



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

(Группа)

(Подпись, дата)

Е.В. Брянская

(И.О. Фамилия)

Преподаватель

Л.Л. Волкова

(Подпись, дата)

(И.О. Фамилия)

Москва, 2020

Оглавление

Введение	3
1 Аналитическая часть	4
1.1 Цель и задачи	4
1.2 Описание словаря	4
1.3 Используемые алгоритмы поиска	4
1.3.1 Поиск полным перебором	4
1.3.2 Поиск в упорядоченном словаре двоичным поиском	4
1.3.3 Поиск полным перебором с использованием сегментов	5
Вывод	5
2 Конструкторская часть	6
2.1 Поиск полным перебором	6
2.2 Поиск в упорядоченном словаре двоичным поиском	7
2.3 Поиск полным перебором с использованием сегментов	8
2.4 Требования к ПО	11
2.5 Заготовки тестов	11
Вывод	11
3 Технологическая часть	12
3.1 Выбранный язык программирования	12
3.2 Листинг кода	12
3.3 Результаты тестов	15
3.4 Оценка времени	19
Вывод	22
4 Исследовательская часть	23
4.1 Характеристики ПК	23
4.2 Измерения	23
Вывод	24
Заключение	25
Список литературы	26

Введение

В этой лабораторной работе будут рассматриваться различные алгоритмы поиска ключа в словаре и проводиться их сравнение по затрачиваемому времени.

Словарь – структура данных, представляющая собой специальным образом организованный набор элементов хранимых данных.

Каждый элемент словаря состоит из двух объектов: ключа и значения (также называется сопутствующим данным). Значения ключей – уникальны, двух одинаковых в словаре не может быть. Для того, чтобы получить значение, нужно сначала дойти до соответствующего ключа.

Словари могут быть очень большими по объёму, и поиск нужного слова порой требует значительных временных затрат, поэтому необходимо найти такой алгоритм, который бы решал поставленную задачу за меньший промежуток времени для всех возможных случаев расположения ключа.

В данной лабораторной работе будут использоваться такие алгоритмы, как полный перебор, поиск в упорядоченном словаре двоичным поиском и поиск полным перебором с сегментацией.

1. Аналитическая часть

В этом разделе будут поставлены цель и основные задачи лабораторной работы, которые будут решаться по мере её выполнения.

1.1. Цель и задачи

Цель данной работы: реализовать и сравнить алгоритмы поиска в словаре.

Для достижения поставленной цели необходимо решить следующий ряд **задач**:

- 1) дать описание используемого словаря;
- 2) описать алгоритмы поиска;
- 3) реализовать все рассмотренные алгоритмы;
- 4) провести замеры процессорного времени работы алгоритмов;
- 5) найти минимальное/максимальное/среднее время и время работы при несуществующем ключе.

1.2. Описание словаря

Для этой лабораторной работы был составлен словарь сайтов и паролей к ним, где ключ – это url сайта, а значение – пароль.

1.3. Используемые алгоритмы поиска

Будут использованы следующие алгоритмы поиска ключа.

1.3.1 Поиск полным перебором

Этот алгоритм один из самых простых в реализации. В нём используется метод полного перебора, последовательно просматриваются все элементы словаря до тех пор, пока не найдётся соответствие или пока не проанализуются все возможные варианты. [1]

1.3.2 Поиск в упорядоченном словаре двоичным поиском

Для этого алгоритма необходимо в качестве подготовительного этапа сначала упорядочить данные. Затем на получившемся наборе производится двоичный поиск ключа,

использующий дробление массива на половины.

На каждом шаге алгоритма массив данных делится пополам, и дальнейшая работа производится с той частью, где потенциально должно находиться искомое значение. [2]

1.3.3 Поиск полным перебором с использованием сегментов

В этом алгоритме производится разбиение исходных данных на сегменты, которые объединены общим признаком. Помимо общих черт, в качестве принципа разделения можно взять частоту обращения к каждому элементу, и на основе анализа частот выделить группы.

Для того, чтобы осуществить поиск заданного ключа, необходимо сначала найти сегмент, в котором он может потенциально находиться, а затем в этом сегменте попытаться его найти.

Вывод

Были поставлены цель и задачи текущей лабораторной работы, также дано краткое описание рассматриваемых алгоритмов.

2. Конструкторская часть

Рассмотрим работу алгоритмов на словаре, где каждый элемент имеет следующую структуру: *key* – ключ, *value* – значение. Длина словаря – *len*.

2.1. Поиск полным перебором

Осуществляется поэлементный проход по словарю, и на каждом шаге ключ сравнивается с ключём текущего элемента. Если значения совпали, значит, цель достигнута, элемент найден. В таком случае, алгоритм завершает свою работу. Если же до конца словаря совпадение не было найдено, делается вывод о том, что такого ключа нет.

