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»

Отчёт

по лабораторной работе №7

Название: Поиск в словаре

Дисциплина: Анализ алгоритмов

Студент ИУ7-55Б _____ Хетагуров П.К

(Группа) (Подпись, дата) (И.О. Фамилия)

Преподователь _____ Л.Л. Волкова

(Подпись, дата) (И.О. Фамилия)

Москва, 2020

Содержание

Введение

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

1. полный перебор;
2. двоичный поиск;
3. поиск по сегментам.

1 Аналитическая часть

В данном разделе будут поставлены цели и задачи работы, будут рассмотренны основные теоритические сведения связанные с алгоритмами сортировки.

1.1 Цель и задачи работы

Цель работы: Изучить алгоритмы поиска.

Задачи работы:

1. описать алгоритм полного перебора, двоичного поиска, поиска по сегментам;
2. реализовать описанные алгоритмы;
3. провести эксперименты по замеру времени работы.

1.2 Алгоритм полного перебора

Алгоритм полного перебора подразумевает проверку каждого элемента множества. В случае словаря проверяется каждый элемент словаря на соответствие ключу. Сложность - n.

1.3 Алгоритм двоичного поиска

Алгоритм двоичного поиска осуществляет поиск по упорядоченному множеству объектов.

Двоичный поиск заключается в том, что на каждом шаге множество объектов делится на две части и в работе остаётся та часть множества, где находится искомый объект. Или же, в зависимости от постановки задачи, мы можем остановить процесс, когда мы получим первый или же последний индекс вхождения элемента. Последнее условие — это левосторонний/правосторонний двоичный поиск [1].

1.4 Алгоритм поиска по сегментам

Для применения этого алгоритма множество разбивается на сегменты по определеннуму признаку. При поиске у ключевого элемента определяется сначала этот признак, а затем осуществляется поиск по соответствующему сегменту множества.

1.5 Вывод

В данной части были поставлены задачи и цель работы, описаны алгоритмы полного перебора, двоичного поиска, поиска по сегментам.

2 Конструкторская часть

В данном разделе будут рассмотренны схемы алгоритмов, требования к функциональности ПО.

2.1 Требования к ПО

ПО должно иметь два режима работы, выбираемые из меню

1. Режим демонстрации. В этом режиме должен осуществляться ввод слова и последующий поиск его в словаре с помощью всех алгоритмов.

2. Режим тестирования. В этом режиме должны проводится замеры времени выполнения реализованных алгоритмов.

2.2 Схемы алгоритмов

На рисунке 1 изображена схема алгоритма полного перебора.

