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 $2^{64} - 1$.
   - 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

Для хранения входных данных и блоков (карманов) был написан вектор с выделением памяти на куче. Также для сортировки элементов в карманах была написана сортировка слиянием.

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 {
private:
    T * data;
    uint64_t size;
    uint64_t capacity;
public:
    my_vector() : data(nullptr), size(0), capacity(0) {}
    my_vector(uint64_t new_size) {
        data = new T[new_size];
```

```
    size = 0;
    capacity = new_size;
}
~my_vector() {
    if (data != nullptr) {
        delete[] data;
        data = nullptr;
    }
}
uint64_t get_size() {
    return size;
}
my_vector(const my_vector& other) : size(other.size), capacity(other.ca
    data = new T[capacity];
    std::copy(other.data, other.data + size, data);
}

my_vector(my_vector&& other) noexcept : data(other.data), size(other.si
    other.data = nullptr;
    other.size = 0;
    other.capacity = 0;
}

my_vector& operator=(const my_vector& other) {
    if (this != &other) {
        delete[] data;
        size = other.size;
        capacity = other.capacity;
        data = new T[capacity];
        std::copy(other.data, other.data + size, data);
    }
    return *this;
}
my_vector& operator=(my_vector&& other) noexcept {
    if (this != &other) {
        delete[] data;
        data = other.data;
        size = other.size;
        capacity = other.capacity;
        other.data = nullptr;
        other.size = 0;
        other.capacity = 0;
```

```
        }
        return *this;
    }
    void push_back(const T& elem) {
        if (size >= capacity) {
            throw std::out_of_range("Index out of range. Size: " + std::to_
        }
        data[size++] = elem;
    }
    void resize(const int k) {
        if (k <= 0) return;
        capacity += k;
        T* temp = new T[capacity];
        if (data != nullptr) {
            std::copy(data, data + size, temp);
        }
        delete[] data;
        data = temp;
    }
    T& operator[](uint64_t index) {
        return data[index];
    }
};
```

Структура `my_pair` необходима для для хранения пар "ключ-значение". Для ключа использовался тип данных `uint64_t`, который в большей мере подходит для заданных ограничений. Для значений был выбран массив `char` на стеке.

```
struct my_pair {
    uint64_t key;
    char value[SIZE_OF_STRING];
    my_pair() {
        std::memset(value, 0, SIZE_OF_STRING);
    }
    bool operator<(const my_pair& other) const {
        return key < other.key;
    }
    bool operator<=(const my_pair& other) const {
        return key <= other.key;
    }
};
```

Функции `merge_sort()` и `merge()` содержат реализацию алгоритма сортировки слиянием.

```
template <class T>
void merge(my_vector<T>& arr, my_vector<T>& buf, std::size_t left, std::siz
    std::size_t it1 = 0;
    std::size_t it2 = 0;
    while (left + it1 < mid && mid + it2 < right) {
        if (arr[left + it1] <= arr[mid + it2]) {
            buf[it1 + it2] = std::move(arr[left + it1]);
            it1 += 1;
        } else {
            buf[it1 + it2] = std::move(arr[mid + it2]);
            it2 += 1;
        }
    }
    while (left + it1 < mid) {
        buf[it1 + it2] = std::move(arr[left + it1]);
        it1 += 1;
    }

    while (mid + it2 < right) {
        buf[it1 + it2] = std::move(arr[mid + it2]);
        it2 += 1;
    }
    for (std::size_t i = 0; i < it1 + it2; ++i) {
        arr[left + i] = std::move(buf[i]);
    }
}

template <class T>
void merge_sort(my_vector<T>& arr, my_vector<T>& buf) {
    std::size_t count = arr.get_size();
    for (size_t i = 1; i < count; i *= 2) {
        for (size_t j = 0; j < count - i; j += 2 * i) {
            merge(arr, buf, j, j + i, min(j + 2 * i, count));
        }
    }
}
```

Функция `bucket_sort()` содержит реализацию алгоритма сортировки. Для поддержания необходимосй скорости алгоритма для сортировки элементов карманов используется сортировка слиянием. В ходе алгоритма для экономии используемой памяти вектор $\texttt{my}_v ector < my_p air > v$.

