TREE

Trees are very flexible, versatile and powerful non-liner data structure that can be used to represent data items possessing hierarchical relationship between the grand father and his children and grand children as so on.

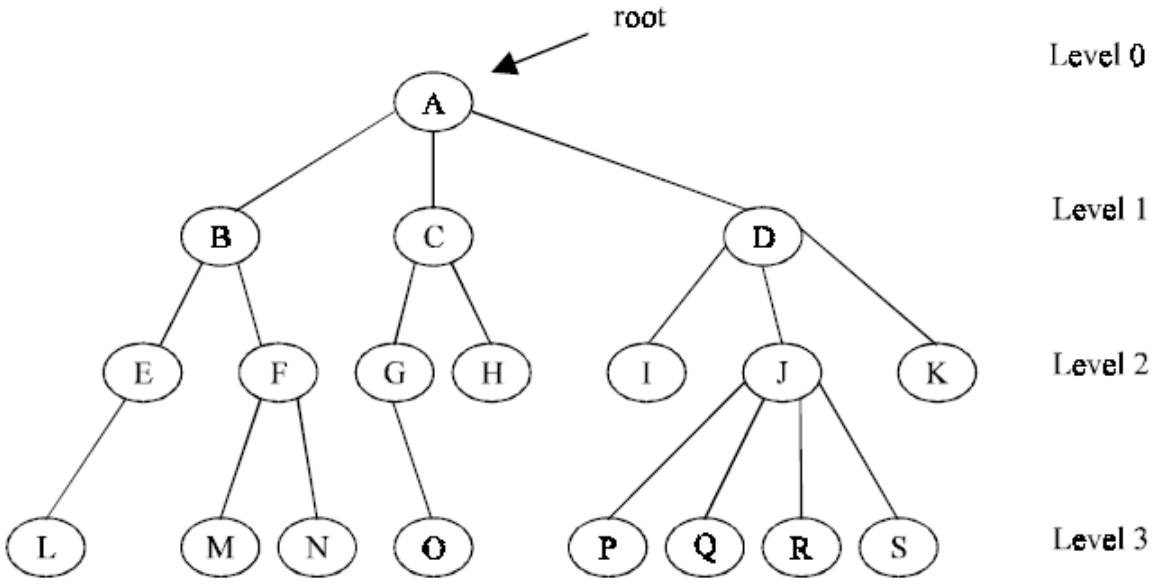

Fig. 8.1. A tree

A tree is an ideal data structure for representing hierarchical data. A tree can be theoretically defined as a finite set of one or more data items (or nodes) such that :

1. There is a special node called the root of the tree.
2. Removing nodes (or data item) are partitioned into number of mutually exclusive (*i.e.,* disjoined) subsets each of which is itself a tree, are called sub tree.

BASIC TERMINOLOGIES

Root is a specially designed node (or data items) in a tree. It is the first node in the hierarchical arrangement of the data items. 'A' is a root node in the Fig. 8.1. Each data item in a tree is called a *node*. It specifies the data information and links (branches) to other data items.

Degree of a node is the number of subtrees of a node in a give tree. In Fig. 8.1

The degree of node A is 3

The degree of node B is 2

The degree of node C is 2

The degree of node D is 3

The *degree of a tree* is the maximum degree of node in a given tree. In the above tree, degree of a node J is 4. All the other nodes have less or equal degree. So the degree of the above tree is 4. A node with degree zero is called a *terminal node* or a *leaf*. For example in the above tree Fig. 8.1. M, N, I, O etc. are leaf node. Any node whose degree is not zero is called *non-terminal node*. They are intermediate nodes in traversing the given tree from its root node to the terminal nodes.

The tree is structured in different *levels*. The entire tree is leveled in such a way that the root node is always of level 0. Then, its immediate children are at level 1 and their immediate children are at level 2 and so on up to the terminal nodes. That is, if a node is at level n, then its children will be at level $n + 1$.

Depth of a tree is the maximum level of any node in a given tree. That is a number of level one can descend the tree from its root node to the terminal nodes (leaves). The term *height* is also used to denote the depth.

