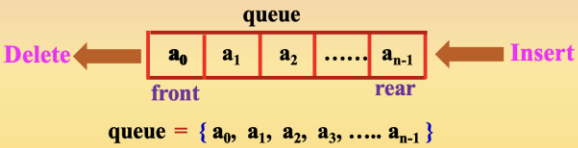
What is a queue? How queue can be represented?

Definition: Queue is a special type of data structure where elements are inserted from one end and are deleted from the other end.

- The elements are inserted into the queue in the order $a_0, a_1, a_2, a_3, \ldots a_{n-1}$
- The end from where the elements are inserted is called REAR END
- Since a_0 is at the front end of queue, it is removed first, then a_1 and so on.
- The end from where the elements are deleted is called FRONT END.
- Using the above approach, the First element Inserted is the First element to be deleted Out.
- Hence, queue is also called FIFO data structure.
- The queue $= \{a_0, a_1, a_2, a_3, \ldots a_{n-1}\}$ is pictorially represented as shown in fig:

Queue can be represented using:

- Arrays
- Linked lists



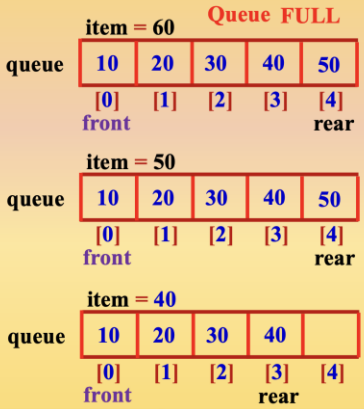
queue $= \{a_0, a_1, a_2, a_3, \ldots a_{n-1}\}$

What are the operations that can be performed on queue?

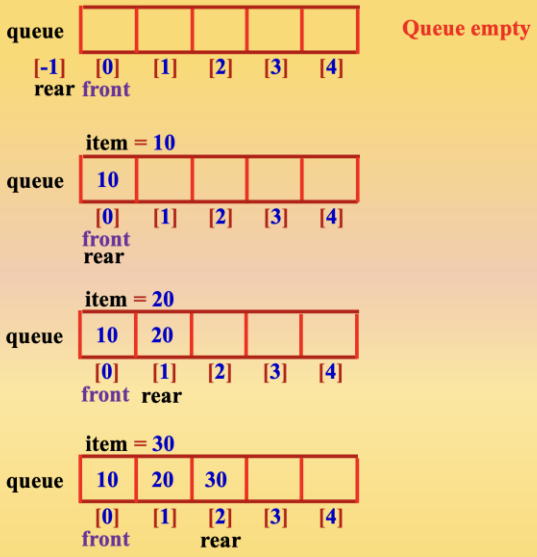
The various operations that can be performed on queue are:

- Insertion : An element is inserted at the rear end.
- Deletion : An element is deleted from the front end.

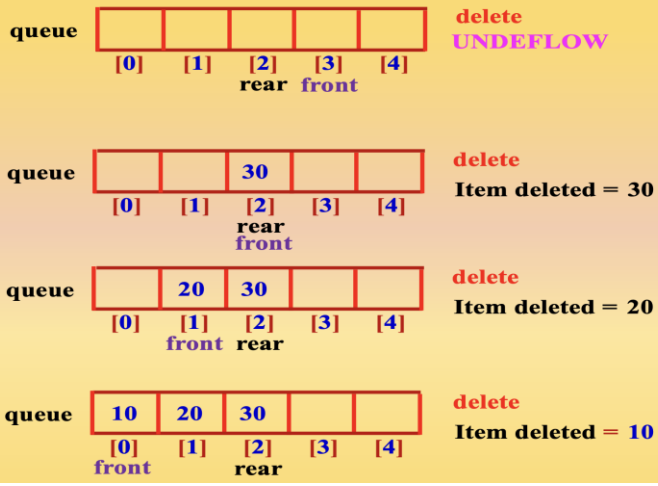
Q_SIZE = 5



Insertion



Deletion



How to insert an element into queue?

```
// Function to insert item into the queue
void  insert_rear ( int  item )
{
      // Check for overflow of queue
      if ( rear ==  Q_SIZE  -  1 )
      {
            printf ("Queue Overflow");
            return;
      }

      // Increment rear by 1
      rear++;
      // Insert an item into the queue
      queue [ rear ] =  item;

            OR

      // Insert an item into the queue
      queue [++rear ] =  item;
}
```

Algorithm insert_rear

// Input: item : element to be inserted

// Global/Parameters : queue, rear

```
// Check for overflow of queue
if ( rear ==  Q_SIZE  -  1 )
print ( "Queue Overflow")
return
```

```
// Increment rear by 1
rear =  rear  + 1  /
rear += 1          /
rear++ / ++rear

// Insert an item at the rear end of queue
queue [ rear ]=  item
```

Case 1: Insertion not possible

Q_SIZE = 5

Queue FULL

item = 60

Case 2: Insertion possible

Before Insertion

item = 40

After Insertion

item = 50

How to design an algorithm to delete an item from queue?

```
// Function to delete an item from queue
void delete_front ( )
{
      // Check for underflow of queue
      if ( front  >  rear)
      {
            printf ("Queue Underflow" );
            return;
      }

      printf ("Item deleted :%d",queue[front]);
      // Increment front by 1
      front =  front  + 1;

            OR

      printf ("Item deleted:%d ",queue[ front++]);

   // Reset to initial values
      if ( front  > rear )  front =  0 , rear  =  -1;
}
```

Algorithm delete_front

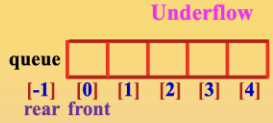
// Input: none

// Global/Parameters: queue, front, rear

```
// Check for underflow of queue
if ( front  > rear )
print ( "Queue Underflow" )
return
```

```
print ("Item deleted =", queue [front] )
// Increment front by 1
front  =  front  + 1  /
front +=1             /
front++ / ++front

if ( front  > rear )
front  =  0
rear  =  -1
```

Case 1: Deletion not possible



Case 2: Deletion possible

Before deletion

delete Item deleted = 20

After deletion

delete Item deleted = 30

How to design an algorithm to display queue contents?

```
// Function to display the contents of queue
void  display ( )
{
      int     i;

      // Check for empty queue
      if ( front  >  rear )
      {
            printf ( "Queue is empty" );
            return;
      }

      printf ( "Queue : " );

      for ( i  =  front;  i <= rear;  i++)
            printf ( " %d     ", queue [i] );

      printf ( "\n " );
}
```

Algorithm display

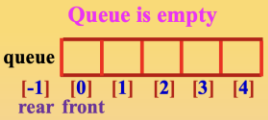
// Input: none

// Global: queue, front, rear

```
// Check for empty queue
if ( front  >  rear )
print ( "Queue is empty" )
return
```

```
print ( "Queue : " )
print  queue  [i]  ∀ i = front   to   rear
print   "\n"
```

Case 1: Display not possible



Case 2: Display possible

How to write a C program to implement queue operations using global variables?

