

#pragma once

#include <iostream>

using namespace std;

class Node

{

public:

int key;

Node\* left;

Node\* right;

int height;

Node()

{

key = -1;

left = nullptr;

right = nullptr;

height = 1;

}

Node(int value)

{

key = value;

left = nullptr;

right = nullptr;

height = 1;

}

};

class AVLTree

{

public:

Node\* root;

AVLTree()

{

root = nullptr;

}

int getHeight(Node\* node)

{

if (node == NULL)

return 0;

return node->height;

}

int calc\_BF(Node\* node)

{

if (node == nullptr)

return 0;

int bf = getHeight(node->left) - getHeight(node->right);

return bf;

}

void updateHeight(Node\* node)

{

if (node == NULL)

return;

int greater = std::max(getHeight(node->left), getHeight(node->right));

node->height = 1 + greater;

}

Node\* rightRotate(Node\* ptr)

{

Node\* l\_ptr = ptr->left;

Node\* layer2 = l\_ptr->right;

l\_ptr->right = ptr;

ptr->left = layer2;

updateHeight(ptr);

updateHeight(l\_ptr);

return l\_ptr;

}

Node\* leftRotate(Node\* ptr)

{

Node\* r\_ptr = ptr->right;

Node\* layer2 = r\_ptr->left;

r\_ptr->left = ptr;

ptr->right = layer2;

updateHeight(ptr);

updateHeight(r\_ptr);

return r\_ptr;

}

Node\* balance(Node\* node) {

int balanceFactor = calc\_BF(node);

if (balanceFactor > 1 && calc\_BF(node->left) >= 0)

return rightRotate(node);

if (balanceFactor > 1 && calc\_BF(node->left) < 0) {

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

return rightRotate(node);

}

if (balanceFactor < -1 && calc\_BF(node->right) <= 0)

return leftRotate(node);

if (balanceFactor < -1 && calc\_BF(node->right) > 0) {

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

return leftRotate(node);

}

return node;

}

Node\* findMinNode(Node\* node)

{

while (node->left != nullptr)

{

node = node->left;

}

return node;

}

Node\* findMaxNode(Node\* node)

{

while (node->right != nullptr)

{

node = node->right;

}

return node;

}

bool isAVLTree()

{

int minHeight, maxHeight;

return check\_AVL(root, minHeight, maxHeight);

}

bool check\_AVL(Node\* node, int& minHeight, int& maxHeight)

{

if (node == nullptr)

{

minHeight = 0;

maxHeight = 0;

return 1;

}

int left\_min, left\_max;

int right\_min, right\_max;

bool leftIsAVL = check\_AVL(node->left, left\_min, left\_max);

bool rightIsAVL = check\_AVL(node->right, right\_min, right\_max);

minHeight = 1 + std::min(left\_min, right\_min);

maxHeight = 1 + std::max(left\_max, right\_max);

if (leftIsAVL && rightIsAVL && (abs(left\_max - right\_max) <= 1))

return 1;

return 0;

}

void insert(const int& value)

{

root = insertNode(root, value);

}

Node\* insertNode(Node\* node, const int& value)

{

if (node == nullptr)

{

Node\* ptr = new Node;

ptr->key = value;

return ptr;

}

if (value < node->key)

{

node->left = insertNode(node->left, value);

}

else if (value > node->key)

{

node->right = insertNode(node->right, value);

}

else

{

return node;

}

updateHeight(node);

int balance = calc\_BF(node);

if (balance > 1 && value < node->left->key)

{

return rightRotate(node);

}

if (balance < -1 && value > node->right->key)

{

return leftRotate(node);

}

if (balance > 1 && value > node->left->key)

{

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

return rightRotate(node);

}

if (balance < -1 && value < node->right->key)

{

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

return leftRotate(node);

}

return node;

}

void inOrderTraversal(Node\* node)

{

if (node != nullptr)

{

inOrderTraversal(node->left);

std::cout << node->key << " ";

inOrderTraversal(node->right);

}

}

void preOrderTraversal(Node\* node)

