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: RB-дерево vs Хеш-таблица (открытая адресация). Исследование.

Студент гр. 1303 _____ Коренев Д. А.

Преподаватель _____ Иванов Д. В.

Санкт-Петербург

2022

ЗАДАНИЕ

НА КУРСОВУЮ РАБОТУ

Студент Коренев Данил Алексеевич

Группа 1303

Тема работы: RB-дерево vs Хеш-таблица (открытая адресация). Исследование.

Исходные данные: Вариант 10 RB-дерево vs Хеш-таблица (открытая адресация). Исследование.

Задание: "Исследование" - 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.

Студент гр. 1303 _____ Коренев Д. А.

Преподаватель _____ Иванов Д. В.

АННОТАЦИЯ

В курсовой работе реализована программа, посредством которой было проведено исследование таких структур данных как RB-дерево и Хеш-таблица.

На основе алгоритмов вставки, поиска и удаления элементов в лучшем, среднем и худших случаях входных данных была проанализирована сложность каждого операции в обоих структурах. Выводы были сделаны путём сравнения полученных, в ходе работы программы, данных и теоретических значений.

Реализация всех алгоритмов написана на Python3. Оценка сложности в каждом случае основывается на графиках зависимости числа элементов в структуре от времени выполнения той или иной операции.

Исходный код см. в приложении А.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель работы: исследование RB-дерева и хэш-таблицы с открытой адресацией. Генерация входных данных (вид входных данных определеяется студентом). Использование входных данных для измерения количественных характеристик структур данных, алгоритмов, действий. Сравнение экспериментальных результатов с теоретическими.

1) Теоретические сведения.

Хеш-таблица

Хеш-таблица — это структура данных, в которой все элементы хранятся в виде пары ключ-значение, где:

• Ключ — уникальное число, которое используется для индексации значений;

• Значение — это данные, которые с этим ключом связаны.

В хеш-таблице обработка новых индексов производится при помощи ключей. А элементы, связанные с этим ключом, сохраняются в индексе. Этот процесс называется хешированием. Пусть k — ключ, а h(x) — Хеш-функция. Тогда h(k) в результате даст индекс, в котором мы будем хранить элемент, связанный с k.

Когда хеш-функция генерирует один индекс для нескольких ключей, возникает конфликт (неизвестно, какое значение нужно сохранить в этом индексе). Это называется коллизией Хеш-таблицы.

Есть несколько методов борьбы с коллизиями:

• Метод цепочек.

• Метод открытой адресации: линейное, квадратичное, двойное хэширование.

При линейном пробировании выбирается шаг q. При попытке добавить элемент в занятую ячейку i начинаем последовательно просматривать ячейки i+(1·q), i+(2·q), i+(3·q) и так далее, пока не найдется свободная ячейка. В неё записывается элемент. При квадратичном пробировании шаг q не фиксирован, а изменяется квадратично: q=1,4,9,16… Соответственно при попытке добавить элемент в занятую ячейку i начинаем последовательно просматривать ячейки i+1,i+4,i+9 и так далее, пока не найдется свободная ячейка. Двойное хеширование, основанное на функции: H(k, i) = (H1(k) + i * H2(k)) mod m (где H1 и H2 – независящие друг от друга хеш-функции).

Таким образом, операции вставки, удаления и поиска в лучшем случае выполняются за O(1), в худшем — за O(n), где n – число элементов. В среднем, при грамотном выборе хеш-функций, двойное хеширование будет выдавать O (a), где a – число возникших, ходе работы, коллизий.

RB-дерево

Красно-чёрное дерево — двоичное дерево поиска, в котором каждый узел имеет атрибут цвета. При этом:

• Узел может быть либо красным, либо чёрным и имеет двух потомков;

• Корень — как правило чёрный. Это правило слабо влияет на работоспособность модели, так как цвет корня всегда можно изменить с красного на чёрный;

• Все листья, не содержащие данных — чёрные.

• Оба потомка каждого красного узла — чёрные.

• Любой простой путь от узла-предка до листового узла-потомка содержит одинаковое число чёрных узлов.

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

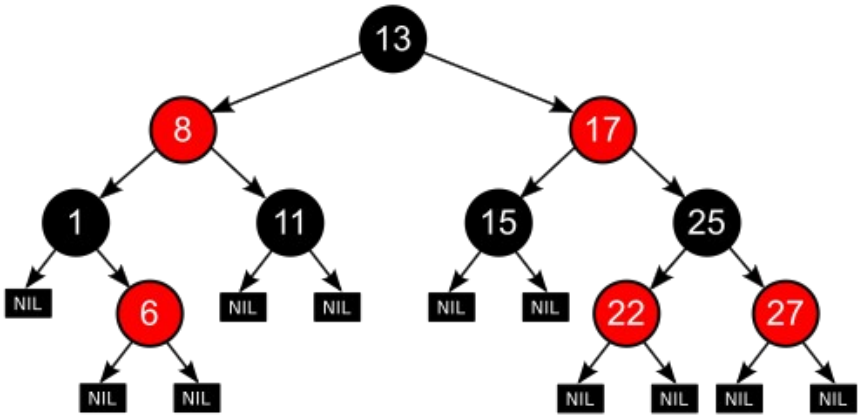

Рис. 1 - пример красно-чёрного дерева

2) Реализация.

Хеш таблица

Для реализации хеш таблицы был создан класс HashTable принимающий в качестве аргументов тип пробирования и коэффициент заполняемости, при котором необходимо расширять таблицу.

Метод __setitem__ перегружен, он принимает ключ и значение, вставляет элемент на нужное место. Для этого рассчитывалось хэш значение от ключа с помощью метода __hash, брался остаток от деления на размер таблицы. Далее начинался безусловный цикл while пока не получистя вставить элемент в свободное место либо обновить значение лежащее по этому ключу. Для этого если место по индексу, полученному от хеш функции, занято к нему прибавлялось смещение. Если было выбрано линейное пробирование, то каждую новую попытку вставки индекс смещался на 1, если квадратичное — брался следующий квадрат, если двойное хеширование прибавлялось новое значение второй хеш функции - __hesh_second. После успешной вставки счетчик количества элементов увеличивался на 1, делается проверка на превышение коэффициента заполняемости, и в случаи необходимости выполняется метод __resize().

Метод __resize() расширяет таблицу в 2 раза. Для этого он создает новый массив пар ключ — значение по умолчанию равными None, «пробегается» по всем элементам старого массива и при нахождении нового элемента вставляет его уже в новую таблицу, заново рассчитывая хеш ключей.

Метод __getitem__ тоже перегружен. Он возвращает значение лежащее по ключу. Для этого рассчитывается хеш от ключа, возвращается значение лежащее по индексу, если ключи совпали, иначе — произошла коллизия, и в зависимости от выбора проибрования идет поиск нужного элемента (это происходит аналогично вставке элемента). Если количество попыток равно размеру таблицы процесс поиска заканчивается, так как каждый элемент был

просмотрен и не найден необходимый, пробрасывается ошибка отсутствия ключа в таблице.

Метод remove получает ключ и удаляет значение лежащее по этому ключу. Работает аналогично методу __getitem__ только при нахождении элемента он заменяется на пару (None, None) что эквивалентно отсутствию элемента в таблице.

