Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №3

по «Алгоритмам и структурам данных»

Базовые задачи / Timus

Выполнил:

Студент группы P3211

Болорболд А.

Преподаватели:

Косяков М.С.

Сосновцев Г.А.

Санкт-Петербург

2024

Задача №9 «Машинки»

1. **Пояснение к применённому алгоритму:**

Чтобы решать эту задачу, надо использовать некоторые свойства структур данных. Очередь операции составляется с помощью таких действий, как pop_front(), front() и top(). Число всего операции определяется тем, находится ли машинка (которым хочет играть Петя) в конце набора.

Исходный код:

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

int main() {
    const int max_p = 500000;
    int n, k, p, tact = 0;
    std::unordered_set<int> on_the_floor;
    std::priority_queue<std::pair<int, int>, std::vector<std::pair<int,
int>>, std::less<>> cars;

    std::cin >> n >> k >> p;

    int input[p];
    std::list<int> nodes[n];

    for (int i = 0; i < p; ++i) {
        std::cin >> input[i];
        nodes[--input[i]].push_back(i);
    }

    for (int i = 0; i < p; ++i) {
        nodes[input[i]].pop_front();
        if(on_the_floor.find(input[i]) == on_the_floor.end()) {
            tact++;
//            std::cout << i << " снять машинку " << input[i] + 1 << " ";
            if(on_the_floor.size() >= k) {
//                std::cout << "поднять машинку " << cars.top().second + 1;
                on_the_floor.erase(cars.top().second);
                cars.pop();
            }
            on_the_floor.insert(input[i]);
//            std::cout << std::endl;
        }
        int priority;
        if(nodes[input[i]].empty()) {
            priority = max_p;
        } else {
            priority = nodes[input[i]].front();
        }
        cars.emplace(priority, input[i]);
    }
    std::cout << tact;
    return 0;
}
```

2. **Корректность применяемого алгоритма:**

Алгоритм прост: *заместится машина, которая не будет использоваться в течение самого длительного периода времени*.

Тогда для решения задачи нам нужно для каждого шага знать, когда в следующий раз Петя захочет использовать машинку, которую он использует на этом шаге — тривиальная задача. Для шагов, после которых используемая машинка не потребуется, используем как можно большее число, которое не может представлять номер шага (потребуется в дальнейшем для определения порядка замещения).

Так, для использований машинок *[1, 2, 3, 1, 3, 1, 2]* из примера, этот массив следующих шагов использования будет:

- *[3, 6, 4, 5, INT_MAX, INT_MAX, INT_MAX]*, где *INT_MAX* — последние 3 шага, так как это последние шаги использований *3*, *1* и *2* машинок соответственно.

Очевидно, что машинка, которая не будет использоваться в течение са́мого длительного периода времени — это машинка, шаг использования которой максимально далёк от текущего шага, то есть та, следующий шаг использования которой максимален.

Теперь будем хранить текущие следующие шаги машинок в *priority_queue*, чтобы автоматически сортировать шаги и легко удалять последние элементы (максимальные элементы, «следующие» шаги, которые не будут использоваться в течении самого длительного периода времени).

Суммируя вышеприведённые утверждения, получим для шагов *[1, …, p]*:

- если очередь *содержит* текущий шаг, это значит, что Петя снова использует ту же машинку, которую задействовал на шаге, для которого текущий шаг являлся следующим, использующим ту же машинку. Значит, мы удаляем текущий шаг из очереди — замещение машинки не требуется, так как этот шаг был в очереди и не убирался из него.
- если очередь *не содержит* текущий шаг, это значит, что нужно заменить машинку по оптимальному критерию (то есть *инкрементируем счётчик* операций мамы Пети). Если в очереди меньше K шагов, значит его ещё не наполнили, иначе мы удаляем максимальный следующий шаг — последний элемент
- в конце добавляем в очередь *следующий* шаг для *текущего* шага.

Таким образом, в счётчике мы получаем минимальное количество операций, которое надо совершить маме Петя. В итоге получается корректный ответ.

3. **Оценка сложности по времени:**

Здесь по идее сложность по времени должна составлять O(n), но здесь надо учитывать некоторые нюансы, связанные с операциями структурами данных.

Рассмотрим первую структуру: неупорядоченный набор. Все операции (вставка, нахождение, размер) занимают постоянное время (O(1)). Разве что операции удаления занимают O(<число удаляемых элементов>), но в любом случае они не превосходит O(n). Та же самая история со списком.

Потом рассмотрим вторую структуру: приоритетная очередь. Если бы очередь построилась из n элементов сразу, то операция занял бы O(n). Если бы очередь построилась из n элементов по одному, то операция занял бы O(n*logn).

Надо ещё учитывать от чего именно зависит количество итераций. В этом случае P — количество необходимых итераций, а K — размер приоритетной очереди.

В нашем алгоритме наблюдается второй сценарий, поэтому в итоге сложность по времени должна составлять **O(P*logK)**.

4. **Оценка сложности по памяти:**

Здесь нет ничего особого на самом деле, все структуры занимают **O(P)** (так как размер структур данных зависит от него).

Задача №10 «Гоблины и очереди»

1. **Пояснение к применённому алгоритму:**

Это по моему мнению очень хорошая задача для понимания таких структур, как очереди. Каждому гоблину (символу) соответствует своя операция. Здесь создаются 2 очереди: для зашедших к шаманам и для ждущих. Если разница между размерами очередей равна 2, то гоблин, стоящий в самом позади первой очереди, переходит на вторую, а если разница равен -1, то происходит обратное (автобалансировка, подробнее ниже).

Исходный код:

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

int main() {

    int n;
    std::cin >> n;

    std::deque<std::string> q1;
    std::deque<std::string> q2;

    for (int i = 0; i < n; i++) {
        char action;
        std::cin >> action;
        if(action == '-') {
            std::cout << q1.front() << std::endl;
            q1.pop_front();
```

```
        } else {
            std::string number;
            std::cin >> number;
            if(action == '+') {
                q2.push_back(number);
            } else {
                q2.push_front(number);
            }
        }
        int diff = q1.size() - q2.size();
        if(diff == 2) {
            q2.push_front(q1.back());
            q1.pop_back();
        } else if(diff == -1) {
            q1.push_back(q2.front());
            q2.pop_front();
        }
    }
    return 0;
}
```

