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»

Отчет по лабораторной работе №7 по курсу "Анализ алгоритмов"

Тема Поиск в словаре

Студент Якуба Д. В.

Группа ИУ7-53Б

Оценка (баллы) _____

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

Москва — 2020 г.

Оглавление

Введение

Цель лабораторной работы

Реализация алгоритмов поиска по словарю: перебором, бинарным поиском и с применением частотного анализа.

Задачи лабораторной работы

1) изучить алгоритм поиска по словарю полным перебором;
2) изучить алгоритм бинарного поиска по словарю;
3) изучить алгоритм поиска по словарю с применением частотного анализа;
4) протестировать реализованные алгоритмы;
5) провести анализ временных характеристик реализованных алгоритмов;
6) подготовить отчёт по проведенной работе.

1 Аналитическая часть

В данном разделе описаны определение словаря как структуры данных, а также алгоритмы поиска по словарю.

1.1 Словарь

Словарь (ассоциативный массив)[1] — это абстрактный тип данных, хранящий пары вида «(ключ, значение)» и поддерживающий операции добавления пары, а также поиска и удаления пары по ключу:

- $INSERT$(*ключ, значение*;
- $FIND$(*ключ*);
- $REMOVE$(*ключ*).

Предполагается, что все ключи в словаре являются уникальными.

В паре (*ключ, значение*) *значение* называется значением, ассоциированным с ключом.

Операция $FIND$(*ключ*) возвращает значение, ассоциированное с заданным ключом, или некоторый специальный объект, означающий, что значения, ассоциированного с заданным ключом, нет. Две другие операции ничего не возвращают (за исключением, возможно, информации о том, успешно ли была выполнена данная операция).

1.2 Алгоритм поиска по словарю полным перебором

Алгоритм полного перебора [2] — это алгоритм разрешения математических задач, который можно отнести к классу способов нахождения решения рассмотрением всех возможных вариантов.

Для решения поставленной задачи поиска с использованием метода полного перебора потребуется последовательно просматривать каждую запись в словаре. В том случае, если у рассматриваемой пары ключ совпадает с искомым - алгоритм завершает свою работу, задача выполнена.

Трудоёмкость алгоритма зависит от того, присутствует ли искомый ключ в словаре, и, если присутствует - насколько он далеко от начала массива ключей.

При решении задачи возможно возникновение $(N + 1)$ случаев, где N - это количество записей в словаре: ключ не найден и N возможных случаев расположения ключа в словаре.

Лучшим случаем для рассматриваемого алгоритма будет факт того, что искомый ключ был обнаружен за одно сравнение (то есть ключ находится в начале словаря).

Худший случай наступает при следующих стечениях обстоятельств:

- элемент не был найден за N сравнений;
- ключ был обнаружен на последнем сравнении.

Пусть на старте алгоритм поиска затрачивает k_0 операций, а при каждом сравнении k_1 операций. Тогда в лучшем случае потребуется $k_0 + k_1$ операций; в случае, если ключ был найден на втором сравнении - потребуется $k_0 + 2k_1$ операций; в случае нахождения ключа на последней позиции или его отсутствия в словаре - потребуется $k_0 + Nk_1$ операций.

Средняя трудоёмкость алгоритма может быть рассчитана как математическое ожидание по формуле 1.2 (Ω - множество всех возможных исходов).

$$\sum_{i\in\Omega} p_i \cdot f_i = (k_0 + k_1) \cdot \frac{1}{N+1} + (k_0 + 2 \cdot k_1) \cdot \frac{1}{N+1} + (k_0 + 3 \cdot k_1) \cdot \frac{1}{N+1} +$$
$$+ (k_0 + Nk_1)\frac{1}{N+1} + (k_0 + N \cdot k_1) \cdot \frac{1}{N+1} =$$
$$= k_0 \frac{N+1}{N+1} + k_1 + \frac{1 + 2 + \cdots + N + N}{N+1} = k_0 + k_1 \cdot \left(\frac{N}{N+1} + \frac{N}{2}\right) =$$
$$= k_0 + k_1 \cdot \left(1 + \frac{N}{2} - \frac{1}{N+1}\right) \quad (1.1)$$

1.3 Алгоритм бинарного поиска по словарю

При двоичном поиске обход можно представить деревом, поэтому трудоёмкость в худшем случае составит $\log_2 N$ (спуск по двоичному дереву от корня до листа). Скорость роста функции $\log_2 N$ меньше, чем у N.

