Память?

1. Массив

2. Оперативная

3. Дин. массив
(example vector в C++)

Динамические структуры данных

Почему не массив, чем он тут не подходит ?

- Размер заранее не известен
- Размер во время задачи постоянно изменяется
- Набор данных может быть полностью заменен или удален
- Часто нужно удалять данные
- Часто нужно добавлять данные

Операции удаления и добавления - ключевые!

Размер постоянно изменяется!

Выделить память под потенциально возможный размер – растрата ресурсов


ATTENTION, DEAR GUESTS!

СТЕК

СТЕК

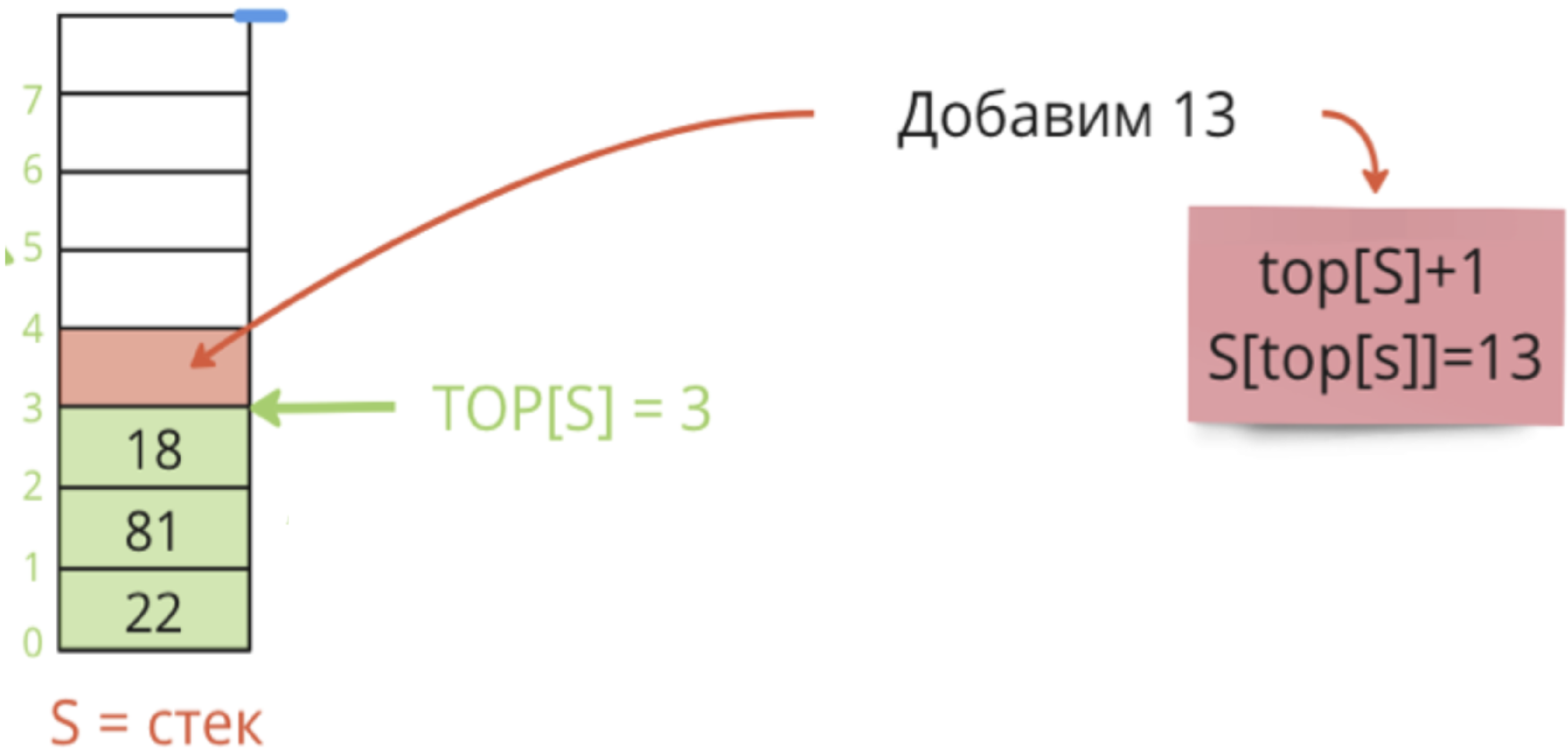
Если добавлять больше чем вместимость, то просто упадет с пирамидки

