AISSMS
INSTITUTE OF INFORMATION TECHNOLOGY
ADDING VALUE TO ENGINEERING

**Department Of Computer Engineering**

# Data Structure And Algorithms Lab

# Group-B

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AISSMS IOIT

**SE COMPTER ENGINEERING**

**SUBMITTED BY**

**Kaustubh S Kabra**

**ERP No.- 34**

**Teams No.-20**

**2020 -2021**

# Experiment Number:-3

- **Aim:-** *A book consists of chapters, chapters consist of sections and sections consist of sub sections. Construct a tree and print the nodes. Find the time and space complexity of the program.*

- **Objective:-**
    1) To learn the basics of tree data structure in C++ and apply it .

- **Theory:-**

**Trees:**

Unlike Arrays, Linked Lists, Stack and queues, which are linear data structures, trees are hierarchical data structures.

Trees are non-linear hierarchical data structures. A tree is a collection of nodes connected to each other by means of "edges" which are either directed or undirected. One of the nodes is designated as "Root node" and the remaining nodes are called child nodes or the leaf nodes of the root node. In general, each node can have as many children but only one parent node.

**Trees In C++**

**Tree with its various parts.**

- **Root node:** This is the topmost node in the tree hierarchy. In the above diagram, Node A is the root node. Note that the root node doesn't have any parent.
- **Leaf node:** It is the Bottom most node in a tree hierarchy. Leaf nodes are the nodes that do not have any child nodes. They are also known as external nodes. Nodes E, F, G, H and C in the above tree are all leaf nodes.
- **Subtree:** Subtree represents various descendants of a node when the root is not null. A tree usually consists of a root node and one or more subtrees. In the above diagram, (B-E, B-F) and (D-G, D-H) are subtrees.
- **Parent node:** Any node except the root node that has a child node and an edge upward towards the parent.
- **Ancestor Node:** It is any predecessor node on a path from the root to that node. Note that the root does not have any ancestors. In the above diagram, A and B are the ancestors of E.
- **Key:** It represents the value of a node.

- ***Level:*** *Represents the generation of a node. A root node is always at level 1. Child nodes of the root are at level 2, grandchildren of the root are at level 3 and so on. In general, each node is at a level higher than its parent.*
- ***Path:*** *The path is a sequence of consecutive edges. In the above diagram, the path to E is A=>B->E.*
- ***Degree:*** *Degree of a node indicates the number of children that a node has. In the above diagram, the degree of B and D is 2 each whereas the degree of C is 0.*

***Tree types in c++:***

1. *General tree*
2. *Forest*
3. *Binary tree*
4. *Binary Search tree*
5. *Expression tree*

***Binary Tree:***

*A Binary tree is a widely used tree data structure. When each node of a tree has at most two child nodes then the tree is called a Binary tree.*

***So a typical binary tree will have the following components:***

1. *A left subtree*
2. *A root node*
3. *A right subtree*

***Binary Tree Types:***

1. *Full Binary Tree*
2. *Complete Binary tree*
3. *Perfect Binary tree*
4. *Balanced Binary*

- **Algorithm:-**

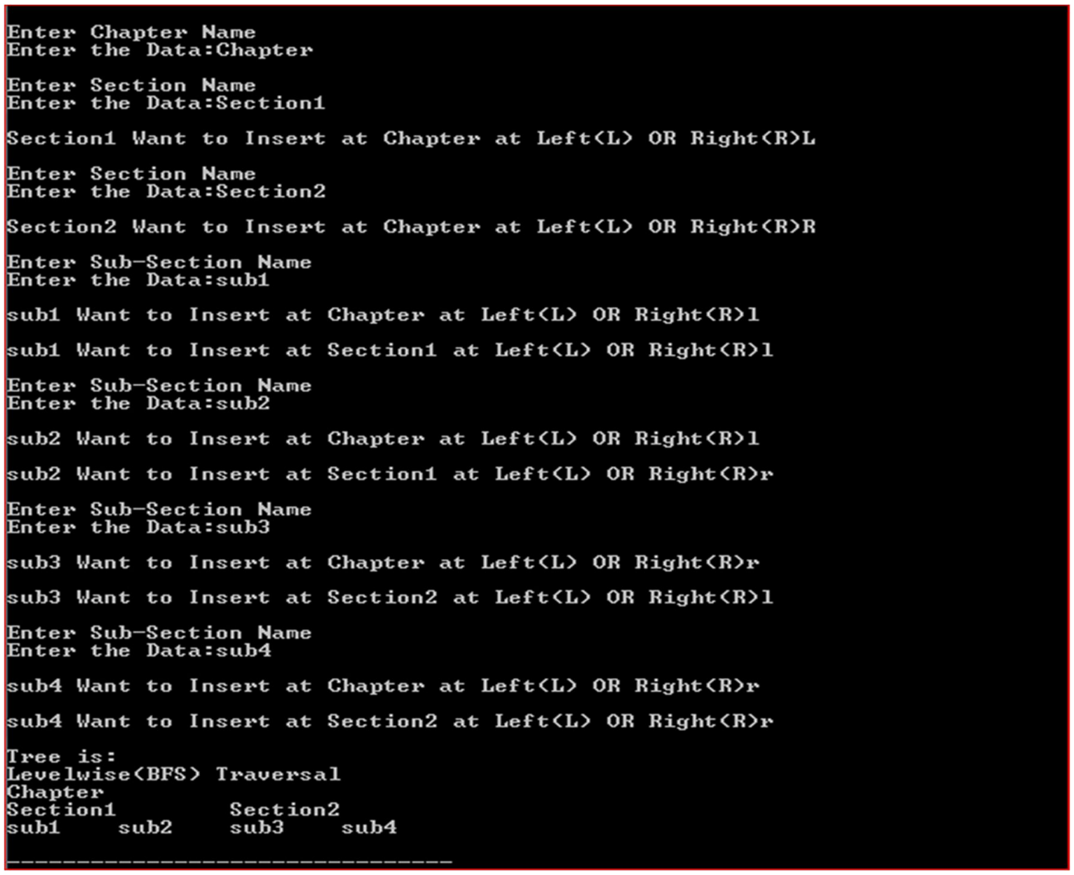
```
1) Start
        a. //Insertion
2) If root is NULL then create root node and return
3) If root exists then compare the data with node.data
        a. while until insertion position is located.
4) If data is greater than node.data
        a. Goto right subtree
5) Else. Goto left subtree
6) End while, insert data

        a. //Searching
7) If root.data is equal to search.data, return root
8) Else, while data not found
9) If data is greater than node.data, goto right subtree
10) Else ,goto left subtree
11) If data found, return node
12) End while
13) Return data not found

        a. //Display
14) Repeat until all nodes are traversed −
        a. Step 1 − Recursively traverse left subtree.
        b. Step 2 − Recursively traverse right subtree.
        c. Step 3 − Visit root node.
15) Stop
```

