Федеральное государственное автономное образовательное учреждение высшего образования Национальный исследовательский университет ИТМО

Факультет программной инженерии и компьютерной техники

Алгоритмы и структуры данных

Задачи I, J, K, L (Яндекс.Контест)

Выполнил: студент группы P3208, Васильев Н. А.

Преподаватель: Косяков М. С.

Санкт-Петербург 2025

# Задача I. Машинки

Для каждой машины я храню все индексы её появлений в carsOrder. Это позволяет быстро получать информацию о следующем появлении конкретной машины.

В active хранится множество машин, которые в данный момент находятся у ребенка. Проверка и удаление работают за O(1), поэтому unordered\_set подходит идеально.

pq это очередь с приоритетом, где я храню пары: <время следующего использования, номер машины>. Таким образом, я всегда могу быстро найти ту машину, которая будет использована позже всего и удалить её из активной зоны.

Алгоритм работает за **O(p log k)**, так как основная нагрузка идёт на pq, которая содержит максимум k элементов.

## Код:

#include <deque>

#include <iostream>

#include <queue>

#include <unordered\_map>

#include <unordered\_set>

#include <vector>

using namespace std;

int main() {

  int n, k, p = 0;

  cin >> n >> k >> p;

  vector<int> carsOrder(p);

  unordered\_map<int, deque<int>> cars(p);

  for (int i = 0; i < p; i++) {

    int value;

    cin >> value;

    carsOrder[i] = value;

    cars[value].push\_back(i);

  }

  int operations = 0;

  unordered\_set<int> active;

  priority\_queue<pair<int, int>> pq;

  for (int i = 0; i < p; i++) {

    int currentCar = carsOrder[i];

    cars[currentCar].pop\_front();

    if (active.count(currentCar)) {

      int nextUsage = cars[currentCar].empty() ? p + 1 : cars[currentCar].front();

      pq.emplace(nextUsage, currentCar);

      continue;

    }

    operations++;

    if ((int)active.size() >= k) {

      while (!pq.empty()) {

        auto [nextUsage, car] = pq.top();

        pq.pop();

        if (active.find(car) != active.end()) {

          active.erase(car);

          break;

        }

      }

    }

    active.insert(currentCar);

    int nextUsage = cars[currentCar].empty() ? p + 1 : cars[currentCar].front();

    pq.emplace(nextUsage, currentCar);

  }

  cout << operations << endl;

  return 0;

}

# Задача J. Гоблины и очереди

Нашу память удобно разделить пополам для выполнения операции \*. Двунаправленные очереди подходят для этого лучше всего, потому что позволяют быстро вставлять в начало и конец за O(1) и быстро удалять из начала и с конца за O(1). В результате получаем, что каждая операция (+ i, \* i, -) выполняется за **O(1)** благодаря deque. В каждой итерации максимум 1 перенос элемента из одной очереди в другую, соответственно, O(1). Так как m — средняя длина строки команды, s — средняя длина значений, которые добавляются, то v занимает O(n \* m) памяти, result O(n \* s). В итоге получаем **O(n \* max(m, s))**.

Код:

#include <algorithm>

#include <iostream>

#include <queue>

#include <string>

using namespace std;

int main() {

  int n;

  cin >> n;

  cin.ignore();

  deque<string> left, right;

  vector<string> v(n);

  vector<string> result;

  for (int i = 0; i < n; i++) {

    getline(cin, v[i]);

    v[i].erase(remove(v[i].begin(), v[i].end(), ' '), v[i].end());

  }

  for (const string& s : v) {

    switch (s[0]) {

      case '+': {

        string x = s.substr(1, s.length());

        right.push\_back(x);

        break;

      }

      case '\*': {

        string x = s.substr(1, s.length());

        right.push\_front(x);

        break;

      }

      case '-': {

        result.push\_back(left.front());

        left.pop\_front();

        break;

      }

      default:

        break;

    }

    while (left.size() < right.size()) {

      left.push\_back(right.front());

      right.pop\_front();

