Московский авиационный институт
(национальный исследовательский университет)

Факультет компьютерных наук и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №1 по курсу «Дискретный анализ»

Студент: О. А. Мезенин
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б-21
Дата:
Оценка:
Подпись:

Москва, 2023

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

Основная идея поразрядной сортировки заключается в сортировке чисел устойчивой сортировкой по каждому разряду, начиная с младшего.

Воспользуемся следующей леммой [1]: «Пусть имеется n b-битовых чисел и натуральное число $r \leq b$. Алгоритм позволяет выполнить корректную сортировку этих чисел за время $\Theta((b/r)(n + 2^r))$».

Переведём шестнадцатиричные числа в десятичные. Для этого нам понадобится структура на $b = 32 * 4 = 128$ бит. Затем возьмём битовую маску размера r и, сдвигая её влево, будем сортировать числа, полученные наложением маски. В качестве устойчивой сортировки выберем сортировку подсчётом.

2 Исходный код

На каждой непустой строке входного файла располагается пара «ключ-значение», поэтому создадим новую структуру `TMD5String`, в которой будем хранить ключ в виде числа типа `unsigned __int128` и значение как статический массив char'ов.

```
const unsigned int HEX_BYTES_NUMBER = 32;
const unsigned int VALUE_BYTES_NUMBER = 64;
__extension__ using Tuint128 = unsigned __int128;

class TMD5String {
private:
    Tuint128 HexToInt(char c);
    char IntToHex(Tuint128 c);
    Tuint128 HexToInt128(std::string& st);
    void Int128ToHex(Tuint128 number, char st[HEX_BYTES_NUMBER]);
    Tuint128 key;
    char value[VALUE_BYTES_NUMBER];
public:
    TMD5String(std::string& key, std::string& value);
    Tuint128 GetIntKey();
    std::string GetHexKey();
    std::string GetValue();
};
```

kv_struct.cpp	
Tuint128 HexToInt(char c)	Функция, переводящая шестнадцатиричную цифру в десятичную
char IntToHex(Tuint128 c)	Функция, переводящая десятичную цифру в шестнадцатиричную
Tuint128 HexToInt128(std::string& st)	Функция, переводящая шестнадцатиричное число в десятичное
void Int128ToHex(Tuint128 number, char st[HEX_BYTES_NUMBER])	Функция, переводящая десятичное число `number` в шестнадцатиричное `st`
parser.cpp	
void Parser(TVector<TMD5String*>& elems)	Функция, читающая из стандартного ввода строки и заполняя вектор *elems*

В файле **sort.cpp** реализованы поразрядная и сортировка подсчётом.
Поразрядная генерирует маску размером `MASK_SIZE`, затем создаёт временный вектор `tmp` и запускает цикл от `0` до `KEY_BITS / MASK_SIZE`, где `KEY_BITS` в нашем случае равняется `128`. На каждой чётной итерации запускается сортировка подсчётом исходного вектора `elems`, а результат кладётся в временный вектор `tmp`. На нечётной итерации векторы меняются местами.

```
void RadixSort(TVector<TMD5String*> &elems) {
    Tuint128 mask = std::pow(2, MASK_SIZE)-1;

    TVector<TMD5String*> tmp(elems.Size(), nullptr);

    for (unsigned short i = 0; i < KEY_BITS / MASK_SIZE; ++i) {
        if ((i & 1) == 0) {
            CountingSort(elems, tmp, mask, i);
        } else {
            CountingSort(tmp, elems, mask, i);
        }
    }
}
```

Сортировка подсчётом помимо векторов принимает маску `mask` и количество сдвигов `shiftCount`. Тогда числа будут сортироваться по следующему ключу: $(N_i$ `» (shiftCount * MASK_SIZE)) & mask`.

```
const unsigned int KEY_BITS = 128;
const unsigned int MASK_SIZE = 16;

void CountingSort(TVector<TMD5String*> &elems, TVector<TMD5String*> &result, Tuint128
    mask, size_t shiftCount) {
    if (elems.Empty()) {
        return;
    }
    TVector<size_t> tmp(mask+1, 0);

    for (unsigned int i = 0; i < elems.Size(); ++i) {
        Tuint128 sortKey = elems[i]->GetIntKey() >> (shiftCount * MASK_SIZE);
        sortKey = sortKey & mask;
        ++tmp[sortKey];
    }

    for (unsigned int i = 1; i < tmp.Size(); ++i) {
        tmp[i] += tmp[i-1];
    }

    for (int i = elems.Size() - 1; i >= 0; --i) {
        Tuint128 sortKey = elems[i]->GetIntKey() >> (shiftCount * MASK_SIZE);
        sortKey = sortKey & mask;
        size_t pos = tmp[sortKey]--;
        result[pos - 1] = elems[i];
    }
}
```

