Міністерство освіти і науки України
Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"
Факультет інформатики та обчислювальної техніки

Кафедра автоматизованих систем обробки інформації і управління

Звіт

з лабораторної роботи № 8 з дисципліни
«Алгоритми та структури даних 2. Структури даних»

„ **Проектування і аналіз алгоритмів пошуку**”

Виконав(ла) *ІП-02 Гущін Д.О.* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Головченко М.М.* _____
(прізвище, ім'я, по батькові)

Київ 2021

ЗМІСТ

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ

Мета роботи – вивчити основні підходи аналізу обчислювальної складності алгоритмів пошуку оцінити їх ефективність на різних структурах даних.

2 ЗАВДАННЯ

Згідно варіанту (таблиця 2.1), написати алгоритм пошуку за допомогою псевдокоду (чи іншого способу за вибором).

Провести аналіз часової складності пошуку в гіршому, кращому і середньому випадках і записати часову складність в асимптотичних оцінках.

Виконати програмну реалізацію алгоритму на будь-якій мові програмування для пошуку індексу елемента по заданому ключу в масиві і двохзв'язному списку з фіксацією часових характеристик оцінювання (кількість порівнянь).

Для варіантів з **Хеш-функцією** замість масиву і двохзв'язного списку використати безіндексну структуру даних розмірності *n*, що містить пару ключ-значення рядкового типу. Ключ – унікальне рядкове поле до 20 символів, значення – рядкове поле до 200 символів. Виконати пошук значення по заданому ключу. Розмірність хеш-таблиці регулювати відповідно потребам, а початкову її розмірність обрати самостійно.

Провести ряд випробувань алгоритму на структурах різної розмірності (100, 1000, 5000, 10000, 20000 елементів) і побудувати графіки залежності часових характеристик оцінювання від розмірності структури.

Для проведення випробувань у варіантах з хешуванням рекомендується розробити генератор псевдовипадкових значень полів структури заданої розмірності.

Зробити висновок з лабораторної роботи.

Таблиця 2.1 – Варіанти алгоритмів

№	**Алгоритм пошуку**
1	Однорідний бінарний пошук
2	Метод Шарра
3	Пошук Фібоначчі
4	Інтерполяційний пошук

5	Метод Хеш-функції (Хешування FNV 32), вирішення колізій методом ланцюжків
6	Метод Хеш-функції (Хешування MurmurHash2), вирішення колізій методом ланцюжків
7	Метод Хеш-функції (Хешування MurmurHash2a), вирішення колізій методом ланцюжків
8	Метод Хеш-функції (Хешування PJW-32), вирішення колізій методом ланцюжків
9	Метод Хеш-функції (Хешування Пірсона), вирішення колізій методом ланцюжків
10	Метод Хеш-функції (Хешування Дженкінса), вирішення колізій методом ланцюжків
11	Метод Хеш-функції (Хешування FNV 32), вирішення колізій методом відкритої адресації з лінійним пробуванням
12	Метод Хеш-функції (Хешування MurmurHash2), вирішення колізій методом відкритої адресації з лінійним пробуванням
13	Метод Хеш-функції (Хешування MurmurHash2a), вирішення колізій методом відкритої адресації з лінійним пробуванням
14	Метод Хеш-функції (Хешування PJW-32), вирішення колізій методом відкритої адресації з лінійним пробуванням
15	Метод Хеш-функції (Хешування Пірсона), вирішення колізій методом відкритої адресації з лінійним пробуванням
16	Метод Хеш-функції (Хешування Дженкінса), вирішення колізій методом відкритої адресації з лінійним пробуванням
17	Метод Хеш-функції (Хешування FNV 32), вирішення колізій методом відкритої адресації з квадратичним пробуванням
18	Метод Хеш-функції (Хешування MurmurHash2), вирішення колізій методом відкритої адресації з квадратичним пробуванням
19	Метод Хеш-функції (Хешування MurmurHash2a), вирішення колізій

