Lecture Notes on
Analysis and Design of Algorithms
BCS401

Module-3: Greedy Method

Contents

1. Introduction to Greedy method

1.1 General method

The greedy method is the straight forward design technique applicable to variety of applications.

The greedy approach suggests constructing a solution through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached. On each step the choice made must be:

- ***feasible***, i.e., it has to satisfy the problem's constraints
- ***locally optimal***, i.e., it has to be the best local choice among all feasible choices available on that step
- ***irrevocable***, i.e., once made, it cannot be changed on subsequent steps of the algorithm

As a rule, greedy algorithms are both intuitively appealing and simple. Given an optimization problem, it is usually easy to figure out how to proceed in a greedy manner, possibly after considering a few small instances of the problem. What is usually more difficult is to prove that a greedy algorithm yields an optimal solution (when it does).

```
Algorithm Greedy(a, n)
// a[1 : n] contains the n inputs.
{
    solution := ∅; // Initialize the solution.
    for i := 1 to n do
    {
        x := Select(a);
        if Feasible(solution, x) then
            solution := Union(solution, x);
    }
    return solution;
}
```

Greedy method control abstraction for the subset paradigm

1.2. Coin Change Problem

Problem Statement: Given coins of several denominations find out a way to give a customer an amount with **fewest** number of coins.

Example: if denominations are 1,5,10, 25 and 100 and the change required is 30, the solutions are,

Amount : 30

Solutions : 3 x 10 (3 coins), 6 x 5 (6 coins)

1 x 25 + 5 x 1 (6 coins) **1 x 25 + 1 x 5 (2 coins)**

The last solution is the **optimal** one as it gives us change only with 2 coins.

Solution for coin change problem using greedy algorithm is very intuitive and called as cashier's algorithm. Basic principle is: **At every iteration for search of a coin, take the largest coin which can fit into remain amount to be changed at that particular time.** At the end you will have optimal solution.

1.3. Knapsack Problem (Fractional knapsack problem)

Let us try to apply the greedy method to solve the knapsack problem. We are given n objects and a knapsack or bag. Object i has a weight w_i and the knapsack has a capacity m. If a fraction x_i, $0 \leq x_i \leq 1$, of object i is placed into the knapsack, then a profit of $p_i x_i$ is earned. The objective is to obtain a filling of the knapsack that maximizes the total profit earned. Since the knapsack capacity is m, we require the total weight of all chosen objects to be at most m. Formally, the problem can be stated as

$$\text{maximize} \sum_{1 \leq i \leq n} p_i x_i \tag{4.1}$$

$$\text{subject to} \sum_{1 \leq i \leq n} w_i x_i \leq m \tag{4.2}$$

$$\text{and } 0 \leq x_i \leq 1, \quad 1 \leq i \leq n \tag{4.3}$$

The profits and weights are positive numbers.

A feasible solution (or filling) is any set $(x_1, \ldots, x_n)$ satisfying (4.2) and (4.3) above. An optimal solution is a feasible solution for which (4.1) is maximized.

Consider the following instance of the knapsack problem:

n=3, m=20, (p_1, p_2, p_3) =(25, 24, 15), (w_1, w_2, w_3) =(18, 15, 10)

There are several greedy methods to obtain the feasible solutions. Three are discussed here

a) At each step fill the knapsack with the object with **largest profit -** If the object under consideration does not fit, then the fraction of it is included to fill the knapsack. This method does not result optimal solution. As per this method the solution to the above problem is as follows;

Select Item-1 with profit p_1=25, here w_1=18, x_1=1. Remaining capacity = 20-18 = 2
Select Item-2 with profit p_1=24, here w_2=15, x_1=2/15. Remaining capacity = 0
Total profit earned = 28.2.
Therefore optimal solution is (x_1, x_2, x_3) = (1, 2/15, 0) with profit = 28.2

b) At each step fill the object with **smallest weight**

Select Item-3 with profit p_1=15, here w_1=10, x_3=1. Remaining capacity = 20-10 = 10
Select Item-2 with profit p_1=24, here w_2=15, x_1=10/15. Remaining capacity = 0
Total profit earned = 31.
Optimal solution using this method is (x_1, x_2, x_3) = (0, 2/3, 1) with profit = 31

Note: Optimal solution is not guaranteed using method a and b

c) At each step include the object with **maximum profit/weight ratio**

Select Item-2 with profit p_1=24, here w_2=15, x_1=1. Remaining capacity = 20-15=5

Select Item-3 with profit p_1=15, here w_1=10, x_1=5/10. Remaining capacity = 0

Total profit earned = 31.5

Therefore, optimal solution is (x_1, x_2, x_3) = (0, 1, 1/2) with profit = 31.5

This greedy approach always results ***optimal solution***.

Algorithm: The algorithm given below assumes that the objects are sorted in non-increasing order of profit/weight ratio

```
void GreedyKnapsack(float m, int n)
// p[1:n] and w[1:n] contain the profits and weights
// respectively of the n objects ordered such that
// p[i]/w[i] >= p[i+1]/w[i+1]. m is the knapsack
// size and x[1:n] is the solution vector.
{
    for (int i=1; i<=n; i++) x[i] = 0.0; // Initialize x.
    float U = m;
    for (i=1; i<=n; i++) {
       if (w[i] > U) break;
       x[i] = 1.0;
       U -= w[i];
    }
    if (i <= n) x[i] = U/w[i];
}
```

Analysis:

Disregarding the time to initially sort the object, each of the above strategies use O(n) time,

0/1 Knapsack problem

[0/1 Knapsack] Consider the knapsack problem discussed in this section. We add the requirement that $x_i = 1$ or $x_i = 0$, $1 \le i \le n$; that is, an object is either included or not included into the knapsack. We wish to solve the problem

$$\max \sum_{1}^{n} p_i x_i \quad \text{subject to} \sum_{1}^{n} w_i x_i \le m \text{ and } \quad x_i = 0 \text{ or } 1, \ 1 \le i \le n$$

One greedy strategy is to consider the objects in order of nonincreasing density p_i/w_i and add the object into the knapsack if it fits.

Note: The greedy approach to solve 0/1 knapsack problem does not necessarily yield an optimal solution

1.4. Job sequencing with deadlines

We are given a set of n jobs. Associated with job i is an integer deadline $d_i \geq 0$ and a profit $p_i > 0$. For any job i the profit p_i is earned iff the job is completed by its deadline. To complete a job, one has to process the job on a machine for one unit of time. Only one machine is available for processing jobs. A feasible solution for this problem is a subset J of jobs such that each job in this subset can be completed by its deadline. The value of a feasible solution J is the sum of the profits of the jobs in J, or $\sum_{i \in J} p_i$. An optimal solution is a feasible solution with maximum value. Here again, since the problem involves the identification of a subset, it fits the subset paradigm.

