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

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

Тема: «Исследование абстрактных структур данных. Реализация хеш-таблицы с методом цепочек»

Студент гр. 0304 _____ Мажуга Д.Р.

Преподаватель _____ Берленко Т.А.

Санкт-Петербург,
2021

ЗАДАНИЕ

НА КУРСОВУЮ РАБОТУ

Студент Мажуга Д.Р.

Группа 0304

Тема работы: Исследование абстрактных структур данных. Реализация хеш-таблицы с методом цепочек

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:

Перечисляются требуемые разделы пояснительной записки (обязательны разделы «Содержание», «Введение», «Заключение», «Список использованных источников»)

Предполагаемый объем пояснительной записки:

Не менее 4 страниц.

Дата выдачи задания: 26.10.2021

Дата сдачи реферата: 23.12.2021

Дата защиты реферата: 24.12.2021

АННОТАЦИЯ

В данной работе была исслледована структура данных — хэш-таблица с методом цепочек. Данная структура данных реализована на Python. В работе была проведена проверка корректности реализаций и замеры фактического времени работы некоторых операций в исследуемой структуре данных.

СОДЕРЖАНИЕ

1.ВВЕДЕНИЕ

1.1. Цель курсовой работы и исходные условия.

Исходные условия:

"*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.

Целью данной работы является реализация хэш-таблицы с методом цепочек и тестирование ее характеристик производительности в зависимости от входных данных.

1.2.Задачи

Для решения поставленной задачи необходимо:

1. Реализовать хэш-таблицу.
2. Убедиться в корректности реализованной структуры.
3. Исследовать вставку, поиск и удаление элемента в структуре данных.

2. РЕАЛИЗАЦИЯ ТРЕБУЕМОЙ СТРУКТУРЫ ДАННЫХ

Требуемые структуры данных и их обработка была написана с помощью языка Python 3.9, с использованием библиотеки matplotlib для построения графиков, модуля random для генерации входных данных.

2.1. Хэш-таблица. Сведения.

Хеш-таблица — это структура данных, реализующая интерфейс ассоциативного массива, а именно, она позволяет хранить пары (ключ, значение) и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу.

Существуют два основных варианта хеш-таблиц: с цепочками и открытой адресацией. Хеш-таблица содержит некоторый массив H, элементы которого есть пары (хеш-таблица с открытой адресацией) или списки пар (хеш-таблица с цепочками).

Выполнение операции в хеш-таблице начинается с вычисления значения хеш-функции от переданного ключа. Полученное значение — хеш — играет роль индекса в массиве H.

Ситуация, когда для различных ключей получается одно и то же хешзначение, называется коллизией. Для эффективного использования хеш-таблицы очень важно подобрать хорошую хеш-функцию, которая на случайном наборе входных данных будет выдавать равномерно распределённые хеш значения.

Для разрешения коллизий элементов используются различные методы. В данной работе исследуется метод цепочек.

2.2 Хэш-таблица с методом цепочек.

Технология сцепления элементов состоит в том, что *элементы множества*, которым соответствует одно и то же хеш-значение, связываются в цепочку-список. В позиции номер *i* хранится указатель на *голову списка* тех элементов, у которых хеш-значение ключа равно *i*, если таких элементов в

множестве нет, в позиции i записан *NULL*. На рис. 1 демонстрируется реализация метода цепочек при разрешении *коллизий*. На ключ 002 претендуют два значения, которые организуются в линейный список.

