Министерство образования и науки Украины
Национальный технический университет Украины "Киевский политехнический институт имени Игоря Сикорского"
Факультет информатики и вычислительной техники

Кафедра автоматизированных систем обработки информации и управления

ОТЧЕТ

по лабораторной работе № 2 по дисциплине
«Проектирование и анализ вычислительных алгоритмов»

„ **Проектирование и анализ алгоритмов поиска** ”

Выполнил *ІП-61 Кушка М.О.* _____
(шифр, фамилия, имя, отчество)

Проверил *Головченко М.Н.* _____
(фамилия, имя, отчество)

Киев 2018

СОДЕРЖАНИЕ

1 ЦЕЛЬ ЛАБОРАТОРНОЙ РАБОТЫ

Цель работы – изучить основные подходы к анализу вычислительной сложности алгоритмов поиска и оценить их эффективность на различных структурах данных.

2 ЗАДАНИЕ

Согласно варианту (таблица 2.1), записать алгоритм поиска при помощи псевдокода (или другого способа по выбору).

Провести анализ временной сложности в худшем, лучшем и среднем случае и записать временную сложность в асимптотических оценках.

Выполнить программную реализацию алгоритма на любом языке программирования для поиска индекса элемента по заданному ключу в массиве и двусвязном списке с фиксацией временных оценочных характеристик (количество сравнений) и времени поиска.

Провести ряд испытаний алгоритма на структурах разной размерности (100, 1000, 5000, 10000, 20000 элементов) и построить графики зависимости временных оценочных характеристик и времени поиска от размерности структуры (2 графика).

Сделать обобщенный вывод по лабораторной работе.

Таблица 2.1 – Варианты алгоритмов

№	**Алгоритм сортировки**
1	Однородный бинарный поиск
2	Метод Шарра
3	Фибоначчиев поиск
4	Интерполяционный поиск
5	Метод Хеш-функции(«Хеш-функции», основанные на делении), разрешения коллизий методом цепочек
6	Метод Хеш-функции(«Хеш-функции», основанные на делении), разрешения коллизий методом открытой адресация с линейным пробированием
7	Метод Хеш-функции(«Хеш-функции», основанные на делении), разрешения коллизий методом открытой адресация с квадратичным пробированием

8	Метод Хеш-функции(«Хеш-функции», основанные на делении), разрешения коллизий методом открытой адресация с двойным хешированием
9	Метод Хеш-функции(«Хеш-функции», основанные на умножении), разрешения коллизий методом цепочек
10	Метод Хеш-функции(«Хеш-функции», основанные на умножении), разрешения коллизий методом открытой адресация с линейным пробированием
11	Метод Хеш-функции(«Хеш-функции», основанные на умножении), разрешения коллизий методом открытой адресация с квадратичным пробированием
12	Метод Хеш-функции(«Хеш-функции», основанные на умножении), разрешения коллизий методом открытой адресация с двойным хешированием
13	Однородный бинарный поиск
14	Метод Шарра
15	Фибоначчиев поиск
16	Интерполяционный поиск
17	Метод Хеш-функции(«Хеш-функции», основанные на делении), разрешения коллизий методом цепочек
18	Метод Хеш-функции(«Хеш-функции», основанные на делении), разрешения коллизий методом открытой адресация с линейным пробированием
19	Метод Хеш-функции(«Хеш-функции», основанные на делении), разрешения коллизий методом открытой адресация с квадратичным пробированием
20	Метод Хеш-функции(«Хеш-функции», основанные на делении), разрешения коллизий методом открытой адресация с двойным хешированием

