

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

КУРСОВАЯ РАБОТА
по дисциплине «Алгоритмы и структуры данных»
Тема: Демонстрация красно-чёрного дерева

Студент гр. 9304

Тиняков С.А.

Преподаватель

Филатов Ар.Ю.

Санкт-Петербург

2020

ЗАДАНИЕ НА КУРСОВУЮ РАБОТУ

Студент Тиняков С.А.

Группа 9304

Тема работы: Демонстрация красно-чёрного дерева

Исходные данные:

В режиме в демонстрации — отсутствуют

В режиме тестирования: 1) количество элементов; 2) элементы ;3) элемент для поиска

Содержание пояснительной записки:

- ◆ Аннотация
- ◆ Содержание
- ◆ Введение
- ◆ Формальная постановка задачи
- ◆ Структура данных красно-чёрное дерево
- ◆ Описание структур данных и функций
- ◆ Описание интерфейса пользователя
- ◆ Тестирование
- ◆ Заключение
- ◆ Список использованных источников

Предполагаемый объем пояснительной записки:

Не менее 26 страниц.

Дата выдачи задания: 23.11.2020

Дата сдачи реферата: 24.12.2020

Дата защиты реферата: 24.12.2020

Студент

Тиняков С.А.

Преподаватель

Филатов Ар.Ю.

АННОТАЦИЯ

Была разработана программа на языке программирования C++ для работы с красно-чёрным деревом. Был разработан TUI для взаимодействия с программой. Функционал программы следующий: вставка в дерево, поиск, вывод дерева. Для вывода дерева используется псевдографика. Для лучшего восприятия использовались различные цвета.

SUMMARY

A program was developed in the C ++ programming language for working with a red-black tree. TUI was developed to interact with the program. The functionality of the program is as follows: insert into a tree, search, output a tree. To display the tree, pseudo-graphics are used. Various colors were used for better use.

СОДЕРЖАНИЕ

Введение	5
1. Формальная постановка задачи	6
2. Структура данных красно-чёрное дерево	7
2.1. Определение красно-чёрного дерева	7
2.2. Алгоритм вставки	7
3. Описание структур данных и функций	10
3.1. Макросы и константы	10
3.2. Класс RedBlackTreeNode	10
3.3. Класс RedBlackTree	11
3.4. Функция main	13
4. Описание интерфейса пользователя	15
4.1. Описание	15
4.2. Текущее дерево	15
4.3. Вставка элемента	16
4.4. Поиск элемента	20
4.5. Ввод некорректной команды	22
4.6. Команда выхода	22
4.7. Обработка исключений	23
5. Тестирование	24
Заключение	25
Список использованных источников	26
Приложение А. Исходный код программы	27

ВВЕДЕНИЕ

Целью данной работы является изучение структуры данных красно-чёрное дерево и алгоритма вставки в него и реализация на языке программирования C++. Также необходимо сделать визуализацию с подробными пояснениями, чтобы программу можно было использовать в целях обучения.

1. ФОРМАЛЬНАЯ ПОСТАНОВКА ЗАДАЧИ

Реализовать визуализацию красно-чёрного дерева и алгоритма вставки элемента. Демонстрация должна быть подробной и понятной (в том числе сопровождаться пояснениями), чтобы программу можно было использовать в обучении для объяснения используемой структуры данных и выполняемых с ней действий

2. СТРУКТУРА ДАННЫХ КРАСНО-ЧЁРНОЕ ДЕРЕВО

2.1. Определение красно-чёрного дерева

Красно-чёрное дерево — это бинарное самобалансирующееся дерево поиска. Каждый узел имеет дополнительный параметр — цвет: красный или чёрный, который используется для балансировки дерева при вставке и удалении. Бинарное дерево поиска является красно-чёрным деревом если:

1. Каждый узел красный или чёрный
2. Корень дерева всегда чёрный
3. Все пустые узлы (NIL) чёрные
4. У красного узла оба сына чёрные
5. Любой путь от заданного узла до пустого (NIL) потомка проходит через одинаковое количество черных вершин — чёрная высота.

2.2. Алгоритм вставки

Введём следующие обозначения: текущий узел(N) — узел, для которого выполняется перебалансировка, отец(P) — узел, для которого данный является сыном, дед(G) — узел, для которого отец является сыном, дядя(U) — сын деда, который не является отцом. Алгоритм вставки работает следующим образом: сначала новый элемент вставляется как в обычное бинарное дерево поиска. Новый узел имеет красный цвет. Далее делается перебалансировка дерева. Возможны четыре случая:

1. Узел N является корнем. В этом случае узел N перекрашивается в чёрный цвет
2. Отец P — чёрный. В этом случае ни одно из требований красно-чёрного дерева не нарушается. Никаких дополнительных действий делать не надо.

3. Отец P и дядя U — красные. В этом случае узлы P и U перекрашиваются в чёрный цвет, а дед G — в красный. Пример приведён на рис. 1. Далее вызывается перебалансировка для узла G .

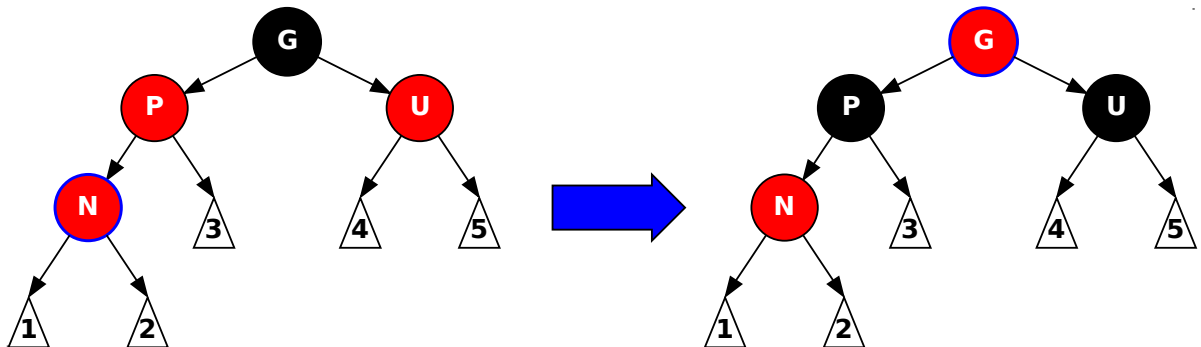


Рисунок 1 — Пример для случая 3

4. Отец P — красный, дядя U — чёрный. В этом случае необходимо сделать поворот дерева(возможно два). Сначала проверяется, находятся ли узел N и P в одной стороне. Т.е. если отец — левый сын, то и текущий узел тоже должен быть левым сыном. Аналогично для правой стороны. Далее будет считаться, что отец — левый сын. Для ситуации, когда отец — правый сын, все действия аналогичны, только симметричны. Если отец и текущий узел в разных сторонах, то сначала нужно сделать малый поворот: узлы P и N меняются местами, при этом отец становится левым сыном текущего узла, а левый сын узла N становится правым сыном узла P . Пример приведён на рис. 2.

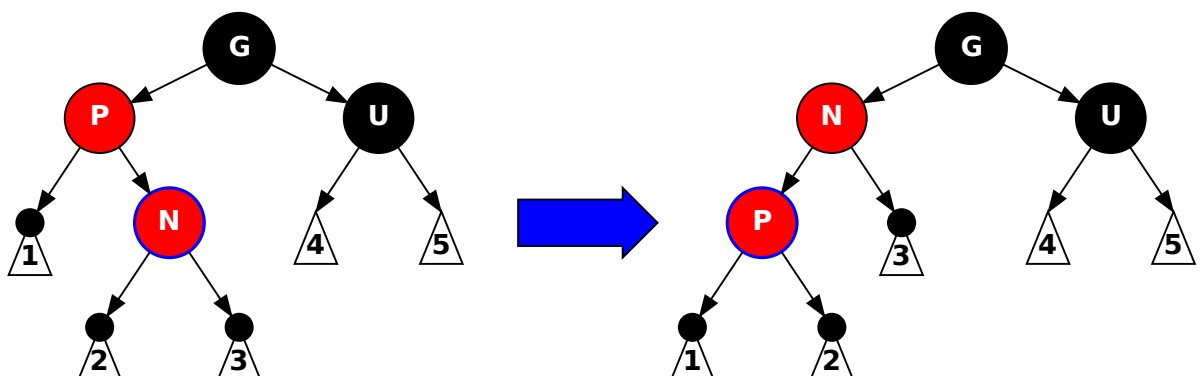


Рисунок 2 — Пример малого поворота

Если был выполнен малый поворот, то в большом повороте узел P — это узел N , а N — это P . Если отец и текущий узел находятся в одной стороне, то

выполняется большой поворот: отец помещается на место деда, перекрашивается в чёрный цвет, правым сыном становится дед. Дед перекрашивается в красный цвет, левым сыном становится бывший правый сын отца. Пример приведён на рис. 3.

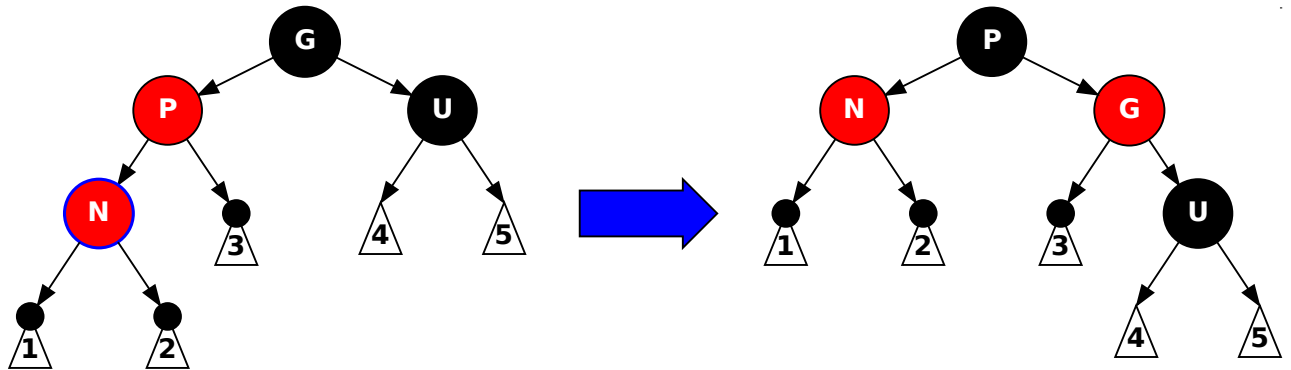


Рисунок 3 — Пример большого поворота

3. ОПИСАНИЕ СТРУКТУР ДАННЫХ И ФУНКЦИЙ

3.1. Макросы и константы

Если макрос *PRINT* определён, программа будет выводить визуализацию и дополнительные пояснения во время работы. Также будут определены следующие константы:

- *kRED* — красный цвет
- *kBLACK* — чёрный цвет
- *kBLUE* — синий цвет
- *kORANGE* — оранжевый цвет
- *kNORMAL* — сброс цвета
- *kNULL* — *nullptr* данные
- *SPACER* — макрос, определяющий разделитель
- *kVERTICAL* — вертикальная черта
- *kHORIZONTAL* — горизонтальная черта
- *kLEFT_UPPER_CORNER* — левый верхний угол
- *kLEFT_DOWN_CORNER* — левый нижний угол
- *kRIGHT_UPPER_CORNER* — правый верхний угол
- *kRIGHT_DOWN_CORNER* — правый нижний угол
- *kCONNECT_UP* — горизонтальная черта с разветвлением наверх
- *kCONNECT_LEFT* — вертикальная черта с разветвлением влево
- *kCONNECT_RIGHT* — вертикальная черта с разветвлением вправо

3.2. Класс *RedBlackTreeNode*

Шаблонный класс *RedBlackTreeNode* является реализацией узла дерева. Все поля находятся под модификатором *public*.

