
Изучаем новые структуры данных

# Список

Что у нас сегодня?

1. Асимптотика сортировки
2. Сортировка подсчетом
3. Связный список
4. Двусвязный список
5. Стек
6. Очередь
7. Циклический массив

Какова оптимальная асимптотика сортировки в общем случае?

Вопрос нетривиальный :^)

*Воспользуемся теорией информации, чтобы получить ответ.*

# Найдем оптимальную асимптотику

Сколько информации дает нам каждое сравнение?

**1 бит**

Сколько существует вариантов упорядочения массива длины N?

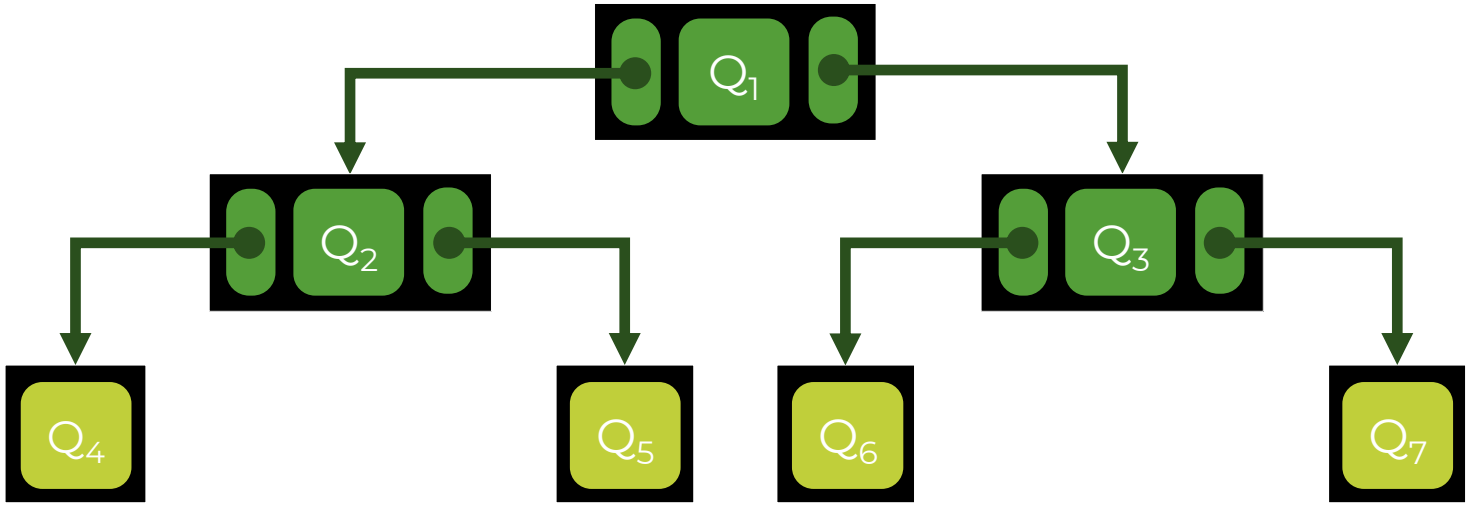
**N!**

Сколько бит нам необходимо, чтобы однозначно указать на вариант сортировки?

**$\log_2 N!$**

# Или проще:

- Каждое сравнение дает 1 бит информации.
- Строим бинарное дерево с **N!** листьями.
- Его минимальная высота – $\mathbf{\log_2(N!)}$.



Найдем асимптотику количества операций по формуле Стирлинга...

$$\log_2 N! = \log_2 \left( \sqrt{2\pi N} \left( \frac{N}{e} \right)^N \right) =$$

$$= N \log N + O(N) =$$

$$= \Omega(N \log N)$$

Оценка снизу!

# Сортировка подсчетом

Дан массив целых числовых элементов, находящихся в определенном диапазоне **[A; B]**:

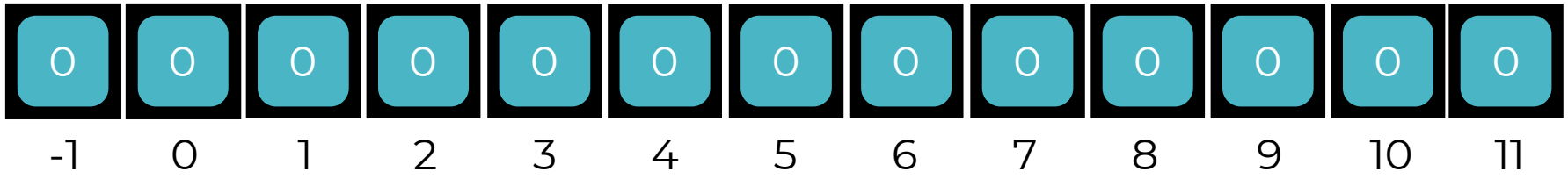

В этом примере мы знаем, что все ключи лежат в **[-1; 11]**

**Min** и **Max** можно найти за **O(N)**.

# Сортировка подсчетом

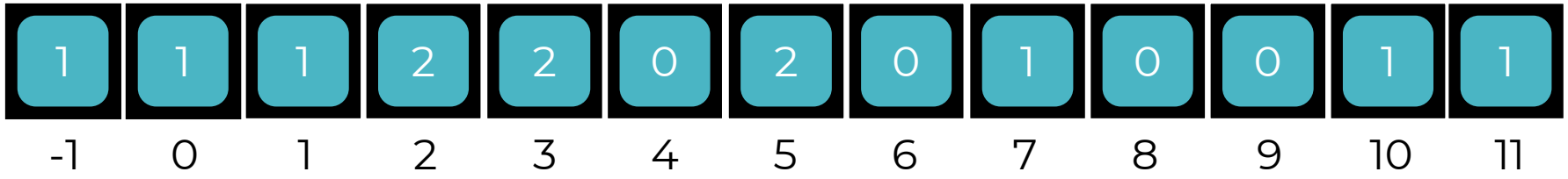
Дан массив целых числовых элементов, находящихся в определенном диапазоне **[A; B]**:


Создадим массив размера **B - A + 1** = 11 - (-1) + 1 = **13** и заполним его нулями



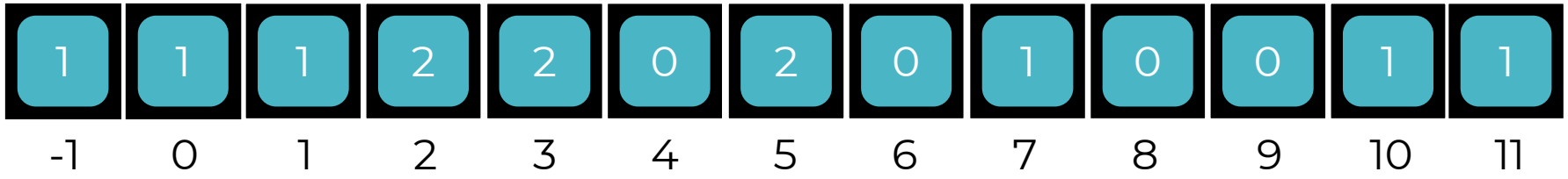
# Сортировка подсчетом


Затем проходимся по массиву один раз и для каждого индекса делаем: **count[arr[i] - A] += 1**



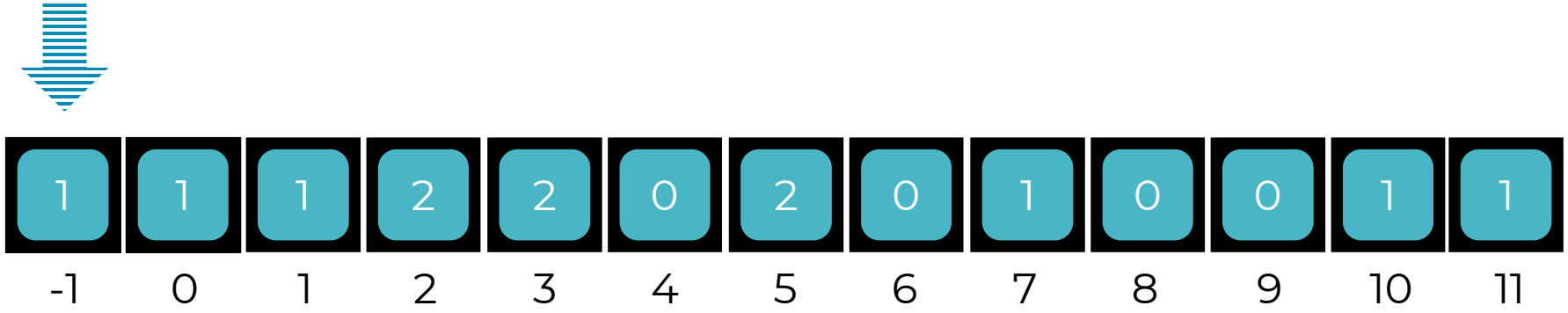
# Сортировка подсчетом

Теперь создаем пустой массив размера исходного



# Сортировка подсчетом

И заполняем пустой массив по массиву с подсчетами.

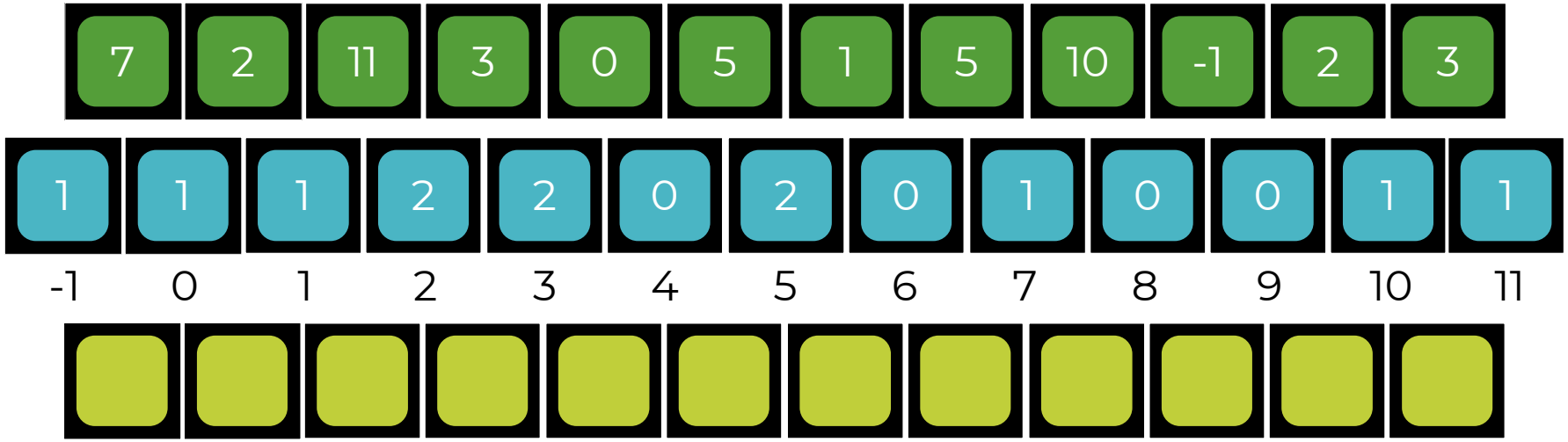



# Что не так?

Каждый элемент – это пара:
**(Ключ, значение)**.
Непонятно, что делать со значениями.

# Сортировка подсчетом

Придется делать иначе: сделаем так, чтобы массив счетчиков указывал, на какую позицию нам нужно ставить элемент...

