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

по курсу «Анализ Алгоритмов»

на тему: «Поиск в словаре»

Студент ИУ7-53Б _____ Миронов Г. А.
(Группа) (Подпись, дата) (И. О. Фамилия)

Преподаватель _____ Волкова Л. Л.
(Подпись, дата) (И. О. Фамилия)

2021 г.

СОДЕРЖАНИЕ

Введение

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. 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, строк.

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

Целью данной лабораторной работы является изучение способа эффективного оп времени и памяти поиска по словарю. Для достижения данной цели необходимо решить следующие задачи:

- исследовать основные алгоритмы поиска по словарю;
- привести схемы рассматриваемых алгоритмов;
- описать использующиеся структуры данных;
- описать структуру разрабатываемого программного обеспечения;
- определить средства программной реализации;
- определить требования к программному обеспечению;
- привести сведения о модулях программы;
- провести тестирование реализованного программного обеспечения;
- провести экспериментальные замеры временных характеристик реализованных алгоритмов.

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 работы может стремиться к экспоненциальному.

Пусть алгоритм нашел элемент на первом сравнении. Тогда, в лучшем случае, будет затрачено $k_0 + k_1$ операций, на втором – $k_0 + 2k_1$, на N – $k_0 + Nk_1$. тогда, средняя трудоемкость может быть рассчитано по формуле (1.1), где Ω - множество всех возможных случаев.

$$\sum_{i \in \Omega} p_i t_i = (k_0 + k_1)\frac{1}{N+1} + (k_0 + 2k_1) * \frac{1}{N+1} + \cdots + (k_0 + Nk_1) * \frac{1}{N+1} \quad (1.1)$$

Из (1.1), сгруппировав слагаемые, получим итоговую формулу для расчета средней трудоемкости работы алгоритма:

$$k_0 + k1(\frac{N}{N+1} + \frac{N}{2}) = k_0 + k1(1 + \frac{N}{2} - \frac{1}{N+1}) \quad (1.2)$$

1.2 Алгоритм двоичного поиска

Данный алгоритм применяется к заранее упорядоченным словарям. Процесс двоичного поиска можно описать при помощи шагов:

- сравнить значение ключа, находящегося в середине рассматриваемого интервала (изначально – весь словарь), с данным;
- в случае, если значение меньше (в контексте типа данных) данного, продолжить поиск в левой части интервала, в обратном - в правой;
- продолжать до тех пор, пока найденное значение не будет равно данно-

му или длина интервала не станет равной нулю (означает отсутствие искомого ключа в словаре).

Использование данного алгоритма в для поиска в словаре в любом из случаев будет иметь трудоемкость равную $O(log_2(N))$ [1]. Несмотря на то, что в среднем и худшем случаях данный алгоритм работает быстрее алгоритма полного перебора, стоит отметить, что предварительная сортировка больших данных требует дополнительных затрат по времени и может оказать серьезное действие на время работы алгоритма. Тем не менее, при многократном поиске по одному и тому же словарю, применение алгоритм сортировки понадобится всего один раз.

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 сегмента при сложности $O(nlog(n))$

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 (1.3).

$$\sum_{i \in [1,M]} (f_{select_i} + f_{search_i}) \tag{1.3}$$

1.4 Описание словаря

В данной работе будет использован словарь вида $\{username : string, password : string\}$, что представляет собой базу данных о паролях

пользователей. Поиск будет осуществляться по полю `username`.

1.5 Вывод

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

Входными данными для программного обеспечения являются:

- словарь из записей, вида

$$\{username : string, password : string\}$$

  для поиска по нему;

- ключ для поиска в словаре.

Выходными данными является найденная в словаре запись для каждого из реализуемых алгоритмов.

2 Конструкторская часть

В данном разделе представлены схемы рассматриваемых алгоритмов.

