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

Выполнил студент группы М80-208Б-22 МАИ *Цирулев Николай*.

Условие

Кратко описывается задача:

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:

   - Сортировка подсчётом.
   - Тип ключа: числа от 0 до 10^5.
   - Тип значения: буквы латинского алфавита.

Метод решения

Для хранения входных данных был написан вектор с динамическим выделением памяти, так как количество строк входных данных заранее неизвестно. Сортировка подсчетом была реализована с использованием массива префиксных сумм для более быстрой работы алгоритма.

Описание программы

Исходный код программы содержится в файле main.cpp. В классе `my_vector` реализованы все необходимые для задания методы, конструкторы, деструкторы а также перегрузки необходимых для работы операторов:

```
template <typename T>
class my_vector {
    T* _data;
    size_t _size, _capacity;

   public:
    my_vector();
    my_vector(size_t init_size);
    my_vector(const my_vector& other);
    my_vector(my_vector&& other);
    ~my_vector();
```

```
    bool is_empty();
    void resize(size_t new_size);
    void push_back(const T& new_elem);

    T* data() const;
    size_t size() const;
    size_t capacity() const;

    T& operator[](size_t index);
    const T& operator[](size_t index) const;

  private:
    static T* allocate(size_t new_capacity);
    void reallocate(size_t new_capacity);
};

template <typename T>
size_t my_vector<T>::size() const {
    return _size;
}

template <typename T>
size_t my_vector<T>::capacity() const {
    return _capacity;
}

template <typename T>
T* my_vector<T>::data() const {
    return _data;
}

template <typename T>
T* my_vector<T>::allocate(size_t new_capacity) {
    return new T[new_capacity];
}

template <typename T>
void my_vector<T>::reallocate(size_t new_capacity) {
    T* temp = allocate(new_capacity);
    std::copy(_data, _data + _size, temp);
    _capacity = new_capacity;
```

```
    delete[] _data;
    _data = temp;
}

template <typename T>
my_vector<T>::my_vector() : _size(0), _capacity(0), _data(nullptr) {}

template <typename T>
my_vector<T>::my_vector(size_t init_size) : _size(init_size), _capacity(ini
    _data = allocate(_capacity);
}

template <typename T>
my_vector<T>::my_vector(const my_vector& other)
    : _size(other.size()), _capacity(other.capacity()) {
    _data = allocate(_capacity);
    std::copy(other.data(), other.data() + _size, _data);
}

template <typename T>
my_vector<T>::my_vector(my_vector&& other)
    : _size(other._size), _capacity(other._capacity), _data(other._data) {
    other._data = nullptr;
    other._size = 0;
    other._capacity = 0;
}

template <typename T>
my_vector<T>::~my_vector() {
    if (_data != nullptr) {
        delete[] _data;
        _data = nullptr;
    }
}

template <typename T>
bool my_vector<T>::is_empty() {
    return _size == 0;
}

template <typename T>
void my_vector<T>::resize(size_t new_size) {
```

```
    if (new_size <= _capacity) {
        _size = new_size;
    } else {
        size_t new_capacity = _capacity > size_t(1) ? _capacity : size_t(1)
        while (new_capacity < new_size) {
            new_capacity *= 2;
        }
        reallocate(new_capacity);
        _size = new_size;
    }
}

template <typename T>
void my_vector<T>::push_back(const T& new_elem) {
    resize(_size + 1);
    (*this)[_size - 1] = new_elem;
}

template <typename T>
T& my_vector<T>::operator[](size_t index) {
    return _data[index];
}

template <typename T>
const T& my_vector<T>::operator[](size_t index) const {
    return _data[index];
}
```

Структура `my_pair` необходима для для хранения пар "ключ-значение". Для ключа использовался тип данных `uint32_t`, который в большей мере подходит для заданных ограничений. Для значений был выбран тип `char`.

```
struct my_pair {
    uint32_t key;
    char value;
};
```

Функция `counting_sort()` содержит реализацию алгоритма сортировки.

```
void counting_sort(my_vector<size_t>& cnt, my_vector<my_pair>& arr) {
    for (size_t i = 1; i < cnt.size(); ++i) {
        cnt[i] = cnt[i - 1] + cnt[i];
    }

    my_vector<my_pair> res(arr.size());
```

```
    for (size_t i = arr.size(); i--> 0;) {
        uint32_t key = arr[i].key;
        res[cnt[key] - 1] = arr[i];
        --cnt[key];
    }
    arr = res;
}
```

