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

Студент Топорков Павел
Группа ИУ7-53Б
Дисциплина Анализ алгоритмов

Преподаватели: _____ Строганов Ю.В., Волкова Л.Л.
подпись, дата Фамилия, И.О.

Оценка _____

Москва — 2021 г.

Оглавление

Введение

Словарь, как тип данных, применяется везде, где есть связь “ключ - значение” или “объект - данные”: поиск истории болезни пациента по номеру его амбулаторной карты, поиск налогов по ИНН и другое. Поиск - основная задача при использовании словаря. Данная задача решается различными способами, которые дают различную скорость решения.

Цель данной работы: получить навык работы со словарём, как структурой данных, реализовать алгоритмы поиска по словарю с применением оптимизаций.

В рамках выполнения работы необходимо решить следующие задачи:

- реализовать алгоритм поиска по словарю, использующий полный перебор;
- реализовать алгоритм поиска по словарю, использующий двоичный поиск;
- применить частотный анализ для эффективного поиска по словарю;
- сравнить полученные результаты;
- сделать выводы по проделанной работе.

1 Аналитическая часть

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:

- `ВСТАВКА(ключ, значение)`;
- `ПОИСК(ключ)`;
- `УДАЛЕНИЕ(ключ)`.

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

Операция `ПОИСК(ключ)` возвращает значение, ассоциированное с заданным ключом, или некоторый специальный объект `НЕ_НАЙДЕНО`, означающий, что значения, ассоциированного с заданным ключом, нет. Две другие операции ничего не возвращают (за исключением, возможно, информации о том, успешно ли была выполнена данная операция).

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.

1.1 Алгоритм полного перебора

Алгоритмом полного перебора [**?**] 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.

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

1.2 Алгоритм поиска в упорядоченном словаре двоичным поиском

При двоичном поиске [1] обход можно представить деревом, поэтому трудоёмкость в худшем случае составит $\log_2 N$ (в худшем случае нужно спуститься по двоичному дереву от корня до листа). Скорость роста функции $\log_2 N$ меньше, чем скорость линейной функции, полученной для полного перебора.

1.3 Частотный анализ

Алгоритм на вход получает словарь и на его основе составляется частотный анализ. По полученным значениям словарь разбивается на сегменты так, что все элементы с некоторым общим признаком попадают в один сегмент (для букв это может быть первая буква, для чисел - остаток

от деления).

Сегменты упорядочиваются по значению частотной характеристики так, чтобы к элементам с наибольшей частотной характеристикой был самый быстрый доступ. Такой характеристикой может послужить, например, размер сегмента. Вероятность обращения к определенному сегменту равна сумме вероятностей обращений к его ключам, то есть $P_i = \sum_j p_j = N \cdot p$, где P_i - вероятность обращения к i-ому сегменту, p_j - вероятность обращения к j-ому элементу, который принадлежит i-ому сегменту. Если обращения ко всем ключам равновероятны, то можно заменить сумму на произведение, где N - количество элементов в i-ом сегменте, а p - вероятность обращения к произвольному ключу.

Далее ключи в каждом сегменте упорядочиваются по значению. Это необходимо для реализации бинарного поиска, который обеспечит эффективный поиск со сложностью $O(\log_2 m)$ (где m - количество ключей в сегменте) внутри сегмента.

Таким образом, сначала выбирается нужный сегмент, а затем в нем проводится бинарный поиск нужного элемента. Средняя трудоёмкость при множестве всех возможных случаев Ω может быть рассчитана по формуле (1.2).

$$\sum_{i \in \Omega} (f_{\text{выбор сегмента i-ого элемента}} + f_{\text{бинарный поиск i-ого элемента}}) \cdot p_i \qquad (1.2)$$

Вывод

В данном разделе был рассмотрен абстрактный тип данных словарь и возможные реализации поиска в нём.

