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. 0303	________________	Беззубов Д.В.
Преподаватель	________________	Иванов Д.В.

Санкт-Петербург

2022

ЗАДАНИЕ

Тема работы: Исследование структур данных RB-дерево и хэш-таблица (двойное хэширование)

Исходные данные:

RB-дерево vs Хеш-таблица (двойное хеширование).

Исследование

"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.

АННОТАЦИЯ

Курсовая работа заключается в исследовании двух разных структур данных: RB-дерево (Чёрно-красное дерево) и Хэш-таблица (двойное хэширование). В программном коде были реализованы классы RB_tree и HashTable, которые представляют из себя реализацию двух упомянутых выше структур на языке программирования Python. С помощью методов происходит добавление, удаление, поиск элементов в структуре данных. Остальные вспомогательные методы необходимы для корректной работы и реализации структур данных.

Содержание

ВВЕДЕНИЕ

Цель: Реализовать структуры данных (RB-дерево и хэш-таблица(двойное хэширование)) и исследовать их на сложность работы алгоритмов поиска, вставки и удаления элементов.

Для выполнения поставленной задачи необходимо:

1) Реализовать структуру данных RB-дерево и соответствующие методы.

2) Реализовать структуру данных хэш-таблица (двойное хэширование) и соответствующие методы.

3) Определить время работы двух структур при разных входных данных.

4) Построить графики зависимости времени работы алгоритмов от кол-ва элементов.

5) Сравнить время работы алгоритмов для данных структур.

СТРУКТУРА ДАННЫХ RB-ДЕРЕВО

Структура класса *RB_tree*

Для реализации класса *RB_tree* был реализован класс *Node*.

Класс *Node* – класс, представляющий узлы чёрно-красного дерева.

Поля данного класса:

1. *key* – поле, хранящее ключ узла
2. *left, right* – поля, хранящие ссылки на левого и правого ребенка узла соответственно
3. *color* – поле, хранящее цвет узла
4. *parent* – поле, хранящее ссылку на родительский узел, по умолчанию – *None.*

Для класса *Node* перегружена функция *__str__(self)* – при печати узла выводится информация о детях, родителе узла, а так же цвете.

Класс *RB_tree*

Единственное поле данного класса – поле *root,* которое хранит указатель на корень дерева.

Реализованы следующие методы:

- *def insert(self, key)* – метод, принимающий ключ на вход. Метод добавляет узел в дерево, в случае, если нарушаются свойства – вызывает метод *fix_insert(self, node).*
- *def fix_insert(self, node)* – метод, восстанавливающий свойства КЧ-дерева после вставки узла
- *def search(self, key)* – метод, осуществляющий поиск по дереву и возвращающий искомый узел
- *def delete_elem(self, key)* – метод, удаляющий узел по ключу.
- *def fix_delete(self, x)* – метод восстанавливающий свойства КЧ-дерева после удаления узла

- *def left_rotate(self, node), def right_rotate(self, node)* – методы, осуществляющие левый и правый повороты соответственно
- *def transplant(self, node, new_node)* – метод для перемещения поддеревьев в дереве. Заменяет одно поддерево, являющееся дочерним по отношению к своему родителю другим поддеревом.
- *def minimum(self, node)* – функция поиска минимального узла.

Сложность работы чёрно-красного дерева.

Как известно, сложность работы вставки, поиск и удаление элемента в чёрно-красное дерево при любом случае равно O(log n).

При помощи встроенной библиотеки *time* определим время работы структуры во время вставки элементов, при разном количестве входных данных. Результат измерений представлен в табл. 1.

Таблица 1 - Время работы вставки элемента в RB-дереве.

Количество элементов (n)	Время работы (с)
250	0,0000035
500	0,0000041
750	0,0000044
1000	0,0000044
1250	0,0000045
1500	0,0000048
1750	0,0000049

При помощи результатов представленных в табл. 1 построим график зависимости времени работы от количества элементов (рис. 1) и сравним с графиком теоретического значения сложности алгоритма (рис. 2).

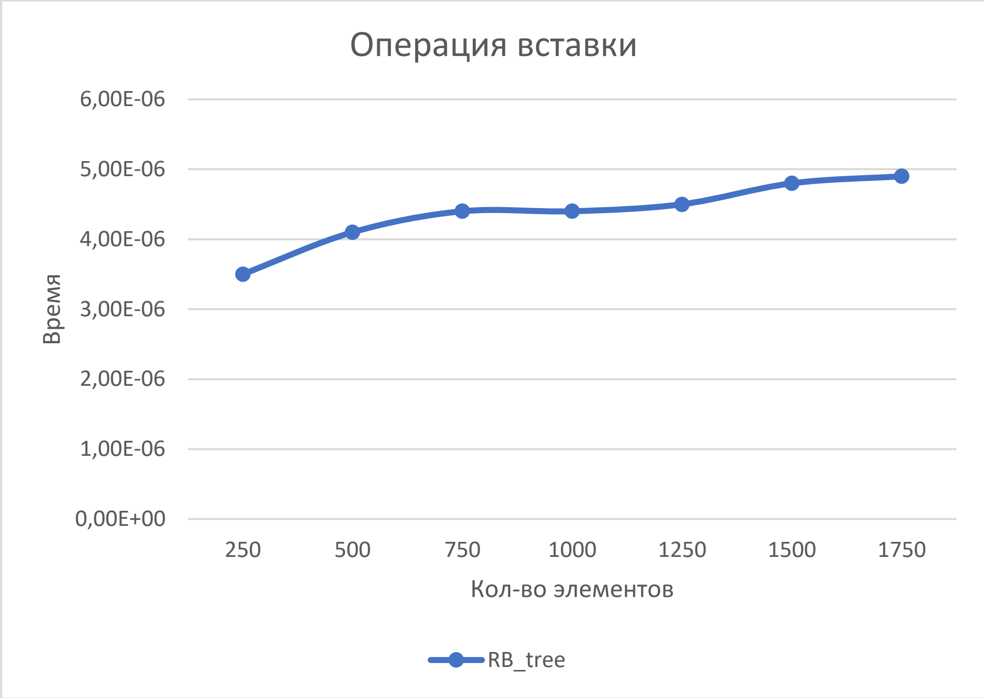

Рисунок 1 — Практическое время работы вставки элементов.

