Deshpande Sakshi Mahesh

SPPU New Syllabus

A Book Of

DATA STRUCTURES AND ALGORITHMS-II

For S.Y.B.Sc. Computer Science : Semester – IV (Paper – I)

[Course Code CS 241 : Credits - 2]

CBCS Pattern

As Per New Syllabus, Effective from June 2020

Dr. Ms. Manisha Bharambe

M.Sc. (Comp. Sci.), M.Phil. Ph.D. (Comp. Sci.)

Vice Principal, Associate Professor, Department of Computer Science

MES's Abasaheb Garware College

Pune

Price ₹ 190.00

NIRALI PRAKASHAN

ADVANCEMENT OF KNOWLEDGE

N5541

DATA STRUCTURES & ALGORITHMS-II ISBN 978-93-90506-31-6

First Edition : January 2021

Published By :
NIRALI PRAKASHAN
Abhyudaya Pragati, 1312, Shivaji Nagar,
Off J.M. Road, Pune – 411005
Tel - (020) 25512336/37/39, Fax - (020) 25511379
Email : niralipune@pragationline.com

Polyplate

Printed By :
YOGIRAJ PRINTERS AND BINDERS
Survey No. 10/1A, Ghule Industrial Estate
Nanded Gaon Road
Nanded, Pune - 411041
Mobile No. 9404233041/9850046517

DISTRIBUTION CENTRES

PUNE

Nirali Prakashan (For orders within Pune) : 119, Budhwar Peth, Jogeshwari Mandir Lane, Pune 411002, Maharashtra
Tel : (020) 2445 2044; Mobile : 9657703145
Email : niralilocal@pragationline.com

Nirali Prakashan (For orders outside Pune) : S. No. 28/27, Dhayari, Near Asian College Pune 411041
Tel : (020) 24690204; Mobile : 9657703143
Email : bookorder@pragationline.com

MUMBAI

Nirali Prakashan : 385, S.V.P. Road, Rasdhara Co-op. Hsg. Society Ltd.,
Girgaum, Mumbai 400004, Maharashtra; Mobile : 9320129587
Tel : (022) 2385 6339 / 2386 9976, Fax : (022) 2386 9976
Email : niralimumbai@pragationline.com

DISTRIBUTION BRANCHES

JALGAON

Nirali Prakashan : 34, V. V. Golani Market, Navi Peth, Jalgaon 425001, Maharashtra,
Tel : (0257) 222 0395, Mob : 94234 91860; Email : niralijalgaon@pragationline.com

KOLHAPUR

Nirali Prakashan : New Mahadvar Road, Kedar Plaza, 1st Floor Opp. IDBI Bank, Kolhapur 416 012
Maharashtra. Mob : 9850046155; Email : niralikolhapur@pragationline.com

NAGPUR

Nirali Prakashan : Above Maratha Mandir, Shop No. 3, First Floor,
Rani Jhanshi Square, Sitabuldi, Nagpur 440012, Maharashtra
Tel : (0712) 254 7129; Email : niralinagpur@pragationline.com

DELHI

Nirali Prakashan : 4593/15, Basement, Agarwal Lane, Ansari Road, Daryaganj
Near Times of India Building, New Delhi 110002 Mob : 08505972553
Email : niralidelhi@pragationline.com

BENGALURU

Nirali Prakashan : Maitri Ground Floor, Jaya Apartments, No. 99, 6th Cross, 6th Main,
Malleswaram, Bengaluru 560003, Karnataka; Mob : 9449043034
Email: niralibangalore@pragationline.com

Other Branches : Hyderabad, Chennai

niralipune@pragationline.com | www.pragationline.com
Also find us on www.facebook.com/niralibooks

Preface ...

I take an opportunity to present this Text Book on **"Data Structures and Algorithms-II"** to the students of Second Year B.Sc. (Computer Science) Semester-IV as per the New Syllabus, June 2020.

The book has its own unique features. It brings out the subject in a very simple and lucid manner for easy and comprehensive understanding of the basic concepts. The book covers theory of Tree, Efficient Search Trees, Graph and Hash Table.

A special word of thank to Shri. Dineshbhai Furia, and Mr. Jignesh Furia for showing full faith in me to write this text book. I also thank to Mr. Amar Salunkhe and Mr. Akbar Shaikh of M/s Nirali Prakashan for their excellent co-operation.

I also thank Ms. Chaitali Takle, Mr. Ravindra Walodare, Mr. Sachin Shinde, Mr. Ashok Bodke, Mr. Moshin Sayyed and Mr. Nitin Thorat.

Although every care has been taken to check mistakes and misprints, any errors, omission and suggestions from teachers and students for the improvement of this text book shall be most welcome.

Author

Syllabus ...

1. Tree **(10 Hrs.)**

1.1 Concept and Terminologies

1.2 Types of Binary Trees - Binary Tree, Skewed Tree, Strictly Binary Tree, Full Binary Tree, Complete Binary Tree, Expression Tree, Binary Search Tree, Heap

1.3 Representation - Static and Dynamic

1.4 Implementation and Operations on Binary Search Tree - Create, Insert, Delete, Search, Tree Traversals - Preorder, Inorder, Postorder (Recursive Implementation), Level-Order Traversal using Queue, Counting Leaf, Non-Leaf and Total Nodes, Copy, Mirror

1.5 Applications of Trees

1.5.1 Heap Sort, Implementation

1.5.2 Introduction to Greedy Strategy, Huffman Encoding (Implementation using Priority Queue)

2. Efficient Search Trees **(8 Hrs.)**

2.1 Terminology: Balanced Trees - AVL Trees, Red Black Tree, Splay Tree, Lexical Search Tree -Trie

2.2 AVL Tree - Concept and Rotations

2.3 Red Black Trees - Concept, Insertion and Deletion

2.4 Multi-Way Search Tree - B and B+ Tree - Insertion, Deletion

3. Graph **(12 Hrs.)**

3.1 Concept and Terminologies

3.2 Graph Representation - Adjacency Matrix, Adjacency List, Inverse Adjacency List, Adjacency Multi-list

3.3 Graph Traversals - Breadth First Search and Depth First Search (With Implementation)

3.4 Applications of Graph

3.4.1 Topological Sorting

3.4.2 Use of Greedy Strategy in Minimal Spanning Trees (Prim's and Kruskal's Algorithm)

3.4.3 Single Source Shortest Path - Dijkstra's Algorithm

3.4.4 Dynamic Programming Strategy, All Pairs Shortest Path - Floyd Warshall Algorithm

3.4.5 Use of Graphs in Social Networks

4. Hash Table **(6 Hrs.)**

4.1 Concept of Hashing

4.2 Terminologies - Hash Table, Hash Function, Bucket, Hash Address, Collision, Synonym, Overflow etc.

4.3 Properties of Good Hash Function

4.4 Hash Functions: Division Function, Mid Square, Folding Methods

4.5 Collision Resolution Techniques

4.5.1 Open Addressing - Linear Probing, Quadratic Probing, Rehashing

4.5.2 Chaining - Coalesced, Separate Chaining

■■■

Contents ...

■■■

CHAPTER 1

Tree

Objectives ...

- To study Basic Concepts of Tree Data Structure
- To learn Basic Concepts of Binary Tree and its Types
- To study Representation of Tree
- To understand Binary Search Tree (BST)
- To study the Applications of Tree

1.0 INTRODUCTION

- A tree is a non-linear data structure. A non-linear data structure is one in which its elements do not forms a sequence.
- A tree data structure is a widely used Abstract Data Type (ADT) that simulates a hierarchical tree structure, with a root value and subtrees of children with a parent node, represented as a set of linked nodes.
- A tree data structure stores the data elements in a hierarchical manner. A tree is a hierarchical data structure that consists of nodes connected by edges.
- Let us see an example of a directory structure stored by operating system. The operating system organizes files into directories and subdirectories. This can be viewed as tree shown in Fig. 1.1.

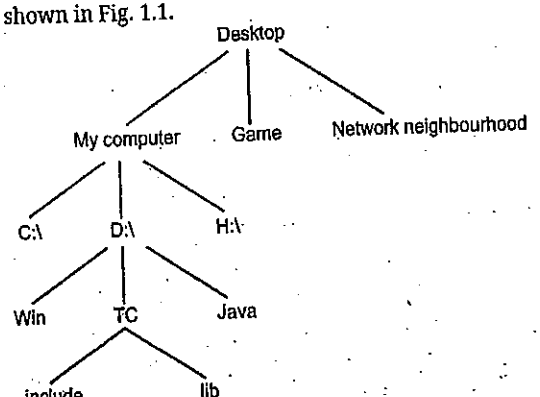


Fig. 1.1: Tree Representation of Directory Structure

- Common uses for tree data structure are given below:
  1. To manipulate hierarchical data.
  2. To make information easily searchable.
  3. To manipulate sorted lists of data.
- A tree data structure widely used for improving database search time, game programming, 3D graphics, data compression and in file systems.

1.1 BASIC CONCEPTS AND TERMINOLOGY

- A tree is a non-linear data structure used to represent the hierarchical structure of one or more data elements, which are known as **nodes** of the tree.
- Each node of a tree stores a data value and has zero or more pointers pointing to the other nodes of the tree which are known as its child nodes.
- Each node in a tree can have zero or more **child nodes**, which is at one level below it. However, each child node can have only one **parent node**, which is at one level above it.
- The node at the top of the tree is known as the **root** of the tree and the nodes at the lowest level are known as the **leaf nodes**.
- Fig. 1.2 shows a tree, in which G node have no child. The nodes without any child node are called **external nodes** or **leaf nodes**, whereas, the nodes having one or more child nodes are called **internal nodes**.
- **Siblings** represent the collection of all of the child nodes to one particular parent. An **edge** is the route between a parent and child node.
- A **subtree** of a tree is a tree with its nodes being a descendent of some other tree.

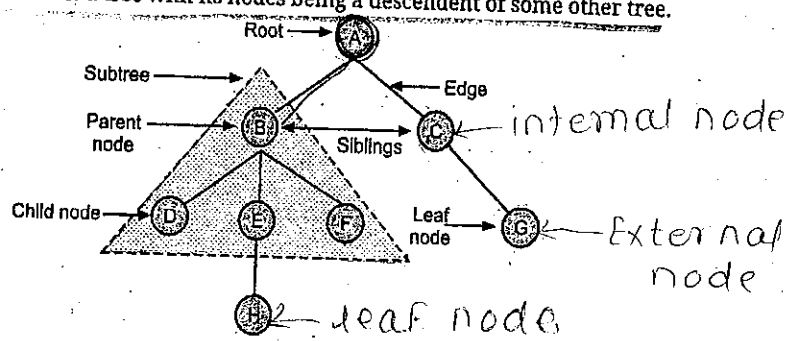


Fig. 1.2: Structure of Tree

1.1.1 Definition

- A tree may be defined as, a finite set 'T' of one or more nodes such that there is a node designated as the root of the tree and the other nodes (excluding the root) are divided into $n \geq 0$ disjoint sets $T_1, T_2, \ldots T_n$ and each of these sets is a tree in turn. The trees $T_1, T_2 \ldots T_n$ are called the sub-trees or children of the root.

- The Fig. 1.3 (a) shows the empty tree, there are no nodes. The Fig. 1.3 (b) shows the tree with only one node. The tree in Fig. 1.3 (c) has 12 nodes.

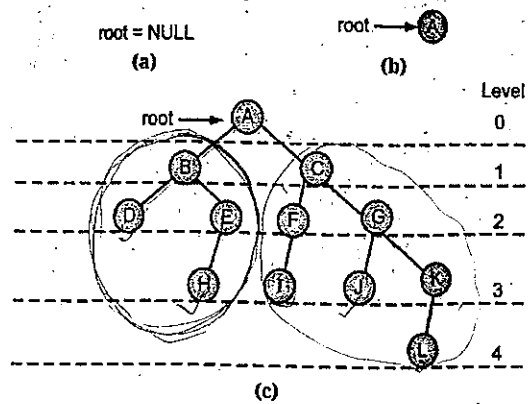


Fig. 1.3: Examples of Tree

- The root of tree is A, it has 2 subtrees. The roots of these subtrees are called the children of the root A.
- The nodes with no subtrees are called terminal node or leaves. There are 5 leaves in the tree of Fig. 1.3 (c).
- Because family relationship can be modeled as trees, we often call the root of a tree (or subtree) the parent and the nodes of the subtrees the children; the children of a node are called siblings.

1.2 Operations on Trees

- Various operations on tree data structure are given below:
  1. **Insert:** An insert operation allows a new node to be added or inserted as a child of an existing node in the tree.
  2. **Delete:** The delete operation will remove a specified node from the tree.
  3. **Search:** The search operation searches an element in a tree.
  4. **Prune:** The prune operation in tree will remove a node and all of its descendants from the tree. Removing a whole selection of a tree called pruning.
  5. **Graft:** The graft operation is similar to insert operation except, that the node being inserted has descendants of its own, meaning it is a multilayer tree. Adding a whole section to a tree called grafting.
  6. **Enumerate:** An enumeration operation will return a list or some other collection containing every descendant of a particular node, including the root node itself.
  7. **Traversal:** Traversal means to visit all nodes in a binary tree but only once.

1.1.3 Terminology

- A tree consists of following terminology:

1. **Node:** In a tree, every individual element is called as node. Node in a tree data structure, stores the actual data of that particular element and link to next element in hierarchical structure. Fig. 1.3 (c) has 12 nodes.
2. **Root Node:** Every tree must have a root node. In tree data structure, the first node from where the tree originates is called as a root node. In any tree, there must be only one root node.
3. **Parent Node:** In tree data structure, the node which is predecessor of any node is called as parent node. Parent node can be defined as, "the node which has child/children". In simple words, the node which has branch from it to any other node is called as parent node.
4. **Child Node:** In tree data structure, the node which is descendant of any node is called as child node. In simple words, the node which has a link from its parent node is called as child node. In tree data structure, all the nodes except root node are child nodes.
5. **Leaf Node:** The node which does not have any child is called as a leaf node. Leaf nodes are also called as external nodes or terminal nodes.
6. **Internal Node:** In a tree data structure, the leaf nodes are also called as external nodes. External node is also a node with no child. In a tree, leaf node is also called as terminal node.
7. **Edge:** In tree data structure, the connecting link between any two nodes is called as edge. In a tree with 'n' number of nodes there will be a maximum of 'n-1' number of edges.
8. **Path:** In tree data structure, the sequence of nodes and edges from one node to another node is called as path between that two nodes. Length of a path is total number of nodes in that path.
9. **Siblings:** Nodes which belong to the same parent are called as siblings. In other words, nodes with the same parent are sibling nodes. [Oct. 17]
10. **Null Tree:** A tree with no nodes is called as a null tree (Refer Fig. 1.3 (a)).
11. **Degree of a Node:** Degree of a node is the total number of children of that node. The degree of A is 2, degree of K is 1 and degree of L is 0, (Refer Fig. 1.3 (c)).
12. **Degree of a Tree:** The highest degree of a node among all the nodes in a tree is called as degree of tree. In Fig. 1.3 (c), degree of the tree is 2. [April 18]
13. **Depth or Level of a Node:** In tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on. In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one (1) at each level (step).
14. **Descendants:** The descendents of a node are those nodes which are reachable from node. In Fig. 1.3 nodes J, K, L are descendents of G.

15. **Ancestor:** The ancestor of a node are all the nodes along the path from the root to that node. In Fig. 1.3 nodes A and C are ancestor of G.
16. **In-degree:** The in-degree of a node is the total number of edges coming to that node.
17. **Out-degree:** The out-degree of a node is the total number of edges going outside from the node.
18. **Forest:** A forest is a set of disjoint trees.
19. **Sub Tree:** In tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.
20. **Height:** In tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as height of that node. In a tree, height of the root node is said to be height of the tree. In a tree, height of all leaf nodes is '0'.
21. **Depth:** Total number of edges from root node to a particular node is called as depth of that node. In tree, depth of the root node is '0'. The tree of Fig. 1.3 (c), has depth 4.

1.2 BINARY TREE AND TYPES OF BINARY TREE

- In this section we will study binary tree and its types.

1.2.1 Binary Tree

- A tree in which every node can have a maximum of two children is called as binary tree.
- Binary tree is a special type of tree data structure in which every node can have a maximum of two children. One is known as left child and the other is known as right child.
- Fig. 1.4 represents binary tree in which node A has two children B and C. Each child has one child namely D and E respectively.

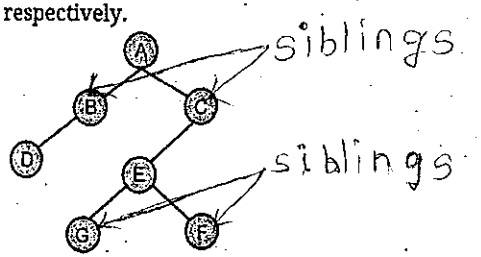
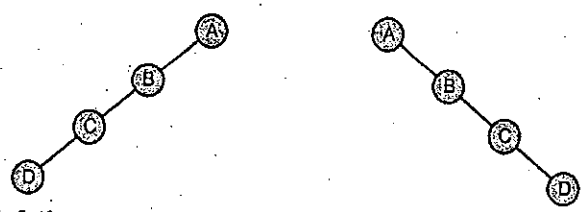


Fig. 1.4: Binary Tree

1.2.2 Skewed Binary Tree [April 17, 19]

- A tree in which every node has either only left subtree or right subtree is called as skewed binary tree.
- The tree can be left skewed tree or right skewed tree (See Fig. 1.5).



(a) Left Skewed Binary Tree (b) Right Skewed Binary Tree

Fig. 1.5: Skewed Binary Tree

1.2.3 Strictly Binary Tree

- A strictly binary tree is a binary tree where all non-leaf nodes have two branches.
- When every non-leaf node in binary tree is filled with left and right sub-trees, the tree is called strictly binary tree.

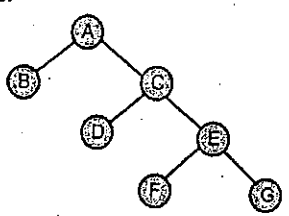


Fig. 1.6: Strictly Binary Tree of Height 3

1.2.4 Full Binary Tree

- If each node of binary tree has either two children or no child at all, is said to be a full binary tree.
- A full binary tree is defined as a binary tree in which all nodes have either zero or two child nodes.
- A full binary tree is a binary tree in which all of the leaves are on the same level and every non-leaf node has two children (See Fig. 1.7).

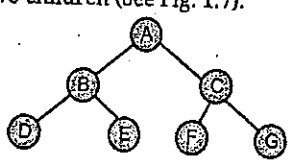


Fig. 1.7: Full Binary Tree

1.2.5 Complete Binary Tree [Oct. 18]

- A binary tree is said to be complete binary tree, if all its level, except the last level, have maximum number of possible nodes, and all the nodes of the last level appear as far left as possible.
- A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

- Fig. 1.8 shows is a complete binary tree.

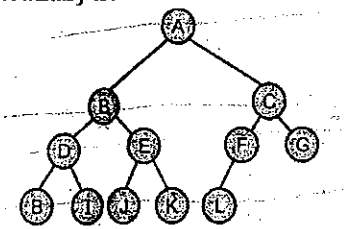
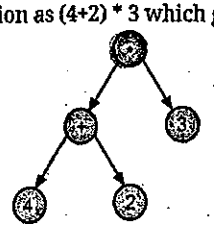


Fig. 1.8: Complete Binary Tree

1.2.6 Expression Tree

- Binary tree representing an arithmetic expression is called expression tree. The leaves of expression trees are operands (variables or constants) and interior nodes are operators.
- Expression tree is a binary tree in which each internal node corresponds to operator and each leaf node corresponds to operand.
- Expression tree is a tree which represent an expression where leaves are labeled with operands of the expression and nodes other than leaves are labeled with operators of the expression.
- A binary expression tree is a specific kind of a binary tree used to represent expressions.
- Consider the expression tree in the Fig. 1.9, what expression this tree represents? what is the value of expression?
- As per the properties of expression tree, always expressions are solved from bottom to up.so first expression we will get is 4+2, then this expression will be multiplied by 3. Hence, we will get the expression as (4+2) * 3 which gives the result as 18.



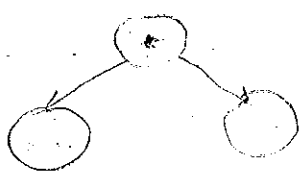


Fig. 1.9

1.2.7 Binary Search Tree

- A binary search tree is a binary tree in which the nodes are arranged according to their values.
- The left node has a value less than its parent and the right node has a value greater than the parent node.
- Hence, all nodes in the left subtree have values less than the root and the nodes in the right subtree have values greater than the root.

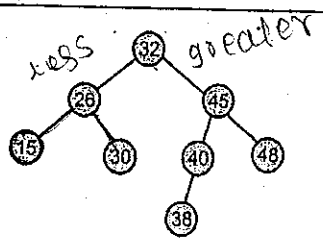
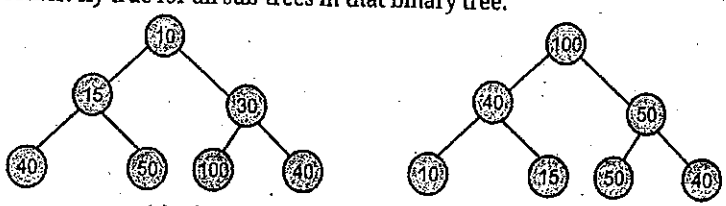


Fig. 1.10: Binary Search Tree

1.2.8 Heap

- A heap is a special tree-based data structure in which the tree is a complete binary tree.
- Generally, heaps can be of following two types:
  1. **Max-Heap:** In a max-heap the key present at the root node must be greatest among the keys present at all of its children. The same property must be recursively true for all sub-trees in that binary tree.
  2. **Min-Heap:** In a min-heap the key present at the root node must be minimum among the keys present at all of its children. The same property must be recursively true for all sub-trees in that binary tree.



(a) Min Heap (b) Max Heap

Fig. 1.11: Binary Search Tree

1.3 REPRESENTATION OF TREE (STATIC AND DYNAMIC)

- Tree data structure can be represented in following two ways:
  1. Using an array (sequential/linear/static representation).
  2. Using a linked list (link/dynamic representation).
- We shall give more emphasis to linked representation as is more popular than the corresponding sequential structure. The two main reasons are:
  1. A tree has a natural implementation in linked storage.
  2. The linked structure is more convenient for insertions and deletions.

1.3.1 Static Representation of Tree (Using Array)

- In static representation, tree is represented sequentially in the memory by using single one-dimensional array.
- In static representation of tree, a block of memory for an array is to be allocated before going to store the actual tree in it.

- Hence, nodes of the tree are stored level by level, starting from the zero level where the root is present.
- The root node is stored in the first memory location as the first element in the array.
- Static representation of tree needs sequential numbering of the nodes starting with nodes on level zero, then those on level 1 and so on.
- A complete binary tree of height h has $(2^{h+1}-1)$ nodes in it. The nodes can be stored in one dimensional array. A array of size $(2^{h+1}-1)$ or $2^d - 1$ (where d = no. of levels) is required.
- Following rules can be used to decide the location of any i^{th} node of a tree:

  For any node with index i, where $1 \le i \le n$

  (a) PARENT (i) = $\left\lfloor \frac{i}{2} \right\rfloor$ if $i \neq 1$

  If i = 1 then it is root which has no parent

  (b) LCHILD (i) = 2 * i, if $2i \le n$

  If $2i > n$, then i has no left child

  (c) RCHILD(i) = 2i + 1, if $2i + 1 \le n$

  If $(2i + 1) > n$, then i has no right child.
- Let us, consider complete binary tree in the Fig. 1.12.

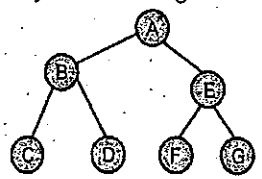


Fig. 1.12

- The representation of the above tree in Fig. 1.12 using array is given in Fig. 1.13.

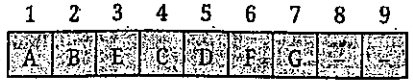


Fig. 1.13

- The parent of node i is at location (i/2)

  Example, Parent of node D = (5/2) =2 i.e. B
- Left child of a node i is present at position 2i + 1 if $2^i + 1 < n$

  Left child of node B = $2^i = 2*2 = 4$ i.e. C
- Right child of a node i is present at position $2^i + 1$

  Right child of E = $2^i+1 = (2*3) +1 = 6+1=7$ i.e. G.

Examples:

Example 1: Consider the complete binary tree in Fig. 1.14.

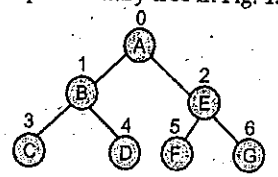


Fig. 1.14

In Fig. 1.14,

Number of levels = 3 (0 to 2) and height = 2

Therefore, we need the array of size $2^3 - 1$ or $2^{2+1} - 1$ is $2^3 - 1 = 7$.

The representation of the above binary tree using array is as followed:

1	2	3	4	5	6	7
A	B	C	D	E	F	G

We will number each node of tree starting from the root. Nodes on same level will be numbered left to right.

Example 2: Consider almost complete binary tree in Fig. 1.15.

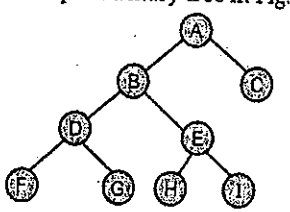


Fig. 1.15

Here, depth = 4 (level), therefore we need the array of size $2^4 - 1 = 15$. The representation of the above binary tree using array is as follows:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	E	–	–	F	G	H	I	–	–	–	–

level 0 1 2 3

We can apply the above rules to find array representation.

1. Parent of node E (node 5) = $\frac{i}{2} = \frac{5}{2} = 2$ i.e. B.

   Hence, node B is at position 2 in the array.

2. Left (i) = 2^i.

   For example, left of E = 2 × 5 = 10 i.e. H.

   Since, E is the 5th node of tree.

3. Right (i) = $2^i + 1$

For example: Right of D = 2 × 4 + 1 = 8 + 1 = 9 i.e. G.

Since, D is the 4th node of tree and G is the 9th element of an array.

Example 3: Consider the example of skewed tree.

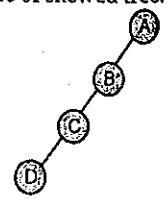


Fig. 1.16: Skewed Binary Tree

The tree has following array representation.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	–	C	–	–	–	D	–	–	–	–	–	–	–

We need the array of size $2^4 - 1 = 15$.

Thus, only four positions are occupied in the array and 11 are wasted.

From the above example it is clear that binary tree using array representation is easier, but the method has certain drawbacks. In most of the representation, there will be lot of unused space. For complete binary trees, the representation is ideal as no space is wasted.

But for the skewed binary tree (See Fig. 1.16), less than half of the array is used and the more is left unused. In the worst case, a skewed binary tree of depth k will require $2^k - 1$ locations of array and occupying only few of them.

Advantages of Array Representation of Tree:

1. In static representation of tree, we can access any node from other node by calculating index and this is efficient from the execution point of view.
2. In static representation, the data is stored without any pointers to their successor or ancestor.
3. Programming languages like BASIC, FORTRAN etc., where dynamic allocation is not possible, use array representation to store a tree.

Disadvantages of Array Representation of Tree:

1. In static representation the number of the array entries are empty.
2. In static representation, there is no way possible to enhance the tree structure. Array size cannot be changed during execution.
3. Inserting a new node to it or deleting a node from it are inefficient for this representation, (here, considerable data movement up and down the array happens, thus more/excessive processing time is used).

Program 1.1: Program for static (array) representation of tree (Converting a list of array in to binary tree).

```
#include<stdio.h>
typedef struct node
{
    struct node*left;
    struct node*right;
    char data;
}node;
node* insert(char c[],int n)
{
    node*tree=NULL;
    if(c[n]!='\0')
 {
    tree=(node*)malloc(sizeof(node));
    tree->left=insert(c,2*n+1);
    tree->data=c[n];
    tree->right=insert(c,2*n+2);
 }
    return tree;
}
/*traverse the tree in inorder*/
void inorder(node*tree)
{
   if(tree!=NULL)
   {
      inorder(tree->left);
      printf("%c\t",tree->data);
      inorder(tree->right);
   }
}
void main()
{
   node*tree=NULL;
   char c[]={'A','B','C','D','E','F','\0','G','\0','\0','\0','\0','\0',
                                 '\0','\0','\0','\0','\0','\0','\0','\0'};
   tree=insert(c,0);
   inorder(tree);
}
```

Output:

```
G    D    B    E    A    F    C
```

1.3.2 Dynamic Representation of Tree (Using Linked List)

- Another way to represent a binary tree is using a linked list. In a linked list representation, every node consists of three fields namely, left child (Lchild), data and right child (Rchild).
- Fig. 1.17 shows node structure in linked list representation of tree.
- Linked list representation of tree, is more memory efficient than the array representation. All nodes should be allocated dynamically.
- In dynamic representation of tree, a block of memory required for the tree need not be allocated beforehand. They are allocated only on demand.

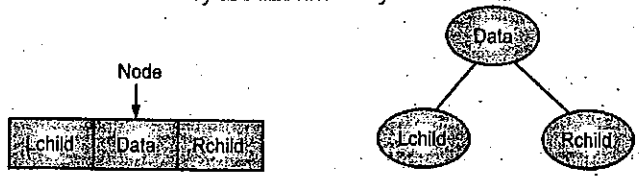
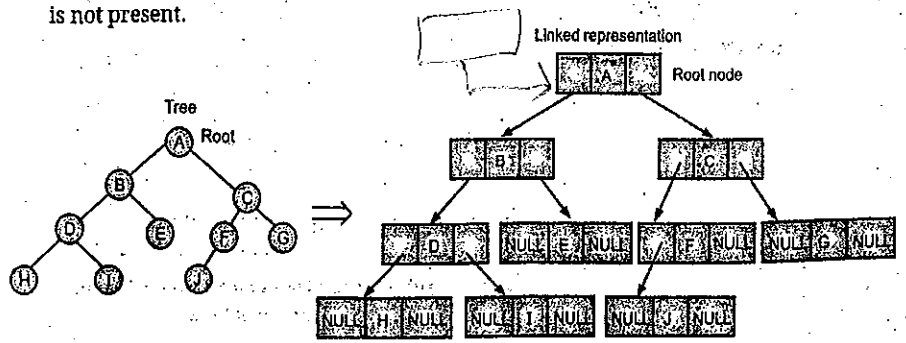
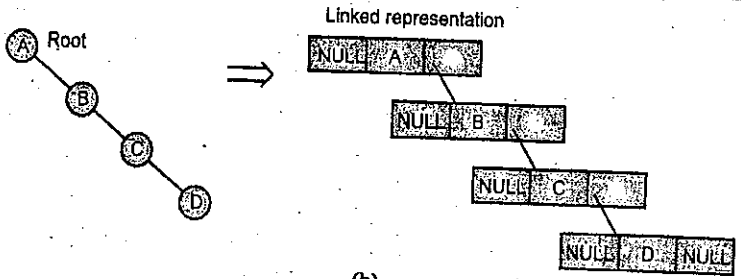


Fig. 1.17: Fields of a Node of a Binary Tree in Linked Representation

- A node in linked in a linked representation of tree has two pointers (left and right) fields, one for each child. When a node had no children, the corresponding pointer fields are NULL.
- The data field contains the given values. The Lchild field holds the address of its left node and the Rchild field holds the address of right node.
- The Fig. 1.18 (a) and (b) shows the linked representation of binary tree.
- In Fig. 1.18 (a) and (b), NULL stored at Lchild and Rchild field represent that respective is not present.



(a)



(b)

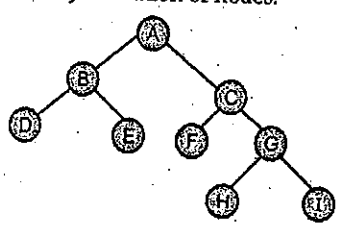
Fig. 1.18: Linked Representation of Binary Tree

- Each node represents information (data) and two links namely, Lchild and Rchild are used to store addresses of left child and right child of a node.

Declaration in 'C': We can define the node structure as follows:

```
struct tnode
{
    struct tnode *left;
    <datatype> data;    /* data type is a data type of field data */
    struct tnode *right;
};
```

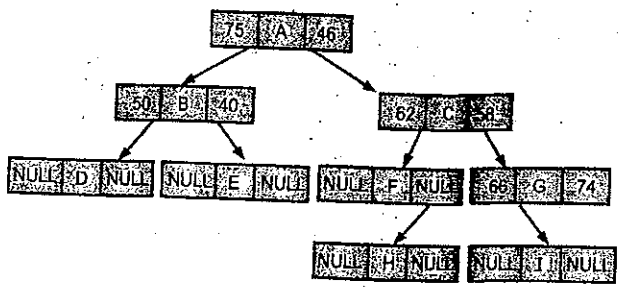
- Using this declaration for linked representation, the binary tree representation can be logically viewed as in Fig. 1.19 (c). In Fig. 1.19 (b) physical representation shows the memory allocation of nodes.



(a) A Binary Tree

Address	left	node data	right
50	NULL	D	NULL
75	50	B	40
40	NULL	E	NULL
89	75	A	46
62	NULL	F	NULL
46	62	C	58
66	NULL	H	NULL
58	66	G	74
74	NULL	I	NULL

(b) A Binary Tree and its various Nodes (Physical View)



(c) Logical View

Fig. 1.19: Linked Representation of Binary Tree

Advantages of Linked Representation of Binary Tree:

1. Efficient use of memory than that of static representation.
2. Insertion and deletion operations are more efficient.
3. Enhancement of tree is possible.

Disadvantages of Linked Representation of Binary Tree:

1. If we want to access a particular node, we have to traverse from root to that node; there is no direct access to any node.
2. Since, two additional pointers are present (left and right), the memory needed per node is more than that of sequential/static representation.
3. Programming languages not supporting dynamic memory management are not useful for dynamic representation.

Program 1.2: Program for dynamic (linked list) representation of tree.

```
#include <stdio.h>
#include <malloc.h>
struct node {
      struct node * left;
      char data;
      struct node * right;
      };
      struct node *constructTree( int );
      void inorder(struct node *);
      char array[] = {'A', 'B', 'C', 'D', 'E', 'F', 'G', '\0', '\0', 'H'};
      int leftcount[] = {1,   3,   5,   -1,   9,  -1,  -1,   -1,   -1,  -1};
      int rightcount[] = {2,   4,   6,   -1,  -1,  -1,  -1,   -1,   -1,  -1};
      void main() {
          struct node *root;
          root = constructTree( 0 );
```

```
        printf("In-order Traversal: \n");
        inorder(root);
    }
    struct node * constructTree( int index ) {
        struct node *temp = NULL;
        if (index != -1) {
            temp = (struct node *)malloc( sizeof ( struct node ) );
            temp->left = constructTree( leftcount[index] );
            temp->data = array[index];
            temp->right = constructTree( rightcount[index] );
        }
        return temp;
    }
    void inorder( struct node *root ) {
        if (root != NULL) {
            inorder(root->left);
            printf("%c\t", root->data);
            inorder(root->right);
        }
    }
```

Output:

```
In-order Traversal:
D    B    H    E    A    F    C    G
```

- The primitive operations on binary tree are as follows:
  1. **Create:** Creating a binary tree.
  2. **Insertion:** To insert a node in the binary tree. Insertion operation is used to insert a new node into a binary tree. Fig. 1.20 shows the insertion of node with data G as a left of a node having data 'D'. The Insertion involves two steps:
     (i) Search for a node in the given binary tree after which insertion is to be made.
     (ii) Create a link for a new node and new node becomes either left or right.

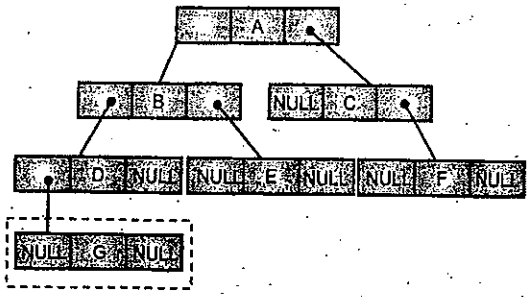


Fig. 1.20: Insertion of Node G

3. **Deletion:** To delete a node from the binary tree. Deletion operation deletes any node from any non-empty binary tree. In order to delete a node in a binary tree, it is required to reach at the parent node of the node to be deleted. The link field of the parent node which stores the address of the node to be deleted is then set by a NULL entry as shown in Fig. 1.21.

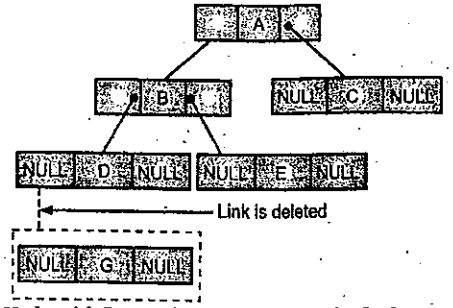


Fig. 1.21: Node with Data G is Deleted and Left of D becomes NULL

4. **Traversal:** Binary tree traversal means visiting every node exactly once. There are three methods of traversing a binary tree. These are called as inorder, postorder and preorder traversals.

1.4 BINARY SEARCH TREE (BST) IMPLEMENTATION AND OPERATIONS

- Consider an example where we want to search a data from the linked list. We have to search sequentially which is slower than binary search. If the list is ordered list stored in contiguous sequential storage, binary search is faster.
- If we want to insert or delete a data from the list, then in sequential list more data movements are required. With link list only few pointer manipulations are required.
- Binary search tree provides quick insertion and deletion operation. Hence, binary trees provide an excellent solution for searching, inserting and deleting a node.
- The Binary Search Tree (BST) is a binary tree with the property that the value in a node is grater than any value in a node's left subtree and less than any value in the node's right subtree.
- There are two ways to represent binary search tree i.e., static and dynamic. **Static representation** of binary search tree in which the set of values in the nodes is known in advance and in **dynamic representation**, the values in a tree may change over time.

Definition of BST:

- A Binary Search Tree (BST) is a binary tree which is either empty or non-empty. If it is non-empty, then every node contains a key which is distinct and satisfies the following properties:
  1. Values less than its parent are placed at left side of parent node.
  2. Values greater than its parent are placed at right side of parent node.
  3. The left and right subtrees are again binary search trees.

- Fig. 1.22 shows examples of binary search trees.

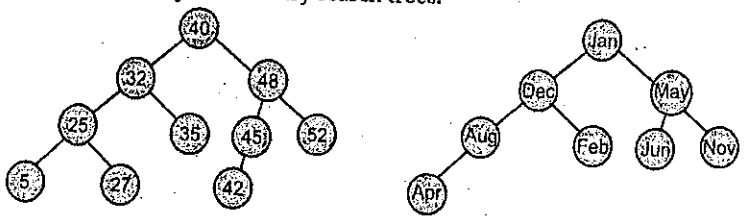


Fig. 1.22: Binary Search Trees

1.4.1 Operations on Binary Search Tree

- Following operations are commonly performing on BST,
  1. Search a key
  2. Insert a key
  3. Delete a key
  4. Traverse the tree.

1.4.1 Creating Binary Search Tree

[April 16, 18, 19, Oct. 17]

- The function create() simply creates an empty binary search tree. This initializes the root pointer to NULL.

```
struct tree * root = NULL;
```

- The actual tree is build through a call to insert BST function, for every key to be inserted.

Algorithm:

1. root = NULL
2. read key and create a new node to store key value
3. if root is null then new node is root
4. t = root, flag = false
5. while (t ≠ null) and (flag = false) do
   - **case 1:** if key < t → data
     attach new node to t → left
   - **case 2:** if key > t → data
     attach new node to t → right
   - **case 3:** if t → data = key
     then print "key already exists"
6. Stop.

C Function for Creating BST:

```
BSTNODE createBST(BSTNode *root)
{
   BSTNODE *newnode, *temp;
   char ans;
   do
   {
      newnode=(BSTNODE *) malloc(sizeof(BSTNODE));
      printf("\n Enter the element to be inserted:");
      scanf("%d", &newnode→data);
      newnode→left=newnode→right=NULL;
      if(root==NULL)
         root=newnode;
      else
      {
         temp=root;
         while(temp!=NULL)
         {
            if(newnode→data<temp→data)
            {
               if(temp→left==NULL)
               {
                  temp→left=newnode;
                  break;
               }
                  else
               temp=temp→left;
               }
               else if(newnode→data>temp→data)
               {  if(temp→right==NULL)
                  {
                     temp→right=newnode;
                     break;
                  }
                  else
                     temp=temp→right;
               }
         }
         printf("\n Do u want to add more numbers?");
         scanf("%c", &ans);
      }  while(ans=='y' || ans =='y')
      return(root);
   }
```

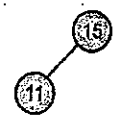
Example: Construct Binary Search Tree (BST) for the following elements:

15, 11, 13, 8, 9, 17, 16, 18

1.

