easy-solutions®

New Edition SPPU

As per the New Revised Syllabus (2019 course)
of Savitribai Phule Pune University w.e.f. academic year 2020-2021

DATA STRUCTURES AND ALGORITHMS

(Code : 210252)

"QUICK READ SERIES"

Semester IV
Computer Engineering /
Artificial Intelligence & Data Science

Chapterwise Solved University Paper Solution
For End Semester Examination

TechKnowledge™ Publications

easy – solutions

Savitribai Phule Pune University

As per New Credit System Syllabus(Rev. 2019) of Savitribai Phule Pune University with effective from Academic Year 2020-2021

Data Structures and Algorithms

(Code : 210252)

"Quick Read Series"

Semester IV - Computer Engineering / Artificial Intelligence & Data Science

TechKnowledge™ Publications

EPE122A Price ₹ 120/-

Syllabus

In-Sem. Exam

Unit I : Hashing

07 hrs

Hash Table : Concepts-hash table, hash function, basic operations, bucket, collision, probe, synonym, overflow, open hashing, closed hashing, perfect hash function, load density, full table, load factor, rehashing, issues in hashing, hash functions- properties of good hash function, division, multiplication, extraction, mid-square, folding and universal, Collision resolution strategies - open addressing and chaining, Hash table overflow- open addressing and chaining, extendible hashing, closed addressing and separate chaining.

Unit II : Trees

08 hrs

Tree : basic terminology, General tree and its representation, representation using sequential and linked organization, Binary tree- properties, converting tree to binary tree, binary tree traversals(recursive and non-recursive)- inorder, preorder, post order, depth first and breadth first, Operations on binary tree. Huffman Tree (Concept and Use), Binary Search Tree (BST), BST operations, Threaded binary search tree- concepts, threading, insertion and deletion of nodes in in-order threaded binary search tree, in order traversal of in-order threaded binary search tree.

End-Sem. Exam

Unit III : Graphs

07 hrs

Basic Concepts, Storage representation, Adjacency matrix, adjacency list, adjacency multi list, inverse adjacency list. **Traversals :** depth first and breadth first, Minimum spanning Tree, Greedy algorithms for computing minimum spanning tree- Prims and Kruskal Algorithms, Dikjtra's Single source shortest path, All pairs shortest paths- Floyd-Warshall Algorithm Topological ordering.

Unit IV : Search Trees

08 hrs

Symbol Table : Representation of Symbol Tables- Static tree table and Dynamic tree table, Weight balanced tree - Optimal Binary Search Tree (OBST), OBST as an example of Dynamic Programming, Height Balanced Tree- AVL tree. Red-Black Tree, AA tree, K-dimensional tree, Splay Tree.

Unit V : Indexing and Multiway Trees

07 hrs

Indexing and Multiway Trees : Indexing, indexing techniques-primary, secondary, dense, sparse, Multiway search trees, B-Tree- insertion, deletion, B+Tree - insertion, deletion, use of B+ tree in Indexing, Trie Tree.

07 hrs

Unit VI : File Organization

Files : Concept, need, primitive operations. Sequential file organization- concept and primitive operations, Direct Access File- Concepts and Primitive operations, Indexed sequential file organization-concept, types of indices, structure of index sequential file, Linked Organization- multi list files, coral rings, inverted files and cellular partitions.

□□□

Table of Contents

□□□

Savitribai Phule Pune University
Semester IV (Computer Engineering)

Data Structures and Algorithms

Unit III : Graphs

Q.1 Define a graph. **SPPU - May 15, 2 Marks**

Ans.:

Definition of Graph

A graph G is a set of vertices (V) and set of edges (E). The set V is a finite, nonempty set of vertices. The set E is a set of pair of vertices representing edges.

$G = (V, E)$

$V(G)$ = Vertices of graph G

$E(G)$ = Edges of graph G

An example of graph is shown in Fig. 3.1.

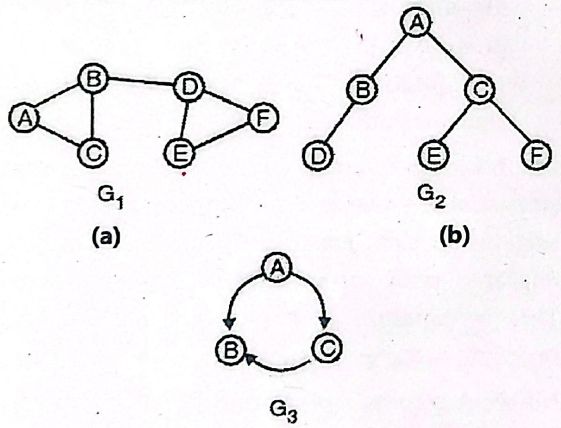


Fig. 3.1 : Graphs

The set representation for each of these graphs is given by

$V(G_1) = \{A, B, C, D, E, F\}$

$V(G_2) = \{A, B, C, D, E, F\}$

$V(G_3) = \{A, B, C\}$

$E(G_1) = \{(A, B),(A, C),(B, C),(B, D),(D, E),(D, F),(E, F)\}$

$E(G_2) = \{(A,B),(A, C),(B, D),(C, E),(C, F)\}$

$E(G_3) = \{(A, B),(A, C),(C, B)\}$

Q.2 Define Undirected and Directed Graph. **(4 Marks)**

Ans.:

Undirected Graph

A graph containing unordered pair of vertices is called an undirected graph. In an undirected graph, pair of vertices (A, B) and (B, A) represent the same edge.

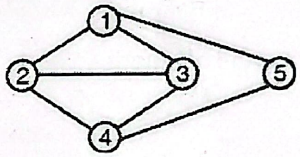


Fig. 3.2(a) : Example of an undirected graph

The set of vertices V = {1, 2, 3, 4, 5}.

The set of edges E = {(1, 2), (1, 3), (1, 5), (2, 3), (2, 4), (3, 4), (4, 5)}.

Directed Graph

A graph containing ordered pair of vertices is called a directed graph. If an edge is represented using a pair of vertices (V_1, V_2) then the edge is said to be directed from V_1 to V_2.

The first element of the pair, V_1 is called the start vertex and the second element of the pair, V_2 is called the end vertex. In a directed graph, the pairs (V_1, V_2) and (V_2, V_1) represent two different edges of a graph. Example of a directed graph is shown in Fig. 3.2(b).

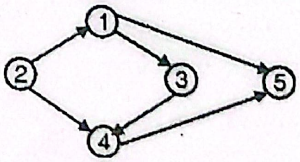


Fig. 3.2(b) : Example of a directed graph

The set of vertices V = {1, 2, 3, 4, 5, 5}.

The set of edges E = {(1,3), (1,5), (2,1), (2,4), (3,4), (4,5)}.

Q.3 With example define the path and cycle w.r.t graphs. **SPPU - May 15, 2 Marks**

Ans.:

Path

A path from vertex V_0 to V_n is a sequence of vertices $V_0, V_1, V_2 \ldots V_{n-1}, V_n$. Here, V_0 is adjacent to V_1, V_1 is adjacent to V_2 and V_{n-1} is adjacent to V_n. The length of a path is the number of edges on the path. A path with n vertices has a length of n – 1. A path is simple if all vertices on the path, except possibly the first and last, are distinct.

Cycle

A cycle is a simple path that begins and ends at the same vertex. Fig. 3.3 is an example of a graph with cycle.

- A B D A is a cycle of length 3
- B D C B is a cycle of length 3
- A B C D A is a cycle of length 4

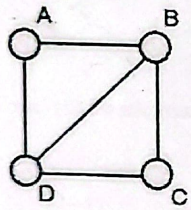


Fig. 3.3 : A graph with cycles

Q.4 With example define the degree of node w.r.t graphs. **SPPU - May 15, 1 Mark**

OR With example define the isolated node w.r.t graphs. **SPPU - May 15, 1 Mark**

Ans.:

Degree of node

The total number of edges linked to a vertex is called its degree. The indegree of a vertex is the total number of edges coming to that node. The outdegree of a node is the total number of edges going out from that node. A vertex, which has only outgoing edges and no incoming edges, is called a source.

A vertex having only incoming edges and no outgoing edges is called a sink. When indegree of a vertex is one and outdegree is zero then such a vertex is called a pendant vertex. When the degree of a vertex is 0, it is an isolated vertex.

Q.5 Explain any three applications of graphs in the area of Computer Engineering. **SPPU - Dec. 13, 3 Marks**

Ans.:

Applications of Graphs

1. The applications of graph split broadly into three categories :
   a) First, analysis to determine structural properties of a network, such as the distribution of vertex degrees and the diameter of the graph. A vast number of graph measures exist.
   b) Second, analysis to find a measurable quantity within the network, for example, for a transportation network, the level of vehicular flow within any portion of it.
   c) Third, analysis of dynamic properties of network. Map of a country can be represented using a graph. Road network, Air network or rail network can be represented using a graph. Connection among routers in a communication network can be represented using a graph. Routing of a packet between two communicating nodes can be done through the shortest path.
2. Graph theory is useful in biology and conservation efforts where a vertex can represent regions where certain species exist and the edges represent migration paths, or movement between the regions. This information is important when looking at breeding patterns or tracking the spread of disease.
3. Different activities of a project can be represented using a graph. This graph can be useful in project scheduling.

Q.6 Define minimal spanning tree with example. **(3 Marks)**

Ans.:

Minimal Spanning Tree

The cost of a graph is the sum of the costs of the edges in the weighted graph. A spanning tree of a graph G = (V, E) is called minimum cost spanning tree or simply minimal spanning tree of G if its cost is minimum.

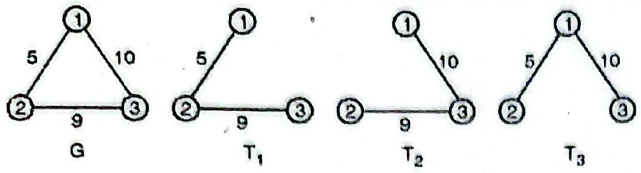


Fig. 3.4

G→ A sample weighted graph.

T_1→ A spanning tree of G with cost 5 + 9 = 14

T_2→ A spanning tree of G with cost 10 + 9 = 19

T_3→ A spanning tree of G with cost 5 + 10 = 15

Therefore, T_1 with cost 14 is the minimal cost spanning tree of the graph G.

A typical application for minimum-cost spanning trees can be seen in the design of communication networks. The vertices of a graph represent cities and the edges possible communication links between the cities. A minimum-cost spanning tree represents a communication network that connects all the cities at minimal cost.

Q.7 Explain with suitable example the various storage structures for the graph.

SPPU - May 14, Dec. 19, 6 Marks

Ans.:

Representation of Graphs

Methods for representation of graph, includes :

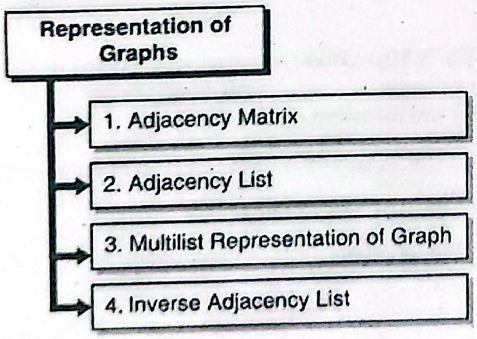


Fig. 3.5 : Representation of graphs

1. Adjacency Matrix

A two dimensional matrix can be used to store a graph. A graph G = (V, E) where V = {0, 1, 2, ... n – 1} can be represented using a two dimensional integer array of size n × n.

int adj[20][20]; can be used to store a graph with 20 vertices.

adj[i][j] = 1, indicates presence of edge between two vertices i and j

= 0, indicates absence of edge between two vertices i and j

A graph is represented using a square matrix.

Adjacency matrix of an undirected graph is always a symmetric matrix, i.e. an edge (i, j) implies the edge (j, i).

Adjacency matrix of a directed graph is never symmetric adj[i][j] = 1, indicates a directed edge from vertex i to vertex j.

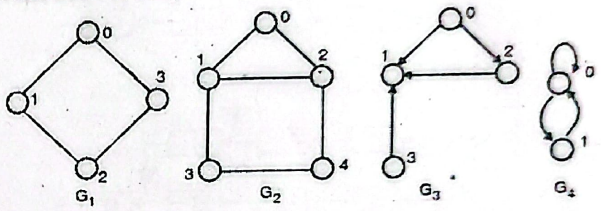


Fig. 3.5(a) : Graphs G_1, G_2, G_3 and G_4

2. Adjacency List

A graph can be represented using a linked list. For each vertex, a list of adjacent vertices is maintained using a linked list. It creates a separate linked list for each vertex V_i in the graph G = (V, E)

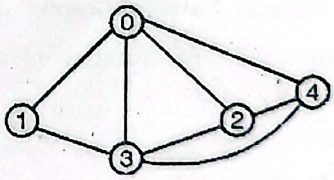
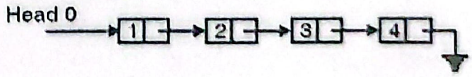
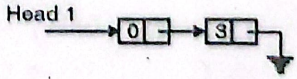


Fig. 3.5(b) : A graph

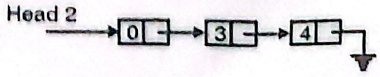
List of adjacent vertices to vertex 0



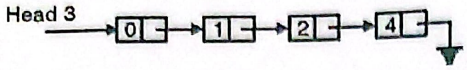
List of adjacent vertices to vertex 1



List of adjacent vertices to vertex 2



List of adjacent vertices to vertex 3



List of adjacent vertices to vertex 4

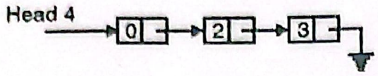


Fig. 3.5(c) : Adjacency list for each vertex of graph

Adjacency list of a graph with n nodes can be represented by an array of pointers. Each pointer points to a linked list of the corresponding vertex. Fig. 3.5(d) shows the adjacency list representation of graph of Fig. 3.5(b).

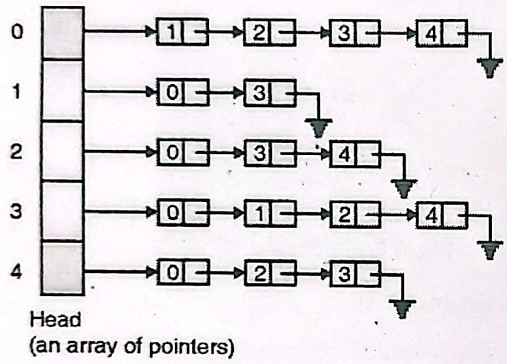


Fig. 3.5(d) : Adjacency list representation of the graph

3. Multilist Representation of Graph

In adjacency list representation of a graph, each edge (i, j) is included twice :

1. Once in the list of vertex "i".
2. Once in the list of vertex "j".

This drawback may be removed if the adjacency lists are maintained as multilists. In a multilist, a node can be shared by several lists.

Each edge must be connected to two lists and the structure of an edge (i, j) looks like the following.

i	j	Next edge in adjacency list for vertex "i"	Next edge in adjacency list for vertex "j"

The adjacency multi-list representation of the sample graph shown in Fig. 3.5(e).

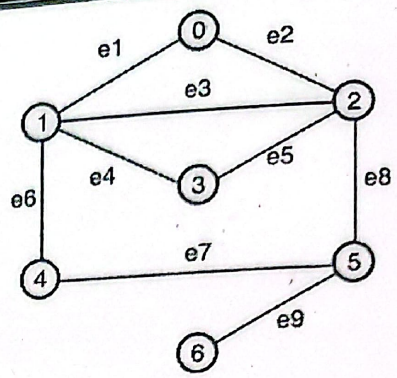


Fig. 3.5(e) : A sample graph for multilist

4. Inverse Adjacency List

Inverse adjacency list can be used to determine the in-degree of a vertex. The inverse adjacency list keeps track of edges coming into a vertex. The inverse adjacency list of the graph shown in Fig. 3.5(f) is shown in Fig. 3.5(g).

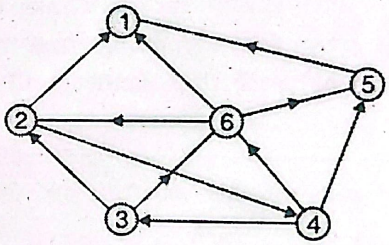


Fig. 3.5(f) : A sample graph for inverse adjacency list

1→ (2, 5, 6)

2 → (3, 6)

3 → (4)

4 → (2)

Fig. 3.5(g) : Inverse adjacency list for graph

Q.8 For the following graph obtain :

(i) The in degree and out degree of each vertex

(ii) Its adjacency matrix

(iii) Its adjacency list representation. **(5 Marks)**

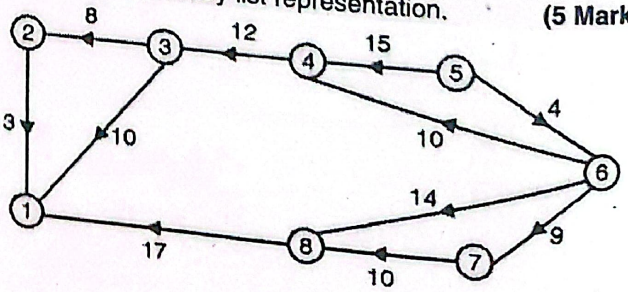


Fig. 3.6

Ans. :

1) Indegree and outdegree of each vertex

Vertex No.	Indegree	Outdegree
1	3	0
2	1	1
3	1	2
4	2	1
5	1	1
6	1	3
7	1	1
8	2	1

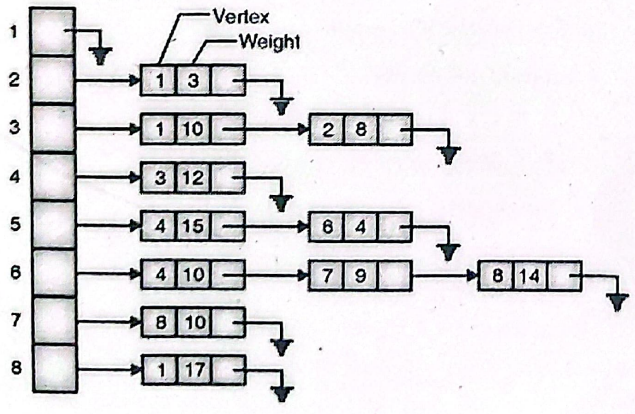
(a)

2) Adjacency matrix

	1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	0	0
2	3	0	0	0	0	0	0	0
3	10	8	0	0	0	0	0	0
4	0	0	12	0	0	0	0	0
5	0	0	0	15	0	4	0	0
6	0	0	0	10	0	0	9	14
7	0	0	0	0	0	0	0	10
8	17	0	0	0	0	0	0	0

(b)

3) Adjacency list



(c)

Fig. 3.6

Q.9 List and explain the techniques of traversal of graphs. **(6 Marks)**

Ans.:

Traversal of Graphs

Most of graph problems involve traversal of a graph. Traversal of a graph means visited each node and visiting exactly once. Two commonly used techniques are :

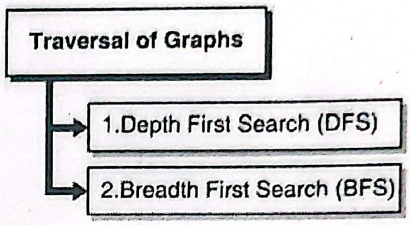


Fig. 3.7 : Traversal of graph

1. Depth First Search (DFS)

It is like preorder traversal of tree. Traversal can start from any vertex, say V_i . V_i is visited and then all vertices adjacent to V_i are traversed recursively using DFS. DFS (G, 1) is given by

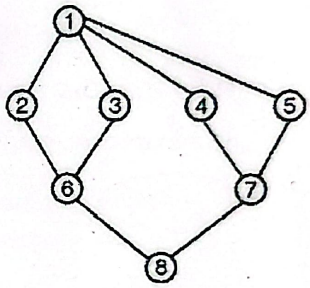


Fig. 3.7(a) : Graph G

(a) Visit (1)

(b) DFS (G, 2)

DFS (G, 3)
DFS (G, 4)
DFS (G, 5)
} all nodes adjacent to 1

Since, a graph can have cycles. Avoid re-visiting a node. To do this, when we visit a vertex V, we mark it visited. A node that has already been marked as visited, should not be selected for traversal. Marking of visited vertices can be done with the help of a global array visited[]. Array visited[] is initialized to false (0).

Algorithm for DFS

n ← number of nodes

1) Initialize visited[] to false (0)
   for (i = 0; i < n; i++)
   visited[i] = 0;

2) void DFS (vertex i) [DFS starting from i]
   {
   visited[i] = 1;

```
        for each w adjacent to i
            if(! visited[w])
                DFS(w);
    }
```

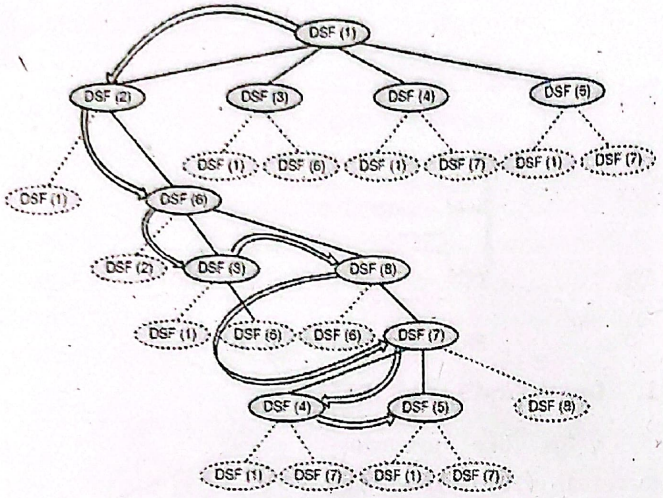


Fig. 3.7(b) : DFS traversal on graph of Fig. 3.7

DFS traversal on graph of Fig. 3.7.

DSF(i) — Node i can be used for recursive traversal using DFS().

DFS(i) — Node i is already visited

2. Breadth First Search (BFS)

- It is another popular approach used for visiting the vertices of a graph. This method starts from a given vertex V_0. V_0 is marked as visited. All vertices adjacent to V_0 are visited next.
- Let the vertices adjacent to V_0 are V_{10}, V_3, V_{12} ... V_{1n}. V_{11}, V_{12} ... V_{1n} are marked as visited. All unvisited vertices adjacent to V_{11}, V_{12} .....V_{1n} are visited next. The method continues until all vertices are visited. The algorithm for BFS has to maintain a list of vertices which have been visited but not explored for adjacent vertices.
- The vertices which have been visited but not explored for adjacent vertices can be stored in queue.
- Initially the queue contains the starting vertex.
- In every iteration, a vertex is removed from the queue and its adjacent vertices which are not visited as yet are added to the queue.
- The algorithm terminates when the queue becomes empty. Fig. 3.7(c) gives the BFS sequence on various graphs.

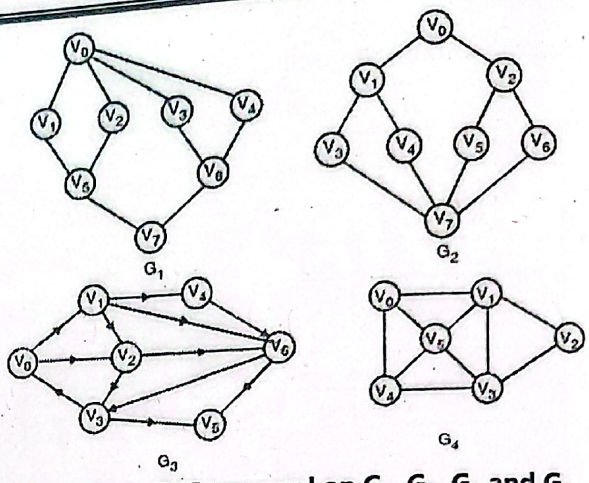


Fig. 3.7(c) : BFS traversal on G_1, G_2, G_3 and G_4

BFS sequence :

$G_1 \rightarrow V_0 \mid V_1\ V_2\ V_3\ V_4 \mid V_5\ V_6 \mid V_7$

$G_2 \rightarrow V_0 \mid V_1\ V_2 \mid V_3\ V_4\ V_5\ V_6 \mid V_7$

$G_3 \rightarrow V_0 \mid V_1\ V_2 \mid V_4\ V_6\ V_3 \mid V_5\ V_6$

$G_4 \rightarrow V_0 \mid V_1\ V_4\ V_5 \mid V_2\ V_3$

Algorithm for BFS

```
/* Array visited[] is initialize to 0 */
/* BFS traversal on the graph G is carried out beginning
at vertex V */
void BFS(int V)
{
    q : a queue type variable;
    initialize q;
    visited[V] = 1;     /* mark V as visited */
    add the vertex V to queue q;
    while(q is not empty)
    {
        V ← delete an element from the queue;
        for all vertices w adjacent from V
        {
            if(!visited[w])
            {
                visited[w] = 1;
                add the vertex w to queue q;
            }
        }
    }
}
```

Q.10 Draw the BFS traversal of the following graph represented using adjacency list.

SPPU - Dec. 14, 4 Marks

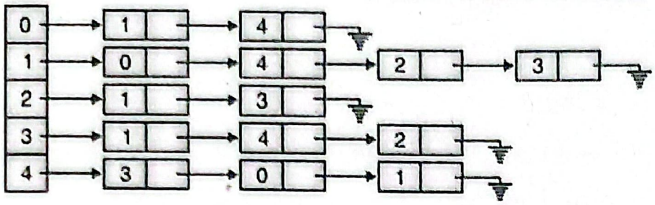
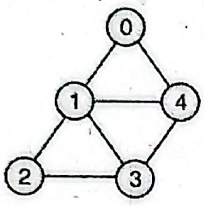


Fig. 3.8

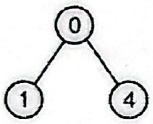
Ans.:

The graph is given by :



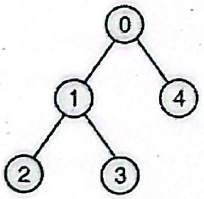
BFS traversal

Step 1 :



0 – 1, 4

Step 2 :



0 – 1, 4 – 2, 3

∴ BFS sequence = 0, 1, 4, 2, 3

Q. 11 Define Greedy method. **(2 Marks)**

Ans. :

The greedy method is a very simple technique and it can be applied to a wide variety of problems. Greedy algorithms work in phases in each phase, a decision is made that appears to be good, without regard for future consequences.

- At each stage, select an input.
- Input selected is added to the set of optimal solution.
- Selection is made on the basis of some selection procedure.
- Inclusion of next input into the partially constructed optimal solution should result in a feasible solution.

Q.12 Write a short note on : Knapsack Problem. **(5 Marks)**

Ans.:

Knapsack Problem

Given n objects and a knapsack(bag).

- Each object is of fixed weight.
- Knapsack has a capacity m (total weight of objects used for filing it)
- There are different types of objects.
- Each type of object gives us a certain profit.

Objective

Fill the knapsack with suitable objects to maximize the profit.

Constraint

Total weight of objects ≤ m

Formally, the problem can be stated as :

- Object i has a weight w_i and it gives a profit of P_i.
- A fraction $x_i (1 \geq x_i \geq 0)$ of object i is used to fill the knapsack.

Thus,

Objective is to maximize $\sum_{i=1}^{n} P_i x_i$

With the constraint $\sum_{i=1}^{n} w_i x_i \leq m$

Greedy approach

1. Select the object that gives, maximum profit per unit weight.

   i.e. Select the object with largest value of P_i/w_i

   Use the object for filling up of knapsack. Now one of the two things can happen.

   Knapsack is filled to its complete capacity.

   Object i is exhausted and some more objects can be added to knapsack.

Example

Consider the following example of knapsack

There are three types of objects, n=3

Capacity of knapsack m = 20

Profit due to objects = (25, 24, 15)

(Profit due to object 1=25

Profit due to Object 2=24

Profit due to Object 3=15)

Weights of objects = (18, 15, 10)

Calculating profit per unit weight :

$i = 1, \quad \frac{25}{18}$

$i = 2, \quad \frac{24}{15}$

$i = 3, \quad \frac{15}{10}$

Here $\frac{24}{15} > \frac{15}{10} > \frac{25}{18}$

Step 1 : Select the object 2 for filling the knapsack.

$X_2 = 1$(All objects of type 2 will be used for filling the knapsack.)

Capacity remaining after filling of object of type 2

$= 20 - 15 = 5.$

Step 2 : Select 1/2 of objects of type 3 to fill the knapsack.

$\therefore X_3 = 1/2$

Step 3 : Since, the knapsack is filled to its complete capacity :

$X_1 = 0$

$\therefore$ Total profit $= 0 \times 25 + 1 \times 24 + 1/2 \times 15$

$= 24 + 7.5 = 31.5$

Connected Components

Traversal algorithm DFS or BFS can be used for finding out the connected components of a graph. All the connected components of a graph can be obtained by making repeated calls to DFS() or BFS() with V a vertex not yet visited.

'C' Function for Printing Connected Components of a Graph

```
#define MAX[20]
int G[MAX][MAX];
int visited[MAX];
int n;
int component()
{
    int count,i;
    count=0;
    for(i=0;i<n;i++)
        visited[i]=0;
    for(i=0;i<n;i++)
        if(visited[i]==0)
        {
            BFS(i);
            count++;
        }
        return(count);
}
```

Q.13 List and explain popular techniques for constructing a minimum cost spanning tree. **(8 Marks)**

Ans.:

Minimum Cost Spanning Tree

With applications of weighted graphs, it is often necessary to find a spanning tree for which the total weight of the edges in the tree is as small as possible. Such a spanning tree is called a minimal spanning tree or minimum cost spanning tree.

Fig. 3.9 shows a weighted graph and its minimum-cost spanning tree.

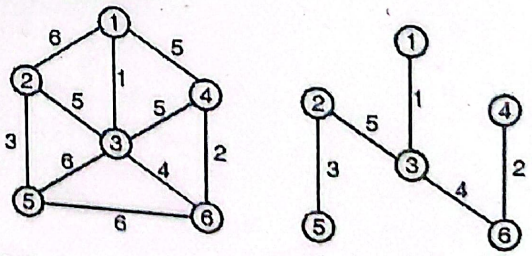


Fig. 3.9 : A graph and its minimum cost spanning tree

There are two popular techniques for constructing a minimum cost spanning tree from a weighted graph G = (V, E).

