МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТА
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

Тема: Исследование: RB-деревья против Хэш-таблиц (двойное хэширование)

Студент гр. 1384 _____ Белокобыльский И. В.

Преподаватель _____ Иванов Д. В.

Санкт-Петербург
2022

ЗАДАНИЕ
НА КУРСОВУЮ РАБОТУ

Студент Белокобыльский И. В.

Группа 1384

Тема работы: Исследование: RB-деревья против Хэш-таблиц (двойное хэширование)

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 отчета, Введение, Отчет, Примеры работы программы

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 _____ Белокобыльский И. В.

Преподаватель _____ Иванов Д. В.

АННОТАЦИЯ

Задание курсовой работы состоит в реализации предложенных структур данных: Красно-Черное дерево (КЧ-дерево) и Хэш-таблицу, основанную на двойном хешировании. Реализованные структуры данных позволяют вставлять, искать и удалять элементы. Основная часть заключается в сравнении КЧ-дерева и Хэш-таблиц между собой, а также сравнении лучшего, худшего и среднего случая в них.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель работы.

Целью работы является оценка производительности для Хэш-таблиц и КЧ-деревьев.

Задачи работы.

Для достижения заданной цели необходимо выполнение следующих задач:

1. Реализация необходимых структур данных
2. Проведение измерений времени работы для различных входных данных
3. Сравнение полученных результатов

1. РЕАЛИЗАЦИЯ НЕОБХОДИМЫХ СТРУКТУР ДАННЫХ

1.1 КЧ-дерево

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

1. Каждая вершина – либо красная, либо черная
2. Каждый лист – черный
3. Если вершина красная, оба ее ребенка черные
4. Все пути, идущие вниз от корня к листьям, содержат одинаковое количество черных вершин

Для начала был реализован класс Node, у которого в качестве полей есть ссылка на левого и правого детей, а также цвет, ключ и значение. В качестве методов у данного класса были перегружены методы операторы сравнения и привидения к строке.

Затем для класса RBTree была реализованы повороты, вставка, вывод на экран и удаление элементов. В качестве листьев используется единственный элемент – self.nil, что обеспечивает более экономное расходование памяти.

Повороты необходимы для балансировки в случае, если простой смены цвета вершин недостаточно при удалении или добавлении элемента в дерево.

Операция вставки разделяется на несколько случаев, обрабатываемых отдельно:

1. Если нет корня, то мы его создаем. Иначе находим подходящий лист для вставки
2. Вставляем на нужное место вершину красного цвета
3. Если родитель рассматриваемой вершины черный, завершаем работу
4. Если родитель красный, всегда существует дед (иначе нарушаются свойства). Переходим к следующим пунктам
5. Если дядя красный, просто меняем раскраску и переходим к пункту 3 относительно деда

6. Если дядя черный и если родитель с той же стороны от деда, что и ребенок от родителя, то поворачиваем и меняем цвет деда
7. В ином случае совершаем два поворота и смена цвета

Вывод дерева на экран происходит с помощью библиотеки graphviz. Алгоритмом поиска в ширину мы обходим дерево и записываем его в объект Digraph. Аналогично работает метод get_size, необходимый для получения израсходованной памяти

Поиск происходит в цикле по потомкам, основываясь на главном свойстве бинарных деревьев поиска: левый потомок меньше элемента, а правый – больше либо равен него.

Удаление аналогично является перебором возможных случаев.

1. Находим вершину для удаления при помощи метода поиска и удаляем его
2. Если удаляемый элемент имеет детей переходим к пункту 8
3. Если удаляемый элемент красный или корень, завершаем работу
4. Если дядя красный, дед обязательно черный, поэтому меняем цвета и поворачиваем, переходя таким образом к случаю черного дяди
5. Дядя черный. Ребенок дяди с его стороны – красный. Меняем цвета дяди, родителя и ребенка дяди с его стороны, а затем поворачиваем относительно дяди
6. Дядя черный. Ребенок дяди с другой стороны – красный. Меняем дерево так, чтобы оно удовлетворяло предыдущему пункту.
7. Дядя черный. Оба ребенка черные. Независимо от дальнейшей работы, меняем цвет дяди. Если родитель красный, мы просто меняем его цвет и завершаем работу программы. В ином случае переходим к пункту 3
8. В случае, если элемент имеет только одного ребенка, то он всегда черный, а ребенок красным. Меняем местами ребенка и удаляемый элемент, не забыв поменять цвета и обработать ситуацию, если мы удалили корень

9. Если элемент имеет несколько детей, мы берем самого левого ребенка в правом поддереве и заменяем удаляемый элемент на него. После этого переходим к пункту 2 относительно этого ребенка

1.2 Хэш-таблица

Для хэш-таблицы аналогично было реализован класс HashNode выполняющий ту же роль.

Сам класс HashTable использует поданные на вход хэш функции, поэтому время работы по большей части зависит от входных данных. Так как через конструктор регулируется размер буффера и «полнота», при которой необходимо полностью удалять элементы, помеченные флагом deleted.

Для вставки элемента сначала проверяется необходимость выделения нового участка памяти в массиве данных и рехэша – удаления элемента с флагом deleted. Затем берутся значения хэш-функций и по циклу находится свободная ячейка в таблице по индексу h1 + h2 * i, где h1 – значение первой хэш-функции, h2 – второй, i – первый свободный (или помеченный deleted) индекс. Вся арифметика происходит по модулю размера таблицы.

Для поиска элемента аналогично вставке берутся значения хэш-функций и перебираются значения по заданному индексу. Таким образом, если предположить, что на все поданные ключи хэш-функция вернула одно и то же, время поиска, как и вставки, будет иметь асимптотику O(n), но более подробно это будет описано в дальнейших разделах.

Для удаления элемента находится его индекс при помощи метода поиска, а затем соответствующая ячейка помечается флагом deleted.

Для работы программы была реализована хэш-функция для строк, интерпретирующая символы строки как коэффициенты многочлена. Она рассчитывает хэш значение, свернув полином по схеме Горнера.

Для тестирования и сравнения случаев была использована встроенная в Python хэш-функция hash, принятая за эталон в поиске лучшего случая.

