Lab #9

Task #1:

#include <iostream>

#include<algorithm>

#include <string>

using namespace std;

struct Vehicle

{

int id; //attributes

string location;

bool available;

Vehicle(int id, const string& location, bool available) : id(id), location(location), available(available)// using intializer list

{

}

};

struct Node

{ //attributes

Vehicle vehicle;

Node\* left;

Node\* right;

int height;

Node(Vehicle vehicle): vehicle(vehicle), left(nullptr), right(nullptr), height(1) // using intializer list

{}

};

// Class representing AVL tree

class AVLTree

{

public:

AVLTree()

{

root = nullptr;

}

// Insert a new vehicle into the AVL tree

void insert(Vehicle vehicle)

{

root = insertNode(root, vehicle);

}

// Remove a vehicle from the AVL tree

void remove(int id)

{

root = removeNode(root, id);

}

// Search for a vehicle by its identification num

Vehicle\* search(int id)

{

Node\* node = searchNode(root, id);

return node ? &(node->vehicle) : nullptr;// using tertiary operator

}

// Update the location and availability status of a vehicle

void update(int id, const string& location, bool available)

{

Node\* node = searchNode(root, id);

if (node) {

node->vehicle.location = location;

node->vehicle.available = available;

}

}

private:

// Get the height of a node

int getHeight(Node\* node) {

if (node == nullptr)

return 0;

return node->height;

}

// Get the balance factor of a node

int getBalance(Node\* node) {

if (node == nullptr)

return 0;

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

}

// Update the height of a node

void updateHeight(Node\* node)

{

int leftHeight = getHeight(node->left);

int rightHeight = getHeight(node->right);

node->height = 1 + max(leftHeight, rightHeight);

}

// Right rotate a subtree rooted with 'node'

Node\* rotateRight(Node\* node) {

Node\* newRoot = node->left;

node->left = newRoot->right;

newRoot->right = node;

updateHeight(node);

updateHeight(newRoot);

return newRoot;

}

// Left rotate a subtree rooted with 'node'

Node\* rotateLeft(Node\* node)

{

Node\* newRoot = node->right;

node->right = newRoot->left;

newRoot->left = node;

updateHeight(node);

updateHeight(newRoot);

return newRoot;

}

// Balance the AVL tree after insertion or deletion

Node\* balanceTree(Node\* node)

{

updateHeight(node);

int balance = getBalance(node);

// Left Left Case

if (balance > 1 && getBalance(node->left) >= 0)

return rotateRight(node);

// Right Right Case

if (balance < -1 && getBalance(node->right) <= 0)

return rotateLeft(node);

// Left Right Case

if (balance > 1 && getBalance(node->left) < 0) {

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

return rotateRight(node);

}

// Right Left Case

if (balance < -1 && getBalance(node->right) > 0) {

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

return rotateLeft(node);

}

return node;

}

// Insert a vehicle node into the AVL tree

Node\* insertNode(Node\* node, Vehicle vehicle)

{

if (node == nullptr)

return new Node(vehicle);

if (vehicle.id < node->vehicle.id)

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

else if (vehicle.id > node->vehicle.id)

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

else

return node; // Duplicate keys are not allowed

return balanceTree(node);

}

// Find the node with the minimum key value in the AVL tree

Node\* findMinNode(Node\* node) {

if (node == nullptr)

return nullptr;

while (node->left != nullptr)

node = node->left;

return node;

}

// Remove a vehicle node from the AVL tree

Node\* removeNode(Node\* node, int id)

{

if (node == nullptr)

return nullptr;

if (id < node->vehicle.id)

node->left = removeNode(node->left, id);

else if (id > node->vehicle.id)

node->right = removeNode(node->right, id);

else {

// Node to be deleted found

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

Node\* temp = node->left ? node->left : node->right;

if (temp == nullptr) {

temp = node;

node = nullptr;

}

else

\*node = \*temp;

delete temp;

}

else {

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

node->vehicle = temp->vehicle;

node->right = removeNode(node->right, temp->vehicle.id);

}

}

if (node == nullptr)

return nullptr;

return balanceTree(node);

}

// Search for a vehicle node by its identification number

Node\* searchNode(Node\* node, int id) {

if (node == nullptr || node->vehicle.id == id)

return node;

if (id < node->vehicle.id)

return searchNode(node->left, id);

else

return searchNode(node->right, id);

}

// function for printing the AVL tree

void printInOrder(Node\* node) {

if (node == nullptr)

return;

printInOrder(node->left);

cout << "ID: " << node->vehicle.id << ", Location: " << node->vehicle.location << ", Available: "

<< (node->vehicle.available ? "Yes" : "No") << endl;

printInOrder(node->right);

}

public:

// Print the AVL tree (in-order traversal)

void print() {

printInOrder(root);

}

private:

Node\* root;

};

// Helper function to return the maximum of two integers

int max(int a, int b) {

return (a > b) ? a : b;

}

int main()

{

AVLTree avlTree;

int choice;

do {

cout << "AVL Tree Menu:" << endl;

cout << "1-> Insert a new vehicle" << endl;

cout << "2-> Remove a vehicle" << endl;

cout << "3-> Search for a vehicle" << endl;

cout << "4-> Update vehicle information" << endl;

cout << "5-> Print AVL tree" << endl;

cout << "6-> Exit" << endl;

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1: {

int id;

string location;

bool available;

cout << "Enter vehicle ID: ";

cin >> id;

cout << "Enter vehicle location: ";

cin >> location;

cout << "Is vehicle available (1 for Yes, 0 for No): ";

cin >> available;

avlTree.insert(Vehicle(id, location, available));

cout << "Vehicle inserted successfully!" << endl;

break;

}

case 2: {

int id;

cout << "Enter the ID of the vehicle to remove: ";

cin >> id;

avlTree.remove(id);

cout << "Vehicle removed successfully!" << endl;

break;

}

case 3: {

int id;

cout << "Enter the ID of the vehicle to search: ";

cin >> id;

Vehicle\* searchResult = avlTree.search(id);

if (searchResult) {

cout << "Vehicle found:" << endl;

cout << "ID: " << searchResult->id << ", Location: " << searchResult->location

<< ", Available: " << (searchResult->available ? "Yes" : "No") << endl;

}

else

{

cout << "Vehicle not found." << endl;

}

break;

}

case 4: {

int id;

string location;

bool available;

cout << "Enter the ID of the vehicle to update: ";

cin >> id;

cout << "Enter the new location: ";

cin >> location;

cout << "Is the vehicle available (1 for Yes, 0 for No): ";

cin >> available;

avlTree.update(id, location, available);

cout << "Vehicle updated successfully!" << endl;

break;

}

case 5:

{

cout << "AVL Tree:" << endl;

avlTree.print();

break;

}

case 6: {

cout << "Exiting..." << endl;

break;

}

default:

{

cout << "Invalid choice. Please try again." << endl;

break;

}

}

cout << endl;

} while (choice != 6);

system("pause");

return 0;

}

Output:

