35 50 15 25 80 20 90 45

# 10

# TREES

The data structures that we have discussed in previous chapters are linear data structures. The examples of these linear data structures are array, queue, linked list, and stack. However, there are many applications where linear data structures are not applied (used). In such cases, there is a need for some non-linear data structures such as tree and graph. In this chapter, an introduction to the tree data structure is given. The advanced topics about tree data structure are given in the next chapter i.e. chapter 11.

## 10.1 TREE

Tree is the most widely used *abstract data type* (ADT) or data structure. It is a non-linear data structure. It is a collection of nodes (or entities) that are organized in a hierarchical structure (without having any cycle). In a tree data structure, every individual entity or element is called a *node*. The nodes are connected by edges (The connection line between two nodes is called an *edge*). Each node contains a value or data. The first node of the tree is called the *root node* and it may contain several links.

Suppose a person's name is Majeed and he has four children, Aslam, Saleem, Haleem, and Naseem.

- ★ Aslam has three children, Aqsa, Hadia and Iqra
- ★ Saleem has four children, Sajid, Aysa, Sadia, and Fatima
- ★ Haleem and Naseem are un-married.

The graphical representation of Majeed's family is shown in figure 10.1.

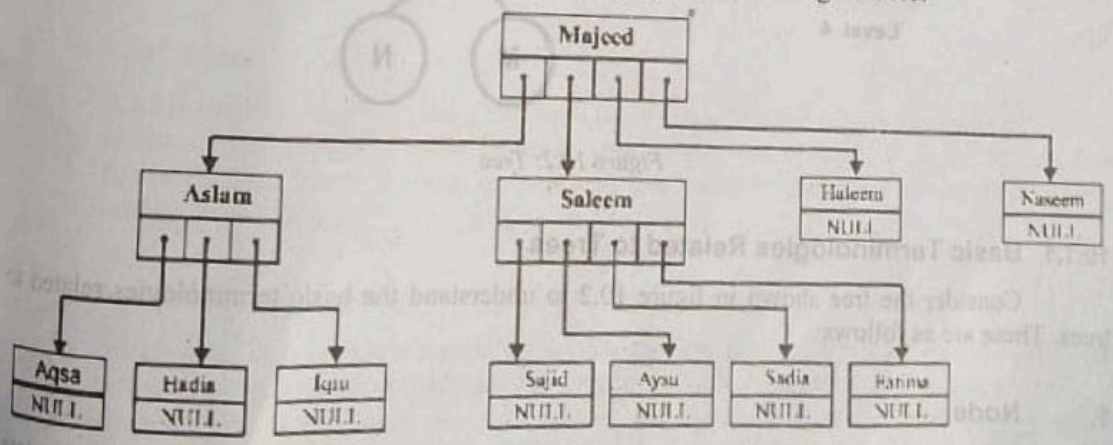
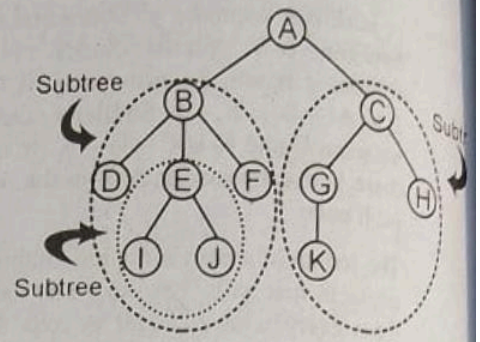



*Figure 10.1: Graphical representation of Majeed's family*

### 14. Subtree

The child node of the root node that has its own child nodes is called a *subtree*. The figure shows subtrees and the roots of these subtrees are B, C, and E.




### 15. Degree of Node

The total number of children (child nodes) of a node is called the *degree of that node*. In figure 10.2, the degree of the node E is one and the degree of node K is two.

### 16. Empty Tree

A tree that has no node is called an *empty tree* or *null tree*. Its height is – 1.

### 17. Singleton Tree

A tree whose root is its only node is called a *singleton tree*.

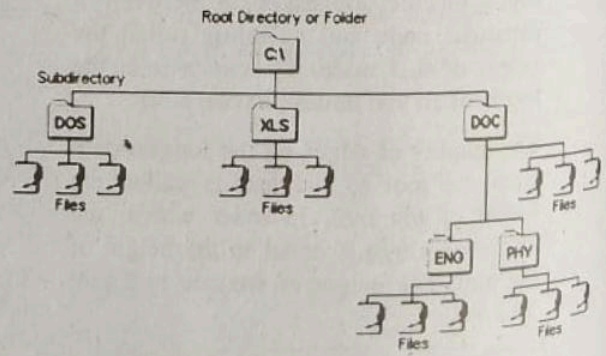
### 18. General Tree

A tree in which a node may have no child, one child, or any number of children is called a *general tree*.

## 10.1.2 Applications of Tree Data Structure

Tree is a very important data structure. In computer science, the tree data structure is used for many purposes. Some of these are as follows:

- **File Systems:** It is used to implement hierarchical file systems of several popular operating systems such as UNIX and DOS. In DOS, each '\' represents an edge, and in UNIX, it is represented as '/'. Each directory/folder contains subdirectories and files which represent its children (child nodes).



In this family tree structure, each node contains a name (for data) and has one or more links (pointers) that point to the other nodes. It is a general tree structure.

Tree is a dynamic data structure in which nodes are added and removed dynamically during program execution. The nodes are linked to each other through a chain of pointers. Tree data structure is used in various computer applications.

In a tree, any of the nodes cannot be duplicated. In a linked list, each node has a link that points to the next node (points to the previous node as well – in case of a doubly-linked list). In a tree structure, however, each node may point to several other nodes. Therefore, each node of a tree structure stores actual data and links of other related nodes. Examples of a tree can be a company's organizational chart and a family structure (i.e. family tree).

A tree with no nodes is called a *null tree* or *empty tree*. A non-empty tree consists of a *root node* and possibly many levels of additional nodes that form a hierarchy. If the root node is connected by another node, then the root node is called the *parent node* and the connected node is called a *child node*. A typical tree is shown in the following figure:

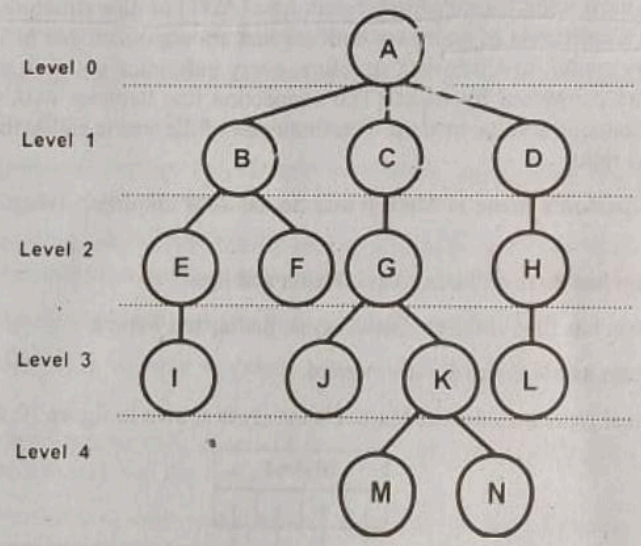



*Figure 10.2: Tree*

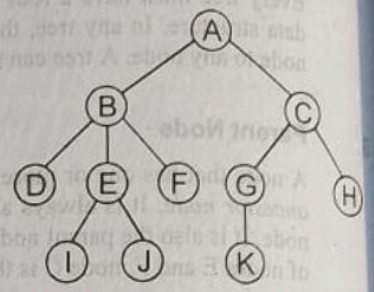
### 10.1.1 Basic Terminologies Related to Trees

Consider the tree shown in figure 10.2 to understand the basic terminologies related to trees. These are as follows:

#### 1. Node

In a tree data structure, every individual element or entity is called a *node*. It holds data and links to other nodes (child nodes). In figure 10.2, all small circles or bubbles represent the nodes of a tree.

Node 'n1' is an *ancestor* of node 'n2' and node 'n2' is a *descendant* of node 'n1' if node 'n1' is either the father of node 'n2' or the father of some ancestor of node 'n2'. In other words, a parent is an ancestor to its children and its children are descendants. For example, in the following tree:




**Ancestors:**

- Nodes E, B, and A are ancestors of node J.
- Nodes C and A are ancestors of node H.
- Node A is an ancestor of both nodes J and H.
- Node A is the ancestor of all other nodes in the tree.

**Descendants:**

- Node H is a descendant of node C.
- Nodes E, I, and J are descendants of node B.
- Every node of the tree other than node A is a descendant of node A.

## 8. Internal Node

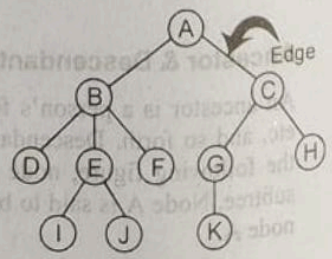
A node that has at least one child (i.e. non-leaf node) is called an *internal node*. It is also called an *interior node*. In other words, all nodes other than leaf nodes are called *internal nodes*. The root node is also said to be an *internal node* if the tree has more than one node. Internal nodes are also called as *non-terminal* nodes.

## 9. Level of a Node

The level of a node represents the rank of hierarchy in a tree. The root node is said to be at Level 0 and the children of the 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 a *Level* and the Level count starts with '0' and is incremented by one at each level (step). For example in the tree of figure 10.2, the nodes I, J, K, and L are at level 3, and M & N are at level 4.

## 10. Edge

A connection from one node to another node is called an *edge*. In other words, a line drawn from a parent node to its child node is said to be an *edge*. The nodes are connected by edges. In a tree with 'N' number of nodes, there will be a maximum of 'N-1' number of edges.



2. **Root Node**

The topmost node in the tree that has no parent node is called the *root node*. It represents the beginning of the tree. Element A in figure 10.2 is the root node of the tree.

Every tree must have a root node. We can say that the root node is the origin of a tree data structure. In any tree, there must be only one root node and one path from the root node to any node. A tree can never have multiple root nodes.

3. **Parent Node**

A node that has one or more child nodes is called a *parent node*. It is also known as an *ancestor node*. It is always above its child nodes. The node A in figure 10.2 is the root node. It is also the parent node of nodes B, C, and D. Similarly, node B is the parent node of nodes E and F, node C is the parent node of node G, node D is the parent node of node H, and so on.

Note that a root node is the only node in the tree which has no parent. Except for the root node, each node of the tree has one parent. Any node cannot have more than one parent.

4. **Child Node**

The node that is directly connected to a parent node is called the *child node*. In figure 10.2, nodes B, C, and D are the child nodes of A. Child nodes are always written one level below the parent.

In a tree, any parent node can have any number of child nodes. All the nodes of a tree except the root node are child nodes.

5. **Siblings**

The nodes which have the same parent are called *siblings*. The nodes E and F in figure 10.2 are siblings as both these nodes are child nodes (children) of node B.

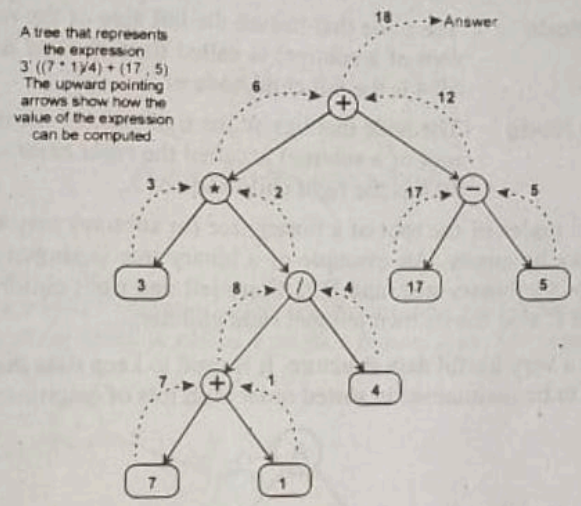
6. **Leaf Node**

A node that does not have any child node (or successor) is called the *leaf node* or a *leaf*. It is also called a *terminal node or an external node*. The nodes I, F, J, M, N, and L in figure 10.2 are terminal nodes or leaves. A tree can have only one root node and many leaf nodes (or leaves).

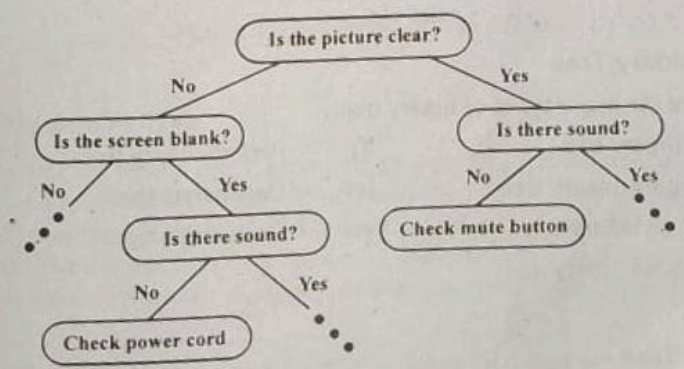
7. **Ancestor & Descendants**

An ancestor is a person's forefather such as his/her father, grandfather or grandmother, etc. and so forth. Descendants are the person's children, grandchildren, and so forth. In the following figure, node A is the root of the tree, and node B is the root of its left subtree. Node A is said to be the *father* of node B and node B is said to be the *left son* of node A.

- **Data Storage:** It is used to organize a large volume of data in a linked structure. Data stored in this manner can be accessed and searched more efficiently.
- **Expression Evaluation:** It can be used to evaluate arithmetic expressions. When a parenthesized expression is represented by a tree, each operand becomes a leaf node and each operator becomes an internal node. We can evaluate an expression very easily as shown here.