2. **Корректность применяемого алгоритма:**

Существует очевидная наивная реализация, где мы храним очередь просто как дек (очень часто происходит изменение очереди, в отличие от чтения от него) и спокойно изменяем очередь по условиям задачи при каждом запросе. Однако такая реализация не проходит по времени (TL), поэтому нам нужно чуть адаптироваться.

Для дека проанализируем имеющие методы:

- в случае push_back() {+}: *amortized* O(1);
- в случае pop_front() {–}: *amortized* O(1);
- в случае insert {*}: O(n).

Амортизация константы в push/pop_front почти не играет особой роли, так как длина очереди изменяется не слишком значительно. Однако вставка привилегированного гоблина в середину — O(n). Значит, этого и надо оптимизировать.

Для того, чтобы вставка в середину также занял константное время, «разделим» дек на две половины. Чтобы поддерживать очередь в удобном для нас состоянии (очереди равны друг другу по размеру), нам после выхода из очереди или добавления обычного гоблина нужно *сбалансировать половины* (сдвигать по необходимости): если длина очереди чётная, перемещать гоблинов из начала второй очереди в конец первой. Сложность должна составлять *amortized* O(1).

Таким образом, вне зависимости от метода, очередь поддерживается в состоянии, *при котором длина первой половины либо равна длине второй половины (значит длина обеих очередей чётная), либо превосходит длину второй на 1 (значит длина обеих очередей нечётная)*. Это помогает нам легко добавлять привилегированного гоблина со сложностью *amortized* O(1).

