**ЛАБОРАТОРНАЯ РАБОТА №2**

**«ИССЛЕДОВАНИЕ НЕЛИНЕЙНЫХ СТРУКТУР ДАННЫХ. AVL-ДЕРЕВЬЯ»**

**Цель работы**

Исследовать возможности применения AVL-деревьев – для хранения, поиска и обработки информации. Приобрести практические использования классов, реализующих AVL-деревья. Оценить эффективность использования AVL-деревья по сравнению с бинарными деревьями поиска.

**Вариант задания**

Вариант 8

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Ключевое поле | N1 | N2 | N3 | N4 | N5 |
| Абонент | 30 | 500 | 1900 | 4000 | 7500 |

**Ход выполнения работы**

Была написана программа на языке С++, которая реализует классы для работы с AVL-деревом и с бинарным деревом поиска. Текст программы представлен в приложении А.

В качестве информационного поля для дерева и списка используется структура Data, которая содержит 5 полей, а также перегруженные операторы ввода и вывода. Для заполнения информационных полей были написаны функции для генерации случайных значений.

Для описания элемента AVL-дерева используется класс Node, он содержит информационное поле и указатели на левого и правого потомков и показатель сбалансированности. Для описания узла дерева используется класс BinaryTreeNode, он также содержит информационное поле и указатели на левого и правого потомков.

Для работы с самими структурами данных написаны соответствующие классы AVLTree и BinarySearchTree. Для них определены методы добавления, поиска и удаления. Также для них определен метод, который генерирует файл для пакета утилит визуализации графов. В этом файле описываются все связи необходимые для визуального построения AVL-дерева и бинарного дерева поиска.

В таблице 2.1 отражено время выполнения операций при разном количестве элементов в AVL-дереве и бинарном дереве.

Таблица 2.1. Результаты проделанной работы

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Время N | № | AVL-дерево | | | Бинарное дерево | | |
| Добавление | Удаление | Поиск | Добавление | Удаление | Поиск |
| N1 = 30 | 1 | 0,040 | 0,001 | 0,003 | 0,052 | 0,045 | 0,044 |
| 2 | 0,055 | 0,001 | 0,001 | 0,044 | 0,089 | 0,057 |
| 3 | 0,041 | 0,001 | 0,002 | 0,046 | 0,046 | 0,043 |
| 4 | 0,040 | 0,001 | 0,001 | 0,064 | 0,044 | 0,052 |
| 5 | 0,046 | 0,002 | 0,001 | 0,213 | 0,063 | 0,051 |
| Ср. | 0,044 | 0,002 | 0,001 | 0,084 | 0,049 | 0,057 |
| N2 = 500 | 1 | 0,049 | 0,002 | 0,002 | 0,055 | 0,047 | 0,045 |
| 2 | 0,042 | 0,002 | 0,001 | 0,089 | 0,045 | 0,049 |
| 3 | 0,038 | 0,002 | 0,001 | 0,044 | 0,049 | 0,212 |
| 4 | 0,037 | 0,004 | 0,002 | 0,065 | 0,043 | 0,044 |
| 5 | 0,043 | 0,002 | 0,002 | 0,066 | 0,048 | 0,043 |
| Ср. | 0,042 | 0,002 | 0,002 | 0,064 | 0,079 | 0,046 |
| N3 = 1900 | 1 | 0,059 | 0,041 | 0,041 | 0,044 | 0,003 | 0,002 |
| 2 | 0,049 | 0,043 | 0,052 | 0,048 | 0,003 | 0,001 |
| 3 | 0,045 | 0,053 | 0,067 | 0,038 | 0,002 | 0,004 |
| 4 | 0,050 | 0,059 | 0,047 | 0,037 | 0,002 | 0,001 |
| 5 | 0,049 | 0,050 | 0,044 | 0,039 | 0,003 | 0,001 |
| Ср. | 0,051 | 0,050 | 0,049 | 0,041 | 0,002 | 0,003 |
| N4 = 4000 | 1 | 0,069 | 0,042 | 0,049 | 0,041 | 0,003 | 0,002 |
| 2 | 0,042 | 0,054 | 0,040 | 0,045 | 0,003 | 0,002 |
| 3 | 0,049 | 0,044 | 0,064 | 0,039 | 0,004 | 0,002 |
| 4 | 0,039 | 0,066 | 0,063 | 0,039 | 0,005 | 0,002 |
| 5 | 0,042 | 0,063 | 0,062 | 0,036 | 0,003 | 0,002 |
| Ср. | 0,048 | 0,055 | 0,054 | 0,040 | 0,002 | 0,003 |
| N5 = 7500 | 1 | 0,049 | 0,042 | 0,039 | 0,043 | 0,002 | 0,002 |
| 2 | 0,064 | 0,049 | 0,074 | 0,044 | 0,003 | 0,002 |
| 3 | 0,059 | 0,044 | 0,054 | 0,044 | 0,003 | 0,001 |
| 4 | 0,048 | 0,049 | 0,044 | 0,038 | 0,002 | 0,001 |
| 5 | 0,049 | 0,046 | 0,064 | 0,056 | 0,003 | 0,001 |
| Ср. | 0,054 | 0,055 | 0,046 | 0,045 | 0,002 | 0,003 |

