Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №3

по «Алгоритмам и структурам данных»

Базовые задачи

Выполнила:

Студент группы P3232

Копалина М.А.

Преподаватели:

Косяков М.С.

Тараканов Д.С.

Санкт-Петербург

2024

Задача №I «Машинки»

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

using namespace std;
int totOperations = 0;
map<int, int> floorMachines;
void addReq(int machineType, int requestIndex, vector<list<int>>& machineInd) {
  machineInd[machineType].push_back(requestIndex);
}
void createReq(int totMachineTypes, int maxFloorMachines, int numOfRequests,
vector<int>& requests, vector<list<int>>& machineInd) {
  for (int currRequest = 0; currRequest < numOfRequests; ++currRequest) {
    int currMachineType = requests[currRequest];
    auto machineIter = floorMachines.find(machineInd[currMachineType].front());
    if (machineIter != floorMachines.end()) {
      floorMachines.erase(machineIter);
    } else {
      if (floorMachines.size() == maxFloorMachines) {
        auto machineIter = floorMachines.end();
        --machineIter;
        floorMachines.erase(machineIter);
      }
      ++totOperations;
    }
    machineInd[currMachineType].pop_front();
    if (!machineInd[currMachineType].empty()) {
      floorMachines[machineInd[currMachineType].front()] = currMachineType + 1;
    }
  }
}
int main() {
  int totMachineTypes, maxFloorMachines, numOfRequests;
  cin >> totMachineTypes >> maxFloorMachines >> numOfRequests;
  vector<int> requests(numOfRequests);
  vector<list<int>> machineInd(totMachineTypes);
  for (int i = 0; i < numOfRequests; ++i) {
    cin >> requests[i];
    requests[i]--;
```

```
    addReq(requests[i], i, machineInd);
  }
  createReq(totMachineTypes, maxFloorMachines, numOfRequests, requests,
machineInd);
  cout << totOperations;
  return 0;
}
```

Пояснение к примененному алгоритму:

Цель: определить минимальное количество операций, которые необходимо совершить маме, чтобы Петя мог поиграть со всеми машинками в указанном порядке.

Алгоритм: Этот алгоритм решает задачу оптимизации операций по перемещению машинок с полки на пол и наоборот, чтобы удовлетворить запросы Пети с минимальным количеством операций. В данном алгоритме используется жадная стратегия, основанная на приоритетной очереди.

Программа эффективно решает задачу за счет использования структуры данных map для хранения машинок на полке и списков для хранения индексов запросов для каждого типа машинок.

Алгоритмическая сложность программы составляет O(P*log(K)), где P - количество запросов Пети, K - максимальное количество машинок на полу.

Что делает программа:

1. Сначала считывает параметры N (общее кол-во машинок), K (макс. кол-во машинок на полу) и P (общее кол-во запросов Пети).
2. Затем считывает последовательность запросов Пети.
3. Создает вектор запросов и список списков (machineInd), в котором каждый внутренний список представляет тип машинки, а элементы этого списка - индексы запросов для соответствующего типа машинок.
4. Происходит обработка запросов:
    - Проверяется, есть ли машинка нужного типа на полу.
    - Если машинка на полу есть, она убирается.
    - Если машинка на полке и место на полу закончилось, выбирается машинка с пола для замены.
    - Если машинка на полке и есть место на полу, она забирается, а на ее место ставится машинка с пола, которая будет использоваться в следующем запросе.
5. После обработки всех запросов выводится общее кол-во операций, которые совершила мама.

Задача №J «Гоблины и очереди»

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

using namespace std;

class GoblinQueue {
public:
    vector<int> goblins;

    void addGoblin(int goblinCount) {
        goblins.push_back(goblinCount);
    }

    void addPrivGoblin(int goblinCount) {
        int insertIndex = (goblins.size() + 1) / 2;
        goblins.insert(goblins.begin() + insertIndex, goblinCount);
    }
    int removeFirstGoblin() {
        int firstGoblin = goblins.front();
        goblins.erase(goblins.begin());
        return firstGoblin;
    }
};

int main() {
    int numOfQuery;
    cin >> numOfQuery;
    GoblinQueue queue;
    for (int i = 0; i < numOfQuery; i++) {
        char queryType;
        cin >> queryType;

        if (queryType == '+') {
            int goblinCount;
            cin >> goblinCount;
            queue.addGoblin(goblinCount);
        } else if (queryType == '*') {
            int goblinCount;
            cin >> goblinCount;
            queue.addPrivGoblin(goblinCount);
        } else if (queryType == '-') {
            cout << queue.removeFirstGoblin() << endl;
        }
    }
    return 0;
}
```

Пояснение к примененному алгоритму:

Программа представляет собой реализацию структуры данных "очередь гоблинов" с помощью класса GoblinQueue.