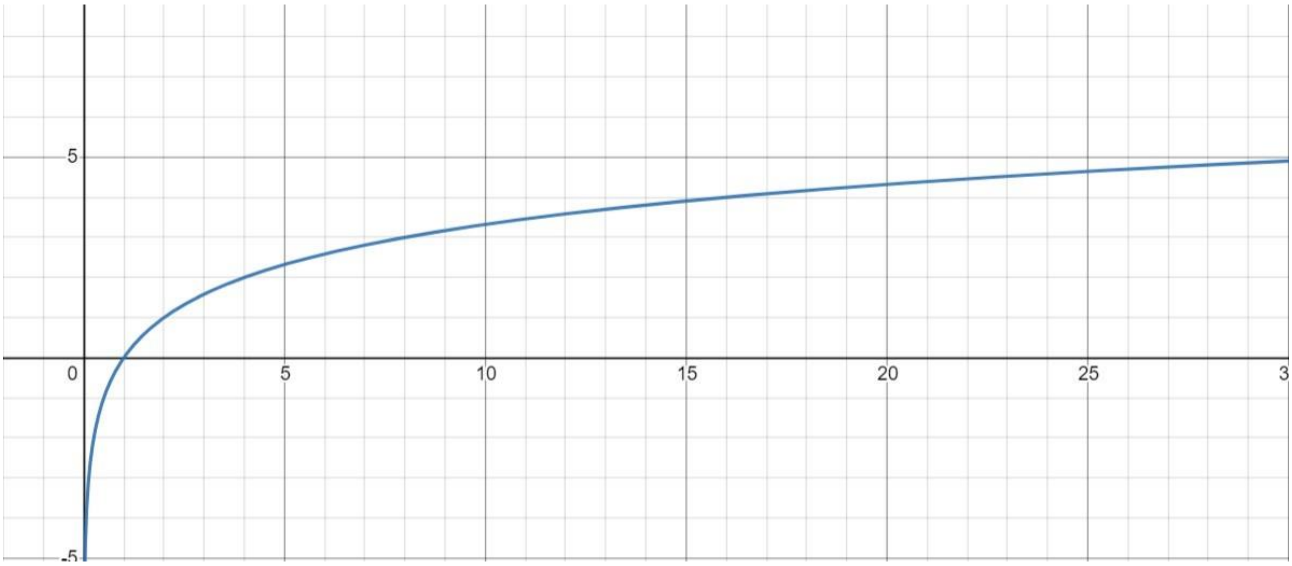

Рисунок 2 — Теоретическое время работы алгоритма вставки

Сравнив данные графики, можно сделать вывод, что теоретическое и практическое время работы вставки элементов в чёрно-красном дереве совпадают. Однако наблюдаются некоторые различия, связанные, в первую очередь с коэффициентами, которые в О-семантике принято опускать. Более того, данные измерения нельзя считать абсолютно точными, т.к. на время выполнения процессов влияют различные факторы, связанные с ОС и работой компьютера в целом. Но они позволяют оценить поведение структуры данных, которое совпало с теоретическим.

Проведём аналогичные измерения для поиска элемента и удаления элемента. Измерения представлены табл. 2 и табл. 3 соответственно.

Таблица 2 - Время работы поиска элемента в RB-дереве.

Количество элементов (n)	Время работы (с)
250	0,000021
500	0,000045
750	0,000062
1000	0,000083
1250	0,000089
1500	0,000088
1750	0,000091

При помощи результатов представленных в табл. 2 построим график зависимости времени работы от количества элементов (рис. 3) и сравним с графиком теоретического значения сложности алгоритма (рис. 4).

Рисунок 3 - Практическое время работы поиска элемента.

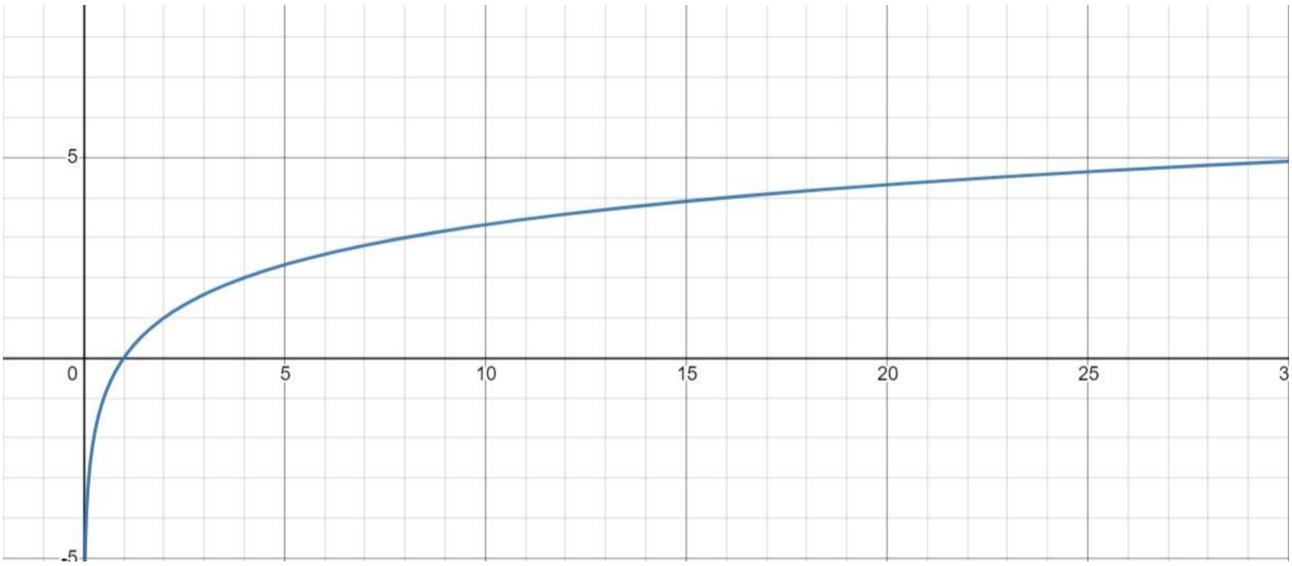

Рисунок 4 — Теоретическое время работы поиска элемента.

В данном случае время так же совпало с ожидаемыми результатами, отличия от O(log n) возникают с причинами, описанными выше.

Таблица 3 - Время работы удаления элемента в RB-дереве.

Количество элементов (n)	Время работы (с)
250	0,000032
500	0,000038
750	0,000041
1000	0,000041
1250	0,000043
1500	0,000045
1750	0,000047

