

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: Красно-черное дерево — вставка. Демонстрация

Студент гр. 9381

Авдеев Илья

Преподаватель

Фирсов М.А.

Санкт-Петербург

2020

ЗАДАНИЕ НА КУРСОВУЮ РАБОТУ

Студент: Авдеев Илья

Группа: 9381

Тема работы:

Вариант 27. Красно-черное дерево — вставка. **Демонстрация**

Исходные данные:

Файл с последовательностью чисел

Содержание пояснительной записки:

«Содержание», «Введение», «Формальная постановка задачи», «Описание алгоритма», «Описание структур данных и функций», «Заключение», «Список использованных источников»

Предполагаемый объем пояснительной записки:

Не менее 20 страниц.

Дата выдачи задания: 31.10.2020

Дата сдачи реферата: 29.12.2020

Дата защиты реферата: 29.12.2020

Студент гр. 9381

Авдеев Илья

Преподаватель

Фирсов М.А.

АННОТАЦИЯ

Задача курсовой работы состоит в реализации БДП красно-черное дерево и демонстрации вставки элемента в него. В качестве интерфейса для пользователя было решено реализовать консольный интерфейс.

Курсовая работа состоит из пояснительной записки и исходного кода разработанной программы.

В ходе работы была разработана программа с консольным интерфейсом для построения БДП красно-черное дерево. Для написания программы использовался язык программирования C++.

SUMMARY

The task of the course work is to implement the red-black tree BST and demonstrate the insertion of an element into it. As a user interface, it was decided to implement a console interface.

The course work consists of an explanatory note and the source code of the developed program.

In the course of the work, a program with a console interface was developed for building a red-black tree BST. The C++programming language was used to write the program.

СОДЕРЖАНИЕ

	Введение	5
1.	Формальная постановка задачи	7
2.	Описание алгоритма	8
2.1.	Первый случай	8
2.2.	Второй случай	9
2.3.	Третий случай	9
3	Демонстрация	11
3.1	Вставка в пустое дерево	11
3.2	Первый случай	11
3.3	Второй случай	12
3.4	Третий случай	14
4.	Описание структур данных и функций	15
4.1.	Перечисление RBtree	15
4.2.	Перечисление node	18
4.3.	Функция main	19
5.	Описание интерфейса пользователя	20
	Заключение	21
	Список использованных источников	22
	Приложение А. Тестирование	23
	Приложение Б. Исходный код программы	27

ВВЕДЕНИЕ

Цель работы.

Реализация БДП Красно-черное дерево и демонстрация вставки элемента в него.

Задачи.

- Изучение такой структуры данных как БДП, конкретно красно-черное дерево.
- Изучение алгоритмов балансирования БДП;
- Написание исходного кода программы;
- Сборка программы;
- Тестирование программы.

Основные теоретические положения.

Красно-черное дерево - это еще одна форма сбалансированного бинарного поискового дерева. Впервые оно было представлено в 1972 году как еще одна разновидность сбалансированного бинарного дерева. Время поиска, вставки или удаления узла для красно-черного дерева является логарифмической функцией от числа узлов.

Данный тип деревьев отличается свойствами:

1. Узел может быть либо красным, либо чёрным и имеет двух потомков;
2. Корень — как правило чёрный. Это правило слабо влияет на работоспособность модели, так как цвет корня всегда можно изменить с красного на чёрный;
3. Все листья, не содержащие данных — чёрные.
4. Оба потомка каждого красного узла — чёрные.
5. Любой простой путь от узла-предка до листового узла-потомка содержит одинаковое число чёрных узлов.

Благодаря этим ограничениям, путь от корня до самого дальнего листа не более чем вдвое длиннее, чем до самого ближнего и дерево примерно сбалансировано. Операции вставки, удаления и поиска требуют в худшем случае времени, пропорционального длине дерева, что позволяет красно-чёрным деревьям быть более эффективными в худшем случае, чем обычные двоичные деревья поиска.

Пусть для красно-чёрного дерева T число чёрных узлов от корня до листа равно B . Тогда кратчайший возможный путь до любого листа содержит B узлов и все они чёрные. Более длинный возможный путь может быть построен путём включения красных узлов. Однако, благодаря п.4 в дереве не может быть двух красных узлов подряд, а согласно пп. 2 и 3, путь начинается и кончается чёрным узлом. Поэтому самый длинный возможный путь состоит из $2B-1$ узлов, попеременно красных и чёрных.

1.ФОРМАЛЬНАЯ ПОСТАНОВКА ЗАДАЧИ

На вход подается последовательность чисел из которых требуется построить самобалансирующееся двоичное дерево поиска «Красно-черное дерево». Для этого должен быть реализован метод вставки узла в бинарное дерево поиска со значением, красящий узлы в красный и черный цвет. При этом корневой узел, должен быть всегда окрашен в черный. Длина пути от корня до всех листьев, должна быть одинаковой. Красные узлы не учитываются в пути. Два красных узла не могут идти друг за другом.

2. ОПИСАНИЕ АЛГОРИТМА

Новый узел в красно-чёрное дерево добавляется на место одного из листьев, окрашивается в красный цвет и к нему прикрепляется два листа (так как листья являются абстракцией, не содержащей данных, их добавление не требует дополнительной операции). Что происходит дальше, зависит от цвета близлежащих узлов.

2.1 Первый случай

У черного родителя есть два красных потомка, один из них имеет красного потомка, тогда создается ситуация:

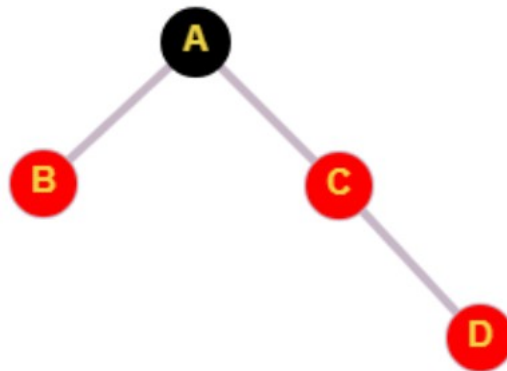


Рисунок 1: Ситуация 1

Чтобы сбалансировать такой узел, нужно перекрасить двух потомков черного родителя в черный, а родителя в красный.