Благодаря такой оптимизации правильность алгоритма сохраняется, а время вставки привилегированного гоблина сократится.

3. **Оценка сложности по времени:**

Опять преследуем тому же исследованию. Здесь используется очередь с двойным концом (ʕ◔ϖ◔ʔ), где всякие операций обычно занимает *amortized* O(1). Раз только операции вставки (в этом задаче ключевая) занимает O(n), но здесь операции вставки только в концах, благодаря которому занимает amortized O(1). В итоге, сложность алгоритма по времени должна составлять **O(n)**.

4. **Оценка сложности по памяти:**

Сложность по памяти составляет **O(n)** для хранения самих гоблинов. Для операции в алгоритме, связанных с очередями, не требуется вспомогательной памяти.

Задача №11 «Менеджер памяти-I»

1. **Пояснение к применённому алгоритму:**

Что-то вроде задачи №3 «Конфигурационный файл». Здесь пошла реализация собственных функции, где я просто совместил операций двух мапов в одну. Потом я сохранил изначальное начало, размер и конец, после чего произошли операции непосредственно с самим «запросом». Не использовать мапы для решения таких задач — это то, чего я делаю делать в далёком будущем.

Исходный код:

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

std::map<int,int> blocks_ends;
std::multimap<int,int> blocks_sz_start;

void remove_by_sz(const std::map<int, int>::iterator& it){
    auto it_sizes = blocks_sz_start.find(it -> second - it -> first);
    while (it_sizes -> second != it -> first) it_sizes++;
    blocks_sz_start.erase(it_sizes);
    blocks_ends.erase(it);
}
void remove(const std::multimap<int, int>::iterator& it){
    blocks_ends.erase(it -> second);
    blocks_sz_start.erase(it);
}

void insert(const std::pair<int, int>& p){
    blocks_ends.insert(p);
    blocks_sz_start.insert({p.second - p.first, p.first});
}


int main() {

    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    std::cout.tie(nullptr);
```

