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

ОТЧЕТ

по лабораторной работе №5
по дисциплине «Алгоритмы и структуры данных»
Тема: Бинарное дерево поиска

Студент(ка) гр. 9382 _____ Русинов Д.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2020

Цель работы.

Изучить бинарные деревья поиска.

Задание.

БДП: красно-чёрное дерево; действие: 1+2а. Выбранное из 2а задание – поиск всех элементов с заданным значением и вывод количества на экран.

Основные теоретические положения.

Двоичное дерево поиска (англ. binary search tree, BST) — это двоичное дерево, для которого выполняются следующие дополнительные условия (свойства дерева поиска):

Оба поддерева — левое и правое — являются двоичными деревьями поиска.

У всех узлов левого поддерева произвольного узла X значения ключей данных меньше либо равны, нежели значение ключа данных самого узла X.

У всех узлов правого поддерева произвольного узла X значения ключей данных больше либо равны, нежели значение ключа данных самого узла X.

Функции и структуры данных.

template<typename Elem>

struct Node;

Поля:

1) Elem data – значение узла
2) Node* left – указатель на левую ветку
3) Node* right – указатель на правую ветку
4) Node* parent – указатель на отца
5) bool color – цвет узла

template<typename Elem>

class RBTree;

Поля:

1) Node<Elem>* root – корень дерева

Методы:

1) Node<Elem>* grandparent(Node<Elem>* n) – возвращает дедушку переданного узла n
2) Node<Elem>* uncle(Node<Elem>* n) – возвращает дядю переданного узла n
3) void restoreRoot(Node<Elem>* n) – восстанавливает корень дерева на основе переданного узла
4) void leftRotate(Node<Elem>* n) – поворот поддерева влево
5) void rightRotate(Node<Elem>* n) – поворот поддерева вправо
6) int find(Elem const& data, Node<Elem>* node, int count = 0) – поиск кол-ва заданного элементов data в дереве node, count – глубина рекурсии
7) void insertCase1(Node<Elem>* n) – первый случай для исправления дерева после вставки элемента. Передается элемент, где возможна проблема
8) void insertCase2(Node<Elem>* n) – второй случай для исправления дерева после вставки элемента. Передается элемент, где возможна проблема
9) void insertCase3(Node<Elem>* n) – третий случай для исправления дерева после вставки элемента. Передается элемент, где возможна проблема
10) void insertCase4(Node<Elem>* n) – четвертый случай для исправления дерева после вставки элемента. Передается элемент, где возможна проблема
11) void insertCase5(Node<Elem>* n) – пятый случай для исправления дерева после вставки элемента. Передается элемент, где возможна проблема
12) RBTree() – конструктор
13) ~RBTree() – деструктор
14) Node<Elem>* getRoot() – метод для получения корня

15) int count(Elem const& data) – публичный метод для поиска количества заданного элемента в дереве
16) void insert(Elem stuff) – вставка элемента в дерево

Функции:

1) std::ostream& operator<<(std::ostream& out, Node<Elem> node) – определен оператор вывода для Node
2) std::string generateFormatPrint(int count) – функция, которая генерирует форматную строку для отображения глубины рекурсии, count – глубина рекурсии
3) void printTree(Node<int>* tree, int level) – функция для печати дерева в формате ПКЛ
4) RBTree<int>* insert(int file = 0) - функция для построения пользователем дерева. Аргумент для ввода из файла/в режиме реального времени.
5) char executeTask(RBTree<int>* tree) – функция для выполнения задания над переданным деревом. Возвращает символ, который говорит о том, нужно ли продолжать выполнять задание или нет.

Описание алгоритма.

Свойства красно-черного дерева:

1) Узел может быть либо красным, либо черным и имеет двух потомков.
2) Корень – как правило черный.
3) Все листья, не содержащие данных – черные.
4) Оба потомка красного узла – черные.
5) Любой простой путь от узла-предка до листового узла-потомка содержит одинаковое число черных узлов.

- Выполнение поиска всех элементов в дереве с определенным значением

  Если узел пустой, то результат ноль

Если искомое значение меньше значения текущего узла, то нужно проверить, сколько узлов с искомым значением в левой ветке текущего узла. Результат = (результат левой ветки)

Если искомое значение больше значения текущего узла, то нужно проверить, сколько узлов с искомым значением в правой ветке текущего узла. Результат = (результат правой ветки)

Если искомое значение равно значению текущего узла, то нужно проверить, сколько узлов с искомым значением в правой и левой ветках текущего узла. Результат = 1 + (результат левой ветки) + (результат правой ветки)

Данная операция выполняется рекурсивно.

- Выполнение вставки в дерево

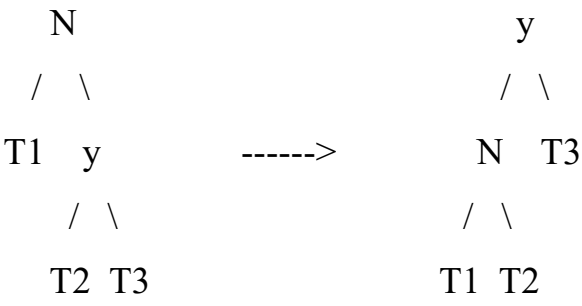
  Новый узел в красно-черное дерево добавляется на место одного из листьев и окрашивается в красный цвет

  Если значение вставляемого элемента меньше значения текущей ветки, то следующей веткой будет левая.

  Если значение вставляемого элемента больше либо равно значению текущей ветки, то следующей веткой будет правая

  Если следующая ветка пустая, то туда вставляется заданный элемент

- Выполнение левого поворота дерева N



  В правую ветку N подставляется левая ветка y, в левую ветку y подставляется N

- Выполнение правого поворота дерева N

```
   y     T3      ------->      T1   N
  /  \                              /  \
 T1   T2                          T2   T3
```

В левую ветку N подставляется правая ветка y, в правую ветку y подставляется N.

- Балансировка дерева после вставки

  А) Свойство 3 (все листья черные) – выполняется всегда

  Б) Свойство 4 (оба потомка любого красного узла - черные) может нарушиться только при добавлении красного узла, при перекрашивании черного узла в красный или при повороте

  В) Свойство 5 (Все пути от любого узла до листовых узлов содержат одинаковое число чёрных узлов) может нарушиться только при добавлении чёрного узла, перекрашивании красного узла в чёрный (или наоборот), или при повороте.

  Буквой N будем обозначать текущий узел (окрашенный красным). Сначала это новый узел, который вставляется, но эта процедура может применяться рекурсивно к другим узлам. P будем обозначать предка N, через G обозначим дедушку N, а U будем обозначать дядю (узел, имеющий общего родителя с узлом P). В некоторых случаях роли узлов могут меняться, но, в любом случае, каждое обозначение будет представлять тот же узел, что и в начале.