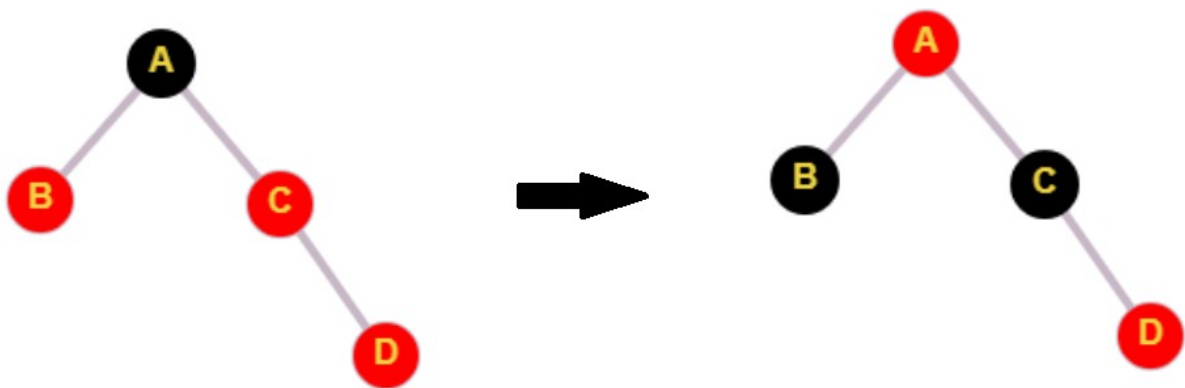


Рисунок 2: Перекрашивание

2.2 Второй случай

У черного узла есть черный потомок и красный, левый потомок красного — красный, тогда создается ситуация:

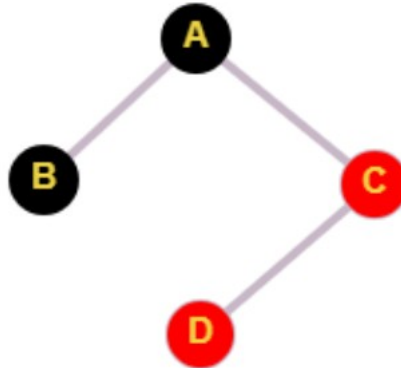


Рисунок 3: Ситуация 2

Чтобы сбалансировать такой узел, нужно сделать переворот без покраски вершин.

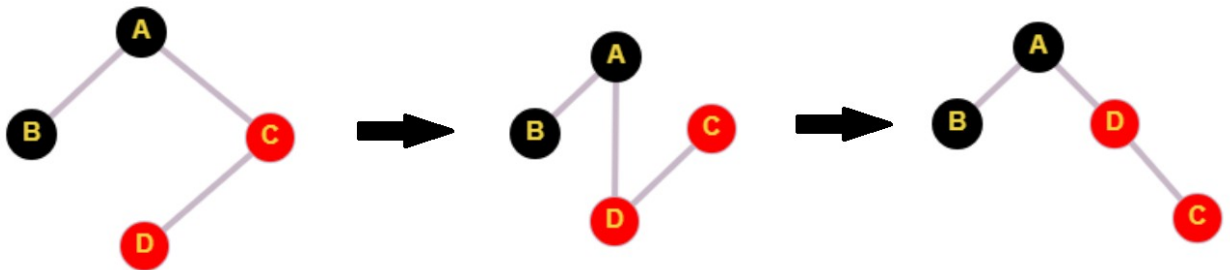


Рисунок 4: Переворот

Создается Третий случай.

2.3 Третий случай

У черного узла один из красных потомков имеет красного потомка

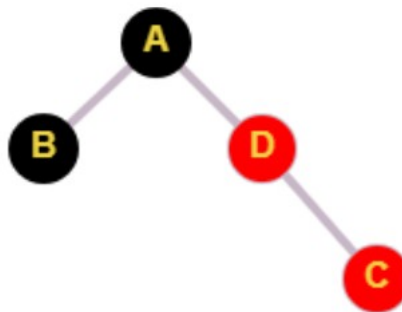


Рисунок 5: Ситуация 3

Чтобы сбалансировать такой узел нужно:

1. Перекрасить узел в красный, а красного потомка в черный

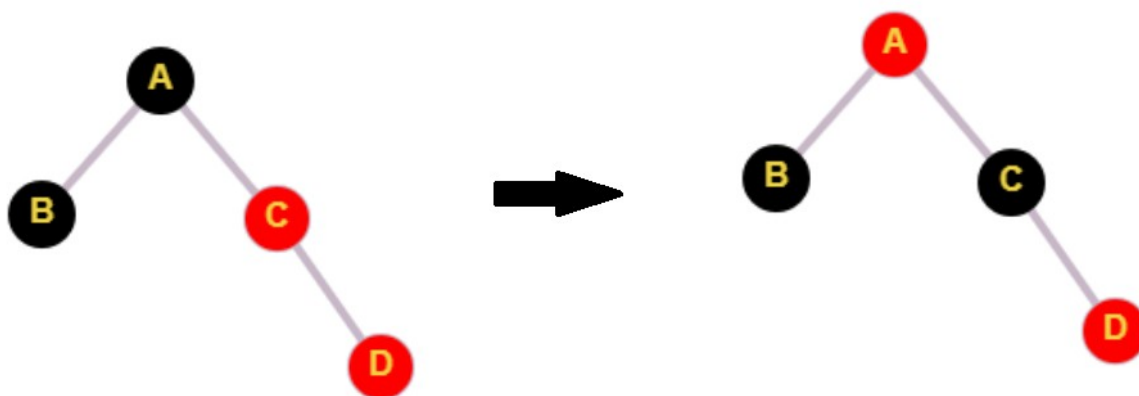


Рисунок 6: Перекрашивание

2. Выполнить переворот

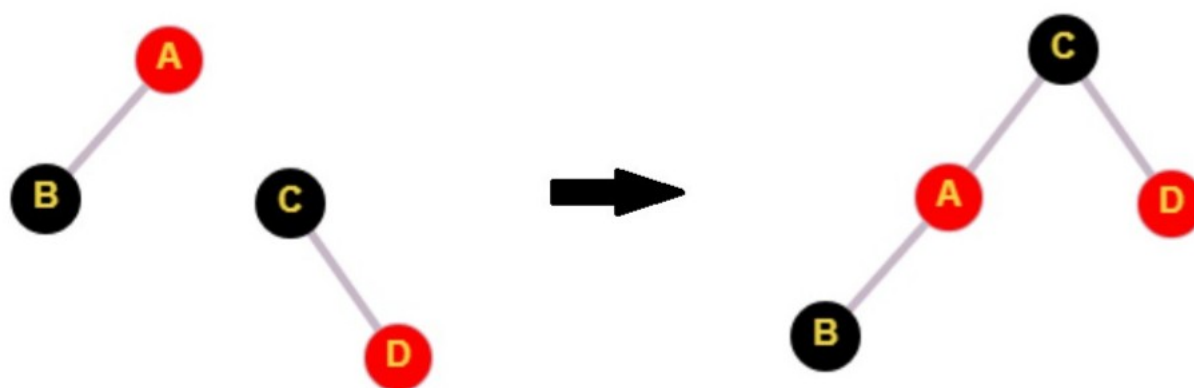
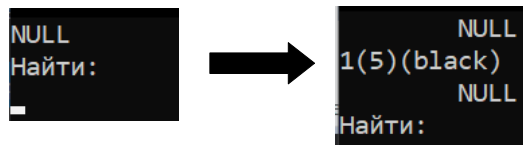


