**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: Ле Ни Куанг

Группа: ИУ7и-56Б

Преподаватель: Волкова Л. Л.

Строганов Ю. В.

Москва

2021

# Оглавление

# Введение

В данной работе под словарем понимается набор значений - ключей, он имеет множество практических приложений.

**Целью работы:** изучить способы поиска по словарю.

**Задачи работы:**

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

# 1 Аналитический раздел

В данном разделе представлены теоретические сведения о алгоритмах поиска в словаре.

## 1.1 Алгоритм полного перебора

Алгоритм перебирает ключи словаря, пока не будет найден искомый ключ. Возможно (N + 1) случаев: ключ не найден и N возможных случаев расположения ключа в словаре. Лучший случай: за одно сравнение ключ найден в начале словаря. Худших случаев два: за N сравнений либо элемент не найден, либо ключ найден на последнем сравнении. Трудоемкость в среднем:

$$\sum_{i \in \Omega} p_i \cdot f_i = k_0 + k_1(1 + \frac{N}{2} - \frac{1}{N+1}) \tag{1.1}$$

## 1.2 Алгоритм двоичного поиска

Алгоритм требует ключи словаря отсортированы. При двоичном поиске обход можно представить деревом, поэтому трудоемкость 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 у N.

## 1.3 Алгоритм частотного анализа

Алгоритм на вход получает словарь и на его основе составляется частотный анализ. 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 $O(\log n)$.

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.

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

## 1.4 Описание словаря

Словарь представляет собой набор информации о 1000 распространенных криптовалютах на 24.01.21. Запись словаря, реализованная на данной работе, имеет вид ( Rank: int, Name: string, Symbol: string, Market Cap: int64, Price: float32 ). Поиск по полю Name.

## 1.5 Вывод

В данном разделе были описаны два алгоритма и способ оптимизации для поиска в словаре. Так же был рассмотрен описание словаря.

# 2 Конструкторский раздел

Получил зачет, схемы рисовать не буду )

# 3 Технологический раздел

В данном разделе будет приведены требования к программу и листинг кода.

## 3.1 Требования к программному обеспечению

Программа должна принимать название криптовалюты в качестве входных данных.

Результатом работы программы является информация о криптовалюте.

## 3.2 Средства реализации

Язык программирования: Go, Python (обработка данных)

Редактор: VS Code

Go - это новый мощный язык программирования, с простым и понятным синтаксисом. Я учил Go недавно, поэтому хочу использовать его на практике.

## 3.3 Листинг кода

В листингах ниже представлены важные файлы кода программа.

