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.

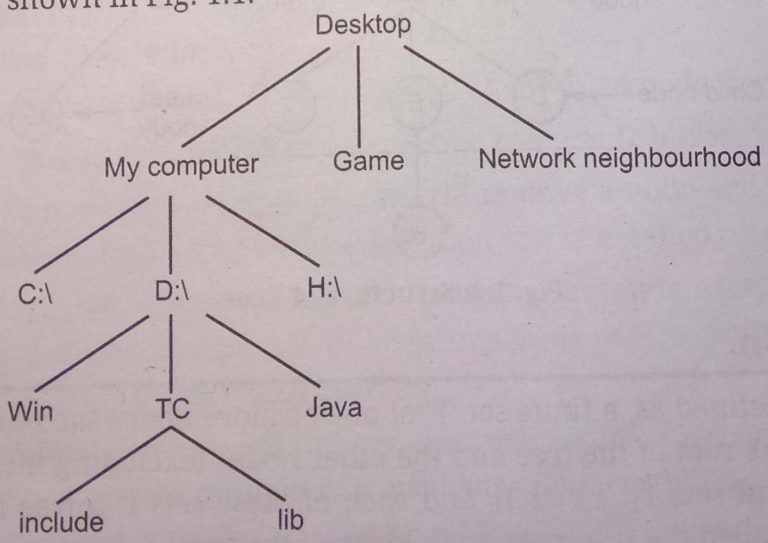


Fig. 1.1: Tree Representation of Directory Structure

- 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.

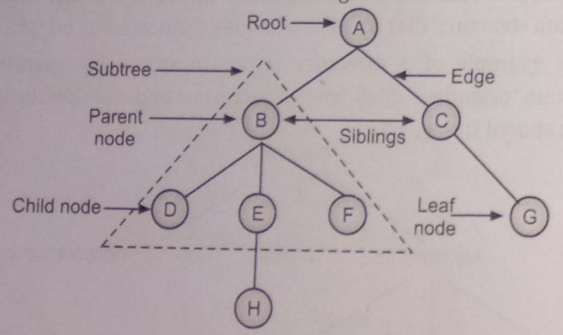


Fig. 1.2: Structure of Tree

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.

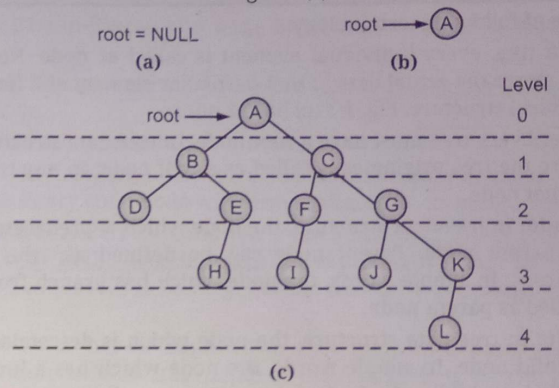


Fig. 1.3: Examples of Tree

- 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.

1.1.2 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.

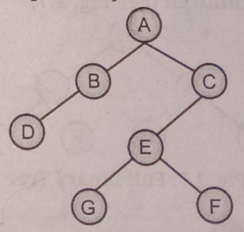
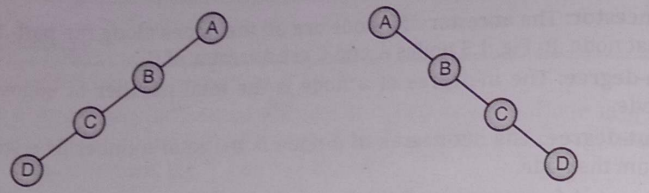


Fig. 1.4: Binary Tree

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.

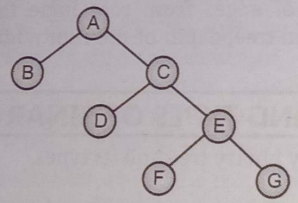


Fig. 1.6: Strictly Binary Tree of Height 3

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).

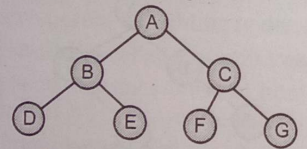


Fig. 1.7: Full Binary Tree

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.

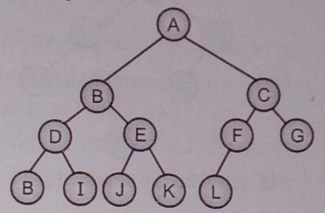


Fig. 1.8: Complete Binary Tree

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.