2.1 Схемы

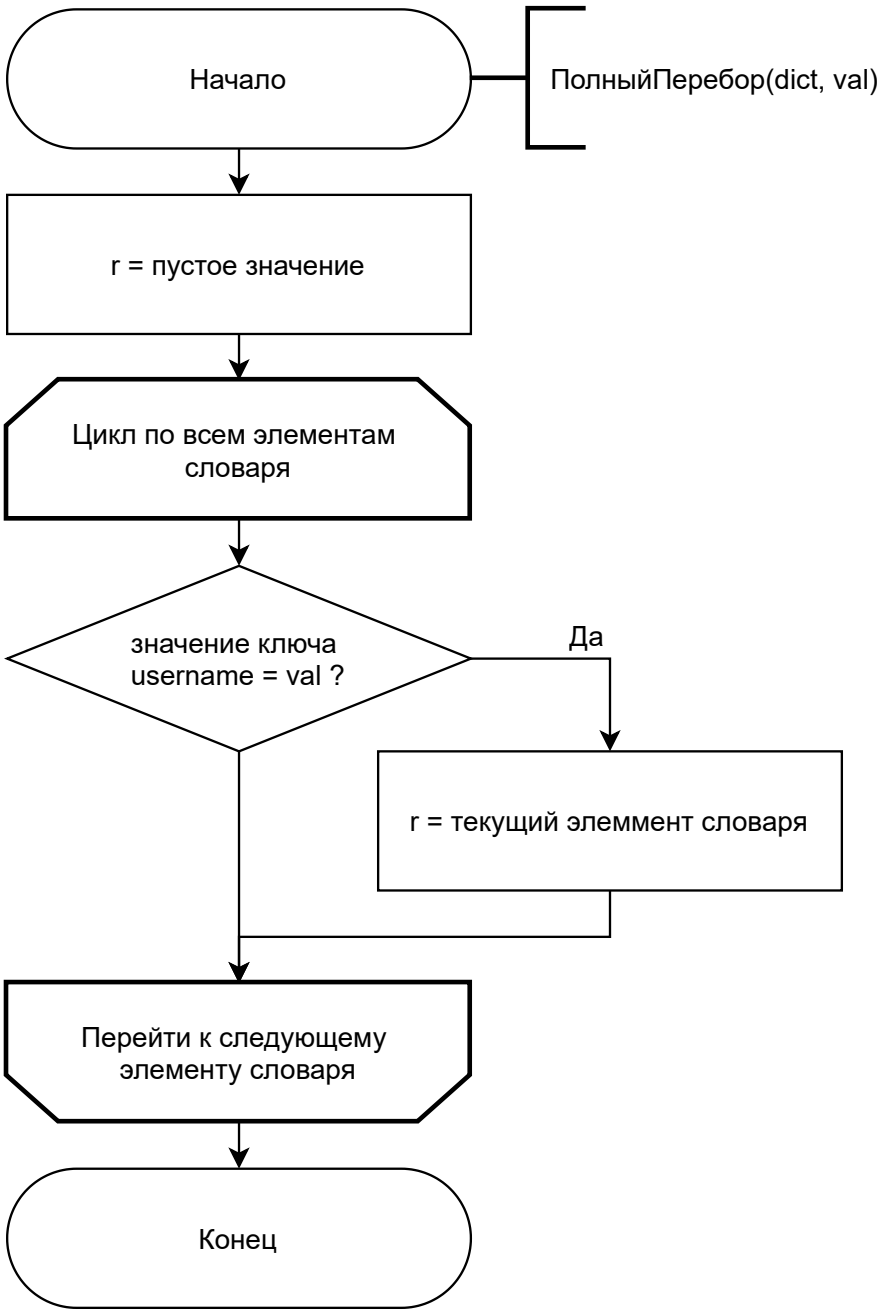

Рисунок 2.1 – Схема алгоритма поиска с полным перебором

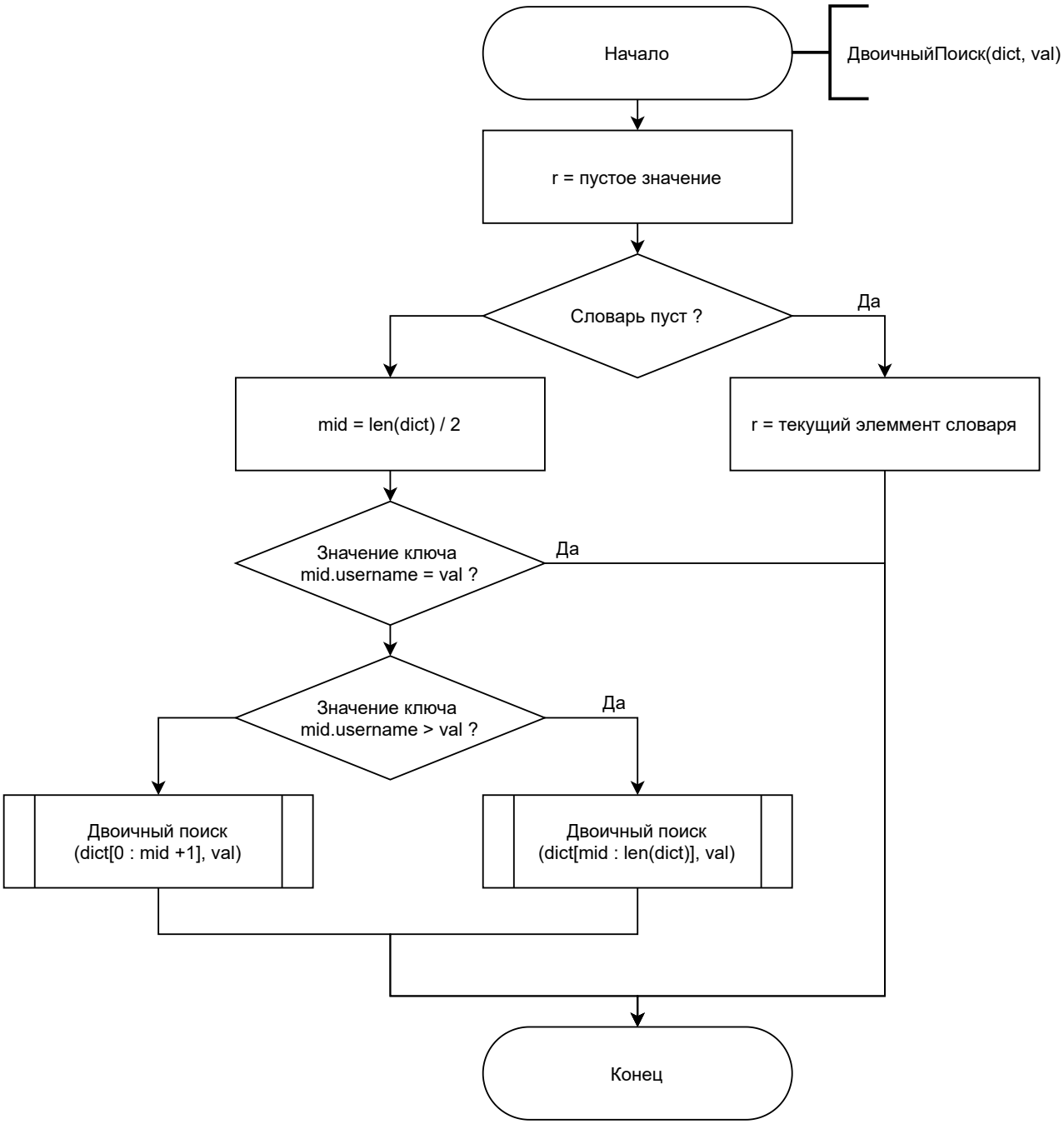

Рисунок 2.2 – Схема алгоритма с бинарным поиском

