Последовательные контейнеры

Обзор задач на LeetCode

Вектора

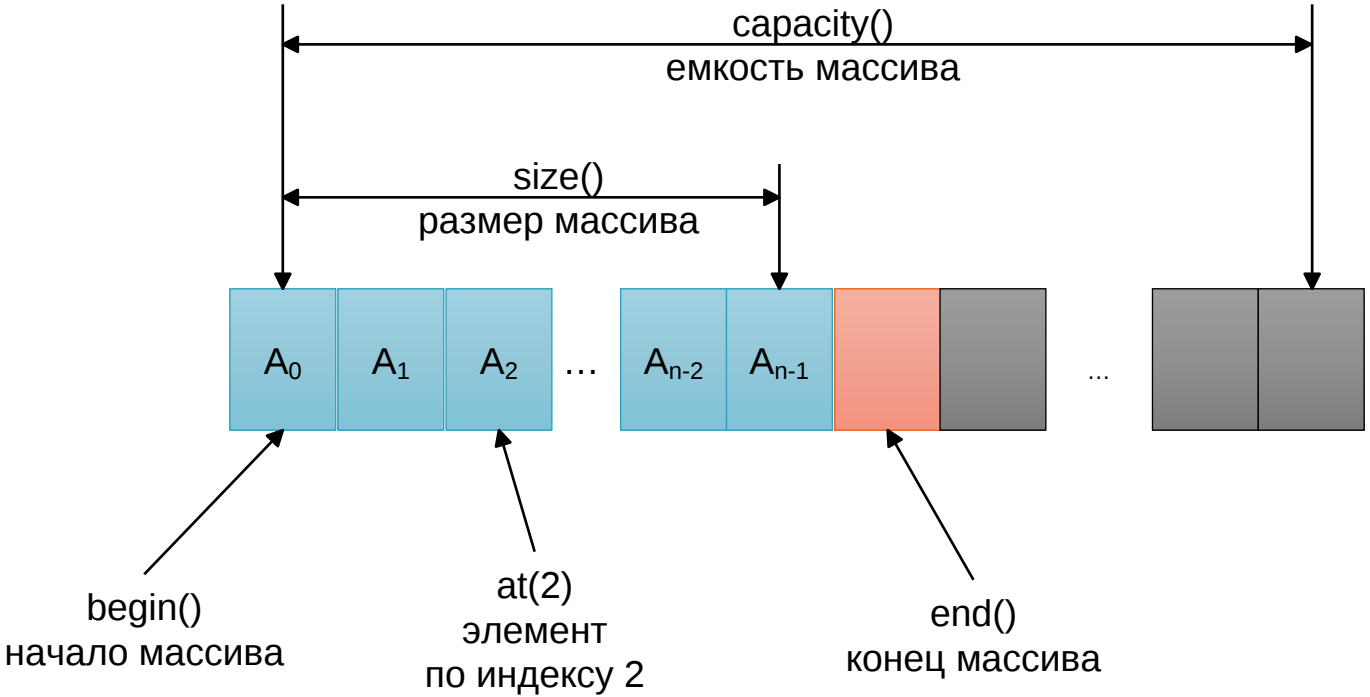
STL Вектор (std::vector)

В C++ вектор (**std::vector**) - это динамический массив, который хранит коллекцию элементов одного типа в непрерывной памяти.

Имеет возможность автоматически изменять свой размер при вставке или удалении элемента.

Реализует технологию RAII (парадигму управления ресурсами, которая означает, что ресурсы выделяются и освобождаются автоматически при создании и уничтожении объектов).

Описание std::vector



Методы класса vector*

Доступ к элементу	
at	доступ к указанному элементу с проверкой границ
operator[]	доступ к указанному элементу
front	доступ к первому элементу
back	доступ к последнему элементу
data	прямой доступ к базовому хранилищу
Итераторы	
begin cbegin	возвращает итератор на начало контейнера
end cend	возвращает итератор на конец контейнера
rbegin crbegin	возвращает «обратный» итератор на начало перевернутого контейнера (указывает на последний элемент)
rend crend	возвращает «обратный» итератор на конец перевернутого контейнера (указывает на первый элемент)

Вместимость	
empty	проверяет, пуст ли контейнер
size	возвращает количество элементов
max_size	возвращает максимально возможное количество элементов
reserve	выделяет память для хранения указанного количества элементов
capacity	возвращает количество элементов, которые могут храниться в выделенном в данный момент хранилище (емкость)
Модификаторы	
clear	очищает содержимое контейнера
insert	вставляет элементы
insert_range	вставляет диапазон элементов
erase	стирает элементы
push_back	добавляет элемент в конец
pop_back	удаляет последний элемент
resize	изменяет количество сохраненных элементов
swap	меняет местами содержимое

*) Полный список можно найти в документации, например, https://en.cppreference.com/w/cpp/container/vector

LeetCode 525

Дан двоичный массив nums, найдите максимальную длину непрерывной подпоследовательности элементов массива с равным числом нулей и единиц.

Пример 1

Дано: nums = [0,1]
Ответ: 2
Пояснение: [0, 1] является самой длинной подпоследовательностью с одинаковым количеством нулей и единиц.

Пример 2

Дано : nums = [0,1,0]
Ответ : 2
Пояснение : [0, 1] (или [1, 0]) является самой длинной подпоследовательностью с одинаковым количеством нулей и единиц.

Пример 3

Дано : nums = [0,1,1,1,1,1,0,0,0]
Ответ : 6
Пояснение : [1,1,1,0,0,0] является самой длинной подпоследовательностью с одинаковым количеством нулей и единиц.

https://leetcode.com/problems/contiguous-array/description/

Решение. Перебор

Требуется рассмотреть все возможные непрерывные подпоследовательности и посчитать в них количество нулей и единиц.

- Для прохода по массиву используются два вложенных цикла.
- Во внешнем цикле итератор пробегает элементы от начала до конца массива, фиксируя начало подпоследовательности.
- Во внутреннем цикле итератор ведет конец подпоследовательности, формально включая в нее каждый следующий элемент за началом, пока не закончатся варианты.
- На каждой итерации внутреннего цикла подсчитывается количество нулей и единиц в подпоследовательности.
- Каждый раз, когда количество нулей и единиц становится равным, обновляется значение максимальной длины.