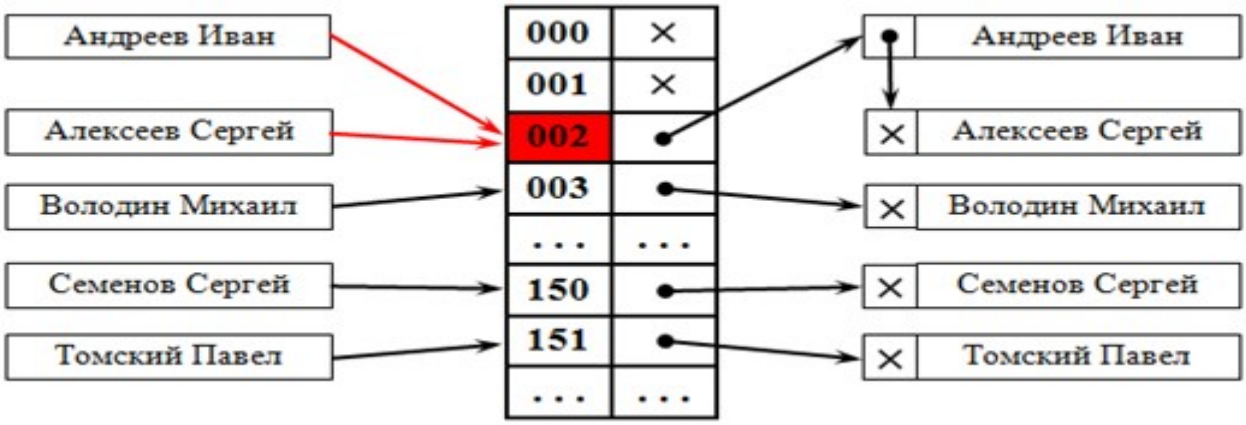

Рис 1. - Разрешение коллизий при помощи цепочек

Каждая ячейка массива является указателем на связный список (цепочку) пар ключ-значение, соответствующих одному и тому же хеш-значению ключа. *Коллизии* просто приводят к тому, что появляются цепочки длиной более одного элемента.

Операции поиска или удаления данных требуют просмотра всех элементов соответствующей ему цепочки, чтобы найти в ней элемент с заданным ключом. Для добавления данных нужно добавить элемент в конец или начало соответствующего списка, и, в случае если коэффициент заполнения станет слишком велик, увеличить размер массива и перестроить таблицу.

2.3. Реализация хэш-таблицы с методом цепочек.

Для реализации хеш-таблицы был реализован класс *ListNode* который создаёт узел списка, который нам потребуется для реализации хэш-таблицы. В классе *Node* был написан конструктор, при вызове которого будет создан узел списка в котором находятся ключ и его значение, а также указатель на следующий элемент списка, все эти значения по-умолчанию равны None. Далее был реализован класс *HashTable* в котром мы не посредственно реализуем нашу хэш-таблицу с методом цепочек. В конструкторе класса, у нас хранится размер таблицы и клетки в которых будут находиться наши списки.

В классе реализованны следующие методы:

1. *hash(key)* — непосредственно наша хеш-функция, которая переводит строку-ключ в последовательность символов, и возвращает значения от 0 до *n*-1, где n это размер нашей таблицы.
2. *add(key, value)* — метод реализующий добавления ключа и его значения в нашу таблицу. Находиться хеш переданного нами ключа, после чего мы проверяем существует ли элемент с таким хешем если нет то мы записываем пару(ключ, значение) в эту клетку, иначе мы вставляем элемент в начало списка и указываем на пару которая там лежала. Также в данном методе проверяется существует ли в списке пара с таким ключом, если да, то мы перезаписываем пару на новую.
3. *search(key)* — метод реализующий поиск пары в нашей таблице. Сначала мы находим хеш переданного нам ключа, после чего мы проверяем есть ли хоть одна пара по данному хешу, если же нет то возвращаем *False* и завершаем поиск, в противном случае мы совершаем обход по списку проверяя ключи, когда ключ найден возвращаем *True* и заканчиваем поиск.
4. *remove(key)* — метод реализующий удаление пары из нашей таблицы. Вычисляем хеш переданного нам ключа, совершаем проверку на существование элементов по данному хешу, если нет то возвращаем *False*, если пара всё же существует, то совершаем обход по списку, для успешного удаления нам требуются сохранять предыдущую пару в пременную prev, как только мы находим нужную нам пару то мы меняем указатели таким образом, что предыдущий элемент теперь будет указывать на следующий элемент текущего, если же список состоял из одной пары, то после удаления в клетки будет лежать значение *None.*

