CHAPTER 4
TREES

4.1 INTRODUCTION

In this chapter we will explore a new type of data structure known as non-linear i.e. trees. It generally represents a hierarchical structure of data.

1. A tree may be defined as
   - a graph without cycles.
2. A tree is **nonlinear data structure.**
3. It represents a **hierarchical structure** of data.

4.2 BINARY TREE

Binary Tree T is a special tree with the following properties.

1. T has a finite set of elements called nodes.
2. T may be empty or
3. T contains a special node called the **Root (R)** and the other nodes form part of one of the two disjoint binary subtrees **T**1 and **T2.**

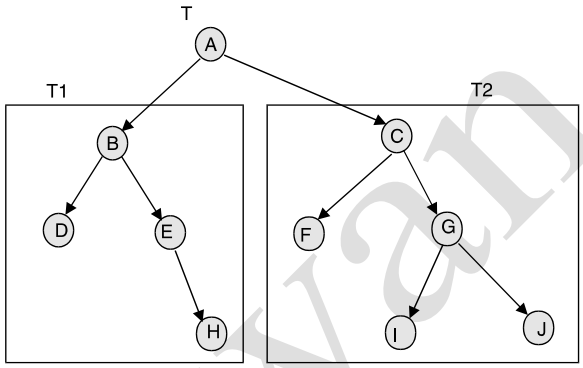


Figure 4.1

Lets have a look on the following example of trees and find out the definitions and properties of binary trees.

Definitions and Properties of Binary Trees:

1. Root of Trees :

The root of tree is the origin of tree i.e. from where the tree starts. Node A is the root of the tree T shown in figure 4.1.

2. Left Subtree and Right Subtree :

The left and right subtree is nothing but the left and right nodes of any tree. As in case of figure we have the following left and right subtree.

Node	Left Subtree	Right Subtree
A	B	C
B	D	E
C	F	G
D	NULL	NULL
E	NULL	H
F	NULL	NULL
G	I	J
H	NULL	NULL
I	NULL	NULL
J	NULL	NULL

3. Successors or Children of Node:

Left and right subtrees of trees are also know as Successors or Children of Node. As we can see from the figure 4.1 A is having 2 Successors or Children as B and C but H is having no successors or children.

4. Terminal Node:

A node is said to be terminal node if it has no children. In figure 4.1 the nodes D, F, H, I and J are the terminal nodes.

5. Number of Children:

A tree can have either 0, 1 or 2 children. Node A is having 2 children while J is having 0 children.

6. Parent:

Node A is said to be the parent of B and C. Similarly E is a parent of H and G is a parent of I and J.

7. Sibling:

The nodes are said to be sibling if they have same parent. As B, C are said to be siblings of node A as they are from same parent i.e. node A.

8. Level of Node:

The level of node in a binary tree is determined by setting the level of Root as 1. The children nodes are at level 1 + 1 if the parent node is at level 1.

Node	Level
A	1
B and C	2
D, E, F and G	3
I and J	4

The maximum number of nodes at level L of binary tree can be

$2^{L}-1, \quad L >= 1.$

9. Height or Depth of Tree:

The height of the binary tree is equal to the number of levels in the tree. The height of the binary tree in figure 4.1 is 4

The maximum number of nodes in a binary tree of depth n can be

$2^{n}-1, \quad n >= 1$

10. Complete Tree:

A tree is said to be complete if it has maximum number of nodes at each level except the last level and if all the nodes at the last level appear as far left as possible.

A complete tree of depth K has $2^{k}-1$ nodes.

The depth of a complete tree with n nodes is given by

$\mathbf{D_n = [\log n + 1]}$

The following is the complete tree.

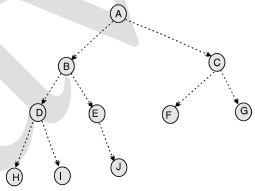


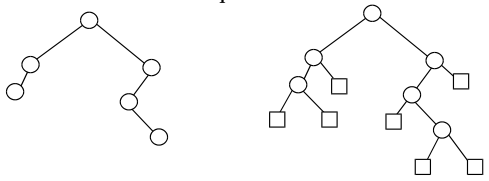
Figure 4.2

Example:

Depth of the binary tree with 10,00,000 nodes is 21 Advantageous for searching, as with 10,00,000 we only require 21 comparisons for any search.

4.3 2-TREE OR EXTENDED BINARY TREES

A binary tree T is said to be a 2-tree or an extended binary tree if each node N has either 0 or 2 children. In such a case, the nodes with 2 children are called internal nodes, and the nodes with 0 children are called external nodes. Sometimes the nodes are distinguished in diagrams by using circles for internal nodes and squares for external nodes.



(a) Binary Tree (b) Extended 2-tree

Figure 4.3 Converting a binary tree into a 2-tree

Similar tree:

Two trees are said to be similar it they have similar structure. For examples T1 and T2 are similar trees. But T3 and T4 are different trees because there is a distinction between left or right successor.

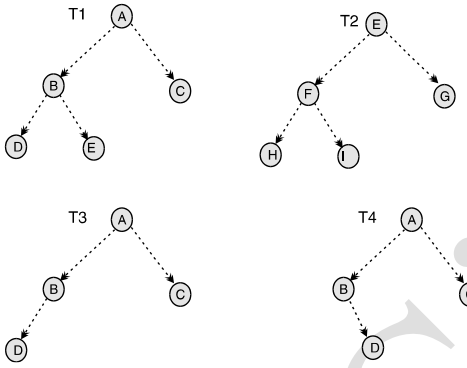


Figure 4.4

Equivalent Tree:

Two trees are said to be equivalent if they are exactly of similar structure and similar data in their corresponding nodes T1 and T2 are equivalent tree.

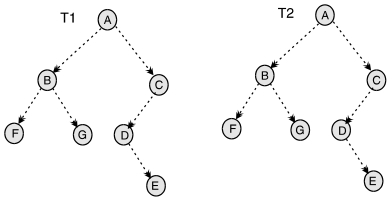


Figure 4.5

4.4 REPRESENTATION OF BINARY TREE

The Binary tree can be represented in two ways :

1. Linked List Representation.
2. Sequential Representation.

1. Link List Representation:

Linked List can represent binary tree as shown below.

In link list each node has the following fields.

Data **:** May be record or just a basic data element.

Left **:** Points to the **Left Tree** or **Left Child** of the node.

Right **:** Points to the Right Tree or Right Child of the node.

The Start pointer to the tree is a pointer to root of the tree.

Let's represents the figure 4.1 by using link list.

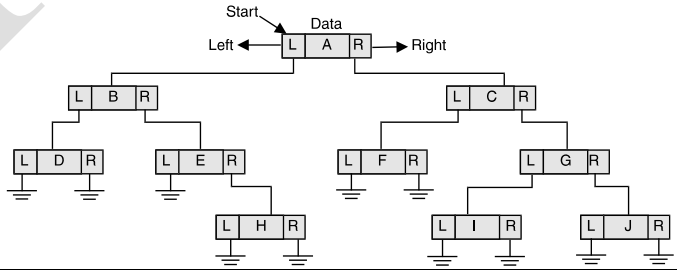


Figure 4.6

2. Sequential Representation of Binary Tree:

The tree can also be represented using linear array. If a binary tree is complete or nearly complete then the tree can be efficiently represented using a linear array. Let's say array TREE [ ] represents a tree.

The entries in the array TREE [ ] is done according to the following rules.

a) The root R of the binary is stored in TREE [0]

b) If a node occupies TREE [K] then the left child is stored in TREE [2*K+1] and right child is located in TREE [2*(k+1)].

c) A NULL in any of the tree location indicates an empty sub tree.

Traversing Binary Trees

There are three standard ways of traversing a binary tree T with root R. These three algorithms, called preorder, inorder and postorder, are as follows :

Preorder (1) Process the root R.
(2) Traverse the left subtree of R in preorder.
(3) Traverse the right subtree of R in preorder.

Inorder (1) Traverse the left subtree of R in inroder.
(2) Process the root R.
(3) Traverse the right subtree of R in inorder.

Postorder (1) Traverse the left subtree of R in postorder.
(2) Traverse the right subtree of R in postorder.
(3) Process the root R.

Example :

Consider the binary tree T in figure.4.7. Observe that A is the root, that is its left subtree L_T consists of nodes B, D and E and that its right subtree R_T consists of nodes C and F.