2. ПРОВЕДЕНИЕ ИЗМЕРЕНИЙ ВРЕМЕНИ РАБОТЫ ДЛЯ РАЗЛИЧНЫХ ВХОДНЫХ ДАННЫХ

2.1. Рассмотрение случаев для структур данных по отдельности

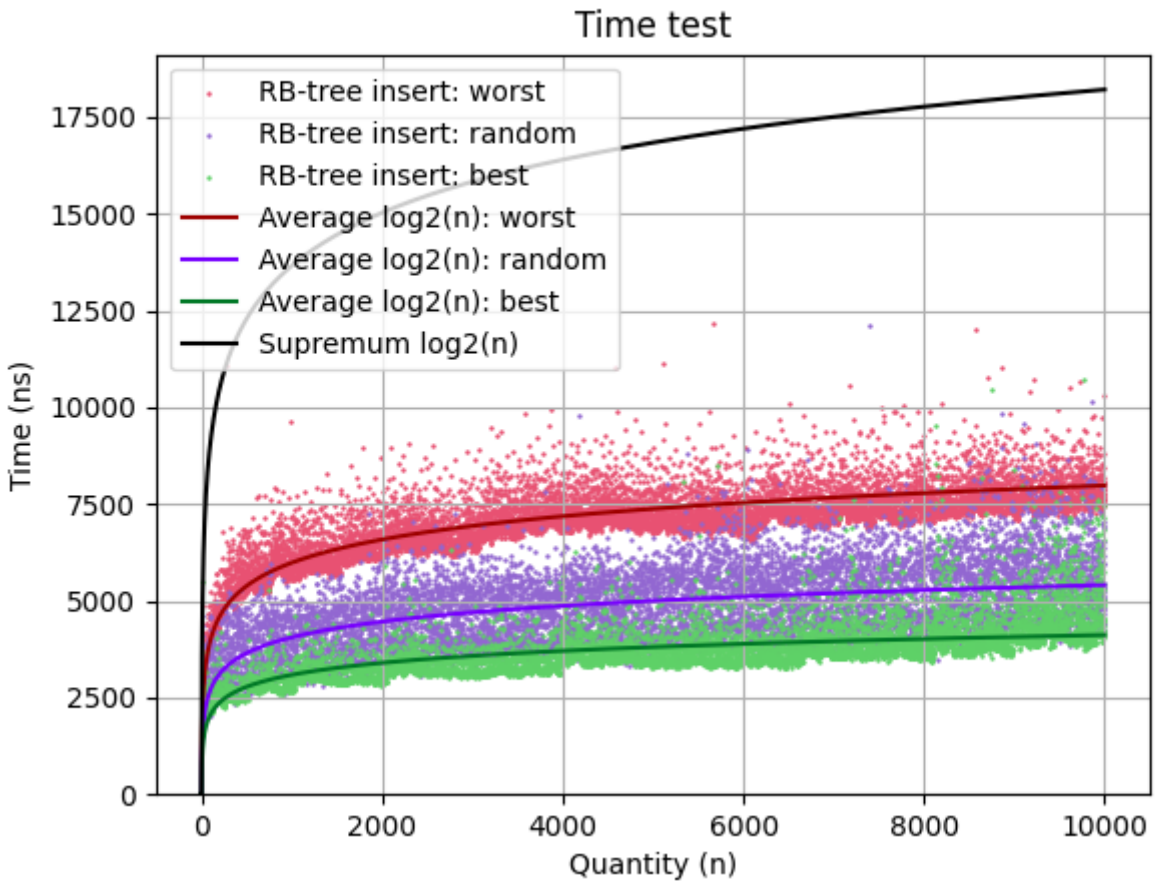

Рис. 1. Сравнение случаев вставки в КЧ-дерево

На рис. 1 рассматривается вставка одного элемента в КЧ-дерево из n элементов. В качестве худшего случая выбран массив из одного и того же элемента, так как он будет всегда вставляться вправо, из-за чего будет максимально возможное количество поворотов. Для среднего случая генерируются псевдослучайные строки. Для лучшего эти же строки прежде вынимаются из другого КЧ-дерева при помощи поиска в ширину – так количество поворотов будет минимальным. Как видно из графика, лучший случай не так сильно отличается от среднего, как худший. Для всех графиков были построены средние приближения логарифмом, а также построен график логарифма, покрывающего все полученные значения.

