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 (открытая адресация) vs Хеш-таблица (метод цепочек). Исследование.

Студент гр. 1304 _____ Заика Т.П.

Преподаватель _____ Иванов Д.В.

Санкт-Петербург

2022

ЗАДАНИЕ
НА КУРСОВУЮ РАБОТУ

Студент: Заика Т.П.

Группа: 1304

Тема работы: Хеш-таблица (открытая адресация) vs Хеш-таблица (метод цепочек). Исследование.

Исходные данные:
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.
Содержание пояснительной записки:
«Содержание», «Введение», «Анализ операций хеш-таблицы (метод цепочек)», «Анализ операций хеш-таблицы (открытая адресация)», «Сравнение операций хеш-таблицы (метод цепочек) и хеш-таблицы (открытая адресация)», «Заключение», «Список использованных источников»

Предполагаемый объем пояснительной записки:
Не менее 25 страниц.

Дата выдачи задания: 25.10.2022

Дата сдачи реферата: 20.12.2022

Дата защиты реферата: 23.12.2022

Студент _____ Заика Т.П.

Преподаватель _____ Иванов Д.В.

АННОТАЦИЯ

Исследование операций вставки, поиска и удаления для хеш-таблицы (метод цепочек) и хеш-таблицы (открытая адрессация). Сравнение операций между структурами данных. Реализация структур произведена на языке программирования Python, анализ операций произведен при помощи библиотек matplotlib и numpy. Определено, что хеш-таблица является абстрактной структурой данных, то есть временная сложность операций не зависит от способа реализации структуры данных.

SUMMARY

Investigation of insert, lookup and delete operations for a hash table (chaining method) and a hash table (open addressing). Comparison of operations between data structures. The structures are implemented in the Python programming language, the operations are analyzed using the matplotlib and numpy libraries. It is defined that a hash table is an abstract data structure, that is, the time complexity of operations does not depend on how the data structure is implemented.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель работы заключается в сравнении операций вставки, поиска и удаления хеш-таблицы, которая разрешает колизии методом цепочек, и хеш-таблицы, которая разрешает колизиии методом открытой адресации, с линейным, квадратичным и двойного хеширования пробированием (исследованием). Для достижения цели выполнены задачи реализации абстрактных структур данных на языке программирования Python и анализ операций каждой из них в лучшем, среднем и худшем случаях для дальнейшго сравнения операций между структурами данных. В ходе решения задач применены методы разработки в парадигме ООП, использование библиотек matplotlib и numpy, а также стандартной библиотеки Python, для построения графиков временной сложности в терминах О-большое.

1. АНАЛИЗ ОПЕРАЦИЙ ХЕШ-ТАБЛИЦЫ (МЕТОД ЦЕПОЧЕК)

1.1. Вставка

Входные данные для анализа операций хеш-таблицы (метод цепочек) опередлены для таблицы с размером 5 в среднем и лучшем случае, с размером 9 в худшем случае. В худшем случае результат хеш-функции для всех ключей из входного набора данных будет одинаковым, что приведет к колизии. В качестве ключей принимаются только строки произвольной длины, т.к. используется полиномиальная хеш-функция, в качестве значений принимается любое изменяемое значение.

Входные данные представлены ниже:

```
inp_dict = {
    "123": "23",
    "14": "14",
    "1": "3",
    "556": "900",
    "1456": "12"
}
```

Рис. 1 — Входные данные в среднем и лучшем случае

```
inp_dict = {
    "'": "23",
    ":": "14",
    "M": "3",
    "`": "900",
    "s": "12",
}
```

Рис. 2 — Входные данные в худшем случае

Результат в среднем и лучшем случае (Рис. 3) по временной сложности не превосходит теоретический, сложность которого O(1).

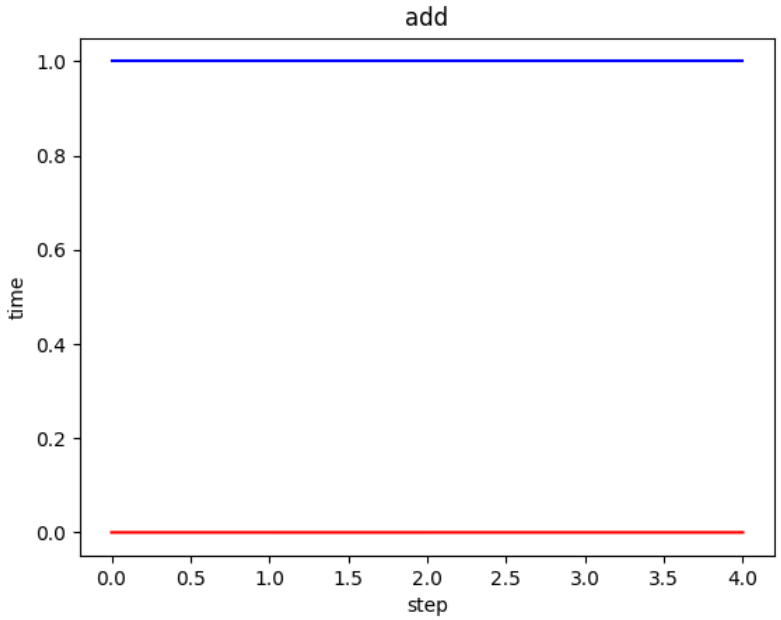


Рис. 3 — Затрата времени на операцию вставки с среднем и лучшем случае в сравнении с теоретической затратой времени (красным — время на входных данных, синим — теоретическое время)

Результат в худшем случае (Рис. 4), когда при каждой операции вставки возникает колизия и нужно произвести вставку в линейный список по полученному номеру ячейки, по временной сложности не превосходит теоретический, сложность которого O(n).

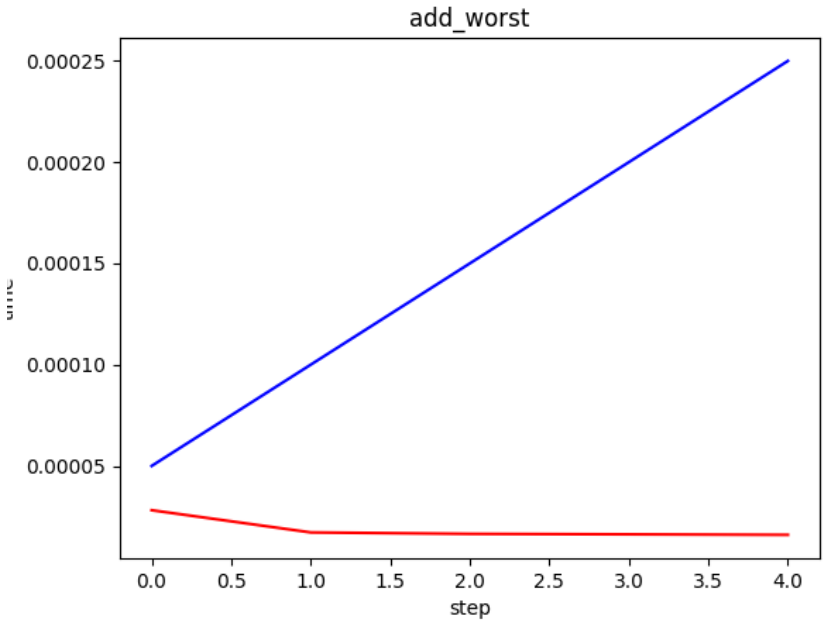


Рис. 4 — Затрата времени на операцию вставки в худшем случае в сравнении с теоретической затратой времени (красным — время на входных данных, синим — теоретическое время)

1.2. Поиск

Результат в среднем и лучшем случае (Рис. 5) по временной сложности не превосходит теоретический, сложность которого O(1).

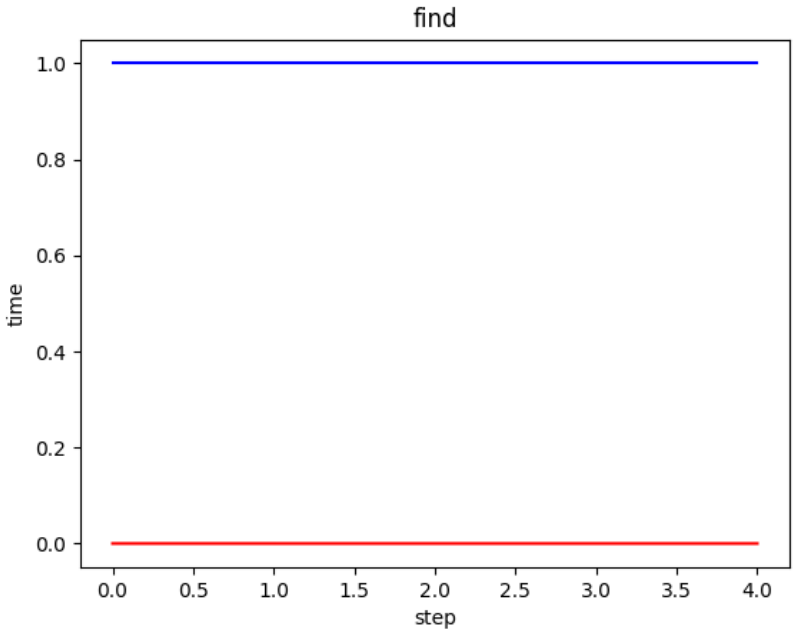


Рис. 5 — Затрата времени на операцию поиска в среднем и лучшем случае в сравнении с теоретической затратой времени (красным — время на входных данных, синим — теоретическое время)

Результат в худшем случае (Рис. 5), когда необходимо совершить поиск элемента, находящегося в линейном списке, по полученному номеру ячейки, по временной сложности не превосходит теоретический, сложность которого O(n).