21	Метод Хеш-функции(«Хеш-функции», основанные на умножении), разрешения коллизий методом цепочек
22	Метод Хеш-функции(«Хеш-функции», основанные на умножении), разрешения коллизий методом открытой адресация с линейным пробированием
23	Метод Хеш-функции(«Хеш-функции», основанные на умножении), разрешения коллизий методом открытой адресация с квадратичным пробированием
24	Метод Хеш-функции(«Хеш-функции», основанные на умножении), разрешения коллизий методом открытой адресация с двойным хешированием
25	Однородный бинарный поиск
26	Метод Шарра
27	Фибоначчиев поиск
28	Интерполяционный поиск
29	Метод Хеш-функции(«Хеш-функции», основанные на делении), разрешения коллизий методом цепочек
30	Метод Хеш-функции(«Хеш-функции», основанные на делении), разрешения коллизий методом открытой адресация с линейным пробированием

3 ВЫПОЛНЕНИЕ

3.1 Псевдокод алгоритма

```
class HashTable is
    function init(size) is
        size = size
        hashTable = doublyLinkedList[size]
        currentSize = 0

        A = (5**(1/2) - 1) / 2
        M = 2**64

    function getKey(value) is
        return floor(M * (value * A % 1))

    function getKey2(value) is
        return value % size

    function add(value):
        if (currentSize >= size) then
            print("Out of free space in the hash table.")
            return

        i = 0
        x = getKey(value)
        y = getKey2(value)

        while True do
            key = (x + i*y) % size

            if (key >= 0 and key < size):
                if (hashTable[key] == NULL):
                    hashTable[key] = value
                    currentSize += 1
                    break
                else:
                    i += 1

    function search(key) is
        if (key >= 0 and key < size):
            return hashTable[key]

        return NULL
```

3.2 Анализ временной сложности поиска / вставки / удаления

Худшее время	$O(n)$
Лучшее время	$O(1)$
Среднее время	$O(1)$

Теорема 11.1

В хеш-таблице с разрешением коллизий методом цепочек время неудачного поиска в среднем случае в предположении простого равномерного хеширования составляет $\Theta(1+\alpha)$.

Доказательство. В предположении простого равномерного хеширования любой ключ k, который еще не находится в таблице, может быть помещен с равной вероятностью в любую из m ячеек. Математическое ожидание времени неудачного поиска ключа k равно времени поиска до конца списка $T[h(k)]$, ожидаемая длина которого — $\mathrm{E}\left[n_{h(k)}\right] = \alpha$. Таким образом, при неудачном поиске математическое ожидание количества проверяемых элементов равно α, а общее время, необходимое для поиска, включая время вычисления хеш-функции $h(k)$, равно $\Theta(1+\alpha)$. ■

Успешный поиск несколько отличается от неудачного, поскольку вероятность поиска в списке различна для разных списков и пропорциональна количеству содержащихся в нем элементов. Тем не менее и в этом случае математическое ожидание времени поиска остается равным $\Theta(1+\alpha)$.

Теорема 11.2

В хеш-таблице с разрешением коллизий методом цепочек время успешного поиска в среднем случае в предположении простого равномерного хеширования в среднем равно $\Theta(1+\alpha)$.

Доказательство. Мы полагаем, что искомый элемент с равной вероятностью может быть любым элементом, хранящимся в таблице. Количество элементов, проверяемых в процессе успешного поиска элемента x, на 1 больше, чем количество элементов, находящихся в списке перед x. Элементы, находящиеся в списке

до x, были вставлены в список после того, как элемент x был сохранен в таблице, так как новые элементы помещаются в начало списка. Для того чтобы найти математическое ожидание количества проверяемых элементов, мы возьмем среднее по всем n элементам x в таблице значение, которое равно 1 плюс математическое ожидание количества элементов, добавленных в список x после самого́ искомого элемента. Пусть x_i обозначает i-й элемент, вставленный в таблицу ($i = 1, 2, \ldots, n$), и пусть $k_i = x_i.key$. Определим для ключей k_i и k_j индикаторную случайную величину $X_{ij} = \mathrm{I}\{h(k_i) = h(k_j)\}$. В предположении простого равномерного хеширования $\Pr\{h(k_i) = h(k_j)\} = 1/m$ и, в соответствии с леммой 5.1, $\mathrm{E}[X_{ij}] = 1/m$. Таким образом, математическое ожидание количества проверяемых элементов в случае успешного поиска равно

