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: С. А. Арапов
Преподаватель: Н. С. Капралов
Группа: М8О-208Б-19

Дата: 23.10.2020
Оценка:
Подпись:

Москва, 2020

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: `Автомобильные номера в формате A 999 BC (используются буквы латинского алфавита).`

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

Основная идея поразрядной сортировки заключена в том, чтобы перебрать элементы по порядку и перегруппировать в зависимости от того, какая цифра окажется в оперделённом разряде числа. Возникает новая последовательность, которую мы снова группируем, но уже по следующему с конца разряду и так далее пока не будут перебраны все разряды числа. Обработка чисел происходит от младшего разряда к старшему.

Для сортировки нам необходимо знать два параметра: k - количество разрядов в самом длинном ключе, m - количество возможных значений разряда ключа.

Пускай есть исходный массив данных $data$. Создаём массив счётчиков $counters$ из m элементов и инициализируем его нулями. Проходимся по массиву сортируемых элементов и присваиваем каждому элементу $counters[i]$ количество элементов сортируемого массива, которые равны i. Теперь прибавляем к каждому элементу $counters$ сумму значений всех предыдущих элементов. Теперь можно произвести расстановку чисел, ведь мы знаем сколько элементов меньше данного, значит знаем его место в упорядоченном массиве. Будем идти справа налево по $data$,одновременно заполняя выходной массив $result$. При проходе выполняем следующие действия: $result[counters[i] - 1] = data[i]$ и $counters[data[i]] - -$.Первым действием заполняем выходной массив. Второе действие необходимо на случай, если есть повторы.

По итогу получим массив $result$, элементы которого отсортированы по одному из разрядов ключа. Приэтом каждый раз нас интересует самый младший и ещё не сортированный разряд.

Оценим сложность поразрядной сортировки. Положим m - количество разрядов, n - количество сортируемых объектов. Необходимо считать все элементы, создать массив счётчиков, заполнить массив счётчиков, прибавить к каждому счётчику сумму предыдущих ему счётчиков, заполнить выходной массив. Итак имеем: $O(n) + O(1) + O(n) + O(m) + O(n) = 3 * O(n) + O(m) + O(1) = O(3n + m) = O(n)$

2 Исходный код

Входные данные поступают в виде пары «ключ-значение», поэтому создаём класс $TItem$. В каждом объекте этого класса будет храниться число номера типа int, буквы номера типа $char[3]$ и значения типа $char[65]$(нужно добавить символ окончания строки).

Так же необходимо создать динамический массив $TVector$ для хранения всех данных, потому как заранее не известно, сколько строк будет подано для обработки. Объект этого класса содержит в себе указатель на динамический массив, вместимость и его текущий размер. Конструктор по умолчанию создаст динамический массив на 16 элементов. Так же можно воспользоваться конструктором, в который передаётся параметр типа int, который будет означать текущий размер создаваемого массива. Если количество элементво станет больше 16, то вместимость увеличивается вдвое. Есть методы позволяющие добавить элемент в конец массива и узнать текущий размер. Деструктор освобождают всю выделенную память. $TVector$ имеет шаблонную реализацию, поэтому в нём можно хранить объекты любого типа.

Входные данные сохраняются в $TVector < TItem > data$. Создаётся вспомогательный динамический массив $result[data]$ для хранения промежуточного результата.

