Московский авиационный институт (национальный исследовательский университет)

Институт информационных технологий и прикладной математики
«Кафедра вычислительной математики и программирования»

Лабораторная работа №1
по предмету "Дискретный анализ"

Студент: Елистратова П.А.
Преподаватель: Макаров Н.К.
Группа: М8О-210Б-21
Дата:
Оценка:
Подпись:

Москва 2023 г.

Оглавление

Цель работы.....3
Постановка задачи.....3
Реализация.....3
Оценка сложности алгоритма.....6
Вывод.....7

Цель работы

Реализовать сортировку за линейное время на C++.

Постановка задачи

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

Вариант: D 2-1:
Сортировка: подсчётом.
Тип ключа: почтовые индексы.
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.

Реализация

Программа принимает на вход пары «ключ-значение», разделённые знаком табуляции, осуществляет их упорядочивание по возрастанию ключа и выводит отсортированные пары в том же виде, как они подавались на вход.

Так как неизвестно количество входных данных, для их хранения реализован свой класс `vectorClass`, с методами `push(unsigned int code, char* input_string)` — добавления элемента, `get(int index)` — получения элемента по индексу и `size()` — для получения длины вектора.

Сама реализация сортировки подсчётом в моём случае следующая:

В вектор `inputArray` считываем наши входные данные, которые описываются для удобства в виде структур `InputData`. Затем вызывается собственно функция сортировки подсчётом — `CountSort(vectorClass<InputData> &inputArray)`. В ней мы определяем вспомогательный массив `countArray[1000000]`, в котором каждый индекс соответствует возможному значению ключа. Сначала для этого массива подсчитываем количество встретившихся ключей во входных данных, затем во втором цикле для каждого входного элемента определяем количество элементов, которые меньше этого элемента. С помощью этой информации элемент можно разместить на той позиции выходного массива, где он должен находиться. Это мы и делаем в третьем цикле: для каждого элемента результирующего массива `sortedArray`, идя с конца исходного переданного массива (для сохранения устойчивости алгоритма) мы находим на какой позиции должен располагаться каждый отсортированный элемент. Возвращаем из функции отсортированный массив `sortedArray`, состоящий из структур `InputData`, и выводим отсортированную последовательность на экран в нужном виде.

Код программы

lab1.cpp

```
#include <iostream>
#include <iomanip>
#include <cstring>


template <typename T>
class vectorClass
{
    T* data;
    int capacity_;
    int current_;

public:
    vectorClass()
    {
        data = new T[1];
        capacity_ = 1;
        current_ = 0;
    }
    ~vectorClass()
    {
        delete[] data;
    }

    void push(unsigned int code, char* input_string)
    {
        if (current_ == capacity_)
        {
            T* temp = new T[2 * capacity_];

            for (int i = 0; i < capacity_; ++i)
            {
                temp[i] = data[i];
            }

            delete[] data;
            capacity_ *= 2;
            data = temp;
        }

        data[current_].postcode = code;
        strcpy(data[current_].value, input_string);
        current_ += 1;
    }

    T get(int index)
    {
        return data[index];
    }

    int size()
    {
        return current_;
    }
};

struct InputData
{
    unsigned int postcode;
    char value[65];
};
```

```
InputData* CountSort(vectorClass<InputData> &inputArray)
{
    unsigned int countArray[1000000] = {0};
    InputData pair;

    for (int i = 0; i < inputArray.size(); ++i)
    {
        pair = inputArray.get(i);
        countArray[pair.postcode] += 1;
    }

    for (int i = 1; i < 1000000; ++i)
    {
        countArray[i] += countArray[i - 1];
    }

    InputData* sortedArray = new InputData[inputArray.size()];

    for (int i = inputArray.size() - 1; i > -1; --i)
    {
        pair = inputArray.get(i);
        sortedArray[countArray[pair.postcode] - 1] = pair;
        countArray[pair.postcode] -= 1;
    }

    return sortedArray;
}


int main(int argc, char *argv[])
{
    std::ios::sync_with_stdio(false);
    std::cin.tie(0);
    std::cout.tie(0);

    vectorClass<InputData> inputArray;
    unsigned int postcode;
    char value[65];

    while (std::cin >> postcode >> value)
    {
        inputArray.push(postcode, value);
    }

    InputData* sortedData = new InputData[inputArray.size()];
    sortedData = CountSort(inputArray);

    for (int i = 0; i < inputArray.size(); ++i)
    {
        std::cout << std::setw(6) << std::setfill('0') << sortedData[i].postcode <<
"\t" << sortedData[i].value << "\n";
    }

    delete[] sortedData;
    return 0;
}
```

Пример работы

```
polina@pelis:~/diskran$ cat input.txt
000000 n399tann9nnt3ttnaaan9nann93na9t3a3t9999na3aan9antt3tn93aat3naatt
999999 n399tann9nnt3ttnaaan9nann93na9t3a3t9999na3aan9antt3tn93aat3naat
000000 n399tann9nnt3ttnaaan9nann93na9t3a3t9999na3aan9antt3tn93aat3naa
999999 n399tann9nnt3ttnaaan9nann93na9t3a3t9999na3aan9antt3tn93aat3na
```

```
polina@pelis:~/diskran$ g++ lab1.cpp
polina@pelis:~/diskran$ ./a.out < input.txt
000000  n399tann9nnt3ttnaaan9nann93na9t3a3t9999na3aan9antt3tn93aat3naatt
000000  n399tann9nnt3ttnaaan9nann93na9t3a3t9999na3aan9antt3tn93aat3naa
999999  n399tann9nnt3ttnaaan9nann93na9t3a3t9999na3aan9antt3tn93aat3naat
999999  n399tann9nnt3ttnaaan9nann93na9t3a3t9999na3aan9antt3tn93aat3na
```

