Министерство образования и науки Российской Федерации федеральное государственное
автономное образовательное учреждение высшего образования
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Факультет «Программной инженерии и компьютерной техники.»

Алгоритмы и структуры данных

Лабораторная работа №3
Базовые задачи

Выполнил
Григорьев Давид Владимирович
Группа: P3215
Преподаватели
Косяков Михаил Сергеевич
Тараканов Денис Сергеевич

Санкт-Петербург 2025

Содержание

1 Задача I. Машинки

Основная идея

Мама Пети действует как "умный кэш":

- Одновременно на полу может быть до K машинок
- Если Петя просит машинку, которой нет на полу — это «промах» (требуется замена)
- При замене убираем ту машинку, которая понадобится **позже всех**

Шаги решения

1. **Предобработка:** Для каждой машинки запоминаем, когда она понадобится в следующий раз 2. **Симуляция:** - Если машинка уже на полу — обновляем её приоритет - Если места нет — удаляем машинку с самым дальним следующим использованием

Сложность

- **Время:** $O(P \log K)$
  - Обработка P запросов с операциями вставки/удаления за $O(\log K)$
- **Память:** $O(P + K)$
  - Хранение списка запросов (P) и текущих машинок на полу (K)

Код на C++

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

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

  size_t total_unique_items = 0;
  size_t cache_size = 0;
  size_t num_accesses = 0;
  std::cin >> total_unique_items >> cache_size >> num_accesses;

  std::vector<int> page_requests(num_accesses);
  for (size_t i = 0; i < num_accesses; ++i) {
    std::cin >> page_requests[i];
  }

  std::vector<int> next_occurrence_indices(num_accesses, static_cast<int>(num_accesses));
  std::unordered_map<int, int> last_occurrence_map;

  // Use reverse loop with size_t safe idiom
  for (size_t i = num_accesses; i-- > 0;) {
    int current_page = page_requests[i];
    if (last_occurrence_map.find(current_page) != last_occurrence_map.end()) {
```

```
      next_occurrence_indices[i] = last_occurrence_map[current_page];
    } else {
      next_occurrence_indices[i] = static_cast<int>(num_accesses);
    }
    last_occurrence_map[current_page] = static_cast<int>(i);
  }

  std::unordered_map<int, int> element_next_occurrence_map;
  std::set<std::pair<int, int>, std::greater<std::pair<int, int>>> eviction_priority_set

  int cache_misses = 0;

  for (size_t i = 0; i < num_accesses; ++i) {
    int current_page = page_requests[i];
    if (element_next_occurrence_map.find(current_page) != element_next_occurrence_map.en
      int previous_next_occurrence = element_next_occurrence_map[current_page];
      eviction_priority_set.erase({previous_next_occurrence, current_page});
      int updated_next_occurrence = next_occurrence_indices[i];
      eviction_priority_set.insert({updated_next_occurrence, current_page});
      element_next_occurrence_map[current_page] = updated_next_occurrence;
    } else {
      cache_misses++;
      if (element_next_occurrence_map.size() < cache_size) {
        int updated_next_occurrence = next_occurrence_indices[i];
        element_next_occurrence_map[current_page] = updated_next_occurrence;
        eviction_priority_set.insert({updated_next_occurrence, current_page});
      } else {
        auto it = eviction_priority_set.begin();
        int evicted_page = it->second;
        eviction_priority_set.erase(it);
        element_next_occurrence_map.erase(evicted_page);
        int updated_next_occurrence = next_occurrence_indices[i];
        element_next_occurrence_map[current_page] = updated_next_occurrence;
        eviction_priority_set.insert({updated_next_occurrence, current_page});
      }
    }
  }

  std::cout << cache_misses << '\n';

  return 0;
}
```

2 Задача J. Гоблины и очереди

Основная идея

Очередь разделена на две части:

- `frontSection` — первая половина (или на 1 больше при нечётной длине)
- `backSection` — вторая половина

Операции:

- **+** (добавить в конец) → в `backSection`

- * (добавить в середину) → в конец `frontSection`
- - (удалить первого) → из `frontSection` (если не пуст)

После каждой операции происходит балансировка — перераспределение элементов между деками.

Шаги решения

1. Добавление: - Обычный гоблин → `backSection.push_back()` - Привилегированный → `frontSection.push_back()` 2. Удаление: - Берём первого из `frontSection` (или `backSection`, если пуст) 3. Балансировка: - Поддерживаем размер `frontSection` (общая длина + 1)/2 - При необходимости перемещаем элементы между деками

Сложность

- **Время:** $O(N)$
  - Все операции с деками и балансировка — $O(1)$ в среднем
- **Память:** $O(N)$

Код на C++

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

struct BalancedDeque {
  std::deque<int> frontSection, backSection;

  void BalanceSections() {
    const size_t total_elements = frontSection.size() + backSection.size();
    const size_t desired_front_size = (total_elements + 1) / 2;

    if (frontSection.size() > desired_front_size) {
      backSection.push_front(frontSection.back());
      frontSection.pop_back();
    } else if (frontSection.size() < desired_front_size && !backSection.empty()) {
      frontSection.push_back(backSection.front());
      backSection.pop_front();
    }
  }

  void PushMiddle(int value) {
    frontSection.push_back(value);
    BalanceSections();
  }

  void PushBack(int value) {
    backSection.push_back(value);
    BalanceSections();
  }

  void PopFront() {
    if (!frontSection.empty()) {
      frontSection.pop_front();
```

```
    } else {
      backSection.pop_front();
    }
    BalanceSections();
  }

  [[nodiscard]] int Front() const {
    return !frontSection.empty() ? frontSection.front() : backSection.front();
  }
};

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

  int number_of_commands = 0;
  std::cin >> number_of_commands;

  BalancedDeque balanced_deque;

  while ((number_of_commands--) != 0) {
    char command = 0;
    std::cin >> command;

    if (command == '-') {
      std::cout << balanced_deque.Front() << '\n';
      balanced_deque.PopFront();
    } else {
      int value = 0;
      std::cin >> value;
      if (command == '+') {
        balanced_deque.PushBack(value);
      } else {
        balanced_deque.PushMiddle(value);
      }
    }
  }

  return 0;
}
```