Поля:

- *is_red* — определяет, является ли узел красным.
- *is_left* — определяет, является ли узел левым потомком.
- *left_pos* — левая координата для визуализации

- *right_pos* — правая координата для визуализации
- *data* — данные
- *left* — левый сын
- *right* — правый сын
- *parent* — отец

Методы:

- *RedBlackTreeNode()* — конструктор по умолчанию
- *RedBlackTreeNode(T& data)* — конструктор с инициализацией данных через ссылку на них
- *RedBlackTreeNode(T&& data)* — конструктор с инициализацией данных через r-value ссылку на них
- *~RedBlackTreeNode()* — деструктор по умолчанию
- *RedBlackTreeNode(const RedBlackTreeNode<T>& node)* — конструктор копирования
- *RedBlackTreeNode& operator=(const RedBlackTreeNode<T>& node)* — оператор копирования
- *RedBlackTreeNode(RedBlackTreeNode<T>&& node)* — конструктор перемещения
- *RedBlackTreeNode& operator=(RedBlackTreeNode<T>&& node)* — оператор перемещения

3.3. Класс **RedBlackTree**

Шаблонный класс *RedBlackTree* является реализацией структуры данных красно-чёрное дерево. Все поля находятся под модификатором доступа *protected*.

Поля:

- *head* — указатель на корень дерева
- *out* — поток для вывода визуализации

Методы под модификатором доступа *protected*:

- *bool IsBlackNode(NodePtr node)* — возвращает *true*, если узел *node* является чёрным, иначе *false*
- *bool IsRedNode(NodePtr node)* — возвращает *true*, если узел *node* является красным, иначе *false*
- *bool Recolor(NodePtr node)* — выполняет перекраску дерева(случай №3) для узла *node*. Метод возвращает *true*, если была осуществлена перекраска, иначе *false*
- *void SmallRotate(NodePtr node)* — выполняет малый поворот(случай №4) для узла *node*
- *void BigRotate(NodePtr node)* — выполняет большой поворот(случай №5) для узла *node*
- *bool Rotate(NodePtr& node)* — метод вызывает *SmallRotate* и *BigRotate*, если необходимо сделать повороты для узла *node*. Метод возвращает *true*, если был совершён поворот, иначе *false*.
- *void Balance(NodePtr node, bool is_start = true)* — метод делает перебалансировку для узла *node*, т. е. Вызывает методы *Rotate* и *Recolor*. Параметр *is_start* определяет, был ли метод вызван после вставки в дерево элемента(*true*) или рекурсивно из метода *Recolor*(*false*).
- *void PrintTree(std::wostream& os = std::wcout, NodePtr new_node = nullptr, NodePtr start_node = nullptr)* — метод выводит дерево в поток вывода *os*. При выводе дерева *new_node* будет выводиться цветом *kORANGE*. Если *new_node* — *nullptr*, то вывод будет происходить начиная с узла *start_node*. Если *start_node* — *nullptr*, то вывод начнётся с корня дерева. Если *new_node* не является *nullptr*, а *start_node* — *nullptr*, то вывод начнётся с деда *new_node*(если его нет, то с отца, а если и его нет, то с *new_node*)

Методы под модификатором доступа *public*:

- *RedBlackTree()* — конструктор по умолчанию
- *~RedBlackTree()* — деструктор по умолчанию

- *RedBlackTree(std::wostream& os)* — конструктор с инициализацией вывода для визуализации
- *RedBlackTree(const RedBlackTree<T>& tree)* — конструктор копирования
- *RedBlackTree& operator=(const RedBlackTree<T>& tree)* — оператор копирования
- *RedBlackTree(RedBlackTree<T>&& tree)* — конструктор перемещения
- *RedBlackTree& operator=(RedBlackTree<T>&& tree)* — оператор перемещения
- *std::wostream& GetOutputStream()* — метод возвращает поток, в который выводится дерево
- *void SetOutputStream(std::wostream& os)* — метод меняет значение поля *out* на *os*
- *void Insert(T new_data)* — метод выполняет вставку элемента *new_data* в дерево. После для перебалансировки дерева вызывается метод *Balance*, в который аргументом передаётся указатель на новый узел с данными *new_data*
- *void PrintData(std::wostream& os = std::wcout)* — выводит данные в ЛКП порядке в поток вывода *os*
- *void Print(std::wostream& os = std::wcout)* — если определён макрос *PRINT*, то вызывается метод *PrintTree* с аргументом *os*. Если макрос *PRINT* не определён, то вызывается метод *PrintData* с аргументом *os*
- *bool Find(T find_data)* — метод выполняет поиск элемента в дереве. Возвращает *true*, если элемент присутствует в дереве, иначе *false*

3.4. Функция *main*

У функции *main* есть два режима работы: режим тестирования и режим визуализации. Если макрос *PRINT* не определён, то работа происходит всегда в режиме тестирования. Если макрос *PRINT* определён, то работа происходит в режиме визуализации, для запуска режиме тестирования необходимо, чтобы

первым аргументом было „*test*“. В режиме тестирования сначала считывается N — количество элементов, затем N элементов, а после элемент для поиска. В режиме визуализации используется *TUI*.

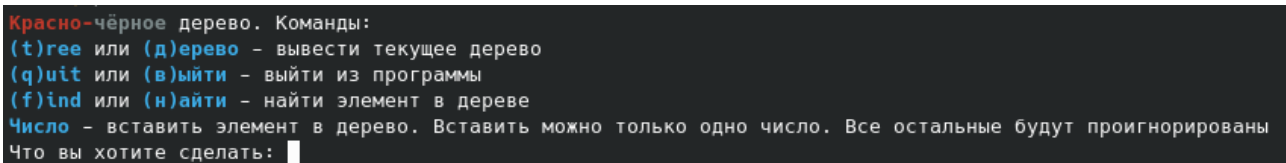
4. ОПИСАНИЕ ИНТЕРФЕСА ПОЛЬЗОВАТЕЛЯ

4.1. Описание

При запуске программы в визуальном режиме появляется подсказка и ожидается ввод команды(рис. 4). Следующие команды возможны:


- (q)uit — выйти из программы
- (t)ree — вывести текущее дерево
- (f)ind — найти элемент в дереве. При вводе данной команды откроется подменю(рис. 5). Возможны следующие команды:
 - (q)uit — выйти из поиска
 - Число — будет выполнен поиск введённого числа. Считано будет только одно число, всё остальное будет проигнорировано
- Число — будет выполнена вставка введённого числа. Считано будет только одно число, всё остальное будет проигнорировано

Все команды можно вводить с большой буквы, также присутствуют команды на русском.



```
Красно-чёрное дерево. Команды:  
(t)ree или (d)ерево - вывести текущее дерево  
(q)uit или (v)ыйти - выйти из программы  
(f)ind или (n)айти - найти элемент в дереве  
Число - вставить элемент в дерево. Вставить можно только одно число. Все остальные будут проигнорированы  
Что вы хотите сделать: █
```

Рисунок 4 — Главное меню



```
Поиск элемента.  
Введите число для поиска, (q)uit или (v)ыйти для выхода из поиска  
Что вы хотите сделать: █
```

Рисунок 5 — Меню поиска

4.2. Текущее дерево

Пример отображения текущего дерева представлен на рис. 6.

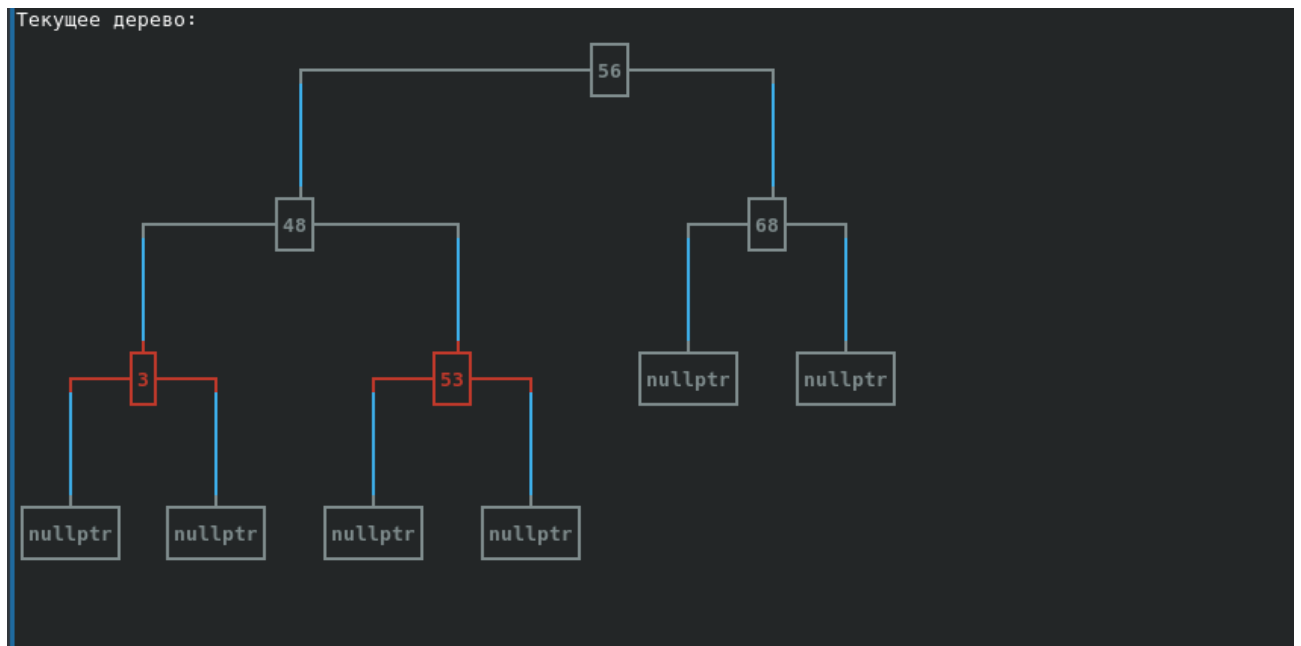


Рисунок 6 — Пример выполнения команды *tree*

4.3. Вставка элемента

При вставке элемента отображаются все выполняемые действия со структурой данных, сопровождающиеся пояснениями. Пример вставки элемента в дерево изображён на рис. 7 — 12.

