



Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное
учреждение высшего образования
«Московский государственный технический университет имени
Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Отчет по лабораторной работе №7 по курсу "Анализ алгоритмов"

Тема Поиск в словаре

Студент Криков А.В.

Группа ИУ7-53Б

Преподаватели Волкова Л.Л., Строганов Ю.В.

Москва — 2021 г.

Содержание

Введение	2
1 Аналитическая часть	4
1.1 Алгоритм полного перебора	4
1.2 Алгоритм двоичного поиска	4
1.3 Алгоритм частотного анализа	6
2 Конструкторская часть	7
2.1 Разработка алгоритмов	7
2.2 Структура ПО	10
2.3 Описание структур данных	10
2.4 Тестирование	11
3 Технологическая часть	12
3.1 Средства реализации	12
3.2 Листинг кода	12
3.3 Тестирование функций	15
4 Исследовательская часть	16
4.1 Технические характеристики	16
4.2 Временные характеристики	16
Заключение	19
Список литературы	20

Введение

Структура данных, позволяющая идентифицировать ее элементы не по числовому индексу, а по произвольному, называется словарем или ассоциативным массивом. Каждый элемент словаря состоит из двух объектов: ключа и значения. В жизни широко распространены словари, например, привычные бумажные словари (толковые, орфографические, лингвистические). В них ключом является слово-заголовок статьи, а значением — сама статья. Для того, чтобы получить доступ к статье, необходимо указать слово-ключ.

Особенностью ассоциативного массива является его динамичность: в него можно добавлять новые элементы с произвольными ключами и удалять уже существующие элементы. При этом размер используемой памяти пропорционален размеру ассоциативного массива. Доступ к элементам ассоциативного массива выполняется хоть и медленнее, чем к обычным массивам, но в целом довольно быстро.

С появлением словарей появилась нужда в том, чтобы уметь быстро находить нужное значение по ключу. Со временем стали разрабатывать алгоритмы поиска в словаре.

Целью данной работы является изучение следующих алгоритмов поиска в словаре:

- Поиск полным перебором;
- Бинарный поиск;
- Частотный анализ.

В рамках выполнения работы необходимо решить следующие задачи:

- исследовать основные алгоритмы поиска в словаре;
- привести схемы реализации алгоритмов поиска в словаре;
- описать структуру разрабатываемого ПО;
- определить средства программной реализации;
- протестировать разработанное ПО;

- привести сведения о модулях программы;
- определить требования к ПО;
- провести экспериментальные замеры временных характеристик реализованных алгоритмов;
- на основании проделанной работы сделать выводы и подготовить отчет.

1 Аналитическая часть

В данном разделе будут представлены теоретические сведения о рассматриваемых алгоритмах.

1.1 Алгоритм полного перебора

Алгоритмом полного перебора [1] называют метод решения задачи, при котором по очереди рассматриваются все возможные варианты. В нашем случае мы последовательно будем перебирать ключи словаря до тех пор, пока не найдём нужный. Трудоёмкость алгоритма зависит от того, присутствует ли искомый ключ в словаре, и, если присутствует - насколько он далеко от начала массива ключей.

Пусть алгоритм нашёл элемент на первом сравнении (лучший случай), тогда будет затрачено $k_0 + k_1$ операций, на втором - $k_0 + 2 \cdot k_1$, на последнем (худший случай) - $k_0 + N \cdot k_1$. Если ключа нет в массиве ключей, то мы сможем понять это, только перебрав все ключи, таким образом трудоёмкость такого случая равно трудоёмкости случая с ключом на последней позиции. Средняя трудоёмкость может быть рассчитана как математическое ожидание по формуле (1.1), где Ω – множество всех возможных случаев.

$$\begin{aligned} \sum_{i \in \Omega} p_i \cdot f_i &= (k_0 + k_1) \cdot \frac{1}{N+1} + (k_0 + 2 \cdot k_1) \cdot \frac{1}{N+1} + \\ &+ (k_0 + 3 \cdot k_1) \cdot \frac{1}{N+1} + (k_0 + N k_1) \frac{1}{N+1} + (k_0 + N \cdot k_1) \cdot \frac{1}{N+1} = \\ &= k_0 \frac{N+1}{N+1} + k_1 + \frac{1+2+\dots+N+N}{N+1} = \\ &= k_0 + k_1 \cdot \left(\frac{N}{N+1} + \frac{N}{2} \right) = k_0 + k_1 \cdot \left(1 + \frac{N}{2} - \frac{1}{N+1} \right) \end{aligned} \quad (1.1)$$

1.2 Алгоритм двоичного поиска

Бинарный поиск[2] производится в упорядоченном словаре.