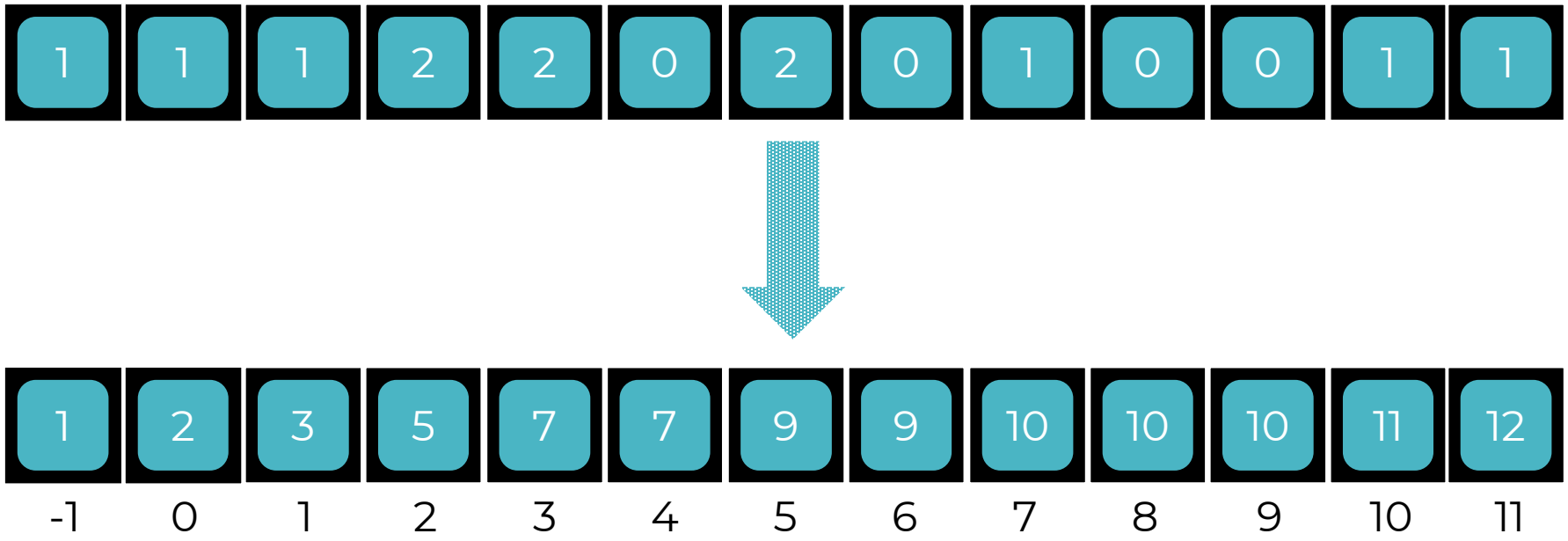



А сам новый массив будем заполнять, идя по старому в обратную сторону.

# Сортировка подсчетом

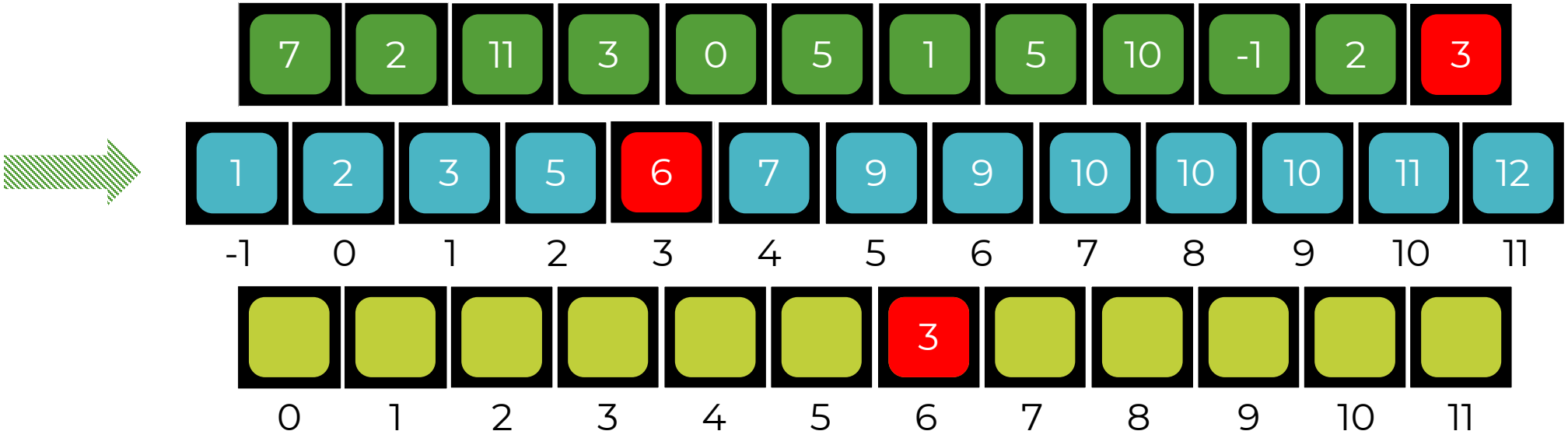
Для этого делаем массив счетчиков кумулятивным:

Проходимся **count[i] += count[i - 1]**.