Схема алгоритма представлена на Рис.2.1.

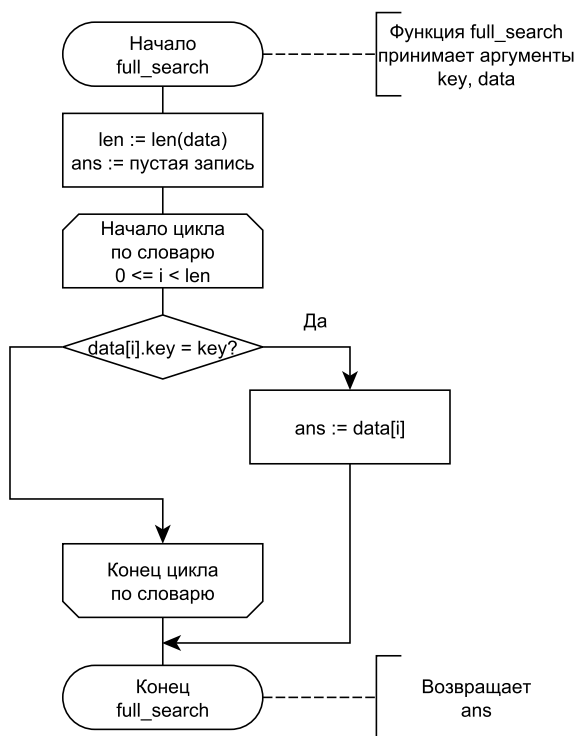


Рис. 2.1 — Поиск полным перебором

2.2. Поиск в упорядоченном словаре двоичным поиском

Перед применением алгоритма нужно предварительно отсортировать массив. В двоичном поиске вводятся такие понятия как левая (*left*) и правая (*right*) границы массива, индексы которых хранятся в соответствующих переменных. В начале работы алгоритма индекс левой границы равен 0, а правой $len - 1$. На каждой итерации цикла вычисляется индекс элемента, с которым будет производиться сравнение ключа. Находится он по формуле 2.1.

$$middle = \frac{left + right}{2} \quad (2.1)$$

Если данный ключ больше значения, которое находится по индексу *middle*, то в таком случае левая граница принимает значение $middle + 1$, если меньше, то правая граница становится равной $middle - 1$. В случае совпадения, делается вывод, что нужное значение успешно найдено, и алгоритм завершает свою работу. Если же правая граница становится меньше левой, это свидетельствует о том, что такого ключа в словаре нет, и нужно завершать работу.

Схема алгоритма представлена на Рис.2.2.

