Министерство высшего образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

**«Пермский национальный исследовательский политехнический университет» (ПНИПУ)**

Электротехнический факультет

Кафедра «Информационные технологии и автоматизированные системы»

ОТЧЁТ

по лабораторной работе №1

Тема: «Бинарные деревья»

Выполнил

Студент группы РИС-22-2б

Зубов Р. А.

Проверил доц. Кафедры ИТАС

Полякова О. А.

Пермь 2023

**Постановка задачи**

1. Сформировать идеально сбалансированное бинарное дерево, тип

информационного поля указан в варианте.

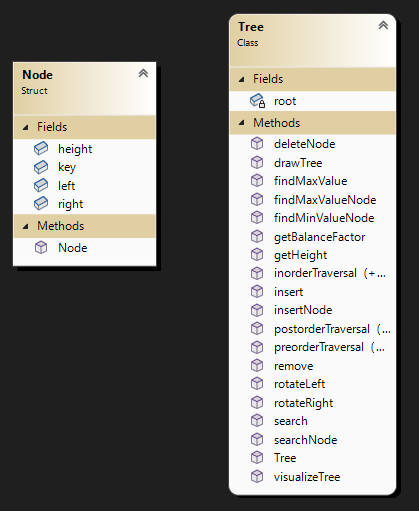
2. Распечатать полученное дерево.

3. Выполнить обработку дерева в соответствии с заданием, вывести полученный

результат.

4. Преобразовать идеально сбалансированное дерево в дерево поиска.

5. Распечатать полученное дерево.  
  
Вариант 2: Тип информационного поля int. Найти максимальный элемент в дереве.

UML  
  
  
**Код программы**

**Source.cpp:**

#include <iostream>

#include "Tree.h"

int main() {

Tree tree;

tree.insert(50);

tree.insert(30);

tree.insert(20);

tree.insert(40);

tree.insert(70);

tree.insert(60);

tree.insert(80);

std::cout << "Preorder Traversal: ";

tree.preorderTraversal();

std::cout << "Inorder Traversal: ";

tree.inorderTraversal();

std::cout << "Postorder Traversal: ";

tree.postorderTraversal();

std::cout << "Searching for key 40: " << (tree.search(40) ? "Found" : "Not found") << std::endl;

std::cout << "Searching for key 90: " << (tree.search(90) ? "Found" : "Not found") << std::endl;

tree.visualizeTree();

tree.remove(30);

std::cout << "Inorder Traversal after removing key 30: ";

tree.inorderTraversal();

std::cout << std::endl;

std::cout << "Max value: " << tree.findMaxValue() << std::endl;

tree.visualizeTree();

return 0;

}

**Tree.h:**

#pragma once

#include <SFML/Graphics.hpp>

#include <iostream>

#include <algorithm>

struct Node {

int key;

Node\* left;

Node\* right;

int height;

Node(int k) {

key = k;

left = nullptr;

right = nullptr;

height = 1;

}

};

class Tree {

public:

Tree() {

root = nullptr;

}

int getHeight(Node\* node) {

if (node == nullptr) {

return 0;

}

return node->height;

}

int getBalanceFactor(Node\* node) {

if (node == nullptr) {

return 0;

}

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

}

Node\* rotateLeft(Node\* node) {

Node\* newRoot = node->right;

Node\* subtree = newRoot->left;

newRoot->left = node;

node->right = subtree;

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

newRoot->height = std::max(getHeight(newRoot->left), getHeight(newRoot->right)) + 1;

return newRoot;

}

Node\* rotateRight(Node\* node) {

Node\* newRoot = node->left;

Node\* subtree = newRoot->right;

newRoot->right = node;

node->left = subtree;

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

newRoot->height = std::max(getHeight(newRoot->left), getHeight(newRoot->right)) + 1;

return newRoot;

}

void insert(int key) { root = insertNode(root, key); }

void remove(int key) { root = deleteNode(root, key); }

Node\* insertNode(Node\* root, int key) {

if (root == nullptr) {

return new Node(key);

}

if (key < root->key) {

root->left = insertNode(root->left, key);

}

else if (key > root->key) {

root->right = insertNode(root->right, key);

}

else {

return root;