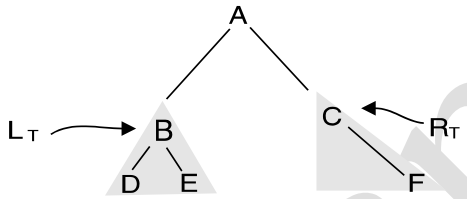


Figure 4.7

(a) The preorder traversal of T processes A, traverses L_T and traverse R_T. However, the preorder traversal of L_T processes the root B and then D and E, and the preorder traversal of R_T processes the root C and then F. Hence ABDECF is the preorder traversal of T.

(b) The inorder traversal of T travereses L_T, processes A and traverses R_T. However, the inorder traversal of L_T processes D, B and then E, and the inorder traversal of R_T processes C and then F. Hence DBEACF is the inorder traversal of T.

(c) The postorder traversal of T traverse L_T, traverse R_T and processes A. However, the postorder traversal of L_T process D,E and then B, and the postorder traversal of R_T processes F and then C. Accordingly, DEBFCA is the postorder traversal of T.

Traversal Algorithms Using Stacks

Suppose a binary tree T is maintained in memory by some linked representation.

This section discusses the implementation of the three standard traversals ofT. which were defined recursively in the last section, by means of nonrecursive procedures using stacks, we discuss the three traversals separately.

4.5 PREORDER TRAVERSAL

Recursive Preorder Traversing:

```
Algorithm Preorder (root)   //Recursive
      if (root ! = NULL)
Step 1 :   {      print date ;
Step 2 :          preorder (root->lchild);
Step 3 :          preorder (root->rchild) ;
        }
Step 4 :          end Preorder
```

Non-Recursive Preorder Traversing:

Algorithm Non-Recursive Preorder Traversing:

```
Algorithm Preorder (root)        //Non Recursive or Iterative
/*
      binary tree T is in memory and array stack is used to temporarily hold address of nodes
 */

Step 1 -  [Initialize top of stack with null and initialize temporary pointer to node type is ptr]
Top = 0;
Stack[Top] = NULL;
ptr = root;

Step 2 :   repeat steps 3 to 5 while ptr! = NULL

Step 3 :    visit ptr->Info

Step 4 :   [Test existence of right child]
if(ptr ->rchild! = NULL)
{
Top = Top + 1;
Stack[Top] = ptr->rchild;
}

Step 5:    [Test for left child]
if(ptr->lchild! = NULL)
       ptr=ptr ->lchild
else
  [Pop from stack]
        ptr = stack[Top];
       Top = Top-1;
[End of loop in Step2.]
 Step 6:    end Preorder.
```

Explanation: Initially push NULL onto STACK and then set PTR: = ROOT. Then repeat the following steps until PTR = NULL or, equivalently, while PTR! = NULL.

(a) Proceed down the left-most path rooted at PTR, processing each node N on the path and pushing each right child R (N), if any, onto STACK. The traversing ends after a node N with no left child L (N) is processed. (Thus PTR is updated using the assignment PTR: = LEFT [PTR], and the traversing stops when LEFT [PTR] = NULL.)

(b) [Backtracking.] Pop and assign to PTR the top element on STACK. If PTR! = NULL. Then return to Step (a); otherwise Exit.

(We note that the initial element NULL on STACK is used as a sentinel.)

4.6 INORDER TRAVERSAL

Recursive Inorder Traversing :

```
Algorithm inorder (root)   //Recursive
      if (root ! = NULL)
Step 1 : {      inorder (root-> lchild);
Step 2 :         print date ;
Step 3 :         inorder (root->rchild) ;
          }
```

Step 4 : end inorder

Non-Recursive Inorder Traversing

Algorithm Non-Recursive Inorder Traversing

```
Algorithm Inorder (root)                    // Non Recursive

Step 1 : [Initially push NULL on stack and initialize ptr]
         Top = 0;.
         Stack [Top] = NULL;
         ptr = root ;
Step 2 : [In order first we visit left child ]
         while (ptr ! = NULL)
         {
                  Top = Top + 1 ;
                  stack [Top] = ptr ;
                  ptr = ptr ->lchild ;
         }
Step 3 : ptr = stack [Top] ;
         Top = Top - 1;
Step 4 : repeat steps, 5 to 7 while (ptr ! = NULL)
Step 5 : print ptr -> Info ;
Step 6 : if (ptr->rchild ! = NULL)
             ptr = ptr-> rchild ;
             goto step 2 ;
Step 7:  ptr = stack [Top] ;
         Top = Top -1 ;
         [End of while loop.]
Step 8 : end Inorder
```

Explanation:

Initially push NULL onto STACK (for a sentinel) and then set PTR: =ROOT. Then repeat the following steps until NULL is popped from STACK.

(a) Proceed down the left-most path rooted at PTR, pushing each node N onto STACK and stopping when a node N with no left child is pushed onto STACK.

(b) [Backtracking] Pop and process the nodes on STACK. If NULL is popped, then Exit. If a node N with a right child R (N) is processed, set PTR = R (N) (by assigning PTR: =RIGHT [PTR]) and return to step (a).
We emphasize that a node N is processed only when it is popped from STACK.

4.7 POSTORDER TRAVERSAL

Recursive Postorder Traversing:

```
Algorithm Postorder (root)  // Recursive
Step 1 :    if (root ! = NULL)
            {
                call Postorder (root->lchild) ;
                call Postorder (root->rchild);
                print data ;
            }
Step 2 :    End Postorder
```

Non Recursive Postorder Traversing:

Algorithm Non Recursive Postorder Traversing:

```
Algorithm Postorder (root)          //Non Recursive
Step 1 :    [Push NULL to stack and initialize ptr]
            Top = 0;
            stack [Top] = NULL ;
            ptr = root ;
Step 2 :    repeat steps 3 to 5 while (ptr ! = NULL)
Step 3 :    Top = Top+1;
            stack[Top] = ptr ;
Step 4:     If (ptr-> rchild ! =NULL)
                Top = Top + 1
                stack [Top] = - ptr-> rchild;
Step 5 :    ptr = ptr-> lchild
Step 6 :    [pop from stack]
                ptr = stack [Top];
                Top = Top -1;
Step 7 :    while (ptr > 0)
            {
                Visit ptr -> Info;
                ptr = stack[Top];
                Top = Top-1
            }
Step 8 :    if ptr < 0 then
            {
                ptr = - ptr ;
                go to step 2 ;
            }
Step 9:     end Postorder
```

Explanation:

Initially push NULL onto STACK (as a sentinel) and then set PTR: = ROOT. The repeat the following steps until NULL is popped from STACK.

(a) Proceed down the left-most path rooted at PTR. At each node N of the path, push N onto STACK and, if N has right child R(N), push - R(N) onto STACK.

(b) [Backtracking.] Pop and process positive nodes on STACK. If NULL is popped, then Exit. If a negative node is popped, that is, if PTR = - N for some node N, set PTR = N (by assigning PTR: = -PTR) and return to Step (a).

We emphasize that a node N is processed only when it is popped from STACK and it is positive.

4.8 THREADS: INORDER THREADING

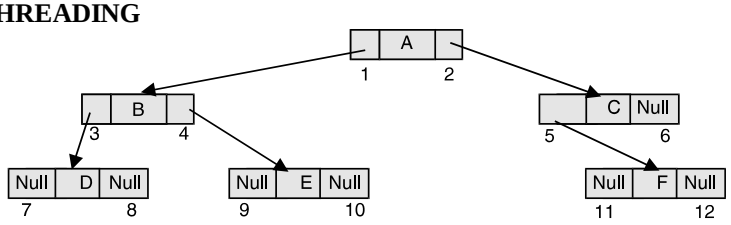


Figure 4.8

Let us consider a binary tree. Shown in the figure.4.8.

In the above binary tree, there are 7 null pointers (number 6 through 12) and 5 actual pointers. In all there are 12 pointers. We can generalize it that for any binary tree with n nodes there will be (n+1) null pointers and 2n total pointers. The objective here is to make effective use of these null pointers. **A J.Perlis and C.Thornton** jointly proposed an idea to make use of these null pointers. According to this idea we are going to replace all the null pointers by the appropriate pointer values called **threads**. A binary tree is threaded according to a particular traversal method. And ,the null pointer replacement scheme is defined as follows.

