

**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**

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

Исходные данные:

Вставка в БДП красно-черное дерево.

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: 28.12.2020

Дата защиты реферата: 29.12.2020

Студент		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
	Формальная постановка задачи	7
1.	Описание алгоритма	8
1.1.	Первый случай	8
1.2.	Второй случай	9
1.3.	Третий случай	9
2.	Описание структур данных и функций	11
2.1.	Перечисление RBtree	11
2.2.	Перечисление node	15
2.3.	Функция main	15
3.	Описание интерфейса пользователя	16
	Заключение	17
	Список использованных источников	18
	Приложение А. Тестирование	19
	Приложение Б. Исходный код программы	23

## **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.

## **ФОРМАЛЬНАЯ ПОСТАНОВКА ЗАДАЧИ**

На вход подается файл или в консоль вводятся числа. Требуется из вводимых данных построить БДП. Для этого потребуется написать функции красящие и балансирующие бинарное дерево.

## 1. 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.

### 1.1 Первый случай

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

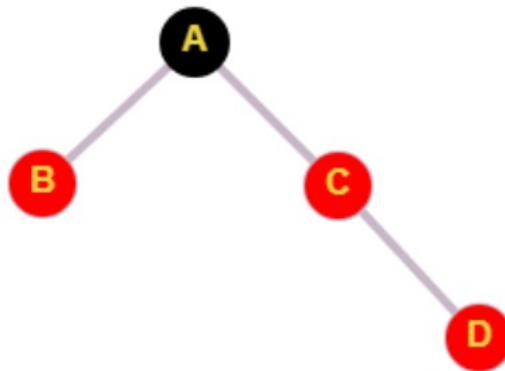


Рисунок 1: Ситуация 1

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

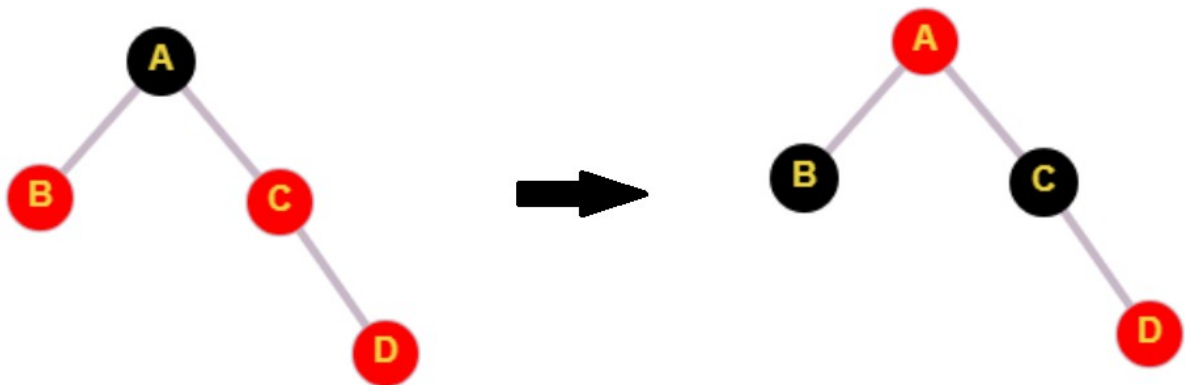


Рисунок 2: Перекрашивание



## 1.2 Второй случай

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

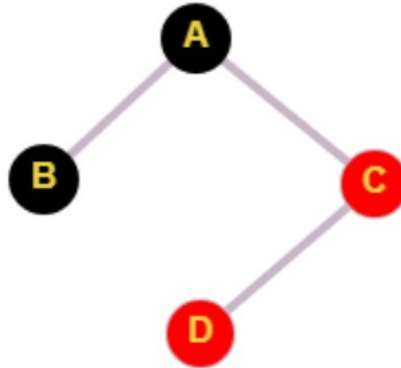


Рисунок 3: Ситуация 2

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

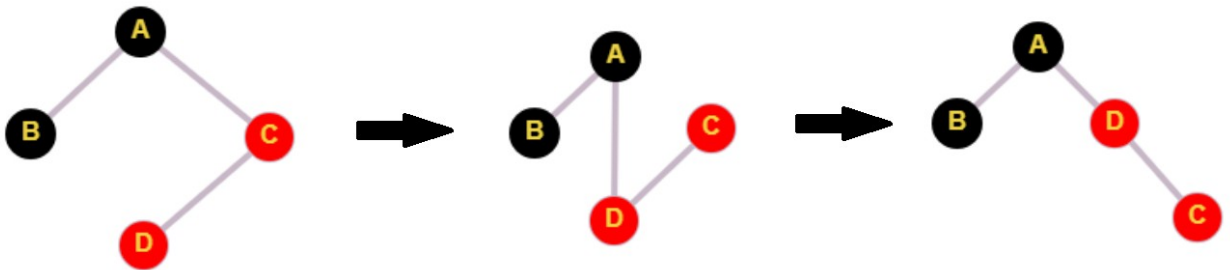


Рисунок 4: Переворот

Создается Третий случай.