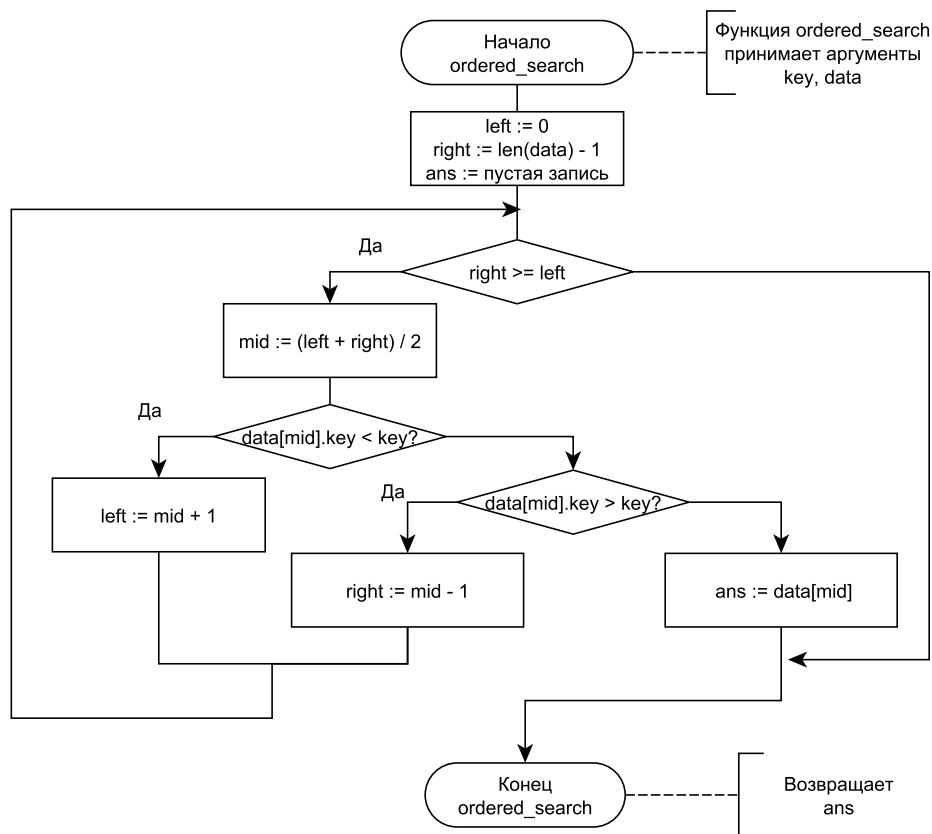


Рис. 2.2 — Поиск в упорядоченном словаре двоичным поиском

2.3. Поиск полным перебором с использованием сегментов

Для этого алгоритма также требуется предварительная подготовка – разбиение словаря на сегменты. В данном случае словарь разбивается по частоте запросов. Будем работать со словарём, в котором url, оканчивающиеся на *ru*, *com*, *io*, встречаются примерно одинаковое количество раз, в то время как *net*, *biz*, *org*, *info* употреблены в два раза меньше.

На основе этого были выделены сегменты с ключами: $\{ru\}$, $\{com\}$, $\{io\}$, $\{net, biz\}$, $\{org, info\}$. И на основе этих ключей словарь разделяется на соответствующие сегменты. Каждый из которых состоит из одного из ключей выше и массива элементов словаря (каждый элемент представляет из себя `key : value`).

Схема алгоритма разбиение на сегменты представлена на Рис.2.3.