Решение. Перебор: пример

Пример: = [0,1,1,1,1,1,0,0,0]

index	0	1	2	3	4	5	6	7	8
nums	0	1	1	1	1	1	0	0	0

начало подпоследовательности

конец

Считаем: количество нулей / количество единиц

Начало подпоследовательности \ Конец подпоследовательности	0	1	2	3	4	5	6	7	8
0	–	**1/1**	1/2	1/3	1/4	1/5	2/5	3/5	4/5
1	–	–	0/2	0/3	0/4	0/5	1/5	2/5	3/5
2	–	–	–	0/2	0/3	0/4	1/4	2/4	3/4
3	–	–	–	–	0/2	0/3	1/3	2/3	**3/3**
4	–	–	–	–	–	0/2	1/2	**2/2**	3/2
5	–	–	–	–	–	–	1/1	2/1	3/1
6	–	–	–	–	–	–	–	2/0	3/0
7	–	–	–	–	–	–	–	–	2/0
8	–	–	–	–	–	–	–	–	–

Решение. Перебор: оценка сложности

Временная сложность: O(n^2).
Количество возможных значений для начала подпоследовательности равно ($n - 1$).
Для индекса начала последовательности $start$, количество возможных подпоследовательностей $n - start - 1$

Пространственная сложность : O(1).
Для решения задачи требуются две переменные - нули и единицы для подсчета и переменная, в которой будет обновляться значение максимальной длины.

Решение. Вспомогательный массив

Переберем все элементы массива nums с самого начала и определим переменную sum для хранения относительного количества единиц и нулей, встреченных на данномшаге прохода по массиву.

sum увеличивается на единицу всякий раз, когда встречается 1, и уменьшается на единицу всякий раз, когда встречается 0. Значение sum может варьироваться от -n (когда все элементы равны нулям) до +n (когда все элементы равны единицам).

Если на каком-либо индексе sum становится равной нулю, это означает равное количество нулей и единиц с начала и до текущего индекса.

Если на некотором индексе мы получаем sum, которая уже встречалась при каком-то предыдущем индексе, это означает, что количество 0 и 1 равно между этими индексами.

Отслеживаем индексы, которые соответствуют одному тому же значению sum, но отстоят друг от друга на максимальное расстояние.

Во вспомогательном массиве temp размером 2n+1 будем фиксировать индексы, при которых первый раз встречается соответствующее значение sum.

Всякий раз, когда мы снова встречаем ту же сумму, мы определяем длину подмассива, лежащего между индексами, соответствующими тем же значениям суммы.

Решение. Вспомогательный массив: пример

Требуется рассмотреть все возможные непрерывные подпоследовательности и посчитать в них количество нулей и единиц.

Пример: = [0,1,1,1,1,1,0,0,0]

Исходный массив (длина массива n = 9)

index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
nums	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**0**
sum	-1	0	1	2	3	4	3	2	1

Вспомогательный массив temp (длина массива 2n+1 = 19)

index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**
sum	-9	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8	9
temp	#	#	#	#	#	#	#	#	**0**	**-1**	**2**	**3**	**4**	**5**	#	#	#	#	#

sum =0 встретилась еще раз в индексе 1

sum =1 встретилась еще раз в индексе 8

sum =2 встретилась еще раз в индексе 7

sum =3 встретилась еще раз в индексе 6

Решение. Вспомогательный массив: оценка сложности

Временная сложность: O(*n*).
Для решения задачи требуется единственный проход по массиву из n элементов, за который собирается и анализируется информация о количестве 0 и 1.

Пространственная сложность : O(n).
Используется дополнительный массив размера (2n+1) и вспомогательная переменная sum.

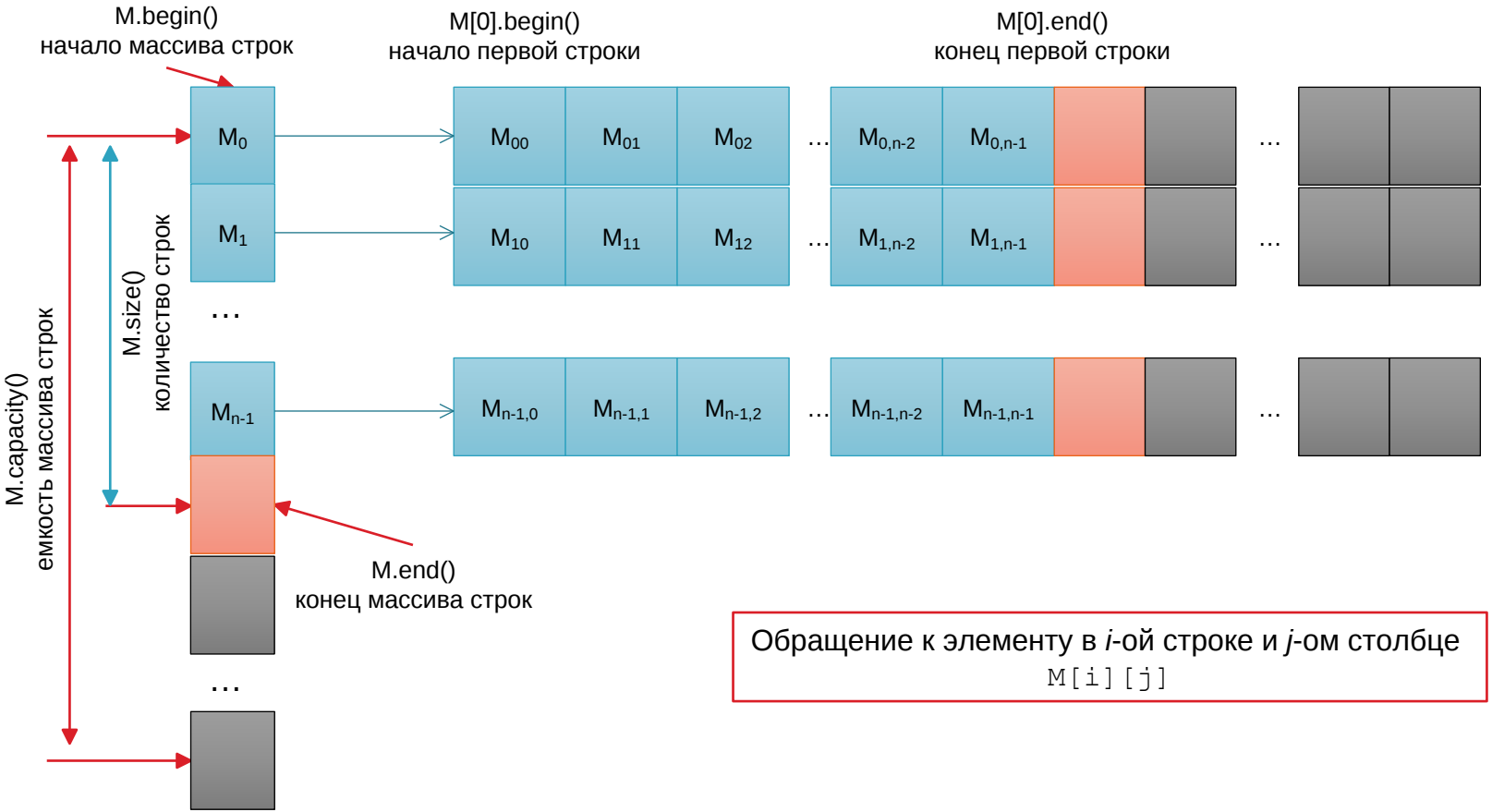
Матрицы

Матрица (std::vector <std::vector<>>)

Матрица в STL строится как вектор векторов, представляя собой либо набор строк, либо набор столбцов, в зависимости от того, как удобнее просматривать, анализировать и изменять матрицу.