3. ПРОВЕРКА КОРРЕКТНОСТИ РЕАЛИЗОВАННОЙ СТРУКТУРЫ

2.1. Описание тестового модуля тестирования

Для проверки корректности работы выше описанной хеш-таблицы с методом цепочек, были написаны тесты , проверяющий корректность вставки, нахождения и удаления элемента. Для проверки, методы вставки, нахождения и удаления элемента, возвращают *True* либо *False* от успеха операции тем самым в тестовом модуле мы проверяем возвращаемые значения.

2.2. Тестирование

В тестовом модуле генерируются 100 тыс. строк которые в последствии вставляются, находятся, и удаляются. С помощью assert мы проверяем значение метода с *True*(поскольку при успешном выполнении метод возвращает *True*), если же на какой-то из оперций происходит ошибка, то тестирование завершается и в консоль выводиться в какой из оперций произошла ошибка.

При тестировании мы в цикле выполняем выше описанные тесты(число итераций при тестах было равно 10). В ходе тестирования ни одного сообщения об ошибке не было выведено, из чего следует , что таблица работает в соответсвии с желаемым результатом.

4. АНАЛИЗ ТРЕБУЕМЫХ ОПЕРАЦИЙ

4.1. Реализация функционала для анализа данных

Для исследования необходимо генерировать случайные входные данные. В качестве входных данных были выбраны строки. Для генерации случайных строк была написана функция, которая принимает в качестве аргумента требуемую длину строки. Случайная генерация происходит следующим образом: из строки, в которой указаны ascii символы, которые можно отобразить в консоли. Происходит итеративное добавления случайного символа к строке с помощью метода *random.choice()*, пока не будет сформирована строка требуемой длинны.Также была написанна функция *get_time_process(string_number, size)* — которая считает время затраченное процесором на выполнение операции, в данной функции создаётся хеш-таблица размер которой передаётся во втором аргументе, после чего случайно генерируются строки, количество которое нам нужно сгенерировать мы передаём первым аргументом в функцию, далее в цикле мы выполняем операции вставки, поиска и удаления, с помощью метода *time.process_time()* мы считаем время на выполнение каждой операции, функция возвращает список с затраченным временем на каждую оперцию.

Написаны функции тестов для тестов лучшего, среднего и худшего случаев вставки, поиска и удаления. В которых мы в цикле увеличиваем размер таблицы (размер увеличивался на степень двойки тем самым таблица имела следующие размеры [256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072]) с читаем время произведенных операций с помощью выше описанной функции *get_process_time()*, в качестве первого аргумента мы передаём размер таблицы умноженный на коэффициент загруженности(для хкдшего случая коэффициент заполения равен 0.9 , для среднего равен 0.6 и для лучшего он равен 0.3), в качестве второго мы передаём сам размер. По завершению всех итераций мы возвращаем списки времени работы каждой опреции для худшего, среднего и лучшего случая соответсвенно.

Была написана функция сравнения каждого кейса, в котором мы выводим графики с помощью импортированного модуля *matplotlib.pyplot*. На графиках отображена зависимость скорости работы операций и размера хеш-таблицы.

