Федеральное государственное автономное
образовательное учреждение высшего образования
«Научно-образовательная корпорация ИТМО»

Факультет программной инженерии и компьютерной техники
Направление подготовки 09.03.04 Программная инженерия

Отчёт по лабораторной работе №3

По дисциплине «Алгоритмы и структуры данных» (4 семестр)
Третий блок задач

Студент:
Дениченко Александр P3212
Преподаватели:
Косяков М.С.
Тараканов Д.С.

Санкт-Петербург
2024 г.

1 Задача №I «Машинки»

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

using namespace std;

struct Car {
    int type;
    int next_use_index;
    bool operator<(const Car& other) const {
    return next_use_index < other.next_use_index;
    }
};

int main() {
    int N, K, size;
    cin >> N >> K >> size;

    vector<int> queue_start(size);
    unordered_map<int, queue<int>> map_indx_cars;

    for (int i = 0; i < size; ++i) {
        cin >> queue_start[i];
        map_indx_cars[queue_start[i]].push(i);
    }

    priority_queue<Car> cars;

    unordered_map<int, int> floor;

    int counter = 0;

    for (int i = 0; i < size; ++i) {
        Car newCar;
        newCar.type = queue_start[i];
        map_indx_cars[newCar.type].pop();

        if (floor.find(newCar.type) == floor.end()) {
            if (floor.size() == K) {
            floor.erase(cars.top().type);
            cars.pop();
            }
            floor[newCar.type] = i;
            counter++;
        }

        newCar.next_use_index = map_indx_cars[newCar.type].front();
        if (map_indx_cars[newCar.type].empty()) {
            newCar.next_use_index = size + 1;
        }
        cars.push(newCar);
    }

```

```
    cout << counter;
    return 0;
}
```

Описание: В начале был план: разместить машинки на полу, если Петя запрашивает машинку, то, если она на полу, то просто продолжаем, иначе, если пол заполнен, то судя по частотам машинок мы убираем с пола ту машинку у, которой меньше частота использования, и уменьшаем в таблице частот у этой машинки счётчик частоты. Этот код упал на 10 тесте, с Time Limit, что странно, так как этот подход был неверен, и вот пример на котором он уже некорректно отрабатывал: 1 9 3 8 3 2 5 2 1 2 6 2 4 2 3 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9. Нет смысла держать цифру 9, в какой-то момент можно играть с 2кой. Следующая идея была в создании очень странной структуры, map< int, priority queue<int, vector<int>, Compare> >, смысл этой структуры был следующий: в самом начале добавляются машинки и их индексы, причём они сортируются в порядке возрастания. Но проблема опять заключалась в огромном цикле, который проходил и искал элемент, который был на полу, так выбиралась машинка, у которой следующий индекс использования был максимальным. Пришлось полностью пересмотреть подход к хранению машинок: При считывании данных о машинах в вектор, мы ещё заполняем массив, где для каждого типа машины сохраняются индексы их использования. Инициализируем приоритетную очередь, в которой машины будут упорядочены по времени следующего использования. Хэш-таблицы нужна для проверки на вхождение машинки (её наличие на полу). Если машина еще не находится на полу, и на полу уже есть K машин, удаляем из очереди машину с наибольшим временем следующего использования (приоритетная очередь сортирует по этому параметру машинки, что позволило избавиться от огромного цикла, так как машинка с наибольшим временем использования будет первой). Добавляем текущую машину на пол и в приоритетную очередь.

Сложность: Основной цикл обработки каждой машинки также имеет сложность O(N). Операции вставки и удаления из приоритетной очереди cars, а также операции вставки и удаления из хэш-таблицы в среднем имеют сложность O(log K) и O(1) соответственно, где K — максимальное количество машинок на полу. Таким образом, общая временная сложность будет O(N log K).

2 Задача №J «Гоблины и очереди»

```
#include <iostream>
#include <queue>

using namespace std;

int main()
{
    int count;
    cin >> count;

    string type;
    int goblin;
    queue<int> first;
    deque<int> second;
    for (int i = 0; i < count; i++)
    {
        cin >> type;
        if (type == "-")
        {
            if (!first.empty())
            {
                cout << first.front() << endl;
                first.pop();
            }
        }
        else
        {
            cin >> goblin;
```

```
            if (type == "+")
            {
                second.push_back(goblin);
            }
            else
            {
                second.push_front(goblin);
            }
        }
        while (first.size() > second.size() + 1)
        {
            second.push_front(first.back());
            first.pop();
        }
        while (second.size() > first.size())
        {
            first.push(second.front());
            second.pop_front();
        }
    }
    return 0;
}
```

Описание: Данный алгоритм управляет очередью гоблинов, желающих посетить шаманов. Он использует две структуры данных: стандартную очередь (queue<int>) для хранения гоблинов, ожидающих своей очереди, и двустороннюю очередь (deque<int>) для более гибкого управления порядком гоблинов, в частности, для вставки элементов в середину. В начале идея разделения была такая-же, но я хотел вручную предусмотреть все случаи для баланса очереди, но код упал на 7 тесте, поэтому было принято решение каждый раз просто балансировать очередь двумя циклами, так как предугадать всё будет долго. Для каждой операции считывается её тип (type). Если type равен -, извлекается элемент из начала очереди first и выводится его значение. Если type равен +, считывается значение гоблина и добавляется в конец deque (second). Если type является любой другой строкой (подразумевается, что это * по условию задачи, хотя это не явно проверяется в коде), считывается значение гоблина и добавляется в начало deque (second). После каждой операции добавления производится проверка размеров двух коллекций. Если размер first больше размера second более чем на один элемент, элементы перемещаются из first в second, чтобы уравнять размеры. Аналогично, если размер second превышает размер first, элементы перемещаются из second в first

Сложность: Сложность алгоритма составляет O(n). Балансировка очереди, хоть и сделана в тупую, но она не будет делать слишком много действий, скорее на каждом ходе от 1 до 2х действий.

3 Задача №K «Менеджер памяти-1»