При бинарном поиске искомый ключ сравнивается с ключом среднего элемента в словаре. Если они равны, то поиск успешен. В противном случае поиск осуществляется аналогично в левой или правой частях словаря.

Алгоритм может быть определен в рекурсивной и нерекурсивной формах.

Бинарный поиск также называют поиском методом деления отрезка пополам или дихотомии.

На каждом шаге осуществляется поиск середины отрезка по формуле $mid = (left + right)/2$.

Если искомый элемент равен элементу с индексом mid , поиск завершается. В случае если искомый элемент меньше элемента с индексом mid , на место mid перемещается правая граница рассматриваемого отрезка, в противном случае — левая граница. На рис 1.1 приведен пример бинарного поиска.

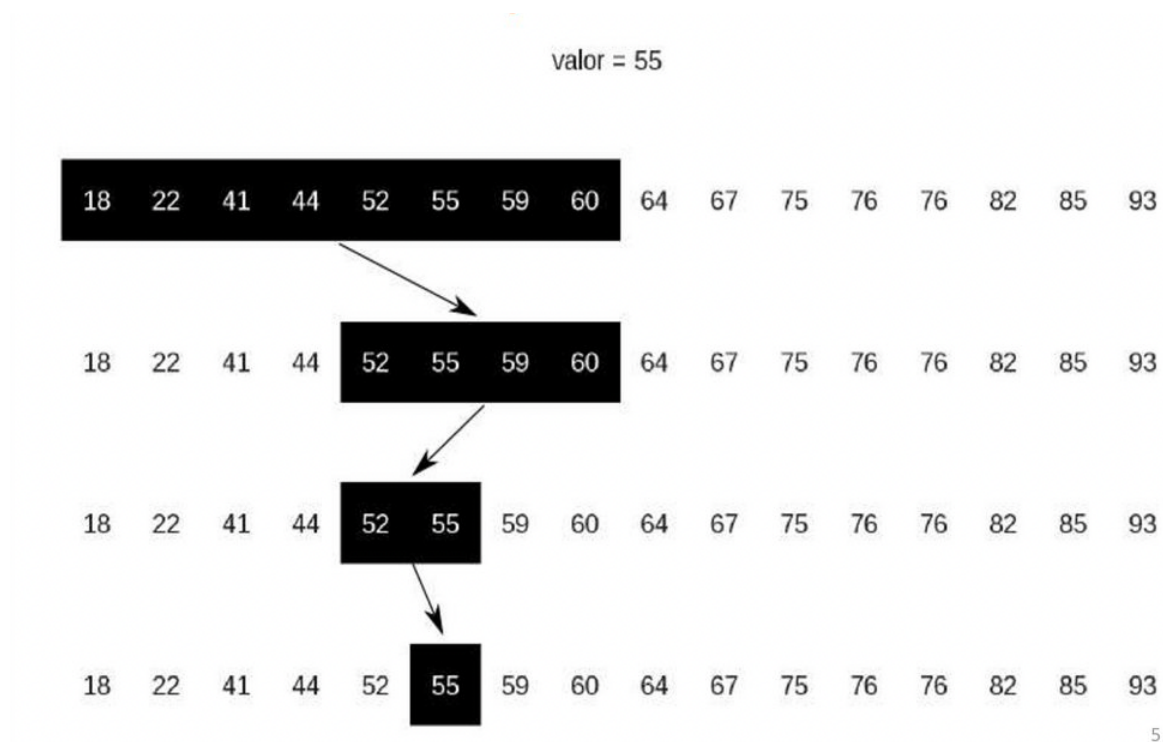


Рис. 1.1: Бинарный поиск

1.3 Алгоритм частотного анализа

Алгоритм частотного анализа строит частотный анализ полученного словаря. Чтобы провести частотный анализ, нужно взять первый элемент каждого значения в словаре по ключу и подсчитать частотную характеристику, т.е. сколько раз этот элемент встречался в качестве первого. По полученным данным словарь разбивается на сегменты так, что все записи с одинаковым первым элементом оказываются в одном сегменте.