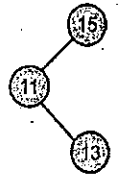
Key = 15

2.



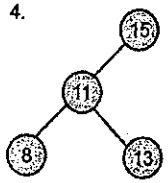
Key < 15
Key = 11

3.



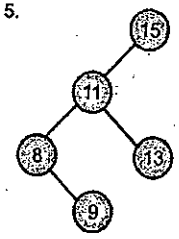
Key = 13

4.



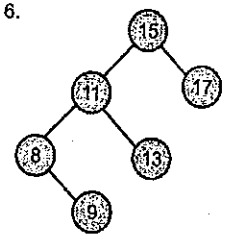
Key = 8

5.



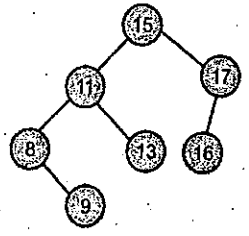
Key = 9

6.



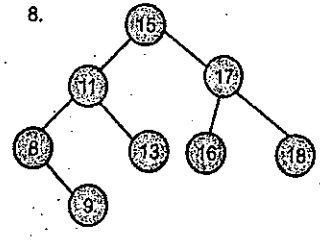
Key = 17

7.



Key = 16

8.



Key = 18

- Here, all values in the left subtrees are less than the root and all values in the right subtrees are greater than the root.

Program 1.3: Program to create a BST.

```
#include<stdio.h>
#include<malloc.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node *right;
   struct node *left;
};
```

left data right


```
struct node *create(struct node *, int);
void postorder(struct node *);
int main()
{
   struct node *root = NULL;
   setbuf(stdout, NULL);
   char ch;
   int item;
   root = NULL;
   do{
      printf("\nEnter data for node ");
      scanf("%d",&item);
      root=create(root,item);
      printf("Do you want to insert more elements?");
      scanf(" %c",&ch);  //use a space before %c to clear stdin
   }while(ch=='y'||ch=='Y');
   printf("****BST created***\n");
}
struct node *create(struct node *root, int item)
{
   if(root == NULL)
   {
      root = (struct node *)malloc(sizeof(struct node));
      root->left = root->right = NULL;
      root->data = item;
      return root;
   }
   else
   {
      if(item < root->data )
      {
         printf("%d inserted left of %d\n",item,root->data);
         root->left = create(root->left,item);
      }
      else if(item > root->data )
      {
         printf("%d inserted right of %d\n",item,root->data);
         root->right = create(root->right,item);
      }
      else
         printf(" Duplicate Element is Not Allowed !!!");
      return(root);
   }
}
```

Output:

```
Enter data for node 10
Do you want to insert more elements?y
Enter data for node 20
20 inserted right of 10
Do you want to insert more elements?y
Enter data for node 5
5 inserted left of 10
Do you want to insert more elements?y
Enter data for node 15
15 inserted right of 10
15 inserted left of 20
Do you want to insert more elements?y
Enter data for node 2
2 inserted left of 10
2 inserted left of 5
Do you want to insert more elements?
```

1.4.2 Searching in a BST [April 17]

- To search a target key, we first compare it with the key at the root of the tree. If it is the same, then the search ends and if it is less than key at root, search the target key in left subtree else search in the right subtree.
- **Example:** Consider BST in the Fig. 1.23.

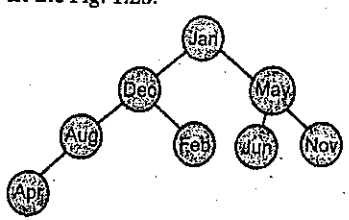


Fig. 1.23: Binary Search Tree

- To search 'Apr', we first compare 'Apr' with key of root 'Jan'. Since, Apr < Jan (alphabetical order) we move to left and next compare with 'Dec'. Since, Apr < Dec, we move to the left again and compare with 'Aug'.
- As Apr < Aug we move to the left. Here we find the key 'Apr' and search is successful. If not, we continue searching until we hit an empty subtree.
- We can return the value of the pointer to send the result of the search function back to the caller function.

Procedure: Search – BST (Key)

Steps:

1. Initialize t = root, flag = false
2. while (t ≠ null) and (flag = false) do
   - **case 1:** t· data = key
     flag = true /*successful search*/
   - **case 2:** Key < t → data
     t = t → left /* goto left subtree*/
   - **case 3:** Key > t → data
     t = t → right /*goto right subtree*/

   end case
   end while
3. if (flag = true) then
   display "Key is found at node", t
   else
   display "Key is not exist"
   end if;
4. Stop

The Non Recursive C Function for search key:

```
BSTNODE *search (BSTNODE *root, int key)
{
   BSTNODE *temp=root;
   while (temp != NULL) && (temp ··data != NULL)
   {
      if (key < temp → data)
      temp = temp → left;
      else
      temp = temp → right;
   }
   return(temp);
}
```

The recursive C function for search key:

```
BSTNODE *research (BSTNODE *root, int key)
{
   BSTNODE *temp=root;
   if (temp == NULL) || (temp → data == key)
      return(temp);
   else
      if (key < temp → data)
         research (temp → left, key);
      else
         research (temp → right, key);
}
```

1.4.1 Inserting a Node into BST

[April 16, 18, Oct. 17, 18]

- We insert a node into binary search tree in such a way that resulting tree satisfies the properties of BST.

Algorithm: Insert_BST (Key)

Steps: 1. t = root, flag = false

2. while (t ≠ null) & (flag = false) do

case 1: key < t → data
t1 = t;
t = t → left;

case 2: key > t → data
t1 = t
t = t → right

case 3: t → data = key
flag = true
display "item already exist"
break

end case

end while

3. if (t = null) then
new = getnode (node) //create node
new → data = key
new → left = null //initialize a node
new → right = null
if (t1 → data < key) then //insert at right
t1 → right = new
else t1 → left = new //insert at left
endif

4. Stop

Recursive C function for inserting node into BST:

```
BSTNODE *Insert-BST (BSTNODE *root, int n)
{
   if (root == NULL)
   {
      root = (BSTNODE *) malloc (sizeof(BSTNODE));
      root → data = n;
      root → left = root → right = NULL;
   }
```



```
   else
   if (n < root → data)
      root → left = Insert-BST (root → left, n);
   else
      root → right = Insert-BST (root → right, n);
   return(root);
}
```

Non-Recursive C function for inserting node into BST:

```
BSTNODE *Insert-BST (BSTNODE *root, int n)
{
   BSTNODE *temp, *newnode;
   newnode = (BSTNODE*) malloc (sizeof(BSTNODE));
   newnode → data = n;
   newnode → left = root → right = NULL;
   if (root==NULL)
      root = newnode;
   else
   {
      temp = root;
      while (temp)
      {
         if (n < temp → data)
         {
            if (temp → left==NULL)
            {
               temp → left = newnode;
               break;
            }
            else
               temp = temp → left;
            else if (n > temp → data)
            {
               if(temp → right==NULL)
               {
                  temp → right = newnode;
                  break;
               }
               else
                  temp = temp → Rchild;}
            }
         }
      return root;
      }
   }
   }
```

- **Example:** Consider the insertion of keys: Heena, Deepa, Tina, Meena, Beena, Anita.
- Initially tree is empty. The first key 'Heena' is inserted, it becomes a root. Since, 'Deepa' < 'Heena', it is inserted at left. Similarly insert remaining keys in such a way that they satisfy BST properties.
- Now, suppose if we want to insert a key 'Geeta'. It is first compared with root. Since, 'Geeta' < 'Heena', search is proceeding to left side. Since, 'Geeta' > 'Deepa' and right of 'Deepa' is null then 'Geeta' is inserted as a right of 'Deepa'.

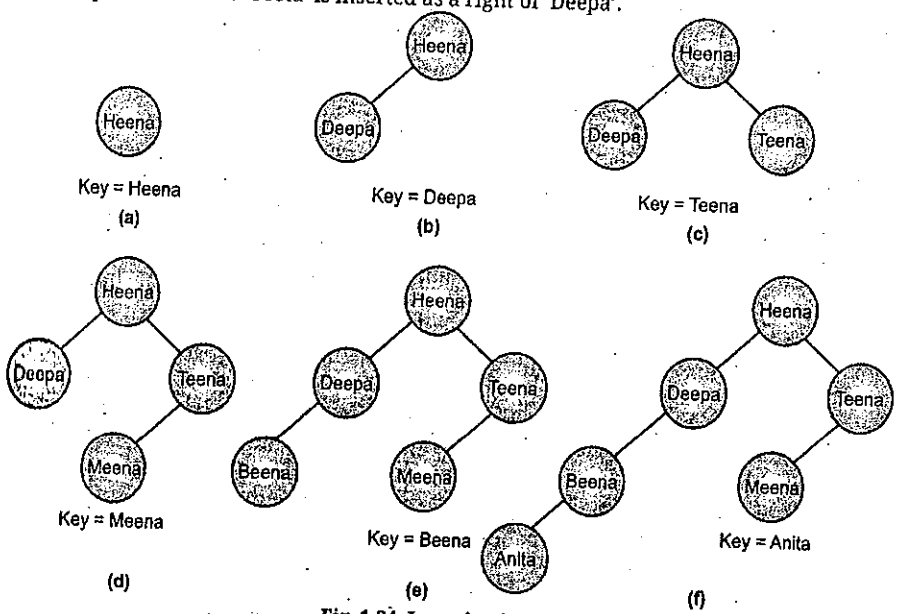


Fig. 1.24: Insertion in BST

1.4.1.4 Deleting Node from BST

- Delete operation is frequently used operation of BST. Let T be a BST and X is the node with key (K) to be deleted from T, if it exists in the tree let y be a parent node of X.
- There are three cases with respective to the node to be deleted:
  1. X is a leaf node.
  2. X has one (either left or right).
  3. X has both nodes.

Algorithm: Delete BST (key)

Steps:

1. t = root, flag = false
2. while (t ≠ null) and (flag = false) do
   - **case 1:** key < t→data
     parent = t
     t = t → left
   - **case 2:** key > t → data
     parent = t
     t = t → right
   - **case 3:** t → data = key
     flag = true

   end case
   end while
3. if flag = false
   then display "item not exist".
   exit.
4. /* case 1 if node has no child */
   if (t → left = null) and (t → right = null) then
   if (parent → left = t) then
   parent → left = null //set pointer to its parent when node is left child
   else
   parent → right = null
5. /* case 2 if node contains one child */
   if (parent → left = t) then //when node contains only one child
   if (t → left = null) then //if node is left child
   parent → left = t → right
   else
   parent → left = t → left
   endif
   else
   if (parent → right = t) then
   if (t → left = null) then
   parent → right = t → right
   else
   parent → right = t → left
   endif
   endif
   endif
6. /* Case 3 if node contains both child */
   t1 = succ(t) //find inorder successor of the node
   key1 = t1 → data
   Delete BST (key1) //delete inorder successor
   t → data = key //replace data with the data of an order successor
7. Stop.

Recursive C function for deletion from BST:

```
BSTNODE *rec_deleteBST(BSTNODE *root, int n)
{
    BSTNODE *temp, *succ;
    if(root==NULL)
    {
        printf("\n Number not found.");
        return(root);
    }
    if(n<root→data)  //deleted from left subtree
        root→left=rec_deleteBST(root→left, n);
    elseif(n>root→data) //deleted from right subtree
        root→right=rec_deleteBST(root→right, n);
    else  //Number to be deleted is found
    {
        if(root→left ! = NULL && root→right ! =NULL)//2 children
        {
            succ=root→right;
            while(succ→left)
            succ=succ→left;
            root→data=succ→data;
            root→right=rec_deleteBST(root→right, succ→data);
        }
        else
        {
            temp=root;
            if(root→left !=NULL)  //only left child
                root=root→left;
            elseif (root→right!=NULL) //only right child
                root=root→right;
            else  //no child
                root=NULL;
            free(temp);
        }
    }
    return(root);
}
```


Non-recursive C function for deleting node from BST:

```
BSTNODE *non_rec_DeleteBST (BSTNODE *root, int n)
{
    BSTNODE *temp, *parent, *child, *succ, *parsucc;
    temp==root;
    parent=NULL;
    while(temp != NULL)
    {
        if (n == temp → data)
            break;
        parent = temp;
        if(n < temp → data)
            temp = temp → left;
        else
            temp = temp → right;
        }
        if(temp == NULL)
        {
            printf("\nNumber not found.");
            return(root);
        }
        if(temp→left !=NULL && temp→right !=NULL)
        // a node to be deleted has 2 children
        {
            parsucc=temp;
            succ=temp→right;
            while(succ→left!=NULL)
            {
                parsucc=succ;
                succ=succ→left;
            }
            temp → data = succ → data;
            temp = succ;
            parent = parsucc;
        }
        if(temp→left! = NULL) //node to be deleted has left child
            child=temp→left;
        else //node to be deleted has right child or no child
            child=temp→right;
```

```
        if(parent==NULL) //node to be deleted is root node
           root=child;
        elseif(temp==parent → left) //node is left child
           parent→left=child;
        else //node is right child
           parent→right=child;
        free(temp);
        return(root);
    }
```

- **Example:** Consider all above three cases. Nodes with double circles indicates node to be deleted.

Case 1: If the node to be deleted is a leaf node, then we only replace the link to the deleted node by NULL.

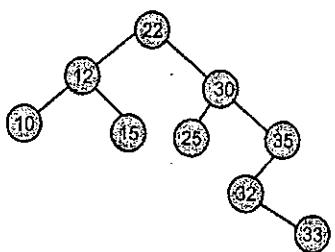


Fig. 1.25: Binary Search Tree

After deletion of leaf node with data 33.

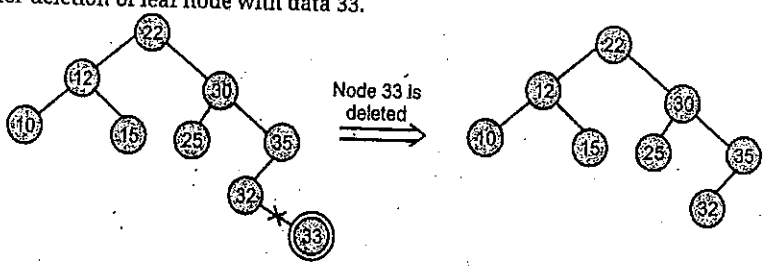


Fig. 1.26: Deletion of node 33

Case 2: If the node to be deleted has a single node, then we adjust the link from parent node to point to its subtree.

Consider tree of Fig. 1.27, delete the node with data ≠ 35 which has only left with data 32.

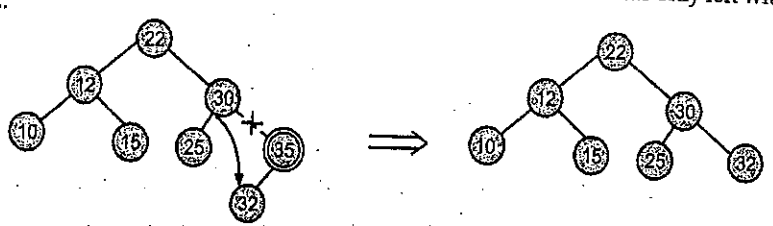


Fig. 1.27: Deletion of Node 35

Case 3: The node to be deleted having both nodes. When the node to be deleted has both non-empty subtrees, the problem is difficult.

One of the solution is to attach the right subtree in place of the deleted node and then attach the left subtree to the appropriate node of right subtree.

From Fig. 1.28, delete the node with data 12.

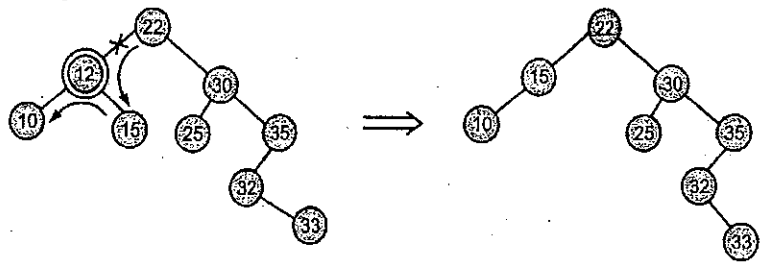


Fig. 1.28: Deletion of Node 12

Pictorial representation is shown in Fig. 1.29.

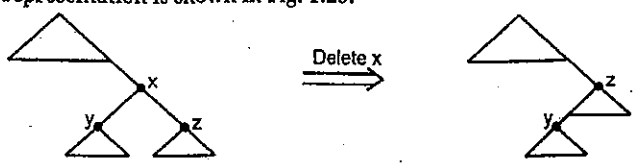


Fig. 1.29: Pictorial Representation

The another approach is to delete x from T by first deleting inorder successor of node x say z, then replace the data content in node x by the data content in node z. Inorder successor means the node which comes after node x during the inorder traversal of T.

- **Example:** Consider 1.30 and delete node with data 15.

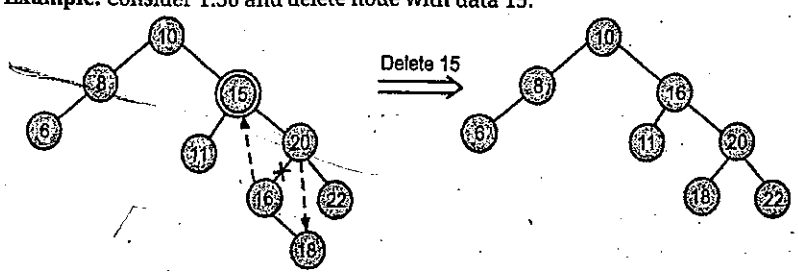


Fig. 1.30: Deletion of Node 15

1.4.2 Compute the Height of a Binary Tree

- Recursive function which returns the height of linked binary tree:

```
int tree_height(struct node *root)
{
    if (root == NULL)
       return 0;
    else
    return (1+ max (tree_height (root → left), tree_height
                                                    (root → right)));
}
```

Here, function max() is defined as follows:

```
int max(int x, int y)
{
   if (x > y)
      return x;
   else
      return y;
}
```

1.4.3 Tree Traversals (Preorder, Inorder and Postorder) [April 16, 19]

- Traversing a binary tree refers to the process of visiting each and every node of the tree exactly once.
- Traversal operation is used to visit each node (or visit each node exactly once). There are two methods of traversal namely, recursive and non-recursive.
  1. **Recursive** is a straight forward approach where the programmer has to convert the definitions into recursions. The entire load is on the language translator to carry out the execution.
  2. **Non-recursive** approach makes use of stack. This approach is more efficient as it requires less execution time.
- In addition, it is good for the programming language which do not support dynamic memory management scheme.
- The sequence in which these entities processed defines a particular traversal method.
- There are three traversal methods i.e. inorder traversal, preorder traversal and post order traversal.
- These traversal techniques are applicable on binary tree as well as binary search tree.

Preorder Traversal (DLR):

- In preorder traversal, the root node is visited before traversing it's the left child and right child nodes.

- In this traversal, the root node is visited first, then its left child and later its right chil
  This preorder traversal is applicable for every root node of all subtrees in the tree.
  (i) Process the root Data (D) (Data)
  (ii) Traverse the left subtree of D in preorder (Left)
  (iii) Traverse the right subtree of D in preorder (Right)
  Preorder ⇒ Data – Left – Right (DLR)

Example:

- A preorder traversal of a tree in Fig. 1.31 (a) visit node in a sequence ABDECF.

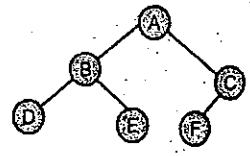


Fig. 1.31 (a): Tree

- For the expression tree, preorder yields a prefix expression.

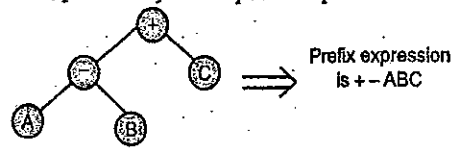


Fig. 1.31 (b): Prefix Expression

- In preorder traversal, we visit a node, traverse left and continue again. When v cannot continue, move right and begin again or move back, until we can move rig and stop. Preorder function can be written as both recursive and non-recursive way.

Recursive preorder traversal:

Algorithm:

Step 1 : begin
Step 2 : if tree not empty
visit the root
preorder (left child)
preorder(right child)
Step 3 : end

C Function for Recursive Preorder Traversal:

```
void preorder (struct treenode * root)
{
   if (root)
   {
      printf("%d \t", root → data);      /*data is integer type */;
      preorder (root → left);
      preorder (root → right);
   }
}
```

Example:

- In the Fig. 1.32, tree contains an arithmetic expression. It gives us prefix expression as, + * – A/BCDE. The preorder traversal is also called as depth first traversal.

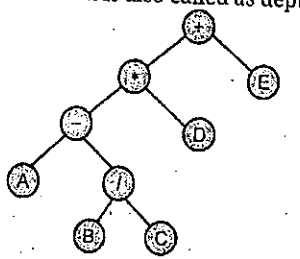


Fig. 1.32: Binary Tree for Expression ((A – B/C) * D) + E

Inorder Traversal (LDR):

- In Inorder traversal, the root node is visited between the left child and right child.
- In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting the right child node.
- This inorder traversal is applicable for every root node of all subtrees in the tree. This is performed recursively for all nodes in the tree.

(i) Traverse the left substree of D in inorder (Left)
(ii) Process of root Data (D) (Data)
(iii) Traverse the right subtree of R in inorder (Right)
Inorder ⇒ Left – Data – Right (LDR)

Algorithm for Recursive Inorder Traversal:

Step 1 : begin
Step 2 : if tree is not empty
inorder (left child)
visit the root
inorder (right child)
Step 3 : end

'C' function for Inorder Traversal:

- The in order function calls for moving down the tree towards the left until, you can go on further. Then visit the node, move one node to the right and continue again.

```
void inorder (tnode * root)
{
   if(root)
   {
      inorder (root → left);
      printf("%d", root → data);
      inorder (root → right);
   }
}
```

- This traversal is also called as symmetric traversal.

Postorder Traversal (LRD):

- In Postorder traversal, the root node is visited after left child and right child. In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited.

(i) Traverse the left subtree of D in postorder (Left)
(ii) Traverse the right subtree of D in postorder (Right)
(iii) Process of root Data (D) (Data)
Postorder ⇒ Left – Right – Data (LRD)

Algorithm for Recursive Postorder Traversal:

Step 1: begin
Step 2: if tree not empty
postorder (left child)
postorder(right)
visit the root
Step 3: end

'C' Function for Postorder Traversal:

```
void postorder (tnode * root)
  {
     if (root)
     {
        postorder(root → left);
        postorder(root → right);
        printf("%d", root → data);
     }
  }
```

Example: Traverse each of the following binary tree in inorder, preorder and postorder.

(a)

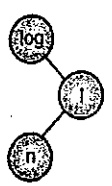


Fig. 1.33

Traversals: **Preorder Traversal:** log ! n
Inorder Traversal: log n!
Postorder Traversal: n! log

(b)

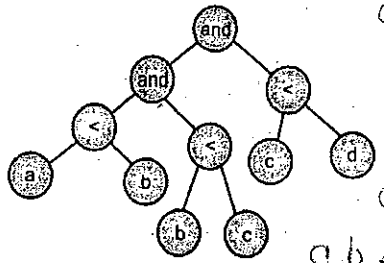


Fig. 1.34

Traversals: **Preorder Traversal:** and and < a b < b c < c d
Inorder Traversal: a < b and b < c and c < d
Postorder Traversal: a b < b c < and c d < and

(c)

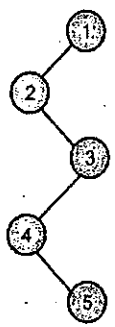


Fig. 1.35

Traversals: **Preorder Traversal:** 1 2 3 4 5
Inorder Traversal: 2 4 5 3 1
Postorder Traversal: 5 4 3 2 1

(d)

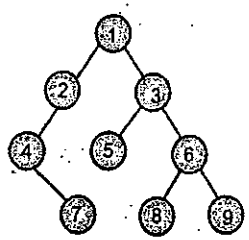


Fig. 1.36

Traversals: **Preorder Traversal:** 1 2 4 7 3 5 6 8 9
Inorder Traversal: 4 7 2 1 5 3 8 6 9
Postorder Traversal: 7 4 2 5 8 9 6 3 1

Examples:

Example 1: Perform in order, post order and pre order traversal of the binary t shown in Fig. 1.37.

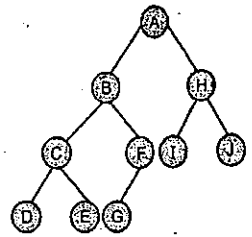


Fig. 1.37

In order traversal (left-root-right):
D C E B G F A I H J

Pre order traversal (root-left-right):
A B C D E F G H I J

Post order traversal (left-right-root):
D E C G F B I J H A

Example 2: Perform in order, post order and pre-order traversal of the binary tree shown in Fig. 1.38.

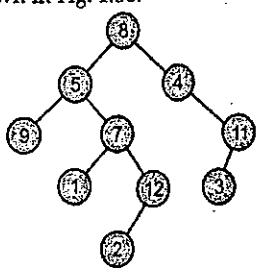


Fig. 1.38

Pre-order: 8, 5, 9, 7, 1, 12, 2, 4, 11, 3
In-order: 9, 5, 1, 7, 2, 12, 8, 4, 3, 11
Post-order: 9, 1, 2, 12, 7, 5, 3, 11, 4, 8

Program 1.4: Menu driven program for binary search tree creation and tree traversing.

```
#include<stdio.h>
#include<malloc.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node *right;
   struct node *left;
};
struct node *Create(struct node *, int);
void Inorder(struct node *);
void Preorder(struct node *);
void Postorder(struct node *);
```

```
int main()
{
   struct node *root = NULL;
   setbuf(stdout, NULL);
   int choice, item, n, i;
   printf("\n*** Binary Search Tree ***\n");
   printf("\n1. Creation of BST");
   printf("\n2. Traverse in Inorder");
   printf("\n3. Traverse in Preorder");
   printf("\n4. Traverse in Postorder");
   printf("\n5. Exit\n");
   while(1)
   {
      printf("\nEnter Your Choice :(1.Create 2.Inorder 3.Preorder
                                                  4.Postorder 5.Exit)\n");
      scanf("%d",&choice);
      switch(choice)
      {
      case 1:
         root = NULL;
         printf("Enter number of nodes:\n");
         scanf("%d",&n);
         for(i = 1; i <= n; i++)
         {
            printf("\nEnter data for node %d : ", i);
            scanf("%d",&item);
            root = Create(root,item);
         }
         break;
      case 2:
         Inorder(root);
         break;
      case 3:
         Preorder(root);
         break;
      case 4:
         Postorder(root);
         break;
```



```
      case 5:
         exit(0);
      default:
         printf("Wrong Choice !!\n");
      }
   }
   return 0;
}
struct node *Create(struct node *root, int item)
{
   if(root == NULL)
   {
      root = (struct node *)malloc(sizeof(struct node));
      root->left = root->right = NULL;
      root->data = item;
      return root;
   }
   else
   {
      if(item < root->data )
         root->left = Create(root->left,item);  //recursive function call
         else if(item > root->data )
               root->right = Create(root->right,item);
         else
               printf(" Duplicate Element is Not Allowed !!!");
      return(root);
   }
}
void Inorder(struct node *root)
{
   if( root != NULL)
   {
      Inorder(root->left); //recursive function call
      printf("%d ",root->data);
      Inorder(root->right);
   }
}
```

```
void Preorder(struct node *root)
{
    if( root != NULL)
    {
        printf("%d ",root->data);
        Preorder(root->left);//recursive function call
        Preorder(root->right);
    }
}
void Postorder(struct node *root)
{
    if( root != NULL)
    {
        Postorder(root->left);//recursive function call
        Postorder(root->right);
        printf("%d ",root->data);
    }
}
```

Output:

```
*** Binary Search Tree ***
1. Creation of BST
2. Traverse in Inorder
3. Traverse in Preorder
4. Traverse in Postorder
5. Exit
Enter Your Choice :(1.Create 2.Inorder 3.Preorder 4.Postorder 5.Exit)
1
Enter number of nodes:
5
Enter data for node 1 : 10
Enter data for node 2 : 20
Enter data for node 3 : 5
Enter data for node 4 : 15
Enter data for node 5 : 25
Enter Your Choice :(1.Create 2.Inorder 3.Preorder 4.Postorder 5.Exit)
2
5 10 15 20 25
Enter Your Choice :(1.Create 2.Inorder 3.Preorder 4.Postorder 5.Exit)
3
10 5 20 15 25
Enter Your Choice :(1.Create 2.Inorder 3.Preorder 4.Postorder 5.Exit)
4
5 15 25 20 10
Enter Your Choice :(1.Create 2.Inorder 3.Preorder 4.Postorder 5.Exit)
5
```


1.4.4 Level-order Traversal using Queue

- The level-order traversal of a binary tree traverses the nodes in a level-by-le manner from top to bottom and among the nodes of the same level they are traver from left to right. A data structure called queue is used to keep track of the eleme yet to be traverse.
- Level order traversal of a tree is breadth first traversal for the tree. Level or traversal of the tree in Fig. 1.39 is 1 2 3 4 5.

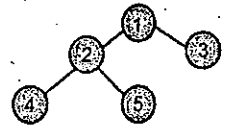


Fig. 1.39

1.4.5 Count Total Nodes of a Binary Tree

- To count total nodes of a binary tree, we traverse a tree. We can use any of the thr traversal methods.

'C' Function:

```
int CountNode(struct node *root)
{
    static int count;
    if (root== NULL)
      return 0;
  else
      count=1+CountNodes(root · left)+CountNodes(root · right);
    return count;
}
```

OR

```
int CountNode (struct node * root)
{
    if(root == NULL)
      return 0;
    else
    return (1 + CountNode(root → left) + CountNode (root → right));
}
```

1.6 Count Leaf Nodes of a Binary Tree

- The node which do not have child node is called as leaf node.

'C' Function:

```
int CountLeaf(Tree * root)
  {
     if (root==NULL)
        leafcount = 0;
     else
     if ((root · left == NULL) && (root · right == NULL))
        return(1);
        else
        return((CountLeaf (root · left) + CountLeaf (root · right));
  }
```

1.7 Count Non-Leaf Nodes of a Binary Tree

- The node which have at least one child is called as non-leaf node.

'C' Function:

```
int count_non_leaf(struct node * root)
{
if(root == NULL)
      return 0;
if(root -> left == NULL && root -> right == NULL)
      return 0;
return(1+ count_non_leaf(root -> left) + count_non_leaf(root->right));
}
```

1.8 Mirror Image of a Binary Tree

- The mirror image of a tree contains its left and right subtrees interchanged as shown in Fig. 1.40.

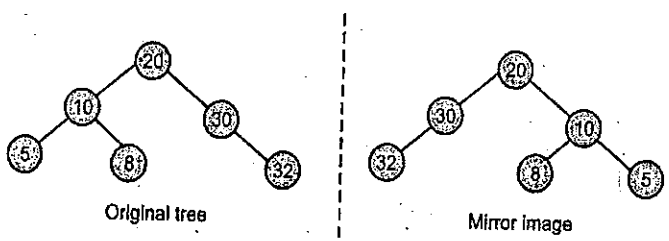


Fig. 1.40: Mirror image of a Binary Tree

- Here, we start with lower level (leaves) and move upwards till the children of the root are interchanged.

'C' Function:

```
struct node * mirror(struct node * root)
{
   struct node * temp=NULL;
   if (root != NULL)
   {
      temp = root -> left;
      root -> left = mirror(root · right);
      root -> right = mirror(temp);
   }
   return root;
}
```

1.9 APPLICATIONS [April 16]

- In this section we study various applications on tree such as heap sort, priority queue implementation (Huffman encoding).

1.9.1 Heap Sort with its Implementation [April 19]

- Heap is a special tree-based data structure. Heap sort is one of the sorting algorithms used to arrange a list of elements in order.
- The heaps are mainly used for implementing priority queue and for sorting an array using the heap sort technique.
- Heap sort algorithm uses one of the tree concepts called Heap Tree. A heap tree data structure is a complete binary tree, where the child of each node is equal to or smaller in value than the value of its parent.
- There can be two types of heap:
  1. **Max Heap:** In this type of heap, the value of parent node will always be greater than or equal to the value of child node across the tree and the node with highest value will be the root node of the tree.
  2. **Min Heap:** In this type of heap, the value of parent node will always be less than or equal to the value of child node across the tree and the node with lowest value will be the root node of tree.
- **Example:** Given the following numeric data,

  2 7 15 25 40 55 75

  Max heap and min heap trees are shown in Fig. 1.41.

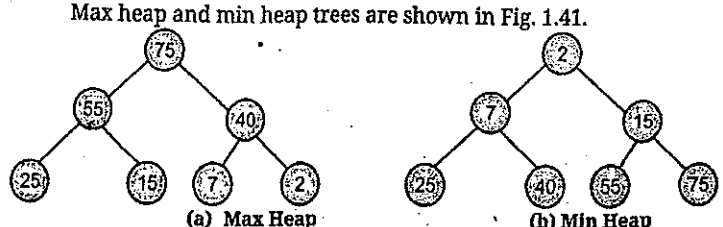


Fig. 1.41: Heap Trees

- Heap sort It is one of the important application of heap tree.
- Heap sort is based on heap tree and it is an efficient sorting method. We can sort either in ascending or descending using heap sort.

- Sorting includes three steps:
  1. Built a heap tree with a given data.
  2. Delete the root node from the heap.
     - Rebuilt the heap after deletion and
     - Place the deleted node is the output.
  3. Continue step 2 until heap tree is empty.

Example: Consider the following set of data to be sort in ascending order.

32 15 64 2 75 67 57 80

We first create binary tree and then convert into heap tree.

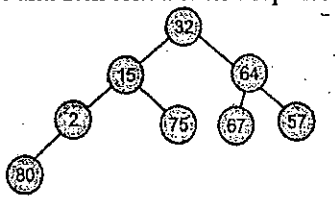


Fig. 1.42

The tree of Fig. 1.42 is not a heap (max) tree, we convert it into heap tree as follows:

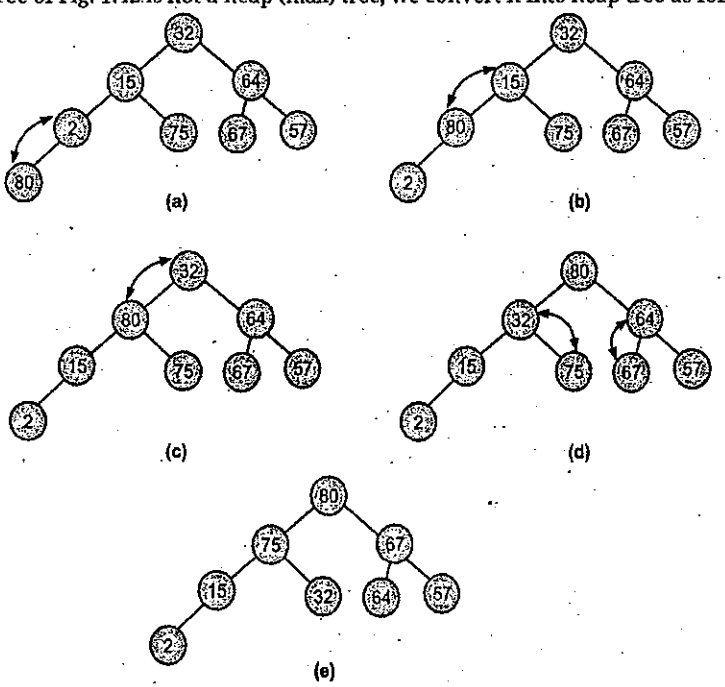


Fig. 1.43: Heap Tree (Max)

Here, if the keys in the children are greater than the parent node, the key in parent and in child are interchanged.

Heap Sort Method/Implementation:

Algorithm Heap_Sort:

Step 1 : Accept n elements in the array A.

Step 2 : Convert data into heap (A).

Step 3 : Initialize i = n

Step 4 : while (i > 1) do

```
{swap (A[1], A[i])        /*swapping first (top) and last element*/
i = i – 1                 /*pointer is shifted to left*/
j = 1                     /*rebuilt the heap*/
```

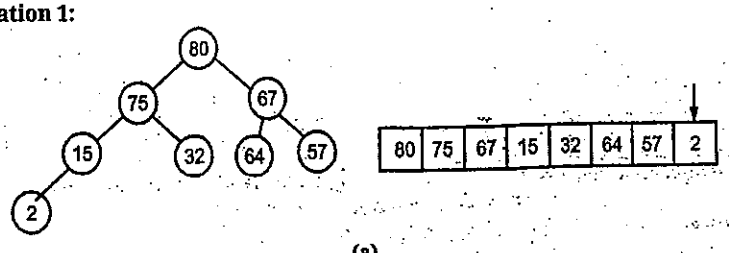
Step 5 : while (j < i) do

```
{
lchild = 2 * j            /*left child
rchild = 2 * j + 1        /*right child
if (A[j] < A[lchild]) and (A[lchild] > A[rchild]) then
{ swap(A[j]; A[lchild])
  j = lchild
else
if (A[j] < A[rchild]) & (A[rchild] > A[lchild]) then
{
  swap (A[j], A[rchild])
        j = rchild
else
  break
}
} /*endif*/
} /*endwhile*/
}/*endwhile of step 4*/
```

Step 6: Stop

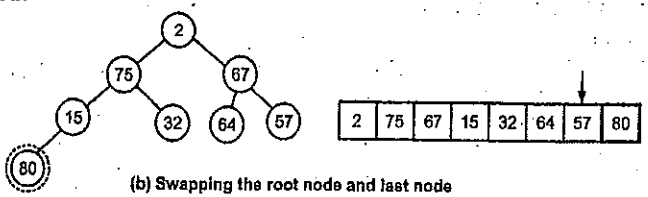
Consider the heap tree of Fig. 1.44.

Iteration 1:



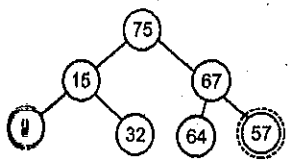
(a)

Iteration 2:

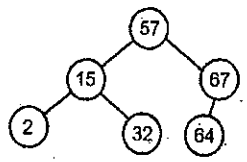


(b) Swapping the root node and last node

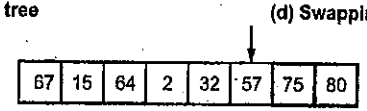
Iteration 3:

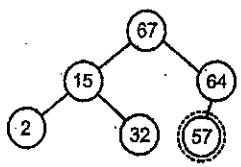


(c) Rebuilt the heap tree

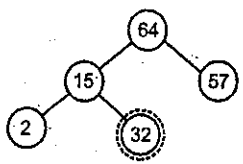


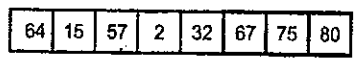
(d) Swapping 57 and root node



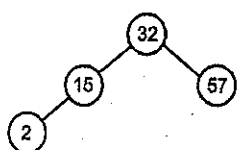


(e) Rebuilt

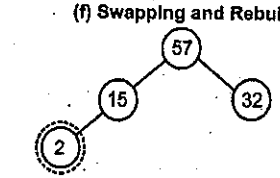


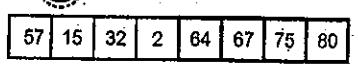


(f) Swapping and Rebuilt

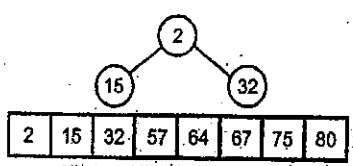


(g) After swapping 32 and 64

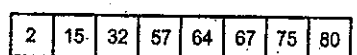




(h) Rebuilt



(i) After swapping 2 and 57



Sorted list when heap is empty

(j) Continue and we get emptyheap

Fig. 1.44: Tracing of Heap Sort

Hence we get the array sorted in ascending order.