```
#include <iostream>
#include <vector>
#include <map>
using namespace std;

int main() {
    int size, count;
    cin >> size >> count;

    map<int, int> free_segments_index;
    multimap<int, int> free_segments_size;

    free_segments_index.emplace(make_pair(1, size));
    free_segments_size.emplace(make_pair(size, 1));

    vector<pair<int, int> > history(count);
```

```
17
18      for (int i = 0; i < count; i++) {
19          int req;
20          cin >> req;
21          if (req > 0) {
22              auto minimax = free_segments_size.lower_bound(req);
23              if(minimax!= free_segments_size.end()){
24                  cout << minimax->second << endl;
25                  history[i] = make_pair(minimax->second, req);
26                  if(minimax->first - req > 0){
27                      free_segments_index.emplace(make_pair(minimax->second + req, minimax->fir
28                      free_segments_size.emplace(make_pair(minimax->first - req, minimax->secon
29                  }
30                  free_segments_index.erase(minimax->second);
31                  free_segments_size.erase(minimax);
32              }else{
33                  history[i] = make_pair(-1, req);
34                  cout << "-1" << endl;
35              }
36          } else {
37              req = -req;
38              int start = history[req - 1].first;
39              int length = history[req - 1].second;
40              auto it = free_segments_index.lower_bound(start);
41              if(start == -1){
42                  continue;
43              }
44              if (it != free_segments_index.begin()) {
45                  auto prev_it = prev(it);
46                  if (prev_it->first + prev_it->second == start) {
47                      length += prev_it->second;
48                      start = prev_it->first;
49                      auto tmp = free_segments_size.find(prev_it->second);
50                      while(tmp->second != prev_it ->first){
51                          tmp++;
52                      }
53                      free_segments_size.erase(tmp);
54                      free_segments_index.erase(prev_it);
55                  }
56              }
57
58              if (it != free_segments_index.end() && start + length == it->first) {
59                  length += it->second;
60                  auto tmp = free_segments_size.find(it->second);
61                  while(tmp->second != it -> first){
62                      tmp++;
63                  }
64                  free_segments_size.erase(tmp);
65                  free_segments_index.erase(it);
66              }
67
68              free_segments_index[start] = length;
69              free_segments_size.emplace(make_pair(length, start));
70          }
71      }
72      return 0;
```

```
}
```

Описание: В начале мне было даже сложно представить как реализовывать задачу, но через некоторое время, при помощи разбора на листочке. Но алгоритм вообще не хотел работать, так как алгоритм ещё делал лишнюю работу - выбирал максмально подходящий кусочек памяти по размеру, но по факту этого не требовалось. Далее я долго тупил с использованием insert и пытался, как в дальнейшем оказалось, сделать из него emplace. На ноуте установлен компилятор, который не поддерживает emplace, поэтому через некоторое время дошло потестить emplace на компе и о чудо у меня получилось. Алгоритм заключается в просмотре доступной памяти на каждом шаге и выборе кусочка памяти (теперь повторюсь, что он не обязательно максмально подходящий)

Сложность: Учитывая, что каждая операция с map и multimap имеет логарифмическую сложность, и каждый запрос обрабатывается за константное количество таких операций (за исключением возможного объединения сегментов, которое может потребовать несколько дополнительных операций вставки и удаления), общая сложность алгоритма за время выполнения всех запросов составляет O(count * log size), где count — количество запросов, а size — изначальный размер памяти или максимальное количество элементов в контейнерах.

4 Задача №L «Минимум на отрезке»

```
#include <iostream>
#include <vector>
#include <map>
#include <queue>
using namespace std;

struct Value {
    int index;
    int obj;
    bool operator<(const Value& other) const {
        return obj > other.obj;
    }
};

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    int size, count, x;
    cin >> count >> size;


    priority_queue<Value> values;

    for (int i = 0; i < count; i++)
    {
        cin >> x;
        Value new_value;
        new_value.index = i;
        new_value.obj = x;
        values.push(new_value);
        if(i>=size-1){
            while(true){
                if(values.top().index >= i-(size-1)){
                    cout << values.top().obj << "␣";
                    break;
                }else if(values.empty()){
                    break;
                }else{
```

```
                    values.pop();
                }
            }

        }
    }
    return 0;
}
```

Описание: Алгоритм лёгкий и сразу пришёл в голову. Есть то самое окно, которое двигается по, так скажем, отрезку из чисел. На каждом шаге я храню самое минимальное число среди старых + текущее. Так же на каждом шаге предусмотрена очистка очереди от слишком старых значений.

Сложность: Общая временная сложность алгоритма будет O(count * log size), так как каждая операция с каждым элементом в худшем случае требует O(log size) времени.