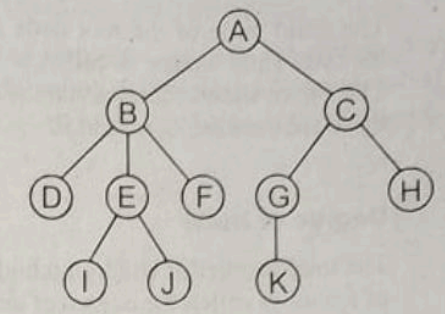

- **Compiler:** It is used for translating program source code into machine code (i.e. binary code).
- **Making Decision:** A tree data structure can be used in a complicated decision-making process. For example, the following tree structure can be used to find out (making decisions) the fault of a television set.




- **Image Processing:** Tree data structure is suitable for digital image processing for visual effects.
- **Searching:** It provides a quick and efficient way of searching (using tree traversals).

## 11. Path & Length of Path

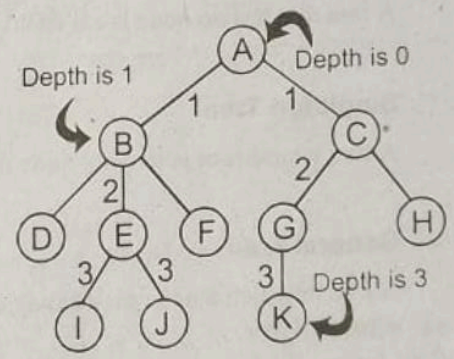
A *path* is a sequence of nodes and edges between two specific nodes. In the following figure, the path between A and J is A – B – E – J. Similarly, the path between C and K is C – G – K. In a tree, there is exactly one path from the root to each node.




The length of a path is the total number of edges in that path. The length of the path from every node to itself is zero. In the following figure, the path between A and J is A – B – E – J and has length 3.

## 12. Depth & Depth of Tree

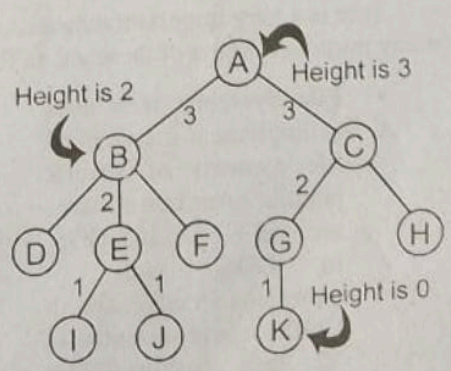
In a tree data structure, the total number of edges (i.e. length of path) from the root node to a particular node is called the *depth* of that node. The depth of the root node is '0'.




In a tree, the total number of edges from the root node to a leaf node that exists on the longest path is said to be the *depth of the tree*. In simple words, the highest depth of any leaf node in a tree is said to be the *depth of that tree*.

## 13. Height & Height of Tree

In a tree data structure, the total number of edges on the longest path between a particular node and a leaf is called the *height* of that node. Thus in a tree, the height of all leaf nodes (leaves) is '0'.




The number of edges on the longest path from the root to the leaf is called the *height of the tree*. In other words, the height of a tree is equal to the height of the root. The height of the tree in figure 10.2 is 4.

## 10.2 BINARY TREE

A tree in which each node can have a maximum of two children (zero, one, or two child nodes) is called a *binary tree*. A binary tree may be empty or non-empty but a general tree cannot be empty. A non-empty binary tree has the following three parts:

**Root Node** The node of the binary tree that has no parent is called the *root node*. It is the starting node of any binary tree.

**Left Child Node** The node that lies on the left side of the root of a binary tree (or root of a subtree) is called the *left child node*. Node B in figure 10.4 is the left child node of A.

**Right Child Node** The node that lies on the right side of the root of a binary tree (or root of a subtree) is called the *right child node*. Node C in figure 10.4 is the right child node of A.

Both the child nodes of the root of a binary tree (or subtree) may be empty or either one of the child nodes may be empty. An example of a binary tree is shown in figure 10.4. In this figure, A is the root of the binary tree, and B & C are left and right children respectively, of the root A. The right child C also has its own left and right children.

Binary tree is a very useful data structure. It is used to keep data that needs to be searched quickly, or that needs to be maintained in sorted order with lots of insertions or deletions.

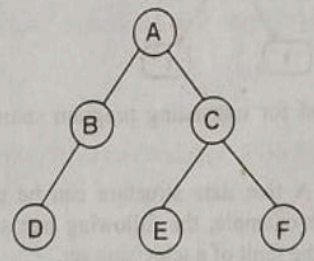



*Figure 10.4: Binary Tree*

### Types of Binary Tree

Following are the major types of binary tree:

| | | | |
|---|---|---|---|
| i) | Full binary tree | ii) | Perfect binary tree |
| iii) | Complete binary tree | iv) | Degenerate tree |
| v) | Skewed binary tree | vi) | Degenerate tree |
| vii) | Extended binary tree | | |

### 10.2.1 Full Binary Tree

A binary tree in which every node other than leaves has two children is called a *full binary tree*. In this type of binary tree, every node has either 0 or 2 children. Full binary tree is used to represent mathematical expressions. The full binary trees are shown in the following figures.

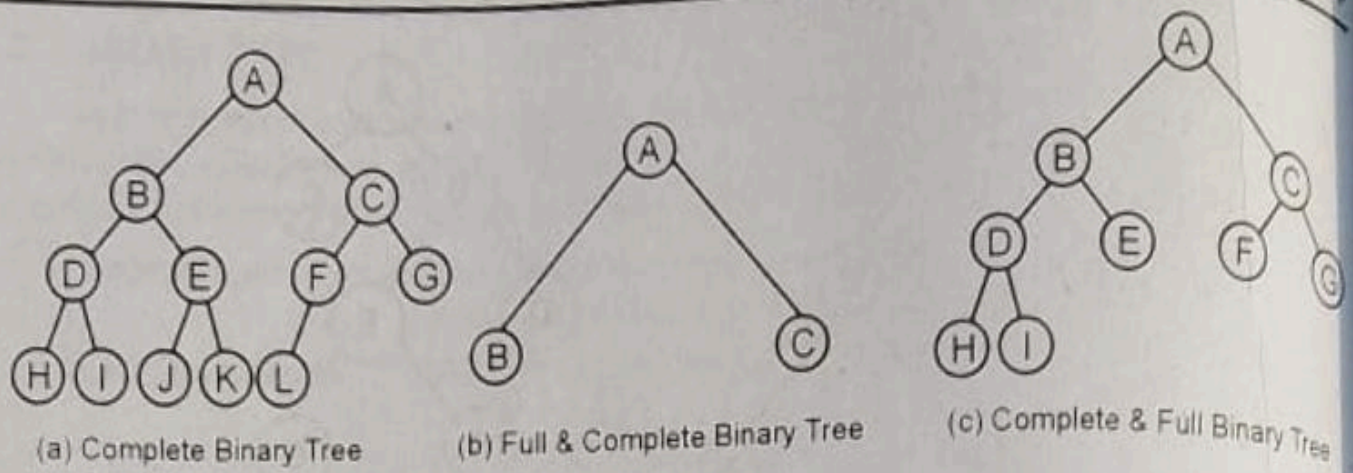



*Figure 10.7: Complete Binary Trees*

### 10.2.4 Degenerate Tree

A tree where for each parent node, there is only one associated child node (either left or right), is called a *degenerate tree*. It is also called a *pathological tree*.

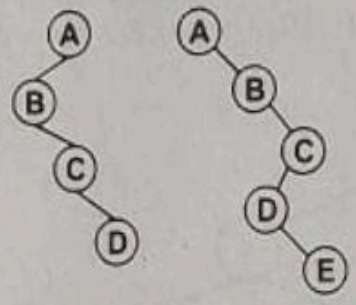



*Figure 10.8: Degenerate Trees*

### 10.2.5 Skewed Binary Tree

A binary tree in which every internal node has only one child is called a *skewed binary tree*. A skewed binary tree is divided into two categories/forms:

- **Left Skewed Binary Tree:** In this form of a skewed binary tree, all internal nodes have the left child without the corresponding right child.
- **Right Skewed Binary Tree:** In this form of a skewed binary tree, all internal nodes have the right child without the corresponding left child.

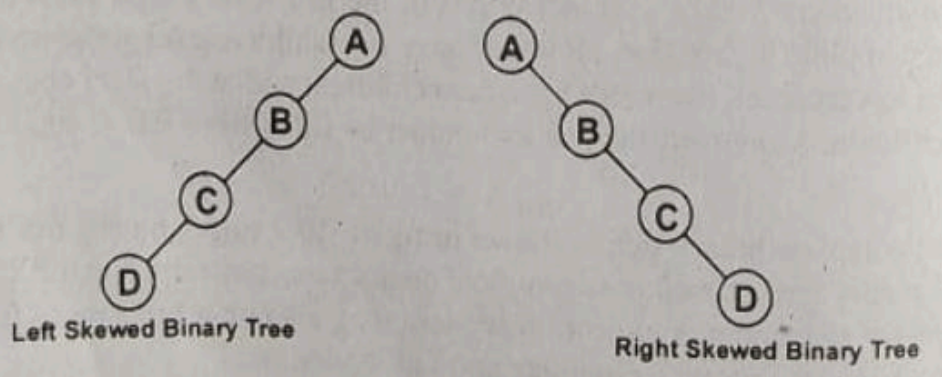



*Figure 10.9: Left and Right Skewed Binary Trees*

Note that all skewed trees are degenerate or pathological trees, but all pathological trees are not skewed trees.

### 10.2.6 Balanced Binary Tree

A binary tree in which the left and right subtrees of every node differ in height by no more than one is called *a balanced binary tree*. It means that a binary tree is said to be a balanced binary tree if the difference between the height of the left subtree and right subtree is maximum one for all the nodes.

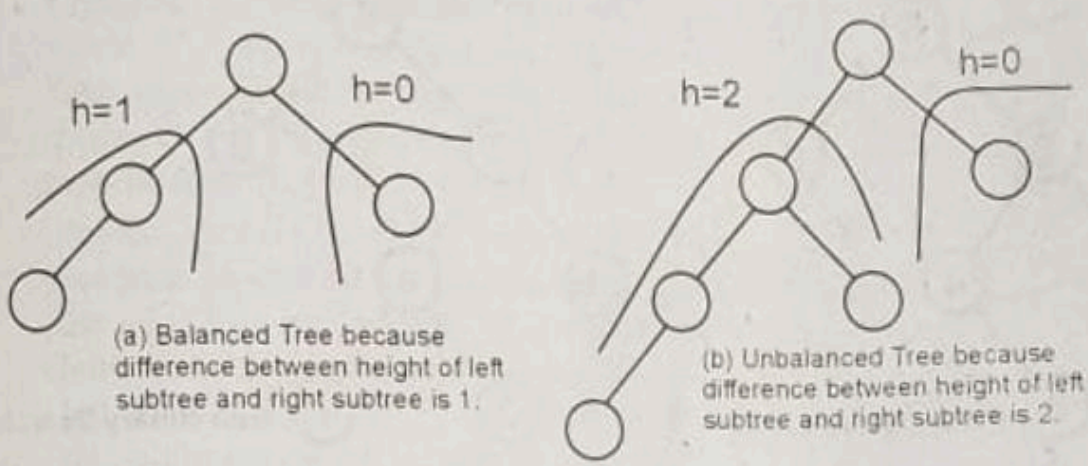



*Figure 10.10: Balanced and Unbalanced Binary Trees*

### 10.2.7 Extended Binary Tree (2–Tree)

A binary tree in which special nodes are added in place of every null subtree is called an *extended binary tree*. It is also called *2–Tree*. The nodes from the original tree are called *internal nodes* and the special nodes are called *external nodes*. The internal node is represented by a circle and the external node is represented by a box. The data can be stored either in the internal nodes or the external nodes.

Every internal node in the extended binary tree has exactly two children and every external node has no children (i.e. external nodes are the leaves of the extended tree). It displays the result which is a *complete binary tree*. Figure 10.11 (a) shows a binary tree before adding special nodes and figure 10.11 (b) extended binary tree.

