

Федеральное государственное автономное образовательное учреждение
высшего образования «Пермский национальный исследовательский
политехнический университет»

Лабораторная работа №1

«Бинарные деревья»

Вариант №25

Выполнил:

студент первого курса

ЭТФ группы РИС-23-36

Акбашева Софья Руслановна

Проверила:

Доцент кафедры ИТАС О. А. Полякова

Пермь, 2024

Бинарные деревья

Цель работы

Получить практические навыки работы с бинарными деревьями.

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 задания

Тип информационного поля double. Найти минимальный элемент в дереве.

Анализ задачи

1) Бинарное дерево – это динамическая структура данных, состоящая из узлов, каждый из которых содержит, кроме данных, не более двух ссылок на различные бинарные деревья. Для описания дерева использую класс.

```
class Tree {  
private:  
    Tree* left; //левая ветка  
    Tree* right; //правая ветка  
    Tree* parent; //родительская ветка  
    double data; //данные в узле  
    int Node_radius; //радиус узла  
};
```

2) Для визуализации дерева и интерфейса будет использована библиотека SFML.

```
#include <SFML/Graphics.hpp>
```

3) Для вставки элемента в дерево используется функция `insert`. В методе используется шаблонный класс `Tree`, который содержит указатели на данные и ссылки на дочерние узлы (`left` и `right`). В методе `insert` выполняется поиск места для вставки нового элемента `data`. Если элемент уже присутствует в дереве, то он не будет добавлен повторно. После нахождения подходящего места для вставки вызывается метод `insert_right`, который выполняет фактическую вставку элемента и обновление ссылок.

4) Для того, чтобы выполнить определенную операцию над всеми узлами дерева, все узлы надо обойти. Такая задача называется обходом дерева. При обходе узлы должны посещаться в определенном порядке.

5) Метод `direct_way` класса `Tree` выполняет прямой обход дерева (то есть, посещение каждого узла ровно один раз), добавляя элементы данных в вектор `vect`. Обход начинается с текущего узла `current` и продолжается рекурсивно для левого и правого поддеревьев. Если текущий узел равен `nullptr`, то обход возвращается без изменений вектора. Элементы данных могут быть также выведены на экран с помощью комментария в коде.

6) Метод `symmetric_way` класса `Tree` выполняет симметричный обход дерева, добавляя элементы данных в вектор `vect`. Обход начинается с левого поддерева и продолжается к правому поддереву, затем добавляются данные текущего узла. Элементы данных могут быть также выведены на экран с помощью комментария в коде. Если текущий узел равен `nullptr`, то обход возвращается без изменений вектора.

7) Метод `reverse_way` класса `Tree` выполняет обратный обход дерева, добавляя элементы данных в вектор `vect`. Обход начинается с правого поддерева и продолжается к левому поддереву, затем добавляются данные текущего узла. Элементы данных могут быть также выведены на экран с помощью комментария в коде. Если текущий узел равен `nullptr`, то обход возвращается без изменений вектора.

8) Функция `get_height` возвращает целое число, которое является количеством уровней дерева.

9) Функция `get_amount_of_nodes()` возвращает целое число, которое является количеством узлов дерева, т.е. поддеревьев.

10) Функция `replace_NULL_for_Empty()` преобразует дерево из неполного в полное, посредством добавления пустых узлов в копию текущего дерева.

11) Метод `erase` класса `Tree` удаляет узел из двоичного дерева поиска, основанного на данных. Сначала находится узел, который нужно удалить. Затем, в зависимости от наличия дочерних узлов, либо просто удаляется узел, если он листовой, либо заменяется данными следующего узла, если такой существует, и удаляется следующий узел. Если узел имеет только одного дочернего узла, этот дочерний узел становится новым местом для удаленного узла. Если узел имеет двух дочерних узлов, они объединяются, и данные из следующего узла копируются в удаляемый узел.

12) Функция `search_by_key` находит узел дерева по ключу. Метод рекурсивно проверяет сначала левую ветвь дерева, затем правую. Если узел с заданным ключом найден, возвращается указатель на этот узел. В противном случае возвращается `nullptr`.

13) Функция `printVTree` используется для печати дерева в виде строки символов. Она принимает параметр `k`, который определяет количество пробелов между узлами дерева. Код начинается с определения нескольких переменных и структур данных. Затем создается копия текущего дерева (`tree_2`) и выполняется его достройка до идеальной симметрии. После этого создаются два вектора: `vest_1` для хранения деревьев и `vest_2` для хранения позиций узлов. Далее происходит перебор всех узлов дерева, добавление их в векторы и заполнение позиций узлов. После этого корректируются позиции узлов в векторе `vest_2`. Затем код проходит через вектор деревьев и печатает узлы, учитывая количество пробелов и номер строки.

14) Функция `Draw` выводит текущее бинарное дерево в окне интерфейса. Код выполняет следующие задачи:

1. Создание экземпляра класса `RenderWindow` с заданными размерами и заголовком.

2. Загрузка шрифта для кнопки закрытия окна.

3. Создание экземпляра класса `RectButton`, который представляет кнопку с возможностью обработки событий мыши.

4. Цикл обработки событий, включая обработку закрытия окна и нажатий кнопок мыши.

5. Очистка окна перед каждым новым кадром.

6. Определение позиций вершин дерева в двумерном массиве `Positions`.

7. Построение и отрисовка дерева поиска с использованием рекурсии и логики, основанной на структуре данных `levels`.

8. Отрисовка кнопки закрытия окна.

9. Обновление и отображение содержимого окна.

Функция `Draw_node` выводит узел, и связующую его ветвь.

15) Для получения значения узла, который надо добавить или удалить, необходимо текстовое поле, куда пользователь будет вводить данные. Класс `TextBox` в этом коде представляет собой текстовый редактор, который может быть использован в приложениях, созданных с помощью библиотеки SFML. Он предоставляет различные методы для настройки размера, положения, текста и других параметров. Также класс имеет внутренние структуры для рисования рамки и мигающего курсора. Методы класса включают:

- `draw`: для отрисовки текстового поля и его содержимого.

- `handleEvent`: для обработки событий ввода, таких как нажатия клавиш.

- `getCurrentText`: для получения текущего текста в текстовом поле.

Кроме того, класс `TextBox` содержит вложенный класс `Text`, который управляет отображением текста внутри текстового поля. Этот класс предоставляет методы для установки текста, его позиции и размера.

16) Класс `Button` представляет собой абстрактный класс, который определяет базовые методы для всех типов кнопок. Подклассы должны реализовать эти методы для конкретной реализации кнопки.

17) Класс `RectButton` наследуется от `Button` и реализует конкретные методы для прямоугольной кнопки. Он также содержит дополнительные члены данных, такие как `sf::RectangleShape button`, которые используются для рисования самой кнопки. Конструктор `RectButton` принимает параметры для определения размеров и положения кнопки. Деструктор не делает ничего особенного, так как он пустой. Метод `getButtonStatus` обрабатывает события мыши и обновляет состояние кнопки (`isHover`, `isPressed`). Метод `draw` отвечает за отрисовку кнопки на экране. Метод `setButtonLabel` устанавливает надпись на кнопке.

18) Функция `enter_the_data` создает окно с заголовком "Ведите..." и содержит кнопку "Продолжить". Когда пользователь нажимает кнопку, программа закрывается. В окне также есть текстовое поле, где пользователь может ввести информацию. После закрытия окна программа возвращает введенную информацию с форматом `string`.

19) Функция `string_to_double_bool` проверяет, можно ли перевести строку в число с типом `double`. Функция используется для обработки строки, полученной из функции `enter_the_data`.

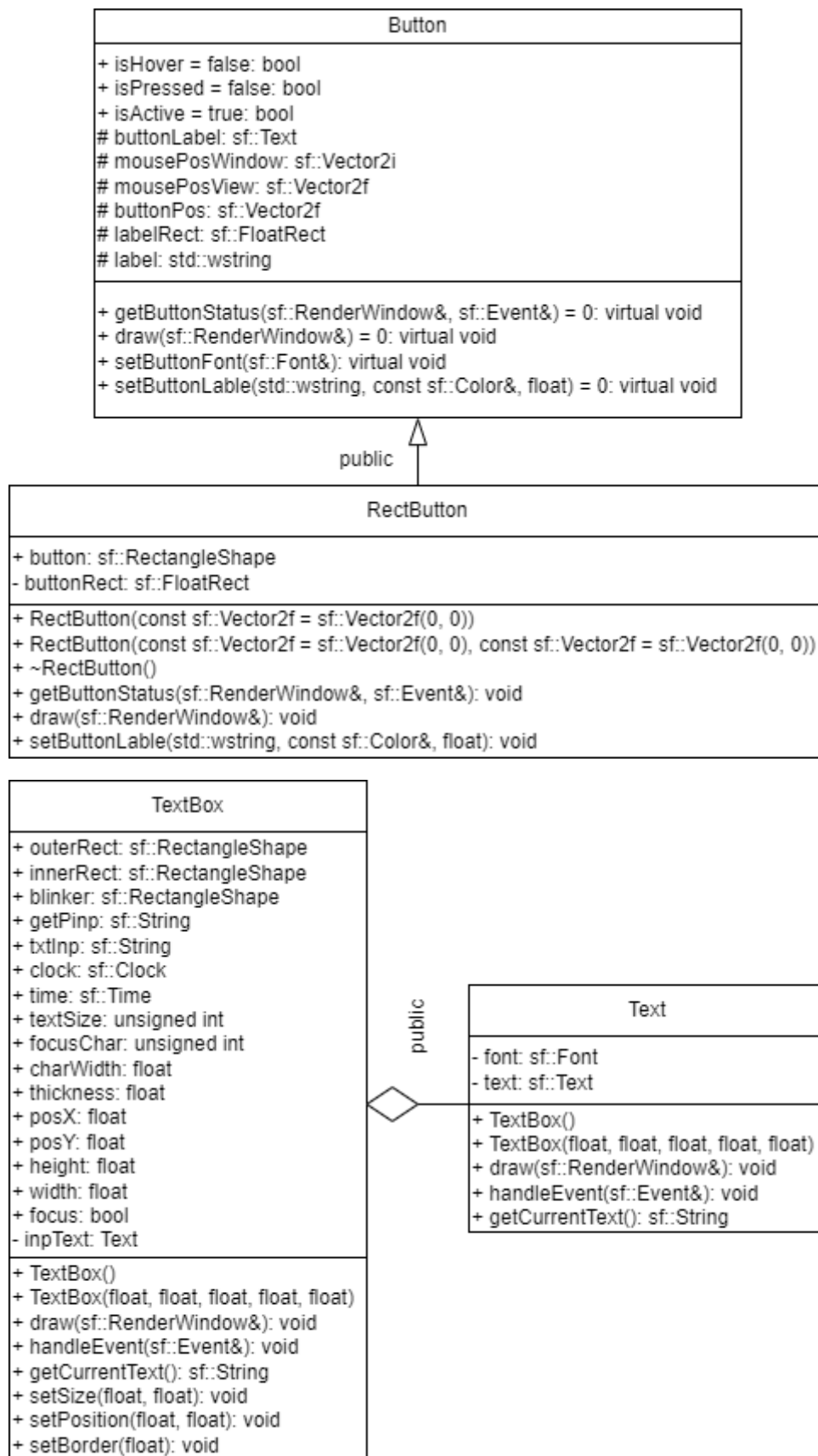
20) Функция `string_to_double` переводит строку с типом `string` в число с типом `double`. Функция используется для обработки строки, полученной из функции `enter_the_data`.

21) Функция `error_or_success_message` создает окно с заголовком `title` и отображает сообщение `message` в центре окна. Она также добавляет кнопку "Ok" внизу окна. При нажатии на кнопку "Ok" окно закрывается. Пользовательский ввод обрабатывается через событие мыши. Шрифт загружается из файла `"ofont.ru_American TextC.ttf"`.

22) Для использования какого-либо шрифта в интерфейсе необходимо загрузить в папку с проектом файл с расширением .ttf, в котором хранится необходимый шрифт.

UML диаграмма

| Tree |
|---|
| <pre> - left: Tree* - right: Tree* - parent: Tree* - data: double - Node_radius: int + Tree(T) + Tree() + ~Tree() + get_data(): T + replace(T): void + delete_left(): void + delete_right(): void + insert_left(T): void + insert_right(T): void + direct_way(Tree<T>*, vector<T>&): vector<T> + symmetric_way(Tree<T>*, vector<T>&): vector<T> + reverse_way(Tree<T>*, vector<T>&): vector<T> + parse(Tree<T>*, list<T>&): void + delete_tree(): void + insert(T): void + erase(T): void + add_left(Tree<T>* temp): void + add_right(Tree<T>* temp): void + printVTree(int): void + replace_NULL_for_Empty(): Tree<T>* + get_height(): int + get_amount_of_nodes(): int + replace_help(Tree<T>* node, int H, int cur_level): Tree<T>* + get_pos(int index, int width, int cur_level, int max_level): int + level_scan(): void + in_ascending_order(vector<T>& result): void + create_bst_from_balanced_tree(Tree<T>* tree_1): static Tree<T>* + build_balanced_bst(const vector<T>& data, int start_ind, int end_ind, Tree<T>* parent = nullptr): static + Tree<T>* + eject_left(): Tree<T>* + eject_right(): Tree<T>* + get_left(): Tree<T>* + get_right(): Tree<T>* + get_parent(): Tree<T>* + copy_tree(): Tree<T>* + search_by_key(T): Tree<T>* + find(T): Tree<T>* + ptr_to_prev(): Tree<T>* + ptr_to_next(): Tree<T>* + find_min(): Tree<T>* + find_max(): Tree<T>* + balanced_tree(int): Tree<T>* + read_into_vector(vector<vector<Tree<T>*>>& container, int height): void + Draw(std::wstring title, bool flag = true): void + Draw_node(Tree<T>* branch, map < Tree<T>*, Vector2f>& positions, RenderWindow& window, bool flag): void + find_by_min(): void + find_node_by_key(): void </pre> |