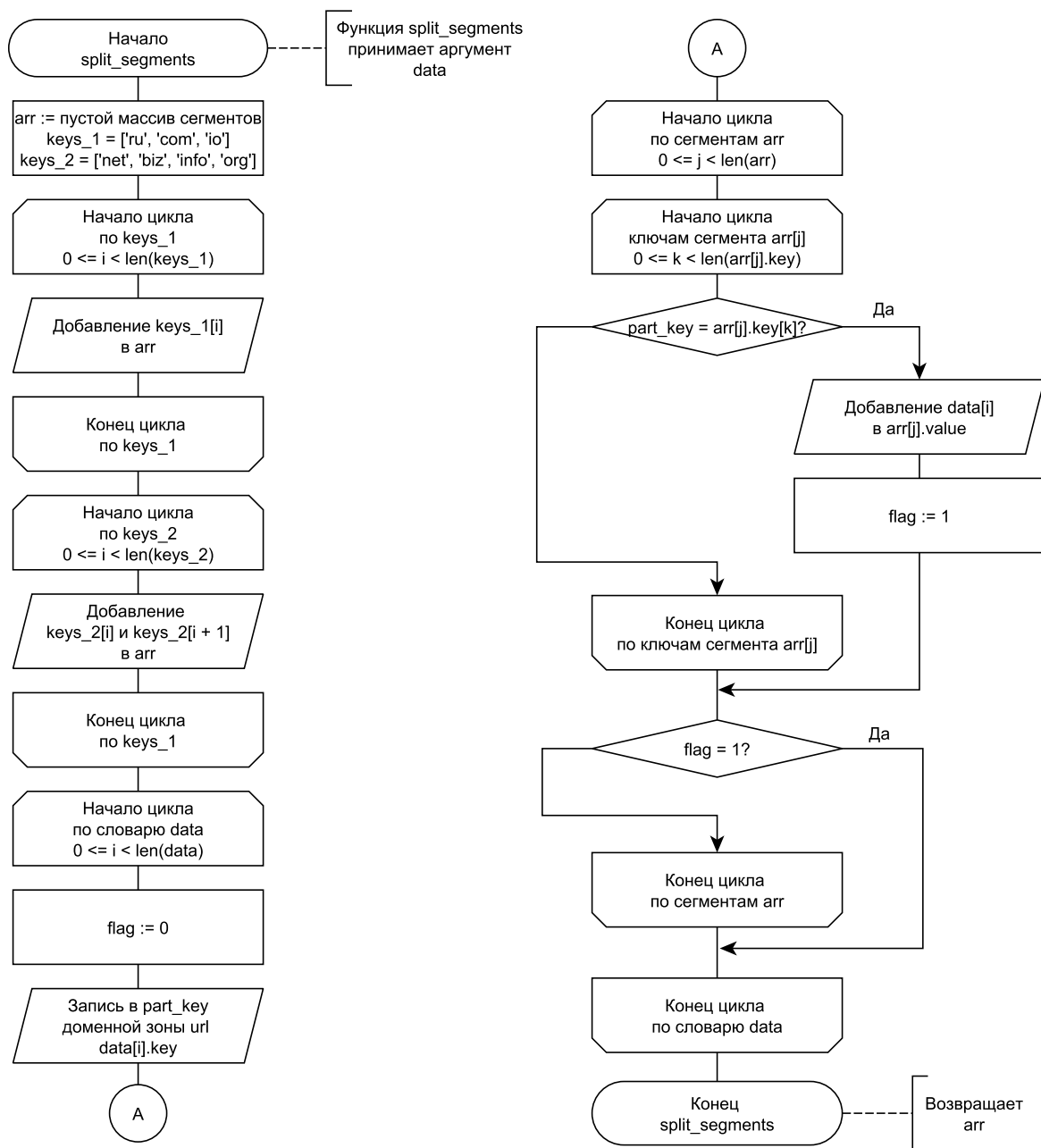


Рис. 2.3 — Разбиение словаря на сегменты

В этом алгоритме сначала находится подходящий сегмент, затем уже внутри сегмента производится последовательный поиск ключа.

Схема алгоритма представлена на Рис.2.4.

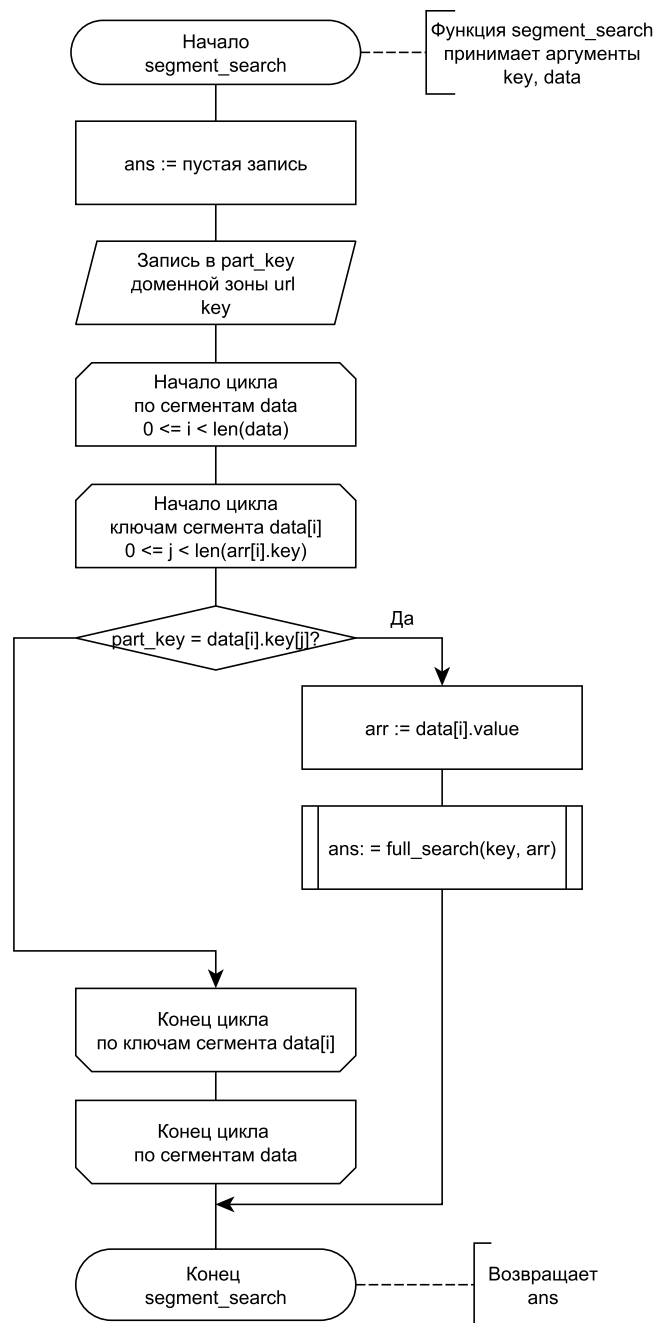


Рис. 2.4 — Поиск полным перебором с использованием сегментов

2.4. Требования к ПО

Для корректной работы алгоритмов и проведения тестов необходимо выполнить следующее.

- Обеспечить возможность ввода ключа и выбора алгоритма через консоль.
- В случае ввода некорректных данных вывести соответствующее сообщение. Программа не должна аварийно завершаться.
- Реализовать возможность вывода на экран времени, затрачиваемое на поиск каждого ключа из словаря, а также несуществующего ключа. Вывести максимальное, минимальное и среднее значение.

2.5. Заготовки тестов

При проверке на корректность работы необходимо провести следующие тесты:

- поиск первого элемента словаря;
- поиск последнего элемента словаря;
- поиск каждого сотого элемента словаря;
- поиск несуществующего ключа.

Вывод

В этом разделе разобраны основные принципы выбранных алгоритмов, приведены схемы работы каждого из них, сформулированы требования к программному обеспечению и сделаны заготовки тестов.