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

#define  Q_SIZE     5

int       front  =  0, rear  =  -1;
int       queue[10];

// Function to insert an item into queue
void   insert_rear ( int  item )
{
    // Write the complete function
}

// Function to delete an element from queue
void   delete_front ( )
{
    // Write the complete function
}

// Function to display the contents of queue
void  display ( )
{
    // Write the complete function
}
```

```
void  main ( )
{
    int        choice, item;
    // Perform queue operations any number of times
    for   ( ;; )
    {
        printf ("1:Insert rear  2:Delete front  3:Display  4:Exit : ");
        scanf ("%d ", &choice );
        switch ( choice )
        {
            case 1 :  printf (" Enter the item : ");
                      scanf ( "%d ", &item );
                      insert_rear ( item ) ;
                      break;
            case 2 :  delete_front ( );
                      break;
            case 3 :  display ( );
                      break;
            default: exit ( 0 );
        }
    }
}
```

How to implement queues using dynamic arrays? (Using global variables)

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

int       Q_SIZE  =  1;
int       front  =  0, rear  =  -1;
int      *queue;

// Function to insert item into queue
void   insert_rear ( int  item )
{
      // Check for overflow of queue
      if ( rear  ==  Q_SIZE  -  1 )
      {
            printf ("Queue Overflow");
            Q_SIZE ++;
            queue = realloc (queue, Q_SIZE * sizeof ( int ) );
      }

      // Insert an item at the rear end of queue
      queue [++rear ]= item;
}
```

```
void  main ( )
{
    int        choice, item;
    queue  =  ( int ) malloc ( Q_SIZE * sizeof ( int ) );
    for   ( ;; )
    {
        printf ("1:Insert rear  2:Delete front  3:Display  4:Exit : ");
        scanf ("%d ", &choice );
        switch ( choice )
        {
            case 1 :  printf (" Enter the item : ");
                      scanf ( "%d ", &item );
                      insert_rear ( item ) ;
                      break;
            case 2 :  delete_front ( );
                      break;
            case 3 :  display ( );
                      break;
            default: exit ( 0 );
        }
    }
}
```

How to insert an element into queue (By passing parameters)?

```
// Function to insert item into the queue
void insert_rear (int item, int  queue[], int *rear)
{
      // Check for overflow of queue
      if (*rear ==  Q_SIZE  -  1 )
      {
            printf ("Queue Overflow" );
            return;
      }

      // Increment rear by 1
      (* rear ) ++;

      // Insert an item into the queue
      queue [*rear ] =  item;

                  OR

      // Insert an item into the queue
      queue [ ++ (*rear ) ] =  item;
}
```

Algorithm insert_rear

// Input: item : to be inserted

// Global/Parameters : queue, rear

```
// Check for overflow of queue
if ( rear ==  Q_SIZE  -  1 )
    print ( "Queue Overflow")
    return
```

```
// Increment rear by 1
rear =  rear  + 1  /
rear += 1          /
rear++ / ++rear

// Insert an item at the rear end q
queue [ rear ]=  item
```

Case 1: Insertion not possible

Q_SIZE = 5

item = 60 — Queue FULL

10	20	30	40	50
[0]	[1]	[2]	[3]	[4]
front				rear

Case 2: Insertion possible

Before Insertion

item = 40

10	20	30	40	
[0]	[1]	[2]	[3]	[4]
front			rear	

After Insertion

item = 50

10	20	30	40	50
[0]	[1]	[2]	[3]	[4]
front				rear

How to design an algorithm to delete an item from queue? (By passing parameters)

```
// Function to delete an item from queue
void delete_front(int queue[], int *front, int *rear)
{
    // Check for underflow of queue
    if ( *front  >  *rear)
    {
        printf ("Queue Underflow" );
        return;
    }

    printf ("Item deleted :%d",queue[*front]);

    // Increment front by 1
    *front =  *front  + 1;

                OR

    printf ("Item deleted:%d ",queue[(*front)++]);

    // Reset to initial values
    if (*front  >*rear ) *front  =  0 , *rear  =  -1;
}
```

Algorithm delete_front

// Input: none

// Global/Parameters: queue, front, rear

```
// Check for underflow of queue
if ( front  > rear )
    print ( "Queue Underflow" )
    return
```

```
print ("Item deleted =", queue [front] )

// Increment front by 1
front =  front  + 1  /
front +=1            /
front++ / ++front

if ( front  > rear )
    front  =  0
    rear   =  -1
```

Case 1: Deletion not possible

Underflow

queue					
	[-1]	[0]	[1]	[2]	[3]	[4]
	rear	front				

Case 2: Deletion possible

Before deletion

delete Item deleted = 20

queue		20	30		
	[0]	[1]	[2]	[3]	[4]
		front	rear		

After deletion

delete Item deleted = 30

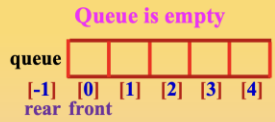
queue			30		
	[0]	[1]	[2]	[3]	[4]
			rear		
			front		

How to design an algorithm to display queue contents? (By passing parameters)

```
// Function to display the contents of queue
void display (int queue[], int front, int rear)
{
      int     i;

      // Check for empty queue
      if (front  >  rear)
      {
            printf ("Queue is empty");
            return;
      }

      printf ( "Queue : " );

      for ( i  =  front;  i <= rear;  i++)
            printf ( " %d     ", queue [i] );

      printf ( "\n " );
}
```

```
Algorithm    display
// Input:    none
// Global/Parameters: queue, front, rear
// Check for empty queue
if ( front  >  rear )
print ( "Queue is empty" )
return

print ("Queue : " )
print  queue  [i]   ∀ i = front   to   rear
print   "\n"
```

Case 1: Display not possible



Case 2: Display possible

How to write a C program to implement queue operations by passing parameters?