  1) Случай 1

     Текущий узел N в корне дерева. В этом случае, он перекрашивается в чёрный цвет, чтобы оставить верным Свойство 2 (Корень — чёрный). Так как это действие добавляет один чёрный узел в каждый путь, Свойство 5 (Все пути от любого данного узла до листовых узлов содержат одинаковое число чёрных узлов) не нарушается.

  2) Случай 2

Предок P текущего узла чёрный, то есть Свойство 4 (Оба потомка каждого красного узла — чёрные) не нарушается. В этом случае дерево остаётся корректным. Свойство 5 (Все пути от любого данного узла до листовых узлов содержат одинаковое число чёрных узлов) не нарушается, потому что текущий узел N имеет двух чёрных листовых потомков, но так как N является красным, путь до каждого из этих потомков содержит такое же число чёрных узлов, что и путь до чёрного листа, который был заменен текущим узлом, так что свойство остается верным.

3) Случай 3

   Если и родитель P, и дядя U — красные, то они оба могут быть перекрашены в чёрный, и дедушка G станет красным (для сохранения свойства 5 (Все пути от любого данного узла до листовых узлов содержат одинаковое число чёрных узлов)). Теперь у текущего красного узла N чёрный родитель. Так как любой путь через родителя или дядю должен проходить через дедушку, число чёрных узлов в этих путях не изменится. Однако, дедушка G теперь может нарушить свойства 2 (Корень — чёрный) или 4 (Оба потомка каждого красного узла — чёрные) (свойство 4 может быть нарушено, так как родитель G может быть красным). Чтобы это исправить, вся процедура рекурсивно выполняется на G из случая 1.

4) Случай 4

   Родитель P является красным, но дядя U — чёрный. Также, текущий узел N — правый потомок P, а P в свою очередь — левый потомок своего предка G. В этом случае может быть произведен поворот дерева, который меняет роли текущего узла N и его предка P. Тогда, для бывшего родительского

узла P в обновленной структуре используем случай 5, потому что Свойство 4 (Оба потомка любого красного узла — чёрные) все ещё нарушено. Вращение приводит к тому, что некоторые пути (в поддереве, обозначенном «1» на схеме) проходят через узел N, чего не было до этого. Это также приводит к тому, что некоторые пути (в поддереве, обозначенном «3») не проходят через узел P. Однако, оба эти узла являются красными, так что Свойство 5 (Все пути от любого данного узла до листовых узлов содержат одинаковое число чёрных узлов) не нарушается при вращении. Однако Свойство 4 всё ещё нарушается, но теперь задача сводится к Случаю 5.

5) Случай 5

   Родитель P является красным, но дядя U — чёрный, текущий узел N — левый потомок P и P — левый потомок G. В этом случае выполняется поворот дерева на G. В результате получается дерево, в котором бывший родитель P теперь является родителем и текущего узла N и бывшего дедушки G. Известно, что G — чёрный, так как его бывший потомок P не мог бы в противном случае быть красным (без нарушения Свойства 4). Тогда цвета P и G меняются и в результате дерево удовлетворяет Свойству 4 (Оба потомка любого красного узла — чёрные). Свойство 5 (Все пути от любого данного узла до листовых узлов содержат одинаковое число чёрных узлов) также остается верным, так как все пути, которые проходят через любой из этих трех узлов, ранее проходили через G, поэтому теперь они все проходят через P. В каждом случае, из этих трёх узлов только один окрашен в чёрный.

