Now, let us see "How binary trees are represented?" The storage representation of binary trees can be classified as shown below:

- Array representation (uses static allocation technique)
- Linked representation (uses dynamic allocation technique)

10.6.1 Linked representation

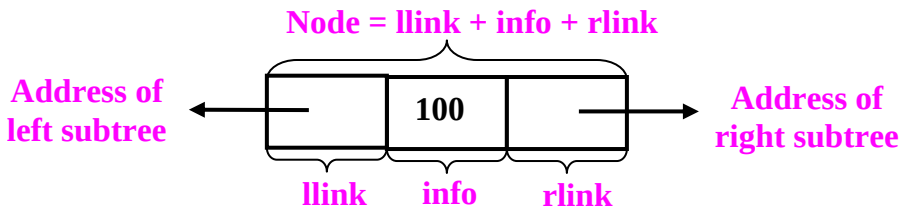
Now, let us see "How a tree is represented using linked allocation technique?" In a **linked** representation, a node in a tree has three fields:

- **info** – which contains the actual information
- **llink** – which contains address of the left subtree
- **rlink** – contains address of the right subtree.

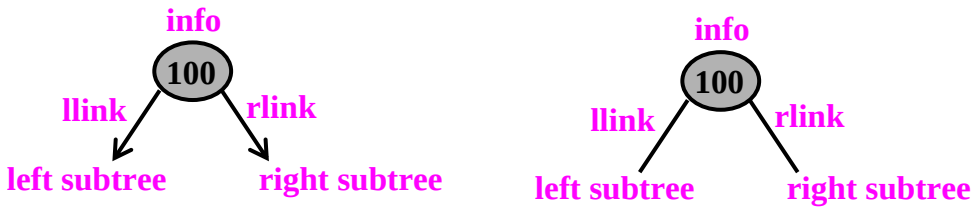
So, a node can be represented using self-referential structure as shown below:

```
struct node
{
        int             info;
        struct node     *llink;
        struct node     *rlink;
};

typedef struct node*  NODE;
```

The pictorial representation of a typical node in a binary tree is shown below:



The pictorial representation of above node can also be written as shown below:



A pointer variable *root* can be used to point to the root node always. If the tree is empty, the pointer variable *root* points to NULL indicating the tree is empty. The pointer variable *root* can be declared and initialized as shown below:

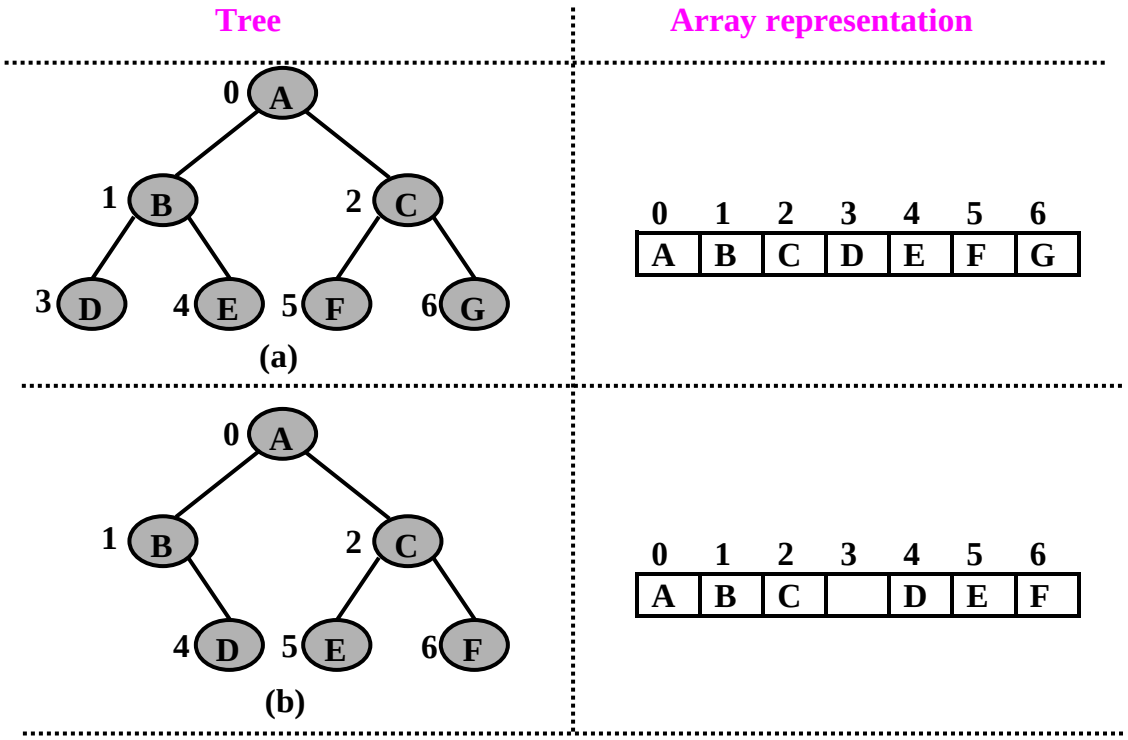
```
NODE  root = NULL;
        or
struct node *  root = NULL;
```

Memory can be allocated or de-allocated using the C function malloc() or calloc().

10.6.2 Array representation

Now, let us see "How a tree is represented using static allocation (using arrays) technique?"

A tree can also be represented using an array, which is called sequential representation. Consider the trees shown in fig. below along with array representation.



Observe the following points:

- The nodes are numbered sequentially from 0.
- The node with position 0 is considered as the root node.
- If an index ***i*** is 0, it gives the position of the root node.

- Given the position of any other node ***i***, **2i+1** gives the position of the left child and **2i+2** gives the position of the right child.
- If ***i*** is the position of the left child, **i+1** gives the position of the right child
- If ***i*** is the position of the right child, **i-1** gives the position of the left child.
- Given the position of any node ***i***, the parent position is given by **(i-1) / 2**. If ***i*** is odd, it points to the left child otherwise, it points to the right child.

A tree can be represented using arrays in two different methods as shown below:

Method 1: In the first representation shown below, some of the locations may be used and some may not be used. For this, a **flag** field namely, ***used*** is used just to indicate whether a memory location is used to represent a node or not. If flag field ***used*** is 0, the corresponding memory location is not used and indicates the absence of node at that position. So, each node contains two fields:

- **info** where the information is stored
- **used** indicates the presence or the absence of a node

The structure declaration for this is shown below:

```
#define MAX_SIZE 200
struct node
{
        int info;
        int used;
};
typedef struct node NODE;
```

An array ***a*** of type **NODE** can be used to store different items and the declaration for this is shown below:

```
NODE  a [MAX_SIZE];
```

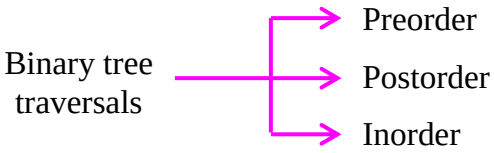
Method 2: An alternate representation is that, instead of using a separate flag field ***used*** to check the presence of a node, one can initialize each location of the array to 0 indicating the node is not used. Non-zero value in the location indicates the presence of the node.

10.7 Binary tree Traversals

Now, let us see "What is the meaning of traversing a tree?"

Definition: Traversing is a method of visiting each node of a tree exactly once in a systematic order. During traversal, we may print the **info** field of each node visited.

Now, let us see "What are the different traversal techniques of a binary tree?" The various traversal techniques of a binary tree are shown below:

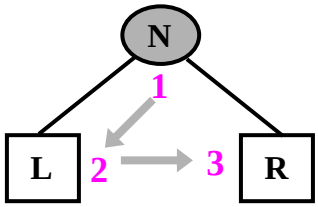


10.7.1 Preorder traversal

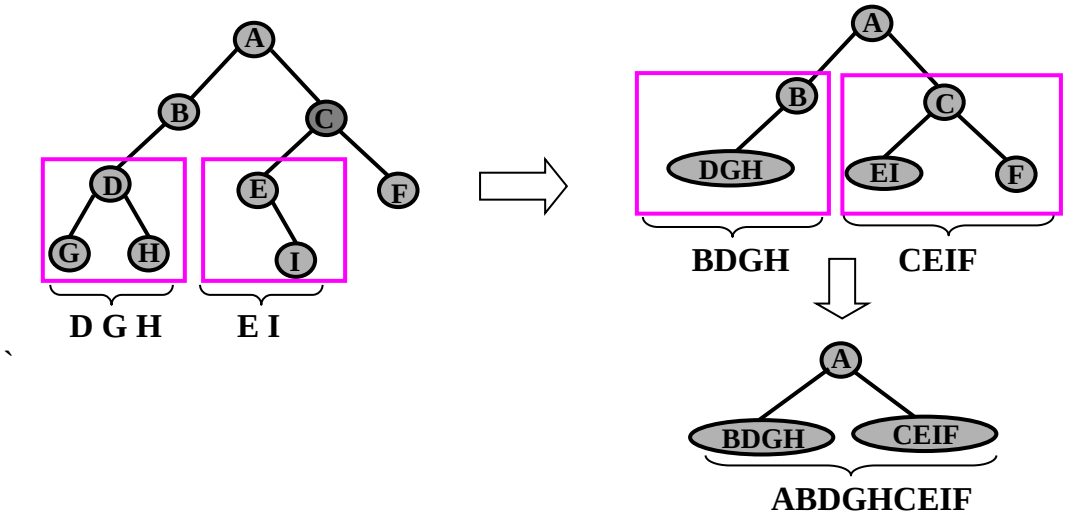
Now, let us see "What is preorder traversing of a binary tree?"

Definition: The preorder traversal of a binary tree can be recursively defined as follows:

1. Process the root **N**ode **[N]**
2. Traverse the **L**eft subtree in preorder **[L]**
3. Traverse the **R**ight subtree in preorder **[R]**



For example, let us traverse the following tree in preorder.



`

The C function to traverse the tree in preorder can be recursively defined as shown below:

Example 10.1: Function to traverse the tree in preorder

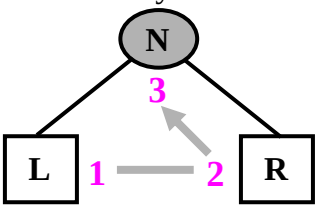
```
void preorder(NODE root)
{
        if ( root == NULL ) return;

        printf("%d ",root->info);          /* visit the node */
         preorder(root->llink);            /* visit left subtree in preorder*/
         preorder(root->rlink);            /* visit right subtree in preorder*/
}
```

10.7.2 Postorder traversal

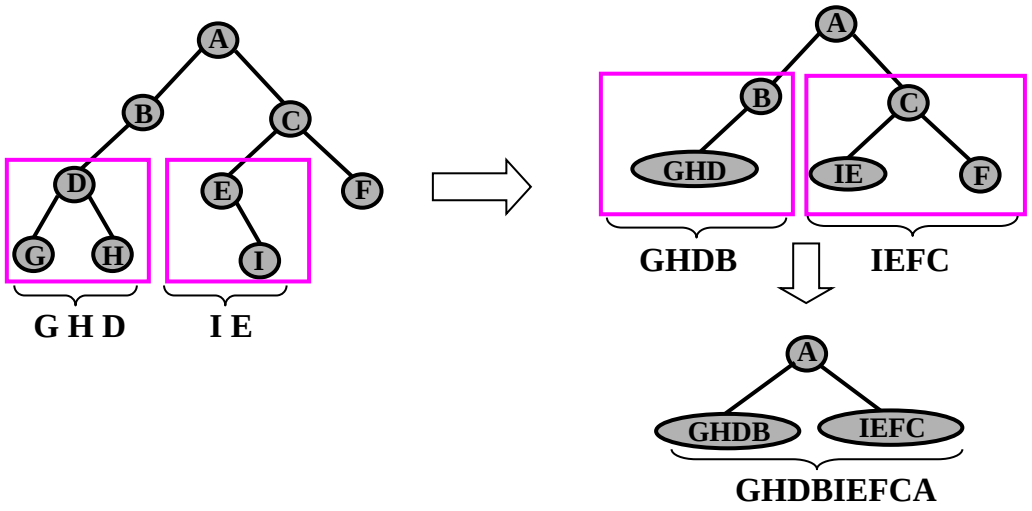
Now, let us see "What is postorder traversing of a binary tree?"

Definition: The postorder traversal of a binary tree can be recursively defined as follows:

1. Traverse the **L**eft subtree in postorder **[L]**
2. Traverse the **R**ight subtree in postorder **[R]**
3. Process the root **N**ode **[N]**



For example, let us traverse the following tree in preorder.



The C function to traverse the tree in postorder is shown below:

Example 10.2: Function to traverse the tree in postorder

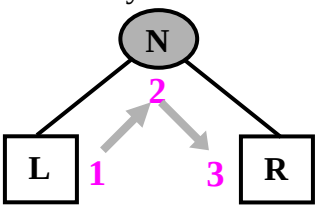
```
void postorder(NODE root)
{
        if ( root == NULL ) return;

        postorder(root->llink);          /* visit leftsubtree in postorder*/
        postorder(root->rlink);          /* visit rightsubtree in postorder*/
        printf("%d ", root->info);       /* visit the node */
}
```

10.7.3 Inorder traversal

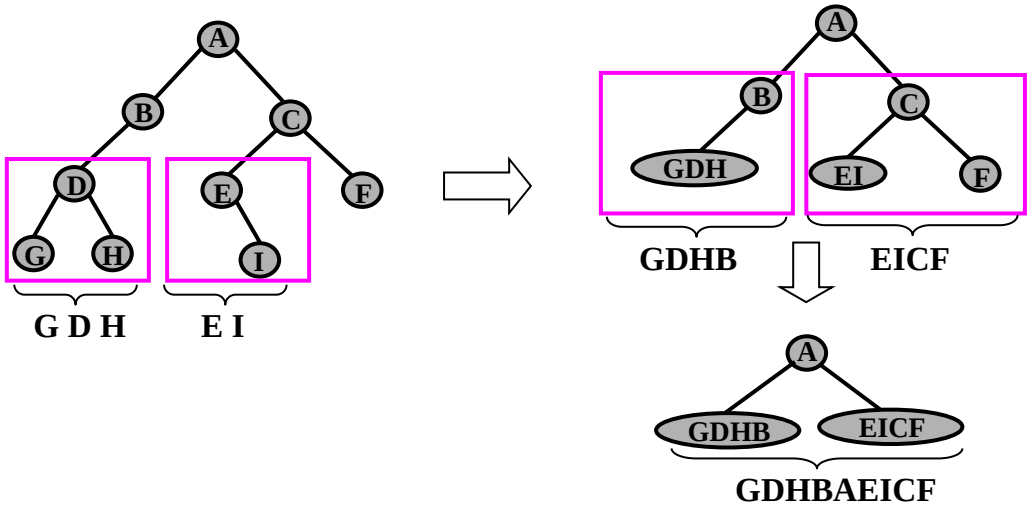
Now, let us see "What is inorder traversing of a binary tree?"

Definition: The inorder traversal of a binary tree can be recursively defined as follows:

1. Traverse the **L**eft subtree in inorder **[L]**
2. Process the root **N**ode **[N]**
3. Traverse the **R**ight subtree in inorder **[R]**



For example, let us traverse the following tree in preorder.



The C function to traverse the tree in inorder is shown below:

Example 10.3: Function to traverse the tree in inorder

```
void inorder(NODE root)
{
        if ( root == NULL ) return;

        inorder(root->llink);                   /* visit left-subtree in inorder */
        printf("%d ",root->info);               /* visit the node */
        inorder(root->rlink);                   /* visit right-subtree in inorder */
}
```

Now, let us see "How to print the tree in the tree form?" The tree can be printed in the form of a tree sideways on the screen using converse inorder (RNL) as shown below:

Example 10.4: C function to print the tree in tree form (sideways)

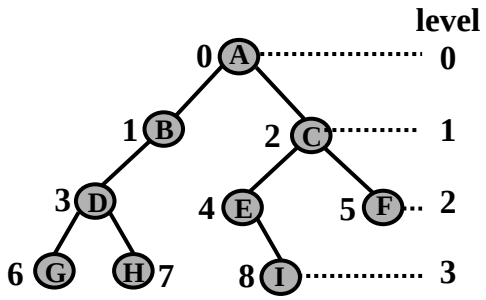
```
void display(NODE root, int level)
{
        int i;

        if (root == NULL) return;

        display(root->rlink, level + 1);

        for (i = 0; i < level; i++)  printf("    ");

        printf("%d\n", root->info);

        display(root->llink, level + 1);
}
```

Note: All tree traversal techniques just discussed are recursive algorithms and uses stack. Now, let us see how to traverse the tree that uses queue data structure.

10.7.4 Level order traversal

Now, let us see "What is level order traversing of a binary tree?"

Definition: The nodes in a tree are numbered starting with the root on level 0, continuing with the nodes on level 1, level 2 and so on. Nodes on any level are numbered from left to right. Visiting the nodes using the ordering suggested by the node numbering is called level order traversing.



Output of level order traversing:

A B C D E F G H I

Design: Insert the root node identified by variable *root* into queue. This is done using the statement:

```
front = 0, rear = -1;
q[++rear] = root;
```

Step 1: Delete a node from the front end and visit that node by displaying using the statements:

```
cur = q[front++];
printf("%d\n", cur->info);
```

Step 2: If node visited in step 1 has a left subtree, insert that node into queue. This is done using the statement:

```
if (cur->llink != NULL)
        q[++rear] = cur->llink;
```

Step 3: If node visited in step 1 has a right subtree, insert that node into queue. This is done using the statement:

```
if (cur->rlink != NULL)
        q[++rear] = cur->rlink;
```

Step 4: Repeat through step 1 as long as queue is not empty.

Now, the complete function can be written as shown below:

Example 10.5: C function to print the tree using level order traversal

```
void level_order(NODE root)
{
        NODE    q[MAX_QUEUE], cur;
        int     front = 0, rear = -1;   // queue is empty
```

```
        q[++rear] = root;                    // Insert root node into queue

        while (front <= rear)                // As long as queue is not empty
        {
                cur = q[front++];                    // Delete from queue
                printf("%d ", cur->info);            // visit the node

                if (cur->llink != NULL)              // Insert left subtree into queue
                        q[++rear] = cur->llink;

                if (cur->rlink != NULL)              // Insert right subtree into queue
                        q[++rear] = cur->rlink;
        }
        printf("\n");
}
```

Note: The symbolic constant MAX_QUEUE can be defined as shown below:

```
#define MAX_QUEUE  10
```

10.8 Iterative traversals of binary tree

This chapter deals with, mostly of problems involving recursion. In this section we develop the functions for traversal of trees using iterative technique.

10.8.1 Iterative preorder traversal

We know that in preorder traversal, node is visited first, then the left subtree is traversed in preorder and finally right subtree is traversed in preorder. Visiting the node here is nothing but display the corresponding item in the node. This can be achieved using the statement

```
printf("%d  ",cur->info);
```

only if *cur* points to root node. So, initially *cur* points to the root node. Once the node is visited, next is to traverse the left subtree in preorder. For this descend the tree towards left. Once all the nodes in the left subtree are displayed, it may be required to ascend the parts of the tree to display the nodes in the right subtree. To ascend the tree later, just before descending towards left, push the address of current node *cur* and then descend the tree left. This process has to be repeated till end of left subtree is encountered. The code corresponding to this can be written as

```
while ( cur != NULL )
{
        printf("%d  ",cur->info);          /* Visit the node */
        s[++top] = cur;                    /* push(cur, &top, s) */
        cur = cur->llink;                  /* traverse left */
}
```

In the tree shown in figure below, after executing these statements 100, 50 and 25 will be displayed.