Рисунок 7: Переворот

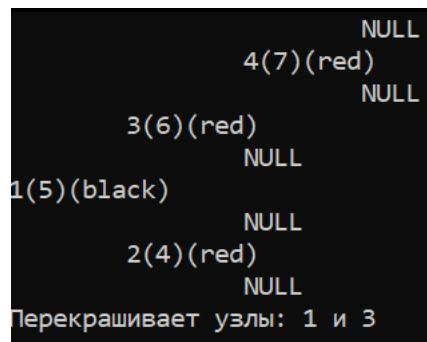
3.ДЕМОНСТРАЦИЯ

3.1 Вставка элемента в пустое дерево

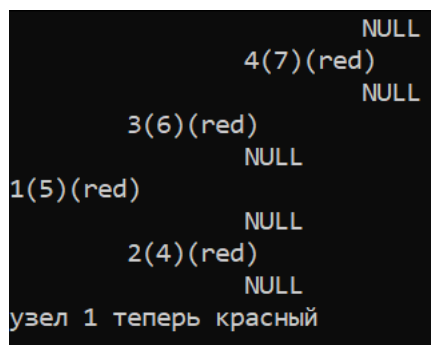


3.2 Рассмотрим случай 1.

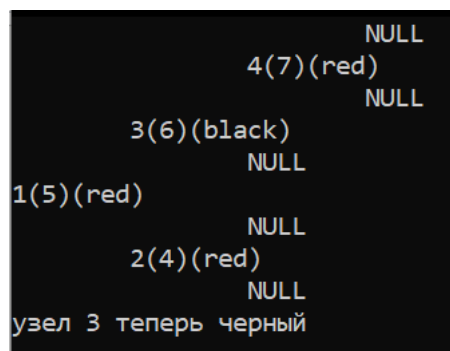
1.



2.



3.



4.

```
                NULL
              4(7)(red)
                NULL
            3(6)(black)
              NULL
1(5)(red)
              NULL
            2(4)(red)
              NULL
узел 3 теперь черный
узел 2 красный, значит нужно покрасить его в черный, а узел 4 в красный
```

5.

```
                NULL
              4(7)(red)
                NULL
            3(6)(black)
              NULL
1(5)(black)
              NULL
            2(4)(black)
              NULL
Найти:
```

3.3 Рассмотрим случай 2.

1.

```
                NULL
              2(10)(red)
                NULL
              3(9)(red)
                NULL
1(7)(black)
  NULL
Обнаружена ситуация! Находится в узле 1
```

2.

```
                NULL
              2(10)(red)
                NULL
              3(9)(red)
                NULL
1(7)(black)
  NULL
```

3.

```
                NULL
              2(10)(red)
                NULL
            3(9)(red)
              NULL
1(7)(black)
  NULL
Перекрашивает узлы: 1 и 3
```

4.

```
                NULL
              2(10)(red)
                NULL
            3(9)(red)
              NULL
1(7)(red)
  NULL
узел 1 теперь красный ■
```

5.

```
                NULL
              2(10)(red)
                NULL
            3(9)(black)
              NULL
1(7)(red)
  NULL
узел 3 теперь черный
```

6.

```
                NULL
              2(10)(red)
                NULL
            3(9)(black)
              NULL
1(7)(red)
  NULL
узел 3 теперь черный
Осуществляет поворот влево относительно узла 1
```

7.

```
                NULL
              2(10)(red)
                NULL
            3(9)(black)
              NULL
              1(7)(red)
                NULL
```

3.4 Рассмотрим случай 3.

1.

```
      NULL
1(5)(black)
      NULL
    2(4)(red)
      NULL
    3(3)(red)
      NULL
Перекрашивает узлы: 1 и 2
```

2.

```
      NULL
1(5)(red)
      NULL
    2(4)(red)
      NULL
    3(3)(red)
      NULL
узел 1 теперь красный
```

3.

```
      NULL
1(5)(red)
      NULL
    2(4)(black)
      NULL
    3(3)(red)
      NULL
узел 2 теперь черный
```

4.

```
      NULL
1(5)(red)
      NULL
    2(4)(black)
      NULL
    3(3)(red)
      NULL
узел 2 теперь черный
Осуществляет поворот вправо относительно узла 1
```

5.

```
      NULL
    1(5)(red)
      NULL
2(4)(black)
      NULL
    3(3)(red)
      NULL
```

4. ОПИСАНИЕ СТРУКТУР ДАННЫХ И ФУНКЦИЙ

Для решения поставленной задачи был написан класс *RBtree* для хранения БДП красно-черное дерева.

В программе используются следующие синонимы

- *left* – Левый потомок узла
- *right* – Правый потомок узла
- *lleft, lright* – потомки левого узла
- *rleft, rright* – потомки правого узла

Результаты тестирования см. в приложении А.

Разработанный программный код см. в приложении Б.

4.1. Класс RBtree

Класс предоставляет функционал для хранения, построения, удаления и вывода БДП красно-черного дерева. Поля и методы класса приведены в таблицах 1 и 2.