Код

Файл Tree.h

```
#pragma once
#include <SFML/Graphics.hpp>
```



```

#include <iomanip>
#include <sstream>
#include <list>
#include <vector>
#include <iostream>
#include "other functions.h"
#include <map>
#include <string>
#include <algorithm>

using namespace std;
using namespace sf;

template <class T>
class Tree {
private:
    Tree* left; // левая ветка
    Tree* right; // правая ветка
    Tree* parent; // родительская ветка
    double data; // данные в узле
    int Node_radius; // радиус узла
public:
    Tree(T); // конструктор со значением
    Tree(); // конструктор по умолчанию
    ~Tree(); // деструктор

    T get_data(); // получаю значение
    void replace(T); // замена значения в текущем узле

    void delete_left(); // удаление левой ветки
    void delete_right(); // удаление правой ветки
    void insert_left(T); // вставка нового узла в левую ветвь с указанным значением
    void insert_right(T); // вставка нового узла в правую ветвь с указанным значением

    vector<T> direct_way(Tree<T>*, vector<T>&); // прямой обход дерева
    vector<T> symmetric_way(Tree<T>*, vector<T>&); // симметричный обход дерева
    vector<T> reverse_way(Tree<T>*, vector<T>&); // обратный обход дерева

    void parse(Tree<T>*, list<T>&); // обход дерева и запись значений в переданный список
    void delete_tree() { delete this; }; // удаление дерева

    void insert(T); // вставка узла в нужное место
    void erase(T); // полное удаление узла с указанным значением

    void add_left(Tree<T>* temp) { left = temp; }; // установка левой ветви
    void add_right(Tree<T>* temp) { right = temp; }; // установка правой ветви

    void printVTree(int); // вертикальная печать
    Tree<T>* replace_NULL_for_Empty(); // достраивание дерева до ИДЕАЛЬНОЙ симметрии
    int get_height(); // высота дерева
    int get_amount_of_nodes(); // количество узлов в дереве
    Tree<T>* replace_help(Tree<T>* node, int H, int cur_level); // преобразую дерево из неполного в полное
    int get_pos(int index, int width, int cur_level, int max_level); // количество пробелов для данного узла
    void level_scan(); // вывод элементов уровень за уровнем

    void in_ascending_order(vector<T>& result); // обход дерева в порядке возрастания и сохранение значений
    в векторе
    static Tree<T>* create_bst_from_balanced_tree(Tree<T>* tree_1); // из сбалансированного дерева в дерево
    поиска
    static Tree<T>* build_balanced_bst(const vector<T>& data, int start_ind, int end_ind, Tree<T>* parent =
    nullptr); // построение сбалансированного дерева из отсортированных данных

```

```

Tree<T>* eject_left();//удаление и возврат левой ветви
Tree<T>* eject_right();//удаление и возврат правой ветви
Tree<T>* get_left();//указатель на левую ветвь
Tree<T>* get_right();//указатель на левую ветвь
Tree<T>* get_parent();//указатель на родителя
Tree<T>* copy_tree();//копия текущего дерева
Tree<T>* search_by_key(T);//поиск по ключу
Tree<T>* find(T);//находит узел с указанным значением
Tree<T>* ptr_to_prev();//указатель на предыдущий узел
Tree<T>* ptr_to_next();//указатель на следующий узел
Tree<T>* find_min();//минимальное значение из дерева
Tree<T>* find_max();//максимальное значение из дерева
Tree<T>* balanced_tree(int);//сбалансированное дерево заданной высоты

void read_into_vector(vector<vector<Tree<T>*>>& container, int height);//сканирую дерево
void Draw(std::wstring title, bool flag = true);//рисую дерево
void Draw_node(Tree<T>* branch, map<Tree<T>*, Vector2f>& positions, RenderWindow& window, bool
flag);//рисую узел
void find_by_min();//минимальный элемент
void find_node_by_key();//поиск по ключу
};

template <class T>
Tree<T>::Tree(T data) {//конструктор со значением
    this->data = data;//присваиваю данные
    left = right = parent = nullptr;//указатели в никуда
}

template <class T>
Tree<T>::Tree() {//конструктор по умолчанию
    left = right = parent = nullptr;//указатели в никуда
}

template <class T>
Tree<T>::~Tree() {//деструктор
    delete_left();//удаляю левую ветвь
    delete_right();//удаляю правую ветвь
}

template <class T>
T Tree<T>::get_data() {//получаю значение
    return data;//возвращаю значение текущего узла
}

template <class T>
Tree<T>* Tree<T>::get_left() {//указатель на левую ветвь
    return left;
}

template <class T>
Tree<T>* Tree<T>::get_right() {//указатель на правую ветвь
    return right;
}

template <class T>
Tree<T>* Tree<T>::get_parent() {//указатель на родителя
    return parent;
}

template <class T>
Tree<T>* Tree<T>::ptr_to_next() {//указатель на следующий узел
    Tree<T>* current = this;//указатель на текущий узел

```

```

        if (current->right != nullptr) { //если правая ветвь существует
            return current->right->find_min();
        }
        Tree<T>* tmp = current->parent; //родительский узел
        while (tmp != nullptr && current == tmp->right) {
            current = tmp;
            tmp = tmp->parent;
        }
        return tmp;
    }

template <class T>
Tree<T>* Tree<T>::ptr_to_prev() { //указатель на предыдущий узел
    if (left != nullptr) { //если левая ветвь существует
        Tree<T>* tmp = left; //указатель на левую ветвь
        while (tmp->right != nullptr) { //пока правая ветвь существует
            tmp = tmp->right; //следующая правая ветвь
        }
        return tmp;
    }
    else {
        Tree<T>* tmp = this; //указатель на текущий узел
        while (tmp->parent != nullptr && tmp->parent->left == tmp) { //пока есть родитель и правая ветвь от
родителя
            tmp = tmp->parent; //родитель родителя
        }
        return tmp->parent;
    }
}

template <class T>
void Tree<T>::insert(T data) {
    Tree<T>* current = this; //текущий узел
    while (current != nullptr) { //пока узел существует
        if (data > current->data) { //если данные больше данных в текущем узле
            if (current->right != nullptr) { //если правая ветвь существует
                current = current->right; //следующий правый узел
            }
            else { //если правой ветви нет
                current->insert_right(data); //устанавливаю данные
                return; //выход из цикла
            }
        }
        else if (data < current->data) { //если данные меньше данных в текущем узле
            if (current->left != nullptr) { //если левая ветвь существует
                current = current->left; //следующий левый узел
            }
            else { //если левой ветви нет
                current->insert_right(data); //устанавливаю данные
                return; //выход из цикла
            }
        }
    }
}

template <class T>
void Tree<T>::insert_right(T data) { //вставка нового узла в правую ветвь с указанным значением
    Tree<T>* new_node = new Tree<T>(data); //новое дерево с данными
    if (this->right != nullptr) { //если правая ветвь от текущей существует
        this->right->parent = new_node; //текущий узел заменяется на new_node
        new_node->right = this->right; //правой ветви присваивается новое поддерево
    }
    this->right = new_node; //правая ветвь текущей становится новой
}

```

```

        new_node->parent = this;//у новой ветви присваивается родитель - текущий узел
    }

template <class T>
void Tree<T>::insert_left(T data) {//вставка нового узла в правую ветвь с указанным значением
    Tree<T>* new_node = new Tree<T>(data);//новый дерево с данными
    if (this->left != nullptr) {//если правая ветвь от текущей существует
        this->left->parent = new_node;//текущий узел заменяется на new_node
        new_node->left = this->left;//правой ветви присваивается новое поддерево
    }
    this->left = new_node;//правая ветвь текущей становится новой
    new_node->parent = this;//у новой ветви присваивается родитель - текущий узел
}

template <class T>
vector<T> Tree<T>::direct_way(Tree<T>* current, vector<T>& vect) {//прямой обход
    if (current == nullptr) {//если (под)дерева нет
        return vect;
    }
    else {//если (под)дерево существует
        vect.push_back(current->get_data());
        //cout << current->get_data() << " "; //вывод данных в текущем узле
        direct_way(current->get_left(), vect);//для левой ветви
        direct_way(current->get_right(), vect);//для правой ветви
    }
    return vect;
}

template <class T>
vector<T> Tree<T>::symmetric_way(Tree<T>* tree, vector<T>& vect) {//симметричный обход
    if (tree != nullptr) {//если (под)дерево существует
        symmetric_way(tree->left, vect);//для левой ветви
        vect.push_back(tree->get_data());
        //cout << tree->data << " "; //вывод данных в текущем узле
        symmetric_way(tree->right, vect);//для правой ветви
    }
    return vect;
}

template <class T>
vector<T> Tree<T>::reverse_way(Tree<T>* tree, vector<T>& vect) {//обратный обход
    if (tree != nullptr) {//если (под)дерево существует
        reverse_way(tree->left, vect);//для левой ветви
        reverse_way(tree->right, vect);//для правой ветви
        //cout << tree->data << " "; //вывод данных в текущем узле
        vect.push_back(tree->get_data());
    }
    return vect;
}

template <class T>
void Tree<T>::level_scan() {//вывод элементов на определенном уровне
    vector<Tree<T>*> vect;//вектор узлов
    Tree<T>* current = this;
    vect.push_back(current);
    for (int i = 0; i < this->get_amount_of_nodes(); i++) {//пока не пройду все узлы в дереве
        if (vect.at(i)->left != NULL) {//если левая ветвь существует
            vect.push_back(vect.at(i)->left);
        }
        if (vect.at(i)->right != NULL) {//если правая ветвь существует
            vect.push_back(vect.at(i)->right);
        }
    }
}

```

```

    }
    for (int i = 0; i < vect.size(); i++) {
        cout << vect.at(i)->get_data() << " ";
    }
    cout << endl;
}

template <class T>
int Tree<T>::get_height() { //высота дерева
    int h1 = 0, h2 = 0;
    if (this == NULL) { //если дерево пустое
        return 0;
    }
    if (this->left != NULL) { //если левая ветвь не пустая
        h1 = this->left->get_height();
    }
    if (this->right != NULL) { //если правая ветвь не пустая
        h2 = this->right->get_height();
    }
    if (h1 >= h2) { //выбираю наибольшую высоту из правой и левой ветви
        return h1 + 1; //+ 1 т.к. корень не считается
    }
    return h2 + 1;
}

template <class T>
int Tree<T>::get_amount_of_nodes() { //количество узлов в дереве
    if (this == NULL) { //если дерево пустое
        return 0;
    }
    if ((this->left == NULL) && (this->right == NULL)) { //если левая и правая ветви - листья
        return 1;
    }
    int left_branch = 0, right_branch = 0;
    if (this->left != NULL) { //если левая ветвь существует
        left_branch = this->left->get_amount_of_nodes();
    }
    if (this->right != NULL) { //если правая ветвь существует
        right_branch = this->right->get_amount_of_nodes();
    }
    return (left_branch + right_branch + 1);
}

template <class T>
Tree<T>* Tree<T>::replace_NULL_for_Empty() { //преобразую дерево из неполного в полное
    Tree<T>* node = this->copy_tree(); //копирую настоящее дерево
    int H = node->get_height(); //вычисляю высоту дерева
    node = replace_help(node, H, 0); //дополняю дерево пустыми узлами
    return node;
}

template <class T>
Tree<T>* Tree<T>::replace_help(Tree<T>* node, int H, int cur_level) {
    if ((node->get_left() == NULL) && (cur_level != H - 1)) { //если ветвь пустая и высота не конечная
        node->insert_left(NULL);
    }
    if ((node->get_right() == NULL) && (cur_level != H - 1)) { //если ветвь пустая и высота не конечная
        node->insert_right(NULL);
    }

    if (node->get_left() != NULL) { //если левая ветвь существует
        node->add_left(replace_help(node->get_left(), H, cur_level + 1));
    }
}

```

```

        if (node->get_right() != NULL) { //если правая ветвь существует
            node->add_right(replace_help(node->get_right(), H, cur_level + 1));
        }
        return node;
    }
}

struct pos { //структура для вертикального вывода дерева
    int column; //столбец - x
    int row; //строка - y
};

template <class T>
int Tree<T>::get_pos(int index, int width, int cur_level, int max_level) { //определяю позицию
    int x1 = 0, x2 = pow(2, cur_level) - 1;
    int y1 = width / pow(2, cur_level + 1), y2 = width - pow(2, max_level - cur_level);
    if (x1 == x2) {
        return y1;
    }
    double k = (y1 - y2) / (x1 - x2);
    double m = -x1 * k + y1;
    int y = (int)(k * index + m);
    return y;
}

template <class T>
void Tree<T>::printVTree(int k) {
    int height = this->get_height();
    int max_leafs = pow(2, height - 1); //максимальное число листьев внизу
    int width = 2 * max_leafs - 1; //минимальная ширина дерева
    int cur_level = 0; //номер строки
    int index = 0; //индекс элемента в строке
    int fact_spaces = get_pos(index, width, cur_level, height - 1); //число пробелов перед корнем

    pos node; // Определение структуры для хранения позиции узла
    vector<Tree<T>*> vect_1; //вектор деревьев
    vector<pos> vect_2; // Вектор для хранения позиций узлов

    Tree<T>* tree_2 = this->copy_tree(); //копирую текущее дерево
    tree_2 = tree_2->replace_NULL_for_Empty(); //доставление дерева до ИДЕАЛЬНОЙ симметрии

    Tree<T>* tree_3 = tree_2;
    vect_1.push_back(tree_3); // Добавление первого дерева в вектор
    // Заполнение начальной позиции корня
    node.column = fact_spaces; //число пробелов перед корнем
    node.row = cur_level; //номер строки
    vect_2.push_back(node);

    for (int i = 0; i < tree_2->get_amount_of_nodes(); i++) { // Перебор всех узлов дерева
        if (pow(2, cur_level) <= index + 1) { // Если индекс меньше или равен степени двойки,
            соответствующей текущему уровню, увеличиваем уровень
                index = 0;
                cur_level++;
            }
        if (vect_1.at(i)->left != NULL) { // Проверка наличия левого потомка
            vect_1.push_back(vect_1.at(i)->left); //добавление потомка в вектор
            fact_spaces = get_pos(index, width, cur_level, height - 1); // Вычисление количества
            пробелов перед новым корнем
            node.column = fact_spaces; // Заполнение позиции нового узла
            node.row = cur_level;
            vect_2.push_back(node);
            index++;
        }
    }
}

```

```

        if (vect_1.at(i)->right != NULL) { // Проверка наличия правого потомка
            vect_1.push_back(vect_1.at(i)->right); // добавление потомка в вектор
            fact_spaces = get_pos(index, width, cur_level, height - 1); // Вычисление количества
пробелов перед новым узлом
            node.column = fact_spaces; // Заполнение позиции нового узла
            node.row = cur_level;
            vect_2.push_back(node);
            index++;
        }
    }
    for (int i = vect_1.size() - 1; i >= 0; i--) { // прохожу вектор деревьев с конца
        if (i != 0) { // если это не первый элемент
            if (vect_2.at(i - 1).row == vect_2.at(i).row) {
                vect_2.at(i).column = vect_2.at(i).column - vect_2.at(i - 1).column - 1;
            }
        }
    }
    int flag = 0; // следит за y
    for (int i = 0; i < vect_1.size(); i++) { // прохожу вектор деревьев
        node = vect_2.at(i);
        cur_level = node.row;
        if (flag < cur_level) { // перехожу на новую строку, когда y1 будет меньше y
            flag = cur_level;
            cout << endl; // переход на новую строку
        }
        fact_spaces = node.column; // число пробелов перед узлом
        int real_spaces = k * fact_spaces;

        print_spaces(0, real_spaces);

        if (vect_1.at(i)->get_data() == NULL) {
            cout << " ";
        }
        else {
            cout << vect_1.at(i)->get_data(); // вывод узла
        }
        print_spaces(0, k);
    }
    cout << endl;
}

template <class T>
void Tree<T>::parse(Tree<T>* current, list<T>& list_1) { // горизонтальный вывод
    if (current == nullptr) { // если (под)дерево не существует
        return;
    }
    else {
        list_1.push_back(current->get_data());
        parse(current->get_left(), list_1);
        parse(current->get_right(), list_1);
    }
}

template <class T>
Tree<T>* Tree<T>::balanced_tree(int count) { // создание сбалансированного дерева
    if (count <= 0) { // пока не введем все данные для дерева
        return nullptr;
    }
    T data;
    cout << "Введите данные для сбалансированного дерева: ";
    cin >> data;