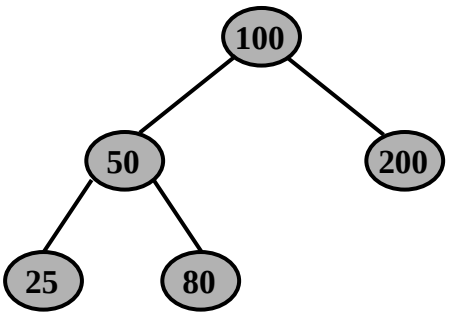


Figure Binary search tree

Once traversing the left subtree in preorder is over, traverse the right subtree in preorder i.e., ascend the tree by popping the address of the node most recently pushed and descend towards right subtree. This is possible only when the stack is not empty. If the stack is empty traversing the tree in preorder is complete and the program terminates. The code corresponding to this is shown below:

```
if ( top != -1 )                    /* If stack is not empty */
{
        cur = s[top--];             /* Obtain the recent node from the stack */
        cur = cur->rlink;           /* traverse right */
}
else
        return;
```

After descending the tree towards right, entire right subtree has to be traversed in preorder. So, the above set of statements has to be repeated. The complete C function for this is shown below:

Example 10.6: Function for iterative preorder traversal

```
void preorder(NODE root)
{
        NODE cur, s[20];
        int  top = -1;

        if ( root == NULL )
        {
                printf("Tree is empty\n");
                return;
        }

        cur = root;

        for (;;)
        {
                while ( cur != NULL )
                {
                        printf("%d  ",cur->info);          /* Visit the node */
                        s[++top] = cur;                    /* push(cur, &top, s) */
                        cur = cur->llink;                  /* traverse left */
                }

                if ( top != -1 )                  /* If stack is not empty */
                {
                        cur = s[top--];           /* Obtain the recent node from the stack */
                        cur = cur->rlink;         /* traverse right */
                }
                else
                        return;
        }
}
```

10.8.2 Iterative inorder traversal

In this traversal technique, first left subtree is traversed in inorder, then the node is visited and finally the right subtree is traversed. As in iterative preorder traversal, traverse the left subtree in inorder by descending the tree towards left until **cur** which initially points to root node is NULL. Before updating the pointer **cur** towards left, push its address so as to descend towards right subtree later. The code corresponding to this can be written as shown below:

```
while ( cur != NULL )
{
        s[++top] = cur;          /* push(cur,&top,s); */
        cur = cur->llink;        /* traverse left*/
}
```

Once the left subtree is traversed, visit the node by popping the most recently pushed node on to the stack and then traverse towards right if the stack is not empty. If the stack is empty, traversing the tree in inorder is over and the procedure is terminated. The code to achieve this can be

```
if (top != -1 )                  /* If stack not empty */
{
        cur = s[top--];          /* Remove recent node from stack */
        printf("%d  ",cur->info); /* visit node */
        cur = cur->rlink;        /* traverse right */
}
else
        return;
```

After traversing towards right, traverse the tree in inorder and repeat the process. This can be achieved by executing all the above statements. The C function to traverse the tree in inorder is shown below:

Example 10.7: Function for iterative inorder traversal

```
/* display the contents of the tree in inorder */
void inorder(NODE root)
{
        NODE cur,s[20];
        int  top = -1;

        if ( root == NULL )
        {
                printf("Tree is empty\n");
                return;
        }

        cur = root;
```

```
        for (;;)
        {
                while ( cur != NULL )
                {
                        s[++top] = cur;         /* push(cur,&top,s); */
                        cur = cur->llink;       /* traverse left*/
                }

                if (top != -1 )                 /* If stack not empty */
                {
                        cur = s[top--];         /* Remove recent node from stack */
                        printf("%d  ",cur->info);     /* visit node */
                        cur = cur->rlink;               /* traverse right */
                }
                else
                        return;
        }
}
```

10.8.3 Iterative postorder traversal

In postorder traversal, traverse the left subtree in postorder, then traverse the right subtree in postorder and finally visit the node. Here also stack is used. But, each node will be stacked twice once during traversing the left-subtree and another while traversing towards right. Just to distinguish that traversing the right subtree is over, we set the **flag** to –1. So, check the **flag** field of the corresponding node. If **flag** field of a node is –ve, right subtree is traversed and one can visit the node. Otherwise traverse the right subtree. This process is repeated till the stack is empty. The C function for this is shown below:

Example 10.8: Function for iterative postorder traversal

```
void postorder(NODE root)
{
        struct stack
        {
                NODE address;
                int      flag;
        };

        NODE        cur;
```

```
        struct stack    s[20];
        int             top = -1;

        if ( root == NULL )                     /* Check for empty stack */
        {
                printf("Tree is empty\n");
                return;
        }

        cur = root;

        for (;;)
        {
                while ( cur != NULL )
                {
                        top++;
                        s[top].address = cur;           /* push(cur,&top,s) */
                        s[top].flag = 1;
                        cur = cur->llink;               /* traverse left */
                }

                /* -ve values on stack indicate right subtree is
                * traversed and the node can be visited
                */
                while ( s[top].flag < 0 )
                {
                        cur = s[top].address;           /* cur = pop(&top,s); */
                        top--;

                        printf("%d ",cur->info);        /* visit the node */

                        if ( top == -1 ) return;
                }

                /* ascend to traverse the right subtree */
                cur = s[top].address;   /* cur = pop(&top,s); */
                cur = cur->rlink;
                s[top].flag = -1;       /*-ve indicates  right  subtree is traversed */
        }
}
```

10.9 Threaded binary trees

Now, let us see "What are the disadvantages of binary trees?" The various disadvantages of the binary tree are shown below:

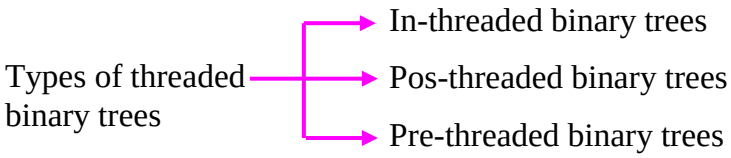
- In a binary tree, more than 50% of **link** fields have \0 (null) values and more memory space is wasted by storing \0 (null) values
- Traversing a tree with binary tree is time consuming. This is because, the traversal of a tree either uses implicit stack (in case of recursive programs) or explicit stack (in case of iterative programs). Whatever it is stack is must. Most of the time is spent in pushing and popping activities during traversing.
- Computations of predecessor and successor of given nodes is time consuming
- In binary trees, only downward movements are possible

All these disadvantages can be overcome using **threaded binary tree.** Now, let us see "What is threaded binary tree?"

Definition: In a binary tree, more than 50% of **link** fields have \0 (null) values and more space is wasted by the presence of \0 (null) values. These **link** fields which contains \0 characters can be replaced by address of some nodes in the tree which facilitate upward movement in the tree. These extra **links** which contains addresses of some nodes (pointers) are called threads and the tree is termed as threaded binary tree.

In general, a threaded binary tree is a binary tree which contains threads (i.e., addresses of some nodes) which facilitate upward movement in the tree.

Now, let us see "What are the various types of threaded binary trees?" A binary tree is threaded based on the traversal technique. Since there are three traversal techniques, threaded binary trees are classified into three types as shown below:



Now, let us see "What is in-threaded binary trees?"

Definition: In a binary tree, if **llink** (left link) of any node contains \0 (null) and if it is replaced by address of the inorder predecessor, then the resulting tree is called left in-threaded binary tree. In a binary tree, if **rlink** (right link) of a node is NULL and if it is replaced by address of inorder successor, the resulting tree is called right in-threaded binary tree. An in-threaded binary tree or inorder threading of a binary tree is the once which is both left in-threaded and right in-threaded.

For example, consider the binary tree shown below:

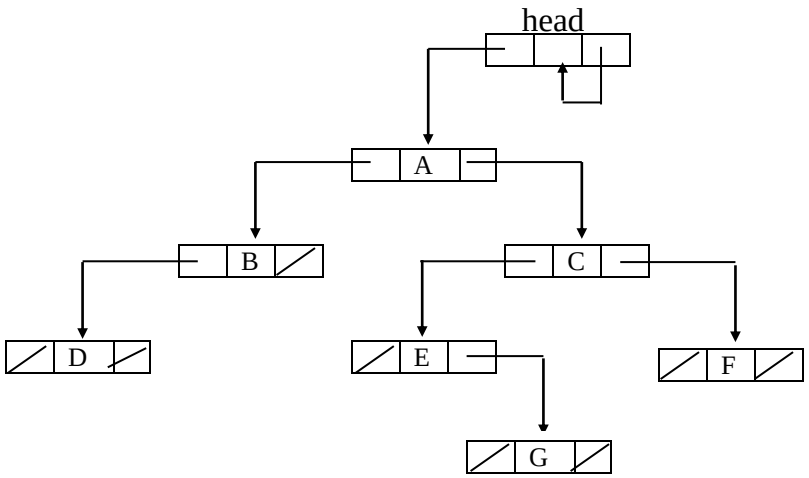


Fig. 10.3 Binary tree with a header node

In the above binary tree, if the right link of a node is NULL and if it is replaced by the address of the inorder successor as shown using dotted lines in fig.10.4, then the tree is said to be **right in-threaded** binary tree.

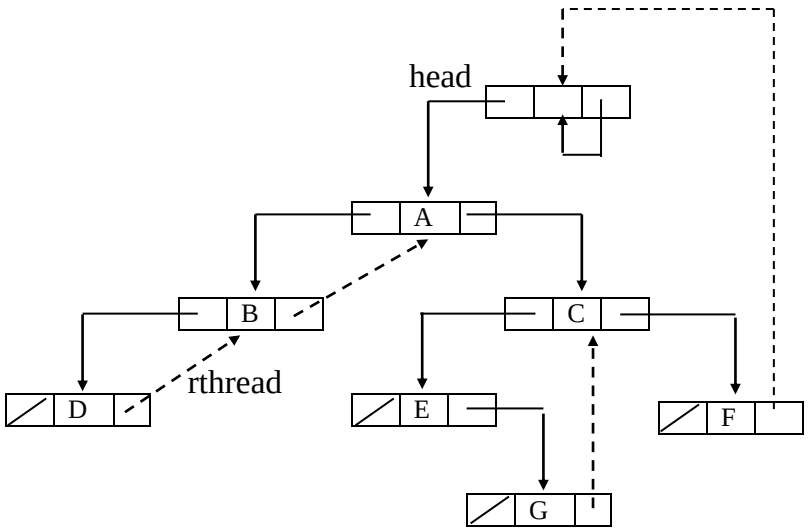


Figure 10.4: Right in-threaded binary tree

Now, let us see "How to implement right in-threaded binary tree in C language?"

To implement a right in-thread an extra field *rthread* is used. If *rthread* is 1 the corresponding right link represents a thread and if *rthread* is 0 the right link represents an ordinary link connecting the right subtree. Thus a node can be defined as shown below:

```
struct node
{
        int info;
        struct node *llink;    /* Pointer to the left subtree */
        struct node *rlink;    /* Pointer to the right subtree */
        int  rthread;          /* 1 indicates the presence of a thread*/
                               /* 0 indicates the absence of a thread */
};

typedef struct node* NODE;
```

For a binary tree shown in figure 10.3, if the left field of a node is NULL and is replaced by the inorder predecessor as shown in fig.10.5, then the tree is said to be **left in-threaded** binary tree. Here also an extra field *lthread* is used where 1 indicates the presence of a thread and 0 indicates ordinary link connecting the left subtree.

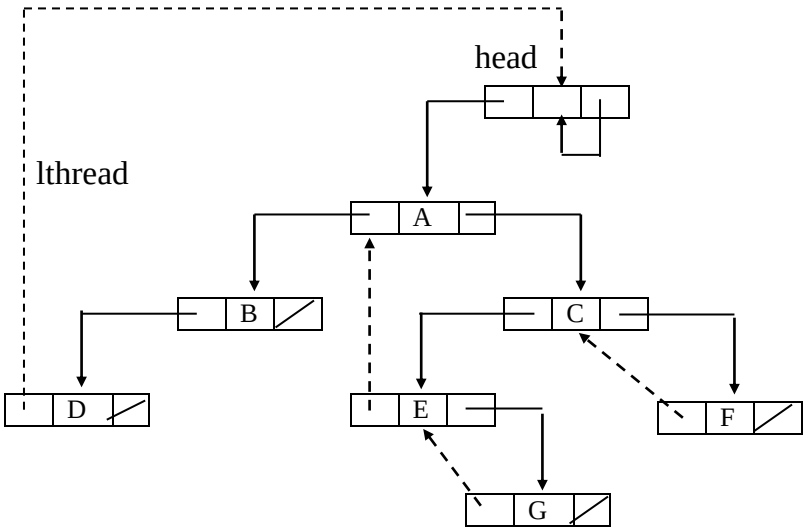


Figure 10.5 Left in threaded binary tree

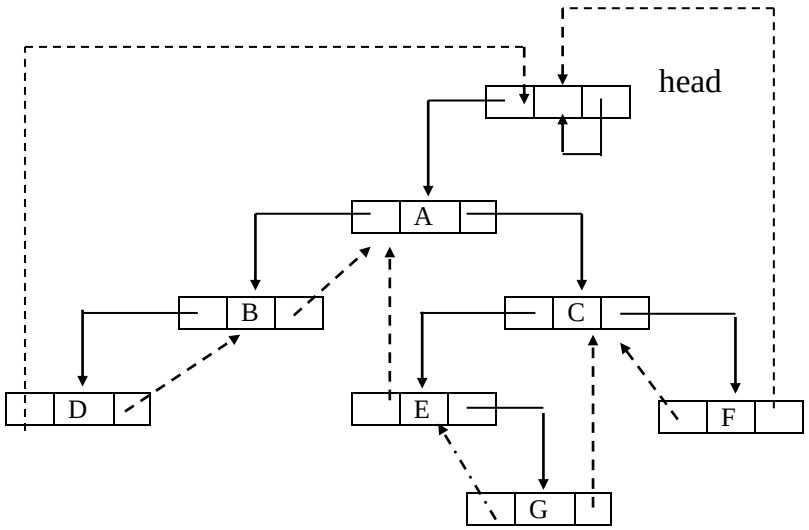


Figure 10.6 In-threaded binary tree (inorder threading of binary tree)

Thus a node can be defined as

```
struct node
{
        int             info;
        struct node     *llink;         /* Pointer to the left subtree */
        struct node     *rlink;         /* Pointer to the right subtree */
        int             lthread;        /* 1 indicates a thread else ordinary link */
};
typedef struct node* NODE;
```

An **inorder threading** of a binary tree or **in-threaded** binary tree is one which is left in-threaded and right in-threaded and is shown in fig.10.6. Here two extra fields *lthread* and *rthread* as defined earlier are used and the structure of the node can be defined as shown below:

```
struct node
{
        int info;
        struct node *llink;     /* Pointer to the left subtree */
        struct node *rlink;     /* Pointer to the right subtree */
        int  lthread;           /* 1 indicate a thread else ordinary link */
        int  rthread;
};
```

It is desirable to have a **header node** so as to solve the problems more easily. In this case, the header node serves as the predecessor of the first node and successor of the last node. This imposes a circular structure on the tree. When the tree is empty, the header node is represented as shown in fig.10.7. The tree is attached always to the left field of the header node.

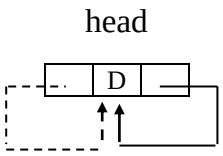


Fig.10.7 Empty tree with a header node

Now let us see, "How to find the inorder successor and predecessor?" and "How to traverse the tree in inorder?"

10.9.1 Inorder successor for right in-thread

Consider the right in-threaded binary tree shown in fig.10.4. Given a node X, if *rthread* is 1, which indicates the presence of thread, its *rlink* gives the address of the inorder successor. If *rthread* is 0, *rlink* contains the address of the right subtree. Left most node in the right subtree is the inorder successor. The C function is shown below:

Example 10.9: Function to find the inorder successor

```
NODE inorder_successor(NODE x)
{
        NODE temp;

        temp = x->rlink;

        if ( x->rthread == 1 ) return temp;

        /*Obtain the left most node in the right subtree */
        while ( temp->llink != NULL )
                temp = temp->llink;

        return temp;
}
```

10.9.2 Inorder traversal of right in-thread

The C function to traverse the tree in inorder is straightforward and the reader is required to trace this program. The C function is shown below:

Example 10.10: Function to traverse the tree in inorder

```
void inorder(NODE head)
{
        NODE temp;

        if ( head->llink == head )
        {
                printf("Tree is empty\n");
                return;
        }
        printf("The inorder traversal of the tree is\n");
        temp = head;

        for (;;)
        {
                temp = inorder_successor(temp);

                if ( temp == head ) return;

                printf("%d  ",temp->info);
        }
}
```

Now, let us see "What is pre-threaded binary trees?"

Definition: In a binary tree, if **llink** (left link) of any node contains \0 (null) and if it is replaced by address of the preorder predecessor, then the resulting tree is called left pre-threaded binary tree.

In a binary tree, if **rlink** (right link) of a node is NULL and if it is replaced by address of preorder successor, the resulting tree is called right pre-threaded binary tree.

A pre-threaded binary tree or preorder threading of a binary tree is the once which is both left pre-threaded and right pre-threaded.

Now, let us see "What is post-threaded binary trees?"

Definition: In a binary tree, if **llink** (left link) of any node contains \0 (null) and if it is replaced by address of the postorder predecessor, then the resulting tree is called left post-threaded binary tree. In a binary tree, if **rlink** (right link) of a node is NULL and if it is replaced by address of postorder successor, the resulting tree is called right post-threaded binary tree. A post-threaded binary tree or postorder threading of a binary tree is the one which is both left post-threaded and right post-threaded.

10.9.3 Advantages and disadvantages

Now, let us see "What are the advantages of threaded binary trees?" The various advantages of the binary tree are shown below:

- In a binary tree, more than 50% of **link** fields have \0 (null) values and more memory space is wasted by storing \0 (null) values. This wastage of memory space is avoided by storing addresses of some nodes.
- Traversing of a threaded binary tree is very fast. This is because, it does not use implicit or explicit stack.
- Computations of predecessor and successor of given nodes is very easy and efficient
- Any node can be accessed from any other node. Using threads, upward movement is possible and using **links** downward movement is possible. Thus, in a threaded binary tree, we can move in either directions. This is not possible in un-threaded binary trees.
- Even though insertion into a threaded binary tree and deletion from a threaded binary are time consuming operations, they are very easy to implement.

Now, let us see "What are the disadvantages of threaded binary trees?" The various disadvantages of threaded binary trees are shown below:

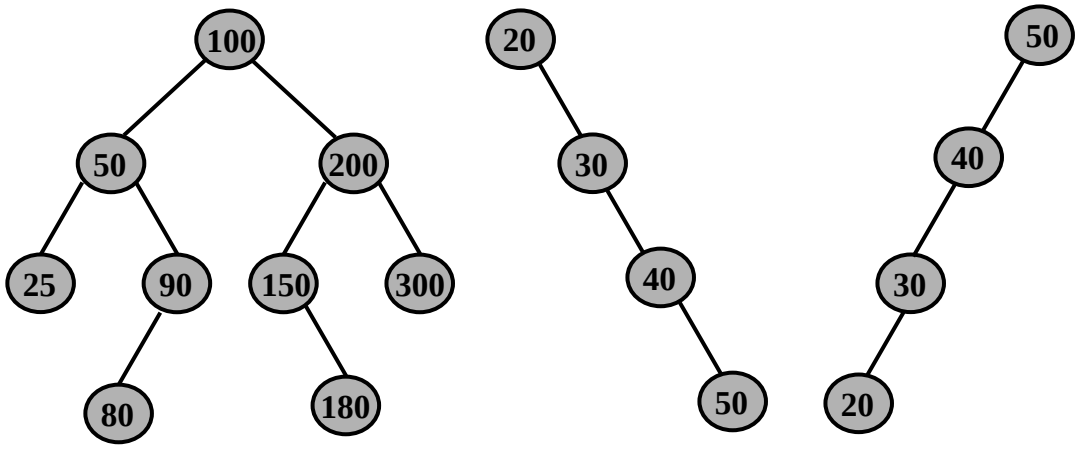
- Here extra fields are required to check whether a link is a thread or not and hence occupy more memory when compared with un-threaded binary trees.
- Insertion and deletion of a node consumes more time than its counter part because many fields have to be modified.

10.10 Binary search tree (BST)

Now, let us see "What is a binary search tree?"

Definition: A binary search tree is a binary tree in which for each node say **x** in the tree, elements in the left-subtree are less than *info(x)* and elements in the right subtree are greater than *info(x)*. Every node in the tree should satisfy this condition, if left

subtree or right subtree exists. A binary search tree can be empty. For example, the figures below shows some of the binary search trees.



(a) **(b)** **(c)**

Figure Binary search trees

Note: Traversal of a binary search tree is same as traversal of a binary tree as explained in section 10.7. Other common operations performed on binary search trees are:

- **Insertion** – Insert an item into binary search tree
- **Searching** – Search for a specific item in the tree
- **Deletion** – Deleting a node from a given tree.

10.10.1 Insertion

Now, let us see "How to insert an element into a binary search tree?"

Design: A binary search tree can be created by repeatedly inserting items into the tree as shown below:

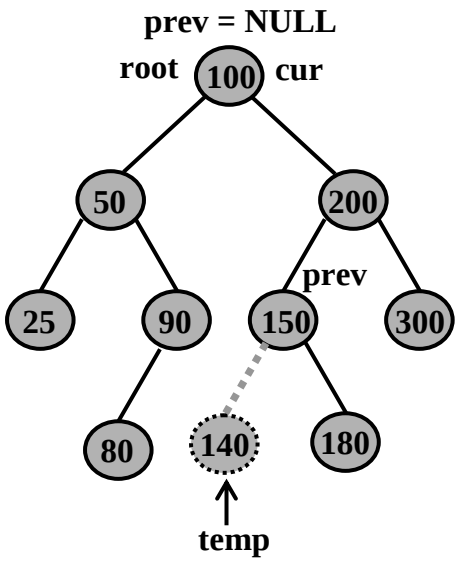
Step 1: The *item* read from the keyboard must be stored in a node. This is achieved using malloc() function with the help of getnode() function defined in example 8.2. The equivalent code can be written as shown below:

```
temp = getnode();
temp->info = item;
temp->llink = NULL;
temp->rlink = NULL;
```

Step 2: If tree does not exist, then return the above node itself as the root node. The code can be written as shown below:

if (root == NULL) **return** temp;

Step 3: If tree already exists, *root* will not be NULL. In such case, we have to insert the node created in step 1 at the appropriate place. We know that in a binary search tree, items towards left subtree of a node *x* will be less than *info(x)* and the items in the right subtree are greater or equal to *info(x).* Consider the binary search tree shown in the figure. Let *temp* with *info* of 140 is the node to be inserted.



Now, we have to find the appropriate position to insert. Let us assume that the variable *root* always points to the root node of the tree.

Since search starts from the root, we use two pointers *cur* and *prev* to find the appropriate position in the tree. The pointer variable *prev* always points to parent of *cur* node. Initially *cur* points to root node and *prev* points to NULL as shown below:

```
prev = NULL;
cur  = root;
```

Now, as long as the item is less than *info(cur),* keep updating pointer variable *cur* towards left using the statement:

```
cur = cur->llink;
```

Otherwise, update *cur* towards right using the statement:

```
cur = cur->rlink;
```

Before updating *cur* towards left or right, save its address in *prev* so that the pointer variable *prev* always points to the parent node of *cur.* Now, the code can be written as shown below:

```
prev = cur;
if (item < cur->info )
        cur = cur->llink;
else
        cur = cur->rlink;
```

Note that while updating *cur* towards left or right, it may become NULL. In such case, we have found the appropriate position to insert and stop updating *cur*. So, the above code can be repeatedly executed as long as *cur* is not NULL. The code for this can be written as shown below:

```
while (cur != NULL)
{
        prev = cur;
        if (item < cur->info )
                cur = cur->llink;
        else
                cur = cur->rlink;
}
```

Once *cur* points to NULL, insert the node *temp* towards *left(prev)* if item is less than *info(prev)*, otherwise insert towards right. This can be done using the following code:

```
if ( item < prev->info )
        prev->llink = temp;
else
        prev->rlink = temp;
```

Finally, we return the address of the root node using the statement:

```
        return root;
```

Now, the complete code can be written as shown below:

Example 10.11: To insert an item into a binary search tree (duplicate elements)

```
NODE insert(int item, NODE root)
{
        NODE temp,cur,prev;

        temp = getnode();                   /* Create a node and */
        temp->info = item;                  /* Copy appropriate data */
        temp->llink = NULL;
        temp->rlink = NULL;

        if ( root == NULL ) return temp;    /* Insert a node for the first time */
```

```
        prev = NULL;                    /* find the position to insert */
        cur  = root;

        while ( cur != NULL )
        {
                prev = cur;             /* Obtain parent position */
                if (item < cur->info )
                        cur = cur->llink;   /* Obtain left child position */
                else                        /* or */
                        cur = cur->rlink;   /* Obtain right child position */
        }

        if ( item < prev->info )        /* If node to be inserted < parent */
                prev->llink = temp;     /* Insert towards left of the parent */
        else
                prev->rlink = temp;     /* else, insert towards right of the parent */

        return root;                    /* Return the root of the tree */
}
```

Note: In the above function, observe that if an item is less than the node, it is inserted towards left. Otherwise, it is inserted towards right. So, duplicate items are also inserted towards right. So, to avoid duplicate elements, we check whether the item is same as *info* of a node. If so, do not insert. The code can be written as shown below:

```
        prev = cur;                     /* Obtain parent position */

        if ( item == cur->info )        /* do not insert duplicate item */
        {
                printf("Duplicate items not allowed\n");
                free(temp);

                return root;
        }

        if (item < cur->info )
                cur = cur->llink;       /* Obtain left child position */
        else                            /* or */
                cur = cur->rlink;       /* Obtain right child position */
```

Rest of the code remains same. Now, the complete function that avoids insertion of duplicate elements can be written as shown below:

Example 10.12: To insert an item into a binary search tree (No duplicate items)

```
NODE insert(int item, NODE root)
{
        NODE temp, cur, prev;

        temp = getnode();                   /* Create a node and */
        temp->info = item;                  /* Copy appropriate data */
        temp->llink = NULL;
        temp->rlink = NULL;

        if ( root == NULL ) return temp;    /* Insert a node for the first time */

        prev = NULL;                        /* find the position to insert */
        cur  = root;

        while ( cur != NULL )
        {
                prev = cur;                 /* Obtain parent position */

                if ( item == cur->info )    /* do not insert duplicate item */
                {
                        printf("Duplicate items not allowed\n");
                        free(temp);
                        return root;
                }

                if (item < cur->info )
                        cur = cur->llink;   /* Obtain left child position */
                else                                /* or */
                        cur = cur->rlink;   /* Obtain right child position */
        }

        if ( item < prev->info )            /* If node to be inserted < parent */
                prev->llink = temp;         /* Insert towards left of the parent */
        else
                prev->rlink = temp;         /* else, insert towards right of the parent */

        return root;                        /* Return the root of the tree */
}
```

10.10.2 Searching

Now, let us see "How to search for an element in a binary search tree?"

