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: Исследование структур данных.

Студент гр. 0382 _____ Кривенцова Л.С.

Преподаватель _____ Берленко Т.А.

Санкт-Петербург

2021

ЗАДАНИЕ
НА КУРСОВУЮ РАБОТУ

Студентка: Кривенцова Л.С.

Группа 0382.

Тема работы: Исследование структур данных.

Вариант 8.

Исследование RB-дерева и Хеш-таблицы (двойное хеширование).

"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.

Дата выдачи задания: 18.10.2021

Дата сдачи реферата: 30.12.2021

Дата защиты реферата: 30.12.2021

Студент гр. 0382 _____ Кривенцова Л.С.

Преподаватель _____ Берленко Т.А.

АННОТАЦИЯ

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

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 Хеш-таблицы (двойное хеширование).

Задачи:

1. Изучить теоретические характеристики исследуемых структур данных;
2. Реализовать структуры, включая операции вставки, удаления и поиска элементов, с сохранением свойств той или иной структуры:
    a. Реализовать Хеш-таблицу методом двойного хеширования;
    b. Реализовать RB-дерево;
3. Реализовать наглядную демонстрацию времени работы каждой операции в разных случаях.
4. Сравнить полученные данные с теоретическими.
5. Сделать вывод по проделанной работе.

1. Теоретические сведения

1.1. Хеш-таблица (двойное хеширование)

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 и квадратичное зондирование.
- Среди методов также существует 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.

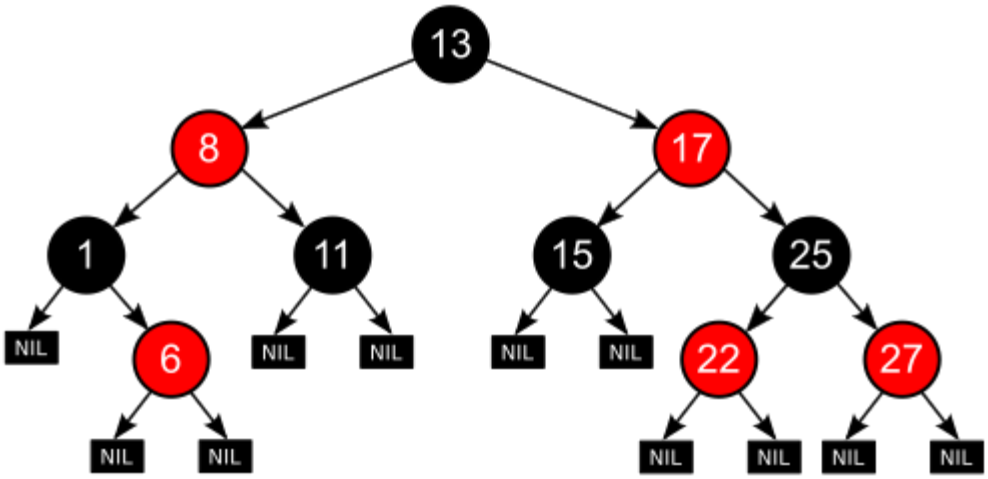
1.2. RB-дерево

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

2.1. Хеш-таблица (двойное хеширование)

Для реализации хеш-функции был создан класс *DoubleHashing*.

В нём объявлены следующие методы:

- *Hashing1(self, key)* – хеш-функция, использующая метод середины квадрата (значение ключа подвергается сдвигу на 11 бит и возвращаются 10 младших бит от получившегося числа).
- *Hashing2(self, key)* – хеш-функция, использующая метод деления.
- *Insert(self, key, data, i=0)* – добавление нового элемента в хеш-таблицу. Сначала по ключу, переданному методу в качестве аргумента, вычисляется заданная хеш-функция и определяется место в таблице, куда нужно поместить ключ. Если это место доступно, то в него записывается пара ключ – значение. В другом случае ищется другое доступное место, вычисление которого отличается от предыдущего на один шаг параметра *i* (в сторону увеличения). В процессе метода отслеживается переполнение (превышение на 2/3) с помощью поля *Overflow*, и при необходимости таблица создаётся заново с увеличенным числом ячеек (и копированием в новую таблицу старых значений).
- *Remove(self, key, i=0)* – удаление заданного элемента. Рекурсивно находит индекс заданного элемента в таблице и заменяет пару ключ – значение на *None*.
- *InPrimes(self)* – проверка размера хеш-таблицы на простоту.
- *FixSize(self)* – изменение поля величины таблицы таким образом, чтобы оно оставалось простым числом (необходимость при двойном хешировании).
- *FindKey(self, key, i=0)* – поиск элемента в хеш-таблице по ключу. К переданному методу в качестве аргумента ключу применяется функция двойного хеширования. Метод возвращает значение по

вычисленному индексу. Если значение не получено, поиск продолжится рекурсивно (также посредством увеличения передающегося в аргументах параметра *i).* При отсутствии элемента с заданным ключом в таблице (если значение *i* превысил размер таблицы и поиск завершен), метод возвращает "*None*".

Промежуточное тестирование:

Отрывок кода с генерацией входных данных:

```
hash = DoubleHashing(10)
for i in range(10):
    hash.Insert(i,str(i))
print(hash.Hash_Table)
hash.Insert(9,9)
print(hash.Hash_Table)
hash.Insert(11,'WoW')
print(hash.FindKey(10))
print(hash.FindKey(11))
hash.Remove(2)
print(hash.Hash_Table)
```

Выходные данные:

[[0, '0'], [1, '1'], [2, '2'], [3, '3'], [4, '4'], [5, '5'], [6, '6'], [7, '7'], [8, '8'], [9, '9'], [], [], [], [], [], [], [], [], [], [], [], [], []]

[[0, '0'], [1, '1'], [2, '2'], [3, '3'], [4, '4'], [5, '5'], [6, '6'], [7, '7'], [8, '8'], [9, 9], [], [], [], [], [], [], [], [], [], [], [], [], []]

None

WoW

[[0, '0'], [1, '1'], [None], [3, '3'], [4, '4'], [5, '5'], [6, '6'], [7, '7'], [8, '8'], [9, 9], [], [11, 'WoW'], [], [], [], [], [], [], [], [], [], [], []]

Следовательно, программа работает верно.

2.2. RB-дерево

Для реализации RB-дерева был созданы классы *RBTree* (структура дерева) и *Node* (элемент дерева).

Элемент дерева хранит следующие свойства:

Значение ключа(*key*), цвет элемента(*color*) , родитель(*parent*), левый и правый сыновья (*left, right*). Также для удобства реализованы методы нахождения дяди и дедушки (методы *Uncle(self)* и *Grandfather(self*)). Были переопределены некоторые методы, в частности *__str__(self),* который позволяет выводить элемент дерева на экран в следующем виде: *'key: {}, left: {}, right: {}, color: {}, parent: {}'*.

