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-чёрные деревья. Вставка, удаление, поиск элемента. Демонстрация.

Студент гр. 1381 _____ Возмитель В. Е.

Преподаватель _____ Иванов Д. В.

Санкт-Петербург

2022

ЗАДАНИЕ
НА КУРСОВУЮ РАБОТУ

Студент Возмитель Влас Евгеньевич

Группа 1381

Тема работы: Красно-чёрные деревья. Вставка, удаление, поиск элемента. Демонстрация

Содержание пояснительной записки: «Содержание», «Введение», «Ход выполнения работы», «Заключение», «Список использованных источников».

Дата выдачи задания: 25.10.22

Дата сдачи реферата: 23.12.2022

Дата защиты реферата: 24.12.2022

Студент _____ Возмитель В. В.

Преподаватель _____ Иванов Д. В.

АННОТАЦИЯ

В данной курсовой работе реализуется и рассматривается структура данных «красно-черные деревья». Реализация на языке #python с использованием библиотеки graphviz для демонстрации структуры.

SUMMARY

In this course work, the "red-black trees" data structure is implemented and considered. Implementation in #python using graphviz library to demonstrate the structure.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ..... 5
1.1 Цель работы..... 5
1.2 Теоретические сведения..... 5
ХОД ВЫПОЛНЕНИЯ РАБОТЫ..... 6
1.1. Хранение дерева..... 6
1.2. Поиск элемента. 6
1.3. Вставка элемента. 6
1.4 Удаление элемента и восстановление свойств. 9
1.5 Повороты. 12
1.6 Демонстрация и тесты. 12
ЗАКЛЮЧЕНИЕ 15
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 16
ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД ПРОГРАММЫ...... 17

ВВЕДЕНИЕ

1.1 Цель работы.

Визуализация работы структуры данных красно-черных деревьев процесса работы операций.

1.2 Теоретические сведения.

Красно-черное дерево – это бинарное дерево поиска, то есть такое бинарное дерево, в котором у всех узлов левого поддерева произвольного узла X значения ключей данных меньше, нежели значение ключа данных самого узла X, а в правом поддереве больше или равны. В красно-черном дереве каждый узел является либо красным, либо черным и выполнены следующие свойства:

- Корень дерева является черным узлом.
- Каждый фиктивный лист (nil) является черным узлом.
- Если узел красный, то оба дочерних узла черные.
- Для каждого узла все простые пути от него до его потомков листьев содержит одинаковое количество черных узлов.

При выполнении операций вставки или удаления элемента в красно-черное дерево все свойства сохраняются благодаря балансировке дерева. Из-за того, что все перечисленные свойства выполняются, красно-черное дерево гарантирует логарифмическую зависимость высоты дерева от количества узлов. Красно-черное дерево с n узлами имеет высоту не превышающую $2\log(n+1)$, поэтому операции вставки, удаления и поиска элемента работают в нем за время $O(\log(n))$.

ХОД ВЫПОЛНЕНИЯ РАБОТЫ

1.1. Хранение дерева.

Реализуемое дерево описывается в классе RBTree, узлы которого в свою очередь хранятся в классе Node, имеющий следующие поля в конструкторе:

- *self.item* – значение узла
- *self.parent* – «родитель» узла
- *self.left* – левый лист узла
- *self.right* – правый лист узла
- *self.color* – цвет узла

Также в классе переопределен метод *__str__* для удобства демонстрации любого узла дерева.

Класс имеет конструктор, не принимающий никаких значений, который при создании объекта создает дерево, состоящее из одного листа.

1.2. Поиск элемента.

Просмотр содержимого узлов дерева не может нарушить его красно-черные свойства, поэтому поиск элемента в красно-черном дереве не отличается от поиска элемента в любом бинарном дереве поиска.

Поиск начинается с корня и продолжается, пока не найден узел с искомым значением или достигнут фиктивный лист дерева (в этом случае элемент отсутствует в дереве). На каждой итерации алгоритма поиска рассматривается определенный узел дерева, если значение в нем меньше искомого, то поиск продолжается в левом поддереве, иначе в правом.