Design: The following steps are followed to search for an item in binary search tree.

Step 1: Check for empty tree: If tree does not exist, we return NULL indicating search is unsuccessful. The code for this can be written as:

```
if (root == NULL) return NULL;
```

Step 2: Search for the node in left subtree or right subtree. Same as program discussed in example in previous section. Only change is that once *cur* becomes NULL control comes out of the while-loop and we say *key* is not present in the tree and return NULL indicating search is unsuccessful. The complete algorithm is written in C as shown below:

Example 10.13: Function to search for an item in binary search tree using iteration

```
NODE search(int item, NODE root)
{
        NODE cur;

        if (root == NULL) return NULL;          /* empty tree */

        cur = root;
        while ( cur != NULL )                   /* search for the item */
        {
                if (item == cur->info) return cur;  /* If found return the node */

                if ( item < cur->info )
                        cur = cur->llink;       /* Search towards left */
                else
                        cur = cur->rlink;       /* Search towards right */
        }

        return NULL;                            /* Key not found */
}
```

The recursive function to search for an item is shown below:

Example 10.14: Function to search for an item in BST using recursion

```
NODE search(int item, NODE root)
{
    if ( root == NULL ) return root;              /* Item not found */

    if ( item == root->info ) return root;        /* Item found */

    if ( item < root->info )                      /* Search recursively left side */
            return search(item, root->llink);

    return search(item, root->rlink);             /* Search recursively right side */
}
```

The complete program for creating a tree, traversing and searching can be written as shown below:

Example 10.15: C program to create, traverse and to search for an item in the tree

```
#include <stdio.h>
#include <stdlib.h>

struct node
{
    int             info;
    struct node     *llink;
    struct node     *rlink;
};

typedef struct node* NODE;

/* Include: Example 8.2: function to create a node using malloc() */
/* Include: Example 10.1: Display the contents of the tree in preorder */
/* Include: Example 10.2: Display the contents of the tree in postorder */
/* Include: Example 10.3: Display the contents of the tree in inorder */
/* Include: Example 10.4: C function to print the tree in tree form */
/* Include: Example 10.11: Function to insert duplicate items into BST */
            or
/* Include: Example 10.12: Function to insert item (no-duplicate) into BST */
/* Include: Example 10.13: Function to search for an item in a tree */
            or
/* Include: Example 10.14: Function to search for an item in a tree */
```

```
void main()
{
        NODE root, cur;
        int     choice, item;
        root = NULL;
        for (;;)
        {
                printf("1:Insert        2: Preorder\n");
                printf("3:Postorder   4: Inorder\n”);
                printf(“5: Search      6:Exit\n");
                printf("Enter the choice\n");
                scanf("%d", &choice);
                switch(choice)
                {
                        case 1:
                                printf("Enter the item to be inserted\n");
                                scanf("%d", &item);
                                root = insert(item, root);
                                break;
                        case 2:
                                if ( root == NULL )
                                {
                                        printf("Tree is empty\n");
                                        break;
                                }
                                printf(“The given tree in tree form is\n”);
                                display(root, 1);
                                printf("Preorder traversal is\n");
                                preorder(root);
                                printf("\n");
                                break;
                        case 3:
                                if ( root == NULL )
                                {
                                        printf("Tree is empty\n");
                                        break;
                                }
```

```
                printf("The given tree in tree form is\n");
                display(root, 1);

                printf("Postorder traversal is\n");
                postorder(root);
                printf("\n");

                break;

        case 4:
                if ( root == NULL )
                {
                        printf("Tree is empty\n");
                        break;
                }

                printf("The given tree in tree form is\n");
                display(root, 1);

                printf("Inorder traversal is\n");
                inorder(root);
                printf("\n");

                break;

        case 5:
                printf("Enter the item to be searched\n");
                scanf("%d",&item);

                cur = search(item, root);

                if (cur == NULL)
                        printf("Item not found\n");
                else
                        printf("Item found\n");

                break;

        default: exit(0);
        }
    }
}
```

10.11 Other operations

Various other useful operations are shown below.

- find maximum – to return maximum item in the tree
- find minimum – to return minimum item in the tree
- to find height of the tree
- to count the number of nodes
- to count the leaf nodes
- deletion – deleting a node from the tree

10.11.1 To find maximum value in a binary search tree

Let us see "How to find maximum value in a binary search tree?" Given a binary search tree, a node with maximum value is obtained by traversing and obtaining the right most node in the tree. If there is no right subtree then return **root** itself as the node containing the ***item*** with highest value. The corresponding C function is shown below:

Example 10.16: Function to return the address of highest item in BST

```
NODE maximum(NODE root)
{
        NODE cur;

        if ( root == NULL ) return root;

        cur = root;
        while ( cur->rlink != NULL )
        {
                cur = cur->rlink;               /* obtain right most node in BST */
        }

        return cur;
}
```

10.11.2 To find minimum value in a BST

Let us see "How to find minimum value in a binary search tree?" Given a binary search tree, a node with least value is obtained by traversing and obtaining the left most node in the tree. If there is no left subtree then return **root** itself as the node containing an ***item*** with least value. The corresponding C function is shown below:

Example 10.17: Function to return the address of least item in BST

```
NODE minimum(NODE root)
{
        NODE cur;
        if ( root == NULL ) return root;
        cur = root;
        while ( cur->llink != NULL )
                cur = cur->llink;              /* obtain left most node in BST */
        return cur;
}
```

10.11.3 Height of tree

Let us see "How to find the height of the tree?"

Height of the tree is nothing but the maximum level in a tree. The recursive definition to compute the height of the tree is shown below:

$$\text{Height(root)} = \begin{cases} -1 & \text{if root == NULL} \\ 1 + \max(\text{height(root->llink)}, \text{height(root->rlink)}) & \text{otherwise} \end{cases}$$

The corresponding C function to find the height of the tree is shown below:

Example 10.18: Function to find the height of the tree

```
/* function to find maximum of two numbers */
int max(int a, int b)
{
        return ( a > b ) ? a : b;
}

/* Function to find the height of the tree */
int height(NODE root)
{
        if ( root == NULL ) return -1;
        return 1 + max( height(root->llink), height(root->rlink) );
}
```

10.11.4 Count nodes in a tree

Now, let us see "How to count the nodes in a tree?"

The number of nodes in the tree is obtained by traversing the tree in any of the traversal technique and increment the counter whenever a node is visited. The variable ***count*** can be a global variable and it is initialed to zero before calling this function. In this example, the inorder traversal is used to visit each node. The C function to obtain the number of nodes in a tree is shown below:

Example 10.19: Function to count the number of nodes in a tree

```
void count_node(NODE root)
{
        if ( root = = NULL ) return;
        count_node(root->llink);
        count++;
        count_node(root->rlink);
}
```

10.11.5 Count leaves in a tree

Now, let us see "How to compute the number of leaves in a tree?"

As in the previous case visit each node in a given tree. Whenever a leaf is encountered update ***count*** by one. A leaf or a terminal node is one, which has an empty left and empty right child. The variable ***count*** can be a global variable and it is initialed to zero to start with. The function to count the ***leaves*** in a binary tree is shown below:

Example 10.20: Function to count the leaves or terminal nodes in a tree

```
void count_leaf(NODE root)
{
        if ( root == NULL ) return ;
        count_leaf(root->llink);              /* Traverse recursively towards left */

        /* if a node has empty left and empty right child ? */
        if (root->llink == NULL && root->rlink == NULL ) count++;
        count_leaf(root->rlink);              /* Traverse recursively towards right */
}
```

10.11.6 Delete a node from the tree

It is very important to remember that once the node is deleted, the ordering of the tree should be maintained i.e., even after deleting a node, elements in the left subtree should be lesser and elements in the right subtree should be greater or equal. Let us see "What are the activities to be performed to delete a node?"

The various activities to be performed are shown below:

Case 1: An empty left subtree and nonempty right subtree or an empty right subtree and nonempty left subtree **Note:** A node having empty left child and empty right child is also deleted using this case.

Consider the figures shown below where *cur* denotes the node to be deleted and in both cases one of the subtrees is empty and the other is non-empty. The node identified by *parent* is the parent of node *cur*. The non-empty subtree can be obtained and is saved in a variable *q*. The corresponding code is:

```
if ( cur->llink == NULL)          /* If left subtree is empty */
        q = cur->rlink;           /* non empty right subtree  is saved*/
else if ( cur->rlink == NULL )    /* If right subtree is empty */
        q = cur->llink;           /* non empty left subtree  is saved*/
```

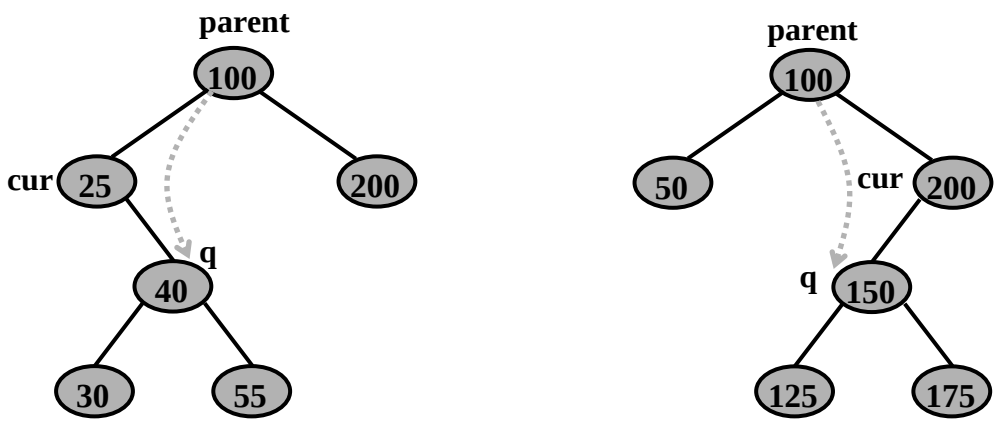


Figure To delete a node identified by **cur** from the tree

Case 2: Non empty left subtree and non-empty right subtree: Consider the figures shown below where *cur* denotes the node to be deleted. In both cases both the subtrees are non-empty. The node identified by *parent* is the parent of the node *cur*.

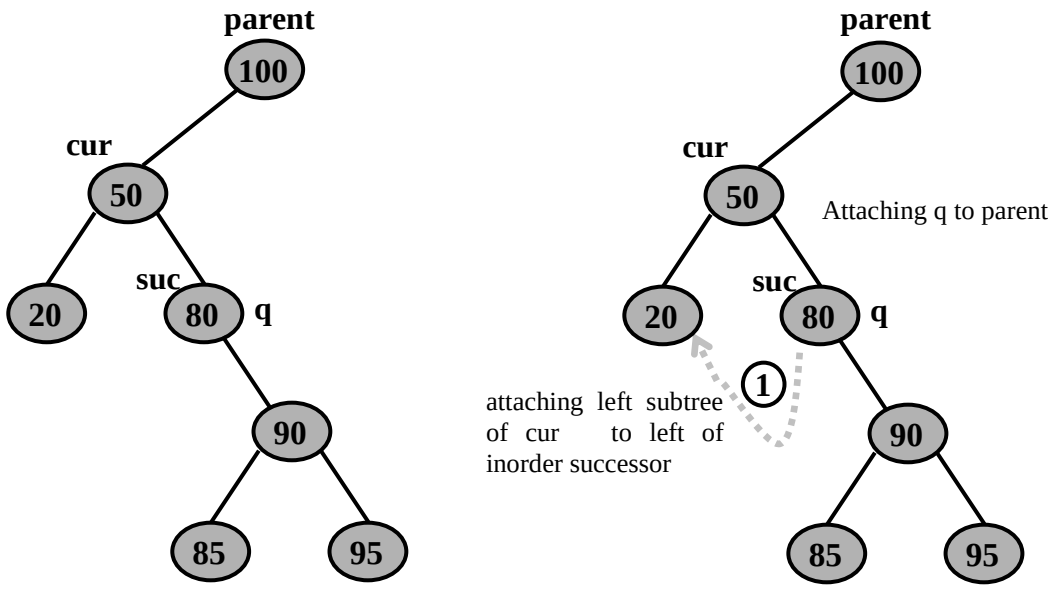


Figure To delete a node **cur** from the tree

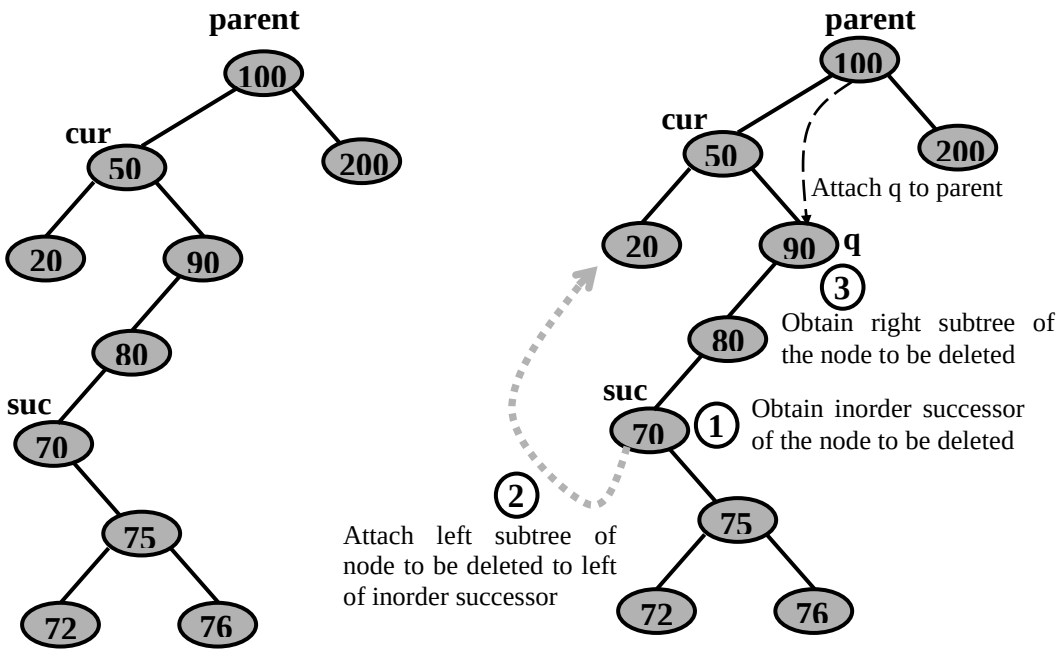


Fig. To delete node **cur** from the tree

The node can be easily deleted using the following procedure in sequence:

Step 1: Find the inorder successor of the node to be deleted. The corresponding code is shown below:

```
suc = cur->rlink;              /* Inorder successor always lies towards right */
while ( suc->llink != NULL )   /* and immediately keep traversing left */
{
        suc = suc->llink;
}
```

Step 2: Attach left subtree of the node to be deleted to the left of successor of the node to be deleted. The corresponding code is:

```
suc->llink = cur->llink;       /* Attach left of node to be deleted to left of
                                  successor of the node to be deleted */
```

Step 3: Obtain the right subtree of the node to be deleted. The corresponding code is:

```
q = cur->rlink;          /* Right subtree is obtained */
```

Attach the node q to parent: Now, after case 1 or case 2, it is required to attach the right subtree of the node to be deleted to the parent of the node to be deleted. If parent of the node to be deleted does not exist, then return *q* itself as the root node using the statement:

```
if (parent == NULL) return q;
```

If parent exists for the node to be deleted, attach the subtree pointed to by *q*, to the parent of the node to be deleted. In this case, attach *q* to *parent* based on the direction. If *cur* is the left child, attach *q* to *left(parent)* otherwise attach *q* to *right(parent)*. This can be achieved by using the following statement

```
/* connecting parent of the node to be deleted to q */
if ( cur == parent->llink )
        parent->llink = q;
else
        parent->rlink = q;
```

Once the node *q* is attached to the *parent*, the node pointed to by *cur* can be deleted and then return the address of the root node. The corresponding statements are:

```
free (cur);
return root;
```

All these statements have been written by assuming the node to be deleted *cur* and its *parent* node is known. So, just before deleting, search for the specified node, obtain

its parent and then delete the node. The complete function to delete an item from the tree is shown below:

Example 10.21: Function to delete an item from the tree

```
NODE delete_item(int item, NODE root)
{
        NODE cur ,parent, suc, q;

        if ( root == NULL )
        {
                printf("Tree is empty! Item not found\n");
                return root;
        }

        parent = NULL;
        cur = root;             /*obtain node to be deleted, its parent */

        while ( cur != NULL)
        {
                if (item == cur->info) break;

                parent = cur;

                if ( item < cur->info)
                        cur = cur->llink
                else
                        cur->rlink;
        }
        if ( cur == NULL )
        {
                printf("Item not found\n");
                return root;
        }

        /* Item found and delete it */          /*  CASE 1  */
        if ( cur->llink == NULL)                /* If left subtree is empty */
                q = cur->rlink;                 /* obtain non empty right subtree */
        else if ( cur->rlink == NULL )          /* If right subtree is empty */
                q = cur->llink;                 /* obtain non empty left subtree */
```

```
        else
        {                                       /* CASE 2  */
                suc = cur->rlink;               /* obtain the inorder successor */
                while ( suc->llink != NULL )
                        suc = suc->llink;

                suc->llink = cur->llink;        /* Attach left of node to be deleted to */
                                                /*left of successor  of node to be deleted */

                q = cur->rlink;                 /* Right subtree is obtained */
        }
        if (parent == NULL) return q;       /* If parent does not exist return q as root */

        /* connecting parent of the node to be deleted to q */
        if ( cur == parent->llink )
                parent->llink = q;
        else
                parent->rlink = q;

        free(cur);

        return root;
}
```

10.12 Additional Binary tree operations

By using the definition of a binary tree and recursive versions of inorder, preorder and postorder traversals, we can easily create C functions for other binary tree operations. The various operations considered in this section are:

- Copying a tree – copy one binary tree to other tree
- Test for equality – check whether two tress are equal or not

10.12.1 Copying a tree

The C function to obtain the exact copy of the given tree using recursion is shown below: It is self-explanatory. In this function address of the root node is given and after copying, it returns address of the root node of the new tree.

Example 10.22: Function to get the exact copy of a tree

```
NODE copy(NODE root)
{
        NODE temp;

        if ( root == NULL ) return NULL;            /* Tree does not exist */

        temp = getnode();                           /*Create a new node */

        temp->info = root->info;                    /*copy the information */
        temp->lptr = copy(root->lptr);              /*Copy the appropriate links */
        temp->rptr = copy(root->rptr);

        return temp;                                /* return address of the new root */
}
```

10.12.2 Check whether two trees are equal or not

The C function to check whether two trees are equal or not is shown below. It is self-explanatory.

Example 10.23: Function to get the exact copy of a tree

```
int equal (NODE r1, NODE r2)
{
        if (r1 == NULL && r2 == NULL) return 1;        // two trees are equal

        if (r1 == NULL && r2 != NULL) return 0;        // two trees are not equal

        if (r1 != NULL && r2 == NULL) return 0;        // two trees are not equal

        if (r1->info != r2->info) return 0;            // two trees are not equal

        if (r1->info == r2->info) return 1;            // two trees are equal

        /* Recursively check left subtree of both trees and right subtree of both trees */
        return equal(r1->llink, r2->llink) && equal(r1->rlink, r2->rlink);
}
```

10.13 Expression trees

A sequence of operators and operands that reduces to a single value is called an expression. Let us consider only arithmetic operators such as: +, -, * and /. First, let us see "What is an expression tree?"

Definition: An expression tree is a binary tree that satisfy the following properties:

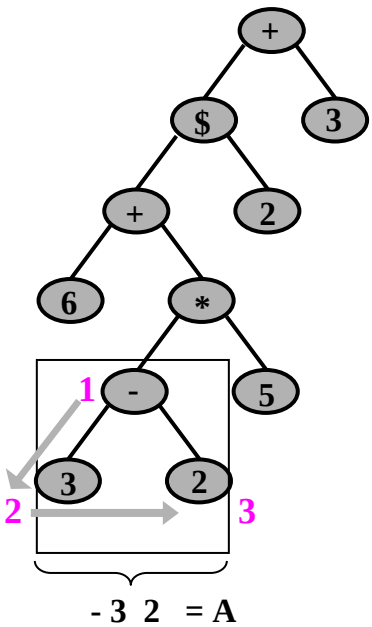
- Any leaf is an operand
- The root and internal nodes and operators
- The subtrees represent sub expressions with root of the subtree as an operator.