```
    int n, m;
    std::cin >> n >> m;
    n++;
    std::vector<std::pair<int,int>> memory_table(m,{0,0});
    insert({1,n});
    for (int i = 0; i < m; ++i) {
        int query;
        std::cin >> query;
        if(query < 0){
            query *= -1;
            query--;
            if(memory_table[query].first < 0) continue;
            int start_orig = memory_table[query].first,
            sz_orig = memory_table[query].second,
            end_orig = start_orig + sz_orig;
            auto after_block = blocks_ends.lower_bound(end_orig),
            before_block = (after_block != blocks_ends.begin()) ?
prev(after_block) : blocks_ends.end();
            if(after_block != blocks_ends.end() && after_block -> first ==
end_orig) {
                if(before_block != blocks_ends.end() && before_block ->
second == start_orig) {
                    int new_start = before_block -> first;
                    int new_end = after_block -> second;
                    remove_by_sz(before_block);
                    remove_by_sz(after_block);
                    insert({new_start,new_end});
                } else {
                    int new_end = after_block -> second;
                    remove_by_sz(after_block);
                    insert({start_orig,new_end});
                }
            } else {
                if(before_block != blocks_ends.end() && before_block ->
second ==start_orig) {
                    int new_start = before_block -> first;
                    remove_by_sz(before_block);
                    insert({new_start,end_orig});
                } else {
                    insert({start_orig,end_orig});
                }
            }
        } else {
            auto it = blocks_sz_start.lower_bound(query);
            if(it == blocks_sz_start.end()) {
                memory_table[i] = std::make_pair(-1,query);
                std::cout << -1 << std::endl;
            } else {
                std::cout << it -> second << std::endl;
                int new_start = it -> second + query;
                int new_end = it -> first + it -> second;
                memory_table[i] = std::make_pair(it -> second, query);
                remove(it);
                insert({new_start,new_end});
            }
        }
    }
    return 0;
}
```

2. **Корректность применяемого алгоритма:**

Разобьём всю память на блоки, каждый из которых свободен или занят (эта часть памяти выделена в ответ на запрос). Свободные блоки не могут следовать друг за другом, так как аллоцированный блок памяти может находиться только либо в начале, либо после занятого блока. Для реализации менеджера памяти будем использовать мапы (обычный мап и мультимап, где отсутствует ограничение уникальности ключа) для сохранения самих блоков и «таблицу запросов» (на самом деле вектор их пар).

Реализуем требуемые методы следующим образом:

- Выделение памяти: возьмём изначальный блок, если не выполняется вышеприведённые требования, то запрос отклоняется, иначе уберётся нужное количество памяти и добавляется занятый блок.
- Освобождение памяти: из таблицы запросов возьмём нужный блок. Если запрос был отклонён, то алгоритм сидит сложа руки, иначе помечаем блок удалённым и смотрит на соседние блоки. Если по обе стороны находятся занятые блоки, то найденный блок становится свободным и возвращается в мап. Если же по обеим сторонам от занятого найденного блока находятся свободные, то удаляются текущий и правый блок и удлиняется левый блок на размеры текущего и соседнего. Если один из соседних блоков занят, а другой свободен, то удаляется текущий блок и удлиняется соседний свободный блок на его длину.

3. **Оценка сложности по времени:**

Операции с мапами стабильно составляют O(logn) из-за красно-чёрного дерева, операции с мультимапами составляют O(n). Значит, аллокация и освобождение памяти так же будут иметь O(logn). В общем, сложность алгоритма по времени при M запросов должна составлять **O(M*log(n))**.

4. **Оценка сложности по памяти:**

Для хранения самих блоков сложность алгоритма по памяти должна составлять **O(M)**. Почти все операции, связанные с мапами, требуют константной вспомогательной памяти.

Задача №12 «Минимум на отрезке»

1. **Пояснение к применённому алгоритму:**

Здесь применяется техника «скользящего окна». Оттуда просто находится минимум в каждом отрезке. В принципе задачу можно решать просто через массивы, но тогда нет смысла просто не использовать структуры данных.

Из-за того, что у меня возникли ряд проблемы с предыдущими решениями, я решил обращаться к секретному оружию — использование deque для решения данной задачи.

Здесь я создал двустороннюю очередь с емкостью k, который хранит только полезные элементы текущего окна из k элементов. Элемент полезен, если он находится в текущем окне и больше, чем все остальные элементы справа от него в текущем окне. Потом постепенно обрабатываются все элементы массива один за другим и сохраняются в двустороннюю очередь, чтобы он содержал полезные элементы текущего окна, и эти полезные элементы сохраняются в отсортированном порядке. Элемент, стоящий в самом начале двусторонней очереди, является самым большим, а элемент, стоящий позади двусторонней очереди, является **самым маленьким** в текущем окне.

Исходный код:

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

void sliding_window_min(std::vector<int> vec, int N, int K) {
    std::deque<int> window(K);
    int i;
    for (i = 0; i < K; ++i) {
        while ((!window.empty()) && vec[i] <= vec[window.back()])
            window.pop_back();
        window.push_back(i);
    }
    for (; i < N; ++i) {
        std::cout << vec[window.front()] << " ";
        while ((!window.empty()) && window.front() <= i - K)
            window.pop_front();
        while ((!window.empty()) && vec[i] <= vec[window.back()])
            window.pop_back();
        window.push_back(i);
    }
    std::cout << vec[window.front()];
}
int main() {
    int n, k;
    std::cin >> n >> k;

    std::vector<int> numbers;
    for (int i = 0; i < n; i++) {
        int num;
        std::cin >> num;
        numbers.push_back(num);
    }
    sliding_window_min(numbers, n, k);
    return 0;
}
```

2. **Корректность применяемого алгоритма:**

Сначала считаем все введённые числа. Наивная реализация (когда мы ищем минимум среди *K* чисел *N – K + 1* раз) имеет сложность O(n * k) и очевидно приведёт к TL.

