Министерство науки и высшего образования Российской Федерации Федеральное государственное автономное учреждение высшего образования «Пермский национальный исследовательский политехнический университет»

ПНИПУ

**Лабораторная работа  
Бинарные деревья**

Выполнил:   
студент группы ИВТ-23-2б   
Чудинов Данил Николаевич

Проверила:   
доцент кафедры ИТАС   
О.А. Полякова

Пермь, 2024 г.

**«<Бинарные деревья»**

**Задача: 1.**

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 22:***

Тип информационного поля char. Найти количество элементов с заданным

ключом.

**Анализ задачи:**

insert(T data): Эта функция вставляет новый узел со значением data в нужное место в дереве, с учетом порядка значений. Она выполняет поиск места для вставки нового узла и вызывает функции insert\_right и insert\_left для добавления нового узла в правое или левое поддерево соответственно.

insert\_right(T data): Эта функция добавляет новый узел со значением data в правое поддерево текущего узла.

insert\_left(T data): Эта функция добавляет новый узел со значением data в левое поддерево текущего узла.

delete\_tree(): Эта функция удаляет полностью дерево, но имеет ошибку в реализации. Удаление должно быть выполнено правильно для всех узлов дерева.

get\_data(), get\_right(), get\_left(), get\_parent(): Эти функции возвращают значение в узле, указатель на правое поддерево, указатель на левое поддерево и указатель на родительский узел соответственно.

erase(T data): Эта функция удаляет узел с заданным значением, ищет узел по значению, а затем выполняет операцию удаления в зависимости от типа удаляемого узла (узел без потомков, узел с одним потомком, узел с двумя потомками).

delete\_left(), delete\_right(): Эти функции удаляют левое и правое поддерево текущего узла соответственно.

add\_right(Tree<T> \*temp), add\_left(Tree<T> \*temp): Эти функции устанавливают правое и левое поддерево текущего узла соответственно.

search(T key): Эта функция ищет узел по заданному ключу и возвращает его.

find(T data): Эта функция ищет узел с заданным значением в дереве и возвращает его.

direct\_way(Tree<T> \*current), symmetric\_way(Tree<T> \*tree), reverse\_way(Tree<T> \*tree): Эти функции выполняют прямой, симметричный и обратный обход дерева соответственно.

balanced(int count): Эта функция создает сбалансированное дерево с заданной высотой, запрашивая данные от пользователя истроит дерево рекурсивно.

getHeight(), getAmountOfNodes(): Эти функции возвращают высоту дерева и количество узлов в дереве соответственно.

obh(Tree<T> \*node), printVert(): Эти функции печатают дерево вертикально и используют вспомогательный файл print.txt для хранения данных об узлах.

print\_horizontal(int depth = 0, char branch = ' '): Эта функция печатает дерево горизонтально, обозначая ветви узлов.

build\_bst(const vector<T> data, int start, int end): Эта функция рекурсивно строит сбалансированное дерево по заданным данным вектора.

**Код на языке С++:**

#include <iostream>

#include <string>

#include <ctime>

#include <list>

#include <fstream>

#include <queue>

#include <algorithm>

using namespace std;

template <typename T>

class Tree {

private:

Tree<T>\* left;

Tree<T>\* right;

Tree<T>\* parent;

T data;

public:

Tree<T>() {//Конструктор без значений

left = right = parent = nullptr;

}

Tree<T>(T data) {//Конструктор со значением

this->data = data;

left = right = parent = nullptr;

}

~Tree<T>() {//Деструктор

delete\_right();

delete\_left();

delete\_tree();

}

void insert(T data) {//Вставляет новый узел со значением data в нужное место в дереве

Tree<T>\* current = this;

while (current != nullptr) {

if (data > current->data) {

if (current->right != nullptr) {

current = current->right;

}

else {

current->insert.right(data);

return;

}

}

else if (data < current->data) {

if (current->left != nullptr) {

current = current->left;

}

else {

current->insert.left(data);

return;

}

}

else return;

}

}

void insert\_right(T data) {//Вставляет новый узел со значением data в правое поддерево

Tree<T>\* new\_node = new Tree(data);

if (this->right != nullptr) {

this->right->parent = new\_node;

new\_node->right = this->right;

}

this->right = new\_node;

new\_node->parent = this;

