Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №1 по курсу «Дискретный анализ»

Студент: В. Д. Медведев
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б-22

Дата:
Оценка:
Подпись:

Москва, 2024

Лабораторная работа №1

Задача: Требуется разработать программу, осуществляющую ввод пар «ключ-значение», их упорядочивание по возрастанию ключа указанным алгоритмом сортировки за линейное время и вывод отсортированной последовательности.

Вариант сортировки: Поразрядная сортировка.

Вариант ключа: `MD5-суммы (32-разрядные шестнадцатиричные числа).`

Вариант значения: `строки фиксированной длины 64 символа, во входных данных могут встретиться строки меньшей длины, при этом строка дополняется до 64-х нулевыми символами, которые не выводятся на экран.`

1 Описание

Требуется написать реализацию алгоритма поразрядной сортировки.

Основная идея поразрядной сортировки заключается в том, чтобы на каждом шаге упорядочивать ключи по текущему разряду. Будем осуществлять проход по разрядам справа налево. Сами разряды будут упорядочены с помощью сортировки подсчетом, так как максимальный размер использующегося в данной сортировке вспомогательного массива будет не больше, чем мощность алфавита, из знаков которого составлены ключи.

2 Исходный код

На каждой непустой строке входного файла располагается пара «ключ-значение», поэтому создадим новый класс $TElem$, в котором будем хранить ключ и значение. Внутри они будут храниться в статическом массиве длиной 32 и 64 знака соответственно. Вынесем эти числа в глобальные константы. Также, в силу реализации программы на динамическом массиве, ограничим его максимальный размер. У класса TElem есть два метода - напечатать элемент и изменить его поля. Далее пропишем функцию перевода шестнадцатиричного числа в его десятичный аналог. Это потребуется впоследствии для того, чтобы реализовать сортировку подсчетом. Функция, осуществляющая поразрядную сортировку, принимает на вход указатель на сортируемый массив и его размер. Для каждого из 32 разрядов ключа запускаем сортировку подсчетом, результат которой помещаем во временный массив. Далее следует важная деталь: чтобы нам каждый раз не перемещать arSize элементов из временного массива в старый, что стоит O(arSize), будем просто обменивать указатели через еще одну временную переменную, которая весит всего лишь 4 или 8 байт в зависимости от архитектуры. Это значительно ускоряет работу программы. В точке входа int main() считываем пары строк, пока это возможно. На каждом шаге считанные строки присваиваем текущему элементу при помощи метода void Set(std::string, std::string). Запускаем сортировку, после чего выводим элементы с помощью метода void PrintElem().

```cpp
#include <iostream>

const int KEY_LENGTH = 32;
const int VALUE_LENGTH = 64;
const int MAX_ARRAY_SIZE = 1000000;

class TElem {
private:
    int valueSize;
public:
    char key[KEY_LENGTH];
    char value[VALUE_LENGTH];
    void PrintElem();
    void Set(std::string &key, std::string &value);
};

void TElem::PrintElem() {
    for (int i = 0; i < KEY_LENGTH; ++i) {
        std::cout << this->key[i];
    }
    std::cout << '\t';
    for (int i = 0; i < this->valueSize; ++i) {
        std::cout << this->value[i];
    }
    std::cout << '\n';
}

void TElem::Set(std::string &k, std::string &v) {
    for (size_t i = 0; i < KEY_LENGTH; ++i) {
        this->key[i] = k[i];
    }
    for (size_t i = 0; i < v.size(); ++i) {
        this->value[i] = v[i];
    }
    this->valueSize = v.size();
}

int HexadecimalToInt(char s) {
    int ans;
    if((s >= '0') && (s <= '9')) {
        ans = s - '0';
    }
    else if((s >= 'a') && (s <= 'f')) {
        ans = s - 'a' + 10;
    }
    else if((s >= 'A') && (s <= 'F')) {
        ans = s - 'A' + 10;
    }
    return ans;
```

```
50 }
51
52 void RadixSort(TElem *ar, int arSize) {
53     TElem *newAr = new TElem[arSize];
54     for(int j = KEY_LENGTH - 1; j >= 0; --j) {
55         // CountSort
56         int count[16];
57         for(int k = 0; k < 16; ++k) {
58             count[k] = 0;
59         }
60         for(int i = 0; i < arSize; ++i) {
61             int idx = HexadecimalToInt(ar[i].key[j]);
62             ++count[idx];
63         }
64         int prev = 0;
65         for(int k = 0; k < 16; ++k) {
66             count[k] += prev;
67             prev = count[k];
68         }
69         for(int i = arSize - 1; i >= 0; --i) {
70             int idxInCount = HexadecimalToInt(ar[i].key[j]);
71             int idxInResult = count[idxInCount] - 1;
72             newAr[idxInResult] = ar[i];
73             --count[idxInCount];
74         }
75         TElem *tmp = ar;
76         ar = newAr;
77         newAr = tmp;
78     }
79
80     delete[] newAr;
81 }
82
83 int main() {
84     TElem *ar = new TElem[MAX_ARRAY_SIZE];
85     int arSize = 0;
86     std::string key;
87     std::string value;
88     while(std::cin >> key >> value) {
89         ar[arSize].Set(key, value);
90         ++arSize;
91     }
92     RadixSort(ar, arSize);
93     for (int i = 0; i < arSize; ++i) {
94         ar[i].PrintElem();
95     }
96     delete[] ar;
97     return 0;
98 }
```

