Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №3

по «Алгоритмам и структурам данных»

Базовые задачи (блок 3)

Выполнила:

Студентка группы P3230

Вавилина Е. А.

Преподаватели:

Косяков М.С.

Тараканов Д.С.

Санкт-Петербург

2025

Задача №1 «I. Машинки»

Пояснение к примененному алгоритму:
Пробежимся по всему списку машинок и будем по очереди снимать их на пол по следующим правилам (при этом будем хранить список всех машинок на полу):

1. Если надо снять машинку, которая уже на полу – ничего не делаем.
2. Если машинки еще нет на полу и там еще есть место – просто достанем машинку. +1 действие
3. Если машинка есть на полу, но места нет – выберем машинку, которая встретится позже всех. Для этого заранее для каждой машинки заполним очередь ее появлений в последовательности. При каждом новом добавлении машинки независимо от условий удалим первый элемент из очереди и добавим машинку на пол с указанием номера следующего ее запроса. Из всех машинок, представленных на полу, машинка с наибольшим номером следующего запроса убирается на полку. (если больше машинка не встретится – ставим номер следующей встречи равный числу машинок при счете от 0 до n). + 1 действие

Оценки:

1. Время: **O(p · log k)**

   Заполнение вектора paths занимает O(p). Основной цикл выполняется p раз. На каждой итерации операции с std::set (вставка, удаление, поиск) занимают O(log k), так как в худшем случае в множестве хранится до k элементов.

2. Память: **O(p+k)**

   Вектор cars занимает O(p). Вектор paths содержит n очередей, суммарно хранящих p элементов (каждый индекс из cars попадает в одну из очередей). Множество current_cars хранит до k элементов. Остальное - константы

Код:

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

int main() {
  int n = 0;
  int k = 0;
  int p = 0;
  int operations = 0;
  std::cin >> n >> k >> p;

  std::vector<int> cars(p);
  std::vector<std::queue<int>> paths(n);
```

```
  for (int i = 0; i < static_cast<int>(cars.size()); i++) {
    std::cin >> cars[i];
    paths[cars[i] - 1].push(i);
  }

  std::set<std::pair<int, int>> current_cars;
  for (const int car : cars) {
    const int last_use = paths[car - 1].front();
    paths[car - 1].pop();
    const int next_use = paths[car - 1].empty() ? p : paths[car - 1].front();

    if (static_cast<int>(current_cars.size()) < k && current_cars.count({last_use,
car}) == 0) {
      current_cars.insert({next_use, car});
      operations += 1;
      continue;
    }

    if (current_cars.count({last_use, car}) != 0) {
      current_cars.erase({last_use, car});
      current_cars.insert({next_use, car});
      continue;
    }

    if (static_cast<int>(current_cars.size()) == k && current_cars.count({last_use,
car}) == 0) {
      auto to_remove = --current_cars.end();
      current_cars.erase(to_remove);
      current_cars.insert({next_use, car});
      operations += 1;
    }
  }