1.3. Вставка элемента.

Вставка узла в красно-черное дерево с n узлами может быть выполнена за время O(log n). Для того чтобы вставка сохраняла красно-черные свойства

дерева, после нее вызывается вспомогательная процедура *fix_insert,* которая перекрашивает узлы и выполняет повороты.

Для вставки элемента необходимо спуститься от корня до фиктивного листа – места куда будет вставлен элемент. Поиск места для вставки элемента осуществляется аналогично тому, как это происходит при просто поиске элемента в дереве. После того, как найдено место для вставки элемента, в это место вставляется красный узел с новым значением. В методе создается объект класса *Node*. Если в дереве только лист, то добавленная вершина становится черным корнем. Если родитель нового узла черный, то необходимые свойства сохраняются, иначе необходима балансировка, при которой рассматриваются два случая:

- "Дядя" этого узла тоже красный. Тогда, чтобы сохранить красно-черные свойства, 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. Если в результате этих перекрашиваний мы дойдём до корня, то в нём в любом случае ставим чёрный цвет, чтобы дерево удовлетворяло свойству о том, что корень всегда черный.

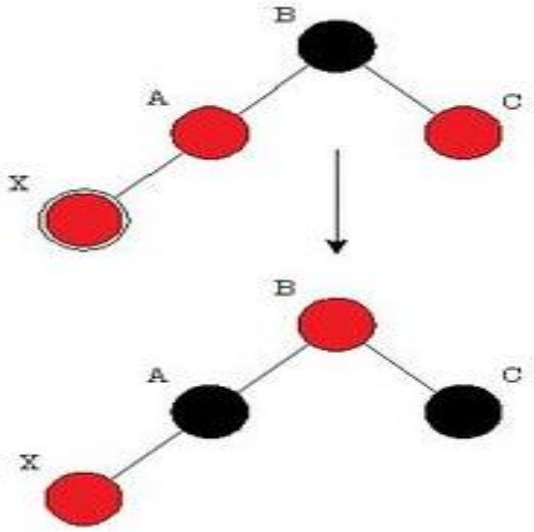

Рис №1. Первый случай балансировки при добавлении элемента (X – добавляемый элемент).

- "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, а его родитель левым, то необходимо сначала выполнить левое вращение для родителя, которое сделает добавляемый элемент левым потомком (аналогично, если добавляемый узел – левый потомок, а его родитель правый, то необходимо правое вращение для родителя). Далее родителю присваивается черный цвет, “деду” красный и происходит правое вращение для “деда”, если родитель был левым ребенком (если он был правым ребенком, то левое вращение). Таким образом, красно-черные свойства дерева сохраняются.

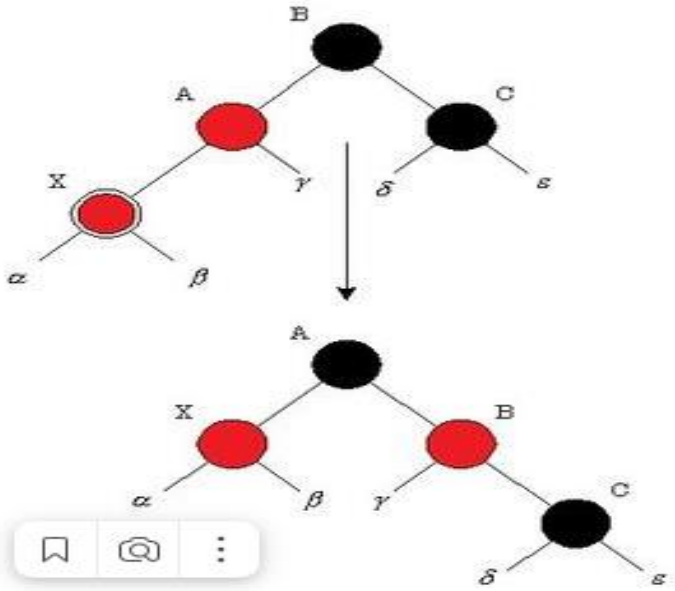

Рис №2. Пример балансировки второго случая при добавлении элемента.

