# 21F-9516

# Suresh Kumar

# Bs (SE) 3A

# Assignment 3

## Task 1: -

### Header File: -

Code: -

#pragma once

#include<iostream>

#ifndef BINARYSEARCHTREE\_H

#define BINARYSEARCHTREE\_H

class BST

{

struct Node

{

int data;

Node\* left;

Node\* right;

Node()

{

data = 0;

left = NULL;

right = NULL;

}

Node(int x) :Node()

{

data = x;

}

};

public:

Node\* root;

BST();

~BST();

void Insert(int val);

void DeleteKey(int val);

void SearchKey(int x); //: searches the desired element in the tree recursively

int findMax(); //finds the maximum element in the tree recursively

int findMin(); //: finds the minimum element in the tree recursively

void inorderTraversal(Node\*p);// : prints in - order traversal of the tree

void preorderTraversal(Node\* p); //: prints pre - order traversal of the tree

void postorderTraversal(Node\* p); //: prints post - order traversal of the tree

int treeHeight(Node\*p);// : returns the height of the tree recursively

int treeNodeCount(Node\*p); //: returns the count of nodes in the tree

int treeLeavesCount(Node\*p); //: returns the count of leaves in the tree

int printNodeLevel(Node\*p,int d,int lev);

int getNodeLevel(Node\* n, int num);

};

#endif // !BST\_H

### Definition File: -

#include"binarySearchTree.h"

#include<iostream>

using namespace std;

BST::BST()

{

root = NULL;

}

BST::~BST()

{

}

void BST::DeleteKey(int x)

{

Node\* cur = root;

Node\* temp = cur;

Node temp3;

while (cur)

{

if (x < cur->data)

{

temp = cur;

cur = cur->left;

}

else if (x > cur->data)

{

temp = cur;

cur = cur->right;

}

else

{

break;

}

}

if (cur==NULL)

{

cout << "\nNot Found." << endl;

return;

}

if (cur->left == NULL)

{

temp->left = cur->right;

cur->right = NULL;

delete cur;

cout << "\nNode Is Deleted." << endl;

}

else if (cur->right == NULL)

{

temp->left = cur->left;

cur->left = NULL;

delete cur;

cout << "\nNode Is Deleted." << endl;

}

else

{

Node\* temp1 = cur;

Node\* temp2 = cur;

temp1 = cur->right;

if (temp1->left != NULL)

{

while (temp1->left)

{

temp2 = temp1;

temp1 = temp1->left;

}

}

else

{

temp2 = cur;

}

if (temp1->right == NULL)

{

temp3 = \*temp1;

\*temp1 = \*cur;

\*cur = temp3;

delete temp1;

cout << "\nNode Is Deleted." << endl;

if (temp2 != cur)

{

temp2->left = NULL;

}

else

{

temp2->right = NULL;

}

}

else

{

temp3 = \*temp1;

\*temp1 = \*cur;

\*cur = temp3;

temp2->left = temp1->right;

temp1->right = NULL;

delete temp1;

cout << "\nNode Is Deleted." << endl;

}

}

}

void BST::Insert(int x)

{

Node\* newNode = new Node(x); // Create a new node

if (!root)

root = newNode; // If tree is empty.

else

{ // Tree is not empty

Node\* nodePtr = root; // create a pointer to traverse the tree

while (true)

{

if (x < nodePtr->data)

{ // Left subtree

if (nodePtr->left != NULL)

{

nodePtr = nodePtr->left;

}

else

{

nodePtr->left = newNode;

return;

}

}

else if (x > nodePtr->data)

{ // Right subtree

if (nodePtr->right != NULL)

{

nodePtr = nodePtr->right;

}

else

{

nodePtr->right = newNode;

return;

}

}

else

{

cout << "\nDuplicate value found in tree.\n";

break;

}

}

}

}

void BST::SearchKey(int x)

{

Node\* ptr = root;

bool flag = false;

while (ptr != NULL)

{

if (ptr->data == x)

{

flag = true;

break;

}

else if(x<ptr->data)

{

ptr = ptr->left;

}

else

{

ptr = ptr->right;

}

}

if (flag = true)