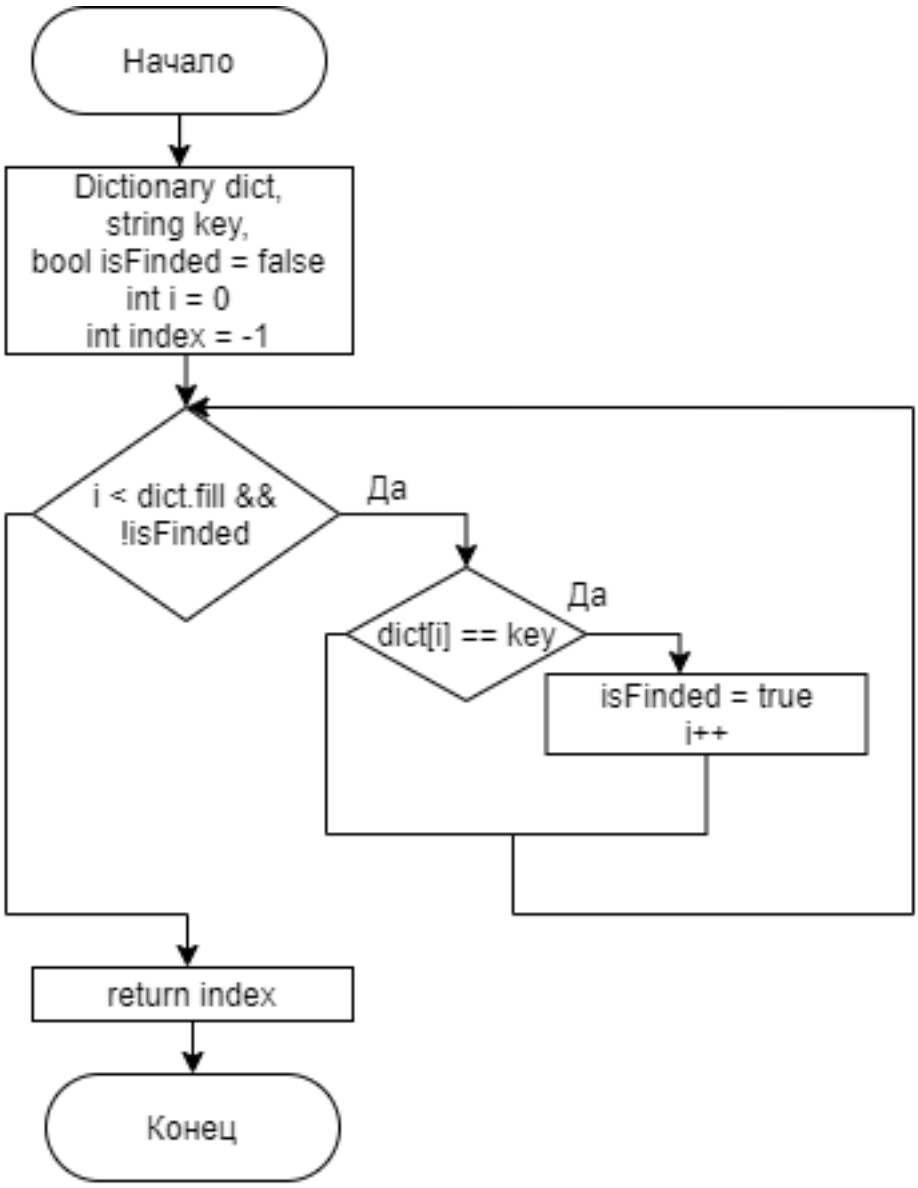

Рисунок 1 – Схема алгоритма полного перебора

На рисунке 2 изображена схема алгоритма двоичного поиска.

