Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №2

по «Алгоритмам и структурам данных»

Базовые задачи

Выполнил:

Студент группы P3233

Гуменник П. О.

Преподаватели:

Косяков М.С.

Тараканов Д.С.

Санкт-Петербург

2023

Оглавление

Задача J. Гоблины и очереди

Код:

```cpp
#include <iostream>
#include <deque>
#include <string>

void processGoblins() {
    int n;
    std::cin >> n;

    std::deque<std::string> left, right;
    std::string command, goblin_id;

    auto rebalance = [&]() {
        // Maintain balance between left and right
        while (left.size() > right.size() + 1) {
            right.push_front(left.back());
            left.pop_back();
        }
        while (right.size() > left.size()) {
            left.push_back(right.front());
            right.pop_front();
        }
    };

    for (int i = 0; i < n; ++i) {
        std::cin >> command;
        if (command[0] == '+') {
            std::cin >> goblin_id;
            right.push_back(goblin_id);
        } else if (command[0] == '*') {
            std::cin >> goblin_id;
            if (left.size() <= right.size()) {
                left.push_back(goblin_id);
            } else {
                right.push_front(goblin_id);
            }
        } else if (command == "-") {
            if (left.empty()) {
                // Transfer elements from right to
                while (!right.empty()) {
                    left.push_back(right.front());
                    right.pop_front();
                }
            }
            std::cout << left.front() << '\n';
            left.pop_front();
        }
        rebalance();
    }
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);

    processGoblins();

    return 0;
}
```

Поояснение к примененному алгоритму:

Алгоритмическая сложность

1. Вставка элемента в конец очереди (+ i): Вставка в deque в конец имеет сложность O(1).

2. Вставка элемента в середину очереди (* i): Операции вставки в начало или конец deque также имеют сложность O(1).

3. Удаление первого элемента из очереди (-): Удаление из начала deque (т.е. left) имеет сложность O(1)..

Перебалансировка: Это критическая операция, которая гарантирует, что операции - могут быть выполнены эффективно. Каждая операция перебалансировки требует перемещения элемента из одной deque в другую, если длины deque не удовлетворяют условиям баланса. Это O(1) на каждую операцию, так как перемещается только один элемент.

Общая алгоритмическая сложность:

Средняя амортизированная сложность каждой операции в нашем алгоритме составляет O(1). Это подразумевает, что для N операций сложность будет O(N).

Доказательство корректности:

1. Очередь и сохранение порядка: Мы используем две deque для представления очереди, где left содержит первую половину элементов, а right — вторую половину. Это обеспечивает корректное порядковое расположение элементов при их добавлении и удалении.

2. Вставка в конец: Элементы, добавляемые в конец (+ i), добавляются в right, что соответствует тому, что они идут после всех элементов в left, поддерживая порядок FIFO (first-in, first-out).

3. Вставка в середину: Элементы, добавляемые в середину (* i), вставляются либо в конец left, либо в начало right, что соответствует их позиции в середине полной очереди. Конкретный выбор зависит от текущего баланса между left и right, чтобы поддерживать балансировку и минимизировать необходимость переноса элементов при удалении.

4. Удаление: Элементы удаляются из начала left, что соответствует первому элементу в полной очереди. Если left пуст, элементы из right переносятся в left, сохраняя порядок FIFO.

5. Перебалансировка: Перебалансировка после каждой операции гарантирует, что left и right делят все элементы примерно поровну. Это обеспечивает эффективное добавление в середину и удаление из начала, а также уменьшает количество операций переноса элементов.

