

Государственное образовательное учреждение высшего профессионального образования

"Московский государственный технический университет имени Н.Э.Баумана"

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

**Отчет по лабораторной работе №8
курса Анализ алгоритмов
на тему "Поиск по словарю"**

Студент: Денисенко А. А., группа ИУ7-646

Преподаватели: Волкова Л.Л., Строганов Ю.В.

Москва, 2020 г.

Содержание

Введение	2
1 Аналитическая часть	3
1.1 Линейный поиск	3
1.2 Частотный анализ	3
1.3 Бинарный поиск	3
1.4 Поиск по бинарному дереву	3
2 Конструкторская часть	5
2.1 Частотный анализ	5
2.2 Поиск по бинарному дереву	7
3 Технологическая часть	10
3.1 Требования к программному обеспечению	10
3.2 Средства реализации	10
3.3 Листинг кода	10
4 Экспериментальная часть	12
4.1 Примеры работы программы	12
4.2 Сравнение времени работы алгоритмов	12
Заключение	17

Введение

Одна из наиболее часто встречающихся в программировании задач – поиск. Существует несколько основных вариантов поиска, и для них создано много различных алгоритмов. Целью данной работы является получение навыка реализации алгоритмов эффективного по скорости поиска по словарю. Для достижения поставленной цели необходимо решить следующие задачи:

1. описать алгоритмы поиска по словарю;
2. реализовать проанализированные алгоритмы;
3. привести примеры работы.

1 Аналитическая часть

Далее будут рассмотрены различные подходы к осуществлению поиска в словаре.

1.1 Линейный поиск

Алгоритм линейного поиска представляет собой простой последовательный просмотр массива со сравнением каждого элемента массива a с заданным словом x . Сложность этого алгоритма составляет в лучшем случае $O(1)$, в худшем - $O(n)$.

В связи с малой по сравнению с другими алгоритмами скоростью выполнения, линейный поиск обычно используют только если массив содержит малое количество элементов. Тем не менее, линейный поиск не требует дополнительной памяти или обработки/анализа функции, так что он может работать в потоковом режиме при непосредственном получении данных из любого источника. Также линейный поиск часто используется в виде линейных алгоритмов поиска максимума/минимума.

1.2 Частотный анализ

При работе с естественным языком известно, что разные буквы встречаются в тексте с разной частотой. Опираясь на этот факт, можно улучшить алгоритм линейного поиска, реорганизовав словарь таким образом, чтобы при обходе попасть в более частотный сегмент быстрее. Словарь сортируется от наиболее частотной буквы до наименее. Таким образом, хотя сложность этого алгоритма в лучшем и худшем случаях такая же, как у алгоритма линейного поиска, более вероятные случаи становятся лучшими, а менее - худшими.

1.3 Бинарный поиск

Можно составить более быстрый алгоритм поиска, если данные будут предварительно упорядочены. Основная идея алгоритма заключается в случайном выборе некоторого элемента, предположим a_m , и сравнении его с аргументом поиска x . Если он равен x , то поиск заканчивается, если он меньше x , то делается вывод, что все элементы с индексами, меньшими или равными m , можно исключить из дальнейшего поиска; если же он больше x , то исключаются индексы больше и равные m .

Выбор m совершенно не влияет на корректность алгоритма, но влияет на его скорость выполнения. Очевидно, что чем большее количество элементов исключается на каждом шаге алгоритма, тем этот алгоритм быстрее. Оптимальным решением будет выбор среднего элемента, так как при этом в любом случае будет исключаться половина массива. Сложность бинарного поиска в лучшем случае составляет $O(1)$, а в худшем - $O(\log(n))$.

Так как бинарный поиск требует предварительной сортировки массива, его выгоднее применять в том случае, если обрабатываемый словарь содержит большое количество элементов и поиск осуществляется неоднократно.

1.4 Поиск по бинарному дереву

Бинарное дерево — это иерархическая структура данных, в которой каждый узел имеет значение (оно же является в данном случае и ключом) и ссылки на левого и правого потомка. Узел, находящийся на самом верхнем уровне (не являющийся чьим либо потомком) называется корнем. Узлы, не имеющие потомков, называются листьями.

Бинарное дерево поиска — это бинарное дерево, обладающее дополнительными свойствами: значение левого потомка меньше значения родителя, а значение правого потомка больше значения родителя для каждого узла дерева. То есть, данные в бинарном дереве поиска хранятся в отсортированном виде. При

поиске элемента сравнивается искомое значение с корнем. Если искомое больше корня, то поиск продолжается в правом потомке корня, если меньше, то в левом, если равно, то значение найдено. Пример бинарного дерева представлен на рисунке 1.