Let us see, "How an infix expression can be written using expression tree?" An infix expression consisting of operators and operands can be represented using a binary tree with root as the operator. The left and right subtrees are the left and right operands of that operator. A node containing an operator is not a leaf where as a node containing an operand is a leaf. The tree representation for the infix expression

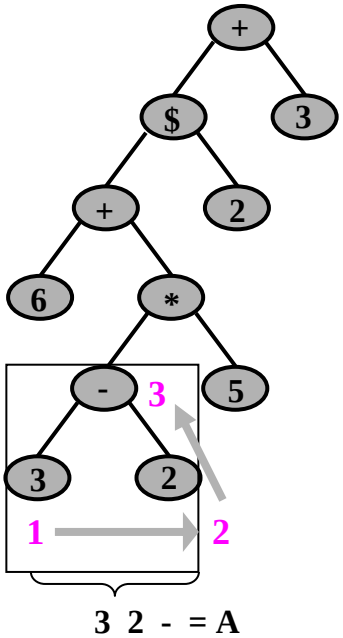
((6+(3-2)*5) ^ 2 + 3)

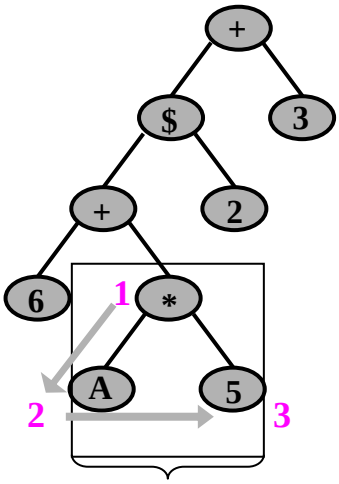
is shown in figure below: Let us traverse the tree in preorder and postorder as shown:

Preorder traversal

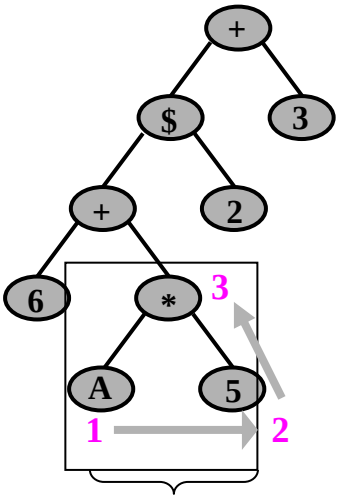


Postorder traversal

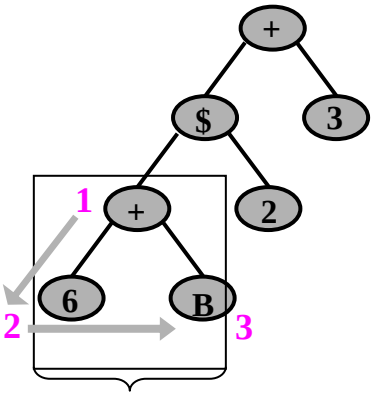




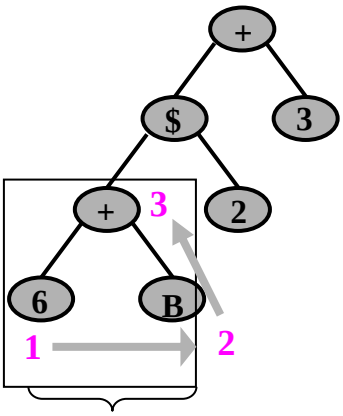
* A 5 = B



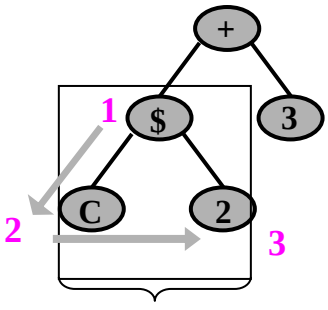
A 5 * = B



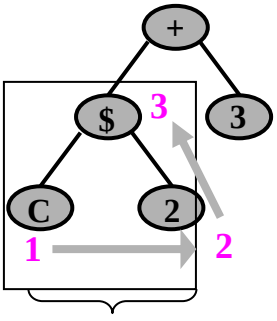
+ 6 B = C



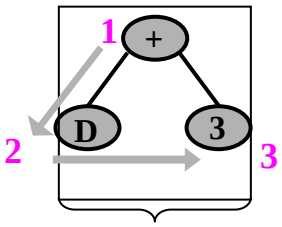
6 B + = C



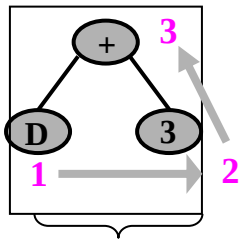
$ C 2 = D



C 2 $ = D



+ D 3
+ $ C 2 3
+ $ + 6 B 2 3
+ $ + 6 * A 5 2 3
+ $ + 6 * - 3 2 5 2 3
(Prefix expression)



D 3 +
C 2 $ 3 +
6 B + 2 $ 3 +
6 A 5 * + 2 $ 3 +
6 3 2 – 5 * + 2 $ 3 +
(Postfix expression)

Note: In an expression tree, we observe that if the expression tree is traversed in preorder we get prefix expression and if we traverse in postorder we get postfix expression. In fact, if we traverse in inorder, we get infix expression without parentheses.

10.13.1 Create binary tree for the postfix expression

Let us see "What is the procedure to obtain an expression tree from the postfix expression?" The procedure to be followed while creating an expression tree using postfix expression is shown below:

- Scan the symbol from left to right.
- Create a node for the scanned symbol.
- If the symbol is an operand push the corresponding node onto the stack.
- If the symbol is an operator, pop one node from the stack and attach to the right of the corresponding node with an operator. The first popped node represents the right operand. Pop one more node from the stack and attach to the left. Push the node corresponding to the operator on to the stack.
- Repeat through step 1 for each symbol in the postfix expression. Finally when scanning of all the symbols from the postfix expression is over, address of the root node of the expression tree is on top of the stack.

The following C function creates a binary tree for the valid postfix expression.

Example 10.24: Function to create an expression tree for the postfix expression

```
NODE creat_tree(char postfix[])
{
        NODE temp, st[20];
        int  i,k;
        char symbol;

        for ( i = k = 0; ( symbol = postfix[i] ) != '\0'; i++)
        {
                temp = getnode();              /* Obtain a node for each operator */
                temp->info = symbol;
                temp->llink = temp->rlink = NULL;

                if( isalnum(symbol) )
                        st[k++] = temp;        /* Push the operand node on to the stack */
                else
                {
                        temp->rlink = st[--k];         /* Obtain 2nd  operand from stack */
                        temp->llink = st[--k];         /* Obtain 1st operand from stack */
                        st[k++] = temp;                /* Push operator node on to stack */
                }
        }

        return st[--k];                        /* Return the root of the expression tree */
}
```

10.13.2 Evaluation of expression

Now, let us see "How to evaluate the expression?" In the expression trees, whenever an operator is encountered, evaluate the expression in the left subtree and evaluate the expression in the right subtree and perform the operation. The recursive definition to evaluate the expression represented by an expression tree is shown below:

$$\text{Eval (root) } = \begin{cases} \text{Eval(root->llink) } \textbf{op} \text{ Eval(root->rlink)} & \text{if root->info is operator} \\ \text{Root->info – '0'} & \text{if root->info is operand} \end{cases}$$

The C function for the above recurrence relation to evaluate the expression represented by an expression tree is shown below:

Example 10.25: Function to evaluate the expression represented as a binary tree

```
float eval(NODE root)
{
        float num;

        switch(root->info)
        {
                case '+': return eval(root->llink) + eval(root->rlink);

                case '-': return eval(root->llink) - eval(root->rlink);

                case '/': return eval(root->llink) / eval(root->rlink);

                case '*': return eval(root->llink) * eval(root->rlink);

                case '$':
                case '^': return pow (eval(root->llink),eval(root->rlink));

                default :
                        if ( isalpha(root->info) )
                        {
                                printf("%c = ",root->info);
                                scanf("%f",&num);
                                return num;
                        }
                        else
                                return root->info - '0';
        }
}
```

The complete C program to evaluate an expression using expression tree is shown below:

Example 10.26: C program to create an expression tree and evaluate

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#define STACK_SIZE 20
```

```
struct node
{
        char            info;
        struct node     *llink;
        struct node     *rlink;
};

typedef struct node* NODE;

/* Function to display in tree form */
void display(NODE root, int level)
{
        int i;

        if (root == NULL) return;

        display(root->rlink, level + 1);

        for (i = 0; i < level; i++)  printf("    ");

        printf("%c\n", root->info);

        display(root->llink, level + 1);
}
```

/* Include: **Example 8.2:** C Function to get a new node from availability list */

/* Include: **Example 10.25:** Function to evaluate the tree */

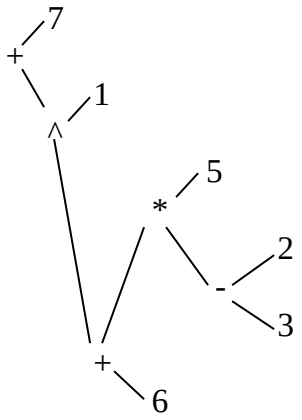
/* Include: **Example 10.24:** Function to create expression tree for postfx exprsin */

```
void main()
{
        char  postfix[40];
        float res;

        NODE  root = NULL;

        printf("Enter the postfix expression\n");
        scanf("%s",postfix);
```

Input

```
Enter postfix expression
632-5*+1^7+
```

```
        root = creat_tree(postfix);

        printf("The expression tree is:\n");
        display(root, 1);

        res = eval(root);

        printf("Result = %f\n", res);
}
```



Output

Result = 18.00000

Output

```
Enter the postfix expression:
abc-d*+e^f+
e = 1, a = 6, b = 3, c = 2, d = 5, f = 7
Result = 18.0000
```

10.14 Tree operations using sequential representation

In section 10.3 we have seen as to how a tree can be represented using an array. The creation of a binary search tree and different traversal techniques using array representation is discussed in this section. The advantages and disadvantages of sequential representation over linked representation is also discussed.

10.14.1 Creation of a binary search tree and traversal techniques

The complete program in C to create the binary search tree and to traverse the tree in inorder, preorder and postorder is shown below:

Example 10.27: C Program to create a tree and traverse the tree array representation

```
#include <stdio.h>
include <process.h>
#define MAX_SIZE 100
```

```
typedef int NODE;

/*function to insert an item */
void insert(int item, NODE a[])
{
        int i;

        i = 0;                                          /* root node */

        while ( i < MAX_SIZE && a[i] != 0 )             /* obtain position where to insert */
        {
                if (item < a[i] )
                        i = 2*i + 1;                    /* Move towards left link */
                else
                        i = 2*i + 2;                    /* Move towards right link */
        }

        a[i] = item;                                    /* insert the item */
}

void inorder(NODE a[], int i)
{
        if ( a[i] == 0 ) return;

        inorder(a, 2*i + 1);            /* Traverse left subtree */
        printf("%d  ",a[i]);            /* Visit the node */
        inorder(a, 2*i + 2);            /* Traverse right subtree */
}

void preorder(NODE a[], int i)
{
        if ( a[i] == 0 ) return;

        printf("%d  ",a[i]);            /* Visit the node */
        preorder(a, 2*i + 1);           /* Traverse left subtree */
        preorder(a, 2*i + 2);           /* Traverse right subtree */
}
```

```
void postorder(NODE a[], int i)
{
        if ( a[i] == 0 ) return;

        postorder(a, 2*i + 1);          /* Traverse left subtree */
        postorder(a, 2*i + 2);          /* Traverse right subtree */
        printf("%d  ",a[i]);            /* Visit the node */
}

void main()
{
        NODE a[MAX_SIZE];
        int  item,choice,i;

        for ( i = 0; i < MAX_SIZE; i++) a[i] = 0;

        for(;;)
        {       printf("1:Insert   2:Inorder\n");
                printf("3:Preorder 4:Postorder\n");
                printf("5:Exit\n");

                printf("Enter the choice\n");
                scanf("%d",&choice);

                switch(choice)
                {       case 1:
                                printf("Enter the item to be inserted\n");
                                scanf("%d",&item);
                                insert(item,a);
                                break;
                        case 2:
                                if ( a[0] == 0 )
                                        printf("Tree is empty\n");
                                else
                                {
                                        printf("The inorder traversal is\n");
                                        inorder(a,0);
                                        printf("\n");
                                }
                                break;
```