Основными операциями в красно-чёрном дереве (*RBTree*) являются: вставка(*Insert(self, key*)), удаление (*Remove(self, key)),* поиск (*Find(self, key*) и *_find(self, key, root)*), правый и левый повороты *(RightRotate(self, node)* и *LeftRotate(self, node)).*

Вставка.

Новый узел в красно-чёрное дерево добавляется на место одного из листьев, окрашивается в красный цвет и к нему прикрепляется два листа *nil* (так как листья являются абстракцией, не содержащей данных, их добавление не требует дополнительной операции). Что происходит дальше, зависит от цвета близлежащих узлов.

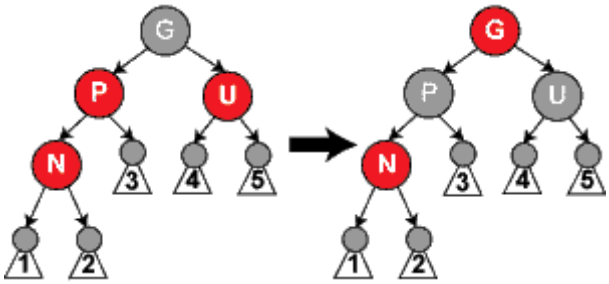
Непосредственно вставка узла происходит в методе *Insert(self, key*), а починка дерева (обработка случаев, рассмотренных ниже) в методе *FixTree_INSERT(self, node).*

Случай 1: Текущий узел N в корне дерева. В этом случае, он перекрашивается в чёрный цвет, чтобы оставить верным свойство (Корень — чёрный). Так как это действие добавляет один чёрный узел в каждый путь, свойство (о том что все пути от любого данного узла до листовых узлов содержат одинаковое число чёрных узлов) не нарушается.

Случай 2: Предок текущего узла чёрный, то есть свойство (Оба потомка каждого красного узла — чёрные) не нарушается. В этом случае дерево остаётся корректным. Свойство (Все пути от любого данного узла до листовых узлов содержат одинаковое число чёрных узлов) не нарушается, потому что текущий узел N имеет двух чёрных листовых потомков, но так как N является красным, путь до каждого из этих потомков содержит такое же число чёрных узлов, что и путь до чёрного листа, который был заменен текущим узлом, так что свойство остается верным.

Случай 3: Если и родитель, и дядя — красные, то они оба могут быть перекрашены в чёрный, и дедушка станет красным (для сохранения свойства (Все пути от любого данного узла до листовых узлов содержат одинаковое число чёрных узлов)). Теперь у текущего красного узла N чёрный родитель. Так как любой путь через родителя или дядю должен проходить через дедушку, число чёрных узлов в этих путях не изменится. Однако, дедушка G теперь может нарушить свойства (Корень — чёрный) или (Оба потомка каждого красного узла — чёрные) (свойство может быть нарушено, так как родитель G может быть красным).

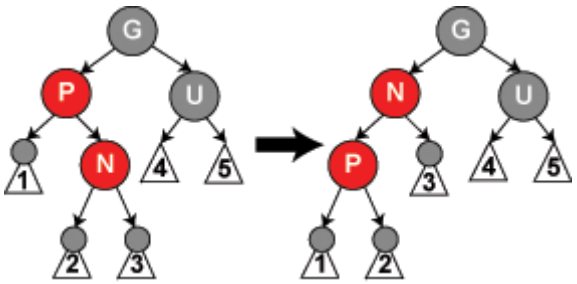
Рис.4. - иллюстрация случая 3 красно-чёрного дерева



Случай 4: Родитель является красным, но дядя — чёрный. Также, текущий узел N — правый потомок P, а P в свою очередь — левый потомок своего предка G. В этом случае может быть произведен поворот дерева, который меняет роли текущего узла N и его предка P. Тогда, для бывшего родительского узла P в обновленной структуре используем случай 5, потому что Свойство 4 (Оба потомка любого красного узла — чёрные) все ещё нарушено. Вращение приводит к тому, что некоторые пути (в поддереве, обозначенном «1» на схеме) проходят

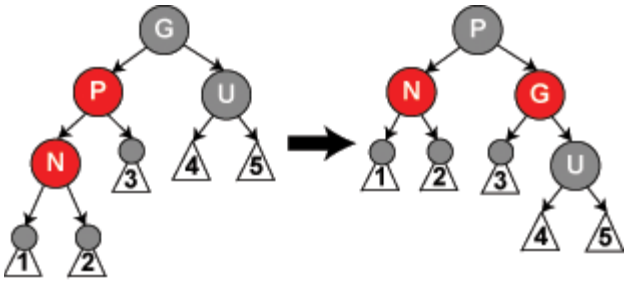
через узел N, чего не было до этого. Это также приводит к тому, что некоторые пути не проходят через узел P. Однако, оба эти узла являются красными, так что свойство (Все пути от любого данного узла до листовых узлов содержат одинаковое число чёрных узлов) не нарушается при вращении. Однако свойство всё ещё нарушается, но теперь задача сводится к Случаю 5.

Рис.5. - иллюстрация случая 4 красно-чёрного дерева



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

Рис.6. - иллюстрация случая 3 красно-чёрного дерева



Удаление.

При удалении узла с двумя нелистовыми потомками в обычном двоичном дереве поиска мы ищем либо наибольший элемент в его левом поддереве, либо наименьший элемент в его правом поддереве и перемещаем его значение в удаляемый узел. Затем мы удаляем узел, из которого копировали значение. Копирование значения из одного узла в другой не нарушает свойств красно-чёрного дерева, так как структура дерева и цвета узлов не изменяются. Стоит заметить, что новый удаляемый узел не может иметь сразу два дочерних нелистовых узла, так как в противном случае он не будет являться наибольшим/наименьшим элементом. Таким образом, получается, что случай удаления узла, имеющего два нелистовых потомка, сводится к случаю удаления узла, содержащего как максимум один дочерний нелистовой узел. Поэтому дальнейшее описание будет исходить из предположения существования у удаляемого узла не более одного нелистового потомка.

Непосредственно вставка узла происходит в методе *Remove(self, key)*, а починка дерева (обработка случаев, рассмотренных ниже) в методе *FixTree_REMOVE(self, node).* Также используется вспомогательный метод *Next(node).*

Будем использовать обозначение M для удаляемого узла; через C обозначим потомка M, который также будем называть просто «его потомок». Если M имеет нелистового потомка, возьмем его за C. В противном случае за C возьмем любой из листовых потомков.

Если M является красным узлом, заменим его своим потомком C, который по определению должен быть чёрным. (Это может произойти только тогда, когда M имеет двух листовых потомков, потому что если красный узел M имеет чёрного нелистового потомка с одной стороны, а с другой стороны — листового, то число чёрных узлов на обеих сторонах будет различным, таким образом дерево станет недействительным красно-чёрным деревом из-за нарушения свойства.) Все пути через удаляемый узел просто будут содержать на один красный узел меньше, предок и потомок удаляемого узла должны быть чёрными, так что

свойство («Все листья — чёрные») и свойство («Оба потомка красного узла — чёрные») все ещё сохраняется.