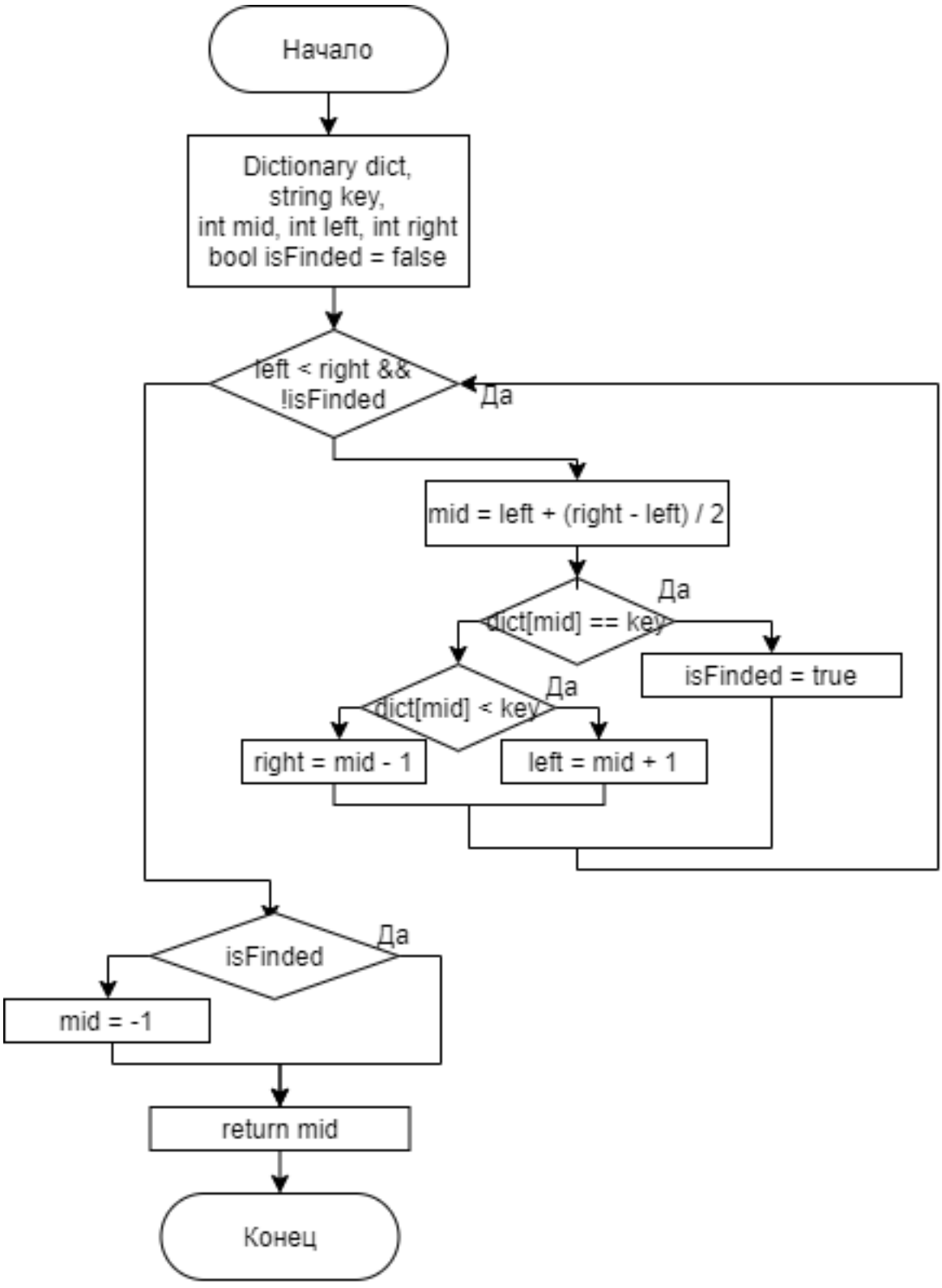

Рисунок 2 – Схема алгоритма двоичного поиска

На рисунке 3 изображена схема алгоритма поиска по сегментам.