- **Program:-**

```
#include<iostream>
#include<stdio.h>
#include<queue>
using namespace std;
class Tree
 {
   typedef struct node
        {
                char data[10];
                struct node *left;
                struct node * right;
        }btree;
   public:
      btree *New,*root;
      Tree();
      void create();
      void insert(btree *root,btree *New);
      void display();
  };
Tree::Tree()
  {
    root=NULL;
  }

void Tree::create()
  {
     New=new btree;
     New->left=New->right=NULL;
     cout<<"\n\tEnter the Data: ";
     cin>>New->data;
     if(root==NULL)
           {
                   root=New;
           }
     else
        {
                   insert(root,New);
        }
   }

void Tree::insert(btree *root,btree *New)
  {
     char ans;
     cout<<"\n\t"<<New->data<<" Want to Insert at "<<root->data<<" at Left(L) OR Right(R)";
           cin>>ans;
     if(ans=='L'||ans=='l')
        {
             if(root->left==NULL)
                   root->left=New;
           else
               insert(root->left,New);
           }
     else
        {
             if(root->right==NULL)
                   root->right=New;
           else
               insert(root->right,New);
           }
```

```
 }

void Tree::display()
{
        int i=1;
        if(root==NULL){
                cout<<"\n NULL Tree";
                return;
        }

        queue<btree *> q;
        q.push(root);

        cout<<"\n\tLevelwise(BFS) Traversal\n";
        while(q.empty()==false)
        {
                btree *node=q.front();
                if(i==1)
                        cout<<node->data<<"\n";
                if(i==2)
                        cout<<node->data<<"\t";
                if(i==3)
                        cout<<node->data<<"\n";
                if(i==4||i==5||i==6||i==7)
                        cout<<node->data<<"\t";

                i++;
                q.pop();
                if(node->left!=NULL)
                        q.push(node->left);
                if(node->right!=NULL)
                        q.push(node->right);
        }
}



int main()
 {
   Tree tr;
   int i=0;
   do
     {
       if(i==0)
        {
                cout<<"\n\tEnter Chapter Name";
                tr.create();
                i++;
        }
      if(i==1||i==2)
        {
                cout<<"\n\tEnter Section Name";
            tr.create();
            i++;
        }
      if(i==3||i==4||i==5||i==6)
        {
                cout<<"\n\tEnter Sub-Section Name";
            tr.create();
            i++;
         }
         if(i==7)
         {
           cout<<"\n tree is:";
                tr.display();
            break;
         }
   }while(1);
}
```

- ***Output:-***



```
Enter Chapter Name
Enter the Data:Chapter

Enter Section Name
Enter the Data:Section1

Section1 Want to Insert at Chapter at Left(L) OR Right(R)L

Enter Section Name
Enter the Data:Section2

Section2 Want to Insert at Chapter at Left(L) OR Right(R)R

Enter Sub-Section Name
Enter the Data:sub1

sub1 Want to Insert at Chapter at Left(L) OR Right(R)l

sub1 Want to Insert at Section1 at Left(L) OR Right(R)l

Enter Sub-Section Name
Enter the Data:sub2

sub2 Want to Insert at Chapter at Left(L) OR Right(R)l

sub2 Want to Insert at Section1 at Left(L) OR Right(R)r

Enter Sub-Section Name
Enter the Data:sub3

sub3 Want to Insert at Chapter at Left(L) OR Right(R)r

sub3 Want to Insert at Section2 at Left(L) OR Right(R)l

Enter Sub-Section Name
Enter the Data:sub4

sub4 Want to Insert at Chapter at Left(L) OR Right(R)r

sub4 Want to Insert at Section2 at Left(L) OR Right(R)r

Tree is:
Levelwise(BFS) Traversal
Chapter
Section1        Section2
sub1    sub2    sub3    sub4

--------------------------------
```

- ***Analysis:-***

***Time Complexity:***

1 *Creating node: O(1)*
2 *Inserting Node : O(1)*
3 *Displaying Tree : O(n)* *{n=number of node tree have}*

***Space Complexity:***

*Space complexity: O(h)* *{h= height of tree}*
*O(n)* *(in worst case )*

- ***Conclusion:-***

*Hence, Trees are a non-linear hierarchical data structure that is used in many applications in the software field. Unlike linear data structures that have only one way to traverse the list, we can traverse trees in a variety of ways.*

# Experiment Number:-4

- ## Aim:-

Beginning with an empty binary search tree, Construct binary search tree by inserting the values in the order given. After constructing a binary tree –

1)Insert new node
2) Find number of nodes from longest path from root
3) Minimum data value found in the tree
4) change a tree so that the roles of the left & right pointers are swapped at every node
5) search a value.

- ## Objective:-

1) To learn the basics of tree data structure in C++ and apply it.

- ## Theory:-

**Definition:**

A tree T is a set of nodes storing elements such that the nodes have a parent-child relationship that satisfies the following
• if T is not empty, T has a special tree called the root that has no parent
• each node v of T different than the root has a unique parent node w; each node with parent w is a child of w

**Recursive definition**
• T is either empty
• or consists of a node r (the root) and a possibly empty set of trees whose roots are the children of r

Tree is a widely-used data structure that emulates a tree structure with a set of linked nodes. The tree graphical is represented most commonly as on Picture 1. The circles are the nodes and the edges are the links between them.

Trees are usually used to store and represent data in some hierarchical order. The data are stored in the nodes, from which the tree is consisted of.
A node may contain a value or a condition or represent a separate data structure or a tree of its own. Each node in a tree has zero or more child nodes, which are one level lower in the tree hierarchy (by convention, trees grow down, not up as they do in nature). A node that has a child is called the child's parent node (or ancestor node, or superior). A node has at most one parent. A node that has no child is called a leaf, and that node is of course at the bottom most Level of the tree. The height of a node is the length of the longest path to a leaf from that node.