Другим простым является случай, когда M — чёрный и C — красный. Простое удаление чёрного узла нарушит свойство («Оба потомка красного узла — чёрные») и свойство («Всякий простой путь от данного узла до любого листового узла, содержит одинаковое число чёрных узлов»), но если мы перекрасим C в чёрный, оба эти свойства сохранятся.

Сложным является случай, когда и M и C — чёрные. (Это может произойти только тогда, когда удаляется чёрный узел, который имеет два листовых потомка, потому что если чёрный узел M имеет чёрного нелистового потомка с одной стороны, а с другой — листового, то число чёрных узлов на обеих сторонах будет различным и дерево станет недействительным красно-чёрным деревом из-за нарушения свойств.) Мы начнём с замены узла M своим потомком C. Будем называть этого потомка (в своем новом положении) N, а его «брата» (другого потомка его нового предка) — S. (До этого S был «братом» M.) На рисунках ниже мы также будем использовать обозначение P для нового предка N (старого предка M), SL для левого потомка S и SR для правого потомка S (S не может быть листовым узлом, так как если N по нашему предположению является чёрным, то поддерево P, которое содержит N, чёрной высоты два и поэтому другое поддерево P, которое содержит S должно быть также чёрной высоты два, что не может быть в случае, когда S — лист).

Поиск.

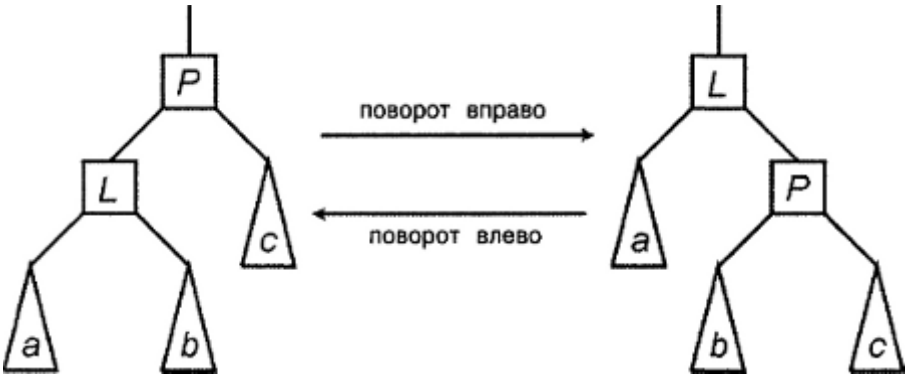
В методе *_find(self, key, root)* поиск начинается с проверки корневого узла, а затем циклично перемещается по дереву вниз, сравнивая ключ текущего узла, и переданного методу. Метод *Find(self, key)* сравнивает полученный от первого метода результат с тем, что нужно найти, и при неудаче возвращает *None*.

Повороты.

Это операция на бинарном дереве, которая изменяет структуру без вмешательства в порядок элементов. Поворот дерева перемещает один узел вверх в дереве и один узел вниз. Он используется для изменения формы дерева и, в

частности, для уменьшения его высоты путем перемещения меньших подчиненных деревьев вниз и больших подчиненных деревьев вверх, что приводит к повышению производительности многих операций дерева.

Рис. 7. – иллюстрация правого и левого вращений бинарного дерева.



Промежуточное тестирование:

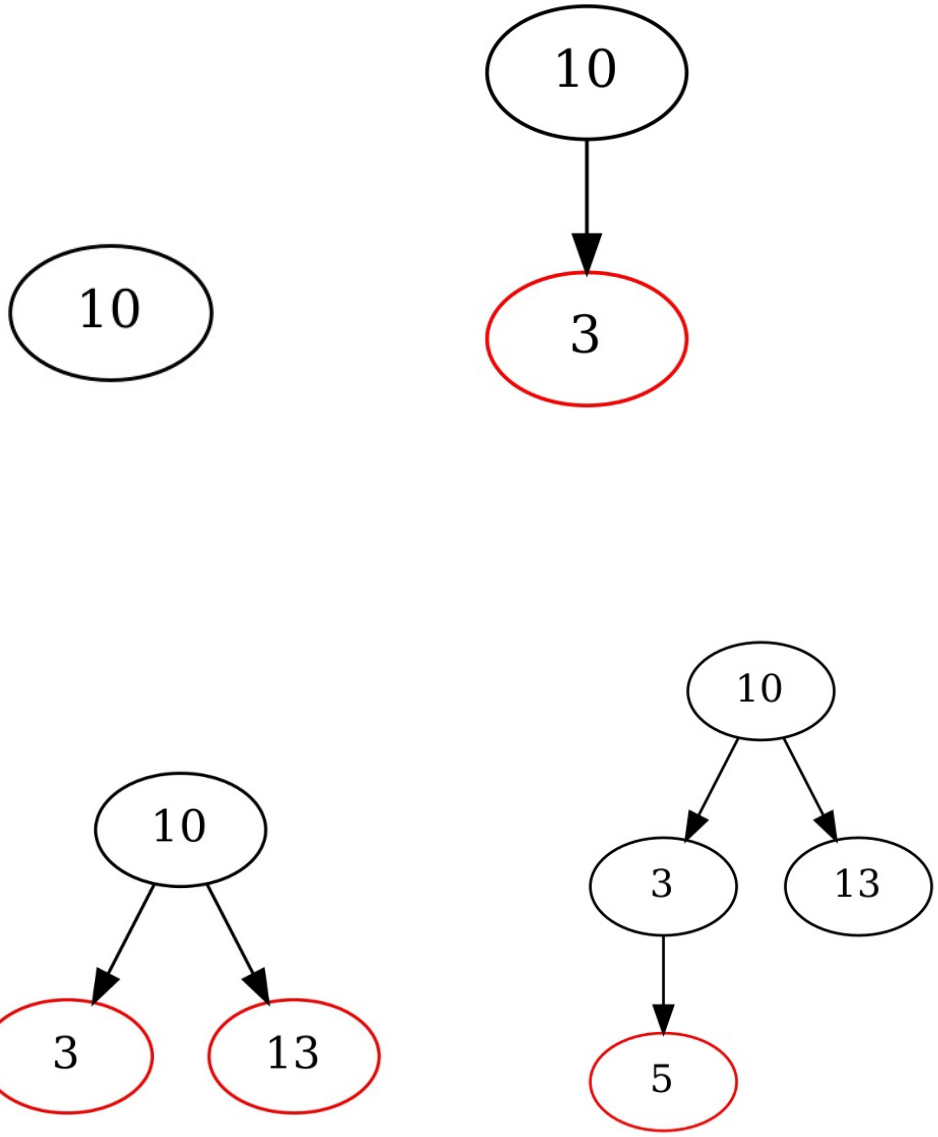
Для тестирования корректности работы реализованного RB-дерева использовалась библиотека *graphviz*. Для неё был написан метод (вне классов) обхода дерева в ширину *BreadthSearch(root, dot).* Модуль *graphviz* позволяет увидеть проиллюстрированные шаги работы алгоритма, но не рекомендуется к применению на больших деревьях.