If the left link (child) of a node p is normally equal to NULL, then this link is replaced by a pointer to the node which immediately precedes node p in inorder traversal. And similarly, if the right link (rchild) of a node p is normally equal to NULL, then this link is replaced by a pointer to its inorder successor node. A threaded binary tree is shown in figure.4.9

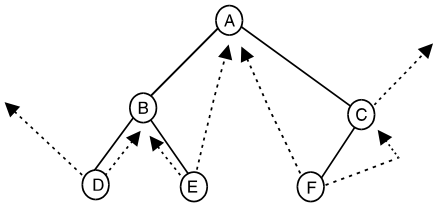


Figure.4.9

Threaded binary tree corresponding to figure.4.8

The inorder traversal of a binary tree shown in figure 4.9

D B E A F C

So,

rchild of D is made to point to B.
lchild of E is made to point to B.
rchild of E is made to point to A.
lchild of F is made to point to A.
rchild of F is made to point to C.

Two dangling pointers such as Ichild (D) and rchild (C) are made to point to a header node as shown below:

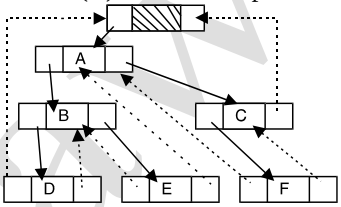


Figure 4.10

In the memory representation of a threaded binary tree, it is necessary to distinguish between a normal pointer and a thread. Therefore, we have an alternate node representation for a threaded binary tree which contains five fields as shown in figure.4.11

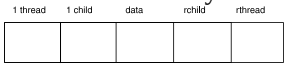


Figure 4.11

For any node p, in a threaded binary tree.

Ithread (p) = 1 indicates Ichild (p) is a normal pointer.

Ithread (p) = 0 indicates Ichild (p) is a thread.

rthread (p) = 1 indicates rchild (p) is a normal pointer.

rthread (p) = 0 incidates rchild (p) is a thread.

Now, the threaded binary tree shown in figure.4.12

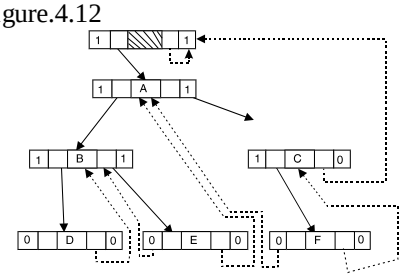
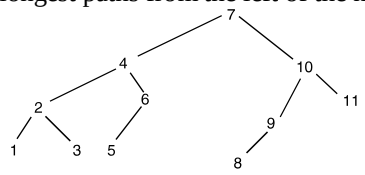


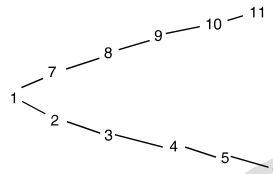
Figure 4.12

New representation of threaded binary tree.

4.9 HEIGHT BALANCED TREES OR AVL TREE

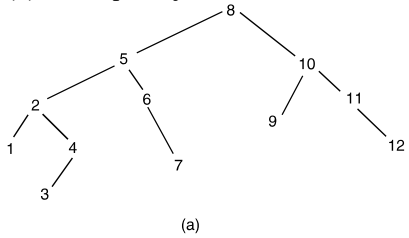
A binary tree of height h is completely balanced if all leaves occur at nodes of level h or h -1 and if all nodes at levels lower than h - 1 have two children. According to this definition, the tree in figure 4.13 (a) is balanced, because all leaves occur at level 3 considering at level 1 and all node's at levels 1 and 2 have two children. Intuitively we might consider a tree to be well balanced if, for each node, the longest paths from the left of the node are about the same length as the longest paths on the right.





(a) A nearly full binary tree Figure 4.13 (b) A degenerate binary tree

More precisely, a tree is height balanced if, for each node in the tree, the height of the left subtree differs from the heights, of the right subtree by no more than 1. The tree in figure 4.14 (a) height balanced, but it is not completely balanced. On the other hand, the tree in Fig. 4.14 (b) is completely balanced tree.



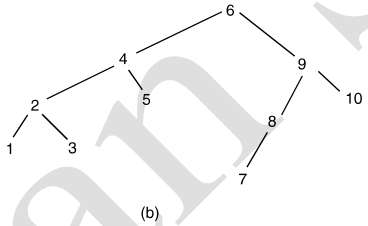


Figure 4.14

An almost height balanced tree is called an AVL tree after the Russian mathematician **G.M.Adelson-Velskii and E.M.Lendis**, who first defined and studied this form of a tree. **AVL Tree** may or may not be perfectly balanced.

Let us determine how many nodes might be there in a balanced tree of height h.

The, root will be the only node at level 1 ;

Each subsequent level will be as full as possible i.e., 2 nodes at level 2, 4 nodes at level 3 and so on, i.e. in general there will be 2^{1-1} nodes at level 1. Therefore the number of nodes from level 1 through level h-1 will be

$$1+2+2^2+2^3+.....2^{h-2} = 2^{h-1}-1$$

The number of nodes at level h may range from a single node to a maximum of 2 h-1 nodes. Therefore the total number of nodes n of the tree may range of (2h-1-1+1) or 2h-1 to 2h-1.

Building Height Balanced Tree :

Each node of an AVL tree has the property that the height of the left subtree is either on more, equal, or one less than the height of the right subtree. We may define a balance factor (BF) as

BF = (Height of Left subtree - Height of Right subtree)

Further

If two subtree are of same height BF =0

If right subtree is higher BF = +1

If left subtree is higher BF = -1

AVL Tree Balance Requirements

The complexity of an AVL Tree comes from the balance requirements it enforces on each node. A node is only allowed to possess one of three possible states (or balance factors):

Left-High (balance factor -1)

The left-sub tree is one level taller than the right-sub tree

Balanced (balance factor 0)

The left and right sub-trees are both the same heights

Right-High (balance factor +1)

The right sub-tree is one level taller than the left-sub tree.

If the balance of a node becomes -2 (it was left high and a level was lost from the left sub-tree) or +2 (it was right high and a level was lost from the right sub-tree) it will require re-balancing. This is achieved by performing a rotation about this node (see the section below on rotations).

Inserting in an AVL Tree

Nodes are initially inserted into AVL Trees in the same manner as an ordinary binary search tree (that is, they are always inserted as leaf nodes). After insertion, however, the insertion algorithm for an AVL Tree travels back along the path it took to find the point of insertion, and checks the balance at each node on the path. If a node is found that is unbalanced (that is, it has a balance factor of either -2 or +2), then a rotation is performed (see the section below on rotations) based on the inserted nodes position relative to the node being examined (the unbalanced node).

NB. There will only ever be at most one rotation required after an insert operation.

Deleting from an AVL Tree

The deletion algorithm for AVL Trees is a little more complex, as there are several extra steps involved in the deletion of a node. If the node is not a leaf node (that is, is has at least one child), then the node must be swapped with either its in-order successor or predecessor (based on availability). Once the node has been swapped we can delete it (and have its parent pick up any children it may have - bear in mind that it will only ever have at most one child). If a deletion node was originally a leaf node, then it can simply be removed.

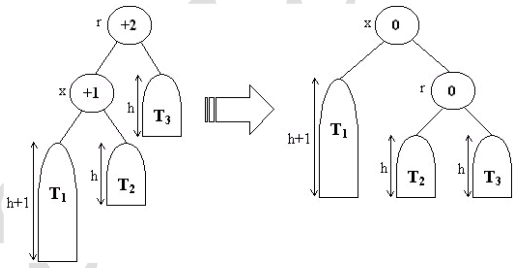
Now, as with the insertion algorithm, we traverse back up the path to the root node, checking the balance of all nodes along the path. If we encounter an unbalanced node we perform an appropriate rotation to balance the node (see the section below on rotations).

NB. Unlike the insertion algorithm, more than one rotation may be required after a delete operation, so in some cases we will have to continue back up the tree after a rotation.