Поскольку высота красно-черного дерева с n узлами равна O(log n), выполнение процедуры *rb_insert* требует O(log n) времени. В процедуре *rb_insert_fix* цикл *while* повторно выполняется только в случае 1, и в этом случае указатель k перемещается вверх по дереву на два уровня. Таким образом, общее количество возможных выполнений тела цикла *while* равно O (log n). Таким образом, общее время работы процедуры *rb_insert* равно O(log n)

1.4 Удаление элемента и восстановление свойств.

Как и остальные базовые операции над красно-черными деревьями с n узлами, удаление узла выполняется за время O(log n). Удаление оказывается несколько более сложной задачей, чем вставка.

Если узел y — красный, красно-черные свойства при извлечении y из дерева сохраняются в силу следующих причин:

- никакая черная высота в дереве не изменяется;
- никакие красные узлы не становятся соседними;
- так как y не может быть корнем в силу своего цвета, корень остается черным

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

1. Если у вершины нет детей, то изменяем указатель на неё у родителя на *nil*.
2. Если у неё только один ребёнок, то делаем у родителя ссылку на него вместо этой вершины.
3. Если же имеются оба ребёнка, то находим вершину со следующим значением ключа. У такой вершины нет левого ребёнка. Удаляем уже эту вершину описанным во втором пункте способом, скопировав её ключ в изначальную вершину.

Проверим балансировку дерева. Т. к. 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, однако полную балансировку пока нельзя гарантировать.

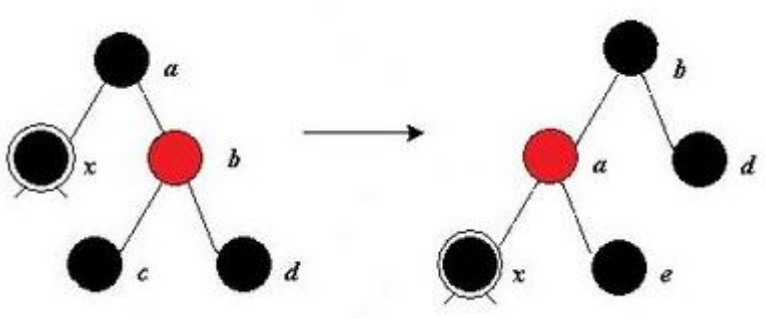

Рис №3. Пример балансировки для 1 случая (удаляем X).

- Если брат текущей вершины был черным, то получаем три случая:

1. Если 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 брата, но добавит один к числу чёрных узлов на путях, проходящих через текущую вершину, восстанавливая тем самым то, что мы удалили черный узел. Таким образом, после удаления узла черная глубина от отца этой вершины до всех листьев в этом поддереве будет одинаковой.

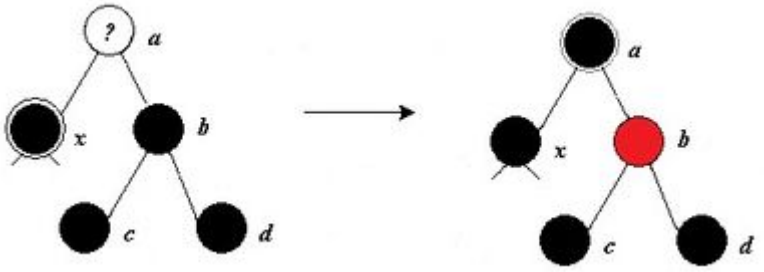

Рис №4. Пример балансировки для 2.1 случая (удаляем X).

2. 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 текущего 10 узла есть чёрный брат с красным правым потомком, и мы переходим к следующему случаю.

