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

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

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

Студент гр. 1384 _____ Камынин А. А.

Преподаватель _____ Иванов Д. В.

Санкт-Петербург

2022

ЗАДАНИЕ
НА КУРСОВУЮ РАБОТУ

Студент: Камынин А. А.

Группа 1384

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

Исходные данные:
"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.
Содержание пояснительной записки:
Аннотация, Содержание, Введение, Отчет, Примеры работы программы, Заключение

Предполагаемый объем пояснительной записки:
Не менее 10 страниц.

Дата выдачи задания: 25.10.2022

Дата сдачи реферата: 24.12.2022

Дата защиты реферата: 24.12.2022

Студент _____ Камынин А. А.

Преподаватель _____ Иванов Д. В.

АННОТАЦИЯ

Курсовая работа заключается в реализации таких структур данных, как красно-черное дерево (оно же RB-дерево) и хеш таблицы с открытой адресацией. Необходимо реализовать заданные структуры данных, сгенерировать входные данные, определяемые самостоятельно, для измерения количественных характеристик, сравнение экспериментальных результатов с теоретическими.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью работы является реализация RB-дерева и Хеш-таблицы с открытой адресацией, сравнению данных структур с их теоретическими оценками, а также между собой в следующих операциях:

- Вставка элемента вида ключ-значение
- Удаление элемента по ключу
- Поиск элемента по ключу

1. РЕАЛИЗАЦИЯ СТРУКТУР ДАННЫХ

1.1. Красно-черное дерево (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.

Для реализации красно-черного дерева были написаны два класса на языке Python: Node и RBTree. Класс Node представляет собой узел, элемент дерева, который хранит в себе поля key, value (значение узла), left, right и parent (указатели на левого и правого ребенка, а также на родителя данного узла в дереве) и цвет (красный или черный). Для обозначения цветов были использованы строки, записанные в глобальные переменные BLACK и RED.

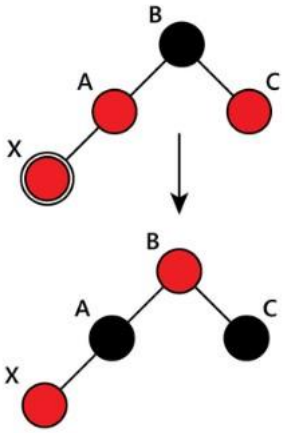
При создании красно-черного дерева указателю на корень присваивается None, количеству неудаленных узлов 0 (данное поле необходимо для проверки корректности удаления в дереве), а фиктивному листу nil, не несущему никакой значительной информации, но тоже являющемся частью красно-черного дерева (необходим для корректного удаления в дереве), присваивается объект класса Node со значением -1 и черным цветом.

Реализация вставки. Для вставки в красно-черное дерево узла с ключом key и значением value были реализованы методы insert(key, value) и fix_insert(node). В первом методе проверяется, не пусто ли текущее дерево (указатель на корень не равен None), и если это так, то узел с переданным

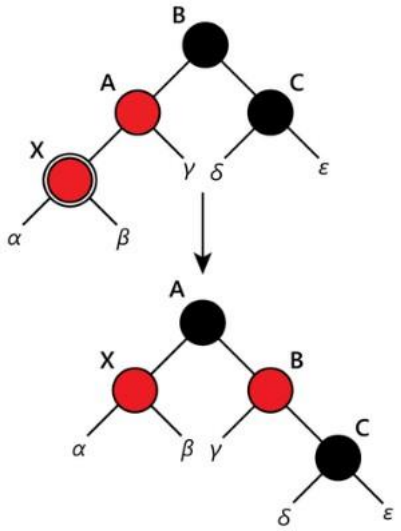
ключом записывается в корень. В ином случае, пользуясь свойством красно-черного дерева, что слева от текущего узла располагаются элементы с меньшим ключом, а справа с большим, ищется подходящий лист для заданного ключа. Таким образом, мы проходимся от корня к нужному листу, тем самым осуществив h сравнений, где h - высота красно-черного дерева. В случае, если мы нашли уже существующий ключ, так как мы хотим поддерживать ассоциативный массив, что значение данного ключа заменяется, при этом вставки не происходит. В ином случае после вставки элемента свойства могли нарушиться, то вызывается метод fix_insert(node), восстанавливающий их. При восстановлении в цикле узел node поднимается каждый раз снизу вверх (если при этом не оказалось, что свойства уже восстановлены), в худшем случае доходя до корня дерева. При восстановлении свойств используется левое и правое малое вращение. Так как после восставления корень может стать красным, то в таком случае он перекрашивается в черный цвет.

Случаи при вставке:

1. Дядя узла красный. Тогда для перекрашиваем отца и дядю в черный цвет, а деда в красный. В таком случае высота в поддереве для всех листьев одинакова, а у всех вершин отцы черные.



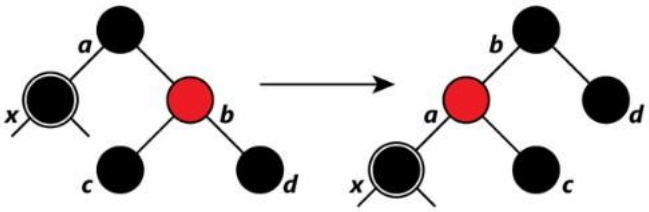
2. Дядя узла черный. Выполняем поворот. Если добавляемый узел был правым потомком, то необходимо выполнить левое вращение, которое сделает его левым потомком.



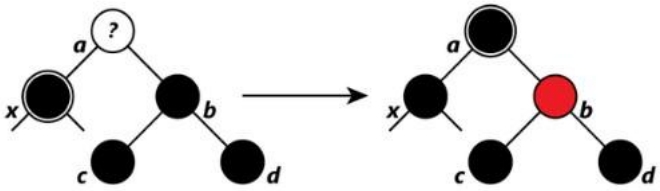
Реализация удаления. Для удаления из красно-черного дерева узла с переданным ключом key были реализованы методы delete(key) и fix_delete(node). В методе delete(key) проходом от корня до листа ищется узел к удалению: если данный узел не найден, об этом выводится сообщение, значение поля count_not_delete_nodes увеличивается на 1; в ином случае записывается указатель на удаляемый элемент в node_to_delete. При удалении вершины проверяется три случая: у вершины нет детей, тогда указатель на родителя у фиктивного листа nil изменяется на данный элемент (это делается еще до проверок остальных случаев); у вершины один ребенок, тогда меняем местами существующего ребенка и удаляемый узел; у вершины есть оба ребенка, то находится узел с большим значением ключа. Так как нарушение свойств красно-черного дерева может нарушиться только тогда, когда удаляемая вершина красная, то метод fix_delete(node) вызывается только в том случае, если она красная. При восстановлении свойств меняются, в зависимости от случая, цвета узлов, применяются левые и правые вращения.

Случаи при удалении:

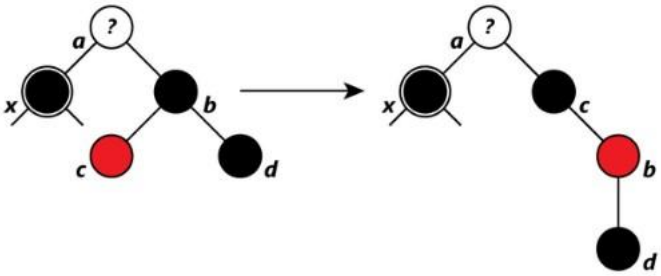
1. Если брат ребенка удаляемой вершины красный, то делаем вращение вокруг ребра между отцом и братом, тогда брат становится родителем отца. Красим в черный, а отца в красный.



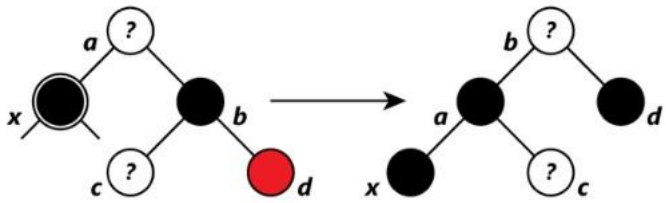
2. Оба ребенка у брата черные. Тогда красим брата в красный цвет и рассматриваем отца вершины. Делаем его черным, тем самым восстанавливая количество черных узлов.



3. Если у брата правый ребенок черный, а левый красный, то перекрашиваем брата и его левого сына и делаем вращение. Теперь у у и х есть черный брат с красным правым потомком.



4. Если у брата правый ребенок красный, то перекрашиваем брата в цвет отца, его ребенка и отца - в черный, делаем вращение.



Данные случаи продолжаются в рассматрении до тех пора, пока текущая вершина не черная и пока мы не дошли до корня дерева.

Реализация поиска. Поиск узла со значением key осуществлен в методе search(key). В данном методе осуществляется проход от корня до листов с проверками, больше или меньше ключ key текущего

рассматриваемого узла (соответственно, это интерпретируется, как левее или правее, согласно свойствам красно-черного дерева). Как только ключ совпал, осуществляется выход из цикла и выводится сообщение, что ключ найден. Так как проход осуществляется от корня к последнему уровню, то в общем случае будет h итераций, где h - высота дерева.

1.2. Хеш-таблица с открытой адресацией.

Хеш-таблица — это структура данных, реализующая интерфейс ассоциативного массива, а именно, она позволяет хранить пары (ключ, значение) и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу.

Для реализации хеш-таблицы с открытой адресацией были созданы классы Node и HashTable. Класс Node описывает собой элемент хеш-таблицы, который хранит в себе ключ и значение value.

При создании хеш-таблицы инициализируется ее размер (поле size), текущее количество записанных элементов current_size иницализируется 0. Сама таблица представляет собой список размера size, в котором будут храниться объекты класса Node (изначально записывается None). Также определены поля hash_type для выбора конкретного метода пробирования открытой адресации, параметры k, c1, c2, используемые в линейном и квадратичном пробировании. Вызывается метод choose_hash_type(), который позволяет выбрать желаемое хеширование.

Реализация вставки. Для вставки в хеш-таблицу реализован метод insert(key, value). В данном методе в текущее смещение idx записывается 0, а значение хеш-функции (результат метода hash_function(key)) в hash_value. В цикле while вычисляется на основе хеш-значения и перебираемых от 0 целых idx номер ячейки: вызывается метод hashing(hash_value, i), который на основании выбранного пользователем метода пробирования вызывает необходимый; линейное пробирование реализовано в методе linear_hashing(hash_value, i), квадратичное - в quadratic_hashing(hash_value, i),

двойное хеширование - double_hashing(hash_value, i). Основная хеш-функция - (key % table_size), вспомогательная функция (для двойного хеширования) - (7 - key % 7). После проверяется, что в списке ячейка пустая или удаленная, и в таком случае элемент записывается в хеш-таблицу. Если текущий размер таблицы стал равен 2/3 от максимального размера, то вызывается метод resize(), расширяющий таблицу вдвое. В случае, если ключ уже существует, то просто перезаписывается значение.