Сегменты упорядочиваются по значению частотной характеристики таким образом, чтобы к элементу с наибольшим значением характеристики был предоставлен самый быстрый доступ.

Затем каждый из сегментов упорядочивается по значению. Это необходимо для реализации бинарного поиска, который обеспечит эффективный поиск в сегментах при сложности $O(n \log(n))$

таким образом, сначала выбирается нужный сегмент, а затем в нем проводится бинарный поиск нужного элемента. Средняя трудоемкость при длине алфавита M может быть рассчитана по формуле (1.2).

$$\sum_{i \in [1, M]} (f_{select_i} + f_{search_i}) \quad (1.2)$$

Вывод

В данной работе стоит задача реализации поиска в словаре. Входными данными будет являться словарь записей вида: $\{car : string, description : string\}$, а также ключ для поиска в этом словаре key типа $string$. Выходными данными будет являться найденная запись для каждого из реализуемых алгоритмов. В связи с ограничениями накладываемыми на ПО, тип ключа и значения входного словаря должен быть $string$.

2 Конструкторская часть

В данном разделе будут рассмотрены схемы вышеизложенных алгоритмов, описаны способы тестирования и определены структуры данных.

2.1 Разработка алгоритмов



Рис. 2.1: Схема алгоритма поиска полным перебором

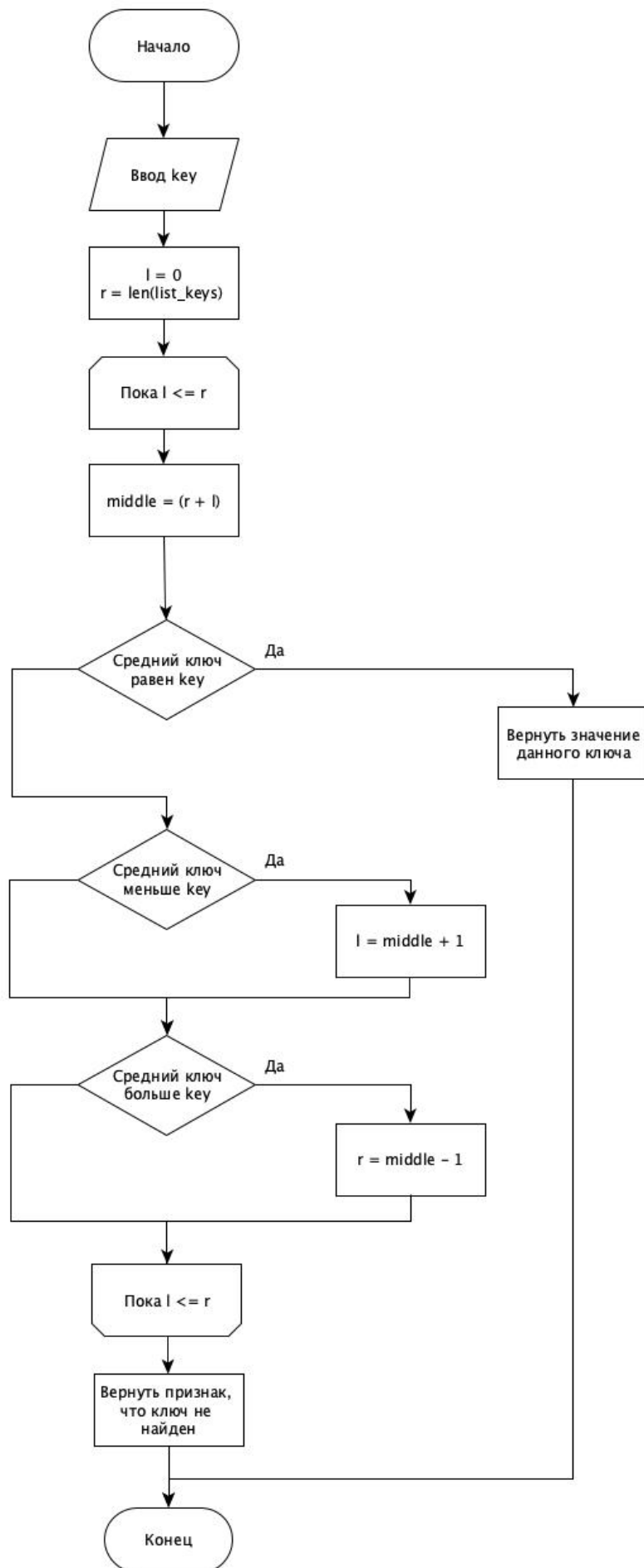


Рис. 2.2: Схема алгоритма с бинарным поиском

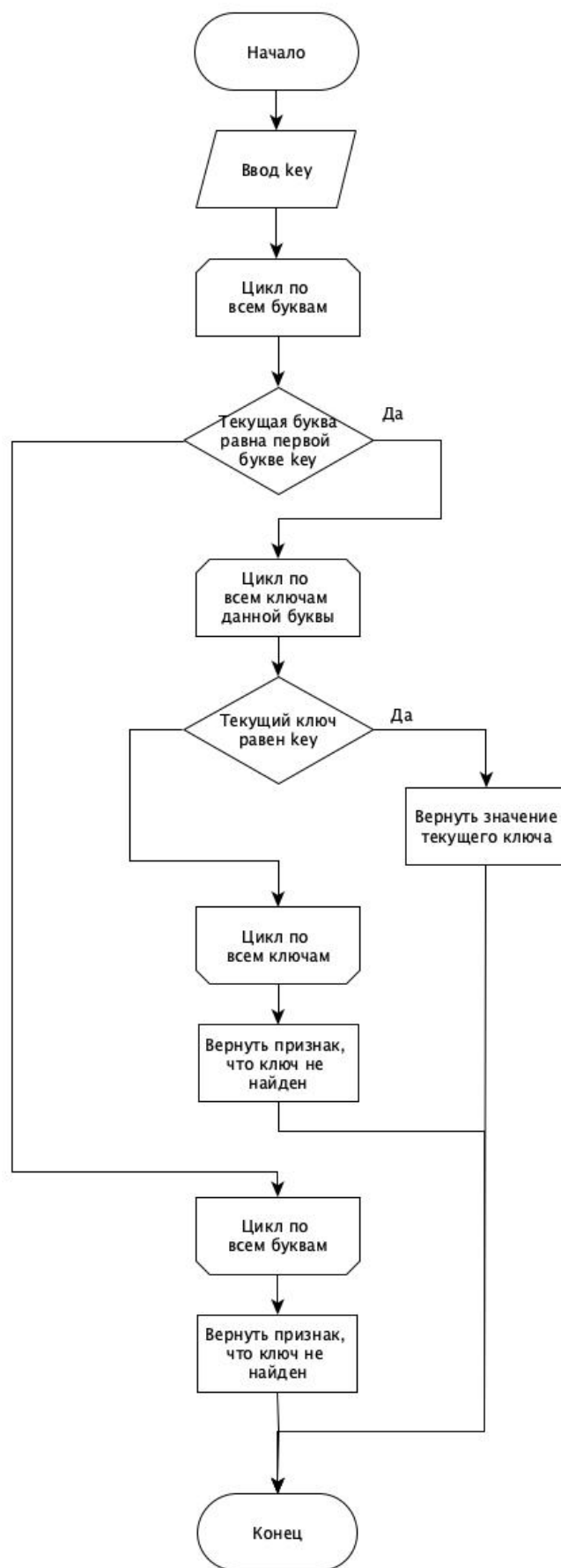


Рис. 2.3: Схема алгоритма с частотным анализом

2.2 Структура ПО

На рисунке 2.4 представлена uml-диаграмма разрабатываемого ПО.

