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: Реализация структуры данных

Студент гр. 2384 _____ Кузьминых Е.М.

Преподаватель _____ Иванов Д.В.

Санкт-Петербург

2023

ЗАДАНИЕ

НА КУРСОВУЮ РАБОТУ

Студент Кузьминых Е.М.

Группа 2384

Тема работы : Реализация структуры данных

Исходные данные:

Вариант 3

Хэш-таблицы (метод цепочек) - удаление

Содержание пояснительной записки:

«Аннотация», «Содержание», «Введение», «Теоретическое описание структуры данных», «Реализация структуры данных», «Исследование структуры данных», «Заключение», «Список использованных источников», «Исходный код программы»

Предполагаемый объем пояснительной записки:

Не менее 15 страниц.

Дата выдачи задания: 13.12.2023

Дата сдачи реферата: 24.12.2023

Студент _____ Кузьминых Е.М.

Преподаватель _____ Иванов Д.В.

АННОТАЦИЯ

В ходе выполнения курсовой работы были разработаны классы, реализующие такую структуру данных, как хэш-таблица (способ разрешения коллизий – метод цепочек), методы для взаимодействия с ней, в частности, был исследован метод удаления элемента из хэш-таблицы. Кроме этого были написаны тесты для проверки корректности работы программы. Тестирование происходило при помощи библиотеки pytest. Для реализации структуры данных был выбран язык программирования Python.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель работы

Целью в ходе выполнения курсовой работы является реализация хэш-таблицы, разрешающая коллизию с помощью метода цепочек на языке программирования Python, проведение исследования разработанной структуры данных, анализ ее работы и создание тестов для проверки корректности работы программы.

Задачи:

• Реализовать хэш-таблицу с разрешением коллизии с помощью метода цепочек.

• Реализовать несколько хэш-функций для анализа работы хэш-таблицы, провести анализ работы хэш-таблицы в зависимости от используемой хэш-функции.

• Написать тесты для проверки корректности работы написанной программы.

1. ТЕОРЕТИЧЕСКОЕ ОПИСАНИЕ СТРУКТУРЫ ДАННЫХ

1.1. Основные понятия и определения

Хэш-функция - это функция, которая преобразует входные данные произвольной длины в выходные данные фиксированной длины. Эти выходные данные, которые обычно называют хэшкодом, используются для определения позиции элемента в хэш-таблице. Хэш-функции широко используются в информационных технологиях для ускорения поиска данных, проверки целостности данных и шифрования.

Хэш-таблица - это структура данных, которая использует хэш-функции для преобразования ключей в индексы массива, где хранятся соответствующие значения. Это позволяет быстро выполнять операции поиска, вставки и удаления. Однако, когда два разных входных значения при применении хэш-функции дают одинаковый хэш-код, происходит коллизия. Коллизии могут привести к проблемам при использовании хэш-таблиц, так как два разных элемента попытаются занять одну и ту же позицию в таблице.

Важно отметить, что хорошо спроектированная хэш-функция будет минимизировать вероятность коллизий, равномерно распределяя хэш-коды. Это особенно важно для эффективного использования хэш-таблиц.

Существует несколько методов разрешения коллизий в хэш-таблицах, а именно метод цепочек, метод открытой адресации (линейное исследование, квадратичное исследование и двойное хэширование). Один из них - линейное пробирование, при котором, если позиция уже занята, то ищется следующая свободная позиция с шагом 1. Другой метод - метод цепочек, при котором каждый элемент хэш-таблицы является узлом в связном списке. Если происходит коллизия, новый элемент добавляется в связный список. При

квадратичном исследовании интервал между ячейками с каждым шагом увеличивается на константу. При двойном хэшировании интервал между ячейками фиксирован, как при линейном исследовании, но размер интервала вычисляется с помощью второй, вспомогательной хэш-функцией, из-за чего он различен для двух разных ключей.

1.2. Обзор и описание основных операций и алгоритмов

Вставка: Вставка элемента в хэш-таблицу начинается с вычисления хэш-кода ключа. Этот хэш-код используется для определения индекса, где будет храниться значение. Если в этом месте уже есть элемент (произошла коллизия), используется метод линейного пробирования для поиска следующего свободного места. Если таблица заполняется до определенного порога, происходит процесс перехэширования, при котором размер таблицы увеличивается, и все элементы перехэшируются в новую, большую таблицу.