Пример 1.

Входные данные:

10 3 13 5

Рис.9.,10.,11.,12. – пошаговая иллюстрация создания красно-черного дерева на основе массива чисел



Замечание: листы (*nil*) на иллюстрациях не отображены, но учитываются в коде программы.

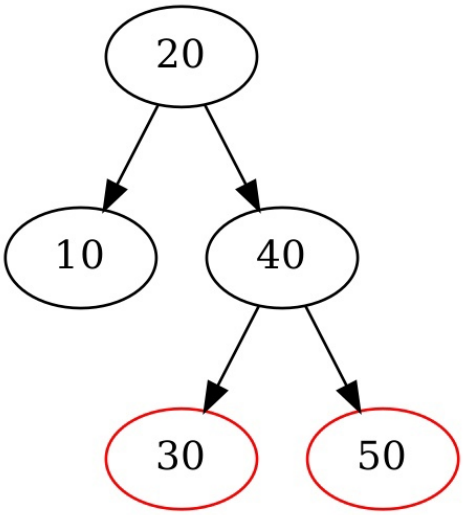
Пример 2:

Входные данные:

10 20 30 40 50

Результат дерева (после цикла, до удаления элемента)

Рис.14. –иллюстрация построенного программой дерева



Вывод: программа работает верно.

3. Реализация оценки 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, двойное хеширование.

Таблица 1. Теоретическая сложность операций хеш-таблицы

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

Примечание: a – количество возникших коллизий.

- В качестве лучшего случая на вход поступал упорядоченный массив,

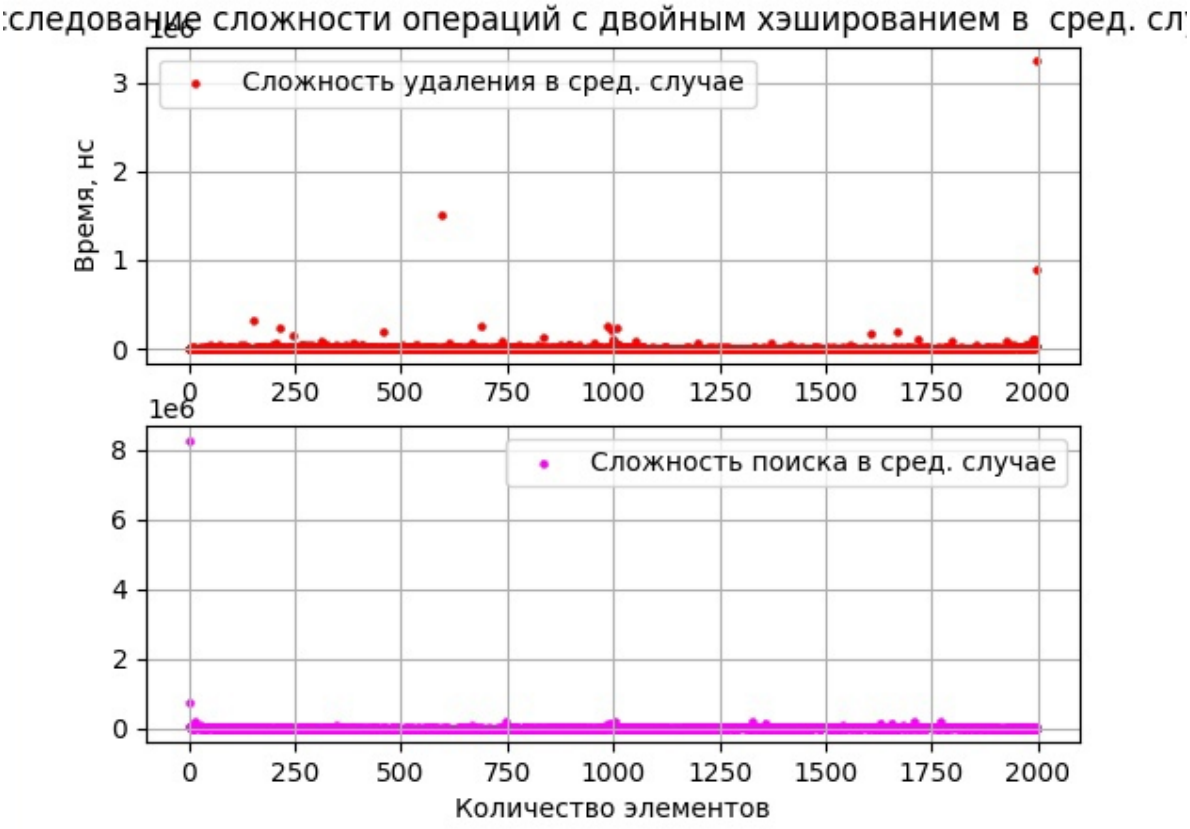
осуществлялся поиск и удаление статичного элемента, заданный размер таблицы не увеличивался.

- В качестве среднего случая на вход поступал упорядоченный массив, но с искусственно созданными коллизиями (число возникающих коллизий отличается при различных операциях), осуществлялся поиск и удаление статичного элемента, заданный размер таблицы увеличивался только один раз за всю обработку
- В качестве худшего случая на вход поступал массив псевдослучайных чисел, с искусственно созданными коллизиями (число возникающих коллизий отличается при различных операциях), осуществлялся поиск и удаление не статичного элемента, заданный размер таблицы постоянно увеличивается.

Сложность операций в среднем случае:

Рис.15. – график сложности вставки в хеш-таблицу (двойное хеширование) в среднем случае

Рис.16. – график сложности удаления и поиска в хеш-таблице (двойное хеширование) в среднем случае



Результаты, полученные в ходе реализации хеш-таблицы корректны (как мы видим, сложность и правда линейная, зависящая от константы, полученной в ходе выполнения.)