3. Технологическая часть

В данном разделе будут приведены листинги функций разрабатываемых алгоритмов поиска ключа.

3.1. Выбранный язык программирования

Для выполнения этой лабораторной работы был выбран язык программирования C++, так как есть большой навык работы с ним и с подключаемыми библиотеками, которые также использовались для проведения тестирования и замеров. [3]

Использованная среда разработки - Visual Studio. [4]

3.2. Листинг кода

Ниже представлены Листинги 3.1, 3.2, 3.3 функций, реализующих алгоритмы поиска ключей в словаре. В Листинге 3.4 показаны собственные типы, используемые в написании функций. А в Листингах 3.5, 3.6 приведены вспомогательные функции сортировки и разделения словаря по сегментам.

Листинг 3.1 — Поиск полным перебором

```
1 s_p_full_search(string key, sp_arr& data)
2 {
3     int len = data.size();
4
5     for (int i = 0; i < len; i++)
6         if (data[i].key == key)
7             return data[i];
8     return not_found();
9 }
```

Листинг 3.2 — Поиск в упорядоченном словаре двоичным поиском

```
1 s_p_ordered_search(string key, sp_arr& data)
2 {
3     int left = 0, right = data.size() - 1;
4
5     while (right >= left) {
6         int middle = (left + right) / 2;
7
8         if (data[middle].key < key)
9             left = middle + 1;
```

```

10     else if (data[middle].key > key)
11         right = middle - 1;
12     else
13         return data[middle];
14 }
15 return not_found();
16 }

```

Листинг 3.3 — Поиск полным перебором с использованием сегментов

```

1 s_p segment_search(string key, sgm_arr& data)
2 {
3     string part_key = key.substr(key.rfind('.') + 1);
4
5     for (size_t i = 0; i < data.size(); i++)
6     {
7         for (size_t j = 0; j < data[i].key.size(); j++)
8             if (data[i].key[j] == part_key)
9             {
10                 sp_arr& arr = data[i].value;
11                 return full_search(key, arr);
12             }
13     }
14
15     return not_found();
16 }

```

Листинг 3.4 — Используемые типы

```

1 using s_p = struct
2 {
3     string key;
4     string value;
5 };
6 using sp_arr = vector<s_p>;
7
8 using sgm = struct
9 {
10     vector<string> key;
11     sp_arr value;
12 };
13 using sgm_arr = vector<sgm>;

```

Листинг 3.5 — Функция сортировки словаря

```

1 void sort_arr(sp_arr& data)
2 {
3     s_p temp;
4
5     for (size_t i = 0; i < data.size() - 1; i++)
6         for (size_t j = 0; j < data.size() - i - 1; j++)
7             if (data[j].key > data[j + 1].key) {
8                 temp = data[j];
9                 data[j] = data[j + 1];
10                data[j + 1] = temp;
11            }
12 }

```

Листинг 3.6 — Функция разделения словаря по сегментам

```

1 sgm_arr split_segments(sp_arr& data)
2 {
3     sgm_arr arr;
4     vector<string> keys_1 = { "ru", "com", "io" }, keys_2 = { "net", "biz",
5         "org", "info" };
6     string part_key;
7     int flag = 0;
8
9     for (size_t i = 0; i < keys_1.size(); i++)
10    {
11        sgm temp_1;
12
13        temp_1.key.push_back(keys_1[i]);
14        arr.push_back(temp_1);
15    }
16
17    for (size_t i = 0; i < keys_2.size() - 1; i += 2)
18    {
19        sgm temp_2;
20
21        temp_2.key.push_back(keys_2[i]);
22        temp_2.key.push_back(keys_2[i + 1]);
23
24        arr.push_back(temp_2);
25    }
26 }

```

```

25
26   for (size_t i = 0; i < data.size(); i++)
27   {
28       flag = 0;
29       part_key = data[i].key.substr(data[i].key.rfind('.') + 1);
30
31       for (size_t j = 0; j < arr.size(); j++)
32       {
33           for (size_t k = 0; k < arr[j].key.size(); k++)
34           {
35               if (part_key == arr[j].key[k])
36               {
37                   arr[j].value.push_back(data[i]);
38                   flag = 1;
39                   break;
40               }
41           }
42           if (flag)
43               break;
44       }
45   }
46
47   return arr;
48 }

```

3.3. Результаты тестов

Для тестирования были написаны функции, проверяющие, согласно заготовкам выше, случаи. Выводы о корректности работы делаются на основе сравнения результатов.

Все тесты пройдены успешно. Сами тесты представлены ниже (Листинг 3.7).

