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 (ИУ7)**

НАПРАВЛЕНИЕ ПОДГОТОВКИ **09.03.04 Программная инженерия;**

О Т Ч Е Т

по лабораторной работе № 7

Название: Поиск в словаре

Дисциплина: Анализ алгоритмов

Студент ИУ7-52б _____ Кузин А.А.
(Группа) (Подпись, дата) (И.О. Фамилия)

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

Москва, 2020

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель работы: провести сравнительный анализ методов поиска в выбранном словаре.

При выполнении лабораторной работы поставлены такие задачи:

1) реализовать поиск полным перебором и поиск в упорядоченном словаре бинарным поиском;

2) провести частотный анализ;

3) сделать выводы о результатах сравнения.

1 Аналитический раздел

В данном разделе будут даны описания базовых и используемого дополнительного алгоритмов.

При заданном словаре формата ключ-значение, чтобы получить значение сначала нужно дойти до ключа, для этого используются разные методы.

В работе используется словарь уникальных слов из романа Владимира Набокова «Камера обскура», содержащий 6979 вхождений.

1.1 Полный перебор

Идея заключается в последовательном проходе по словарю и сравнении ключа с искомым значением. В результате возможно (N+1) случаев: ключ не найден и N возможных расположений ключа в словаре. Лучший случай: ключ найден за 1 сравнение в начале словаря. Худших случаев 2: за N сравнений элемент не найден, либо он найден на последнем сравнении.

1.2 Двоичный поиск

Для работы этого метода словарь должен быть отсортирован. Метод использует стратегию «разделяй и властвуй», а именно: заданная последовательность делится на две равные части и поиск осуществляется в одной из этих частей, которая потом также делится надвое, и так до тех пор, пока обнаружится наличие искомого элемента или его отсутствие [1].

Найдя средний элемент, и сравнив его значение с искомым, можно сказать, где относительно среднего элемента находится искомый элемент.

При двоичном поиске обход можно представить деревом, поэтому трудоёмкость в худшем случае $\log_2 N$(при спуске от корня до листа).

1.3 Сегментирование и частотный анализ

Можно провести сегментацию словаря – разбить его на определённые сегменты, например по первой букве ключа. Тогда трудоёмкость будет рас-

считываться для двух этапов: трудоёмкость выбора сегмента и трудоёмкость поиска в словаре.

В совокупности с частотным анализом, по частоте использования ключа на реальных данных в обучающей выборке или по частоте первого символа ключа, и сортировкой сегментов по частоте, позволит снизить трудоёмкость поиска более часто встречающихся ключей, снизив трудоёмкость первого этапа. Редко встречающиеся сегменты можно объединить в один.

1.4 Вывод

В данном разделе были рассмотрены методы поиска в словаре, дано их описание.

2 Конструкторский раздел

В данном разделе будут рассмотрены схемы алгоритма полного перебора, бинарного поиска и сегментации.

2.1 Схемы базовых алгоритмов

На рисунке 2.1 представлена схема поиска полным перебором.

