**9 January 2020**

Fatima Seemab

291310  BSCS-9B

Assignment 3

**CODE**

class Node {

public:

int data;

Node\* leftchild;

Node\* rightchild;

};

**TASK 1:**

**INSERT**

void Insertion(int value)

{//To insert values in tree

Node\* nn = new Node();

//New node is created and assigned a value

nn->data = value;

if (isEmpty())

{//If tree is empty newnode is made as root

cout << "insertion of " << value<<endl;

root = nn;

return;

}

else {

//cout << "insertion of " << value<<endl;

//otherwise loc and ploc finds for the logical position and insert it

ploc = NULL;

loc = root;

while (loc != NULL)

{

if (loc->data>value)

{

ploc = loc;

loc = loc->leftchild;

}

else

{

ploc = loc;

loc = loc->rightchild;

}

}

if (ploc->data>value)

{

cout << "left insertion" << endl;

ploc->leftchild = nn;

if (!isbalanced(root))

{

cout << "node disturbs avial condition";

if (leftheavy)

{

balanceFactor(root->leftchild);

if (leftheavy)

{

cout << "leftleftheavy";

RotateforLeft(root);

}

else

{

cout << "leftrightheavy";

RotateforLeftRight(root);

}

}

else

{

balanceFactor(root->rightchild);

if (!leftheavy)

{

cout << "rightrightheavy";

RotateforRight(root);

}

else

{

cout << "rightleftheavy";

RotateforRightLeft(root);

}

}

}

else { cout << endl<<"Tree is balanced"; }

}

else {

cout << "rightinsertion";

ploc->rightchild = nn;

cout <<"ploc->rightchild"<<ploc->rightchild->data<<endl;

if (!isbalanced(root))

{

cout << "node disturbs avial condition";

if (leftheavy)

{

balanceFactor(root->leftchild);

if (leftheavy)

{

cout << "leftleftheavy";

RotateforLeft(root);

}

else

{

cout << "leftrightheavy";

RotateforLeftRight(root);

}

}

else

{

balanceFactor(ploc->rightchild);

if (!leftheavy)

{

cout << "rightrightheavy";

RotateforRight(root);

}

else

{

cout << "rightleftheavy";

RotateforRightLeft(root);