Функция $RadixSort$ применяется для сортировки $data$. Её аргументами являются два массива: массив данных и вспомогательный массив. В основе реализации $RadixSort$ находятся функции $SortChar$ и $SortInteger$, которые крайне схожи в своей логической реализации. $SortChar$ применяет поразрядную сортировку к типу $char$, аргументами являются два массива и число типа int. $SortInteger$ позволяет поразрядно отсортировать числа типа int, аргументами являются два массива. В $SortInteger$ создаётся и инициализируется нулём массив счётчиков длиной 1000, заполняется, к каждому счётчику прибавляется сумма предыдущих, числа расставляются во вспомогательном массиве соответствии со своей позицией. Работа функции $SortChar$ аналогична. В $RadixSort$ эти функции применяются последовательно и это позволяет на выходе получить отсортированный массив.

```cpp
#include <iostream>
#include <cstdlib>
#include <iomanip>

class TItem{
public:
    int numeric = 0;
    char key[3]={'A','A','A'};
    char value[65] = "";
    void printer(){
        std::cout << key[0] << " ";
  std::cout << std::setfill('0') << std::setw(3);
  std::cout << numeric << " ";
  std::cout << key[1] << key[2] << "\t" ;
  std::cout << value << "\n";
    }
};

template <typename T>
class TVector{
private:
    T* Data;
    int Capacity;
    int CurSize;

    void Extend(){
        Data = (T*)realloc(Data, 2 * Capacity * sizeof(T));
        Capacity *= 2;
    }

public:
    TVector(){
        Data = (T*)calloc(64, sizeof(T));
        Capacity = 16;
        CurSize = 0;
    }

    TVector(int size){
        Data = (T*)calloc(size, sizeof(T));
        Capacity = size;
        CurSize = size;
    }

    ~TVector(){
        free(Data);
    }

    int Size(){
        return CurSize;
```

```
50      }
51
52      void PushBack(T elem){
53          if (CurSize == Capacity){
54              Extend();
55          }
56
57          Data[CurSize] = elem;
58          CurSize++;
59      }
60
61      T& operator[] (int i){
62          return Data[i];
63      }
64
65  };
66
67  void SortChar(TVector <TItem>& data, TVector <TItem>& result, int j){
68      int counter[26] = {0};
69
70      for(int i = 0; i < data.Size(); i++){
71          counter[ (int)data[i].key[j] - 65 ]++;
72      }
73
74      for(int i = 1; i < 26; i++){
75          counter[i] += counter[i-1];
76      }
77
78      for(int i = data.Size() - 1; i > -1; i--){
79          result[ counter[(int)data[i].key[j] - 65] - 1] = data[i];
80          counter[(int)data[i].key[j] - 65]--;
81      }
82  }
83
84  void SortInteger(TVector <TItem>& data, TVector <TItem>& result){
85      int counter[1000] = {0};
86
87      for(int i = 0; i < data.Size(); i++){
88          counter[data[i].numeric]++;
89      }
90
91      for(int i = 1; i < 1000; i++){
92          counter[i] += counter[i-1];
93      }
94
95      for(int i = data.Size() - 1; i > -1; i--){
96          result[ counter[data[i].numeric] -1] = data[i];
97          counter[data[i].numeric]--;
98      }
```

```
99
100 }
101
102 void RadixSort(TVector <TItem>& data, TVector <TItem>& result){
103     SortChar(data, result, 2);
104     SortChar(result, data, 1);
105     SortInteger(data, result);
106     SortChar(result, data, 0);
107 }
108
109 int main(){
110     std::ios_base::sync_with_stdio(false);
111     std::cin.tie(nullptr);
112     TItem elem;
113     TVector <TItem> data;
114
115      while(std::cin >> elem.key[0] >> elem.numeric >> elem.key[1]>> elem.key[2] >> elem
            .value)
116     {
117         data.PushBack(elem);
118     }
119     TVector <TItem> result(data.Size());
120     RadixSort(data, result);
121
122     for (int i = 0; i < data.Size(); i++){
123         data[i].printer();
124     }
125 }
```