Реализация удаления. Для удаления из хеш-таблицы элемента с ключом key реализован метод delete(key). В данном методе проверяется, что таблица не пустая - тогда удалять нечего. В ином случае, аналогично вставке, вычисляется хеш-значение, записываемое в hash_value, осуществляется перебор idx от 0 до size-1 в цикле for. Постоянно пересчитывая номер текущей рассматриваемой ячейки cell_number сверяется значения ключа к удалению и ключа в ячейке с текущим номером - если они совпали, то данная ячейка помечается как deleted, текущий размер current_size уменьшается на 1. Ячейка помечается, как удаленная, так как если в нее записать None, то в дальнейшем мы не сможем найти те элементы, в момент которых в таблице данное место было занято (и из-за чего был выбран дальний элемент в последовательности испробованных мест).

Реализация поиска. Для поиска элемента с заданным ключом был создан метод search(key). В данном методе проверяется, что таблица не пустая - тогда искать нечего. В ином случае, аналогично удалению и вставке, вычисляется хеш-значение, записываемое в hash_value, осуществляется перебор idx от 0 до size-1 в цикле for. Постоянно пересчитывая номер текущей рассматриваемой ячейки cell_number сверяется значения ключа для поиска и ключа в ячейке с текущим номером - если они совпали, то возвращается найденный элемент Node.

2. СРАВНЕНИЕ СТРУКТУР ДАННЫХ

2.1. Теоретическая оценка сложностей базовых операций.

Все операции в красно-черном дереве, как в общем случае для почти всех самобалансирующихся бинарных деревьев, занимает O(log n). После удаления и вставки, для того, чтобы поддерживать необходимые свойства, необходимо сделать перекраску, и в худшем случае, не более трех поворотов (для вставки - не более двух). Данные операции поддержания красно-черного дерева занимают O(log h) или O(1), где h - высота дерева. Сложность всех случаев для RB-дерева также представлена на таблице 1.

Таблица 1. Сложности основных операций в RB-дереве

	В лучшем	В среднем	В худшем
Вставка	O(log n)	O(log n)	O(log n)
Удаление	O(log n)	O(log n)	O(log n)
Поиск	O(log n)	O(log n)	O(log n)

Сложность операций в хеш-таблице в целом составляет O(1), однако есть несколько тонкостей. Во-первых, по мере увеличения таблицы при открытой адресации возрастает количество коллизий - совпадение хеш-значений при разных ключах, так как занято большинство ячеек, и необходимо пробированием исскать новую. Во-вторых, разные пробирования (линейное, квадратичное исследование, двойное хеширование) могут дать разный результат, так как используют разные подходы в поиске нужной ячейки. Безусловно, задать хеш-таблицу заранее очень большой - вполне беспроигрышный вариант, однако такой подход не экономит память. Сложность всех операций представлена ниже на таблице 2. Стоит отметить, что a в таблице ниже представляет собой количество повторений получения коллизии (и, соответственно, время для разрешения этой коллизии; в случае открытой адресации - поиск свободного места путем пробирования).

Таблица 2. Сложности основных операций в Хеш-таблице (открытая адресация)			
	В лучшем	В среднем	В худшем
Вставка	O(1)	O(a)	O(n)
Удаление	O(1)	O(a)	O(n)
Поиск	O(1)	O(a)	O(n)

2.2. Сравнение экспериментальных значений.

Экспериментальное значение замерялось 100 раз для вставки, удаления и поиска одного элемента при постоянно увеличивающемся размере структуры. Для красно-черного дерева входные пары ключ-значения при каждой итерации перемешиваются с помощью random.shuffle(). Проверка вставки элементарна - необходимо постоянно добавлять элемент в дерево, замеряя время работы его метода insert(key, value). Проверка удаления и поиска аналогична - необходимо замерить время начала работы метода и время конца, при этом необходимо учесть, что размер структуры при удалении и поиске убывает, а, следовательно, конечное время необходимо «развернуть» с помощью метода reverse() для списка. На рисунках 1-3 представлены средние случаи работы красно-черного дерева.

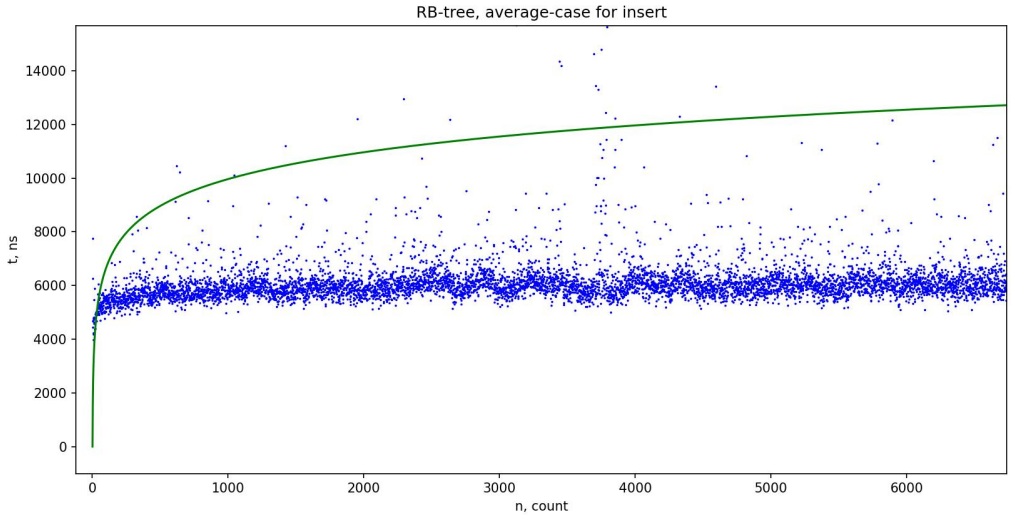


Рисунок 1. Вставка в RB-дерево (приближенный масштаб).

По оси абсцисс откладывается количество элементов в красно-черном дереве, при котором была осуществена операция с одним элементом, по оси ординат - время, в наносекундах. На рисунке 1 в приближенном масштабе видно, что зависимость логарифмическая (для синих точек). При этом также видно, что на графике некоторые точки значительно выше, чем зеленая кривая, отображающая логарифм - это связанно с тем, что при данных элементах была осуществлена балансировка, поэтому график является корректным.

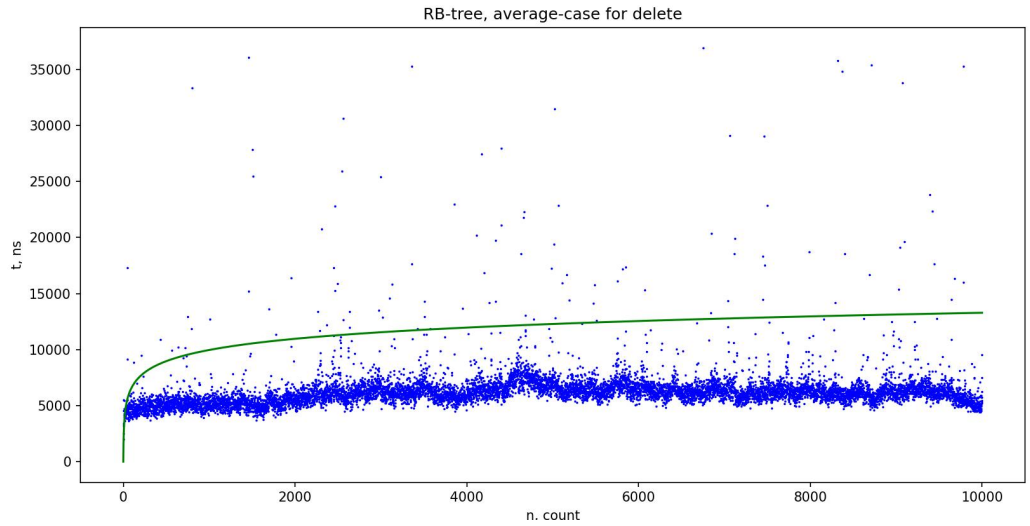


Рисунок 2. Удаление из RB-дерева.

На рисунке 2 видно, что зависимость логарифмическая (хотя и растет достаточно медленно, ведь проверка идет для одного элемента). Зеленая кривая представляет собой O(log n), ограничивая кривую сверху. График совпадает с теоретической оценкой.

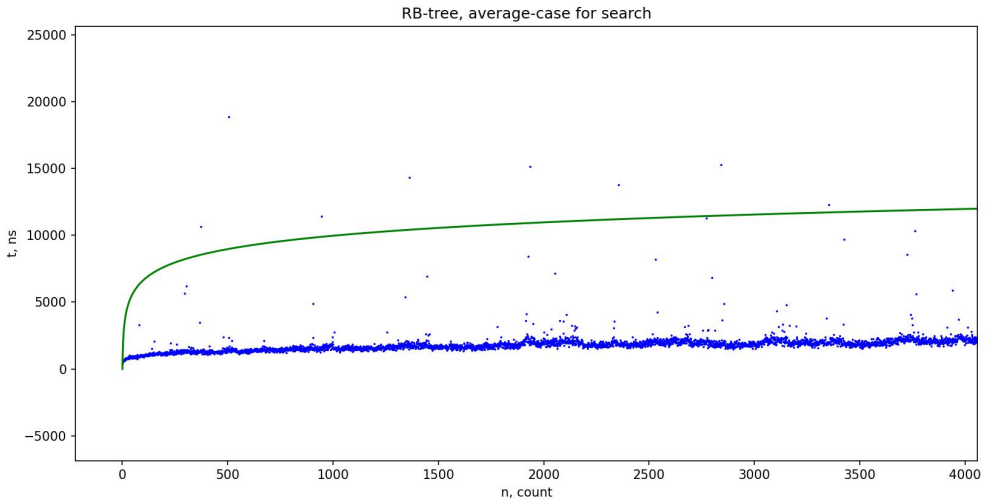


Рисунок 3. Поиск в RB-дереве (прбилиженный масштаб).

На рисунке 3 видно, что зависимость поиска одного элемента в красно-черном дереве - логарифмическая, сверху ее ограничивает график O(log n). Поиск в красно-черном дереве совпадает с теоретической оценкой.

На рисунках 4-12 представлены основные операции для хеш-таблицы с открытой адресацией (для линейного, квадратичного исследования, двойного хеширования). Так как значения ключей при данном исследовании задается случайно, то вполне возможна вероятность коллизий - поэтому данные графики в теории должны соответствовать среднему случаю. На некоторых графиках далее видно, что при генерации псевдослучайных чисел оказались лучшие и худшие случаи.