Листинг 3.7 — Тесты

```

1 bool is_equal(s_p res, s_p ans)
2 {
3     if (res.key == ans.key && res.value == ans.value)
4         return true;
5     return false;
6 }

```

```

7
8 // Нахождение первого элемента словаря
9 void test_first_key(sp_arr& data)
10 {
11     sgm_arr sgm_data;
12
13     if (!is_equal(full_search(data[0].key, data), data[0]))
14         cout << endl << __FUNCTION__ << " full_search " << ":\tFAILED\n";
15     else
16         cout << endl << __FUNCTION__ << "\tfull_search " << ":\tSUCCESS\n";
17
18     sort_arr(data);
19     if (!is_equal(ordered_search(data[0].key, data), data[0]))
20         cout << __FUNCTION__ << " ordered_search " << ":\tFAILED\n";
21     else
22         cout << __FUNCTION__ << "\tordered_search " << ":\tSUCCESS\n";
23
24     sgm_data = split_segments(data);
25     if (!is_equal(segment_search(data[0].key, sgm_data), data[0]))
26         cout << __FUNCTION__ << " segment_search " << ":\tFAILED\n";
27     else
28         cout << __FUNCTION__ << "\tsegment_search " << ":\tSUCCESS\n";
29 }
30
31 // Нахождение каждого 100 элемента словаря
32 void test_each_100_key(sp_arr& data)
33 {
34     sgm_arr sgm_data;
35     int flag = 1;
36
37     for (size_t i = 0; i < data.size(); i += 100)
38         if (!is_equal(full_search(data[i].key, data), data[i]))
39         {
40             cout << endl << __FUNCTION__ << " full_search " << ":\tFAILED\n";
41             flag = 0;
42             break;
43         }
44     if (flag)
45         cout << endl << __FUNCTION__ << "\tfull_search " << ":\tSUCCESS\n";
46

```



```

47     flag = 1;
48     sort_arr(data);
49     for (size_t i = 0; i < data.size(); i += 100)
50         if (!is_equal(ordered_search(data[i].key, data), data[i]))
51         {
52             cout << __FUNCTION__ << " ordered_search " << ":\tFAILED\n";
53             flag = 0;
54             break;
55         }
56     if (flag)
57         cout << __FUNCTION__ << "\tordered_search " << ":\tSUCCESS\n";
58
59     flag = 1;
60     sgm_data = split_segments(data);
61     for (size_t i = 0; i < data.size(); i += 100)
62         if (!is_equal(segment_search(data[i].key, sgm_data), data[i]))
63         {
64             cout << __FUNCTION__ << " segment_search " << ":\tFAILED\n";
65             flag = 0;
66             break;
67         }
68     if (flag)
69         cout << __FUNCTION__ << "\tsegment_search " << ":\tSUCCESS\n";
70 }
71
72 // Нахождение последнего элемента словаря
73 void test_last_key(sp_arr& data)
74 {
75     sgm_arr sgm_data;
76
77     if (!is_equal(full_search(data[data.size() - 1].key, data), data[data.size() - 1]))
78         cout << endl << __FUNCTION__ << " full_search " << ":\tFAILED\n";
79     else
80         cout << endl << __FUNCTION__ << "\tfull_search " << ":\tSUCCESS\n";
81
82     sort_arr(data);
83     if (!is_equal(ordered_search(data[data.size() - 1].key, data), data[data.size() - 1]))
84         cout << __FUNCTION__ << " ordered_search " << ":\tFAILED\n";

```

```

85     else
86         cout << __FUNCTION__ << "\tordered_search " << ":\tSUCCESS\n";
87
88     sgm_data = split_segments(data);
89     if (!is_equal(segment_search(data[data.size() - 1].key, sgm_data),
90         data[data.size() - 1]))
91         cout << __FUNCTION__ << " segment_search " << ":\tFAILED\n";
92     else
93         cout << __FUNCTION__ << "\tsegment_search " << ":\tSUCCESS\n";
94 }
95
96 // Нахождение несуществующего элемента словаря
97 void test_not_exist_key(sp_arr& data)
98 {
99     sgm_arr sgm_data;
100     s_p key_not_exst;
101
102     key_not_exst.key = "123345";
103     key_not_exst.value = "000";
104
105     if (!is_equal(full_search(key_not_exst.key, data), not_found()))
106         cout << endl << __FUNCTION__ << " full_search " << ":\tFAILED\n";
107     else
108         cout << endl << __FUNCTION__ << "\tfull_search " << ":\tSUCCESS\n";
109
110     sort_arr(data);
111     if (!is_equal(ordered_search(key_not_exst.key, data), not_found()))
112         cout << __FUNCTION__ << " ordered_search " << ":\tFAILED\n";
113     else
114         cout << __FUNCTION__ << "\tordered_search " << ":\tSUCCESS\n";
115
116     sgm_data = split_segments(data);
117     if (!is_equal(segment_search(key_not_exst.key, sgm_data), not_found()))
118         cout << __FUNCTION__ << " segment_search " << ":\tFAILED\n";
119     else
120         cout << __FUNCTION__ << "\tsegment_search " << ":\tSUCCESS\n";
121 }
122 void run_tests(sp_arr& data)

