Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования «Национальный исследовательский университет ИТМО»

Факультет Программной инженерии и компьютерной техники

Решения задач блока 3

Алгоритмам и структурам данных

Работу выполнил:

Гаврилин О.С.

Группа:

P3230

Санкт-Петербург,

2025

**1. I**

#include <iostream>

#include <set>

#include <unordered\_map>

#include <unordered\_set>

#include <vector>

using namespace std;

void solve() {

int totalCars, maxOnFloor, totalRequests;

cin >> totalCars >> maxOnFloor >> totalRequests;

vector<int> requests(totalRequests);

for (int i = 0; i < totalRequests; ++i) { cin >> requests[i];

}

const int notUsed = totalRequests + 1;

vector<int> nextUsage(totalRequests);

vector<int> futurePosition(totalCars + 1, notUsed);

for (int i = totalRequests - 1; i >= 0; --i) { nextUsage[i] = futurePosition[requests[i]]; futurePosition[requests[i]] = i;

}

unordered\_set<int> floorSet; floorSet.reserve(maxOnFloor \* 2); unordered\_map<int, int> nextIndex; nextIndex.reserve(maxOnFloor \* 2); set<pair<int, int>> usageOrder;

int actionCount = 0;

for (int i = 0; i < totalRequests; ++i) { int carId = requests[i];

int upcoming = nextUsage[i];

if (floorSet.count(carId)) { usageOrder.erase({nextIndex[carId], carId}); nextIndex[carId] = upcoming; usageOrder.insert({upcoming, carId});

} else { ++actionCount;

if (floorSet.size() ==

static\_cast<size\_t>(maxOnFloor)) {

auto it = prev(usageOrder.end());

int toRemove = it->second;

usageOrder.erase(it);

floorSet.erase(toRemove);

nextIndex.erase(toRemove);

}

floorSet.insert(carId); nextIndex[carId] = upcoming; usageOrder.insert({upcoming, carId});

}

}

cout << actionCount;

}

int main() {

ios::sync\_with\_stdio(false);

cin.tie(nullptr);

solve();

return 0;

}

**Описание:** Использую set для быстрого поиска машинки, которую можно заменить, текущий набор машинок на полу храню через unordered\_set. В цикле обновляем информацию о следующем использовании текущей машинки в очереди, обновляем данные в зависимости от того на полу она или нет.

**Сложность по времени: O(P log K)**

**Сложность по памяти: O (N + P + K)**

**2. J**

#include <deque>

#include <iostream>

#include <string>

using namespace std;

void solve() {

int q;

cin >> q;

deque<int> front\_half, back\_half;

while (q--) {

string action;

cin >> action;

if (action[0] == '+') {

int val;

cin >> val;

back\_half.emplace\_back(val);

} else if (action[0] == '\*') { int val;

cin >> val;

front\_half.emplace\_back(val); } else {

if (!front\_half.empty()) {

cout << front\_half.front() << '\n'; front\_half.pop\_front();

} else {

cout << back\_half.front() << '\n'; back\_half.pop\_front();

}

}

if (front\_half.size() < back\_half.size()) { front\_half.push\_back(back\_half.front()); back\_half.pop\_front();

} else if (front\_half.size() > back\_half.size() + 1)

{

back\_half.push\_front(front\_half.back()); front\_half.pop\_back();

}

}

}

int main() {

ios\_base::sync\_with\_stdio(false);

cin.tie(nullptr);

solve();

}

**Описание:** Реализуем решение с помощью двух деков. Обычные гоблины добавляются в конец очереди, привелигерованный в середину (в данном случае конец первой половины). При вызове

“-” удаляется первый гоблин из начала очереди и проверяется front\_half, затем back\_half если первый пустой

**Сложность по времени: O(q)**

**Сложность по памяти: O(q)**

**3. K**

#include <iostream>

#include <map>

#include <vector>

using namespace std;

void solve() {

multimap<int, int> blocks\_by\_size; map<int, int> blocks;

auto remove = [&](const multimap<int, int>::iterator& it)