Было проведено добавление 100000 элементов в AVL-дерево и бинарное дерево поиска. Результаты изображены на рисунке 1.

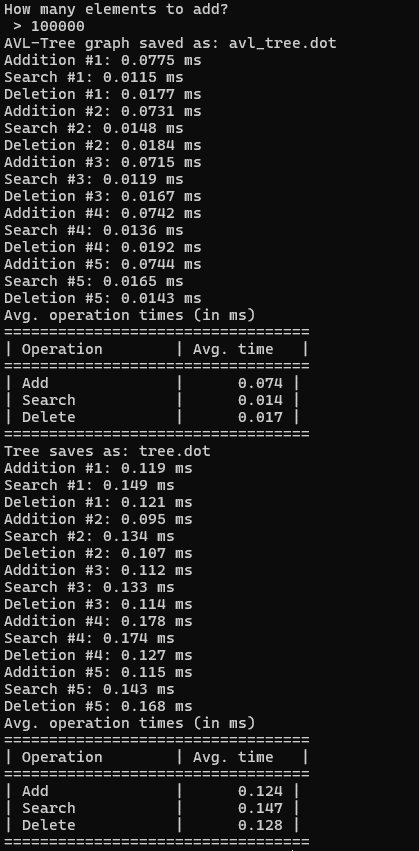


Рисунок 2.2 – Результаты операций добавления, удаления и поиска для бинарного дерева поиска и AVL-дерева при 100000 элементов

При помощи пакета утилит для визуализации графов была выведена структура бинарного дерева поиска на 150 элементов. Для этих же элементов было построено AVL дерево. Это изображено на рисунках 2 и 3.