$$
\begin{aligned}
&\mathrm{E}\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right] \\
&= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\mathrm{E}[X_{ij}]\right) \quad \text{(из линейности математического ожидания)} \\
&= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right) \\
&= 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i) \\
&= 1+\frac{1}{nm}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right) \\
&= 1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right) \quad \text{(согласно (A.1))} \\
&= 1+\frac{n-1}{2m} \\
&= 1+\frac{\alpha}{2}-\frac{\alpha}{2n}\,.
\end{aligned}
$$

Таким образом, полное время, необходимое для проведения успешного поиска (включая время вычисления хеш-функции), составляет $\Theta(2 + \alpha/2 - \alpha/2n) = \Theta(1 + \alpha)$. ■

3.3 Программная реализация алгоритма

3.3.1 Исходный код

```
import random
import math
import time
```

```
class HashTable():
    """Hash table implementation."""
    def __init__(self, size, structType='dict'):
        """Init all necessary parameters."""
        self.size = size
        self.structType = structType
        self.hashTable = dict()
        self.lst = [[x, None] for x in range(self.size)]

        self.initHashTable()
        self.currentSize = 0
        self.comparisons = 0

        self.A = (5**(1/2) - 1) / 2
        self.M = 2**64

    def initHashTable(self):
        """Init blank hash table."""
        for i in range(self.size):
            self.hashTable[i] = None

    def getKey(self, value):
        """Get key using multiplying algorithm."""
        return math.floor(self.M * (value * self.A % 1))

    def getKey2(self, value):
        """Get key using divide algorithm."""
        return value % self.size

    def add(self, value):
        """Add new element to the hash table."""
        if (self.currentSize >= self.size):
            print("Out of free space in the hash table.")
            return

        self.comparisons += 1

        i = 0
        x = self.getKey(value)
        y = self.getKey2(value)

        key = x
```

```
        while True:
            # key = (x + i*y) % self.size
            key = key % self.size

            if (self.structType == 'dict'):
                if (self.hashTable[key] is None):
                    self.hashTable[key] = value
                    self.currentSize += 1
                    break
                else:
                    i += 1
                    key += 1
            else:
                if (self.lst[key][1] is None):
                    self.lst[key][1] = value
                    self.currentSize += 1
                    break
                else:
                    i += 1
                    key += 1

            self.comparisons += 2

    def search(self, key):
        """Get element by key from the hash table."""
        if (key >= 0 and key < self.size):
            if (self.structType == 'dict'):
                return self.hashTable[key]
            else:
                return self.lst[key][1]

        return None

    def show(self):
        """Show all non-empty elements in the hash table."""
        counter = 0

        if (self.structType == 'dict'):
            for (key, value) in self.hashTable.items():
                if value is not None:
                    print(key, value)
                    counter += 1
        else:
            for key, value in self.lst:
```

```
                if value is not None:
                    print(key, value)
                    counter += 1

        print("\nSize:", counter, "\n")

    def getNumOfComparisons(self):
        """Get number of comparisons in the algorithm."""
        return self.comparisons


def main():
    values = [random.randint(0, 100) for x in range(100)]

    tableSize = 101
    structType = 'arr'

    hashTable = HashTable(tableSize, structType)

    for value in values:
        hashTable.add(value)

    hashTable.show()

    start = time.time()

    print("Search(0):", hashTable.search(0))
    print("Search(1):", hashTable.search(1))
    print("Search(2):", hashTable.search(2))

    end = time.time()

    print("Search time (of one element): {:.8f} sec".format((end - start) / 3))

    print("\nNumber of comparisons:", hashTable.getNumOfComparisons())

    if (structType == 'dict'):
        print('\nStructure type: doubly linked list')
    else:
        print('\nStructure type: 2 dimensional array')


if __name__ == "__main__":
    main()
```

3.3.2 Примеры работы

На рисунках 3.1 и 3.2 показаны примеры работы программы для поиска индекса элемента по ключу для массива на 100 и двусвязного списка на 1000 элементов.

