Московский авиационный институт (национальный исследовательский университет)

Институт информационных технологий и прикладной математики

«Кафедра вычислительной математики и программирования»

Лабораторная работа по предмету "Дискретный анализ" №1

Студент: Шипилов К. Ю.

Преподаватель: Макаров Н. К.

Группа: М8О-203Б-22

Дата: 18.03.2024

Оценка:

Подпись:

Москва 2024 г.

Оглавление

Цель работы

Приобретение практических навыков в использовании алгоритмов сортировки за линейное время

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 задания определяется типом ключа (и соответствующим ему методом сортировки) и типом значения:
Сортировка подсчётом.
Тип ключа: числа от 0 до 65535.
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.

Формат ввода:

На каждой непустой строке входного файла располагается пара «ключ-значение», в которой ключ указан согласно заданию, затем следует знак табуляции и указано соответствующее значение.

Формат вывода:

Выходные данные состоят из тех же строк, что и входные, за исключением пустых и порядка следования.

Общие сведения о программе

Программа представлена файлом – main.cpp.

Для хранения пар «ключ-значение» в программе используется структура **`Pair`**

со следующими полями:

`key` – ключ элемента;

`value` – значение элемента (строка фиксированной длины).

Также для структуры реализован конструктор по умолчанию.

Для хранения массива пар используется класс **`Container`** со следующими полями:

`_size` – количество элементов в массиве;

`_capacity` – максимальное количество элементов, которые могут храниться в массиве без перевыделения памяти;

`_array` – указатель на непрерывную область памяти, в которой хранятся пары.

Методы реализованные в классе **`Container`**:

`Container()` – конструктор по умолчанию;

`Container(const size_t& size)` – конструктор, который создает массив с заданным количеством пар, при этом всем парам присваивается значение по умолчанию;

`Container(const Container& other)` – конструктор копирования;

`Container(Container&& other)` – конструктор перемещения;

`~Container() noexcept` – деструктор;

`void pushBack(const Pair& pair)` – добавление пары в конец массива;

`size_t size() const noexcept` – возвращение размера массива;

`Container& operator=(const Container& other)` – оператор присваивания копирования;

`Container& operator=(Container&& other)` – оператор присваивания перемещения;

`Pair& operator[](const size_t index)` – оператор для обращения к элементу массива по индексу;

`const Pair& operator[](const size_t index) const` – оператор для просмотра элемента массива по индексу;

`void print(std::ostream& ostream) const noexcept` – печать всех элементов массива в поток вывода.

Функция сортировки:

```
void countingSort(const Container& array,
                  container& sortedArray, unsigned int maxKey)
```

Общий алгоритм решения

Функция сортировки получает ссылку на исходный массив; ссылку на массив, в котором будут хранится данные после сортировки; максимальное значение ключа для всех пар в массиве с исходными данными.

Создается массив для подсчета пар, заполненный нулями, в котором индекс каждого элемента соответствует ключу пары. Необходимо пройтись по всем элементам исходного массива и для каждой пары увеличить на 1 значение элемента массива для подсчета по индексу, соответствующему ключу пары. Таким образом, массив для подсчета содержит в каждом элементе количество пар с ключом, равным индексу текущего элемента.

После этого необходимо к каждому элементу массива для подсчета, начиная с первого, прибавить значение предыдущего элемента. После этого каждый элемент массива для подсчета содержит количество элементов с ключами меньшими или равными индексу текущего элемента.

Если вычесть из элемента массива для подсчета 1, то если пара с соответствующим ключом существует, получим индекс последней (правой) пары с соответствующим ключом в отсортированном массиве. Необходимо пройтись по каждой паре в исходном массиве и скопировать ее в массив с результатом сортировки по этому индексу, при этом при каждом копировании соответствующий элемент массива для подсчета должен уменьшаться на 1. Для того чтобы сортировка была устойчивой, обход исходного массива необходимо осуществлять в обратном порядке.