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

#define  Q_SIZE     5

// Function to insert an item into queue
void insert_rear (int item, int  queue[], int *rear)
{
    // Write the complete function
}

// Function to delete an item from queue
void delete_front(int queue[], int *front, int *rear)
{
    // Write the complete function
}

// Function to display the contents of queue
void display (int queue[], int front, int rear)
{
    // Write the complete function
}
```

```
void  main ( )
{
    int       choice, item, queue[10],  front  =  0, rear  =  -1;
    // Perform queue operations any number of times
    for   ( ;; )
    {
        printf ("1:Insert rear  2:Delete front 3:Display  4:Exit : ");
        scanf ("%d ", &choice );
        switch ( choice )
        {
            case 1 :  printf (" Enter the item : ");
                      scanf ( "%d ", &item );
                      insert_rear ( item,  queue,  &rear ) ;
                      break;
            case 2 :  delete_front (queue,  &front,  &rear ) ;
                      break;
            case 3 :  display (queue,  front,  rear ) ;
                      break;
            default: exit ( 0 );
        }
    }
}
```

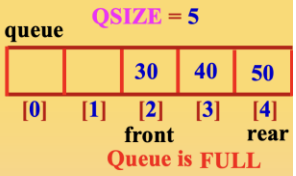
How to write C program to implement dynamic Q operations by passing parameters?

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

int       Q_SIZE  =  1;


 // Function to insert item into the queue
 void insert_rear (int item, int  queue[], int *rear)
 {
       // Check for overflow of queue
       if (*rear ==  Q_SIZE  -  1 )
       {     printf ("Queue Overflow" );
             Q_SIZE ++;
             queue = realloc (queue, Q_SIZE * sizeof ( int ) );
       }
        // Insert an item into the queue
        queue [ ++(*rear ) ] = item;
 }
```

```
void  main ( )
{
    int       choice, item, *queue, front  =  0, rear  =  -1;
    queue  =  ( int ) malloc ( Q_SIZE * sizeof ( int ) );
    for   ( ;; )
    {
        printf ("1:Insert rear  2:Delete front 3:Display  4:Exit : ");
        scanf ("%d ", &choice );
        switch ( choice )
        {
            case 1 :  printf (" Enter the item : ");
                      scanf ( "%d ", &item );
                      insert_rear ( item,  queue,  &rear ) ;
                      break;
            case 2 :  delete_front (queue,  &front,  &rear ) ;
                      break;
            case 3 :  display (queue,  front,  rear ) ;
                      break;
            default: exit ( 0 );
        }
    }
}
```

What is the disadvantage of a queue?

- Once queue is full, even if some elements are deleted from queue, it is not possible to insert the items into queue. The message "Queue is full" is displayed on the screen.
- Shifting queue elements towards left after every deletion, consume lot of time.
- The above disadvantages can be overcome using circular queues.

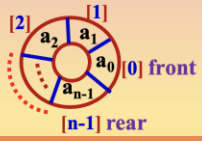


What is a circular queue? How circular queue can be represented?

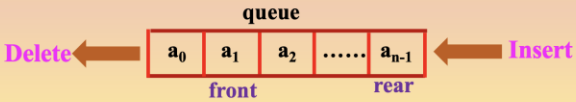
Definition: Circular queue is a special type of data structures where elements are inserted from one end and are deleted from the other end.

- The elements are inserted into the queue in the order a_0, a_1, a_2, a_3, ….. a_{n-1}
- The end from where the elements are inserted is called REAR END
- Since a_0 is at the front end of queue, it is removed first, then a_1 and so on.
- The end from where the elements are deleted is called FRONT END.
- Here, the First element Inserted is the First element to be deleted Out and hence, called FIFO data structure.

Pictorial representation



Array representation



queue = { a_0, a_1, a_2, a_3, ….. a_{n-1} }

What are the operations that can be performed on circular queue?

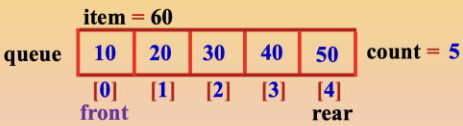
The various operations that can be performed on queue are:

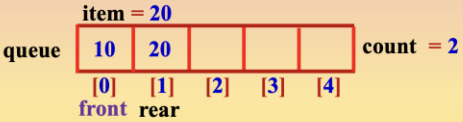
- Insertion : An element is inserted at the rear end.
- Deletion : An element is deleted from the front end.

Insertion

Queue empty

Q_SIZE = 5 Queue FULL





Q_SIZE = 5 Queue FULL

queue | 10 | 20 | 30 | 40 | 50 | count = 5

[0] front [1] [2] [3] [4] rear

delete item deleted = 10

queue | | 20 | 30 | 40 | 50 | count = 4

[0] [1] front [2] [3] [4] rear

delete item deleted = 20

queue | | | 30 | 40 | 50 | count = 3

[0] [1] [2] front [3] [4] rear

delete Queue empty

queue | | | | | | count = 0

[0] front [1] [2] [3] [4] rear

delete item deleted = 40

queue | | | | | 50 | count = 1

[0] [1] [2] [3] [4] rear front

delete item deleted = 30

queue | | | | 40 | 50 | count = 2

[0] [1] [2] [3] front [4] rear

if (count == 0) Queue empty

if (count == Q_SIZE) Queue FULL

Initial values: front = 0 rear = -1

OR

front = 0 rear = Q_SIZE - 1

Show the contents of queue (circular and linear representation) for various operations?

Initial queue = { 10, 20, 30 } Insert (40), Insert (50), Insert (60), Delete(), Delete(), Delete()

Step 1: Initial circular queue Q_SIZE = 5

Circular representation

Array representation

count = 3

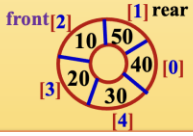
		10	20	30
[0]	[1]	[2] front	[3]	[4] rear

Step2: Insert item = 40

count = 4

40		10	20	30
[0] rear	[1]	[2] front	[3]	[4]

Step 3: Insert item = 50

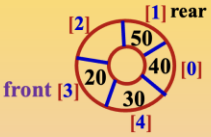


count = 5

40	50	10	20	30
[0] rear	[1]	[2] front	[3]	[4]

Step 4: Insert item = 60 Queue full

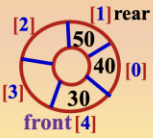
Step 5: Delete Item deleted = 10



count = 4

40	50		20	30
[0]	[1] rear	[2]	[3] front	[4]

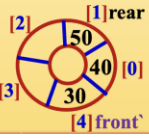
Step 6: Delete Item deleted = 20



count = 3

40	50			30
[0]	[1] rear	[2]	[3]	[4] front

Step 7: Delete Item deleted =



count = 3

40	50			30
[0]	[1] rear	[2]	[3]	[4] front

How to insert an element into circular queue? (Global variables)

```
// Function to insert item into the queue
void  insert_rear (int  item )
{
      // Check for overflow of queue
      if (  count == Q_SIZE )
      {
            printf ("Queue Overflow" );
            return;
      }

      // Increment rear by 1
       rear  =  ( rear + 1 )  %  Q_SIZE;

      // Insert an item into the queue
      queue [  rear ] = item;

      // Update count by 1
       count++;
}
```

Algorithm insert_rear

// Input: item : element to be inserted

// Global/Parameters : queue, rear, count

```
// Check for overflow of queue
if ( count == Q_SIZE )
print ( "Queue Overflow")
return

// Increment rear by 1
rear  =  ( rear + 1 )  %  Q_SIZE;

// Insert an item at the rear end of queue
queue [ rear ]=  item

// Update count by 1
count  =  count + 1
```