3 Задача K. Менеджер памяти

Пояснение к примененному алгоритму

Алгоритм реализует менеджер памяти с использованием:

- **Кучи (max-heap)** для хранения свободных блоков, отсортированных по размеру (в порядке убывания).
- **Двусвязного списка** для отслеживания соседних блоков и быстрого объединения при освобождении памяти.

Обработка запросов:

- **Выделение памяти:**
  - Из кучи извлекается наибольший свободный блок.
  - Если блок достаточно большой, он разделяется на выделенную часть и остаток (который возвращается в кучу).
  - Если подходящего блока нет, запрос отклоняется.
- **Освобождение памяти:**
  - Освобождаемый блок помечается как свободный.
  - Проверяются соседние блоки (предыдущий и следующий). Если они свободны, происходит объединение в один блок.
  - Объединенный блок добавляется в кучу.

Сложность по времени:

- $O(M \log M)$ — для M запросов.
  - Операции вставки/удаления в куче: $O(\log K)$, где K — количество свободных блоков.
  - Объединение блоков при освобождении: $O(1)$ за счет связных списков.

Сложность по памяти:

- $O(M)$ — хранение информации о запросах и блоках:
  - Векторы `request_status`, `blocks_for_requests`: $O(M)$.
  - Куча свободных блоков: $O(M)$ в худшем случае.