```

```

    Tree<T>* tmp = new Tree<T>(data);
    tmp->add_left(balanced_tree(count / 2));
    tmp->add_right(balanced_tree(count - count / 2 - 1));
    return tmp;
}

template <class T>
void Tree<T>::erase(T data) { //удаление узла по данным
    Tree<T>* to_erase = this->find(data); //узел который необходимо удалить
    Tree<T>* to_parent = to_erase->parent; // Родительский узел для узла, который нужно удалить

    if (to_erase->left == nullptr && to_erase->right == nullptr) { //если узел - листок
        if (to_parent->left == to_erase) { //если надо удалить левую ветвь
            to_parent->left = nullptr; //родительский узел никуда не указывает
        }
        else { //если надо удалить правую ветвь
            to_parent->right = nullptr; //родительский узел никуда не указывает
        }
        delete to_erase; // Удаляем ссылку на дочерний узел
    }
    else if ((to_erase->left != nullptr && to_erase->right == nullptr) || (to_erase->left == nullptr && to_erase->right
!= nullptr)) { //если ветка только одна - левая или правая
        if (to_erase->left == nullptr) { //если левая ветвь не существует
            if (to_erase == to_parent->left) { //если надо удалить левую ветвь
                to_parent->left = to_erase->right; // Устанавливаем новую левую ветвь
            }
            else { //если надо удалить правую ветвь
                to_parent->right = to_erase->right; // Устанавливаем новую правую ветвь
            }
            to_parent->right->parent = to_parent; //устанавливаю нового родителя для правой ветви
        }
        else { //если левая ветвь существует
            if (to_erase == to_parent->left) { //если надо удалить левую ветвь
                to_parent->left = to_erase->left; // Устанавливаем новую левую ветвь
            }
            else { //если надо удалить правую ветвь
                to_parent->right = to_erase->left; // Устанавливаем новую правую ветвь
            }
            to_parent->left->parent = to_parent; //устанавливаю нового родителя для левой ветви
        }
    }
    else {
        Tree<T>* next = to_erase->ptr_to_next(); // Получаем следующий узел
        to_erase->data = next->data; // Копируем данные следующего узла в текущий
        if (next == next->parent->left) { // Проверяем, является ли следующий узел левым потомком
своего родителя
            next->parent->left = next->right; // Устанавливаем новую левую ветвь для родителя
            if (next->right != nullptr) { // Проверяем, есть ли правая ветвь
                next->right->parent = next->parent; // Устанавливаем нового родителя для
правой ветви
            }
        }
        else {
            next->parent->right = next->right; // Устанавливаем новую правую ветвь для родителя
            if (next->right != nullptr) { // Проверяем, есть ли правая ветвь
                next->right->parent = next->parent; // Устанавливаем нового родителя для
правой ветви
            }
        }
        delete next; // Освобождаем память, занятую следующим узлом
    }
}

```



```

}

template <class T>
void Tree<T>::delete_left() { //удаление левого узла
    if (left != NULL) { //если левая ветка существует
        left->delete_left(); //удаление левой ветви узла
        left->delete_right(); //удаление правой ветви узла
        delete left; //удаление самого узла
    }
}

template <class T>
void Tree<T>::delete_right() { //удаление правого узла
    if (right != NULL) { //если правая ветка существует
        right->delete_left(); //удаление левой ветви узла
        right->delete_right(); //удаление правой ветви узла
        delete right; //удаление самого узла
    }
}

template <class T>
Tree<T>* Tree<T>::eject_left() { //удаление левой ветки
    Tree<T>* temp = left; //левая ветвь
    left = nullptr; //зануляю ветку
    if (temp != nullptr) { //если ветка существует
        temp->parent = nullptr; //родитель никуда не ссылается
    }
    return temp; //возвращаю удаленный элемент
}

template <class T>
Tree<T>* Tree<T>::eject_right() { //удаление правой ветки
    Tree<T>* temp = right; //правая ветвь
    right = nullptr; //зануляю ветку
    if (temp != nullptr) { //если ветка существует
        temp->parent = nullptr; //родитель никуда не ссылается
    }
    return temp; //возвращаю удаленный элемент
}

template <class T>
Tree<T>* Tree<T>::search_by_key(T key) { //поиск по ключу
    if (data == key) { //если данные в узле равны ключу
        return this; //возвращаю текущий узел
    }
    if (left != nullptr) { //если левая ветвь существует
        Tree<T>* result = left->search_by_key(key);
        if (result != nullptr) { //если результат найден
            return result; //возвращаю поддереву
        }
    }
    if (right != nullptr) { //если правая ветвь существует
        Tree<T>* result = right->search_by_key(key);
        if (result != nullptr) { //если результат найден
            return result; //возвращаю поддереву
        }
    }
    return nullptr; //возвращаю поддереву
}

template <class T>
Tree<T>* Tree<T>::find(T data) { //ищу узел по данным
    if (this == nullptr || this->data == data) { //если элемент найден или больше узлов нет

```

```

        return this; //указатель на текущий узел
    }
    else if (data > this->data) { //для правой ветви
        return this->right->find(data);
    }
    else { //для левой ветви
        return this->left->find(data);
    }
}

template <class T>
Tree<T>* Tree<T>::find_max() { //ищу максимальный элемент
    if (this->right == nullptr) { //если больше нет правых узлов
        return this;
    }
    return this->right->find_max();
}

template <class T>
Tree<T>* Tree<T>::find_min() { //ищу минимальный элемент
    Tree<T>* min_node = this;
    while (min_node->left != nullptr) { //если больше нет левых узлов
        min_node = min_node->left;
    }
    return min_node;
}

template <class T>
void Tree<T>::in_ascending_order(vector<T>& result) { //обход дерева в порядке возрастания и сохранение значений
    в векторе
        if (left != nullptr) { //если левая ветвь существует
            left->in_ascending_order(result);
        }
        result.push_back(data);
        if (right != nullptr) { //если правая ветвь существует
            right->in_ascending_order(result);
        }
    }

template <class T>
Tree<T>* Tree<T>::create_bst_from_balanced_tree(Tree<T>* tree_1) { //из сбалансированного дерева в дерево
поиска
    vector<T> sorted_data; //вектор отсортированных данных
    tree_1->in_ascending_order(sorted_data);
    cout << endl;
    sort(sorted_data.begin(), sorted_data.end());
    return build_balanced_bst(sorted_data, 0, sorted_data.size() - 1);
}

template <class T>
Tree<T>* Tree<T>::build_balanced_bst(const vector<T>& data, int start_ind, int end_ind, Tree<T>* parent)
{ //построение сбалансированного дерева из отсортированных данных

    if (start_ind > end_ind) { //есть ли данные для обработки. Если нет, возвращается nullptr, так как дальше
    обрабатывать нечего
        return nullptr;
    }

    int middle = start_ind + (end_ind - start_ind) / 2; //вычисление среднего индекса подмассива для
    разделения данных на две части.
    Tree<T>* new_node = new Tree<T>(data[middle]); //создание нового узла дерева поиска с значением из
    середины подмассива.

```

```

        new_node->left = build_balanced_bst(data, start_ind, middle - 1, new_node); //рекурсивный вызов функции
        build_balanced_bst для построения
        new_node->right = build_balanced_bst(data, middle + 1, end_ind, new_node); //рекурсивный вызов функции
        build_balanced_bst для построения
        if (parent != nullptr) {
            new_node->parent = parent;
        }
        return new_node;
    }
}

```

```

template <class T>
void Tree<T>::replace(T data) { //замена текущего узла на переданное значение
    Tree<T>* search_tree = new Tree<T>();
    list<T> list_1;
    parse(this, list_1);
    for (auto data : list_1) {
        search_tree->insert(data);
    }
    Tree<T>* current = search_tree->find(data);
    if (current == nullptr) {
        return;
    }
    cout << "Затем: " << endl;
    if (current->left == nullptr && current->right == nullptr) {
        current->data = -1;
        search_tree->print_tree(2);
        return;
    }
}
}

```

```

template <class T>
Tree<T>* Tree<T>::copy_tree() { //копия дерева

    Tree<T>* new_tree = new Tree<T>(data); //новое дерева
    if (left != nullptr) { //если левая ветвь существует
        new_tree->left = left->copy_tree(); //копирую левую ветвь
        new_tree->left->parent = new_tree; //устанавливаю родителя
    }
    if (right != nullptr) { //если правую ветвь существует
        new_tree->right = right->copy_tree(); //копирую правую ветвь
        new_tree->right->parent = new_tree; //устанавливаю родителя
    }
    return new_tree; //копия дерева
}

```

```

template <class T>
void Tree<T>::read_into_vector(vector<vector<Tree<T>*>> & container, int height) { //сканирую дерево
    vector<Tree<T>*> buffer_of_branches; //буффер деревьев в потомков
    buffer_of_branches.push_back(this); //заношу корень дерева

    for (int i = 0; i < height; i++) {
        vector<Tree<T>*> branches_on_yhe_level; //вектор вершин на уровне i
        vector<Tree<T>*> updated_buffer_of_Trees; //вектор потомков

        for (int j = 0; j < buffer_of_branches.size(); j++) { //прохожу по потомкам
            Tree<T>* current = buffer_of_branches[j]; //текущая вершина
            if (current == nullptr) { //если нулевой узел
                branches_on_yhe_level.push_back(nullptr); //заношу нулевой элемент
                for (int i = 0; i < 2; i++) { //нулевые элементы в вектор потомков
                    updated_buffer_of_Trees.push_back(nullptr);
                }
            }
        }
    }
}

```

```

    }
    else { //если узел есть
        updated_buffer_of_Trees.push_back(current->left); //заношу потомков
        updated_buffer_of_Trees.push_back(current->right);

        branches_on_yhe_level.push_back(current); //заношу сам элемент
    }
    container.push_back(branches_on_yhe_level); //запись вершин уровня
    buffer_of_branches.clear(); //очищаю буффер

    for (int k = 0; k < updated_buffer_of_Trees.size(); k++) {
        buffer_of_branches.push_back(updated_buffer_of_Trees[k]); //заношу новые элементы
    }
}
}

```

```

template <class T>
void Tree<T>::Draw(std::wstring title, bool flag) {
    vector<vector<Tree<T>*>> levels; //массив как дерево
    int height = this->get_height(); //высота
    read_into_vector(levels, height); //заполнение массива
    int* amount_of_spaces = new int[height]; //массив пробелов

    amount_of_spaces[0] = 1; //в низу 1 пробел
    for (int i = 1; i < height; i++) {
        amount_of_spaces[i] = amount_of_spaces[i - 1] * 2 + 1; //расчитываю кол-0 пробелов
    }
    this->Node_radius = 100 / height - 10; //радиус вершины дерева
    int height_difference = (this->Node_radius * 2) + 10; //разница между уровнями
    int h_wind = 10, w_wind = 0;
    if (flag) {
        h_wind = amount_of_spaces[height - 1] * this->Node_radius * 4;
        w_wind = ((height * height_difference) + this->Node_radius) * 1.8;
    }
    else {
        w_wind = amount_of_spaces[height - 1] * this->Node_radius * 4;
        h_wind = ((height * height_difference) + this->Node_radius) * 1.8;
    }
    RenderWindow window_tree(VideoMode(h_wind, w_wind + 50), title);

    Font font;
    font.loadFromFile("ofont.ru_American TextC.ttf");

    RectButton button_exit(sf::Vector2f(200, 40), sf::Vector2f(h_wind / 2 - 100, w_wind)); //Найти элемент по
ключу
    button_exit.setButtonFont(font);
    button_exit.setButtonLable(L"Ok", text_color, 30);

    while (window_tree.isOpen()) {
        Event event;
        button_exit.getButtonStatus(window_tree, event);
        while (window_tree.pollEvent(event))
        {
            if (event.type == Event::Closed)
                window_tree.close();
            if (event.type == Event::MouseButtonPressed) {
                if (event.key.code == Mouse::Left) {
                    if (button_exit.isPressed) {
                        window_tree.close();
                    }
                }
            }
        }
    }
}

```

```

    }
}
window_tree.clear(background_color);

map<Tree<T>*, Vector2f> Positions;//словарь вершин дерева и их координат
int y = height - height_difference + (this->Node_radius * 2);//первоначальный y

for (int i = height - 1; i > -1; i--) {
    int x = this->Node_radius * 2 * amount_of_spaces[i];
    y += height_difference * 1.5;//расчитываю y
    vector<Tree<T>*> cur_level = levels[height - i - 1];
    for (int k = 0; k < cur_level.size(); k++) {
        x += (k == 0 ? 0 : this->Node_radius * 2 * amount_of_spaces[i + 1]);

        if (cur_level[k] != nullptr) {//если элемент не на уровне 0
            Positions[cur_level[k]] = Vector2f(x, y);//заносу вершину в словарь
            Draw_node(cur_level[k], Positions, window_tree, flag);//рисую вершину
        }
        x += this->Node_radius * 2;//сдвиг по x
    }
}
button_exit.draw(window_tree);
window_tree.display();
}
}

template <class T>
void Tree<T>::Draw_node(Tree<T>* branch, map < Tree<T>*, Vector2f>& positions, RenderWindow& window, bool flag)
{
    Vector2f position = positions[branch];{//позиция ветки
    if (!flag) {
        swap(position.x, position.y);
    }
    CircleShape circle_1(this->Node_radius);//генерирую круг
    circle_1.setFillColor(button_color);//цвет внутри круга
    circle_1.setOutlineColor(text_color);//цвет снаружи круга
    circle_1.setOutlineThickness(2);//толщина внешнего контура
    circle_1.setPosition(position.x - this->Node_radius, position.y - this->Node_radius);//позиция

    Text text_1;
    ostringstream buffet;
    buffet << fixed << setprecision(1) << branch->data;//обработка вещественного числа

    Font font;
    font.loadFromFile("ofont.ru_Dealerplate.ttf");//загружаю шрифт
    text_1.setFont(font);

    text_1.setString(buffet.str());//настраиваю текст
    text_1.setFillColor(text_color);
    text_1.setCharacterSize(this->Node_radius);

    FloatRect textRect = text_1.getLocalBounds();//центрирую текст
    text_1.setOrigin(textRect.left + textRect.width / 2.0f, textRect.top + textRect.height / 2.0f);
    text_1.setPosition(Vector2f(position.x, position.y));

    if (branch->parent != nullptr) {//если есть родитель
        Vector2f Parent_Position = positions[branch->parent];{//позиция родителя
        int crutch;

        if (!flag) {
            swap(Parent_Position.x, Parent_Position.y);
            Parent_Position.x += this->Node_radius;

