# MODULE -3

Non Linear Data Structures – Trees – Binary Tree - Definition - Basic Terminologies - Node, Parent, Child, Link, Root, Leaf, Level, Height of a tree and node, Depth of a tree and node, Degree of a tree and node, sibling, Ancestors, Path, Path Length - Types of Binary Trees: Full, Complete, Strict, Perfect, - Representations of a Binary Tree Linked Lists - Operations on a Binary Search Tree: Insertion –Traversal – Deletion – Searching – Sorting application – Count number of nodes – Height - Expression Tree – Threaded Binary Tree.

## TREES

Tree is a non-linear data structure.

Definition:A tree is recursively defined as a set of one or more nodes where one node is designated as the root of the tree and all remaining nodes can be partitioned into non empty set each of which is a subtree of the root.

Example:

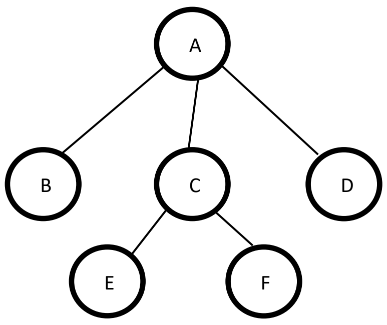


Figure 3.1 An example tree

Subtree: is a portion of a tree that can be viewed as a tree itself.

Basic terminologies of a tree

1. **Node**. This is the main component of any tree structure. The concept of the node is the same as that used in a linked list. A node of a tree stores the actual data and links to the other connecting node.

2. **Parent**. The parent of a node is the immediate predecessor of a node. In the above tree, A is the parent of B, C and D. A node has atmost one parent.

3. **Child**. All immediate successors of a node are known as child. In the above tree,  B, C and D are the two child nodes of A. Each node in a tree has zero or more child nodes

4. **Link**. This is a pointer to a node in a tree. There may be zero or more links for a node.

5. **Root**. This is a specially designated node which has no parent. In the above tree, A is the root node

6. **Leaf**. The node which does not have any child is called leaf node. In the above tree, B, D, E and F are the leaf nodes. A leaf node is also called a terminal node.
7. **Level**. Level is the rank in the hierarchy. Every node is assigned a level number in such a way that the root is at level zero, children of the root node are at level 1 and so on. Thus every node is at one level higher than its parent.
8. **Height**. The number of nodes that is possible in a path starting from the root node to a node is called the height of that node. In the above tree, height of node B is 2. It can be easily seen that height = level + 1.
   The maximum number of nodes that is possible in a path starting from the root node to a leaf node is called the height of a tree. Height of the above tree is 3.
9. **Degree**. It is the number of children that a node has. For example, the degree of node A of the above tree is 3. Degree of a leaf node is zero. The maximum degree represents the degree of the tree.
10. **Siblings**. The nodes which have the same parent are called siblings. For example, in the above tree, E and F are siblings.
11. **Ancestors**. A node that is connected to all lower-level nodes is called an "ancestor". The connected lower-level nodes are "descendants" of the ancestor node.
12. **Path**. Path refers to the sequence of nodes along the edges of a tree. There is only one path from the root node to any node.
13. Path Length. The **length** of a path is the number of nodes in the path.

**Types of Trees**

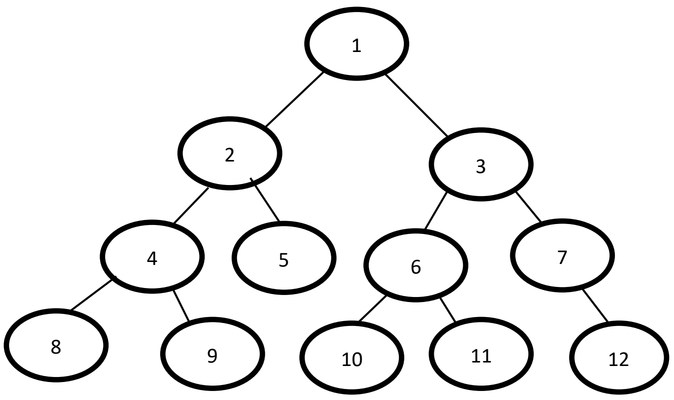
Different types of trees are:

1. Binary Trees
2. Binary Search Trees
3. Expression Trees
4. Threaded Binary Trees.

## BINARY TREES

A binary tree is a non-linear data structure in which the topmost element is called the root node and each node has 0, 1 or 2 children. A node that has zero children is called a leaf node or a terminal node. A binary tree is recursive by definition as every node in the tree contains a left binary subtree and a right binary subtree

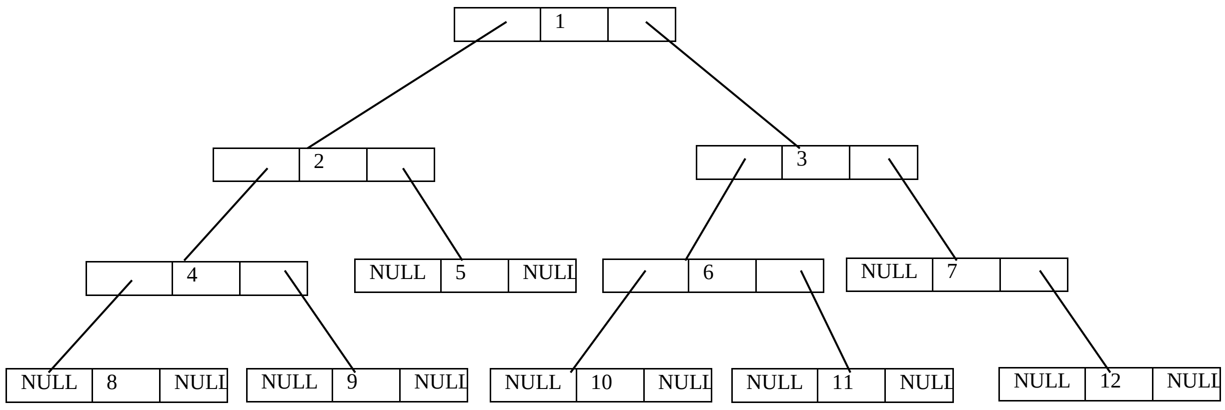
Example:



**Linked representation of Binary Trees**

Every node contains three fields – data field, left pointer field which stores the address of left child and a right pointer field which stores the address of right child.

The address of root node is stored in a special pointer variable called root. If root==NULL, it indicates that the tree is empty. Linked representation of the above binary tree can be drawn as:



A node can be represented as :

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

The data field is used to store the data elements. It can be of any type (int, float, char….). The left pointer is used to store the address of left child and the right pointer field is used to store the address of right child. If a node has no left child, its left pointer field points to NULL. Similarly if a node has no right child, its right pointer field points to NULL.