Case 1: Insertion not possible

Q_SIZE = 5 Queue FULL

item = 60 count = 5

queue	40	50	10	20	30
	[0]	[1]	[2]	[3]	[4]
	rear		front		

Case 2: Insertion possible

Before Insertion

item = 40 count = 4

queue	40		10	20	30
	[0]	[1]	[2]	[3]	[4]
	rear		front		

After Insertion

item = 50 count = 5

queue	40	50	10	20	30
	[0]	[1]	[2]	[3]	[4]
		rear	front		

How to design an algorithm to delete an item from circular queue? (Global variables)

```
// Function to delete an item from queue
void delete_front ( )
{
     // Check for underflow of queue
     if ( count == 0 )
     {
          printf ("Queue Underflow");
          return;
     }

     // Delete the item from circular queue
     printf ("Item deleted :%d",queue[front]);

     // Increment front by 1
     front = (front +1) %  Q_SIZE;

     // Update count by 1
     count  =  count -  1;
}
```

Algorithm delete_front

Input: none

Global/Parameters: queue, front, count

```
// Check for underflow of queue
if ( count  ==  0 )
print ( "Queue Underflow" )
return

print ("Item deleted =", queue [front] )

// Increment front by 1
front = ( front + 1 )  %  Q_SIZE

// Update count by 1
count  =  count -  1
```

Case 1: Deletion not possible

count = 0

delete Underflow

queue					
	[0]	[1]	[2]	[3]	[4]
	rear	front			

Case 2: Deletion possible

Before deletion count = 2

queue	20	30			
	[0]	[1]	[2]	[3]	[4]
	front	rear			

After deletion count = 1

delete Item deleted = 20

queue		30			
	[0]	[1]	[2]	[3]	[4]
		rear			
		front			

How to design an algorithm to display contents of circular queue? (Global variables)

```
// Function to display the contents of queue
void  display ( )
{
      int     i, temp;

      // Check for empty queue
      if (count  ==  0 )
      {
            printf ("Queue is empty");
            return;
      }

      printf ( "Queue : " );
      temp  =  front;
      for ( i  =  1;  i <= count;  i++)
      {
          printf (" %d     ", queue [temp] );
          temp  =  (temp  +1) %  Q_SIZE;
      }
      printf ( "\n " );
}
```

Algorithm display

// Input: none

// Global/Parameters: queue, front, count

```
// Check for empty queue
if ( count  ==  0 )
print ( "Queue is empty" )
return

print ("Queue : " )
temp  =  front
∀ i  =  1  to  count
print  queue [temp]
temp  =  (temp  +1) %  Q_SIZE
```

Case 1: Display not possible

count = 0

Underflow

queue					
	[-1]	[0]	[1]	[2]	[3]	[4]
	rear	front	rear	front		rear

Case 2: Display possible

count = 3

queue	20	30			10
	[0]	[1]	[2]	[3]	[4]
		rear			front

How to write a C program to implement circular queue using global variables?

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

#define  Q_SIZE     5

int        front = 0, rear = -1,  count = 0;
int        queue[10];

// Function to insert an item into circular queue
void  insert_rear ( int  item )
{
    // Write the complete function
}

// Function to delete an element from queue
void  delete_front ( )
{
    // Write the complete function
}

// Function to display the contents of queue
void  display ( )
{
    // Write the complete function
}
```

```
void  main ( )
{
    int        choice, item;
    // Perform queue operations any number of times
    for  ( ;; )
    {
        printf ( "1:Insert rear  2:Delete front 3:Display  4:Exit : " );
        scanf ( "%d ", &choice );
        switch ( choice )
        {
            case 1 :  printf (" Enter the item : ");
                      scanf ( "%d ", &item );
                      insert_rear ( item ) ;
                      break;
            case 2 :  delete_front ( );
                      break;
            case 3 :  display ( );
                      break;
            default: exit ( 0 );
        }
    }
}
```

How to insert an element into circular queue? (Passing Parameters)

```
// Function to insert item into the queue
void  insert_rear (int  item, int *queue,
                   int *rear , int *count )
{
    // Check for overflow of queue
    if (*count == Q_SIZE )
    {
        printf ("Queue Overflow" );
        return;
    }

    // Increment rear by 1
    *rear = (*rear + 1 ) % Q_SIZE;

    // Insert an item into the queue
    queue [*rear ] = item;

    // Update count by 1
    *count = *count + 1;
}
```

```
Algorithm   insert_rear
// Input:   item : element to be inserted
// Global/Parameters :queue, rear, count
// Check for overflow of queue
if ( count == Q_SIZE )
print ( "Queue Overflow")
return

// Increment rear by 1
rear = ( rear + 1 ) % Q_SIZE;

// Insert an item at the rear end of queue
queue [ rear ]= item

// Update count by 1
count = count + 1
```

Case 1: Insertion not possible

Q_SIZE = 5 Queue FULL

item = 60 count = 5

	[0]	[1]	[2]	[3]	[4]
queue	40	50	10	20	30
		rear	front		

Case 2: Insertion possible

Before Insertion

item = 40 count = 4

	[0]	[1]	[2]	[3]	[4]
queue	40		10	20	30
	rear		front		

After Insertion

item = 50 count = 5

	[0]	[1]	[2]	[3]	[4]
queue	40	50	10	20	30
		rear	front		

```c
// Function to delete an item from queue
void delete_front ( int *queue,  int *front,
                                 int *count)
{
     // Check for underflow of queue
     if (*count == 0 )
     {
          printf ("Queue Underflow" );
          return;
     }

     // Delete the item from circular queue
     printf ("Item deleted :%d",queue[*front]);

     // Increment front by 1
     *front = (*front  + 1) %  Q_SIZE;

     // Update count by 1
     *count = *count -  1;
}
```

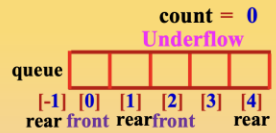
```
Algorithm   delete_front
Input:   none
Global/Parameters:  queue, front, count

// Check for underflow of queue
if ( count  ==  0 )
    print ( "Queue Underflow" )
    return

print ("Item deleted =", queue [front] )

// Increment front by 1
front =  ( front + 1 )  %  Q_SIZE

// Update count by 1
count =  count -  1
```

How to design an algorithm to display contents of C queue? (Passing parameters)

```c
// Function to display the contents of queue
void  display ( int *queue,  int   front,
                             int   count)
{
     int     i;

     // Check for empty queue
     if (count  == 0 )
     {
          printf ( "Queue is empty" );
          return;
     }

     printf ( "Queue : " );
     for ( i = 1;  i <= count;  i++)
     {
          printf ( " %d    ", queue [front] );
          front =  ( front  + 1) %  Q_SIZE;
     }
     printf ( "\n " );
}
```