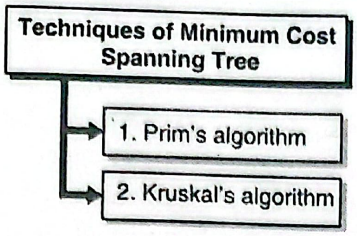


Fig. 3.9(a) : Techniques of minimum cost spanning tree

1. Prim's Algorithm

Let the graph G = (V, E) has n vertices.

Step 1 : Choose a vertex V_1 of G. Let $V_T = \{V_1\}$ and $E_T = \{\}$.

Step 2 : Choose a nearest neighbour V_i of V_T that is adjacent to V_j, $V_j \in V_T$ and for which the edge (V_i, V_j) does not form a cycle with members of E_T. Add V_i to V_T and add (V_i, V_j) to E_T.

Step 3 : Repeat step 2 until $| E_T | = n - 1$. Then V_T contains all n vertices of G and E_T contains the edges of the minimum cost spanning tree of G.

In the above algorithm, begin at any vertex of G and construct a minimum cost spanning tree by adding an edge to a nearest neighbour of the set of vertices already linked. The edge to be added should not form a cycle.

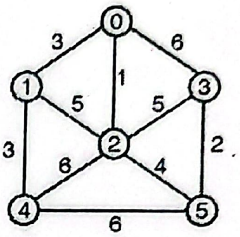
Algorithm avoids cycle in the minimum cost spanning through a rather better approach.

- Algorithm works in stages. In each stage, one new vertex is added through an edge to the tree.
- At each stage, a set of vertices that have already been added to the tree and the remaining that have not been. If a vertex i is in the tree then

  vertex[i] = 1

  else

  vertex[i] = 0 //vertex i is not in the tree.
- Find a new vertex V to be added to the tree, such that
  - (u, V) is an edge in the graph
  - u is in the tree and V is not
  - (u, V) is the smallest weight edge among the remaining edge.
- Two arrays Distance[] and from[] are used to find the new vertex V to be added to the tree.
- Initial values of visited[], Distance[] and from[] are shown in the Fig. 3.9(b).



	0	1	2	3	4	5	
visited	1	0	0	0	0	0	
distance	0	3	1	6	∞	∞	
from	0	0	0	0	0	0	

(0) → Node added to the tree

Fig. 3.9(b) : Graph and the initial values of visited[], distance[], from[] and the tree

Start constructing the spanning tree beginning with the vertex 0. Array Distance[] shows the distance of every vertex from the vertex 0. Array from[] says that the distances in the distance[] array are from the vertex 0.

2. Kruskal's Algorithm

It is another method for finding the minimum cost spanning tree of the given graph. In Kruskal's algorithm, edges are added to the spanning tree in increasing order of cost. If the edge E forms a cycle in the spanning, it is discarded. Fig. 3.9(c) shows the sequence in which the edges are added to the spanning tree.

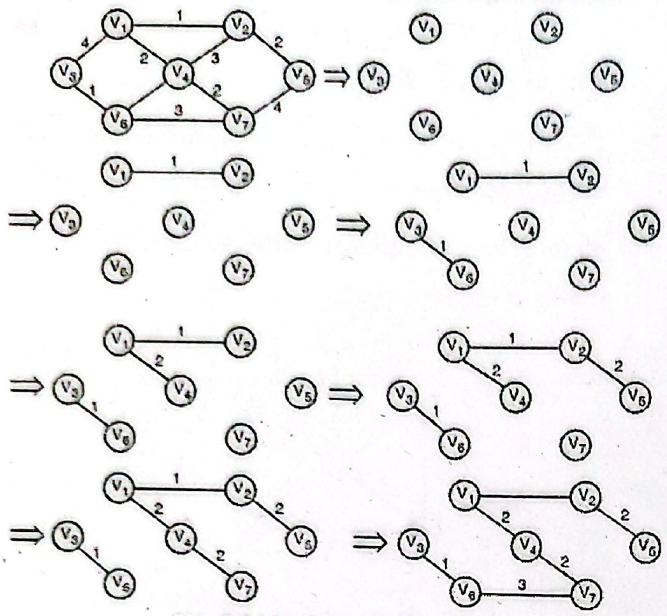
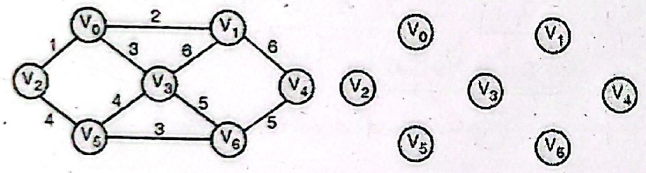


Fig. 3.9(c) : A graph G and its minimum cost spanning tree

Formally, Kruskal's algorithm maintains a collection of components. Initially, there are n components. Every node is a component.



Component number 0 = $((V_0), \phi)$ –[A component with one vertex V_0 and no edges]

Components number 1 = $((V_1), \phi)$

Components number 2 = $((V_2), \phi)$

Components number 3 = $((V_3), \phi)$

Components number 4 = $((V_4), \phi)$

Components number 5 = $((V_5), \phi)$

A vertex V_i belongs to component number k if belongs[V_i] is equal to k.

Initial configuration of belongs[] is

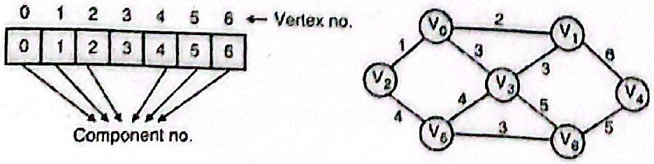


Fig. 3.9(d) : A graph G and initial list of components

Adding an edge to spanning tree merges two components into one.

When an edge (V_0, V_2) with weight 1 is added to the spanning tree, the component with one of the vertices as V_0 and the component with one of the vertices as V_2 will be merged into a single component. To carry out the above operation, two functions are written :

Find (V) → gives the component number of the vertex V

Union (C_1, C_2) → merges two components C_1 and C_2 into C_1.

Q.14 Represent the following graph using adjacency list and fixed the minimum spanning tree using prim's algorithm. Write all the sequence of steps used in algorithm. **(8 Marks)**

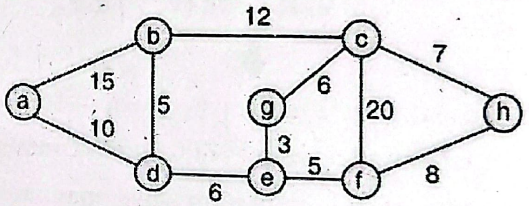
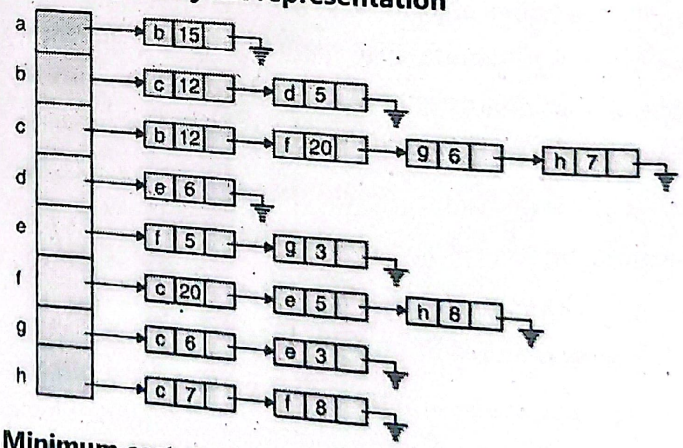


Fig. 3.10

Ans. : Adjacency list representation



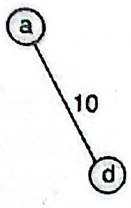
Minimum cost spanning tree using Prim's algorithm

Step 1 :

Nodes	(A)	B	C	D	E	F	G	H
Distance	0	15	∞	10	∞	∞	∞	∞
Distance from	-	A	A	A	A	A	A	A

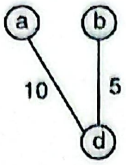
↑

Node at minimum distance



Step 2 :

Nodes	(A)	B	C	(D)	E	F	G	H
Distance	0	5	∞	10	6	∞	∞	∞
Distance from	-	D	A	A	D	A	A	A

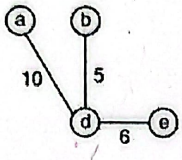


↑

Node at minimum distance

Step 3 :

Nodes	(A)	(B)	C	(D)	E	F	G	H
Distance	0	5	12	10	6	∞	∞	∞
Distance from	-	D	B	A	D	A	A	A

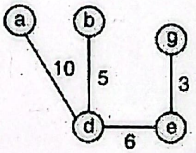


↑

Node at minimum distance

Step 4 :

Nodes	(A)	(B)	C	(D)	(E)	F	G	H
Distance	0	5	12	10	6	5	3	∞
Distance from	-	D	B	A	D	E	E	A

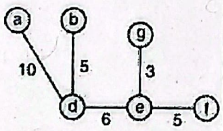


↑

Node at minimum distance

Step 5 :

Nodes	(A)	(B)	C	(D)	(E)	F	(G)	H
Distance	0	5	6	10	6	5	3	∞
Distance from	-	D	G	A	D	E	E	A



↑

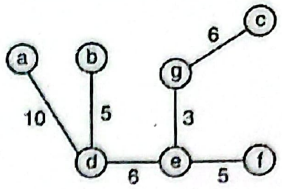
Node at minimum distance

Step 6 :

Nodes	(A)	(B)	C	(D)	(E)	(F)	(G)	H
Distance	0	5	6	10	6	5	3	8
Distance from	-	D	G	A	D	E	E	F

↑

Node at minimum distance

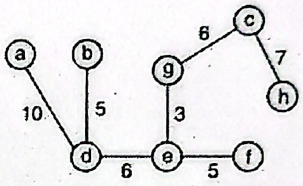


Step 7 :

Nodes	(A)	(B)	(C)	(D)	(E)	(F)	(G)	(H)
Distance	0	5	6	10	6	5	3	7
Distance from	-	D	G	A	D	E	E	C

↑

Node at minimum distance



Q.15 Consider the following graph represented using adjacency list. Find the minimum spanning tree for the graph by using Prim's algorithm.

SPPU - Dec. 14, 4 Marks

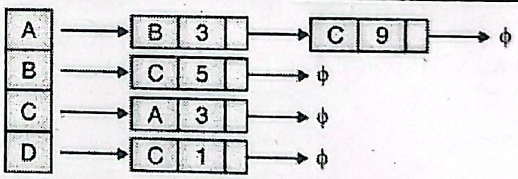
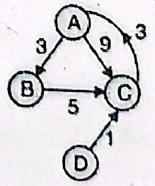


Fig. 3.11

Soln. : The graph is shown as follows :

Prim's algorithm cannot be used for a directed graph.



Q.16 Program for constructing a minimum cost spanning tree of a graph using Kruskal's algorithm. **(5 Marks)**

Ans.:

```
#include<iostream.h>
#include<conio.h>
#define MAX 30
```

```
struct edge
{ int u,v,w;
};
class  edgelist
{ edge data [MAX];
 int n;
 public:
    friend class graph;
    edgelist(){n=0;}
    void sort( );
    void print( );
};
void edgelist::sort( )
{  int i,j;
  edge temp;
  for(i=1;i<n;i++)
   for(j=0;j<n-i;j++)
    if(data[j].w>data[j+1].w)
      { temp=data[j];
        data[j]=data[j+1];
        data[j+1]=temp;
      }
}
 void edgelist::print()
 { int i; int cost=0;
  for(i=0;i<n;i++)
 { cout<<"\n"<<data[i].u<<"  "<<data[i].v<<""<<data[i].w;
     cost=cost+data[i].w;
   }
  cout<<"\ncost of the spanning tree = "<<cost;
 }
 //edgelist elist;

 class graph
 { int G[MAX][MAX];
  int n;
 public:
    graph()
    { n=0; }
    void readgraph();
    void printgraph();
    void kruskal(edgelist &spanlist );
};
void graph:: readgraph()
{ int i,j;
 cout <<"\nEnter No. of vertices : ";
 cin>>n;
 cout<<"\nEnter the adjacency matrix : ";
    for(i=0;i<n;i++)
     for(j=0;j<n;j++)
      cin>>G[i][j];
}
void  graph:: printgraph()
 { int i,j;
  for(i=0;i<n;i++)
  { cout<<"\n";
    for(j=0;j<n;j++)
      cout<<" "<<G[i][j];
  }
 }
int find(int belongs [], int vertexno);
// component no .of vertexno
void union1(int belongs[], int c1, int c2, int n);
// combining two components
void main( )
{  edgelist spanlist;
  graph g1;
  //int i, j, total_cost;
  g1.readgraph();
  g1.kruskal(spanlist);
  spanlist.print();
  getch();
```

```
}
void graph::kruskal(edgelist &spanlist )
{ int belongs[MAX],i,j,cno1,cno2;
 edgelist elist;
 for(i=1;i<n;i++)
  for(j=0;j<i;j++)
  { if(G[i][j] !=0)
    { elist.data[elist.n].u=i;
     elist.data[elist.n].v=j;
     elist.data[elist.n].w=G[i][j];
     elist.n++;
    }
  }
  elist.sort();
  for(i=0;i<n;i++)
   belongs[i]=i;
  for(i=0;i<elist.n;i++)
   {
       cno1=find(belongs,elist.data[i].u);
       cno2=find(belongs,elist.data[i].v);
       if(cno1!=cno2)
       { spanlist.data[spanlist.n]=elist.data[i];
            spanlist.n=spanlist.n+1;
            union1(belongs,cno1,cno2,n);
       }
   }
}
int find(int belongs[],int vertexno)
{ return(belongs[vertexno]);
}
void union1(int belongs[],int c1,int c2,int n)
{  int i;
   for(i=0;i<n;i++)
    if(belongs[i]==c2)
     belongs[i]=c1;
}
```

Output

```
enter no of vertices:6

enter the adjacency matrix:0 3 1 6 0 0
3 0 5 0 3 0
1 5 0 5 6 4
6 0 5 0 0 2
0 3 6 0 0 6
0 0 4 2 6 0

2   0   1
5   3   2
1   0   3
4   1   3
5   2   4
cost of the spanning tree =19
```

Q.17 Differentiate between Prim's and Kruskal's algorithm for generating the spanning tree of the graph.

SPPU - Dec. 13, 3 Marks

Ans.: Comparison of Time Complexity of Prim's and Kruskal's Algorithm

The complexity of Prim's algorithm = $O(n^2)$

Where, n is number of vertices.

Time complexity of Kruskal's algorithm

= O(e loge) + O(e logn)

Where, n is number of vertices and e is number of edges.

For a dense graph, O(e logn) may become worse than $O(n^2)$. Hence, the Kruskal's algorithm should be avoided for a dense graph. Kruskal's algorithm performs better than Prim's algorithm for a sparse graph.

Q.18 Draw the minimum cost spanning tree using Kruskal for the graph given below. Also find its cost show all steps. **(5 Marks)**

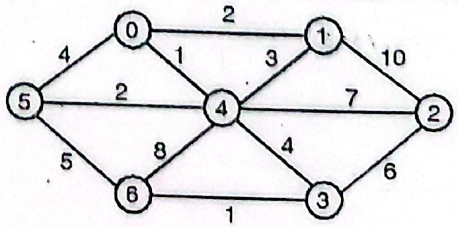


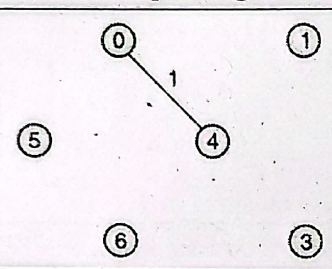
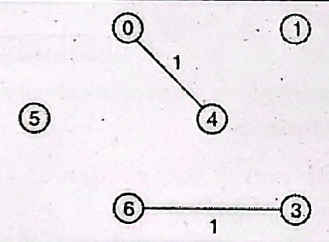
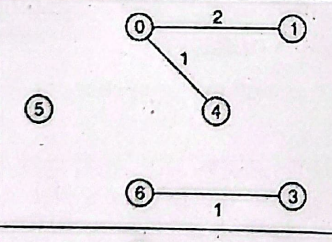
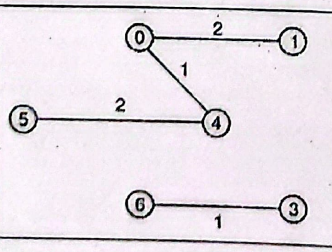
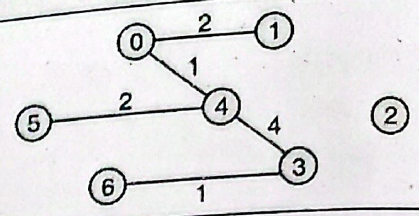
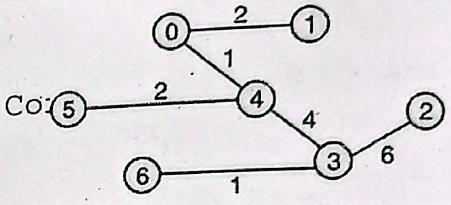
Fig. 3.12

Ans.:

Step 1 : Edges are sorted in ascending order on weight.

Edges : E1 : 0 4 1 E2 : 3 6 1
E3 : 0 1 2 E4 : 4 5 2
E5 : 1 4 3 E6 : 0 5 4
E7 : 3 4 4 E8 : 5 6 5
E9 : 2 3 6 E10 : 2 4 7
E11 : 4 6 8 E12 : 1 2 10

Step 2 : Edges are added in sequence from E1 to E12 to spanning tree. If an edge forms a cycle, it is discarded.

Edge added	Spanning tree
E1 : 0 4 1	
E2 : 3 6 1	
E3 : 0 1 2	
E4 : 4 5 2	
E5 : 1 4 3	Edge is discarded.
E6 : 0 5 4	Edge is discarded.
E7 : 3 4 4	
E8 : 5 6 5	Edge is discarded.
E9 : 2 3 6	

Cost of minimum spanning tree

$= 2 + 1 + 2 + 4 + 1 + 6 = 16.$

Q.19 Convert given graph into MST refer Fig. 3.13

SPPU - May 14, 3 Marks

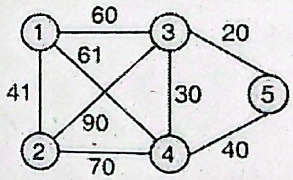
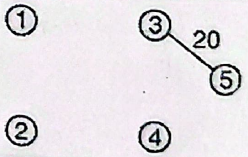
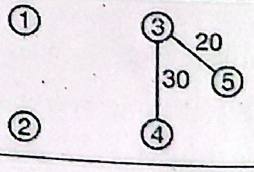
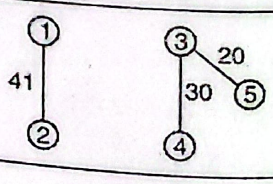
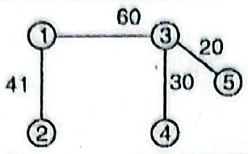


Fig. 3.13

Ans.: Edges are added in the ascending order of weight.

Sr. No.	Edge to be added	Graph
1.	(3, 5)	
2.	(3, 4)	
3.	(4, 5)	This can't be added. It will form a cycle.
4.	(1, 2)	

Sr. No.	Edge to be added	Graph
5.	(1, 3)	

Q.20 Write a short note on : Dijkstra Algorithm. **(5 Marks)**

Ans.: Dijkstra Algorithm

Dijkstra algorithm is also called single source shortest path algorithm. It is based on "greedy" technique. The algorithm maintains a list 'visited[ ]' of vertices, whose shortest distance from the source is already known.

If visited[1], equals 1, then the shortest distance of vertex i is already known. Initially, visited[i] is marked as 1 for source vertex.

At each step, mark visited[V] as 1 Vertex V is a vertex at shortest distance from the source vertex. At each step of the algorithm, shortest distance of each vertex is stored in an array 'distance[ ]'.

Algorithm

1. Create cost matrix C[ ][ ] from adjacency matrix adj[ ][ ]. C[i][j] is the cost of going from vertex i to vertex j. If there is no edge between vertices i and j then C[i][j] is infinity.
2. Array visited [ ] is initialized to zero.

```
for (i = 0; i < n; i++)
    visited[i] = 0;
```

3. If the vertex 0 is the source vertex then visited[0] is marked as 1.
4. Create the distance matrix, by storing the cost of vertices from vertex no. 0 to n – 1 from the source vertex 0.

```
for (i = 1; i < n; i++)
distance[i] = cost[0][i];
```

Initial, distance of source vertex is taken as 0.

i.e. distance[0] = 0;

5. for (i = 1; i < n; i++)
   - Choose a vertex w, such that distance[w] is minimum and visited[w] is 0.
   - Mark visited[w] as 1.
   - Recalculate the shortest distance of remaining vertices from the source. Only, the vertices not marked as 1 in array visited[ ] should be considered for recalculation of distance.

   i.e. for each vertex v

   if (visited[v] = = 0)

distance[v] = min (distance[v], distance[w] + cost[w][v])

Timing Complexity

The program contains two nested loops each of which has a complexity of O(n). n is number of vertices. So the complexity of algorithm is $O(n^2)$.

Q.21 What is topological ordering ? List their applications.

SPPU - Dec. 19, 3 Marks

Ans.: Topological Sorting

A topological sort is an ordering of vertices in a directed a cyclic graph, such that if there is a path from V_i to V_j then V_j appears after V_i in the ordering.

Topological sorting is important in project scheduling.

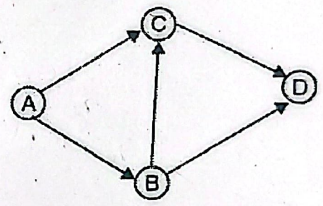


Fig. 3.14(a)

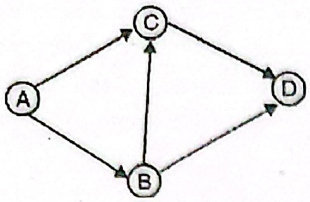
Four events A,B,C and D are interconnected as shown in the Fig. 3.14(a).

- Event B can be started after completion of event A.
- Event C can be started after completion of events A and B.
- Event D can be started after completion events C and B.

Therefore, events must be scheduled as per the given sequence :

A B C D.

A simple algorithm to find a topological ordering is first to find any vertex, indegree = 0. Print this vertex, and remove it, along with edges , form the graph. Then apply this same strategy to the rest of the graph shown below, is step wise implementation of topological sort on graph of Fig. 3.14(a).

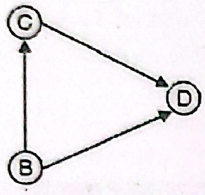


A	B	C	D
0	1	2	2

Indegree of nodes

Fig. 3.14(b) : Initial condition

Step 1 : Delete node A along with edges. Reduce indegree of nodes adjacent to A by 1.

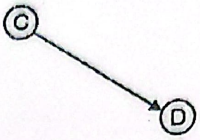


B	C	D
0	1	2

Indegree

Graph after deletion of node A.

Step 2 : Delete node B along with edges. Reduce indegree of nodes adjacent to B by 1.



C	D
0	1

Indegree

Graph after deletion of node B.

Step 3 : Delete node C along with edges. Reduce indegree of nodes adjacent to C by 1.

0

Indegree

Step 4 : Finally, Delete the only node D to complete the algorithm.

Q.22 Find a topological sorting of given graph. **SPPU - Dec. 19, 3 Marks**

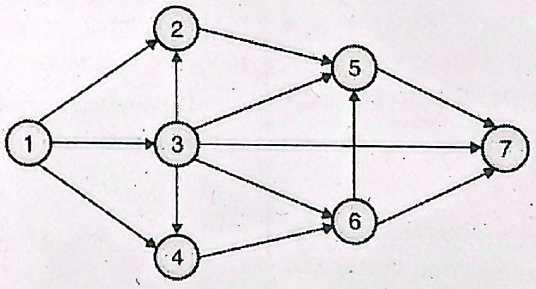


Fig. 3.15

Soln. : Refer Fig. 3.15.

Sr. No.	Node Deleted (A node of indegree 0)	Indegree ① ② ③ ④ ⑤ ⑥ ⑦
1.	–	0 2 1 2 3 2 3
2.	①	– 1 0 1 3 2 3
3.	③	– 0 – 0 2 1 2
4.	②	– – – 0 1 1 2
5.	④	– – – – 1 0 2
6.	⑥	– – – – 0 – 1
7.	⑤	– – – – – – 0
8.	⑦	– – – – – – –

Topological ordering = 1, 3, 2, 4, 6, 5, 7.

Q.23 Sort the digraph for topological sort. Refer Fig. 3.16. **SPPU - May 14, 3 Marks**

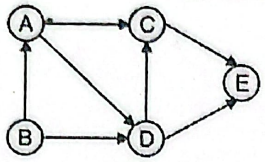
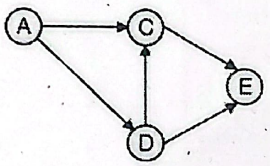


Fig. 3.16

Ans. :

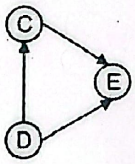
	A	B	C	D	E
Indegree	1	0	2	2	2

1) Delete node B along with edges B is a node of indegree 0.



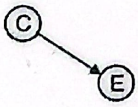
	A	C	D	E
Indegree	0	2	1	2

2) Delete node A along with edges. A is a node of indegree 0.



	C	D	E
Indegree	1	0	2

3) Delete node D along with edges. D is a node of indegree 0.



	C	E
Indegree	0	1

4) Delete node C.

5) Finally delete node E.

Topological sorting list = B, A, D, C, E

Q.24 Explain Floyd-warshall Algorithm. **(6 Marks)**

Ans.:

Floyd-warshall Algorithm

It is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph. This algorithm follows the dynamic programming approach to find the shortest path.

A C-function for a N × N graph is given below. The function stores the all pair shortest path in the matrix cost [N][N]. The cost matrix of the given graph is available in cost Mat [N][N].

```
# define N 4
void floydwarshall()
{
    int cost [N][N];
    int i, j, k;
    for(i=0; i<N; i++)
    for(j=0; j<N ; j++)
    cost [i] [j] = cost Mat [i] [j];
    for(k=0; k<N; k++)
{
    for(i = 0; i<N; i++)
    for(j = 0; j<N; j++)
    if(cost [i] [j] > cost [i] [k] + cost [k] [j];
    cost [i] [j] = cost [i] [k] + cost [k] [j];
}
//display the matrix cost [N] [N]
}
```

The algorithm is explained step-wise for the graph given in Fig. 3.17.

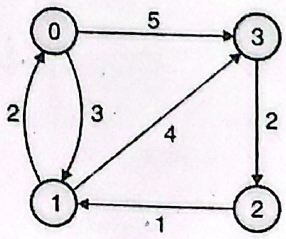


Fig. 3.17 : Given graph

Step 1 : k = 0

Create a 4 × 4 matrix A^0 as the cost matrix.

$$A^0 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & \infty & 4 \\ \infty & 1 & 0 & \infty \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

Step 2 :

Create the matrix A^1, k = 1

$$A^1 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & 9 & 4 \\ \infty & 1 & 0 & 8 \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

Step 3 :

Create the matrix A^2, k = 2

$$A^2 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 3 & 9 & 5 \\ 2 & 0 & 9 & 4 \\ 3 & 1 & 0 & 5 \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

Step 4 :

Create the matrix A^3, k = 3

$$A^3 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 3 & 9 & 5 \\ 2 & 0 & 9 & 4 \\ 3 & 1 & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

Step 5 :

Create the matrix A^4 containing the shortest path k = 4.

$$A^4 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 3 & 7 & 5 \\ 2 & 0 & 6 & 4 \\ 3 & 1 & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

❑❑❑

Unit IV : Search Trees

Q.1 Define Symbol Tables. **(2 Marks)**

Ans.: Symbol Tables

> **Definition :** *It is a standard data structure used in language processing. Information about various source language constructs can be stored in a symbol table. For example, a compiler scans the source file and stores an identifier, a string of characters in a symbols - table entry. Here, an identifier is a symbol. Thus a table of symbols is called a symbol table.*

Each symbol can have a list of associated attributes. A symbol along with the list of associated attributes forms an entry of a symbol table.

Symbol	Attributes
Total	----------
X	----------
i	----------

Fig. 4.1 : A symbol table entries for a c - statement, total = x + i

Attributes of a symbol (identifier) could be :

- Type of identifier
- Its usage (e.g. function name, variable, lable)
- Its memory address

Symbol tables are concerned primarily with saving and retrieving of symbols. The following operations are performed on the symbol table.

Insert (s,t) : Inserts a new entry for string s and associated attributes t.

Lookup (s) : Returns index of the entry for string s.

Q.2 Enlist the names of static tree tables with suitable example. **SPPU - Dec. 13, Dec. 18, 2 Marks**

Ans.: Static Tree Tables

In a static table, symbols are known in advance and no insertions are deletions are allowed. Huffman tree and OBST are examples of static tree tables. Cost of searching a symbol occurring with higher frequency should be small.

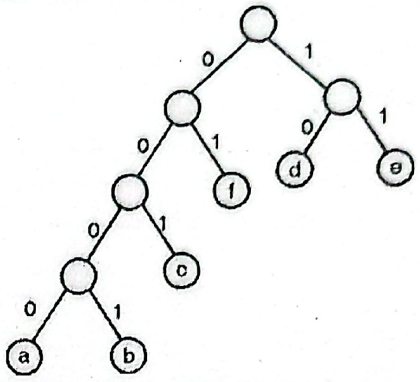


Fig. 4.2(a) : A Huffman's tree for code with prefix property

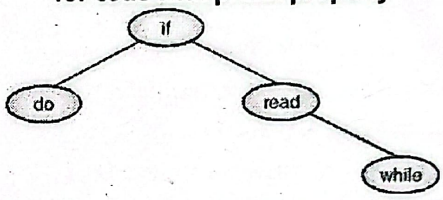


Fig. 4.2(b) : Optimal search tree when the frequency of symbol's have been specified

Q. 3 Explain the dynamic tree tables with suitable example. **SPPU - Dec. 18, 3 Marks**

Ans.: Dynamic Tree Tables

In a dynamic table, symbols are inserted as and when they come. They are deleted when they are no longer required. AVL tree is an example of dynamic tree table.