3 Консоль

```
rocket@LAPTOP-ADULJM7A:~$ sudo apt install gitsome
[sudo] password for rocket:

rocket@LAPTOP-ADULJM7A:~$ ls
test
rocket@LAPTOP-ADULJM7A:~$ mkdir da_lab1
rocket@LAPTOP-ADULJM7A:~$ cd da_lab1
rocket@LAPTOP-ADULJM7A:~/da_lab1$ ls
rocket@LAPTOP-ADULJM7A:~/da_lab1$ gh repo clone ogrocket/DA_LAB_1
Usage: gh repo [OPTIONS] USER_REPO
Try "gh repo --help" for help.

Error: Got unexpected extra argument (ogrocket/DA_LAB_1)
rocket@LAPTOP-ADULJM7A:~/da_lab1$ gh repo clone ogrocket/DA_LAB_1
Usage: gh repo [OPTIONS] USER_REPO
Try "gh repo --help" for help.

Error: Got unexpected extra argument (ogrocket/DA_LAB_1)
rocket@LAPTOP-ADULJM7A:~/da_lab1$ https://github.com/ogrocket/DA_LAB_1.git
-bash: https://github.com/ogrocket/DA_LAB_1.git: No such file or directory
rocket@LAPTOP-ADULJM7A:~/da_lab1$ git clone https://github.com/ogrocket/DA_LAB_1.git
Cloning into 'DA_LAB_1'...
Username for 'https://github.com': ogrocket
Password for 'https://ogrocket@github.com':
remote: Enumerating objects: 10,done.
remote: Counting objects: 100% (10/10),done.
remote: Compressing objects: 100% (8/8),done.
remote: Total 10 (delta 2),reused 9 (delta 1),pack-reused 0
Unpacking objects: 100% (10/10),1.83 MiB | 1.74 MiB/s,done.
rocket@LAPTOP-ADULJM7A:~/da_lab1$ ls
DA_LAB_1
rocket@LAPTOP-ADULJM7A:~/da_lab1$ cd DA_LAB1
-bash: cd: DA_LAB1: No such file or directory
rocket@LAPTOP-ADULJM7A:~/da_lab1$ cd DA_LAB_1
rocket@LAPTOP-ADULJM7A:~/da_lab1/DA_LAB_1$ ls
bench.cpp  lib.hpp  main.cpp  test  test_generator.ipynb
rocket@LAPTOP-ADULJM7A:~/da_lab1/DA_LAB_1$ g++ main.cpp
rocket@LAPTOP-ADULJM7A:~/da_lab1/DA_LAB_1$ ls
rocket@LAPTOP-ADULJM7A:~/da_lab1/DA_LAB_1$ head 10 test
```

```
head: cannot open ’10’for reading: No such file or directory
==>test <==
E 591 JQ ASKHAMCVGIXXFMAJJJYGDUWMGIFSVYFISZIBINDTSTJ
Z 529 PA RDCSZJPDUSXNHGGRQBDURGUUGYUY
V 102 TF PXEZGDYLKEKE
F 912 HH FMEAHQJTKDMEEYUFVNIVSMGNGKWCSMSYEOJTTCNIXCVETLABPYVBMN
D 642 YM YCIXTXHGLBCULJQFCKSDZKMRKMDATZUSWRKVRXBIOXLEFXSXYWKBPOZ
Z 326 AS FIEZKVCHRIREYVZGESFOEUEAESEAHVQEJDDRFIZOT
N 937 VE ELAXDXZZU
W 827 ND XECAIXIYIAFWABVWHL
C 200 IR ZYIMCYPPCJEZSZXTZDTZCHIVVLNRIZKBPQHZPZFABOBTKRXWMMELICYN
V 294 CV EBTNARZQML
rocket@LAPTOP-ADULJM7A:~/da_lab1/DA_LAB_1$ g++ main.cpp
rocket@LAPTOP-ADULJM7A:~/da_lab1/DA_LAB_1$ ls
a.out  bench.cpp  lib.hpp  main.cpp  test  test_generator.ipynb
rocket@LAPTOP-ADULJM7A:~/da_lab1/DA_LAB_1$ ./a.out <test
A 000 FZ        XGASVWMXAHEIUANIGLBU
A 000 JC        LHRHGFGZJKK
A 000 RL        EHNCKXIYCJLKYCSGZQ
A 000 TO        VIUXIZWSNCCEUSQASAROQBGPG
A 002 MP        PLCJTDNSIBBIJMORWZBMCVQTNLYLHICLPTJWZVP
A 002 SV        JRQLUFUJMNZKICPKRZSXQFLRWYVJDFRZXDFOVUSWVGJMOTXBYGJJVVKNN
A 003 AY        WDKDTHBOHFRXPAXPCTQTSQSOUCABJTVHDSQGWZNMEOTVGYKQMNK
A 004 ID        MHLJYALHROFZC
A 004 NO        JHUYJBNXPUHVHAZHDROTPCDJTV
A 004 SJ        BEGVZNX
A 004 UL        SEDVGAZFBSLLVSDRKFAISRDPTRHMBMYPDRKRCLHTQLZCUTQLYYEUY
A 004 WZ        PFYOMPSMIBOHBUYWPACBXFHMHDHGUVWOEDHLDKARSWMLNUAEWITDUBWOLJJ
...
```

4 Тест производительности

Для сравнения скорости работы сортировки я написал программу *bench.cpp*, в которой воспользовался средствами стандартной библиотеки C++, а именно алгоритм устойчивой сортировки $std :: stable_sort$. Подготовил тестовый файл, с помощью программы $test_generator.ipynb$, которая генерирует тесты. Количество строк в тестовом файле находится в диапозоне от 10000 до 100000.

