Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №3

по «Алгоритмам и структурам данных»

Базовые задачи

Выполнил:

Студент группы P3233
Богатов Александр Сергеевич

Преподаватели:

Косяков М.С.

Тараканов Д.С.

Санкт-Петербург

2022

Задача I: Машинки

```
#include <iostream>
#include <list>
#include <algorithm>
#include <unordered_set>
#include <queue>

using namespace std;

int main()
{

    int n, k, p;

    int ops = 0;

    cin >> n >> k >> p;

    int cars_order[p];

    unordered_set<int> on_floor;

    list<int> entry_positions[n];

    priority_queue<pair<int, int>> order;

    for (int i = 0; i < p; i++) {
        cin >> cars_order[i];
        entry_positions[--cars_order[i]].push_back(i);
    }

    for (int i = 0; i < p; i++) {
        int current = cars_order[i];
        entry_positions[current].pop_front();
        if(on_floor.find(current) == on_floor.end()) {
            if (on_floor.size() + 1 > k) {
                on_floor.erase(order.top().second);
                order.pop();
            }
            on_floor.insert(current);
            ops++;
        }
        if (!entry_positions[current].empty())
            order.push({entry_positions[current].front(), current});
        else order.push({500001,current});
    }

    cout << ops;

    return 0;
}
```

Пояснение к примененному алгоритму:

Когда мы достигаем лимита машинок на полу, выгоднее всего убрать машинку, которая встречается снова в последовательности ближе всего к концу. Заведем массив списков мест в

последовательности, на которых встречается каждая машинка. При каждой встрече с этой машинкой, снимаем элемент из соответствующего списка. Если необходимая машинка уже на полу – ничего не делаем. Иначе, если лимит не достигнут - спускаем машинку; если лимит достигнут – убираем с пола верхний элемент приоритетной очереди. Приоритетная очередь пополняется в конце каждого прохода цикла, в нее записывается пара ближайшей позиции машинки и номера этой машинки. Так, верхний элемент очереди всегда будет той самой машинкой, которую нужно убрать с пола. В самом приоритете – машинки, которые больше не понадобятся в будущем. В среднем сложность алгоритма – O(N*const), однако учитывая худшие случаи применения методов unordered_set – O(N^2).

Задача J: Гоблины в очереди

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

using namespace std;

int main()
{
    deque<string> goblins_one;
    deque<string> goblins_two;

    int n;

    cin >> n;

    vector<string> solution;

    for (int i = 0; i < n; i++) {
        string current;
        cin >> current;
        if (current == "+") {
            cin >> current;
            goblins_two.push_front(current);
        }
        else if (current == "*") {
            cin >> current;
            goblins_two.push_back(current);
        } else {
            solution.push_back(goblins_one.back());
            goblins_one.pop_back();
        }

        if (goblins_one.size() < goblins_two.size()) {
            goblins_one.push_front(goblins_two.back());
            goblins_two.pop_back();
        }
    }

    for (int i = 0 ; i < solution.size(); i++)
        cout << solution.at(i) << endl;

    return 0;
}
```

Пояснение к примененному алгоритму:

Операция вставки элемента в середину двусторонней очереди крайне неоптимальна, поэтому заведем две очереди. Если мы встречаемся с новым непривилегированным гоблином – отправляем его в конец второй очереди. Если с привилегированным – в начало. Если пропускаем гоблина – снимаем с начала первой очереди элемент. По окончанию каждой итерации проверяем число гоблинов в очередях – если в первой их меньше, то первого гоблина из второй очереди передвигаем в первую очередь. Сложность – O(N).

Задача K: Менеджер памяти – 1

```
#include <iostream>
#include <map>
#include <algorithm>
#include <bits/stdc++.h>

using namespace std;

int len = 0;

struct Block_List {

    int idx;
    int start;
    int size;
    int finish;
    bool is_free;
    Block_List* previous;
    Block_List* next;

    Block_List(int idx, int start, int finish, bool is_free,
Block_List* previous, Block_List* next) {
              this->idx = idx;
              this->start = start;
              this->finish = finish;
              this->is_free = is_free;
              this->previous = previous;
              this->next = next;
              if (previous) {
                     previous->next = this;
              }
              if (next) {
                     next->previous = this;
              }
       }

    void rid() {
        if (previous)
            previous->next = next;
        if (next)
            next->previous = previous;
    }
};

struct Block_List* blocks[100000];
struct Block_List* requested[100000];

int get_block_size(Block_List* block) {
```

```
    return 1 + block->finish - block->start;
}

void swap(int i, int j) {
    Block_List* tmp = blocks[i];
    blocks[i] = blocks[j];
    blocks[j] = tmp;
    blocks[i]->idx = i;
    blocks[j]->idx = j;
}

int parent(int i) {
    return (i - 1) / 2;
}

int left(int i) {
    return 2 * i + 1;
}

int right(int i) {
    return 2 * i + 2;
}

void siftup(int i) {
    while (i > 0 && get_block_size(blocks[parent(i)]) <
get_block_size(blocks[i])) {
        swap(i, parent(i));
        i = parent(i);
    }
}

