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

Кафедра ІСТ

Звіт

з лабораторної роботи № 7 з дисципліни
«Теорія алгоритмів»

„ **Проектування і аналіз алгоритмів пошуку**”

Виконали ІА-31 Клим’юк В.Л, Самелюк А.С, Дук М.Д, Сакун Д.С

Перевірив Степанов А.С

Київ 2024

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

```
function binary_search(list, target):
    left = 0
    right = length(list) - 1
    while left <= right:
        mid = (left + right) // 2
        if list[mid] == target:
            return mid
        elif list[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

3.2 Аналіз часової складності

```
DateTime startTime = DateTime.Now;
int comparisons;
int result = Algorithm.BinarySearchArray(array, key, out comparisons);
DateTime endTime = DateTime.Now;
TimeSpan executionTime = endTime - startTime;
```

```
DateTime startTime = DateTime.Now;
int comparisons;
DoublyLinkedListNode result = Algorithm.BinarySearchDoublyLinkedList(list, key, out
comparisons);
DateTime endTime = DateTime.Now;
TimeSpan executionTime = endTime - startTime;
```

3.3 Програмна реалізація алгоритму

```
using System;

namespace Classes
{

    public class Algorithm
    {
        public static int BinarySearchArray(int[] array, int key, out int comparisons)
        {
            int left = 0;
            int right = array.Length - 1;
            comparisons = 0;

            while (left <= right)
```

```
            {
                int mid = left + (right - left) / 2;
                comparisons++;

                if (array[mid] == key)
                    return mid;

                if (array[mid] < key)
                    left = mid + 1;
                else
                    right = mid - 1;
            }

            return -1;
        }

        public static DoublyLinkedListNode
BinarySearchDoublyLinkedList(DoublyLinkedList list, int key, out int comparisons)
        {
            comparisons = 0;
            int left = 0;
            int right = list.Count - 1;

            while (left <= right)
            {
                int mid = left + (right - left) / 2;
                DoublyLinkedListNode midNode = GetNodeAtIndex(list, mid);
                comparisons++;

                if (midNode.Value == key)
                {
                    return midNode;
                }
                else if (midNode.Value < key)
                {
                    left = mid + 1;
                }
                else
                {
                    right = mid - 1;
                }
            }

            return null;
        }

        public static DoublyLinkedListNode GetNodeAtIndex(DoublyLinkedList list, int
index)
        {
            if (index < 0 || index >= list.Count)
```

```
            {
                throw new IndexOutOfRangeException("Invalid index");
            }

            DoublyLinkedListNode current = list.Head;

            for (int i = 0; i < index; i++)
            {
                current = current.Next;
            }

            return current;
        }
    }

    public class DoublyLinkedListNode
    {
        public int Value { get; set; }
        public int Index { get; set; }
        public DoublyLinkedListNode Previous { get; set; }
        public DoublyLinkedListNode Next { get; set; }

        public DoublyLinkedListNode(int value, int index)
        {
            Value = value;
            Index = index;
        }



    }

    public class DoublyLinkedList
    {
        public DoublyLinkedListNode Head { get; private set; }
        public DoublyLinkedListNode Tail { get; private set; }
        public int Count { get; private set; }

        public void AddLast(int value)
        {
            DoublyLinkedListNode newNode = new DoublyLinkedListNode(value, Count);

            if (Head == null)
            {
                Head = newNode;
                Tail = newNode;
            }
            else
            {
                newNode.Previous = Tail;
```

```
                Tail.Next = newNode;
                Tail = newNode;
            }

            Count++;
        }
        private static DoublyLinkedListNode
InsertNodeIntoSortedDoublyLinkedList(DoublyLinkedListNode sorted, DoublyLinkedListNode
newNode)
        {
            if (sorted == null)
            {
                newNode.Next = null;
                newNode.Previous = null;
                return newNode;
            }

            if (newNode.Value <= sorted.Value)
            {
                newNode.Next = sorted;
                newNode.Previous = null;
                sorted.Previous = newNode;
                return newNode;
            }