Метод __hash принимает ключ для которого необходимо рассчитать хеш значение. Переменная hash_code накапливает значение, в цикле for пробегающем по длине ключа символьное представление элемента ключа с конца умножается на 19 в степени номера итерации. Возвращается значение по модулю размера таблицы.

Метод __hash_second используется для пробирования двойным хешированием. Переменная x равна 263, каждую итерацию она умножается на номер элемента ключа (по которому идет итерация), в промежуточную сумму прибавляется символьное представление элемента умноженное на x, берется остаток от деления на размер таблицы во избежании больших чисел, замедляющие работу. Для того чтобы не было зацикливания про поиске элемента он должен возвращает взаимопростое число с размеров таблицы, для этого высчитаный хеш делится с остатком на размер таблицы без единицы, и прибавляется 1. Так как размер таблицы — степени двойки, то это гарантирует взаимопростого хеша.

RB-дерево

Для реализации RB-дерева были созданы классы RBTree и Node.

Элемент дерева — Node — харнит значения хеш ключа, сам ключ, данные, цвет, указатели на родителя и сыновей (по умолчанию None). Также имеет метод хеширования ключа.

Для вставки элемента в дерево реализован метод insert, принимающий ключ и значение. Создается Node для этих данных и находится место куда нужно вставить этот элемент сравнивая хеши других узлов, если он меньше —

то идет в левое поддерево иначе — в правое, аналогично бинарному дереву поиска. Так как красно-черное дерево — самобалансирующее, то при необходимости вызывается метод __fixNodeInsert__ принимающее в качестве аргумента вставленный узел.

Метод __fix_node_insert__ баллансирует дерево после вставки восстанавливая свойства дерева. Опишу случай, когда родитель вставленного узла красного цвета, так как в обратном случае все действия аналогичные, но симметричные относительно левой и правой сторон. Находится дядя узла и если он красного цвета, то дядя, родитель и дедушка узла перекрашиваются в инвертированные цвета, таким образом восстанавливаются свойства дерева. В противном случае сначала проверяется условие, если ребенок левый, происходит правый поворот относительно его родителя, переменная node переключается на своего родителя. После этого, даже если узел был правым, родитель узла перекрашивается в черный, дедушка в красный, выполняется левый поворот от дедушки узла. Этот алгоритм выполняется пока родитель узла красный или пока цикл не дойдет до узла корня. Корень дерева необходимо перекрасить в черный цвет, так как это является свойством красно-черного дерева.

Опишу метод LR (левы поворот), использующийся в методах __fix_node_insert__ и __fix_remove__. Метод RR (правый поворот) аналогичен, но симметричен относительно правой и левой сторон. Он принимает узел, создает переменную хранящую правого ребенка этого узла и меняет отношения между узлами так, чтобы элемент и его родитель поднялись на уровень выше, а дедушка и дядя — опустились.

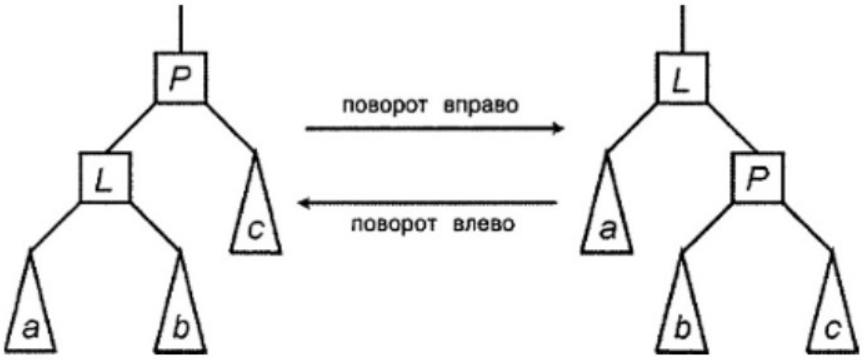

Рис. 2 - повороты

Метод delete предназначен для удаления элемента. Для этого он принимает ключ и вызывает вспомогательный метод __remove и передает ему корень дерева и ключ удаляемого элемента. Он ищет узел, который необходимо удалить сравнивая хеш значения. И присваивает его и его цвет в переменные contain_node и contaion_color соответственно. Далее рассматриваются случаи, когда у элемента нет левого ребенка, нет правого ребенка, есть оба ребенка. В первых двух случаях вызывается метод __transplant для узла и его существующего ребенка, который заменяет первый элемент на второй, при этом удаляя его. В третьем случае находится минимальный элемент в правом поддереве (он гарантированно больше узла), он и его цвет присваиваются в переменные contain_node и contaion_color соответственно. Этот узел меняется местами с удаляемым узлом, который потом удаляется. Далее идет балансировка дерева с помощью метода __fix_remove__.

Метод __fix_remove__ балансирует дерево после удаления узла. Цикл while выполняется пока узел не является корнем и имеет черный цвет. Далее выполняются аналогичные алгоритмы если узел левый или правый ребенок, отличающиеся только симметрией относительно правой и левой стороны. Если узел оказался левым, сначала запоминается брат узла в переменную contain_node. Если он красный, перекраиваются брат в черный, а родитель в красный, выполняется левый поворот относительно родителя, contain_node указывает на бывшего родителя. Далее возможны два случая: оба ребенка

черные или нет. В первом случае родитель этих узлов перекрашивается в красный, а в переменную tmp_node запоминается его родитель. Во втором случае Если цвет правого ребенка у брата черный, левый ребенок перекрашивается в черный, в переменную contain_node записывается красный цвет, выполняется правый поворот от брата, запоминается правый сын узла. Далее родитель и ребенок перекрашиваются, выполняется левый поворот относительно родителя узла. В переменную tmp_node присваивается корень дерева, поэтому процесс балансировки закончился.

Для наглядности был реализован метод print печатающий дерево в терминал. Для этого он вызывает метод __print_node, который рекурсивно печатает узлы с необходимым отступом справа, вызывает себя же для левого и правого ребенка.

3) Оценка сложности.

Для того иллюстрирования зависимости времени работы каждой операции (вставка, поиск, удаление) от количества элементов в структуре использовалась библиотека Matplotlib для визуализации данных двумерной графикой на языке программирования Python.

4) Сравнение теоретических и полученных в результате исследования данных о сложности работы алгоритмов.

4.1. Хеш-таблица

Операция	Лучший	Средний	Худший
Вставка	O(1)	O(a)	O(n)
Поиск	O(1)	O(a)	O(n)
Удаление	O(1)	O(a)	O(n)

Где n — количество элементов в таблице, a — количество возникших коллизий.

- В качестве лучшего случая на вход поступал массив ключей с разными хеш-значениями

- В качестве среднего случая на вход поступал массив псевдослучайных ключей среди которых наблюдались коллизии
- В качестве худшего случая на вход поступал массив ключей с одинаковыми хэш-значениями.

График зависимости времени добавления n-го количества элементов в хеш таблицу в лучшем случае представлен на рисунке 3.