	методом відкритої адресації з квадратичним пробуванням
20	Метод Хеш-функції (Хешування PJW-32), вирішення колізій методом відкритої адресації з квадратичним пробуванням
21	Метод Хеш-функції (Хешування Пірсона), вирішення колізій методом відкритої адресації з квадратичним пробуванням
22	Метод Хеш-функції (Хешування Дженкінса), вирішення колізій методом відкритої адресації з квадратичним пробуванням
23	Метод Хеш-функції (Хешування FNV 32), вирішення колізій методом відкритої адресації з подвійним хешуванням
24	Метод Хеш-функції (Хешування MurmurHash2), вирішення колізій методом відкритої адресації з подвійним хешуванням
25	Метод Хеш-функції (Хешування MurmurHash2a), вирішення колізій методом відкритої адресації з подвійним хешуванням
26	Метод Хеш-функції (Хешування PJW-32), вирішення колізій методом відкритої адресації з подвійним хешуванням
27	Метод Хеш-функції (Хешування Пірсона), вирішення колізій методом відкритої адресації з подвійним хешуванням
28	Метод Хеш-функції (Хешування Дженкінса), вирішення колізій методом відкритої адресації з подвійним хешуванням
29	Однорідний бінарний пошук
30	Метод Шарра
31	Пошук Фібоначчі
32	Інтерполяційний пошук
33	Метод Хеш-функції (Хешування FNV 32), вирішення колізій методом ланцюжків
34	Метод Хеш-функції (Хешування MurmurHash2), вирішення колізій методом ланцюжків
35	Метод Хеш-функції (Хешування MurmurHash2a), вирішення колізій методом ланцюжків

3 ВИКОНАННЯ

3.1 Псевдокод алгоритму

```
procedure insert(string value)
    NumberOfElements++
    if TableSize <= NumberOfElements
        resize()
    end if
    unsigned int hash = PearsonHashing(value)
    while (Table[hash] != NULL)
        hash++;
        hash %= TableSize;
    end while
    Table[hash] = new Cell(to_string(hash), value)


function PearsonHashing(string value)
     h := 0
     for each c in C loop
         h := T[h xor c]
     end loop
     return h


function search(string key)
    int hash = PearsonHashing(key)
    while (Table[hash] != NULL)
        if Table[hash]->getKey() == key
            return Table[hash]
         end if
        hash++
        hash = hash % TableSize
    end while
    return NULL
```

3.2 Аналіз часової складності

Кращий випадок

В найкращому випадку операція пошуку в хеш-таблиці виконується за O(1), оскільки досить велика таблиця може зменшити колізії майже до постійного часу.

Середнє

В середньому випадку операція пошуку в хеш-таблиці виконується за O(1). Хоча твердження O(1) не завжди правильне, воно приблизно вірно для багатьох реальних ситуацій.

Найгірший випадок

В найгіршому випадку операція пошуку в хеш-таблиці виконується за O(n). Однак, при сильно заповненій таблиці пошук елемента прагне до O(n), тому що через велику кількість колізій ми будемо перебирати майже всі елементи, поки вони не закінчаться або випадково не потрапимо в null.