```

```

        position.x -= this->Node_radius;
        crutch = 1;
    }
    else {
        Parent_Position.y += this->Node_radius;
        position.y -= this->Node_radius;
        crutch = 2;
    }

    Parent_Position.x += crutch;
    position.x += crutch;
    VertexArray line(Lines, 2);

    line[0].position = Parent_Position;
    line[1].position = position;
    line[0].color = sf::Color(117, 90, 87);
    line[1].color = sf::Color(193, 135, 107);

    for (int i = crutch * 2; i >= 0; i--) {
        line[0].position.x -= 1;
        line[1].position.x -= 1;;

        window.draw(line);
    }
}
window.draw(circle_1); //рисую круг
window.draw(text_1); //рисую текст
}

template <class T>
void Tree<T>::find_by_min() {
    Tree<T>* tmp_tree = this->find_min()->copy_tree();
    tmp_tree->parent = nullptr;
    tmp_tree->Draw(L"Минимальный элемент", false);
}

template <class T>
void Tree<T>::find_node_by_key() {
    string value = enter_the_data(L"Введите элемент, который хотите найти");
    if (string_to_double_bool(value)) { //если можно перевести в double
        double required_value = string_to_double(value); //перевод в double
        Tree<T>* tmp_tree_1 = this->search_by_key(required_value);
        if (tmp_tree_1 != nullptr) { //если такой элемент есть
            Tree<T>* tmp_tree_2 = tmp_tree_1->copy_tree();
            tmp_tree_2->parent = nullptr;
            tmp_tree_2->Draw(L"Найденный элемент");
        }
        else {
            error_or_success_message(L"Такого элемента нет!", L"Ошибка");
        }
    }
    else {
        error_or_success_message(L"Такого элемента нет!", L"Ай-ай-ай");
    }
}

template <class T>
void min_el_in_tree(Tree<T>* tree) {
    RenderWindow window(sf::VideoMode(400, 250), "Error!");

    double number = tree->find_min()->get_data();

```

```

std::wstring wideString = std::to_wstring(number);

Font font;
font.loadFromFile("ofont.ru_American TextC.ttf");//загружаю шрифт

Text mes;
mes.setFont(font);
mes.setString(wideString);
mes.setFill_color(text_color);
mes.setCharacterSize(40);
mes.setPosition(30, 45);

RectButton button1(sf::Vector2f(150, 60), sf::Vector2f(125, 140));//Вертикальная печать дерева
button1.setFont(font);
button1.setButtonLabel(L"Ok", text_color, 30);

while (window.isOpen())
{
    Vector2i mousePoz = Mouse::getPosition(window);//позиция мыши в окне

    sf::Event event;

    button1.getButtonStatus(window, event);

    while (window.pollEvent(event))
    {
        if (event.type == sf::Event::Closed)
            window.close();
        if (event.type == Event::MouseButtonPressed) {
            if (event.key.code == Mouse::Left) {
                if (button1.isPressed) {
                    window.close();
                }
            }
        }
    }
    window.clear(background_color);

    button1.draw(window);
    window.draw(mes);

    window.display();
}

template <class T>
void tree_traversal(Tree<T>* tree) {//вывод всех обходов дерева
    vector<vector<T>> vector_tree_trav(3);
    vector<string> vect_wstring;
    tree->direct_way(tree, vector_tree_trav[0]);//прямой обход
    tree->reverse_way(tree, vector_tree_trav[1]);//обратный обход
    tree->symmetric_way(tree, vector_tree_trav[2]);//симметричный обход

    for (int i = 0; i < 3; i++) {//иду по всем обходам
        std::string all_str = "";//собираю строку
        for (int j = 0; j < vector_tree_trav[0].size(); j++) {
            ostreambufet buffet;//обрабатываю число с .
            buffet << fixed << setprecision(1) << vector_tree_trav[i][j];
            all_str = all_str + buffet.str() + " ";//собираю строку
        }
        vect_wstring.push_back(all_str);
    }
}

```

```
RenderWindow window(sf::VideoMode(vect_wstring[0].size() * 20, 500), L"Обходы бинарного дерева");
```

```
Font font;
```

```
font.loadFromFile("ofont.ru_American TextC.ttf");//загружаю шрифт
```

```
Text obxod_binary_tree;
```

```
obxod_binary_tree.setFont(font);
```

```
obxod_binary_tree.setString(L"Обходы бинарного дерева");
```

```
obxod_binary_tree.setFillColor(text_color);
```

```
obxod_binary_tree.setCharacterSize(50);
```

```
obxod_binary_tree.setPosition(30, 10);
```

```
Text obxod_1_name;
```

```
obxod_1_name.setFont(font);
```

```
obxod_1_name.setString(L"Прямой обход");
```

```
obxod_1_name.setFillColor(text_color);
```

```
obxod_1_name.setCharacterSize(40);
```

```
obxod_1_name.setPosition(30, 80);
```

```
Text obxod_1_value;
```

```
obxod_1_value.setFont(font);
```

```
obxod_1_value.setString(vect_wstring[0]);
```

```
obxod_1_value.setFillColor(text_color);
```

```
obxod_1_value.setCharacterSize(40);
```

```
obxod_1_value.setPosition(30, 130);
```

```
Text obxod_2_name;
```

```
obxod_2_name.setFont(font);
```

```
obxod_2_name.setString(L"Обратный обход");
```

```
obxod_2_name.setFillColor(text_color);
```

```
obxod_2_name.setCharacterSize(40);
```

```
obxod_2_name.setPosition(30, 190);
```

```
Text obxod_2_value;
```

```
obxod_2_value.setFont(font);
```

```
obxod_2_value.setString(vect_wstring[1]);
```

```
obxod_2_value.setFillColor(text_color);
```

```
obxod_2_value.setCharacterSize(40);
```

```
obxod_2_value.setPosition(30, 240);
```

```
Text obxod_3_name;
```

```
obxod_3_name.setFont(font);
```

```
obxod_3_name.setString(L"Симметричный обход");
```

```
obxod_3_name.setFillColor(text_color);
```

```
obxod_3_name.setCharacterSize(40);
```

```
obxod_3_name.setPosition(30, 300);
```

```
Text obxod_3_value;
```

```
obxod_3_value.setFont(font);
```

```
obxod_3_value.setString(vect_wstring[2]);
```

```
obxod_3_value.setFillColor(text_color);
```

```
obxod_3_value.setCharacterSize(40);
```

```
obxod_3_value.setPosition(30, 350);
```

```
RectButton button1(sf::Vector2f(150, 60), sf::Vector2f(window.getSize().x / 2 - 75, 420));
```

```
button1.setButtonFont(font);
```

```
button1.setButtonLabel(L"Ok", text_color, 30);
```

```
while (window.isOpen()) {
```

```
    Vector2i mousePoz = Mouse::getPosition(window);//позиция мыши в окне
```

```
    sf::Event event;
```

```
    button1.getButtonStatus(window, event);
```



```

while (window.pollEvent(event))
{
    if (event.type == sf::Event::Closed)
        window.close();
    if (event.type == Event::MouseButtonPressed) {
        if (event.key.code == Mouse::Left) {
            if (button1.isPressed) {
                window.close();
            }
        }
    }
    window.clear(background_color);
    button1.draw(window);
    window.draw(obxod_binary_tree);
    window.draw(obxod_1_name);
    window.draw(obxod_1_value);
    window.draw(obxod_2_name);
    window.draw(obxod_2_value);
    window.draw(obxod_3_name);
    window.draw(obxod_3_value);
    window.display();
}
}

template <class T>
Tree<T>* add_node_to_tree_not_find(Tree<T>* tree) {
    string value = enter_the_data(L"Введи значение узла, куда необходимо добавить элемент");
    if (string_to_double_bool(value)) {
        double required_value = string_to_double(value);
        Tree<T>* indicator = tree->search_by_key(required_value);
        if (indicator == nullptr) {
            error_or_success_message(L"Такого элемента нет!", L"Ошибка");
        }
        else {
            value = enter_the_data(L"Введи значение, которое хотите добавить в дерево");
            if (string_to_double_bool(value)) {
                indicator->insert(string_to_double(value));
                error_or_success_message(L"Операция успешно завершена!", L"Успех!");
            }
            else {
                error_or_success_message(L"Это не число!", L"Ай-ай-ай");
            }
        }
    }
    else {
        error_or_success_message(L"Такого элемента нет!", L"Ай-ай-ай");
    }
    return tree;
}

template <class T>
Tree<T>* add_node_to_tree_find(Tree<T>* tree) {
    string value = enter_the_data(L"Введи значение, которое хотите добавить в дерево");
    if (string_to_double_bool(value)) {
        tree->insert(string_to_double(value));
        error_or_success_message(L"Операция успешно завершена!", L"Успех!");
    }
    else {
        error_or_success_message(L"Это не число!", L"Ай-ай-ай");
    }
    return tree;
}

```

```

}

template <class T>
Tree<T>* delete_node_from_tree(Tree<T>* tree) {
    string value = enter_the_data(L"Введи значение узла, которое хотите удалить");
    if (string_to_double_bool(value)) {
        Tree<T>* indicator = tree->search_by_key(string_to_double(value));
        if (indicator == nullptr) {
            error_or_success_message(L"Такого элемента нет!", L"Ошибка");
        }
        else {
            indicator->erase(string_to_double(value));
            error_or_success_message(L"Операция успешно завершена!", L"Успех");
        }
    }
    else {
        error_or_success_message(L"Такого элемента нет!", L"Ай-ай-ай");
    }
    return tree;
}

```

Файл sfml_button.hpp

```

#pragma once

#ifndef BUTTON_HPP_INCLUDED
#define BUTTON_HPP_INCLUDED
#include <SFML/Graphics.hpp>
#include <iostream>
#include <string>

const sf::Color defaultHovered = sf::Color(231, 221, 213); //цвет кнопки
const sf::Color defaultPressed = sf::Color(169, 105, 70); //цвет кнопки если она нажата

class Button{
public:
    virtual void getButtonStatus(sf::RenderWindow&, sf::Event&) = 0; //статус кнопки
    virtual void draw(sf::RenderWindow&) = 0; //отображение кнопки
    virtual void setButtonFont(sf::Font&); //шрифт текста на кнопке
    virtual void setButtonLabel(std::wstring, const sf::Color&, float) = 0; //установка надписи на кнопке

    bool isHover = false; //курсор наведен?
    bool isPressed = false; //нажата или нет
    bool isActive = true; //состояние кнопки

protected:
    sf::Text buttonLabel; //буквы на кнопке
    sf::Vector2i mousePosWindow; //позиция мыши
    sf::Vector2f mousePosView;
    sf::Vector2f buttonPos; //позиция кнопки
    sf::FloatRect labelRect;
    std::wstring label; //надпись
};

class RectButton : public Button{ //подкласс прямоугольных кнопок
public:
    RectButton(const sf::Vector2f = sf::Vector2f(0, 0)); //конструкторы
    RectButton(const sf::Vector2f = sf::Vector2f(0, 0), const sf::Vector2f = sf::Vector2f(0, 0));
    ~RectButton(); //деструктор

    void getButtonStatus(sf::RenderWindow&, sf::Event&); //статус кнопки (нажата/не нажата)

```

```

void draw(sf::RenderWindow&);//отображение кнопки
void setButtonLabel(std::wstring, const sf::Color&, float);//отображение надписи
sf::RectangleShape button;

private:
    sf::FloatRect buttonRect;
};
#endif

```

Файл RectButton.cpp

```

#include "sfml_button.hpp"

RectButton::RectButton(const sf::Vector2f size){//конструктор
    this->button.setSize(size);
    this->buttonRect = this->button.getLocalBounds();
}

RectButton::RectButton(const sf::Vector2f size, const sf::Vector2f position){//конструктор
    this->button.setSize(size);
    this->button.setPosition(position);
    this->buttonPos = position;
    this->buttonRect = this->button.getLocalBounds();
}

RectButton::~RectButton({}){//Деструктор

void RectButton::getButtonStatus(sf::RenderWindow& window, sf::Event& event)//статус
{
    this->mousePosWindow = sf::Mouse::getPosition(window);//позиция мыши в окне
    this->mousePosView = window.mapPixelToCoords(this->mousePosWindow);

    this->isHover = false;//курсор наведен?
    this->isPressed = false;//нажато?

    if (isActive){//если кнопка активна
        if (button.getGlobalBounds().contains(this->mousePosView)){//курсор наведен
            this->isHover = true;
        }
        if (button.getGlobalBounds().contains(this->mousePosView)){//после 1 нажатия кнопки
            this->isPressed = true; //состояние меняется на активное - кнопка нажата
        }

        if (isHover){//курсор наведен
            button.setFillColor(defaultPressed);//меняю цвет при наведении
        }
        else button.setFillColor(defaultHovered);//обычный цвет

        if (isPressed){//кнопка нажата
            button.setFillColor(defaultPressed);//возвращаю истинный цвет
        }
    }
    else{//обычный цвет
        button.setFillColor(defaultPressed);
    }
}

void RectButton::draw(sf::RenderWindow& window){ //отображаю кнопку в окне
    window.draw(button);//кнопка
    window.draw(buttonLabel);//надпись
}

```

```

void RectButton::setButtonLabel(std::wstring label, const sf::Color& color, float charSize){//устанавливаю надпись
    this->buttonLabel.setString(label);//надпись
    this->buttonLabel.setCharacterSize(charSize);//размер символов
    this->buttonLabel.setFillColor(color);//устанавливаю цвет
    this->label = label;//присваиваю надпись

    this->labelRect = this->buttonLabel.getLocalBounds();
    this->buttonLabel.setOrigin(this->labelRect.width / 2.0f, this->labelRect.height / 2.0f);//ставлю координаты

    this->buttonLabel.setPosition(this->buttonPos.x + (this->buttonRect.width / 2.0f),//ставлю координаты
        this->buttonPos.y + (this->buttonRect.height / 4.0f) + 7);
}

```

Файл textbox.hpp

```

#include<SFML/Graphics.hpp>
#ifndef SDX_TEXTBOX
#define SDX_TEXTBOX

namespace sdx {
    class TextBox {
    private:
        sf::RectangleShape outerRect;
        sf::RectangleShape innerRect;
        sf::RectangleShape blinker;
        sf::String getPinp;
        sf::String txtInp;
        sf::Clock clock;
        sf::Time time;

        unsigned int textSize;
        unsigned int focusChar;
        float charWidth;
        float thickness;
        float posX;
        float posY;
        float height;
        float width;
        bool focus;
    public:
        class Text {
        private:
            sf::Font font;
            sf::Text text;
        public:
            Text(sf::String, float, float); //конструктор, первый параметр - текстовая строка, второй - позиция x, третий -
            позиция y
            sf::Text get(); //возвращает класс sf::Text, который можно отрисовать
            void setText(sf::String); //установка текста
            void setPosition(float, float); //позиция текста
            void setSize(unsigned int); //размер текста
        };
        TextBox(); //конструктор
        TextBox(float, float, float, float, float); //первые два параметра задают размер, вторые два - положение, а
        последний - толщину
        void draw(sf::RenderWindow&); //отрисовка
        void handleEvent(sf::Event&); //обрабатывает ввод текста
        sf::String getCurrentText(); //получаю то, что в текстовом поле
    public:
        void setSize(float, float); //размер окна обновления - первый параметр для x, второй для y
        void setPosition(float, float); //ставлю позицию (x,y)
        void setBorder(float); //ставлю толщину границы
    private:

