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-чёрное дерево. Поиск и удаление. Исследование.

Студентка гр. 1304 _____ Чернякова В.А.

Преподаватель _____ Иванов Д.В.

Санкт-Петербург
2022

ЗАДАНИЕ

НА КУРСОВУЮ РАБОТУ

Студентка Чернякова В.А.

Группа 1304

Тема работы: Исследование структур данных красно-чёрные деревья.

Исходные данные:

Вариант 15.

Красно-черные деревья. Исследование операций удаления и поиска в лучшем, в среднем и в худшем случае. Реализация требуемых структур данных/алгоритмов; генерация входных данных (вид входных данных определяется студентом); использование входных данных для измерения количественных характеристик структур данных, алгоритмов, действий; сравнение экспериментальных результатов с теоретическими. Вывод промежуточных данных не является строго обязательным, но должна быть возможность убедиться в корректности алгоритмов.

Содержание пояснительной записки:

«Содержание», «Введение», «Основные теоретические положения», «Разработка программного кода», «Исследование», «Заключение», «Список использованных источников».

Предполагаемый объем пояснительной записки:

Не менее 20 страниц

Дата выдачи задания: 25.10.2022

Дата сдачи реферата: 23.12.2022

Дата защиты реферата: 24.12.2022

Студентка _____ Чернякова В.А.

Преподаватель _____ Иванов Д.В.

АННОТАЦИЯ

В ходе данной курсовой работе было проведено исследование структуры данных красное-чёрные деревья. Осуществлен сравнительный анализ операций вставки и удаления на основе теоретических знаний и полученных практических значений. В программном коде был реализованы класс RBTree. Он представляет собой реализацию упомянутой выше структуры на языке программирования Python. С помощью методов происходит добавление, удаление, поиск элементов в структуре данных. Другие методы являются вспомогательными и необходимы для корректной работы и реализации структуры данных.

SUMMARY

In the course of this course work, a study of the data structure of RB-trees was conducted. A comparative analysis of insertion and deletion operations is carried out on the basis of theoretical knowledge and the obtained practical values. The RBTree class was implemented in the program code. It is an implementation of the structure mentioned above in the Python programming language. The methods are used to add, delete, and search for elements in the data structure. Other methods are auxiliary and necessary for the correct operation and implementation of the data structure.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью данной работы является реализация структуры данных – RB-Tree и исследование этой структуры на сложность работы алгоритмов, таких как поиск и удаление. Также необходимо сравнить теоретическое и практическое время работы и по полученным данным осуществить анализ.

На основе вышеизложенной цели были сформулированы задачи, необходимые для ее достижения:

1) Реализовать структуру данных RB-дерево и соответствующие методы.
2) Определить время работы структуры при разных входных данных.
3) Построить графики полученных данных.
4) Сравнить практические и теоретические положения.
5) Сделать выводы.

1. ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ ПОЛОЖЕНИЯ

1.1. Определение красно-чёрного дерева

Красно-чёрное дерево (англ. red-black tree) — двоичное дерево поиска, в котором баланс осуществляется на основе "цвета" узла дерева, который принимает только два значения: "красный" (англ. red) и "чёрный" (англ. black).

При этом все листья дерева являются фиктивными и не содержат данных, но относятся к дереву и являются чёрными.

1.2. Свойства

Красно-чёрным называется бинарное поисковое дерево, у которого каждому узлу сопоставлен дополнительный атрибут — цвет и для которого выполняются следующие свойства:

- Каждый узел промаркирован красным или чёрным цветом.
- Корень и конечные узлы (листья) дерева — чёрные.
- У красного узла родительский узел — чёрный.
- Все простые пути из любого узла x до листьев содержат одинаковое количество чёрных узлов.
- Чёрный узел может иметь чёрного родителя.

Также к особым свойствам красно-чёрного дерева можно отнести:

- Каждая вершина — либо красная, либо черная.
- Каждый лист — черный.
- Если вершина красная, оба ее ребенка черные.
- Все пути, идущие от корня к листьям, содержат одинаковое количество черных вершин.

1.3. Высота красно-чёрного дерева

Определение: будем называть чёрной высотой (англ. black-height) вершины x число чёрных вершин на пути из x в лист.

Лемма: в красно-черном дереве с черной высотой hb количество внутренних вершин не менее $2^{hb-1}-1$.

