Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«Пермский национальный исследовательский**

**политехнический университет»**

Кафедра «Информационные технологии и автоматизированные системы»

**О Т Ч Ё Т**

**по лабораторной работе №20**

Дисциплина: Основы теории алгоритмов и структуры данных.

Тема: “ Бинарные деревья”

**Вар.24**

Выполнила работу:

студент группы ИВТ-20-2Б

Ананина Арина Юрьевна

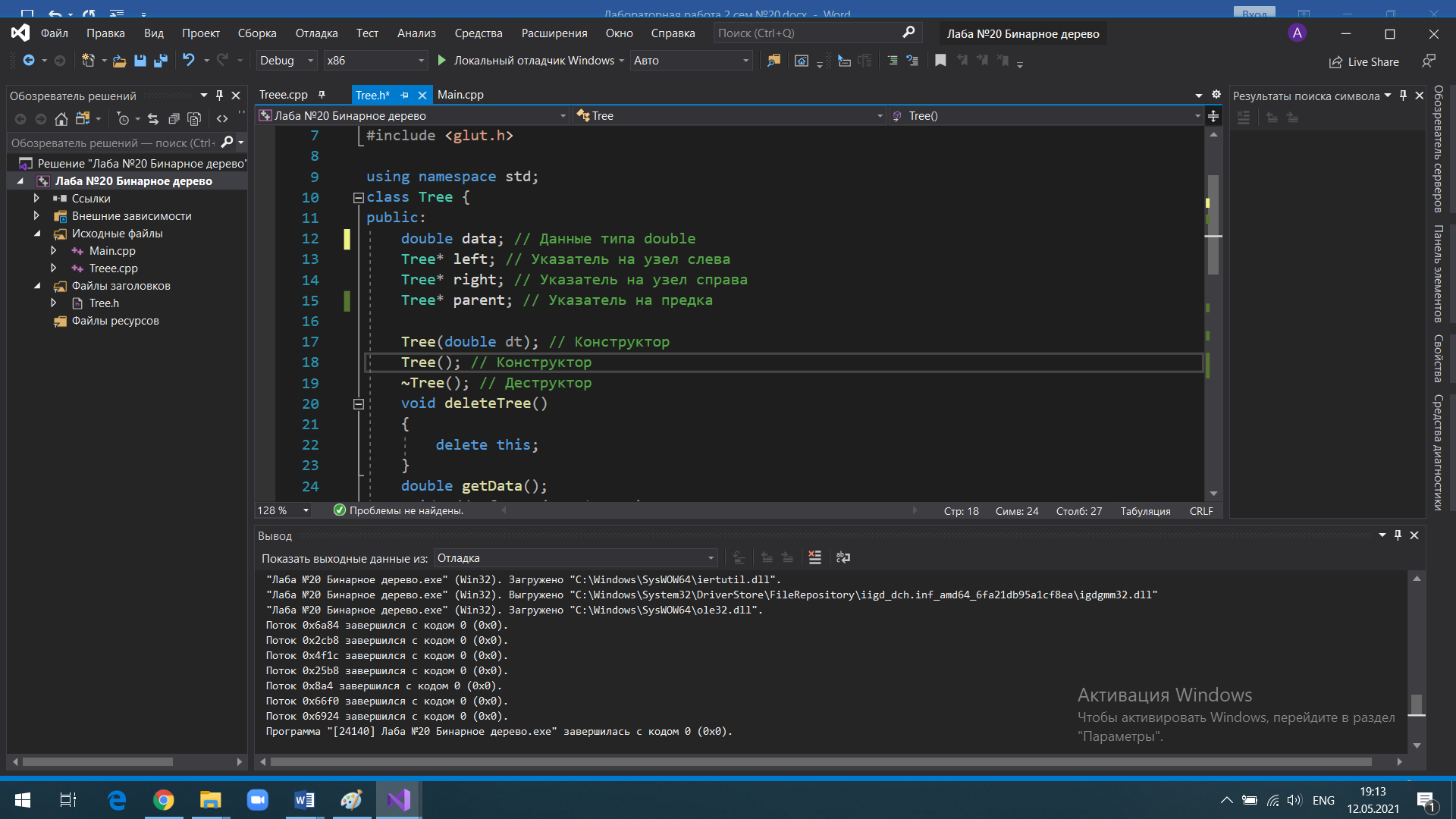
Проверила:

Доцент кафедры ИТАС

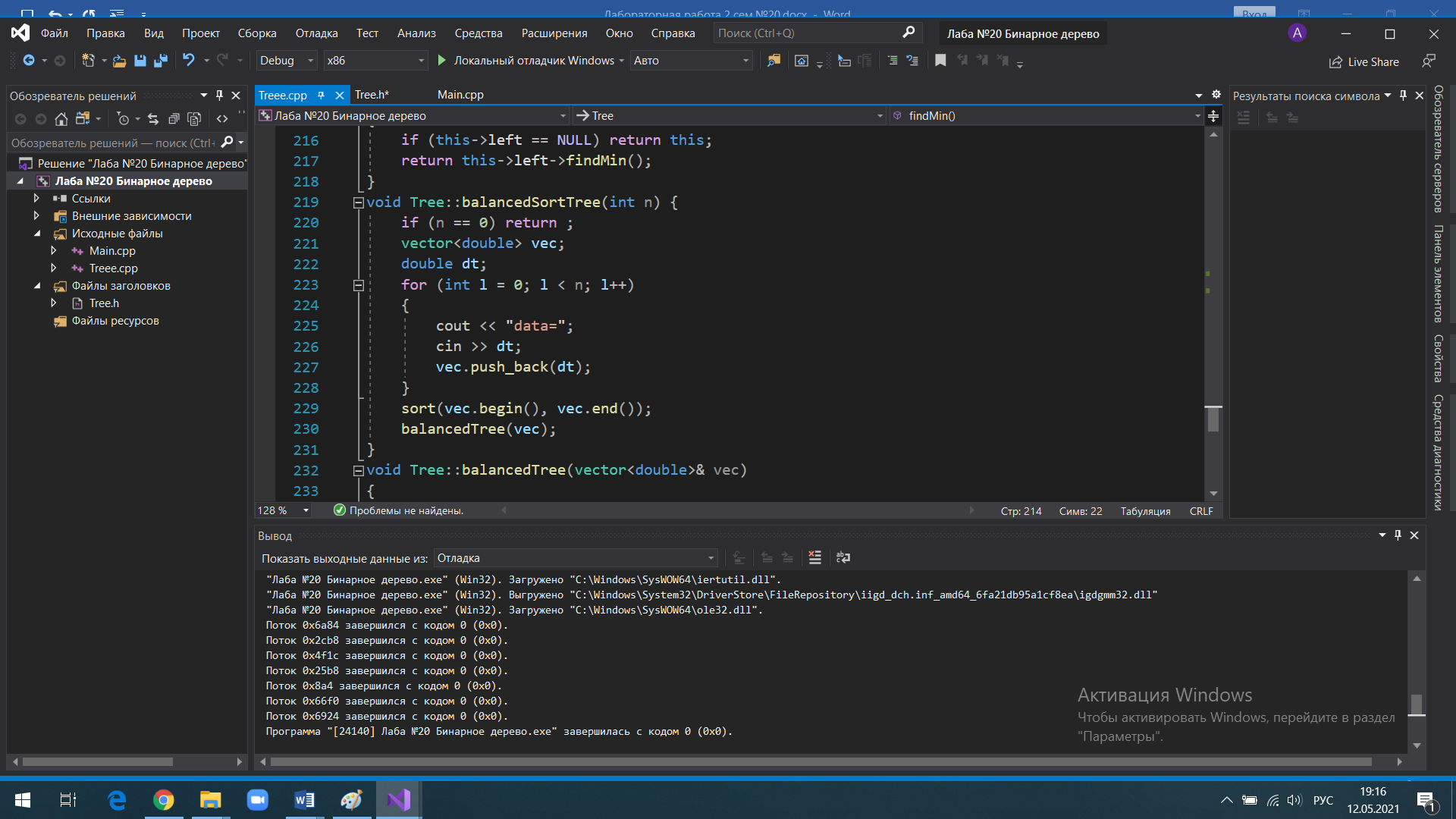
Полякова О.А.

