Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №3

по «Алгоритмам и структурам данных»

Базовые задачи

Выполнил:

Студент группы P3232

Чмурова М. В.

Преподаватели:

Косяков М.С.

Тараканов Д.С.

Санкт-Петербург

2024

Задача №I «Машинки»

Пояснение к примененному алгоритму:

Для того, чтобы реализовать эту программу необходимо знать какую из машинок нужно поднимать на полку, если Пете понадобилась еще одна машинка. Для этого необходимо для каждой машинки, стоящей на полу сохранять информацию о том, на каком шаге она понадобится. Для этого необходимо изначально сохранить информацию о том, на каком шаге какая машинка будет нужна и сохранить это в структуре вида list<int> cars_history[n];

После этого необходимо пройтись по всему массиву машинок и посчитать количество операций для каждого спуска машинки с полки.

Для определения какая машинка должна быть поднята необходимо исопльзовать структуру данных priority_queue<pair<int, int>>, так как она позволит быстро находить ту пару, у которой первое значение максимально.

Сложность алгоритма:

$O(n^2)$, где n – количество машинок

Код алгоритма:

```
#include <iostream>
#include <list>
#include <unordered_set>
#include <algorithm>
#include <queue>
#include <limits.h>

using namespace std;

int main() {
    int operations = 0;

    int n; // всего машинок
    int k; // машинок на полу
    int p; // номера машинок, с которыми играет

    cin >> n;
    cin >> k;
    cin >> p;

    // считать input в массив order_of_cars
    int order_of_cars[p];
    for (int i = 0; i < p; i++) {
```

```
        cin >> order_of_cars[i];
    }

    // посчитать для каждой машинки на каком шаге она будет нужна
    list<int> cars_history[n];
    for (int i = 0; i < p; i++) {
        int element = --order_of_cars[i];
        cars_history[element].push_back(i);
    }

    unordered_set<int> cars_on_the_floor;                        // для
определения какие машинки уже на полу
    priority_queue<pair<int, int>> next_time_of_necessity;  // для
определения какая машинка понадобится раньше

    for (int i = 0; i < p; i++) {

        int element = order_of_cars[i];
        cars_history[element].pop_front();

        // проверяем, что если машинки нет на полу, то берем с полки и
ставим на пол
        if (cars_on_the_floor.find(element) == cars_on_the_floor.end()) {
            // если еще есть место без замены, то ставим
            if (cars_on_the_floor.size() < k) {
                cars_on_the_floor.insert(element);
                operations++;
            // иначе меняем местами с машинкой, которая встретится позже
всех
            } else {
                // удаляем машинку, которая встретится позже всех

cars_on_the_floor.erase(next_time_of_necessity.top().second);
                // удаляем ее из списка так как она больше не на полу
                next_time_of_necessity.pop();
                // и ставим на пол другую машинку
                cars_on_the_floor.insert(element);
                operations++;
            }
        }

        // если эта машинка больше не понадобится, то она должна быть
первой на полку в следующий раз
        if (cars_history[element].empty()) {
            next_time_of_necessity.push({INT_MAX, element});
        }
        // иначе просто добавляем номер, когда она встретится в следующий
раз
        else {
            next_time_of_necessity.push({cars_history[element].front(),
element});
        }
    }
```

```
    cout << operations;
}
```

Задача №J «Гоблины и очереди»

Пояснение к примененному алгоритму:

В данной задаче используются очереди, поэтому логично использовать структуру данных deque<int> так как необходимо вставка в конец и удаление из начала – то есть операции обращения к началу и концу массива.

Для того, чтобы вставлять приоритетных гоблинов в середину очереди можно разделить одну очередь на две половины и вставлять приоритетного гоблина в конец первой половины или в начало второй в зависимости от размеров очередей.

Таким образом, получится достичь максимальной оптимальности выполнения кода

Сложность алгоритма:

O(n), где n – количество передвижений гномов

Код алгоритма:

```
#include<iostream>
#include<deque>
#include <sstream>
#include <string>

using namespace std;

int main() {

    int n;                  // общее количество операций
    cin >> n;
    cin.ignore();

    string line;
    string sign;            // для считывания введенного знака
    int number;             // для считывания введенного номера
    deque<int> first_half;   // левая половина очереди
    deque<int> second_half;  // правая половина очереди

    while (n--) {
        getline(cin, line);
```

```
        istringstream iss(line);

        iss >> sign;

        if (sign == "-") {
            cout << first_half.front() << " ";
            first_half.pop_front();
            if (first_half.size() != second_half.size()) {
                first_half.push_back(second_half.front());
                second_half.pop_front();
            }
        }

        if (sign == "+") {
            iss >> number;
            if (first_half.size() != second_half.size()) {
                second_half.push_back(number);
            } else {
                if (!first_half.empty()) {
                    first_half.push_back(second_half.front());
                    second_half.pop_front();
                    second_half.push_back(number);
                }
                else
                {
                    first_half.push_back(number);
                }
            }
        }

        if (sign == "*") {
            iss >> number;
            if (first_half.size() == second_half.size()) {
                first_half.push_back(number);
            } else {
                second_half.push_front(number);
            }
        }
    }
}
```

Задача №K «Менеджер памяти - 1»

Пояснение к примененному алгоритму:

Для реализации менеджера памяти необходимо хранить информацию о том, какие операции по вставке и удалению были совершены в pair <int, int>, а также массив массивов хранения свободных позиций в памяти. Кроме того, свободные позиции необходимо хранить

в двух экземплярах – отсортированных по индексам и по количеству свободных ячеек. Это необходимо для оптимальных операций поиска во время освобождения и выделения памяти.