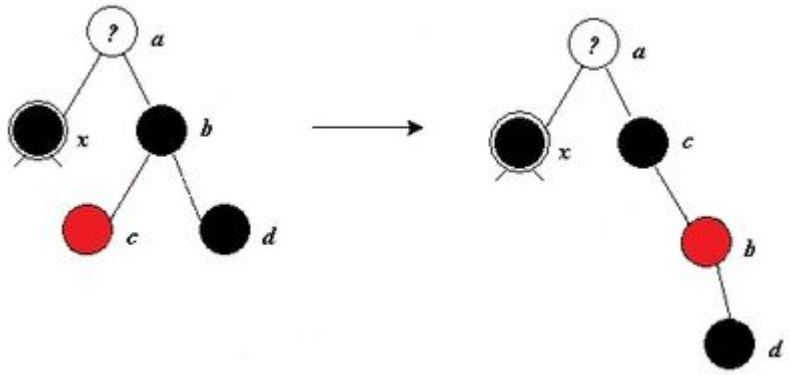

Рис №5. Пример балансировки для 2.2 случая (удаляем X).

3. 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. Но у текущей вершины теперь появился дополнительный чёрный предок: либо старый дедушка стал чёрным, или он и был чёрным, и новый дедушка стал черным. Таким образом, проходящие через текущий узел пути проходят через один дополнительный чёрный узел. Этот алгоритм повторяется пока мы не дошли до корня и текущая вершина черная.

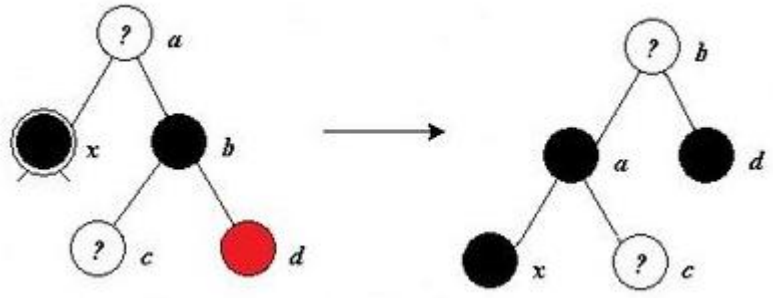

Рис №6. Пример балансировки для 3 случая (удаляем X).

Поскольку высота дерева с n узлами равна O(log n), общее время работы процедуры без выполнения вспомогательной процедуры *delete_fix* равно O (lg n). В процедуре *delete_fix* в случаях 1, 3 и 4 завершение работы происходит после выполнения постоянного числа изменений цвета и не более трех поворотов. Случай 2 — единственный, после которого возможно выполнение очередной итерации цикла while, причем указатель x перемещается вверх по дереву не более чем O(log n) раз, и никакие повороты при этом не

выполняются. Таким образом, время работы процедуры *delete_fix* составляет O(log n), причем она выполняет не более трех поворотов. Общее время работы процедуры *delete_node_helper*, само собой разумеется, также равно O(log n).

1.5 Повороты.

Операции над деревом поиска изменяют дерево, в результате их работы могут нарушаться красно-черные свойства. Для восстановления этих свойств мы должны изменить цвета некоторых узлов дерева, а также структуру его указателей. Изменения в структуре указателей будут выполняться при помощи поворотов (*left_rotate*, *right_rotate*), которые представляют собой локальные операции в дереве поиска, сохраняющие свойство бинарного дерева поиска. На рис. 13.2 показаны два типа поворотов — левый и правый (здесь α, β и γ — произвольные поддеревья). При выполнении левого поворота в узле x предполагается, что его правый дочерний узел y не является листом nil. Левый поворот выполняется “вокруг” связи между x и y, делая y новым корнем поддерева, левым дочерним узлом которого становится x, а бывший левый потомок узла y — правым потомком x.

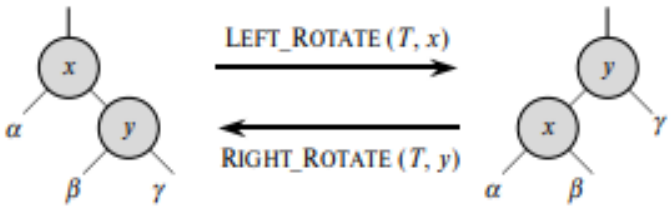

Рис №7. Операции поворота в бинарном дереве поиска.

Обе эти процедуры выполняются за время O(1). При повороте изменяются только указатели, все остальные поля сохраняют свое значение

1.6 Демонстрация и тесты.