Могу снять только верхний элемент, что добавил последним

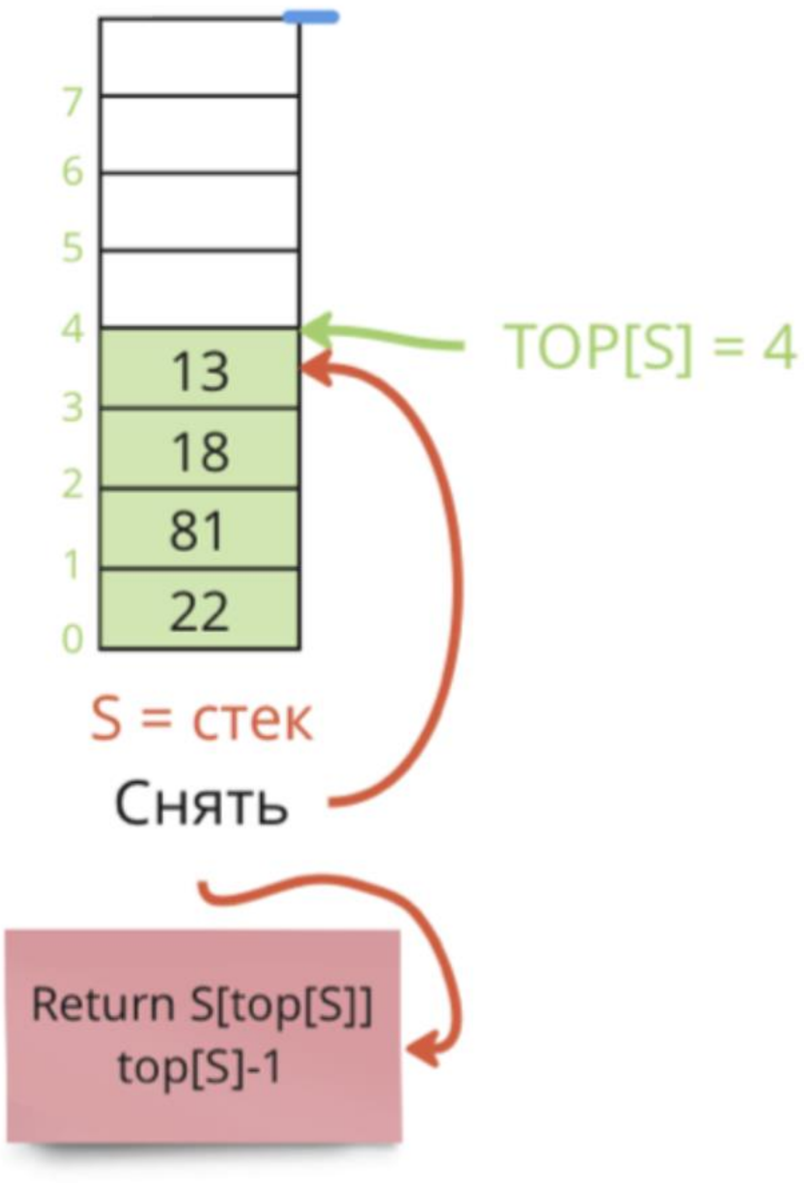
когда закончили элементы снять больше нечего

Стек

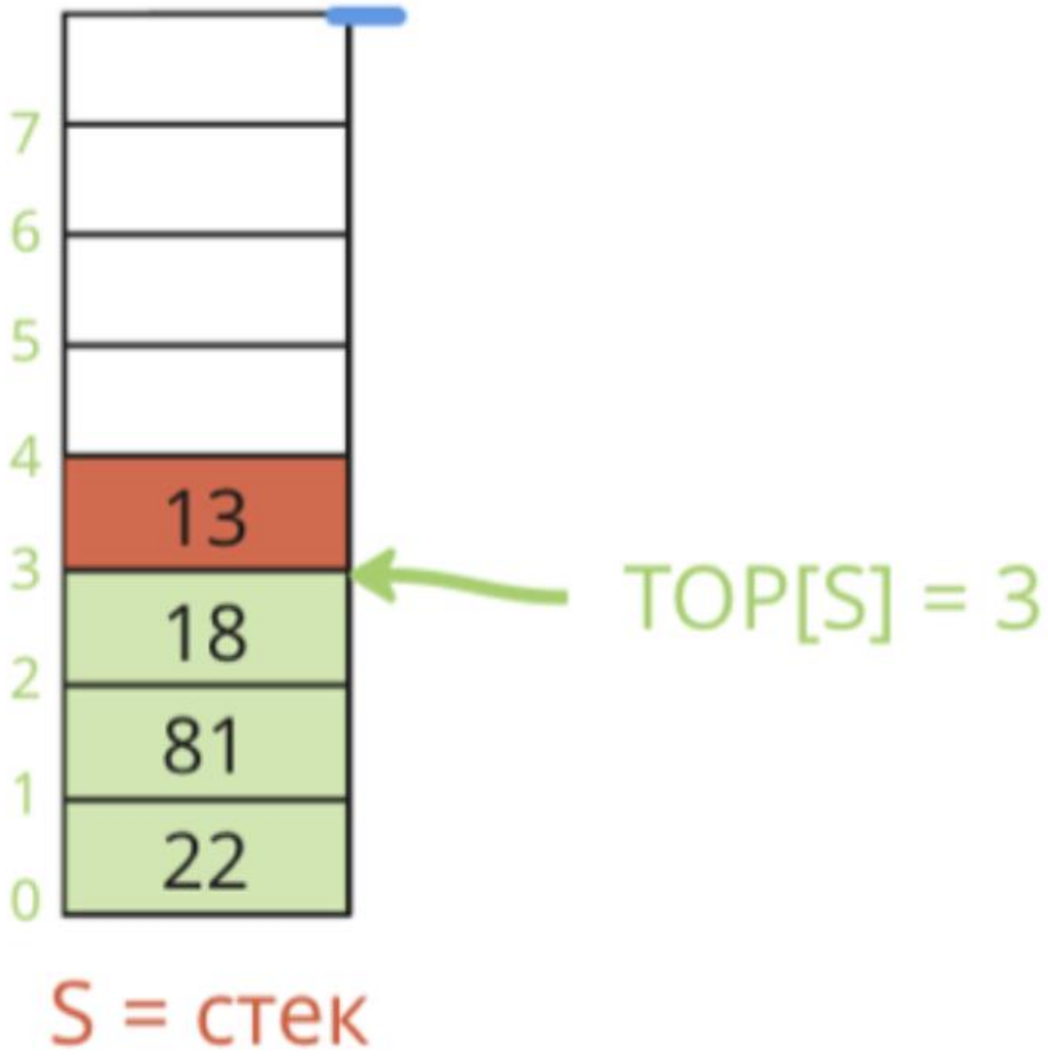
Стек (Добавление)