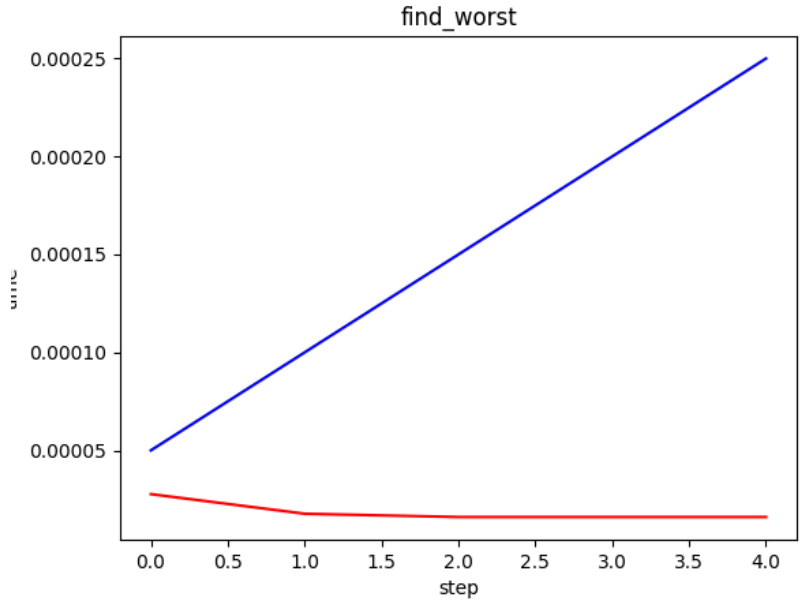


Рис. 6 — Затрата времени на операцию поиска в худшем случае в сравнении с теоретической затратой времени (красным — время на входных данных, синим — теоретическое время)

1.3 Удаление

Результат в среднем и лучшем случае (Рис. 7) не превсходит теоретический, сложность которого O(1).

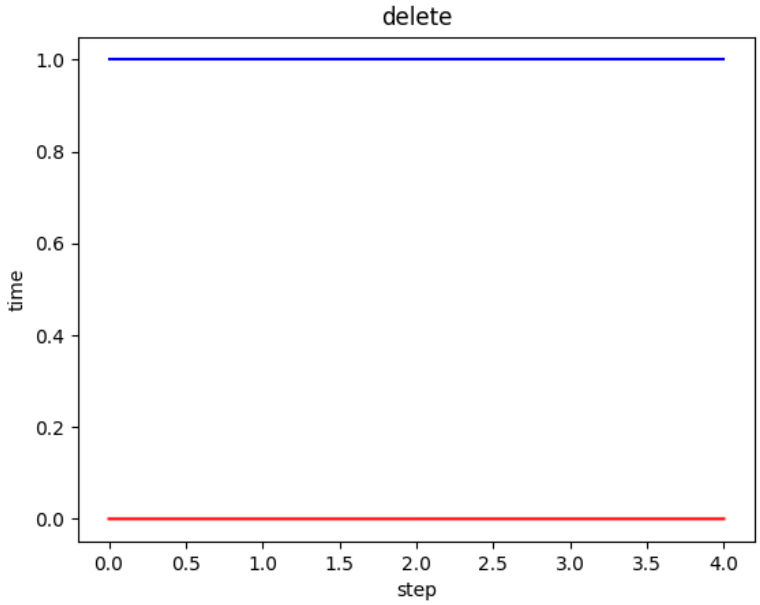


Рис. 7 — Затрата времени на операцию удаления в среднем и лучшем случае в сравнении с теоретической затратой времени (красным — время на входных данных, синим — теоретическое время)

Результат в худшем случае (Рис. 8), когда необходимо удалicounted x for better speedть узел из линенейного списка, расположенного по полученному номеру ячейки, не превосходит теоретический, сложность которого O(n).

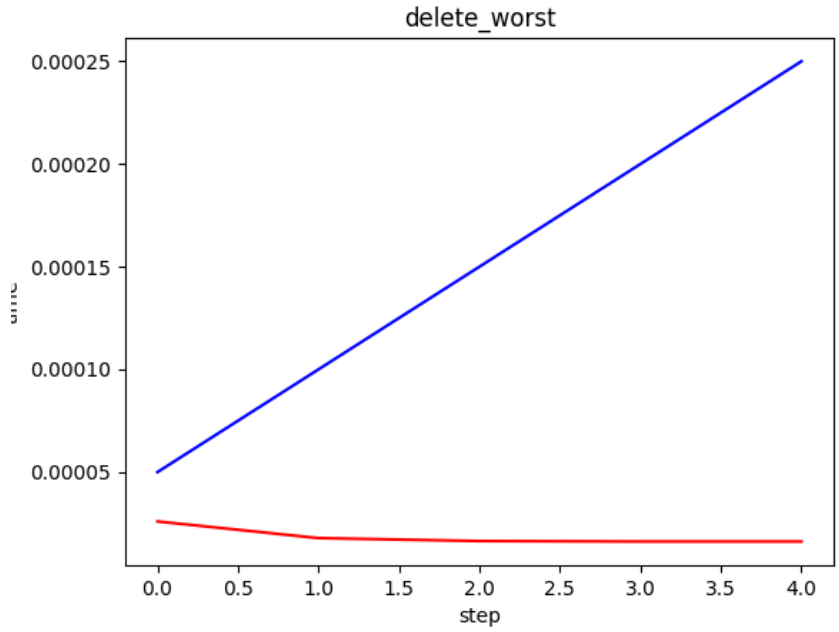


Рис. 8 — Затрата времени на операцию удаления в худшем случае в сравнении с теоретической затратой времени (красным — время на входных данных, синим — теоретическое время)

2. АНАЛИЗ ОПЕРАЦИЙ ХЕШ-ТАБЛИЦЫ (ОТКРЫТАЯ АДРЕСАЦИЯ)

2.1. Линейное пробирование

2.1.1. Вставка

Входные данные для анализа операций хеш-таблицы (открытая адрессация) для всех типов пробирования опередедены для таблицы с размером 5 в среднем и лучшем случае, с размером 9 в худшем случае. В худшем случае результат хеш-функции для всех ключей из входного набора данных будет одинаковым, что приведет к колизии. В качестве ключей принимаются только строки произвольной длины, т.к. используется полиномиальная хеш-функция, в качестве значений принимается любое изменяемое значение.

Входные данные представлены ниже:

```
inp_dict = {
    "123": "23",
    "14": "14",
    "1": "3",
    "556": "900",
    "1456": "12"
}
```

Рис. 9 — Входные данные в среднем и лучшем случае

```
inp_dict = {
    "'": "23",
    ":": "14",
    "M": "3",
    "`": "900",
    "s": "12",
}
```

Рис. 10 — Входные данные в худшем случае

В качестве интервала между пробами взята 1.

Результат в среднем и лучшем случае (Рис. 11) не превосходит теоретический, сложность которого O(1).

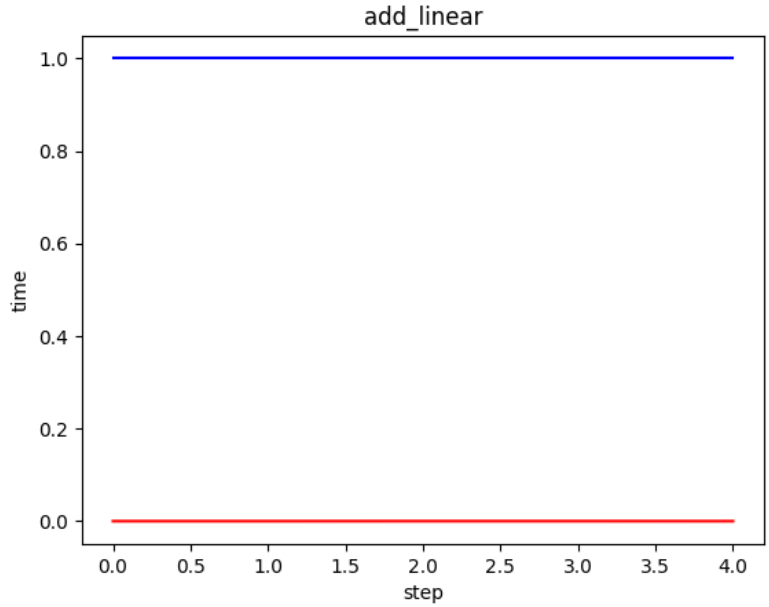


Рис. 11 — Затрата времени на операцию вставки при линейном пробировании в среднем и лучшем случае в сравнении с теоретической затратой времени (красным — время на входных данных, синим — теоретическое время)

Результат в худшем случае (Рис. 12) , когда необходимо провести исследование для поиска места для вставки ячейки в таблицу, не превосходит теоретический, сложность которого O(n).

