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»

Лабораторная работа № 7

Дисциплина Анализ алгоритмов

Тема Поиск в словаре

Студент Мишина Е.В.

Группа ИУ7-54Б

Преподаватель Волкова Л. Л. Строганов Ю.В.

Москва — 2020 г.

Оглавление

Введение

Структура данных, позволяющая идентифицировать ее элементы не по числовому индексу, а по произвольному, называется словарем или ассоциативным массивом. Каждый элемент словаря состоит из двух объектов: ключа и значения.

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]. В данной лабораторной работе будут рассмотренны и реализованы такие алгоритмы поиска в словаре как:

1) полный перебор;
2) бинарный поиск;
3) поиск по сегментам.

Цель работы: изучить и применить на практике алгоритмы поиска по словарю.

Задачи работы:

1) Описать и реализовать алгоритмы полного перебора, двоичного поиска, поиска по сегментам.
2) Провести сравнительный анализ алгоритмов поиска по словарю.
3) Сделать выводы о применимости алгоритмов к решению задачи поиска по словарю.

1. Аналитическая часть

В данном разделе содержится описание алгоритмов поиска по словарю.

1.1. Алгоритм полного перебора

Алгоритмом полного перебора называют метод решения задачи, при котором по очереди рассматриваются все возможные варианты исходного набора данных [2]. 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 решения может стремиться к экспоненциальному времени работы.

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 двоичного поиска

Алгоритм двоичного поиска применяется к заранее упорядоченному словарю.

При бинарном поиске искомый ключ сравнивается с ключом среднего элемента в словаре. Если они равны, то поиск успешен. В противном случае поиск осуществляется аналогично в левой или правой частях словаря [3].

На каждом шаге осуществляется поиск середины отрезка по формуле (1.2).

$$mid = \frac{left + right}{2} \tag{1.2}$$

Если искомый элемент равен элементу с индексом mid, поиск завершается. В случае если искомый элемент меньше элемента с индексом mid, на место mid перемещается правая граница рассматриваемого отрезка, в противном случае — левая граница.

Поиск в словаре с использованием данного алгоритма в худшем случае (необходимо спуститься по двоичному дереву от корня до листа) будет иметь трудоемкость $O(log_2 N)$, что

быстрее поиска при помощи алгоритма полного перебора. Но стоит учитывать тот факт, что данный алгоритм работает только для заранее упорядоченного словаря. В случае большого объема данных и обратного порядка сортировки может произойти так, что алгоритм полного перебора будет эффективнее по времени, чем алгоритм двоичного поиска.

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 некоторым общим признаком попадают в один сегмент (для строк это может быть первая буква, для чисел - остаток от деления) [4].

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 $O(\log_2 n)$, где n - 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 (1.3).

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

1.4. Вывод

В данном разделе были рассмотрены алгоритмы поиска по словарю.

2. Конструкторская часть

В данном разделе содержатся схемы алгоритмов решения поиска по словарю и представлена структура рассматриваемого словаря.

2.1. Структура словаря

Словарь состоит из пар вида `<number - type>`, где `number` - номер счета, `type` - тип оплаты.

2.2. Схемы алгоритмов

На рисунке 2.1 представлена схема алгоритма полного перебора.

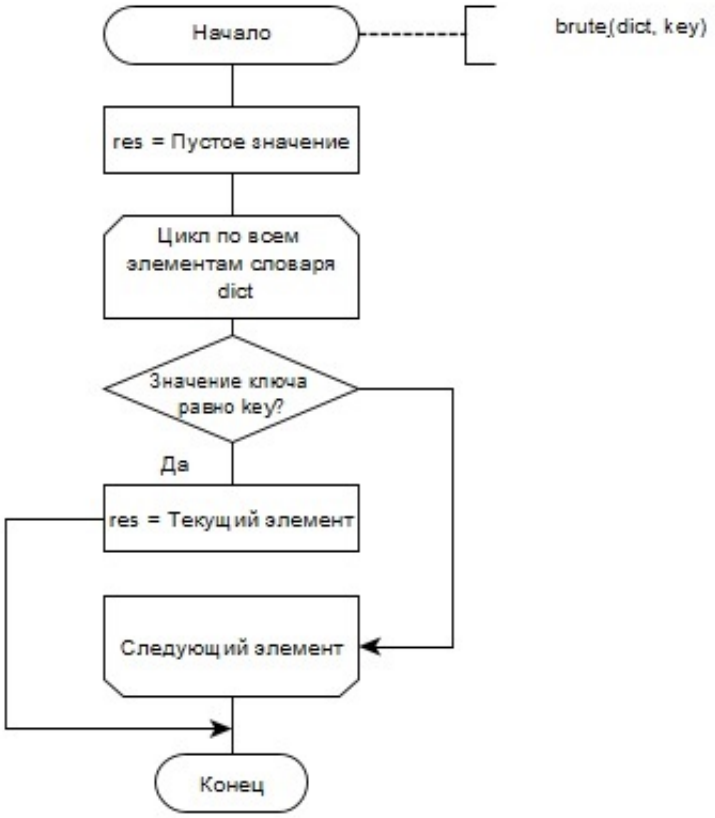

Рис. 2.1. Схема алгоритма полного перебора.