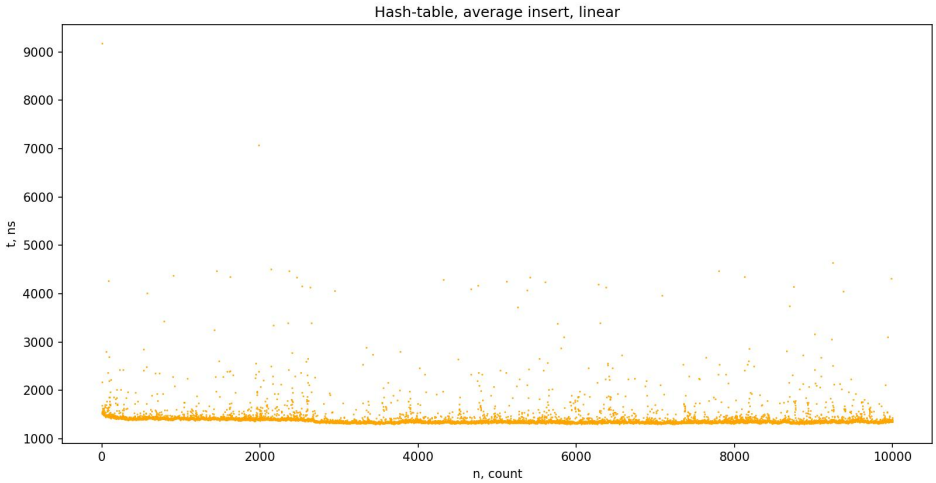


Рисунок 4. Вставка в хеш-таблицу с линейным исследованием.

На рисунке 4 видно, что зависимость от количества элементов при вставке в хеш-таблицу неоднозначна. Это связано с нечастными коллизиями, которые образуются при линейном пробировании - им соответсвуют точки, значительно отошедшие от графика, в свою очередь точки, лежащие снизу - соответствуют элементам при отсутствии коллизии (лучший частный случай). Таким образом, практическая и теоретическая оценка совпали.

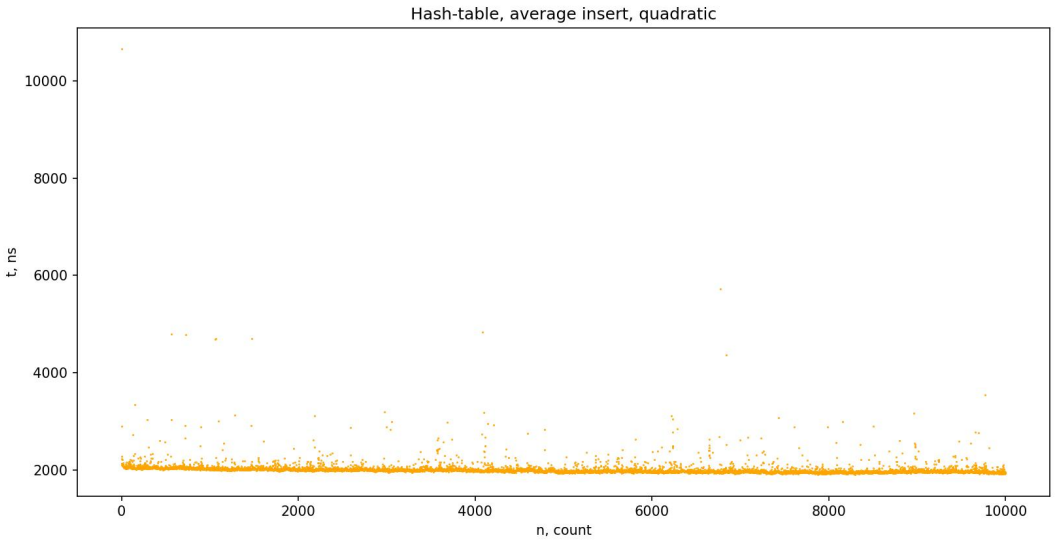


Рисунок 5. Вставка в хеш-таблицу с квадратичным исследованием

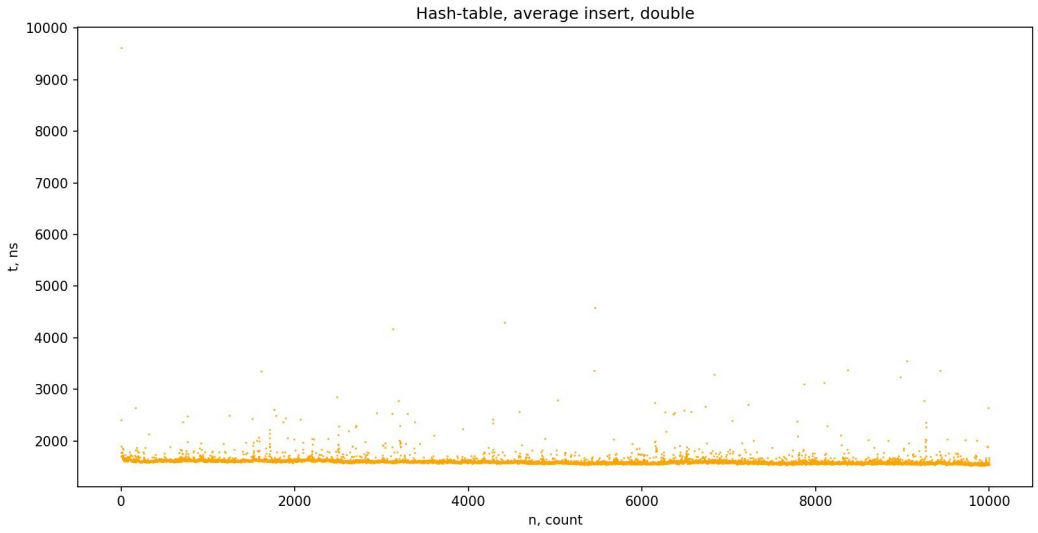


Рисунок 6. Вставка в хеш-таблицу с двойным хешированием.

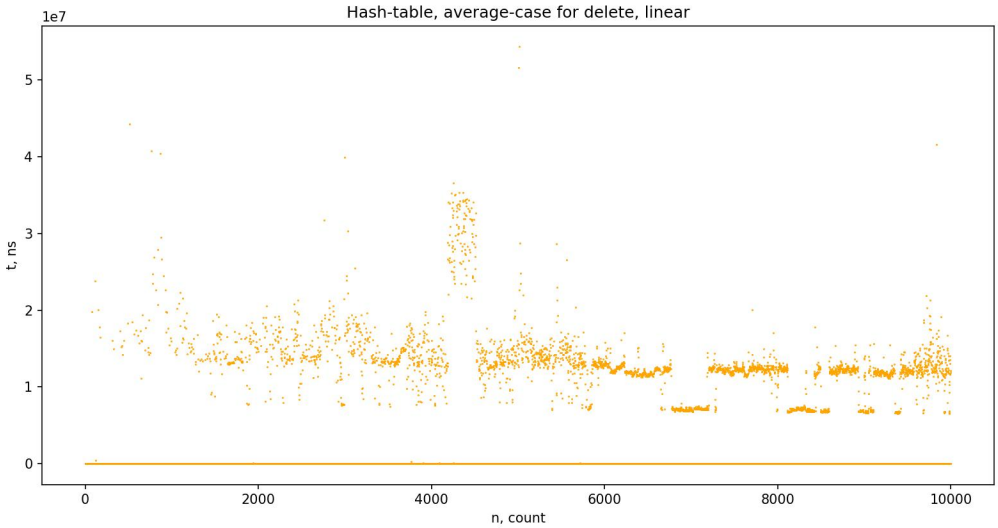


Рисунок 7. Удаление в хеш-таблице с линейным исследованием.

На рисунке 7 видно, что уже значительное число коллизий замедлило работу хеш-таблицы. Это связано с тем, что при удалении сначало необходимо «попасть» в ячейку с нужным ключом, что занимает значительно больше времени, чем вставка. Тем не менее, зависимость здесь от частоты коллизий, что соответствует теоретической оценке.

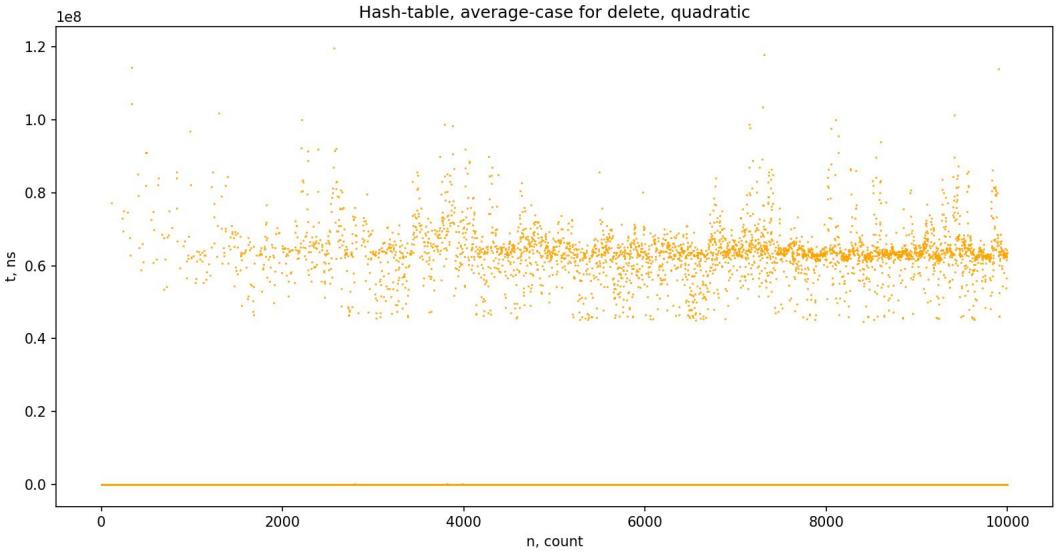


Рисунок 8. Удаление в хеш-таблице с квадратичным хешированием.

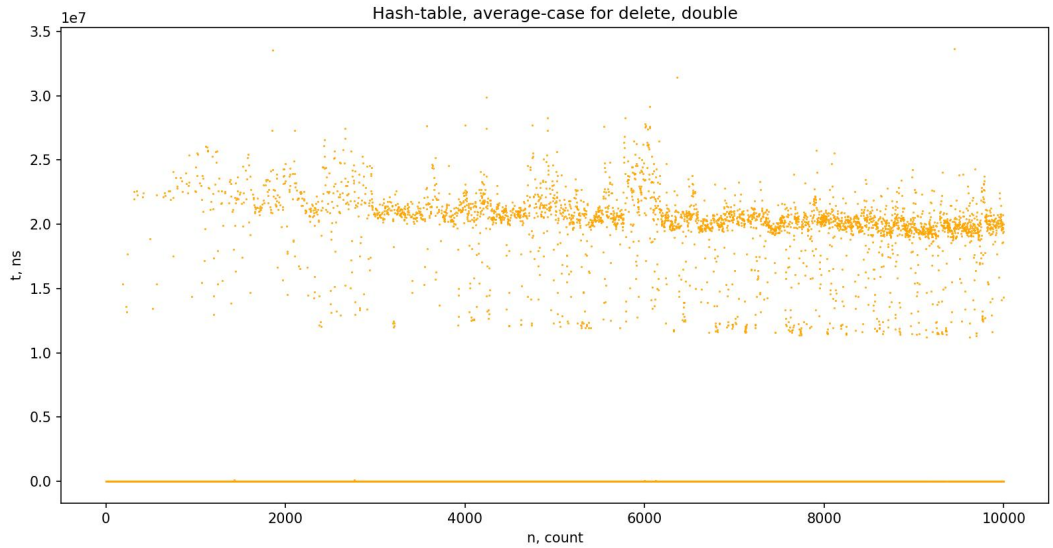


Рисунок 9. Удаление в хеш-таблице с двойным хешированием.