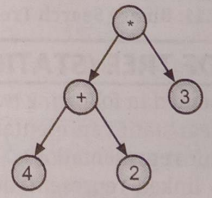


Fig. 1.9

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.

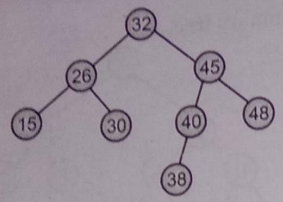
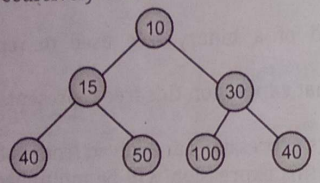


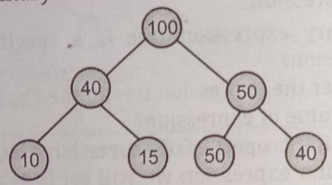
Fig. 1.10: Binary Search Tree

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 i, $1 \le i \le n$
  (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.

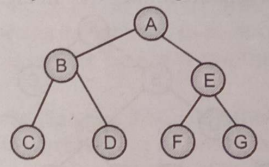


Fig. 1.12

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

1	2	3	4	5	6	7	8	9
A	B	E	C	D	F	G	–	–

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.

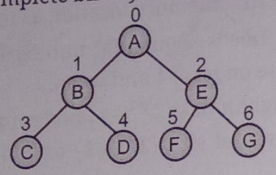


Fig. 1.14

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.

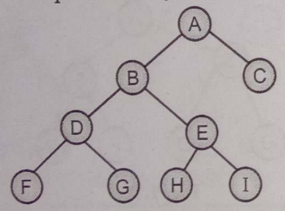


Fig. 1.15

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 = $2 \times 5 = 10$ i.e. H.

   Since, E is the 5^{th} node of tree.

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

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

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

Example 3: Consider the example of skewed tree.

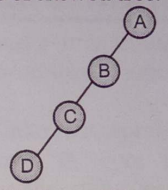


Fig. 1.16: Skewed Binary Tree

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.

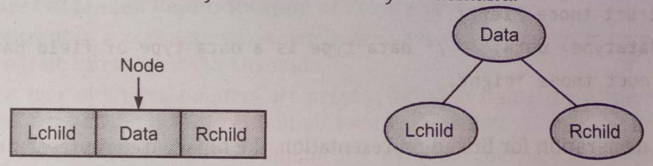
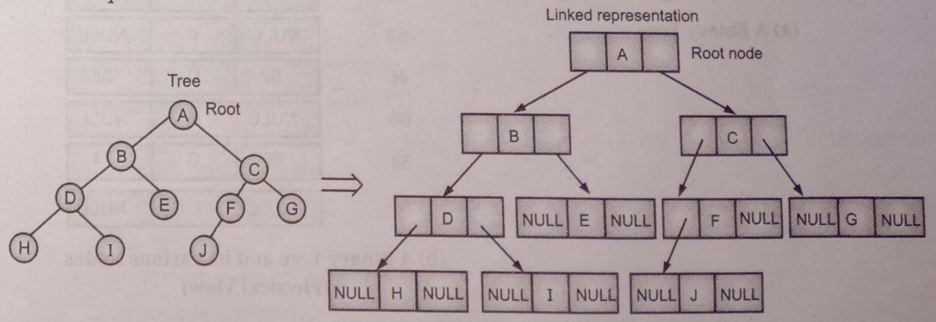
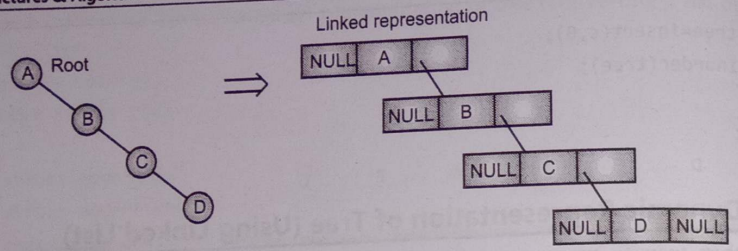


Fig. 1.17: Fields of a Node of a Binary Tree in Linked Representation

- 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)