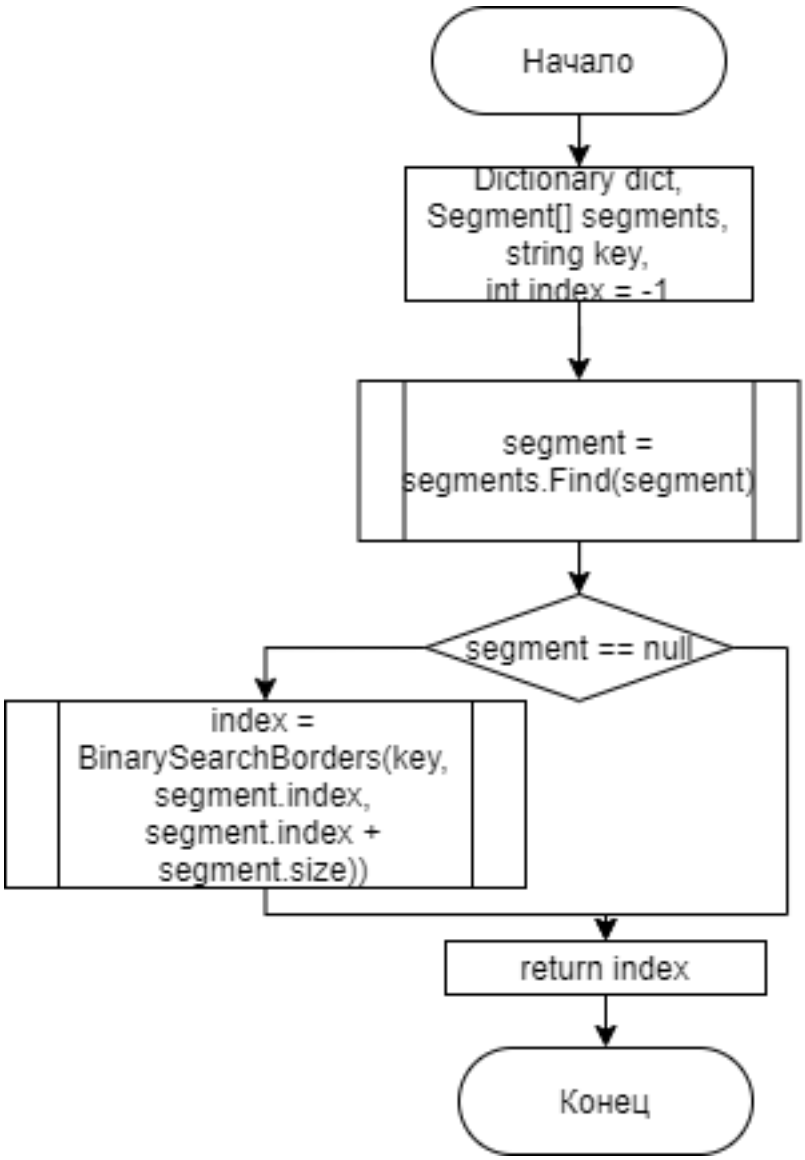

Рисунок 3 – Схема алгоритма поиска по сегментам

2.3 Вывод

В данном разделе были рассмотрена реализуемое ПО и обозначены требования к нему.

3 Технологическая часть

Ниже будут представлены средства реализации и листинги реализованной программы.

3.1 Средства реализации

Выбранный язык программирования - c#, так как не выдвигалось требований по выбору языка и он мне знаком [2]. Среда разработки - Visual Studio [3].

Технические характеристики машины, на которой проводились тесты:

- Windows 10 x64;
- 8 ГБ оперативной памяти;
- CPU: AMD FX(tm)-6350 Six-Core Processor 3.90GHz;
- 6 логических ядер.

3.2 Реализации алгоритмов

Ниже представлены листинг класса Dictionary, реализующего весь функционал. Метод BruteForce реализует полный перебор, BinarySearch - бинарный поиск, FindBySegments - поиск по сегментам.

На листинге 1 представлен класс Dictionary.

Листинг 1 – Dictionary

```
1   class Dictionary
2   {
3       string[] body;
4       Segment[] segments;
5       int segmentsCount = 0;
6       public int fill = 0;
7       public string this[int i]
8       {
9           get { return body[i]; }
10          set { body[i] = value; }
11      }
12
13      public Dictionary(string pathToFile)
14      {
15          if (File.Exists(pathToFile))
16          {
17              body = new string[100];
18              using (StreamReader sr = File.OpenText(pathToFile))
19              {
```

```
20                      string s;
21                      fill = 0;
22                      while ((s = sr.ReadLine()) != null)
23                      {
24                          if (fill == body.Length)
25                          {
26                              Array.Resize(ref body, (int)(fill * 1.5));
27                          }
28                          this[fill] = s.ToLower();
29                          fill++;
30                      }
31                  }
32                  Array.Resize(ref body, fill);
33                  Sort();
34                  FormSegments();
35              }
36          }
37
38          public void Sort()
39          {
40              Array.Sort(body);
41          }
42
43          public void FormSegments()
44          {
45              if (fill > 0)
46              {
47                  segments = new Segment[33];
48                  segmentsCount = 0;
49                  char key = this[0][0];
50                  int indexStart = 0;
51                  for (int i = 1; i < fill; i++)
52                  {
53                      if (key != this[i][0])
54                      {
55                          if (segmentsCount == segments.Length)
56                          {
57                              Array.Resize(ref segments, (int)(segmentsCount * 1.5));
58                          }
59                          segments[segmentsCount] = (new Segment(key, indexStart, i -
                                indexStart));
```

```
60                      segmentsCount++;
61                      key = this[i][0];
62                      indexStart = i;
63                  }
64              }
65              segments[segmentsCount] = (new Segment(key, indexStart, fill -
                    indexStart));
66              segmentsCount++;
67              Array.Resize(ref segments, segmentsCount);
68              Array.Sort(segments);
69          }
70      }
71      // return index of key
72      public int BruteForce(string key)
73      {
74          int index = -1;
75          bool isFinded = false;
76          for (int i = 0; i < fill && !isFinded; i++)
77          {
78              if (key.CompareTo(this[i]) == 0)
79              {
80                  isFinded = true;
81                  index = i;
82              }
83          }
84
85          return index;
86      }
87
88      public int BinarySearch(string key)
89      {
90          return BinarySearchBorders(key, 0, fill);
91      }
92
93      private int BinarySearchBorders(string key, int left, int right)
94      {
95          int mid = 0;
96          int finded = 1;
97          bool isFinded = false;
98
99          while (!isFinded && left < right)
```

```
100             {
101                 mid = left + (right - left) / 2;
102                 finded = key.CompareTo(this[mid]);
103                 if (finded == 0)
104                 {
105                     isFinded = true;
106                 }
107                 else if (finded < 0)
108                 {
109                     right = mid - 1;
110                 }
111                 else
112                 {
113                     left = mid + 1;
114                 }
115             }
116             mid = isFinded ? mid : -1;
117             return mid;
118         }
119 
120         public int FindBySegments(string key)
121         {
122             int index = -1;
123             char keyChar = key[0];
124             Segment segment = Array.Find(segments, (a) => (a.key == keyChar));
125 
126             if (segment != null)
127                 index = BinarySearchBorders(key, segment.index, segment.index + segment
                        .size);
128             return index;
129         }
130     }
```

