

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

Выполнил студент группы М80-208Б-22 МАИ *Кочкожаров Иван*.

Условие

Краткое описание задачи:

1. Требуется разработать программу, осуществляющую ввод пар "ключ-значение" сортировку по возрастанию ключа указанным алгоритмом сортировки за линейное время и вывод получившейся последовательности.
2. Вариант задания:
 - Карманная сортировка.
 - Тип ключа: вещественные числа от -100 до 100.
 - 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.

Метод решения

Неизвестно, сколько конкретно пар "ключ-значение" будет подано в программу, поэтому необходимо реализовать динамический массив, он же вектор, способный менять свой размер для хранения данных.

Далее требуется реализовать стабильную карманную сортировку. Ее суть заключается в том, что определяется функция, которая ставит в соответствие элементам массива номер кармана, в который он будет помещен. элементы все элементы в i -ом кармане должны быть меньше, чем все элементы в $i+1$ -ом кармане. Затем каждый карман сортируется стабильной сравнительной сортировкой (выбрана сортировка слиянием) и полученные карманы конкатенируются. Если входные элементы подчиняются равномерному закону распределения, то сортировка работает за $O(n)$.

Описание программы

Разделение по файлам, описание основных типов данных и функций.

- Класс TVector. Реализованы конструкторы, основные методы, также перегружены некоторые операторы вроде индексатора.

```

template <class T>
class TVector {
    private:
        T* data{nullptr};
        std::size_t capacity{0}, size{0};
        static constexpr std::size_t NewCapacity(std::size_t capacity);
    public:
        void Reserve(std::size_t newCapacity);
        void PushBack(const T& value);
        void PushBack(T&& value);
        [[nodiscard]] constexpr std::size_t Size() const { return size; }
        [[nodiscard]] constexpr std::size_t Capacity() const { return size }
        TVector() = default;
        TVector(std::size_t size) : data{new T[size]}, capacity{size} {}
        TVector(const TVector& other);
        TVector(TVector&& other) noexcept;
        TVector& operator=(const TVector& other);
        TVector& operator=(TVector&& other) noexcept;
        T operator[](std::size_t idx) const { return data[idx]; }
        T& operator[](std::size_t idx) { return data[idx]; }
        virtual ~TVector() noexcept { delete[] data; }
};

```

- Структура TKeyValuePair. Необходима для хранения пар "ключ-значение".

```

struct TFixedSizeLine {
    inline static const int SIZE = 64;
    char str[SIZE + 1];
    TFixedSizeLine() = default;
    TFixedSizeLine(const char* a) { strcpy(str, a); }
    operator const char*() { return str; }
};

class TKeyValuePair {
public:
    double key;
    TFixedSizeLine* value;
    TKeyValuePair();
    TKeyValuePair(int key, const char* value);
    TKeyValuePair(const TKeyValuePair& other);
    TKeyValuePair& operator=(const TKeyValuePair& other);
    TKeyValuePair& operator=(TKeyValuePair&& other);
    TKeyValuePair(TKeyValuePair&& other);
};

```

```

~TKeyValuePair();
void Print(FILE* stream);
bool Scan(FILE* stream);
}

```

- Реализация карманной сортировки

```

void BucketSort(TVector<TKeyValuePair>& arr) {
    const int minElement = -100;
    const int maxElement = 100;
    const int range = maxElement - minElement;
    const size_t numBuckets = range;
    TVector<TVector<TKeyValuePair>> buckets(numBuckets);
    for (size_t i = 0; i < arr.Size(); ++i) {
        int bucketIndex = (arr[i].key - minElement) * (numBuckets - 1);
        buckets[bucketIndex].PushBack(std::move(arr[i]));
    }

    size_t cnt = 0;
    TVector<TKeyValuePair> buf(arr.Size());
    for (size_t i = 0; i < numBuckets; ++i) {
        MergeSort(buckets[i], buf);
        for (size_t j = 0; j < buckets[i].Size(); ++j) {
            arr[cnt++] = std::move(buckets[i][j]);
        }
    }
}

```

Дневник отладки

Изначально в качестве вспомогательного алгоритма использовалась сортировка вставками, но только после замены ее на сортировку слиянием решения стало проходить чекер. Так же сильно улучшило производительность хранение строки из 64 символов не в самом элементе вектора, а в куче.

Тест производительности

Померить время работы кода лабораторной и теста производительности на разных объемах входных данных. Сравнить результаты. Проверить, что рост времени работы при увеличении объема входных данных согласуется с заявленной сложностью.

Карманная сортировка работает за линейное время. Для большей наглядности приведём таблицу, в которой написанная сортировка сравнивается со стандартными функциями языка C++.

Количество пар "ключ-значение"	BucketSort(), мс	MergeSort(), мс
100000	32458	48899
200000	104580	142907
300000	212336	284310
400000	371465	461866
500000	535948	693257
600000	798625	1016245
700000	1025358	1328589
800000	1533539	1783103
900000	1692229	2142605
1000000	2043095	2654381

Карманная сортировка работает быстрее MergeSort, и на больших объемах данных время ее работы становится пропорциональным количеству данных.

Ниже приведена программа benchmark.cpp, использовавшаяся для засечения времени работы функций:

```
#include <chrono>
#include <iostream>
#include <random>

#include "header.h"

int main() {
    TVector<TKeyValuePair> data;
    TVector<TKeyValuePair> data2;

    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<double> distr(-100, 100);
    for (size_t i = 100000; i <= 1000000; i+=100000) {
        const size_t numberOfElements = i;
        std::cout << numberOfElements << '\n';
        for (size_t i = 0; i < numberOfElements; ++i) {
            data.PushBack(TKeyValuePair(distr(gen), "test"));
            data2.PushBack(TKeyValuePair(distr(gen), "test"));
        }

        auto start = std::chrono::high_resolution_clock::now();
        BucketSort(data);
```

```

auto end = std::chrono::high_resolution_clock::now();
std::cout << "Bucket_sort_time:_ "
           << std::chrono::duration_cast<std::chrono::microseconds>
               (end - start)
               .count()
           << "_microseconds" << std::endl;

// Benchmarking std::sort
TVector<TKeyValuePair> buf(data2.Size());
start = std::chrono::high_resolution_clock::now();
MergeSort(data2, buf);
end = std::chrono::high_resolution_clock::now();
std::cout << "merge_sort:_ "
           << std::chrono::duration_cast<std::chrono::microseconds>
               (end - start)
               .count()
           << "_microseconds" << std::endl;
}

return 0;
}

```

Выводы

В ходе выполнения программы был реализован алгоритм карманной сортировки. Данная сортировка предполагает равномерное распределение входных данных, но позволяет сортировать вещественные числа за $O(n)$, в отличие от сортировки подсчетом. Помимо этого был получен опыт реализации шаблонных структур данных.