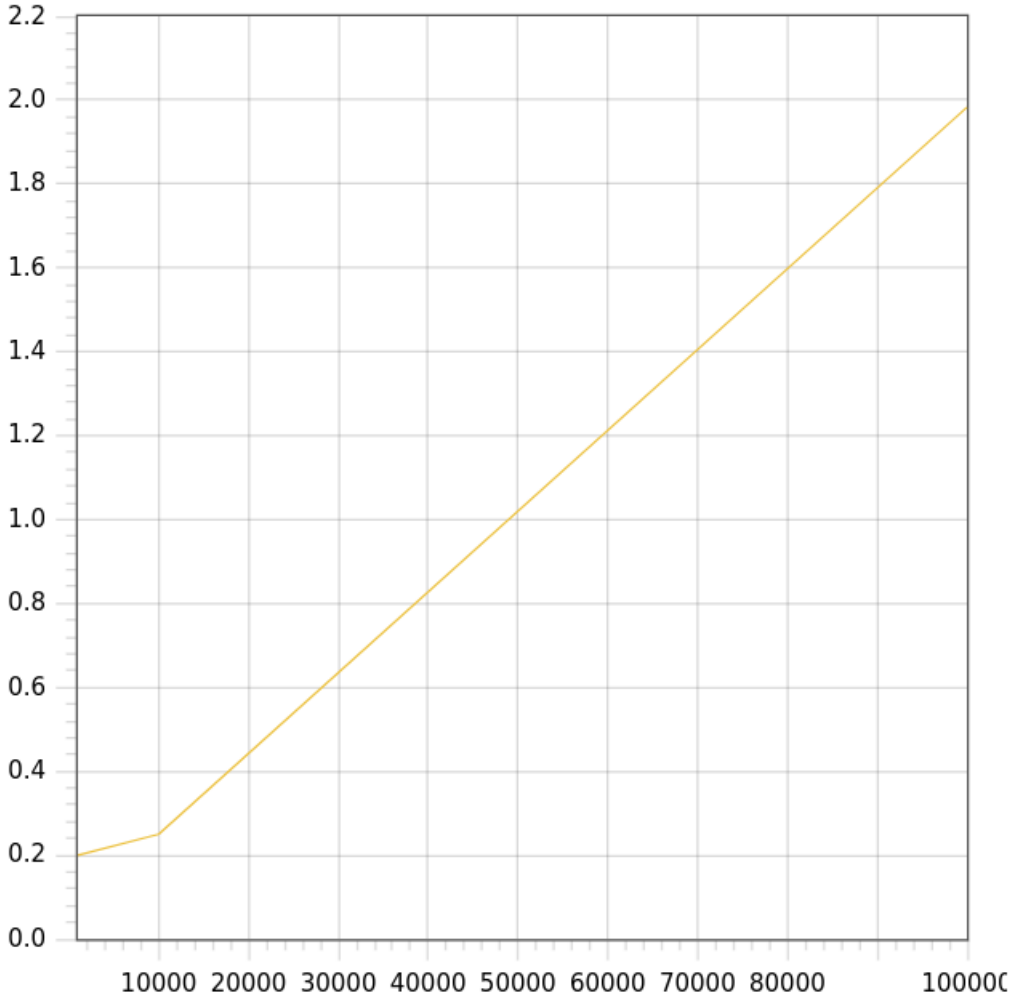
Оценка сложности алгоритма

n — количество входных данных;
m — константа, диапазон целочисленных значений, в пределах которого могут меняться входящие ключи (в моём случае m=1000000, так как ключи от 0 до 999999).
Мы считываем элементы и добавляем их в вектор. Сложность таких манипуляций — O(n). При этом если мы увеличиваем capacity в два раза каждый раз, когда у нас заканчивается место это ещё O(log(n)) операций. Итого сложность последовательного добавления элементов в конец вектора — O(n + log(n)).
Дальше запускается функция сортировки подсчетом, в которой в циклах мы два раза проходимся по исходной последовательности элементов и один раз по вспомогательному массиву. Следовательно сложность функции `CountSort` — O(2 * n + m) .
И последним циклом в теле программы, мы выводим отсортированную последовательность в нужном виде. Сложность — O(n).
Итого, общая сложность алгоритма — O(n + log(n) + 2 * n + m + n) = O(n) — линейная.

Сравнения производительности

Сравним производительность нашего алгоритма с Standart STL Sort (Quick Sort) и с Bubble Sort.

Тип алгоритма сортировки	Сложность алгоритма	Количество входных данных	Время работы программы
Count Sort	O(n)	1000 10000 100000	0,02 0,25 1,983
Standart STL Sort (Quick Sort)	O(n*log(n))	1000 10000 100000	0,013 0,18 2,21
Bubble Sort	$O(n^2)$	1000 10000 100000	0,021 1,75 217,98



Как мы видим на графике зависимости количества входных данных от времени работы программы, сложность у алгоритма сортировки подсчётом действительно линейная.

Вывод

Выполнив данную лабораторную работу, я реализовала сортировку подсчётом, которая работает за линейное время. Использование данной сортировки подразумевает, что все ключи — целые числа, принадлежащие интервалу от 0 до k, где k — некоторая целая константа. Важное свойство алгоритма сортировки подсчетом заключается в том, что он устойчив: элементы с одним и тем же значением находятся в выходном массиве в том же порядке, что и во входном. Однако необходимо помнить, что хоть сортировка подсчетом и работает за линейное время, она требует дополнительной памяти.