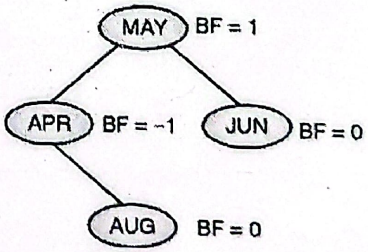


Fig. 4.3 : An AVL tree

Q. 4 Explain Dynamic programming with principle of optimality. **SPPU - Dec. 18, 3 Marks**

Ans.: Dynamic Programming

Dynamic programming is based on non-recursive approach. In this approach, answers to the subproblems are recorded. The concept of dynamic programming with the help of fibonacci numbers. The natural recursive program to compute fibonacci numbers is very

inefficient. To compute fibonacci F_N, there is one call to F_{N-1} and F_{N-2}. However, since F_{N-1} recursively makes a call to F_{N-2} and F_{N-3}, there are actually two separate calls to F_{N-2}. If trace our algorithm then see that :

F_{N-2} is computed two times

F_{N-3} is computed three times

F_{N-4} is computed four times and so on.

Avoid computation of same number again and again by recorded the value. i.e. if F_{N-3} is computed once, it is recorded and subsequently, recursive calls are not made.

For Example

$$F_2 = F_1 + F_0 = 1 + 0 = 1$$

F_2, is recorded and it can be used for computation of F_3

$$F_3 = F_2 + F_2 = 1 + 1 = 2$$

Q.5 List binary search trees with three words.

(w_1, w_2, w_3) = (do, if, stop)

Words occur with the following probabilities.

(p_1, p_2, p_3) = (0.4, 0.2, 0.1)

Probabilities of failures are given below.

(q_0, q_1, q_2, q_3) = (0.1, 0.1, 0.05, 0.05)

Find the expected access time in each case.

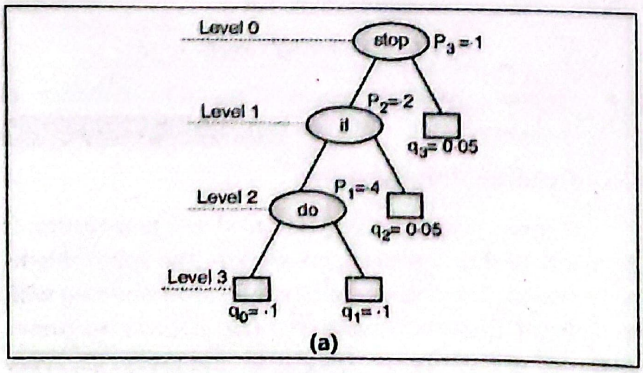
(5 Marks)

Ans. : In a binary search tree, the number of comparisons needed to search an element with success at depth d is d + 1. Similarly, the number of comparisons needed to search an element with failure (failure node at depth p) = d'

∴ Expected number of comparisons (both success and failures)

$$= p_1(d_1+1) + p_2(d_2+2) + \ldots + p_n(d_n+1) + q_0 \cdot d_{f0} + q_1 d_{f1} + \ldots + q_n d_{fn}$$

↓ depth of a failure node whose probability of occurrence is q_1.

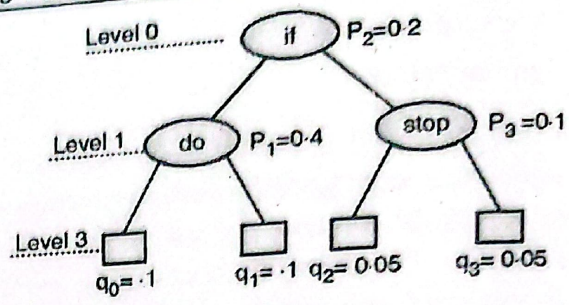


(a)

Expected number of comparisons

$= q_0 \times 3 + q_1 \times 3 + q_2 \times 2 + q_3 \times 1 + p_1 \times 3 + p_2 \times 2 + p_3 \times 1$

$= 0.3 + 0.3 + 0.10 + 0.05 + 1.2 + 0.4 + 0.1 = 2.45$

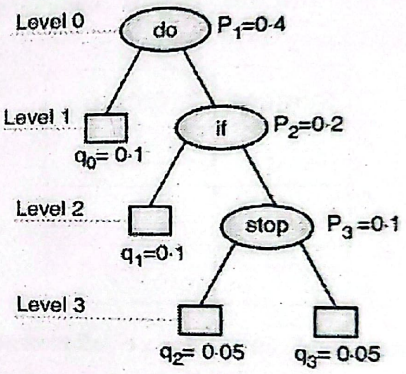


(b)

Expected number of comparisons

$= q_0 \times 2 + q_1 \times 2 + q_2 \times 2 + q_3 \times 2 + p_1 \times 2 + p_2 \times 1 + p_3 \times 2$

$= 0.2 + 0.2 + 0.1 + 0.1 + 0.8 + 0.2 + 0.2 = 1.8$

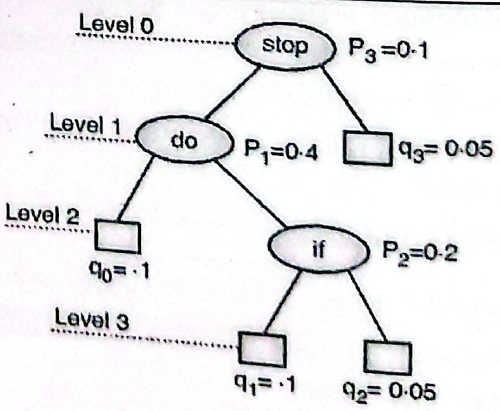


(c)

Expected number of comparisons

$= q_0 \times 1 + q_1 \times 2 + q_2 \times 3 + q_3 \times 3 + p_1 \times 1 + p_2 \times 2 + p_3 \times 3$

$= 0.1 + 0.2 + 0.15 + 0.15 + 0.4 + 0.4 + 0.3 = 1.7$

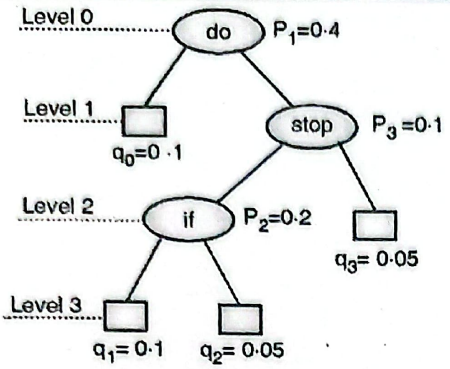


(d)

Expected number of comparisons

$= q_0 \times 2 + q_1 \times 3 + q_2 \times 3 + q_3 \times 1 + p_1 \times 2 + p_2 \times 3 + p_3 \times 1$

$= 0.2 + 0.3 + 0.15 + 0.05 + 0.8 + 0.6 + 0.1 = 2.2$



(e)

Expected number of comparisons

$= q_0 \times 1 + q_1 \times 3 + q_2 \times 3 + q_3 \times 2 + p_1 \times 1 + p_2 \times 3 + p_3 \times 2$

$= 0.1 + 0.3 + 0.15 + 0.1 + 0.4 + 0.6 + 0.2 = 1.85$

Fig. 4.4 : Possible binary search trees with failure nodes

Binary search tree of Fig. 4.4(c) is optimal with expected number of comparisons = 1.7.

Q.6 Define AVL tree. **(2 Marks)**

Ans.: AVL Trees

An AVL (Adelson-Velskii and Landis) tree is a height balance tree. These trees are binary search trees in which the heights of two siblings are not permitted to differ by more than one.

i.e. | Height of the left subtree – height of the right subtree | ≤ 1

Searching time in a binary search tree is O(h), where h is the height of the tree. For efficient searching, it is necessary that height should be kept to minimum. A full binary search tree with n nodes will have a height of $O(\log_2 n)$. In practice, it is very difficult to control the height of a BST. It lies between O(n) to $O(\log_2 n)$. An AVL tree is a close approximation of full binary search tree.

Q.7 Explain Insertion of a Node into an AVL Tree. **(8 Marks)**

Ans.: Insertion of a Node into an AVL Tree

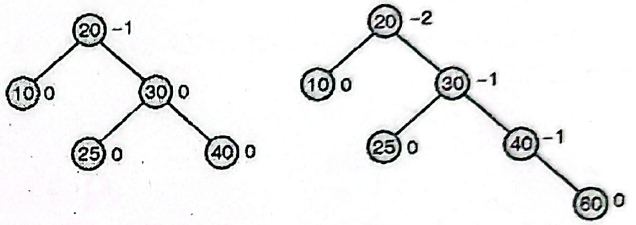
Insertion of a new data, say k into an AVL tree is carried out in two steps :

1) Insert the new element, treating the AVL tree as a binary search tree.

2) Update the balance factors (information, height) working upward from the point of insertion to the root. It should be clear that after insertion, the nodes on the path from point of insertion to the root may have their height altered.

The steps to insert a new element with value k into an AVL tree begins with comparing k with the value stored in the root.

If k is found to be larger than the value stored in T, then insertion is carried out into the right subtree else insertion is carried out into the left subtree. The recursion terminates when the left or right subtree to which insertion is to be made happens to be empty.



(a) A sample AVL tree with balance factors **(b) Tree of Fig. 4.5 (a) after insertion of 60. Tree is not longer an AVL tree**

Fig. 4.5

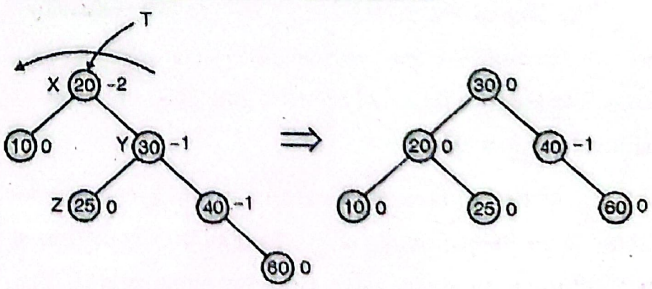
Consider the AVL tree of Fig. 4.5. Balance factor of each node is shown against the node.

A new element with value 60 is inserted in a way insert an element in a binary search tree. After insertion of 60, the balance factor of root has become –2. Hence, it is no longer an AVL tree. The balance factor of the root has become –2, because the height of its right subtree rooted at (30) has increased by 1.

Rebalancing of the tree is carried out through rotation of the deepest node with BF = 2 or BF = –2.

Rotation

Fig. 4.6 shows the rotation of tree of Fig. 4.5. After rotation, the tree becomes an AVL tree.



(a) Before rotation **(b) After rotation**

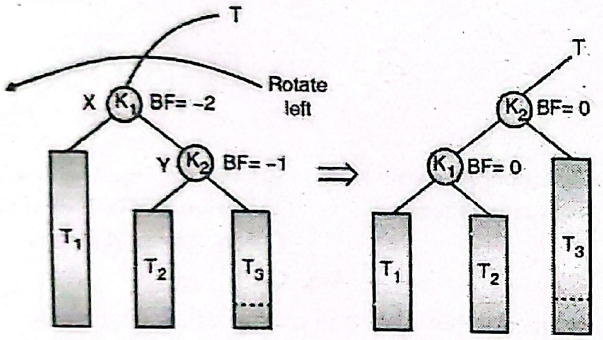
Fig. 4.6 : AVL property is destroyed by insertion of 60. AVL property is fixed by left rotation of tree rooted at X

- Balance factor of node X in Fig. 4.6(a) is –2.
- Balance factor of –2, tells that the right subtree of X is heavier.
- In order to rebalance the tree, the tree rooted at X (node with BF = –2) is rotated left.

1. Rotate Left

- When a tree rooted at X is rotated left, the right child of X i.e. Y will become the root.
- The node X will become the left child of Y.
- Tree T_2, which was the left child of Y will become the right child of X.

A program segment to rotate left a tree T, rooted at node X.



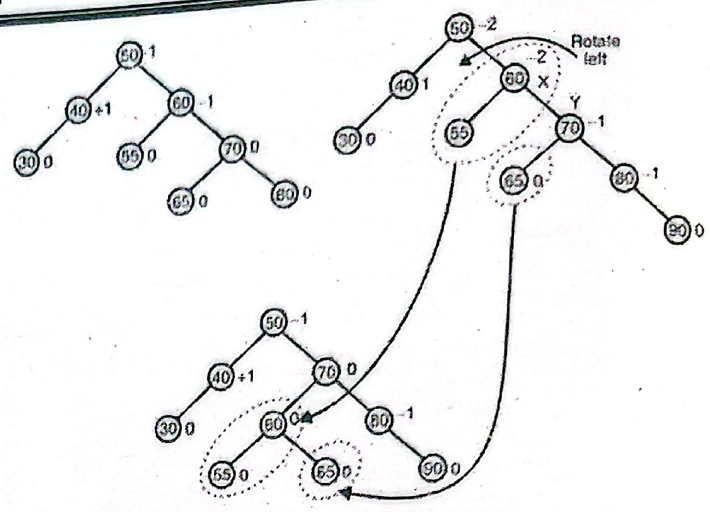
(a) Tree with BF = – 2 at node X **(b) Tree after rotation**

Fig. 4.7

```
node *temp;
temp = Y → left ; save the address of left child of Y.
T = Y ; Node Y becomes the root node.
Y → left = X ; X becomes the left child of Y.
X → right = temp; left child of Y becomes the right child of X.
```

The tree of Fig. 4.7(a) is an AVL tree. After insertion of the element 90, the tree no longer remains an AVL tree. The balance factor of nodes X and Z has becomes –2 shown in Fig. 4.7(b).

In order to restore back the balance nature, the node X (the deepest node with BF equal to ± 2) is rotated left. After rotation, tree once again becomes an AVL tree. Tree after rotation is shown in the Fig. 4.8(c).

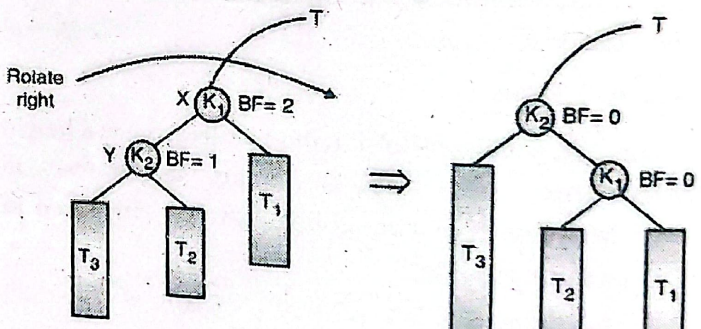


(a) An AVL tree **(b) Tree after insertion of 90** **(c) Tree after rotation**

Fig. 4.8 : An example of left rotation

2. Rotate Right

- When a tree rooted at X is rotated right, the left child of X i.e. Y will become the root.
- The node X will become the right child of Y.

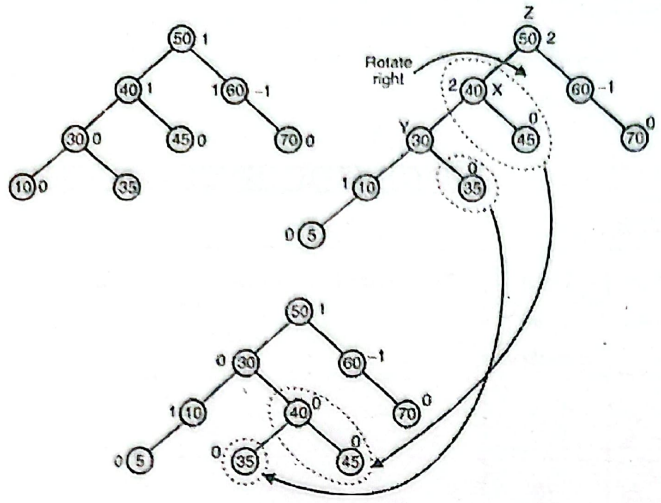


(a) Tree with BF = 2 at node X **(b) Tree after rotation**

Fig. 4.9

- Tree T_2, which was the right child of Y will become the left child of X.
- Balance factor of +2, tells that the left subtree of X is heavier. In order to rebalance the tree, the tree rooted at X (node with BF = 2) is rotated right;
- A program segment to rotate right a tree T, rooted at node X (as shown in Fig. 4.9).

```
node *temp;
temp = Y → right; save the address of right child of Y.
T = Y; Node Y becomes the root node.
Y → right = X ; X becomes the right child of Y.
X → left = temp; right child of Y becomes the left child of X.
```



(a) An AVL tree (b) Tree after insertion of 5 (c) Tree after rotation

Fig. 4.10 : An example of right rotation

The tree of Fig. 4.10(a) is an AVL tree. After insertion of the element with value 5, the tree no longer remains an AVL tree.

The balance factors of nodes X and Z has become 2. Shown in Fig. 4.10(b). In order to restore back the balance nature, the node X (the deepest node with BF equal ± 2) is rotated right. After rotation, the tree has once again become an AVL tree. Tree after rotation is shown in Fig. 4.10(c).

Q.8 Write functions for LL and LR rotation with respect to AVL tree. SPPU - May 17, 6 Marks

Ans.:

Left of left rotation

Let X be the node with BF equal to +2 after insertion of the new node A.

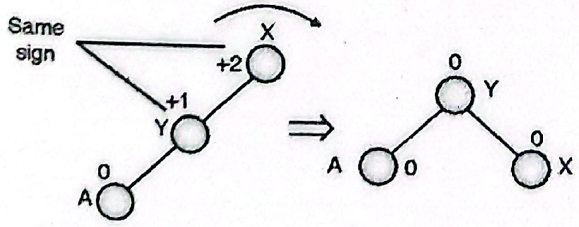


Fig. 4.11(a) : Situation known as LL (left of left)

New node A is inserted in the left subtree of the left subtree of X. Balance nature of the tree can be restored through single right rotation of the node X. It is shown in the Fig. 4.11(a).

LR : Let X be the node with BF equal to +2, after insertion of the new node A. Balance factor of the node Y, the left child of the node X becomes –1 after insertion of A.

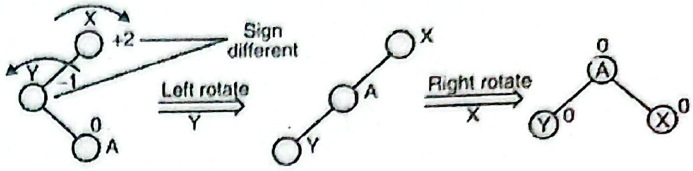


Fig. 4.11(b) : Situation known as LR (right of left)

New node A is inserted in the right subtree of the left subtree of X. Balance nature of the tree can be restored through double rotation

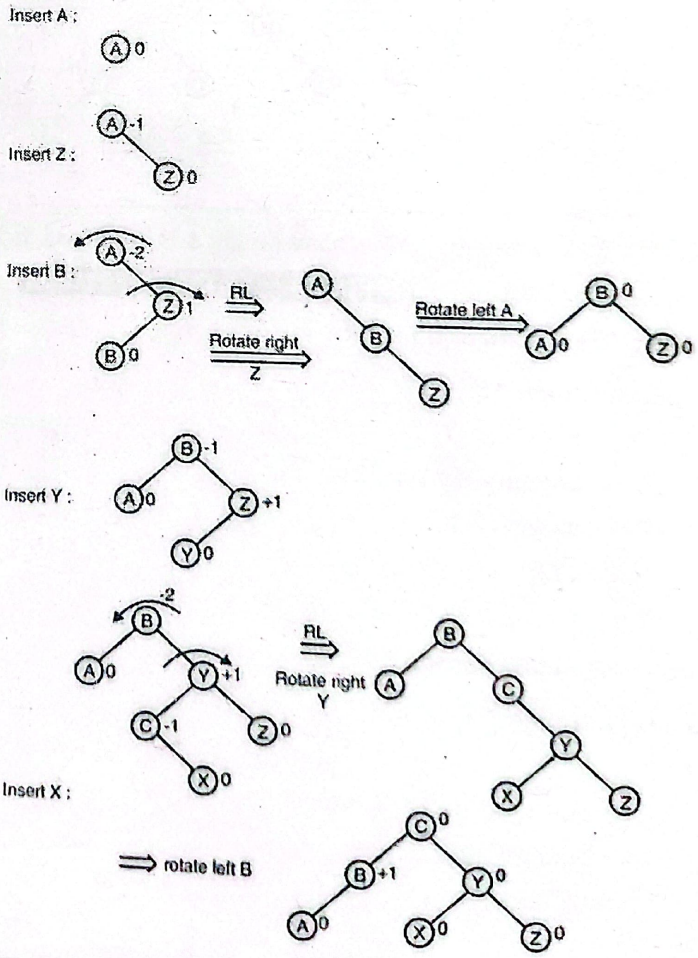
a) node Y is rotated left

b) node X is rotated right.

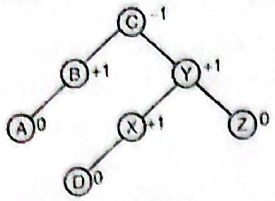
It is shown in the Fig. 4.11(b).

Q.9 Draw diagram to show different stages during the building of AVL tree for the following sequence of keys : A, Z, B, Y, C, X, D, U, E. In each case show the balanced factor of all the nodes and name the type of rotation used for balancing. **(6 Marks)**

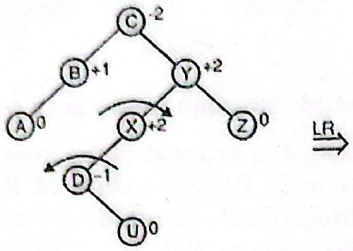
Ans. :



Insert D :

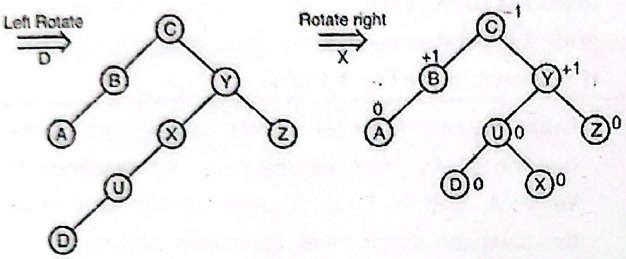


Insert U :



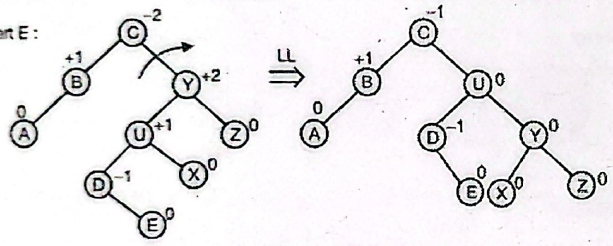
Left Rotate D

Rotate right X



Insert E :

LL



Q.10 Write a pseudo C/C++ code for LR & RL rotations in AVL Trees. **SPPU - Dec. 14, Dec .19, 6 Marks**

Ans.: 'C++' Function for LR

```
node *LR(node *T)
{
    T->left=rotateleft(T->left);
    T=rotateright(T);
    return(T);
}
```

'C++' Function for RL

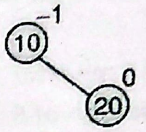
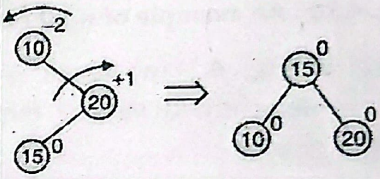
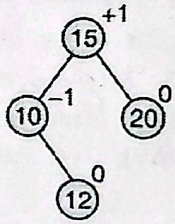
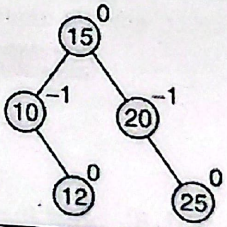
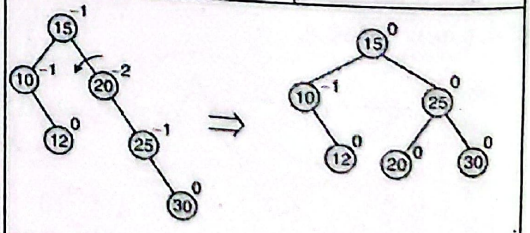
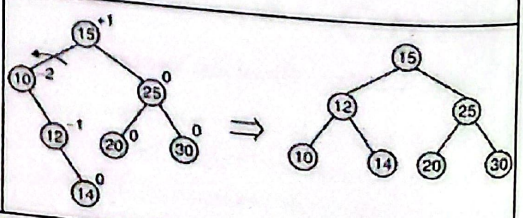
```
node *RL(node *T)
{
    T->right=rotateright(T->right);
    T=rotateleft(T);
    return(T);
}
```

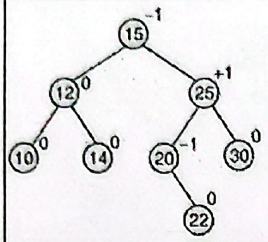
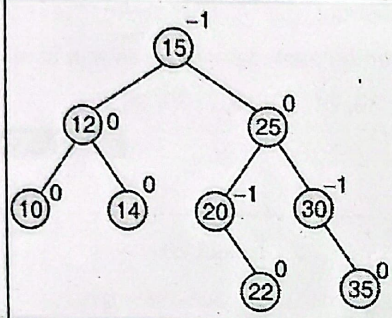
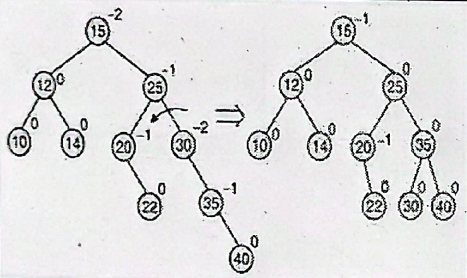
Q.11 Construct the AVL tree for the following data by inserting each of the following data item one at a time :

10, 20, 15, 12, 25, 30, 14, 22, 35, 40.

SPPU - Dec. 13, Dec. 14, Dec. 19, 5 Marks

Ans. :

Sr. No.	Data to be inserted	Tree after insertion	Tree after rotating
1.	10		
2.	20		
3.	15		
4.	12		
5.	25		
6.	30		
7.	14		

Sr. No.	Data to be inserted	Tree after insertion	Tree after rotating
8.	22		
9.	35		
10.	40		

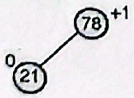
Q.12 Create all AVL tree for the following data : 78, 21, 14, 11, 97, 85, 74, 63, 42, 45, 57, 16, 20, 19, 52.

(6 Marks)

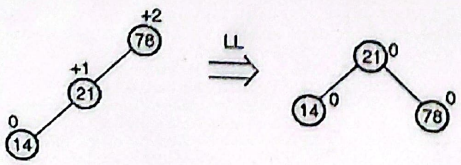
Ans. :

Insert 78 :

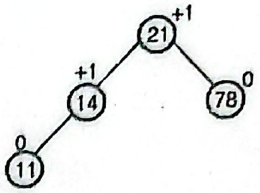
Insert 21 :



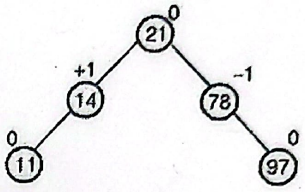
Insert 14 :



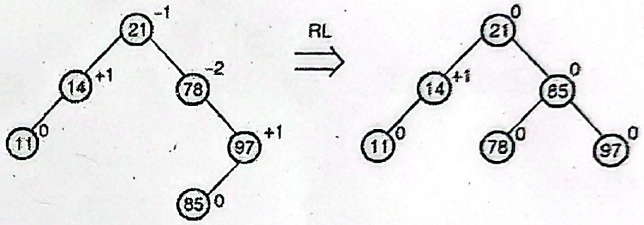
Insert 11 :



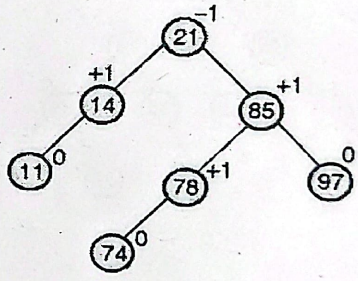
Insert 97 :



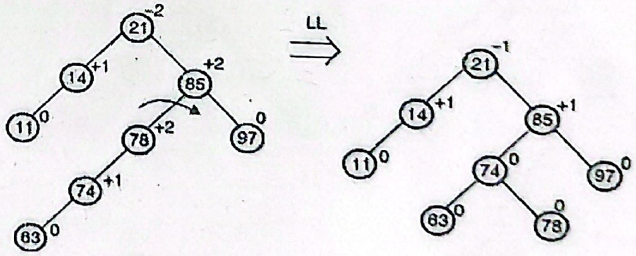
Insert 85 :



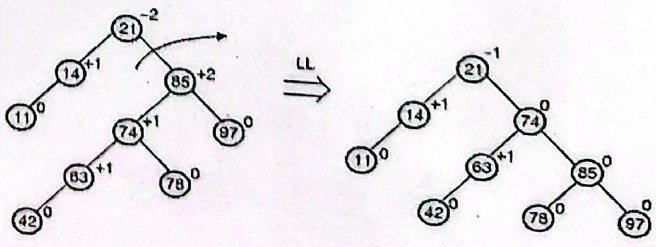
Insert 74 :



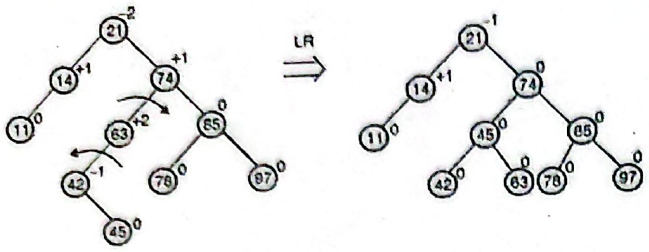
Insert 63 :



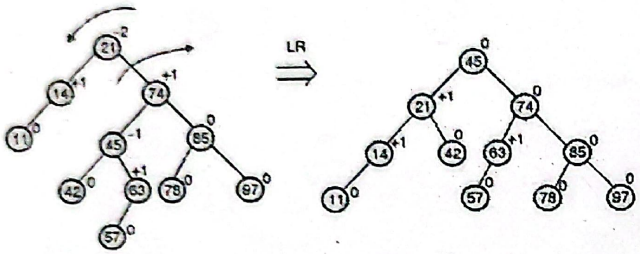
Insert 42 :



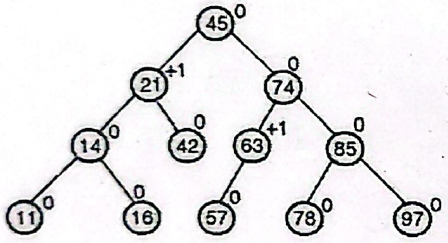
Insert 45 :



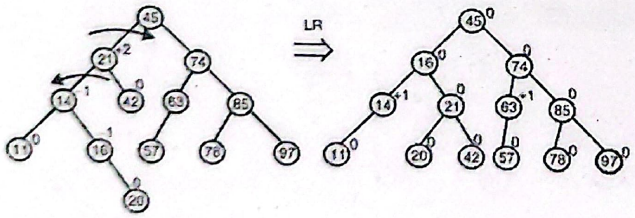
Insert 57 :



