Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №3

по «Алгоритмам и структурам данных»

Структуры

Выполнил:

Студент группы P3231

Савон Г.К.

Преподаватели:

Косяков М.С.

Тараканов Д.С.

Санкт-Петербург

2021

Задача I «Машинки»

```
#include <iostream>
#include <queue>
#include <deque>
#include <map>
#include <vector>
#include <set>

using namespace std;

int main()
{
    int n, k, p;
    int car;
    cin >> n >> k >> p;
    vector<int> traffic_jam;
    vector<deque<int> > cars_positions;
    cars_positions.resize(n + 1);
    for (int i = 0; i < p; i++) {
        cin >> car;
        traffic_jam.push_back(car);
        cars_positions[car].push_back(i);
    }
    set<int> bad_set;
    set<int> now_cars;
    int car_pos;
    int now_car;
    int ans = 0;
    int i = 0;
    while (i < p && now_cars.size() < k) {
        now_car = traffic_jam[i];
        if (now_cars.find(now_car) != now_cars.end())
            bad_set.erase(i);
        else {
            now_cars.insert(now_car);
            ans++;
        }
        cars_positions[now_car].pop_front();
        if (cars_positions[now_car].size() == 0) {
            bad_set.insert(i);
        }
        else
            bad_set.insert(cars_positions[now_car].front());
        i++;
    }
    int delete_car_pos;
    int new_car;
    int del_car;
    set<int>::iterator it;
    for (int j = i; j < p; j++) {
        new_car = traffic_jam[j];
        if (*bad_set.begin() < j)
            delete_car_pos = *bad_set.begin();
        else {
            it = bad_set.end();
```

```
            --it;
            delete_car_pos = *it;
        }
        del_car = traffic_jam[delete_car_pos];
        if (!now_cars.count(new_car)) {
            ans++;
            now_cars.erase(del_car);
            now_cars.insert(new_car);
            bad_set.erase(delete_car_pos);
        }
        else {
            bad_set.erase(cars_positions[new_car].front());
        }
        cars_positions[new_car].pop_front();
        if (cars_positions[new_car].size() == 0) {
            bad_set.insert(j);
        }
        else
            bad_set.insert(cars_positions[new_car].front());
    }
    cout << ans;
    return 0;
}
```

Пояснение к примененному алгоритму:

На полу стоят машинки, нужно решить, какую убрать – ту которая не встретится больше, ну или ту, которая встретится позже, чем все остальные. Поэтому для каждой машинки запишем все номера, под которыми она идет в общем списке. Будем сравнивать их и еще проверять на то, не последнее ли это появление было для какой-нибудь из них. Но чтобы на каждом шаге не искать для каждой машинки на полу ее следующую встречу, будем хранить следующие встречи стоящих на полу машинок отдельно в сете, так они уже будут отсортированы. А если эта машинка встретилась последний раз, запишем ее нынешнее положение в очереди машинок и перед удалением будем проверять, нет ли в сете позиций имеющихся машинок позиции, которая меньше, чем мы сейчас обрабатываем.

Сложность: O(p log(k))

Задача J «Гоблины и Очереди»

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

using namespace std;

int main()
{
```

```
    int n;
    cin >> n;
    char c;
    int number;
    deque<int> que_start;
    deque<int> que_end;
    for (int i = 0; i < n; i++) {
        cin >> c;
        if (c == '-') {
            cout << que_start.front() << endl;
            que_start.pop_front();
        }
        else {
            cin >> number;
            if (c == '+')
                que_end.push_back(number);
            else
                que_end.push_front(number);
        }
        if (que_start.size() < que_end.size()) {
            que_start.push_back(que_end.front());
            que_end.pop_front();
        }
    }
    return 0;
}
```

Пояснение к примененному алгоритму:

Сначала я пыталась пихать всех гоблинов в один список и хранить итератор на серединке, двигать его из стороны в стороны в зависимости от того, какой шальной гоблин куда придет или уйдет, но как-то не задалось, да и вообще как-то грустно, когда задача называется «гоблины и очереди», а я их даже не в очередь записываю.

В целом я поняла, что этот итератор постоянно сдвигаться должен на 1(или не сдвигаться вообще), поэтому можно поделить список пополам и перекидывать лишних гоблинов из второй части в первую. И никакой мороки с итератором не будет.

Сложность: O(n)

Задача K «Менеджер памяти»

```
#include <iostream>
#include <set>
#include <map>
#include <vector>

using namespace std;

