Лабораторная работа № 1 по курсу дискретного анализа: сортировка за линейное время

Выполнил студент группы М8О-208Б-20 Морозов Артем Борисович.

Условие

Кратко описывается задача:

1. Требуется написать программу на языке C++, которая сортирует входные данные, подающиеся пользователю парой “ключ-значение”.
2. Вариант задания: 7-3. Поразрядная сортировка. Ключами служат автомобильные номера вида A 373 BC (используются только буквы латинского алфавита), значением же является число от 0 до 2^64 – 1. Ключ и значение при вводе разделены знаком табуляции.

Метод решения

Метод решения достаточно прост и понятен: в самом начале работы программы в основной функции создается std::vector, в который считываются пары ключ-значение. Ключ у меня хранится в виде std::string, а значение в виде числа типа unsigned long long. Потом, если этот вектор не пуст, то вызывается функция поразрядной сортировки, в которую передается уже целиком заполненный вектор данных. Поразрядная сортировка использует в себе сортировку подсчетом: так как в автомобильном номере всего 8 символов, то в качестве ключей у элементов заполненного вектора хранятся строки, в которых нулевой индекс – начало нашего автомобильного номера (первая буква), а седьмой индекс – конец нашего автомобильного номера (последняя буква). Следовательно, мы в цикле от 7 до 0 запускаем сортировку подсчетом, которая, соответственно, будет выдавать разные версии нашего отсортированного массива после каждой итерации. На последней итерации функция заканчивает сортировку и выдает нам правильно отсортированный массив. Далее в цикле мы выводим у каждого элемента массива два его поля: ключ и значение.

Описание программы

Для выполнения данной лабораторной работы я создал единственный файл main.cpp, в котором сначала реализовал собственную структуру под именем Map, хранящую ключ и значение:

```
struct Map {
   std:: string key;
   unsigned long long value;
};
```

Далее я реализовал непосредственно саму void-функцию сортировки наших входных данных: я назвал ее RadixSort, а принимает она вектор из указателей на объекты типа Map, причем по ссылке, так как нам нужно менять тот вектор, который мы принимаем:

```
void RadixSort (std:: vector<Map*>& inputed)
```

В самой функции я предварительно создаю массив chars, рассчитанный на 26 элементов (в программе это значение обозначено как `const short int AMOUNT_OF_LETTERS = 26` в соответствии с правилами оформления лабораторных работ по дискретному анализу). Этот массив нам нужен для того, чтобы действовать согласно алгоритму: считать количество вхождений того или иного элемента в поле key наших объектов, а затем считать сумму i-того и i+1-того элементов.

```
for (int x = 0; x < inputed.size(); ++x) {
    ++chars[inputed[x]->key[i] - '0'];
}
for (short int t = 1; t < AMOUNT_OF_DIGITS; ++t) {
            chars[t] += chars[t - 1];
}
```

Далее мы, проходясь с последнего элемента вектора до самого начала, встречая какой-либо элемент, декрементируем соответствующее этому элементу значение в массиве chars, а в специально созданный для отсортированных данных вектор по этому индексу записываем текущий элемент изначального вектора:

```
for (int x = inputed.size(); x != 0; --x) {
          --chars[inputed[x - 1]->key[i] - '0'];
          sorted[chars[inputed[x - 1]->key[i] - '0']] = inputed[x - 1];
}
```

После каждой итерации мы зануляем все элементы массива chars и меняем местами значения в исходном векторе и отсортированном. Таким образом, на последней итерации у нас функция std::swap сделает изначальный вектор уже отсортированным:

```
for (short int p = 0; p < AMOUNT_OF_LETTERS; ++p) {
          chars[p] = 0;
}
std:: swap(inputed, sorted)
```

В главной же функции int main() просто создается вектор, в цикле до EOF считываются значения, динамически выделяется память для каждого объекта типа Map*, и адрес каждого элемента добавляется в вектор. Потом, если вектор

не пустой, мы вызываем функцию сортировки, а после выводим все элементы уже отсортированного вектора и освобождаем динамически выделенную память:

```
 std:: vector<Map*> inputed;
std:: string first_string, second_string, third_string;
unsigned long long value;
while (std:: cin >> first_string >> second_string >> third_string >> value) {
   Map* data = new Map();
   data->key = first_string + " " + second_string + " " + third_string;
   data->value = value;
   inputed.push_back(data);
}
if (inputed.size() != 0) {
   RadixSort(inputed);
}
for (int i = 0; i < inputed.size(); ++i) {
   std:: cout << inputed[i]->key << "\t" << inputed[i]->value << "\n";
}
for (int i = 0; i < inputed.size(); ++i) {
   delete inputed[i];
}
```

На этом программа завершается.

Дневник отладки

По мере отправки лабораторной работы на автоматическую систему проверки программа практически всегда выдавала верный ответ (неверным он был из-за забытого знака табуляции), однако далеко не всегда проходила по скорости и по памяти. Для увеличения производительности программы были добавлены строчки std::ios::sync_with_stdio(false), std::cin.tie(0), std::cout.tie(0). Первая строчка отключает синхронизацию потока ввода-вывода C++ и C. Следующие две строчки гарантируют, что ввод и вывод будут работать быстрее. Был удален второй массив для подсчитывания цифр, и теперь программа прекрасно справляется с одним массивом для подсчета, что дает выигрыш как по памяти, так и по времени. Аналогично и по памяти, и по времени преимущество дает создание в int main() не вектора из объектов, а вектора из указателей на объекты, так как копировать указатели быстрее (улучшение по скорости) из-за того, что они меньше весят (улучшение по памяти).

Тест производительности

Давайте исследуем производительность нашего алгоритма на нескольких входных данных n:

1) При n = 1000 время составляет ~ 0,056 секунд.
2) При n = 10000 время составляет ~ 0,067 секунд.
3) При n = 100000 время составляет ~ 0,173 секунды.

4) При n = 1000000 время составляет ~ 2,1 секунды.
5) При n = 10000000 время составляет ~ 37 секунд.

Недочёты

Программа работает довольно быстро, не затратно по памяти и, что самое главное, правильно. Все недочеты были исправлены, о них упоминалось в дневнике отладки. Единственное “но”, которое можно добавить – программа хорошо работает для корректных входных данных. Для некорректных же входных данных работа данной программы не предусмотрена, так как делалась исключительно под правильные тесты и в учебных целях

Выводы

Данный алгоритм отлично работает для сортировки относительно небольшого количества входных данных, однако для большого количества больших данных, как мы видели на тестах производительности, будет не эффективен, ведь его сложность в общем случае – $O(d*n)$, где d – количество разрядов, по которым происходит сортировка, n – объем входных данных. В моем же случае количество разрядов равно шести, поэтому в итоге сложность получается O(n).