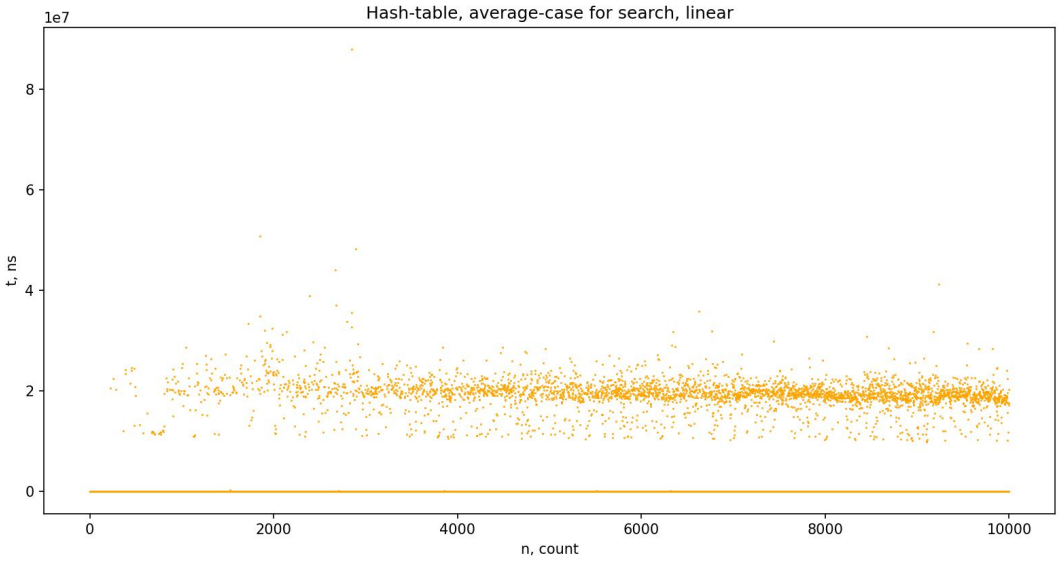


Рисунок 10. Поиск в хеш-таблице с линейным исследованием.

На рисунке 10 видно, что аналогично удалению, в поиске также частые коллизии.

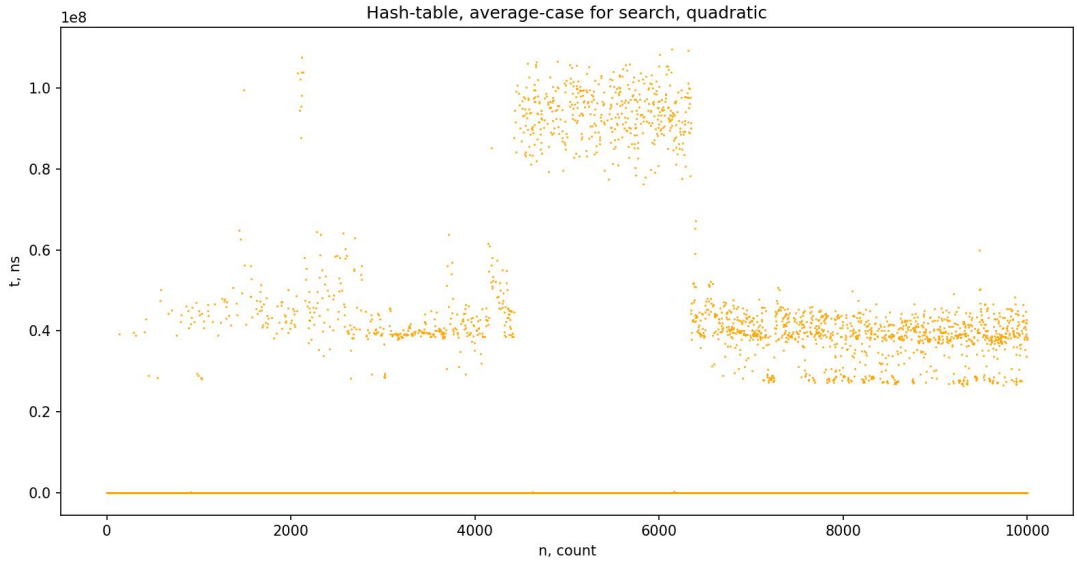


Рисунок 11. Поиск в хеш-таблице с квадратичным хешированием.

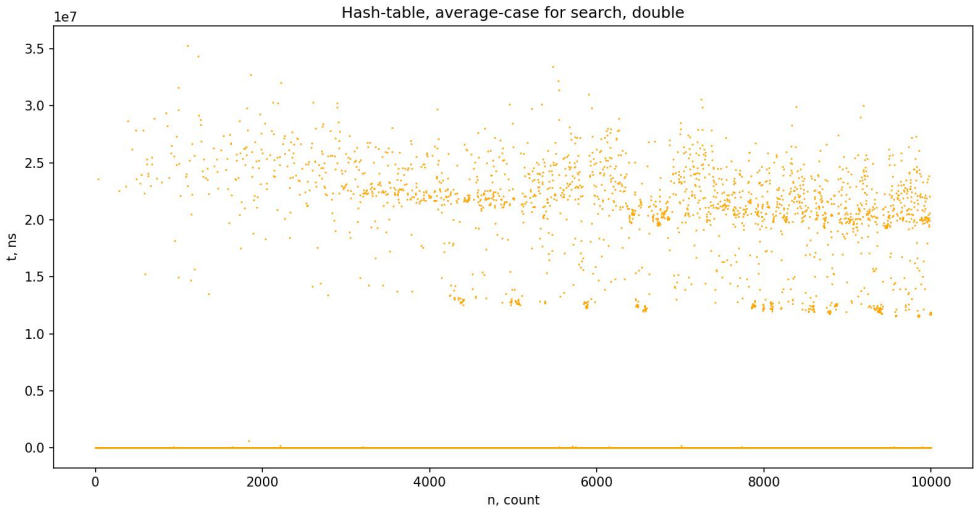


Рисунок 12. Поиск в хеш-таблице с двойным хешированием.

По итогу исследования подтверждены теоретические оценки для RB-дерева и хеш-таблицы. Исходя из графиков, видно, что RB-дерево более менее стабильно для любого количества данных. В хеш-таблице, не смотря на значительную эффективность при вставке (и при отсутствии расширения таблицы resize), удаление и поиск работают довольно медленно (хотя зависимость и O(a), где a - количество коллизий).

2.3. Сравнение структур между собой.

Для того, чтобы сравнить структуры между собой, были взяты псевдослучайные числа в количестве 10000, тестирование было проведено 100 раз на одних и тех же данных для обоих структур (10 раз для удаления и поиска). Первая группа тестирования проводилось с учетом коллизий (диапозон случайно генерируемых ключей был уменьшен на 10%). Данному тестированию соответсвуют рисунки с номерами 13-15. Во второй группе тестирования все ключи уникальны (рисунки 16-18).

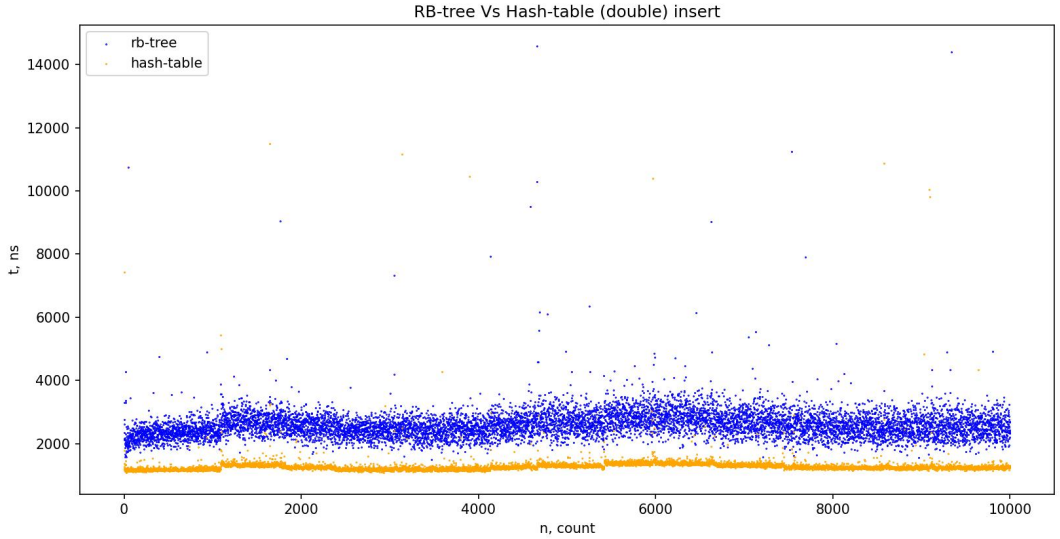


Рисунок 13. Сравнение RB-дерева и Хеш-таблицы во вставке

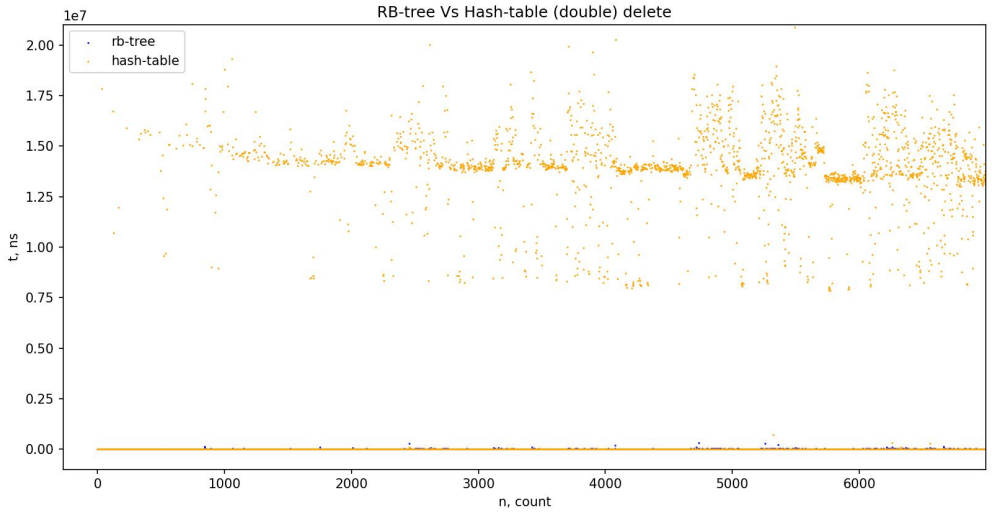


Рисунок 14. Сравнение RB-дерева и Хеш-таблицы в удалении (синяя кривая прижата к низу из-за масштабирования)