Пермь, 2021

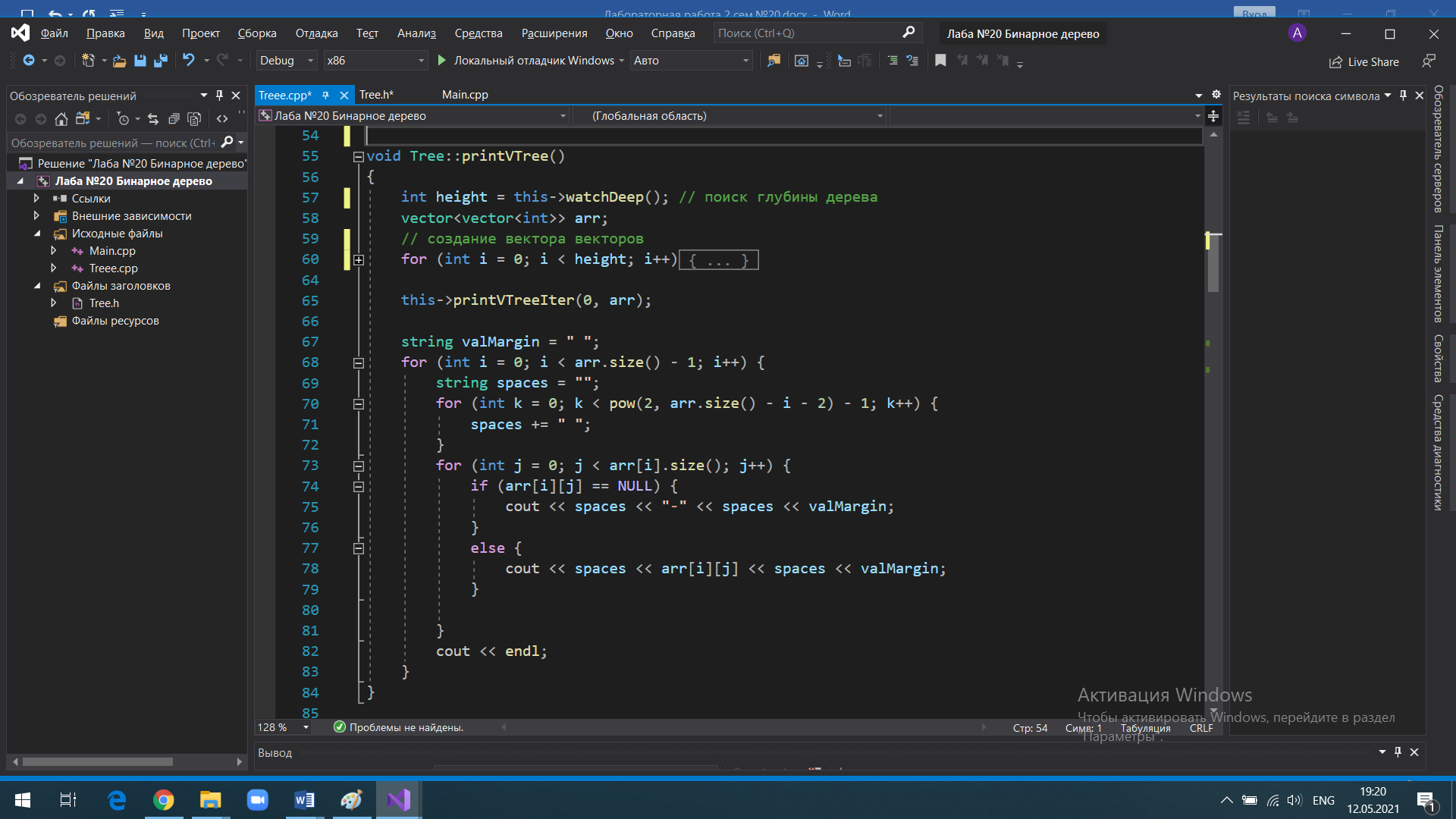
1. **Постановка задачи**
2. Сформировать идеально сбалансированное бинарное дерево, тип информационного поля double.
3. Преобразовать дерево в дерево поиска.
4. Распечатать полученное дерево.
5. Выполнить обработку дерева: найти максимальный элемент в дереве. Вывести результат.
6. **Анализ работы**
7. Какие типы данных были использованы.
   1. Были использованы типы данных: int, double, float и char.
8. Какие с этими данными надо выполнить действия.
   1. Переменные типа int используются в качестве счетчиков в циклах, для хранения констант и переменных для отрисовки дерева в OpenGL.
   2. Тип данных double используется для информационного поля внутри класса Tree.
   3. Переменные типа float и char необходимы для хранения, координат, размеров и символов для отрисовки дерева с помощью OpenGL.
9. В каком виде эти данные представлены.
   1. Данные представлены виде класса Tree.