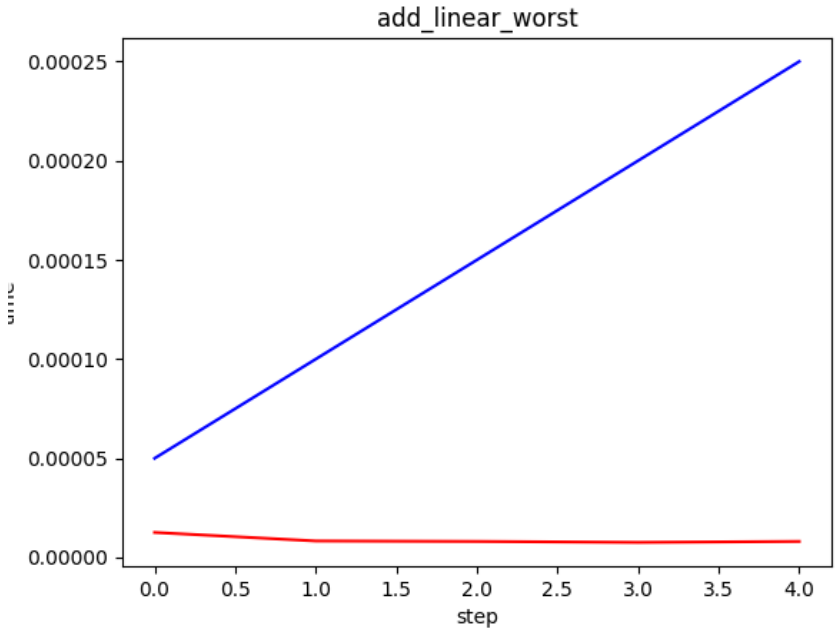


Рис. 12 — Затрата времени на операцию вставки при линейном пробировании в худшем случае в сравнении с теоретической затратой времени (красным — время на входных данных, синим — теоретическое время)

2.1.2. Поиск

Результат в среднем и лучшем случае (Рис. 13) не превосходит теоретический, сложность которого O(1).

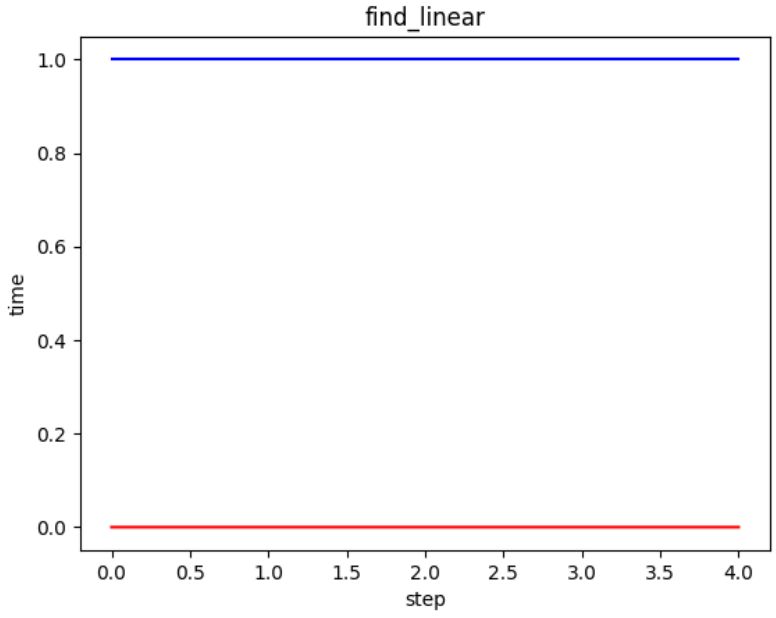


Рис. 13 — Затрата времени на операцию поиска при линейном пробировании в среднем и лучшем случае в сравнении с теоретической затратой времени (красным — время на входных данных, синим — теоретическое время)

Результат в худшем случае (Рис. 14), когда необходимо провести исследование для поиска ячейки в таблице с нужным ключом, не превосходит теоретический, сложность которого O(n).

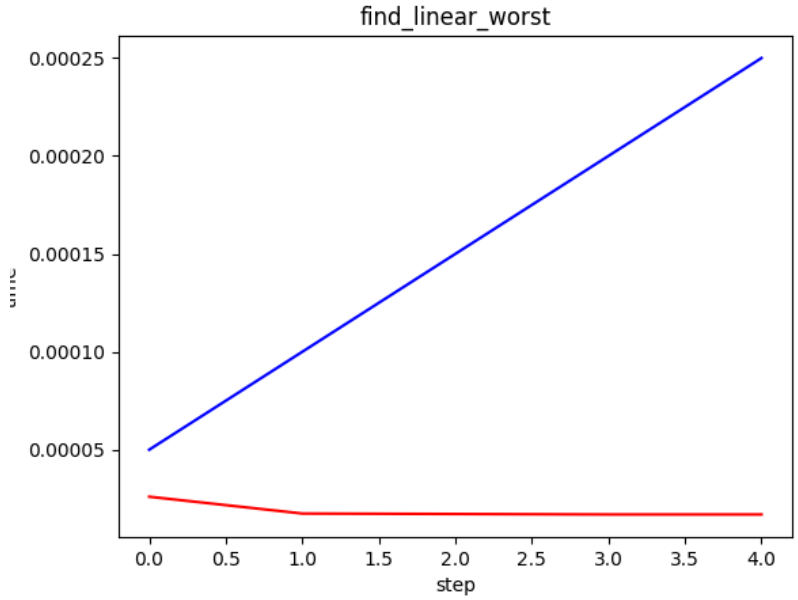


Рис. 14 — Затрата времени на операцию поиска при линейном пробировании в худшем случае в сравнении с теоретической затратой времени (красным — время на входных данных, синим — теоретическое время)

2.1.3. Удаление

Результат в среднем и худшем случае (Рис. 15) не превосходит теоретический, сложность которого O(1).

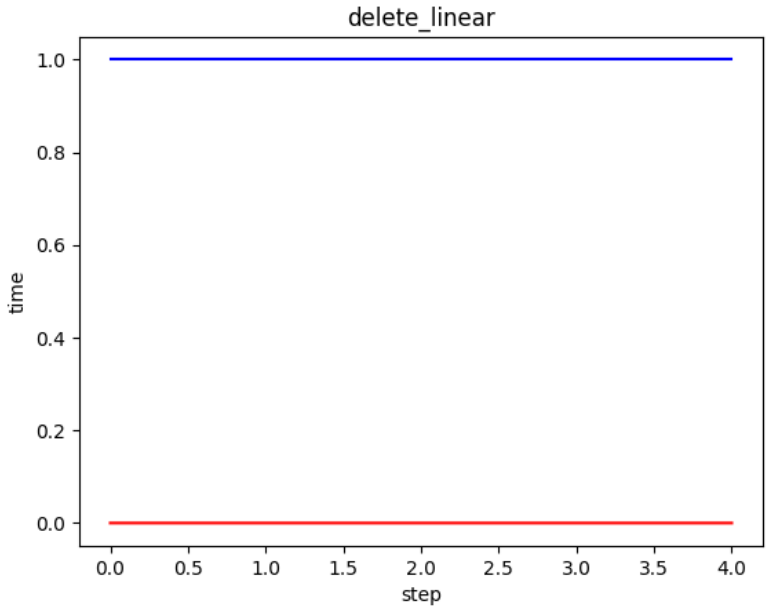


Рис. 15 — Затрата времени на операцию удаления при линейном пробировании в среднем и лучшем случае в сравнении с теоретической затратой времени (красным — время на входных данных, синим — теоретическое время)

Результат в худшем случае (Рис. 16), когда необходимо провести исследование для поиска ячейки в таблице с нужным ключом для удаления, не превосходит теоретический, сложность которого O(n).

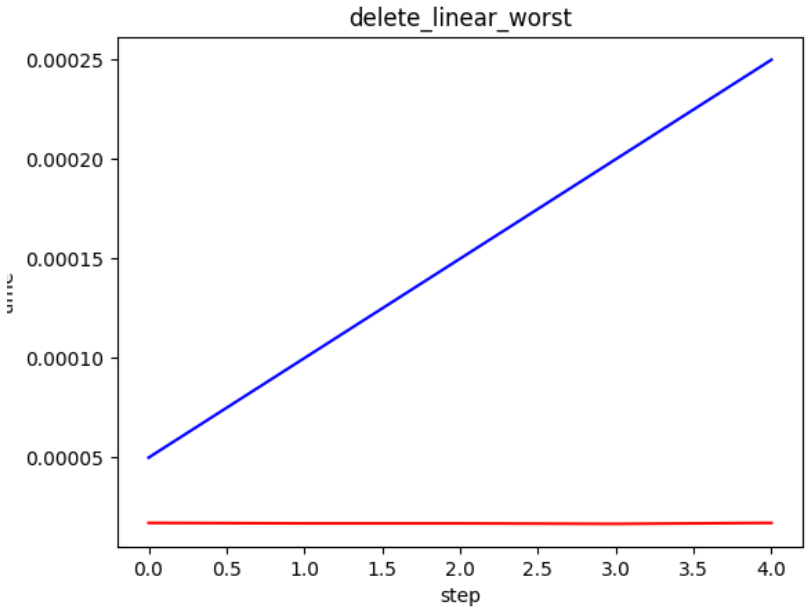


Рис. 16 — Затрата времени на операцию удаления при линейном пробировании в худшем случае в сравнении с теоретической затратой времени (красным — время на входных данных, синим — теоретическое время)

2.2. Квадратичное пробирование

2.2.1. Вставка

В качестве первой константы выбран 0, в качестве второй константы выбрана 1.

