Университет ИТМО

Факультет программной инженерии и компьютерной техники

«Алгоритмы и структуры данных»

**отчет по блоку задач №3 (Яндекс.Контест)**

**Выполнил:**

Студент группы P3210

Федоров Евгений Константинович

**Преподаватель:**

Тараканов Денис Сергеевич

Санкт-Петербург, 2025г.

**Задача №1 «J. Гоблины и очереди»**

**Пояснение к примененному алгоритму:**

Задача состоит в том, чтобы за константное время добавлять гоблинов в середину очереди, реализовать это можно, используя двусвязный список как структуру данных, поддерживая итератор на середине списка. Если текущая операции – добавление гоблина в конец очереди, то сдвигаем итератор на 1 только в том случае, если количество гоблинов нечетное. В случае добавления в середину очереди, добавляем гоблина и итератор икрементируем только если количество гоблинов до добавления было четным, то есть указатель будет ровно на середине. В случае удаления так же двигаем итератор, если до удаления из очереди итератор указывал на начало списка, важно чтобы он и дальше на него указывал. Если количество гоблинов после удаления стало нечетным, нужно увеличить итератор.

**Алгоритмическая сложность:**

* **По времени**: Алгоритм выполняется со скорость **O(n)**
* **По памяти**: Вектор хранит N строк, список в худшем случае так же хранит N записей. Сложность **O(n)**

Код:

|  |
| --- |
| #include <cmath>  #include <iostream>  #include <queue>  #include <vector>  using namespace std;  int main() {  int places, cows;  cin >> places >> cows;  vector<int> v(places);  for (int i = 0; i < places; i++) {  cin >> v[i];  }  int best = 0;  int minRange = 0, maxRange = v.back() - v.front(), average;  while (minRange <= maxRange) {  average = (maxRange + minRange) / 2;  int remainingCows = cows - 1;  int lastPlacesCow = v[0];  for (size\_t i = 0; i < v.size(); i++) {  if (abs(v[i] - lastPlacesCow) >= average) {  lastPlacesCow = v[i];  remainingCows--;  }  }  if (remainingCows <= 0) {  minRange = average + 1;  best = average;  } else {  maxRange = average - 1;  }  }  cout << best << endl;  return 0;  } |

**Задача №2 «K. Менеджер памяти-1»**

**Пояснение к примененному алгоритму:**

Основная идея алгоритма заключается в том, чтобы быстро находить подходящий свободный блок при и выделении и эффективно объединять освобождаемые блоки с соседними. Для этого имеется две мапы, первая из которых хранит информацию о начале и конце блока, а вторая хранит информацию о размере блоков памяти. При запросе на выделение памяти смотрим, был ли уже такой запрос ранее(в operation\_history), если нет, то склеиваем соседние блоки и удаляем старый при помощи remove\_chunk и добавляем новый при помощи register\_chunk. При запросе на выделение находим первый блок с необходимым размером, если не найден, записываем -1. Если блок нашелся, выделяем нужное количество ячеек и остаток, записываем в соответствующие структуры данных и возвращаем начало выделившегося блока.

**Алгоритмическая сложность:**

1. **По времени:** Сложность по времени составляет **O(log n),** так как используются сбалансированные деревья.
2. **По памяти:** По памяти алгоритм имеет сложность **O(n)**, так как в худшем случае будет храниться информация о N блоках