Первая строчка `main` отключает синхронизацию потоков ввода-вывода. Вторая отвязывает стандартный поток ввода от стандартного потока вывода, блягодаря чему при каждом вызове `std::cin` не сбрасывается буфер. Обе эти строчки позволяют значительно ускорить ввод-вывод в программе. Далее идет ввод данных, вызывается `counting_sort()` и выводятся результаты.

```
int main() {
    std::ios_base::sync_with_stdio(false);
    std::cin.tie(0);
    my_vector<size_t> cnt(MAX_KEY);

    for (size_t i = 0; i < cnt.size(); ++i) {
        cnt[i] = 0;
    }

    my_vector<my_pair> arr;
    my_pair pair;

    while (std::cin >> pair.key >> pair.value) {
        ++cnt[pair.key];
        arr.push_back(pair);
    }
    counting_sort(cnt, arr);

    for (size_t i = 0; i < arr.size(); ++i) {
        std::cout << arr[i].key << '\t' << arr[i].value << '\n';
    }
    return 0;
}
```

Дневник отладки

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.

Сортировка подсчётом работает за линейное время. Если обратить на код программы, то видно, что реализация алгоритма состоит из нескольких последовательных циклов. Для большей наглядности приведём таблицу, в которой написанная сортировка сравнивается со стандартными функция языка C++.

Количество пар "ключ-значение"	counting_sort(), мс	std::sort(), мс	std::stable_sort(), мс
10000	5747	2084	976
100000	7019	25295	11696
1000000	32683	284048	149074
10000000	560556	3049667	1603556
100000000	6310612	33929127	18263965

На больших объёмах входных данных (от ста тысяч до десяти миллионов) становится заметно, что время сортировки пропорционально количеству пар "ключ-значение". Причём написанная функция оказывается заметно быстрее стандартных функций языка C++, так как те используют алгоритмы с временной сложностью $O(n \log n)$.

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

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

#include "main.cpp"

bool cmp(my_pair a, my_pair b) {
    return a.key < b.key;
}

int main() {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> key_dist(0, 100000);
    std::uniform_int_distribution<> char_dist(0, 25);
    for (int k = 1; k < 1000000000; k *= 10) {
```

```
        std::cout << k << '\t';
        std::vector<my_pair> benchmark_data;

        my_vector<size_t> cnt(MAX_KEY);

        for (size_t i = 0; i < cnt.size(); ++i) {
            cnt[i] = 0;
        }

        my_vector<my_pair> arr;
        my_pair pair;
        for (int i = 0; i < k; ++i) {
            pair.key = key_dist(gen);
            pair.value = static_cast<char>('a' + char_dist(gen));
            ++cnt[pair.key];
            arr.push_back(pair);
            benchmark_data.push_back(pair);
        }

        auto start1 = std::chrono::high_resolution_clock::now();
        counting_sort(cnt, arr);
        auto finish1 = std::chrono::high_resolution_clock::now();
        auto duration1 = std::chrono::duration_cast<std::chrono::microsecon
        std::cout << duration1.count() << '\t';

        auto start2 = std::chrono::high_resolution_clock::now();
        sort(benchmark_data.begin(), benchmark_data.end(), cmp);
        auto finish2 = std::chrono::high_resolution_clock::now();
        auto duration2 = std::chrono::duration_cast<std::chrono::microsecon

        std::cout << duration2.count() << '\t';

        auto start3 = std::chrono::high_resolution_clock::now();
        std::stable_sort(benchmark_data.begin(), benchmark_data.end(), cmp)
        auto finish3 = std::chrono::high_resolution_clock::now();
        auto duration3 = std::chrono::duration_cast<std::chrono::microsecon

        std::cout << duration3.count() << '\t';
        std::cout << std::endl;

    }
    return 0;
```

```
}
```

Выводы

В ходе выполнения данной работы были изучены алгоритмы линейных сортировок, также был реализован алгоритм сортировки подсчетом. При написании алгоритма не возникло проблем, так как он достаточно прост и понятен. Реализованный алгоритм сортировки имеет множество применений. Линейная временная сложность делает алгоритм эффективным для задач с ограниченным диапазоном значений.