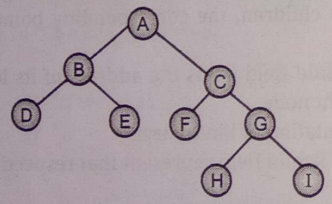
Fig. 1.18: Linked Representation of Binary Tree

- 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;
};
```

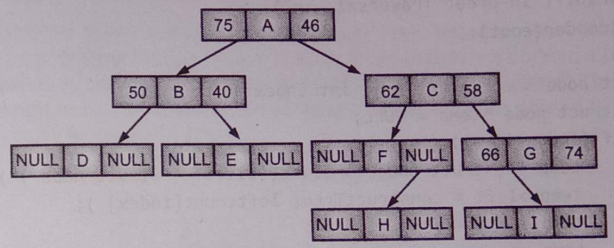
- Using this declaration for linked representation, the binary tree representation can be logically viewed as in Fig. 1.19 (c). In Fig. 1.19 (b) physical representation shows the memory allocation of nodes.



(a) A Binary Tree

Address	node		
	left	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.

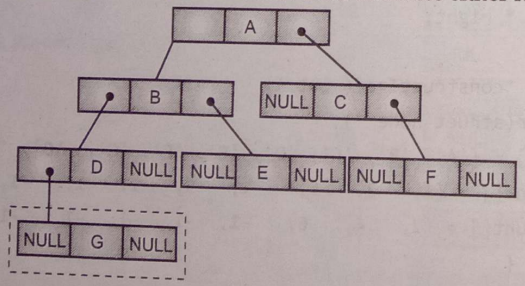


Fig. 1.20: Insertion of Node G

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.

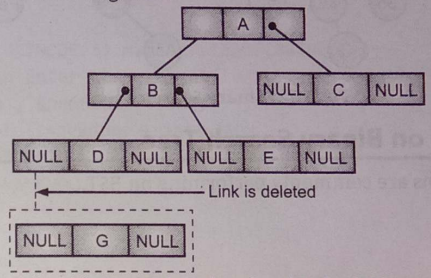


Fig. 1.21: Node with Data G is Deleted and Left of D becomes NULL

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 (BST)

- 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.

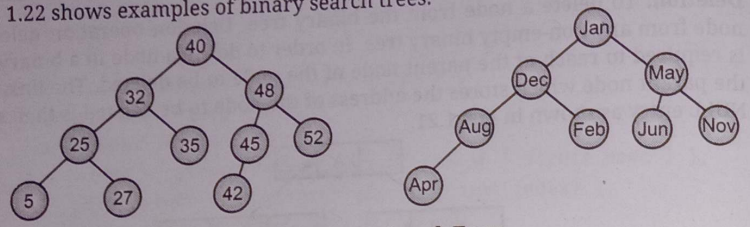


Fig. 1.22: Binary Search Trees

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.

1.4.1.1 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);
   }
```

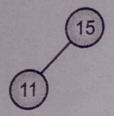
Example: Construct Binary Search Tree (BST) for the following elements:

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

1.

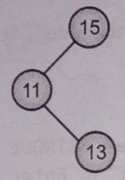
Key = 15

2.



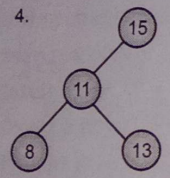
Key < 15
Key = 11

3.



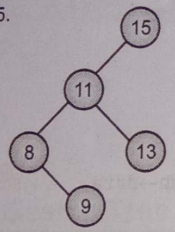
Key = 13

4.



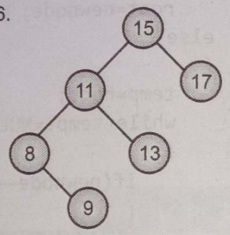
Key = 8

5.



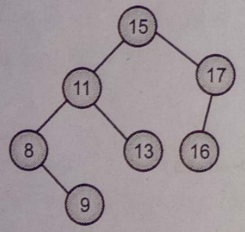
Key = 9

6.



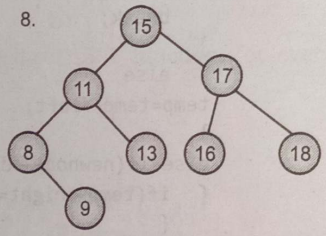
Key = 17

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;
};
```



```
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?
```

1.4.1.2 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.

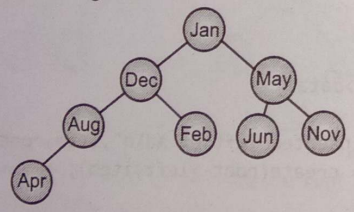


Fig. 1.23: Binary Search Tree

- 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);
}
```

1.4.1.3 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'.