Example 4.2 Let $n = 4$, $(p_1, p_2, p_3, p_4) = (100, 10, 15, 27)$ and $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$. The feasible solutions and their values are:

	feasible solution	processing sequence	value
1.	(1, 2)	2, 1	110
2.	(1, 3)	1, 3 or 3, 1	115
3.	(1, 4)	4, 1	127
4.	(2, 3)	2, 3	25
5.	(3, 4)	4, 3	42
6.	(1)	1	100
7.	(2)	2	10
8.	(3)	3	15
9.	(4)	4	27

Solution 3 is optimal. In this solution only jobs 1 and 4 are processed and the value is 127. These jobs must be processed in the order job 4 followed by job 1. Thus the processing of job 4 begins at time zero and that of job 1 is completed at time 2. □

The greedy strategy to solve job sequencing problem is, "At each time select the job that that satisfies the constraints and gives **maximum profit.** i.e consider the jobs in the non-increasing order of the p_i's"

By following this procedure, we get the 3rd solution in the example 4.3. It can be proved that, this greedy strategy always results optimal solution

```
Algorithm GreedyJob(d, J, n)
// J is a set of jobs that can be completed by their deadlines.
{
    J := {1};
    for i := 2 to n do
    {
        if (all jobs in J ∪ {i} can be completed
            by their deadlines) then J := J ∪ {i};
    }
}
```

High level description of job sequencing algorithm

Algorithm/Program 4.6: Greedy algorithm for sequencing unit time jobs with deadlines and profits

```
1   int JS(int d[], int j[], int n)
2   // d[i]>=1, 1<=i<=n are the deadlines, n>=1. The jobs
3   // are ordered such that p[1]>=p[2]>= ... >=p[n]. J[i]
4   // is the ith job in the optimal solution, 1<=i<=k.
5   // Also, at termination d[J[i]]<=d[J[i+1]], 1<=i<k.
6   {
7       d[0] = J[0] = 0; // Initialize.
8       J[1] = 1; // Include job 1.
9       int k=1;
10      for (int i=2; i<=n; i++) {
11      // Consider jobs in nonincreasing
12      // order of p[i]. Find position for
13      // i and check feasibility of insertion.
14         int r = k;
15         while ((d[J[r]] > d[i]) && (d[J[r]] != r)) r--;
16         if ((d[J[r]] <= d[i]) && (d[i] > r)) {
17            // Insert i into J[].
18            for (int q=k; q>=(r+1); q--) J[q+1] = J[q];
19            J[r+1] = i; k++;
20         }
21      }
22      return (k);
23  }
```

Analysis:

For JS there are two possible parameters in terms of which its complexity can be measured. We can use n, the number of jobs, and s, the number of jobs included in the solution J. The **while** loop of line 15 in Algorithm 4.6 is iterated at most k times. Each iteration takes $\Theta(1)$ time. If the conditional of line 16 is true, then lines 19 and 20 are executed. These lines require $\Theta(k-r)$ time to insert job i. Hence, the total time for each iteration of the **for** loop of line 10 is $\Theta(k)$. This loop is iterated $n-1$ times. If s is the final value of k, that is, s is the number of jobs in the final solution, then the total time needed by algorithm JS is $\Theta(sn)$. Since $s \leq n$, the worst-case time, as a function of n alone is $\Theta(n^2)$. If we consider the job set $p_i = d_i = n - i + 1$, $1 \leq i \leq n$, then algorithm JS takes $\Theta(n^2)$ time to determine J. Hence, the worst-case computing time for JS is $\Theta(n^2)$. In addition to the space needed for d, JS needs $\Theta(s)$ amount of space for J. Note that the profit values are not needed by JS. It is sufficient to know that $p_i \geq p_{i+1}$, $1 \leq i < n$.

Fast Job Scheduling Algorithm

The computing time of JS can be reduced from $O(n^2)$ to nearly $O(n)$ by using the disjoint set union and find algorithms and a different method to determine the feasibility of a partial solution. If J is a feasible subset of jobs, then we can determine the processing times for each of the jobs using the rule: if job i hasn't been assigned a processing time, then assign it to the slot $[\alpha - 1, \alpha]$, where α is the largest integer r such

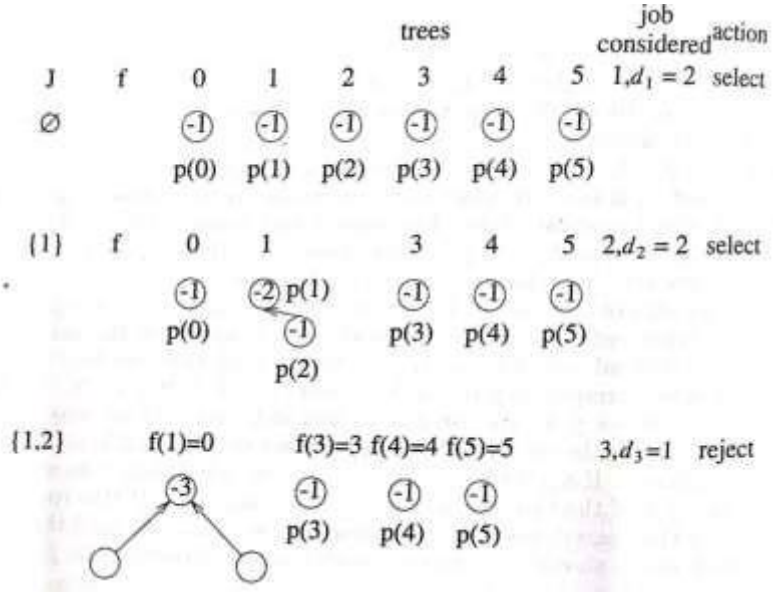
that $1 \leq r \leq d_i$ and the slot $[\alpha - 1, \alpha]$ is free. This rule simply delays the processing of job i as much as possible. Consequently, when J is being built up job by job, jobs already in J do not have to be moved from their assigned slots to accommodate the new job. If for the new job being considered there is no α as defined above, then it cannot be included in J. The proof of the

Example 4.3 Let $n = 5, (p_1, \ldots, p_5) = (20, 15, 10, 5, 1)$ and $(d_1, \ldots, d_5) = (2, 2, 1, 3, 3)$. Using the above feasibility rule, we have

J	assigned slots	job considered	action	profit
$\emptyset$	none	1	assign to [1, 2]	0
{1}	[1, 2]	2	assign to [0, 1]	20
{1, 2}	[0, 1], [1, 2]	3	cannot fit; reject	35
{1, 2}	[0, 1], [1, 2]	4	assign to [2, 3]	35
{1, 2, 4}	[0, 1], [1, 2], [2, 3]	5	reject	40

The optimal solution is $J = \{1, 2, 4\}$ with a profit of 40. □

Example 4.4 The trees defined by the $p(i)$'s for the first three iterations in Example 4.3 are shown in Figure 4.4. □



Algorithm: Fast Job Scheduling is shown in next page

Analysis

The fast algorithm appears as FJS (Algorithm 4.7). Its computing time is readily observed to be $O(n\alpha(2n, n))$ (recall that $\alpha(2n, n)$ is the inverse of Ackermann's function defined in Section 2.5). It needs an additional $2n$ words of space for f and p.