Стек: Снятие



Стек



Стек

1.Снять
2.Снять
3.Снять
7
6
5
4
13
3
18
2
81
1
22
0
TOP[S] = 0
S = стек
Стек пуст!

СТЕК:

LIFO (last in, first out) последний добавленный, будет удален первым

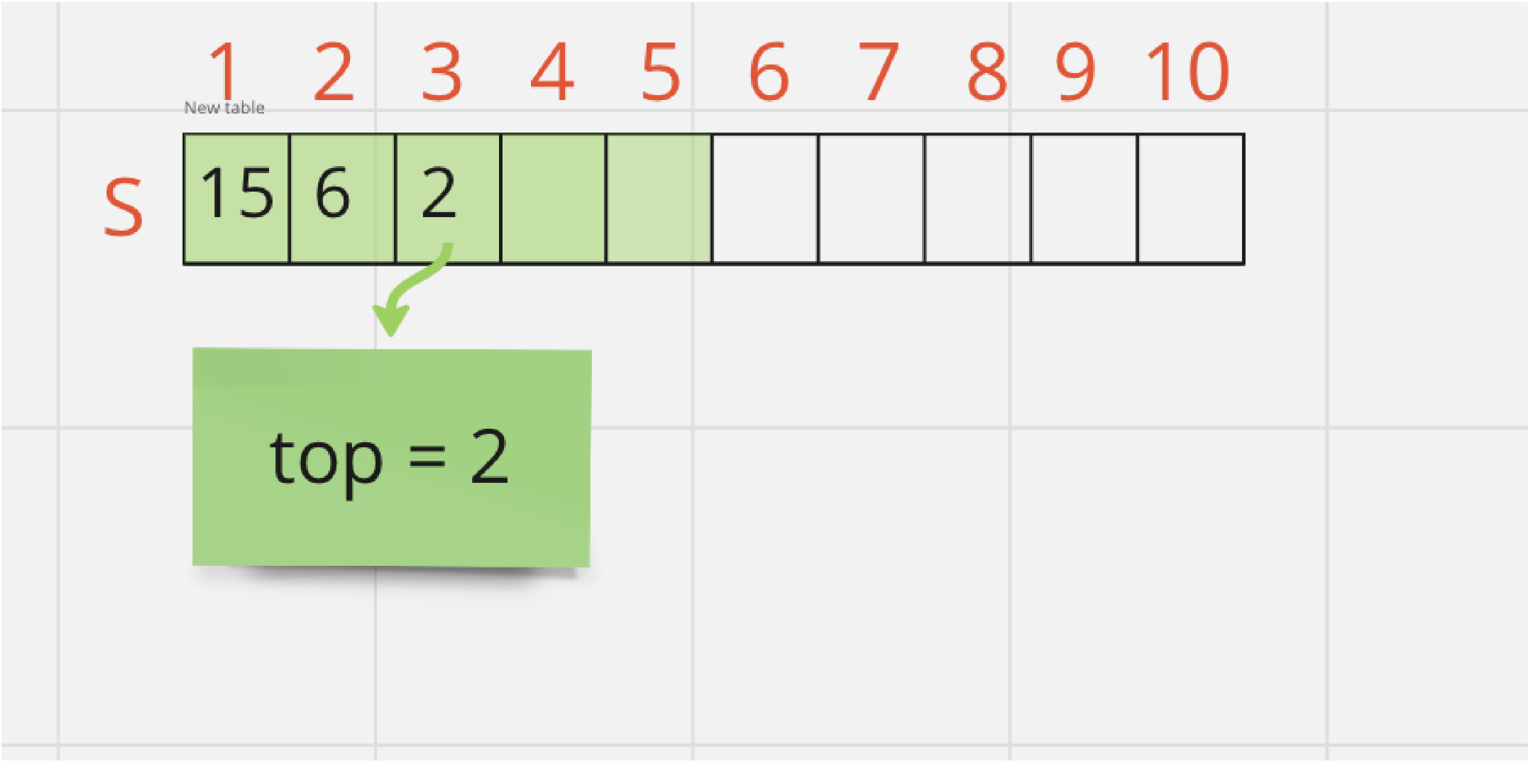
СТЕК: LIFO (last in, first out) последний добавленный, будет удален первым