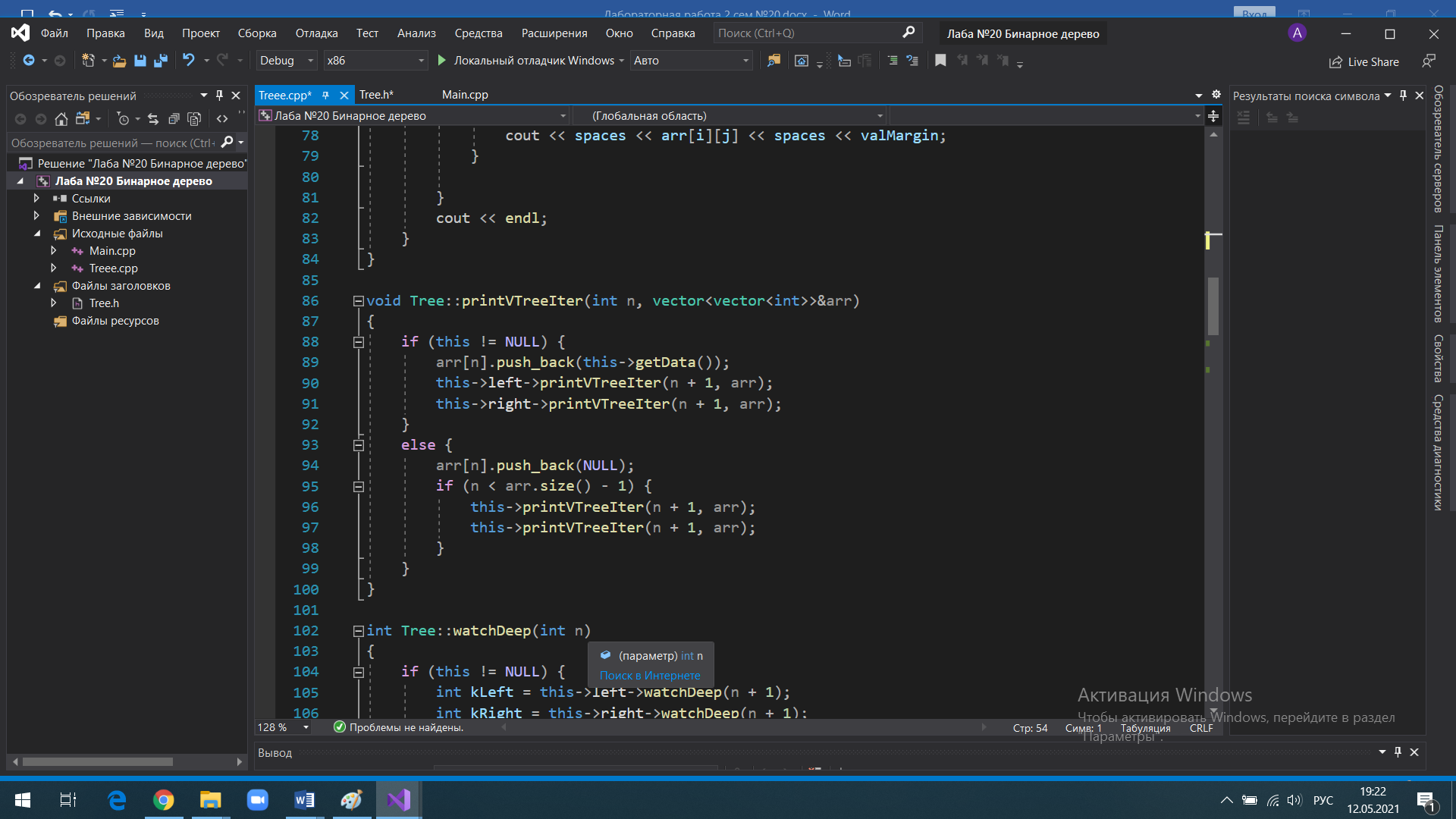


1. Какими операторами будет организован ввод и вывод.
   1. Ввод элементов дерева происходит в методе balancedSortTree с помощью оператора cin. Первоначально элементы записываются в вектор и сортируются для последующей обработки и записи в дерево.



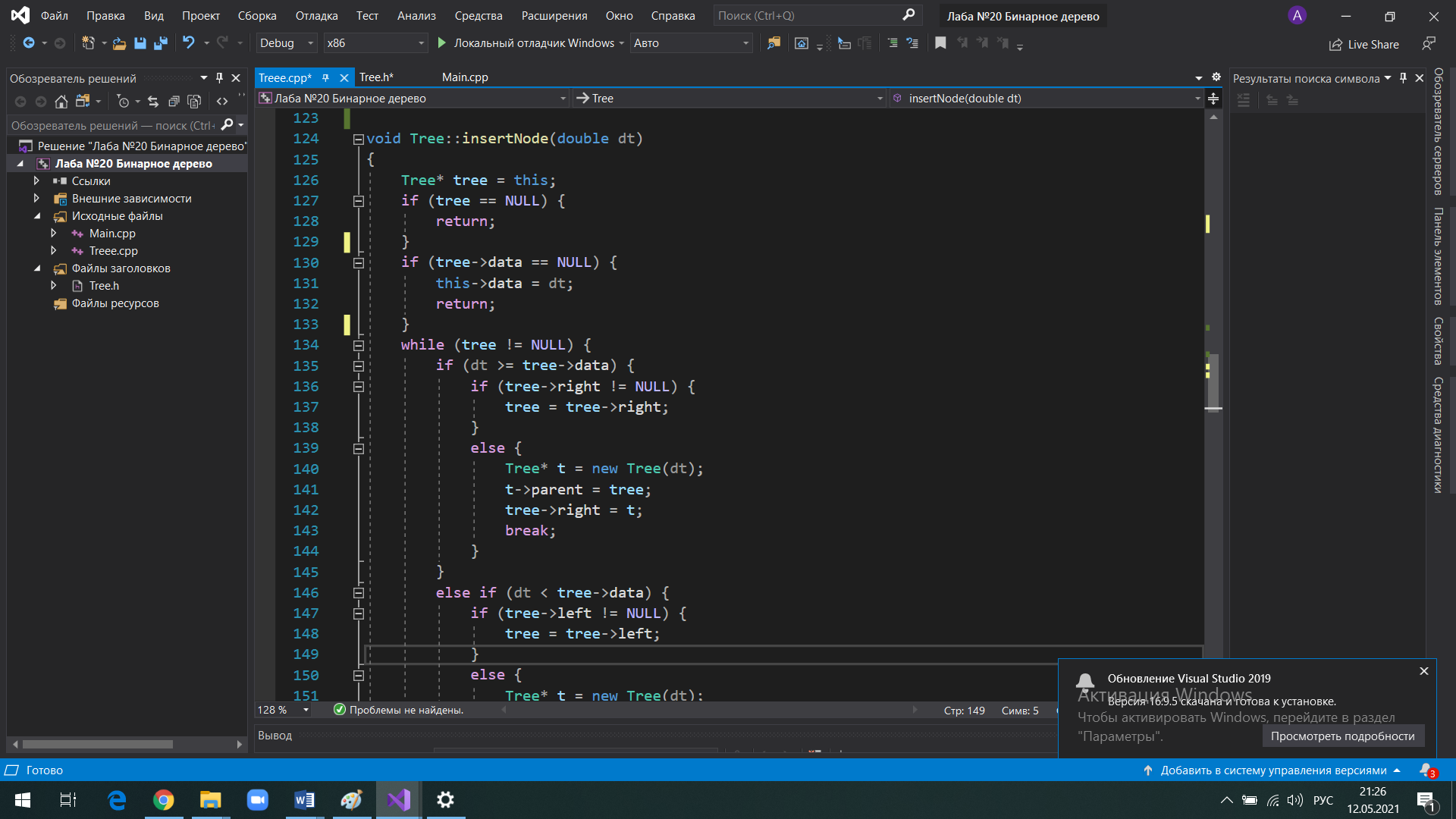
* 1. Вывод элементов дерева происходит с помощью двух методов printVTree и printVTreeIter и оператора cout.