Листинг 3.1: Файл ant.go

```
1  package main
2
3  import (
4      "fmt"
5      "os"
6      "sort"
7  )
8
9  // # - symbol, number, ... (< A)
10 // * - lowercase, ... (> Z)
11 const (
12     Groups = "#ABCDEFGHIJKLMNOPQRSTUVWXYZ*"
13     L      = len(Groups)
14 )
15
16 type DictRecord struct {
17     k string
18     v interface{}
19 }
20
```

```
type FreqRecord struct {
    char  rune
    count int
    start int
    end   int
}

type Dict struct {
    data []DictRecord
    freq []FreqRecord
}

func (d *Dict) put(records []DictRecord) {
    for _, r := range records {
        d.data = append(d.data, r)
    }
    d.update()
}

func (d *Dict) print() {
    for _, r := range d.data {
        fmt.Printf("%v␣:\t%v\n", r.k, r.v)
    }
}

func (d *Dict) printFreqTable() {
    for _, r := range d.freq {
        fmt.Printf("%c␣:␣%v\t(%v-%v)\n", r.char, r.count, r.start, r.end)
    }
}

func (d *Dict) init(filename string) {
    d.freq = make([]FreqRecord, L)
    d.loadDictFromFile(filename)
    d.update()
}

func (d *Dict) update() {
    sort.Slice(d.data, func(i, j int) bool {
        return d.data[i].k < d.data[j].k
    })

    d.updateFreqTable()

    sort.Slice(d.freq, func(i, j int) bool {
        return d.freq[i].count > d.freq[j].count
    })
}
```

```
69
70 func (d *Dict) loadDictFromFile(filename string) {
71     f, err := os.Open(filename)
72     if err != nil {
73         panic(err.Error())
74     }
75     defer f.Close()
76
77     // depending on specific dictionary and file format
78     d.readDict(f)
79 }
80
81 func (d *Dict) updateFreqTable() {
82     lData := len(d.data)
83
84     for i, c := range Groups {
85         d.freq[i].char = c
86     }
87
88     start, end := 0, 0
89     for end < lData && d.data[end].k[0] < Groups[1] {
90         end++
91     }
92     d.freq[0].setFreqRecord(start, end)
93     start = end
94
95     for i := 1; i < L-1; i++ {
96         for end < lData && d.data[end].k[0] == Groups[i] {
97             end++
98         }
99         d.freq[i].setFreqRecord(start, end)
100        start = end
101    }
102
103    d.freq[L-1].setFreqRecord(end, lData)
104 }
105
106 func (r *FreqRecord) setFreqRecord(start, end int) {
107    r.start = start
108    r.end = end - 1
109    r.count = end - start
110 }
111
112 func (d *Dict) linearSearch(key string) interface{} {
113    for _, r := range d.data {
114        if r.k == key {
115            return r.v
116        }
```

```
117     }
118     return nil
119 }
120 
121 func (d *Dict) binarySearch(key string) interface{} {
122     return d._binarySearch(key, 0, len(d.data)-1)
123 }
124 
125 func (d *Dict) _binarySearch(key string, start, end int) interface{} {
126     if start > end {
127         return nil
128     }
129 
130     mid := (start + end) / 2
131     cur := d.data[mid]
132 
133     if key < cur.k {
134         return d._binarySearch(key, start, mid-1)
135     } else if key > cur.k {
136         return d._binarySearch(key, mid+1, end)
137     } else {
138         return cur.v
139     }
140 }
141 
142 func (d *Dict) hybridSearch(key string) interface{} {
143     segment := rune(key[0])
144     if segment > 'Z' {
145         segment = '*'
146     } else if segment < 'A' {
147         segment = '#'
148     }
149 
150     for _, v := range d.freq {
151         if segment == v.char {
152             return d._binarySearch(key, v.start, v.end)
153         }
154     }
155 
156     return nil
157 }
158 
159 func (d *Dict) search(key string, searchFunc func(string) interface{}) {
160     var r interface{}
161     t := measureTime(func() {
162         r = searchFunc(key)
163     })
164 
```

```
165     if r != nil {
166         fmt.Println(key, ":\t", r, "\t", t)
167     } else {
168         fmt.Println("Not found!\t", t)
169     }
170 }
```

Листинг 3.2: Файл utils.go

```
1  package main
2
3  import (
4      "encoding/csv"
5      "os"
6      "strconv"
7  )
8
9  const (
10     DataPath = "data/data.csv"
11 )
12
13 type Cryptocurrency struct {
14     // name    string (key)
15     rank      int
16     symbol    string
17     marketcap int64
18     price     float32
19 }
20
21 func (d *Dict) readDictRecord(r []string) {
22     rank, _ := strconv.ParseInt(r[0], 10, 32)
23     marketcap, _ := strconv.ParseInt(r[3], 10, 64)
24     price, _ := strconv.ParseFloat(r[4], 32)
25     d.data = append(d.data, DictRecord{
26         r[1],
27         Cryptocurrency{
28             int(rank),
29             r[2],
30             marketcap,
31             float32(price),
32         },
33     })
34 }
35
36 func (d *Dict) readDict(f *os.File) {
37     records, err := csv.NewReader(f).ReadAll()
38     if err != nil {
39         panic(err.Error())
40     }
```

```
41
42      for _, r := range records[1:] {
43          d.readDictRecord(r)
44      }
45  }
```

## 3.4 Вывод

В этом разделе было рассмотрено требования к программу и кода программы.

