



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 Кононенко С.С.

Группа ИУ7-53Б

Оценка (баллы) _____

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

Оглавление

Введение	3
1 Аналитическая часть	4
1.1 Алгоритм полного перебора	4
1.2 Алгоритм двоичного поиска	4
1.3 Алгоритм частотного анализа	5
1.4 Описание словаря	6
2 Конструкторская часть	7
2.1 Разработка алгоритмов	7
3 Технологическая часть	11
3.1 Требования к ПО	11
3.2 Средства реализации	11
3.3 Листинг кода	11
3.4 Тестирование функций	16
4 Исследовательская часть	17
4.1 Пример работы	17
4.2 Технические характеристики	18
4.3 Время выполнения алгоритмов	18
Заключение	22
Литература	23

Введение

Целью данной лабораторной работы является изучение способа эффективного по времени и по памяти поиска по словарю.

Словарь, или ассоциативный массив, – абстрактный тип данных (интерфейс к хранилищу данных), позволяющий хранить пары вида «(ключ, значение)» и поддерживающий операции добавления пары, а также поиска и удаления пары по ключу.

В паре (k, v) значение v называется значением, ассоциированным с ключом k . Где k – это *key*, а v – *value*. Семантика и названия вышеупомянутых операций в разных реализациях ассоциативного массива могут отличаться.

Ассоциативный массив с точки зрения интерфейса удобно рассматривать как обычный массив, в котором в качестве индексов можно использовать не только целые числа, но и значения других типов – например, строки.

Поддержка ассоциативных массивов есть во многих интерпретируемых языках программирования высокого уровня, таких, как Perl, PHP, Python, Ruby, Tcl, JavaScript и других. Для языков, которые не имеют встроенных средств работы с ассоциативными массивами, существует множество реализаций в виде библиотек.

Задачи лабораторной работы:

- рассмотреть и изучить алгоритмы полного перебора, двоичного поиска и эффективного поиска по словарю;
- сравнить временные характеристики каждого из рассмотренных алгоритмов;
- на основании проделанной работы сделать выводы.

1 Аналитическая часть

В данном разделе представлены теоретические сведения о рассматриваемых алгоритмах.

1.1 Алгоритм полного перебора

Алгоритмом полного перебора называют метод решения задачи, при котором по очереди рассматриваются все возможные варианты исходного набора данных. В случае словарей будет произведен последовательный перебор элементов словаря до тех пор, пока не будет найден необходимый. Сложность такого алгоритма зависит от количества всех возможных решений, а время решения может стремиться к экспоненциальному времени работы.

Пусть алгоритм нашёл элемент на первом сравнении, тогда, в лучшем случае, будет затрачено $k_0 + k_1$ операций, на втором – $k_0 + 2k_1$, на последнем – $k_0 + Nk_1$. Тогда средняя трудоёмкость может быть рассчитана по формуле 1.1, где Ω – множество всех возможных случаев.

$$\begin{aligned}\sum_{i \in \Omega} p_i \cdot f_i &= (k_0 + k_1) \frac{1}{N+1} + (k_0 + 2k_1) \frac{1}{N+1} + \\ &+ (k_0 + 3k_1) \frac{1}{N+1} + (k_0 + Nk_1) \frac{1}{N+1} + (k_0 + Nk_1) \frac{1}{N+1} = \\ &= k_0 \frac{N+1}{N+1} + k_1 + \frac{1+2+\dots+N+N}{N+1} = \\ &= k_0 + k_1 \left(\frac{N}{N+1} + \frac{N}{2} \right) = k_0 + k_1 \left(1 + \frac{N}{2} - \frac{1}{N+1} \right)\end{aligned}\tag{1.1}$$

1.2 Алгоритм двоичного поиска

Алгоритм двоичного поиска применяется к заранее упорядоченному словарю. Процесс двоичного поиска можно описать при помощи шагов:

- 1) получить значение ключа, находящееся в середине словаря, и сравнить его с данным,
- 2) в случае, если значение меньше (в контексте типа данных) данного, продолжить поиск в левой части словаря, в обратном случае – в правой,
- 3) на новом интервале получить значение ключа из середины этого интервала и сравнить его с данным,
- 4) продолжать поиск до тех пор, пока найденное значение ключа не будет равно данному.