1.4. Вставка элемента

Каждый элемент вставляется вместо листа, поэтому для выбора места вставки идём от корня до тех пор, пока указатель на следующего сына не станет nil (то есть этот сын — лист). Вставляем вместо него новый элемент с нулевыми потомками и красным цветом. Теперь проверяем балансировку. Если отец нового элемента черный, то никакое из свойств дерева не нарушено. Если же он красный, то нарушается свойство 3, для исправления достаточно рассмотреть два случая:

"Дядя" этого узла тоже красный. Тогда, чтобы сохранить свойства 3 и 4, просто перекрашиваем "отца" и "дядю" в чёрный цвет, а "деда" — в красный. В таком случае черная высота в этом поддереве одинакова для всех листьев и у всех красных вершин "отцы" черные. Проверяем, не нарушена ли балансировка. Если в результате этих перекрашиваний мы дойдём до корня, то в нём в любом случае ставим чёрный цвет, чтобы дерево удовлетворяло свойству 2. Untitled-1.png

"Дядя" чёрный. Если выполнить только перекрашивание, то может нарушиться постоянство чёрной высоты дерева по всем ветвям. Поэтому выполняем поворот. Если добавляемый узел был правым потомком, то необходимо сначала выполнить левое вращение, которое сделает его левым потомком. Таким образом, свойство 3 и постоянство черной высоты сохраняются.

1.5. Удаление элемента

При удалении вершины могут возникнуть три случая в зависимости от количества её детей:

Если у вершины нет детей, то изменяем указатель на неё у родителя на nil.

Если у неё только один ребёнок, то делаем у родителя ссылку на него вместо этой вершины.

Если же имеются оба ребёнка, то находим вершину со следующим значением ключа. У такой вершины нет левого ребёнка (так как такая вершина находится в правом поддереве исходной вершины, и она самая левая в нем, иначе бы мы взяли ее левого ребенка. Иными словами, сначала мы переходим в правое поддерево, а после спускаемся вниз в левое до тех пор, пока у вершины есть левый ребенок). Удаляем уже эту вершину описанным во втором пункте способом, скопировав её ключ в изначальную вершину.

Проверим балансировку дерева. Так как при удалении красной вершины свойства дерева не нарушаются, то восстановление балансировки потребуется только при удалении чёрной. Рассмотрим ребёнка удалённой вершины.

Если брат этого ребёнка красный, то делаем вращение вокруг ребра между отцом и братом, тогда брат становится родителем отца. Красим его в чёрный, а отца — в красный цвет, сохраняя таким образом черную высоту дерева. Хотя все пути по-прежнему содержат одинаковое количество чёрных узлов, сейчас x имеет чёрного брата и красного отца. Таким образом, мы можем перейти к следующему шагу.

Если брат текущей вершины был чёрным, то получаем три случая:

Оба ребёнка у брата чёрные. Красим брата в красный цвет и рассматриваем далее отца вершины. Делаем его черным, это не повлияет на количество чёрных узлов на путях, проходящих через b, но добавит один к числу чёрных узлов на путях, проходящих через x, восстанавливая тем самым влияние удаленного чёрного узла. Таким образом, после удаления вершины черная глубина от отца этой вершины до всех листьев в этом поддереве будет одинаковой.

Если у брата правый ребёнок чёрный, а левый красный, то перекрашиваем брата и его левого сына и делаем вращение. Все пути по-прежнему содержат одинаковое количество чёрных узлов, но теперь у x есть чёрный брат с красным правым потомком, и мы переходим к следующему случаю. Ни x, ни его отец не влияют на эту трансформацию.

Если у брата правый ребёнок красный, то перекрашиваем брата в цвет отца, его ребёнка и отца — в чёрный, делаем вращение. Поддерево по-прежнему имеет тот же цвет корня, поэтому свойство 3 и 4 не нарушаются. Но у x теперь появился дополнительный чёрный предок: либо a стал чёрным, или он и был чёрным и b был добавлен в качестве чёрного дедушки. Таким образом, проходящие через x пути проходят через один дополнительный чёрный узел. Выходим из алгоритма.

Продолжаем тот же алгоритм, пока текущая вершина чёрная и мы не дошли до корня дерева. Из рассмотренных случаев ясно, что при удалении выполняется не более трёх вращений.

1.6. Сложности алгоритма красно-чёрного дерева

Сложность любой операции в красно-черном дереве для любого случая составляет O(logN). Конкретная оценка сложности представлена в таблице 1.

Таблица 1. Оценка сложности операций в красно-черном дереве.

