МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Отчет по выполнению практического задания 6.2

Тема: «АЛГОРИТМЫ ПОИСКА»

Дисциплина: Структуры и алгоритмы обработки данных

Выполнил студент Комисарик М.А.
группа ИКБО-20-23

Москва 2024

Цель работы: освоить приёмы реализации алгоритмов поиска образца в тексте.

Задание 1

Формулировка задачи

Дан текст, состоящий из слов, разделенных знаками препинания. Сформировать массив из слов, в которых заданная подстрока размещается в конце слова.

Модель решения

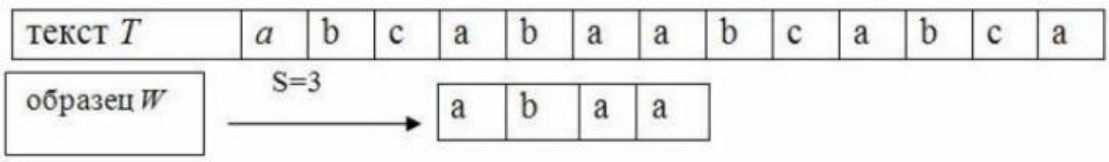
Для выполнения этого задания был выбран простой линейный поиск, из-за простоты реализации.

Поиск строки формально определяется следующим образом.

Пусть задан массив T из N элементов и массив W из M элементов, причем $0<M\leq N$.

Поиск строки обнаруживает первое вхождение W в T, результатом будем считать индекс i, указывающий на первое с начала строки (с начала массива T) совпадение с образом (словом).

Пример. Требуется найти все вхождения образца W = abaa в текст T=abcabaabcabca. Это последовательный поиск.



Образец входит в текст только один раз, со сдвигом S=3, индекс i=4.

Сложность алгоритма – O(N * M), где N – длина текста, M – длина слова для поиска. В худшем случае сложность $O(N * M^2)$

Для данной задачи необходимо немного модифицировать этот поиск. Кроме проверки на совпадение символов подстроки также требуется проверить, что следующий после последнего символ в тексте является знаком препинания или пробелом, переносом строки, и если оба этих условия выполняются, то взять в ответ все слово (слово ограничено справа знаком препинания или пробелом, а слева пробелом или переносом строки).

Код программы

Код линейного поиска слов по данному суффиксу:

```
1:   void LinearTextSuffixSearch(const std::string& T, const std::string& S,
     std::vector<std::string*>& out)
2:   {
3:
4:       for (unsigned int i = 0; i + S.length() < T.length(); ++i)
5:       {
6:           bool found = true;
7:           for (unsigned int j = 0; j < S.length(); ++j)
8:           {
9:               if (S[j] != T[i + j])
10:              {
11:                  found = false;
12:                  break;
13:              }
14:          }
15:          if (!found) continue;
16:          if (i + S.length() < T.length())
17:          {
18:              found = false;
19:              for (char c : StopSymbols)
20:              {
21:                  if (c == T[i + S.length()])
22:                  {
23:                      found = true;
24:                  }
25:              }
26:          }
27:          if (!found) continue;
28:
29:          std::string* currentString = new std::string(S);
30:          unsigned int k = i - 1;
31:          while (k > 0 && T[k] != ' ' && T[k] != '\n')
32:          {
33:              currentString->insert(0, 1, T[k]);
34:              --k;
35:          }
36:          out.push_back(currentString);
37:      }
38:  }
```

Результаты тестирования

Результаты тестирования программы на большом файле:

```
Введите название файла: text-sample.txt
Введите суффикс для поиска: их
Найденные слова:
знавших
наших
лучших
хороших
настоящих
таких
таких
их
их
их
таких
своих
Затраченное время: 1.077 милисекунд.
```

Тестирование алгоритма поиска на разных входных данных:

Количество символов	Затраченное время
58	52.8 микросекунд
10519	1.077 миллисекунд

Задание 2

Формулировка задачи

В текстовом файле хранятся входные данные: на первой сроке – подстрока (образец) длиной не более 17 символов для поиска в тексте; со второй строки – текст (строка), в котором осуществляется поиск образца. Строка, в которой надо искать, не ограниченна по длине. Применяя алгоритм Рабина-Карпа определить количество вхождений в текст заданного образца.