Реализация

main.cpp

```
#include <string.h>

#include <iostream>

struct Pair {
public:
unsigned int key;
char value[65];

Pair() : key(0), value("") {}
};

class Container {
private:
size_t _size;
size_t _capacity;
Pair* _array;

public:
Container() noexcept : _size(0), _capacity(0), _array(nullptr) {}

Container(const size_t& size) : _size(size), _capacity(size) {
_array = new Pair[_capacity];
}

Container(const Container& other)
: _size(other._size), _capacity(other._capacity) {
_array = new Pair[_capacity];
for (size_t i{0}; i < _size; ++i) {
_array[i] = other._array[i];
}
}

Container(Container&& other)
: _size(other._size), _capacity(other._capacity) {
_array = other._array;
other._size = 0;
other._capacity = 0;
other._array = nullptr;
```

```
}

~Container() noexcept { delete[] _array; }

void reallocate(const size_t& newCapacity) {
Pair* newArray = new Pair[newCapacity];
for (size_t i{0}; i < _size; ++i) {
newArray[i] = _array[i];
}
delete[] _array;
_array = newArray;
_capacity = newCapacity;
}

void pushBack(const Pair& pair) {
if (_size >= _capacity) {
reallocate((_size + 1) * 2);
}
_array[_size++] = pair;
}

size_t size() const noexcept { return _size; }

Container& operator=(const Container& other) {
_size = other._size;
_capacity = other._capacity;
delete[] _array;
_array = new Pair[_capacity];
for (size_t i{0}; i < _size; ++i) {
_array[i] = other._array[i];
}
return *this;
}

Container& operator=(Container&& other) {
_size = other._size;
_capacity = other._capacity;
_array = other._array;
return *this;
}

Pair& operator[](const size_t& index) { return _array[index]; }

const Pair& operator[](const size_t& index) const { return _array[index]; }
```

```
void print(std::ostream& ostream) const noexcept {
for (size_t i{0}; i < _size; ++i) {
ostream << _array[i].key << "\t" << _array[i].value << std::endl;
}
}
};

std::ostream& operator<<(std::ostream& ostream, const Container& container) {
container.print(ostream);
return ostream;
}

void countingSort(const Container& array, Container& sortedArray,
unsigned int maxKey) {
unsigned int countingArray[maxKey + 1];
for (size_t i = 0; i <= maxKey; ++i) {
countingArray[i] = 0;
}
for (size_t i = 0; i < array.size(); ++i) {
++countingArray[array[i].key];
}
for (size_t i = 1; i <= maxKey; ++i) {
countingArray[i] += countingArray[i - 1];
}
for (int i = array.size() - 1; i >= 0; --i) {
sortedArray[countingArray[array[i].key] - 1] = array[i];
--countingArray[array[i].key];
}
}

int main() {
Container array;
unsigned int maxKey{0};
unsigned int key;
char value[65];
while (std::cin >> key) {
std::cin >> value;
Pair pair;
pair.key = key;
strcpy(pair.value, value);
array.pushBack(pair);
if (key > maxKey) maxKey = key;
}
```

```
Container sortedArray(array.size());
countingSort(array, sortedArray, maxKey);
std::cout << sortedArray;
}
```

Пример работы

Test 1

Input	Output
0 n399tann9nnt3ttnaaan9nann93na9t3a3t9999na3aan9antt3tn93aat3naatt 65535 n399tann9nnt3ttnaaan9nann93na9t3a3t9999na3aan9antt3tn93aat3naat 0 n399tann9nnt3ttnaaan9nann93na9t3a3t9999na3aan9antt3tn93aat3naa 65535 n399tann9nnt3ttnaaan9nann93na9t3a3t9999na3aan9antt3tn93aat3na	0 n399tann9nnt3ttnaaan9nann93na9t3a3t9999na3aan9antt3tn93aat3naatt 0 n399tann9nnt3ttnaaan9nann93na9t3a3t9999na3aan9antt3tn93aat3naa 65535 n399tann9nnt3ttnaaan9nann93na9t3a3t9999na3aan9antt3tn93aat3naat 65535 n399tann9nnt3ttnaaan9nann93na9t3a3t9999na3aan9antt3tn93aat3na

