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: Е. А. Суханов
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б
Дата:
Оценка:
Подпись:

Москва, 2020

Лабораторная работа №1

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

Вариант сортировки: Поразрядная сортировка.

Вариант ключа: `Даты в формате DD.MM.YYYY, например 1.1.1, 1.9.2009, 01.09.2009, 31.12.2009.`

Вариант значения: `Числа от 0 до` $2^{64} - 1$.

1 Описание

Разобьем задачу на подзадачи:

- Структура «ключ-значение»
- Ввод-вывод струкруты «ключ-значение»
- Тип данных «вектор»
- Поразрядная сортировка вектора
- Сравнение эффективности сортировок

Структура «ключ-значение» будет состоять из двух полей: ключ – еще одна структура, которая хранит день, месяц и год; значение – целое число .

Ввод-вывод этой структуры будет происходить с помощью перегрузки опператоров » и « [1]. Как я заметил позже, требуется сохранять лидирующие нули, для этого в струтуру была добавленна маска. Это позволяет хранить лидирующие нули за 1 байт дополнительной памяти.

Тип данных «вектор» имеет урезанный функционал (только те функции, которые нужны для работы основной программы). Вектор состоит из буфера, в котором хранятся его элементы. Буффер имеет размер Cap, который больше или равен кол-ву элементов Size. Это нужно для уменьшения кол-ва запросов выделения памяти и, соотвественно, уменьшения времени на добавление элемента в вектор.

Идея поразрядной сортировки наудивление проста: для каждого разряда от самого маленького до самого большого нужно выполнить устойчивую сортировку массива по данному разряду . Если использовать устойчивую сотрировку подсчетом, то сложность такого алгоритма будет составлять $\theta(d(n+k))$, где d – кол-во разрядов, n – кол-во элементов массива, k –мощность алфавита сиситмы счисления[2]. Нужно определить, как именно осуществлить поразрядную сортировку для нашего вектора. Так как ключ имеет три поля: день, месяц и год, сортировать нужно сначала по полю «день», потом по полю «месяц», затем – «год». Будем выполнять поразрядную сортировку для каждого поля отдельно. Выбор оптимальной системы счисления завивсит от метрики эффектинвости. В данном случае, так как максимальные значения полей невелики (максимальное значение года равно 9999), будем минимизировать время работы. Если $k < n$, то оптимальным вариантом является указать систему счисления с основанием k. Если $k \geq n$, то оптимально выбрать $\lfloor \lg n \rfloor$. Для ускорения вычиселений, будем использовать системы счисления, которые являются степенью числа 2. Это позволит использовать побитовые операции, которые вычисляются быстрее, чем операции умножения и деления.

Алгоритм устойчивой сортировки подсчетом тоже относительно прост. Как написано в [2]: «Основная идея сортировки подсчетом заключается в том, чтобы для каждого входного элемента x определить кол-во элементов, которые меньше x. С помощью этой информации элемент x можно разместить в той позиции выходного массива, где он должен находиться». Таким образом нам нужно 2 дополнительных массива: массив *count*, где в $count[x]$ будет храниться информация о кол-ве элементов, которые меньше; массив *sorted* – отсортированный массив, который ялвяется результатом работы алгоритма. Сначала нужно подсчитать кол-во повторов ключей в исходном массиве и записать их в *count*, затем можно пройтись по массиву *count* и прибавить к текущему элементу – прошлый (для нулевого элемента делать эту операцию не нужно, или можно считать, что $count[-1] = 0$). Осталось только записать элементы в правильном порядке в массив *sorted*. Для этого нужно пройтись по исходному массиву *input* с конца в начало и поставить i-й элемент на свое место: $sorted[count[input[i]]-1]$, а затем уменьшить $count[input[i]]$ на 1, так как теперь следующий так же элемент должен стоять на предыдущей позиции. Очевидно, что сложность такой сортировки равна $\theta(n+k)$, где n – кол-во элементов, k – максимально возможное значение ключа.

2 Исходный код

Начнем с реализации шаблонного вектора. Потому что без него нельзя ни хранить, ни сортировать элементы. При этом реализуем минимум, который покрывает наши нужды.

```
#pragma once
#include <stddef.h>
template <class T>
class TVector {
public:
  TVector();
  explicit TVector(size_t size);
  explicit TVector(size_t size, size_t cap);
  ~TVector();

  static void Swap(TVector<T> \&a, TVector<T> \&b);

  size_t GetSize();
  void PushBack(T value);

  T* begin();
  T* end();
  T\& operator[] (size_t i);
private:
  T* Buf;
  size_t Cap;
  size_t Size;
};
```

vector.hpp	
TVector()	Конструктор по умолчанию, создает вектор с нулевым размером и вместимостью.
explicit TVector(size_t size)	Создает вектор с размером и вместимостью size.
explicit TVector(size_t size, size_t cap)	Создает конструктор с размером size и вместимостью cap.
~TVector()	Деструктор, освобождает выделенную память.
static void Swap(TVector<T> &T, TVector<T> &b)	Меняет местами два вектора без глубокого компирования.
T* begin()	Возвращает указатель на начало массива.