Модель решения

Алгоритм Рабина — Карпа — это алгоритм поиска строки, который ищет образец (подстроку), в тексте, используя хеширование.

Алгоритм редко используется для поиска одиночного шаблона, но имеет значительную теоретическую важность и очень эффективен в поиске совпадений множественных шаблонов одинаковой длины. Одно из простейших практических применений алгоритма Рабина — Карпа состоит в определении плагиата. Алгоритм Рабина — Карпа пытается ускорить проверку эквивалентности образца с подстроками в тексте, используя хэш функцию. Хеш-функция — это функция, преобразующая каждую строку в числовое значение, называемое хеш-значением (хэш);

В алгоритме Рабина-Карпа создадим хэш для образца, который мы ищем, и будем проверять, соответствует текущий хэш текста хэшу шаблона или нет. Если не соответствует, мы можем гарантировать, что шаблон не существует в тексте. Однако, если он соответствует, шаблон может присутствовать в тексте.

Пример

S1: Олег Иванович по фамилии Иваникин

S2: Иван

Чтобы вычислить хэш, нам нужно использовать простое число. Это может быть любое простое число. Для этого примера возьмем prime = 11.

Мы определим хеш-значение, используя эту формулу:

(код 1 символа) * (prime)0+ (код 2 символа) * (prime)¹ + (код 3 символа) * (prime)² + ...

Обозначим коды алфавита образца Иван

И -> 1

в -> 2

а -> 3

н -> 4

о -> 5

и -> 6

ч -> 7

ф -> 8

л -> 9

к -> 10

О->11

е-> 12

г->13

м->14

Хеш-значение Иван будет:

1 * 11⁰ + 2 * 11¹ + 3 * 11² +4*11(3) =5710

Теперь мы находим текущий хэш нашего текста. Рассматриваемый текст из исходной строки длиной 4 символа: Олег

Если скользящий хэш совпадает с хеш-значением нашего шаблона, мы проверим соответствие строк или нет. Поскольку наш шаблон имеет 4 буквы, мы возьмем с 1-ой 4 буквы и вычислим значение хэша. Мы получаем:

11 * 11⁰ + 9 * 11¹ +12 * 11² +13*11(3)

О л е г полученная сумма не совпадает с хэшом образца.

Эффективность в лучшем случае O(N), в худшем случае O(N * M)

Для ускорения нахождения хэш значений была проведена оптимизация подсчета: для вычисления каждой последующей хэш суммы можно вычесть

из предыдущей суммы первую степень 11-ти, поделить на 11 и прибавить следующую по тексту степень 11-ти. Таким образом хэш функцию можно вычислить за константное время.

За алфавит была взята таблица ASCII.

Код программы

Код строковой хэш функции:

```
1:   unsigned long long StringHash(const std::string& x)
2:   {
3:       unsigned long long out = 0;
4:       unsigned long long power = 1;
5:       for (unsigned int i = 0; i < x.length(); ++i)
6:       {
7:           out += power * static_cast<unsigned char>(x[i]);
8:           power *= 11;
9:       }
10:      return out;
11:  }
```

Код счетчика подстрок с помощью алгоритма поиска Рабина-Карпа:

```
1:   unsigned long long RkTextCount(const std::string& T, const std::string& S)
2:   {
3:       size_t count = 0;
4:       const unsigned long long sHash = StringHash(S);
5:       size_t sLen = S.length();
6:       unsigned long long currentHash = StringHash(T.substr(0, sLen));
7:       const unsigned long long maxPower = PowI(11, sLen - 1);
8:       for (size_t i = sLen; i + sLen < T.length(); ++i)
9:       {
10:          currentHash -= static_cast<unsigned char>(T[i - sLen]);
11:          currentHash /= 11;
12:          currentHash += static_cast<unsigned char>(T[i]) * maxPower;
13:          count += currentHash == sHash;
14:      }
15:
16:      return count;
17:  }
```