Поэтому для нахождения минимума в окне будем использовать дек, в которой будем хранить сами значения в окне (конечно мог бы использовать их индексы, но Тараканов уже поторопил меня, так что увы).

В цикле по всем введённым числам, после того как прошли первые K элементов (размер текущего окна), каждый новый элемент добавляется в дек, а старые элементы удаляются, чтобы поддерживать размер окна K.

Затем удаляется слева от текущего элемента все элементы, числа которых больше или равно предыдущему значению (то есть до тех пор, пока не встретится число меньше текущего). То есть мы убеждаемся, что все элементы, которые меньше нового элемента и находится левее его, удаляются из дека. Таким образом, мы удаляем элементы, у которых уже нет шансов стать минимумом в окне.

Это гарантирует, что в деке всегда будут храниться только числа, которые могут быть потенциальным минимумом в текущем и следующих окнах. Значит, **минимальным значением** в текущем окне в таком случае будет первый элемент в деке (так как слева от него уже нет чисел меньше, потому что мы бы их удалили, а справа, если есть число меньше, то для него мы по идее должны были удалить первый элемент в цикле с удалением (потому что он меньше), но он есть в деке, поэтому справа чисел меньше никак не должны быть).

3. **Оценка сложности по времени:**

По ощущению кажется, что здесь квадратичная сложность. Но можно наблюдать, что каждый элемент вектора добавляется и вытаскивается только 1 раз. Поэтому здесь на самом деле 2*n операций, и в итоге сложность алгоритма по времени должна составлять **O(n)**.

4. **Оценка сложности по памяти:**

Из-за самих значений в векторе сложность алгоритма по памяти должна составлять **O(n)**. Важен здесь ещё тот факт, что при работе двусторонней очереди используется ещё и вспомогательная память **O(k)**, но очевидно, что N всегда больше K, так что это не будет оказывать существенного влияния на сложность по памяти.

В итоге: **O(n+k)**.

Задача №1494 «Монобильярд»

1. **Пояснение к применённому алгоритму:**

Довольно простая задача, здесь используются свойства стека. Так как Чичиков забивает бильярдные шарики в лузу по порядку, уместнее всего использовать стек. Этого должен проверять ревизор: если шарик не соответствует тому, что должно было быть, то сразу же оборвётся цикл проверки, из-за которого из стека прекратится процесс вытаскивания шариков из стека, вследствие которого можно выявить, жульничал ли Чичиков или нет.

Исходный код:

```cpp
#include <array>
#include <iostream>
#include <set>
#include <stack>
#include <unordered_set>

int main() {
    int n, p = 0;
    std::cin >> n;
    int balls[n];
    for(int i = 0; i < n; ++i) {
        std::cin >> balls[i];
    }
    std::stack<int> st;
    for(int i = 1; i <= n; i++) {
        st.push(i);
        while(!st.empty() && balls[p] == st.top()) {
            p++;
            st.pop();
        }
    }
    while(p < n) {
        if(balls[p] == st.top()) {
            p++;
            st.pop();
        } else {
            break;
        }
    }
    (st.empty()) ? std::cout << "Not a proof" : std::cout << "Cheater";
    return 0;
}
```

2. Корректность применяемого алгоритма:

Для того, чтобы точно воспроизвести корректный порядок, по которому шарики должны были кататься в лузу, а потом сравнивать его с порядком от Чичикова, надо сначала заполнить стек по корректному порядку. Наряду с этим уже проходит проверка на честность (то есть когда ревизор подходит после забитого шарика): если забитый шарик не соответствует с тем что должно было быть, то пропускается операция. Если опять там всё хорошо, то после итерации стек должен быть пустым, что гарантирует честность в игре. В противном случае всё останавливается, и так как стек не пуст, то это означает, что Чичиков жульничал в игре.

Рассмотрим этого на конкретном примере:

10

5 8 7 6 4 10 2 1 3 9

Пускай мы встретили во входе 5, значит докидываем в стек 1 2 3 4 5.