1.4 Частотный анализ

Некоторый алгоритм на получает словарь и по нему составляется частотный анализ:

- по частоте использования ключа на реальных данных;
- по частоте появления в выборке первого символа ключа (или его остатка от деления на некоторое значение, в случае чисел).

По предоставленному отчёту словарь разбивается на сегменты. В каждом сегменте находятся элементы с некоторым, определённым анализатором, признаком.

Сегменты также могут быть упорядочены, например, по размеру сегмента, если множество исходов обращений к сегменту имеет высокую дисперсию.

Вероятность обращения к определенному сегменту равна сумме вероятностей обращений к его ключам (формула 1.2, где P_i - вероятность обращения к i-ому сегменту, p_j - вероятность обращения к j-ому элементу, который принадлежит i-ому сегменту).

$$P_i = \sum_j p_j \tag{1.2}$$

Если обращения ко всем ключам равновероятны, то можно заменить сумму на произведение (формула 1.3, где N - количество элементов в i-ом сегменте, а p - вероятность обращения к произвольному ключу).

$$P_i = N \cdot p \tag{1.3}$$

Ключи в сегментах также упорядочиваются для проведения бинарного поиска.

Как итог, сначала с помощью бинарного поиска выбирается требующийся сегмент. В найденном сегменте с помощью алгоритма бинарного поиска обнаруживается требующийся ключ. Средняя трудоёмкость представленного алгоритма действий будет определяться формулой 1.4, где M - количество сегментов.

$$f_c p = \sum_{i \in [1,M]} (f_{\text{выбора i-го сегмента}} + f_{\text{ср. поиска в i-м сегменте}}) \cdot p_i \tag{1.4}$$

Вывод

Были рассмотрены определение словаря как структуры данных, а также алгоритмы поиска по словарю и оптимизации поиска.

В данной работе стоит задача реализации трёх рассмотренных алгоритмов поиска.

2 | Конструкторская часть

В данном разделе представлены структура записей в словаре, а также схемы алгоритма поиска по словарю полным перебором, с использованием бинарного поиска, а также с использованием сегментирования и частотного анализа.

2.1 Структура записи в словаре

Каждая запись в словаре описана парой вида ($ID_{студента}$: *Название курсового проекта*).

2.2 Схема алгоритма поиска по словарю полным перебором

Схема алгоритма поиска полным перебором предоставлена на рисунке 2.1.

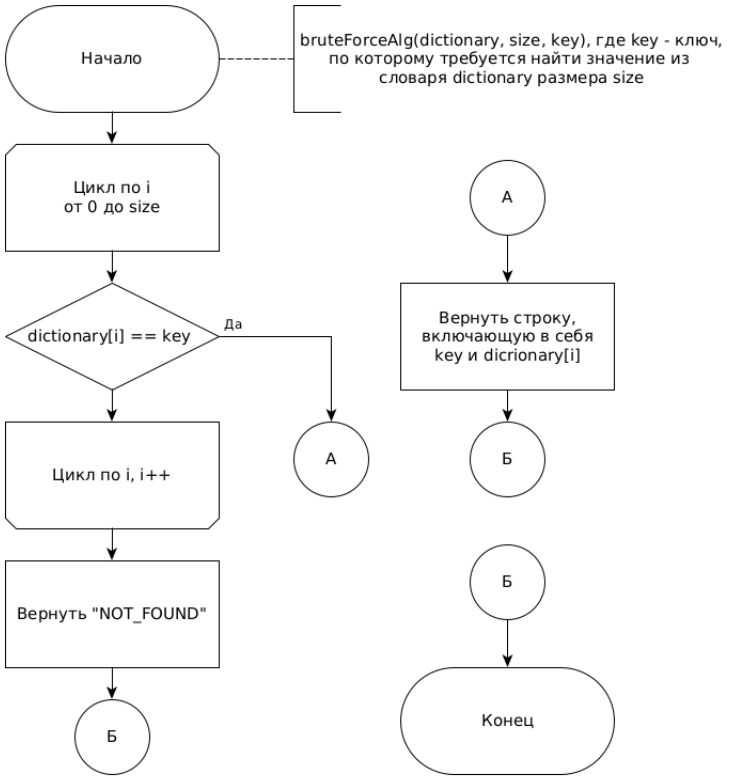

Рис. 2.1: Схема алгоритма поиска полным перебором.