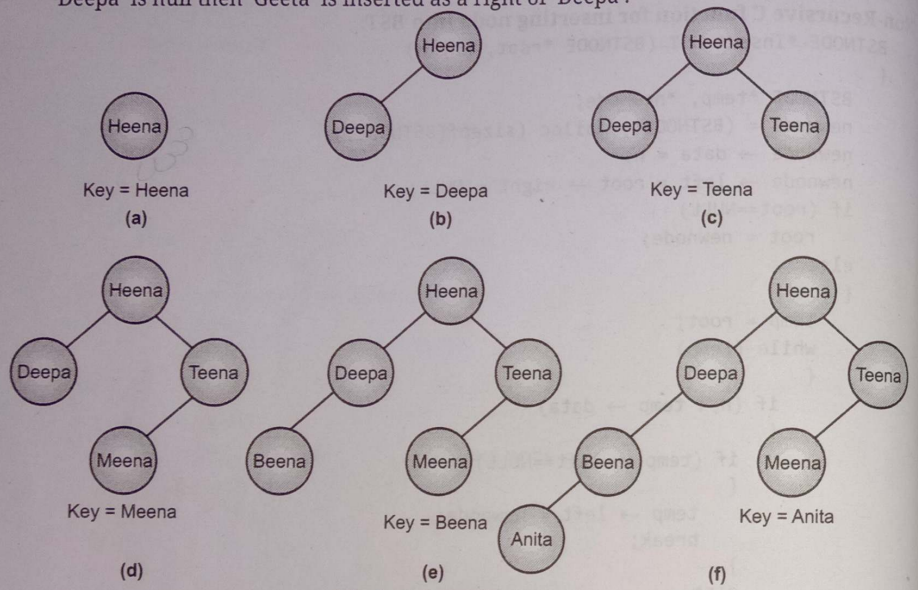


Fig. 1.24: Insertion in BST

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.

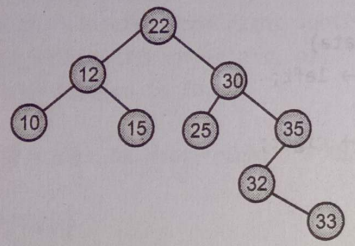


Fig. 1.25: Binary Search Tree

After deletion of leaf node with data 33.

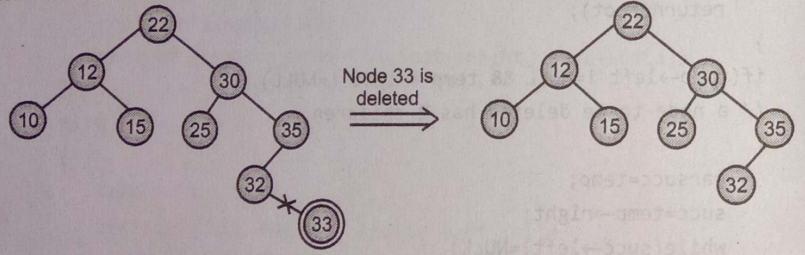


Fig. 1.26: Deletion of node 33

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.

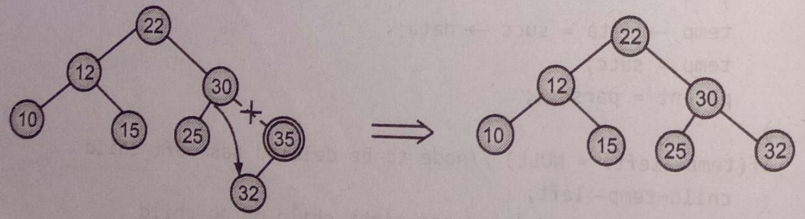


Fig. 1.27: Deletion of Node 35

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.

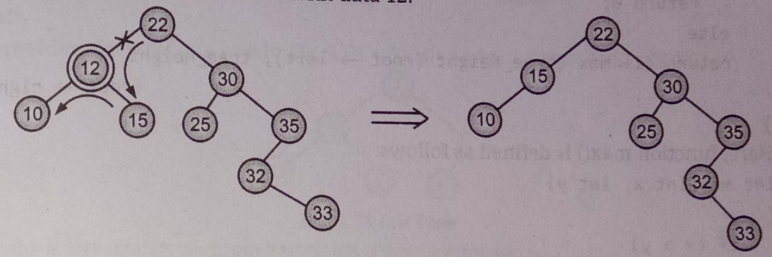


Fig. 1.28: Deletion of Node 12

Pictorial representation is shown in Fig. 1.29.