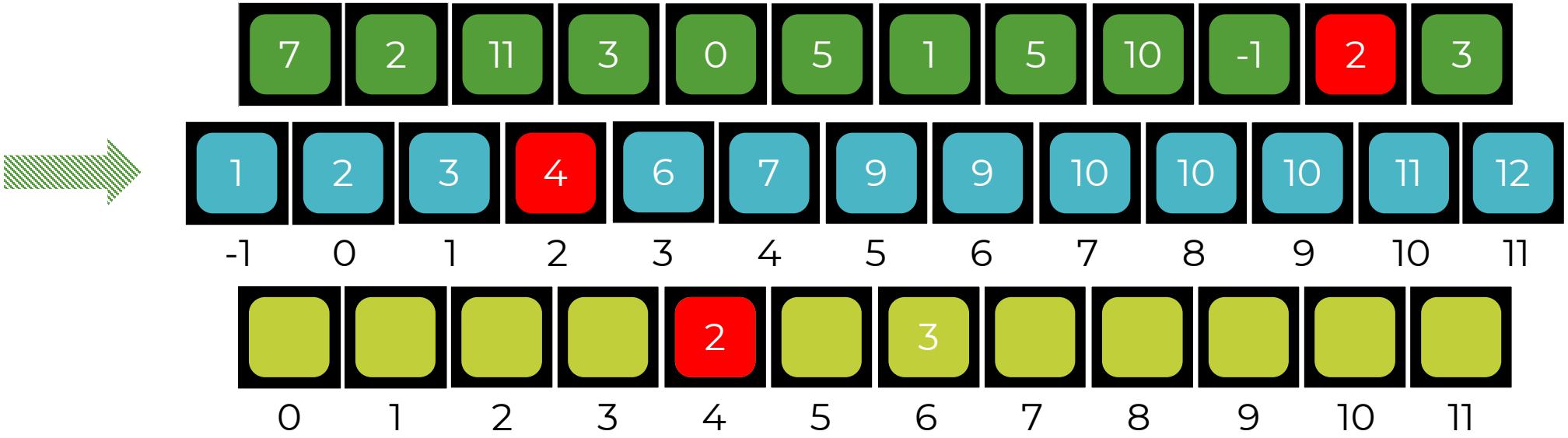
# Сортировка подсчетом

Теперь массив счетчиков содержит позицию последнего элемента с таким номером **+ 1**.



# Сортировка подсчетом

Теперь массив счетчиков содержит позицию последнего элемента с таким номером **+ 1**.



Какова асимптотика сортировки подсчетом?

В общем случае?

**O(N + B - A)**

Затраты памяти?

**O(N + B - A)**

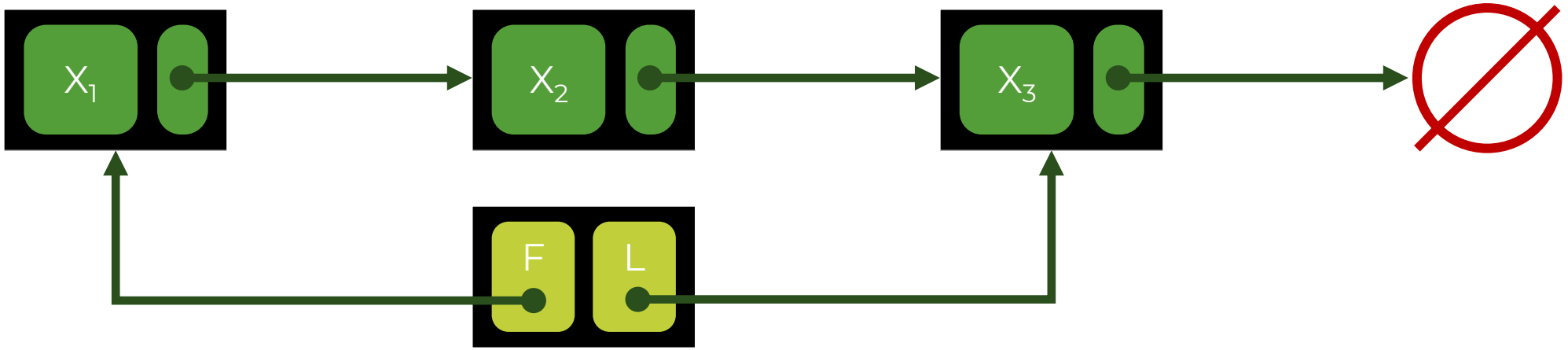
# Какова мораль?

Каждая задача может иметь лучшее решение, если у вас есть дополнительная информация!

# Перейдем к структурам данных

# Связный список

**Связный список** — это структура данных, состоящая из узлов, каждый из которых содержит данные и ссылку («связку») на следующий узел списка.



# Операции на связном списке

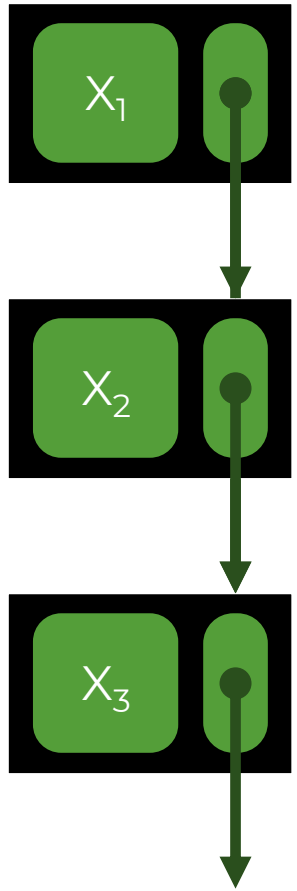
| Операция | Массив | Дин. массив | Связный список |
|---|---|---|---|
| Индексация | O(1) | O(1) | O(N) |
| Вставка (append) | O(N) | O(1)* | O(1) |
| Вставка (insert) | O(N) | O(N) | **O(1) → \| ← O(N)** |
| Удаление ($pop_0$) | O(N) | O(N) | O(1) |
| Удаление ($pop_x$) | O(N) | O(N) | **O(N)** |
| Вставка (индекс) | O(N) | O(N) | O(N) |
| Поиск | O(N) | O(N) | O(N) |

# Вставка в связный список

Асимптотика зависит от того, известен ли вам элемент, после которого вы вставляете новый, или нет.