Trees can be divided in different classes as follows :

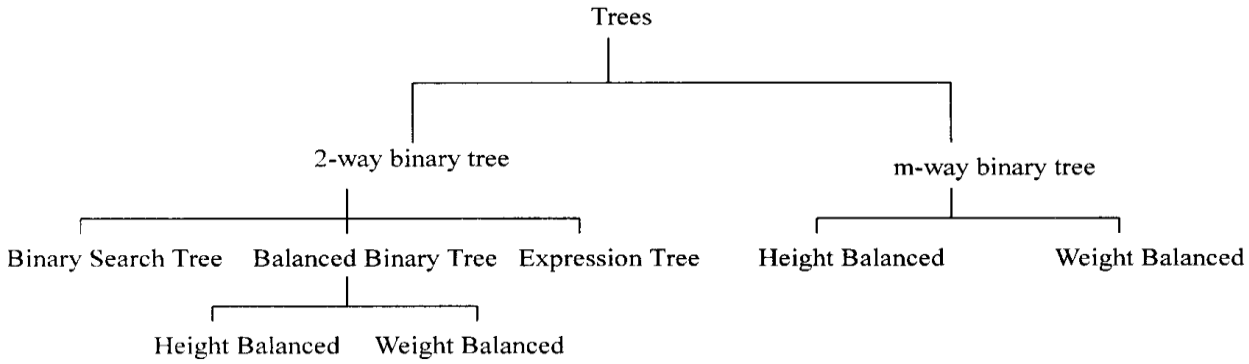

Fig. 8.2

Binary Tree

A binary tree is a tree in which no node can have more than two children. Typically these children are described as "left child" and "right child" of the parent node.

A binary tree T is defined as a finite set of elements, called nodes, such that :

1. T is empty (*i.e.*, if T has no nodes called the *null tree* or *empty tree*).

2. T contains a special node R, called root node of T, and the remaining nodes of T form an ordered pair of disjoined binary trees T1 and T2, and they are called left and right sub tree of R. If T1 is non empty then its root is called the left successor of R, similarly if T2 is non empty then its root is called the right successor of R.

Consider a binary tree T in Fig. 8.3. Here 'A' is the root node of the binary tree T. Then 'B' is the left child of 'A' and 'C' is the right child of 'A' *i.e.*, 'A' is a father of 'B' and 'C'. The node 'B' and 'C' are called brothers, since they are left and right child of the same father. If a node has no child then it is called a leaf node. Nodes P,H,I,F,J are leaf node in Fig. 8.3.

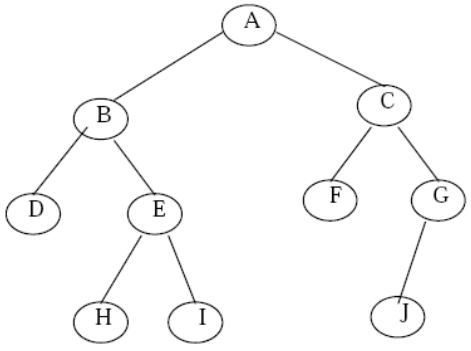

Fig. 8.3. Binary tree

The tree is said to be strictly binary tree, if every non-leaf made in a binary tree has non-empty left and right sub trees. A strictly binary tree with n leaves always contains 2n–1 nodes. The tree in Fig. 8.4 is strictly binary tree, whereas the tree in Fig. 8.3 is not. That is every node in the strictly binary tree can have either no children or two children. They are also called 2-tree or extended binary tree.

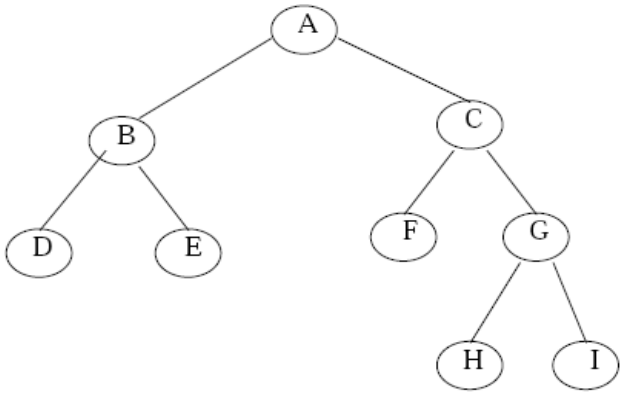

Fig. 8.4. Strictly binary tree

The main application of a 2-tree is to represent and compute any algebraic expression using binary operation.

For example, consider an algebraic expression E.

$$E = (a + b)/((c - d)*e)$$

E can be represented by means of the extended binary tree T as shown in Fig. 8.5. Each variable or constant in E appears as an internal node in T whose left and right sub tree corresponds to the operands of the operation.

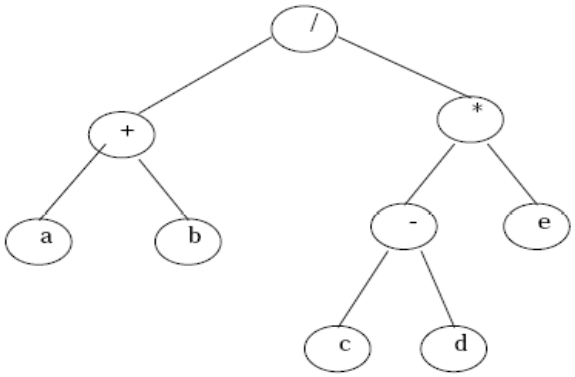

Fig. 8.5. Expression tree

A complete binary tree at depth '*d*' is the strictly binary tree, where all the leaves are at level *d*. Fig. 8.6 illustration the complete binary tree of depth 2.

