Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №1

по «Алгоритмам и структурам данных»

Блок 3

Выполнил:
Студент группы P3216
Билошийцкий Михаил Владимирович

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

Санкт-Петербург
2024

I. Машинки

Решение:

Решение эффективно определяет минимальное количество операций, необходимых маме Пети, используя две ключевые структуры данных: vector last_seen и vector next_use. Vector last_seen отслеживает, когда в последний раз играли с каждой машинкой, а vector next_use хранит информацию о том, когда каждая машинка будет использована в следующий раз.

Путем перебора желаемой последовательности машинок и поддержания множества on_floor, представляющего машинки, находящиеся в данный момент на полу, алгоритм выбирает, какую машинку заменить, когда пол заполнен. Используя next_use, гарантируется, что машинка, которая будет использоваться в следующий раз как можно позже, будет убрана на полку. Это минимизирует необходимость будущих замен и оптимизирует действия мамы.

Дополнительные пояснения в коде:

```
#include <iostream>
#include <vector>
#include <set>

#define NOT_SEEN -1

int main() {
    using namespace std;
    int32_t n, k, p, car;
    cin >> n >> k >> p;

    // last_seen[i] хранит индекс последнего появления машинки i+1
    // next_use[i] хранит индекс следующего использования машинки i+1
    vector<int32_t> last_seen(n, NOT_SEEN), next_use(p, p);

    // Считываем последовательность машинок и заполняем next_use
    for (int32_t i = 0; i < p; i++) {
        cin >> car;
        car--;
        if (last_seen[car] != NOT_SEEN)
            next_use[last_seen[car]] = i;
        last_seen[car] = i;
    }

    int32_t res = 0;
```

```
    // on_floor - индексы машинок, которые сейчас на полу
    set<int32_t> on_floor;
    for (int32_t i = 0; i < p; i++) {
        if (on_floor.count(i) <= 0) {
            res++;
            if (on_floor.size() == k) {
                auto it = on_floor.end();
                on_floor.erase(prev(it));
            }
            on_floor.insert(next_use[i]);
            continue;
        }
        on_floor.erase(i);
        on_floor.insert(next_use[i]);
    }
    cout << res << endl;
    return 0;
}
```

J. Гоблины и очереди

Решение:
Решение реализует собственный класс очереди под названием GoblinsQueue, чтобы эффективно управлять порядком гоблинов. Эта очередь разделена на две половины, но является по сути одной очередью, что позволяет выполнять вставку в середину за O(1), добавляя значения между половинками.

Операция push_back добавляет обычного гоблина в конец, а push_center вставляет привилегированного гоблина в середину, поддерживая баланс между половинами. Операция pop_front удаляет и возвращает первого гоблина в очереди, гарантируя сохранение правильного порядка. Эта реализация эффективно обрабатывает операции с очередью и выполняет все операции за O(1) во избежание TL.

Дополнительные пояснения в коде:

```
#include <iostream>
#include <list>

// Основные операции
#define OP_DELETE '-'
#define OP_ADD '+'
```

```cpp
#define OP_ADD_CENTER '*'

// Очередь с гоблинами из двух половин (для достияжения O(1) при добавлении в середину)
// Реализованы 3 основные операции
// push_back - добавление гоблина в конец очереди
// push_center - добавление привелигированного гоблина в середину очереди
// pop_front - удаление гоблина из начала очереди
class GoblinsQueue {
private:
    // Первая половина очереди
    std::list<int32_t> first_half;
    // Вторая половина очереди
    std::list<int32_t> second_half;
public:
    GoblinsQueue() = default;

    void push_back(int32_t goblin_number) {
        second_half.push_back(goblin_number);
        if (second_half.size() > first_half.size()) {
            first_half.push_back(second_half.front());
            second_half.pop_front();
        }
    }

    void push_center(int32_t goblin_number) {
        first_half.push_back(goblin_number);
        if (first_half.size() - 1 > second_half.size()) {
            second_half.push_front(first_half.back());
            first_half.pop_back();
        }
    }

    int32_t pop_front() {
        int32_t value = first_half.front();
        first_half.pop_front();
        if (second_half.size() > first_half.size()) {
            first_half.push_back(second_half.front());
            second_half.pop_front();
```

```
        }
        return value;
    }
};

int main(int argc, char** argv) {
    using namespace std;
    size_t n;
    cin >> n;
    // Очередь с гоблинами
    GoblinsQueue goblins_queue;
    int32_t goblin_number;
    char operation;
    for (size_t i = 0; i < n; i++) {
        cin >> operation;
        if (operation == OP_ADD) {
            // Добавляем обычного гоблина в конец очереди
            cin >> goblin_number;
            goblins_queue.push_back(goblin_number);
        } else if (operation == OP_ADD_CENTER) {
            // Добавляем привелигированного гоблина в середину очереди
            cin >> goblin_number;
            goblins_queue.push_center(goblin_number);
        } else if (operation == OP_DELETE) {
            // Вывод номера гоблина, что ушёл к шаману с начала очереди
            cout << goblins_queue.pop_front() << endl;
        }
    }
    return 0;
}
```