The height of the root is the height of the tree. In other words, the "height" of tree is the "number

*of levels" in the tree. Or more formal, the height of a tree is defined as follows:*
*1. The height of a tree with no elements is 0*
*2. The height of a tree with 1 element is 1*
*3. The height of a tree with > 1 element is equal to 1 + the height of its tallest subtree.*

*The depth of a node is the length of the path to its root (i.e., its root path). Every child node is always one level lower than his parent.*

*The topmost node in a tree is called the root node. Being the topmost node, the root node will not have parents. It is the node at which operations on the tree commonly begin (although some algorithms begin with the leaf nodes and work up ending at the root). All other nodes can be reached from it by following edges or links. (In the formal definition, a path from a root to a Node, for each different node is always unique). In diagrams, it is typically drawn at the top. In some trees, such as heaps, the root node has special properties.*

*A subtree is a portion of a tree data structure that can be viewed as a complete tree in itself. Any node in a tree T, together with all the nodes below his height, that are reachable from the node, comprise a subtree of T. The subtree corresponding to the root node is the entire tree; the subtree corresponding to any other node is called a proper subtree (in analogy to the term proper subset).*

*Every node in a tree can be seen as the root node of the subtree rooted at that node. An internal node or inner node is any node of a tree that has child nodes and is thus not a leaf node. There are two basic types of trees. In an unordered tree, a tree is a tree in a purely structural sense — that is to say, given a node, there is no order for the children of that node.*

*A tree on which an order is imposed — for example, by assigning different natural numbers to each child of each node — is called an ordered tree, and data structures built on them are called ordered tree data structures. Ordered trees are by far the most common form of tree data structure. Binary search trees are one kind of ordered tree.*

***Important Terms***

*Following are the important terms with respect to tree.*

1) ***Path*** *− Path refers to the sequence of nodes along the edges of a tree.*
2) ***Root*** *− The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.*
3) ***Parent*** *− Any node except the root node has one edge upward to a node called parent.*
4) ***Child*** *− The node below a given node connected by its edge downward is called its child node.*
5) ***Leaf*** *− The node which does not have any child node is called the leaf node.*
6) ***Subtree*** *− Subtree represents the descendants of a node.*
7) ***Visiting*** *− Visiting refers to checking the value of a node when control is on the node.*
8) ***Traversing*** *− Traversing means passing through nodes in a specific order.*
9) ***Levels*** *− Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.*
10) ***keys*** *− Key represents a value of a node based on which a search operation is to be carried out for a node.*

***Advantages of trees***

*Trees are so useful and frequently used, because they have some very serious advantages:*

1) *Trees reflect structural relationships in the data*
2) *Trees are used to represent hierarchies*
3) *Trees provide an efficient insertion and searching*
4) *Trees are very flexible data, allowing to move subtrees around with minimum effort.*

- ## ***Algorithm:-***

1) *Start*
    a. *//Insertion*
2) *If root is NULL then create root node and return*
3) *If root exists then compare the data with node.data*
    a. *while until insertion position is located.*
4) *If data is greater than node.data*
    a. *Goto right subtree*
5) *Else. Goto left subtree*
6) *End while, insert data*

    a. *//Searching*
7) *If root.data is equal to search.data, return root*
8) *Else, while data not found*
9) *If data is greater than node.data, goto right subtree*
10) *Else ,goto left subtree*
11) *If data found, return node*
12) *End while*
13) *Return data not found.*
    a. *//Deletion*
14) *Starting at root, find the deepest and rightmost node in binary tree and node which we want to delete.*
15) *Replace the deepest rightmost node's data with node to be deleted.*
16) *Then delete the deepest rightmost node.*
    a. *//Display*
17) *Repeat until all nodes are traversed −*
    a. *Step 1 − Recursively traverse left subtree.*
    b. *Step 2 − Recursively traverse right subtree.*
    c. *Step 3 − Visit root node.*
    d. *//Minimum Value*
18) *Just traverse the node from root to left recursively until left is NULL. The node whose left is NULL is the node with minimum value.*
    a. *//Maximum Value*
19) *Just traverse the node from root to right recursively until right is NULL. The node whose right is NULL is the node with maximum value.*
20) *Stop*

- ***Program:-***

```
#include <iostream>
using namespace std;
#define COUNT 10

//Building a BST by structure tree

struct tree
{
    int key,count,depth;
    struct tree *left,*right;
};

//Inserting a node in the tree. The function would not add duplicate nodes. But the count of how many times the
//node appears in the tree is stored.

struct tree *insert(struct tree *root,int value)
{
    if(root==NULL)
    {
        root=(struct tree *)malloc(sizeof(struct tree));
        root->key=value;
        root->count=1;
        root->left=NULL;
        root->right=NULL;
        return root;
    }
    //Adding the count when found a duplicate node
    else if (value == root->key)
    {
        (root->count)++;
        return root;
    }
    else
    {
        //Insert in right substree
        if(root->key < value)
            root->right=insert(root->right,value);
        else
        {
        //Insert in left subtree
            if(root->key > value)
                root->left=insert(root->left,value);
        }
    }
    return root;
}

//Deleting a node from tree
struct tree *deletion(struct tree *root,int value)
{
    if(root == NULL)
        return root;
```

```
if(root->key < value)
{
    //The node to be deleted is in right subtree
    root->right=deletion(root->right,value);
    return root;
}

else if(root->key > value)
{
    //The node to be deleted is in the left subtree
    root->left=deletion(root->left,value);
    return root;
}

else
{
    //Decreasing the count of deleted node
    if (root->count > 1)
    {
           (root->count)--;
           return root;
    }

    if(root->left == NULL)
    {
        struct tree *temp=root->right;
        delete root;
        return temp;
    }

    else if(root->right == NULL)
    {
        struct tree *temp=root->left;
        delete root;
        return temp;
    }

    //Finding the sucessor
    else
    {
        struct tree *succparent=root->right;
        struct tree *succ=root->right;
        while(succ->left != NULL)
        {
            succparent=succ;
            succ=succ->left;
        }
        succparent->left=succ->right;
        root->key=succ->key;
        delete succ;
        return root;
    }
}
```