```

```

    Text inpText;
};
}
#endif

```

Файл textbox.cpp

```

#include "textbox.hpp"

namespace sdx {
    TextBox::TextBox(sf::String string, float x, float y) {
        font.loadFromFile("monospace.ttf");
        text.setFont(font);
        text.setString(string);
        text.setFillColor(sf::Color::Black);
        text.setCharacterSize(18);
        text.setPosition(sf::Vector2f(x, y));
        text.setLineSpacing(0);
        text.setOutlineThickness(0);
    }

    sf::Text TextBox::Text::get() { return text; }
    void TextBox::Text::setText(sf::String string) { text.setString(string); }
    void TextBox::Text::setPosition(float x, float y) { text.setPosition(sf::Vector2f(x, y)); }
    void TextBox::Text::setSize(unsigned int x) { text.setCharacterSize(x); }

    TextBox::TextBox() : inpText("", 6, 5) {
        outerRect.setSize(sf::Vector2f(460, 32));
        innerRect.setSize(sf::Vector2f(456, 28));
        outerRect.setPosition(sf::Vector2f(0, 0));
        innerRect.setPosition(sf::Vector2f(2, 2));
        outerRect.setFillColor(sf::Color::Black);
        innerRect.setFillColor(sf::Color::White);

        blinker.setSize(sf::Vector2f(1, 26));
        blinker.setPosition(sf::Vector2f(4, 3));
        blinker.setFillColor(sf::Color::Black);

        time = sf::Time::Zero;
        textSize = 18;
        getPinp = "";
        txtInp = "";
        thickness = 2;
        posX = 0;
        posY = 0;
        height = 32;
        width = 460;
        focusChar = 0;
        focus = false;
        charWidth = 0;
    }

    TextBox::TextBox(float x1, float x2, float y1, float y2, float z) : inpText("", y1 + z + 2, y2 + z - 1) {
        outerRect.setSize(sf::Vector2f(x1, x2));
        innerRect.setSize(sf::Vector2f(x1 - 2 * z, x2 - 2 * z));
        outerRect.setPosition(sf::Vector2f(y1, y2));
        innerRect.setPosition(sf::Vector2f(y1 + z, y2 + z));
        outerRect.setFillColor(sf::Color::Black);
        innerRect.setFillColor(sf::Color::White);

        blinker.setSize(sf::Vector2f(1, x2 - 2 * z - 2));
        blinker.setPosition(sf::Vector2f(y1 + z + 2, y2 + z + 1));
        blinker.setFillColor(sf::Color::Black);
    }
}

```

```

time = sf::Time::Zero;
textSize = (unsigned int)(x2 - 4 - 2 * z);
getPinp = "";
txtInp = "";
thickness = z;
posX = y1;
posY = y2;
height = x2;
width = x1;
focusChar = 0;
focus = false;
charWidth = 0;
inpText.setSize(textSize);
}

void TextBox::setSize(float x, float y) {
    height = y;
    width = x;
    textSize = (unsigned int)(y - 4 - 2 * thickness);
    outerRect.setSize(sf::Vector2f(x, y));
    innerRect.setSize(sf::Vector2f(x - 2 * thickness, y - 2 * thickness));
    blinker.setSize(sf::Vector2f(1, y - 2 * thickness - 2));
    inpText.setSize(textSize);
    inpText.setPosition(posX + thickness + 2, posY + thickness - 1);
}

void TextBox::setPosition(float x, float y) {
    posX = x;
    posY = y;
    outerRect.setPosition(sf::Vector2f(x, y));
    innerRect.setPosition(sf::Vector2f(x + thickness, y + thickness));
    blinker.setPosition(sf::Vector2f(x + thickness + 2, y + thickness + 1));
    inpText.setPosition(x + thickness + 2, y + thickness - 1);
}

void TextBox::setBorder(float x) {
    thickness = x;
    textSize = (unsigned int)(height - 4 - 2 * x);
    innerRect.setSize(sf::Vector2f(width - 2 * x, height - 2 * x));
    inpText.setSize(textSize);
    setPosition(posX, posY);
}

sf::String TextBox::getCurrentText() { return getPinp; }

void TextBox::handleEvent(sf::Event& event) {
    if (event.type == sf::Event::TextEntered && focus) {
        if ((inpText.get().findCharacterPos(focusChar).x + 1.2 * textSize) < (width + posX) && 31 < int(event.text.unicode)
        && 256 > int(event.text.unicode)) {
            if (focusChar == getPinp.getSize()) getPinp += event.text.unicode;
            else {
                getPinp = getPinp.substr(0, focusChar) + event.text.unicode + getPinp.substr(focusChar,
                getPinp.getSize() - focusChar);
            }
            focusChar++;
        }
    }
    if (event.type == sf::Event::KeyPressed && focus) {
        if (event.key.code == sf::Keyboard::BackSpace) {
            if (focusChar != 0) {
                getPinp.erase(focusChar - 1, 1);
            }
        }
    }
}

```

```

        if (focusChar > 0) focusChar--;
    }
}
if (event.key.code == sf::Keyboard::Delete) {
    if (focusChar != getPinp.getSize()) {
        getPinp.erase(focusChar);
    }
}
else if (event.key.code == sf::Keyboard::Left) {
    if (focusChar > 0) { focusChar--; }
}
else if (event.key.code == sf::Keyboard::Right) {
    if (focusChar < getPinp.getSize()) focusChar++;
}
}
if (event.type == sf::Event::MouseButtonPressed) {
    if (event.mouseButton.button == sf::Mouse::Left) {
        if (getPinp.getSize() > 0) {
            if (charWidth == 0) charWidth = inpText.get().findCharacterPos(1).x - inpText.get().findCharacterPos(0).x;
            unsigned int temp = (unsigned int)((event.mouseButton.x - posX) / charWidth);
            if (temp > getPinp.getSize()) focusChar = getPinp.getSize();
            else focusChar = temp;
        }
        if (event.mouseButton.x > posX && event.mouseButton.x < posX + width && event.mouseButton.y > posY &&
event.mouseButton.y < posY + height) focus = true;
        else focus = false;
    }
}
}
}

void TextBox::draw(sf::RenderWindow& window) {
    time += clock.restart();
    if (focus) {
        if (time.asSeconds() > 1) {
            time = sf::Time::Zero;
            blinker.setFillColor(sf::Color::Black);
        }
        else if (time.asSeconds() > 0.5) blinker.setFillColor(sf::Color::White);
    }
    else {
        blinker.setFillColor(sf::Color::White);
        if (time.asSeconds() > 300) time = sf::Time::Zero;
    }
    if (focusChar == 0) blinker.setPosition(posX + thickness + 2, posY + thickness + 1);
    else blinker.setPosition(sf::Vector2f(inpText.get().findCharacterPos(focusChar).x, posY + thickness + 1));
    inpText.setText(getPinp);
    window.draw(outerRect);
    window.draw(innerRect);
    window.draw(blinker);
    window.draw(inpText.get());
}
}

```

Файл other functions.h

```

#pragma once
#include <SFML/Graphics.hpp>
#include <iostream>
#include <vector>
#include "Tree.h"
#include "textbox.hpp"
#include "sfml_button.hpp"

```

```

#include <sstream>

using namespace sf;
using namespace std;

sf::Color background_color(236, 205, 177); //фон
sf::Color button_color(231, 221, 213); //кнопка
sf::Color button_press_color(169, 105, 70); //кнопка нажата
sf::Color text_color(39, 16, 7); //текст

void print_spaces(int start, int end) {
    for (int j = start; j < end; j++) { //пробелы до узла
        cout << " ";
    }
}

double string_to_double(const std::string& s) { //из строки делаю double
    std::istringstream i(s);
    double x;
    if (!(i >> x))
        return -1;
    return x;
}

bool string_to_double_bool(const std::string& s) { //проверяю можно ли сделать double из string
    std::istringstream i(s);
    double x;
    if (!(i >> x))
        return false;
    return true;
}

string enter_the_data(wstring mes) { //получение данных от пользователя
    RenderWindow window(VideoMode(700, 350), L"Ведите...", Style::Titlebar | Style::Close);

    Font font;
    font.loadFromFile("ofont.ru_American TextC.ttf"); //загружаю шрифт

    RectButton button_1(sf::Vector2f(340, 40), sf::Vector2f(325, 240));
    button_1.setButtonFont(font);
    button_1.setButtonLabel(L"Продолжить", text_color, 30);

    Text text_mes;
    text_mes.setFont(font);
    text_mes.setString(mes);
    text_mes.setFillColor(text_color);
    text_mes.setCharacterSize(30);
    text_mes.setPosition(30, 70);

    sdx::TextBox::Text text("", 124, 220); //Текстовый бокс
    text.setSize(20);
    sdx::TextBox textBox(440, 32, 130, 160, 2);

    while (window.isOpen()) {
        Vector2i mousePoz = Mouse::getPosition(window); //позиция мыши в окне
        sf::Event event;
        button_1.getButtonStatus(window, event);
        while (window.pollEvent(event)) {
            textBox.handleEvent(event);

            if (event.type == sf::Event::Closed) {
                window.close();
            }
        }
    }
}

```



```

    }
    if (event.type == Event::MouseButtonPressed) {
        if (event.key.code == Mouse::Left) {
            if (button_1.isPressed) {
                window.close();
            }
        }
    }
}

window.clear(background_color);
button_1.draw(window);
textBox.draw(window);
window.draw(text_mes);
window.display();
}
return textBox.getCurrentText();//извлекаю введенный текст
}

void error_or_success_message(std::wstring message, std::wstring title) { //сообщение о выполнении операции
    RenderWindow window(sf::VideoMode(message.size() * 20, 250), title);

    Font font;
    font.loadFromFile("ofont.ru_American TextC.ttf");//загружаю шрифт

    Text mes;
    mes.setFont(font);
    mes.setString(message);
    mes.setFillColor(Color::Black);
    mes.setCharacterSize(40);
    mes.setPosition(30, 45);

    RectButton button1(sf::Vector2f(150, 60), sf::Vector2f(window.getSize().x / 2 - 75, 140)); //Вертикальная печать
    дерева
    button1.setButtonFont(font);
    button1.setButtonLabel(L"Ok", sf::Color::Black, 30);

    while (window.isOpen())
    {
        Vector2i mousePoz = Mouse::getPosition(window); //позиция мыши в окне
        sf::Event event;
        button1.getButtonStatus(window, event);

        while (window.pollEvent(event))
        {
            if (event.type == sf::Event::Closed)
                window.close();
            if (event.type == Event::MouseButtonPressed) {
                if (event.key.code == Mouse::Left) {
                    if (button1.isPressed) {
                        window.close();
                    }
                }
            }
        }
    }

    window.clear(background_color);

    button1.draw(window);
    window.draw(mes);

    window.display();
}

```

```
}
```

Файл Source.cpp

```
#include <SFML/Graphics.hpp>
#include <iostream>
#include "Tree.h"
#include "other functions.h"
#include "textbox.hpp"
#include "sfml_button.hpp"

using namespace std;

sf::Font jackInput;

int main(){
    system("chcp 1251 > Null");
    sf::RenderWindow window(sf::VideoMode(750, 570), "Binary tree - menu");//главное окно
    jackInput.loadFromFile("ofont.ru_American TextC.ttf");

    Text menu;
    menu.setFont(jackInput);
    menu.setString(L"Меню");
    menu.setFillColor(Color(39, 16, 7));
    menu.setCharacterSize(40);
    menu.setPosition(320, 10);

    RectButton button_1(sf::Vector2f(340, 40), sf::Vector2f(205, 70));//Вертикальная печать дерева
    button_1.setButtonFont(jackInput);
    button_1.setButtonLabel(L"Вертикальная печать дерева", text_color, 30);

    RectButton button_2(sf::Vector2f(340, 40), sf::Vector2f(205, 120));//Горизонтальная печать дерева
    button_2.setButtonFont(jackInput);
    button_2.setButtonLabel(L"Горизонтальная печать дерева", text_color, 30);

    RectButton button_3(sf::Vector2f(340, 40), sf::Vector2f(205, 170)); //Обходы бинарного дерева
    button_3.setButtonFont(jackInput);
    button_3.setButtonLabel(L"Обходы бинарного дерева", text_color, 30);

    RectButton button_4(sf::Vector2f(340, 40), sf::Vector2f(205, 220)); //Вставка узла в дерево
    button_4.setButtonFont(jackInput);
    button_4.setButtonLabel(L"Добавить узел в дерево", text_color, 30);

    RectButton button_5(sf::Vector2f(340, 40), sf::Vector2f(205, 270)); //Удаление узла из дерева
    button_5.setButtonFont(jackInput);
    button_5.setButtonLabel(L"Удалить узел из дерева", text_color, 30);

    RectButton button_6(sf::Vector2f(340, 40), sf::Vector2f(205, 320)); //теперь это дерево поиска
    button_6.setButtonFont(jackInput);
    button_6.setButtonLabel(L"Сделать деревом поиска", text_color, 30);

    RectButton button_7(sf::Vector2f(340, 40), sf::Vector2f(205, 370)); //Найти минимальный элемент
    button_7.setButtonFont(jackInput);
    button_7.setButtonLabel(L"Найти минимальный элемент", text_color, 30);

    RectButton button_8(sf::Vector2f(340, 40), sf::Vector2f(205, 420)); //Найти элемент по ключу
    button_8.setButtonFont(jackInput);
    button_8.setButtonLabel(L"Найти элемент по ключу", text_color, 30);

    RectButton button_exit(sf::Vector2f(340, 40), sf::Vector2f(205, 500)); //Найти элемент по ключу
    button_exit.setButtonFont(jackInput);
    button_exit.setButtonLabel(L"Выход", text_color, 30);
```

```

bool flag = false;

Tree<double>* root_1 = new Tree<double>;

vector<double> vect_node{ 20,14,2,78,56,3,-10};
root_1 = root_1->build_balanced_bst(vect_node, 0, vect_node.size() - 1);//строю дерево

while (window.isOpen())
{
    Vector2i mousePoz = Mouse::getPosition(window);//позиция мыши в окне

    sf::Event event;

    button_1.getButtonStatus(window, event);
    button_2.getButtonStatus(window, event);
    button_3.getButtonStatus(window, event);
    button_4.getButtonStatus(window, event);
    button_5.getButtonStatus(window, event);
    button_6.getButtonStatus(window, event);
    if (flag) //если дерево поиска
        button_7.getButtonStatus(window, event);
        button_8.getButtonStatus(window, event);
    }
    button_exit.getButtonStatus(window, event);

    while (window.pollEvent(event)) {
        if (event.type == sf::Event::Closed)
            window.close();
        if (event.type == Event::MouseButtonPressed) {
            if (event.key.code == Mouse::Left) {
                if (button_1.isPressed) {
                    root_1->Draw(L"Вертикальный вывод дерева", true);
                }
                else if (button_2.isPressed) {
                    root_1->Draw(L"Горизонтальный вывод дерева", false);
                }
                else if (button_3.isPressed) {
                    tree_traversal(root_1);
                }
                else if (button_4.isPressed) {
                    if (!flag) {
                        root_1 = add_node_to_tree_not_find(root_1);
                    }
                    else {
                        root_1 = add_node_to_tree_find(root_1);
                    }
                }
                else if (button_5.isPressed) {
                    root_1 = delete_node_from_tree(root_1);
                }
                else if (button_6.isPressed) {
                    root_1 = root_1->create_bst_from_balanced_tree(root_1);
                    flag = true;
                    error_or_success_message(L"Операция успешно завершена!", L"Успех");
                }
                else if (button_7.isPressed) {
                    root_1->find_by_min();
                }
                else if (button_8.isPressed) {
                    root_1->find_node_by_key();
                }
            }
        }
    }
}