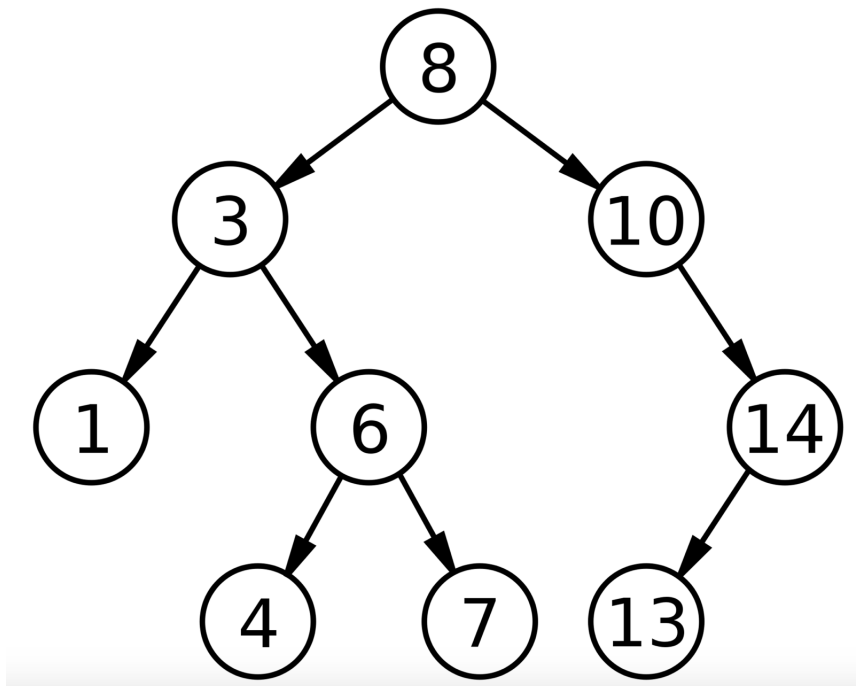


Рисунок 1 : Пример бинарного дерева поиска

Так как алгоритм поиска по бинарному дереву отличается от бинарного поиска лишь используемой структурой данных, его сложность остается той же.

Вывод

Проанализированные в этом разделе алгоритмы можно разделить на две группы: алгоритмы с линейной и логарифмической сложностью. Лишь один из них - линейный - не требует предварительной обработки данных. Остальные, хоть и работают в среднем быстрее, требуют, как минимум, предварительной сортировки. Это значит, что у каждого из описанных алгоритмов есть своя сфера применимости.

2 Конструкторская часть

В этой лабораторной работе будут реализованы алгоритмы частотного анализа и поиска по двоичному дереву.

Будет рассмотрена фиксированная группа данных, в которой необходимо найти заданный элемент. В общем случае множество a из N элементов задано в виде массива, который описывает запись с некоторым полем key , играющим роль ключа. Задача заключается в поиске элемента, ключ которого равен заданному аргументу поиска x . Полученный в результате индекс i , удовлетворяющий условию $a[i].key = x$, обеспечивает доступ к другим полям обнаруженного элемента.

2.1 Частотный анализ

На рисунках 2 и 3 изображены схемы алгоритмов предварительной обработки массива и поиска.