}

void insert\_left(T data) {//Вставляет новый узел со значением data в левое поддерево

left = new Tree<T>(data);

left->parent = this;

}

void delete\_tree(){//Удаляет полностью дерево

delete this;

}

T get\_data() {//Возвращает значение в data

return this->data;

}

Tree<T>\* get\_right() {//Возвращает указатель на правое поддерево

return this->right;

}

Tree<T>\* get\_left() {//Возвращает указатель на левое поддерево

return this->left;

}

Tree<T>\* get\_parent() {//Возвращает указатель на родительский узел

return this->parent;

}

void erase(T data) {//Функция, которая удаляет узел с заданным значением

Tree<T>\* to\_erase = this->find(data);

Tree<T>\* to\_parent = to\_erase->parent;

if (to\_erase->left == nullptr && to\_erase->right == nullptr) {

if (to\_parent->left == nullptr) {

to\_parent->left == nullptr;

delete to\_erase;

}

else {

to\_parent->right == nullptr;

delete to\_erase;

}

}

else if ((to\_erase->left != nullptr && to\_erase->right != nullptr) || ((to\_erase->left == nullptr && to\_erase->right != nullptr))) {

if (to\_erase->left==nullptr) {

if (to\_erase == to\_parent->left) {

to\_parent->left = to\_erase->right;

}

else {

to\_parent->righjt == to\_erase->right;

}

to\_erase->right->parent = to\_parent;

}

else {

if (to\_parent->left == to\_erase) {

to\_parent->left = to\_erase->left;

}

else {

to\_parent->right = to\_erase->left;

}

to\_erase->left->parent = to\_parent;

}

}

else {

Tree<T>\* next = to\_erase->next();

to\_erase->data = next->data;

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

next->parent->left = next->right;

if (next->right != nullptr) {

next->right->parent = next->parent;

}

}

else {

next->parent->right = next->right;

if (next->right != nullptr) {

next->right->parent = next->parent;

}

}

delete next;

}

}

void delete\_left() {//Удаляет левое поддерево

if (left != NULL) {

left->delete\_left();

left->delete\_right();

delete left;

}

}

void delete\_right() {//Удаляет правое поддерево

if (right != NULL) {

right->delete\_right;

right->delete\_left();

delete right;

}

}

void add\_right(Tree<T>\* temp) {//Функция, которая устанавливает правое поддерево

right = temp;

}

void add\_left (Tree<T>\*temp){//Функция, которая устанавливает левое поддерево

left = temp;

}

Tree<T> search(T key) {//ищет узел по заданному ключу

if (data == key) {

return this;

}

if (left != nullptr) {

Tree<T>\* result = left->search(key);

if (result != nullptr) {

return result;

}

}

if (right != nullptr) {

Tree<T>\* result = right->search(key);

if (result != nullptr) {

return result;

}

}

return nullptr;

}

Tree<T> find(T data) {//Функция, которая ищет узел с исходным значением

if (this == nullptr || this->data == data) {

return this;

}

else if (data > this->data) {

return this->right->find(data);

}

else {

return this->left->find(data);

}

}

void direct\_way(Tree<T>\* current) {//Прямой обход дерева

if (current == nullptr) {

return;

}

else {

cout << current->get\_data() << " ";

direct\_way(current->get\_left());

direct\_way(current->get\_right());

}

}

void symmetric\_way(Tree<T>\* tree) {//Симметричный обход дерева

if (tree !=nullptr) {

symmetric\_way(tree->left);

cout << tree->data << " ";

symmetric\_way(tree->right);

}

}

void reverse\_way(Tree<T>\* tree) {//Обратный обход дерева

if (tree != nullptr) {

reverse\_way(tree->left);

reverse\_way(tree->right);

cout << tree->data << " ";

}

}

Tree<T>\* balanced(int count) {//Функция, которая создаёт сбалансированное дерево с нужной высотой

if (count <= 0) {

return nullptr;

}

T data;

cout << "Введите данные для сбалансированного дерева: ";

cin >> data;

Tree<T>\* temp = new Tree<T>(data);

temp->add\_left(balanced(count / 2));

temp->add\_right(balanced(count - count / 2 - 1));

return temp;

}

int getHeight() {//Функция, которая возвращает высоту дерева

int h1 = 0, h2 = 0, hadd = 0;

if (this == NULL) {

return 0;