Сложность операций в лучшем случае:

Рис.17. – график сложности поиска в хеш-таблице (двойное хеширование) в лучшем случае

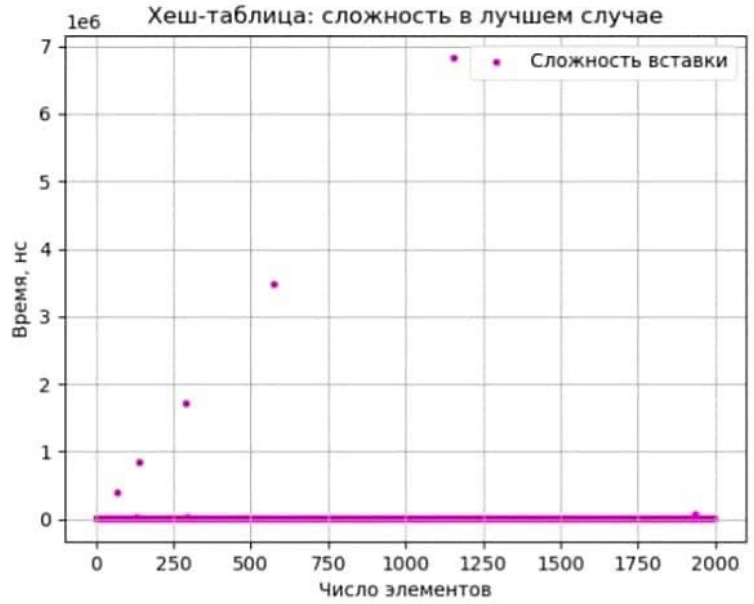
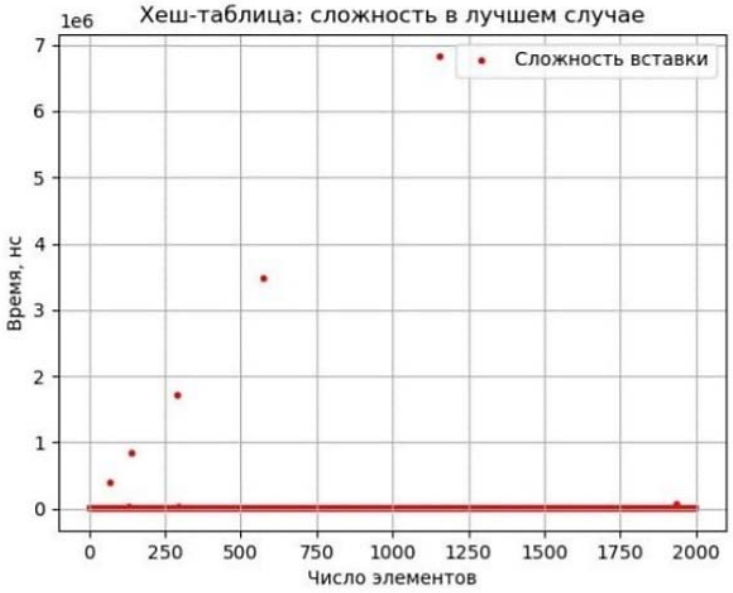


Рис.18. – график сложности удаления в хеш-таблице (двойное хеширование) в лучшем случае

Рис.19. – график сложности вставки в хеш-таблицу (двойное хеширование) в лучшем случае



Следовательно, результаты корректны – на графиках абсолютно точно видна линейная сложность.

Сложность операций в худшем случае:

Рис.20. – график сложности вставки в хеш-таблицу (двойное хеширование) в худшем случае

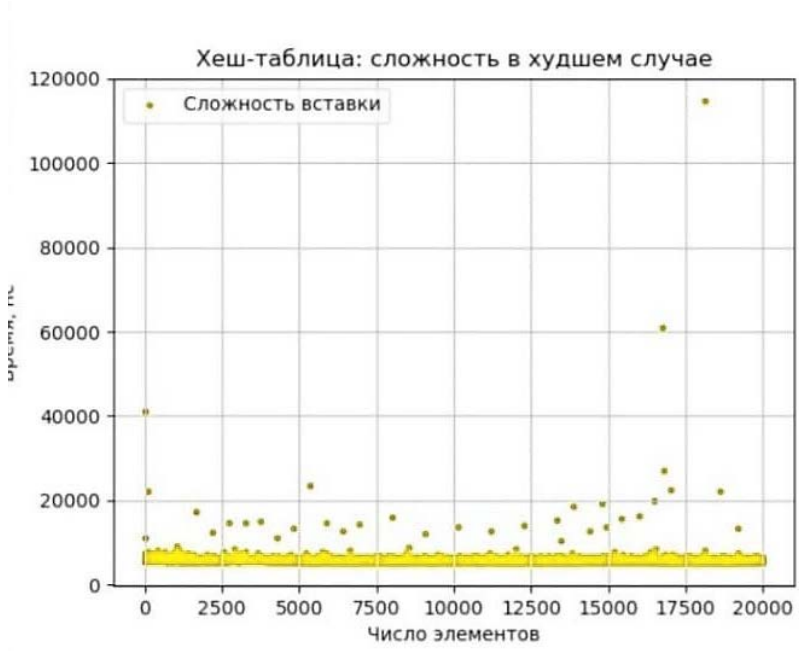


Рис.21. – график сложности удаления из хеш-таблицы (двойное хеширование) в худшем случае

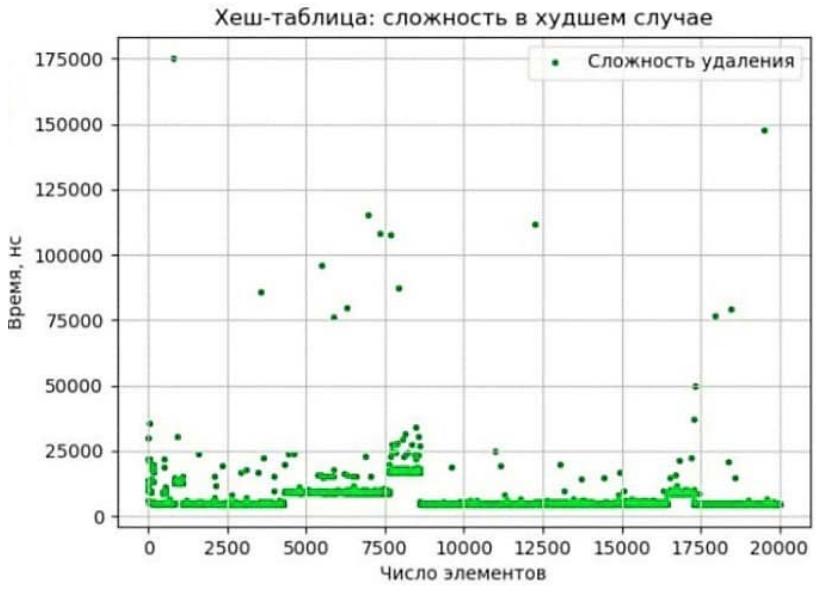
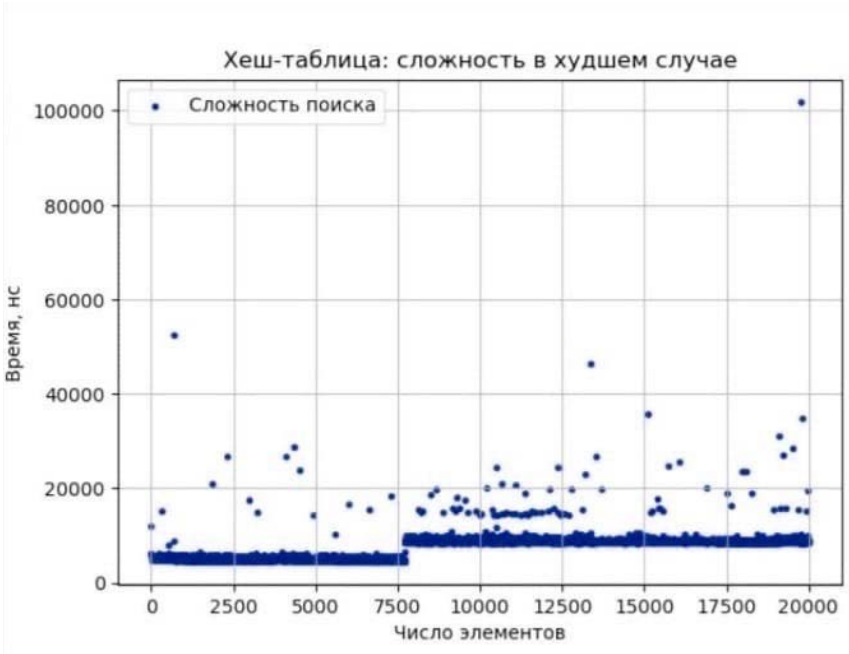