{

cout << "\nEntered Number Is -> " << x << endl;

cout << "The Number Is Found In Tree -> " << ptr->data << endl;

}

else

{

cout << "\nNumber Is Not Exist In Tree." << endl;

}

}

int BST::findMax()

{

Node\* newnode = root;

int max=0;

while (newnode)

{

if (newnode->data == max)

{

max = newnode->data;

return max; // value is found

}

else if (newnode->data < max)

{

max = newnode->data;

newnode = newnode->left;

}

else

{

max = newnode->data;

newnode = newnode->right;

}

}

return max; // value is found

}

int BST::findMin()

{

Node\* newnode = root;

int min = 0;

while (newnode)

{

if (newnode->data > min)

{

min = newnode->data;

return min; // value is found

}

else if (newnode->data < min)

{

newnode = newnode->left;

min = newnode->data;

}

else

{

newnode = newnode->right;

min = newnode->data;

}

}

return min; // value is found

}

void BST::inorderTraversal(Node\*p)

{

if (p != NULL)

{

inorderTraversal(p->left);

cout << p->data << " ";

inorderTraversal(p->right);

}

}

void BST::preorderTraversal(Node\* p)

{

if (p != NULL)

{

cout << p->data << " ";

inorderTraversal(p->left);

inorderTraversal(p->right);

}

}

void BST::postorderTraversal(Node\* p)

{

if (p != NULL)

{

inorderTraversal(p->left);

inorderTraversal(p->right);

cout << p->data << " ";

}

}

int BST::printNodeLevel(Node\* p, int d, int lev)

{

if (p == NULL)

return 0;

if (p->data == d)

return lev;

int dlev = printNodeLevel(p->left, d, lev + 1);

if (dlev != 0)

return dlev;

dlev = printNodeLevel(p->right, d, lev + 1);

return dlev;

}

int BST::getNodeLevel(Node\* n, int num)

{

return printNodeLevel(n, num, 1);

}

int BST::treeNodeCount(Node\*p)

{

int count = 0;

if (p)

{

count++;

count = count + treeNodeCount(p->left);

count = count + treeNodeCount(p->right);

}

return count;

}

int BST::treeLeavesCount(Node\*p)

{

if (p == NULL)

return 0;

if (p->left == NULL && p->right == NULL)

return 1;

else

return treeLeavesCount(p->left) + treeLeavesCount(p->right);

}

int BST::treeHeight(Node\*p)

{

if (p == NULL)

return -1;

int lh = treeHeight(p->left);

int rh = treeHeight(p->right);

return lh > rh ? lh + 1 : rh + 1;

}

### Main File: -

#include<iostream>

#include"binarySearchTree.h"

using namespace std;

int main()

{

BST obj;

int num,num1;

cout << "Enter Number Of Entry -> ";

cin >> num;

for (int i = 0; i < num; i++)

{

cout << "Enter Value of Node -> ";

cin >> num1;

obj.Insert(num1);

}

cout << "Maximum Value In Tree -> " << obj.findMax() << endl;

cout << "Minimum Value In Tree -> " << obj.findMin() << endl;

if (obj.root == NULL)

cout << "Tree Is Empty." << endl;

else

{

cout << "\nInorder Traversal." << endl;

obj.inorderTraversal(obj.root);

cout << "\nPreorder Traversal." << endl;

obj.preorderTraversal(obj.root);

cout << "\nPostorder Traversal." << endl;

obj.postorderTraversal(obj.root);

}

cout << "\nEnter Value To Search -> ";

cin >> num1;

obj.SearchKey(num1);

cout << "\nHeight Of Tree Is ->" << obj.treeHeight(obj.root) << endl;

cout << "\nTotal Nodes In Tree Is ->" << obj.treeNodeCount(obj.root) << endl;

cout << "\nTotal Leave Nodes In Tree Is ->" << obj.treeLeavesCount(obj.root) << endl;

cout << "Enter Number To Find Level -> ";

cin >> num1;

cout << "\nTotal Leave Nodes In Tree Is ->" << obj.getNodeLevel(obj.root,num1) << endl;

cout << "\nEnter Value To Delete -> ";