3.3 Програмна реалізація алгоритму

3.3.1 Вихідний код

```
#pragma once
#include <iostream>
#include <string>
#include <algorithm>    // std::shuffle
#include <chrono>       // std::chrono::system_clock
#include <random>       // std::default_random_engine
#include <vector>
using namespace std;

class Cell {
private:
     string key, value;
     bool isDeleted;
public:
     Cell();
     Cell(string key, string value);
     Cell(const Cell& other);
     void setKey(string value);
     void setValue(string key);
     void setDeleted(bool isDeleted);
     string getKey();
     string getValue();
     bool getDeleted();
};


class HashTable {
private:
     int TableSize, NumberOfElements;
     Cell** Table;
     string charset;
     unsigned int PearsonHashing(string x);
```

```
    void resize();
public:
    HashTable();
    HashTable(int size);
    string StringGenerator();
    int getSize();
    void insert(string value);
    void remove(string key);
    Cell* search(string key);
    void print();
    };


#include "HashTable.hpp"

using namespace std;

Cell::Cell() {
    key = value = "";
    isDeleted = false;
}


Cell::Cell(string key, string value)
{
    this->key = key;
    this->value = value;
    isDeleted = false;
}


Cell::Cell(const Cell& other) {
    this->key = key;
    this->value = value;
}


void Cell::setKey(string key)
{
    this->key = key;
}


void Cell::setValue(string value)
```

```
{
    this->value = value;
}


void Cell::setDeleted(bool isDeleted) {
    this->isDeleted = isDeleted;
}


string Cell::getKey()
{
    return key;
}


string Cell::getValue()
{
    return value;
}


bool Cell::getDeleted() {
    return isDeleted;
}


HashTable::HashTable() {
    TableSize = NumberOfElements = 0;
    charset =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ123456789
0";
    Table = new Cell*[TableSize];
    for (int i = 0; i < TableSize; i++)
        Table[i] = NULL;
}


HashTable::HashTable(int size)
{
    NumberOfElements = 0;
    this->TableSize = size;
```

```
    charset =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ123456789
0";
    Table = new Cell*[TableSize];
    for (int i = 0; i < TableSize; i++)
        Table[i] = NULL;
}


int HashTable::getSize() {
    return TableSize;
}


string HashTable::StringGenerator() {
    string result;
    int length = rand() % 200;
    for (int i = 0; i < length; i++)
        result += charset[rand() % charset.size()];
    return result;
}


unsigned int HashTable::PearsonHashing(string str)
{
    vector <unsigned char> T(256);
    for (int i = 0; i < 256; i++) {
        T[i] = i - '0';
    }
    unsigned int seed =
chrono::system_clock::now().time_since_epoch().count();
    shuffle(T.begin(), T.end(), default_random_engine(seed));

    unsigned int hash = 0;
    unsigned int current_hash = 0;
    for (int i = 0; i < 8; ++i) {
        current_hash = T[(str[0] + i) % 256];
        for (unsigned char symbol : str) {
            current_hash = T[current_hash ^ symbol];
        }
        hash += current_hash;
    }

    return hash % TableSize;
```

```
}


void HashTable::insert(string value)
{
    NumberOfElements++;
    if (TableSize <= NumberOfElements) {
        resize();
    }
    unsigned int hash = PearsonHashing(value);
    while (Table[hash] != NULL)
        ++hash %= TableSize;
    Table[hash] = new Cell(to_string(hash), value);

}


void HashTable::print()
{
    cout << endl;
    for (int i = 0; i < TableSize; i++)
        if (Table[i] != NULL && !Table[i]->getDeleted())
            cout << Table[i]->getKey() + "   " + Table[i]-
>getValue() << endl;
    cout << endl;
}


Cell* HashTable::search(string key)
{
    int hash = PearsonHashing(key);
    for (int i = 0; i < TableSize; i++)
    {
        if (Table[hash] != NULL && !Table[hash]->getDeleted()
&& Table[hash]->getKey() == key)
            return Table[hash];
        hash = ++hash % TableSize;
    }

    return NULL;
}


void HashTable::resize() {
```

```
    int oldSize = TableSize;
    TableSize *= 2;
    Cell** tmp = new Cell*[TableSize];
    for (int i = 0; i < TableSize; i++)
        tmp[i] = nullptr;
    swap(Table, tmp);
    for (int i = 0; i < oldSize; i++)
        if (tmp[i] != nullptr)
            insert(tmp[i]->getValue());
    for (int i = 0; i < oldSize; i++)
        if (tmp[i] != nullptr)
            delete tmp[i];
    delete[] tmp;
}


void HashTable::remove(string key)
{
    bool isDeleted = false;
    for (int i = 0; i < TableSize; i++)
    {
        int hash = PearsonHashing(key);
        if (Table[hash] != NULL && !Table[hash]->getDeleted())
{
            if (Table[hash]->getKey() == key)
            {
                delete Table[hash];
                this->Table[hash]->setDeleted(true);
                cout << "Deleted" << endl;
                isDeleted = true;
                break;
            }
        }
    }
    if (!isDeleted)
        cout << "Not found" << endl;
}
```

3.3.2 Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми для пошуку індекса елемента за ключем для масиву на 100 елементів і двохзв'язного списку на 1000 елементів.

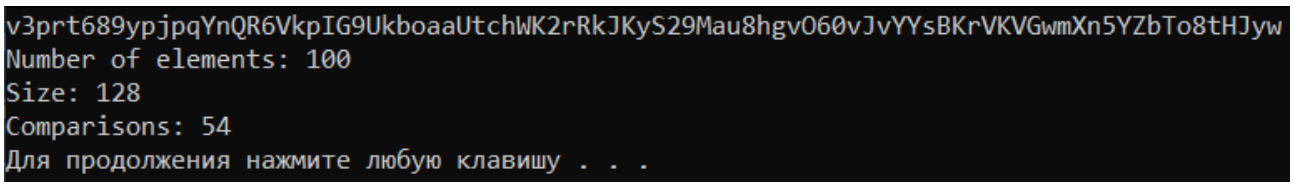

Рисунок 3.1 – Пошук елемента в масиві на 100 елементів

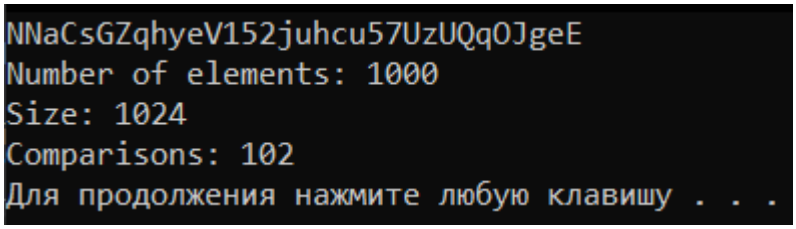

Рисунок 3.2 – Пошук елемента в двохзв'язному списку на 1000 елементів

3.4 Тестування алгоритму

3.4.1 Часові характеристики оцінювання

В таблиці 3.1 наведені характеристики оцінювання числа порівнянь при пошуку елемента в хеш-таблиці для масивів структур розмірності.