4.2. Анализ полученных данных

4.2.1. Анализ вставки элемента в лучшем, среднем и худшем случаях

Теоретически операция вставки элемента должна занимать O(1) в лучшем случае, O(1 + a) в среднем и O(n) в худшем случае. На рис. 2. преведены практические результаты исследования:

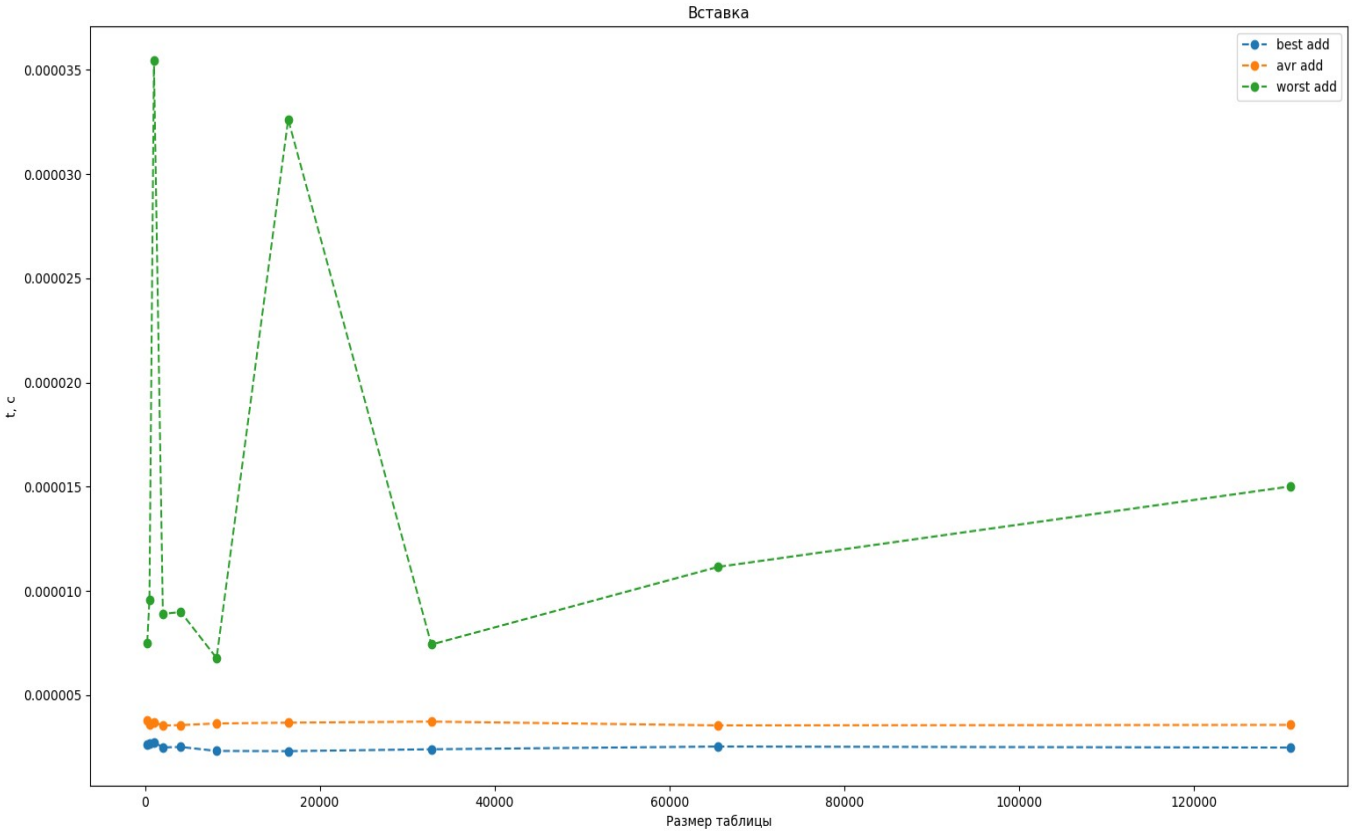

Рис. 2. - График зависимости времени от размера таблицы на операции вставки

Исходя из данных на графике, операция вставки в лучшем случае действительно занимает O(1) времени, в среднем же случае вставка занимает O(1 + a), а в худшем случае мы видим сильную тенденцию к росту, при увеличении таблицы в несколько раз время на операцию тоже увеличивается в несколько раз, следовательно в худшем случае вставка занимает O(n).