Algorithm: Fast Job Scheduling

```
Algorithm FJS(d, n, b, j)
// Find an optimal solution J[1 : k]. It is assumed that
// p[1] ≥ p[2] ≥ ··· ≥ p[n] and that b = min{n, max_i(d[i])}.
{
    // Initially there are b + 1 single node trees.
    for i := 0 to b do f[i] := i;
    k := 0; // Initialize.
    for i := 1 to n do
    { // Use greedy rule.
        q := CollapsingFind(min(n, d[i]));
        if (f[q] ≠ 0) then
        {
            k := k + 1; J[k] := i; // Select job i.
            m := CollapsingFind(f[q] − 1);
            WeightedUnion(m, q);
            f[q] := f[m]; // q may be new root.
        }
    }
}
```

Problem: Find solution generated by job sequencing problem with deadlines for 7 jobs given profits 3, 5, 20, 18, 1, 6, 30 and deadlines 1, 3, 4, 3, 2, 1, 2 respectively.

Solution: Given

	J_1	J_2	J_3	J_4	J_5	J_6	J_7
Profit	3	5	20	18	1	6	30
Deadline	1	3	4	3	2	1	2

Sort the jobs as per the decreasing order of profit

	J_7	J_3	J_4	J_6	J_2	J_1	J_5
Profit	30	20	18	6	5	3	1
Deadline	2	4	3	1	3	1	2

Maximum deadline is 4. Therefore create 4 slots. Now allocate jobs to highest slot, starting from the job of highest profit

Select Job 7 – Allocate to slot-2
Select Job 3 – Allocate to slot-4
Select Job 4 – Allocate to slot-3
Select Job 6 – Allocate to slot-1

Slot	1	2	3	4
Job	J_6	J_7	J_4	J_3

Total profit earned is = 30+20+18+6=74

Problem: What is the solution generated by job sequencing when n = 5, (P1, P2, P3, P4, P5) = (20, 15, 10, 5, 1), (d1, d2, d3, d4, d5) = (2, 2, 1, 3, 3)

Solution

The Jobs are already sorted according to decreasing order of profit.

Maximum deadline is 3. Therefore create 4 slots. Allocate jobs to highest slot, starting from the job of highest profit

Select Job 1 – Allocate to slot-2
Select Job 2 – Allocate to slot-1 as 2 is already filled
Select Job 3 –Slot-2 &1 are already filled. Cannot be allocated.
Select Job 4 – Allocate to slot-3

Slot	1	2	3
Job	J_2	J_1	J_4

Total profit earned is = 20+15+5=40

2. Minimum cost spanning trees

Definition: A **spanning tree** of a connected graph is its connected acyclic subgraph (i.e., a tree) that contains all the vertices of the graph. A **minimum spanning tree** of a weighted connected graph is its spanning tree of the smallest weight, where the weight of a tree is defined as the sum of the **weights** on all its edges. The **minimum spanning tree problem** is the problem of finding a minimum spanning tree for a given weighted connected graph.

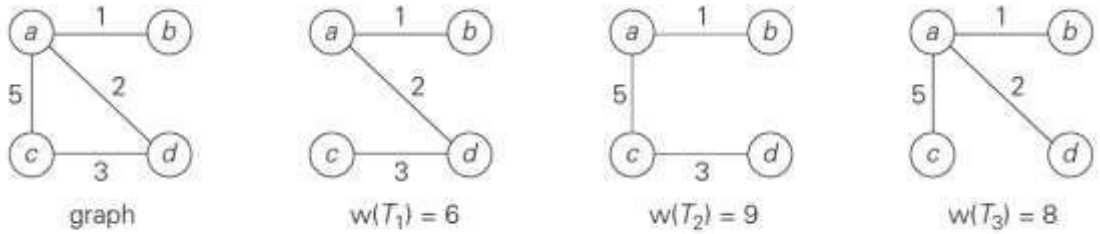


FIGURE 9.2 Graph and its spanning trees, with T_1 being the minimum spanning tree.

2.1. Prim's Algorithm

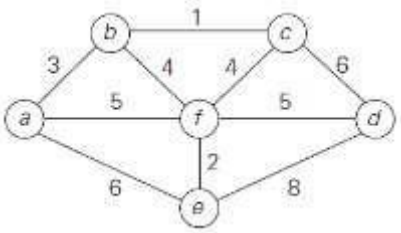
Prim's algorithm constructs a minimum spanning tree through a sequence of expanding subtrees. The initial subtree in such a sequence consists of a single vertex selected arbitrarily from the set V of the graph's vertices. On each iteration it expands the current tree in the greedy manner by simply attaching to it the nearest vertex not in that tree. The algorithm stops after all the graph's vertices have been included in the tree being constructed. Since the algorithm expands a tree by exactly one vertex on each of its iterations, the total number of such iterations is n - 1, where n is the number of vertices in the graph. The tree generated by the algorithm is obtained as the set of edges.

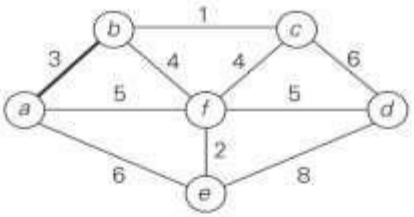
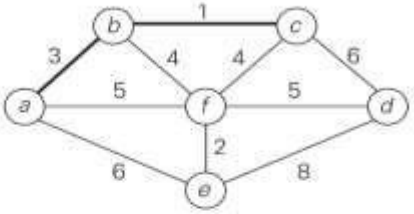
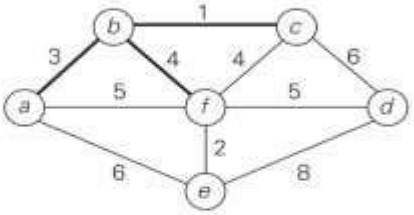
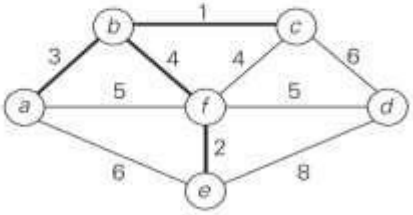
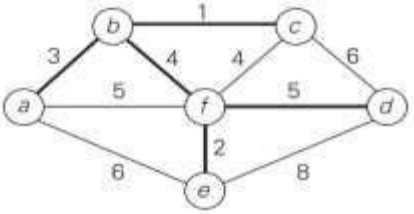
ALGORITHM *Prim(G)*

//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
$V_T \leftarrow \{v_0\}$ //the set of tree vertices can be initialized with any vertex
$E_T \leftarrow \varnothing$
for $i \leftarrow 1$ **to** $|V| - 1$ **do**
  find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges (v, u) such that v is in V_T and u is in $V - V_T$
  $V_T \leftarrow V_T \cup \{u^*\}$
  $E_T \leftarrow E_T \cup \{e^*\}$
return E_T

Correctness: Prim's algorithm always yields a minimum spanning tree.