Поиск в словаре с использованием данного алгоритма в худшем случае будет иметь трудоемкость $O(\log_2 N)$, что быстрее поиска при помощи алгоритма полного перебора. Но стоит учитывать тот факт, что данный алгоритм работает только для заранее упорядоченного словаря. В случае большого объема данных и обратного порядка сортировки может произойти так, что алгоритм полного перебора будет эффективнее по времени, чем алгоритм двоичного поиска.

1.3 Алгоритм частотного анализа

Алгоритм частотного анализа на вход получает словарь и на его основе составляется частотный анализ. Чтобы провести частотный анализ нужно взять первый элемент каждого значения в словаре по ключу и подсчитать частотную характеристику, т.е. сколько раз этот элемент встречается в качестве первого элемента. По полученным значениям словарь разбивается на сегменты так, что все элементы с одинаковым первым элементом оказываются в одном сегменте.

Сегменты упорядочиваются по значению частотной характеристики так, чтобы к элементы с наибольшей частотной характеристикой был самый быстрый доступ.

Далее каждый сегмент упорядочивается по значению. Это необходимо для реализации бинарного поиска, который обеспечит эффективный поиск в сегменте при сложности $O(n \log n)$

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

$$\sum_{i \in [1, M]} (f_{\text{выбор } i\text{-го сегмента}} + f_{\text{поиск в } i\text{-ом сегменте}}) \cdot p_i \quad (1.2)$$

1.4 Описание словаря

Словарь, реализованный в данной работе, имеет вид $\{id : number, gamertag : string\}$, что представляет собой базу данных о игроках в компьютерные игры. Поиск в работе будет реализован по полю *gamertag*.

Вывод

В данной работе стоит задача реализации поиска в словаре. Были рассмотрены алгоритмы реализации поиска.

2 Конструкторская часть

В данном разделе представлены схемы рассматриваемых алгоритмов.

2.1 Разработка алгоритмов

На рисунках 2.1, 2.2 и 2.3 приведены схемы алгоритмов поиска в словаре перебором, двоичным поиском и частотным анализом соответственно.