Временная сложность в О-символике			
	В лучшем случае	В среднем случае	В худшем случае
Расход памяти	O(n)	O(n)	O(n)
Поиск	O(log n)	O(log n)	O(log n)
Вставка	O(log n)	O(log n)	O(log n)
Удаление	O(log n)	O(log n)	O(log n)

2. РАЗРАБОТКА ПРОГРАММНОГО КОДА

2.1. Класс RBTree

Для реализации класса *RBTree* был реализован дополнительный класс – класс *Node*.

Класс *Node* является классом, который представляет узлы красно-чёрного дерева.

Поля:

key — поле, хранящее значение узла.

left — поле, хранящее левый дочерний узел, исходного узла.

right — поле, хранящее правый дочерний узел, исходного узла.

parent — поле, хранящее родительский узел, исходного узла.

color — поле, хранящее цвет узла. *RED* или *BLACK* соответственно.

Методы класса *RBTree:*

def left_rotate(self, node) – вспомогательный метод, необходимый для левого поворота.

def right_rotate(self, node) – вспомогательный метод, необходимый для поворота узлов дерева вправо.

def insert(self, key) – метод, принимающий на вход значение и вставляющий его в дерево по определенным свойствам.

def fix_insert(self, node) – вспомогательный метод, необходимый для исправления ошибок при вставке нового элемента.

def search(self, key) – метод, позволяющий найти узел с определенным значением.

def delete(self, key) – метод, удаляющий узел с заданным значением с помощью вспомогательного метода *delete_helper().*

def delete_helper(self, node, key) - вспомогательный метод для удаления узла из дерева.

def change(self, p, c) – вспомогательный метод для того, чтобы поменять местами два узла.

def minimum(self, node) — метод, получающий на вход узел дерева и находит минимальный элемент из всех дочерних узлов.

def fix_delete(self, node) – вспомогательный метод, необходимый для исправления ошибок при удалении элемента.

def print_node(self, node, indent, last) – вспомогательный метод, выводящий дерево в консоль.

def print_tree(self) – метод вывода в консоль дерева с помощью вспомогательного метода *print_node().*

3. ИССЛЕДОВАНИЕ

3.1. Операция удаления в красно-чёрном дереве

Как и другие операции, удаление вершины из красно-чёрного дерева требует времени *O(log n)*.

В каждом случае: худшем, среднем, лучшем – удаление происходит за *O(log n)*.

Для исследования была подключена библиотека *import time,* для того, чтобы засечь время работа алгоритма.

Исследования проводилось с деревом, состоящим из 5000 узлов. При удалении элементов их количество уменьшалось, таким образом при удалении всех узлов исследовалась сложность данной операции для деревьев различной высоты. Также такой подход к исследованию позволяет учесть всевозможные случаи, то есть худший, средний и лучший.

Далее представлен график зависимости количества узлов от времени совершения операции.