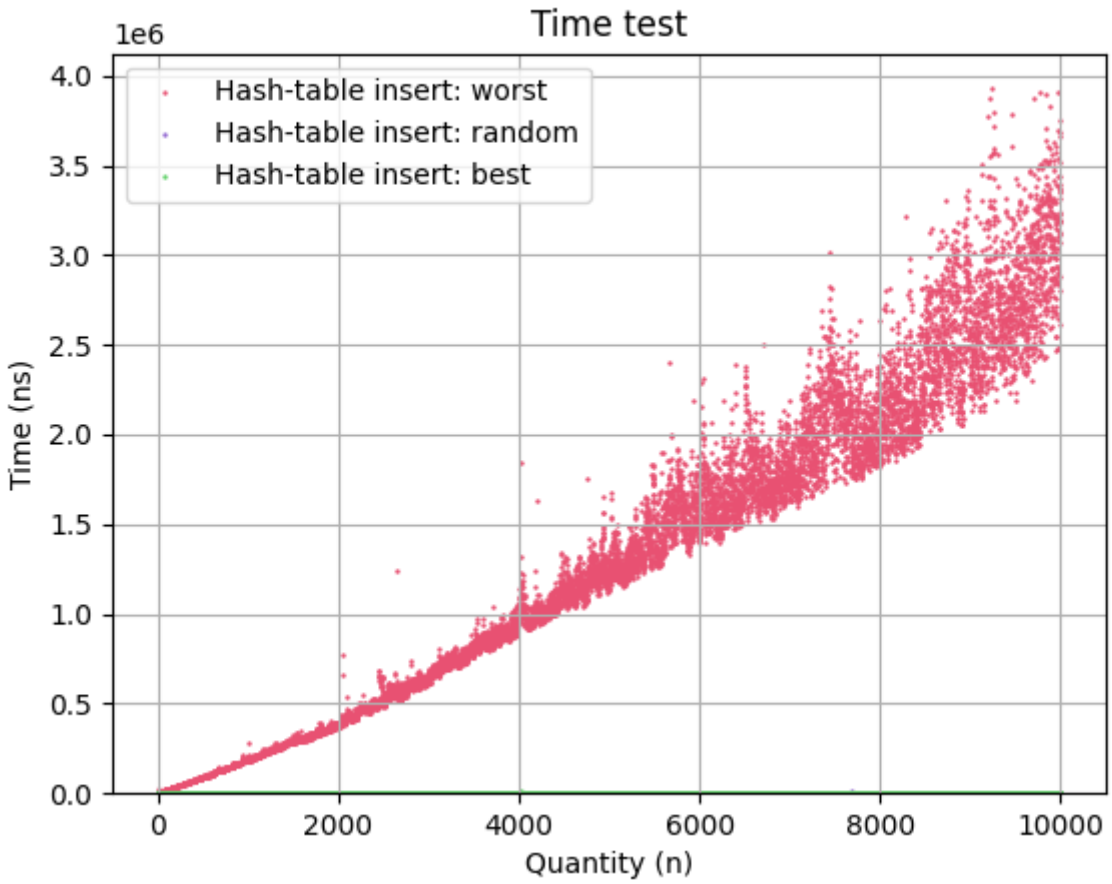

Рис. 2. Сравнение случаев вставки в Хэш-таблицу

На рис. 2 показаны худший, средний и лучший случаи вставки в Хэш-таблицу. В качестве худшего случая был выбран для вставки массив с одинаковыми элементами, чтобы получить максимальное количество коллизий. Как можно видеть, вставка в худшем случае принимает асимптотику O(n).

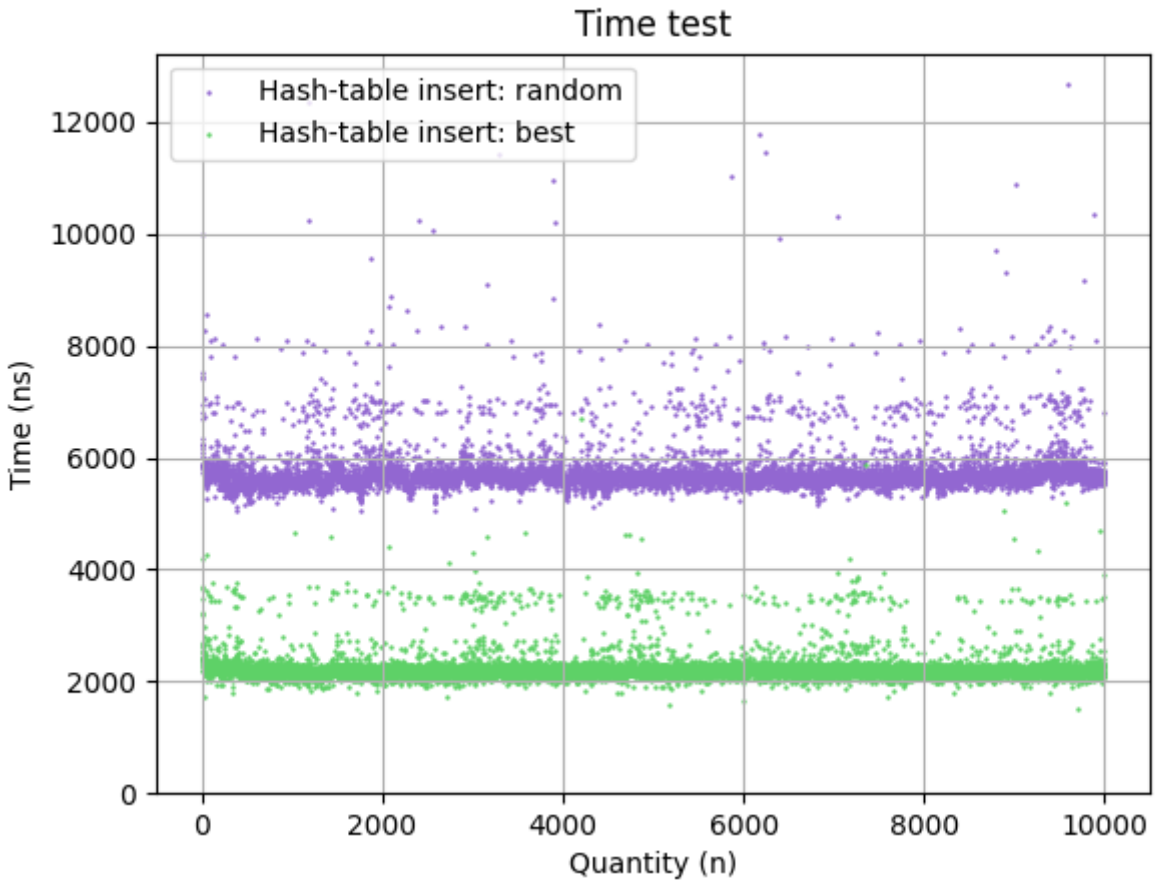

Рис. 3. Сравнение случаев вставки в Хэш-таблицу: лучший и средний случаи

На рис. 3 показаны лучший и средний случаи вставки для Хэш-таблиц. Лучший случай определялся как тот, при котором будет наименьшее количество коллизий, с целью чего была использована встроенная функция hash для сравнения с написанной самостоятельно. Как можно видеть, асимптотика на 10000 значениях практически не отличается, вследствие чего можно сделать вывод, что случаи отличаются лишь тем, насколько оптимизирована хэш-функция.