Рис.22. – график сложности поиска в хеш-таблице (двойное хеширование) в худшем случае



Теоретические и полученные результаты совпадают, с учётом погрешностей (ведь сложность в данных случаях напрямую зависит от количества возникших коллизий).

4.2. Красно-чёрное дерево

Таблица 2. Теоретическая сложность операций в RB-дереве

Операция\Случай	**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).

Сложность операций в среднем случае:

Рис.23. – график сложности удаления из хеш-таблицы (двойное хеширование) в среднем случае

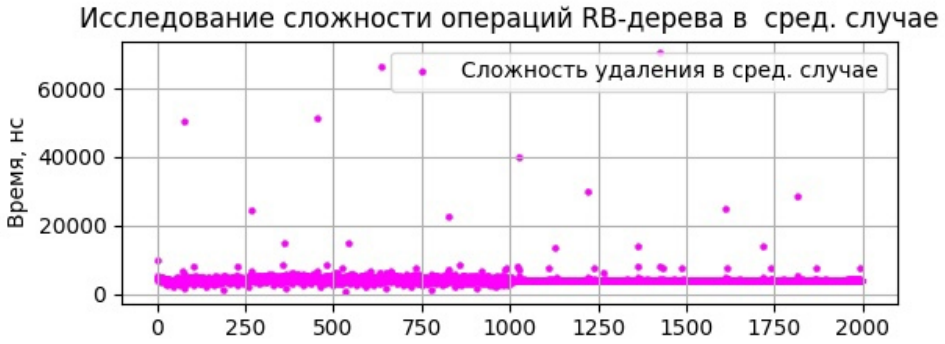
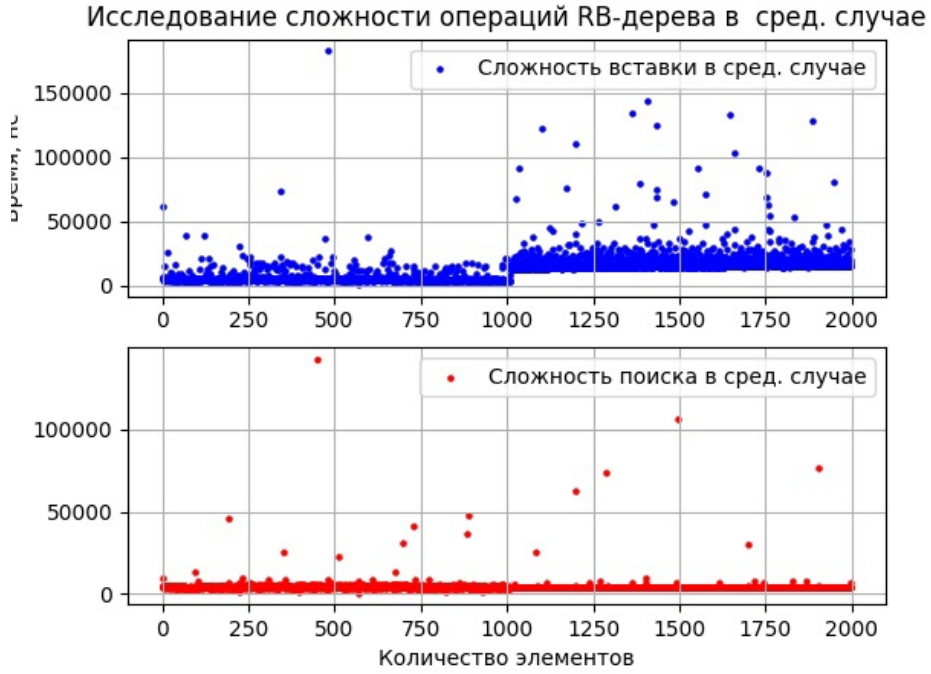


Рис.24. – графики сложности вставки и поиска в хеш-таблицу (двойное хеширование) в среднем случае



Сложность операций в лучшем случае:

Рис.25. – график сложности вставки в хеш-таблицу (двойное хеширование) в лучшем случае

Рис.26. – график сложности удаления из хеш-таблицы (двойное хеширование) в лучшем случае

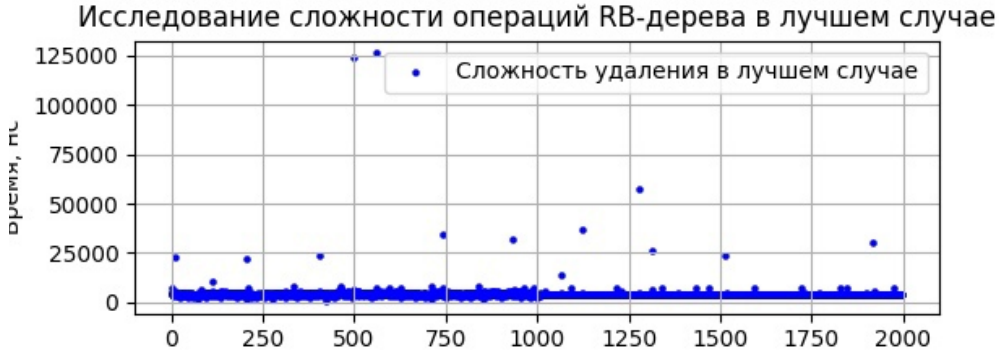
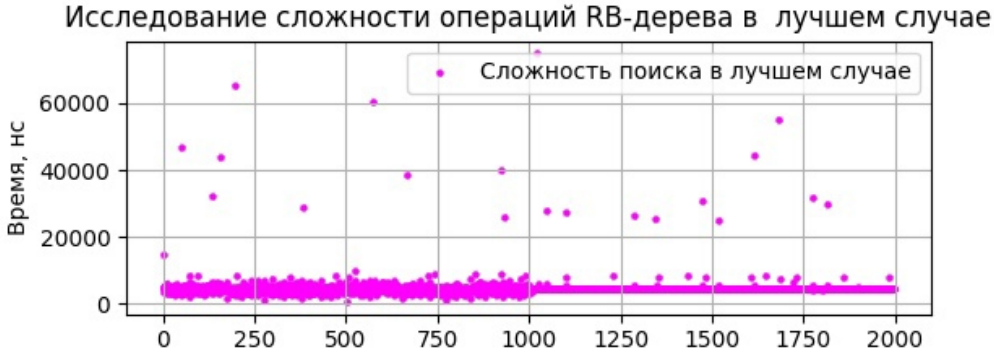