T* end()	Возвращает указатель на конец массива.
T& operator[] (size_t i)	Перегрузка оператора [], для более естественного обращения к элементу массива. Возвращает ссылку на i-ый элемент.

Реализация вектора проходила в иниркементном режиме: сначала нужно написать тест для еще нереализованной функции, затем саму функцию. Во-первых, это позволяет сразу понять на сколько удобен придуманный интерфейс. Во-вторых, такой подход создает меньше ошибок (большинство сразу же находятся и исправляются). В-третьих, увеличивается концентрация.

После реализации веткора и его проверки. Я начал писать реализацию структуры «ключ-значение». Сразу скажу, что сначала не обратил должного внимания на протокол ввода-вывода. А именно на то, что в ключе могут быть лидирующие нули, поэтому я добавил «маску», которая хранит эту информацию, а так же модифицировал операторы ввода-вывода.

Принцип работы «маски»: Пусть имеется дата DD.MM.YYYY, вместо каждой буквы может быть ноль. Так как маска занимает 1 байт, то можно легко сопоставить номер бита и номер буквы. Если i-й бит маски равен 1, то тогда i-й знак даты равен 0, иначе –i-й знак даты не равен 0. Для облеглечения работы с маской я написал перечисление «PrintMask».

```
1  #pragma once
2  #include <stdint.h>
3  #include <iostream>
4
5  namespace NDate {
6      struct TKey {
7          uint8_t Days;
8          uint8_t Months;
9          uint16_t Years;
10
11         uint8_t PrintMask;
12
13         friend std::istream & operator >> (std::istream &in, TKey &key);
14         friend std::ostream & operator << (std::ostream &out, const TKey &key);
15     };
16
17     const uint8_t MAX_DAYS = 31;
18     const uint8_t MAX_MONTHS = 12;
19     const uint16_t MAX_YEARS = 9999;
20
21     enum PrintMask : uint8_t {
```

```
        FIRST_D_IS_ZERO = 0b10000000,
        FIRST_M_IS_ZERO = 0b00100000,
        THIRD_Y_IS_ZERO = 0b00000010,
        SECOND_Y_IS_ZERO = 0b00000100,
        FIRST_Y_IS_ZERO = 0b00001000,
    };
}

struct TKeyValue {
    NDate::TKey Key;
    uint64_t Value;

    bool operator < (const TKeyValue b) const;

    friend std::istream & operator >> (std::istream &in, TKeyValue &kv);
    friend std::ostream & operator << (std::ostream &out, const TKeyValue &kv);
};

uint8_t GetDays(const TKeyValue &el);
uint8_t GetMonths(const TKeyValue &el);
uint16_t GetYears(const TKeyValue &el);
```

common.hpp	
uint8_t GetDays(const TKeyValue &el)	Возвращает поле Days.
uint8_t GetMonths(const TKeyValue &el)	Возвращает поле Months.
uint16_t GetYears(const TKeyValue &el)	Возвращает поле Years.
TKey	
friend std::istream & operator >> (std::istream &in, TKey &key)	Перегруженный оператор ввода для ключа. Приблежение к стандартному виду.
friend std::ostream & operator << (std::ostream &out, const TKey &key)	Перегруженный оператор вывода для ключа. Приблежение к стандартному виду.
TKeyValue	
friend std::istream & operator >> (std::istream &in, TKeyValue &kv)	Перегруженный оператор ввода.
friend std::ostream & operator << (std::ostream &out, const TKeyValue &kv)	Перегруженный оператор вывода.

После успешной реализации структуры «ключ-занчение» можно приступать к самому интересному – сортировке. Пойдем по порядку. Модифицированная устойчивая сортировка подсчетом «CountSort» принимает 4 аргумента: вектор v, который нужно отсортировать;

функцию getKey, которая возращает целочисленный положительный ключ элемента; основание системы счисления, а точнее его логарифм rank по основанию 2; разряд digit, по которому надо сортировать. Сначала вычисляются:

- base – основание системы счисления;
- mask – маска, с помощью которой можно взять остаток от деления на base;
- shift – смещение, с помощью которого можно выполнить деление.

Таким образом, операция $\frac{element}{base^{digit}}\%base$ заменяется на $(element >> shift)\&mask$. Данная оптимизация дает неплохой прирост к сокрости работы поразрядной сортировки. Далее выполняется вполне стандартная сортировка, а после ее работы выполняется подмена исходного массива на отсортированный.