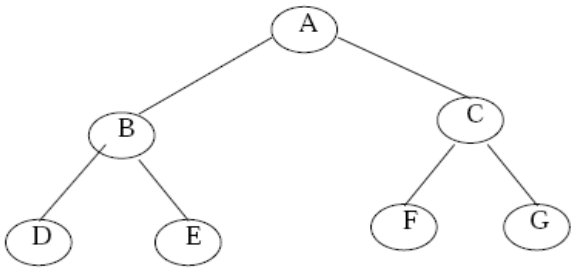

Fig. 8.6. Complete binary tree

A binary tree with n nodes, $n > 0$, has exactly $n - 1$ edges. A binary tree of depth d, $d > 0$, has at least d and at mast $2^d - 1$ nodes in it. If a binary tree contains n nodes at level l, then it contains at most $2n$ nodes at level $l + 1$. A complete binary tree of depth d is the binary tree of depth d contains exactly 2^l nodes at each level l between 0 and d.

Finally, let us discuss in briefly the main difference between a binary tree and ordinary tree is:

1. A binary tree can be empty where as a tree cannot.
2. Each element in binary tree has exactly two sub trees (one or both of these sub trees may be empty). Each element in a tree can have any number of sub trees.
3. The sub tree of each element in a binary tree are ordered, left and right sub trees. The sub trees in a tree are unordered.

If a binary tree has only left sub trees, then it is called left skewed binary tree. Fig. 8.7(*a*) is a left skewed binary tree.

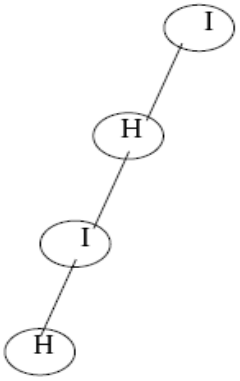

Fig. 8.7(*a*). Left skewed

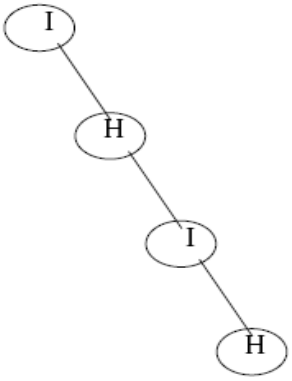

Fig. 8.7(*b*). Right skewed

If a binary tree has only right sub trees, then it is called right skewed binary tree. Fig. 8.7(*b*) is a right skewed binary tree.

Binary Tree Representation

This section discusses two ways of representing binary tree T in memory:

1. Sequential representation using arrays
2. Linked list representation

ARRAY REPRESENTATION

An array can be used to store the nodes of a binary tree. The nodes stored in an array of memory can be accessed sequentially

Suppose a binary tree T of depth d. Then at most 2^{d-1} nodes can be there in T.(*i.e.*, SIZE = 2^d–1) So the array of size "SIZE" to represent the binary tree. Consider a binary tree in Fig. 8.8 of depth 3. Then SIZE = $2^3 - 1 = 7$. Then the array A[7] is declared to hold the nodes.

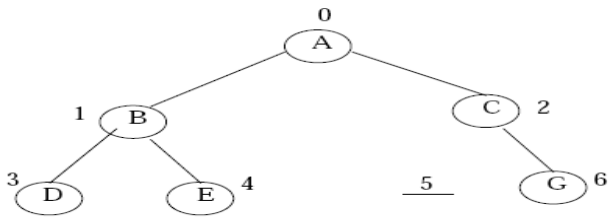

Fig. 8.8. Binary tree of depth 3

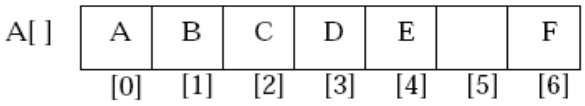

Fig. 8.9. Array representation of the binary tree

The array representation of the binary tree is shown in Fig. 8.9. To perform any operation often we have to identify the father, the left child and right child of an arbitrary node.

1. The father of a node having index n can be obtained by $(n - 1)/2$. For example to find the father of D, where array index $n = 3$. Then the father nodes index can be obtained

$$= (n - 1)/2$$
$$= 3 - 1/2$$
$$= 2/2$$
$$= 1$$

i.e., A[1] is the father D, which is B.

2. The left child of a node having index n can be obtained by (2n+1). For example to find the left child of C, where array index $n = 2$. Then it can be obtained by

$$= (2n + 1)$$
$$= 2*2 + 1$$
$$= 4 + 1$$
$$= 5$$

i.e., A[5] is the left child of C, which is NULL. So no left child for C.