Example: An example of prim's algorithm is shown below. The parenthesized labels of a vertex in the middle column indicate the nearest tree vertex and edge weight; selected vertices and edges are shown in bold.



Tree vertices	*Remaining vertices*	*Illustration*
a(−, −)	**b(a, 3)** c(−, ∞) d(−, ∞) e(a, 6) f(a, 5)	
b(a, 3)	**c(b, 1)** d(−, ∞) e(a, 6) f(b, 4)	
c(b, 1)	d(c, 6) e(a, 6) **f(b, 4)**	
f(b, 4)	d(f, 5) **e(f, 2)**	
e(f, 2)	**d(f, 5)**	
d(f, 5)		

Analysis of Efficiency

The efficiency of Prim's algorithm depends on the data structures chosen for the **graph** itself and for the **priority queue** of the set $V - V_T$ whose vertex priorities are the distances to the nearest tree vertices.

1. If a graph is represented by its **weight matrix** and the priority queue is implemented as an **unordered array**, the algorithm's running time will be in $\Theta(|V|^2)$. Indeed, on each

of the |V| − 1iterations, the array implementing the priority queue is traversed to find and delete the minimum and then to update, if necessary, the priorities of the remaining vertices.

We can implement the priority queue as a **min-heap**. (A min-heap is a complete binary tree in which every element is less than or equal to its children.) Deletion of the smallest element from and insertion of a new element into a min-heap of size n are **O(log n)** operations.

2. If a graph is represented by its **adjacency lists** and the priority queue is implemented as a **min-heap**, the running time of the algorithm is in **O(|E| log |V |).**

This is because the algorithm performs |V| − 1 deletions of the smallest element and makes |E| verifications and, possibly, changes of an element's priority in a min-heap of size not exceeding |V|. Each of these operations, as noted earlier, is a O(log |V|) operation. Hence, the running time of this implementation of Prim's algorithm is in

(|V| − 1+ |E|) O (log |V |) = O(|E| log |V |) because, in a connected graph, |V| − 1≤ |E|.

2.2. Kruskal's Algorithm

Background: Kruskal's algorithm is another greedy algorithm for the minimum spanning tree problem that also always yields an optimal solution. It is named Kruskal's algorithm, after Joseph Kruskal. Kruskal's algorithm looks at a minimum spanning tree for a weighted connected graph G = (V, E) as an acyclic sub graph with |V | - 1 edges for which the **sum of the edge weights is the smallest**. Consequently, the algorithm constructs a minimum spanning tree as an expanding sequence of sub graphs, which are always **acyclic** but are not necessarily connected on the intermediate stages of the algorithm.

Working: The algorithm begins by **sorting** the graph's edges in **non-decreasing** order of their **weights**. Then, starting with the empty subgraph, it scans this sorted list adding the next edge on the list to the current sub graph if such an inclusion **does not create a cycle** and simply **skipping the edge otherwise.**

ALGORITHM *Kruskal(G)*

```
//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph G = ⟨V, E⟩
//Output: E_T, the set of edges composing a minimum spanning tree of G
sort E in nondecreasing order of the edge weights w(e_{i_1}) ≤ ··· ≤ w(e_{i_|E|})
E_T ← ∅;  ecounter ← 0        //initialize the set of tree edges and its size
k ← 0                         //initialize the number of processed edges
while ecounter < |V| − 1 do
    k ← k + 1
    if E_T ∪ {e_{i_k}} is acyclic
        E_T ← E_T ∪ {e_{i_k}};  ecounter ← ecounter + 1
return E_T
```

The fact that E_T ,the set of edges composing a minimum spanning tree of graph G actually a tree in Prim's algorithm but generally just an acyclic sub graph in Kruskal's algorithm.

Kruskal's algorithm is **not simpler** because it has to check whether the addition of the next edge to the edges already selected would **create a cycle**.

We can consider the algorithm's operations as a progression through **a series of forests** containing all the vertices of a given graph and some of its edges. The initial forest consists of |V| trivial trees, each comprising a **single vertex of the graph**. The **final forest** consists of a **single tree,** which is a **minimum spanning tree** of the graph**.** On each iteration, the algorithm takes the next edge (u, v) from the sorted list of the graph's edges, finds the trees containing the vertices u and v, and, if these trees are not the same, unites them in a larger tree by adding the edge (u, v).

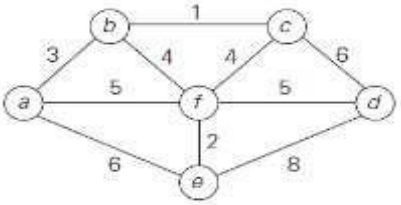
Analysis of Efficiency

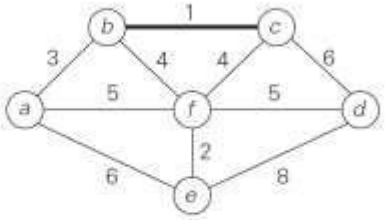
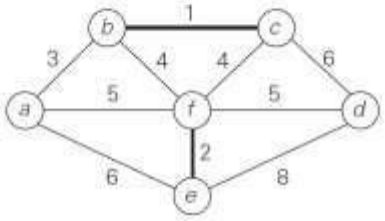
The crucial check whether two vertices belong to the same tree can be found out using **union-find algorithms.**

Efficiency of Kruskal's algorithm is based on the time needed for **sorting the edge weights** of a given graph. Hence, with an efficient sorting algorithm, the time efficiency of Kruskal's algorithm will be in **O (|E| log |E|).**

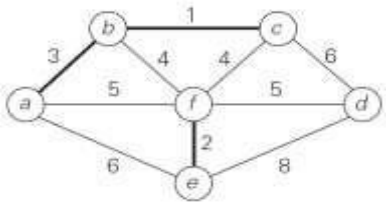
Illustration

An example of Kruskal's algorithm is shown below. The selected edges are shown in bold.

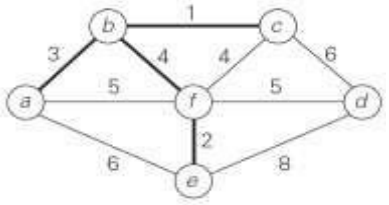


Tree edges	Sorted list of edges	Illustration
	bc ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bc 1	bc **ef** ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	

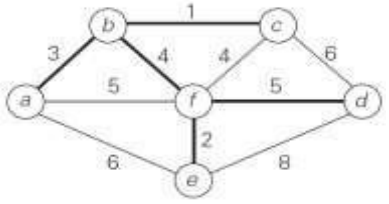
ef	bc	ef	**ab**	bf	cf	af	df	ae	cd	de
2	1	2	3	4	4	5	5	6	6	8



ab	bc	ef	ab	**bf**	cf	af	df	ae	cd	de
3	1	2	3	4	4	5	5	6	6	8



bf	bc	ef	ab	bf	cf	af	**df**	ae	cd	de
4	1	2	3	4	4	5	5	6	6	8



df
5

3. Single source shortest paths

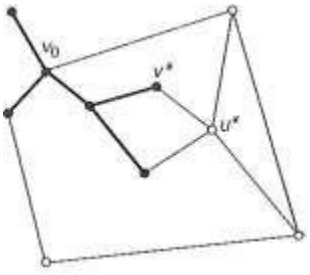
Single-source shortest-paths problem is defined as follows. For a given vertex called the *source* in a weighted connected graph, the problem is to find shortest paths to all its other vertices. The single-source shortest-paths problem asks for a family of paths, each leading from the source to a different vertex in the graph, though some paths may, of course, have edges in common.