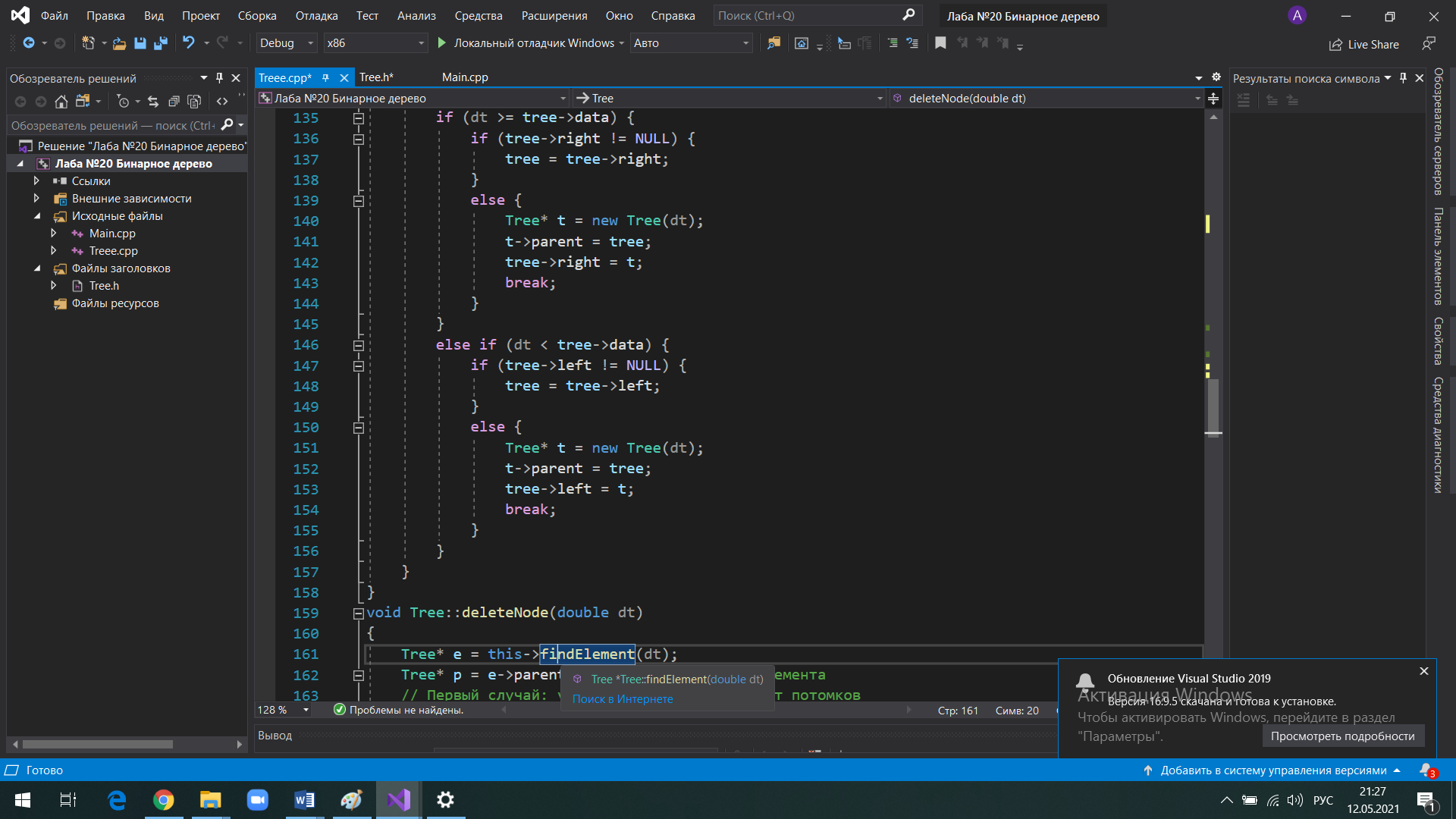




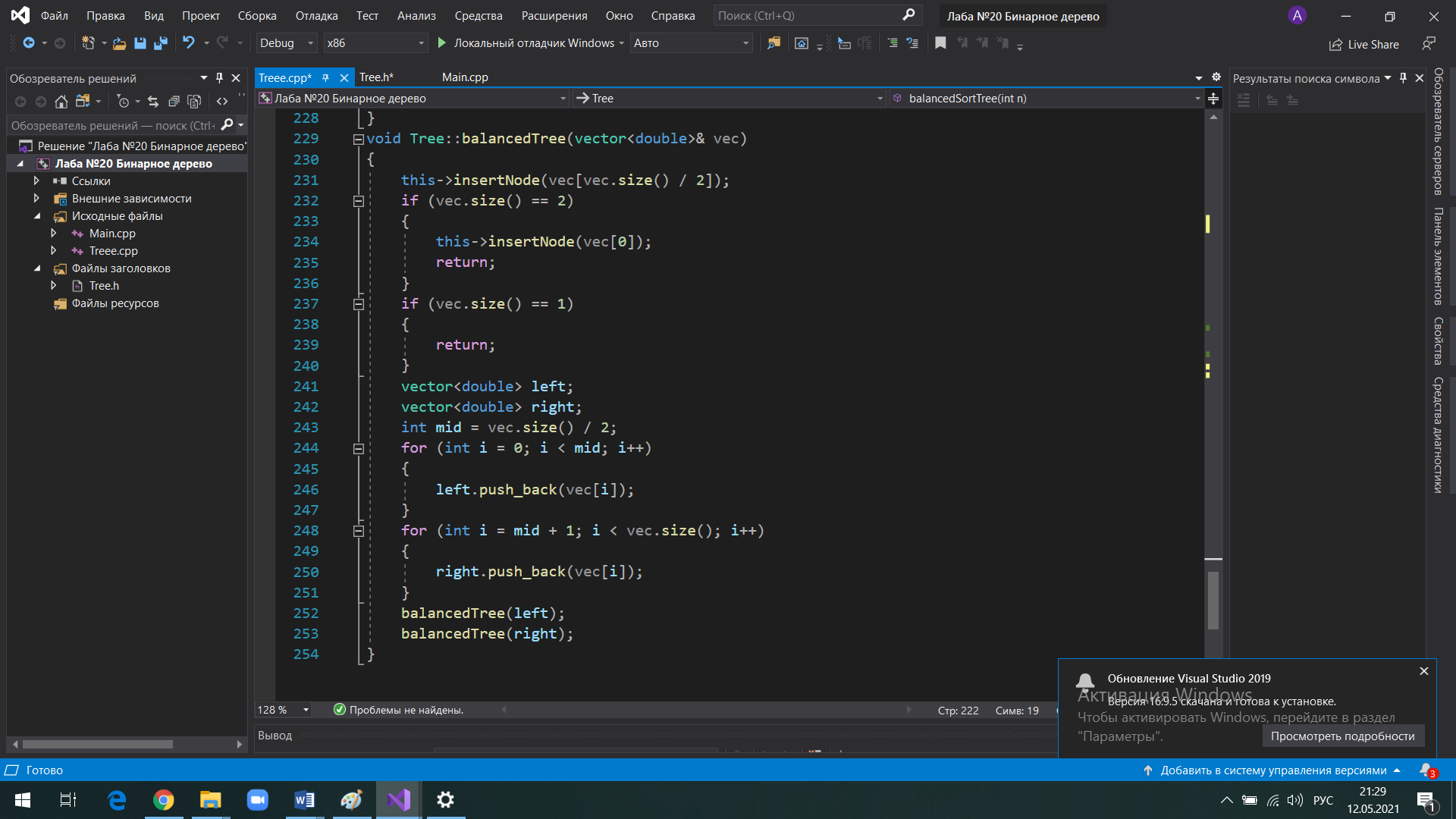
Отрисовка начинается с листьев дерева и заканчивается корнем.

1. Какими действиями будут решены поставленные задачи.
   1. С помощью метода insertNode осуществляется вставка узла, причём если вставляемый узел больше или равен текущему, то сравнение идёт со следующим правым узлом, иначе с левым. Таким образом создаётся бинарное дерево поиска.