2 Конструкторская часть

2.1 Структура словаря

Словарь состоит из пар вида `<id - ФИ>`, где `id` - id игрока на официальном сайте лиги, Teamname - название команды.[2]

2.2 Схемы алгоритмов

На рисунке 2.1 представлена схема алгоритма поиска полным перебором, на рисунке 2.2 представлена схема поиска с использованием двоичного поиска, на рисунках 2.3 и 2.4 представлена схема поиска по сегментам, которые в результате частотного анализа (анализа длины) упорядочены в порядке убывания длины и отсортированы для возможности использования двоичного поиска внутри сегмента.

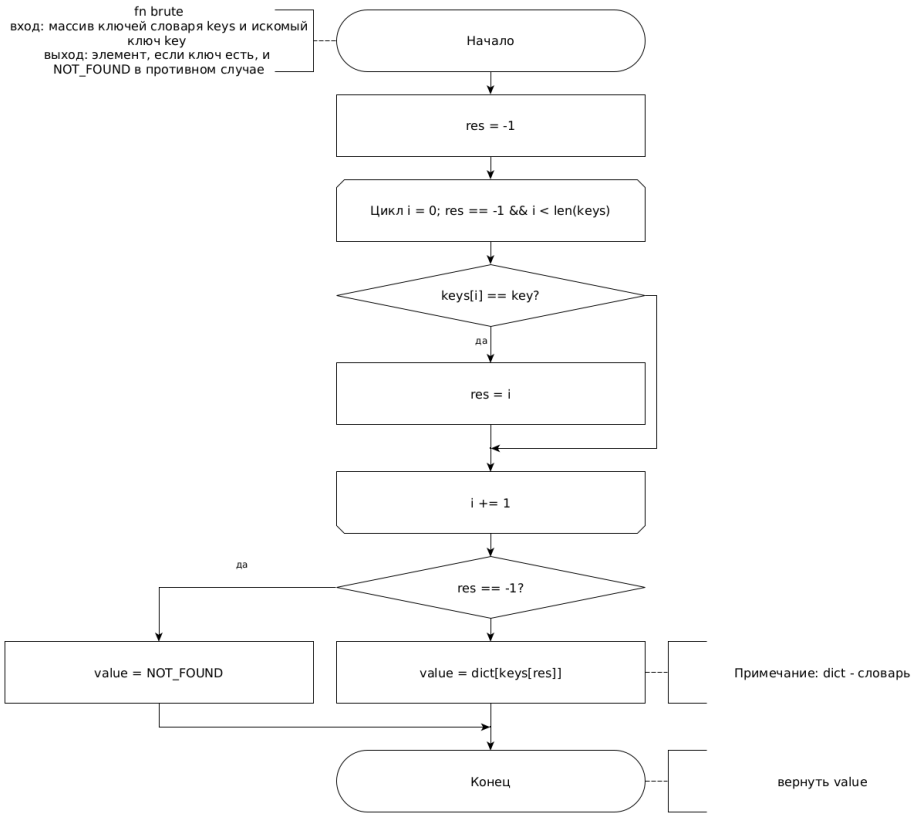

Рис. 2.1: Схема алгоритма поиска полным перебором.