Рисунок 3.1 – Поиск элемента в массиве на 100 элементов

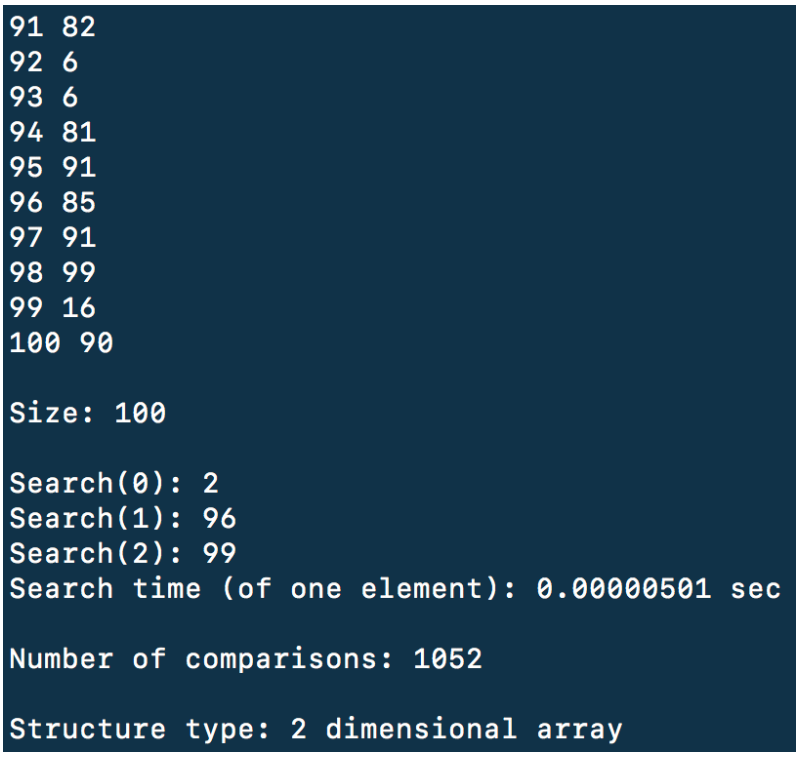
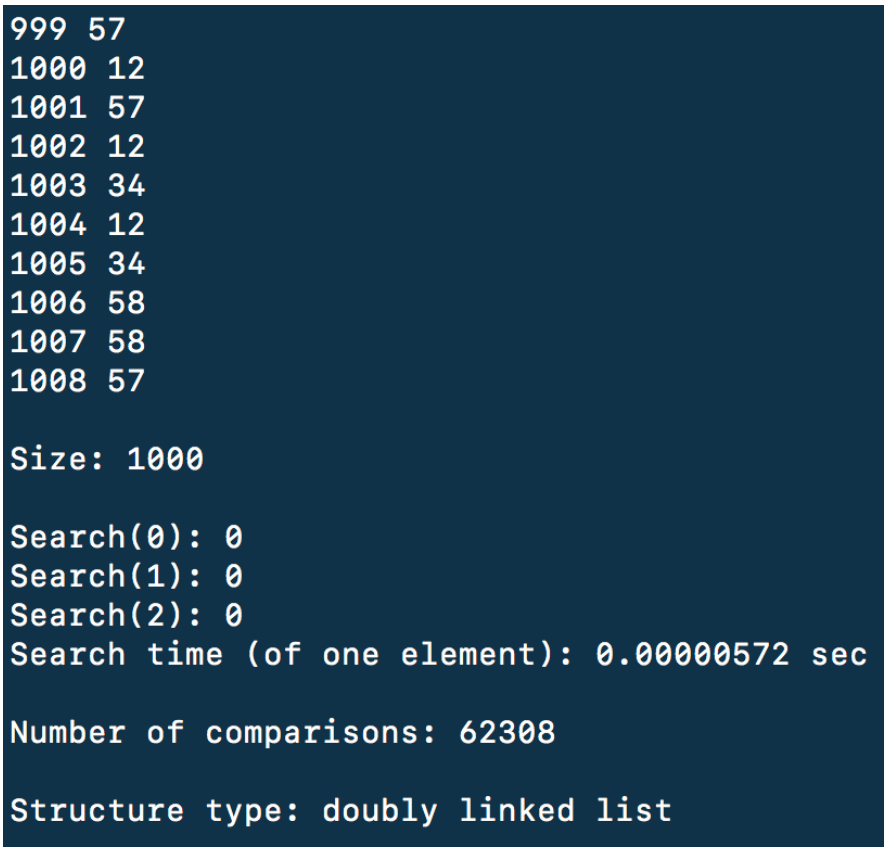