Вставка элемента **51**

Производится вставка элемента **51**, как в обычное бинарное дерево поиска.

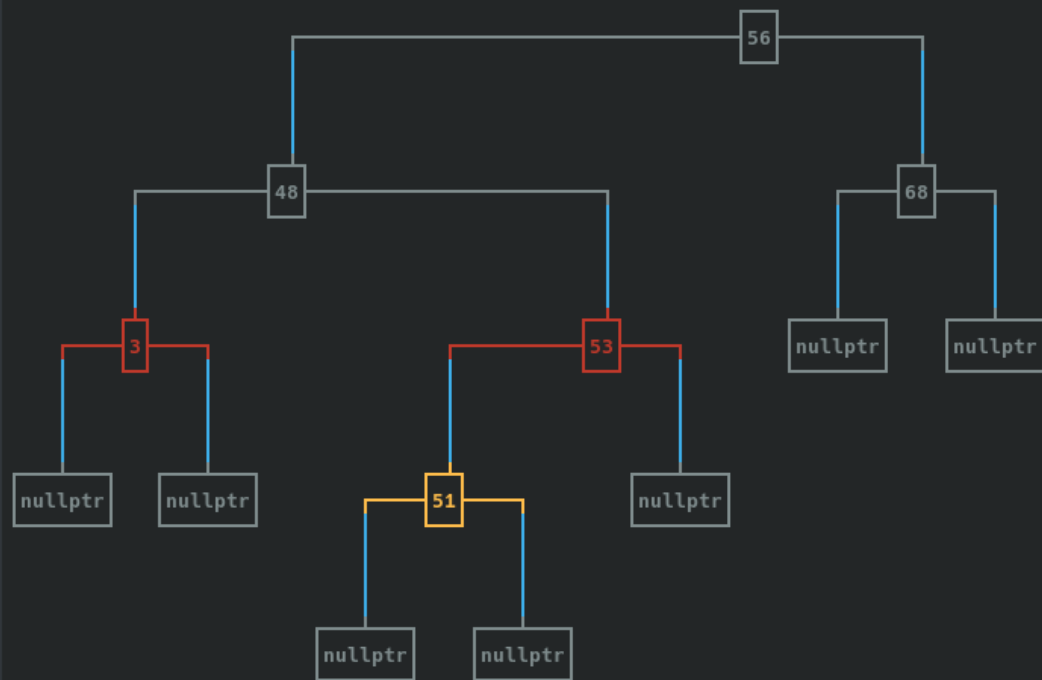


Рисунок 7 — Вставка элемента в дерево

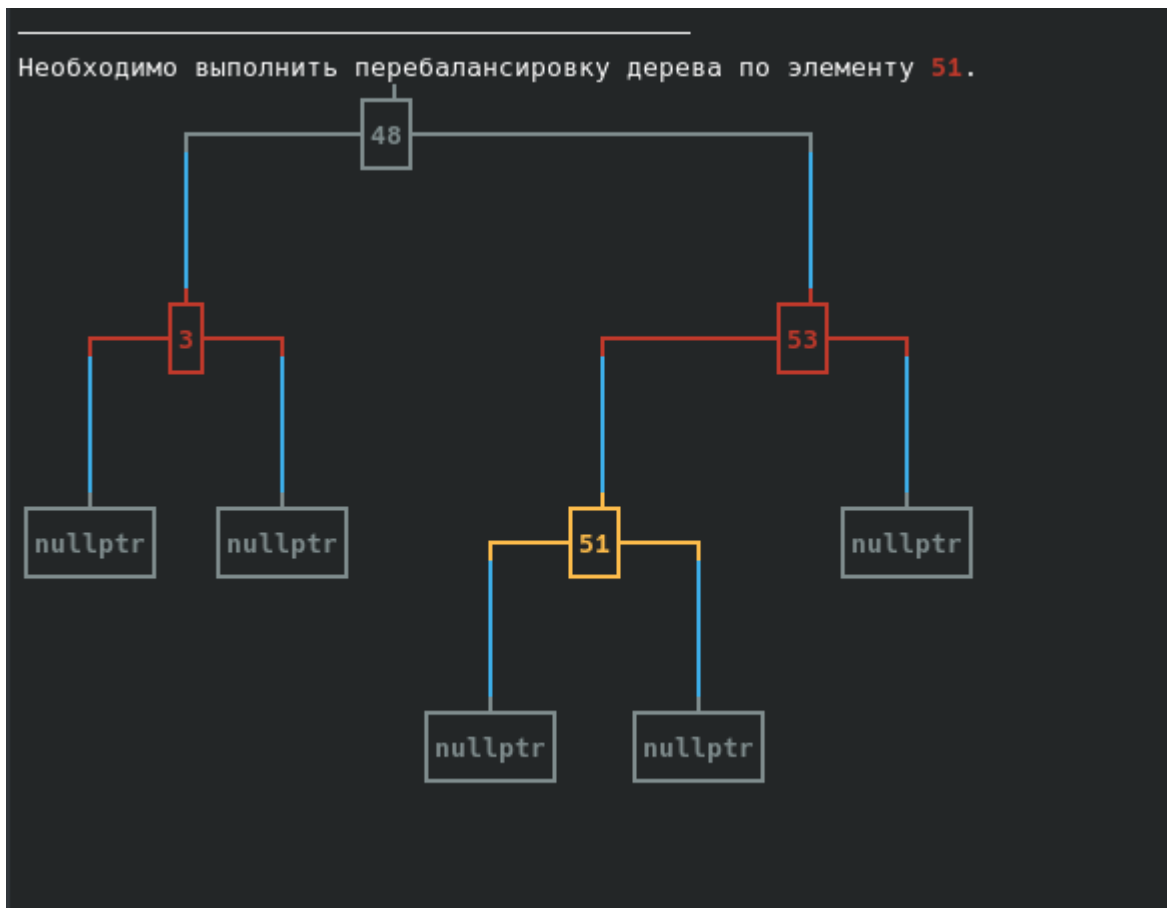


Рисунок 8 — Начало перебалансировки дерева по новому элементу

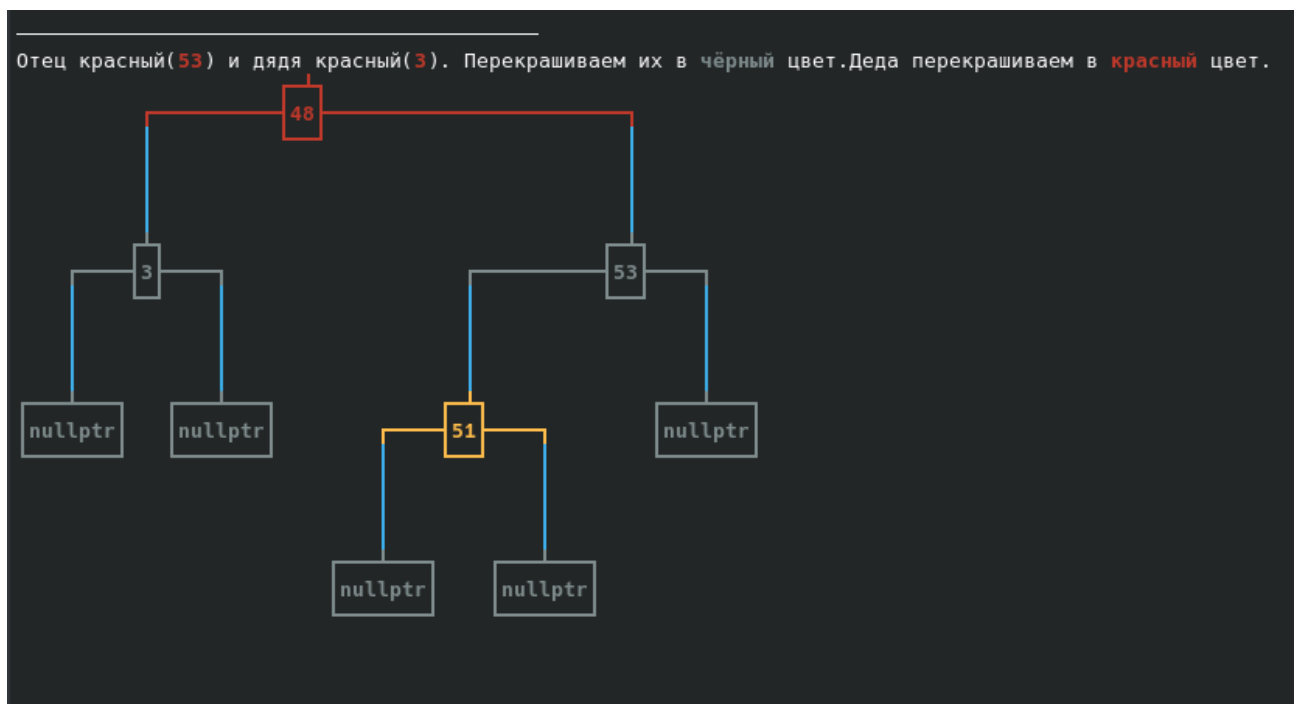


Рисунок 9 — Случай №3, выполняется перекраска элементов

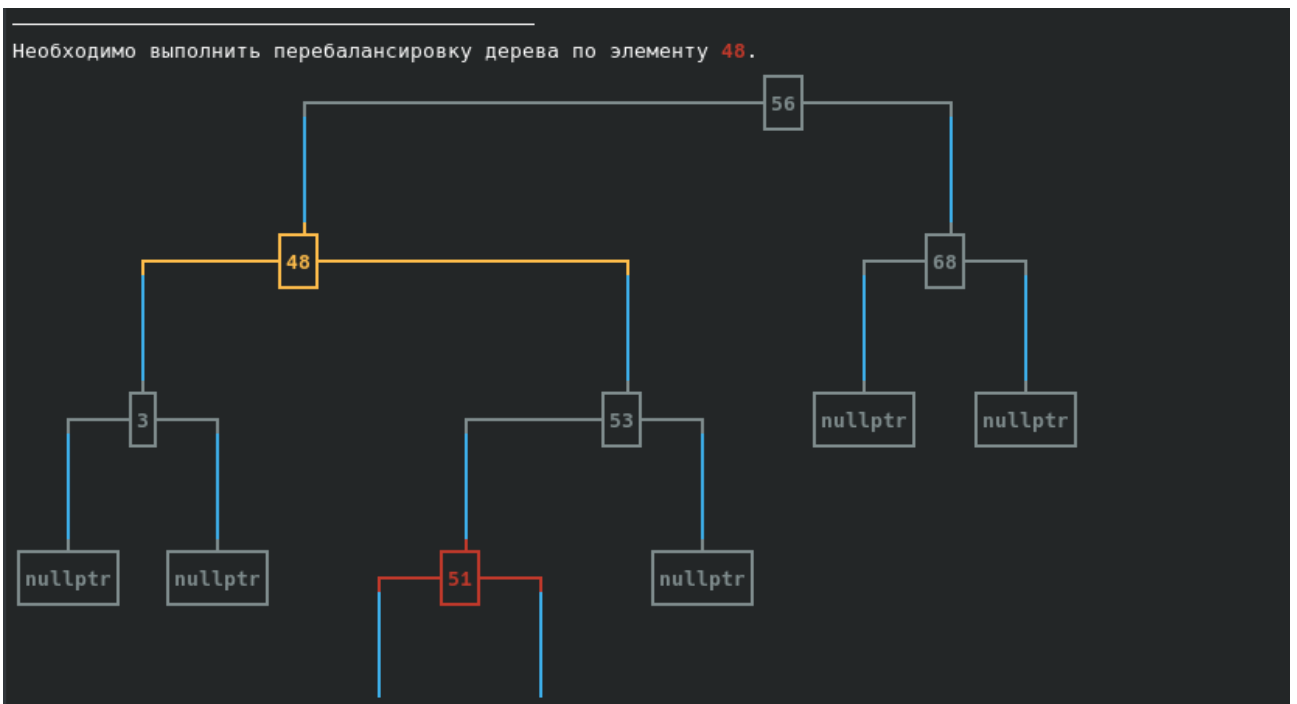


