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: Хэш-таблица(квадратичное исследование) - вставка

Студентка гр. 2384 _____ Валеева А. А.

Преподаватель _____ Иванов Д. В.

Санкт-Петербург

2023

ЗАДАНИЕ
НА КУРСОВУЮ РАБОТУ

Студентка: Валеева А. А.

Группа 2384

Тема работы: Хэш-таблица(квадратичное исследование) - вставка

Исходные данные:

Вариант 7

Хэш-таблица(квадратичное исследование) - вставка

Содержание пояснительной записки:
«Аннотация», «Содержание», «Введение», «Описание выполнения работы», «Исследование структур данных», «Заключение», «Список использованных источников», «Исходный код».

Предполагаемый объем пояснительной записки:
Не менее 15 страниц.

Дата выдачи задания:

Дата сдачи реферата: 25.12.2023

Дата защиты реферата: 25.12.2023

Студентка _____ Валеева А. А.

Преподаватель _____ Иванов Д. В.

АННОТАЦИЯ

В ходе выполнения курсовой работы была создана программа на языке программирования *Python*, которая реализует структуру хэш-таблицы. В программе реализовано квадратичное исследование для решение возникающих коллизий. Разработка велась на операционной системе *Linux Ubuntu 22.04 в IDE PyCharm.*

Исходный код программы приведен в приложении А.

СОДЕРЖАНИЕ

1.	Введение	5
2.	Ход выполнения работы	6
2.1	Теоретическое описание структуры данных	6
2.2	Реализация структуры данных	8
2.3	Тестирование	10
2.4	Исследование структуры данных	11
3.	Заключение	15
4.	Список использованных источников	16
5.	Приложение А. Исходный код программы	17

ВВЕДЕНИЕ

Цель работы:

Реализовать структуру данных и заданный в варианте функционал, провести исследования этой структуры.

Задачи:

Для выполнения работы необходимо:

1. Создать структуру хэш-таблицы;
2. Реализовать метод квадратичного исследования функции вставки;
3. Исследовать реализованную структуру с различными хэш-функциями.

В ходе выполнения задания было создано три файла — *main.py, hash_table.py, tests.py.*

2. ХОД ВЫПОЛНЕНИЯ РАБОТЫ

2.1. Теоретическое описание структуры данных.

Хэш-таблица - это структура данных, которая использует массив для хранения данных и хэш-функцию для определения индекса, в который должен быть помещен элемент. Каждый элемент данных хранится в форме пары ключ-значение, где ключ - это уникальное число, используемое для индексации значений, а значение - это данные, которые связаны с этим ключом.

Основной механизм хэш-таблицы - это хэш-функция, которая преобразует ключ в индекс массива(хэш-ключ). Однако, коллизии могут возникнуть, когда две разные ключа преобразуются в один и тот же индекс. Для решения этой проблемы используются различные методы, в моей работе рассмотрен метод квадратичного исследования.

Операции и алгоритмы:

Основные операции, связанные с хэш-таблицей, включают:

- Вставка: добавление нового элемента в хэш-таблицу.
- Поиск: поиск элемента в хэш-таблице по ключу.
- Удаление: удаление элемента из хэш-таблицы по ключу.

При вставке нового элемента, хэш-функция вычисляет индекс ячейка, в которой этот элемент будет храниться. Если происходит коллизия (когда два ключа преобразуются в один и тот же индекс), то используются методы решения коллизий.

Алгоритм поиска в хэш-таблице использует хэш-функцию для определения индекса, в котором хранится искомый элемент. Далее сравниваем значение в хэш-таблице по полученному индексу с искомым значением. Этот процесс обычно выполняется за константное время, что делает поиск в хэш-таблице одним из самых быстрых типов поиска

Удаление элемента из хэш-таблицы также включает использование хэш-функции, но вместо того чтобы просто находить и возвращать значение, мы удаляем элемент из хэш-таблицы. Однако, простое удаление элемента может привести к проблемам, таким как коллизии при поиске элемента, поэтому обычно используется подход, при котором удаленный элемент помечается специальным значением или флагом

Временная и пространственная сложность:

В среднем, вставка, поиск и удаление элемента в хэш-таблице имеют временную сложность *O(1)*, что делает их очень эффективными для большинства задач. Однако, при наличии коллизий или плохо выбранной хэш-функции, производительность хэш-таблицы может снизиться. В таких случаях время выполнения операций может увеличиться до *O(n)*, где *n* - количество элементов в таблице.