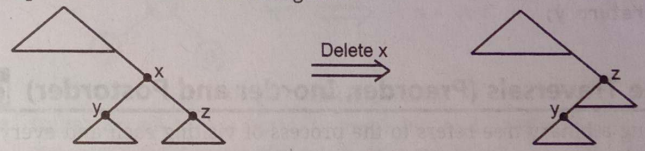


Fig. 1.29: Pictorial Representation

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.

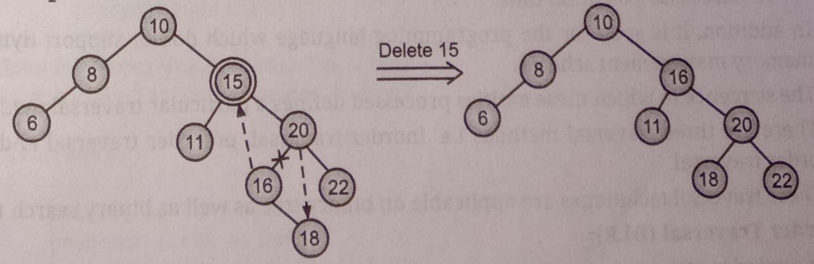


Fig. 1.30: Deletion of Node 15

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 child. 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.

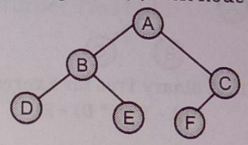


Fig. 1.31 (a): Tree

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

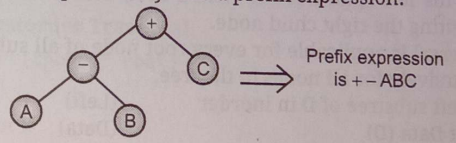


Fig. 1.31 (b): Prefix Expression

- In preorder traversal, we visit a node, traverse left and continue again. When we cannot continue, move right and begin again or move back, until we can move right 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.

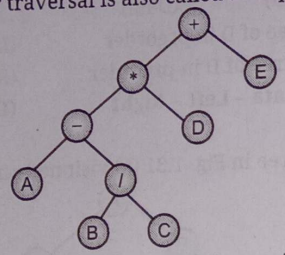


Fig. 1.32: Binary Tree for Expression ((A – B/C) * D) + E

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)

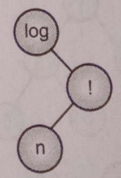


Fig. 1.33

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

(b)

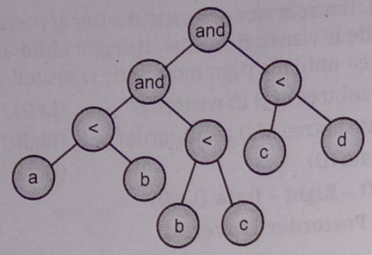


Fig. 1.34

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)

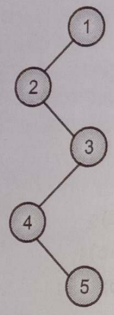


Fig. 1.35

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

(d)

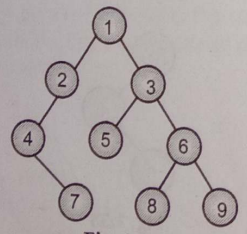


Fig. 1.36

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 tree shown in Fig. 1.37.

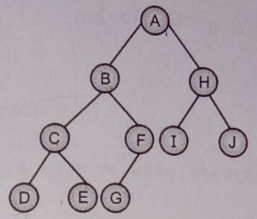


Fig. 1.37

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.