Таблица 1 - Поля класса *RBtree*

Модификатор доступа	Тип и название поля	Предназначение	Значение по умолчанию
<i>private</i>	<i>Struct node</i>	Типа данных узла. Поля и методы приведены в таблице 3	-
<i>private</i>	<i>node* tree root</i>	Указатель на корень дерева	<i>NULL</i>
<i>private</i>	<i>int size</i>	Хранит число узлов дерева	<i>0</i>

Таблица 2 - Методы класса *RBtree*

Модификатор доступа	Возвращаемое значение	Название метода и принимаемые аргументы	Описание
---------------------	-----------------------	---	----------

<i>private</i>	<i>node*</i>	<i>make_node(int value)</i>	Принимает value - значение. Создает узел с этим значением и красит его в красный цвет. Возвращает созданный узел.
<i>private</i>	<i>void</i>	<i>del_node(node*)</i>	Принимает node* - узел дерева и удаляет его.
<i>private</i>	<i>void</i>	<i>clear(node*)</i>	Является рекурсивным методом. Принимает на вход *root – корень дерева, которое нужно удалить. Удаляет дерево целиком с помощью метода <i>del node</i> .
<i>private</i>	<i>node*</i>	<i>rotate_right(node*)</i>	Принимает на вход node* - узел. Осуществляет поворот поддерева этого узла вправо. Возвращает родительский узел.
<i>private</i>	<i>node*</i>	<i>rotate_left(node*)</i>	Принимает на вход node* - узел. Осуществляет поворот поддерева этого узла влево. Возвращает родительский узел.
<i>private</i>	<i>void</i>	<i>balance_insert(node**)</i>	Является рекурсивным методом. Принимает на вход node** - узел. Производит поиск ситуаций балансировки дерева после добавления нового узла.
<i>private</i>	<i>bool</i>	<i>balance_remove_case1(node**)</i>	Является рекурсивным методом. Принимает на вход node** - узел. Производит поиск ситуаций балансировки дерева после удаления левого узла. Возвращает true если нужен баланс.
<i>private</i>	<i>bool</i>	<i>balance_remove_case2(node**)</i>	Является рекурсивным методом. Принимает на вход node** - узел. Производит поиск ситуаций балансировки

			дерева после удаления правого узла. Возвращает true если нужен баланс.
<i>private</i>	<i>bool</i>	<i>insert(int, node**)</i>	Является рекурсивным методом. Принимает на вход <i>value</i> — значение и <i>node*</i> - узел. Находит место для вставки узла со значением <i>value</i> . При помощи метода <i>make_node</i> создает узел и с помощью метода <i>balance_insert</i> производит балансировку дерева. Возвращает true если нужен баланс.
<i>private</i>	<i>bool</i>	<i>getmin(node**, node**)</i>	Является рекурсивным методом. Принимает на вход <i>**root</i> — корень дерева и <i>*res</i> – узел который был удален. Удаляет узел с максимальным значением. Возвращает true если нужен баланс.
<i>private</i>	<i>bool</i>	<i>remove(node**, int)</i>	Является рекурсивным методом. Принимает на вход <i>*root</i> – корень дерева в котором нужно удалить узел со значением и <i>value</i> – значение, узел с которым нужно удалить. Вызывая себя находит узел, с помощью метода <i>del_node</i> удаляет его, возвращает true если нужен баланс и с помощью методов <i>balance_remove_case1</i> или <i>balance_remove_case2</i> балансирует дерево.
<i>private</i>	<i>bool</i>	<i>print_tree(node*& ptr, int u)</i>	Является рекурсивным методом. Принимает на вход <i>**ptr</i> — корень дерева и <i>u</i> — глубина на котором сейчас обход. Выводит в консоль дерево.

<i>public</i>	<i>void</i>	<i>clear()</i>	Ничего не принимает. Вспомогательный метод для запуска метода <i>clear(node*)</i> .
<i>public</i>	<i>void</i>	<i>insert(int)</i>	Ничего не принимает. Вспомогательный метод для запуска метода <i>insert(int, node**)</i> .
<i>public</i>	<i>void</i>	<i>remove(int);</i>	Принимает на вход value – значение, узел с которым нужно удалить. Вспомогательный метод для запуска метода <i>remove(node*, int)</i> .
<i>public</i>	<i>void</i>	<i>print()</i>	Ничего не принимает. Вспомогательный метод для запуска метода <i>print_tree(node**, int)</i> .
<i>public</i>	<i>int</i>	<i>getsize()</i>	Ничего не принимает. Возвращает количество дерева.
<i>public</i>		<i>RBtree()</i>	Конструктор. Ничего не принимает. Инициализирует корень дерева равным листу.
<i>public</i>		<i>~RBtree()</i>	Деструктор. Ничего не принимает. Освобождает память от дерева с помощью метода <i>clear(node*)</i> .

4.2. Структура Node

Тип данных элемента списка. Поля структуры приведены в таблице 3.

Таблица 3 - Поля структуры node

Модификатор доступа	Тип и название поля	Предназначение	Значение по умолчанию
<i>public</i>	<i>struct node* left</i>	Указатель на левого потомка	<i>NULL</i>
<i>public</i>	<i>struct node* right</i>	Указатель на правого потомка	<i>NULL</i>
<i>public</i>	<i>int value</i>	Значение узла	-
<i>public</i>	<i>bool col</i>	Цвет узла	1

<i>public</i>	<i>int value</i>	Порядковый номер узла	-
---------------	------------------	-----------------------	---

4.3. Функция **main**

Для начала объявляются следующие переменные:

- *n* — вводимое значение для записи в дерево;
- *tree* — Хранит дерево;
- *cin* – переменная для проверки ввода пользователя
- *c* — переменная хранящая выбор пользователя
- *exit* – переменная для проверки условия выхода из программы

Далее производится настройка русского языка для консоли.

Далее происходит вход в цикл выбора способа ввода. Считывается выбранное пользователем действие (цифра от 1 до 3). Если пользователь выбрал создать из случайных чисел дерево, программа входит в цикл ввода размера дерева. Затем программа входит в первый основной цикл программы. В нем, если пользователь выбрал консольный ввод, он будет вводить числа, из которых создается дерево. Если он выбрал случайное дерево, то в цикле от 0 до введенного размера, сгенерируются числа из которых будет состоять дерево. Если выбрал ввод из файла, создастся дерево из элементов полученных из текстового документа "lin.txt".

5. ОПИСАНИЕ ИНТЕРФЕЙСА ПОЛЬЗОВАТЕЛЯ

Существует три варианта запуска инициализации дерева:

1. Консольный — тогда пользователю предлагается вводить элементы в консоль.
2. Заполнение дерева заданного пользователем размера случайно сгенерированными числами.
3. Заполнение дерева считываемыми числами из файла `lin.txt`.

Пользователь выбирает ввести цифру от 1 до 3. В зависимости от выбранного действия выполняется инициализация дерева одним из вышеперечисленных способов.

После заполнения пользователь может самостоятельно дополнить дерево вводя в консоль числа, которые нужно вставить в дерево.