Временная сложность квадратичного исследования в хэш-таблице составляет O(*n*). Это происходит потому, что в худшем случае, когда каждый новый элемент вызывает коллизию, мы можем проверить каждую позицию в хэш-таблице перед тем, как найти свободную позицию. Пространственная сложность квадратичного исследования также составляет O(*n*), так как все элементы должны быть хранимы в хэш-таблице. Но квадратичное исследование помогает уменьшить вероятность дальнейших коллизий.

Как мы видим, от выбора хэш-функции зависит частота коллизий, которая увеличивает время выполнения операций. Поэтому проведем исследование на различной длины наборах данных, на которых рассмотрим влияние выбора хэш-фунции.

2.2. Реализация структуры данных.

Файл: *hash_table.py*

Для реализации структуры хэш-таблицы был создан класс *Hash_table.* Конструктор класса получает на вход размер хэш-таблицы, инициализирует поле *size,* отвечающее за количество ячеек в таблице и поле *hash_table*, которое представлено в виде массива длины *size*. Он хранит пары ключ-значение в массивах *[None, None]*.

Далее рассмотрим метод *hash_func(self, key,i).* На вход получает ключ и переменную контроля количества попыток (про нее более подробно будет написано далее). Обе переменные типа *int.* Используя эти параметры, мы вычисляем новое значение — хэш-ключ, который будет индексом хэш-таблицы для хранения пары ключ-значения. Для вычисления используем хэш-функцию, как мы помним, от ее выбора зависит частота коллизий. За идеальную функцию берется встроенная функция *hash().*

Следующий метод — *insert(self, key, value).* Значение *key* имеет тип *int*, а *value* — может быть любого типа данных. Для начала инициализируем переменную контроля количества попыток — *i,* она необходима для поддержания информации о том, сколько раз было произведено попытка разрешения коллизии, и для контроля над тем, сколько раз должен быть выполнен цикл для поиска свободного места в хэш-таблице. Изначально данная переменная получается значение 0. После мы получаем значение hash_key, который будет играть роль индекса в хэш-функции. Его вычисление происходит через описанный ранее метод *hash_func(self, key, i),* куда мы передаем полученный ключ и значения переменной счетчика. Если ячейка по данному индексу занята, то входим в цикл *while,* в котором проверяем следующее: возникла коллизия и разные ключи дали одинаковый индекс или просто совпали ключи, а следовательно, и значение хэш-функции. Рассмотрим подробнее случай с коллизией. Проверяем на совпадение переданный ключ и

ключ, хранимый в ячейке, если они не совпали — вышла коллизия, которую будем решать методом квадратичного исследования. Увеличиваем счетчик *i* на 1 и вызываем снова *hash_func()*, вычисляется новое значение индекса, это делается путем добавления к исходному хэш-ключу некоторого смещения, которое зависит от счетчика *i*. Далее проходим в цикле, пока не найдем таким образом свободную ячейку для записи. Когда она будет найдена — то в *hash_table[hash_key][0]* будет записан ключ вместо *None,* а в *hash_table[hash_key][1]* — значение. Это был рассмотрен случай коллизии. А если проверка на совпадение переданного ключа и ключа, хранимого в ячейке, показала совпадения, то в таком случае просто перезаписываем хранимое в ячейке значение. Выходим из функции.

Выбор алгоритма для разрешения коллизий был основан на методе квадратичного исследования. Этот метод использует квадратичный полином для вычисления следующего индекса при наличии коллизии. Это позволяет эффективно разрешать коллизии, особенно когда они происходят редко.

Метод *print_table()* применяется для удобного вывода хэш-таблицы на экран. Его работа: циклом *for* проходим по всем ячейкам таблицы, если ячейка не пустая — берем из неё пару элементов ключ-значение и с помощью *f-строк* печатаем индекс ячейки, ключ и значение.

Для удобства тестирования были добавлены два метода — *return_element()* и *return_size().* Первый метод принимает индекс и возвращает пару ключ-значение в виде массива по данному индексу. А второй метод возвращает значения поля *size* у класса *Hash_table.*

Файл: *main.py*

Импортируем используемые файлы.

Для начала создаем таблицу *ht* как объект класса *Hash_table* и указываем размер. После на экран выводится строка с объявлением длины хэш-таблицы и сообщение о том, что нужно ввести число от 0 до длины. Пользователь вводит

выбранное число, а далее циклом *for* вызываем описанный метод *insert()*. Передаваемые значение генерируются методом *random_randint*. После вызывается метод печати *print_table()*.

