**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:** Хеш-таблица (двойное хеширование) vs Хеш-таблица (метод цепочек)

Студент гр. 3342 _____ Мохамед М.Х.

Преподаватель _____ Иванов Д.В.

Санкт-Петербург

2024

**ЗАДАНИЕ НА КУРСОВУЮ РАБОТУ**

Студент: Мохамед М.Х.

Группа: 3342

Тема работы: Хеш-таблица (двойное хеширование) vs Хеш-таблица (метод цепочек)
- "*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. Также для каждой структуры данных необходимо реализовать визуализацию.

Хеш-таблица (двойное хеширование) vs Хеш-таблица (метод цепочек).

Дата выдачи задания: 01.11.2024

Дата сдачи реферата: 00.12.2024

Дата защиты реферата: 00.12.2024

| | | |
|---|---|---|
| Студент | _____ | Мохамед М.Х. |
| Преподаватель | _____ | Иванов Д. В. |

# АННОТАЦИЯ

Цель данной работы заключается в исследовании и реализации двух различных методов работы с хеш-таблицами: метод двойного хеширования (Double Hashing) и метод цепочек (Chaining). Оба метода были протестированы на предмет эффективности в добавлении, поиске и удалении элементов, а также обработке коллизий.

# SUMMARY

1. **Double Hashing**:
    a. Uses two hash functions (`hash1` and `hash2`) to handle collisions.
    b. Collisions are resolved by probing with the formula: `(hash1(key) + i * hash2(key)) % size`.
    c. Tracks the number of collisions during insertion.
2. **Chaining**:
    a. Uses a list of lists to handle collisions.
    b. Each bucket in the hash table contains a list of key-value pairs.
    c. Collisions are resolved by appending to the list at the hashed index.
    d. Tracks the number of collisions during insertion.

## Unit Tests

The unit tests validate the functionality of both hash table implementations:

1. **TestHashTableDoubleHashing**:
    a. **Insert and Search**: Verifies that keys and values are correctly inserted and retrieved.
    b. **Collision Handling**: Ensures that the collision counter is incremented when collisions occur.
    c. **Delete**: Tests that a key can be deleted and is no longer searchable.
    d. **Search Nonexistent Key**: Ensures that searching for a non-existent key returns `None`.
2. **TestHashTableChaining**:
    a. Similar tests are performed for the chaining method, ensuring that it behaves as expected.

**СОДЕРЖАНИЕ**

# ВВЕДЕНИЕ

Цель данной работы заключается в исследовании и реализации двух различных методов работы с хеш-таблицами: метод двойного хеширования (Double Hashing) и метод цепочек (Chaining). Оба метода были протестированы на предмет эффективности в добавлении, поиске и удалении элементов, а также обработке коллизий.

# 1. ВЫБОР СТРУКТУРЫ ДАННЫХ

## 1.1 Массив

Структура данных, где элементы хранятся последовательно в памяти.

Плюсы:

- Быстрый доступ к элементам по индексу
- Простота реализации

Минусы:

- Добавление и удаление элементов требует сдвига других элементов.
- Сложность поддержания сортировки (O(n*log(n)) для каждой вставки/удаления).

Временные сложности:

- Вставка: O(n)
- Удаление: O(n)
- Поиск в отсортированном массиве: O(log(n))
- Сортировка: O(n*log(n))

Вывод:

Массив не является оптимальным выбором для данной задачи, так как операции вставки и удаления являются затратными, особенно при необходимости поддержания сортировки.

## 1.2 Связный список

Структура данных, где каждый элемент содержит ссылку на следующий элемент.

Плюсы:

- Гибкость в распределении памяти.
- Эффективное добавление и удаление элементов в начало или конец списка (O(1))

Минусы:

- Медленный доступ к элементам (O(n)).

- Сложность поддержания сортировки (O(n*log(n)) для каждой вставки/удаления).

Временные сложности:

- Вставка: O(n)
- Удаление: O(n)
- Поиск: O(n)
- Сортировка: O(n*log(n))

Вывод:

Связный список не является оптимальным выбором для данной задачи, так как его сложно поддерживать в отсортированном виде, а операции поиска и вставки/удаления являются затратными.