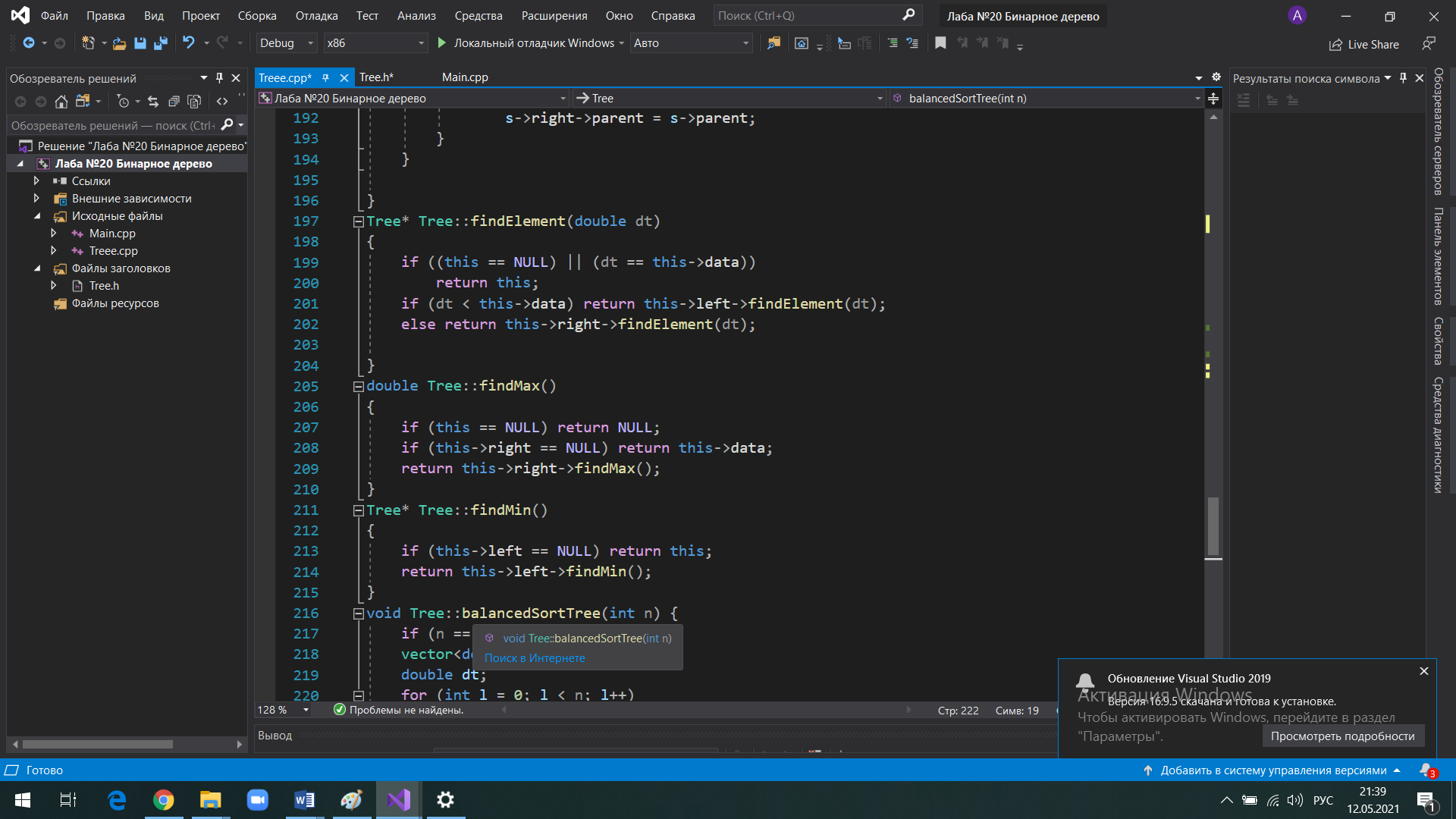




* 1. С помощью метода balancedTree осуществляется распределение элементов в дереве по принципу идеально сбалансированного дерева. Для этого изначально в качестве параметра поступает отсортированный вектор из элементов дерева, вектор делится пополам и средний элемент вставляется в дерево, после вызывается рекурсия и с правой и левой частью исходного вектора выполняются те же действия. Условие выхода из рекурсии – длина вектора 2 или 1, тогда просто идет вставка в дерево оставшихся элементов и выход из функции.



* 1. Для поиска максимального элемента применяется метод findMax. Внутри метода происходит обход дерева по правым узлам, пока ссылка текущего элемента на правый дочерний узел не станет равна NULL, если условие выполнено, то функция возвращает значение поля data у текущего узла. Таким образом будет найдено максимальный элемент дерева.



1. **Код программы**

Tree.h

#pragma once

#include <iostream>

#include <vector>

#include <Windows.h>

#include <gl/GL.h>

#include <gl/GLU.h>

#include <glut.h>

using namespace std;

class Tree {

public:

double data; // Данные типа double

Tree\* left; // Указатель на узел слева

Tree\* right; // Указатель на узел справа

Tree\* parent; // Указатель на предка

Tree(double dt); // Конструктор

Tree(); // Конструктор

~Tree(); // Деструктор

void deleteTree()

{

delete this;

}

double getData();

void addLeftTree(Tree\* tree);

void addRightTree(Tree\* tree);

// Удалить дерево

void printTree(int );

void printVTree();

void printVTreeIter(int, vector<vector<int>>& arr);

int watchDeep(int n = 1);

void setData(double dt)

{

data = dt;

}

Tree\* next(); // Найти следующий элемент

void insertNode(double dt); // Вставить узел

void deleteNode(double dt); // Удалить узел

Tree\* findElement(double dt); // Найти элемент

double findMax(); // Найти максимум

Tree\* findMin();

void balancedSortTree(int n);

void balancedTree(vector<double>& vec);

};

Tree.cpp

#include <string>

#include <algorithm>

using namespace std;

Tree::Tree(double dt)

{

data = dt;

left = NULL;

right = NULL;

parent = NULL;

}

Tree::Tree()

{

data = NULL;

left = NULL;

right = NULL;

parent = NULL;

}

Tree::~Tree()

{

delete this->left;

delete this->right;

}

double Tree::getData()

{

return data;

}

void Tree::addLeftTree(Tree\* tree)

{

left = tree;

}

void Tree::addRightTree(Tree\* tree)

{

right = tree;

}

void Tree::printTree(int n)

{

if (this != NULL) {

this->left->printTree(n + 1);

for (int i = 1; i < n; i++) cout << " ";

cout << this->getData() << endl;

this->right->printTree(n + 1);

}

}

void Tree::printVTree()

{

int height = this->watchDeep(); // поиск глубины дерева

vector<vector<int>> arr;

// создание вектора векторов

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

vector<int> in\_arr;

arr.push\_back(in\_arr);

}

this->printVTreeIter(0, arr);

string valMargin = " ";

for (int i = 0; i < arr.size() - 1; i++) {

string spaces = "";

for (int k = 0; k < pow(2, arr.size() - i - 2) - 1; k++) {

spaces += " ";

}

for (int j = 0; j < arr[i].size(); j++) {

if (arr[i][j] == NULL) {

cout << spaces << "-" << spaces << valMargin;

}

else {

cout << spaces << arr[i][j] << spaces << valMargin;

}

}

cout << endl;

}

}

void Tree::printVTreeIter(int n, vector<vector<int>>&arr)

{

if (this != NULL) {

arr[n].push\_back(this->getData());

this->left->printVTreeIter(n + 1, arr);

this->right->printVTreeIter(n + 1, arr);

}

else {

arr[n].push\_back(NULL);

if (n < arr.size() - 1) {

this->printVTreeIter(n + 1, arr);

this->printVTreeIter(n + 1, arr);

}

}

}

int Tree::watchDeep(int n)

{

if (this != NULL) {

int kLeft = this->left->watchDeep(n + 1);

int kRight = this->right->watchDeep(n + 1);

return (kLeft > kRight) ? kLeft : kRight;

}

return n;

}

Tree\* Tree::next()

{

Tree\* tree = this;

if (tree->right != NULL)

return tree->right->findMin();

Tree\* t = tree->parent;

while ((t != NULL) && (tree == t->right)) {

tree = t;

t = t->parent;

}

return t;

}

void Tree::insertNode(double dt)

{

Tree\* tree = this;

if (tree == NULL) {

return;

}

if (tree->data == NULL) {

this->data = dt;

return;

}

while (tree != NULL) {

if (dt >= tree->data) {

if (tree->right != NULL) {

tree = tree->right;