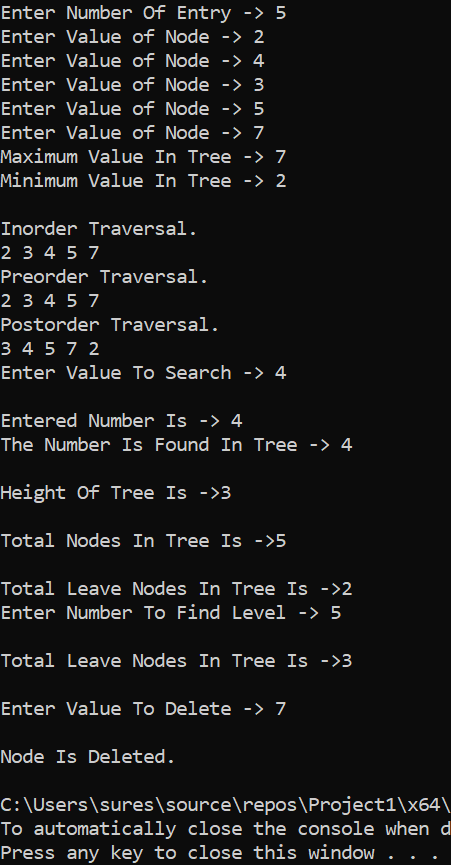
cin >> num1;

obj.DeleteKey(num1);

return 0;

}

Output:-



## Task 2:-

Code:-

#include<iostream>

using namespace std;

class BST

{

struct Node

{

int data;

Node\* left;

Node\* right;

Node()

{

data = 0;

left = NULL;

right = NULL;

}

Node(int x) :Node()

{

data = x;

}

};

public:

Node\* root;

BST()

{

root = NULL;

};

~BST()

{

};

void Insert(int x)

{

Node\* newNode = new Node(x); // Create a new node

if (!root)

root = newNode; // If tree is empty.

else

{ // Tree is not empty

Node\* nodePtr = root; // create a pointer to traverse the tree

while (true)

{

if (x < nodePtr->data)

{ // Left subtree

if (nodePtr->left != NULL)

{

nodePtr = nodePtr->left;

}

else

{

nodePtr->left = newNode;

return;

}

}

else if (x > nodePtr->data)

{ // Right subtree

if (nodePtr->right != NULL)

{

nodePtr = nodePtr->right;

}

else

{

nodePtr->right = newNode;

return;

}

}

else

{

cout << "\nDuplicate value found in tree.\n";

break;

}

}

}

};

void inorderTraversal(Node\* p)// : prints in - order traversal of the tree

{

if (p != NULL)

{

inorderTraversal(p->left);

cout << p->data << " ";

inorderTraversal(p->right);

}

}

void preorderTraversal(Node\* p) //: prints pre - order traversal of the tree

{

if (p != NULL)

{

cout << p->data << " ";

inorderTraversal(p->left);

inorderTraversal(p->right);

}

}

void postorderTraversal(Node\* p) //: prints post - order traversal of the tree

{

if (p != NULL)

{

inorderTraversal(p->left);

inorderTraversal(p->right);

cout << p->data << " ";

}

}

void MirrorTree(Node\* p)

{

if (p == NULL)

return;

else

{

Node\* temp;

MirrorTree(p->left);

MirrorTree(p->right);

temp = p->left;

p->left = p->right;

p->right = temp;

}

}

};

int main()

{

BST obj;

int num, num1;

cout << "Enter Total Numbers Of Nodes -> ";

cin >> num;

cout << "\nInsertion In BST" << endl;

for (int i = 0; i < num; i++)

{

cout << "Enter Node Value -> ";

cin >> num1;

obj.Insert(num1);

}

if (obj.root == NULL)

cout << "Tree Is Empty." << endl;

else

{

cout << "\nInorder Traversal." << endl;

obj.inorderTraversal(obj.root);

cout << "\nPreorder Traversal." << endl;

obj.preorderTraversal(obj.root);

cout << "\nPostorder Traversal." << endl;

obj.postorderTraversal(obj.root);