3. The right child of a node having array index n can be obtained by the formula $(2n + 2)$. For example to find the right child of B, where the array index $n = 1$. Then

$$= (2n + 2)$$
$$= 2*1 + 2$$
$$= 4$$

i.e., A[4] is the right child of B, which is E.

4. If the left child is at array index n, then its right brother is at $(n + 1)$. Similarly, if the right child is at index n, then its left brother is at $(n - 1)$.

The array representation is more ideal for the complete binary tree. The Fig. 8.8 is not a complete binary tree. Since there is no left child for node C, *i.e.,* A[5] is vacant. Even though memory is allocated for A[5] it is not used, so wasted unnecessarily.

LINKED LIST REPRESENTATION

The most popular and practical way of representing a binary tree is using linked list (or pointers). In linked list, every element is represented as nodes. A node consists of three fields such as :

(*a*) Left Child (LChild)
(*b*) Information of the Node (Info)
(*c*) Right Child (RChild)

The L Child links to the left child node of the parent node, Info holds the information of every node and R Child holds the address of right child node of the parent node. Fig. 8.10 shows the structure of a binary tree node.

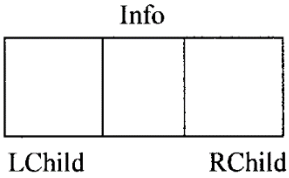

Fig. 8.10

Following figure (Fig. 8.11) shows the linked list representation of the binary tree in Fig. 8.8.

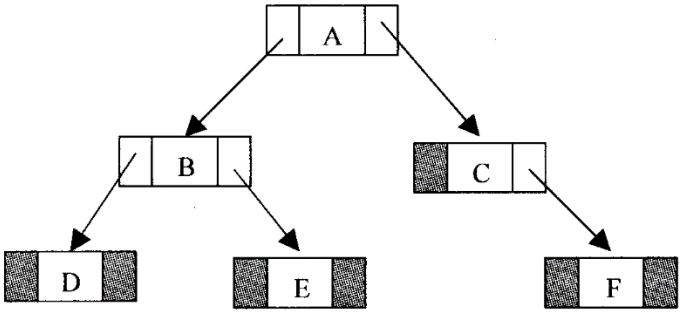

Fig. 8.11

If a node has left or/and right node, corresponding L Child or R Child is assigned to NULL. The node structure can be logically represented in C/C++ as:

```
struct Node
{
        int Info;
        struct Node *Lchild;
        struct Node *Rchild;
};

typedef struct Node *NODE;
```

OPERATIONS ON BINARY TREE

The basic operations that are commonly performed an binary tree is listed below :

1. Create an empty Binary Tree
2. Traversing a Binary Tree
3. Inserting a New Node
4. Deleting a Node
5. Searching for a Node

TRAVERSING BINARY TREES RECURSIVELY

Tree traversal is one of the most common operations performed on tree data structures. It is a way in which each node in the tree is visited exactly once in a systematic manner. There are three standard ways of traversing a binary tree. They are:
1. Pre Order Traversal (Node-left-right)
2. In order Traversal (Left-node-right)
3. Post Order Traversal (Left-right-node)

PRE ORDERS TRAVERSAL RECURSIVELY
To traverse a non-empty binary tree in pre order following steps one to be processed
1. Visit the root node
2. Traverse the left sub tree in preorder
3. Traverse the right sub tree in preorder
That is, in preorder traversal, the root node is visited (or processed) first, before traveling through left and right sub trees recursively.

```
void preorder (Node * Root)
{
  If (Root != NULL)
    {
      printf ("%d\n",Root → Info);
     preorder(Root → L child);
     preorder(Root → R child);
    }
}
```

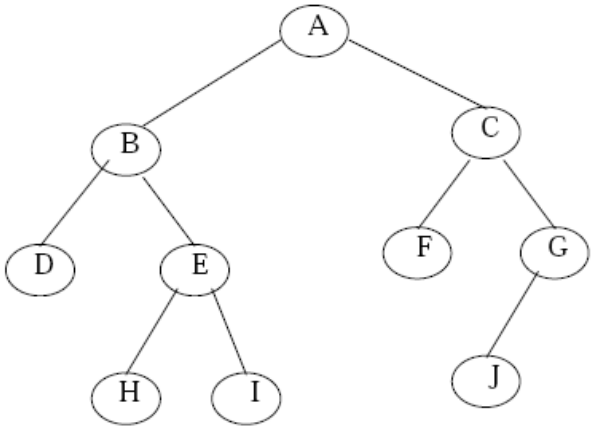

Fig. 8.12

The preorder traversal of a binary tree in Fig. 8.12 is A, B, D, E, H, I, C, F, G, J.