K. Менеджер памяти - 1

Решение:

Структуры в алгоритме.

number_block: Неупорядоченный map, хранящий в себе занятые блоки, связывая каждый блок с его номером запроса на выделение.
size_block: multimap, хранящее свободные блоки, отсортированные по их размеру. Это позволяет быстро найти подходящий блок при поступлении запроса на выделение.

start_block: Карта, хранящая свободные блоки, отсортированные по их начальному адресу.

Алгоритм обрабатывает каждый запрос следующим образом:

Запрос на выделение:

Поиск подходящего блока: В size_block ищется свободный блок размером не менее запрашиваемого.
Блок найден: Если подходящий блок найден, он удаляется из size_block и start_block. Если блок больше, чем нужно, он разделяется на два, а оставшаяся часть добавляется обратно в структуры данных свободных блоков. Выделенный блок затем добавляется в number_block и его начальный адрес выводится на экран.

Блок не найден: Если подходящий блок не найден, выводится -1, указывающее на ошибку выделения.

Запрос на освобождение:

Поиск блока: Блок, соответствующий заданному номеру запроса на выделение, извлекается из number_block.

Объединение с соседними блоками: Код проверяет наличие смежных свободных блоков (используя start_block) и объединяет их с освобожденным блоком, если они существуют. Это обеспечивает эффективное использование свободного пространства.

Добавление в свободные блоки: Результирующий свободный блок добавляется как в size_block, так и в start_block для будущих выделений.

Эта реализация эффективно управляет памятью, быстро находя подходящие блоки для выделения и объединяя смежные свободные блоки при освобождении, минимизируя фрагментацию и максимизируя использование памяти.

Дополнительные пояснения в коде:

```
#include <iostream>
#include <map>
#include <unordered_map>

// Блок памяти
struct block {
    int32_t start;
    int32_t end;
};

void remove_from_size_block(std::multimap<int32_t, block>& size_block, block b) {
    auto it = size_block.find(b.end - b.start + 1);
    while (it->second.start != b.start) it++;
```

```
    size_block.erase(it);
}

int main(int argc, char** argv) {
    using namespace std;
    ios_base::sync_with_stdio(false);
    // n - количество ячеек памяти, m - количество запросов
    int32_t n, m;
    cin >> n >> m;

    // map номер -> блок, занятые блоки, отсортированные по номеру
    unordered_map<int32_t, block> number_block;
    // multimap размер -> блок, освобождённые блоки, отсортированные по размеру
    multimap<int32_t, block> size_block;
    // map начало -> блок, освобождённые блоки, отсортированные по началу
    map<int32_t, block> start_block;

    int32_t query;
    cin >> query;
    // Заполняем самый первый блок
    if (query > n) {
        cout << -1 << endl;
    } else {
        number_block[1] = { 1, query };
        cout << 1 << endl;
    }

    // Заполняем свободное пространство
    if (query != n) {
        if (query > n) query = 0;
        size_block.insert({n - query, { query + 1, n } });
        start_block[query + 1] = { query + 1, n };
    }

    for (int32_t i = 2; i <= m; i++) {
        cin >> query;
        // Запрос на выделение памяти
        if (query > 0) {
            int32_t block_size = query;
```

```
            // Поиск блока из освобождённых, в который можно влезть
            auto it = size_block.lower_bound(block_size);
            if (it != size_block.end() && block_size <= it->first) {
                // Найден блок, который подходит по размеру
                block b = it->second;
                // Удаляем его из освобождённых блоков
                size_block.erase(it);
                // Удаляем его также из блоков, отсортированных по началу
                start_block.erase(b.start);
                // Разбиваем блок на две части, если блок больше запрашиваемого
размера
                if (b.end - b.start + 1 > block_size) {
                    // Добавляем оставшуюся часть блока в освобождённые блоки
                    size_block.insert({ b.end - b.start + 1 - block_size, { b.start +
block_size, b.end } });
                    // Добавляем оставшуюся часть блока в блоки, отсортированные по
началу
                    start_block[b.start + block_size] = { b.start + block_size, b.end
};
                }
                // Выделяем блок
                number_block[i] = { b.start, b.start + block_size - 1 };
                // Выводим номер первой ячейки памяти в выделенном блоке
                cout << b.start << endl;
            } else {
                // Не удалось найти подходящий блок
                cout << -1 << endl;
            }
        }
        // Если запрос на освобождение памяти
        else if (query < 0) {
            int32_t number = -query;
            if (number_block.find(number) == number_block.end()) continue;
            block b = number_block[number];

            // Убираем блок из занятых
            number_block.erase(number);

            // Находим правый прилегающий блок
```

```
            auto it_right = start_block.upper_bound(b.start);
            if (it_right != start_block.end()) {
                block right_b = it_right->second;
                if (b.end + 1 == right_b.start) {
                    // Сливаем блоки
                    b = { b.start, right_b.end };
                    // Удаляем правый блок
                    start_block.erase(right_b.start);
                    remove_from_size_block(size_block, right_b);
                }
            }

            // Находим левый прилегающий блок
            auto it_left = start_block.lower_bound(b.start);
            if (it_left != start_block.begin()) {
                it_left--;
                block left_b = it_left->second;
                if (left_b.end + 1 == b.start) {
                    // Сливаем блоки
                    b = { left_b.start, b.end };
                    // Удаляем левый блок
                    start_block.erase(left_b.start);
                    remove_from_size_block(size_block, left_b);
                }
            }

            // Добавляем в освобождённые, отсортрированные по размеру
            size_block.insert({ b.end - b.start + 1, b });
            // Добавляем в освобождённые, отсортрированные по началу
            start_block[b.start] = b;
        }
    }
    return 0;
}
```