}

cout << "\nAfter Converting Tree Into Mirror Tree." << endl;

obj.MirrorTree(obj.root);

if (obj.root == NULL)

cout << "Tree Is Empty." << endl;

else

{

cout << "\nInorder Traversal." << endl;

obj.inorderTraversal(obj.root);

cout << "\nPreorder Traversal." << endl;

obj.preorderTraversal(obj.root);

cout << "\nPostorder Traversal." << endl;

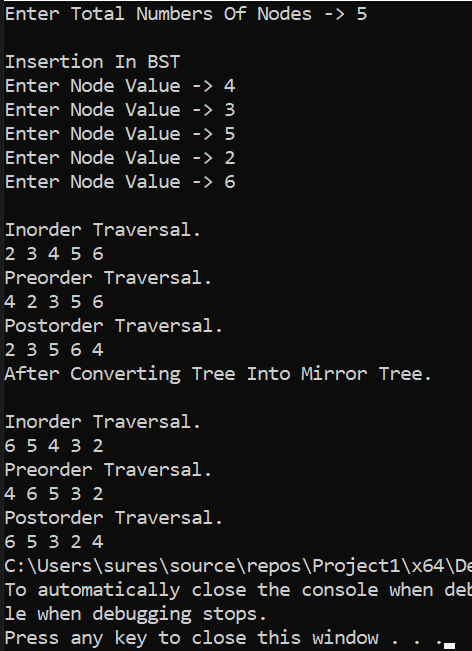
obj.postorderTraversal(obj.root);

}

return 0;

}

Output:-



## Task 3:-

Code:-

#include<iostream>

using namespace std;

class BST

{

struct Node

{

int data;

Node\* left;

Node\* right;

Node()

{

data = 0;

left = NULL;

right = NULL;

}

Node(int x) :Node()

{

data = x;

}

};

public:

Node\* root;

BST()

{

root = NULL;

};

~BST()

{

};

void Insert(int x)

{

Node\* newNode = new Node(x); // Create a new node

if (!root)

root = newNode; // If tree is empty.

else

{ // Tree is not empty

Node\* nodePtr = root; // create a pointer to traverse the tree

while (true)

{

if (x < nodePtr->data)

{ // Left subtree

if (nodePtr->left != NULL)

{

nodePtr = nodePtr->left;

}

else

{

nodePtr->left = newNode;

return;

}

}

else if (x > nodePtr->data)

{ // Right subtree

if (nodePtr->right != NULL)

{

nodePtr = nodePtr->right;

}

else

{

nodePtr->right = newNode;

return;

}

}

else

{

cout << "\nDuplicate value found in tree.\n";

break;

}

}

}

};

void inorderTraversal(Node\* p)// : prints in - order traversal of the tree

{

if (p != NULL)

{

inorderTraversal(p->left);

cout << p->data << " ";

inorderTraversal(p->right);

}

}

void preorderTraversal(Node\* p) //: prints pre - order traversal of the tree

{

if (p != NULL)

{

cout << p->data << " ";

inorderTraversal(p->left);

inorderTraversal(p->right);

}

}

void postorderTraversal(Node\* p) //: prints post - order traversal of the tree

{

if (p != NULL)

{

inorderTraversal(p->left);

inorderTraversal(p->right);

cout << p->data << " ";

}

}

};

void Sort(int a[], int n)

{

int min, temp;

for (int i = 0; i < n - 1; i++)

{

min = i;

for (int j = i + 1; j < n; j++)

if (a[j] < a[min])

min = j;

temp = a[i];

a[i] = a[min];

a[min] = temp;

}

}

int main()

{

BST obj;

int arr[10];

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

{

cout << "Enter Array Value -> ";

cin >> arr[i];

}

cout << "\nDisplay." << endl;

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

{

cout<< arr[i]<<" ";

}

Sort(arr, 10);

cout << "\nAfter Srote Array." << endl;

cout << "\nDisplay." << endl;

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

{

cout<< arr[i] << " ";

}

cout << "\nInsertion In BST" << endl;

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

{

obj.Insert(arr[i]);