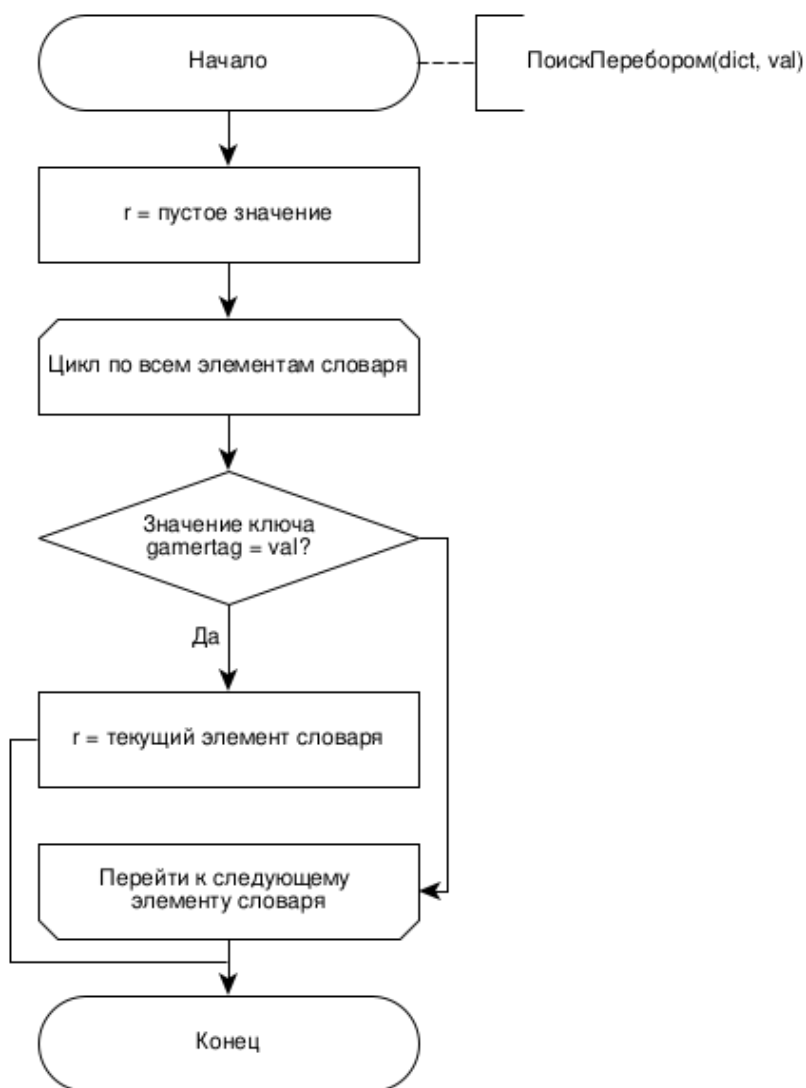


Рисунок 2.1 – Схема алгоритма поиска с использованием перебора

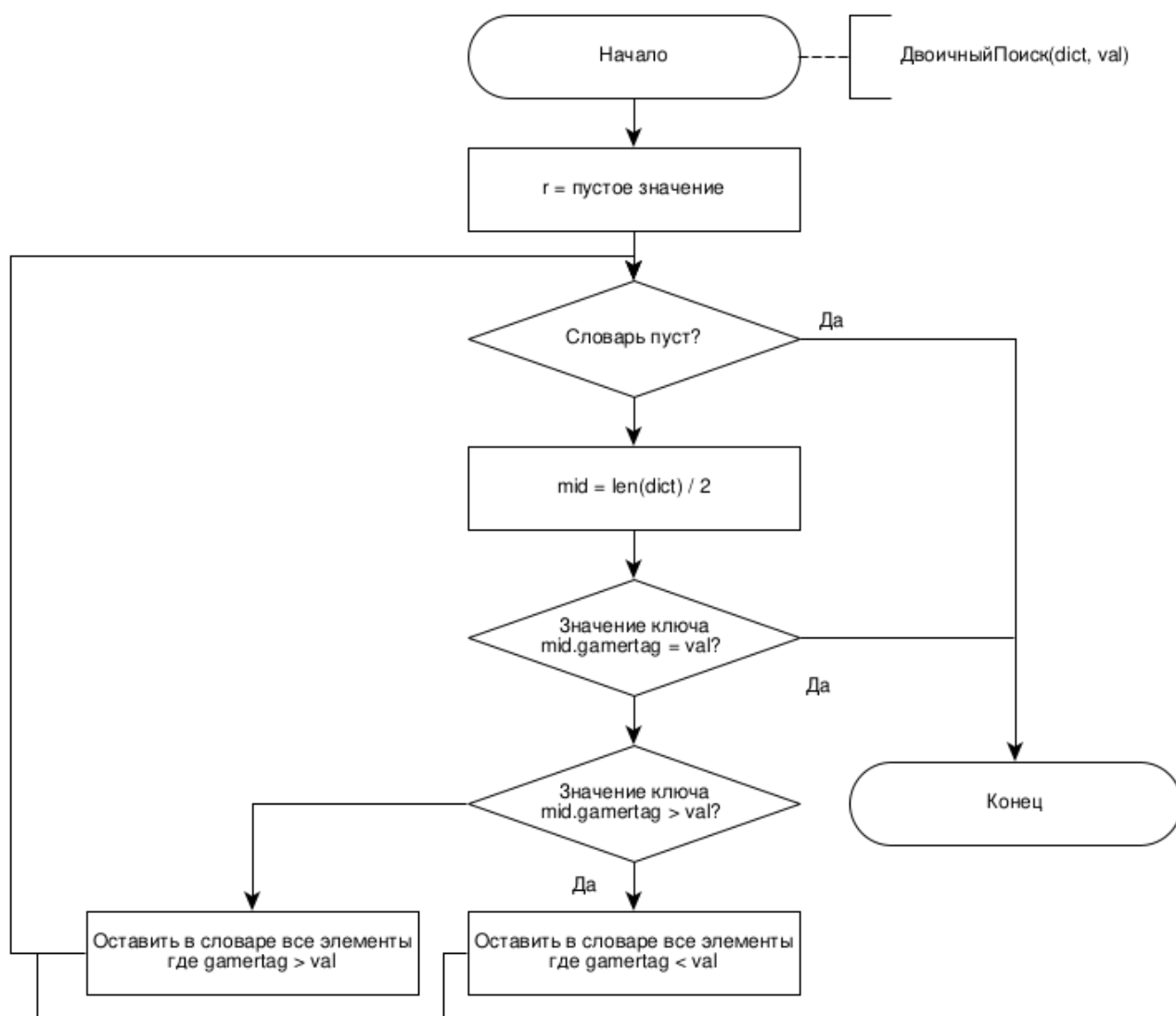


Рисунок 2.2 – Схема алгоритма поиска с использованием двоичного поиска

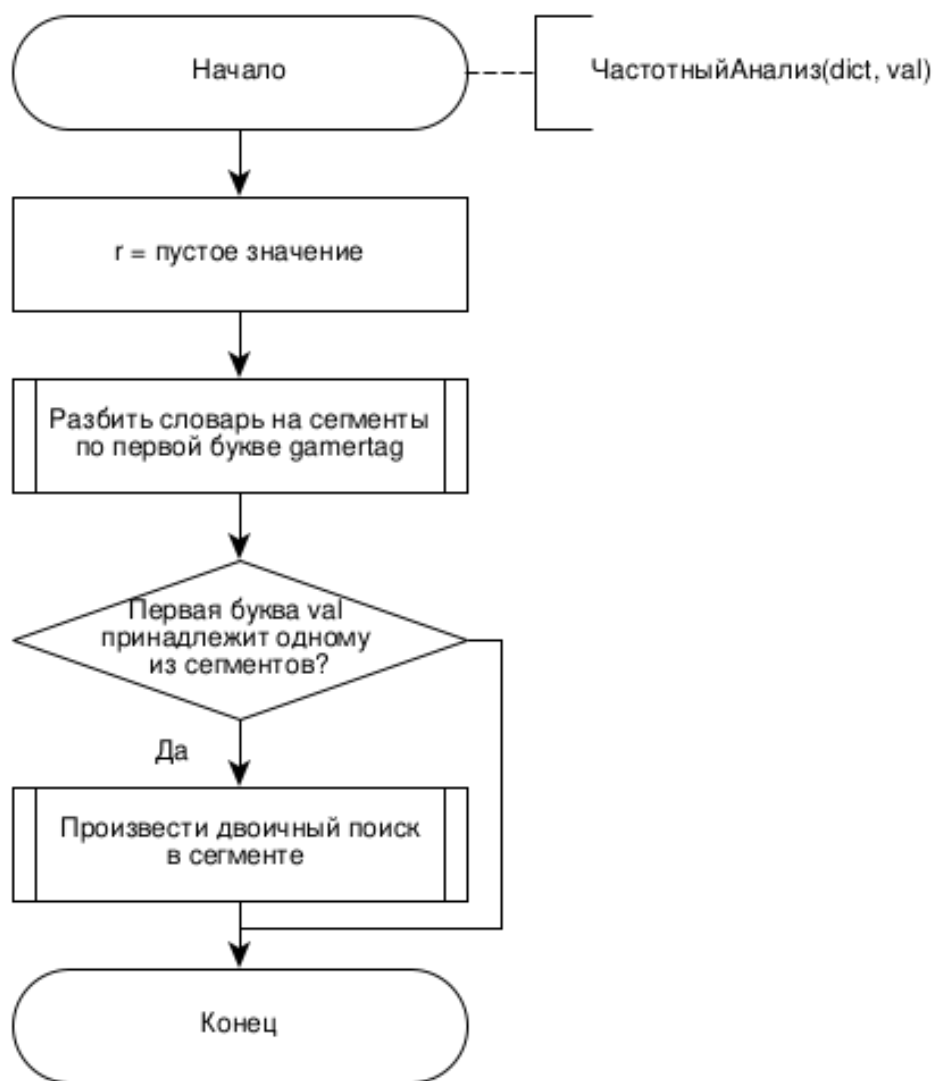


Рисунок 2.3 – Схема алгоритма поиска с использованием частотного анализа

Вывод

На основе теоретических данных, полученных из аналитического раздела, были построены схемы требуемых алгоритмов.

3 Технологическая часть

В данном разделе приведены требования к программному обеспечению, средства реализации и листинги кода.

3.1 Требования к ПО

К программе предъявляется ряд требований:

- на вход подаётся словарь и значение ключа *gamertag*;
- на выходе — элемент словаря, значение ключа *gamertag* которого равно введенному.

3.2 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран многопоточный язык Golang [1]. Данный выбор обусловлен моим желанием расширить свои знания в области применения данного языка. Помимо этого, встроенные средства языка предоставляют высокоточные средства тестирования разработанного ПО.

3.3 Листинг кода

В листингах 3.1 и 3.2 приведены реализации алгоритмов поиска в словаре и дополнительные типы данных соответственно.