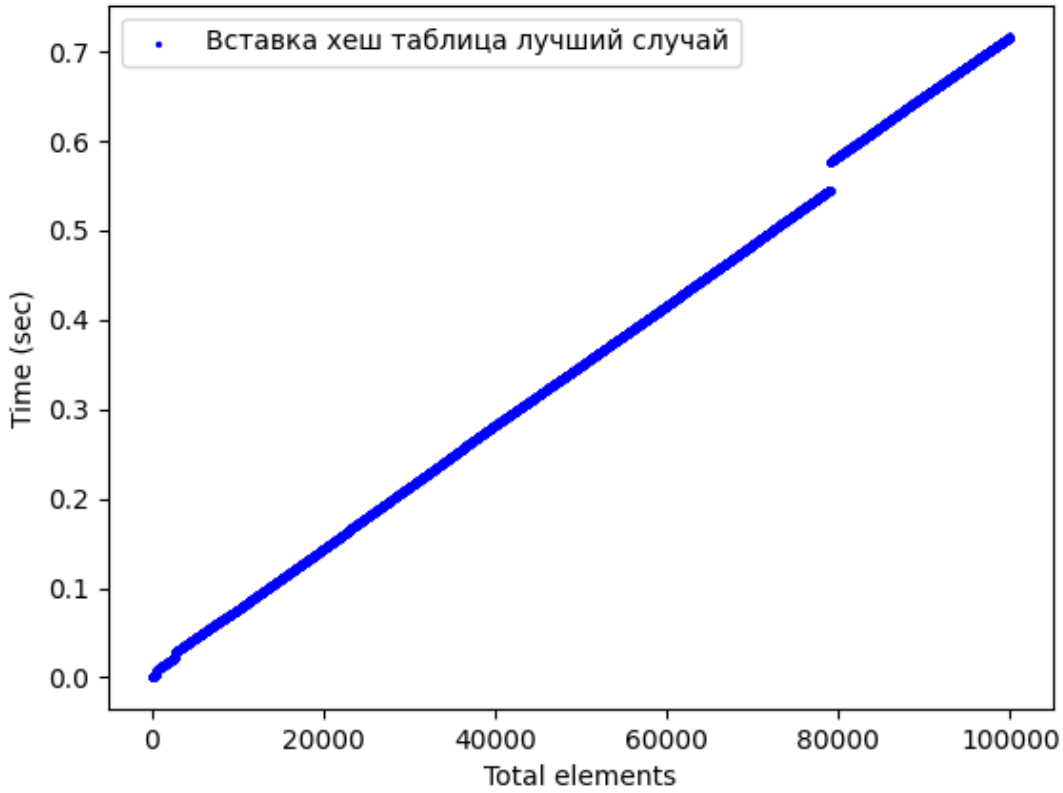

Рис. 3 - график вставки элементов в лучшем случае

Виден линейный рост времени от количества элементов, что соответствует теоретическим значениям.

График зависимости времени добавления n-го количества элементов в хеш таблицу в среднем случае представлен на рисунке 4.

Рис. 4 - график вставки элементов в среднем случае

В данном случае тоже линейный рост времени от количества элементов, что соответствует теоретическим значениям.

График зависимости времени добавления n-го количества элементов в хеш таблицу в худшем случае представлен на рисунке 5.

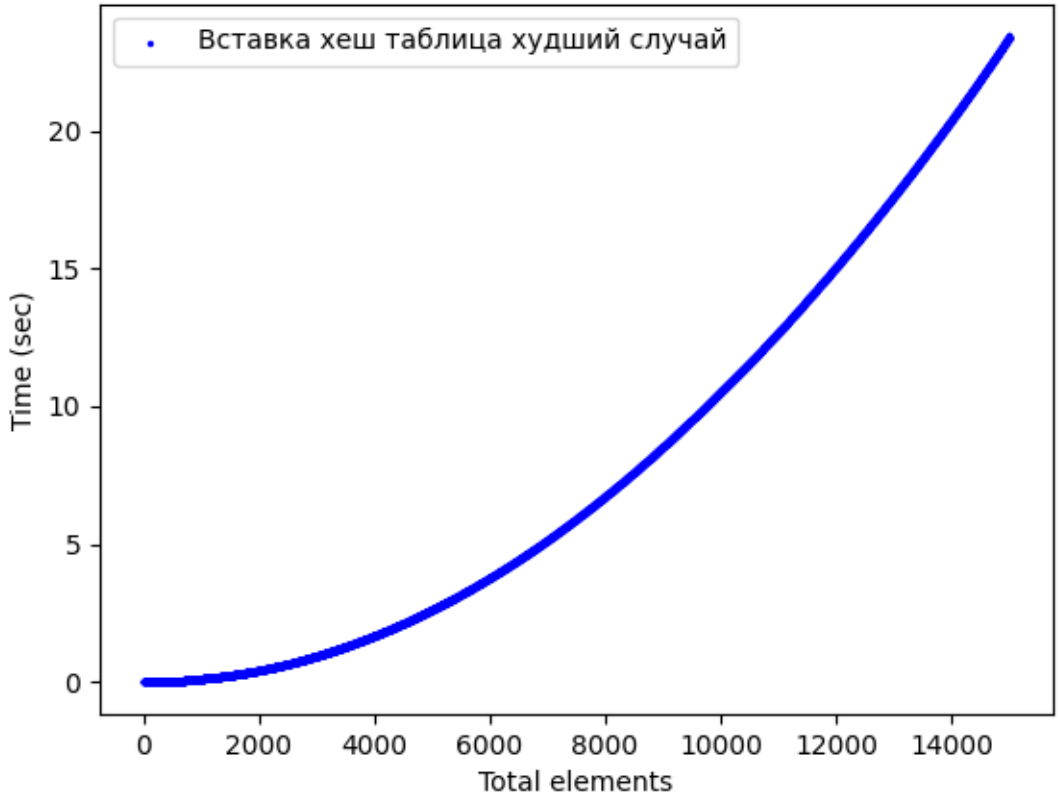

Рис. 5 - график вставки элементов в худшем случае

В данном случае виден экспоненциальный рост времени от числа добавляемых элементов, так как для каждого нового n-го элемента необходимо сделать n проверок. Данные соответствуют теоретическим.

Так как лучший и худший случаи не отличаются идеологически при реализации разных способов пробирования, то достаточно сравнить средний случай разных для разных типов. График зависимости времени добавления n-го количества элементов в хеш таблицу в среднем случае представлен на рисунке 6, где красный график — линейная пробирование, зеленый — квадратичное, синий — двойное хеширование.