Затем процесс удаления: сравниваем 5 из входа с последним элементом, они равны, значит становится 1 2 3 4.

Потом встретили 8, оно больше нашего последнего закинутого шара, докидываем 6 7 8, получаем 1 2 3 4 6 7 8.

И повторяем процесс удаления, описанный выше.

Потом допустим 7, 6, 4, и каждый раз число из входа совпадает с последним элементом массива.

После удаления получим 1 2 3.

Потом получаем 10, оно больше последнего закинутого шара, докидываем в массив, получаем 1 2 3 9 10 (и сразу же удаляем, получаем 1 2 3 9).

Потом получаем 2, но последний элемент 9, а не 2, и тут мы понимаем, что ответ Cheater.

3. **Оценка сложности по времени:**

Все операции со стеком в алгоритме занимает O(1). Так как здесь нет какого-то нюанса с структурами данных, который может мешать нормальному оцениванию сложности по времени, алгоритм должен занимать **O(n)** времени.

4. **Оценка сложности по памяти:**

Стек — линейный тип данных, поэтому для хранения шариков сложность по памяти должна составлять **O(n)**.

Задача №1628 «Белые полосы»

1. **Пояснение к применённому алгоритму:**

Читаем координаты занятых ячеек и записываем их в вектор в виде пары координат, потом добавляем в вектор ячейки, формирующие границы календаря. Оттуда с помощью сортировок можем найти все полоса (горизонтальные, потом вертикальные).

Замечание: здесь оптимально было применить красно-черное дерево, но штош ¯_(ツ)_/¯

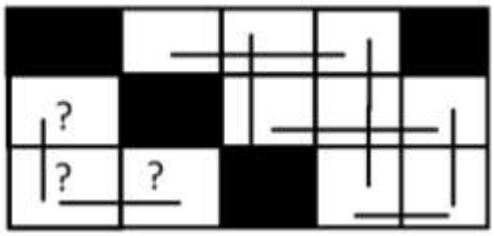
Исходный код:

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

static int m, n, k;

bool horizontal_comparator(std::pair<int, int> p1, std::pair<int, int> p2)
{
    if (p1.first < p2.first) {
        return true;
    } else if (p1.first > p2.first) {
        return false;
    } else {
        return p1.second < p2.second;
    }
}

bool vertical_comparator(std::pair<int, int> p1, std::pair<int, int> p2) {
    if(p1.second < p2.second) {
        return true;
    } else if(p1.second > p2.second) {
        return false;
    } else {
```