Поиск: Поиск элемента в хэш-таблице также начинается с вычисления хэш-кода ключа. Этот хэш-код используется для определения индекса, где будет храниться значение.

Удаление: Удаление элемента из хэш-таблицы аналогично операции поиска, но вместо возврата найденного значения, элемент удаляется из таблицы. Вычисляется хэш-код ключа, после с его помощью происходит поиск искомого элемента.

Перехэширование: Когда коэффициент заполненности таблицы достигает определенного порога, происходит перехэширование. Это означает, что создается новая таблица с удвоенным размером, копируется таблица с предыдущими значениями и все элементы из старой таблицы вставляются в новую, с увеличенным размером, с использованием заново

вычисленных хэшей, т.е. для каждого метода заново вызывается вызывается метод вставки в новую таблицу.

1.3. Временная и пространственная сложность упомянутых операций и алгоритмов

Временная и пространственная сложность операций, связанных с хэш-таблицей, следующая:

Вставка: Временная сложность вставки в хэш-таблицу в среднем составляет O(1), так как хэш-функция обычно выполняется за постоянное время, и вставка элемента в массив также выполняется за постоянное время. Однако в худшем случае, когда происходит коллизия, временная сложность может достигать O(n), где n - количество элементов в таблице. Пространственная сложность вставки составляет O(1), так как вставка одного элемента не требует дополнительного пространства.

Поиск: Временная сложность поиска в хэш-таблице в среднем также составляет O(1) за счет быстрого доступа к элементам массива. Однако в худшем случае, когда происходит коллизия, временная сложность может достигать O(n). Пространственная сложность поиска составляет O(1), так как поиск не требует дополнительного пространства.

Удаление: Временная и пространственная сложности удаления аналогичны сложностям операции поиска. Обоснование также аналогично поиску и удалению.

Перехэширование: Временная сложность перехэширования составляет O(n), так как все элементы в таблице должны быть перехэшированы и вставлены в новую таблицу. Пространственная сложность

перехэширования составляет O(n), так как требуется создать новую таблицу, которая в два раза больше старой.

2. РЕАЛИЗАЦИЯ СТРУКТУРЫ ДАННЫХ

2.1. Подробное описание разработанных классов, методов и используемых структур данных.

Класс *HashTableEntry* представляет собой внутренний класс, используемый для хранения пар ключ-значение в хэш-таблице. Он имеет три поля:

key: Ключ пары ключ-значение.

value: Значение пары ключ-значение.

next: Ссылка на следующий элемент в списке в случае коллизии.

Конструктор *__init__* класса принимает два аргумента: *key* и *value*, и инициализирует поля этими значениями. Поле *next* инициализируется как None, что означает, что этот элемент является последним в списке.

Класс *HashTable* представляет собой основной класс для реализации хэш-таблицы. Он имеет следующие поля:

capacity: Размер хэш-таблицы, то есть количество ячеек в таблице.

max_load_factor: Максимальное загруженность таблицы, после которой происходит ее перехэширование.

size: Текущее количество элементов в таблице.

table: Список, представляющий собой саму хэш-таблицу. Каждая ячейка в этом списке может содержать либо один элемент *HashTableEntry*, либо None.

Конструктор *__init__* класса принимает два аргумента: *capacity* и *max_load_factor*, и инициализирует поля этими значениями. Поле *size* инициализируется как 0, а поле *table* инициализируется как список из *capacity* элементов, каждый из которых равен None.

Метод *calculate_hash* класса *HashTable* принимает ключ и возвращает его хэш-код. В данной реализации этот метод всегда возвращает 0, что означает, что все ключи будут хэшироваться в одну и ту же ячейку, что приведет к

большим коллизиям. В реальной реализации этого метода следует использовать более сложную хэш-функцию, которая равномерно распределяет ключи по ячейкам таблицы.

Метод *add* класса *HashTable* принимает ключ и значение, вычисляет хэш-код ключа, и добавляет пару ключ-значение в таблицу. Если в ячейке, соответствующей хэш-коду, уже есть элемент, то происходит коллизия, и новая пара ключ-значение добавляется в начало списка в этой ячейке. После добавления пары ключ-значение метод проверяет, не превысил ли фактор загрузки максимального значения, и если это так, то происходит перехэширование таблицы.

Метод *find_value* класса *HashTable* принимает ключ, вычисляет его хэш-код, и ищет в соответствующей ячейке таблицы пару ключ-значение с данным ключом. Если такой пары нет, метод возвращает None. В противном случае метод возвращает значение этой пары.