4.2.2. Анализ поиска элемента в лучшем, среднем и худшем случаях

Теоретически операция вставки элемента должна занимать O(1) в лучшем случае, O(1 + a) в среднем и O(n) в худшем случае. На рис. 3. преведены практические результаты исследования:

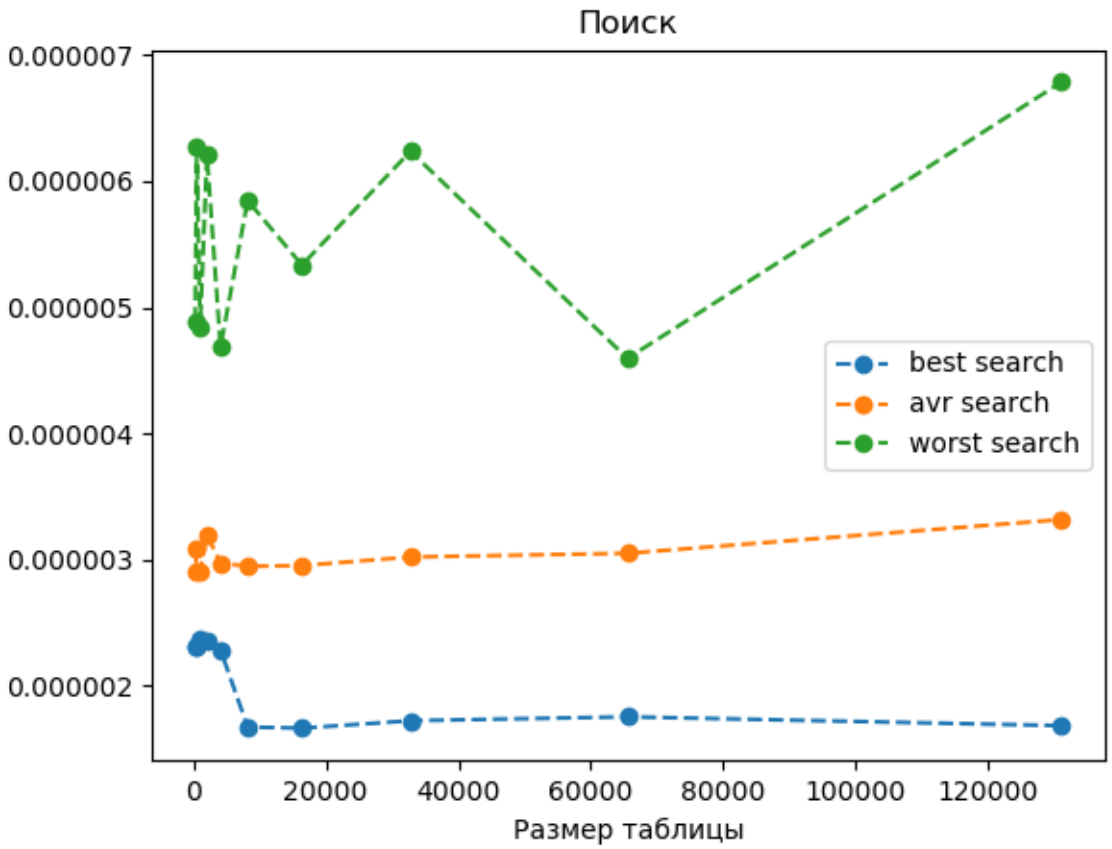

Рис. 3. - График зависимости времени от размера таблицы на операции поиска

Исходя из данных на графике, операция вставки в лучшем случае действительно занимает O(1) времени, при среднем же случае мы видим не большую тенденцию к росту и в среднем случае поиск занимает O(1 + a), всё это происходит из-за того что при увеличении коэффициента загруженности количество элементов в списке растёт, соответсвенно и мы тратим больше времени на обход списка. В худшем же случае мы видим сильную тенденцию к росту и скорость работы поиска можно оценить в O(n), это связано всё с тем же ростом элементов в списке.