Реализация:

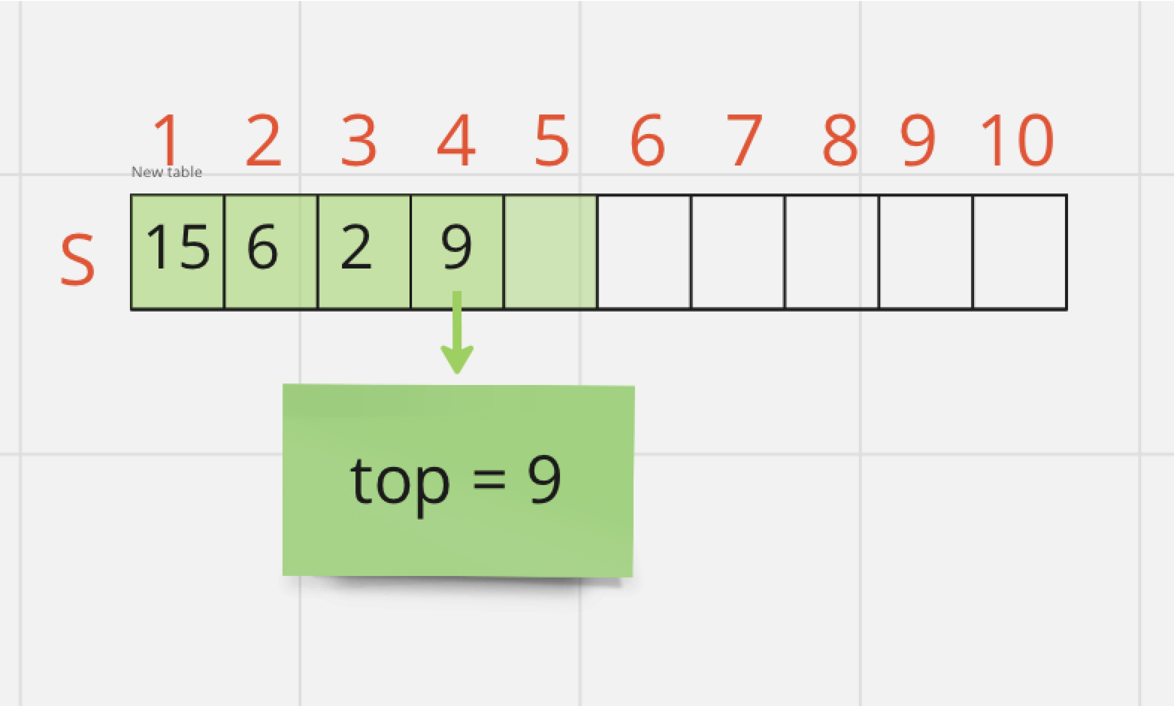
Массив **S** – это ячейки памяти ограниченного размера N,