}

if (this->left != NULL) {

h1 = this->left->getHeight();

}

if (this->right != NULL) {

h2 = this->right->getHeight();

}

if (h1 >= h2) {

return h1 + 1;

}

else return h2 + 1;

}

int getAmountOfNodes() {//Функция, которая возвращает кол-во узлов в дереве

if (this == NULL) {

return 0;

}

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

return 1;

}

int l = 0;

int r = 0;

if (this->left != NULL) {

l = this->left->getAmountOfNodes();

}

if (this->right != NULL) {

r = this->right->getAmountOfNodes();

}

return (l + r + 1);

}

void obh(Tree<T>\* node) { //Дополнительная функция для вертикальной печати дерева

ofstream f("print.txt");

int amount = node->getAmountOfNodes();

queue<Tree<T>\*>q;

q.push(node);

while (!q.empty()) {

Tree<T>\* temp = q.front();

q.pop();

f << temp->data << endl;

if (temp->left) {

q.push(temp->left);

}

if (temp->right) {

q.push(temp->right);

}

}

f.close();

}

void printVert() { //Вертикальная печать дерева

obh(this);

ifstream f("print.txt");

int height = this->getHeight();

int count = 0;

int\* spaces = new int[height];

spaces[0] = 0;

for (int i = 1; i < height; i++) {

spaces[i] = spaces[i - 1] \* 2 + 1;

}

char str[255];

for (int i = 0, l = height - 1; i < height; i++, l--) {

for (int j = 0; j < pow(2, i); j++) {

if (j == 0) {

for (int u = 0; u < spaces[l]; u++) {

cout << " ";

}

}

else {

for (int u = 0; u < spaces[l + 1]; u++) {

cout << " ";

}

}

if (f.getline(str, 255)) {

cout << str;

}

else {

cout << " ";

}

}

cout << endl;

}

delete[] spaces;

f.close();

}

void print\_horizontal (int depht = 0, char branch = ' ') { //Горизонтальная печать дерева

if (right != nullptr) {

right->print\_horizontal(depht + 1, '/');

}

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

cout << " ";

}

cout << branch << "--" << data << endl;

if (left != nullptr) {

left->print\_horizontal(depht + 1, '\\');

}

}

static Tree<T>\* build\_bst(const vector<T> data , int start, int end) {//Рекурсивная функции для постройки сбалансированного дерева

if (start > end) {

return nullptr;

}

int mid = start + (end - start) / 2;

Tree<T>\* new\_node = new Tree<T>(data[mid]);

new\_node->left = build\_bst(data, start, mid - 1);

new\_node->right = build\_bst(data, mid + 1, end);

return new\_node;

}

void in\_order\_traversal(vector<T>& result) {//Обход дерева в порядке возрастания и сохранение значений в векторе

if (left != nullptr) {

left->in\_order\_traversal(result);

}

result.push\_back(data);

if (right != nullptr) {

right->in\_order\_traversal(result);

}

}

static Tree<T>\* create\_bst(Tree<T>\* root) {//Фунция для создания дерева поиска из сбалансированного дерева

vector<T> sorted\_data;

root->in\_order\_traversal(sorted\_data);

return build\_bst(sorted\_data, 0, sorted\_data.size() - 1);

}

};

int main() {

system("chcp 1251>NULL");

Tree<char>\* root = new Tree<char>('a');

root->insert\_left('b');

root->insert\_right('c');

root->get\_left()->insert\_left('d');

root->get\_left()->insert\_right('e');

root->get\_right()->insert\_left('f');

root->get\_right()->insert\_right('g');

cout << "Горизонтальный вывод дерева: " << endl;

root->print\_horizontal();

cout << endl << endl;

cout << "Вертикальный вывод дерева: " << endl;

root->printVert();

cout << endl << endl;

cout << "Прямой обход: " << endl;

root->direct\_way(root);

cout << endl;

cout << "Симметричный обход: " << endl;

root->symmetric\_way(root);

cout << endl;

cout << "Обратный обход: " << endl;

root->reverse\_way(root);

cout << endl << endl;

Tree<char>\* bal = new Tree<char>('a');

int count;

cout << "Введите кол-во элементов в сбалансированном дереве: ";

cin >> count;