}

else {

Tree\* t = new Tree(dt);

t->parent = tree;

tree->right = t;

break;

}

}

else if (dt < tree->data) {

if (tree->left != NULL) {

tree = tree->left;

}

else {

Tree\* t = new Tree(dt);

t->parent = tree;

tree->left = t;

break;

}

}

}

}

void Tree::deleteNode(double dt)

{

Tree\* e = this->findElement(dt);

Tree\* p = e->parent; //предок удаляемого элемента

// Первый случай: удаляемый элемент не имеет потомков

if ((e->left == NULL) && (e->right == NULL)) {

if (p->left == e) p->left = NULL;

if (p->right == e) p->right = NULL;

}

else if ((e->left == NULL) || (e->right == NULL)) {

if (e->left == NULL) {

if (p->left == e) p->left = e->right;

else p->right = e->right;

e->right->parent = p;

}

else {

if (p->left == e) p->left = e->left;

else p->right = e->left;

e->left->parent = p;

}

}

else {

Tree\* s = e->next();

// Следующий элемент за удаляемым

e->data = s->data;

if (s->parent->left == s) {

s->parent->left = s->right;

if (s->right != NULL)

s->right->parent = s->parent;

}

else {

s->parent->right = s->right;

if (s->right != NULL)

s->right->parent = s->parent;

}

}

}

Tree\* Tree::findElement(double dt)

{

if ((this == NULL) || (dt == this->data))

return this;

if (dt < this->data) return this->left->findElement(dt);

else return this->right->findElement(dt);

}

double Tree::findMax()

{

if (this == NULL) return NULL;

if (this->right == NULL) return this->data;

return this->right->findMax();

}

Tree\* Tree::findMin()

{

if (this->left == NULL) return this;

return this->left->findMin();

}

void Tree::balancedSortTree(int n) {

if (n == 0) return ;

vector<double> vec;

double dt;

for (int l = 0; l < n; l++)

{

cout << "data=";

cin >> dt;

vec.push\_back(dt);

}

sort(vec.begin(), vec.end());

balancedTree(vec);

}

void Tree::balancedTree(vector<double>& vec)

{

this->insertNode(vec[vec.size() / 2]);

if (vec.size() == 2)

{

this->insertNode(vec[0]);

return;

}

if (vec.size() == 1)

{

return;

}

vector<double> left;

vector<double> right;

int mid = vec.size() / 2;

for (int i = 0; i < mid; i++)

{

left.push\_back(vec[i]);

}

for (int i = mid + 1; i < vec.size(); i++)

{

right.push\_back(vec[i]);

}

balancedTree(left);

balancedTree(right);

}

Main.cpp

#include "Tree.h"

#include <iostream>

#include <Windows.h>

#include <gl/GL.h>

#include <gl/GLU.h>

#include <glut.h>

#include <string>

using namespace std;

Tree\* bt = new Tree();

int MAX\_LEVEL = 0;

int LEVEL\_HEIGHT = 0;

float WIDTH\_EL = 0.0f;

const int NODE\_RADIUS = 20;

const int WINDOW\_WIDTH = 800;

const int WINDOW\_HEIGHT = 600;

float getY(int level = 1)

{

return (MAX\_LEVEL - level + 1) \* LEVEL\_HEIGHT;

}

void drawNode(float cx, float cy, double value, int num\_segments = 20) {

float th = 2.0f \* 3.1415926f / float(num\_segments);

// Set new viewport

glViewport(cx - NODE\_RADIUS, cy - NODE\_RADIUS, NODE\_RADIUS \* 2, NODE\_RADIUS \* 2);

// Draw circle

glColor3ub(255, 255, 255);

glBegin(GL\_TRIANGLE\_FAN);

for (int ii = 0; ii < num\_segments; ii++) {

glVertex2f(cosf(th \* ii), sinf(th \* ii));

}

glEnd();

// Prepare text 0,1230000 => 0,123

string text = to\_string(value);

int foundedIndex = text.length();

for (int i = text.length() - 1; i >= 0; i--) {

if (text[i] == (char)'0') {

foundedIndex = i;

}

else {

continue;

}

}

if (text[foundedIndex - 1] == (char)',') {

foundedIndex++;

}

text = text.substr(0, foundedIndex);

// Draw Value

glColor3ub(11, 65, 179);

float xPos = -0.25 \* (text.length() - 0.5);

glRasterPos2f(xPos, -0.31);

for (const char\* p = text.c\_str(); \*p != '\0'; p++) {

glutBitmapCharacter(GLUT\_BITMAP\_HELVETICA\_18, \*p);

}

// Return viewport

glViewport(0, 0, WINDOW\_WIDTH, WINDOW\_HEIGHT);

}

void drawLine(float sx, float sy, float ex, float ey) {

float x1 = sx < ex ? -1 : 1;

float y1 = sy > ey ? 1 : -1;

float x2 = sx < ex ? 1 : -1;

float y2 = sy > ey ? -1 : 1;

glViewport(min(sx, ex), min(sy, ey), abs(sx - ex) + 1, abs(sy - ey) + 1);

glLineWidth(4);

glColor3ub(100, 100, 100);

glBegin(GL\_LINES);

glVertex2f(x1, y1);

glVertex2f(x2, y2);

glEnd();

glViewport(0, 0, WINDOW\_WIDTH, WINDOW\_HEIGHT);

}

void displayNodeIter(Tree& inner, float x, float y, int level = 1) {

float k = WIDTH\_EL \* pow(2, MAX\_LEVEL - level - 1);

float nextY = getY(level + 1);

if (inner.left != NULL) {

drawLine(x, y, x - k, nextY);

displayNodeIter(\*inner.left, x - k, nextY, level + 1);

}

if (inner.right != NULL) {

drawLine(x, y, x + k, nextY);

displayNodeIter(\*inner.right, x + k, nextY, level + 1);

}

drawNode(x, y, inner.data);

}

void display() {

glClear(GL\_COLOR\_BUFFER\_BIT); // очищение

float y = getY(1);

displayNodeIter(\*bt, WINDOW\_WIDTH / 2, y);

glFlush();

}

void drawTreeGL(int argc, char\*\* argv) {

MAX\_LEVEL = bt->watchDeep(0);

WIDTH\_EL = round(WINDOW\_WIDTH / (pow(2, MAX\_LEVEL - 1) + 1) / 2);

LEVEL\_HEIGHT = round(WINDOW\_HEIGHT / (MAX\_LEVEL + 1));

glutInit(&argc, argv);

glutInitWindowSize(WINDOW\_WIDTH, WINDOW\_HEIGHT); //создание окна

glutInitWindowPosition(80, 80); // указание позиции окна

glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);

glutCreateWindow("Tree");

glutDisplayFunc(display);

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

glutMainLoop();

}

int main(int argc, char\*\* argv)

{

system("color F0");

setlocale(LC\_ALL, "rus");

cout << "n=";

int n;

cin >> n;

Tree tree;

bt->balancedSortTree(n);

bt->printVTree();