            DoublyLinkedListNode current = sorted;
            while (current.Next != null && current.Next.Value < newNode.Value)
            {
                current = current.Next;
            }

            newNode.Next = current.Next;
            newNode.Previous = current;
            if (current.Next != null)
            {
                current.Next.Previous = newNode;
            }
            current.Next = newNode;

            return sorted;
        }

        public static DoublyLinkedList SortDoublyLinkedList(DoublyLinkedList list)
        {
            if (list == null || list.Head == null || list.Head.Next == null)
            {
                return list;
            }

            DoublyLinkedListNode sorted = null;
```

```
            DoublyLinkedListNode current = list.Head;

            while (current != null)
            {
                DoublyLinkedListNode next = current.Next;
                sorted = InsertNodeIntoSortedDoublyLinkedList(sorted, current);
                current = next;
            }

            list.Head = sorted;
            return list;
        }

    }



}
```

3.3.1 Вихідний код

```
using Classes;

namespace Lab7
{
    internal class Program
    {
        static void Main(string[] args)
        {
            int[] sizes = { 100, 1000, 5000, 10000, 20000 };

            foreach (int size in sizes)
            {
                int[] array = GenerateRandomArray(size);
                int key = array[new Random().Next(size)];

                DateTime startTime = DateTime.Now;
                int comparisons;
                int result = Algorithm.BinarySearchArray(array, key, out comparisons);
                DateTime endTime = DateTime.Now;
                TimeSpan executionTime = endTime - startTime;

                Console.WriteLine($"Array size: {size}, Time:
{executionTime.TotalMilliseconds} ms, Comparisons: {comparisons}");
            }
```

```
            foreach (int size in sizes)
            {
                DoublyLinkedList list = GenerateRandomDoublyLinkedList(size);
                int key = new Random().Next(1000);

                DateTime startTime = DateTime.Now;
                int comparisons;
                DoublyLinkedListNode result =
Algorithm.BinarySearchDoublyLinkedList(list, key, out comparisons);
                DateTime endTime = DateTime.Now;
                TimeSpan executionTime = endTime - startTime;

                Console.WriteLine($"List size: {size}, Time:
{executionTime.TotalMilliseconds} ms, Comparisons: {comparisons}");
            }
        }

        public static int[] GenerateRandomArray(int size)
        {
            int[] array = new int[size];
            Random rand = new Random();

            for (int i = 0; i < size; i++)
            {
                array[i] = rand.Next(1000);
            }

            Array.Sort(array);
            return array;
        }

        public static DoublyLinkedList GenerateRandomDoublyLinkedList(int size)
        {
            DoublyLinkedList list = new DoublyLinkedList();
            Random rand = new Random();

            for (int i = 0; i < size; i++)
            {
                list.AddLast(rand.Next(1000));
            }