Сложность алгоритма:

O(m * log n), где m – количество запросов, n – размер памяти

Код алгоритма:

```
#include <iostream>
#include <map>
#include <vector>
using namespace std;

int main() {
    int n;
    int m;

    cin >> n;
    cin >> m;

    multimap<int, int> free_by_size;
    map<int, int> free_by_index;
    vector<pair<int, int>> busy(m);

    int current;
    int index;
    int size;

    free_by_index.insert({1, n});
    free_by_size.insert({n, 1});

    for (int i = 0; i < m; busy[i] = {current, index}, i++){
        cin >> current;

        if (current > 0) {
            auto it = free_by_size.lower_bound(current);
            if (it == free_by_size.end()) index = -1;
            else {
                index = it->second;
                size = it->first - current;

                free_by_index.erase(it->second);
                free_by_size.erase(it);
                if (size > 0) {

                    free_by_index.insert({index + current, size});
                    free_by_size.insert({size, index + current});
                }
```

```
            }
            cout << index << endl;
        } else {
            int index_x = busy[abs(current) - 1].second;
            int size_x = busy[abs(current) - 1].first;
            if (index_x == -1) continue;

            auto it_current = free_by_index.lower_bound(index_x);
            decltype(free_by_index.begin()) it_next;
            if (it_current == free_by_index.begin()) {
                it_next = free_by_index.end();
            } else {
                it_next = prev(it_current);
            }

            if (it_current != free_by_index.end() && it_current->first ==
index_x + size_x){
                if (it_next != free_by_index.end() && it_next->first +
it_next->second == index_x) {
                    index = it_next->first;
                    size = it_next->second + it_current->second;

                    auto it_1 = free_by_size.find(it_next->second);
                    while (it_1->second != it_next->first) it_1++;
                    free_by_size.erase(it_1);
                    free_by_index.erase(it_next);

                    auto it_2 = free_by_size.find(it_current->second);
                    while (it_2->second != it_current->first) it_2++;
                    free_by_size.erase(it_2);
                    free_by_index.erase(it_current);

                    free_by_index.insert({index, size + size_x});
                    free_by_size.insert({size + size_x, index});

                } else {
                    size = it_current->second;

                    auto it_d = free_by_size.find(it_current->second);
                    while (it_d->second != it_current->first) it_d++;
                    free_by_size.erase(it_d);
                    free_by_index.erase(it_current);

                    free_by_index.insert({index_x, size + size_x});
                    free_by_size.insert({size + size_x, index_x});
                }
            } else {
                if (it_next != free_by_index.end() && it_next->first +
it_next->second == index_x) {
                    index = it_next->first;
                    size = it_next->second;

                    auto it_d = free_by_size.find(it_next->second);
```

```
                    while (it_d->second != it_next->first) it_d++;
                    free_by_size.erase(it_d);
                    free_by_index.erase(it_next);

                    free_by_index.insert({index, size + size_x});
                    free_by_size.insert({size + size_x, index});

                } else {
                    free_by_index.insert({index_x, size_x});
                    free_by_size.insert({size_x, index_x});
                }
            }

            index = 0;
        }
    }
    return 0;
}
```

Задача №L «Минимум на отрезке»

Пояснение к примененному алгоритму:

Аналогично задаче «Гоблины и очереди» будем использовать deque<int> для того, чтобы при перемещении «окна» удалять элемент из конца очереди и вставлять следующий элемент в конец очереди.

В начало очереди будет ставить индекс минимального элемента и потом снимать его, чтобы вывести как результат. После этого будет удалять тот элемент, который уходит из окна при его перемещении. Затем, аналогично, в начало очереди будем ставить индекс минимального элемента массива.

Сложность алгоритма:

O(n), где n – количество элементов в данном массиве

Код алгоритма:

```
#include<iostream>
#include<deque>
#include<vector>

using namespace std;

int main() {
```

```
    int n; // общее количество чисел
    int k; // количество чисел в окне

    cin >> n >> k;

    int numbers[n];
    for (int i = 0; i < n; i++) {
        cin >> numbers[i];
    }

    deque<int> window;

    for (int i = 0; i < k; i++) {

        // если в начале очереди элемент меньше чем текущий, то снимаем
элемент с начала очереди
        while (!window.empty() && numbers[i] < numbers[window.front()]) {
            window.pop_front();
        }
        // индекс в начало очереди
        window.push_front(i);
    }

    cout << numbers[window.back()];

    for (int i = k; i < n; i++) {

        // избавляемся от индексов, которые не могут быть в текущем окне
        while (!window.empty() && window.back() == i - k) {
            window.pop_back();
        }

        // добавляем следующий индекс
        while (!window.empty() && numbers[i] < numbers[window.front()]) {
            window.pop_front();
        }
        window.push_front(i);

        // в конце очереди получаем наименьший элемент
        cout << " " << numbers[window.back()];

    }
}
```