```

```

        else if (button_exit.isPressed) {
            window.close();
        }
    }
}

}
window.clear(background_color);

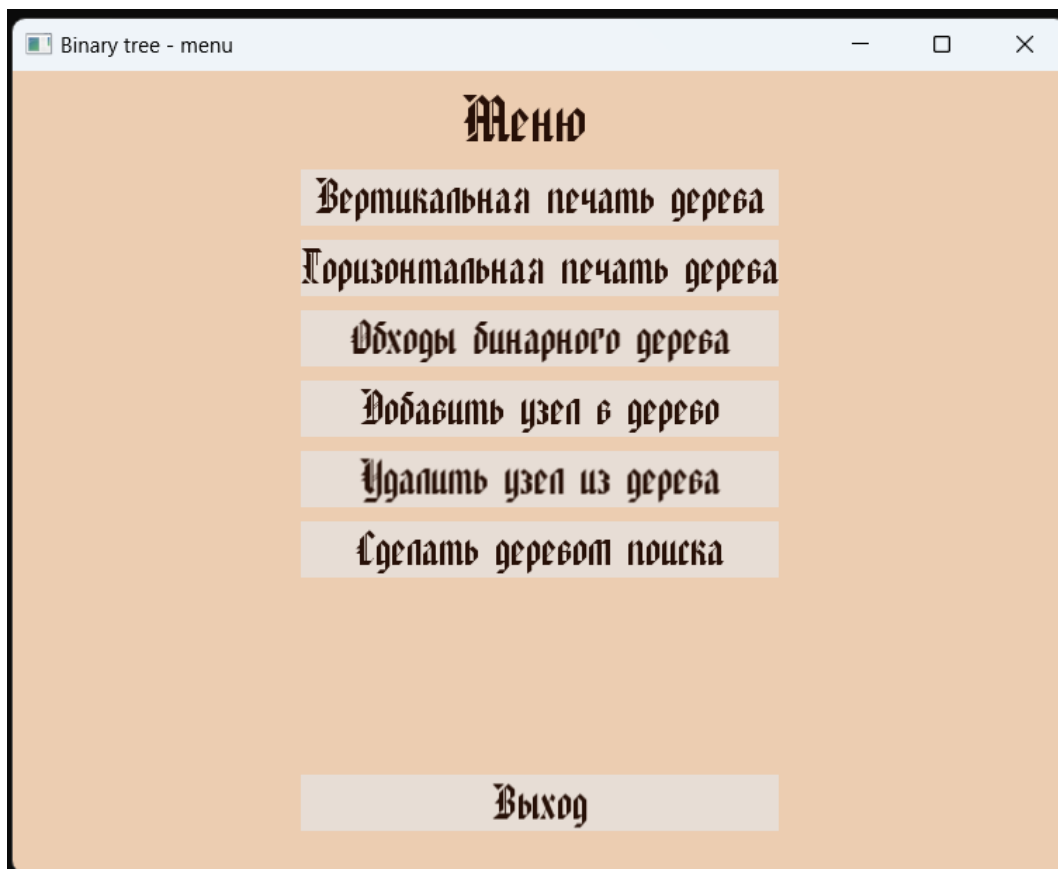
window.draw(menu);
button_1.draw(window);
button_2.draw(window);
button_3.draw(window);
button_4.draw(window);
button_5.draw(window);
button_6.draw(window);
if (flag) { //если дерево поиска
    button_7.draw(window);
    button_8.draw(window);
}
button_exit.draw(window);

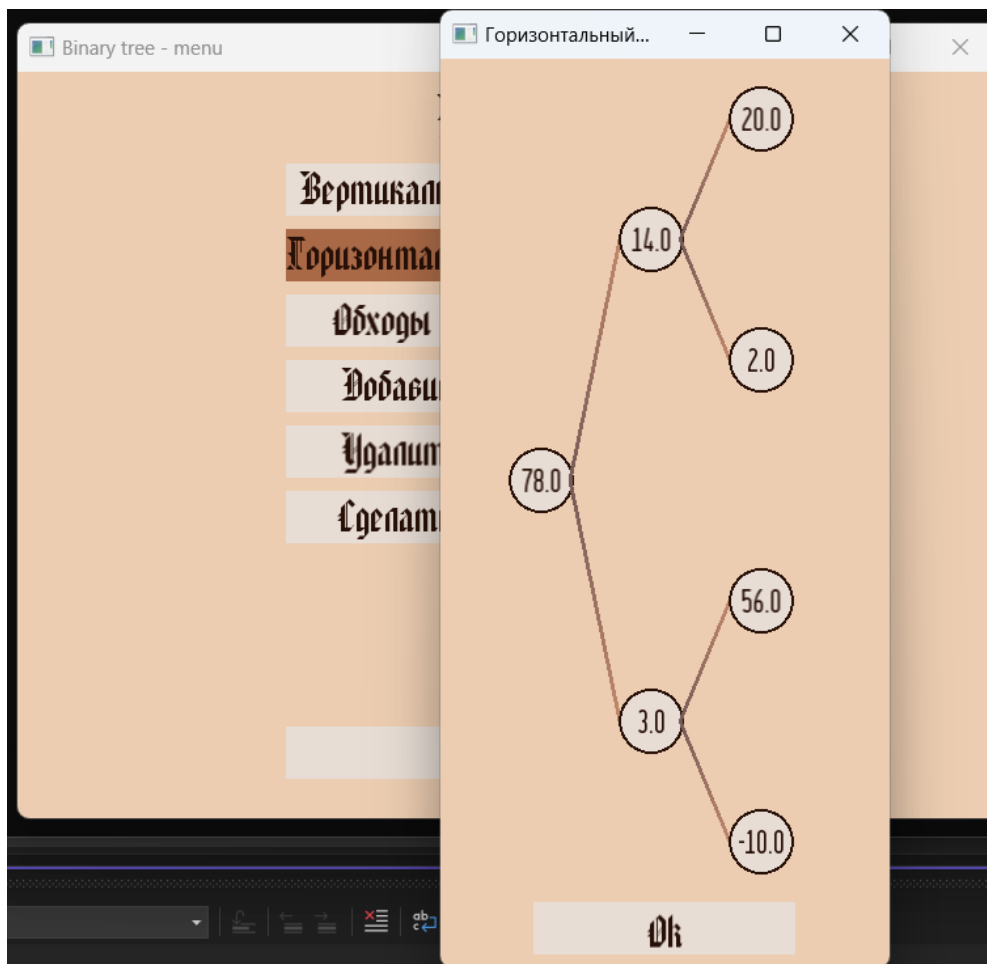
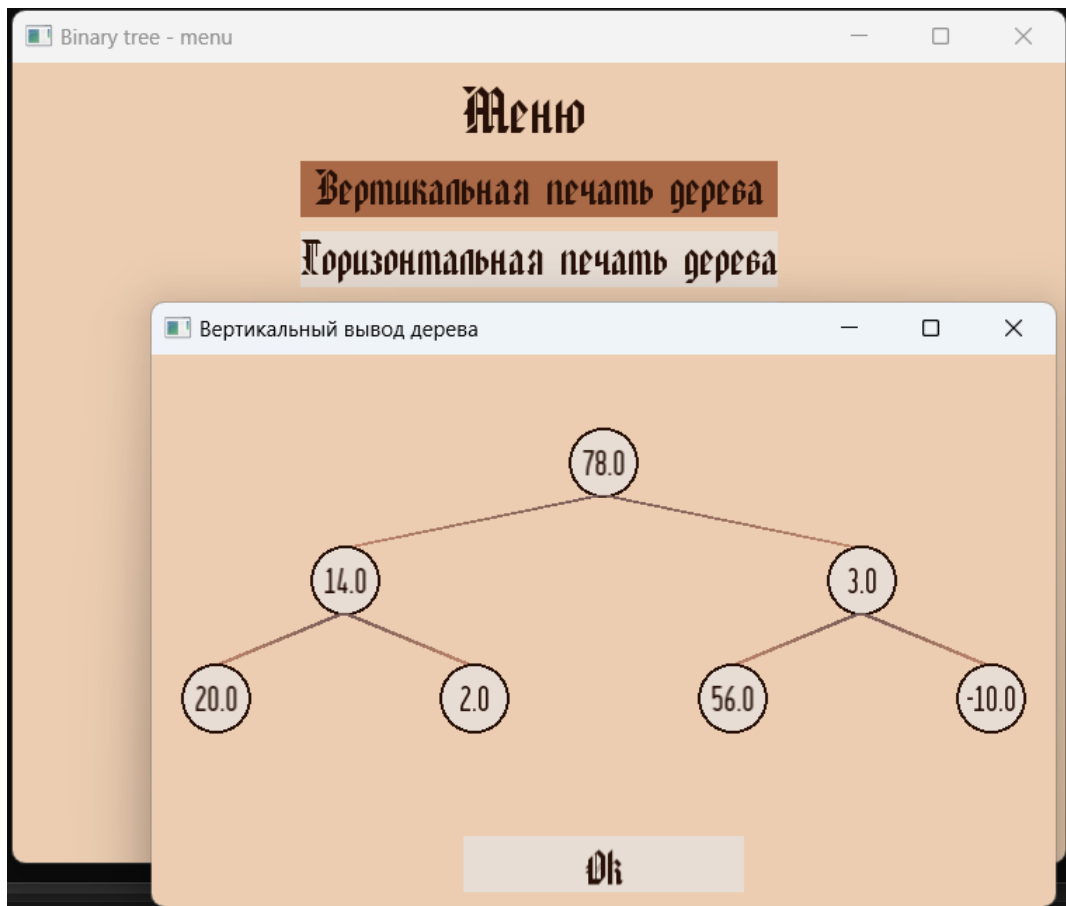
window.display();
}

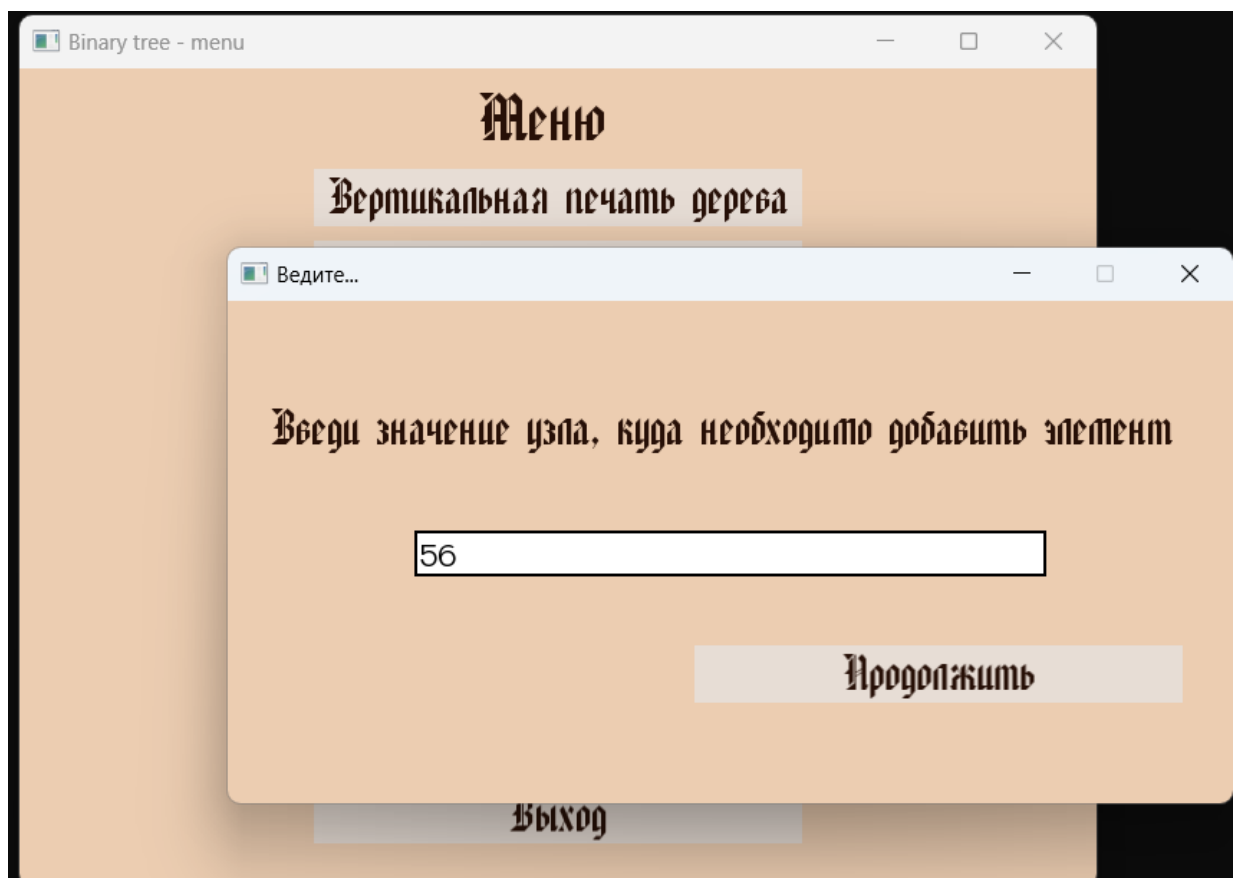
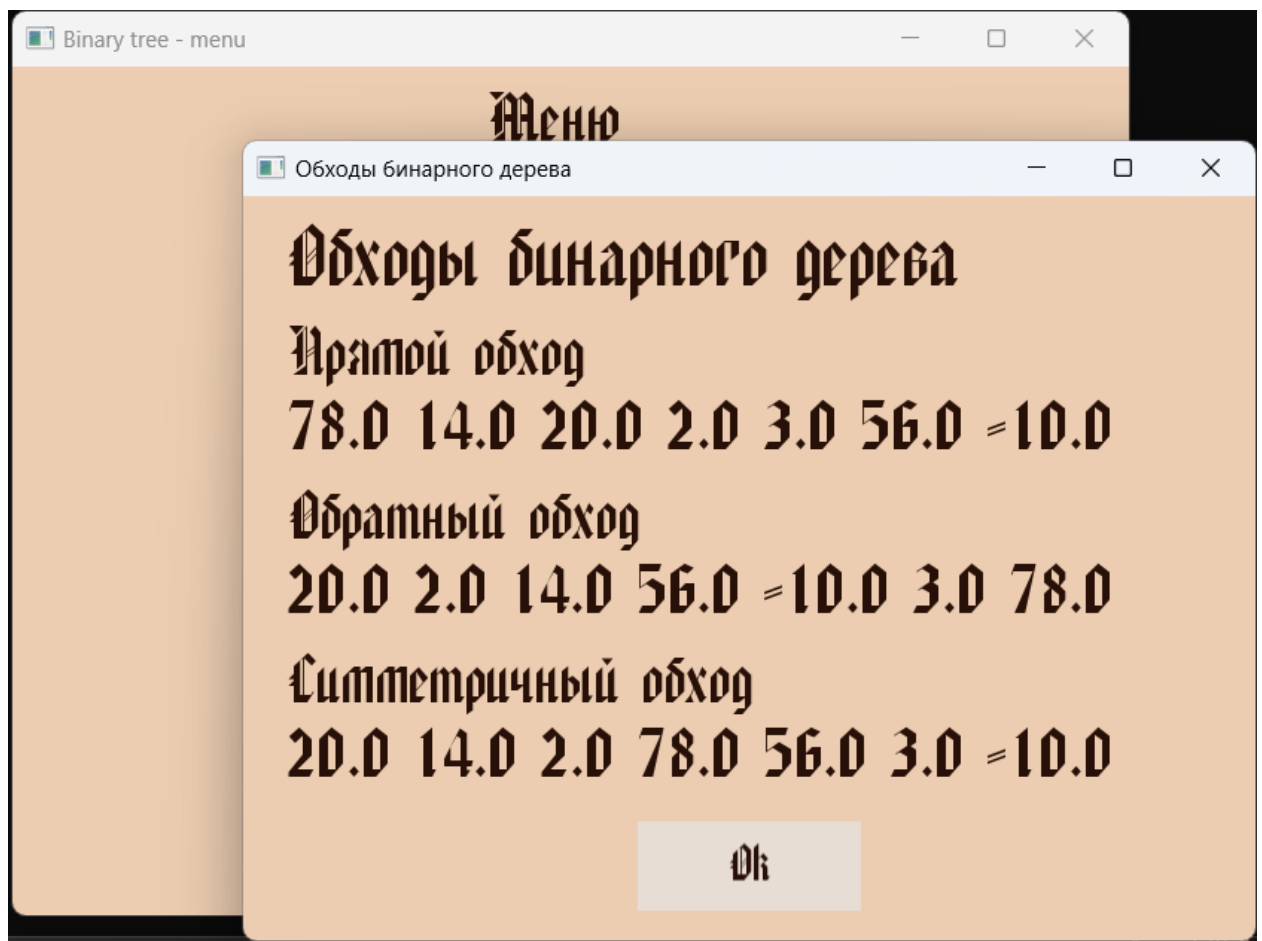
return 0;
}

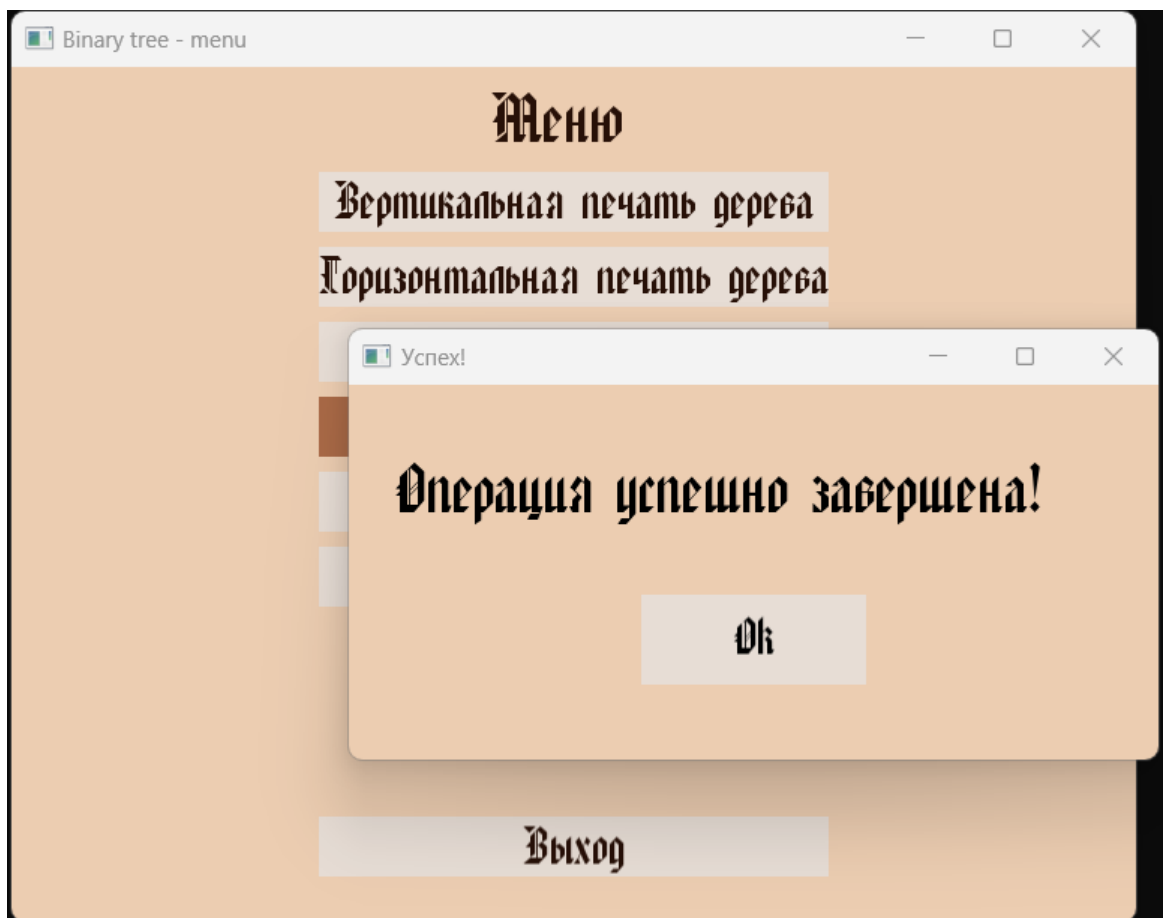
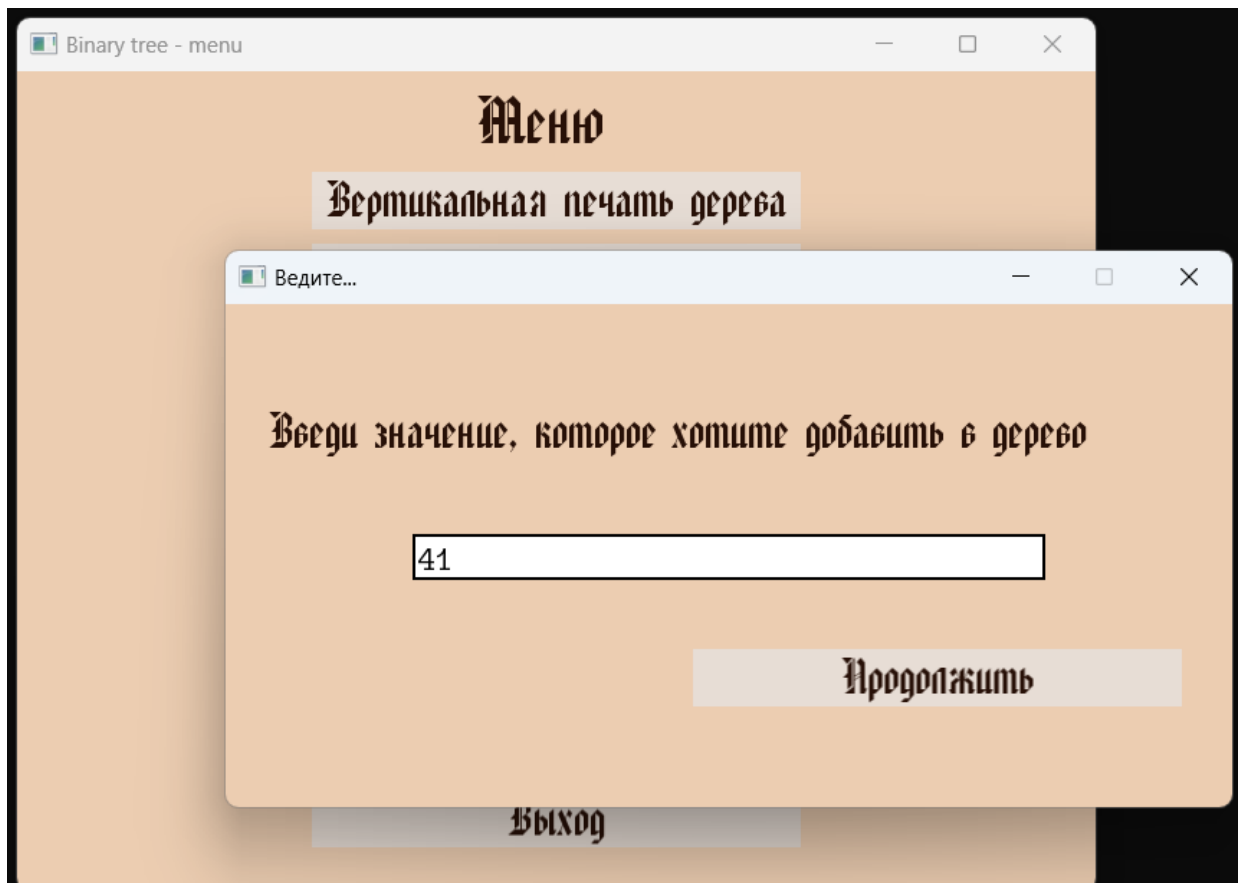
```