```
                case 3:
                        if ( a[0] == 0 )
                                printf("Tree is empty\n");
                        else
                        {
                                printf("The preorder traversal is\n");
                                preorder(a,0);
                                printf("\n");
                        }
                        break;
                case 4:
                        if ( a[0] == 0 )
                                printf("Tree is empty\n");
                        else
                        {
                                printf("The postorder traversal is\n");
                                postorder(a,0);
                                printf("\n");
                        }
                        break;
                default: exit(0);
        }
    }
}
```

10.14.2 Advantages and disadvantages

The sequential representation is simple and it saves space if the tree is complete or almost complete as there is no need to have fields such as left-link, right-link etc. If the tree is not complete or not almost complete binary tree, too much space may be wasted. The index ***i*** used to move downward while doing some operations should not exceed the array bound i.e., MAX_SIZE. This is advantageous only when the number of items in the tree is known in advance.

The linked representation is useful most of the time during repeated deletion or manipulations, which can be easily done by adjusting the links and where the number of items in the tree is unpredictable.

EXERCISES

1. What is a tree? Define the following terms:
   a) Root node b) Child node c) Siblings d) Ancestors
   e) Descendants f) Left descendants g) Right descendants h) Leftsubtree
   i) Right subtree j) Parent k) Degree l) Leaf
   m)Internal nodes n) External nodes o) Level p) Height
   q) Depth
2. What are the different ways of representing a tree?
3. How a tree is represented using lists? What is left child-right sibling representation?
4. What is binary tree representation?
5. Prove that maximum number of nodes on level *i* of a binary tree = 2^i for $i \geq 0$
6. Prove that maximum number of nodes in a binary tree of depth *k* = $2^k - 1$
7. Prove that the number of leaf nodes is equal to number of nodes of degree 2
8. What is the relation between the number of leaf nodes and degree-2 nodes?
9. Define the following: Strictly binary tree, skewed tree, complete binary tree, binary search tree
10. How binary trees are represented? How a tree is represented using linked allocation technique
11. How a tree is represented using static allocation(using arrays) technique?
12. What is the meaning of traversing a tree? What are the different traversal techniques of a binary tree?
13. Write C functions to traverse the tree in Inorder, Preorder, Postorder, level order
14. How to print the tree in the tree form?
15. What is level order traversing of a binary tree?
16. Write C function to insert arbitrary elements into a binary tree
17. What are the disadvantages of binary trees? What is threaded binary tree
18. What are the various types of threaded binary trees?
19. What is in-threaded binary trees?"

10.64 💻 Trees

20. Write C function to implement right in-threaded binary tree
21. Write C function to implement inorder successor and predecessor?
22. Write C function to traverse threaded binary tree in inorder
23. What is a binary search tree?
24. How to insert an element into a binary search tree?
25. Write a function to insert an item into a binary search tree (duplicate elements)
26. Write a function to insert an item into a binary search tree (No duplicate items)
27. Write a function to search for an item in binary search tree using iteration
28. Write a function to search for an item in BST using recursion
29. Write a function to return the address of highest item in BST
30. Write a function to return the address of least item in BST
31. Write a function to find the height of the tree
32. Write a function to count the number of nodes in a tree
33. Write a function to count the leaves or terminal nodes in a tree
34. Write a function to delete an item from the tree
35. Write a C function to copy a tree to other tree
36. Write a C function to check whether two trees are equal or not

Chapter 11: Graphs

What are we studying in this chapter?

- Definitions
- Terminologies
- Matrix and Adjacency List Representation of Graphs
- Elementary Graph operations
- Traversal methods:
  - Breadth First Search
  - Depth First Search

11.1 Introduction

In this chapter, let us concentrate another important and non-linear data structure called graph. In this chapter, we discuss basic terminologies and definitions, how to represent graphs and how graphs can be traversed.

11.2 Graph Theory terminology

First, let us see "What is a vertex?"

Definition: A vertex is a synonym for a node. A vertex is normally represented by a circle. For example, consider the following figure:

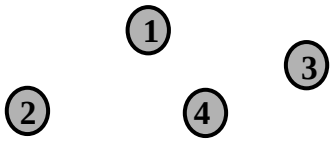


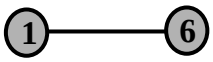
Fig Vertices

In the above figure, there are four nodes identified by 1, 2, 3, 4. They are also called vertices and normally denoted by a set V = {1, 2, 3, 4}.

Now, let us see "What is an edge?"

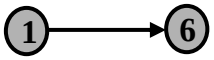
Definition: If *u* and *v* are vertices, then an *arc* or a *line* joining two vertices *u* and *v* is called an *edge*.

Example 1: Consider the figure:



Observe the following points from above figure:

- *There is no direction* for the edge between vertex 1 and vertex 6 and hence it is undirected edge.
- The undirected edge is denoted by an ordered pair (1, 6) where 1 and 6 are called *end points of the edge* (1, 6). In general, if e = (u, v), then the nodes u and v are called *end points of directed edge.*
- In this graph, edge (1, 6) is same as edge (6, 1) since there is no direction associated with that edge. So, (u, v) and (v, u) represent same edge.

Example 2: consider the figure: 

Observe the following points from above figure:

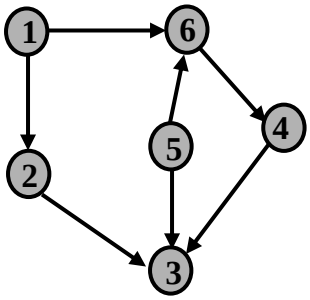
- *There is a direction* for the edge originating at vertex 1 (called tail of the edge) and heading towards vertex 6 (called head of the edge) and hence it is called directed edge.
- The directed edge is denoted by the directed pair <1, 6> where 1 is called *tail of the edge* and 6 is the *head of the edge*. So, the directed pair <1, 6> is not same as directed pair <6, 1>.
- In general, if a directed edge is represented by directed pair <u, v>, *u* is called the *tail of the edge* and *v* is the *head of the edge*. So, the directed pair <u, v> is different from the directed pair <v, u>. So, <u, v> and <v, u> represent two different edges.

Now, let us see "What is a graph?"

Definition: Formally, a graph **G** is defined as a pair of two sets V and E denoted by

G = (V, E)

where V is set of vertices and E is set of edges. For example, consider the graph shown below:



Here, graph G = (V, E) where

- V = {1, 2, 3, 4, 5, 6} is set of vertices
- E = { <1, 6>, <1, 2>, <2, 3>, <4, 3>, <5, 3>, <5, 6>, <6, 4> } is set of directed edges

Note:

- |V| = |{1, 2, 3, 4, 5, 6}| = 6 represent the number of vertices in the graph.

- |E| = |{<1, 6>, <1, 2>, <2, 3>, <4, 3>, <5, 3>, <5, 6>, <6, 4> }| = 7 represent the number of edges in the graph.

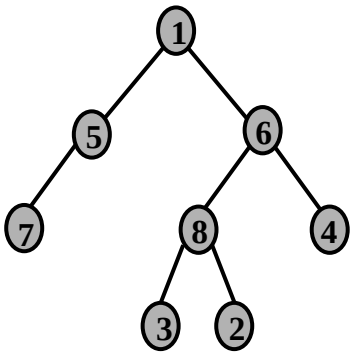
Now, let us see "What is a directed graph? What is an undirected graph?"

Definition: A graph G = (V, E) in which every edge is directed is called a directed graph. The directed graph is also called digraph. A graph G = (V, E) in which every edge is undirected is called an undirected graph. Consider the following graphs:

Here, graph G = (V, E) where

- V = {0, 1, 2} is set of vertices
- E = {<0, 1>, <1, 0>, <1, 2>} is set of edges

Note: Since all edges are directed it is a directed graph. In directed graph we use angular brackets < and > to represent an edge



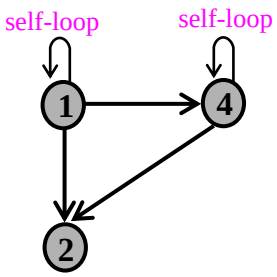
Here, graph G = (V, E) where

- V = {1, 2, 3, 4, 5, 6, 7, 8} is set of vertices
- E = {(1, 5), (1, 6), (5, 7), (6, 8), (6, 4), (8, 3), (8, 2)} is set of edges

Note: Since all edges are undirected, it is an undirected graph. In undirected graph we use parentheses (and) to represent an edge (u, v).

Now let us see "What is a self-loop (or self-edge)?

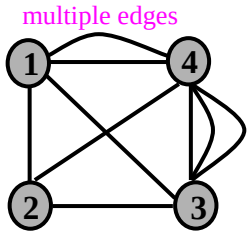
Definition: A loop is an edge which starts and ends on the same vertex. A loop is represented by an ordered pair (i, i). This indicates that the edge originates and ends in the same vertex. A loop is also called self-edge or self-loop. In the given graph shown below, there are two self-loops namely, <1, 1> and <4, 4>.



Now, let us see "What is a multigraph?"

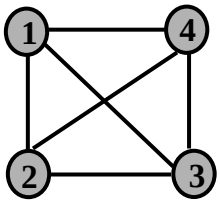
11.4 Graphs

Definition: A graph with multiple occurrence of the same edge between any two vertices is called multigraph. Here, there are two edges between the nodes 1 and 4 and there are three edges between the nodes 4 and 3.

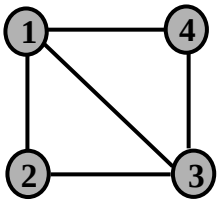


Now, let us see "What is a complete graph?"

Definition: A graph G = (V, E) is said to be a complete graph, if there exists an edge between every pair of vertices. The graph (a) below is complete. Observe that in a complete graph of *n* vertices, there will be n(n-1)/2 edges. Substituting *n* = 4, we get 6 edges. Even if one edge is removed as shown in graph (b) below, it is not complete graph.



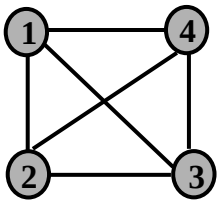
Complete graph



Not a complete graph

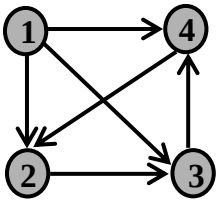
Now, let us see "What is a path?"

Definition: Let G = (V, E) be a graph. A *path* from vertex *u* to vertex *v* in an undirected graph is a sequence of adjacent vertices (u, v_0, v_1, v_2,….v_k, v) such that: (u, v_0), (v_0, v_1),….(v_k, v) are the edges in G. Consider the following graph:



In the graph, the path from vertex 1 to 4 is denoted by: 1, 2, 3, 4 which can also be written as (1, 2), (2, 3), (3, 4).

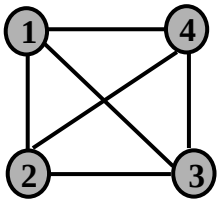
Definition: Let G = (V, E) be a graph. A *path* from vertex *u* to vertex *v* in a directed graph is a sequence of adjacent vertices <u, v_0, v_1, v_2,….v_k, v> such that <u, v_0>, <v_0, v_1>,….<v_k, v> are the edges in G. Consider the following graph:



In the graph, the path from vertex 1 to 3 is denoted by 1, 4, 2, 3 which can also be written as <1, 4>, <4, 2>, <2, 3>

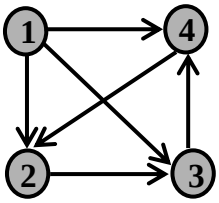
Now, let us see "What is simple path?"

Definition: A *simple path* is a path in which all vertices except possibly the first and last are distinct. Consider the undirected and directed graph shown below:



Ex 1: In the graph, the path 1, 2, 3, 4 is simple path since each node in the sequence is distinct.

Ex2: In the graph, the path 1, 2, 3, 2 is not a simple path since the nodes in sequence are not distinct. The node 2 appears twice in the path



Ex 1: In the graph, the path 1, 4, 2, 3 is simple path since each node in the sequence is distinct.

Ex 2: The sequence 1, 4, 3 is not a path since there is no edge <4, 3> in the graph.

Now, let us see "What is length of the path?"

Definition: The *length* of the path is the number of edges in the path.

Ex 1: In the above undirected graph, the path (1, 2, 3, 4) has length 3 since there are three edges (1, 2), (2, 3), (3, 4). The path 1, 2, 3 has length 2 since there are two edges (1, 2), (2, 3).

Ex 2: In the above directed graph, the path <1, 2, 3, 4> has length 3 since there are three edges <1, 2>, <2, 3>, <3, 4>. The path <1, 4, 2> has length 2 since there are two edges <1, 4>, <4, 2>.

Now, let us "Define the terms cycle (circuit)?"

Definition: A cycle is a path in which the first and last vertices are same.

For example, the path <4, 2, 3, 4> shown in above directed graph is a cycle, since the first node and last node are same. It can also be represented as <4, 2>, <2, 3>, <3, 4> <4, 2>.

Note: A graph with at least one cycle is called a cyclic graph and a graph with no cycles is called ***acyclic*** graph. A tree is an acyclic graph and hence it has no cycle.

Now, let us see "What is a connected graph?"

11.6 Graphs

Definition: In an undirected graph G, two vertices *u* and *v* are said to be connected if there exists a path from *u* to *v*. Since G is undirected, there exists a path from *v* to *u* also. A graph G (directed or undirected) is said to be connected if and only if there exists a path between every pair of vertices.

For example, the graphs shown in figure below are connected graphs.

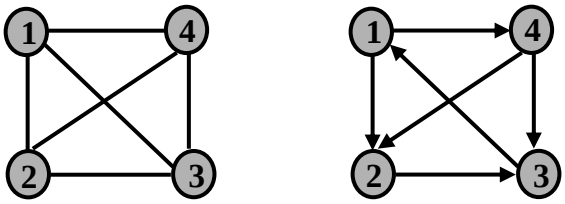
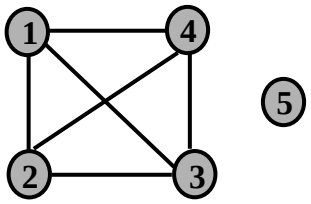


Figure Connected graphs

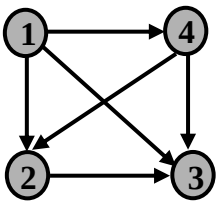
Now, let us see "What is a disconnected graph?"

Definition: Let G = (V, E) be a graph. If there exists at least one vertex in a graph that cannot be reached from other vertices in the graph, then such a graph is called disconnected graph. For example, the graph shown below is a disconnected graph.



Not connected

Since vertex 1 is not reachable from 3, the graph is not connected



11.3 Representation of graph

Now, let us see "What are the different methods of representing a graph?" The graphs can be represented in two different methods:

Representation of graph → Adjacency matrix
Representation of graph → Adjacency linked list

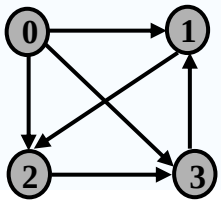
Let us see "What is an adjacency matrix? explain with example"

Definition: Let G = (V, E) be a graph where V is set of vertices and E is set of edges. Let N be the number of vertices in graph G. The adjacency matrix A of a graph G is formally defined as shown below:

$$A[i][j] = \begin{cases} 1 & \text{if there is an edge from vertex } i \text{ to vertex } j. \\ 0 & \text{if there is no edge from vertex } i \text{ to vertex } j. \end{cases}$$

- It is clear from the definition that an adjacency matrix of a graph with *n* vertices is a Boolean square matrix with *n* rows and *n* columns with entries 1's and 0's (bit-matrix)
- In an undirected graph, if there exists an edge (i, j) then a[i][j] and a[j][i] is made 1 since (i, j) is same as (j, i)
- In a directed graph, if there exists an edge <i, j> then a[i][j] is made 1 and a[j][i] will be 0.
- If there is no edge from vertex *i* to vertex *j*, then a[i][j] will be 0.

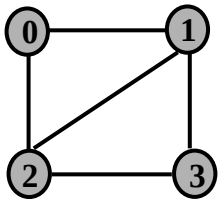
Note: The above definition is true both for directed and undirected graph. For example, following figures shows the directed and undirected graphs along with equivalent adjacency matrices:



(a) Directed graph

	0	1	2	3
0	0	1	1	1
1	0	0	1	0
2	0	0	0	1
3	0	1	0	0

Adjacency matrix



(b) Undirected graph

	0	1	2	3
0	0	1	1	0
1	1	0	1	1
2	1	1	0	1
3	0	1	1	0

Adjacency matrix

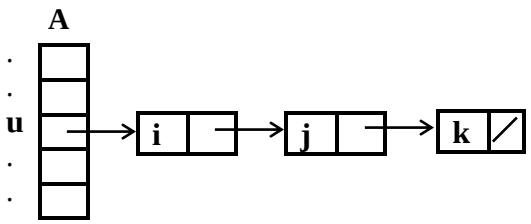
Fig. Graphs and equivalent adjacency matrices

Now, let us see "What is an adjacency list? explain with example"

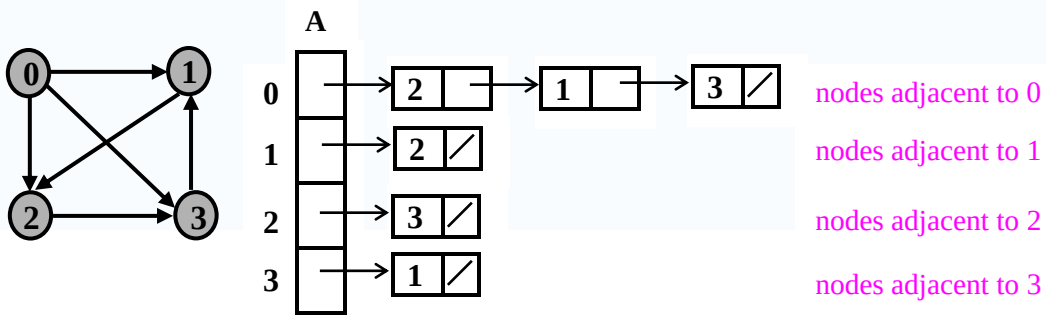
11.8 Graphs

Definition: Let G = (V, E) be a graph. An *adjacency linked list* is an array of *n* linked lists where *n* is the number of vertices in graph G. Each location of the array represents a vertex of the graph. For each vertex $u \in V$, a linked list consisting of all the vertices adjacent to *u* is created and stored in A[u]. The resulting array A is an adjacency list.

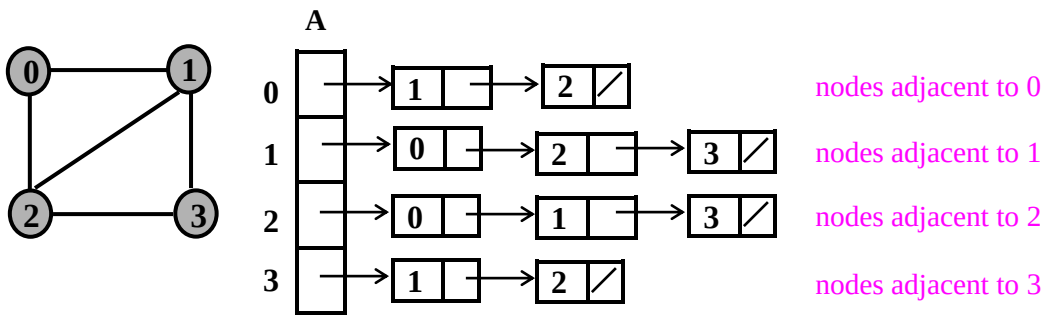
Note: It is clear from the above definition that if *i*, *j* and *k* are the vertices adjacent to the vertex *u*, then *i*, *j* and *k* are stored in a linked list and starting address of linked list is stored in A[u] as shown below:



For example, figures below shows the directed and undirected graphs along with equivalent adjacency linked list:



(a) Directed graph **Adjacency linked list**



(b) Undirected graph **Adjacency linked list**

Fig.: Graphs and equivalent adjacency linked lists

Now, let us see "Which graph representation is best?" The graph representation to be used depends on the following factors:

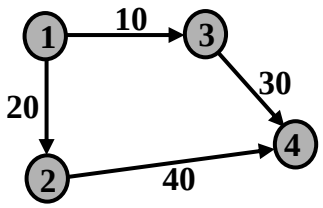
- Nature of the problem
- Algorithm used for solving
- Type of the input.
- Number of vertices and edges:
  - If a graph is *sparse*, less number of edges are present. In such case, the adjacency list has to be used because this representation uses lesser space when compared to adjacency matrix representation, even though extra memory is consumed by the pointers of the linked list.
  - If a graph is *dense*, the adjacency matrix has to be used when compared with adjacency list since the linked list representation takes more memory.

Note: So, based on the nature of the problem and based on whether the graph is *sparse* or *dense*, one of the two representations can be used.

Now, let us see "What is a weighted graph?"

Definition: A graph in which a number is assigned to each edge in a graph is called weighted graph. These numbers are called costs or weights. The weights may represent the cost involved or length or capacity depending on the problem.

For example, in the following graph shown in figure the values 10, 20, 30 and 40 are the weights associated with four edges <1,3>, <1,2>, <3,4> and <2,4>



Let us see "How the weighted graph can be represented?" The weighted graph can be represented using adjacency matrix as well as adjacency linked list. The adjacency matrix consisting of costs (weights) is called cost adjacency matrix. The adjacency linked list consisting of costs (weights) is called cost adjacency linked list. Now, let us see "What is cost adjacency matrix?"

Definition: Let G = (V, E) be the graph where V is set of vertices and E is set of edges with *n* number of vertices. The cost adjacency matrix A of a graph G is formally defined as shown below:

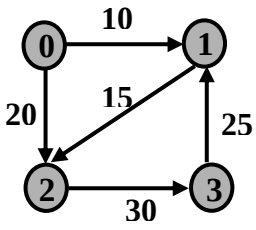
$$A[i][j] = \begin{cases} w & \text{if there is a weight associated with edge from vertex } i \text{ to vertex } j. \\ \infty & \text{if there is no edge from vertex } i \text{ to vertex } j. \end{cases}$$

11.10 Graphs

It is clear from the above definition that

- The element in i^{th} row and j^{th} column is weight *w* provided there exist an edge from i^{th} vertex to j^{th} vertex with cost *w*
- ∞ if there is no edge from vertex *i* to vertex *j*.
- The cost from vertex *i* to vertex *i* is ∞ (assuming there is no loop).

For example, the weighted graph and its *cost adjacency matrix* is shown below:



(a) Weighted graph

	0	1	2	3
0	∞	10	20	∞
1	∞	∞	15	∞
2	∞	∞	∞	30
3	∞	25	∞	∞

Note: Diagonal values can be replaced by 0's

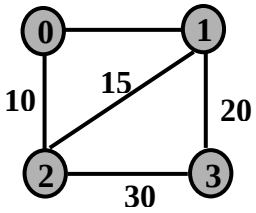
(b) Adjacency matrix

Figure A weighted digraph and the cost adjacency matrix

For the undirected graph, the elements of the cost adjacency matrix are obtained using the following definition:

$$A[i][j] = \begin{cases} w & \text{if there is a weight associated with edge (i, j) or (j, i)} \\ \infty & \text{if there is no edge from vertex } i \text{ to vertex } j. \end{cases}$$

The undirected graph and its equivalent adjacency matrix is shown below:



(a)

	0	1	2	3
0	∞	25	10	∞
1	25	∞	15	20
2	10	15	∞	30
3	∞	20	30	∞

Diagonal values can be replaced by 0's

(b)

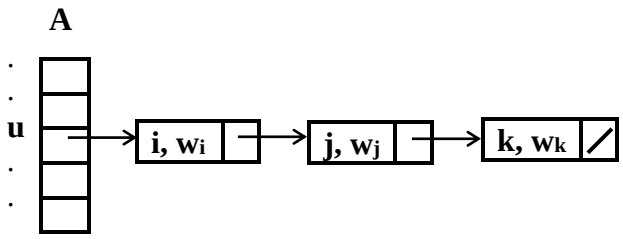
Figure: Weighted undirected graph and the adjacency matrix

Note: The cost adjacency matrix for the undirected graph is symmetric (i.e., a[i, j] is same as a[j, i]) whereas the cost adjacency matrix for a directed graph may not be symmetric.

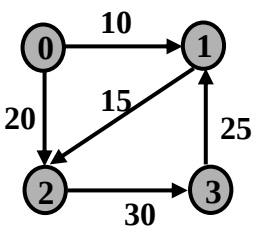
Note: For some of the problems, it is more convenient to store 0's in the main diagonal of cost adjacency matrix instead of ∞.

Now, let us see "What is cost adjacency linked list?"

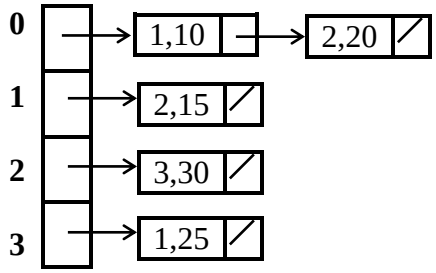
Definition: Let G = (V, E) be a graph where V is set of vertices and E is set of edges with *n* number of vertices. A *cost adjacency linked list* is an array of *n* linked lists. For each vertex u ∈ V, A[u] contains the address of a linked list. All the vertices which are adjacent from vertex *u* are stored in the form of a linked list (in an arbitrary manner) and the starting address of first node is stored in A[u]. If *i, j* and *k* are the vertices adjacent to the vertex u, then *i, j* and *k* are stored in a linked list along with the weights in A[u] as shown below:



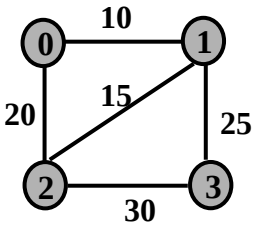
For example, the figure below shows the weighted diagraph and undirected graph along with equivalent adjacency list.



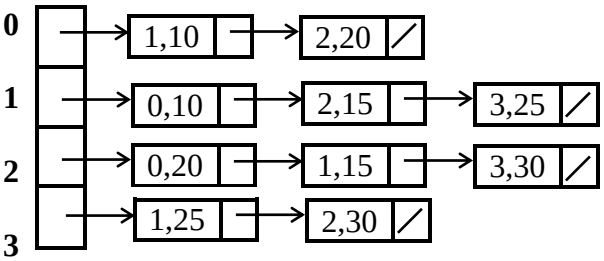
(a) weighted digraph



(b) adjacency list



(c) weighted undirected graph



(d) adjacency list

Figure weighted graph and equivalent adjacency list

Now, the function to read an adjacency matrix can be written as shown below:

Example 11.1: Function to read adjacency matrix

```
void read_adjacency_matrix(int a[10][10], int n)
{
        int     i, j;

        for (i = 0; i < n; i++)
        {
                for (j = 0; j < n; j++)
                {
                        scanf("%d", &a[i][j]);
                }
        }
}
```

The function to read adjacency list can be written as shown below:

Example 11.2: Function to read adjacency list

```
void read_adjacency_list (NODE a[], int n)
{
        int     i, j, m, item;

        for (i = 0; i < n; i++)
        {
                printf("Enter the number of nodes adjacent to %d:", i);
                scanf("%d", &m);

                if (m ==0) continue;

                printf("Enter nodes adjacent to %d : ", i);

                for (j = 0; j < m; j++)
                {
                        scanf("%d", &item);

                        a[i] = insert_rear(item, a[i]);
                }
        }
}
```

11.4 Graph traversals

Now, we concentrate on a very important topic namely graph traversal techniques and see "What is graph traversal? Explain different graph traversal techniques"

Definition: The process of visiting each node of a graph systematically in some order is called graph traversal. The two important graph traversal techniques are:

- **B**readth **F**irst **S**earch (**BFS**)
- **D**epth **F**irst **S**earch (**DFS**)

11.4.1 Breadth First Search (BFS)

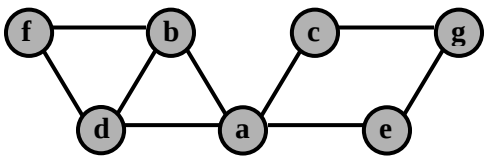
Now, let us see "What is breadth first search (BFS)?"

Definition: The breadth first search is a method of traversing the graph from an arbitrary vertex say *u*. First, visit the node *u*. Then we visit all neighbors of *u*. Then we visit the neighbors of neighbors of *u* and so on. That is, we visit all the neighboring nodes first before moving to next level neighbors. The search will terminate when all the vertices have been visited.

BFS traversal can be implemented using a queue. As we visit a node, it is inserted into queue. Now, delete a node from a queue and see the adjacent nodes which have not been visited. The unvisited nodes are inserted into queue and marked as visited. Deleting and inserting operations as discussed are continued until queue is empty.

Now, let us take an example and see how BFS traversal can be used to see what are all the nodes which are reachable from a given source vertex.

Example 11.3: Traverse the following graph by breadth-first search and print all the vertices reachable from start vertex *a*. Resolve ties by the vertex alphabetical order.



Solution: It is given that source vertex is *a*. Perform the following activities:

Initialization: Insert source *vertex a into queue* and *add a to S* as shown below:

	(i) u = del(Q)	(ii) v = adj. to u	(iii) Nodes visited S	(iii) queue
Initialization	-	-	a	a

Step 1: i): Delete an element *a* from queue
ii): Find the nodes adjacent to *a* but not in S: i.e., *b, c, d* and *e*
iii): Add *b, c, d* and *e* to S, insert into queue as shown in the table:

	(i) u = del(Q)	(ii) v = adj. to u	(iii) Nodes visited S	(iii) queue
	-	-	a	a
Step 1	a	b, c, d, e	a, b, c, d, e	b, c, d, e

Step 2: i): Delete *b* from queue
ii): Find nodes adjacent to *b* but not in S: i.e., f
iii): Add *f* to S and insert *f* into queue as shown in table:

	(i) u = del(Q)	(ii) v = adj. to u	(iii) Nodes visited S	(iii) queue
	-	-	a	a
Step 1	a	b, c, d, e	a, b, c, d, e	b, c, d, e
Step 2	b	f	a, b, c, d, e, f	c, d, e, f

Stage 3: i) : Delete *c* from queue
ii): Find nodes adjacent to *c* but not in S: i.e., g
iii): Add *g* to S, insert *g* into queue as shown in table

	(i) u = del(Q)	(ii) v = adj. to u	(iii) Nodes visited S	(iii) queue
	-	-	a	a
Step 1	a	b, c, d, e	a, b, c, d, e	b, c, d, e
Step 2	b	f	a, b, c, d, e, f	c, d, e, f
Step 3	c	g	a, b, c, d, e, f, g	d, e, f

The remaining steps are shown in the following table:

(i) (ii) ←——————(iii)——————→

	u = del(Q)	v = adj. to u	Nodes visited S	queue
Initialization	-	-	a	a
Step 1	a	b, c, d, e	a, b, c, d, e	b, c, d, e
Step 2	**b**	a, d, **f**	a, b, c, d, e, f	c, d, e, f
Step 3	c	a, **g**	a, b, c, d, e, f, g	d, e, f, g
Step 4	d	a, b, f	a, b, c, d, e, f, g	e, f, g
Step 5	e	a,g	a, b, c, d, e, f, g	f, g
Step 6	f	b, d	a, b, c, d, e, f, g	g
Step 7	g	c. e	a, b, c, d, e, f, g	empty

Thus, the nodes that are reachable from source *a*: **a, b, c, d, e, f, g**

11.4.1.1 Breadth First Search (BFS) using adjacency matrix

The above activities are shown below in the form of an algorithm along with pseudocode in C when graph is represented as an adjacency matrix.