Результат в худшем случае (Рис. 17) , когда необходимо провести исследование для поиска места для вставки ячейки в таблицу, не превосходит теоретический, сложность которого O(n).

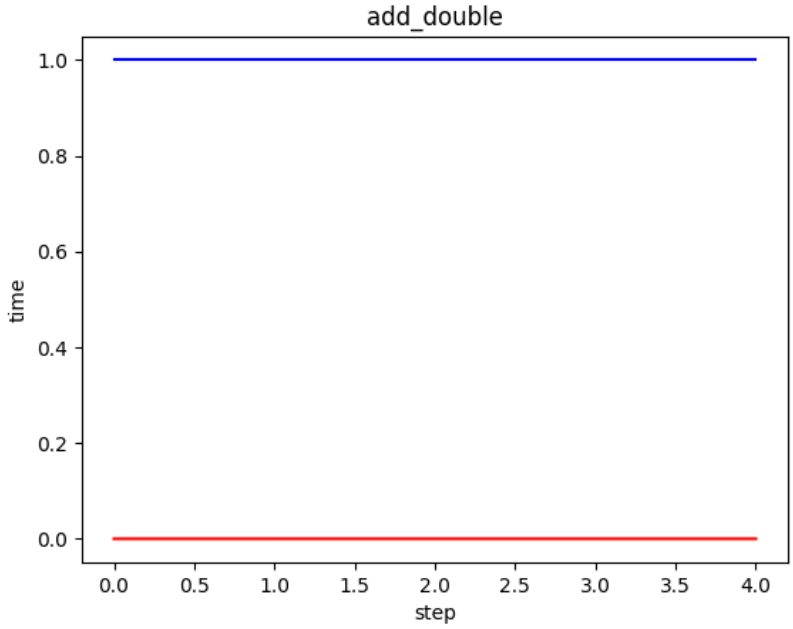


Рис. 17 — Затрата времени на операцию вставки при квадратичном пробировании в среднем и лучшем случае в сравнении с теоретической затратой времени (красным — время на входных данных, синим — теоретическое время)

Результат в худшем случае (Рис. 18) , когда необходимо провести исследование для поиска места для вставки ячейки в таблицу, не превосходит теоретический, сложность которого O(n).

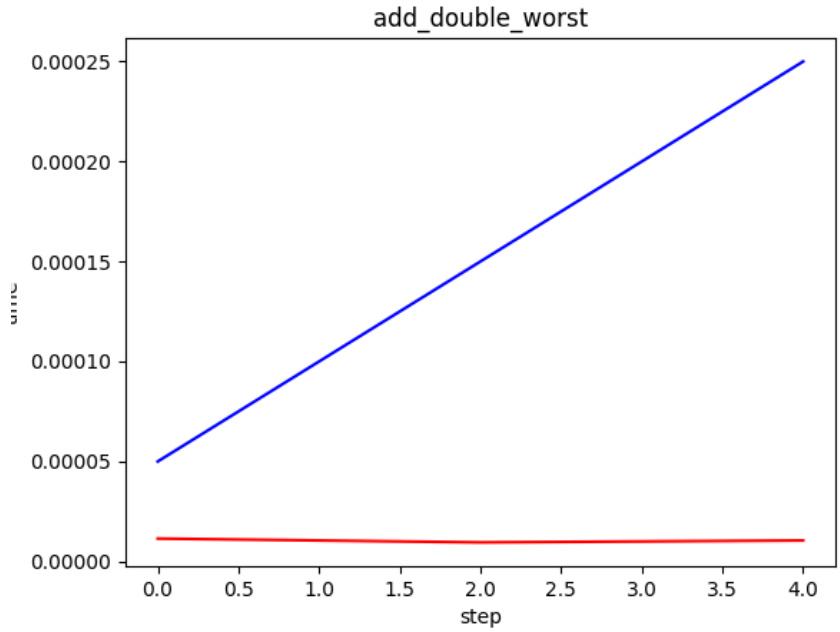


Рис. 18 — Затрата времени на операцию вставки при квадратичном пробировании в худшем случае в сравнении с теоретической затратой времени (красным — время на входных данных, синим — теоретическое время)

2.2.2. Поиск

Результат в среднем и лучшем случае (Рис. 19) не превосходит теоретический, сложность которого O(1).

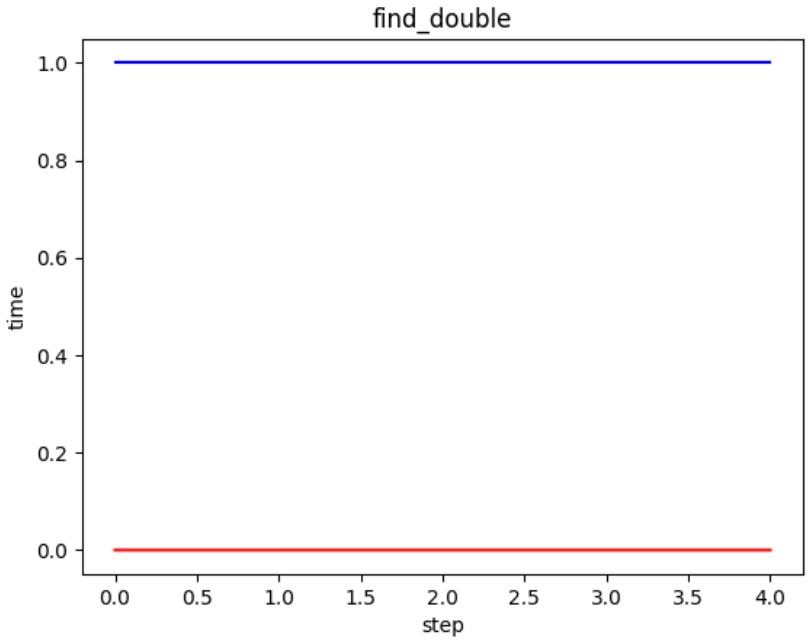


Рис. 19 — Затрата времени на операцию поиска при квадратичном пробировании в среднем и лучшем случае в сравнении с теоретической затратой времени (красным — время на входных данных, синим — теоретическое время)

Результат в худшем случае (Рис. 20), когда необходимо провести исследование для поиска ячейки в таблице с нужным ключом, не превосходит теоретический, сложность которого O(n).

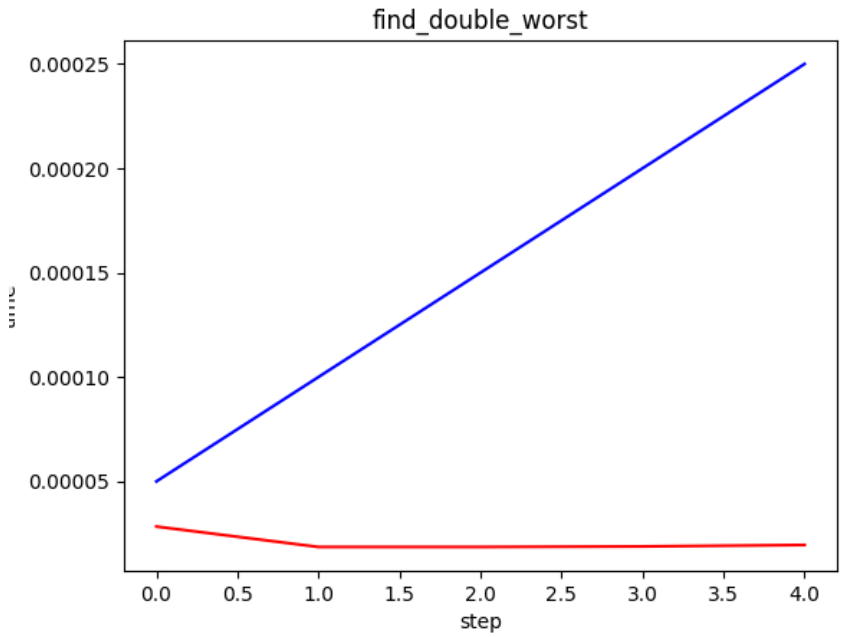


Рис. 20 — Затрата времени на операцию поиска при квадратичном пробировании в худшем случае в сравнении с теоретической затратой времени (красным — время на входных данных, синим — теоретическое время)

2.2.3. Удаление

Результат в среднем и худшем случае (Рис. 21) не превосходит теоретический, сложность которого O(1).