```
Algorithm   display
// Input:   none
// Global/Parameters: queue, front, count
   // Check for empty queue
   if ( count  ==  0 )
       print ( "Queue is empty" )
       return

   print ("Queue : " )

   ∀ i  =  1  to  count
       print  queue  [front]
       front =  ( front  + 1) %  Q_SIZE
```

Case 1: Display not possible



Case 2: Display possible

How to write a C program to implement circular queue by passing parameters?

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

#define  Q_SIZE    5

// Function to insert an item into C queue
void  insert_rear (int  item, int *queue,
{                  int *rear , int *count)
    // Write the complete function
}

// Function to delete an element from queue
void delete_front ( int *queue,  int *front,
{                                int *count)
    // Write the complete function
}

void  display ( int *queue,  int   front,
{                            int   count)
    // Write the complete function
}
```

```c
void  main ( )
{
    int     choice, item, queue[10],  front  =  0, rear  =  -1,  count =  0;
    // Perform queue operations any number of times
    for  ( ;; )
    {
        printf ( "1:Insert rear  2:Delete front 3:Display  4:Exit : " );
        scanf ( "%d ", &choice );
        switch ( choice )
        {
            case 1 :  printf ( " Enter the item : " );
                      scanf ( "%d ", &item );
                      insert_rear ( item,  queue,  &rear, &count ) ;
                      break;
            case 2 :  delete_front (queue,  & front,  &count) ;
                      break;
            case 3 :  display (queue,  front,  count ) ;
                      break;
            default: exit ( 0 );
        }
    }
}
```

How to insert an element into circular queue dynamically? (Global variables)

Case 1: front < rear

Before Insertion

Q_SIZE = 5 Queue FULL

item = 60 count = 5

10	20	30	40	50
[0]	[1]	[2]	[3]	[4]
front				rear

After Insertion

Q_SIZE = 6 Queue FULL

item = 60 count = 6

10	20	30	40	50	60
[0]	[1]	[2]	[3]	[4]	[5]
front					rear

Case 2: front > rear

Before Insertion

Q_SIZE = 5 Queue FULL

item = 60 count = 5

10	20	30	40	50
[0]	[1]	[2]	[3]	[4]
	rear	front		

After Insertion

Q_SIZE = 6 Queue FULL

item = 60 count = 6

10	20	60	30	40	50
[0]	[1]	[2]	[3]	[4]	[5]
		rear	front		

```
Algorithm insert_rear
// Input: item : element to be inserted
// Global/Parameters : queue, front, rear, count
// Check for overflow of queue
if ( count == Q_SIZE )
    print ( "Queue Overflow")
    Q_SIZE++
    queue = realloc ( Q_SIZE )
if ( front > rear)
    src = &queue[front];
    dest = src + 1;
    no_of_byts = (count - front) * sizeof(int)
    memcpy ( dest, src, no_of_byts)
    front++
// Increment rear by 1
rear = ( rear + 1 ) % Q_SIZE
// Insert an item at the rear end of queue
queue [ rear ]= item
// Update count by 1
count = count + 1
```

How to insert an element into circular queue dynamically? (Global variables)

```
void insert_rear (int item )
{
    // Check for overflow of queue and allocate extra one memory location
    if ( count == Q_SIZE )
    {
        printf ("Queue Overflow" );
        Q_SIZE++;
        queue = realloc (queue, Q_SIZE * sizeof ( int ) );
        if ( front > rear)
        {
            src = &queue[front];
            dest = src + 1;
            no_of_bits = (count - front) * sizeof(int);
            memcpy ( dest, src, no_of_bits );
            front++;
        }
    }
    // Increment rear by 1
    rear = ( rear + 1 ) % Q_SIZE;
    // Insert an item into the queue
    queue [ rear ] = item;
    // Update count by 1
    count = count + 1;
}
```

```
Algorithm insert_rear
// Input: item : element to be inserted
// Global/Parameters : queue, front, rear, count
// Check for overflow of queue
if ( count == Q_SIZE )
    print ( "Queue Overflow")
    Q_SIZE++
    queue = realloc ( Q_SIZE )
if ( front > rear)
    src = &queue[front];
    dest = src + 1;
    no_of_byts = (count - front) * sizeof(int)
    memcpy ( dest, src, no_of_byts)
    front++
// Increment rear by 1
rear = ( rear + 1 ) % Q_SIZE
// Insert an item at the rear end of queue
queue [ rear ]= item
// Update count by 1
count = count + 1
```

How to design an algorithm to delete an item from circular queue? (Global variables)

```
// Function to delete an item from queue
void delete_front ( )
{
    // Check for underflow of queue
    if ( count == 0 )
    {
        printf ("Queue Underflow");
        return;
    }

    // Delete the item from circular queue
    printf ("Item deleted :%d",queue[front]);

    // Increment front by 1
    front = (front +1) % Q_SIZE;

    // Update count by 1
    count = count - 1;
}
```

Algorithm delete_front

- **Input:** none
- **Global/Parameters:** queue, front, count

```
// Check for underflow of queue
if ( count == 0 )
    print ( "Queue Underflow" )
    return

print ("Item deleted =", queue [front] )

// Increment front by 1
front = ( front + 1 ) % Q_SIZE

// Update count by 1
count = count - 1
```

Case 1: Deletion not possible

count = 0

delete Underflow

Case 2: Deletion possible

Before deletion count = 2

After deletion count = 1

delete Item deleted = 20

How to design an algorithm to display contents of circular queue? (Global variables)

```
// Function to display the contents of queue
void display ( )
{
    int  i, temp;

    // Check for empty queue
    if (count == 0 )
    {
        printf ("Queue is empty");
        return;
    }

    printf ( "Queue : " );
    temp = front;
    for ( i = 1; i <= count; i++)
    {
        printf (" %d  ", queue [temp] );
        temp = (temp +1) % Q_SIZE;
    }
    printf ( "\n " );
}
```

Algorithm display

- **// Input:** none
- **// Global/Parameters:** queue, front, count

```
// Check for empty queue
if ( count == 0 )
    print ( "Queue is empty" )
    return

print ("Queue : " )
temp = front
∀ i = 1 to count
    print queue [front]
    temp = (temp +1) % Q_SIZE
```