```
no node is visited to start with                // int s [10] = {0};

insert source u to q                            // f = 0, r = -1, q[++r] = u
print u                                         // print u
mark u as visited i.e., add u to S              // s[u] = 1
while queue is not empty                        // while f <= r

    Delete a vertex u from q                    //      u = q[f++]
    For every v adjacent to u                   //      for each v, if a[u][v] == 1
    If v is not visited                         //          if s[v] == 0
        print v                                 //              print v
        mark v as visited                       //              s[v] = 1
        Insert v to queue                       //              q[++r] = v
    end if                                      //          endif
                                                //      endif
end while                                       // end while
```

The above algorithm can be written using C function as shown below:

Example 11.4: C function to show the nodes visited using BFS traversal

```
void bfs(int a[10][10], int n, int u)
{
        int     f, r, q[10], v;
        int     s[10] = {0};   /* initialize all elements in s to 0 i.e, no node is visited */

        printf("The nodes visited from %d : ", u);
        f = 0, r = -1;              // queue is empty
        q[++r] = u;                 // Insert u into queue

        s[u] = 1;                   // insert u to s
        printf("%d  ", u);          // print the node visited

        while ( f <= r )
        {
                u = q[f++];                             // delete an element from q
                for (v = 0; v < n; v++)
                {
                        if (a[u][v] == 1)               // If v is adjacent to u
                        {
                                if (s[v] == 0)  // If v is not in S i.e., v has not been visited
                                {
                                        printf("%d  ", v);   // print the node visited
                                        s[v] = 1;            // add v to s, mark it as visited
                                        q[++r] = v;          // Insert v into queue
                                }
                        }
                }
        }
        printf("\n");
}
```

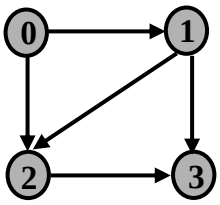
Now, the C program that prints all the nodes that are reachable from a given source vertex is shown below:

Example 11.5: Algorithm to traverse the graph using BFS

```
#include <stdio.h>
/* Insert: Example  11.1: Function to read an adjacency  matrix*/
/* Insert: Example  11.4: Function to traverse the graph in BFS */
```

```
void main()
{
        int     n, a[10][10], source, i, j;

        printf("Enter the number of nodes : ");
        scanf("%d", &n);

        printf("Enter the adjacency matrix:\n");
        for (i = 0; i < n; i++)
        {
                for (j = 0; j < n; j++) scanf("%d", &a[i][j]);
        }

        for (source = 0; source < n; source++)
                bfs(a, n, source);
}
```

Now, let us see how to obtain the nodes reachable from each node of the following graph using the above program:



Given graph

	0	1	2	3
0	0	1	1	0
1	0	0	1	1
2	0	0	0	1
3	0	0	0	0

Adjacency matrix

Output

```
Enter the number of nodes: 4
Enter the adjacency matrix:
0 1 1 0
0 0 1 1
0 0 0 1
0 0 0 0
The nodes visited from 0: 0  1  2  3
The nodes visited from 1: 1  2  3
The nodes visited from 2: 2  3
The nodes visited from 3: 3
```

11.4.1.2 Breadth First Search (BFS) using adjacency list

We know that BFS traversal uses queue data structure which require insert rear function and delete front function. We can use insert rear function given in example 8.6. But, the delete front function shown in example 8.5 is modified after deleting the printf() function.

Example 11.6: C function to delete an item from the front end of singly linked list

```
NODE delete_front(NODE first)
{
        NODE temp;

        if ( first == NULL )   return NULL;

        temp = first;                   /* Retain address of the node to be deleted */
        temp = temp->link;              /* Obtain address of the second node */
        free(first);                    /* delete the front node */
        return temp;                    /* return address of the first node */
}
```

The algorithm for BFS along with pseudocode when a graph is represented as an adjacency list can be written as shown below:

```
no node is visited to start with        // int s [10] = {0}
insert source u to q                    // q = NULL, q = insert_rear(u, q);
mark u as visited i.e., add u to S      // s[u] = 1, printf("%d  ", u);
while queue is not empty                // while q != NULL
                                                u = q->info;
        Delete a vertex u from q        //      q = delete_front(q),
                                                list = a[u]; // list of vertices adj. to u
        Find vertices v adjacent to u   //      while (list != NULL)
                                                        v = list->info;
        If v is not visited             //              if (s[v] == 0)
                print v                 //                      print v
                mark v as visited       //                      s[v] = 1
                Insert v to queue       //                      q = insert_rear(v, q);
        end if                          //              endif
                                                        list = list->link
                                        //      end while
end while                               // end while
```

Now, the complete C function to traverse the graph using BFS when a graph is represented as adjacency list can be written as shown below:

Example 11.7: C function to show the nodes visited using BFS traversal

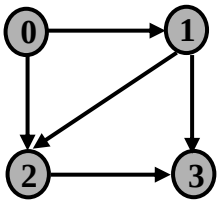
```
void bfs(NODE a[], int n, int u)
{
        NODE        q, list;
        int         v;

        int     s[10] = {0};   /* initialize all elements in s to 0 i.e, no node is visited */

        printf("The nodes visited from %d : ", u);

        q = NULL;                       // queue is empty
        q = insert_rear(u, q);          // Insert u into queue

        s[u] = 1;                       // insert u to s
        printf("%d  ", u);              // print the node visited
        while ( q != NULL )                             // as long as queue is not empty
        {
                u = q->info;                            // delete a node from queue
                q = delete_front(q);

                list = a[u];                            // obtain nodes adjacent to u
                while (list != NULL)                    // as long as adjacent nodes exist
                {
                        v = list->info;                 // v is the node adjacent to u
                        if (s[v] == 0)  // If v is not in S i.e., v has not been visited
                        {
                                printf("%d  ", v);      // print the node visited

                                s[v] = 1;                // add v to s, mark it as visited
                                q = insert_rear(v, q);  // Insert v into queue
                        }
                        list = list->link;
                }
        }
        printf("\n");
}
```

Now, the complete C program to see the nodes reachable from each of the nodes in the graph can be written as shown below:

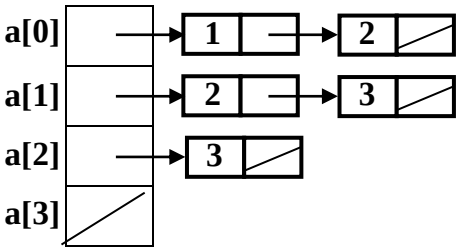
Example 11.8: Program to print nodes reachable from a vertex (bfs using adjacency list)

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
        int             info;
        struct node     *link;
};

typedef struct node *NODE;

/* Insert: Example  8.2: Function to get a node */
/* Insert: Example  8.6: Function to insert an element into queue */
/* Insert: Example  11.2: Function to read adjacency list */
/* Insert: Example  11.6: Function to delete an element from front end of queue */
/* Insert: Example  11.7: Function to traverse the graph in BFS (adjacency list) */
void main()
{
        int             n, i, source;
        NODE            a[10];

        printf("Enter the number of nodes : ");
        scanf("%d", &n);

        for (i = 0; i < n; i++) a[i] = NULL;            // Graph is empty to start with

        read_adjacency_list(a, n);

        for (source = 0; source < n; source++)
                bfs(a, n, source);
}
```

Now, let us see how to obtain the nodes reachable from each node of the following graph using the above program:



Given graph



Adjacency linked list

Input

```
Enter the number of nodes: 4
Enter the number of nodes adjacent 0: 2
Enter nodes adjacent to 0: 1  2

Enter the number of nodes adjacent 1: 2
Enter nodes adjacent to 1: 2  3

Enter the number of nodes adjacent 2: 1
Enter nodes adjacent to 2: 3

Enter the number of nodes adjacent 3: 0
Enter nodes adjacent to 3:
```

Output

```
The nodes visited from 0: 0  1  2  3
The nodes visited from 1: 1  2  3
The nodes visited from 2: 2  3
The nodes visited from 3: 3
```

11.4.2 Depth First Search (DFS)

The depth first search is a method of traversing the graph by visiting each node of the graph in a systematic order. As the name implies *depth-first-search* means "to search deeper in the graph". Now, let us see "What is depth first search (DFS)?"

Definition: In DFS, a vertex *u* is picked as source vertex and is visited. The vertex *u* at this point is said to be unexplored. The exploration of the vertex *u* is postponed and a vertex *v* adjacent to *u* is picked and is visited. Now, the search begins at the vertex

v. There may be still some nodes which are adjacent to *u* but not visited. When the vertex *v* is completely examined, then only *u* is examined. The search will terminate when all the vertices have been examined.

Note: The search continues deeper and deeper in the graph until no vertex is adjacent or all the vertices are visited. Hence, the name DFS. Here, the exploration of a node is postponed as soon as a new unexplored node is reached and the examination of the new node begins immediately.

Design methodology The iterative procedure to traverse the graph in DFS is shown below:

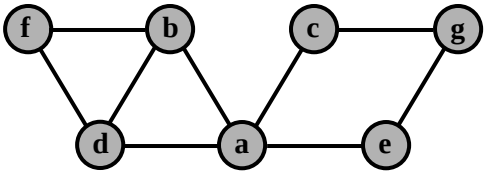
Step 1: Select node *u* as the start vertex (select in alphabetical order), push *u* onto stack and mark it as visited. We add *u* to S for marking

Step 2: While stack is not empty

```
        For vertex u on top of the stack, find the next immediate adjacent vertex.
        if v is adjacent
            if a vertex v not visited then
                push it on to stack and number it in the order it is pushed.
                mark it as visited by adding v to S
            else
                ignore the vertex
            end if
        else
            remove the vertex from the stack
            number it in the order it is popped.
        end if
    end while
```

Step 3: Repeat step 1 and step 2 until all the vertices in the graph are considered

Example 11.9: Traverse the following graph using DFS and display the nodes reachable from a given source vertex



Solution: Since vertex *a* is the least in alphabetical order, it is selected as the start vertex. Follow the same procedure as we did in BFS. But, there are two changes:

- Instead of using a queue, we use stack
- In BFS, all the nodes adjacent and which are not visited are considered. In DFS, only one adjacent which is not visited earlier is considered. Rest of the procedure remains same.

Now, the graph can be traversed using DFS as shown in following table

	Stack	**v = adj(s[top])**	**Nodes visited S**	**pop(stack)**
Initial step	a	-	a	
Stage 1	a	b	a, b	-
Stage 2	a, b	d	a, b, d	-
Stage 3	a, b, d	f	a, b, d, f	-
Stage 4	**a, b, d, f**	-	a, b, d, f	f
Stage 5	a, b, d	-	a, b, d, f	d
Stage 6	a, b	-	a, b, d, f	b
Stage 7	a	c	a, b, d, f	-
Stage 8	a, c	g	a, b, d, f, g	-
Stage 9	a, c, g	e	a, b, d, f, g, e	-
Stage 10	a, **c, g, e**	-	a, b, d, f, g, e	e
Stage 11	a, c, g	-	a, b, d, f, g, e	g
Stage 12	a, c	-	a, b, d, f, g, e	c
Stage 13	a_1	-	a, b, d, f, g, e	$a_{1,7}$

11.4.2.1 Depth First Search (DFS) using adjacency matrix

It is clear from the above example that the *stack* is the most suitable data structure to implement DFS. Whenever a vertex is visited for the first time, that vertex is pushed on to the stack and the vertex is deleted from the stack when a dead end is reached and the search resumes from the vertex that is deleted most recently. If there are no vertices adjacent to the most recently deleted vertex, the next node is deleted from the stack and the process is repeated till all the vertices are reached or till the stack is empty.

The recursive function can be written as shown below: (Assuming adjacency matrix *a*, number of vertices *n* and array *s* as global variables)

Example 11.10: Program to print nodes reachable from a vertex (dfs - adjacency matrix)

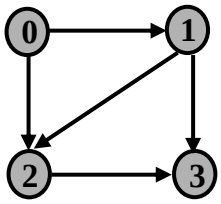
```
void dfs(int u)
{
        int v;

        s[u] = 1;

        printf("%d ", u);

        for (v = 0; v < n; v++)
        {
                if (a[u][v] == 1 && s[v] == 0) dfs(v);
        }
}
```

The complete program that prints the nodes reachable from each of the vertex given in the graph can be written as shown below:

Example 11.11: Program to print nodes reachable from a vertex (dfs - adjacency matrix)

```
#include <stdio.h>

int     a[10][10], s[10], n;            // Global variables

/* Insert: Example 11.1: Function to read an adjacency  matrix*/
/* Insert: Example 11.10: Function to traverse the graph in DFS */

void main()
{
        int     i, source;

        printf("Enter the number of nodes in the graph : ");
        scanf("%d", &n);
        printf("Enter the adjacency matrix:\n");
        read_adjacency_matrix(a, n);

        for (source = 0; source < n; source++)
        {
                for (i = 0; i < n; i++) s[i] = 0;

                printf("\nThe nodes reachable from %d: ", source);
                dfs(source);
        }
}
```

Now, let us see how to obtain the nodes reachable from each node of the following graph using the above program:



Given graph

	0	1	2	3
0	0	1	1	0
1	0	0	1	1
2	0	0	0	1
3	0	0	0	0

Adjacency matrix

Output

```
Enter the number of nodes: 4
Enter the adjacency matrix:
0 1 1 0
0 0 1 1
0 0 0 1
0 0 0 0
The nodes visited from 0: 0  1  2  3
The nodes visited from 1: 1  2  3
The nodes visited from 2: 2  3
The nodes visited from 3: 3
```

11.4.2.2 Depth First Search (DFS) using adjacency linked list

The procedure remains same. But, instead of using adjacency matrix, we use adjacency list. The recursive function can be written as shown below: (Assuming adjacency list *a*, number of vertices *n* and array *s* as global variables.)

Example 11.12: Program to print nodes reachable from a vertex (dfs - adjacency list)

```
void dfs(int u)
{
        int v;
        NODE temp;

        s[u] = 1;

        printf("%d ", u);

        for (temp = a[u];   temp != NULL;   temp = temp->link)
                if (s[temp->info] == 0) dfs(temp->info);
}
```

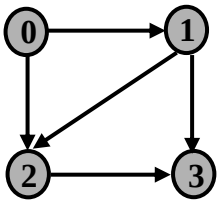
11.26 💻 Graphs

The complete program that prints the nodes reachable from each of the vertex given in the graph using DFS represented using adjacency list can be written as shown below:

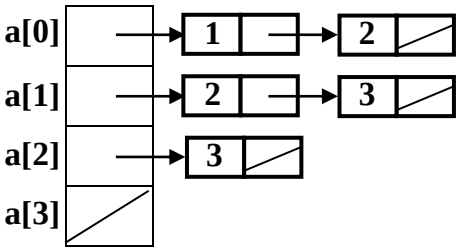
Example 11.13: Program to print nodes reachable from a vertex (dfs - adjacency matrix)

```
#include <stdio.h>
#include <stdlib.h>

struct node
{
        int             info;
        struct node    *link;
};

typedef struct node *NODE;

NODE        a[10];
int         s[10], n;               // Global variables

/* Insert: Example 8.2: Function to get a node */

/* Insert: Example 8.6: Function to insert an element into queue */

/* Insert: Example 11.2: Function to read adjacency list */

/* Insert: Example 11.12: Function to traverse the graph in DFS */
void main()
{
        int     i, source;

        printf("Enter the number of nodes in the graph : ");
        scanf("%d", &n);

        printf("Enter the adjacency list:\n");
        read_adjacency_list(a, n);

        for (source = 0; source < n; source++)
        {
                for (i = 0; i < n; i++) s[i] = 0;

                printf("\nThe nodes reachable from %d: ", source);
                dfs(source);
        }
}
```

Now, let us see how to obtain the nodes reachable from each node of the following graph using the above program:



Given graph



Adjacency linked list

Input

```
Enter the number of nodes: 4
Enter the number of nodes adjacent 0: 2
Enter nodes adjacent to 0: 1  2

Enter the number of nodes adjacent 1: 2
Enter nodes adjacent to 1: 2  3

Enter the number of nodes adjacent 2: 1
Enter nodes adjacent to 2: 3

Enter the number of nodes adjacent 3: 0
Enter nodes adjacent to 3:
```

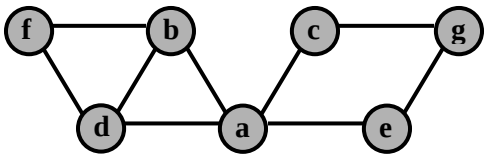
Output

```
The nodes visited from 0: 0  1  2  3
The nodes visited from 1: 1  2  3
The nodes visited from 2: 2  3
The nodes visited from 3: 3
```

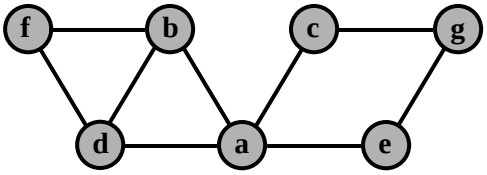
Exercises

1) Define the terms: a) vertex b) edge c) graph d) directed graph
   e) undirected graph
2) Define the terms: a) self-loop (or self-edge) b) multigraph c) complete graph
3) Define the terms: a) path b) simple path c) length of the path
4) Define the terms: a) cycle (circuit) b) Connected graph c) disconnected graph
5) What are the different methods of representing a graph?
6) What is an adjacency matrix? explain with example

7) What is an adjacency list? Explain with example
8) What is a weighted graph?
9) How the weighted graph can be represented?
10) What is cost adjacency matrix? What is cost adjacency linked list?
11) What is graph traversal? Explain different graph traversal techniques
12) What is breadth first search (BFS)?"
13) Traverse the following graph by breadth-first search and print all the vertices reachable from start vertex *a*. Resolve ties by the vertex alphabetical order.



14) Write a C function to show the nodes visited using BFS traversal (adjacency matrix)
15) Write a C function to show the nodes visited using BFS traversal (adjacency list)
16) What is depth first search (DFS)?"
17) Traverse the following graph using DFS and display the nodes reachable from a given source vertex



18) Write a program to print nodes reachable from a vertex (dfs - adjacency matrix)
19) Write a program to print nodes reachable from a vertex (dfs - adjacency matrix)

Chapter 12: Sorting and Searching

What are we studying in this chapter?

- **Sorting**
  - Insertion Sort
  - Radix sort
  - Address Calculation Sort
- **Hashing:**
  - Hash Table organizations
  - Hashing Functions
  - Static hashing
  - Dynamic Hashing

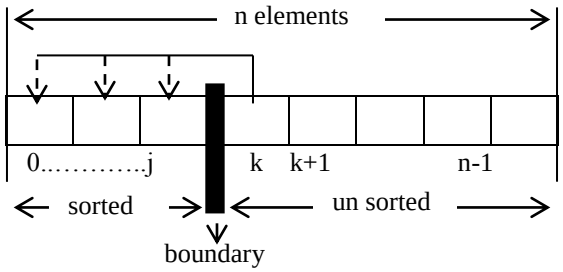
12.1 Sorting

We have already seen how to sort elements using bubble sort. In this chapter, we study three sorting techniques: Insertion sort, Radix sort and address calculation sort.

12.2 Insertion Sort

In this section, let us see "How insertion sort works?"

Procedure: The sorting procedure is similar to the way we play cards. After shuffling the cards, we pick each card and insert it into the proper place so that cards in hand are arranged in ascending order. The same technique is being followed while arranging the elements in ascending order. The given list is divided into two parts: sorted part and unsorted part as shown below:

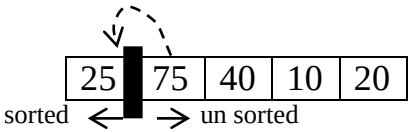


Note that all the elements from 0 to j are sorted and elements from k to n-1 are not sorted. The k^{th} item can be inserted into any of the positions from 0 to j so that elements towards left of boundary are sorted. As each item is inserted towards the sorted left part, the boundary moves to the right decreasing the unsorted list. Finally,

once the boundary moves to the right most position, the elements towards the left of boundary represent the sorted list.

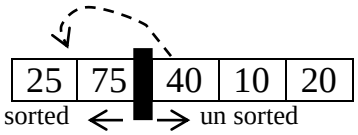
Example 12.1: Sort the elements 25 75 40 10 20 using insertion sort

Step 1: Item to be inserted is 75. i.e., item = a[1]



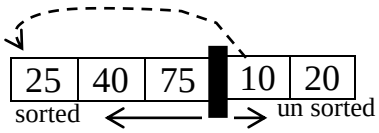
75 is inserted after 25

Step 2: Item to be inserted is 40 i.e., item = a[2]



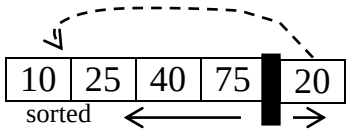
40 is inserted between 25 and 75

Step 3: Item to be inserted is 10 i.e., item = a[3]



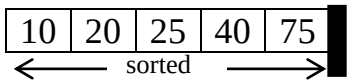
10 is inserted before 25

Step 4: Item to be inserted is 20 i.e., item = a[4]



20 is inserted between 10 and 25

Output



Final sorted list

Design: Consider an array of ***n*** elements to sort. The item to be inserted can be accessed as shown below:

in step 1: item = a[**1**]
in step 2: item = a[**2**]
in step 3: item = a[**3**]
in step 4: item = a[**4**]

i.e., item = a[i] where i = 1 to 4
i.e., i = 1 to 5-1
i.e., i = 1 to n -1

So, in general item = a[i] for i = 1 to n-1

Now, the **item** has to compared with **a[j]** as long as **item < a[j]** and j >= 0 with initial value of j = i-1. As long as the above condition is true perform the following activities:

- copy a[j] to a[j+1]
- decrement j by 1.