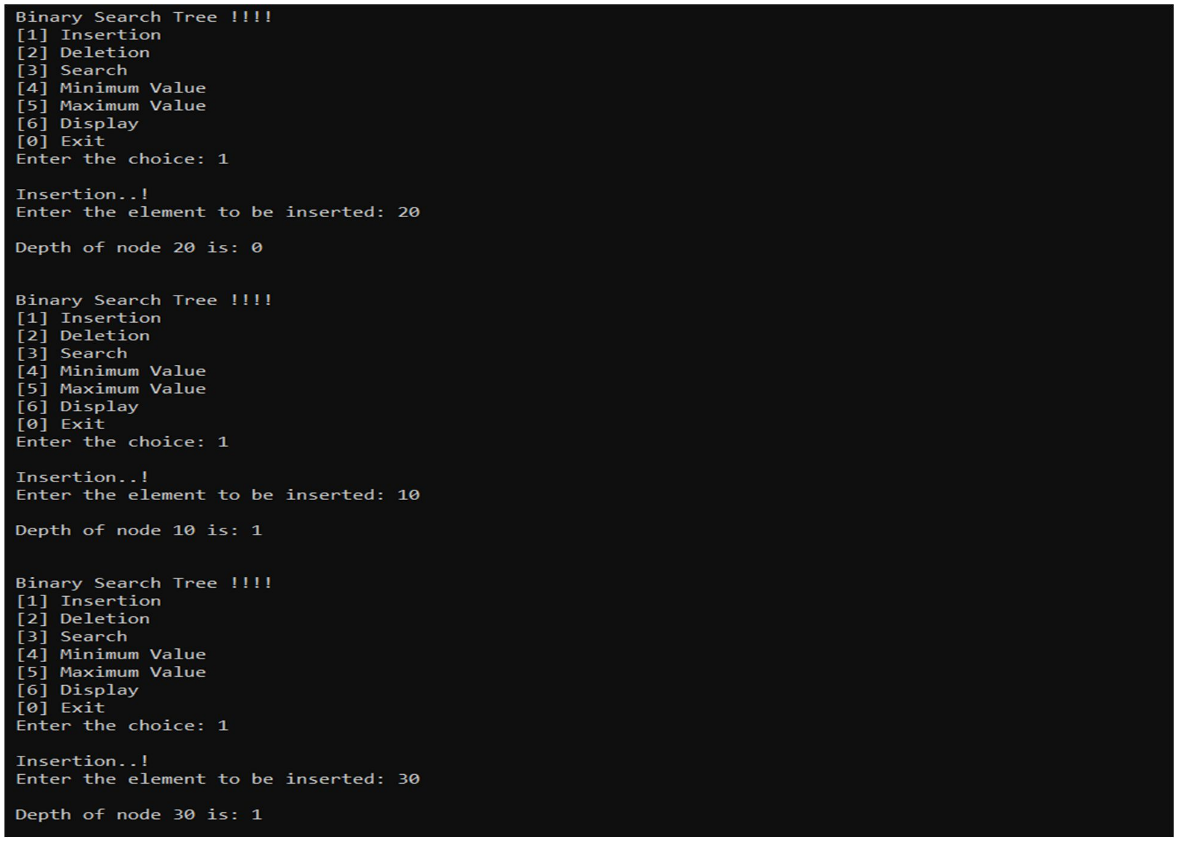
```
}

//Minimum Value in the tree
struct tree *minimumval(struct tree *root)
{
    struct tree *current=root;
    while(current->left !=NULL)
        current=current->left;
    return current;
}

//Maximum Value in the tree
struct tree *maximumval(struct tree *root)
{
   struct tree *current=root;
    while(current->right !=NULL)
        current=current->right;
    return current;
}

//Searching a node in the tree
struct tree *search(struct tree *root, int value)
{
    if (root == NULL || root->key == value)
       return root;

    //Search in right subtree

    if (root->key < value)
       return search(root->right, value);

    //Search in left subtree

    return search(root->left, value);
}


//Finding depth of a node
int depth(struct tree *root,int value, int level)
{

    if(root == NULL)
        return 0;

    if(root->key == value)
        return level;
    int temp = depth(root->left,value,level+1);
    if(temp != 0)
        return temp;

    temp=depth(root->right,value,level+1);
    return temp;
}
```

```
void print2DUtil(struct tree *root, int level)
{
   // Base case
   if (root == NULL)
       return;

   // Increase distance between levels
   level += COUNT;

   // Process right child first
   print2DUtil(root->right, level);

   // Print current node after space
   // count
   cout<<endl;
   for (int i = COUNT; i < level; i++)
       cout<<" ";
   cout<<root->key<<"\n";

   // Process left child
   print2DUtil(root->left, level);
}

// Wrapper over print2DUtil()
void print2D(struct tree *root)
{
   // Pass initial space count as 0
   print2DUtil(root, 0);
}




int main()
{
       int choice=0,value=0,value1=0;
       struct tree *root=NULL,*searchh=NULL,*position=NULL;
   do
   {
       //Menu for selecting the operation to perform
       cout<<"\n\n";
       cout<<"\n Binary Search Tree !!!!";
       cout<<"\n [1] Insertion ";
       cout<<"\n [2] Deletion ";
       cout<<"\n [3] Search ";
       cout<<"\n [4] Minimum Value ";
       cout<<"\n [5] Maximum Value ";
       cout<<"\n [6] Display ";
       cout<<"\n [0] Exit ";

       cout<<"\n Enter the choice: ";
       cin>>choice;
```

