Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №3

по «Алгоритмам и структурам данных»

Базовые задачи

Выполнила:

Студентка группы P3215

Павличенко Софья Алексеевна

Преподаватели:

Косяков М.С.

Тараканов Д.С.

Санкт-Петербург

2025

Задача I «Машинки»

Код:

```cpp
#include <iostream>
#include <set>
#include <unordered_map>
#include <vector>

using namespace std;

int main() {
  int n, p, car, r = 0;
  size_t k;
  cin >> n >> k >> p;
  unordered_map<int, set<int>> car_using;
  vector<int> requests(p);
  set<pair<int, int>> on_floor;
  for (int i = 0; i < p; ++i) {
    cin >> requests[i];
    car_using[requests[i]].insert(i);
  }
  for (int i = 0; i < p; ++i) {
    car = requests[i];
    if (on_floor.count({i, car}) == 1) {
      on_floor.erase({i, car});
    } else
      r++;
    if (on_floor.size() == k)
      on_floor.erase(--on_floor.end());
    car_using[car].erase(i);
    if (!car_using[car].empty())
      on_floor.insert({*car_using[car].begin(), car});
    else
      on_floor.insert({1e7, car});
  }
  cout << r;
  return 0;
}

```

Пояснение к примененному алгоритму:

Алгоритм использует жадный подход с приоритетной заменой машинок, основанный на будущих запросах. Он отслеживает, какие машинки уже на полу (on_floor — множество {следующее использование, машинка}) и заменяет ту, которая понадобится позже всех.

Если машинка уже на полу, она просто убирается и добавляется с обновлённым приоритетом. Если её нет на полу, выполняется замена: удаляется машинка с самым поздним будущим использованием или которая больше не нужна.

Алгоритм работает за **O(PlogK)**:

> set позволяет эффективно искать и удалять машинку с максимальным приоритетом за **O(log K)**.
>
> unordered_map поддерживает быстрый доступ к будущим использованиям **O(1)**.

Память — **O(N+P)**, так как храним все запросы и предстоящие индексы использования машинок.

Задача J «Гоблины и очереди»

Код:

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

using namespace std;

int main() {
  int n;
  cin >> n;
  deque<int> first_half;
  deque<int> second_half;
  for (int i = 0; i < n; ++i) {
    char op;
    int id;
    cin >> op;
    if (op == '+') {
      cin >> id;
      second_half.push_back(id);
    } else if (op == '*') {
      cin >> id;
      for (size_t j = 0; j < first_half.size() - (first_half.size() + second_half.size() + 1) / 2;
           ++j) {
        first_half.push_back(second_half.front());
        second_half.pop_front();
      }
      second_half.push_front(id);
    } else {
      if (first_half.empty()) {
        cout << second_half.front() << endl;
        second_half.pop_front();
      } else {
        cout << first_half.front() << endl;
        first_half.pop_front();
      }
    }
  }
  return 0;
}

```

Пояснение к примененному алгоритму:

Алгоритм использует два двусторонних дека для эффективного управления очередью гоблинов. Обычные гоблины встают в конец очереди, а привилегированные — в середину, что достигается перемещением элементов между двумя дека. Запросы на удаление гоблина обрабатываются с помощью выборки из передней части одного из дека. Таким образом, поддерживается правильный порядок очереди с минимальными затратами времени для обычных запросов и с учетом особенностей вставки привилегированных гоблинов.

Операции + и - выполняются за **O(1)**, а операция * может требовать перемещения элементов между дека, что в худшем случае дает **O(N)** время для одного запроса, что приводит к общей сложности **O(N^2)** для всех запросов.

Память — **O(N)**, так как используется два дека для хранения всех гоблинов, где N — количество запросов.

Задача K «Менеджер памяти-1»

Код:

