Module III
GREEDY METHOD

Contents

1. INTRODUCTION TO GREEDY METHOD

1.1 General Method
Greedy method is one of the straightforward design technique which is applicable to wide variety of applications

Most problems have n inputs which requires us to get a solution which is subset of inputs which satisfies certain constraints.

Solution which contains a subset of inputs that satisfies a given constraint is known as Feasible solution.

A feasible solution that either maximizes or minimizes a given objective function is known as optimal solution

The greedy method suggests to devise an algorithm that works in stages (subset paradigm) consider the inputs in an order based on some selection procedure

Use some optimization measure for selection procedure ,at every stage, examine an input to see whether it leads to an optimal solution

If the inclusion of input into partial solution yields an infeasible solution, discard the input; otherwise, add it to the partial solution

Control abstraction for greedy method is given below:

```
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;
}
```

Figure: Control Abstraction for greedy method

Here, **Select** is a function which is used to select an input from the set a[1:n] and removes it.

The selected input"s value is assigned to „x".

Feasible is a boolean valued function which verifies the constraints and determines whether the „x" can be included into the solution vector or not.

Union is a function which is used to add solution „x" to the partially constructed set and updates the objective function.

Feasible Solution:-Any subset of the solutions that satisfies the constraints of the problem is known as a feasible solution.

Optimal Solution:- The feasible solution that maximizes or minimizes the given objective function is an optimal solution. Every problem will have a unique optimal solution.

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:
First finding the number of ways of making changes for a particular amount of cents, n, using a given set of denominations C={$c_1...c_d$} (e.g, the US coin system: {1, 5, 10, 25, 50, 100})

If n = 4,C = {1,2,3}, feasible solutions: {1,1,1,1}, {1,1,2},{2,2},{1,3}.

Second, minimizing the number of coins returned for a particular quantity of change

If available coins are {1, 5, 10, 25}) and n= 30 Cents. Then the solutions obtained are shown below:

Solution 1,2 &3 are feasible solution to the problem where as Solution 4 is the optimal solution

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

Given *n* objects each have a *weight w_i* and a *profit p_i*, and given a knapsack of total *capacity m*.

The problem is to pack/fill the knapsack with these objects in order to maximize the total value or profit of those objects packed without exceeding the knapsack"s capacity.

More formally, let x_i denote the fraction of the object i to be included in the knapsack, 0 x_i 1, for 1 i n. The problem is to find values for the x_i.
If a fraction x_i $(0 \leq x_i \leq 1)$ of the object i is placed in the bag. A profit $p_i * x_i$ is made.

The objective in the problem is to obtain the maximum profit by filling the bag with given objects.

If it is not possible to include an object entirely a fraction of the object can be included and accordingly a fraction of the profit is earned.

The knapsack problem can be stated mathematically as follows.
Maximize Σ p_i x_i $1 \leq i \leq n$ where n is the number of objects. Such that,

Σ w_i $x_i \leq m$ For every object x_i : $0 \leq x_i \leq 1$ and $1 \leq i \leq n$.

Example : Consider 3 objects whose profits and weights are defined as Total No of objects : n=3 and Knapsack capacity : m=20 (P₁, P₂, P₃) = (25, 24, 15) & (W₁, W₂, W₃) = (18, 15, 10) Problem is to determine the optimum strategy for placing the objects in to the knapsack. The problem can be solved by the greedy approach where in the inputs are arranged according to selection process (greedy strategy) and solve the problem in stages. The various greedy strategies for the problem could be as follows.

(1) **Greedy about profit**: - Select an object which has highest profit i.e arrange the objects in descending order of profits.

(2) **Greedy about weight**: - Select an object which has least weight i.e arrange the objects in ascending order of weights.

(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum p_i x_i$
(0, 2/3, 1)	$\frac{2}{3}$x15+10x1= 20	15x1+$\frac{2}{3}$x24 = **31**

(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum p_i x_i$
(1, 2/15, 0)	18x1+$\frac{2}{15}$x15 = 20	25x1+$\frac{2}{15}$x24= **28.2**

(3) **Greedy about profit per unit weight**: -
✓ Obtain the ratio p_i /w_i of all objects

✓ Select an object which has highest profit/weight ratio i.e arrange the objects in descending order of the ratio profit/weight.