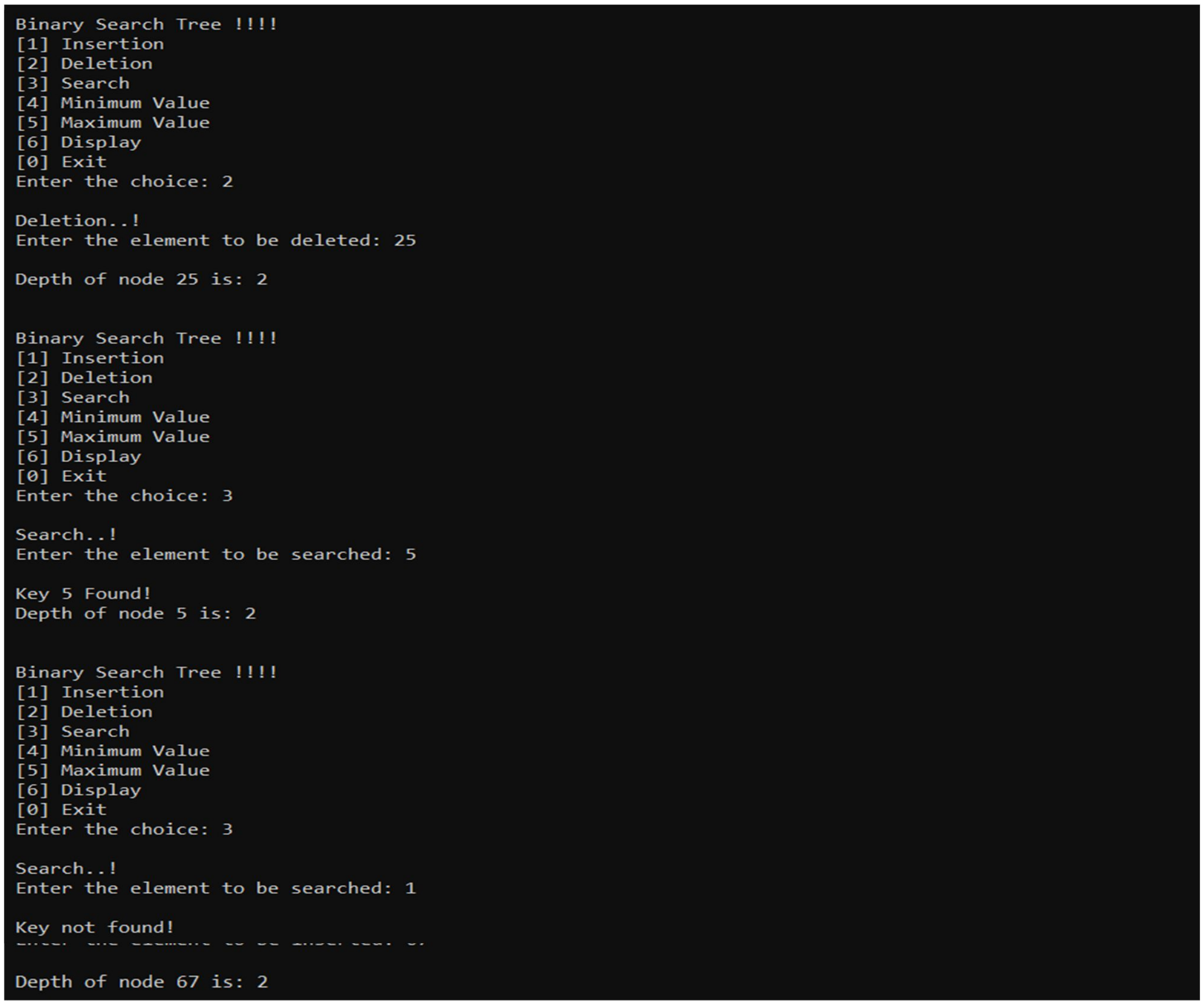
```
switch(choice)
{
   case 1:
         cout<<"\n Insertion..!";
         cout<<"\n Enter the element to be inserted: ";
         cin>>value;
         root=insert(root,value);
         root->depth=depth(root,value,0);
         cout<<"\n Depth of node "<<value<<" is: "<<root->depth;
         break;
   case 2:
         cout<<"\n Deletion..!";
         cout<<"\n Enter the element to be deleted: ";
         cin>>value;
         if(search(root,value) == NULL)
            cout<<"\n Deletion Unsuccessful. Value does not exists in the tree!";
         else
         {
            root->depth=depth(root,value,0);
            cout<<"\n Depth of node "<<value<<" is: "<<root->depth;
            root=deletion(root,value);
         }
         break;
   case 3:
         cout<<"\n Search..!";
         cout<<"\n Enter the element to be searched: ";
         cin>>value;
         searchh=search(root,value);
         if(searchh == NULL)
            cout<<"\n Key not found!";
         else
         {
            root->depth=depth(root,value,0);
            cout<<"\n"<<" Key "<<searchh->key<<" Found!";
            cout<<"\n Depth of node "<<value<<" is: "<<root->depth;
         }
         break;
   case 4:
         cout<<"\n Minimum Value..!";
         if(root == NULL)
            cout<<"\n No minimum values in empty tree";
         else
         {
            value=minimumval(root)->key;
            root->depth=depth(root,value,0);
            cout<<"\n Smallest value in the tree: "<<value;
            cout<<"\n Depth of node "<<value<<" is: "<<root->depth;
         }
         break;
   case 5:
         cout<<"\n Maximum Value..!";
         if(root == NULL)
            cout<<"\n No maximum values in empty tree";
         else
```

```
            {
               value=maximumval(root)->key;
               root->depth=depth(root,value,0);
               cout<<"\n Largest value in the tree: "<<value;
               cout<<"\n Depth of node "<<value<<" is: "<<root->depth;
            }

            break;
        case 6:{
         cout<<"\n BST Display";
         print2D(root);
                                break;
                  }

        case 0:

            cout<<"\n Exiting..!";
            break;
        default:
            cout<<"\n Invalid Choice!";
            break;
      }

    }while(choice!=0);

    return 0;
}
```