Метод *resize_table* класса *HashTable* удваивает размер хэш-таблицы и перехэширует все существующие пары ключ-значение. Сначала метод сохраняет старую таблицу в переменной *old_table*. Затем он удваивает размер таблицы, обновляя поле *capacity*. Далее, метод создает новую таблицу с удвоенным размером, инициализируя ее как список из *capacity* элементов, каждый из которых равен None. Затем он обнуляет счетчик размера таблицы (*size*), так как все пары ключ-значение будут вставлены заново. Наконец, метод проходит по всем элементам в старой таблице и добавляет их в новую таблицу. Для каждого элемента он вызывает метод *add*, передавая ключ и значение этого элемента. Таким образом, все пары ключ-значение перехэшируются и распределяются по новым ячейкам таблицы.

2.2. Обоснование принятых решений по выбору типов данных и применяемых алгоритмов.

В данной реализации хеш-таблицы используются следующие типы данных: строки и списки.

Строки используются в качестве ключей для хеш-таблицы. Выбор строки в качестве ключа обусловлен тем, что строки могут представлять собой любые данные, включая числа и сложные объекты, если они преобразованы в строку. Кроме того, строки легко хешировать, и для них существует множество эффективных хеш-функций.

Списки используются для реализации самой хеш-таблицы. Список был выбран для реализации хеш-таблицы, поскольку он предоставляет быстрый доступ к элементам по индексу, что является ключевым аспектом работы хеш-таблицы. Кроме того, списки в Python динамически расширяемы, что позволяет легко изменять размер хеш-таблицы при перехешировании.

Внутренний класс *HashTableEntry* используется для хранения пар ключ-значение в хеш-таблице. Каждый экземпляр этого класса представляет одну пару ключ-значение и содержит следующие поля:

key: Ключ пары ключ-значение.

value: Значение пары ключ-значение.

next: Ссылка на следующий элемент в списке в случае коллизии.

Этот класс используется для организации данных в списке, который представляет собой ячейку в хеш-таблице.

2.3. Интерфейс и примеры использования реализованных классов и методов.

В файле main.py приводится реализация хэш-таблицы. В файле tests.py написаны тесты для проверки корректности работы программы и демонстрации ее функционала. Результаты тестирования приведены ниже.

N	Название теста	Результат	Комментарий
1	test_add_and_find_value	PASSED	Ответ верный
2	test_delete_key	PASSED	Ответ верный
3	test_resize_table	PASSED	Ответ верный
4	test_hash_collision	PASSED	Ответ верный

Таблица 1 – результаты тестирования

3. ИССЛЕДОВАНИЕ СТРУКТУРЫ ДАННЫХ

3.1. Эксперименты по оценке производительности

С помощью встроенных библиотек timeit и random были замерены значения работы удаления N элементов из хэш-таблицы. Для наглядности, при исследовании были использованы различные хэш-функции, “идеальная” – использующая встроенный в Python метод hash(), “плохая” функция – возвращающая всегда одно и то же значение для любого ключа – 0, что приводит к постоянной коллизии и “средняя” – сумма ascii кодов со смещением, графики с результатами исследования прикреплены ниже.

N	Время работы программы (сек)
10	1.98000343516469e-05
100	0.0003872000379487872
1000	0.026095999986864626
10000	2.2231078000040725
100000	295.8046079000924

Таблица 2 – исследование для “плохой” хэш-функции

Можно наблюдать, что при 100% коллизии удаление элемента из хэш-таблицы сводится к удалению элемента из связного списка, что значительно замедляет работу программы.

N	Время работы программы (сек)
10	1.7600017599761486e-05
100	0.00011889997404068708
1000	0.0012818999821320176
10000	0.011218999978154898
100000	0.10281039995606989

Таблица 3 – исследование для “идеальной” хэш-функции

Во втором случае же можно видеть, что при выборе “хорошей” хэш-функции (в нашем случае это *hash(key)%self.capacity*) время удаления элемента из списка происходит намного эффективнее

N	Время работы программы (сек)
10	2.4099950678646564e-05
100	0.00016950001008808613
1000	0.0018047000048682094
10000	0.020459000021219254
100000	0.28297509998083115

Таблица 4 – исследование для “средней” хэш-функции

В третьем исследовании использовалась хэш-функция, которая вычисляет суммарное значение ASCII-кодов всех символов в ключе. Затем это число добавляется к случайному смещению, и результат делится на self.capacity, чтобы получить индекс в хеш-таблице. По времени работы можно заметить, что она не так эффективна, как встроенная в Python функция hash().