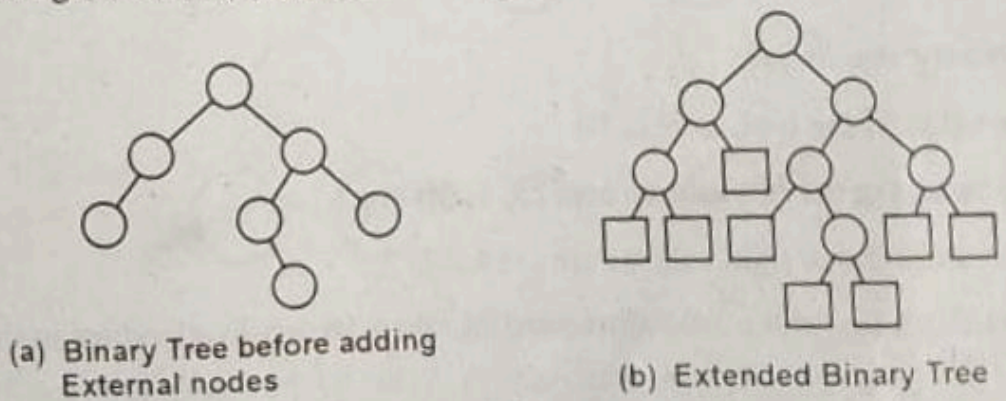



*Figure 10.11: Non-Extended and Extended Binary Tree*

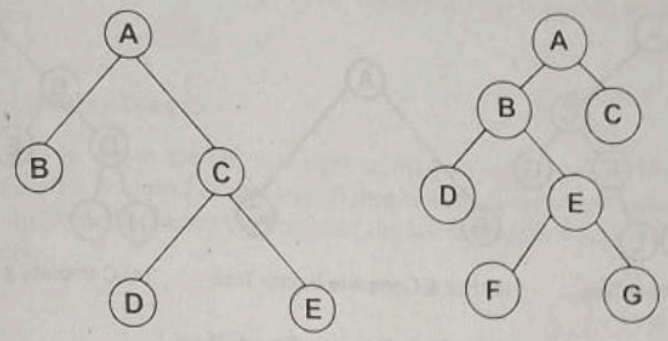



*Figure 10.5: Full Binary Trees*

### 10.2.2 Perfect Binary Tree

A binary tree in which all interior or internal nodes have two children and all leaves have the same depth or the same level is called a *perfect binary tree*. The perfect binary trees are shown in the following figures:

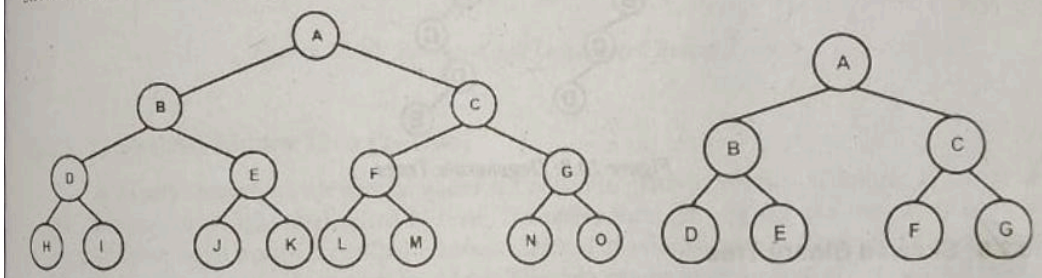



*Figure 10.6: Perfect Full Binary Trees*

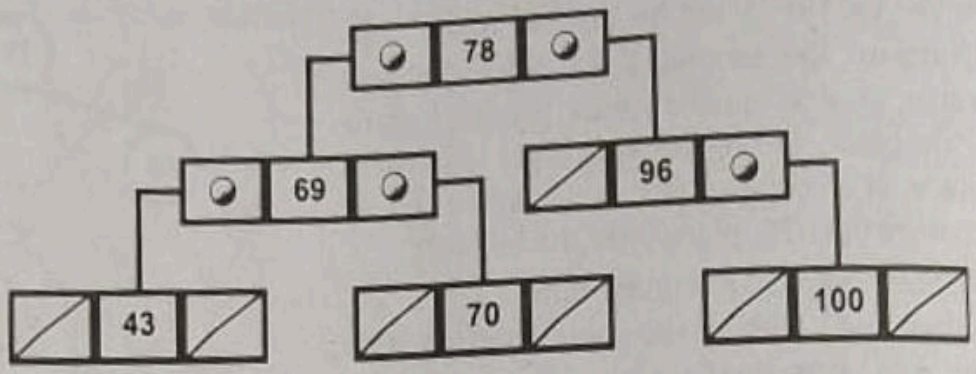
### 10.2.3 Complete Binary Tree

A binary tree in which every level, except possibly the deepest or last, is complete filled, is called a *complete binary tree*. All nodes in the last level are as far left as possible. other words, in a complete binary tree, all nodes have two children except the nodes at the low two levels. At the lowest level, the nodes have zero children, and at the level above that nodes have 0, 1, or 2 children. Moreover, the last level must be filled from left to right without le any gaps.

A typical complete binary tree is shown in figure 10.7 (a). A binary tree shown in 10.7 (b) is a full binary tree as well as a complete binary tree. It means that a full binary t be a complete binary tree and a complete binary tree may also be a full binary tree. But it is always true. Figure 10.7 (c) shows a complete and full binary tree.

### 10.3.1.1 Linked Storage Technique (Using Linked List)

Linked storage technique is the most commonly used method for representing binary trees inside the computer memory. In this technique, the double linked list is used to represent a binary tree. Each node of the binary tree consists of three fields; one for data field and two for pointer fields. The data field is used to store the value of the node. The pointer fields are used to store the memory addresses of the left child node and right child node. A graphical figure of a binary search tree is shown in the following figure using a linked storage technique.




If any node has no child then pointer fields are assigned with value NULL. The pointer field values for both left and right children of leave nodes are always NULL.

The node structure to represent a binary search tree in the memory is as follows:

| Left Child Address | Data | Right Child Address |
|---|---|---|

```
struct node
{
        int data;
        node *left, *right;
};
```

This node structure of the tree is similar to a doubly-linked list. Many of the rules that apply to the linked lists also apply to represent binary trees inside the computer.

### 10.3.1.2 Sequential Storage Technique (using Array)

The sequential storage technique is very simple to represent the tree inside the computer memory. In this technique, a linear array is used to represent the tree inside the computer memory. This technique is efficient if the structure of the tree does not change frequently during its existence in memory. It is best used to represent full binary trees. However, the array is a static data structure.

## 10.3 BINARY SEARCH TREE (BST)

A binary tree in which every node contains smaller values in its left subtree and larger values in its right subtree is called a *binary search tree (BST)*. For example, in BST, all the nodes in the left subtree of the root node contain smaller values than the value of the root node, while all the nodes in the right subtree contain larger values than the value of the root node. Similarly, every node of BST satisfies this condition. In figure 10.12, the tree on the left side is a binary search tree, but the tree on the right is not. The tree on the right has a node with item 7 in the left subtree which is greater than the value of the root node.

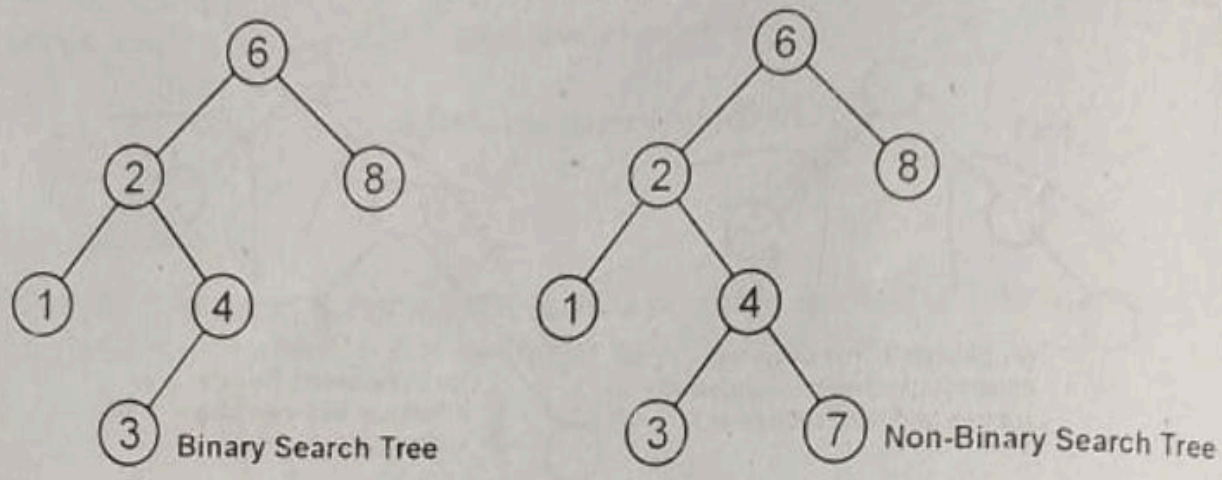
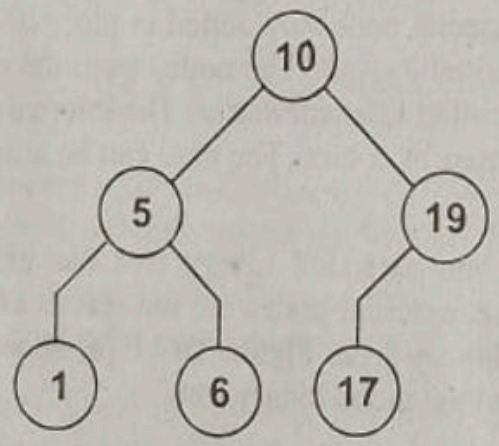



*Figure 10.12: Binary search tree and non-binary search tree*

Consider the following binary search tree:




In this binary search tree:

- The value of the root node is 10.
- The values in the left subtree are: {5, 1, 6}
- The values in the right subtree are: {19, 17}

Note that all values in the left subtree are less than 10, while all values in the right subtree are greater than 10.

The main advantage of a binary search tree is that it is easy to perform operations like creating a new tree, insertion, searching, and deletion of data.

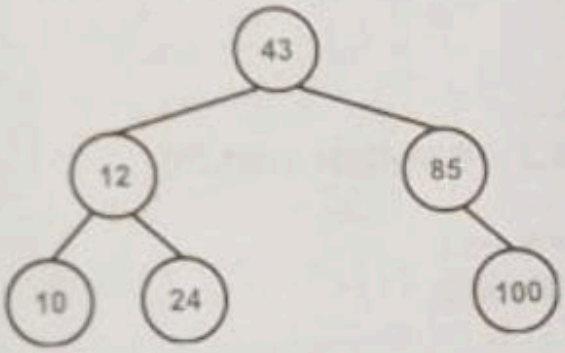
The following rules are used to represent a binary tree using a sequential storage technique (i.e. mapping the values of tree in an array). Suppose the root of a tree is represented by 1 and the number of nodes by K.

- The value of root 'R' is stored in the first element of the array.
- The value of the left child of the node 'K' is stored in element 2*K of the array.
- The value of the right child of the node K is stored in element 2*K+1 of the array.
- A NULL value indicates that the tree or subtree is empty.

The following figure shows a sequential representation of a binary tree using an array. The tree has the values:

43, 12, 85, 10, 24, 100

The binary tree and its sequential representation are shown below:




| Position | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| Elements | 43 | 12 | 85 | 10 | 24 | | 100 |

*Figure: Sequential Representation of Binary Tree*

The linked storage technique is recommended to represent complete or full binary trees. If the tree is not full, a lot of elements in the array remain empty. A binary tree of depth 'n' needs an array of approximately $2^{n+1}$ elements. Thus for a tree that has only 11 nodes and depth 5, the array size required to represent the tree is approximately $2^6 = 64$ elements.

**Program Code 10.1**

```
// Program that creates a binary search tree and displays the data of all nodes.

#include <iostream.h>
#include <conio.h>

        struct node
        {
                int n;
                node *right;
```

⚠ *Every binary search tree is a binary tree but the reverse may not be true (i.e. every binary tree may not be a binary search tree).*

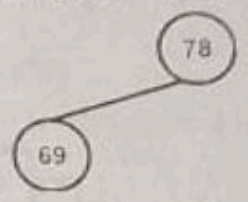
### Constructing Binary Search Tree

Suppose a binary search tree has the following values is to be constructed:

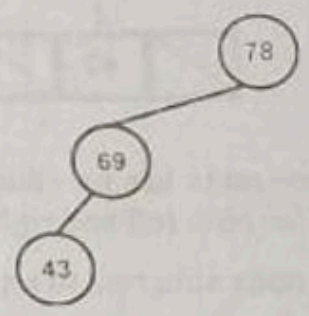
78, 69, 43, 70, 17, 96, 68, 82, 100, 99

The following rules are followed to construct the binary search tree on the given values:

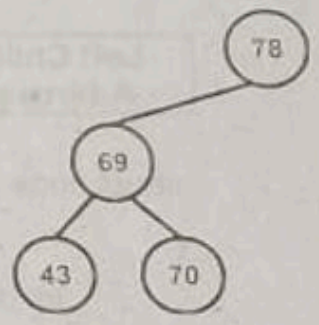
1. Select the first value for the root node.
2. Compare the second value with the root value, if it is smaller than the root value then place it on the left side; otherwise place it on the right side. Since 69 is smaller than 78, place it on the left side.



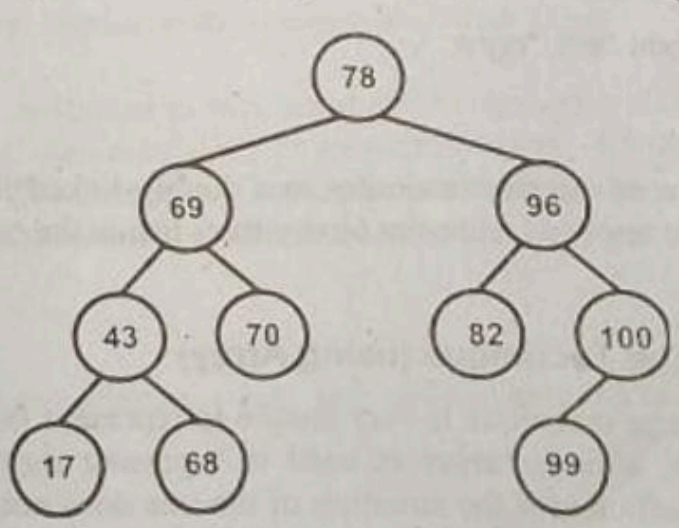

3. Repeat step-2 for the third value i.e. compare 43 with all the values starting from the root. It is smaller than 78, so go to the left side of the root and compare it with the left child node value. It is also smaller than 69, so place it on the left side of this node.