ЗАКЛЮЧЕНИЕ

На основании анализа исследованной литературы были изучены такие структуры данных как самобалансирующиеся двоичные деревья поиска. В ходе работы был реализован класс для работы со структурой данных «красно-черное дерево», содержащий в себе методы балансировки дерева, вставки и удаления в него узлов, а также его отрисовки. Приведенный в работе материал позволяет заключить, что требующая операция покраски дерева после вставки или удаления занимает $O(\log n)$ или $O(1)$ смен цветов, что на практике довольно быстро и не более чем трёх поворотов дерева (для вставки — не более двух). Хотя вставка и удаление сложны, их трудоёмкость остается $O(\log n)$.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. https://neerc.ifmo.ru/wiki/index.php?title=Красно-черное_дерево
2. <https://graphonline.ru>
3. https://ru.wikipedia.org/wiki/Красно-чёрное_дерево

ПРИЛОЖЕНИЕ А

ТЕСТИРОВАНИЕ

Таблица А.1 - Примеры тестовых случаев на некорректных данных

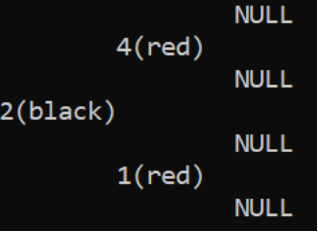
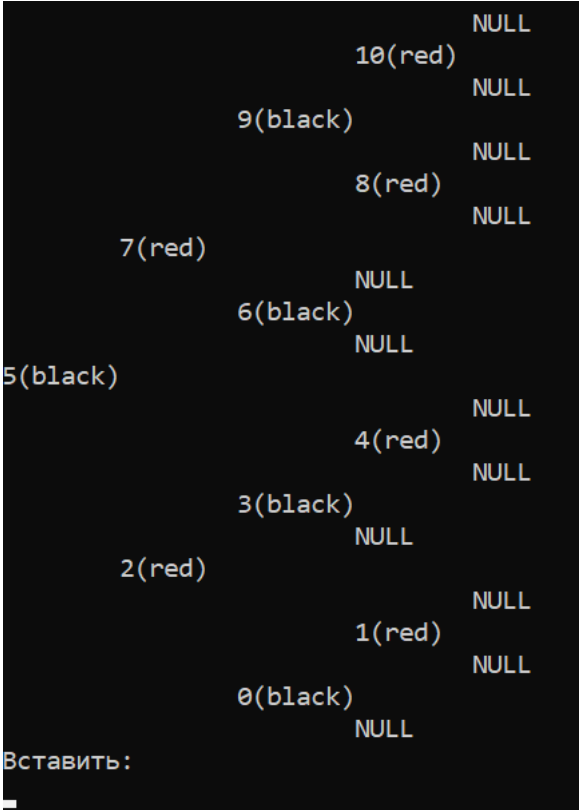
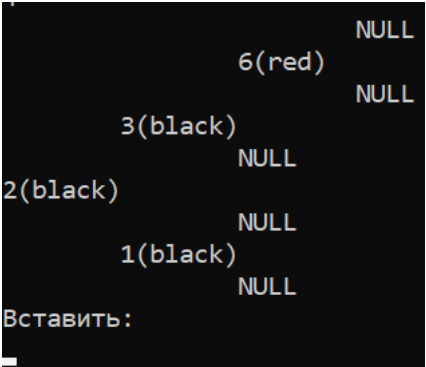
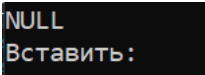
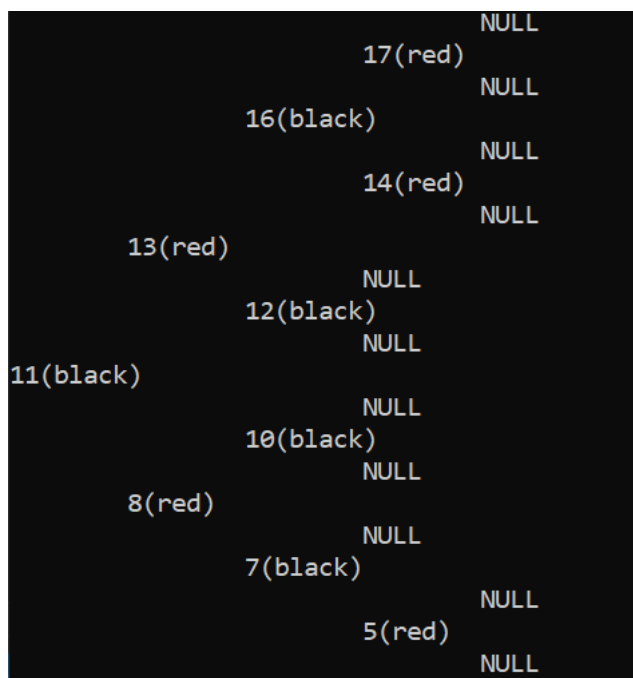
№ п/п	Входные данные	Выходные данные
1.	f	Нужно ввести число, попробуй еще
2.	9	Нужно ввести 1 или 2, чтобы выйти из программы нужно нажать esc
3.	2 2 4	 <pre> 4(red) NULL 2(black) NULL 1(red) NULL </pre>
4.	1 f	Нужно ввести число, попробуй еще
5.	3	 <pre> 10(red) NULL 9(black) NULL 8(red) NULL 7(red) NULL 6(black) NULL 5(black) NULL 4(red) NULL 3(black) NULL 2(red) NULL 1(red) NULL 0(black) NULL Вставить: </pre>