L. Минимум на отрезке

Заполнение deque: Для первого окна размером K deque заполняется индексами элементов в окне. Deque поддерживает убывающий порядок значений, причем индекс наименьшего элемента находится в начале. Если новый элемент меньше или равен последнему элементу в деке, то последний элемент (и все последующие элементы с большим или равным значением) уже не смогут быть минимумом в будущем окне, поэтому их можно удалить.

Скольжение окна:

Вывод минимума: Значение в начале deque является минимумом в текущем окне и выводится на экран.
Удаление устаревших индексов: Индексы элементов, которые больше не находятся в окне (т.е. их позиции меньше i - k), удаляются из начала дека.
Поддержание убывающего порядка: Аналогично этапу инициализации, элементы из конца deque удаляются, если текущий элемент меньше или равен. Это гарантирует, что deque всегда содержит потенциальные минимальные значения в убывающем порядке.
Добавление текущего индекса: Индекс текущего элемента добавляется в конец deque.
Этот процесс продолжается до конца последовательности, эффективно находя минимальное значение для каждой позиции скользящего окна.

Ключевые моменты:

Deque эффективно поддерживает динамический список потенциальных минимальных значений, избегая необходимости сканировать все окно каждый раз.
Поддержание убывающего порядка внутри deque позволяет быстро определять и удалять элементы, которые больше не актуальны.
Этот подход имеет временную сложность O(N), поскольку каждый элемент обрабатывается только один раз.

Дополнительные пояснения в коде:

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

int main() {
    using namespace std;
    int32_t n, k;
    cin >> n >> k;
    int32_t arr[n];
    for (int32_t i = 0; i < n; i++) {
        cin >> arr[i];
    }
    deque<int32_t> nums;
    // Заполняем deque для первого окна длины k
    for (int32_t i = 0; i < k; i++) {
        while (!nums.empty() && arr[i] <= arr[nums.back()]) {
            nums.pop_back();
```

```
        }
        nums.push_back(i);
    }
    // Печатаем минимумы и обновляем deque при сдвиге окна
    for (int32_t i = k; i < n; i++) {
        cout << arr[nums.front()] << ' ';
        // Удаляем индексы элементов, которые больше не в окне
        while (!nums.empty() && nums.front() <= i - k) {
            nums.pop_front();
        }
        // Удаляем индексы элементов, которые больше не могут быть минимумами
        while (!nums.empty() && arr[i] <= arr[nums.back()]) {
            nums.pop_back();
        }
        nums.push_back(i);
    }
    cout << arr[nums.front()] << endl;
    return 0;
}
```