Tree<char>\* bal1 = bal->balanced(count);

cout << endl;

cout << "Горизонтальный вывод сбалансированного дерева: " << endl;

bal1->print\_horizontal();

cout << endl << endl;

cout << "Вертикальный вывод сбалансированного дерева: " << endl;

bal1->printVert();

cout << endl << endl;

cout << "Преобразуем дерево, в дерево поиска: " << endl;

bal1->create\_bst(bal1);

cout << "Горизонтальный вывод дерева поиска: " << endl;

bal1->print\_horizontal();

cout << endl << endl;

cout << "Введите символ для задания: ";

char s;

cin >> s;

int l = 0;

queue<Tree<char>\*>q;

q.push(bal1);

while (!q.empty()) {

Tree<char>\* current = q.front();

q.pop();

if (current->get\_data() == s) {

l++;

}

if (current->get\_left() != nullptr) {

q.push(current->get\_left());

}

if (current->get\_right() != nullptr) {

q.push(current->get\_right());

}

}

cout << "Количество элементов с заданным ключом " << s << " : " << l << endl;

return 0;

}

OpenGL:

#include<GL/glut.h>

#include<stdio.h>

#define \_USE\_MATH\_DEFINES

#include <Windows.h>

#include <iostream>

#include <string>

#include <ctime>

#include <list>

#include <fstream>

#include <queue>

#include <algorithm>

using namespace std;

template <typename T>

class Tree {

private:

Tree<T>\* left;

Tree<T>\* right;

Tree<T>\* parent;

T data;

public:

GLfloat x = 0, y = 3;

int state, level = 1;

Tree<T>() {//Конструктор без значений

left = right = parent = nullptr;

}

Tree<T>(T data) {//Конструктор со значением

this->data = data;

left = right = parent = nullptr;

}

~Tree<T>() {//Деструктор

delete\_right();

delete\_left();

delete\_tree();

}

void insert(T data) {//Вставляет новый узел со значением data в нужное место в дереве

Tree<T>\* current = this;

while (current != nullptr) {

if (data > current->data) {

if (current->right != nullptr) {

current = current->right;

}

else {

current->insert.right(data);

return;

}

}

else if (data < current->data) {

if (current->left != nullptr) {

current = current->left;

}

else {

current->insert.left(data);

return;

}

}

else return;

}

}

void insert\_right(T data) {//Вставляет новый узел со значением data в правое поддерево

Tree<T>\* new\_node = new Tree(data);

if (this->right != nullptr) {

this->right->parent = new\_node;

new\_node->right = this->right;

}

this->right = new\_node;

new\_node->parent = this;

}

void insert\_left(T data) {//Вставляет новый узел со значением data в левое поддерево

left = new Tree<T>(data);

left->parent = this;

}

void delete\_tree() {//Удаляет полностью дерево

delete this;

}

T get\_data() {//Возвращает значение в data

return this->data;

}

Tree<T>\* get\_right() {//Возвращает указатель на правое поддерево

return this->right;

}

Tree<T>\* get\_left() {//Возвращает указатель на левое поддерево

return this->left;

}

Tree<T>\* get\_parent() {//Возвращает указатель на родительский узел

return this->parent;

}

void erase(T data) {//Функция, которая удаляет узел с заданным значением

Tree<T>\* to\_erase = this->find(data);

Tree<T>\* to\_parent = to\_erase->parent;

if (to\_erase->left == nullptr && to\_erase->right == nullptr) {

if (to\_parent->left == nullptr) {

to\_parent->left == nullptr;

delete to\_erase;

}

else {

to\_parent->right == nullptr;

delete to\_erase;

}

}

else if ((to\_erase->left != nullptr && to\_erase->right != nullptr) || ((to\_erase->left == nullptr && to\_erase->right != nullptr))) {

if (to\_erase->left == nullptr) {

if (to\_erase == to\_parent->left) {

to\_parent->left = to\_erase->right;

}

else {

to\_parent->righjt == to\_erase->right;

}

to\_erase->right->parent = to\_parent;

}

else {

if (to\_parent->left == to\_erase) {

to\_parent->left = to\_erase->left;

}

else {

to\_parent->right = to\_erase->left;

}

to\_erase->left->parent = to\_parent;

}

}

else {

Tree<T>\* next = to\_erase->next();

to\_erase->data = next->data;

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

next->parent->left = next->right;

if (next->right != nullptr) {

next->right->parent = next->parent;