Листинг 3.1 – Реализация алгоритмов поиска в словаре

```

1 package dict
2
3 import (
4     "fmt"
5     "math/rand"
6     "reflect"
7     "sort"
8
9     "github.com/brianvoe/gofakeit"
10 )
11
12 // CreateArray used to create DictArray with given size.
13 func CreateArray(n int) DictArray {
14     var (
15         darr DictArray
16         g Dict
17     )
18
19     darr = make(DictArray, n)
20
21     for i := 0; i < n; i++ {
22         dup := true
23         for dup != false {
24             g = Dict{
25                 "id": gofakeit.Uint8(),
26                 "gamertag": gofakeit.Gamertag(),
27             }
28             dup = g.IsDup(darr[:i])
29         }
30
31         darr[i] = g
32     }
33
34     return darr
35 }
36
37 // IsDup used to check whether Dict presents in given DictArray.
38 func (d Dict) IsDup(darr DictArray) bool {
39     for _, v := range darr {
40         if reflect.DeepEqual(d, v) {
41             return true
42         }
43     }
44     return false
45 }
46
47 // Print used to print single Dict.

```

```

48 func (d Dict) Print() {
49     fmt.Printf("ID:_%v\nGamertag:_%v\n", d["id"], d["gamertag"])
50 }
51
52 // Print used to print single DictArray.
53 func (darr DictArray) Print() {
54     for _, d := range darr {
55         d.Print()
56     }
57 }
58
59 // Pick used to get gamertag with first letter.
60 func (darr DictArray) Pick(l string) string {
61     for _, d := range darr {
62         if d["gamertag"].(string)[:1] == l {
63             return d["gamertag"].(string)
64         }
65     }
66
67     i := rand.Int() % len(darr)
68
69     return darr[i]["gamertag"].(string)
70 }
71
72 // Brute used to find value using brute force method.
73 func (darr DictArray) Brute(gt string) Dict {
74     for _, d := range darr {
75         if d["gamertag"] == gt {
76             return d
77         }
78     }
79
80     return darr[0]
81 }
82
83 // Binary used to find value using binary search method.
84 func (darr DictArray) Binary(gt string) Dict {
85     var (
86         l int = len(darr)
87         mid int = l / 2
88         r Dict
89     )
90
91     switch {
92     case l == 0:
93         return r
94     case darr[mid]["gamertag"].(string) > gt:
95         r = darr[:mid].Binary(gt)

```

```

96     case darr[mid]["gamertag"].(string) < gt:
97         r = darr[mid+1:].Binary(gt)
98     default:
99         r = darr[mid]
100 }
101
102 return r
103 }
104
105 // FAnalysis used to analyse frequency of given DictArray.
106 func (darr DictArray) FAnalysis() FreqArray {
107     var (
108         az string = "abcdefghijklmnopqrstuvwxyz"
109         farr FreqArray = make(FreqArray, len(az))
110     )
111
112     for i, v := range az {
113         a := Freq{
114             l: string(v),
115             cnt: 0,
116             darr: make(DictArray, 0),
117         }
118         farr[i] = a
119     }
120
121     for _, v := range darr {
122         l := v["gamertag"].(string)[:1]
123         for i := range farr {
124             if farr[i].l == l {
125                 farr[i].cnt++
126             }
127         }
128     }
129
130     sort.Slice(farr, func(i, j int) bool {
131         return farr[i].cnt > farr[j].cnt
132     })
133
134     for i := range farr {
135         for j := range darr {
136             if darr[j]["gamertag"].(string)[:1] == farr[i].l {
137                 farr[i].darr = append(farr[i].darr, darr[j])
138             }
139         }
140
141         sort.Slice(farr[i].darr, func(l, m int) bool {
142             return farr[i].darr[l]["gamertag"].(string) <
143                 farr[i].darr[m]["gamertag"].(string)

```

```

143     })
144 }
145
146 return farr
147 }
148
149 // Combined used to find value using binary search and frequency analysis method.
150 func (farr FreqArray) Combined(w string) Dict {
151     var (
152         l string = w[:1]
153         r Dict = farr[0].darr[0]
154     )
155
156     for _, d := range farr {
157         if string(d.l) == l {
158             r = d.darr.Binary(w)
159         }
160     }
161
162     return r
163 }

