Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №3
по дисциплине “Алгоритмы и структуры данных”
Базовые задачи

Выполнила:
Шевченко Д. П.,
группа P3230

Преподаватели:
Косяков М.С.
Тараканов Д. С.

Санкт-Петербург
2024

Задача I. Машинки

```cpp
int main() {
    int n, k, p;
    std::cin >> n >> k >> p;
    int cars[p];
    std::map<int, int> floor;
    std::vector<std::list<int>> last(n + 1);
    int result = 0;
    for (int i = 0; i < p; i++) {
        std::cin >> cars[i];
        last[cars[i]].push_front(i);
    }
    for (int i = 0; i < p; i++) {
        int car = cars[i];
        if (floor.find(last[car].back()) == floor.end()) {
            if (static_cast<int>(floor.size()) != k) {
                result++;
            } else {
                auto iterator = floor.end();
                iterator--;
                floor.erase(iterator);
                result++;
            }
        } else {
            auto delete_car = floor.find(last[car].back());
            floor.erase(delete_car);
        }
        if (last[car].size() != 1) {
            last[car].pop_back();
            floor[last[car].back()] = car;
        } else {
            last[car].pop_back();
        }
    }
    std::cout << result;

    return 0;
}
```

Пояснение:
Если машинки нет на полу и на полу еще есть место, то количество операций увеличиваем на 1, иначе удаляем с пола машинку, которая первый раз встречается позже всех, среди тех, которые на полу, и количество операций увеличиваем на 1.
Если машинка есть на полу, то меняем индекс машинки.
Если на полу есть машинка, которая больше ни разу не встретится, то первой надо будет убрать ее, а если такой нет то убираем ту, которая встречается позже всех (в первый раз)
Сложность алгоритма:
Поиск и удаление в map O(logn)- n количество строк
Тем самым учитывая, что мы это делаем в цикле сложность алгоритма равна O(n*logn) – n – количество введенных строк

Задача J. Гоблины и очереди

```cpp
int main() {
    int n;
    std::cin >> n;
    std::deque<std::string> numbers;
    std::vector<std::string> data;
    std::deque<std::string> numbers_second;
    std::string s;
    for (int i = 0; i < n; i++) {
        std::cin >> s;
        if (s == "+") {
            std::cin >> s;
            numbers_second.push_back(s);
        } else if (s == "*") {
            std::cin >> s;
            numbers_second.push_front(s);
        } else if (s == "-") {
            data.push_back(numbers.front());
            numbers.pop_front();
        }
        if (numbers_second.size() > numbers.size()) {
            numbers.push_back(numbers_second.front());
            numbers_second.pop_front();
        }
    }
    for (const auto &i: data) {
        std::cout << i << std::endl;
    }
    return 0;

}
```

Пояснение:

Если встречаем гоблина с “+”, то ставим его в конец очереди. Если с “*”, то ставим в начало. Если “-“, то убираем первого гоблина из очереди, записываем его в вектор, чтобы потом вывести то, что он вышел. Так как вставлять гоблинов в середину очереди (когда встречается “*”) у меня не получилось, решение не проходило по времени, я сделала две очереди, поэтому каждый раз нужно проверять, если размер очереди гоблинов с “+” и “*” больше, чем размер очереди numbers, то переносим первого гоблина в numbers.
Сложность:
Цикл for выполняется n раз, где n – количество введенных строк. Сложность этого цикла - O(n).
В каждой итерации цикла выполняется несколько операций:
Считывание строки, что занимает O (1)
Вставка и удаление элементов в очередях и векторе, которые имеют сложность O (1).
То есть сложность кода будет O(n)