}

if (obj.root == NULL)

cout << "Tree Is Empty." << endl;

else

{

cout << "\nInorder Traversal." << endl;

obj.inorderTraversal(obj.root);

cout << "\nPreorder Traversal." << endl;

obj.preorderTraversal(obj.root);

cout << "\nPostorder Traversal." << endl;

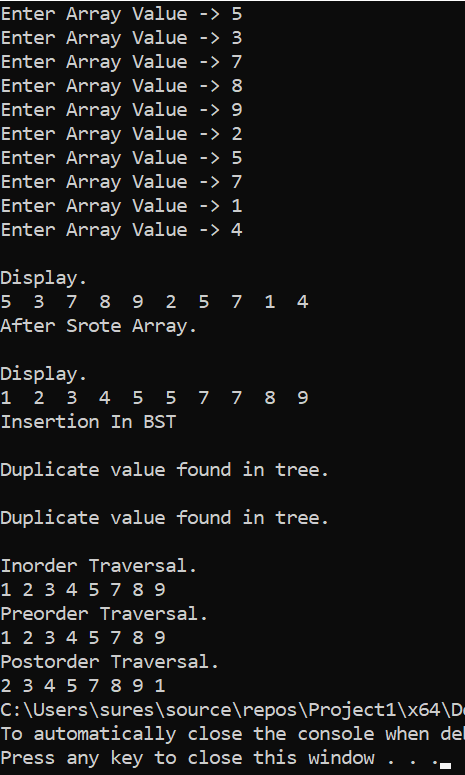
obj.postorderTraversal(obj.root);

}

return 0;

}

Output:-



## Task 4(A):-

Code:-

#include<iostream>

using namespace std;

class BST

{

struct Node

{

int data;

Node\* left;

Node\* right;

Node()

{

data = 0;

left = NULL;

right = NULL;

}

Node(int x) :Node()

{

data = x;

}

};

public:

Node\* root;

BST()

{

root = NULL;

};

~BST()

{

};

void Insert(int x)

{

Node\* newNode = new Node(x); // Create a new node

if (!root)

root = newNode; // If tree is empty.

else

{ // Tree is not empty

Node\* nodePtr = root; // create a pointer to traverse the tree

while (true)

{

if (x < nodePtr->data)

{ // Left subtree

if (nodePtr->left != NULL)

{

nodePtr = nodePtr->left;

}

else

{

nodePtr->left = newNode;

return;

}

}

else if (x > nodePtr->data)

{ // Right subtree

if (nodePtr->right != NULL)

{

nodePtr = nodePtr->right;

}

else

{

nodePtr->right = newNode;

return;

}

}

else

{

cout << "\nDuplicate value found in tree.\n";

break;

}

}

}

};

void inorderTraversal(Node\* p)// : prints in - order traversal of the tree

{

if (p != NULL)

{

inorderTraversal(p->left);

cout << p->data << " ";

inorderTraversal(p->right);

}

}

void preorderTraversal(Node\* p) //: prints pre - order traversal of the tree

{

if (p != NULL)

{

cout << p->data << " ";

inorderTraversal(p->left);

inorderTraversal(p->right);

}

}

void postorderTraversal(Node\* p) //: prints post - order traversal of the tree

{

if (p != NULL)

{

inorderTraversal(p->left);

inorderTraversal(p->right);

cout << p->data << " ";

}

}

int SumOfTree(Node\* p)

{

if (p == NULL)

return 0;

else

return p->data + SumOfTree(p->left) + SumOfTree(p->right);

}

};

int main()

{

BST obj;

int num, num1;

cout << "Enter Total Numbers Of Nodes -> ";

cin >> num;

cout << "\nInsertion In BST" << endl;

for (int i = 0; i < num; i++)

{

cout << "Enter Node Value -> ";

cin >> num1;

obj.Insert(num1);

}

if (obj.root == NULL)

cout << "Tree Is Empty." << endl;

else

{

cout << "\nInorder Traversal." << endl;

obj.inorderTraversal(obj.root);

cout << "\nPreorder Traversal." << endl;

obj.preorderTraversal(obj.root);

cout << "\nPostorder Traversal." << endl;

obj.postorderTraversal(obj.root);