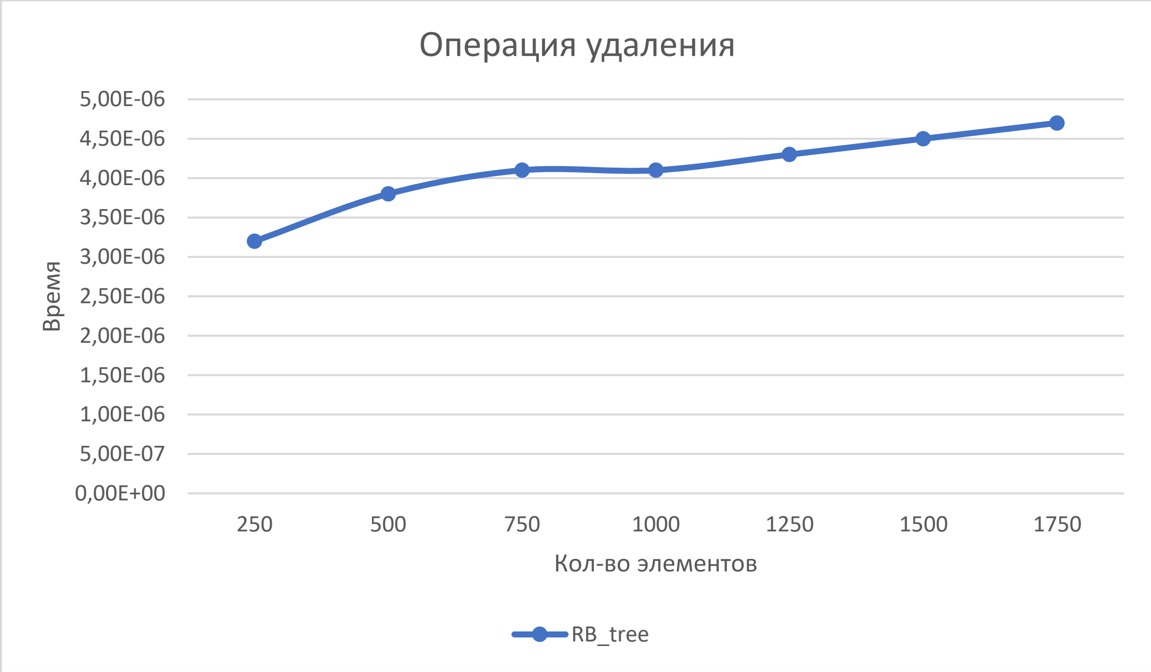

Рисунок 5 — Практическое время работы удаления элемента

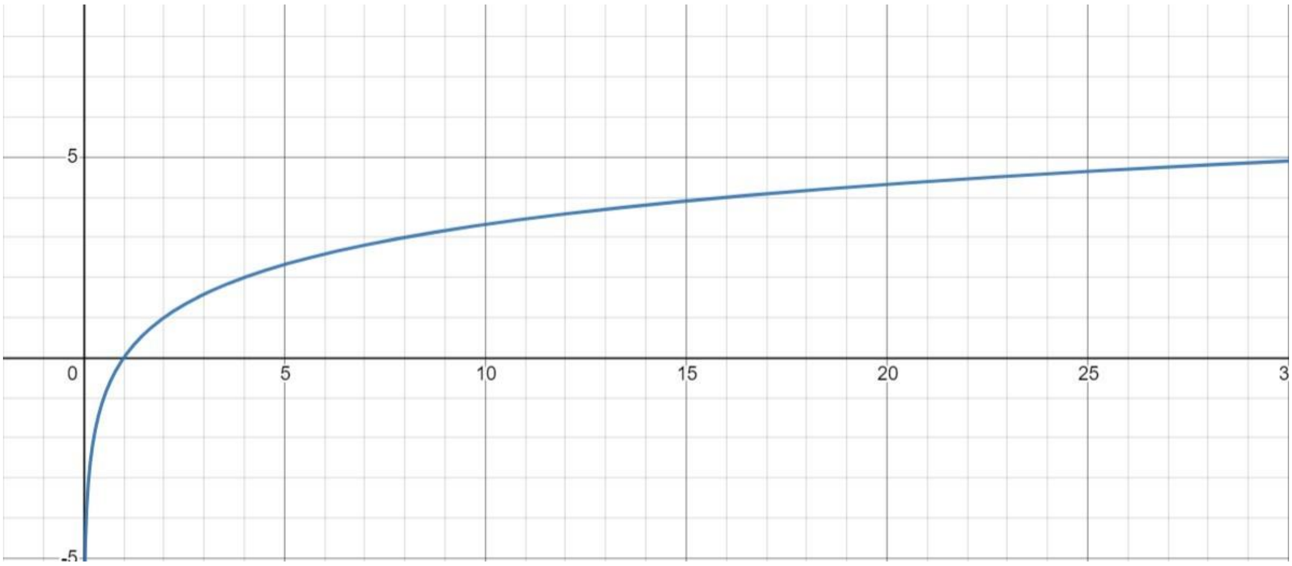

Рисунок 6 — Теоретическое время работы удаления элемента

В случае удаления элемента аналогично можно заметить закономерность, что при переходе от 250 до 500 элементов время растет быстрее, чем при дальнейших измерениях. Время при удалении узла в дереве на 1000 узлов совпало со временем при 750 узлах, но данная погрешность возникает в связи с точностью измерения времени. В итоге результат совпал с ожидаемым.

Исходя из представленных выше результатов измерений, можно сделать вывод, что сложность работы реализованных алгоритмов вставки, поиска и удаления элемента совпадает с теоретическими. Таким образом, было подтверждено теоретическое значение сложности алгоритмов, равное O(log n). Данное значение будет в дальнейшем использоваться для сравнения со сложностью работы алгоритмов вставки, поиска и удаления элемента в хэш-таблице.

СТРУКТУРА ДАННЫХ ХЭШ-ТАБЛИЦА (ДВОЙНОЕ ХЭШИРОВАНИЕ)

Структура класса *HashTable*

Класс *HashTable* содержит следующие поля:

- *size* – поле хранит размер хэш-таблицы
- *list* – поле, которое хранит элементы хэш-таблицы
- *count* – поле, содержащее кол-во элементов в таблице

Методы:

- *def __hash_1(self, key) -> int* – приватный метод, возвращающий значение первой хэш-функции
- *def __hash_2(self, key) -> int* – приватный метод, возвращающий значение второй хэш-функции
- *def add(self, key)* – метод, обеспечивающий вставку элемента в таблицу. Если таблица заполнена – будет проброшено исключение
- *def search(self, key)* – метод, реализующий поиск по таблице
- *def delete_elem(self, key)* – метод, удаляющий элемент из таблицы (помечает удаленную ячейку как *DELETED)*
- *def __getPrime(self) -> int* – приватный метод, возвращающий наибольшее простое число, которое при этом меньше размера таблицы
- *def __isPrime(self, value) -> bool* – приватный метод, проверяющий является ли число простым

Сложность работы хэш-таблицы (двойное хэширование)

В отличие от чёрно-красного дерева у хэш-таблицы меняется сложность в зависимости от входных данных. В худшем случае сложность алгоритма будет O(n), а в лучшем O(const). Проверим все случаи.

При помощи встроенной библиотеки time определим время работы структуры во время вставки элементов, при разном количестве входных данных, причём все элементы не требуют использования второй хэш-функции.

Результат измерений представлен в табл. 4.

Таблица 4 - Время работы вставки элемента в хэш-таблицу.

Количество элементов (n)	Время работы (с)
250	0.0000014
500	0.0000013
750	0.0000015
1000	0.0000015
1250	0.0000015
1500	0.0000013
1750	0.0000014

При помощи результатов представленных в табл. 4 построим график зависимости времени работы от количества элементов (рис. 7) и сравним с графиком теоретического значения сложности алгоритма (рис. 8).

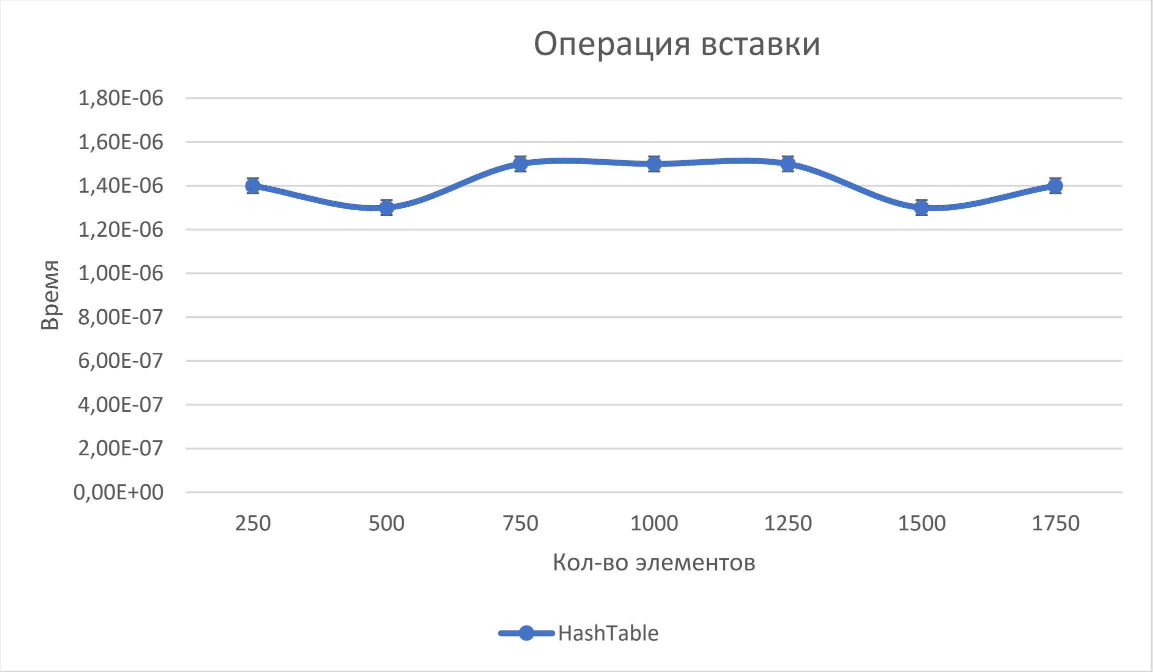

Рисунок 7 — Практическое время работы алгоритма вставки.