IN ORDER TRAVERSAL RECURSIVELY
The in order traversal of a non-empty binary tree is defined as follows:
1. Traverse the left sub tree in order
2. Visit the root node
3. Traverse the right sub tree in order
In order traversal, the left sub tree is traversed recursively, before visiting the root. After visiting the root the right sub tree is traversed recursively, in order fashion. The procedure for an in order traversal is given below :

```
void inorder (NODE *Root)
{
If (Root != NULL)
{
inorder(Root → L child);
```

```
printf ("%d\n",Root → info);
inorder(Root → R child);
}
}
```
The in order traversal of a binary tree in Fig. 8.12 is D, B, H, E, I, A, F, C, J, G.

POST ORDER TRAVERSAL RECURSIVELY

The post order traversal of a non-empty binary tree can be defined as :
1. Traverse the left sub tree in post order
2. Traverse the right sub tree in post order
3. Visit the root node

In Post Order traversal, the left and right sub tree(s) are recursively processed before visiting the root.

```
void postorder (NODE *Root)
{
If (Root != NULL)
{
postorder(Root → Lchild);
postorder(Root → Rchild);
printf ("%d\n",Root à info);
}
}
```
The post order traversal of a binary tree in Fig. 8.12 is D, H, I, E, B, F, J, G, C, A

BINARY SEARCH TREES

A Binary Search Tree is a binary tree, which is either empty or satisfies the following properties :
1. Every node has a value and no two nodes have the same value (i.e., all the values are unique).
2. If there exists a left child or left sub tree then its value is less than the value of the root.
3. The value(s) in the right child or right sub tree is larger than the value of the root node.

All the nodes or sub trees of the left and right children follows above rules. The Fig. 8.14 shows a typical binary search tree. Here the root node information is 50. Note that the right sub tree node's value is greater than 50, and the left sub tree nodes value is less than 50. Again right child node of 25 has large values than 25 and left child node has small values than 25. Similarly right child node of 75 has large values than 75 and left child node has small values that 75 and so on.

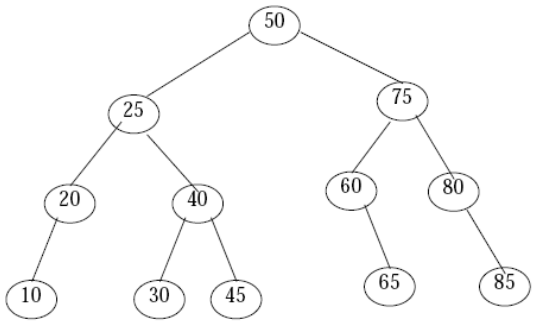

Fig. 8.14

The operations performed on binary tree can also be applied to Binary Search Tree (BST). But in this section we discuss few other primitive operators performed on BST :
1. Inserting a node
2. Searching a node
3. Deleting a node

Another most commonly performed operation on BST is, traversal. The tree traversal algorithm (pre-order, post-order and in-order) are the standard way of traversing a binary search tree.

INSERTING A NODE

A BST is constructed by the repeated insertion of new nodes to the tree structure. Inserting a node in to a tree is achieved by performing two separate operations.

1. The tree must be searched to determine where the node is to be inserted.
2. Then the node is inserted into the tree.

Suppose a "DATA" is the information to be inserted in a BST.

Step 1: Compare DATA with root node information of the tree

(*i*) If (DATA < ROOT → Info)

Proceed to the left child of ROOT

(*ii*) If (DATA > ROOT → Info)

Proceed to the right child of ROOT

Step 2: Repeat the Step 1 until we meet an empty sub tree, where we can insert the DATA in place of the empty sub tree by creating a new node.

Step 3: Exit

For example, consider a binary search tree in Fig. 8.14. Suppose we want to insert a DATA = 55 in to the tree, then following steps one obtained :

1. Compare 55 with root node info (*i.e.,* 50) since 55 > 50 proceed to the right sub tree of 50.
2. The root node of the right sub tree contains 75. Compare 55 with 75. Since 55 <75 proceed to the left sub tree of 75.
3. The root node of the left sub tree contains 60. Compare 55 with 60. Since 55 < 60 proceed to the right sub tree of 60.
4. Since left sub tree is NULL place 55 as the left child of 60 as shown in Fig. 8.15.

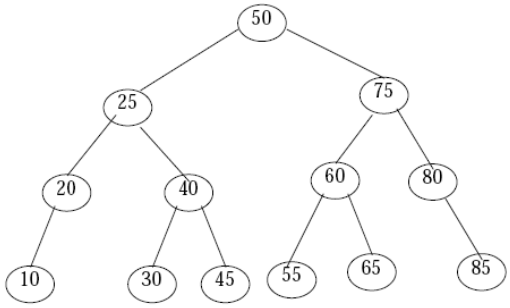

Fig. 8.15

ALGORITHM

NEWNODE is a pointer variable to hold the address of the newly created node. DATA is the information to be pushed.