Program 1.5: Program for heap sort.

```
#include <stdio.h>
void main()
{
   int heap[10], no, i, j, c, root, temp;
   printf("\n Enter no of elements :");
   scanf("%d", &no);
   printf("\n Enter the nos : ");
   for (i = 0; i < no; i++)
   scanf("%d", &heap[i]);
   for (i = 1; i < no; i++)
   {
      c = i;
      do
      {
         root = (c - 1) / 2;
         if (heap[root] < heap[c]).  /* to create MAX heap array */
         {
            temp = heap[root];
            heap[root] = heap[c];
            heap[c] = temp;
         }
         c = root;
      } while (c != 0);
   }
   printf("Heap array : ");
   for (i = 0; i < no; i++)
   printf("%d\t ", heap[i]);
   for (j = no - 1; j >= 0; j--)
   {
      temp = heap[0];
      heap[0] = heap[j];   /* swap max element with rightmost leaf element */
      heap[j] = temp;
      root = 0;
      do
      {
         c = 2 * root + 1;    /* left node of root element */
         if((heap[c] < heap[c + 1]) && c < j-1) c++;
```

```
            if (heap[root]<heap[c] && c<j)/* again rearrange to max heap array */
            {
               temp = heap[root];
               heap[root] = heap[c];
               heap[c] = temp;
            }
            root = c;
         } while (c < j);
      }
      printf("\n The sorted array is : ");
      for (i = 0; i < no; i++)
      printf("\t %d", heap[i]);
      printf("\n Complexity : \n Best case = Avg case = Worst case = O(n logn) \n");
   }
```

Output:

```
Enter no of elements :5
Enter the nos :
10
6
30
9
40
Heap array : 40   30   10   6   9
The sorted array is : 6   9   10   30   40
Complexity :
Best case = Avg case = Worst case = O(n logn)
```

1.2 Introduction to Greedy Strategy

- Generally, optimization problem or the problem where we have to find maximum or minimum of something or we have to find some optimal solution, greedy technique/strategy is used.
- Greedy algorithm is an algorithm is designed to achieve optimum solution for a given problem.
- In greedy algorithm approach, decisions are made from the given solution domain. As being greedy, the closest solution that seems to provide an optimum solution is chosen.
- Greedy algorithms try to find a localized optimum solution, which may eventually lead to globally optimized solutions. However, generally greedy algorithms do not provide globally optimized solutions.

- An optimization problem has two types of solutions:
  1. **Feasible Solution:** This can be referred as approximate solution (subset of solution) satisfying the objective function and it may or may not build up to the optimal solution.
  2. **Optimal Solution:** This can be defined as a feasible solution that either maximi or minimizes the objective function.
- A greedy algorithm proceeds step-by-step, by considering one input at a time. At e stage, the decision is made regarding whether a particular input (say x) chosen g an optimal solution or not.
  - Our choice of selecting input x is being guided by the selection function (select).
  - If the inclusion of x gives an optimal solution, then this input x is added into partial solution set.
  - On the other hand, if the inclusion of that input x results in an infeasible solut then this input x is not added to the partial solution.
  - The input we tried and rejected is never considered again.
  - When a greedy algorithm works correctly, the first solution found in this wa always optimal.
- In brief, at each stage, the following activities are performed in greedy method:
  1. First we select an element, say x, from input domain C.
  2. Then we check whether the solution set S is feasible or not. That is, we ch whether x can be included into the solution set S or not. If yes, then solution se no, then this input x is discarded and not added to the partial solution se Initially S is set to empty.
  3. Continue until S is filled up (i.e. optimal solution found) or C is exhaus whichever is earlier.

     (Note: From the set of feasible solutions, particular solution that satisfies or nea satisfies the objective of the function (either maximize or minimize, as the c may be), is called optimal solution.

1.3 Huffman Encoding (Implementation using Priority Queue)

- Huffman encoding developed by David Huffman. Data can be encoded efficien using Huffman codes.
- Huffman code is a data compression algorithm which uses the greedy technique for implementation. The algorithm is based on the frequency of the characters appear in a file.
- Huffman code is a widely used and beneficial technique for compressing da Huffman's greedy algorithm uses a table of the frequencies of occurrences of e character to build up an optimal way of representing each character as a bin string.
- Huffman encoding is used to compress a file that can reduce the memory storage.

- How can we represent the data in a compact way?
  1. **Fixed Length Code:** Each letter represented by an equal number of bits. With a fixed length code, at least three (3) bits per character.
  2. **Variable Length Code:** It can do considerably better than a fixed-length code, by giving many characters' short code words and infrequent character long code words.

Greedy Algorithm for Constructing a Huffman Code:

- Huffman invented a greedy algorithm that creates an optimal prefix code called a Huffman Code.
- There are following mainly two major parts in Huffman Coding:
  1. Build a Huffman Tree from input characters.
  2. Traverse the Huffman Tree and assign codes to characters.

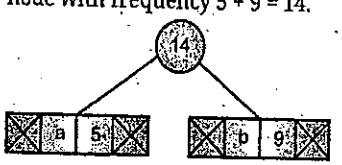
Steps to build Huffman Tree:

- Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.
  1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root).
  2. Extract two nodes with the minimum frequency from the min heap.
  3. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
  4. Repeat Steps 2 and 3 until the heap contains only one node. The remaining node is the root node and the tree is complete.
- Let us understand the algorithm with an example:

Character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

Step 1 : Build a min heap that contains 6 nodes where each node represents root of a tree with single node.

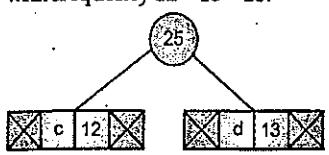
Step 2 : Extract two minimum frequency nodes from min heap. Add a new internal node with frequency 5 + 9 = 14.



Now min heap contains 5 nodes where 4 nodes are roots of trees with single element each, and one heap node is root of tree with 3 elements

Character	Frequency
c	12
d	13
Internal Node	14
e	16
f	45

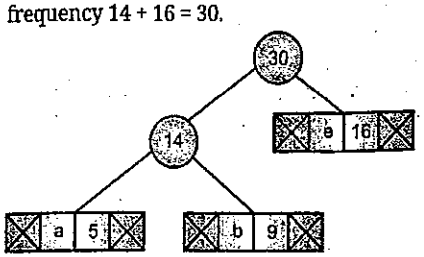
Step 3 : Extract two minimum frequency nodes from heap. Add a new internal node with frequency 12 + 13 = 25.



Now min heap contains 4 nodes where 2 nodes are roots of trees with single element each, and two heap nodes are root of tree with more than one nodes.

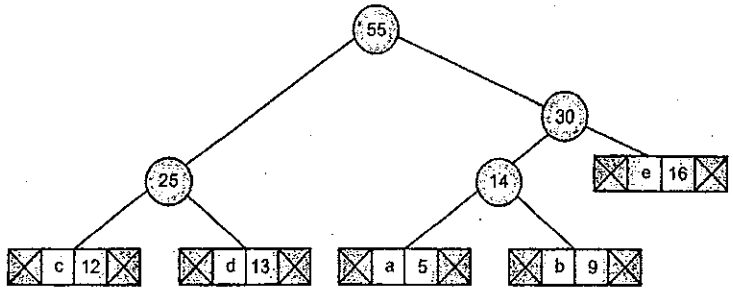
Character	Frequency
Internal Node	14
e	16
Internal Node	25
f	45

Step 4 : Extract two minimum frequency nodes. Add a new internal node with frequency 14 + 16 = 30.



Now min heap contains 3 nodes.

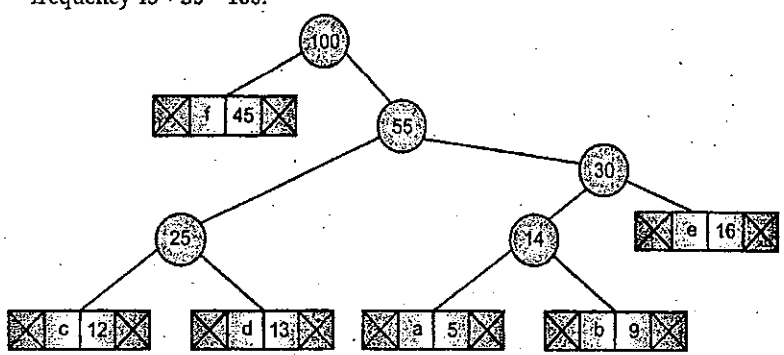
Character	Frequency
Internal Node	25
Internal Node	30
f	45



Now min heap contains 2 nodes.

Character	Frequency
f	45
Internal Node	55

Step 6 : Extract two minimum frequency nodes. Add a new internal node with frequency 45 + 55 = 100.



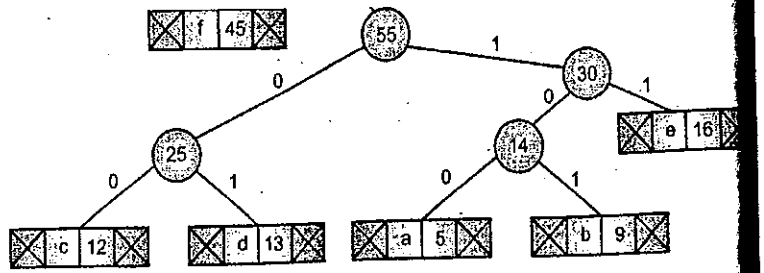
Now min heap contains only one node.

Character	Frequency
Internal Node	100

Since, the heap contains only one node, the algorithm stops here.

Steps to print codes from Huffman Tree:

Traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to the array. Print the array when a leaf node is encountered.



The codes are as follows:

Character	Code word
f	0
c	100
d	101
a	1100
b	1101
e	111

Program 1.6: Program for Huffman Coding.

```
#include <stdio.h>
#include <stdlib.h>
/*This constant can be avoided by explicitly*/
/*calculating height of Huffman Tree*/
#define MAX_TREE_HT 100
/*A Huffman tree node*/
struct MinHeapNode {
    /* One of the input characters */
    char data;
    /* Frequency of the character */
    unsigned freq;
    /* Left and right child of this node */
    struct MinHeapNode *left, *right;
};
/* A Min Heap:  Collection of */
/* min-heap (or Huffman tree) nodes */
struct MinHeap {
    /* Current size of min heap */
    unsigned size;
```

```
    /* capacity of min heap */
    unsigned capacity;
    /* Array of minheap node pointers */
    struct MinHeapNode** array;
};
/* A utility function allocate a new */
/* min heap node with given character */
/* and frequency of the character */
struct MinHeapNode* newNode(char data, unsigned freq)
{
    struct MinHeapNode* temp = (struct MinHeapNode*)malloc
                                        (sizeof(struct MinHeapNode));
    temp->left = temp->right = NULL;
    temp->data = data;
    temp->freq = freq;
    return temp;
}
/* A utility function to create */
/* a min heap of given capacity */
struct MinHeap* createMinHeap(unsigned capacity)
{
    struct MinHeap* minHeap = (struct MinHeap*)malloc(sizeof(struct
                                                        MinHeap));
    /* current size is 0 */
    minHeap->size = 0;
    minHeap->capacity = capacity;
    minHeap->array = (struct MinHeapNode**)malloc(minHeap->
                        capacity * sizeof(struct MinHeapNode*));
    return minHeap;
}
/* A utility function to */
/* swap two min heap nodes */
void swapMinHeapNode(struct MinHeapNode** a, struct MinHeapNode** b)
{
   struct MinHeapNode* t = *a;
    *a = *b;
    *b = t;
}
/* The standard minHeapify function. */
```



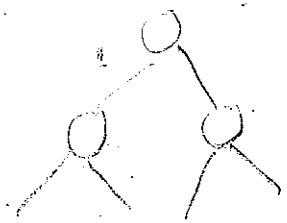
```
void minHeapify(struct MinHeap* minHeap, int idx)
{
    int smallest = idx;
    int left = 2 * idx + 1;
    int right = 2 * idx + 2;
    if (left < minHeap->size && minHeap->array[left]->
                freq < minHeap->array[smallest]->freq) smallest = left;
    if (right < minHeap->size && minHeap->array[right]->
                freq < minHeap->array[smallest]->freq) smallest = right;
    if (smallest != idx) {
        swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);
        minHeapify(minHeap, smallest);
    }
}
/* A utility function to check */
/* if size of heap is 1 or not */
int isSizeOne(struct MinHeap* minHeap)
{
    return (minHeap->size == 1);
}
/* A standard function to extract */
/* minimum value node from heap */
struct MinHeapNode* extractMin(struct MinHeap* minHeap)
{
    struct MinHeapNode* temp = minHeap->array[0];
    minHeap->array[0] = minHeap->array[minHeap->size - 1];
    --minHeap->size;
    minHeapify(minHeap, 0);
    return temp;
}
/* A utility function to insert */
/* a new node to Min Heap */
void insertMinHeap(struct MinHeap* minHeap, struct MinHeapNode*
                                                    minHeapNode)
{
    ++minHeap->size;
    int i = minHeap->size - 1;
    while (i && minHeapNode->freq < minHeap->array[(i - 1) / 2]->freq) {
```

```
            minHeap->array[i] = minHeap->array[(i - 1) / 2];
            i = (i - 1) / 2;
        }
        minHeap->array[i] = minHeapNode;
    }
    /* A standard function to build min heap */
    void buildMinHeap(struct MinHeap* minHeap)
    {
        int n = minHeap->size - 1;
        int i;
        for (i = (n - 1) / 2; i >= 0; --i)
            minHeapify(minHeap, i);
    }
    /* A utility function to print an array of size n */
    void printArr(int arr[], int n)
    {
        int i;
        for (i = 0; i < n; ++i)
            printf("%d", arr[i]);
        printf("\n");
    }
    /* Utility function to check if this node is leaf */
    int isLeaf(struct MinHeapNode* root)
    {
        return !(root->left) && !(root->right);
    }
    /* Creates a min heap of capacity */
    /* equal to size and inserts all character of */
    /* data[] in min heap. Initially size of */
    /* min heap is equal to capacity */
    struct MinHeap* createAndBuildMinHeap(char data[], int freq[], int size)
    {
        struct MinHeap* minHeap = createMinHeap(size);
        for (int i = 0; i < size; ++i)
            minHeap->array[i] = newNode(data[i], freq[i]);
        minHeap->size = size;
        buildMinHeap(minHeap);
        return minHeap;
    }
```



```
    /* The main function that builds Huffman tree */
    struct MinHeapNode* buildHuffmanTree(char data[], int freq[], int size)
    {
        struct MinHeapNode *left, *right, *top;
        /* Step 1: Create a min heap of capacity */
        /* equal to size.  Initially, there are */
        /* modes equal to size. */
        struct MinHeap* minHeap = createAndBuildMinHeap(data, freq, size);
        /* Iterate while size of heap doesn't become 1
        while (!isSizeOne(minHeap)) {
            /* Step 2: Extract the two minimum */
            /* freq items from min heap */
            left = extractMin(minHeap);
            right = extractMin(minHeap);
            /* Step 3:  Create a new internal */
            /* node with frequency equal to the */
            /* sum of the two nodes frequencies. */
            /* Make the two extracted node as */
            /* left and right children of this new node. */
            /* Add this node to the min heap */
            /* '$' is a special value for internal nodes, not used */
            top = newNode('$', left->freq + right->freq);
            top->left = left;
            top->right = right;
            insertMinHeap(minHeap, top);
        }
        /* Step 4: The remaining node is the */
        /* root node and the tree is complete. */
        return extractMin(minHeap);
    }
    /* Prints huffman codes from the root of Huffman Tree. */
    /* It uses arr[] to store codes */
    void printCodes(struct MinHeapNode* root, int arr[], int top)
    {
        /* Assign 0 to left edge and recur */
        if (root->left) {
            arr[top] = 0;
            printCodes(root->left, arr, top + 1);
        }
        /* Assign 1 to right edge and recur */
```

```
        if (root->right) {
            arr[top] = 1;
            printCodes(root->right, arr, top + 1);
        }
        /* If this is a leaf node, then */
        /* it contains one of the input */
        /* characters, print the character */
        /* and its code from arr[] */
        if (isLeaf(root)) {
            printf("%c: ", root->data);
            printArr(arr, top);
        }
    }
    /* The main function that builds a */
    /* Huffman Tree and print codes by traversing */
    /* the built Huffman Tree */
    void HuffmanCodes(char data[], int freq[], int size)
    {
        /* Construct Huffman Tree */
        struct MinHeapNode* root = buildHuffmanTree(data, freq, size);
        /* Print Huffman codes using */
        /* the Huffman tree built above */
        int arr[MAX_TREE_HT], top = 0;
        printCodes(root, arr, top);
    }
    /* Driver program to test above functions */
    int main()
    {
        char arr[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
        int freq[] = { 5, 9, 12, 13, 16, 45 };
        int size = sizeof(arr) / sizeof(arr[0]);
        HuffmanCodes(arr, freq, size);
        return 0;
    }
```



Output:

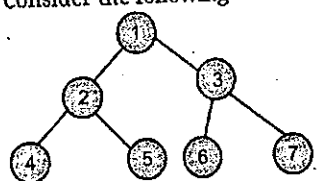
```
f: 0
c: 100
d: 101
a: 1100
b: 1101
e: 111
```


PRACTICE QUESTIONS

Q. I Multiple Choice Questions:

1. Which is widely used non-linear data structure?
   (a) Tree (b) Array
   (c) Queue (d) Stack
2. Which in a tree data structure stores the actual data of that particular element and link to next element in hierarchical structure?
   (a) Root (b) Node
   (c) Child (d) Leaf
3. Which is the operation in tree will remove a node and all of its descendants from the tree?
   (a) Prune (b) Graft
   (c) Insert (d) Delete
4. The depth of the root node = .
   (a) 1 (b) 3
   (c) 0 (d) 4
5. Which is a set of several trees that are not linked to each other.
   (a) Node (b) Forest
   (c) Leaf (d) Root
6. In which tree, every node can have a maximum of two children, which are known as left child and right child.
   (a) Binary (b) Binary search
   (c) Strictly (d) Extended
7. Which is data structure like a tree-based data structure that satisfies a property called heap property?
   (a) Tree (b) Graph
   (c) Heap (d) Stack
8. How many roots contains a tree?
   (a) 1 (b) 3
   (c) 0 (d) 4
9. The total number of edges from root node to a particular node is called as,
   (a) Height (b) Path
   (c) Depth (d) Degree
10. In which binary tree every node has either two or zero number of children?
    (a) Binary (b) Binary search
    (c) Strictly (d) Extended
11. Which tree operation will return a list or some other collection containing every descendant of a particular node, including the root node itself?
    (a) Prune (b) Graft
    (c) Insert (d) Enumerate

12. The ways to represent binary trees are,
    (a) Array (b) Linked list
    (c) Both (a) and (b) (d) None of these
13. Which coding is a technique of compressing data to reduce its size without losing any of the details?
    (a) Huffman (b) Greedy
    (c) Both (a) and (b) (d) None of these
14. Which strategy provides optimal solution to the problem?
    (a) Huffman (b) Greedy
    (c) Both (a) and (b) (d) None of these
15. Consider the following tree:



If the postorder traversal gives (ab–cd*+) then the label of the nodes 1, 2, 3, 4, 5, 6 will be,
    (a) +, –, *, a, b, c, d (b) a, –, b, +, c, *, d
    (c) a, b, c, d, –, *, + (d) –, a, b, +, *, c, d
16. Which of the following statement about binary tree is correct?
    (a) Every binary tree is either complete or full
    (b) Every complete binary tree is also a full binary tree
    (c) Every full binary tree is also a complete binary tree
    (d) A binary tree cannot be both complete and full
17. Which type of traversal of binary search tree outputs the value in sorted order?
    (a) Preorder (b) Inorder
    (c) Postorder (d) None of these

Answers

1. (a)	2. (b)	3. (a)	4. (c)	5. (b)	6. (a)	7. (c)
8. (a)	9. (c)	10.(c)	11. (d)	12. (c)	13. (a)	14. (b)
15. (a)	16. (c)	17. (b)				

Q. II Fill in the Blanks:

1. A tree is a non-linear _____ data structure.
2. There is only _____ root per tree and one path from the root node to any node.
3. In tree data structure, every individual element is called as _____.
4. Nodes which belong to _____ parent are called as siblings.
5. The _____ operation will remove a specified node from the tree.
6. Height of all leaf nodes is _____.
7. A _____ tree is simply a tree with zero nodes.

8. In a binary tree, every node can have a maximum of _____ children.
9. _____External node is also a node with no child.
10. In a _____ binary tree, every internal node has exactly two children and all le nodes are at same level.
11. In array representation of binary tree, we use a _____ dimensional array. represent a binary tree.
12. Binary tree representing an arithmetic expression is called _____ tree.
13. In a binary search tree, the value of all the nodes in the left sub-tree is _____ tha the value of the root.
14. In _____ heap all parent node's values are greater than or equal to children node values, root node value is the largest.

Answers

1. hierarchical	2. one	3. Node	4. same	5. delete	6. 0
7. null	8. sequence	9. two	10. complete	11. one	12. expression
13. less	14. max				

Q. III State True or False:

1. Tree is a linear data structure which organizes data in hierarchical structure.
2. The total number of children of a node is called as height of that Node.
3. A binary tree in which every internal node has exactly two children and all lea nodes are at same level is called Complete Binary Tree.
4. In any tree, there must be only one root node.
5. A tree is hierarchical collection of nodes.
6. The root node is the origin of tree data structure.
7. the leaf nodes are also called as External Nodes.
8. Removing a whole selection of a tree called grafting.
9. Total number of edges that lies on the longest path from any leaf node to particular node is called as height of that node.
10. A binary tree is a tree data structure in which each parent node can have at mos two children.
11. Adding a whole section to a tree called pruning.
12. In min heap all parent node's values are less than or equal to children node's values, root node value is the smallest.
13. A heap is a complete binary tree.
14. An algebraic expression can be represented in the form of binary tree which is known as expression tree.

Answers

1. (F)	2. (F)	3. (T)	4. (T)	5. (T)	6. (T)	7. (T)
8. (F)	9. (T)	10. (T)	11. (F)	12. (T)	13. (T)	14. (T)

Q. IV Answer the following Questions:

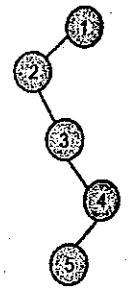
(A) Short Answer Questions:

1. What is tree?
2. List operations on tree.
3. Define the term binary tree.
4. What are the types of binary trees?
5. Define heap.
6. What is binary search tree?
7. List tree traversals.
8. Define expression tree.
9. What is heap sort?
10. What are the applications of trees?
11. List representations on trees.
12. Define node of tree.
13. What is path of tree.
14. Define skewed tree.

(B) Long Answer Questions:

1. Define tree. Describe array and linked representation of binary tree.
2. Explain various types of tree with diagram.
3. Define:
   (i) Height of tree
   (ii) Level of tree
   (iii) Complete binary tree
   (iv) Expression tree
   (v) Binary search tree.
4. Describe full binary tree with example.
5. With the help of example describe binary tree.
6. Write a program to construct binary search tree of given numbers (data).
7. Root of a binary tree is an ancestor of every node, comment.
8. Write a function to count the number of leaf nodes of a given tree.
9. Write a function for postorder and preorder traversal of binary tree.
10. Define binary tree and its types.
11. Write a 'C' program to create a tree and count total number of nodes in a tree.
12. Write a recursive function in C that creates a mirror image of a binary tree.
13. Construct Binary search tree for the following data and give inorder, preorder and postorder tree traversal.
    20, 30, 10, 5, 16, 21, 29, 45, 0, 15, 6.
14. Write a 'C' function to print minimum and maximum element from a given binary search tree.
15. Explain sequential representation of binary tree.
16. Define the following terms:
    (i) Complete binary tree.
    (ii) Strictly binary tree.
17. Write an algorithm to count leaf nodes in a tree.

18. What are different tree traversal methods? Explain with example.
19. What is binary search tree? How to implement it? Explain with example.
20. Traverse following trees in:
    (i) Inorder (ii) Preorder (iii) Postorder



UNIVERSITY QUESTIONS AND ANSWERS

April 2016

1. List any two applications of tree data structure. [1 M]

Ans. Refer to Section 1.5.

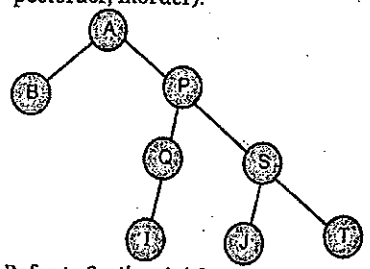
2. Write a recursive 'C' function to insert an element in a binary search tree. [5 M]

Ans. Refer to Section 1.4.1.3.

3. Write the steps for creating a binary search tree for the following data:
   13, 4, 25, 3, 21, 20, 7 [3 M]

Ans. Refer to Section 1.4.1.1.

4. Traverse the following binary tree using three traversals techniques (preorder, postorder, inorder). [3 M]



Ans. Refer to Section 1.4.3.

April 2017

1. Write a 'C' function to compare two BST. [5 M]

Ans. Refer to Section 1.4.1.2.

2. Define skewed binary tree. [1 M]

Ans. Refer to Section 1.2.2.

October 2017

1. What is siblings? [1 M]

Ans. Refer to Section 1.1.3, Point (9).

2. Write a 'C' function to insert an element in a binary search tree. [5 M]

Ans. Refer to Section 1.4.1.3.

3. Show steps in creating a binary search tree for the data: [3 M]
40, 70, 60, 50, 65, 20, 25

Ans. Refer to Section 1.4.1.1.

April 2018

1. Define degree of the tree. [1 M]

Ans. Refer to Section 1.1.3, Point (12).

2. Write a recursive 'C' function to insert an element in a binary search tree. [5 M]

Ans. Refer to Section 1.4.1.3.

3. Write the steps for creating a binary search tree for the following data: [4 M]
15, 11, 13, 8, 9, 18, 16

Ans. Refer to Section 1.4.1.1.

October 2018

1. What is complete binary tree? [1 M]

Ans. Refer to Section 1.2.5.

2. Write a recursive 'C' function to insert an element in a binary search tree. [5 M]

Ans. Refer to Section 1.4.1.3.

April 2019

1. Define the term right skewed binary tree. [1 M]

Ans. Refer to Section 1.2.2.

2. The indegree of the root node of a tree is always zero. Justify (T/F). [1 M]

Ans. Refer to Section 1.1.3, Point (16).

3. Write a Recursive 'C' function to count total nodes in a BST. [1 M]

Ans. Refer to Section 1.4.5.

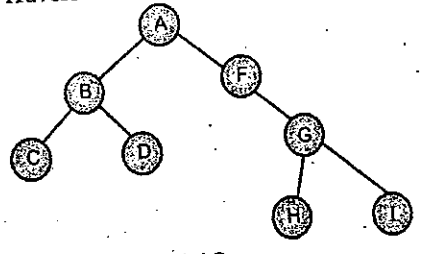
4. Write the steps for creating a BST for the following data: [4 M]
22, 13, 4, 6, 25, 23, 20, 18, 7, 27.

Ans. Refer to Section 1.4.1.1.

5. Define the term heap tree. [1 M]

Ans. Refer to Section 1.5.1.

6. Traverse the following binary tree using preorder, postorder, inorder. [3 M]



Ans. Refer to Section 1.4.3.

CHAPTER 2

Efficient Search Tre

Objectives ...

- To study AVL Trees with its Operations
- To learn Red Black Trees
- To understand B and B+ Tree with its Operations

2.0 INTRODUCTION

- The efficiency of many operations on trees is related to the height of the tree example searching, inserting, and deleting.
- The efficiency of various operations on a Binary Search Tree (BST) decreases increase in the differences between the heights of right sub tree and left sub tree o root node.
- Hence, the differences between the heights of left sub tree and right sub tree shoul kept to the minimum.
- Various operations performed on binary tree can lead to an unbalanced tree, in w either the height of left sub tree is much more than the height of right sub tree or versa.
- Such type of tree must be balanced using some techniques to achieve better efficie level.
- The need to have balanced tree led to the emergence of another type of binary sea tree known as height-balanced tree (also known as AVL tree) named after t inventor G. M. Adelson-Velsky and E. M. Landis.
- The AVL tree is a special kind of binary search tree, which satisfies the following conditions:
  1. The heights of the left and right sub trees of a node differ by one.
  2. The left and right sub trees of a node (if exist) are also AVL trees.

2.1 TERMINOLOGY

- In 1962, Adelson-Velsky and Landis introduced a binary tree structure that is balan with respect to the height of subtree. That is why it is called a height balanced tree.
- The basic objective of the height balanced tree is to perform the searching, insert and deletion operations efficiently.
- A balanced binary tree is a binary tree in which the heights of two sub trees (left right) of every node never differ by more than 1.

- A binary search tree is said to height balanced binary tree if all its nodes have a balance factor of 1, 0 or – 1 i.e.,

  $|h_L - h_R| \leq 1$

  Where, h_L and h_R are heights of left and right subtrees respectively.

1. Balance Factor: [Oct. 17, April 19]

- The term Balancing Factor (BF) is used to determine whether the given binary search tree is balanced or not.
- The BF of a code is calculated as the height of its left sub tree minus the height of right sub tree i.e.,

  Balance factor = $h_L - h_R$
- The balance factor of a node is a binary tree can have value +1, –1 or 0 depending on whether the height of its left sub tree is greater than or equal to the height of its right subtree.
- The balance factor of each node is indicated in the Fig. 2.1.
  - If balance factor of any node is +1, it means that the left sub-tree is one level higher than the right sub-tree.
  - If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.
  - If balance factor of any node is –1, it means that the left sub-tree is one level lower than the right sub-tree.

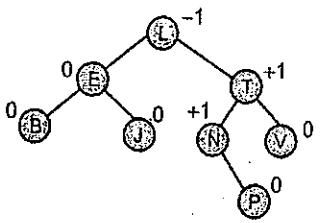


Fig. 2.1: Balance Factor in Binary Tree

2. AVL Tree:

- AVL tree is a binary search tree in which the difference of heights of left and right subtrees of any node is less than or equal to one.
- The technique of balancing the height of binary trees was developed by in the year 1962 by G. M. Adelson-Velsky and E. M. Landis in the year 1962 and hence given the short form as AVL tree Balanced Binary Tree.
- AVL trees are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference (balance factor) is not more than 1.
- Fig. 2.2 shows a binary search tree and every node is satisfying balance factor condition. So this tree is said to be an AVL tree.

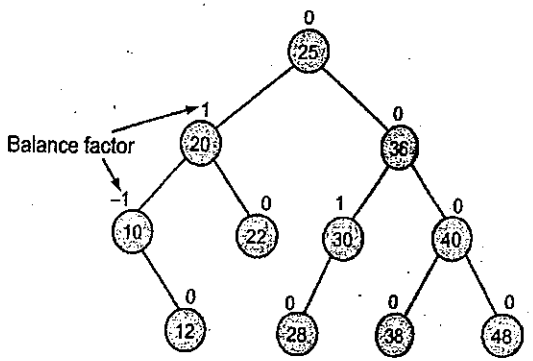


Fig. 2.2

3. Red Black Tree:

- A red black tree is a variant of Binary Search Tree (BST) in which an additional attribute, 'color' is used for balancing the tree. The value of this attribute can be either red or black.
- The red black trees are self-balancing binary search tree. In this type of tree, the leaf nodes are the NULL/NIL child nodes storing no data.
- In addition to the conditions satisfied by binary search tree, the following conditions/rules must also be satisfied for being a red black tree:

  (i) Each and every node is either red or black.

  (ii) The roof node and leaf nodes are always black-in color.

  (iii) If any node is red, then both its child nodes are black.

  (iv) Each and every path from a given node to the leaf node contains same number of black nodes. The number of black nodes on such a path is known as black-height of a node.
- Fig. 2.3 shows a red back tree.

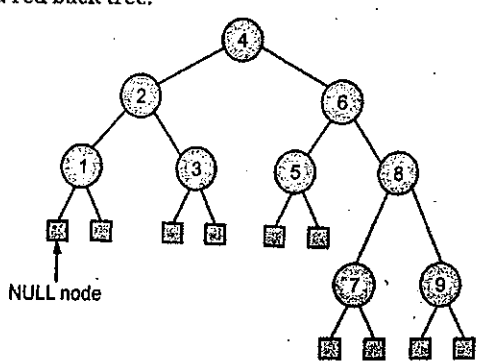


Fig. 2.3

- The AVL trees are more balanced compared to red black trees, but they may cause more rotations during insertion and deletion.
- So if the application involves many frequent insertions and deletions, then Red Black trees should be preferred.
- And if the insertions and deletions are less frequent and search is a more frequent operation, then AVL tree should be preferred over red black tree.

4. Splay Tree:

- Splay tree was invented by D. D. Sleator and R. E. Tarjan in 1985. According them the tree is called splay tree (splay means to spread wide apart).
- Splay tree is another variant of a Binary Search Tree (BST). In a splay tree, recently accessed element is placed at the root of the tree.
- Splay Tree is a self-adjusted/self-balancing Binary Search Tree in which every operation on element rearranges the tree so that the element is placed at the root position of the tree.
- All normal operations on a binary search tree are combined with one basic operation, called splaying.
- Splaying the tree for a certain element rearranges the tree so that the element is placed at the root of the tree.
- Fig. 2.4 shows an example of splay tree.

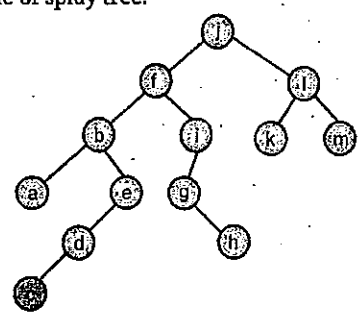


Fig. 2.4

Rotations in Splay Tree:

- In zig rotation, every node moves one position to the right from its current position.
- In zag rotation, every node moves one position to the left from its current position.
- In zig-zig rotation, every node moves two positions to the right from its current position.
- In zag-zag rotation, every node moves two positions to the left from its current position.
- In zig-zag rotation, every node moves one position to the right followed by one position to the left from its current position.
- In zag-zig rotation, every node moves one position to the left followed by one position to the right from its current position.

5. Lexical Search Tree (Trie):

- Instead of searching a tree using the entire value of a key, we can consider the k be a sequence of characters, such as word or non-numeric identifier.
- When placed in a tree, each node has a place for each of the possible values th characters in the lexical tree can assume.
- For example, if a key can contain the complete alphabet, each node has 26 entrie for each of the letters of the alphabet and known as a lexical 26-ary tree.
- Each and every entry in the lexical search tree contains a pointer to the nest lev addition, each node of 26-ary tree contains 26 pointers, the first representin letter A, the second the letter B, and so forth until the last pointer which represent letter Z.
- Because each letter in the first level must point to a complete set of values, the se level contains 26*26 entries, one node of 26 entries for each of the 26 letters in the level.
- At the third level 26*26*26 entries and finally we store the actual key at the leaf.
- If a key has three letters, these are at least three levels in the tree. If a key ha letters, these are ten levels in the tree. Because of a lexical tree can contain r different keys, the largest word determines the height of the tree.
- Fig. 2.5 shows a lexical tree.

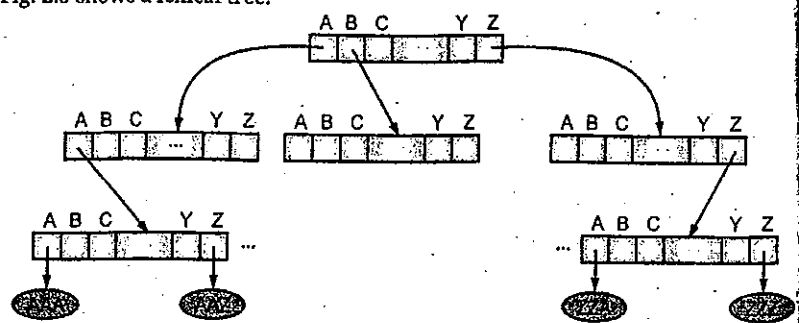


Fig. 2.5: Lexical Tree Structure

Trie:

- A trie is a lexical m-ary tree in which the pointers pointing to non-existing charac are replaced by null pointers.
- All the search trees are used to store the collection of numerical values but they not suitable for storing the collection of words or strings.
- Trie is a data structure which is used to store the collection of strings and m searching of a pattern in words more easy.
- The term trie came from the word retrieval. Trie is an efficient information sto and retrieval data structure.

data structure makes retrieval of a string from the collection of strings more ...

... is also called as Prefix Tree and sometimes Digital Tree. Trie is a tree like data ... used to store collection of strings.

... shows an example of trie. Consider a list of strings Cat, Bat, Ball, Rat, Cap, Be.

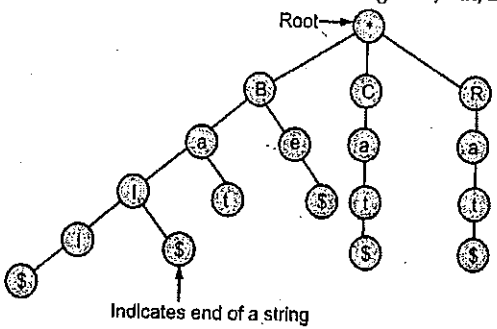


Fig. 2.6

AVL TREE

- In this section we will study concept of AVL tree and its rotations.

Concept

- AVL, named after inventors Adelson-Velsky and Landis, is a binary tree that self-balances by keeping a check on the balance factor of every node.
- The balance factor of a node is the difference in the heights of the left and right subtrees. The balance factor of every node in the AVL tree should be either +1, 0 or -1.
- AVL trees are special kind of binary search trees. In AVL trees, difference of heights of left subtree and right subtree of any node is less than or equal to one.
- AVL trees are also called as self-balancing binary search trees. The node structure of AVL tree are given below:

```
struct AVLNode
{
   int data;
   struct AVLNode *left, *right;
   int balfactor;
   };
```

- AVL tree can be defined as, let T be a non-empty binary tree with T_L and T_R as its left and right subtrees. The tree is height balanced if:
  - T_L and T_R are height balanced.
  - $h_L - h_R \leq 1$, where $h_L - h_R$ are the heights of T_L and T_R

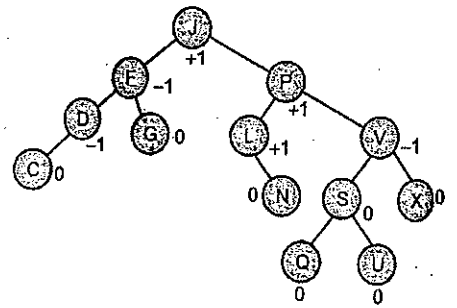


Fig. 2.7: AVL Tree with Balance Factors

- Balance factor of a node is the difference between the heights of the left and right subtrees of that node. Consider the following binary search tree in Fig. 2.8.

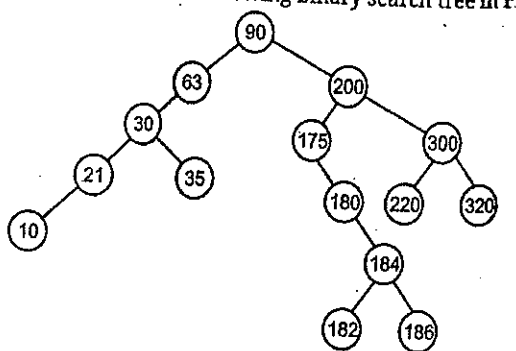


Fig. 2.8: Binary Search Tree (BST)

Height of tree with root 90 (90) = 1 + max (height (63), height (200))
Height of (63) = 1 + height (30)
Height (30) = 1 + max (height (21), height (35))
Height (21) = 1 + height (10)
Height (10) = 1

Therefore,

Height (21) = 1 + 1 = 2
Height (30) = 1 + max (2, 1) = 1 + 2 = 3
Height (63) = 1 + 3 = 4
Height (200) = 1 + max (height (175), height (300))
Height (175) = 1 + height (180)
Height (180) = 1 + height (184)
Height (184) = 1 + max (height (182), height (186))
Height (182) = 1
Height (186) = 1

Height (184) = 1 + 1 = 2
Height (180) = 1 + 2 = 3
Height (175) = 1 + 3 = 4
Height (200) = 1 + max. (height (175), height (300))
= 1 + max (4, 2)
= 1 + 4 = 5
Height (90) = 1 + max (height (63), height (200))
= 1 + max (4, 5)
= 1 + 5 = 6

- Thus, this tree has height 6. But from this we do not get any information about balance of height. The tree is said to be balanced if the difference in the right subtree and left subtree is not more than 1.
- Consider the above example in which all the leaf nodes have a balance factor of 0.

BF (21) = h_L (10) – 0 = 1 – 0 = 1
BF (30) = $h_L - h_R$ = 2 – 1 = 1
BF (63) = 3 – 0 = 3
BF (184) = 1 – 1 = 0
BF (180) = 0 – 2 = – 2
BF (175) = 0 – 3 = – 3
BF (300) = 1 – 1 = 0
BF (200) = 4 – 2 = 2
BF (90) = 4 – 5 = – 1

- Hence, the above tree is not height balanced. In order to balance a tree, we have to perform rotations on the tree.