Case 1: Display not possible

count = 0

Underflow

Case 2: Display possible

count = 3

How to insert an element into circular queue dynamically? (Passing parameters)

```
void insert_rear (int item, int *queue, int *front, int *rear, int *count )
{
    int    *src, *dest, no_of_byts;
    // Check for overflow of queue and allocate extra one memory location
    if (*count == Q_SIZE )
    {
        printf ("Queue Overflow" );
        Q_SIZE++;
        queue = realloc (queue, Q_SIZE * sizeof ( int ) );
        if (*front > *rear)
        {
            src  = &queue[*front];
            dest =  src + 1;
            no_of_byts = (*count - *front ) * sizeof(int);
            memcpy ( dest, src, no_of_byts);
            ( *front ) ++ ;
        }
    }
    // Increment rear by 1
    *rear = (*rear + 1 ) % Q_SIZE;
    // Insert an item into the queue
    queue [*rear ] = item;
    // Update count by 1
    *count = *count + 1;
}
```

```
Algorithm insert_rear
// Input:  item : element to be inserted
// Global/Parameters : queue, front, rear, count
// Check for overflow of queue
if ( count == Q_SIZE )
    print ( "Queue Overflow")
    Q_SIZE++
    queue = realloc ( Q_SIZE )
if ( front > rear)
    src  = &queue[front];
    dest =  src + 1;
    no_of_byts = (count - front) * sizeof(int)
    memcpy ( dest, src, no_of_byts)
    front++
// Increment rear by 1
rear = ( rear + 1 ) % Q_SIZE
// Insert an item at the rear end of queue
queue [ rear ]= item
// Update count by 1
count = count + 1
```

How to design an algorithm to delete an item from C queue? (Passing parameters)

```
// Function to delete an item from queue
void delete_front ( int *queue, int *front, int *count )
{
    // Check for underflow of queue
    if (*count == 0 )
    {
        printf ("Queue Underflow" );
        return;
    }

    // Delete the item from circular queue
    printf ("Item deleted :%d",queue[*front]);

    // Increment front by 1
    *front = (*front +1) % Q_SIZE;

    // Update count by 1
    *count = *count - 1;
}
```

```
Algorithm delete_front
Input:  none
Global/Parameters:  queue, front, count

// Check for underflow of queue
if ( count == 0 )
    print ( "Queue Underflow" )
    return

print ("Item deleted =", queue [front] )

// Increment front by 1
front = ( front + 1 ) % Q_SIZE

// Update count by 1
count = count - 1
```

How to design an algorithm to display contents of C queue? (Passing parameters)

```
// Function to display the contents of queue
void display ( int *queue, int front, int count )
{
    int    i;

    // Check for empty queue
    if (count == 0 )
    {
        printf ("Queue is empty" );
        return;
    }

    printf ( "Queue : " );
    for ( i = 1; i <= count; i++)
    {
        printf (" %d  ", queue [front] );
        front = (front +1) % Q_SIZE;
    }
    printf ( "\n " );
}
```

```
Algorithm display
// Input:  none
// Global/Parameters: queue, front, count
    // Check for empty queue
    if ( count == 0 )
        print ( "Queue is empty" )
        return

    print ("Queue : " )

    ∀ i = 1 to count
        print queue [front]
        front = (front +1) % Q_SIZE
```

How to implement circular queue operations dynamically in C by passing parameters?

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

int       Q_SIZE = 1;

// Function to insert an item into C queue
void  insert_rear (int  item, int *queue,
{                       int  *front,  int *rear , int *count)
    // Write the complete function
}

// Function to delete an element from queue
void delete_front ( int *queue,  int *front,
{                                          int *count)
    // Write the complete function
}

void  display ( int *queue,  int  front,
{                                    int  count)
    // Write the complete function
}
```

```
void  main ( )
{
    int     choice, item, *queue,  front  =  0, rear  =  -1,  count =  0;
    queue  =  ( int )  malloc ( Q_SIZE  *  sizeof ( int ) );
    for  ( ;; )
    {
        printf ( "1:Insert rear  2:Delete front  3:Display  4:Exit : " );
        scanf ( "%d ", &choice );
        switch ( choice )
        {
            case 1 :  printf ( " Enter the item : " );
                      scanf ( "%d ", &item );
                      insert_rear ( item, queue, &front, &rear, &count);
                      break;
            case 2 :  delete_front (queue,  & front,  &count) ;
                      break;
            case 3 :  display (queue,  front,  count ) ;
                      break;
            default:  exit ( 0 );
        }
    }
}
```

What are the advantages and disadvantages of arrays?

```
int  a [500]  =  { 10,  20,  ..... 40,  50 } ;
```

	2000	2004	…..	3992	3996
a	10	20	.....	40	50
	[0]	[1]	….	[498]	[499]

Advantages of arrays

- Any data structure can be represented very easily and usage of array is very simple.
- Data can be accessed very fast just by specifying array name and the corresponding index. Location of a[i] is given by:

  Location of A [i] = Base address + W * (i - LB)

  The time taken to access a[0] is same as the time taken to access a[10000].

Disadvantages of arrays

- Arrays uses static memory allocation technique i.e., even though memory is allocated in the stack area when the program is being executed, its size is fixed by the compiler during compile time. So, array size cannot be increased to accommodate more data and its size cannot be decreased to accommodate less data during execution.
- Inserting an element into ith position is very time consuming since all elements towards to right of ith element are to be moved towards right to make room to insert at ith position.
- Deleting an element from ith position is very time consuming since all elements towards to right of ith element are to be moved towards left by one position.

What is a linked list? Explain with example

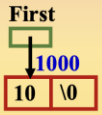
Definition: A linked list is a data structure which is a collection of zero, one or more nodes where each node is connected to next node by a link which contains address of the next node.

- Each node has two fields namely: info and link
- The info field represent the data to be accessed or manipulated.
- The link field contains address of next node.
- Using the address of the first node any node in the list can be accessed.
- The link field of the last node contains NULL and it is denoted by NULL or \0. If we use NULL, we need to include header file "stdio.h". In "stdio.h", it is declared as:

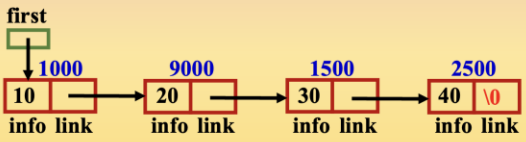
```
#define NULL \0
```

Empty list:

List with one node:



List with more than one node:



Pictorial representation of linked list

How to represent a node of linked list? What is a self-referential structure?

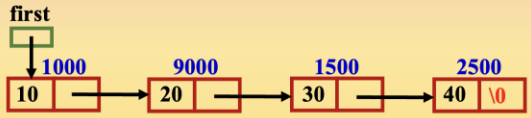
Representation of a node

- A structure is a collection of one or more fields which are of same type or different types.
- Each node has two fields namely: info and link
- The info field represent the data to be accessed or manipulated.
- The link field contains address of next node.
- A node can be represented in C/C++ language using structure as shown below:

```
// C++ syntax
struct node
{
    int     info;
    node    *link;
};
```

```
// C syntax
struct node
{
    int             info;
    struct node     *link;
};
```

Definition: A self-referential structure is a structure which has at least one field which is a pointer to itself (i.e., a structure with one or more pointers pointing to the same structure).



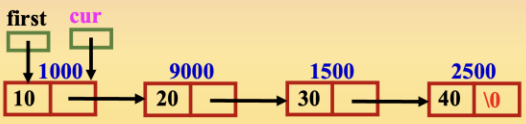
How to declare a variable which contains address of a node?

- A structure is a collection of one or more fields which are of same type or different types.
- Each node has two fields namely: info and link
- The info field represent the data to be accessed or manipulated.
- The link field contains address of next node.
- A node can be represented in C/C++ language using structure as shown below:

```
// C++ syntax
struct node
{
    int     info;
    node    *link;
};
```

```
// C syntax
struct node
{
    int             info;
    struct node     *link;
};
```

- The variable which contains address of a node must be declared as a pointer to a structure.
- The variable first contains address of a node. So, it must be declared as a pointer to a node.



Method 1:

```
struct node *first;
struct node *cur ;
```

Method 2:

```
typedef struct node * NODE;

NODE    first;
NODE    cur ;
```

How to access various fields of a node?