Таблиця 3.1 –Характеристики оцінювання пошуку хеш-функції

Розмірність структури	Число порівнянь в хеш-таблиці
100	94
1000	578
5000	3545
10000	9043
20000	14385

3.4.2 Графіки залежності часових характеристик оцінювання від розмірності структури

На рисунку 3.3 показані графіки залежності часових характеристик оцінювання від розмірності структури.

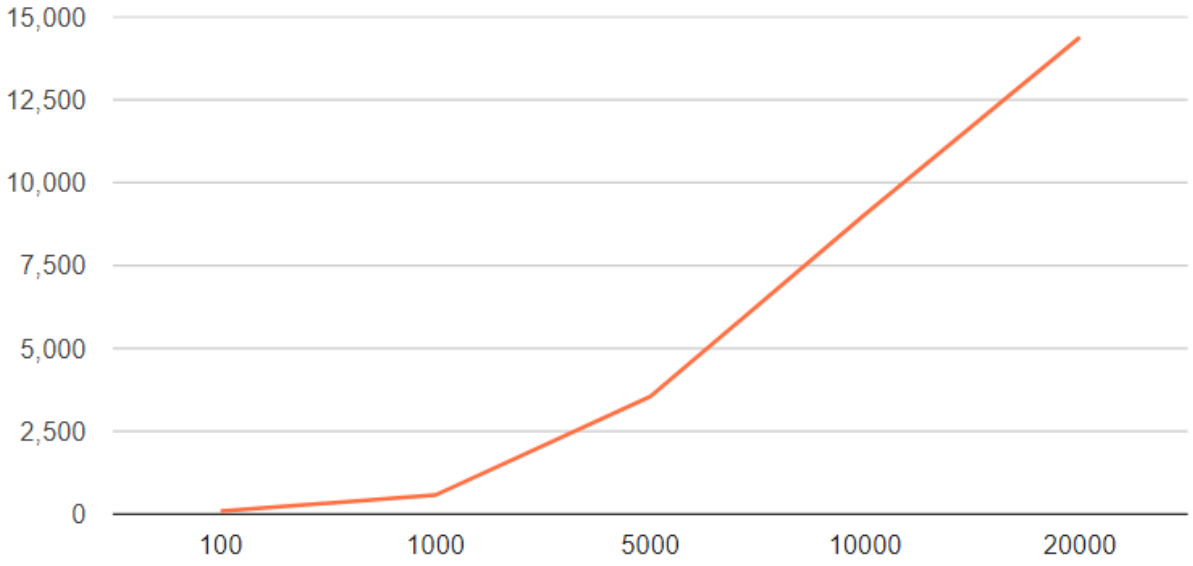

Рисунок 3.3 – Графіки залежності часових характеристик оцінювання

ВИСНОВОК

При виконанні даної лабораторної роботи ми вивчили основні підходи аналізу обчислювальної складності алгоритмів пошуку, оцінили їх ефективність на різних структурах даних сортування, розробили алгоритм розв’язання задачі відповідно до варіанту, виконали аналіз алгоритму пошуку, записали алгоритм за допомогою псевдокоду, а також виконали програмну реалізацію задачі, провели аналіз часової складності в гіршому, кращому і середньому випадках та записали часову складність в асимптотичних оцінках, провели ряд випробувань алгоритму на даних різної розмірності і побудували графіки залежності часових характеристик оцінювання від розмірності масиву.

КРИТЕРІЇ ОЦІНЮВАННЯ

За умови здачі лабораторної роботи до 1.06.2020 включно надається можливість виправити помилки без втрати балів, за бажанням. Після 1.06.2020 оцінка за лабораторну роботу ставиться по факту здачі. Максимальний бал дорівнює 5.

Критерії оцінювання у відсотках від максимального балу:

- псевдокод алгоритму – 20%;
- аналіз часової складності– 20%;
- програмна реалізація алгоритму– 25%;
- тестування алгоритму– 20%;
- висновки – 5%.