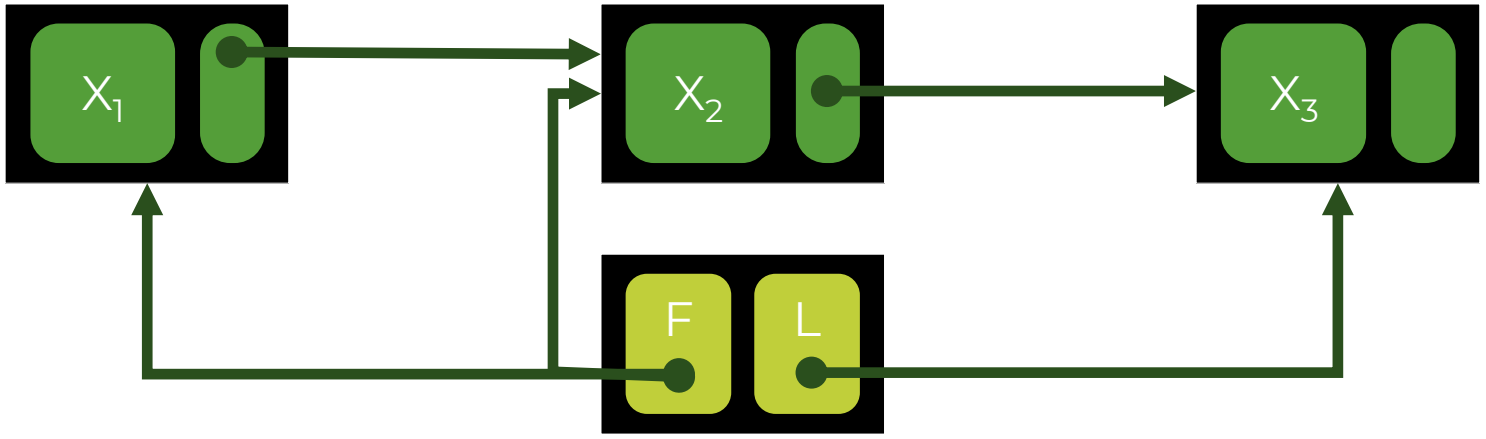
Если элемент **известен**, то достаточно создать новый узел и переопределить две связи. Асимптотика – **O(1)**.

Если нет, то предыдущий узел необходимо искать с первого элемента («головы») связного списка. Асимптотика – **O(N)** в общем случае.



# Добавление в начало

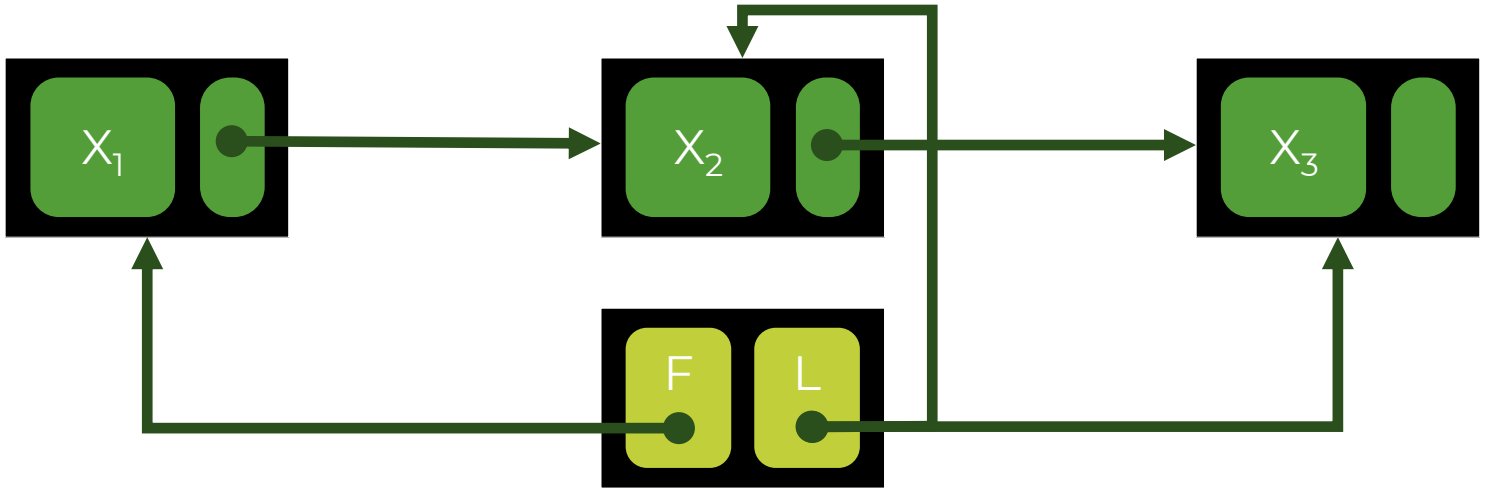
1. Создаем новый узел.
2. Создаем ссылку на старый первый элемент из нового первого элемента.
3. Меняем ссылку на первый элемент.




O(1)

# Удаление из конца

1. Меняем ссылку на последний элемент.
2. Меняем указатель нового последнего элемента на **Null**.




Почему O(N)?

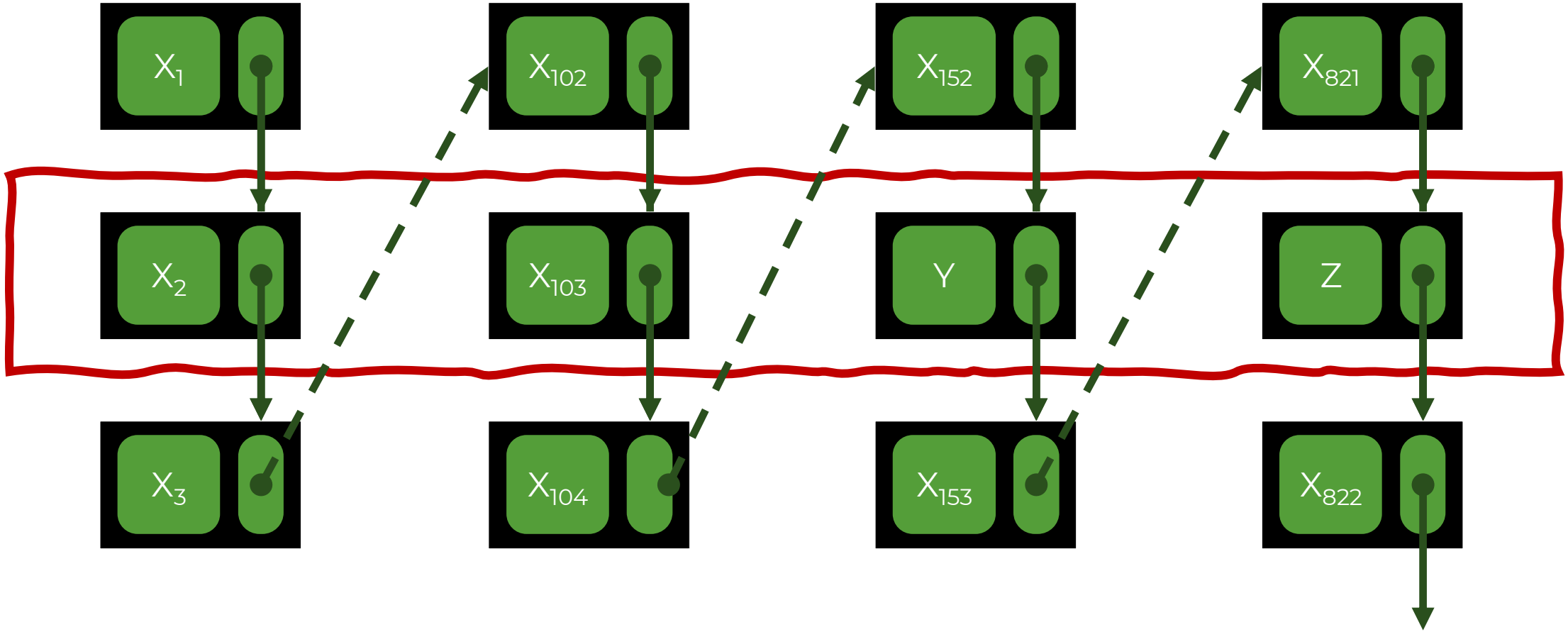
В чем преимущества связного списка?