Если матрица – это набор строк, то добавление и удаление строки являются методами «внешнего» вектора, а добавление/удаление строк требует работы с каждым «внутренним» вектором отдельно.

Описание матрицы (массив строк)



Обращение к элементу в *i*-ой строке и *j*-ом столбце

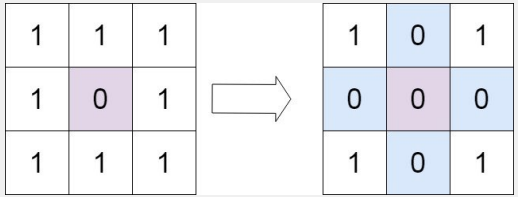
```
M[i][j]
```

LeetCode 73

Дана целочисленная матрица m x n. Для каждого ее нулевого элемента, обнулите также и значения элементов в строке и столбце, в которых он находится.

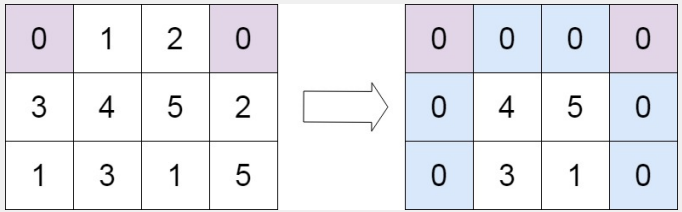
Дано: matrix = [[1,1,1],[1,0,1],[1,1,1]]]
Ответ: [[1,0,1],[0,0,0],[1,0,1]]

Пример 1



Дано : matrix = [[0,1,2,0],[3,4,5,2],[1,3,1,5]]
Ответ : [[0,0,0,0],[0,4,5,0],[0,3,1,0]]

Пример 2



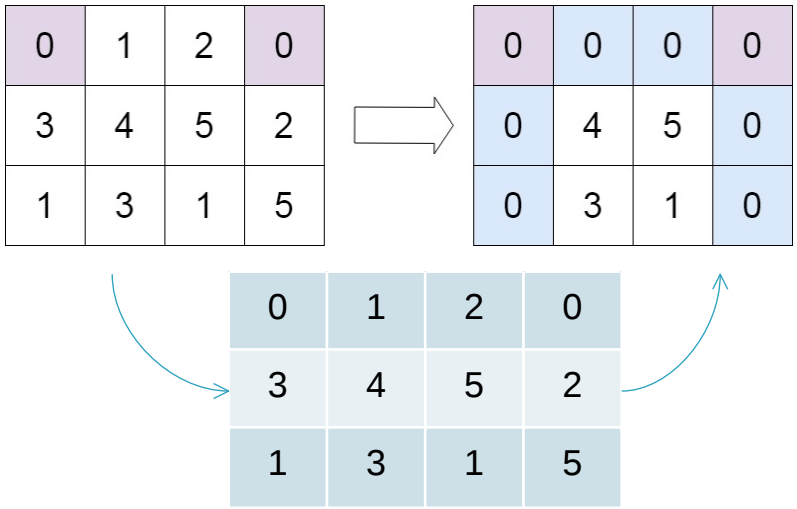
https://leetcode.com/problems/set-matrix-zeroes/description/

Решение. Вспомогательная матрица

Создаем вспомогательную матрицу temp, являющуюся копией исходной матрицы.

На основании вспомогательной матрицы принимаем решение об изменении исходной:

если элемент вспомогательной матрицы равен нулю, в исходной матрице обнуляем элементы в соответствующих строке и столбце.



Решение. Вспомогательная матрица: оценка сложности

Временная сложность: O($n \times m \times (n+m)$).
Для решения требуется проход по матрице, который требует $n \times m$ шагов. Для каждого нулевого элемента (в худшем случае все элементы могут быть нулевыми) требуется пройти по его строке и столбцу, что требует (n + m) шагов.

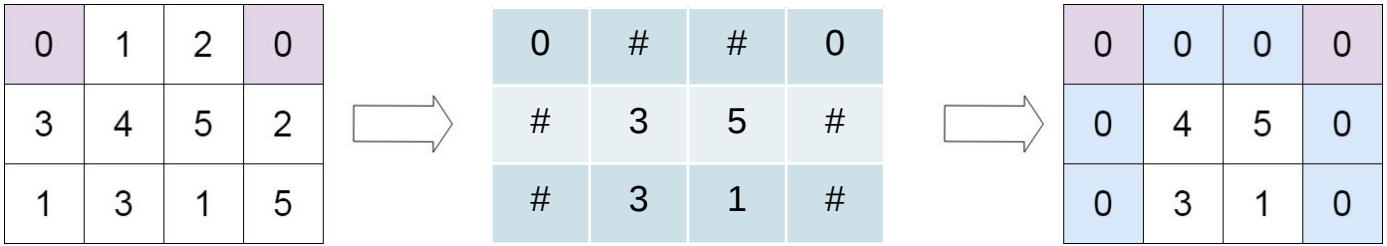
Пространственная сложность : O($n \times m$).
Потребовала вспомогательная матрица $n \times m$.

Решение. След изменений

Изменение матрицы проводится на том же месте, без создания вспомогательных массивов или матриц.

Первый проход по матрице отмечает те элементы, которые надо будет изменить, то есть стоящие в строках и столбцах, где есть нули.

Второй проход обращает в нуль отмеченные элементы.



Решение. След изменений: оценка сложности

Временная сложность: O($n \times m \times (n+m)$).
Для решения задачи организуем два прохода по матрице, каждый из которых требует $n \times m$ шагов. При первом проходе для каждого нулевого элемента (в худшем случае все элементы могут быть нулевыми) требуется пройти по его строке и столбцу, что требует (n + m) шагов.

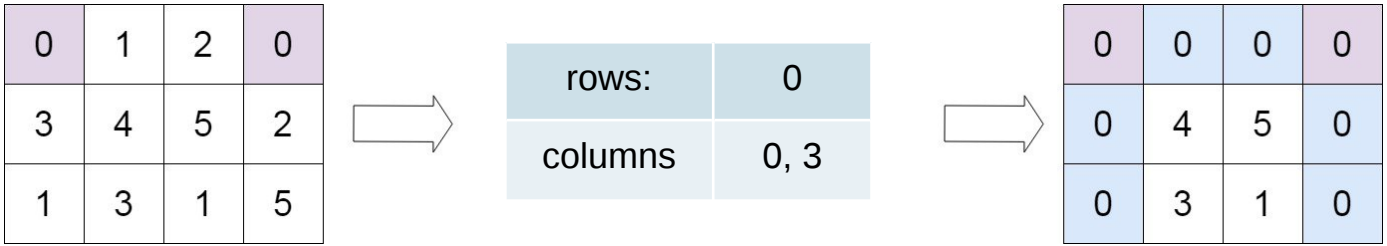
Пространственная сложность : O(*1*).
Дополнительная память не требуется (только отдельные вспомогательные переменные - индексы и размер матрицы).