Таблица А.2 - Примеры тестовых случаев на корректных данных

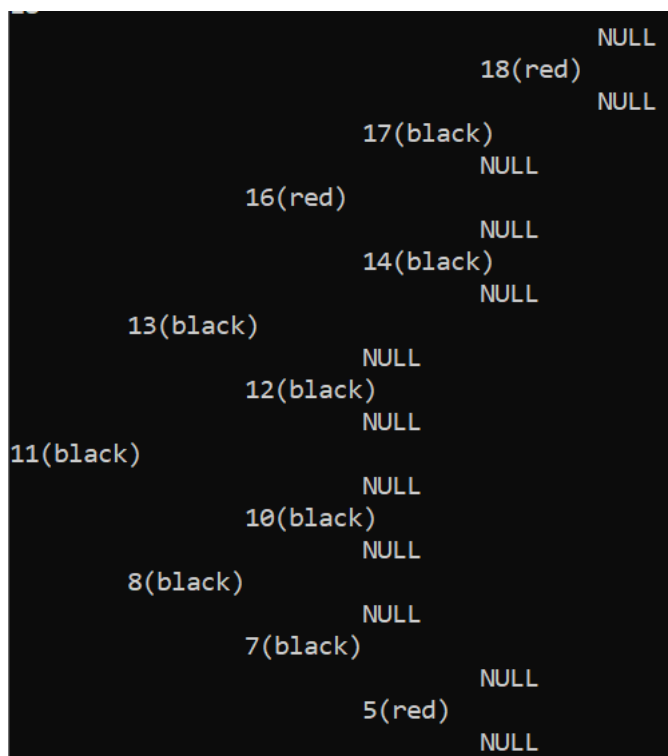
№ п/п	Входные данные	Выходные данные
6.	2 4 0 у	 <p>Выход из программы</p>
7.	8 esc	Выход из программы
8.	2 200	Построено дерево из 200 элементов, заполненное случайными числами от 0 до 400
9.	-1	Нужно ввести число, попробуй еще
10.	1 0	 <p>Пустое дерево</p>

Рассмотрим дерево

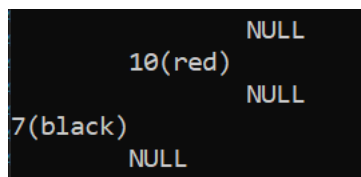


Если в него добавить узел с номером 18, то получится случай 1. В такой ситуации запустится метод балансировки, узел 14 окрасится в черный, узел 16 — в красный, узел 17 — в черный.

Результат



Рассмотрим дерево



Вставив в него узел со значением девять, то получится случай 2, а входе его балансировки получится случай 3.

В ходе балансировки 7 станет родительским узлом для узла 9, а узел 9 для узла 10. Так же узел 9 окрасится в черный, а узел 7 в красный. Узел 9 станет родительским для узлов 7 и 9.

ПРИЛОЖЕНИЕ Б

ИСХОДНЫЙ КОД ПРОГРАММЫ

```
#include <iostream>
#include <time.h>
#include <fstream>
#include <conio.h>

class RBtree
{
    struct node
    {
        node* left, * right;
        int value;
        int num;
        bool red;
    };
    node* tree_root;
    int size;
private:
    node* make_node(int value);
    void del_node(node*);
    void clear(node*);
    node* rotate_right(node*);
    node* rotate_left(node*);
    void balance_insert(node**);
    void balance_insert_demo(node**);
    bool balance_remove_case1(node**);
    bool balance_remove_case2(node**);
    bool insert(int, node**);
    bool insert_demo(int, node**);
    bool getmin(node**, node**);
    bool remove(node**, int);
    void print_tree(node*& ptr, int u);
public:
    RBtree();
    ~RBtree();
    void clear();
    int find(int);
    int findlog(int);
    void insert(int);
    void insert_demo(int);
    void remove(int);
```

```

        void print();
        int getsize() { return size; }
};

RBtree::RBtree()
{
    tree_root = 0;
    size = 0;
}

RBtree::~~RBtree()
{
    clear(tree_root);
}

RBtree::node* RBtree::make_node(int value)
{
    size++;
    node* n = new node;
    n->num = size;
    n->value = value;
    n->left = n->right = NULL;
    n->red = true;
    return n;
}

void RBtree::del_node(node* node)
{
    size--;
    delete node;
}

void RBtree::clear(node* node)
{
    if (!node) return;
    clear(node->left);
    clear(node->right);
    del_node(node);
}

RBtree::node* RBtree::rotate_left(node* n)
{
    node* right = n->right;

```

```

        node* rleft = right->left;
        right->left = n;
        n->right = rleft;
        return right;
    }

RBtree::node* RBtree::rotate_right(node* n)
{
    node* left = n->left;
    node* lright = left->right;
    left->right = n;
    n->left = lright;
    return left;
}

void RBtree::balance_insert(node** root)
{
    node* left, * right, * lleft, * lright;
    node* node = *root;
    if (node->red) return;
    left = node->left;
    right = node->right;
    if (left && left->red)
    {
        lright = left->right;
        if (lright && lright->red)
        {
            left = node->left = rotate_left(left);
        }
        lleft = left->left;
        if (lleft && lleft->red)
        {
            node->red = true;
            left->red = false;
            if (right && right->red)
            {
                lleft->red = true;
                right->red = false;
                return;
            }
        }
        *root = rotate_right(node);
        return;
    }
}

```

```

    }

    if (right && right->red)
    {
        lleft = right->left;
        if (lleft && lleft->red)
        {
            right = node->right = rotate_right(right);
        }
        lright = right->right;
        if (lright && lright->red)
        {
            node->red = true;
            right->red = false;
            if (left && left->red)
            {
                lright->red = true;
                left->red = false;
                return;
            }
            *root = rotate_left(node);
            return;
        }
    }
}

void RBtree::balance_insert_demo(node** root)
{
    node* left, * right, * lleft, * lright;
    node* node = *root;
    if (node->red) return;
    left = node->left;
    right = node->right;
    if (left && left->red)
    {
        lright = left->right;
        if (lright && lright->red)
        {
            system("cls");
            this->print();
            std::cout << "Обнаружена ситуация! Находится в узле " <<
(*root)->num << "\n";
            _getch();

```

```

        left = node->left = rotate_left(left);
        system("cls");
        this->print();
        _getch();
    }
    lleft = left->left;
    if (lleft && lleft->red)
    {
        system("cls");
        this->print();
        std::cout << "Перекрашивает узлы: " << node->num << " и
" << left->num << "\n";
        _getch();
        node->red = true;
        system("cls");
        this->print();
        std::cout << "узел " << node->num << " теперь красный";
        _getch();
        left->red = false;
        system("cls");
        this->print();
        std::cout << "узел " << left->num << " теперь черный\n";
        _getch();
        if (right && right->red)
        {
            std::cout << "узел " << right->num << " красный,
значит нужно покрасить его в черный, а узел " << lleft->num << " в красный\n";
            _getch();
            lleft->red = true;
            right->red = false;
            return;
        }
        std::cout << "Осуществляет поворот вправо относительно
узла " << node->num << "\n";
        _getch();
        *root = rotate_right(node);
        system("cls");
        this->print();
        _getch();
        return;
    }
}