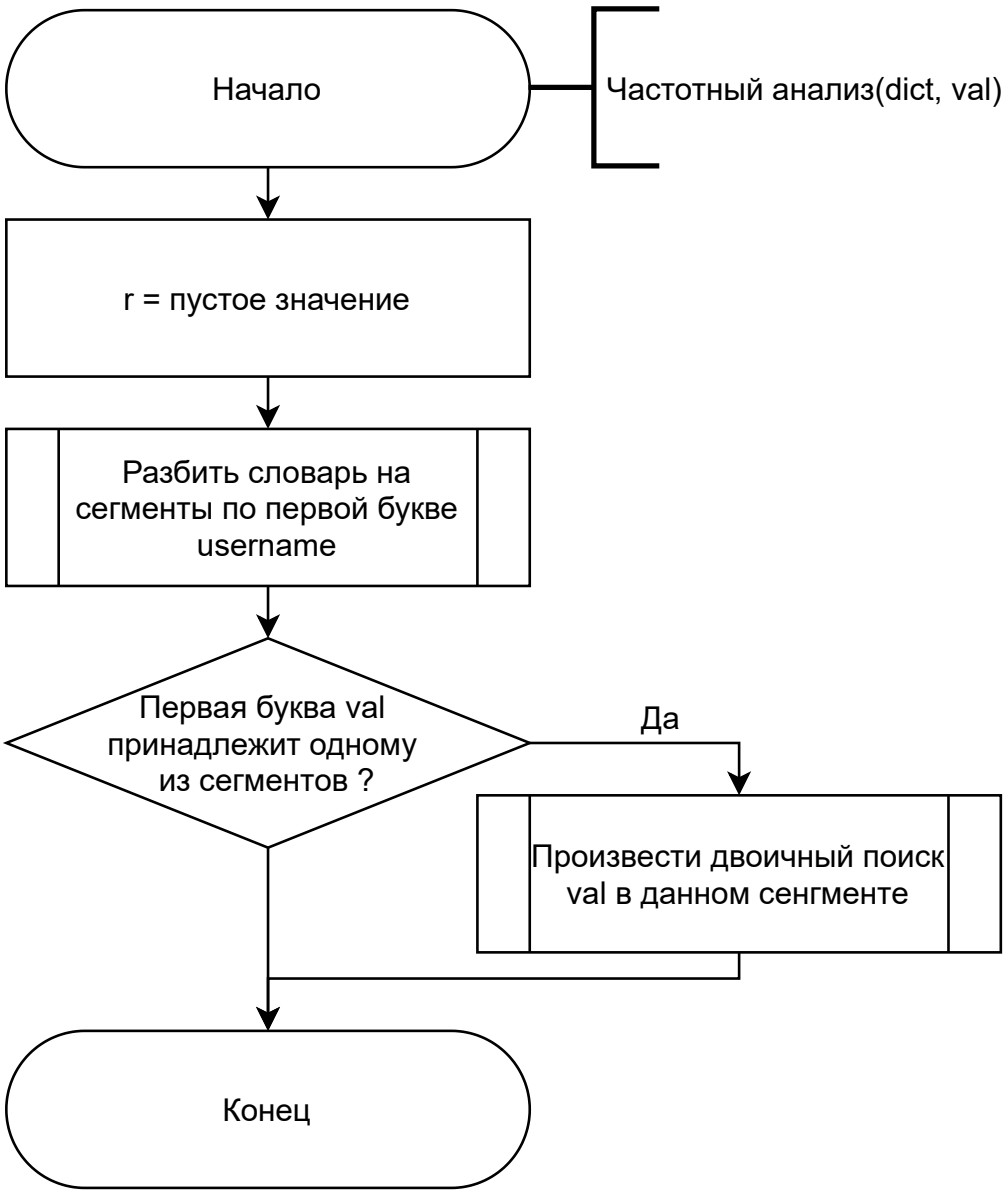

Рисунок 2.3 – Схема алгоритма с частотным анализом

2.2 Описание структуры программного обеспечения

На Рисунке 2.4 представлена `uml`–диаграмма разрабатываемого программного обеспечения.