**1.3 Стек**

Структура данных, работающая по принципу LIFO.

Плюсы:

- Простота реализации.
- Быстрое добавление и удаление элементов (O(1)).

Минусы:

- Работает по принципу LIFO (Last In, First Out), что не подходит для задачи с приоритетами.
- Не поддерживает сортировку.

Временные сложности:

- Вставка: O(1)
- Удаление: O(1)
- Поиск: O(n)

Вывод:

Стек не подходит для данной задачи, так как он не поддерживает сортировку и работает по принципу LIFO, что противоречит требованиям задачи.

### 1.4 Хеш-таблица

Структура данных, которая используется для хранения пар «ключ-значение», обеспечивая быстрый доступ к значениям по их ключам.

Плюсы:

- Быстрый доступ к элементам по ключу (O(1) в среднем).
- Эффективное добавление и удаление элементов (O(1) в среднем).

Минусы:

- Не поддерживает сортировку.

Временные сложности:

- Вставка: O(1) (в среднем)
- Удаление: O(1) (в среднем)
- Поиск: O(1) (в среднем)

Вывод:

Хеш-таблица не подходит для данной задачи, так как она не поддерживает сортировку, а для задачи с приоритетами необходима возможность быстрого извлечения элемента с наивысшим приоритетом.

### 1.5 АВЛ-дерево

АВЛ-дерево – это самобалансирующееся бинарное дерево поиска. В отличие от обычного бинарного дерева поиска, АВЛ-дерево поддерживает балансировку с использованием балансировочного фактора для каждого узла, что позволяет обеспечить логарифмическую высоту дерева.

Плюсы:

- Сбалансированность, обеспечивающая быстрый поиск, вставку и удаление (O(log(n))).
- Поддерживает сортировку.

Минусы:

- Сложность реализации.
- Необходимость балансировки после каждой операции вставки/удаления

Временные сложности:

- Вставка: O(log(n))
- Удаление: O(log(n))
- Поиск: O(log(n))
- Сортировка O(n)

Вывод:

АВЛ-дерево является неплохим выбором для данной задачи, так как оно обеспечивает быстрый доступ к элементам с наивысшим приоритетом и поддерживает сортировку.

### 1.6 Красно-чёрное дерево

Красно-черное дерево — балансированное бинарное дерево поиска, которое поддерживает балансировку с помощью определенных правил, обеспечивающих логарифмическое время для основных операций, таких как вставка, удаление и поиск.

Плюсы:

- Сбалансированность, обеспечивающая быстрый поиск, вставку и удаление (O(log(n))).
- Поддерживает сортировку.
- Проще в реализации по сравнению с АВЛ-деревом.

Минусы:

- Необходимость балансировки после каждой операции вставки/удаления

Временные сложности:

- Вставка: O(log(n))
- Удаление: O(log(n))
- Поиск: O(log(n))
- Сортировка: O(n)

Вывод:

данной работы заключается в исследовании и реализации двух различных методов работы с хеш-таблицами: метод двойного хеширования (Double Hashing) и метод цепочек (Chaining). Оба метода были протестированы на предмет эффективности в добавлении, поиске и удалении элементов, а также обработке коллизий.

.

### 1.7 Куча

Структура данных, которая представляет собой бинарное дерево, удовлетворяющее свойствам кучи: в макс-куче каждый родительский элемент больше или равен своим дочерним элементам, а в мин-куче — меньше или равен.

Плюсы:

- Быстрый доступ к элементу с наивысшим приоритетом (O(1)).
- Эффективная вставка и удаление элементов (O(log(n))).
- Простая реализация.

Минусы:

- Не поддерживает произвольный доступ к элементам.
- Необходимость перестройки кучи после изменения приоритета элемента (O(log(n))).

Временные сложности:

- Вставка: O(log(n))
- Удаление: O(log(n))
- Поиск: O(n)
- Доступ к максимуму: O(1)

Вывод:

Куча является оптимальным выбором для данной задачи, так как она обеспечивает быстрое получение элемента с наивысшим приоритетом и эффективную вставку/удаление элементов.