4. Compare the fourth value i.e. 70 with the root value. It is smaller than the root value. Compare it with the left child node, it is larger than 69. Place it to the right of this node.




5. By following the same procedure, the binary search tree for the other given values is constructed as shown in the figure below:




### 10.3.1 Representation of Binary Search Tree in Memory

Different techniques can be used to represent a binary search tree in the computer memory. The most commonly used techniques are as follows:

i) Linked storage technique ii) Sequential storage technique

```
                if(c->n == data)
                {
                        cout<<"Data already exists"<<endl;
                        getch();
                        return;
                }
                if(c->n<data)
                {
                        p = c;
                        c = c->right;
                        row+=2;
                        col+=6;
                }
                else
                {
                        p = c;
                        c = c->left;
                        row+=2;
                        col-=6;
                }
        } //end of while loop
                // create new node and assign data to it
        t = new node;
        t->n = data;
        t->left = t->right = NULL;
        gotoxy(col, row);
        cout<<data;
        if(p == NULL)
                p = t;
        else if(p->n>data)
                p->left = t;
            else
                p->right = t;
    }
} //end of create()
```

**Output of the program**

```
        22
    21      33
11              76
        43      88
    34      55      89
```

```
        node *left;
    };

    class bst
    {
        private:
            node *s,*c,*t;
        public:
        bst(void)
        {
            s = NULL;
        }
        void create(int);
    };

main()
{
    bst  obj;
    int  x[] = {22,33,21,76,88,43,34,11,89,55};
    clrscr();
    for(int i = 0; i<= 9; i++)
        obj.create(x[i]);
} //end of main()

    // member function to create tree
void bst::create(int data)
{
    int col = 40,row = 1;

        // to create root node and assign data to it
    if(s == NULL)
    {
        s = new node;
        s->n = data;
        s->left = s->right = NULL;
        gotoxy(col, row);
        cout<<data;
    }
    else
    {
        node *p;
            c = s;

            // go to proper position to add node
        while(c!= NULL)
        {
```

### 10.3.2 Operations on Binary Search Trees

Like other data structures, different operations can be performed on the binary search trees too. The most common of these operations are as under:

i) Search ii) Traversing

iii) Insertion iv) Deletion

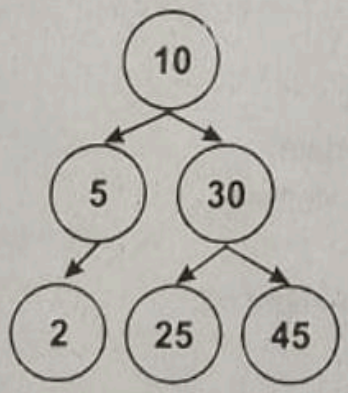
#### 10.3.2.1 Search Operation on BST

Binary search trees can be searched using different methods. Usually, the recursive search method is used to search the binary search trees. It is the easiest way to search a binary tree.

The following steps are taken to search a given value N in the binary search tree using the recursive search method:

1. The search process starts from the root of the tree.
2. If the root value is Null, the Tree is empty and the search process terminates.
3. If the given value N is equal to the value of the root then the search is successful and the searching process terminates.
4. If given value N is smaller than the value of the root, only the left subtree is searched recursively.
5. If given value N is larger than the value of the root, only the right subtree is searched recursively.
6. If the value is found then the search process terminates; otherwise, the whole subtree is searched until its leaves are reached.

Consider the following binary search tree:




Suppose we want to search a node that has value 5. We have to perform the following steps to search this value:

- We will start the search process from the root node.
- The value 5 is smaller than the value of the root node (i.e. 5 < 10). This value will be in the left subtree.
- The value 5 is equal to the value of the root node of the left subtree (i.e. 5 = 5). The required value is found, so the search process is terminated.

**Algorithm – Searching BST**

**Write an algorithm that searches a node in a binary search tree. Suppose the starting address of the tree is stored in pointer BST. The pointer CUR stores the address of the current node. DATA represents the data field and L & R represent left and right pointers respectively of a node of the tree. Value N represents the value that is to be searched.**

```
1.    START
2.    INPUT value to search in N
3.    IF BST = NULL THEN
            PRINT "Tree is Empty"
            RETURN
      END IF
4.    CUR = BST
5.    REPEAT STEP-6 TO 8 WHILE CUR != NULL
6.    IF N = CUR->DATA THEN
            PRINT "Number is found"
            RETURN
      END IF
7.    IF N < CUR->DATA THEN
            CUR = CUR->L
      ELSE
            CUR =  CUR->R
      END IF
8.    IF CUR = NULL THEN PRINT "Number not fund"
      [End of Step-5 Loop]
9.    EXIT
```

### Complexity Analysis of Search Operation on BST

Searching a node in a BST is a binary process. The search algorithm of BST starts the search process from the root node toward the leaf nodes. It takes time proportional to the tree's *height*. On average, a binary search tree with $n$ nodes has $\Theta(\log n)$ height i.e.

$$n = 1 + 2 + 4 + \ldots + 2^h - 1 + 2^h = 2^{h+1} - 1$$

Solving this with respect to 'h', we obtain:

$$h = O(\log n)$$

The binary search tree with $n$ nodes has a minimum of O(log n) levels, it takes at least (log n) comparisons to find a particular node. Thus, the average-case time complexity of the search operation of BST is $\Theta(\log n)$. However, in the worst-case, the binary search tree can have $O(n)$ height. We have to traverse all the nodes of height O(n) to find the required node. Thus, the worst-case time complexity of the search operation of BST is O(n).

The space complexity of the search operation of BST is O(n).

**Program Code 10.2**

```
//Program to create a binary tree and search a given value in it.

#include <iostream.h>
#include <conio.h>

        struct node
        {
                int data;
                node *right;
                node *left;
        };

        class bst
        {
                private:
                        node *start, *cur, *parent, *temp;
                public:
                bst(void)
                {
                        start = NULL;
                }
                void create(int);
                node* search(node*, int);
        };

void main(void)
{
        bst  obj;
        node *res, *t;
        res = t = NULL;
        int  x[] = {22, 33, 21, 76, 88, 43, 34, 11, 89, 55}, val;
        clrscr();
        for(int i = 0; i<=9; i++)
                obj.create(x[i]);
        cout<<"\nEnter value to search :";
        cin>>val;
        res = obj.search(t, val);
        if(res== NULL)
                cout<<"Node not found in this tree :"<<endl;
        else if(res->left== NULL && res->right== NULL)
                cout <<"Node found in this tree. It is a leaf node :"<<endl;
        else if(res->left!= NULL && res-> right!= NULL)
                cout<<"Node found in this tree. It is two children node :"<<endl;
        else
                cout<<"Node found in this tree. It is one child node :"<<endl;
        getch();
} //end of main()

        // member function to create a binary search tree
void bst::create(int n)
```

```
{       int col = 40, row = 1;
        // to create root node and assign data to it
        if(start == NULL)
        {
                start = new node;
                start->data = n;
                start->left =  start->right = NULL;
                gotoxy(col, row);
                cout<<n;
        }
        else
        {
                cur= start;
                        // go to proper position to add node
                while(cur!=NULL)
                {
                        if(cur->data == n)
                        {
                                cout<<"Data already exists"<<endl;
                                return;
                        }
                        if(cur->data < n)
                        {
                                parent = cur;
                                cur = cur->right;
                                row+= 2;
                                col+= 6;
                        }
                        else
                        {
                                parent = cur;
                                cur = cur->left;
                                row+= 2;
                                col-= 6;
                        }
                }
                        // create new node and assign data to it
                temp = new node;
                temp->data = n;
                temp->left = temp->right = NULL;
                gotoxy(col, row);
                cout<<n;
                if(parent==NULL)
                        parent = temp;
                else if(parent->data>n)
```

```
                    parent->left = temp;
            else
                    parent->right = temp;
        }
} //end of create()

        // member function to search a node in the tree
node* bst::search(node* s, int x)
{
        s = start;
        if(s == NULL)
        {
                cout<<"Tree is empty";
                getch();
                return NULL;
        }
        else
        {
                cur = s;
                        // search value
                while(cur!= NULL)
                {
                        if(cur->data == x)
                        {
                                return cur;
                        }
                        if(cur->data < x)
                                cur = cur->right;
                        else
                                cur = cur->left;
                }
        }
        return cur;
} //end of search()
```

### 10.3.2.2 Traversal Operation

Traversal of a binary search tree is one of the most important operations. It is a proce visiting each node in the tree exactly once. It is also called *visiting*. There are three po methods used for visiting the binary search tree. These methods are as follows:

1. Inorder Traversal
2. Preorder Traversal
3. Postorder Traversal

**Step-3:** Loop While TOP != NULL

1. The top value of the stack is popped and stored into P, i.e. P = 109. The data of address P, i.e. D is visited.
2. The top value of the stack is popped and stored into P, i.e. P = 102. The data of address P i.e. B is visited.
3. The right address of P is not NULL. Thus loop at step-2 is executed. The values in the stack will be 106 and 110.
4. The top value of the stack is popped and stored into P i.e. P = 110. The data of address P i.e. E is visited.
5. Top value of the stack is popped and stored into P, i.e. P=106. The data of address P, i.e. A is visited. The right address of P is not NULL. Thus step-2 is performed and the value in the stack is 108.
6. The top value of the stack is popped and stored into P, i.e. P=108. The data of address P, i.e. C is visited. The right address of P is not NULL. Thus step-2 is performed and the new value of the stack is 112.
7. The top value of the stack is popped and stored into P, i.e. P= 112. The data of address P, i.e. F is visited.
8. The stack is NULL. The loop condition is satisfied, so the loop terminates.

**Program Code 10.3**

```
// Program to create and visit a binary search tree using inorder
// traversal method with the help of stack.

#include <iostream.h>
#include <conio.h>

        struct node
        {
                int n;
                node *right;
                node *left;
        };

        class bst
        {
                private:
                        node *s, *c, *t, *stack[10];
                        int top;
                public:
                bst(void)
                {
```

```
                s = NULL;
                top = -1;
        }
        void create(int);
        void inorder(void);
        void push (node *);
};

void main(void)
{
        bst  obj;
        int  x[] = {22,33,21,76,88,43,34,11,89,55};
        clrscr();
        for(int i = 0; i<=9; i++)
                obj.create(x[i]);
        obj.inorder();
        getch();
} //end of main()

        // member function to create tree
void bst::create(int data)
{
        int col = 40, row = 1;

                // to create root node and assign data to it
        if(s == NULL)
        {
                s = new node;
                s->n = data;
                s->left = s->right = NULL;
                gotoxy(col, row);
                cout<<data;
        }
        else
        {
                node *p;
                c = s;
                        // go to proper position to add node
                while(c!=NULL)
                {
                        if(c->n == data)
                        {
                                cout<<"Data already exists"<<endl;
                                return;
                        }
                        if(c->n < data)
```

## Inorder Traversal

1. Using the inorder traversal method, the non-empty binary search tree is visited in the following sequence:

   i) Visit the left sub-tree in inorder

   ii) Visit the root

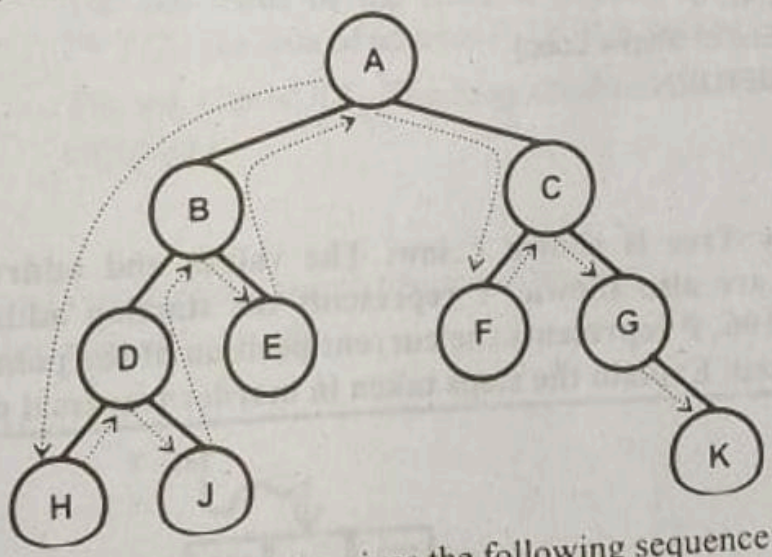
   iii) Visit the right sub-tree in inorder

When a binary search tree is visited inorder, and the data of the node is accessed, then the data is printed (or displayed) in ascending order.

Usually, a stack is used to implement the inorder traversal method. Moreover, a pointer variable is used to hold the location of the node being visited currently.

Following steps are performed to visit a binary search tree in inorder using stack:

1. Proceed down the left-most path from the root node to the last left node. Push addresses of each left node onto the stack.
2. Pop the top address from the stack. It will be the address of the leftmost node. Visit the node, i.e. access the data of this node to print it.
3. If the node being visited has a right child node, then repeat steps 1 and 2.
4. The above procedure is repeated until all the nodes of the tree are visited.