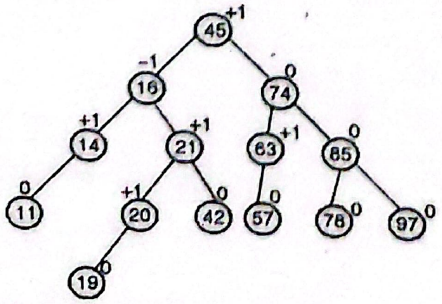
Insert 16 :



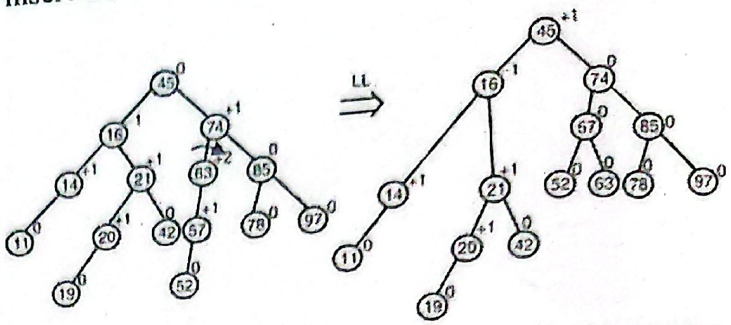
Insert 20 :



Insert 19 :



Insert 52 :



Q.13 Construct the AVL tree for the following data by inserting each data item one at a time. :

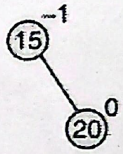
15, 20, 24, 10, 13, 7, 30, 36, 25

SPPU - May 14, 6 Marks

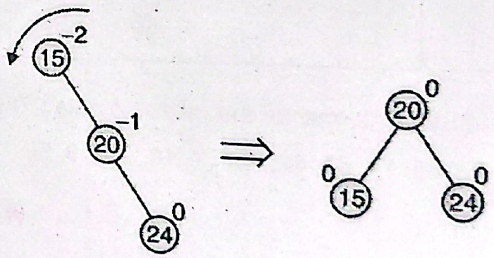
Ans. :

Insert 15 :

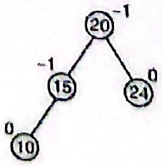
Insert 20 :



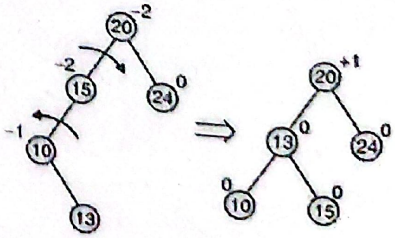
Insert 24 :



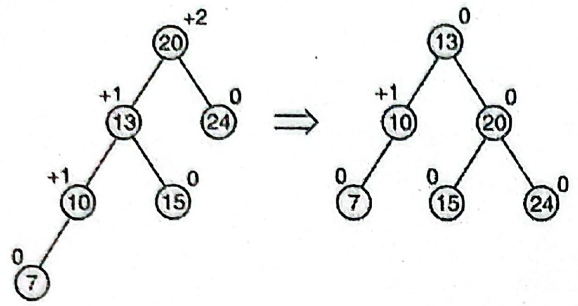
Insert 10 :



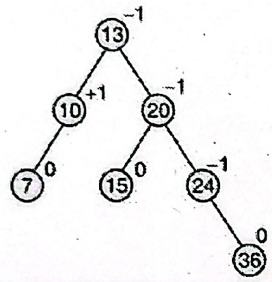
Insert 13 :



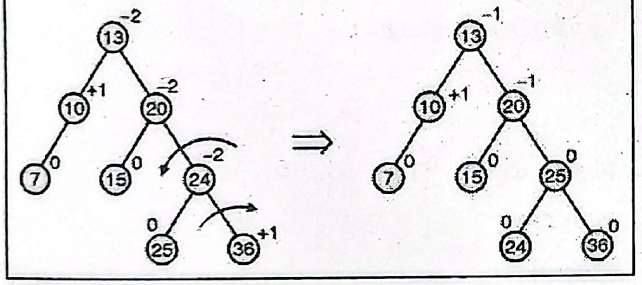
Insert 7 :



Insert 36 :



Insert 25 :



Q.14 Write non-recursive functions for insertion and deletion for AVL tree. **(6 Marks)**

Ans. :

```
/*Non-recursive insertion and deletion into an AVL */
#include<conio.h>
#include<stdio.h>
#include<iostream.h>
struct node
{ int data;
  node *left,*right;
  int ht;
  node *father;
};
int height(node *T)
{
    int lh,rh;
    if(T==NULL)
        return(0);
    if(T->left==NULL)
        lh=0;
    else
        lh=1+T->left->ht;
    if(T->right==NULL)
        rh=0;
    else
        rh=1+T->right->ht;
    if(lh>rh)
        return(lh);
    return(rh);
}
node * rotateright(node *x)
{
    node *y;
    y=x->left;
    x->left=y->right;
    y->right=x;
    x->ht=height(x);
    y->ht=height(y);
    return(y);
}
node * rotateleft(node *x)
{
    node *y;
    y=x->right;
    x->right=y->left;
    y->left=x;
    x->ht=height(x);
    y->ht=height(y);
    return(y);
```

```
}
node * RR(node *T)
{
    T=rotateleft(T);
    return(T);
}
node * LL(node *T)
{
    T=rotateright(T);
    return(T);
}
node * LR(node *T)
{
    T->left=rotateleft(T->left);
    T=rotateright(T);
    return(T);
}
node * RL(node *T)
{
    T->right=rotateright(T->right);
    T=rotateleft(T);
    return(T);
}
int BF(node *T)
{
    int lh,rh;
    if(T==NULL)
    return(0);
    if(T->left==NULL)
        lh=0;
    else
        lh=1+T->left->ht;
    if(T->right==NULL)
        rh=0;
    else
        rh=1+T->right->ht;
    return(lh-rh);
}
void rebalance(node **p)
  { while((*p)!=NULL)
    { (*p)->ht=height(*p);
      if(BF(*p)==2)
        if(BF((*p)->left)==1)
          (*p)=LL(*p);
        else
          (*p)=LR(*p);
      else
        if(BF(*p)==-2)
          if(BF((*p)->right)==-1)
            (*p)=RR(*p);
          else
            (*p)=RL(*p);

      p=&((*p)->father);
    }
  }
void insert(node **T,int x)
{  node **p;
   node *q;
    if(*T==NULL)
    {
        (*T)=new node;
        (*T)->data=x;
        (*T)->left=NULL;
        (*T)->right=NULL;
        (*T)->ht=0;
        (*T)->father=NULL;
    }
    else
     {  p=T;
        while(*p!=NULL)
        { if(x>(*p)->data  && (*p)->right==NULL)
```

```
        {q=new node;
        q->data=x;q->left=q->right=NULL;
        q->ht=0;q->father=(*p);
        (*p)->right=q;
        rebalance(p);
        break;
        }
       if(x<(*p)->data  && (*p)->left==NULL)
        {q=new node;
        q->data=x;q->left=q->right=NULL;
        q->ht=0;q->father=(*p);
        (*p)->left=q;
        rebalance(p);
        break;
        }
       if(x > (*p)->data)
        p=&((*p)->right);
       else
        p=&((*p)->left);
      }
   }
}
void Delete(node **t,int x)
{node **p,*q;
 p=t;
 while(*p!=NULL && (*p)->data!=x)
  if(x>(*p)->data)
   p=&((*p)->right);
  else
   p=&((*p)->left);
  if(*p!=NULL)
   {q=(*p)->father;
   if((*p)->left==NULL && (*p)->right==NULL)
    *p=NULL;
   else
    if((*p)->left==NULL)
     *p=(*p)->right;
    else
     if((*p)->right==NULL)
      *p=(*p)->left;
     else
      { node *r=*p;
       p=&((*p)->right);
       while((*p)->left !=NULL)
        p=&((*p)->left);
       r->data=(*p)->data;
       q=(*p)->father;
       *p=(*p)->right;
      }
    rebalance(&q);
   }
}
```

Q.15 Explain with example : Splay Tree.

SPPU - May 19, Dec. 19, 6 Marks

Ans.:

Splay Tree

- Splay tree is a binary search tree.
- In a splay tree, M consecutive operations can be performed in O (M log N) time.
- A single operation may require O(N) time but average time to perform M operations will need O (M Log N) time.
- When a node is accessed, it is moved to the top through a set of operations known as splaying. Splaying technique is similar to rotation in an AVL tree. This will make the future access of the node cheaper.
- Unlike AVL tree, splay trees do not have the requirement of storing **Balance Factor** of every node. This saves the space and simplifies algorithm to a great extent.

There are two standard techniques of splaying.

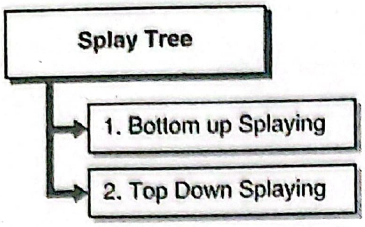


Fig. 4.12 : Techniques of splaying

1. Bottom up Splaying

Idea behind bottom up splaying is explained below :

- Rotation is performed bottom up along the access path.
- Let X be a (non root) node on the access path at which we are rotating.

a) If the parent of X is the root of the tree, rotate X and the parent of X. This will be the last rotation required.

b) If X has both a parent (P) and Grand parent (G) then like an AVL tree there could be four cases. These four cases are :

1. X is a left child and P is a left child.
2. X is a left child and P is right child
3. X is a right child and P is a left child
4. X is a right child and P is a right child.

Top Down Splaying

When an item X is inserted as a leaf, a series of tree rotations brings X at the root. These rotations are known as splaying. A splay is also performed during searches, and if an item is not found, a splay is performed on the last node on the access path.

- A top down traversal is performed to locate the leaf node.
- Splaying is done using bottom up traversal.
- This can be done by storing the access path, during top down traversal on a stack.

Top down splaying is based on splaying on the initial traversal path. A stack is not required to save the traversal path.

At any point in the access :

1. A current node X that is the root of its sub tree and represented as the "middle" tree.
2. Tree L stores nodes in the tree T that are less than X, but not in the X's sub tree.
3. Tree R stores nodes in the tree T that are larger than X, but not in X's sub tree.
4. Initially, X is the root of T, and L and R are empty.

Q.16 Perform splaying on the tree given below element 19 is accessed. **(6 Marks)**

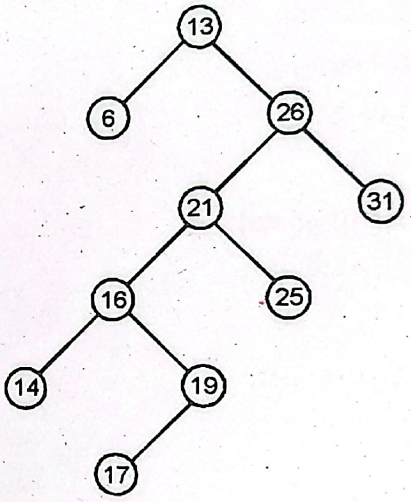
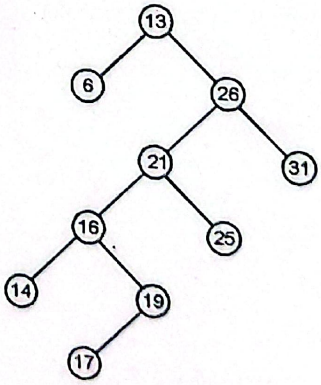


Fig. 4.13

Ans. :

L	Centre	R
Empty		Empty

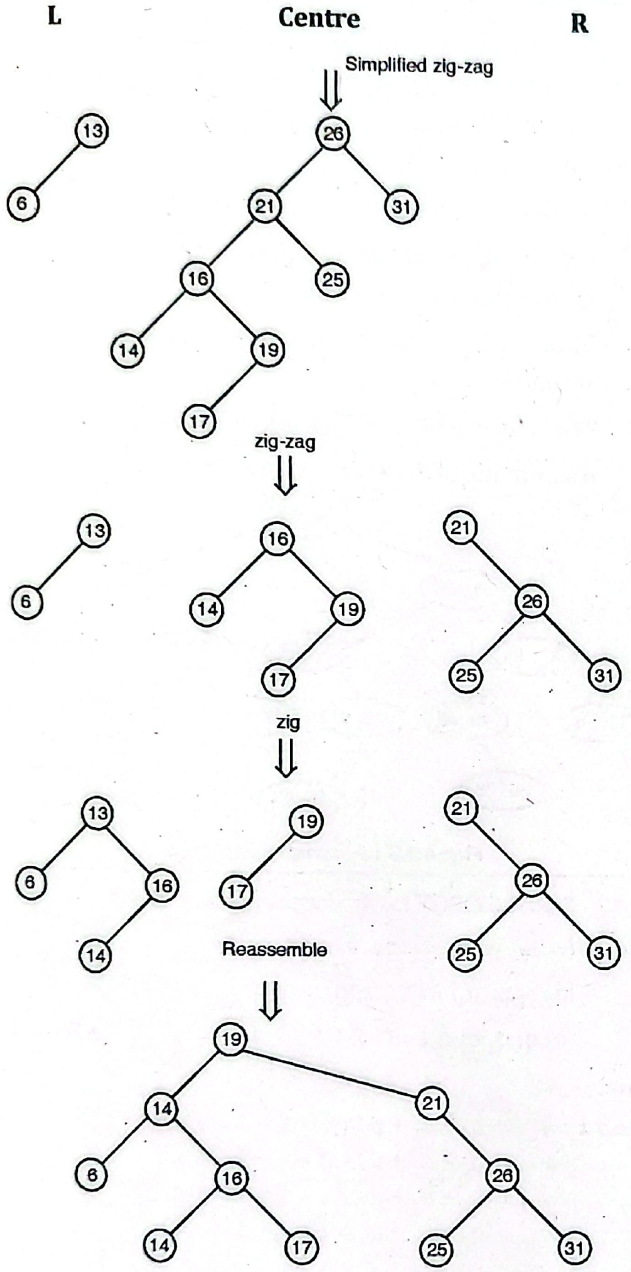


Fig. 4.13(a) : Steps in top – down splay

Q.17 Explain with example : Red Black Tree.

SPPU - May 19, Dec. 19, 6 Marks

Ans.:

Red-Black Tree

Red – black tree is an alternative to the AVL tree. A red black tree is a binary search tree with the following coloring properties :

1. Every node is coloured either red or black.
2. The root is black.
3. If a node is red, its children must be black
4. Every path from a node to a NULL pointer must contain the same number of black nodes.

A sample red-black tree is shown in Fig. 4.14(a).

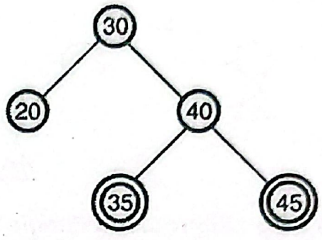


Fig. 4.14(a) : A sample red black tree

Red nodes are shown with double circles. Black nodes are shown with single circle.

- Height of a red black tree is at most 2 log (N + 1).
- Searching of a node can be done in logarithmic time.
- Unlike AVL tree, non – recursive algorithm for various operations can be done effortlessly.

Insertion of a node in a red – black tree

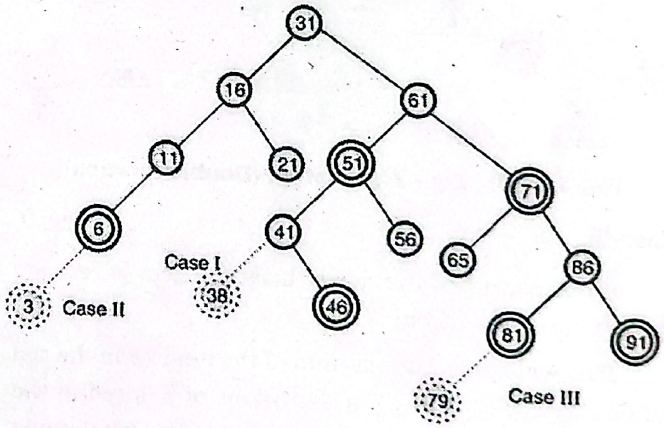


Fig. 4.14(b)

The various cases in insertion with the help of the red black tree of Fig. 4.14(b).

Case I

If the parent of the newly inserted item is black.

It the parent of the newly inserted item is black then the newly inserted item is colored red. This insertion is trivial. If the item 38 is to be inserted in the red black tree of Fig. 4.14(b) then the element 38 is coloured red.

Case II

If the parent of the newly inserted item is red and the sibling of the parent is black (NULL nodes are treated as black). This will apply for insertion of 3 in the red black tree of Fig. 4.14(b).

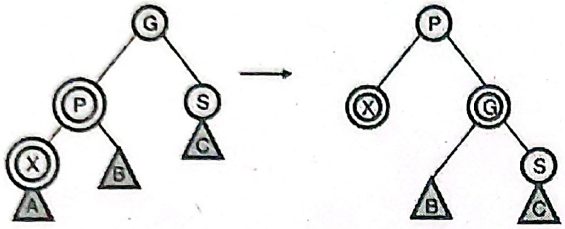


Fig. 4.14(c) : Zig - Zig rotation (Single rotation)

Let X be the newly added leaf, P be its parent and S be the sibling of the parent P. G be the father of P. X and P are red in this case. X,P,G can form either a Zig – Zig chain or Zig – Zag chain. Splaying can be used to handle this situation. Splaying is shown in the Fig. 4.14(c) and Fig. 4.14(d).

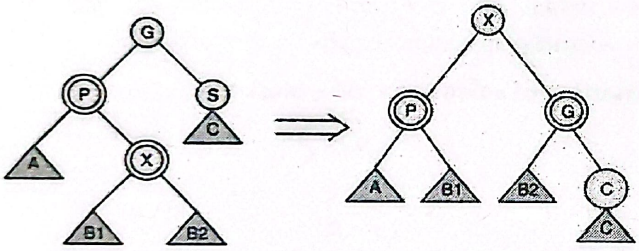


Fig. 4.14(d) : Zig - Zig rotation (Double rotation)

Case III

If the parent of the newly inserted item is red and the sibling of the parent is red.

This will apply for insertion of the item 79 in the red black tree of Fig. 4.14(a). If the colour of S is red in the sub tree of Fig. 4.14(a) then there is a single black node on the path from the sub tree's root to C. After rotation, there must still be only one black node. After rotation, there will be consecutive red nodes (G and S). To colour both S and the subtree's new root red, and G black. If the great grand parent is also red then can continue the process up toward the root, until we no longer have two consecutive red nodes.

Q.18 Explain with example : KD Tree.

SPPU - May 19, 6 Marks

Ans.: K-dimensional Tree

A k-d tree, also known as k-dimensional tree, is a data structure for arranging some data in a space with k dimensions. A k-d tree is a special case of binary space partitioning trees.

A 2-d search tree has the following properties :

1. Branching on odd level is done with respect is done with respect to the first key.
2. Branching on even levels is done with respect to the second key.
3. The root is arbitrarily chosen to be an odd level.

An example of 2-d tree is shown in the Fig. 4.15.

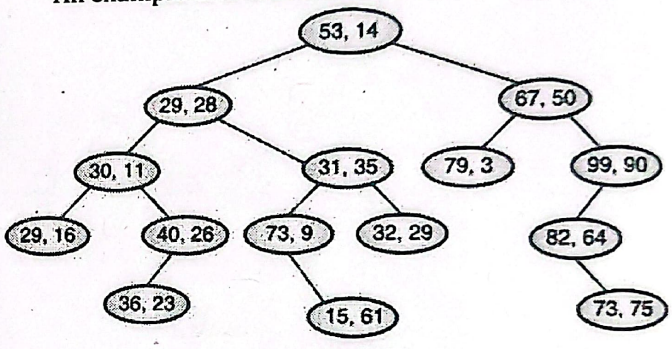


Fig. 4.15 : A sample 2-d tree

Q.19 Find the OBST for the following data : N = 4

(w_1,w_2,w_3,w_4) = (do, if, read, while)

$(p_1,p_2,p_3,p_4) = (1,3,1,3)$

$(q_0,q_1,q_2,q_3,q_4) = (1,2,1,1,3)$. **(6 Marks)**

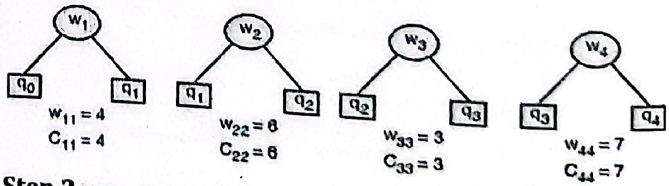
Ans. :

Step 1 : $w_{11} = q_0 + q_1 + p_1 = 1 + 1 + 2 = 4 = c_{11}$

$w_{22} = q_1 + q_2 + p_2 = 2 + 1 + 3 = 6 = c_{22}$

$w_{33} = q_2 + q_3 + p_3 = 1 + 1 + 1 = 3 = c_{33}$

$w_{44} = q_3 + q_4 + p_4 = 1 + 3 + 3 = 7 = c_{44}$



Step 2 : $w_{12} = w_{11} + q_2 + p_2 = 4 + 1 + 3 = 8$

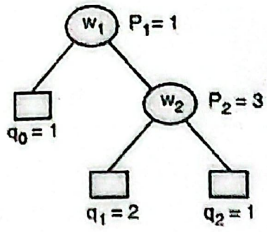
$$c_{12} = \min_{1 \le k \le 2} \left(C_{1,k-1} + C_{k+1,2}\right) + w_{12}$$

$$= \min\left((C_{10} + C_{22}), (C_{11} + C_{32})\right) + w_{12}$$

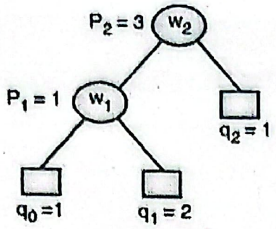
$$= \min(c_{22}, c_{11}) + w_{12}$$

$$= \min(6,4) + 8 = 4 + 8 = 12$$

$r_{12} = 2$



$w_{12} = 8$
Cost = 14



$w_{12} = 8$
Cost = 12

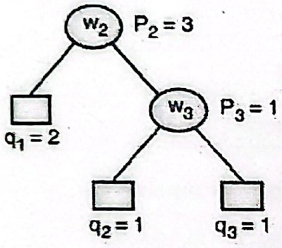
$w_{23} = w_{22} + q_3 + p_3 = 6 + 1 + 1 = 8$

$c_{23} = \min_{2 \le k \le 3} \left(C_{2,k-1} + C_{k+1,3}\right) + w_{23}$

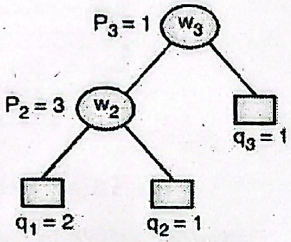
$= \min \left((C_{21} + C_{33}), (C_{22} + C_{43})\right) + w_{23}$

$= \min (3,6) + 8 = 3 + 8 = 11$

$r_{23} = 2$



$w_{12} = 8$
Cost = 11



$w_{12} = 8$
Cost = 14

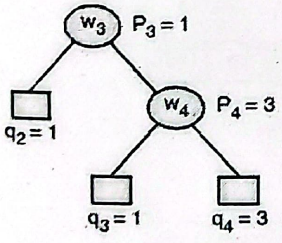
$w_{34} = w_{33} + q_4 + p_4 = 3 + 3 + 3 = 9$

$c_{34} = \min_{3 \le k \le 4} \left(C_{3,k-1} + C_{k+1,4}\right) + w_{34}$

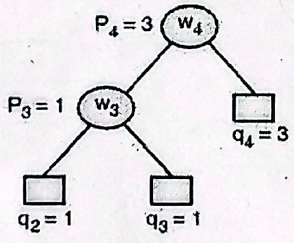
$= \min \left((C_{32} + C_{44}), (C_{33} + C_{54})\right) + w_{34}$

$= \min (7,3) + 9 = 3 + 9 = 12$

$r_{34} = 4$



$w_{34} = 9$
Cost = 16



$w_{34} = 9$
Cost = 12

Step 3 :

$w_{13} = w_{12} + p_3 + q_3 = 8 + 1 + 1 = 10$

$c_{13} = \min_{1 \le k \le 3} \left(C_{1,k-1} + C_{k+1,3}\right) + w_{13}$

$= \min \left((C_{10} + C_{23}), (C_{11} + C_{33}), (C_{12} + C_{43})\right) + w_{13}$

$c_{13} = \min (11,7,12) + 10 = 7 + 10 = 17$

$r_{13} = 2$

$w_{2,4} = w_{2,3} + p_4 + q_4 = 8 + 3 + 3 = 14$

$c_{2,4} = \min_{2 \le k \le 4} \left(C_{2,k-1} + C_{k+1,4}\right) + w_{24}$

$= \min \left((C_{21} + C_{34}), (C_{22} + C_{44}), (C_{23} + C_{54})\right) + w_{24}$

$= \min (12,13,11) + 14 = 25$

$r_{24} = 4$

Step 4 :

$w_{14} = w_{13} + p_4 + q_4 = 10 + 3 + 3 = 16$

$c_{14} = \min_{1 \le k \le 4} \left(C_{1,k-1} + C_{k+1,4}\right) + w_{14}$

$= \min \left((C_{10}+C_{24}),(C_{11}+C_{34}),(C_{12}+C_{44}),(C_{13}+C_{54})\right)+w_{14}$

$= \min \{(0+25),(4+12),(12+7),(17+7)\} + w_{14}$

$= \min (25,16,19,24) + 16 = 16 + 16 = 32$

$r_{1,4} = 2$

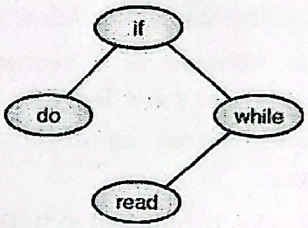


Fig. 4.16(a) : OBST

$w_{11} = 4$ $c_{11} = 4$ $r_{11} = 1$	$w_{22} = 6$ $c_{22} = 6$ $r_{22} = 2$	$w_{33} = 3$ $c_{33} = 3$ $r_{33} = 3$	$w_{44} = 7$ $C_{44} = 7$ $r_{44} = 4$
$w_{12} = 8$ $c_{12} = 12$ $r_{12} = 2$	$w_{23} = 8$ $c_{23} = 11$ $r_{23} = 2$	$w_{34} = 9$ $c_{34} = 12$ $r_{34} = 4$	
$w_{13} = 10$ $c_{13} = 17$ $r_{13} = 2$	$w_{24} = 14$ $c_{24} = 25$ $r_{24} = 4$		
$w_{14} = 16$ $c_{14} = 32$ $r_{14} = 2$			

Fig. 4.16(b) : Computation of w_{14}, c_{14}, r_{14}

❑❑❑

Unit V : Indexing & Multiway Trees

Q.1 Give advantages of indexing. **(3 Marks)**

Ans.:

Advantages of Indexing

1. Sequential file can be searched effectively on ordering key. When it is necessary to search for a record on the basis of some other attribute than the ordering key field, sequential file representation is inadequate.

   Multiple indexes can be maintained for each type of field to be used for searching. Thus, indexing provides much better flexibility.

2. An index file usually requires less storage space than the main file. A binary search on sequential file will require accessing of more blocks. This can be explained with the help of the following example.

   Consider the example of a sequential file with r = 1024 records of fixed length with record size R = 128 bytes stored on disk with block size B = 2048 bytes.

   Number of blocks 'b' required to store the file

   $$= \frac{1024 \times 128}{2048} = 64$$

   Number of block accesses for searching a record $= \log_2^{64} \approx 6$.

   Suppose, to construct an index on a key field that is V = 4 bytes long and the block pointer is P = 4 bytes long. A record of an index file is of the form $<V_i, P_i>$ and it will need 8 bytes per entry.

   Total Number of index entries = 1024

   Number of blocks needed for the index is hence bi

   $$= \frac{1024 \times 8}{2048} = 4 \text{ blocks}$$

   Binary search on the index will require $\log_2^4 = 2$ block accesses.

3. With indexing, new records can be added at the end of the main file. It will not require movement of records as in the case of sequential file. Updation of index file requires fewer block accesses compare to sequential file.

Q.2 List and explain any two Indexing Techniques. **(5 Marks)**

Ans.: Indexing Techniques

There are several techniques for indexing, including :

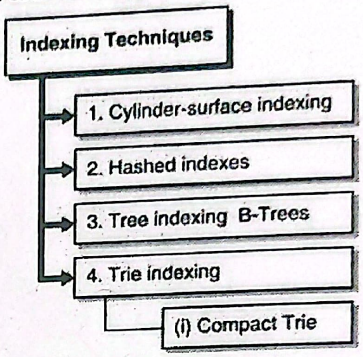


Fig. 5.1(a) : Indexing techniques

1. Cylinder Surface Indexing

It is useful only for the primary key index of sequentially ordered file. Such files are known as ISAM files. ISAM file is spread over several cylinders. Cylinder 0 is used to maintain the cylinder index. Track index is maintained on track 0 of every cylinder. The cylinder index has an entry for each cylinder of the file and the largest entry on that cylinder. The typical cylinder index is shown in Fig. 5.1(a).

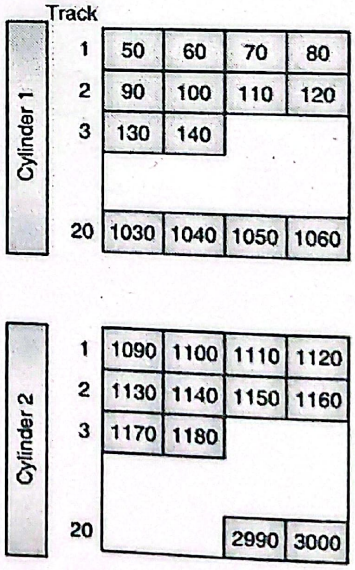


Fig. 5.1(b) : Record storage

13	1750	14	1850	15	2300	16	3000	...
Cy1	key	Cy1	key	Cy1	key	Cy1	Key	

Fig. 5.1(b) : A typical cylinder index

1	1800	2	1880	3	1990	...
track	key	Track	key	track	key	

Fig. 5.1(c) : A typical track index

A typical track index is shown in Fig. 5.1(c).