# 4 Экспериментальный раздел

В данном разделе будет приведены пример работы программы и сравнение времени работы программы.

## 4.1 Примеры работы

На рисунке 4.1 приведен пример работы программы.

```
→ make
Cryptocurrencies
Search by name
{ Rank, Symbol, Market Cap($), Price($) }

Search: Bitcoin
Linear  Bitcoin :       {1 BTC 600888568010 32289.38}   623ns
Binary  Bitcoin :       {1 BTC 600888568010 32289.38}   440ns
Hybrid  Bitcoin :       {1 BTC 600888568010 32289.38}   424ns

Search: Ethereum
Linear  Ethereum :      {2 ETH 159184556292 1391.61}    18.807µs
Binary  Ethereum :      {2 ETH 159184556292 1391.61}    450ns
Hybrid  Ethereum :      {2 ETH 159184556292 1391.61}    309ns

Search: Chainlink
Linear  Chainlink :     {7 LINK 9942734313 24.7}        568ns
Binary  Chainlink :     {7 LINK 9942734313 24.7}        450ns
Hybrid  Chainlink :     {7 LINK 9942734313 24.7}        247ns

Search: 0Chain
Linear  0Chain :        {344 ZCN 32692001 0.6754}       116ns
Binary  0Chain :        {344 ZCN 32692001 0.6754}       12.842µs
Hybrid  0Chain :        {344 ZCN 32692001 0.6754}       162ns

Search: key
Linear  Not found!      1.96µs
Binary  Not found!      401ns
Hybrid  Not found!      294ns

#### BENCHMARK ####
        Average         Best            Worst
Linear  6.968173ms      107.004µs       12.412629ms
Binary  1.336294ms      152.611µs       1.627863ms
Hybrid  920.293µs       864.991µs       1.051912ms

Average,Best,Worst
6968,107,12413
1336,153,1628
920,865,1052
```

Рис. 4.1: Примеры работы программы

## 4.2 Результат тестирования

На рисунке 4.2 приведен результат тестирования. Словарь состоит из 1000 элементов, были протестированы элементы с номерами 0, 999, 499, 500, 101, 777 и несуществующий ключ.

```
→ go test -v
=== RUN   TestLinearSearch
--- PASS: TestLinearSearch (0.00s)
=== RUN   TestBinarySearch
--- PASS: TestBinarySearch (0.00s)
=== RUN   TestHybridSearch
--- PASS: TestHybridSearch (0.00s)
PASS
ok      _/mnt/Work/IU7/5/AA/src/7       0.003s
```

Рис. 4.2: Результат тестирования

## 4.3 Сравнение времени работы

Операционная система - Ubuntu 20.04.1 LTS

Процессор - Intel® CoreTM i5-7300HQ CPU @ 2.50GHz × 4 (ЦП 4 ядра 4 потока)

В таблице 4.1 приведена таблица частотного анализа.