Листинг *bench.cpp*:

```
#include "lib.hpp"
#include <iostream>
#include <algorithm>
#include <chrono>
#include <cstdlib>
#include <iomanip>

int main(){
    std::ios_base::sync_with_stdio(false);
    std::cin.tie(nullptr);

    TItem elem;
    TVector <TItem> data;

    while(std::cin >> elem.key[0] >> elem.numeric >> elem.key[1]>> elem.key[2] >> elem.
        value){
        data.PushBack(elem);
    }
    TVector <TItem> result(data.Size());
    TVector <TItem> data2(data.Size());
    for (int i = 0; i < data.Size(); i++){
        data2[i] = data[i];
    }

    auto start = std::chrono::system_clock::now();
    RadixSort(data, result);
    auto finish = std::chrono::system_clock::now();
    std::cout << std::chrono::duration_cast<std::chrono::milliseconds>(finish - start).
        count() << " :RadixSort Time in ms\n";

    auto start_2 = std::chrono::system_clock::now();
    std::stable_sort(data2.Begin(), data2.End());
    auto finish_2 = std::chrono::system_clock::now();

    std::cout << std::chrono::duration_cast<std::chrono::milliseconds>(finish_2 -
        start_2).count() << " :STL Sort Time in ms\n";

    std::cout << "Number of lines in test " << data.Size() << " \n";

```

```
37
38          return 0;
39  }
```

Листинг *test_generator.ipynb*:

```
1  import random
2  import string
3
4  K = random.randint(10_000, 100_000)
5  file = open("test", "w")
6
7  for _ in range(K):
8      file.write(chr(random.randint(65,90)))
9      file.write(' ')
10     file.write(str(random.randint(0, 9)))
11     file.write(str(random.randint(0, 9)))
12     file.write(str(random.randint(0, 9)))
13     file.write(' ')
14     file.write(chr(random.randint(65,90)))
15     file.write(chr(random.randint(65,90)))
16     file.write(' ')
17     for __ in range(random.randint(1, 64)):
18         file.write(str(chr(random.randint(65,90))).lower())
19     file.write('\n')
20 file.close
```

Замеры времени работы стандартной сортировки и поразрядной сортировки:

```
rocket@LAPTOP-ADULJM7A:~/da_lab1/DA_LAB_1$ ls
a.out  bench.cpp  lib.hpp  main.cpp  test  test2.txt  test_generator.ipynb
rocket@LAPTOP-ADULJM7A:~/da_lab1/DA_LAB_1$ pwd
/home/rocket/da_lab1/DA_LAB_1
rocket@LAPTOP-ADULJM7A:~/da_lab1/DA_LAB_1$ g++ -o bench bench.cpp
rocket@LAPTOP-ADULJM7A:~/da_lab1/DA_LAB_1$ ls
a.out  bench  bench.cpp  lib.hpp  main.cpp  test  test2.txt  test_generator.ipynb
rocket@LAPTOP-ADULJM7A:~/da_lab1/DA_LAB_1$ ./bench <test
9 :RadixSort Time in ms
25 :STL Sort Time in ms
Number of lines in test 70231
rocket@LAPTOP-ADULJM7A:~/da_lab1/DA_LAB_1$ ./bench <test
9 :RadixSort Time in ms
26 :STL Sort Time in ms
Number of lines in test 70231
rocket@LAPTOP-ADULJM7A:~/da_lab1/DA_LAB_1$ ./bench <test
9 :RadixSort Time in ms
```

```
26 :STL Sort Time in ms
Number of lines in test 70231
rocket@LAPTOP-ADULJM7A:~/da_lab1/DA_LAB_1$
```

По результатам теста видно, что поразрядная сортировка даёт существенный выигрышь. Происходит это потому-что сложность у поразрядной сортировки $O(n)$, а у устойчивой $O(n * log(n))$

5 Выводы

Проделав данную первую лабораторную работу по курсу «Дискретный анализ», я понял, для чего нужны сортировки за линейное время. Всё дело в том что стандартные сортировки, основанные на сравнении имеют сложность $O(n * log(n))$, поэтому нужно придумать какой-то более совершенный способ, который не будет использовать сравнения и поэтому работать быстрее.

Поразрядная сортировка, которую мне пришлось изучить и реализовать, работает за линейное время. Однако при этом появляется существенный минус - затраты дополнительной памяти. Иногда это может быть очень существенным.

Еще одним минусом является ограниченная область применимости сортировок за линейное время. Не получится отсортировать строки или абстрактные типы данных.

Именно из-за таких недостатков линейных сортировок в стандартной библиотеке C++ используются более общные сортировки , хотя они и менее быстрые.

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.))