2.2 Рассмотрение совместных графиков

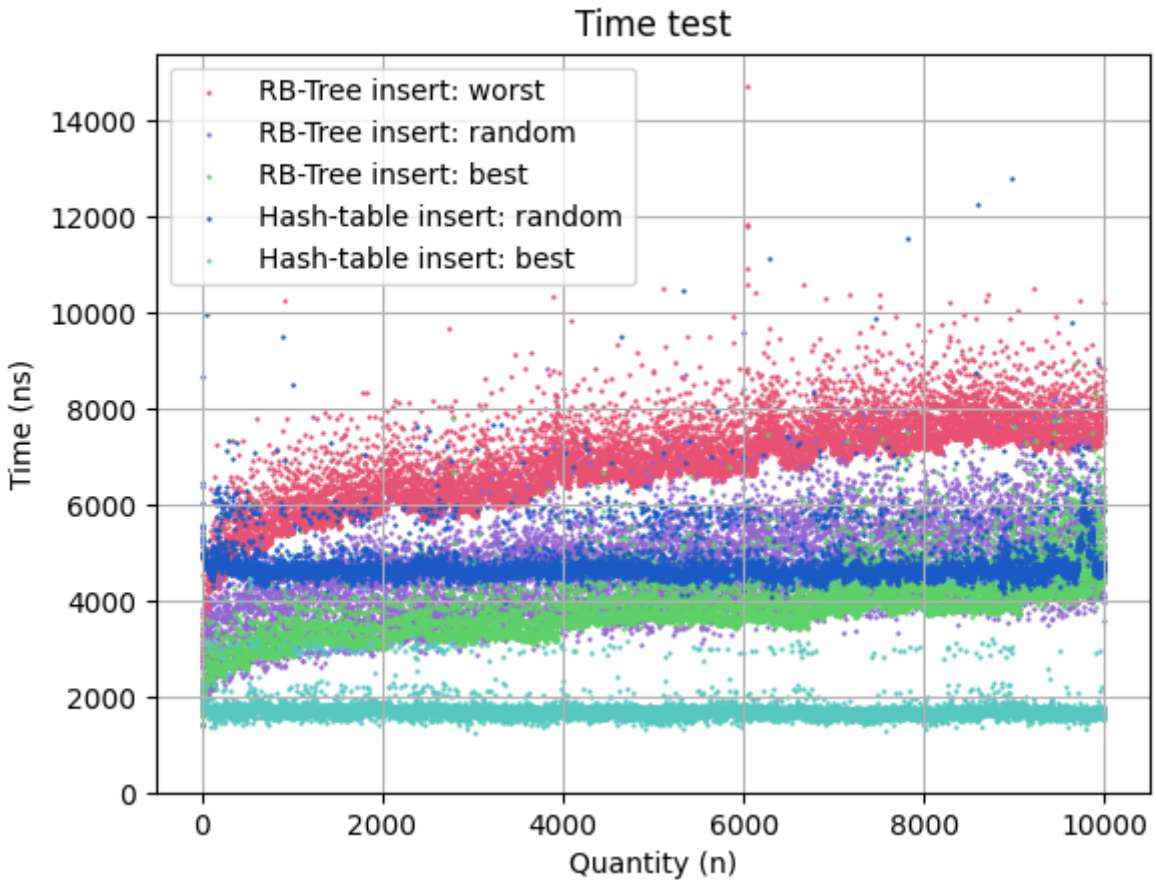

Рис. 4. Сравнение вставки в КЧ-дерево и в Хэш-таблицу

На рис. 4 рассматривается вставка в Хэш-таблицу и КЧ-дерево. Из графика очевидно, что наиболее эффективно использование функции hash. Также, как можно видеть, вставка в КЧ-дерево на начальных этапах эффективнее, чем использование хэш-функции, написанной мной, но из-за своей асимптотики на большом количестве входных данных Хэш-таблицы становятся эффективнее.

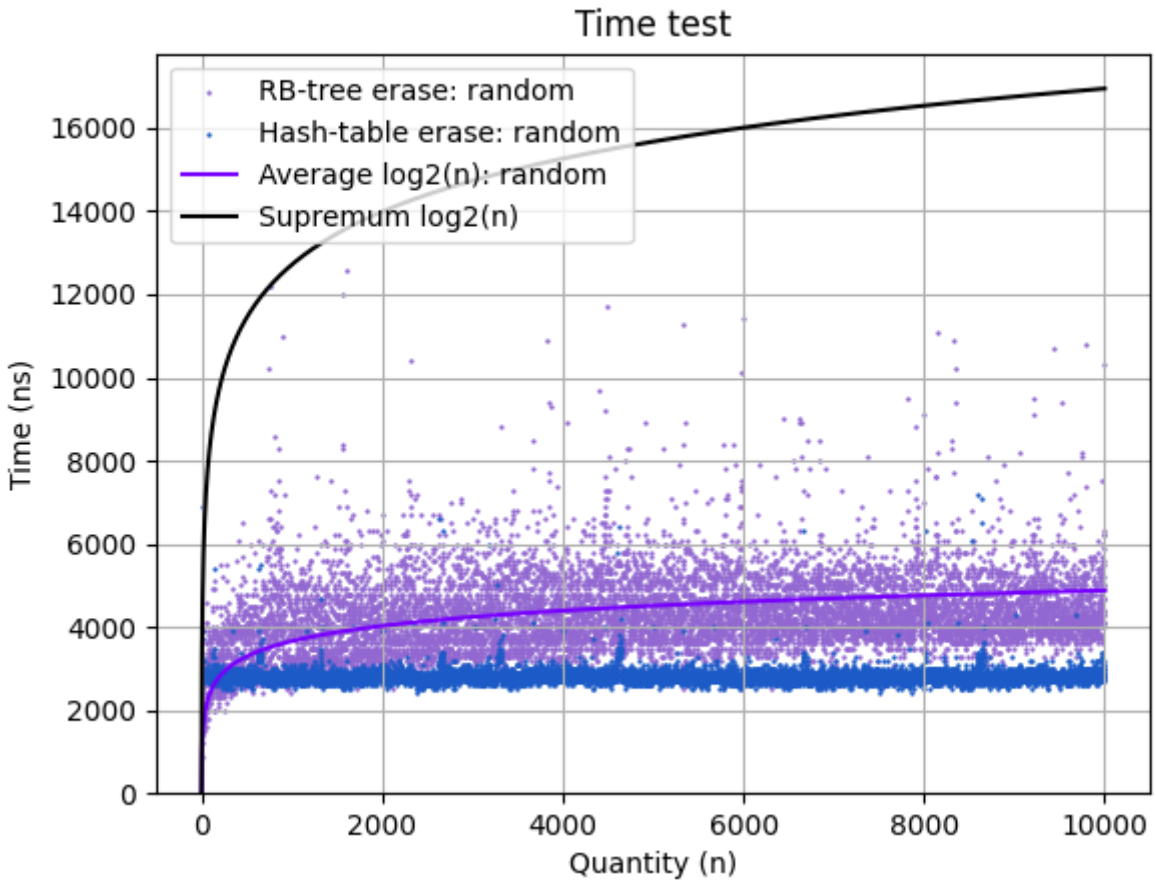

Рис. 5. Сравнение удаления из КЧ-дерева и в Хэш-таблицы

На рис. 5 отображен средний случай удаления в КЧ-дереве и Хэш-таблице. Аналогично предыдущему, в КЧ-дереве на малом количестве элементов время работы занимает меньше, чем в Хэш-таблице, но из-за асимптотики, Хэш-таблицы с увеличением количества элементов становятся эффективнее. Этот график отражает также и время поиска элементов, так как для удаления элемент необходимо сначала найти.