  std::cout << operations;
}
```

Задача №2 «J. Гоблины и очереди»

Пояснение к примененному алгоритму:

Для заполнения очереди гоблинов разделим всю очередь на 2 двунаправленные части: первая половина и 2 половина. При этом если в очереди нечетное число элементов – в первой половине будет на 1 элемент больше. Далее действуем в зависимости от следующего действия:

1. Если удаляем машинку – убираем первый элемент из 1 очереди. Записываем номер гоблина, который ушел из очереди.

2. Если в первой половине стало меньше гоблинов чем во 2 – переносим первого гоблина 2 очереди в коней первой очереди.
3. Добавляем гоблина в конец очереди. Если во 2 очереди стало больше элементов, чем в первой – переносим первого из 2 очереди в конец первой очереди.
4. Если гоблин встает в середину очереди – ставим его в конец первой очереди. Если после этого в первой очереди стало элементов на 2 больше, чем во 2 – переносим последнего гоблина из первой очереди в начало второй.

Оценки:

1. Время: **O(n)**

   Все операции с std::deque (добавление/удаление элементов с концов) выполняются за O(1). Цикл обработки n команд выполняется за O(n), так как каждая итерация содержит константное число операций. Уравнивание середины очереди между first_half и second_half также выполняется за O(1), так как требует только переброски элементов между деками.

2. Память: **O(n)**

   Деки first_half и second_half вместе хранят не более n элементов (по одному на каждую операцию добавления * или +). Вектор result может содержать до n элементов (если все команды -). Остальные переменные занимают константную память.

Код:

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

int main() {
  int n = 0;
  std::cin >> n;

  std::deque<int> first_half;
  std::deque<int> second_half;

  std::vector<int> result;

  for (int i = 0; i < n; i++) {
    std::string curr_str;
    std::cin >> curr_str;
    int curr_numb = 0;

    if (curr_str[0] == '-') {
      result.push_back(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 (curr_str[0] == '*') {
      std::cin >> curr_numb;

      if (first_half.size() > second_half.size()) {
        second_half.push_front(curr_numb);
      } else {
        first_half.push_back(curr_numb);
      }
    }

    if (curr_str[0] == '+') {
      std::cin >> curr_numb;
      second_half.push_back(curr_numb);

      if (first_half.size() < second_half.size()) {
        first_half.push_back(second_half.front());
        second_half.pop_front();
      }
    }
  }

  for (const int res : result) {
    std::cout << res << '\n';
  }
}
```

Задача №3 «K. Менеджер памяти-1»

Пояснение к примененному алгоритму:

Будем хранить все блоки по следующему принципу: начало блока, размер и статус (1 – занят, 0 – свободен). Отдельно сделаем еще список пустых блоков в формате размер блока + указатель на блок. Для каждой команды будем еще хранить указатель на блок, который она заняла.

При получении новой команды:

1. Если команда добавляет блок – ищем в списке свободных блоков такой, в который поместится наш объем занятой памяти. Возьмем минимальный из всех подходящих блоков. Заменим этот блок на 2: занятый нужного размера (пойдет в список блоков) и свободный остаток, если такой остался (пойдет в список блоков и в список свободных блоков). Добавим команду в список команд со ссылкой на новый занятый блок. И в результат добавим индекс занятого блока.

Если мы не нашли подходящего свободного блока, то добавим команду в список команд без указателя на блок. К результату добавим -1.

2. Если дана команда на удаление – найдем в списке команд указатель на нужный блок. Пометим его как свободный. Посмотрим блоки слева и справа от него (если есть). Если эти блоки свободны – объединим их с текущим блоком и заменим из все на 1 большой объединенный блок. Все 3 свободных блока в списке свободных блоков так же заменим на 1 большой объединенный.

Оценки:

1. Время: **O(m·m)**

   Перебираем m команд.

   Операции с free_blocks: lower_bound, вставка, удаление — O(log m) (в худшем случае m/2 свободных блоков)

   Обработка освобождения памяти: Поиск в commands (unordered_map) — O(1). Объединение с соседними блоками требует поиска элементов в free_blocks через equal_range — O(log m) (в худшем случае, т.к. может быть до m/2 блоков). Удаления элементов из free_blocks — O(1) на элемент.

   Обработка выделения памяти: Поиск подходящего блока через lower_bound — O(log m). Разделение блока и вставка новых элементов — O(1).

2. Память: **O(m·n)**

   blocks хранит до O(m) элементов (каждое выделение/освобождение может создавать новые блоки). commands - O(m) (по одной записи на каждую команду выделения). free_blocks - O(m) (хранит свободные блоки). result - O(m) (результаты для вывода).