            return DoublyLinkedList.SortDoublyLinkedList(list);
        }


    }

}
```

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

```
Array size: 100, Time: 3,7788 ms, Comparisons: 4
Array size: 1000, Time: 0,0068 ms, Comparisons: 7
Array size: 5000, Time: 0,0031 ms, Comparisons: 9
Array size: 10000, Time: 0,0021 ms, Comparisons: 10
Array size: 20000, Time: 0,002 ms, Comparisons: 9
List size: 100, Time: 0,2341 ms, Comparisons: 6
List size: 1000, Time: 0,0088 ms, Comparisons: 9
List size: 5000, Time: 0,0961 ms, Comparisons: 12
List size: 10000, Time: 0,2373 ms, Comparisons: 14
List size: 20000, Time: 0,0776 ms, Comparisons: 14
```

3.4.1 Часові характеристики оцінювання

В таблиці 3.1 наведені характеристики оцінювання числа порівнянь при пошуку елемента і числа звертань при «двійковому пошуку» для масивів різної розмірності і двохзв'язних списків різної розмірності.

Таблиця 3.1 –Характеристики оцінювання алгоритму двійкового пошуку

Розмірність масиву/списку/ структури	Число порівнянь в масиві/двохзв'язному списку/хеш-таблиці	Число звертань до елементів масиву	Число звертань до елементів двохзв'язного списку
100	4/6	4	6
1000	7/9	7	9
5000	9/12	9	12
10000	10/14	10	14
20000	9/14	9	14

3.4.2 Графіки залежності часових характеристик оцінювання від розмірності структури

На рисунку 3.3 показані графіки залежності часових характеристик оцінювання від розмірності масиву і двохзв'язного списку.

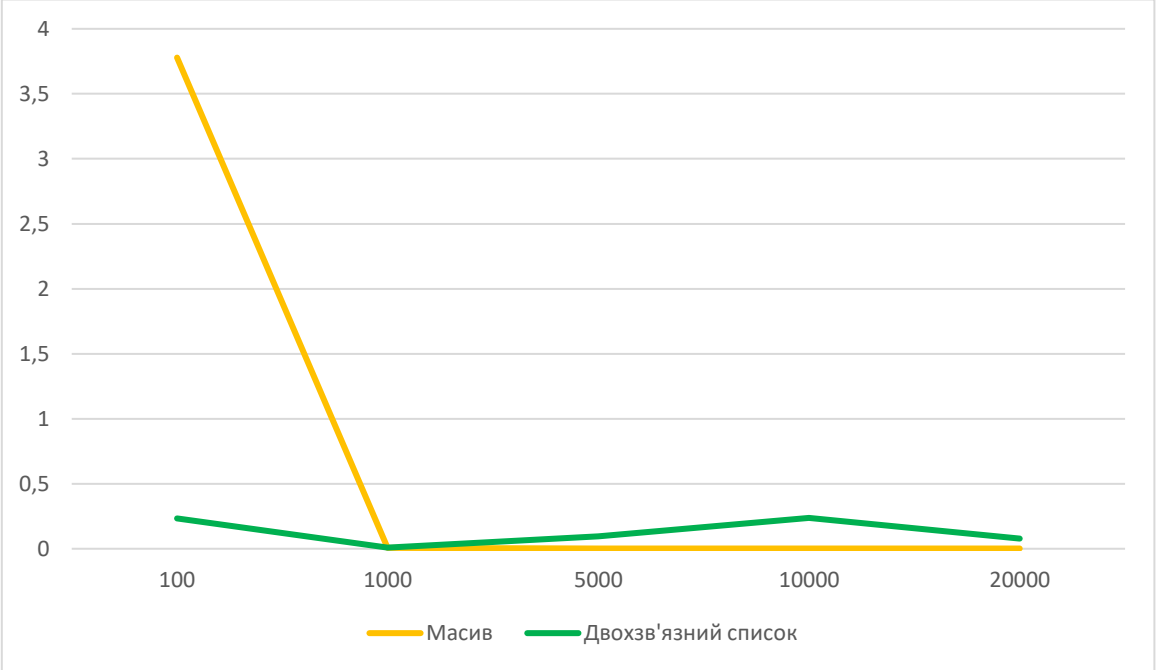

Рисунок 3.3 – Графіки залежності часових характеристик оцінювання

ВИСНОВОК

Лабораторна робота дозволила вивчити основні підходи аналізу обчислювальної складності алгоритмів пошуку та оцінити їх ефективність на різних структурах даних, таких як масиви і двохзв'язні списки. Отже, в ході лабораторної роботи ми оцінили ефективність бінарного пошуку на різних структурах даних і виявили, що для операцій пошуку впорядкованих даних масив є більш ефективним варіантом, особливо на великих обсягах даних.

КРИТЕРІЇ ОЦІНЮВАННЯ

Див РСО