3. СРАВНЕНИЕ ПОЛУЧЕННЫХ РЕЗУЛЬТАТОВ

Полученные в предыдущем разделе данные дают полное представление о поведении каждой структуры данных во всех ситуациях. Резюмируя их, можно сказать следующее:

1. Для вставки большого количества элементов с различными ключами отлично подходят Хэш-таблицы из-за более низкого расхода памяти (не нужно хранить цвет и указатели на детей и родителя), а также вследствие лучшей асимптотики: O(1) против O(log n)
2. При небольшом количестве данных все же предпочтительнее использование КЧ-деревьев, так как, несмотря на асимптотику O(log n), как можно видеть из графиков, в сравнении с написанной самостоятельно хэш-функция, константа у КЧ-деревьев значительно меньше. В целом, использование Хэш-таблиц более выгодно, если использовать встроенную в язык функцию hash.
3. Если необходимо хранить много одинаковых элементов, гораздо предпочтительнее будут КЧ-деревья. Во-первых, у них лучше асимптотика в данном случае: у Хэш-таблиц из-за большого количества коллизий она становится O(n). Во-вторых, константа ниже, если использовать написанную мною хэш-функцию.
4. Если необходимо частое удаление или поиск элементов на большом их количестве, Хэш-таблицы опять-таки получаются эффективнее из-за лучшей асимптотики. На малом количестве элементов выигрывают КЧ-деревья

ЗАКЛЮЧЕНИЕ

В ходе проделанной работы была написана программа на Python, позволяющая в полной мере сравнить Хэш-таблицу и КЧ-дерево. Данные структуры данных были написаны с нуля. Для них реализованы методы, позволяющие проверить корректность данных: __str__ и print соответственно.

Из проделанной работы можно сделать вывод: выбирать структуру данных необходимо в соответствии с задачами, которая она должна решать. В некоторых случаях лучше подходит Хэш-таблица, в некоторых – КЧ-дерево.

ИСТОЧНИКИ

1. Алгоритмы: построение и анализ // Кормен, Т., Лейзерсон, Ч., Ривест, Р., Штайн, М.: Вильямс, 2005.
2. Красно-черное дерево // Викиконспекты ИТМО. URL: https://neerc.ifmo.ru/wiki/index.php?title=%D0%9A%D1%80%D0%B0%D1%81%D0%BD%D0%BE-%D1%87%D0%B5%D1%80%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE
3. Хэш-таблица // Викиконспекты ИТМО. URL: https://neerc.ifmo.ru/wiki/index.php?title=%D0%A5%D0%B5%D1%88-%D1%82%D0%B0%D0%B1%D0%BB%D0%B8%D1%86%D0%B0