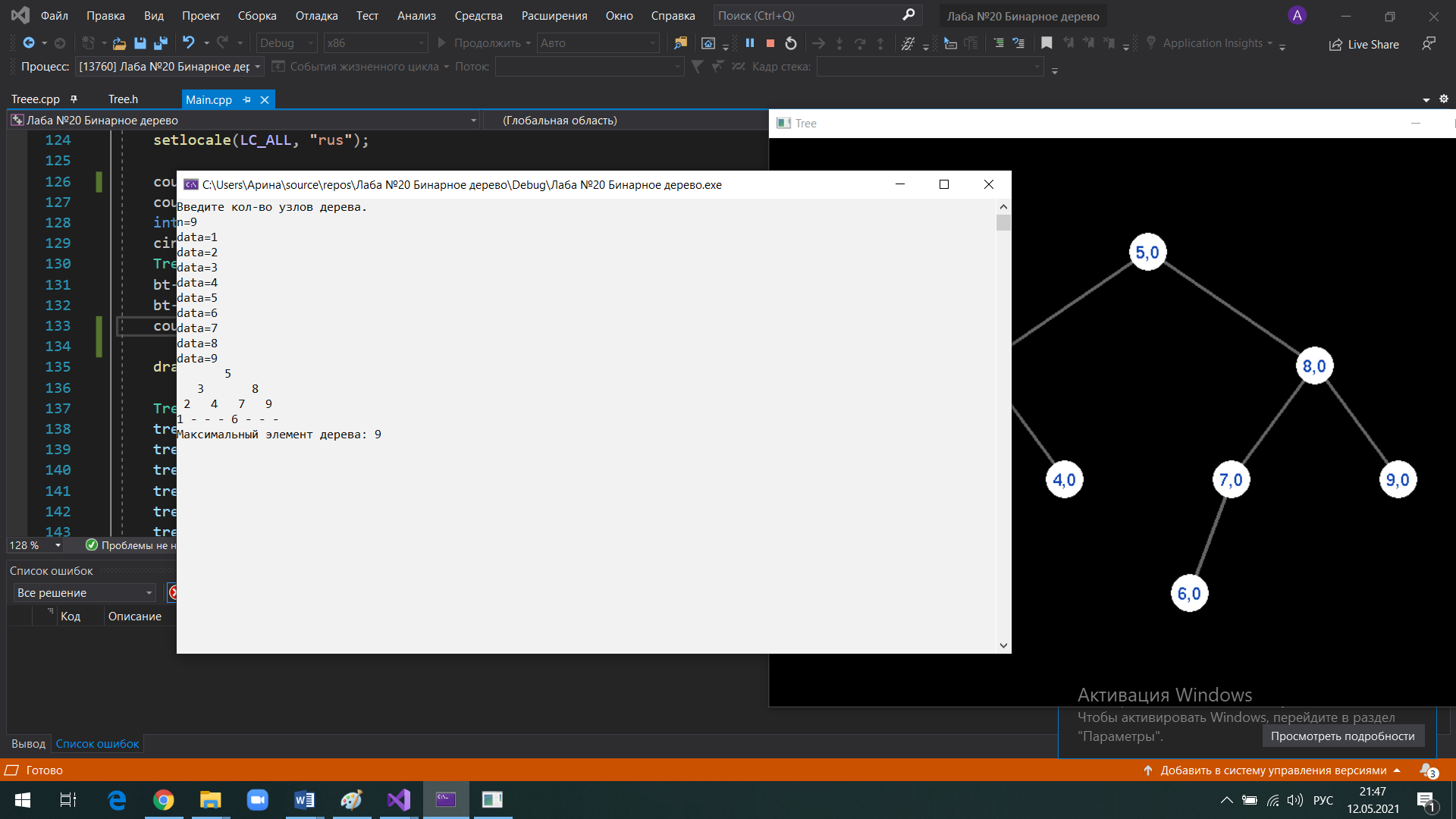
cout << bt->findMax() << endl;

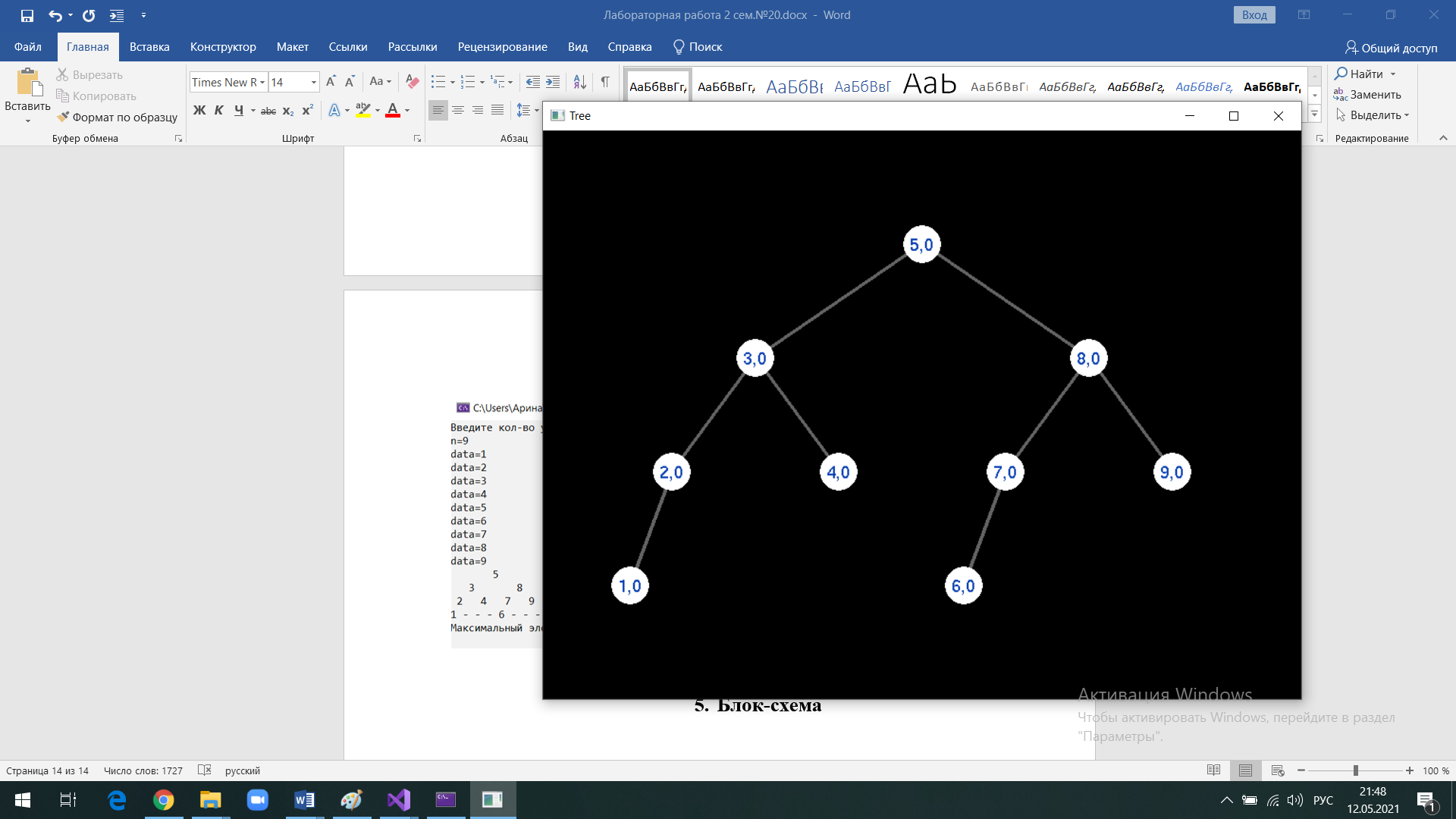
drawTreeGL(argc, argv);