**Не требует фиксированной памяти**

**Параллелизуемый**

**Быстро добавляет и удаляет элементы с концов**

# Параллелизация



Уже спешим использовать вместо массивов?

**Хаотично расположен в памяти**

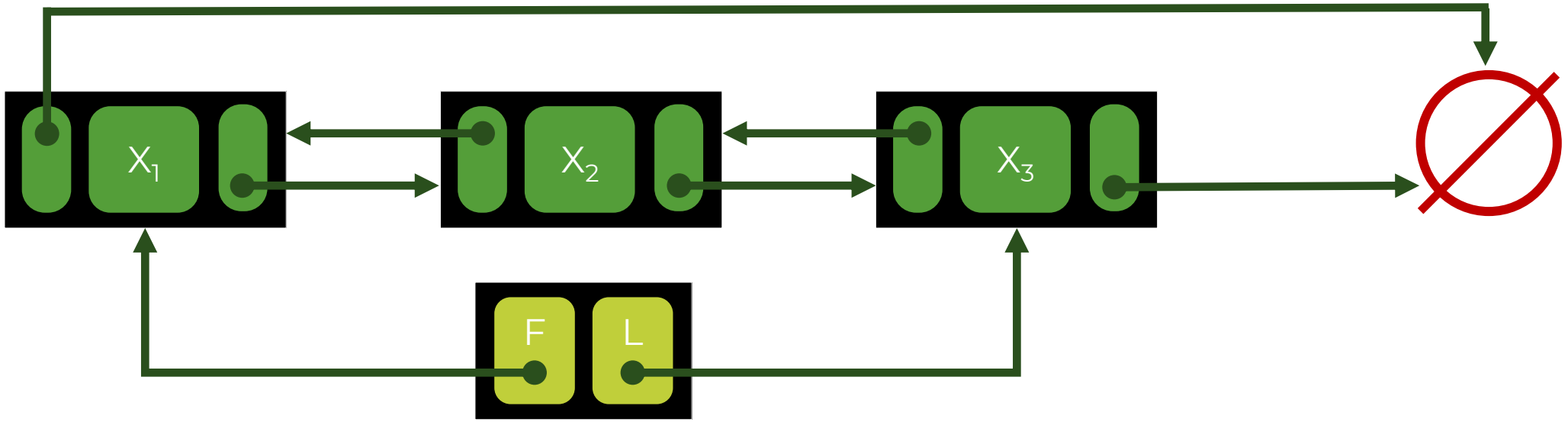
**Трудно искать элементы**

**Индексировать тоже трудно**

# Пишем псевдокод!

# Двусвязный список

**Двусвязный список** — это структура данных, состоящая из узлов, каждый из которых содержит данные и ссылку («связку») на следующий **и предыдущий** узлы списка.



# Операции на двусвязном списке

| Операция | Массив | Дин. массив | Двусвязный список |
|---|---|---|---|
| Индексация | O(1) | O(1) | O(N) |
| Вставка (append) | O(N) | O(1)* | O(1) |
| Вставка (insert) | O(N) | O(N) | **O(1)** |
| Удаление ($pop_0$) | O(N) | O(N) | O(1) |
| Удаление ($pop_x$) | O(N) | O(N) | **O(1)** |
| Вставка (индекс) | O(N) | O(N) | O(N) |
| Поиск | O(N) | O(N) | O(N) |

Зачем нам нужен двусвязный список?

Если мы часто удаляем элементы не с начала

Если мы зачем-то часто смотрим на элементы, близкие к концу

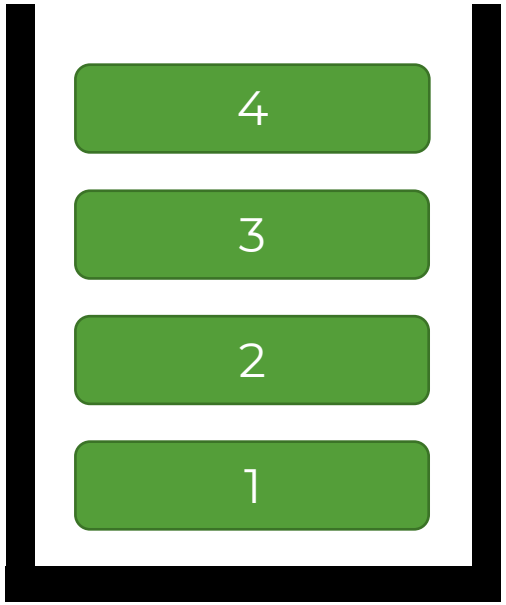
# Пишем псевдокод!