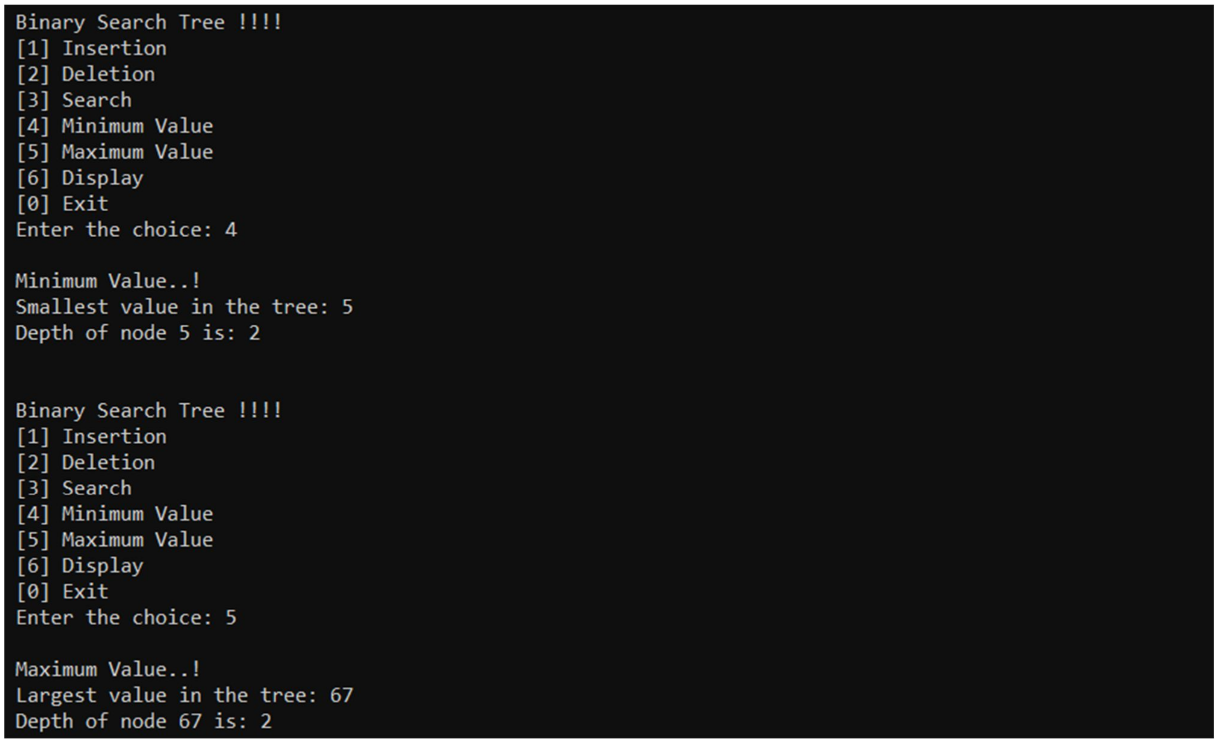
- ***Output:-***



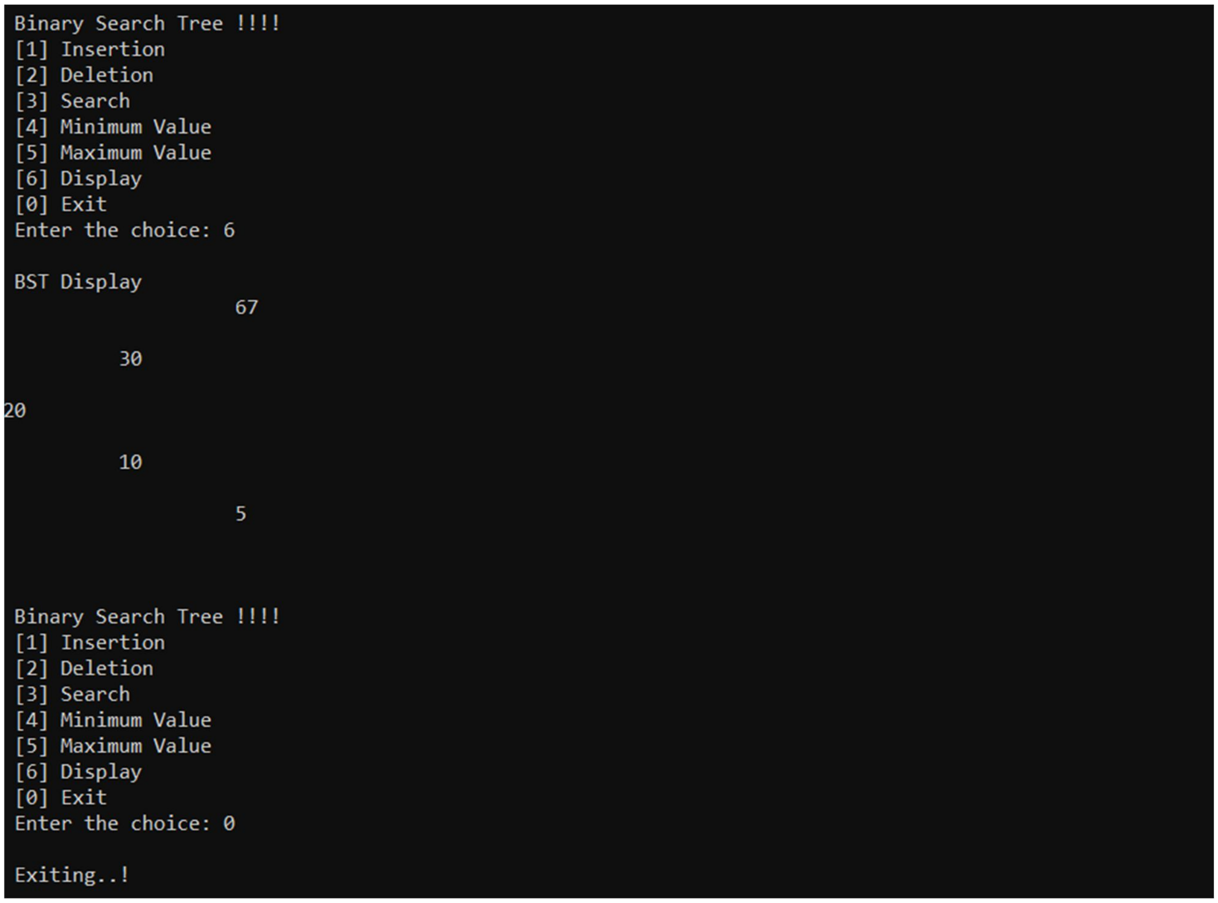
```
Binary Search Tree !!!!
[1] Insertion
[2] Deletion
[3] Search
[4] Minimum Value
[5] Maximum Value
[6] Display
[0] Exit
Enter the choice: 1

Insertion..!
Enter the element to be inserted: 20

Depth of node 20 is: 0


Binary Search Tree !!!!
[1] Insertion
[2] Deletion
[3] Search
[4] Minimum Value
[5] Maximum Value
[6] Display
[0] Exit
Enter the choice: 1

Insertion..!
Enter the element to be inserted: 10

Depth of node 10 is: 1


Binary Search Tree !!!!
[1] Insertion
[2] Deletion
[3] Search
[4] Minimum Value
[5] Maximum Value
[6] Display
[0] Exit
Enter the choice: 1

Insertion..!
Enter the element to be inserted: 30

Depth of node 30 is: 1
```



```
Binary Search Tree !!!!
[1] Insertion
[2] Deletion
[3] Search
[4] Minimum Value
[5] Maximum Value
[6] Display
[0] Exit
Enter the choice: 2

Deletion..!
Enter the element to be deleted: 25

Depth of node 25 is: 2


Binary Search Tree !!!!
[1] Insertion
[2] Deletion
[3] Search
[4] Minimum Value
[5] Maximum Value
[6] Display
[0] Exit
Enter the choice: 3

Search..!
Enter the element to be searched: 5

Key 5 Found!
Depth of node 5 is: 2


Binary Search Tree !!!!
[1] Insertion
[2] Deletion
[3] Search
[4] Minimum Value
[5] Maximum Value
[6] Display
[0] Exit
Enter the choice: 3

Search..!
Enter the element to be searched: 1

Key not found!

Depth of node 67 is: 2
```



```
Binary Search Tree !!!!
[1] Insertion
[2] Deletion
[3] Search
[4] Minimum Value
[5] Maximum Value
[6] Display
[0] Exit
Enter the choice: 4

Minimum Value..!
Smallest value in the tree: 5
Depth of node 5 is: 2


Binary Search Tree !!!!
[1] Insertion
[2] Deletion
[3] Search
[4] Minimum Value
[5] Maximum Value
[6] Display
[0] Exit
Enter the choice: 5

Maximum Value..!
Largest value in the tree: 67
Depth of node 67 is: 2
```



```
Binary Search Tree !!!!
[1] Insertion
[2] Deletion
[3] Search
[4] Minimum Value
[5] Maximum Value
[6] Display
[0] Exit
Enter the choice: 6

BST Display
                    67

          30

20

          10

                    5



Binary Search Tree !!!!
[1] Insertion
[2] Deletion
[3] Search
[4] Minimum Value
[5] Maximum Value
[6] Display
[0] Exit
Enter the choice: 0

Exiting..!
```