✓ Here $p_1/w_1=25/18=1.38$, $p_2/w_2=24/15=1.6$, $p_3/w_3=15/10=1.5$

(x_1, x_2, x_3)	$\sum w_i\, x_i$	$\sum p_i\, x_i$
$(0, 1, \frac{1}{2})$	$1\times15+\dfrac{1}{2}\times10 = 20$	$1\times24+\dfrac{1}{2}\times15 = \mathbf{31.5}$

(4) If an additional constraint of including each and every object is placed then the greedy strategy could be:

(x_1, x_2, x_3)	$\Sigma\ w_i\ x_i$	$\Sigma\ p_i\ x_i$
$(\frac{1}{2}, \frac{1}{3}, \frac{1}{4})$	X1/2 18+ x1/3 15+ x ¼ 10 = 16. 5	x25+ x24+ x15= 24.25

Greedy Strategy 3 always yields an optimal solution to the knapsack problem

Algorithm for greedy knapsack is given below
Algorithm Greedy knapsack (m, 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 i: = 1 to n do x[i] = 0.0
// initialize x U : = m; for i: = 1 to n do
{
 if (w[i] > u) then break; x[i]: = 1.0; U: = U-w[i];
}
if (i ≤ n) then x[i]:
= U/w[i];
}
Analysis: -
If we do not consider the time considered for sorting the inputs then all of the three greedy strategies complexity will be O(n).

If we consider the time considered for sorting the inputs then all of the three greedy strategies complexity will be O(n log n).

1.4 Job Sequencing with deadlines
Problem:-
We are given a set of „n" jobs.

Associated with each job there is an integer dead line d_i 0 and a profit p_i >0.

For any job i the profit p_i is earned if and only if the job is completed by its dead line.

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 the jobs.

A **feasible solution** for the problem will be 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".

An **optimal solution** is a feasible solution with maximum value of the profit.

The problem involves identification of a subset of jobs which can be completed by its

deadline. Therefore the problem suites the subset methodology and can be solved by the greedy method.

Example : - Obtain the optimal sequence for the following jobs where n=4 j_1 j_2 j_3 j_4 **Profit : (P_1, P_2, P_3, P_4)** = (100, 10, 15, 27) and **Deadlines : (d_1, d_2, d_3, d_4)** = (2, 1, 2, 1)

The following table provides the solution for the above problem:

Feasible soln	Processing sequence	Value
(1,2)	(2,1)	100+10=110
(1,4)	**(4,1)**	**100+27=127**
(2,3)	(2,3)	10+15=25
(3,4)	(4,3)	15+27=42
(1)	(1)	100
(2)	(2)	10
(3)	(3)	15
(4)	(4)	27
(1,4)	**(4,1)**	**100+27=127**

In the example solution „3" is the optimal.

In this solution only jobs 1&4 are processed and the value is 127. These jobs must be processed in the order j_4 followed by j_1.

The job4 processing begins at time 0 and ends at time 1. And the processing of job 1 begins at time 1 and ends at time 2. Therefore both the jobs are completed within their deadlines.

The optimization measure for determining the next job to be selected in to the solution is according to the profit. The next job to include is that which increases Σp_i the most, subject to the constraint that the resulting "J" is the feasible solution.

Therefore the greedy strategy for obtaining optimal solution is to consider the jobs in decreasing order of profits.

If job i has not been assigned a processing time, then assign it to slot [-1,] where is the integer such that 1 d_i where d_i is the deadline of the i_{th} job and if the slot [-1,] is not free then assign

[-2, -1] and so on .

If there is no slot, the new job is not included.

As there are only n jobs and each job takes one unit of time, it is necessary to consider the time slots [i-1,i] 1 i b where b = min {n, max {d_i}}

The time slots are partitioned into b sets like [0,1][1,2][2,3]…………[b-1,b]
Example: let n = 5, (p_1, p_2, p_3, p_4, p_5) = (20,15,10,5,1) and (d_1, d_2, d_3, d_4, d_5) = (2,2,1,3,3). Using the above rule

| Selected jobs J	Assigned slots	Job considered	Action	Profit
Ø	None	J1	Assign J_1 to slot [1, 2]	0
{ J1}	[1,2]	J2	Assign J_2 to slot [0,1]	20
{J_1,J_2}	[0,1],[1,2]	J3	Cannot assign, reject job J3 *As [0,1] slot is not free*	35
{J_1,J_2}	[0,1],[1,2]	J4	Assign J_4 to slot [2,3]	40
{J_1,J_2,J_4}	[0,1],[1,2],[2,3]	J5	Cannot assign, reject job J5 *As [0,1],[1,2],[2,3] slots are not free*	40

The optimal solution is: Jobs selected= {J_1,J_2,J_4} Processing sequence of jobs={J_2,J_1,J_4} Profit obtained =40

Algorithm for job sequencing with deadlines is given below
Algorithm Greedy Job (d,j,n) //input: n is the number of jobs and d[1:n] is the deadlines of the jobs according to their descending order of profits //output: J is a set of jobs that can be completed by their dead lines { J: = {1}; for i: = 2 to n do { if (all jobs in J U {i} can be completed by their dead lines) then J: = J U { i}; } }
Analysis: Time complexity is O(n)