The equivalent statements can be written as shown below:

```
item = a[i];
j = i – 1;

while (item < a[j] && j >= 0)
{
        a[j+1] = a[j];
        j = j – 1;
}
```

Once control comes out of the above loop, insert the *item* into a[j+1]. Now, the partial code can be written as shown below:

```
item = a[i];
j = i – 1;
while (item < a[j] && j >= 0)
{
        a[j+1] = a[j];
        j = j – 1;
}
a[j+1] = item;
```

These statement should be executed for each item = a[i], where i = 1 to n-1

Now, we can write the above program segment as shown below:

```
for (i = 0; i < n; i++)
{
        item = a[i];
        j = i – 1;
        while (item < a[j] && j >= 0)
        {
                a[j+1] = a[j];
                j = j – 1;
        }
        a[j+1] = item;
}
```

12.4 Sorting and searching

The C function to sort *n* items using insertion sort is shown below:

Example 12.2: C function to sort items using insertion sort

```
void insertion_sort( int a[], int n)
{
        int     i, j, item;

        for (i = 1; i < n; i++)
        {
                item = a[i];                    /* Insert the item from unsorted part */
                j = i - 1;

                while (item < a[j] && j >= 0) /* Find the appropriate place to insert */
                {
                        a[j+1] = a[j];
                        j = j - 1;
                }

                a[j+1] = item;                  /* Insert at the appropriate place */
        }
}
```

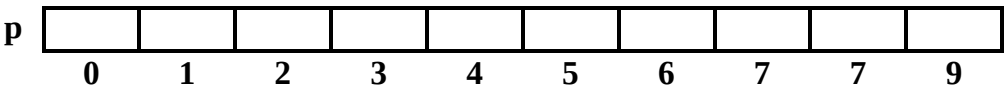
The program to arrange items in ascending order using insertion sort is shown below:

Example 12.3 : C program to arrange items in ascending order using insertion sort

```
#include <stdio.h>

/* Include: Example 2.3: Function to read n elements */
/* Include: Example 2.4: Function to display elements */
/* Include: Example 12.2: Function to sort items using insertion sort */
void main()
{
        int i, n, j, item, a[10];

        printf("Enter the number of elements\n");
        scanf("%d",&n);

        printf("Enter n elements\n");          create_array(a, n);

        insertion_sort(a, n);

        printf("The sorted items : ");         display_array(a, n);
}
```

12.3 Radix sort

Let us arrange numbers in ascending order using radix sort. Since, we are sorting decimal numbers (having base 10), we assume there are 10 pockets ranging from 0 to 9. Initially all 10 pockets are empty as shown below:



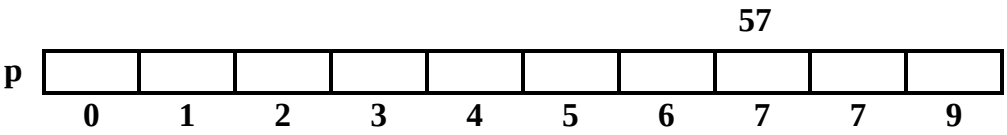
Now, let us see "How to sort the following elements using Radix sort?" Consider the following elements:

57, 45, 67, 91, 28, 79, 35, 68, 89, 20, 62, 43, 84, 55, 86, 96, 78, 25

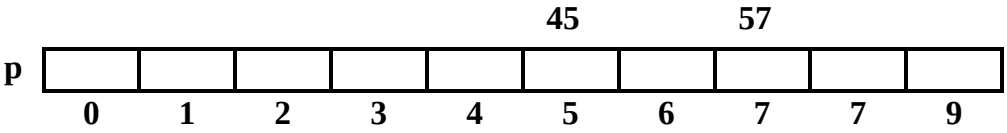
Pass 1: Activities:

Step 1: Scan each *item* (number) in the list. Obtain *least significant digit* say *d.* Insert *item* into pocket at position p[d]. If an *item* is already in p[d] then, insert *item* above the last *item* in p[d]. Each item in the list can be inserted into appropriate pocket as shown below:

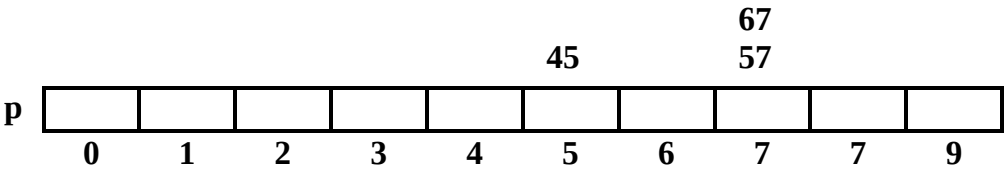
item = 57, *least significant digit d* = 7. So, insert 57 into p[7] as shown below:



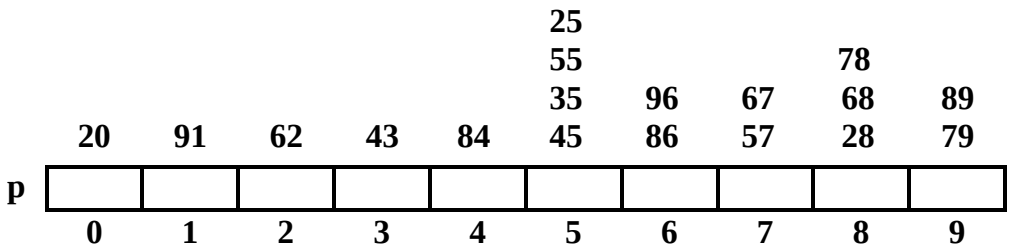
item = 45, *least significant digit d* = 5. So, insert 45 into p[5] as shown below:



item = 67, *least significant digit d* = 7. So, insert 67 into p[5] as shown below:



Thus, all other items are scanned one by one and inserted into appropriate pocket as shown below:

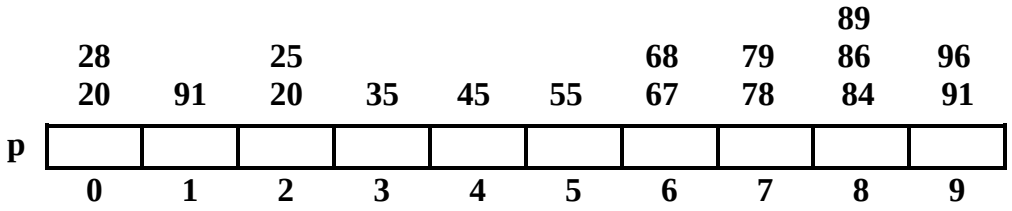


Step 2: Now, access each pocket and copy the items present in each pocket one after the other starting from position 0 through 9 into the original array as shown below:

20, 91, 62, 43, 84, 45, 35, 55, 25, 86, 96, 57, 67, 28, 68, 78, 79, 89

Pass 2: Activities: Consider the above items as input to the pass:

Step 1: Scan each *item* (number) in the above list. Obtain next *least significant digit* say *d*. Insert *item* into pocket at position p[d] as shown below:



Step 2: Now, access the pockets one by one and copy the items present in each pocket one after the other into the original array as shown below:

20, 28, 91, 20, 25, 35, 45, 55, 67, 68, 78, 79, 84, 86, 89, 91, 96

Design: Now, using the above example, let us think of how to design the algorithm or the C function. First, let us see how to separate a digit from a given number say: 789.

Pass 1: Given item 789, in pass 1 we have to separate digit 9 i.e., $789/10^0 \% 10 = 9$

Pass 2: Given item 789, in pass 2 we have to separate digit 8 i.e., $789/10^1 \% 10 = 8$

Pass 3: Given item 789, in pass 3 we have to separate digit 7 i.e., $789/10^2 \% 10 = 7$

So, in pass *j* separate the digit using : item / 10^{j-1} % 10

So, in j^{th} pass, the digit separated is given by $d = item / 10^{j-1} \% 10$ *j* is pass number

So, the function to separate a digit, given *item* and pass number *j* can be written as shown below:

Example 12.4 : Function to separate a digit of a number given pass number *j*

```
int separate_digit ( int item, int j)
{
        return item / ( int ) pow (10, j – 1) % 10;          /*  d = item / 10^(j-1) % 10  */
}
```

Another function insert_rear() is used to insert an element at the rear end of the list. Now, using these functions, perform the following operations for each pass, where pass number depends on number of digits of a maximum number in the list.

Step 1: Initialization: Each location in the packet can be considered as a linked list consisting of one or more items. So, the variable *p* can be considered as an array of 10 linked lists. Initially all 10 pockets contain NULL and it can be represented as shown below:

p	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0
	0	1	2	3	4	5	6	7	8	9

The corresponding code can be written as shown below:

```
for (i = 0; i < 10; i++)
{
        p[i] = '\0';
}
```

Step2: Access an item and insert into appropriate pocket. This can be done by separing jth digit from each number (where j is the pass number) and inserting the item into pocket p[digit] using the following code:

```
for (i = 0; i < n; i++)
{
        digit  = separate_digit(a[i], j);          // separate j^th digit from a[i]
        p[digit] = insert_rear(a[i], p[digit])  // insert a[i] at the end
}
```

Step 3: Copy items from each of the pocket into array: This can be done by accessing each pocket using the following code:

```
for (i = 0; i < 10; i++)
{
        temp = p[i];

        // copy items from each pocket
}
```

Copying item from each pocket is nothing but traversing the linked list till the end and copy all items in the linked list into array a. Now, the above code can be modified as shown below:

```
k = 0;
for (i = 0; i < 10; i++)
{
        temp = p[i];

        while (temp != NULL)                    // copy items from each list
        {
                a[k++] = temp->info;
                temp = temp->link;
        }
}
```

The code given in above three steps have to be executed for each pass j where

$1 \le j \le m$ where *m* is the number of digits in a given number.

Now, the code can be written as shown below:

```
for (j = 1; j <= m; j++)
{
        for (i = 0; i < 10; i++) p[i] = '\0';          // Initialize all pockets to NULL

        for (i = 0; i < n; i++)                        // Access each item
        {
                digit  = separate_digit(a[i], j);      // separate j^th digit from a[i]
                p[digit] = insert_rear(a[i], p[digit]); // insert a[i] at the end
        }
```

```
        k = 0;

        for (i = 0; i < 10; i++)
        {
                temp = p[i];                    // Access each pocket

                while (temp != NULL)            // copy items from each list
                {
                        a[k++] = temp->info;
                        temp = temp->link;
                }
        }
}
```

Now, the value for *m* can be calculated using the statement:

m = log10(big) + 1 where *big* is the largest number in the array.

The value of *m* thus calculated gives total number of passes to made. After executing the statement *m* contains the number of passes required. For example, if big is 234, after the execution of this statement *m* will be 3 which is the number of digits in the number. Now, the C function to sort numbers using radix sort can be written as shown below:

Example 12.5 : Function to sort the numbers in ascending order using radix sort

```
void radix_sort(int a[], int n)
{
    int i, j, k, m, big, digit;

    NODE p[10], temp;

    big = largest (a,n);
    m = log10(big) + 1;

    for ( j = 1; j <= m; j++)
    {
        for ( i = 0; i <= 9; i++ ) p[i] = NULL;    // Initialize all pockets to NULL
```

```
            for ( i = 0; i < n; i++)                    // Access each item
            {
                digit = separate_digit(a[i], j);        // separate j^th digit from a[i]
                p[digit] = insert_rear(a[i], p[digit]); // insert a[i] at the end
            }

            k = 0;

            for ( i = 0; i <= 9; i++ )          // Access each pocket
            {
                temp = p[i];                    // get the list of items

                while ( temp != NULL )          // copy items from each list
                {
                    a[k++] = temp->info;
                    temp = temp->link;
                }
            }
        }
}
```

The function largest which returns the largest item in an array consisting of *n* elements can be written as shown below:

Example 12.6 : Function to find the largest element in an array of *n* elements

```
int largest (int a[], int n)
{
        int i, pos = 0;

        for (i = 1; i < n; i++)
        {
                if (a[i] > a[pos]) pos = i;          // Find the position of largest item
        }

        return a[pos];
}
```

The complete program to arrange numbers in ascending order is shown below:

Example 12.7: C program to arrange numbers in ascending order using Radix Sort

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

struct node
{
        int     info;
        struct node *link;
};

typedef struct node *NODE;

/* Include: Example 2.3: Function to read n elements into array */

/* Include: Example 2.4: Function to write n elements into array */

/* Include: Example 8.2: C Function to get a new node from the availability list */

/* Include: Example 8.6: Function to insert an item at the rear end of the list */

/* Include: Example 12.4: Function to separate digit  */

/* Include: Example 12.5: Function to sort numbers in ascending order */

/* Include: Example 12.6: Function to find the largest element in an array */

void main()
{
        int n, i, a[20];

        printf("Enter n items\n");
        scanf("%d",&n);

        printf("Enter the elements to sort\n");
        create_array(a, n);

        radix_sort(a,n);

        printf("The sorted array is\n");
        display_array(a, n);
}
```

12.4 Address calculation sort

It is discussed in the section: 12.9

12.5 Hashing

We have already seen that the time required to access any element in the array irrespective of its position is same. The physical location of i^{th} item in the array can be calculated by multiplying the size of each element of the array with *i* and adding to the base address as shown below:

$$Loc(A_i) = Base(A) + w*(i - lb)$$

Here, *i* is the index, *w* is the size of each element of the array and *lb* is the lower bound. Using this formula, the address of any item at any specified position *i* can be obtained and from the address obtained, we can access the item. The similar idea is used in *hashing* to store and retrieve the data. ***So, the hashing technique is essentially independent of n where n is number of elements.***

Example 12.8: Create a hash table and using hash function for 5 items:

For example, consider the elements: 11, 12, 13, 14 and 15 and consider the following function:

```
H(k) = k % 5          // Hash function is just a function
```

The value of each function for the items: 10, 11, 12, 13 and 14 can be obtained as shown below:

```
H (k ) =   k   % 5      // Hash function
```

```
H (10) = 10 % 5 = 0 }                  Insert 10 into table a[0]
H (11) = 11 % 5 = 1 }                  Insert 11 into table a[1]
H (12) = 12 % 5 = 2 }  hash values     Insert 12 into table a[2]
H (13) = 13 % 5 = 3 }                  Insert 13 into table a[3]
H (14) = 14 % 5 = 4 }                  Insert 14 into table a[4]
```

Thus, the table obtained after inserting each of the items using hash value as the index is shown below:

a[0]	10
a[1]	11
a[2]	12
a[3]	13
a[4]	14

The table thus created using elements and hash values is the hash table

Now, let us see "What is hash value? What is hash function?"

Definition: A function can be used to provide a mapping between large original data and the smaller table by transforming a key information into an index to the table. The index value returned by this function is called *hash value.* The function that transforms a data into hash value to a table is called ***hash function.***

Now, let us see "What is a hash table?"

Definition: The data can be stored in the form of a table using arrays with the help of hash function which gives a hash value as an index to access any element in the table. This table on which insertion, deletion and retrieve operations takes place with the help of hash value is called ***hash table.***

A *hash table* can be implemented as an array ht[0..m-1]. The size of the hash table is limited and so it is necessary to map the given data into this fairly restricted set of integers. The hash function assigns an integer value from 0 to m-1 to keys and these values which act as index to the hash table are called *hash addresses or hash values.*

Now, let us see "What is hashing?"

Definition: This process of mapping large amounts of data into a smaller table using hash function, hash value and hash table is called *hashing.*

Now, let us see "What are the different types of hashing techniques?" The two types of hashing techniques are:

- Static hashing
- Dynamic hashing

12.6 Static hashing

Now, let us see "What is static hashing?"

Definition: This process of mapping large amounts of data into a table whose size is fixed during compilation time is called *static hashing.* In static hashing, all the data items are stored in a fixed size table *t* called hash table. The hash table is implemented as an array ht[0..m-1] with 0 as the low index and m – 1 as the high index. Each item in the table can be stored in ht[0], ht[1],…….ht[m – 1] where *m* is the size of the table usually with a prime number such as 5, 7, 11 and so on. The items are inserted into the table based on the hash value obtained from the hash function.

12.6.1 Hash table

Now, let us take the following example, where identifiers are inserted into the hash table.

Example 12.9: Construct a hash table *ht* for storing various identifiers.

Solution: We need to store identifiers in a hash table. Let us assume all identifiers starts with only letters from 'A' to 'Z' having the range 0 to 25. Since, there are 26 letters, let us have a hash table *ht* whose size is *m* = 26. Assume identifiers to be inserted into hash table *ht* are: **ant**, **dog**, **cat**, **bat**, **eagle**, **fish** and **ape**. To insert an identifier into the hash table, we require hash function. Let us define hash function as:

```
h(x) = toupper(x[0]) – 65      // range is  0  to 25 for 'a' to 'z'
```

where 65 is the ASCII value of 'A'. Now, let us complete hash value for each of the identifier using the above hash function:

- Hash value of identifier "**a**nt" = 'A' – 65 = 65 – 65 = 0. So, insert "ant" into ht[0]
- Hash value of identifier "**d**og"= 'D' – 65 = 68 – 65 = 3. So, insert "dog" into ht[3]
- Hash value of identifier "**c**at" = 'C' – 65 = 67 – 65 = 2. So, insert "cat" into ht[2]
- Hash value of identifier "**b**at" = 'B' – 65 = 66 – 65 = 1. So, insert "bat" into ht[1]
- Hash value of identifier "**e**agle" = 'E' – 65 = 69 – 65 = 4. So, insert "eagle" into ht[4]
- Hash value of identifier "**f**ish" = 'F' – 65 = 70 – 65 = 5. So, insert "eagle" into ht[5]
- Hash value of identifier "**a**pe" = 'A' – 65 = 65 – 65 = 0. So, insert "ape" into ht[0]

Now, the hash table for first six identifiers is shown below:

ht[0]	ht[1]	ht[2]	ht[3]	ht[4]	ht[5]	ht6]		ht[25]
ant	bat	cat	dog	eagle	fish		….	

- The seventh identifier "ape" whose hash value is 0 cannot be inserted into ht[0] because, an identifier is already placed in ht[0]. This condition is called overflow or collision. How to avoid "overflow" is discussed in the section 12.6.3.1.
- When there is no overflow, the time required to insert, delete or search depends only on the time required to compute the hash function and the time to search on location in ht[i]. Hence, the insert, delete and search times are independent of *n* which is the number of items

12.6.2 Hash functions

A good hash function should satisfy two criteria:

- A hash function should generate the hash addresses such that all the keys are distributed as evenly as possible among the various cells of the hash table.
- Computation of a key by hash function should be simple.

Now, let us see "What are various types of hash functions?" The popular hash functions are:

- Division method
- Mid square method
- Folding method
- Converting keys to integers

12.6.2.1 Division method

In this method, we choose a number *m* which is prime value and it is larger than the number of given elements *n* in array *a*. Here, the key item *k* is divided by some number *m* and the remainder is used as the hash value. Formally, it is written as:

$$h(k) = k \% m$$

Because, we are taking modulo *m* values, the integers we get from the above function are in the range: 0 to m – 1. Refer example 12.8 to know the details of how to create hash table and using the hash value, how an item can be accessed.

12.6.2.2 Mid-square method

In this method, the key *k* is squared. A number in the middle of k^2 is selected by removing the digits from both ends. The hash function is defined as shown below:

$$h(k) = l$$

where *l* is obtained by removing digits from both ends of k^2. By selecting the middle portion of squared number, different keys are expected to give different hash values. Normally, the size of the hash table using this technique will be power of 2.

For example, consider key $k = 2345$. Its square i.e., $k^2 = 574525$

$h(2345) = 45$ by discarding 57 in the beginning and 25 from the end in k^2.

12.6.2.3 Folding method

In this method, the key *k* is divided into number of parts k1, k2, k3,…..kn of same length except the last part. Then all parts are added together as shown below:

h(k) = k1 + k2 + k3 + ……kn

For example, consider key *k* = 123987234876. The partitions are k1 = 12, k2 = 39, k3 = 87, k4 = 23, k5 = 48, k6 = 76. Now, the hash value can be obtained by adding all the partitioned keys as shown below:

h(k) = 12 + 39 + 87 + 23 + 48 + 76 = 285

12.6.2.4 Converting keys to integers

In this method, if key *k* is a string, the hash value can be obtained by converting this string into integer. This can be done by adding all ASCII values of each character in the string. For example, let k = "ABCD". The hash value can be calculated as shown below:

h("ABCD") = 'A' + 'B' + 'C' + 'D' = 206

The equivalent function can be written as shown below:

```
int string_to_int(char k[])
{
        int i, n = 0;

        while (k[i] != '\0')            // Scan the string till the end
        {
                n += k[i];              // Add the ASCII value of each character
                i++;
        }

        return n;                       // hash value of string k is returned
}
```

12.6.3 Overflow handling (Collision and its detection)

First, let us see "When overflow occurs?" or "What is collision during hashing?"

Definition: If the size of the hash table is fixed and the size of the table is smaller than the total number of keys generated, then two or more keys will have the same hash address (also called hash value). This phenomenon of two or more keys being hashed to the same location of hash table is called ***collision***.

Observe the following points:

- We can expect collision even if the number of keys is less than the size of the hash table (See example 12.9).
- Most of the time collision cannot be avoided and in the worst case all keys may have the same hash value. In such situation all the keys may be stored in only one cell in the form of a list and so, it is required to search all *n* keys in the worst case.
- But, with appropriately chosen size of the hash table and using a good hash function, this phenomenon will not occur and under reasonable assumptions, the expected time to search for an element in a hash table will be θ(1).
- So, in practical situation, hashing is extremely effective where insertion, deletion and searching takes place frequently.

Now, let us see "How to avoid overflow in hashing?" or "How to avoid collision?" The various hashing techniques using which collision can be avoided are:

- Open addressing
- Chaining

12.7 Open addressing

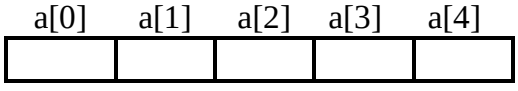
In open addressing hashing, the amount of space available for storing various data is fixed at compile time by declaring a fixed array for the hash table. So, all the keys are stored in this fixed hash table itself without the use of linked lists. In such a table, collision can be avoided by finding another, unoccupied location in the array. The collision can be avoided using ***linear probing***.

Let us create a hash table. To create a hash table, we need the following:

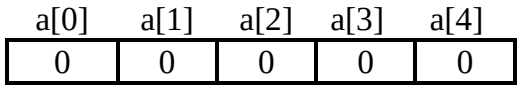
- Initial hash table
- Select the hashing function
- Find the index of each location in the hash table

12.7.1 Create initial hash table

Now, let us see ***"How to create initial hash table?"*** Assume the size of the hash table is HASH_SIZE = 5 and the hash table can be pictorially represented as shown below:



The empty hash table is indicated by storing 0 values in each location as shown below:



The code:

```
for (i = 0; i < HASH_SIZE; i++)
{
        a[i] = 0;
}
```

The function to create initial hash table can be written as shown below:

Example 12.10: Create initial hash table

```
void initialize_hash_table(int a[])
{
        int     i;

        for (i = 0; i < HASH_SIZE; i++) a[i] = 0;
}
```

12.7.2 Identify the hash function

Let the hash function is: H(k) = k % m where *m* is HASH_SIZE. The equivalent code can be written as shown below:

Example 12.11: Compute hash value using the function: **H(k) = k % m**

```
int H(int k)
{
        return k % HASH_SIZE;
}
```

12.7.3 Find the index for hash table given hash value

Now, let us display index of each item. As you know, it is very easy. This can be done using the following code:

```
for (i = 0; i < HASH_SIZE; i++)
{
        printf("%d ", i);                    // 0 1 2 3 4
}
```

Now, the above code can also be written using % operator as shown below:

```
for (i = 0; i < HASH_SIZE; i++)
{
        printf("%d ", i % HASH_SIZE);    // 0 1 2 3 4
}
```

Now, the question is *"I want to display all indices starting from specified position say pos. Hoy we can achieve?"* This can be done by modifying the above code by adding *pos* to *i* and then perform mod operation as shown below:

```
for (i = 0; i < HASH_SIZE; i++)
{
        printf("%d ", (pos + i) % HASH_SIZE);
}
```

Now, let us see the output for various values of *pos*:

- If *pos* = 0, the output will be: 0 1 2 3 4
- If *pos* = 1, the output will be: 1 2 3 4 0
- If *pos* = 2, the output will be: 2 3 4 0 1
- If pos = 3, the output will be: 3 4 0 1 2
- If pos = 4, the output will be: 4 0 1 2 3

So, index to hash table from *pos* is given by: index = (pos + i) % HASH_SIZE

So, Index to hash table from *h_value* is given by: index = (h_value + i) % HASH_SIZE

12.7.4 Insert an item into hash table using linear probing

We can insert an item into the hash table only in the empty position. Now, let us see *"How to find the empty position in the hash table using the hash value?"* This is achieved by accessing each item in the hash table. Each item in the hash table can be accessed using the following code:

```
h_value = H(item);
for (i = 0; i < HASH_SIZE; i++)
{
        index = (h_value + i) % HASH_SIZE;
        a[index];
}
```

The empty slot can be obtained by comparing a[index] with 0 in the above code. The modified code can be written as shown below:

```
h_value = H(item);
for (i = 0; i < HASH_SIZE; i++)
{
        index = (h_value + i) % HASH_SIZE;
        if ( a[index] == 0)     break;          // empty slot found
}
```

When control comes out of the loop, "if a[index] is 0" then we can insert the item. Otherwise, it means that "Hash table is full". The corresponding code can be written as shown below:

```
if ( a[index] == 0)                             // empty slot found
        a[index] = item;
else
        printf("Hash table is full\n");         // No empty slot
```

Now, the complete function can be written as shown below:

Example 12.12: Insert an item into the empty slot using linear probing

```
void insert_hash_table (int item, int a[])
{
        int     i , index, h_value;

        h_value = H(item);
        for (i = 0; i < HASH_SIZE; i++)
        {
                index = (h_value + i) % HASH_SIZE;
                if ( a[index] == 0)     break;          // empty slot found
        }

        if ( a[index] == 0)                             // empty slot found
                a[index] = item;                        // insert item into empty slot
        else
                printf("Hash table is full\n");         // No empty slot
}
```

12.7.5 Search for an item in the hash table

We can easily search for an item in the hash table. We access each item in the hash table using the following code:

```
h_value = H(key);
for (i = 0; i < HASH_SIZE; i++)
{
        index = (h_value + i) % HASH_SIZE;
        a[index]                                    // access each item
}
```

After accessing each item, we check whether *key* to be searched is present in *a[index]*. If found, search is successful. If a[index] is zero, we get unsuccessful search. Now, the above code can be modified as shown below:

```
h_value = H(key);
for (i = 0; i < HASH_SIZE; i++)
{
        index = (h_value + i) % HASH_SIZE;

        if ( key == a[index]) return 1;             // Search successful

        if ( a[index] == 0)     return 0;           // empty slot found. So, key
}                                                   // not found
```

If all elements have been compared and still item is not present, control comes out of the loop and *key* is not present and we display the message "Unsuccessful". Now the above code can be written as shown below:

```
h_value = H(key);
for (i = 0; i < HASH_SIZE; i++)
{
        index = (h_value + i) % HASH_SIZE;

        if ( key == a[index]) return 1;             // Search successful

        if ( a[index] == 0)     return 0;           // empty slot found. So, key
}                                                   // not found

if (i == HASH_SIZE)         return 0;               // Unsuccessful
```

Now, the complete function can be written as shown below:

Example 12.13: Search for an item in the hash table

```
int search_hash_table (int key, int a[])
{
        int     i , index, h_value;

        h_value = H(key);
        for (i = 0; i < HASH_SIZE; i++)
        {
                index = (h_value + i) % HASH_SIZE;

                if ( key == a[index]) return 1;              // Search successful

                if ( a[index] == 0)     return 0;            // empty slot found. So, key
        }                                                    // not found

        if (i == HASH_SIZE)          return 0;               // Unsuccessful
}
```

Now, the complete program to create a hash table, insert an item into the table and to search for a key in the hash table can be written as shown below:

Example 12.14: C program to create hash table and to search for key

```
#include <stdio.h>
#include <stdlib.h>

#define HASH_SIZE 5

/* Insert: Example 12.10: Create initial hash table */

/* Insert: Example 12.11: Compute hash value using the function: H(k) = k % m */

/* Insert: Example 12.12: Insert an item into the empty slot using linear probing */

/* Insert: Example 12.13: Search for an item in the hash table */

/* Insert: Example 2.4: Display the hash table */

void main()
{
        int     a[10], item, key, choice, flag;

        initialize_hash_table(a);              // Create initial hash table
```

```
        for (; ;)
        {
                printf("1: Insert   2: Search\n");
                printf("3: Display  4: Exit\n");
                printf("Enter the choice : ");
                scanf("%d", &choice);

                switch (choice)
                {
                        case 1:printf("Enter the item : ");   scanf("%d", &item);
                                insert_hash_table(item, a);
                                break;

                        case 2: printf("Enter key item : "); scanf("%d", &key);

                                flag = search_hash_table (key, a);

                                if (flag == 0)
                                        printf("Key not found\n");
                                else
                                        printf("Key found\n");

                                break;

                        case 3: printf("Contents of hash table\n");
                                display_array(a, n);
                                printf("\n");
                                break;

                        default: exit(0);
                }
        }
}
```

Example 12.15: Construct a hash table *ht* of size 15 (using open addressing method) to store the following words: like, a, tree, you, first, find, a, place, to, grow, and, then, branch and out

Solution: Now, we "Find hash address for each word". Let us find the hash address by taking the sum of positions of alphabets in the word and taking the remainder by dividing it by 15. The various hash addresses or hash values are shown below:

12.24 💻 Sorting and searching

Sl. No	Words	Sum of positions	hash address = sum %15
1.	like	12+9+11+5 = 37	7
2.	a	1 = 1	1
3.	tree	20+18+5+5 = 48	3
4.	you	25+15+21 = 61	1
5.	first	6+9+18+19+20 = 72	12
6.	find	6+9+14+4 = 33	3
7.	a	1 =1	1
8.	place	16+12+1+3+5 = 37	7
9.	to	20+15 = 35	5
10.	grow	7+18+15+23 = 63	3
11.	and	1+14+4 = 19	4
12.	then	20+8+5+14 = 47	2
13.	branch	2+18+1+14+3+8 = 46	1
14.	out	15+21+20 = 56	11

Creating hash table: The words are taken in the order of serial number from the above table and inserted into the hash table one by one based on the hash address (value or index). **The words "like", "a" and "tree" are inserted into positions 7, 1 and 3** respectively as shown below into hash table:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	a		tree				like								

ht

The next word to be selected is "you" whose hash value is 1 and is already full. So, it is inserted in the next available location 2 as shown below:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	a	you	tree				like								

ht

The word "first" with hash value 12 is inserted into ht[12] a shown below:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	a	you	tree				like					first			

ht

The next word "find" with hash value 3 should be inserted at ht[3]. But, a word is already present. So, insert "find" at next free slot i.e., 4 as shown below:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	a	you	tree	find			like					first			

ht

The next word “a” with hash value 1 is already in ht[1]. So, select the next word “place” with hash value 7. It is already occupied and hence insert at next empty space 8 as shown below:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	a	you	tree	find			like	place				first			

ht

The word “to” with hash value 5 is inserted at position 5 as shown below:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	a	you	tree	find	to		like	place				first			

ht

The word “grow” has the hash value 3 and since the location is full, the next available location is 6 and so it is inserted at position 6 as show below:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	a	you	tree	find	to	grow	like	place				first			

ht

The word “and” has the hash value 4. It is full, next free slot is 9 and hence “and” is inserted at location 9 as shown below:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	a	you	tree	find	to	grow	like	place	and			first			

ht

The word “then” has the hash value 2. It is full. It is inserted into next free location 10 as shown below:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	a	you	tree	find	to	grow	like	place	and	then		first			

ht

The word “branch” has the hash value 1. It is full. It is inserted into next free location 11 as shown below:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	a	you	tree	find	to	grow	like	place	and	then	branch	first			

ht

The word “out” has the hash value 11. It is full. It is inserted into next free location 13 as shown below:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	a	you	tree	find	to	grow	like	place	and	then	branch	first	out		

ht

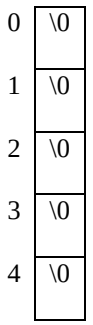
12.8 Chaining

The chaining technique is a very simple method using which collisions can be avoided. This is achieved using an array of linked lists. If an item has to be inserted, using a hash function, the hash value is obtained. This hash value is used to find the list to which the item is to be added and using the appropriate function an item can be inserted at the end of the list. If more than one key has same hash value, all the keys hashed into same hash value will be inserted at the end of the list and thus collision is avoided.

The searching using this technique takes less time. When an item has to be searched, we find the hash value using hash function. This hash value corresponds to an index which will be used to obtain the address of the list. If the address of the list is NULL, the item we search for is not present and report unsuccessful search. If the address of the list is not NULL, that list is traversed sequentially to search for the item. If an item is found in the list, report successful search and if end of list is encountered report unsuccessful search.

Example 12.16: Construct a hash table *ht* of size 5 (using chaining method to avoid collision) and store the following words: like, a, tree, you, first, find, a, place, to, grow, and, then, branch, out

Solution: A hash table of size 5 indicates that it is required to construct a hash table by using an array of 5 linked lists. The empty hash table can be written as shown below:



The code: ⇨ **for** (i = 0; i < HASH_SIZE; i++) a[i] = NULL;

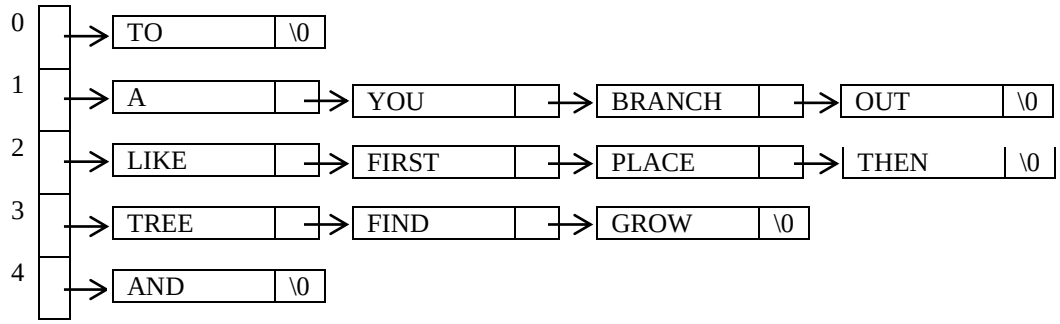
Design of Hashing function: Let us design a simple hashing function which will accept a string consisting of a key and add the positions of each letter of the string and compute the remainder by dividing the sum by the size of the table. The size of the hash table is assumed to be 5 and so while taking the remainder, divide the sum by 5 so that all the hash values of all the words lie within 0 and 4. So, the hash function is given by:

$H(str) = p_0 + p_1 + p_2 \ldots p_{n-1}$ where each p_i is the position of letter in string *str*.

The various hash values for all the words in the given list are shown below:

Sl. No	Words	Sum of positions	hash address = sum %5
1.	like	12+9+11+5 = 37	2
2.	a	1 = 1	1
3.	tree	20+18+5+5 = 48	3
4.	you	25+15+21 = 61	1
5.	first	6+9+18+19+20 = 72	2
6.	find	6+9+14+4 = 33	3
7.	a	1 =1	1
8.	place	16+12+1+3+5 = 37	2
9.	to	20+15 = 35	0
10.	grow	7+18+15+23 = 63	3
11.	and	1+14+4 = 19	4
12.	then	20+8+5+14 = 47	2
13.	branch	2+18+1+14+3+8 = 46	1
14.	out	15+21+20 = 56	1

Construction of Hash table: Now, let us see how to construct a hash table. Since the size of the hash table is 5, we will have an array of 5 rows with each row having a linked list. Obtain the words one by one, find the hash address and insert at the end of the list in the appropriate location in the array. The hash table obtained by computing the hash values is shown below:



Observe that hash value of word 'TO' is 0 and so, it is inserted into h[0]. The hash value of the words 'LIKE', 'FIRST', 'PLACE' and 'THEN' is 2 and the words are inserted into h[2] at the end of the list one by one as and when they are scanned. Thus, a hash table can be created. Now, the equivalent code can be written as shown below:

12.28 💻 Sorting and searching

Example 12.17: Insert an item into hash table (represented as array of linked lists)

```
void insert_hash_table (int item, NODE a[])
{
        int     h_value;

        h_value = H(item);

        a[h_value] = insert_rear (item, a[h_value]);
}
```

Similarly, we can search for a *key* using the following function:

Example 12.18: search for a key in hash table (represented as array of linked lists)

```
int search_hash_table (int key, NODE a[])
{
        int             h_value;
        NODE            cur;

        h_value = H(key);                       // Find the hash value

        cur = search(key, a[h_value]);          // Obtain address of searched node

        if (cur == NULL) return 0;              // Key not found in the list

        return 1;                               // Key found in the list
}
```

Example 12.19: function to display contents of hash table

```
void display_hash_table (NODE a[])
{
        int             i;
        NODE            temp;

        for (i = 0; i < HASH_SIZE; i++)
        {
                printf("a[%d] = ", i);
                temp = a[i];
```

```
                if (temp == NULL)
                        printf("NULL\n");
                else
                {
                        temp = a[i];
                        while (temp->link != NULL)
                        {
                                printf("%d -->", temp->info);
                                temp = temp->link;
                        }
                        printf("%d\n", temp->info);
                }
        }
}
```

Now, the complete program to create a hash table, insert an item into the table and to search for a key in the hash table can be written as shown below:

Example 12.20: C program to create hash table and to search for key

```
#include <stdio.h>
#include <stdlib.h>

#define HASH_SIZE 5

struct node
{
        int             info;
        struct node     *link;
};

typedef struct node *NODE;

/* Insert: Example 8.2: To get a node from the availability list */

/* Insert: Example 8.6: To insert an item at the rear end of the list */

/* Insert: Example 8.13: Search for a key item in the list */

/* Insert: Example 12.11: Compute hash value using the function: H(k) = k % m */

/* Insert: Example 12.17: Insert item into hash table (as array of linked lists)*/
```

/* **Insert: Example 12.18:** Search for an item in the hash table (adjacency list */

/* **Insert: Example 12.19:** Display the hash table */

```
void main()
{
        NODE        a[10];
        int         item, key, choice, flag, i;

        for ( i = 0; i < HASH_SIZE; i++) a[i] = NULL;     // Create initial hash table

        for (; ;)
        {
                printf("1: Insert   2: Search\n");
                printf("3: Display  4: Exit\n");
                printf("Enter the choice : ");
                scanf("%d", &choice);

                switch (choice)
                {
                        case 1:printf("Enter the item : ");   scanf("%d", &item);
                               insert_hash_table(item, a);
                               break;

                        case 2: printf("Enter key item : "); scanf("%d", &key);
                               flag = search_hash_table (key, a);
                               if (flag == 0)
                                       printf("Key not found\n");
                               else
                                       printf("Key found\n");

                               break;

                        case 3: printf("Contents of hash table\n");
                               display_hash_table(a);
                               printf("\n");
                               break;

                        default: exit(0);
                }
        }
}
```

12.9 Address calculation sort

The *Address Calculation Sort* is also called *Hash Sort*. In this technique, a particular kind of hashing is used. The hashing function *hash*() should have the property that if x1 ≤ x2, then *hash*(x1) ≤ *hash*(x2). The function which exhibits this property is called ***order-preserving*** or ***non-decreasing*** hashing function. If this *hashing function* is used to hash an *item* to which some previous *items* have already been hashed, then this *item* is placed in such a way that all the items hashed to a particular value should be arranged in ascending or descending order. Thus all the items in one sub-file will be less than or equal to the elements in the subsequent sub-files. When all the items have been placed in this order into sub-files, finally concatenate the items in each sub-file to get the sorted elements. For example, consider the items shown below.

57, 45, 36, 91, 28, 79, 35, 68, 89, 20, 62, 43, 84, 55, 86

Since the digits available are 0 to 9, we use an array of pointers a[10], where each location in this array points to the first item in the sub-file whose first digit is same. Hash value for this problem can be obtained by selecting the largest number and finding at which position the most significant digit is present. For example, for the item 45 hash value is 10 because the digit 4 is in tens position. For the item 199 hash value is 3 since the digit 1 is in hundredth position. The hash value of a largest number can be obtained using the function shown below:

Example 12.21: C function to find the hash value

```
int hash(int a[], int n)
{
        int     big, i, pos;

        pos = 0;
        for ( i = 1; i < n; i++)                /* Find the largest of all the numbers */
        {
                if ( a[i] > a[pos] ) pos = i;
        }

        big = a[pos];

        i = log10(big);

        return  pow(10,i);    /* The position of largest number */
}
```

After finding the hash value, divide each *item* to be sorted using this hash value and insert the *item* into the appropriate location. For example, if *item* is 45 and hash value is 10, 45/10 is 4. So, insert 45 into the 4th location. If some items are already present in this location, insert it in the appropriate position such that items in that location are in order. For this reason, we use an array of ordered linked list. Once all the items are inserted in this way, obtain the items present in each list in the array one by one and display. The C function to sort the items using *Address Calculation Sort*(*Hash Sort*) is shown below:

Example 12.22: C function for Address Calculation Sort (Hash Sort)

```
void hash_sort(int a[], int n)
{
    NODE b[10], temp;
    int i, j, digit, h_value;

    h_value = hash(a,n);

    for ( i = 0; i < 10; i++) b[i] = NULL;          /* Empty hash table */

    for ( i = 0; i < n; i++)                        /* Insert orderly into hash table */
    {
        digit = a[i] / h_value;

        b[digit] = insert(a[i],b[digit]);
    }

    for( i = j = 0; i < 10; i++)                    /* Copy from hash table to array */
    {
        temp = b[i];
        while ( temp != NULL )
        {
            a[j++] = temp->info;
            temp = temp->link;
        }
    }
}
```

The C program to sort the items using Address calculation sort (hash sort) is shown below:

Example 12.23: C program to sort using Address Calculation Sort (Hash Sort)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

struct node
{
      int                info;
      struct node       *link;
};

typedef struct node* NODE;

/* Include Example 2.3: C Function read array elements */
/* Include Example 2.4: C Function display array elements */
/* Include Example 8.2: C Function to get a new node from the availability list */
/* Include Example 8.17: C function to create an ordered linked list */
/* Include Example 12.21: C function to find the  hash value */
/* Include Example 12.22: C function for Address Calculation Sort (Hash Sort) */

void main()
{
        int  n, a[40];

        printf("Enter the value of n\n");
        scanf("%d",&n);

        printf("Enter the items to sort\n");
        create_array(a, n);

        hash_sort(a,n);

        printf("The sorted vector is\n");
        display_array(a, n);
}
```

12.10 Search for employee details using hashing

Now, let us create a hash table for storing records of employee details consisting of *id* and name of the employee and search for an employee id. The structure declaration can be written as shown below:

```
#define HASH_SIZE 5

typedef struct employee
{
        int             id;
        char            name[20];
} EMPLOYEE;
```

The program designed in the section 12.7 (see section 12.7 for detailed design) is slightly modified to incorporate structures. The complete program is shown below:

Example 12.24: Create a hash table for employee record and search using id

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define HASH_SIZE 5

typedef struct employee
{
        int             id;
        char            name[20];
} EMPLOYEE;

/* Create initial hash table */
void initialize_hash_table(EMPLOYEE a[])
{
        int     i;

        for (i = 0; i < HASH_SIZE; i++)
        {
                a[i].id = 0;
        }
}
```

```
/* Compute hash value using the function: H(k) = k % m */
int H(int k)
{
        return k % HASH_SIZE;
}
/* Insert an item into the empty slot using linear probing */
void insert_hash_table (int id, char name[], EMPLOYEE a[])
{
        int     i , index, h_value;
        h_value = H(id);

        for (i = 0; i < HASH_SIZE; i++)
        {
                index = (h_value + i) % HASH_SIZE;
                if ( a[index].id == 0)                  // empty slot found
                {
                        a[index].id = id;               // insert employee id
                        strcpy(a[index].name, name);    // insert employee name
                        break;
                }
        }
        if (i == HASH_SIZE) printf("Hash table full\n"); // No empty slot
}
/* Search for employee id in the hash table */
int search_hash_table (int key, EMPLOYEE a[])
{
        int     i , index, h_value;
        h_value = H(key);
        for (i = 0; i < HASH_SIZE; i++)
        {
                index = (h_value + i) % HASH_SIZE;
                if ( key == a[index].id) return 1;      // Search successful
                if ( a[index].id == 0)  return 0;       // empty slot found. So, key
        }                                               // not found
        if (i == HASH_SIZE)         return 0;           // Unsuccessful
}
```

```
/* Display the hash table */
void display_hash_table(EMPLOYEE a[], int n)
{
        int     i;

        for (i = 0; i < n; i++)
        {
                printf("a[%d] = %d %s\n", i, a[i].id, a[i].name);
        }
}

void main()
{
        EMPLOYEE  a[10];
        char      name[20];
        int       key, id, choice, flag;

        initialize_hash_table(a);           // Create initial hash table
        for (; ;)
        {
                printf("1: Insert   2: Search\n");
                printf("3: Display  4: Exit\n");
                printf("Enter the choice : ");
                scanf("%d", &choice);

                switch (choice)
                {
                        case 1: printf("Enter emp id : ");     scanf("%d", &id);
                                printf("Enter the name : "); scanf("%s", name);

                                insert_hash_table(id, name, a);

                                break;

                        case 2: printf("Enter key key : ");   scanf("%d", &key);

                                flag = search_hash_table (key, a);

                                if (flag == 0)
                                        printf("Key not found\n");
                                else
                                        printf("Key found\n");

                                break;
```

```
                    case 3: printf("Contents of hash table\n");
                            display_hash_table (a, HASH_SIZE);
                            printf("\n");
                            break;

                    default: exit(0);
                }
        }
}
```

12.38 💻 Sorting and searching

Exercises

1) How insertion sort works? Sort the elements 25 75 40 10 20 using insertion sort
2) Write C function to sort items using insertion sort
3) How to sort the following elements using Radix sort?
4) Write a function to sort the numbers in ascending order using radix sort
5) What is hash value? What is hash function? What is a hash table? What is hashing?
6) What are the different types of hashing techniques?
7) What is static hashing?
8) Construct a hash table ht for storing various identifiers such as “cat”, “rat”, “mat”, “sat”, “pat”, “bat”, “eat”, “fat”, “hat”, “vat” using linear probing whose hash table size is 5
9) What are various types of hash functions?
10) When overflow occurs? or What is collision during hashing?
11) How to avoid overflow in hashing? or How to avoid collision?
12) What is linear probing? Write a function to insert an item into the empty slot using linear probing
13) Write a function to search for an item in the hash table using linear probing
14) Construct a hash table ht of size 15 (using open addressing method) to store the following words: like, a, tree, you, first, find, a, place, to, grow, and, then, branch and out
15) Construct a hash table ht of size 5 (using chaining method to avoid collision) and store the following words: like, a, tree, you, first, find, a, place, to, grow, and, then, branch, out
16) C function for Address Calculation Sort (Hash Sort)