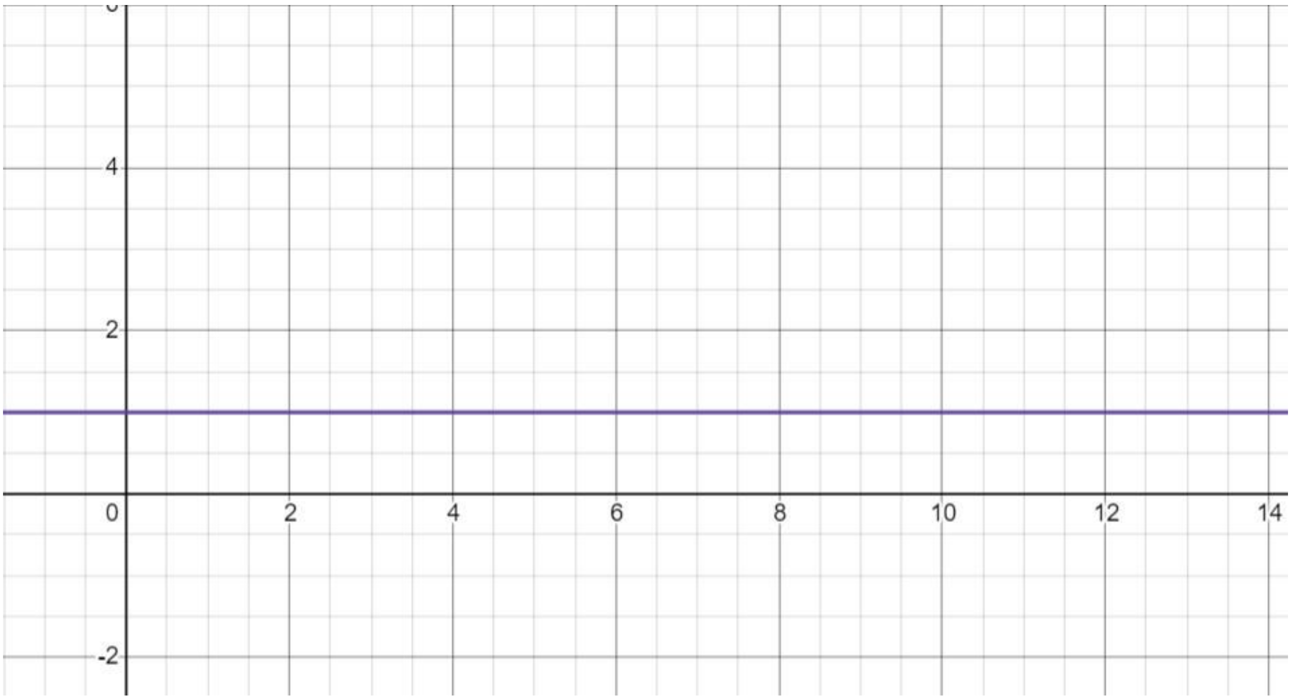

Рисунок 8 — Теоретическое время работы алгоритма вставки.

Сравнив данные графики, можно сделать вывод, что теоретическое и практическое время работы вставки элементов в хэш-таблице совпадают, вставка осуществляется за постоянное время в пределах погрешности. Причины возникновения данной погрешности описаны в измерениях для операций с RB-tree.

Проведём аналогичные измерения для вставки элемента в наихудшем случае. Измерения представлены табл. 5

Таблица 5 - Время работы вставки элемента в хэш-таблицу.

Количество элементов (n)	Время работы (c)
250	0.000018
500	0.000019
750	0.000019
1000	0.000020
1250	0.000021
1500	0.000023
1750	0.000022

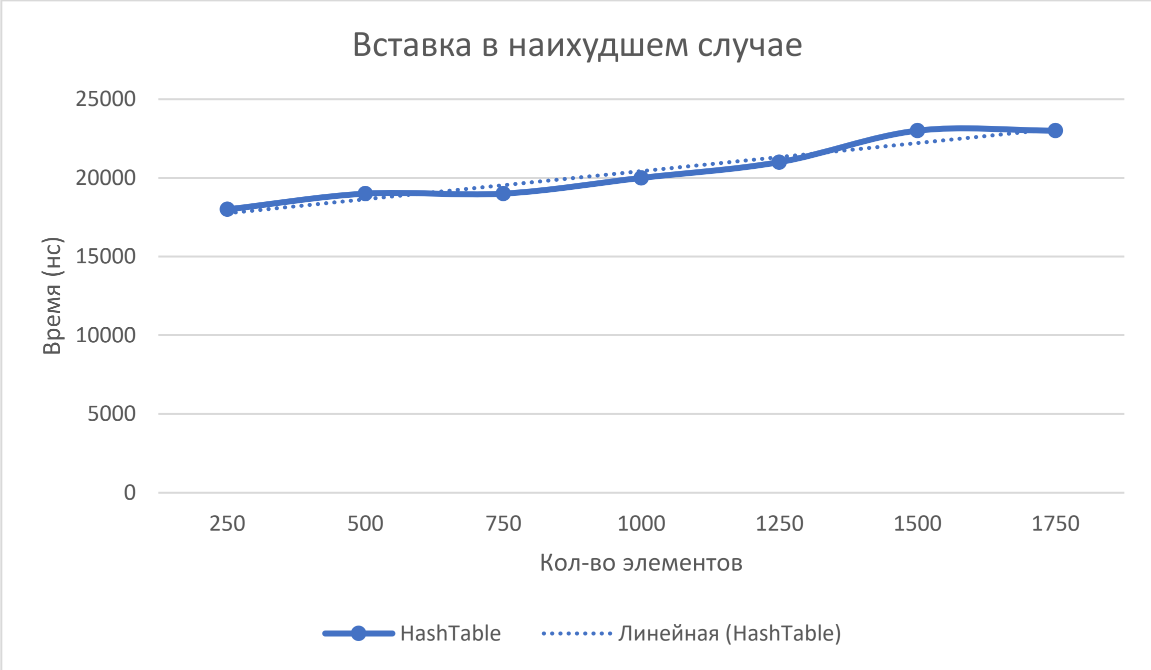

Рисунок 9 — Практическое время работы алгоритма вставки.