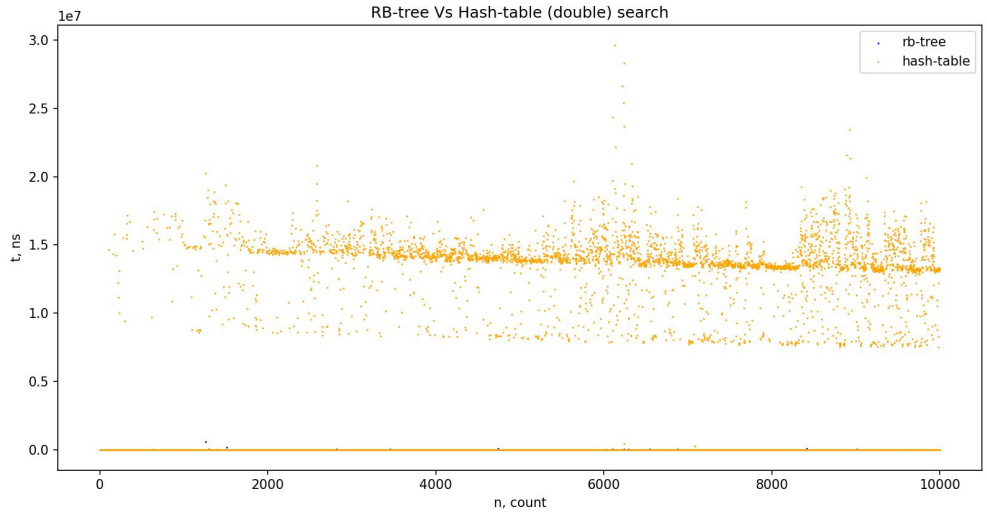


Рисунок 15. Сравнение RB-дерева и Хеш-таблицы в поиске

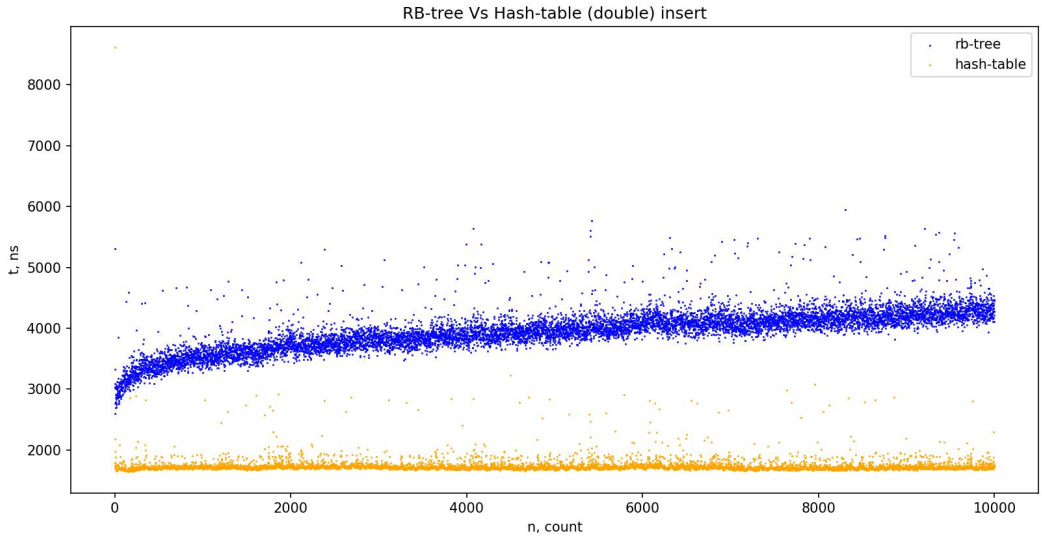


Рисунок 16. Сравнение RB-дерева и Хеш-таблицы во вставке (меньше коллизий для хеш-таблицы)

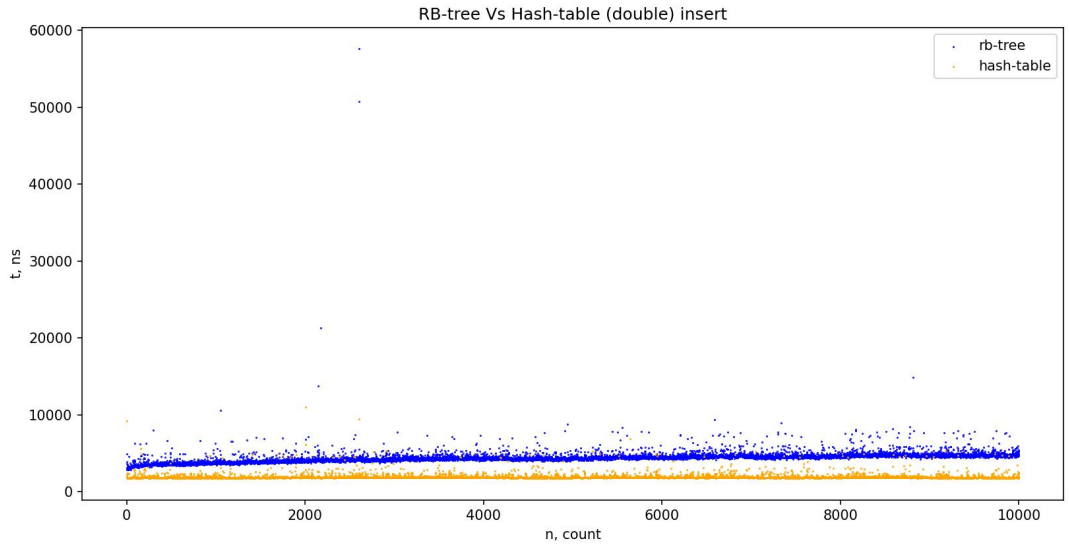


Рисунок 17. Сравнение RB-дерева и Хеш-таблицы в удалении(меньше коллизий для хеш-таблицы)

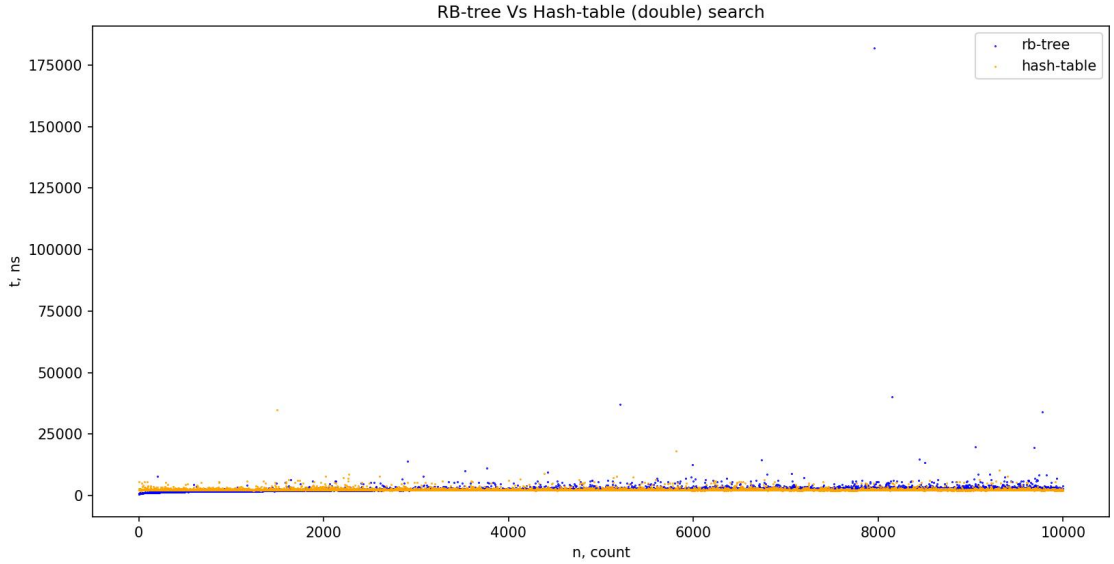


Рисунок 18. Сравнение RB-дерева и Хеш-таблицы в поиске(меньше коллизий для хеш-таблицы)

Из графиков первой группы видно, что коллизии оказывают существенное влияние на работоспособность хеш-таблицы, в то время как RB-дерево показывает стабильность в своих операциях. По графикам второй группы видно, что хеш-таблица не сильно быстрее красно-черного дерева в удалении и поиске, хотя эффективнее во вставке. Исходя из этого, можно сделать вывод, что RB-деревья являются более стабильными в своих операциях

вне зависимости от входных параметров, так как рост дерева не быстрый, в то время как хеш-таблицы наоборот, сильно к ним привязываются.

3. ПРИМЕРЫ РАБОТЫ ПРОГРАММЫ

```
C:\АиСД\cw\Scripts\python.exe C:\АиСД\cw\research.py
Enter the count of pair (key,value) for add in rb-tree
5
Enter the pairs (key,value) to add in rb-tree
1 2
2 3
3 4
4 5
-2 7
```

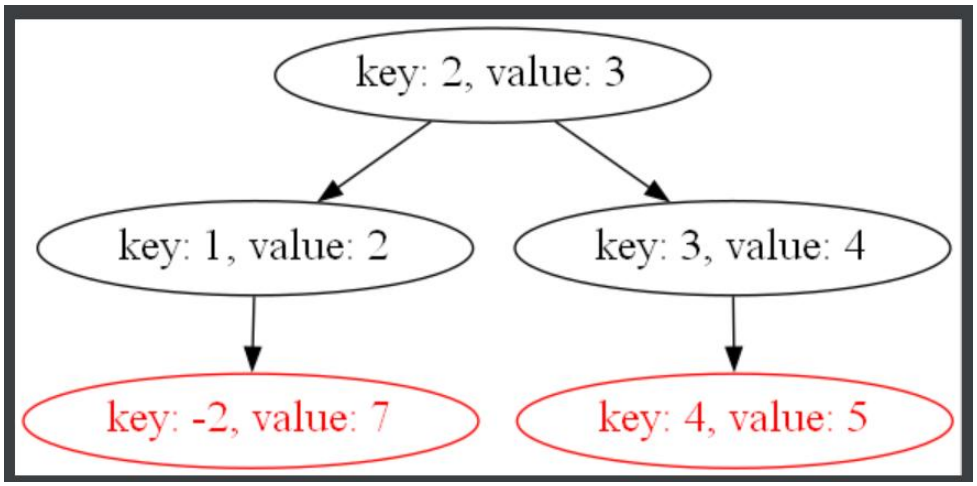


Рис.19-20 Проверка вставки в RB-дерево

```
Enter the count of pair (key,value) for add in rb-tree
5
Enter the pairs (key,value) to add in rb-tree
1 2
2 3
3 4
4 5
-2 7
Enter the key to delete
3
```

