Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №3

по «Алгоритмам и структурам данных»

Базовые задачи

Выполнил:

Студент группы P3216

Брагин Р.А.

Преподаватели:

Косяков М.С.

Тараканов Д.С.

Санкт-Петербург

2025

I. Машинки

```cpp
#include <cstdint>
#include <iostream>
#include <queue>
#include <set>
#include <unordered_map>
#include <unordered_set>
#include <vector>

using namespace std;

int calculateReplacements(size_t maxCapacity, const vector<int>&
requests) {
  unordered_map<int, queue<size_t>> carPositions;
  unordered_set<int> activeCars;
  set<pair<size_t, int>> futureCars;
  int replacements = 0;

  for (size_t i = 0; i < requests.size(); ++i) {
    carPositions[requests[i]].push(i);
  }

  for (size_t i = 0; i < requests.size(); ++i) {
    int currentCar = requests[i];
    carPositions[currentCar].pop();

    if (activeCars.count(currentCar)) {
      futureCars.erase({i, currentCar});
    } else {
```

```
      if (activeCars.size() == maxCapacity) {
        auto farthestCar = *futureCars.rbegin();
        futureCars.erase(farthestCar);
        activeCars.erase(farthestCar.second);
      }
      activeCars.insert(currentCar);
      ++replacements;
    }

    if (carPositions[currentCar].empty()) {
      futureCars.emplace(SIZE_MAX, currentCar);
    } else {
      futureCars.emplace(carPositions[currentCar].front(),
currentCar);
    }
  }

  return replacements;
}

int main() {
  size_t numCars, maxCapacity, numRequests;
  cin >> numCars >> maxCapacity >> numRequests;

  vector<int> requests(numRequests);
  for (size_t i = 0; i < numRequests; ++i) {
    cin >> requests[i];
  }
  cout << calculateReplacements(maxCapacity, requests) << endl;
  return 0;
}
```

Описание: Алгоритм определяет, сколько раз маме Пети нужно будет поменять машинку на полу, чтобы он мог играть с ними в том порядке, в котором захочет. На полу может лежать не больше K машинок одновременно. Если нужная машинка уже на полу — всё хорошо. Если нет, а места больше нет, мама убирает ту машинку, которая понадобится позже всех (или вообще больше не понадобится), и ставит нужную.

Чтобы сделать это эффективно, заранее сохраняются позиции всех будущих обращений к каждой машинке. Так можно в любой момент понять, какую из машинок стоит убрать. В процессе алгоритм следит за тем, какие машинки сейчас на полу и когда они понадобятся снова.

Сложность:

Время: O(n log n), где n — количество запросов. Основные операции — вставка и удаление в set, которые работают за логарифм.

Память: O(n), так как нужно хранить позиции всех будущих обращений и текущие машинки на полу.

J. Гоблины и очереди

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

using namespace std;

int main() {
  deque<int> firstHalf, secondHalf;

  int n, goblin;
  cin >> n;
```

```
  char operation;

  for (int i = 0; i < n; ++i) {
    cin >> operation;
    switch (operation) {
      case '+':
        cin >> goblin;
        secondHalf.push_back(goblin);
        break;
      case '-':
        cout << firstHalf.front() << endl;
        firstHalf.pop_front();
        break;
      case '*':
        cin >> goblin;
        firstHalf.push_back(goblin);
        break;
    }

    if (firstHalf.size() < secondHalf.size()) {
      firstHalf.push_back(secondHalf.front());
      secondHalf.pop_front();
    } else if (firstHalf.size() > secondHalf.size() + 1) {
      secondHalf.push_front(firstHalf.back());
      firstHalf.pop_back();
    }
  }

  return 0;
}
```

Описание: Программа отслеживает очередь гоблинов, которые приходят к шаманам. Обычные гоблины встают в конец очереди, а привилегированные — в середину. Чтобы эффективно обрабатывать такие операции, очередь разбивается на две части: первая половина (firstHalf) и вторая (secondHalf). Привилегированные гоблины добавляются в конец первой половины, а обычные — в конец второй.

После каждой операции структура ребалансируется: если вторая половина становится длиннее первой, первый элемент из второй

переносится в конец первой; если первая слишком длинная (больше второй более чем на 1), ее последний элемент уходит в начало второй.

Таким образом, середина всегда находится между этими двумя частями, и вставка в центр или конец выполняется быстро.

Сложность:

Время: O(1) на каждый запрос — все операции вставки и удаления выполняются эффективно благодаря использованию deque.

Память: O(n), так как храним все элементы очереди.

L. Минимум на отрезке

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

using namespace std;

int main() {
  int n, k;
  cin >> n >> k;
  vector<int> nums(n);

  for (int& num : nums) {
    cin >> num;
  }

  deque<int> dq;

  for (int i = 0; i < n; ++i) {
    while (!dq.empty() && dq.front() <= i - k) {
      dq.pop_front();
    }

    while (!dq.empty() && nums[dq.back()] >= nums[i]) {
      dq.pop_back();
    }

    dq.push_back(i);
```

```
    if (i >= k - 1) {
      cout << nums[dq.front()] << " ";
    }
  }

  return 0;
}
```

Описание: Задача требует нахождения минимума в каждом окне длины K в последовательности из N чисел. Окно скользит по последовательности, начиная с первых K чисел и двигаясь дальше до конца. Для каждого положения окна нужно вывести минимальное число внутри этого окна.

Для эффективного нахождения минимума используется дека. В деке храним индексы чисел, которые могут быть минимальными для текущего окна. Чтобы поддерживать только потенциально минимальные числа, из деки удаляются те элементы, которые больше текущего числа, так как они больше не могут быть минимумами для следующих окон. Также из деки удаляются те индексы, которые выходят за пределы текущего окна. Когда окно сдвигается, старые индексы, выходящие за пределы окна, удаляются. В каждом окне минимум будет на первом месте в деке.

Сложность:

Время: O(N), так как каждый элемент добавляется и удаляется из деки не более одного раза.

Память: O(K), так как в деке хранится не более K индексов.