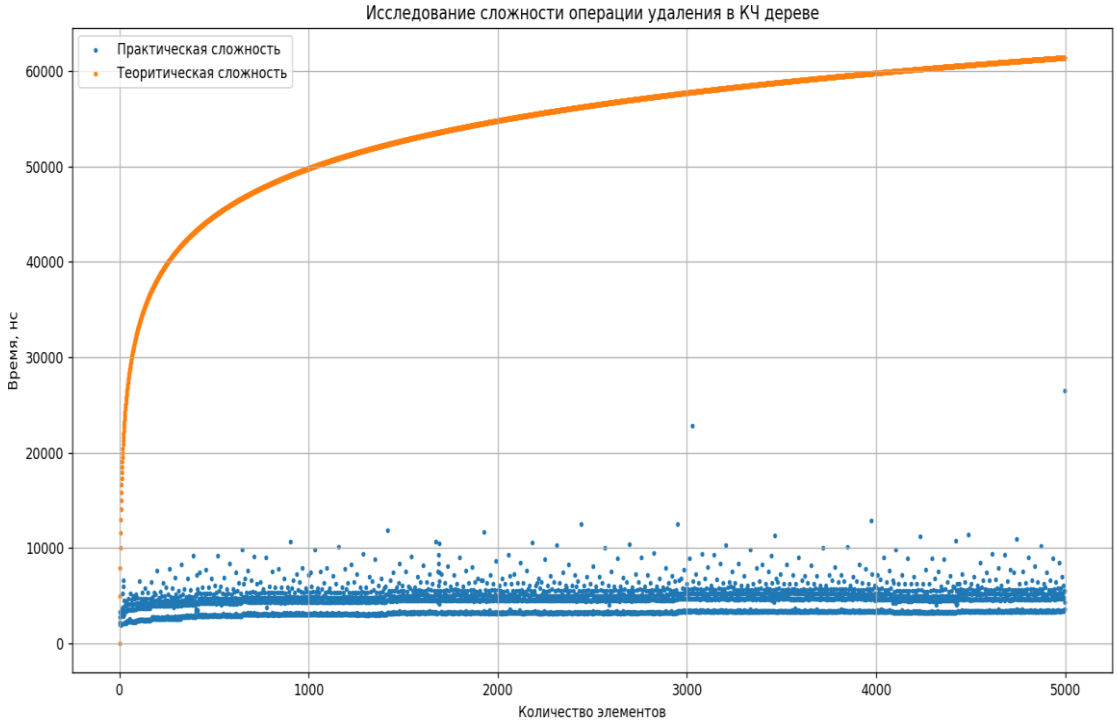

График 1 – сравнение практического и теоретического времени работы удаления элемента в красно-чёрном дереве.

Сравнив данные графики, можно сделать вывод, что теоретическое и практическое время работы удаления элементов в чёрно-красном дереве совпадают, однако наблюдаются небольшие различия. Данные различия, скорее всего, связаны с погрешностью и коэффициентами, которые появляются в практическом подсчёте, которые в теоретическом отбрасываются.

3.2. Операция поиска в красно-чёрном дереве

Как и другие операции, поиск вершины в красно-чёрном дереве требует времени *O(log n)*.

В каждом случае: худшем, среднем, лучшем – поиск происходит за *O(log n)*.

Для исследования была подключена библиотека import time, для того, чтобы засечь время работа алгоритма.

Исследования проводилось с деревом, состоящим из 5000 узлов. Алгоритм поиска был применен к каждой вершине полученного дерева после ее добавления в само дерево, таким образом исследовалась сложность данной операции для деревьев различной высоты. Также такой подход к исследованию позволяет учесть всевозможные случаи, то есть худший, средний и лучший.

Далее представлен график зависимости количества узлов от времени совершения операции.

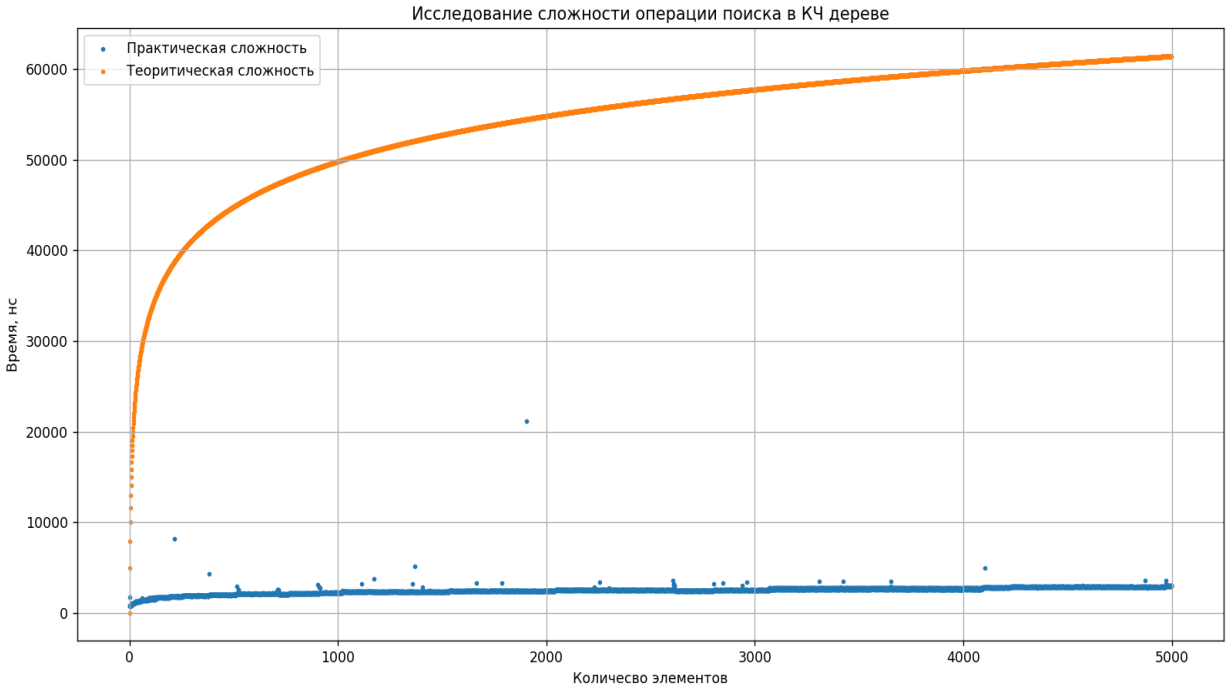

График 2 – сравнение практического и теоретического времени работы поиска элемента в красно-чёрном дереве.