Как я уже писал выше демонстрация структуры данных реализуется при помощи библиотеки graphviz. Используя методы библиотеки, создается

директория и ряд конечных *pdf* файлов, в которых содержится изображение графа на каждом шаге взаимодействия с ним.

Создан метод класс с модификатором *public* – *showing(self, index, tree),* который предназначен для создания узлов и дуг красно-черного дерева. В конце метода создаются файлы соответствующего шага с текущим деревом.

Ниже приведены примеры работы программы:

Пример удаления:

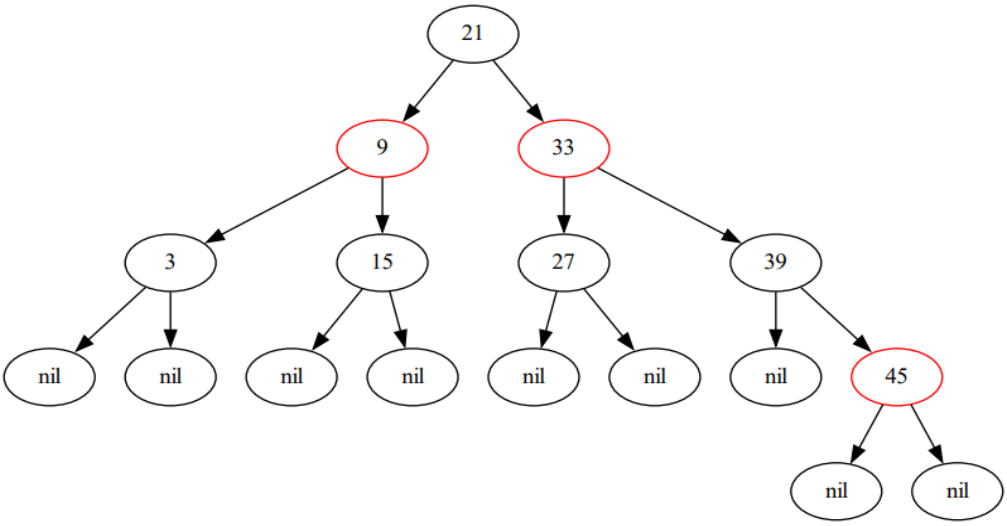

Рис №8. Пример красно-черного дерева.

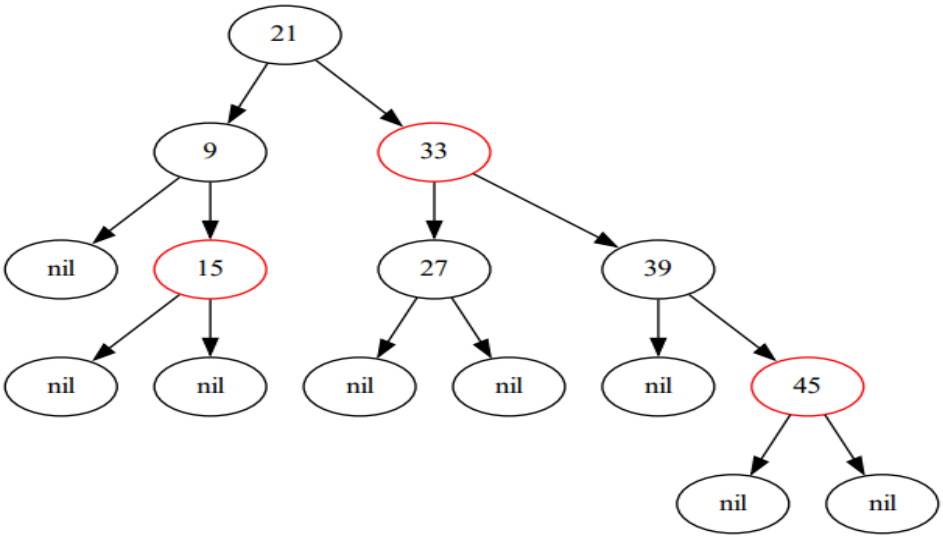

Рис №9. Удалили узел с ключом «3».