}

}

else {

next->parent->right = next->right;

if (next->right != nullptr) {

next->right->parent = next->parent;

}

}

delete next;

}

}

void delete\_left() {//Удаляет левое поддерево

if (left != NULL) {

left->delete\_left();

left->delete\_right();

delete left;

}

}

void delete\_right() {//Удаляет правое поддерево

if (right != NULL) {

right->delete\_right();

right->delete\_left();

delete right;

}

}

void add\_right(Tree<T>\* temp) {//Функция, которая устанавливает правое поддерево

right = temp;

}

void add\_left(Tree<T>\* temp) {//Функция, которая устанавливает левое поддерево

left = temp;

}

Tree<T> search(T key) {//ищет узел по заданному ключу

if (data == key) {

return this;

}

if (left != nullptr) {

Tree<T>\* result = left->search(key);

if (result != nullptr) {

return result;

}

}

if (right != nullptr) {

Tree<T>\* result = right->search(key);

if (result != nullptr) {

return result;

}

}

return nullptr;

}

Tree<T> find(T data) {//Функция, которая ищет узел с исходным значением

if (this == nullptr || this->data == data) {

return this;

}

else if (data > this->data) {

return this->right->find(data);

}

else {

return this->left->find(data);

}

}

void direct\_way(Tree<T>\* current) {//Прямой обход дерева

if (current == nullptr) {

return;

}

else {

cout << current->get\_data() << " ";

direct\_way(current->get\_left());

direct\_way(current->get\_right());

}

}

void symmetric\_way(Tree<T>\* tree) {//Симметричный обход дерева

if (tree != nullptr) {

symmetric\_way(tree->left);

cout << tree->data << " ";

symmetric\_way(tree->right);

}

}

void reverse\_way(Tree<T>\* tree) {//Обратный обход дерева

if (tree != nullptr) {

reverse\_way(tree->left);

reverse\_way(tree->right);

cout << tree->data << " ";

}

}

Tree<T>\* balanced(int count) {//Функция, которая создаёт сбалансированное дерево с нужной высотой

if (count <= 0) {

return nullptr;

}

T data;

cout << "Введите данные для сбалансированного дерева: ";

cin >> data;

Tree<T>\* temp = new Tree<T>(data);

temp->add\_left(balanced(count / 2));

temp->add\_right(balanced(count - count / 2 - 1));

return temp;

}

int getHeight() {//Функция, которая возвращает высоту дерева

int h1 = 0, h2 = 0, hadd = 0;

if (this == NULL) {

return 0;

}

if (this->left != NULL) {

h1 = this->left->getHeight();

}

if (this->right != NULL) {

h2 = this->right->getHeight();

}

if (h1 >= h2) {

return h1 + 1;

}

else return h2 + 1;

}

int getAmountOfNodes() {//Функция, которая возвращает кол-во узлов в дереве

if (this == NULL) {

return 0;

}

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

return 1;

}

int l = 0;

int r = 0;

if (this->left != NULL) {

l = this->left->getAmountOfNodes();

}

if (this->right != NULL) {

r = this->right->getAmountOfNodes();

}

return (l + r + 1);

}

static Tree<T>\* build\_bst(const vector<T> data, int start, int end) {//Рекурсивная функции для постройки сбалансированного дерева

if (start > end) {

return nullptr;

}

int mid = start + (end - start) / 2;

Tree<T>\* new\_node = new Tree<T>(data[mid]);

new\_node->left = build\_bst(data, start, mid - 1);

new\_node->right = build\_bst(data, mid + 1, end);

return new\_node;

}

void in\_order\_traversal(vector<T>& result) {//Обход дерева в порядке возрастания и сохранение значений в векторе

if (left != nullptr) {

left->in\_order\_traversal(result);

}

result.push\_back(data);

if (right != nullptr) {

right->in\_order\_traversal(result);

}

}

static Tree<T>\* create\_bst(Tree<T>\* root) {//Фунция для создания дерева поиска из сбалансированного дерева

vector<T> sorted\_data;

root->in\_order\_traversal(sorted\_data);

return build\_bst(sorted\_data, 0, sorted\_data.size() - 1);