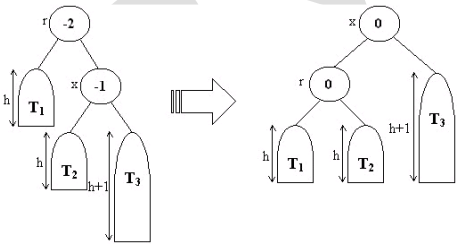
AVL Tree Rotations

As mentioned previously, an AVL Tree and the nodes it contains must meet strict balance requirements to maintain is O (log n) search capabilities. These balance restrictions are maintained using various rotation functions. Below is a diagrammatic overview of the four possible rotations that can be performed on an unbalanced AVL Tree, illustrating the before and after states of an AVL Tree requiring the rotation.

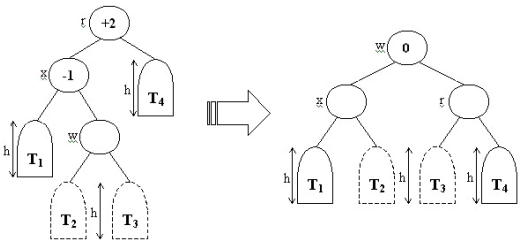
LL Rotation



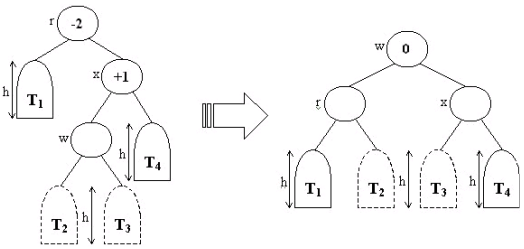
RR Rotation



LR Rotation



RL Rotation



Q.4.1 A Binary tree T has 9 nodes. The inorder and precorder traversals of T yield the following sequences of nodes:
Inorder : E A C K F H D B G
Preorder : F A E K C D H G B
Draw the tree T.

Solution:

The tree T is drawn from its root downward as follows:

(a) The root of T is obtained by choosing the first node in its preorder. Thus F is the root of T.

(b) The left child of the node F is obtained as follows. First use the inorder of T to findthe nodes in the left subtree T_1 of F. Thus T_1 consists of the nodes E,A,C and K. Then the left Child of F is obtained by choosing the first node in the preorder of T_1 (which appears in the preorder of T). Thus A is the left son of F.

(c) Similarly, the right subtree T_2 of F Consists of the nodes H, D, B and G and D is the root of T_2, that is D is the right child of F.

Repeating the above process with each new node, we finally obtain the required tree in Fig.1

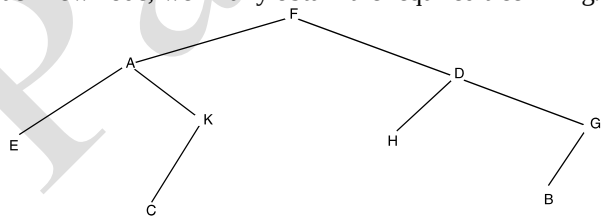


Figure 1

Q.4.2 Consider the algebraic expression E = $(2x+y)(5a-b)^3$

(a) Draw the tree T which corresponds to the expression E.

(b) Find the scope of the exponential operator; i.e., find the subtree rooted at the exponential operator.

(c) Find the prefix Polish expression P which is equivalent to E, and find the preorder of T.

Solution:

(a) Use an arrow ( ) for exponentiation and an asterisk (*) for multiplication to obtain the tree shown in Fig.2

(b) The scope of is the tree shaded in Fig. 2 It corresponds to the sub expression.$(5a-b)^3$

(c) There is no difference between the prefix Polish expression P and the preorder of T. Scan the tree T from the left, as in Fig.2 to obtain
* + * 2 x y - * 5 a b 3

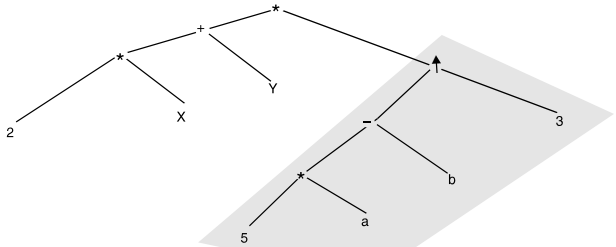


Figure 2

4.10 BINARY SEARCH TREE

Binary Search Tree is a very important and special kind of data structure used. In binary search tree the search, the operations insert and delete are done very easily as compared to other data structures.

Comparison

The **Binary Search Tree** differs from the Sorted Linear Arrays and Linked List in the following ways.

S.No.	Sorted Linear Array	Link Lists	Binary Search Tree
1.	The Searching time is $O(\log_2 n)$ $O(\log_2 n)$.	The Searching time is O (n)	the time taken to search a element in BST is
2.	Difficult to insert and Delete	Insert and Deletion can be done at ease.	Insertion and Deletion can be done at ease.

From the above table one can find out that Binary search Tree is basically the combination of Sorted Linear Array and Link Lists. As binary search tree takes only $O(\log_2 n)$ for searching and also insertion and deletion can be done at ease like in link list. Also inorder traversal of the binary search tree will give as the sorted list of the elements.

A tree is known as Binary Search Tree if each node N of the tree follows the following properties.

1. The value of the node N is greater than every node to its left sub tree.
2. The value of the node N is less than or equal to every node to its right sub tree.

Example

The tree shown in figure 4.15 is the Binary Search Tree. As we can see from the figure every node N is greater than its left sub tree and less than its right sub tree.

Like 12 is greater than 4, hence, to its left and less than 17 which is at its right. Similarly 4 is greater than 3 which is to its left and less than 8 which is to it's right.

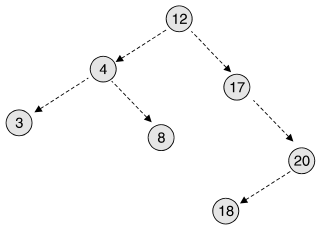


Figure 4.15

4.11 OPERATIONS ON THE BINARY SEARCH TREE

In this section we will discuss some of the basic functions of the Binary Search tree. Like searching and inserting in BST. Searching is just to find out if the data is present in the tree.

Searching

Searching a particular data starts with the root, the data say n is compared with root.

And the following actions are taken.

1. If the data of root is greater than n than we look only the left sub-tree and ignoring completely right sub tree.
2. If the data of the root is lesser then n than we move to the right sub-tree.

Let's look at the following binary search tree (figure 4.16) and say we are looking for n=24. This is how we proceed.

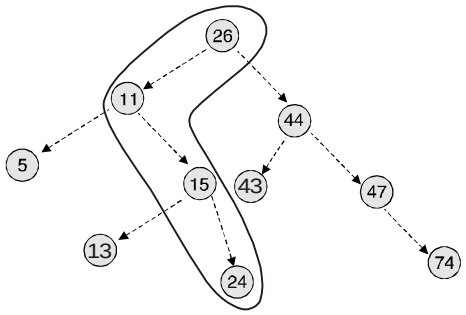


Figure 4.16

1. Compare n =24 with root i.e. 26 since 26 is greater than 24 we proceed towards left sub tree of root.
2. Now root becomes 11, since 11 is less than n =24 we move towards right sub tree of 11.
3. The right sub tree of 11 is having root of 15 now 15 is less than n =24 so we move towards right of 15.
4. The right sub tree of 15 is 24.

Algorithm 4.4 (To Search a Node in a Binary Search Tree)

Algorithm To Search a Node in a Binary Search Tree

```
Algorithm Search_BST (root,n)
{
    While (root!=NULL)
    {   if (data (root)= = n)
        {
            return root ;
        }
        if (data (root) < n)
            Search _BST(right(root),n)
        else
            Search_BST(left(root),n)
    }
        return Null          // Data is not present
}
```

Explanation

The above algorithm is the recursive algorithm in which we pass the root of the tree and the data to be find out in the tree. The algorithm gets the recursive calls with the left or right sub tree depending upon the data on the root and data n to be searched. The non recursive algorithm can be written and left for the readers.

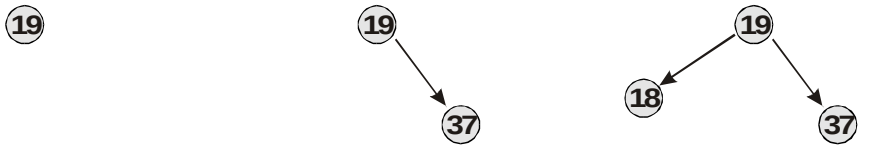
Inserting a Data

To insert a data in the binary tree we need to find out the position of the data where it exactly fits into the tree so that to the tree remains binary search tree even after inserting a node in the binary search tree.