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: RBTree.py

```
import sys

import graphviz
import random


class RBTreeNode:
    def __init__(self, color, key=None, value=None, parent=None):
        self.color = color
        self.key = key
        self.value = value
        self.parent = parent
        self.right_child = None
        self.left_child = None

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

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

    def __gt__(self, other):
        if isinstance(other, RBTreeNode):
            return self.key > other.key
        return self.key > other

    def __ge__(self, other):
        if isinstance(other, RBTreeNode):
            return self.key >= other.key
        return self.key >= other
```

```
        def __le__(self, other):
            if isinstance(other, RBTreeNode):
                return self.key <= other.key
            return self.key <= other

        def __str__(self):
            return f'[Node {self.color} [{self.key}: {self.value}]]'


    class RBTree:
        def __init__(self, repeat_keys=False):
            self.repeats = repeat_keys
            self.nil = RBTreeNode('black')
            self.root = self.nil

        def print(self):
            que = [self.root]
            dot = graphviz.Digraph()
            dot.attr('node', fontsize='20')

            def print_node(node, parent_id=''):
                node_id = id(node)
                shape = 'ellipse'
                label = f"{node.key}"
                if node.value is not None:
                    label += f": {node.value}"
                if id(node) == id(self.nil):
                    shape = 'rectangle'
                    node_id += print_node.nils
                    print_node.nils += 1
                    label = 'nil'
                dot.node(str(node_id),  label=label,  color=node.color,
fontcolor=node.color, shape=shape)
                if parent_id:
                    dot.edge(parent_id, str(node_id))

            print_node.nils = 0
            print_node(self.root)
```

```
    dot.format = 'png'
    while que:
        tmp_que = []
        for el in que:
            el_id = str(id(el))
            if el.left_child:
                print_node(el.left_child, el_id)
                tmp_que.append(el.left_child)
            if el.right_child:
                print_node(el.right_child, el_id)
                tmp_que.append(el.right_child)
        que = tmp_que
    dot.render(directory='../../')

def create_node(self, parent, color, key, value=None):
    node = RBTreeNode(color, key, value, parent)
    node.left_child = self.nil
    node.right_child = self.nil
    if key < parent.key:
        parent.left_child = node
    else:
        parent.right_child = node
    return node

def insert(self, key, value=None):
    if key is None:
        return
    if id(self.root) == id(self.nil):
        self.root = RBTreeNode('black', key, value)
        self.root.left_child = self.nil
        self.root.right_child = self.nil
        return
    add_node = self.root
    while True:
        parent_node = add_node
        if add_node == key and not self.repeats:
            add_node.value = value
```

```
                    return
                elif add_node <= key:
                    add_node = add_node.right_child
                else:
                    add_node = add_node.left_child
                if id(add_node) == id(self.nil):
                    add_node = parent_node
                    break
            created = self.create_node(add_node, 'red', key, value)

            def check_colors(node):
                parent = node.parent
                if parent is None or parent.color == 'black':
                    return
                # Т.к. отец красный, дед всегда существует
                grandparent = parent.parent
                if id(grandparent) != id(self.root):
                    grandparent.color = 'red'
                if id(grandparent.left_child) == id(parent):
                    uncle = grandparent.right_child
                else:
                    uncle = grandparent.left_child
                if uncle.color == 'red':
                    """
                           black             red
                          /     \    ->     /   \
                        red     red      black  black
                    """
                    parent.color = 'black'
                    uncle.color = 'black'
                    # Нужно еще проверить относительно деда
                    check_colors(grandparent)
                    return
                if (
                        (id(parent.left_child)    ==    id(node)    and
id(grandparent.left_child) == id(parent)) or
                        (id(parent.right_child)    ==    id(node)    and
id(grandparent.right_child) == id(parent))
```

```
                ):
                    # Если родитель с той же стороны от деда, что и
ребенок от родителя, то поворачиваем и меняем цвет деда
                    self.rotate(parent)
                    parent.color = 'black'
                    grandparent.color = 'red'
                else:
                    self.rotate(node)
                    self.rotate(node)
                    node.color = 'black'
                    grandparent.color = 'red'

            check_colors(created)

        def rotate(self, node):
            if node is None:
                return
            parent = node.parent
            grandparent = parent.parent
            node.parent = grandparent
            parent.parent = node
            if grandparent is not None:
                if id(grandparent.left_child) == id(parent):
                    grandparent.left_child = node
                else:
                    grandparent.right_child = node
            else:
                # Если деда нет, значит родитель -- корень
                self.root = node

            if id(parent.left_child) == id(node):
                parent.left_child = node.right_child
                parent.left_child.parent = parent
                node.right_child = parent
                # node.right_child.parent = node
            else:
                parent.right_child = node.left_child
                parent.right_child.parent = parent
```

```
                node.left_child = parent

        @staticmethod
        def get_uncle(node, parent=None):
            if parent is None:
                parent = node.parent
            if parent is None:
                return None
            if id(parent.left_child) == id(node):
                return parent.right_child
            return parent.left_child

        def erase_node(self, node_del):
            children = (node_del.left_child, node_del.right_child)
            none_children = [child.key is None for child in children]

            def check_colors(node):
                if node.color == 'red':
                    # 1. Node - красный, после него идут листья
                    return

                # 2. Node - черный
                if id(self.root) == id(node):
                    # 2.1. Node - корень
                    return

                parent = node.parent
                uncle = self.get_uncle(node, parent)
                if uncle.color == 'red':
                    # 2.2.1. Дядя - красный
                    """
                    Родитель node и дети дяди всегда будут черными,
причем дети дяди не будут
                    None, так как иначе бы не сходилось по черной высоте
до удаления
                    """
                    uncle.color = 'black'
                    parent.color = 'red'
```

```
                self.rotate(uncle)
                # Сразу же переходим к случаю черного дяди
                uncle = self.get_uncle(node)
                parent = node.parent
            if uncle.color == 'black':
                # 2.2.2. Дядя - черный
                if id(parent.left_child) == id(uncle):
                    same_pos_child = uncle.left_child
                    another_pos_child = uncle.right_child
                else:
                    same_pos_child = uncle.right_child
                    another_pos_child = uncle.left_child

                def same_pos_black(bro, same_child):
                    bro.color = parent.color
                    parent.color = 'black'
                    same_child.color = 'black'
                    self.rotate(bro)

                if same_pos_child.color == 'red':
                    # 2.2.2.1. Ребенок дяди с его стороны - красный
                    same_pos_black(uncle, same_pos_child)
                elif another_pos_child.color == 'red':
                    # 2.2.2.2. Ребенок дяди со стороны node - красный
                    another_pos_child.color = 'black'
                    uncle.color = 'red'
                    before_sibling = uncle
                    new_sibling = another_pos_child
                    self.rotate(another_pos_child)
                    uncle = new_sibling
                    # Перешли к 2.2.2.1
                    same_pos_black(uncle, before_sibling)
                else:
                    # 2.2.2.3. Оба ребенка дяди - черные
                    uncle.color = 'red'
                    if parent.color == 'red':
                        parent.color = 'black'
                    else:
```

```
                    check_colors(parent)

            if all(none_children):
                check_colors(node_del)
                if id(node_del) == id(self.root):
                    self.root = self.nil
                elif id(node_del.parent.left_child) == id(node_del):
                    node_del.parent.left_child = self.nil
                else:
                    node_del.parent.right_child = self.nil

                del node_del
            elif any(none_children):
                # 3. Node имеет только одного ребенка
                child = children[0] if children[0].key is not None else
children[1]
                if id(self.root) == id(node_del):
                    del node_del
                    self.root = child
                    child.color = 'black'
                    return
                """
                Node всегда будет черным, а child -- красным
                Доказательство:
                    Child -- не лист. В таком случае он обязан быть
красным, так как в другом поддереве от node
                    количество черных вершин == 1, следовательно, в
текущем поддереве количество черных вершин
                    в любой ветви также == 1. Таким образом, child всегда
красный, а node всегда черный,
                    так как не может идти две красных вершины подряд
                """
                if id(node_del.parent.left_child) == id(node_del):
                    node_del.parent.left_child = child
                else:
                    node_del.parent.right_child = child
                child.parent = node_del.parent
                child.color = node_del.color
```

```
                del node_del
            else:
                # 4. Node имеет двух детей. Берем минимум в правом
поддереве
                child = node_del.right_child
                while id(child.left_child) != id(self.nil):
                    child = child.left_child
                # Теперь child -- лист. Нам нужен его родитель
                node_del.key = child.key  # swap
                node_del.value = child.value
                self.erase_node(child)

        def erase(self, key):
            node = self.find(key)
            if node is None:
                return
            self.erase_node(node)

        def find(self, key):
            if key is None:
                return None
            cur_node = self.root
            while cur_node.key is not None:
                if cur_node == key:
                    return cur_node
                if cur_node < key:
                    cur_node = cur_node.right_child
                elif cur_node > key:
                    cur_node = cur_node.left_child
            return None

        def __getitem__(self, key):
            node = self.find(key)
            if node is None:
                return None
            return node.value

        def __setitem__(self, key, value):
```

```
            self.insert(key, value)

        def get_size(self):
            res = sys.getsizeof(self.root) + sys.getsizeof(self.nil)
            que = [self.root]
            while que:
                tmp_que = []
                for el in que:
                    if    el.left_child    and    id(el.left_child)    !=
id(self.nil):
                        res += sys.getsizeof(el.left_child)
                        tmp_que.append(el.left_child)
                    if    el.right_child    and    id(el.right_child)    !=
id(self.nil):
                        res += sys.getsizeof(el.right_child)
                        tmp_que.append(el.right_child)
                que = tmp_que
            return res
```