3.2. Построение графика для проведенных экспериментов.

Были построены графики, отображающие результаты эксперимента.

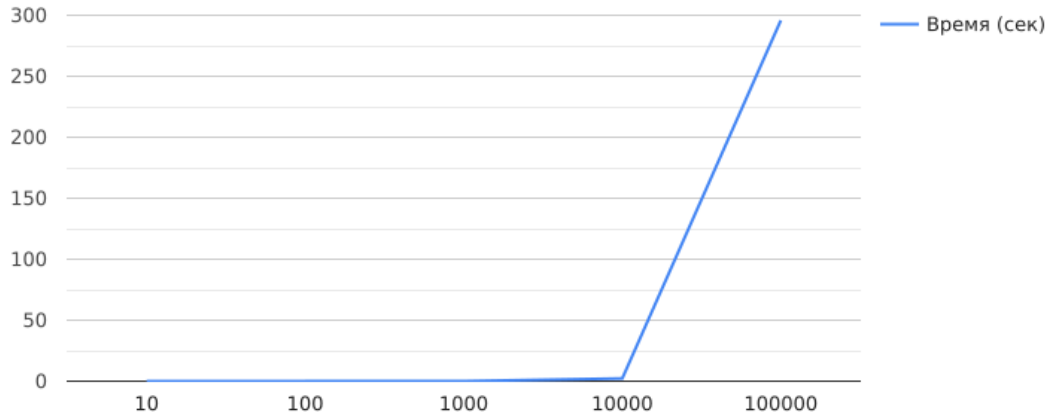

Рисунок 1 – график при использовании “плохой” хэш-функции

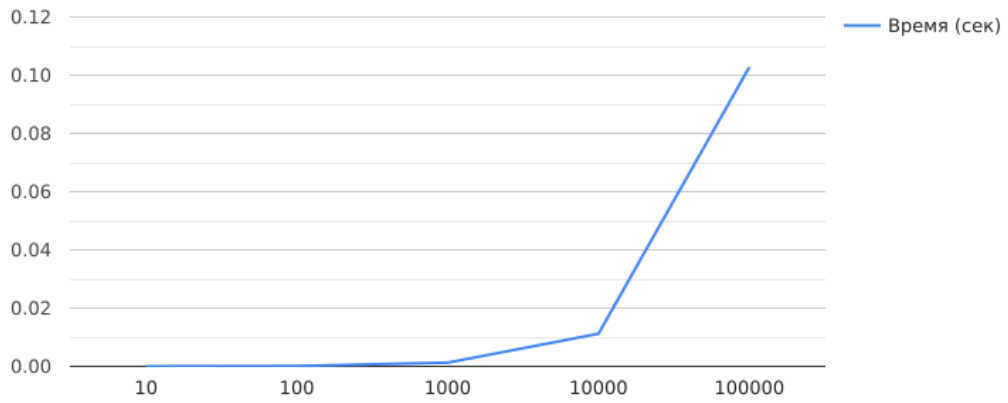

Рисунок 2 – график при использовании “хорошей” хэш-функции

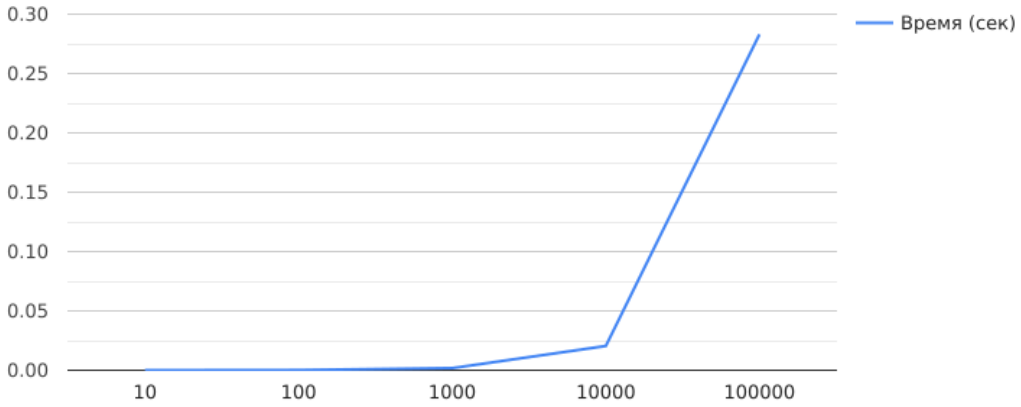

Рисунок 3 – график при использовании “средней” хэш-функции

3.3. Анализ полученных результатов и выводы.