2. MINIMUM COST SPANNNING TREES DEFINITION A *spanning tree* of an undirected connected graph is a connected acyclic subgraph (i.e., a tree) that contains all the vertices of the graph and the number of edges is equal to one less than the number of vertices of the graph. If such a graph has weights assigned to its edges, a *minimum spanning tree* is a 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. The following figure presents a simple example illustrating these notions.
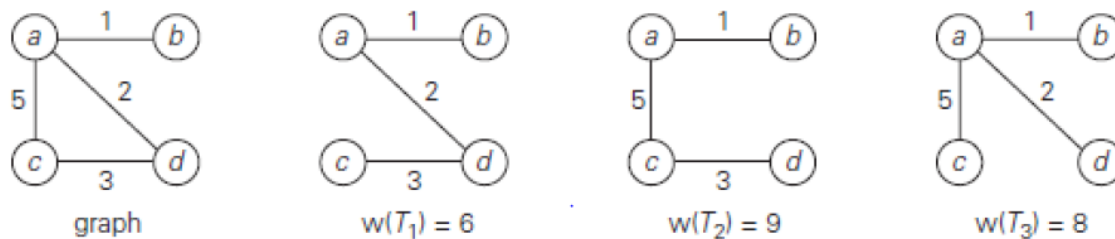
FIGURE : Graph and its spanning trees, with T_1 being the minimum spanning tree.

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, the algorithm expands the current tree in the greedy manner by simply attaching to it the nearest vertex not in that tree. (By the nearest vertex, we mean a vertex not in the tree connected to a vertex in the tree by an edge of the smallest weight. Ties can be broken arbitrarily.)

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 used for the spanning tree

Here is pseudocode of this algorithm.

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

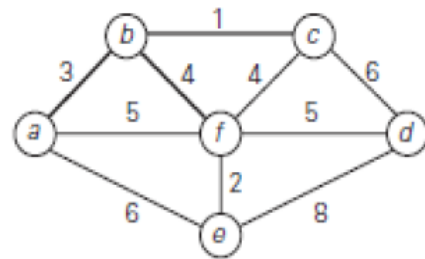**Example:** An example of prim"s algorithm is shown below. The parenthesized labels of a

vertex in the middle column indicate the nearest tree vertex and edge weight; selected vertices and edges are shown in bold.
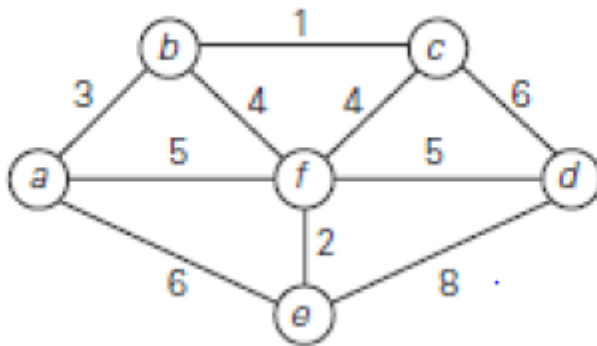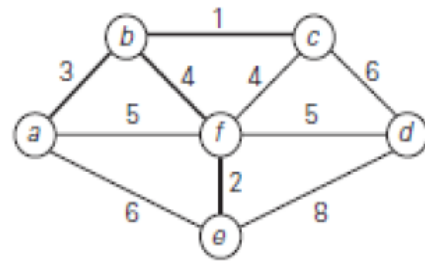

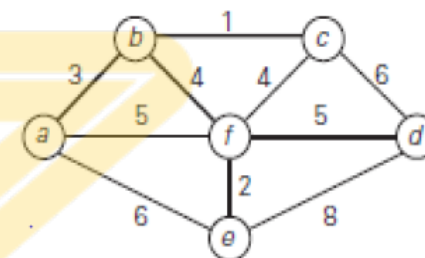| Tree vertices	Remaining vertices	Illustration
a(−, −)	b(a, 3) c(−, ∞) d(−, ∞) e(a, 6) f(a, 5)	
b(a, 3)	c(b, 1) d(−, ∞) e(a, 6) f(b, 4)	

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)

FIGURE: Application of Prim's algorithm.

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.

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 $|V| - 1$ iterations, 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 also 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,and so is the operation of changing an element"s priority

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 minimumspanning tree problem that also always yields an optimal solution. It is named *Kruskal's algorithm* after Joseph Kruskal, who discovered this algorithm when he was a second-year graduate student.

Kruskal"s algorithm looks at a minimum spanning tree of a weighted connected graph G = (V, E) as an acyclic subgraph 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 subgraphs that 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 nondecreasing 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 subgraph if such an inclusion does not create a cycle and simply skipping the edge otherwise.

Here is pseudocode of this algorithm.

ALGORITHM *Kruskal(G)*

//Kruskal'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
sort E in nondecreasing order of the edge weights $w(e_{i_1}) \leq \cdots \leq w(e_{i_{|E|}})$
$E_T \leftarrow \varnothing;$ *ecounter* $\leftarrow 0$ //initialize the set of tree edges and its size
$k \leftarrow 0$ //initialize the number of processed edges
while *ecounter* $< |V| - 1$ **do**
    $k \leftarrow k + 1$
    if $E_T \cup \{e_{i_k}\}$ is acyclic
        $E_T \leftarrow E_T \cup \{e_{i_k}\};$ *ecounter* $\leftarrow$ *ecounter* $+ 1$
return E_T

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.