Рисунок 10 — Начало перебалансировки по другому элементу

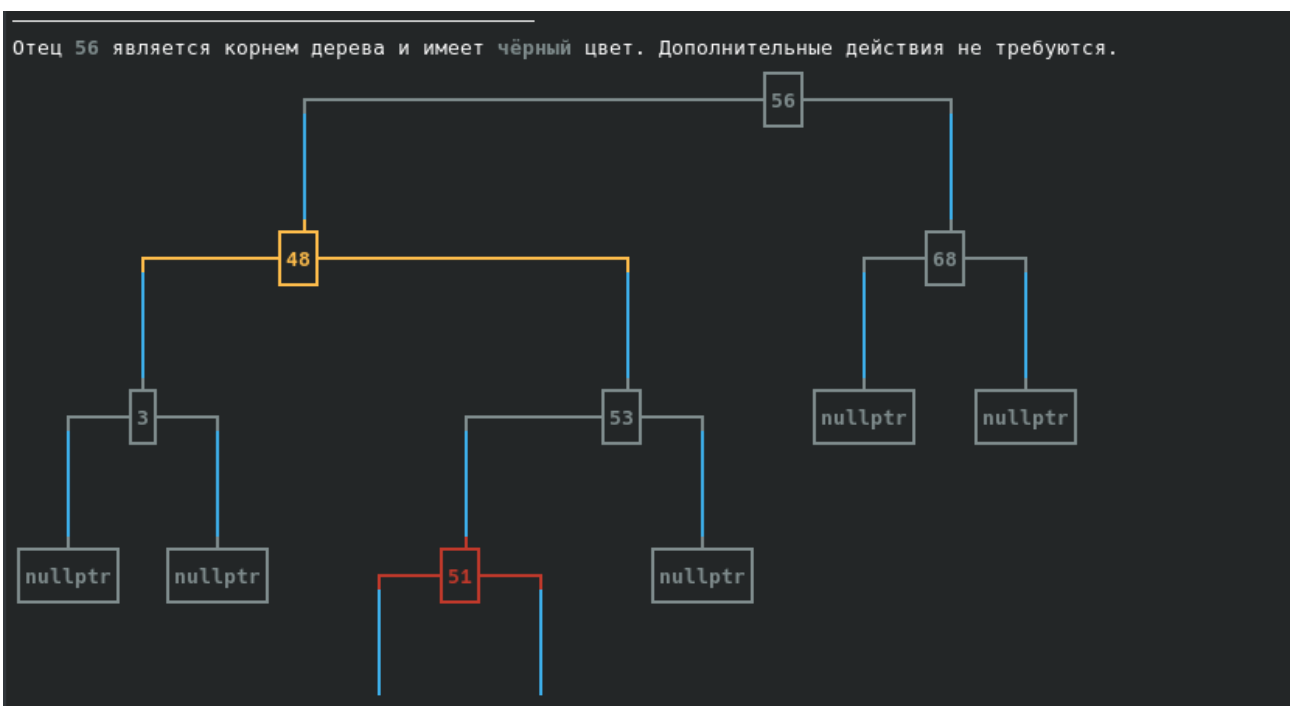


Рисунок 11 — Случай №2, дополнительные действия не требуются

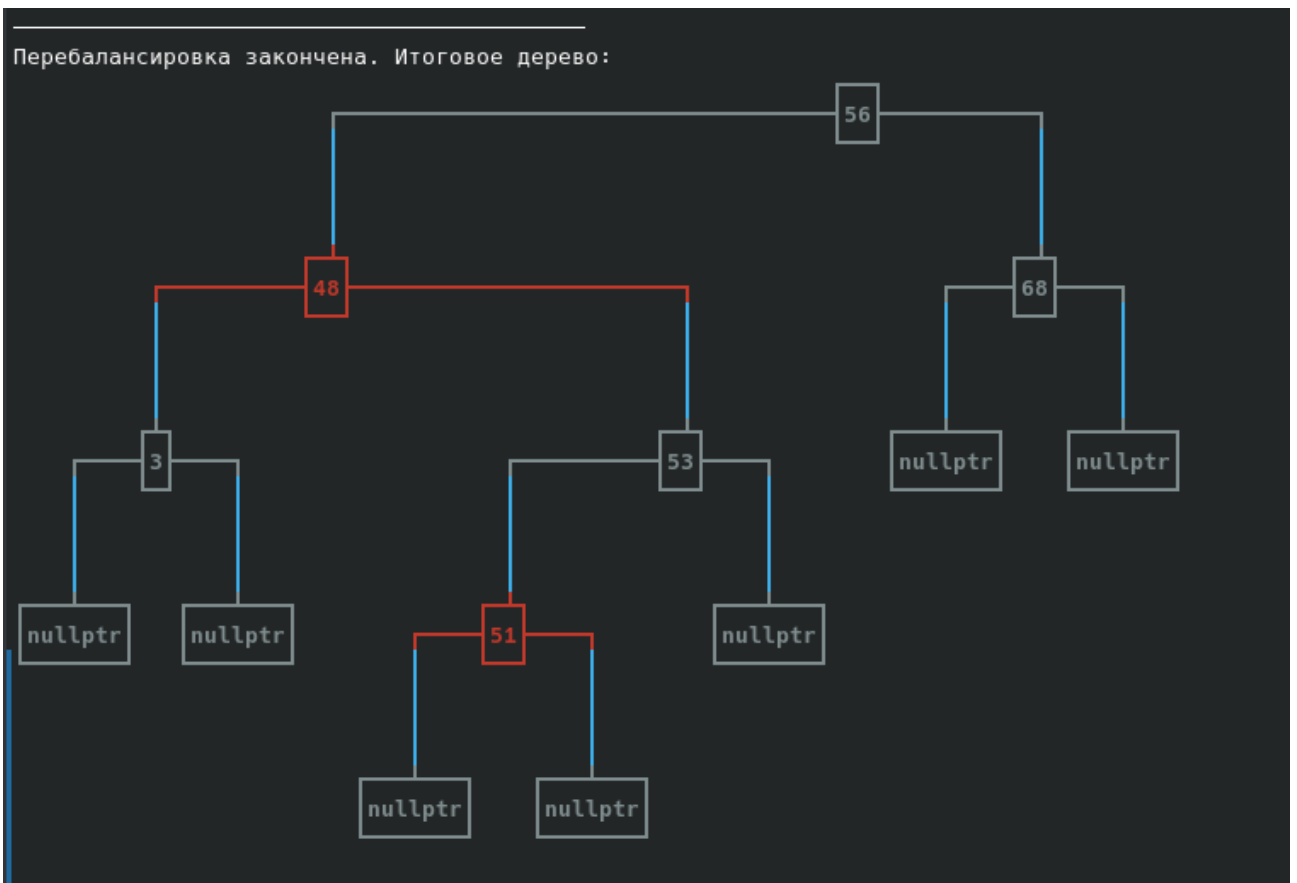


Рисунок 12 — Окончательное дерево после перебалансировки

4.4. Поиск элемента

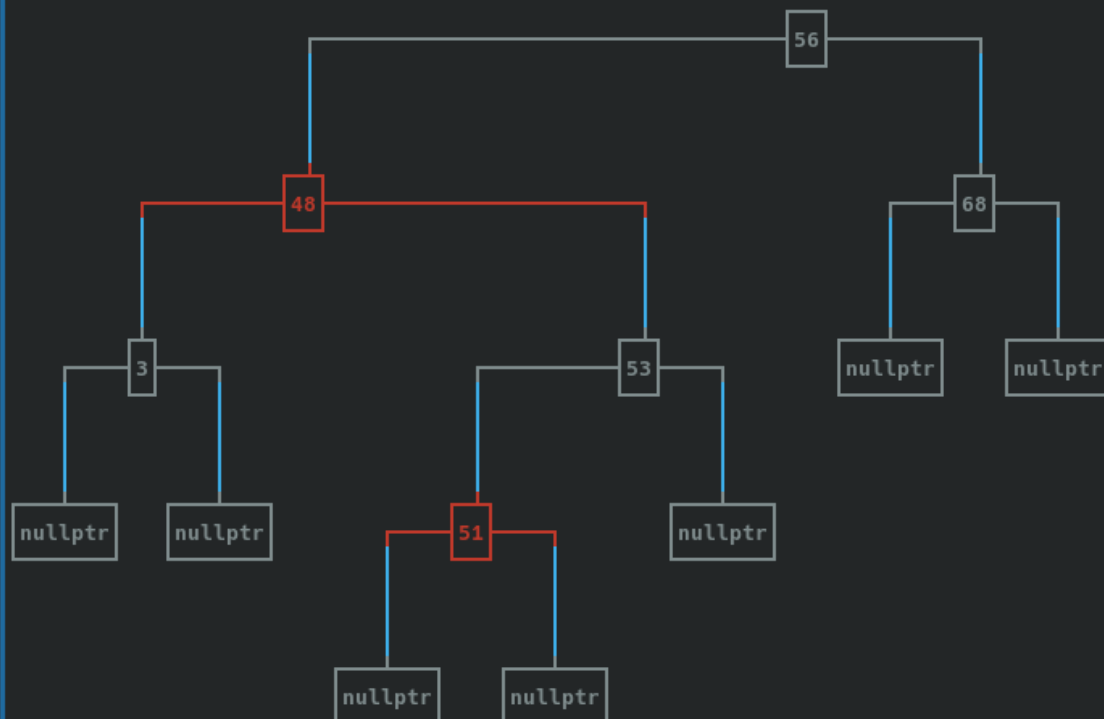
Пример поиска элемента представлен на рис. 13 — 14.

Поиск элемента.