Сортировка «RadixSort» принимает два аргумента: вектор v, который надо отсортировать и функцию getKey, с помощью которой можно получить ключ. Сначала вычисляется оптимальыая система счисления: функция «calcRank» возвращает степень оптимальной системы счисления; функция digits возвращает количество разрядов в системе счисления 2^{rank} для числа number. Далее выполняется сама сортировка.

Кроме прочего, для удобства, я перегрузил функцию «RadixSort» с одним аргументом NVector::TVector<TKeyValue> &v, что бы было удобнее ее использовать.

```
1  #pragma once
2  #include <stdint.h>
3  #include "common.hpp"
4  #include "vector.hpp"
5
6  namespace NSort {
7      template <class T, class K>
8      void CountSort(NVector::TVector<T> &v, K (*getKey)(const T&), const int rank, const
           int digit) {
9          NVector::TVector<T> sorted(v.GetSize());
10         int base = 1 << rank;
11         int mask = (1 << rank) - 1;
12         int shift = rank * digit;
13         int64_t *count = new int64_t[base];
14
15         for(int64_t i = 0; i < base; ++i) {
16             count[i] = 0;
17         }
18
19         for(int64_t i = 0; i < (int64_t)v.GetSize(); ++i) {
20             count[(getKey(v[i]) >> shift) & mask]++;
21         }
22
```

```
        for(int64_t i = 1; i < base; ++i) {
            count[i] += count[i-1];
        }

        for(int64_t i = v.GetSize() - 1; i >= 0; --i) {
            sorted[count[(getKey(v[i]) >> shift) & mask] - 1] = v[i];
            count[(getKey(v[i]) >> shift) & mask]--;
        }

        NVector::TVector<T>::Swap(v, sorted);

        delete[] count;
    }

    int CalcRank(const int maxValue, const size_t size);
    int CalcMaxDigits(const int rank, int number);

    template<class T, class K>
    void RadixSort(NVector::TVector<T> &v, const int maxValue, K (*getKey)(const T&)) {
        int rank = CalcRank(maxValue, v.GetSize());
        int digits = calcMaxDigits(rank, maxValue);
        for (int digit = 0; digit < digits; ++digit) {
            CountSort(v, getKey, rank, digit);
        }
    }

    void RadixSort(NVector::TVector<TKeyValue> &v);
}
```

```
#include "sort.hpp"

namespace NSort {
    int CalcRank(const int maxValue, size_t size) {
        int rank = 1;
        size_t k = 2;
        while( k < (size_t)maxValue && k < size) {
            k <<= 1;
            rank++;
        }
        return rank;
    }
    int CalcMaxDigits(const int rank, int number) {
        int digits = 0;
        while(number != 0) {
            number >>= rank;
            digits++;
        }
        return digits;
    }

```

```
    void RadixSort(NVector::TVector<TKeyValue> &v)
    {
        NSort::RadixSort(v, NDate::MAX_DAYS, GetDays);
        NSort::RadixSort(v, NDate::MAX_MONTHS, GetMonths);
        NSort::RadixSort(v, NDate::MAX_YEARS, GetYears);
    }
}
```

Далее я привожу листинг файлов main.cpp, benchmark.cpp

```
#include <iostream>
#include "common.hpp"
#include "vector.hpp"
#include "sort.hpp"

int main() {
    std::ios_base::sync_with_stdio(false);
    std::cin.tie(NULL);
    std::cout.tie(NULL);
    NVector::TVector<TKeyValue> v(0);

    TKeyValue kv;
    while (std::cin >> kv) {
        v.PushBack(kv);
    }

    NSort::RadixSort(v);

    for (const auto &el : v) {
        std::cout << el << '\n';
    }
}
```

```
#include <iostream>
#include <algorithm>
#include <chrono>

#include "sort.hpp"
#include "vector.hpp"
#include "common.hpp"


using TDuration = std::chrono::microseconds;
const std::string DURATION_PREFIX = "us";

int main()
{
    NVector::TVector<TKeyValue> input, inputStl;
    TKeyValue kv;
    while (std::cin >> kv)
```

```
18      {
19          input.PushBack(kv);
20          inputStl.PushBack(kv);
21      }
22
23      std::cout << "Count of lines is " << input.GetSize() << std::endl;
24
25      std::chrono::time_point<std::chrono::system_clock> startTs = std::chrono::
            system_clock::now();
26      NSort::RadixSort(input);
27      auto endTs = std::chrono::system_clock::now();
28      uint64_t radixSortTs = std::chrono::duration_cast<TDuration>( endTs - startTs ).
            count();
29
30      startTs = std::chrono::system_clock::now();
31      std::stable_sort(std::begin(inputStl), std::end(inputStl));
32      endTs = std::chrono::system_clock::now();
33
34      uint64_t stl_sort_ts = std::chrono::duration_cast<TDuration>( endTs - startTs ).
            count();
35      std::cout << "Radix sort time: " << radixSortTs << DURATION_PREFIX << std::endl;
36      std::cout << "STL stable sort time: " << stl_sort_ts << DURATION_PREFIX << std::
            endl;
37  }
```