}

root->height = std::max(getHeight(root->left), getHeight(root->right)) + 1;

int balanceFactor = getBalanceFactor(root);

if (balanceFactor > 1 && key < root->left->key) {

return rotateRight(root);

}

if (balanceFactor < -1 && key > root->right->key) {

return rotateLeft(root);

}

if (balanceFactor > 1 && key > root->left->key) {

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

return rotateRight(root);

}

if (balanceFactor < -1 && key < root->right->key) {

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

return rotateLeft(root);

}

return root;

}

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

if (root == nullptr) {

return nullptr;

}

if (key < root->key) {

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

}

else if (key > root->key) {

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

}

else {

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

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

if (temp == nullptr) {

temp = root;

root = nullptr;

}

else {

\*root = \*temp;

}

delete temp;

}

else {

Node\* minValueNode = findMinValueNode(root->right);

root->key = minValueNode->key;

root->right = deleteNode(root->right, minValueNode->key);

}

}

if (root == nullptr) {

return nullptr;

}

root->height = std::max(getHeight(root->left), getHeight(root->right)) + 1;

int balanceFactor = getBalanceFactor(root);

if (balanceFactor > 1 && getBalanceFactor(root->left) >= 0) {

return rotateRight(root);

}

if (balanceFactor > 1 && getBalanceFactor(root->left) < 0) {

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

return rotateRight(root);

}

if (balanceFactor < -1 && getBalanceFactor(root->right) <= 0) {

return rotateLeft(root);

}

if (balanceFactor < -1 && getBalanceFactor(root->right) > 0) {

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

return rotateLeft(root);

}

return root;

}

Node\* findMinValueNode(Node\* node) {

Node\* current = node;

while (current && current->left != nullptr) {

current = current->left;

}

return current;

}

bool search(int key) {

Node\* result = searchNode(root, key);

return (result != nullptr);

}

Node\* searchNode(Node\* root, int key) {

if (root == nullptr || root->key == key) {

return root;

}

if (key < root->key) {

return searchNode(root->left, key);

}

return searchNode(root->right, key);

}

void preorderTraversal() {

preorderTraversal(root);

std::cout << std::endl;

}

void preorderTraversal(Node\* root) {

if (root != nullptr) {

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

preorderTraversal(root->left);

preorderTraversal(root->right);

}

}

void inorderTraversal() {

inorderTraversal(root);

std::cout << std::endl;

}

void inorderTraversal(Node\* root) {

if (root != nullptr) {

inorderTraversal(root->left);

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

inorderTraversal(root->right);

}

}

void postorderTraversal() {

postorderTraversal(root);

std::cout << std::endl;

}

void postorderTraversal(Node\* root) {

if (root != nullptr) {

postorderTraversal(root->left);

postorderTraversal(root->right);

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

}

}

int findMaxValue() {

if (root == nullptr) {

std::cout << "Tree is empty." << std::endl;

return INT\_MIN;

}

Node\* maxNode = findMaxValueNode(root);

return maxNode->key;

}

Node\* findMaxValueNode(Node\* node) {

Node\* current = node;

while (current && current->right != nullptr) {

current = current->right;

}

return current;

}

void drawTree(Node\* node, sf::RenderWindow& window, int x, int y, int xOffset, int levelHeight) {

if (node == nullptr) {

return;

}

int nodeRadius = 25;

int nodeSpacing = 50;

sf::CircleShape circle(nodeRadius);

circle.setFillColor(sf::Color::White);

circle.setOutlineThickness(2);

circle.setOutlineColor(sf::Color::Black);

circle.setPosition(x - nodeRadius, y - nodeRadius);

sf::Font font;

if (!font.loadFromFile("arial.ttf")) {

std::cerr << "Failed to load font." << std::endl;

return;

}

sf::Text text(std::to\_string(node->key), font, 20);

text.setFillColor(sf::Color::Black);

text.setPosition(x - nodeRadius + 15, y - nodeRadius + 10);

window.draw(circle);

window.draw(text);

int childOffset = xOffset / 2;

int nextLevelHeight = levelHeight + nodeSpacing;

if (node->left != nullptr) {

sf::Vertex line[] = {

sf::Vertex(sf::Vector2f(x, y + nodeRadius), sf::Color::Black),

sf::Vertex(sf::Vector2f(x - childOffset, y + nextLevelHeight - nodeRadius), sf::Color::Black)