```

```

123 {
124     cout << "—— START TESTING ——" << endl;
125
126     test_first_key(data);
127     test_each_100_key(data);
128     test_last_key(data);
129     test_not_exist_key(data);
130
131     cout << endl << "—— FINISHED ——" << endl;
132 }

```

3.4. Оценка времени

Процессорное время измеряется с помощью функции QueryPerformanceCounter библиотеки windows.h. [5] Осуществление замеров показано ниже (Листинг 3.8).

Листинг 3.8 — Замеры процессорного времени

```

1  double PCFreq = 0.0;
2  __int64 CounterStart = 0;
3
4  void start_measuring()
5  {
6      LARGE_INTEGER li;
7      QueryPerformanceFrequency(&li);
8
9      PCFreq = double(li.QuadPart) / 1000;
10
11     QueryPerformanceCounter(&li);
12     CounterStart = li.QuadPart;
13 }
14
15 double get_measured()
16 {
17     LARGE_INTEGER li;
18     QueryPerformanceCounter(&li);
19
20     return double(li.QuadPart - CounterStart) / PCFreq;
21 }
22
23 void measure_time(func_t f, sp_arr& data)

```

```

24 {
25     double min_t = 1e5, max_t = -1, avg_t = 0, t;
26     string key, not_exist_key = "123456";
27     int count;
28
29     for (size_t i = 0; i < data.size(); i++)
30     {
31         key = data[i].key;
32         count = 0;
33
34         start_measuring();
35         while (get_measured() < 0.07 * 1000)
36         {
37             f(key, data);
38             count++;
39         }
40
41         t = get_measured() / 1000 / count;
42
43         cout << i + 1 << " " << t << endl;
44
45         if (min_t > t)
46             min_t = t;
47         if (max_t < t)
48             max_t = t;
49         avg_t += t;
50     }
51
52     avg_t /= data.size();
53
54     count = 0;
55     start_measuring();
56     while (get_measured() < 0.07 * 1000)
57     {
58         f(not_exist_key, data);
59         count++;
60     }
61
62     t = get_measured() / 1000 / count;
63     cout << "NOT EXISTS" << " " << t << endl;

```

```

64
65     cout << "\\Максимальное время:\t" << max_t << endl;
66     cout << "Минимальное время:\t" << min_t << endl;
67     cout << "Среднее время:\t\t" << avg_t << endl;
68 }
69
70 void measure_time_sgm(func_sgm_t f, sgm_arr& data)
71 {
72     double min_t = 1e5, max_t = -1, avg_t = 0, t;
73     string key, not_exist_key = "123456";
74     int count, ind = 0;
75
76     for (size_t i = 0; i < data.size(); i++)
77     {
78         for (size_t j = 0; j < data[i].value.size(); j++)
79         {
80             ind++;
81             key = data[i].value[j].key;
82             count = 0;
83
84             start_measuring();
85             while (get_measured() < 0.03 * 1000)
86             {
87                 f(key, data);
88                 count++;
89             }
90
91             t = get_measured() / 1000 / count;
92
93             cout << ind << " " << t << endl;
94
95             if (min_t > t)
96                 min_t = t;
97             if (max_t < t)
98                 max_t = t;
99             avg_t += t;
100         }
101     }
102
103     avg_t /= data.size();

```

```

104
105     count = 0;
106     start_measuring();
107     while (get_measured() < 0.07 * 1000)
108     {
109         f(not_exist_key, data);
110         count++;
111     }
112
113     t = get_measured() / 1000;
114     cout << "NOT EXISTS" << " " << t << endl;
115
116     cout << "\Максимальное время:\t" << max_t << endl;
117     cout << "Минимальное время:\t" << min_t << endl;
118     cout << "Среднее время:\t\t" << avg_t << endl;
119 }

```

Вывод

Таким образом, приведены листинги кода каждой из функций, реализующих алгоритмы поиска ключа в словаре, а также листинг тестовых функций, направленных на проверку корректности их работы.

4. Исследовательская часть

Проведём замеры процессорного времени, которое затрачивается каждым алгоритмом на поиск ключа, найдём максимальное, минимальное и среднее и сравним полученные результаты.

4.1. Характеристики ПК

При проведении замеров времени использовался компьютер, имеющий следующие характеристики:

- ОС - Windows 10 Pro
- Процессор - Intel Core i7 10510U (1800 МГц)
- Объём ОЗУ - 16 Гб

4.2. Измерения

Для проведения замеров процессорного времени использовался словарь в 1200 элементов. Его содержимое генерируется случайным образом, и программно обеспечивается уникальность всех ключей. Находится время, затрачиваемое на поиск каждого ключа из словаря, а также несуществующего ключа.