}

}

}

else { cout << endl<<"Tree is balanced"<< endl; }

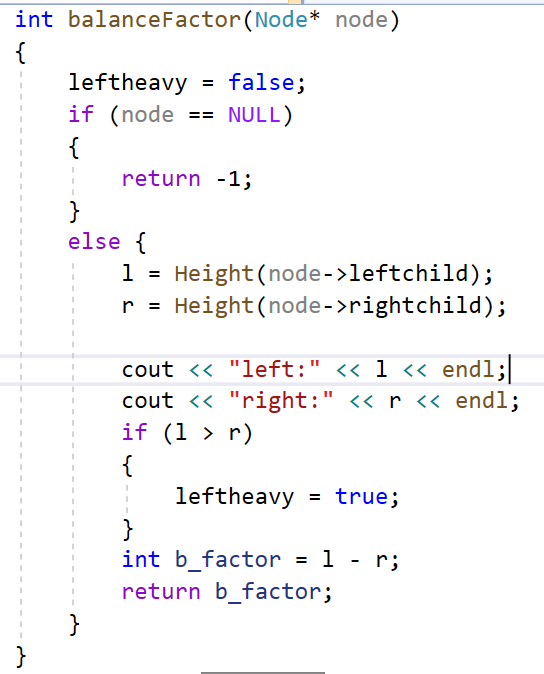
}

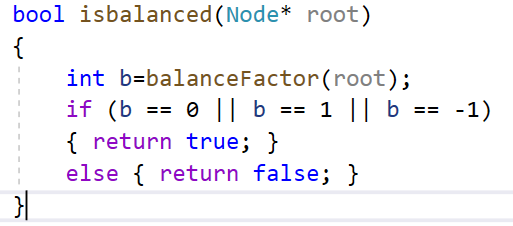
}

}

**TASK 2:**

**Balance Factor:**

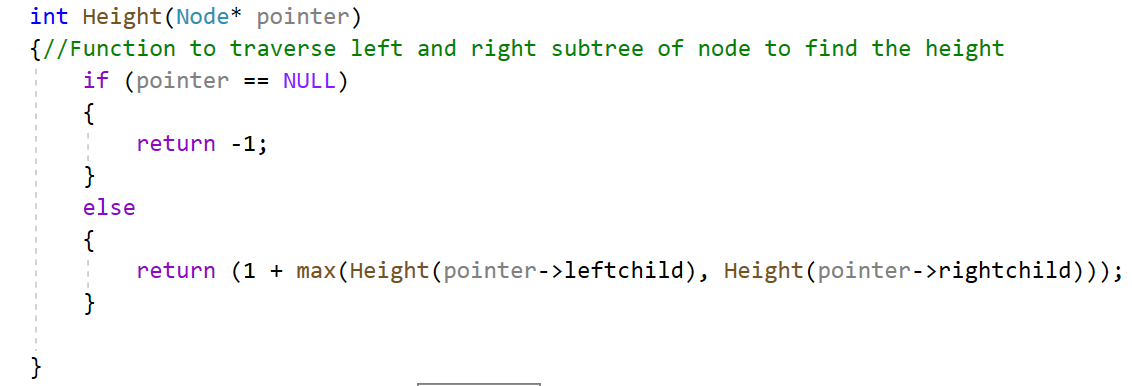




**TASK 4 :**

**CALCULATE HEIGHT**

**CODE:**



**OUTPUT:**



**Task 3 Delete**

void DeleteSubTree(int pointer)

{

bool right = NULL;

Search(pointer);

if (loc != NULL)

{//Case1: Delete Leaf Node

if (loc->leftchild == NULL && loc->rightchild == NULL)

{//leaf is also the root means tree has only one node

if (ploc == NULL)

{

root = NULL;

}

//leaf is right child of parent

else {

if (ploc->rightchild == loc)

{

ploc->rightchild = NULL;

right = true;

}

//leaf is leftchild of parent

else {

ploc->leftchild = NULL;

right = false;

}

}

//delete loc;

}

//Case2: Delete Node with only left subtree

if (loc->leftchild != NULL && loc->rightchild == NULL)

{

if (ploc == NULL)

{

root = loc->leftchild;

}

else

{

if (ploc->rightchild == loc)

{

ploc->rightchild = loc->leftchild;

}

if (ploc->leftchild == loc)

{

ploc->leftchild = loc->leftchild;

}

}

}

//Case2: Delete Node with only right subtree

if (loc->rightchild != NULL && loc->leftchild == NULL)

{

if (ploc == NULL)

{

root = loc->rightchild;

}

else

{

if (ploc->rightchild == loc)

{

ploc->rightchild = loc->rightchild;

}

if (ploc->leftchild == loc)

{

ploc->leftchild = loc->leftchild;

}

}

}

//Case4: Delete Node with both subtree

else

{

Node\* ptemp = NULL;

Node\* temp = root;

while (temp != NULL)

{

ptemp = temp;

temp = temp->rightchild;

}

if (ploc->leftchild==loc)

{

ploc->leftchild = ptemp;

ploc->leftchild->rightchild=loc->rightchild;

}

if (ploc->rightchild == loc)

{

ploc->rightchild = ptemp;

ploc->rightchild->leftchild=loc->leftchild;

}

}

delete loc;

loc = root;

SearchUnbalanced(ploc);

}

else { cout << "Node is not in the tree"; }

}

**SearchUnbalanced**

void SearchUnbalanced(Node\* node)

{

Node\* temp;

if (loc == NULL)

{

return;

}

if (!isbalanced(loc))

{

if (leftheavy)

{

balanceFactor(loc->leftchild);

if (leftheavy)

{

cout << "leftleftheavy";

RotateforLeft(loc);

}

else

{

cout << "leftrightheavy";