Inorder traversal of the above binary tree gives the following sequence:

H D J B E A F C G K

## Algorithm – Inorder Traversal of Binary Tree

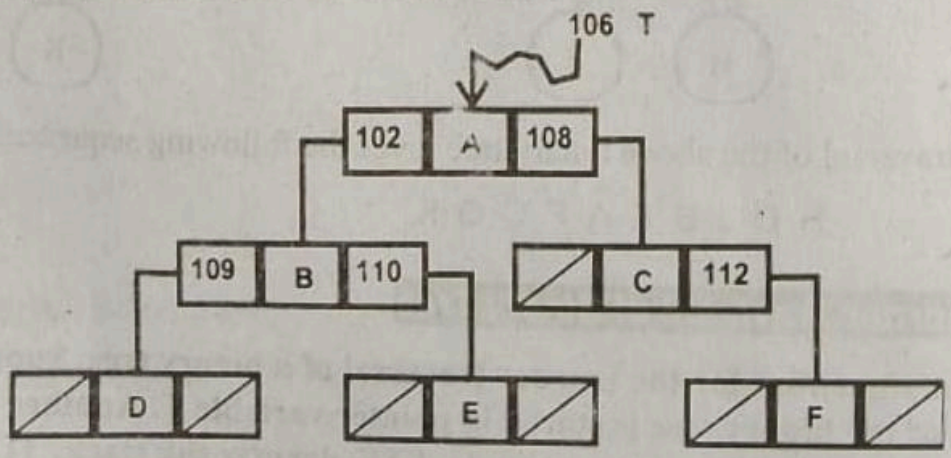
**Write an algorithm for the inorder traversal of a binary tree. Suppose the root node address of the binary tree is stored in pointer variable T. Another pointer variable P holds the address of the current node. STK denotes the stack. TOP denotes the top position of stack STK. PUSH function is used to insert a value onto the stack. The POP function is used to get or remove the value from the top of the stack. Assume L and R represent pointer variables to hold addresses of left and right children.**

1. [Read starting address of Tree]

```
        IF T = NULL THEN
                PRINT "Binary Tree is Empty"
                RETURN
        ELSE            [initialize the top of stack and assign the address of root to P]
                TOP = 0
                P = T
        END IF
2.      REPEAT STEP-3 WHILE P! = NULL
        [Proceed to the left most node by pushing left node to stack]
3.      CALL PUSH(STK, TOP, P)
        P = P -> L
        [End of Step-2 loop]
4.      REPEAT STEP-5 to 7 WHILE TOP > 0
        [Pop top value of stack STK into P to visit the Tree Inorder]
5.      P = POP(STK, TOP)
6.      PRINT P->DATA
7.      IF P->R!= NULL THEN
                P = P->R
                PERFORM STEP-2
        END IF
        [End of Step-4 Loop]
8.      RETURN
```

**Example:**

**A Binary Tree is shown below. The values and addresses of its left and right children are also shown. T represents the starting address of the tree and has a value of 106. P represents the current position of the pointer through which the tree is traversed. Explain the steps taken in inorder traversal of the tree using a stack.**




**Step-1:** Assign the starting address of the tree to P, thus P = 106.

**Step-2:** Loop While P != NULL

Start from the root node, proceed down the left-most path to the last left node while pushing addresses of each left node onto the stack. At the end of the loop, the value in the stack will be 106,102,109, and P = NULL

```
                {
                        p = c;
                        c = c->right;
                        row+= 2;
                        col+= 6;
                }
                else
                {
                        p = c;
                        c = c->left;
                        row+= 2;
                        col-= 6;
                }
        }
                // create new node and assign data to it
        t = new node;
        t->n = data;
        t->left = NULL;
        t->right = NULL;
        gotoxy(col, row);
        cout<<data;
        if(p == NULL)
                p = t;
        else if(p->n > data)
                p->left = t;
            else
                p->right = t;
    }
} //end of create()

    // definition of inorder function
void bst::inorder(void)
{
            // read starting address of tree
    if(s == NULL)
    {
            cout<<"Tree is empty\n";
            return;
    }
    else
    {
            cout<<endl<<endl;
            cout<<"Traversal in inorder\n";
            c = s;
            push(c);
            cout<<endl;
```

```
            else
                parent->right = temp;
        }
    } //end of create()
```

2. **Preorder Traversal**

Through the preorder traversal method, a non-empty binary search tree is visited in the following order;

(i) Visit the root.

(ii) Visit the left subtree in preorder.

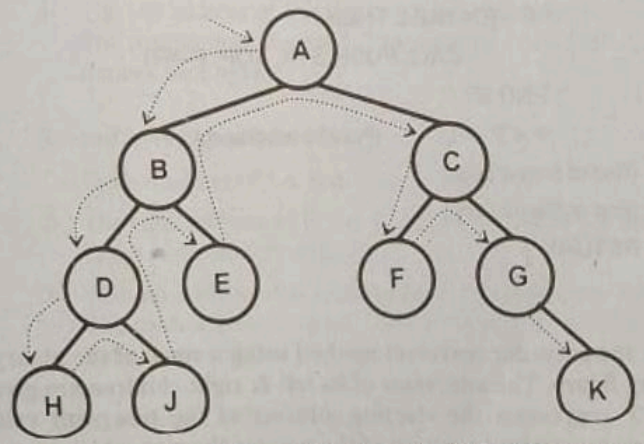
(iii) Visit the right sub-tree in preorder.

Usually, a stack is used to implement the preorder traversal method. Moreover, a pointer variable is used to hold the location of the node being visited currently.

To visit a binary search tree in preorder using stack, follow these steps:

1. Root node is visited. The address of its right child node (if any) is pushed onto the stack.
2. The left sub-tree of the node is visited. The address of its right child node (if any) is pushed onto the stack.
3. After visiting the last leftmost child node, the top value from the stack is popped and visited in preorder sequence as explained in steps 1 and 2.

Consider the following binary tree:




Preorder traversal of the above binary tree gives the following sequence:

A B D H J E C F G K

```
                while(top>=0)
                {
                        c = stack[top];
                        top--;
                        cout<<c->n<<"\t";
                        if(c->right!= NULL)
                        push(c->right);
                }
        }
} //end of inorder()

        // definition of push function
void bst::push(node *cc)
{
        while(cc!=NULL)
        {
                top++;
                stack[top] = cc;
                cc = cc->left;
        }
} //end of push ()
```

**Output of the program**

```
            22
        21      33
    11              76
            43      88
        34      55      89
```

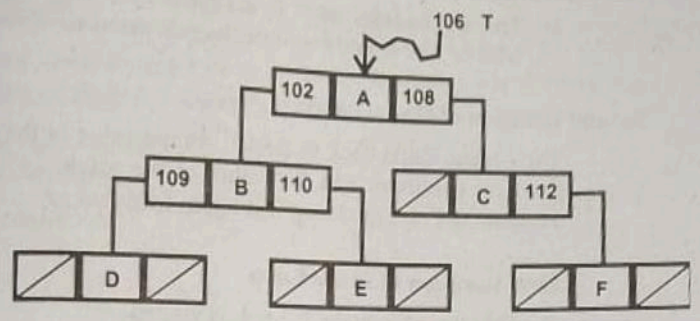
Traversal in inorder

11 21 22 33 34 43 55 76 88 89

**Program Code 10.4**

```
// Program to create and visit the binary search tree using inorder recursively
#include <iostream.h>
#include <conio.h>

        struct node
        {
                int data;
                node *left;
                node *right;
        };

        class tree
        {
```

```
        private:
                node *start, *cur, *temp;
                int top;
        public:
        tree()
        {
                start = NULL;
        }
        void create(int);
        void inorder(void);
        void inord(node *s);
    };

void main(void)
{
        tree obj;
        int val, p;
        clrscr();
        cout<<"Enter ten values \n";
        for(int i = 1; i<=10; i++)
        {
                cin>>val;
                obj.create(val);
        }
        obj.inorder();
        getch();
} //end of main()

        //member function to call the recursive function "inord"
void tree::inorder()
{
        inord(start);
} //end of inorder()

        // definition of inord recursive function
void tree::inord(node *s)
{
        if(s!=NULL)
        {
                inord(s->left);
                cout<<s->data<<"\t";
                inord(s->right);
        }
} //end of inord()
```




**Step-1:** Starting Address of the Tree (i.e. 106) is pushed onto the stack.

The top value of stack = 106

**Step-2:** Execute Outer Loop WHILE TOP !=NULL

**First Iteration of Outer Loop**

The top value of the stack is popped and the value is assigned to P. Thus P = 106.

**Step-3:** Inner Loop While P! = NULL

**First Iteration of Inner Loop**

1- Data of address P, i.e. A is visited.

2- The right address of P, i.e. 108 is pushed onto the stack. The top value of the stack is 108.

3- The left address of P is assigned to P and the new value of P is 102. The control goes to step-3. The value of P is not NULL and the loop iterates once again.

**Second iteration of Inner Loop**

1- Data of address P i.e. B is visited.

2- The right address of P, i.e. 110 is pushed onto the stack. The values in the stack are 108 and 110.

3- The left address of P is assigned to P and the new value of P is 109. The control goes to step-3. The value of P is not NULL and the loop iterates once again.

**Third iteration of Inner Loop**

1- The data of address P, i.e. D is visited.

2- The right address of P is NULL. It is not pushed onto the stack. The values in the stack remain 108 and 110.

3- The left address of P is assigned to P and the new value of P is NULL. The condition of the loop is satisfied. The loop terminates.

**Second Iteration of Outer Loop**

The control shifts back to step-2. As the value in the stack is not NULL, the loop continues and the value of the stack, i.e. 110 is popped and assigned to P. Loop at step-3 is again executed as:

**First Iteration of Inner Loop**

1- The data of address P, i.e. E is visited.

2- The right address of P is NULL. It is not pushed onto the stack. The value of the stack remains 108.

3- The left address of P is assigned to P and the new value of P is NULL. The condition of the loop is satisfied. The loop terminates.

**Third Iteration of Outer Loop**

The control shifts back to step-2. As the value in the stack is not NULL, the loop continues and the value of the stack, i.e. 108 is popped and assigned to P. Loop at step-3 is again executed as:

**First Iteration of Inner Loop**

1. The data of address P, i.e. C is visited.
2. The right address of P i.e. 112 is pushed onto the stack. The top value of the stack remains 112.
3. The left address of P is assigned to P and the new value of P is NULL. The condition of the loop is satisfied. The loop terminates.

**Fourth Iteration of Outer Loop**

The control shifts back to step-2. As the value of the stack is not NULL, the loop continues and the value of the stack, i.e. 112 is popped and assigned to P. Loop at step-3 is again executed as:

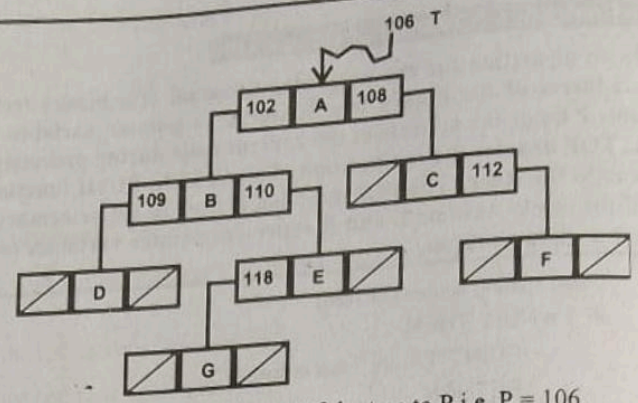
**First Iteration of Inner Loop**

1. The data of address P, i.e. F is visited.
2. The right address of P, i.e. NULL is pushed onto the stack. The top value of the stack is NULL.
3. The left address of P is assigned to P and the new value of P is NULL. The condition of the loop is satisfied. The loop terminates.

The control shifts back to step-2. As the value of the stack is NULL, the loop condition is satisfied. The loop terminates.

```
#include <conio.h>
        struct node
        {
                int data;
                node *left;
                node *right;
        };
        class tree
        {
                private:
                        node *start, *cur, *temp;
                        int top;
                public:
                tree() { start = NULL; }
                void create(int);
                void preorder(void);
                void preord(node *s);
        };

void main(void)
{
        tree obj;
        int val, p;
        clrscr();
        cout<<"Enter ten values\n";
        for(int i = 1; i<=10; i++)
        {
                cin>>val;
                obj.create(val);
        }
        obj.preorder();
        getch();
} //end of main()

        //member function to call the recursive function preord
void tree::preorder()
{
        preord(start);
} //end of preorder()

        //preord recursive function to visit tree in preorder traversal
void tree::preord(node *s)
{
        if(s!=NULL)
        {
```

```
//member function to create and assign data into tree nodes
void tree::create(int x)
{
        if(start==NULL)
        {
                //create root node and assign data
                start = new node;
                start->data = x;
                start->left = NULL;
                start->right = NULL;
        }
        else
        {
                node *parent;
                cur = start;
                //search value in tree and go to proper position to insert
                while(cur!=NULL)
                {
                        if(cur->data == x)
                        {
                                cout<<"Data already exist/n";
                                return;
                        }
                        if(x > cur->data)
                        {
                                parent = cur;
                                cur = cur->right;
                        }
                        else
                        {
                                parent = cur;
                                cur = cur->left;
                        }
                } // end of while loop
                // create and insert new node
                temp = new node;
                temp->data = x;
                temp->left = NULL;
                temp->right = NULL;
                if(parent == NULL)
                        parent = temp;
                else if(parent->data > x)
                        parent->left = temp;
```




**Step-1:** Assign the starting address of the tree to P i.e. P = 106.

**Step-2:** Loop While P ! = NULL

Starting from the root node, proceed down the leftmost path to the last left node while pushing addresses of each left node onto the stack. At the end of the loop, the values in the stack will be:

106, 102, 109 and P = NULL

**Step-3:** Loop While TOP ! = NULL

1. The top value of the stack is popped and stored into P, i.e. P = 109.

   The data of address P, i.e. D is visited.