{

if (node == NULL)

return;

std::cout << node->key << " ";

preOrderTraversal(node->left);

preOrderTraversal(node->right);

}

void postOrderTraversal(Node\* node)

{

if (node == nullptr)

return;

postOrderTraversal(node->left);

postOrderTraversal(node->right);

cout << node->key << " ";

}

void display()

{

cout << "In-order traversal: ";

inOrderTraversal(root);

cout << "\n";

}

void remove(int x)

{

root = deleteNode(root, x);

}

Node\* deleteNode(Node\* node, const int& value)

{

if (node == nullptr)

return node;

if (value < node->key)

{

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

}

else if (value > node->key)

{

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

}

else

{

if (node->left == nullptr || node->right == nullptr) {

Node\* temp;

if (node->left != NULL)

temp = node->left;

else

temp = node->right;

if (temp == nullptr)

{

temp = node;

node = nullptr;

}

else

{

\*node = \*temp;

}

delete temp;

}

else

{

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

node->key = temp->key;

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

}

}

if (node == nullptr)

return node;

updateHeight(node);

int balance = calc\_BF(node);

if (balance > 1 && calc\_BF(node->left) >= 0)

{

return rightRotate(node);

}

if (balance > 1 && calc\_BF(node->left) < 0)

{

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

return rightRotate(node);

}

if (balance < -1 && calc\_BF(node->right) <= 0)

{

return leftRotate(node);

}

if (balance < -1 && calc\_BF(node->right) > 0)

{

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

return leftRotate(node);

}

return node;

}

Node\* sortedArrayToAVL(int arr[], int start, int end) {

if (start > end)

return nullptr;

int mid = (start + end) / 2;

Node\* node = new Node(arr[mid]);

node->left = sortedArrayToAVL(arr, start, mid - 1);

node->right = sortedArrayToAVL(arr, mid + 1, end);

updateHeight(node);

return node;

}

Node\* joinTrees(Node\* T1, Node\* T2) {

if (T1 == nullptr) return T2;

if (T2 == nullptr) return T1;

int maxInT1 = findMaxNode(T1)->key;

T1 = deleteNode(T1, maxInT1);

Node\* newRoot = new Node(maxInT1);

newRoot->left = T1;

newRoot->right = T2;

updateHeight(newRoot);

return balance(newRoot);

}

void convertBSTToAVL(int arr[], int size) {

root = sortedArrayToAVL(arr, 0, size - 1);

}

void join(AVLTree& T1, AVLTree& T2) {

root = joinTrees(T1.root, T2.root);

}

};

int main() {

AVLTree tree1, tree2;

// Task 1

tree1.insert(10);

tree1.insert(20);

tree1.insert(30);

tree1.insert(40);

tree1.insert(50);

tree1.insert(25);

cout << "Tree1 Inorder Traversal after inserts: ";

tree1.inOrderTraversal(tree1.root);

cout << endl;

tree1.remove(10);

cout << "Tree1 Inorder Traversal after deleting 10: ";

tree1.inOrderTraversal(tree1.root);

cout << endl;

// Task 2

cout << "Is tree1 a valid AVL tree? " << (tree1.isAVLTree() ? "Yes" : "No") << endl;

// Task 3

int bstArray[] = { 1, 2, 3, 4, 5, 6, 7 };

int size = sizeof(bstArray) / sizeof(bstArray[0]);

AVLTree bstToAvlTree;

bstToAvlTree.convertBSTToAVL(bstArray, size);

cout << "BST converted to AVL (inorder traversal): ";

bstToAvlTree.inOrderTraversal(bstToAvlTree.root);

cout << endl;

// Task 4

tree2.insert(60);

tree2.insert(70);

tree2.insert(80);

cout << "Tree2 Inorder Traversal: ";

tree2.inOrderTraversal(tree2.root);

cout << endl;

tree1.join(tree1, tree2);

cout << "Joined Tree Inorder Traversal: ";

tree1.inOrderTraversal(tree1.root);

cout << endl;

return 0;

}