3 Консоль

```
aprold@SAI:~/Documents/GitHub/MAI-DA/lab1$ make all
g++    -c -o kv_struct.o kv_struct.cpp
g++    -c -o sort.o sort.cpp
g++    -c -o parser.o parser.cpp
g++ -std=c++2a -Wpedantic -Wall -Wextra -Wno-unused-variable kv_struct.o parser.o
sort.o vector.hpp lab1.cpp -o lab1
aprold@SAI:~/Documents/GitHub/MAI-DA/lab1$ cat test
d131ed4e7a4eb8be3d3baa2a8e7aeea3 pnmvwppzzybnsbfjlohzjynnzdpjvguhqywhulzsxbtwsogfakm
6d8cf67a3f244179d3f3723a112db572 rnkwngty
ca44d5408b6eff8b0eadeb96f4b31af3 mvvbnegarvxbgyqqrvajagjowxitspaattjuweprijmimvt
0affe8eaa52d8ee7442e09f9da313b6d efsbxtcrjlbfqczvbbsfjtapkhxssgjnmkxsvcpuindndfgoccv
d8abc4a628e884eb00b95cfd1af3dd8c ky
3876a892bd74981e3ec26713f171c57c jvoccvfuzsowkducxozvkvavgpondhffyefmuzibrhmcb
54c56e6618268dda36cd58b20e31188e fz
3e5542b7ea19d7555fbc9f67807d4d88 agiqrnjfnkgcebvpwpnclkaojemhakmczfjgovhefzsiutc
ea24e0dc2f8338a55a605d70fb02f0a4 medczzlhkyzhcdkoecfitooqrrdsx
a508db2370969d71e6ca7fe9c9bca0be deikamkqrihfkedxfhvfzsxchuxvneuhdvpqjixgtrwvrssbenv
aprold@SAI:~/Documents/GitHub/MAI-DA/lab1$ ./lab1 <test
0affe8eaa52d8ee7442e09f9da313b6d efsbxtcrjlbfqczvbbsfjtapkhxssgjnmkxsvcpuindndfgoccv
3876a892bd74981e3ec26713f171c57c jvoccvfuzsowkducxozvkvavgpondhffyefmuzibrhmcb
3e5542b7ea19d7555fbc9f67807d4d88 agiqrnjfnkgcebvpwpnclkaojemhakmczfjgovhefzsiutc
54c56e6618268dda36cd58b20e31188e fz
6d8cf67a3f244179d3f3723a112db572 rnkwngty
a508db2370969d71e6ca7fe9c9bca0be deikamkqrihfkedxfhvfzsxchuxvneuhdvpqjixgtrwvrssbenv
ca44d5408b6eff8b0eadeb96f4b31af3 mvvbnegarvxbgyqqrvajagjowxitspaattjuweprijmimvt
d131ed4e7a4eb8be3d3baa2a8e7aeea3 pnmvwppzzybnsbfjlohzjynnzdpjvguhqywhulzsxbtwsogfakm
d8abc4a628e884eb00b95cfd1af3dd8c ky
ea24e0dc2f8338a55a605d70fb02f0a4 medczzlhkyzhcdkoecfitooqrrdsx
```

4 Тест производительности

Тесты производительности представляют из себя следующее: поразрядная сортировка сравнивается со стабильной сортировкой из STL. Время чтения элементов и создания векторов не учитывается. Тестов будет три: на 10^5, 10^6 и 10^7 элементов.

```
aprold@SAI:~/Documents/GitHub/MAI-DA/lab1$ ./benchmark <./tests/07.t
Count of lines is 100000
Radix sort time: 132545us
STL stable sort time: 36487us
aprold@SAI:~/Documents/GitHub/MAI-DA/lab1$ ./benchmark <./tests/08.t
Count of lines is 1000000
Radix sort time: 2035777us
STL stable sort time: 1096319us
aprold@SAI:~/Documents/GitHub/MAI-DA/lab1$ ./benchmark <./tests/09.t
Count of lines is 10000000
Radix sort time: 22637489us
STL stable sort time: 13895116us
```

Как видно, на всех тестах STL-сортировка выигрывает, но при увеличении числа элементов разница во времени уменьшается и поразрядная догоняет STL-сортировку.
Stable sort работает за $\Theta(n * log(n))$ (где n – количество элементов), если доступна дополнительная память [2]. Поразрядная сортировка работает, как было оговорено ранее, за $\Theta((b/r)(n + 2^r))$.
«Если $b = O(\log_2 n)$, как это часто бывает, и мы выбираем $r \approx \log_2 n$, то время работы алгоритма поразрядной сортировки равно $\Theta(n)$, что выглядит предпочтительнее среднего времени выполнения быстрой сортировки $\Theta(n \log_2 n)$. Однако в этих выражениях, записанных в Θ-обозначениях, разные постоянные множители. Несмотря на то, что для поразрядной сортировки n ключей может понадобиться меньше проходов, чем для их быстрой сортировки, каждый проход при поразрядной сортировке может длиться существенно дольше».[1]
Можно сделать вывод, что константа у поразрядной сортировки довольно большая, но существует число n_0 такое, что все тесты, в которых количество элементов больше, чем n_0, будут сортироваться быстрее поразрядной сортировкой, чем STL-сортировкой.

5 Выводы

Выполнив первую лабораторную работу по курсу «Дискретный анализ», узнал об устойчивом алгоритме сортировки подсчётом, который умеет сортировать структуры по ключу. Узнал, как работает поразрядная сортировка и шаблоны (templates) в C++. Приобрёл практические навыки в написании сортировок; вектора, используя шаблоны; генератора тестов и бенчмарка.
Хочу отметить особую важность в генерации тестов и использовании бенчмарков. В целом тестирование помогает выявить ошибки или плохую оптимизацию программы до её релиза. В моём случае это помогло сохранить попытки, которых было хоть и много, но которые были всё же ограничены. При разработке реального продукта всё куда серьёзнее, и без тщательного тестирования не обойтись.

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.))

[2] *std::stable_sort – cppreference*
URL: `https://en.cppreference.com/w/cpp/algorithm/stable_sort` (дата обращения: 22.03.2023).