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 камранную сортировку. Ее суть заключается в том, что определяеятся функция, которая ставит в соответсвие 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 полученный 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 вспомгательного 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 функция языка C++.

Количество пар "ключ-значение"	BucketSort(), мс	std::stable_sort(), мс
100000	24874	31686
200000	51995	69242
300000	88199	88199
400000	104921	136714
500000	144899	170380
600000	184554	199280
700000	182360	234054
800000	227148	292439
900000	269784	326355
1000000	283568	398958
5000000	1654703.84	2106763

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

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

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

#include "header.h"

int main() {
    TVector<TKeyValuePair> data;
    std::vector<TKeyValuePair> benchmarkData;

    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<double> distr(-100, 100);

    const std::size_t numberOfElements = 100000;

    for (size_t i = 0; i < numberOfElements; ++i) {
        data.PushBack(TKeyValuePair(distr(gen), "test"));
        benchmarkData.emplace_back(distr(gen), "test");
    }
    // Benchmarking bucket sort
    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
    start = std::chrono::high_resolution_clock::now();
    std::stable_sort(benchmarkData.begin(), benchmarkData.end());
    end = std::chrono::high_resolution_clock::now();
    std::cout << "std::stable_sort time: "
    << 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 распрделение входны данных, но позволяет сортировать вещественные числа за O(n), в отличие от сортировки подсчетом. Помимо этого был получен опыт реализации шаблонных структур данных.