2.3. Тестирование.

Файл: tests.py

Для проверки корректности работы функции были написаны тесты *pytest*. Тестирование программы и примеры использования реализованных классов представлено в *Таблице 1*.

Таблица 1 - Тестирование

№ п/п	Входные данные	Выходные данные	Комментарии
1.	Введите число от 0 до 10: 3	index: 2, key: 33, value: 45 index: 5, key: 36, value: 62 index: 8, key: 37, value: 54	Заданы различные ключи, значения хэш-функций подсчитано верно
2.	Введите число от 0 до 10: 2	index: 0, key: 9, value: hi! index: 4, key: 5, value: hi!	Вставка строки сработала корректно, значения хэш-функций подсчитано верно
3.	Введите число от 0 до 10: 5 ht.insert(0, value)	index: 9, key: 0, value: hi!	При одинаковом ключе значения *value* будет заменяться
4.	Введите число от 0 до 10: 0	index: 0, key: None, value: None index: 1, key: None, value: None index: 2, key: None, value: None index: 3, key: None, value: None index: 4, key: None, value: None index: 5, key: None, value: None	Изменена функция вывода — печать всех ячеек, даже пустых. Таблица не заполнена

		index: 6, key: None, value: None index: 7, key: None, value: None index: 8, key: None, value: None index: 9, key: None, value: None	
5.	ht.insert(3, "hi") ht.insert(7, "alina")	index: 5, key: 7, value: alina index: 8, key: 3, value: hi	При ключах 7 и 3 значение хэщ-функции равно 8, возникла коллизия. Для 7 верно пересчитано значение хэш-функции

Разработанный программный код см. в приложении А.

2.4. Исследование структуры данных.

Как уже было описано, от выбора хэш-функции будет зависеть частота коллизий, следовательно и сложность выполнения задачи. Исследование будет проводиться на различных объемах данных 10, 100, 1000, 10000, 100000, для которых мы будем использовать три различных хэш-функции — лучшую: встроенную функцию хэширования *hash(),* среднюю, которая будет делать смещение в зависимости от значения счетчика *i,* и худшую, при которой будет возникать большое количество коллизий. Изначальна идея эксперимента заключалась в замере времени вставки одного элемента в таблицу, заполненную на 50%, но из-за слишком малого наличия коллизий эксперимент не дал особого результата, на всех функциях и наборах данным время выходило примерно одинаковое: 4-7*10^(-7) секунд. Поэтому проведем следующий эксперимент: замерим время заполнения тремя различными функциями всей таблицы. Засекать время будет через библиотеку *time,* которую прежде импортируем.

Результаты исследования представлены на Рис. 1 — Рис. 5.

Результаты исследования:

Рис. 1 — Исследование на худшей хэш-функции

Рис. 2 — Исследование на средней хэш-функции

Рис. 3 — Исследование на лучшей хэш-функции

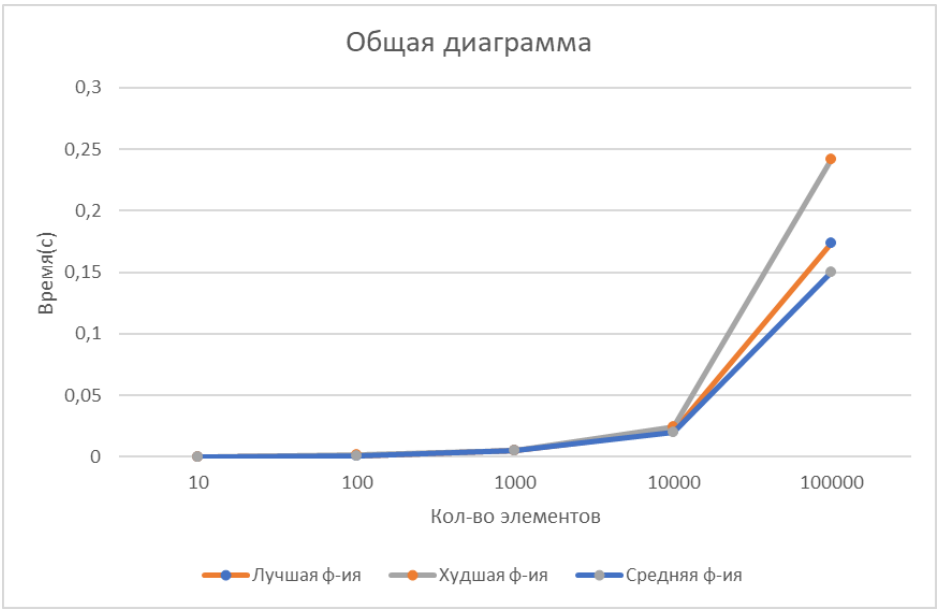

Рис. 4 - График всех трех функций

Из результатов исследований видно, что средняя и лучшая функции дают примерно равный по времени результат, в некоторых случаях лучшая функция уступает средней. Значит функция, выбранная мной, была выбрана верно и давала минимальное количество коллизий.