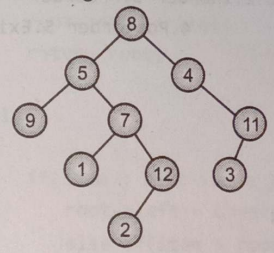


Fig. 1.38

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

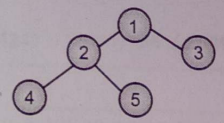


Fig. 1.39

1.4.5 Count Total Nodes of a Binary Tree [April 19]

- To count total nodes of a binary tree, we traverse a tree. We can use any of the three 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));
}
```

1.4.6 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));
  }
```

1.4.7 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));
}
```

1.4.8 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.*

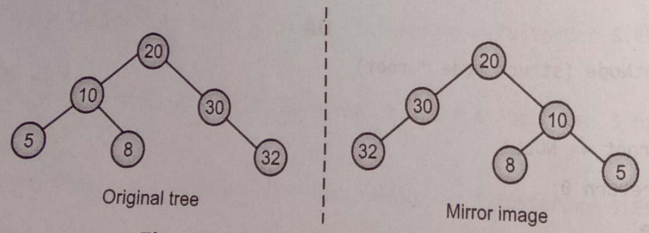


Fig. 1.40: Mirror image of a Binary Tree

- *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;
}
```

1.5 APPLICATIONS [April 16]

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

1.5.1 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.

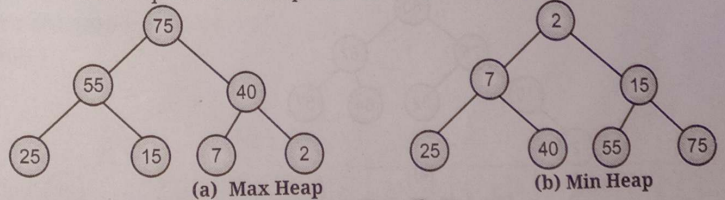


Fig. 1.41: Heap Trees

- 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.

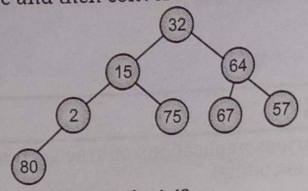


Fig. 1.42

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

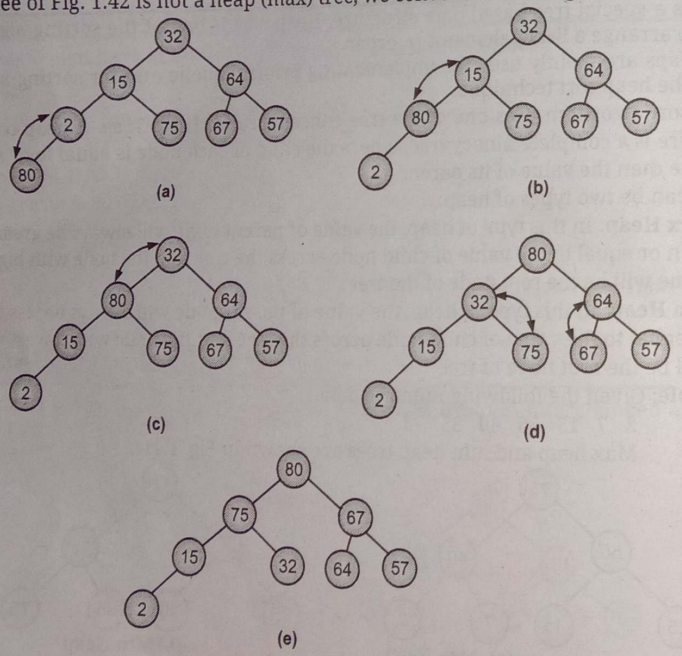


Fig. 1.43: Heap Tree (Max)

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*/
```

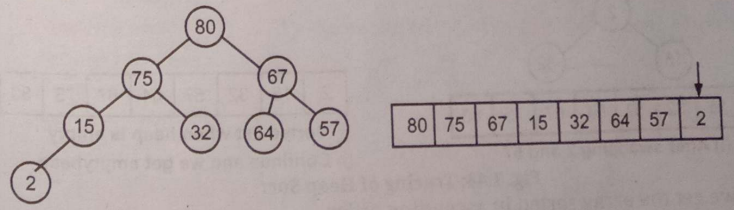
Step 5 : while (j < i) do

```
{
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

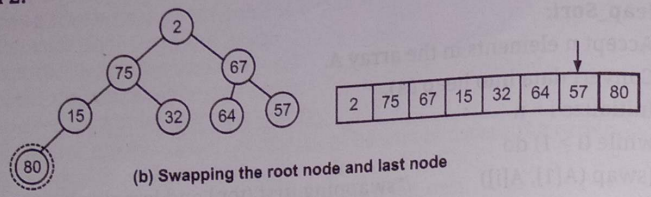
Consider the heap tree of Fig. 1.44.