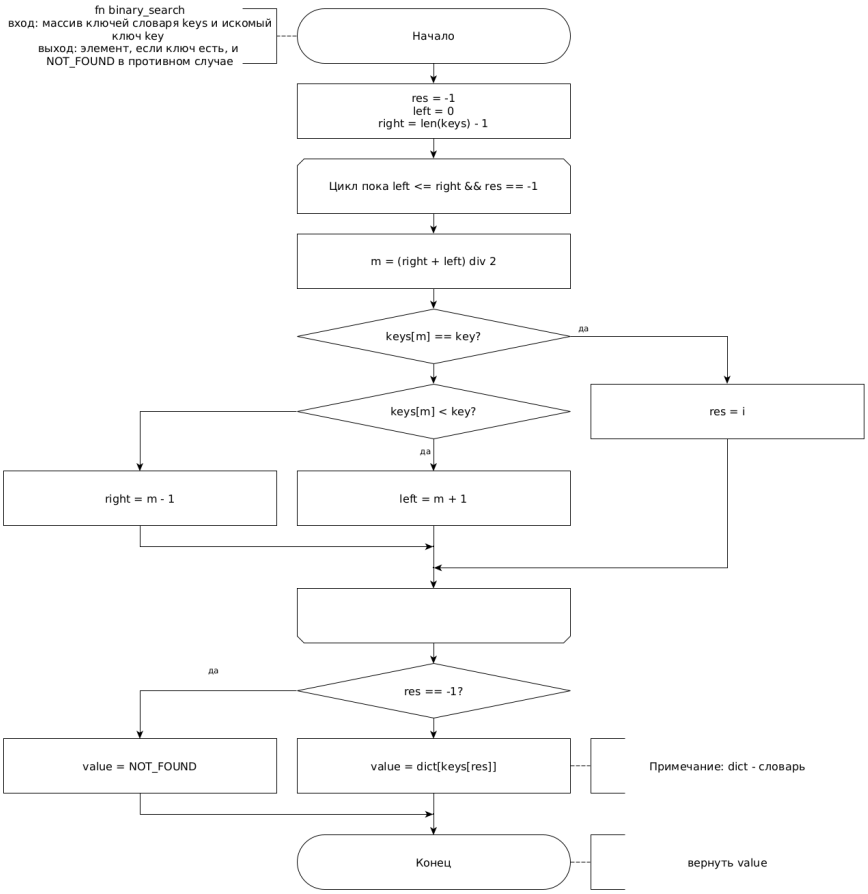

Рис. 2.2: Схема алгоритма поиска с использованием двоичного поиска.