It is not difficult to see that a new cycle is created if and only if the new edge connects two vertices already connected by a path, i.e., if and only if the two vertices belong to the same connected component (Figure below- (a) leads to cycle whereas (b) may not).

(a)   (b)
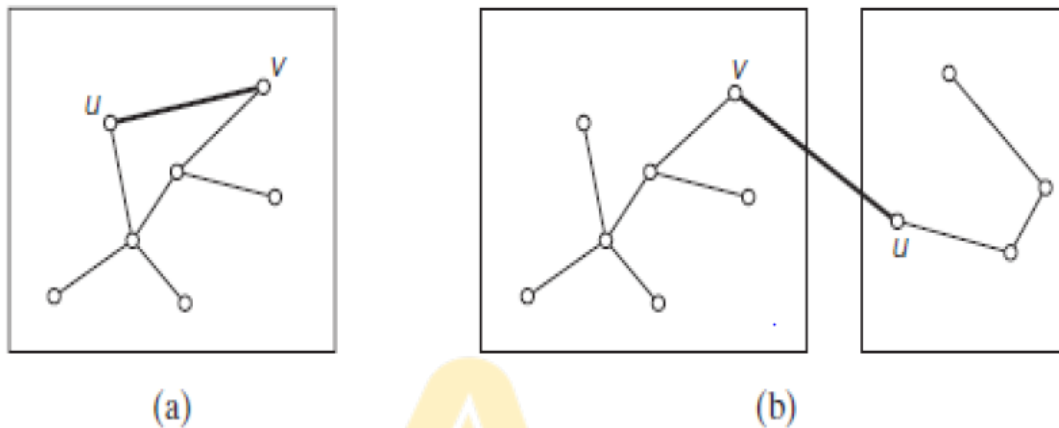
Note also that each connected component of a subgraph generated by Kruskal"s algorithm is a tree because it has no cycles.

In view of these observations, it is convenient to use a slightly different interpretation of Kruskal"s algorithm.

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 for whether two vertices belong to the same tree can be found by using *union-find* algorithms.

With an efficient union-find algorithm, the running time of Kruskal"s algorithm will be dominated by 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 $O(|E| \log |E|)$.

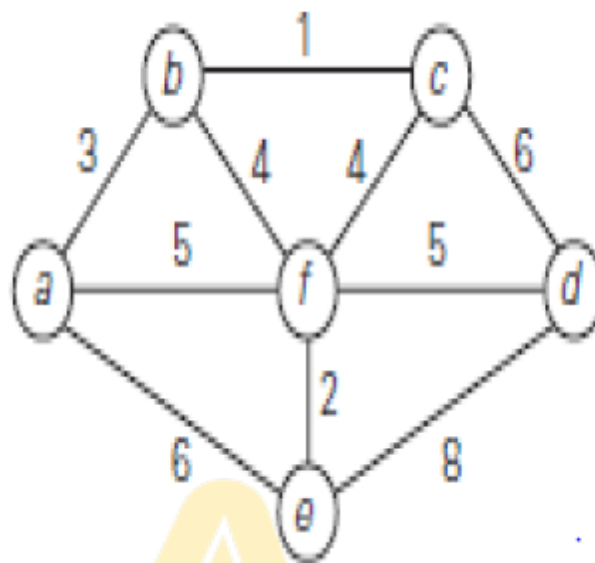**Example:** An example of kruskal"s algorithm is shown below. Selected edges are shown in bold.
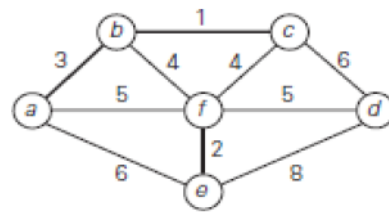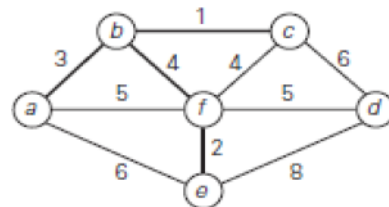
Figure:Graph

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

df
5

Application of krushkal's algorithm

Application of minimum cost spanning trees

It deals with the design of all kinds of networks including communication, computer, transportation, and electrical by providing the cheapest way to achieve connectivity.

It also deals with identifiying clusters of points in data sets.

It has been used for classification purposes in archeology, biology, sociology, and other sciences.

It is also helpful for constructing approximate solutions to more difficult problems such the traveling salesman problem

2. 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. **2.1 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 source s 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:
o Move u* from the fringe to the set of tree vertices.

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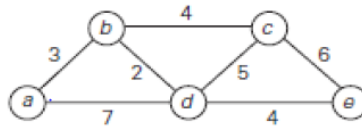
Tree vertices	Remaining vertices	Illustration
a(−, 0)	b(a, 3) c(−, ∞) d(a, 7) e(−, ∞)	
b(a, 3)	c(b, 3 + 4) **d(b, 3 + 2)** e(−, ∞)	
d(b, 5)	**c(b, 7)** e(d, 5 + 4)	
c(b, 7)	**e(d, 9)**	
e(d, 9)		

Figure : Application of Dijkstra's algorithm.