4.2.3.Анализ удаления элемента в лучшем, среднем и худшем случаях

Теоретически операция вставки элемента должна занимать O(1) в лучшем случае, O(1 + a) в среднем и O(n) в худшем случае.. На рис. 4. преведены практические результаты исследования:

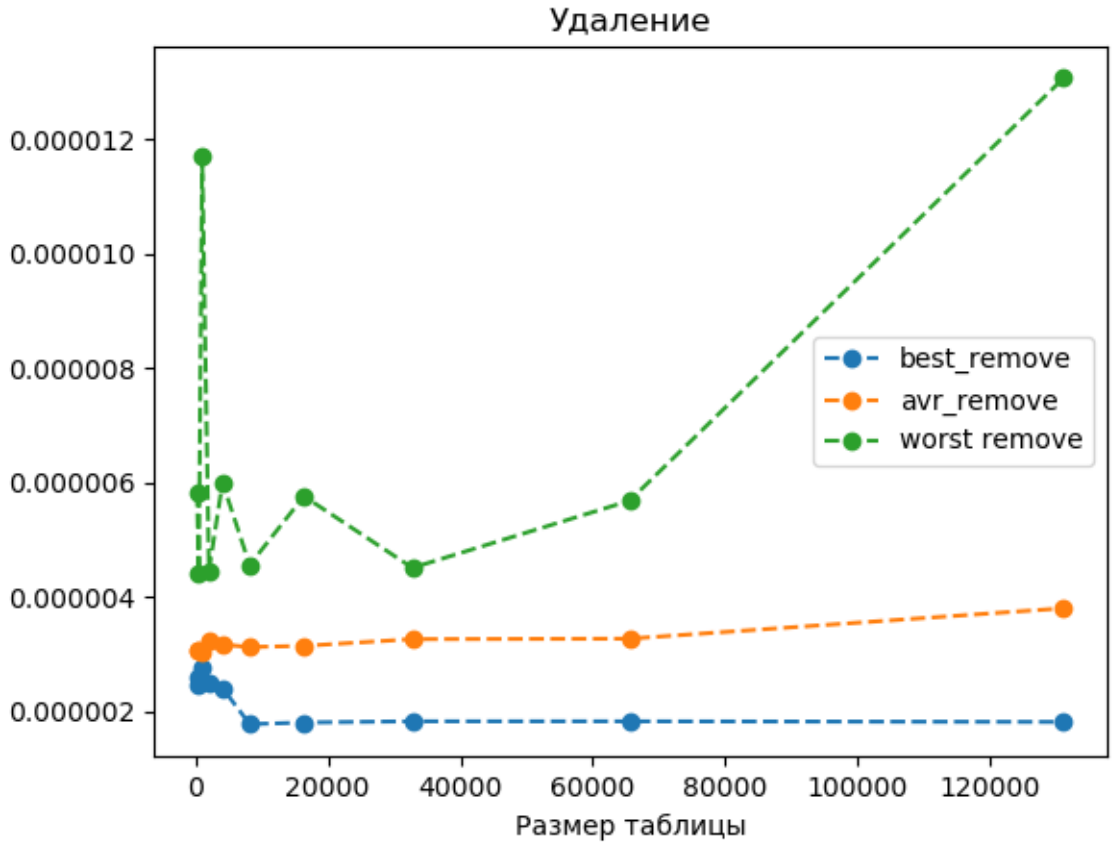

Рис. 4. - График зависимости времени от размера таблицы на операции удаления

Анализируя данные с графика мы видим, что лучшем случае удаление действительно занимает O(1), это связано с тем что при маленьком коэффициенте загруженности, количество элементов в списке которые мы должны обойти очень мало или же элемент вовсе одни, исходя из этого время которые должно затратиться на удаление в среднем случае должно расти, и действительно среднее время удаления занимает O(1 + a), на графике видно не большую тенденцию к росту, соответсвенно в худшем случае график имеет сильную тенденцию к росту из-за большого коэфициента заполнености и время затраченое на удаление занимает O(n).