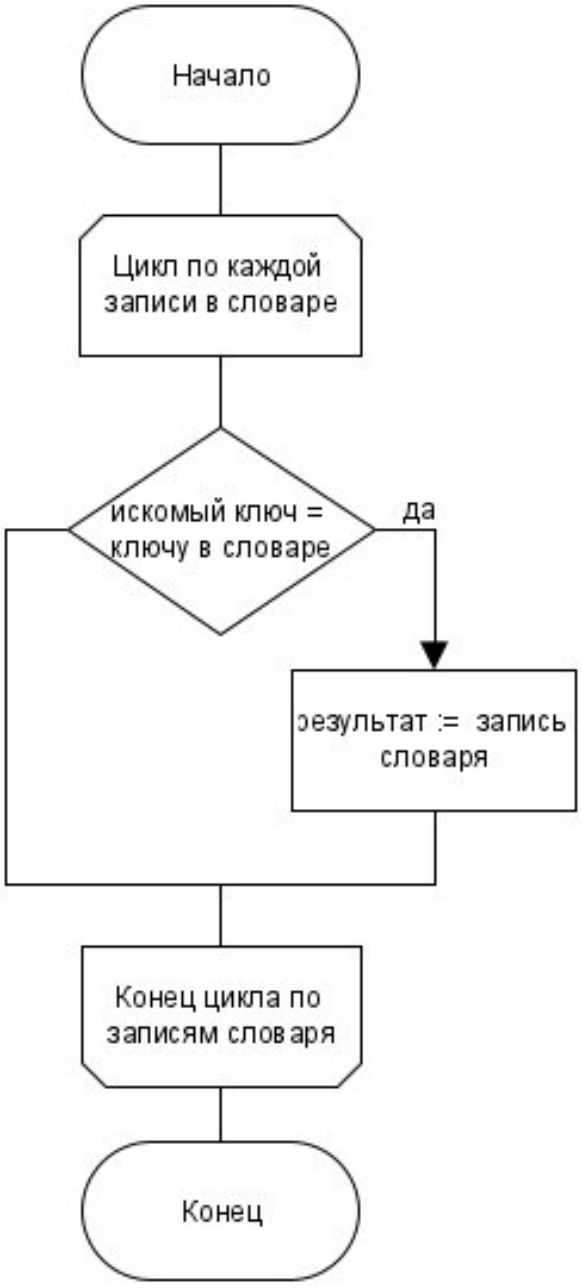

Рисунок 2.1 — Полный перебор

На рисунке 2.2 представлена схема поиска двоичным перебором.

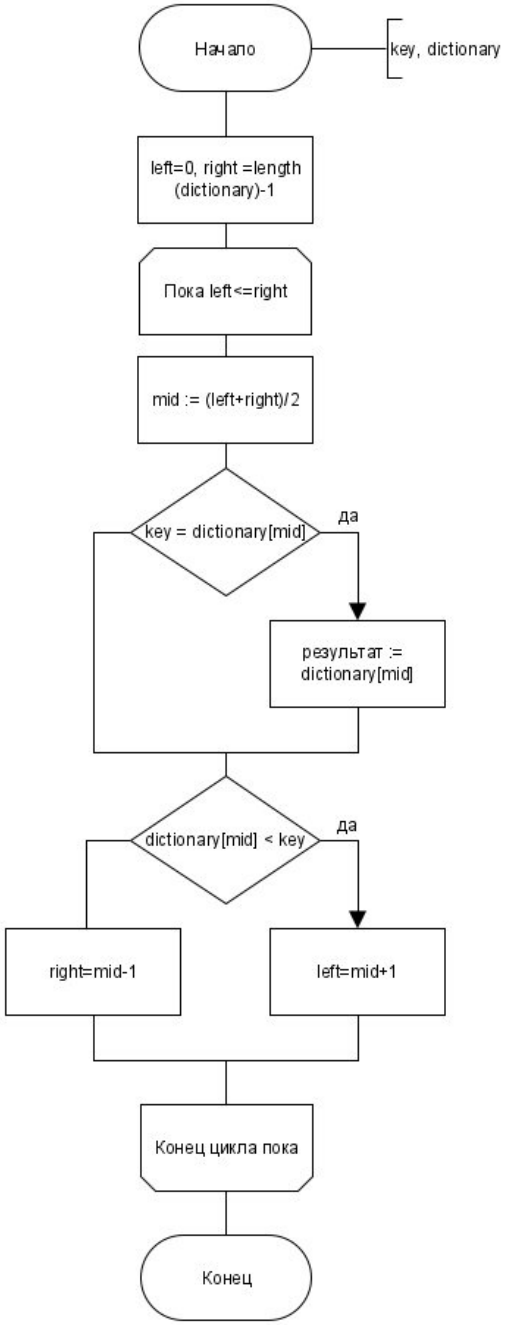

Рисунок 2.2 — Двоичный поиск

2.2 Частотный анализ

В качестве 3-его алгоритма была реализована сегментация вместе с частотным анализом. Частотный анализ проводится по первому символу ключа, как и сегментация. В результате получается множество сегментов, упорядоченных по убыванию частоты первого символа. Наиболее короткие

сегменты объединяются, пока их суммарная длина не будет выше константы. На рисунке 2.3 представлена схема алгоритма сегментирования словаря.