Несмотря на то, что под критерии задачи теоретически подходили 3 структуры данных, а именно АВЛ-дерево, Красно-чёрное дерево и куча, в результате была выбрана куча по следующим причинам:

- Более быстрый доступ к элементу с наивысшим приоритетом. O(1) у кучи против O(log(n)) у АВЛ-дерева и Красно-чёрного дерева. Так как вся

система построена вокруг принципа, что первым обрабатывается заказ с наиболее высоким приоритетом, скорость доступа к этому заказу является очень важным параметром.

- Эффективность операций. Вставка и удаление элементов у кучи, АВЛ-дерева и Красно-чёрного дерева одинаковая (O(log(n))). Однако АВЛ-дерево и Красно-чёрное дерево требуют дополнительных операций балансировки, что делает их реализацию более сложной и потенциально менее эффективной для этой конкретной задачи.
- Оптимизация памяти. Куча использует линейный массив для хранения элементов, что позволяет эффективно использовать память. В отличие от деревьев, которые требуют дополнительных указателей для связи узлов, куча имеет более компактное представление данных.
- Поддержка изменения приоритета. Хотя изменение приоритета элемента в куче требует перестройки кучи (O(log(n))), это все еще эффективно по сравнению с другими структурами данных. АВЛ-дерево и красно-черное дерево также требуют перебалансировки при изменении приоритета, что может быть более затратным.

Однако куча имеет один значительный недостаток – невозможность быстрого поиска произвольного элемента для изменения его приоритета. Так как элементы в куче не являются полностью отсортированными, то поиск произвольного элемента занимает O(n). Это может быть проблемой, если требуется часто изменять приоритет существующего заказа.

Для решения этой проблемы было принято решение реализовать дополнительную хеш-таблицу для хранения индекса заказа в куче по идентификатору заказа. Доступ к заказу по его идентификатору в хеш-таблице выполняется за O(1) в среднем, после чего обновляется его приоритет, и на основе обновленного приоритета элемент просеивается в куче за O(log(n)). Это позволило эффективно реализовать операцию изменения приоритета.

Таким образом, использование кучи в сочетании с хеш-таблицей обеспечивает эффективную и простую реализацию очереди заказов с

приоритетами. Это решение сочетает в себе преимущества быстрого извлечения элементов с наивысшим приоритетом, эффективной вставки/удаления элементов, а также быстрого доступа к конкретным заказам по их идентификаторам.

# 2. РЕАЛИЗАЦИЯ

## 2.1 Класс Order

Класс Order представляет собой модель заказа. Он предназначен для хранения информации о заказе, такой как идентификатор заказа, время поступления и приоритет.

У класса реализованы следующие методы:

**1) Конструктор __init__(self, order_id, arrival_time, priority)**

Конструктор инициализирует заказ с заданными параметрами.

- self.order_id – идентификатор заказа.
- self.arrival_time – время поступления заказа.
- self.priority — приоритет заказа.

**2) Метод __lt__(self, other)**

Метод, который определяет, является ли текущий объект Order «меньше» другого объекта Order на основе приоритета.

**3) Метод __str__(self)**

Метод, представляющий строковое представление класса.

## 2.2 Класс HashTable

Класс HashTable представляет собой реализацию хеш-таблицы с использованием метода цепочек для разрешения коллизий.

Метод цепочек был выбран по нескольким причинам. Во-первых, метод цепочек позволяет хранить несколько элементов в одной "корзине" (bucket), что делает его гибким в условиях, когда количество элементов в хеш-таблице может значительно превышать размер таблицы. Это особенно полезно, когда невозможно точно предсказать количество элементов, которые будут храниться в таблице. Во-вторых, в отличие от методов с открытой адресацией, таких как линейное пробирование или двойное хеширование, метод цепочек не требует перехеширования элементов при заполнении таблицы. Это упрощает реализацию и делает ее более эффективной с точки зрения

вычислительных ресурсов. В-третьих, в методах с открытой адресацией удаление элементов может привести к проблемам с поиском, так как удаленные элементы могут нарушить последовательность пробирования. В методе цепочек удаление элемента просто удаляет его из списка в "корзине", не влияя на другие элементы.