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.

ALGORITHM *Dijkstra(G, s)*

    //Dijkstra's algorithm for single-source shortest paths
    //Input: A weighted connected graph $G = \langle V, E \rangle$ with nonnegative weigh
    //   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
        $d_v \leftarrow \infty$; $p_v \leftarrow$ **null**
        Insert(Q, v, d_v)   //initialize vertex priority in the priority queue
    $d_s \leftarrow 0$; *Decrease(Q, s, d_s)* //update priority of s with d_s
    $V_T \leftarrow \varnothing$
    for $i \leftarrow 0$ **to** $|V| - 1$ **do**
        $u^* \leftarrow DeleteMin(Q)$ //delete the minimum priority element
        $V_T \leftarrow V_T \cup \{u^*\}$
        for every vertex u in $V - V_T$ that is adjacent to u^* **do**

$$\textbf{if } d_{u^*} + w(u^*, u) < d_u$$
$$d_u \leftarrow d_{u^*} + w(u^*, u); \quad p_u \leftarrow u^*$$
$$Decrease(Q, u, d_u)$$

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. It is in $\theta(|V|^2)$ 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
Suppose we have to encode a text that comprises symbols from some *n*-symbol alphabet by assigning to each of the text"s symbols some sequence of bits called the ***codeword***. There are two ways of encoding: fixed-length encoding and variable-length encoding.
Fixed-length encoding:

This method assigns to each symbol a bit string of the same length m ($m \geq \log2 n$). 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 codewords to more frequent symbols and longer codewords to less frequent symbols.

Variable-length encoding:
This method assigns codewords of different lengths to different symbols, 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 (or, more generally, the ith) symbol?

To avoid this complication, we can limit ourselves to the so-called *prefix-free* (or simply *prefix*) *codes*.

In a prefix code, no codeword is a prefix of a codeword of another symbol. 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 symbol, replace these bits by this symbol, 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 symbols with leaves of a binary tree in which all the left edges are labeled by 0 and all the right edges are labeled by 1. The codeword of a symbol can then be obtained by recording the labels on the simple path from the root to the symbol"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 symbol occurrences, how can we construct a tree that would assign shorter bit strings to high-frequency symbols and longer ones to low-frequency symbols? It 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 symbols of the alphabet given. Record the frequency of each symbol 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 **Huffman tree**. It defines—in the manner described above—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