Введите **число** для поиска, **(q)uit** или **(в)ыйти** для выхода из поиска

Что вы хотите сделать: -684

Поиск элемента **-684...**



Элемент **-684** **отсутствует** в дереве

Рисунок 13 — Пример поиска элемента отсутствующего в дереве

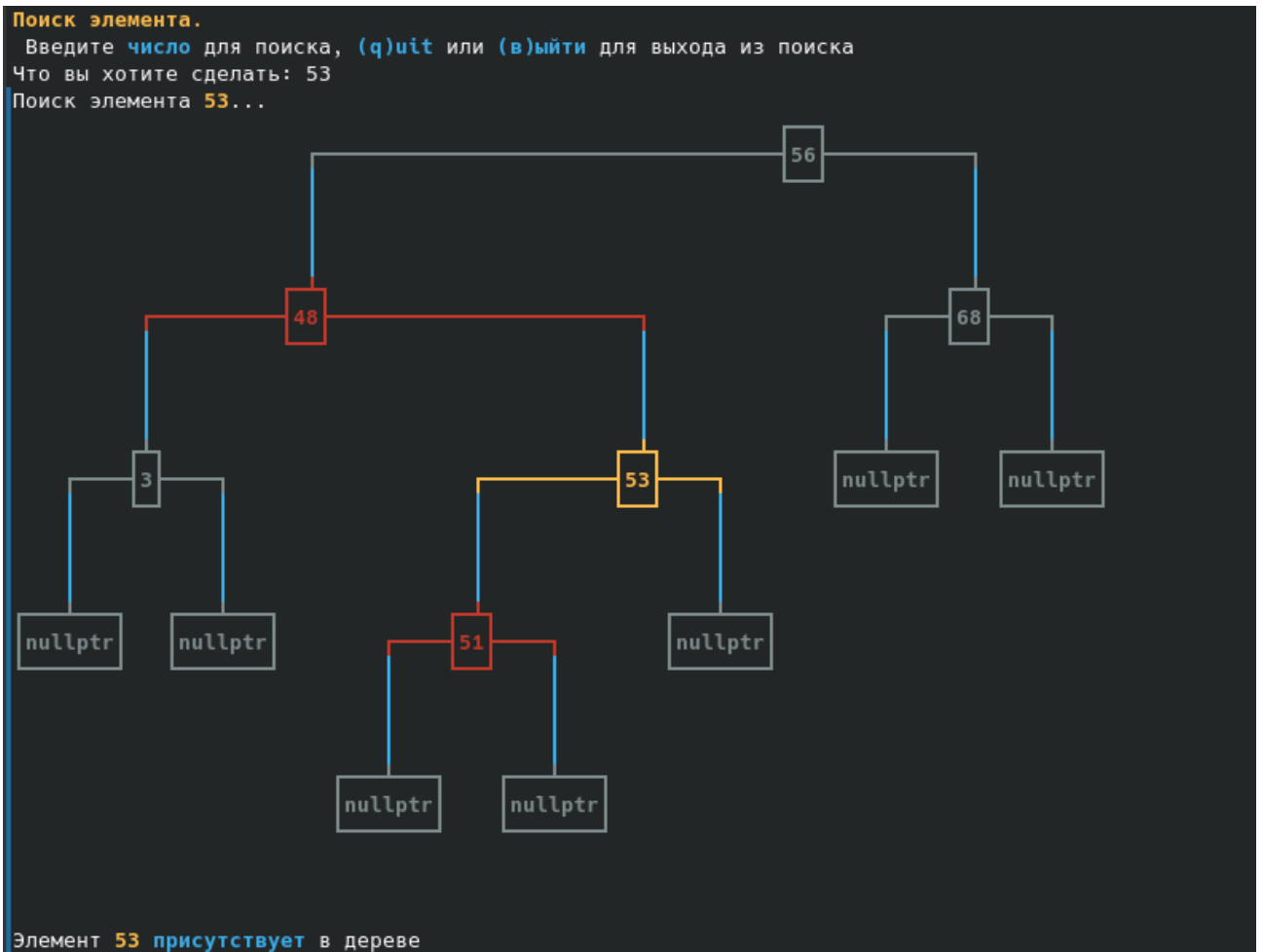


Рисунок 14 — Пример поиска элемента присутствующего в дереве

4.5. Ввод некорректной команды

Пример ввода некорректной команды представлен на рис. 15.

```

Красно-чёрное дерево. Команды:
(t)ree или (д)ерево - вывести текущее дерево
(q)uit или (в)ыйти - выйти из программы
(f)ind или (н)айти - найти элемент в дереве
Число - вставить элемент в дерево. Вставить можно только одно число. Все остальные будут проигнорированы
Что вы хотите сделать: test
Некорректная команда!

Красно-чёрное дерево. Команды:
(t)ree или (д)ерево - вывести текущее дерево
(q)uit или (в)ыйти - выйти из программы
(f)ind или (н)айти - найти элемент в дереве
Число - вставить элемент в дерево. Вставить можно только одно число. Все остальные будут проигнорированы
Что вы хотите сделать: █

```

Рисунок 15 — Пример ввода некорректной команды

4.6. Команда выхода

Пример работы команды выхода представлен на рис. 16.

```

Красно-чёрное дерево. Команды:
(t)ree или (д)ерево - вывести текущее дерево
(q)uit или (в)ыйти - выйти из программы
(f)ind или (н)айти - найти элемент в дереве
Число - вставить элемент в дерево. Вставить можно только одно число. Все остальные будут проигнорированы
Что вы хотите сделать: в
Программа завершает свою работу...

```

Рисунок 16 — Пример работы команды выхода

4.7. Обработка исключений

Пример обработки исключений представлен на рис. 17. Для этого в методе *Find* класса *RedBlackTree* в самом начале было прописано, чтобы выбрасывалось исключение.

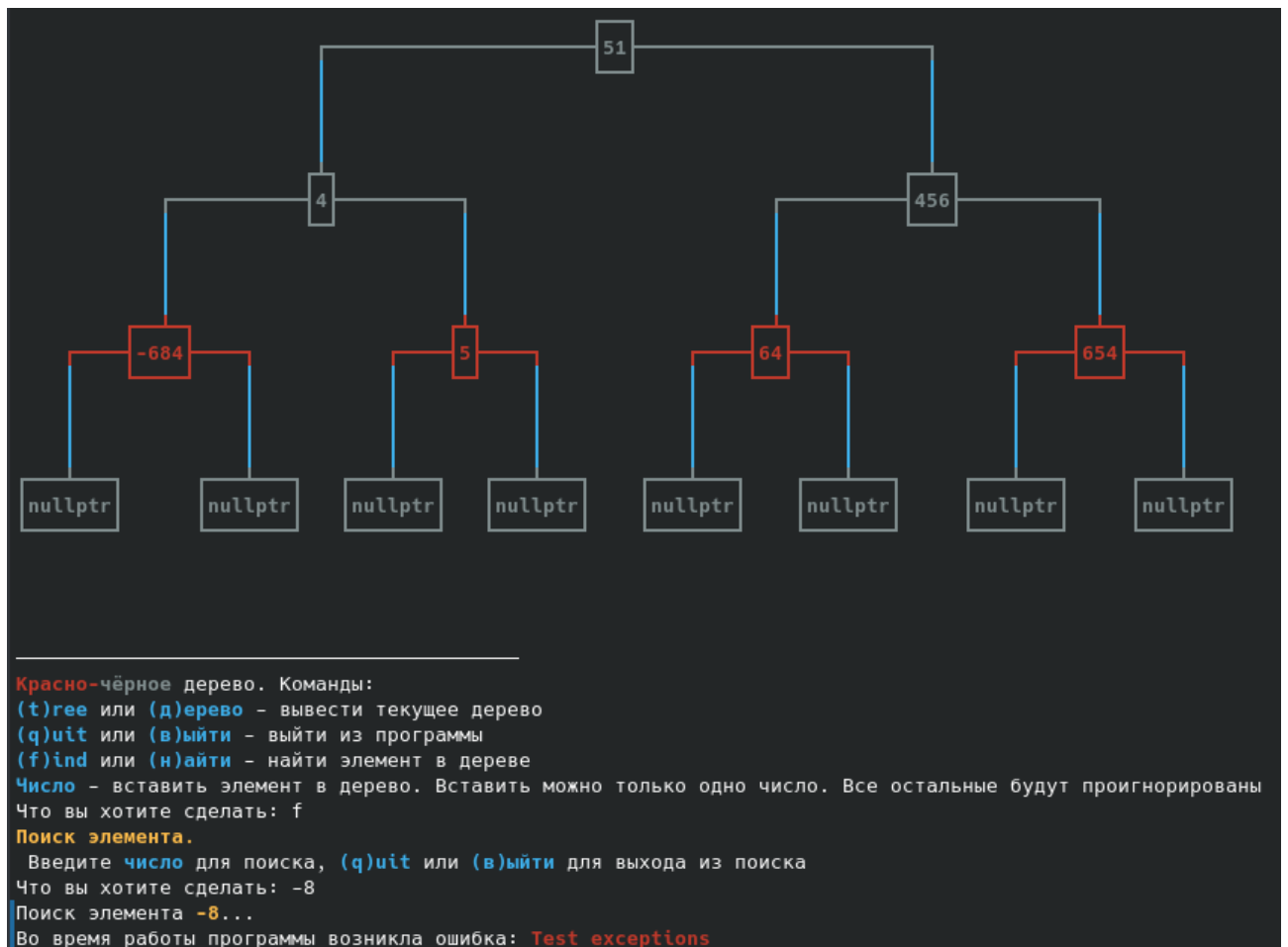


Рисунок 17 — Пример обработки исключений

5. ТЕСТИРОВАНИЕ

Тестирование проводилось при помощи *python*-скрипта. Для заданного размера генерировался набор случайны чисел, которые подавались на вход программе. Также случайно генерировалось число для поиска. Проверка происходила через удаление дубликатов и сортировки средствами языке программирования *Python*. Набор размеров для тестирования: 25, 73, 549, 1091.

ЗАКЛЮЧЕНИЕ

Была изучена структура данных красно-чёрное дерево и алгоритм вставки для неё. Была разработана программа на языке программирования C++, которая визуализирует работу с красно-чёрным деревом. Также визуализация сопровождается пояснениями. Для взаимодействия с программой был реализован *TUI*. Для лучшего восприятия использовались различные цвета. При реализации структуры данных и алгоритма вставки использовались такие возможности 17-ого стандарта C++, как лямбда-функции и умные указатели.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Wikipedia. URL: https://en.wikipedia.org/wiki/Red%E2%80%93black_tree
(дата обращения: 20.12.2020).
2. The C++ Resources Network. URL: <http://www.cplusplus.com/> (дата обращения: 20.12.2020).