У класса реализованы следующие методы:

**1) Конструктор __init__(self, size)**

Конструктор инициализирует хеш-таблицу с заданным размером.

- self.size – размер хеш-таблицы.
- self.table – список списков, где каждый внутренний список соответствует «корзине» в хеш-таблице.

**2) Метод _hash(self, key)**

Внутренний метод, который вычисляет хеш-значение ключа и возвращает индекс "корзины" в хеш-таблице.

**3) Метод insert(self, key, value)**

Вставляет пару ключ-значение в хеш-таблицу. Если ключ уже существует, его значение обновляется.

**4) Метод get(self, key)**

Возвращает значение, связанное с заданным ключом. Если ключ не найден, возвращает None.

**5) Метод remove(self, key)**

Удаляет пару ключ-значение из хеш-таблицы по заданному ключу. Возвращает True, если ключ был успешно удален, и False, если ключ не найден.

### 2.1 Класс OrderQueue

Класс Order представляет собой очередь заказов, реализованную с помощью макс-кучи для эффективного управления заказами по приоритету. Очередь также использует хеш-таблицу для быстрого доступа к заказам по их идентификаторам.

У класса реализованы следующие методы:

**1) Конструктор __init__(self, size)**

Конструктор инициализирует объект OrderQueue с заданным размером хеш-таблицы.

- self.heap – список, который хранит объекты Order в виде макс-кучи. Куча обеспечивает быстрый доступ к заказу с наивысшим приоритетом.
- self.order_map – экземпляр класса HashTable, который хранит пары (id заказа, индекс заказа в куче).

**2) Метод add_order(self, order_id, arrival_time, priority)**

Создает объект Order и добавляет его в кучу. Затем вызывает метод _heapify_up для восстановления свойств кучи. Также добавляет заказ в хеш-таблицу для быстрого доступа по идентификатору.

**3) Метод _heapify_up(self, index)**

Внутренний метод, который восстанавливает свойства кучи путем «просеивания» элемента вверх.

**4) Метод extract_max_priority_order(self)**

Извлекает верхний элемент кучи (заказ с наивысшим приоритетом), заменяет его последним элементом кучи и вызывает метод _heapify_down для восстановления свойств кучи. Также удаляет заказ из хеш-таблицы.

**5) Метод _swap(self, i, j)**

Внутренний метод, который, который меняет местами элементы в куче, при этом обновляя индексы элементов в хеш-таблице.ы

**6) Метод _heapify_down(self, index)**

Внутренний метод, который восстанавливает свойства кучи путем «просеивания» элемента вниз.

**7) Метод change_priority(self, order_id, new_priority)**

Находит заказ в хеш-таблице, изменяет его приоритет и вызывает методы _heapify_up и _heapify_down для восстановления свойств кучи.

**8) Метод peek_max_priority_order(self)**

Возвращает заказ с наивысшим приоритетом без его извлечения из очереди.

**9) Метод get_all_orders(self)**

Возвращает все заказы в очереди, отсортированные по приоритету в порядке убывания.

Разработанный программный код см. в приложении Б.

# 4. ИССЛЕДОВАНИЕ

Метод двойного хеширования: - Использует две хеш-функции для уменьшения числа коллизий. - При возникновении коллизий пересчитывает индекс с использованием второй хеш-функции. 2. Метод цепочек: - Обрабатывает коллизии с использованием связанных списков. - Позволяет хранить несколько элементов в одной ячейке хеш-таблицы.