Каждый замер проводится 5 раз для получения более точного среднего результата. Выделяется максимальное, минимальное и среднее время.

В таблице 4.1 представлены результаты замеров процессорного времени работы реализаций алгоритмов (в сек).

Таблица 4.1 — Результаты измерений на словаре в 1200 элементов

Алгоритм / Время	Поиск полным перебором	В упорядоченном словаре двоичным поиском	Поиск полным перебором с использованием сегментов
Максимальное	$1.68 \cdot 10^{-4}$	$1.115 \cdot 10^{-5}$	$8.763 \cdot 10^{-5}$
Минимальное	$3.689 \cdot 10^{-6}$	$4.121 \cdot 10^{-6}$	$6.918 \cdot 10^{-6}$
Среднее	$8.068 \cdot 10^{-5}$	$8.071 \cdot 10^{-6}$	$2.690 \cdot 10^{-5}$
Несуществующий ключ	$1.527 \cdot 10^{-4}$	$9.642 \cdot 10^{-6}$	$5.285 \cdot 10^{-5}$

Согласно полученным данным можно сделать следующие **выводы**:

- алгоритм поиска полным перебором показывает наибольшее максимальное время по сравнению с другими рассматриваемыми алгоритмами (на один порядок больше, чем два других алгоритма);
- с другой стороны, алгоритм полного перебора демонстрирует наименьшее минимальное время среди остальных (алгоритм с двоичным поиском больше в 1.1 раза, а алгоритм с сегментами в 1.88);
- минимальное время у всех трёх алгоритмов имеет одинаковый порядок;
- наименьшее среднее время показывает алгоритм с использованием двоичного поиска;
- время, затрачиваемое на поиск несуществующего ключа, примерно равно максимальному значению (наибольшая разница наблюдается у алгоритма с сегментами, объясняется тем, что сегменты значительно сужают область поиска, тем самым, не нужно рассматривать все элементы словаря).

Вывод

Проведены замеры процессорного времени, и на основе полученных данных были составлены сравнительные таблицы, описывающие время, которое каждый из алгоритмов затрачивает на поиск. В результате анализа получившихся таблиц были сделаны выводы, приведённые выше.

Заключение

В ходе лабораторной работы была достигнута поставленная цель, а именно, исследованы, реализованы и сопоставлены алгоритмы поиска ключа в словаре.

В процессе выполнения были решены все задачи. Описаны словарь, все рассматриваемые алгоритмы. Все проработанные алгоритмы реализованы, кроме того, были проведены замеры процессорного времени работы поиска и проведён сравнительный анализ, сделаны выводы.

По результатам замеров процессорного времени сделаны следующие заключения.

- алгоритм поиска полным перебором показывает как худшее максимальное время, так и лучшее минимальное по сравнению с другими рассматриваемыми алгоритмами;
- минимальное время у всех трёх алгоритмов одинакового порядка;
- алгоритм с использованием двоичного поиска имеет наименьшее среднее время;
- время, затрачиваемое на поиск несуществующего ключа, примерно равно максимальному значению (наибольшая разница наблюдается у алгоритма с сегментами, объясняется тем, что сегменты значительно сужают область поиска, тем самым, не нужно рассматривать все элементы словаря).

Список литературы

1. Кормен, Томас Х. и др Алгоритмы: построение и анализ, 3-е изд. : Пер. с англ. - М. : ООО "И.Д. Вильямс 2018. - 1328 с. : ил. - Парал. тит. англ. - ISBN 978-5-8459-2016-4 (рус.).
2. Клейнберг Дж., Тардос Е. Алгоритмы: разработка и применение. Классика Computer Science /Пер. с англ. Е. Матвеева. - СПб.: Питер, 2016. - 800 с.: ил. - (Серия "Классика computer science"). ISBN 978-5-496-01545-5
3. Документация по C++ [Электронный ресурс]. Режим доступа: <https://docs.microsoft.com/ru-ru/cpp/cpp/?view=msvc-160>, свободный (дата обращения: 22.11.2020)
4. Документация по Visual Studio 2019 [Электронный ресурс]. Режим доступа: <https://docs.microsoft.com/ru-ru/visualstudio/windows/?view=vs-2019>, свободный (дата обращения: 21.11.2020)
5. QueryPerformanceCounter function [Электронный ресурс]. Режим доступа: <https://docs.microsoft.com/en-us/windows/win32/api/profileapi/nf-profileapi-queryperformancecounter>, свободный (дата обращения: 22.11.2020).