2.3 Схема алгоритма бинарного поиска по словарю

Схема алгоритма бинарного поиска по словарю предоставлена на рисунке ??.

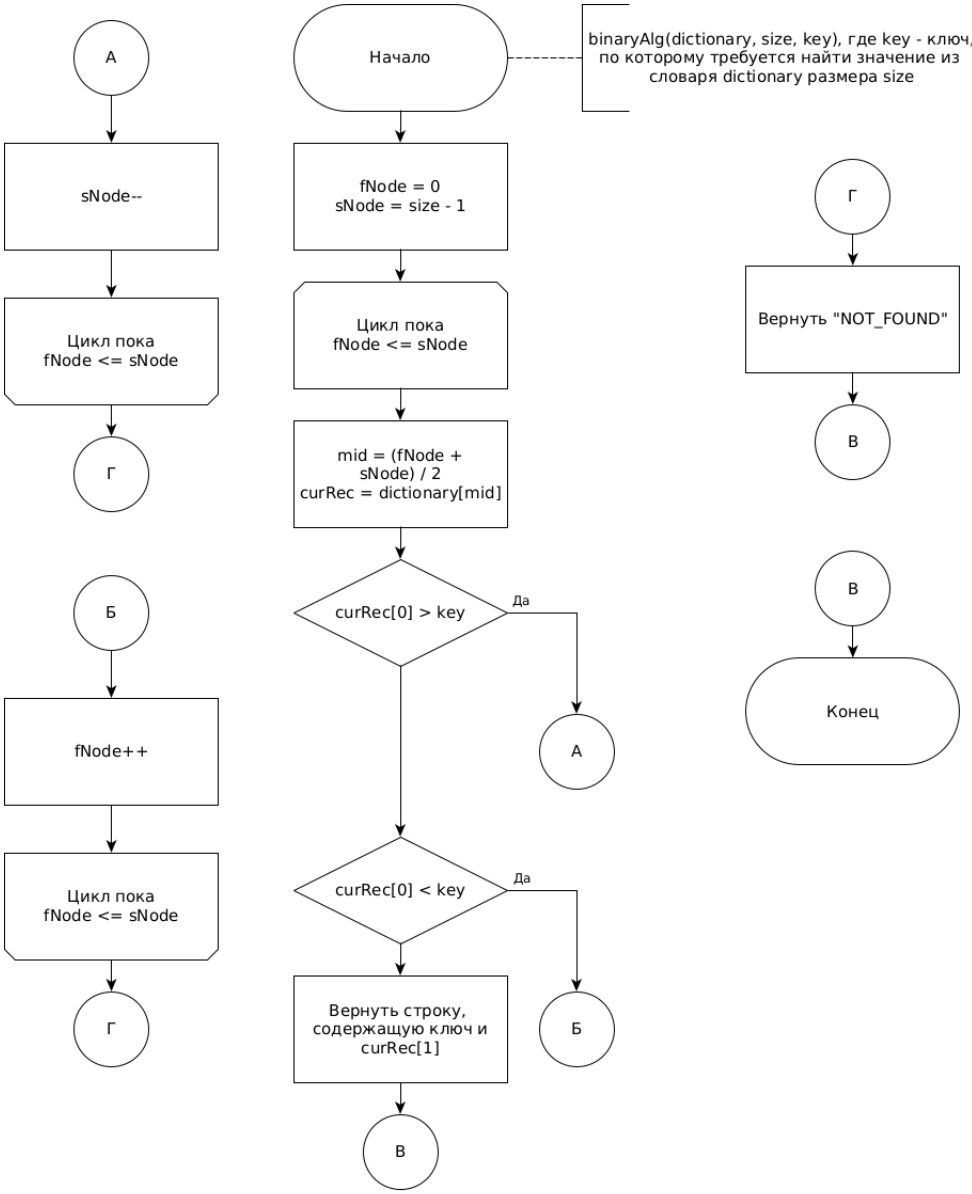

Рис. 2.2: Схема алгоритма бинарного поиска по словарю.

