**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

«Теория алгоритмов»

„ **Проектирование и анализ алгоритмов поиска** ”

**Выполнил**

(шифр, фамилия, имя, отчество)

*ІП - 8121*

**Проверил**

(фамилия, имя, отчество )

*Головченко М.Н.*

Киев 2019

СОДЕРЖАНИЕ

[**1**](#_1ksv4uv) **ЦЕЛЬ ЛАБОРАТОРНОЙ РАБОТЫ 3**

[**2**](#_44sinio) **ЗАДАНИЕ 4**

[**3**](#_2jxsxqh) **ВЫПОЛНЕНИЕ 8**

[3.1](#_z337ya) Псевдокод алгоритма 8

[3.2](#_3j2qqm3) Анализ временной сложности 8

[3.3](#_1y810tw) Программная реализация алгоритма 8

[*3.3.1*](#_4i7ojhp) *Исходный код 8*

[*3.3.2*](#_2xcytpi) *Примеры работы 8*

[3.4](#_1ci93xb) Испытания алгоритма 9

[*3.4.1*](#_3whwml4) *Временные оценочные характеристики 9*

[*3.4.2*](#_2bn6wsx) *Графики зависимости временных оценочных характеристик от размерности структур 10*

[**ВЫВОДЫ 11**](#_qsh70q)

[**КРИТЕРИИ ОЦЕНИВАНИЯ 12**](#_3as4poj)

1. ЦЕЛЬ ЛАБОРАТОРНОЙ РАБОТЫ

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

1. ЗАДАНИЕ

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

Провести анализ временной сложности поиска 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).

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

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 |
| 31 | Метод Хеш-функции(«Хеш-функции», основанные на делении), разрешения коллизий методом открытой адресация с квадратичным пробированием |
| 32 | Метод Хеш-функции(«Хеш-функции», основанные на делении), разрешения коллизий методом открытой адресация с двойным хешированием |
| 33 | Метод Хеш-функции(«Хеш-функции», основанные на умножении), разрешения коллизий методом цепочек |
| 34 | Метод Хеш-функции(«Хеш-функции», основанные на умножении), разрешения коллизий методом открытой адресация с линейным пробированием |
| 35 | Метод Хеш-функции(«Хеш-функции», основанные на умножении), разрешения коллизий методом открытой адресация с квадратичным пробированием |

1. ВЫПОЛНЕНИЕ
   1. Псевдокод алгоритма

pos = size \*((hash(key)\*0.618033)%1)

**for** (int i = 0; i < items[pos].length; i++)

**if** (items[pos][i] == key)

return items[pos][i].Value

**end if**

**end for**

* 1. Анализ временной сложности

Поиск элемента в Хеш-таблице:

Худший случай = O(n);

Средний случай = Ω(1 + n/p), где n – количество элементов, p – размерность массива/списка;

Лучший случай = ϴ(1);

* 1. Программная реализация алгоритма
     1. Исходный код

public class HashTable<TKey, TValue>

{

private LinkedList<Tuple<TKey, TValue>>[] \_items;

private int \_fillFactor = 1;

private int \_size = 4;

private int \_elements;

public int count = 0;

public HashTable()

{

\_items = new LinkedList<Tuple<TKey, TValue>>[4];

}

private HashTable(int length)

{

\_items = new LinkedList<Tuple<TKey, TValue>>[length];

\_size = length;

}

public void Add(TKey key, TValue value)

{

var pos = GetPosition(key);

if (\_items[pos] == null)

{

\_items[pos] = new LinkedList<Tuple<TKey, TValue>>();

}

if (\_items[pos].Any(x => x.Item1.Equals(key)))

{

throw new Exception("Duplicate key, cannot insert.");

}

\_elements++;

if (NeedToGrow())

{

GrowAndReHash();

}

pos = GetPosition(key);

if (\_items[pos] == null)

{

\_items[pos] = new LinkedList<Tuple<TKey, TValue>>();

}

\_items[pos].AddFirst(new Tuple<TKey, TValue>(key, value));

}

public void Remove(TKey key)

{

var pos = GetPosition(key);

if (\_items[pos] != null)

{

var objToRemove = \_items[pos].FirstOrDefault(item => item.Item1.Equals(key));

if (objToRemove == null) return;

\_items[pos].Remove(objToRemove);

\_elements--;

}

else

{

throw new Exception("Value not in HashTable.");

}

}

public TValue Get(TKey key)

{

var pos = GetPosition(key);

for (int i = 0; i < \_items[pos].Count; i++)

{

count++;

if (\_items[pos].ElementAt(i).Item1.Equals(key))

return \_items[pos].ElementAt(i).Item2;

}

throw new Exception("Key does not exist in HashTable.");

}

private void GrowAndReHash()

{

\_size = \_size \* 2;

var newHashTable = new HashTable<TKey, TValue>(\_size);

foreach (var item in \_items.Where(x => x != null))

{

foreach (var value in item)

{

newHashTable.Add(value.Item1, value.Item2);

}

}

\_items = newHashTable.\_items;

}

private int GetPosition(TKey key)

{

var hash = key.GetHashCode();

var pos = Math.Abs(hash);

pos = (int)(\_size \* ((pos \* 0.618033) % 1));

return pos;

}

private bool NeedToGrow()

{

return \_elements/\_size >= \_fillFactor;