ЗАКЛЮЧЕНИЕ

Поставленные задачи были выполнены и в результате работы была реализована хэш-таблица с методом цепочек и проведено исследование методов вставки, поиска и удаления. Методы реализованной структуры данных корректно работают, выдавая ожидаемые значения. Сложность методов по вставке, поиску и удалению составляет O(n) в худшем случае, O(1 + a) в среднем случае и O(1) в лучшем случае, что было установлено теоретически и подтверждено в ходе исследования.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Википедия:https://ru.m.wikipedia.org/wiki/%D0%A5%D0%B5%D1%88-%D1%82%D0%B0%D0%B1%D0%BB%D0%B8%D1%86%D0%B0

2. Видео урок по построению графиков с помощью matplotlib: https://www.youtube.com/watch?v=ELFPpTzxNu8&t=284s

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Node.py:

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

HashTable.py:

```
from Node import ListNode


class HashTable:

    def __init__(self, size):
        self.size = size
        self.cells = [None] * self.size

    def hash(self, key):
        sum = 0
        i = 1
        for pos in range(len(key)):
            sum += ord(key[pos]) * i
            i += 1
        return sum % self.size

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

    def add(self, key, value) -> True:
        pos = self.hash(key)
        if self.cells[pos] is None:
            self.cells[pos] = ListNode(key, value)
            return True
        h = head = self.cells[pos]
        while h:
            if h.key == key:
                h.value = value
                return True
            h = h.next
        self.cells[pos] = ListNode(key, value, head)
        return True

    def search(self, key) -> True:
        pos = self.hash(key)
        if self.cells[pos] is None:
            return False
        head = self.cells[pos]
        while head:
            if head.key == key:
```

```
            return True
        head = head.next
    return False

def remove(self, key) -> True:
    pos = self.hash(key)
    if self.cells[pos] is None:
        return False
    else:
        head = self.cells[pos]
        if head.key == key:
            self.cells[pos] = head.next
            return True
        prev = h = head
        while h:
            if h.key == key:
                prev.next = h.next
                return True
            prev, h = h, h.next
```