Индекс последнего элемента **top[S]** – граница используемой памяти

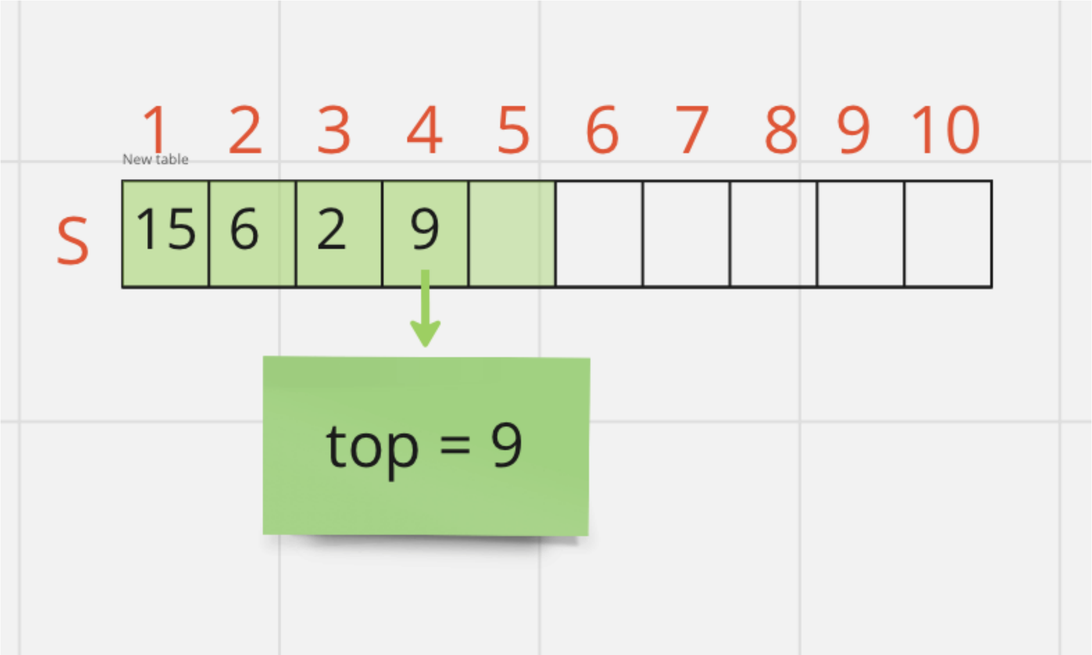
СТЕК: **LIFO** (last in, first out) последний добавленный, будет удален первым