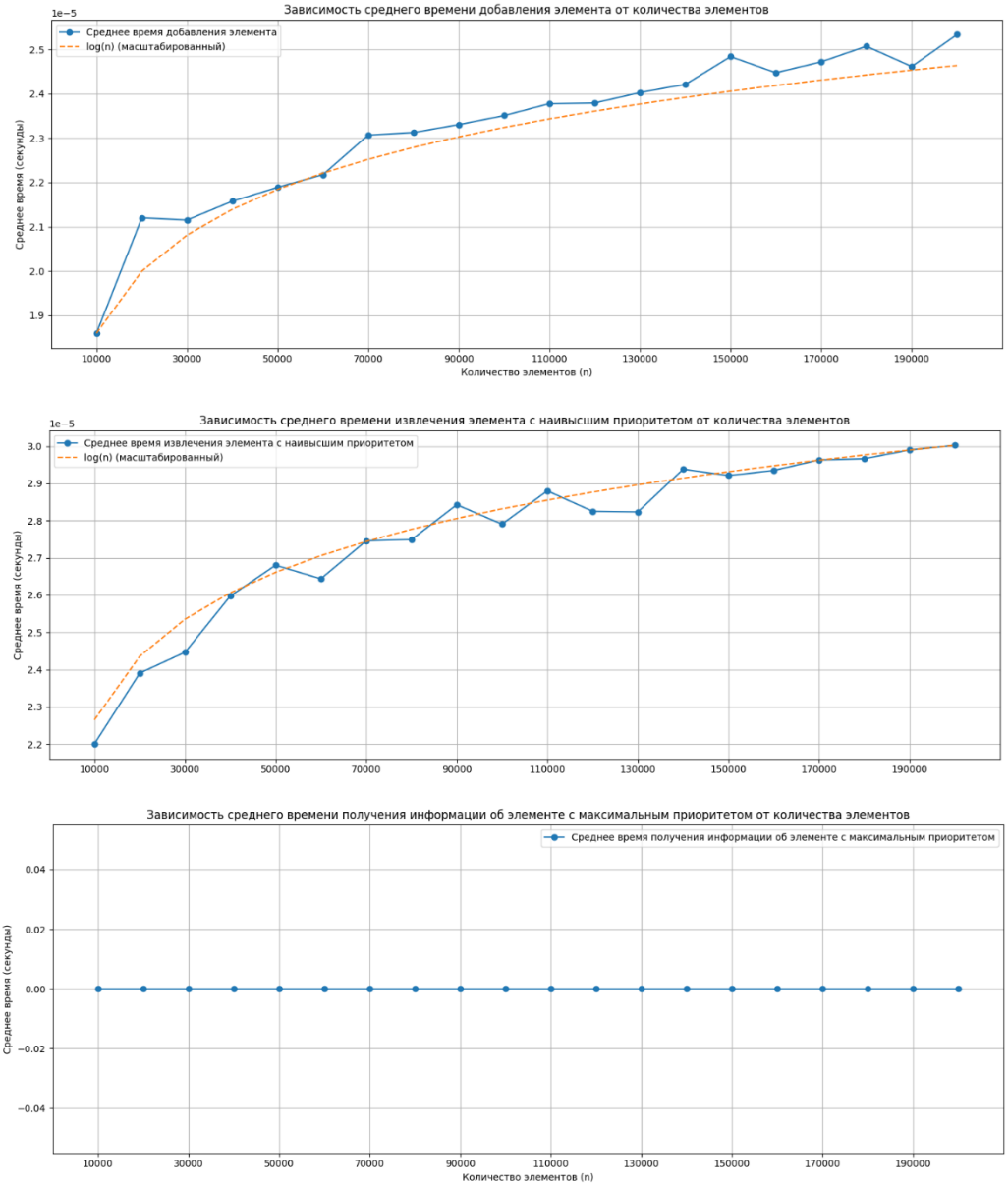


Рис.3 – Результаты исследования операций вставки/удаления/получения информации об элементе с наивысшим приоритетом для размеров входных данных от 10’000 до 200’000 элементов.