research.py:

```
from HashTable import HashTable
import random
import time
import matplotlib.pyplot as plt



def generate_random_string(str_size: int):
    result = ''
    for i in range(str_size):
        result +=
random.choice("0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"
                      "#$%&'()*+,-./:;<=>?\@[]^_`{|}~")
    return result



def check_if_exist(list_str: list, string: str):
    for i in list_str:
        if HashTable.hash(i) is HashTable.hash(string):
            return True
    return False



def generate_uniq_string(list_size: int):
    result = []
    for i in range(list_size):
        string = generate_random_string(int(random.random() * 100) % 11 +
int(random.random() * 100) % 5 + 3)
        if not check_if_exist(result, string):
            result.append(string)
        return result

def get_time_process(string_number: int, size: int):
    ht = HashTable(size)
    strings = [generate_random_string(int(random.random() * 100) % 11 + int(random.random()
* 100) % 5 + 3) for _ in
```

```
            range(string_number)]
    add_times = []
    for i, string in enumerate(strings):
        time_start = time.process_time()
        ht.add(string, i + 1)
        add_times.append(time.process_time() - time_start)
    search_times = []
    for i, string in enumerate(strings):
        time_start = time.process_time()
        ht.search(string)
        search_times.append(time.process_time() - time_start)
    remove_times = []
    for i, string in enumerate(strings):
        time_start = time.process_time()
        ht.remove(string)
        remove_times.append(time.process_time() - time_start)

    return add_times, search_times, remove_times



def best_research_case():
    add_times_list = []
    search_time_list = []
    remove_times_list = []
    load_factor = 0.3
    n = 8
    size = 2**n
    number_of_iter = 10

    for _ in range(number_of_iter):
        times = get_time_process(int(size*load_factor), size)
        add_times_list.append(times[0])
        search_time_list.append(times[1])
        remove_times_list.append(times[2])
        n += 1
    return add_times_list, search_time_list, remove_times_list



def average_research_case():
    add_times_list = []
    search_time_list = []
    remove_times_list = []
    load_factor = 0.6
    n = 8
    size = 2**n
    number_of_iter = 10

    for _ in range(number_of_iter):
        times = get_time_process(int(size*load_factor), size)
        add_times_list.append(times[0])
        search_time_list.append(times[1])
        remove_times_list.append(times[2])
        n += 1
    return add_times_list, search_time_list, remove_times_list



def worst_research_case():
    add_times_list = []
```

```
    search_time_list = []
    remove_times_list = []
    load_factor = 0.9
    n = 8
    size = 2**n
    number_of_iter = 10

    for _ in range(number_of_iter):
        times = get_time_process(int(size*load_factor), size)
        add_times_list.append(times[0])
        search_time_list.append(times[1])
        remove_times_list.append(times[2])
        n += 1
    return add_times_list, search_time_list, remove_times_list



def compare_result(best_result, avrage_result, worst_result):
    best_add = best_result[0]
    best_search = best_result[1]
    best_remove = best_result[2]

    average_add = avrage_result[0]
    average_search = avrage_result[1]
    average_remove = avrage_result[2]

    worst_add = worst_result[0]
    worst_search = worst_result[1]
    worst_remove = worst_result[2]

    size_table = [2**i for i in range(8, 18)]

    #Графики иследованиий
    plt.title('Вставка')
    plt.xlabel('Размер таблицы')
    plt.ylabel('t, c')
    plt.plot(size_table, [min(elem) for elem in best_add],  '--o', label="best add")
    plt.plot(size_table, [sum(elem) / len(elem) for elem in average_add],  '--o', label="avr add")
    plt.plot(size_table, [max(elem) for elem in worst_add],  '--o', label="worst add")
    plt.legend()
    plt.show()

    plt.title('Поиск')
    plt.xlabel('Размер таблицы')
    plt.ylabel('t, c')
    plt.plot(size_table, [min(elem) for elem in best_search],  '--o', label="best search")
    plt.plot(size_table, [sum(elem) / len(elem) for elem in average_search],  '--o', label="avr
search")
    plt.plot(size_table, [max(elem) for elem in worst_search],  '--o', label="worst search")
    plt.legend()
    plt.show()

    plt.title('Удаление')
    plt.xlabel('Размер таблицы')
    plt.ylabel('t, c')
    plt.plot(size_table, [min(elem) for elem in best_remove], '--o', label="best_remove")
    plt.plot(size_table, [sum(elem) / len(elem) for elem in average_remove], '--o',
label="avr_remove")
    plt.plot(size_table, [max(elem) for elem in worst_remove], '--o', label="worst remove")
```

```
    plt.legend()
    plt.show()


if __name__ == '__main__':
    compare_result(best_research_case(), average_research_case(), worst_research_case())
```

tests.py:

```
from HashTable import HashTable
from research import generate_uniq_string


class MyTestCase:
    def test_hash_table(self):
        ht = HashTable(997)
        strings = generate_uniq_string(1000000)
        iter_number = 10

        for _ in range(iter_number):
            for i, string in enumerate(strings):
                assert ht.add(string, i + 1) is True
            print('test add: OK')
            for i, string in enumerate(strings):
                assert ht.search(string) is True
            print('test search: OK')
            for i, string in enumerate(strings):
                assert ht.remove(string) is True
            print('test remove: OK')
        print('all tests passed')


if __name__ == '__main__':
    case = MyTestCase()
    case.test_hash_table()
```