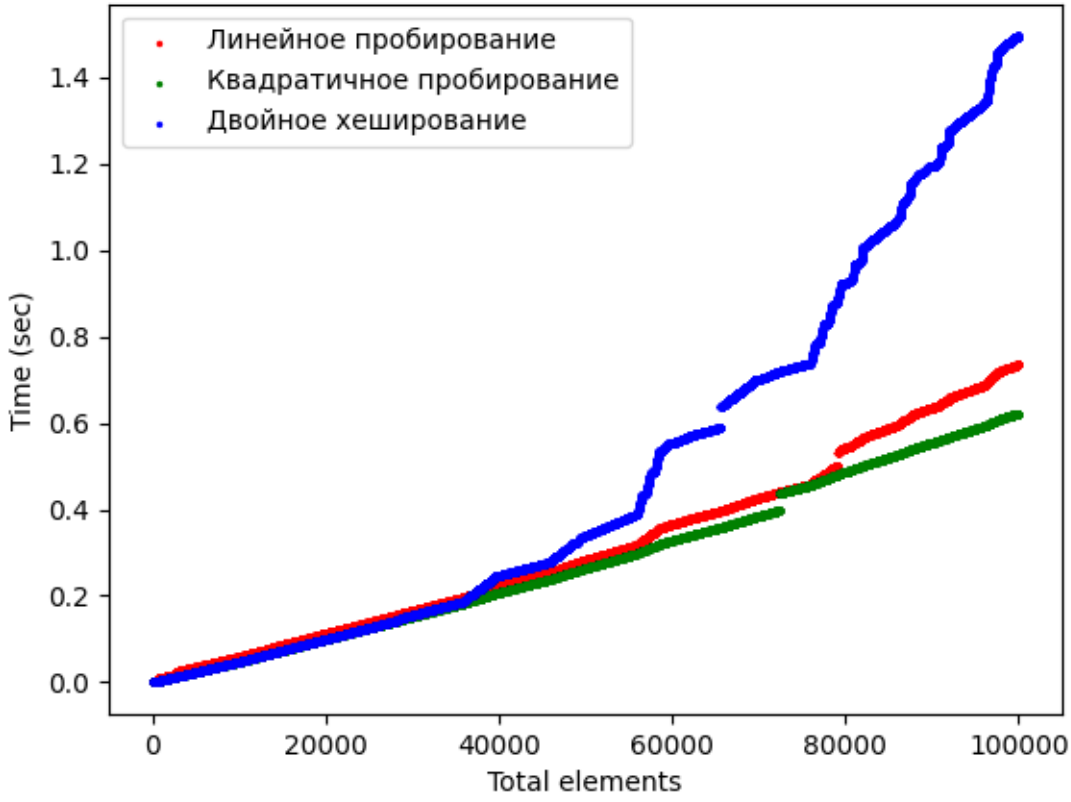

Рис. 6: график вставки элемента в среднем случае для разных типов пробирования

По графику видно, что квадратичный метод пробирования оптимальный, а метод двойного хеширования самый медленный. Это связано с тем, что вычисление второй хеш функции несколько затратнее чем линейное смещение в линейном методе или возведение в квадрат в случае квадратичного пробирования.

Так же необходимо сравнить скорость удаления элементов из таблиц. Рассмотрю лучший случай — отсутствие коллизий при доступе к элементу, график представлен на рисунке 7.

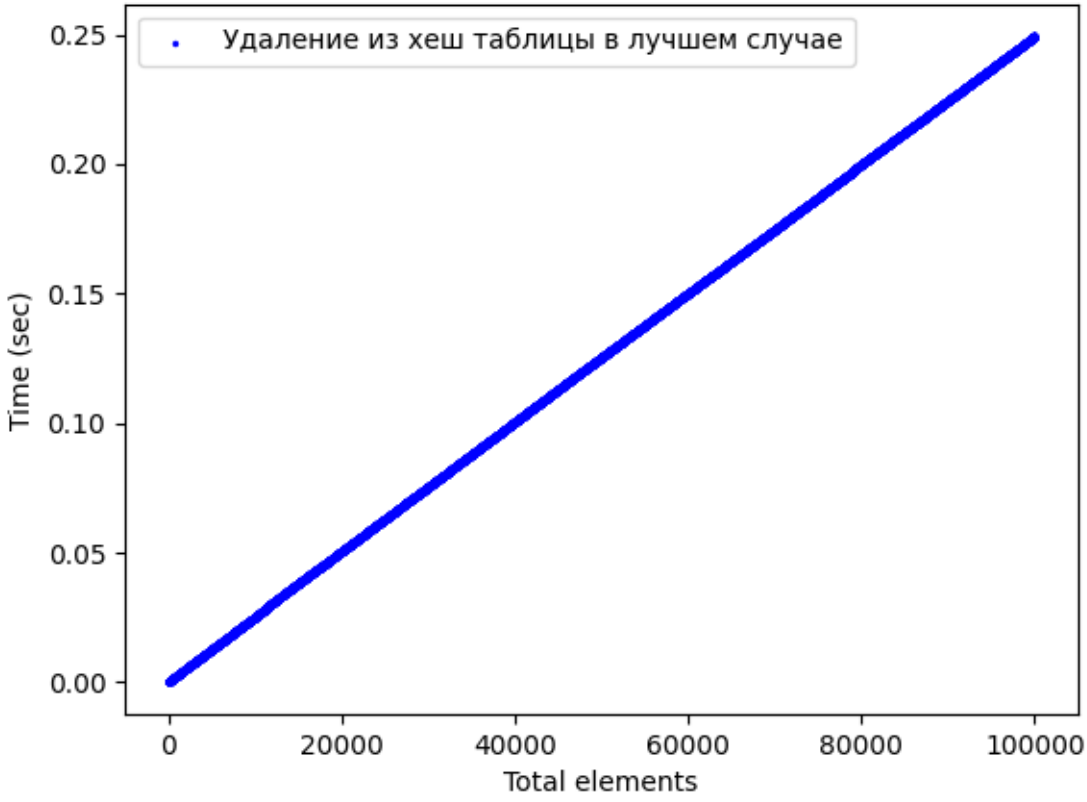

Рис. 7: график удаление элементов в лучшем случае

Виден линейный рост времени от количества элементов, что соответствует теоретическим значениям.

График зависимости времени удаления n-го количества элементов в хеш таблице в худшем случае представлен на рисунке 8.

