**Experiment No.6**

**Title:**  Implementation of Operations in BST using C++

**Problem Statement:**

**6a**. Program to demonstrate insertion in a BST recursively.  
**6b**. Program to demonstrate Binary Tree Traversal.  
**6c**. Program to demonstrate Search operation in Binary Tree.

**Algorithm:**

**6a.Insertion in BST**

**Step 1:** Check, whether value in current node and a new value are equal. If so, duplicate is found. Otherwise,

**Step 2:** If a new value is less, than the node's value:

**a)**if a current node has no left child, place for insertion has been found;

**b)**otherwise, handle the left child with the same algorithm.

**Step 3:** If a new value is greater, than the node's value:

**a)**if a current node has no right child, place for insertion has been found;

**b)**otherwise, handle the right child with the same algorithm.

**6b. Binary Tree Traversal**

**Inoder Traversal:**

**Step 1:**Traverse the left subtree, i.e., call Inorder(left-subtree)

**Step 2:** Visit the root.

**Step 3:** Traverse the right subtree, i.e., call Inorder(right-sub.

**Preorder Traversal:**

**Step 1:** Visit the root.

**Step 2:** Traverse the left subtree, i.e., call Preorder(left-subtree)

**Step 3:** Traverse the right subtree, i.e., call Preorder(right-subtree)

**Postorder Traversal:**

**Step 1:** Traverse the left subtree, i.e., call Postorder(left-subtree)

**Step 2:** Traverse the right subtree, i.e., call Postorder(right-subtree)

**Step 3:** Visit the root.

**6c. Search operation in Binary Tree.**

TREE-SEARCH(x, k)  
     **if** x == NIL or k == x.key  
         **return** x  
     **if** k < x.key  
         **return** TREE-SEARCH(x.left, k)  
     **else return** TREE-SEARCH(x.right, k)

**Code:**

**Expt 6a:C++ program to demonstrate insertion in a BST recursively.**

#include <iostream>

using namespace std;

class BST

{

int data;

BST \*left, \*right;

public:

// Default constructor.

BST();

// Parameterized constructor.

BST(int);

// Insert function.

BST\* Insert(BST \*, int);

// Inorder traversal.

void Inorder(BST \*);

};

// Default Constructor definition.

BST :: BST() : data(0), left(NULL), right(NULL){}

// Parameterized Constructor definition.

BST :: BST(int value)

{

data = value;

left = right = NULL;

}

// Insert function definition.

BST\* BST :: Insert(BST \*root, int value)

{

if(!root)

{

// Insert the first node, if root is NULL.

return new BST(value);

}

// Insert data.

if(value > root->data)

{

// Insert right node data, if the 'value'

// to be inserted is greater than 'root' node data.

// Process right nodes.

root->right = Insert(root->right, value);

}

else

{

// Insert left node data, if the 'value'

// to be inserted is greater than 'root' node data.

// Process left nodes.

root->left = Insert(root->left, value);

}

// Return 'root' node, after insertion.

return root;

}

// Inorder traversal function.

// This gives data in sorted order.

void BST :: Inorder(BST \*root)

{

if(!root)

{

return;

}

Inorder(root->left);

cout << root->data << endl;

Inorder(root->right);

}

// Driver code

int main()

{

BST b, \*root = NULL;

cout<<"Binary tree :"<<endl;

root = b.Insert(root, 50);

b.Insert(root, 30);

b.Insert(root, 20);

b.Insert(root, 40);

b.Insert(root, 70);

b.Insert(root, 60);

b.Insert(root, 80);

b.Inorder(root);

return 0;

}

**Expt 6b:C++ program to demonstrate Binary Tree Traversal.**

#include<iostream>

using namespace std;

//binary tree node declaration

struct bintree\_node{

bintree\_node \*left;

bintree\_node \*right;

char data;

} ;

class bintree\_class{

bintree\_node \*root;

public:

bintree\_class(){

root=NULL;

}

int isempty() {

return(root==NULL);

}

void insert\_node(int item);

void inorder\_seq();

void inorder(bintree\_node \*);

void postorder\_seq();

void postorder(bintree\_node \*);

void preorder\_seq();

void preorder(bintree\_node \*);

};

void bintree\_class::insert\_node(int item){

bintree\_node \*p=new bintree\_node;

bintree\_node \*parent;

p->data=item;

p->left=NULL;

p->right=NULL;

parent=NULL;

if(isempty())

root=p;

else{

bintree\_node \*ptr;

ptr=root;

while(ptr!=NULL) {

parent=ptr;

if(item>ptr->data)

ptr=ptr->right;

else

ptr=ptr->left;

}

if(item<parent->data)

parent->left=p;

else

parent->right=p;

}

}

void bintree\_class::inorder\_seq()

{

inorder(root);

}

void bintree\_class::inorder(bintree\_node \*ptr)

{

if(ptr!=NULL){

inorder(ptr->left);

cout<<" "<<ptr->data<<" ";

inorder(ptr->right);

}

}

void bintree\_class::postorder\_seq()

{

postorder(root);

}

void bintree\_class::postorder(bintree\_node \*ptr)

{

if(ptr!=NULL){

postorder(ptr->left);

postorder(ptr->right);

cout<<" "<<ptr->data<<" ";

}

}

void bintree\_class::preorder\_seq()

{

preorder(root);

}

void bintree\_class::preorder(bintree\_node \*ptr)

{

if(ptr!=NULL){

cout<<" "<<ptr->data<<" ";

preorder(ptr->left);

preorder(ptr->right);

}

}

int main()