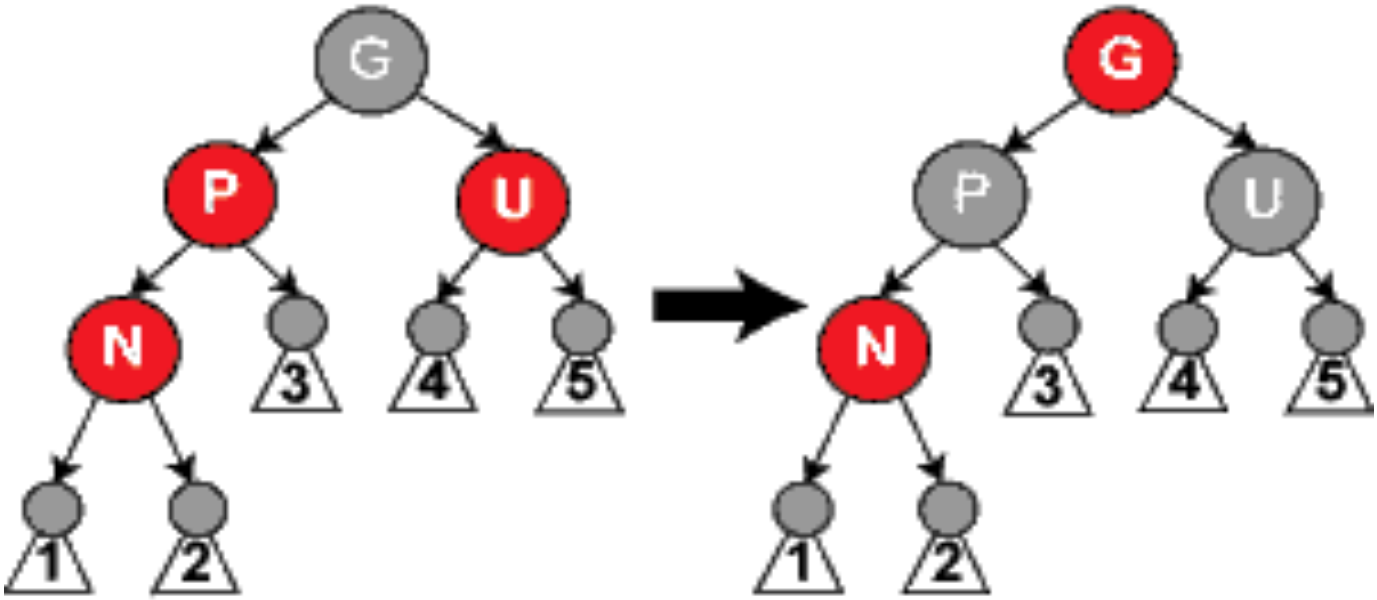


Рисунок 1 Иллюстрация случая 3

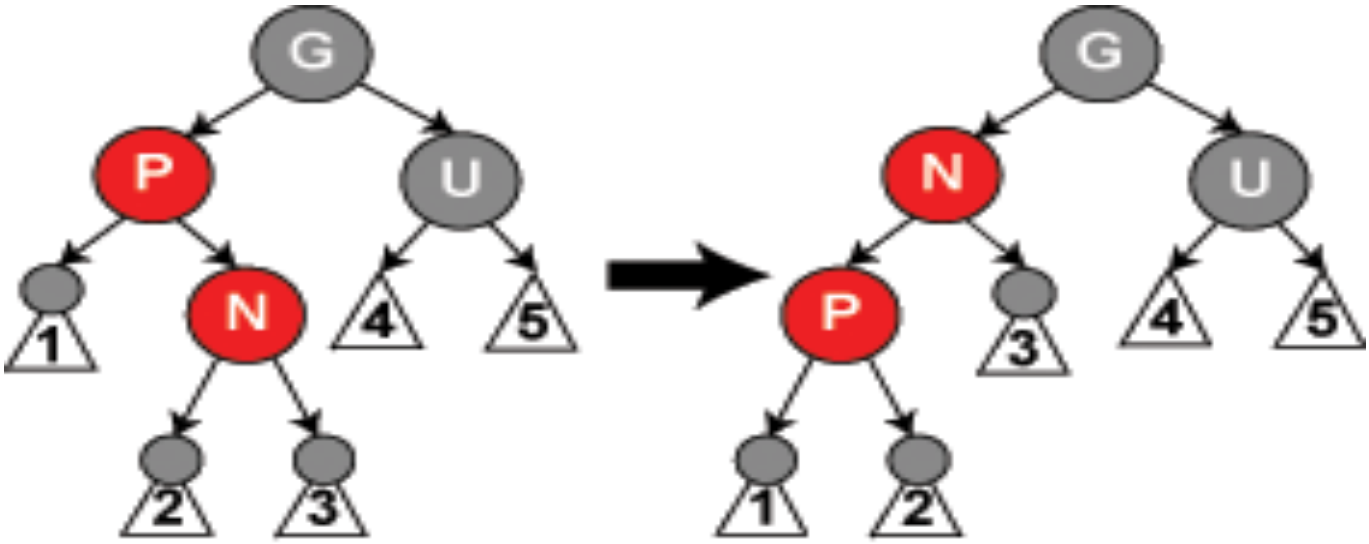


Рисунок 2 Иллюстрация случая 4

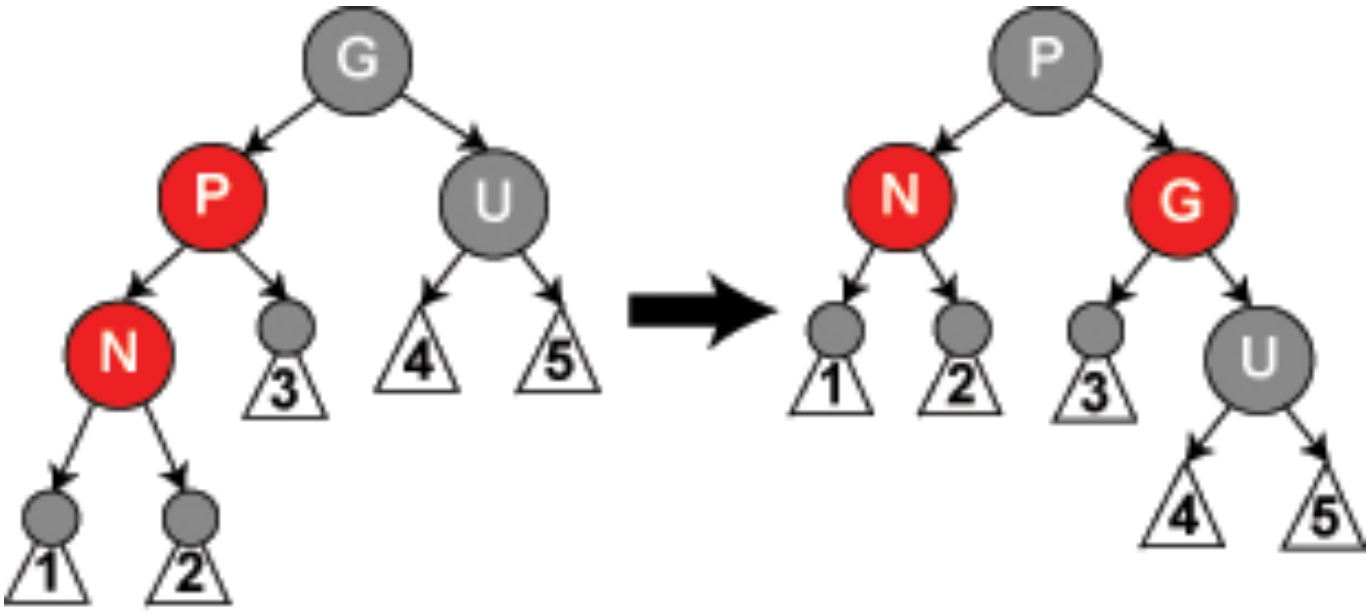


Рисунок 3 Иллюстрация случая 5

После вставки в дерево выполняется балансировка дерева.

Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	5 5 3 9 2 6 5	Если хотите строить дерево в live-режиме, введите 0 Иначе любое другое число Ввод: 0 ---Построение структуры данных начато--- Введите кол-во элементов: 5 [1] Введите число: 5 ---Вставка элемента--- Элемент (5, 0, #, #, #) вставлен -----Балансировка-----	Ввод в live

		[CASE 1] (5, 1, #, #, #) - не имеет отца, поэтому стал корнем дерева [2] Введите число: 3 ---Вставка элемента--- (3, 0, #, #, #) < (5, 1, #, #, #), переход к левой ветке Левая ветка пустая. Элемент (3, 0, #, #, *) вставлен -----Балансировка----- [CASE 1] (3, 0, #, #, *) - имеет отца, переход к следующему случаю [CASE 2] Отец (5, 1, *, #, #) является черным, свойство 3 не нарушено [3] Введите число: 9 ---Вставка элемента--- (9, 0, #, #, #) >= (5, 1, *, #, #), переход к правой ветке Правая ветка пустая. Элемент (9, 0, #, #, *) вставлен -----Балансировка----- [CASE 1] (9, 0, #, #, *) - имеет отца, переход к следующему случаю [CASE 2] Отец (5, 1, *, *, #) является черным, свойство 3 не нарушено [4] Введите число: 2 ---Вставка элемента---	