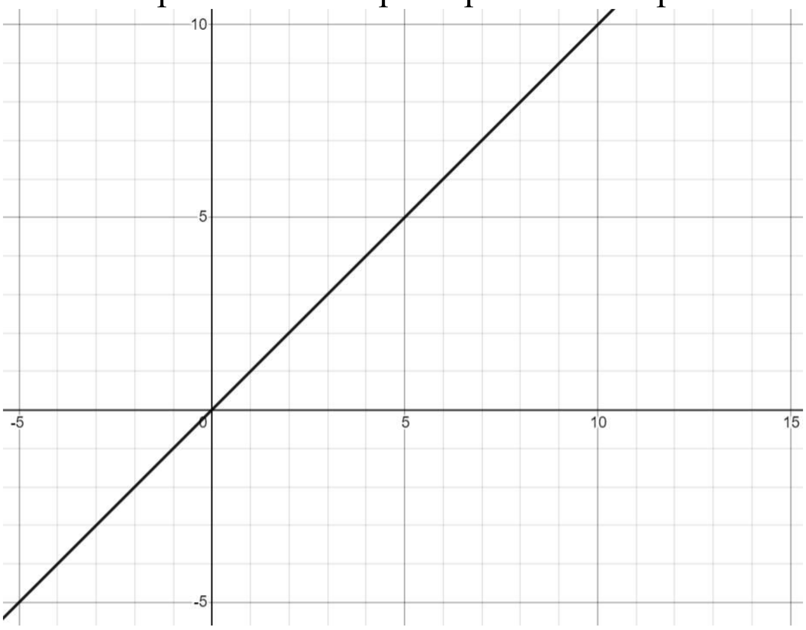

Рисунок 10 — Теоретическое время работы алгоритма.

В данном случае время растет, но при этом заметно медленнее, чем функция $f(x) = x$, это связано с тем, что в O-семантике опускаются коэффициенты, таким образом, наблюдаемый результат совпал с ожидаемым.

Проведём аналогичные расчёты для алгоритма поиска, поскольку алгоритм удаления включает в себя алгоритм поиска. Измерения представлены в табл. 6

Таблица 6 - Время работы поиска элемента в хэш-таблицу.

Количество элементов (n)	Время работы (с)
250	0,0000012
500	0,0000013
750	0,0000013
1000	0,0000014
1250	0,0000012
1500	0,0000014
1750	0,0000013

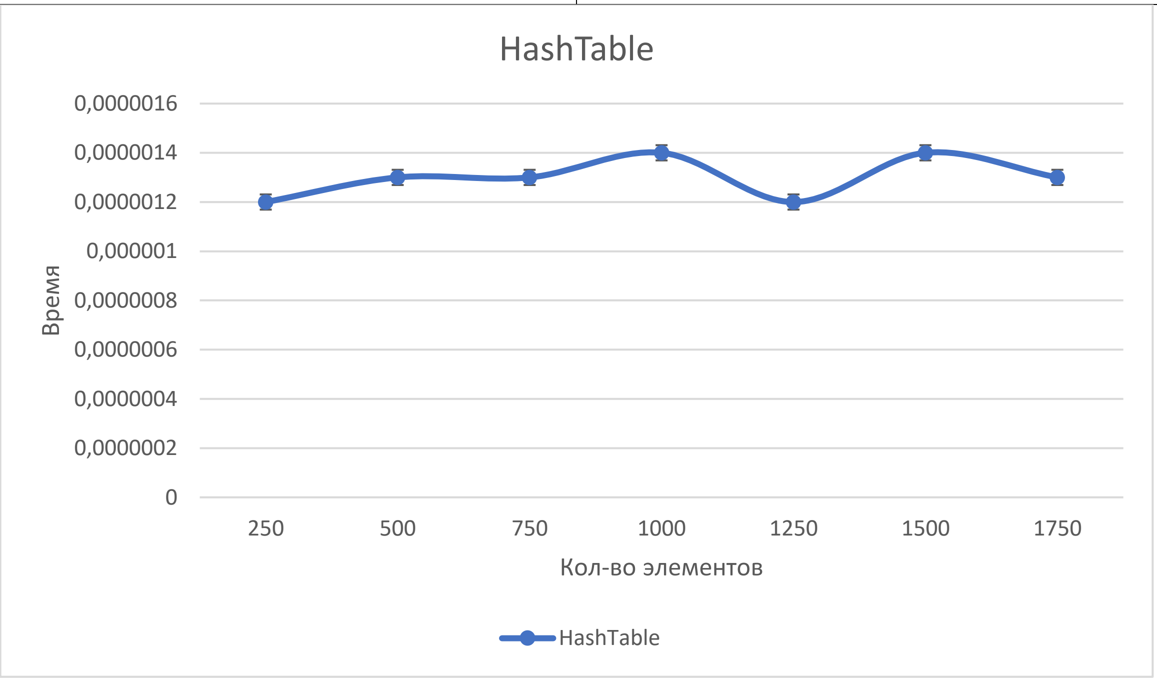

Рисунок 11 — Практическое время работы алгоритма поиска

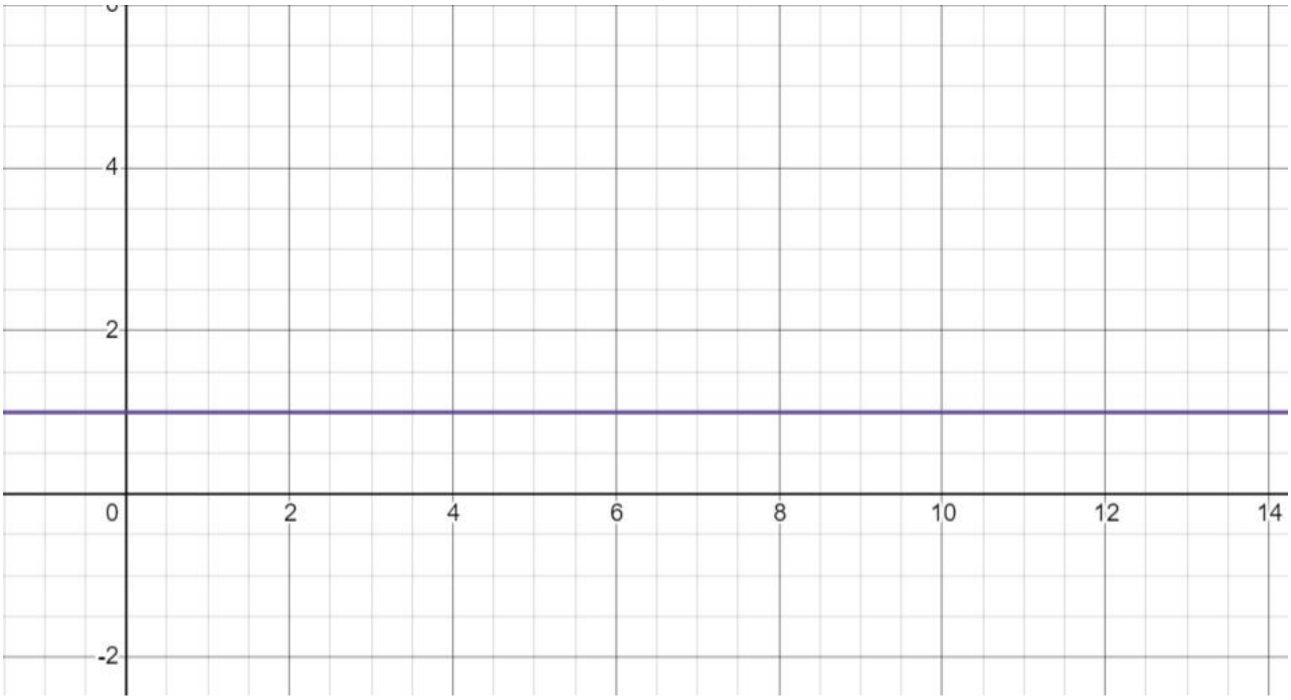

Рисунок 12 — Теоретическое время работы алгоритма поиска.