Let's construct a binary search tree by following data to be added in a tree one by one.

19, 37, 18, 97, 82, 16, 9

The Construction of the above binary search tree involves the 7 steps which are shown in figure from 4.17 (a) to 4.17 (g)



n = 19 added (a) | **n = 37 added to its RHS (Since 37>19) (b)** | **n = 18 added (since 18<19, added with left) (c)**

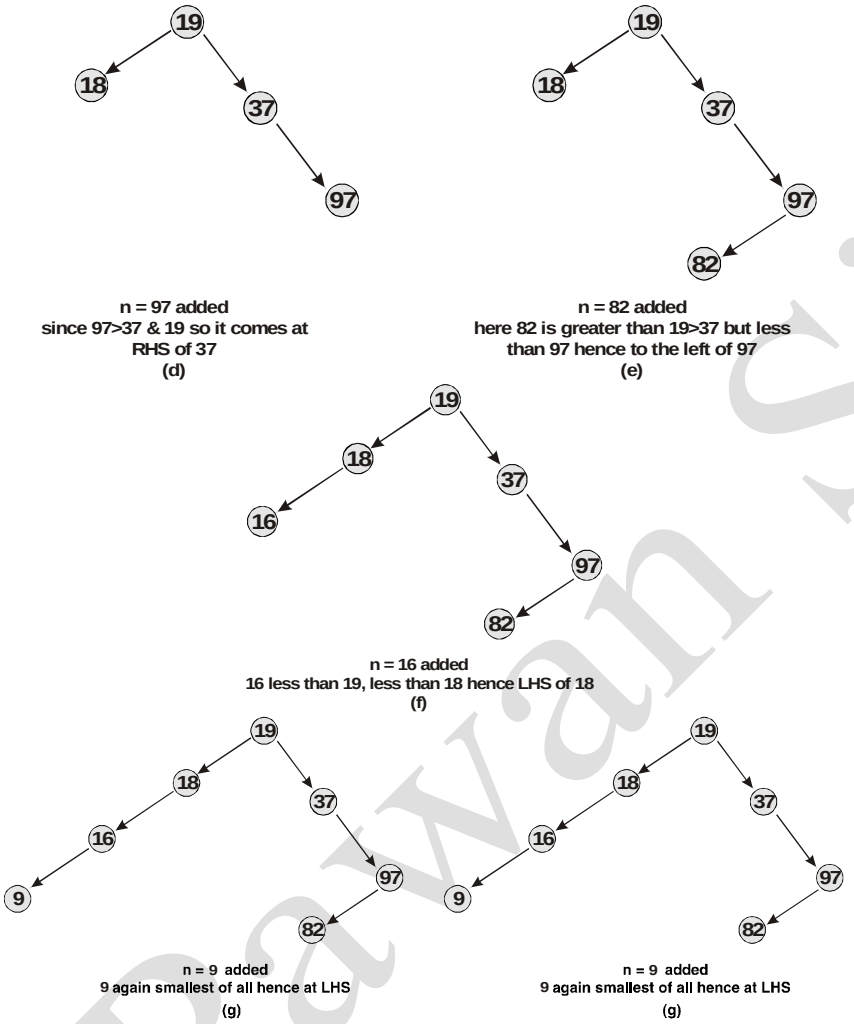


Figure 4.17

Algorithm 4.5 (To Insert in the Binary Search Tree)

Algorithm To Insert a Node in the Binary Search Tree

```
Algorithm Insert_BST (root, new)
{
        if (root == Null)
        {
                root =new
                return
        }
        while (root! = Null)
        {
                if (data(root) )>data(new))
                {
                        if (left (root) = =NULL)
                        {
                                left(root)=new
                                return ;
                        }
```

```
                        root = left (root) ;
                    }
                    else
                    {
                        if (right(root) == NULL)
                        {
                        right (root) = new
                        return
                        }
                       root = right (root)
                }
            }
        }
```

Explanation

The above algorithm is non-recursive algorithm where **root** and the **new** node to be added is passed to it.

There can be following three conditions.

1. The tree is empty i.e. root is NULL and then new node will become root.
2. The new node can be added to the left of the root.
3. New node can be added to the right of the root.

We start with the root and keep on comparing data of root with data of new node have to add and depending upon the condition whether lesser or greater we move the root to it's left sub tree or right sub tree. On finding any left or right of node as NULL, we add the new node there itself.

Deleting a Node of Binary Search Tree

Deleting a node from a tree T requires finding the particular node with data n say N is the node and P is the parent of the node to be deleted.

Deleting a node N from a tree T can have three cases

1. The node N to be deleted has no children. Then deletion is simplest by replacing **N** by **NULL** in the parent node i.e. P.

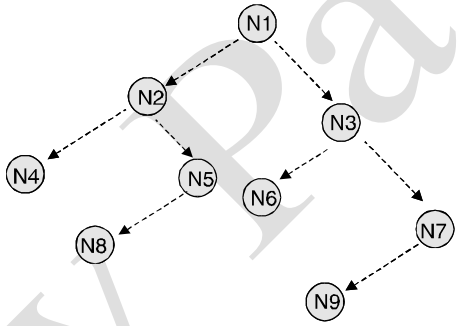


Figure 4.18

Delete N6

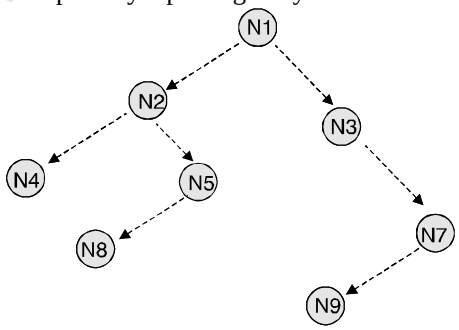


Figure 4.19

2. N has only one child either left or right say C. Here in this case the N is replaced by C in its parent.

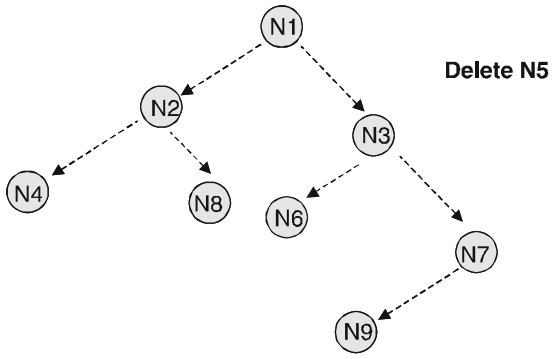


Figure 4.20

3. N has both the children left and right. In this case we have to find first the inorder successor of the N. Let's say M is the inorder successor of N. First M is deleted using case 1 or case 2. Then N is replaced by M. See figure.

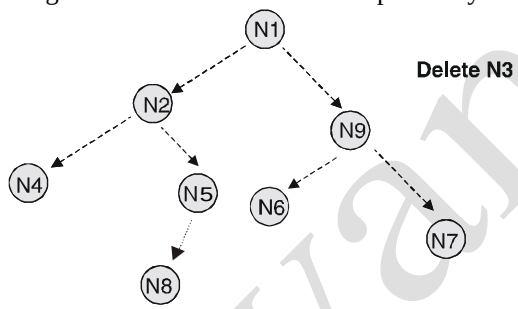


Figure 4.21

Algorithm 4.6 (To Delete a Node from a Binary Search Tree)

Algorithm To Delete a Node from a Binary Search Tree

```
Algorithm Delete_BST (root, n)
{
    P the parent of the node N to deleted with data n. flag is having value 0
    or 1 depending upon N present is left or right child of P.
    N = Search_BST (root,n)
    P = find_BST(root,n,flag)
    if (left(N)==NULL and right(N) = =NULL)
    {
        if (P==NULL)
            root = NULL
        else
        if(flag == 0)
            left (P) = NULL
        else
            right (P) = NULL
    }
    else
    {
    if (left(N) = =NULL)
    {
        if (P==NULL)
            root = right (N)
```

```
                        else
                        if (flag == 0)
                                left (P) = right (N)
                        else
                                right (P) = right (N)
                }
                else
                        {
                        if (right(N) ==NULL)
                        {
                        if (P==NULL)
                                root = left (N)
                        else
                        if (flag == 0)
                                left (P) = left (N)
                        else
                                right(P) = left (N)
        }
        else
        {
                S = right (N)
                PS = N
                while (left (S) ! = NULL)
                {
                PS = S
                S = left(S)
                 }
                left (PS) = right (S)
                left(S) = left (N)
                right(S) = right (N)
        if (P==NULL)
                root = S
                else
                        right (P) =S
        }
    }
    }
        delete (N)
}
```

Algorithm Find_BST (root, n, flag)

Algorithm Find_BST (root, n, flag)
{
If (root = null)
{
Not found // the data is not there
P = NULL
Return P
}
If (data (root) = n)

```
                { if (left (p) = root)
                        flag = 0
                   else
                        flag = 1
                  return P
                }
                If (data (root)<n)
                {
                        P = root
                        Find_BST (right(root), n)
                }
        Else
        {
                P = root
                Find _ BST(left(root), n)
        }
    }
```

Explanation

The above algorithm use one function named find_BST, this function does two things firstly it finds the parent of the node N i.e. P and secondly it sets the flag to 0 or 1 depending upon whether N is it's left or right sub-tree. P would be equal to NULL if the node to be deleted is root.