		(2, 0, #, #, #) < (5, 1, *, *, #), переход к левой ветке (2, 0, #, #, #) < (3, 0, #, #, *), переход к левой ветке Левая ветка пустая. Элемент (2, 0, #, #, *) вставлен -----Балансировка----- [CASE 1] (2, 0, #, #, *) - имеет отца, переход к следующему случаю [CASE 2] Отец (3, 0, *, #, *) является красным, свойство 3 нарушено [CASE 3] У (2, 0, #, #, *) есть дядя - (9, 0, #, #, *) и он красный [CASE 3] Отец стал черным, дядя стал черным, дедушка стал красным. Повтор балансировки для дедушки [CASE 1] (5, 1, *, *, #) - не имеет отца, поэтому стал корнем дерева [5] Введите число: 6 ---Вставка элемента--- (6, 0, #, #, #) >= (5, 1, *, *, #), переход к правой ветке (6, 0, #, #, #) < (9, 1, #, #, *), переход к левой ветке Левая ветка пустая. Элемент (6, 0, #, #, *) вставлен -----Балансировка-----	

		[CASE 1] (6, 0, #, #, *) - имеет отца, переход к следующему случаю [CASE 2] Отец (9, 1, *, #, *) является черным, свойство 3 не нарушено ---Построение структуры данных завершено--- Введите элемент, который хотите найти в дереве: 5 ---Подсчет количества элементов со значением '5' начат--- 5 == (5, 1, *, *, #) Вызов поиска для правой ветви .5 < (9, 1, *, #, *). Вызов поиска для левой ветви ..5 < (6, 0, #, #, *). Вызов поиска для левой ветви ...Ветвь пустая, возвращаю 0 ... Получено значение от левой ветви - 0 .. Получено значение от левой ветви - 0 Получено значение от правой ветви - 0 Вызов поиска для левой летви .5 > (3, 1, *, #, *). Вызов поиска для правой ветви ..Ветвь пустая, возвращаю 0 .. Получено значение от правой ветви - 0	

		Получено значение от левой ветви - 0
		Результат подсчета: 1
		---Подсчет количества элементов окончен---
		
		Если хотите продолжить, введите '+': -
		---Представление дерева---
		0 - КРАСНЫЙ
		1 - ЧЕРНЫЙ
		  1
		  0
		1
		  1
		  0	

2.	10 5 3 9 2 6 0 -1 3 4 1 3	Если хотите строить дерево в live-режиме, введите 0 Иначе любое другое число Ввод: 0 ---Построение структуры данных начато--- Введите кол-во элементов: 10 [1] Введите число: 5 ---Вставка элемента--- Элемент (5, 0, #, #, #) вставлен -----Балансировка----- [CASE 1] (5, 1, #, #, #) - не имеет отца, поэтому стал корнем дерева	Live ввод

		[2] Введите число: 3 ---Вставка элемента--- (3, 0, #, #, #) < (5, 1, #, #, #), переход к левой ветке Левая ветка пустая. Элемент (3, 0, #, #, *) вставлен -----Балансировка----- [CASE 1] (3, 0, #, #, *) - имеет отца, переход к следующему случаю [CASE 2] Отец (5, 1, *, #, #) является черным, свойство 3 не нарушено [3] Введите число: 9 ---Вставка элемента--- (9, 0, #, #, #) >= (5, 1, *, #, #), переход к правой ветке Правая ветка пустая. Элемент (9, 0, #, #, *) вставлен -----Балансировка----- [CASE 1] (9, 0, #, #, *) - имеет отца, переход к следующему случаю [CASE 2] Отец (5, 1, *, *, #) является черным, свойство 3 не нарушено [4] Введите число: 2 ---Вставка элемента--- (2, 0, #, #, #) < (5, 1, *, *, #), переход к левой ветке	

		(2, 0, #, #, #) < (3, 0, #, #, *), переход к левой ветке Левая ветка пустая. Элемент (2, 0, #, #, *) вставлен -----Балансировка----- [CASE 1] (2, 0, #, #, *) - имеет отца, переход к следующему случаю [CASE 2] Отец (3, 0, *, #, *) является красным, свойство 3 нарушено [CASE 3] У (2, 0, #, #, *) есть дядя - (9, 0, #, #, *) и он красный [CASE 3] Отец стал черным, дядя стал черным, дедушка стал красным. Повтор балансировки для дедушки [CASE 1] (5, 1, *, *, #) - не имеет отца, поэтому стал корнем дерева [5] Введите число: 6 ---Вставка элемента--- (6, 0, #, #, #) >= (5, 1, *, *, #), переход к правой ветке (6, 0, #, #, #) < (9, 1, #, #, *), переход к левой ветке Левая ветка пустая. Элемент (6, 0, #, #, *) вставлен -----Балансировка----- [CASE 1] (6, 0, #, #, *) - имеет отца, переход к следующему случаю	

		[CASE 2] Отец (9, 1, *, #, *) является черным, свойство 3 не нарушено [6] Введите число: 0 ---Вставка элемента--- (0, 0, #, #, #) < (5, 1, *, *, #), переход к левой ветке (0, 0, #, #, #) < (3, 1, *, #, *), переход к левой ветке (0, 0, #, #, #) < (2, 0, #, #, *), переход к левой ветке Левая ветка пустая. Элемент (0, 0, #, #, *) вставлен -----Балансировка----- [CASE 1] (0, 0, #, #, *) - имеет отца, переход к следующему случаю [CASE 2] Отец (2, 0, *, #, *) является красным, свойство 3 нарушено [CASE 3] У (0, 0, #, #, *) нет дяди или дядя не красный [CASE 4] Дедушка (0, 0, #, #, *) - (3, 1, *, #, *) [CASE 4] Ни один из вариантов для случая 4 не выполнен для (0, 0, #, #, *) [CASE 4] Переход к случаю 5 [CASE 5] Отец стал черным, дедушка стал красным	

		[CASE 5] (0, 0, #, #, *) является левым сыном отца (2, 1, *, #, *), и отец является левым сыном [CASE 5] Правый поворот дедушки [7] Введите число: -1 ---Вставка элемента--- (-1, 0, #, #, #) < (5, 1, *, *, #), переход к левой ветке (-1, 0, #, #, #) < (2, 1, *, *, *), переход к левой ветке (-1, 0, #, #, #) < (0, 0, #, #, *), переход к левой ветке Левая ветка пустая. Элемент (-1, 0, #, #, *) вставлен -----Балансировка----- [CASE 1] (-1, 0, #, #, *) - имеет отца, переход к следующему случаю [CASE 2] Отец (0, 0, *, #, *) является красным, свойство 3 нарушено [CASE 3] У (-1, 0, #, #, *) есть дядя - (3, 0, #, #, *) и он красный [CASE 3] Отец стал черным, дядя стал черным, дедушка стал красным. Повтор балансировки для дедушки [CASE 1] (2, 0, *, *, *) - имеет отца, переход к следующему случаю [CASE 2] Отец (5, 1, *, *, #) является черным, свойство 3 не нарушено	