Задача K. Менеджер памяти-1

```
std::multimap<int, int> block_of_size;
std::map<int, int> block_of_index;

void delete_elements(const std::map<int, int>::iterator &it) {
    auto size_block_pair = block_of_size.equal_range(it->second);
    for (auto x = size_block_pair.first; x != size_block_pair.second; ++x) {
        if (x->second == it->first) {
            block_of_size.erase(x);
            break;
        }
    }
    block_of_index.erase(it);
}

void add(const std::pair<int, int> &pair) {
    block_of_index.insert(pair);
    block_of_size.insert({pair.second, pair.first});
}

void erase(int s, int index) {
    block_of_size.erase(block_of_size.lower_bound(s));
    block_of_index.erase(index);
}

void print_result(const std::vector<int> &result) {
    for (int i: result) {
        std::cout << i << std::endl;
    }
}

int allocate(int s, int index) {
    index = -1;
    if (block_of_size.lower_bound(s) != block_of_size.end()) {
        int size = block_of_size.lower_bound(s)->first - s;
        index = block_of_size.lower_bound(s)->second;
        erase(s, index);
        if (size > 0) {
            block_of_index.insert({index + s, size});
            block_of_size.insert({size, index + s});
        }
    }
    return index;
}

void merge_blocks(bool merge_left, bool merge_right, int index_of_free, int
size_of_free,
                  const std::map<int, int>::iterator left, const
std::map<int, int>::iterator right) {
    if (!merge_right && !merge_left) {
        add({index_of_free, size_of_free});
    }
    if (merge_right && !merge_left) {
        delete_elements(right);
        add({index_of_free, right->second + size_of_free});
    }
    if (merge_right && merge_left) {
        delete_elements(left);
        delete_elements(right);
        add({left->first, left->second + right->second + size_of_free});
    }
    if (!merge_right && merge_left) {
```

```
        delete_elements(left);
        add({left->first, left->second + size_of_free});
    }
}

int main() {
    int n, m, s;
    int index;
    std::cin >> n >> m;
    std::vector<std::pair<int, int>> numbers(m + 1);
    add({1, n});
    std::vector<int> result;
    for (int i = 0; i < m; numbers[i + 1] = {s, index}, i++) {
        index = 0;
        std::cin >> s;
        if (s > 0) {
            index = allocate(s, index);
            result.push_back(index);
        } else {
            int index_of_free = numbers.at(-s).second;
            int size_of_free = numbers.at(-s).first;
            if (index_of_free != -1) {
                auto right = block_of_index.lower_bound(index_of_free);
                auto left = (right != block_of_index.begin()) ?
std::prev(right) : block_of_index.end();
                bool merge_left = (left != block_of_index.end() && left-
>first + left->second == index_of_free);
                bool merge_right = (right != block_of_index.end() && right-
>first == index_of_free + size_of_free);
                merge_blocks(merge_left, merge_right, index_of_free,
size_of_free, left, right);
            }
        }
    }
    print_result(result);
    return 0;
}
```

Пояснение:
Если мы получаем запрос на выделение памяти, то проверяем есть ли блок памяти размера больше или равным запрашиваемого. Если есть, то берем его первую и последнюю ячейки и удаляем из памяти, тем самым делаем занятыми и записываем индексы первой и последней занятых ячеек.
Если получили запрос на освобождение памяти, то существует ли номер запроса, который нужно освободить. Если да, то проверяем 4 ситуации:
- если от нужного отрезка только справа есть свободный блок, то освобождаем нужный блок памяти и объединяем его с блоком справа;
- если от нужного отрезка только слева есть свободный блок, то освобождаем нужный блок памяти и объединяем его с блоком слева;
-если от нужного блока есть свободные блоки и справа, и слева, то освобождаем нужный блок и объединяем все три.
- если от нужного блока ни справа, ни слева нет свободных, то просто освобождаем нужный блок.

Сложность:

1. std::map и std::multimap: Вставка и удаление элементов: O(log n), где n - количество элементов в контейнере. Поиск O(log n).

Таким образом сложность равна O(m*logm) – где m – количество запросов

Задача L. Минимум на отрезке

```cpp
int main() {
    int n, k;
    std::cin >> n >> k;
    std::vector<int> numbers(n);
    std::deque<int> deque;
    for (int i = 0; i < n; i++) {
        std::cin >> numbers[i];
    }
    for (int i = 0; i < n; i++) {
        while (!deque.empty() and numbers[deque.back()] >= numbers[i]) {
            deque.pop_back();
        }
        while (!deque.empty() and deque.front() + k <= i) {
            deque.pop_front();
        }
        deque.push_back(i);
        if (i + 1 >= k) {
            std::cout << numbers[deque.front()] << " ";
        }
    }

    return 0;
}
```

Пояснение:

Пока очередь deque не пустая и элемент в конце очереди больше или равен numbers[i], последний элемент удаляется из очереди. Я это сделала для того, чтобы очередь содержала индексы элементов в порядке возрастания соответствующих значений. Если самый старый элемент очереди (то есть первый), находится вне окна, то он удаляется, тем самым мы двигаем окно.
Индекс добавляем в очередь. Когда окно имеет размер k, то выводим минимальный элемент. Используем numbers [deque. front()], так как очередь deque отсортирована по возрастанию, соответственно индекс минимального элемента находится в начале.
Сложность: O(n) – где n – количество чисел в последовательности, так как с каждый элементом мы делаем одну операцию.