Пример вставки:

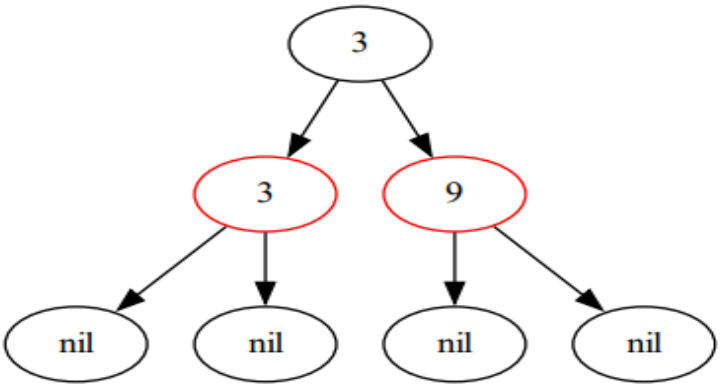

Рис. №10. Пример красно-черного дерева.

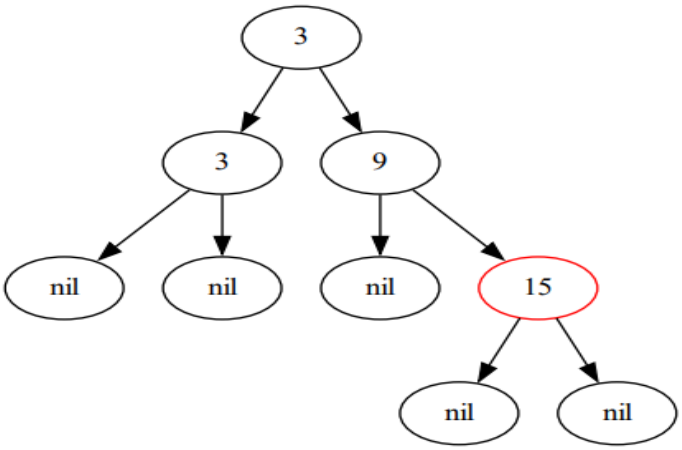

Рис. №11. Добавление узла «15» к дереву.

ЗАКЛЮЧЕНИЕ

В ходе выполнения данной курсовой работы была реализована программа, визуализирующая работу алгоритмов поиска, вставки и удаления элементов в структуре данных красно-черное дерево. Было изучено, что красно-черное дерево является самобалансирующимся деревом поиска, которое гарантирует выполнение операций поиска, вставки и удаления элемента за время O(log(n)), были изучены свойства красно-черного дерева, а также левый и правый повороты для его балансировки и сохранения этих свойств.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- Кафедральный репозиторий с примерами структур данных
  Ссылка: https://github.com/moevm/algorithms_code
- Wikipedia (статья о красно-черном дереве)
  Ссылка: https://ru.wikipedia.org/wiki/Красно-чёрное_дерево
- Алгоритмы. Построение и анализ - Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн
- Сайт для демонстрации RB
  Ссылка: https://www.cs.usfca.edu/~galles/visualization/redblack.html
- Википедия ИТМО
  Ссылка: https://neerc.ifmo.ru/wiki/index.php?title=Красно-чёрное_дерево_(удалить)

ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД ПРОГРАММЫ.