## 1.3 Второй случай

У черного узла один из красных потомков имеет красного потомка

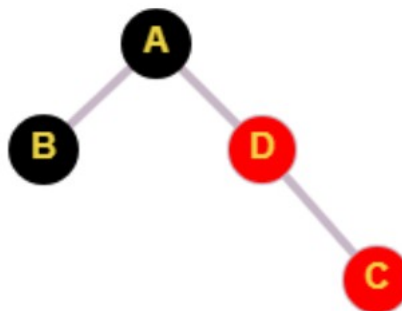


Рисунок 5: Ситуация 3

Чтобы сбалансировать такой узел нужно:

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

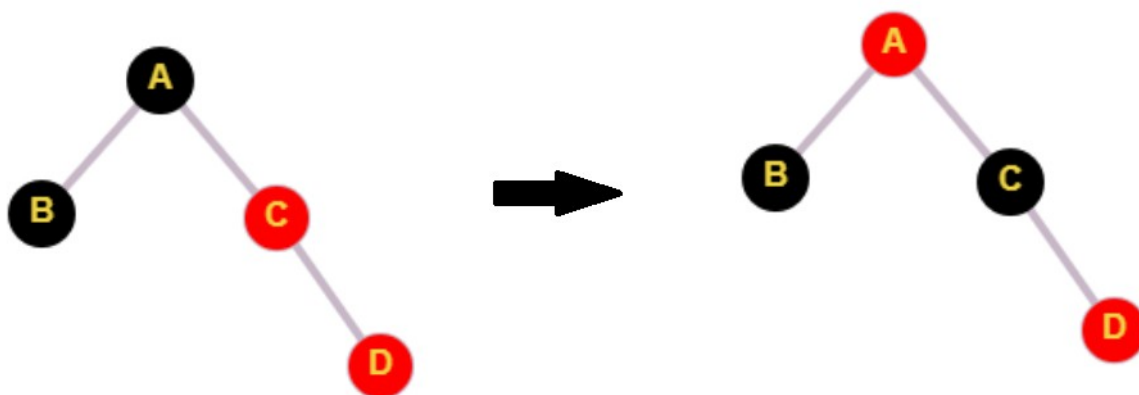


Рисунок 6: Перекрашивание

2. Выполнить переворот

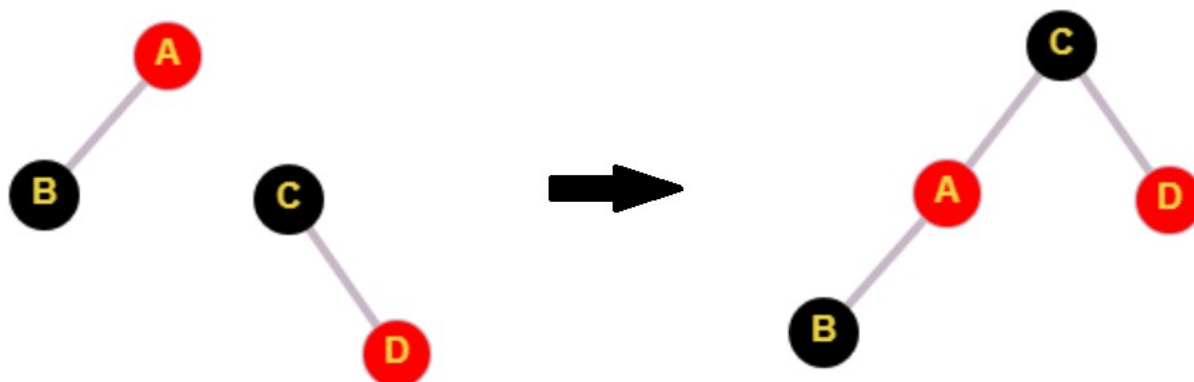


Рисунок 7: Переворот

## 2. 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.

### 2.1. 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
<i>private</i>	<i>node*</i>	<i>make_node(int value)</i>
<i>private</i>	<i>void</i>	<i>del_node(node*)</i>
<i>private</i>	<i>void</i>	<i>clear(node*)</i>
<i>private</i>	<i>node*</i>	<i>rotate_right(node*)</i>
<i>private</i>	<i>node*</i>	<i>rotate_left(node*)</i>
<i>private</i>	<i>void</i>	<i>balance_insert(node**)</i>
<i>private</i>	<i>bool</i>	<i>balance_remove_case1(node**)</i>
<i>private</i>	<i>bool</i>	<i>balance_remove_case2(node**)</i>