## Traversing a Binary Tree

Traversing a binary tree is the process of visiting each node in the tree exactly once in a systematic way. Since binary tree is a non-linear data structure, its elements can be traversed in many different ways. There are mainly three methods to traverse a binary tree. These methods differ in the order in which the nodes are visited. The methods are :

1. Pre-order traversal.
2. In-order traversal
3. Post-order traversal

### Pre-order Traversal

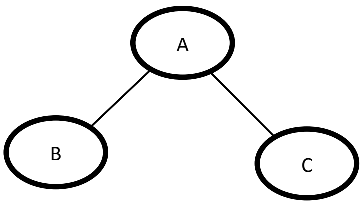
To traverse a non-empty binary tree in pre-order, the following operations are performed recursively at each node.

1. Visit the root node

2. Traverse the left sub-tree
3. Traverse the right sub-tree

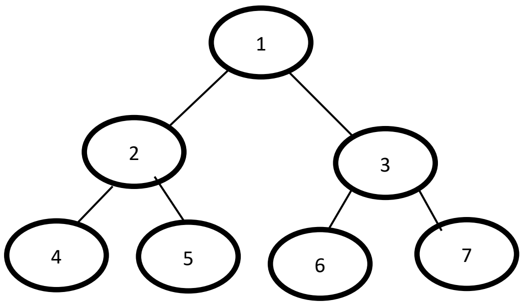
This algorithm is also called NLR algorithm (Node, Left, Right)

Example 1:



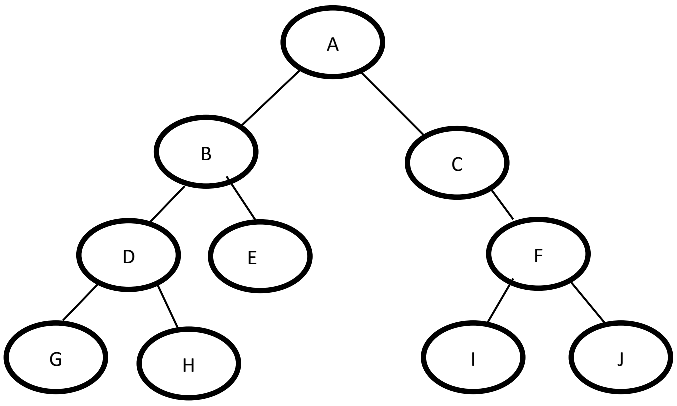
Pre-order Traversal : A B C

Example 2:



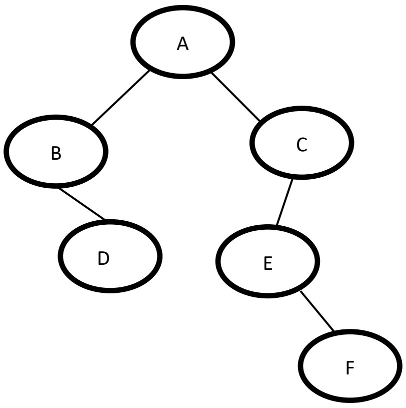
Pre-order Traversal : 1 2 4 5 3 6 7

Example 3:



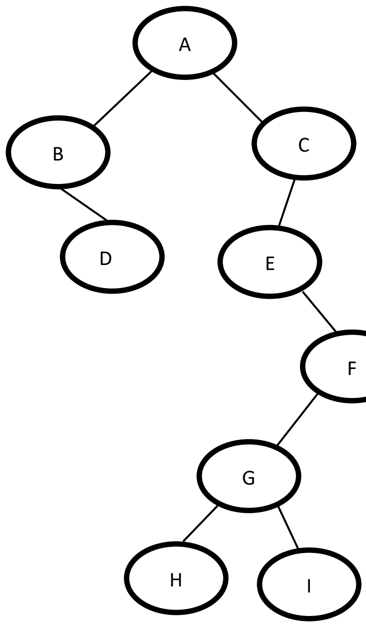
Pre-order Traversal : A B D G H E C F I J

Example 4:



Pre-order Traversal : A B D C E F

Example 5:



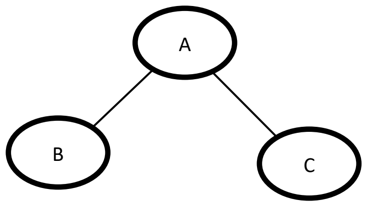
Pre-order Traversal : A B D C E F G H I

## In-order Traversal

To traverse a non-empty binary tree in in-order, the following operations are performed recursively at each node.

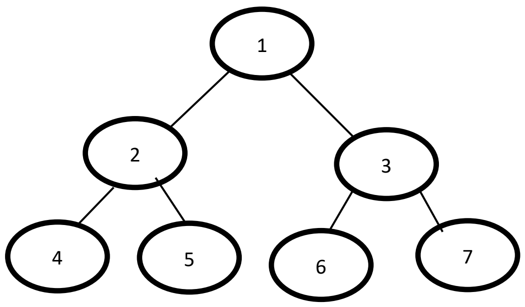
1. Traverse the left sub-tree
2. Visit the root node
3. Traverse the right sub-tree

This algorithm is also called LNR algorithm (Left, Node, Right)
Example 1:



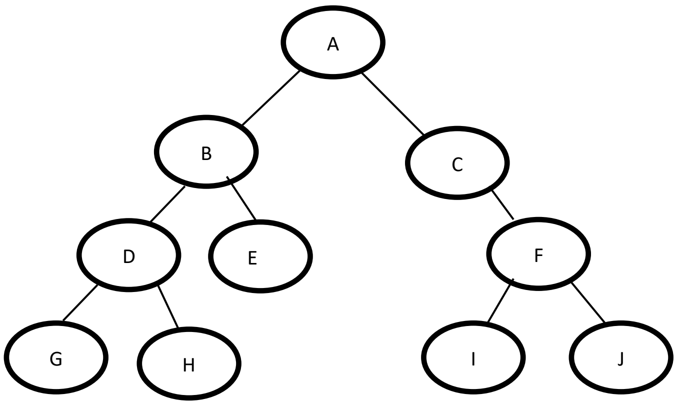
In-order Traversal : B A C

Example 2:



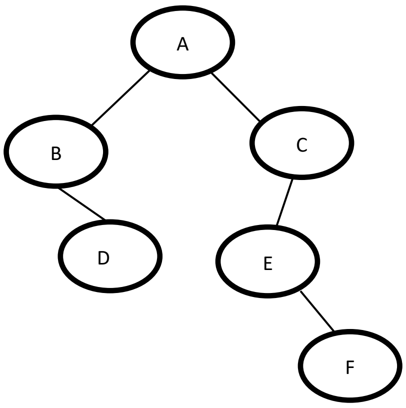
In-order Traversal : 4 2 5 1 6 3 7

Example 3:



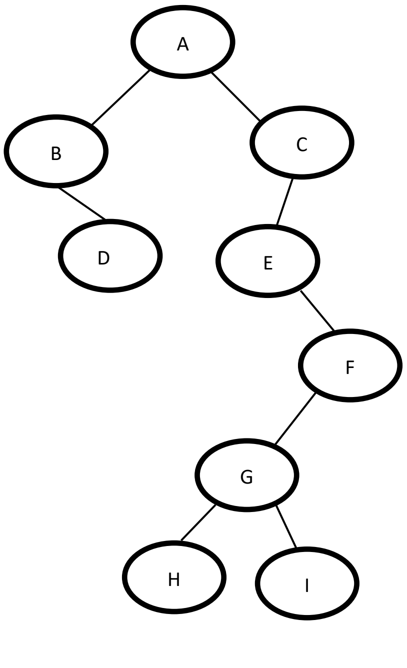
In-order Traversal : G D H B E A C I F J

Example 4:



In-order Traversal : B D A E F C

Example 5:



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

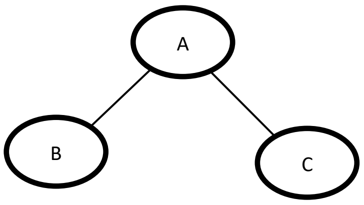
## Post-order Traversal

To traverse a non-empty binary tree in post-order, the following operations are performed recursively at each node.

1. Traverse the left sub-tree
2. Traverse the right sub-tree
3. Visit the root node

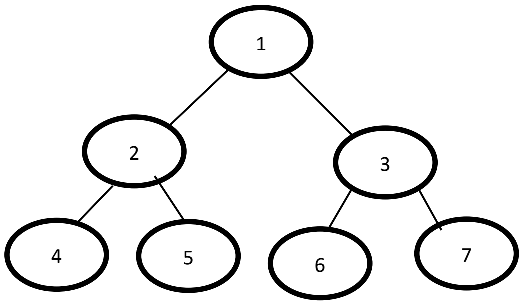
This algorithm is also called LRN algorithm (Left, Right, Node)

Example 1:



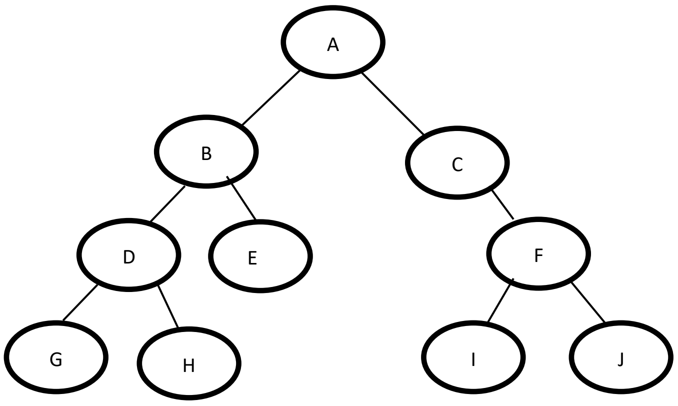
Post-order Traversal : B C A

Example 2:



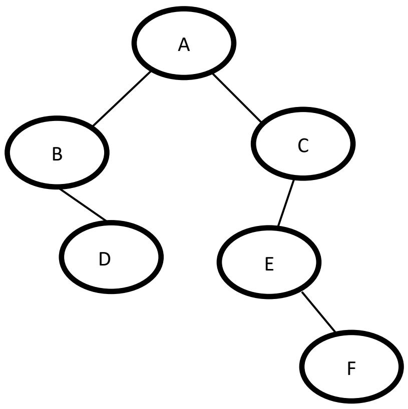
Post-order Traversal : 4 5 2 6 7 3 1

Example 3:



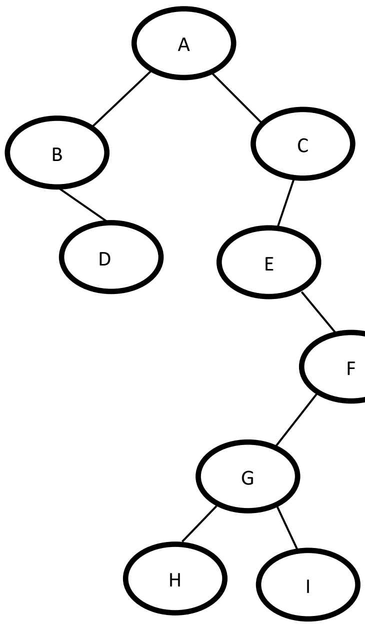
Post-order Traversal : G H D E B I J F C A

Example 4:



Post-order Traversal : D B F E C A

Example 5:



Post-order Traversal : D B H I G F E C A

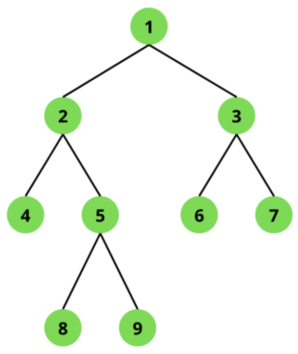
**<u>Types of Binary Trees</u>**

Different types of binary trees are:

1. Strictly Binary Trees/ Full Binary Trees
2. Complete Binary Trees
3. Perfect Binary Trees.

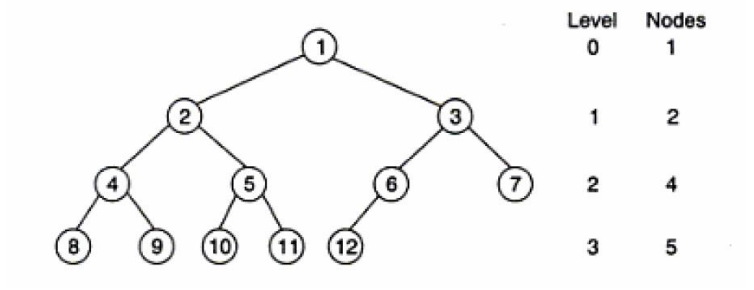
**Strictly Binary Trees/ Full binary tree**

A binary tree in which every node has either two or zero number of children is called strictly binary tree. Strictly binary tree is also called as **Full Binary Tree** or **Proper Binary Tree** or **2-Tree.**



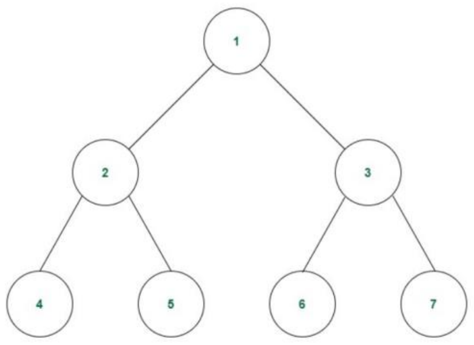
**Complete binary tree**

A binary tree is said to be a complete binary tree if all its levels, except possibly the last level, have the maximum number of possible nodes, and all the nodes in the last level appear as far left as possible. The following figure depicts a complete binary tree.