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: Source/cw.cpp

```
#include <iostream>
#include <memory>
#include <cwchar>
#include <cwctype>
#include <cctype>
#include <sstream>
#include <vector>
#include <deque>
#include <utility>
#include <cstring>
#include <locale>

// Макрос PRINT нужен для визуализации
#define PRINT

#ifndef PRINT

// Определение цветов для визуализации
constexpr const wchar_t* kRED = L"\033[1;31m";
constexpr const wchar_t* kBLACK = L"\033[1;30m";
constexpr const wchar_t* kBLUE = L"\033[1;34m";
constexpr const wchar_t* kORANGE = L"\033[1;33m";
constexpr const wchar_t* kNORMAL = L"\033[0m";
constexpr const wchar_t* kNULL = L"nullptr";

// Разделитель
#define SPACER (L'\n' + std::wstring(42, 0x2500) + L'\n')

// Определение символов из Unicode
constexpr wchar_t kVERTICAL = 0x2502;
constexpr wchar_t kHORIZONTAL = 0x2500;
constexpr wchar_t kLEFT_UPPER_CORNER = 0x250c;
constexpr wchar_t kLEFT_DOWN_CORNER = 0x2514;
constexpr wchar_t kRIGHT_UPPER_CORNER = 0x2510;
constexpr wchar_t kRIGHT_DOWN_CORNER = 0x2518;
constexpr wchar_t kCONNECT_UP = 0x2534;
constexpr wchar_t kCONNECT_LEFT = 0x2524;
constexpr wchar_t kCONNECT_RIGHT = 0x251c;

#endif

// Шаблонный класс узла дерева
template<typename T>
class RedBlackTreeNode{
public:

    bool is_red = true; // Является ли узел красным
    bool is_left = false; // Является ли узел левым потомком. Если true,
    то левым, иначе правым

    // Поля left_pos и right_pos хранят позиции узла в дереве для печати
```

```

int left_pos = 0, right_pos = 0;

T data; // Данные, которые хранит узел
std::shared_ptr<RedBlackTreeNode<T>> left{nullptr},
right{nullptr}; // Левый и правый сыновья
std::weak_ptr<RedBlackTreeNode<T>> parent; // Отец

RedBlackTreeNode() = default; // Конструктор по умолчанию

// Конструктор по данным через ссылку
RedBlackTreeNode(T& data){
    this->data = data;
}

// Конструктор по данным через r-value ссылку
RedBlackTreeNode(T&& data){
    this->data = data;
}

~RedBlackTreeNode() = default; // Деструктор по умолчанию

//Конструктор копирования
RedBlackTreeNode(const RedBlackTreeNode<T>& node){
    data = node.data;
    left = node.left;
    right = node.right;
    parent = node.parent;
    is_red = node.is_red;
    is_left = node.is_left;
    left_pos = left_pos;
    right_pos = right_pos;
}

// Оператор копирования
RedBlackTreeNode& operator=(const RedBlackTreeNode<T>& node){
    if(&node == this) return *this;
    data = node.data;
    left = node.left;
    right = node.right;
    parent = node.parent;
    is_red = node.is_red;
    is_left = node.is_left;
    left_pos = left_pos;
    right_pos = right_pos;
    return *this;
}

// Конструктор перемещения
RedBlackTreeNode(RedBlackTreeNode<T>&& node){
    data = std::move(node.data);
    left = std::move(node.left);
    right = std::move(node.right);
    parent = std::move(node.parent);
    is_red = node.is_red;
    is_left = node.is_left;
    left_pos = left_pos;
    right_pos = right_pos;
}

```

```

// Оператор перемещения
RedBlackTreeNode& operator=(RedBlackTreeNode<T>&& node){
    if(&node == this) return *this;
    data = std::move(node.data);
    left = std::move(node.left);
    right = std::move(node.right);
    parent = std::move(node.parent);
    is_red = node.is_red;
    is_left = node.is_left;
    left_pos = left_pos;
    right_pos = right_pos;
    return *this;
}

};

// Оператор вывода для узла
template<typename T>
std::wostream& operator<<(std::wostream& os, const RedBlackTreeNode<T>&
node){
    os << "{data: " << node.data << "; is_red: " << node.is_red
        << "; is_left: " << node.is_left << "; left: ";
    if(node.left) os << node.left->data;
    else os << "nullptr";
    os << "; right: ";
    if(node.right) os << node.right->data;
    else os << "nullptr";
    os << "; parent: ";
    if(node.parent.lock()) os << node.parent.lock()->data;
    else os << "nullptr";
    os << "};";
    return os;
}

// Шаблонный класс красно-чёрного дерева
template<typename T>
class RedBlackTree{

using NodePtr = std::shared_ptr<RedBlackTreeNode<T>>;

protected:
    NodePtr head{nullptr}; // Корень дерева
    std::wostream& out = std::wcout; // Потока для вывода визуализации на
    промежуточных этапах

    // Метод IsBlackNode определяет, является ли узел чёрным
    bool IsBlackNode(NodePtr node){
        return (node == nullptr || !node->is_red);
    }

    // Метод IsRedNode определяет, является ли узел красным
    bool IsRedNode(NodePtr node){
        return !IsBlackNode(node);
    }

    /* Метод Recolor проверяет узел node на необходимость перекраски,
    * и перекрашивает узлы, если необходимо. Возвращается true
    * если была выполнена перекраска, иначе false

```

```

*/
bool Recolor(NodePtr node){
    // Если узел -- корень дерева
    if(!node->parent.lock()){
        #ifdef PRINT
            out << L"Элемент " << (node->is_red ? kRED : kBLACK)
            << node->data << kNORMAL << L" является корнем дерева.
Перекрашиваем его в "
            << kBLACK << L"чёрный" << kNORMAL << L" цвет.\n";
            node->is_red = false;
            PrintTree(out, node);
            out << SPACER;
        #else
            // Изменить цвет корня на чёрный
            node->is_red = false;
        #endif
        return true;
    }
    // Если нет деда
    if(!node->parent.lock()->parent.lock()){
        auto parent = node->parent.lock();
        #ifdef PRINT
            out << L"Отец " << kBLACK << parent->data << kNORMAL << L"
является корнем дерева и имеет "
            << kBLACK << L"чёрный" << kNORMAL << L" цвет. Дополнительные
действия не требуются.\n";
            parent->is_red = false;
            PrintTree(out, node);
            out << SPACER;
        #else
            // Изменить цвет отца на чёрный
            parent->is_red = false;
        #endif
        return true;
    }
    // Получение узлов отца, деда и дяди
    auto parent = node->parent.lock();
    auto grandparent = parent->parent.lock();
    auto uncle = (parent->is_left ? grandparent->right : grandparent-
>left);
    // Если отец и дядя красный
    if(IsRedNode(parent) && IsRedNode(uncle)){
        #ifdef PRINT
            out << L"Отец красный(" << kRED << parent->data << kNORMAL <<
L") и дядя красный(" << kRED
            << uncle->data << kNORMAL << L"). Перекрашиваем их в " <<
kBLACK << L"чёрный" << kNORMAL << L" цвет."
            << L"Деда перекрашиваем в " << kRED << L"красный" << kNORMAL
<< L" цвет.\n";
            parent->is_red = uncle->is_red = false;
            grandparent->is_red = true;
            PrintTree(out, node);
            out << SPACER;
            Balance(grandparent, false);
        #else
            // Изменение цвета отца и дяди на чёрный
            parent->is_red = uncle->is_red = false;
            // Изменение цвета деда на красный

```

```

        grandparent->is_red = true;
        // Перебалансировка дерева по деду
        Balance(grandparent);
    #endif
} else if (IsRedNode(parent)) { // Если отец красный
    #ifdef PRINT
        out << L"Отец красный(" << kRED << node->data << kNORMAL <<
L").Перекрашиваем его в "
        << kBLACK << L"чёрный" << kNORMAL << L" цвет.\n";
        parent->is_red = false;
        PrintTree(out, node);
        out << SPACER;
    #else
        // Изменить цвет отца на чёрный
        parent->is_red = false;
    #endif
} else {
    // Отец чёрный, ничего делать не нужно
    #ifdef PRINT
        out << L"Отец чёрный(" << kBLACK << parent->data << kNORMAL
<< L").Дополнительные действия не требуются.\n";
        PrintTree(out, node);
        out << SPACER;
    #endif
    return false;
}
return true;
}

// Метод SmallRotate выполняет малый поворот по узлу node
void SmallRotate(NodePtr node) {
    // Если node == nullptr или нет отца и деда, то завершить метод
    if (!node) return;
    if (!node->parent.lock()) return;
    if (!node->parent.lock()->parent.lock()) return;
    // Получение отца и деда
    auto parent = node->parent.lock();
    auto grandparent = parent->parent.lock();
    #ifdef PRINT
        auto uncle = (parent->is_left ? grandparent->right : grandparent-
>left);
        out << L"Дядя чёрный(" << kBLACK;
        if (uncle) out << uncle->data;
        else out << kNULL;
        out << kNORMAL << L"), отец красный("
        << kRED << parent->data << kNORMAL << L"). Отец и новый элемент
находятся в разных сторонах. Необходимо выполнить малый поворот.\n";
        PrintTree(out, node);
        out << SPACER;
    #endif
    // Изменение отца узла на деда
    node->parent = grandparent;
    // Если отец левый потомок
    if (parent->is_left) {
        grandparent->left = node;
        node->is_left = true;
        parent->right = node->left;
        if (node->left) {

```

```

        node->left->is_left = false;
        node->left->parent = parent;
    }
    parent->parent = node;
    node->left = parent;
} else { // Если отец правый потомок
    grandparent->right = node;
    node->is_left = false;
    parent->left = node->right;
    if (node->right) {
        node->right->is_left = true;
        node->right->parent = parent;
    }
    parent->parent = node;
    node->right = parent;
}
#ifdef PRINT
out << L"Малый поворот для элемента " << kORANGE
<< node->data << kNORMAL << L" выполнен.\n";
PrintTree(out, node, grandparent);
out << SPACER;
#endif
}

// Метод BigRotate выполняет большой поворот по узлу node
void BigRotate(NodePtr node) {
    // Если node == nullptr или нет отца и деда, то завершить метод
    if (!node) return;
    if (!node->parent.lock()) return;
    if (!node->parent.lock()->parent.lock()) return;
    // Получение отца, деда и дяди
    auto parent = node->parent.lock();
    auto grandparent = parent->parent.lock();
    auto uncle = (parent->is_left ? grandparent->right : grandparent-
>left);
    #ifdef PRINT
    out << L"Дядя чёрный(" << kBLACK;
    if (uncle) out << uncle->data;
    else out << kNULL;
    out << kNORMAL << L"), отец красный("
    << kRED << parent->data << kNORMAL << L"). Отец и новый элемент
находятся в одной стороне. Необходимо выполнить большой поворот.\n";
    PrintTree(out, node);
    out << SPACER;
    #endif
    // Обмен данными между отцом и дедом
    std::swap(parent->data, grandparent->data);
    // Если отец левый потомок
    if (parent->is_left) {
        // Перемещение отца, у которого данные деда, на другую
сторону
        parent->left = parent->right;
        if (parent->left) parent->left->is_left = true;
        parent->right = uncle;
        // Если дядя не nullptr, то отцом дяди назначить отца
        if (uncle) uncle->parent = parent;
        // Дополнительные переназначения для отца, деда и узла
        grandparent->right = parent;