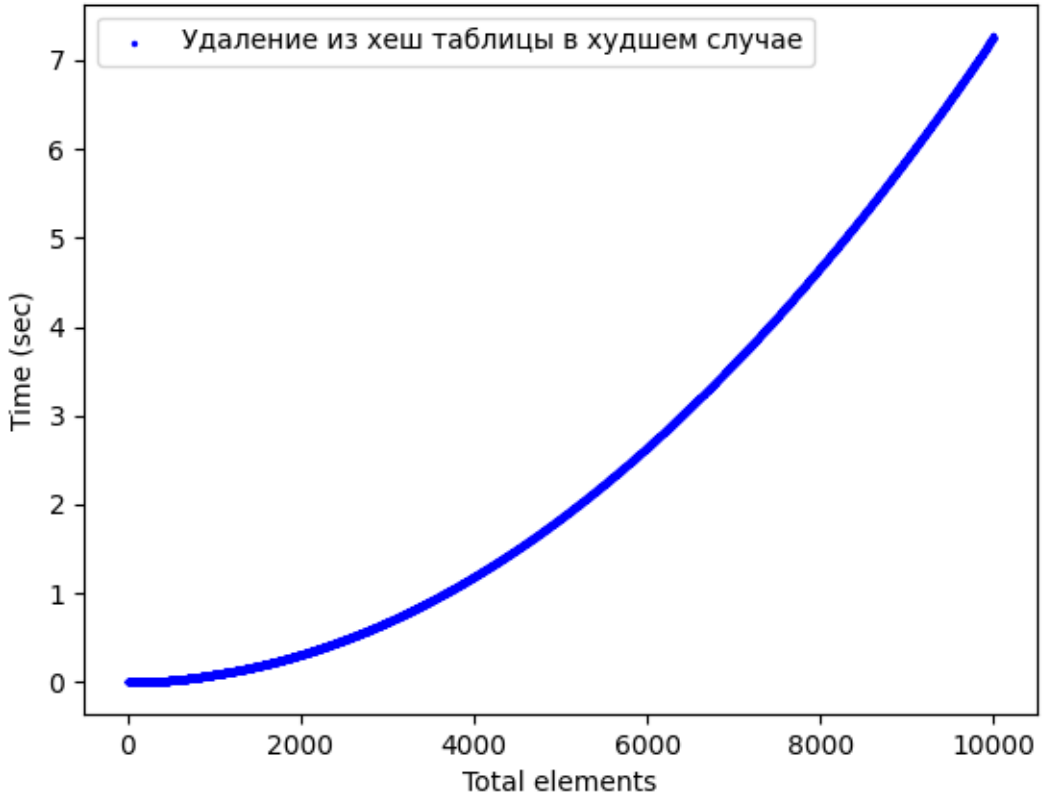

Рис. 8 - график зависимости удаления элементов в худшем случае

Так как лучший и худший случаи не отличаются идеологически при реализации разных способов пробирования, то достаточно сравнить средний случай удаления элементов для разных типов. График зависимости времени удаления n-го количества элементов из хеш таблицы в среднем случае представлен на рисунке 9, где красный график — линейная пробирование, зеленый — квадратичное, синий — двойное хеширование.

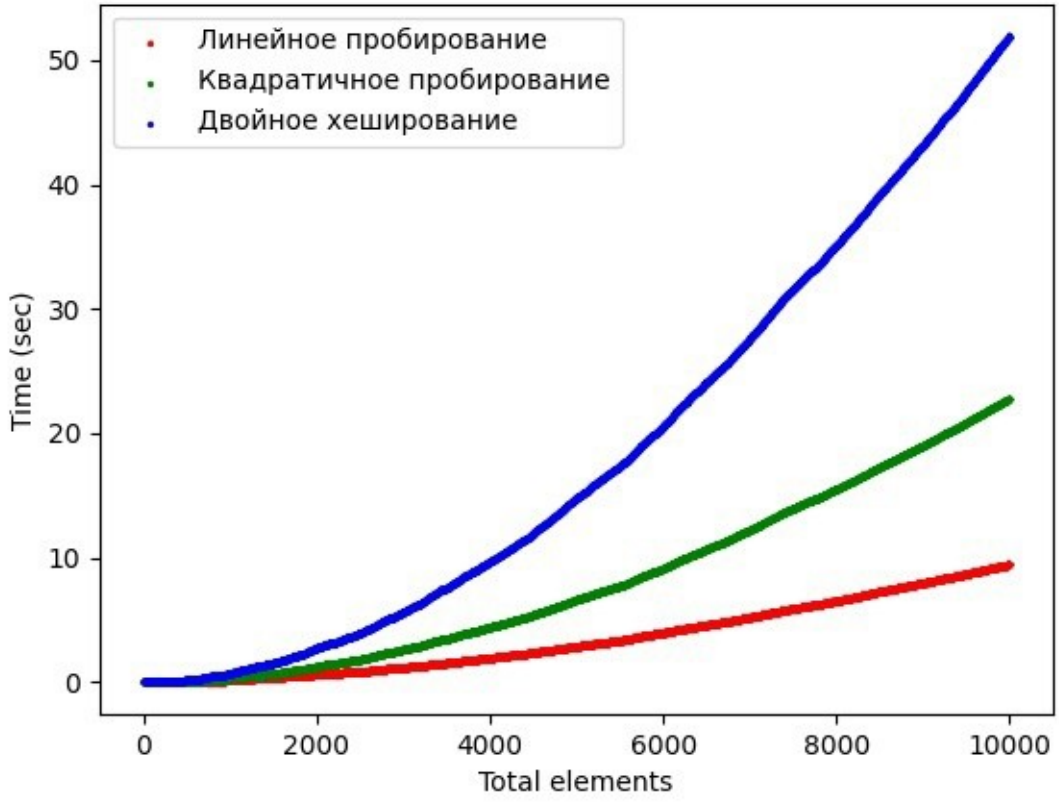

Рис. 9: график удаления элемента в среднем случае для разных типов пробирования

По графику видно, что линейный способ пробирования оптимальный, а метод двойного хеширования самый медленный. Я считаю у этого причина такая же, как и в случае со вставкой.

Графики поиска элемента будут аналогичными с графиками удаления элемента, так как в обоих случаях идет поиск элемента в хэш таблице, а процесс удаления элемента занимает O(1) времени.

Подводу промежуточный итог реализации хеш таблицы. Данные работы программы совпадают с теоретическими, а линейное пробирование самое оптимальное в моей реализации.

4.2. RB-дерево

Операция	Лучший	Средний	Худший
Вставка	O(logn)	O(logn)	O(logn)

Поиск	O(logn)	O(logn)	O(logn)
Удаление	O(logn)	O(logn)	O(logn)

Исследования проводились с аналогичными условиями (входными данными).

График сложности вставки элемента в дерево и в хеш таблицу (линейное пробирование) в среднем случае представлено на рисунке 10, график вставки для rb-дерева — зеленый, хеш таблицы — красный, хеш таблицы с resize — синий.

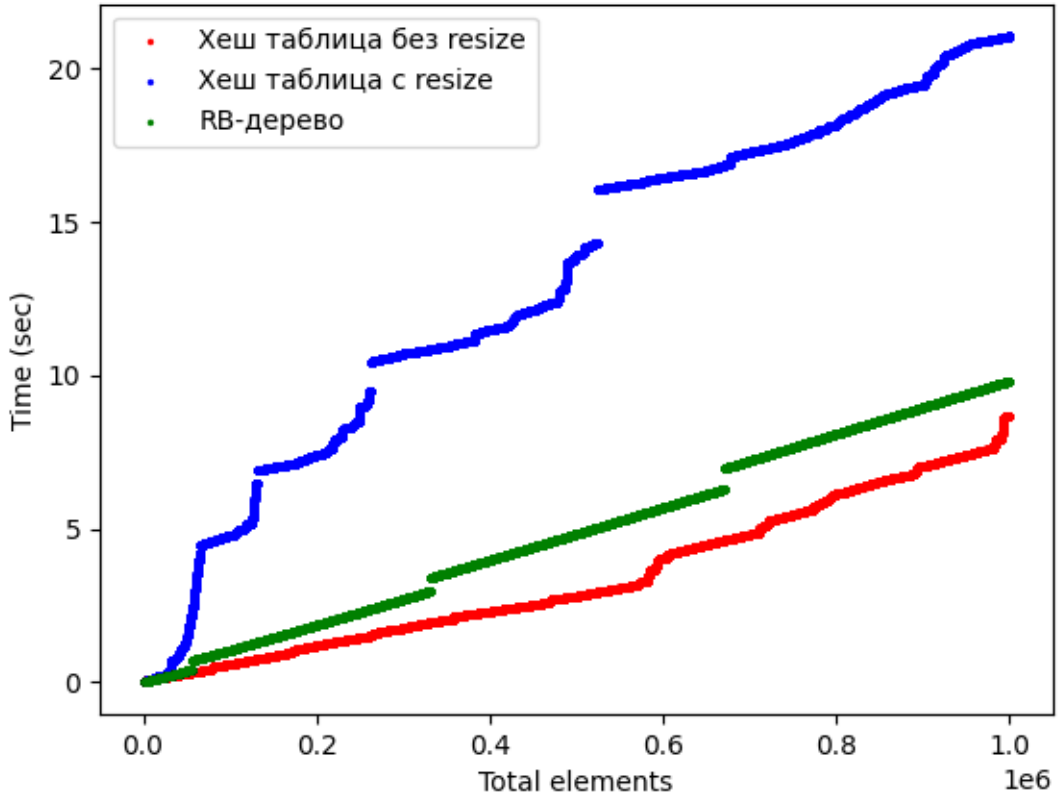

Рис. 10: график вставки элементов в rb-дерево и хеш таблицу

Скорость работы RB-дерева быстрее хеш таблицы, но только теоретически, когда resize таблицы не учитывается. В реальных условиях работы, когда хеш таблица расширяется, rb-дерево оказывается в 2 раза быстрее и стабильнее хеш таблицы.