Test 2

Input	Output
1 first 64808 first 54533 first 54533 second 52596 first 1 second	1 first 1 second 1 third 52596 first 52596 second 52596 third

64808 second 1 third 52596 second 54533 third 52596 third 64808 third	54533 first 54533 second 54533 third 64808 first 64808 second 64808 third

Test 3

Input	Output
27965 BH1uU_@TKiMnFOSFFut=@F4rJgnAqyx886xANQF_9b*CdeMeTDtyXtJ48NDnAb1$ 35988 E&-ZP6t^13_AjXqxyawzLXHLN*bT@X-Edau7%JZCmwyKQ9CX@myvLA$i! 49596 bg_8#%qJ!JfHdvvQ 0 Wc@kFmn5SeYruw5asa2Mc0k1^ApNy4UFneh$NoB--*r7S^^7AKiQXW03Y$dpl 35058 x8B0XR852P@*NALtsCCsM$Y9g9FOgHKj1s%zCjigF_Sca!41nSfYV 43181 R&mDxy3Mi8+=D %Wdhkb1osq8jaF^y1bF-l+x 60853 - KOx_n6D7zzkPBDGM0@iCx8MaOqABl9*V&Jtkd+NT0XYaQ 713 t 54533 fN+FVCCrHJ97T+P+u4D76 64808 qiK&r$ERvUO! g+AGv48&V0feBpu$nJDRbk6lfDKHax2R0J+zFwWI9EI7ikyL4rCm 52596 &cfG12Bu83lCXM9_kk16HmbBfekSqEikQIsf-	0 Wc@kFmn5SeYruw5asa2Mc0k1^ApNy4UFneh$NoB--*r7S^^7AKiQXW03Y$dpl 713 t 27965 BH1uU_@TKiMnFOSFFut=@F4rJgnAqyx886xANQF_9b*CdeMeTDtyXtJ48NDnAb1$ 35058 x8B0XR852P@*NALtsCCsM$Y9g9FOgHKj1s%zCjigF_Sca!41nSfYV 35988 E&-ZP6t^13_AjXqxyawzLXHLN*bT@X-Edau7%JZCmwyKQ9CX@myvLA$i! 43181 R&mDxy3Mi8+=D %Wdhkb1osq8jaF^y1bF-l+x 49596 bg_8#%qJ!JfHdvvQ 52596 &cfG12Bu83lCXM9_kk16HmbBfekSqEikQIsf-XNwUlFl%D 54533 fN+FVCCrHJ97T+P+u4D76 54533 F@NqDPbmcpQ5AB*WNdA5_J^KvaTBJS@*KKWrIIyIXebSUt2!jbRSXXBp 60853 - KOx_n6D7zzkPBDGM0@iCx8MaOqABl9*V&Jtk

XNwUlFl%D 54533 F@NqDPbmcpQ5AB*WNdA5_J^KvaTBJS@*KK WrIIyIXebSUt2!jbRSXXBp	d+NT0XYaQ 64808 qiK&r$ERvUO! g+AGv48&V0feBpu$nJDRbk6lfDKHax2R0J+zFw WI9EI7ikyL4rCm

Сравнительная таблица

Количество пар	Максимальный ключ	Время работы сортировки подсчетом, с	Время работы сортировки пузырьком, с	Время работы std::sort, с
1000000	5	0,510	>600	0,783
20	400000	0,004	0,002	0,002

Вывод

Выполняя лабораторную работу я изучил алгоритмы сортировки за линейное время и на практике применил алгоритм сортировки подсчетом. Такие сортировки сортировки имеют преимущество перед сортировками с асимптотической сложностью O(n*log(n)) и O(n^2) при работе с большим объемом данных. При этом их сложность зависит не только от количества входных данных, но и от ключей для сортировки, поэтому существуют такие наборы данных, для которых сортировки за O(n*log(n)) будут работать быстрее. Например, если максимальный ключ элемента значительно больше, чем количество элементов в наборе входных данных, сортировка подсчетом может оказаться медленнее, чем быстрая или даже пузырьковая сортировка.