Решение. Множество индексов

Создаем два вспомогательных множества для того, чтобы запомнить, в каких строках (rows) и столбцах (columns) требуется провести изменения.

За первый проход по матрице добавляем в множество индексов строк номера строк, в которых есть нулевые элементы, а в множество индексов столбцов – номера соответствующих столбцов.

Второй проход обращает в нуль элементы в строках и столбцах, записанных в множествах rows и columns.

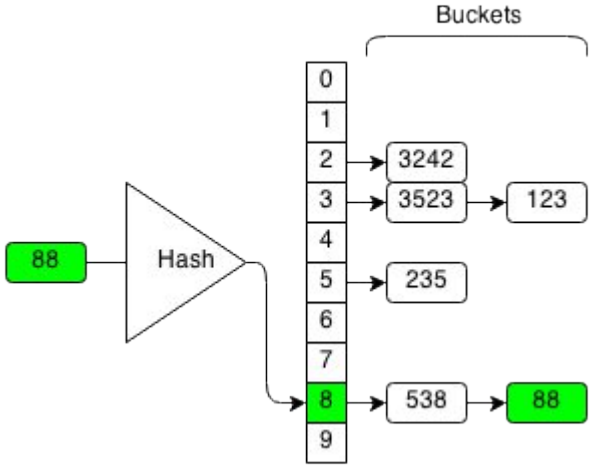


Множество

unordered_set - это неупорядоченный ассоциативный контейнер, котором хранятся **уникальные** элементы.

Для организации множества используется технология хеширования, когда элементу ставится в соответствие кодовая последовательность фиксированной длины.

Операции поиска, вставки и удаления выполняются в среднем за время O(1), но при этом элементы не сортируются в каком-либо определенном порядке.



Создание множества:
```
unordered_set<int> rows;
```

Вставка элемента:
```
rows.insert(i);
```

Ссылка

Решение. Множество индексов: оценка сложности

Временная сложность: O($n \times m$).
Для решения требуется проход по матрице, $n \times m$ шагов, для сбора информации.
Далее по результатам собранной информации заполняем нулями выбранные строки и столбцы, что также требует по $n \times m$ в худшем случае.

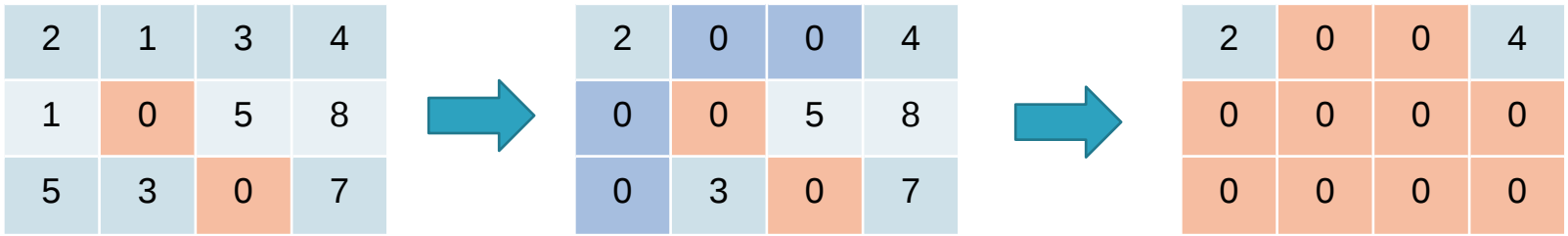
Пространственная сложность : O($n + m$).
Дополнительная память требуется для хранения индексов строк (n) и столбцов (m).

Решение. Метки

Изменение матрицы проводится на том же месте, без создания вспомогательных массивов или матриц.

Первый проход по матрице в первой строке отмечает нулями те столбцы, где есть нули, а в первом столбце – строки, где есть нули.

Второй проход обращает в нуль элементы в отмеченных строках и столбцах: сначала во всех строках и столбцах кроме первых, потом, если требуется, «закрашивает» первые.



Решение. Метки: оценка сложности

Временная сложность: O($n \times m$).
Для решения требуется проход по матрице, $n \times m$ шагов, для сбора информации.
Далее по результатам собранной информации заполняем нулями выбранные строки и столбцы, что также требует по $n \times m$ шагов.

Пространственная сложность : O(*1*).
Дополнительная память не требуется.

LeetCode 79

Дана символьная матрица $m \times n$ и слово длины w. Определите, находится ли слово в матрице.

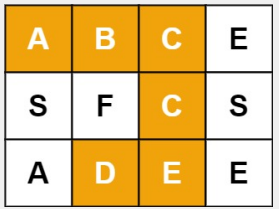
Требуемое слово может быть составлено из букв, расположенных в соседних ячейках.

Соседние ячейки это те, у которых есть общие грани.

Одна и та же ячейка не может использоваться более одного раза.

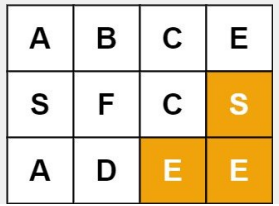
Пример 1

Дано: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCCED"
Ответ: true



Пример 2

Дано : board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "SEE“
Ответ : true



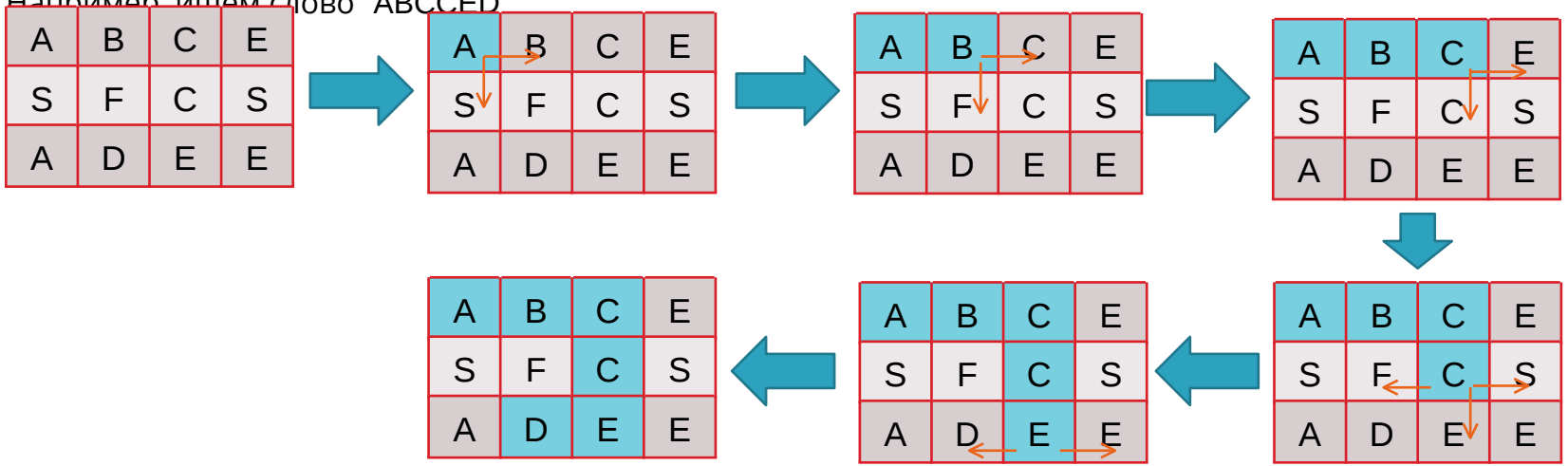
https://leetcode.com/problems/word-search/description/