График сложности удаления элемента из дерева и из хеш таблицы (линейное пробирование) в среднем случае представлено на рисунке 11, график удаления для rb-дерева — зеленый, хеш таблицы — красный.

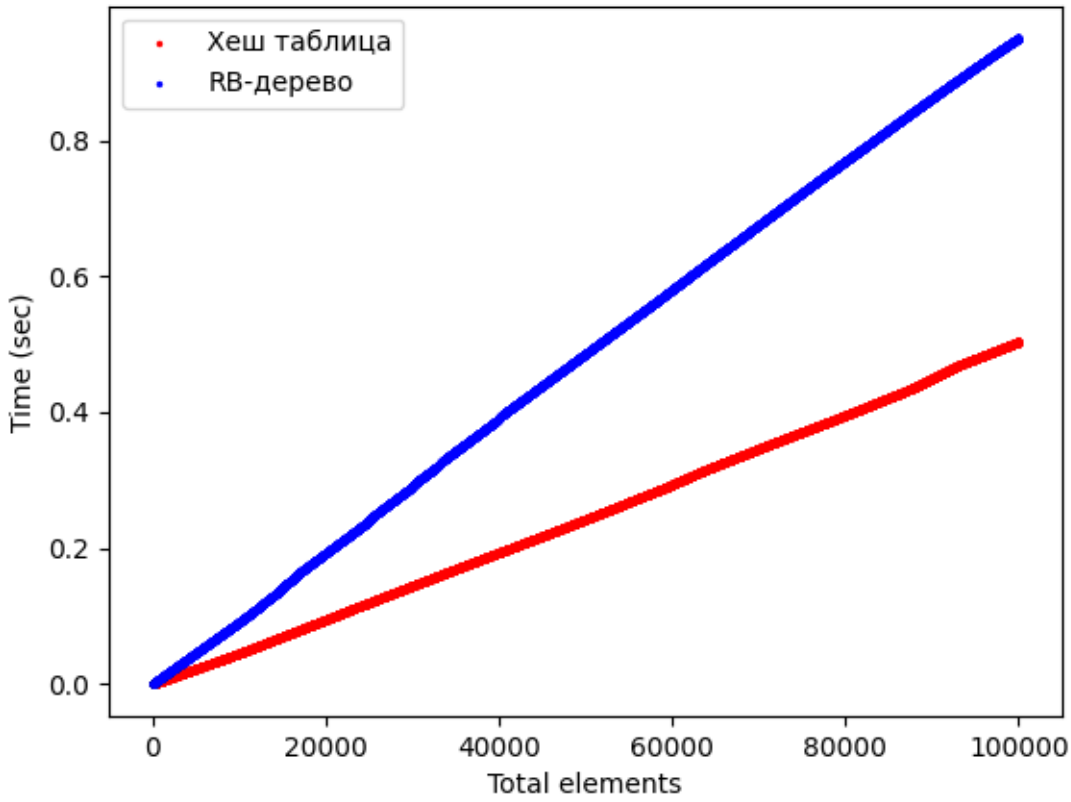

Рис. 11: график удаления элемента из rb-дерева и хеш таблицы

По графику видно, что удаление элементов в rb-дереве медленнее чем из хеш таблицы.

График сложности поиска элемента из дерева и из хеш таблицы (линейное пробирование) в среднем случае представлено на рисунке 12, график удаления для rb-дерева — зеленый, хеш таблицы — красный.

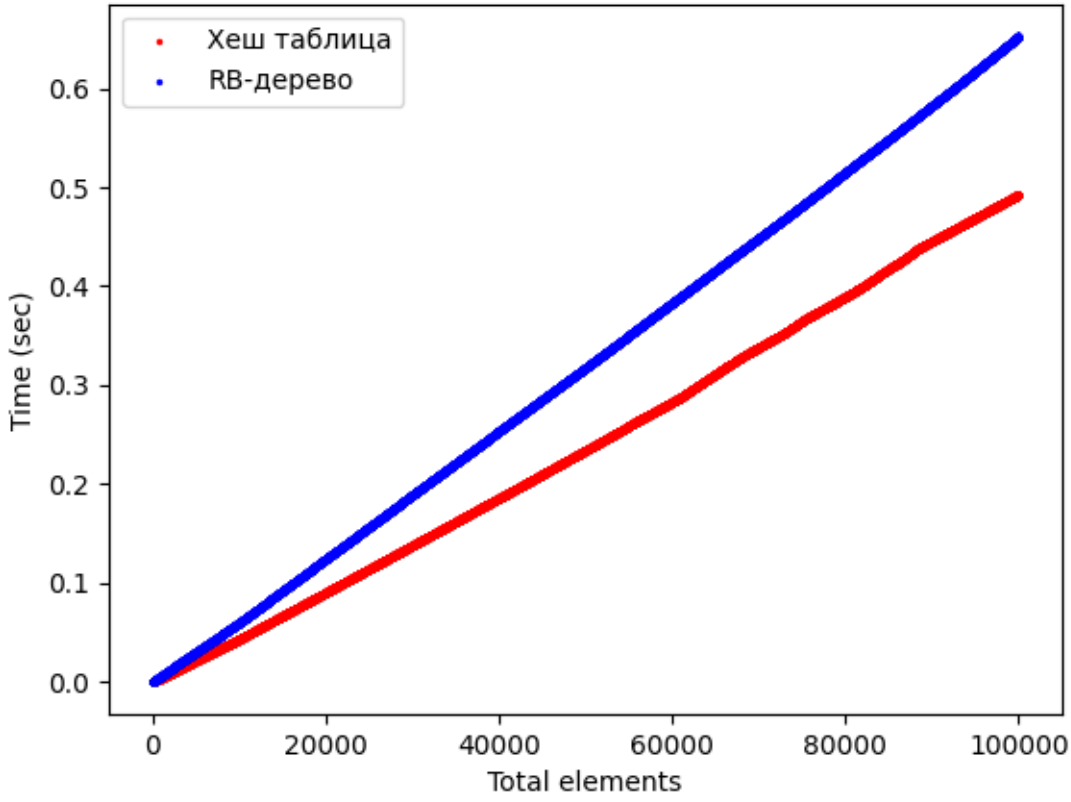

Рис. 12: график поиска элементов в rb-дереве и хеш таблице

По графику можно определить, что доступ к элементу у rb-дерева медленнее, чем у хеш таблицы.