int main()
{
```

```
    int m, n;
    cin >> m >> n;
    int the_ask;
    set<vector<int> > space_length, space_start; //пустые отрезки(длина,
начало.. начало, длина)
    map<int, vector<int> > ask_info; //запросник(номер: начало, длина)
    vector<int> vec;
    vec.push_back(m);
    vec.push_back(1);
    space_length.insert(vec);
    vec.clear();
    vec.push_back(1);
    vec.push_back(m);
    space_start.insert(vec);
    int using_start;
    int using_length;
    vector<int> new_one, del;
    int new_len, new_start;
    vector<int> post_vec, prev_vec;
    set<vector<int> >::iterator it;
    for (int i = 0; i < n; i++) {
        cin >> the_ask;
        if (the_ask > 0) {
            if (space_length.size() == 0 || the_ask > (*--space_length.end())[0])
                cout << "-1" << endl;
            else {
                cout << (*--space_length.end())[1] << endl;
                using_length = (*--space_length.end())[0];
                using_start = (*--space_length.end())[1];
                ask_info[i].push_back((*--space_length.end())[1]);
                ask_info[i].push_back(the_ask);
                space_length.erase(*--space_length.end());
                vec.clear();
                vec.push_back(using_start);
                vec.push_back(using_length);
                space_start.erase(vec);
                if (using_length != the_ask) {
                    vec.clear();
                    vec.push_back(using_length - the_ask);
                    vec.push_back(using_start + the_ask);
                    space_length.insert(vec);
                    vec.clear();
                    vec.push_back(using_start + the_ask);
                    vec.push_back(using_length - the_ask);
                    space_start.insert(vec);
                }
            }
        }
        else {
            if (ask_info.find(-1 * the_ask - 1) != ask_info.end()) {
                vec.clear();
                vec = ask_info[-1 * the_ask - 1];
                new_start = vec[0];
                new_len = vec[1];
                if (space_start.size() != 0) {
                    new_one.clear();
                    new_one.push_back(new_start);
```

```
                    new_one.push_back(new_len);
                    if (space_start.lower_bound(new_one) == space_start.end()
|| space_start.size() < 2) {
                        post_vec = *(--space_start.end());
                        prev_vec = *(--space_start.end());
                    }
                    else {
                        post_vec = *space_start.lower_bound(new_one);
                        if (post_vec != *space_start.begin())
                            prev_vec = *(--space_start.lower_bound(new_one));
                        else
                            prev_vec = post_vec;
                    }
                    if (vec[0] + vec[1] == post_vec[0]) {
                        new_len += post_vec[1];
                        del.clear();
                        del.push_back(post_vec[1]);
                        del.push_back(post_vec[0]);
                        space_length.erase(del);
                        space_start.erase(post_vec);
                    }
                    if (prev_vec[0] + prev_vec[1] == vec[0]) {
                        new_len += prev_vec[1];
                        new_start = prev_vec[0];
                        del.clear();
                        del.push_back(prev_vec[1]);
                        del.push_back(prev_vec[0]);
                        space_length.erase(del);
                        space_start.erase(prev_vec);
                    }
                }
                new_one.clear();
                new_one.push_back(new_len);
                new_one.push_back(new_start);
                space_length.insert(new_one);
                new_one.clear();
                new_one.push_back(new_start);
                new_one.push_back(new_len);
                space_start.insert(new_one);
                ask_info.erase(-1 * the_ask - 1);
            }
        }
    }

    return 0;
}
```

<u>Пояснение к примененному алгоритму:</u>

Чтобы занять память нужно знать:

-какого размера есть свободные куски

-где эти куски начинаются

Чтобы удалять запросы надо знать:

-что в этих запросах находилось(какое кол-во памяти запрашивалось)

-где оно лежит(где начинаются)

Чтобы после удаления рядомстоящие куски сливать:

-нужно знать индексы начал пустых кусков(желательно отсортированные)

-и длины кусков

Таким образом мы имеем мапу вида «номер запроса: где и какой кусок она занимает» и два сета с пустыми отрезками – в одном «номер, длина», в другом – «длина, номер»

При добавлении нового элемента достаем самый большой и красивый пустой отрезок с конца сета «длина, номер», уменьшаем его на длину запроса, уменьшаем его во втором сете, добавляем запрос в мапу.

При удалении – достаем из мапы информацию об отрезке. Находим в сете пустых отрезков «начало, длина» место, где бы его можно было положить(между какими кусками), смотрим, соприкасаются ли они, и если это так – удаляем маленькие пустые отрезки из обоих сетов и добавляем один большой-большой.

Сложность: (nlog(n))

Задача L «Минимум на отрезке»

```
#include <iostream>
#include <queue>
#include <set>

using namespace std;

int main()
{
    int n, k;
    cin >> n >> k;
    int number;
    queue<int> que;
    multiset<int> mset;
    for (int i = 0; i < k; i++) {
        cin >> number;
        que.push(number);
        mset.insert(number);
    }
```

```
    cout << *mset.begin() << " ";
    for (int i = k; i < n; i++) {
        mset.erase(mset.find(que.front()));
        que.pop();
        cin >> number;
        que.push(number);
        mset.insert(number);
        cout << *mset.begin() << " ";
    }
    return 0;
}
```

Пояснение к примененному алгоритму:

Приятная задача, вспоминаю ее, аж на душе тепло. Ну значит нам надо знать имеющиеся на данный момент все числа в отрезке, порядок, в котором они пришли и уйдут, а еще заодно знать, какое из них самое маленькое.

Было бы глупо каждый раз бегать по всему отрезку ища маленькое, поэтому будем отдельно хранить отсортированные числа с отрезка, это вцелом можно сделать почти в любом виде, но вот ведь незадача, их надо удалять оттуда, когда отрезок сдвигается. Есть невероятная структура set, отвечающая всем этим запросам. Мы легко достаем наименьший элемент и удаляем элемент по значению, не структура, а сказка! (Числа могут повторяться, так что берем мультисет)

Сложность: O(nlog(n))