3 Консоль

```
$ cat tests/1.t
8cd4ecede13bee9597eeda9f7fb6f235 VZwCTtMq
1372eee6db6a75bc27eb22293318f791 LYHmwpjf
62d7aad87ca23c92b8dcedbff7e3e33b DYWySEAN
962545cb1e5d6ac498715dd2f2418277 MtQCxQek
76e7d2c18ef6bfeaae154b986e716aa8 UcmusLio
7aa6ebcb3e67453dbdf51614d461329b qoPiFQYV
8f55ba73232d69d1ffd858a7a154898f VBiOHcCs
463cd899f86cb998e2d38c6c57ba18c9 eBjpTzGX
a87ff2cc6a3d97bffddb5bd9f1bd771b CFUqydQG
cd28f4c7b125da45b81675ba9db62f23 pqKlDnxg
$
$
$ g++ -std=c++20 -pedantic -Wall -Wextra -Wno-unused-variable radix_sort.cpp -o lab1
    && ./lab1 < tests/1.t
1372eee6db6a75bc27eb22293318f791 LYHmwpjf
463cd899f86cb998e2d38c6c57ba18c9 eBjpTzGX
62d7aad87ca23c92b8dcedbff7e3e33b DYWySEAN
76e7d2c18ef6bfeaae154b986e716aa8 UcmusLio
7aa6ebcb3e67453dbdf51614d461329b qoPiFQYV
8cd4ecede13bee9597eeda9f7fb6f235 VZwCTtMq
8f55ba73232d69d1ffd858a7a154898f VBiOHcCs
962545cb1e5d6ac498715dd2f2418277 MtQCxQek
a87ff2cc6a3d97bffddb5bd9f1bd771b CFUqydQG
cd28f4c7b125da45b81675ba9db62f23 pqKlDnxg
```