    }

    while (left.size() > right.size() + 1) {

      right.push\_front(left.back());

      left.pop\_back();

    }

  }

  for (size\_t i = 0; i < result.size(); i++) {

    cout << result[i] << endl;

  }

  return 0;

}

# Задача K. Менеджер памяти-1

Для удобства используем структуру Block, где будем хранить начало и длину блока. Структуры будем хранить в векторе, чтобы быстро находить место для вставки и проверять соседние блоки. Также создадим вектор историй, который хранит историю выделений, то есть размер и адрес начала блока для каждого запроса, чтобы точно знать, какой блок нужно освободить по индексу. Вставка и изменение блока при таком подходе занимает O(1), поиск позиции вставки – O(log f), где f – количество свободных блоков. Итого получаем **O(m log f)**. Вектор free\_blocks хранит максимум n блоков, history и results содержат максимум m значений, в итоге получаем **O(n + m)** памяти.

Код:

#include <iostream>

#include <iterator>

#include <vector>

using namespace std;

struct Block {

  int start, size;

  bool operator<(const Block& other) const {

    return start < other.start;

  }

};

int main() {

  int n, m;

  cin >> n >> m;

  vector<Block> free\_blocks = {

      {1, n}

  };

  vector<pair<int, int>> history(m);

  vector<int> result;

  for (int i = 0; i < m; ++i) {

    int req;

    cin >> req;

    if (req > 0) {

      int idx = -1;

      for (size\_t j = 0; j < free\_blocks.size(); ++j) {

        if (free\_blocks[j].size >= req) {

          idx = free\_blocks[j].start;

          history[i] = {req, idx};

          if (free\_blocks[j].size == req) {

            free\_blocks.erase(free\_blocks.begin() + j);

          } else {

            free\_blocks[j].start += req;

            free\_blocks[j].size -= req;

          }

          break;

        }

      }

      result.push\_back(idx);

    } else {

      int prev\_idx = -req - 1;

      int size = history[prev\_idx].first;

      int start = history[prev\_idx].second;

      if (start == -1)

        continue;

      Block new\_block = {start, size};

      auto it = lower\_bound(free\_blocks.begin(), free\_blocks.end(), new\_block);

      if (it != free\_blocks.end() && start + size == it->start) {

        new\_block.size += it->size;

        free\_blocks.erase(it);

      }

      if (it != free\_blocks.begin()) {

        auto prev = std::prev(it);

        if (prev->start + prev->size == new\_block.start) {

          new\_block.start = prev->start;

          new\_block.size += prev->size;

          free\_blocks.erase(prev);

        }

      }

      free\_blocks.insert(lower\_bound(free\_blocks.begin(), free\_blocks.end(), new\_block), new\_block);

    }

  }

  for (int x : result) {

    cout << x << endl;

  }

  return 0;

}

# Задача L. Минимум на отрезке

В двунаправленной очереди q храним индексы элементов массива consistency, которые могут быть минимумом в текущем и следующих окнах, при этом минимум всегда находится в начале. Каждый элемент массива один раз вставляется в очередь и один раз удаляется из неё. Получаем сложность **O(n)**. Вектор consistency хранит входной массив, следовательно O(n), а очередь максимум k индексов одновременно, то есть O(k). Общая память: **O(n)**.

Код:

#include <deque>

#include <iostream>

#include <vector>

using namespace std;

int main() {

  int n, k;

  cin >> n >> k;

  vector<int> consistency(n);

  for (int i = 0; i < n; i++) {

    cin >> consistency[i];

  }

  deque<int> q;

  vector<int> result;

  for (int i = 0; i < n; i++) {

    while (!q.empty() && q.front() <= i - k) {

      q.pop\_front();

    }

    while (!q.empty() && consistency[q.back()] >= consistency[i]) {

      q.pop\_back();

    }

    q.push\_back(i);

    if (i >= k - 1) {

      result.push\_back(consistency[q.front()]);

    }

  }

  for (int i : result) {

    cout << i << " ";

  }

  cout << endl;

  return 0;

}