2.2 Rotations

- Rotation is the process of moving nodes either to left or to right to make the tree balanced. To balance itself, an AVL tree may perform the following four kinds of rotations:
  1. Left Rotation (LL Rotation)
  2. Right Rotation (RR Rotation)
  3. Left-Right Rotation (LR Rotation)
  4. Right-Left Rotation (RL Rotation)
- The first two rotations are single rotations and the next two rotations are double rotations.

Single Left Rotation (LL Rotation):

- In LL Rotation, every node moves one position to left from the current position.
- To understand LL Rotation, let us consider the following insertion operation in AVL tree.

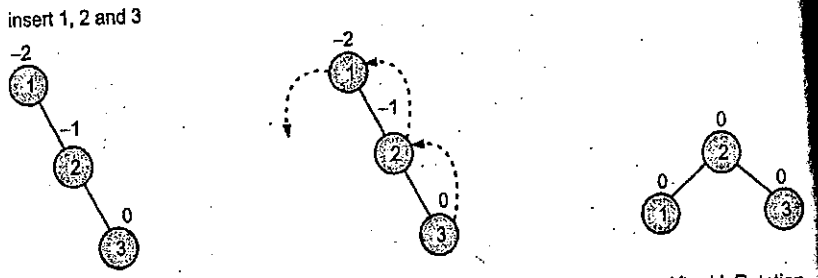


Fig. 2.9

Single Right Rotation (RR Rotation):

- In RR Rotation, every node moves one position to right from the current position.
- To understand RR Rotation, let us consider the following insertion operation in A tree.

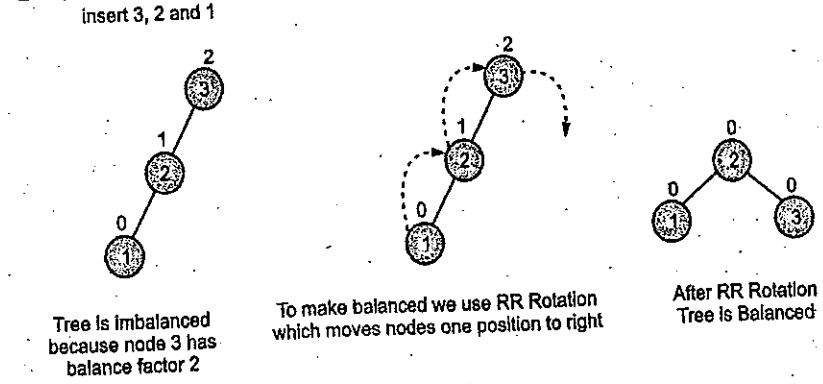


Fig. 2.10

Left Right Rotation (LR Rotation):

- The LR Rotation is a sequence of single left rotation followed by a single right rotation
- In LR Rotation, at first, every node moves one position to the left and one position t right from the current position.
- To understand LR Rotation, let us consider the following insertion operation in AV tree.

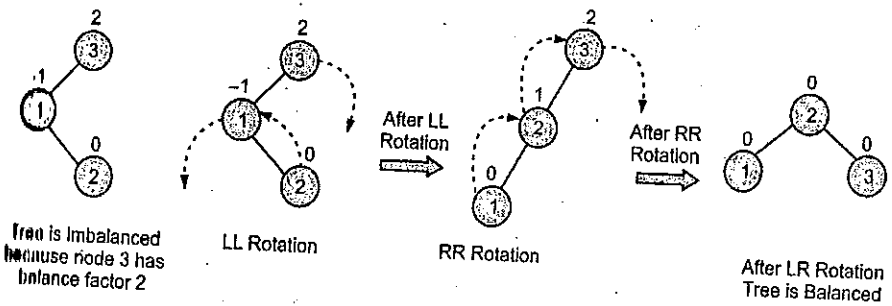


Fig. 2.11

Right Left Rotation (RL Rotation):

- The RL Rotation is sequence of single right rotation followed by single left rotation.
- In RL Rotation, at first every node moves one position to right and one position to left from the current position.
- To understand RL Rotation, let us consider the following insertion operation in AVL tree.

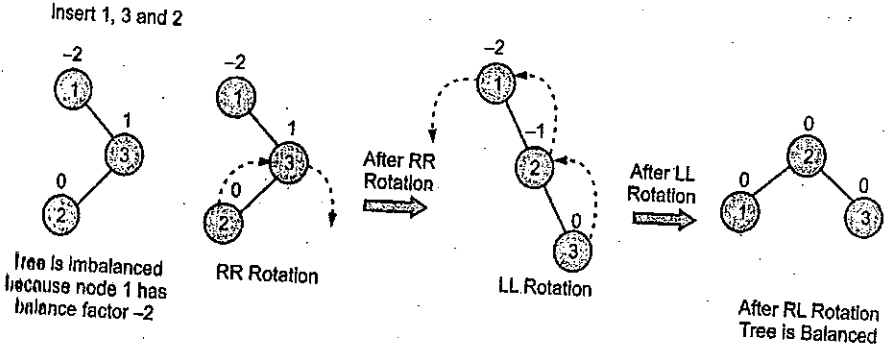
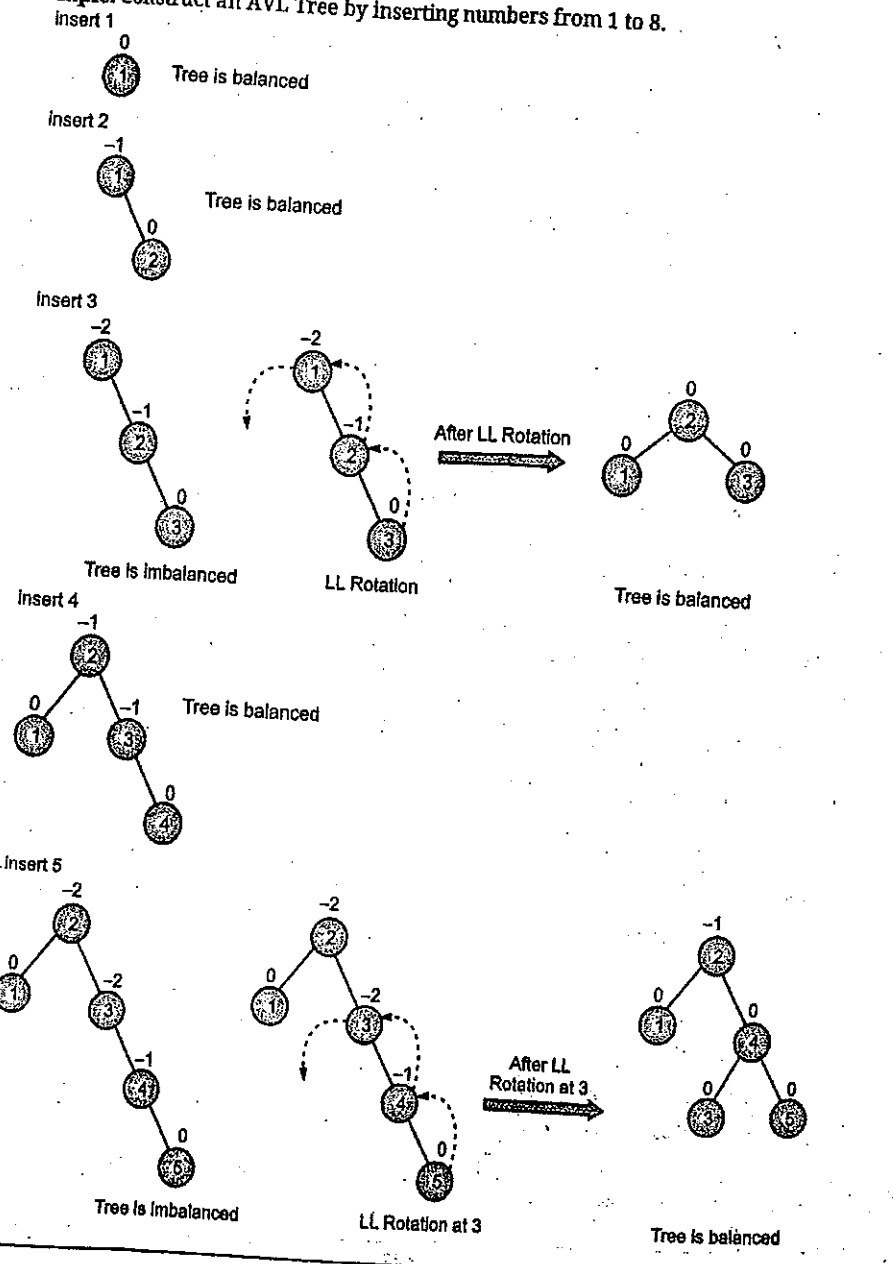


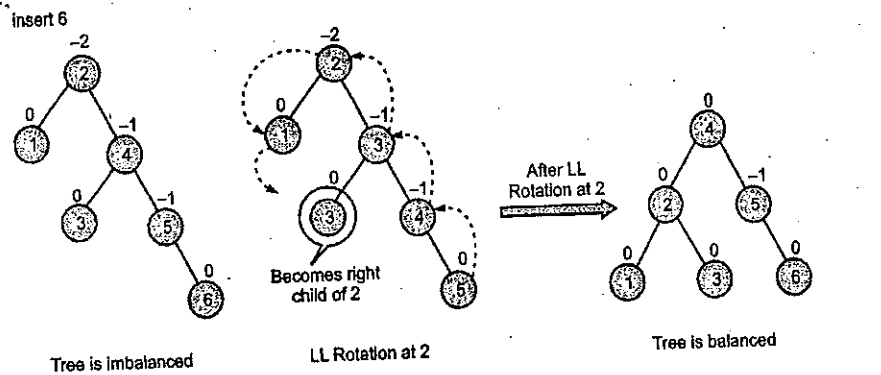
Fig. 2.12

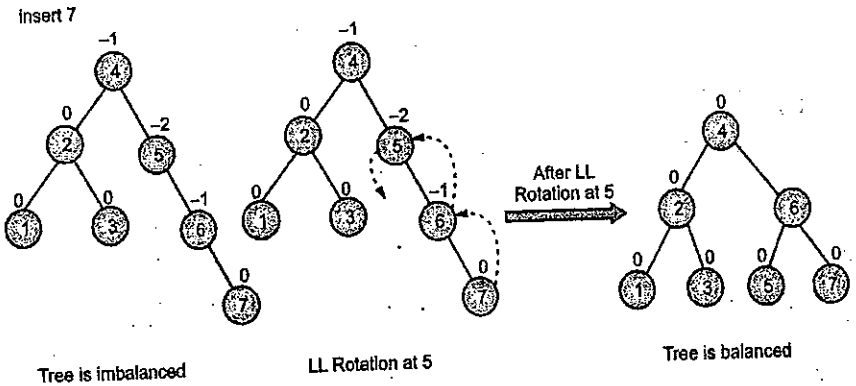
Operations on AVL Tree:

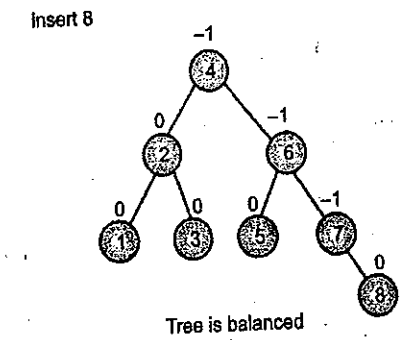
- The operations are performed on AVL tree are search, insert and delete.
- The search operation in the AVL tree is similar to the search operation in a binary search tree. In AVL tree, a new node is always inserted as a leaf node.
- The deletion operation in AVL tree is similar to deletion operation in BST. But after every deletion operation, we need to check with the Balance Factor (BF) condition. If the tree is balanced after deletion go for next operation otherwise perform suitable rotation to make the tree balanced.

Example: Construct an AVL Tree by inserting numbers from 1 to 8.









Example: Delete the node 30 from the AVL tree shown in the Fig. 2.13.

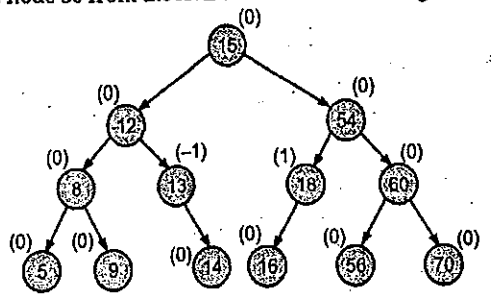


Fig. 2.13

Step 1:

- The node to be deleted from the tree is 8.
- If we observe it is the parent node of the node 5 and 9.
- Since the node 8 has two children it can be replaced by either of its child nodes.

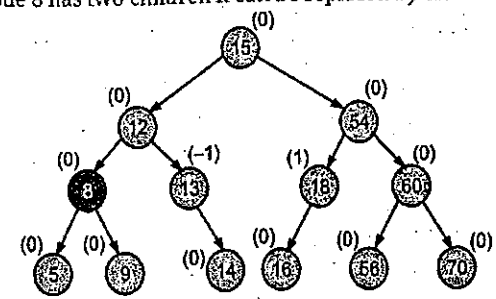


Fig. 2.14

Step 2:

- The node 8 is deleted from the tree.
- As the node is deleted we replace it with either of its children nodes.
- Here we replaced the node with the inorder successor, i.e. 9.
- Again we check the balance factor for each node.

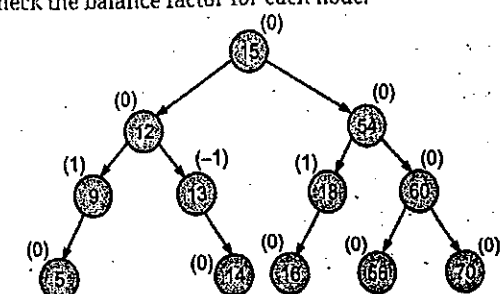


Fig. 2.15

Step 3:

- Now The next element to be deleted is 12.
- If we observe, we can see that the node 12 has a left subtree and a right subtree.
- We again can replace the node by either it's inorder successor or inorder predecessor.
- In this case we have replaced it by the inorder successor.

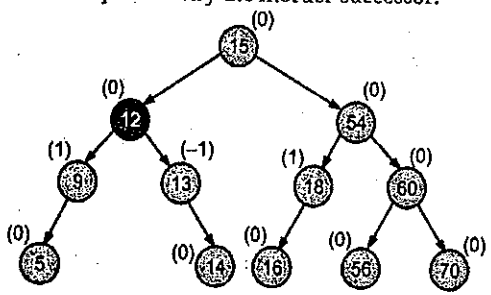


Fig. 2.16

Step 4:

- The node 12 is deleted from the tree.
- Since we have replaced the node with the inorder successor, the tree structure looks like shown in the Fig. 2.17.
- After removal and replacing check for the balance factor of each node of the tree.

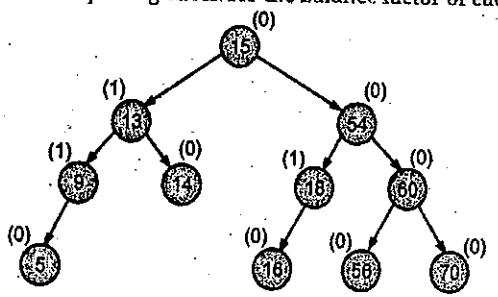


Fig. 2.17

Step 5:

- The next node to be eliminated is 14.
- It can be seen clearly in the image that 14 is a leaf node.
- Thus it can be eliminated easily from the tree.

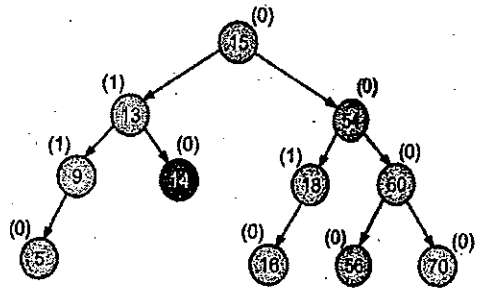


Fig. 2.18

Step 6:

- As the node 14 is deleted, we check the balance factor of all the nodes.
- We can see the balance factor of the node 13 is 2.
- This violates the terms of the AVL tree thus we need to balance it using the rotation mechanism.

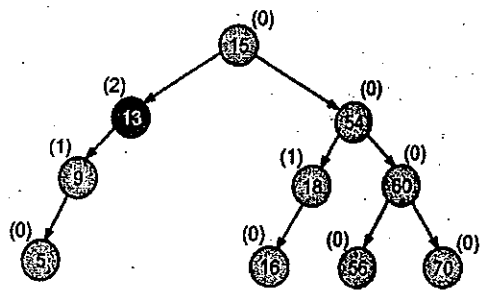


Fig. 2.19

Step 7:

- In order to balance the tree, we identify the rotation mechanism to be applied.
- Here, we need to use LL Rotation.
- The nodes involved in the rotation is shown in Fig. 2.20.

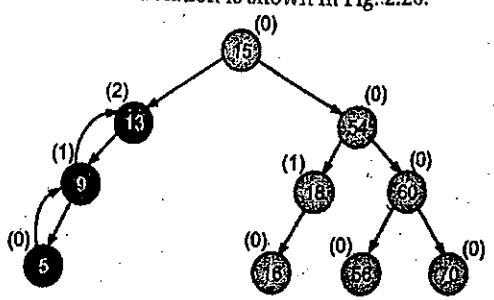


Fig. 2.20

Step 8:

- The nodes are rotated and the tree satisfies the conditions of an AVL tree.
- The final structure of the tree is shown as follows.
- We can see all the nodes have their balance factor as '0', '1' and '-1'.

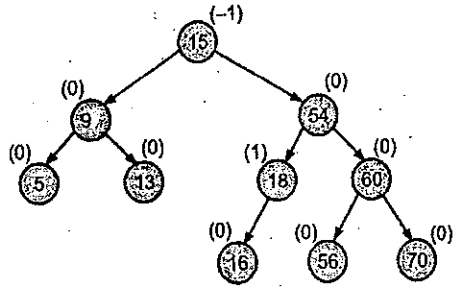


Fig. 2.21

Examples: [April 16, 17, 18, 19, Oct. 17, 18]

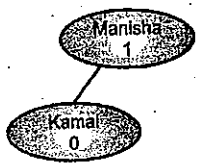
Example 1: Create AVL tree for the following data:

Manisha, Kamal, Archana, Reena, Nidhi, Shalaka, Priya, Leena, Meena

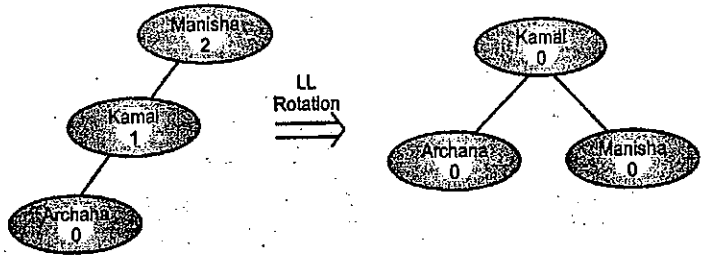
Solution: We check the BST is balanced, if not, rotation is performed. The number within each node indicates the balance factor. It is – 1, 0, 1 if tree is balanced.

(i) Manisha

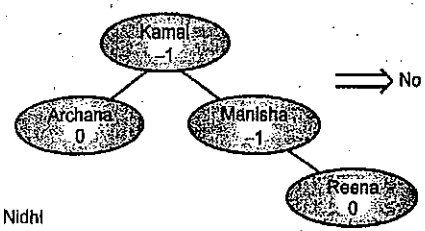
(ii) Kamal



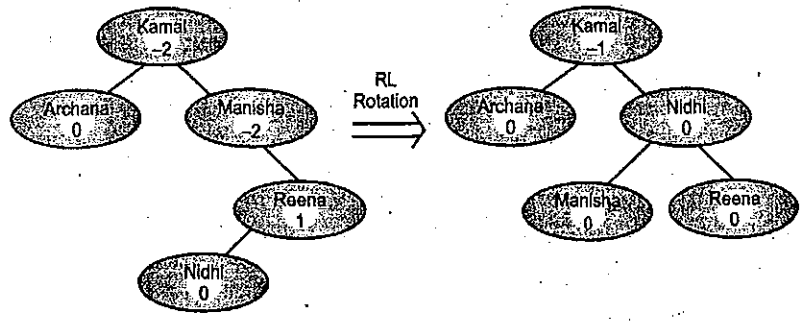
(iii) Archana



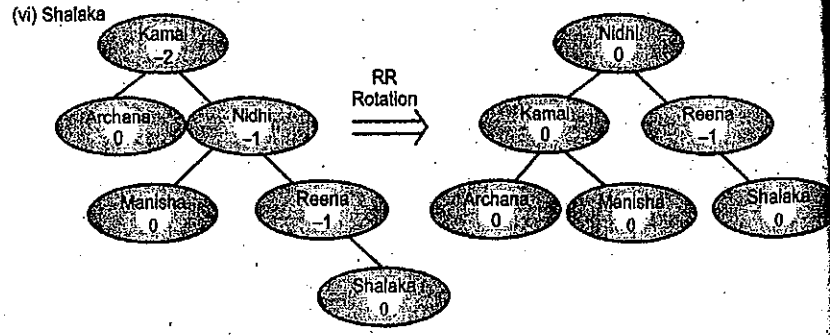
(iv) Reena



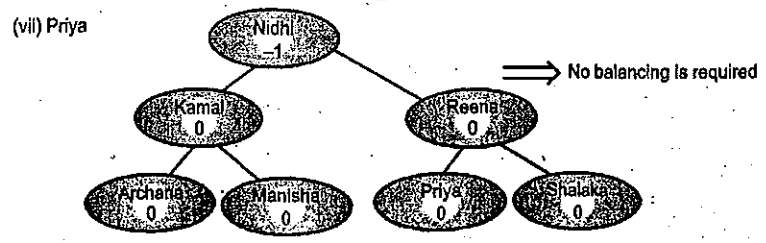
(v) Nidhi



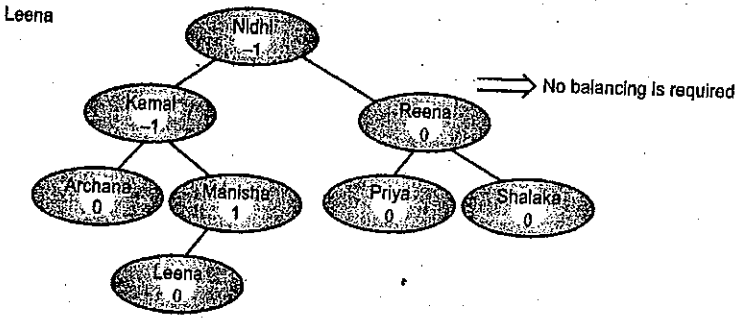
(vi) Shalaka



(vii) Priya

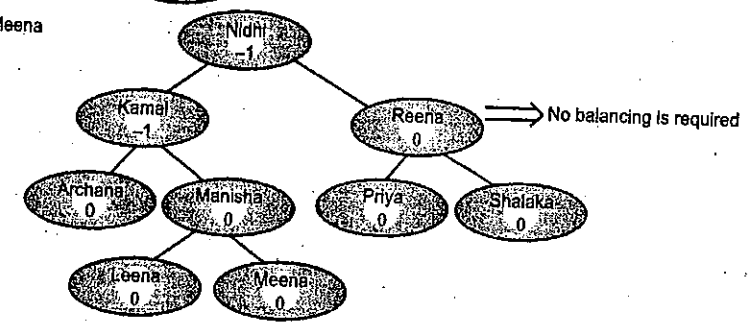


(viii) Leena



⟹ No balancing is required

(ix) Meena



⟹ No balancing is required

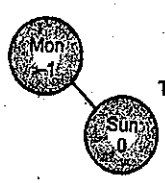
Example 2: Consider AVL tree for following data:

Mon Sun Thur Fri Sat Wed Tue

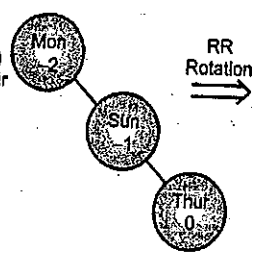
Solution:

(I)

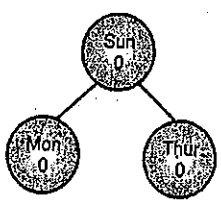
(II) Sun



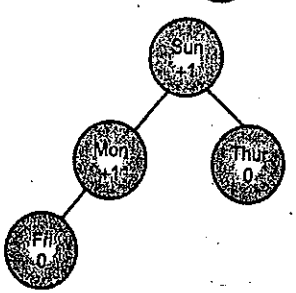
(III) Thur



RR Rotation ⟹

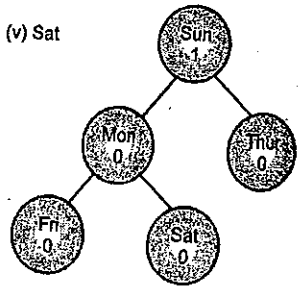


(iv) Fri

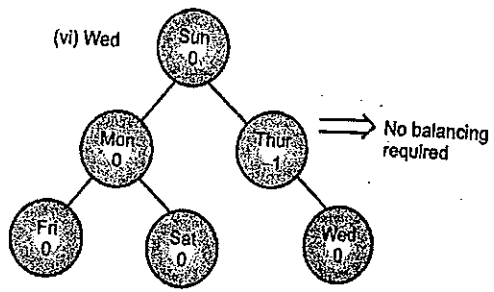


⟹ No balancing required

(v) Sat

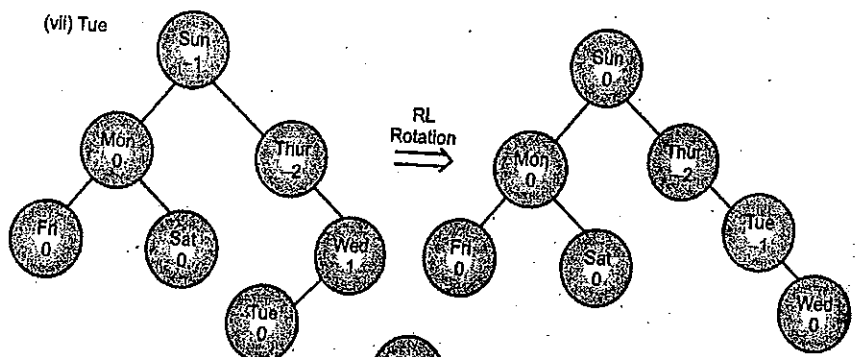


(vi) Wed



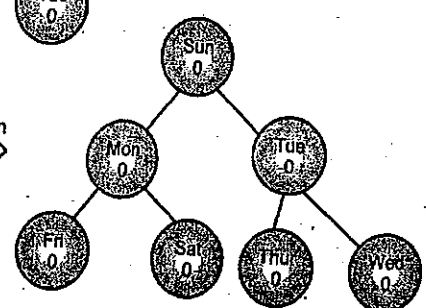
⟹ No balancing required

(vii) Tue



RL Rotation ⟹

RL Rotation ⟹

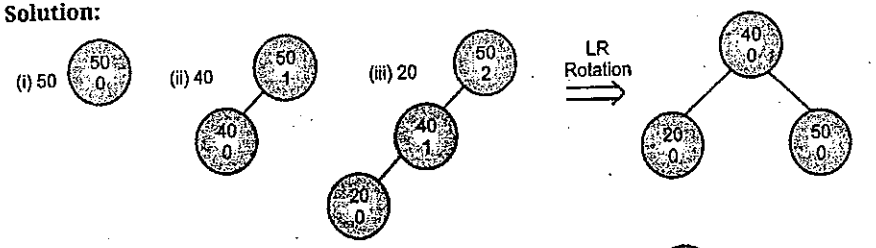


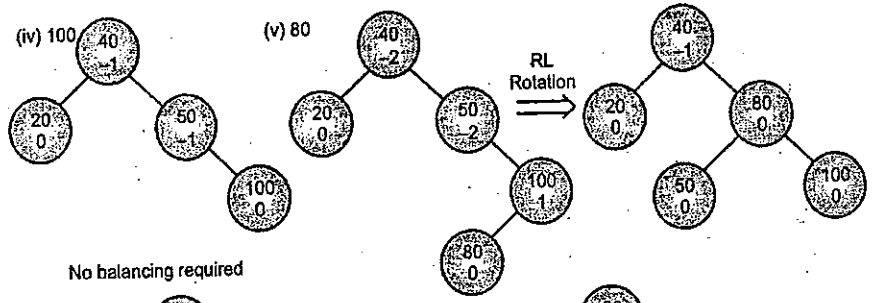
This is now balance tree.

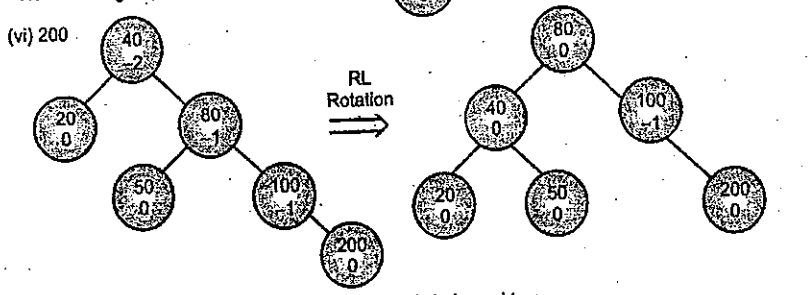
Example 3: Construct an AVL tree for the following data:

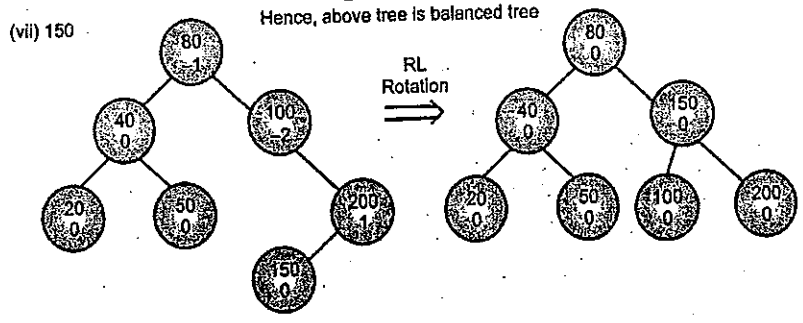
50, 40, 20, 100, 80, 200, 150

Solution:







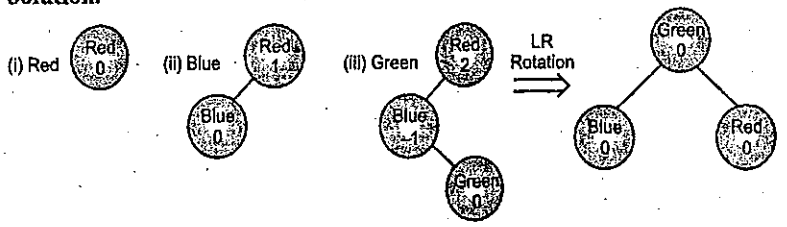


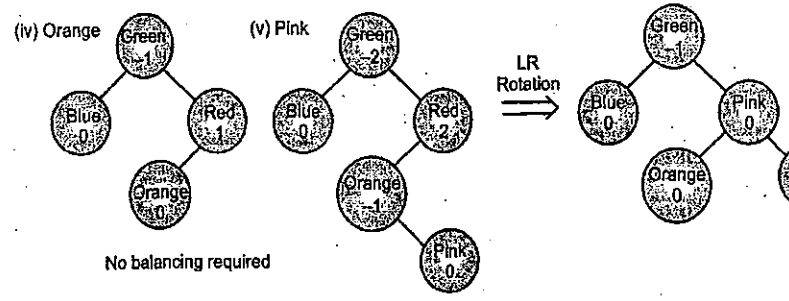
Hence, above tree is balanced tree.

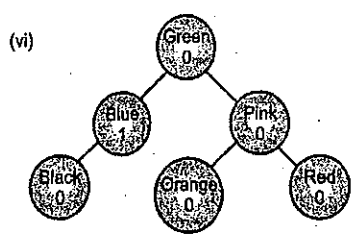
Example 4: Construct AVL tree for following data:

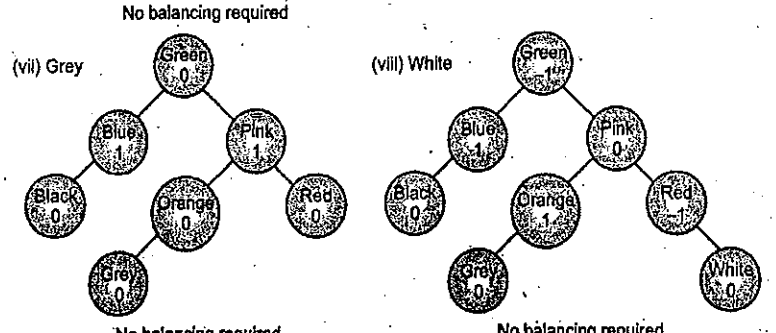
Red, blue, green orange, pink, black, grey, white, violet

Solution:









Hence, the tree is balance tree.

RED BLACK TREE

- In this section we will study concept of red black tree and its operations like insertion and deletion.

Concept

- A red-black tree is a type of binary search tree. It is self-balancing like the AVL tree.
- They are called red-black trees because each node in the tree is labeled/colored as red or black.

Properties of Red-Black Tree:

1. Each node is either red or black.
2. The root of the tree is always black.
3. All leaves are NULL/NIL and they are black.
4. If a node is red, then its children are black.
5. Any path from a given node to any of its descendant leaves contains the same amount of black nodes.

- Fig. 2.22 shows a representation of a red-black tree. Notice how each leaf is actually a black, null value.

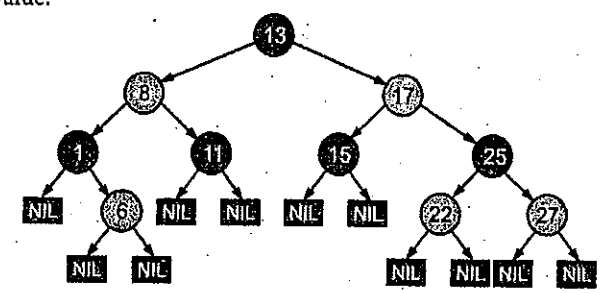


Fig. 2.22: Red-Black Tree

Operations

- In a red-black tree, there are two operations that can change the structure of the tree, insert and delete.
- These changes might involve the addition or subtraction of nodes, the changing of a node's color, or the re-organization of nodes via a rotation.

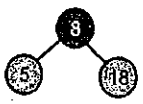
Insertion into Red Black Tree:

- In a red black tree, every new node must be inserted with the color red. The insertion operation in red black tree is similar to insertion operation in binary search tree. But it is inserted with a color property.
- After every insertion operation, we need to check all the properties of red-black tree. If all the properties are satisfied, then we go to next operation otherwise we perform the operations like recolor, rotation, rotation followed by recolor to make it red black tree.

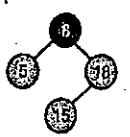
Insert (8)
Tree is Empty. So insert new node as Root node with black color.

Insert (18)
Tree is not Empty. So insert new node with red color.

Insert (5)
Tree is mat Empty. So insert new node with red color.

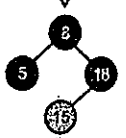


Insert (15)
Tree is not Empty. So insert new node with red color.



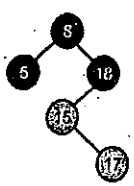
Here there are two consecutive Red nodes (18 and 15).
The new node's parent sibling color is Red
and parent's parent is root node.
So we use RECOLOR to make it Red Black Tree.

After RECOLOR



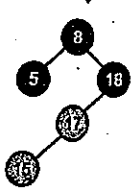
After Recolor operation, the tree is satisfying all Red Black Tree properties.

Insert (17)
Tree is not Empty. So insert new node with red color.

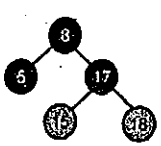


Here there are two consecutive
Red nodes (15 and 17).
The new node's parent sibling is NULL.
So we need rotation.
Here, we need LR Rotation and Recolor

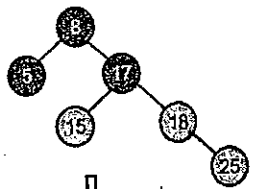
After Left Rotation



After Right Rotation
and Recolor

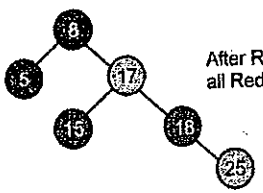


insert (25)
Tree is not Empty. So insert new node with red color.



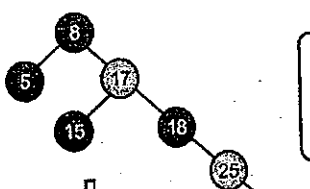
Here there are two consecutive
Red nodes (18 and 25),
The new node's parent sibling color is Red
and parent's parent is not root node.
So we use RECOLOR and Recheck.

After Recolor



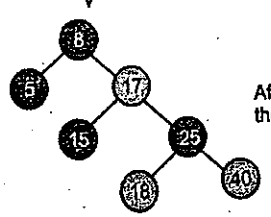
After Recolor operation, the tree is satisfying
all Red Black Tree properties.

insert (40)
Tree is not Empty. So insert new node with red color.



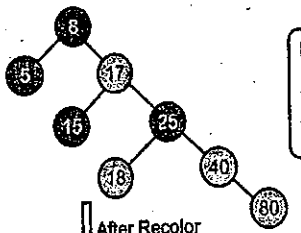
Here there are two consecutive
Red nodes (25 and 40).
The new node's parent sibling is NULL
So we need a Rotation and Recolor.
Here, we use LL Rotation and Recheck

After LL Rotation
and Recolor



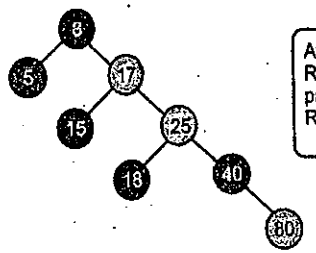
After LL Rotation and Recolor operation,
the tree is satisfying all Red Black Tree properties.

Insert (80)
Tree is not Empty. So insert new node with red color.



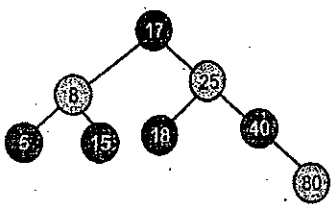
Here there are two consecutive
Red nodes (40 and 80). The new node's
parent sibling color is Red
and parent's parent is not root node.
So we use RECOLOR and Recheck.

After Recolor



After Recolor again there are two consecutive
Red nodes (17 and 25). The new node's
parent sibling color is Black. So we need
Rotation. We use Left Rotation and Recolor.

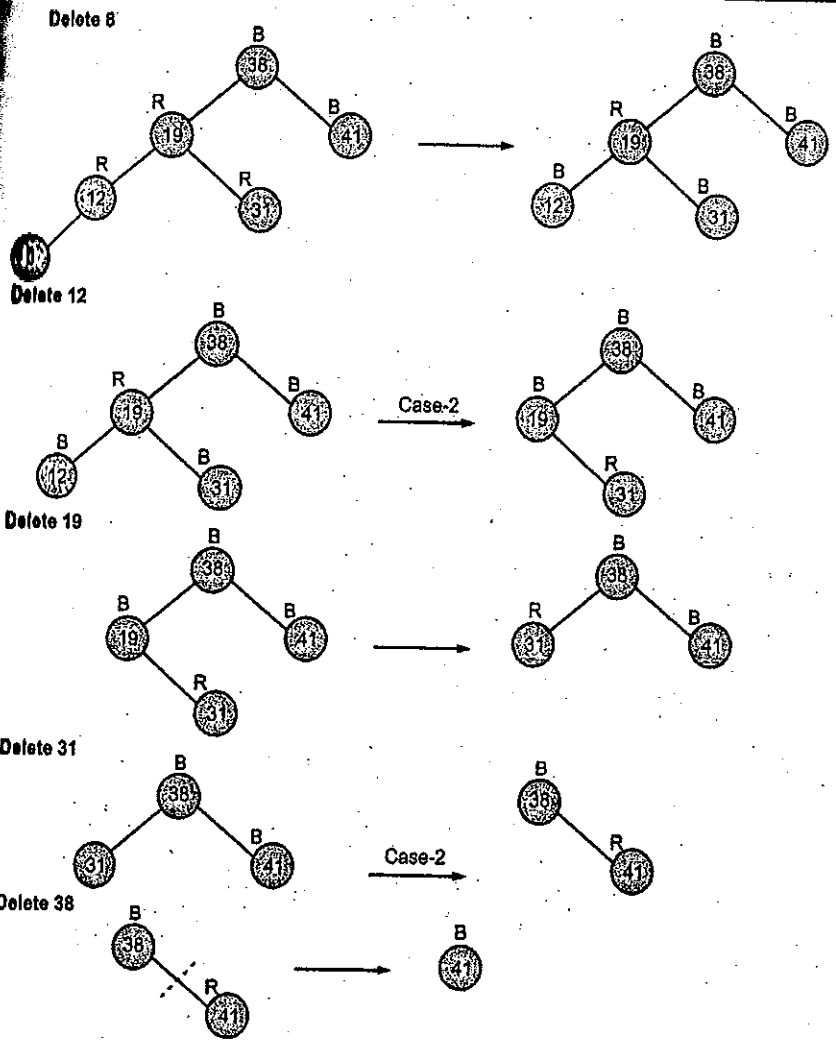
After Left Rotation and Recolor



Finally above tree is satisfying all the properties of Red Black Tree and
It is a perfect Red Black tree.