   The right address of P is not NULL. Thus step-2 is executed and the values in the stack at the end of the loop will be: 106, 102, 110, and 118.

2. The top value of the stack is popped and stored into P, i.e. P = 118.

   The data of address P, i.e. G is visited.

3. The top value of the stack is popped and stored into P, i.e. P = 110.

   The data of address P, i.e. E is visited.

4. The top value of the stack is popped and stored into P, i.e. P = 102.

   The data of address P i.e. B is visited.

   The right address of P is not NULL. Thus step-2 is performed and new values in the stack are 106, 108, and 112.

5. The top value of the stack is popped and stored into P, i.e. P = 112.

   The data of address P, i.e. F is visited.

6. The top value of the stack is popped and stored into P, i.e. P = 108.

   The data of address P, i.e. C is visited.

7. The top value of the stack is popped and stored into P, i.e. P = 106.

```
        else if(parent->data > x)
                parent->left = temp;
        else
                parent->right = temp;
    }
} //end of create()
```

### 3. Postorder Traversal

Using the postorder traversal method, the non-empty binary search tree is visited as follows;

(i) Visit the left subtree in postorder.

(ii) Visit the right sub-tree in postorder.

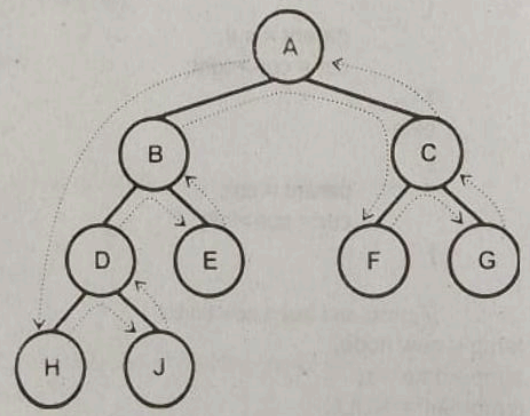
(iii) Visit the root.

Usually, a stack is used to implement the postorder traversal method. Moreover, a pointer variable is used to hold the location of the node being visited currently.

To visit a binary search tree in a postorder using stack, follow these steps:

1. From the root node, move toward the left node and push the address of each node onto the stack. Continue this process until the last leftmost node is reached.
2. When the last left node is reached, pop the top address from the stack and visit the node. If the sibling right node exists then proceed to the right node and repeat step-1.
3. Visit the root node at the end.

Consider the following binary tree:




Postorder traversal of the tree gives the following sequence:

H J D E B F G C A

```
                cout<<s->data<<"\t";
                preord(s->left);
                preord(s->right);
        }
} //end of preord()

        //member function to create and assign data into tree nodes
void tree::create(int x)
{
        if(start == NULL)
        {
                //create root node and assign data
                start = new node;
                start->data = x;
                start->left = start->right = NULL;
        }
        else
        {
                node *parent;
                cur = start;
                // search value in tree and go to proper position to insert
                while(cur!=NULL)
                {
                        if(cur->data == x)
                        {
                                cout<<"Data already exist";
                                return;
                        }
                        if(cur->data < x)
                        {
                                parent = cur;
                                cur = cur->right;
                        }
                        else
                        {
                                parent = cur;
                                cur = cur->left;
                        }
                }
                // create and insert new nodes
                temp = new node;
                temp->data = x;
                temp->left = NULL;
                temp->right = NULL;
                if(parent == NULL)
                        parent = temp;
```

The data of address P, i.e. A is visited at the end.

The stack is NULL. The loop condition is satisfied and the loop terminates.

**Program Code 10.7**

```
// Program to create and visit binary search tree postorder recursively

#include <iostream.h>
#include <conio.h>

        struct node
        {
                int data;
                node *left;
                node *right;
        };

        class tree
        {
                private:
                        node *start, *cur, *temp;
                public:
                tree(){ start = NULL; }
                void create(int);
                void postorder(void);
                void post(node *s);
        };

void main(void)
{
        tree obj;
        int val, p;
        clrscr();
        cout<<"Enter ten values \n";
        for(int i = 1; i<=10; i++)
        {
                cin>>val;
                obj.create(val);
        }
        obj.postorder();
        getch();
} //end of main()

        // postorder recursive function to call recursive function
void tree::postorder()
{
```

## Algorithm – Preorder Traversal of Binary Tree

**Write an algorithm for the preorder traversal of the binary search tree. Take a binary tree having its root node address stored in pointer variable T and pointer variable P holding the address of the current node during processing. STK denotes the stack and TOP denotes the top position of the stack STK. The function PUSH is used to insert the value in the stack STK and the function POP is used to remove the value from the top of the stack. Assume L and R represent pointer variables that hold addresses of left & right children.**

```
1.  [Read starting address of BST]
    IF T = NULL THEN
            PRINT "Binary Tree is Empty"
            RETURN
    ELSE                [place pointer to root of Tree on top of stack STK]
            TOP = 0
            CALL PUSH(STK, TOP, T)
    END IF
2.  REPEAT STEP-3 TO 4 WHILE TOP > 0
3.  [Pop top value of stack STK into P to visit Tree in preorder]
    P = POP(STK, TOP)
4.  [Visit the left most nodes of the tree or subtree and assign addresses of each right node to
    top of the stack]
    REPEAT WHILE P!= NULL
          PRINT P->DATA [Print data of node]
          IF P->R!= NULL THEN
                  CALL PUSH(STK, TOP, P->R)
          END IF
          P = P -> L        [Point to left branch]
    [End of Step-4 Loop]
    [End of Step-2 Loop]
5.  RETURN
```

**Example:**

**Explain the preorder traversal method using a stack of the binary tree shown in the following figure. The addresses of its left & right children are given along with node values. T represents the starting address of the tree with value 106. Assume P represents the current position of the pointer through which the tree is visited.**

**Program Code 10.5**

```
// Program to create and visit the binary search tree using preorder
// traversal method with the help of stack.
#include <iostream.h>
#include <conio.h>
        struct node
        {
                int n;
                node *right;
                node *left;
        };

        class bst
        {
                private:
                        node *s, *c, *t, *stack[10];
                        int top;
                public:
                bst(void)
                {
                        s = NULL;
                        top = -1;
                }
                void create(int);
                void preorder(void);
        };
void main(void)
{
        bst  obj;
        int  x[] = {22, 33, 21, 76, 88, 43, 34, 11, 89, 55};
        clrscr();
        for(int i = 0; i<=9; i++)
                obj.create(x[i]);
        obj.preorder();
        getch();
} //end of main()

        // member function to create tree
void bst::create(int data)
{
        int col = 40, row = 1;
                // to create root node and assign data to it
        if(s == NULL)
        {
                s = new node;
```

```
                parent->left = temp;
            else
                parent->right = temp;
        }
} //end of create()

        //member function to call the recursive function "inord"
void bst::inorder()
{
        inord(start);
} //end of inorder()
        // definition of inord recursive function
void bst::inord(node *ss)
{
        if(ss!=NULL)
        {
                inord(ss->left);
                cout<<ss->data<<", ";
                inord(ss->right);
        }
} //end of inord()

        // definition of insert() function.
void bst::insert (int n)
{
        if(start== NULL)
        {
                start = new node;
                start->data = n;
                start->left = start->right = NULL;
        }
        else
        {
                cur = start;
                        // search value in the tree
                while(cur!= NULL)
                {
                        if(cur->data == n)
                                return;
                        if(cur->data < n)
                        {
                                parent = cur;
                                cur = cur->right;
                        }
                        else
                        {
```

## Algorithm – Postorder Traversal of Binary Tree

**Write an algorithm for the postorder traversal of a binary tree. Suppose the root node address of the binary tree is stored in pointer variable T. Another pointer variable P holds the address of the current node during processing. STK denotes the stack. TOP denotes the top position of stack STK. PUSH function is used to insert a value onto the stack. The POP function is used to get or remove the value from the top of the stack. Assume L and R represent pointer variables to hold the address of the left & right children.**

```
1.  [Read starting address of Tree]
    IF T = NULL THEN
            PRINT "Binary Tree is Empty"
            RETURN
    ELSE            [Initialize top of stack and assign root address to P]
            TOP = 0
            P = T
    END IF
2.  [Proceed to left most node by pushing left node onto stack]
    REPEAT STEP-3 to 4 WHILE P!= NULL
3.  CALL PUSH(STK, TOP, P)
4.  IF P->L = NULL THEN
            P = P->R
    ELSE
            P = P->L
    END IF
    [End of step-2 loop]
5.  [Pop top value of stack into P to visit the tree in postorder sequence]
    REPEAT STEPS-6 to 8 WHILE TOP > 0
6.  P = POP(STK, TOP)
7.  PRINT P->Data
8.  IF P->R!= NULL THEN
            P = P->R
            PERFORM STEP-2
    END IF
    [End of step-5 Loop]
9.  RETURN
```

**Example:**

**A Binary Tree is given below. The values and addresses of its left & right children are also shown. T represents the starting address of the tree with value 106. P represents the current position of the pointer through which the tree is traversed. Explain the steps to traverse the tree in postorder sequence using a stack.**

### Complexity Analysis of Insertion Operation on BST

The insertion operation of a binary search tree is performed along with the search operation. The new node is inserted at the point where the search operation terminates. Therefore, the average-case time of insertion operation is $\Theta(\log n)$, while the worst-case time complexity is $O(n)$.

The space complexity of the insertion operation is $O(n)$.

**Program Code 10.8**

```
// Program to create and insert a new node in the binary search tree
#include <iostream.h>
#include <conio.h>
        struct node
        {
                int data;
                node *right;
                node *left;
        };

        class bst
        {
                private:
                        node *start, *cur, *temp, *parent;
                public:
                bst(void)
                {
                        start = NULL;
                }
                void create(int);
                void insert(int);
                void inorder (void);
                void inord(node*);
        };

void main(void)
{
        bst obj;
        int x[] = {22, 33, 21, 76, 88, 43, 34, 11, 89, 55};
        clrscr();
        for(int i = 0; i<=9; i++)
                obj.create(x[i]);
        cout<<"Data of tree in order \n";
        obj.inorder();
        obj.insert (99);
        cout<<"\nData of tree in order after inserting node of value 99:\n";
```

```
        post(start);
} //end of postorder()

        //member function to visit tree recursively
void tree::post(node *s)
{
        if(s!=NULL)
        {
                post(s->left);
                post(s->right);
                cout<<s->data<<"\t";
        }
} //end of post()

        //member function to create and assign data into tree nodes
void tree::create(int x)
{
        if(start == NULL)
        {
                        //create root node and assign data
                start = new node;
                start->data = x;
                start->left = start->right = NULL;
        }
        else
        {
                node *parent;
                cur = start;

                        //search value in tree and go to proper position to insert value
                while(cur!=NULL)
                {
                        if(cur->data == x)
                        {
                                cout<<"Data already exist\n";
                                return;
                        }
                        if(x > cur->data)
                        {
                                parent = cur;
                                cur = cur->right;
                        }
                        else
                        {
                                parent = cur;
                                cur = cur->left;
                        }
```

```
        } // end of while loop

                // create and insert new node
        temp = new node;
        temp->data = x;
        temp->left = temp->right = NULL;
        if(parent == NULL)
                parent = temp;
        else if(parent->data > x)
                        parent->left = temp;
                else
                        parent->right = temp;
    }
} //end of create()
```

### Complexity Analysis of Traversal Algorithms of BST

Each traversal algorithm of BST requires constant work at each node (not including recursive calls). Traversing every node of BST with *n* nodes requires O(n) times. Thus, the worst-case time complexity of each traversal algorithm is O(n). However, the average -case time complexity of each traversal algorithm is Θ(log n) for traversing a BST with *n* nodes having O(log *n*) height.

The space complexity of each traversal algorithm is O(n).

### 10.3.2.3 Insertion Operation on BST

In a binary search tree, a new node is always inserted at the leaf. We traverse the tree from root to leaf. We search the location in the tree (i.e. the leaf node) where the new node is to be inserted. If the value of the new node already exists in any node, then the insertion operation is terminated; otherwise, the new node is inserted at the point the search terminates.

The following steps are taken to insert a new node into the binary search tree:

1- Create a new node with a given value and set its left and right child nodes to NULL.

2- Check whether the tree is empty.

3- If the tree is empty, then set root to a new node.

4- If the tree is not empty, then search the value of the new node in the tree. If a value already exists in the tree, then terminate the insertion operation; otherwise, insert the value at the point the search terminates.