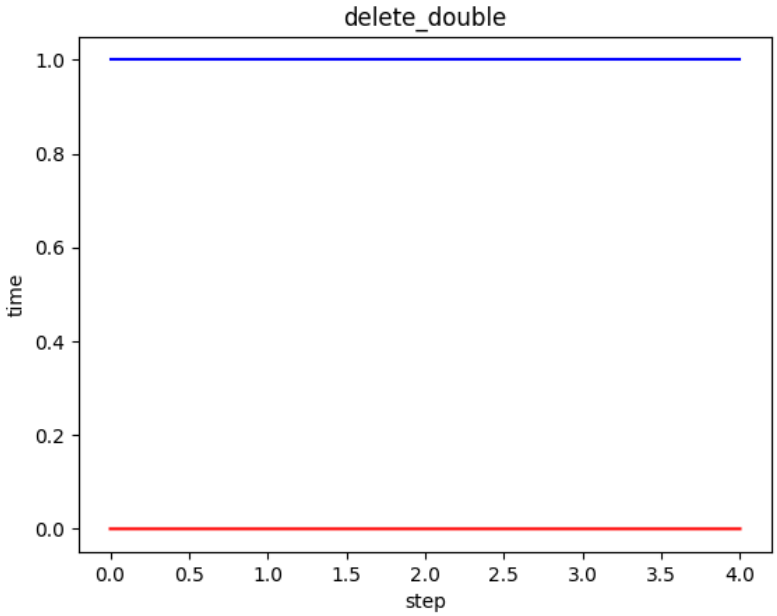


Рис. 21 — Затрата времени на операцию удаления при квадратичном пробировании в среднем и лучшем случае в сравнении с теоретической затратой времени (красным — время на входных данных, синим — теоретическое время)

Результат в худшем случае (Рис. 22), когда необходимо провести исследование для поиска ячейки в таблице с нужным ключом для удаления, не превосходит теоретический, сложность которого O(n).

Рис. 22 — Затрата времени на операцию удаления при линейном пробировании в худшем случае в сравнении с теоретической затратой времени (красным — время на входных данных, синим — теоретическое время)

2.3. Пробироваение методом двойного хеширования

2.3.1. Вставка

В качестве второй хеш функции выбрана модифицированная полиномиальная хеш-функция, в которой каждый x делится по модулую на некоторую большую по значению (более 10^7) простую константу p, итоговая сумма также делится по модулю на p.

Результат в среднем и лучшем случае (Рис. 23) не превосходит теоретический, сложность которого O(1).

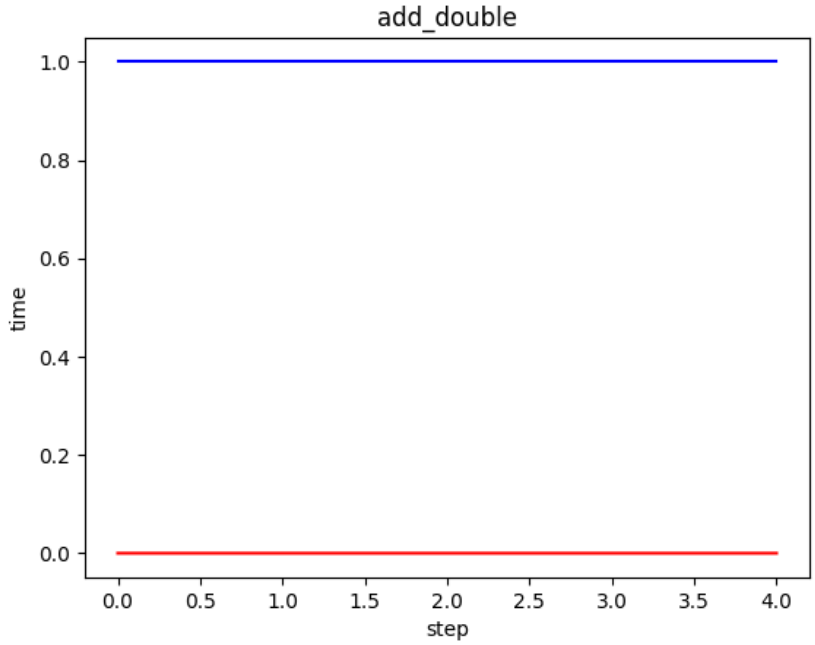


Рис. 23 — Затрата времени на операцию вставки при пробировании методом двойного хеширования в среднем и лучшем случае в сравнении с теоретической затратой времени (красным — время на входных данных, синим — теоретическое время)

Результат в худшем случае (Рис. 24) , когда необходимо провести исследование для поиска места для вставки ячейки в таблицу, не превосходит теоретический, сложность которого O(n).

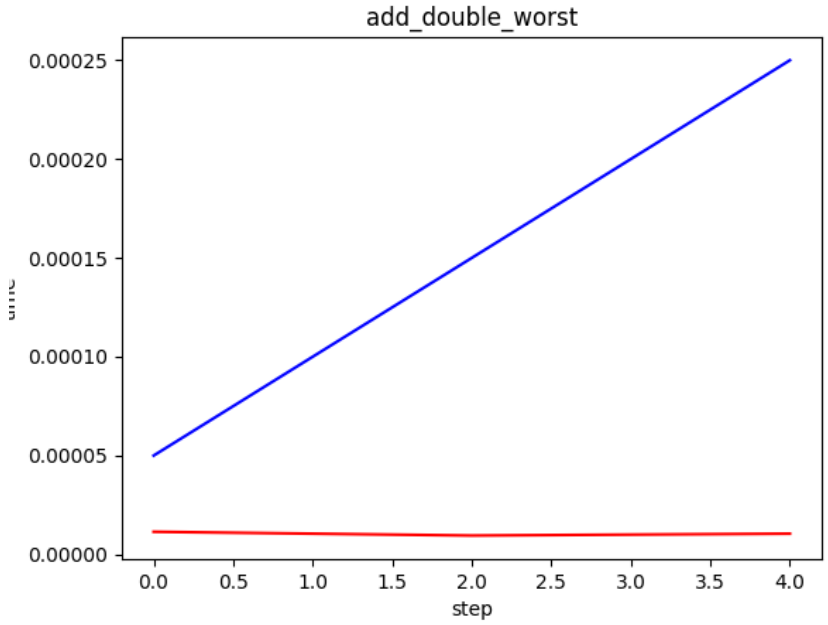


Рис. 24 — Затрата времени на операцию вставки при пробировании методом двойного хеширования в худшем случае в сравнении с теоретической затратой времени (красным — время на входных данных, синим — теоретическое время)

2.3.2. Поиск

Результат в среднем и лучшем случае (Рис. 25) не превосходит теоретический, сложность которого O(1).

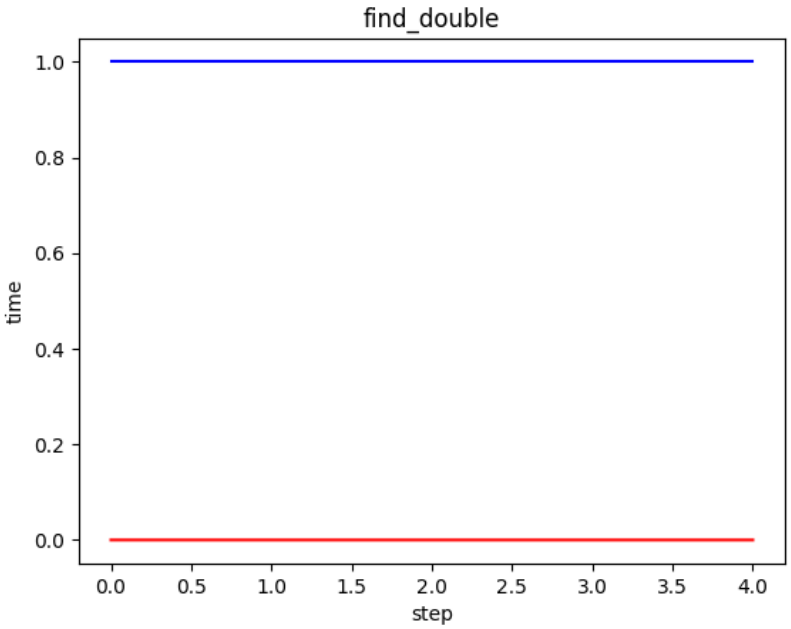


Рис. 25 — Затрата времени на операцию поиска при пробировании методом двойного хеширования в среднем и лучшем случае в сравнении с теоретической затратой времени (красным — время на входных данных, синим — теоретическое время)

Результат в худшем случае (Рис. 26), когда необходимо провести исследование для поиска ячейки в таблице с нужным ключом, не превосходит теоретический, сложность которого O(n).