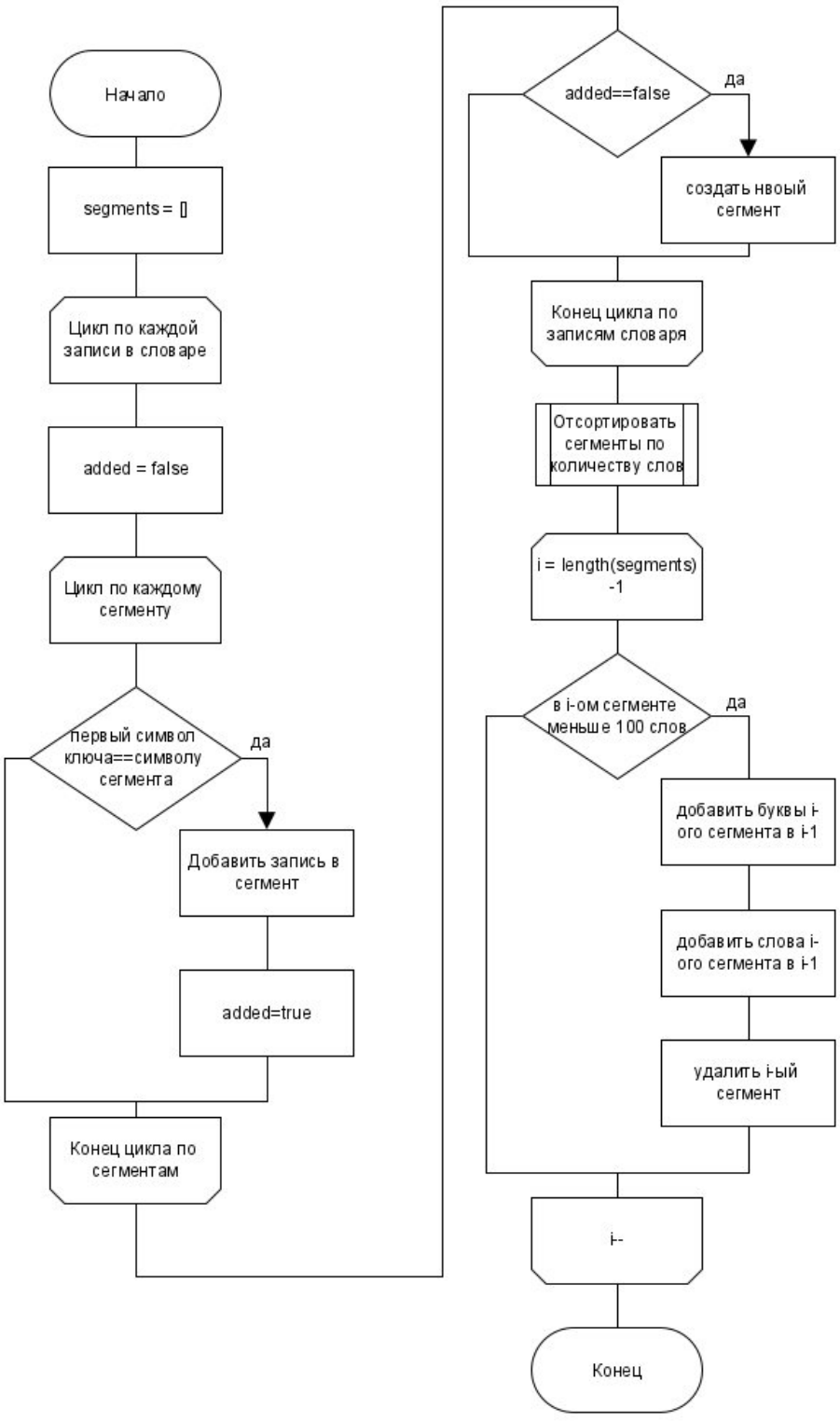

Рисунок 2.3 — Алгоритм сегментации

После чего полученные сегменты используются для поиска, на рисунке 2.4 представлен алгоритм поиска в сегменте.

Рисунок 2.4 — Поиск с использованием сегментов

2.3 Вывод

В данном разделе были рассмотрены схемы алгоритма полного перебора и двоичного поиска, а также сегментирования.

3 Технологический раздел

В данном разделе будут представлены листинги кода реализованных алгоритмов и проведено тестирование.

3.1 Средства реализации

В данной работе используется язык программирования Python. Среда разработки Visual Studio Code. Для замера процессорного времени используется функция perf_counter() из библиотеки time.

Замеры времени были произведены на: Intel(R) Core(TM) i5-8250U CPU @1.60GHz 1.80 Ghz, 4 ядра, 8 логических процессоров.

Для формирования словаря была использована библиотека pymorphy2 [2].

3.2 Листинг кода

В листинге 3.1 приведен код алгоритма полного перебора.

Листинг 3.1 — Алгоритм полного перебора

```
1  def fullsearch(key, dictionary):
2      for record in dictionary:
3          if record[0] == key:
4              return record
5      return None
```

В листинге 3.2 представлен код двоичного поиска.

Листинг 3.2 — Муравьиный алгоритм

```
1  def binsearch(key, dictionary):
2      lb = 0
3      rb = len(dictionary) - 1
4      while lb <= rb :
5          mid = (lb + rb) // 2
6          if dictionary[mid][0] == key:
7              return dictionary[mid]
8          elif dictionary[mid][0] < key:
9              lb = mid + 1
10         else:
```

```
11                rb = mid - 1
12        return None
```

В листинге 3.3 представлен код сегментации и поиска в сегментах.