Рисунок 3.2 – Поиск элемента в двусвязном списке на 1000 элементов



3.4 Испытания алгоритма

3.4.1 Временные оценочные характеристики

В таблице 3.1 приведены оценочные характеристики числа сравнений при поиске элемента и времени поиска алгоритма «хеш-функции» для массивов разной размерности и двусвязных списков разной размерности.

Таблица 3.1 – Оценочные характеристики алгоритма «хеш-функции»

Размерность массива/списка	Число сравнений	Время поиска в массиве	Время поиска в двусвязном списке
100	2032/2060	0.00000493	0.00000469
1000	84036/83374	0.00000461	0.00000469
5000	2720280/2701506	0.00000636	0.00000612
10000	18255384/18721246	0.00000628	0.00000596
20000	23083124/20013112	0.00000660	0.00000636

3.4.2 Графики зависимости временных оценочных характеристик и времени поиска от размерности структур

На рисунке 3.3 показаны графики зависимости временных оценочных характеристик от размерности массива и двусвязного списка.

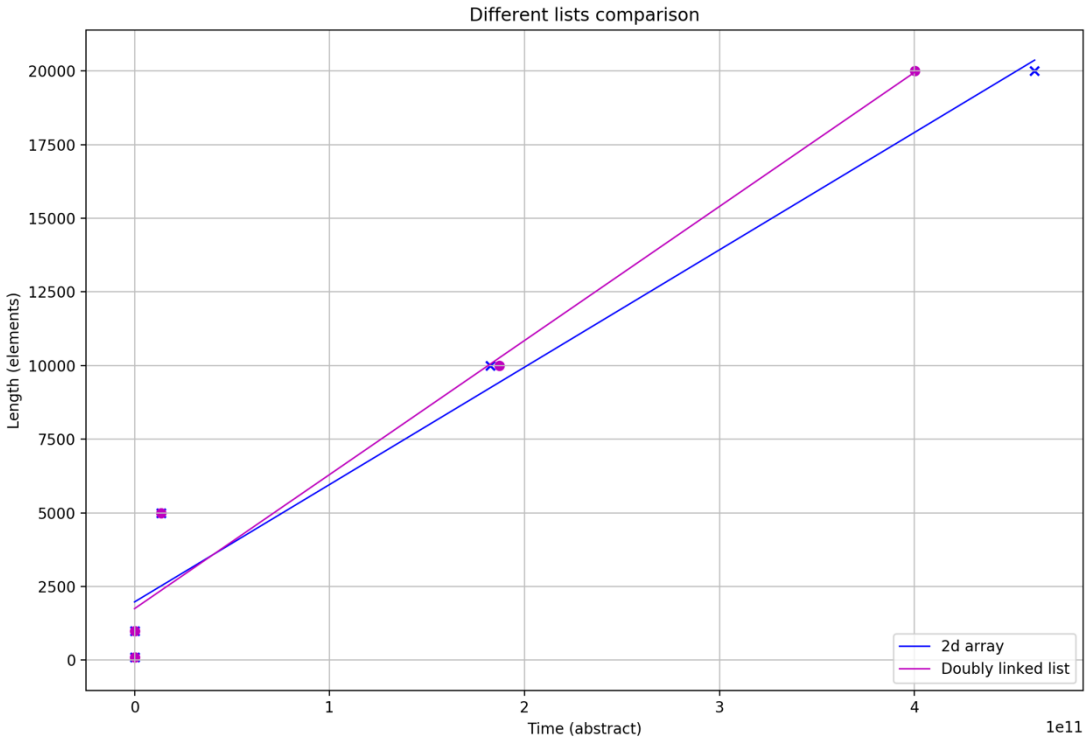
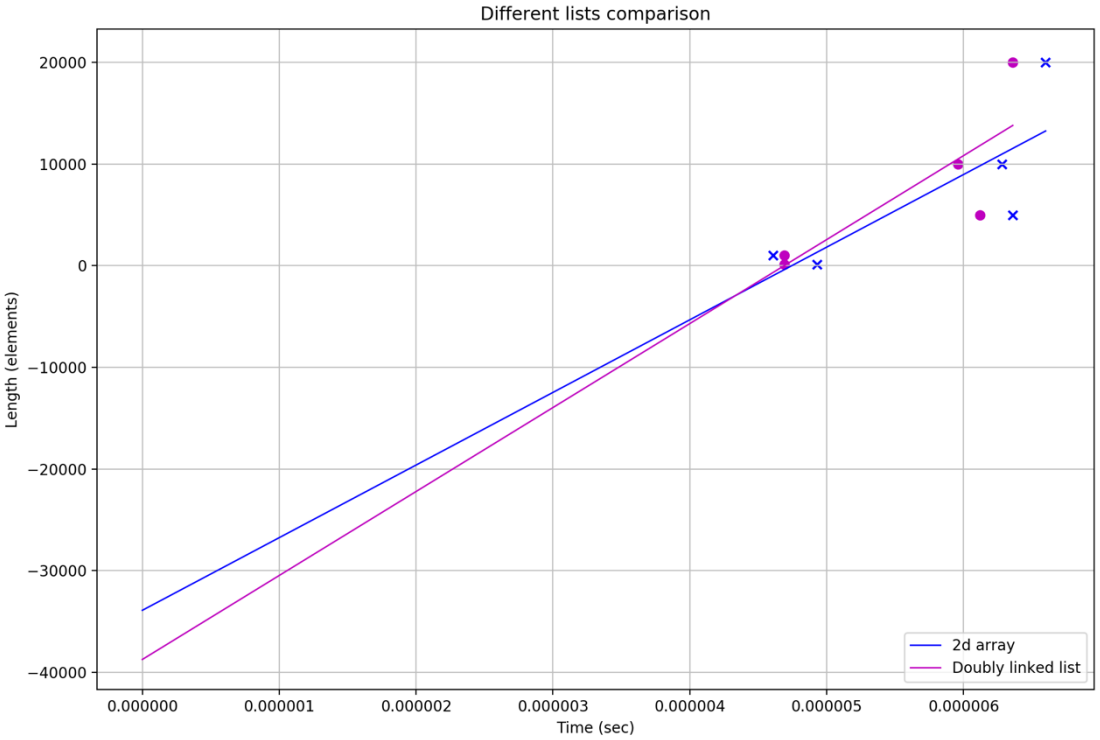


Рисунок 3.3 – Графики зависимости временных оценочных характеристик

На рисунке 3.4 показаны графики зависимости поиска от размерности массива и двусвязного списка.

Рисунок 3.4 – Графики зависимости времени поиска



ВЫВОДЫ

В рамках данной лабораторной работы я познакомился с таким методом хранения данных, как хеш-таблицы, а также рассмотрел некоторые возможные способы избежания коллизий. Также в процессе реализации лабораторной работы посредством языка программирования Python столкнулся (и успешно решил) с особенностями хеш-таблиц для этого языка.