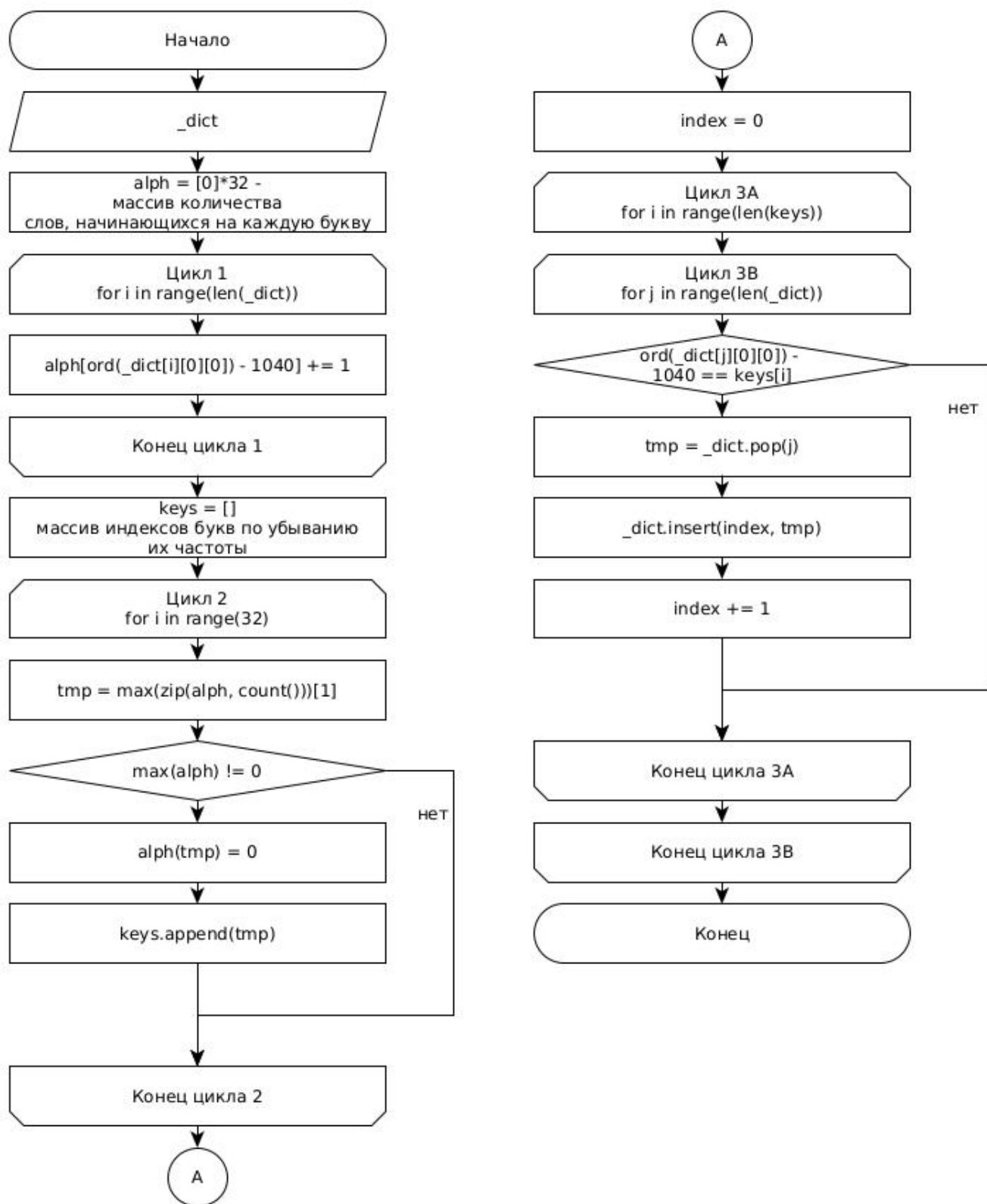


Рисунок 2 : Схема алгоритма обработки массива

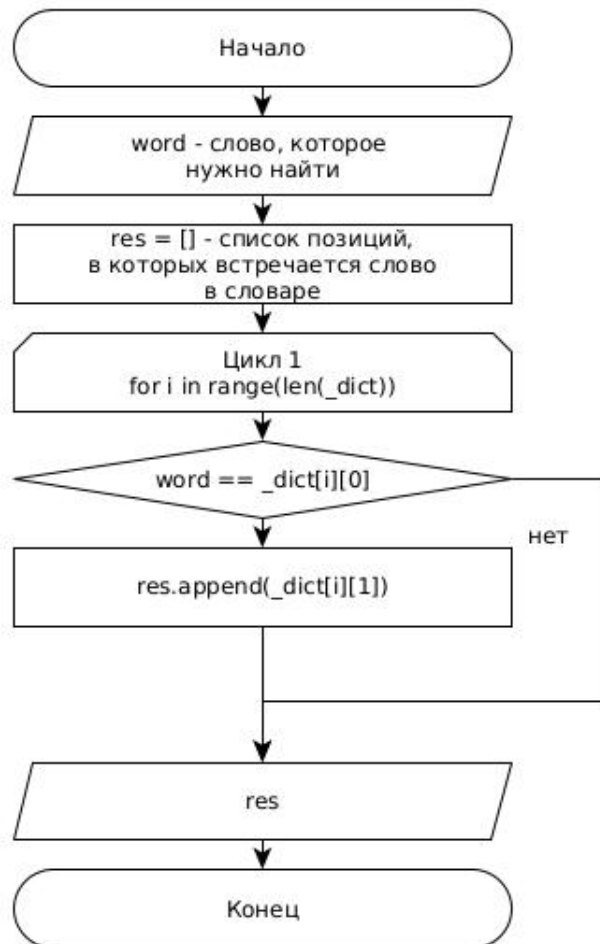


Рисунок 3 : Схема алгоритма линейного поиска

2.2 Поиск по бинарному дереву

На рисунках 4 и 5 изображены схемы алгоритмов построения бинарного дерева и поиска.