		[8] Введите число: 3 ---Вставка элемента--- (3, 0, #, #, #) < (5, 1, *, *, #), переход к левой ветке (3, 0, #, #, #) >= (2, 0, *, *, *), переход к правой ветке (3, 0, #, #, #) >= (3, 1, #, #, *), переход к правой ветке Правая ветка пустая. Элемент (3, 0, #, #, *) вставлен -----Балансировка----- [CASE 1] (3, 0, #, #, *) - имеет отца, переход к следующему случаю [CASE 2] Отец (3, 1, #, *, *) является черным, свойство 3 не нарушено [9] Введите число: 4 ---Вставка элемента--- (4, 0, #, #, #) < (5, 1, *, *, #), переход к левой ветке (4, 0, #, #, #) >= (2, 0, *, *, *), переход к правой ветке (4, 0, #, #, #) >= (3, 1, #, *, *), переход к правой ветке (4, 0, #, #, #) >= (3, 0, #, #, *), переход к правой ветке Правая ветка пустая. Элемент (4, 0, #, #, *) вставлен	

		-----Балансировка----- [CASE 1] (4, 0, #, #, *) - имеет отца, переход к следующему случаю [CASE 2] Отец (3, 0, #, *, *) является красным, свойство 3 нарушено [CASE 3] У (4, 0, #, #, *) нет дяди или дядя не красный [CASE 4] Дедушка (4, 0, #, #, *) - (3, 1, #, *, *) [CASE 4] Ни один из вариантов для случая 4 не выполнен для (4, 0, #, #, *) [CASE 4] Переход к случаю 5 [CASE 5] Отец стал черным, дедушка стал красным [CASE 5] (4, 0, #, #, *) является правым сыном отца (3, 1, #, *, *), и отец является правым сыном [CASE 5] Левый поворот дедушки [10] Введите число: 1 ---Вставка элемента--- (1, 0, #, #, #) < (5, 1, *, *, #), переход к левой ветке (1, 0, #, #, #) < (2, 0, *, *, *), переход к левой ветке (1, 0, #, #, #) >= (0, 1, *, #, *), переход к правой ветке Правая ветка пустая. Элемент (1, 0, #, #, *) вставлен	

		-----Балансировка----- [CASE 1] (1, 0, #, #, *) - имеет отца, переход к следующему случаю [CASE 2] Отец (0, 1, *, *, *) является черным, свойство 3 не нарушено ---Построение структуры данных завершено--- Введите элемент, который хотите найти в дереве: 3 ---Подсчет количества элементов со значением '3' начат--- 3 < (5, 1, *, *, #). Вызов поиска для левой ветви .3 > (2, 0, *, *, *). Вызов поиска для правой ветви ..3 == (3, 1, *, *, *) ..Вызов поиска для правой ветви ...3 < (4, 0, #, #, *). Вызов поиска для левой ветви ....Ветвь пустая, возвращаю 0 .... Получено значение от левой ветви - 0 ..Получено значение от правой ветви - 0 ..Вызов поиска для левой летви ...3 == (3, 0, #, #, *) ...Вызов поиска для правой ветви	

		....Ветвь пустая, возвращаю 0 ...Получено значение от правой ветви - 0 ...Вызов поиска для левой летви ....Ветвь пустая, возвращаю 0 ...Получено значение от левой ветви - 0 ..Получено значение от левой ветви - 1 .. Получено значение от правой ветви - 2 . Получено значение от левой ветви - 2 Результат подсчета: 2 ---Подсчет количества элементов окончен--- Если хотите продолжить, введите '+': - ---Представление дерева--- 0 - КРАСНЫЙ 1 - ЧЕРНЫЙ 1 0 1 0 1 0 0 0 1 0	

3.	0	Если хотите строить дерево в live-режиме, введите 0 Иначе любое другое число Ввод: 0 ---Построение структуры данных начато--- Введите кол-во элементов: 0 ---Построение структуры данных завершено--- Введите элемент, который хотите найти в дереве: 0 ---Подсчет количества элементов со значением '0' начат--- Ветвь пустая, возвращаю 0 Результат подсчета: 0 ---Подсчет количества элементов окончен--- Если хотите продолжить, введите '+': - ---Представление дерева--- 0 - КРАСНЫЙ 1 - ЧЕРНЫЙ	
4.	-1	Если хотите строить дерево в live-режиме, введите 0 Иначе любое другое число Ввод: 0	Проверка на некорректных данных

		---Построение структуры данных начато--- Введите кол-во элементов: -1 Кол-во элементов должно быть >= 0	
5.	5 1 1 1 1 1 1	Если хотите строить дерево в live-режиме, введите 0 Иначе любое другое число Ввод: 1 ---Построение структуры данных начато--- ---Вставка элемента--- Элемент (1, 0, #, #, #) вставлен -----Балансировка----- [CASE 1] (1, 1, #, #, #) - не имеет отца, поэтому стал корнем дерева ---Вставка элемента--- (1, 0, #, #, #) >= (1, 1, #, #, #), переход к правой ветке Правая ветка пустая. Элемент (1, 0, #, #, *) вставлен -----Балансировка----- [CASE 1] (1, 0, #, #, *) - имеет отца, переход к следующему случаю [CASE 2] Отец (1, 1, #, *, #) является черным, свойство 3 не нарушено	Ввод из файла

		---Вставка элемента--- (1, 0, #, #, #) >= (1, 1, #, *, #), переход к правой ветке (1, 0, #, #, #) >= (1, 0, #, #, *), переход к правой ветке Правая ветка пустая. Элемент (1, 0, #, #, *) вставлен -----Балансировка----- [CASE 1] (1, 0, #, #, *) - имеет отца, переход к следующему случаю [CASE 2] Отец (1, 0, #, *, *) является красным, свойство 3 нарушено [CASE 3] У (1, 0, #, #, *) нет дяди или дядя не красный [CASE 4] Дедушка (1, 0, #, #, *) - (1, 1, #, *, #) [CASE 4] Ни один из вариантов для случая 4 не выполнен для (1, 0, #, #, *) [CASE 4] Переход к случаю 5 [CASE 5] Отец стал черным, дедушка стал красным [CASE 5] (1, 0, #, #, *) является правым сыном отца (1, 1, #, *, *), и отец является правым сыном [CASE 5] Левый поворот дедушки ---Вставка элемента--- (1, 0, #, #, #) >= (1, 1, *, *, #), переход к правой ветке	