Определим зависимость времени работы алгоритма поиска в худшем случае. Результаты измерения представлены в табл. 7.

Таблица 7 — Время работы алгоритма вставки.

Количество элементов (n)	Время работы (с)
250	0.000085
500	0.000088
750	0.000088
1000	0.000089
1250	0.000089
1500	0.000090
1750	0.000090

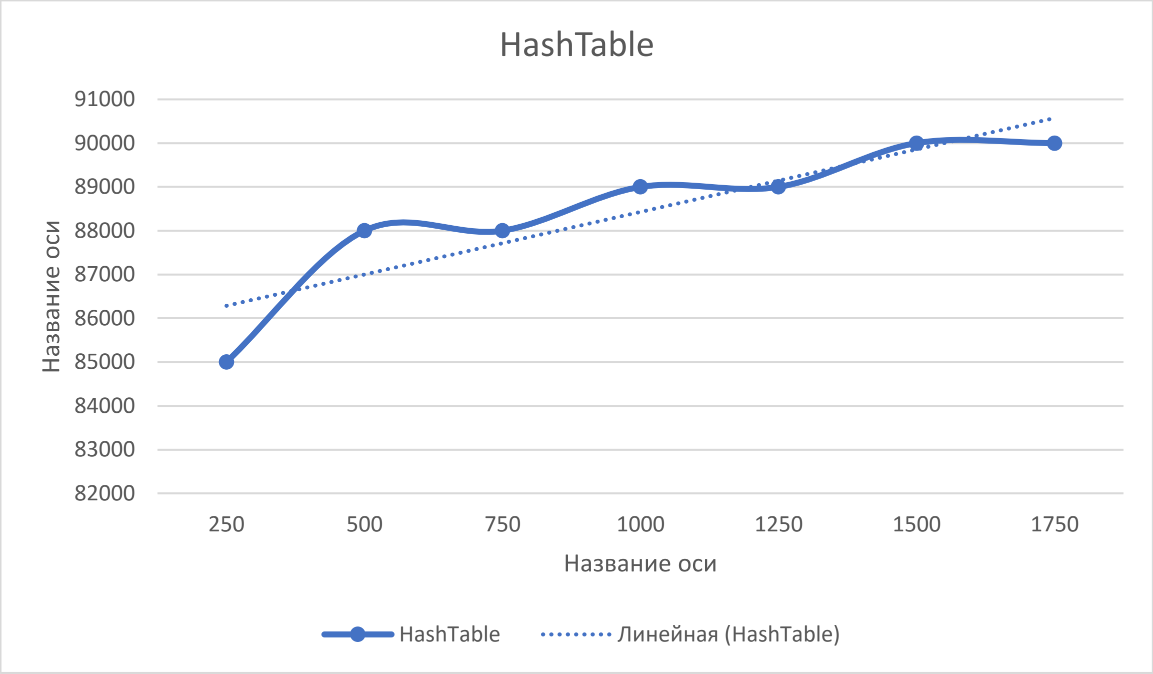

Рисунок 13 — практическое время работы алгоритма поиска.

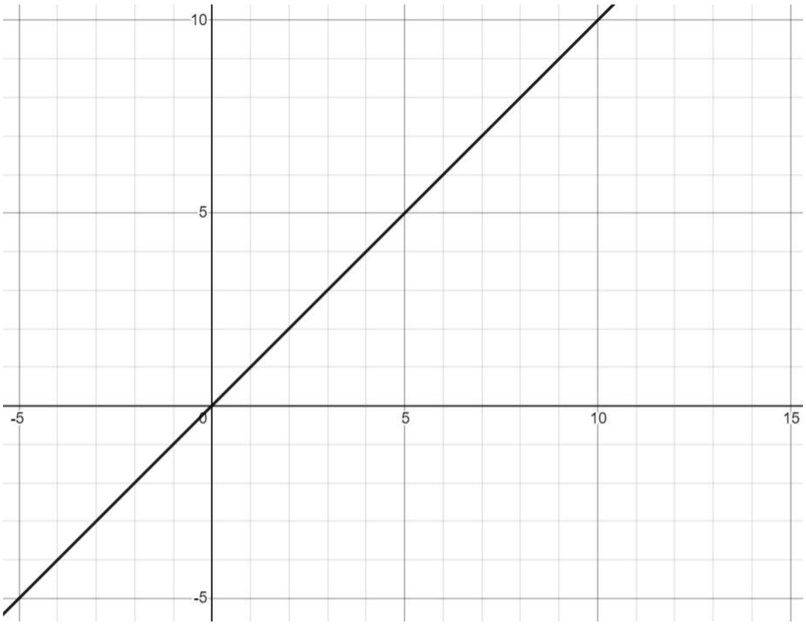

Рисунок 14 — теоретическое время работы алгоритма поиска.

Сравнив данные графики, можно сделать вывод, что теоретическое и практическое время работы поиска элементов в хэш-таблице совпадают, однако наблюдаются небольшие различия. Данные различия связаны с погрешностью интерполирования и коэффициентами, которые появляются в практическом подсчёте, при этом в теоретическом отбрасываются.

СРАВНЕНИЕ СТРУКТУР ДАННЫХ

После того, как мы получили экспериментальные значения для КЧ-дерева и Хеш-таблицы с двойным хешированием, построим графики и сравним время работы алгоритмов для данных структур.

На рисунке 15 представлены графики для алгоритмов вставки:

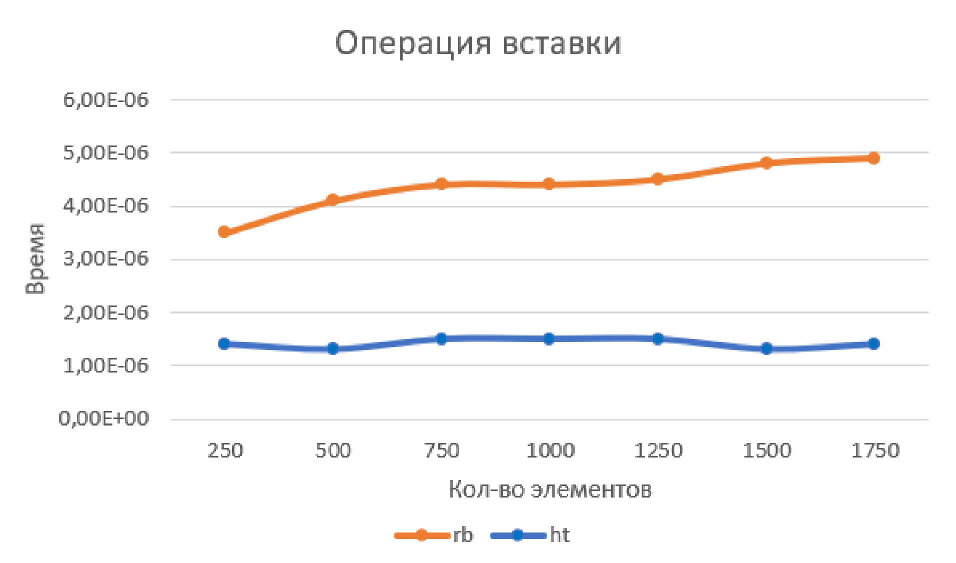

Рисунок 15 – Сравнение времени работы алгоритмов вставки

В данном случае заметим, что вставка в хеш-таблицу происходит заметно быстрее, чем в КЧ-дерево, но данные графики отражают сложность работы алгоритмов в среднем случае.