```

```

        if (right && right->red)
        {
            lleft = right->left;
            if (lleft && lleft->red)
            {
                system("cls");
                this->print();
                std::cout << "Обнаружена ситуация! Находится в узле " <<
(*root)->num << "\n";
                _getch();
                right = node->right = rotate_right(right);
                system("cls");
                this->print();
                _getch();
            }
            lright = right->right;
            if (lright && lright->red)
            {
                system("cls");
                this->print();
                std::cout << "Перекрашивает узлы: " << node->num << " и
" << right->num << "\n";
                _getch();
                node->red = true;
                system("cls");
                this->print();
                std::cout << "узел " << node->num << " теперь красный ";
                _getch();
                right->red = false;
                system("cls");
                this->print();
                std::cout << "узел " << left->num << " теперь черный\n";
                _getch();
                if (left && left->red)
                {
                    std::cout << "узел " << left->num << " красный,
значит нужно покрасить его в черный, а узел " << lright->num << " в красный\n";
                    _getch();
                    lright->red = true;
                    left->red = false;
                    return;
                }
            }
        }
    }
}

```



```

        }
        std::cout << "Осуществляет поворот влево относительно
узла " << node->num << "\n";
        _getch();
        *root = rotate_left(node);
        system("cls");
        this->print();
        _getch();
        return;
    }
}

}

bool RBtree::balance_remove_case1(node** root)
{
    node* n = *root;
    node* left = n->left;
    node* right = n->right;
    if (right && left->red)
    {
        left->red = false; return false;
    }
    if (right && right->red)
    {
        n->red = true;
        right->red = false;
        n = *root = rotate_left(n);
        if (balance_remove_case1(&n->left)) n->left->red = false;
        return false;
    }
    unsigned int mask = 0;
    node* rleft = right->left;
    node* rright = right->right;
    if (rleft && rleft->red) mask |= 1;
    if (rright && rright->red) mask |= 2;
    switch (mask)
    {
    case 0:
        right->red = true;
        return true;
    case 1:
    case 3:
        right->red = true;

```

```

        rright->red = false;
        right = n->right = rotate_right(right);
        rright = right->right;
    case 2:
        right->red = n->red;
        rright->red = n->red = false;
        *root = rotate_left(n);
    }
    return false;
}

bool RBtree::balance_remove_case2(node** root)
{
    node* n = *root;
    node* left = n->left;
    node* right = n->right;
    if (right && right->red) { right->red = false; return false; }
    if (left && left->red)
    {
        n->red = true;
        left->red = false;
        n = *root = rotate_right(n);
        if (balance_remove_case2(&n->right)) n->right->red = false;
        return false;
    }
    unsigned int mask = 0;
    node* lleft = left->left;
    node* lright = left->right;
    if (lleft && lleft->red) mask |= 1;
    if (lright && lright->red) mask |= 2;
    switch (mask)
    {
    case 0:
        left->red = true;
        return true;
    case 2:
    case 3:
        left->red = true;
        lright->red = false;
        left = n->left = rotate_left(left);
        lleft = left->left;
    case 1:
        left->red = n->red;

```

```

        lleft->red = n->red = false;
        *root = rotate_right(n);
    }
    return false;
}

int RBtree::find(int value)
{
    node* n = tree_root;
    int i = 0;
    while (n)
    {
        if (n->value == value)
            i++;
        n = n->value > value ? n->left : n->right;
    }
    return i;
}

int RBtree::findlog(int value)
{
    int f = 0;

    node* n = tree_root;
    int i = 0;
    while (n)
    {
        std::cout << ++f << " -й запуск цикла\n";
        std::cout << "находится в узле " << n->num << "\n";
        if (n->value == value)
        {
            std::cout << "Элемента найден\n";
            i++;
        }
        if (n->value > value)
        {
            std::cout << "Искомый элемент меньше, чем содержимое
узла поэтому идет в ";
            std::cout << "левый узел\n";
            n = n->left;
        }
        else
        {

```

```

        std::cout << "Искомый элемент больше, чем содержимое
узла поэтому идет в ";
        std::cout << "правый узел\n";
        n = n->right;
    }
}
return i;
}

bool RBtree::insert(int value, node** root)
{
    node* n = *root;
    if (!n) *root = make_node(value);
    else
    {
        // if (value == n->value) return true;
        if (insert(value, value <= n->value ? &n->left : &n->right))
return true;

        balance_insert(root);
    }
    return false;
}

bool RBtree::insert_demo(int value, node** root)
{
    node* n = *root;
    if (!n) *root = make_node(value);
    else
    {
        // if (value == n->value) return true;
        if (insert_demo(value, value <= n->value ? &n->left : &n-
>right)) return true;

        balance_insert_demo(root);
    }
    return false;
}

bool RBtree::getmin(node** root, node** res)
{
    node* node = *root;
    if (node->left)
    {

```

```

        if (getmin(&node->left, res)) return
balance_remove_case1(root);
    }
    else
    {
        *root = node->right;
        *res = node;
        return !node->red;
    }
    return false;
}

bool RBtree::remove(node** root, int value)
{
    node* t, * node = *root;
    if (!node) return false;
    if (node->value < value)
    {
        if (remove(&node->right, value)) return
balance_remove_case2(root);
    }
    else if (node->value > value)
    {
        if (remove(&node->left, value)) return
balance_remove_case1(root);
    }
    else
    {
        bool res;
        if (!node->right)
        {
            *root = node->left;
            res = !node->red;
        }
        else
        {
            res = getmin(&node->right, root);
            t = *root;
            t->red = node->red;
            t->left = node->left;
            t->right = node->right;
            if (res) res = balance_remove_case2(root);
        }
    }
}

```

```

        del_node(node);
        return res;
    }
    return 0;
}

void RBtree::insert(int value)
{
    insert(value, &tree_root);
    if (tree_root) tree_root->red = false;
}

void RBtree::insert_demo(int value)
{
    insert_demo(value, &tree_root);
    if (tree_root) tree_root->red = false;
}

void RBtree::remove(int value)
{
    remove(&tree_root, value);
}

void RBtree::clear()
{
    clear(tree_root);
    tree_root = 0;
}

void RBtree::print_tree(node*& ptr, int u)
{
    if (ptr == nullptr)
    {
        u++;
        for (int i = 0; i < u - 1; ++i) std::cout << "\t";
        std::cout << "NULL\n";
        return;
    }
    else
    {
        print_tree(ptr->right, ++u);
        for (int i = 0; i < u - 1; ++i) std::cout << "\t";
        std::cout << ptr->num << "(" << ptr->value << ") (";
    }
}