Рис.27. – график сложности поиска в хеш-таблице (двойное хеширование) в лучшем случае



Сложность операций в худшем случае:

Рис.28. – график сложности вставки в хеш-таблицу (двойное хеширование) в худшем случае

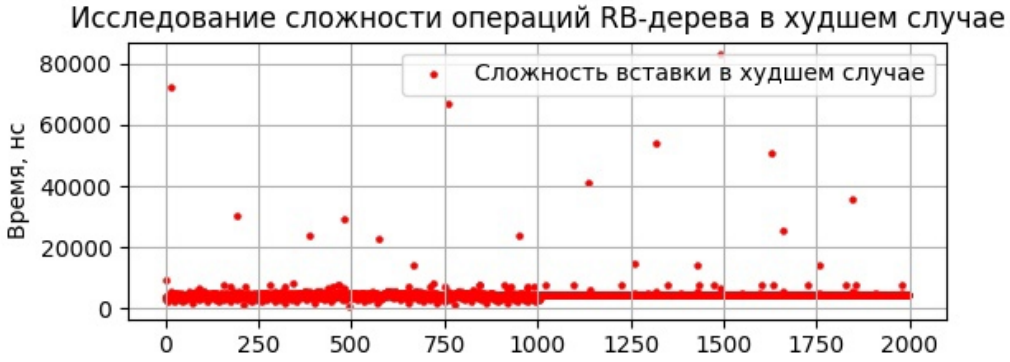
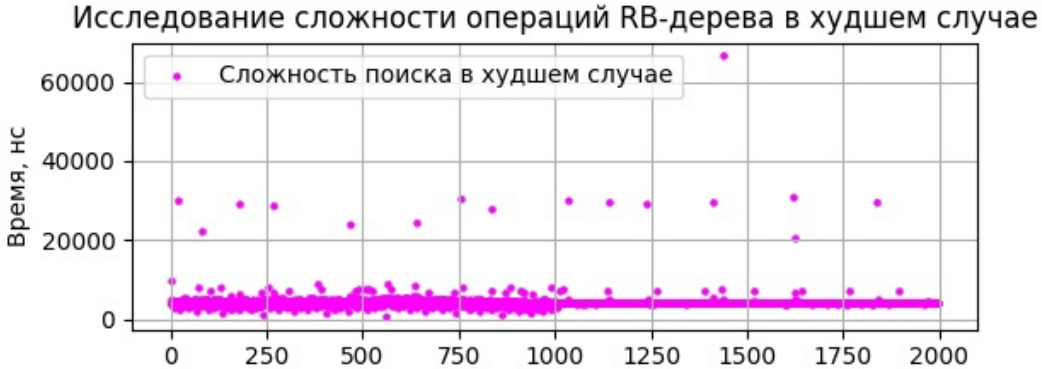


Рис.29. – график сложности удаления из хеш-таблицы (двойное хеширование) в худшем случае

Рис.30. – график сложности поиска в хеш-таблице (двойное хеширование) в худшем случае



Теоретические и полученные результаты совпадают. Скопление точек к началу графика (где количество элементов стремится к нулю) говорит нам о том, что это график функции logn, в недостаточно приближенном масштабе.

5. 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, можно с уверенностью сказать что это очень надёжный и эффективный способ хранения данных (по крайней мере, делая выбор между RB и остальными видами бинарного дерева).

ЗАКЛЮЧЕНИЕ

В ходе курсовой работы было проведено исследование и выполнена реализация структур данных Хеш-таблицы (двойное хеширование) 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. “Алгоритмы построение и анализ” – Т.Кормен, Р. Ривест, Ч. Лейзерсон.
2. https://ru.wikipedia.org/wiki/ Красно-чёрное_дерево
3. https://neerc.ifmo.ru/wiki/index.php?title =Красно-черное_дерево
4. https://graphviz.org/documentation/