**Perfect Binary Trees**

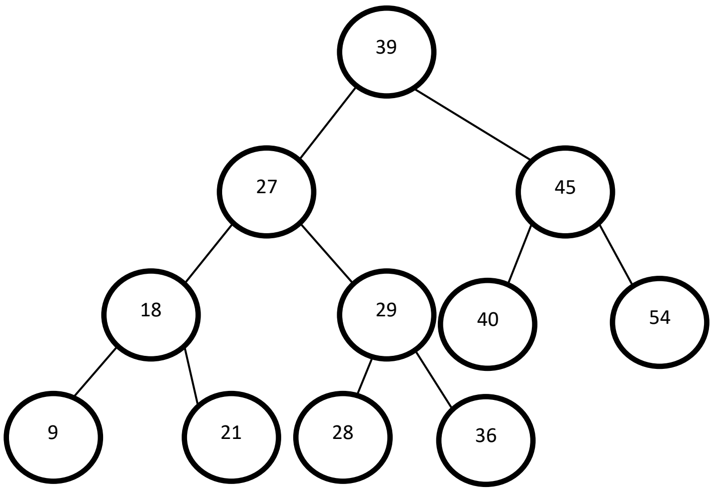
A **perfect binary tree** is a special type of binary tree in which all the leaf nodes are at the same depth, and all non-leaf nodes have two children.



## BINARY SEARCH TREES (BST)

A Binary Search Tree also known as Ordered Binary Tree is a variant of binary tree in which the nodes are arranged in an order. In a BST, for each node

a) The left sub-tree contains only nodes with values less than the parent node.
b) The right sub-tree contains only nodes with values greater than or equal to the parent node.



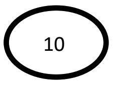
In-order traversal of a BST will list the elements in ascending order. For example, In-order traversal of the above BST is : 9 18 21 27 28 29 36 39 40 45 54

Operations that can be performed on BSTs are:

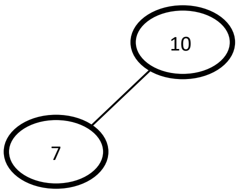
1. Creation
2. Traversals
3. Insertion of a new node
4. Deletion of a node
5. Searching for a node

**1. <u>Creation of a BST</u>**

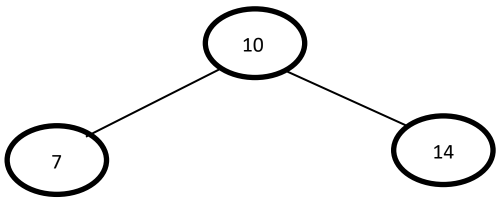
Consider the following array of elements:
10, 7, 14, 20, 1, 5 and 8
Step 1: Make 10 as the root node.



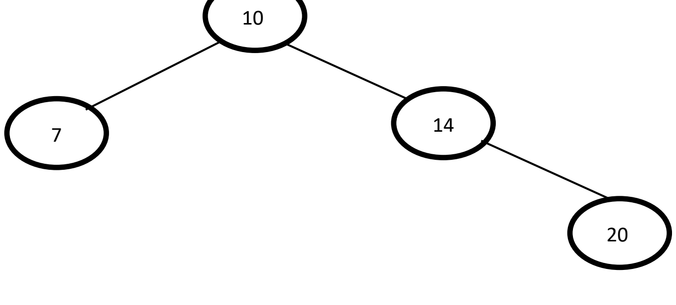
Step 2: The next value to be considered is 7. Compare it with the root node. Since it is less than 10, make it as left child of 10.



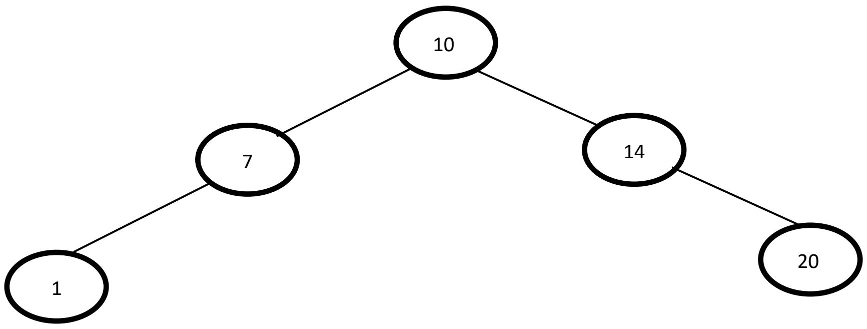
Step 3: The next value to be considered is 14. Compare it with the root node. Since it is greater than 10, make it as right child of 10.



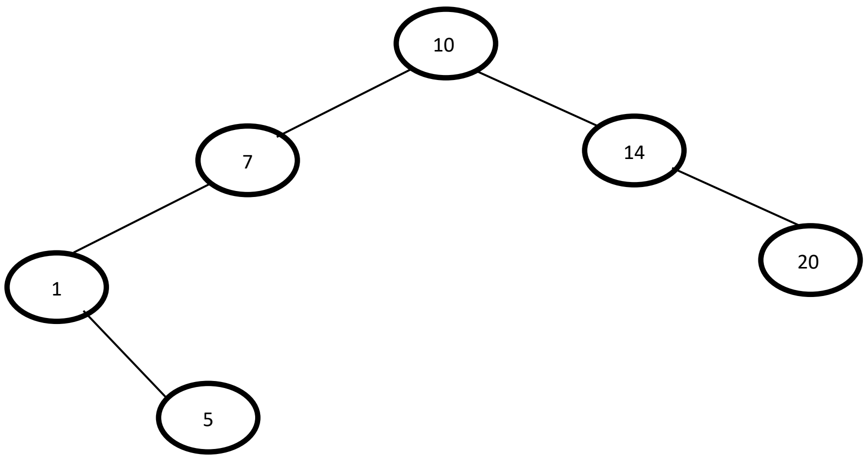
Step 4: The next value to be considered is 20. Compare it with the root node. Since it is greater than 10, we move to the right, compare it with 14 and since it is greater than 14 make it as right child of 14.