3.1. Dijkstra's Algorithm

Dijkstra's Algorithm is the best-known algorithm for the single-source shortest-paths problem. This algorithm is applicable to undirected and directed graphs with nonnegative weights only.

Working - Dijkstra's algorithm finds the shortest paths to a graph's vertices in order of their distance from a given source.

- First, it finds the shortest path from the source to a vertex nearest to it, then to a second nearest, and so on.
- In general, before its i^{th} iteration commences, the algorithm has already identified the shortest paths to i-1 other vertices nearest to the source. These vertices, the source, and the edges of the shortest paths leading to them from the source form a subtree T_i of the given graph shown in the figure.



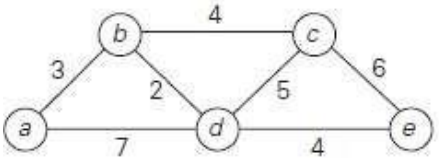
- Since all the edge weights are nonnegative, the next vertex nearest to the source can be found among the vertices adjacent to the vertices of T_i. The

set of vertices adjacent to the vertices in *Ti* can be referred to as "fringe vertices"; they are the candidates from which Dijkstra's algorithm selects the next vertex nearest to the source.

- To identify the i^{th} nearest vertex, the algorithm computes, for every fringe vertex *u*, the sum of the distance to the nearest tree vertex *v* (given by the weight of the edge (*v*, u)) and the length *d.,* of the shortest path from the source to *v* (previously determined by the algorithm) and then selects the vertex with the smallest such sum. The fact that it suffices to compare the lengths of such special paths is the central insight of Dijkstra's algorithm.
- To facilitate the algorithm's operations, we label each vertex with two labels.
  - The numeric label **d** indicates the length of the shortest path from the source to this vertex found by the algorithm so far; when a vertex is added to the tree, d indicates the length of the shortest path from the source to that vertex.
  - The other label indicates the name of the next-to-last vertex on such a path, i.e., the parent of the vertex in the tree being constructed. (It can be left unspecified for the sources and vertices that are adjacent to none of the current tree vertices.)

  With such labeling, finding the next nearest vertex u* becomes a simple task of finding a fringe vertex with the smallest d value. Ties can be broken arbitrarily.
- After we have identified a vertex u* to be added to the tree, we need to perform two operations:
  - Move *u** from the fringe to the set of tree vertices.
  - For each remaining fringe vertex *u* that is connected to *u** by an edge of weight *w(u*, u)* such that $d_{u^*} + w(u^*, u) < d_u$, update the labels of *u* by *u** and $d_{u^*} + w(u^*, u)$, respectively.

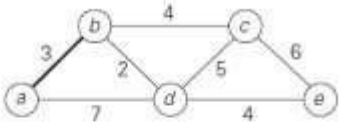
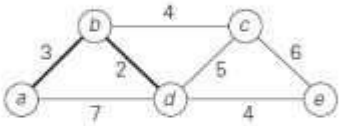
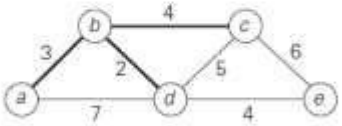
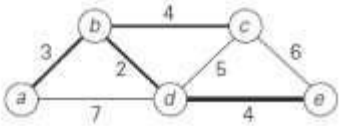
Illustration: An example of Dijkstra's algorithm is shown below. The next closest vertex is shown in bold. (see the figure in next page)



The shortest paths (identified by following nonnumeric labels backward from a destination vertex in the left column to the source) and their lengths (given by numeric labels of the tree vertices) are as follows:

from *a* to *b* :	$a - b$	of length 3
from *a* to *d* :	$a - b - d$	of length 5
from *a* to *c* :	$a - b - c$	of length 7
from *a* to *e* :	$a - b - d - e$	of length 9

The pseudocode of Dijkstra's algorithm is given below. Note that in the following pseudocode, V_T contains a given source vertex and the fringe contains the vertices adjacent to it *after* iteration 0 is completed.

Tree vertices	Remaining vertices	Illustration
a(−, 0)	**b(a, 3)** c(−, ∞) d(a, 7) e(−, ∞)	
b(a, 3)	c(b, 3 + 4) **d(b, 3 + 2)** e(−, ∞)	
d(b, 5)	**c(b, 7)** e(d, 5 + 4)	
c(b, 7)	**e(d, 9)**	
e(d, 9)		

ALGORITHM *Dijkstra(G, s)*

```
//Dijkstra's algorithm for single-source shortest paths
//Input: A weighted connected graph G = ⟨V, E⟩ with nonnegative weights
//       and its vertex s
//Output: The length d_v of a shortest path from s to v
//        and its penultimate vertex p_v for every vertex v in V
Initialize(Q)  //initialize priority queue to empty
for every vertex v in V
    d_v ← ∞;  p_v ← null
    Insert(Q, v, d_v)  //initialize vertex priority in the priority queue
d_s ← 0;  Decrease(Q, s, d_s)  //update priority of s with d_s
V_T ← ∅
for i ← 0 to |V| − 1 do
    u* ← DeleteMin(Q)  //delete the minimum priority element
    V_T ← V_T ∪ {u*}
    for every vertex u in V − V_T that is adjacent to u* do
        if d_u* + w(u*, u) < d_u
            d_u ← d_u* + w(u*, u);  p_u ← u*
            Decrease(Q, u, d_u)
```

Analysis:

The time efficiency of Dijkstra's algorithm depends on the data structures used for implementing the priority queue and for representing an input graph itself.

Efficiency is **Θ(|V|²)** for graphs represented by their ***weight matrix*** and the priority queue implemented as an ***unordered array***.

For graphs represented by their ***adjacency lists*** and the priority queue implemented as a ***min-heap***, it is in **O (|E| log |V|)**

Applications

- Transportation planning and packet routing in communication networks, including the Internet
- Finding shortest paths in social networks, speech recognition, document formatting, robotics, compilers, and airline crew scheduling.

4. Optimal Tree problem

Background:

Suppose we have to encode a text that comprises characters from some n-character alphabet by assigning to each of the text's characters some sequence of bits called the ***codeword.*** There are two types of encoding: Fixed-length encoding, Variable-length encoding

Fixed-length encoding: This method assigns to each character a bit string of the same length *m* (m >= $\log_2 n$). This is exactly what the standard ASCII code does.