<i>private</i>	<i>bool</i>	<i>insert(int, node**)</i>
<i>private</i>	<i>bool</i>	<i>getmin(node**, node**)</i>
<i>private</i>	<i>bool</i>	<i>remove(node**, int)</i>
<i>private</i>	<i>bool</i>	<i>print_tree(node*&amp; ptr, int u)</i>
<i>public</i>	<i>void</i>	<i>clear()</i>
<i>public</i>	<i>void</i>	<i>insert(int)</i>
<i>public</i>	<i>void</i>	<i>remove(int);</i>
<i>public</i>	<i>void</i>	<i>print()</i>
<i>public</i>	<i>int</i>	<i>getsize()</i>
<i>public</i>		<i>RBtree()</i>
<i>public</i>		<i>~RBtree()</i>

### ***Метод Rbtree::RBtree.***

Конструктор. Ничего не принимает. Инициализирует корень дерева равным листом.

### ***Метод Rbtree::~~RBtree.***

Деструктор. Ничего не принимает. Освобождает память от дерева с помощью метода *clear(node\*)*.

### ***Метод Rbtree::make\_node.***

Принимает value - значение. Создает узел с этим значением и красит его в красный цвет. Возвращает созданный узел.

### ***Метод RBtree::del\_node.***

Принимает node\* - узел дерева и удаляет его.

### ***Метод Rbtree::clear.***

Является рекурсивным методом. Принимает на вход \*root — корень дерева, которое нужно удалить. Удаляет дерево целиком с помощью метода *del\_node*.

***Memod RBtree::rotate\_right.***

Принимает на вход `node*` - узел. Осуществляет поворот поддереву этого узла вправо. Возвращает родительский узел.

***Memod RBtree::rotate\_left.***

Принимает на вход `node*` - узел. Осуществляет поворот поддереву этого узла влево. Возвращает родительский узел.

***Memod RBtree::balance\_insert.***

Является рекурсивным методом. Принимает на вход `node**` - узел. Производит поиск ситуаций балансировки дерева после добавления нового узла.

***Memod Rbtree::balance\_remove\_case1.***

Является рекурсивным методом. Принимает на вход `node**` - узел. Производит поиск ситуаций балансировки дерева после удаления левого узла. Возвращает `true` если нужен баланс.

***Memod Rbtree::balance\_remove\_case2.***

Является рекурсивным методом. Принимает на вход `node**` - узел. Производит поиск ситуаций балансировки дерева после удаления правого узла. Возвращает `true` если нужен баланс.

***Memod RBtree::insert.***

Является рекурсивным методом. Принимает на вход `value` — значение и `node*` - узел. Находит место для вставки узла со значением `value`. При помощи метода `make_node` создает узел и с помощью метода `balance_insert` производит балансировку дерева. Возвращает `true` если нужен баланс.

### ***Memod Rbtree::getmin.***

Является рекурсивным методом. Принимает на вход ***\*\*root*** — корень дерева и ***\*res*** — узел который был удален. Удаляет узел с максимальным значением. Возвращает true если нужен баланс.

### ***Memod Rbtree::remove.***

Является рекурсивным методом. Принимает на вход ***\*root*** — корень дерева в котором нужно удалить узел со значением и ***value*** — значение, узел с которым нужно удалить. Вызывая себя находит узел, с помощью метода *del\_node* удаляет его, возвращает true если нужен баланс и с помощью методов *balance\_remove\_case1* или *balance\_remove\_case2* балансирует дерево.

### ***Memod Rbtree::print\_tree.***

Является рекурсивным методом. Принимает на вход ***\*\*ptr*** — корень дерева и ***u*** — глубина на котором сейчас обход. Выводит в консоль дерево.

### ***Memod Rbtree::clear.***

Ничего не принимает. Вспомогательный метод для запуска метода *clear(node\*)*.

### ***Memod Rbtree::remove.***

Принимает на вход ***value*** — значение, узел с которым нужно удалить. Вспомогательный метод для запуска метода *remove(node\*, int)*.