{

blocks.erase(it->second);

blocks\_by\_size.erase(it);

};

auto remove\_by\_size = [&](const map<int, int>::iterator& it) {

auto it\_d = blocks\_by\_size.find(it->second); while (it\_d->second != it->first)

++it\_d;

blocks\_by\_size.erase(it\_d);

blocks.erase(it);

};

auto insert = [&](const pair<int, int>& pair) {

blocks.insert(pair);

blocks\_by\_size.insert({pair.second, pair.first});

};

int n, m;

cin >> n >> m;

vector<pair<int, int>> history(m);

insert({1, n});

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

int k, index = 0, size = 0;

cin >> k;

if (k > 0) {

auto it = blocks\_by\_size.lower\_bound(k); if (it == blocks\_by\_size.end()) {

index = -1;

} else {

index = it->second; size = it->first - k; remove(it);

if (size > 0) insert({index + k, size});

}

cout << index << '\n';

history[i] = {k, index};

} else {

history[i] = {k, 0}; int idx = abs(k) - 1;

int index\_x = history[idx].second; int size\_x = history[idx].first;

if (index\_x == -1) continue;

auto it\_r = blocks.lower\_bound(index\_x);

auto it\_l = (it\_r != blocks.begin()) ? prev(it\_r) :

blocks.end();

if (it\_r != blocks.end() && it\_r->first == index\_x + size\_x) {

if (it\_l != blocks.end() && it\_l->first + it\_l->second == index\_x) {

index = it\_l->first;

size = it\_l->second + it\_r->second; remove\_by\_size(it\_l); remove\_by\_size(it\_r); insert({index, size + size\_x});

} else {

size = it\_r->second; remove\_by\_size(it\_r); insert({index\_x, size + size\_x});

}

} else {

if (it\_l != blocks.end() && it\_l->first + it\_l->second == index\_x) {

index = it\_l->first;

size = it\_l->second;

remove\_by\_size(it\_l);

insert({index, size + size\_x});

} else { insert({index\_x, size\_x});

}

}

}

}

}

int main() {

ios\_base::sync\_with\_stdio(false);

cin.tie(nullptr);

solve();

}

**Описание:** Использую map для отображения начала свободного блока и его размера, что позволяет быстро искать соседей, multimap хранит те же блоки, но позволяет быстро находить минимально подходящий по размеру блок. Запрос на выделение памяти – ищем самый левый свободный блок с размером >= k, если находим такой то удаляем блок из обоих контейнеров и возвращаем начало выделенного блока. Запрос на освобождение – ищем history[|t| - 1] пару выделенного блока и его индекса, если находим то пытаемся объединить с соседними свободными блоками слева и справа, обновляем структуры.

**Сложность по времени: O (log n)**

**Сложность по памяти: O (m + n)**

**4. L**

#include <deque>

#include <iostream>

#include <vector>

using namespace std;

void solve() {

int n, k;

cin >> n >> k;

vector<int> a(n);

for (int& x : a)

cin >> x;

deque<int> dq;

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

while (!dq.empty() && dq.front() <= i - k) { dq.pop\_front();

}

while (!dq.empty() && a[dq.back()] >= a[i]) { dq.pop\_back();

}

dq.push\_back(i);

if (i >= k - 1) {

cout << a[dq.front()] << ' ';

}

}

}

int main() {

ios::sync\_with\_stdio(false);

cin.tie(nullptr);

solve();

return 0;

}

**Описание:** Задача сводится к тому, чтобы эффективно анализировать новое окно и заранее знать минимальные элементы в нем. При нахождении нового окна и обладании данными о старом, нам необходимо проанализировать ушедший

* добавленный элементы. Дек в данном случае позволяет хранить элементы в текущем окне в порядке возрастания, что позволяет за O (n) находить следующий минимум в окне, если предыдущей был убран.

**Сложность по времени: O (n)**

**Сложность по памяти: O (n)**