One way of getting a coding scheme that yields a shorter bit string on the average is based on the old idea of assigning shorter code-words to more frequent characters and longer code-words to less frequent characters.

Variable-length encoding: This method assigns code-words of different lengths to different characters, introduces a problem that fixed-length encoding does not have. Namely, how can we tell how many bits of an encoded text represent the first character? (or, more generally, the i^{th}) To avoid this complication, we can limit ourselves to *prefix-free* (or simply *prefix) codes*. In a prefix ode, no code word is a prefix of a codeword of another character. Hence, with such an encoding, **we can simply scan a bit string until we get the first group of bits that is a codeword for some character**, replace these bits by this character, and repeat this operation until the bit string's end is reached.

If we want to create a binary prefix code for some alphabet, it is natural to associate the alphabet's characters with leaves of a binary tree in which all the left edges are labelled by 0 and all the right edges are labelled by 1 (or vice versa). The codeword of a character can then be obtained by recording the labels on the simple path from the root to the character's leaf. Since there is no simple path to a leaf that continues to another leaf, no codeword can be a prefix of another codeword; hence, any such tree yields a prefix code.

Among the many trees that can be constructed in this manner for a given alphabet with known frequencies of the character occurrences**, con**struction of such a tree that would assign shorter bit strings to high-frequency characters and longer ones to low-frequency characters can be done by the following greedy algorithm, invented by **David Huffman.**

4.1 Huffman Trees and Codes

Huffman's Algorithm

Step 1*:* Initialize *n* one-node trees and label them with the characters of the alphabet. Record the frequency of each character in its tree's root to indicate the tree's *weight.* (More generally, the weight of a tree will be equal to the sum of the frequencies in the tree's leaves.)

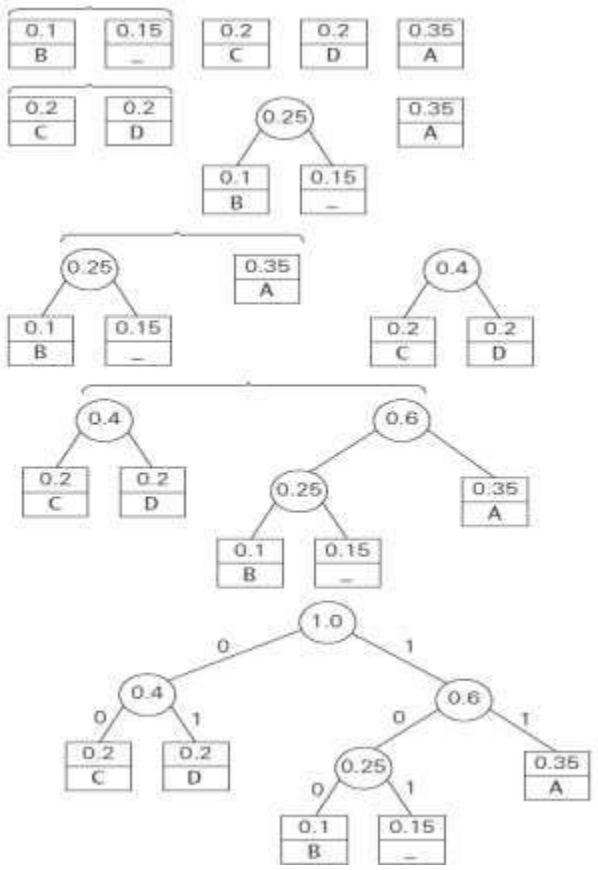
Step 2*:* Repeat the following operation until a single tree is obtained. Find two trees with the smallest weight. Make them the left and right subtree of a new tree and record the sum of their weights in the root of the new tree as its weight.

A tree constructed by the above algorithm is called **a *Huffmantree***. It defines-in the manner described-a ***Huffman code***.

Example: Consider the five-symbol alphabet {A, B, C, D, _} with the following occurrence frequencies in a text made up of these symbols:

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15

The Huffman tree construction for the above problem is shown below:



The resulting codewords are as follows:

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101

Hence, DAD is encoded as 011101, and 10011011011101 is decoded as BAD_AD.

With the occurrence frequencies given and the code word lengths obtained, the **average number of bits per symbol in** this code is

$$2 *0.35 + 3 *0.1+ 2 *0.2 + 2 *0.2 + 3 *0.15 = 2.25.$$

Had we used a fixed-length encoding for the same alphabet, we would have to use at least 3 bits per each symbol. Thus, for this example, Huffman's code achieves the ***compression ratio*** (a standard measure of a compression algorithm's effectiveness) of (3−2.25)/3*100%= 25%. In other words, Huffman's encoding of the above text will use 25% less memory than its fixed-length encoding.

5. Transform and Conquer Approach

We call this general technique transform-and-conquer because these methods work as two-stage procedures. First, in the transformation stage, the problem's instance is modified to be, for one reason or another, more amenable to solution. Then, in the second or conquering stage, it is solved.

There are three major variations of this idea that differ by what we transform a given instance to (Figure 6.1):

- Transformation to a simpler or more convenient instance of the same problem—we call it ***instance simplification***.
- Transformation to a different representation of the same instance—we call it ***representation change***.
- Transformation to an instance of a different problem for which an algorithm is already available—we call it ***problem reduction***.

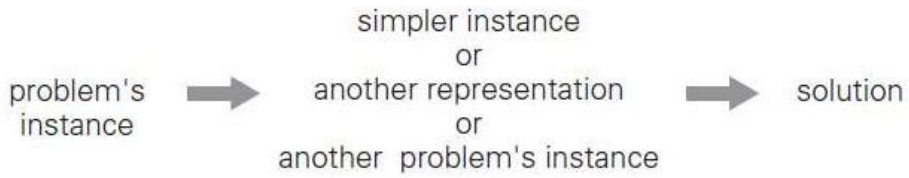


FIGURE 6.1 Transform-and-conquer strategy.