Код:

|  |
| --- |
| #include <algorithm>  #include <iostream>  #include <map>  #include <vector>  struct ChunkMatcher {  int target;  bool operator()(const std::pair<const int, int>& p) const {  return p.second == target;  }  };  class MemoryHandler {  std::map<int, int> memory\_chunks;  std::multimap<int, int> size\_map;  std::vector<std::pair<int, int>> operation\_history;  void remove\_chunk(std::map<int, int>::iterator chunk) {  auto [begin, end] = size\_map.equal\_range(chunk->second - chunk->first);  auto found = std::find\_if(begin, end, ChunkMatcher{chunk->first});  if (found != end)  size\_map.erase(found);  memory\_chunks.erase(chunk);  }  void remove\_by\_size(std::multimap<int, int>::iterator size\_entry) {  memory\_chunks.erase(size\_entry->second);  size\_map.erase(size\_entry);  }  void register\_chunk(int start, int end) {  memory\_chunks[start] = end;  size\_map.insert({end - start, start});  }  public:  MemoryHandler(int capacity, int max\_ops) : operation\_history(max\_ops) {  register\_chunk(1, capacity + 1);  }  void handle\_request(int op\_index, int req\_value) {  if (req\_value < 0) {  int alloc\_index = -req\_value - 1;  if (operation\_history[alloc\_index].first < 0)  return;  auto [base, length] = operation\_history[alloc\_index];  int limit = base + length;  auto next = memory\_chunks.lower\_bound(limit);  auto prev = next != memory\_chunks.begin() ? std::prev(next) : memory\_chunks.end();  if (next != memory\_chunks.end() && next->first == limit) {  if (prev != memory\_chunks.end() && prev->second == base) {  int new\_base = prev->first;  int new\_limit = next->second;  remove\_chunk(prev);  remove\_chunk(next);  register\_chunk(new\_base, new\_limit);  } else {  int new\_limit = next->second;  remove\_chunk(next);  register\_chunk(base, new\_limit);  }  } else {  if (prev != memory\_chunks.end() && prev->second == base) {  int new\_base = prev->first;  remove\_chunk(prev);  register\_chunk(new\_base, limit);  } else {  register\_chunk(base, limit);  }  }  } else {  auto found = size\_map.lower\_bound(req\_value);  if (found == size\_map.end()) {  operation\_history[op\_index] = {-1, req\_value};  std::cout << -1 << '\n';  } else {  std::cout << found->second << '\n';  int new\_base = found->second + req\_value;  int new\_limit = found->first + found->second;  operation\_history[op\_index] = {found->second, req\_value};  remove\_by\_size(found);  register\_chunk(new\_base, new\_limit);  }  }  }  };  int main() {  int N, M;  std::cin >> N >> M;  MemoryHandler alloc\_system(N, M);  for (int i = 0; i < M; ++i) {  int cmd;  std::cin >> cmd;  alloc\_system.handle\_request(i, cmd);  }  return 0;  } |

**Задача №3 «L. Минимум на отрезке»**

**Пояснение к примененному алгоритму:**

Используем структуру данных deque, так как время на вставку константное. В передней части всегда храним индекс минимального элемента в окне, на каждой итерации цикла удаляем все индексы в очереди, элементы по которым меньше текущего. Как только индекс на лицевой части очереди выходи за окно, удаляем его.

**Алгоритмическая сложность:**

1. **По времени:** **O(n)**, так как работа со всеми элементами займет в худшем случае n итераций.
2. **По памяти:** **O(n),** так как может хранить n элементов в худшем случае.

Код:

|  |
| --- |
| #include <deque>  #include <iostream>  #include <vector>  using namespace std;  int main() {  int n, window;  cin >> n >> window;  vector<int> nums(n);  for (size\_t i = 0; i < nums.size(); ++i) {  cin >> nums[i];  }  deque<int> deque;  vector<int> result;  for (int i = 0; i < window; ++i) {  while (!deque.empty() && nums[i] <= nums[deque.back()]) {  deque.pop\_back();  }  deque.push\_back(i);  }  result.push\_back(nums[deque.front()]);  for (int i = window; i < n; ++i) {  if (!deque.empty() && deque.front() <= i - window) {  deque.pop\_front();  }  while (!deque.empty() && nums[i] <= nums[deque.back()]) {  deque.pop\_back();  }  deque.push\_back(i);  result.push\_back(nums[deque.front()]);  }  for (int val : result) {  cout << val << " ";  }  return 0;  } |