5) Тестирование

Для тестирования создавались экземпляры хеш таблицы, rb-дерева и встроенного в python словаря. Далее в них вставлялись пары ключ значения, доставались по этим же ключам значения и сравнивались. Так как все значение были одинаковыми, то можно сделать вывод, что реализованные структуры работают корректно. Корректность выполнения данной процедуры представлена на рисунке 13.

Рис. 13: корректно работающие структуры данных

6) Вывод

Результаты, полученные путём исследования и реализации структур данных, соответствуют теоретическим значениям сложности работы алгоритмов. Глядя на результаты работы и затрачиваемого времени красно-чёрного дерева и хеш таблицы можно сделать выводы: линейное пробирование лучше во всех случаях по сравнению с другими методами. Добавление элементов в хше таблицу быстрее, чем в красно черное дерево, но поиск — медленнее. Удаление в красно черном дереве работает быстрее чем в хеш таблице до определенного количества элементов.

ЗАКЛЮЧЕНИЕ

В ходе курсовой работы было проведено исследование и выполнена реализация структур данных Хеш-таблицы и RB-дерева. Проведенное промежуточное тестирование показало, что реализация выполнена корректно. Само исследование сложности операций так же дало положительный и приблизительно точный результат (без расхождений с теоретическими значениями).

ИСТОЧНИКИ

1. https://ru.wikipedia.org/wiki/ Красно-чёрное_дерево
2. https://neerc.ifmo.ru/wiki/index.php?title =Красно-черное_дерево
3. http://e.moevm.info/mod/lesson/view.php?id=2048&pageid=1383
4. https://ru.wikipedia.org/wiki/%D0%A5%D0%B5%D1%88-%D1%82%D0%B0%D0%B1%D0%BB%D0%B8%D1%86%D0%B0

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл HashTable.py

```
class HashTable:
    def __init__(self, probing="linear", fill_factor=0.75):
        if 0 < fill_factor < 1:
            self.fill_factor = fill_factor
        else:
            self.fill_factor = 0.75
        self.probing = probing
        self.__current_size = 1048576
        self.__total_item = 0
        self.total_collision = 0
        self.__array = [(None, None) for _ in range(self.__current_size)]

    def __str__(self):
        return self.__array

    def __setitem__(self, key, value):
        # берем хэш от ключа mod размер
        hash_value = self.__hash(key) % self.__current_size
        iteration = 0
        offset = 0
        while True:
            # если на новом месте нет элемента - вставить
             if self.__array[(hash_value + offset) % self.__current_size]
[0] is None:
                # хранится хэш без округления
                self.__array[(hash_value + offset) % self.__current_size]
= [key, value] # хранится пара (ключ - значение)
                break
                            elif  self.__array[(hash_value  +  offset)  %
self.__current_size][0] == key:
                self.__array[(hash_value + offset) % self.__current_size]
[1] = value
                break

            iteration+=1
            if self.probing == "linear":
                    offset = iteration # линейное пробирование
            elif self.probing == "square":
                    offset = iteration**2 # квадратичное пробирование
            elif self.probing == "double hashing":
                                                                  offset   =
iteration*self.__hash_second(hash_value+offset)
            else:
                offset = iteration # по дефолту линейное

        self.__total_item+=1
        if self.__total_item >= self.fill_factor*self.__current_size:
            self.__resize()

        return None
```

```
    def __resize(self):
        new_array = [(None, None) for _ in range(self.__current_size*2)]
        old_array = self.__array
        old_size = self.__current_size
        self.__array = new_array
        self.__current_size*=2

        self.__total_item = 0
        self.total_collision = 0
        for index in range(old_size):
            if old_array[index][0] is not None: # найден элемент который
надо скопировать в новую таблицу
                self.__setitem__(old_array[index][0], old_array[index]
[1])


    def __getitem__(self, key):
        hash_value = self.__hash(key) % self.__current_size
        iteration = 0
        offset = 0
        while True:
            # если на новом месте нет элемента - вставить
            if self.__array[(hash_value + offset) % self.__current_size]
[0] == key:
                return self.__array[(hash_value + offset) %
self.__current_size][1]

            iteration += 1
            if iteration > self.__current_size:
                raise KeyError("No such key in hash-table!")

            if self.probing == "linear":
                offset = iteration  # линейное пробирование
            elif self.probing == "square":
                offset = iteration ** 2  # квадратичное пробирование
            elif self.probing == "double hashing":
                offset = iteration * self.__hash_second(hash_value +
offset)
            else:
                offset = iteration  # по дефолту линейное

    def __str__(self):
        return str(self.__array)

    def remove(self, key):
        hash_value = self.__hash(key) % self.__current_size

        iteration = 0
        offset = 0
        while True:
            # если на новом месте нет элемента - вставить
            if self.__array[(hash_value + offset) % self.__current_size]
[0] == key:
                data = self.__array[(hash_value + offset) %
self.__current_size][1]
```

```
                self.__array[(hash_value + offset) % self.__current_size]
= (None, None)
                return data

            iteration += 1
            if iteration > self.__current_size:
                raise KeyError("No such key in hash-table!")

            if self.probing == "linear":
                offset = iteration  # линейное пробирование
            elif self.probing == "square":
                offset = iteration ** 2  # квадратичное пробирование
            elif self.probing == "double hashing":
                offset = iteration * self.__hash_second(key)
            else:
                offset = iteration  # по дефолту линейное


    def __hash(self, key, size = None):
        if size is None:
            size = self.__current_size
        hash_code = 0
        for i in range(len(key)):
            hash_code += (ord(key[-i-1]) * (19 ** i))
        return hash_code % size

    def __hash_second(self, key):
        x = 263
        tmp_sum = 0
        for k, element in enumerate(str(key)):
            x*= k
            tmp_sum = (ord(element) * x) % self.__current_size

        return (tmp_sum % (self.__current_size - 1)) + 1

    def find_collision(self):
        arr = []
        for i in self.__array:
            if i == (None, None): continue
            hc = self.__hash(i[0])
            if hc not in arr:
                arr.append(hc)
        return len(arr)

    def size(self):
        return self.__total_item
```

Файл Node.py

```
class Node:
      def __init__(self, val, data, color = 1, left=None, right=None,
parent=None):
        self.val = self.hash_key(val)
        self.key = val
        self.data = data
        self.parent = parent
        self.left = left
```

```
        self.right = right
        self.color = color

    @classmethod
    def hash_key(cls, key):
        key = str(key)
        hash_code = 0
        for i in range(len(key)):
            hash_code += (ord(key[-i - 1]) * (19 ** i))
        return hash_code
```