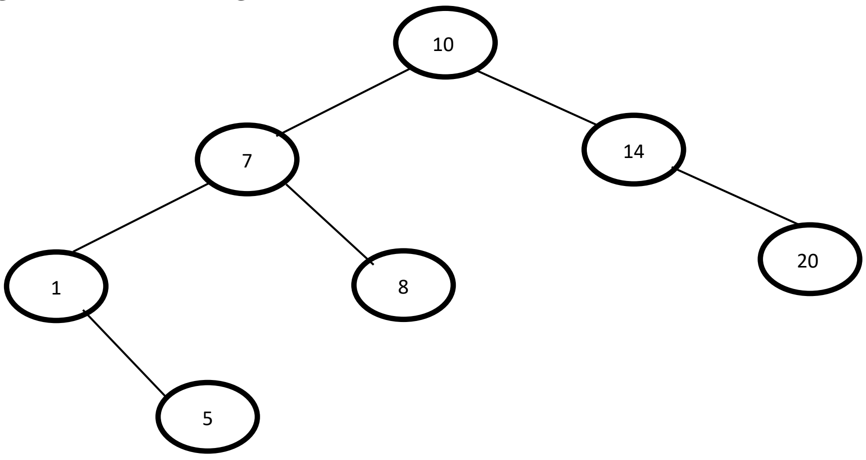
Step 5: Now we have 1. Compare it with the root node. Since it is less than 10, we move to the left, compare it with 7 and since it is less than 7 make it as left child of 7.



Step 6: Now we have 5. Compare it with the root node. Since it is less than 10, we move to the left, compare it with 7, since it is less than 7, again move to the left, compare it with 1 and since it is greater than 1, move to the right and make it as right child of 1



Step 7: Now the last element to be considered is 8. Compare it with the root node. Since it is less than 10, we move to the left, compare it with 7, since it is greater than 7, move to the right and make it as right child of 7.



In-order traversal of the above tree is : 1, 5, 7, 8, 10, 14, 20

Exercise: Create a BST with following elements: I A C K L Q B D

**Program**

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
        int data;
        struct node* left;
        struct node* right;
};
struct node* root=NULL;
int a[50],n,count=0;
void read();
void create();
void insertion();
struct node *makenode(int);
struct node* insert(struct node*, struct node *);
void inorder(struct node *);
void preorder(struct node *);
void postorder(struct node *);
void searching();
void deletion();
void deletenode(struct node *, struct node *);
void traversal();
void counting(struct node*);
int main()
{
        int op;
        printf("Enter the total no.of nodes:");
        scanf("%d",&n);
        printf("\nEnter the datas:");

        read();
        create();
        printf("MENU\n1. Insertion\n2. Deletion\n3. Searching\n4. Traversal\n5. Count\n6. Exit");
        do
        {
                printf("\nEnter the option: ");
                scanf("%d",&op);
                switch(op)
                {
                case 1:
                        insertion();
                        break;
                case 2:
                        deletion();
                        break;
                case 3:
                        searching();
                        break;
                case 4:
                        traversal();
                        break;
                case 5:
                        count=0;
                        counting(root);
                        printf("Number of nodes: %d",count);
                        break;
                case 6:
                        exit(0);
                default:
                        printf("Invalid option");
                }
        }while(1);

}

void read()
{
        int i;
        for(i=0;i<n;i++)
                scanf(“%d”,&a[i]);
}
void create()
{
        struct node* head;
        int i;
        for(i=0;i<n;i++)
        {
                 head=makenode(a[i]);
                root=insert(head,root);
        }

}
struct node* makenode(int item)
{
        struct node *head=(struct node *)malloc(sizeof(struct node));
         head->data=item;
         head->left=NULL;
         head->right=NULL;
        return head;
}
```

```
struct node* insert(struct node *head,struct node *root)
{
        if(root==NULL)
                root=head;
         else
        {
                 if(head->data<root->data)
                        root->left=insert(head,root->left);
                else
                        root->right=insert(head,root->right);
        }
        return root;
}
```

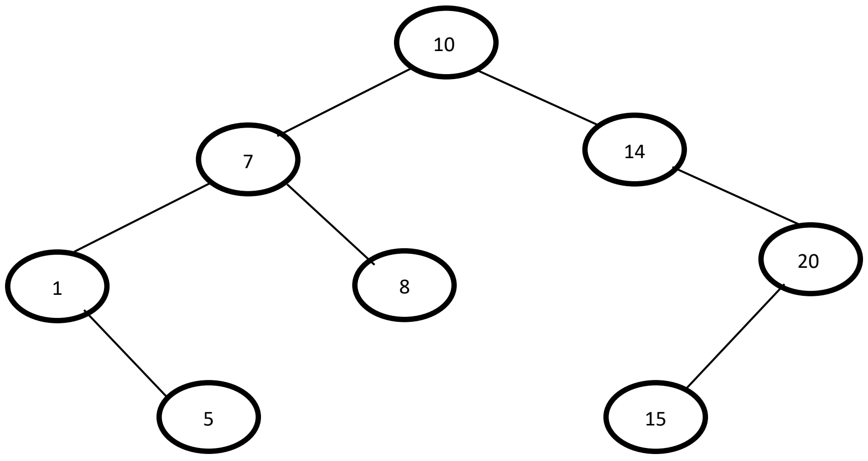
2. **Insertion of a new node**

For inserting a new node into a BST, we first find the correct position at which the insertion has to be done and then add the node at that position. Finding the correct position means that the insertion of a new node must not violate the properties of BST.

Algorithm

```
Step1: If root=NULL, allocate memory for root
        Set root->data = value
        Set root->left = NULL;
        Set root->right = NULL
        else if value<root->data, insert value to the left sub-tree, otherwise insert value to the right sub-tree.
Step2 : Stop.
```

Example: Suppose we want to insert 15 into the above tree. First we compare it with the root node. Since it is greater than 10, move to the right, compare it with 14, since it is greater than 14, again move to the right, compare it with 20, since it is less than 20, move to the left and make it as left child of 20. The final BST will be as:



```
void insertion()
{

    int item;
    struct node* head;
    printf("Enter the data:");
    scanf("%d",&item);
    head=makenode(item);
    root=insert(head,root);
}
```

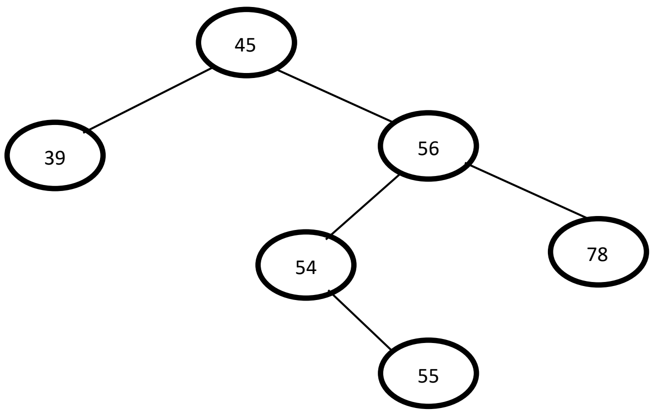
3. **Searching for a node in BST**

The searching process begins at root node. First check whether the BST is empty. If so, we declare that the value we are searching for is not present in the tree and the search is unsuccessful. Otherwise, compare the value with root node. If it is less than root, search the value in the left-sub-tree, otherwise search in the right sub-tree. The searching process continues until either matching occurs or a NULL pointer is encountered.

Example: Suppose we have to search for 8 in the above tree. First compare it with the root 10. Since it is less than 10, move to the left sub-tree and compare it with its root 7. Since it is greater than 7, move to the right sub-tree and compare it with its root 8. Since matching occurs at this stage, the algorithm terminates with a successful search. Otherwise the algorithm will continue until a NULL pointer is encountered.