Результаты тестирования

Результаты тестирования программы на большом файле:

```
Введите название файла: text-sample.txt
Введите слово для поиска: их
Количество найденных слов: 14
Затраченное время: 1.0173 милисекунд.
```

Тестирование алгоритма поиска на разных входных данных:

Количество символов	Затраченное время
58	21.6 микросекунд
10519	1.0173 миллисекунды

ВЫВОДЫ

В рамках задания были реализованы и протестированы линейный поиск и алгоритм Рабина-Карпа. Алгоритм Рабина-Карпа хоть и имеет преимущество над линейным поиском, но оно оказалось пренебрежимо мало для рассматриваемой задачи с имеющимися размерами данных.

Приложения

Приложение 1 – Все использованные функции.

```
1:  void GenerateStudentsFile(ofstream& file, unsigned int studentsAmount,
    unsigned int keySize, unsigned int groupSize, unsigned int nameSize)
2:  {
3:      ifstream names("names.txt");
4:      string nameBuffer;
5:
6:      const unsigned int idSize = keySize + groupSize + nameSize;
7:      char* idBuffer = new char[idSize];
8:
9:      vector<unsigned int> keys;
10:     srand(time(0));
11:     GenerateRandomKeys(studentsAmount * 2, studentsAmount, keys);
12:
13:     for (unsigned int key : keys)
14:     {
15:         for (unsigned int i = 0; i < idSize; i++)
16:         {
17:             idBuffer[i] = 0;
18:         }
19:         getline(names, nameBuffer);
20:         *reinterpret_cast<unsigned int*>(idBuffer) = key;
21:         strncpy(idBuffer + keySize, "ИКБО-20-23", groupSize);
22:         strncpy(idBuffer + keySize + groupSize, nameBuffer.c_str(),
    nameBuffer.size());
23:         file.write(idBuffer, idSize);
24:     }
25:
26:     delete[] idBuffer;
27:     names.close();
28: }
29:
30: char* LinearFileSearch(const string& fileName, const unsigned int key, const
    unsigned int entrySize, const unsigned int keyOffset)
31: {
32:     ifstream file(fileName, ios::binary | ios::in);
33:     if (!file)
34:     {
35:         file.close();
36:         throw invalid_argument("No file found with name " + fileName);
37:     }
38:
39:     char* buffer = new char[entrySize];
40:     while (file.read(buffer, entrySize))
41:     {
42:         unsigned int readKey = *reinterpret_cast<unsigned int*>(buffer +
    keyOffset);
43:         if (readKey == key)
44:         {
45:             file.close();
46:             return buffer;
47:         }
48:     }
49:     file.close();
50:     return nullptr;
51: }
52:
53: void PrintStudent(char* student, const unsigned int keySize, const unsigned
    int groupSize, const unsigned int nameSize)
54: {
55:     cout << "Номер зачетной книжки: " << *reinterpret_cast<unsigned
    int*>(student) << '\n';
```

```
56:       cout << "Номер группы: ";
57:       cout.write(student + keySize, groupSize) << '\n';
58:       cout << "ФИО: ";
59:       cout.write(student + keySize + groupSize, nameSize) << '\n';
60:   }
61:
62:   void GenerateKeyTable(vector<unsigned long long>& table, ifstream& file, const
      unsigned int entrySize, const unsigned int keyOffset)
63:   {
64:       table.clear();
65:       char* entry = new char[entrySize];
66:       unsigned int i = 0;
67:       while (file.read(entry, entrySize))
68:       {
69:           unsigned int key = *reinterpret_cast<unsigned int*>(entry +
      keyOffset);
70:           table.push_back(0);
71:           *reinterpret_cast<unsigned int*>(&table[i]) = key;
72:           *(reinterpret_cast<unsigned int*>(&table[i]) + 1) = i;
73:           i++;