Deletion Red Black Tree:

- The deletion operation in red black tree is similar to deletion operation in BST. after every deletion operation, we need to check with the red-black tree properties.
- If any of the properties are violated then make suitable operations like reco rotation and rotation followed by recolor to make it red-black tree.
- In this example, we show the red black trees that result from the successful deletio the keys in the order 8, 12, 19,31,38,41.



MULTIWAY SEARCH TREE

- A multiway tree is a tree that can have more than two children. If a multiway tree can have maximum m children, then this tree is called as multiway tree of order m (or an m-way search tree).

- A multiway tree can have more than one value/child per node. They are written as m-way trees where the m means the order of the tree. A multiway tree can have m-1 values per node and m children.
- An m-way tree is a search tree in which each node can have from 0 to m subtrees, where m is defined as the B-tree order.
- Fig. 2.23 shows an example of m-way tree of order 4.

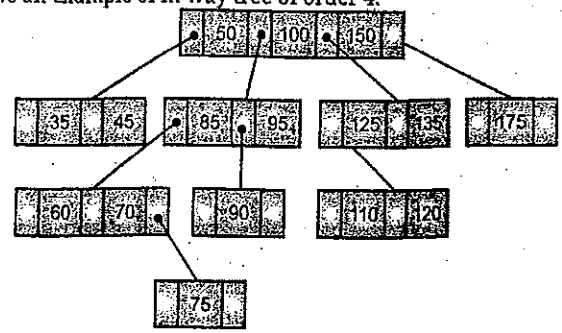


Fig. 2.23: Four-way Tree

2-3 Multi-way Search Tree:

- The 2-3 trees were invented by John Hopcroft in 1970. A 2-3 tree is a B-tree of order 3.
- A 2–3 tree is a tree data structure, where every node with children (internal node) has either two children (2-node) and one data element or three children (3-nodes) and two data elements.
- Fig. 2.24 shows complete 2-3 multi-way search tree. It has the maximum number of entries for its heights.

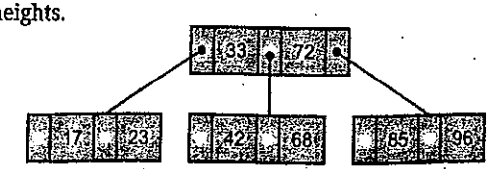


Fig. 2.24: Complete 2-3 Tree

2–3–4 Multi-way Search Tree:

- A 2-3-4 tree is a B-tree of order 4. It is also called as called a 2–4 tree.
- In 2-3-4 tree every node with children (internal node) has either two, three, or four child nodes.
- Fig. 2.25 shows 2-3-4 tree.

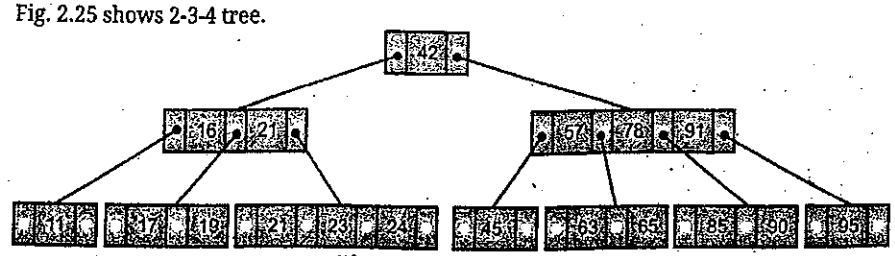


Fig. 2.25: 2-3-4 Tree

2.4.1 B-Tree

- In search trees like binary search tree, AVL tree, red-black tree, etc., every node contains only one value (key) and a maximum of two children.
- But there is a special type of search tree called B-tree in which a node contains more than one value (key) and more than two children.
- B-Tree was developed in the year 1972 by Bayer and McCreight with the name Height Balanced m-way Search Tree. Later it was named as B-Tree.
- A B-tree is a specialized m-way tree that is widely used for disk access. A B tree of order m can have a maximum of m–1 keys and m pointers to its sub-trees.
- B-tree is a type of tree in which each node can store multiple values and can point to multiple subtrees.
- The B-trees are useful in case of very large data that cannot accommodated the main memory of the computer and is stored on disks in the form of files.
- The B-tree is a self-balancing search tree like AVL and red-black trees. The main objective of B-trees is to minimize/reduce the number of disk accesses for accessing a record.
- Every B-tree has an order. B-tree of order m has the following properties:
  1. All leaf nodes must be at same level.
  2. All nodes except root must have at least [m/2]-1 keys and maximum of m-1 keys.
  3. All non-leaf nodes except root (i.e. all internal nodes) must have at least m/2 children.
  4. If the root node is a non-leaf node, then it must have at least two children.
  5. A non-leaf node with n-1 keys must have n number of children.
  6. All the key values in a node must be in Ascending Order.
- Fig. 2.26 shows an example of B-tree of order 5.

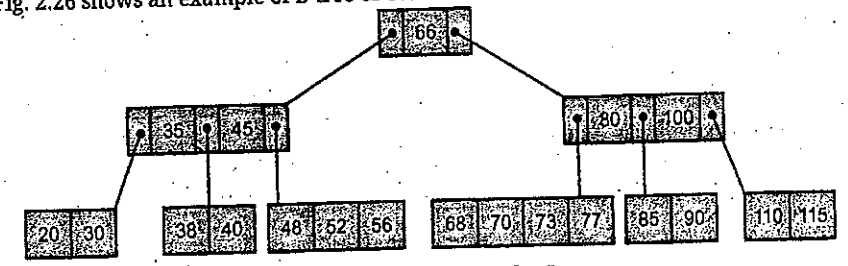


Fig. 2.26: B-Tree of Order 5

2.4.2 Insertion and Deletion Operations on B-Tree

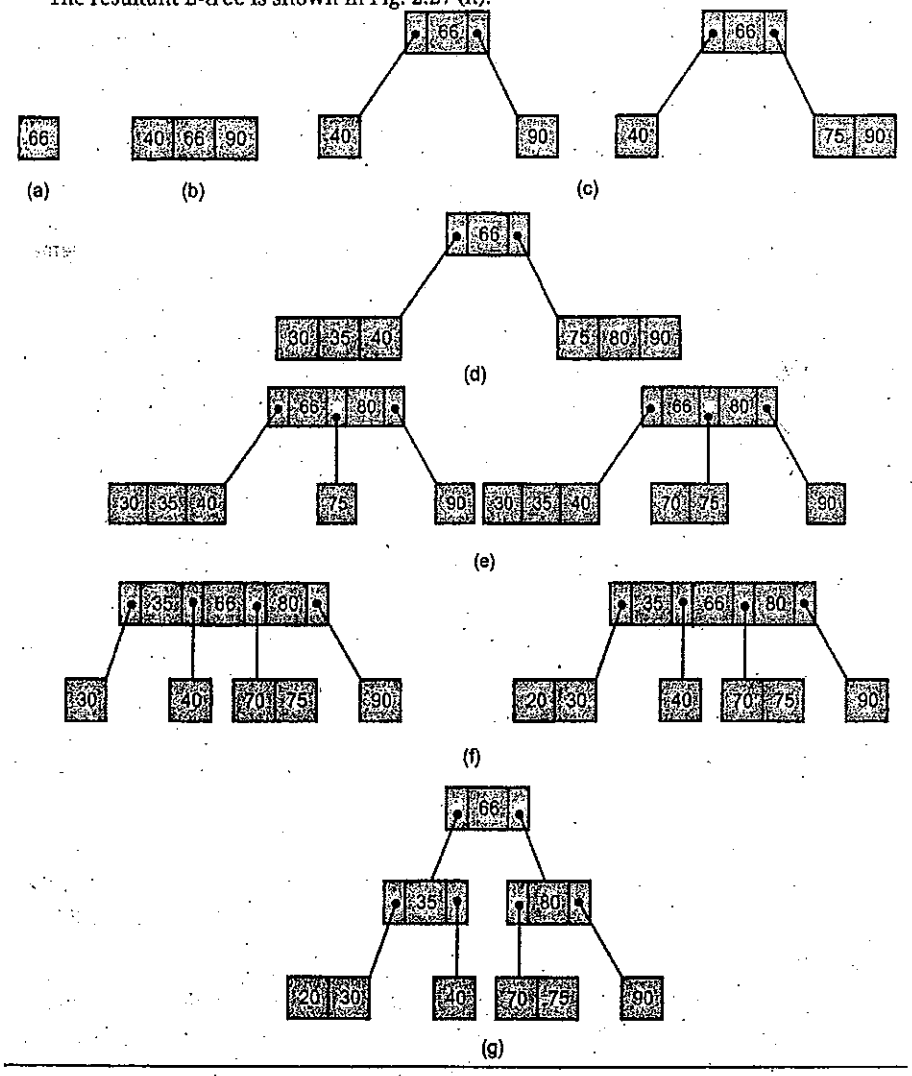
- The two most common operations insert and delete are performed on B-trees.

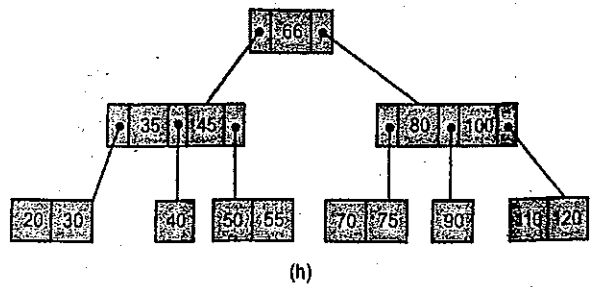
Insertion Operation on B-Tree:

- In a B-tree, all the insertion operations take place at the leaf nodes. To insert an element, first the appropriate leaf node is found and then an element is inserted in that node.

- Now, while inserting the element in the searched leaf node following one of the t cases may arise:
  1. There may be a space in the leaf node to accommodate more elements. In t case, the element is simply added to the node in such a way that the order elements is maintained.
  2. There may not be a space in the leaf node to accommodate more elements. In t case, after inserting new element in the full leaf node, a single middle elemen selected from these elements and is shifted to the parent node and the leaf is s into two nodes namely, left node and right node (at the same level).
- All the elements less than the middle element are placed in the left node and all elements greater than the middle element are placed in the right node.
- If there is no space for middle element in the parent node, the splitting of the par node takes place using the same procedure.
- The process of splitting may be repeated all the way to the root. In case the splitting root node takes place, a new root node is created that comprises the middle elem from the old root node.
- The rest of the elements of the old root node are distributed in two nodes created a result of splitting. The splitting of root node increases the height of B-tree by one.
- For example, consider the following step-by-step procedure for inserting elements the B-tree of order 4, i.e. any node can store at most 3 elements and can point to most 4 subtrees.
- The elements to be inserted in the B-tree are 66, 90, 40, 75, 30, 35, 80, 70, 20, 50, 45, 110, 100, and 120.
- The element 66 forms the part of new root node, as B tree is empty initi [Fig. 2.27 (a)].
- Since, each node of B tree can store up to 3 elements, the elements 90 and 40 a become part of the root node [Fig. 2.27 (b)].
- Now, since the root node is full, it is split into two nodes. The left node stores 40, th right node stores 90, and middle element 66 becomes the new root node. Since, 75 less than 90 and greater than 66, it is placed before 90 in the right node [Fig. 2.27 (c)]
- The elements 30 and 35 are inserted in left sub tree and the element 80 is inserted the right sub tree such that the order of elements is maintained [Fig. 2.27 (d)].
- The appropriate position for the element 70 is in the right sub tree, and since there no space for more elements, the splitting of this node takes place.
- As a result, the middle element 80 is moved to the parent node, the element 75 forr the part of the left sub tree (of element 80) and the element 90 forms the part of t right sub tree (of element 80). The new element 70 is placed before the element [Fig. 2.27 (e)].
- The appropriate position for the element 20 is in the left most sub tree, and sin there is no space for more elements, the splitting of this node takes place as discuss in the previous step. The new element 20 is placed before the element [Fig. 2.27 (f)].

- This tree can be used for future insertions, but a situation may arise when any of the sub trees splits and it will be required to adjust the middle element from that sub tree to the root node where there is no space for more elements.
- Hence, keeping in mind the future requirements, as soon as root node becomes full, splitting of root node must take place [Fig. 2.27 (g)]. This splitting of root node increases the height of tree by one.
- Similarly, other elements 50, 45, 55, 110, 100, and 120 can be inserted in this B-tree. The resultant B-tree is shown in Fig. 2.27 (h).





(h)

Fig. 2.27: Insertion Operation in B-Tree Deletion Operation on B-Tree:

- Deletion of an element from a B-tree involves following two steps:
  1. Searching the desired element and
  2. Deleting the element.
- Whenever, an element is deleted from *a* B-tree, it must be ensured that no property of B-tree is violated after the deletion.
- The element to be deleted may belong to either leaf node or internal node.
- Consider a B-tree in Fig. 2.28.

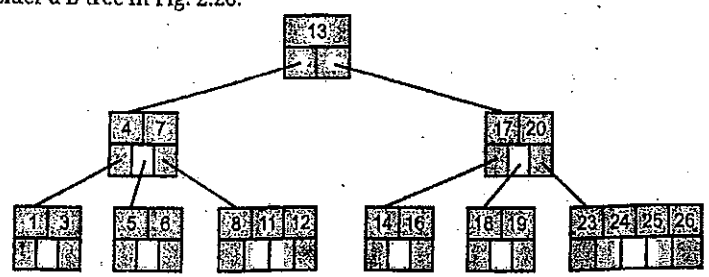


Fig. 2.28

- If we want to delete 8 then it is very simple (See Fig. 2.29).

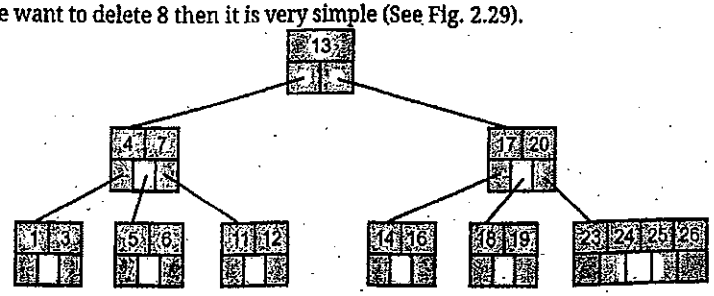


Fig. 2.29

- Now we will delete 20, the 20 is not in a leaf node so we will find its successor which is 23. Hence, 23 will be moved upto replace 20.

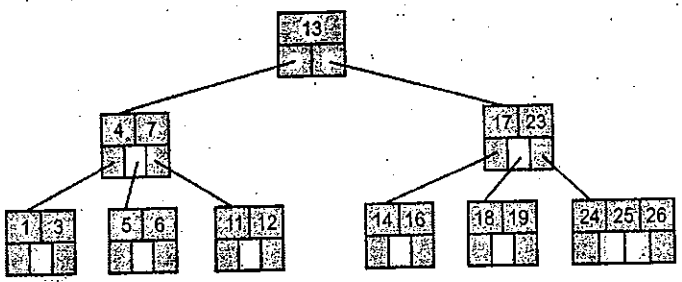


Fig. 2.30

- Next we will delete 18. Deletion of 18 from the corresponding node causes the node with only one key, which is not desired in B-tree of order 5.
- The slibling node to immediate right has an extra key. In such a case we can borrow a key from parent and move spare key of sibling to up (See Fig. 2.31).

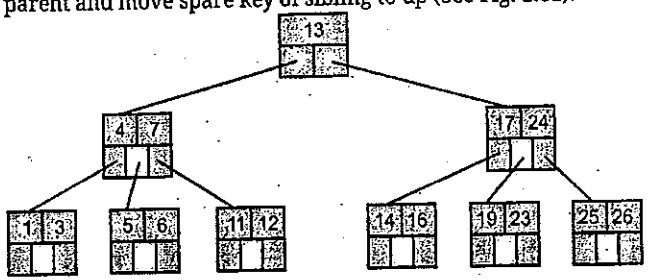


Fig. 2.31

- Now delete 5. But deletion of 5 is not easy. The first thing is 5 is from leaf node. Secondly this leaf node has no extra keys nor siblings to immediate left or right.
- In such a situation we can combine this node with one of the siblings. That means removes 5 and combine 6 with the node 1, 3.
- To make the tree balanced we have to move parent's key down. Hence, we will move 4 down as 4 is between 1, 3 and 6. The tree will be looks like as shown in Fig. 2.32.

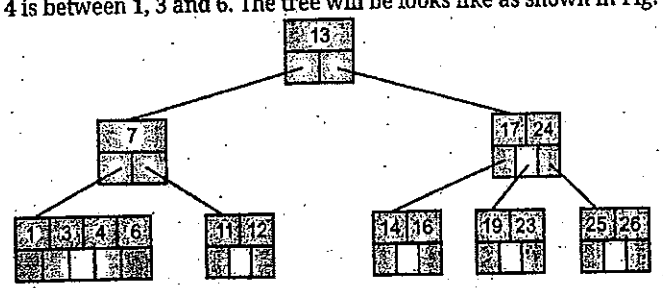


Fig. 2.32.

- But again internal node of 7 contains only one key which not allowed in B-tree. We then will try to borrow a key from sibling.

- But sibling 17, 24 has no spare key. Hence, what we can do is that, combine 7 wit and 17, 24. Hence the B-tree will be looks like as shown in Fig. 2.33.

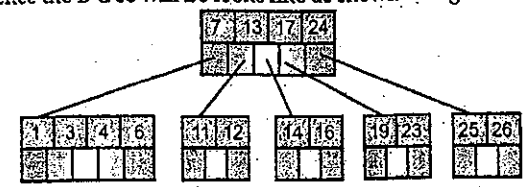


Fig. 2.33 ,

2.4.2 B+ Tree

- B+ (B Plus) Tree is a balanced, multi-way binary tree. The B+ Trees are exten version of B-Trees.
- A B+ tree is an m-ary tree with a variable but often large number of children per no A B+ tree consists of a root, internal nodes and leaves. The root may be either a lea a node with two or more children.
- A B+ tree can be viewed as a B-tree in which each node contains only keys (not k value pairs), and to which an additional level is added at the bottom with lin leaves.
- The B+ tree is a variation of B-tree in a sense that unlike B-tree, it includes all the values and record pointers only in the leaf nodes, and key values are duplicated internal nodes for defining the paths that can be used for searching purposes.
- In addition, each leaf node contains a pointer, which points to the next leaf node, t is, leaf nodes are linked sequentially (See Fig. 2.34).
- Hence, B+ tree supports fast sequential access of records in addition to the rand access feature of B-tree.
- Note that in case txt B+ trees, if key corresponding to the desired record is found any internal node, the traversal continues until its respective leaf node is reached access the appropriate record pointer.
- Fig. 2.34 shows a sample B+ tree.

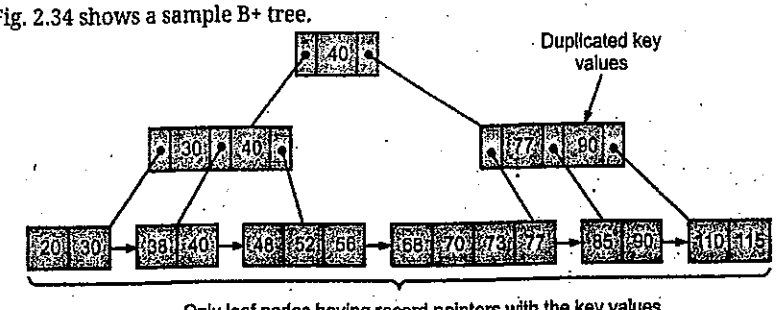


Fig. 2.34

4.2.2.1 Insertion and Deletion in B+ Tree

- The two most common operations insert and delete performed on B+ trees.

Insertion in B+ Tree:

- In B+ tree insert the new node as a leaf node.
- **Example:** Insert the value 195 into the B+ tree of order 5 shown in the Fig. 2.35.

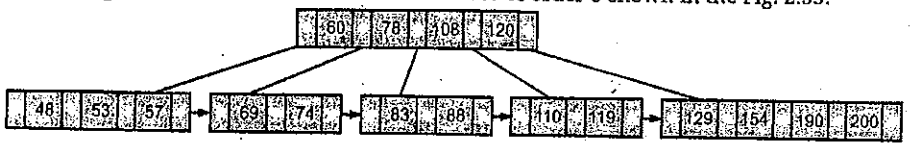


Fig. 2.35

- The value 195 will be inserted in the right sub-tree of 120 after 190. Insert it at the desired position (See Fig. 2.36).

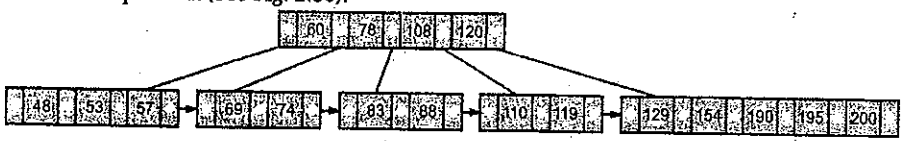


Fig. 2.36

- The node contains greater than the maximum number of elements i.e. 4, therefore split it and place the median node up to the parent.

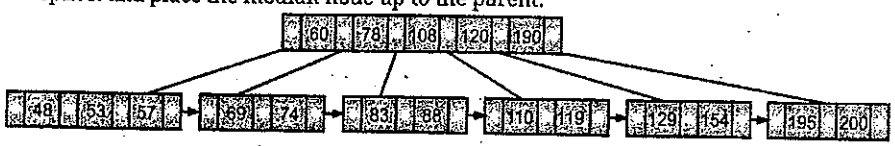


Fig. 2.37

- Now, the index node contains 6 children and 5 keys which violates the B+ tree properties, therefore we need to split it, shown in Fig. 2.38.

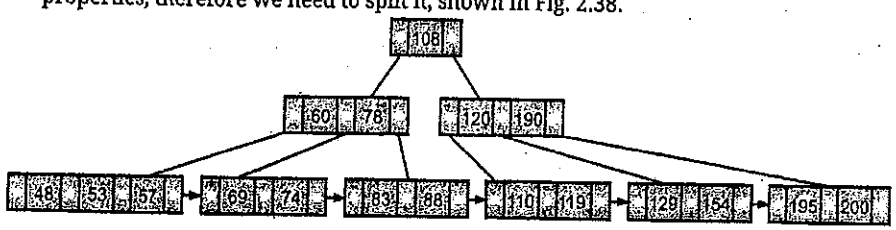


Fig. 2.38

Deletion in B+ Tree:

- Delete the key and data from the leaves.
- Delete the key 200 from the B+ tree shown in the Fig. 2.39.

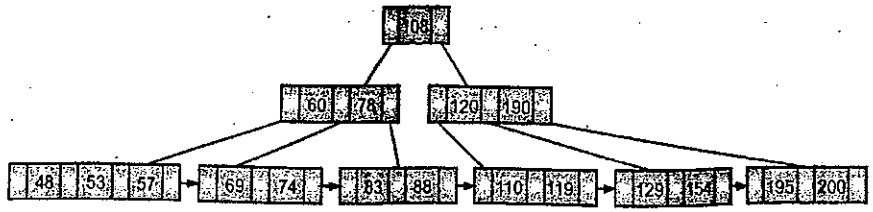


Fig. 2.39

- The value 200 is present in the right sub-tree of 190, after 195, delete it.

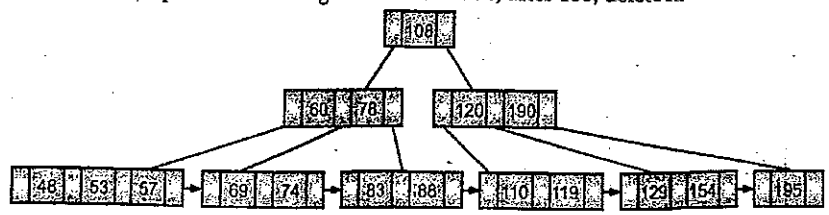


Fig. 2.40

- Merge the two nodes by using 195, 190, 154 and 129.

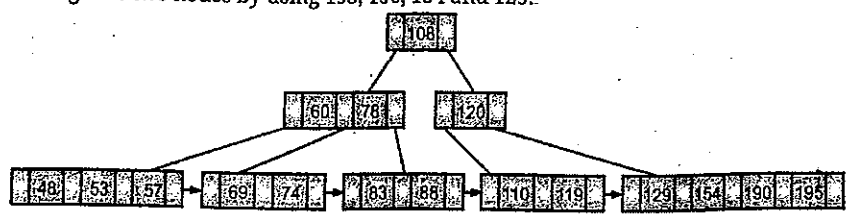


Fig. 2.41

- Now, element 120 is the single element present in the node which is violating the B+ Tree properties. Therefore, we need to merge it by using 60, 78, 108 and 120.
- Now, the height of B+ tree will be decreased by 1.

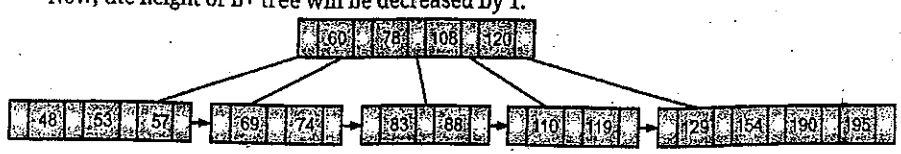


Fig. 2.42

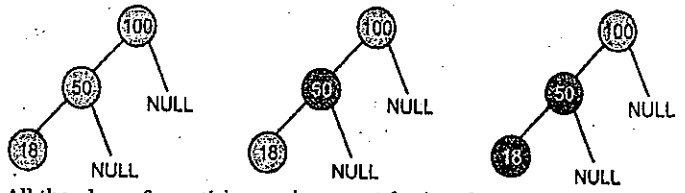
PRACTICE QUESTIONS

Q. I Multiple Choice Questions:

1. Which factor on many tree operations related to the height of the tree?
   (a) Efficiency (b) Degree
   (c) Sibling (d) Path

2. An AVL tree named after inventors,
   (a) Adelson (b) Velsky
   (c) Landis (d) All of these
3. Which tree is a binary search tree that is height balanced?
   (a) BST (b) B+
   (c) AVL (d) Red-black
4. Which tree is a self-balancing binary search tree in which each node contains an extra bit for denoting the color of the node, either red or black?
   (a) AVL (b) BST
   (c) Red-black (d) None of these
5. Which trees are a self-adjusting binary search tree?
   (a) BST (b) Splay
   (c) B+ (d) None of these
6. Which is is a tree-like data structure whose nodes store the letters of an alphabet?
   (a) Trie (b) AVL
   (c) B+ (d) Extended
7. Which search tree is one with nodes that have two or more children?
   (a) Multiway (b) Red-black
   (c) Splay (d) AVL
8. A binary tree is said to be balanced if the height of left and right children of every node differ by either,
   (a) -1 (b) +1
   (c) 0 (d) All of these
9. A 2-3 tree is a B-tree of order,
   (a) 2 (b) 1
   (c) 3 (d) None of these
10. In which tree a new element must be added only at the leaf node.
    (a) B-Tree (b) AVL
    (c) Red-black (d) Splay
11. A balanced binary tree is a binary tree in which the height of the left and right subtree of any node differ by not more than ____.
    (a) 0 (b) 1
    (c) 3 (d) 2
12. In an AVL tree which factor is the difference between the height of the left subtree and that of the right subtree of that node.
    (a) Root (b) Node
    (c) Degree (d) Balance
13. Which of the following is the most widely used external memory data structure?
    (a) AVL tree (b) B-tree
    (c) Lexical tree (d) Red-black tree

14. A B-tree of order 4 and of height 3 will have a maximum of _____ keys.
    (a) 255 (b) 63
    (c) 127 (d) 188
15. Consider the below formations of red-black tree:



All the above formations are incorrect for it to be a red black tree. then what m be the correct order?
    (a) 50-black root, 18-red left subtree, 100-red right subtree
    (b) 50-red root, 18-red left subtree, 100-red right subtree
    (c) 50-black root, 18-black left subtree, 100-red right subtree
    (d) 50-black root, 18-red left subtree, 100-black right subtree
16. What is the special property of red-black trees and what root should always be?
    (a) a color which is either red or black and root should always be black color
    (b) height of the tree
    (c) pointer to next node
    (d) a color which is either green or black

Answers

1. (a)	2. (d)	3. (c)	4. (c)	5. (b)	6. (a)	7. (a)
8. (d)	9. (c)	10.(a)	11. (b)	12. (d)	13. (b)	14. (a)
15. (a)	16. (a)					

Q. II Fill in the Blanks:

1. Balancing or self-balancing (height balanced) tree is a _____ search tree.
2. A binary tree is said to be balanced if, the difference between the heights of lef and right subtrees of every node in the tree is either _____.
3. The _____ tree is a variant of Binary Search Tree (BST) in which every node i colored either red or black.
4. _____ is a process in which a node is transferred to the root by performing suitable rotations.
5. _____ is a tree-based data structure, which is used for efficient retrieval of a key in a large data-set of strings.
6. _____ tree is a self-balancing binary search tree in which each node maintains extra information called a balance factor whose value is either -1, 0 or +1.
7. In the _____ m-ary tree, the key is represented as a sequence of characters.

8. Trie is an efficient information _____ data structure.
9. A splay tree also known as _____ tree is a type of binary search tree which reorganizes the nodes of tree to move the most recently accessed node to the root of the tree.
10. A _____ is a specialized M-way tree which is widely used for disk access.
11. A _____ tree is a multi-way search tree in which each node has two children (referred to as a two node) or three children (referred to as a three node).
12. 2-3-4 trees are B-trees of order _____.
13. The B-tree generalizes the binary search tree, allowing for nodes with more than _____ children.
14. In a splay tree, splaying an element rearranges all the elements in the tree so that splayed element is placed at the _____ of the tree.
15. A red-black tree is a _____ search tree in which each node is colored either red or black.

Answers

1. binary	2. -1, 0 or +1	3. red black	4. Splaying	5. Trie	6. AVL
7. lexical	8. retrieval	9. self-adjusting	10. B-Tree	11. 2-3	12. 4
13. two	14. root	15. binary			

Q. III State True or False:

1. An AVL tree is a self-balancing binary search tree.
2. The self-balancing property of an AVL tree is maintained by the balance factor. The value of balance factor should always be -1, 0 or +1.
3. An AVL tree is a binary search tree where the sub-trees of every node differ in height by at most 1.
4. B Tree is a self-balancing data structure based on a specific set of rules for searching, inserting, and deleting the data in a faster and memory efficient way.
5. Red-black Tree is a self-balancing Binary Search Tree (BST).
6. In a AVL tree, every operation is performed at the root of the tree. All the operations in splay tree are involved with a common operation called Splaying (the process of bringing it to the root position by performing suitable rotation operations).
7. A trie, also called digital tree or prefix tree used to store collection of strings.
8. To have an unbalanced tree, we at least need a tree of height 1.
9. Rotation is required only if, the balance factor of any node is disturbed upon inserting the new node, otherwise the rotation is not required.
10. The B-tree is a generalization of a binary search tree in that a node can have more than two children.
11. B+ Tree is an extension of B Tree which allows efficient insertion, deletion and search operations.

Answers

1. (T)	2. (T)	3. (T)	4. (T)	5. (T)	6. (F)	7. (T)
8. (F)	9. (T)	10. (T)	11. (T)			

Q. IV Answer the following Questions:

(A) Short Answer Questions:

1. What is height balance tree?
2. List operations on AVL tree.
3. What is Trie?
4. Define B-Tree.
5. List rotations on AVL tree.
6. Define multi-way search trees.
7. Define splay tree.
8. What is lexical search tree?
9. Define balance factor.

(B) Long Answer Questions:

1. Describe need for height balanced trees.
2. With the help of example describe concept of AVL tree.
3. Explain concept of red-black tree diagrammatically.
4. Write short note on: Trie.
5. Describe lexical search tree with example.
6. How to insert and delete operations carried by red-black tree? Explain with example.
7. Describe AVL tree rotations with example,
8. With explain B-Tree insert and delete operation.
9. Compare B-Tree and B+ Tree (any four points).
10. Write short note on: Splay tree.
11. With the help of example describe multi-way search tree.

UNIVERSITY QUESTIONS AND ANSWERS

April 2016

1. Construct AVL tree for the following data: [5 M]
   Mon., Wed., Tue., Sat., Sun., Thur.

Ans. Refer to Section 2.2, Examples.

April 2017

1. Construct AVL tree for the following data: [5 M]
   Pen, Eraser, Book, Scale, Sketch pen, Crayon, Color pencil.

Ans. Refer to Section 2.2, Examples.

October 2017

1. Define balance factor. [1 M]

Ans. Refer to Section 2.1, Point (1).

2. Construct AVL tree for the following data: [5 M]
NFD, ZIM, IND, AUS, NEL, ENG, SRL, PAK.

Ans. Refer to Section 2.2, Examples.

April 2018

1. Construct AVL tree for the following data: [5 M]
SUN, FRI, MON, WED, TUE, THUR, SAT.

Ans. Refer to Section 2.2, Examples.

October 2018

1. Construct AVL tree for the following data: [5 M]
55, 40, 25, 100, 80, 200, 150.

Ans. Refer to Section 2.2, Examples.

April 2019

1. Define balance factor. [1 M]

Ans. Refer to Section 2.1, Point (1).

2. Construct the AVL tree for the following data: [5 M]
Chaitra, Magh, Vaishakh, Kartik, Falgun, Aashadh.

Ans. Refer to Section 2.2, Examples.

■■■

CHAPTER 3

Grap

Objectives ...

- To study Basic Concepts of Graph Data Structure
- To learn Graph Terminology
- To understand Representation and Operations of Graph
- To study Graph Traversals
- To learn Greedy Strategy, Dynamic Programming Strategy of Graphs

3.0 INTRODUCTION

- Graph is one of the most important non-linear data structure. A graph is a pictor representation of a set of objects where some pairs of objects (vertices) are connecte by links (edges).
- A graph is a pair of sets (V, E), where V is the set of vertices and E is the set of edge connecting the pairs of vertices.

3.1 CONCEPT AND TERMINOLOGY

- In this section we study basic concepts and terminology in graph.

3.1.1 Concept

- A graph G is a set of two tuples G = (V, E) where, V is a finite non-empty set of vertices and E is the set of pairs of vertices called edges.
- Fig. 3.1 shows an example of graph.

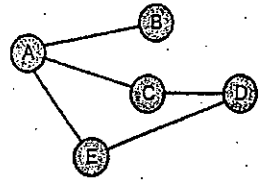


Fig. 3.1: Graph

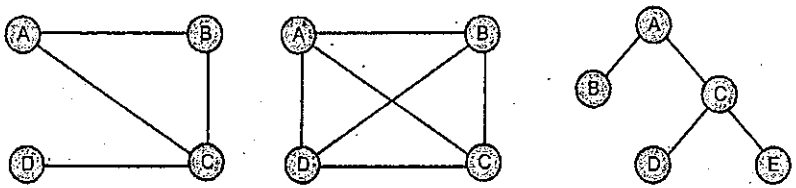
The set of vertices in the above graph is, V = {A, B, C, D, E}.
The set of edges, E = {(A, B), (A, C), (C, D), (A, E), (E, D)}.

- There are two types of Graph, Undirected graph and Directed graph.

1. **Undirected Graph:**

- In an undirected graph, the pair of vertices representing any edge is unordered i.e. the pairs (v_1, v_2) and (v_2, v_1) represent the same edge.
- In other words, the edges have no direction in undirected graph.

Example: Consider the Fig. 3.2.



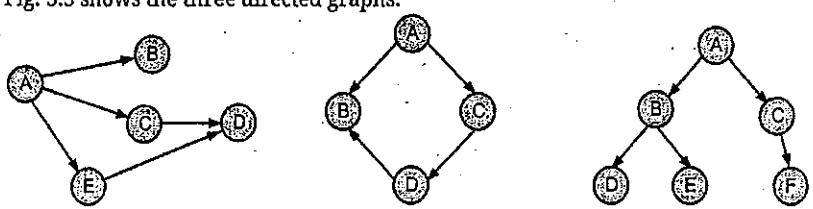
(a) (b) (c)

Fig. 3.2: Undirected Graph

- In Fig. 3.2 (a) G = (V, E) where V = {A, B, C, D} and E = {(A, B), (A, C), (B, C), (C, D)}.
- In Fig. 3.2 (b) G = (V, E) where V = {A, B, C, D} and E = {(A, B), (A, C), (A, D), (B, D), (B, C), (D, C)}.
- In Fig. 3.2 (c) G = (V, E) where V = {A, B, C, D, E} and E= {(A, B), (A, C), (C, D), (C, E)}. This undirected graph is called Tree. Tree is a special case of graph.

2. **Directed Graph:**

- In a directed graph each edge is represented by a directed pair (v_1, v_2), v_1 is the tail and v_2 is head of the edge i.e. the pairs (v_1, v_2) and (v_2, v_1) are different edges.
- In other words, the edges have direction in directed graph. Directed graph is also called as Digraph.
- Fig. 3.3 shows the three directed graphs.



(a) (b) (c)

Fig. 3.3: Directed Graphs

- In Fig. 3.3 (a) G = (V, E) where V = {A, B, C, D, E} and E = {(A, B), (A, C), (C, D), (A, E), (E, D)}.
- In Fig. 3.3 (b) G = (V, E) where V = {A, B, C, D} and E = {(A, B), (A, C), (C, D), (D, B)}.
- In Fig. 3.3 (c) G = (V, E) where V = {A, B, C, D} and E = {(A, B), (A, C), (B, D), (B, E), (C, F)}.

- Following table compares tree and graph data structures:

Sr. No.	Tree	Graph
1.	A tree is a data structure in which each node is attached to one or more nodes as children.	A graph is a collection of vertices or nodes, which are joined as pairs by lines (links) or edges.
2.	Tree is a non-linear data structure.	Graph is also a non-linear data structure.
3.	All the trees can be graphs.	All the graphs are not trees.
4.	The common tree traversal methods are inorder, preorder, postorder traversals.	The two common graph traversal method are Breadth First Search (BFS) and Depth First Search (DFS).
5.	It is undirected and connected.	It can be directed or undirected; can be connected or not-connected.
6.	It cannot be cyclic.	It can be cyclic or acyclic.
7.	There is a root (first) node in trees.	There is no root node in graph.
8.	Tree data is represented in hierarchical manner so parent to child relation exists between the nodes.	Graph did not represent the data is hierarchical manner so there is no parent child relation between data representation.
9.	A tree cannot have a loop structure.	A graph can have a loop structure.

3.2 Terminology

- Basic graph terminology listed below:

1. **Adjacent Vertex:** When there is an edge from one vertex to another then these vertices are called adjacent vertices. Node 1 is called adjacent to node 2 as there exists an edge from node 1 and node 2 as shown in Fig. 3.4.

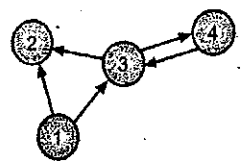


Fig. 3.4

2. **Cycle:** A path from a vertex to itself is called a cycle. Thus, a cycle is a path in which the initial and last vertices are same.

Example: Fig. 3.5 shows the path (A, B, C, A) or (A, C, D, B, A) are cycles of different lengths. If a graph contains a cycle it is cyclic otherwise acyclic.

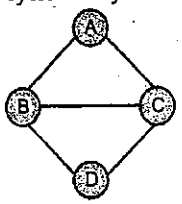


Fig. 3.5: Cycle

3. **Complete Graph:** A graph G is said to be complete if every vertex in a graph is adjacent to every other vertex. In this graph, number of edges = n(n-1)/2, where n = no. of vertices.

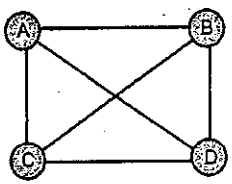


Fig. 3.6: Complete Graph

Example: In Fig. 3.6, Number of vertices n=4, Number of edges=4(4–1)/2=6.

4. **Connected Graph:** An undirected graph G is said to be connected if for every pair of distinct vertices v_i, v_j in V(G) there is a path from v_i to v_j.

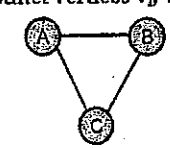


Fig. 3.7: Connected Graph

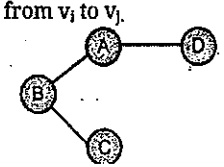
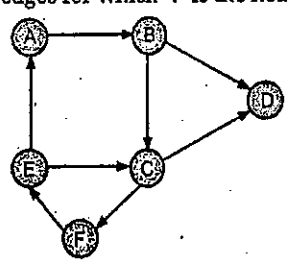


Fig. 3.8 Non-connected Graph

5. **Degree of a Vertex:** It is the number of edges incident to a vertex. It is written as degree(V), where V is a vertex.
6. **In-degree of a Vertex:** In directed graph, in-degree of a vertex 'v' is the number of edges for which 'v' is the head.



Indegree of vertex B = 1
Indegree of vertex C = 2
Indegree of vertex D = 2
Indegree of vertex E = 1
Indegree of vertex F = 1
Indegree of vertex A = 1

Fig. 3.9: Digraph

7. **Out-degree of a Vertex:** In directed graph, the out-degree of a vertex 'v' is the t number of edges for which 'v' is the tail.

From Fig. 3.9:
Outdegree of A = 1
Outdegree of B = 2
Outdegree of C = 2
Outdegree of D = 0
Outdegree of E = 2
Outdegree of F = 1

8. **Isolated Vertex:** If any vertex does not belong to any edge then it is called isolat vertex.

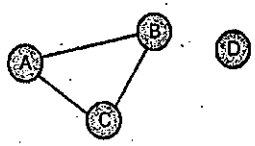


Fig. 3.10: Isolated Node D in the Graph

9. **Source Vertex:** A vertex with in-degree zero is called a source vertex, i.e., vert has only outgoing edges and no incoming edges. For example, in Fig. 3.11, 'C' source vertex.
10. **Sink Vertex:** A vertex with out-degree zero is called a sink vertex i.e. vertex ha only incoming edge and no outgoing edge. For example, in Fig. 3.11, 'B' is sin node.
11. **Acyclic Graph:** A graph without cycle is called acyclic graph. [Oct. 18]

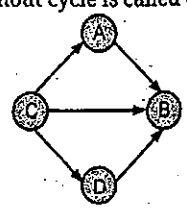
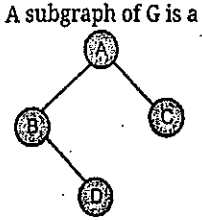


Fig. 3.11: Acyclic Graph

12. **Subgraph:** A subgraph of G is a graph G' such that V(G') ⊆ V(G) and E(G') ⊆ E(G).



(a) Graph G

(b) Subgraph of G

Fig. 3.12

13. **Weighted Graph:** A weighted graph is a graph in which every edge is assigned a weight. In Fig. 3.13 weight in a graph denote the distance between the two vertices connected by the corresponding edge. The weight of an edge is also called its cost. In case of a weighted graph, an edge is a 3-tuple (U, V, W) where U and V are vertices and W is weight of an edge (U, V).

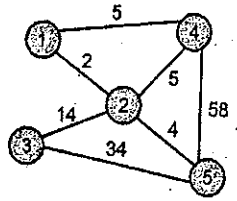


Fig. 3.13: Weighted Graph

14. **Pendant Vertex:** When in-degree of vertex is one and out-degree is zero then such a vertex is called pendant vertex.

15. **Spanning Tree:** A spanning tree of a graph G = (V, E) is a subgraph of G having all vertices of G and containing only those edges that are necessary to join all the vertices in the graph. A spanning tree does not contain cycle in it. Fig. 3.14 shows spanning Trees for graph in Fig. 3.6.

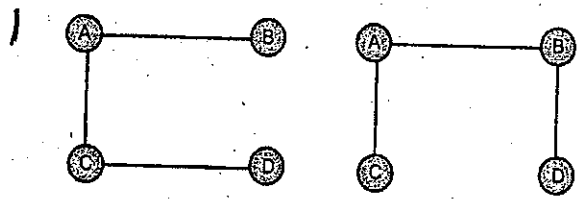


Fig. 3.14: Spanning Trees

16. **Sling or Loop:** An edge of a graph, which joins a node to itself, is called a sling or loop. Fig. 3.15 shows an example of loop.

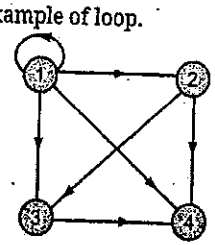


Fig. 3.15: Loop

3.2 REPRESENTATION OF GRAPH

- Representation of graph is a process to store the graph data into the computer memory. Graph is represented by the following three ways:
  1. Sequential representation using Arrays, (by means of Adjacency Matrix).
  2. Linked representation using Linked List, (by means of Adjacency List).
  3. Inverse adjacency list.
  4. Adjacency multi-list representation.

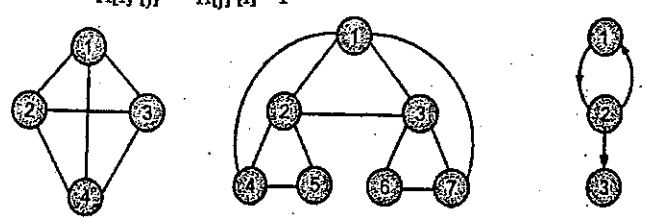
3.2.1 Sequential Representation of Graph [April 16, 17, 18, 19 Oct. 17, 18]

- Graphs can be represented through matrix (array) in computer system's memory. This is sequential in nature. This type of representation is called sequential representation of graphs.
- The graph when represented using sequential representation using matrix, is commonly known as Adjacency Matrix.
- Let G = (V, E) be a graph with n vertices, where n>=1. An adjacency matrix of G is a 2-dimentional n × n array, say A, with the following property:

  A[i][j] = 1 if the edge (v_i, v_j) is in E(G)
  = 0 if there is no such edge in G

- If the graph is undirected then,

  A[i] [j] = A[j] [i] = 1



G_1 G_2 G_3

Fig. 3.16: Undirected Graph G_1, G_2 and Directed Graph G_3

- The graphs G_1, G_2 and G_3 of Fig. 3.16 are represented using adjacency matrix in Fig. 3.17.

	1	2	3	4
1	0	1	1	1
2	1	0	1	1
3	1	1	0	1
4	1	1	1	0

G1

	1	2	3
1	0	1	0
2	1	0	1
3	0	0	0

G3

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

G_2

Fig. 3.17 (a): Adjacency Matrix for G_1, G_2 and G_3 of Fig. 3.16

- **Adjacency Matrix Representation of a Weighted Graph:**

For weighted graph, the matrix A is represented as,

A[i] [j] = weight of the edge (i, j)
= 0 otherwise 1

Here, weight is labelled associated with edge.

Example, following is the weighted graph and its associated adjacency matrix.

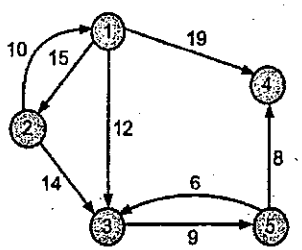
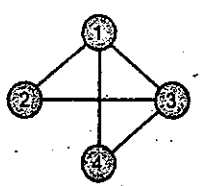


Fig. 3.17 (b)

	1	2	3	4	5
1	0	15	12	19	0
2	10	0	14	0	0
3	0	0	0	0	9
4	0	0	0	0	0
5	0	0	6	8	0

Example 1: Consider the following undirected graph and provide adjacency matrix. The graph has 4 vertices and it is undirected graph. Write 1 to Number of vertices i.e. 1 to 4 as row and column headers to represent it in adjacency matrix. If edge exists between any two nodes (row, column headers indicate nodes) write it as 1 otherwise 0 in the matrix.



(a) Undirected Graph

	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0

(b) Adjacency Matrix

Fig. 3.18

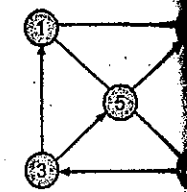


Fig. 3.19

Example 2: Consider the following directed graph and provide adjacency matrix. The graph has 5 vertices and it is directed graph. Write 1 to Number of vertices i.e. 1 to 5 as row and column headers to represent it in adjacency matrix. If edge exists between any two nodes (row, column headers indicate nodes) write it as 1 at (i, j) and (j, i) position in matrix otherwise 0 in the matrix.

The representation of the above graph using adjacency matrix is given below:

Adj-matrix =

	1	2	3	4	5
1	0	1	0	0	0
2	0	0	1	0	0
3	0	1	0	1	0
4	1	0	0	0	1
5	0	1	1	0	0

Program 3.1: Program to represent graph as adjacency matrix.

```
#include <stdio.h>
#define MAX 10
void degree(int adj[][MAX],int x,int n)
{
   int i,incount=0, outcount =0;
   for(i=0;i<n;i++)
   {
      if( adj[x][i] ==1)
         outcount++;
      if( adj[i][x] ==1)
         incount++;
   }
   printf("The indegree of the node %d is %d\n",x,incount++);
   printf("The outdegree of the node %d is %d\n",x,outcount++);
}
int main()
{
   int adj[MAX][MAX],n,i,j;
   setbuf(stdout, NULL);
   printf("Enter the total number of nodes in graph");
   scanf("%d",&n);
   for(i=0;i<n;i++)
```

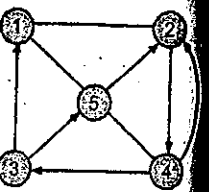


Fig. 3.19

+);
t++);


```
        for(j=0;j<n;j++)
        {
            printf("Enter Edge from %d to %d,(1: Edge 0: No edge) \n",i,j);
            scanf("%d",&adj[i][j]);
        }
    for(i=0;i<n;i++)
    {
        degree(adj,i,n);
    }
    return 0;
}
```

Output:

```
Enter the total number of nodes in graph4
Enter Edge from 0 to 0,(1: Edge 0: No edge)
0
Enter Edge from 0 to 1,(1: Edge 0: No edge)
1
Enter Edge from 0 to 2,(1: Edge 0: No edge)
0
Enter Edge from 0 to 3,(1: Edge 0: No edge)
1
Enter Edge from 1 to 0,(1: Edge 0: No edge)
1
Enter Edge from 1 to 1,(1: Edge 0: No edge)
0
Enter Edge from 1 to 2,(1: Edge 0: No edge)
1
Enter Edge from 1 to 3,(1: Edge 0: No edge)
1
Enter Edge from 2 to 0,(1: Edge 0: No edge)
0
Enter Edge from 2 to 1,(1: Edge 0: No edge)
1
Enter Edge from 2 to 2,(1: Edge 0: No edge)
0
Enter Edge from 2 to 3,(1: Edge 0: No edge)
1
Enter Edge from 3 to 0,(1: Edge 0: No edge)
1
Enter Edge from 3 to 1,(1: Edge 0: No edge)
1
Enter Edge from 3 to 2,(1: Edge 0: No edge)
1
Enter Edge from 3 to 3,(1: Edge 0: No edge)
0
```



```
The indegree of the node 0 is 2
The outdegree of the node 0 is 2
The indegree of the node 1 is 3
The outdegree of the node 1 is 3
The indegree of the node 2 is 2
The outdegree of the node 2 is 2
The indegree of the node 3 is 3
The outdegree of the node 3 is 3
```

Advantages of Array representation of Graph:

1. Simple and easy to understand.
2. Graphs can be constructed at run-time.
3. Efficient for dense (lots of edges) graphs.
4. Simple and easy to program.
5. Adapts easily to different kinds of graphs.

Disadvantages of Array representation of Graph:

1. Adjacency matrix consumes huge amount of memory for storing big or large graphs.
2. Adjacency matrix requires huge efforts for adding/removing a vertex.
3. The matrix representation of graph does not keep track of the information related to the nodes.
4. Requires that graph access be a command rather than a computation.

3.2.2 Linked Representation of Graphs

[April 16, 17, 18, 19 Oct. 17, 18]

- We use the adjacency list for the linked representation of the graph. In adjacency lists representation the n rows of the adjacency matrix are represented as n linked lists.
- The adjacency list representation maintains each node of the graph and a link to the nodes that are adjacent to this node. When we traverse all the adjacent nodes, we set the next pointer to null at the end of the list.
- **Example:** Consider the graph G. The Fig. 3.20 shows the graph G and linked representation of G in memory. The linked representation will contain two lists:
  1. **A node vertex list:** To keep the track of all the N nodes of the graph.
  2. **An edge list:** To keep the information of adjacent vertices of each and every vertex of a graph.
- Head node is used to represent each list, i.e. we can represent G by an array Head, where Head [i] is a pointer to the adjacency list of vertex i.

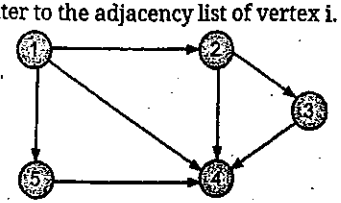


Fig. 3.20: Directed graph

Vertex and its Adjacent Vertices	
Vertex	Adjacent Vertices
1	2, 4, 5
2	3, 4
3	4
4	–
5	4

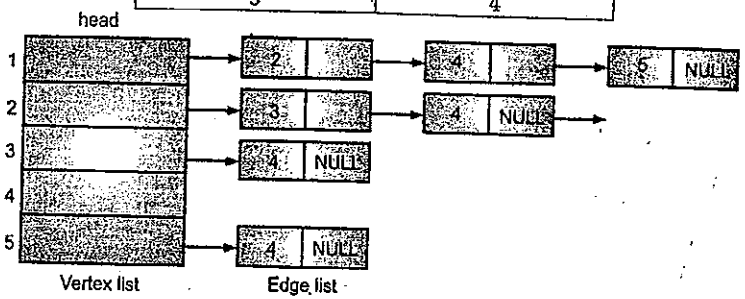
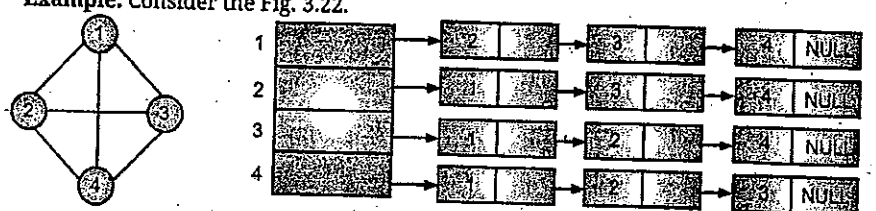


Fig. 3.21: Adjacency List Representation (Directed Graph)

Adjacency List Representation for Undirected Graph:

- In this representation, n rows of the adjacency matrix are represented as n linked lists. The nodes in list i represent the vertices that are adjacent to vertex i. head[i] is used to represent i^{th} vertex adjacency list.
- **Example:** Consider the Fig. 3.22.



Undirected graph G_1

Fig.3.22: Adjacency List Representation of G_1 (Undirected Graph)

- The structure for adjacency list representation can be defined in C as follows:

```
#define MAX_V 20
struct node
{
    int vertex;
    struct node * next;
}
struct node head[MAX_V];
```

- In this representation every edge (v_i, v_j) of an undirected graph is represented twice, once in the list of v_i and in the list of v_j.
- For directed graph time required is O(n) to determine whether, there is an edge from vertex i to vertex j and for undirected graph time is O(n + e).

Example 1: Consider the undirected graph in Fig. 3.23 and provide adjacency list.

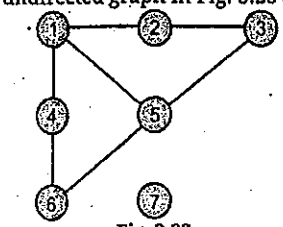


Fig. 3.23

Solution: The adjacency list is given in Fig. 3.24

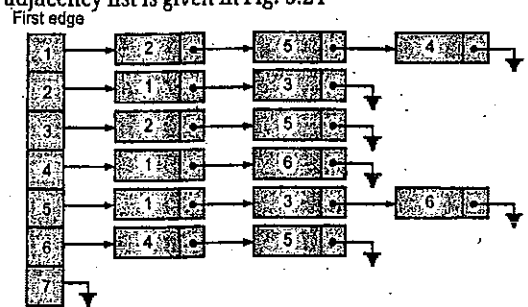


Fig. 3.24

Example 2: Consider the weighted undirected graph in Fig. 3.25 and provid adjacency list.

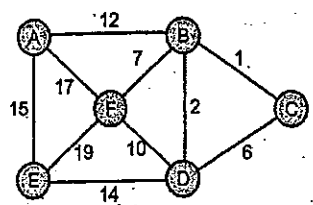


Fig. 3.25

Solution: An adjacency list for this graph is:

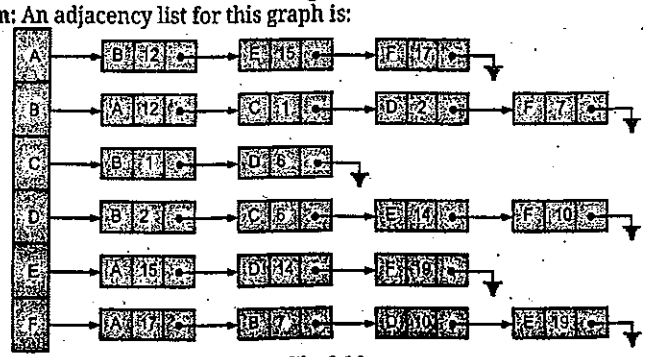


Fig. 3.26

Advantages of Linked List representation n of Graph:

1. Less storage for sparse (few edges) graphs.
2. Easy to store additional information in the data structure like vertex degree, edge weight etc.
3. Better memory space usage.
4. Better graph traversal times.
5. Generally better for most algorithms.

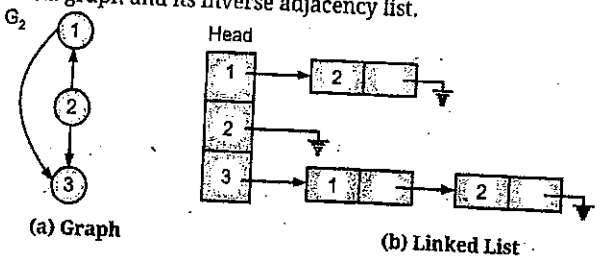
Disadvantages of Linked List representation of Graph:

1. Generally, takes some pre-processing to create Adjacency list.
2. Algorithms involving edge creation, deletion and querying edge between two vertices are better way with matrix representation than the list representation.
3. Adding/removing an edge to/from adjacent list is not so easy as for adjacency matrix.

3.2.3 Inverse Adjacency List

[Oct. 17]

- Inverse adjacency lists are a set of lists that contain one list for vertex. Each list contains a node per vertex adjacent to the vertex it represents.
- Fig. 3.27 shows a graph and its inverse adjacency list.



(a) Graph (b) Linked List

Fig. 3.27

3.2.4 Adjacency Multi-Lists

- Adjacency multi-list is an edge, rather than vertex based graph representation.
- With adjacency lists, each edge in an undirected graph appears twice in the list. Also, there is an obvious asymmetry for digraphs - it is easy to find the vertices a given vertex is adjacent to (simply follow its adjacency list), but hard to find the vertices adjacent to a given vertex (we must scan the adjacency lists of all vertices). These can be rectified by a structure called an adjacency multi-list.
- In adjacency multi-list nodes may be shared among several list (an edge is shared by two different paths).
- Typically, the following structure is used to represent an edge.

Visited	Vertex at tail	Vertex at head	Pointer to next edge containing vertex at tail	Pointer to next edge containing vertex at head

Example: Consider following undirected graph G = (V, E)

where, V={0,1,2,3}, E={(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)}

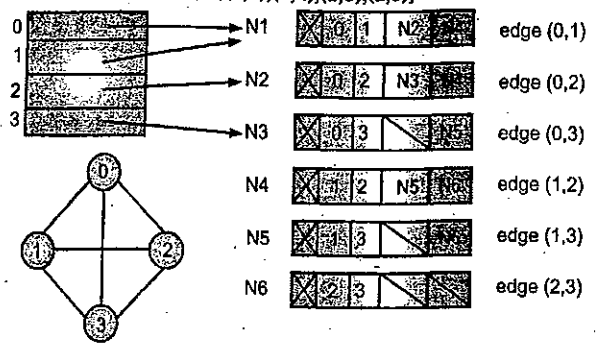


Fig. 3.28: Adjacency Multi-List Representation

3.3 GRAPH TRAVERSAL

- Graph traversal means visiting every vertex and edge exactly once in a well-defined order.
- In many situations, we need to visit all the vertices and edges in a systematic fashion. The graph traversal is used to decide the order of vertices to be visited in the search process.
- A graph traversal finds the edges to be visited without creating loops that means using graph traversal we visit all vertices of graph without getting into looping path.
- Traversal means visiting each element of the data structure representation, in graph it refers to visiting each vertex of the graph. Traversing a graph means visiting all the vertices in a graph exactly one.
- The two techniques are used for traversals:
  1. Depth First Search (DFS), and
  2. Breadth First Search (BFS).

3.3.1 Depth First Search (DFS)

[April 16, 17, 18, 19 Oct. 17, 18]

- Depth First Search (DFS) is used to perform traversal of a graph.
- In this method, all the vertices are stored in a Stack and each vertex of the graph is visited or explored once.
- The newest vertex (added last) in the Stack is explored first. To traverse the graph, Adjacent List of a graph is created first.

- We use the following steps to implement DFS traversal:

  Step 1 : Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.

  Step 2 : Visit any one of the adjacent vertex of the vertex which is at top of the stack which is not visited and push it on to the stack.

  Step 3 : Repeat step 2 until there are no more vertices to be visited which are adjacent to the vertex which is on top of the stack.

  Step 4 : When there are no more vertices to be visited then use back tracking and pop one vertex from the stack. (Back tracking is coming back to the vertex from which we came to current vertex)

  Step 5 : Repeat steps 2, 3 and 4 until stack becomes empty.

  Step 6 : When stack becomes empty, then stop. DFS traversal will be sequence of vertices in which they are visited using above steps.

- The algorithm for DFS can be outlined as follows:

1. for V = 1 to n do (Recursive)
   visited[V] = 0 {unvisited}
2. i = 1 {start at vertex 1)
3. DFS (i)
   begin
   visited[i] = 1
   display vertex i
   for each vertex j adjacent to i do
   if (visited[j] = 0)
   then DFS(j)
   end.

- Non-recursive DFS can be implemented by using stack for pushing all unvisited vertices adjacent to the one being visited and popping the stack to find the next unvisited vertex.

Algorithm for DFS (Non-recursive) using Stack:

1. Push start vertex onto STACK
2. While (not empty (STACK)) do
   begin
   v = POP (STACK)
   if (not visited (v))

   begin
   visited[v] = 1
   display vertex i
   push anyone adjacent vertex x of v onto STACK which is not visted
   end
   end
3. Stop.

Pseudo 'C' Code for Recursive DFS:

```
int visited[MAX]={0};
DFS (int i)
{    int k;
       visited[i] = 1;
       printf(" %d",i);
        for(k=0;k<n;k++)
          if (A[i,k]==1  && visited[k]==0)
               DFS (k);
}
```

- Let us consider graph 3.29 drawn below:

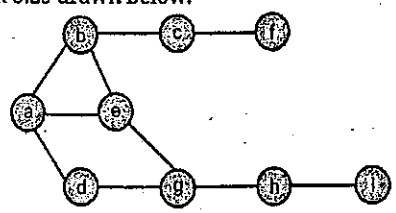
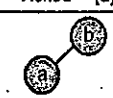


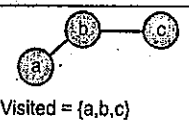
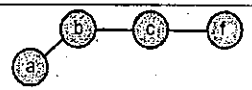
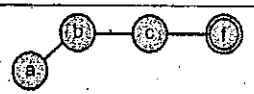
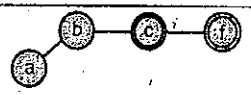
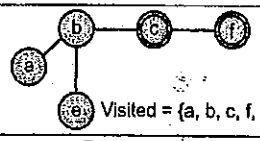
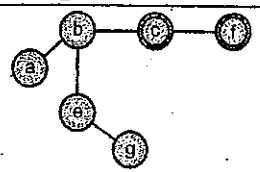
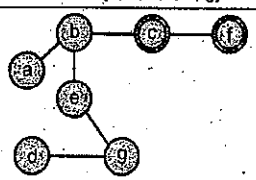
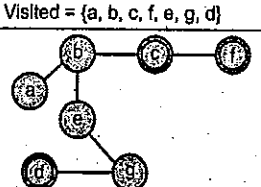
Fig. 3.29: Graph

- Let us traverse the graph using DFS algorithm which uses stack. Let 'a' be a start vertex. Initially stack is empty.

Note: Doubled circle indicates backtracking point.

Sr. No.	Steps	Stack	Spanning Tree
1.	Select vertex 'a' as starting point (visit 'a'). Push 'a' onto the stack		 Visited = {a}
2.	Visit any adjacent vertex of 'a' which is not visited ('b'). Push newly visited vertex 'b' onto the stack.		 Visited = {a,b}

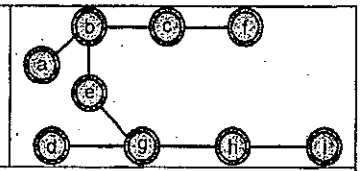
contd. ...

3.	Visit any adjacent vertex of 'b' which is not visited ('c'). Push 'c' onto the stack.		 Visited = {a,b,c}
4.	Visit any adjacent vertex of 'c' which is not visited ('f'). Push 'f' onto the stack.		 Visited ={a,b,c,f }
5.	There is no new vertex to be visited from 'f'. So backtrack. Pop 'f' from the stack.		
6.	There is no new vertex to be visited form 'c'. So backtrack. Pop 'c' from the stack.		
7.	Visit any adjacent vertex of 'b' which is not visited ("e"). Push 'e' onto the stack.		 Visited = {a, b, c, f, e)
8.	Visit any adjacent vertex of 'e' which is not visited ('g'). Push 'g' onto the stack.		 Visited = {a, b, c, f, e, g}
9.	Visit any adjacent vertex of 'g' which is not visited ('d'). Push 'd' onto the stack.		 Visited = {a, b, c, f, e, g, d}
10.	There is no new vertex to be visited from 'd'. So backtrack. Pop 'd' from the stack.		

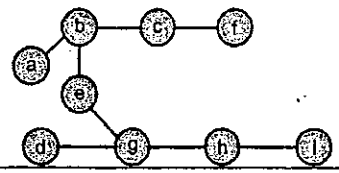
contd. ...

11.	Visit any adjacent vertex of 'g' which is not visited ('h'). Push 'h' onto the stack.		Visited = {a, b, c, f, e, g, d, h}
12.	Visit any adjacent vertex of 'h' which is not visited ('i'). Push 'i' onto the stack.		Visited = {a, b, c, f, e, g, d, h, i}
13.	There is no new vertex to be visited from 'i'. So backtrack pop 'i' from the stack.		
14.	There is no new vertex to be visited from 'h'. So backtrack. Pop 'h' from the stack.		
15.	There is no new vertex to be visited from 'g'. So backtrack. Pop 'g' from the stack.		
16.	There is no new vertex to be visited from 'e'. So backtrack. Pop 'e' from the stack.		
17.	There is no new vertex to be visited from 'b'. So backtrack. Pop 'b' from the stack.		

contd. ...

18.	There is no new vertex to be visited from 'a'. So backtrack. Pop 'a' from the stack.		

Stack becomes Empty. So Stop DFS traversal, final result of DFS traversal is following spanning tree.



3.3.2 Breadth First Search (BFS)

[April 16, 17, 18, 19 Oct. 17, 18]

- Another systematic way of visiting the vertices is Breadth-First Search (BFS). Starting at vertex v and mark it as visited, the BFS differs from DFS, in that all unvisited vertices adjacent to i are visited next.
- Then unvisited vertices adjacent to these vertices are visited and so on until the all vertices has been in visited. The approach is called 'breadth first' because from vertex i that we visit, we search as broadly as possible by next visiting all the vertices adjacent to i.
- In BFS method, all the vertices are stored in a Queue and each vertex of the graph is visited or explored once.
- The oldest vertex (added first) in the Queue is explored first. To traverse the graph using breadth first search, Adjacent List of a graph is created first.
- For example, the BFS of graph of Fig. 3.30 results in visiting the nodes in the following order: 1, 2, 3, 4, 5, 6, 7, 8.

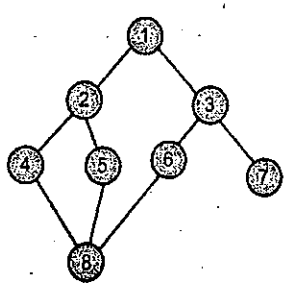


Fig. 3.30

- This search algorithm uses a queue to store the adjacent vertices of each vertex of the graph as and when it is visited.

- These vertices are then taken out from queue in a sequence (FIFO) and their adj vertices are visited and so on until all the vertices have been visited. The algor terminates when the queue is empty.
- The algorithm for BFS is given below. The algorithm initializes the Boolean a visited[·] to 0 (false) i.e. marks each vertex as unvisited.

```
for i = 1 to n do
    visited[i] = 0.
```

Algorithm:

```
BFS (i)
begin
    visited[i] = 1
    add (Queue, i)
        while not empty (Queue) do
            begin
                i = delete (Queue)
                for all vertices j adjacent to i do
                    begin
                    if (visited[j] = 0)
                        add (Queue, j)
                        visited[j] = 1
                    end
            end
        end
```

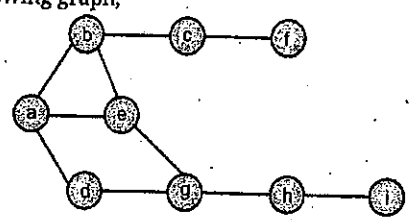
- Here, while loop is executed n time as n is the number of vertices and each vertex is inserted in a queue once. If adjacency list representation is used, then adjacent nodes are computed in for loop.

Pseudo 'C' code for recursive BFS:

```
BFS (int i)
    {
        int k,visited[MAX];
        struct queue Q;
        for(k=0;k<n;k++)
        visited[k] = 0;
        visited[i] = 1;
        printf(" %d",i);
        insert(Q,i);
```

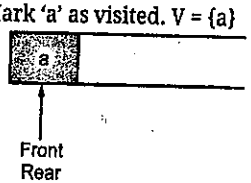
```
        while (!Empty (Q))
        {
          j = delete(Q);
          for(k=0;k<n;k++)
          {
             if (A[j,k]&&!visited[k])
             {
               insert(Q,k);
               visited[k] = 1;
             printf(" %d",k);
             }
          }
        }
    }
```

- Let us consider following graph,



- Let us traverse the graph using non-recursive algorithm which uses queue. Let 'a' be a start vertex. Initially queue is empty. Initial set of visited vertices, V = ϕ.

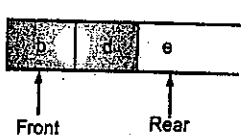
(i) Add 'a' to the queue. Mark 'a' as visited. V = {a}



(ii) As queue is not empty,

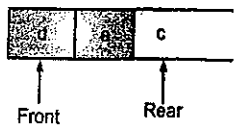
vertex = delete( ) = 'a'.

Add all unvisited adjacent vertices of 'a' to the queue. Also mark them visited.
V = {a, b, d, e}



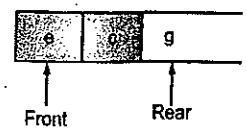
(iii) As queue is not empty,

Vertex = delete( ) = 'b'. Insert all adjacent, unvisited vertices of 'b' to the queue.
V = {a, b, d, e, c}



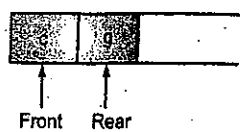
(iv) As queue is not empty,

Vertex = delete( ) = 'd'. Now insert all adjacent, unvisited vertices of 'd' to the queue and mark them as visited
V = {a, b, d, e, c, g}



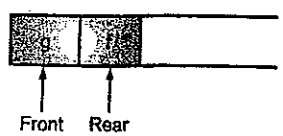
(v) As queue is not empty,

Vertex = delete( ) = 'e'. There are no unvisited adjacent vertices of 'e'. The queue is as:
V = {a, b, d, e, c, g}.



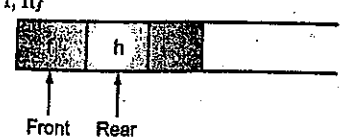
(vi) As queue is not empty, vertex = delete( ) = 'c'.

Insert all adjacent, unvisited vertices of 'c' to the queue and mark them visited.
V = {a, b, d, e, c, g, f}



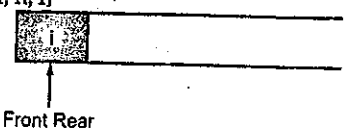
(vii) As queue is not empty, vertex = delete( ) = 'g'.

Insert all unvisited adjacent vertices of 'g' to the queue and mark them visited.
V = {a, b, d, e, c, g, f, h}



(viii) As queue is not empty, vertex = delete( ) = 'f'.
There are no unvisited adjacent vertices of 'f'. The queue is as:
V = {a, b, d, e, c, g, f, h}

(ix) As queue is not empty, vertex = delete( ) = 'h'.
Insert its unvisited adjacent vertices to queue and Mark them.
V = {a, b, d, e, c, g, f, h, i}



(x) As queue is not empty,
Vertex = delete( ) = 'i', No adjacent vertices of i are unvisited.

(xi) As queue is empty, stop.
The sequence in which vertices are visited by BFS is as:
a, b, d, e, c, g, f, h, i.

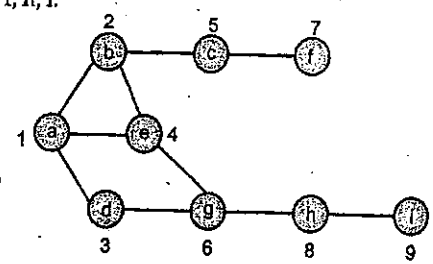


Fig. 3.31

Program 3.2: To create a graph and represent using adjacency matrix and adjacency list and traverse in BFS order.

```
#include<stdio.h>
#include<stdlib.h>
struct q
{
   int data[20];
   int front, rear;
} q1;
struct node
{
   int vertex;
   struct node * next;
} * v[10];
```



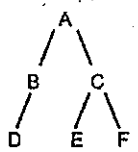
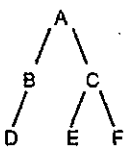
```
void add (int n)
{
   q1.rear++;
   q1.data[q1.rear]=n;
}
int del()
{
   q1.front++;
   return q1.data[q1.front];
}
void initq()
{
   q1.front = q1.rear = - 1;
}
int emptyq()
{
   return (q1.rear == q1.front);
}
void create(int m[10][10], int n)
{
   int i, j;
   char ans;
   for (i=0; i<n; i++)
   for (j=0; j<n; j++)
   {
      m[i][i] = 0;
      if(i != j)
      {
         printf("\n \t Is there an edge between %d and %d:", i+1, j+1);
         scanf("%d", &m[i][j]);
      }
   }
} /* end of create */
void disp(int m[10][10], int n)
{
   int i, j;
   printf("\n \t the adjacency matrix is: \n");
```

```
        for(i=0; i<n; i++)
        {
           for (j=0; j<n; j++)
           printf("%5d", m[i][j]);
           printf("\n");
        }
  } /* end of display */
  void create1 (int m[10][10], int n)
  {
     int i, j;
     struct node *temp, *newnode;
     for(i=0; i<n; i++)
     {
        v[i] = NULL;
        for(j=0; j<n; j++)
        {
           if(m[i][j] == 1)
           {
                 newnode=(struct node *) malloc(sizeof(struct node));
                 newnode -> next = NULL;
                 newnode -> vertex = j+1;
                 if(v[i]==NULL)
                 v[i] = temp = newnode;
                 else
                 {
                       temp -> next = newnode;
                       temp = newnode;
                 }
           }
        }
     }
  }
  void displist(int n)
  {
     struct node *temp;
     int i;
     for (i=0; i<n; i++)
     {
        printf("\nv%d | ", i+1);
          temp = v[i];
```



```
           while (temp)
           {
                 printf("v%d -> ", temp -> vertex);
                 temp = temp -> next;
           }
           printf("null");
     }
  }
  void bfs(int m[10][10], int n)  //create adjacency list
  {
     int i, j, v, w;
     int visited[20];
     initq();
     for(i=0; i<n; i++)
     visited[i] = 0;
     printf("\n \t The BFS traversal is: \n");
     v=0;
     visited[v] = 1;
     add(v);
     while(! emptyq())
     {
        v = del();
        printf("\n v%d ", v + 1);
           printf("\n");
           for(w = 0; w < n; w++)
           if((m[v][w] ==1) && (visited[w] == 0))
           {
                 add(w);
                 visited[w] = 1;
           }
     }
  }
  /* main program */
  void main()
  {
     int m[10][10], n;
     printf("\n \t enter no. of vertices");
        scanf("%d", &n);
        create(m,n);
        disp(m,n);
        create1(m,n);
        displist(n);
        bfs(m,n);
  }
```

Differentiation between **DFS and BFS:**

Sr. No.	DFS	BSI
1.	DFS stands for Depth First Search.	BFS stands for Breadth First Search.
2.	DFS uses Stack implementation i.e. LIFO.	BFS uses Queue implementation i.e. FIFO.
3.	DFS is faster.	Slower than DFS.
4.	DFS requires less memory.	BFS uses a large amount of memory.
5.	DFS is easy to implement in a procedural language.	DFS is complex or hard to implement in a procedural language.
6.	The aim of DFS algorithm is to traverse the graph in such a way that it tries to go far from the root node.	The aim of BFS algorithm is to traverse the graph as close as possible to the root node.
7.	Used for topological sorting.	Used for finding the shortest path between two nodes.
8.	**Example:**  A, B, D, C, E, F	**Example:**  A, B, C, D, E, F

3.4 APPLICATIONS OF GRAPH [April 18, Oct. 18]

- In this we study various operations on graph like topological sorting, minimal spanning tree, and so on.

3.4.1 Topological Sort [April 16, 19]

- A topological sort has important applications for graphs. Topological sorting is possible if and only if the graph is a directed acyclic graph.
- Topological sorting of vertices of a directed acyclic graph is an ordering of the vertices $v_1, v_2, \ldots v_n$ in such a way, that if there is an edge directed towards vertex v_j from vertex v_i, then v_i comes before v_j.
- Topological ordering is not possible if the graph has a cycle. The Fig. 3.32 shows the topological sorting.

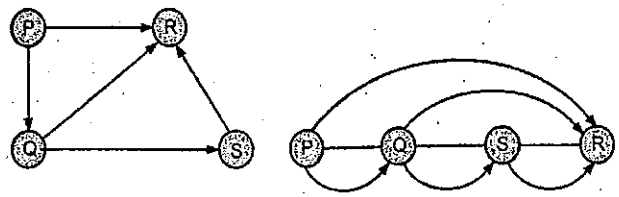


Fig. 3.32: A Graph and its Topological Ordering

- The topological sort of a directed acyclic graph is a linear ordering of the vertices, such that if there exists a path from vertex x to y, then x appears before y in the topological sort.
- Formally, for a directed acyclic graph G = (V, E), where V = (V_1, V_2, V_3, ..., V_n), if there exists a path from any V_i to V_j, the V_i appears before V_j in the topological sort.
- An acyclic directed graph and have more than one topological sorts. For example, two different topological sorts for the graph shown in Fig. 3.33 are (1, 4, 2, 3) and (1, 2, 4, 3).

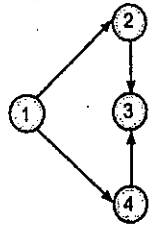


Fig. 3.33: Acyclic Directed Graph

- Clearly, if a directed graph contains a cycle, topological ordering of vertices is not possible. It is because for any two vertices V_i and V_j in the cycle, V_j precedes V_i as well as V_j precedes V_i. To exemplify this, consider a simple cyclic directed graph shown in Fig. 3.34.
- The topological sort for this graph is (1, 2, 3, 4) (assuming the vertex 1 as starting vertex). Now, since there exists a path from the vertex 4 to 1; then according to the definition of topological sort, the vertex 4 must appear before the vertex 1, which contradicts the topological sort generated for this graph. Hence, topological sort can exist only for the acyclic graph.

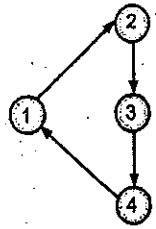


Fig. 3.34: Cyclic Directed Graph

- In an algorithm to find the topological sort of an acyclic directed graph, the indegree of the vertices is considered. Following are the steps that are repeated until the graph is empty.
  1. Select any vertex V_i with 0 indegree.
  2. Add vertex V_i to the topological sort (initially the topological sort is empty).
  3. Remove the vertex V_i along with its edges from the graph and decrease the indegree of each adjacent vertex of V_i by one.
- To illustrate this algorithm, consider an acyclic directed graph shown in Fig. 3.35.