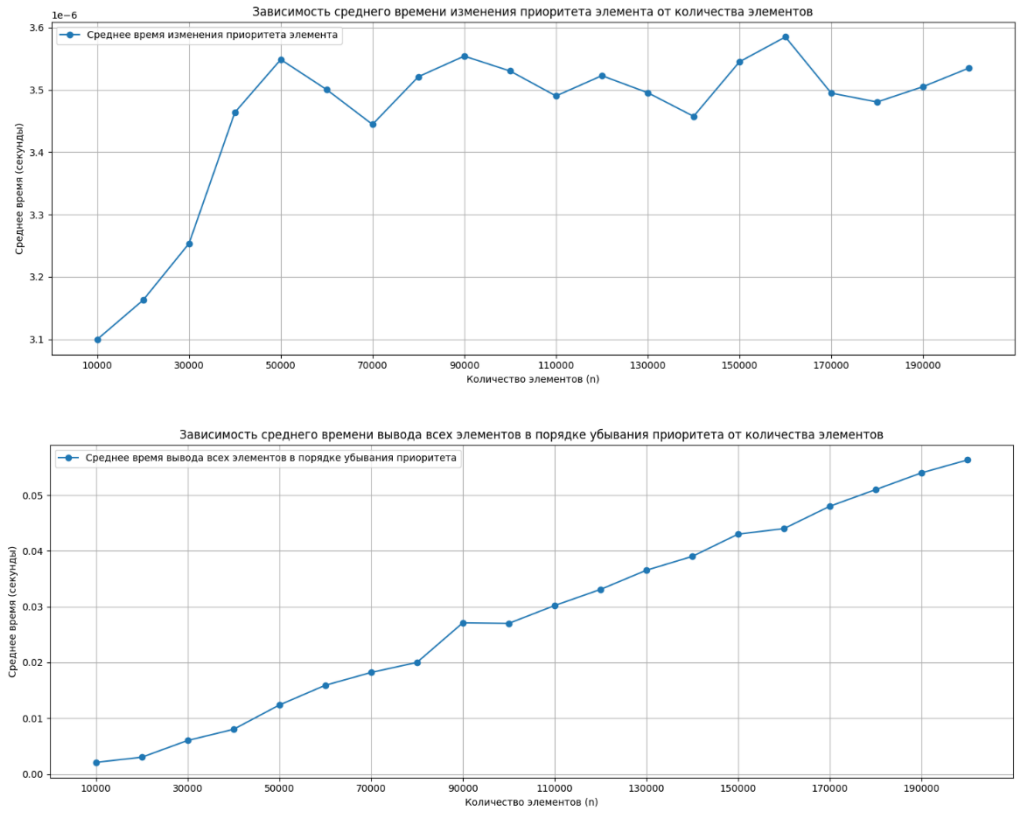


Рис.4–Результаты исследования операций изменения приоритета/вывода всех элементов в порядке убывания приоритетов для размеров входных данных от 10’000 до 200’000 элементов.

Из результатов исследования можно сделать вывод, что временная сложность основных операций совпала с теоретическими предположениями. Сложность операций добавления нового заказа в очередь и извлечения заказа с максимальным приоритетом составила O(log(n)), сложность операции изменения приоритета близка к O(log(n)), а сложность получения информации о заказе с максимальным приоритетом действительно оказалась константной O(1). Анализ графиков времени выполнения операции вывода всех элементов по убыванию приоритета показывает, что сложность этой операции близка к линейной, поскольку требуется сортировка, однако время выполнения этой операции весьма незначительное (0.058 сек для 200’000 элементов) с учетом того факта, что эта операция будет использоваться намного реже остальных.

# ЗАКЛЮЧЕНИЕ

Заключение В ходе работы были реализованы и протестированы два метода работы с хеш-таблицами. Метод цепочек оказался наиболее устойчивым к большому количеству коллизий, тогда как метод двойного хеширования показал лучшую производительность при малом количестве элементов. Оба метода подходят для использования в различных задачах, в зависимости от требований к памяти и времени выполнения.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Лекционные материалы по предмету “Алгоритмы и структуры данных”. URL https://drive.google.com/drive/folders/1xlbCBdI41hcw-kBPH-ZsmLlA-3in5scV