		(1, 0, #, #, #) >= (1, 0, #, #, *), переход к правой ветке Правая ветка пустая. Элемент (1, 0, #, #, *) вставлен -----Балансировка----- [CASE 1] (1, 0, #, #, *) - имеет отца, переход к следующему случаю [CASE 2] Отец (1, 0, #, *, *) является красным, свойство 3 нарушено [CASE 3] У (1, 0, #, #, *) есть дядя - (1, 0, #, #, *) и он красный [CASE 3] Отец стал черным, дядя стал черным, дедушка стал красным. Повтор балансировки для дедушки [CASE 1] (1, 1, *, *, #) - не имеет отца, поэтому стал корнем дерева ---Вставка элемента--- (1, 0, #, #, #) >= (1, 1, *, *, #), переход к правой ветке (1, 0, #, #, #) >= (1, 1, #, *, *), переход к правой ветке (1, 0, #, #, #) >= (1, 0, #, #, *), переход к правой ветке Правая ветка пустая. Элемент (1, 0, #, #, *) вставлен -----Балансировка----- [CASE 1] (1, 0, #, #, *) - имеет отца, переход к следующему случаю	

		[CASE 2] Отец (1, 0, #, *, *) является красным, свойство 3 нарушено [CASE 3] У (1, 0, #, #, *) нет дяди или дядя не красный [CASE 4] Дедушка (1, 0, #, #, *) - (1, 1, #, *, *) [CASE 4] Ни один из вариантов для случая 4 не выполнен для (1, 0, #, #, *) [CASE 4] Переход к случаю 5 [CASE 5] Отец стал черным, дедушка стал красным [CASE 5] (1, 0, #, #, *) является правым сыном отца (1, 1, #, *, *), и отец является правым сыном [CASE 5] Левый поворот дедушки ---Построение структуры данных завершено--- Введите элемент, который хотите найти в дереве: 1 ---Подсчет количества элементов со значением '1' начат--- 1 == (1, 1, *, *, #) Вызов поиска для правой ветви .1 == (1, 1, *, *, *) .Вызов поиска для правой ветви ..1 == (1, 0, #, #, *) ..Вызов поиска для правой ветви	

		...Ветвь пустая, возвращаю 0 ..Получено значение от правой ветви - 0 ..Вызов поиска для левой летви ...Ветвь пустая, возвращаю 0 ..Получено значение от левой ветви - 0 .Получено значение от правой ветви - 1 .Вызов поиска для левой летви ..1 == (1, 0, #, #, *) ..Вызов поиска для правой ветви ...Ветвь пустая, возвращаю 0 ..Получено значение от правой ветви - 0 ..Вызов поиска для левой летви ...Ветвь пустая, возвращаю 0 ..Получено значение от левой ветви - 0 .Получено значение от левой ветви - 1 Получено значение от правой ветви - 3 Вызов поиска для левой летви .1 == (1, 1, #, #, *) .Вызов поиска для правой ветви ..Ветвь пустая, возвращаю 0 .Получено значение от правой ветви - 0 .Вызов поиска для левой летви ..Ветвь пустая, возвращаю 0 .Получено значение от левой ветви - 0 Получено значение от левой ветви - 1	

		Результат подсчета: 5 ---Подсчет количества элементов окончен--- Если хотите продолжить, введите '+': - ---Представление дерева--- 0 - КРАСНЫЙ 1 - ЧЕРНЫЙ 0 1 0 1 1	
6.	5 1 -1 1 -1 1 -1	Если хотите строить дерево в live-режиме, введите 0 Иначе любое другое число Ввод: 1 ---Построение структуры данных начато--- ---Вставка элемента--- Элемент (1, 0, #, #, #) вставлен -----Балансировка----- [CASE 1] (1, 1, #, #, #) - не имеет отца, поэтому стал корнем дерева ---Вставка элемента--- (-1, 0, #, #, #) < (1, 1, #, #, #), переход к левой ветке	Ввод из файла

		Левая ветка пустая. Элемент (-1, 0, #, #, *) вставлен -----Балансировка----- [CASE 1] (-1, 0, #, #, *) - имеет отца, переход к следующему случаю [CASE 2] Отец (1, 1, *, #, #) является черным, свойство 3 не нарушено ---Вставка элемента--- (1, 0, #, #, #) >= (1, 1, *, #, #), переход к правой ветке Правая ветка пустая. Элемент (1, 0, #, #, *) вставлен -----Балансировка----- [CASE 1] (1, 0, #, #, *) - имеет отца, переход к следующему случаю [CASE 2] Отец (1, 1, *, *, #) является черным, свойство 3 не нарушено ---Вставка элемента--- (-1, 0, #, #, #) < (1, 1, *, *, #), переход к левой ветке (-1, 0, #, #, #) >= (-1, 0, #, #, *), переход к правой ветке Правая ветка пустая. Элемент (-1, 0, #, #, *) вставлен -----Балансировка----- [CASE 1] (-1, 0, #, #, *) - имеет отца, переход к следующему случаю	

		[CASE 2] Отец (-1, 0, #, *, *) является красным, свойство 3 нарушено [CASE 3] У (-1, 0, #, #, *) есть дядя - (1, 0, #, #, *) и он красный [CASE 3] Отец стал черным, дядя стал черным, дедушка стал красным. Повтор балансировки для дедушки [CASE 1] (1, 1, *, *, #) - не имеет отца, поэтому стал корнем дерева ---Вставка элемента--- (1, 0, #, #, #) >= (1, 1, *, *, #), переход к правой ветке (1, 0, #, #, #) >= (1, 1, #, #, *), переход к правой ветке Правая ветка пустая. Элемент (1, 0, #, #, *) вставлен -----Балансировка----- [CASE 1] (1, 0, #, #, *) - имеет отца, переход к следующему случаю [CASE 2] Отец (1, 1, #, *, *) является черным, свойство 3 не нарушено ---Построение структуры данных завершено--- Введите элемент, который хотите найти в дереве: -1	

		---Подсчет количества элементов со значением '-1' начат--- -1 < (1, 1, *, *, #). Вызов поиска для левой ветви .-1 == (-1, 1, #, *, *) .Вызов поиска для правой ветви ..-1 == (-1, 0, #, #, *) ..Вызов поиска для правой ветви ...Ветвь пустая, возвращаю 0 ..Получено значение от правой ветви - 0 ..Вызов поиска для левой летви ...Ветвь пустая, возвращаю 0 ..Получено значение от левой ветви - 0 .Получено значение от правой ветви - 1 .Вызов поиска для левой летви ..Ветвь пустая, возвращаю 0 .Получено значение от левой ветви - 0 . Получено значение от левой ветви - 2 Результат подсчета: 2 ---Подсчет количества элементов окончен--- Если хотите продолжить, введите '+': - ---Представление дерева--- 0 - КРАСНЫЙ 1 - ЧЕРНЫЙ 0	

		1 1 0 1	

Выводы.