}

friend void reshape(int height, int width);

friend void display();

void drawTree(int argc, char\*\* argv, int win\_height, int win\_width);

friend void LevelCounter(Tree\* root);

friend void CountLevels(Tree\* root, void(\*LevelConter)(Tree\* root));

friend void Coords(Tree\* node);

friend void CoordsCalculate(Tree\* node, void(\*Coords)(Tree\* node));

friend void DrawOneNode(Tree\* root);

friend void DrawNodes(Tree\* root, void (\*DrawOneNode)(Tree\* root));

friend void DrawOneLine(Tree\* root);

friend void DrawLines(Tree\* root, void (\*DrawOneLine)(Tree\* root));

};

int depth = 0, width = 0;

Tree<char>\* tree = new Tree<char>('a');

float RadiusA = 0.35;

void reshape(int height, int width);

void display();

void Tree<char>::drawTree(int argc, char\*\* argv, int win\_height, int win\_width) {

glutInit(&argc, argv);

glutInitWindowPosition(0, 0);

glutInitWindowSize(win\_height, win\_width);

glutInitDisplayMode(GLUT\_RGB | GLUT\_DOUBLE);

glutCreateWindow("Tree");

glutReshapeFunc(reshape);

glutDisplayFunc(display);

glutMainLoop();

}

void LevelCounter(Tree<char>\* root) {

if (root->parent != NULL) {

root->level = root->parent->level + 1;

}

if (depth < root->level) {

depth = root->level;

}

}

void CountLevels(Tree<char>\* root, void(\*LevelConter)(Tree<char>\* root)) {

if (root == NULL) {

return;

}

(LevelCounter)(root);

CountLevels(root->left, LevelConter);

CountLevels(root->right, LevelConter);

}

void Coords(Tree<char>\* node) {

if (node->parent != NULL) {

if (node->level == 2) {

node->x = node->parent->x + node->state \* (pow(2, depth - 1) / 2);

}

else

node->x = node->parent->x + node->state \* (pow(2, depth - 1) / pow(2, node->level - 1));

node->y = node->parent->y - 1;

}

}

void CoordsCalculate(Tree<char>\* node, void(\*Coords)(Tree<char>\* node)) {

if (node == NULL) {

return;

}

(\*Coords)(node);

if (node->left != NULL) {

node->left->state = -1;

CoordsCalculate(node->left, Coords);

}

if (node->right != NULL) {

node->right->state = 1;

CoordsCalculate(node->right, Coords);

}

return;

}

void DrawCircle(char colour, GLfloat x, GLfloat y, float radiusB, int count) {

glColor3f(0.0, 250.0, 0.0);

glBegin(GL\_TRIANGLE\_FAN);

glVertex2f(x, y);

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

glVertex2f(

(x + (RadiusA \* cos(i \* 2 \* M\_PI / count))),

(y + (radiusB \* sin(i \* 2 \* M\_PI / count)))

);

}

glEnd();

}

void DrawOutline(float tmp\_x, float tmp\_y, float radiusB) {

glColor3f(0.0, 250.0, 0.0);

glBegin(GL\_POINTS);

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

for (int j = 0; j <= 540; j++) {

tmp\_x = RadiusA \* sin(j) + tmp\_x;

tmp\_y = radiusB \* cos(j) + tmp\_y;

glVertex2f(tmp\_x - 0.35, tmp\_y - 0.1);

}

}

glEnd();

}

void drawNode(const char\* str, GLfloat x, GLfloat y, char colour) {

double c = 0;

c = (4 + depth) / pow(2, depth);

float radiusB = c \* RadiusA;

int count = 50;

DrawCircle('g', x, y, radiusB, RadiusA);

DrawOutline(x, y, radiusB);

glColor3f(0.0, 0.0, 0.0);

glRasterPos2f(x - 0.05, y - 0.05);

const char\* p;

for (p = str; \*p != '\0'; p++) {

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

}

}

void DrawOneLine(Tree<char>\* root) {

if (root->parent != NULL) {

glBegin(GL\_LINES);

glVertex2d(root->parent->x, root->parent->y);

glVertex2d(root->x, root->y);

glEnd();

}

}

void DrawLines(Tree<char>\* root, void (\*DrawOneLine)(Tree<char>\* root)) {

if (root == NULL) {

return;

}

(\*DrawOneLine)(root);

DrawLines(root->left, DrawOneLine);

DrawLines(root->right, DrawOneLine);