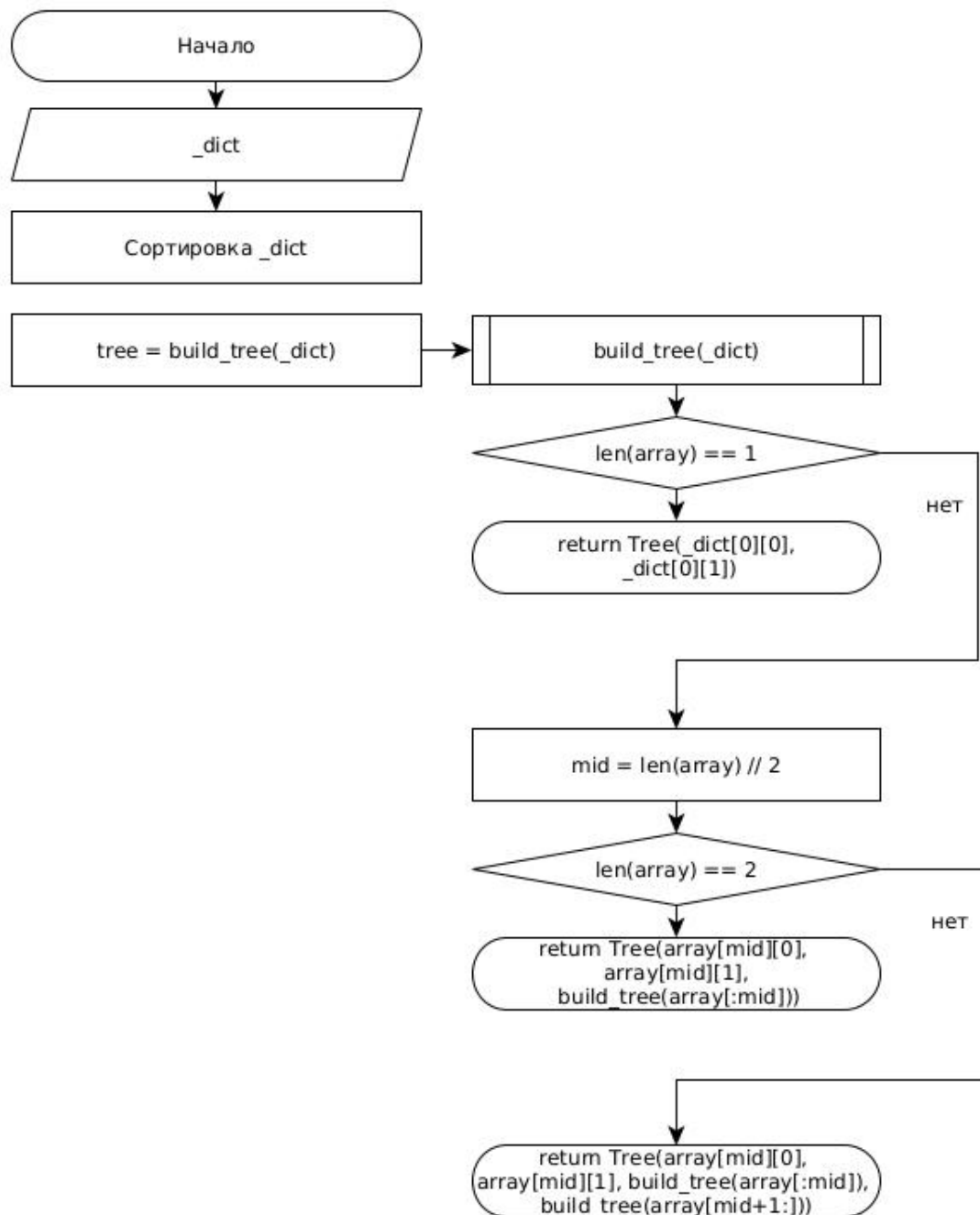


Рисунок 4 : Схема алгоритма построения двоичного дерева

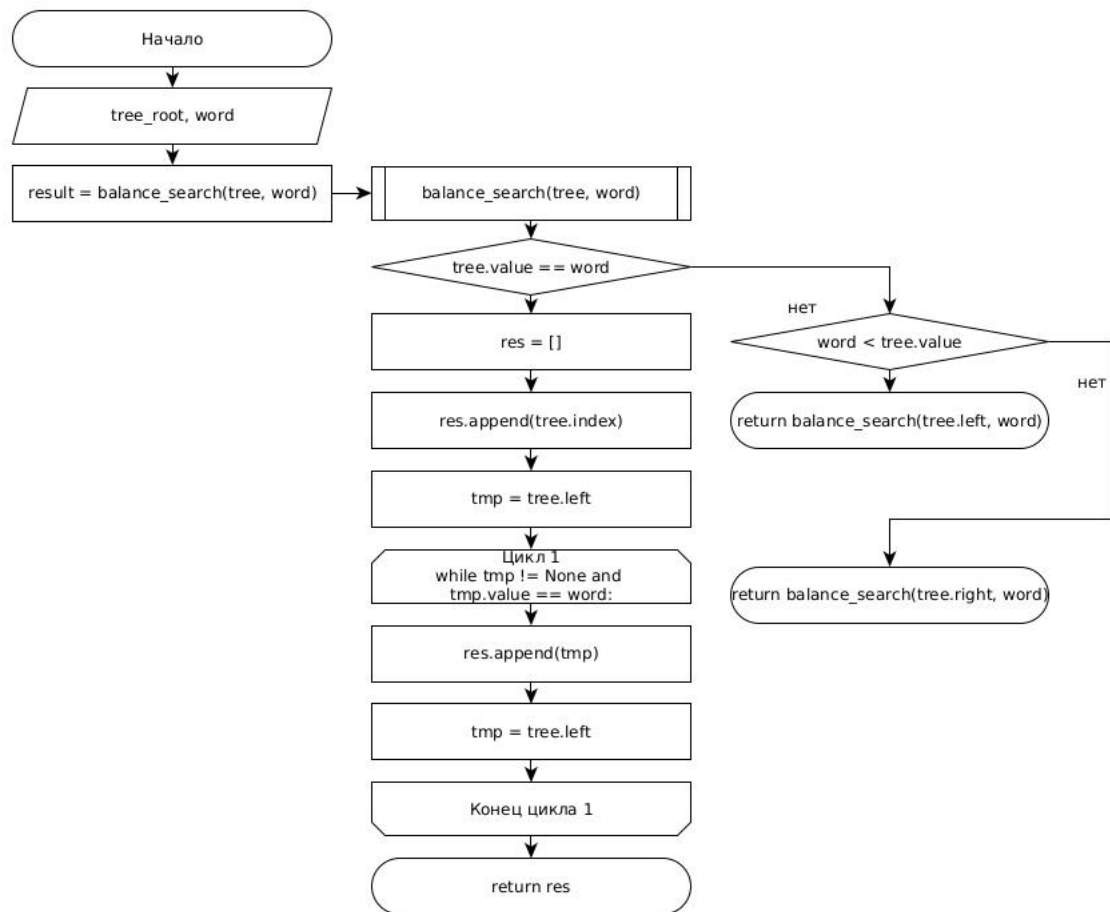


Рисунок 5 : Схема алгоритма поиска по двоичному дереву

Вывод

В этом разделе были приведены схемы алгоритмов частотного анализа и поиска по бинарному дереву. В основе реализации этих алгоритмов лежат цикл и рекурсия соответственно, что влияет на их производительность.

3 Технологическая часть

3.1 Требования к программному обеспечению

Программа должна выводить позицию введенного пользователем слова в массиве. Данные в массив подаются через текстовый файл file.txt. Если слово не было найдено, необходимо вывести сообщение об этом.

3.2 Средства реализации

Для реализации программы был выбран язык Python.

3.3 Листинг кода

Далее приведены листинги алгоритма частотного анализа (листинг 1) и поиска по бинарному дереву (листинг 2).

Листинг 1 Алгоритм частотного анализа

```
1 def freq_search(word, _dict):
2     print( '>_Алгоритм_частотного_анализа' )
3     alph = [0]*32
4     for i in range(len(_dict)):
5         alph[ord(_dict[i][0][0]) - 1040] += 1
6
7     keys = []
8
9     for i in range(32):
10        tmp = max(zip(alph, count()))[1]
11        if max(alph) != 0:
12            alph[tmp] = 0
13            keys.append(tmp)
14    index = 0
15
16    for i in range(len(keys)):
17        for j in range(len(_dict)):
18            if ord(_dict[j][0][0]) - 1040 == keys[i]:
19                tmp = _dict.pop(j)
20                _dict.insert(index, tmp)
21                index += 1
22
23    res = None
24    for i in range(len(_dict)):
25        if word == _dict[i][0]:
26            res = _dict[i][1]
27            break
28
29    if not res:
30        print( "Слово_не_найдено" )
31    else:
32        print( "Слово_было_найдено_в_позиции_=", res )
```