Были изучены деревья бинарного поиска и реализовано красно-черное дерево

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
#include <fstream>
#include "iostream"

#define BLACK true
#define RED false

template<typename Elem>
struct Node {
    Elem data{};
    Node* left = nullptr;
    Node* right = nullptr;
    Node* parent = nullptr;
    bool color{};
};

template<typename Elem>
std::ostream& operator<<(std::ostream& out, Node<Elem> node) {
    std::cout << "(" << node.data << ", " << node.color << ", ";
    if (!node.left) std::cout << "#, ";
    else std::cout << "*, ";

    if (!node.right) std::cout << "#, ";
    else std::cout << "*, ";

    if (!node.parent) std::cout << "#)";
    else std::cout << "*)";
    return out;
}

std::string generateFormatPrint(int count) {
    auto string = std::string();
    for (int i = 0; i < count; ++i) string += ".";
    return string;
}

template<typename Elem>
class RBTree {
    Node<Elem>* root;

    Node<Elem>* grandparent(Node<Elem>* n) { // дедушка
        if ((n != nullptr) && (n->parent != nullptr)) return n->parent-
>parent;
        return nullptr;
    }

    Node<Elem>* uncle(Node<Elem>* n) { // дядя
        Node<Elem> *g = grandparent(n);
        if (g == nullptr) return nullptr;
        if (n->parent == g->left) return g->right;
        return g->left;
    }
```

```
    void restoreRoot(Node<Elem>* n) { // восстановление корня
        while (n->parent) n = n->parent;
        root = n;
    }

    /*
     * Функция левого поворота
     *        n                    y
     *       / \                  / \
     *      T1  y     --->       n  T3
     *         / \               / \
     *        T2 T3            T1 T2
     *
     * В правую ветку N подставляется левая ветка Y
     * В левую ветку Y подставляется N
     *
     */

    void leftRotate(Node<Elem>* n) {
        Node<Elem>* y = n->right;

        y->parent = n->parent; /* при этом, возможно, y становится корнем
дерева */
        if (n->parent != nullptr) {
            if (n->parent->left == n)
                n->parent->left = y;
            else
                n->parent->right = y;
        }

        n->right = y->left;
        if (y->left != nullptr)
            y->left->parent = n;

        n->parent = y;
        y->left = n;
    }

    /*
     * Функция правого поворота
     *          n                          y
     *         / \                        / \
     *        y  T3       --->           T1  n
     *       / \                            / \
     *      T1 T2                          T2 T3
     *
     * В левую ветку N подставляется правая ветка Y
     * В правую ветку Y подставляется N
     *
     */

    void rightRotate(Node<Elem>* n) {
        Node<Elem>* y = n->left;

        y->parent = n->parent; /* при этом, возможно, pivot становится
корнем дерева */
        if (n->parent != nullptr) {
            if (n->parent->left==n)
```

```
                n->parent->left = y;
            else
                n->parent->right = y;
        }

        n->left = y->right;
        if (y->right != nullptr)
            y->right->parent = n;

        n->parent = y;
        y->right = n;
    }

    // Поиск элемента + подсчитывание количества заданного элемента
    int find(Elem const& data, Node<Elem>* node, int count = 0) {
        auto format = generateFormatPrint(count);
        if (!node) { // Если Node пустой, то элементов 0
            std::cout << format << "Ветвь пустая, возвращаю 0" <<
std::endl;
            return 0;
        }
        // Если node->data > data, то нужно пойти в левую ветку
        if (data < node->data) {
            std::cout << format << data << " < " << *node << ". Вызов по-
иска для левой ветви" << std::endl;
            int result = find(data, node->left, count+1);
            std::cout << format << ". Получено значение от левой ветви -
" << result << std::endl;
            return result;
        }
        // Если node->data < data, то нужно пойти в правую ветку
        else if (data > node->data) {
            std::cout << format << data << " > " << *node << ". Вызов по-
иска для правой ветви" << std::endl;
            int result = find(data, node->right, count+1);
            std::cout << format << ". Получено значение от правой ветви -
" << result << std::endl;
            return result;
        }
        // Если node->data == data, то элемент найден и нужно посетить
left и right
        else {
            std::cout << format << data << " == " << *node << std::endl;
            std::cout << format << "Вызов поиска для правой ветви" <<
std::endl;
            int rightResult = find(data, node->right, count+1);
            std::cout << format << "Получено значение от правой ветви - "
<< rightResult << std::endl;

            std::cout << format << "Вызов поиска для левой летви" <<
std::endl;
            int leftResult = find(data, node->left, count+1);
            std::cout << format << "Получено значение от левой ветви - "
<< leftResult << std::endl;
            return leftResult + rightResult + 1;
        }
    }
```

```
    // случай, когда нет корня
    void insertCase1(Node<Elem>* n) {
        std::cout << "[CASE 1] ";

        if (n->parent == nullptr) {
            n->color = BLACK;
            std::cout << *n << " - не имеет отца, поэтому стал корнем де-
рева" << std::endl;
        } else {
            std::cout << *n << " - имеет отца, переход к следующему слу-
чаю" << std::endl;
            insertCase2(n);
        }
    }

    // случай, когда отец черный
    void insertCase2(Node<Elem>* n) {
        std::cout << "[CASE 2] ";
        if (n->parent->color == BLACK) {
            std::cout << "Отец " << *n->parent << " является черным,
свойство 3 не нарушено" << std::endl;
            return;
        } else {
            std::cout << "Отец " << *n->parent << " является красным,
свойство 3 нарушено" << std::endl;
            insertCase3(n);
        }

    }

    // случай, когда отец красный и есть красный дядя
    void insertCase3(Node<Elem>* n) {
        std::cout << "[CASE 3] ";
        Node<Elem>* u = uncle(n), *g;

        if ((u != nullptr) && (u->color == RED)) {
            // && (n->parent->color == RED) Второе условие проверяется в
insertCase2, то есть родитель уже является красным.
            std::cout << "У " << *n << " есть дядя - " << *u << " и он
красный" << std::endl;
            n->parent->color = BLACK;
            u->color = BLACK;
            g = grandparent(n);
            g->color = RED;
            std::cout << "[CASE 3] Отец стал черным, дядя стал черным,
дедушка стал красным. Повтор балансировки для дедушки" << std::endl;
            insertCase1(g);
        } else {
            std::cout << "У " << *n << " нет дяди или дядя не красный" <<
std::endl;
            insertCase4(n);
        }
    }

    // случай, когда нет красного дяди
    void insertCase4(Node<Elem>* n) {
        Node<Elem>* g = grandparent(n);
        std::cout << "[CASE 4] Дедушка " << *n << " - " << *g <<
```

```
std::endl;