4 Тест производительности

Тест производительности представляет из себя следующее: измеряем время выполнения std::stable_sort() и radix_sort() - реализованной поразрядной сортировки - на одном и том же массиве данных. Далее сравниваем времена.
Оценим сложность реализованной поразрядной сортировки. Пусть N - количество элементов в исходном массиве, p - основание системы счисления, в которой представлены ключи(в нашем случае 16-ричные числа), k - максимальная длина ключа(в нашем случае все ключи имеют фиксированную длину - 32 знака).
Главный цикл перебирает все разряды ключей, поэтому его сложность - O(k). Для каждого разряда мы запускаем сортировку подсчетом, в которой дважды осуществляется проход по N элементам массива, а также по p элементам вспомогательного массива. Итоговая сложность составляет O(k(2N+p)). При N >> p и N >> k сложность превращается в O(N), то есть в линейную. Однако на малых N константа довольно большая.

```
1  #include <iostream>
2  #include <chrono>
3  #include <algorithm>
4  #include "sort.hpp"
5
6  using duration_t = std::chrono::microseconds;
7  const std::string DURATION_PREFIX = "us";
8
9  bool cmp(TElem el1, TElem el2) {
10     return el1.key < el2.key;
11 }
12
13 int main() {
14     TElem *input = new TElem[MAX_ARRAY_SIZE];
15     TElem *input_stl = new TElem[MAX_ARRAY_SIZE];
16
17     std::string key, value;
18     int arSize = 0;
19     while (std::cin >> key >> value) {
20         input[arSize].Set(key, value);
21         ++arSize;
22     }
23     for(int i = 0; i < arSize; ++i) {
24         input_stl[i] = input[i];
25     }
26     std::cout << "Count of lines is " << arSize << std::endl;
27
28     std::chrono::time_point<std::chrono::system_clock> start_ts = std::chrono::
           system_clock::now();
29     NSort::RadixSort(input, arSize);
30     auto end_ts = std::chrono::system_clock::now();
```

```
31      uint64_t RadixSort_ts = std::chrono::duration_cast<duration_t>( end_ts - start_ts )
            .count();
32
33      start_ts = std::chrono::system_clock::now();
34      std::stable_sort(input_stl, input_stl + arSize, cmp);
35      end_ts = std::chrono::system_clock::now();
36
37      uint64_t stl_sort_ts = std::chrono::duration_cast<duration_t>( end_ts - start_ts ).
            count();
38      std::cout << "Radix sort time: " << RadixSort_ts << DURATION_PREFIX << std::endl;
39      std::cout << "STL stable sort time: " << stl_sort_ts << DURATION_PREFIX << std::
            endl;
40
41      delete[] input;
42      delete[] input_stl;
43  }
```

```
1   $ g++ -std=c++20 -pedantic -Wall -Wextra -Wno-unused-variable banchmark.cpp sort.cpp -
        o banchmark
2   $ ./banchmark < 07.t
3   Count of lines is 100000
4   Radix sort time: 194484us
5   STL stable sort time: 43548us
6   $ ./banchmark < 06.t
7   Count of lines is 10000
8   Radix sort time: 21722us
9   STL stable sort time: 3281us
10  $ ./banchmark < 05.t
11  Count of lines is 100
12  Radix sort time: 208us
13  STL stable sort time: 21us
14  $ ./banchmark < tests/06.t
15  Count of lines is 8010000
16  Radix sort time: 11214341us
17  STL stable sort time: 4719321us
```

Как можно заметить, с ростом количества элементов уменьшается разница во времени исполнения. Хоть наша сортировка и работает за линейное время, однако у нее довольно большая константа, поэтому сортировка из стандартной бибилотеки при небольших размерах массива работает в 2,5 раза быстрее. Разумеется, на очень больших размерах наступит момент, когда поразрядная сортировка обгонит std::stable_sort(), сложность которой O(NlogN).

5 Выводы

Выполнив первую лабораторную работу по курсу «Дискретный анализ», я реализовал поразрядную сортировку, которая работает за линейное время. Как правило, она применяется в случаях, когда ключи(в качестве которых можно представить числа, строки) имеют довольно большую длину. При реализации алгоритма я обратил внимание на то, что нет нужды сравнивать непосредственно сами ключи. Мы постепенно формируем ответ, сравнивая их фрагменты.
Данную сортировку можно будет применить, когда необходимо, например, использовать бинарный поиск, для которого строго необходим критерий отсортированности элементов.
Также было интересно применять wrapper.sh, banchmark.cpp и generator.py для автоматизации процесса тестирования.

Список литературы

[1] Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. *Алгоритмы: построение и анализ, 2-е издание.* — Издательский дом «Вильямс», 2007. Перевод с английского: И. В. Красиков, Н. А. Орехова, В. Н. Романов. — 1296 с. (ISBN 5-8459-0857-4 (рус.))