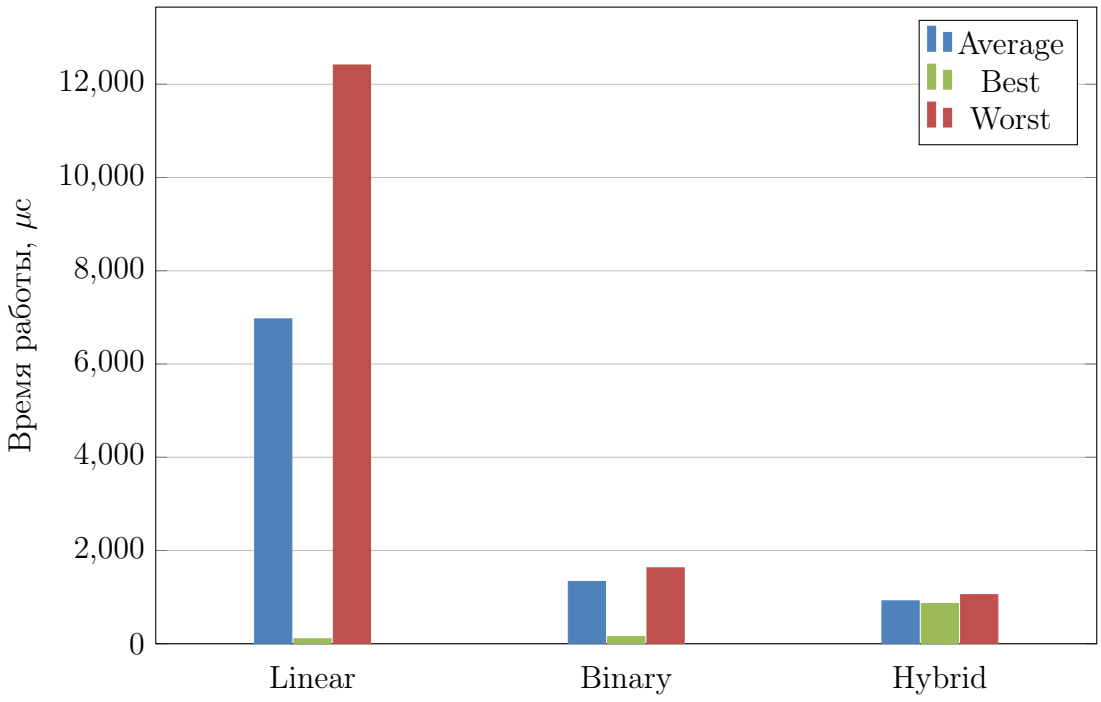


Рис. 4.3: Зависимость времени работы алгоритмов поиска

| **Буква слова** | **Количество слов** |
|---|---|
| B | 80 |
| M | 70 |
| A | 63 |
| C | 62 |
| D | 59 |
| P | 57 |
| T | 54 |
| E | 40 |
| N | 40 |
| H | 37 |
| G | 33 |
| * (>Z) | 32 |
| R | 31 |
| V | 30 |
| F | 30 |
| O | 28 |
| L | 28 |
| I | 22 |
| U | 22 |
| W | 22 |
| K | 18 |
| Z | 15 |
| Q | 14 |
| # (<A) | 8 |
| J | 7 |
| X | 6 |
| Y | 6 |

Таблица 4.1: Частотный анализ

## 4.4 Вывод

Эксперимент показывает, что в среднем алгоритм линейного поиска худший, а лучший - гибридный алгоритм (сегментация + бинарный поиск). Алгоритм линейного поиска не требует сортировки данных, но для отсортированных данных он работает очень быстро, чтобы найти первые (например, найти самую популярную криптовалюту). Гибридный алгоритм может работать даже лучше с большим количеством сегментов, если вместо линейного поиска сегмента мы будем искать с использованием хэш-карты.

# Заключение

В ходе лабораторной работы было изучены алгоритмы полного перебора, двоичного поиска и эффективного поиска по словарю. Было сравнить временные характеристики алгоритмов поиска и сделаны следующие выводы:

- эксперимент показывает, что в среднем алгоритм линейного поиска худший, а лучший - гибридный алгоритм (сегментация + бинарный поиск);
- алгоритм линейного поиска не требует сортировки данных, но для отсортированных данных он работает очень быстро, чтобы найти первые;
- для гибридного алгоритма разница во времени среднего, лучшего и худшего случаев не велика.

# Литература

[1] Effective Go
`https://golang.org/doc/effective_go.html` [Электронный ресурс] (дата обращения: 28.01.21)

[2] testing - The Go Programming Language
`https://golang.org/pkg/testing` [Электронный ресурс] (дата обращения: 28.01.21)