RotateforLeftRight(loc);

}

}

else

{

balanceFactor(loc->leftchild);

if (leftheavy)

{

cout << "rightleftheavy";

RotateforRightLeft(loc);

}

else

{

cout << "rightrightheavy";

cout << loc->data<<endl;

RotateforRight(loc);

}

}

return;

}

else

{

PreOrderTraversal(loc->leftchild);

PreOrderTraversal(loc->rightchild);

}

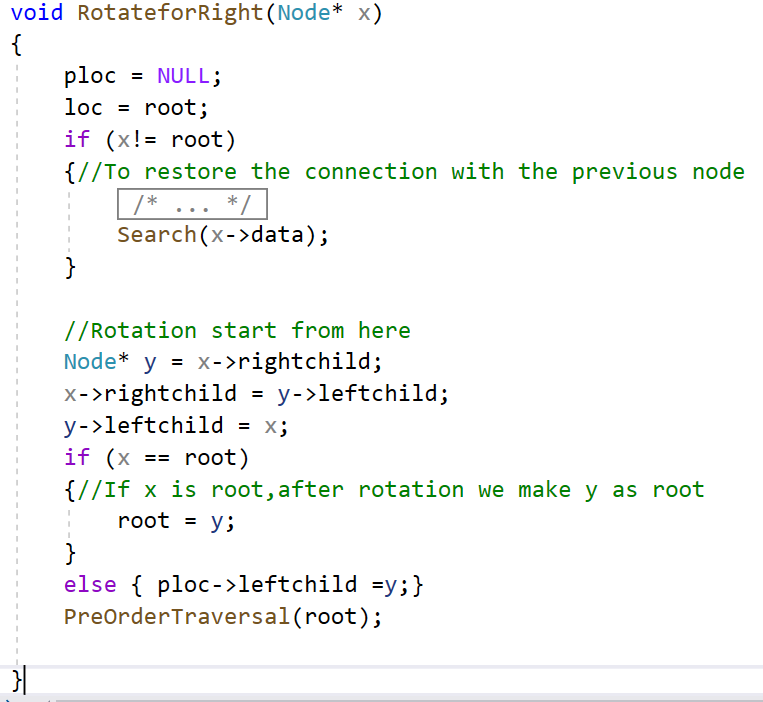
}

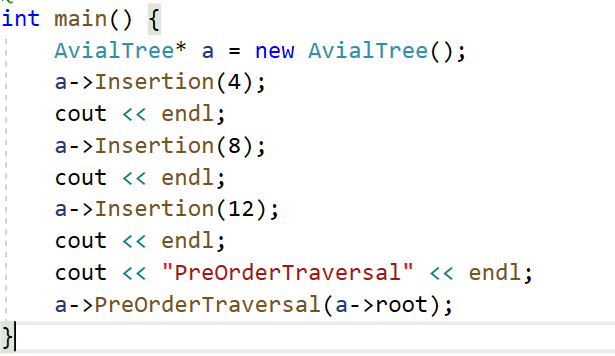
**Task 4 Rotation Left:**

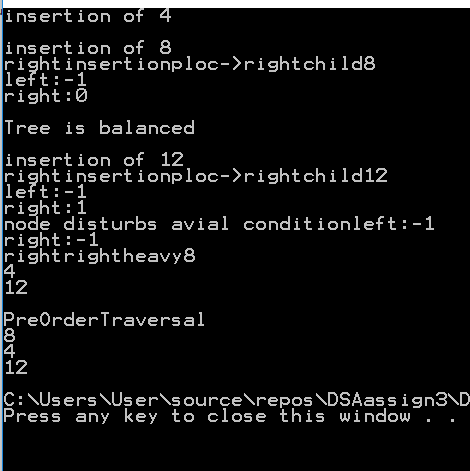
**ASSUMPTION/NOTE:**

Sir, for my own understanding I have named rotation right as rotation for left and rotation left as rotation for right. Because I got super confused in