Аналогично выглядит график оценки времени работы алгоритма поиска. Совмещенные результаты измерений представлены на рисунке 16.

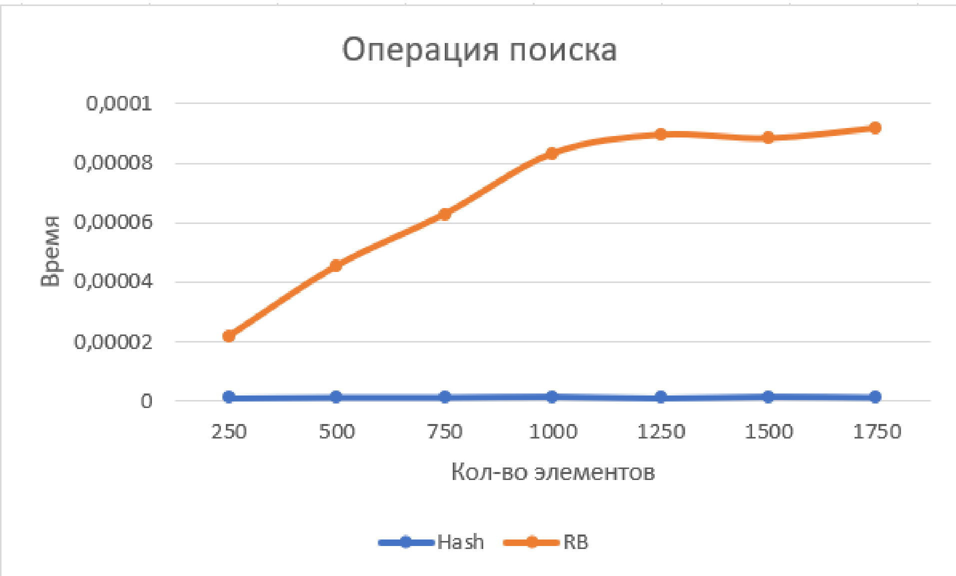

Рисунок 16 – Сравнение скорости поиска элемента

При этом в случае с удалением элемента время выполнения алгоритмов практически совпадает. Но все-таки удаление элемента из КЧ-дерева происходит медленнее, чем из Хеш-таблицы.

Результаты сравнения представлены на рисунке 17.

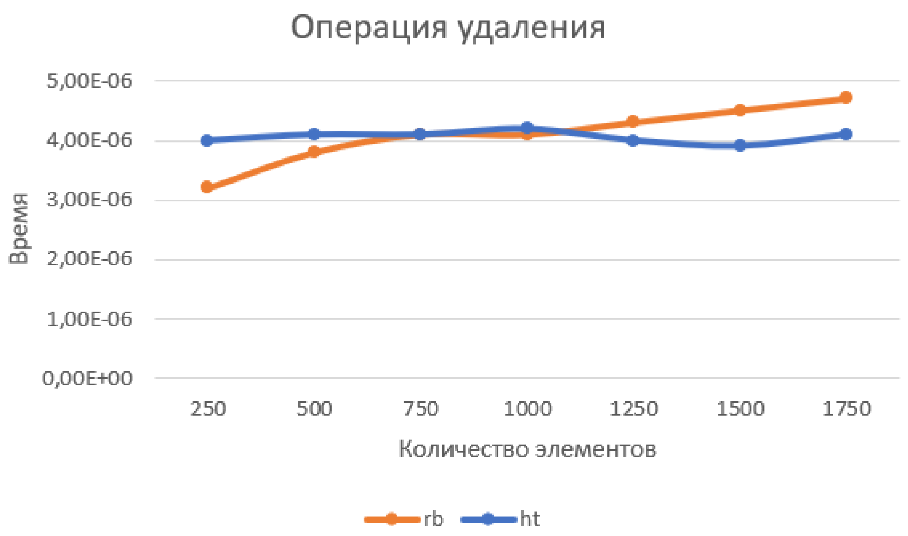

Рисунок 17 – Сравнение скорости удаления элементов

Несмотря на то, что в данных сравнениях время выполнения алгоритмов поиска, вставки и удаления для Хеш-таблицы меньше, у КЧ-дерева есть преимущество в том, что оценка времени выполнения всех алгоритмов O(log n), при том, что в наихудшем случае для Хеш-таблицы данная оценка становится O(n), таким образом при большом n время для Хеш-таблицы будет возрастать заметно быстрее, чем для КЧ-дерева.

ЗАКЛЮЧЕНИЕ

Были исследованы две разных структуры данных: RB-дерево (Красно-черное дерево) и Хеш-таблица (с двойным хешированием). В программном коде были реализованы классы *RB_tree* и *HashTable*, которые представляют из себя реализацию двух упомянутых выше структур на языке программирования Python.

Полученные значения сложности для разных структур данных позволяют сравнить сложность алгоритмов красно-чёрного дерева и хеш-таблицы с двойным хешированием.

Сложность алгоритма чёрно-красного дерева равно O(log n) в любом случае, а сложность хэш-талицы равна O(const) в лучшем случае и O(n) в худшем. Для стабильной и предсказуемой работы лучше использовать
Красно-черное дерево, однако при определённых входных данных хэш-таблица покажет лучший результат при прочих равных.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: rb_tree.py

```
import sys

BLACK = 'black'
RED = 'red'

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

class RB_tree:
    def __init__(self):
        self.root = None
        self.TNULL = Node(None)
        self.TNULL.left = None
        self.TNULL.right = None
        self.root = self.TNULL

    def insert(self, key):
        if self.root == self.TNULL:
            self.root = Node(key)
            self.root.left = self.TNULL
            self.root.right = self.TNULL
        else:
            current = self.root
            while current != self.TNULL:
                if key < current.key:
                    if current.left == self.TNULL:
                        new_node = Node(key, RED, parent=current)
                        new_node.left = self.TNULL
                        new_node.right = self.TNULL
                        current.left = new_node
                        break
                    current = current.left
                else:
                    if current.right == self.TNULL:
                        new_node = Node(key, RED, parent=current)
                        new_node.left = self.TNULL
                        new_node.right = self.TNULL
                        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:
            grandparent = node.parent.parent

            if grandparent is None:
                return

            if node.parent == grandparent.left:
                uncle = grandparent.right

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

                else:  # дядя красный
                    uncle.color = BLACK
                    node.parent.color = BLACK
                    grandparent.color = RED
                    node = grandparent
            else:  # если родитель нового узла правый сын
                uncle = grandparent.left

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

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

    def search(self, key) -> Node:
        root = self.root
        while root != self.TNULL and root.key != key:
            if key > root.key:
                root = root.right
            else:
                root = root.left
```

```
        return root

    def delete_elem(self, key):
        node = self.search(key)

        if not node:
            return

        y = node
        y_original_color = y.color

        if node.left == self.TNULL:
            x = node.right
            self.transplant(node, node.right)
        elif node.right == self.TNULL:
            x = node.left
            self.transplant(node, node.left)
        else:
            y = self.minimum(node.right)
            y_original_color = y.color
            x = y.right
            if y.parent == node:
                x.parent = y
            else:
                self.transplant(y, y.right)
                y.right = node.right
                y.right.parent = y

            self.transplant(node, y)
            y.left = node.left
            y.left.parent = y
            y.color = node.color
        if y_original_color == BLACK:
            self.fix_delete(x)

    def fix_delete(self, x):
        while x != self.root and x.color == BLACK:
            if x == x.parent.left:
                s = x.parent.right
                if s.color == RED:
                    s.color = BLACK
                    x.parent.color = RED
                    self.left_rotate(x.parent)
                    s = x.parent.right

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

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

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

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

    # if self.root.parent:
    #     self.root.parent = None

def left_rotate(self, node): # node - отец нового элемента
    new_node = node.right
    parent = node.parent

    node.right = new_node.left # Lb
    if node.right:
        node.right.parent = node

    new_node.left = node
    node.parent = new_node

    if not parent:
        self.root = new_node

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


def right_rotate(self, node):
```

```
        new_node = node.left
        parent = node.parent

        node.left = new_node.right
        if node.left:
            node.left.parent = node

        new_node.right = node
        node.parent = new_node

        if not parent:
            self.root = new_node

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

    def transplant(self, node, new_node):
        if node.parent == None:
            self.root = new_node
        elif node == node.parent.left:
            node.parent.left = new_node
        else:
            node.parent.right = new_node
        new_node.parent = node.parent

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

    def maximum(self, node) -> Node:
        while node.right != self.TNULL:
            node = node.right
        return node

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

    def __print_helper(self, node, indent, last):
        if node != self.TNULL:
            sys.stdout.write(indent)
            if last:
                sys.stdout.write("R----")
                indent += "     "
            else:
                sys.stdout.write("L----")
                indent += "|    "

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