На рисунке 2.2 представлена схема алгоритма двоичного поиска.

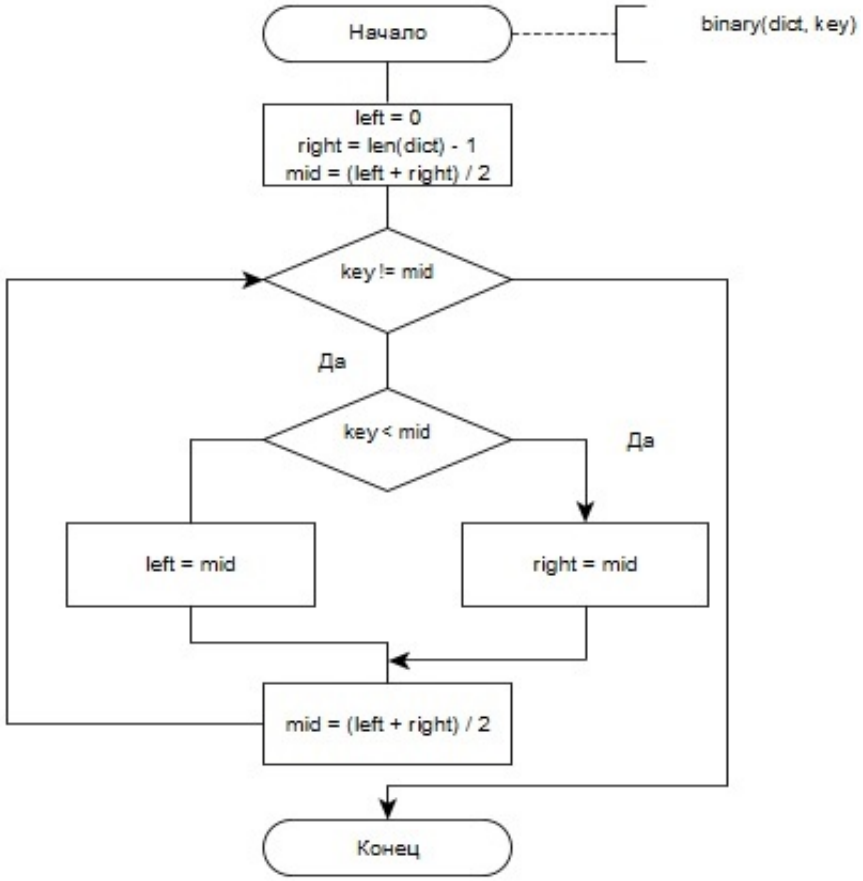

Рис. 2.2. Схема алгоритма двоичного поиска.

На рисунке 2.3 представлена схема алгоритма с использованием частотного анализа.

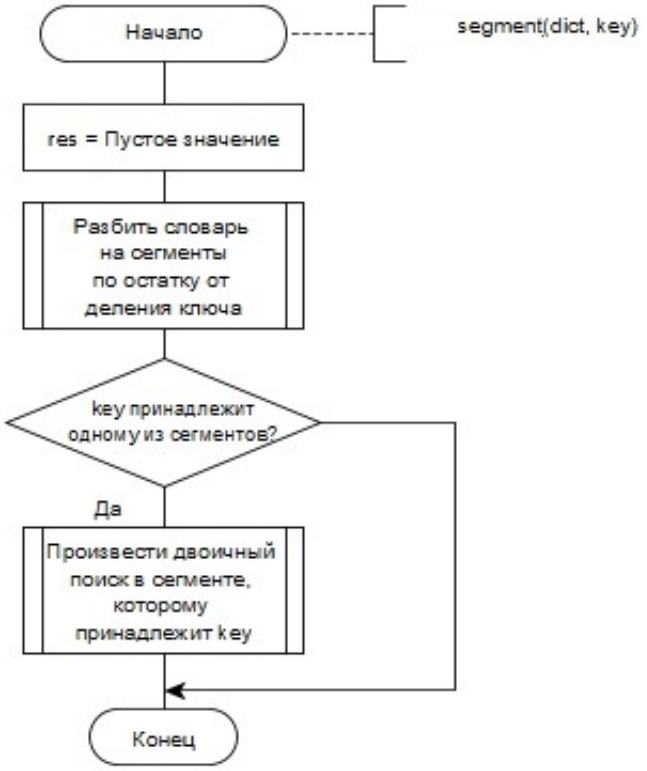

Рис. 2.3. Схема алгоритма с использованием частотного анализа.