### ***Memod Rbtree::print.***

Ничего не принимает. Вспомогательный метод для запуска метода *print\_tree(node\*\*, int)*.

### ***Memod Rbtree::getsize.***

Ничего не принимает. Возвращает количество дерева.

## 2.2. 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

## 2.3. 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 2). 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.

### **3. ОПИСАНИЕ ИНТЕРФЕЙСА ПОЛЬЗОВАТЕЛЯ**

Пользователь выбирает действие (вводит цифру от 1 до 3). В зависимости от выбранного действия выполняется:

- Ввод из консоли
- Ввод из файла
- Генерирование случайного дерева

Пользователь смотря на дерево, вводит элементы которые нужно вставить в дерево.



## **ЗАКЛЮЧЕНИЕ**

В ходе работы над поставленным заданием был изучена такая структура данных, как БДП красно-черное дерево, а также был разработан класс, включающий в себя методы работы с БДП красно-черное дерево. Программа была успешно протестирована на работоспособность.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. [https://neerc.ifmo.ru/wiki/index.php?title=Красно-черное\\_дерево](https://neerc.ifmo.ru/wiki/index.php?title=Красно-черное_дерево)
2. <https://graphonline.ru>
3. [https://ru.wikipedia.org/wiki/Красно-чёрное\\_дерево](https://ru.wikipedia.org/wiki/Красно-чёрное_дерево)

## ПРИЛОЖЕНИЕ А

### ТЕСТИРОВАНИЕ

Таблица А.1 - Примеры тестовых случаев на некорректных данных

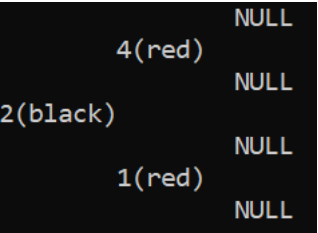
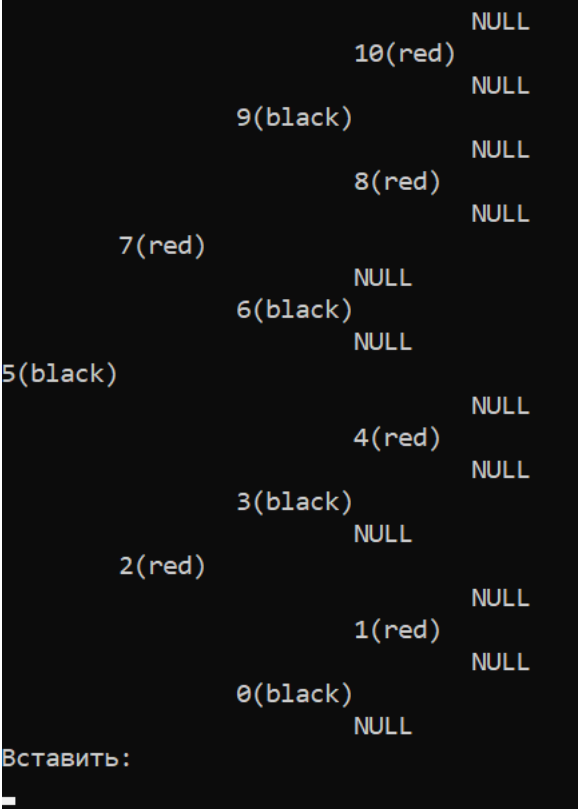
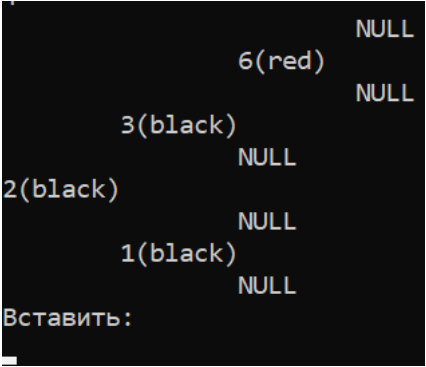
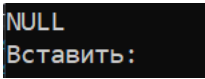
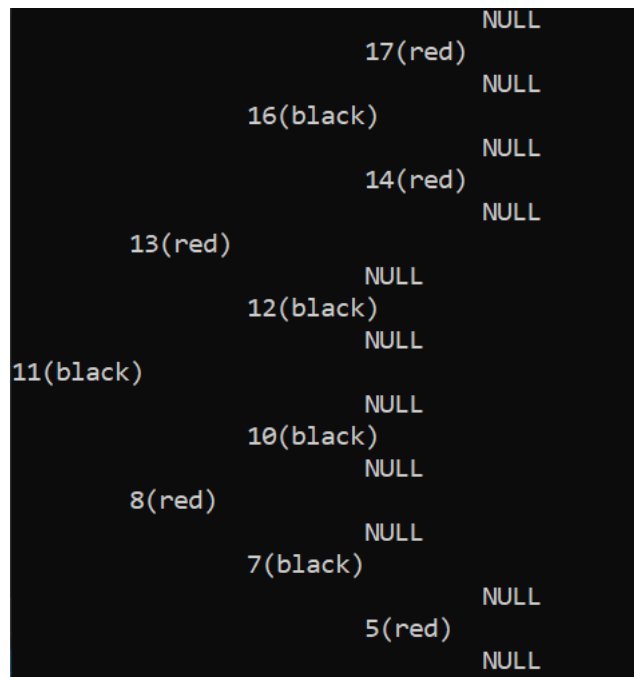
№ п/п	Входные данные	Выходные данные
1.	f	Нужно ввести число, попробуй еще
2.	9	Нужно ввести 1 или 2, чтобы выйти из программы нужно нажать esc
3.	2 2 4	 <pre> 4(red) NULL 2(black) NULL 1(red) NULL </pre>
4.	1 f	Нужно ввести число, попробуй еще
5.	3	 <pre> 10(red) NULL 9(black) NULL 8(red) NULL 7(red)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