```
def main():
    rb = RB_tree()

    rb.insert(70)
    rb.insert(60)
    rb.insert(85)
    rb.insert(80)
    rb.insert(95)
    rb.insert(65)


    rb.print_tree()

if __name__ == '__main__':
    main()
```

Название файла: hash_table.py

```
import prettytable as pt

DELETED = -2
EMPTY = -1

class Hash_table:
    def __init__(self, size):
        self.size = size
        self.list = [EMPTY for _ in range(size)]
        self.count = 0

    def printTable(self):
        res = pt.PrettyTable()
        res.field_names = ['key', 'value']
        for k, v in enumerate(self.list):
            res.add_row([k,v])
        print(res)

    def __hash_1(self, key) -> int:
        return key % self.size

    def __hash_2(self, key) -> int:
        return self.__getPrime() - (key % self.__getPrime())

    def add(self, key):
        if self.count == self.size:
            raise Exception("Table is full")
        elif self.list[self.__hash_1(key)] == EMPTY or
self.list[self.__hash_1(key)] == DELETED:
            self.list[self.__hash_1(key)] = key
            self.count += 1
        else:
            i = 1
            probe = self.__hash_1(key)
            offset = self.__hash_2(key)
            index = (probe + offset) % self.size
```

```
            while self.list[index] != EMPTY and self.list[index] !=
DELETED:
                i+=1
                if i > self.size:
                    print("No space")
                    return
                index = (probe + i*offset) % self.size
            self.list[index] = key
            return

    def search(self, key):
        if self.count == 0: return
        probe = self.__hash_1(key)

        if self.list[probe] == key: return probe
        else:
            i = 1
            offset = self.__hash_2(key)
            index = (probe + offset) % self.size
            while self.list[index] != key:
                i+=1
                if i > self.size:
                    return
                index = (probe + i*offset) % self.size
            return index

    def delete_elem(self, key):
        if not self.search(key):
            return

        probe = self.__hash_1(key)
        offset = self.__hash_2(key)

        while self.list[probe] != EMPTY:
            if self.list[probe] == key:
                self.list[probe] = DELETED
                self.count-=1
                return
            else:
                probe = (probe + offset) % self.size

    def __getPrime(self) -> int:
        maximum = self.size
        while True:
            if self.__isPrime(maximum):
                return maximum
            maximum-=1

    def __isPrime(self, value) -> bool:
        for i in [2] + list(range(3, int(value**0.5)+1, 2)):
            if not value % i:
                return False
        return True

def main():
```

```
    table_size = 10
    table = Hash_table(table_size)
    for i in range(25):
        table.add(i)
    elem = table.search(23)
    table.delete_elem(4)
    table.printTable()
    print(elem)


if __name__ == '__main__':
    main()
```

Название файла: measuring.py

```
import  rb_tree as rb
import hash_table as ht
import time


def main():

    rb_tree = rb.RB_tree()
    hash_table = ht.Hash_table(3000)

    for elem in range(1750):
        rb_tree.insert(elem)
        hash_table.add(elem)

    start_time = time.perf_counter()
    rb_tree.insert(10000)
    time_t = (time.perf_counter() - start_time)
    print(time_t)

    start_time = time.perf_counter()
    hash_table.add(10000)
    time_t = (time.perf_counter() - start_time)
    print(time_t)

    start_time = time.perf_counter()
    rb_tree.search(10000)
    time_t = (time.perf_counter() - start_time)
    print(time_t)

    start_time = time.perf_counter()
    hash_table.search(10000)
    time_t = (time.perf_counter() - start_time)
    print(time_t)

    start_time = time.perf_counter()
    rb_tree.delete_elem(10000)
    time_t = (time.perf_counter() - start_time)
    print(time_t)
```

```
    start_time = time.perf_counter()
    hash_table.delete_elem(10000)
    time_t = (time.perf_counter() - start_time)
    print(time_t)

if __name__ == '__main__':
    main()
```

Названиие файла: test.py

```
import pytest
import rb_tree as rb
import hash_table as ht

def test_insert_rb():
    rb_tree = rb.RB_tree()
    rb_tree.insert(70)
    rb_tree.insert(60)
    rb_tree.insert(85)
    rb_tree.insert(80)
    rb_tree.insert(95)
    rb_tree.insert(65)
    assert rb_tree.search(70).color == rb.BLACK
    assert rb_tree.search(65).color == rb.RED

def test_delete_rb():
    rb_tree = rb.RB_tree()
    rb_tree.insert(70)
    rb_tree.insert(60)
    rb_tree.insert(85)
    rb_tree.insert(80)
    rb_tree.insert(95)
    rb_tree.insert(65)
    rb_tree.delete_elem(80)
    assert rb_tree.search(80) == None
    assert rb_tree.search(85).color == rb.BLACK

def test_search_rb():
    rb_tree = rb.RB_tree()
    rb_tree.insert(70)
    rb_tree.insert(60)
    rb_tree.insert(85)
    rb_tree.insert(80)
    rb_tree.insert(95)
    rb_tree.insert(65)
    assert rb_tree.search(80).parent == rb_tree.search(85)
    assert rb_tree.search(60).right == rb_tree.search(65)

def test_add_ht():
    hash_table = ht.Hash_table(30)
    for i in range(25):
        hash_table.add(i)
    assert hash_table.search(24) == 24
```

```
def test_delete_ht():
    hash_table = ht.Hash_table(30)
    for i in range(25):
        hash_table.add(i)
    hash_table.delete_elem(0)
    assert hash_table.list[0] == ht.DELETED

def test_search_ht():
    hash_table = ht.Hash_table(30)
    for i in range(25):
        hash_table.add(i)
    assert hash_table.search(0) == 0
    assert hash_table.search(1000) == None
```