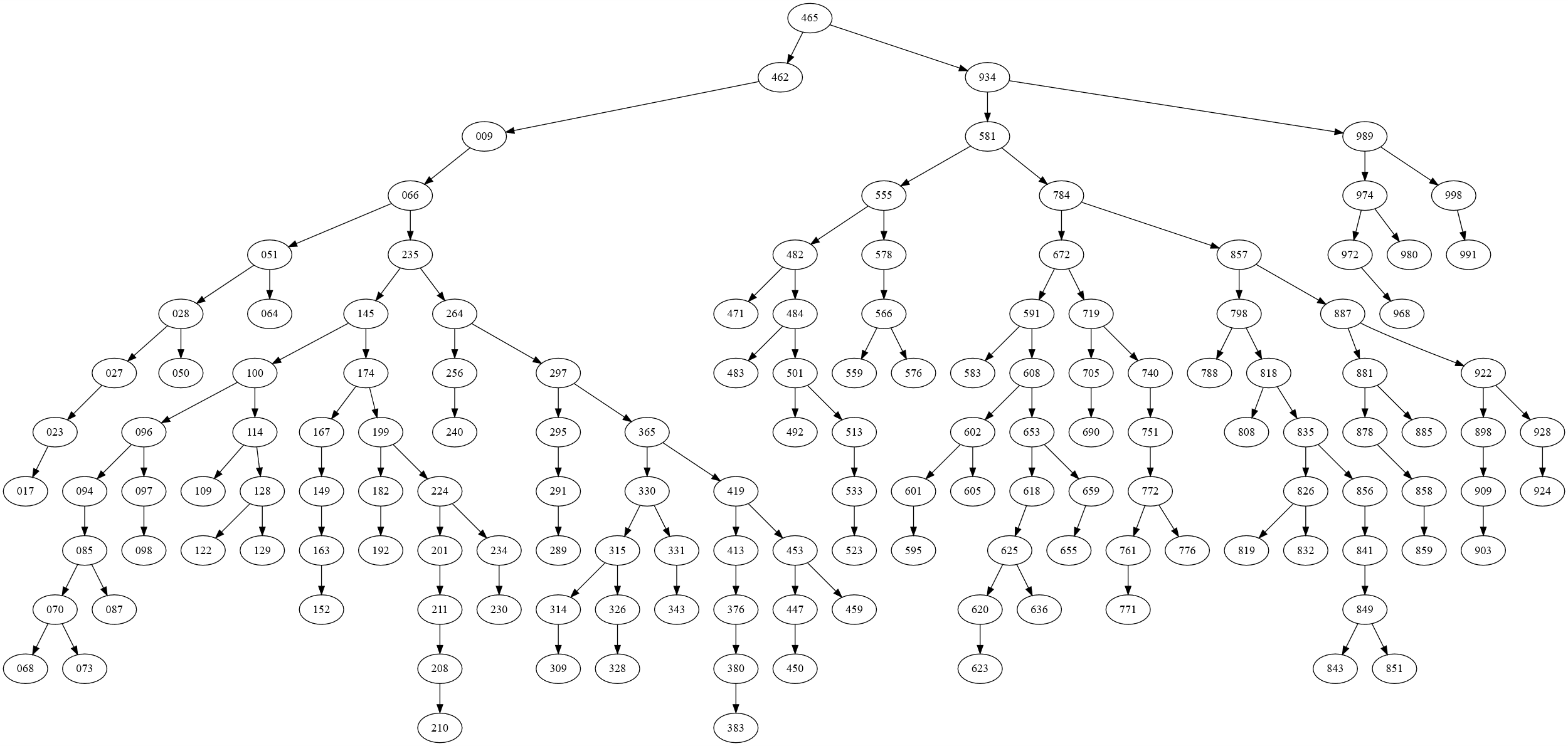


Рисунок 2 – Бинарное дерево из 150 элементов

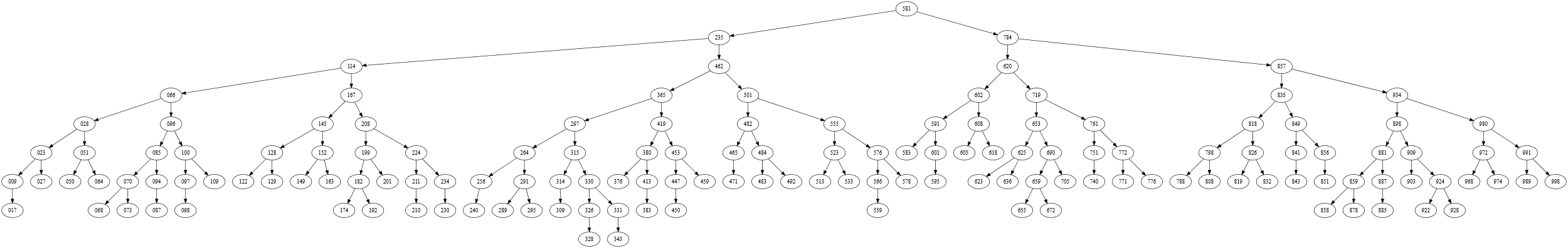


Рисунок 3 – AVL-дерево из 150 элементов

**Вывод**

В ходе работы была разработана программа, в которой реализованы классы, осуществляющие различные операции с AVL-деревом и бинарным деревом поиска. Было выявлено, что AVL-дерево лучше показывает себя по всем операциям. Это достигается за счет сбалансированности AVL-дерева.

**ПРИЛОЖЕНИЕ А**

Листинг 1 - Текст программы с реализацией классов для работы с бинарными деревьями и с AVL-деревьями.

#include <iostream>

#include <windows.h>

#include <fstream>

#include <random>

#include <ctime>

#include <vector>

#include <chrono>

#include <iomanip>

#include <string>

using namespace std;

struct Data {

int ip;

string subscriber;

string phoneNumber;

string name;

int amount;

friend ostream& operator<<(ostream& os, const Data& data);

friend istream& operator>>(istream& is, Data& data);

};

ostream& operator<<(ostream& os, const Data& data) {

os << data.ip << " " << data.subscriber << " " << data.phoneNumber << " " << data.name << " " << data.amount;

return os;

}

istream& operator>>(istream& is, Data& data) {

is >> data.ip >> data.subscriber >> data.phoneNumber >> data.name >> data.amount;

return is;

}

string generateRandomPhoneNumber() {

random\_device rd;

mt19937 gen(rd());

uniform\_int\_distribution<> dis(0, 9);

string phoneNumber = "";

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

if (i == 0) {

phoneNumber += to\_string(dis(gen) + 1);

}

else {

phoneNumber += to\_string(dis(gen));

}

}

return "+7" + phoneNumber;