Iteration 1:



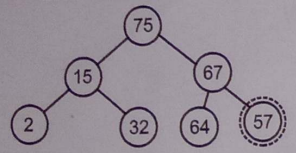
(a)

Iteration 2:

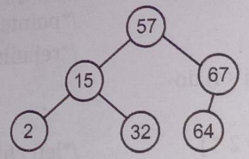


(b) Swapping the root node and last node

Iteration 3:

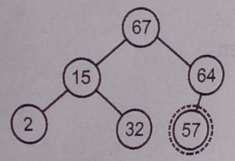


(c) Rebuilt the heap tree

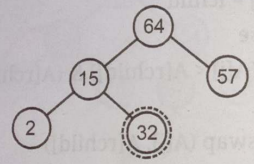


(d) Swapping 57 and root node

67	15	64	2	32	57	75	80

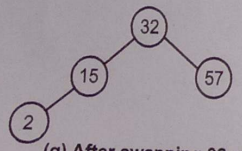


(e) Rebuilt

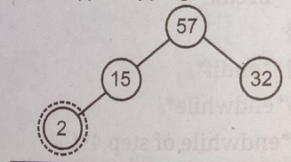


64	15	57	2	32	67	75	80

(f) Swapping and Rebuilt

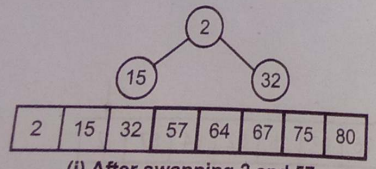


(g) After swapping 32 and 64



57	15	32	2	64	67	75	80

(h) Rebuilt



(i) After swapping 2 and 57

2	15	32	57	64	67	75	80

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)
```

1.5.2 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 maximizes or minimizes the objective function.
- A greedy algorithm proceeds step–by-step, by considering one input at a time. At each stage, the decision is made regarding whether a particular input (say x) chosen gives an optimal solution or not.
  - Our choice of selecting input x is being guided by the selection function (say select).
  - If the inclusion of x gives an optimal solution, then this input x is added into the partial solution set.
  - On the other hand, if the inclusion of that input x results in an infeasible solution, 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 way is 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 check whether x can be included into the solution set S or not. If yes, then solution set. If no, then this input x is discarded and not added to the partial solution set S. Initially S is set to empty.
  3. Continue until S is filled up (i.e. optimal solution found) or C is exhausted whichever is earlier.

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

1.5.3 Huffman Encoding (Implementation using Priority Queue)

- Huffman encoding developed by David Huffman. Data can be encoded efficiently using Huffman codes.
- Huffman code is a data compression algorithm which uses the greedy technique for its implementation. The algorithm is based on the frequency of the characters appearing in a file.
- Huffman code is a widely used and beneficial technique for compressing data. Huffman's greedy algorithm uses a table of the frequencies of occurrences of each character to build up an optimal way of representing each character as a binary 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.

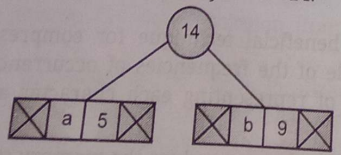
Steps to build Huffman Tree:

- 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.