5.1. Heaps

Heap is a partially ordered data structure that is especially suitable for implementing priority queues. **Priority queue** is a multiset of items with an orderable characteristic called an item's **priority**, with the following operations:

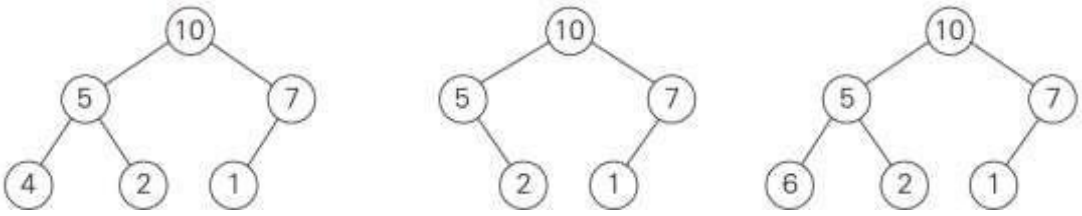
- finding an item with the highest (i.e., largest) priority
- deleting an item with the highest priority
- adding a new item to the multiset

Notion of the Heap

Definition: A ***heap*** can be defined as a binary tree with keys assigned to its nodes, one key per node, provided the following two conditions are met:

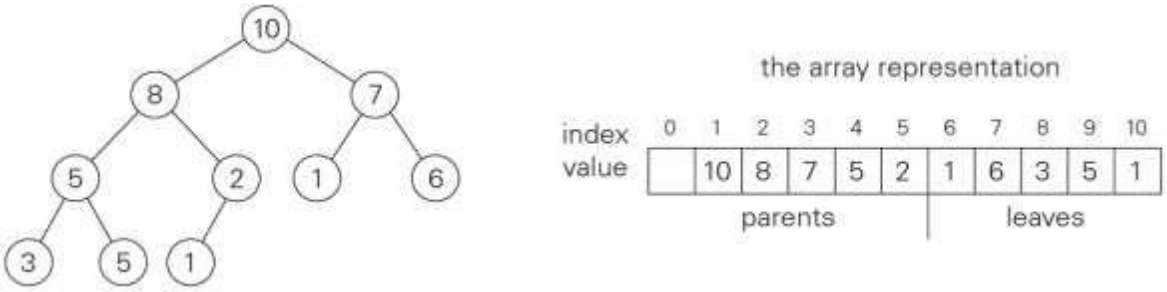
1. The **shape property**—the binary tree is **essentially complete** (or simply **complete**), i.e., all its levels are full except possibly the last level, where only some rightmost leaves may be missing.
2. The **parental dominance** or **heap property**—the key in each node is greater than or equal to the keys in its children.

Illustration: The illustration of the definition of heap is shown bellow: only the left most tree is heap. The second one is not a heap, because the tree's shape property is violated. The left child of last subtree cannot be empty. And the third one is not a heap, because the parental dominance fails for the node with key 5.



Properties of Heap

1. There exists exactly one essentially complete binary tree with *n* nodes. Its height is equal to $\lfloor log_2 n \rfloor$
2. The root of a heap always contains its largest element.
3. A node of a heap considered with all its descendants is also a heap.
4. A heap can be implemented as an **array** by recording its elements in the top down, left-to-right fashion. It is convenient to store the heap's elements in positions 1 through *n* of such an array, leaving *H*[0] either unused or putting there a sentinel whose value is greater than every element in the heap. In such a representation,
    a. the parental node keys will be in the first $\lfloor n/2 \rfloor$. positions of the array, while the leaf keys will occupy the last $\lfloor n/2 \rfloor$ positions;
    b. the children of a key in the array's parental position i ($1 \leq i \leq \lfloor n/2 \rfloor$) will be in positions 2*i* and 2*i* + 1, and, correspondingly, the parent of a key in position *i (*$2 \leq i \leq n$*)* will be in position $\lfloor n/2 \rfloor$.



Heap and its array representation

Thus, we could also define a heap as an array $H[1..n]$ in which every element in position i in the first half of the array is greater than or equal to the elements in positions $2i$ and $2i + 1$, i.e.,

$$H[i] \geq \max \{H[2i], H[2i + 1]\} \text{ for } i = 1 \ldots \lfloor n/2 \rfloor$$

Constructions of Heap - There are two principal alternatives for constructing Heap.
1) Bottom-up heap construction 2) Top-down heap construction

Bottom-up heap construction:

The bottom-up heap construction algorithm is illustrated bellow. It initializes the essentially complete binary tree with n nodes by placing keys in the order given and then "heapifies" the tree as follows.

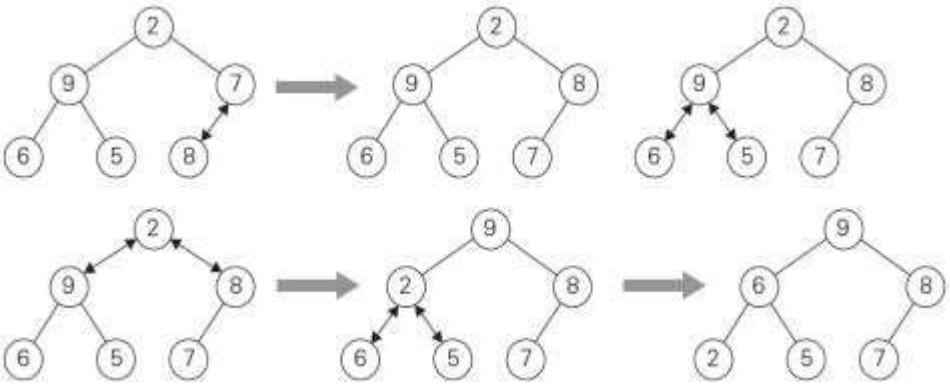
- Starting with the last parental node, the algorithm checks whether the parental dominance holds for the key in this node. If it does not, the algorithm exchanges the node's key K with the larger key of its children and checks whether the parental dominance holds for K in its new position. This process continues until the parental dominance for K is satisfied. (Eventually, it has to because it holds automatically for any key in a leaf.)
- After completing the "heapification" of the subtree rooted at the current parental node, the algorithm proceeds to do the same for the node's immediate predecessor.
- The algorithm stops after this is done for the root of the tree.

ALGORITHM *HeapBottomUp*$(H[1..n])$

```
//Constructs a heap from elements of a given array
// by the bottom-up algorithm
//Input: An array H[1..n] of orderable items
//Output: A heap H[1..n]
for i ← ⌊n/2⌋ downto 1 do
    k ← i;  v ← H[k]
    heap ← false
    while not heap and 2 * k ≤ n do
        j ← 2 * k
        if j < n   //there are two children
            if H[j] < H[j + 1]  j ← j + 1
        if v ≥ H[j]
            heap ← true
        else H[k] ← H[j];   k ← j
    H[k] ← v
```

Illustration

Bottom-up construction of a heap for the list 2, 9, 7, 6, 5, 8. The double headed arrows show key comparisons verifying the parental dominance.



Analysis of efficiency - bottom up heap construction algorithm:

Assume, for simplicity, that $n = 2^k - 1$ so that a heap's tree is full, i.e., the largest possible number of nodes occurs on each level. Let h be the height of the tree.

According to the first property of heaps in the list at the beginning of the section, $h = \lfloor log_2 n \rfloor$ or just $\lfloor log_2(n+1) \rfloor = k - 1$ for the specific values of n we are considering.

Each key on level I of the tree will travel to the leaf level h in the worst case of the heap construction algorithm. Since moving to the next level down requires two comparisons—one

to find the larger child and the other to determine whether the exchange is required—the total number of key comparisons involving a key on level *I* will be 2*(h − i).*

Therefore, the total number of key comparisons in the worst case will be

$$C_{worst}(n) = \sum_{i=0}^{h-1} \sum_{\text{level } i \text{ keys}} 2(h-i) = \sum_{i=0}^{h-1} 2(h-i)2^i = 2(n - \log_2(n+1)),$$

where the validity of the last equality can be proved either by using the closed-form formula for the sum $\sum_{i=1}^{h} i2^i$ or by mathematical induction on *h*.