};

vector<string> firstNames = {

"Liam", "Emma", "Noah", "Olivia", "William", "Ava", "James", "Isabella",

"Oliver", "Sophia", "Benjamin", "Mia", "Elijah", "Charlotte", "Lucas", "Harper"

};

vector<string> lastNames = {

"Smith", "Johnson", "Williams", "Brown", "Davis", "Miller", "Wilson", "Moore",

"Taylor", "Anderson", "Thomas", "Jackson", "White", "Harris", "Martin", "Thompson"

};

string generateRandomName() {

static random\_device rd;

static mt19937 gen(rd());

static uniform\_int\_distribution<> firstNameDist(0, firstNames.size() - 1);

static uniform\_int\_distribution<> lastNameDist(0, lastNames.size() - 1);

string firstName = firstNames[firstNameDist(gen)];

string lastName = lastNames[lastNameDist(gen)];

return firstName + " " + lastName;

}

int generateRandomNumber() {

random\_device rd;

mt19937 gen(rd());

uniform\_int\_distribution<> dis(100000, 999999);

int randomNumber = dis(gen);

return randomNumber;

}

string generateRandomString() {

static random\_device rd;

static mt19937 gen(rd());

static uniform\_int\_distribution<> charDist(48, 57);

string randomString;

for (int i = 0; i < 3; i++) {

randomString += static\_cast<char>(charDist(gen));

}

return randomString;

}

Data generateRandomData(int i) {

Data data;

data.ip = i + 1;

data.subscriber = generateRandomString();

data.phoneNumber = generateRandomPhoneNumber();

data.name = generateRandomName();

data.amount = generateRandomNumber();

return data;

}

class Node {

public:

Data data;

Node\* left;

Node\* right;

int height;

Node(Data data) : data(data), left(nullptr), right(nullptr), height(1) {}

};

class AVLTree {

private:

Node\* root;

public:

int height(Node\* node) {

if (node == nullptr)

return 0;

return node->height;

}

Node\* rightRotate(Node\* z) {

Node\* y = z->left;

Node\* T3 = y->right;

y->right = z;

z->left = T3;

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

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

return y;

}

Node\* leftRotate(Node\* z) {

Node\* y = z->right;

Node\* T2 = y->left;

y->left = z;

z->right = T2;

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

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

return y;

}

int getBalance(Node\* node) {

if (node == nullptr)

return 0;

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

}

Node\* insert(Data data, Node\* node) {

if (node == nullptr)

return new Node(data);

if (data.subscriber < node->data.subscriber)

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

else if (data.subscriber > node->data.subscriber)

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

else

return node;

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

int balance = getBalance(node);

if (balance > 1 && data.subscriber < node->left->data.subscriber)

return rightRotate(node);

if (balance < -1 && data.subscriber > node->right->data.subscriber)

return leftRotate(node);

if (balance > 1 && data.subscriber > node->left->data.subscriber) {

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

return rightRotate(node);

}

if (balance < -1 && data.subscriber < node->right->data.subscriber) {

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

return leftRotate(node);

}

return node;

}

Node\* remove(string subscriber, Node\* node) {

if (node == nullptr)

return node;

if (subscriber < node->data.subscriber)

node->left = remove(subscriber, node->left);

else if (subscriber > node->data.subscriber)

node->right = remove(subscriber, node->right);

else {

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 = minValueNode(node->right);

node->data = temp->data;

node->right = remove(temp->data.subscriber, node->right);

}

}

if (node == nullptr)

return node;

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

int balance = getBalance(node);

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

return rightRotate(node);

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

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

return rightRotate(node);

}

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

return leftRotate(node);

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

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

return leftRotate(node);

}

return node;

}

Node\* minValueNode(Node\* node) {

Node\* current = node;

while (current->left != nullptr)

current = current->left;

return current;

}

void inorderTraversal(Node\* node) {

if (node != nullptr) {

inorderTraversal(node->left);

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

inorderTraversal(node->right);

}

}

void preorderTraversal(Node\* node) {

if (node != nullptr) {

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

preorderTraversal(node->left);

preorderTraversal(node->right);

}

}

void postorderTraversal(Node\* node) {

if (node != nullptr) {

postorderTraversal(node->left);

postorderTraversal(node->right);

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

}

}

AVLTree() : root(nullptr) {}

void insert(Data data) {

root = insert(data, root);

}

void deleteKey(string subscriber) {

root = remove(subscriber, root);

}

void inorderTraversal() {

inorderTraversal(root);

cout << endl;

}

void preorderTraversal() {

preorderTraversal(root);

cout << endl;

}

void postorderTraversal() {

postorderTraversal(root);

cout << endl;

}

Node\* find(Node\* node, string subscriber) {

if (node == nullptr || node->data.subscriber == subscriber)

return node;

if (subscriber < node->data.subscriber)

return find(node->left, subscriber);

else

return find(node->right, subscriber);

}

private:

void createDotFile(Node\* node, ofstream& file) {

if (node == nullptr)

return;

file << node->data.subscriber << endl;

if (node->left) {

file << node->data.subscriber << " -> " << node->left->data.subscriber << ";" << endl;

createDotFile(node->left, file);

}

if (node->right) {

file << node->data.subscriber << " -> " << node->right->data.subscriber << ";" << endl;

createDotFile(node->right, file);

}

}

public:

void visualizeTree(const string& filename) {

ofstream dotFile(filename);

if (!dotFile.is\_open()) {

cerr << "Cannot access file: " << filename << endl;

return;

}

dotFile << "digraph AVLTree {" << endl;

createDotFile(root, dotFile);

dotFile << "}" << endl;

dotFile.close();

cout << "AVL-Tree graph saved as: " << filename << endl;