74:       }
75:       SortKeyTable(table);
76:   }
77:
78:   void SortKeyTable(vector<unsigned long long>& table)
79:   {
80:       sort(table.begin(), table.end(), [](unsigned long long a, unsigned long
      long b)
81:       {
82:           return *reinterpret_cast<unsigned int*>(&a) <
      *reinterpret_cast<unsigned int*>(&b);
83:       });
84:   }
85:
86:   void GenerateLookupTable(unsigned int len, std::vector<unsigned int>& table)
87:   {
88:       table = vector<unsigned int>(static_cast<unsigned int>(log2(len)) + 3, 0);
89:       unsigned int power = 1;
90:       unsigned int count = 0;
91:
92:       do
93:       {
94:           power <<= 1;
95:           table[count] = (len + (power >> 1)) / power;
96:       }
97:       while (table[count++] != 0);
98:   }
99:
100:  unsigned int BinarySearchInKeyTable(const vector<unsigned long long>&
      keyTable, const unsigned int key, const vector<unsigned int>& lookupTable)
101:  {
102:      unsigned int index = lookupTable[0] - 1;
103:      unsigned int count = 0;
104:      while (lookupTable[count] != 0)
105:      {
106:          if (key == *reinterpret_cast<const unsigned int*>(&keyTable[index]))
107:          {
108:              return *(reinterpret_cast<const unsigned int*>(&keyTable[index]) +
      1);
109:          }
110:
111:          if (key < *reinterpret_cast<const unsigned int*>(&keyTable[index]))
112:          {
113:              index -= lookupTable[++count];
114:          }
```

```
115:         else
116:         {
117:             index += lookupTable[++count];
118:         }
119:     }
120:     return keyTable.size();
121: }
122:
123: char* AccessFileByRef(ifstream& file, const unsigned int ref, const unsigned
     int entrySize)
124: {
125:     file.clear();
126:     file.seekg(ref * entrySize, ios::beg);
127:     char* entry = new char[entrySize];
128:     file.read(entry, entrySize);
129:     return entry;
130: }
131:
132: void GenerateRandomKeys(const unsigned int maxAmount, const unsigned int
     keysAmount, vector<unsigned int>& buffer)
133: {
134:     buffer.clear();
135:     vector<bool> keyTable(maxAmount, false);
136:     for (unsigned int i = 0; i < keysAmount; i++)
137:     {
138:         unsigned int r = rand() % maxAmount;
139:         int k = 0;
140:         while (keyTable[(r + k) % maxAmount])
141:         {
142:             k *= -1;
143:             if (!keyTable[(r + k) % maxAmount])
144:             {
145:                 break;
146:             }
147:             k *= -1;
148:             k++;
149:         }
150:         buffer.push_back((r + k) % maxAmount);
151:         keyTable[(r + k) % maxAmount] = true;
152:     }
153: }
154:
155: void DisplayTimeDuration(std::chrono::steady_clock::duration duration)
156: {
157:     auto time = chrono::duration_cast<chrono::nanoseconds>(duration);
158:     string timeUnit = "наносекунд";
159:     long double convertedTime = time.count();
160:     if (convertedTime >= 1000)
161:     {
162:         convertedTime /= 1000;
163:         timeUnit = "микросекунд";
164:         if (convertedTime >= 1000)
165:         {
166:             convertedTime /= 1000;
167:             timeUnit = "милисекунд";
168:             if (convertedTime >= 1000)
169:             {
170:                 convertedTime /= 1000;
171:                 timeUnit = "секунд";
172:             }
173:         }
174:     }
175:     cout << "Затраченное время: " << convertedTime << ' ' << timeUnit <<
     ".\n";
176: }
```