Consider the following binary search tree:

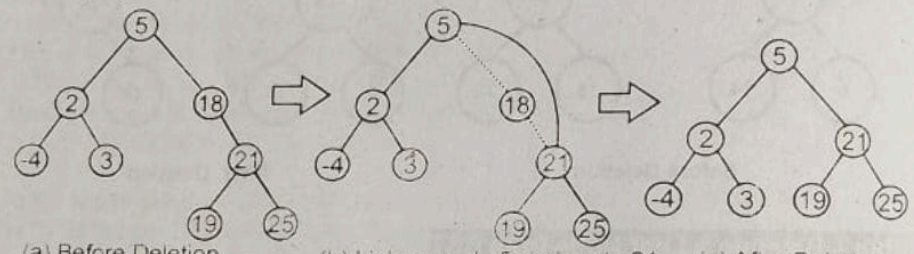
```
        node *left;
    };
    class bst
    {
        private:
            node *start, *cur, *temp, *parent;
        public:
        bst(void)
        {
            start = NULL;
        }
        void create(int);
        void delete_node (void);
        void delete_leaf_node (node*, int);
        void delete_onechild_node (node*, int);
        void delete_twochildren_node (node*, int);
        void preorder (void);
        void preord(node*);
    };
    bst  obj;          // object of class 'bst' is declared as global

void main(void)
{
    int  x[] = {22, 33, 21, 76, 88, 43, 34, 11, 89, 55};
    clrscr();
    for(int i = 0; i<=9; i++)
        obj.create(x[i]);
    cout<<"\nData of tree in preorder "<<endl;
    obj.preorder();
    obj.delete_node();
} //end of main()

    // member function to create tree
void bst::create(int n)
{
    int col = 40, row = 1;
        // to create root node and assign data to it
    if(start == NULL)
    {
        start = new node;
        start->data = n;
        start->left =  start->right = NULL;
        gotoxy(col, row);
        cout<<n;
    }
```

### ii) Deleting a Node with One Child

Deleting a node with one child is also a simple case. This case is similar to delete a node from a linked list. Following steps are performed to delete a node with one child:

- Find the node to be deleted using the search operation.
- If the node to be deleted is found and has only one child which can be a left child or right child.
- If the node to be deleted has a left child, then we link the left child directly to the parent of the deleted node.
- If the node to be deleted has a right child, then we link the right child directly to the parent of the deleted node.

Suppose we want to delete a node 18 from the BST as shown in the following figure (a). This node has only one child (i.e. right child) which is node 21. Node 18 is the right child of its parent node i.e. node 5. We simply link node 5 (right child node of node 18) to the right pointer of node 18 (parent node of node 18). The node 18 will be disconnected from the tree and thus it is automatically deleted. The binary search trees before and after deletion operation are shown in the following figures:




(a) Before Deletion (b) Linking node 5 and node 21 (c) After Deletion

### iii) Deleting a Node with Two Children

This is a difficult case as we need to deal with two subtrees. Following steps are performed to delete a node with two children:

1- Find the node to be deleted using the search operation.

2- If it has two children, then find the node in the left subtree that has the largest value.

OR find the node in the right subtree that has the smallest value.

3- Copy the value of the node found in step-2 to the deleting node. Suppose we find the node in the right subtree that has the smallest value.

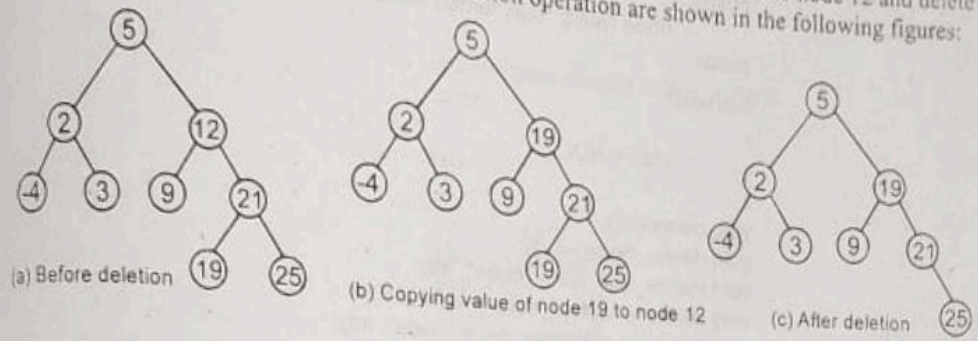
4- Now delete the node whose value is copied to the deleting node in step-3 (the node that has the smallest value in the right subtree of deleting node).

- If it is a leaf node, then use deletion case 1.
- If it has one child, then use deletion case 2.

5- Repeat the same process until the node is deleted from the tree.

```
                                        // checks one child node that has right child but no left child
                                else if (cur->left ==NULL && cur->right !=NULL)
                                        break;                  // if one child node found then loop break

                        if(cur->data < n)
                        {
                                parent = cur;
                                cur = cur->right;
                        }
                        else
                        {
                                parent = cur;
                                cur = cur->left;
                        }
                } //end of while loop
                if(cur == NULL)
                {
                        cout<<"One child node not found";
                        getch();
                        return;
                }
        } //end of else part ()

                // Delete one child node
        if(cur->left!=NULL)
                parent->left = cur->left;
        else
                parent->right = cur->right;
        cout<<"\nData of tree in preorder after deleting the node " <<endl;
        obj.preorder();
        getch();
} //end of delete_onechild_node()

        // function definition to delete two children node from BST.
void bst::delete_twochildren_node (node *ds, int n)
{
        int min;
        if(ds == NULL)
        {
                cout<<"Tree is empty";
                return;
        }
        else
        {
                cur = parent= ds;
                        // search value
                while(cur!=NULL)
```

Suppose we want to delete a node 12 from the BST as shown in the following figure (a). This node has two children. Node 12 is the right child of its parent node i.e. node 5. First of all, we will find the node that has the largest value in the left subtree of node 12 or the node that has the smallest value in the right subtree of node 12. Let us find the node that has the smallest value in the right subtree of node 12. It is node 19. We copy the value of this node to node 12 and delete it. The binary search trees before and after deletion operation are shown in the following figures:




(a) Before deletion (b) Copying value of node 19 to node 12 (c) After deletion

⚠ *If we search the smallest value in the right subtree and the root node of this subtree has a left child, then the smallest value will be in the left branch of the right subtree. We have to search all next coming left child nodes of parent nodes successively. However, if the root node of the right subtree has no left child, then the value of the root of this subtree will be the smallest value.*

*If we search the largest value in the left subtree and the root node of this subtree has a right child, then the largest value will be in the right branch of the left subtree. We have to search all the next coming right child nodes of parent nodes successively. However, if the root node of the left subtree has no right child, then the value of the root of this subtree will be the largest value.*

### Complexity Analysis of Deletion Operation on BST

The deletion operation of a binary search tree is performed along with the search operation The node is deleted from the BST if found. Like insertion operation, the average-case time of deletion operation is $\Theta(\log n)$, while the worst-case time complexity is $O(n)$.

The space complexity of the deletion operation is $O(n)$.

**Program Code 10.9**

```
// Program to create and delete a node from the binary search tree
#include <iostream.h>
#include <conio.h>
        struct node
        {
                int data;
                node *right;
```

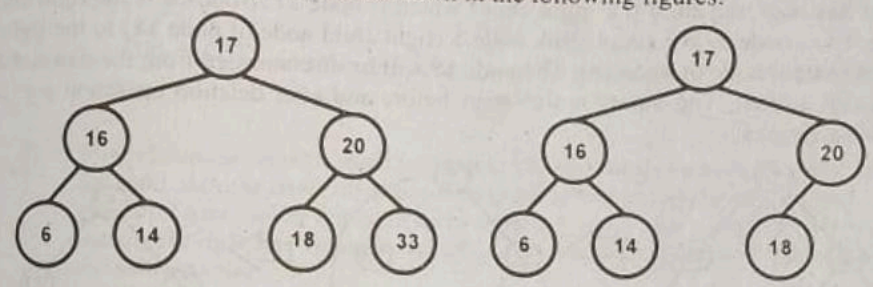
```
        //preord recursive function to visit the tree in a preorder traversal
void bst::preord(node *ss)
{
        if(ss!=NULL)
        {
                cout<<ss->data<<"  ";
                preord(ss->left);
                preord(ss->right);
        }
} //end of preord()

        // function definition for deleting node from BST.
void bst::delete_node()
{

        int val;
        cout<<"\nEnter value of leaf node to delete:";
        cin>>val;
        obj.delete_leaf_node (start, val);
        cout<<"\nEnter value of one child node to delete:";
        cin>>val;
        obj.delete_onechild_node (start, val);
        cout<<"\nEnter value of two child nodes to delete:";
        cin>>val;
        obj.delete_twochildren_node (start, val);
} //end of delete_node ()

        // function definition to delete leaf node from BST.
void bst::delete_leaf_node (node *ds, int n)
{
        if(ds == NULL)
        {
                cout<<"Tree is empty";
                return;
        }
        else            // find the leaf node
        {


                cur = parent= ds;
                        // search value
                while(cur!=NULL)
                {
                        if(cur->data == n)
                                if (cur->left ==NULL && cur->right ==NULL)
                                        break;          // if leaf node found then loop break
                        if(cur->data < n)
```

### Deleting a Leaf Node

Deleting a leaf node is a simple case because this node has no children. Following steps are performed to delete a leaf node:

- Find the leaf node to be deleted using a search operation.
- If the leaf node to be deleted is the left child of its parent node, then assign a NULL value to the left pointer of its parent node; otherwise, assign a NULL value to the right pointer of its parent node.

Suppose we want to delete a leaf node of value 33 from the BST as shown in the following left side figure. Move to this node using a search operation. It is the right child of its parent node (i.e. node 20). Assign a NULL value to the right pointer of its parent node. The leaf node will be disconnected from the tree and thus it is automatically deleted. The binary search trees before and after deletion operation are shown in the following figures:




Before Deletion After Deletion

**Algorithm – Deleting a Leaf Node from BST**

**Write an algorithm for deleting a leaf node from a binary search tree.**

```
1-  START
2-  INUPT value of the node to be deleted, in variable N
3-  Find the node to be deleted that has value N in the Tree
4-  IF node is not found in the tree THEN
            PRINT "Node not found"
            Return
    END IF
5-  IF node is found in the tree THEN
            IF node is the left child of the tree THEN
                    Assign a NULL value to the left pointer of its parent node
            ELSE IF node is the right child of the tree THEN
                    Assign a NULL value to the right pointer of its parent node
            END IF
    END IF
6-  EXIT
```

```
    else
    {
            cur= start;
                    // go to proper position to add node
            while(cur!=NULL)
            {
                    if(cur->data == n)
                    {
                            cout<<"Data already exists"<<endl;
                            return;
                    }
                    if(cur->data < n)
                    {
                            parent = cur;
                            cur = cur->right;
                            row+= 2;
                            col+= 6;
                    }
                    else
                    {
                            parent = cur;
                            cur = cur->left;
                            row+= 2;
                            col-= 6;
                    }
            }
                    // create new node and assign data to it
            temp = new node;
            temp->data = n;
            temp->left = temp->right = NULL;
            gotoxy(col, row);
            cout<<n;
            if(parent==NULL)
              parent = temp;
            else if(parent->data>n)
                    parent->left = temp;
              else
                    parent->right = temp;
    }
} //end of create()

        //member function to call the recursive function preord
void bst::preorder()
{
        preord(start);
} //end of preorder()
```

```
                    parent = cur;
                    cur = cur->left;
                }
            } // end of while loop
                //Insert the value at the point the search terminates
            temp = new node;
            temp->data = n;
            temp->left = temp->right = NULL;
            if(parent == NULL)
                parent = temp;
            else if(parent->data > n)
                parent ->left = temp;
            else
                parent->right = temp;
        }
} //end of insert ()
```

**Output of the program**

Data of tree in order
11 21 22 33 34 43 55 76 88 89
Data of tree in order after inserting node of value 99:
11 21 22 33 34 43 55 76 88 89 99
Data of tree in order after inserting node of value 2:
2 11 21 22 33 34 43 55 76 88 89 99

In the above program, the while loop is used in the insert() function to search the node with value 'n' in the tree. If this loop ends in a normal way, then it means that the node with value 'n' is not found in the tree. The value of pointer 'cur' will be NULL, while the value of the pointer 'parent' will be the address of the leaf node where the search process is terminated. After termination of the loop, a new node is created and attached to the tree.

### 10.3.2.4 Deletion Operation on BST

The deletion operation on binary search trees is more complicated than search and insertion operations. Each node of BST has a parent except the root node. Therefore, deleting any internal node could affect all subtrees of that node. The value of the node that is to be deleted is searched in the binary search tree and if found, it is deleted.

To delete a node from BST, there are three possible cases to consider:

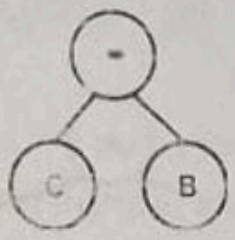
Case 1: Deleting a leaf node (A node with no children)

Case 2: Deleting a node with one child

Case 3: Deleting a node with two children

| | | | | |
|---|---|---|---|---|
| A | B | C | | |

★ Next, the operator "-" is read. Pop the first two operands C and B from the stack. The first tree T1 is created with the operator at its root node and the two operands as its leaves.