стек overflow

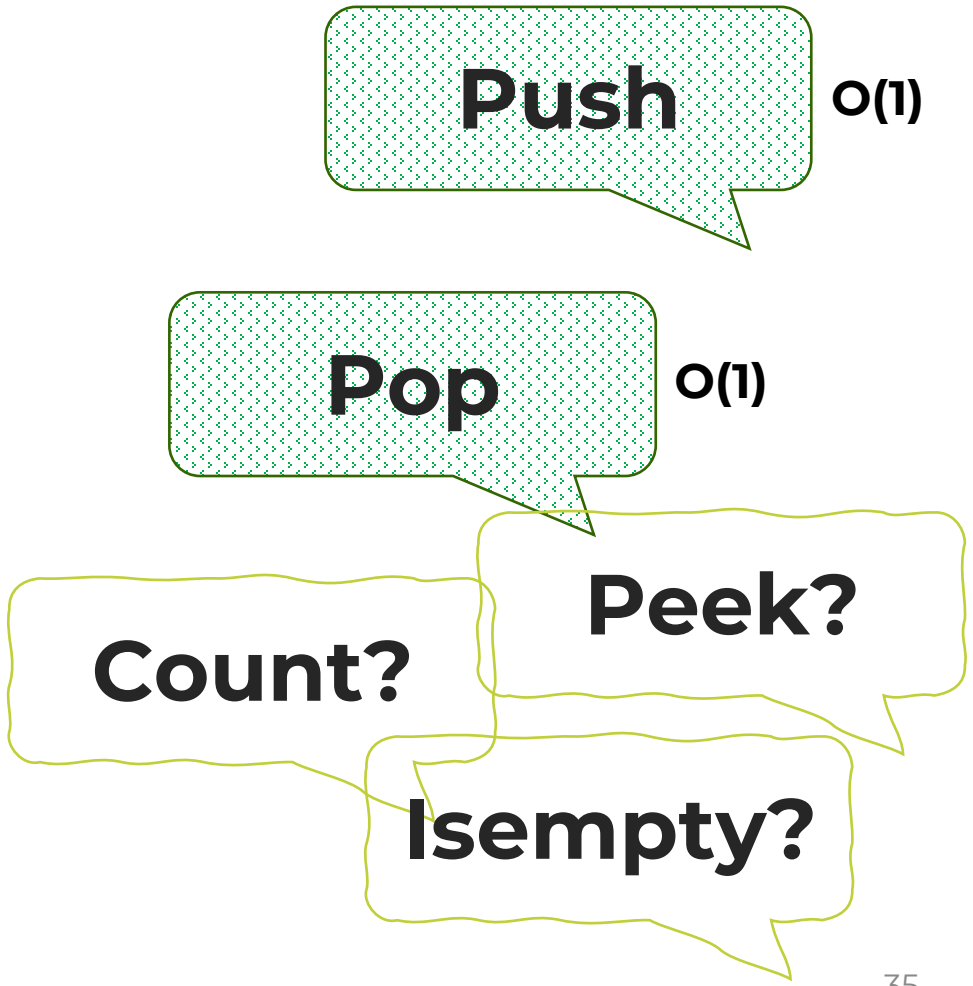
# Стек (stack)

**Стек** – это **абстрактная** структура данных, которая представляет собой набор элементов, упорядоченных по принципу **LIFO**.

**L**ast **I**n – **F**irst **O**ut.



Какие операции нужны для стека?

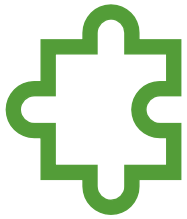


На какой структуре можно организовать стек?

**Дин. массив**

**Связный список**

**Двусвязный список**



# Пишем псевдокод!

Зачем вообще нужен этот ваш стек?

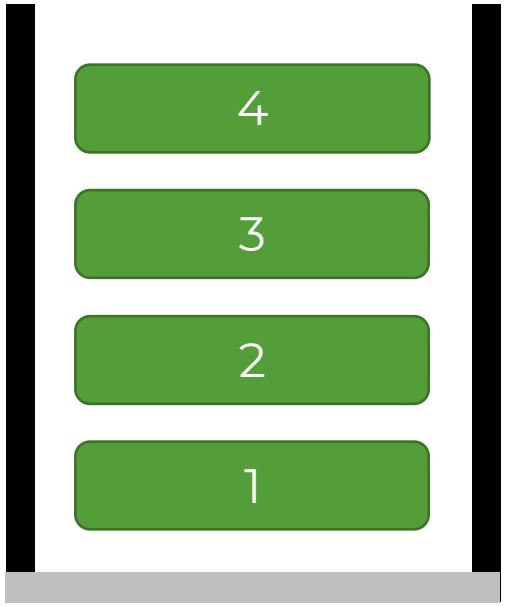
Вызов рекурсивных функций часто удобно осуществлять с помощью стека

Кроме того, он часто используется для обхода других, более сложных структур данных, например, дерево или граф

# Очередь (queue)

**Очередь** – это **абстрактная** структура данных, которая представляет собой набор элементов, упорядоченных по принципу **FIFO**.

***F****irst* **I**n – **F**irst **O**ut.



Какие операции нужны для очереди?

**Enqueue** O(1)

**Dequeue** O(1)

**Peek?**

**Count?**

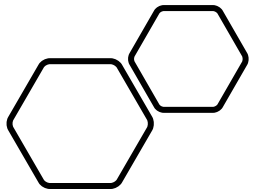
**Isempty?**

На какой структуре можно организовать очередь?

**Связный список**

**Стек**

**Цикл. массив**



# Пишем псевдокод!

Зачем нужна очередь?

Когда нужно совершить какие-то действия в порядке их поступления.