- ## *Analysis:-*

***Time Complexity:***

4 *Creating node: O(1)*
5 *Inserting Node : O(1)*
6 *Displaying Tree : O(n)* *{n=number of node tree have}*

***Space Complexity:***

*Space complexity: O(h)* *{h= height of tree}*
*O(n)* *(in worst case )*

- ## *Conclusion:-*

*Hence, Trees are a non-linear hierarchical data structure that is used in many applications in the software field. Unlike linear data structures that have only one way to traverse the list, we can traverse trees in a variety of ways.*

# *Experiment Number:-5*

- ***Aim:-***

*Construct an expression tree from the given prefix expression eg. +--a*bc/def and traverse it using post order traversal (non recursive) and then delete the entire tree.*

- ***Objectives:-***

*To understand concept of Tree & Binary Tree.*
*To analyse the working of various Tree operations.*

- ***Theory:-***

***Tree***

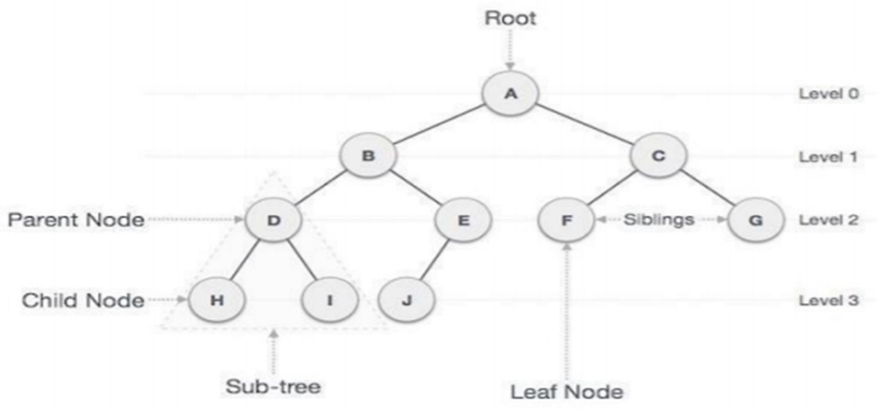
*Tree represents the nodes connected by edges also a class of graphs that is acyclic is termed as Trees. Let us now discuss an important class of graphs called trees and its associated terminology.*

*Trees are useful in describing any structure that involves hierarchy. Familiar examples of such structures are family trees, the hierarchy of positions in an organization, and so on.*

***Binary Tree***

*A binary tree is made of nodes, where each node contains a "left" reference, a "right" reference, and a data element. The topmost node in the tree is called the root.*

*Every node (excluding a root) in a tree is connected by a directed edge from exactly one other node. This node is called a parent. On the other hand, each node can be connected to arbitrary number of nodes, called children. Nodes with no children are called leaves, or external nodes. Nodes which are not leaves are called internal nodes. Nodes with the same parent are called siblings.*



### *Insert Operation*

*The very first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.*

### *Traversals*

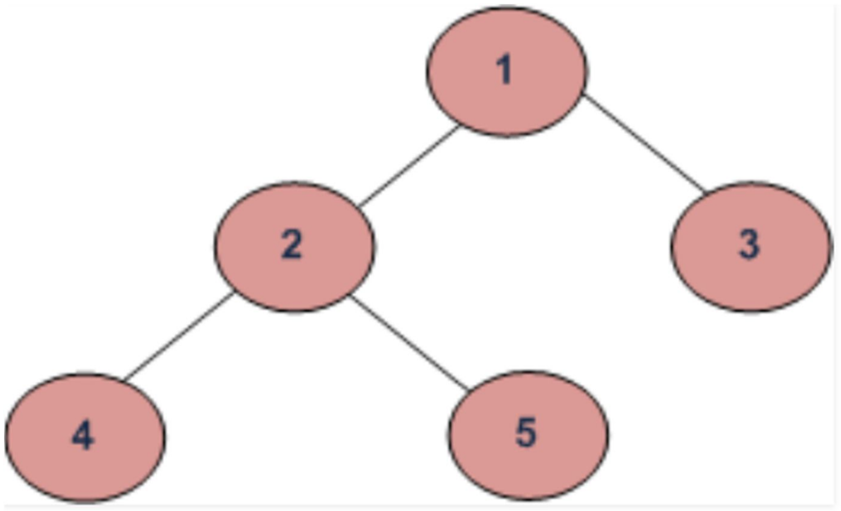
*A traversal is a process that visits all the nodes in the tree. Since a tree is a non-linear data structure, there is no unique traversal. We will consider several traversal algorithms with we group in the following two kinds*

1) *depth-first traversal*
2) *breadth-first traversal*

*There are three different types of depth-first traversals, :*

1) *Pre-order traversal - visit the parent first and then left and right children;*
2) *In Order traversal - visit the left child, then the parent and the right child;*
3) *Post-order traversal - visit left child, then the right child and then the parent;*

*There is only one kind of breadth-first traversal the level order traversal. This traversal visits nodes by levels from top to bottom and from left to right. As an example consider the following tree and its four traversals:*



Depth First Traversals:

(a) Inorder (Left, Root, Right) : 4 2 5 1 3

(b) Preorder (Root, Left, Right) : 1 2 4 5 3

(c) Postorder (Left, Right, Root) : 4 5 2 3 1

- ***Algorithm:-***

1) *Algorithm to insert a node :*
    a. *Step 1 -*
    b. *Search for the node whose child node is to be inserted. This is a node at some level i, and a node is to be inserted at the level i +1 as either its left child or right child. This is the node after which the insertion is to be made.*
    c. *Step 2 -*
    d. *Link a new node to the node that becomes its parent node, that is, either the L child or the R child.*

2) *Algorithm to traverse a tree :*
    a. *In order traversal*
    b. *Until all nodes are traversed −*
    c. *Step 1 − Recursively traverse left subtree.*
    d. *Step 2 − Visit root node.*
    e. *Step 3 − Recursively traverse right subtree.*

3) *Pre-order traversal*
    a. *Until all nodes are traversed −*
    b. *Step 1 − Visit root node.*
    c. *Step 2 − Recursively traverse left subtree.*
    d. *Step 3 − Recursively traverse right subtree*