- ISAM is good for static tables because there are usually fewer index levels than B-tree.
- Because the whole structure is ordered to a large extent, partial and range based retrievals can often benefit from the use of this type of index.
- In general there are fewer disk I/O required to access data, provided there is no overflow.
- Overflow can be a real problem in volatile tables.

2. Hashed Indexes

- Hashed indexes are same as those discussed for hash tables. Same hash functions and overflow handling techniques are used. Hash table must be organised properly to minimise number of disk accesses.
- Records of a file are divided among buckets. A bucket is either one disk block or clusture of contiguous blocks. A hashing function maps a key into a bucket number. The buckets are numbered 0, 1, 2, ... b – 1. A hash function f maps each key into one of the integers o through b – 1. If x is a key, f (x) is the bucket number that contains the record with key x. The blocks making up each bucket could either be contiguous blocks or they can be chained together in a linked list.
- Translation of bucket number to disk block address is done with the help of bucket directory. It gives the address of the first block of the chained block in a linked list. This arrangement is shown in the Fig. 5.1(d).

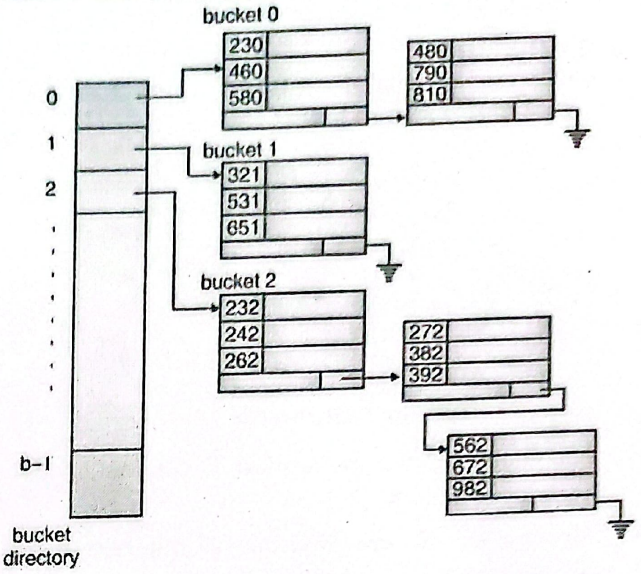


Fig. 5.1(d) : Hashing with buckets of chained blocks

Q.3 Explain with example : Trie Tree.

SPPU - May 19, 6 Marks

Ans.:

Trie Indexing

- Trie indexing is useful when key values are of varying size.
- A trie is a tree of degree $P \geq 2$.
- Tries are useful for storing words as a string of characters.
- In a trie, each path from the root to a leaf corresponds to one word.
- Node of the trie correspond to the prefixed of words.
- A sample trie of words {THEN, THIS, SIN} is shown in the Fig. 5.2(a).

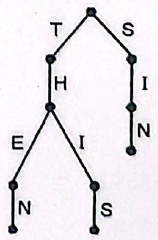


Fig. 5.2(a) : A sample trie

- To avoid confusion between words like THE and THEN, a special end marker symbol '\0' is added at the end of each word.

- In Fig. 5.2(b), there is a trie representing the set of words {THE, THEN, TIN, SIN, THIN, SING}. Endmarker symbol '\0' is added to end each word.

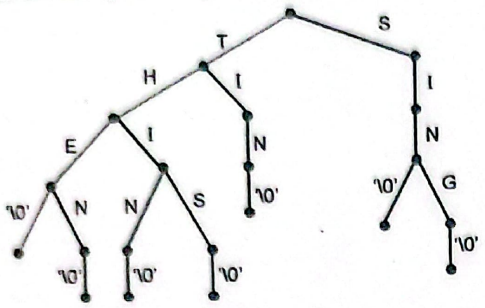


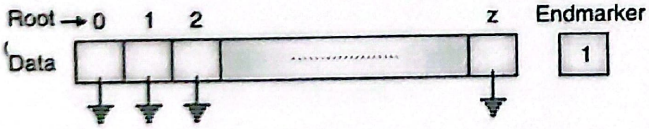
Fig. 5.2(b) : A trie

- Each node of a trie has at most 27 children one for each letter and for '\0'.
- Most nodes will have fewer than 27 children.
- A leaf node reached by an edge labelled '\0' cannot have any children and it need not be there.

Structure of a trie node

```
Struct trie
{  trie *data [26] ;
int endmarker ; //endmarke = 1 indicates end of a word.
};
data [o] is for character A
:
data [25] is for character t
```

An empty trie is represented by a tree of single node. Data pointers are set to NULL.



C++ function to create an empty trie

```
root = new trie ;
root → endmarker = 1 ;
for (i=0; i< 26; i++)
root → data [i] = NULL ;
```

Insertion

A character word can be added recursively to a trie. Trace the existing prefix of a word in the trie and remaining characters can be added to the trie.

```
Insert (trie *head, char *x)
{  int i ;
   if (*x = = '\o' )
   head → endmarker = 1 ;  // end of word
else
   {  head → endmarker = 0;
      if (head → data [*x – 65] = = NULL)
      {  head → data [*x - 65] = new trie ;
         head = head → data [*x - 65];
/* 65 has been subtracted from x to map it to an integer
location */
      for (i=0 ; i < 26 ; i++)
         head → data [i] = NULL ;
         insert (head, x+1);
      }
   }
}
```

Deletion

Deletion of a word from trie can be implemented with the help of a stack.

As traverse the trie to locate the word, address of each node is pushed on top of the stack.

Now, the chain leading to node containing the current word is deleted.

```
void Delete (trie *head, char *x)
{  stack S ;
   while (*x ! = '\0')
   {  S.push (head → data [*x - 65])
      head = head → data [*x – 65] ;
      x++;
   }
      while (! S.empty ()) && chain (head = S.pop)
      delete head;
}
   int chain (trie * head)
{  n = 0;
```

```
for (i=0; i < 26; i++)
if (head → data [i] ! = NULL )
        n++ ;
if (n = = 0 || n = = 1)
return 1 ; // part of chain
else
return 0 ;
}
```

Function chain ( ), checks whether the node is part of the chain leading to terminal node of a trie.

Q.4 What is B tree ? SPPU - Dec.19, 4 Marks

Ans.:

B-Trees

B-tree is another very popular search tree. The node in a binary tree like AVL tree contains only one record. AVL tree is commonly stored in primary memory. In database application, where huge volume of data is handled, the search tree cannot be accommodated in primary memory. B-trees are primarily meant for secondary storage.

A B-tree is a M-way tree. An M-way tree can have maximum of M children.

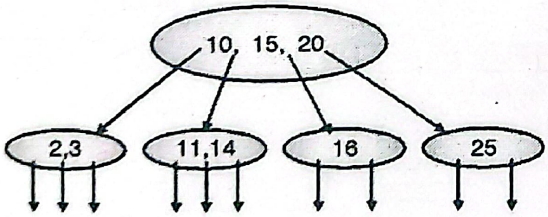


Fig. 5.3(a) : An example of 4-way tree

An M-way tree contains multiple keys in a node. This leads to reduction in overall height of the tree. If a node of M-way tree holds K number of keys then it will have K+1 children.

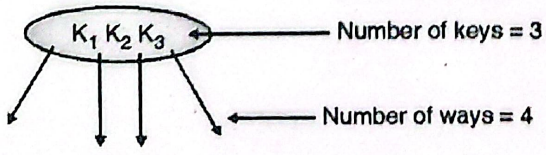
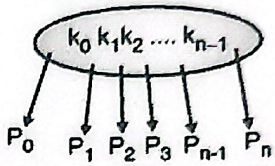


Fig. 5.3(b) : An M-way tree with 3 keys and 4 children

Definition

A B-tree of order M is a M-way search tree with the following properties :

1. The root can have 1 to M–1 keys.
2. All nodes (except the root) have between [(M–1)/2] and M–1 keys.
3. All leaves are at the same depth.
4. If a node has t number of children then it must have (t – 1) number of keys.
5. Keys of a node are stored in ascending order.



6. K_0, K_1, K_2 ... K_{n-1} are the keys stored in the node. Subtrees are pointed by P_0, P_1 ... P_n.

   then $K_0 \geq$ all keys of the subtree P_0

   $K_1 \geq$ all keys of the subtree P_1

   ⋮

   $K_{n-1} \geq$ all keys of the subtree P_{n-1}

   $K_{n-1} <$ all keys of the subtree P_n.

An example of B-tree of order 4 is shown in Fig. 5.3(c).

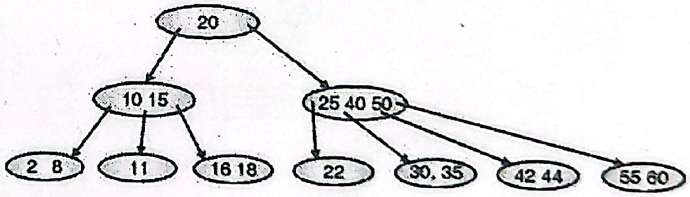
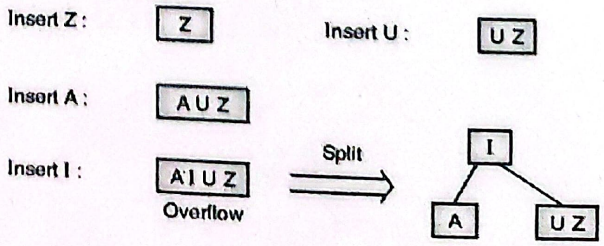


Fig. 5.3(c)

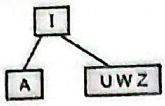
Q.5 Consider the following sequence of nodes and show the growth of the B-tree of order – 4

Z, U, A, I, W, L, P, X, C, J, D, M, T, B, Q, E, H, S, K, N, R, G, Y, F, O, Y. **(6 Marks)**

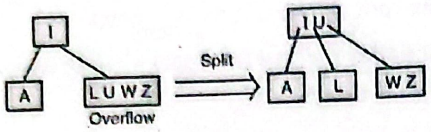
Ans. :



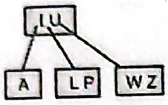
Insert W :



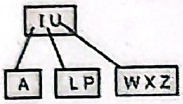
Insert L :



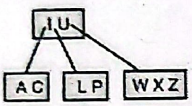
Insert P :



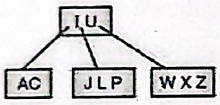
Insert X :



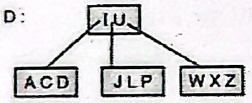
Insert C :



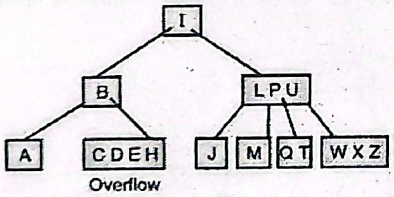
Insert J :

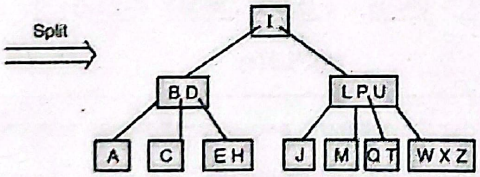


Insert D :

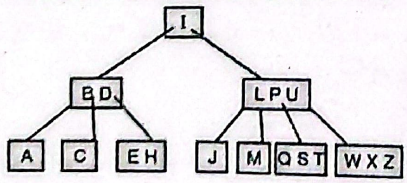


Insert H :

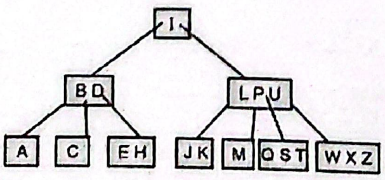




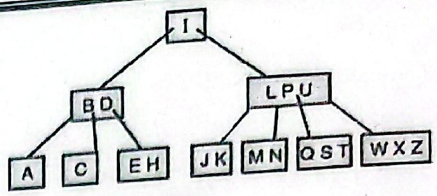
Insert S :



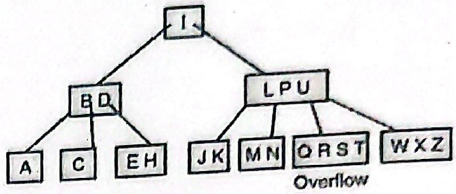
Insert K :

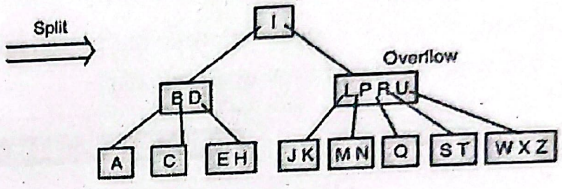


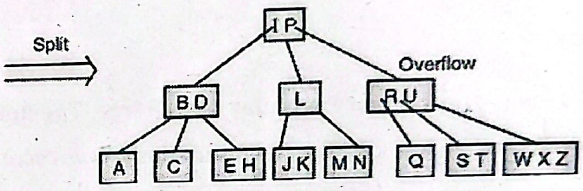
Insert N :



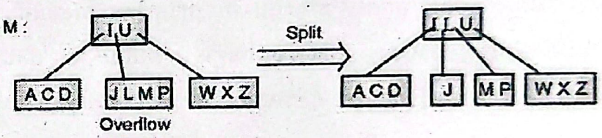
Insert R :



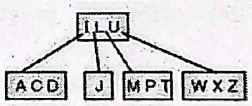




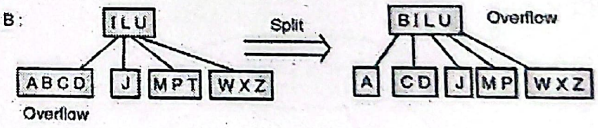
Insert M :

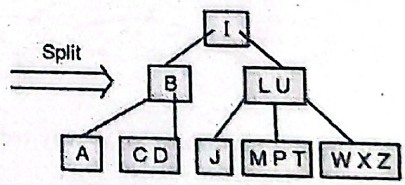


Insert T :

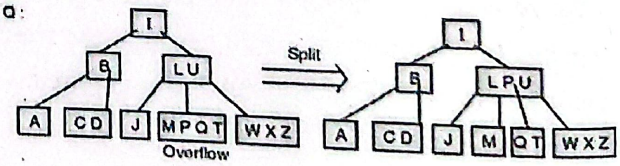


Insert B :

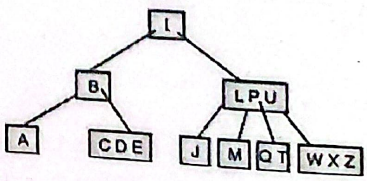


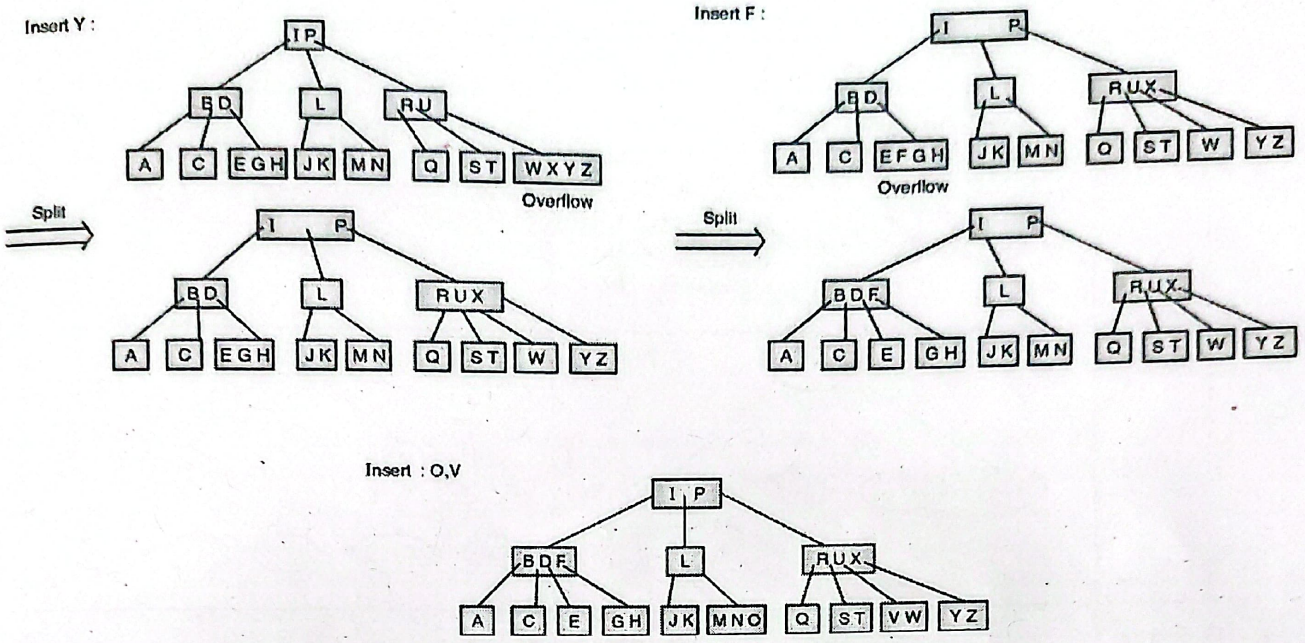


Insert Q :



Insert E :

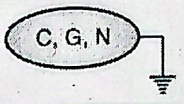
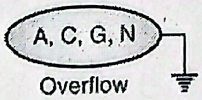
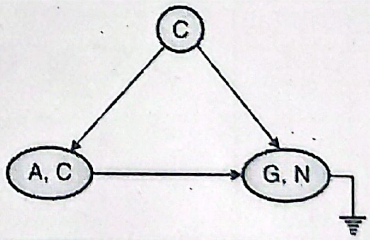
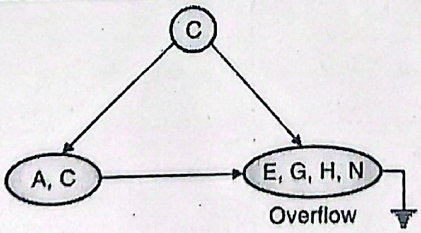
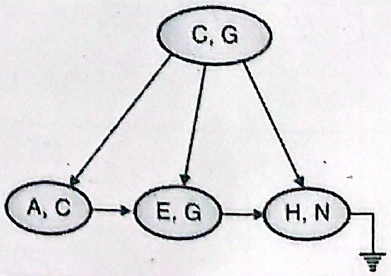


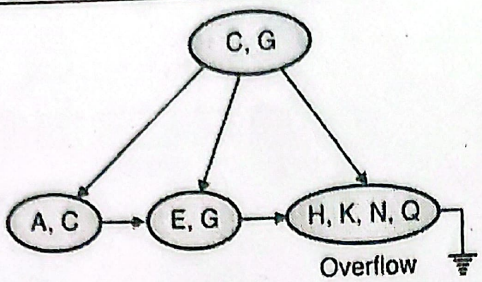
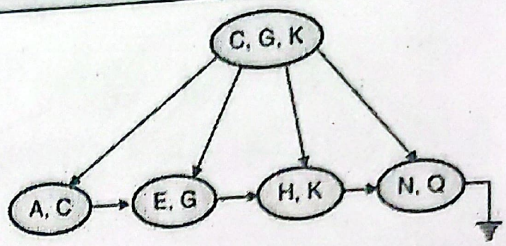
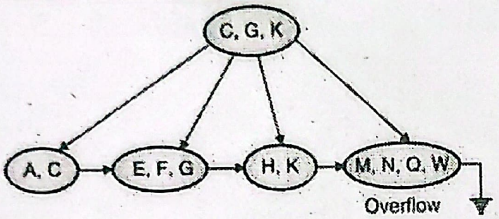
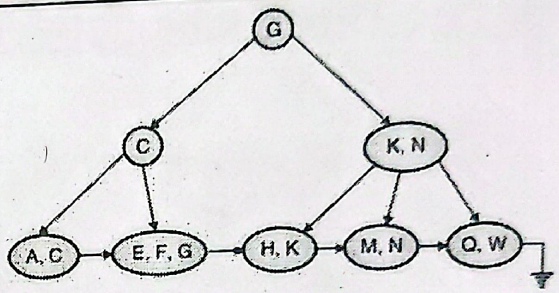
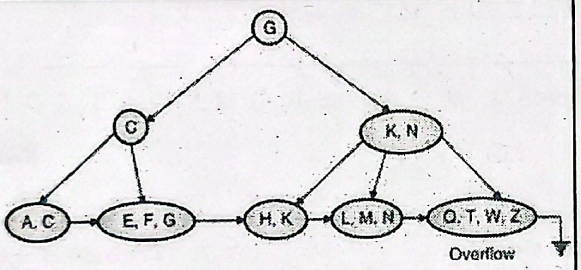
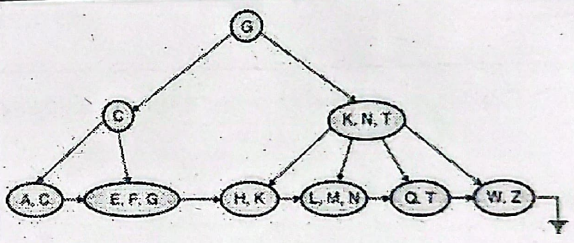
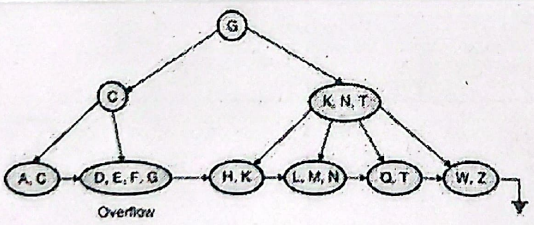
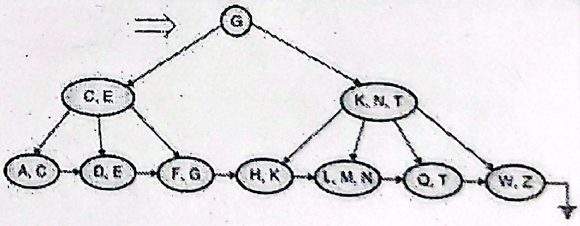
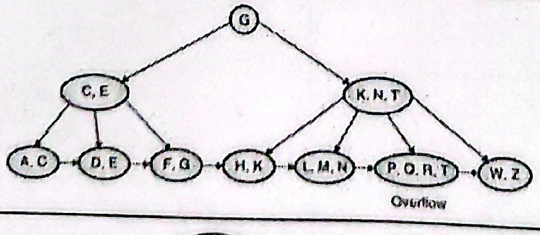
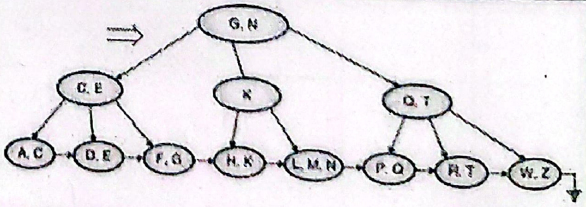
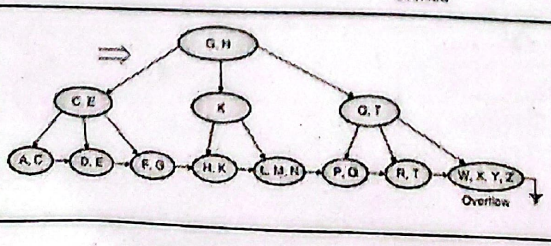
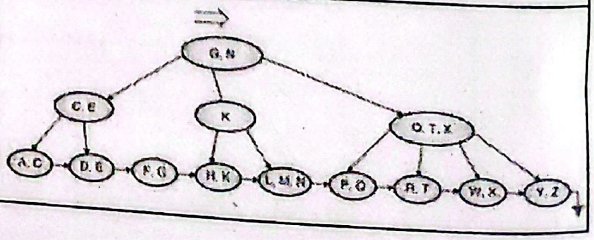


Q.6 Construct B+ tree of order 4 for the following data : C, N, G, A, H, E, K, Q, M, F, W, L, T, Z, D, P, R, X, Y.

SPPU - Dec. 19, 6 Marks

Ans. :

Sr. No.	Insert	Tree after insertion	Tree after splitting
1.	C, N, G		
2.	A		
3.	H, E		

Sr. No.	Insert	Tree after insertion	Tree after splitting
4.	K, Q		
5.	M, F, W		
6.	L, T, Z		
7.	D		
8.	P, R		
9.	X, Y		

Q.7 Consider the following 5 way B tree : Delete root node i.e. a node with key value 16 from the above tree and redraw the tree by maintaining its B tree property.

SPPU - Dec. 14, 6 Marks

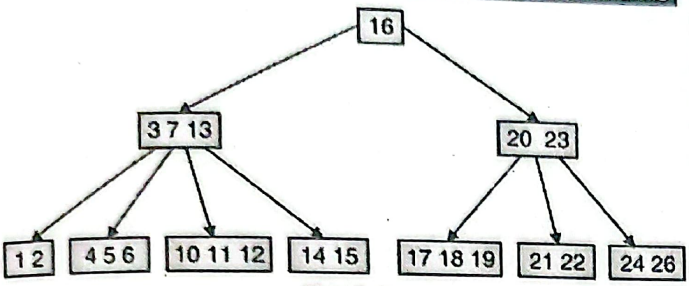
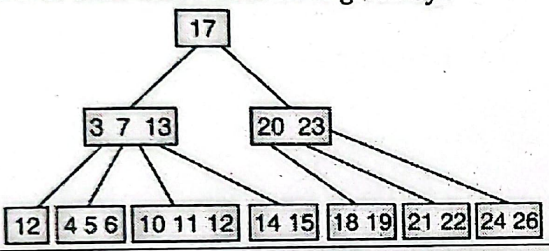


Fig. 5.4

Ans. : To delete 16, the smallest value 16 in the right subtree can be moved to the root node.

∴ Tree after deletion of 16 is given by :



Q.8 Explain the delete operation in B tree with example.

SPPU - Dec.19, 5 Marks

Ans.: Deleting a Value from a B-tree

Recall our deletion algorithm for binary search trees; if the value to be deleted is in a node of degree 2, we would replace the value with the smallest value (inorder successor) in its right subtree and then delete the node in the right subtree containing the smallest value.

Use a similar strategy to delete a value from a B-tree. If the value to be deleted does not occur in a leaf, we replace it with the smallest value (successor) in its right subtree and then proceed to delete the value from the leaf node. For example, if we wish to delete 67 from the B-tree of Fig. 5.5(a), find the smallest in 67's right subtree, replace 67 with 68 and then delete the occurrence of 68 in the right subtree.

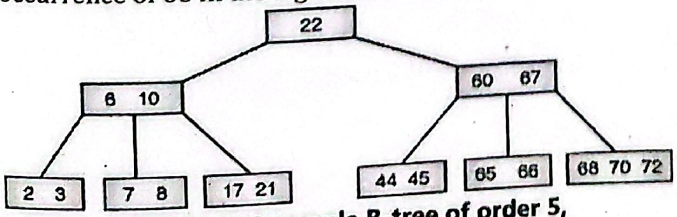


Fig. 5.5(a) : A sample B-tree of order 5, before deletion of the key with value 67

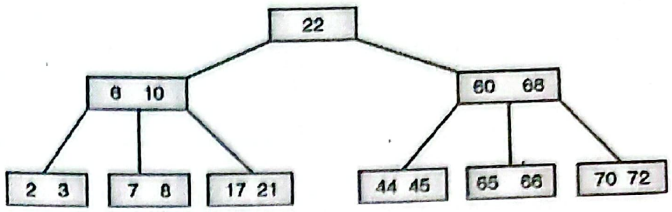


Fig. 5.5(b) : B-tree of Fig. 5.5(a) after deletion of the key with value 67

- If deletion of a key causes the node to have less than $\frac{M-1}{2}$ (for a M-way tree) keys, an underflow has occurred.
- If underflow does not occur, the deletion algorithm finishes. If it does occur, it must be fixed.

Strategy for fixing underflow

1. If the left neighbour contains more than $\frac{M-1}{2}$ keys, it can borrow a key from the left neighbour. The concept is shown in the Fig. 5.5(c).

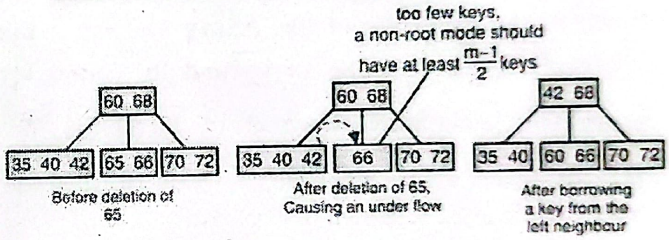
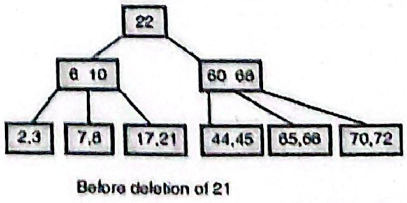


Fig. 5.5(c) : Borrowing a key from the left neighbour to fix an underflow

2. If the left neighbour contains $\frac{M-1}{2}$ keys and the right neighbor contains more than $\frac{M-1}{2}$ keys, it can borrow a key from the right neightbour.
3. If both left and right neighbours contain $\frac{M-1}{2}$ keys, we join together the current node and its neighbour, either left or right, to form a combined node and we must also include in the combined node the value in the parent node that is in-between these two nodes.



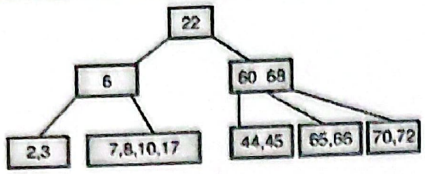


Fig. 5.5(d) : Node [7, 8] and the node [17] are merged along with their common parent 10

Suppose 21 is to be deleted from the B-tree of Fig. 5.5(d). Since it is a B-tree of order 5, every node except the root node must have at least $\frac{5-1}{2} = 2$ keys. If 21 is deleted from the leaf node [17, 21] then this node will not have minimum number of keys after deletion. Fig. 5.5(d) shows the situation of the tree before deletion of 21 and subsequent merging of this node with its left sibling.

After merging the node [6, 10] becomes [6] and does not have the required minimum number of keys, as shown in the right tree of Fig. 5.5(d). To correct this problem, the right sibling of [6] and its parent are merged with [6] to form a new node [6, 22, 60, 68] .

This node, now becomes the new root. The final tree after deletion of 21 is shown in Fig. 5.5(e).