Решение. Поиск в глубину

Создаем множество посещенных ячеек, чтобы не попадать на один и тот же элемент дважды.

Реализуем рекурсивный алгоритм поиска в глубину. Если нашли какую-то букву слова в матрице, при условии, что все предыдущие тоже найдены, следующую ищем в соседних ячейках, исключая уже посещенные.

Например, ищем слово "ABCCED"



Решение. Обход в глубину: оценка сложности

Временная сложность: O($n \times m \times 3^w$).
Для решения требуется проход по матрице, $n \times m$ шагов.
Для каждой ячейки может потребоваться проверка наличия слова, что предполагает просмотр максимум трех соседних ячеек (всегда исключаем ту, откуда пришли).
Поиск слова, начиная с текущей ячейки оцениваем в 3^W операций.

Пространственная сложность : O(3^w).
Дополнительная память требуется для отслеживания посещенных ячеек, что не превышает *3w.*

Стек

Стек (stack<>)

Стек - это линейная структура данных, которая соответствует определенному порядку выполнения операций.

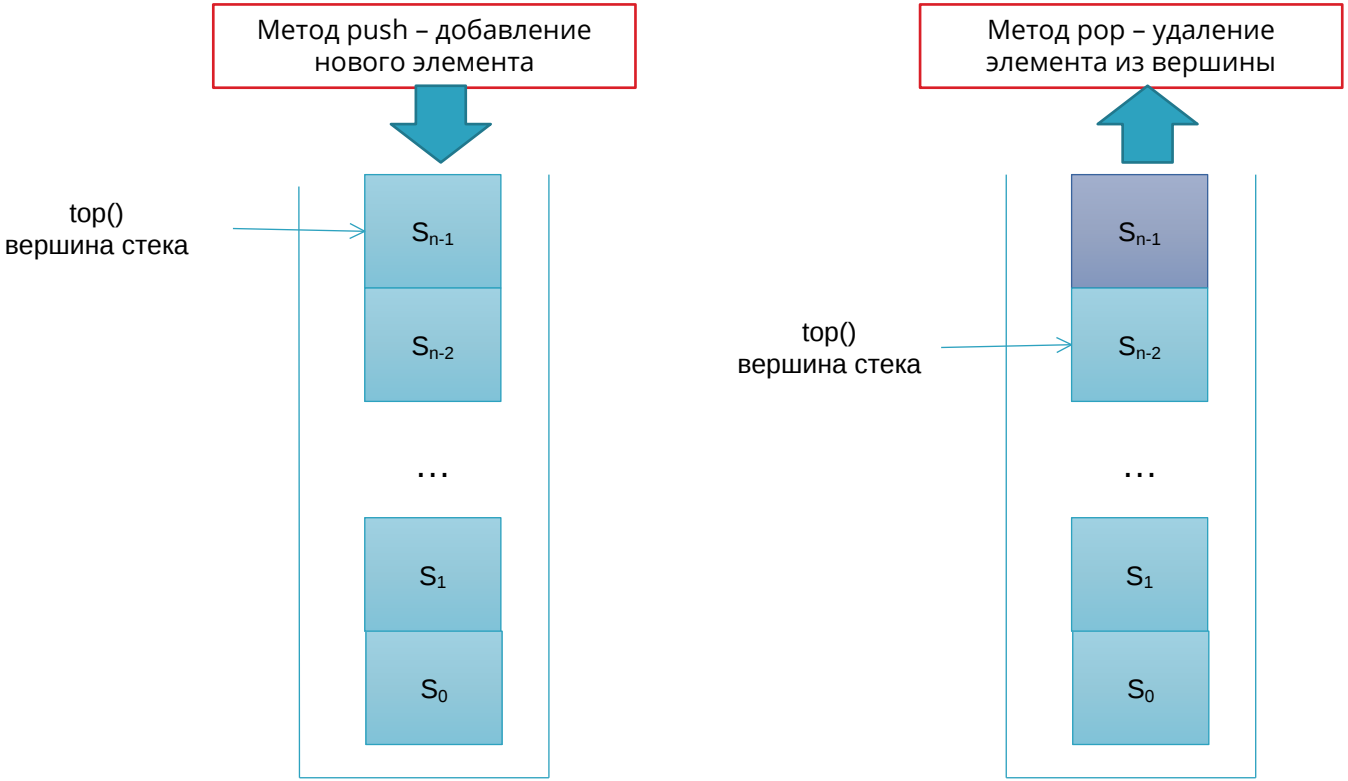
Порядок может быть LIFO (Last Input First Output = Кто последний пришел, тот первым ушел)

или

FILO (First Input Last Output = Кто первый пришел, тот последним ушел).

Стек в STL является адаптером последовательных контейнеров, ограничивая их функциональность только необходимыми для собственной реализации методами.

Описание стека



LeetCode 155

Разработайте стек, который поддерживает функции:

- Добавления элемента в стек (push),
- Удаления элемента из вершины стека (pop),
- Получения значения элемента в вершине стека (top)
- Получение значения минимального элемента (getMin).