2.3. Вывод

В данном разделе были представлены схемы алгоритмов поиска по словарю.

3. Технологическая часть

В данном разделе будут рассмотрены требования к программному обеспечению, средства реализации, представлен листинг кода, функциональное тестирование.

3.1. Требования к программному обеспечению

Требования к программе:

- на вход подаётся словарь и значение ключа;
- на выходе — элемент словаря, значение ключа которого равно введенному.

3.2. Средства реализации

В качестве языка программирования был выбран Python т.к. я хорошо знаю этот язык, он простой и лаконичный, имеющий немногословный и понятный синтаксис, похожий на псевдокод, обладающий сильной динамической типизацией, которая способствует быстрому написанию кода.

Время работы алгоритмов было замерено с помощью функции process_time() из библиотеки time [5].

Для тестирования использовался компьютер на базе процессора Intel(R) Core(TM) i5-4200U, 2 ядра, 4 логических процессоров.

3.3. Листинг кода

В листинге 3.1 представлена реализация поиска по словарю с помощью алгоритма полного перебора.

Листинг 3.1. Реализация поиска по словарю с помощью алгоритма полного перебора.

```
def brute(dictionary, number):
    for record in dictionary:
        if record['number'] == number:
            return record
    return None
```

В листинге 3.2 представлена реализация алгоритма двоичного поиска.

Листинг 3.2. Реализация алгоритма двоичного поиска.

```
def binary(dictionary, number):
    left = 0
    right = len(dictionary) - 1

    if dictionary[left]['number'] > number or dictionary[right]['number'] <
       number:
        return None

    if dictionary[left]['number'] == number:
        return dictionary[left]
    if dictionary[right]['number'] == number:
        return dictionary[right]

    mid = (left + right) // 2
    res = dictionary[mid]['number']
    while number != res:
        if number < res:
            right = mid
        elif number > res:
            left = mid
        mid = (left + right) // 2
        res = dictionary[mid]['number']
    return dictionary[mid]
```

В листинге 3.3 представлена реализация алгоритма поиска по словарю с использованием частотного анализа. Словарь разбивается на сегменты так, что все элементы с одинаковым остатком от деления ключа попадают в один сегмент.

Листинг 3.3. Реализация алгоритма поиска по словарю с использованием частотного анализа.

```
def divide_dict(dictionary, segment_count):
    segment_list = [[] for _ in range(segment_count)]
    for record in dictionary:
        segment_list[record['number'] % segment_count].append(record)
    return segment_list


def segment(segment_list, number):
    if len(segment_list) == 0:
        return None
    return binary(segment_list[number % len(segment_list)], number)
```

3.4. Функциональное тестирование

Реализовано функциональное тестирование. Полученные результаты функций сравниваются с контрольными значениями.

Тестирование проходит по следующим данным:

- ключ отсутсвует в словаре.
- ключ является первым элементом словаря.
- ключ является последним элементом словаря.
- ключ является произвольным элементом словаря.

В таблице 3.1 представлено тестирование программы.

Таблица 3.1. Функциональное тестирование

Ключ	**Ожидаемый результат**	**Полный перебор**	**Бинар**
’0’	None	None	
’1’	’Личные средства(банковский счет)’	’Личные средства(банковский счет)’	’Личные средст
’1000’	’За счет организации’	’За счет организации’	’За счет
’992’	’Материнский капитал’	’Материнский капитал’	’Материн

Программа успешно прошла все тестовые случаи.

3.5. Вывод

В этом разделе обоснован выбор языка програмирования, описаны технические характеристики,приведены листинги кода реализованных алгоритмов. Программа прошла тестирование и работает правильно.

4. Исследовательская часть

В данном разделе приведены примеры работы программы, а также проведен сравнительный анализ алгоритмов по времени.