Теоретическая оценка дает сложность операции — *O(n)*, где *n* — количество элементов в таблице. Из график можно сделать вывод, что данная оценка была подтверждена на практике. На небольших наборах данных оценка вышла *O(1)*, что означает, что среднее время выполнения этой операции

остается постоянным, независимо от размера хэш-таблицы или количества элементов в ней.

ЗАКЛЮЧЕНИЕ

В результате выполнения курсовой работы была реализована структура данных хэш-таблица, описана одна из функций — функция вставки, а так же проведено исследование данной структуры.

Исследование дало положительный результат — теоретические показатели подтвердились на практике, полученный результат соответствует поставленной цели. В моей курсовой работе можно изменить способ хранения ключа-значения, добавить, например, пометку свободной ячейки, что будет удобно для реализации других функций. Также можно оптимизировать алгоритм разрешения коллизий с помощью преобразования выбранной хэш-функции.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Хеш-таблицы:

https://ru.wikipedia.org/wiki/Хеш-таблица

2. Лекции по алгоритмам и структурам данных

https://proglib.io/p/data-structure-algorithms/

3. Современные словари в Python:

https://www.youtube.com/watch?v=37S53yFg9wc

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл: hash_table.py

```
class Hash_table:
   def __init__(self, size_value):
       self.size = size_value
       self.hash_table = [[None, None] for _ in range(self.size)]

   def hash_func(self, key, i):
       hash_key = ((key**2 - 1) % self.size + 2*i + 5*(i**2) )

       return hash_key % self.size

   def insert(self, key, value):
       i = 0
       hash_key = self.hash_func(key, 0)
       while (self.hash_table[hash_key][0] is not None): #ячейка
занята
           if self.hash_table[hash_key][0] == key: # но ключи
совпали
               self.hash_table[hash_key][1] = value #перезаписываю
               return
           i += 1 #иначе ищу другую ячейку
           hash_key = self.hash_func(key, i)

       self.hash_table[hash_key][0] = key
       self.hash_table[hash_key][1] = value
   def print_table(self):
       for i in range(self.size):
           my_set = self.hash_table[i]
           if my_set[1] is not None:
                print(f"index: {i}, key: {my_set[0]}, value:
{my_set[1]}")
```

```
    def return_element(self, ind):
        return self.hash_table[ind]

    def return_size(self):
        return self.size
```

Файл: main.py

```
import random
import time
from hash_table import Hash_table
ht = Hash_table(10)
print("Введите число от 0 до 100: ")
n = int(input())
start_time = time.time()
for i in range(n):
    key = random.randint(0, 10)
    value = random.randint(-1000, 1000)
    ht.insert(key, value)
end_time = time.time()
execution_time = end_time - start_time
ht.print_table()
answer = ht.return_element(2)
print(answer)
print(f"Время выполнения: {execution_time} секунд")
```

Файл: tests.py

```
import pytest
from hash_table import Hash_table

def test_1():
    ht = Hash_table(10)
    ht.insert(33, 45)
    ht.insert(36, 62)
    ht.insert(37, 54)
    my_answer = ht.return_element(2)
```

```
    answer = [36, 62]
    assert my_answer == answer

def test_2():
    ht = Hash_table(10)
    ht.insert(1, "hi!")
    ht.insert(1, "alina")
    my_answer = ht.return_element(2)[1]
    answer = "alina"
    assert my_answer == answer

def test_3():
    ht = Hash_table(10)
    ht.insert(1, "hi!")
    ht.insert(1, "alina")
    my_answer = ht.return_element(2)[1]
    answer = "alina"
    assert my_answer == answer

def test_4():
    ht = Hash_table(10)
    my_answer = ht.return_size()
    answer = 10
    assert my_answer == answer

def test_4():
    ht = Hash_table(10)
    my_answer = ht.return_element(3)
    answer = [None, None]
    assert my_answer == answer
```