Для всех операций алгоритмическая сложность должна быть O(1).

```
class MinStack {
public:
    MinStack() {

    }

    void push(int val) {

    }

    void pop() {

    }

    int top() {

    }

    int getMin() {

    }
};
```

https://leetcode.com/problems/min-stack/description/

LeetCode 155

Какую структуру данных лучше использовать, чтобы гарантировать алгоритмическую сложность O(1)?

```
class MinStack {
public:
    MinStack() {

    }

    void push(int val) {

    }

    void pop() {

    }

    int top() {

    }

    int getMin() {

    }
};
```

https://leetcode.com/problems/min-stack/description/

Очередь

Очередь (queue<>)

Очередь - это линейная структура данных, которая соответствует порядку выполнения операций FIFO (First Input First Output = Кто последний пришел, тот первым ушел)

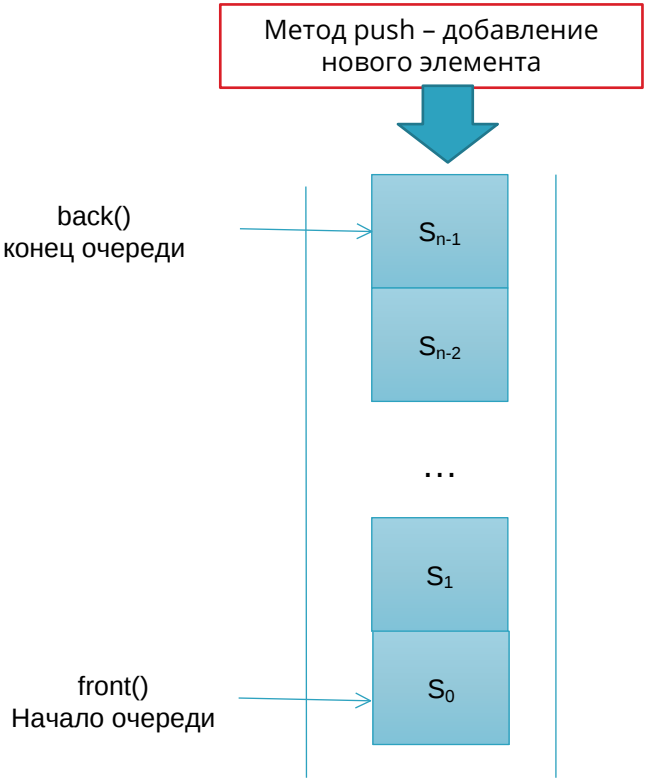
В STL очередь, как и стек, является адаптером последовательных контейнеров, ограничивая их функциональность только необходимыми для собственной реализации методами.

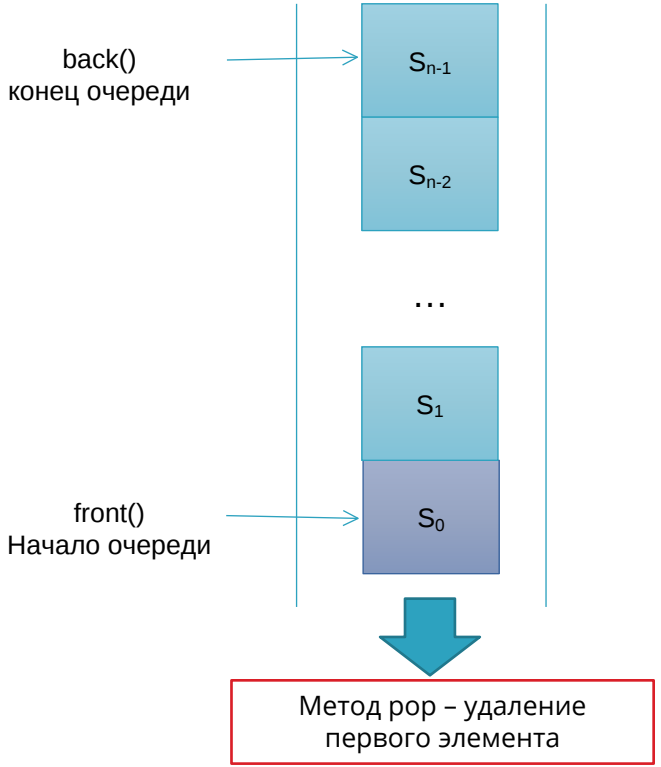
Он используется в качестве буфера в компьютерных системах, где есть несоответствие скорости между двумя взаимодействующими устройствами.

Очередь также используется в алгоритмах операционной системы, таких как планирование работы процессора и управление памятью.

Широко применяется во многих стандартных алгоритмах, например, поиск по ширине графа и обход дерева по уровням.

Описание очереди





LeetCode 239

Дан массив, в котором записаны n целых чисел. Также дано также скользящее окно размером k. Скользящим окном ширины k называют непрерывный блок ячеек массива, состоящий из k элементов, соответственно, можно видеть только те элементы, что находятся в окне, остальные считаются скрытыми.

Правила:

- Окно перемещается от левого края массива к правому правый.
- Каждый раз скользящее окно перемещается вправо на одну позицию.

Для каждого положения скользящего окна укажите максимальный элемент.

https://leetcode.com/problems/sliding-window-maximum/description/

LeetCode 239. Пример

Дано: nums = [1,3,-1,-3,5,3,6,7],
k = 3
Ответ: [3,3,5,5,6,7]

								max
1	3	-1	-3	5	3	6	7	3
1	3	-1	-3	5	3	6	7	3
1	3	-1	-3	5	3	6	7	5
1	3	-1	-3	5	3	6	7	5
1	3	-1	-3	5	3	6	7	6
1	3	-1	-3	5	3	6	7	7

https://leetcode.com/problems/sliding-window-maximum/description/

Решение. Перебор

Проверяем каждое положение скользящего окна и вычисляем максимальное значение.

Решение. Перебор: оценка сложности

Временная сложность: O($n \times k$).
Для решения требуется последовательный перебор всех положений окна, которых $n - k + 1$.
Для каждого окна ищем максимум за k-1 шаг.
Итого (n-k+1)(k-1)~nk

Пространственная сложность : O(n) или O(1), если не включать итоговый массив.
Потребовался массив из n-k+1~n элементов для хранения результата

Очередь с приоритетами

Очередь с приоритетами

Priority_queue — это контейнер-адаптер в C++, который позволяет управлять коллекцией элементов с приоритетами.

В отличие от обычной очереди, где элементы обрабатываются по принципу «первый пришёл — первый вышел» (FIFO), в priority_queue элементы упорядочены по приоритету, при этом элемент с наивысшим приоритетом всегда находится в начале очереди.

Priority_queue подходит для сценариев, где необходимо обрабатывать элементы на основе их приоритета, например для планирования задач, алгоритма Дейкстры и других.

Приоритет

По умолчанию приоритетом является величина элемента: чем больше его значение, тем выше приоритет.

Если однозначно определить приоритет нельзя, требуется снабдить тип элементов оператором сравнения

Элементы целого типа: `queue<int> q({1,2,3});` Порядок в очереди: 3, 2, 1

```
struct Item{
    int key;
    bool operator<(const Item &item) const{return key < item.key;}
};
```

Элементы пользовательского типа: `queue<Item> q({Item{1},Item{2},Item{3}});`

Порядок в очереди: `Item{3}, Item{2}, Item{1}`

Приоритет

Схему приоритета это можно изменить, указав правила сравнения элементов.

```
int data[]{3,2,1,5,4};
   std::priority_queue<int, std::vector<int>, std::greater<int>>
      min_pq(begin(data), end(data));
```

1,2,3,4,5

```
struct{
      bool operator()(const int l, const int r) const {
         return l % 10 > r % 10; }
   } less_sum;
int data[]{82,11,131,49, 115};
std::priority_queue sum_pq(begin(data), end(data), less_sum);
```

11,131,82,115,49

Основные методы priority_queue

push() — добавляет элемент в очередь с сохранением порядка.

pop() — удаляет верхний элемент (с наивысшим приоритетом).

top() — извлекает верхний элемент без его удаления.