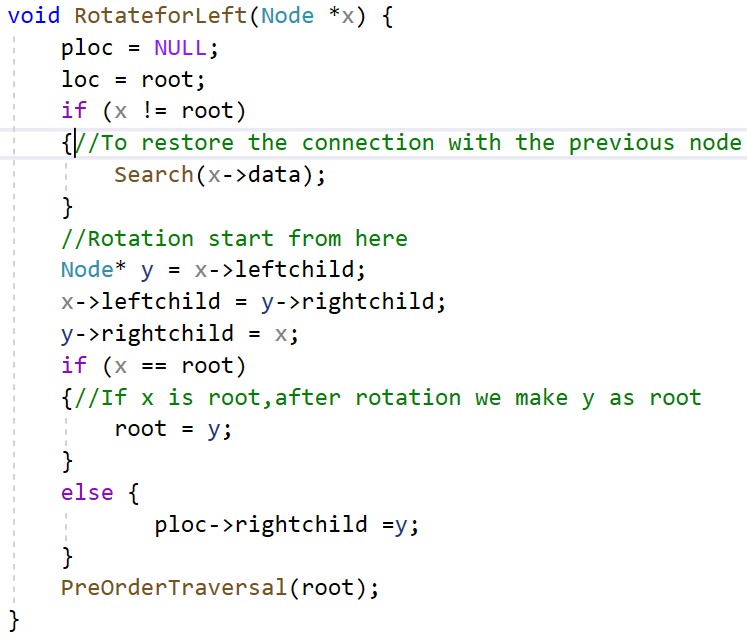
names like rotation right for rotating left nodes.