1. Input the DATA to be pushed and ROOT node of the tree.
2. NEWNODE = Create a New Node.
3. If (ROOT == NULL)

(*a*) ROOT=NEW NODE

4. Else If (DATA < ROOT □ Info)

(*a*) ROOT = ROOT → Lchild

(*b*) GoTo Step 6

5. Else If (DATA > ROOT → Info)

(*a*) ROOT = ROOT → Rchild

(*b*) GoTo Step 7

6. If (DATA < ROOT → Info)

(*a*) ROOT → LChild = NEWNODE

7. Else If (DATA > ROOT → Info)

(*a*) ROOT → RChild = NEWNODE

8. NEW NODE → Info = DATA
9. NEW NODE → LChild = NULL
10. NEW NODE → RChild = NULL
11. EXIT

SEARCHING A NODE

Searching a node was part of the operation performed during insertion. Algorithm to search as element from a binary search tree is given below.

ALGORITHM

1. Input the DATA to be searched and assign the address of the root node to ROOT.
2. If (DATA == ROOT → Info)
(*a*) Display "The DATA exist in the tree"
(*b*) GoTo Step 6
3. If (ROOT == NULL)
(*a*) Display "The DATA does not exist"
(*b*) GoTo Step 6
4. If(DATA > ROOT→Info)
(*a*) ROOT = ROOT→RChild
(*b*) GoTo Step 2
5. If(DATA < ROOT→Info)
(*a*) ROOT = ROOT→Lchild
(*b*) GoTo Step 2
6. Exit

Suppose a binary search tree contains *n* data items, A1, A2, A3……..A*n*. There are *n*! Permutations of the *n* items. The average depth of the n! tree is approximately C log 2 *n*, where C=1.4. The average running time *f* (*n*) to search for an item in a binary tree with *n* elements is proportional to $\text{Log}_2 n$, that is $f(n) = O(\text{Log}_2 n)$

DELETING A NODE

This section gives an algorithm to delete a DATA of information from a binary search tree. First search and locate the node to be deleted. Then any one of the following conditions arises:

1. The node to be deleted has no children
2. The node has exactly one child (or sub tress, left or right sub tree)
3. The node has two children (or two sub tress, left and right sub tree)

Suppose the node to be deleted is N. If N has no children then simply delete the node and place its parent node by the NULL pointer.

If N has one child, check whether it is a right or left child. If it is a right child, then find the smallest element from the corresponding right sub tree. Then replace the smallest node information with the deleted node. If N has a left child, find the largest element from the corresponding left sub tree. Then replace the largest node information with the deleted node.

The same process is repeated if N has two children, *i.e*., left and right child. Randomly select a child and find the small/large node and replace it with deleted node. NOTE that the tree that we get after deleting a node should also be a binary search tree.

Deleting a node can be illustrated with an example. Consider a binary search tree in Fig. 8.15. If we want to delete 75 from the tree, following steps are obtained:

Step 1: Assign the data to be deleted in DATA and NODE = ROOT.

Step 2: Compare the DATA with ROOT node, *i.e*., NODE, information of the tree. Since (50 < 75)
NODE = NODE → RChild

Step 3: Compare DATA with NODE. Since (75 = 75) searching successful. Now we have located the data to be deleted, and delete the DATA. (See Fig. 8.16)

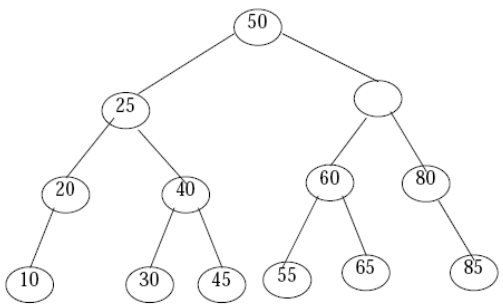

Fig. 8.16

Step 4: Since NODE (*i.e.*, node where value was 75) has both left and right child choose one. (Say Right Sub Tree) - If right sub tree is opted then we have to find the smallest node. But if left sub tree is opted then we have to find the largest node.
Step 5: Find the smallest element from the right sub tree (*i.e.*, 80) and replace the node with deleted node. (See Fig. 8.17)