Код:

```
#include <cstdlib>
#include <iostream>
#include <iterator>
#include <list>
#include <map>
#include <tuple>
#include <unordered_map>
#include <vector>

int main() {
  int n = 0;
  int m = 0;
  std::cin >> n >> m;
  std::cin.ignore();

  using Block = std::tuple<int, int, int>;
```

```
  std::list<Block> blocks;
  std::unordered_map<int, std::list<Block>::iterator> commands;
  std::multimap<int, std::list<Block>::iterator> free_blocks;
  std::vector<int> result;

  blocks.emplace_back(1, n, 0);
  free_blocks.emplace(n, prev(blocks.end()));

  for (int i = 1; i <= m; i++) {
    int value = 0;
    std::cin >> value;

    if (value < 0) {
      value = std::abs(value);

      if (commands[value] == blocks.end()) {
        continue;
      }

      auto block_it = commands[value];
      auto [start, size, status] = *block_it;

      auto next_it = std::next(block_it);

      if (block_it != blocks.begin()) {
        auto prev_it = std::prev(block_it);
        auto [prev_start, prev_size, prev_status] = *prev_it;
        if (prev_status == 0) {
          start = prev_start;
          size = prev_size + size;

          *block_it = {start, size, 0};

          auto same_size = free_blocks.equal_range(prev_size);
          for (auto it_map = same_size.first; it_map != same_size.second; it_map++)
{
            if (it_map->second == prev_it) {
              free_blocks.erase(it_map);
              break;
            }
          }

          blocks.erase(prev_it);
        }
      }

      if (next_it != blocks.end()) {
        auto [next_start, next_size, next_status] = *next_it;
        if (next_status == 0) {
```

```
          size = next_size + size;

          *block_it = {start, size, 0};

          auto same_size = free_blocks.equal_range(next_size);
          for (auto it_map = same_size.first; it_map != same_size.second; it_map++)
{
            if (it_map->second == next_it) {
              free_blocks.erase(it_map);
              break;
            }
          }

          blocks.erase(next_it);
        }
      }

      *block_it = {start, size, 0};
      free_blocks.emplace(size, block_it);

    } else {
      auto it = free_blocks.lower_bound(value);

      if (it == free_blocks.end()) {
        commands.emplace(i, blocks.end());
        result.push_back(-1);
        continue;
      }

      auto block_it = it->second;
      auto [start, size, status] = *block_it;
      free_blocks.erase(it);

      const int new_start = start + value;
      const int new_size = size - value;

      *block_it = {start, value, 1};
      commands.emplace(i, block_it);

      if (new_size > 0) {
        auto next_it = blocks.insert(std::next(block_it), {new_start, new_size,
0});
        free_blocks.emplace(new_size, next_it);
      }

      result.push_back(start);
    }
  }
```

```
  for (const int res : result) {
    std::cout << res << '\n';
  }
}
```

Задача №4 «L. Минимум на отрезке»

Пояснение к примененному алгоритму:

Будем хранить все уникальные элементы внутри окна. При этом для каждого элемента будем записывать, сколько раз он встречается в интервале.

При получении нового числа возьмем число, которое стояло k элементов назад, найдем его в окне и уменьшим количество повторений числа внутри интервала (т.к. мы убираем его). Если количество повторений стало равно 0 – полностью удаляем элемент. Потом добавляем новый элемент: если он уже есть в окне – увеличиваем число повторений, если нет – добавим его с числом повторений 1. Т.к. мы храним элементы в мапе – первым всегда будет самый маленький элемент (ри этом ключ – элемент, значение – число его повторений), значит после формирования каждого нового окна будем брать самый первый элемент.

Оценки:

1. Время: **O(n· log k)**

   Основной цикл выполняется n раз. Поиск в окне - O(log k).

2. Память: **O(n + k)**

   Векторы data и result хранят n элементов — O(n). std::map хранит до k уникальных элементов (в худшем случае, если все элементы в окне уникальны) — O(k).

Код:

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

int main() {
  std::map<int, int> window;

  int n = 0;
  int k = 0;
  std::cin >> n >> k;

  std::vector<int> result;
  std::vector<int> data;

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

```
    int cur_numb = 0;
    std::cin >> cur_numb;
    data.push_back(cur_numb);
  }

  for (int i = 0; i < n; i++) {
    if (i >= k) {
      result.push_back(window.cbegin()->first);

      if (window[data[i - k]] == 1) {
        window.erase(data[i - k]);
      } else {
        window[data[i - k]] -= 1;
      }
    }

    if (window.find(data[i]) == window.end()) {
      window[data[i]] = 1;
    } else {
      window[data[i]] += 1;
    }
  }

  result.push_back(window.cbegin()->first);

  for (const int res : result) {
    std::cout << res << ' ';
  }
}
```