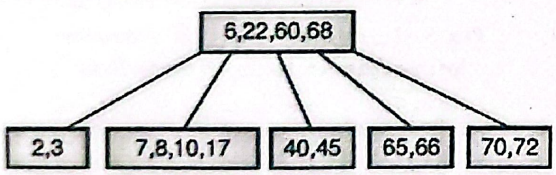


Fig. 5.5(e) : The B-tree after deletion of key 21 from B-tree of Fig. 5.5(d)

Q.9 Write C++ function for delete operation

SPPU - May 17, 7 Marks

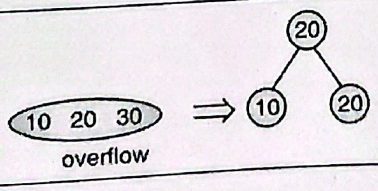
Ans.:

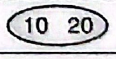
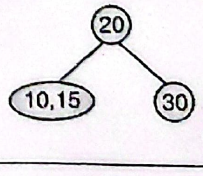
```
void btree::Delete(int x)
 {
   node *left,*right;
   pair *centre;
   node *p,*q;
   int i,j,centreindex;
   p=search(x);
   for(i=0;p->data[i].key != x; i++)

   // if p is not a leaf node then locate its successor in a
     leaf node,
   // replace x with its successor/predecessor and
delete it.
   if(!p->leafnode())
   {
      q=p->data[i].next;
      while(!q->leafnode())
         q=q->first;
      p->data[i].key=q->data[0].key;
      p=q;          //p points to leaf node
      x=q->data[0].key;
      i=0;
   }
   // if(p->leafnode()), p will always be a leaf node
   for(i=i+1;i<p->noofkeys;i++)
   p->data[i-1]=p->data[i];
   p->noofkeys--;
   while(1)
   {
      if(p->noofkeys >= mkeys/2 )
      return;
      if(p==root )
      if(p->noofkeys>0)
      return;
      else
      {
         root=p->first;
         return;
      }
       // otherwise
      q=p->father;
      if(q->first==p || q->data[0].next==p)
      {
         left=q->first;
         right=q->data[0].next;
         centre=&(q->data[0]);
```

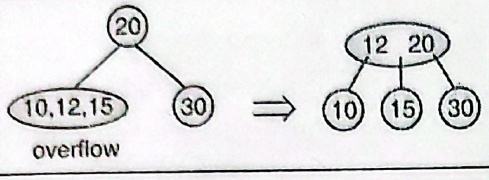
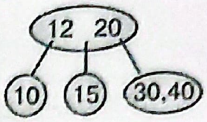
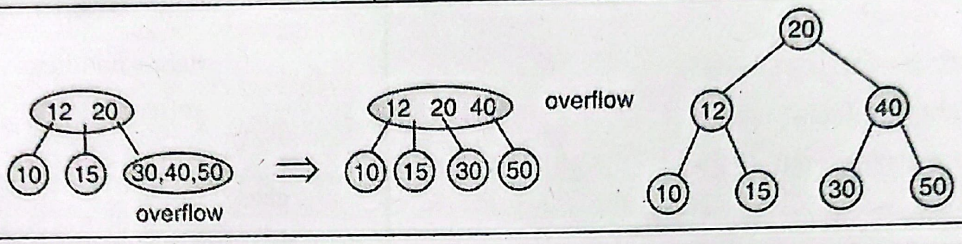
```
    centreindex=0;
}
else
{
    for(i=1;i<q->noofkeys;i++)
    if(q->data[i].next==p)
        break;
    left=q->data[i-1].next;
    right=q->data[i].next;
    centre=&(q->data[i]);
    centreindex=i;
}
//case 1 : left has one extra key, move a
      key from left
f(left->noofkeys > mkeys/2)
{
    for(i=right->noofkeys-1;i>=0;i--)
    right->data[i+1]=right->data[i];
    right->noofkeys ++;
    right->data[0].key=centre->key;
centre->key=left->data[left->noofkeys - 1].key;
left->noofkeys--;
return;
}
 // case 2 : right has one extra key, move
      a key from right
    else
      If(right->noofkeys >mkeys/2)
    {
    left->data[left->noofkeys].key= centre->key;
          left->noofkeys++;
          centre->key=right->data[0].key;
          for(i=1;i<right->noofkeys;i++)
          right->data[i-1]=right->data[i];
          right->noofkeys--;
          return;
    }
    else
    {   //merge left and right
    left->data[left->noofkeys].key=centre->key;
    left->noofkeys++;
    for(j=0;j<right->noofkeys;j++)
     left->data[left->noofkeys+j]=right->data[j];
    left->noofkeys+=right->noofkeys;
    //delete the pair from the parent
    //cout<<"\ncentre index,noofkeys ";
    //cout<<centreindex<<" "<<q->noofkeys;
    for(i=centreindex+1;i<q->noofkeys ;i++)
        q->data[i-1]=q->data[i];
        q->noofkeys--;
        p=q;//for next iteration
       }
  }
}
```

Q.10 Create a B tree of order 3 for the following data : 20, 10, 30, 15, 12, 40, 50. **SPPU - Dec. 13, 4 Marks**

Ans. :

Sr. No.	Data	B Tree
1.	20	
3.	30	

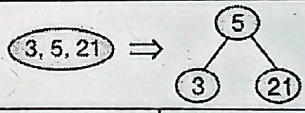
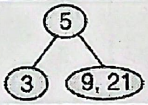
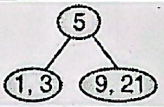
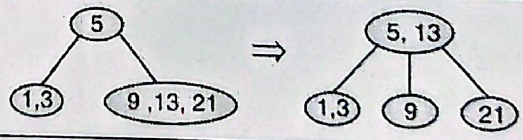
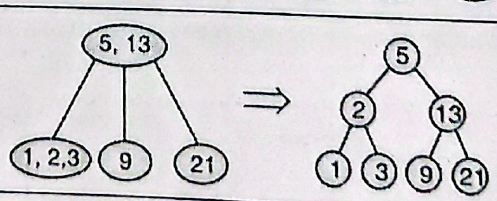
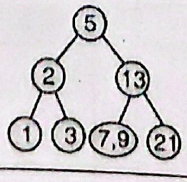
Sr. No.	Data	B Tree
2.	10	
4.	15	

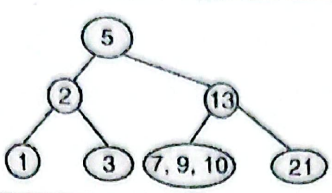
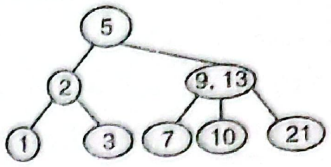
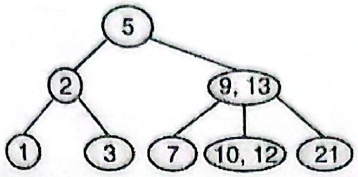
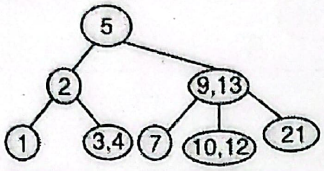
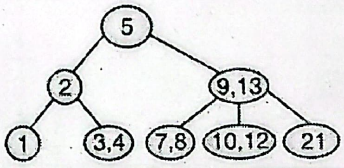
Sr. No.	Data	B Tree
5.	12	
6.	40	
7	50	

Q.11 Create a 3 way B tree by inserting the following data one at a time : 5, 3, 21, 9, 1, 13, 2, 7, 10, 12, 4, 8.

SPPU - Dec. 14, 6 Marks

Ans. :

Sr. No.	Data to be inserted	B-tree after insertion	B-tree after adjustment
1.	5, 3		
2.	2		
3.	9		
4.	1		
5.	13		
6.	2		
7.	7		

Sr. No.	Data to be inserted	B-tree after insertion	B-tree after adjustment
8.	10		
9.	12		
10.	4		
11.	8		

Q.12 What is a B^+ tree? Give structure of its internal node. What are the order of B^+ tree and characteristics of B^+ tree ? **SPPU - May 14, 7 Marks**

Ans.: B^+ Trees

- B^+ tree is a variation of B-tree data structure. In a B^+ tree , data pointers are stored only at the leaf nodes of the tree. In a B^+ tree structure of a leaf node differs from the structure of internal nodes.
- The leaf nodes have an entry for every value of the search field, along with a data pointer to the record (or to the block that contains this record).
- The leaf nodes of the B^+ tree are linked together to provide ordered access on the search field to the records.
- Internal nodes of a B^+ tree are used to guide the search.
- Some search field values from the leaf nodes are repeated in the internal nodes of the B^+ tree.

Structure of internal node

The structure of the internal node is shown in Fig. 5.6(a).

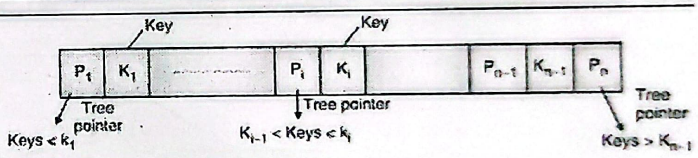


Fig. 5.6(a) : Structure of an internal node of B^+ tree

- Each internal node is of the form
  $<P_1, K_1, P_2, K_2 \ldots P_{n-1}, k_{n-1}, P_n>$
  k_i is the key and P_i is a tree pointer
- Within each internal node, $k_1 < k_2, \ldots < k_{n-1}$
- For all search field value x in the subtree pointed at by P_i,we have $K_{i-1} < x \leq K_i$.
- Each internal node has at most p tree pointers.
- Each internal node, except the root, has at least [(P/2)] tree pointers.

Structure of a leaf node

The structure of a leaf node of a B^+ tree is shown in Fig. 5.6(b).

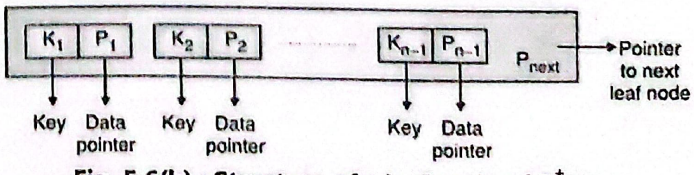
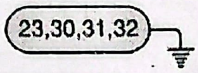
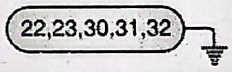
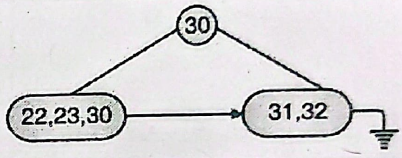
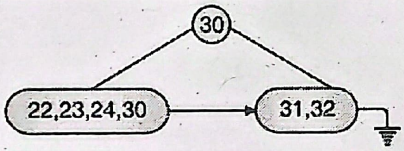
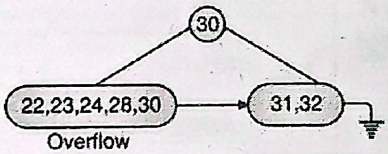
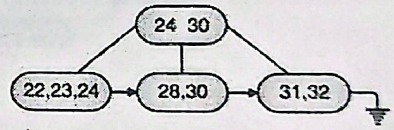
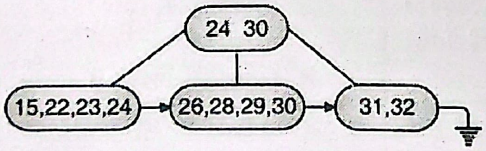
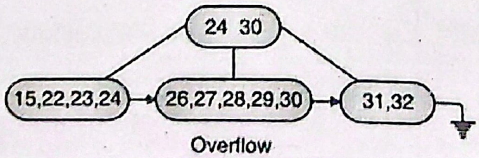
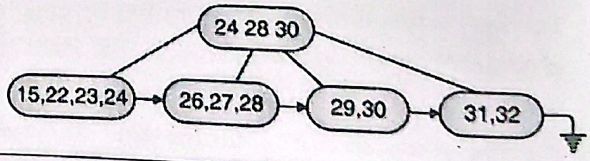
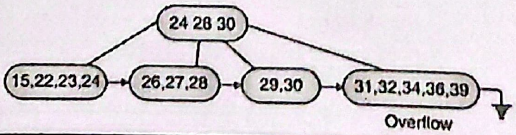
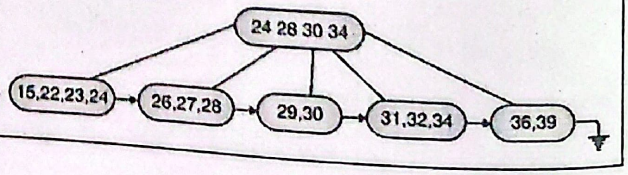


Fig. 5.6(b) : Structure of a leaf node of B^+ tree

- Each leaf node is of the form
  $<<K_1, P_1>, <K_2,P_2> \ldots <K_{n-1}, P_{n-1}>, P_{next}>$
- Within each leaf node, $K_1 < K_2 \ldots < K_{n-1}$.
- P_i is a data pointer that points to the record whose search field value is k_i.
- Each leaf node has at least [(P/2)] values.
- All leaf nodes are at the same level.

Q.13 Construct a B^+ tree of order 5 for the following data : 30, 31, 23, 32, 22, 28, 24, 29, 15, 26, 27, 34, 39, 36. **(5 Marks)**

Ans. :

Insert	Tree after insertion	Tree after splitting (if required)
30, 31, 23, 32		
22	 Overflow	
24		
28	 Overflow	
29, 15, 26		
27	 Overflow 	
34, 39, 36	 Overflow 	

Q.14 Explain UNION and FIND Operations for Disjoint Sets. **(5 Marks)**

Ans.: UNION and FIND Operations for Disjoint Sets

A relation over a set of elements $a_1, a_2, \ldots a_n$ can be divided into equivalent classes.

- The equivalent class of an element $a \in s$ is the subset of S that contains all the elements of S that are related to a.
- Divide a set of elements into equivalent classes through the two operations
  1. UNION 2. FIND
- A set is divided into subsets. Each subset contains related elements. If we come to know that the two element a_i and a_j are related, then we can do the followings :
  1. Find the subset S_i containing a_i
  2. Find the subset S_j containing a_j
  3. If S_i and S_j are two independent subsets

  then we create a new subset by taking union of S_i and S_j

  New subset $= S_i \cup S_j$.
- This algorithm is **dynamic** as during the course of the algorithm, the sets can change via the union operation.

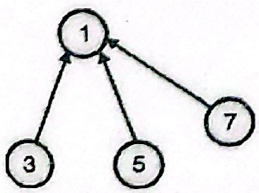
Example

Consider two disjoint sets

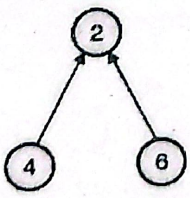
$S_1 = \{1, 7, 5, 3\}$

$S_2 = \{2, 4, 6\}$

Representation of these two sets are shown in the Fig. 5.7.



(a) Representation of S_1



(b) Representation of S_2

1	2	3	4	5	6	7
0	0	1	2	1	2	1

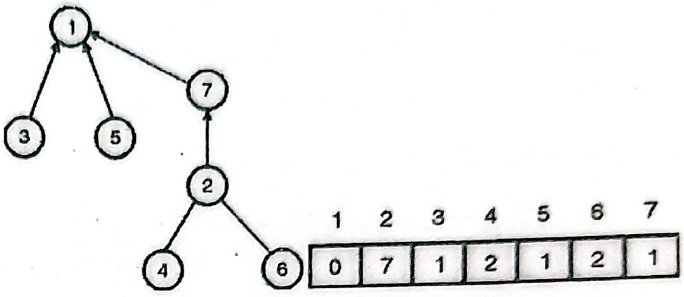
(c) Representation of S_1 and S_2 in an array

Fig. 5.7 : Representation of S_1 and S_2

Now, if we come to know that $7 R 4$ [7 is related to 4]

- First find the set S_1 containing the element 7.
- Find the set S_2 containing the element 4.
- Finally, we take union of S_1 and S_2.

Union of two sets S_1 and S_2 is shown in the Fig. 5.8.



(a) Tree representation of $S_1 \cup S_2$ (b) Array representation of $S_1 \cup S_2$

Fig. 5.8 : One possible representation of $S_1 \cup S_2$

Q.15 Write a short note on : Smart Union Algorithms. **(5 Marks)**

Ans.:

Smart Union Algorithms

UNION algorithm is very easy to implement, but its performance is not very good. The FIND operation requires traversing the chain from any element x upto the root node. Thus the time required to FIND an element x at level i is O(i). The UNION algorithm can be improved using some minor modifications. Two such algorithms are :

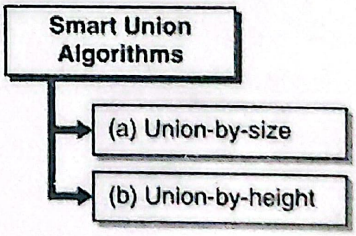


Fig. 5.9 : Smart union algorithm

1. Union-by-size

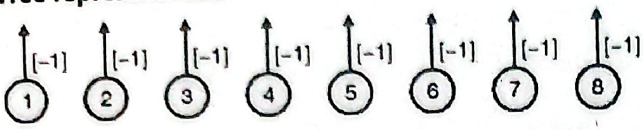
The height of the resultant tree representing $S_1 \cup S_2$, the union of two sets S_1 and S_2 can be reduced by making the smaller tree a subtree of the larger tree.

Example : Consider the behaviour of Union-by-size algorithm on the following sequence of unions, starting from the initial configuration.

(1, 2), (3, 4), (5, 6), (7, 8), (1, 3), (1, 5)

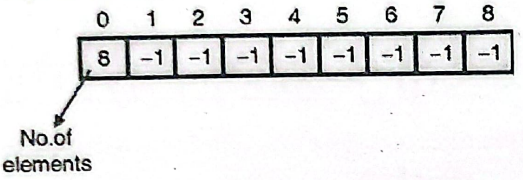
Step 1 : Initial configuration

Tree representation



The value [– 1] against the node shows number of values in the tree but with –ve sign.

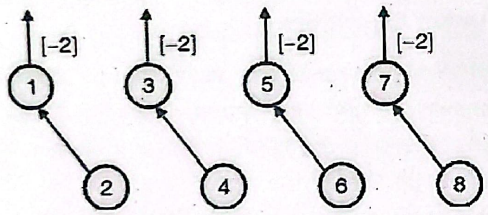
Array representation



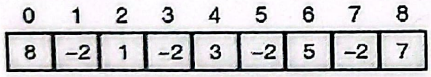
A root node contains a negative value. This gives the number of elements in the subtree but with –ve sign.

Step 2 : After taking union of (1, 2), (3, 4), (5, 6) and (7, 8)

Tree representation

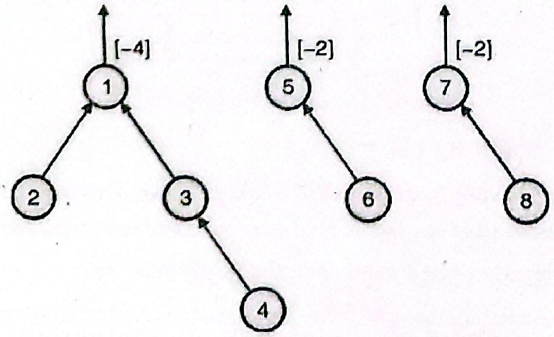


Array representation

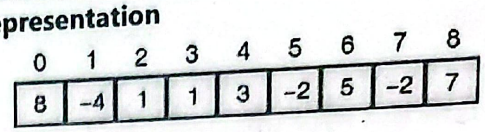


Step 3 : After taking union of (1, 3)

Tree representation

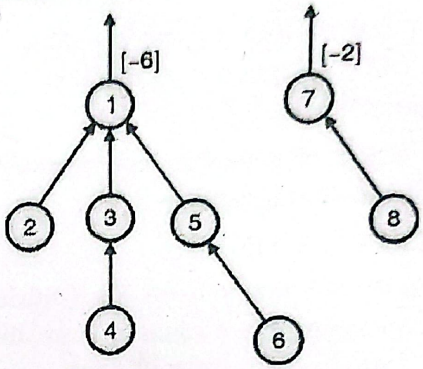


Array representation

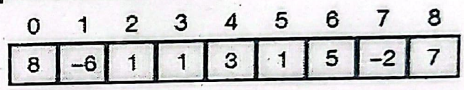


Step 4 : After taking union of (1, 5)

Tree representation



Array representation



2. Union-by-height

In this approach, we store the height instead of size of each tree in the root node. This guarantees that all trees will have depth of at most O(log N).

In this approach, while taking the union, a tree with smaller height is made a subtree of the tree will larger height.

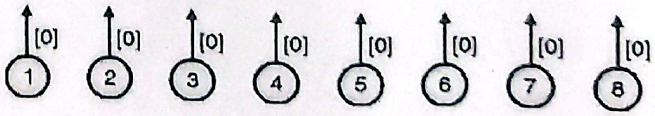
Example

Consider the behavior of Union-by-height algorithm on the following sequence of unions, starting from the initial configuration.

(1, 2), (3, 4), (5, 6), (7, 8), (1, 3), (1, 5)

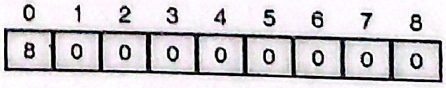
Step 1 : Initial configuration

Tree representation



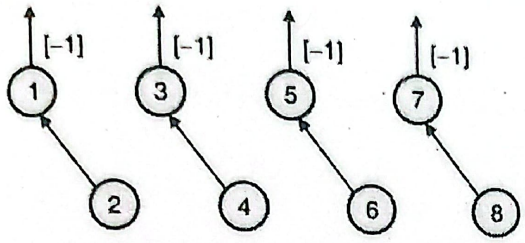
The value [0] against the node show that the height of the subtree is zero.

Array representation



Step 2 : After taking union of (1, 2), (3, 4), (5, 6) and (7, 8)

Tree representation



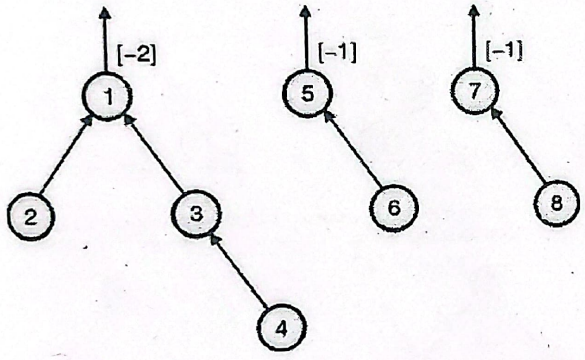
The value [– 1] against the root node gives the height of the subtree but with –ve sign.

Array representation

0	1	2	3	4	5	6	7	8
8	-1	1	-1	3	-1	5	-1	7

Step 3 : After taking union (1, 3)

Tree representation

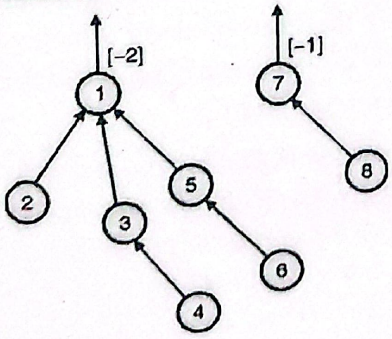


Array representation

0	1	2	3	4	5	6	7	8
8	-2	1	1	3	-1	5	-1	7

Step 4 : After taking union of (1, 5)

Tree representation



Array representation

0	1	2	3	4	5	6	7	8
8	-2	1	1	3	1	5	-1	7

Q.16 What is max heap ? Write a function to insert an element in max heap. What is the time complexity of inserting an element in max heap ?

SPPU - May 17, 7 Marks

Ans.: Max heap :

- A max heap is a complete binary tree with the property that the value of each node is at least as large as the values at its children.

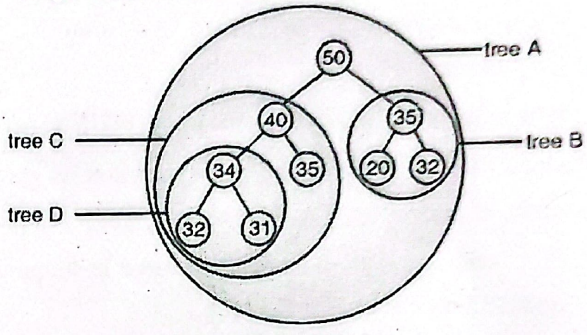


Fig. 5.10(a) : Max heap

- Fig. 5.10(a), shows an example of max heap. It can easily be verified that the value at each node is at least as large as the value at its children.

Basic Operations for Max Heap

The basic operations for max heap are

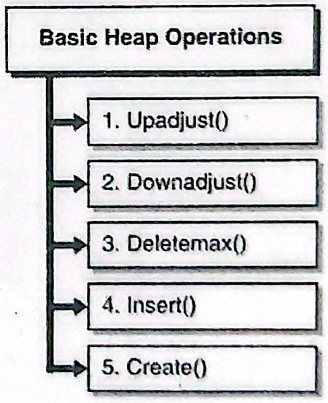


Fig. 5.10(b) : Operations for max heap

1. Upadjust()

To insert an element X into a heap, X is added at the next available place.

If after addition of X, heap property is not satisfied i.e. the value of X is larger than the value of its parent node.

The current element moves up (through simple exchange of X and the value stored at its parent node). The process of up adjustment stops when :

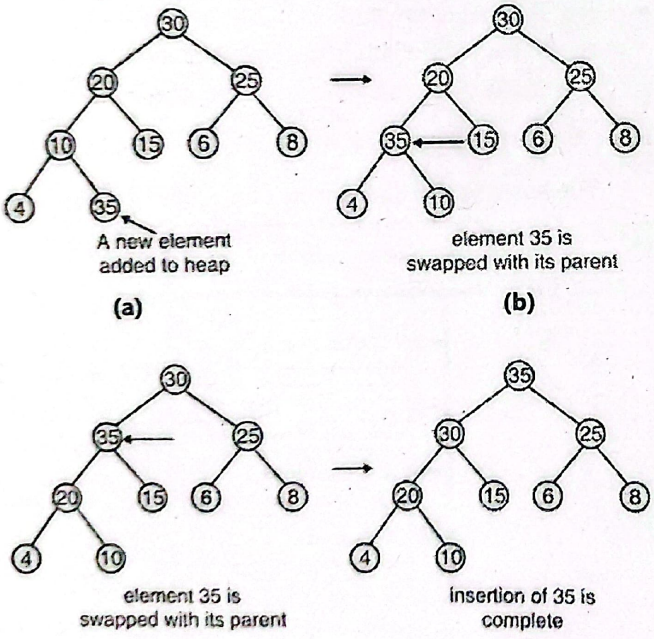
1. X reaches the root node

   or

2. The value stored at the parent node of X is larger than the value X.

If a new element "35" is to be added to an existing max heap of Fig. 5.11.

- It is added at the next available place, Fig. 5.11(a).
- Since 35 is larger than the value stored at its parent node, 35 moves up. Fig. 5.11(b).
- 35 is still larger than the value stored at its parent node, 35 moves up. Fig. 5.11(c).
- Upadjustment stops after 35 reaches the root node.



(a) (b) (c) (d)

Fig. 5.11 : Process of upadjustment, after addition of new element "35"

'C++' Function for Inserting an Element in a Heap

```
void insert(int heap[],int x)
{
   int n;
   /* no. of elements in a heap is given by heap[0] */
   n=heap[0];
/* new element is inserted at the next available location
*/
   heap[n+1]=x;
/* new element is upadjusted after insertion */
   heap[0]=n+1;
   upadjust(heap,n+1);
}
```

2) Deletemax() / downadjust()

Finding the maximum value is easy; removing it is difficult. When the maximum is deleted, the last element X in the heap is moved to the root node. This helps in keeping the 'complete binary tree' property of heap intact.

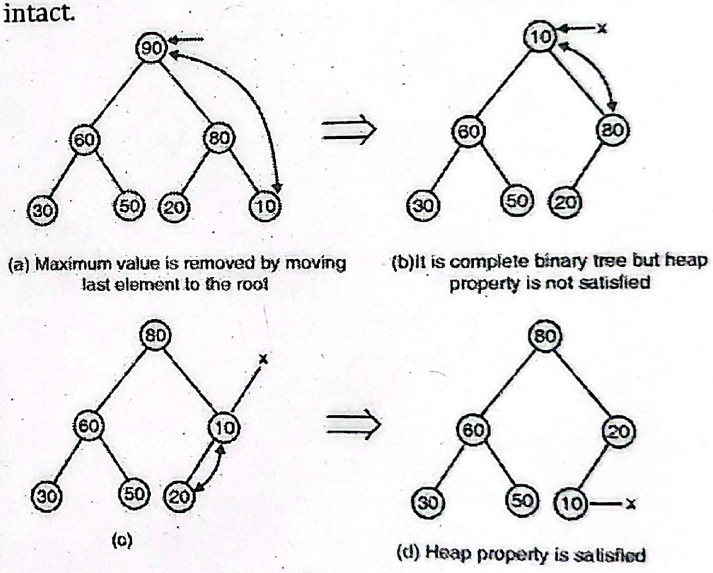


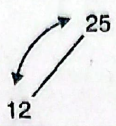
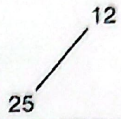
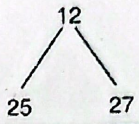
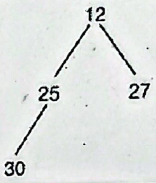
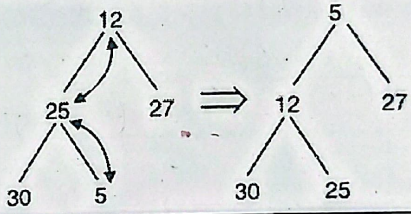
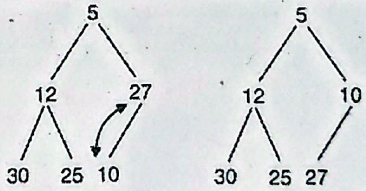
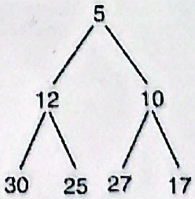
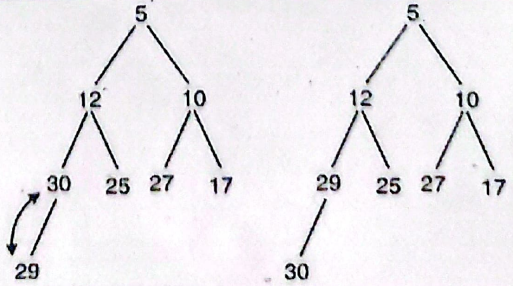
Fig. 5.12

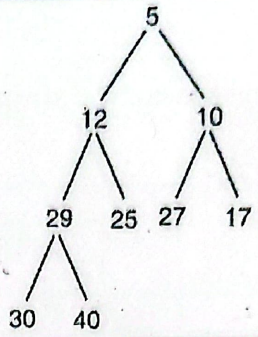
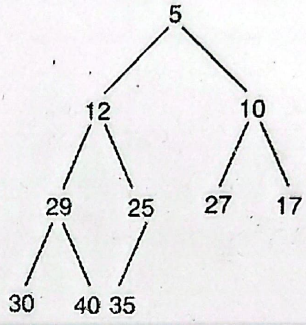
After the value of the root node is changed, it may be possible that the heap property is not satisfied. For example, after deletion of 90 in Fig. 5.12, the last element "10" is moved to the root.