Let's understand the above algorithm by examples. Take figure 4.22

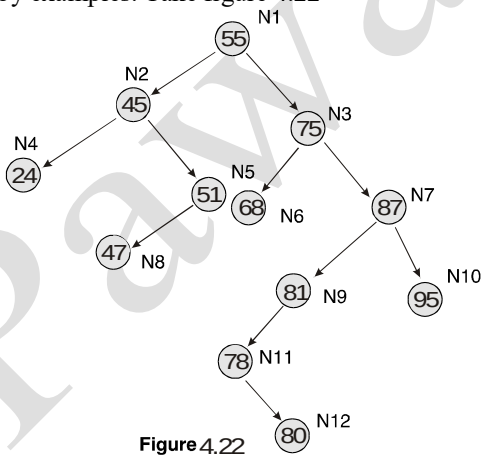


Figure 4.22

Lets do following

Step. 1 Delete 68 form BST shown in figure.4.23. We have root = N1. n = 68 is what is passed to the function Delete_BST ().Function search_BST would return the address of the node to be deleted i.e. N=N6. Function find_BST would return parent of N i.e. P=N3. and also flag would be set to . 0 because N6 is the left child of N3.Now since left(N)=NULL and also right(N)=NULL that means N is the leaf and according to algorithm we just put left (P)=NULL. That means left of N3=NULL and we get the tree shown is figure 4.23

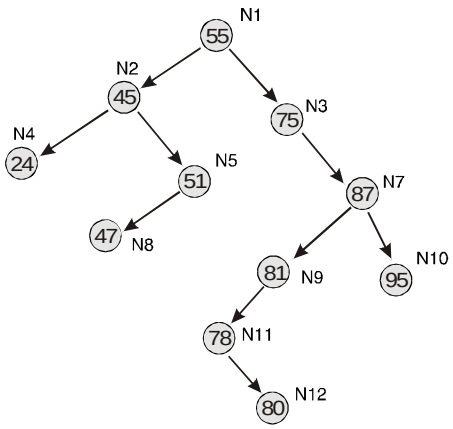


Figure 4.23

Step.2 Delete 51 from BST shown in figure 4.24

Here again root = N1, n=51.

Two function will set the following values

N=N5

P=N2

Flag=1

Here in this case it's only the right of N is NULL. And hence we check for P not equal to NULL and then flag not equal to zero hence right (P) = left (N). i.e. N8 is attached as a right child of the parent N2 as shown in figure 4.23 and at last N5 is deleted.

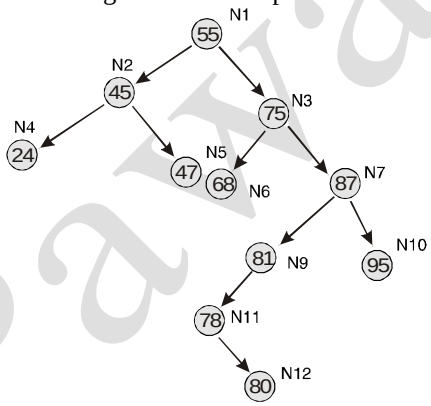


Figure 4.24

Step 3. Delete 75 from BST Shown in figure 4.25 (a) to 4.25 (d).

Here root = N1, n =75

Two functions find_BST and Search_BST will give us the following.

N=N3

P=N1

Flag =1

Now since N3 has both the right and left children we will have to find the inorder successor of N3.

S= right (N3) = N7

While (left(S)!=NULL) we move along with the left child of the nodes. And reaches at N11, PS denotes the parent of successor N11.

After execution of while loop we have the following.

S= N11.

PS = N9

left (PS) = right (S) will give us left (N9) = right (N11) i.e.N12

Hence N12 would be attached as left child of N9 as shown in figure 4.25(a)

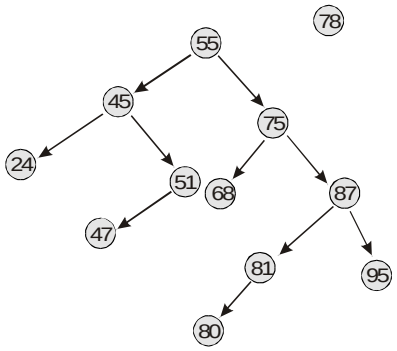


Figure 4.25 (a)

left(S)=left(N) i.e. left(N11)= left (N3) i.e. N6 will become the left child of N11 as shown in figure 4.25 (b)
right (S) = right (N) i.e. right (N11) = right (N3) i.e. N7 will become the right child of N11 as shown in figure in 4.25 (b).

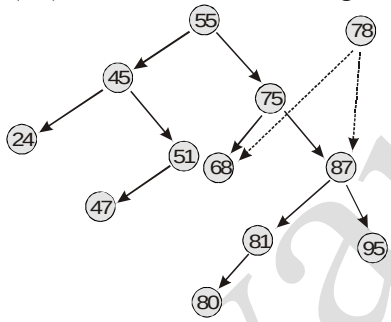


Figure 4.25(b)

if (P = NULL) condition is false and hence right (P) = S i.e.
root = S right (N1) will becomes N11 as shown in figure 4.25 (c)
else
right (P) = S

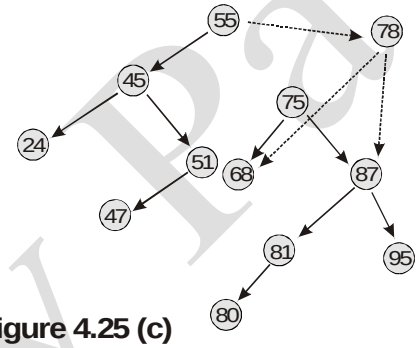


Figure 4.25 (c)

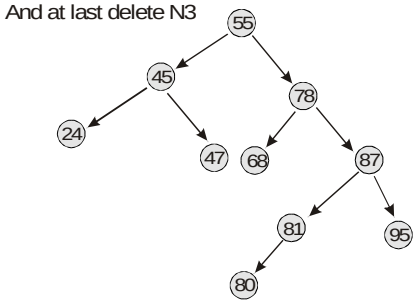


Figure 4.25 (d)

Step 4. Delete 55 from the BST shown in figure 4.26

In this case we have same case of having two children and but P = NULL would be true in this case. And 68 i.e. Node N6 will become root and we get the resultant BST as shown in figure 4.26

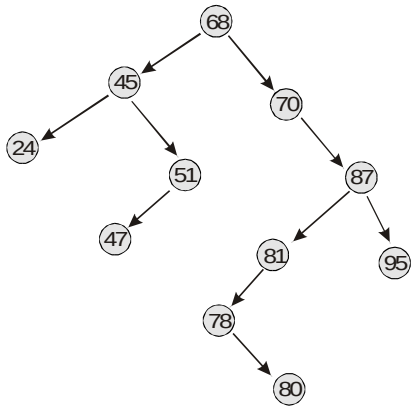


Figure 4.26

Readers can work out in the similar way as we did in removing 75 from BST to work step by step to get the resultant BST of figure 4.26

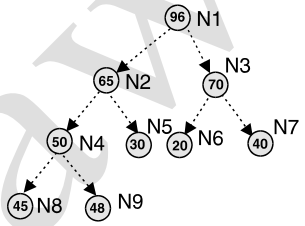
4.12 HEAP

Heap in another important data Structure basically used for sorting the particularly elements. The beauty of this sort is that it generally takes less time to sort the elements as compare to another sorting algorithm.