The Huffman

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

The Huffman tree construction for this input is shown below:
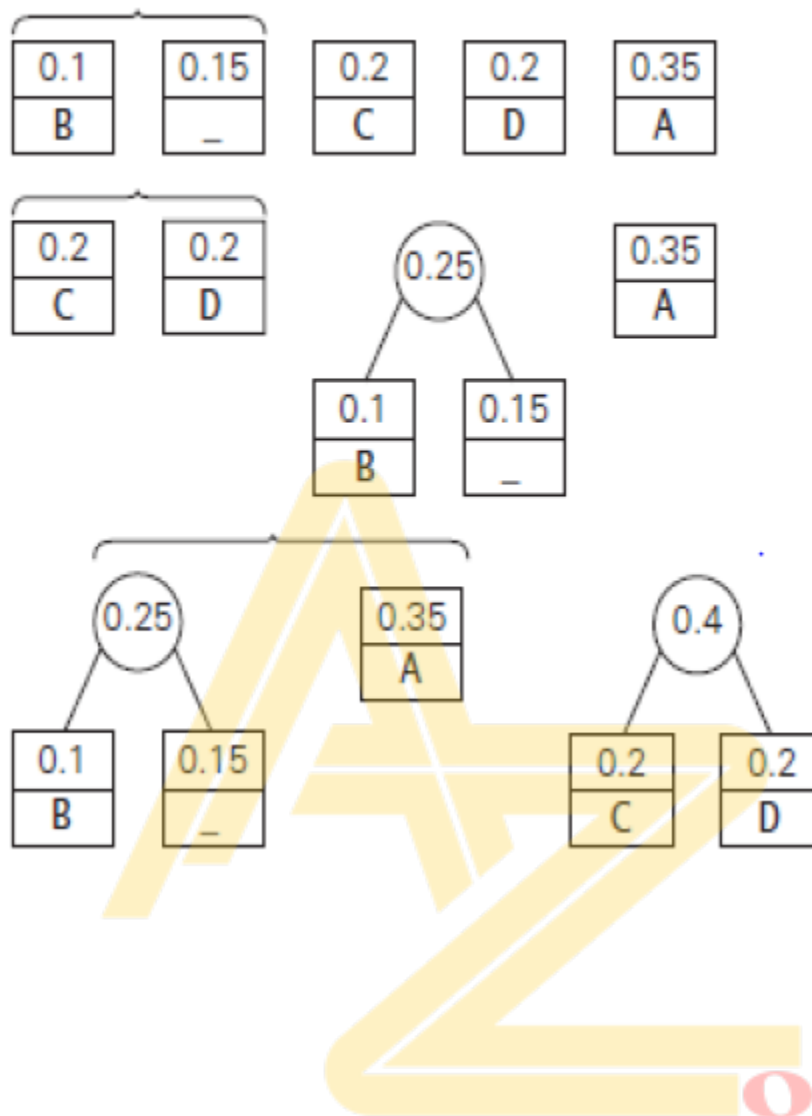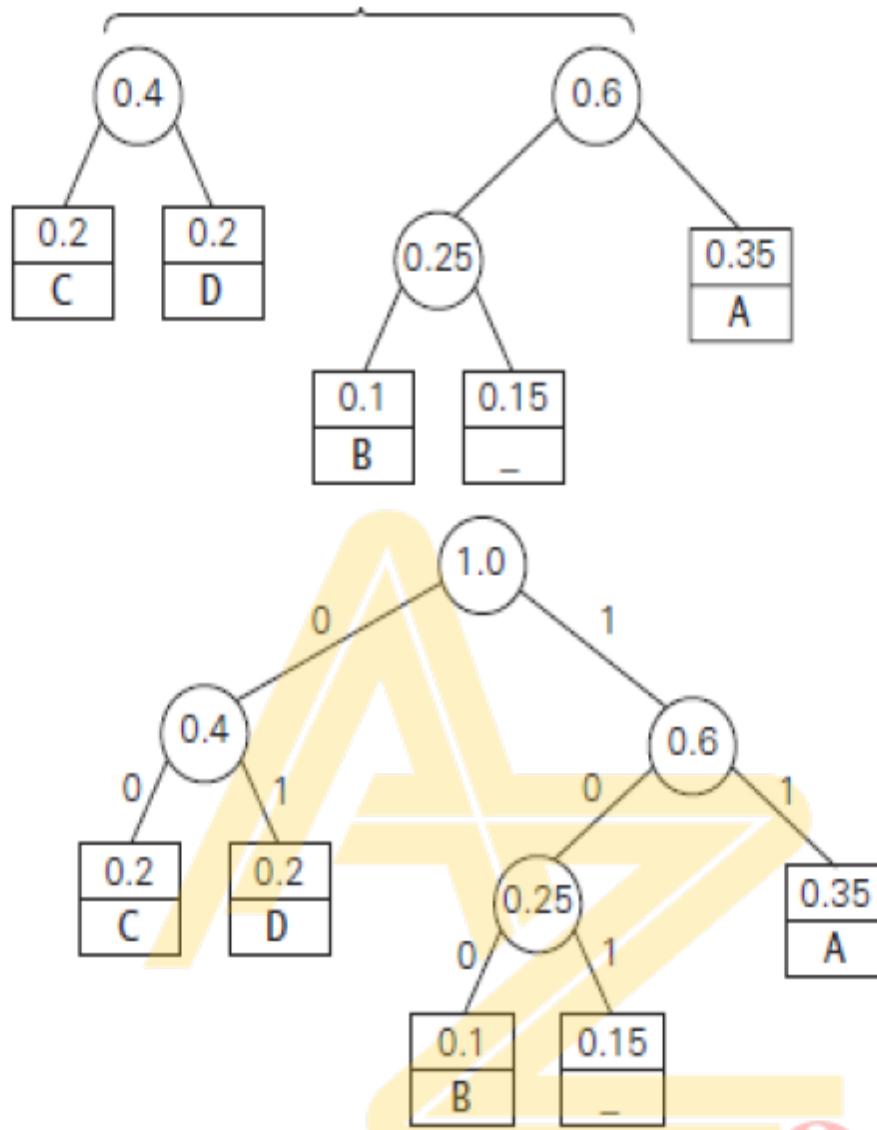
FIGURE:Example of constructing a Huffman coding tree. The resulting codewords are as follows:

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 codeword 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.
If we had 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 text will use 25% less memory than its fixed-length encodin

5. TRANSFORM AND CONQUER APPROACH

Transform-and-conquer approach 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 below):

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**.
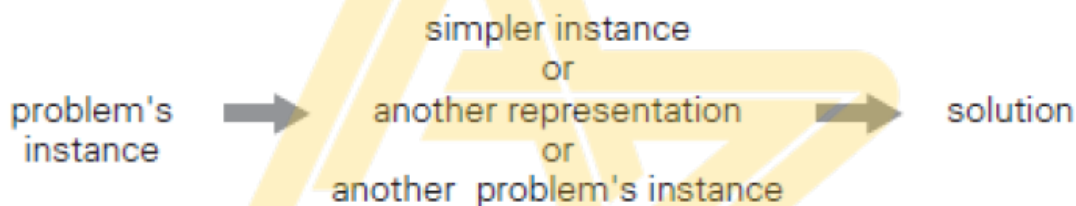
Figure: Transform and Conquer strategy

5.1 Heaps
A heap is a partially ordered data structure that is especially suitable for implementing priority queues.

A *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. (This condition is considered automatically satisfied for all leaves.)