Название файла: HashTable.py

```
import sys


class HashNode:
    def __init__(self, key, value=None):
        self.deleted = False
        self.key = key
        self.value = value

    def __eq__(self, other):
        if isinstance(other, HashNode):
            return self.value == other.value
        return self.value == other

    def __lt__(self, other):
        if isinstance(other, HashNode):
            return self.value < other.value
        return self.value < other
```

```
        def __str__(self):
            return f'"{self.key}": {self.value}'


    class HashTable:
        def __init__(self, hash_first, hash_second, buffer=1000,
rehash_size=0.8, repeat_keys=False):
            if not (0 < rehash_size <= 1):
                rehash_size = 0.8
            self.size = 0
            self.repeat_keys = repeat_keys
            self.exists_size = 0
            self.buffer = buffer
            self.data = [HashNode(None) for _ in range(self.buffer)]
            self.rehash_size = rehash_size
            self.hash_first = hash_first
            self.hash_second = hash_second

        def insert(self, key):
            self.__setitem__(key, None)

        def __setitem__(self, key, value):
            if self.exists_size + 1 > self.rehash_size * self.buffer:
                self.resize()
            if self.size > 2 * self.exists_size:
                self.rehash()
            h1 = self.hash_first(key, self.buffer)
            h2 = self.hash_second(key, self.buffer)
            first_erased = None
            for i in range(self.buffer):
                if self.data[h1].key is None:
                    break
                elif self.data[h1].key == key:
                    if not self.data[h1].deleted and not
self.repeat_keys:
                        self.data[h1].value = value
                        return
                    elif self.data[h1].deleted:
```

```
                    first_erased = h1
                h1 = (h1 + h2) % self.buffer

            if first_erased is not None:
                self.data[first_erased].deleted = False
                self.data[first_erased].value = value
            else:
                self.data[h1] = HashNode(key, value)
                self.size += 1
            self.exists_size += 1

        def resize(self):
            buffer_before = self.buffer
            self.buffer = self.buffer * 2
            self.data += [HashNode(None) for _ in range(buffer_before,
self.buffer)]

        def rehash(self):
            before_data = self.data
            self.data = [HashNode(None) for _ in range(self.buffer)]
            self.size = 0
            self.exists_size = 0
            for i in range(len(before_data)):
                if before_data[i].key is not None and not
before_data[i].deleted:
                    self.__setitem__(before_data[i].key,
before_data[i].value)

        def find(self, key):
            if key is None:
                return None
            h1 = self.hash_first(key, self.buffer)
            h2 = self.hash_second(key, self.buffer)
            for i in range(self.buffer):
                if self.data[h1].key is None:
                    break
                elif self.data[h1].key == key and not
self.data[h1].deleted:
```

```
                    return h1
                h1 = (h1 + h2) % self.buffer
            return None

        def __getitem__(self, key):
            idx = self.find(key)
            if idx is None:
                return None
            return self.data[idx].value

        def erase(self, key):
            idx = self.find(key)
            if idx is None:
                return
            self.data[idx].deleted = True
            self.exists_size -= 1

        def __str__(self):
            res = '{'
            for i in range(self.buffer):
                if self.data[i].key is not None:
                    val                                                    =
"\n\t".join(str(self.data[i].value).split('\n'))
                    res += f'\n\t"{self.data[i].key}": {val},\t\t[{i}]'
                    if self.data[i].deleted:
                        res += "\tDELETED"
            if len(res) > 1:
                res += '\n'
            return res + '}'

        def get_size(self):
            return sys.getsizeof(self.data)
```

Название файла: HashFunction.py

```
def horner(s, table_size, key):
    s = str(s)
    hash_res = 0
    for i in range(len(s)):
        hash_res = (key * hash_res + ord(s[i])) % table_size
```

```
        return (hash_res * 2 + 1) % table_size


    def horner_first(s, table_size):
        return horner(s, table_size, 333667)


    def horner_second(s, table_size):
        return horner(s, table_size, 426389)


    def embedded_hash(s, table_size):
        return hash(s) % table_size
```

Название файла: tests.py

```
    import gc
    import random
    import time
    from math import log2
    import matplotlib.pyplot as plt
    import HashFunction
    from HashTable import HashTable
    from RBTree import RBTree


    def test_insert(data, tests_quantity, struct_name, avg_coef=0,
embedded_hash=False):
        times = [0 for _ in range(len(data))]
        for i in range(tests_quantity):
            if struct_name == "Hash-table":
                hash_first = HashFunction.embedded_hash if embedded_hash
else HashFunction.horner_first
                hash_second = HashFunction.embedded_hash if
embedded_hash else HashFunction.horner_second
                struct = HashTable(hash_first, hash_second,
buffer=2**20, repeat_keys=True)
            else:
                struct = RBTree(repeat_keys=True)
            for j in range(len(data)):
```

```
                gc.disable()
                start = time.perf_counter_ns()
                struct.insert(data[j])
                times[j] += (time.perf_counter_ns() - start) //
tests_quantity
                gc.enable()
            del struct
        avg = sum(times) / len(times)
        logs = {
            'avg': [],
            'sup': []
        }
        log_coefs = []
        res_times = []
        res_n = []
        for i in range(len(times)):
            if avg_coef <= 0 or times[i] / avg < avg_coef:
                res_n.append(i)
                res_times.append(times[i])
                lg = log2(i) if i > 1 else 0
                if lg > 1:
                    log_coefs.append(times[i] / lg)

        max_log_coef = max(log_coefs)
        avg_log_coef = sum(log_coefs) / len(log_coefs)
        for i in res_n:
            lg = log2(i) if i > 1 else 0
            logs['avg'].append(avg_log_coef * lg)
            logs['sup'].append(max_log_coef * lg)
        return {
            'times': res_times,
            'n': res_n,
            'logs': logs
        }


    def get_best_rb_case(data):
        tree = RBTree(repeat_keys=True)
```

```
        for i in range(len(data)):
            tree.insert(data[i])
        que = [tree.root]
        res = [tree.root.key]
        while que:
            tmp_que = []
            for el in que:
                if el.left_child and id(el.left_child) != id(tree.nil):
                    res.append(el.left_child.key)
                    tmp_que.append(el.left_child)
                if    el.right_child    and    id(el.right_child)    !=
id(tree.nil):
                    res.append(el.right_child.key)
                    tmp_que.append(el.right_child)
            que = tmp_que
        return res


    def display_tests(tests, additional_plots=[]):
        fig, ax = plt.subplots()
        for test in tests:
            ax.scatter(test['n'],         test['times'],         s=0.5,
color=test['color'], label=test['label'])
        for plot in additional_plots:
            ax.plot(plot['n'],    plot['data'],    color=plot['color'],
label=plot['label'])
        ax.set_ylim(0)
        ax.set(xlabel='Quantity (n)', ylabel='Time (ns)', title=f"Time
test")
        ax.grid()
        ax.legend(loc=2)
        plt.show()


    def   test_erase(data,   struct_name,   avg_coef=0,   shuffle=False,
embedded_hash=False):
        times = [0 for _ in range(len(data))]
        if struct_name == "Hash-table":
```

```
            hash_first = HashFunction.embedded_hash if embedded_hash
else HashFunction.horner_first
            hash_second = HashFunction.embedded_hash if embedded_hash
else HashFunction.horner_second
            struct = HashTable(hash_first, hash_second, buffer=2**20,
repeat_keys=True)
        else:
            struct = RBTree(repeat_keys=True)
        for j in range(len(data)):
            struct.insert(data[j])
        if shuffle:
            random.shuffle(data)
        for j in range(len(data)):
            gc.disable()
            start = time.perf_counter_ns()
            struct.erase(data[j])
            times[-j] = time.perf_counter_ns() - start
            gc.enable()
        del struct
        avg = sum(times) / len(times)
        logs = {
            'avg': [],
            'sup': []
        }
        log_coefs = []
        res_times = []
        res_n = []
        for i in range(len(times)):
            if avg_coef <= 0 or times[i] / avg < avg_coef:
                res_n.append(i)
                res_times.append(times[i])
                lg = log2(i) if i > 1 else 0
                if lg > 1:
                    log_coefs.append(times[i] / lg)

        max_log_coef = max(log_coefs)
        avg_log_coef = sum(log_coefs) / len(log_coefs)
        for i in res_n:
```

```
            lg = log2(i) if i > 1 else 0
            logs['avg'].append(avg_log_coef * lg)
            logs['sup'].append(max_log_coef * lg)
        return {
            'times': res_times,
            'n': res_n,
            'logs': logs
        }
```