Вывод

Были представлены схемы алгоритма Брезенхема, а также реализации конвейерной обработки данных для данного алгоритма.

3 | Технологическая часть

В данном разделе приведены требования к программному обеспечению, средства реализации программного обеспечения, а также листинг кода.

3.1 Требования к программному обеспечению

- входные данные - количество выполняемых задач (количество растеризуемых отрезков);
- выходные данные - записи времени прихода и ухода обрабатываемых заявок для каждого реализованного конвейера.

3.2 Средства реализации программного обеспечения

При написании программного продукта был использован язык программирования C++ [3].

Данный выбор обусловлен следующими факторами:

- данный язык программирования преподавался в рамках курса объектно-ориентированного программирования;
- высокая вычислительная производительность;
- большое количество справочной и учебной литературы в сети Интернет;
- наличие реализации нативных потоков.

При написании программного продукта использовалась среда разработки QT Creator [4].

Данный выбор обусловлен следующими факторами:

- основы работы с данной средой разработки преподавался в рамках курса программирования на Си;
- QT Creator позволяет работать с расширением QtDesign, позволяющим создавать визуализируемый объект.

Для проведения замеров времени использовалась сторонняя библиотека Boost [5]. Данная библиотека позволила фиксировать время прихода и ухода каждой заявки с точностью до наносекунд.

3.3 Листинг кода

В листингах 3.1 и 3.2 предоставлены реализации рассматриваемых алгоритмов.

Листинг 3.1: Разбиение алгоритма Брезенхема

```
1  std::string now_str()
2  {
3      const boost::posix_time::ptime now = boost::posix_time::
           microsec_clock::local_time();
4
5      const boost::posix_time::time_duration td = now.time_of_day()
           ;
6
7      const long hours = td.hours();
8      const long minutes = td.minutes();
9      const long seconds = td.seconds();
10     const long nanoseconds =
11         td.total_nanoseconds() - ((hours * 3600 + minutes * 60 +
               seconds) * 1000000000);
12
13     char buf[40];
14     sprintf(buf, "%02ld:%02ld:%02ld.%09ld", hours, minutes,
           seconds, nanoseconds);
15
16     return buf;
17 }
18
```

```
19 SegmentRasterizator::SegmentRasterizator(int xStart_, int yStart_
       , int xEnd_, int yEnd_)
20 {
21     xStart = xStart_;
22     yStart = yStart_;
23     xEnd = xEnd_;
24     yEnd = yEnd_;
25     if (xStart == xEnd)
26         xEnd += 1;
27     else if (yStart == yEnd)
28         yEnd += 1;
29
30     image = new QImage(WIDTH, HEIGHT, QImage::Format_RGB32);
31     image->fill(Qt::white);
32 }
33
34 int sign(float num)
35 {
36     return (num < -__FLT_EPSILON__) ? -1 : ((num >
           __FLT_EPSILON__) ? 1 : 0);
37 }
38
39 void SegmentRasterizator::prepareConstantsForRB(int index)
40 {
41     std::printf(ANSI_BLUE_BRIGHT "From START worker: task %d
           BEGIN %s" ANSI_RESET "\n", index, now_str().c_str());
42
43     deltaX = xEnd - xStart;
44     deltaY = yEnd - yStart;
45
46     stepX = sign(deltaX);
47     stepY = sign(deltaY);
48
49     deltaX = std::abs(deltaX);
50     deltaY = std::abs(deltaY);
51
52     if (deltaX < deltaY)
53     {
54         std::swap(deltaX, deltaY);
55         stepFlag = true;
56     }
57     else
58         stepFlag = false;
59
60     tngModule = deltaY / deltaX;
```

```
61      mistake = tngModule - 0.5;
62
63      std::printf(ANSI_BLUE_BRIGHT "From START worker: task %d
            ENDED %s" ANSI_RESET "\n", index, now_str().c_str());
64  }
65
66  void SegmentRasterizator::rastSegment(int index)
67  {
68      std::printf(ANSI_MAGENTA_BRIGHT "From MIDDLE worker: task %d
            BEGIN %s" ANSI_RESET "\n", index, now_str().c_str());
69
70      float curX = xStart, curY = yStart;
71      for (int i = 0; i <= deltaX; i++)
72      {
73          dotsOfSegment.push_back(std::pair<int, int>(curX, curY));
74          if (stepFlag)
75          {
76              if (mistake >= 0)
77                  (curX += stepX, mistake--);
78              curY += stepY;
79          }
80          else
81          {
82              if (mistake >= 0)
83                  (curY += stepY, mistake--);
84              curX += stepX;
85          }
86          mistake += tngModule;
87      }
88
89      std::printf(ANSI_MAGENTA_BRIGHT "From MIDDLE worker: task %d
            ENDED %s" ANSI_RESET "\n", index, now_str().c_str());
90  }
91
92  void SegmentRasterizator::createImg(int index)
93  {
94      std::printf(ANSI_CYAN_BRIGHT"From END worker: task %d BEGIN %
            s" ANSI_RESET "\n", index, now_str().c_str());
95
96      for (auto iter = dotsOfSegment.begin(); iter < dotsOfSegment.
            end(); iter++)
97          image->setPixel(iter->first, iter->second, Qt::black);
98
99      std::printf(ANSI_CYAN_BRIGHT "From END worker: task %d ENDED
            %s" ANSI_RESET "\n", index, now_str().c_str());
```

```
100 }
101
102 std::vector<std::pair<int, int>> SegmentRasterizator::
        getDotsOfSegment()
103 {
104     return dotsOfSegment;
105 }
```