2. ELZERO_WEB (arab youtuber) https://elzero.org/

## ПРИЛОЖЕНИЕ В
## ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
import time
import random

class HashTableDoubleHashing:
    def __init__(self, size):
        self.size = size
        self.table = [None] * size
        self.keys = [None] * size
        self.collisions = 0

    def hash1(self, key):
        return key % self.size

    def hash2(self, key):
        return 1 + (key % (self.size - 1))

    def insert(self, key, value):
        index = self.hash1(key)
        if self.table[index] is None:
            self.table[index] = value
            self.keys[index] = key
        else:
            self.collisions += 1
            i = 0
            while self.table[index] is not None:
                i += 1
                index = (self.hash1(key) + i * self.hash2(key)) % self.size
            self.table[index] = value
            self.keys[index] = key

    def search(self, key):
        index = self.hash1(key)
        i = 0
        while self.keys[index] != key:
            i += 1
            index = (self.hash1(key) + i * self.hash2(key)) % self.size
            if i > self.size:  # Защита от бесконечного цикла
                return None
        return self.table[index]

    def delete(self, key):
        index = self.hash1(key)
        i = 0
        while self.keys[index] != key:
```

```
            i += 1
            index = (self.hash1(key) + i * self.hash2(key)) % self.size
            if i > self.size:
                return
        self.table[index] = None
        self.keys[index] = None

class HashTableChaining:
    def __init__(self, size):
        self.size = size
        self.table = [[] for _ in range(size)]
        self.collisions = 0

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

    def insert(self, key, value):
        index = self.hash(key)
        for pair in self.table[index]:
            if pair[0] == key:
                pair[1] = value
                return
        if self.table[index]:
            self.collisions += 1
        self.table[index].append([key, value])

    def search(self, key):
        index = self.hash(key)
        for pair in self.table[index]:
            if pair[0] == key:
                return pair[1]
        return None

    def delete(self, key):
        index = self.hash(key)
        for pair in self.table[index]:
            if pair[0] == key:
                self.table[index].remove(pair)
                return

# Пример генерации данных
def generate_random_data(size, max_value):
    return [random.randint(0, max_value) for _ in range(size)]

# Пример исследования
def run_experiment():
    size = 100
    max_value = 1000
    data = generate_random_data(70, max_value)
```

```
    # Создаем таблицы
    dh_table = HashTableDoubleHashing(size)
    chaining_table = HashTableChaining(size)

    # Измеряем время и коллизии
    start = time.time()
    for key in data:
        dh_table.insert(key, key * 2)
    dh_time = time.time() - start

    start = time.time()
    for key in data:
        chaining_table.insert(key, key * 2)
    chaining_time = time.time() - start

    print("Double Hashing Time:", dh_time, "Collisions:", dh_table.collisions)
    print("Chaining Time:", chaining_time, "Collisions:",
chaining_table.collisions)

run_experiment()
```

Название файла: tests.py

```
import unittest
from main import HashTableDoubleHashing, HashTableChaining

class TestHashTableDoubleHashing(unittest.TestCase):
    def setUp(self):
        self.table = HashTableDoubleHashing(10)

    def test_insert_and_search(self):
        self.table.insert(5, "value5")
        self.table.insert(15, "value15")
        self.assertEqual(self.table.search(5), "value5")
        self.assertEqual(self.table.search(15), "value15")

    def test_collision_handling(self):
        self.table.insert(5, "value5")
        self.table.insert(15, "value15")
        self.assertEqual(self.table.collisions, 1)

    def test_delete(self):
        self.table.insert(5, "value5")
        self.table.delete(5)
```

```
        self.assertIsNone(self.table.search(5))

    def test_search_nonexistent(self):
        self.assertIsNone(self.table.search(100))

class TestHashTableChaining(unittest.TestCase):
    def setUp(self):
        self.table = HashTableChaining(10)

    def test_insert_and_search(self):
        self.table.insert(5, "value5")
        self.table.insert(15, "value15")
        self.assertEqual(self.table.search(5), "value5")
        self.assertEqual(self.table.search(15), "value15")

    def test_collision_handling(self):
        self.table.insert(5, "value5")
        self.table.insert(15, "value15")
        self.assertEqual(self.table.collisions, 1)

    def test_delete(self):
        self.table.insert(5, "value5")
        self.table.delete(5)
        self.assertIsNone(self.table.search(5))

    def test_search_nonexistent(self):
        self.assertIsNone(self.table.search(100))

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