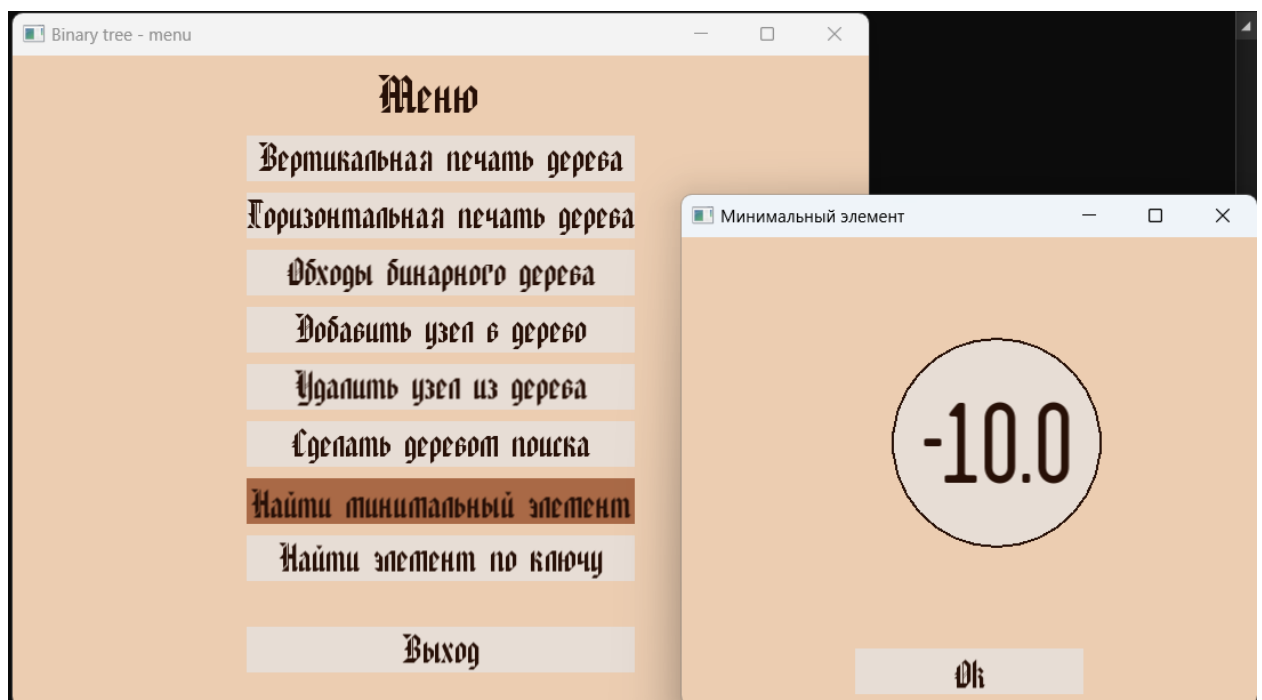
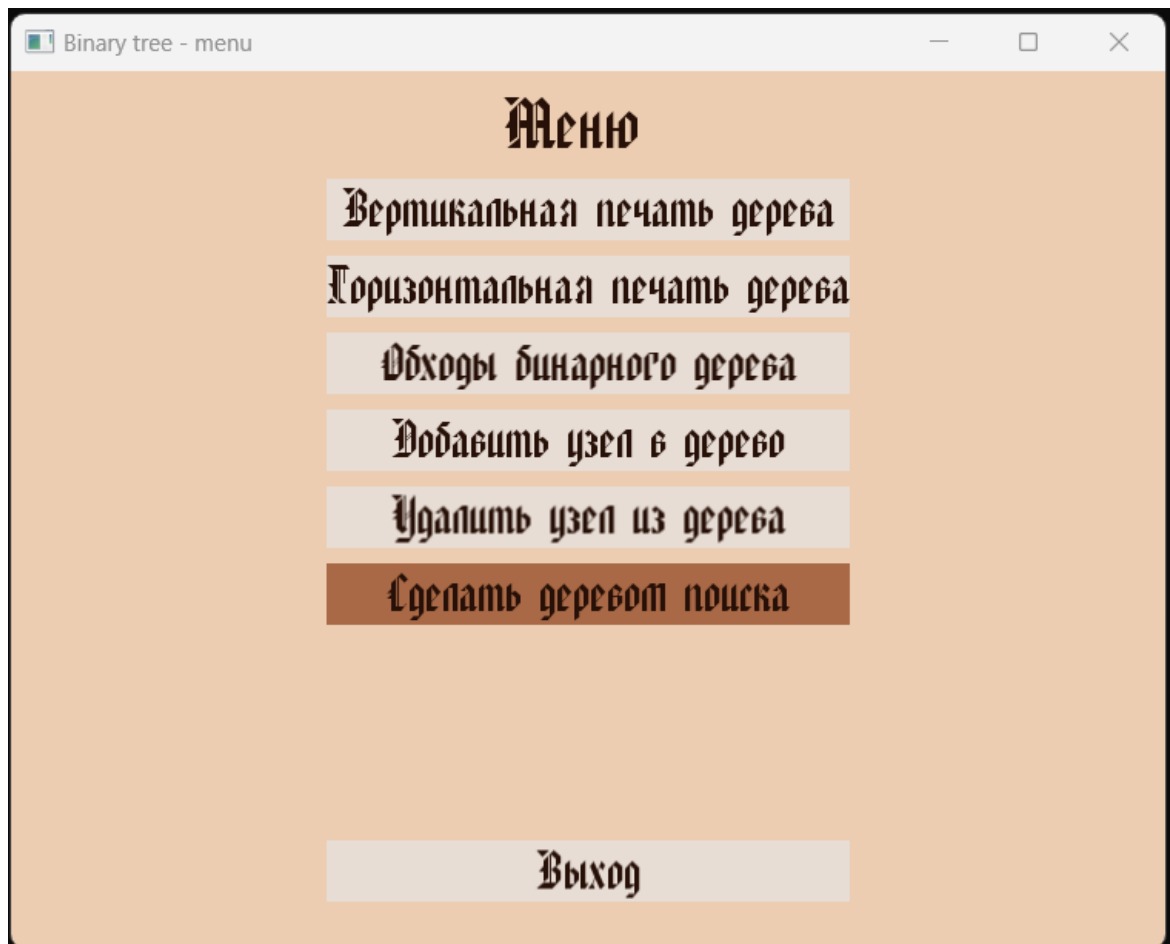
Результаты работы