В строках (4)-(6) происходит заполнение массива `alph`, каждый элемент которого соответствует букве русского алфавита (без учета буквы Ё) и является количеством слов в заданном массиве, которые начинаются с этой буквы. Объявленный в строке (7) массив `keys` является вспомогательным при сортировке `_dict` по частоте. Он заполняется индексами букв в порядке убывания их частоты. В строках (16)-(21) происходит сортировка словаря, а в строках (24)-(27) - линейный поиск по отсортированному массиву.

Листинг 2 Алгоритм поиска по бинарному дереву

```

1  class Tree:
2      def __init__(self, value, index, left = None, right = None):
3          self.value = value
4          self.index = index
5          self.left = left
6          self.right = right
7
8      def __str__(self):
9          return str(self.value)
10
11
12 def balance_search(tree, word, res):
13     if tree == None:
14         return res
15     else:
16         if word < tree.value:
17             res = balance_search(tree.left, word, res)
18         elif word > tree.value:
19             res = balance_search(tree.right, word, res)
20         else:
21             res = tree.index
22     return res
23
24
25 def build_tree(array):
26     if len(array) == 1:
27         return Tree(array[0][0], array[0][1])
28     mid = len(array) // 2
29
30     if len(array) != 2:
31         return Tree(array[mid][0], array[mid][1], build_tree(array[:mid]), \
32             build_tree(array[mid+1:]))
33     else:
34         return Tree(array[mid][0], array[mid][1], build_tree(array[:mid]))
35
36 def tree_search(word, _dict):
37     print(' > Поиск по дереву ')
38
39     _dict = sorted(_dict)
40     tree = build_tree(_dict)
41
42     res = None
43     res = balance_search(tree, word, res)

```

```
44     if not res:
45         print("Слово_не_найдено")
46     else:
47         print("Слово_было_найдено_в_позиции_=", res)
```

В функции `tree_search` происходит сортировка массива `_dict`. В функции `build_tree` рекурсивно создаются элементы класса `tree` из среднего элемента поданного на вход массива. В функции `balance_search` осуществляется рекурсивный поиск по созданному бинарному дереву. На вход функции подается корень дерева и искомое слово.