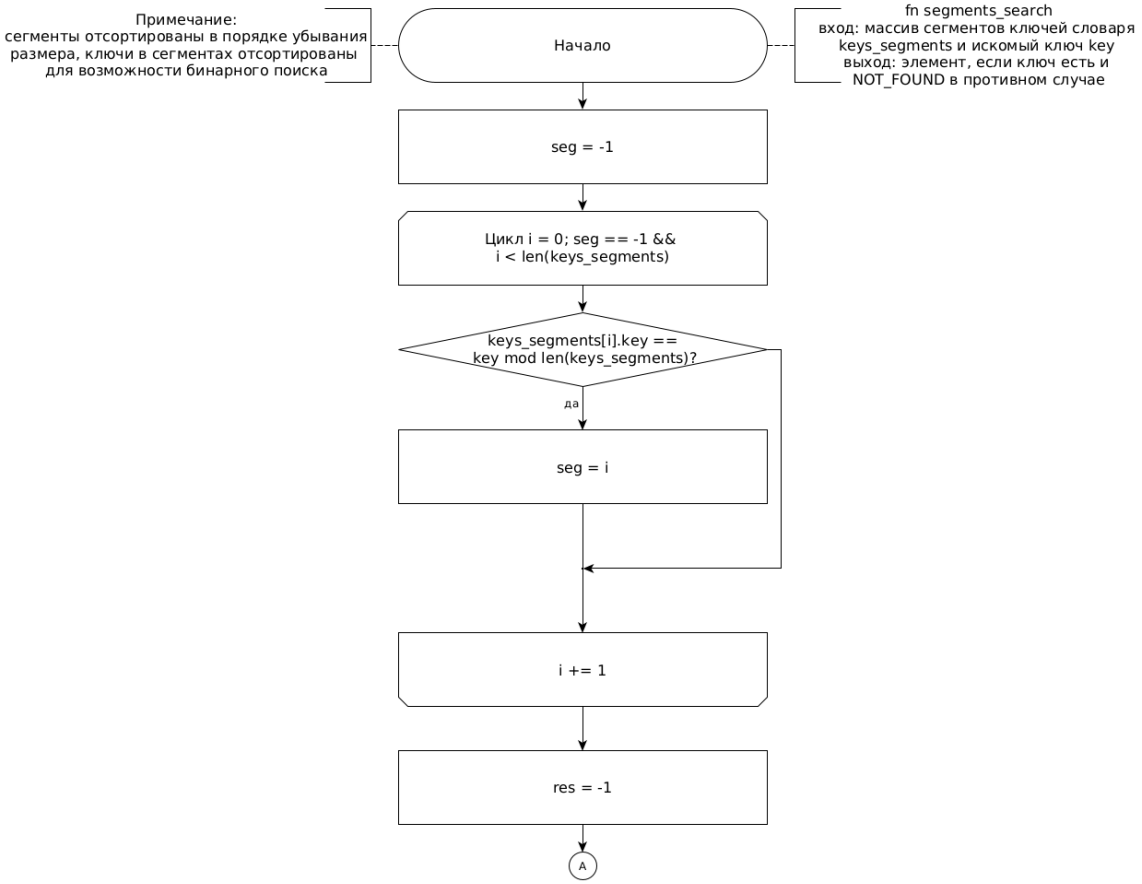

Рис. 2.3: Схема алгоритма поиска с использованием разделения на сегменты и частотного анализа.

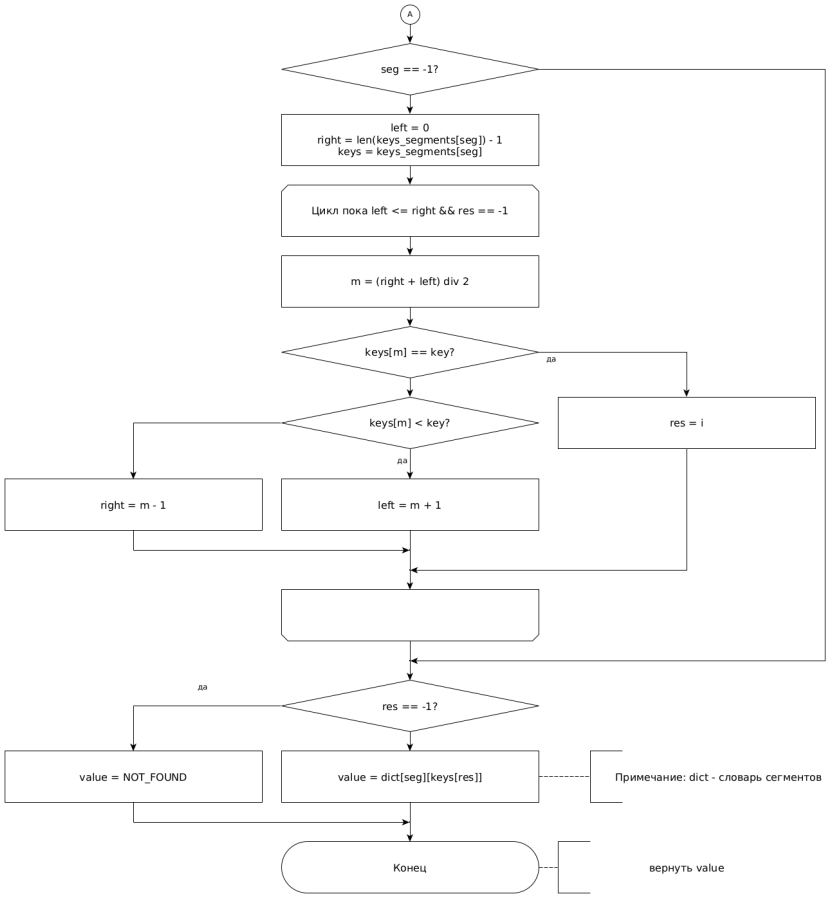

Рис. 2.4: Схема алгоритма поиска с использованием разделения на сегменты и частотного анализа. Продолжение.

Вывод

В данном разделе были рассмотрены структура словаря, на котором будут проводиться эксперименты, а также схемы алгоритмов поисков.

3 Технологическая часть

В данном разделе приведены средства программной реализации и листинг кода.

3.1 Требования к ПО

К программе предъявляется ряд требований:

- на вход подается ключ;
- на выход программа выдает значение, хранящееся в словаре по ключу, если таковое присутствует, “пустое” значение в противном случае.