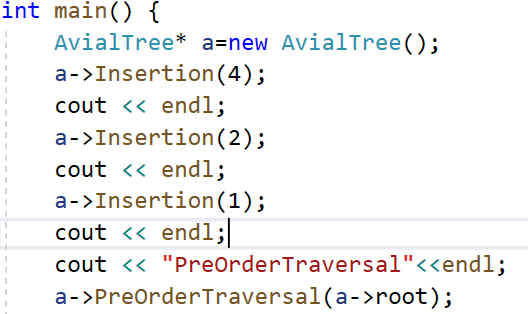


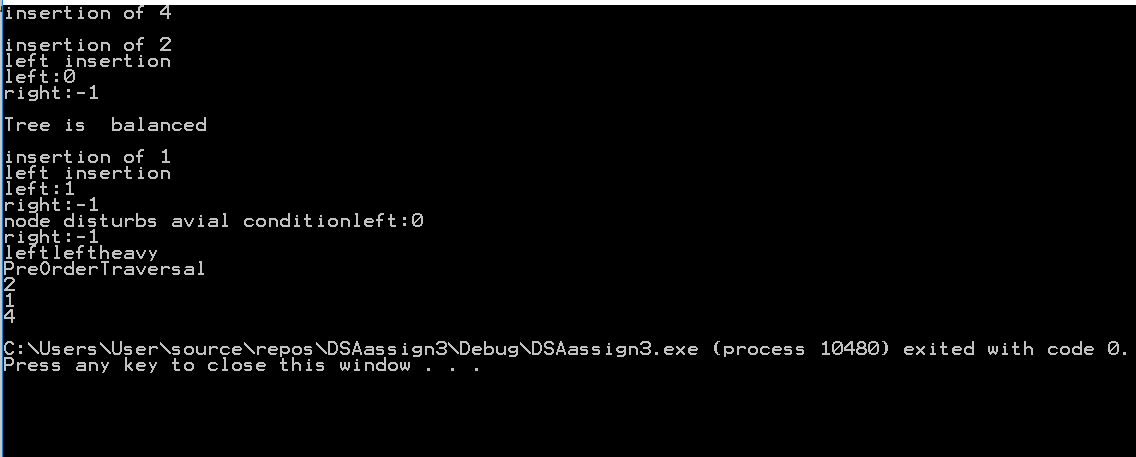




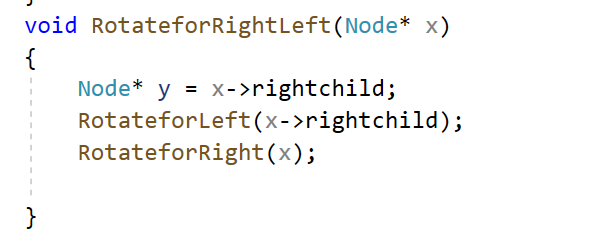
**Task 5 Rotation Right:**

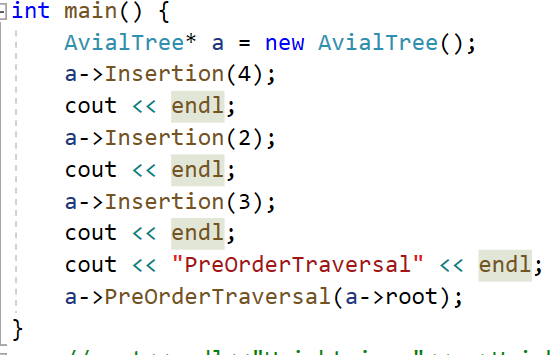


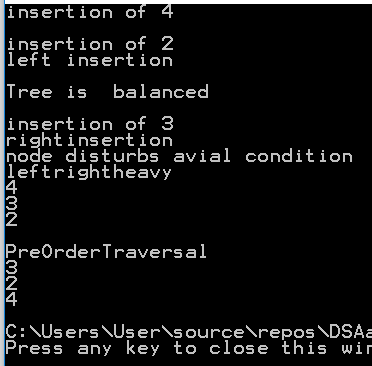




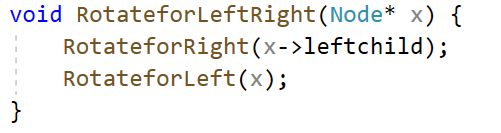
**Task7 Rotation LeftRight**

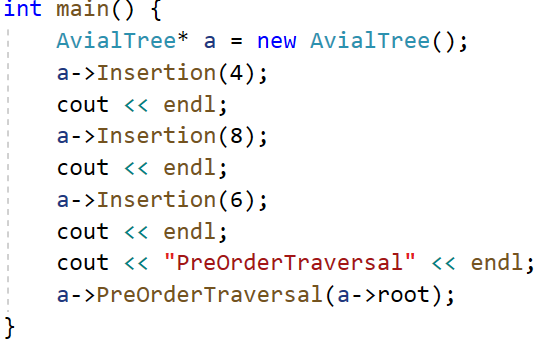


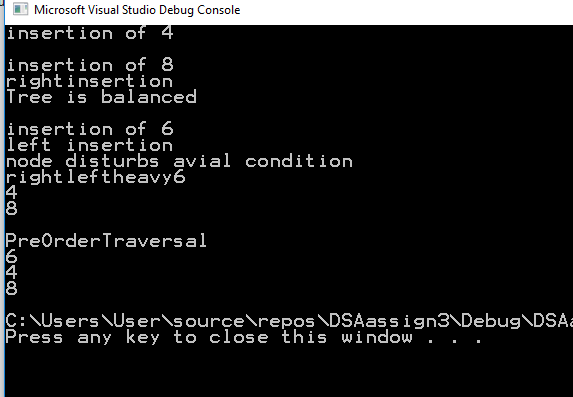




**Task8 Rotation RightLeft**







**Part B**

**Descriprion**

I have designed a simple management system for storing student data using avl tree.

Because of AVL tree, the searching of student in system time complexity is log2n which is very small as compared to arrays, link list, simple trees as well.