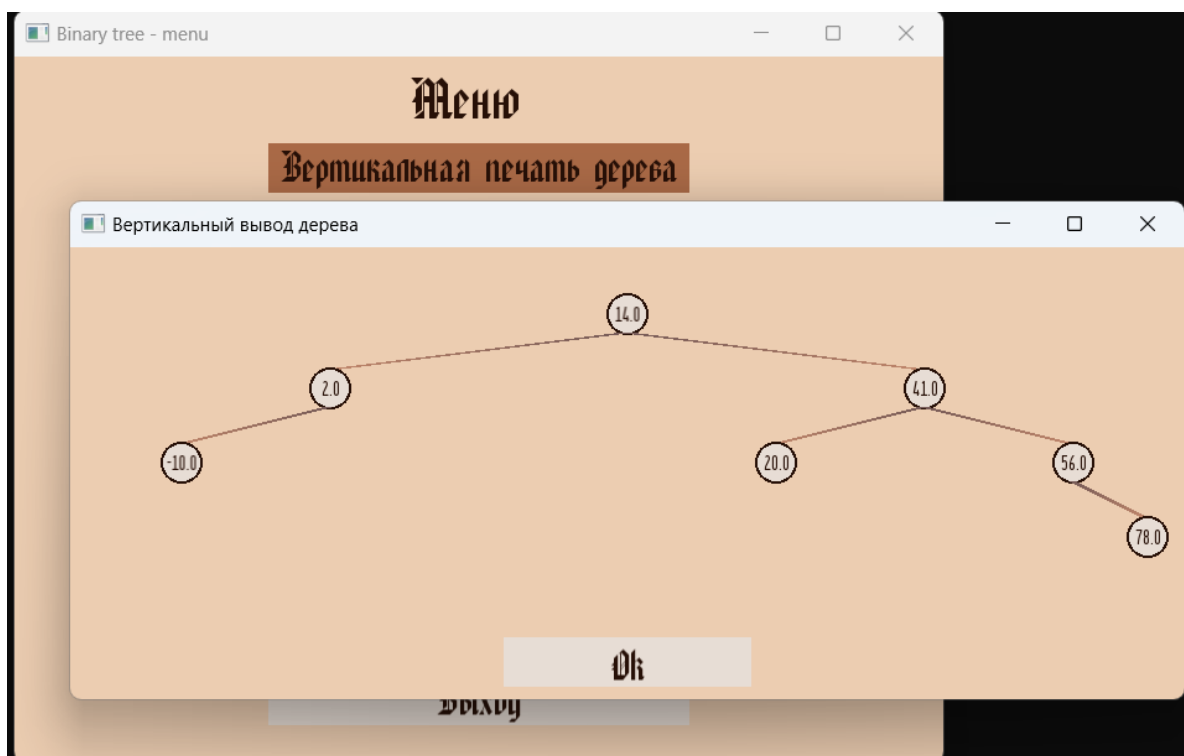
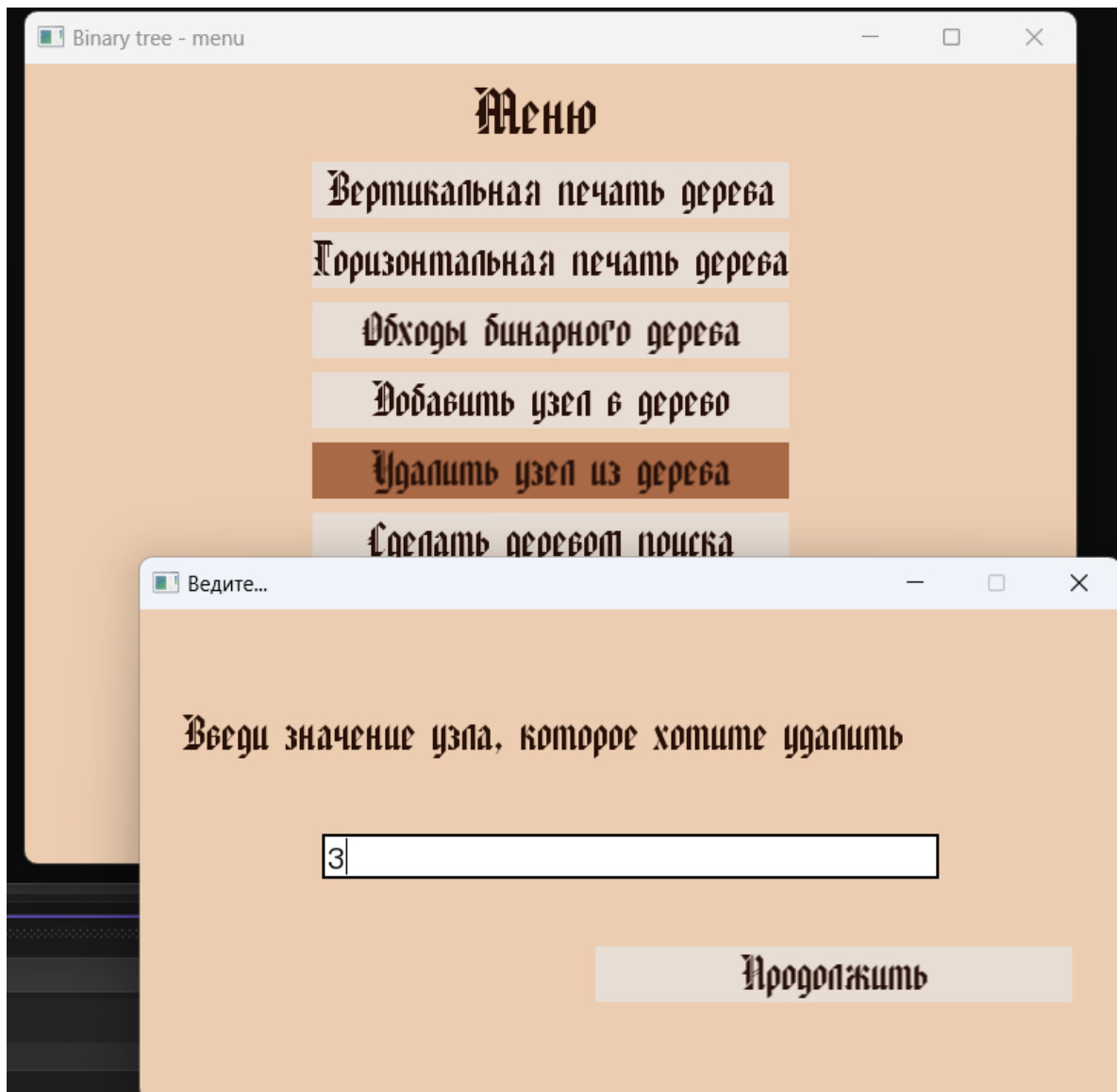


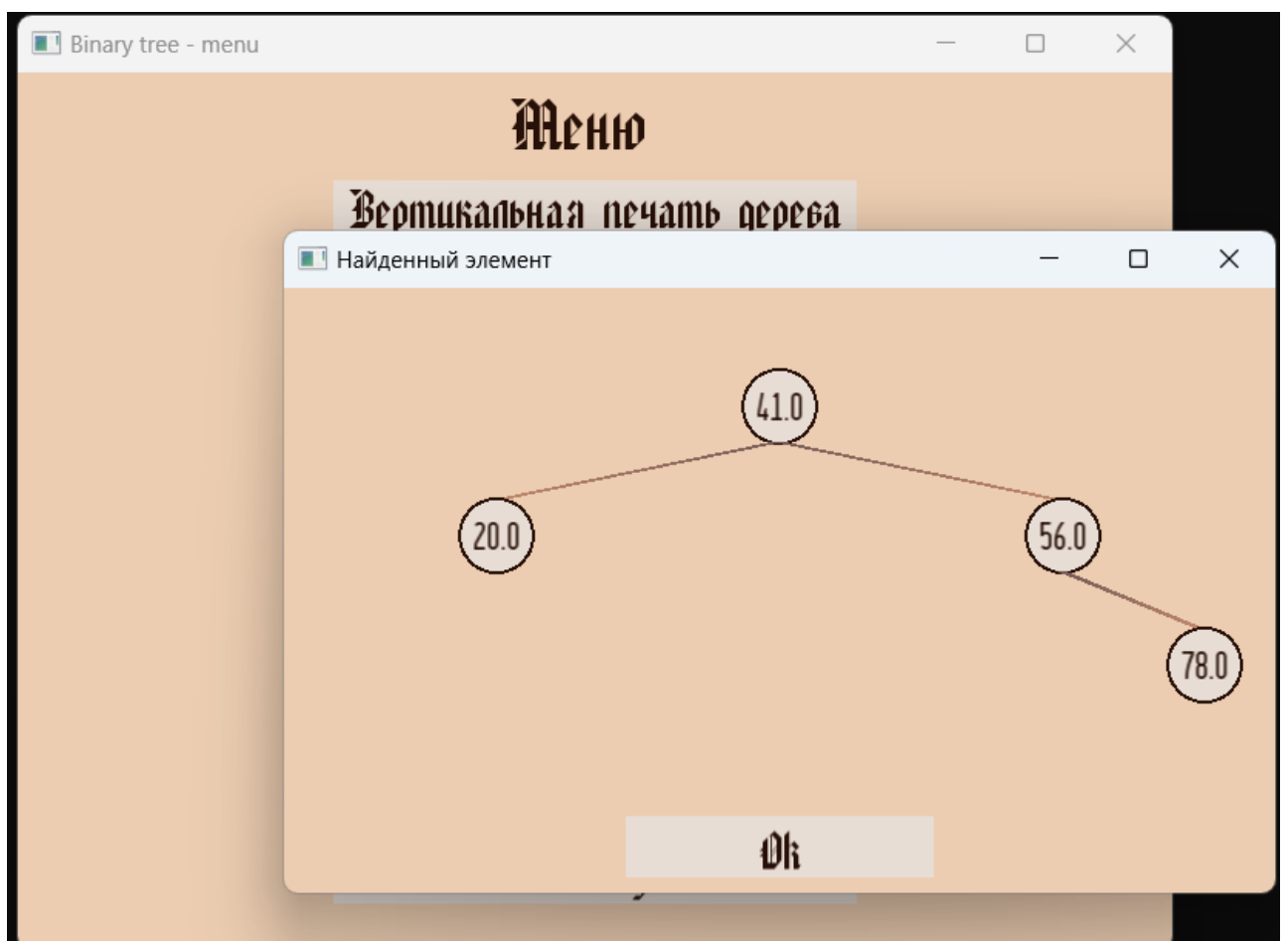
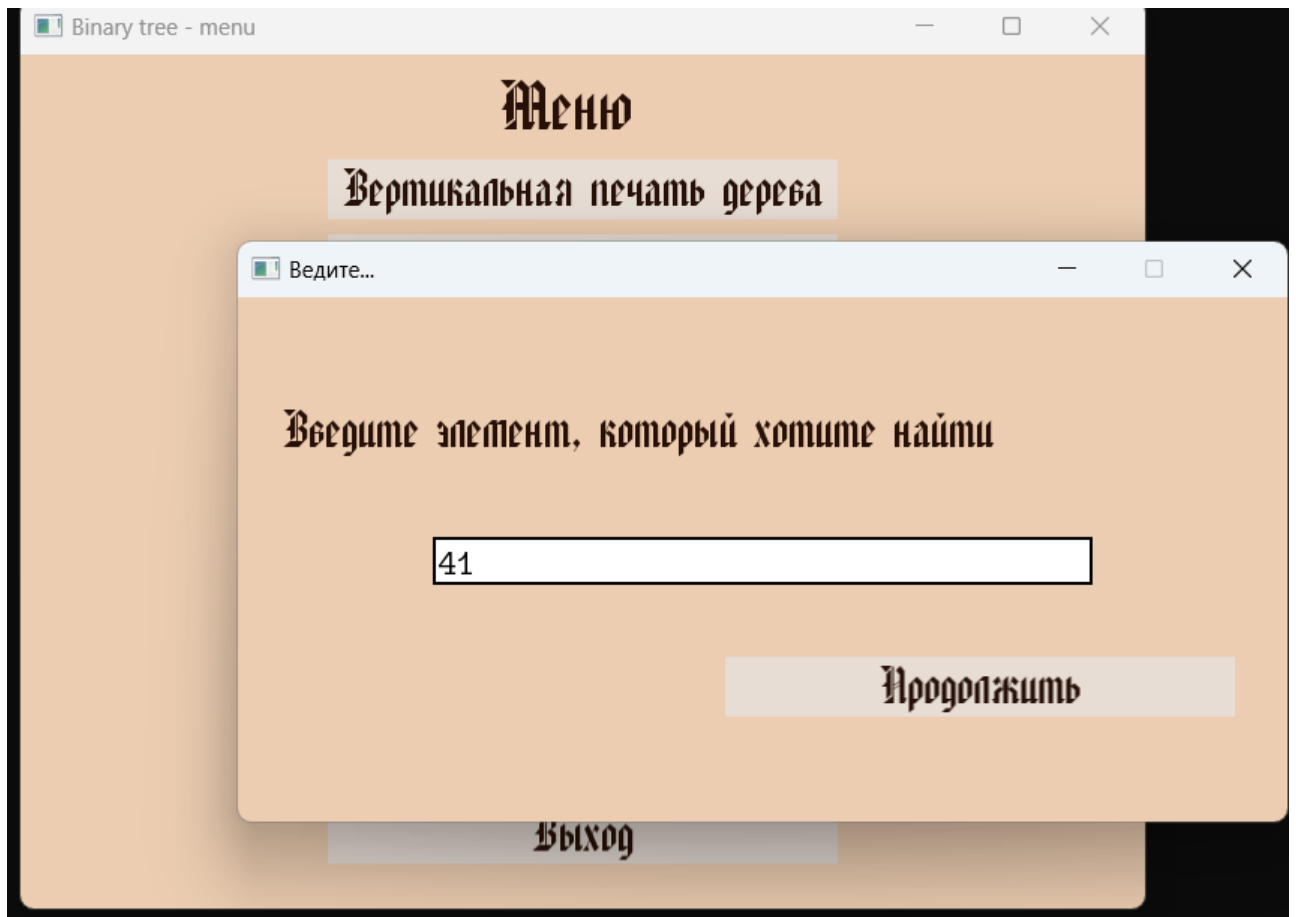












Вывод

















В ходе работы я применила знания о работе с классами, и интерфейсами. По ходу работы было разработано бинарное дерево, операции с которым выполняются посредством с интерфейсом, разработанным с помощью средств SFML. Были реализованы разные методы обхода дерева, функции печати дерева (вертикальная и горизонтальная), функции добавления и удаление узла дерева, а также функции поиска необходимого элемента. В коде были реализованы особые классы, которые реализуют кнопки и текстовые боксы для упрощения реализации интерфейса. По итогу работы было реализовано бинарное дерево, с меню, которое позволяет управлять им.

GitHub

Ссылка: <https://github.com/SonyAkb/Laboratory-works-for-the-2-semester/tree/main/Binary%20tree>



SonyAkb Add files via upload

| Name |
|---|
|  .. |
|  README |
|  RectButton.cpp |
|  Tree.h |
|  main.cpp |
|  monospace.ttf |
|  ofont.ru_American TextC.ttf |
|  ofont.ru_Dealerplate.ttf |
|  other functions.h |
|  sfml_button.cpp |
|  sfml_button.hpp |
|  textbox.cpp |
|  textbox.hpp |
|  uml tree.drawio.png |
|  uml tree_2.drawio.png |
|  uml tree_3.drawio.png |