Листинг 3.2: Менеджер потоков

```
1  Director::Director(std::queue<SegmentRasterizator> &startQueue_)
2  {
3      startQueue = startQueue_;
4  }
5
6  void Director::processPrepare()
7  {
8      int i = 0;
9      for (SegmentRasterizator curSeg(startQueue.front());
           startQueue.size();
10           startQueue.pop(), curSeg = startQueue.front())
11     {
12
13         curSeg.prepareConstantsForRB(i++);
14         middleQueue.push(curSeg);
15     }
16 }
17
18 void Director::processRast()
19 {
20     int i = 0;
21     while (startQueue.size() || middleQueue.size())
22     {
23         if (middleQueue.empty())
24             continue;
25         SegmentRasterizator curSeg(middleQueue.front());
26
27         curSeg.rastSegment(i++);
28
29     endQueue.push(curSeg);
30         middleQueue.pop();
31     }
32 }
33
34 void Director::processCreate()
35 {
```

```
36      int i = 0;
37      while (startQueue.size() || middleQueue.size() || endQueue.
            size())
38      {
39          if (endQueue.empty())
40              continue;
41          SegmentRasterizator curSeg(endQueue.front());
42
43          curSeg.createImg(i++);
44          endQueue.pop();
45          final.push_back(curSeg);
46      }
47  }
48
49  void Director::initWork()
50  {
51      workers[0] = std::thread(&Director::processPrepare, this);
52      workers[1] = std::thread(&Director::processRast, this);
53      workers[2] = std::thread(&Director::processCreate, this);
54
55      workers[0].join();
56      workers[1].join();
57      workers[2].join();
58  }
59
60  std::vector<SegmentRasterizator> Director::getFinal() { return
        final; }
```

3.4 Тестирование программного продукта

В таблице 3.1 приведены тесты для функций, реализующих алгоритм Брезенхема. Тесты пройдены успешно.

Таблица 3.1: Тестирование функций

Точка начала отрезка (x, y)	Точка конца отрезка (x, y)	Ожидаемый результат
(1, 1)	(3, 3)	(1, 1), (2, 2), (3, 3)
(1, 1)	(1, 3)	(1, 1), (1, 2), (1, 3)
(1, 1)	(2, 1)	(1, 1), (2, 1)
(3, 3)	(1, 1)	(1, 1), (2, 2), (3, 3)

Вывод

Спроектированные алгоритмы были реализованы и протестированы.