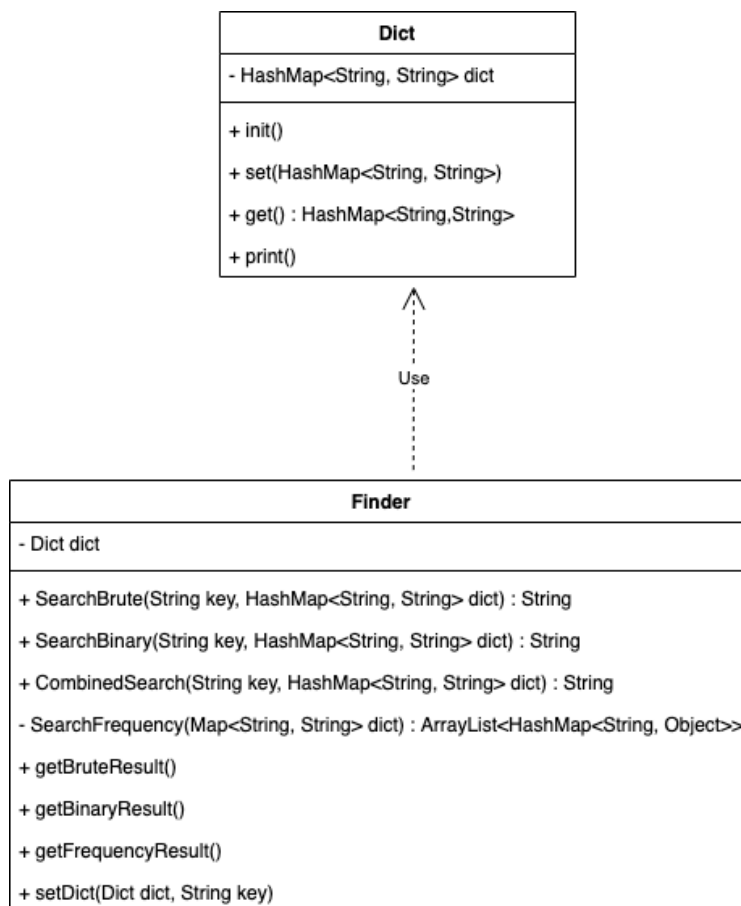


Рис. 2.4: Структура программного обеспечения

2.3 Описание структур данных

Описание используемых в программе типов данных:

- **Dict** - класс содержащий в себе хеш-таблицу с ключами и значениями типа *string*, а также методы для работы над этой таблицей;
- **ArrayList<HashMap<String, Object>** - тип данных, описывающий результат частотного анализа;
- **HashMap<String, String>** - тип данных, описывающий ассоциативный массив с ключами типа *string*.

2.4 Тестирование

В рамках данной лабораторной работы можно выделить следующие классы эквивалентности:

- входными данными является ключ и пустой словарь;
- входными данными является непустой словарь и ключ, который не существует в словаре;
- входными данными является непустой словарь и ключ, значение для которого в словаре определено;

Вывод

На основе теоретических данных, полученных из аналитического раздела, были построены схемы трех алгоритмов поиска в ассоциативных массивах. Приведена структура ПО, описаны структуры данных и способы тестирования.

3 Технологическая часть

В данном разделе приведены средства реализации и листинг алгоритмов.

3.1 Средства реализации

Для разработки ПО был выбран язык Java, поскольку он предоставляет разработчику широкий спектр возможностей и позволяет разрабатывать кроссплатформенные приложения, а также предоставляет большой набор инструментов для работы с многопоточностью. В качестве среды разработки использовалась Visual Studio Code. [3]

3.2 Листинг кода

Листинг 3.1: Реализация алгоритма поиска полным перебором

```
1 package algos;
2
3 import java.util.Map;
4 import java.util.Objects;
5
6 public class Brute {
7     public static String searchBrute(String key, Map<String, String> dict) {
8         for (String dictKey : dict.keySet()) {
9             if (Objects.equals(key, dictKey)) {
10                 return dict.get(key);
11             }
12         }
13         return null;
14     }
15 }
```

Листинг 3.2: Реализация бинарного поиска

```
1 package algos;
2
3 import java.util.*;
4
5 public class Binary {
```

```

6 public static String searchBinary(String key, Map<String, String> dict) {
7     ArrayList<String> keys = new ArrayList<>(dict.keySet());
8     Collections.sort(keys);
9
10    int l = 0, r = keys.size() - 1;
11    int m;
12    while (l <= r) {
13        m = (l + (r - l) / 2);
14
15        if (key.equals(keys.get(m))) {
16            return dict.get(key);
17        }
18        else if (key.compareTo(keys.get(m)) > 0) {
19            l = m + 1;
20        } else {
21            r = m - 1;
22        }
23    }
24    return null;
25 }
26 }