```

Листинг 3.2 – Реализация дополнительных типов данных

```

1 package dict
2
3 // Dict used to represent dictionary with custom types.
4 type Dict map[string]interface{}
5
6 // DictArray used to represent array of Dict instances.
7 type DictArray []Dict
8
9 // Freq used to represent frequency analyser type.
10 type Freq struct {
11     l string
12     cnt int
13     darr DictArray
14 }
15
16 // FreqArray used to represent array of Freq instances.
17 type FreqArray []Freq

```

3.4 Тестирование функций

В таблице 3.1 приведены функциональные тесты для функций, реализующих алгоритмы поиска в словаре. Тестирование проводилось на словаре, имеющем следующую структуру:

```
[{ id: 1, gamertag: "volkovallthebest"}, { id: 2, gamertag: "primitelabypls"}]
```

Все тесты пройдены успешно.

Таблица 3.1 – Тестирование функций

Ключ	Результат	Ожидаемый результат
volkovallthebest	(1)	{ id: 1, gamertag: "volkovallthebest"} (1)
primitelabypls	(2)	{ id: 2, gamertag: "primitelabypls"} (2)
zachetPoAA	пустой словарь	пустой словарь

Вывод

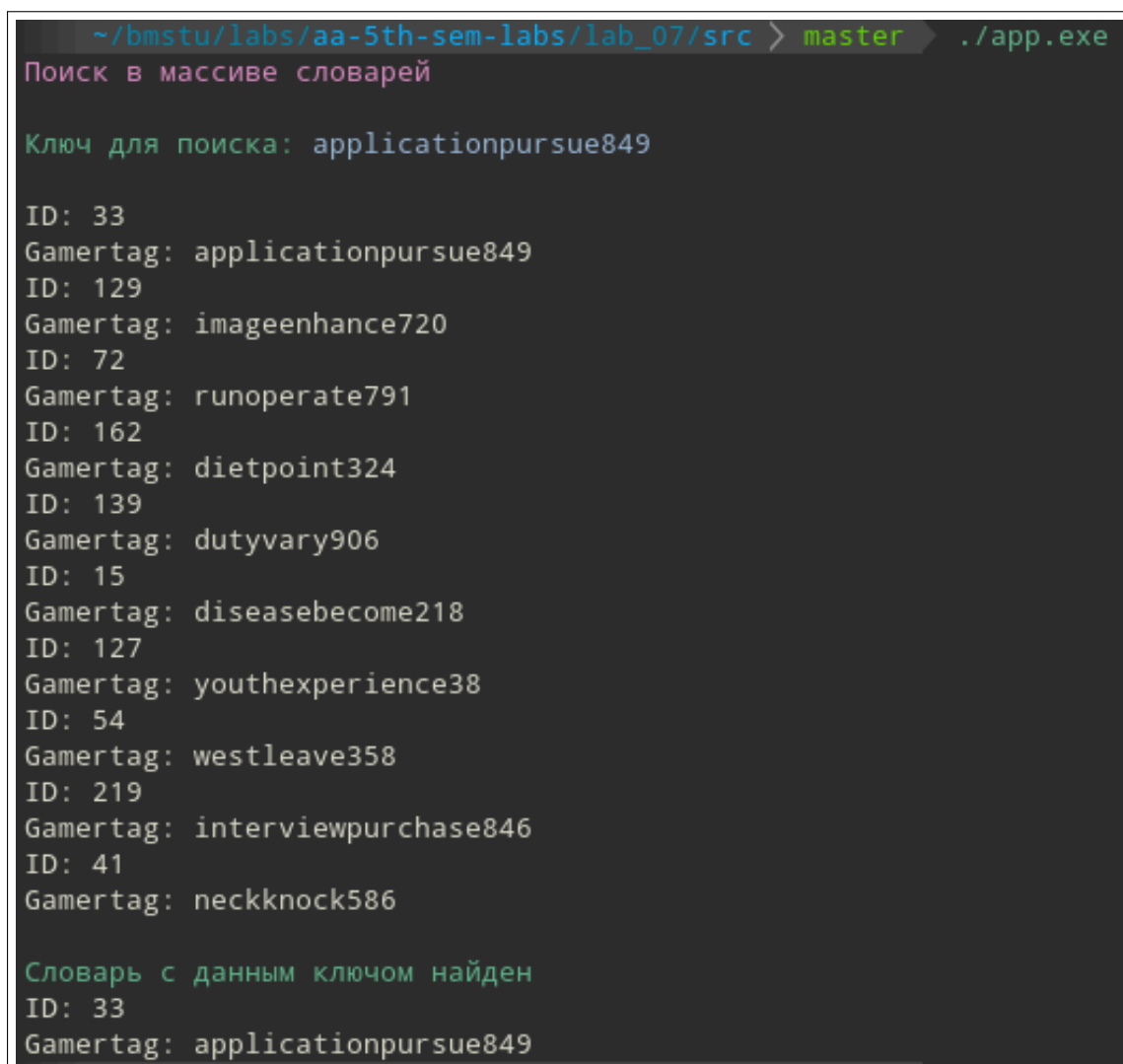
Были разработаны реализации алгоритмов поиска в словаре: с использованием перебора, с использованием двоичного поиска и с использованием частотного анализа.