3.2 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран современный ЯП Golang. [3] Данный выбор обусловлен популярностью языка и скоростью его выполнения, а также тем, что данный язык предоставляет широкие возможности для написания тестов. [4]

3.3 Листинг кода

В листинге 3.1 приведена реализация словарей.

```
1  package main
2
3  // {"id":number, "Teamname": string}
4  // {"id":0, "Teamname": "PostavteZachotPls"}
5  import (
6          "crypto/rand"
7          "fmt"
8          "reflect"
9          "sort"
10
11         "github.com/brianvoe/gofakeit"
```

```
        "github.com/logrusorgru/aurora"
)

func main() {
        fmt.Printf("%v", aurora.Magenta("␣␣␣\n\n"))

        darr := CreateArray(10)
        farr := darr.FAnalysis()
        gt := "applicationpursue849"

        fmt.Printf("%v␣%v\n\n", aurora.Green("␣␣:"), aurora.Blue(gt))
        darr.Print()
        fmt.Println()

        r := farr.Combined(gt)
        if r["teamname"] == nil {
                fmt.Printf("%v\n", aurora.Red("␣␣␣␣␣"))
        } else {
                fmt.Printf("%v\n", aurora.Green("␣␣␣␣"))
                r.Print()
        }
}

// CreateArray used to create DictArray with given size.
func CreateArray(n int) DictArray {
        var (
                darr DictArray
                g Dict
        )

        darr = make(DictArray, n)

        for i := 0; i < n; i++ {
                dup := true
                for dup != false {
                        g = Dict{
                                "id": gofakeit.Uint8(),
                                "teamname": gofakeit.Teams(),
                        }
                        dup = g.IsDup(darr[:i])
                }

                darr[i] = g
        }

        return darr
}

```

```
60 // IsDup used to check whether Dict presents in given DictArray.
61 func (d Dict) IsDup(darr DictArray) bool {
62         for _, v := range darr {
63                 if reflect.DeepEqual(d, v) {
64                         return true
65                 }
66         }
67         return false
68 }
69
70 // Print used to print single Dict.
71 func (d Dict) Print() {
72         fmt.Printf("ID: %v\nTeamname: %v\n", d["id"], d["teamname"])
73 }
74
75 // Print used to print single DictArray.
76 func (darr DictArray) Print() {
77         for _, d := range darr {
78                 d.Print()
79         }
80 }
81
82 func (darr DictArray) Pick(l string) string {
83         for _, d := range darr {
84                 if d["teamname"].(string)[:1] == l {
85                         return d["teamname"].(string)
86                 }
87         }
88
89         i := rand.Int() % len(darr)
90
91         return darr[i]["teamname"].(string)
92 }
93
94 // Brute used to find value using bruteforce method.
95 func (darr DictArray) Brute(gt string) Dict {
96         var r Dict
97
98         for _, d := range darr {
99                 if d["teamname"] == gt {
100                        return d
101                }
102        }
103
104        return r
105 }
106
107 // Binary used to find value using binary search method.
```

```
108 func (darr DictArray) Binary(gt string) Dict {
109         var (
110                 l int = len(darr)
111                 mid int = l / 2
112                 r Dict
113         )
114 
115         switch {
116         case l == 0:
117                 return r
118         case darr[mid]["teamname"].(string) > gt:
119                 r = darr[:mid].Binary(gt)
120         case darr[mid]["teamname"].(string) < gt:
121                 r = darr[mid+1:].Binary(gt)
122         default:
123                 r = darr[mid]
124         }
125 
126         return r
127 }
128 
129 // FAnalysis used to analyse frequency of given DictArray.
130 func (darr DictArray) FAnalysis() FreqArray {
131         var (
132                 az string = "abcdefghijklmnopqrstuvwxyz"
133                 farr FreqArray = make(FreqArray, len(az))
134         )
135 
136         for i, v := range az {
137                 a := Freq{
138                         l: string(v),
139                         cnt: 0,
140                         darr: make(DictArray, 0),
141                 }
142                 farr[i] = a
143         }
144 
145         for _, v := range darr {
146                 l := v["teamname"].(string)[:1]
147                 for i := range farr {
148                         if farr[i].l == l {
149                                 farr[i].cnt++
150                         }
151                 }
152         }
153 
154         sort.Slice(farr, func(i, j int) bool {
155                 return farr[i].cnt > farr[j].cnt
```

```
156         })
157
158         for i := range farr {
159                 for j := range darr {
160                         if darr[j]["teamname"].(string)[:1] == farr[i].l {
161                                 farr[i].darr = append(farr[i].darr, darr[j])
162                         }
163                 }
164
165                 sort.Slice(farr[i].darr, func(l, m int) bool {
166                         return farr[i].darr[l]["teamname"].(string) <
                                farr[i].darr[m]["teamname"].(string)
167                 })
168         }
169
170         return farr
171 }
172
173 // Combined used to find value using binary search and frequency analysis method.
174 func (farr FreqArray) Combined(w string) Dict {
175         var (
176                 l string = w[:1]
177                 r Dict
178         )
179
180         for _, d := range farr {
181                 if string(d.l) == l {
182                         r = d.darr.Binary(w)
183                 }
184         }
185
186         return r
187 }
188
189 // Dict used to represent dictionary with custom types.
190 type Dict map[string]interface{}
191
192 // DictArray used to represent array of Dict instances.
193 type DictArray []Dict
194
195 // Freq used to represent frequency analyser type.
196 type Freq struct {
197         l string
198         cnt int
199         darr DictArray
200 }
201
202 // FreqArray used to represent array of Freq instances.
```

```
203 type FreqArray []Freq
```