```

Листинг 3.3: Реализация поиска с использованием частотного анализа

```

1 package algos;
2
3 import java.util.*;
4
5 public class Frequency {
6     public static ArrayList<HashMap<String, Object>> searchFrequency(Map<String, String>
7         dict) {
8         HashMap<Character, Integer> frequencyDict = new HashMap<>();
9
10        for (String dict_key : dict.keySet()) {
11            if (frequencyDict.keySet().contains(dict_key.charAt(0))) {
12                frequencyDict.put(dict_key.charAt(0),
13                    frequencyDict.get(dict_key.charAt(0)) + 1);
14            } else {
15                frequencyDict.put(dict_key.charAt(0), 1);
16            }
17        }
18
19        ArrayList<HashMap<String, Object>> result = new ArrayList<>();
20
21        for (Character l : frequencyDict.keySet()) {
22            HashMap<String, Object> item = new HashMap<>();
23            item.put("letter", l);
24            item.put("count", frequencyDict.get(l));
25            item.put("dict", new HashMap<String, String>());
26        }
27    }
28 }

```

```

24
25     for (String dict_key : dict.keySet()) {
26         if (dict_key.charAt(0) == 1) {
27             ((HashMap<String, String>) item.get("dict")).put(dict_key,
28                 dict.get(dict_key));
29         }
30     }
31
32     List list = new ArrayList(((HashMap<String, String>)
33         item.get("dict")).entrySet());
34     Collections.sort(list, (Comparator<Map.Entry<String, String>>) (a, b) ->
35         a.getValue().compareTo(b.getValue()));
36     LinkedHashMap<String, String> sortedDict = new LinkedHashMap<>();
37     for (Object entry : list) {
38         sortedDict.put(((Map.Entry) entry).getKey().toString(), ((Map.Entry)
39             entry).getValue().toString());
40     }
41     item.put("dict", sortedDict);
42
43     result.add(item);
44 }
45
46 public static String combinedSearch(String key, ArrayList<HashMap<String, Object>>
47     segmentList) {
48     HashMap<String, String> dict = new HashMap<>();
49
50     for (int i = 0; i < segmentList.size(); i++) {
51         if (segmentList.get(i).get("letter").equals(key.charAt(0))) {
52             dict = (HashMap<String, String>) segmentList.get(i).get("dict");
53         }
54     }
55
56     if (dict.size() == 0) {
57         return null;
58     }
59
60     return Binary.searchBinary(key, dict);
61 }

```

3.3 Тестирование функций

В таблице 3.1 приведены тесты для функций, реализующих алгоритмы поиска в словаре. Тесты пройдены успешно.

Ключ	Словарь	Ожидание	Результат
abc	{car: "abc", desc: "small"}	"small"	"small"
cde	{car: "bva", desc: "big"}	NOT_FOUND	NOT_FOUND
vvv	{}	NOT_FOUND	NOT_FOUND

Таблица 3.1: Тестирование функций.

Вывод

Правильный выбор инструментов разработки позволил эффективно реализовать алгоритмы и выполнить исследовательский раздел лабораторной работы.

4 Исследовательская часть

В данном разделе будет произведено сравнение временных характеристик каждого из реализованных алгоритмов, а также будет приведена демонстрация работы программы.

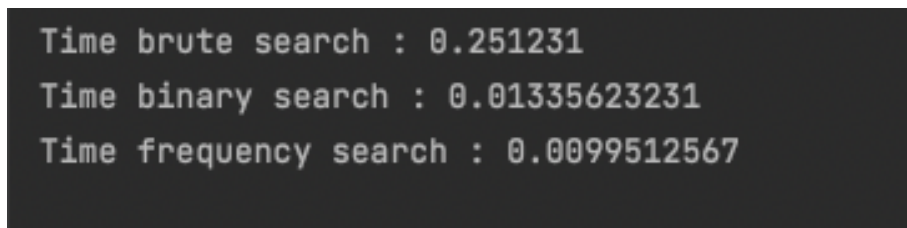
4.1 Технические характеристики

- Операционная система: Windows 10. [4]
- Память: 16 GiB.
- Процессор: Intel(R) Core(TM) i7-4700HQ CPU @ 2.40GHz. [5]

4.2 Временные характеристики

Так как поиск в словаре считается короткой задачей, воспользуемся усреднением массового эксперимента. Для этого сложим результат работы алгоритма n раз ($n \geq 10$), после чего поделим на n . Тем самым получим достаточно точные характеристики времени. Сравнение произведем при $n = 1000$.

