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: Д. С. Ляшун
Преподаватель: А. А. Кухтичев
Группа: М8О-207Б

Дата:
Оценка:
Подпись:

Москва, 2021

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.

Тип сортировки: Поразрядная.

Тип ключа: Телефонные номера в формате +<код страны>-<код города>-телефон.

Тип 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. Как сказано в [1], поразрядная сортировка (англ. radix sort) — один из алгоритмов сортировки, использующих внутреннюю структуру сортируемых объектов. Его идея заключается в том, чтобы разбить исходный объект на разряды, от младшего к старшему. На каждом шаге алгоритма происходит последовательная сортировка объектов по значению текущего рассматриваемого разряда (от младшего к старшему). Т.к. количество значений в разряде обычно небольшое, применяется сортировка подсчётом, работающая за линейное время $O(n + k)$, где n – количество сортируемых объектов, k – количество различных значений разряда, тогда общее время работы поразрядной сортировки составляет $O(m(n + k))$, где n – количество сортируемых объектов, k – количество разрядов в сортируемых объектах, m – количество разрядов.

2 Исходный код

Для хранения входных объектов в программе определим структуру данных TVector:

```
template<class T>
class TVector {
    public :
        void Assert(const T& elem);
        TVector();
        TVector(const unsigned int newSize);
        TVector(const unsigned int newSize, const T& elem);
        ~TVector();
        void PushBack(const T& elem);
        unsigned int GetSize();
        T& operator[](const unsigned int index);
        TVector<T>& operator=(const TVector<T>& vector);
    private :
        unsigned int Size;
        unsigned int Capacity;
        T* Data;
    };
}
```

В классе TVector были описаны следующие методы:

vector.hpp	
void Assert(const T& elem)	Процедура, перезаписывающая все значения вектора на elem.
TVector(const unsigned int newSize)	Конструктор, создающий пустой вектор размера newSize.
TVector(const unsigned int newSize, const T& elem)	Конструктор, создающий вектор с элементом elem в количестве newSize.
void PushBack(const T& elem)	Процедура, закладывающая в конец вектора элемент elem.
unsigned int GetSize()	Функция, возвращающая текущий размер вектора.
T& operator[](const unsigned int index)	Оператор [] для получения элемента в векторе по индексу index.
TVector<T>& operator=(const TVector<T>& vector)	Оператор = для присваивания векторов.
void Convert(char* str)	Производит перевод всех букв символьной строки str в нижний регистр.
void Assign(char* fir, char* sec)	Производит присваивание строки fir значения строки sec.

Ниже приведен исходный код файла radix_sort.hpp, в котором были описаны: структура TItem, представляющая входные объекты, а также сам алгоритм поразрядной сортировки.

```
#pragma once
#include "vector.hpp"
using namespace NMyStd;
const int SIZE_STR = 65;
const int CODE_CNTR_COUNT_DIGITS = 3;
const int CODE_CITY_COUNT_DIGITS = 3;
const int NUMBER_COUNT_DIGITS = 7;
const int ALL_NUMBER_DIGITS = CODE_CNTR_COUNT_DIGITS + CODE_CITY_COUNT_DIGITS +
    NUMBER_COUNT_DIGITS;
const int EXTRA_SYMBOLS = 4;
const int COUNT_DIGITS = 10;
struct TData {
    long long Key;
    char Number[ALL_NUMBER_DIGITS + EXTRA_SYMBOLS];
    char Str[SIZE_STR];
};
void RadixSort(TVector<TData>& data) {
    TVector<int> bitValues(COUNT_DIGITS);
    TVector<TData> result(data.GetSize());
    int flag = 0;
    for (int i = ALL_NUMBER_DIGITS - 1; i >= 0; --i) {
        bitValues.Assert(0);
        for (int j = 0; j < data.GetSize(); ++j) {
            if (flag == 0) {
                long long digit = data[j].Key % COUNT_DIGITS;
                ++bitValues[digit];
            }
            else if (flag == 1) {
                long long digit = result[j].Key % COUNT_DIGITS;
                ++bitValues[digit];
            }
        }
        for (int j = 1; j < COUNT_DIGITS; ++j) {
            bitValues[j] += bitValues[j-1];
        }
        for (int j = data.GetSize() - 1; j >= 0; --j) {
            if (flag == 0) {
                long long digit = data[j].Key % COUNT_DIGITS;
                result[--bitValues[digit]] = data[j];
                result[bitValues[digit]].Key /= COUNT_DIGITS;
            }
            else if (flag == 1) {
                long long digit = result[j].Key % COUNT_DIGITS;
                data[--bitValues[digit]] = result[j];
                data[bitValues[digit]].Key /= COUNT_DIGITS;
            }
```

```
46          }
47          flag = (flag ? 0 : 1);
48      }
49      if (flag == 1) {
50          data = result;
51      }
52  }
```