Сравнив данные графики, можно сделать вывод, что теоретическое и практическое время работы поиска элементов в чёрно-красном дереве совпадают, однако наблюдаются небольшие различия. Данные различия, скорее всего, связаны с погрешностью и коэффициентами, которые появляются в практическом подсчёте, которые в теоретическом отбрасываются.

3.3. Сравнение практических и теоретических значений

Исходя из представленных выше вычислений и основываясь на полученных графиках, можно сделать вывод, что сложность работы реализованных алгоритмов поиска и удаления элемента совпадает с теоретическими. Таким образом, было подтверждено теоретическое значение сложности алгоритмов, равное $O(\log n)$.

ЗАКЛЮЧЕНИЕ

Была исследована структура данных: *RB-дерево* (красно-чёрное дерево). В программном коде был реализован класс *RBTree,* который представляет из себя реализацию упомянутой выше структуры на языке программирования *Python.*

Был проведен анализ сложности алгоритмов поиска и удаления элементов в красно-чёрном дереве, а именно сравнение практических и теоретических значений. По полученным данным отличий почти не выявлено, небольшие отклонения появляются из-за погрешности и коэффициента.

Сложность данных алгоритмов красно-чёрного дерева равно $O(\log n)$ в любом случае. Это говорит о стабильности данной структуры, то есть вне зависимости от обрабатываемого случая время работы любого алгоритма будет всегда одно и тоже.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Сайт https://ru.wikipedia.org/wiki/Красно-чёрное_дерево
2. Т.Кормен, Ч.Лейзерсон, Р. Ривест, К.Штайн «Алгоритмы. Построение и анализ»