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл RBTree.py

```
import graphviz
import functools

BLACK = 'black'
RED = 'red'

@functools.total_ordering

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

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

    def __lt__(self, other):
        return self.key < other.key

    def __bool__(self):
        return self.key is not None

    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: {}, left: {}, right: {}, color: {}, parent: {}'.format(self.key, left, right, self.color, parent)

    def Grandfather(self):
        if self.parent:
            return self.parent.parent
```

```
            return None

        def Uncle(self):
            gp = self.Grandfather()
            if not gp:
                return None
            if gp.left and self.parent == gp.left:
                return gp.right
            if gp.right and self.parent == gp.right:
                return gp.left

    def BreadthSearch(root, dot):
        tree = [root]
        dot.node(str(root.key), color=root.color)
        while tree:
            unit = []
            for element in tree:
                if element.left and element.left.key:
                    dot.node(str(element.left.key),
color=element.left.color)
                    dot.edge(str(element.key), str(element.left.key))
                    unit.append(element.left)
                if element.right and element.right.key:
                    dot.node(str(element.right.key),
color=element.right.color)
                    dot.edge(str(element.key), str(element.right.key))
                    unit.append(element.right)
            tree = unit



    class RBTree:
        def __init__(self, key=None, color=BLACK, parent=None, left=None,
right=None):
            self.root = None
            self.nil = Node()

        def LeftRotate(self, node):
            right = node.right
```

```
        if node.parent:
            if node > node.parent:
                node.parent.right = right
            else:
                node.parent.left = right
        else:
            self.root = node.right
        right.parent = node.parent
        node.parent = right
        node.right = right.left
        if right.left:
            right.left.parent = node
        right.left = node

    def RightRotate(self, node):
        left = node.left
        if node.parent:
            if node > node.parent:
                node.parent.right = node.left
            else:
                node.parent.left = node.left
        else:
            self.root = node.left
        left.parent = node.parent
        node.parent = left
        node.left = left.right
        if left.right:
            left.right.parent = node
        left.right = node

    def _find(self, key, root):
        current = root
        if key == current.key:
            return root
        while current.key and not key == current.key:
            if key > current.key:
                if current.right:
                    current = current.right
                else:
```

```
                return current
            elif key < current.key:
                if current.left:
                    current = current.left
                else:
                    return current
        return current

    def Find(self, key):
        res = self._find(key, self.root)
        if res.key == key:
            return res
        return None

    @staticmethod
    def Next(node):
        if node.right:
            node = node.right
            while node.left:
                node = node.left
            return node
        while node.parent:
            if node.parent > node:
                break
            node = node.parent
        return node.parent

    def Insert(self, key):
        if not self.root:
            self.root = Node(key, BLACK, None, self.nil, self.nil)
            return

        node = self._find(key, self.root)
        if key == node.key:
            return

        if key > node.key:
            node.right = Node(key, RED, node, self.nil, self.nil)
            node = node.right
```

```
        elif key < node.key:
            node.left = Node(key, RED, node, self.nil, self.nil)
            node = node.left

        self.FixTree_INSERT(node)

    def Remove(self, key):
        if not self.root:
            return
        node = self._find(key, self.root)
        if node.key != key:
            return
        if not node.left or not node.right:
            y = node
        else:
            y = self.Next(node)

        if y.left:
            x = y.left
        else:
            x = y.right
        x.parent = y.parent
        if y.parent:
            if y == y.parent.left:
                y.parent.left = x
            else:
                y.parent.right = x
        else:
            self.root = x
        if y != node:
            node.key = y.key
        if y.color == BLACK:
            self.FixTree_REMOVE(x)

    def  FixTree_INSERT(self, node):
        p = node.parent
        if not p:
            node.color = BLACK
            return
```

```
            u = node.Uncle()
            g = node.Grandfather()
            if p.color == BLACK:
                return
            if u and u.color == RED:
                p.color = BLACK
                u.color = BLACK
                g.color = RED
                self.FixTree_INSERT(g)
                return
            if p.right and p.right == node and g.left and p == g.left:
                self.LeftRotate(p)
                node = node.left
            elif p.left and p.left == node and g.right and p == g.right:
                self.RightRotate(p)
                node = node.right
            p = node.parent
            g = node.Grandfather()
            p.color = BLACK
            g.color = RED
            if p.left and p.left == node and g.left and p == g.left:
                self.RightRotate(g)
            else:
                self.LeftRotate(g)

        def FixTree_REMOVE(self, node):
            while self.root and node.parent and node.color == BLACK:
                if node == node.parent.left:
                    brother = node.parent.right
                    if brother.color == RED:
                        brother.color = BLACK
                        node.parent.color = RED
                        self.LeftRotate(node.parent)
                        brother = node.parent.right
                    if     brother.left.color     ==     BLACK     and
brother.right.color == BLACK:
                        brother.color = RED
                        node = node.parent
                    else:
```

```
                        if brother.right.color == BLACK:
                            brother.left.color = RED
                            brother.color = BLACK
                            self.RightRotate(brother)
                            brother = node.parent.right
                        brother.color = node.parent.color
                        node.parent.color = RED
                        brother.right.color = RED
                        self.LeftRotate(node.parent)
                        node = self.root
                elif node == node.parent.right:
                    brother = node.parent.left
                    if brother.color == RED:
                        brother.color = BLACK
                        node.parent.color = RED
                        self.RightRotate(node.parent)
                        brother = node.parent.left
                    if brother.right.color == BLACK and
brother.left.color == BLACK:
                        brother.color = RED
                        node = node.parent
                    else:
                        if brother.left.color == BLACK:
                            brother.right.color = BLACK
                            brother.color = RED
                            self.LeftRotate(brother)
                            brother = node.parent.left
                        brother.color = node.parent.color
                        node.parent.color = BLACK
                        brother.left.color = BLACK
                        self.RightRotate(node.parent)
                        node = self.root
            node.color = BLACK

    def main():
        nodes = list(map(int, input().split()))
        rb_tree = RBTree()
        for index, node in enumerate(nodes):
            dot = graphviz.Digraph()
```

```
        rb_tree.Insert(node)
        BreadthSearch(rb_tree.root, dot)
        dot.render('g{}.gv'.format(index))
main()
```

Файл DoubleHashing.py

```
import math
import sys
sys.setrecursionlimit(100000) #max глубина рекурсии

class DoubleHashing:
    def __init__(self, size):
        self.Size = size
        self.FixSize()
        self.Hash_Table = [[], ] * self.Size
        self.Overflow = 0

    def Hashing1(self, key): #Метод середины квадрата
        key *= key
        key >>= 11  # Отбрасываем 11 младших бит
        return key % 1024 # Возвращаем 10 младших бит

    def Hashing2(self, key):  # Метод деления
        return key % self.Size

    def Insert(self, key, data, i=0):
        percent_overflow = 100 * (2/3)
        percent_one = 100
        if i >= self.Size or self.Overflow >= percent_overflow:
            self.Size = self.Size * 2
            self.FixSize()
            previous_Hash_Table = self.Hash_Table
            self.Hash_Table = [[], ] * self.Size
            self.Overflow = 0
            for key_index in range(len(previous_Hash_Table)):
                if previous_Hash_Table[key_index]:
                    self.Insert(previous_Hash_Table[key_index][0],
previous_Hash_Table[key_index][1], 0)
            self.Insert(key, data, 0)
        else:
            index = (self.Hashing1(key) + i * self.Hashing2(key)) %
self.Size
```

```
                if self.Hash_Table[index] and self.Hash_Table[index] !=
[None]:
                    if self.Hash_Table[index][0] == key:
                        self.Hash_Table[index] = [key, data]
                    else:
                        i += 1
                        self.Insert(key, data, i)
                else:
                    self.Hash_Table[index] = [key, data]
                    self.Overflow += round( percent_one / self.Size)


        def Remove(self, key, i=0):
            if i < self.Size:
                index = (self.Hashing1(key) + i * self.Hashing2(key)) %
self.Size
                if self.Hash_Table[index]:
                    if self.Hash_Table[index][0] == key:
                        self.Hash_Table[index] = [None]
                    else:
                        i += 1
                        self.Remove(key, i)

        def InPrimes(self) -> int:
            if self.Size == 1 or self.Size == 0:
                return 0
            for i in range(2, self.Size // 2):
                if self.Size % i == 0:
                    return 0
            return 1

        def FixSize(self):
            if self.Size % 2 == 0:
                self.Size += 1
            while not self.InPrimes():
                self.Size += 2
            return self.Size
```

```
        def FindKey(self, key, i=0):
            index = (self.Hashing1(key) + i * self.Hashing2(key)) %
self.Size  # DF
            if i >= self.Size or not self.Hash_Table[index]:
                return "None"
            else:
                if self.Hash_Table[index][0] == key:
                    return self.Hash_Table[index][1]
                else:
                    i += 1
                    return self.FindKey(key, i)


    from random import randint
    import time

    hash = DoubleHashing(100)

    x = [i for i in range(20000)]

    ins=[]
    fnd=[]
    dlt=[]

    for i in x:
        start = time.time_ns()
        hash.Insert(randint(500, 1000)*i, ':)')
        end = time.time_ns()
        print(start - end)

    '''
    for i in x:
        start = time.time_ns()
        hash.Remove(20000)
        #tree.delete(tree.root, 80)
        end = time.time_ns()
        dlt.append(end - start)
    '''
```