        // n правый сын и отец левый сын
        if ((n == n->parent->right) && (n->parent == g->left)) {
            std::cout << "[CASE 4] " << *n << " является правым сыном
отца " << *n->parent << ", и отец является левым сыном" << std::endl;
            std::cout << "[CASE 4] " << "Левый поворот отца" <<
std::endl;
            leftRotate(n->parent);
            n = n->left;
        // n левый сын и отец правый сын
        } else if ((n == n->parent->left) && (n->parent == g->right)) {
            std::cout << "[CASE 4] " << *n << " является левым сыном отца
" << *n->parent << ", и отец является правым сыном" << std::endl;
            std::cout << "[CASE 4] " << "Правый поворот отца" <<
std::endl;
            rightRotate(n->parent);
            n = n->right;
        } else {
            std::cout << "[CASE 4] " << "Ни один из вариантов для случая
4 не выполнен для " << *n << std::endl;
        }
        std::cout << "[CASE 4] Переход к случаю 5" << std::endl;
        insertCase5(n);
    }

    // продолжение случая 4
    void insertCase5(Node<Elem>* n)
    {
        Node<Elem>* g = grandparent(n);
        n->parent->color = BLACK;
        g->color = RED;
        std::cout << "[CASE 5] Отец стал черным, дедушка стал красным" <<
std::endl;
        // n левый сын и отец левый сын
        if ((n == n->parent->left) && (n->parent == g->left)) {
            std::cout << "[CASE 5] " << *n << " является левым сыном отца
" << *n->parent << ", и отец является левым сыном" << std::endl;
            std::cout << "[CASE 5] " << "Правый поворот дедушки" <<
std::endl;
            rightRotate(g);
        } else { // n правый сын и отец правый сын
            std::cout << "[CASE 5] " << *n << " является правым сыном
отца " << *n->parent << ", и отец является правым сыном" << std::endl;
            std::cout << "[CASE 5] " << "Левый поворот дедушки" <<
std::endl;
            leftRotate(g);
        }
    }

public:
    RBTree() : root(nullptr) {}
    ~RBTree() { delete root; }

    Node<Elem>* getRoot() { return root; }

    int count(Elem const& data) { return find(data, root); }
```

```
    void insert(Elem stuff) { // Вставка элемента
        auto newNode = new Node<Elem>();
        newNode->data = stuff;
        newNode->color = RED;
        std::cout << "---Вставка элемента---" << std::endl;

        auto linker = root;

        // если node < linker, то идем в левую ветку
        // если node >= linker, то идем в правую ветку
        // когда нет след ветки, то вставляем туда элемент
        while (linker) {
            if (newNode->data < linker->data) {
                std::cout << *newNode << " < " << *linker << ", переход к
левой ветке" << std::endl;
                if (!linker->left) {
                    linker->left = newNode;
                    newNode->parent = linker;
                    std::cout << "Левая ветка пустая. ";
                    break;
                } else linker = linker->left;
            } else {
                std::cout << *newNode << " >= " << *linker << ", переход
к правой ветке" << std::endl;
                if (!linker->right) {
                    linker->right = newNode;
                    newNode->parent = linker;
                    std::cout << "Правая ветка пустая. ";
                    break;
                } else linker = linker->right;
            }
        }
        std::cout << "Элемент " << *newNode << " вставлен" << std::endl;
        std::cout << "-----Балансировка-----" << std::endl;
        insertCase1(newNode);
        restoreRoot(newNode);
        std::cout << "\n";
    }
};


void printTree(Node<int>* tree, int level) // печать дерева
{
    if(tree)
    {
        printTree(tree->right, level + 1);
        for (int i = 0; i < level; ++i) std::cout << "   ";
        std::cout << tree->color << std::endl;
        printTree(tree->left, level + 1);

    }
}


RBTree<int>* insert(int file = 0) {
    int N;
    RBTree<int>* tree;
```

```
    std::cout << "---Построение структуры данных начато---" << "\n\n";

    if (!file) {
        std::cout << "Введите кол-во элементов: ";
        std::cin >> N; // получаем кол-во элементов
        if (N <= 0) {
            if (N < 0) {
                std::cout << "Кол-во элементов должно быть >= 0" <<
std::endl;
                return nullptr;
            }
            else new RBTree<int>;
        }
        tree = new RBTree<int>;
        for (int i = 0; i < N; ++i) { // считываем N элементов
            int k;
            std::cout << "[" << i+1 << "] " << "Введите число: ";
            std::cin >> k;
            tree->insert(k); // Добавляем в список
        }
    } else {
        std::ifstream input("file.txt"); // открываем файл
        if (!input.is_open()) { // проверяем на доступность
            input.close();
            std::cout << "Не удалось открыть файл file.txt" << std::endl;
            return nullptr;
        }
        else {
            if(!(input >> N)) { // считываем N
                std::cout << "Не удалось считать кол-во элементов" <<
std::endl;
                return nullptr;
            }

            if (N <= 0) {
                if (N < 0) {
                    std::cout << "Кол-во элементов должно быть >= 0" <<
std::endl;
                    return nullptr;
                }
                else new RBTree<int>;
            }

            tree = new RBTree<int>;
            for (int i = 0; i < N; ++i) { // Считываем N элементов
                int k;
                if(!(input >> k)) { // проверяем, получилось ли считать
                    std::cout << "Задано " << N << " элементов, но было
введено меньше " << N << std::endl;
                    delete tree;
                    return nullptr; // если не удалось, вызываем деструк-
тор списка
                }
                else tree->insert(k); // добавляем элемент
            }
        }
    }
    std::cout << "---Построение структуры данных завершено---" << "\n\n";
```

```
    return tree;
}

char executeTask(RBTree<int>* tree) {
    int e;
    std::cout << "Введите элемент, который хотите найти в дереве: ";
    std::cin >> e;
    std::cout << "---Подсчет количества элементов со значением '" << e <<
"'" << " начат---" << std::endl;
    int count = tree->count(e);
    std::cout << "Результат подсчета: " << count << std::endl;
    std::cout << "---Подсчет количества элементов окончен---" << "\n\n";

    char next;
    std::cout << "Если хотите продолжить, введите '+': ";
    std::cin >> next;
    return next;
}

int main()
{
    int file;
    std::cout << "Если хотите строить дерево в live-режиме, введите 0\n";
    std::cout << "Иначе любое другое число\n";
    std::cout << "Ввод: ";
    std::cin >> file;

    RBTree<int>* tree = insert(file);
    if (!tree) return 0;

    char flag = '+';
    while (flag == '+') flag = executeTask(tree);

    std::cout << "---Представление дерева---" << std::endl;
    std::cout << "0 - КРАСНЫЙ" << std::endl;
    std::cout << "1 - ЧЕРНЫЙ" << std::endl;
    printTree(tree->getRoot(), 0);

    return 0;
}
```