Код алгоритма

```cpp
#include <bits/stdc++.h>

#include <cstddef>
#include <memory>

using BlockIndex = size_t;

enum class Status { Declined = 0, Allocated = 1, Removed = 2 };

class Block {
public:
  bool is_free = true;
```

```
  size_t start, end;

  BlockIndex index;

  Block *prev, *next;

  Block(Block* prev, Block* next, bool free, size_t start, size_t end, BlockIndex index)
      : is_free(free), start(start), end(end), index(index), prev(prev), next(next) {
    if (prev != nullptr) {
      prev->next = this;
    }
    if (next != nullptr) {
      next->prev = this;
    }
  }

  void Remove() {
    if (prev != nullptr) {
      prev->next = next;
    }
    if (next != nullptr) {
      next->prev = prev;
    }
  }

  size_t Size() const {
    return end - start;
  }
};

class Heap {
public:
  size_t current_request{};
  size_t heap_size{};

  std::vector<Status> request_status;
  std::vector<std::unique_ptr<Block>> heap;
  std::vector<std::unique_ptr<Block>> blocks_for_requests;

  Heap(size_t n, size_t m) : heap_size(1) {
    request_status.resize(m);
    blocks_for_requests.resize(m);

    heap.resize(m);
    heap[0] = std::make_unique<Block>(nullptr, nullptr, true, 0, n, 0);
  }

  void Allocate(size_t request_size) {
    if (heap_size == 0 || heap[0]->Size() < request_size) {
      request_status[current_request++] = Status::Declined;
      std::cout << "-1" << '\n';
      return;
    }

    Block* root = heap[0].get();
```

```
  request_status[current_request++] = Status::Allocated;
  blocks_for_requests[current_request - 1] = std::make_unique<Block>(
      root->prev, root, false, root->start, root->start + request_size, SIZE_MAX
  );

  std::cout << root->start + 1 << '\n';

  root->start += request_size;
  if (root->start < root->end) {
    Heapify(root->index);
  } else {
    root->Remove();
    Pop();
    if (heap_size > 0) {
      heap[heap_size].reset();
    }
  }
}

void Free(size_t request_index) {
  request_index--;

  request_status[current_request++] = Status::Removed;

  if (request_status[request_index] == Status::Declined) {
    return;
  }

  request_status[request_index] = Status::Removed;

  auto& block_ptr = blocks_for_requests[request_index];
  Block* block = block_ptr.get();
  Block* prev_block = block->prev;
  Block* next_block = block->next;

  if (((prev_block == nullptr) || !prev_block->is_free) &&
      ((next_block == nullptr) || !next_block->is_free)) {
    block->is_free = true;
    block->index = heap_size;
    heap[heap_size] = std::move(block_ptr);
    Lift(heap_size++);
    return;
  }
  if ((prev_block == nullptr) || !prev_block->is_free) {
    next_block->start = block->start;
    Lift(next_block->index);
    block->Remove();
    block_ptr.reset();
    return;
  }
  if ((next_block == nullptr) || !next_block->is_free) {
    prev_block->end = block->end;
    Lift(prev_block->index);
    block->Remove();
```

```
      block_ptr.reset();
      return;
    }

    prev_block->end = next_block->end;
    Lift(prev_block->index);

    block->Remove();
    block_ptr.reset();

    next_block->Remove();
    Remove(next_block->index);
    heap[heap_size].reset();
  }

  void Dispatch(int request) {
    if (request > 0) {
      Allocate(static_cast<size_t>(request));
    } else {
      Free(static_cast<size_t>(-request));
    }
  }

private:
  static BlockIndex GetParentIndex(BlockIndex index) {
    return (index - 1) / 2;
  }

  static BlockIndex GetLeftChildIndex(BlockIndex index) {
    return (2 * index) + 1;
  }

  static BlockIndex GetRightChildIndex(BlockIndex index) {
    return (2 * index) + 2;
  }

  void Swap(BlockIndex index1, BlockIndex index2) {
    std::swap(heap[index1], heap[index2]);
    heap[index1]->index = index1;
    heap[index2]->index = index2;
  }

  bool Better(BlockIndex index1, BlockIndex index2) const {
    return heap[index1]->Size() > heap[index2]->Size();
  }

  void Heapify(BlockIndex index) {
    while (true) {
      BlockIndex largest = index;
      BlockIndex left_child = GetLeftChildIndex(index);
      BlockIndex right_child = GetRightChildIndex(index);

      if ((left_child < heap_size) && Better(left_child, largest)) {
        largest = left_child;
      }
```

```
      if ((right_child < heap_size) && Better(right_child, largest)) {
        largest = right_child;
      }
      if (index == largest) {
        return;
      }

      Swap(index, largest);
      index = largest;
    }
  }

  void Pop() {
    if (heap_size == 0) {
      return;
    }
    heap_size--;
    if (heap_size > 0) {
      Swap(0, heap_size);
      Heapify(0);
    }
  }

  void Lift(BlockIndex index) {
    while (index != 0 && Better(index, GetParentIndex(index))) {
      BlockIndex parent = GetParentIndex(index);
      Swap(index, parent);
      index = parent;
    }
  }

  void Remove(BlockIndex index) {
    if (index >= heap_size) {
      return;
    }
    Swap(index, heap_size - 1);
    heap_size--;
    if (index < heap_size) {
      Lift(index);
      Heapify(index);
    }
  }
};

int main() {
  size_t heap_size = 0;
  size_t number_of_mem_operations = 0;
  std::cin >> heap_size >> number_of_mem_operations;

  Heap heap(heap_size, number_of_mem_operations);

  int request = 0;
  for (size_t i = 0; i < number_of_mem_operations; i++) {
    std::cin >> request;
    heap.Dispatch(request);
```

```
  }

  return 0;
}
```

4 Задача L. Минимум на отрезке

Пояснение к примененному алгоритму

Для эффективного поиска минимума в скользящем окне используется дек. В деке хранятся индексы элементов текущего окна в порядке возрастания их значений. При движении окна:

- Удаляются элементы, выходящие за границы окна (с начала дека).
- Новый элемент добавляется в дек после удаления всех элементов, больших или равных ему (с конца дека).
- Минимумом текущего окна всегда является элемент в начале дека.

Сложность по времени:

- $O(N)$ — каждый элемент последовательности добавляется в дек и удаляется из него не более одного раза.
  - Обработка первого окна: $O(K)$
  - Обработка оставшихся $N - K$ элементов: $O(N)$

Сложность по памяти:

- $O(N)$ — хранение исходного массива `arr`.
- $O(K)$ — дек для хранения индексов элементов текущего окна.
- Итого: $O(N)$ (доминирует размер массива).

Код алгоритма

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

int main() {
  size_t number_of_elements = 0;
  std::cin >> number_of_elements;

  size_t size_of_sliding_window = 0;
  std::cin >> size_of_sliding_window;

  std::vector<int> arr(number_of_elements);
  std::deque<size_t> window;

  for (size_t i = 0; i < number_of_elements; ++i) {
    std::cin >> arr[i];
  }

  for (size_t i = 0; i < size_of_sliding_window; ++i) {
    while (!window.empty() && arr[i] <= arr[window.back()]) {
      window.pop_back();
    }
    window.push_back(i);
  }
```

```
  std::cout << arr[window.front()] << ' ';

  for (size_t i = size_of_sliding_window; i < number_of_elements; ++i) {
    if (!window.empty() && window.front() <= i - size_of_sliding_window) {
      window.pop_front();
    }
    while (!window.empty() && arr[i] <= arr[window.back()]) {
      window.pop_back();
    }
    window.push_back(i);

    std::cout << arr[window.front()] << ' ';
  }

  std::cout << '\n';
  return 0;
}
```