Алгоритм работы программы:

1. Программа считывает количество запросов (numOfQuery).

2. Далее, в цикле обрабатываются запросы:

- Если запрос начинается с символа '+', то считывается количество гоблинов и они добавляются в конец очереди.

- Если запрос начинается с символа '*', то считывается количество гоблинов и они добавляются в середину очереди.

- Если запрос начинается с символа '-', то извлекается и выводится первый гоблин из очереди.

Программа позволяет добавлять и удалять гоблинов из очереди, а также добавлять их в середину. Выводится первый гоблин из очереди при запросе на удаление.

Алгоритмическая сложность:

Общая алгоритмическая сложность программы зависит от числа операций, которые выполняются в цикле обработки запросов. В данной программе все операции выполняются над вектором goblins. Пусть N - общее количество запросов, тогда:

- Для каждого запроса добавления (+), сложность O(1).
- Для каждого запроса добавления привилегированного гоблина (*), сложность O(N).
- Для каждого запроса удаления (-), сложность O(N).

Так как каждый из N запросов может потребовать O(N) операций (для вставки привилегированного гоблина и удаления гоблина), общая алгоритмическая сложность в худшем случае составит O(N^2).

Подход, основанный на вставке в середину вектора, как это делается для привилегированных гоблинов, может стать более затратным по времени по мере увеличения размера вектора, так как это потребует сдвига большого количества элементов.

Задача №K «Менеджер памяти-1»

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

using namespace std;
const int INIT_MEM_BLOCK_START = 1;
const int QUERY_NEGATIVE_OFFSET = -1;

struct MemoryBlock {
    int start;
    int end;
    MemoryBlock(int s, int e) : start(s), end(e) {}
};

struct MemManager {
    std::map<int, MemoryBlock> startEnd;
    std::multimap<int, int> sizeStart;

    void releaseMemBySize(const map<int, MemoryBlock>::iterator &it) {
        auto itSizes = sizeStart.find(it->second.end - it->second.start);
        while (itSizes != sizeStart.end() && itSizes->second != it->first) {
            ++itSizes;
        }
        sizeStart.erase(itSizes);
        startEnd.erase(it);
    }

    void releaseMemory(const multimap<int, int>::iterator &it) {
        startEnd.erase(it->second);
        sizeStart.erase(it);
    }

    void allocMem(const MemoryBlock &el) {
        startEnd.emplace(el.start, el);
        sizeStart.emplace(el.end - el.start, el.start);
    }
};

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

    MemManager memManager;

    int totalMem, queries;
    cin >> totalMem >> queries;
    vector<pair<int, int>> queryRes(queries);
    memManager.allocMem({INIT_MEM_BLOCK_START, totalMem});

    for (int i = 0; i < queries; ++i) {
        int query;
        cin >> query;

        if (query < 0) {
            query *= QUERY_NEGATIVE_OFFSET;
            query--;
            if (queryRes[query].first < 0) continue;
            int startRange = queryRes[query].first;
            int sizeRange = queryRes[query].second;
```

```
            int endRange = startRange + sizeRange;

            auto nextEl = memManager.startEnd.find(endRange);
            auto prevEl =
std::prev(memManager.startEnd.upper_bound(startRange));
            if(prevEl == memManager.startEnd.end() || prevEl->second.end !=
startRange)
                prevEl = memManager.startEnd.end();