3.3 Вывод

В данном разделе были описаны программные и аппаратные средства реализации, были представлены листинги программы.

4 Экспериментальная часть

В данной главе будет представлен пример работы программы и представлены результаты замера времени поиска.

4.1 Пример работы программы

Пример работы программы представлен на рисунке 4

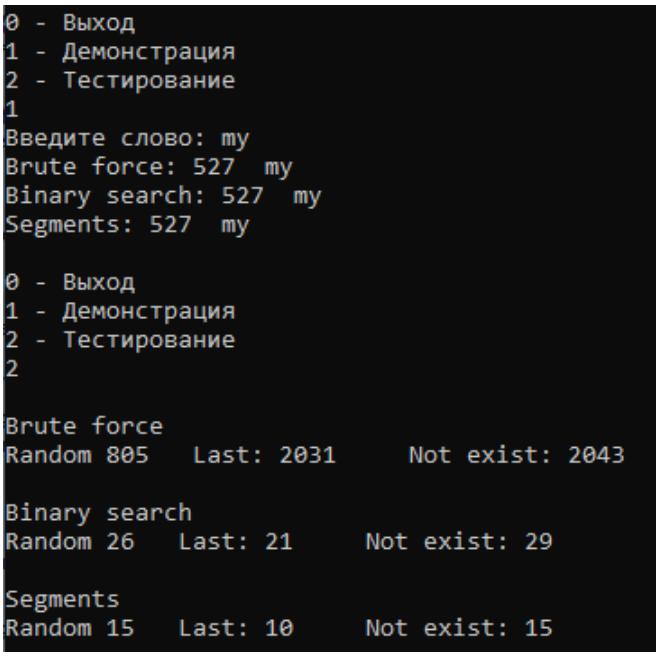

Рисунок 4 – Результаты эксперимента

4.2 Тесты

Тесты проводятся на словаре из 1000 элементов с шагом 25. По оси абсцисс отложено место искомого слова в словаре. Результаты тестирования представлены на рисунке 5

Рисунок 5 – Пример работы программы

4.3 Вывод

Видно, что алгоритм полного перебора самый долгий из реализованных алгоритмов, а алгоритм поиска по сегментам - самый быстрый. Это происходит потому, что при поиске по сегментам мы заранее отсекаем большую часть проверяемых слов.

Заключение

В данной лабораторной работе были описаны и реализованны несколько алгоритмов поиска, проведены замеры времени их выполнения. Цель работы достигнута, все задачи выполнены.

Список литературы

[1] Целочисленный двоичный поиск. ITMO [Электронный ресурс]. Режим доступа: (дата обращения - 20.11.2020) Свободный. URL: https://neerc.ifmo.ru/wiki/index.php?title=Целочисленный_двоичный_поиск

[2] Golang [Электронный ресурс]. Режим доступа: (дата обращения - 20.11.2020) Свободный. URL: https://ru.wikipedia.org/wiki/C_Sharp

[3] Visual Studio [Электронный ресурс]. Режим доступа: (дата обращения - 20.11.2020) Свободный. URL: https://visualstudio.microsoft.com/ru/