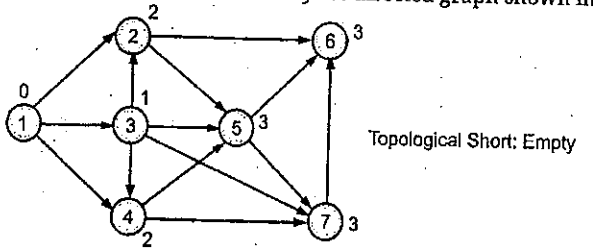
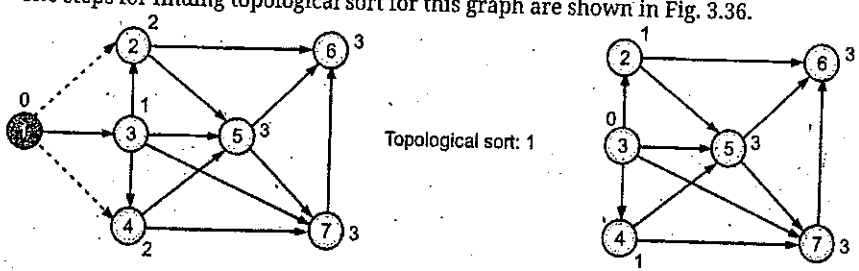
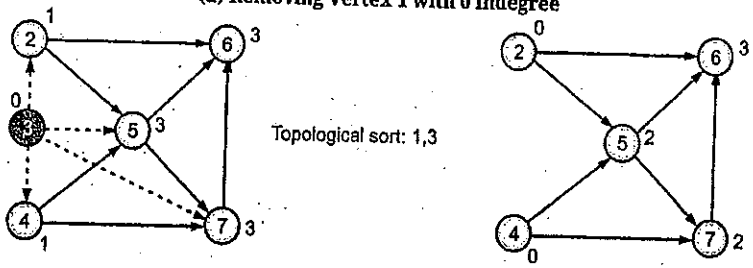


Fig. 3.35: Acyclic Directed Graph

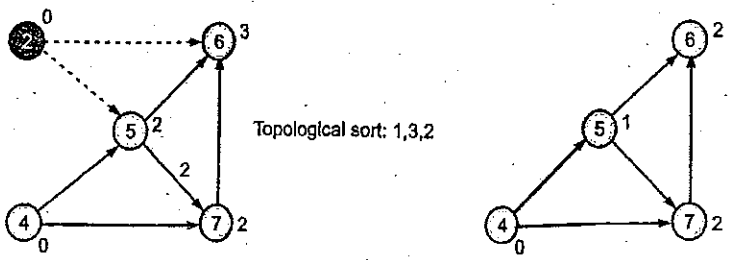
- The steps for finding topological sort for this graph are shown in Fig. 3.36.



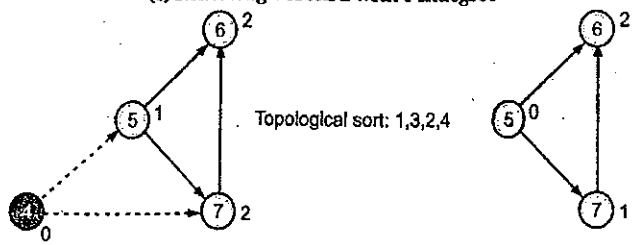
(a) Removing Vertex 1 with 0 Indegree



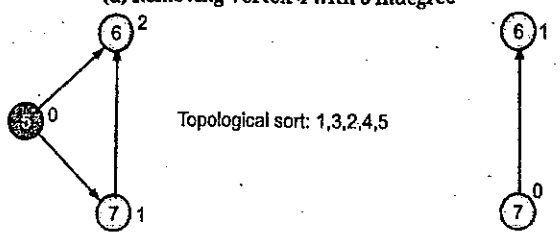
(b) Removing Vertex 3 with 0 Indegree



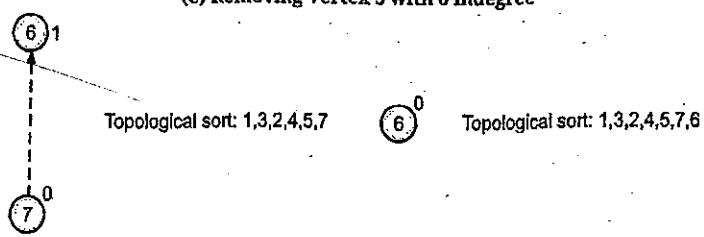
(c) Removing Vertex 2 with 0 Indegree



(d) Removing Vertex 4 with 0 Indegree



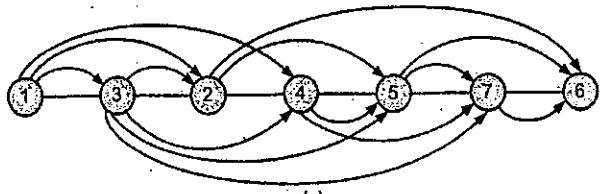
(e) Removing Vertex 5 with 0 Indegree



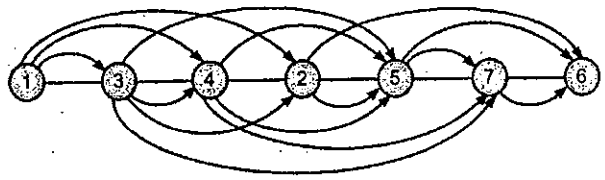
(f) Removing node 7 with 0 Indegree

Fig. 3.36: Steps for Finding Topological Sort

- Another possible topological sort for this graph is (1, 3, 4, 2, 5, 7, 6). Hence, it can be concluded that the topological sort for an acyclic graph is not unique.
- Topological ordering can be represented graphically. In this representation, edges are also included to justify the ordering of vertices (See Fig. 3.37).



(a)



(b)

Fig. 3.37: Graphical representation of Topological Sort

- Topological sort is useful for the proper scheduling of various subtasks to be executed for completing a particular task. In computer field, it is used for scheduling the instructions.
- For example, consider a task in which smaller number is to be subtracted from the larger one. The set of instructions for this task is as follows:

```
1. if A>B then goto Step 2, else goto Step 3
2. C = A-B, goto Step 4
3. C = B-A, goto Step 4
4. Print C
5. End
```

- The two possible scheduling orders to accomplish this task are (1, 2, 4, 5) and (1, 3, 4, 5). From this, it can be concluded that the instruction 2 cannot be executed unless the instruction 1 is executed before it. Moreover, these instructions are non-repetitive, hence acyclic in nature.

3.4.2 Use of Greedy Strategy in Minimal Spanning Trees

- There are many ways to construct a minimum spanning tree. One of the simplest methods is known as a greedy strategy.
- Minimum Spanning Tree (MST) algorithms are a classic example of the greedy method.
- Greedy strategy is used to solve many problems, such as finding the minimal spanning tree in a graph using Prim's /Kruskal's algorithm.

Spanning Tree:

- A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges.

- A spanning tree of a connected graph G is a tree that covers all the vertices and t edges required to connect those vertices in the graph.
- Formally, a tree T is called a spanning tree of a connected graph G if the following tw conditions hold:
  1. T contains all the vertices of G, and
  2. All the edges of T are subsets of edges of G.
- For a given graph G with n vertices, there can be many spanning trees and each tr will have n – 1 edges. For example, consider a graph as shown in Fig. 3.38.
- Since, this graph has 4 vertices, each spanning tree must have 4 – 1 = 3 edges. Some o the spanning trees for this graph are shown in Fig. 3.39.
- Observe that in spanning trees, there exists only one path between any two vertice and insertion of any other edge in the spanning tree results in a cycle.
- The spanning tree generated by using depth-first traversal is known as depth-firs spanning tree. Similarly, the spanning tree generated by using breadth-first traversa is known as breadth-first spanning tree.

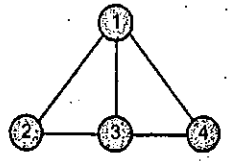


Fig. 3.38: A Graph G

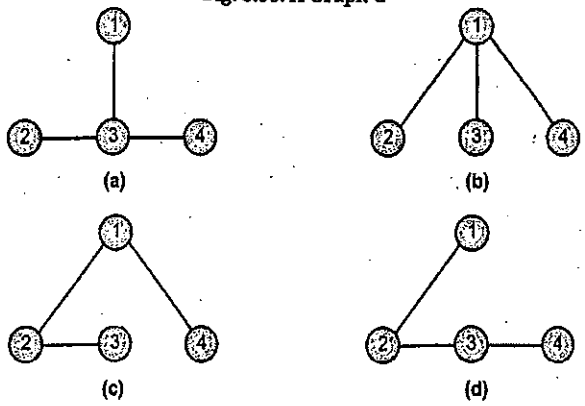
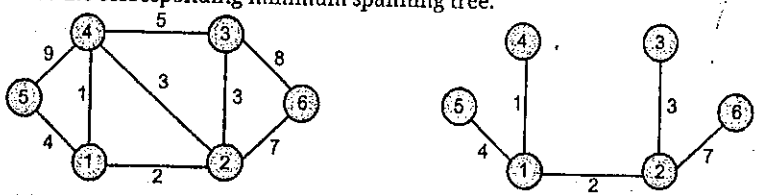


Fig. 3.39: Spanning Trees of Graph G

- For a connected weighted graph G, it is required to construct a spanning tree T such that the sum of weights of the edges in T must be minimum. Such a tree is called a minimum spanning tree.

- There are various approaches for constructing a minimum spanning tree out of which Kruskal's algorithm and Prim's algorithm are commonly used.
- If each edge of the graph is associated with a weight and there exists more than one spanning tree, we need to find the minimum spanning tree of the graph.
- A Minimum Spanning Tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.
- A minimum spanning tree in an undirected connected weighted graph is a spanning tree of minimum weight (among all spanning trees).
- In a minimum spanning tree of a graph, the maximum weight of an edge is the minimum possible from all possible spanning trees of that graph.
- In a minimum spanning tree of a graph, the maximum weight of an edge is the minimum possible from all possible spanning trees of that graph.
- In the left image you can see a weighted undirected graph, and in the right image you can see the corresponding minimum spanning tree.



(a) Weighted Undirected Graph (b) Minimum Spanning Tree

Fig. 3.40

Prim's Minimum Spanning Tree (MST) Algorithm:

- Prim's algorithm is a greedy strategy to find the minimum spanning tree. In this algorithm, to form a MST we can start from an arbitrary vertex.
- Prim's algorithm shares a similarity with the shortest path first algorithms.
- Prim's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.
- Example: Consider the following graph.

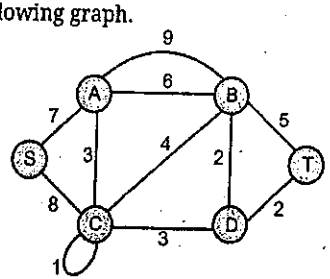


Fig. 3.41

Step 1: Remove all loops and parallel edges:

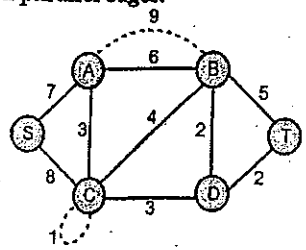


Fig. 3.42

- Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

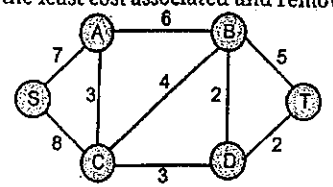


Fig. 3.43

Step 2: Choose any arbitrary node as root node:

- In this case, we choose S node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

Step 3: Check outgoing edges and select the one with less cost:

- After choosing the root node S, we see that S, A and S, C are two edges with weight 7 and 8, respectively. We choose the edge S, A as it is lesser than the other.

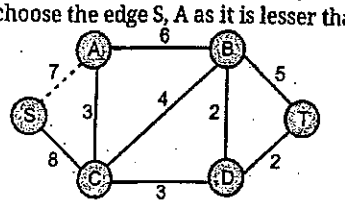


Fig. 3.44

- Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.

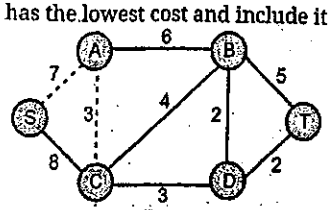


Fig. 3.45

- After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.

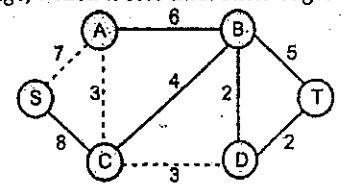


Fig. 3.46

- After adding node D to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.

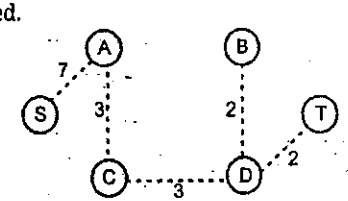


Fig. 3.47

- We may find that the output spanning tree of the same graph using two different algorithms is same.

Kruskal's Minimum Spanning Tree (MST) Algorithm:

- Kruskal's algorithm to find the minimum cost spanning tree uses the greedy strategy.
- Kruskal's algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.
- To understand Kruskal's algorithm let us consider the Fig. 3.48.

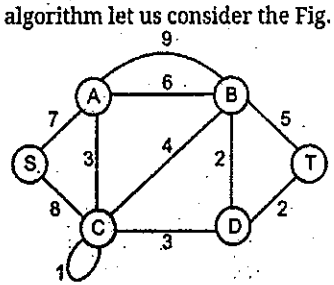


Fig. 3.48

Step 1: Remove all Loops and Parallel Edges:

- Remove all loops and parallel edges from the given graph.

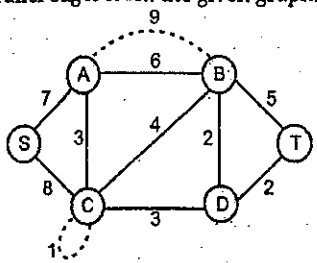


Fig. 3.49

- In case of parallel edges, keep the one which has the least cost associated and remove all others.

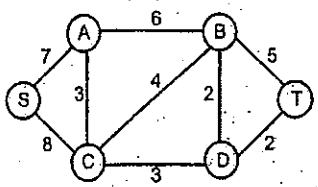


Fig. 3.50

Step 2: Arrange all edges in their increasing order of weight:

- The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

Step 3: Add the edge which has the least weightage:

- Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.

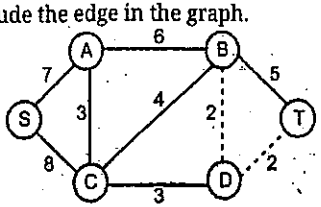


Fig. 3.51

- The least cost is 2 and edges involved are B, D and D, T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.

- Next cost is 3, and associated edges are A, C and C, D. We add them again:

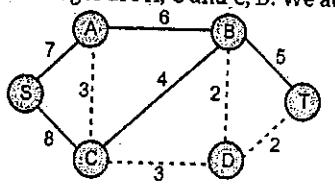


Fig. 3.52

- Next cost in the table is 4, and we observe that adding it will create a circuit in the graph.

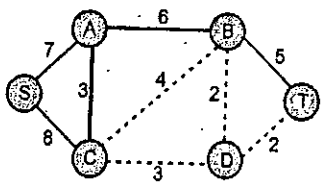


Fig. 3.53

- We ignore it. In the process we shall ignore/avoid all edges that create a circuit.

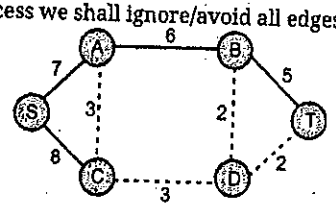


Fig. 3.54

- We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.

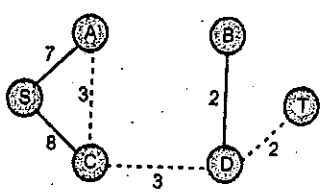


Fig. 3.55

- Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.

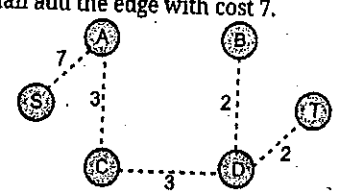


Fig. 3.56

- By adding edge S, A we have included all the nodes of the graph and we now have minimum cost spanning tree.
- **Example:** Consider the graph in Fig. 3.57.

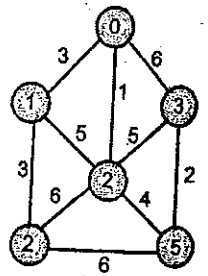
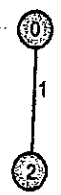


Fig. 3.57

- Procedure for finding Minimum Spanning Tree:

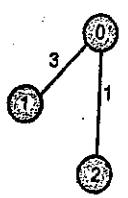
Step 1:

No. of Nodes	0	1	2	3	4	5
Distance	0	3	1	6	∞	∞
Distance From		0	0	0		



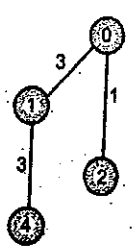
Step 2:

No. of Nodes	0	1	2	3	4	5
Distance	0	3	0	5	6	4
Distance From		0		2	2	2



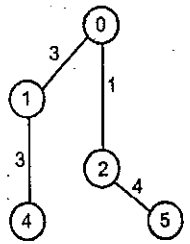
Step 3:

No. of Nodes	0	1	2	3	4	5
Distance	0	0	0	5	3	4
Distance From				2	1	2



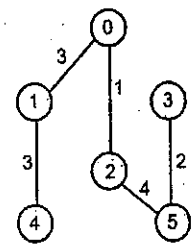
Step 4:

No. of Nodes	0	1	2	3	4	5
Distance	0	0	0	5	0	4
Distance From				2		2



Step 5:

No. of Nodes	0	1	2	3	4	5
Distance	0	0	0	3	0	0
Distance From				2		2



Minimum Cost = 1+2+3+3+4 = 13.

3.4.3 Single Source Shortest Path (Dijkstra's Algorithm)

- Dijkstra's algorithm (or Dijkstra's Shortest Path First algorithm, SPF algorithm)[2] is an algorithm for finding the shortest paths between nodes in a graph.
- Dijkstra's algorithm solves the single-source shortest-paths problem on a directed weighted graph G = (V, E), where all the edges are non-negative (i.e., w(u, v) ≥ 0 for each edge (u, v) ∈ E).
- Dijkstra's Algorithm can be applied to either a directed or an undirected graph to find the shortest path to each vertex from a single source.

Example: Let us consider vertex 1 and 9 as the start and destination vertex respectively. Initially, all the vertices except the start vertex are marked by ∞ and the start vertex is marked by 0.

Vertex	Initial	Step1 V_1	Step2 V_3	Step3 V_2	Step4 V_4	Step5 V_5	Step6 V_7	Step7 V_8	Step8 V_6
1	0	0	0	0	0	0	0	0	0
2	∞	5	4	4	4	4	4	4	4
3	∞	2	2	2	2	2	2	2	2
4	∞	∞	∞	7	7	7	7	7	7
5	∞	∞	∞	11	9	9	9	9	9
6	∞	∞	∞	∞	∞	17	17	16	16
7	∞	∞	11	11	11	11	11	11	11
8	∞	∞	∞	∞	∞	16	13	13	13
9	∞	∞	∞	∞	∞	∞	∞	∞	20

Hence, the minimum distance of vertex 9 from vertex 1 is 20. And the path is
1 → 3 → 7 → 8 → 6 → 9
This path is determined based on predecessor information.

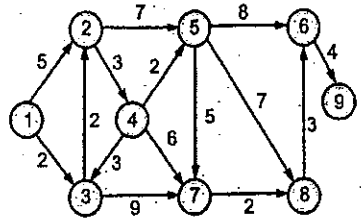


Fig. 3.58

Example: Consider the graph in Fig. 3.59.

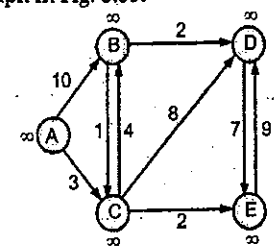
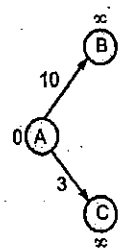


Fig. 3.59

Procedure for Dijkstra's Algorithm:

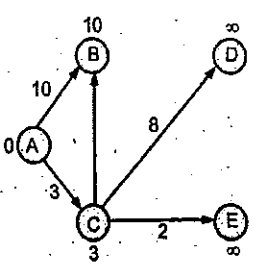
Step 1: Consider A as source Vertex.

No. of Nodes	A	B	C	D	E
Distance	0	10	3	∞	∞
Distance From		A	A		



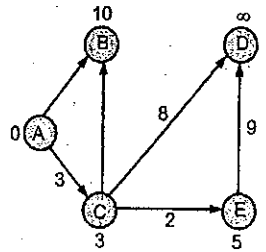
Step 2: Now consider vertex C.

No. of Nodes	A	B	C	D	E
Distance	0	7	3	11	5
Distance From		C	C	C	C



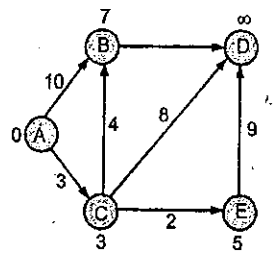
Step 3: Now consider vertex E.

No. of Nodes	A	B	C	D	E
Distance	0	7	3	11	0
Distance From		C	A	C	E



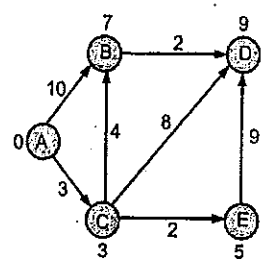
Step 4: Now consider vertex B.

No. of Nodes	A	B	C	D	E
Distance	0	7	3	9	5
Distance From		C	A	B	C



Step 5: Now consider vertex D.

No. of Nodes	A	B	C	D	E
Distance	0	7	3	9	11
Distance From	A	C	A	B	C



Therefore,

	A	B	C	D	E
	0	∞	∞	∞	∞
A	0	10	3	∞	∞
C		7	3	11	5
E				14	5
B				9	
D					16

3.4.4 Dynamic Programming Strategy

- The all pairs shortest paths algorithm of Floyd and Warshall uses a dynamic programming strategy.
- The shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.
- The problem of finding the shortest path between two intersections on a road map may be modeled as a special case of the shortest path problem in graphs, where the vertices correspond to intersections and the edges correspond to road segments, each weighted by the length of the segment.
- Fig. 3.60 shows shortest path (A, C, E, D, F) between vertices A and F in the weighted directed graph.

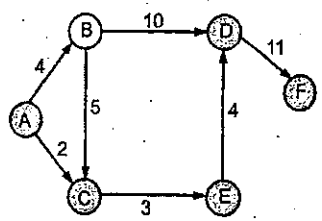
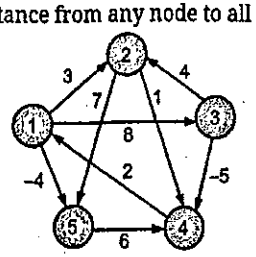


Fig. 3.60

- The all-pairs shortest path problem, in which we have to find shortest paths between every pair of vertices in the graph.
- The Floyd Warshall Algorithm is for solving the All Pairs Shortest Path problem. The all pair shortest path algorithm is also known as Floyd-Warshall algorithm is used to find all pair shortest path problem from a given weighted graph.
- The Floyd-Warshall algorithm is used for finding the shortest path between every pair of vertices in the graph.
- This algorithm works for both directed as well as undirected graphs. This algorithm is invented by Robert Floyd and Stephen Warshall hence it is often called as Floyed - Warshall algorithm.
- Floyd-Warshall algorithm is used to find all pair shortest path problem from a given weighted graph.
- As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any node to all other nodes in the graph.



$$\begin{bmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$

Fig. 3.61

- At first, the output matrix is the same as the given cost matrix of the graph. After that, the output matrix will be updated with all vertices k as the intermediate vertex.

Input and Output:

Input: The cost matrix of the graph.

0 3 6 ∞ ∞ ∞ ∞
3 0 2 1 ∞ ∞ ∞
6 2 0 1 4 2 ∞
∞ 1 1 0 2 ∞ 4
∞ ∞ 4 2 0 2 1
∞ ∞ 2 ∞ 2 0 1
∞ ∞ ∞ 4 1 1 0

Output: Matrix of all pair shortest path.

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

3.4.5 Use of Graphs in Social Networks

- A graph is made up of nodes; just like that a social media is a kind of a social network, where each person or organization represents a node.
- A graph is made up of nodes; just like that a social media is a kind of a social network, where each person or organization represents a node.
- In World Wide Web (WWW), web pages are considered to be the vertices. There is an edge from a page u to other page v if there is a link of page v on page u. This is an example of Directed graph.
- Graphs are awesome data structures that you use every day through Google Search, Google Maps, GPS, and social media.
- They are used to represent elements that share connections. The elements in the graph are called Nodes and the connections between them are called Edges.
- When we need to represent any form of relations in the society in the form of links, it can be termed as Social Network.
- Social graphs draw edges between you and the people, places and things you interact with online.
- Facebook's Graph API is perhaps the best example of application of graphs to real life problems. The Graph API is a revolution in large-scale data provision.

- On The Graph API, everything is a vertice or node. This are entities such as User Pages, Places, Groups, Comments, Photos, Photo Albums, Stories, Videos, Notes, Event and so forth. Anything that has properties that store data is a vertice.
- The Graph API uses this, collections of vertices and edges (essentially graph dat structures) to store its data.
- The Graph API has come into some problems because of it's ability to obtain unusuall rich info about user's friends.

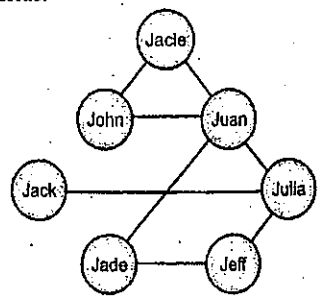
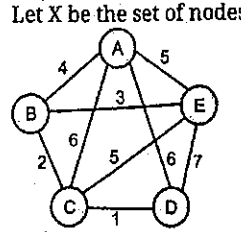


Fig. 3.62: A sample Graph (in this graph, individuals are represented with nodes (circles) and individuals who know each other are connected with edges (lines))

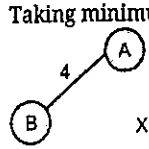
Example: Define spanning tree and minimum spanning tree. Find the minimum spanning tree of the graph shown in Fig. 3.62.

Using Prim's Algorithm:

Let X be the set of nodes explored, initially X = {A}.

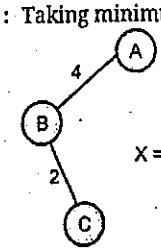


Step 1 : Taking minimum weight edge of all Adjacent edges of X = {A}.



X = {A, B}

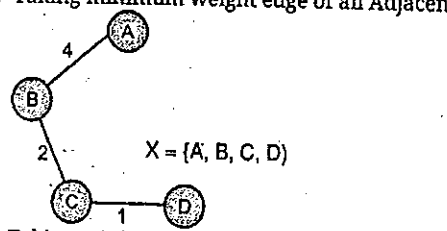
Step 2 : Taking minimum weight edge of all Adjacent edges of X = {A, B}.



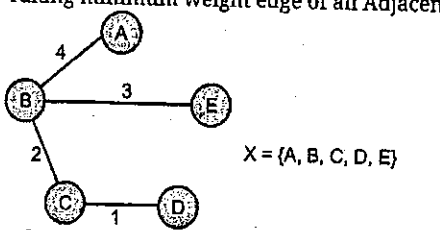
X = {A, B, C}

~~A-B|4~~
A-E|5
A-C|6
A-D|6
~~B-E|3~~
~~B-C|2~~
C-E|6
~~C-D|1~~
D-E|7

Step 3 : Taking minimum weight edge of all Adjacent edges of X = {A, B, C}.



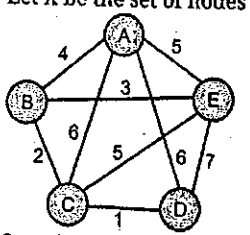
Step 4 : Taking minimum weight edge of all Adjacent edges of X = {A, B, C, D}.



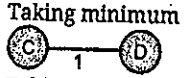
All nodes of graph are there with set X, so we obtained minimum spanning tree of cost: 4 + 2 + 1 + 3 = 10.

Using Kruskal's Algorithm:

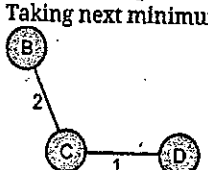
- Let X be the set of nodes explored, initially X = (A).



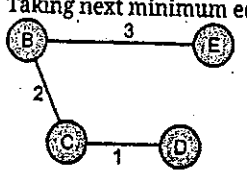
Step 1 : Taking minimum edge (C, D).



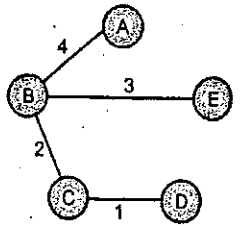
Step 2 : Taking next minimum edge (B, C).



Step 3 : Taking next minimum edge (B, E).



Step 4 : Taking next minimum edge (A, B).



Step 5 : Taking next minimum edge (A, E) it forms cycle so do not consider.

Step 6 : Taking next minimum edge (C, E) it forms cycle so do not consider.

Step 7 : Taking next minimum edge (A, D) it forms cycle so do not consider.

Step 8 : Taking next minimum edge (A, C) it forms cycle so do not consider.

Step 9 : Taking next minimum edge (E, D) it forms cycle so do not consider.

All edges of graph have been visited, so we obtained minimum spanning tree of cost: 4 + 2 + 1 + 3 = 10.

Program 3.3: Program that accepts the vertices and edges of a graph. Create adjacency list and display the adjacency list.

```
#include <stdio.h>
#include <stdlib.h>
#define new_node (struct node*)malloc(sizeof(struct node))
struct node
{
   int vertex;
   struct node *next;
};
void main()
{
   int choice;
   do
   {
      printf("\n A Program to represent a Graph by using an
                                                    Adjacency List \n ");
      printf("\n 1. Directed Graph ");
      printf("\n 2. Un-Directed Graph ");
      printf("\n 3. Exit ");
      printf("\n\n Select a proper choice : ");
      scanf("%d", &choice);
```

```
        switch(choice)
        {
           case 1 : dir_graph();
                    break;
           case 2 : undir_graph();
                    break;
           case 3 : exit(0);
        }
     }while(1);
}
int dir_graph()
{
   struct node *adj_list[10], *p;
   int n;
   int in_deg, out_deg, i, j;
   printf("\n How Many Vertices ? : ");
   scanf("%d", &n);
   for( i = 1 ; i <= n ; i++ )
        adj_list[i] = NULL;
   read_graph (adj_list, n);
   printf("\n Vertex \t In_Degree \t Out_Degree \t Total_Degree ");
   for (i = 1; i <= n ; i++ )
   {
      in_deg = out_deg = 0;
      p = adj_list[i];
      while( p != NULL )
      {
         out_deg++;
         p = p -> next;
      }
      for ( j = 1 ; j <= n ; j++ )
      {
         p = adj_list[j];
         while( p != NULL )
         {
            if ( p -> vertex == i )
               in_deg++;
            p = p -> next;
         }
      }
```



```
      printf("\n\n %5d\t\t\t%d\t\t%d\t\t%d\n\n", i, in_deg,
                                          out_deg, in_deg + out_deg);
   }
   return;
}
int undir_graph()
{
   struct node *adj_list[10], *p;
   int deg, i, j, n;
   printf("\n How Many Vertices ? : ");
   scanf("%d", &n);
   for ( i = 1 ; i <= n ; i++ )
        adj_list[i] = NULL;
   read_graph(adj_list, n);
   printf("\n Vertex \t Degree ");
   for ( i = 1 ; i <= n ; i++ )
   {
      deg = 0;
      p = adj_list[i];
      while( p != NULL )
      {
         deg++;
         p = p -> next;
      }
      printf("\n\n %5d \t\t %d\n\n", i, deg);
   }
   return;
}
int read_graph ( struct node *adj_list[10], int n )
{
   int i, j;
   char reply;
   struct node *p, *c;
   for ( i = 1 ; i <= n ; i++ )
   {
      for ( j = 1 ; j <= n ; j++ )
      {
         if ( i == j )
            continue;
         printf("\n Vertices %d & %d are Adjacent ? (Y/N) :", i, j);
         scanf("%c", &reply);
```

```
            if ( reply == 'y' || reply == 'Y' )
            {
               c = new_node;
               c -> vertex = j;
               c -> next = NULL;
               if ( adj_list[i] == NULL )
                  adj_list[i] = c;
               else
               {
                  p = adj_list[i];
                  while ( p -> next != NULL )
                  p = p -> next;
                  p -> next = c;
               }
            }
         }
      }
      return;
   }
```

Output:

```
A Program to represent a Graph by using an Adjacency Matrix method
1. Directed Graph
2. Un-Directed Graph
3. Exit
Select a proper choice :
How Many Vertices ? :
Vertices 1 & 2 are Adjacent ? (Y/N) : N
Vertices 1 & 3 are Adjacent ? (Y/N) : Y
Vertices 1 & 4 are Adjacent ? (Y/N) : Y
Vertices 2 & 1 are Adjacent ? (Y/N) : Y
Vertices 2 & 3 are Adjacent ? (Y/N) : Y
Vertices 2 & 4 are Adjacent ? (Y/N) : N
Vertices 3 & 1 are Adjacent ? (Y/N) : Y
Vertices 3 & 2 are Adjacent ? (Y/N) : Y
Vertices 3 & 4 are Adjacent ? (Y/N) : Y
Vertices 4 & 1 are Adjacent ? (Y/N) : Y
Vertices 4 & 2 are Adjacent ? (Y/N) : N
Vertices 4 & 3 are Adjacent ? (Y/N) : Y
```


Vertex	In_Degree	Out_Degree	Total_Degree
1	2	0	2
2	1	2	3
3	0	1	1
4	1	1	2

PRACTICE QUESTIONS

Q. I Multiple Choice Questions:

1. Which is a non-linear data structure?
   (a) Graph (b) Array
   (c) Queue (d) Stack
2. A graph G is represented as G = (V, E) where,
   (a) V is set of vertices (b) E is set of edges
   (c) Both (a) and (b) (d) None of these
3. In which representation, the graph is represented using a matrix of size total number of vertices by a total number of vertices.
   (a) Adjacency List (b) Adjacency Matrix
   (c) Adjacency Queue (d) Adjacency Stack
4. Each node of the graph is called a _____.
   (a) Edge (b) Path
   (c) Vertex (d) Cycle
5. Which representation of graph is based on linked lists?
   (a) Adjacency List (b) Adjacency Matrix
   (c) Both (a) and (b) (d) None of these
6. Which of a graph means visiting each of its nodes exactly once?
   (a) Insert (b) Traversal
   (c) Delete (d) Merge
7. Which is a vertex based technique for finding a shortest path in graph?
   (a) DFS (b) BST
   (c) BFS (d) None of these
8. Which usually implemented using a stack data structure?
   (a) DFS (b) BST
   (c) BFS (d) None of these
9. Which is a graph in which all the edges are uni-directional i.e. the edges point in a single direction?
   (a) Undirected (b) Directed
   (c) Cyclic (d) Acyclic

10. Which sorting involves displaying the specific order in which a sequence of vertices must be followed in a directed graph?
    (a) Cyclic (b) Acyclic
    (c) Topological (d) None of these
11. Which is a subset of an undirected graph that has all the vertices connected by minimum number of edges?
    (a) Spanning tree (b) Minimum spanning tree
    (c) Both (a) and (b) (d) None of these
12. Which is a subset of edges of a connected weighted undirected graph that connects all the vertices together with the minimum possible total edge weight?
    (a) Spanning tree (b) Minimum Spanning Tree (MST)
    (c) Both (a) and (b) (d) None of these

Answers

1. (a)	2. (c)	3. (b)	4. (c)	5. (a)	6. (b)	7. (c)
8. (a)	9. (b)	10.(c)	11. (a)	12. (b)		

Q. II Fill in the Blanks:

1. Graph represented as _____.
2. A graph is _____ if the graph comprises a path that starts from a vertex and ends at the same vertex.
3. Graph is a _____ data structure.
4. Individual data element of a graph is called as _____ (also known as node). An edge is a connecting link between two vertices. Edge is also known as _____.
5. Graph traversal is a technique used for a _____ vertex in a graph.
6. We use _____ data structure with maximum size of total number of vertices in the graph to implement DFS traversal.
7. The number of edges connected directly to the node is called as _____ of node.
8. The number edges pointing _____ the node are called in-degree/in-order.
9. A graph which has set of empty edges or is containing only isolated nodes is called a _____ graph or isolated graph.
10. The _____ (or path) between two vertices is called an edge.
11. We use _____ data structure with maximum size of total number of vertices in the graph to implement BFS traversal.
12. The graph traversal is also used to decide the order of vertices is _____ in the search process.

13. The sequence of nodes that we need to follow when we have to travel from on vertex to another in a graph is called the _____.
14. An adjacency _____ is a matrix of size n x n where n is the number of vertices i the graph.
15. Topological ordering of vertices in a graph is possible only when the graph is _____ acyclic graph.
16. A _____ (or minimum weight spanning tree) for a weighted, connected an undirected graph is a spanning tree with weight less than or equal to the weight o every other spanning tree.
17. The _____ algorithm is an example of an all-pairs shortest paths algorithm.
18. A graph _____ cycle is called acyclic graph.

Answers

1. G = (V, E)	2. cyclic	3. non-linear	4. Vertex, Arc
5. searching	6. Stack	7. degree	8. toward
9. Null	10. link	11. Queue	12. visited
13. path	14. matri	15. directed	
16. Minimum Spanning Tree (MST)		17. Floyd-Warshall	18. without

Q. III State True or False:

1. An acyclic graph is a graph that has no cycle.
2. A Graph consists of a finite set of vertices (or nodes) and set of Edges which connect a pair of nodes.
3. A graph in which weights are assigned to every edge is called acyclic graph.
4. The Floyd-Warshall algorithm is for solving the all pairs shortest path problem.
5. The number edges pointing away from the node are called out-degree/out-order.
6. In a graph, if two nodes are connected by an edge then they are called adjacent nodes or neighbors.
7. Dijkstra's Algorithm can be applied to either a directed or an undirected graph to find the shortest path to each vertex from a single source.
8. The graph is a linear data structures.
9. BFS usually implemented using a queue data structure.
10. The spanning tree does not have any cycle (loops).
11. Prim's Algorithm will find the minimum spanning tree from the graph G.
12. Given an undirected, connected and weighted graph, find Minimum Spanning Tree (MST) of the graph using Kruskal's algorithm.

13. The number of edges that are connected to a particular node is called the path of the node.
14. A spanning tree of that graph is a subgraph that is a tree and connects all the vertices together.
15. Directed graph is also called as Digraph.

Answers

1. (T)	2. (T)	3. (F)	4. (T)	5. (T)	6. (T)	7. (T)	8. (T)
9. (F)	10. (T)	11. (T)	12. (T)	13. (F)	14. (T)	15. (T)	

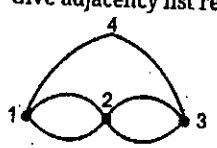
Q. IV Answer the following Questions:

(A) Short Answer Questions:

1. What is graph?
2. List traversals of graph.
3. Define the term cycle and path in graph.
4. Which data structure is used in to represent graph in adjacency list.
5. Define in-degree and out-degree of vertex.
6. What is weighted graph.
7. Define spanning tree.
8. List ways to represent a graph.
9. What are the applications of graph?
10. What is BFS and DFS.
11. Give uses of graphs in social networks.
12. Define MST?

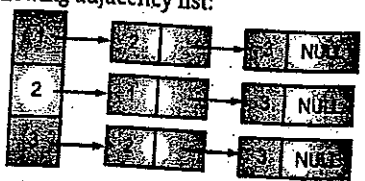
(B) Long Answer Questions:

1. Define graph. How to represent it? Explain with diagram.
2. What is topological sort. Explain with example.
3. What DFS? Describe in detail.
4. Give adjacency list representation of following graph:

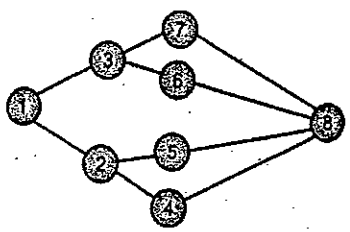


5. Write short note on: Inverse adjacency list of graph.
6. Describe following algorithms with example:
   (i) Dijkstra's algorithm
   (ii) Floyd Warshall.