}

void DrawOneNode(Tree<char>\* root) {

char colour;

if (root->parent != NULL) {

colour = 'g';

drawNode(to\_string(root->data).c\_str(), root->x, root->y, colour);

}

}

void DrawNodes(Tree<char>\* root, void (\*DrawOneNode)(Tree<char>\* root)) {

char colour;

if (root == NULL) {

return;

}

colour = 'g';

(\*DrawOneNode)(root);

DrawNodes(root->left, DrawOneNode);

DrawNodes(root->right, DrawOneNode);

drawNode(to\_string(tree->data).c\_str(), tree->x, tree->y, colour);

}

void reshape(int height, int width) {

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

glViewport(0, 0, height, width);

gluOrtho2D(-pow(2, depth - 1), pow(2, depth - 1), -depth, 5);

}

void display() {

glClearColor(1, 1, 1, 1);

glClear(GL\_COLOR\_BUFFER\_BIT);

glColor3f(0.0, 0.0, 250.0);

glLineWidth(1);

DrawLines(tree, DrawOneLine);

DrawNodes(tree, DrawOneNode);

glutSwapBuffers();

}

void PrintingInfo() {

depth = tree->getHeight();

width = pow(2, depth - 1);

cout << "Глубина дерева: " << depth << endl;

cout << "Ширина дерева: " << width << endl;

}

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

system("chcp 1251>NULL");

tree->insert\_left('b');

tree->insert\_right('c');

tree->get\_left()->insert\_left('d');

tree->get\_left()->insert\_right('e');

tree->get\_right()->insert\_left('f');

tree->get\_right()->insert\_right('g');

PrintingInfo();

CountLevels(tree, LevelCounter);

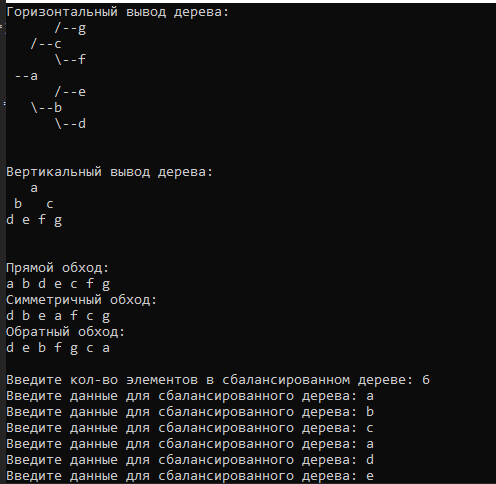
CoordsCalculate(tree, Coords);

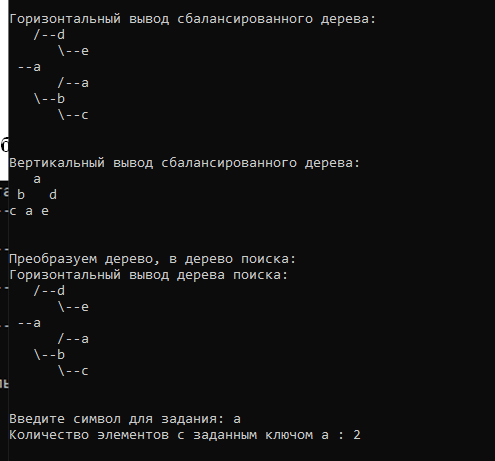
tree->drawTree(argc, argv, 960, 720);

return 0;