            if (nextEl != memManager.startEnd.end() &&
                nextEl->second.start == endRange) {
                if (prevEl != memManager.startEnd.end() &&
                    prevEl->second.end == startRange) {
                    int newStart = prevEl->second.start;
                    int newEnd = nextEl->second.end;
                    (memManager.releaseMemBySize(prevEl),
memManager.releaseMemBySize(nextEl));
                    memManager.allocMem({newStart, newEnd});
                } else {
                    int newEnd = nextEl->second.end;
                    memManager.releaseMemBySize(nextEl);
                    memManager.allocMem({startRange, newEnd});
                }
            } else {
                if (prevEl != memManager.startEnd.end() &&
                    prevEl->second.end == startRange) {
                    int newStart = prevEl->second.start;
                    memManager.releaseMemBySize(prevEl);
                    memManager.allocMem({newStart, endRange});
                } else
                    memManager.allocMem({startRange, endRange});
            }
        } else {
            auto iter = memManager.sizeStart.lower_bound(query);
            while (iter != memManager.sizeStart.end() && iter->first < query) {
                ++iter;
            }
            if (iter == memManager.sizeStart.end()) {
                queryRes[i] = {-1, query};
                cout << -1 << "\n";
            } else {
                cout << iter->second << "\n";
                int newStart = iter->second + query;
                int newEnd = iter->first + iter->second;
                queryRes[i] = {iter->second, query};
                memManager.releaseMemory(iter);
                memManager.allocMem({newStart, newEnd});
            }
        }
    }
    return 0;
}
```

Пояснение к примененному алгоритму:

Этот код реализует менеджер памяти для обработки запросов на выделение и освобождение памяти.

Обработка запросов на выделение памяти:

При поступлении запроса на выделение памяти программа ищет блок памяти, который подходит по размеру и находится в свободном состоянии.

Если такой блок найден, программа выделяет его для приложения и обновляет данные о состоянии памяти.

Если блок не найден, программа возвращает -1, указывая на отклонение запроса.

Обработка запросов на освобождение памяти:

При поступлении запроса на освобождение памяти программа освобождает блок памяти, выделенный ранее для соответствующего запроса.

Если вокруг освобождаемого блока есть другие свободные блоки, программа объединяет их в один, чтобы максимизировать использование памяти.

Алгоритмическая сложность:

Запрос на выделение:

- Поиск свободного блока: O(log N) - поиск в sizeStart с помощью lower_bound().
- Объединение/обновление блоков: O(log N) - операции в startEnd и sizeStart.

Запрос на освобождение:

- Поиск блока: O(log N) - поиск в startEnd с помощью find().
- Обновление контейнеров: O(log N) - операции в startEnd и sizeStart.

Основные шаги программы:

- Инициализация начального блока памяти.
- Чтение запросов из входного файла.
- Обработка каждого запроса согласно описанной логике.
- Вывод результата в соответствии с условиями задачи.

Задача №L «Минимум на отрезке»

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

using namespace std;

void build_tree(int arr[], vector<int>& seg_tree, int node, int left, int right)
{
    if (left == right) {
        seg_tree[node] = arr[left];
    } else {
        int mid = (left + right) / 2;
        build_tree(arr, seg_tree, 2 * node, left, mid);
        build_tree(arr, seg_tree, 2 * node + 1, mid + 1, right);
        seg_tree[node] = min(seg_tree[2 * node], seg_tree[2 * node + 1]);
    }
}

int query_tree(vector<int>& seg_tree, int node, int left, int right, int q_left,
int q_right) {
    if (q_left > right || q_right < left) {
        return numeric_limits<int>::max();
    }
    if (q_left <= left && right <= q_right) {
        return seg_tree[node];
    }
    int mid = (left + right) / 2;
    return min(query_tree(seg_tree, 2 * node, left, mid, q_left, q_right),
               query_tree(seg_tree, 2 * node + 1, mid + 1, right, q_left,
q_right));
}

int main() {
    int n, k;
    cin >> n >> k;

    vector<int> seq(n);
    for (int i = 0; i < n; ++i) {
        cin >> seq[i];
    }

    int seg_tree_size = 1;
    while (seg_tree_size < n) {
        seg_tree_size *= 2;
    }
    seg_tree_size *= 2;

    vector<int> seg_tree(seg_tree_size);

    build_tree(&seq[0], seg_tree, 1, 0, n - 1);

    for (int i = 0; i <= n - k; ++i) {
        cout << query_tree(seg_tree, 1, 0, n - 1, i, i + k - 1) << " ";
    }

    return 0;
}
```

Пояснение к примененному алгоритму:

Данный фрагмент кода представляет собой реализацию построения и использования сегментного дерева для выполнения запросов на поиск минимума на отрезке. Программа считывает массив, строит дерево отрезков и затем выполняет запросы на поиск минимума для всех подотрезков длины k. Полученные минимумы выводятся на экран.

Алгоритм работы программы:

- Построение дерева отрезков (build_tree):
  На этом этапе для каждого узла дерева определяется минимальное значение на соответствующем отрезке массива. Рекурсивно вызывается функция build_tree, которая проходит по всем узлам дерева и заполняет их значениями. В основе лежит стратегия "разделяй и властвуй", где массив разбивается на две части, для каждой из которых строится дерево, а затем значения сливаются наверх.
- Запросы на поиск минимума (query_tree):
  Эта функция позволяет находить минимальное значение на заданном подотрезке массива. Она рекурсивно спускается по дереву отрезков, чтобы найти необходимый отрезок и вернуть минимальное значение. Если запрашиваемый отрезок полностью содержится в текущем узле дерева, возвращается значение этого узла. Иначе функция вызывается для левого и правого поддеревьев.

Алгоритмическая сложность:

- Построение сегментного дерева: O(n), где n - количество элементов в последовательности. O(n) - древообразный обход массива.
- Запрос на поиск минимума на отрезке: O(log n), где n - количество элементов в последовательности. O(log n) - логарифмическая глубина сегментного дерева.