Файл RBTree.py

```
from module.Node import Node
class RBTree:
    NULL = Node(0, None, 1)

    def __init__(self):
        self.root = self.NULL

    def insert(self, key, data):
        node_new = Node(key, data, 1, self.NULL, self.NULL, None)
        tmp_node = None
        node = self.root

        while node != self.NULL:  # поиск места вставки
            tmp_node = node
            if node_new.val < node.val:
                node = node.left
            else:
                node = node.right

           node_new.parent = tmp_node   # tmp_node родитель вставляемого
элемента
        # установить связи между узлами
        if tmp_node is None:
            self.root = node_new
        elif node_new.val < tmp_node.val:
            tmp_node.left = node_new
        else:
            tmp_node.right = node_new

         if node_new.parent is None:  # если вставили в корень, покрасить
в черный
            node_new.color = 0
            return

         if node_new.parent.parent is None:  # если первый ребенка корня,
выход
            return

        self.__fix_node_insert__(node_new)  # фиксить вставку


    def __fix_node_insert__(self, node):
           while node.parent.color == 1:   # пока родитель красный надо
фиксить
```

```
            if node.parent == node.parent.parent.right:  # родитель ноды
- правый
                uncle = node.parent.parent.left  # найти дядю (левый
ребенок деда)
                if uncle.color == 1:  # ситуация, когда дядя - красный ->
надо все перекрасить
                    uncle.color = 0  # дядя -> черный, родитель ->
черный, дед -> красный
                    node.parent.color = 0
                    node.parent.parent.color = 1
                    node = node.parent.parent  # подняться на уровень
деда и продолжить цикл
                else: # если дядя черный
                    if node == node.parent.left:  # ребенок - левый
                        node = node.parent # переключиться на родителя
                        self.RR(node)  # правый поворот относительно
родителя
                    node.parent.color = 0 # отец (вставляемый элемент) ->
черный
                    node.parent.parent.color = 1 # дед (отец
вставляемого) -> красный
                    self.LR(node.parent.parent) # левый поворот от деда
(отца вставляемого)
            else: # все аналогично, но симметрично
                uncle = node.parent.parent.right
                if uncle.color == 1:
                    uncle.color = 0
                    node.parent.color = 0
                    node.parent.parent.color = 1
                    node = node.parent.parent
                else:
                    if node == node.parent.right:
                        node = node.parent
                        self.LR(node)
                    node.parent.color = 0
                    node.parent.parent.color = 1
                    self.RR(node.parent.parent)
            if node == self.root: # добрались до корня -> все дерево
пофикшено -> выход
                break
        self.root.color = 0 # корень всегда черный


    def LR(self, node):
        tmp = node.right
        node.right = tmp.left
        if tmp.left != self.NULL:
            tmp.left.parent = node

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

```
        tmp.left = node
        node.parent = tmp

    def RR(self, node):
        tmp = node.left
        node.left = tmp.right
        if tmp.right != self.NULL:
            tmp.right.parent = node

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

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

    def __remove(self, node, key):
        tmp_node = self.NULL
        hashed_key = Node.hash_key(key)
        while node != self.NULL:
            if node.key == key:
                tmp_node = node
            if node.val <= hashed_key:
                node = node.right
            else:
                node = node.left
        if tmp_node == self.NULL:
            raise KeyError("Node with {} key doesn't exist".format(key))

        contain_node = tmp_node
        contain_color = contain_node.color
        if tmp_node.left == self.NULL:
            child_node = tmp_node.right
            self.__transplant(tmp_node, tmp_node.right)
        elif tmp_node.right == self.NULL:
            child_node = tmp_node.left
            self.__transplant(tmp_node, tmp_node.left)
        else:
            contain_node = self.minimum(tmp_node.right)
            contain_color = contain_node.color
            child_node = contain_node.right
            if contain_node.parent == tmp_node:
                child_node.parent = contain_node
            else:
                self.__transplant(contain_node, contain_node.right)
                contain_node.right = tmp_node.right
                contain_node.right.parent = contain_node
            self.__transplant(tmp_node, contain_node)
            contain_node.left = tmp_node.left
            contain_node.right.parent = contain_node
```

```
            contain_node.color = tmp_node.color
        if contain_color == 0:
            self.__fix_remove__(child_node)

    def __transplant(self, node_1, node_2):
        if node_1.parent is None:
            self.root = node_2
        elif node_1 == node_1.parent.left:
            node_1.parent.left = node_2
        else:
            node_1.parent.right = node_2
        node_2.parent = node_1.parent

    def __fix_remove__(self, tmp_node: Node):
         while tmp_node != self.root and tmp_node.color == 0:  # узел не
корень и черный
            if tmp_node == tmp_node.parent.left: # узел левый ребенок
                contain_node = tmp_node.parent.right  # брат узла
                if contain_node.color == 1:  # брат красный
                    contain_node.color = 0  # брат -> черный
                    tmp_node.parent.color = 1  # родитель -> красный
                              self.LR(tmp_node.parent)   # левый поворот
относительно родителя
                     contain_node = tmp_node.parent.right # указывает на
бывшего отца, но теперь ребенка узла
                                  if  contain_node.left.color  ==  0  and
contain_node.right.color == 0: # оба ребенка красные
                    contain_node.color = 1  # сделать узел красным
                          tmp_node = tmp_node.parent # сместить узел на
родителя
                else:
                     if contain_node.right.color == 0:  # правый ребенок
черный
                         contain_node.left.color = 0  # левый ребенок ->
черный
                        contain_node.color = 1  # сохранение красного
                        self.RR(contain_node)  # правый поворот
                         contain_node = tmp_node.parent.right # запомнить
родителя узла

                            contain_node.color  =  tmp_node.parent.color  #
запомнить цвет родителя
                    tmp_node.parent.color = 0  # установить цвет родителя
черным
                          contain_node.right.color = 0 # установить цвет
правого ребенка запомнившегося узла черным
                    self.LR(tmp_node.parent)  # левый поворот от родителя
узла
                      tmp_node = self.root # дерево сбалансировано, цикл
закончится
            else: # аналогично, но инвертировано
                contain_node = tmp_node.parent.left
                if contain_node.color == 1:
                    contain_node.color = 0
                    tmp_node.parent.color = 1
                    self.RR(tmp_node.parent)
```

```
                    contain_node = tmp_node.parent.left
                                if  contain_node.right.color  ==  0  and
contain_node.right.color == 0:
                    contain_node.color = 1
                    tmp_node = tmp_node.parent
                else:
                    if contain_node.left.color == 0:
                        contain_node.right.color = 0
                        contain_node.color = 1
                        self.LR(contain_node)
                        contain_node = tmp_node.parent.left

                    contain_node.color = tmp_node.parent.color
                    tmp_node.parent.color = 0
                    contain_node.left.color = 0
                    self.RR(tmp_node.parent)
                    tmp_node = self.root
        tmp_node.color = 0


    def get(self, key):
        hashed_key = Node.hash_key(key)
        current_node = self.root
        while current_node:
            if current_node.key == key:
                return current_node.data
            if current_node.val > hashed_key:
                current_node = current_node.left
            else:
                current_node = current_node.right
             raise  KeyError("Node  with  {0}  key  doesn't  exist.  Tree  is
empty".format(key))

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

    def max(self):
        node = self.root
        while node.right != self.NULL:
            node = node.right
        return node.data

    def min(self):
        node = self.root
        while node.left != self.NULL:
            node = node.left
        return node.data

    def print(self) :
        self.__print_node (self.root, "", True)

    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 += " |    "

            s_color = "RED" if node.color == 1 else "BLACK"
             print ("Key <" + str(node.key) + "> data <" + str(node.data)
+ "> (" + s_color + ") " + "hash <" + str(node.val) + ">")
            self.__print_node (node.left, indent, False)
            self.__print_node (node.right, indent, True)
```