Название файла: algos_cw.py

```
import graphviz

BLACK = 'black'
RED = 'red'


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

    def __str__(self):
        left = self.left.item if self.left else None
        right = self.right.item if self.right else None
        parent = self.parent.item if self.parent else None
        return 'key: {}, left: {}, right: {}, color: {}, parent:
{}'.format(self.item, left, right, self.color, parent)


class RBTree:
    def __init__(self):
        self.nil = Node("nil")
        self.nil.color = BLACK
        self.root = self.nil

    def search_tree_helper(self, node, key):
        if node == self.nil or key == node.item:  # если элемента нет или
элемент найден то возвращем его
            return node

        if key < node.item:  # иначе рекурсивно спускаемся вниз и ищем
его
            return self.search_tree_helper(node.left, key)
        return self.search_tree_helper(node.right, key)

    def delete_fix(self, x):
        while x != self.root and x.color == BLACK:
            if x == x.parent.left:  # если x является левым ребенком
                s = x.parent.right  # s - брат x
                if s.color == RED:  # меняем цвет брата и его родителя на
черный
                    s.color = BLACK
                    x.parent.color = RED
```

```
                    self.left_rotate(x.parent)  # выполняется левый
поворот с родителем x
                    s = x.parent.right

                if s.left.color == BLACK and s.right.color == BLACK:  #
если дети брата черные, то меняем цвет брата
                    # на красный, чтобы свойство выполнялось
                    s.color = RED
                    x = x.parent
                else:
                    if s.right.color == BLACK:
                        # если сын брата черный, то левого сына красим в
красный, производим
                        # правый поворот и обратно присваиваем переменной
s брата x
                        s.left.color = BLACK
                        s.color = RED
                        self.right_rotate(s)
                        s = x.parent.right

                    # присваиваем соответствующие цвета и выполняем левый
поворот
                    s.color = x.parent.color
                    x.parent.color = BLACK
                    s.right.color = BLACK
                    self.left_rotate(x.parent)
                    x = self.root
            else:  # тоже самое симметрично выполняем если x является
правым ребенком
                s = x.parent.left
                if s.color == RED:
                    s.color = BLACK
                    x.parent.color = RED
                    self.right_rotate(x.parent)
                    s = x.parent.left

                if s.right.color == BLACK and s.right.color == BLACK:
                    s.color = RED
                    x = x.parent
                else:
                    if s.left.color == BLACK:
                        s.right.color = BLACK
                        s.color = RED
                        self.left_rotate(s)
                        s = x.parent.left

                    s.color = x.parent.color
                    x.parent.color = BLACK
                    s.left.color = BLACK
                    self.right_rotate(x.parent)
```

```
                    x = self.root
        x.color = BLACK

    def __rb_transplant(self, node, child):
        if node.parent is None:
            self.root = child

        elif node == node.parent.left:
            node.parent.left = child

        else:
            node.parent.right = child
        child.parent = node.parent

    def minimum(self, node):
        # поиск самого нижнего левого элемента
        while node.left != self.nil:
            node = node.left
        return node

    def delete_node_helper(self, node, key):
        z = self.nil
        while node != self.nil:  # спускаемся от корня вниз пока не
найдем элемент с нужным ключом
            if node.item == key:
                z = node

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

        if z == self.nil:  # случай для отсутствия узла с данным ключом в
дереве
            print("Cannot find key in the tree")
            return

        y = z
        y_original_color = y.color
        if z.left == self.nil:  # если отсутсвует у найденного элемента
левый ребенок
            x = z.right  # текущий правый ребенок
            self.__rb_transplant(z, z.right)  # производим замены
указателей между нужными узлами

        elif z.right == self.nil:  # если отсутсвует у найденного
элемента правый ребенок
            x = z.left  # текущий левый ребенок
            self.__rb_transplant(z, z.left)  # производим замены
указателей между нужными узлами
```

```
        else:  # если все дети на месте
            y = self.minimum(z.right)  # ползем вниз и ищем самый нижний
левый элемент
            y_original_color = y.color
            x = y.right
            if y.parent == z:  # если родитель мин элемент совпадает с
нужным
                x.parent = y  # присваиваем листу на место родителя y
            else:
                self.__rb_transplant(y, y.right)  # иначе замена сына мин
элемента и мин элемента
                y.right = z.right
                y.right.parent = y

            self.__rb_transplant(z, y)
            y.left = z.left
            y.left.parent = y
            y.color = z.color

        if y_original_color == BLACK:
            # Исходный цвет элемента для удаления был черным; необходимо
сбалансировать дерево после удаления
            self.delete_fix(x)

    def rb_insert_fix(self, k):
        # если родитель нового узла  не черный, необходима балансировка
(2 случая)
        while k.parent.color == RED:  # цикл выполняется пока родитель не
станет черным
            if k.parent == k.parent.parent.right:
                u = k.parent.parent.left
                if u.color == RED:  # 1) дядя узла красный
                    u.color = BLACK
                    k.parent.color = BLACK
                    k.parent.parent.color = RED
                    k = k.parent.parent
                else:  # 2) дядя узла черный
                    if k == k.parent.left:  # Проверяем, не нарушена ли
балансировка для "Деда"
                        k = k.parent
                        self.right_rotate(k)
                    k.parent.color = BLACK
                    k.parent.parent.color = RED
                    self.left_rotate(k.parent.parent)
            else:  # те же самые проверки, если родитель является правым
потомком
                u = k.parent.parent.right

                if u.color == RED:
```