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл main.py

```
BLACK = 'black'
RED = 'red'


class Node:
    def __init__(self, key):
        self.key = key
        self.parent = None
        self.left = None
        self.right = None
        self.color = RED


class RBTree:
    def __init__(self):
        self.NULL = Node(0)
        self.NULL.color = BLACK
        self.NULL.left = None
        self.NULL.right = None
        self.root = self.NULL

    def left_rotate(self, node):
        new_node = node.right
        node.right = new_node.left

        if new_node.left != self.NULL:
            new_node.left.parent = node

        new_node.parent = node.parent

        if node.parent is None:
            self.root = new_node
        elif node == node.parent.left:
```

```
            node.parent.left = new_node
        else:
            node.parent.right = new_node

        new_node.left = node
        node.parent = new_node

    def right_rotate(self, node):
        new_node = node.left
        node.left = new_node.right

        if new_node.right != self.NULL:
            new_node.right.parent = node

        new_node.parent = node.parent

        if node.parent is None:
            self.root = new_node
        elif node == node.parent.right:
            node.parent.right = new_node
        else:
            node.parent.left = new_node

        new_node.right = node
        node.parent = new_node

    def insert(self, key):
        node = Node(key)
        node.parent = None
        node.key = key
        node.left = self.NULL
        node.right = self.NULL
        node.color = RED

        parent = None
        current = self.root

        while current != self.NULL:
```

```
            parent = current
            if node.key < current.key:
                current = current.left
            else:
                current = current.right

        node.parent = parent

        if parent is None:
            self.root = node
        elif node.key < parent.key:
            parent.left = node
        else:
            parent.right = node

        if node.parent is None:
            node.color = BLACK
            return

        if node.parent.parent is None:
            return

        self.fix_insert(node)

    def fix_insert(self, node):
        while node.parent.color == RED:
            if node.parent == node.parent.parent.right:
                uncle = node.parent.parent.left
                if uncle.color == RED:
                    uncle.color = BLACK
                    node.parent.color = BLACK
                    node.parent.parent.color = RED
                    node = node.parent.parent
                else:
                    if node == node.parent.left:
                        node = node.parent
                        self.right_rotate(node)
                    node.parent.color = BLACK
```

```
                node.parent.parent.color = RED
                self.left_rotate(node.parent.parent)
            else:
                uncle = node.parent.parent.right
                if uncle.color == RED:
                    uncle.color = BLACK
                    node.parent.color = BLACK
                    node.parent.parent.color = RED
                    node = node.parent.parent
                else:
                    if node == node.parent.right:
                        node = node.parent
                        self.left_rotate(node)
                    node.parent.color = BLACK
                    node.parent.parent.color = RED
                    self.right_rotate(node.parent.parent)
            if node == self.root:
                break

        self.root.color = BLACK

    def search(self, key):
        current = self.root
        while current is not None:
            if key < current.key:
                current = current.left
            elif key > current.key:
                current = current.right
            elif key == current.key:
                return current
        return None

    def delete(self, key):
        self.delete_helper(self.root, key)

    def delete_helper(self, node, key):
        delete = self.NULL
```

```
            while node != self.NULL:
                if node.key == key:
                    delete = node

                if node.key <= key:
                    node = node.right
                else:
                    node = node.left

            if delete == self.NULL:
                print("There is no such key in the tree! Deletion
failed.")
                return

            n_change = delete
            change_original_color = n_change.color

            if delete.left == self.NULL:
                n_fix = delete.right
                self.change(delete, delete.right)
            elif delete.right == self.NULL:
                n_fix = delete.left
                self.change(delete, delete.left)
            else:
                n_change = self.minimum(delete.right)
                change_original_color = n_change.color
                n_fix = n_change.right
                if n_change.parent == delete:
                    n_fix.parent = n_change
                else:
                    self.change(n_change, n_change.right)
                    n_change.right = delete.right
                    n_change.right.parent = n_change

                self.change(delete, n_change)
                n_change.left = delete.left
                n_change.left.parent = n_change
                n_change.color = delete.color
```

```
            if change_original_color == BLACK:
                self.fix_delete(n_fix)

        def change(self, p, c):
            if p.parent is None:
                self.root = c
            elif p == p.parent.left:
                p.parent.left = c
            else:
                p.parent.right = c
            c.parent = p.parent

        def minimum(self, node):
            while node.left != self.NULL:
                node = node.left
            return node

        def fix_delete(self, node):
            while node != self.root and node.color == BLACK:
                if node == node.parent.left:
                    brother = node.parent.right
                    if brother.color == RED:
                        brother.color = BLACK
                        node.parent.color = RED
                        self.left_rotate(node.parent)
                        brother = node.parent.right
                    if     brother.left.color     ==     BLACK     and
brother.right.color == BLACK:
                        brother.color = RED
                        node = node.parent
                    else:
                        if brother.right.color == BLACK:
                            brother.left.color = BLACK
                            brother.color = RED
                            self.right_rotate(brother)
                            brother = node.parent.right
```

```
                        brother.color = node.parent.color
                        node.parent.color = BLACK
                        brother.right.color = BLACK
                        self.left_rotate(node.parent)
                        node = self.root
                else:
                    brother = node.parent.left
                    if brother.color == RED:
                        brother.color = BLACK
                        node.parent.color = RED
                        self.right_rotate(node.parent)
                        brother = node.parent.left
                    if    brother.right.color    ==    BLACK    and
brother.right.color == BLACK:
                        brother.color = RED
                        node = node.parent
                    else:
                        if brother.left.color == BLACK:
                            brother.right.color = BLACK
                            brother.color = RED
                            self.left_rotate(brother)
                            brother = node.parent.left

                        brother.color = node.parent.color
                        node.parent.color = BLACK
                        brother.left.color = BLACK
                        self.right_rotate(node.parent)
                        node = self.root

            node.color = BLACK

        def print_node(self, node, indent, last):
            if node != self.NULL:
                print(indent, end=' ')
                if last:
                    print("R--->", end=' ')
                    indent += "   "
                else:
```

```
                print("L--->", end=' ')
                indent += "|  "

            node_color = "RED" if node.color == RED else "BLACK"
            print(str(node.key) + "(" + node_color + ")")
            self.print_node(node.left, indent, False)
            self.print_node(node.right, indent, True)

    def print_tree(self):
        self.print_node(self.root, "", True)


if __name__ == "__main__":
    tree = RBTree()

    for i in range(30):
        tree.insert(i)

    tree.print_tree()

    tree.delete(16)

    print("Дерево после удаления элемента ")
    tree.print_tree()

    if tree.search(226) is None:
        print("Данный элемент отсвутсвует в дереве")
    else:
        print("Элемент ", 226, " найден")
```