}

}

* + 1. 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

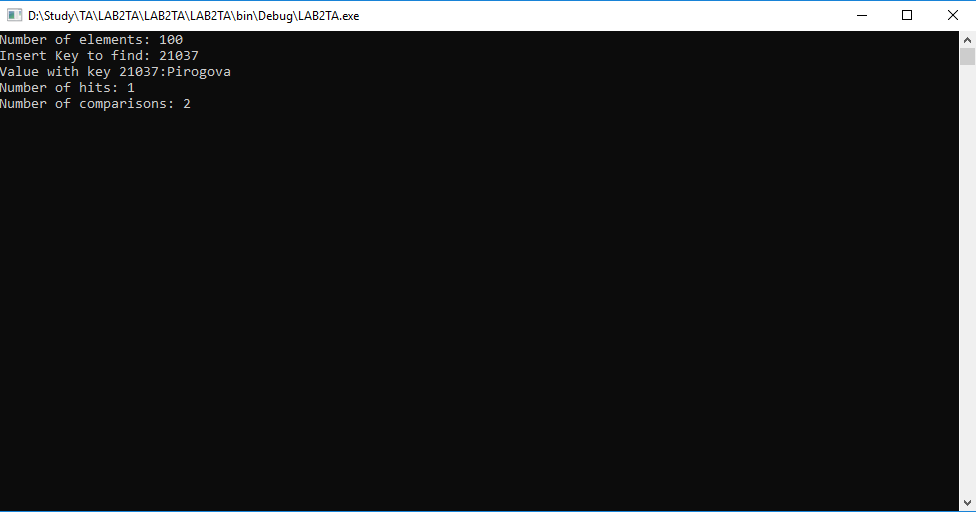
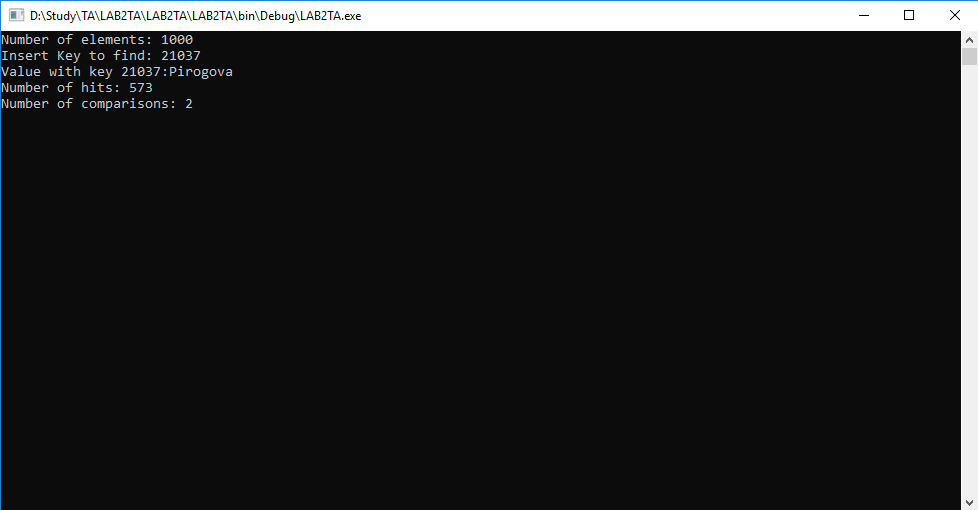


Рисунок 3.2 – Поиск элемента в двусвязном списке на 1000 элементов

Испытания алгоритма

* + 1. Временные оценочные характеристики

В таблице 3.1 приведены оценочные характеристики числа сравнений при поиске элемента и числа обращений при «Метод Хеш-функции(«Хеш-функции», основанные на делении), разрешения коллизий методом цепочек» для массивов разной размерности и двусвязных списков разной размерности.

Таблица 3.1 – Оценочные характеристики Метода Хеш-функции

|  |  |  |  |
| --- | --- | --- | --- |
| Размерность массива/списка | Число сравнений в массиве/двусвязном списке | Число обращений к элементам массива | Число обращений к элементам двусвязного списка |
| 100 | 2 | 1 | 50 |
| 1000 | 2 | 1 | 500 |
| 5000 | 1 | 1 | 2500 |
| 10000 | 2 | 1 | 5000 |
| 20000 | 1 | 1 | 10000 |

* + 1. Графики зависимости временных оценочных характеристик 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

ВЫВОДЫ

В рамках данной лабораторной работы я изучил основные подходы к анализу вычислительной сложности алгоритмов поиска и оценить их эффективность на различных структурах данных. Алгоритм поиска Метод Хеш-функции(«Хеш-функции», основанные на умножении), разрешения коллизий методом цепочек при некоторых разумных допущениях операция поиска в среднем выполняется за время О(1). Но при этом не гарантируется, что время выполнения отдельной операции мало́. Это связано с тем, что при достижении некоторого значения коэффициента заполнения необходимо осуществлять перестройку индекса хеш-таблицы: увеличить значение размера массива  и заново добавить в пустую хеш-таблицу все пары. При предположении, что каждый элемент может попасть в любую позицию таблицы с равной вероятностью и независимо от того, куда попал любой другой элемент, среднее [время работы](https://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D0%BE%D0%B6%D0%BD%D0%BE%D1%81%D1%82%D1%8C_%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC%D0%B0) операции поиска элемента составляет [О](https://ru.wikipedia.org/wiki/%D0%9E-%D0%BD%D0%BE%D1%82%D0%B0%D1%86%D0%B8%D1%8F)(1 + α), где α — коэффициент заполнения таблицы, но при условии перестройки таблицы при достижение некоторого значения α, (1+ α) всегда будет константой и время поиска будет О(1). Из графиков оценочных характеристик может показаться, что построение хеш-таблицы на основе массива всегда намного эффективнее чем на основе таблицы, но дело в том, что в случае операции вставки и удаления все будет противоположно: массив будет выполнять их за О(n), а список за О(1).