```
void searching()
{
        int item;
        struct node *ptr=root;
        printf("Enter the item to be searched:");
        scanf("%d",&item);
        while(ptr!=NULL)
        {
                if(ptr->data == item)
                {
                        printf("Item is present in the tree.");
                        break;
                }
                else if(item<ptr->data)
                        ptr=ptr->left;
                else
                        ptr=ptr->right;

        }
        if(ptr==NULL)
                printf("Item doesn't exist.");
}
```

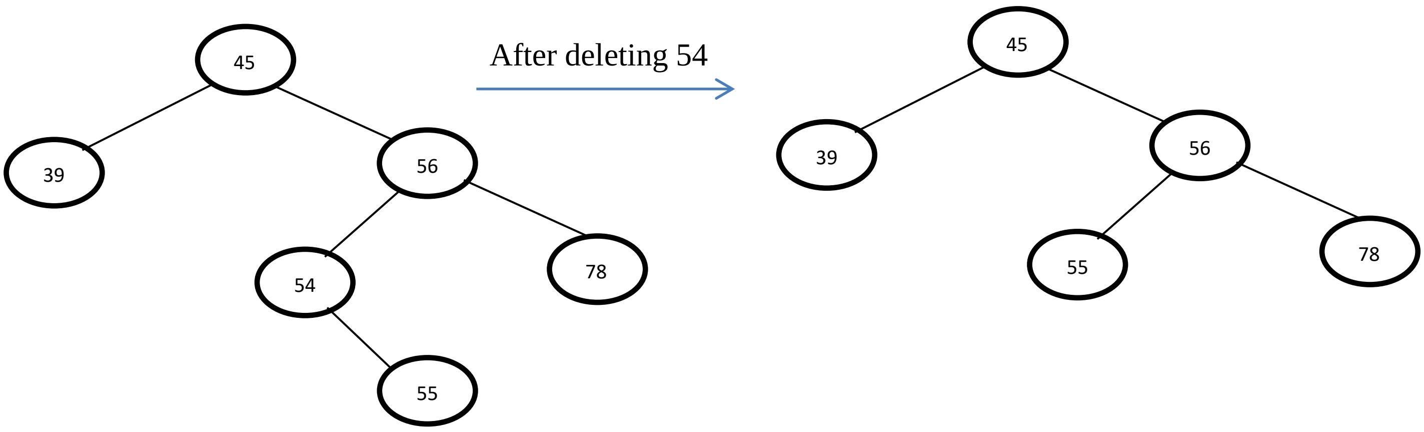
4. **Deletion of a new node in BST**
   After performing deletion, properties of BST must not be violated. There are three cases to consider for deletion.
   Case 1: Deleting a node that has no children.



Suppose we have to delete node 78 which has no children. Here we make right child of 56 (parent of 78) to point to NULL.

If the node to be deleted is right child of its parent, make the right child of its parent to point to NULL. If the node to be deleted is the left child of its parent make the left child of its parent to point to NULL.

Case 2: Deleting a node with one child.



To handle this case, the node's child is set as the child of the node's parent. ie, replace the node with its child. If the node to be deleted is left child of its parent, its child becomes the left child of its parent. Correspondingly, if the node to be deleted is right child of its parent, its child becomes the right child of its parent.

Case 3: Deleting a node with two children.

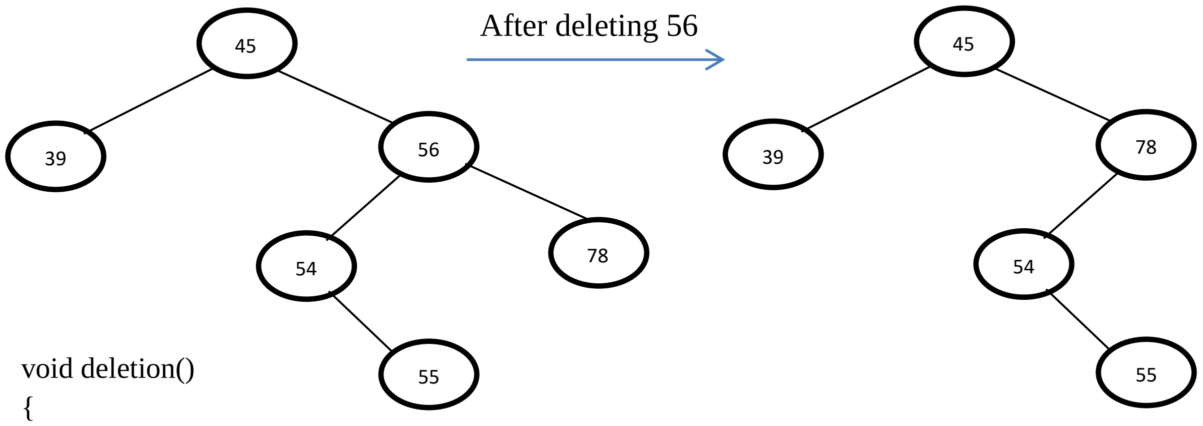
Inorder successor and Inorder predecessor of a node : In in-order traversal of a tree, the node coming after a node is its inorder successor and the node coming before a node is its inorder predecessor. For example, in-order traversal of the above tree is

39 45 54 55 56 78

78 is the inorder successor of 56 and 55 is the inorder predecessor of 56. The first node in the in-order traversal has no in-order predecessor and the last node has no in-order successor.

To handle case 3, replace value of the node which is to be deleted with the value of its in-order successor or predecessor node and then delete the in-order successor or in-order predecessor node using either case1 or case 2.

Example:



```
void deletion()
{
        int item;
        struct node *ptr=root;
        struct node* parent=NULL;
        printf("\nEnter the data to be deleted:”);
        scanf(“%d”,&item);

        while(ptr!=NULL)
        {
                if(item<ptr->data)
                {
                        parent=ptr;
                        ptr=ptr->left;
                }
                else if(item>ptr->data)
                {
                        parent=ptr;
                        ptr=ptr->right;
                }
                else
                deletenode(ptr,parent);
        }
        if(ptr==NULL)
                printf("\nItem doesn't exist in the tree”);
}

void deletenode(struct node *ptr,struct node *parent)
{
```

```
struct node *ptr1;
if(ptr->left==NULL&&ptr->right==NULL)
{
         if(ptr==root)
                 root=NULL;
        else
        {
                if(parent->left==ptr)
                        parent->left=NULL;
                else
                        parent->right=NULL;
        }
        free(ptr);
}
else if(ptr->left!=NULL&&ptr->right!=NULL)
{
        ptr1=ptr->right;
        parent=ptr;
        while(ptr1->left!=NULL)
        {
                parent=ptr1;
                ptr1=ptr1->left;
        }
        ptr->data=ptr1->data;
        deletenode(ptr1,parent);
}
else
{
        if(ptr==root)
        {
                if(ptr->left==NULL)
                        root=ptr->right;
                else
                        root=ptr->left;
        }
        else
        {
                if(parent->left==ptr)
                {
                        if(ptr->left==NULL)
                        parent->left=ptr->right;
                else
                        parent->left=ptr->left;
        }
        else if(parent->right==ptr)
        {
        if(ptr->left==NULL)
                parent->right=ptr->right;
                else
                        parent->right=ptr->left;
                }
        }
        free(ptr);
        }
}
```

**TRVERSALS**

```
void traversal()
{
        printf(inorder treaversal is: “);
        inorder(root);
        printf(“\npreorder treaversal is: “);
        preorder(root);
        printf(“\npostorder treaversal is: “);
        postorder(root);

}

void inorder(struct node* root)
{
        if(root!=NULL)
        {
                inorder(root->left);
                printf("%d  ",root->data);
                inorder(root->right);
        }
}
void preorder(struct node* root)
{
        if(root!=NULL)
        {
                printf("%d  ",root->data);
                preorder(root->left);
                preorder(root->right);
        }
}
void postorder(struct node* root)
{
        if(root!=NULL)
        {
                postorder(root->left);
                postorder(root->right);
                printf("%d  ",root->data);
        }
}
```

**Counting Number of Nodes in a BST**

We can find number of nodes in a tree by traversing the tree in any order.

```
void counting(struct node* root)
{
        if(root!=NULL)
        {
                counting(root->left);
                count++;
                counting(root->right);
        }
}
```

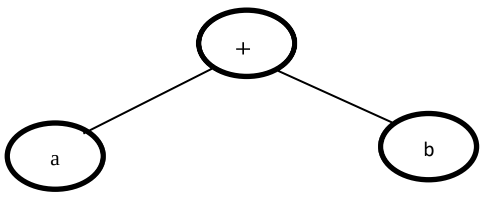
**Height of a BST**

The height of a tree is the length of the path from the root to the deepest node in the tree.

```
int findHeight(struct node * root)
{
   int lefth;
   int righth
   if (root == NULL)
      return -1;

   lefth = findHeight(root->left);
   righth = findHeight(root->right);

   if (lefth > righth)
      return lefth + 1;
   else
      return righth + 1;