```
        return p1.first < p2.first;
    }
}

int main() {
    std::cin >> m >> n >> k;
    std::vector<std::pair<int, int>> occupied_days;
    for (int i = 0; i < m + 2; i++) {
        occupied_days.emplace_back(i, 0);
        occupied_days.emplace_back(i, n + 1);
    }
    for (int i = 1; i < n + 1; i++) {
        occupied_days.emplace_back(0, i);
        occupied_days.emplace_back(m + 1, i);
    }
    for (int i = 0; i < k; i++) {
        int x, y;
        std::cin >> x >> y;
        occupied_days.emplace_back((x), (y));
    }
    std::unordered_set<int> one_days;
    int weeks = 0;
    std::sort(occupied_days.begin(), occupied_days.end(),
horizontal_comparator);
    for (int i = 1; i < occupied_days.size(); i++) {
        std::pair<int, int> prev = occupied_days[i - 1];
        std::pair<int, int> curr = occupied_days[i];
        if (curr.first == prev.first) {
            int dist = curr.second - prev.second - 1;
            if (dist == 1) {
                one_days.emplace(curr.first * n + curr.second - 1);
            } else if (dist > 1) {
                weeks++;
            }
        }
    }
    std::sort(occupied_days.begin(), occupied_days.end(),
vertical_comparator);
    for (int i = 1; i < occupied_days.size(); i++) {
        std::pair<int, int> prev = occupied_days[i - 1];
        std::pair<int, int> curr = occupied_days[i];
        if (curr.second == prev.second) {
            int dist = curr.first - prev.first - 1;
            if (dist == 1) {
                auto item = one_days.find((curr.first - 1) * n +
curr.second);
                if (item != one_days.end()) {
                    weeks++;
                }
            } else if (dist > 1) {
                weeks++;
            }
        }
    }
    std::cout << weeks;
    return 0;
}
```

2. Корректность применяемого алгоритма:

Надо сначала понимать суть задания. Нам надо найти 1*l или l*1 прямоугольники, которые не являются взаимовключенными.



А нафига здесь 8, а не 19? 11?

Здесь точно 2.

Сортируем вектор так, чтобы ячейки шли по порядку своего расположения в строках (слева-направо, сверху-вниз). Проходим по порядку вектор, считая все свободные "серии" длиной > 1, и сохраняя в неупорядоченный набор все одиночные свободные дни. Далее сортируем вектор по порядку расположения в столбцах (сверху-вниз, слева-направо). Опять проходим по нему, считая все свободные "серии" длиной > 1, а, встречая одиночный свободный день, проверяем, был ли он одиночным при прошлом проходе, если да — это изолированный свободный день, и его считаем как "серии" длиной 1.

В итоге получаем то, что в ответе гарантированно будет все полоса без взаимовключения (этому способствует набор).

3. **Оценка сложности по времени:**

Опять же, посмотрим сложности операции, связанные с структурами данных.

S = n + m (измерения календаря) + k (неудачные дни).

O(S) — чтение

O(S*logS) — сортировка (introsort)

O(S) — проход по вектору

В итоге сложность алгоритма по времени должна составлять **O(S*logS)**.

4. **Оценка сложности по памяти:**

Здесь используется вектор, поэтому сложности алгоритма по памяти должна составлять **O(n)**.

Доп: декартовое дерево (treap, дерамида, дуча) по неявному ключу: материализация быстрой сортировки.

1. **Пояснение к применённому алгоритму:**

Декартово дерево, дуча, дерамида (treap) — структура данных, сочетающая в себе двоичное дерево и двоичную кучу. Хранит в каждом узле пары (x, y), где

для ключа x служит бинарным деревом поиска, а для приоритета y — двоичной кучей.

Исходный код:

```
#include <iostream>
#include <cstdlib>
#include <ctime>

struct Node {
    int data;
    int priority;
    Node *l, *r;
    Node(int d) {
        this -> data = d;
        this -> priority = rand() % 100;
        this -> l = this -> r = nullptr;
    }
};
void rotateLeft(Node* &root) {
    Node* R = root -> r;
    Node* X = root -> r -> l;
    R -> l = root;
    root -> r = X;
    root = R;
}
void rotateRight(Node* &root) {
    Node* L = root -> l;
    Node* Y = root -> l -> r;
    L -> r = root;
    root -> l = Y;
    root = L;
}
void insertNode(Node* &root, int d) {
    if (root == nullptr) {
        root = new Node(d);
        return;
    }
    if (d < root -> data) {
        insertNode(root -> l, d);
        if (root -> l != nullptr && root -> l -> priority > root ->
priority) {
            rotateRight(root);
        }
    } else {
        insertNode(root -> r, d);
        if (root -> r != nullptr && root -> r -> priority > root ->
priority) {
            rotateLeft(root);
        }
    }
}
bool searchNode(Node* root, int key) {
    if (root == nullptr) {
        return false;
    }
    if (root -> data == key) {
        return true;
    }
    if (key < root -> data) {
```