В исходном коде из файла main.cpp описан ход выполнения всей программы:

```
1   #include "vector.hpp"
2   #include "radix_sort.hpp"
3   #include <stdio.h>
4   using namespace NMyStd;
5   int main() {
6       TVector<TData> data;
7       TData input;
8       while (scanf("%s\t%s\n", input.Number, input.Str) > 0) {
9           int len = 0;
10          for (int i = 0; i < ALL_NUMBER_DIGITS+EXTRA_SYMBOLS; ++i) {
11            if (input.Number[i] == '\0') {
12                len = i - 1;
13                break;
14            }
15        }
16        long long power = 1;
17        input.Key = 0;
18        while (len != 0) {
19            if (input.Number[len] != '-') {
20                input.Key += (input.Number[len] - '0') * power;
21                power *= 10;
22            }
23            --len;
24        }
25          data.PushBack(input);
26      }
27      RadixSort(data);
28      for (int i = 0; i < data.GetSize(); ++i) {
29          printf("%s\t%s\n", data[i].Number, data[i].Str);
30      }
31      return 0;
32  }
```

Алгоритм решения задачи можно разбить на следующие этапы:

1. Чтение входных пар ключ-значение и их запись в вектор data, при этом номер телефона для удобства представим в виде десятичного числа.

2. Сортировка содержимого вектора data по разрядам ключа. В цикле по общему количеству разряду в номере на каждой его итерации осуществляем сортировку

по соответствующему разряду следующим образом:

(a) Обнуляем содержимое вспомогательного вектора bit_values, который будет хранить количество каждой цифры в данном обрабатываемом разряде для всех элементов сортируемого вектора.

(b) Заполняем вектор bit_values значениями, т.е. подсчитываем количество встречаемости каждой цифры на данном разряде для всех элементов вектора.

(c) Пересчитываем значения в векторе bit_values, где в ячейке с индексом i теперь будет храниться префиксная сумма элементов bit_values с индексами от 0 до i включительно, что означает сколько элементов слева (включая себя) находится от последнего элемента с цифрой i после сортировки по данному разряду.

(d) Заполняем вектор result или data. Какой из них будет использован зависит от значения flag, который будет меняться на противоположное значение на каждой итерации поразрядной сортировки. Это сделано с целью увеличения быстродействия программы, поскольку присваивание векторов происходит медленно. Чтобы сортировка являлась устойчивой, по неотсортированному вектору будем проходить с конца. В свою очередь для заполнения будут использоваться значения в bit_values, поскольку при обработке всякого поля если значение его цифры в текущем разряде ключа равно i, то позиция в отсортированном векторе будет равна bit_values[i]. После обработки текущего поля значение bit_values необходимо уменьшить на 1, чтобы для следующего поля с такой же цифрой можно было узнать его новую позицию.

(e) В конце итерации происходит смена значение flag, таким образом на следующей итерации мы начнём работать с отсортированным вектором.

3. В конце программы осуществляется вывод результата - содержимого отсортированного вектора data.

Оценим сложность полученного алгоритма. Самым продолжительным этапом работы является проведение поразрядной сортировки, который проходит в цикле по общему количеству разрядов r, на каждой итераций которого происходит: обнуление вектора bit_values за $O(c)$, где c - количество всех возможных цифр, его заполнение за $O(n)$, где n - количество входных полей, вычисление префиксной суммы для bitbit_values за $O(c)$, и заполнение отсортированного по данному разряду вектора за $O(n)$. Т.к. общее количество разрядов и количество всех возможных цифр является константным и равняется 13 и 10 соответственно, получаем, что итоговая асимптотика алгоритма составляет $O(n)$.