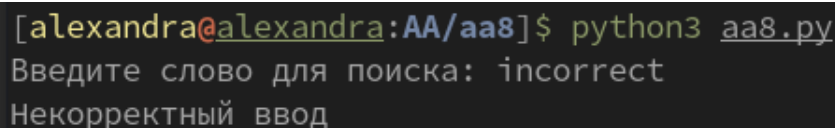
Вывод

В этом разделе были реализованы выбранные алгоритмы с соблюдением всех требований к ПО, приведенных в пункте (3.1).

4 Экспериментальная часть

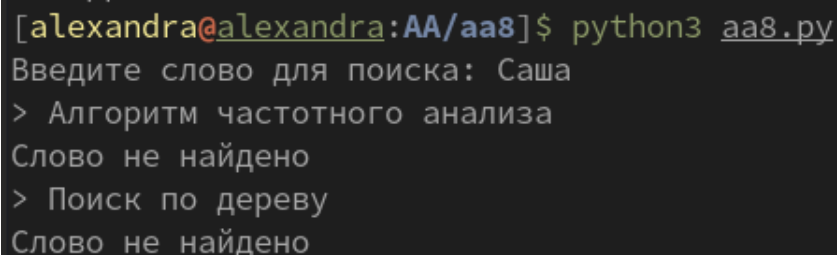
4.1 Примеры работы программы

Примеры работы программы приведены на рисунках 6-9.



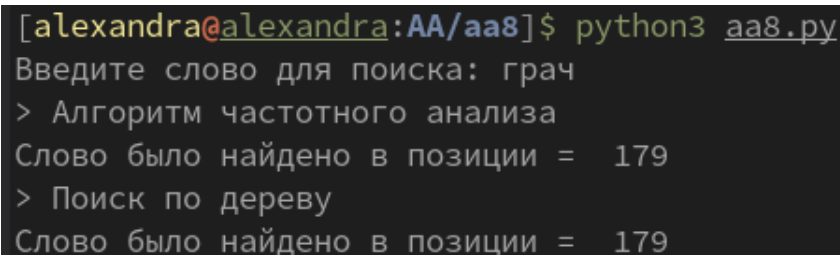
```
[alexandra@alexandra:AA/aa8]$ python3 aa8.py
Введите слово для поиска: incorrect
Некорректный ввод
```

Рисунок 6 : Пример работы программы при некорректном вводе



```
[alexandra@alexandra:AA/aa8]$ python3 aa8.py
Введите слово для поиска: Саша
> Алгоритм частотного анализа
Слово не найдено
> Поиск по дереву
Слово не найдено
```

Рисунок 7 : Пример работы программы со словом, отсутствующим в словаре



```
[alexandra@alexandra:AA/aa8]$ python3 aa8.py
Введите слово для поиска: грач
> Алгоритм частотного анализа
Слово было найдено в позиции = 179
> Поиск по дереву
Слово было найдено в позиции = 179
```

Рисунок 8 : Пример работы программы со словом, которое встречается в словаре однажды

4.2 Сравнение времени работы алгоритмов

В этом разделе будет осуществлено сравнение двух значений: время формирования соответствующей структуры для каждого алгоритма и время поиска для лучшего и худшего случаев. Измерения произво-

дились на процессоре Intel Core i5-8265U.

На рисунке (9) продемонстрирована зависимость времени формирования структуры данных для разных размеров исходного словаря. Были взяты значения от 200 до 1000 с шагом 200.