Листинг 3.1: Реализация словарей.

3.4 Тестирование функций.

В таблице 3.1 представлены данные для тестирования. Все тесты пройдены успешно.

Ключ	Словарь	Ожидание	Результат
1	`{1: "Navi", 2: "VP"}`	"Navi"	"Navi."
3	`{1: "Navi", 2: "VP"}`	NOT_FOUND	NOT_FOUND
1	`{}`	NOT_FOUND	NOT_FOUND

Таблица 3.1: Тестирование функций.

Вывод

Была разработана и протестирована реализация словарей.

4 Исследовательская часть

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

4.1 Технические характеристики

- Операционная система: Manjaro [5] Linux [6] x86_64.
- Память: 8 ГБ.
- Процессор: Intel® Core™ i7-8550U[7].

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

4.2 Замеры и исследование результатов.

Тестирование проводилось на 2 наборах для каждого алгоритма и каждого количества подборов. Первый набор включал в себя исключительно ключи, которые есть в словаре, причём данные ключи перебираются итеративно, что имитирует равновозможность выпадения ключа (в тестовом наборе 2409 ключей, таким образом для 10.000.000 подборов каждый ключ берется ≈ 4151 раз). Второй набор включал исключительно отсутствующие в словаре ключи, что позволяло проверить, насколько быстро алгоритмы поиска способны обнаружить тот факт, что ключа нет среди имеющихся.

В таблицах 4.1, 4.2 и 4.3 представлены времена работы полного перебора, двоичного поиска и сегментированного алгоритмов. В таблице представлены значения:

- количество раз, которое производился поиск (Кол-во);
- суммарное время поиска только присутствующих ключей (ВППК; в нс);

- суммарное время поиска только отсутствующих ключей (ВППК; в нс).

Примечание: для сегментированного алгоритма было выбрано деление на 5 сегментов.

Кол-во	ВППК	ВПОК
1000	16304689	14466108
10000	303890897	327973602
100000	3047600809	3473714374
1000000	33328461139	34389768377
10000000	341210360139	354735971458

Таблица 4.1: Время работы полного перебора.

Кол-во	ВППК	ВПОК
1000	410665	398989
10000	4132803	3969430
100000	40829389	40449005
1000000	407241884	409383404
10000000	4045076157	4065069524

Таблица 4.2: Время работы двоичного поиска.

Кол-во	ВППК	ВПОК
1000	491870	501553
10000	4205062	4243667
100000	42044052	43403032
1000000	406392543	408692222
10000000	4062301277	4084322643

Таблица 4.3: Время работы сегментированного алгоритма с частотным анализом.