3 Консоль

```
dmitry@dmitry-VirtualBox:~/DA_labs/$ make
g++ -std=c++11 -O2 -Wextra -Wall -Werror -Wno-sign-compare -Wno-unused-result
-pedantic -o solution main.cpp
dmitry@dmitry-VirtualBox:~/Work_place/DA_labs/HARD$ cat test1.txt
+9-131-1162528 vdvv7AhrCv
+5-928-2809089 IYAkUhKVCn
+1-016-0153060 sxqpV3Es5j
+6-917-1603326 vjdqDVrVvb
+1-944-7490851 WvmoAv
dmitry@dmitry-VirtualBox:~/DA_labs/$ ./solution <test1.txt
+1-016-0153060 sxqpV3Es5j
+1-944-7490851 WvmoAv
+5-928-2809089 IYAkUhKVCn
+6-917-1603326 vjdqDVrVvb
+9-131-1162528 vdvv7AhrCv
dmitry@dmitry-VirtualBox:~/DA_labs/$ cat test2.txt
+9-875-0555322 nb
+5-129-9332849 cSvsv
+8-099-6682106 ENK6COvvvv
+7-033-8482465 wsPDv
+4-805-4788604 uvfx5ZE5aY
dmitry@dmitry-VirtualBox:~/DA_labs/$ ./solution <test2.txt
+4-805-4788604 uvfx5ZE5aY
+5-129-9332849 cSvsv
+7-033-8482465 wsPDv
+8-099-6682106 ENK6COvvvv
+9-875-0555322 nb
dmitry@dmitry-VirtualBox:~/DA_labs$
```

4 Тест производительности

Тест производительности представляет из себя сравнение времени работы написанной поразрядной сортировки и сортировки из STL – std::stable_sort. Тестирование проводилось на объектах, имеющих ключ размера 9-11 разрядов, а длину значения-строки равную 30-40 символам. Количество объектов составляло тысячу, сто тысяч и миллион.

```
dmitry@dmitry-VirtualBox:~/DA_labs/$ g++ benchmark -o benchmark
dmitry@dmitry-VirtualBox:~/DA_labs/$ ./benchmark <test1000.txt
Radix sort time: 3,187ms
STL stable sort time: 2,474ms
dmitry@dmitry-VirtualBox:~/DA_labs/$ ./benchmark <test10000.txt
Radix sort time: 294,833ms
STL stable sort time: 381,223ms
dmitry@dmitry-VirtualBox:~/DA_labs/$ ./benchmark <test1000000.txt
Radix sort time: 2520,44ms
STL stable sort time: 4628,866ms
```

Можно сделать вывод, что написанная поразрядная сортировка работает быстрее по сравнению с устойчивой только при обработки большого количества объектов, однако значительно уступает ей при сортировке небольшого количества. Это объясняется тем, что время работы поразрядной сортировки, равное примерно $T(kdn) = T(130n)$ при $k = 13$ (длина ключа-номера) и $d = 10$ (число различных значений разряда), превышает среднее время работы std::stable_sort $T(knlog_2n) = T(13nlog_2n)$ при $log_2n < 10$ или $n < 1024$.

5 Выводы

Выполнив первую лабораторную работу по курсу «Дискретный анализ», я научился реализовывать собственный алгоритм поразрядной сортировки, который можно применить при работе с различными базами данных, содержащими телефоны-ключи (например, телефонные справочные или номера сотрудников в компании), когда возникает потребность в упорядочивании их содержимого быстро и эффективно.

В целом, написание алгоритма не вызывает особых трудностей, принцип работы поразрядной сортировки интуитивно понятен, однако при отладке программы можно столкнуться с такими проблемами как потеря памяти при неправильном её освобождении в векторе, неправильный ввод и вывод данных (потеря ведущих нулей при обратном выводе номеров после сортировки).

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.))