```

```

        if (ptr->red)
            std::cout << "red";
        else
            std::cout << "black";
        std::cout << ")\n";
        u--;
    }
    print_tree(ptr->left, ++u);
}

void RBtree::print()
{
    print_tree(tree_root, 0);
}

int withoutlog(RBtree& tree)
{
    char cin[10], exit = 0;
    int n = 1;
    while (n)
    {
        system("cls");
        tree.print();
        std::cout << "Найти: \n";
        std::cin >> cin;
        system("cls");
        if (isdigit(*cin))
        {
            n = atoi(cin);
        }
        else
        {
            std::cout << "Нужно ввести число, попробуйте еще\n";
            continue;
        }
        if (n == 0)
        {
            std::cout << "Закончить или найти 0?\n y - выйти n =
найти\n";

            while (exit != 'y' && exit != 'n')
                exit = _getch();
        }
        if (exit != 'y')
    }
}

```

```

        {
            tree.print();
            if (!tree.find(n))
            {
                std::cout << "Числа в дереве нет, ввести его?\n у
- да n - нет\n";

                while (*cin != 'y' && *cin != 'n')
                    *cin = _getch();
                if (*cin == 'y')
                    tree.insert(n);
                system("cls");
            }
            else
            {
                std::cout << "Число в дереве есть, все равно
ввести его?\n у - да n - нет\n";

                while (*cin != 'y' && *cin != 'n')
                    *cin = _getch();
                if (*cin == 'y')
                    tree.insert(n);
                system("cls");
            }
        }

        if (exit == 'n')
            n = 1;
        else
            return 0;
        exit = 0;
    }
}

int withlog(RBtree& tree)
{
    char cin[10], exit = 0;
    int n = 1;
    while (n)
    {
        system("cls");
        tree.print();
        std::cout << "Найти: \n";
        std::cin >> cin;
        system("cls");
    }
}

```



```

        if (isdigit(*cin))
        {
            n = atoi(cin);
        }
        else
        {
            std::cout << "Нужно ввести число, попробуйте еще\n";
            continue;
        }
        if (n == 0)
        {
            std::cout << "Закончить или найти 0?\n у - найти n =
выйти\n";

            while (exit != 'y' && exit != 'n')
                exit = _getch();
        }
        if (exit != 'n')
        {
            tree.print();
            if (!tree.findlog(n))
            {
                std::cout << "Числа в дереве нет, ввести его?\n у
- да n - нет\n";

                while (*cin != 'y' && *cin != 'n')
                    *cin = _getch();
                if (*cin == 'y')
                    tree.insert_demo(n);
            }
            else
            {
                std::cout << "Число в дереве есть, все равно
ввести его?\n у - да n - нет\n";

                while (*cin != 'y' && *cin != 'n')
                    *cin = _getch();
                if (*cin == 'y')
                    tree.insert_demo(n);
            }
        }

        if (exit != 'n')
            n = 1;
        else

```

```

        return 0;
    exit = 0;
}

}

int main()
{
    int n = 1, c, j = 20;
    char cin[10], exit = 0;
    RBtree tree;
    setlocale(LC_ALL, "ru");

    std::cout << "Способ ввода из консоли 1, заполнить дерево случайными
числами 2, ввести из файла 3\n";
    std::cin >> cin;
    //Выбор способа инициализации
    while (exit != 27)
    {
        if (isdigit(*cin))
        {
            c = atoi(cin);
            if ((c != 1) && (c != 2) && (c != 3))
            {
                std::cout << "нужно ввести 1 или 2 или 3, чтобы
выйти из программы нужно нажать esc\n";
                exit = _getch();
                if (exit == 27)
                    break;
            }
            else
                break;
            std::cout << "Способ ввода из консоли 1, Заполнить
дерево случайными числами 2\n";
            std::cin >> cin;
            continue;
        }
        else
        {
            std::cout << "Нужно ввести число, попробуйте еще\n";
            std::cin >> cin;
        }
    }
}

```

```

if (exit != 27) //Выбор размеров случайного дерева
    while (1)
    {
        if (c == 2)
        {
            std::cout << "сколько узлов создать?\n";
            std::cin >> cin;
            if (isdigit(*cin))
            {
                j = atoi(cin);
                break;
            }
            else
                std::cout << "Нужно ввести число, попробуйте
еще\n";
        }
        else
            break;
    }

std::ifstream lin("lin.txt");
if (c == 3)
    if (!lin.is_open())
    {
        std::cout << "Файл не открыт";
        return -1;
    }

srand(time(0));
//Инициализация
while (n)
{
    switch (c)
    {
        case 1:
            std::cin >> cin;
            if (isdigit(*cin))
            {
                n = atoi(cin);
            }
            else
            {

```

```

        std::cout << "Нужно ввести число, попробуйте еще\
n";

        continue;
    }
    while (n != NULL)
    {
        tree.insert(n);
        std::cin >> cin;
        if (isdigit(*cin))
        {
            n = atoi(cin);
        }
        else
        {
            std::cout << "Нужно ввести число, попробуйте
еще\n";

            continue;
        }
    }
    break;
case 2:
    while (tree.getsize() < j)
    {
        n = rand() % j + rand() % j;
        tree.insert(n);
    }
    n = 0;
    break;
case 3:
    while (!lin.eof())
    {
        lin >> n;
        tree.insert(n);
    }
default:
    n = 0;
    break;
}
}
lin.close();

```

```

while (exit != 27)
{
    std::cout << "Запуск программы с логами 1, Без логов 2\n";
    std::cin >> cin;
    if (isdigit(*cin))
    {
        c = atoi(cin);
        if ((c != 1) && (c != 2) && (c != 3))
        {
            std::cout << "нужно ввести 1 или 2, чтобы выйти из
программы нужно нажать esc\n";
            exit = _getch();
            if (exit == 27)
                break;
        }
        else
            break;
        continue;
    }
    else
    {
        std::cout << "Нужно ввести число, попробуйте еще\n";
        std::cin >> cin;
    }
}

system("cls");

//Запуск программы без логов
try
{
    if (exit != 27)
        if (c == 2)
            withoutlog(tree);
}
catch (const std::exception&)
{
    throw "Ошибка: Программа без логов не запущена";
}

system("cls");

```

```

//Запуск программы с логами
try
{
    if (exit != 27)
        if (c == 1)
            withlog(tree);
}
catch (const std::exception&)
{
    throw "Ошибка: Программа с логами не запущена";
}

getchar();
return 0;
}

```