```
// C++ syntax
struct node
{
    int     info;
    node   *link;
};
```

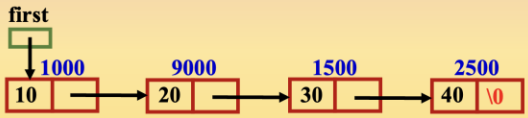
```
// C syntax
struct node
{
    int          info;
    struct node *link;
};
```

Method 1:
```
struct node *first;
struct node *cur ;
```

Method 2:
```
typedef struct node * NODE;
NODE first;
NODE cur ;
```

❖ Using the variable *first*, the *info* and *link* fields can be accessed by writing:

```
(*first) . info     // 10
(*first) . link     // 9000
        OR
first -> info       // 10
first -> link       // 9000
```



How to get address of subsequent nodes?

```
// C++ syntax
struct node
{
    int     info;
    node   *link;
};
```

```
// C syntax
struct node
{
    int          info;
    struct node *link;
};
```

Method 1:
```
struct node *first;
struct node *cur ;
```

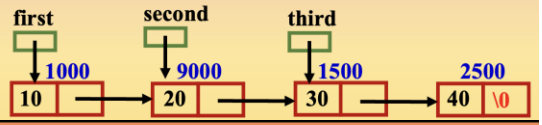
Method 2:
```
typedef struct node * NODE;
NODE first;
NODE cur ;
```

❖ The variable *second* can point to the 2nd node using one of the following statements:

```
second = (*first) . link
        OR
second = first -> link
```

❖ The variable *third* can point to the 3rd node using one of the following statements:

```
third = (*second) . link
        OR
third = second -> link
```



What is an empty list? What is non-empty linked list?

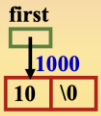
Definition: A linked list is with zero nodes is called an empty linked list. In other words, a pointer variable which contains NULL is called an empty linked list. A linked list with one or more nodes is called non-empty linked list. In other words, a pointer variable which contains address of a node is called non-empty linked list. The code to check for empty linked list and non-empty linked list can be written as shown below:

```
// Check for empty list
if ( first == NULL )
{
    printf ("List is empty\n");
    return;
}
```

Empty list:

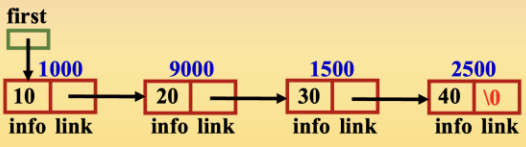

```
// Check for only one node
if ( first -> link == NULL )
{
    printf ("List has one node");
    return;
}
```

List with one node:




```
// Check for more than one node
if ( first -> link != NULL )
{
    printf ("List has more than one node");
    return;
}
```

List with more than one node:



Pictorial representation of linked list

How to design a function to traverse a singly linked list?

Algorithm display (first)

Step 1: //Check for empty list
if (first == NULL)
print ("List is empty")
return

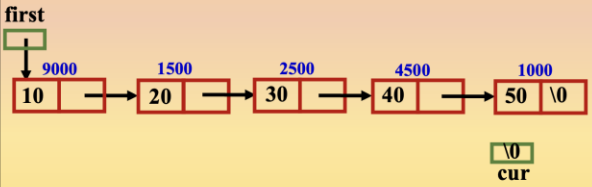
Step 2: //Copy the address of first node
cur = first;

Step 3: //visit each node and print
while (cur != NULL)
print (info[cur])
cur = link[cur]

Step 4: //Finished
return

Case 1: List is empty

Case 2: List is not empty



How to write a C function to traverse a singly linked list?

Algorithm display (first)

Step 1: //Check for empty list
if (first == NULL)
print ("List is empty")
return

Step 2: //Copy the address of first node
cur = first;

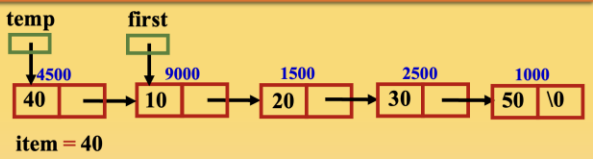
Step 3: //visit each node and print
while (cur != NULL)
print (info[cur])
cur = link[cur]

Step 4: //Finished
return

```
void display ( NODE first )
{
    NODE cur;
    printf ("List : " );
    //Check for empty list
    if ( first == NULL )
    {
        printf ("Empty ");
        return;
    }
    //Copy the address of first node
    cur = first;

    //visit each node and print
    while ( cur != NULL )
    {
        printf ("%d ---> ", cur->info );
        cur = cur -> link ;
    }
    printf ( "NULL");
}
```

How to write a C function to insert an item at the front end of the list?

```
#include <stdio..h>
#include <stdlib.h>
struct  node
{
        int             info;
        struct  node  * link;
};
typedef  struct  node *  NODE;

// Function to get a node from the heap
NODE  getnode ( )
{
     NODE  x;
     // Get a node from the heap
     x  =  ( NODE ) malloc ( sizeof (struct  node) ) ;
     // Check for overflow of heap
     if ( x ==  NULL )
     {
          printf ("Not enough memory");
          exit (0) ;
     }
     // Return address of node
     return  x;
}
```



item = 40

```
// Function to insert an item at the front end of the list
NODE  insert_front  ( int  item,  NODE  first )
{
     NODE  temp;
     // Allocate memory for a node
     temp  =  getnode ( );
     // Copy the item into the new node
     temp –> info =  item;
     // Insert new node at the front end
     temp –> link =  first;
     // Return address of the first node of the list
     return  temp;
}
```

How to write a C function to insert an item at the front end of the list?