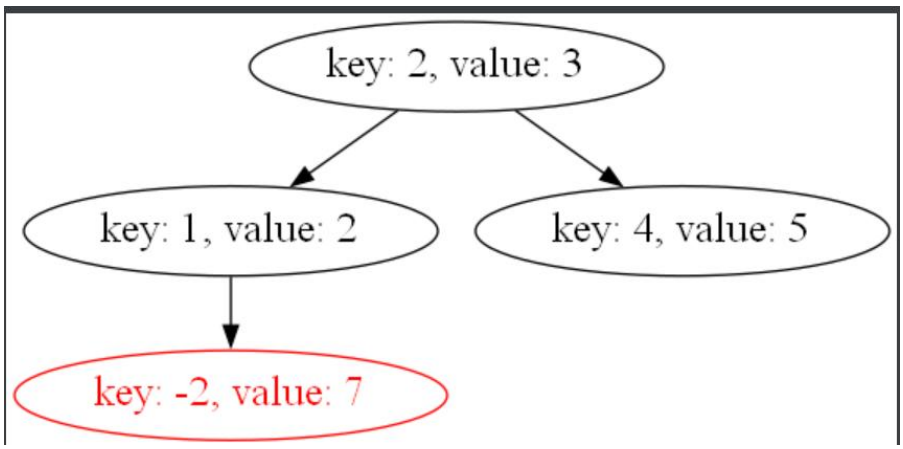


Рис. 21-22 Проверка удаления из RB-дерева

```
Enter the count of pair (key,value) for add in hash-table
5
Enter the pairs (key,value) to add in hash-table
1 1
3 0
2 9
77 4
1 78
Выберите функцию для последовательности проб: linear, quadratic, double
double
```

Рис. 23 Проверка вставки в хеш-таблицу

Результат: Hash table size: 100 (current size: 4), hash table: [['key: 1, value: 78', 'cell: 1'], ['key: 2, value: 9', 'cell: 2'], ['key: 3, value: 0', 'cell: 3'], ['key: 77, value: 4', 'cell: 77']]

```
Enter the key to delete
2 3 77
Table after delete
```

Рис. 24 Проверка удаления из хеш-таблицы

Результат: Hash table size: 100 (current size: 1), hash table: [['key: 1, value: 78', 'cell: 1'], ['deleted', 'cell: 2'], ['deleted', 'cell: 3'], ['deleted', 'cell: 77']]

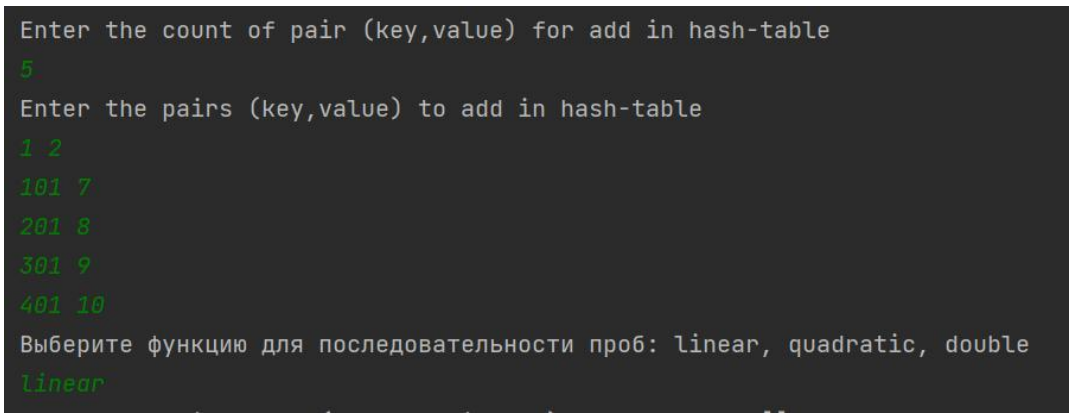


Рис.25 Проверка разрешения коллизий (размер хеш-таблицы = 100, функция - key % table.size)

Результат: Hash table size: 100 (current size: 5), hash table: [['key: 1, value: 2', 'cell: 1'], ['key: 101, value: 7', 'cell: 4'], ['key: 201, value: 8', 'cell: 7'], ['key: 301, value: 9', 'cell: 10'], ['key: 401, value: 10', 'cell: 13']]

ЗАКЛЮЧЕНИЕ

По итогам курсовой работы были реализованы структуры данных RB-дерево и Хеш-таблица с открытой адресацией на языке Python, а также проведено исследование работоспособности структур при различных входных параметрах. В ходе исследования было выяснено, что реализованные структуры совпадают с их теоритеческими оценками. Выявлено, что хеш-таблица работает эффективно только при хорошо подобранных входных параметрах, что соответсвует меньшей частоте их коллизий, в то время как RB-дерево, несмотря на то, что тратит время на балансировку, проявляет стабильность в операциях вне зависимости от них.

СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ

1. Алгоритмы: построение и анализ // Кормен, Т., Лейзерсон, Ч., Ривест, Р., Штайн, М.: Вильямс, 2005. 1296 с

2. Статья на сайте https://neerc.ifmo.ru/wiki/ «Красно-черное дерево». Дата обращения: 02.12.2022

3. Статья на сайте https://neerc.ifmo.ru/wiki/ «Разрешение коллизий». Дата обращения: 11.12.2022

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл: rb.tree.py

```
import sys
import graphviz
import os
import random
import copy
import time

BLACK = 'black'
RED = 'red'

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

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

    def __eq__(self, other):
        if not other:
            return False
        return self.key == other.key


class RBTree:
    def __init__(self):
        self.count_not_delete_nodes = 0
        self.root = None
        self.nil = Node(-1, color=BLACK)

    def search(self, key): #итеративный поиск
        current = self.root
        if not current or current == self.nil:
            # print('RB-дерево пустое')
            return
        while current and current.key != key:
            if key < current.key:
                current = current.left
            else:
                current = current.right
        if not current:
            # print('Узла с ключом {} нет'.format(key))
            return
        # print('Ключ найден: {}'.format(str(current)))
```

```
        return current

    def insert(self, key, value):
        if not self.root:
            self.root = Node(key, color=BLACK, value=value)
        else:
            current = self.root
            while current:
                if key == current.key:
                    current.value = value
                    return
                if key < current.key:
                    if not current.left:
                        new_node = Node(key, color=RED, value=value,
parent=current)
                        current.left = new_node
                        break
                    current = current.left
                else:
                    if not current.right:
                        new_node = Node(key, color=RED, value=value,
parent=current)
                        current.right = new_node
                        break
                    current = current.right
            self.fix_insert(new_node)

    def fix_insert(self, node):
        while node.parent and node.parent.color == RED:
            grand_parent = node.parent.parent #дедушка узла
            if node.parent == grand_parent.left:
                uncle = grand_parent.right

                if not uncle or uncle.color == BLACK:  # дядя отсутствует или
черный
                    if node == node.parent.right:
                        node = node.parent
                        self.left_rotate(node)
                    node.parent.color = BLACK
                    grand_parent.color = RED
                    self.right_rotate(grand_parent)
                else: # иначе дядя красный
                    node.parent.color = BLACK
                    uncle.color = BLACK
                    grand_parent.color = RED
                    node = grand_parent
            else: #если родитель нового узла правый сын
                uncle = grand_parent.left

                if not uncle or uncle.color == BLACK:  # дядя отсутствует или
черный
                    if node == node.parent.left:
                        node = node.parent
                        self.right_rotate(node)
                    node.parent.color = BLACK
                    grand_parent.color = RED
                    self.left_rotate(grand_parent)
```

```
                else:  # иначе дядя красный
                    node.parent.color = BLACK
                    uncle.color = BLACK
                    grand_parent.color = RED
                    node = grand_parent
        if self.root.color == RED:
            self.root.color = BLACK

    def fix_delete(self, node):
        while node != self.root and node.color == BLACK:
            if node == node.parent.left:
                brother = node.parent.right
                if brother and brother.color == RED:
                    brother.color = BLACK
                    node.parent.color = RED
                    self.left_rotate(node.parent)
                    brother = node.parent.right
                # Если оба ребенка черные
                if brother and (brother.left == None or brother.left.color ==
BLACK) and (brother.right == None or brother.right.color == BLACK):
                    brother.color = RED
                    node = node.parent
                else:
                    #если правый черный или None
                    if brother and (brother.right == None or brother.right.color
== BLACK):
                        brother.left.color = BLACK
                        brother.color = RED
                        self.right_rotate(brother)
                        brother = node.parent.right
                    if brother:
                        brother.color = node.parent.color
                    node.parent.color = BLACK
                    if brother and brother.right:
                        brother.right.color = BLACK
                    self.left_rotate(node.parent)
                    node = self.root
            else:
                brother = node.parent.left
                if brother and brother.color == RED:
                    brother.color = BLACK
                    node.parent.color = RED
                    self.right_rotate(node.parent)
                    brother = node.parent.left
                # Если оба ребенка черные
                if brother and (brother.left == None or brother.left.color ==
BLACK) and (brother.right == None or brother.right.color == BLACK):
                    brother.color = RED
                    node = node.parent
                else:
                    if brother and (brother.left == None or brother.left.color ==
BLACK):
                        brother.right.color = BLACK
                        brother.color = RED
                        self.left_rotate(brother)
                        brother = node.parent.left
                    if brother:
```

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

    def change_nodes(self, node1, node2):
        if not node1.parent:
            self.root = node2
        elif node1 == node1.parent.left:
            node1.parent.left = node2
        else:
            node1.parent.right = node2
        if not node2:
            node2 = self.nil
        node2.parent = node1.parent

    def delete(self, key):
        node_to_delete = None
        node = self.root
        if node == self.nil:
            #print("Node with {0} key doesn't exist. Tree is empty".format(key))
            self.count_not_delete_nodes += 1
            return
        while node:
            if node.key == key:
                node_to_delete = node
            if node.key <= key:
                node = node.right
            else:
                node = node.left
        if not node_to_delete:
            #print("Node with {0} key doesn't exist".format(key))
            self.count_not_delete_nodes += 1
            return

        if not node_to_delete.left:
            node_to_delete.left = self.nil
            self.nil.parent = node_to_delete
        if not node_to_delete.right:
            node_to_delete.right = self.nil
            self.nil.parent = node_to_delete

        y = node_to_delete
        y_origin_color = y.color
        if node_to_delete.left == self.nil:
            # у элемента нет левого сына
            x = node_to_delete.right
            self.change_nodes(node_to_delete, node_to_delete.right)
        elif (node_to_delete.right == self.nil):
            # у элемента нет правого сына
            x = node_to_delete.left
            self.change_nodes(node_to_delete, node_to_delete.left)
        else:
            # есть оба ребенка
```