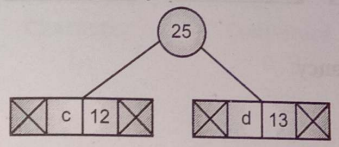
Step 2 : Extract two minimum frequency nodes from min heap. Add a new internal node with frequency 5 + 9 = 14.



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

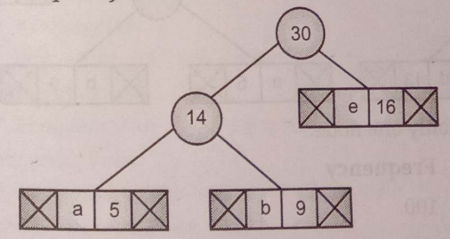
Step 3 : Extract two minimum frequency nodes from heap. Add a new internal node with frequency 12 + 13 = 25.



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

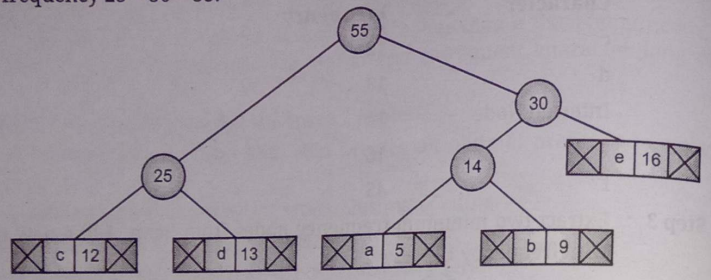
Step 4 : Extract two minimum frequency nodes. Add a new internal node with frequency 14 + 16 = 30.



Now min heap contains 3 nodes.

Character	Frequency
Internal Node	25
Internal Node	30
f	45

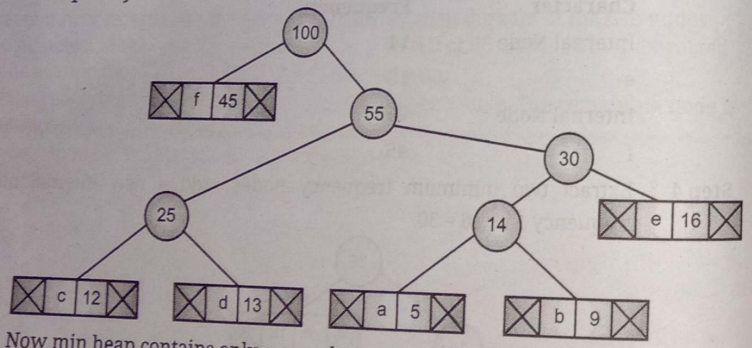
Step 5 : Extract two minimum frequency nodes. Add a new internal node with frequency 25 + 30 = 55.



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.



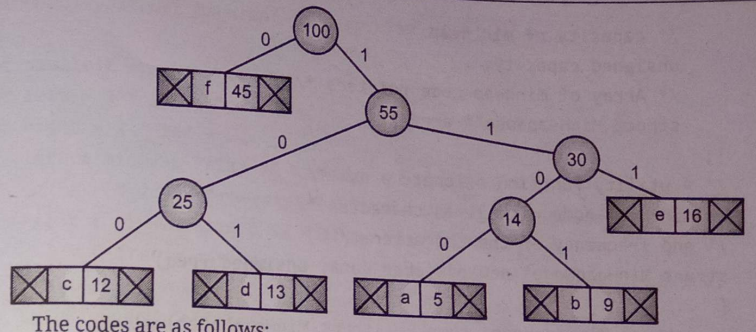
Now min heap contains only one node.

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:

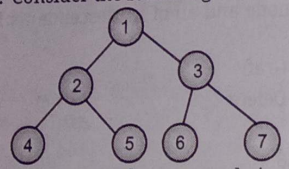
```
f: 0
c: 100
d: 101
a: 1100
b: 1101
e: 111
```


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 leaf nodes are at same level.
11. In array representation of binary tree, we use a _____ dimensional array to 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 _____ than the value of the root.
14. In _____ heap all parent node's values are greater than or equal to children node's 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 leaf 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 a 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 most 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:

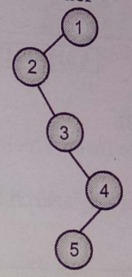
(A) Short Answer Questions:

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? → File system database network routing artifical intelligence,
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.

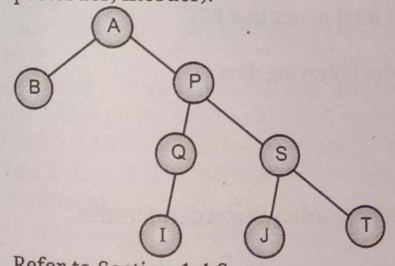
2. Write a recursive 'C' function to insert an element in a binary search tree. [5 M]

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: 40, 70, 60, 50, 65, 20, 25 [3 M]

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: 15, 11, 13, 8, 9, 18, 16 [4 M]

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.

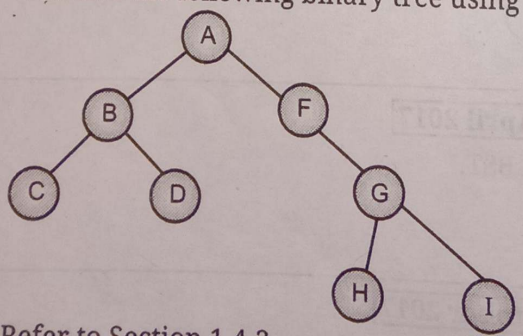
4. Write the steps for creating a BST for the following data: 22, 13, 4, 6, 25, 23, 20, 18, 7, 27. [4 M]

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.

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

■■■