Let H is the complete binary tree with n elements. **H** is called **Heap** or a **maxheap** if each node of H is greater than or equal to the value of its children of H.

Similarly H is called minheap if each node N of H is less than or equal to the value of its children of N.

Example: Let's have a look on the tree shown in figure 4.27(a). The tree shown in figure is called as **maxheap** as we observe that any node N is greater than or equal to the value of its children. The heap shown in the figure can be represented by the sequential representation as shown in figure.4.27 (b) we will take the sequential representation of the heap in our discussion for simplicity.



Position	96	65	70	50	30	20	40	45	48
	0	1	2	3	4	5	6	7	8

Figure 4.27 (a) Figure 4.27 (b)

The parent of any child can be found out by the formula.

P [N] = (N -1) /2 for all odd N

N/2 - 1 for all even N.

Inserting into a Heap:

Inserting a node A into a heap has two steps.

1. Insert the node A at the end of the Heap so that the tree is still the complete tree but not necessarily heap.
2. Now raise the node A to it's appropriate position so that the tree finally becomes the heap by changing the position of a node with the parent node.

Example:

Let's take a heap is already shown in figure 4.27 let's add the element 77 to our heap.

1. Add 77 to the last of the complete binary tree so that the tree is still the complete tree i.e. 77 is added to node N5. as shown in figure 4.28 (a)
2. Now compare 77 with its parent. If the child is having more value than it's parent then change the position of the child i.e. change N5 and N10.

Again compare N5 with its parent i.e.N2. Now since N2 ie. 65 is less than 77 change the values of N5 and N2 and we get tree as shown in figure 4.28 (b)

Again compare N2 with N1 and since N1 is greater than N2, So no more shifting and we get the desired heap as shown in figure. 4.28 (c)

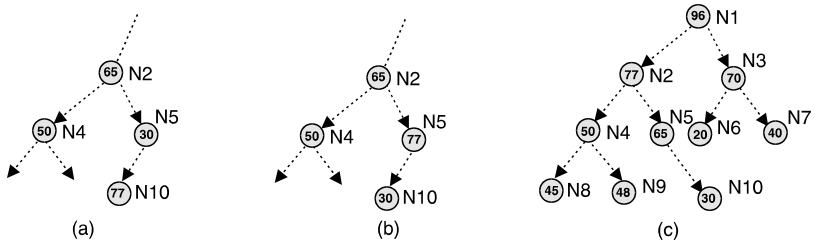


Figure 4.28 (a) (b) (c)

Algorithm 4.7 (To insert a node in a Heap)

In the following algorithm, Heap is the array containing M elements and n is the data to be inserted in heap. Pos is the position where the element would be entered. Ps is the position of parent of element at position Pos.

Algorithm To insert a node in a Heap

```
Algorithm Insert_Heap(Heap, M,n)
{
        Pos = M
        Heap [ pos] = n
        While (pos > 0)
        {
                if (pos is even)
                        ps = pos/2-1
                else
                        ps = (pos-1) /2
                if (Heap [ps] > = n)
                {
                M = M+1
                 Return
                 }
                else
                {
                        Heap [ pos] = Heap [ps]
                        Heap [ps] = n
                        Pos = ps
                }
        }
}
```

Explanation:

Let's apply the above algorithm to the example

The sequential representation of the above is shown in fig. 4.29 (a)

Heap [ ]	96	65	70	50	30	20	40	45	48				
Position	0	1	2	3	4	5	6	7	8	9	10		

Figure 4.29 (a)

Now lets add n = 77.

The element passed to the function would be M = 9 (Number of elements)

Step 1. Heap [Pos] = 77

Step 2. Pos = 9 i.e. odd hence Ps = 4

Now Heap [4] = 30, comparing 30 with 77

Since Heap [4] is not greater than n i.e. 77

Hence following actions

Heap [pos] = Heap [ps] i.e. Heap [9] = Heap [4] i.e. 30

Heap [ps] = n i.e. Heap [4] = 77

And Pos= ps i.e. pos = 4 and we get the following heap [ ]

Heap [ ]	96	65	70	50	77	20	40	45	48	30			
Position	0	1	2	3	4	5	6	7	8	9	10		

Figure 4.29 (b)

Step 3. Pos = 4 i.e. even hence ps =1
Now Heap [1] = 65 again less than 77.
Hence applying the same i.e. changing the elements of 1 and 4 we get the following heap.
Pos now become 1 i.e. equal to ps.

Heap [ ]	96	65	70	50	77	20	40	45	48	30			
Position	0	1	2	3	4	5	6	7	8	9	10		

Figure 4.29 (c)

Step 4. Pos = 1 i.e. odd Hence Ps = 0
Heap [0] = 96 > 77
Hence taking the following actions
M = M + 1 = 10
Return.
And hence we get the desired heap by inserting the element at desired position.

Delete Root from a Heap:

Deleting a Root from a heap required can be achieved by following steps.

1. Find the last node **L** so that deleting this node still makes the complete tree.
2. Store the root in some variable like **Root**. Replace the root in a heap with **L**. Delete **L**.
3. Now again adjust the elements so that **L** sinks to its position.

Now let's understand the above by an example given below.

Example:

Let's delete the root from the Heap shown in Figure.4.27

1. Find the last node **L** i.e. **N9** with data **48**.
2. Store the root i.e. **Root = 90**. And now replace the root of Heap with N9 and delete N9. We get the following tree.

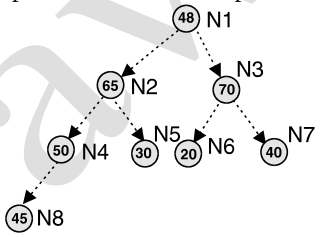


Figure 4.30

3. Now compare N1 with N2, since N1 is less than N2 change the values. Again compare N2 with N4 again 48 is less than 50 so change the values of N2 and N4. After the above steps we get the following tree.

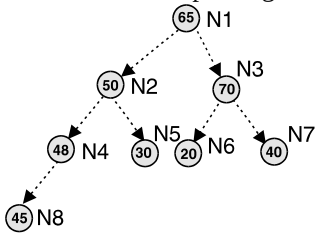


Figure 4.31

4. Now compare N4 with N8 since 48 is greater than 45 so, no more changes so, we get the desired result as shown in figure.4.31

Algorithm 4.8 (To delete a Root from the Heap)

In the algorithm given below the Heap is an array containing the M elements and root of the heap is to be deleted and stored in the variable n. L is the last element of the heap. Leftch and Rightch are the positions of left children and right children of the parent denoted by variable pos.

Algorithm To delete a Root from the Heap

```
Algorithm Delete_Heap(Heap, M, n)
{
        n = Heap[0]  //Root is stored in n
        L = Heap[M-1]  //Deletes the last elements.
        M = M-1
        Heap[0]= L  // stores the last element at the root and now reheap.
        Leftch = 1
        Rightch = 2
        Pos= 0
        While (Rightch < M)
        {
                if (L>= Heap[Leftch] and L > = Heap [rightch])
                {
                        Heap[pos] = L
                        Return
                }
                if (Heap[rightch] <=Heap[leftch])
                {
                        Heap[pos] = Heap[leftch]
                        Pos = left
                        }
                else
                {
                        Heap[pos] = Heap[rightch]
                        Pos= rightch
                }
        leftch = leftch*2+1
        rightch = leftch +1
}
if (leftch = N and L < Heap [leftch])
        pos = leftch
        Heap[pos] = L
}
```

HEAP SORT

Let there are M numbers of elements stored in some array say A to be sorted.

The heapsort consists of two parts.