4 Исследовательская часть

В данном разделе приведены примеры работы и анализ характеристик разработанного программного обеспечения.

4.1 Пример работы

Демонстрация работы программы приведена на рисунке 4.1.



```
~/bmstu/labs/aa-5th-sem-labs/lab_07/src > master > ./app.exe
Поиск в массиве словарей

Ключ для поиска: applicationpursue849

ID: 33
Gamertag: applicationpursue849
ID: 129
Gamertag: imageenhance720
ID: 72
Gamertag: runoperate791
ID: 162
Gamertag: dietpoint324
ID: 139
Gamertag: dutyvary906
ID: 15
Gamertag: diseasebecome218
ID: 127
Gamertag: youthexperience38
ID: 54
Gamertag: westleave358
ID: 219
Gamertag: interviewpurchase846
ID: 41
Gamertag: neckknock586

Словарь с данным ключом найден
ID: 33
Gamertag: applicationpursue849
```

Рисунок 4.1 – Демонстрация работы алгоритмов поиска в словаре

4.2 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- Операционная система: Manjaro [2] Linux [3] 20.1 64-битная.
- Память: 16 ГБ.
- Процессор: AMD Ryzen™ 7 3700U [4] ЦПУ @ 2.30ГГц

Тестирование проводилось на ноутбуке, включенном в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения рабочего стола, окружением рабочего стола, а также непосредственно системой тестирования.

4.3 Время выполнения алгоритмов

Алгоритмы тестировались при помощи написания «бенчмарков» [5], предоставляемых встроенными в Go средствами. Для такого рода тестирования не нужно самостоятельно указывать количество повторов. В библиотеке для тестирования существует константа N , которая динамически варьируется в зависимости от того, был ли получен стабильный результат или нет.

В листинге 4.1 пример реализации бенчмарка.

Листинг 4.1 – Реализация бенчмарка

```
1 package dict
2
3 import (
4     "sort"
5     "testing"
6 )
7
8 var darr = CreateArray(10000)
9 var farr = darr.FAnalysis()
10
11 // Brute benchmarks.
12 func BenchmarkBruteA(b *testing.B) {
```

```

13 w := darr.Pick("a")
14 for i := 0; i < b.N; i++ {
15     darr.Brute(w)
16 }
17 }

```

На рисунке 4.2 показаны результаты работы бенчмарков.

```
~/bmstu/labs/aa-5th-sem-labs/lab_07/src/dict > master go test -bench .
```

goos: linux		
goarch: amd64		
BenchmarkBruteA-8	56454160	22.7 ns/op
BenchmarkBruteB-8	2898944	412 ns/op
BenchmarkBruteC-8	3492493	355 ns/op
BenchmarkBruteD-8	16899224	64.5 ns/op
BenchmarkBruteE-8	1567774	769 ns/op
BenchmarkBruteF-8	3106602	388 ns/op
BenchmarkBruteG-8	301039	3756 ns/op
BenchmarkBruteH-8	3296212	359 ns/op
BenchmarkBruteI-8	33984906	36.6 ns/op
BenchmarkBruteJ-8	402978	3017 ns/op
BenchmarkBruteK-8	315724	3325 ns/op
BenchmarkBruteL-8	505490	2228 ns/op

Рисунок 4.2 – Демонстрация работы бенчмарков

Результаты замеров приведены в таблице 4.1. На рисунке 4.3 приведены график, иллюстрирующий зависимость времени работы алгоритмов поиска в словаре от первой буквы слова.

Все замеры проводились на размере словаря равном 10000 элементов.