4) *Post-order  traversal*
    a. *Until all nodes are traversed −*
    b. *Step 1 − Recursively traverse left subtree.*
    c. *Step 2 − Recursively traverse right subtree.*
    d. *Step 3 − Visit root node*

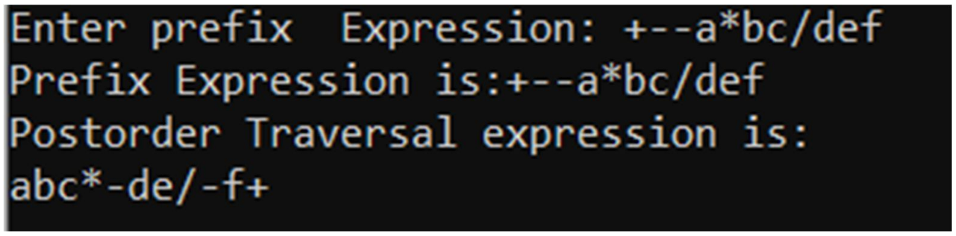
5) *Algorithm to copy one tree into another tree:*
    a. *Step 1 – if (Root == Null) Then return Null*
    b. *Step 2 −Tmp = new TreeNode*
    c. *Step 3 – Tmp->Lchild = TreeCopy(Root- >Lchild);*
    d. *Step 4 – Tmp->Rchild = TreeCopy(Root->Rchild);*
    e. *Step 5 – Tmp-Data = Root->Data;*
    f. *Then return*

- ***Program:-***

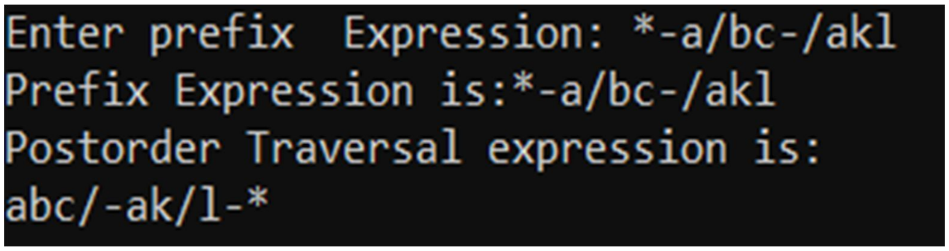
```
 #include <iostream>
 #include<string.h>
using namespace std;

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

class tree
{       char prefix[20];
        public: node *top;
                void expression(char []);
                void display(node *);
                void non_rec_postorder(node *);
                void del(node *);

};

class stack1
  {
        node *data[30];
        int top;
        public:
        stack1()
        {
                top=-1;
        }
                int empty()
                  {
                        if(top==-1)
                                return 1;
                        return 0;
                  }
        void push(node *p)
                  {
                        data[++top]=p;
                  }
        node *pop()
          {
                  return(data[top--]);
          }
```

```
};

void tree::expression(char prefix[])
{char c;
stack1 s;
node *t1,*t2;
int len,i;
 len=strlen(prefix);

        for(i=len-1;i>=0;i--){
                top=new node;
                top->left=NULL;
                top->right=NULL;

                if(isalpha(prefix[i]))
                {
                        top->data=prefix[i];
                        s.push(top);
                }
                else if(prefix[i]=='+'||prefix[i]=='*'||prefix[i]=='-'||prefix[i]=='/')
                {
                        t2=s.pop();
                        t1=s.pop();
                        top->data=prefix[i];
                        top->left=t2;
                        top->right=t1;
                        s.push(top);
      }
        }
        top=s.pop();

}

void tree::display(node * root)
{
        if(root!=NULL)
        {       cout<<root->data;
                display(root->left);
                display(root->right);
         }


}

void tree::non_rec_postorder(node *top)
{
	stack1 s1,s2;	/*stack s1 is being used for flag . A NULL data implies that the right
subtree has not been visited */
	node *T=top;
	cout<<"\n";
	s1.push(T);
	while(!s1.empty())
	{
		T=s1.pop();
		s2.push(T);
		if(T->left!=NULL)
		s1.push(T->left);
		if(T->right!=NULL)
		s1.push(T->right);
	}
	while(!s2.empty())
	{
		top=s2.pop();
		cout<<top->data;
	}
}

void tree::del(node* node)
{
   if (node == NULL) return;
     /* first delete both subtrees */
   del(node->left);
   del(node->right);
      /* then delete the node */
   cout<<" Deleting node:"<<node->data;
   free(node);
}

int main()
{
	char expr[20];
	tree t;

	cout<<"Enter prefix  Expression: ";
	cin>>expr;
	cout<<"Prefix Expression is:"<<expr;
	cout<<"\n";
	t.expression(expr);
	cout<<"Postorder Traversal expression is:";
	t.non_rec_postorder(t.top);

}
```

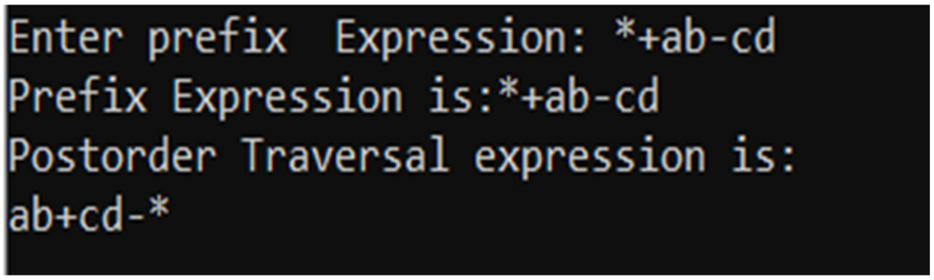
- ***Output:-***



```
Enter prefix  Expression: +--a*bc/def
Prefix Expression is:+--a*bc/def
Postorder Traversal expression is:
abc*-de/-f+
```



```
Enter prefix  Expression: *-a/bc-/akl
Prefix Expression is:*-a/bc-/akl
Postorder Traversal expression is:
abc/-ak/l-*
```



```
Enter prefix  Expression: *+ab-cd
Prefix Expression is:*+ab-cd
Postorder Traversal expression is:
ab+cd-*
```

- ***Analysis:-***

***Time Complexity:***

*Creating node: O(1)*

*Inserting Node : O(1)*

*Displaying Tree : O(n)* *{n=number of node tree have}*

***Space Complexity:***

*Space complexity: O(h)* *{h= height of tree}*

***O(n)*** ***(in worst case )***

- ***Conclusion:-***

*Thus we have studied the implementation of various Binary tree operations*