1. Create a Heap out of array A.
2. Repeatedly delete the root from Heap till it becomes empty.

The algorithm will give us the sorted array because root of the **Heap** is always largest among its elements and hence every time deleting of the elements would give us the largest of among those elements. Use Maxheap if you want to sort in decending order else use **Minheap.**

Algorithm 4.9 (Heap Sort)

In the algorithm below, we use the Array A with M elements and Array Heap

```
Algorithm HeapSort (A,M)
{
//      Let's construct the heap first from array A with M elements.
        For (i = 0 to M-1)
        {
                insert_heap(Heap, i, a[i])
        }
```

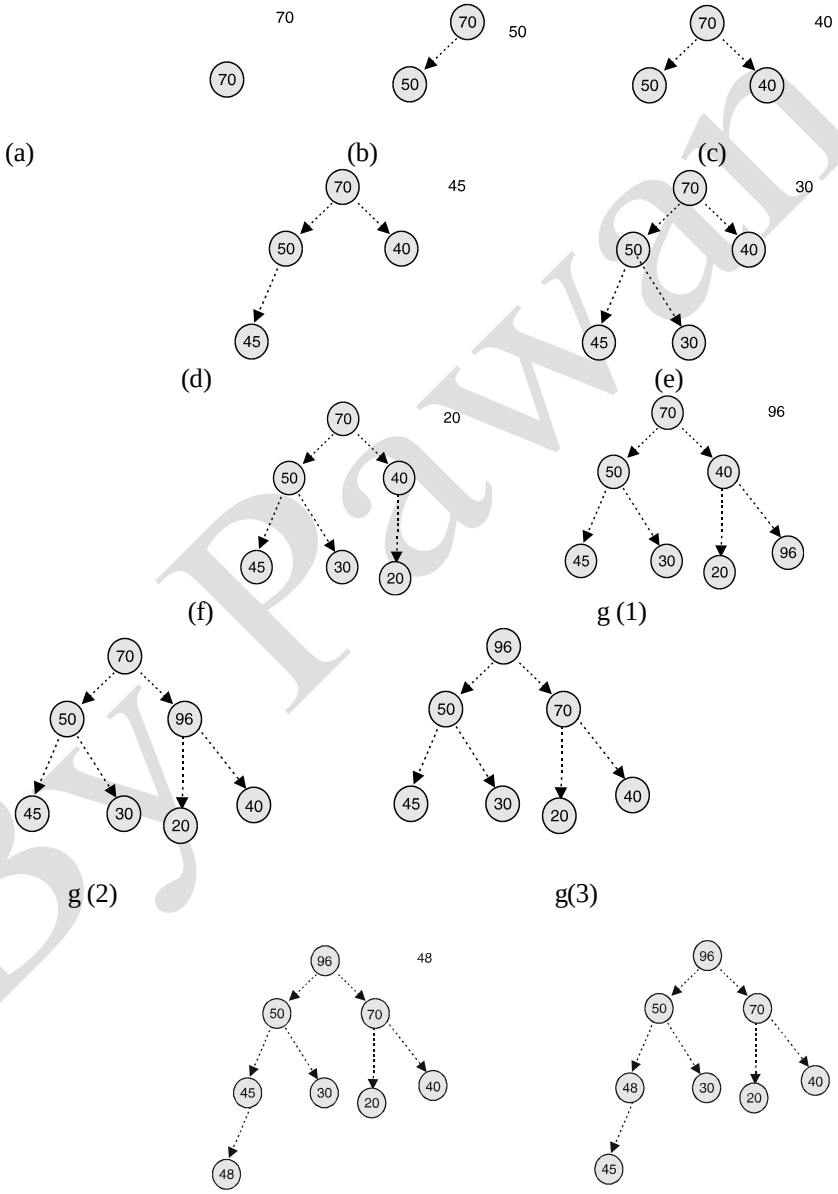
```
//      Now we have Heap as an array with M elements and let's delete the root of this array.
        for (i = 0 to M-1)
        {
                Delete_heap(Heap, M-i, n)
                A[i] = n
        }
// After this loop we have the sorted array of elements in array A.
}
```

The above algorithm is self explanatory as it just inserts the element from **Array A** to the Heap and then deletes the root of the Heap and puts back in **Array A**. As the root would always be the largest (as we are implementing MaxHeap) and deleting one by one from array Heap.

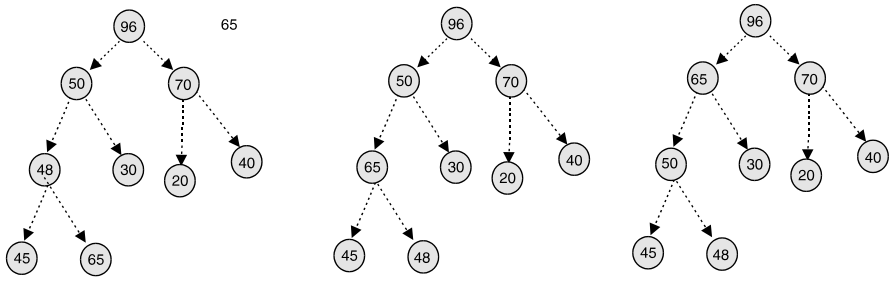
Let's have the following elements in an array A.

70 50 40 45 30 20 96 48 65

Figure 4.32 Create Heap



h(1) h(2)



i(1) i(2) i(3)

Figure 4.3

4.14 GENERAL TREES

A general tree (sometimes called a tree) is defined to be nonempty finite set T of elements, called nodes, such that:

(1)T contains a distinguished element R, called the root of T.

(2)The remaining elements of T form an ordered collection of Zero or more disjoint trees T_1, T_2, ...T_m.

The trees T_1, T_2....T_m are called subtrees of the root R, and the roots of T_1, T_2....T_m are called successors of R.

N is node with successors S_1, S_2,.....S_m then N is called the parent of the S_i's, the S_i's are called children of N, and the S_i's are called siblings of each other.

Example:

Figure 4.35 pictures a general tree T with 13 nodes.

A, B, C, D, E, F, G, H, J, K, L, M,N

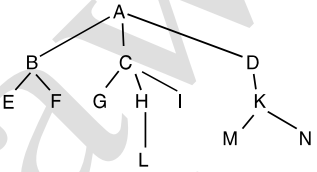


Figure 4.35

A binary tree T' is not a special case of a general tree T: binary trees and general trees are different objects. The two basic differences follow:

(1) A binary tree T' may be empty, but a general tree T is nonempty.

(2) Suppose a node N has only one child. then the child is distinguished as left child or right child in a binary tree T', but no such distinction exists in a general tree T.

The second difference is illustrated by the trees T_1 and T_2 in figure 4.36. Specifically, as binary trees, T_1 and T_2 are distinct trees, since B is the left child of A in the tree T_1 but B is the right child of A in the tree T_2. On the other hand, there is no difference between the trees T_1 and T_2 as general trees.

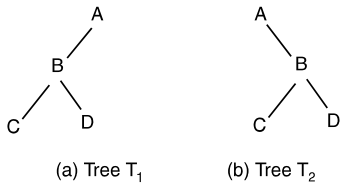


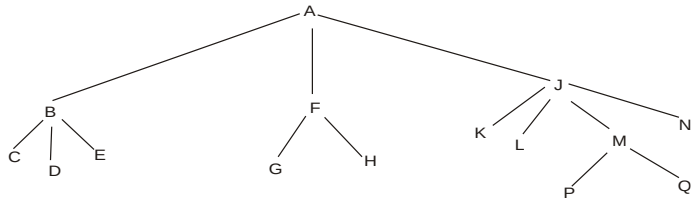
Figure 4.36

MISCELLANEOUS PROBLEMS

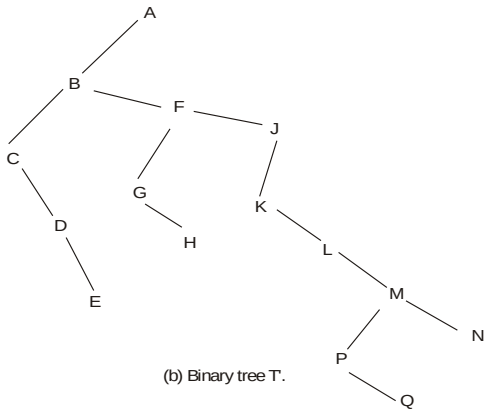
Q. Consider the general tree T in fig. (a). Find the corresponding binary tree T'.

Solution:

The nodes of T' will be the same as the nodes of the general tree T, and in particular, the root of T' will be the same as the root of T. Furthermore, if N is a node in the binary tree T', Then its left child is the first child of N in T and its right child is the next sibling of N in T. Constructing T' from the root down, we obtain the tree in fig (b)



(a) General tree T.



(b) Binary tree T'.