Название файла: main.py

```
import string
from tests import *
import random


def get_all_insert_tests(key_check):
    avg_coef = 3
    rand_data = [''.join(random.choices(string.ascii_letters, k=10)) for i in range(int(1e4))]
    tests = {
        'worst': test_insert(['worst_case' for i in range(int(1e4))], 10, key_check, avg_coef=avg_coef),
        'rand': test_insert(rand_data, 10, key_check, avg_coef=avg_coef),
        'best': test_insert(get_best_rb_case(rand_data), 10, key_check, avg_coef=avg_coef, embedded_hash=True)
    }
    tests['worst']['color'] = '#E85172'
    tests['worst']['label'] = f"{key_check} insert: worst"
    tests['rand']['color'] = '#9267D1'
    tests['rand']['label'] = f"{key_check} insert: random"
    tests['best']['color'] = '#5ED167'
    tests['best']['label'] = f"{key_check} insert: best"
    return tests


def rb_insert_test():
    tests = get_all_insert_tests("RB-tree")
    plots = {
```

```
            'worst': {
                'n': tests['worst']['n'],
                'data': tests['worst']['logs']['avg']
            },
            'rand': {
                'n': tests['rand']['n'],
                'data': tests['rand']['logs']['avg']
            },
            'best': {
                'n': tests['best']['n'],
                'data': tests['best']['logs']['avg']
            },
            'sup': {
                'n': tests['worst']['n'],
                'data': tests['worst']['logs']['sup']
            },
        }
        plots['rand']['color'] = '#7700FF'
        plots['rand']['label'] = 'Average log2(n): random'
        plots['worst']['color'] = '#9E0100'
        plots['worst']['label'] = 'Average log2(n): worst'
        plots['best']['color'] = '#007826'
        plots['best']['label'] = 'Average log2(n): best'
        plots['sup']['color'] = '#000000'
        plots['sup']['label'] = 'Supremum log2(n)'

        display_tests(tests.values(), additional_plots=plots.values())


    def hash_table_insert_test_full():
        tests = get_all_insert_tests("Hash-table")
        display_tests(tests.values())


    def hash_table_insert_test():
        avg_coef = 3
        rand_data = [''.join(random.choices(string.ascii_letters, k=10))
for i in range(int(1e4))]
```

```
        tests = {
            'rand':     test_insert(rand_data,     10,     "Hash-table",
avg_coef=avg_coef),
            'best': test_insert(get_best_rb_case(rand_data), 10, "Hash-
table", avg_coef=avg_coef, embedded_hash=True)
        }
        tests['rand']['color'] = '#9267D1'
        tests['rand']['label'] = f"Hash-table insert: random"
        tests['best']['color'] = '#5ED167'
        tests['best']['label'] = f"Hash-table insert: best"
        display_tests(tests.values())


    def hash_vs_rb_insert_test():
        avg_coef = 3
        rand_data = [''.join(random.choices(string.ascii_letters, k=10))
for i in range(int(1e4))]
        tests = get_all_insert_tests("RB-Tree")
        tests.update({
            'hash_rand':   test_insert(rand_data,   10,   "Hash-table",
avg_coef=avg_coef),
            'hash_best':  test_insert(get_best_rb_case(rand_data),  10,
"Hash-table", avg_coef=avg_coef, embedded_hash=True)
        })
        tests['hash_rand']['color'] = '#1B5AC7'
        tests['hash_rand']['label'] = f"Hash-table insert: random"
        tests['hash_best']['color'] = '#56C7C0'
        tests['hash_best']['label'] = f"Hash-table insert: best"
        display_tests(tests.values())


    def hash_vs_rb_erase_test():
        avg_coef = 3
        rand_data = [''.join(random.choices(string.ascii_letters, k=10))
for i in range(int(1e4))]
        tests = {
            'rb':  test_erase(rand_data,  "RB-tree",  avg_coef=avg_coef,
shuffle=True),
```

```
            'hash':         test_erase(rand_data,         "Hash-table",
avg_coef=avg_coef, shuffle=True),
        }
        tests['rb']['color'] = '#9267D1'
        tests['rb']['label'] = f"RB-tree insert: random"
        tests['hash']['color'] = '#1B5AC7'
        tests['hash']['label'] = f"Hash-table insert: random"
        plots = {
            'avg': {
                'n': tests['rb']['n'],
                'data': tests['rb']['logs']['avg']
            },
            'sup': {
                'n': tests['rb']['n'],
                'data': tests['rb']['logs']['sup']
            },
        }
        plots['avg']['color'] = '#7700FF'
        plots['avg']['label'] = 'Average log2(n): random'
        plots['sup']['color'] = '#000000'
        plots['sup']['label'] = 'Supremum log2(n)'
        display_tests(tests.values(), additional_plots=plots.values())


    hash_vs_rb_erase_test()
```