Thus, with this bottom-up algorithm, a heap of size *n* can be constructed with fewer than **2*n*** comparisons.

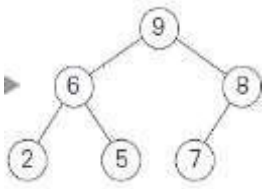
Top-down heap construction algorithm:

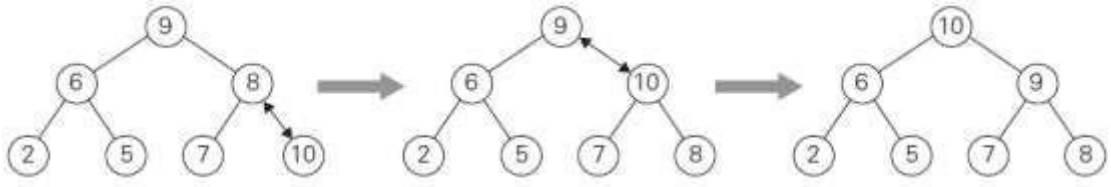
It constructs a heap by successive insertions of a new key into a previously constructed heap.

1. First, attach a new node with key *K* in it after the last leaf of the existing heap.
2. Then shift *K* up to its appropriate place in the new heap as follows.
   a. Compare *K* with its parent's key: if the latter is greater than or equal to *K*, stop (the structure is a heap); otherwise, swap these two keys and compare *K* with its new parent.
   b. This swapping continues until *K* is not greater than its last parent or it reaches root.

Obviously, this insertion operation cannot require more key comparisons than the heap's height. Since the height of a heap with *n* nodes is about $\log_2 n$, the time efficiency of insertion is in *O(*log *n).*

Illustration of inserting a new key: Inserting a new key (10) into the heap is constructed bellow. The new key is shifted up via a swap with its parents until it is not larger than its parents (or is in the root).





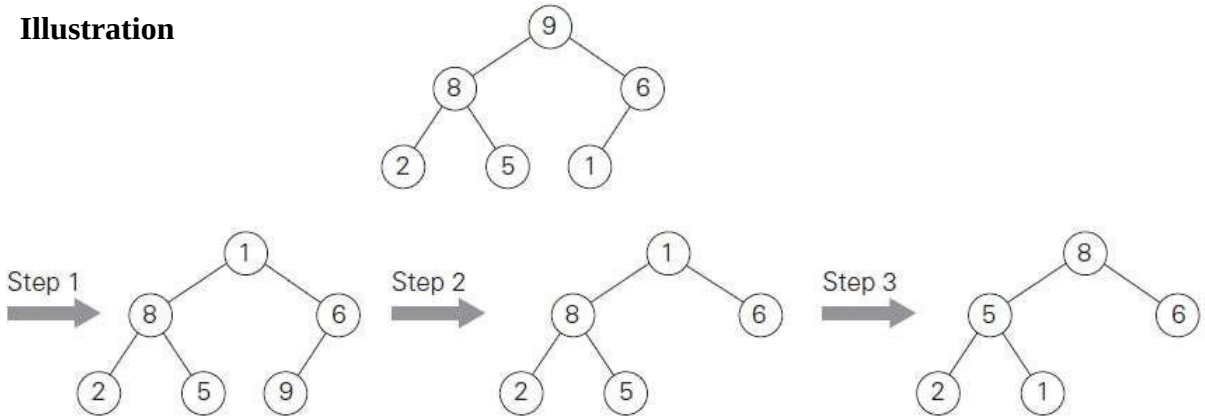
Delete an item from a heap: Deleting the root's key from a heap can be done with the following algorithm:

Maximum Key Deletion from a heap

1. Exchange the root's key with the last key *K* of the heap.
2. Decrease the heap's size by 1.
3. "Heapify" the smaller tree by sifting *K* down the tree exactly in the same way we did it in the bottom-up heap construction algorithm. That is, verify the parental dominance

for *K*: if it holds, we are done; if not, swap *K* with the larger of its children and repeat this operation until the parental dominance condition holds for *K* in its new position.

Illustration



The efficiency of deletion is determined by the number of key comparisons needed to "heapify" the tree after the swap has been made and the size of the tree is decreased by 1.Since this cannot require more key comparisons than twice the heap's height, the time efficiency of deletion is in O (log n) as well.

5.2. Heap Sort

Heapsort - an interesting sorting algorithm is discovered byJ. W. J. Williams. This is a two-stage algorithm that works as follows.

Stage 1 (heap construction): Construct a heap for a given array.

Stage 2 (maximum deletions): Apply the root-deletion operation n−1 times to the remaining heap.

As a result, the array elements are eliminated in decreasing order. But since under the array implementation of heaps an element being deleted is placed last, the resulting array will be exactly the original array sorted in increasing order.

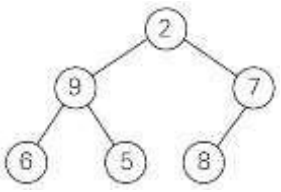
Heap sort is traced on a specific input is shown below:

Stage 1 (heap construction)

2 9 **7** 6 5 8
2 **9** 8 6 5 7
2 9 8 6 5 7
9 **2** 8 6 5 7
9 6 8 2 5 7

Stage 2 (maximum deletions)

9 6 8 2 5 7
7 6 8 2 5 | **9**
8 6 7 2 5
5 6 7 2 | **8**
7 6 5 2
2 6 5 | **7**
6 2 5
5 2 | **6**
5 2
2 | **5**
2



Analysis of efficiency: Since we already know that the heap construction stage of the algorithm is in O(n), we have to investigate just the time efficiency of the second stage. For the number of key comparisons, C(n), needed for eliminating the root keys from the heaps of diminishing sizes from n to 2, we get the following inequality:

$$C(n) \le 2\lfloor \log_2(n-1) \rfloor + 2\lfloor \log_2(n-2) \rfloor + \cdots + 2\lfloor \log_2 1 \rfloor \le 2\sum_{i=1}^{n-1} \log_2 i$$
$$\le 2\sum_{i=1}^{n-1} \log_2(n-1) = 2(n-1)\log_2(n-1) \le 2n \log_2 n.$$

This means that C(n) ∈ O(n log n) for the second stage of heapsort. For both stages, we get **O(n) + O(n log n) = O(n log n).** A more detailed analysis shows that the time efficiency of heapsort is, in fact, in **Θ(n log n)** in both the **worst** and **average** cases. Thus, heapsort's time efficiency falls in the same class as that of mergesort.

Heapsort is **in-place**, i.e., it does not require any extra storage. Timing experiments on random files show that heapsort runs more slowly than quicksort but can be competitive with mergesort.

*****