{

bintree\_class bintree;

bintree.insert\_node('A');

bintree.insert\_node('B');

bintree.insert\_node('C');

bintree.insert\_node('D');

bintree.insert\_node('E');

bintree.insert\_node('F');

bintree.insert\_node('G');

cout<<"Inorder traversal:"<<endl;

bintree.inorder\_seq();

cout<<endl<<"Postorder traversal:"<<endl;

bintree.postorder\_seq();

cout<<endl<<"Preorder traversal:"<<endl;

bintree.preorder\_seq();

}

**//Expt 6c: C++ program to demonstrate Search operation in Binary Tree.**

#include<iostream>

using namespace std;

struct node {

int d;

node \*left;

node \*right;

};

node\* CreateNode(int d) {

node \*newnode = new node;

newnode->d = d;

newnode->left = NULL;

newnode->right = NULL;

return newnode;

}

node\* InsertIntoTree(node\* root, int d) {

node \*temp = CreateNode(d);

node \*t = new node;

t = root;

if(root == NULL)

root = temp;

else {

while(t != NULL) {

if(t->d < d) {

if(t->right == NULL) {

t->right = temp;

break;

}

t = t->right;

} else if(t->d > d) {

if(t->left == NULL) {

t->left = temp;

break;

}

t = t->left;

}

}

}

return root;

}

void Search(node \*root, int d) {

int depth = 0;

node \*temp = new node;

temp = root;

while(temp != NULL) {

depth++;

if(temp->d == d) {

cout<<"\nitem found at depth: "<<depth;

return;

} else if(temp->d > d)

temp = temp->left;

else

temp = temp->right;

}

cout<<"\n item not found";

return;

}

int main() {

char ch;

int n, i, a[10] = {93, 53, 45, 2, 7, 67, 32, 26, 71, 76};

node \*root = new node;

root = NULL;

for (i = 0; i < 10; i++)

root = InsertIntoTree(root, a[i]);

up:

cout<<"\nEnter the Element to be searched: ";

cin>>n;

Search(root, n);

cout<<"\n\n\tDo you want to search more...enter choice(y/n)?";

cin>>ch;

if(ch == 'y' || ch == 'Y')

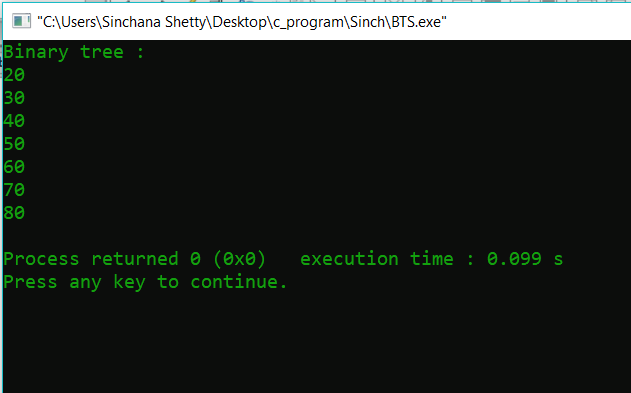
goto up;

return 0;

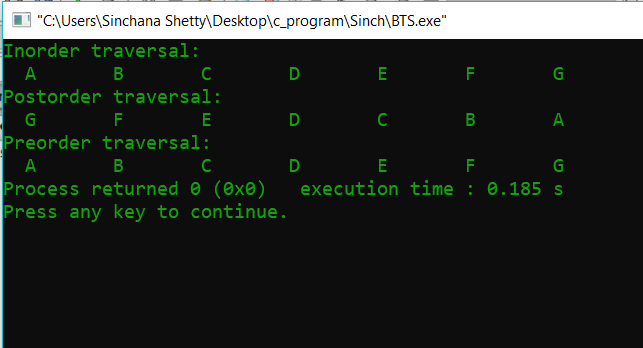
}

**Results:**

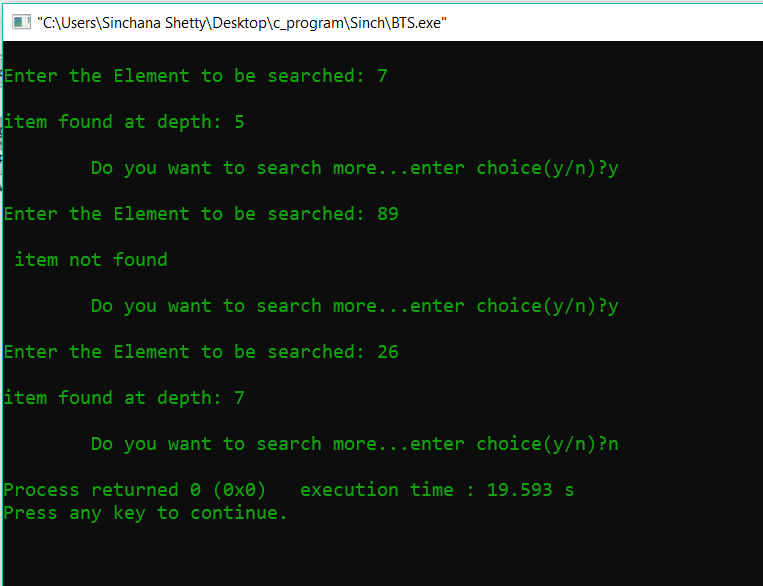
**6a.**



**6b.**



**6c.**



**Analysis(Limitations):**

For inserting element as left child, we have to traverse all elements. Therefore, insertion in binary tree has worst case complexity of O(n).

The current setup works well enough on small amounts of data, but at some point data sets can grow sufficiently large that a single computer cannot hold all of the data at once, or the tree becomes so large that it takes an unreasonable amount of time to complete a traversal.

For searching element , we have to traverse all elements (assuming we do breadth first traversal). Therefore, searching in binary tree has worst case complexity of O(n).