};

window.draw(line, 2, sf::Lines);

drawTree(node->left, window, x - childOffset, y + nextLevelHeight, xOffset / 2, nextLevelHeight);

}

if (node->right != nullptr) {

sf::Vertex line[] = {

sf::Vertex(sf::Vector2f(x, y + nodeRadius), sf::Color::Black),

sf::Vertex(sf::Vector2f(x + childOffset, y + nextLevelHeight - nodeRadius), sf::Color::Black)

};

window.draw(line, 2, sf::Lines);

drawTree(node->right, window, x + childOffset, y + nextLevelHeight, xOffset / 2, nextLevelHeight);

}

}

void visualizeTree() {

sf::RenderWindow window(sf::VideoMode(800, 600), "AVL Tree Visualization");

window.setFramerateLimit(60);

while (window.isOpen()) {

sf::Event event;

while (window.pollEvent(event)) {

if (event.type == sf::Event::Closed) {

window.close();

}

}

window.clear(sf::Color::White);

if (root != nullptr) {

int treeHeight = root->height;

int treeWidth = static\_cast<int>(pow(2, treeHeight - 1)) \* 100;

drawTree(root, window, 400, 50, treeWidth / 2, 100);

}

window.display();

}

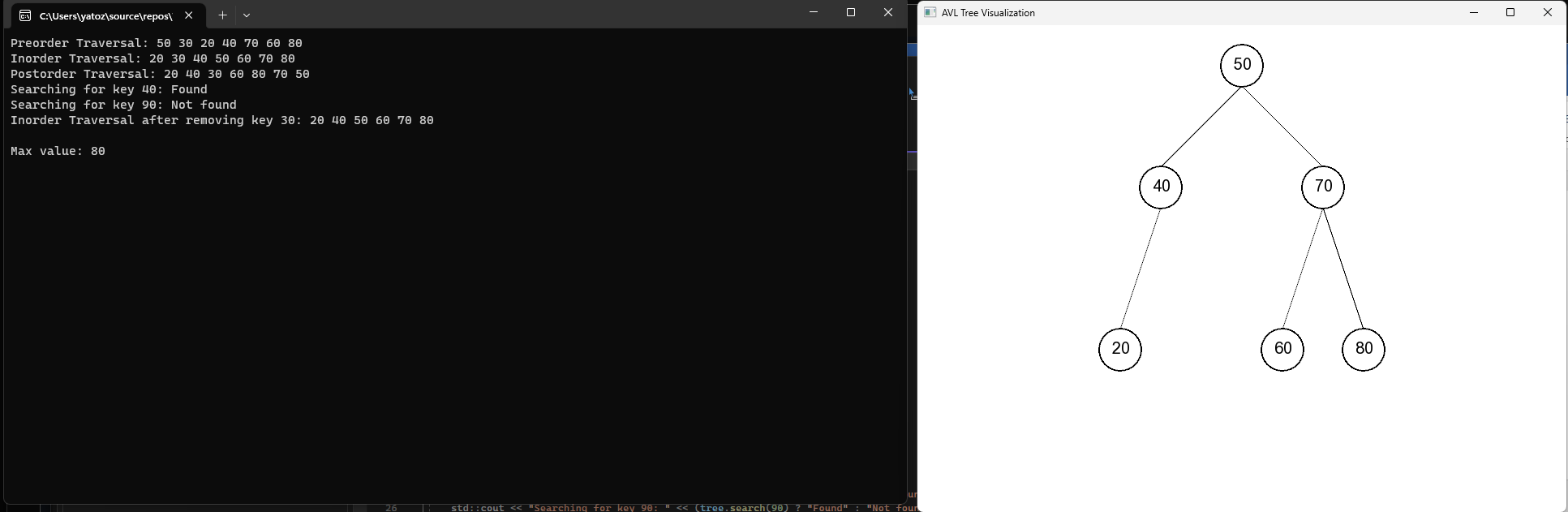
}

private:

Node\* root;

};

**Результат**

****