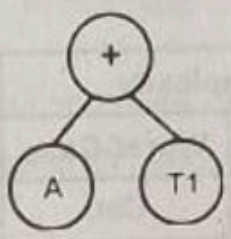



★ The address of the tree is then pushed onto the stack. The stack will be as shown below:

| | | | | |
|---|---|---|---|---|
| A | T1 | | | |

★ When the operator "+" is read, T1 and A are popped from the stack and the tree T2 is created say with "+" operator as root, T1, and A as its leaves. The address of T2 is pushed onto the stack. The stack and the expression tree will look like this:

| | | | | |
|---|---|---|---|---|
| T2 | | | | |

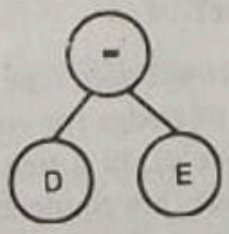



★ Next two operands D and E are pushed onto the stack as shown below:

| | | | | |
|---|---|---|---|---|
| T2 | D | E | | |

★ When the operator "-" is read, both D and E are popped. Tree T3 is created with "-" operator as root and D and E as its leaves. The address of the tree T3 is pushed onto the stack. The stack and the expression tree T3 will be as shown below:

| | | | | |
|---|---|---|---|---|
| T2 | T3 | | | |




★ When the next operator "*" is read, both T2 and T3 are popped and tree T4 is created with "*" operator as its root and its address is pushed onto the stack.

```
            if(parent->left==NULL && parent->right==NULL)      // if node is leaf node
                    obj.delete_leaf_node(temp->right, min);
            else    // if node is one child node
                    obj.delete_onechild_node(temp->right, min);

        }
} //end of delete_twochildren_node()
```

**Output of the program**

```
            22
        21      33
    11              76
            43      88
        34      55      89
```

```
Data of tree in preorder
22  21  11  33  76  43  34  55  88  89
Enter value of leaf node to delete:11

Data of tree in preorder after deleting the node
22  21  33  76  43  34  55  88  89
Enter value of one child node to delete:33

Data of tree in preorder after deleting the node
22  21  76  43  34  55  88  89
Enter value of two child nodes to delete:22

Data of tree in preorder after deleting the node
34  21  76  43  55  88  89
```

## 10.4 EXPRESSION BINARY TREE

A binary expression tree is a specific kind of a binary tree. It is used to represe expressions. It uses arithmetic operators and operands as node values. The internal nod represent the operators of the expression. The leaves of an expression tree represent the operan of the expression which may be either constants or variables. Most operators represent bina operations and thus most internal nodes have two children. However, this is not always the cas Unary operators, such as unary minus, will have only a single child node representing the sing operation of such operator.

Expression trees are useful for representing and evaluating expressions. They are used convert infix expressions into prefix expressions.

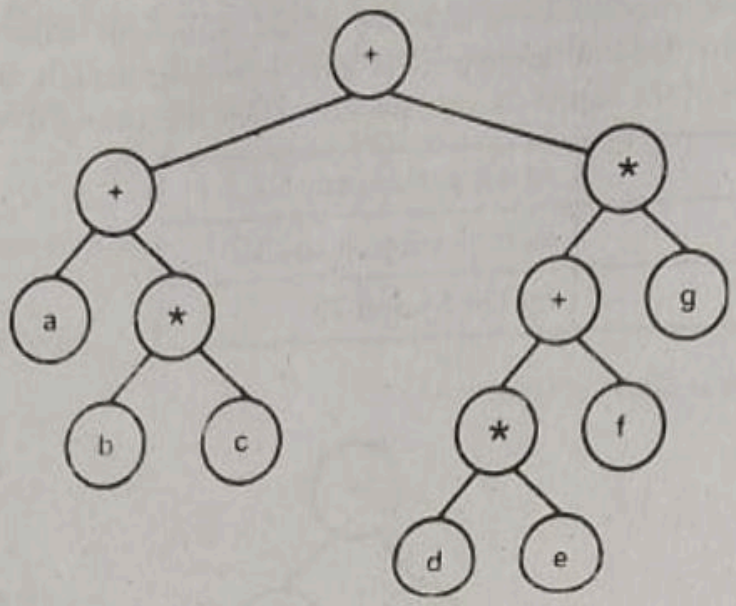
### 10.4.1 Constructing an Expression Binary Tree

Consider the following arithmetic expression:

(a +(b*c))+(((d*e)+f)*g)

```
                {
                        parent = cur;
                        cur = cur->right;
                }
                else
                {
                        parent = cur;
                        cur = cur->left;
                }
            } //end of while loop
            if(cur == NULL)
            {
                cout<<"Leaf node not found";
                getch();
                return;
            }
        } //end of else part ()

            // Delete leaf node
        if(n > parent->data)
            parent->right = NULL;
        else
            parent->left = NULL;
        cout<<"\nData of tree in preorder after deleting the node " <<endl;
        obj.preorder();
        getch();
} //end of delete_leaf()

        // function definition to delete one child node from BST.
void bst::delete_onechild_node (node *ds, int n)
{
        if(ds == NULL)
        {
            cout<<"Tree is empty";
            return;
        }
        else            // find the one child node
        {
            cur = parent= ds;
                // search value
            while(cur!=NULL)
            {
                if(cur->data == n)
                        // checks one child node that has left child but no right child
                    if (cur->left !=NULL && cur->right ==NULL)
                        break;          // if one child node found then loop break
```

```
{
        if(cur->data == n)
                        // check two children node that has left child and right child
                if (cur->left!=NULL && cur->right!=NULL)
                        break;  // if node of two child nodes found then loop break
        if(cur->data < n)
        {
                parent = cur;
                cur = cur->right;
        }
        else
        {
                parent = cur;
                cur = cur->left;
        }
} //end of while loop

if(cur == NULL)
{
        cout<<"Node with two children not found";
        getch();
        return;
}
else    // find the smallest value in right subtree
{
                /* cur contains address of the node to be deleted
                value or address of cur is assigned to temp pointer */
        temp = cur;
                // now address of right child of node to be deleted is assigned to cur
        cur = cur->right;
        min = cur->data;
                // find smallest value in right subtree
        while(cur!=NULL)
        {
                parent = cur;
                if(cur->data < min)
                        min = cur->data ;
                cur = cur->left;
        } //end of while loop
}

// Copy the value smallest node in right subtree into searched node with two child nodes
if(parent->left==NULL)
        temp->data = min;       // copy the smallest into the node to be deleted

        // delete the node that has smallest value in right subtree
```




To convert the expression to prefix Polish notation, the preorder traversal is performed. By using the preorder traversal of the above tree, the expression is obtained in prefix notation as:

+ + a * b c * + * d e f g

To convert the expression to postfix notation, the postorder traversal is performed. By using the postorder traversal of the above tree, the expression is written in postfix Polish notation as:

a b c * + d e * f + g * +

### 10.4.2 Constructing Binary Expression Tree using Stack

A binary expression tree can also be constructed by using a stack. To construct a binary expression tree, follow these steps:

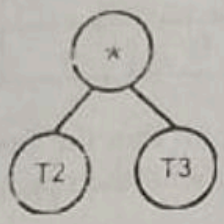
1. Convert the infix expression into postfix form.
2. Read the postfix expression from left to right.
3. If an operand is encountered, push it onto the stack.
4. If an operator is read, pop the first two operands from the stack and create a tree with the read operator as its root node.
5. Store the addresses of the newly created tree onto the stack.
6. Repeat steps 3 and 5 until the end of the expression.

Consider the conversion of the expression [A+(B-C)]*[(D-E)]

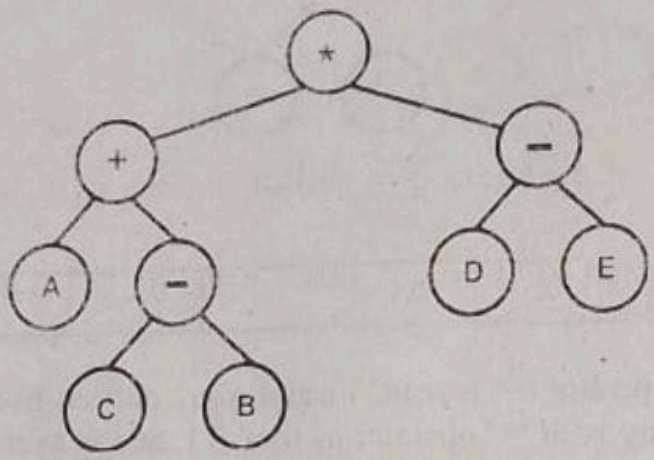
★ The expression is written in postfix expression as:

A B C - + D E - *

★ Read the expression from left to right. Push the first three operands A, B, and C into the stack as they are read (scanned). The stack will be as shown below:




★ All subtrees are combined and the final tree is created as shown in the following figure:




## Summary --- Complexities of Binary Search Tree

**Binary Search Tree (BST)**

| Operation | Time Complexity | | Space Complexity |
|---|---|---|---|
| | Average-Case | Worst-Case | Worst-Case |
| Search | Θ(log n) | O(n) | O(n) |
| Traversal | Θ(log n) | O(n) | O(n) |
| Insertion | Θ(log n) | O(n) | O(n) |
| Deletion | Θ(log n) | O(n) | O(n) |

## Short Answers to the Questions

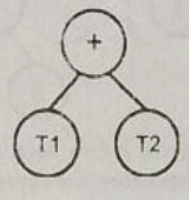
**What is a tree in data structure?**

Tree is the most widely used *abstract data type* (ADT) or data structure. It is a non-linear data structure. It is a collection of nodes (or entities) that are organized in a hierarchical structure (without having any cycle). In a tree data structure, every individual entity or element is called a *node*. The nodes are connected by edges (The connection line between two nodes is called an *edge*). Each node contains a value or data. The first node of the tree is called the *root node* and it may contain several links.

To construct the expression binary tree of the arithmetic expression, the expressions within parenthesis are divided into subtrees. The left part will be the left subtree and the right part will be the right subtree of the expression binary tree. Thus, the given expression is rewritten as:

| | |
|---|---|
| (a+(b*c)) | as left subtree say T1 |
| (((d*e)+f)*g) | as right subtree say T2 |
| + | root of T1 and T2 |

The binary tree is constructed as:




The subtrees T1 and T2 are further divided as follows:

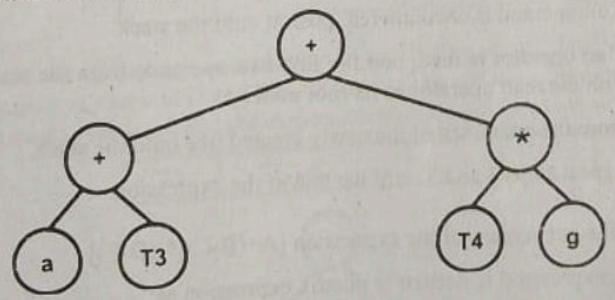
(a+(b*c)) is divided as:

- "a" as the left subtree
- "b*c" as a right subtree T3
- "+" as the root of these subtrees

(((d*e)+f)*g) is divided as:

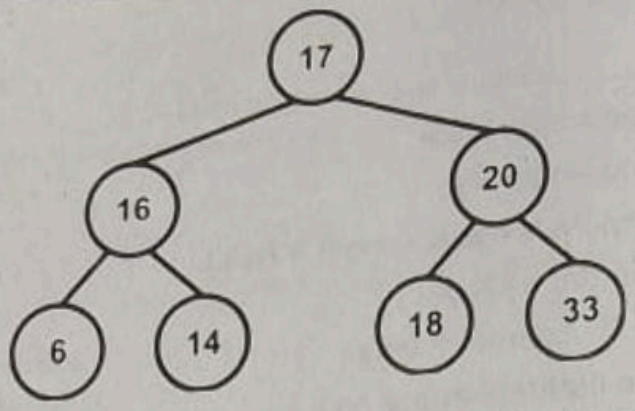
- " ((d*e)+f) " as left subtree T4
- "g" as a right child, and
- "*" as the root of these subtrees.

The above tree becomes as:

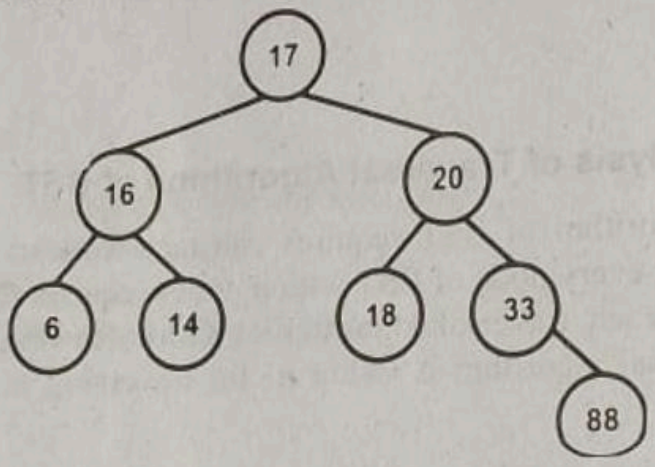



The T3 and T4 are further expanded and the final expression binary tree is as given below:

```
        obj.inorder();
        obj.insert (2);
        cout<<"\nData of tree in order after inserting node of value 2:\n";
        obj.inorder();
        getch();
} //end of main()

        // member function to create tree
void bst::create(int n)
{
                // to create root node and assign data to it
        if(start == NULL)
        {
                start = new node;
                start->data = n;
                start->left =  start->right = NULL;
        }
        else
        {
                cur = start;
                        // go to proper position to add node
                while(cur!=NULL)
                {
                        if(cur->data == n)
                        {
                                cout<<"Data already exists"<<endl;
                                return;
                        }
                        if(cur->data < n)
                        {
                                parent = cur;
                                cur = cur->right;
                        }
                        else
                        {
                                parent = cur;
                                cur = cur->left;
                        }
                }
                        // create new node and assign data to it
                temp = new node;
                temp->data = n;
                temp->left = temp->right = NULL;
                if(parent==NULL)
                        parent = temp;
                else if(parent->data>n)
```




Suppose we need to insert a new node with value 88 into this binary search tree. To insert this node, firstly this node is searched (with the reference to its data value) in the tree. If the node is not found, then it is inserted at the point the search process terminates. The above BST, after insertion of a new node with data value 88, is shown in the following figure:




## Algorithm – Insertion Into BST

**Write an algorithm that inserts a new node in a binary search tree.**

1. START
2. INPUT value for the new node to insert, in variable N
3. IF the BST is empty THEN
   - Create a new node and set this node as root
   - Return i.e. stop the insertion process

   ELSE

   Search value N in the Tree

   IF the node with value N already exists in the tree THEN
   - PRINT "Node already exists"
   - Return

   ELSE

   Insert the node at the point the search terminates

   END IF

   END IF
4. EXIT