****

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

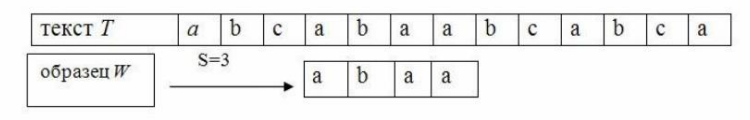
Для выполнения этого задания был выбран простой линейный поиск, из-за простоты реализации.

Поиск строки формально определяется следующим образом.

Пусть задан массив Т из N элементов и массив W из M элементов, причем 0<M≤N.

Поиск строки обнаруживает первое вхождение W в Т, результатом будем считать индекс i, указывающий на первое с начала строки (с начала массива Т) 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 \* M2)

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).

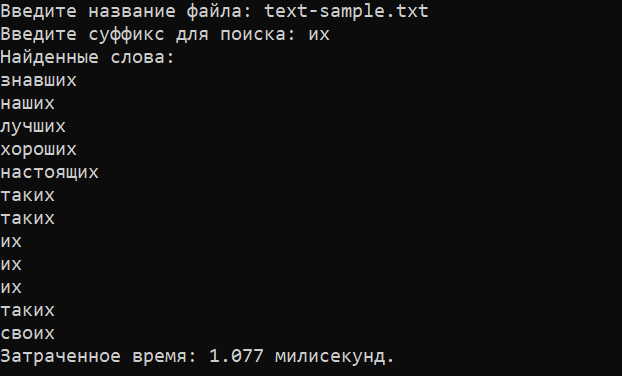
Код программы

Код линейного поиска слов по данному суффиксу:

1. void LinearTextSuffixSearch(const std::string& T, const std::string& S, std::vector<std::string\*>& out)
2. {
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. std::string\* currentString = new std::string(S);
29. unsigned int k = i - 1;
30. while (k > 0 && T[k] != ' ' && T[k] != '\n')
31. {
32. currentString->insert(0, 1, T[k]);
33. --k;
34. }
35. out.push\_back(currentString);
36. }
37. }

Результаты тестирования

Результаты тестирования программы на большом файле:



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:

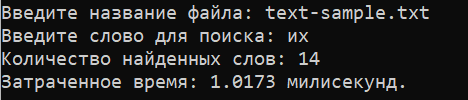
1. unsigned long long StringHash(const std::string& x)
2. {
3. unsigned long long out = 0;
4. unsigned long long power = 1;
5. for (unsigned int i = 0; i < x.length(); ++i)
6. {
7. out += power \* static\_cast<unsigned char>(x[i]);
8. power \*= 11;
9. }
10. return out;
11. }

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. return count;
16. }

Результаты тестирования

Результаты тестирования программы на большом файле:



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. const unsigned int idSize = keySize + groupSize + nameSize;
6. char\* idBuffer = new char[idSize];
7. vector<unsigned int> keys;
8. srand(time(0));
9. GenerateRandomKeys(studentsAmount \* 2, studentsAmount, keys);
10. for (unsigned int key : keys)
11. {
12. for (unsigned int i = 0; i < idSize; i++)
13. {
14. idBuffer[i] = 0;
15. }
16. getline(names, nameBuffer);
17. \*reinterpret\_cast<unsigned int\*>(idBuffer) = key;
18. strncpy(idBuffer + keySize, "ИКБО-20-23", groupSize);
19. strncpy(idBuffer + keySize + groupSize, nameBuffer.c\_str(), nameBuffer.size());
20. file.write(idBuffer, idSize);
21. }
22. delete[] idBuffer;
23. names.close();
24. }
25. char\* LinearFileSearch(const string& fileName, const unsigned int key, const unsigned int entrySize, const unsigned int keyOffset)
26. {
27. ifstream file(fileName, ios::binary | ios::in);
28. if (!file)
29. {
30. file.close();
31. throw invalid\_argument("No file found with name " + fileName);
32. }
33. char\* buffer = new char[entrySize];
34. while (file.read(buffer, entrySize))
35. {
36. unsigned int readKey = \*reinterpret\_cast<unsigned int\*>(buffer + keyOffset);
37. if (readKey == key)
38. {
39. file.close();
40. return buffer;
41. }
42. }
43. file.close();
44. return nullptr;
45. }
46. void PrintStudent(char\* student, const unsigned int keySize, const unsigned int groupSize, const unsigned int nameSize)
47. {
48. cout << "Номер зачетной книжки: " << \*reinterpret\_cast<unsigned int\*>(student) << '\n';
49. cout << "Номер группы: ";
50. cout.write(student + keySize, groupSize) << '\n';
51. cout << "ФИО: ";
52. cout.write(student + keySize + groupSize, nameSize) << '\n';
53. }
54. void GenerateKeyTable(vector<unsigned long long>& table, ifstream& file, const unsigned int entrySize, const unsigned int keyOffset)
55. {
56. table.clear();
57. char\* entry = new char[entrySize];
58. unsigned int i = 0;
59. while (file.read(entry, entrySize))
60. {
61. unsigned int key = \*reinterpret\_cast<unsigned int\*>(entry + keyOffset);
62. table.push\_back(0);
63. \*reinterpret\_cast<unsigned int\*>(&table[i]) = key;
64. \*(reinterpret\_cast<unsigned int\*>(&table[i]) + 1) = i;
65. i++;
66. }
67. SortKeyTable(table);
68. }
69. void SortKeyTable(vector<unsigned long long>& table)
70. {
71. sort(table.begin(), table.end(), [](unsigned long long a, unsigned long long b)
72. {
73. return \*reinterpret\_cast<unsigned int\*>(&a) < \*reinterpret\_cast<unsigned int\*>(&b);
74. });
75. }
76. void GenerateLookupTable(unsigned int len, std::vector<unsigned int>& table)
77. {
78. table = vector<unsigned int>(static\_cast<unsigned int>(log2(len)) + 3, 0);
79. unsigned int power = 1;
80. unsigned int count = 0;
81. do
82. {
83. power <<= 1;
84. table[count] = (len + (power >> 1)) / power;
85. }
86. while (table[count++] != 0);
87. }
88. unsigned int BinarySearchInKeyTable(const vector<unsigned long long>& keyTable, const unsigned int key, const vector<unsigned int>& lookupTable)
89. {
90. unsigned int index = lookupTable[0] - 1;
91. unsigned int count = 0;
92. while (lookupTable[count] != 0)
93. {
94. if (key == \*reinterpret\_cast<const unsigned int\*>(&keyTable[index]))
95. {
96. return \*(reinterpret\_cast<const unsigned int\*>(&keyTable[index]) + 1);
97. }
98. if (key < \*reinterpret\_cast<const unsigned int\*>(&keyTable[index]))
99. {
100. index -= lookupTable[++count];
101. }
102. else
103. {
104. index += lookupTable[++count];
105. }
106. }
107. return keyTable.size();
108. }
109. char\* AccessFileByRef(ifstream& file, const unsigned int ref, const unsigned int entrySize)
110. {
111. file.clear();
112. file.seekg(ref \* entrySize, ios::beg);
113. char\* entry = new char[entrySize];
114. file.read(entry, entrySize);
115. return entry;
116. }
117. void GenerateRandomKeys(const unsigned int maxAmount, const unsigned int keysAmount, vector<unsigned int>& buffer)
118. {
119. buffer.clear();
120. vector<bool> keyTable(maxAmount, false);
121. for (unsigned int i = 0; i < keysAmount; i++)
122. {
123. unsigned int r = rand() % maxAmount;
124. int k = 0;
125. while (keyTable[(r + k) % maxAmount])
126. {
127. k \*= -1;
128. if (!keyTable[(r + k) % maxAmount])
129. {
130. break;
131. }
132. k \*= -1;
133. k++;
134. }
135. buffer.push\_back((r + k) % maxAmount);
136. keyTable[(r + k) % maxAmount] = true;
137. }
138. }
139. void DisplayTimeDuration(std::chrono::steady\_clock::duration duration)
140. {
141. auto time = chrono::duration\_cast<chrono::nanoseconds>(duration);
142. string timeUnit = "наносекунд";
143. long double convertedTime = time.count();
144. if (convertedTime >= 1000)
145. {
146. convertedTime /= 1000;
147. timeUnit = "микросекунд";
148. if (convertedTime >= 1000)
149. {
150. convertedTime /= 1000;
151. timeUnit = "милисекунд";
152. if (convertedTime >= 1000)
153. {
154. convertedTime /= 1000;
155. timeUnit = "секунд";
156. }
157. }
158. }
159. cout << "Затраченное время: " << convertedTime << ' ' << timeUnit << ".\n";
160. }