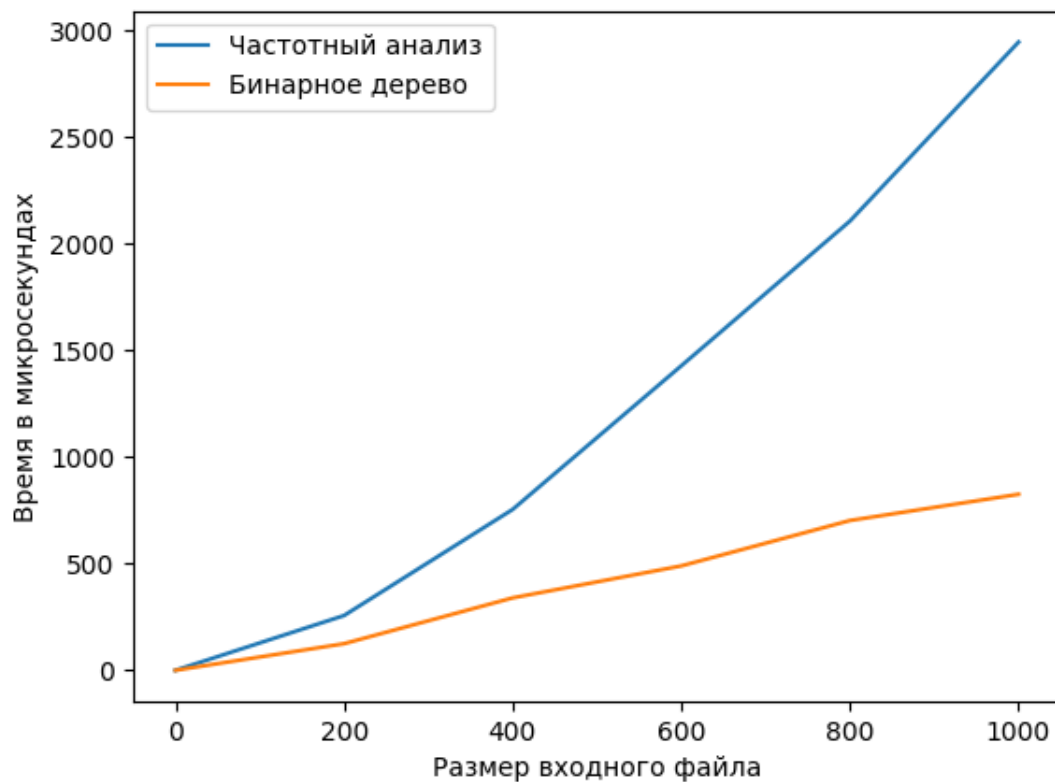


Рисунок 9 : График времени формирования структуры данных

Далее продемонстрировано время работы обоих алгоритмов для их лучших и худших случаев в наносекундах.

Лучший случай для алгоритма частотного анализа: задано первое слово в словаре, начинающееся на самую частотную букву. Худший случай: задано последнее слово в словаре, начинающееся на наименее частотную букву. При обработке массива они окажутся в начале и в конце соответственно.

```
[alexandra@alexandra:AA/aa8]$ python3 aa8_test.py
Введите слово для поиска: Кавра
> Алгоритм частотного анализа
Время поиска (наносекунды): 1267
```

Рисунок 10 : Лучший случай для алгоритма частотного анализа

```
[alexandra@alexandra:AA/aa8]$ python3 aa8_test.py
Введите слово для поиска: Юнко зимний
> Алгоритм частотного анализа
Время поиска (наносекунды): 61543
```

Рисунок 11 : Худший случай для алгоритма частотного анализа

Лучший случай для алгоритма поиска по бинарному дереву: задано слово, находящееся в центре словаря. Худший случай: задано слово, находящееся перед центральным. При формировании дерева поиска первое слово станет корнем дерева, а второе - самым младшим потомком.

```
[alexandra@alexandra:AA/aa8]$ python3 aa8_test.py
Введите слово для поиска: Нырок красноглазый
> Поиск по дереву
Время поиска (наносекунды): 1105
```

Рисунок 12 : Лучший случай для алгоритма поиска по бинарному дереву

```
[alexandra@alexandra:AA/aa8]$ python3 aa8_test.py
Введите слово для поиска: Нырок белоглазый
> Поиск по дереву
Время поиска (наносекунды): 2901
```

Рисунок 13 : Худший случай для алгоритма поиска по бинарному дереву

Далее для тестирования использовались словари длиной от 200 до 1000 слов с шагом 200, содержащие в себе слова длиной в 1 букву. Было проведено сравнение худших случаев для обоих алгоритмов.