```
        return searchNode(root -> l, key);
    }
    return searchNode(root -> r, key);
}
void deleteNode(Node* &root, int key) {
    if (root == nullptr) {
        return;
    }
    if (key < root -> data) {
        deleteNode(root -> l, key);
    } else if (key > root -> data) {
        deleteNode(root -> r, key);
    } else {
        if (root -> l == nullptr && root -> r == nullptr) {
            delete root;
            root = nullptr;
        } else if (root -> l && root -> r) {
            if (root -> l -> priority < root -> r -> priority) {
                rotateLeft(root);
                deleteNode(root -> l, key);
            } else {
                rotateRight(root);
                deleteNode(root -> r, key);
            }
        } else {
            Node* child = (root -> l) ? root -> l: root -> r;
            Node* curr = root;
            root = child;
            delete curr;
        }
    }
}
void displayTreap(Node *root, int space = 0, int height = 10) {
    if (root == nullptr) return;
    space += height;
    displayTreap(root -> l, space);
//    std::cout << "\n";
//    for (int i = height; i < space; i++) {
//        std::cout << ' ';
//    }
    std::cout << root -> data << "(" << root -> priority << ")\n";
//    std::cout << "\n";
    displayTreap(root -> r, space);
}
int main() {
    int nums[] = {1,7,6,4,3, 2,8,9,10};
    Node* root = nullptr;
    srand(time(nullptr));
    for (int n: nums) {
        insertNode(root, n);
    }
    std::cout << "Построенная дуча:\n\n";
    displayTreap(root);
    std::cout << "\nДобавление узла 5:\n\n";
    insertNode(root, 5);
    displayTreap(root);
    std::cout << "\nУдаление узла 8:\n\n";
    deleteNode(root, 8);
    displayTreap(root);
    std::cout << "\nУдаление узла 3:\n\n";
    deleteNode(root, 3);
```

```
    displayTreap(root);
    std::cout << "\nПоиск всех узлов с 1 до 10:\n\n";
    for(int i = 1; i <= 10; i++) {
        std::cout << i << ": ";
        (searchNode(root, i)) ? std::cout << "есть" << "\n" : std::cout <<
"нет" << "\n";
    }
    return 0;
}
```

2. **Корректность применяемого алгоритма:**

Здесь нужно исследовать корректность всех функций путём проверки необходимых условий, которому должно преследовать дерево.

- Функции вращения: вспомогательные функции для удовлетворения свойства кучи при операции, изменяющие структуру дерева.
- Функция вставки: если узел меньше корня по значению, то вставка происходит в левое поддерево, иначе в правое. При вставке нового узла получившееся дерево необходимо должно соблюдать свойство кучи. Чтобы это произошло, здесь на помощь приходят те самые функции вращения, в результате которого корнем кучи становится новый узел, который раньше был в одних из двух поддеревьев.
- Функция поиска: сам по себе этот структура данных представляет из себя поисковое дерево с неявным ключом. По сути это реализация бинарного поиска: если ключ меньше корневого, то дальше рекурсивным образом идёт поиск в левом поддереве, иначе в правом.
- Функция удаления: сначала надо найти тот узел, которого мы хотим удалить. Дальше 3 кейса:
    - Если удаляемый узел является листовым, то сначала деаллоцируется память, потом сам корень обновляется в нулевой указатель. После чего удаляется сам корень.
    - Если удаляемый узел имеет 2 дочерних, то сначала определяется какой дочерний узел находится ниже по приоритету. Потом в соответствии с этим рекурсивно удаляется левый (или правый) дочерний узел.
    - Если удаляемый узел имеет 1 дочернего, то сначала находится этот дочерний узел, потом деаллоцируется связанная память.
- Функция вывода: выводит дучу в результате вышеприведённых операций.

Все эти функции не нарушает свойство кучи, так что можно утверждать, что алгоритм работает корректно.

3. **Оценка сложности по времени:**

Время работы всех этих операции (вставка, поиск, удаление) зависит от глубины самого дерева, так что сложность по времени должна составлять в среднем **O(logn)**, а в худшем **O(n)**. Это происходит из-за того, что дерево может стать несбалансированным (вероятность такого случая мала, но не ноль).

4. **Оценка сложности по памяти:**

По сути сам дерево хранится в виде массива, так что сложность по памяти должна составлять **O(n)**.