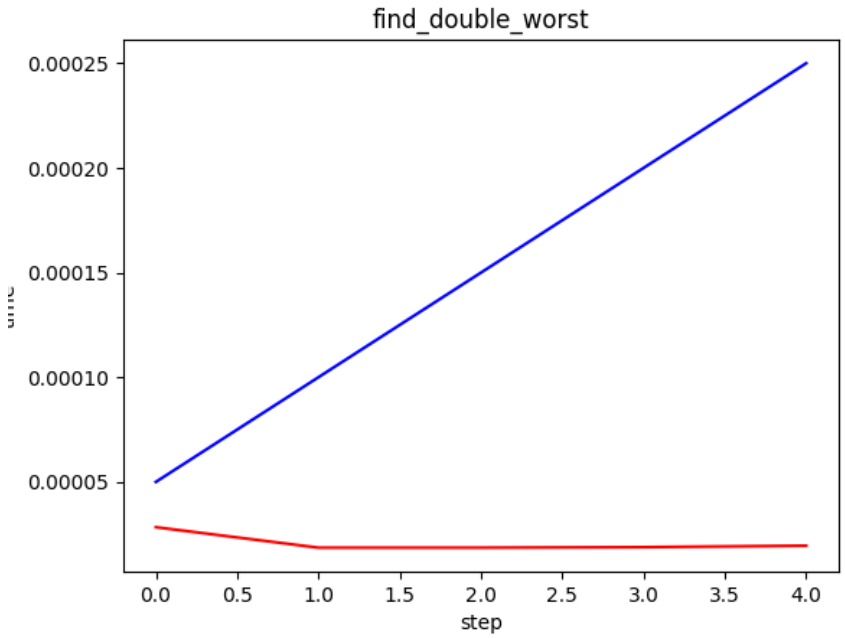


Рис. 26 — Затрата времени на операцию поиска при пробировании методом двойного хеширования в худшем случае в сравнении с теоретической затратой времени (красным — время на входных данных, синим — теоретическое время)

2.3.3. Удаление

Результат в среднем и худшем случае (Рис. 27) не превосходит теоретический, сложность которого O(1).

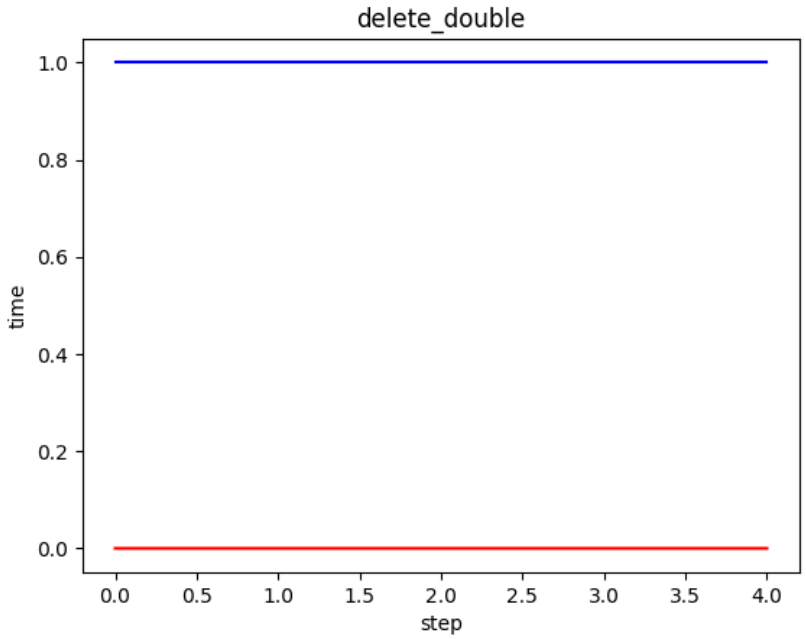


Рис. 27 — Затрата времени на операцию удаления при пробировании методом двойного хеширования в среднем и лучшем случае в сравнении с теоретической затратой времени (красным — время на входных данных, синим — теоретическое время)

Результат в худшем случае (Рис. 28), когда необходимо провести исследование для поиска ячейки в таблице с нужным ключом для удаления, не превосходит теоретический, сложность которого O(n).

Рис. 28 — Затрата времени на операцию удаления при пробировании методом двойного хеширования в худшем случае в сравнении с теоретической затратой времени (красным — время на входных данных, синим — теоретическое время)

3. СРАВНЕНИЕ ОПЕРАЦИЙ ХЕШ-ТАБЛИЦЫ (МЕТОД ЦЕПОЧЕК) И ХЕШ-ТАБЛИЦЫ (ОТКРЫТАЯ АДРЕСАЦИЯ)

3.1. Вставка

Для сравнения в хеш-таблице (открытая адресация) выбрано линейное пробирование.

Результат не отличается для двух структур данных как в среднем и лучшем случае, так и в худшем случае (Рис. 29).

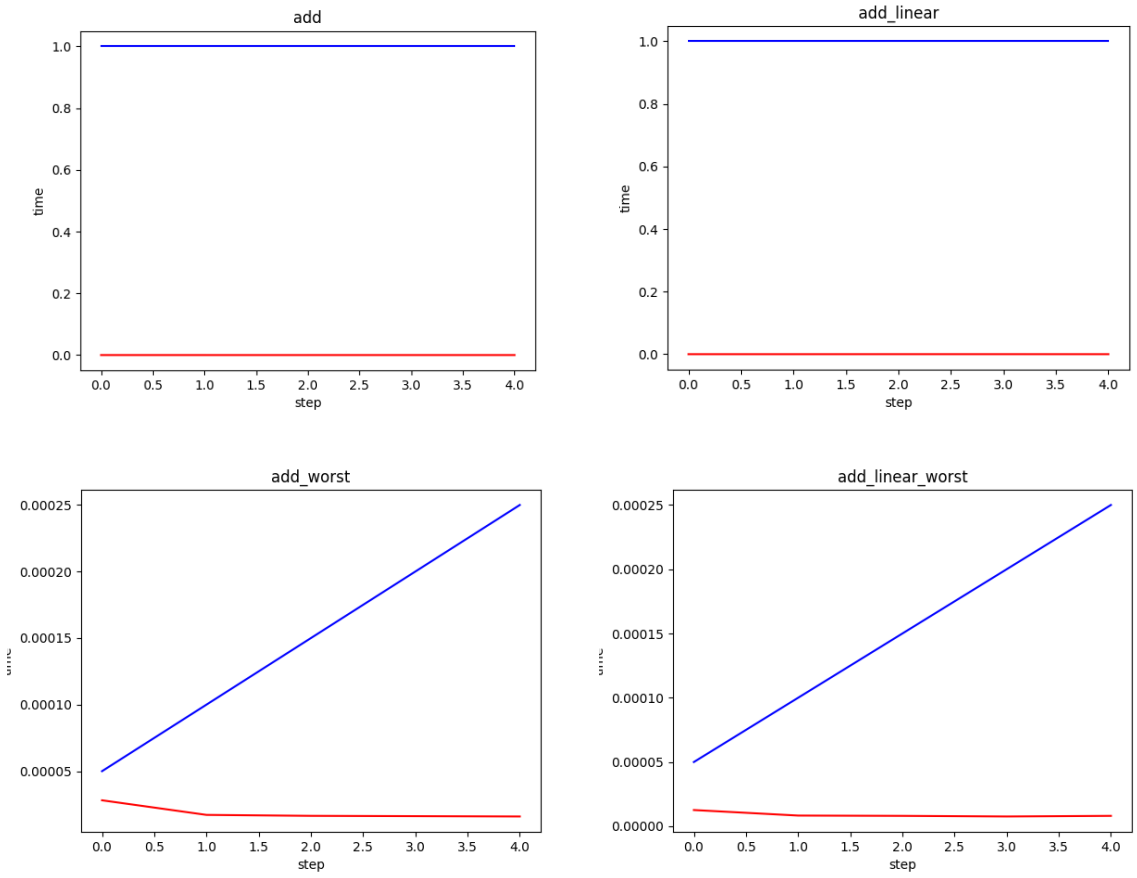


Рис. 29 — Сравнение вставки для хеш-таблицы (метод цепочек) (слева) и для хеш-таблицы (открытая адресация) (справа)

3.2. Поиск

Результат не отличается для двух структур данных как в среднем и лучшем случае, так и в худшем случае (Рис. 30).