```
            y = self.min_node(node_to_delete.right)
            y_origin_color = y.color
            x = y.right
            if not x:
                x = self.nil
            if y.parent == node_to_delete: # если y - ребенок node_to_delete
                x.parent = y
            else:
                self.change_nodes(y, y.right)
                y.right = node_to_delete.right
                y.right.parent = y
            self.change_nodes(node_to_delete, y)
            y.left = node_to_delete.left
            y.left.parent = y
            y.color = node_to_delete.color
        if y_origin_color == BLACK:
            self.fix_delete(x)

        if y.parent and y.parent.left == self.nil:
            y.parent.left = None
        if y.parent and y.parent.right == self.nil:
            y.parent.right = None
        self.nil.parent = None
        return y

    def left_rotate(self, node): # node - отец нового элемента
        if not node.right:
            return
        new_node = node.right
        node.right = new_node.left #LB

        if new_node.left:
            new_node.left.parent = node
        new_node.parent = node.parent
        if not node.parent:
            self.root = new_node
        else:
            if 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):
        if not node.left:
            return
        new_node = node.left
        node.left = new_node.right  # RB

        if new_node.right:
            new_node.right.parent = node
        new_node.parent = node.parent
        if not node.parent:
            self.root = new_node
        else:
            if node == node.parent.left:
```

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

    def min_node(self, node):
        if node:
            while node.left:
                node = node.left
            if not node:
                node = self.nil
            return node
        else:
            return self.nil

    def successor(self, node):
        if node.right != None:
            return self.min_node(node.right)
        y = node.parent
        while y != None and x == y.right:
            x = y
            y = y.parent
        return y
    def print(self, output_name):
        queue = [self.root]
        dot = graphviz.Digraph()
        dot.attr('node', fontsize='20')
        def print_node(node, parent_id=''):
            node_id = str(id(node))
            shape = 'ellipse' if node.key is not None else 'rectangle'
            dot.node(node_id, label="key: {}, value: {}".format(node.key,
node.value), color=node.color, fontcolor=node.color, shape=shape)
            if parent_id:
                dot.edge(parent_id, node_id)
        print_node(self.root)
        dot.format = 'png'
        while queue:
            tmp_queue = []
            for elem in queue:
                elem_id = str(id(elem))
                if elem.left:
                    print_node(elem.left, elem_id)
                    tmp_que.append(elem.left)
                if eleme.right:
                    print_node(elem.right, elem_id)
                    tmp_que.append(elem.right)
            queue = tmp_queue
        dot.render('result/{}'.format(output_name))

def breadth_first_search(root):
    result = []
    queue = [root]
    while queue:
        tmp_queue = []
        for element in queue:
            result.append(element.key)
```

```
            if element.left:
                tmp_queue.append(element.left)
            if element.right:
                tmp_queue.append(element.right)
        queue = tmp_queue
    return result
```

Файл: hash_table.py

```
import random
import time

DELETED = 'deleted' # для обработки удалений элементов

class Node:
    def __init__(self, key = None, value = None):
        self.key = key
        self.value = value

    def __str__(self):
        return 'key: {}, value: {}'.format(self.key, self.value)

    def __eq__(self, other):
        if other == DELETED:
            return False
        return self.key == other.key


class HashTable:
    def __init__(self, size, hash_type=None):
        self.size = size
        self.current_size = 0
        self.table = [None for _ in range(self.size)]
        self.hash_type = hash_type
        self.k = 3  # фиксированный интервал между ячейками при линейном
пробировани
        #Для того, чтобы все ячейки оказались просмотренными по одному разу,
необходимо,
        # чтобы k было взаимно-простым с размером хеш-таблицы.
        self.c1 = 3 # \
                    #  | коэффициенты при квадратичном пробировании
        self.c2 = 2 # /
        self.choose_hash_type()

    def choose_hash_type(self):
        if not self.hash_type:
            print('Выберите функцию для последовательности проб: linear,
quadratic, double')
            hash_type = input()
            if hash_type in ['linear', 'quadratic', 'double']:
                self.hash_type = hash_type
            else:
                print('По умолчанию выбрано линейное пробирование')
                self.hash_type = 'linear'
        else:
            return

    def resize(self):
```

```
        self.table += [None for _ in range(self.size)]
        self.size *= 2

    def hashing(self, i, hash_value=None, hash_value_first=None,
hash_value_second=None):
        if self.hash_type == 'linear':
            return self.linear_hashing(hash_value, i)
        elif self.hash_type == 'quadratic':
            return self.quadratic_hashing(hash_value, i)
        elif self.hash_type == 'double':
            return self.double_hashing(hash_value_first, hash_value_second, i)

    def linear_hashing(self, hash_value, i):
        cell_number = (hash_value + (i * self.k) % self.size) % self.size
        return cell_number

    def quadratic_hashing(self, hash_value, i):
        cell_number = ((hash_value + self.c1 * i) % self.size + self.c2 * pow(i,
2, self.size)) % self.size
        return cell_number

    def double_hashing(self, hash_value_first, hash_value_second, i):
        cell_number = (hash_value_first + (i * hash_value_second) % self.size ) %
self.size
        return cell_number

    def hash_function(self, key):
        return key % self.size

    def second_hash_function(self, key):
        return 7 - (key % 7)

    def insert(self, key, value):
        idx = 0
        if self.hash_type != 'double':
            hash_value = self.hash_function(key)
        else:
            first_hash_value = self.hash_function(key)
            second_hash_value = self.second_hash_function(key)
        while True:
            if self.hash_type != 'double':
                cell_number = self.hashing(idx, hash_value=hash_value)
            else:
                cell_number = self.hashing(idx, hash_value_first=first_hash_value,
hash_value_second=second_hash_value)
            if not self.table[cell_number] or self.table[cell_number] == DELETED:
# если нет такого ключа или он был удален
                self.table[cell_number] = Node(key, value)
                self.current_size += 1
                if self.current_size >= 0.66 * self.size:
                    self.resize()
                break
            else:
                if self.table[cell_number].key == key:
                    self.table[cell_number].value = value
                    break
                idx += 1 #ищем дальше свободное место
```

```
    def delete(self, key):
        if self.current_size == 0:
            #print('Хеш-таблица пустая. Элемент {} нельзя удалить'.format(key))
            return
        if self.hash_type != 'double':
            hash_value = self.hash_function(key)
        else:
            first_hash_value = self.hash_function(key)
            second_hash_value = self.second_hash_function(key)
        for idx in range(self.size):
            if self.hash_type != 'double':
                cell_number = self.hashing(idx, hash_value=hash_value)
            else:
                cell_number = self.hashing(idx, hash_value_first=first_hash_value,
hash_value_second=second_hash_value)
            if  self.table[cell_number] and self.table[cell_number] != DELETED
and self.table[cell_number].key == key:  # если ключ найден
                self.table[cell_number] = DELETED
                self.current_size -= 1
                return
        #print('Элемент с ключом {} не удален, так как не найден в хеш-
таблице'.format(key))

    def search(self, key):
        if self.current_size == 0:
            #print('Хеш-таблица пустая. Элемент {} отсутствует'.format(key))
            return
        if self.hash_type != 'double':
            hash_value = self.hash_function(key)
        else:
            first_hash_value = self.hash_function(key)
            second_hash_value = self.second_hash_function(key)
        for idx in range(self.size):
            if self.hash_type != 'double':
                cell_number = self.hashing(idx, hash_value=hash_value)
            else:
                cell_number = self.hashing(idx, hash_value_first=first_hash_value,
hash_value_second=second_hash_value)
            if self.table[cell_number] and self.table[cell_number] != DELETED and
self.table[cell_number].key == key:  # если ключ найден
                #print('Элемент найден в хеш-таблице:
{}'.format(self.table[hash_value]))
                return self.table[cell_number]
        #print('Элемент с ключом {} не найден в хеш-таблице'.format(key))
        return None

    def __str__(self):
        str_table = []
        for i in range(self.size):
            if self.table[i]:
                node_info = [str(self.table[i]), 'cell: {}'.format(i)]
                str_table.append(node_info)
        return 'Hash table size: {} (current size: {}), hash table:
{}'.format(self.size, self.current_size, str(str_table))
```