**CODE**

#include <iostream>

using namespace std;

class Node {

public:

//Two variables are declared in class having ID and name

int ID = 0;

string name;

public:

//method to return name

string getname()

{

return name;

}

Node\* leftchild;

Node\* rightchild;

};

class AvialTree

{

public:

Node\* loc = NULL;

Node\* ploc = NULL;

Node\* root;

bool leftheavy;

int l;

int r;

int Height(Node\* pointer)

{//Function to traverse left and right subtree of node to find the height

if (pointer == NULL)

{

return -1;

}

else

{

return (1 + max(Height(pointer->leftchild), Height(pointer->rightchild)));

}

}

int balanceFactor(Node\* node)

{

leftheavy = false;

if (node == NULL)

{

return -1;

}

else {

l = Height(node->leftchild);

r = Height(node->rightchild);

//cout << "left:" << l << endl;

//cout << "right:" << r << endl;

if (l > r)

{

leftheavy = true;

}

int b\_factor = l - r;

return b\_factor;

}

}

bool isEmpty()

{//To check wether tree is empty or not

return(root == NULL);

}

void Insertion(int value, string name)

{//To insert values in tree

Node\* nn = new Node();

//New node is created and assigned a value

nn->ID = value;

nn->name = name;

if (isEmpty())

{//If tree is empty newnode is made as root

root = nn;

return;

}

else {

//otherwise loc and ploc finds for the logical position and insert it

ploc = NULL;

loc = root;

while (loc != NULL)

{

if (loc->ID > value)

{

ploc = loc;

loc = loc->leftchild;

}

else

{

ploc = loc;

loc = loc->rightchild;

}

}

if (ploc->ID > value)

{

ploc->leftchild = nn;

if (!isbalanced(root))

{

if (leftheavy)

{

balanceFactor(root->leftchild);

if (leftheavy)

{

RotateforLeft(root);

}

else

{

RotateforLeftRight(root);

}

}

else

{

balanceFactor(root->rightchild);

if (!leftheavy)

{

RotateforRight(root);

}

else

{

RotateforRightLeft(root);

}

}

}

else { cout << endl; }

}

else {

ploc->rightchild = nn;

if (!isbalanced(root))

{

if (leftheavy)

{

balanceFactor(root->leftchild);

if (leftheavy)

{

RotateforLeft(root);

}

else

{

RotateforLeftRight(root);

}

}

else

{

balanceFactor(ploc->rightchild);

if (!leftheavy)

{

RotateforRight(root);

}

else

{

RotateforRightLeft(root);

}

}

}

else { cout << endl; }

}

}

}

void DeleteSubTree(int pointer)

{

bool right = NULL;

Search(pointer);

if (loc != NULL)

{//Case1: Delete Leaf Node

if (loc->leftchild == NULL && loc->rightchild == NULL)

{//leaf is also the root means tree has only one node

if (ploc == NULL)

{

root = NULL;

}

//leaf is right child of parent

else {

if (ploc->rightchild == loc)

{

ploc->rightchild = NULL;

right = true;

}

//leaf is leftchild of parent

else {

ploc->leftchild = NULL;

right = false;

}

}

//delete loc;

}

//Case2: Delete Node with only left subtree

if (loc->leftchild != NULL && loc->rightchild == NULL)

{

if (ploc == NULL)

{

root = loc->leftchild;

}

else

{

if (ploc->rightchild == loc)

{

ploc->rightchild = loc->leftchild;

}

if (ploc->leftchild == loc)

{

ploc->leftchild = loc->leftchild;

}

}

}

//Case2: Delete Node with only right subtree

if (loc->rightchild != NULL && loc->leftchild == NULL)

{

if (ploc == NULL)

{

root = loc->rightchild;

}

else

{

if (ploc->rightchild == loc)

{

ploc->rightchild = loc->rightchild;

}

if (ploc->leftchild == loc)

{

ploc->leftchild = loc->leftchild;

}

}

}

//Case4: Delete Node with both subtree

else

{

Node\* ptemp = NULL;

Node\* temp = root;

while (temp != NULL)

{

ptemp = temp;

temp = temp->rightchild;

}

if (ploc->leftchild == loc)

{

ploc->leftchild = ptemp;

ploc->leftchild->rightchild = loc->rightchild;

}

if (ploc->rightchild == loc)

{

ploc->rightchild = ptemp;

ploc->rightchild->leftchild = loc->leftchild;