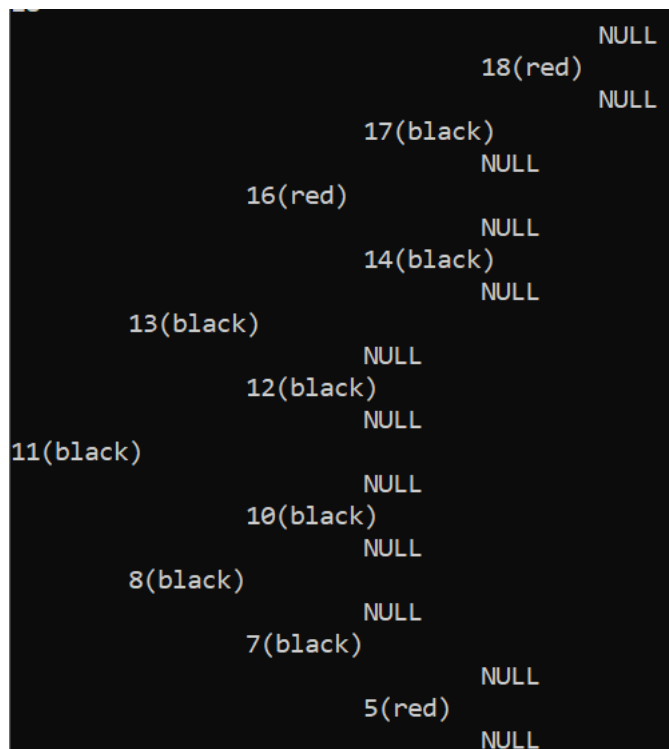
№ п/п	Входные данные	Выходные данные
6.	2 4 0 y	 <p>Выход из программы</p>
7.	8 esc	Выход из программы
8.	2 200	Построено дерево из 200 элементов, заполненное случайными числами от 0 до 400
9.	-1	Нужно ввести число, попробуй еще
10.	1 0	 <p>Пустое дерево</p>

Рассмотрим дерево

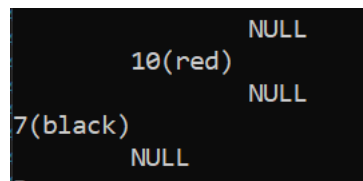


Если в него добавить узел с номером 18, то получится случай 1. В такой ситуации запустится метод балансировки, узел 14 окрасится в черный, узел 16 — в красный, узел 17 — в черный.

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;
        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**);
    bool balance_remove_case1(node**);
    bool balance_remove_case2(node**);
    bool insert(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);
    void insert(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->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;
        }
    }
}

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;
            right->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;
}

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::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::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->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 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;

    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)
    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));
    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;
    }
}
n = 1;

if (exit != 27)
{

    while (n)
    {
        tree.print();
        std::cout << "Вставить: \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 ввести 0?\n Y - выйти
N = ввести\n";

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

        tree.insert(n);

        if (exit == 'n')
            n = 1;
    }
}

```

```
        exit = 0;
    }
}

lin.close();

getchar();
return 0;
}
```