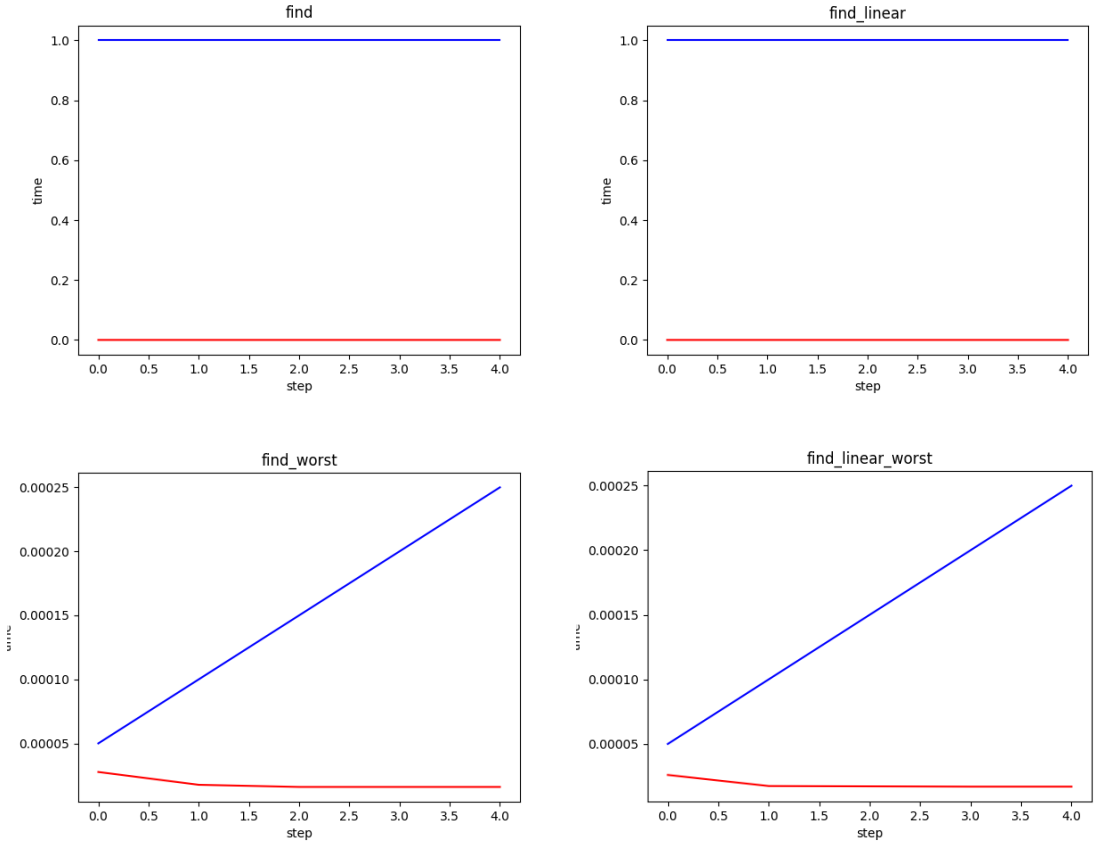
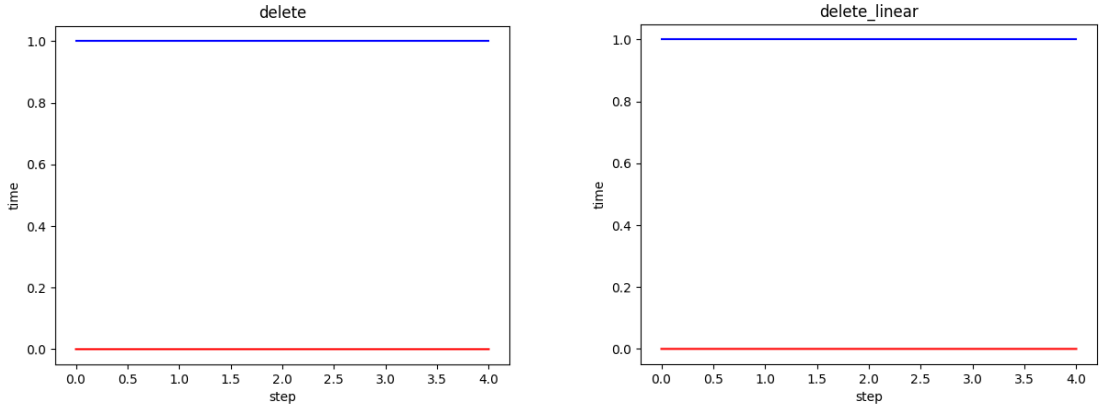


Рис. 30 — Сравнение поиска для хеш-таблицы (метод цепочек) (слева) и для хеш-таблицы (открытая адресация) (справа)

3.3. Удаление

Результат не отличается для двух структур данных как в среднем и лучшем случае, так и в худшем случае (Рис. 31).



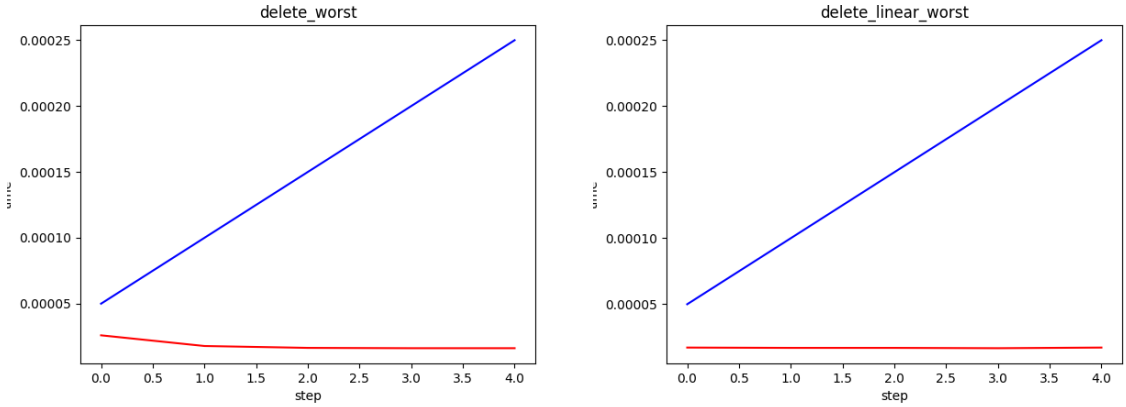


Рис. 31 — Сравнение удаления для хеш-таблицы (метод цепочек) (слева) и для хеш-таблицы (открытая адресация) (справа)

ЗАКЛЮЧЕНИЕ

Проведено исследование операций вставки, поиска и удаления для каждой из структур данных, а также сравнение операций между ними. В ходе сравнения операций для каждой из структур определено, что временная сложность каждой операции не превышает теоретическую сложность как в среднем и лучшем случае, так и в худшем случае. В ходе сравнения операций двух структур данных определено, что временная сложность операций не отличается в каждом из случаев, что говорит об абстрактности хеш-таблицы, т.е. независимости временной сложности операций от способа реализации хеш-таблицы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Hash table // Wikipedia. URL: https://en.wikipedia.org/wiki/Hash_table (дата обращения: 09.12.2022)

2. Хеш-таблица // Wikipedia. URL: https://ru.wikipedia.org/wiki/%D0%A5%D0%B5%D1%88-%D1%82%D0%B0%D0%B1%D0%BB%D0%B8%D1%86%D0%B0 (дата обращения: 09.12.2022)

3. Хеш-таблица // Вики-конспекты 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 (дата обращения: 09.12.2022)

4. Разрешение коллизий // Вики-конспекты ИТМО. URL: https://neerc.ifmo.ru/wiki/index.php?title=%D0%A0%D0%B0%D0%B7%D1%80%D0%B5%D1%88%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BA%D0%BE%D0%BB%D0%BB%D0%B8%D0%B7%D0%B8%D0%B9 (дата обращения: 09.12.2022)

5. Хеширование // Problem Solving with Algorithms and Data Structures. URL: http://aliev.me/runestone/SortSearch/Hashing.html (дата обращения: 09.12.2022)