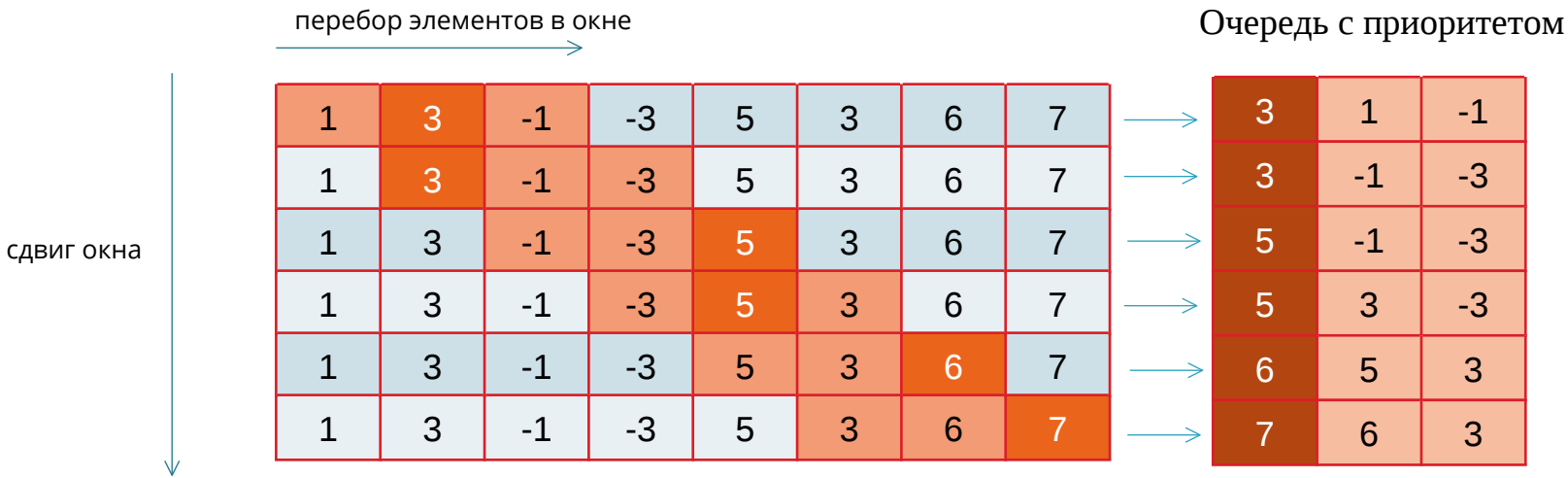
empty() — проверяет, пуста ли очередь с приоритетами.

size() — возвращает количество элементов в очереди.

Решение. Очередь с приоритетами

Идея заключается в том, на месте окна формируется очередь с приоритетом.

На каждом шаге сдвига окна, удаляются элементы, которые находятся за пределами окна, а новый включается в очередь.



Решение. Очередь с приоритетами: оценка сложности

Временная сложность: O(*n log k*) .
Для решения требуется последовательный перебор всех положений окна, которых n - k + 1.
Вставка элемента и удаление элемента занимает O(log k) времени.

Пространственная сложность : O(k) .
Дополнительно требуется очередь с приоритетами из k элементов.

Решение. Очередь с приоритетами: код (множество)

multiset упорядоченное множество, позволяющее дублирование элементов

extract исключает одно вхождение элемента, если есть его дубликаты, они остаются

```
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        if (nums.size() == 0 || k == 0) {
            return vector<int>();
        }
        int n = nums.size();
        auto lastWindow = nums.begin() + n - k + 1;
        vector<int> res(n - k + 1);
        multiset<int> pq(nums.begin(), nums.begin() + k);
        auto first = nums.begin(), cur = res.begin();
        do{
            *cur = *pq.crbegin();
            pq.extract(*first);
            if (first + k != nums.end())pq.insert(*(first + k));
            first++, cur++;
        } while(first != lastWindow);
        return res;
    }
};
```

Дек – двунаправленная очередь

Дек

Дек (deque = double-ended queue) в C++ — это структура, объединяющая стек и очередь , или две очереди (двусторонняя очередь)

В деке реализовано эффективное добавление и удаление элементов в начале и в конце (временная сложность O(1)) , так как старые элементы остаются в тех же ячейках памяти, их не приходится копировать.

Вставка в середину дека и удаление из неё требуют сдвига элементов.

В отличие от очереди и стека реализована операция обращения к элементу по индексу.

Решение (продолжение). Дек

В скользящее окно будем добавлять не сами элементы, а их индексы.

Пусть нам требуется включить i-ый элемент, тогда

1) Из окна удаляем элементы, которые вышли слева за границу окна.
2) Удаляем справа индексы элементов, которые меньше, чем включаемый, так все что меньше не может быть максимумом
3) Добавляем индекс i

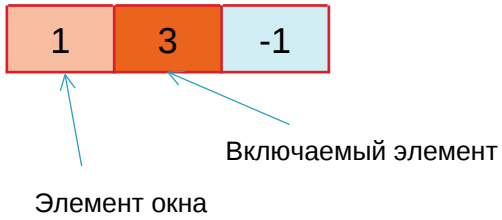
Гарантировано самый большой элемент будет в начале окна.

Решение. Дек. Пример

1	3	-1	-3	5	3	6	7
1	3	-1	-3	5	3	6	7
1	3	-1	-3	5	3	6	7
1	3	-1	-3	5	3	6	7
1	3	-1	-3	5	3	6	7
1	3	-1	-3	5	3	6	7
1	3	-1	-3	5	3	6	7
1	3	-1	-3	5	3	6	7

Дек

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7



Решение. Дек: оценка сложности

Временная сложность: O(n) .
Требуется организовать один проход по массиву, принимая решение на каждом шаге о модификации дека.

Пространственная сложность : O(k) .
Дек на каждом шаге хранит не более k элементов.

LeetCode 1670

Разработайте очередь, которая поддерживает операции добавления и удаления элементов в начало, середину и конец очереди.

Реализуйте класс FrontMiddleBackQueue с методами:

- Конструктор по умолчанию.
- void pushFront(int val) Добавляет значение val в начало очереди.
- void pushMiddle(int val) Добавляет значение val в середину очереди.
- void pushBack(int val) добавляет значение val в конец очереди.
- int popFront() Удаляет начальный элемент очереди и возвращает его. Если очередь пуста, возвращает значение -1.
- int popMiddle() Удаляет средний элемент очереди и возвращает его. Если очередь пуста, возвращает значение -1.
- int popBack() Удаляет последний элемент очереди и возвращает его. Если очередь пуста, возвращает значение -1.

