Task 1 2 3

#include<iostream>

using namespace std;

class Node {

public:

int data;

int height;

Node\* left;

Node\* right;

Node(int data) {

this->data = data;

this->height = 1;

this->left = NULL;

this->right = NULL;

}

};

class AVLTree{

public:

Node\* root;

AVLTree(){

root = NULL;

}

int getHeight(Node\* n) {

if (n == NULL) {

return 0;

}

return n->height;

}

Node\* rightRotation(Node\* y) {

Node\* x = y->left;

Node\* t2 = x->right;

x->right = y;

y->left = t2;

y->height = max(getHeight(y->right), getHeight(y->left))+1;

x->height = max(getHeight(x->right), getHeight(x->left))+1;

return x;

}

Node\* leftRotation(Node\* x) {

Node\* y = x->right;

Node\* t2 = y->left;

y->left = x;

x->right = t2;

y->height = max(getHeight(y->right), getHeight(y->left))+1;

x->height = max(getHeight(x->right), getHeight(x->left))+1;

return y;

}

Node\* insert(Node\* node, int key) {

if(node==NULL){

return new Node(key);

}

if(key<node->data) {

node->left= insert(node->left, key);

}

else if(key>node->data) {

node->right = insert(node->right, key);

}

else{

return node;

}

node->height = 1+max(getHeight(node->left), getHeight(node->right));

int bf = getBalanceFactor(node);

// Cases for rotation

if(bf>1&&key<node->left->data) {

return rightRotation(node);

}

if(bf<-1 && key>node->right->data) {

return leftRotation(node);

}

if (bf>1&&key>node->left->data) {

node->left = leftRotation(node->left);

return rightRotation(node);

}

if(bf<-1&&key<node->right->data) {

node->right = rightRotation(node->right);

return leftRotation(node);

}

return node;

}

void insert(int key) {

root=insert(root, key);

}

int getBalanceFactor(Node\* node) {

if (node==NULL) {

return 0;

}

return getHeight(node->left) - getHeight(node->right);

}

Node\* successor(Node\* node) {

Node\* current = node;

while (current->left != NULL) {

current = current->left;

}

return current;

}

Node\* deleteNode(Node\* node, int key) {

if (node == NULL) {

return node;

}

if(key<node->data) {

node->left=deleteNode(node->left, key);

} else if(key>node->data){

node->right= deleteNode(node->right, key);

} else{

if (node->left==NULL) {

Node\* temp = node->right;

delete node;

return temp;

} else if(node->right== NULL){

Node\* temp = node->left;

delete node;

return temp;

}

Node\* temp = successor(node->right);

node->data = temp->data;

node->right = deleteNode(node->right, temp->data);

}

node->height = 1 + max(getHeight(node->left), getHeight(node->right));

int balance = getBalanceFactor(node);

return node;

}

void deleteNode(int key) {

root = deleteNode(root,key);

}

void inOrder(Node\* node) {

if (node==NULL) {

return;

}

inOrder(node->left);

cout<<node->data << " ";

inOrder(node->right);

}

void inOrderTraversal() {

inOrder(root);

}

void preOrder(Node\* node) {

if (node==NULL) {

return;

}

cout<<node->data << " ";

preOrder(node->left);

preOrder(node->right);

}

void preOrderTraversal() {

preOrder(root);

}

void postOrder(Node\* node) {

if (node==NULL) {

return;

}

postOrder(node->left);

postOrder(node->right);

cout<<node->data << " ";

}

void postOrderTraversal() {

postOrder(root);

}

};

int main() {

AVLTree avl;

avl.insert(1);

avl.insert(2);

avl.insert(3);

avl.insert(4);

avl.insert(5);

avl.insert(5);

avl.insert(6);

avl.insert(7);

cout<<"PreOrder"<<endl;

avl.preOrderTraversal();

cout<<"\nInOrder"<<endl;

avl.inOrderTraversal();

cout<<"\nPostOrder"<<endl;

avl.postOrderTraversal();

avl.deleteNode(3);

avl.deleteNode(5);

avl.deleteNode(7);

cout<<"\n\nAfter deletion {3 5 7}\n"<<endl;

cout<<"\nPreOrder"<<endl;

avl.preOrderTraversal();

cout<<"\nInOrder"<<endl;

avl.inOrderTraversal();

cout<<"\nPostOrder"<<endl;

avl.postOrderTraversal();

return 0;

}