To restore the heap property, the value of the root and the larger of its children is exchanged. This gives a heap as shown in the Fig. 5.12(c). Heap condition is still violated at node X. Value at node X is again exchanged with the value of its only child yielding a heap as shown in Fig. 5.12(d).

Q.17 Build the min-heap for the following data : 25, 12, 27, 30, 5, 10, 17, 29, 40, 35. **SPPU - Dec. 14, 4 Marks**

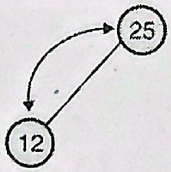
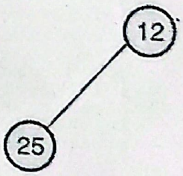
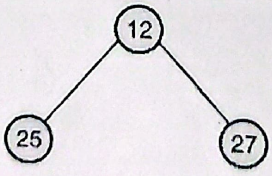
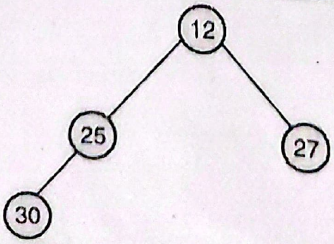
Ans. :

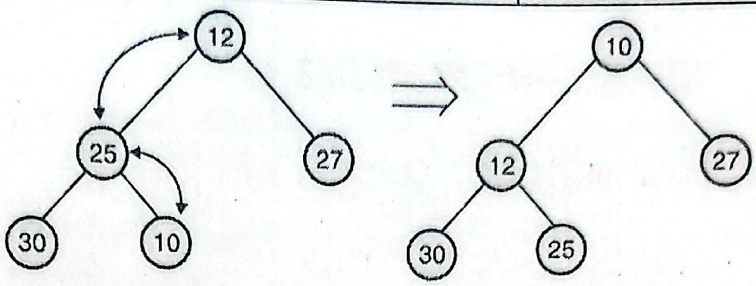
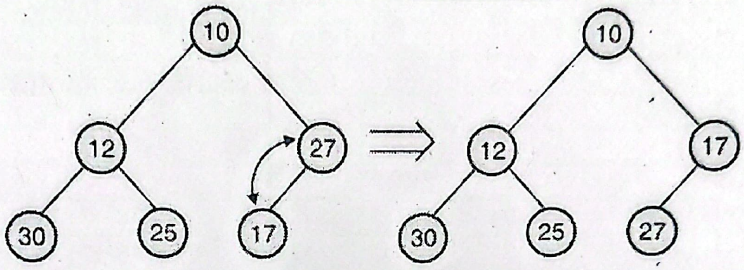
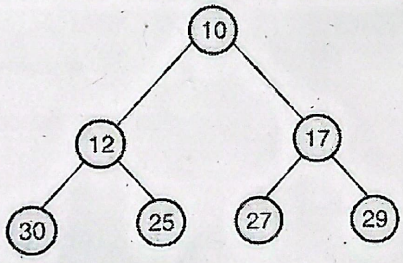
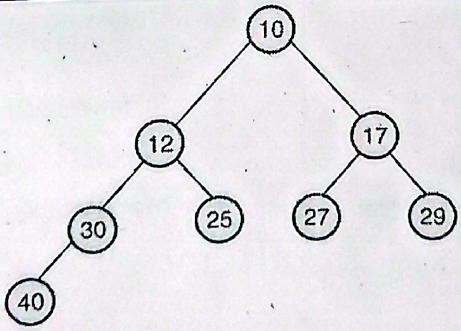
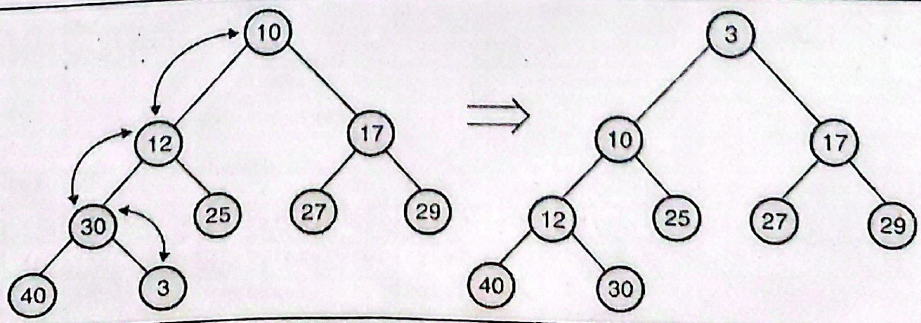
Sr. No.	Data to be inserted	Heap after insertion	Head after adjustment
1.	25	25	
2.	12		
3.	27		
4.	30		
5.	5		
6.	10		
7.	17		
8.	29		

Sr. No.	Data to be inserted	Heap after insertion	Head after adjustment
9.	40		
10.	35		

Q.18 Create the min-heap for given data : 25, 12, 27, 30, 5, 10, 17, 29, 40, 3. **SPPU - Dec. 19, 4 Marks**

Ans. :

Sr. No.	Data	Heap after insertion	Heap after adjustment
1.	25		
2.	12		
3.	27		
4.	30		

Sr. No.	Data	Heap after insertion	Heap after adjustment
5.	10		
6.	17		
7.	29		
8.	40		
9.	3		

Q.19 Write a pseudo C/C++ code to sort the data in ascending order using heap sort.

SPPU - Dec. 13, Dec. 14, 6 Marks

Ans.:

Sorting in Ascending Order (Program)

```
#include<conio.h>
#include<iostream.h>
void create(int[]);
void down_adjust(int[],int);
void main()
{
    int heap[30],n,i,last,temp;
    clrscr();
    cout<<"\nEnter no. of elements :";
    cin>>n;
    cout<<"\nEnter the data to be sort :";
    for(i=1;i<=n;i++)
      cin>>heap[i];
    //create a heap
    heap[0]=n;
    create(heap);
    //sorting
    while(heap[0] > 1)
    {
        //swap heap[1] and heap[last]
        last=heap[0];
        temp=heap[1];
        heap[1]=heap[last];
        heap[last]=temp;
        heap[0]--;
        down_adjust(heap,1);
    }
    //print sorted data
    cout<<"\nsorted data is :";
    for(i=1;i<=n;i++)
      cout<<heap[i]<<" ";
}
void create(int heap[])
{
    int i,n;n=heap[0]; //no. of elements
    for(i=n/2;i>=1;i--)
        down_adjust(heap,i);
}
void down_adjust(int heap[],int i)
{
    int j,temp,n,flag=1;
    n=heap[0];
    while(2*i<=n && flag==1)
    {
        j=2*i;  //j points to left child
        if(j+1<=n && heap[j+1] > heap[j])
            j=j+1;
        if(heap[i] > heap[j])
            flag=0;
        else
        {
            temp=heap[i];
            heap[i]=heap[j];
            heap[j]=temp;
            i=j;
        }
    }
}
```

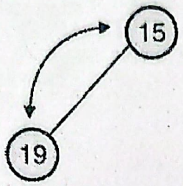
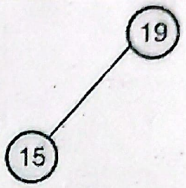
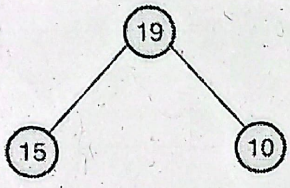
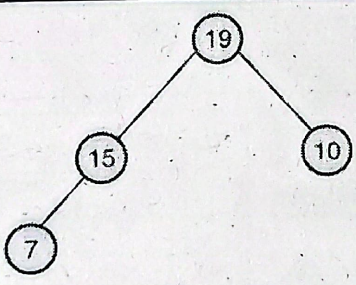
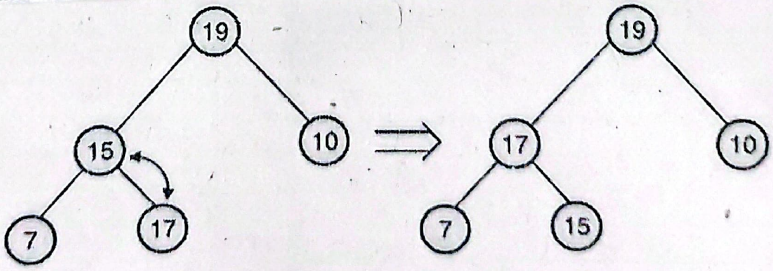
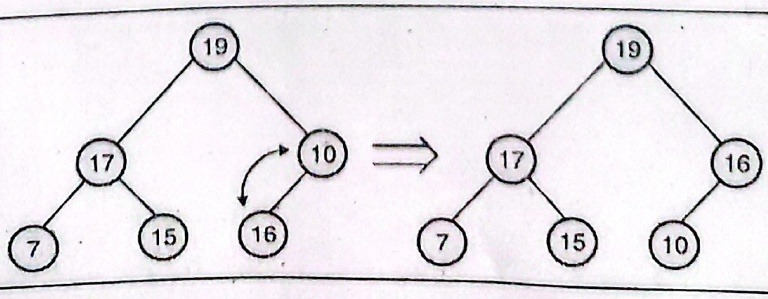
Output

```
Enter no. of elements : 3
Enter the data to be sort : 35  23  56
sorted data is :  23  35  56
```

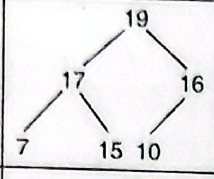
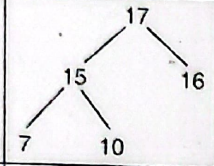
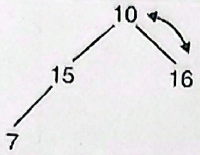
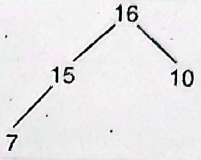
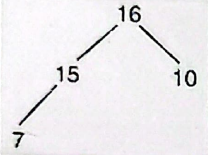
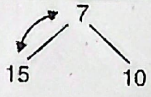
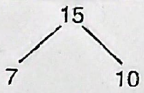
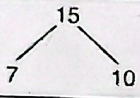
Q.20 Sort the data in ascending order using heap sort : 15, 19, 10, 7, 17, 16. SPPU - May 14, Dec. 19, 6 Marks

Ans. : Creation of max heap

Step 1 : Creation of max-heap

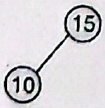
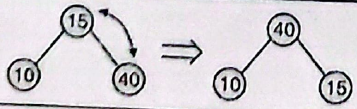
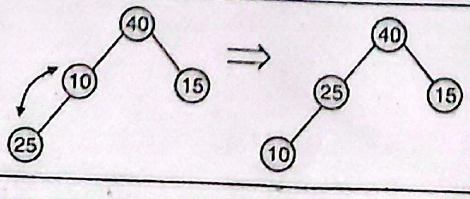
Sr. No.	Data	Heap after insertion	Heap after adjustment
1.	15		
2.	19		
3.	10		
4.	7		
5.	17		
6.	16		

Step 2 : Sorting

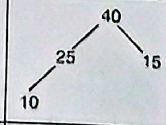
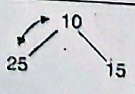
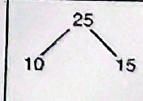
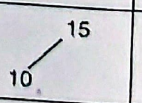
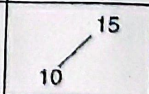
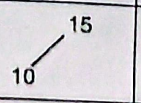
Sr. No.	Given heap	Heap after deletion	Heap after adjustment	Sorted data
1.				19
2.				17, 19
3.				16, 17, 19
4.				15, 16, 17, 19
5.		7	7	10, 15, 16, 17, 19
6.	7	–	–	7, 10, 15, 16, 17, 19

Q.21 Sort the following data in ascending order using heap sort: 15, 10, 40, 25. **SPPU - Dec. 13, 4 Marks**

Ans. : Step 1 : Creation of max-heap

Sr. No.	Data	Tree
1.	15	
2.	10	
3.	40	
4.	25	

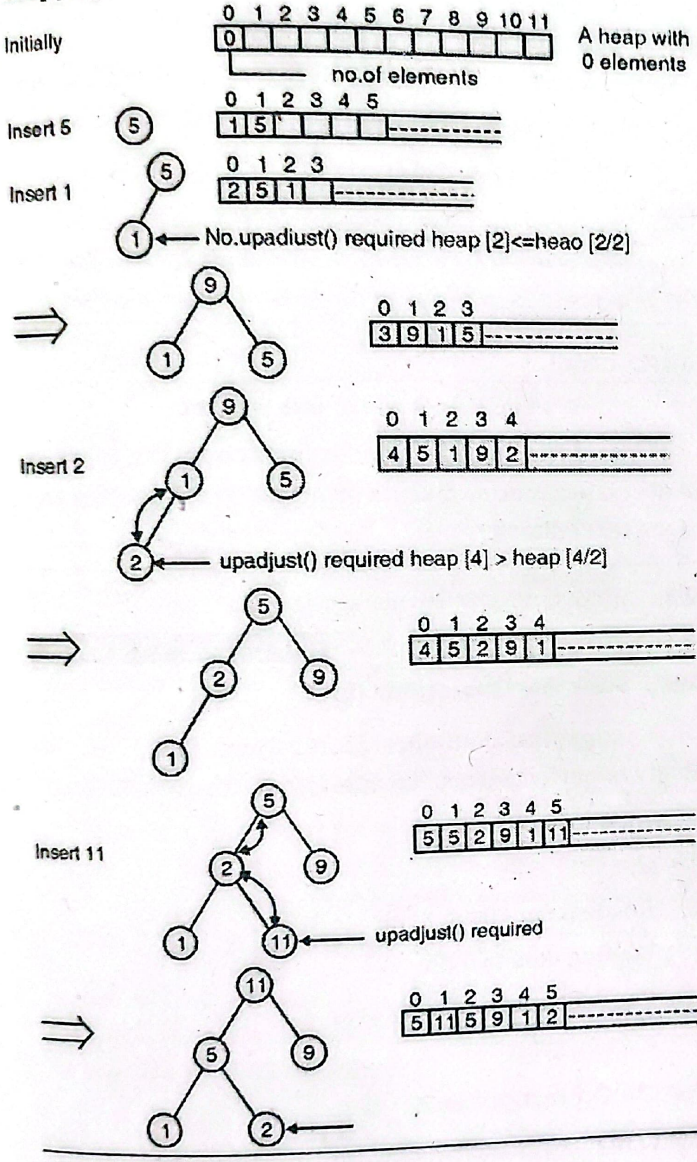
Step 2 : Sorting

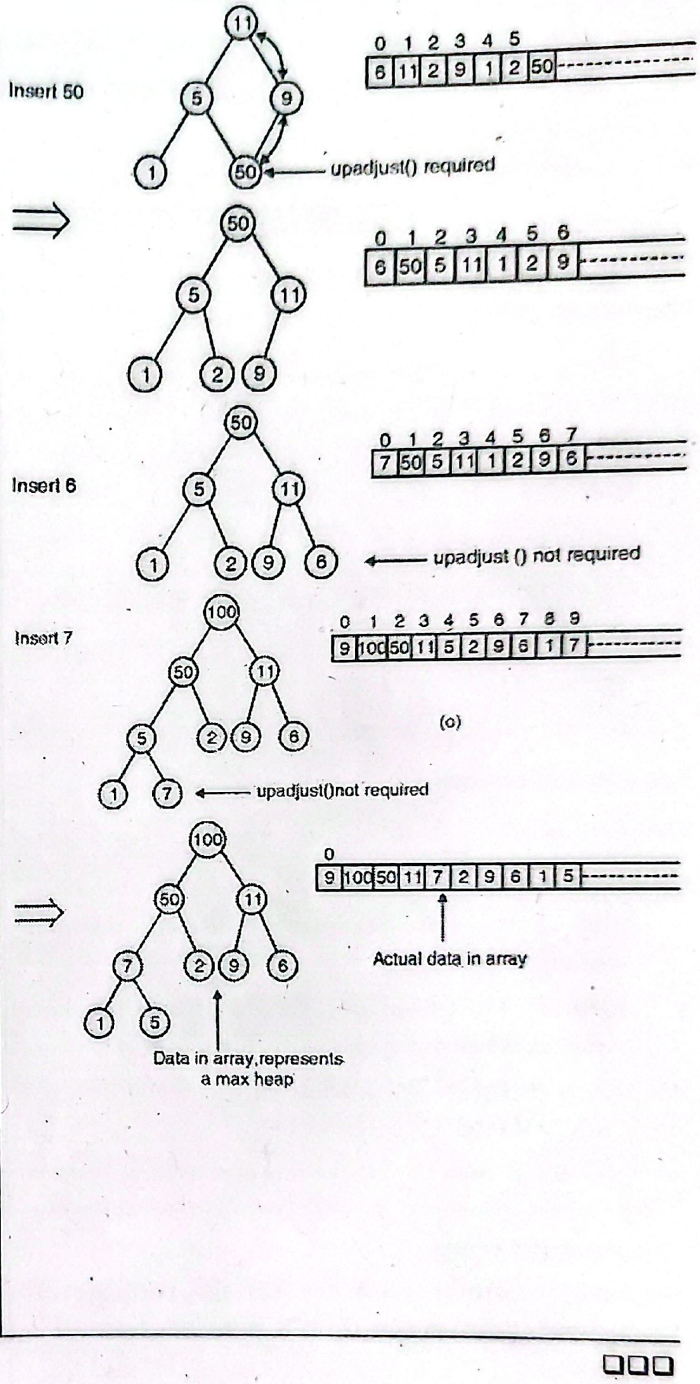
Sr. No.	Given heap	Heap after deletion of root element	Heap after heapification	Sorted data
1.				40
2.				25, 40
3.		10	10	15, 25, 40
4.	10			10, 15, 24, 40

Q. 22 Create a max heap with following elements : 5, 1, 9, 2, 11, 50, 6, 100, 7. **(5 Marks)**

Ans. :

Assume that heap is represented using an array heap[12].





Unit VI : File Organization

Q.1 Explain the various modes of opening the file in C or C++. **SPPU - Dec. 13, Dec. 19, 3 Marks**

Ans.:

Files I/O in 'C'

Before a file can be used for reading or writing , it must be opened. This is done by a call to the fopen() function. The function fopen() requires two arguments :

1) File name
2) Option (r, w, a, etc)

Option indicates what we want to do with the file :

a) read it
b) write to it
c) append to it (writing at the end).

File opening options

For text file

r open existing file for reading

w open (create if necessary), discard pervious contents.

a open (create if need be), do not discard previous contents, write at the end of the file.

r+ open existing file for reading and writing. It behaves like 'r' with the option of writing.

w+ create and open file for reading and writing, discard previous contents. It behaves like 'w' with the option of reading.

a+ open (create if need be) file for reading and appending. It behaves like 'a' with the option for reading.

For binary file

rb open an existing binary file for reading.

wb open (create if necessary) a binary file, discard previous contents.

ab open (create if need be) a binary file, do not discard previous contents, write at the end of the file.

r+b open an existing binary file for reading and writing. It behaves like 'rb' with the option of writing.

w+b create and open a binary file for reading and writing, discard previous contents. It behaves like 'wb' with the option of reading.

a+b open (create if need be) a binary file for reading and appending. It behaves like 'ab' with the option of reading.

File :

A file can be treated as a stream of bytes. Size of a file is expressed in terms of number of bytes in a file.

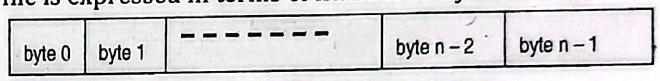


Fig. 6.1 : A file of size n bytes

A **text file** contains visible characters. The contents of file on monitor or take its print-out or edit it using any of the text editors.

Q.2 Enlist out basic file operations in C. **SPPU - Dec. 19, 4 Marks**

Ans.: Basic File Operations (in 'C')

C supports a number of functions that have the ability to perform basic file operations, which include :

- Naming a file
- Opening a file
- Reading data from a file
- Writing data to a file
- Random access of a record
- Closing a file.

List of I\O functions in 'C'

fopen() – Open and existing file or create a new file for use.

fclose() – Closes a file

getc() – Reads a character from a file

putc() – Write a character to a file

fprintf() – Write a set of data values to a file

fscanf() – Read a set of data values from a file

fread() – Reading a record from a file

fwrite() – Writing a record to a file

getw() – Reads an integer from a file

putw() - Writes an integer to a file.

fseek() - Sets the pointer to a desired location in the file

ftell() - Gives the current position in the file (offset from beginning in number of bytes)

rewind() - Set the position to the beginning of a file.

1. fopen()

A call to fopen() on file **myfile** looks like :

fopen("myfile", "r") ; [open myfile in read mode]

If fopen() is successful in its mission of opening the file for reading, it returns a pointer, a pointer of type **FILE**. If fopen() cannot successfully open the file, it returns a **NULL** pointer instead.

A pointer of type **FILE** must be declared :

FILE *fp ;

Now, assign the pointer returned by fopen() to fp:

fp = fopen("myfile", "r");

2. fclose()

A call to fclose() closes a previously opened file. The single argument of fclose() is the relevant pointer.

fclose(fp);

3. getc() and putc()

The simplest file I/O functions are getc() and putc(). getc() reads a character from a file, while putc() writes a character to the file. The general format of the above functions is given below:

putc(c, fp);

Writes the character stored in variable c to the file associated with the pointer fp.

C = getc(fp)

Reads a character from the file with the pointer fp. The getc() function will return a specified character EOF, when end of the file is reached. Reading should terminate on reacting the end.

Q.3 Explain sequential file, random access file organization. **SPPU - May 17, 6 Marks**

Ans.: Sequential and Random Access Files

File functions, getw(), putw(), fscanf(), fprintf(), getc() and putc() are useful in reading and writing data sequentially. In a sequential file both reading and writing is done sequentially. In a sequential file, records are added in the order they arrive.

Characteristic of a sequential file

- In a sequential file, records are added in the order they arrive.
- Length of the record need not be fixed. Such files are suitable for storing text. In a text, two lines need not be of the same size. Sequential organization is easy to implement.
- Searching a record is time consuming. To access the n^{th} record, all records starting from record number 1 to n – 1 must be read to reach the n^{th} record.
- Insertion and deletion is time consuming. All records, ahead of the point of insertion/deletion are required to be moved. This is very time consuming process.

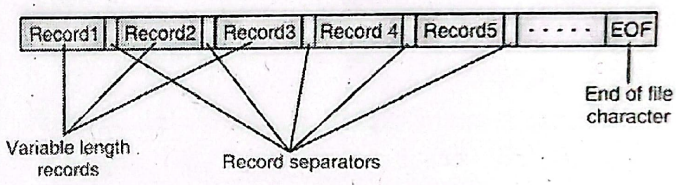


Fig. 6.2 : A sequential file

Random access file

- In a random access file, we can read/write a specific record without having to read previous records. We can position the file pointer to the required location and then perform read/write operation. In a random access file, all records are of same length.

Starting address of i^{th} record = i × (size of each record)

- fseek() function can be used to move the file pointer (place of reading/writing) to the desired location. It takes the following form:

fseek(file pointer, offset, position)

- **Offset** is a number or variable of type long. It specifies the number of bytes to be moved from the current position, beginning or the end. Offset could be a negative offset or a positive offset. The position could take one of the following values :

0 Beginning of file

1 Current position

2 End of file.

Statement	Meaning
fseek(fp, 0, 2)	Go to the end of file.
fseek(fp, resize, 1)	Go to the next record (resize = size of the record)
fseek(fp, – resize ,1)	Go to the previous record.
Fseek(fp, 10* resize, 0)	Go to the 10th record.

- ftell() function gives the current position of the file and rewind() sets the current position to the beginning of the file.

Q.4 Explain binary files with example. **(5 Marks)**

Ans.:

Binary Files

fread() and fwrite() functions can be used to read or write a block of data respectively. General format of read() and write() function is given below :

fread(address of record, size of record, number of records, file pointer);

fwrite(address of record, size of record, number of records, file pointer);

Program : A C- program to handle binary file

```
#include<stdio.h>
#include<conio.h>
typedef struct item
  {
     char name [30];
     int code;
     int price;
  }item;
void main()
{
   item nyitem;
   int n, i;
   char filename [30]; FILE *fp;
   printf("\n Enter file name :");
   gets(filename);
   fp=fopen(filename, "wb");
      printf("\n Enter number of items :");
      scanf("%d", &n);
   for(i=0; i < n; i++)
   {
      printf("\n Enter next item(name, code, price):");
      scanf("%s%d%d", nyitem.name, &nyitem.code,
       &nyitem.price);
      fwrite(&nyitem, sizeof(nyitem), 1, fp);
   }
   fclose(fp);
/* displaying contents of file */
fp=fopen(file name,"rb");
while(! feof(fp))
{
   fread(&nyitem, sizeof(nyitem), 1, fp);
   printf("\n%s\t%d\t", nyitem.name, nyitem.code,
    nyitem.price);
}
getch();
}
```

Q.5 Write a short note on : Methods of blocking. **(5 Marks)**

Ans.: Methods of blocking

There are four methods of blocking that can be used. Methods of blocking are :

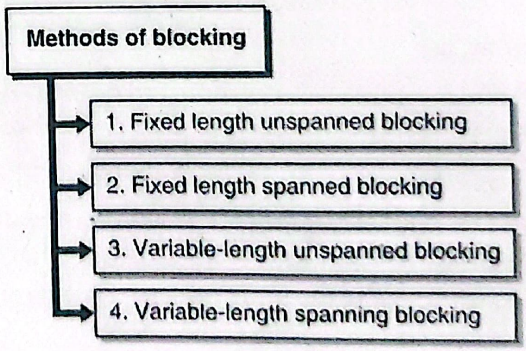


Fig. 6.3(a) : Methods of blocking

1. Fixed length unspanned blocking

Fixed length records are used. ⌊B/R⌋ records per block can be stored. There may be unused space at the end of each block. This is also called **internal fragmentation.**

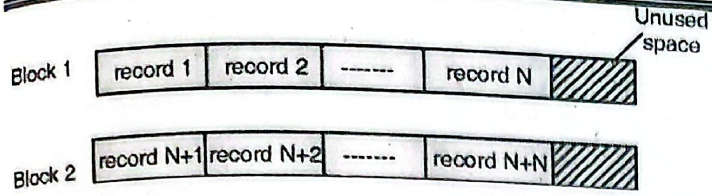


Fig. 6.3(b) : Unspanned, fixed length record

2. Fixed length spanned blocking

To utilize the unused space at the end of the block, we can store part of a record on one block and the rest on another. A pointer at the end of the first block points to the block containing remainder of the record.

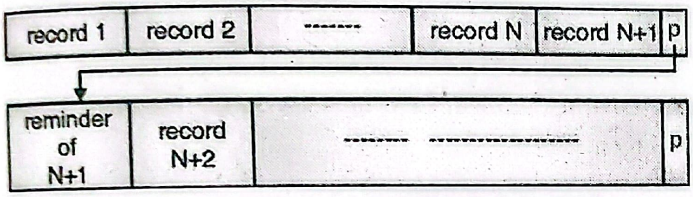


Fig. 6.3(c) : Spanned, fixed length record

3. Variable-length unspanned blocking

Variable length records are used. Spanning of a record is not allowed. There could be unused space at the end of a block.

4. Variable-length spanning blocking

Variable length records are used and are packed into blocks with no unused space. Usually, the last record of a block spans two blocks. Continuity is indicated by a pointer to the successor block.

Q.6 Explain different types of file organization. **(5 Marks)**

Ans.:

File Organization

A file is a collection of records where each record consists of one or more fields. File organization can be defined as the method of storing data records in a file. The primary objective of file organization is to provide means for record retrieval and update. The factors involved in selecting a particular file organization for uses are :

- Economy of storage
- Ease of retrieval
- Convenience of updates
- Reliability
- Security
- Integrity
- Volume of transaction

Commonly, used file organizations are :

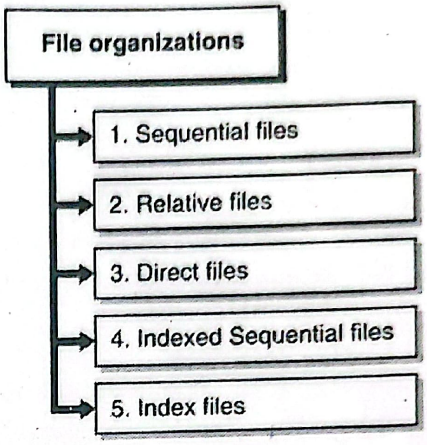


Fig. 6.4 : File organization

1. **Sequential file :** In a sequential file, data records are stored in a specific sequence. Records are physically ordered on the value of one of the fields-called the ordering field. Records could also be stored in the order of arrival.
2. **Relative file :** In a relative file, each record is stored at a fixed place in a file. Each record is associated with an integer key value, which is mapped to a fixed slot in a file.
3. **Direct file :** Direct file is similar to relative file. Direct file is popularly known as a hashed file. Here the key value need not be an integer.
4. **Indexed sequential file :** In an indexed sequential file an index is added to a sequential file to provide random access. An overflow area should be maintained to allow insertions.
5. **Index file :** In an index file, data records need not be sequenced. An index is maintained to improve access.

Q.7 Explain primitive operations on a file. **(5 Marks)**

Ans.:

Primitive Operations on a File

Basic file operations that can be performed on a file are :

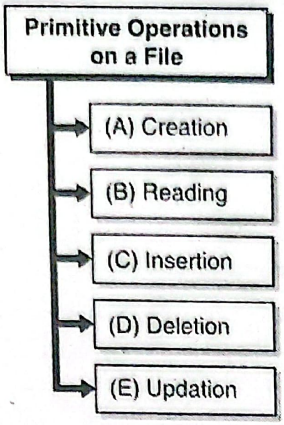


Fig. 6.5 : Primitive operations on a file

1. Creation

Before performing any operation, the file must be created on a secondary storage. After creation, file is opened in either read, write or both mode, depending on the type of operation to be performed.

2. Reading

We can read either a particular record or the entire file. Algorithm for reading depends on the file organization.