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

def search_tree(self, k):
    return self.search_tree_helper(self.root, k)

def delete_node(self, item):
    self.delete_node_helper(self.root, item)

def left_rotate(self, x):
    y = x.right
    x.right = y.left
    if y.left != self.nil:
        y.left.parent = x

    y.parent = x.parent
    if x.parent is None:
        self.root = y
    elif x == x.parent.left:
        x.parent.left = y
    else:
        x.parent.right = y
    y.left = x
    x.parent = y

def right_rotate(self, x):
    y = x.left
    x.left = y.right
    if y.right != self.nil:
        y.right.parent = x

    y.parent = x.parent
    if x.parent is None:
        self.root = y
    elif x == x.parent.right:
        x.parent.right = y
    else:
        x.parent.left = y
```

```
        y.right = x
        x.parent = y

    def rb_insert(self, key):
        node = Node(key)  # создаём новый узел красного цвета
        node.parent = None
        node.item = key
        node.left = self.nil
        node.right = self.nil
        node.color = RED

        y = None
        x = self.root

        while x != self.nil:  # спускаемся от корня до фиктивного листа,
куда будет вставлен элемент
            y = x
            if node.item < x.item:
                x = x.left
            else:
                x = x.right
        # к данному моменту y - является родителем для нового узла
        node.parent = y
        if y is None:  # случай если дерево пустое
            self.root = node
        # далее( если дерево не пустое) сравниваем значения узлов для
реализации бинарного дерева
        elif node.item < y.item:
            y.left = node
        else:
            y.right = node

        if node.parent is None:  # корень окрашен в черный цвет
            node.color = BLACK
            return

        if node.parent.parent is None:
            return

        self.rb_insert_fix(node)  # метод для восстановления сво-в RB
tree

    def showing(self, index, tree):
        root = tree.root
        queue = [root]
        graph = graphviz.Digraph(directory="Result_files")
        cnt_nil = 0

        graph.node(str(id(root)), label=str(root.item), color=BLACK)
        while queue:
```

```
            tmp_queue = []
            for element in queue:
                node_id = str(id(element))
                if element != self.nil:
                    id_element = str(id(element))
                    if element.left != self.nil:
                        graph.node(str(id(element.left)),
label=str(element.left.item), color=element.left.color)
                        graph.edge(node_id, str(id(element.left)))
                        tmp_queue.append(element.left)

                    else:
                        cnt_nil += 1
                        graph.node(str(cnt_nil), label="nil",
color=BLACK)
                        graph.edge(id_element, str(cnt_nil))

                    if element.right != self.nil:
                        graph.node(str(id(element.right)),
label=str(element.right.item), color=element.right.color)
                        graph.edge(node_id, str(id(element.right)))
                        tmp_queue.append(element.right)

                    else:
                        cnt_nil += 1
                        graph.node(str(cnt_nil), label="nil",
color=BLACK)
                        graph.edge(id_element, str(cnt_nil))

            queue = tmp_queue
        name = 'fileout_№_{}'.format(index)
        name = name
        graph.render(name)


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

    RBtree.rb_insert(3)
    RBtree.showing(2, RBtree)

    for i in range(3, 48, 6):
        RBtree.rb_insert(i)
        RBtree.showing(i, RBtree)

    RBtree.delete_node(3)
    RBtree.showing(48, RBtree)
    RBtree.delete_node(15)
    RBtree.showing(49, RBtree)
```