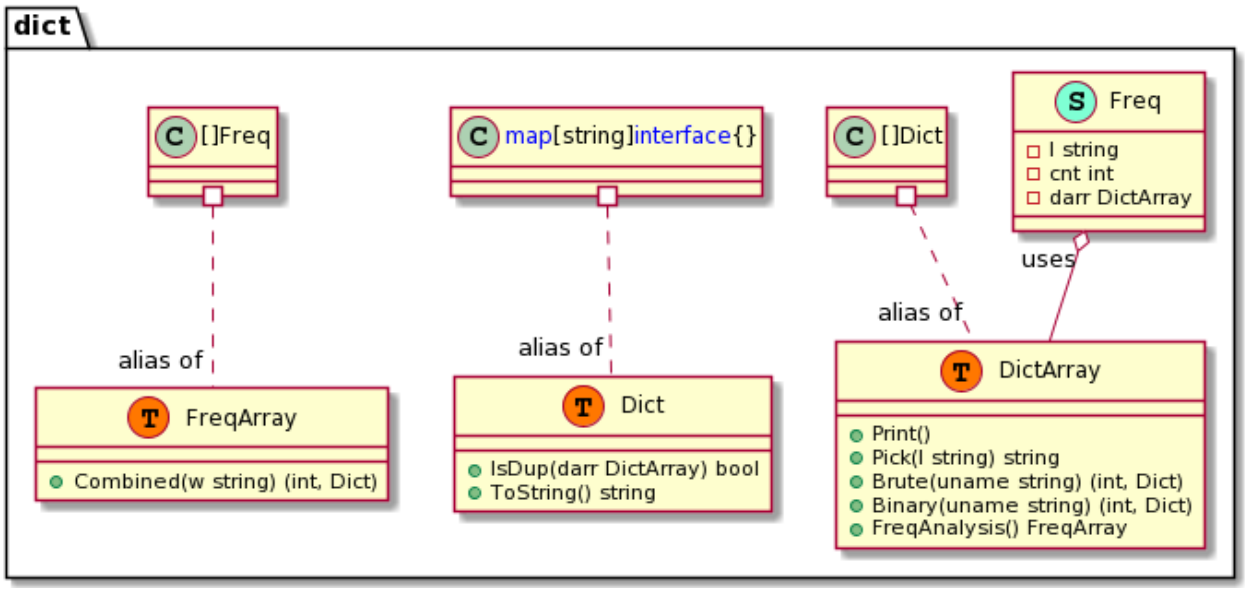

Рисунок 2.4 – Структура программного обеспечения

2.3 Описание структур данных

Для реализации конвейерных вычислений, введем некоторые пользовательские типы данных:

- `Dict` – тип данных, описывающий ассоциативный массив с ключами типа `string`;
- `DictArray` – тип данных, описывающий массив ассоциативных массивов;
- `Freq` – структура, описывающая сегмент частотного анализа ассоциативного массива.
- `FreqArray` – тип данных, описывающий результат частотного анализа ассоциативного массива.

Рассмотрим каждый из введенных пользовательских данных.

2.3.1 Описание пользовательского типа данных `Freq`

Листинг 2.1 – Описание пользовательского типа данных `Freq`

```
type Freq struct {
    l    string
    cnt  int
    darr DictArray
}
```

Здесь:

- `l` – строка, содержащая признак сегмента – букву из используемого алфавита;
- `cnt` – частотная характеристика для данного сегмента;
- `darr` – ассоциативный массив, содержащий записи, соответствующие данному сегменту.

2.4 Вывод

На основе теоретических данных, полученных из аналитического раздела, были построены схемы реализуемыъ алгоритмов поиска в ассоциативных массивах (Рисунки 2.1–2.3).

Так же, было приведено описание пользовательских типов данных, вводимых в рамках реализации конвейерных вычислений.

3 Технологическая часть

3.1 Выбор средств реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран язык Golang [2]. Данный выбор обусловлен тем, что я имею некоторый опыт разработки на нем, а так же наличием у языка встроенных высокоточных средств тестирования и анализа разработанного ПО.

3.2 Требования к программному обеспечению

3.3 Сведения о модулях программы

Данная программа разбита на модули:

- `main.go` - файл, содержащий точку входа в программу;
- `types.go` - файл, содержащий определение пользовательских типов данных;
- `dict.go` - файл, содержащий реализации алгоритмов поиска в словаре.

На листингах 3.1 – 3.7 представлен код программы.

Листинг 3.1 – Определение пользовательских типов данных

```
1
2  type Dict map[string]interface{}
3
4  type DictArray []Dict
5
6  type Freq struct {
7      l    string
8      cnt  int
9      darr DictArray
10 }
11
12 type FreqArray []Freq
```

Листинг 3.2 – Основной файл программы main

```
func printRes(method string, res dict.Dict, count int) {
    fmt.Printf("%v ", aurora.Yellow(method+":"))
    if res["username"] == nil {
        fmt.Printf("%v\n\n", aurora.Red("There is no dict such
           key"))
    } else {
        fmt.Printf("%v [ %d ]\n", aurora.Green("Dict with
           specified key is found"), count)
        fmt.Println(res.ToString())
    }
}

func main() {
    fmt.Printf("%v", aurora.Magenta("Search in array of dict's\n\
       n"))

    darr := dict.CreateArray(10)
    farr := darr.FreqAnalysis()

    fmt.Printf("%v\n\n", aurora.Yellow("Dictionary data:"))
    darr.Print()

    var uname string
    fmt.Printf("%v ", aurora.Green("Key to find:"))
    fmt.Scanln(&uname)
    fmt.Println()

    c, v := darr.Brute(uname)
    printRes("Brute algorythm", v, c)

    sort.Slice(darr, func(i, j int) bool {
        return darr[i]["username"].(string) < darr[j]["username"
           ].(string)
    })
    c, v = darr.Binary(uname)
    printRes("Binary search algorythm", v, c)

    c, v = farr.Combined(uname)
    printRes("Freq Analysis algorythm", v, c)
}
```

Листинг 3.3 – Реализация алгоритмов поиска в словаре

```
1  func CreateArray(n int) DictArray {
2      var (
3          darr DictArray
4          g    Dict
5      )
6
7      darr = make(DictArray, n)
8
9      hash := sha256.New()
10     for i := 0; i < n; i++ {
11         dup := true
12         for dup {
13             hash.Reset()
14             hash.Write([]byte(gofakeit.Password(true, true, true,
                   true, true, i)))
15             g = Dict{
16                 "username": gofakeit.Username(),
17                 "password": hex.EncodeToString(hash.Sum(nil)),
18             }
19             dup = g.IsDup(darr[:i])
20         }
21
22         darr[i] = g
23     }
24
25     return darr
26 }
27
28 func (d Dict) IsDup(darr DictArray) bool {
29     for _, v := range darr {
30         if reflect.DeepEqual(d, v) {
31             return true
32         }
33     }
34     return false
35 }
36
37 func (d Dict) ToString() string {
38     return fmt.Sprintf("Username: %v\nPassword: %v\n", d["
          username"], d["password"])
39 }
```