На рис. 4.1 приведено сравнение времени выполнения трех алгоритмов для поиска последнего значения. По результатам эксперимента видно, что поиск полным перебором затрачивает больше всего времени, т.к. он последовательно обходит все элементы, в то время, как остальные два алгоритма выполняют поиск значительно быстрее.



```
Time brute search : 0.251231
Time binary search : 0.01335623231
Time frequency search : 0.0099512567
```

Рис. 4.1: Последний ключ

На рис. 4.2 произведено аналогичное сравнение, только в качестве искомого ключа взят несуществующий. Аналогично поиск полным перебором затрачивает больше всего времени по вышеописанной причине.

```
Time brute search : 0.274931
Time binary search : 0.01498711211
Time frequency search : 0.009968757
```

Рис. 4.2: Несуществующий ключ

Однако не всегда поиск полным перебором дает худший результат. На рис. 4.3 представлен результат поиска первого значения. Выигрыш алгоритма поиска полным перебором обосновывается тем, что он тратит лишь одно сравнение для того, чтобы найти первый ключ, в то время, когда бинарный поиск затрачивает гораздо больше сравнений. Частичный анализ работает чуть медленнее, так как ему нужно произвести дополнительное сравнение первых букв.

```
Time brute search : 0.0053117561
Time binary search : 0.01816903671
Time frequency search : 0.00144906567
```

Рис. 4.3: Первый ключ

На рис. 4.4 представлен результат поиска произвольного ключа. Алгоритм полного перебора работает медленнее всех.

```
Time brute search : 0.075416562
Time binary search : 0.01437134671
Time frequency search : 0.0144909737
```

Рис. 4.4: Произвольный ключ

Вывод

Исходя из полученных данных, можно сделать вывод, что алгоритм поиска в словаре, использующий частотный анализ, является более эффективным, чем алгоритм полного перебора лишь в ряде случаев, в остальных же, он является менее эффективным, в связи с использованием сегментации и бинарным поиском внутри сегмента.

Отдельно отметим, что алгоритм бинарного поиска требует, в целом, меньшего числа сравнений, в связи с чем является более эффективным, чем алгоритм с частотным анализом. Однако, алгоритм бинарного поиска требует сортировки всего входного массива, что, в среднем случае имеет сложность $O(n \log(n))$, в связи с чем алгоритм бинарного поиска становится менее эффективным, чем алгоритм частотного анализа, сортирующий данные по сегментам.

Заключение

В данной лабораторной работе были изучены алгоритмы поиска по ассоциативному словарю.

Среди рассмотренных алгоритмов наиболее эффективным по времени является алгоритм бинарного поиска.

В связи с вышеуказанным, алгоритм бинарного поиска является более предпочтительным для использования. Однако, при больших размерностях входных массивов алгоритм бинарного поиска становится менее эффективным, чем алгоритм, использующий частотный анализ. Поэтому, при большом числе элементов входного массива стоит использовать данный алгоритм.

В рамках выполнения данной лабораторной работы были достигнуты следующие цели:

- исследованы основные алгоритмы поиска по словарю;
- приведены схемы рассматриваемых алгоритмов;
- описаны используемые структуры данных;
- описана структура разрабатываемого программного обеспечения;
- определены средства программной реализации;
- определены требования к программному обеспечению;
- приведены сведения о модулях программы;
- проведены тестирование реализованного программного обеспечения;
- проведены экспериментальные замеры временных характеристик реализованных алгоритмов.

Список литературы

- [1] Cormen T. H. Introduction to Algorithms // MIT Press. 2001. p. 1292.
- [2] Коршунов Ю. М. Коршуном Ю. М. Математические основы кибернетики // Энергоатомиздат. 1972.
- [3] Documentation for Visual Studio Code [Электронный ресурс]. Режим доступа: <https://code.visualstudio.com/docs>.
- [4] Клиентская документация по Windows для ИТ-специалистов [Электронный ресурс]. Режим доступа: <https://docs.microsoft.com/ru-ru/windows/resources/>.
- [5] Процессор Intel® Core™ i7-4700HQ [Электронный ресурс]. Режим доступа: <https://ark.intel.com/content/www/ru/ru/ark/products/122589/intel-core-i7-8550u-processor-8m-cache-up-to-4-00-ghz.html>.