4 | Исследовательская часть

4.1 Технические характеристики

Технические характеристики ЭВМ, на котором выполнялись исследования:

- ОС: Manjaro Linux 20.1.1 Mikah;
- Оперативная память: 16 Гб;
- Процессор: Intel Core i7-10510U.

При проведении замеров времени ноутбук был подключен к сети электропитания.

4.2 Пример работы программного обеспечения

На рисунке 4.1 приведен пример работы программы для 7 визуализируемых отрезков.

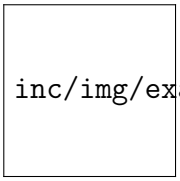

Рис. 4.1: Пример работы ПО.

4.3 Время выполнения алгоритмов

Алгоритм тестировался на данных, сгенерированных случайным образом.

В таблице 4.1 предоставлено время работы над каждым отрезком в предоставленном примере каждого из выделенных этапов.

Из таблицы видно, что среднее время выполнения этапа 1 составляет ≈ 17428.6 наносекунд. Среднее время выполнения этапа 2 составляет ≈ 38285.7 наносекунд. Среднее время выполнения этапа 3 составляет ≈ 18571.4 наносекунд. Таким образом, этап 1 сравним по среднему времени выполнения с этапом 2. Но после выполнения этапа 1 заметно, что последующие вызовы функции работают за константное время, равное 3000 наносекунд, что при наличии начальных "прогревочных"запусков вылилось бы в факт того, что этап 1 не был бы сопоставим по среднему времени выполнения с этапом 3. Этап 2 является самым долго выполняющимся.

Таблица 4.1: Замеры времени для выполнения выделенных этапов.

	Время обработки, нс		
Номер отрезка	Этап 1	Этап 2	Этап 3
0	97000	74000	27000
1	10000	38000	21000
2	3000	32000	16000
3	3000	35000	19000
4	3000	22000	12000
5	3000	35000	16000
6	3000	32000	19000

Вывод

При сравнении результатов замеров по времени стало известно, что самым быстрым этапом конвейера оказался этап 1. При этом, самым медленным из трех рассмотренных - этап 2.

В среднем этап 1 работает быстрее этапа 2 на ≈ 20857.1 наносекунд. При этом, при четвертой обработке отрезка разница в скорости выполнения составила 32000 наносекунд.

Этап 3 в среднем работает быстрее этапа 2 на ≈ 19714.3 наносекунд. При этом, при шестой обработке отрезка разница в скорости выполнения составила 19000 наносекунд.

Таким образом, среднее время выполнения алгоритма для каждого отрезка составило ≈ 74285.71 наносекунд.

Заключение

В ходе выполнения лабораторной работы была выполнена цель и следующие задачи:

1) было изучено асинхронное взаимодействие на примере конвейерной обработки данных;
2) была спроектирована система конвейерных вычислений;
3) была реализована система конвейерных вычислений;
4) была протестирована реализованная система;
5) был подготовлен отчёт по проведенной работе.

Исследования показали, что в среднем:

1) этап 1 работает быстрее этапа 2 на ≈ 20857.1 наносекунд;
2) этап 3 работает быстрее этапа 2 на ≈ 19714.3 наносекунд;
3) среднее время выполнения алгоритма составило ≈ 74285.71 наносекунд.

Литература

[1] National Institute of Standards and Technology [Электронный ресурс]. Режим доступа: https://xlinux.nist.gov/dads/HTML/assocarray.html (дата обращения 13.12.2020).

[2] Н. Нильсон. Искусственный интеллект. Методы поиска решений. М.: Мир, 1973. с. 273.

[3] C++ standart [Электронный ресурс]. Режим доступа: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf (дата обращения 01.12.2020).

[4] Qt documentation [Электронный ресурс]. Режим доступа: https://doc.qt.io/ (дата обращения 01.12.2020).

[5] Boost library [Электронный ресурс]. Режим доступа: https://www.boost.org/ (дата обращения 09.12.2020).