Задача K. Менеджер памяти-1

Код:

```cpp
#include <iostream>
#include <set>
#include <map>

using namespace std;

struct Block {
    int start, size;
    Block() : start(0), size(0) {}  // Конструктор по умолчанию
    Block(int s, int sz) : start(s), size(sz) {}
    bool operator<(const Block& other) const {
        return start < other.start;
    }
};

struct BlockBySize {
    int start, size;
    BlockBySize(int s, int sz) : start(s), size(sz) {}
    bool operator<(const BlockBySize& other) const {
        if (size != other.size) return size < other.size;
        return start < other.start;
    }
};

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int N, M;
    cin >> N >> M;

    set<Block> freeBlocksByStart;
    set<BlockBySize> freeBlocksBySize;
    map<int, Block> allocatedBlocks;

    freeBlocksByStart.insert(Block(1, N));
    freeBlocksBySize.insert(BlockBySize(1, N));

    int query;
    for (int i = 1; i <= M; i++) {
        cin >> query;
        if (query > 0) {
            int K = query;
            auto it = freeBlocksBySize.lower_bound(BlockBySize(0, K));
            if (it != freeBlocksBySize.end() && it->size >= K) {
                int start = it->start;
                int size = it->size;

                // Remove from both sets
                freeBlocksByStart.erase(Block(start, size));
                freeBlocksBySize.erase(it);

                // Allocate memory
                allocatedBlocks[i] = Block(start, K);
                cout << start << "\n";

                // Update free space
                if (size > K) {
                    freeBlocksByStart.insert(Block(start + K, size - K));
                    freeBlocksBySize.insert(BlockBySize(start + K, size - K));
                }
            } else {
                cout << "-1\n";
            }
        } else if (query < 0) {
            int T = -query;
            if (allocatedBlocks.count(T)) {
                Block block = allocatedBlocks[T];
                allocatedBlocks.erase(T);

                // Free memory
                auto it = freeBlocksByStart.lower_bound(Block(block.start, 0));
                if (it != freeBlocksByStart.begin() && (--it)->start + it->size == block.start) {
                    block.start = it->start;
                    block.size += it->size;
                    freeBlocksBySize.erase(BlockBySize(it->start, it->size));
                    freeBlocksByStart.erase(it);
                }

                auto next = freeBlocksByStart.lower_bound(Block(block.start + block.size, 0));
                if (next != freeBlocksByStart.end() && block.start + block.size == next->start) {
                    block.size += next->size;
                    freeBlocksBySize.erase(BlockBySize(next->start, next->size));
                    freeBlocksByStart.erase(next);
                }

                freeBlocksByStart.insert(block);
                freeBlocksBySize.insert(BlockBySize(block.start, block.size));
            }
        }
    }

    return 0;
}
```

Пояснение к примененному алгоритму:

Алгоритмическая сложность

Основные операции:

1. Выделение памяти:

- Поиск подходящего блока по размеру с помощью lower_bound в set<BlockBySize>: O(log n), где n — количество свободных блоков.

- Удаление и вставка в set<Block> и set<BlockBySize>: каждая операция занимает O(log n).

- Обновление информации о выделенных блоках в map<int, Block>: O(log m), где m — количество выделенных блоков.

2. Освобождение памяти:

- Поиск и удаление блока в map<int, Block>: O(log m).

- Операции слияния соседних свободных блоков в set<Block> и set<BlockBySize>: в худшем случае несколько операций поиска, удаления и вставки, каждая из которых O(log n).

Общая сложность: Поскольку каждая операция в худшем случае включает в себя логарифмическое количество шагов относительно числа блоков в соответствующих структурах данных, общая сложность одного запроса будет O(log n + log m). При множестве запросов M общая сложность будет O(M * (log n + log m)).

Доказательство корректности

Основные действия:

- Выделение памяти: Поиск наиболее подходящего свободного блока осуществляется по размеру. Если блок найден, он удаляется из свободных блоков и добавляется в список занятых. Это гарантирует, что блок будет немедленно удалён из доступных для выделения, предотвращая его повторное использование.

- Освобождение памяти: Блок, ранее выделенный, возвращается в список свободных блоков. Алгоритм проверяет возможность слияния с соседними свободными блоками, что эффективно управляет фрагментацией памяти. Блоки объединяются в большие, если их граничные адреса совпадают, что оптимизирует последующие операции выделения.