Файл: research.py

```
import time
import sys
import graphviz
import os
import gc
import copy
import hash_table
import rb_tree
import random
import matplotlib.pyplot as plt
import numpy


def clear_all_directory():
    dir = 'result'
    for f in os.listdir(dir): #clear previous info
        os.remove(os.path.join(dir, f))

def rb_tree_manual_input(): #пользовательская проверка работы дерева и
корректности реализации rb-tree
    clear_all_directory()
    print('Enter the count of pair (key,value) for add in rb-tree')
    count = int(input())
    print('Enter the pairs (key,value) to add in rb-tree')
    nodes_to_add = []
    for i in range(count):
        node = list(map(int, input().split()))
        nodes_to_add.append(node)
    tree = rb_tree.RBTree()
    for node in nodes_to_add:
        tree.insert(node[0], node[1])
    tree.print('rb_tree_graphviz')
    print('Enter the key to delete')
    keys_to_delete = list(map(int, input().split()))
    for keys in keys_to_delete:
        tree.delete(keys)
    tree.print('rb_tree_after_delete')

def hash_table_manual_input(): #пользовательская проверка работы хеш-таблицы и
корректности ее реализации
    print('Enter the count of pair (key,value) for add in hash-table')
    count = int(input())
    print('Enter the pairs (key,value) to add in hash-table')
    nodes_to_add = []
    for i in range(count):
        node = list(map(int, input().split()))
        nodes_to_add.append(node)
    table = hash_table.HashTable(100)
    for node in nodes_to_add:
        table.insert(node[0], node[1])
    print(table)
    print('Enter the key to delete')
    keys_to_delete = list(map(int, input().split()))
    for keys in keys_to_delete:
        table.delete(keys)
    print('Table after delete')
```

```
    print(table)

def create_graphics_with_log(n, times, title_name): #создать график на основе
кол-ва элементов и массива times
    plt.figure()
    plt.title(title_name)
    plt.xlabel("n, count")
    plt.ylabel("t, ns")
    plt.scatter(n, times, s=0.5, color='blue')
    plt.plot(n, numpy.log2(n) * 10**3, color='green')
    plt.savefig("result/{}.png".format(title_name))
    plt.show()

def create_graphics(n, times, title_name):
    plt.figure()
    plt.title(title_name)
    plt.xlabel("n, count")
    plt.ylabel("t, ns")
    plt.scatter(n, times, s=0.2, color='orange')
    plt.savefig("result/{}.png".format(title_name))
    plt.show()

def create_compared_graphics(n, times_tree, times_table, title_name):
    plt.figure()
    plt.title(title_name)
    plt.xlabel("n, count")
    plt.ylabel("t, ns")
    plt.scatter(n, times_tree, s=0.2, color='blue', label='rb-tree')
    plt.scatter(n, times_table, s=0.2, color='orange', label='hash-table')
    plt.legend(loc="best")
    plt.savefig("result/{}.png".format(title_name))
    plt.show()


def average_case_rb_tree_insert():
    print('Average-case for RB-tree inserting...')
    count = 10001
    research_count = 100
    n = [i for i in range(1, count)]
    insert_time = [0 for _ in range(1, count)]
    for i in range(research_count):
        nodes_to_add = [[random.randint(0, int(2*count)), random.randint(0, 1000)]
for i in range(1, count)]
        random.shuffle(nodes_to_add)
        tree = rb_tree.RBTree()
        for idx, elem in enumerate(nodes_to_add):
            gc.disable() #отключение сборщика мусора
            start = time.perf_counter_ns()
            tree.insert(elem[0], elem[1])
            end = time.perf_counter_ns() - start
            gc.enable()
            insert_time[idx] += end
        del tree
    for i in range(len(insert_time)):
        insert_time[i] /= research_count
    create_graphics_with_log(n, insert_time, "RB-tree, average-case for insert")
```

```
def average_case_rb_tree_delete():
    print('Average-case for RB-tree deleting...')
    count = 10001
    research_count = 100
    n = [i for i in range(1, count)]
    delete_time = [0 for i in range(1, count)]
    for i in range(research_count):
        nodes_to_add = [[random(count*3, count*5), random.randint(0, 1000)] for i
in range(1, count)]
        random.shuffle(nodes_to_add)
        tree = rb_tree.RBTree()
        for elem in nodes_to_add:
            tree.insert(elem[0], elem[1])
        for j in range(1, count):
            idx = random.randint(0, len(nodes_to_add) - 1)
            key_to_delete = nodes_to_add[idx][0]
            gc.disable()
            start = time.perf_counter_ns()
            tree.delete(key_to_delete)
            end = time.perf_counter_ns() - start
            gc.enable()
            delete_time[j-1] += end
            nodes_to_add.pop(idx)
        del tree
    delete_time.reverse()
    for i in range(len(delete_time)):
        delete_time[i] /= research_count
    create_graphics_with_log(n, delete_time, "RB-tree, average-case for delete")

def average_case_rb_tree_search():
    print('Average-case for RB-tree searching...')
    count = 10001
    research_count = 100
    n = [i for i in range(1, count)]
    search_time = [0 for i in range(1, count)]
    for i in range(research_count):
        nodes_to_add = [[i, random.randint(0, 1000)] for i in range(1, count)]
        random.shuffle(nodes_to_add)
        tree = rb_tree.RBTree()
        for elem in nodes_to_add:
            tree.insert(elem[0], elem[1])
        for j in range(1, count):
            idx = random.randint(0, len(nodes_to_add) - 1)
            key_to_search = nodes_to_add[idx][0]
            gc.disable()
            start = time.perf_counter_ns()
            tree.search(key_to_search)
            end = time.perf_counter_ns() - start
            gc.enable()
            search_time[j-1] += end
            tree.delete(key_to_search)
            nodes_to_add.pop(idx)
    search_time.reverse()
    for i in range(len(search_time)):
        search_time[i] /= research_count
    create_graphics_with_log(n, search_time, "RB-tree, average-case for search")
```

```
def average_case_hash_table_insert(hash_type):
    print('Hash-table inserting...')
    count = 10001
    research_count = 100
    n = [i for i in range(1, count)]
    insert_time = [0 for _ in range(1, count)]
    for i in range(research_count):
        nodes_to_add = [[random.randint(0, count), random.randint(0, 1000)] for i
in range(1, count)]
        times = []
        table = hash_table.HashTable(count * 3, hash_type)
        for idx, node in enumerate(nodes_to_add):
            gc.disable() #отключение сборщика мусора
            start = time.perf_counter_ns()
            table.insert(node[0], node[1])
            end = time.perf_counter_ns() - start
            gc.enable()
            insert_time[idx] += end
        del table
    for i in range(len(insert_time)):
        insert_time[i] /= research_count
    create_graphics(n, insert_time, "Hash-table, average insert,
{}".format(hash_type))

def average_case_hash_table_delete(hash_type):
    print('Average-case for Hash-table deleting...')
    count = 10001
    research_count = 100
    n = [i for i in range(1, count)]
    delete_time = [0 for i in range(1, count)]
    for i in range(research_count):
        nodes_to_add = [[random.randint(0, count*2), random.randint(0, 1000)] for
i in range(1, count)]
        table = hash_table.HashTable(count * 3, hash_type)
        for node in nodes_to_add:
            table.insert(node[0], node[1])
        for j in range(1, count):
            idx = random.randint(0, len(nodes_to_add) - 1)
            key_to_delete = nodes_to_add[idx][0]
            gc.disable()
            start = time.perf_counter_ns()
            table.delete(key_to_delete)
            end = time.perf_counter_ns() - start
            gc.enable()
            delete_time[j - 1] += end
            nodes_to_add.pop(idx)
        del table
    delete_time.reverse()
    for i in range(len(delete_time)):
        delete_time[i] /= research_count
    create_graphics(n, delete_time, "Hash-table, average-case for delete,
{}".format(hash_type))

def average_case_hash_table_search(hash_type):
    print('Average-case for Hash-table searching...')
    count = 10001
    research_count = 1
```

```
    n = [i for i in range(1, count)]
    search_time = [0 for i in range(1, count)]
    for i in range(research_count):
        nodes_to_add = [[random.randint(0, count*2), random.randint(0, 1000)] for
i in range(1, count)]
        table = hash_table.HashTable(count * 3, hash_type)
        for node in nodes_to_add:
            table.insert(node[0], node[1])
        for j in range(1, count):
            idx = random.randint(0, len(nodes_to_add) - 1)
            key_to_search = nodes_to_add[idx][0]
            gc.disable()
            start = time.perf_counter_ns()
            table.search(key_to_search)
            end = time.perf_counter_ns() - start
            gc.enable()
            search_time[j - 1] += end
            table.delete(key_to_search)
            nodes_to_add.pop(idx)
        del table
    search_time.reverse()
    for i in range(len(search_time)):
        search_time[i] /= research_count
    create_graphics(n, search_time, "Hash-table, average-case for search,
{}".format(hash_type))

def compare_insert(hash_type):
    print('RB-tree Vs Hash-table inserting...')
    count = 10001
    research_count = 100
    n = [i for i in range(1, count)]
    insert_time_tree = [0 for _ in range(1, count)]
    insert_time_table = [0 for _ in range(1, count)]
    for i in range(research_count):
        nodes_to_add = [[q, random.randint(0, 1000)] for q in range(1, count)]
        random.shuffle(nodes_to_add)
        tree = rb_tree.RBTree()
        table = hash_table.HashTable(count * 2, hash_type)
        for idx, elem in enumerate(nodes_to_add):
            gc.disable()  # отключение сборщика мусора
            start = time.perf_counter_ns()
            tree.insert(elem[0], elem[1])
            end = time.perf_counter_ns() - start
            gc.enable()
            insert_time_tree[idx] += end

            gc.disable()  # отключение сборщика мусора
            start = time.perf_counter_ns()
            table.insert(elem[0], elem[1])
            end = time.perf_counter_ns() - start
            gc.enable()
            insert_time_table[idx] += end
        del tree
        del table
    for i in range(len(n)):
        insert_time_tree[i] /= research_count
        insert_time_table[i] /= research_count
```

```
    create_compared_graphics(n, insert_time_tree, insert_time_table, "RB-tree Vs
Hash-table ({}) insert".format(hash_type))

def compare_delete(hash_type):
    print('RB-tree Vs Hash-table deleting...')
    count = 10001
    research_count = 100
    n = [i for i in range(1, count)]
    delete_time_tree = [0 for i in range(1, count)]
    delete_time_table = [0 for i in range(1, count)]
    for i in range(research_count):
        nodes_to_add = [[q, random.randint(0, 1000)] for q in range(1, count)]
        random.shuffle(nodes_to_add)
        tree = rb_tree.RBTree()
        table = hash_table.HashTable(count * 2, hash_type)
        for elem in nodes_to_add:
            tree.insert(elem[0], elem[1])
            table.insert(elem[0], elem[1])
        for j in range(1, count):
            idx = random.randint(0, len(nodes_to_add) - 1)
            key_to_delete = nodes_to_add[idx][0]
            gc.disable()
            start = time.perf_counter_ns()
            tree.delete(key_to_delete)
            end = time.perf_counter_ns() - start
            gc.enable()
            delete_time_tree[j - 1] += end

            gc.disable()
            start = time.perf_counter_ns()
            table.delete(key_to_delete)
            end = time.perf_counter_ns() - start
            gc.enable()
            delete_time_table[j - 1] += end

            nodes_to_add.pop(idx)
        del tree
        del table
    delete_time_tree.reverse()
    delete_time_table.reverse()
    for i in range(len(n)):
        delete_time_tree[i] /= research_count
        delete_time_table[i] /= research_count
    create_compared_graphics(n, delete_time_tree, delete_time_table, "RB-tree Vs
Hash-table ({}) delete".format(hash_type))

def compare_search(hash_type):
    print('RB-tree Vs Hash-table searching...')
    count = 10001
    research_count = 100
    n = [i for i in range(1, count)]
    search_time_tree = [0 for i in range(1, count)]
    search_time_table = [0 for i in range(1, count)]
    for i in range(research_count):
        nodes_to_add = [[q, random.randint(0, 1000)] for q in range(1, count)]
        random.shuffle(nodes_to_add)
        tree = rb_tree.RBTree()
```

```
            table = hash_table.HashTable(count * 2, hash_type)
            for elem in nodes_to_add:
                tree.insert(elem[0], elem[1])
                table.insert(elem[0], elem[1])
            for j in range(1, count):
                idx = random.randint(0, len(nodes_to_add) - 1)
                key_to_search = nodes_to_add[idx][0]
                gc.disable()
                start = time.perf_counter_ns()
                tree.search(key_to_search)
                end = time.perf_counter_ns() - start
                gc.enable()
                search_time_tree[j - 1] += end

                gc.disable()
                start = time.perf_counter_ns()
                table.search(key_to_search)
                end = time.perf_counter_ns() - start
                gc.enable()
                search_time_table[j - 1] += end

                table.delete(key_to_search)
                tree.delete(key_to_search)
                nodes_to_add.pop(idx)
            del tree
            del table
        search_time_tree.reverse()
        search_time_table.reverse()
        for i in range(len(n)):
            search_time_tree[i] /= research_count
            search_time_table[i] /= research_count
        create_compared_graphics(n, search_time_tree, search_time_table, "RB-tree Vs
Hash-table ({}) search".format(hash_type))

def check_rb_tree():
    average_case_rb_tree_insert()
    average_case_rb_tree_delete()
    average_case_rb_tree_search()

def check_hash_table():
    print('Choose hash-type: linear, quadratic, double')
    hash_type = input()
    average_case_hash_table_insert(hash_type)
    average_case_hash_table_delete(hash_type)
    average_case_hash_table_search(hash_type)

def compare_structure():
    print('Choose hash-type: linear, quadratic, double')
    hash_type = input()
    compare_insert(hash_type)
    compare_delete(hash_type)
    compare_search(hash_type)

if __name__ == '__main__':
    clear_all_directory()
    rb_tree_manual_input()
    hash_table_manual_input()
```

```
check_rb_tree()
check_hash_table()
compare_structure()
print('Research is complete! Check */result folder')
```