3. Insertion

A new record can be added to an existing file. In sequential organization, a new record may be inserted between two records. For doing so, records ahead of the point of insertion should be moved by 1 place.

4. Deletion

A record may be deleted from a file, if it is no longer required. A record may be deleted either **logically** or physically. Logical deletion is done with the help of an additional field in every record. This field, tells the status of a record. Physical deletion of a record in sequential organization is a time consuming process. It is done with the help of a temporary file. It is done in two steps.

Step 1 :

To delete nth record from a master file, records 1 to n – 1 and n + 1 to last record are copied to a temporary file.

Step 2 :

Contents of temporary file are copied back to master file.

Algorithm for deletion depends on file organization.

5. Updation

In this operation, we modify contents of a file. Algorithm for deletion is dependent on file organization.

Q.8 Define sequential file organization and state its advantages and disadvantages.

SPPU - May 14, Dec. 18, May 19, Dec. 19, 6 Marks

Ans.:

Sequential Files

It is the most common type of file. In this type of file :

- A fixed format is used for record.
- All records are of the same length.
- Position of each field in record and length of field is fixed.
- Records are physically ordered on the value of one of the fields – called the ordering field.

Block 1

Name	Roll No	Year	Marks
SINHA, AMIT	1000	1	60.48
SINGH, PRATAP	1010	2	53.98
AGRAWAL,MOHAN	1012	1	70.43
SATHE,VIVEK	1015	3	65.4

Block 2

JAISWAL	1016	4	68.90

Fig. 6.6 : Some blocks of an ordered (sequential) file of students records with Roll no. as the ordering field

Advantages of sequential file over unordered files

1. Reading of records in order of the ordering key is extremely efficient.
2. Finding the next record in order of the ordering key usually, does not require additional block access. Next record may be found in the same block.
3. Searching operation on ordering key is must faster. Binary search can be utilized. A binary search will require $\log_2 b$ block accesses where b is the total number of blocks in the file.

Disadvantages of sequential file

1. Sequential file does not give any advantage when the search operation is to be carried out on non-ordering field.
2. Inserting a record is an expensive operation. Insertion of a new record requires finding of place of insertion and then all records ahead of it must be moved to create space for the record to be inserted. This could be very expensive for large files.
3. Deleting a record is an expensive operation. Deletion too requires movement of records.
4. Modification of field value of ordering key could be time consuming. Modifying the ordering field means the record can change its position. This requires deletion of the old record followed by insertion of the modified record.

Q.9 Explain any three operations carried out on sequential file. Write pseudo code for each these three operations.

SPPU - Dec. 13, Dec. 14, May 17, Dec.18, 8 Marks

Ans.:

Operations on a Sequential File

- Operations on a sequential file are same as the primitive operations on a file.

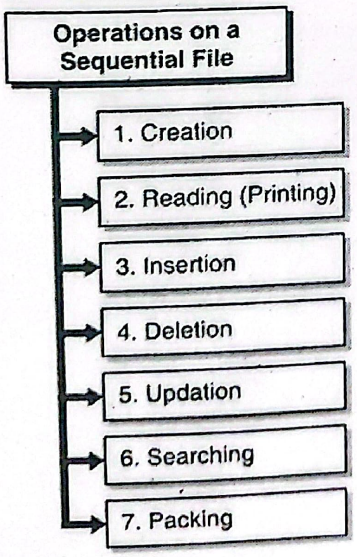


Fig. 6.7 : Operations on a sequential file

- Record are of fixed format and fixed size. For the algorithms, assume the format to be as given as follows :

```
typedef struct student
{   int rollno;
    char name[20];
    float marks;
    int status;/* 0-record is active
                  1-record is deleted by */
} student;
```

- Records are numbered from 0 to n – 1.
- Binary files will be taken for storage.
- Data will be stored in file "MASTER.TXT"
- A temporary file "TEMP.TXT" will be used during deletion.

1. Creation

1. Open the file "master.txt" in "rb" mode. Save the file pointer.
2. If the file pointer is NULL then create the file "master.txt" using "wb" mode. Save the file pointer.

C-Pseudo code for creation

```
void create()
{ FILE *master;
  if(!(master=fopen("master.txt","rb")))
    master=fopen("master.txt","wb");
  close(master);
}
```

2. Reading (Printing)

1. Open the file "master.txt" in "rb" mode. Save the file pointer.
2. Find, number of records in the file.

$$\text{Number of records} = \frac{\text{size of the file}}{\text{size of a record}}$$

3. For each record in the file
   Print the record
4. Close the file "master.txt"
5. Stop.

C-pseudo code for reading (Printing)

```
void read()
{ FILE *master;
```

```
student crec;
int i=1,n;
master=fopen("master.txt","r+b");
fseek(master,0,2);/*go to the end of file */
n=ftell(master)/sizeof(student);
rewind(master);
 for(i=1;i<=n;i++)
  { fread(&crec,sizeof(student),1,master);
   if(crec.status==0)
    printf("\n%d)
    %d\t%s\t%7.2f",i,crec.rollno,crec.name,crec.marks);
   else
    printf("\n%d)    ****** deleted *********",i);
  }
}
```

3. Insertion

1. Open the file "master.txt" in "r + b" mode. Save the file pointer.
2. rec1 ← new record to be inserted.
   (read a record to be inserted)
3. Calculate number of records in the file.

$$n = \frac{\text{size of the file}}{\text{size of the record}}$$

4. If n is 0
   then

```
{ write the record rec1 to the file
  close the file "master.txt"
  goto step 7
}
```

5. Locate the point of insertion.
   k ← point of insertion.
6. for i = n – 1 to k
   move i^{th} record to $(i + 1)^{th}$ position.
7. Write the record rec1 at the k^{th} position.
8. Close the file "master.txt"
9. Stop.

C-pseudo code for insertion

```
void insert(student rec1)
{   FILE *master;
    student crec;
    int n,i,k;
    master=fopen("master.txt","r+b");
    rec1.status=0;
    fseek(master,0,2);/*go to the end of file */
    n=ftell(master)/sizeof(student);
    if(n==0)
    {
        fwrite(&rec1,sizeof(student),1,master);
fclose(master);
    return;
    }
    /* Shift records until the point of insertion */
    i=n-1;
    while(i>=0)
    {   fseek(master,i*sizeof(student),0);
        flushall();
        fread(&crec,sizeof(student),1,master);
        if(crec.rollno>rec1.rollno)
    {   fseek(master,(i+1)*sizeof(student),0);
        fwrite(&crec,sizeof(student),1,master);
        }
        else
        break;
        i--;
    }
    flushall();
    /*insert the record at (i+1)th position */
    i++;
    printf("\ni=%d",i);
    fseek(master,i*sizeof(student),0);
    fwrite(&rec1,sizeof(student),1,master);
    fclose(master);
}
```

4. Deletion

1. Open the file "master.txt" in "r + b" mode. Save the file pointer.
2. Read "rollno" of the record to be deleted.
3. $n = \frac{\text{size of the file}}{\text{size of the record}}$, i = 0
4. if i = n
   then
   [Print "record to be deleted, does not exist'
   close the file "master.txt"
   terminate.
   ]
5. crec ← read the ith record from the file "master.txt"
6. if crec.rollno < rollno or crec.status = 0
   then
   { i = i + 1
   "go to step 4" }
7. if crec.rollno > rollno
   then
   print "record to be deleted does not exist"
   else
   { crec.status = 1
   go to the i^{th} record and write the record crec. If physical deletion is required then the file "master.txt" should be packed after deletion.
   Close the file "master.txt".
   Terminate.
   }
8. Stop.

C-pseudo code for deletion

```
int Delete(int rollno)
{  FILE *master;
   student crec;
   int i,n;
   master=fopen("master.txt","r+b");
   fseek(master,0,2);/*go to the end of file */
   n=ftell(master)/sizeof(student);
   rewind(master);
   for(i=0;i<n;i++)
   { fread(&crec,sizeof(student),1,master);
   if(crec.status==0)
   {
      if(crec.rollno>rollno)
      {printf("\nRecord does not exist ...");
      close(master);
      return(0);
   }
   if(crec.rollno==rollno)
   {crec.status=1;
   fseek(master,i*sizeof(student),0);
   fwrite(&crec,sizeof(student),1,master);
   fclose(master);
   return(1);
   }
   }
}
}
```

Q.10 Explain Searching operations carried out on sequential file. Write pseudo code. **(6 Marks)**

Ans.:

Searching operations

1. Open the file "master.txt" in "r + b" mode. Save the file pointer.
2. Read the "rollno" of the student to be searched.
3. $i = 0, n = \frac{\text{size of the file}}{\text{size of the record}}$
4. if i = n
   then
   { the record does not exist
   terminate with failure.
   }
5. crec ← read the i^{th} record from the file "master.txt"
6. if crec.rollno < rollno or crec.status = 0
   then
   { i = i + 1
   go to step 4
   }
7. if crec.rollno > rollno
   then

```
        { the record does not exist
          terminate with failure
        }
      if crec.rollno = rollno
      then
        { the record is found
        terminate with success.
        }
```

8. Stop.

C-pseudo code for searching

```
int search(int rollno)
{FILE *master;
 student crec;
 int i,n;
  master=fopen("master.txt","r+b");
 fseek(master,0,2);/*go to the end of file */
 n=ftell(master)/sizeof(student);
 rewind(master);
  for(i=0;i<n;i++)
   { fread(&crec,sizeof(student),1,master);
    if(crec.status==0)
     {
     if(crec.rollno>rollno)
      {fclose(master);
       return(-1);
      }
     if(crec.rollno==rollno)
     { fclose(master);
       return(i);
      }
    }
  }
  return(-1);
}
```

Q.11 Explain advantages of indexing over sequential file. SPPU - Dec. 19, 2 Marks

Ans.:

Advantages of Indexing over Sequential File

1. Sequential file can be searched effectively on ordering key. When it is necessary to search for a record on the basis of some other attribute than the ordering key field, sequential file representation is inadequate. Multiple indices can be maintained for each type of field to be used for searching. Thus, indexing provides much better flexibility.
2. An index file usually requires less storage space than the main file. A binary search on sequential file will require accessing of more blocks. This can be explained with the help of the following example.

   Consider the example of a sequential file with r = 1024 records of fixed length with record size R = 128 bytes stored on disk with block size B = 2048 bytes.

   Number of blocks 'b' required to store the file $= \frac{1024 \times 128}{2048} = 64$

   Number of block accesses for searching a record $= \log_2^{64} \approx 6$.

   Suppose, we want to construct an index on a key field that is V = 4 bytes long and the block pointer is P = 4 bytes long. A record of an index file is of the form $<V_i, P_i>$ and it will need 8 bytes per entry.

   Total Number of index entries = 1024

   Number of blocks needed for the index is hence

   $$b_i = \frac{1024 \times 8}{2048} = 4 \text{ blocks}$$

   Binary search on the index will require $\log_2^4 = 2$ block accesses.
3. With indexing, new records can be added at the end of the main file. It will not require movement of records as in the case of sequential file. Updation of index file requires fewer block accesses compare to sequential file.

Q.12 Write a short note on : Hashing. **(5 Marks)**

Ans.: Hashing

- It is a common technique used for fast accessing of records on secondary storage. Records of a file are divided among buckets.

- A bucket is either one disk block or clusture of contiguous blocks. A hashing function maps a key into a bucket number. The buckets are numbered 0, 1, 2 ... b – 1.
- A hash function f maps each key value into one of the integers 0 through b – 1. If x is a key, f(x) is the bucket number that contains the record with key x. The blocks making up each bucket could either be contiguous blocks or they can be chained together in a linked list.
- Translation of bucket number to disk block address is done with the help of bucket directory. It gives the address of the first block of the chained blocks in a linked list. This arrangement is shown in the Fig. 6.8

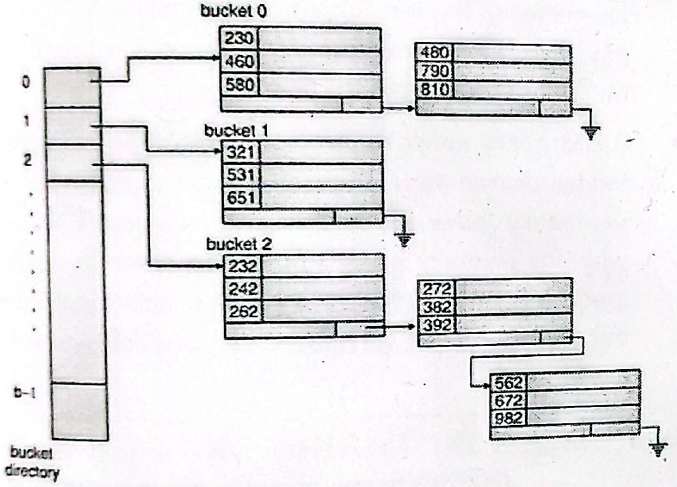


Fig. 6.8 : Hashing with buckets of chained blocks

- Hashing is quite efficient in retrieving a record on hashed key. The average number of block accesses for retrieving a record.

$$= 1 \text{ (bucket directory)} + \frac{\text{No. of records}}{\text{No. of buckets} \times \text{No. of records per block}}$$

- Thus the operation is b times faster (b = number of buckets) than unordered file.
- To insert a record with key value x, the new record can added to the last block in the chain for bucket f(x). If the record does not fit into the existing block, record is stored in a new block and this new block is added at the end of the chain for bucket f(x).
- A well designed hashed structure requires two block accesses for most operations. If the hashing function is good and the number of buckets is equal to the number of records in the file divided by the blocking factor (no. of records per block) then the average bucket will contain one block.

Q.13 Enlist types of indices. Explain any two.

SPPU - Dec. 18, Dec. 19, 5 Marks

Ans.: Types of Indices

The conventional techniques for indexing, include :

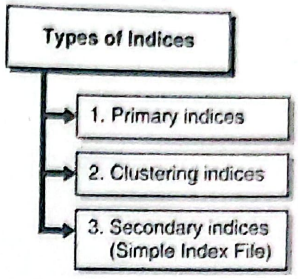


Fig. 6.9 : Types of indices

1. Primary Indices (Indexed Sequential File)

- An indexed sequential file is characterized by
  a) Sequential organization (ordered on primary key)
  b) Indexed on primary key
- An indexed sequential file is both ordered and indexed. Records are organized in sequence based on a key field, known as primary key. An index to the file is added to support random access.
- Each record in the index file consists of two fields : a key field, which is the same as the key field in the main file. Number of records in the index file is equal to the number of blocks in the main file (data file) and not equal to the number of records in the main file (data file).
- To create a primary index on the ordered file shown in the Fig. 6.9(a), we use the rollno field as primary key. Each entry in the index file has rollno value and a block pointer. The first three index entries are as follows.

  <101, address of block 1>

  <201, address of block 2>

  <351, address of block 3>
- Total number of entries in index is same as the number of disk blocks in the ordered data file.
- The index file for primary index requires very few blocks compare to blocks in data file.
- Fewer index entries than number of records, one entry per block.
- Index entry is smaller in size, it has only two fields.

- A binary search on the index file requires very few block accesses.

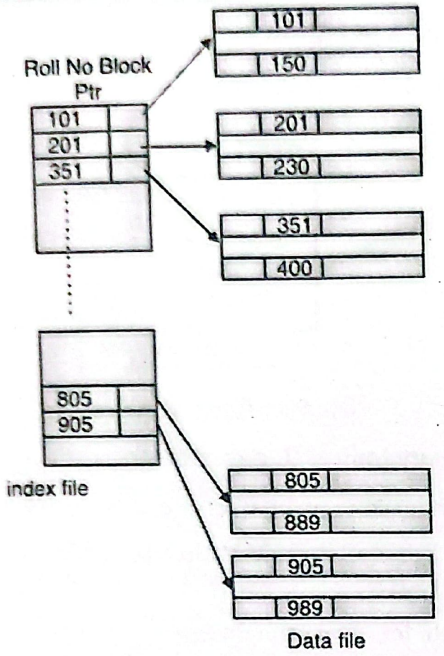


Fig. 6.9(a) : Primary index on the ordering key field roll number

2. Clustering Indices

- If records of a file are ordered on a nonkey field, we can create a different type of index known as **clustering index**. A nonkey field does not have distinct value for each record.
- A clustering index is also an ordered file with two fields :

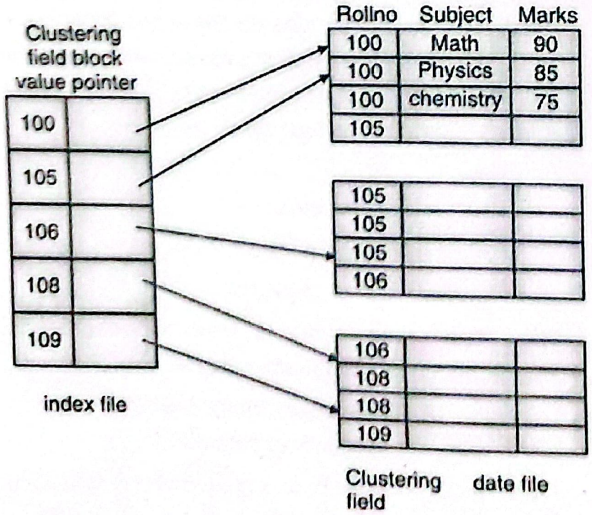


Fig. 6.9(b) : Example of clustering index on roll no

3. Secondary Indices (Simple Index File)

- While the hashed, sequential and indexed sequential files are suitable for operations based on ordering key or the hashed key. Above file organizations are not suitable for operations involving a search on a field other than ordering or hashed key. If searching is required on various keys, secondary indices on these fields must be maintained.
- A secondary index is an ordered file with two fields.
  1. Some non-ordering field of the data file.
  2. A block pointer
- There could be several secondary indices for the same file. One could use binary search on index file as entries of the index file are ordered on secondary key field. Records of the data files are not ordered on secondary key field.
- A secondary index requires more storage space and longer search time than does a primary index. A secondary index file has an entry for every record whereas primary index file has an entry for every block in data file. There is a single primary index file but the number of secondary indices could be quite a few.

Q.14 Explain linked organization with respect to file handling. **SPPU - May 16, May 17, Dec. 18, 6 Marks**

Ans.:

Linked Organization of a File

- In linked organization, logical sequence of records is different from physical sequence. In a linked organization the next logical record is obtained by following a link value from the present record. It is like a linked list.
- Linking records together in order of increasing primary key value facilitates easy insertion and deletion once the place at which the insertion or to be made is known.
- Searching for a record with given primary key value is difficult when no index is available. Only, linear search can be used.
- To facilitate searching on the primary key as well as on secondary keys it is necessary to maintain several indices, one for each key.

- An employee number index, for instance, may contain entries corresponding to ranges of employee numbers.
- One possibility, for example of Fig. 6.10(a) would be to have an entry for each of the ranges 501-700, 701-900 and 901-1100. All records giving E # in the same range will be linked together as in Fig. 6.10(b). Using an index in this way reduces the length of the lists and thus the search time.

	E#	Name	Occupation	Degree	Sex	Location	MS	Salary
A	800	HAWKINS	Programmer	BS	M	Los Angles	S	10,000
B	510	WILLIAMS	Analyst	BS	F	Los angles	M	15,000
C	950	FRAWLEY	Analyst	MS	F	Minneapolis	S	12,000
D	750	AUSTIN	Programmer	BS	F	Los angles	S	12,000
E	620	MESSER	Programmer	BS	M	Minneapolis	M	9,000

Fig. 6.10(a) : Sample data for employee file

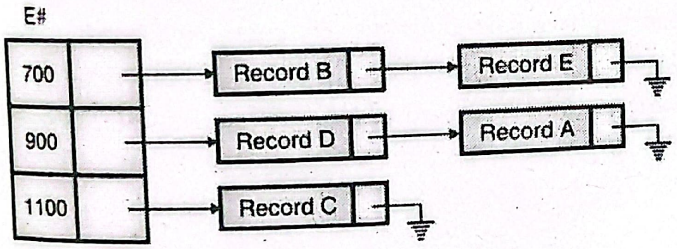
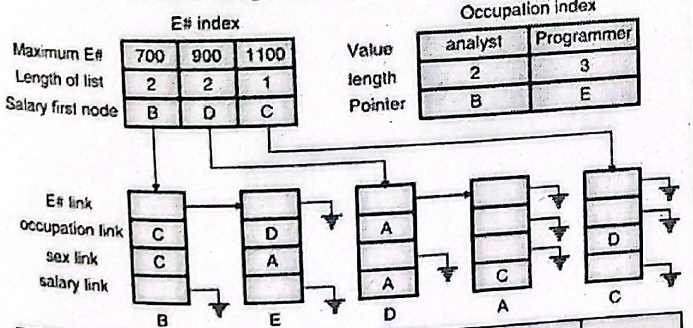


Fig. 6.10(b) : Index on E# , Record in the same range are linked together



Value	Female	Male
Length	3	2
Pointer	B	E

Value	≤ 9000	≤ 12000	≤ 15000
Length	1	3	1
Pointer	E	A	B

Fig. 6.10(c) : Multilist representation for Fig. 6.10(a)

- We can set up indices for each key and allow records to be in more than one list. Such a structure is known as multilist structure.
- Fig. 6.10(c) shows the indices and lists corresponding to multilist representation of data of Fig. 6.10(a).
- Each index entry contains key values pointer to list and the length of the associated list.

Q.15 Explain linked organization with respect to inverted files. **SPPU - May 16, Dec. 19, 7 Marks**

Ans. :

Inverted File Organisation

- Conceptually, inverted files are similar to multilists. The difference is that while in multilists records, With the same key values are linked together.
- Information about link is kept in individual records. In case of inverted files this link information is kept in the index itself. Fig. 6.11 shows the indices for the file.

E# index	
700	B,E
900	A,D
1100	C

Salary Index	
900	E
1200	A,C,D
1500	B

Occupation index	
Analyst	B,C
Programmer	A,D,E

Sex index	
Female	B,C,D
Male	A,E

Fig. 6.11 : Indices for fully inverted file

- In inverted files, only the index structure is important. Record can be stored in any way.
- Inverted files may also result in space saving when record retrieval does not require retrieval of key fields.
- In case of inverted files, the key fields may be deleted from the records.
- Insertion and deletion of records requires only the ability to insert and delete within indices.
- Index maintenance is more complex than for multilist.
- Number of disk accesses to process a query is equal to the number of records being retrieved plus the number to process the indices.

Q.16 Define coral rings. **(2 Marks)**

Ans.: Coral Rings

- The coral ring structure is similar to the doubly linked multilist structure. Back pointers are added to carry out a deletion without having to start at the front of each list containing the record being deleted. A same coral ring in shown in the Fig. 6.12.

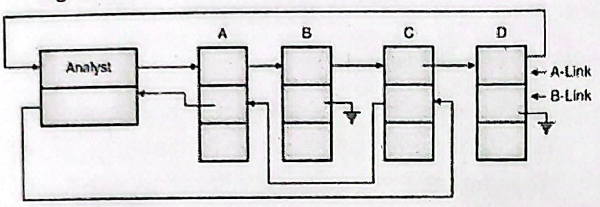


Fig. 6.12 : A sample coral ring

- Forward circular list contains A, B, C, D.

  Reverse circular list contains C, A

  Each node has two link fields :

  1. A-link 2. B-link
- The A-link field is used to link together all records with the same value of the key k_i.
- The head node retains the value of key k_i for the records in the ring.
- The BLINK field connects alternate records in the reverse order.

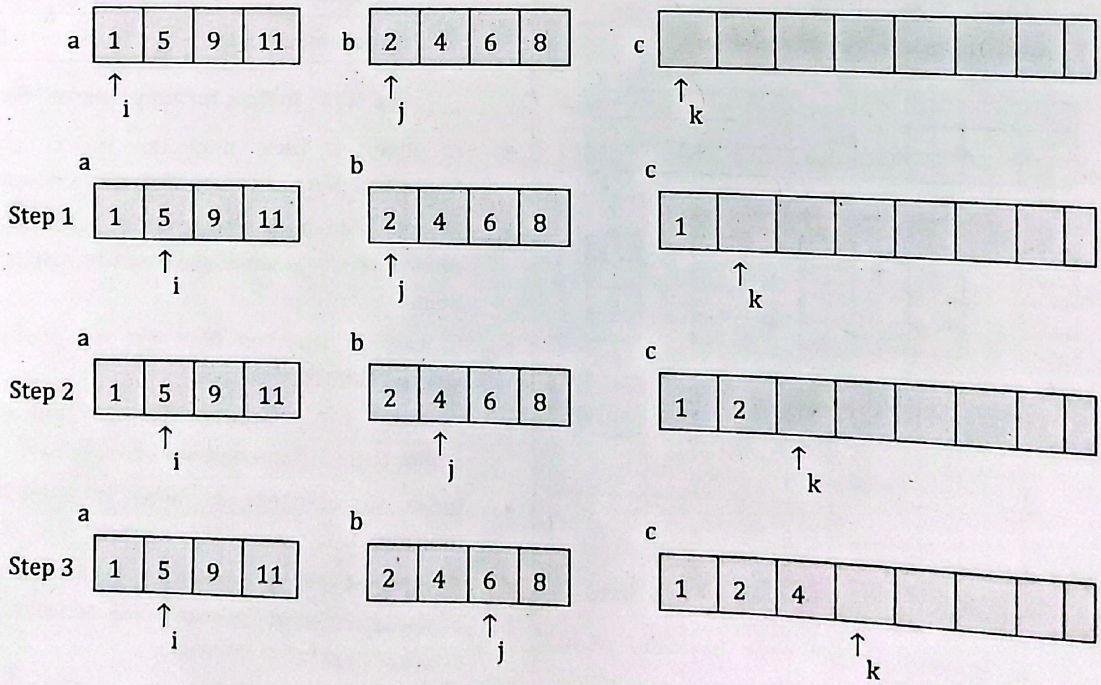
Q.17 Explain merging of two sorted list . **(5 Marks)**

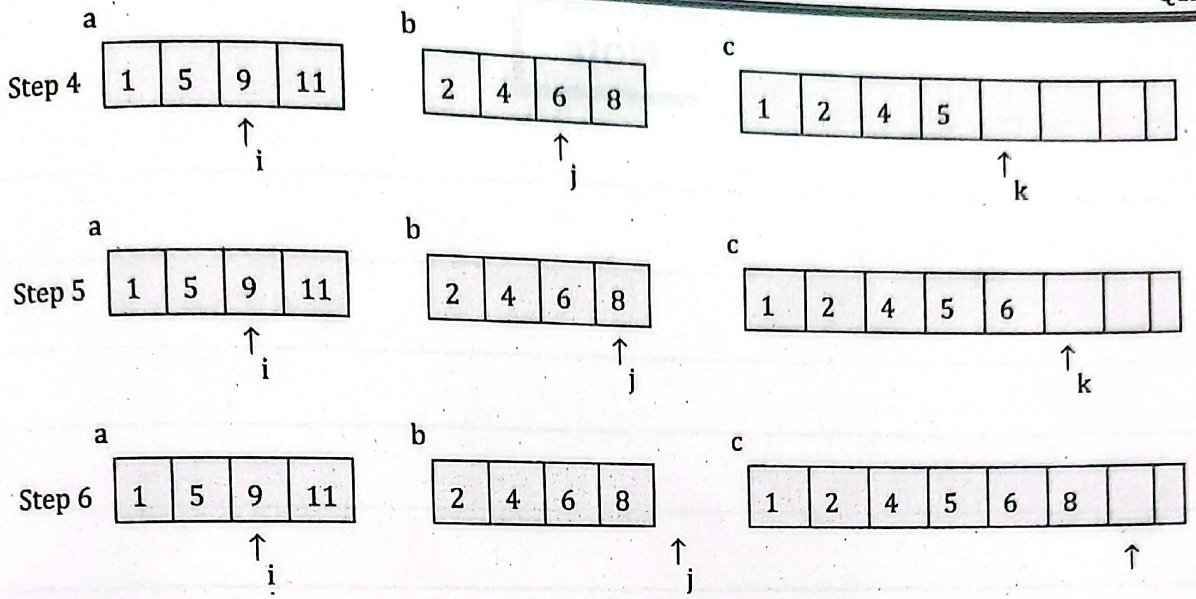
Ans.:

Merging of Two Sorted List

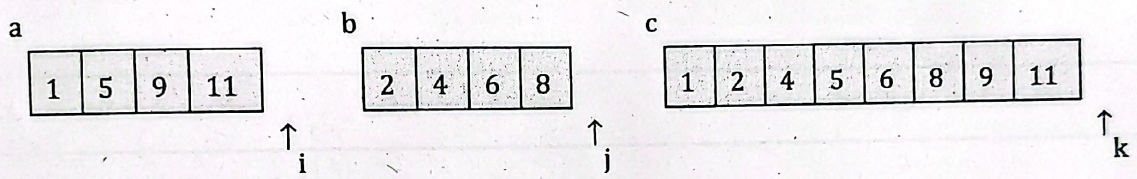
The fundamental operation in merge sort algorithm is merging two sorted arrays. The merging algorithm takes two sorted arrays a[] and b[] as input and the third array c[] as output. Three variables i, j and k are initially set to the beginning of the three arrays a[], b[] and c[]. The smaller of a[i] and b[j] is copied to c[k] and appropriate variables (i, k) or (j, k) are advanced. When either of the input array a[] or b[] is exhausted, the remainder of the other array is copied to c.

Initial conditions





Array b[] is exhausted, the remainder of the array a[] are added to c[].



Q.18 Write pseudo code for two-way merge sort.

SPPU - May 19, 6 Marks

Ans.:

K-way Merge Algorithm

A K-way merge algorithm can be used to merge K-sorted lists into a single sorted list. Let us consider a case, where there are K-files. These files are sorted on name. The algorithm reads the input files file1, file 2,..., file k and produces the output file outfile. It is assumed that the input files do not have duplicates or out of sequence records.

K-way merge

```
while (more –names)
  {
    out_name = min(next name in the sequence)
    write(outfile, out_name)
    if (name1==out_name)  then
        read(file1, name1)
    if(name2==out_name)  then
        read(file2, name2)
    :
    if(namek==out_name) then
        read(filek, namek)
  }
```

Steps for merging K sorted arrays using min-heaps

1. Create an output array of size n * K.
2. Create a min heap of size K by inserting 1st element in all arrays into the heap.
3. Repeat following steps n * K times.
   (a) Delete the minimum element from the heap and insert the same in the output array.
   (b) Insert in the heap, the next element from the array from which the element is extracted.
4. Display the output array.

❑❑❑