Логическая корректность:

- Выделение следует правилам: Каждый запрос на выделение ищет блок достаточного размера, учитывая текущий список свободных блоков, что соответствует условиям задачи.

- Освобождение корректно обновляет данные: Освобождённые блоки правильно возвращаются в список свободных блоков, слияния выполняются корректно, что обеспечивает возможность их повторного использования.

Эффективное управление памятью:

- Использование двух set ускоряет операции поиска и обновления свободных блоков, а map обеспечивает быстрый доступ к информации о занятых блоках. Это сокращает время, необходимое для обработки каждого запроса, и уменьшает вероятность ошибок в управлении памятью.

Этот подход является эффективным для управления памятью в условиях частых запросов на выделение и освобождение, предотвращая при этом чрезмерную фрагментацию и упрощая задачу выделения памяти в будущем.

Задача L. Минимум на отрезке

Код:

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

using namespace std;

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

    int N, K;
    cin >> N >> K;
    vector<int> arr(N);

    for (int i = 0; i < N; i++) {
        cin >> arr[i];
    }

    deque<int> dq;
    vector<int> minima;

    for (int i = 0; i < N; i++) {
        if (!dq.empty() && dq.front() == i - K) {
            dq.pop_front();
        }

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

        dq.push_back(i);

        if (i >= K - 1) {
            minima.push_back(arr[dq.front()]);
        }
    }

    for (int min_val : minima) {
        cout << min_val << " ";
    }

    return 0;
}
```

Пояснение к примененному алгоритму:

Алгоритмическая сложность:

Операции в алгоритме:

1. Перемещение по массиву: Алгоритм проходит через каждый элемент массива один раз.

2. Обновление deque: Каждый элемент массива добавляется в deque и удаляется из deque не более одного раза, что обеспечивает амортизированное время работы O(1) для каждой операции с deque.

- Добавление в deque и удаление из deque выполняются за константное амортизированное время, так как каждый индекс добавляется и удаляется один раз.

3. Вывод минимума: Для каждого окна минимум извлекается из начала deque за время O(1).

Итоговая сложность: Основываясь на вышеизложенном, общая сложность алгоритма составляет O(N), где N — количество элементов в массиве. Это объясняется тем, что каждый элемент рассматривается константное количество раз при добавлении и удалении из deque, и каждое окно требует константного времени для извлечения минимума.

Доказательство корректности:

Логика алгоритма:

Алгоритм использует структуру данных deque для поддержания индексов элементов в порядке их возрастания, что обеспечивает быстрый доступ к текущему минимальному значению в "окне". Далее представлено доказательство корректности работы алгоритма:

1. Инвариант deque:

- В deque хранятся индексы элементов таким образом, что значения элементов массива по этим индексам возрастают от начала к концу deque. То есть если i и j — индексы в deque и i < j, то arr[i] ≤ arr[j].

- Элемент в начале deque всегда представляет минимальное значение в текущем окне.

2. Поддержание инварианта:

- Удаление из начала deque: Если индекс на переднем конце deque выходит за пределы текущего окна (т.е. dq.front() < i - K + 1, где i — текущий индекс элемента), он удаляется. Это гарантирует, что в deque остаются только индексы, соответствующие элементам внутри текущего окна.

- Удаление из конца deque: Перед добавлением нового индекса в deque, удаляются все индексы элементов, значения которых больше значения текущего элемента, так как они не могут быть минимумом для последующих окон, содержащих текущий элемент.

- Добавление индекса: Индекс текущего элемента добавляется в конец deque, сохраняя инвариант возрастающего порядка значений.

3. Доступ к минимальному значению:

- После обработки первого полного окна (и для всех последующих), минимальное значение для текущего окна будет находиться на начале deque, так как инвариант гарантирует, что самый маленький элемент окна будет первым.