Illustration:

The illustration of the definition of heap is shown below: 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 subtree where node 5 is the root cannot be empty. And the third one is not a heap, because the parental dominance fails for the node with key 5.
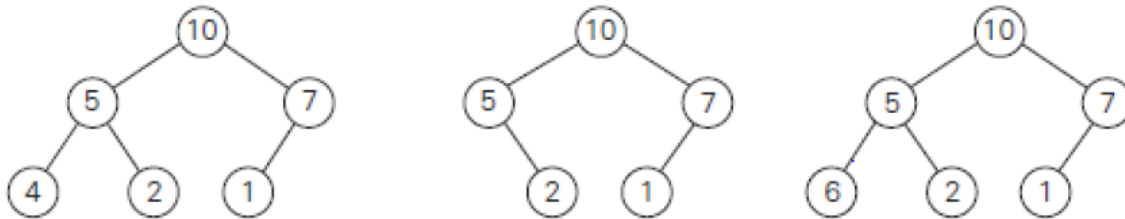

Figure: Illustration of the definition of heap: only the leftmost tree is a heap.

Properties of Heap

Here is a list of important properties of heaps:
1. There exists exactly one essentially complete binary tree with n nodes. Its height is equal to 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 topdown, 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 n/2 positions of the array, while the leaf keys will occupy the last n/2 positions;
b. the children of a key in the array"s parental position i $(1 \le i \le i/2)$ will be in positions $2i$ and $2i + 1$, and, correspondingly, the parent of a key in position i $(2 \le i \le n)$ will be in position i/2.
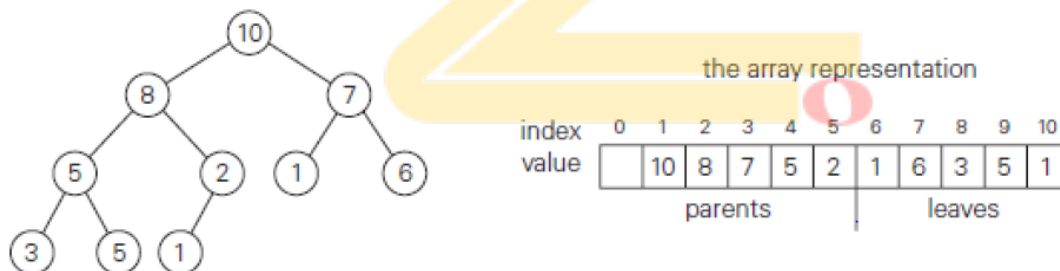
Figure: Heap and its array representation.

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., $H[i] \ge \max\{H[2i], H[2i + 1]\}$ for $i = 1, . . . ,$ **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.

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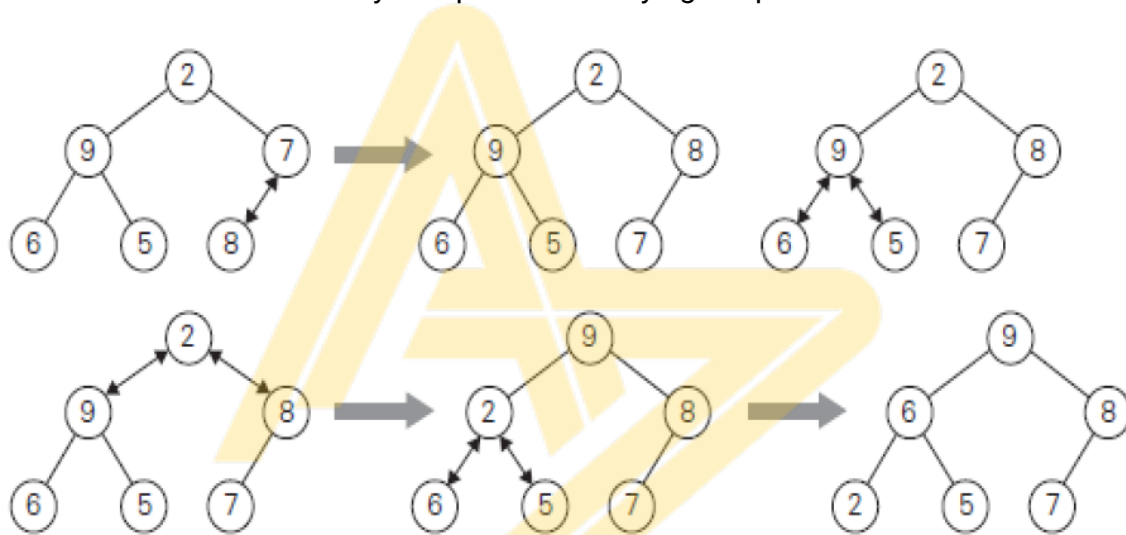
Figure: Bottom-up construction of a heap for the list 2, 9, 7, 6, 5, 8.