void insert(Block_List* block) {
    block -> idx = len;
    blocks[len] = block;
    siftup(len++);
}

void siftdown(int i) {
    while (true) {
        int j = i;
        if (left(j) < len && get_block_size(blocks[i]) <
get_block_size(blocks[left(j)])) {
            i = left(j);
        }
        if (right(j) < len && get_block_size(blocks[i]) <
get_block_size(blocks[right(j)])) {
            i = right(j);
        }
        if (i != j) {
            swap(i, j);
        } else break;
    }
}

void extract(int i) {
    swap(i, --len);
    siftup(i);
    siftdown(i);
```

```
}

int main()
{
    int n, m;

    cin >> n >> m;

    struct Block_List* base = new Block_List(len++, 1, n, true, NULL,
NULL);

    blocks[0] = base;

    for (int i = 0; i < m; i++) {
        int current;
        cin >> current;
        if (current > 0) {
            Block_List* maximum = blocks[0];
            if (!len || get_block_size(maximum) < current) {
                cout << -1 << endl;
            } else {
                cout << maximum->start << endl;
                requested[i] = new Block_List(-1, maximum->start,
maximum->start + current - 1, false, maximum->previous, maximum);
                maximum->start += current;
                if (get_block_size(maximum)) {
                    siftdown(maximum->idx);
                } else {
                    maximum->rid();
                    swap(0, --len);
                    siftdown(0);
                }
            }
        } else {
            Block_List* free_req = requested[-current-1];
            if (free_req) {
                if (free_req->previous && free_req->next && free_req-
>previous->is_free && free_req->next->is_free) {
                    free_req->previous->finish = free_req->next-
>finish;
                    siftup(free_req->previous->idx);
                    free_req->rid();
                    extract(free_req->next->idx);
                    free_req->next->rid();
                }
                else if (free_req->previous && free_req->previous-
>is_free) {
                    free_req->previous->finish = free_req->finish;
                    siftup(free_req->previous->idx);
                    free_req->rid();
                }
                else if (free_req->next && free_req->next->is_free) {
                    free_req->next->start = free_req->start;
                    siftup(free_req->next->idx);
                    free_req->rid();
                } else {
                    free_req->is_free = true;
                    insert(free_req);
```

```
                }
            }
        }
    }

    return 0;
}
```

Пояснение к примененному алгоритму:

:c.

Память должна выделяться последовательно по условию. Свободные отрезки не могут находиться перед занятыми. Для хранения блоков памяти будем использовать кучу (бинарную пирамиду). Когда мы встретились с запросом на выделение памяти, необходимо рассмотреть самый длинный свободный отрезок, который будет находиться на вершине кучи. Если этого отрезка недостаточно, выводим -1, иначе мы должны занять часть блока в соответствии с запросом и сделать соответствующие преобразования в пирамиде. При освобождении памяти, нужно рассмотреть несколько случаев. Если в пирамиде оба потомка освобождаемого отрезка заняты, то мы должны освободить этот блок и расположить его в куче в подходящем месте. Если следующий после освобождаемого блок занят, то мы удлиним этот свободный блок, а от старого избавимся. Аналогично действуем, если занят только предыдущий блок. Если же заняты оба, то мы удлиним самый ранний по порядку блок, избавляясь от двух других. Т.к. в основном цикле мы пользуемся операциями пирамиды, то сложность алгоритма составит O(N*logN).

Задача L: минимум на отрезке

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

using namespace std;

int main()
{
    int n,k;

    cin >> n >> k;

    vector<int> nums;

    for (int i = 0 ; i < n; i++) {
        int current;
        cin >> current;
        nums.push_back(current);
    }

    int min_idx;
    int min = 100001;

    deque<int> window;
    vector<int> solution;

    for (int i = 0; i < n; i++) {
        if (window.size() > 0 && window.front() <= i - k)
            window.pop_front();
```

```
        while (window.size() > 0 && nums.at(window.back()) >=
nums.at(i))
            window.pop_back();
        window.push_back(i);
        if (i >= k - 1) {
            solution.push_back(nums.at(window.front()));
        }
    }

    for (int i = 0; i < solution.size(); i++)
        cout << solution.at(i) << " ";

    return 0;
}
```

Пояснение к примененному алгоритму:

Будем хранить изначальную последовательность в массиве, а работать с окном через двустороннюю очередь. В очереди храним не элементы, а их индексы. Если окно не пустое, а передний элемент устарел, мы его удаляем. Далее, пока элементы последовательности, начиная с последнего в очереди, больше чем i-ый элемент, мы убираем их индексы из окна. Таким образом, передний элемент очереди – индекс минимума на текущем отрезке. Несмотря на вложенный цикл, сложность алгоритма линейная, т.к. число проверок соответствует числу элементов, каждый элемент вставляется и удаляется ровно 1 раз.