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл chaining_hash_table.py:

```
import numpy as np
import matplotlib.pyplot as plt
import time
import random
import string


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


class ChainingHashTable:
    def __init__(self):
        self.capacity = 19
        self.size = 0
        self.hash_table = [None for _ in range(self.capacity)]
        self.x = 343
        self.counted_x = [self.x ** k for k in range(8)]
        self.border = 0.5
        self.expansion = 0.75

    def __hash_func(self, key):
        # classical polynomial hash function
        # x = 257
        # str_sum = sum([ord(element) * x ** k for k, element in enumerate(key)])

        # improved polynomial hash function
        # with using counted x
        # which increases speed by 64%
        if len(key) > len(self.counted_x):
            k = len(self.counted_x)
            degrees_to_add = len(key) - len(self.counted_x)
            for i in range(degrees_to_add):
                self.counted_x.append(self.x ** (k + i))

        str_sum = sum([ord(element) * self.counted_x[i] for i, element in enumerate(key)])
        return str_sum % self.capacity

    def add(self, key, value):
        h = self.__hash_func(key)
        cell = self.hash_table[h]
        node = Node(key, value)

        if not cell:
```

```
            self.hash_table[h] = node
            self.size += 1
        else:
            if self.hash_table[h].key == key:
                self.hash_table[h].value = value
            else:
                node.next = cell
                self.hash_table[h] = node
                self.size += 1

        if self.size >= self.capacity * self.border:
            self.__expand()

    def find(self, key):
        result = "Not found"
        h = self.__hash_func(key)
        cell = self.hash_table[h]

        if not cell:
            print(result)
            return result

        while cell:
            if cell.key == key:
                result = cell.value
                break
            cell = cell.next

        print(result)
        return result

    def delete(self, key):
        h = self.__hash_func(key)
        cell = self.hash_table[h]
        prev_cell = None

        while cell is not None and cell.key != key:
            prev_cell = cell
            cell = cell.next

        if cell is None:
            print("Can't delete element: key not found")
            return None
        else:
            self.size -= 1
            result = cell.value

            if prev_cell is None:
                self.hash_table[h] = None
            else:
                prev_cell.next = prev_cell.next.next
```

```
            print(result)
            return result

    def __expand(self):
        self.size = 0
        old_hash_table = self.hash_table
        old_capacity = self.capacity
        self.capacity = self.capacity + int(self.capacity * self.expansion)

        self.hash_table = [None for _ in range(self.capacity)]

        for i in range(old_capacity):
            cur_cell = old_hash_table[i]
            while cur_cell:
                # print(f"Computing hash for {cur_cell.key}: {cur_cell.value}")
                self.add(cur_cell.key, cur_cell.value)
                cur_cell = cur_cell.next


def main():
    cht = ChainingHashTable()

    # need generate data for best/mean/worst case
    # inp_dict = {
    #     """: "23",
    #     ":": "14",
    #     "M": "3",
    #     "`": "900",
    #     "s": "12",
    # }
    #
    # standard_x = np.array([5e-5, 10e-5, 15e-5, 20e-5, 25e-5])
    # x = []

    # stress test
    inp_dict = {}
    letters = string.ascii_lowercase
    keys = [''.join(random.choice(letters) for _ in range(12)) for _ in range(100000)]
    values = [i+1 for i in range(100000)]

    for i in range(len(values)):
        inp_dict[keys[i]] = values[i]

    std_x = [1 for i in range(100000)]
    standard_x = np.array(std_x)
    x = []

    for key, value in inp_dict.items():
        start_time = time.time()
        cht.add(key, value)
        x.append(time.time() - start_time)
```

```
    for key in inp_dict.keys():
        # start_time = time.time()
        cht.find(key)
        # x.append(time.time() - start_time)

    for key in inp_dict.keys():
        # start_time = time.time()
        cht.delete(key)
        # x.append(time.time() - start_time)

    x = np.array(x)
    print(x)
    plt.plot(x, 'r', standard_x, 'b')
    plt.ylabel("time")
    plt.xlabel("step")
    plt.title("add")
    plt.show()

    # while True:
    #     inp_string = input().split()
    #     command = inp_string[0]
    #     number = inp_string[1]
    #     if command == 'add':
    #         try:
    #             name = inp_string[2]
    #             cht.add(number, name)
    #         except IndexError:
    #             print("You need to type value, try again!")
    #     else:
    #         if command == 'find':
    #             cht.find(number)
    #         elif command == 'del':
    #             cht.delete(number)
    #         else:
    #             print("Bad command!\nType this commands:\nadd, find, del")


main()
```

Файл open_addressing_hash_table.py:

```
import numpy as np
import matplotlib.pyplot as plt
import time
import random
import string


class Node:
    def __init__(self, key, value):
        self.key = key
        self.value = value
```

```
class OpenAddressingHashTable:
    def __init__(self):
        self.capacity = 19
        self.size = 0
        self.hash_table = [None for _ in range(self.capacity)]
        self.x_1 = 343
        self.counted_x_1 = [self.x_1 ** k for k in range(8)]
        self.x_2 = 351
        self.p = 100_000_003
        self.counted_x_2 = [self.x_2 ** k % self.p for k in range(8)]
        self.border = 0.5
        self.expansion = 0.75

    def __hash_func(self, key):
        # classical polynomial hash function
        # x = 257
        # str_sum = sum([ord(element) * x ** k for k, element in enumerate(key)])

        # improved polynomial hash function
        # with using counted x
        # which increases speed by 64%
        if len(key) > len(self.counted_x_1):
            k = len(self.counted_x_1)
            degrees_to_add = len(key) - len(self.counted_x_1)
            for i in range(degrees_to_add):
                self.counted_x_1.append(self.x_1 ** (k + i))

        str_sum = sum([ord(element) * self.counted_x_1[i] for i, element in enumerate(key)])
        return str_sum % self.capacity

    def __second_hash_func(self, key):
        # for classical
        # x = 253
        # p = 100_000_007
        # str_sum = sum([ord(element) * x ** k % p for k, element in enumerate(key)])

        # for improved
        if len(key) > len(self.counted_x_2):
            k = len(self.counted_x_2)
            degrees_to_add = len(key) - len(self.counted_x_2)
            for i in range(degrees_to_add):
                self.counted_x_2.append(self.x_2 ** (k + i) % self.p)

        str_sum = sum([ord(element) * self.counted_x_2[i] for i, element in enumerate(key)])
        return str_sum % self.p % (self.capacity - 1) + 1

    def __rehash_func(self, h, i, key):

        # linear probing
        # k = 1
        # rh = (h + i*k) % self.capacity
```

```
        # quadratic probing
        # c_1 = 0
        # c_2 = 1
        # rh = (h + c_1*i + c_2*(i**2)) % self.capacity

        # double hashing probing
        # need key in input parameters
        rh = (h + i * self.__second_hash_func(key)) % self.capacity
        print(h, i, self.__second_hash_func(key), rh, self.capacity)

        return rh

    def add(self, key, value):
        h = self.__hash_func(key)
        probe_number = 1
        cell = Node(key, value)

        if self.hash_table[h] is None:
            self.hash_table[h] = cell
            self.size += 1
        else:
            if self.hash_table[h].key == key:
                self.hash_table[h].value = value
            else:
                next_h = self.__rehash_func(h, probe_number, key)
                while self.hash_table[next_h] and self.hash_table[next_h].key != key:
                    probe_number += 1
                    next_h = self.__rehash_func(next_h, probe_number, key)

                if self.hash_table[next_h] is None:
                    self.hash_table[next_h] = cell
                    self.size += 1
                else:
                    self.hash_table[next_h].value = value

        if self.size >= self.capacity * self.border:
            self.__expand()

    def find(self, key):
        start_h = self.__hash_func(key)
        probe_number = 1
        result = "Not found"
        cur_h = start_h

        if not self.hash_table[cur_h]:
            print(result)
            return result

        while self.hash_table[cur_h]:
            if self.hash_table[cur_h].key == key:
                result = self.hash_table[cur_h].value
```

```
                break
            cur_h = self.__rehash_func(cur_h, probe_number, key)
            probe_number += 1
            if cur_h == start_h:
                break

        print(result)
        return result

    def delete(self, key):
        start_h = self.__hash_func(key)
        probe_number = 1
        result = "Can't delete element: key not found"
        cur_h = start_h

        while self.hash_table[cur_h] is None:
            cur_h = self.__rehash_func(cur_h, probe_number, key)
            probe_number += 1
            if cur_h == start_h:
                print(result)
                return result

        if self.hash_table[cur_h].key == key:
            self.size -= 1
            result = self.hash_table[cur_h].value
            self.hash_table[cur_h] = None

        print(result)
        return result

    def __expand(self):
        self.size = 0
        old_hash_table = self.hash_table
        old_capacity = self.capacity
        self.capacity = self.capacity + int(self.capacity * self.expansion)

        self.hash_table = [None for _ in range(self.capacity)]

        for i in range(old_capacity):
            cur_cell = old_hash_table[i]
            if cur_cell:
                # print(f"Computing hash for {cur_cell.key}: {cur_cell.value}")
                self.add(cur_cell.key, cur_cell.value)


def main():
    odht = OpenAddressingHashTable()

    # need generate data for best/mean/worst case
    # inp_dict = {
    #     "": "23",
    #     ":": "14",
```

```
#     "M": "3",
#     "`": "900",
#     "s": "12",
# }
#
# standard_x = np.array([5e-5, 10e-5, 15e-5, 20e-5, 25e-5])
# x = []

# stress test
inp_dict = {}
letters = string.ascii_lowercase
keys = [''.join(random.choice(letters) for _ in range(12)) for _ in range(100000)]
values = [i + 1 for i in range(100000)]

for i in range(len(values)):
    inp_dict[keys[i]] = values[i]

std_x = [1 for i in range(1000000)]
standard_x = np.array(std_x)
x = []

for key, value in inp_dict.items():
    start_time = time.time()
    odht.add(key, value)
    x.append(time.time() - start_time)

for key in inp_dict.keys():
    # start_time = time.time()
    odht.find(key)
    # x.append(time.time() - start_time)

for key in inp_dict.keys():
    # start_time = time.time()
    odht.delete(key)
    # x.append(time.time() - start_time)

x = np.array(x)
print(x)
plt.plot(x, 'r', standard_x, 'b')
plt.ylabel("time")
plt.xlabel("step")
plt.title("add_quadratic")
plt.show()

# while True:
#     inp_string = input().split()
#     command = inp_string[0]
#     number = inp_string[1]
#     if command == 'add':
#         if command == 'add':
#             try:
#                 name = inp_string[2]
```

```
#                odht.add(number, name)
#            except IndexError:
#                print("You need to type value, try again!")
#    else:
#        if command == 'find':
#            odht.find(number)
#        elif command == 'del':
#            odht.delete(number)
#        else:
#            print("Bad command!\nType this commands:\nadd, find, del")
```