Из результатов исследования можно утверждать, что скорость работы удаления элемента из хэш-таблицы зависит от выбора хэш-функции и вытекающих из нее количества коллизий, а также на прямую зависит от количества удаляемых элементов. В исследовании в качестве “худшего случая” рассматривалась хэш-функция, намеренно возвращающая одно и то же значение и в результате на большом объеме данных работа хэш-таблицы очень замедляется. Если сравнивать работу “лучшей” и “средней” хэш-функции, то можно утверждать, что они схожи по времени работы, но встроенная хэш-

функция hash() в Python работает более оптимизированно, что отражается в разнице по времени удаления на больших числах.

ЗАКЛЮЧЕНИЕ

В результате курсовой работы можно утверждать, что на скорость работы программы огромное влияние оказывает то, какая хэш-функция выбрана. Оценка удаления элемента подтверждена во время исследования.

Все поставленные исследовательские цели в ходе выполнения курсовой работы были выполнены, была реализована система классов для реализации хэш-таблицы, реализовано несколько хэш-функций для исследования времени работы функции удаления элемента из хэш-таблицы, были написаны тесты для проверки корректности работы программы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Habr. Хэш-таблицы. https://habr.com/ru/articles/509220/
2. Wikipedia. Хэш-таблица. https://en.wikipedia.org/wiki/Hash_table
3. Хэш-таблицы.https://docs.google.com/presentation/d/1rmt2n47Fl4-PphjHAKTD5M8x7WrW-5XzebFp4_3YHi4/edit#slide=id.g277ff226500_0_165

ПРИЛОЖЕНИЕ А

НАЗВАНИЕ ПРИЛОЖЕНИЯ

Main.py

```
class HashTableEntry:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.next = None


class HashTable:
    def __init__(self, capacity, max_load_factor=0.7):
        self.capacity = capacity
        self.max_load_factor = max_load_factor
        self.size = 0
        self.table = [None] * capacity

    def calculate_hash(self, key):
        return hash(key) % self.capacity
        # Примеры других хэш-функций
        # base_hash = sum(ord(char) for char in str(key))
        # random_offset = random.randint(0, self.capacity - 1)
        # return (base_hash + random_offset) % self.capacity
        # return 0

    def add(self, key, value):
        index = self.calculate_hash(key)
        if self.table[index] is None:
            self.table[index] = HashTableEntry(key, value)
            self.size += 1
        else:
            new_entry = HashTableEntry(key, value)
            new_entry.next = self.table[index]
            self.table[index] = new_entry
            self.size += 1

        if self.size / self.capacity > self.max_load_factor:
            self.resize_table()

    def find_value(self, key):
        index = self.calculate_hash(key)
        current = self.table[index]
        while current:
            if current.key == key:
                return current.value
            current = current.next

    def resize_table(self):
        old_table = self.table
        self.capacity *= 2
```

```
        self.table = [None] * self.capacity
        self.size = 0

        for entry in old_table:
            current = entry
            while current:
                self.add(current.key, current.value)
                current = current.next

    def delete(self, key):
        index = self.calculate_hash(key)
        previous = None
        current = self.table[index]
        while current:
            if current.key == key:
                if previous:
                    previous.next = current.next
                else:
                    self.table[index] = current.next
                self.size -= 1
                return
            previous = current
            current = current.next
```

Tests.py

```
from main import HashTable

def test_add_and_find_value():
   ht = HashTable(5)
   ht.add('test_key', 'test_value')
   assert ht.find_value('test_key') == 'test_value'

def test_delete_key():
   ht = HashTable(5)
   ht.add('test_key', 'test_value')
   ht.delete('test_key')
   assert ht.find_value('test_key') is None

def test_resize_table():
   ht = HashTable(2)
   ht.add('test_key1', 'test_value1')
   ht.add('test_key2', 'test_value2')
   assert ht.capacity == 4

def test_hash_collision():
   ht = HashTable(2)
   ht.add('test_key1', 'test_value1')
   ht.add('test_key2', 'test_value2')
   assert ht.find_value('test_key1') == 'test_value1'
   assert ht.find_value('test_key2') == 'test_value2'
```