}
```

## EXPRESSION TREE

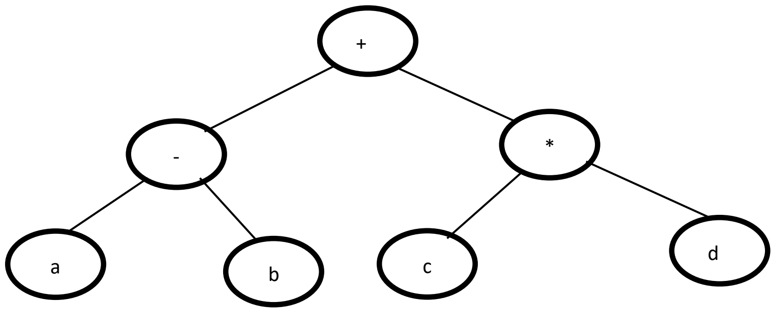
Binary trees can be used to store algebraic expressions. Such type of trees are called expression trees. For example, the expression a+b can be represented as:



In an expression tree, all operands are in leaf nodes and operators are in internal nodes (nodes other than leaf nodes). If we traverse the expression tree in in-order form, we will get the expression in infix form, if we traverse the expression tree in post-order form, we will get the expression in postfix form and if we traverse the expression tree in pre-order form, we will get the expression in prefix form.

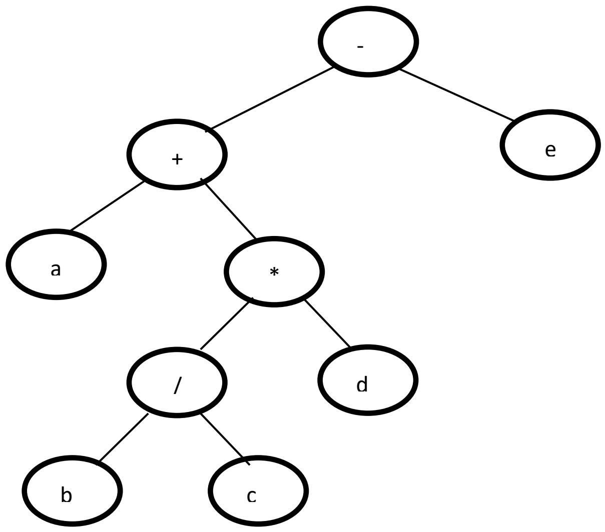
Example:

1. (a-b) + (c*d)



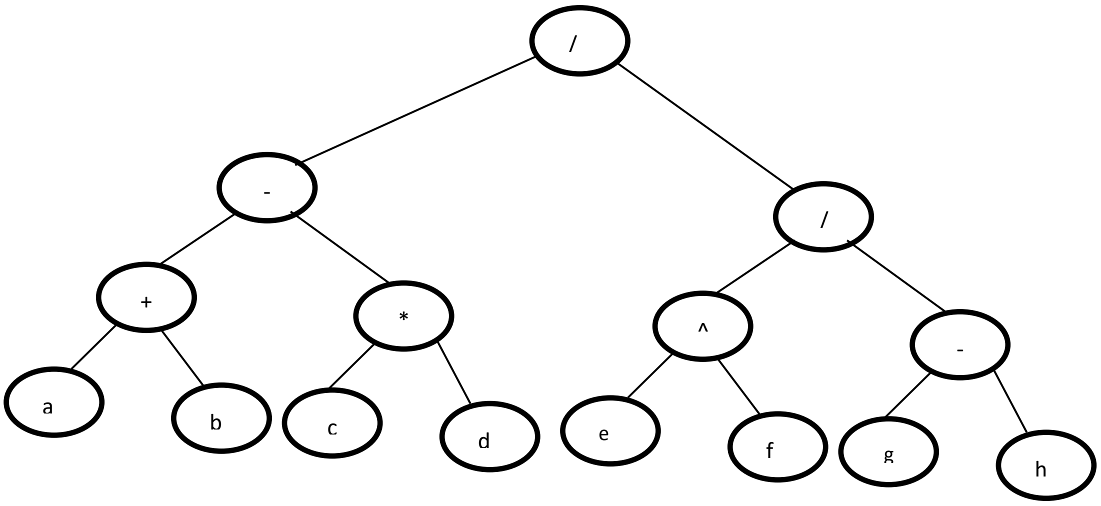
In-order traversal : a-b+c*d
Pre-order traversal: +-ab*cd
Post-order traversal: ab-cd*+

2. a+b/c*d-e



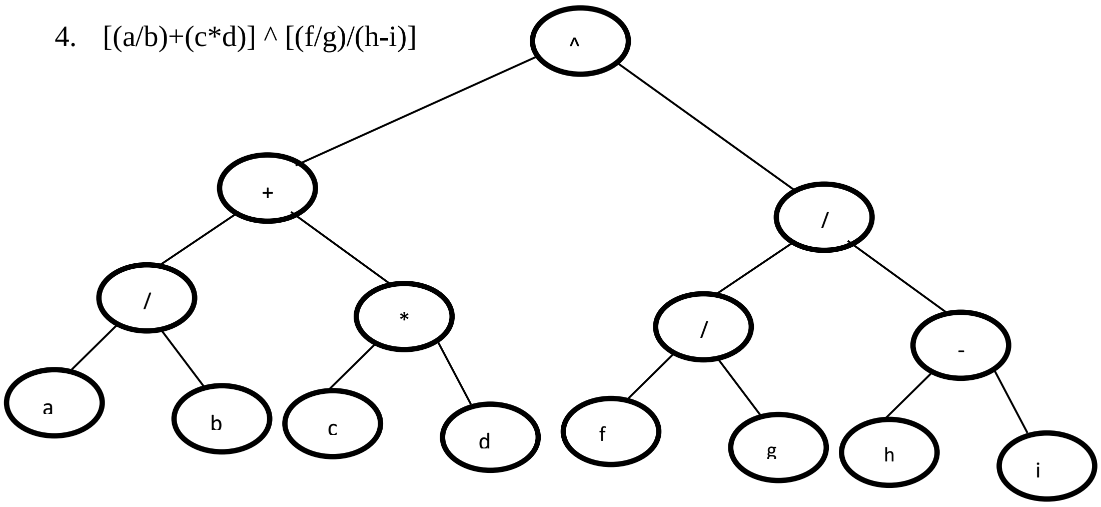
In-order traversal : a+b/c*d-e
Pre-order traversal: -+a*/bcde
Post-order traversal: a+bc/d*e-

3. [(a+b)-(c*d)]/[(e^f)/(g-h)]



In-order traversal : a+b-c*d/e^f/g-h
Pre-order traversal: /-+ab*cd/^ef-gh
Post-order traversal: ab+cd*-ef^gh-//

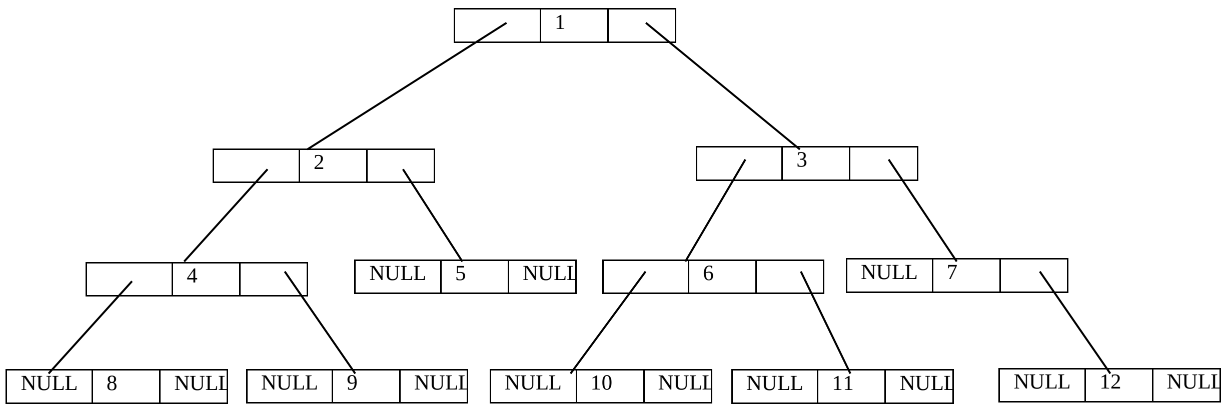
4. [(a/b)+(c*d)] ^ [(f/g)/(h-i)]



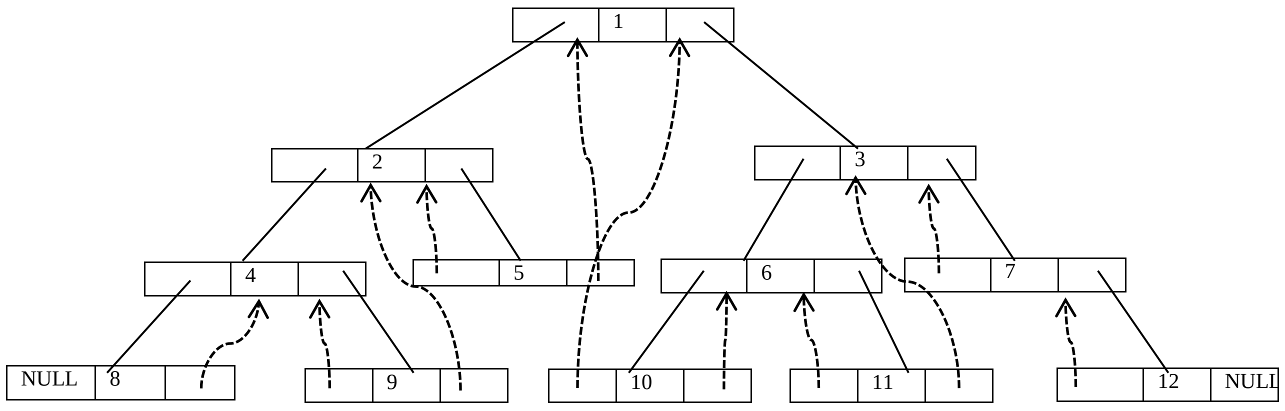
In-order traversal : a/b+c*d^f/g/h-i
Pre-order traversal: ^+/ab*cd//fg-hi
Post-order traversal: ab/cd*+fg/hi-/^

## THREADED BINARY TREE

Threaded Binary Tree (TBT) is same as that of a binary tree but with a difference in storing NULL pointers. For example, in the following tree, there are 13 NULL pointers. The space of storing NULL pointers can be efficiently used to store some other useful information.



In a TBT if left child of a node is NULL, it will point to its in-order predecessor and if right child of a node is NULL, it will point to its in-order successor. These special pointers are called threads and binary trees containing threads are called **threaded binary trees**. Usually threads are represented using dotted lines or lines with arrows and normal pointers are represented using solid lines in pictorial representation.



1
2
3
4
5
6
7
NULL 8
9
10
11
12 NULL