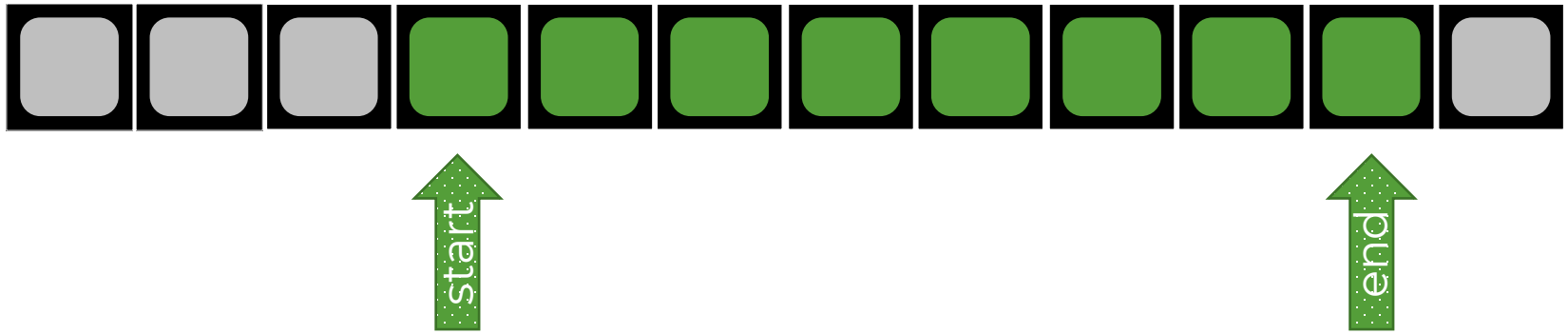
Кроме того, он часто используется для обхода других, более сложных структур данных, например, дерево или граф

# Реализация очереди на стеке

Нам понадобится не один стек, а целых два!

# Реализация очереди на массиве

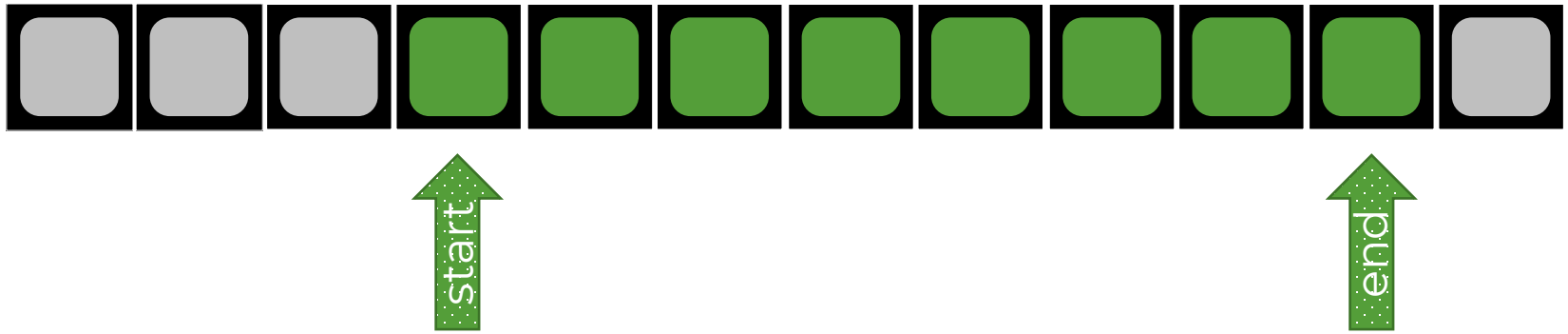
В реализации на массиве мы храним два указателя: **start** и **end**.



# Реализация очереди на массиве

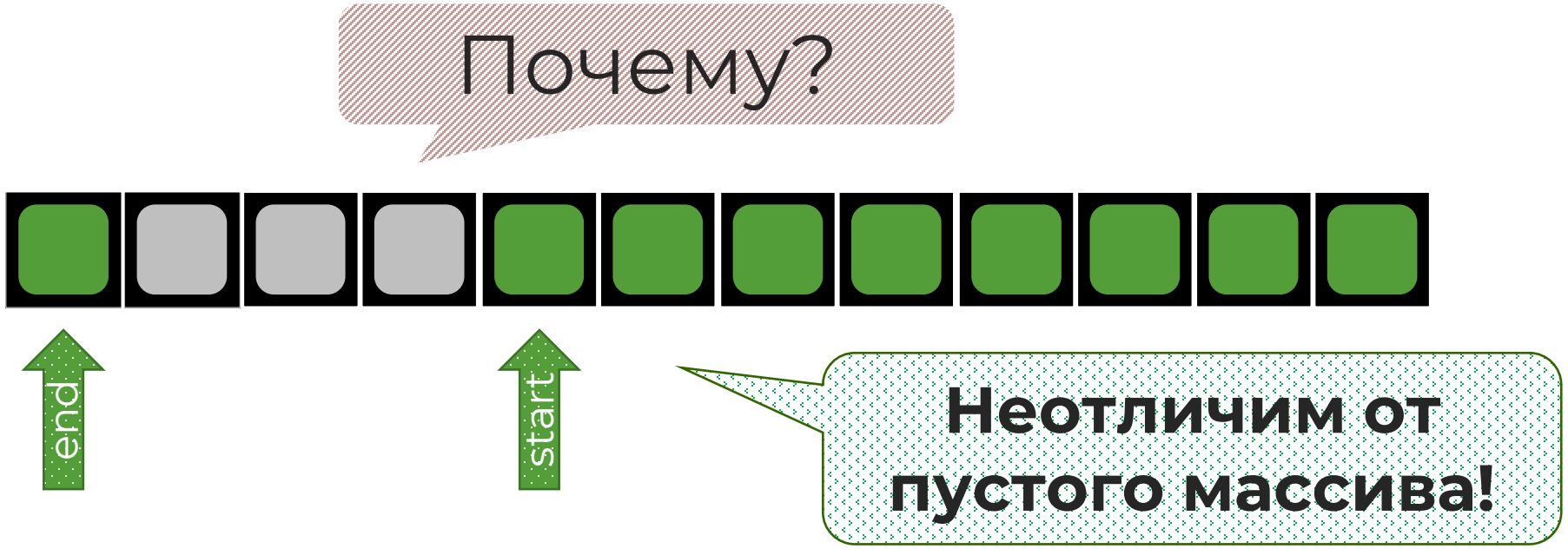
При удалении элемента из очереди в **Q[start]** записывается новый элемент очереди и **start** увеличивается на 1.

При добавлении **end** увеличивается на 1 и элемент записывается в **Q[end]**.



# Циклический массив

С таким алгоритмом одна из N ячеек всегда будет незанятой.



# Спасибо за внимание!