3 Консоль

```
reterer@retcom:~/Desktop/da/lab1$ make -C solution/
make: Entering directory '/home/reterer/Desktop/da/lab1/solution'
g++ -c -Wall -pedantic -std=c++14 main.cpp -o main.o
g++ -c -Wall -pedantic -std=c++14 common.cpp -o common.o
g++ -c -Wall -pedantic -std=c++14 sort.cpp -o sort.o
g++ main.o common.o sort.o -o solution
g++ -c -Wall -pedantic -std=c++14 benchmark.cpp -o benchmark.o
g++ benchmark.o common.o sort.o -o benchmark
make: Leaving directory '/home/reterer/Desktop/da/lab1/solution'
reterer@retcom:~/Desktop/da/lab1$ cat manualtests/test01.t
1.2.1 3
1.1.1 1
13.05.2001 7
04.06.0059 5
01.1.1 2
2.2.1 4
04.06.0059 6
reterer@retcom:~/Desktop/da/lab1$ solution/solution <manualtests/test01.t
1.1.1 1
01.1.1 2
1.2.1 3
2.2.1 4
04.06.0059 5
04.06.0059 6
13.05.2001 7
```

4 Тест производительности

Тест производительности представляет из себя замер времени работы моей реализации поразрядной сортировки, а так же замер устойчивой сортировки из стандартной библиотеки языка C++ на векторах разной длины: 10, 100, 1000, 10000, 1000000. При этом, замер будем проводить 3 раза на разных исходных данных (приведу сразу усредненные, что бы не увеличивать размер отчета). После чего возьмем среднее. Программа будет запускаться с приоритетом -20, что бы минимизировать влияние других факторов.

```
reterer@retcom:~/Desktop/da/lab1$ ./wrapper.sh
[info] Running tests/01.t
Count of lines is 10
Radix sort time: 6us
STL stable sort time: 3us

[info] Running tests/02.t
Count of lines is 100
Radix sort time: 23us
STL stable sort time: 19us

[info] Running tests/03.t
Count of lines is 1000
Radix sort time: 189us
STL stable sort time: 222us

[info] Running tests/04.t
Count of lines is 10000
Radix sort time: 1528us
STL stable sort time: 2803us

[info] Running tests/05.t
Count of lines is 100000
Radix sort time: 15027us
STL stable sort time: 36321us

[info] Running tests/06.t
Count of lines is 1000000
Radix sort time: 153406us
STL stable sort time: 451049us
```

Можно сделать вывод, что моя реализация поразрядной сортировки, начиная с $n \geq$

1000 начинает обгонять стандартную сортировку. Однако, благодаря выбору оптимальной системы счисления, поразрядная сортировка несильно отстаёт от стандартной и на маленьких n.

5 Выводы

Выполнив первую лабораторную работу по курсу «Дискретный анализ», я могу сделать следующие выводы:

- Я глубже узнал об поразрядной сортировке. Использовать ее можно тогда, когда можно представить ключ в виде целого неотрицательного числа с сохранением отношения «<». Имеет ли смысл её использовать? Я вижу её применение, если данных достаточно много, а так же важна скорость сортировки. Но стоит помнить, что она требует $O(n + k)$ памяти, что может быть непозволительной роскошью, особенно, когда у вас очень много данных. Кроме этого, она обладает большой константной, что может съесть все преимущества;
- Перед началом выполнения лабораторной работы нужно очень внимательно изучить задание, а так же проанализировать образец ввода-вывода программы;
- Использование шаблонов может уменьшить повторяемость кода. К тому же, шаблоны работают так же быстро как и обычные функции и классы. Но в них достаточно просто допустить ошибку, а узнать о ней получиться только в конце процесса компиляции;
- Нужно учиться писать код быстро и компактно;
- В C++ нет аналога realloc.

Возможно, было бы целеобразнее использовать в качестве ключа кол-во дней, его можно хранить в int32. В таком случае поразрядная сортировка могла бы оперировать большей системой счисления. С другой стороны еще один атрибут может значительно увеличить размер класса. Или, вместо кол-ва дней, месяцев, лет хранить число Days + (Moths + Years*12)*31, тогда возможно достаточно просто восстановить исходные поля. А в памяти это будет занимать столько же.

Список литературы

[1] *Перегрузка операторов ввода и вывода в C++ | Уроки C++ - Ravesli*
URL: `https://ravesli.com/urok-133-peregruzka-operatorov-vvoda-i-vyvoda/`
(дата обращения: 7.10.2020).

[2] 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.))

[3] Список использованных источников оформлять нужно по ГОСТ Р 7.05-2008