Листинг 3.4 – Реализация алгоритмов поиска в словаре

```
func (darr DictArray) Print() {
    for _, d := range darr {
        fmt.Println(d.ToString())
    }
}

func (darr DictArray) Pick(l string) string {
    for _, d := range darr {
        if d["username"].(string)[:1] == l {
            return d["username"].(string)
        }
    }

    i := rand.Int() % len(darr)

    return darr[i]["username"].(string)
}

func (darr DictArray) Brute(uname string) (int, Dict) {
    var r Dict

    for i, d := range darr {
        if d["username"] == uname {
            return i + 1, d
        }
    }

    return len(darr), r
}
```

Листинг 3.5 – Реализация алгоритмов поиска в словаре

```
func (darr DictArray) Binary(uname string) (int, Dict) {
    var binary func(DictArray, string, int) (int, Dict)

    binary = func(arr DictArray, uname string, idx int) (int,
       Dict) {
        var (
            l   int = len(arr)
            mid int = l / 2
            r   Dict
        )

        switch {
        case l == 0:
            return idx, r
        case arr[mid]["username"].(string) > uname:
            return binary(arr[:mid], uname, idx+1)
        case arr[mid]["username"].(string) < uname:
            return binary(arr[mid+1:], uname, idx+2)
        default:
            return idx + 2, arr[mid]
        }
    }

    return binary(darr, uname, 0)
}
```

Листинг 3.6 – Реализация алгоритмов поиска в словаре

```
func (darr DictArray) FreqAnalysis() FreqArray {
    farr := make(FreqArray, 0)

    for _, v := range darr {
        found := false
        for i := range farr {
            if v["username"].(string)[:1] == farr[i].l {
                farr[i].darr = append(farr[i].darr, v)
                farr[i].cnt++
                found = true
            }
        }

        if !found {
            a := Freq{
                l:    v["username"].(string)[:1],
                cnt:  1,
                darr: make(DictArray, 0),
            }
            a.darr = append(a.darr, v)
            farr = append(farr, a)
        }
    }

    for i := range farr {
        sort.Slice(farr[i].darr, func(l, m int) bool {
            return farr[i].darr[l]["username"].(string) < farr[i
               ].darr[m]["username"].(string)
        })
    }

    sort.Slice(farr, func(i, j int) bool {
        return farr[i].cnt > farr[j].cnt
    })

    return farr
}
```

Листинг 3.7 – Реализация алгоритмов поиска в словаре

```
func (farr FreqArray) Combined(w string) (int, Dict) {
    var (
        l   string = w[:1]
        r   Dict
        cnt int = len(farr)
    )

    for i, d := range farr {
        if string(d.l) == l {
            var t int
            t, r = d.darr.Binary(w)
            cnt = i + 1 + t
        }
    }

    return cnt, r
}
```

3.4 Тестирование

В рамках данной лабораторной работы будет проведено функциональное тестирование реализованного программного обеспечения.

Тестирование проводилось на словаре, содержащем следующие записи:

- `{username: "username_01"; password:"password_01"}`;
- `{username: "username_02"; password:"password_02"}`.

В Таблице 3.1 приведены соответствующие тесты.

Таблица 3.1 – Таблица тестов

Ключ	**Результат**	**Значение**
username_01	(1)	{username: "username_01"; password:"password_01"} (1)
username_02	(1)	{username: "username_02"; password:"password_02"} (2)
mama_mia	–	–

При проведении функционального тестирования, полученные результаты работы программы совпали с ожидаемыми. Таким образом, функциональное тестирование пройдено успешно.

3.5 Вывод

В данном разделе был реализованы вышеописанный алгоритмы. Было разработано программное обеспечение, удовлетворяющее предъявляемым требованиям. Так же были представлены соответствующие листинги 3.1 – 3.7 с кодом программы. Кроме того, было проведено функциональное тестирование разработанного программного обеспечения.

4 Экспериментальная часть

В данном разделе будет проведено функциональное тестирование разработанного программного обеспечения. Так же будет произведено измерение временных характеристик каждого из реализованных алгоритмов.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось исследование:

- процессор: Intel Core™ i5-8250U [3] CPU @ 1.60GHz;
- память: 32 GiB;
- операционная система: Manjaro [4] Linux [5] 21.1.4 64-bit.

Исследование 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.

4.2 Временные характеристики

В реализованном программном обеспечении особый интерес представляет количество сравнений, необходимое для нахождения записи в словаре по ключу, так как это характеристика прямо пропорциональна времени работы алгоритма. Поэтому, будет проводиться анализ количества сравнений каждого из ключей словаря в словаре, а так же – время поиска ключа не находящегося в словаре. Для большей наглядности число записей в словаре возьмем равным 10000.

Поиск будет производиться каждым алгоритмом по отдельности, после чего будут представлены соответствующие гистограммы. Отметим, что, в общем случае, распределение носит достаточно "случайный" характер, так как исходный массив не обязательно является упорядоченным. В связи с этим, каждая из построенных гистограмм будет продублирована второй, отображающей данные по убыванию числа сравнений.

4.2.1 Алгоритм полного перебора

В данном алгоритме зависимости числа сравнений от количества элементов является линейной. На пример, для обнаружения первого элемента понадобиться 1 сравнение, для 2ого – 2, а для N – N. В связи с этим, поиск последнего элемента в массиве потребует $N = 10000$ сравнений.

Рисунок 4.1 – Число сравнений для нахождения имени полным перебором

На Рисунке 4.1 видна линейная зависимость числа сравнений от положения элемента в массиве.

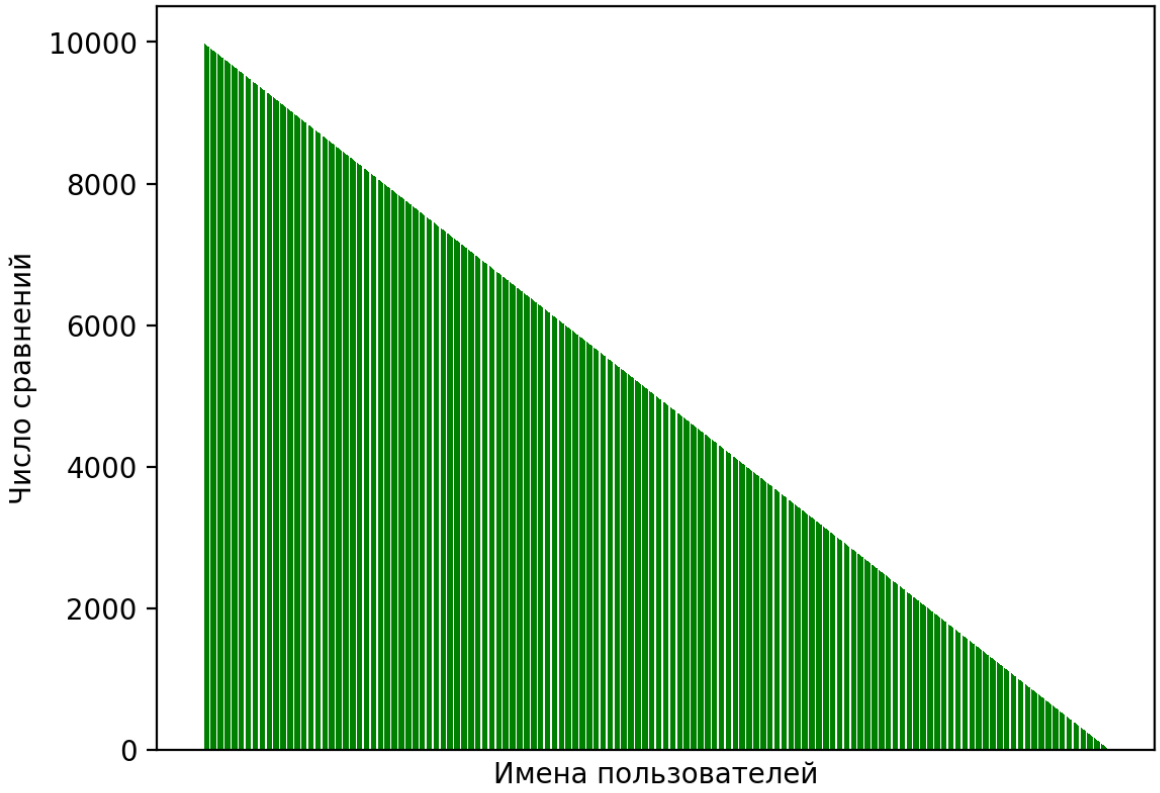

Рисунок 4.2 – Число сравнений для нахождения имени полным перебором по убыванию

4.2.2 Алгоритм бинарного поиска

В данном алгоритме сложность поиска равна $O(log(N))$, в связи с этим, на общей гистограмме будет наблюдаться достаточно произвольное распределение, зависящее от входного массива.