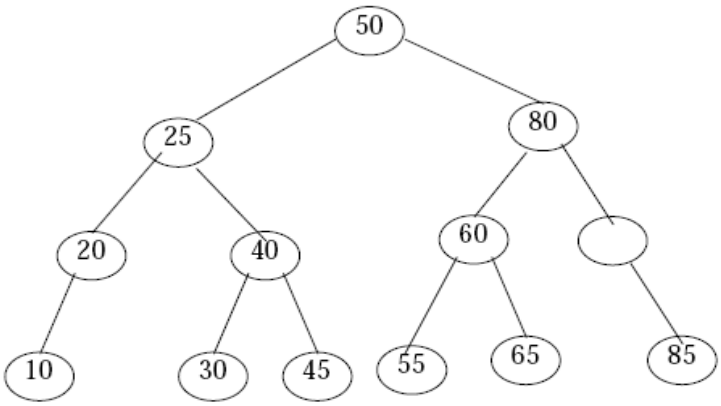

Fig. 8.17

Step 6: Again the (NODE → Rchild is not equal to NULL) find the smallest element from the right sub tree (Which is 85) and replace it with empty node. (See Fig. 8.18)

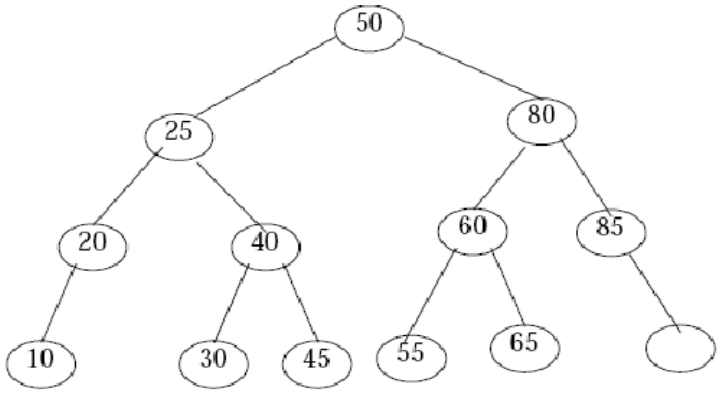

Fig. 8.18

Step 7: Since (NODE→Rchild = NODE→Lchild = NULL) delete the NODE and place NULL in the parent node. (See Fig. 8.19)

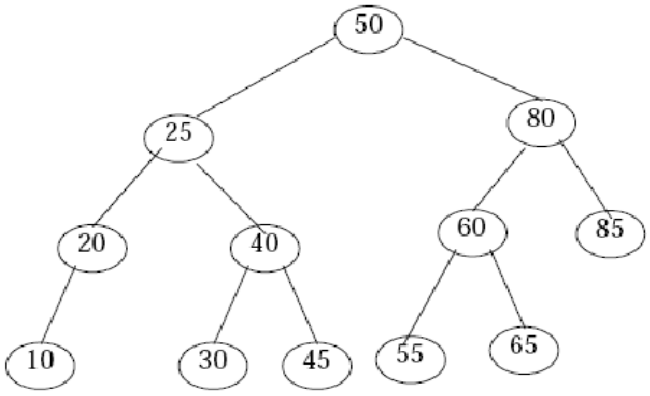

Fig 8.19

Step 8: Exit.

ALGORITHM

NODE is the current position of the tree, which is in under consideration. LOC is the place where node is to be replaced. DATA is the information of node to be deleted.

1. Find the location NODE of the DATA to be deleted.
2. If (NODE = NULL)
(a) Display "DATA is not in tree"
(b) Exit
3. If(NODE → Lchild = NULL)
(a) LOC = NODE
(b) NODE = NODE → RChild
4. If(NODE → RChild= =NULL)
(a) LOC = NODE
(b) NODE = NODE → LChild
5. If((NODE → Lchild not equal to NULL) && (NODE → Rchild not equal to NULL))
(a) LOC = NODE → RChild
6. While(LOC → Lchild not equal to NULL)
(a) LOC = LOC → Lchild
7. LOC → Lchild = NODE → Lchild
8. LOC → RChild= NODE → RChild
9. Exit

AVL TREES

This algorithm was developed in 1962 by two Russian Mathematicians, G.M. Adel'son Vel'sky and E.M. Landis; here the tree is called AVL Tree. An AVL tree is a binary search tree in which the left and right sub tree of any node may differ in height by at most 1, and in which both the sub trees are themselves AVL Trees. Each node in the AVL Tree possesses any one of the following properties:

(*a*) A node is called left heavy, if the largest path in its left sub tree is one level larger than the largest path of its right sub tree.

(*b*) A node is called right heavy, if the largest path in its right sub tree is one level larger than the largest path of its left sub tree.

(*c*) The node is called balanced, if the largest paths in both the right and left sub trees are may differ in height by at most 1. Fig. 8.43 shows some example for AVL trees.