}

void printTimeTable() {

double addTime = 0.0, deleteTime = 0.0, searchTime = 0.0;

for (int i = 0; i < 5; i++) {

auto start = chrono::high\_resolution\_clock::now();

insert(generateRandomData(i));

auto end = chrono::high\_resolution\_clock::now();

double oneElementAddTime = chrono::duration<double, milli>(end - start).count();

cout << "Addition #" << i + 1 << ": " << oneElementAddTime << " ms" << endl;

addTime += oneElementAddTime;

start = chrono::high\_resolution\_clock::now();

find(root, generateRandomString());

end = chrono::high\_resolution\_clock::now();

double oneElementSearchTime = chrono::duration<double, milli>(end - start).count();

cout << "Search #" << i + 1 << ": " << oneElementSearchTime << " ms" << endl;

searchTime += oneElementSearchTime;

start = chrono::high\_resolution\_clock::now();

remove(generateRandomString(), root);

end = chrono::high\_resolution\_clock::now();

double oneElementDeleteTime = chrono::duration<double, milli>(end - start).count();

cout << "Deletion #" << i + 1 << ": " << oneElementDeleteTime << " ms" << endl;

deleteTime += oneElementDeleteTime;

}

cout << "Avg. operation times (in ms)" << endl;

cout << "==================================" << endl;

cout << "| Operation | Avg. time |" << endl;

cout << "==================================" << endl;

cout << "| Add | " << fixed << setprecision(3) << setw(10) << addTime / 5 << " |" << endl;

cout << "| Search | " << fixed << setprecision(3) << setw(10) << searchTime / 5 << " |" << endl;

cout << "| Delete | " << fixed << setprecision(3) << setw(10) << deleteTime / 5 << " |" << endl;

cout << "==================================" << endl;

}

};

class BinaryTreeNode {

private:

Data data;

BinaryTreeNode\* left;

BinaryTreeNode\* right;

public:

BinaryTreeNode(Data value) : data(value), left(nullptr), right(nullptr) {}

BinaryTreeNode\* getLeft() { return left; }

BinaryTreeNode\* getRight() { return right; }

void setLeft(BinaryTreeNode\* node) { left = node; }

void setRight(BinaryTreeNode\* node) { right = node; }

Data getData() { return data; }

void setData(Data setData) { data = setData; }

};

class BinarySearchTree {

private:

BinaryTreeNode\* root;

public:

BinarySearchTree() : root(nullptr) {}

BinaryTreeNode\* insertRecursive(BinaryTreeNode\* node, Data value) {

if (node == nullptr) {

return new BinaryTreeNode(value);

}

if (value.subscriber < node->getData().subscriber) {

node->setLeft(insertRecursive(node->getLeft(), value));

}

else if (value.subscriber > node->getData().subscriber) {

node->setRight(insertRecursive(node->getRight(), value));

}

return node;

}

void insert(Data value) {

root = insertRecursive(root, value);

}

void displayRecursive(BinaryTreeNode\* node, ofstream& file) {

if (node == nullptr) {

return;

}

file << " " << node->getData().subscriber << ";" << endl;

if (node->getLeft() != nullptr) {

file << " " << node->getData().subscriber << " -> " << node->getLeft()->getData().subscriber << ";" << endl;

displayRecursive(node->getLeft(), file);

}

if (node->getRight() != nullptr) {

file << " " << node->getData().subscriber

<< " -> " << node->getRight()->getData().subscriber << ";" << endl;

displayRecursive(node->getRight(), file);

}

}

void displayAsGraph() {

ofstream file("tree.dot", ios::out);

if (!file) {

cerr << "Cannot access file:" << endl;

return;

}

file << "digraph BinarySearchTree {" << endl;

displayRecursive(root, file);

file << "}" << endl;

file.close();

cout << "Tree saves as: tree.dot" << endl;

}

BinaryTreeNode\* findRecursive(BinaryTreeNode\* node, Data value) {

if (node == nullptr || node->getData().subscriber == value.subscriber) {

return node;