На графике 4.1 построены графики зависимости времени от количества подборов из словаря для алгоритмов поиска (для данных из таблиц 4.1, 4.2 и 4.3):

- полного перебора при запросе только присутствующих ключей (BruteGood);

- полного перебора при запросе только отсутствующих ключей (BruteBad);
- бинарного при запросе только присутствующих ключей (BinaryGood);
- бинарного при запросе только отсутствующих ключей (BinaryBad);
- сегментированного с частотным анализом при запросе только присутствующих ключей (SegmentGood);
- сегментированного с частотным анализом при запросе только отсутствующих ключей (SegmentBad).

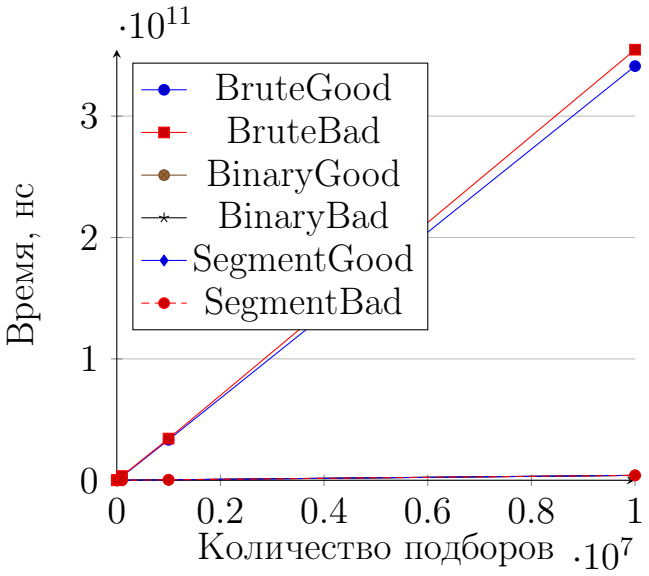

Рис. 4.1: Сравнение алгоритмов.

Вывод

Алгоритм полного перебора оказался самым медленным, при 10.000.000 элементах и поиске только присутствующих ключей среднее время поиска ≈ 34.121 мкс, в то время как среднее время поиска сегментированного алгоритма на тех же данных составило ≈ 0.406 мкс, что приблизительно равно времени бинарного поиска ≈ 0.405 мкс, который оказался менее чем на 1% быстрее. При том же наборе, но поиске только отсутствующих ключей полный перебор работает медленнее на ≈ 35.474 мкс (что составляет примерно 4%), в то время как времена сегментированного алгоритма

(≈ 0.408) и бинарного поиска (≈ 0.406), также ухудшились, но менее, чем на 1%.

Заключение

В рамках выполнения работы были выполнены следующие задачи:

- был реализован алгоритм поиска по словарю, использующий полный перебор;
- был реализован алгоритм поиска по словарю, использующий двоичный поиск;
- был применён частотный анализ для эффективного поиска по словарю;
- были сравнены результаты работы алгоритмов;
- были сделаны выводы по проделанной работе.

Работа показала, что алгоритм поиска полным перебором, работающий за линейное время, в общем случае на несколько порядков медленнее алгоритмов поиска, работающих за логарфмическое время.

Литература

[1] Коршунов Ю. М. Коршуном Ю. М. Математические основы кибернетики // Энергоатомиздат. 1972.

[2] Official teams. Режим доступа: `https://game-tournaments.com/dota-2/team` (дата обращения: 02.10.2020).

[3] Go Programming Language [Электронный ресурс]. URL: `https://golang.org`.

[4] Документация по Golang: бенчмарки [Электронный ресурс]. Режим доступа: `https://golang.org/benchmark` (дата обращения: 10.10.2020).

[5] Manjaro – enjoy the simplicity [Электронный ресурс]. Режим доступа: `https://manjaro.org/` (дата обращения: 10.10.2020).

[6] Русская информация об ОС Linux [Электронный ресурс]. Режим доступа: `https://www.linux.org.ru/` (дата обращения: 10.10.2020).

[7] Процессор Intel® Core™ i7-8550U [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` (дата обращения: 10.10.2020).