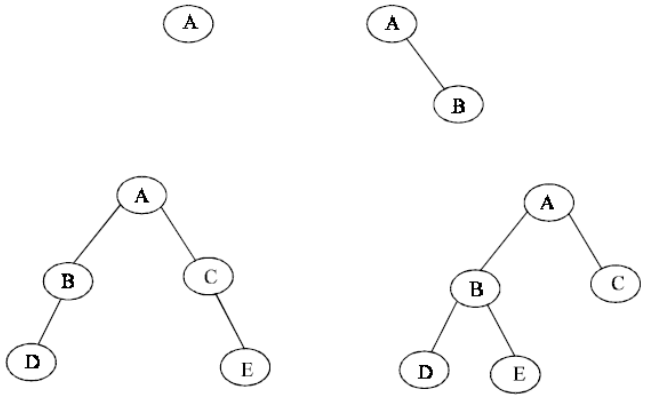

Fig. 8.43. AVL tree

Balance factor

The balance factor of a AVL tree may be 1,0 and -1

1 when the node is left heavy i.e. left height is one more than right height

0 when both left and right height are equal

-1 when the node is right heavy i.e. right height is one more than left height

The construction of an AVL Tree is same as that of an ordinary binary search tree except that after the addition of each new node, a check must be made to ensure that the AVL balance conditions have not been violated. If the new

node causes an imbalance in the tree, some rearrangement of the tree's nodes must be done. Following algorithm will insert a new node in an AVL Tree:

ALGORITHM

1. Insert the node in the same way as in an ordinary binary search tree.
2. Trace a path from the new nodes, back towards the root for checking the height difference of the two sub trees of each node along the way.
3. Consider the node with the imbalance and the two nodes on the layers immediately below.
4. If these three nodes lie in a straight line, apply a single rotation to correct the imbalance.
5. If these three nodes lie in a dogleg pattern (i.e., there is a bend in the path) apply a double rotation to correct the imbalance.
6. Exit.

The above algorithm will be illustrated with an example shown in Fig. 8.44, which is an unbalance tree. We have to apply the rotation to the nodes 40, 50 and 60 so that a balance tree is generated. Since the three nodes are lying in a straight line, single rotation is applied to restore the balance.

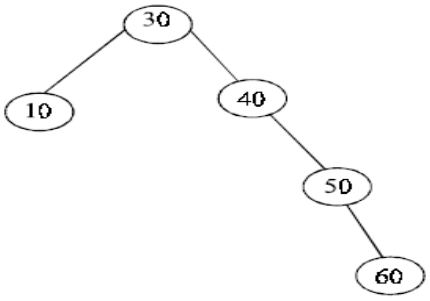

Fig. 8.44

Fig. 8.45 is a balance tree of the unbalanced tree in Fig. 8.44. Consider a tree in Fig. 8.46 to explain the double rotation.

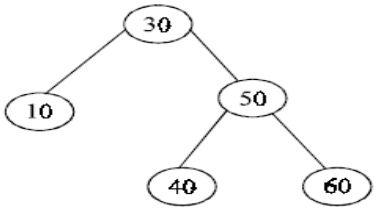

Fig. 8.45

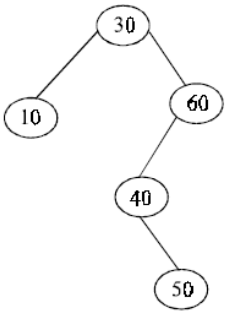

Fig. 8.46

While tracing the path, first imbalance is detected at node 60. We restrict our attention to this node and the two nodes immediately below it (40 and 50). These three nodes form a dogleg pattern. That is there is bend in the path. Therefore we apply double rotation to correct the balance. A double rotation, as its name implies, consists of two single rotations, which are in opposite direction. The first rotation occurs on the two layers (or levels) below the node where the

imbalance is found (i.e., 40 and 50). Rotate the node 50 up by replacing 40, and now 40 become the child of 50 as shown in Fig. 8.47.

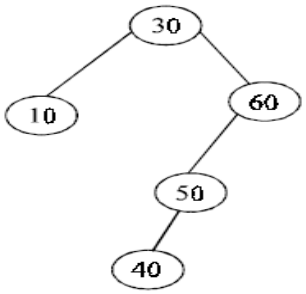

Fig. 8.47

Apply the second rotation, which involves the nodes 60, 50 and 40. Since these three nodes are lying in a straight line, apply the single rotation to restore the balance, by replacing 60 by 50 and placing 60 as the right child of 50 as shown in Fig. 8.48.

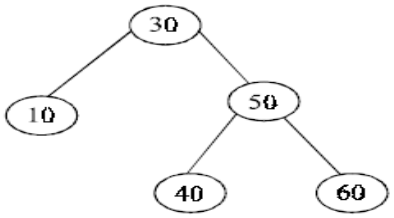

Fig. 8.48

Balanced binary tree is a very useful data structure for searching the element with less time. An unbalanced binary tree takes O(n) time to search an element from the tree, in the worst case. But the balanced binary tree takes only O(Log n) time complexity in the worst case.