```
void bucket_sort(my_vector<my_pair> & v, uint64_t m) {
    uint64_t n = v.get_size();
```

```
    my_vector<my_pair> b[n];
    uint64_t temp1[n];
    int temp[n];
    for (uint64_t i = 0; i < n; ++i) {
        temp[i] = 0;
    }
    for (uint64_t i = 0; i < n; ++i) {
        temp1[i] = n * v[i].key / std::numeric_limits<uint64_t>::max();
        temp[temp1[i]]++;
    }
    for (uint64_t i = 0; i < n; ++i) {
        b[i].resize(temp[i]);
    }
    for (uint64_t i = 0; i < n; ++i) {
        b[temp1[i]].push_back(v[i]);
    }
    for (uint64_t i = 0; i < n; ++i) {
        if (b[i].get_size() > 1) {
            merge_sort(b[i], v);
        }
        for (uint64_t j = 0; j < b[i].get_size(); j++) {
            std::cout << b[i][j].key << "\t" << b[i][j].value << '\n';
        }
    }
}
```

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::sync_with_stdio(false);
    std::cin.tie(nullptr);
    my_vector<my_pair> v(1000000);
    my_pair p;
    uint64_t m = 0;
    while (std::cin >> p.key >> p.value) {
        p.value[SIZE_OF_STRING - 1] = '\0';
        v.push_back(p);
    }
    bucket_sort(v, m);
    return 0;
```

```
}
```

Дневник отладки

За время выполнения лабораторной работы было исправлено несколько проблем, из-за которых решение не проходило тесты.

В основном решение не проходило с ошибками ML(Memory limit exceeded) и TL(Time limit exceeded).

Для решения первой проблемы пришлось использовать сортировку слиянием вместо используемой изначально сортировки вставками. Также были сведены к минимуму долгие арфиметические операции.

Чтобы решить вторую проблему, требовалось уменьшить использование памяти программо Для этого, вместо динамического выделения памяти автоматически во время операции push_back, было решено выделять память вектора заранее, просчитав точное количество требуемой памяти. Также вместо аллоцирования дополнительной памяти под буфер для сортировки карманов использовался вектор с исходными данными.

Данные исправления позволили решить проблемы, из-за которых решение не проходило чекер.

Тест производительности

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

Карманная сортировка в лучшем случае работает за линейное время. Если посмотреть на код программы, то видно, что реализация алгоритма состоит из нескольких последоват циклов и одного вложенного для вывода результата. Также в одном из циклов мы видим вызов функции, которая реализует сортировку слиянием элементов карманов. Из-за того, что значения ключей в массиве, который требуется отсортировать, равномерно распределены, в лучшем случае на каждый карман приходится по одному элементу, поэтому сортировать элементы в карманах в этом случае не нужно.

Для большей наглядности приведём таблицу, в которой написанная сортировка сравнива со стандартными функция языка C++.

Количество пар "ключ-значение"	counting_sort(), мс	std::sort(), мс	std::stable_so
1	3786	2572	6751
10	4151	5474	1681
100	496	6785	6522
1000	6881	9948	6867
10000	88688	125848	77980
100000	1059858	1563252	941387

На больших объёмах входных данных становится заметно, что время сортировки пропорционально количеству пар "ключ-значение". Причём на достаточно больших объемах данных 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 <fstream>

#include "main.cpp"

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

int main() {
    std::ofstream out("outfile.txt", std::ios::app);
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<uint64_t> key_dist;
    std::uniform_int_distribution<> char_dist(0, 25);
    for (int k = 1; k <= 100000; k *= 10) {

        out << k << '\t';
        std::vector<my_pair> benchmark_data;
        my_vector<my_pair> arr(k);
        my_pair pair;
        for (int i = 0; i < k; ++i) {
            pair.key = key_dist(gen);
            for (int j = 0; j < SIZE_OF_STRING - 1; ++j) {
                pair.value[j] = static_cast<char>('a' + char_dist(gen))
            }
            pair.value[SIZE_OF_STRING - 1] = '\0';
            arr.push_back(pair);
            std::cout << k << '\n';
            benchmark_data.push_back(pair);
        }
```

```
            auto start1 = std::chrono::high_resolution_clock::now();
            bucket_sort(arr);
            auto finish1 = std::chrono::high_resolution_clock::now();
            auto duration1 = std::chrono::duration_cast<std::chrono::micros
            out << 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::micros

            out << duration2.count() << '\t';

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

            out << duration3.count() << '\t';
            out << '\n';

        }
        return 0;
}
```

Выводы

В ходе выполнения данной работы были изучены алгоритмы линейных сортировок, также был реализован алгоритм карманной сортировки. При написании алгоритма возникло несколько проблем, связанных со скоростью работы программы и использованием памяти, которые были успешно решены. Реализованный алгоритм сортировки имеет множество применений. Линейная временная сложность делает алгоритм эффективным для задач с ограниченным диапазоном значений и равномерно распределенными неотсортированными данными.