```



```

        parent->is_left = false;
        grandparent->left = node;
        node->parent = grandparent;
    }else{
        // Перемещение отца, у которого данные деда, на другую
сторону
        parent->right = parent->left;
        if(parent->right) parent->right->is_left = false;
        parent->left = uncle;
        // Если дядя не nullptr, то отцом дяди назначить отца
        if(uncle) uncle->parent = parent;
        // Дополнительные переназначения для отца, деда и узла
        grandparent->left = parent;
        parent->is_left = true;
        grandparent->right = node;
        node->parent = grandparent;
    }
    #ifdef PRINT
    out << L"Большой поворот для элемента " << kORANGE
    << node->data << kNORMAL << L" выполнен.\n";
    PrintTree(out, node, grandparent);
    out << SPACER;
    #endif
}

/* Метод Rotate проверяет на необходимость поворота узел node.
 * Если был произведён поворот(ы), то возвращается true, иначе
 * false. Если производится малый поворот, то в node будет записан
отец
 */
bool Rotate(NodePtr& node){
    // Если нет отца или деда, то поворот выполнять не нужно
    if(!node->parent.lock()) return false;
    if(!node->parent.lock()->parent.lock()) return false;
    // Получение отца и дяди
    auto parent = node->parent.lock();
    auto uncle = (parent->is_left ? parent->parent.lock()->right :
        parent->parent.lock()->left);
    // Если отец красный, а дядя чёрный, то необходимо выполнить
поворот
    if(IsRedNode(parent) && IsBlackNode(uncle)){
        // Если отец и узел node находятся в разных сторонах, то
сначала необходимо сделать малый поворот
        if(node->is_left != parent->is_left){
            SmallRotate(node);
            #ifdef PRINT
            out << L"Теперь необходимо выполнить большой поворот по
элементу "
            << kORANGE << parent->data << kNORMAL << L".\n";
            PrintTree(out, parent);
            out << SPACER;
            #endif
            // Большой поворот необходимо сделать по отцу
            BigRotate(parent);
            node = parent;
        }else{ // Если находятся в одной стороне, то сразу большой
поворот
            BigRotate(node);

```

```

    }
    }else return false;
    return true;
}

// Метод Balance вызывает два метода: Recolor и Rotate для узла node
#ifdef PRINT
/* При визуализации параметр is_start сообщает, является вызов
данного метода
* первым(true), или метод был вызван рекурсивно из метода
Recolor(false)
*/
void Balance(NodePtr node, bool is_start = true){
    out << L"Необходимо выполнить перебалансировку дерева по элементу
"
    << kRED << node->data << kNORMAL << L".\n";
    PrintTree(out, node);
    out << SPACER;
    bool was_rotate = Rotate(node);
    bool was_recolor = (was_rotate ? false : Recolor(node));
    if(is_start){
        if(!(was_rotate || was_recolor)) out << L"Дерево уже
сбалансированно, никаких дополнительных действий выполнять не нужно.\n";
        else out << L"Перебалансировка закончена. Итоговое дерево: \
n";

        PrintTree(out, nullptr, head);
        out << SPACER;
    }
}
#else
void Balance(NodePtr node){
    Rotate(node);
    Recolor(node);
}
#endif

#ifdef PRINT
/* Метод PrintTree выводит дерево в поток вывода os. При выводе
дерева new_node будет выводиться жёлтым цветом.
* Если new_node -- nullptr, то вывод будет происходить начиная с
узла start_node. Если start_node -- nullptr,
* то вывод начнётся с корня дерева. Если new_node не является
nullptr, а start_node -- nullptr, то вывод
* начнётся с деда new_node(если его нет, то с отца, а если и его
нет, то с new_node).
*/
void PrintTree(std::wostream& os = std::wcout, NodePtr new_node =
nullptr, NodePtr start_node = nullptr){
    bool is_start_head = (start_node == head); // Переменная
необходимая для определения глубины дерева
    if(!start_node && new_node){
        if(new_node->parent.lock()){
            if(new_node->parent.lock()->parent.lock()) start_node =
new_node->parent.lock()->parent.lock();
            else start_node = new_node->parent.lock();
        }else start_node = new_node;
    }
    if(!start_node){

```

```

        // Если корня нет, то вывести nullptr
        if(!head){
            const int len = wcslen(kNULL);
            os << kBLACK << kLEFT_UPPER_CORNER
            << std::wstring(len, kHORIZONTAL) << kRIGHT_UPPER_CORNER
<< L"\n"
            << kVERTICAL << kNULL << kVERTICAL << L"\n"
            << kLEFT_DOWN_CORNER << std::wstring(len, kHORIZONTAL)
            << kRIGHT_DOWN_CORNER << kNORMAL << SPACER;
            return;
        }
        start_node = head;
    }
    // Лямбда функция для определения длины вывода данных
    auto get_len = [](T& data)->int{
        std::wstringstream sstream;
        sstream << data;
        std::wstring str = sstream.str();
        for(int i = 0; i < str.size(); i++){
            // Вывод данных не должен содержать управляющих символов
            if(iswcntrl(str[i])) throw std::logic_error("Data print
has control characters!");
        }
        return str.size();
    };
    int pos = 0;
    const int len_null = wcslen(kNULL); // Длина nullptr узла
    // Лямбда функция для определения позиции узлов на экране
    auto set_positions = [&pos, &len_null](NodePtr node, auto&&
get_len, auto&& set_positions)->void{
        if(!node){
            pos += len_null + 2;
            return;
        }
        set_positions(node->left, get_len, set_positions);
        node->left_pos = pos;
        pos += get_len(node->data) + 2;
        node->right_pos = pos;
        set_positions(node->right, get_len, set_positions);
    };
    set_positions(start_node, get_len, set_positions);
    // Лямбда функция для определения центра узла
    auto get_center = [&len_null](NodePtr node)->int{
        return (node->right_pos == -1 ? (2 * node->left_pos +
len_null + 2)/2 :
                                                    (node->left_pos + node-
>right_pos)/2 );
    };
    // Очередь для вывода дерева в КЛП порядке
    std::deque<NodePtr> nodes;
    nodes.emplace_back(start_node); // Помещаем начальный узел в
очередь
    /* Лямбда функция для вывода дерева. max_iters определяет
маскимальную глубину дерева.
    * new_node выводится жёлтым цветом.
    */
    auto print_tree = [&os, &nodes, &len_null](int max_iters, NodePtr
new_node, auto&& get_center)->void{

```

```

int iter = 0; // Текущее количество итераций
// Выводить пока очередь не пуста
while(!nodes.empty()){
    int offset = nodes.size(); // Необходимо для определения
индекса сыновей
    // Добавление сыновей узлов, находящихся в очереди
    for(auto node : nodes){
        if(node->right_pos != -1){ // Если не nullptr узел
            if(!node->left){ // Если левый сын -- nullptr
                NodePtr null_node =
std::make_shared<RedBlackTreeNode<T>>();
                null_node->left_pos = node->left_pos -
len_null - 2;

                null_node->right_pos = -1;
                nodes.emplace_back(null_node);
            }else nodes.emplace_back(node->left);
            if(!node->right){ // Если правый сын -- nullptr
                NodePtr null_node =
std::make_shared<RedBlackTreeNode<T>>();
                null_node->left_pos = node->right_pos;
                null_node->right_pos = -1;
                nodes.emplace_back(null_node);
            }else nodes.emplace_back(node->right);
        }
    }
    int last_index = 0;
    int last_pos = 0;
    // Вывод верхней части узлов
    for(int i = 0; i < offset; i++){
        NodePtr node = nodes[i];
        os << std::wstring(node->left_pos - last_pos, L' ');
        if(node->right_pos == -1){ // Если узел nullptr
            os << kBLACK << kLEFT_UPPER_CORNER <<
std::wstring(get_center(node) - node->left_pos - 1, kHORIZONTAL)
            << kCONNECT_UP << std::wstring(len_null -
get_center(node) + node->left_pos, kHORIZONTAL) << kRIGHT_UPPER_CORNER <<
kNORMAL;

            last_pos = node->left_pos + len_null + 2;
        }else{ // Узел не nullptr
            os << (node == new_node ? kORANGE : (node->is_red
? kRED : kBLACK));
            os << kLEFT_UPPER_CORNER <<
std::wstring(get_center(node) - node->left_pos - 1, kHORIZONTAL)
            << (node->parent.lock() ? kCONNECT_UP :
kHORIZONTAL)
            << std::wstring(node->right_pos -
get_center(node) - 2, kHORIZONTAL) << kRIGHT_UPPER_CORNER;
            os << kNORMAL;
            last_pos = node->right_pos;
        }
    }
    os << L"\n";
    last_pos = 0;
    // Вывод серединной части узлов и веток к сыновьям
    for(int i = 0; i < offset; i++){
        NodePtr node = nodes[i];
        if(node->right_pos == -1){ // Если узел nullptr

```

```

        os << std::wstring(node->left_pos - last_pos, L'
')
        << kBLACK << kVERTICAL << kNULL << kVERTICAL <<
kNORMAL;
        last_pos = node->left_pos + len_null + 2;
    }else{ // Узел не nullptr
        int left_center = get_center(nodes[offset + 2 *
last_index]);
        int right_center = get_center(nodes[offset + 2 *
last_index + 1]);
        os << std::wstring(left_center - last_pos, L' ')
        << (node == new_node ? kORANGE : (node->is_red ?
kRED : kBLACK))
        << kLEFT_UPPER_CORNER
        << std::wstring(node->left_pos - left_center - 1,
kHORIZONTAL)
        << kCONNECT_LEFT << node->data << kCONNECT_RIGHT
        << std::wstring(right_center - node->right_pos,
kHORIZONTAL)
        << kRIGHT_UPPER_CORNER
        << kNORMAL;
        last_pos = right_center + 1;
        last_index++;
    }
}
os << L"\n";
last_pos = 0;
last_index = 0;
// Вывод нижней части узлов
for(int i = 0; i < offset; i++){
    NodePtr node = nodes[i];
    if(node->right_pos == -1){ // Если узел nullptr
        os << std::wstring(node->left_pos - last_pos, L'
')
        << kBLACK << kLEFT_DOWN_CORNER <<
std::wstring(len_null, kHORIZONTAL) << kRIGHT_DOWN_CORNER << kNORMAL;
        last_pos = node->left_pos + len_null + 2;
    }else{ // Узел не nullptr
        int left_center = get_center(nodes[offset + 2 *
last_index]);
        int right_center = get_center(nodes[offset + 2 *
last_index + 1]);
        os << std::wstring(left_center - last_pos, L' ')
        << kBLUE << kVERTICAL << kNORMAL
        << std::wstring(node->left_pos - left_center - 1,
L' ')
        << (node == new_node ? kORANGE : (node->is_red ?
kRED : kBLACK))
        << kLEFT_DOWN_CORNER
        << std::wstring(node->right_pos - node->left_pos
- 2, kHORIZONTAL)
        << kRIGHT_DOWN_CORNER << kNORMAL
        << std::wstring(right_center - node->right_pos,
L' ')
        << kBLUE << kVERTICAL << kNORMAL
        << kNORMAL;
        last_pos = right_center + 1;
        last_index++;
    }
}