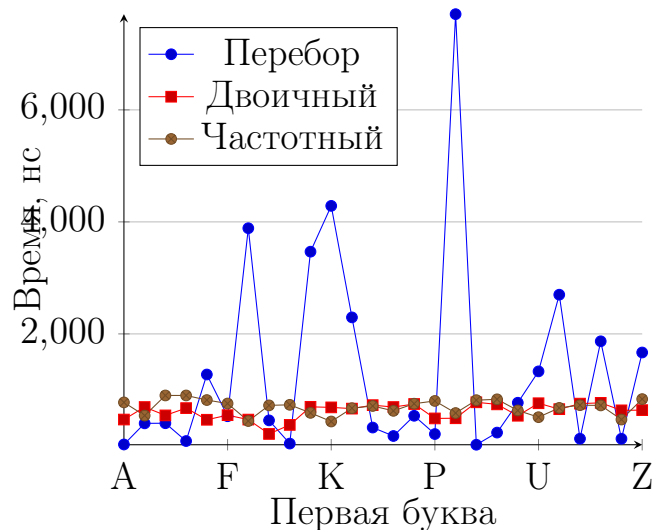


Рисунок 4.3 – Зависимость времени работы реализации алгоритмов поиска в словаре от первой буквы слова

Таблица 4.1 – Время работы реализации алгоритмов поиска в словаре

Первая буква	Время поиска, нс		
	Перебор	Двоичный	Частотный
A	26	471	777
B	403	694	542
C	405	545	901
D	87	675	901
E	1275	467	819
F	530	549	757
G	3887	470	444
H	456	214	724
I	41.7	375	734
J	3468	698	590
K	4286	688	436
L	2295	666	674
M	328	728	713
N	177	694	626
O	538	748	754
P	212	491	803
Q	7709	494	587
R	21	779	817
S	238	742	832
T	771	539	633
U	1331	762	513
V	2700	656	679
W	129	755	727
X	1868	769	723
Y	128	635	471
Z	1668	639	836

Вывод

Исходя из проведенных исследований, можно сделать вывод, что алгоритм поиска в словаре, использующий частотный анализ, выигрывает по скорости у алгоритма полного перебора в ряде случаев, а в ряде случаев проигрывает. Это связано с тем, что для алгоритма частотного анализа изначально необходимо провести частотный анализ, а уже после проводить поиск по сегментам. В случае, когда необходимое значение ключа находит-

ся в начале словаря, алгоритм полного перебора будет выигрывать как у алгоритма двоичного поиска, так и у алгоритма частотного анализа. По графику также видно, что алгоритмы двоичного поиска и частотного анализа имеют похожие результаты, но стоит учитывать, что в итоговое время не было включено время сортировки словаря для алгоритма бинарного поиска, поэтому можно сделать вывод, что алгоритм частотного анализа будет работать быстрее.

Заключение

В ходе выполнения работы была достигнута цель выполнены все поставленные задачи:

- рассмотреть и изучить алгоритмы полного перебора, двоичного поиска и эффективного поиска по словарю;
- сравнить временные характеристики каждого из рассмотренных алгоритмов;
- на основании проделанной работы сделать выводы.

Алгоритмы двоичного поиска и частотного анализа имеют похожие результаты, но стоит учитывать, что в итоговое время не было включено время сортировки словаря для алгоритма бинарного поиска, поэтому можно сделать вывод, что алгоритм частотного анализа будет работать быстрее.

Литература

- [1] The Go Programming Language [Электронный ресурс]. Режим доступа: <https://golang.org/> (дата обращения: 11.09.2020).
- [2] Manjaro – enjoy the simplicity [Электронный ресурс]. Режим доступа: <https://manjaro.org/> (дата обращения: 14.09.2020).
- [3] Linux – Википедия [Электронный ресурс]. Режим доступа: <https://ru.wikipedia.org/wiki/Linux> (дата обращения: 14.09.2020).
- [4] Мобильный процессор AMD Ryzen™ 7 3700U с графикой Radeon™ RX Vega 10 [Электронный ресурс]. Режим доступа: <https://www.amd.com/ru/products/apu/amd-ryzen-7-3700u> (дата обращения: 14.09.2020).
- [5] testing – The Go Programming Language [Электронный ресурс]. Режим доступа: <https://golang.org/pkg/testing/> (дата обращения: 12.09.2020).