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

ПНИПУ

Лабораторная работа
«Бинарные деревья»

Выполнила:
студентка группы ИВТ-23-2б
Соловьева Екатерина Александровна

Проверила:
доцент кафедры ИТАС
О.А. Полякова

Пермь, 2024 г.

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 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 C++:

```
#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");
```

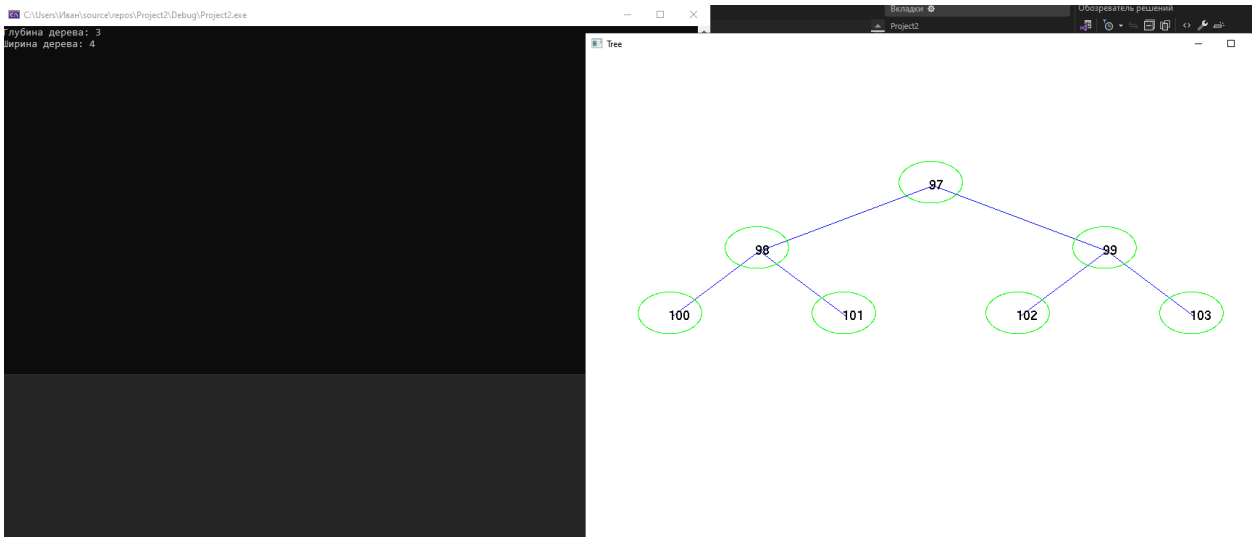
```
        tree->insert_left('b');
        tree->insert_right('c');
        tree->get_left()->insert_left('d');
        tree->get_left()->insert_right('e');
        tree->get_right()->insert_left('f');
        tree->get_right()->insert_right('g');
        PrintingInfo();
        CountLevels(tree, LevelCounter);
        CoordsCalculate(tree, Coords);
        tree->drawTree(argc, argv, 960, 720);
        return 0;
}
```

Работа программы:

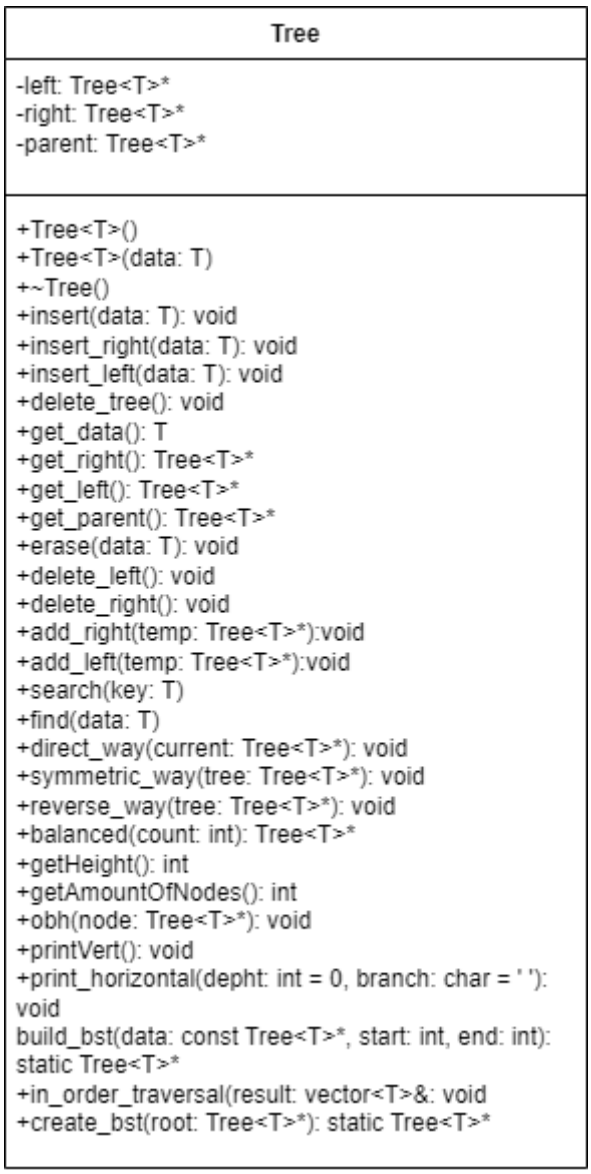
```
Горизонтальный вывод дерева:
      /--g
   /--c
      \--f
 --a
      /--e
   \--b
      \--d


Вертикальный вывод дерева:
   a
 b   c
d e f g


Прямой обход:
a b d e c f g
Симметричный обход:
d b e a f c g
Обратный обход:
d e b f g c a

Введите кол-во элементов в сбалансированном дереве: 6
Введите данные для сбалансированного дерева: a
Введите данные для сбалансированного дерева: b
Введите данные для сбалансированного дерева: c
Введите данные для сбалансированного дерева: a
Введите данные для сбалансированного дерева: d
Введите данные для сбалансированного дерева: e
```

```
Горизонтальный вывод сбалансированного дерева:
   /--d
      \--e
 --a
      /--a
   \--b
      \--c


Вертикальный вывод сбалансированного дерева:
   a
 b   d
c a e


Преобразуем дерево, в дерево поиска:
Горизонтальный вывод дерева поиска:
   /--d
      \--e
 --a
      /--a
   \--b
      \--c


Введите символ для задания: a
Количество элементов с заданным ключом a : 2
```

OpenGL:


C:\Users\Иван\source\repos\Project2\Debug\Project2.exe
Глубина дерева: 3
Ширина дерева: 4
Tree
97
98
99
100
101
102
103

UML:


Tree
-left: Tree<T>*
-right: Tree<T>*
-parent: Tree<T>*
+Tree<T>()
+Tree<T>(data: T)
+~Tree()
+insert(data: T): void
+insert_right(data: T): void
+insert_left(data: T): void
+delete_tree(): void
+get_data(): T
+get_right(): Tree<T>*
+get_left(): Tree<T>*
+get_parent(): Tree<T>*
+erase(data: T): void
+delete_left(): void
+delete_right(): void
+add_right(temp: Tree<T>*):void
+add_left(temp: Tree<T>*):void
+search(key: T)
+find(data: T)
+direct_way(current: Tree<T>*): void
+symmetric_way(tree: Tree<T>*): void
+reverse_way(tree: Tree<T>*): void
+balanced(count: int): Tree<T>*
+getHeight(): int
+getAmountOfNodes(): int
+obh(node: Tree<T>*): void
+printVert(): void
+print_horizontal(depht: int = 0, branch: char = ' '):
void
build_bst(data: const Tree<T>*, start: int, end: int):
static Tree<T>*
+in_order_traversal(result: vector<T>&: void
+create_bst(root: Tree<T>*): static Tree<T>*