Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №3

по «Алгоритмам и структурам данных»

Базовые задачи

Выполнил:

Студент группы P3218

Хромов Даниил Тимофеевич

Преподаватели:

Косяков М.С.

Тараканов Д.С.

Санкт-Петербург

2025

Задача №9 «Машинки»

Пояснение к примененному алгоритму:

Для решения поставленной задачи была использована модификация алгоритма Белади (один из оптимальных алгоритмов кэширования).

В основных шагах мы можем выделить два пункта:

1. Предварительная обработка данных.

   Для каждого запроса в последовательности мы определяем следующий индекс, где эта машинка понадобится снова. Это позволяет нам предсказать, какую машинку выгоднее оставить на полу.

2. Обработка данных.

   Мы всегда должны поддерживать список машинок, находящихся на полу.

   Если запрашиваемая машинка уже на полу, обновляем информацию о ее следующем использовании.

   Если машинки нет на полу:

   1. Если есть свободное место, добавить её.
   2. Если места нет, заменить машинку, которая будет использована позже.

Код:

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

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  cin.tie(0);
  cout.tie(0);

  int N, P;
  size_t K;
  cin >> N >> K >> P;
```

```
vector<int> a(P);
for (int i = 0; i < P; ++i) {
  cin >> a[i];
}

vector<int> next_occurrence(P, P);
unordered_map<int, int> last_pos;
for (int i = P - 1; i >= 0; --i) {
  if (last_pos.find(a[i]) != last_pos.end()) {
    next_occurrence[i] = last_pos[a[i]];
  } else {
    next_occurrence[i] = P;
  }
  last_pos[a[i]] = i;
}

unordered_map<int, int> toy_next;
set<pair<int, int>> priority_set;
int operations = 0;

for (int i = 0; i < P; ++i) {
  int toy = a[i];
  if (toy_next.find(toy) != toy_next.end()) {
    auto old_pair = make_pair(-toy_next[toy], toy);
    auto it = priority_set.find(old_pair);
    if (it != priority_set.end()) {
      priority_set.erase(it);
    }
    int new_next = next_occurrence[i];
    toy_next[toy] = new_next;
    priority_set.insert(make_pair(-new_next, toy));
```

```
    } else {
      operations++;
      if (toy_next.size() < K) {
        int new_next = next_occurrence[i];
        toy_next[toy] = new_next;
        priority_set.insert(make_pair(-new_next, toy));
      } else {
        auto it = priority_set.begin();
        int removed_toy = it->second;
        priority_set.erase(it);
        toy_next.erase(removed_toy);
        int new_next = next_occurrence[i];
        toy_next[toy] = new_next;
        priority_set.insert(make_pair(-new_next, toy));
      }
    }
  }

  cout << operations << endl;

  return 0;
}
```

Задача №10 «Гоблины и очереди»

Пояснение к примененному алгоритму:

Мы разделил очередь на две части, используя два deque: first_half и second_half

1. Для каждой операции:
    - +: добавляем гоблина в конец second_half
    - *: добавляем привилегированного гоблина в начало second_half (что соответствует середине общей очереди)
    - -: удаляем и выводим первого гоблина из first_half
2. После каждой операции выполняем балансировку, чтобы:
    - first_half содержал первую половину очереди, а second_half содержал вторую половину;

- Длина first_half должна быть равна или на 1 больше длины second_half.

Такое разделение гарантирует, что середина очереди всегда находится на границе между двумя половинами, и мы можем эффективно вставлять туда привилегированных гоблинов.

Временная сложность: O(N), где N - количество запросов.

Пространственная сложность: O(N), так как в худшем случае все гоблины могут находиться в очереди одновременно.

Код:

```
#include <deque>
#include <iostream>
using namespace std;

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

  deque<int> first_half, second_half;

  for (int i = 0; i < n; i++) {
    char operation;
    cin >> operation;

    if (operation == '+') {
      int id;
      cin >> id;
      second_half.push_back(id);
    } else if (operation == '*') {
      int id;
      cin >> id;
      second_half.push_front(id);
    } else if (operation == '-') {
      cout << first_half.front() << endl;
```

```
    first_half.pop_front();
   }

   while (first_half.size() < second_half.size()) {
    first_half.push_back(second_half.front());
    second_half.pop_front();
   }

   if (first_half.size() > second_half.size() + 1) {
    second_half.push_front(first_half.back());
    first_half.pop_back();
   }
  }

  return 0;
}
```

Задача №11 «Менеджер памяти-1»

Пояснение к примененному алгоритму:

Основная идея решения состоит в эффективном отслеживании свободных блоков памяти и быстром поиске блоков подходящего размера. Для этого используются следующие структуры данных:

Карта свободных блоков map<int, int> free_blocks - отображает начальную позицию свободного блока на его конечную позицию.

Индекс по размерам map<int, set<int>> size_to_blocks - группирует блоки по размеру для быстрого поиска.

Журнал выделения map<int, pair<int, int>> allocations - хранит информацию о выделенных блоках (начало и размер) с привязкой к номеру запроса.

Временная сложность: O(M * log N), где M - число запросов, N - размер памяти.

Код:

```
#include <iostream>
```

```
#include <map>
#include <set>
#include <vector>

using namespace std;

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

  long long n;
  int m;
  cin >> n >> m;

  map<int, int> free_blocks;
  free_blocks[1] = n;

  map<int, set<int>> size_to_blocks;
  size_to_blocks[n].insert(1);

  map<int, pair<int, int>> allocations;

  vector<int> results;

  for (int i = 1; i <= m; i++) {
    int request;
    cin >> request;

    if (request > 0) {
      int size_needed = request;

      auto it = size_to_blocks.lower_bound(size_needed);
```

```
if (it == size_to_blocks.end()) {
  results.push_back(-1);
  continue;
}

int allocated_pos = -1;
for (int start : it->second) {
  if (start == 1 || free_blocks.find(start - 1) == free_blocks.end()) {
    allocated_pos = start;
    break;
  }
}

if (allocated_pos == -1) {
  results.push_back(-1);
  continue;
}

int block_size = it->first;
int block_start = allocated_pos;
int block_end = free_blocks[block_start];

size_to_blocks[block_size].erase(block_start);
if (size_to_blocks[block_size].empty()) {
  size_to_blocks.erase(block_size);
}
free_blocks.erase(block_start);

if (block_size > size_needed) {
  int new_start = block_start + size_needed;
  int new_size = block_size - size_needed;
  free_blocks[new_start] = block_end;
```

```
    size_to_blocks[new_size].insert(new_start);
  }

  allocations[i] = {block_start, size_needed};
  results.push_back(block_start);
 } else {
  int free_request = -request;

  if (allocations.find(free_request) == allocations.end()) {
    continue;
  }

  int start = allocations[free_request].first;
  int size = allocations[free_request].second;
  int end = start + size - 1;

  allocations.erase(free_request);

  auto next_block = free_blocks.find(end + 1);
  if (next_block != free_blocks.end()) {
    int next_end = next_block->second;
    int next_size = next_end - (end + 1) + 1;

    size_to_blocks[next_size].erase(end + 1);
    if (size_to_blocks[next_size].empty()) {
      size_to_blocks.erase(next_size);
    }
    free_blocks.erase(next_block);

    end = next_end;
    size = end - start + 1;
  }
```

```
    auto prev_it = free_blocks.upper_bound(start);
    if (prev_it != free_blocks.begin()) {
      prev_it--;
      int prev_start = prev_it->first;
      int prev_end = prev_it->second;

      if (prev_end + 1 == start) {
        int prev_size = prev_end - prev_start + 1;

        size_to_blocks[prev_size].erase(prev_start);
        if (size_to_blocks[prev_size].empty()) {
          size_to_blocks.erase(prev_size);
        }
        free_blocks.erase(prev_it);

        start = prev_start;
        size = end - start + 1;
      }
    }

    free_blocks[start] = end;
    size_to_blocks[size].insert(start);
  }
}

for (int result : results) {
  cout << result << "\n";
}

return 0;
```

```
}
```

Задача №12 «Минимум на отрезке»

Пояснение к примененному алгоритму:

В деке хранятся индексы элементов, а не сами элементы, что позволяет легко определять, какие элементы выходят из окна. Элементы в деке упорядочены по возрастанию их значений. Перед добавлением нового элемента из конца дека удаляются все элементы с большими значениями, так как они никогда не станут минимумами в будущих окнах, если в окно входит элемент с меньшим значением. Из начала дека удаляются индексы элементов, которые выходят за пределы текущего окна.

Временная сложность: O(N), где N – длина исходного массива. Каждый элемент массива добавляется в дек и удаляется из него не более одного раза.
Пространственная сложность: O(K), где K – размер окна. В худшем случае дек может содержать до K элементов.

Код:

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

using namespace std;

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

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

  deque<int> dq;
```

```
  for (int i = 0; i < k; i++) {
    while (!dq.empty() && arr[i] <= arr[dq.back()]) {
      dq.pop_back();
    }
    dq.push_back(i);
  }

  cout << arr[dq.front()];

  for (int i = k; i < n; i++) {
    cout << " ";

    while (!dq.empty() && dq.front() <= i - k) {
      dq.pop_front();
    }

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

    dq.push_back(i);

    cout << arr[dq.front()];
  }

  cout << endl;

  return 0;
}
```