}

}

delete loc;

loc = root;

SearchUnbalanced(ploc);

}

else { cout << "Node is not in the tree"; }

}

void SearchUnbalanced(Node\* node)

{

Node\* temp;

if (loc == NULL)

{

return;

}

if (!isbalanced(loc))

{

if (leftheavy)

{

balanceFactor(loc->leftchild);

if (leftheavy)

{

RotateforLeft(loc);

}

else

{

RotateforLeftRight(loc);

}

}

else

{

balanceFactor(loc->leftchild);

if (leftheavy)

{

RotateforRightLeft(loc);

}

else

{

cout << loc->ID << endl;

RotateforRight(loc);

}

}

return;

}

else

{

PreOrderTraversal(loc->leftchild);

PreOrderTraversal(loc->rightchild);

}

}

void Search(int value)

{//To search for a value in tree

//As a result loc and ploc will be updated

ploc = NULL;

loc = root;

//seraches until value is found or tree ends

while (loc != NULL && loc->ID != value)

{//ploc points to the previous node of loc

ploc = loc;

if (loc->ID > value)

{ //In BST,left side has small values and right side has larger

loc = loc->leftchild;

}

else { loc = loc->rightchild; }

}

}

bool isbalanced(Node\* root)

{

int b = balanceFactor(root);

if (b == 0 || b == 1 || b == -1)

{

return true;

}

else { return false; }

}

void RotateforRight(Node\* x)

{

ploc = NULL;

loc = root;

if (x != root)

{

Search(x->ID);

}

//Rotation start from here

Node\* y = x->rightchild;

x->rightchild = y->leftchild;

y->leftchild = x;

if (x == root)

{//If x is root,after rotation we make y as root

root = y;

}

else { ploc->leftchild = y; }

}

void RotateforLeft(Node\* x) {

ploc = NULL;

loc = root;

if (x != root)

{//To restore the connection with the previous node

Search(x->ID);

}

//Rotation start from here

Node\* y = x->leftchild;

x->leftchild = y->rightchild;

y->rightchild = x;

if (x == root)

{//If x is root,after rotation we make y as root

root = y;

}

else {

ploc->rightchild = y;

}

}

void RotateforRightLeft(Node\* x)

{

Node\* y = x->rightchild;

RotateforLeft(x->rightchild);

RotateforRight(x);

}

void RotateforLeftRight(Node\* x) {

RotateforRight(x->leftchild);

RotateforLeft(x);

}

void PreOrderTraversal(Node\* pointer)

{ //In this function tree is traversed first its root than left and right subtree

Node\* temp;

if (pointer != NULL)

{

cout << pointer->ID << endl;

cout << pointer->getname() << endl;

PreOrderTraversal(pointer->leftchild);

PreOrderTraversal(pointer->rightchild);

}

}

};

int main()

{

AvialTree\* a = new AvialTree();

int option = 0;

string name;

int ID = 0;

cout << "------------------------------------------------------------------------------" << endl;

cout << "Welcome to students registration database" << endl;

while (true)

{

cout << "what u need to check" << endl << "1 for registering student and assigning cms id " <<

endl << "2 for deleting student from record" << endl << "3 for seeing all student names and ID" <<endl<<"4 for exit student cms database"<< endl;

//input from user in option variable

cin >> option;

//option passed to switch to select whether register partient,serve patient,cancel all

switch (option)

{

case 1:

{//if queue is not full

cout << "please enter the name of student" << endl;

cin >> name;

cout << "please enter the ID of student" << endl;

cin >> ID;

a->Insertion(ID, name);

//break statement to break from cases

break;

}

case 2:

{//case 2 for serving the patient

cout << "please enter the ID of student who has left" << endl;

cin >> ID;

a->DeleteSubTree(ID);

break;

}

case 3:

{ //case 3 for checking doctor can go or not

a->PreOrderTraversal(a->root);

break;

}

case 4://case 4 show all the records for todays patient

{return 0;

break;

}

}

cout << "------------------------------------------------------------------------------" << endl;

}

}