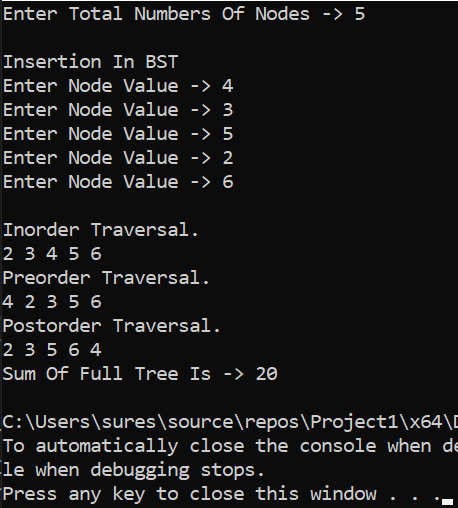
}

cout << "\nSum Of Full Tree Is -> " << obj.SumOfTree(obj.root) << endl;

return 0;

}

Output:-



## Task 5(B):-

Code:-

#include<iostream>

using namespace std;

class BST

{

struct Node

{

int data;

Node\* left;

Node\* right;

Node()

{

data = 0;

left = NULL;

right = NULL;

}

Node(int x) :Node()

{

data = x;

}

};

int sum = 0;

public:

Node\* root;

BST()

{

root = NULL;

};

~BST()

{

};

void Insert(int x)

{

Node\* newNode = new Node(x); // Create a new node

if (!root)

root = newNode; // If tree is empty.

else

{ // Tree is not empty

Node\* nodePtr = root; // create a pointer to traverse the tree

while (true)

{

if (x < nodePtr->data)

{ // Left subtree

if (nodePtr->left != NULL)

{

nodePtr = nodePtr->left;

}

else

{

nodePtr->left = newNode;

return;

}

}

else if (x > nodePtr->data)

{ // Right subtree

if (nodePtr->right != NULL)

{

nodePtr = nodePtr->right;

}

else

{

nodePtr->right = newNode;

return;

}

}

else

{

cout << "\nDuplicate value found in tree.\n";

break;

}

}

}

};

void inorderTraversal(Node\* p)// : prints in - order traversal of the tree

{

if (p != NULL)

{

inorderTraversal(p->left);

cout << p->data << " ";

inorderTraversal(p->right);

}

}

void preorderTraversal(Node\* p) //: prints pre - order traversal of the tree

{

if (p != NULL)

{

cout << p->data << " ";

inorderTraversal(p->left);

inorderTraversal(p->right);

}

}

void postorderTraversal(Node\* p) //: prints post - order traversal of the tree

{

if (p != NULL)

{

inorderTraversal(p->left);

inorderTraversal(p->right);

cout << p->data << " ";

}

}

Node\* ConvertBST(Node\* p)

{

if (p != NULL)

{

ConvertBST(p->right);

sum += p->data;

p->data = sum;

ConvertBST(p->left);

}

return p;

}

};

int main()

{

BST obj;

int num, num1;

cout << "Enter Total Numbers Of Nodes -> ";

cin >> num;

cout << "\nInsertion In BST" << endl;

for (int i = 0; i < num; i++)

{

cout << "Enter Node Value -> ";

cin >> num1;

obj.Insert(num1);

}

if (obj.root == NULL)

cout << "Tree Is Empty." << endl;

else

{

cout << "\nInorder Traversal." << endl;

obj.inorderTraversal(obj.root);

cout << "\nPreorder Traversal." << endl;

obj.preorderTraversal(obj.root);

cout << "\nPostorder Traversal." << endl;

obj.postorderTraversal(obj.root);

}

cout << "\nAfter Converting BST." << endl;

if (obj.root == NULL)

cout << "Tree Is Empty." << endl;

else

{

cout << "\nInorder Traversal." << endl;

obj.inorderTraversal(obj.ConvertBST(obj.root));

cout << "\nPreorder Traversal." << endl;

obj.preorderTraversal(obj.ConvertBST(obj.root));

cout << "\nPostorder Traversal." << endl;

obj.postorderTraversal(obj.ConvertBST(obj.root));

}

return 0;

}

Output:-