Листинг 3.3 — Класс муравья

```
1  def statseg(dictionary):
2      segments = []
3      for record in dictionary:
4          added = False
5          for i in range(len(segments)):
6              if record[0][0] == segments[i][0][0]:
7              segments[i][1].append(record)
8              added = True
9          if not added:
10             segments.append([[record[0][0]], [record]])
11     segments.sort(key=lambda record: len(record[1]), reverse=True)
12
13     i = len(segments) - 1
14     while i > 0:
15         if len(segments[i][1]) < 100:
16             for letter in segments[i][0]:
17                 segments[i - 1][0].append(letter)
18             for word in segments[i][1]:
19                 segments[i - 1][1].append(word)
20             segments.pop(i)
21         i -= 1
22     segments.sort(key=lambda record: len(record[1]), reverse=True)
23     for segment in segments:
24         segment[1].sort(key=lambda record: record[0])
25     return segments
26
27
28 def segmentation(key, segments):
29     for segment in segments:
30         if key[0] in segment[0]:
31             return binsearch(key, segment[2])
```

3.3 Тестирование

Были проведены тесты с первым, последним, средним словом и словом, отсутствующим в словаре.

Таблица 3.1 — Результаты тестирования

	ёкнуть	**абажур**	**пролетать**	—
Полный перебор	['ёкнуть', 6978]	['абажур', 0]	['пролетать', 4768]	None
Двоичный поиск	['ёкнуть', 6978]	['абажур', 0]	['пролетать', 4768]	None
С сегментированием	['ёкнуть', 6978]	['абажур', 0]	['пролетать', 4768]	None

Все тесты пройдены успешно.

3.4 Вывод

В данном разделе были рассмотрены листинги кода реализованных алгоритмов и проведено тестирование.

4 Исследовательский раздел

В данном разделе будет проведено сравнение алгоритмов и сделаны выводы.

4.1 Анализ результатов

В результате статического анализа было определено, что больше всего слов начинающихся на букву 'п', их 1228, на втором месте на букву 'с', в среднем размер сегментов примерно от 500 до 200 слов, самые короткие сегменты – 'ё','й','ю','щ' размером менее 20 слов и также 10 сегментов размером менее 100 символов.

Для поиска всех слов из словаря словарь изначально был перемешан, после чего в цикле по каждому слову производился поиск, для каждого алгоритма время замерялось для 20 циклов поисков всех слов, после чего полученное время делилось на 20. Для алгоритма с сегментированием время на выделение сегментов учитывалось только 1 раз для каждого цикла поисков.

При расчёте максимального и минимального и времени отсутствующего слова чтобы получить значимую часть время замерялось для 50 запусков, после чего результат делился на 50. В таблице 4.1 представлены результаты измерений, время представлено в секундах.

Таблица 4.1 — Результаты измерений времени

	В среднем	**max**	**min**	**Отсутствующее**
Полный перебор	0.882	0.0005	2e-06	2.7e-4
Двоичный поиск	0.02	1.26e-5	3.5e-7	2.6e-6
С сегментированием	0.036	5.5e-6	5.5e-7	3e-6

Из результатов видно, что наиболее быстрые методы – двоичный поиск и поиск с сегментированием, также в связи с тем, что в сегментировании

учитывается частота встречи букв, то шанс того, что будет поиск слова на 'п' из словаря выше, а для них время поиска должно быть меньше. Также следует заметить что максимальное время поиска для алгоритма с сегментированием на порядок меньше.

4.2 Вывод

Для наиболее быстрого поиска следует использовать или двоичный поиск или поиск с сегментированием, который учитывает особенности задачи, учитывая частотный анализ.

ЗАКЛЮЧЕНИЕ

В результате лабораторной работы цель была достигнута – проведён сравнительный анализ методов полного перебора, двоичного поиска и поиска с сегментированием. Все задачи были выполнены: реализован поиск полным перебором и двоичный поиск, проведён частотный анализ, сделаны выводы о результатах сравнения алгоритмов.

Стало ясно, что двоичный поиск и поиск с использованием частотного анализа дают результаты лучше, чем полный перебор, во всех случаях, при этом максимальное время ниже у поиска с частотным анализом и сегментированием, что связано с учётом специфики задачи за счёт расчёта вероятностей встретить тот или иной сегмент.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Двоичный (бинарный) поиск[Электронный ресурс]. — 2018. — Режим доступа: `https://kvodo.ru/dvoichnyiy-poisk-2.html` (дата обращения: 26.12.2020).

2. Korobov Mikhail. Морфологический анализатор pymorphy2[Электронный ресурс]. — 2020. — Режим доступа: `https://pymorphy2.readthedocs.io/en/stable/` (дата обращения: 25.12.2020).