4.1. Примеры работы

На рисунках 4.1 - 4.2 приведены примеры работы программы.

```
Введите номер счета: 977
Результат
Поиск полным перебором:
{'number': 977, 'type': 'Материнский капитал'}
Двоичный поиск:
{'number': 977, 'type': 'Материнский капитал'}
Алгоритм с использованием частотного анализа:
{'number': 977, 'type': 'Материнский капитал'}
```

Рис. 4.1. Пример работы программы (введённый ключ существует)

```
Введите номер счета: 0
Результат
Поиск полным перебором:
None
Двоичный поиск:
None
Алгоритм с использованием частотного анализа:
None
```

Рис. 4.2. Пример работы программы (введённый ключ не существует)

4.2. Сравнение времени работы алгоритмов

Для произведения замеров времени выполнения реализации алгоритмов будет использована формула:

$$t = \frac{T}{N} \tag{4.1}$$

где t — среднее время выполнения алгоритма, N — количество замеров, T — время выполнения N замеров. Неоднократное измерение времени необходимо для получения более точного результа. Количество замеров взято равным 100. Для сравнения времени работы алгоритмов использовался словарь, состоящий из 1751 элемента. На рисунке 4.3 представлены графики времени работы алгоритмов поиска по словарю. Индекс ключа указан на горизонтальной оси.

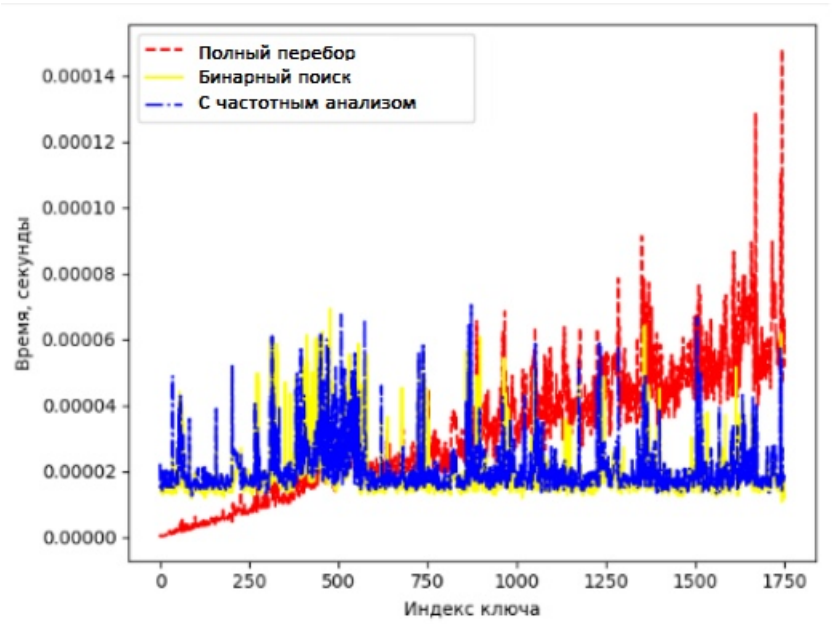

Рис. 4.3. График зависимости времени работы алгоритмов от размера матрицы

Как можно наблюдать на графике, самый медленный алгоритм - алгоритм полного перебора. Время на в нём растет линейно и увеличивается с увеличением индекса элемента словаря. Алгоритм бинарного поиска и алгоритм с использованием частотного анализа трятят примерно одинаковое количество времени, однако стоит учитывать, что они требуют дополнительных расходов времени на подготовку данных к работе с алгоритмом. Однако в лучшем случае алгоритм полного перебора работает в 4 раза быстрее алгоритма двоичного поиска и алгоритма с использованием частотного анализа.

4.3. Вывод

В данном разделе были приведены примеры работы программы, а также проведен сравнительный анализ алгоритмов по времени.

Заключение

Цель работы достигнута. Получены практические навыки реализации алгоритмов поиска, а также проведен анализ времени их работы. Алгоритм полного перебора работает быстро в лучшем случае (когда искомые ключи находятся в самом начале словаря). Однако для работы лучше использовать алгоритм бинарного поиска или алгоритм с использованием частотного анализа. Сравнительный анализ по времени показал, что время их работы примерно одинаково, но алгоритм с использованием частотного анализа выигрывает в случае, если искомые ключи находятся в конце словаря.

Список литературы

1. Словари (ассоциативные массивы) в Python [Электронный ресурс]. - Режим доступа: https://foxford.ru/wiki/informatika/slovari-assotsiativnye-massivy-v-python Дата обращения 30.11.2020.

2. Алгоритмы последовательного поиска [Электронный ресурс]. - Режим доступа: https://cyberpedia.su/9x4e62.html Дата обращения 30.11.2020

3. Бинарный поиск [Электронный ресурс]. Режим доступа: https://prog-cpp.ru/search-binary/ Дата обращения 30.11.2020

4. Алгоритмы программирования [Электронный ресурс]. - Режим доступа: https://otus.ru/nest/post/829/ Дата обращения 30.11.202

5. Официальный сайт Python, документация [электронный ресурс]. Режим доступа: https://docs.python.org/3/library/time.html, свободный (Дата обращения: 16.09.20)