Обратите внимание, что при наличии двух вариантов выбора среднего элемента, операция выполняется с тем, что ближе к началу.

https://leetcode.com/problems/design-front-middle-back-queue/

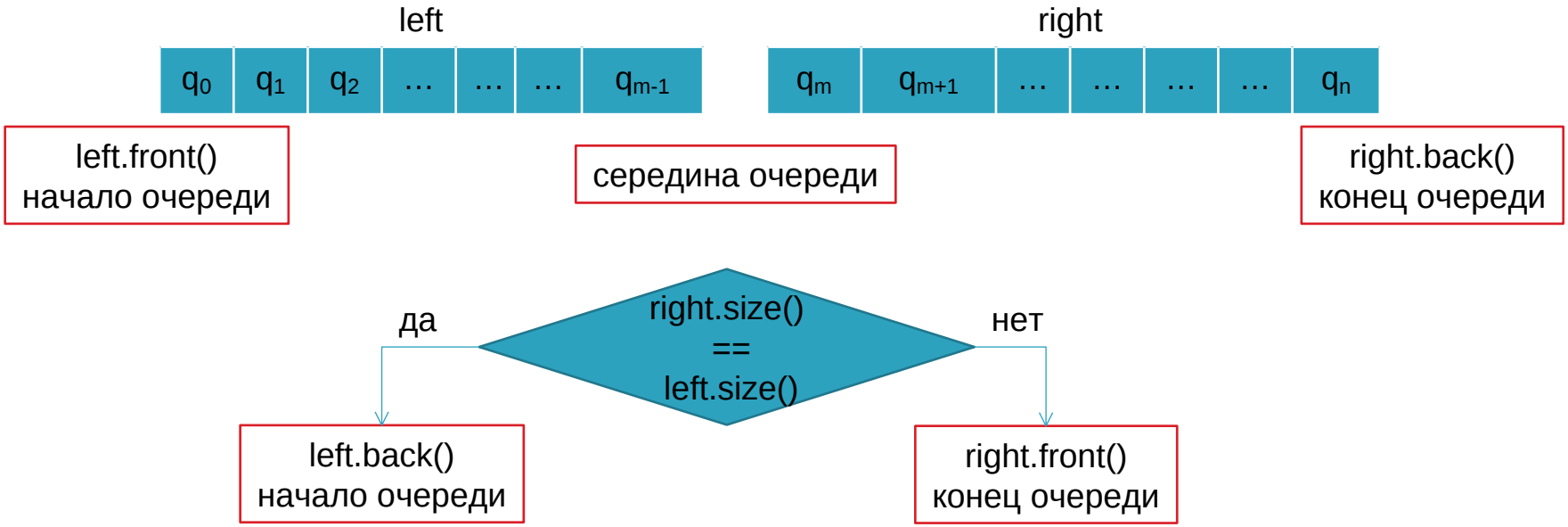
LeetCode 1670

Какую структуры данных лучше использовать, чтобы было как можно меньше перестановок элементов или вообще исключить сдвиги в массиве?

```
class FrontMiddleBackQueue {
public:
    FrontMiddleBackQueue() {}

    void pushFront(int val) {}

    void pushMiddle(int val) {}

    void pushBack(int val) {}

    int popFront() {}

    int popMiddle() {}

    int popBack() {}
};
```

https://leetcode.com/problems/design-front-middle-back-queue/

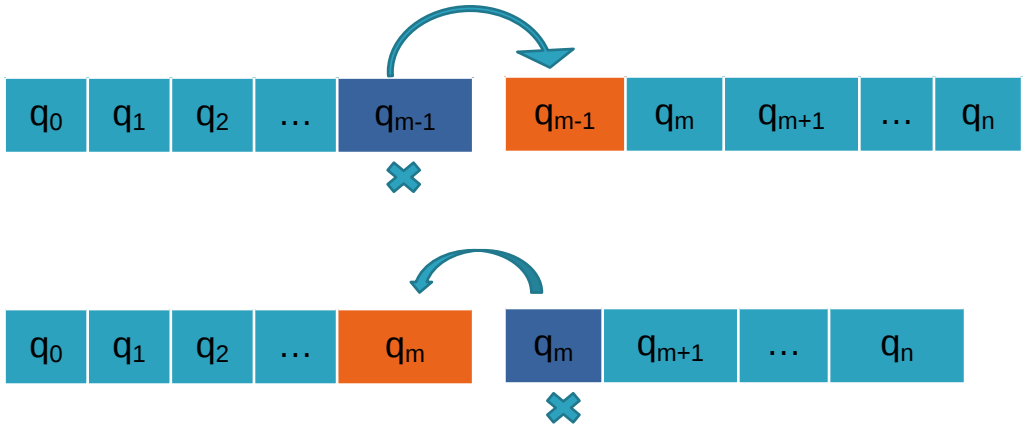
Организация очереди через два дека



Решение. Два дека

Вспомогательные функции

```
class FrontMiddleBackQueue {
    deque<int> left;
    deque<int> right;
public:
    FrontMiddleBackQueue() {}

    void shiftRight(){
        if (right.size() < left.size()){
            int temp = left.back();
            left.pop_back();
            right.push_front(temp);
        }
    }
    void shiftLeft(){
        if (right.size() - left.size() > 1){
            int temp = right.front();
            right.pop_front();
            left.push_back(temp);
        }
    }
```

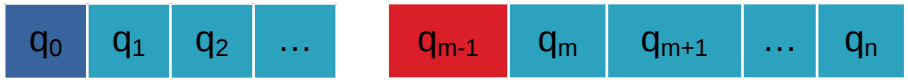


Решение. Вставка элментов

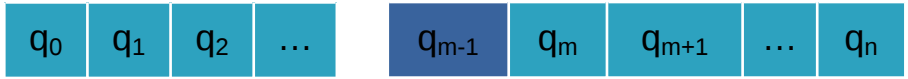
```
void pushFront(int val) {
    left.push_front(val);
    shiftRight();
}

void pushMiddle(int val) {
    if (left.size()==right.size())
        right.push_front(val);
    else left.push_back(val);
}

void pushBack(int val) {
    right.push_back(val);
    shiftLeft();
}
```

Решение. Удаление элементов

```
int popFront() {
  if (left.size() == 0 && right.size() == 0) return -1;
  if (left.size() == 0) {
     int temp = right.front();
     right.pop_front();
     return temp;
  }
  int temp = left.front();
  left.pop_front();
  shiftLeft();
  return temp;
}
```




```
int popMiddle() {
  int temp;
  if (right.size() > left.size()){
     temp = right.front();
     right.pop_front();
  }
  else{
     temp = left.back();
     left.pop_back();
  }
  return temp;
}
```




```
int popBack() {
  if (left.size() == 0 && right.size() == 0)
     return -1;
  int temp = right.back();
  right.pop_back();
  shiftRight();
  return temp;
  }
```