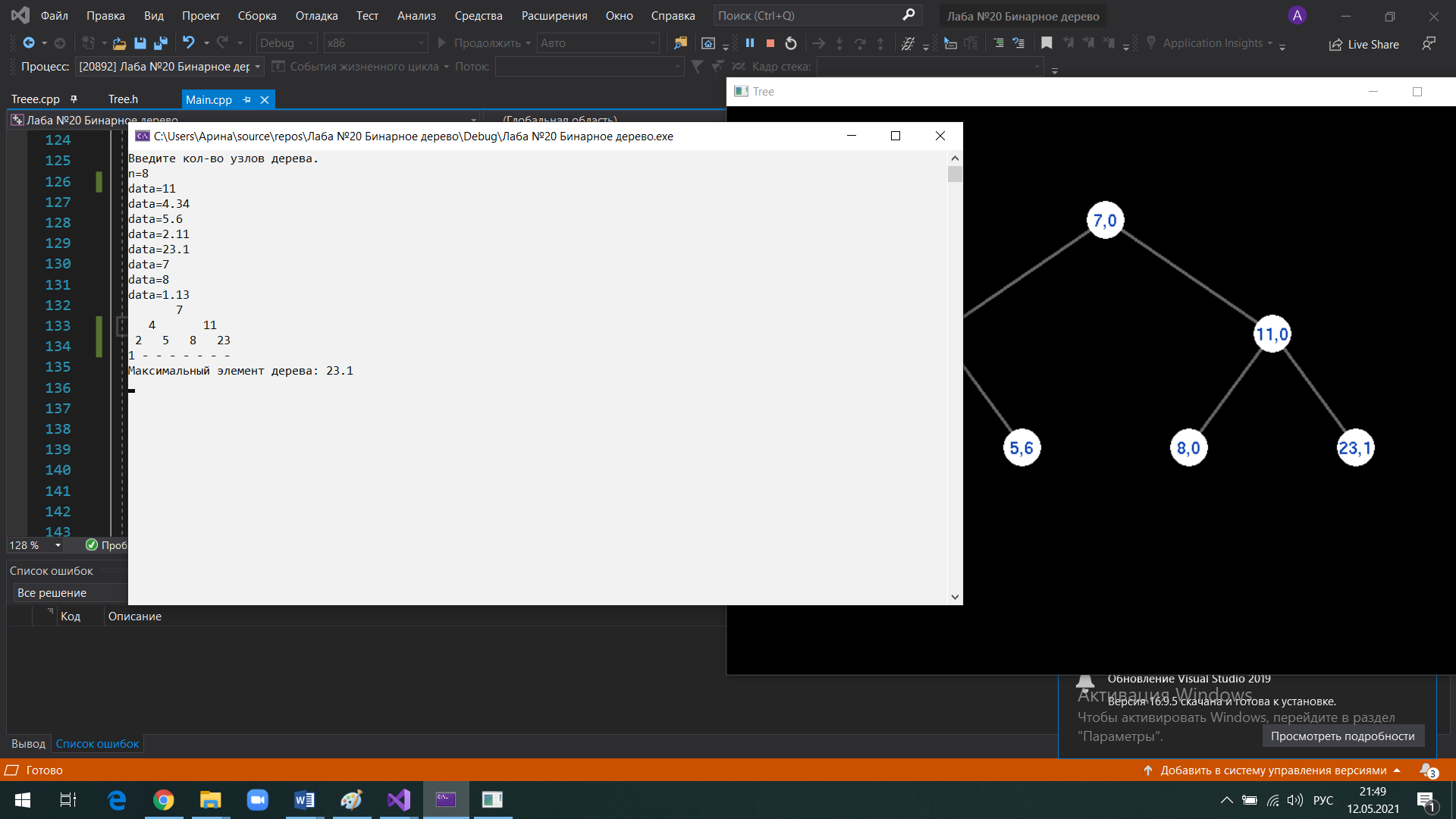
return 0;

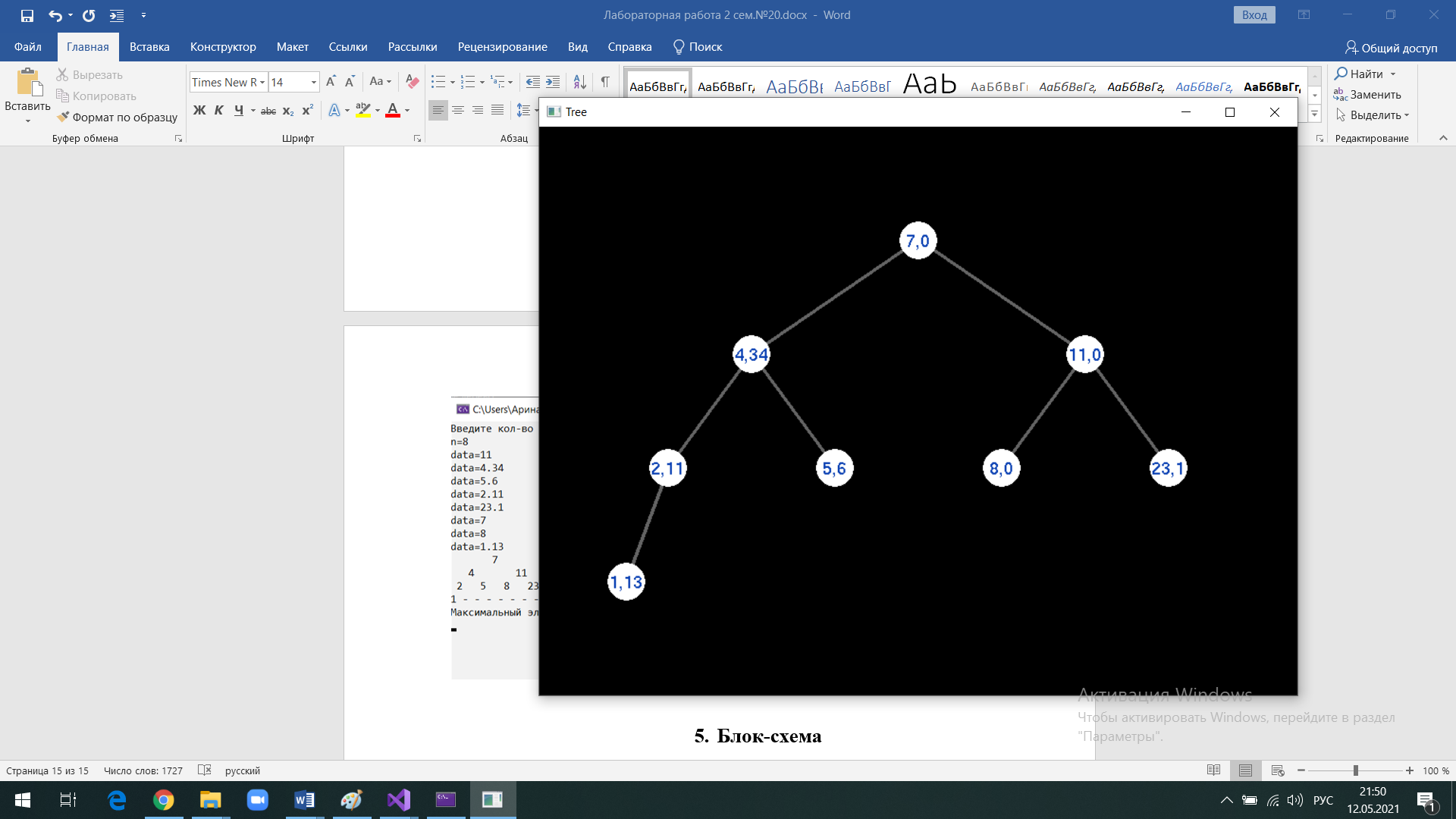
}

1. **Выполнение программы**









1. **Блок-схема**