Algorithm for the same is given below

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 $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1 **do**
    $k \leftarrow i; \quad v \leftarrow H[k]$
    heap $\leftarrow$ **false**
    while not *heap* **and** $2 * k \leq n$ **do**
      $j \leftarrow 2 * k$
      if $j < n$ //there are two children
        if $H[j] < H[j+1] \; j \leftarrow j + 1$
      if $v \geq H[j]$
        heap $\leftarrow$ **true**
      else $H[k] \leftarrow H[j]; \quad k \leftarrow j$
    $H[k] \leftarrow v$

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 $\lceil \log_2 (n+1) \rceil - 1 = k - 1$ 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 **2n** comparisons.
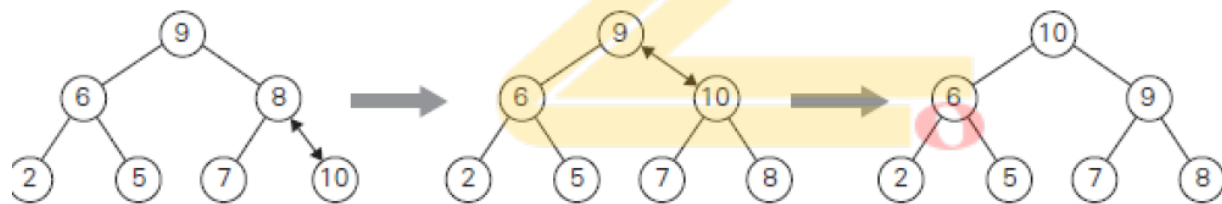
Top-down heap construction algorithm
The algorithm constructs a heap by successive insertions of a new key into a previously constructed heap
First, attach a new node with key K in it after the last leaf of the existing heap.

Then shift K up to its appropriate place in the new heap as follows.

✓   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.

✓   This swapping continues until K is not greater than its last parent or it reaches the 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 log₂ n, the time efficiency of insertion is in O(log n). **Illustration** Inserting a key (10) into the heap constructed above. The new key is shifted up via a swap with its parent until it is not larger than its parent (or is in the root).

Figure: Top-down construction of a heap by insertions

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 shifting 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.
Analysis of Efficiency 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. **Illustration** The key(root) 9 to be deleted is swapped with the last key i.e., 1 after which the smaller tree is "heapified" by exchanging the new key in its root with the larger key in its children until the parental dominance requirement is satisfied.
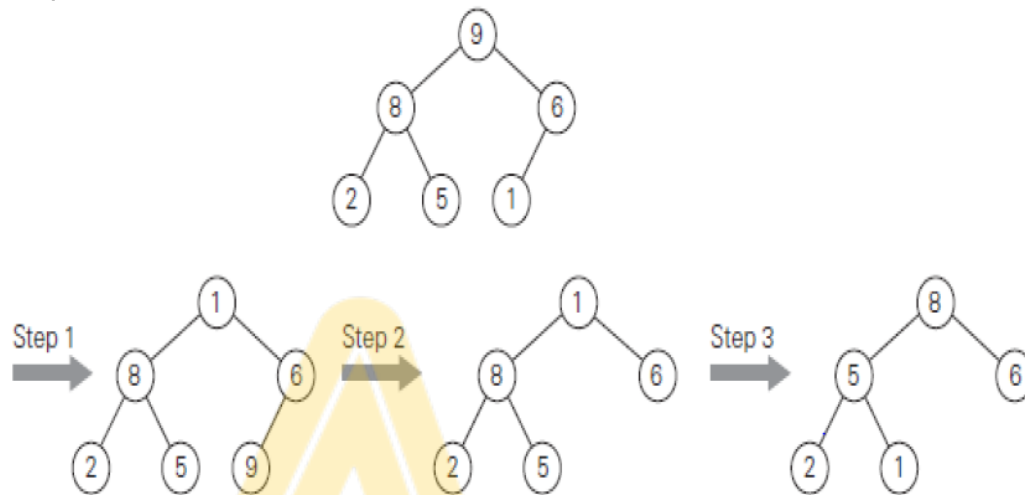

Figure: Deleting the root's key from a heap

5.2 Heap Sort

Heapsort is an interesting sorting algorithm is discovered by J. 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)	Stage 2 (maximum deletions)

```
Stage 1 (heap construction)          Stage 2 (maximum deletions)
   2  9  7  6  5  8                     9   6   8   2   5   7

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

   2  9  8  6  5  7                     8   6   7   2   5

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

   9  6  8  2  5  7                     7   6   5   2

                                        2   6   5 | 7

                                        6   2   5

                                        5   2 | 6

                                        5   2

                                        2 | 5

                                        2
```

Figure : Sorting the array 2, 9, 7, 6, 5, 8 by heapsort.

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. Unlike the latter, 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