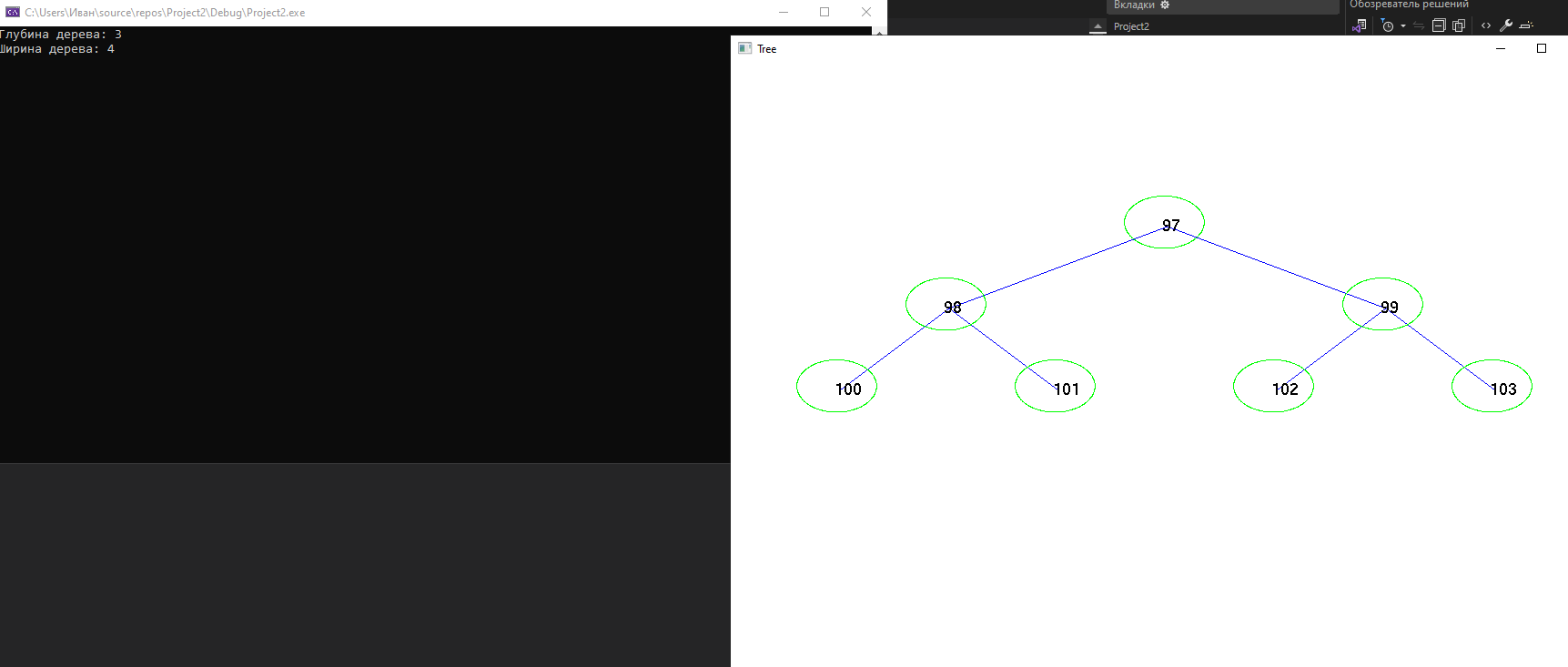
}

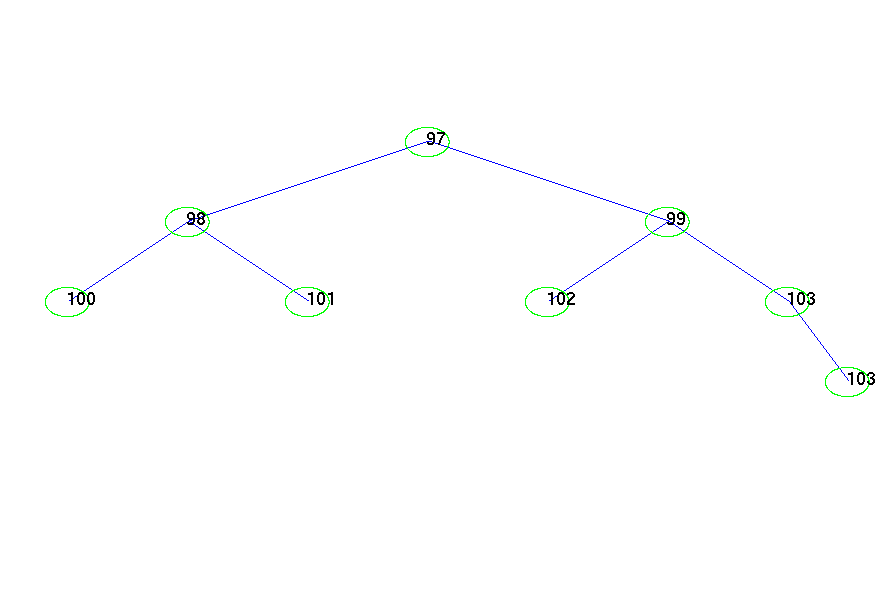
**Работа программы:**

****

****

**OpenGL:**

****

****

UML:

