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»

«Поиск в словаре»

Студент Маслова Марина Дмитриевна

Группа ИУ7-53Б

Оценка (баллы) _____

Преподаватель Волкова Лилия Леонидовна

2021 г.

Содержание

Введение

Словарь (или *ассоциативный массив*) [1] – структура данных, позволяющий хранить пары вида (ключ; значение). Основной операцией, применяющейся к данной структуре, является поиск по ключу. Таким образом, становится актуальной задача повышения скорости поиска в словаре.

Целью данной работы является изучение и сравнительный анализ алгоритмов поиска в словаре.

Для достижения поставленной цели необходимо выполнить следующие **задачи**:

- описать алгоритм поиска полным перебором;
- описать алгоритм бинарного поиска;
- описать алгоритм частотного анализа;
- описать функциональные требования;
- разработать описанные алгоритмы;
- реализовать алгоритмы поиска в словаре;
- провести тестирование реализованных алгоритмов;
- провести сравнительный анализ алгоритмов по времени работы реализаций;
- провести сравнительный анализ алгоритмов по количеству сранений;
- сделать выводы по полученным результатам.

1 Аналитическая часть

В данном разделе представлено теоретическое описание словаря и алгоритмов поиска.

1.1 Словарь

Словарь построен на основе пар (ключ, значение). Для данного типа данных определено три операции:

- вставка;
- удаление;
- поиск.

При поиске по заданному ключу возвращается значение или "сообщение", по которому можно понять, что в словаре нет пары с данным ключем.

В данной работе используется словарь настольных игр, где ключом является имя игры, а значением информация о ней: год выпуска, количество отзывов и рейтинг.

Далее будут рассмотрены алгоритмы поиска в словаре.

1.2 Алгоритм полного перебора

Алгоритм полного перебора [2] подразумевает поочередный просмотр всех возможных вариантов. В случае словаря по очереди просматривают ключи словаря до тех пор, пока не будет найден нужный. Трудоёмкость алгоритма зависит от того, присутствует ли искомый ключ в словаре, и, если присутствует — насколько он далеко от начала массива ключей.

Пусть на старте алгоритм поиска затрачивает k_0 операций, а при каждом сравнении k_1 операций. Тогда при поиске первого элемента (лучший случай) будет затрачено $k_0 + k_1$ операций, i-ого — $k_0 + i \cdot k_1$, последнего (худший случай) — $k_0 + N \cdot k_1$. Ситуация отсутствия ключа обнаруживается только послед перебора всех значений, что соответствует трудоёмкости поиска ключа на последней позиции. Средняя трудоёмкость равна математическому ожиданию и может быть рассчитана по формуле 1.1:

$$f_{\text{ср}} = k_0 + k_1 \cdot (1 + \frac{N}{2} - \frac{1}{N+1}) \tag{1.1}$$

1.3 Алгоритм бинарного поиска

Бинарный поиск [2] осуществляется в отсортированном списке ключей. Искомый ключ сравнивается со средним элементом, если ключ меньше, то поиск продолжается в левой части, если больше, то — в правой части, если равен, то искомый ключ найден.

Таким образом при бинарном поиске [2] 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.

1.4 Алгоритм частотного анализа

Алгоритм частотного анализа разбивает словарь на сегменты по какому-либо признаку. В данной работе рассматривается случай определения ключа в сегмент по первому символу (определены сегменты, соответствующие буквам латинского алфавита, цифрам, а также сегмент включающий ключи, которые не попали в другие сегменты).

Сегменты упорядочиваются по значению частотной характеристики так, чтобы к элементам с наибольшей частотной характеристикой был самый быстрый доступ. В данной работе такой характеристикой служит размер сегмента.

Обращение к сегменту происходит с вероятностью равной сумме вероятностей обращений к его ключам, рассчитывающейся по формуле 1.2:

$$P_i = \sum_j p_j = N \cdot p, \tag{1.2}$$

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.

В каждой сегменте ключи 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 сег-

мента. На этом предварительная обработка словаря заканчивается.

При поиске ключа 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 (1.3).

$$\sum_{i \in \Omega} (f_{\text{выбор сегмента i-ого элемента}} + f_{\text{бинарный поиск i-ого элемента}}) \cdot p_i \tag{1.3}$$

1.5 Вывод

В данном разделе был описан словарь, так же алгоритмы поиска значений по ключу: перебор, бинарный поиск и частотный анализ. Из представленных описаний можно предъявить ряд требований к разрабатываемому программному обеспечению:

- на вход должен подавться словарь, также ключ, поиск значения которого осуществляется;
- при отсутствии ключа в словаре должно выдаваться соответствующее сообщение;
- на выходе должно выдаваться значение, соответствующее данному ключу.

2 Конструкторская часть

В данном разделе разрабатываются алгоритмы поиска в словаре: перебором, бинарный и частотный анализ, также описывается структура программы и способы её тестироваия.

2.1 Разработка алгоритмов

На рисунке 2.1 представлена схема алгоритма перебора ключей.
На рисунке 2.2 представлена схема алгоритма бинарного поиска.
На рисунке 2.3 представлена схема алгоритма частотного анализа.

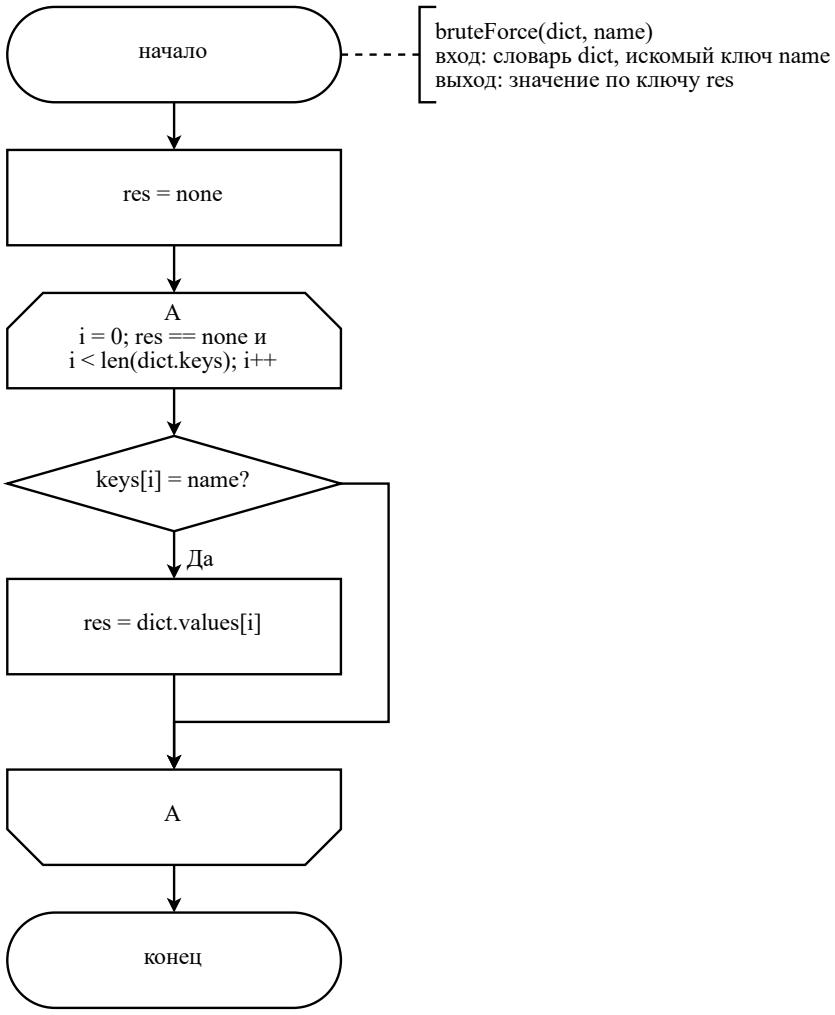

Рисунок 2.1 – Схема алгоритма поиска полным перебором

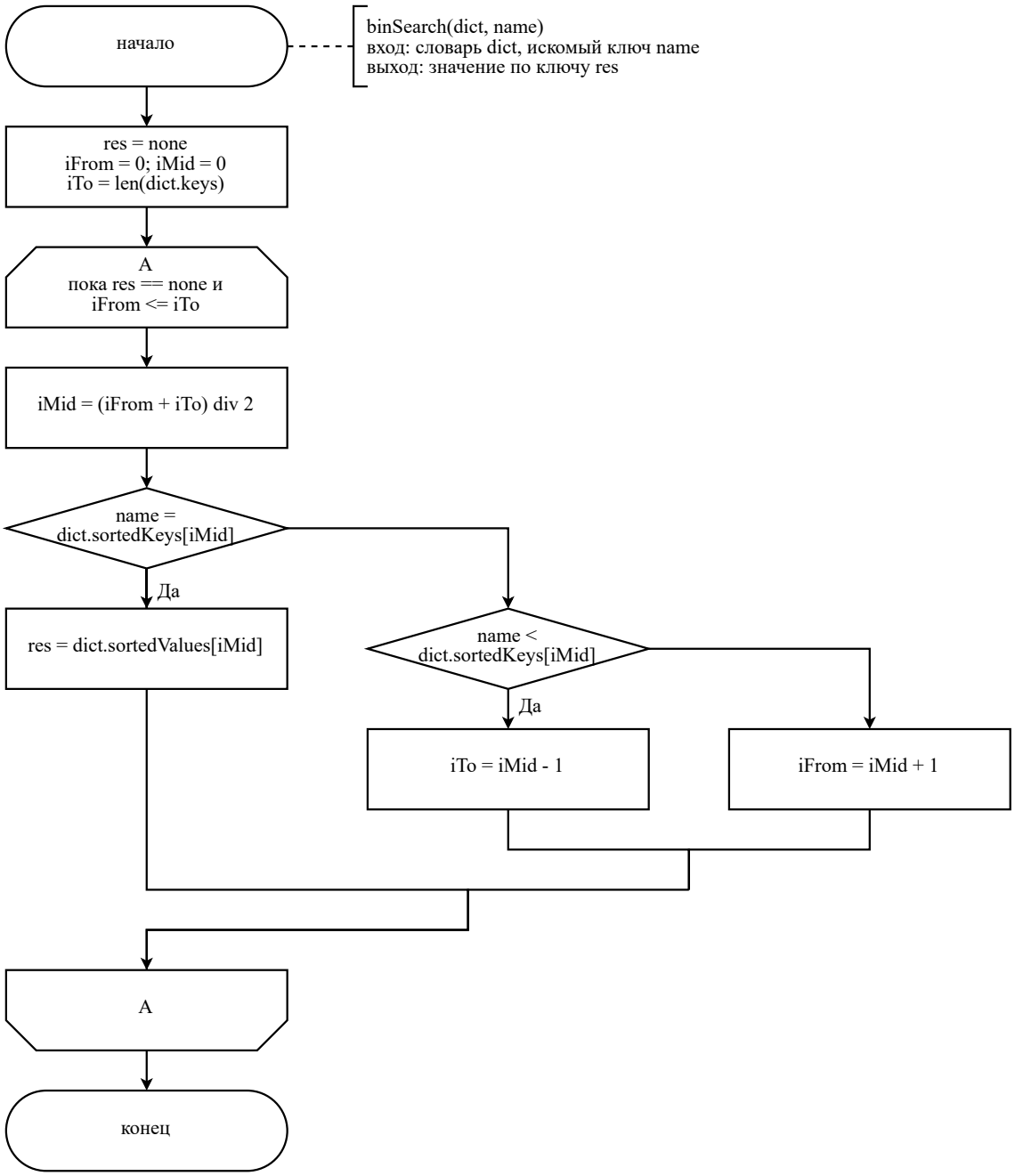

Рисунок 2.2 – Схема алгоритма бинарного поиска

2.2 Структура разрабатываемого ПО

Для реализации взаимодействия с пользователем будет использован метод структурного программирования. Обработка каждого пункта меню будет представлена отдельной функцией, при необходимости будут выделены подпрограммы для каждой из них. Будут реализованы функции для ввода-вывода и функция, вызывающая все подпрограммы для связности и полноценности программы. Также будет реализован класс данных словарь, который будет содержать методы, соответсвующие разработанным алгоритмам.

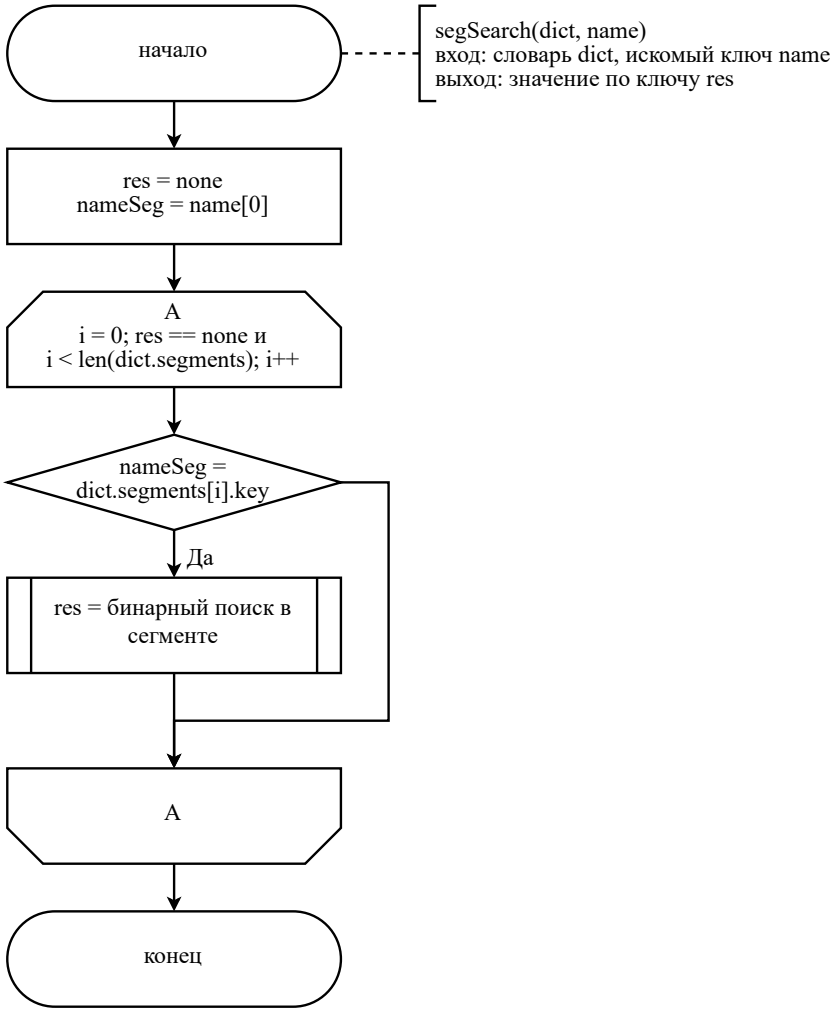

Рисунок 2.3 – Схема алгоритма частотного анализа

2.3 Классы эквивалентности при тестировании

Для тестирования программного обеспечения во множестве тестов будут выделены следующие классы эквивалентности:

- ключа нет в словаре;
- ключ в словаре;
- ключ число, соответсвующее названию игры.

2.4 Вывод

В данном разделе были разработаны алгоритмы поиска в словаре: перебором, бинарный и частотный анализ. Для дальнейшей проверки правильности работы программы были выделены классы эквивалентности тестов.

3 Технологическая часть

В данном разделе описаны требования к программному обеспечению, средства реализации, приведены листинги кода и данные, на которых будет проводиться тестирование.

3.1 Требования к программному обеспечению

Программа должна предоставлять следующие возможности:

- ввода имени файла словаря;
- ввода искомого ключа;
- вывода значения по искомому ключу или сообщения, что такого ключа нет;
- получения времени поиска каждого ключа каждым алгоритмом;
- получения количества сравнений при поиске каждого ключа каждым алгоритмом.

3.2 Средства реализации

Для реализации данной лабораторной работы выбран интерпретируемый язык программирования высокого уровня Python[3], так как он позволяет реализовывать сложные задачи за кратчайшие сроки за счет простоты синтаксиса и наличия большого количества подключаемых библиотек.

В качестве среды разработки выбран текстовый редактор Vim[4] с установленными плагинами автодополнения и поиска ошибок в процессе написания, так как он реализует быстрое перемещение по тексту программы и простое взаимодействие с командной строкой.

Замеры времени проводились при помощи функции process_time_ns из библиотеки time[5].

3.3 Листинги кода

В данном подразделе представлены листинги кода алгоритмов:

– поиск полным перебором (листинг 3.1);

– бинарный поиск (листинги 3.2-3.3);

– частотный анализ (листинги 3.4-3.6).

Листинг 3.1 – Поиск полным перебором

```
    def bruteForce(self, name):
        compNum = 0

        for i, key in enumerate(self.keys):
            compNum += 1

            if key == name:
                return self.values[i], compNum

        return None, compNum
```

Листинг 3.2 – Сортировка для бинарного поиска

```
    def __getSort(self):
        forSort = zip(self.keys, self.values)
        sortedDict = sorted(forSort, key=lambda el: el[0])
        self.sortedKeys = [el[0] for el in sortedDict]
        self.sortedValues = [el[1] for el in sortedDict]
```

Листинг 3.3 – Бинарный поиск

```
    def binSearch(self, name):
        compNum = 0

        iFrom, iTo, iMid = 0, len(self.keys), 0

        while iFrom <= iTo:
            iMid = (iFrom + iTo) // 2

            compNum += 1
            if name == self.sortedKeys[iMid]:
                return self.sortedValues[iMid], compNum

            compNum += 1
            if name < self.sortedKeys[iMid]:
                iTo = iMid - 1
            else:
```

Листинг 3.3 (продолжение)

```
17                 iFrom = iMid + 1
18
19         return None, compNum
```

Листинг 3.4 – Разбиение словаря на сегменты

```
1     def __getSegments(self):
2         freqVals = [[ch, 0] for ch in self.segmMarkers + '@']
3
4         for key in self.keys:
5             segMarker = key[0] if key[0] in self.segmMarkers else '@'
6             for j, freq in enumerate(freqVals):
7                 if freq[0] == segMarker:
8                     freqVals[j][1] += 1
9
10        freqVals.sort(key=lambda el: el[1], reverse=True)
11
12        self.segments = [(freqVal[0], {'keys':[], 'vals':[]})
13                         for freqVal in freqVals]
14
15        for i, key in enumerate(self.sortedKeys):
16            segMarker = key[0] if key[0] in self.segmMarkers else '@'
17
18            for j, segment in enumerate(self.segments):
19                if segment[0] == segMarker:
20                    self.segments[j][1]['keys'].append(key)
21                    self.segments[j][1]['vals'].append(self.sortedValues[i])
```

Листинг 3.5 – Поиск в словаре, разбитом на сегменты

```
1     def segSearch(self, name):
2         compNum = 0
3
4         nameSeg = name[0] if name[0] in self.segmMarkers else '@'
5
6         for i, segment in enumerate(self.segments):
7             compNum += 1
8
9             if nameSeg == segment[0]:
10                vals, tmpCompNum = self.__segBinSearch(i, name)
11                compNum += tmpCompNum
12
13                return vals, compNum
14
15        return None, compNum
```

Листинг 3.6 – Бинарный поиск в сегменте

```
    def __segBinSearch(self, segNum, name):
        compNum = 0
        keys = self.segments[segNum][1]['keys']
        vals = self.segments[segNum][1]['vals']

        iFrom, iTo, iMid = 0, len(keys), 0

        while iFrom <= iTo:
            iMid = (iFrom + iTo) // 2

            compNum += 1
            if name == keys[iMid]:
                return vals[iMid], compNum

            compNum += 1
            if name < keys[iMid]:
                iTo = iMid - 1
            else:
                iFrom = iMid + 1

        return None, compNum
```

3.4 Описание тестирования

В таблице 3.1 приведены функциональные тесты программы.

Таблица 3.1 – Функциональные тесты

Ключ	Ожидаемый результат
j	Нет такой игры!
Munchkin	{2001, 41605, 5.9}
1001	{1900, 31, 6.43}

3.5 Вывод

В данном разделе были реализованы алгоритмы поиска в словаре: перебором, бинарный и частотный анализ. Также были написаны тесты для каждого класса эквивалентности, описанного в конструкторском разделе.

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

– Операционная система: Manjaro [6] Linux x86_64.

– Память: 8 GiB.

– Процессор: Intel® Core™ i5-8265U, 4 физических ядра, 8 логических ядра[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 Примеры работы программы

На рисунке **??** представлены результаты работы последовательной обработки, на рисунке **??** – конвейерной.

4.3 Результаты тестирования

Программа была протестирована на входных данных, приведенных в таблице 3.1. Полученные результаты работы программы совпали с ожидаемыми результатами.

4.4 Постановка эксперимента по замеру времени

Для оценки времени работы последовательной и конвейерной реализации алгоритмов шифрования был проведен эксперимент, в котором определялось влияние количества заявок и количества слов в сообщении на время работы каждого из алгоритмов. Тестирование проводилось на количестве заявок от 5 до 10 с шагом 5, от 25 до 100 с шагом 25 и от 100 до 1000 с

шагом 250, а количество слов принимало значения $5, 10, 25, 50, 75, 100$. Время работы на каждом из значений было получено с помощью *бенчмарков*[8], являющимися встроенными средствами языка Go. В них количество повторов измерений времени изменяется динамически до тех пор, пока не будет получен стабильный результат.

Результаты эксперимента были представлены в виде таблиц и графиков, приведенных в следующем подразделе.

4.5 Результаты эксперимента

В таблице **??** представлены результаты измерения времени работы последовательной и конвейерной реализаций в зависимости от числа заявок. На рисунке **??** представлен соответствующий график.

В таблице **??** представлены результаты измерения времени работы последовательной и конвейерной реализаций в зависимости от числа слов в сообщениях при фиксированном числе заявок, равном 50. На рисунке **??** представлен соответствующий график.

4.6 Вывод

По результатам эксперимента можно сделать следующие выводы:

- при количестве заявок до 10 последовательная и конвейерная реализация генерации зашифрованных сообщений отрабатывают за одинаковое время, а при количестве заявок до 5 последовательная реализация работает 1.3 раза быстрее, что объясняется затратами на передачу данных между лентами с помощью очередей/каналов в конвейерной реализации;
- при большем количестве заявок от 25 конвейерная реализация работает в 1.7 раза быстрее последовательной;
- при фиксированном количестве заявок при различных количествах слов в сообщениях конвейерная реализация работает в 1.7 раза быстрее последовательной.

Таким образом, для обработки количества заявок большего 10 для достижения оптимальной скорости вычислений необходимо использовать конвейерную реализацию. Если работа просходит с количеством заявок до 10 достаточно последовательной реализации, то есть нет необходимости реализовывать более сложный конвейерный алгоритм.

Заключение

В ходе выполения лабораторной работы:

- были описаны и разработаны алгоритмы этапов конвейерной обработки данных;
- были описаны и разработаны линейная и конвейерная обрабоки данных;
- был реализован каждый из описанных алгоритмов;
- по экспериментальным данным были сделаны выводы об эффективности по времени каждого из реализованных алгоритмов;
- были получены зависимости времени работы линейной и конвейерной реализаций от числа и размера заявок.

Таким образом, все поставленные задачи были выполнены, а цель достигнута.

Список литературы

[1] associative array. URL: https://xlinux.nist.gov/dads/HTML/assocarray.html (дата обращения: 12.12.2021).

[2] Макконнелл Дж. Основы современных алгоритмов. М.: Техносфера, 2004.

[3] Welcome to Python [Электронный ресурс]. Режим доступа: https://www.python.org (дата обращения: 12.10.2021).

[4] welcome home : vim online [Электронный ресурс]. Режим доступа: https://www.vim.org/ (дата обращения: 12.10.2021).

[5] Package time [Электронный ресурс]. Режим доступа: https://golang.org/pkg/time/ (дата обращения: 07.12.2021).

[6] Manjaro — enjoy the simplicity [Электронный ресурс]. Режим доступа: https://manjaro.org/ (дата обращения: 17.10.2021).

[7] Процессор Intel® Core™ i5-8265U [Электронный ресурс]. Режим доступа: https://ark.intel.com/content/www/ru/ru/ark/products/149088/intel-core-i5-8265u-processor-6m-cache-up-to-3-90-ghz.html (дата обращения: 17.10.2021).

[8] Testing – The Go Programming Language [Электронный ресурс]. Режим доступа: https://golang.org/pkg/testing/ (дата обращения: 07.12.2021).