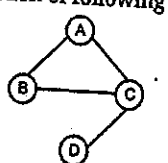
7. With the help of diagram describe adjacency matrix representation of graph.
8. What is adjacency multi-list? How to represent it? Explain with example.
9. With the help of example describe BFS.
10. Describe the term MST with example.
11. Draw graph for following adjacency list:



12. Describe Prim's algorithm in detail.
13. Explain Kruskal's algorithm with example.
14. For the following graph, give result of depth first traversal and breadth first traversal:



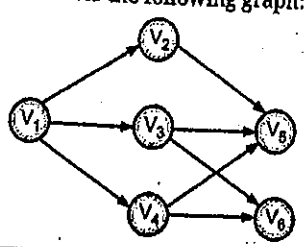
15. Give adjacency list representation of following graph.



UNIVERSITY QUESTIONS AND ANSWERS

April 2016

1. Consider the following graph: **[4 M]**



(i) Write adjacency matrix

(ii) Draw adjacency list

(iii) DFS and BFS traversals (start vertex v_1).

Ans. Refer to Sections 3.2.1, 3.2.2 and 3.3.

2. Define the following terms: **[2 M]**

(i) Degree of vertex

(ii) Topological sort.

Ans. Refer to Sections 3.1.2, Point (5) and 3.4.1.

April 2017

1. Which data structure is used for BFS. **[1 M]**

Ans. Refer to Section 3.3.2.

2. Define the term complete graph. **[1 M]**

Ans. Refer to Section 3.1.2, Point (3).

October 2017

1. Consider the following adjacency matrix: **[5 M]**

	1	2	3	4
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1
4	1	0	0	0

(i) Draw the graph

(ii) Draw adjacency list

(iii) Draw inverse adjacency list.

Ans. Refer to Section 3.2.1, 3.2.2 and 3.2.3.

2. Write an algorithm for BFS traversal of a graph. **[3 M]**

Ans. Refer to Section 3.3.2.

April 2018

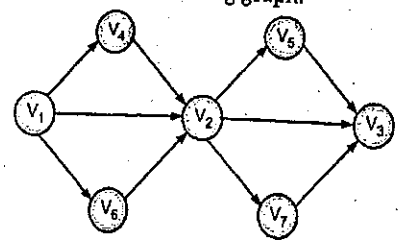
1. List any methods of representing graphs. **[1 M]**

Ans. Refer to Section 3.2.

2. List two applications of graph. **[1 M]**

Ans. Refer to Section 3.4.

3. Consider the following graph: **[5 M]**



Starting vertex v_1

(i) Draw adjacency list

(ii) Give DFS and BFS traversals.

Ans. Refer to Sections 3.2.2 and 3.3.

October 2018

1. State any two applications of graph. **[1 M]**

Ans. Refer to Section 3.4.

2. Consider the following specification of a graph G:

V(G) = {1, 2, 3, 4}

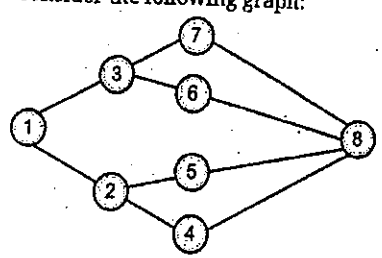
E(G) = {(1, 2),(1, 3),(3, 3),(3, 4),(4, 1)}

(i) Draw a picture of the undirected graph.

(ii) Draw adjacency matrix of lists. **[5 M]**

Ans. Refer to Sections 3.2.1 and 3.2.2.

3. Consider the following graph: **[4 M]**



(i) Write adjacency matrix

(ii) Give DFS and BFS (Source vertex v_1). **[3 M]**

Ans. Refer to Sections 3.2.1 and 3.3.

4. Define the following terms: **[2 M]**

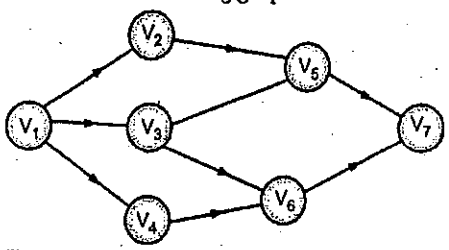
(i) Acrylic graph

(ii) Multigraph.

Ans. Refer to Section 3.1.2.

April 2019

1. Consider the following graph: [1 M]



(i) Write adjacency matrix
(ii) Draw adjacency list
(iii) DFS and BFS traversals (start vertex v_1).

Ans. Refer to Sections 3.2.1, 3.2.2 and 3.3.

2. Define the term topological sort. [1 M]

Ans. Refer to Section 3.4.1.

CHAPTER 4

Hash Table

Objectives ...

- To study Basic Concepts of Hashing
- To learn Hash Table and Hash Function
- To understand Terminologies in Hashing
- To study Collision Resolution Techniques

4.0 INTRODUCTION

- Hashing is a data structure which is designed to use a special function called the hash function which is used to map a given value with a particular key for faster access of elements.
- Hashing is the process of mapping large amount of data item to smaller table with the help of hashing function.
- The mapping between an item and the slot where that item belongs in the hash table is called the hash function.
- The hash function will take any item in the collection and return an integer in the range of slot names, between 0 and m–1.
- A hash table is a collection of items which are stored in such a way as to make it easy to find them later.

Need for Hash Table Data Structure:

- In the linear search and binary search, the location of the item is determined by a sequence/series of comparisons. The data item to be searched is compared with items at certain locations in the list.
- If any item/element matches with the item to be searched, the search is successful. The number of comparisons required to locate an item depends on the data structure like array, linked list, sorted array, binary search tree, etc. and the search algorithm used.
- For example, if the items are stored in sorted order in an array, binary search can be applied which locates an item in O(log n) comparisons.
- On the other hand, if an item is to be searched in a linked list or an unsorted array, linear search has to be applied which locates an item in O(n) comparisons.

- However, for some applications, searching is very critical operation and they need a search algorithm which performs search in constant time, i.e. O(1).
- Although, ideally it is almost impossible to achieve a performance of O(1), but still a search algorithm can be derived which is independent of n and can give a performance very close to O(1).
- That search algorithm is called hashing. Hashing uses a data structure called hash table which is merely an array of fixed size and items in it are inserted using a function called hash function.
- Best case timing behavior of searching using hashing = O(1) and Worst case timing Behavior of searching using hashing = O(n).
- A hash table is a data structure in which the location of a data item is determined directly as a function of the data item itself rather than by a sequence of comparisons.
- Under ideal condition, the time required to locate a data item in a hash table is O(1) i.e. it is constant and does nor depend on the number of data items stored.

4.1 CONCEPT OF HASHING

- Hashing is the process of indexing and retrieving element (data) in a data structure to provide a faster way of finding the element using a hash key.
- Here, the hash key is a value which provides the index value where the actual data is likely to be stored in the data structure.
- In this data structure, we use a concept called hash table to store data. All the data values are inserted into the hash table based on the hash key value.
- The hash key value is used to map the data with an index in the hash table. And the hash key is generated for every data using a hash function.
- That means every entry in the hash table is based on the hash key value generated using the hash function.
- Hash table is just an array which maps a key (data) into the data structure with the help of hash function such that insertion, deletion and search operations are performed.
- Generally, every hash table makes use of a function called hash function to map the data into the hash table.
- Hash function is a function which takes a piece of data (i.e. key) as input and produces an integer (i.e. hash value) as output which maps the data to a particular index in the hash table.
- Basic concept of hashing and hash table is shown in the Fig. 4.1.

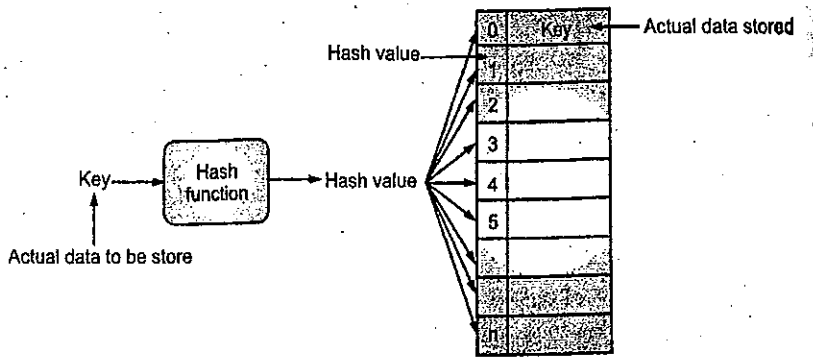


Fig. 4.1: Pictorial representation of Simple Hashing

4.2 TERMINOLOGY

- The basic terms used in hashing are explained below:
  1. **Hash Table:** A hash table is a data structure that is used to store keys/value pairs. Hashing uses a data structure called hash table which is merely an array of fixed size and items in it are inserted using a hash function. It uses a hash function to compute an index into an array in which an element will be inserted or searched. A hash table is a data structure that maps keys to values. a hash table (hash map) is a data structure that implements an associative array abstract data type, a structure that can map keys to values. A hash table uses a hash function to compute an index, also called a hash code, into an array of buckets or slots, from which the desired value can be found.
  2. **Hash Function:** A hash function is a function that maps the key to some slot in the hash table. A hash function, is a mapping function which maps all the set of search keys to the address where actual records are placed.
  3. **Bucket:** A hash file stores data in bucket format. Bucket is considered a unit of storage. A bucket typically stores one complete disk block, which in turn can store one or more records.
  4. Hash Address: A hash function is a function which when given a key, generates an address in the table. Hash index is an address of the data block.
  5. **Collision:** The situation where a newly inserted key maps to an already occupied slot in the hash table is called collision. A collision occurs when two data elements are hashed to the same value and try to occupy the same space in the hash table. In simple words, the situation in which a key hashes to an index which is already occupied by another key is called as collision.
  6. **Synonym:** It is possible for different keys to hash to the same array location. This situation is called collision and the colliding keys are called synonyms.

7. **Overflow:** An overflow occurs at the time of the bucket for a new pair (key, element) is full.
8. **Open Addressing:** It is performed to ensure that all elements are stored directly into the hash table, thus it attempts to resolve collisions implementing various methods.
9. **Linear Probing:** It is performed to resolve collisions by placing the data into the next open slot in the table.
10. **Hashing:** Hashing is a process that uses a hash function to get the key for the hash table and transform it into an index that will point to different arrays of buckets, which is where the information will be stored.
11. **Chaining:** It is a technique used for avoiding collisions in hash tables.
12. **Collision Resolution:** Collision should be resolved by finding some other location to insert the new key, this process of finding another location is called as collision resolution.

4.3 PROPERTIES OF GOOD HASH FUNCTION

- The properties of a good hash function are given below:
  1. The hash function is easy to understand and simple to compute.
  2. A number of collisions should be less while placing the data in the hash table.
  3. The hash function should generate different hash values for the similar string.
  4. The hash function "uniformly" distributes the data across the entire set of possible hash values.
  5. The hash function is a perfect hash function when it uses all the input data. A hash function that maps each item into a unique slot is referred to as a perfect hash function.

4.4 HASH FUNCTIONS

- A hash function h is simply a mathematical formula that manipulates the key in some form to compute the index for the key in the hash table.
- The process of mapping keys to appropriate slots in a hash table is known as hashing. Hash function is a function which is used to put the data in the hash table.
- Hence, one can use the same hash function to retrieve the data from the hash table. Thus, hash function is used to implement the hash table.
- A hash function h is simply a mathematical formula that maps the key to some slot in the hash table T.
- Thus, we can say that the key k hashes to slot h(k), or h(k) is the hash value of key k. If the size of the hash table is N, then the index of the hash table ranges from 0 to N-1. A hash table with N slots is denoted by T[N].
- Hashing (also known as hash addressing) is generally applied to a file F containing R records. Each record contains many fields, out of these one particular field may uniquely identify the records in the file.

- Such a field is known as primary key (denoted by k). The values k_1, k_2,... k_n in the key field are known as keys or key values.
- The key through an algorithmic function determines the location of a particular record.
  The algorithmic function i.e. hashing function basically performs the key-to-address transformation in which key is mapped to the addresses of records in the file as shown in Fig. 4.2.

Fig. 4.2: Hash Function

4.4.1 Division Method

- In division method, the key k is divided by the number of slots N in the hash table, and the remainder obtained after division is used as an index in the hash table. That is, the hash function is,
  $h(k) = k \bmod N$
  where, mod is the modulus operator. Different languages have different operators for calculating the modulus. In C/C++, '%' operator is used for computing the modulus.
- For example, consider a hash table with N=101. The hash value of the key value 132437 can be calculated as follows:
  $h(132437) = 132437 \bmod 101 = 26$
- Note that above hash function works well if the index ranges from 0 to N-1 (like in C/C++). However, if the index ranges from 1 to N, the function will be,
  $h(k) = k \bmod N + 1$
- This technique works very well if N is either a prime number not too close to a power of two. Moreover, since this technique requires only a single division operation, it is quite fast.
- For examole, suppose, k = 23, N = 10 then
  $h(23) = 23 \bmod 10 + 1 = 3 + 1 = 4$. The key whose value is 23 is placed in 4^{th} location.
- Take another example, consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key, value) format.hash function is, h(k) = k mod 20.
  (1, 20)
  (2, 70)
  (42, 80)
  (4, 25)
  (12, 44)
  (14, 32)
  (17, 11)
  (13, 78)
  (37, 98)

- Hashing is a technique to convert a range of key values into a range of indexes of an array.

Sr. No.	Key	Hash	Array Index
1.	1	1 % 20 = 1	1
2.	2	2 % 20 = 2	2
3.	42	42 % 20 = 2	2
4.	4	4 % 20 = 4	4
5.	12	12 % 20 = 12	12
6.	14	14 % 20 = 14	14
7.	17	17 % 20 = 17	17
8.	13	13 % 20 = 13	13
9.	37	37 % 20 = 17	17

Basic Operations:

- Following are the basic primary operations of a hash table:
  1. **Search:** Searches an element in a hash table.
  2. **Insert:** inserts an element in a hash table.
  3. **Delete:** Deletes an element from a hash table.

Program 4.1: Program for operations on hashing.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#define SIZE 20
struct DataItem
{
   int data;
   int key;
};
struct DataItem* hashArray[SIZE];
struct DataItem* dummyItem;
struct DataItem* item;
int hashCode(int key)
{
   return key % SIZE;
}
```



```
struct DataItem *search(int key)
{
   //get the hash
   int hashIndex = hashCode(key);
   //move in array until an empty
   while(hashArray[hashIndex] != NULL)
   {
      if(hashArray[hashIndex]->key == key)
      return hashArray[hashIndex];
      //go to next cell
      ++hashIndex;
      //wrap around the table
      hashIndex %= SIZE;
   }
   return NULL;
}
void insert(int key,int data)
{
   struct DataItem *item = (struct DataItem*) malloc(sizeof(struct DataItem));
   item->data = data;
   item->key = key;
   //get the hash
   int hashIndex = hashCode(key);
   //move in array until an empty or deleted cell
   while(hashArray[hashIndex] != NULL && hashArray[hashIndex]->key != -1)
   {
      //go to next cell
      ++hashIndex;
      //wrap around the table
      hashIndex %= SIZE;
   }
   hashArray[hashIndex] = item;
}
struct DataItem* delete(struct DataItem* item)
{
   int key = item->key;
   //get the hash
   int hashIndex = hashCode(key);
   //move in array until an empty
```

```
   while(hashArray[hashIndex] != NULL)
   {
      if(hashArray[hashIndex]->key == key)
      {
         struct DataItem* temp = hashArray[hashIndex];
         //assign a dummy item at deleted position
         hashArray[hashIndex] = dummyItem;
         return temp;
      }
      //go to next cell
      ++hashIndex;
      //wrap around the table
      hashIndex %= SIZE;
   }
   return NULL;
}
void display()
{
   int i = 0;
   for(i = 0; i<SIZE; i++)
   {
      if(hashArray[i] != NULL)
         printf(" (%d,%d)",hashArray[i]->key,hashArray[i]->data);
      else
         printf(" ~~ ");
   }
   printf("\n");
}
int main()
{
   dummyItem = (struct DataItem*) malloc(sizeof(struct DataItem));
   dummyItem->data = -1;
   dummyItem->key = -1;
   insert(1, 20);
   insert(2, 70);
   insert(42, 80);
   insert(4, 25);
   insert(12, 44);
```



```
   insert(14, 32);
   insert(17, 11);
   insert(13, 78);
   insert(37, 97);
   display();
   item = search(37);
   if(item != NULL)
   {
      printf("Element found: %d\n", item->data);
   } else
   {
      printf("Element not found\n");
   }
   delete(item);
   item = search(37);
   if(item != NULL)
   {
      printf("Element found: %d\n", item->data);
   } else
   {
      printf("Element not found\n");
   }
}
```

Output:

```
$gcc -o hashpgm *.c
$hashpgm
~~ (1,20) (2,70) (42,80) (4,25) ~~ ~~ ~~ ~~ ~~ ~~ ~~ (12,44)
(13,78) (14,32) ~~ ~~ (17,11) (37,97) ~~
Element found: 97
Element not found
```

4.6.2 Mid Square Method

- In this method, we square the value of a key and take the number of digits required to form an address, from the middle position of squared value.
- Suppose a key value is 16, then its square is 256. Now if we want address of two digits, then we select the address as 56 (i.e., two digits starting from middle of 256).
- Mid square method operates in following two steps:
  1. First, the square of the key k (that is, k^2) is calculated and
  2. Then some of the digits from left and right ends of k^2 are removed.

- The number obtained after removing the digits is used as the hash value. Note that the digits at the same position of k^2 must be used for all keys.
- Thus, the hash function is given below,

  $h(k) = s$

  where, s is obtained by deleting digits from both sides of k^2.
- For example, consider a hash table with N = 1000. The hash value of the key value 132437 can be calculated as follows:
  1. The square of the key value is calculated, which is, 17539558969.
  2. The hash value is obtained by taking 5^{th}, 6^{th} and 7^{th} digits counting from right, which is, 955.

4.4.3 Folding Method

- The folding method also operates in two steps. In the first step, the key value k is divided into number of parts, k^1, k^2, k^3, ... , k^r, where each part has the same number of digits except the last part, which can have lesser digits.
- In the second step, these parts are added together and the hash value is obtained by ignoring the last carry, if any. For example, if the hash table has 1000 slots, each part will have three digits, and the sum of these parts after ignoring the last carry will also be three-digit number in the range of 0 to 999.
- For example, if the hash table has 100 slots, then each group will have two digits, and the sum of the groups after ignoring the last carry will also be a 2-digit number between 0 and 99. The hash value for the key value 132437 is computed as follows:
  1. The key value is first broken down into a group of 2-digit numbers from the left most digits. Therefore, the groups are 13, 24 and 37.
  2. These groups are then added like 13 + 24 + 37 = 74. The sum 74 is now used the hash value for the key value 132437.
- Similarly, the hash value of another key value, say 6217569, can be calculated as follows:
  1. The key value is first broken down into a group of 2-digit numbers from the left most digits. Therefore, the groups are 62, 17, 56 and 9.
  2. These groups are then added like 62 + 17 + 56 + 9 = 144. The sum 44 after ignoring the last carry 1 is now used as the hash value for the key value 6217569.

4.5 COLLISION RESOLUTION TECHNIQUES

- Collision resolution is the main problem in hashing. The situation in which a key hashes to an index which is already occupied by another key is called collision.
- Collision should be resolved by finding some other location to insert the new key. This process of finding another location is called collision resolution.

- If the element to be inserted is mapped to the same location, where an element is already inserted then we have a collision and it must be resolved.
- There are several strategies for collision resolution. The most commonly used are:
  1. **Separate Chaining:** Used with open hashing.
  2. **Open Addressing:** Used with closed hashing.

4.5.1 Open Addressing

- Open addressing or closed hashing is a method of collision resolution in hash tables. With this method a hash collision is resolved by probing, or searching through alternate locations in the array (the probe sequence) until either the target record is found.
- In open addressing separate data structure is used because all the key values are stored in the hash table itself. Since, each slot in the hash table contains the key value rather than the address value, a bigger hash table is required in this case as compared to separate chaining.
- Some value is used to indicate an empty slot. For example, if it is known that all the keys are positive values, then –1 can be used to represent a free or empty slot.
- To **insert a key value**, first the slot in the hash table to which the key value hashes is determined using any hash function. If the slot is free, the key value is inserted into that slot.
- In case the slot is already occupied, then the subsequent slots, starting from the occupied slot, are examined systematically in the forward direction, until an empty slot is found. If no empty slot is found, then overflow condition occurs.
- In case of **searching of a key value** also, first the slot in the hash table to which the key value hashes is determined using any hash function. Then the key value stored in that slot is compared with the key value to be searched.
- If they match, the search operation is successful; otherwise alternative slots are examined systematically in the forward direction to find the slot containing the desired key value. If no such slot is found, then the search is unsuccessful.
- The process of examining the slots in the hash table to find the location of a key value is known as probing. The linear probing, quadratic probing and double hashing that are used in open addressing method.

4.5.1.1 Linear Probing

- Linear probing is a technique for resolving collisions in hash tables, data structures for maintaining a collection of key-value pairs and looking up the value associated with a given key.
- In linear probing whenever there is a collision, cells are searched sequentially (with wraparound) for searching the hash-table free location.

- Let us suppose, if k is the index retrieved from the hash function. If the kth index is already filled then we will look for (k+1) %M, then (k+2) %M and so on. When we get a free slot, we will insert the object into that free slot.
- Below table shows the result of Inserting keys (5,18,55,78,35,15) using the hash function h(k) = k % 10 and linear probing strategy.

	Empty Table	After 5	After 18	After 55	After 78	After 35	After 15
0							15
1							
2							
3							
4							
5		5	5	5	5	5	5
6				55	55	55	55
7						35	35
8			18	18	18	18	18
9					78	78	78

- Linear probing is easy to implement but it suffers from "primary clustering".
- When many keys are mapped to the same location (clustering), linear probing will not distribute these keys evenly in the hash table. These keys will be stored in neighborhood of the location where they are mapped. This will lead to clustering of keys around the point of collision.

Example 1: Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key, value) format. Here, we can search the next empty location in the array by looking into the next cell until we find an empty cell. This technique is called linear probing.

(1, 20)
(2, 70)
(42, 80)
(4, 25)
(12, 44)
(14, 32)
(17, 11)
(13, 78)
(37, 98)

Solution:

Sr. No.	Key	Hash	Array Index	After Linear Probing, Array Index
1.	1	1 % 20 = 1	1	1
2.	2	2 % 20 = 2	2	2
3.	42	42 % 20 = 2	2	3
4.	4	4 % 20 = 4	4	4
5.	12	12 % 20 = 12	12	12
6.	14	14 % 20 = 14	14	14
7.	17	17 % 20 = 17	17	17
8.	13	13 % 20 = 13	13	13
9.	37	37 % 20 = 17	17	18

Example 2: A hash table of length 10 uses open addressing with hash function h(k)=k mod 10, and linear probing. After inserting 6 values into an empty hash table, the table is as shown below:

0	
1	
2	42
3	23
4	34
5	52
6	46
7	33
8	
9	

What is a possible order in which the key values could have been inserted in the table?

Solution: 46, 34, 42, 23, 52, 33 is the sequence in which the key values could have been inserted in the table.

How many different insertion sequences of the key values using the same hash function and linear probing will result in the hash table shown above?

Solution: 30

In a valid insertion sequence, the elements 42, 23 and 34 must appear before 52 and 33, and 46 must appear before 33.

Total number of different sequences = $3! \times 5 = 30$

In the above expression, 3! is for elements 42, 23 and 34 as they can appear in any order, and 5 is for element 46 as it can appear at 5 different places.

4.5.1.2 Quadratic Probing

- Quadratic probing is a collision resolving technique in open addressing hash table. It operates by taking the original hash index and adding successive values of an arbitrary quadratic polynomial until an open slot is found.
- One way of reducing "primary clustering" is to use quadratic probing to resolve collision. In quadratic probing, we try to resolve the collision of the index of a hash table by quadratically increasing the search index free location.
- Let us suppose, if k is the index retrieved from the hash function. If the kth index is already filled then we will look for $(k+1^2)$ %M, then $(k+2^2)$ %M and so on. When we get a free slot, we will insert the object into that free slot.
- It does not ensure that all cells in the table will be examined to find an empty cell. Thus, it may be possible that key will not be inserted even if there is an empty cell in the table.

Double Hashing:

- Double hashing is a collision resolving technique in open addressing hash table. Double hashing uses the idea of using a second hash function to key when a collision occurs.
- This method requires two hashing functions for f1(key) and f2 (key). Problem of clustering can easily be handled through double hashing.
- Function f1 (key) is known as primary hash function. In case the address obtained by f1 (key) is already occupied by a key, the function f2 (key) is evaluated.
- The second function f2 (key) is used to compute the increment to be added to the address obtained by the first hash function f1 (key) in case of collision.
- The search for an empty location is made successively at the addresses f1 (key) + f2 (key), f1 (key) + 2f2 (key), f1 (key) + 3f2(key), ...

4.5.1.3 Rehashing

- As the name suggests, rehashing means hashing again. Rehashing is a technique in which the table is resized, i.e. the size of table is doubled by creating a new table.
- Several deletion operations are intermixed with insertion operations while performing operations on hash table. Eventually, a situation arises when the hash table becomes almost full.
- At this time, it might happen that the insert, delete, and search operations on the hash table take too much time. The insert operation even fails in spite of performing open addressing with quadratic probing for collision resolution. This condition/situation indicates that the current space allocated to the hash table is not sufficient to accommodate all the keys.

- A simple solution to this problem is rehashing, in which all the keys in the original hash table are rehashed to a new hash table of larger size.
- The default size of the new hash table is twice as that of the original hash table. Once the new hash table is created, a new hash value is computed for each key in the original hash table and the keys are inserted into the new hash table.
- After this, the memory allocated to the original hash table is freed. The performance of the hash table improves significantly after rehashing.

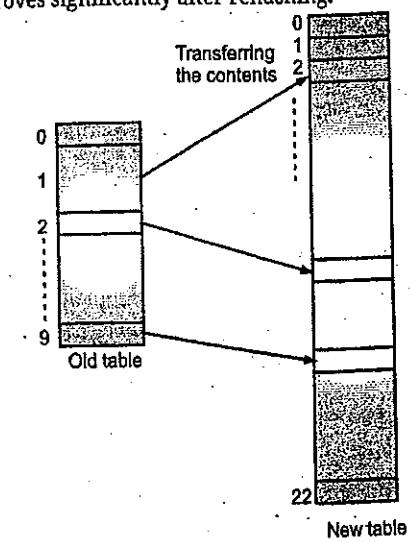


Fig. 4.3: Rehashing

4.5.2 Chaining

- In hashing the chaining is one collision resolution technique. Chaining is a possible way to resolve collisions. Each slot of the array contains a link to a singly-linked list containing key-value pairs with the same hash.
- Chaining allows storing the key elements with the same hash value into linked list as shown in Fig. 4.4.
- Thus, cach slots in the hash table contains a pointer to the head of the linked list of all the elements that hashes to the value h.
- All collisions are chained in the lists attached to the appropriate slot. This allows an unlimited number of collisions to be handled and does not require a prior knowledge of how many elements are contained in the collection.
- The tradeoff is the same as with linked lists versus array implementations of collections: linked list overhead in space and to a lesser extent, in time.

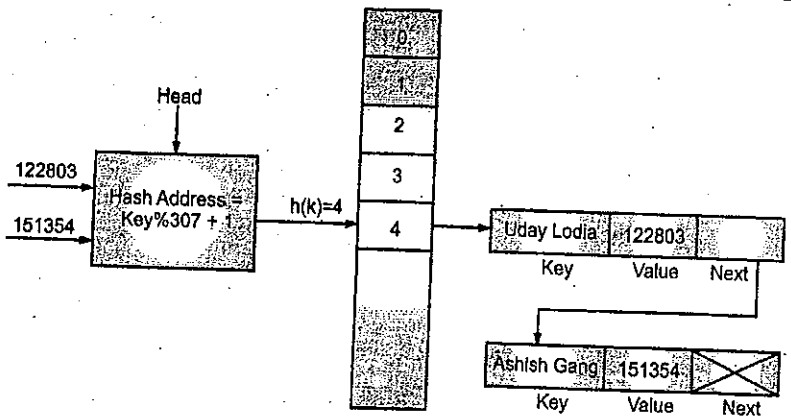


Fig. 4.4: Chaining through Linked List

4.5.1 Coalesced Chaining

- Coalesced hashing also called coalesced chaining is a technique of collision resolution in a hash table that forms a hybrid of separate chaining and open addressing.
- It uses the concept of open addressing (linear probing) to find first empty place for colliding element from the bottom of the hash table and the concept of separate chaining to link the colliding elements to each other through pointers.
- Given a sequence "qrj," "aty," "qur," "dim," "ofu," "gcl," "rhv," "clq," "ecd," "qsu" of randomly generated three character long strings, the following table would be generated with a table of size 10.

0	(null)		
1	"clq"		
2	"qur"		
3	(null)		
4	(null)		
5	"dim"		
6	"aty"	"qsu"	
7	"rhv"		
8	"qrj"	"ofu"	"gel"	"ecd"
9	(null)		

- Fig. 4.5 shows an example of coalesced hashing example (for purpose of this example, collision buckets are allocated increasing order, starting with bucket 0).

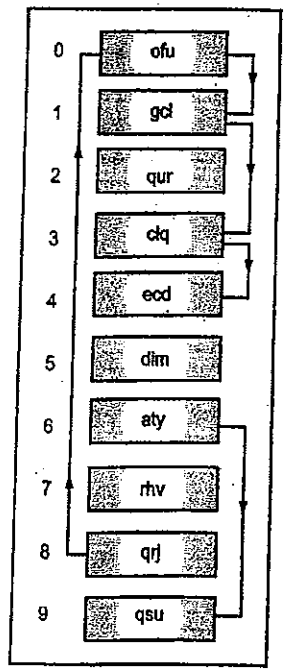


Fig. 4.5

- Coalesced hashing technique is effective, efficient, and very easy to implement.

4.5.2 Separate Chaining

- In separate chaining, hash table is an array of pointers and each element of array points to the first element of the linked list of all the records that hashes to that location.
- Open hashing is a collision avoidance method which uses array of linked list to resolve the collision. It is also known as the separate chaining method (each linked list is considered as a chain).
- In separate chaining collision resolution technique, a linked list of all the key values that hash to the same hash value is maintained. Each node of the linked list contains a key value and the pointer to the next node.
- Each index i (0 <= I < N) in the hash table contains the address of the first node of the linked list containing all the keys that hash to the index i.
- If there is no key value that hashes to the index i, the slot contains NULL value. Therefore, in this method, a slot in the hash table does not contain the actual key values; rather it contains the address of the first node of the linked list containing the elements that hash to this slot.

- In separate chaining technique, a separate list of all elements mapped to the same value is maintained. Separate chaining is based on collision avoidance.
- If memory space is tight, separate chaining should be avoided. Additional memory space for links is wasted in storing address of linked elements.
- Hashing function should ensure even distribution of elements among buckets; otherwise the timing behavior of most operations on hash table will deteriorate.
- Fig. 4.6 shows a separate chaining hash table.

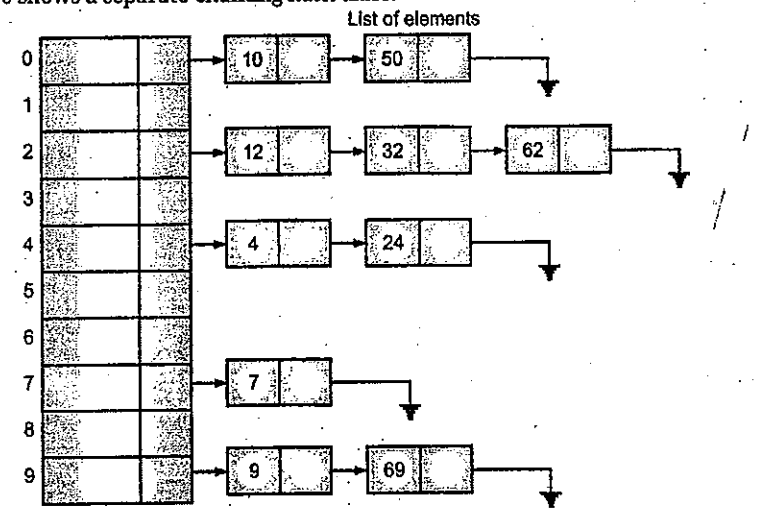


Fig. 4.6: A separate Chaining Hash Table

Example 1: The integers given below are to be inserted in a hash table with 5 locations using chaining to resolve collisions. Construct hash table and use simplest hash function.
1, 2, 3, 4, 5, 10, 21, 22, 33, 34, 15, 32, 31, 48, 49, 50,

Solution: An element can be mapped to a location in the hash table using the mapping function h(k) = k % 10.

Hash Table Location	Mapped Elements
0	5, 10, 15, 50
1	1, 21, 31
2	2, 22, 32
3	3, 33, 48
4	4, 34, 49

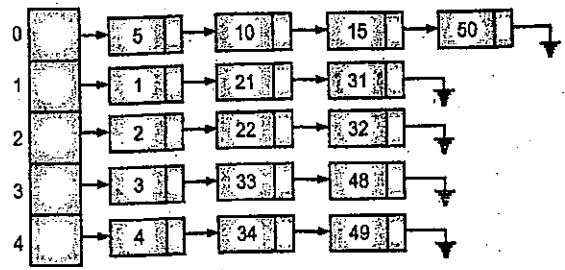


Fig. 4.7

Example 2: Consider the key values 20, 32, 41, 66, 72, 80, 105, 77, 56, 53 that need to be hashed using the simple hash function h(k) = k mod 10. The keys 20 and 80 hash to index 0, key 41 hashes to index 1, keys 32 and 72 hashes to index 2, key 53 is hashed to index 3, key 105 is hashed to index 5, keys 66 and 56 are hashed to index 6 and finally the key 77 is hashed to index 7. The collision is handled using the separate chaining (also known as synonyms chaining) technique as shown in Fig. 4.8.

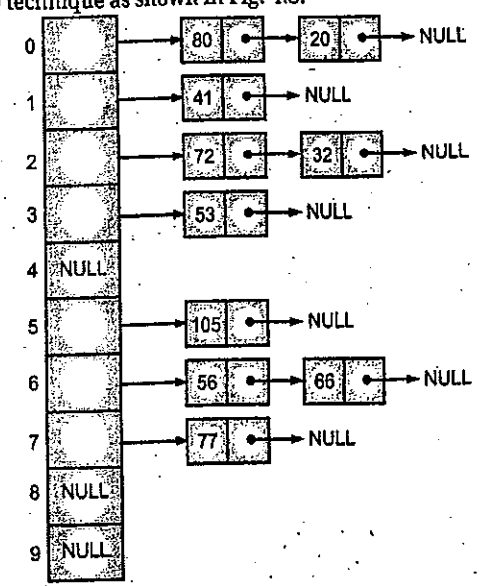


Fig. 4.8: Collision Resoltuion by Separate Chaining

Comparison between Separate Chaining and Open Addressing:

Sr. No.	Separate Chaining	Open Addressing
1.	Chaining is simpler to implement.	Open addressing requires more computation.
2.	In chaining, hash table never fills up, we can always add more elements to chain.	In open addressing, table may become full.
3.	Chaining is less sensitive to the hash function or load factors.	Open addressing requires extra care for to avoid clustering and load factor.
4.	Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.	Open addressing is used when the frequency and number of keys is known.
5.	Cache performance of chaining is not good as keys are stored using linked list.	Open addressing provides better cache performance as everything is stored in the same table.
6.	Wastage of space (some parts of hash table in chaining are never used).	In open addressing, a slot can be used even if an input does not map to it.
7.	Chaining uses extra space for links.	No links in open addressing.

PRACTICE QUESTIONS

Q. I Multiple Choice Questions:

1. Which data structure is designed to use a special function called the hash function?
   (a) Hashing (b) Stacking
   (c) Queueing (d) None of these
2. Which is a data structure that represents data in the form of key-value pairs?
   (a) Hash function (b) Hash table
   (c) Hashing (d) None of these
3. Which is any function that can be used to map a data set of an arbitrary size to a data set of a fixed size, which falls into the hash table?
   (a) Hash function (b) Hash table
   (c) Hash values (d) None of these

4. A good hash function has the following properties:
   (a) **Easy to compute:** It should be easy to compute and must not become an algorithm in itself.
   (b) **Uniform distribution:** It should provide a uniform distribution across the hash table and should not result in clustering.
   (c) **Less collisions:** Collisions occur when pairs of elements are mapped to the same hash value. These should be avoided.
   (d) All of these
5. Which tables are used to perform insertion, deletion and search operations very quickly in a data structure?
   (a) Root (b) Hash
   (c) Routing (d) Root
6. Which is in a hash file is unit of storage that can hold one or more records?
   (a) Bucket (b) Hash values
   (c) Token (d) None of these
7. A _____ occurs when two data elements are hashed to the same value and try to occupy the same space in the hash table.
   (a) Bucket (b) Underflow
   (c) Collision (d) Token
8. In which method all the key values are stored in the hash table itself?
   (a) Open addressing (b) Chaining
   (c) Separate chaining (d) None of these
9. Which is a technique in all the keys in the original hash table are rehashed to a new hash table of larger size?
   (a) Chaining (b) Hashing
   (c) Rehashing (d) None of these
10. In which hashing method, successive slots are searched using another hash function?
    (a) Linear probing (b) Quadratic probing
    (c) Double (d) All of these
11. Name which methods are used by open addressing for hashing?
    (a) Linear probing (b) Quadratic probing
    (c) Double hashing (d) All of these

12. Hashing is implementing using,
    (a) Hash table (b) Hash function
    (c) Both (a) and (b) (d) None of these
13. Hash table is a data structure that represents data in the form of,
    (a) value-key pairs (b) key-value pairs
    (c) key-key pairs (d) None of these

Answers

1. (d)	2. (c)	3. (b)	4. (a)	5. (c)	6. (d)	7. (c)
8. (a)	9. (c)	10. (b)	11. (d)	12. (a)	13. (b)	

Q. II Fill in the Blanks:

1. _____ is the process of mapping large amount of data item to a hash table with the help of hash function.
2. The situation where a newly inserted key maps to an already occupied slot in the hash table is called _____.
3. In rehashing the default size of the new table is _____ as that of the original hash table.
4. The process of examining the slots in the hash table to find the location of a key value is known as _____.
5. A _____ function is simply a mathematical formula that manipulates the key in some form to compute the index for this kay in the hash table.
6. Hash _____ is an array of fixed size and items in it are inserted using a hash function.
7. In _____ method the key is divided by number of slots in the hash table and the remainder obtained after division is used as an index in the hash table.
8. In _____ hash table is an array of pointers and each element/item of array points to the first element of the linked list of all the records that hashes to that location.
9. _____ allows storing the key elements with the same hash value into linked list.
10. _____ hashing is a combination of both Separate chaining and Open addressing.

Answers

1. Hashing	2. collision	3. twice	4. probing	5. hash
6. table	7. division	8. Separate chaining	9. Chaining	10. Coalesced

Q. III State True or False:

1. The process of mapping kays to appropriate slots in a hash table is known as hashing.
2. In mid-square method we first square the item, and then extract some portion of the resulting digits.
3. The hash table is depending upon the remainder of division.
4. In open addressing, all elements are stored in the hash table itself.
5. The idea of separate chaining is to make each cell of hash table point to a linked list of records that have same hash function value.
6. open addressing is a method for handling collisions.
7. In linear probing, we linearly probe for next slot.
8. Double hashing uses the idea of applying a second hash function to key when a collision occurs.
9. A hash function is a data structure that maps keys to values.
10. Quadratic probing operates by taking the original hash index and adding successive values of an arbitrary quadratic polynomial until an open slot is found.
11. Coalesced hashing is a collision avoidance technique when there is a fixed sized data.

Answers

1. (T)	2. (T)	3. (F)	4. (T)	5. (T)	6. (T)	7. (T)	8. (T)	9. (F)	10. (T)	11. (T)

Q. IV Answer the following Questions:

(A) Short Answer Questions:

1. What is hashing?
2. What is hash function?
3. Define bucket.
4. List steps for mid square method.
5. What is collision?
6. List techniques for collision resolution.
7. Define rehashing?
8. What is linear probing?
9. Define quadratic probing.
10. Define separate chaining?

(B) Long Answer Questions:

1. Define hashing. State need for hashing. Also list advantages of hashing.
2. What is chaining? Explain with diagram.
3. With the help of example describe separate chaining?

4. Describe quadratic probing with example.
5. With the help of diagram describe concept of hashing.
6. What is hash table? Explain in detail.
7. Write a short note on: Linear probing.
8. Compare separate chaining and open addressing?
9. What is rehashing? Describe with diagrammatically.
10. Differentiate between hashing and rehashing?
11. With the help of diagram describe hash function.
12. Describe coalesced chaining with example.

■■■