}

if (value.subscriber < node->getData().subscriber) {

return findRecursive(node->getLeft(), value);

}

else {

return findRecursive(node->getRight(), value);

}

}

BinaryTreeNode\* find(Data value) {

return findRecursive(root, value);

}

BinaryTreeNode\* findMinRecursive(BinaryTreeNode\* node) {

if (node->getLeft() == nullptr) {

return node;

}

return findMinRecursive(node->getLeft());

}

BinaryTreeNode\* findMin() {

return findMinRecursive(root);

}

BinaryTreeNode\* removeRecursive(BinaryTreeNode\* node, Data value) {

if (node == nullptr) {

return nullptr;

}

if (value.subscriber < node->getData().subscriber) {

node->setLeft(removeRecursive(node->getLeft(), value));

}

else if (value.subscriber > node->getData().subscriber) {

node->setRight(removeRecursive(node->getRight(), value));

}

else {

if (node->getLeft() == nullptr) {

BinaryTreeNode\* temp = node->getRight();

delete node;

return temp;

}

else if (node->getRight() == nullptr) {

BinaryTreeNode\* temp = node->getLeft();

delete node;

return temp;

}

node->setData(findMinRecursive(node->getRight())->getData());

node->setRight(removeRecursive(node->getRight(), node->getData()));

}

return node;

}

void remove(Data value) {

root = removeRecursive(root, value);

}

void printTimeTable() {

double addTime = 0.0, deleteTime = 0.0, searchTime = 0.0;

for (int i = 0; i < 5; i++) {

auto start = chrono::high\_resolution\_clock::now();

insert(generateRandomData(i));

auto end = chrono::high\_resolution\_clock::now();

double oneElementAddTime = chrono::duration<double, milli>(end - start).count();

cout << "Addition #" << i + 1 << ": " << oneElementAddTime << " ms" << endl;

addTime += oneElementAddTime;

start = chrono::high\_resolution\_clock::now();

find(generateRandomData(i));

end = chrono::high\_resolution\_clock::now();

double oneElementSearchTime = chrono::duration<double, milli>(end - start).count();

cout << "Search #" << i + 1 << ": " << oneElementSearchTime << " ms" << endl;

searchTime += oneElementSearchTime;

start = chrono::high\_resolution\_clock::now();

remove(generateRandomData(i));

end = chrono::high\_resolution\_clock::now();

double oneElementDeleteTime = chrono::duration<double, milli>(end - start).count();

cout << "Deletion #" << i + 1 << ": " << oneElementDeleteTime << " ms" << endl;

deleteTime += oneElementDeleteTime;

}

cout << "Avg. operation times (in ms)" << endl;

cout << "==================================" << endl;

cout << "| Operation | Avg. time |" << endl;

cout << "==================================" << endl;

cout << "| Add | " << fixed << setprecision(3) << setw(10) << addTime / 5 << " |" << endl;

cout << "| Search | " << fixed << setprecision(3) << setw(10) << searchTime / 5 << " |" << endl;

cout << "| Delete | " << fixed << setprecision(3) << setw(10) << deleteTime / 5 << " |" << endl;

cout << "==================================" << endl;

}

};

void addMultiplyElements(int elementsAmount, AVLTree& avl, BinarySearchTree& tree) {

for (int i = 0; i < elementsAmount; i++) {

Data data = generateRandomData(i);

avl.insert(data);

tree.insert(data);

}

}

int main() {

SetConsoleOutputCP(65001);

AVLTree avl;

BinarySearchTree tree;

int elementsAmount;

cout << "How many elements to add?" << endl << " > ";

cin >> elementsAmount;

addMultiplyElements(elementsAmount, avl, tree);

avl.visualizeTree("avl\_tree.dot");

avl.printTimeTable();

tree.displayAsGraph();

tree.printTimeTable();

system("pause");

return 0;

}