```

```

        }
    }
    os << L"\n";
    // Удаление узлов, которые были выведены
    for(int i = 0; i < offset; i++) nodes.pop_front();
    // Вывод аертикальных частей веток к сыновьям
    for(int i = 0; i < 3; i++){
        last_pos = 0;
        for(auto node : nodes){
            int center = get_center(node);
            os << std::wstring(center - last_pos, L' ');
            os << kBLUE << kVERTICAL << kNORMAL;
            last_pos = center + 1;
        }
        os << L"\n";
    }
    // Если достигли максимума итерация, то завершить вывод
    if(++iter == max_iters) break;
}

};
print_tree((is_start_head ? -1 : 4), new_node, get_center);

}
#endif

public:
    RedBlackTree() = default; // Конструктор по умолчанию
    ~RedBlackTree() = default; // Деструктор по умолчанию

    // Конструкотр с выводом
    RedBlackTree(std::wostream& os){
        out = os;
    }

    // Конструктор копирования
    RedBlackTree(const RedBlackTree<T>& tree){
        auto Copy = [](NodePtr parent, NodePtr& dest, const NodePtr& src,
auto&& Copy)->void{
            if(src == nullptr) return;
            dest = std::make_shared<RedBlackTreeNode<T>>(src->data);
            dest->parent = parent;
            Copy(dest, dest->left, src->left, Copy);
            Copy(dest, dest->right, src->right, Copy);
        };
        Copy(nullptr, head, tree.head, Copy);
        out = tree.out;
    }

    // Оператор копирования
    RedBlackTree& operator=(const RedBlackTree<T>& tree){
        if(&tree == this) return *this;
        auto Copy = [](NodePtr parent, NodePtr& dest, const NodePtr& src,
auto&& Copy)->void{
            if(src == nullptr) return;
            dest = std::make_shared<RedBlackTreeNode<T>>(src->data);
            dest->parent = parent;
            Copy(dest, dest->left, src->left, Copy);
            Copy(dest, dest->right, src->right, Copy);

```

```

    };
    Copy(nullptr, head, tree.head, Copy);
    out = tree.out;
    return *this;
}

// Конструктор перемещения
RedBlackTree(RedBlackTree<T>&& tree){
    head = std::move(tree.head);
    out = tree.out;
}

// Оператор перемещения
RedBlackTree& operator=(RedBlackTree<T>&& tree){
    if(&tree == this) return *this;
    head = std::move(tree.head);
    out = tree.out;
    return *this;
}

// Получение потока для вывода
std::wostream& GetOutputStream(){
    return out;
}

// Установка потока для вывода
void SetOutputStream(std::wostream& os){
    out = os;
}

// Вставка элемента в дерево
void Insert(T new_data){
    auto new_node = std::make_shared<RedBlackTreeNode<T>>(new_data);
    // Если дерево пусто
    if(head == nullptr){
        // Новый узел назначить корнем
        head = new_node;
#ifdef PRINT
        out << L"Производится вставка элемента "
        << kORANGE << new_node->data << kNORMAL
        << L", как в обычное бинарное дерево поиска.\n";
        PrintTree(out, new_node, head);
        out << SPACER;
#endif
        // Вызов перебалансировки дерева по новому узлу
        Balance(new_node);
        return;
    }
    NodePtr cur = head;
    while(true){
        // Если значения для вставки уже присутствует
        if(new_data == cur->data){
            // Хамена данных
            cur->data = new_data;
#ifdef PRINT
            out << L"Производится вставка элемента "
            << kORANGE << new_node->data << kNORMAL

```

<< L", как в обычное бинарное дерево поиска. Данный элемент уже существует в дереве, поэтому он заменяется новым. Дополнительные действий для перебалансировки дерева не требуется\n";

```
    PrintTree(out, cur, head);
    out << SPACER;
    #endif
    return;
}
// Если значение для вставки меньше
if(new_data < cur->data){
    if(cur->left == nullptr){
        cur->left = new_node;
        new_node->parent = cur;
        new_node->is_left = true;
        break;
    }else{
        cur = cur->left;
    }
}
// Если значение для вставки больше
if(new_data > cur->data){
    if(cur->right == nullptr){
        cur->right = new_node;
        new_node->parent = cur;
        new_node->is_left = false;
        break;
    }else{
        cur = cur->right;
    }
}
}
#ifdef PRINT
out << L"Производится вставка элемента "
<< kORANGE << new_node->data << kNORMAL
<< L", как в обычное бинарное дерево поиска.\n";
PrintTree(out, new_node, head);
out << SPACER;
#endif
// Вызов перебалансировки дерева по новому узлу
Balance(new_node);
}

// Вывод данных в ЛКП порядке
void PrintData(std::wostream& os = std::wcout){
    auto print = [&os](NodePtr node, auto&& print)->void{
        if(node == nullptr) return;
        print(node->left, print);
        os << node->data << " ";
        print(node->right, print);
    };
    print(head, print);
    os << "\n";
}

#ifdef PRINT
// Вывод дерева
void Print(std::wostream& os = std::wcout){
    PrintTree(os, nullptr, head);
}
```



```

    }
    #else
    // Вывод данных дерева
    void Print(std::wostream& os = std::wcout){
        PrintData(os);
    }
    #endif

    // Поиск элемента
    bool Find(T find_data){
        int count = 0;
        // Рекурсивная лямбда функция для поиска элемента
        auto find = [&find_data, &count](NodePtr node, auto&& find)-
>NodePtr{
            if(node == nullptr) return nullptr;
            if(find_data == node->data) return node;
            if(find_data < node->data) return find(node->left, find);
            else return find(node->right, find);
        };
        NodePtr node = find(head, find);
        #ifdef PRINT
        PrintTree(out, node, head);
        #endif
        return (node != nullptr);
    }

};

// Оператор вывода для красно-чёрного дерева
template<typename T>
std::wostream& operator<<(std::wostream& os, RedBlackTree<T>& rbt){
    rbt.Print(os);
    return os;
}

int main(int argc, char** argv){
    // Установление русской локали
    RedBlackTree<int> rbt;
    #ifdef PRINT
    if(argc > 1){
        // Если первым аргументом передано "test", то запустить программу
        в режиме тестирования
        if(std::string(argv[1]) == "test"){
            #endif
            int count;
            std::wcin >> count; // Получение количество элементов
            // Считывание элементов
            for(int i = 0; i < count; i++){
                int ins;
                std::wcin >> ins;
                rbt.Insert(ins);
            }
            // Считывание элемента для поиска
            std::wcin >> count;
            bool is_find = rbt.Find(count); // Поиск элемента
            rbt.PrintData(); // Вывод данных
            std::wcout << L"Count of element: " << is_find << L"\n";

```

```

        return 0;
#ifdef PRINT
    }
}
setlocale(LC_ALL, "ru_RU.utf8");
// Режим визуализации
std::wstring cmd; // Команда
// Лямбда функция определяющая, является ли строка командой выхода
auto is_quit_cmd = [](std::wstring& cmd)->bool{
    return ((cmd == L"q") || (cmd == L"quit") || (cmd == L"в") ||
    (cmd == L"выйти") ||
        (cmd == L"Q") || (cmd == L"Quit") || (cmd == L"B") || (cmd ==
L"Выйти"));
};
// Лямбда функция определяющая, является ли строка командой вывода
auto is_tree_cmd = [](std::wstring& cmd)->bool{
    return((cmd == L"t") || (cmd == L"tree") || (cmd == L"д") || (cmd
== L"дерево") ||
        (cmd == L"T") || (cmd == L"Tree") || (cmd == L"Д") || (cmd ==
L"Дерево"));
};
// Лямбда функция определяющая, является ли строка командой поиска
auto is_find_cmd = [](std::wstring& cmd)->bool{
    return ((cmd == L"f") || (cmd == L"find") || (cmd == L"н") ||
    (cmd == L"найти") ||
        (cmd == L"F") || (cmd == L"Find") || (cmd == L"N") || (cmd ==
L"Найти"));
};
try{
    while(true){
        // Вывод подсказок
        std::wcout << kRED << L"Красно-" << kBLACK << L"чёрное" <<
kNORMAL << L" дерево. Команды:\n"
            << kBLUE << L"(t)ree" << kNORMAL << L" или " << kBLUE <<
L"(д)ерево" << kNORMAL << L" - вывести текущее дерево\n"
            << kBLUE << L"(q)uit" << kNORMAL << L" или " << kBLUE <<
L"(в)ыйти" << kNORMAL << L" - выйти из программы\n"
            << kBLUE << L"(f)ind" << kNORMAL << L" или " << kBLUE <<
L"(н)айти" << kNORMAL << L" - найти элемент в дереве\n"
            << kBLUE << L"Число" << kNORMAL << L" - вставить элемент в
дерево. Вставить можно только одно число. Все остальные будут
проигнорированы\n"
            << L"Что вы хотите сделать: ";
        std::getline(std::wcin, cmd); // Считывание команды
        try{
            if(is_quit_cmd(cmd)){ // Выход
                std::wcout << kBLUE << L"Программа завершает свою
работу..." << kNORMAL << L"\n";
                break;
            }else if(is_tree_cmd(cmd)){ // Вывод
                std::wcout << L"Текущее дерево:\n" << rbt;
            }else if(is_find_cmd(cmd)){ // Поиск
                while(true){
                    try{
                        // Вывод подсказки
                        std::wcout << kORANGE << L"Поиск элемента."
<< kNORMAL << L"\n Введите "

```

```

        << kBLUE << L"число" << kNORMAL << L" для
поиска, " << kBLUE << L"(q)uit"
        << kNORMAL << L" или " << kBLUE << L"(в)ыйти"
<< kNORMAL << L" для выхода из поиска\n"
        << L"Что вы хотите сделать: ";
        std::getline(std::wcin, cmd); // Считывание
команды

        if(is_quit_cmd(cmd)){ // Выход из поиска
            std::wcout << L"Выход из поиска" <<
SPACER;

            break;
        }else{ // Поиск элемента
            int elem = std::stoi(cmd);
            std::wcout << L"Поиск элемента " <<
kORANGE << elem << kNORMAL << L"... \n";
            bool is_find = rbt.Find(elem);
            std::wcout << L"Элемент " << kORANGE <<
elem << kNORMAL
                << (!is_find ? kRED : kBLUE) << (!is_find
? L" отсутствует" : L" присутствует")
                << kNORMAL << L" в дереве" << SPACER;
        }
    }catch(std::invalid_argument& e){ // Некорректная
команда

        std::wcout << kRED << L"Некорректная
команда!" << kNORMAL << SPACER;
    }
}
    }else{ // Вставка элемента
        int elem = std::stoi(cmd);
        std::wcout << L"Вставка элемента " << kRED << elem <<
kNORMAL << SPACER;
        rbt.Insert(elem);
    }
    }catch(std::invalid_argument& e){ // Некорректная команда
        std::wcout << kRED << L"Некорректная команда!" << kNORMAL
<< SPACER;
    }
}
    }catch(std::exception& e){ // Обработка исключительной ситуации
        std::wcout << L"Во время работы программы возникла ошибка: " <<
kRED << e.what() << kNORMAL << L"\n";
    }
    return 0;
#endif
}

```