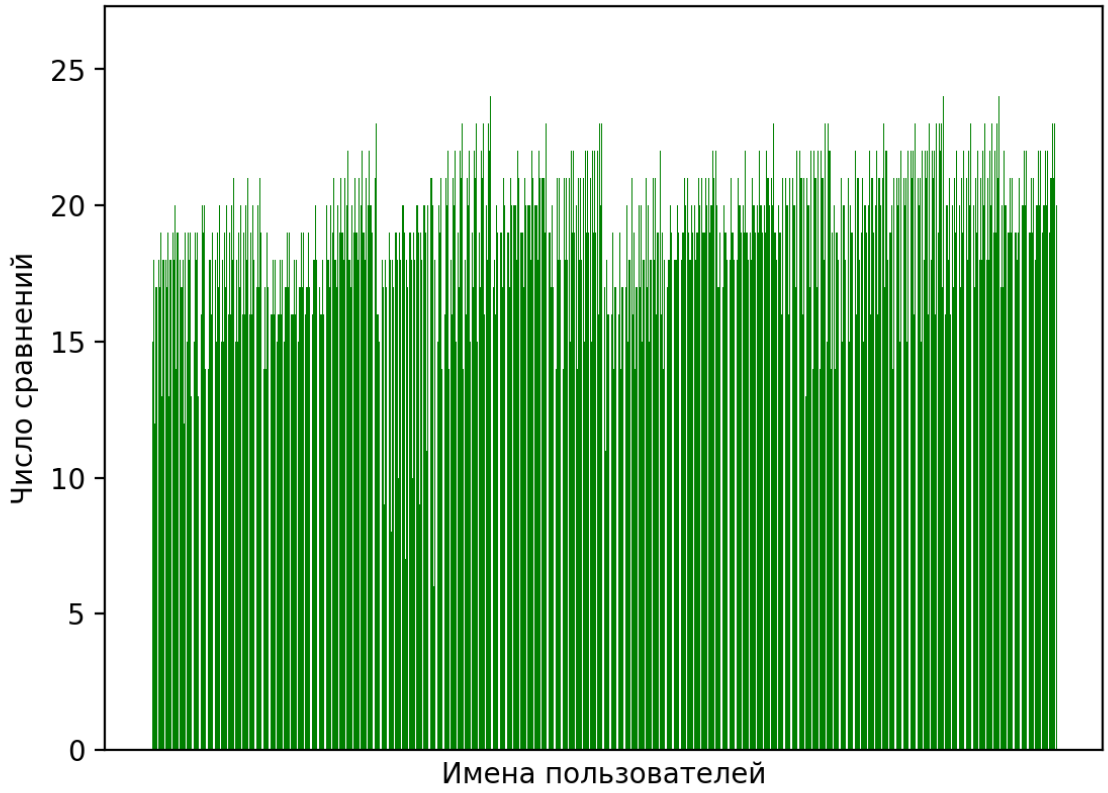

Рисунок 4.3 – Число сравнений для нахождения имени бинарным поиском

На Рисунке 4.3 не наблюдается явной зависимости числа сравнений для поиска элемента от его значения. В связи с этим, стоит упорядочить элементы по числу сравнений и построить вторую гистограмму.

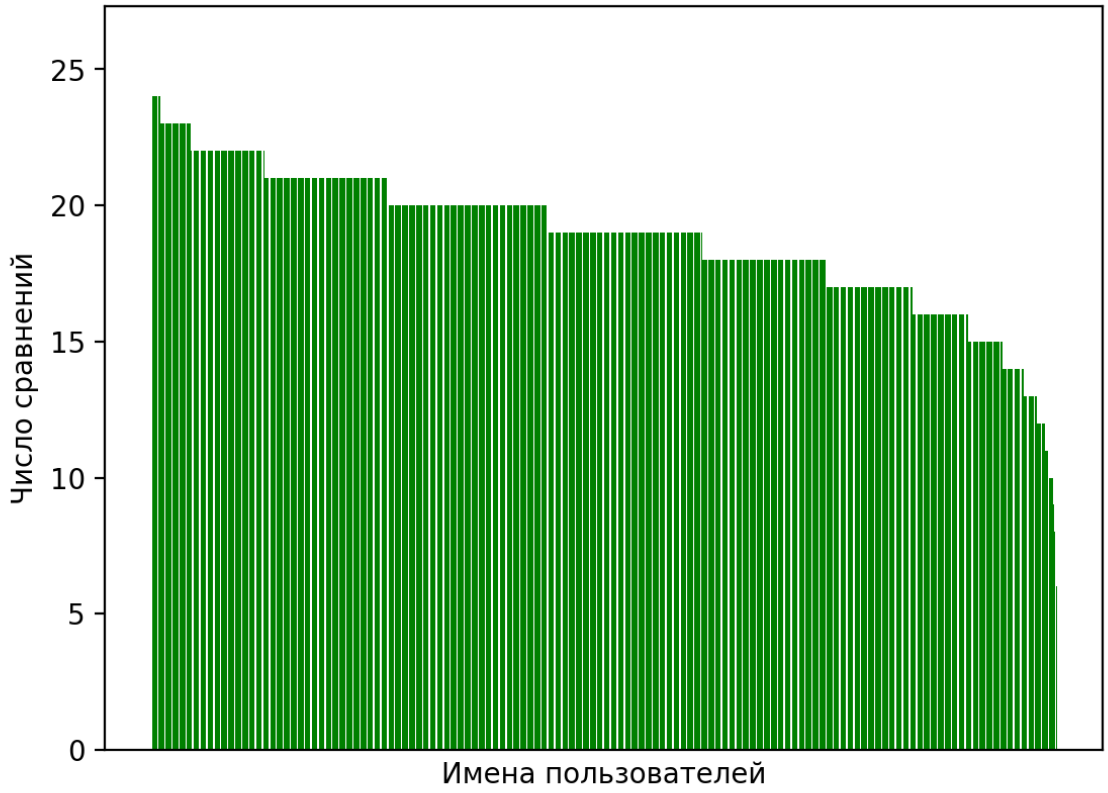

Рисунок 4.4 – Число сравнений для нахождения имени бинарным поиском по убыванию

На Рисунке 4.4 видна зависимость числа сравнений от искомого элемента в данном массиве. Заметим, что в худшем случае число сравнений составило 24, а в лучшем - 1 (элемент находится ровно по середине массива). Таким образом, алгоритм бинарного поиска показывает в 416 раз более высокий результат в сравнении с алгоритмом полного перебора. Стоит отметить, что лучший случай алгоритма полного перебора и лучший случай алгоритма бинарного поиска отличаются, в связи с чем, сравнивать их не корректно.

4.2.3 Алгоритм частотного анализа

В данном алгоритме сложность поиска зависит не только от положения в отсортированном сегменте, но и от размера используемого словаря, так как поиск в нем осуществляется полным перебором. В данной лабораторной работе использовался английский алфавит, содержащий 26 букв. В связи с этим, худший случай потребует не менее 27 сравнений для обнаружения элемента в частотном анализе. Тем не менее, в среднем случае это не должно оказывать серьезного действия на число требуемых сравнений.

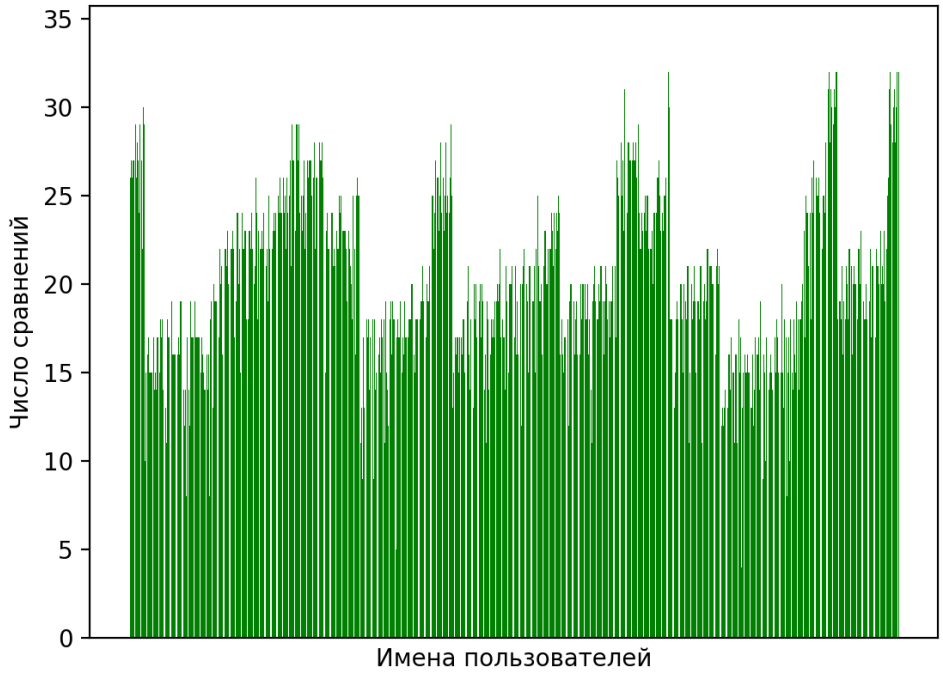

Рисунок 4.5 – Число сравнений для нахождения имени частотным анализом

Как и в случае с бинарным поиском, на Рисунке 4.3 не наблюдается явной зависимости числа сравнений для поиска элемента от его значения. В связи с этим, стоит так же упорядочить элементы по числу сравнений и построить вторую гистограмму.

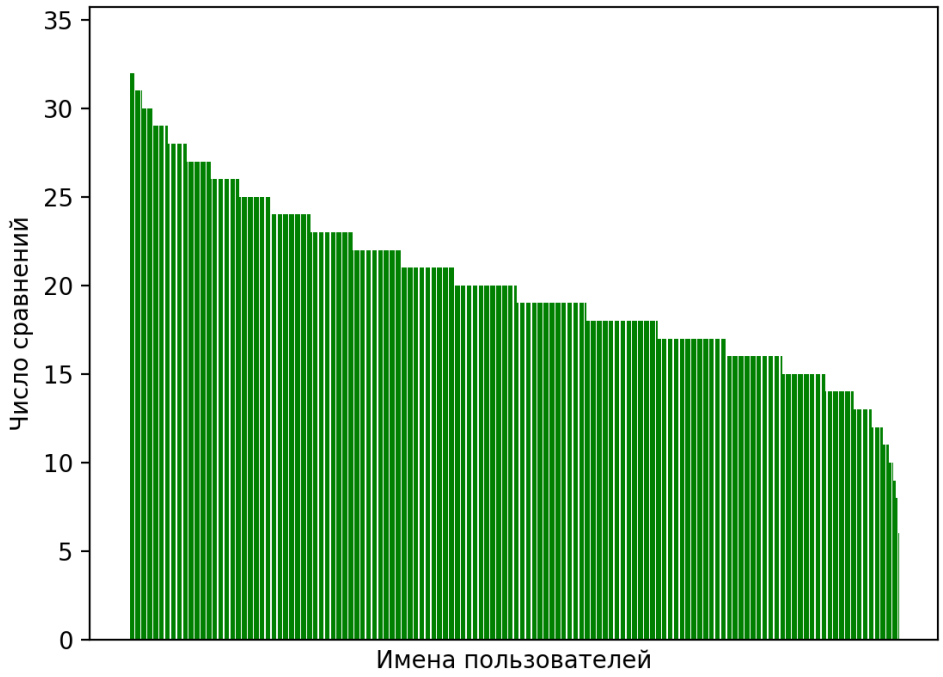

Рисунок 4.6 – Число сравнений для нахождения имени частотным анализом по убыванию

Заметим, что на Рисунке 4.4 видно, что в худшем случае число сравнений составило 33, а в лучшем - 2 (элемент находится ровно по середине массива первого сегмента). Таким образом, алгоритм бинарного поиска показывает в 303 раз более высокий результат в сравнении с алгоритмом полного перебора. Стоит отметить, что лучший случай алгоритма полного перебора и лучший случай алгоритма частотного анализа поиска отличаются, в связи с чем, сравнивать их не корректно.

4.3 Вывод

Исходя из полученных данных, можно сделать вывод, что алгоритм поиска в словаре, использующий частотный анализ, является боле 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 тре-

бует сортировки всего входного массива, что, в среднем случаи имеет сложность $O(nlog(n))$, 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 алгоритмом 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 работы решены следующие задачи:

- исследованы основные алгоритмы поиска по словарю;
- приведены схемы рассматриваемых алгоритмов;
- описаны использующиеся 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 ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Д. К.* Сортировка и поиск. Т. 3. — 3-е изд. — – М.: Вильямс, 2000. — (Искусство программирования).
2. The Go Programming Language [Электронный ресурс]. — Режим доступа: `https://golang.org/` (дата обращения: 24.10.2021).
3. Процессор Intel® Core™ i5-8250U [Электронный ресурс]. — Режим доступа: `https://ark.intel.com/content/www/ru/ru/ark/products/124967/intel-core-i5-8250u-processor-6m-cache-up-to-3-40-ghz.html` (дата обращения: 24.10.2021).
4. Manjaro - enjoy the simplicity [Электронный ресурс]. — Режим доступа: `https://manjaro.org/` (дата обращения: 24.10.2021).
5. LINUX.ORG.RU – Русская информация об ОС Linux [Электронный ресурс]. — Режим доступа: `https://www.linux.org.ru/` (дата обращения: 24.10.2021).