```
#include <stdio..h>
#include <stdlib.h>
struct  node
{
        int             info;
        struct  node  * link;
};
typedef  struct  node *  NODE;

// Function to get a node from the heap
NODE  getnode ( ) ;

// Function to insert an item at the front end of the list
NODE  insert_front  ( int  item,  NODE  first ) ;

// Function to display the contents of list
void  display ( NODE  first );
```

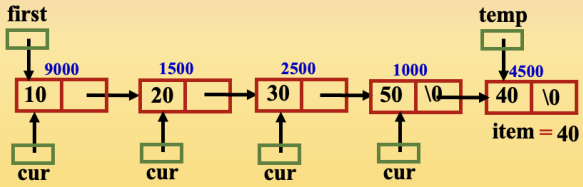
```
void  main ( )
{
     int       choice, item;
     NODE   first;
     first  =  NULL;
     for  ( ;; )
     {
          printf ( " 1:Insert Front   2:Display  3:Exit : " );
          scanf ( "%d ", &choice );
          switch ( choice )
          {
               case 1 :  printf ( " Enter the item : " );
                         scanf ( "%d ", &item );
                         first = insert_front ( item,  first ) ;
                         break;
               case 2 :  display ( first );
                         break;
               default: exit ( 0 );
          }
     }
}
```

How to write a C function to insert an item at the rear end of the list?

Case 1: List is empty


```
// Function to insert an item at the front end of the list
NODE   insert_rear( int item,  NODE  first )
{
       NODE   temp,  cur ;
       // Create a node with item in it
       temp  =  getnode ( );
       temp –> info  =  item;
       temp –> link  =  NULL;
       // Insert the node for the first time
       if ( first  ==  NULL )  return  temp;
```

Case 2: List not empty




```
       // Find the address of last node in the list
       cur  =  first;
       while ( cur –> link  != NULL )
       {
            cur  =  cur –> link ;
       }

       // Insert the node at the end of the list
       cur –> link  =  temp ;

       // Return address of the first node
       return  first;
}
```


How to write a C function to insert an item at the rear end of the list?

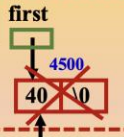
```
#include <stdio..h>
#include <stdlib.h>
struct  node
{
      int                info;
      struct  node  * link;
};
typedef  struct  node *  NODE;

// Function to get a node from the heap
NODE  getnode ( ) ;

// Function to insert an item at the front end of the list
NODE   insert_rear( int  item,  NODE  first ) ;

// Function to display the contents of list
void  display ( NODE  first ) ;
```

```
void  main ( )
{
    int       choice, item;
    NODE   first;
    first  =  NULL;
    for  ( ;; )
    {
        printf ( " 1:Insert Rear    2:Display  3:Exit : " );
        scanf ( "%d ", &choice );
        switch ( choice )
        {
            case 1 :  printf ( " Enter the item : " );
                      scanf ( "%d ", &item );
                      first = insert_rear  ( item,  first ) ;
                      break;

            case 2 :  display ( first );
                      break;

            default: exit ( 0 );
        }
    }
}
```

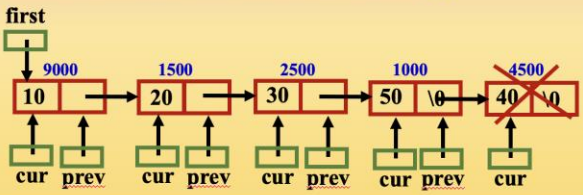
How to write a C function to delete an item from the rear end of the list?

Case 1: List is empty

Case 2: Only one node in the list



Case 3: More than one node in the list

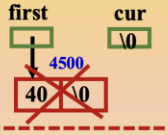



```
// Function to delete an item from the rear end of the list
NODE delete_rear (NODE first)
{
    NODE cur, prev;

    // Check for empty list
    if (first == NULL)
    {
        printf ("List is empty\n");
        return NULL;
    }

    // Delete if there is only one node
    if (first -> link == NULL)
    {
        printf (" Item deleted = %d \n ", first -> info);
        free (first), return NULL;
    }

    // Find address of last and last but one node in the list
    cur = first;
    while (cur -> link != NULL)
    {
        prev = cur,
        cur = cur -> link;
    }
    prev -> link = NULL;      // Make last but node as last node
    printf (" Item deleted = %d \n ", cur -> info);
    free (cur);               // Delete the last node
    return first;             // Return address of first node
}
```

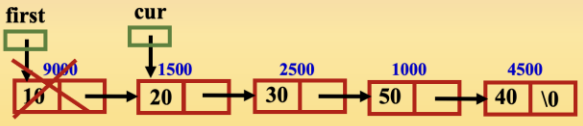
How to write a C function to delete an item from the front end of the list?

Case 1: List is empty

Case 2: Only one node in the list



Case 3: More than one node in the list




```
// Function to delete an item from the rear end of the list
NODE delete_front (NODE first)
{
    NODE cur, prev;

    // Check for empty list
    if (first == NULL)
    {
        printf ("List is empty\n");
        return NULL;
    }

    // Delete if there is only one node
    if (first -> link == NULL)
    {
        printf (" Item deleted = %d \n ", first -> info);
        free (first), return NULL;
    }

    cur = first -> link;      // Obtain address of 2nd node
    printf (" Item deleted = %d \n ", first -> info);
    free (first);             // Remove the first node
    return cur;               // Return 2nd as first node
}
```

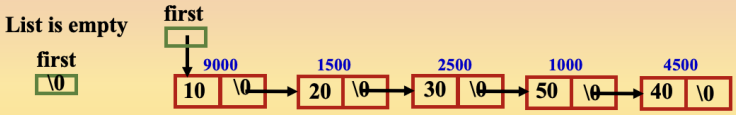
How stack is implemented using linked list?

Definition: Stack is a special type of data structure where elements are inserted from one end and elements are deleted from the same end.

- In this data structure, Last element Inserted into the stack is the First element removed Out of the stack
- Hence, stack is also called LIFO data structure.
- To implement stacks using linked list, the following functions are called:
  - insert_rear()
  - delete_rear()
  - display()

OR

  - insert_front()
  - delete_front()
  - display()



Dr. Padma Reddy A.M | Sai Vidya Institute of Technology | Bengaluru | download: nandipublications.com, saividya.ac.in

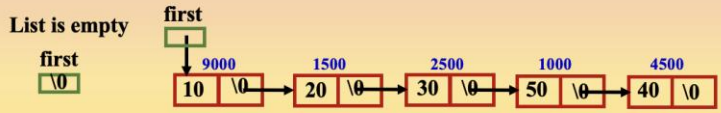
How queue is implemented using linked list?

Definition: Queue is a special type of data structure where elements are inserted from one end and elements are deleted from the other end.

- In this data structure, First element Inserted into queue is the First element to be removed Out of queue.
- Hence, queue is also called FIFO data structure.
- To implement queues using linked list, the following functions are called:
  - insert_rear()
  - delete_front()
  - display()

OR

  - insert_front()
  - delete_rear()
  - display()



How double ended queue is implemented using linked list?

Definition: Dequeue is a special type of data structure where elements are inserted from both ends and elements are deleted from both ends.

- To implement dequeues using linked list, the following functions are called:
  - insert_rear()
  - insert_front()
  - delete_rear()
  - delete_front()
  - display()