```cpp
#include <algorithm>
#include <iostream>
#include <queue>
#include <set>
#include <unordered_map>

using ll = long long;
using namespace std;

struct comp {
  bool operator()(const pair<ll, ll>& a, const pair<ll, ll>& b) const {
    return a.first < b.first;
  }
};

int main() {
  ll n, m;
  int t;
  cin >> n >> m;
  set<pair<ll, ll>, comp> free = {
      {1, n}
  };
  unordered_map<int, pair<ll, ll>> occupied;
  priority_queue<pair<ll, pair<ll, ll>>> max_heap;
  max_heap.push({
      n, {1, n}
  });

  for (int i = 1; i <= m; ++i) {
    cin >> t;
    if (t > 0) {
      while (!max_heap.empty()) {
        auto [length, seg] = max_heap.top();
        if (free.find(seg) != free.end())
          break;
        max_heap.pop();
      }

      if (max_heap.empty() || max_heap.top().first < t) {
        cout << -1 << endl;
      } else {
        auto [length, seg] = max_heap.top();
        max_heap.pop();
        free.erase(seg);
        cout << seg.first << endl;
        occupied[i] = {seg.first, seg.first + t - 1};

        if (seg.first + t <= seg.second) {
          free.insert({seg.first + t, seg.second});
          max_heap.push({
              seg.second - (seg.first + t) + 1, {seg.first + t, seg.second}
          });
        }
      }
    } else {
      t = -t;
      if (occupied.count(t)) {
        pair<ll, ll> seg = occupied[t];
        occupied.erase(t);

        auto next = free.lower_bound({seg.second + 1, seg.second + 1});
        auto prev = free.end();
        if (next != free.begin())
          prev = std::prev(next);

        if (prev != free.end() && prev->second + 1 == seg.first) {
          seg.first = prev->first;
          free.erase(prev);
        }
        if (next != free.end() && next->first == seg.second + 1) {
          seg.second = next->second;
          free.erase(next);
        }

        free.insert(seg);
        max_heap.push({seg.second - seg.first + 1, seg});
      }
    }
  }
  return 0;
}

```

Пояснение к примененному алгоритму:

Этот алгоритм решает задачу менеджера памяти, эффективно управляя запросами на выделение и освобождение памяти с использованием следующих структур данных:

1. Приоритетная очередь (max_heap):
    - Хранит свободные блоки памяти, при этом блоки с большей длиной имеют более высокий приоритет.
    - Используется для быстрого поиска самого большого подходящего блока памяти для запроса на выделение.
2. Множество (free):
    - Содержит все свободные блоки памяти в виде диапазонов.
    - Помогает быстро проверять состояние блоков (заняты или свободны).
3. Хеш-таблица (occupied):
    - Сохраняет информацию о занятой памяти, сопоставляя запрос с диапазоном памяти, который был выделен.

Алгоритм:

- Запрос на выделение**:**
    - Находит самый подходящий свободный блок с помощью приоритетной очереди.
    - Если блок подходит, он выделяется, остаток блока возвращается обратно в очередь.
    - Если подходящий блок не найден, возвращается -1.
- Запрос на освобождение**:**
    - Освобождает ранее выделенную память, обновляя свободные блоки.
    - Проверяется, можно ли объединить соседние блоки для уменьшения фрагментации.

Эффективность:

- Время: Операции с приоритетной очередью и множеством занимают **O(log N)**. Для M запросов общее время — **O(M log N)**.
- Память: **O(N)** для хранения информации о блоках и запросах.

Задача L «Минимум на отрезке»

Код:

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

using namespace std;

int main() {
  int n, k, i;
  cin >> n >> k;
  vector<int> a(n);
  for (i = 0; i < n; ++i)
    cin >> a[i];
  deque<int> mins;
  for (i = 0; i < k; ++i) {
    while (!mins.empty() && mins.back() > a[i]) {
      mins.pop_back();
    }
    mins.push_back(a[i]);
  }
  for (; i < n; ++i) {
    cout << mins.front() << " ";
    if (mins.front() == a[i - k])
      mins.pop_front();
    while (!mins.empty() && mins.back() > a[i]) {
      mins.pop_back();
    }
    mins.push_back(a[i]);
  }
  cout << mins.front() << " ";
  return 0;
}

```

Пояснение к примененному алгоритму:

Алгоритм использует deque для хранения индексов минимальных элементов в окне. Для каждого нового элемента удаляются большие элементы из конца deque, а устаревшие — из начала, если они выходят за пределы окна. Минимум всегда в начале deque.

Алгоритм работает за **O(N)**, так как каждый элемент добавляется и удаляется из deque не более одного раза.

Память — **O(K)** для хранения индексов в deque.