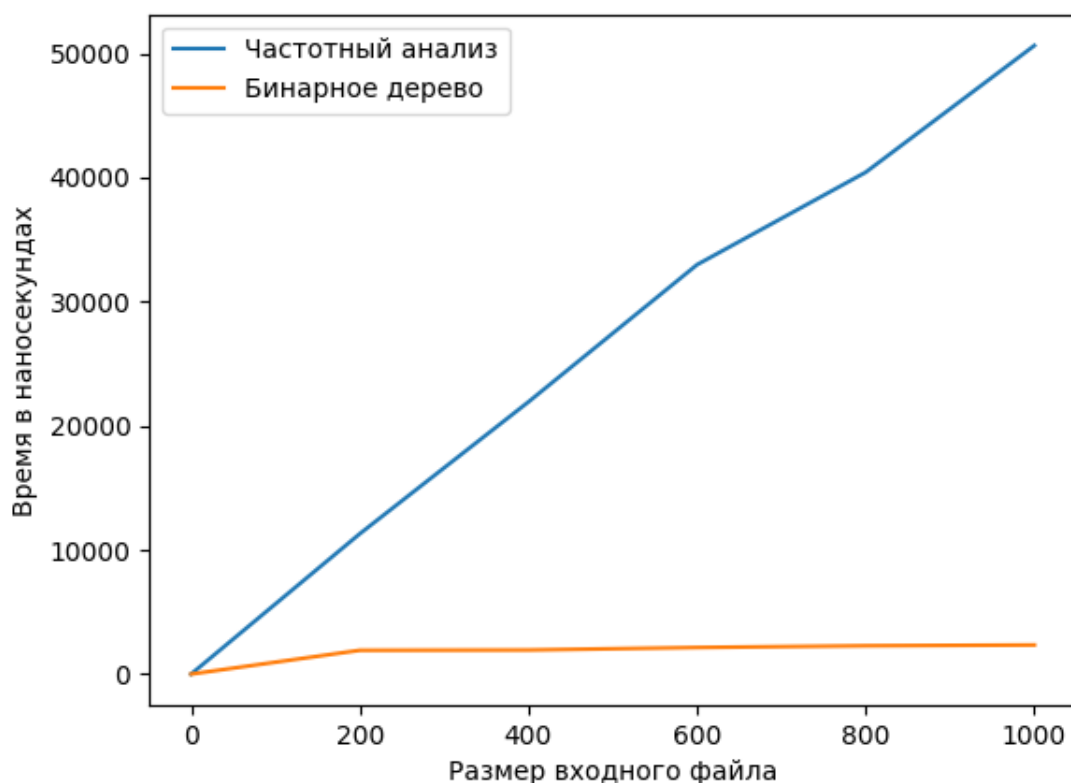


Рисунок 14 : Худшие случаи для разного размера входных данных

Для тестирования на "живом" языке был составлен словарь на основе произведения Л. Н. Толстого "Война и мир". С удаленными повторениями размер входного массива - 43717 элементов. Было проведено

200 экспериментов, в которых на вход программе подавались слова из массива с индексами, кратными 200. Задача эксперимента - выяснить, как алгоритмы покажут себя на приближенном к реальной жизни примере.

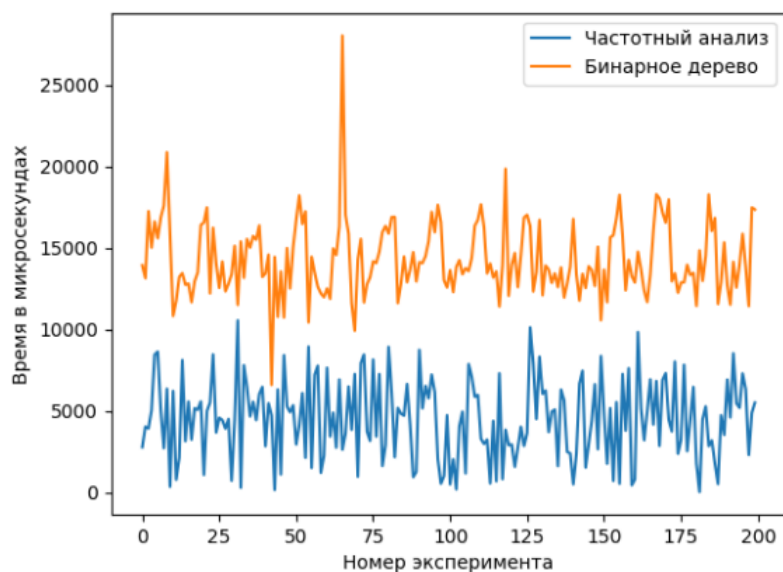


Рисунок 15 : График скорости выполнения программы на 200 экспериментах

Далее приведены индивидуальные результаты работы программы.

```
[alexandra@alexandra:AA/aa8]$ python3 aa8_test.py
Введите слово для поиска: уже
> Алгоритм частотного анализа
Время поиска: 11438
Слово было найдено в позициях = 698
> Поиск по дереву
Время поиска: 11260
Слово было найдено в позициях = 698
```

Рисунок 16 : Пример работы программы на тексте произведения "Война и мир"

```
[alexandra@alexandra:AA/aa8]$ python3 aa8_test.py
Введите слово для поиска: читам
> Алгоритм частотного анализа
Время поиска: 10274
Слово было найдено в позициях = 22219
> Поиск по дереву
Время поиска: 11676
Слово было найдено в позициях = 22219
```

Рисунок 17 : Пример работы программы на тексте произведения "Война и мир"


```
[alexandra@alexandra:AA/aa8]$ python3 aa8_test.py
Введите слово для поиска: спектакль
> Алгоритм частотного анализа
Время поиска: 2411
Слово было найдено в позициях = 11913
> Поиск по дереву
Время поиска: 10404
Слово было найдено в позициях = 11913
```

Рисунок 18 : Пример работы программы на тексте произведения "Война и мир"

Вывод

Было проведено несколько тестов. Один из них - на искусственно сгенерированном словаре, где алгоритмы в худшем случае, как и ожидалось, продемонстрировали линейную и логарифмическую зависимость времени работы от размера входного словаря. Также на вход программе был подан словарь на основе произведения "Война и мир" и частотный анализ в подавляющем числе экспериментов, осуществил поиск быстрее. Кроме того, было продемонстрировано время формирования структуры данных для обоих алгоритмов и время их работы на словаре небольшого объема.

Заключение

В ходе данной работы были изучены алгоритмы линейного поиска, частотного анализа, бинарного поиска и поиска по бинарному сбалансированному дереву. Были реализованы алгоритмы частотного анализа и поиска по бинарному сбалансированному дереву. Тесты показали, что в худшем случае время выполнения алгоритма поиска по бинарному дереву возрастает значительно медленнее, в то время как время выполнения алгоритма частотного анализа возрастает почти линейно. Однако на живом примере в среднем алгоритм частотного анализа справлялся с задачей примерно в 3 раза быстрее, чем алгоритм поиска по бинарному дереву.