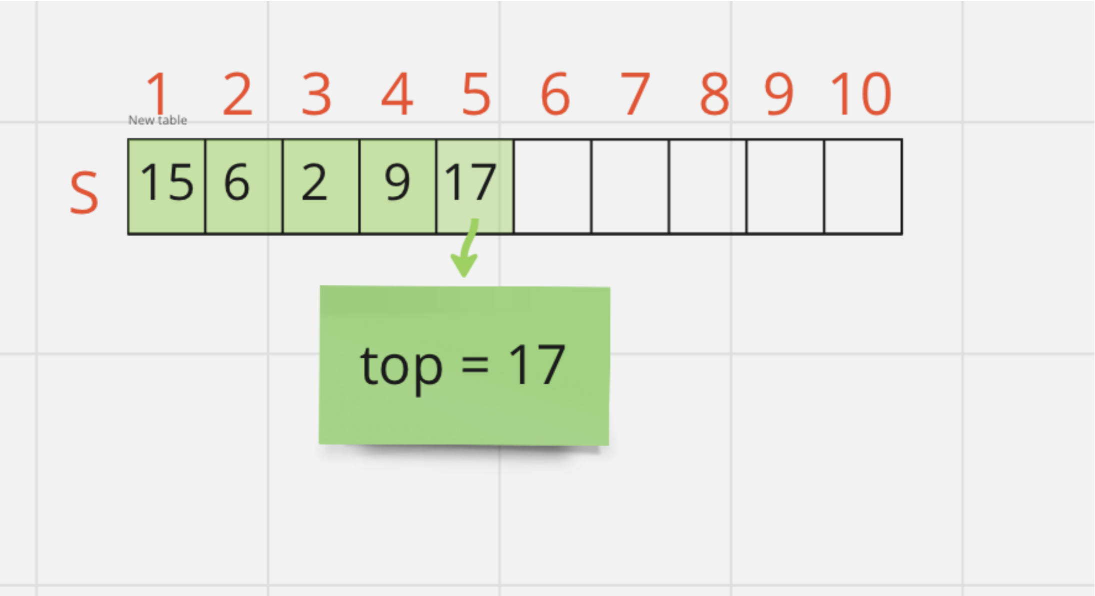


СТЕК: **LIFO** (last in, first out) последний добавленный, будет удален первым



СТЕК: **LIFO** (last in, first out) последний добавленный, будет удален первым





$\textsc{Push}(S, x)$

$top[S] \leftarrow top[S] + 1$

$S[top[S]] \leftarrow x$

ПЕРЕПОЛНЕНИЕ

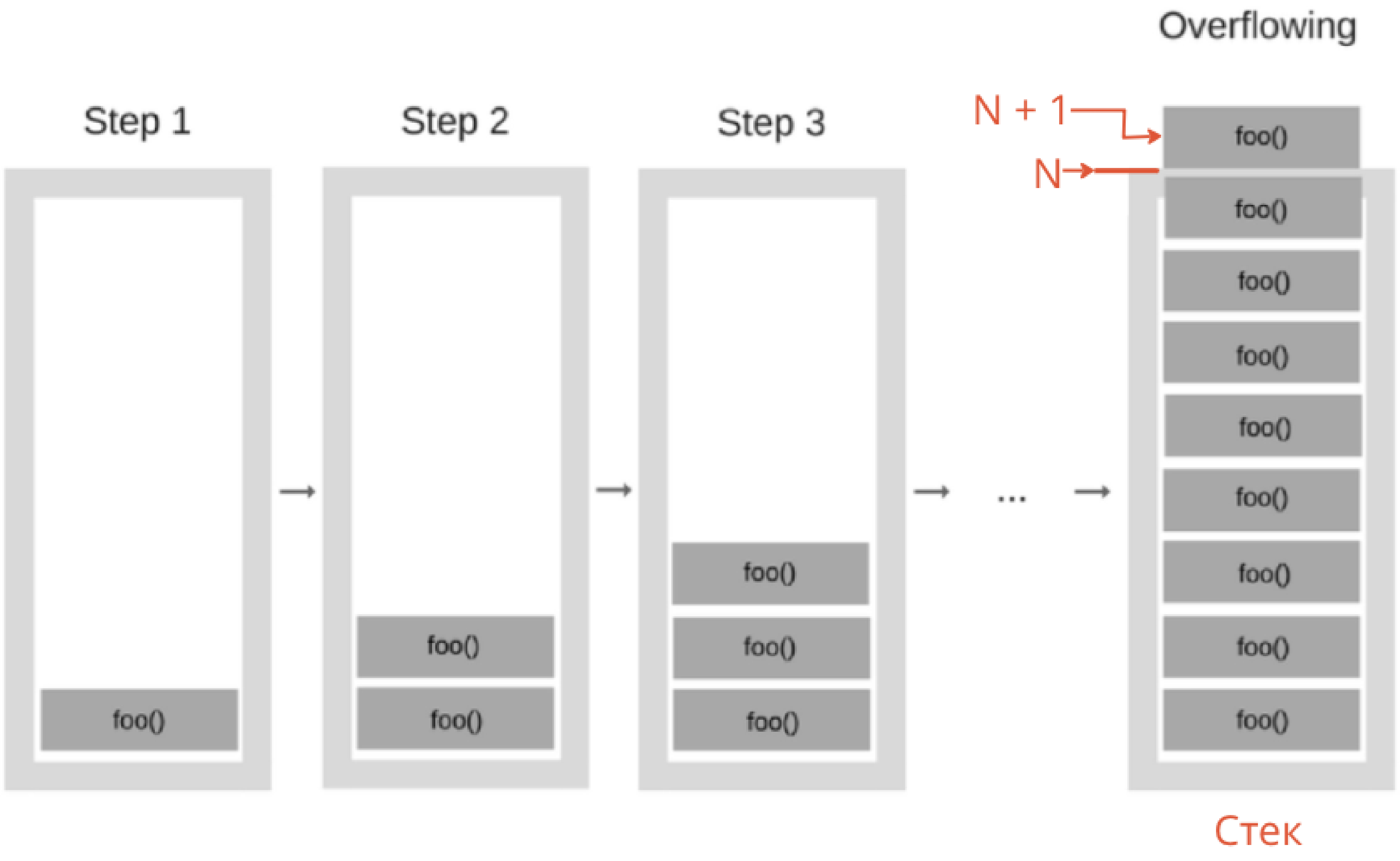
! foo() -> рекурсивные вызовы

!Ограниченный объем памяти на каждом устройстве: следим за тем, чтобы переполнение стало «видимой» ошибкой!

if top[S]+1>N

then “overflowing”

Записать в занятые адреса нельзя

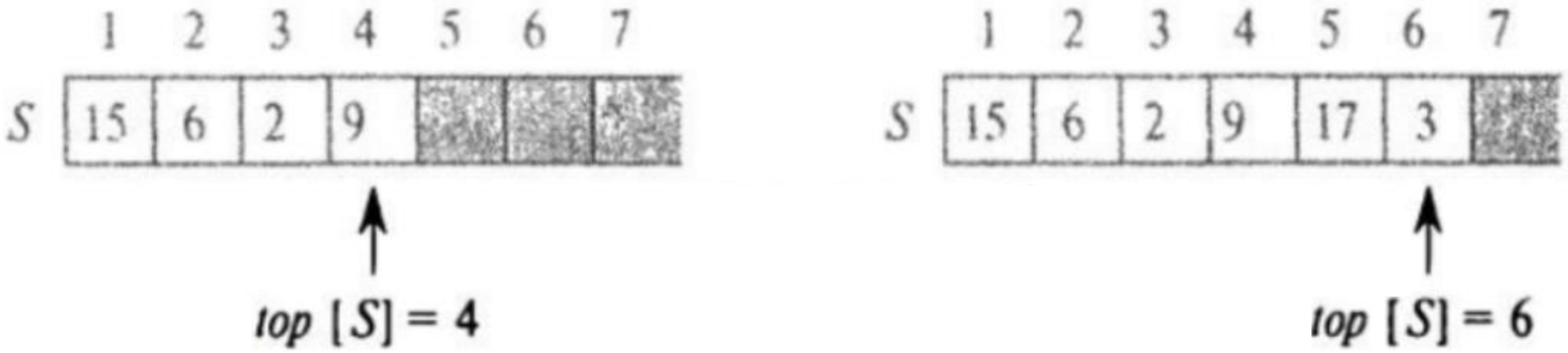


СТЕК: **LIFO** (last in, first out) последний добавленный, будет удален первым

Реализация:

Массив **S** – это ячейки памяти ограниченного размера N,

Индекс последнего элемента **top[S]** – граница используемой памяти



$\text{PUSH}(S, x)$

1 $top[S] \leftarrow top[S] + 1$

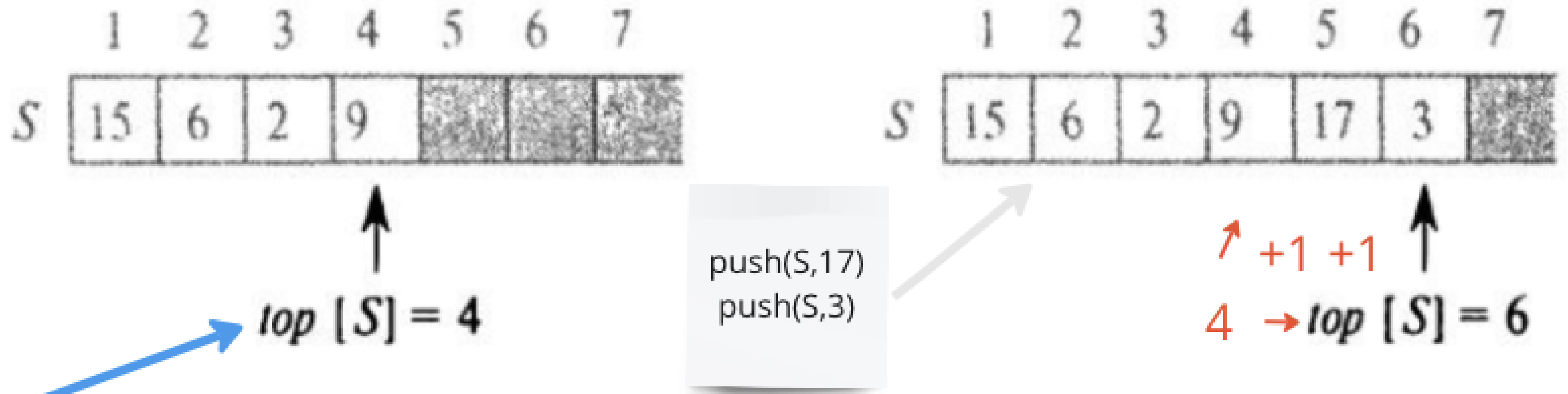
2 $S[top[S]] \leftarrow x$

СТЕК: **LIFO** (last in, first out) последний добавленный, будет удален первым

Реализация:

Массив **S** – это ячейки памяти ограниченного размера N,

Индекс последнего элемента **top[S]** – граница используемой памяти



Верхушка стека

$\text{PUSH}(S, x)$

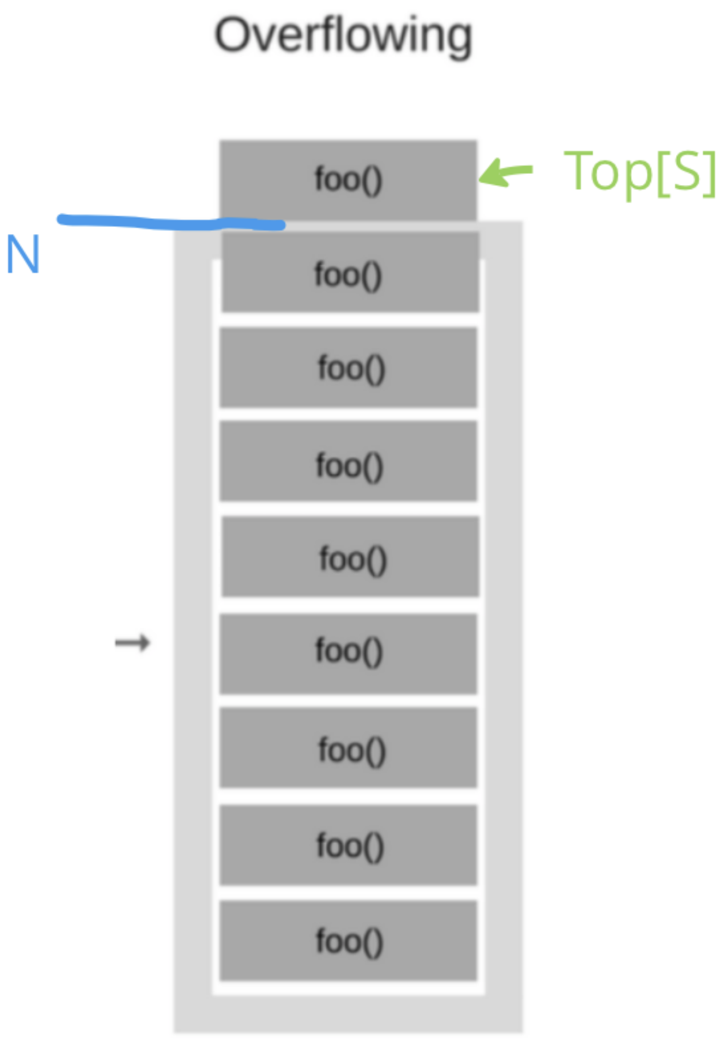
1 $top[S] \leftarrow top[S] + 1$

2 $S[top[S]] \leftarrow x$

Адрес следующего элемента

Записываем над ранее добавленными

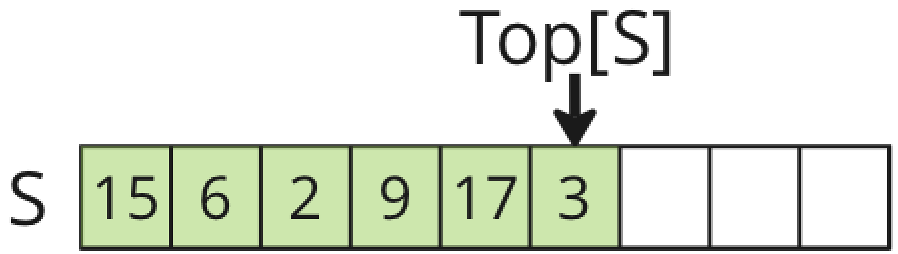
Как проверить на переполнение?



Если Top[S] > N, то стек переполнен

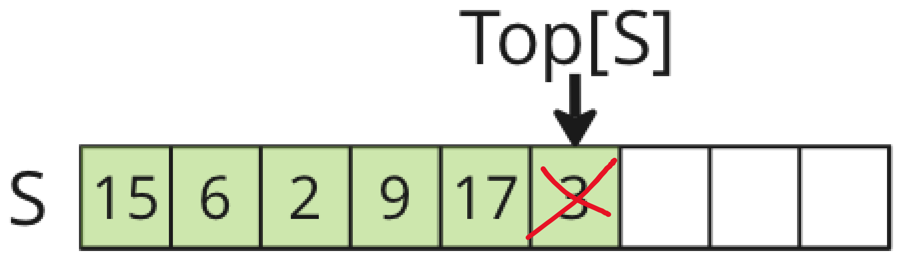
СТЕК: Удаление

```
STACK_EMPTY(S)
if top[S] = 0
    then return TRUE
    else return FALSE
```



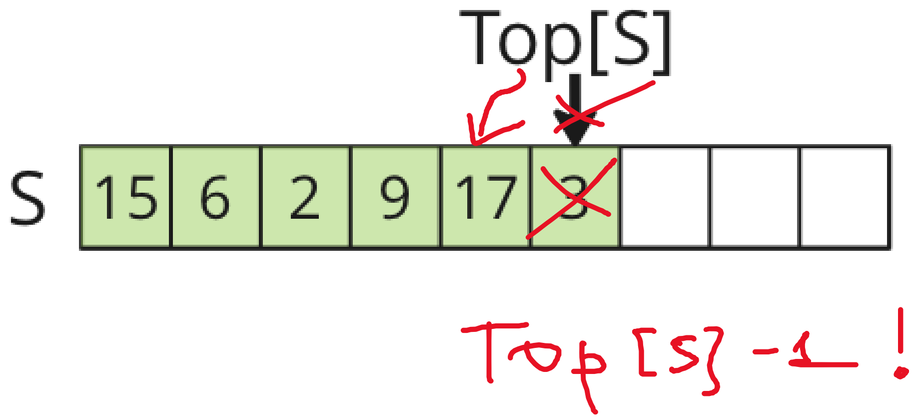
СТЕК: Удаление

```
STACK_EMPTY(S)
if top[S] = 0
    then return TRUE
    else return FALSE
```



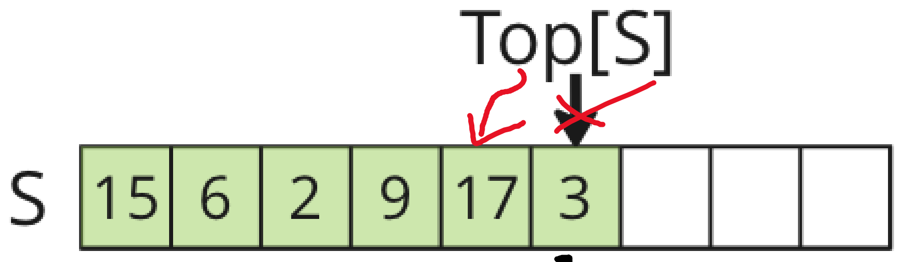
СТЕК: Удаление

```
STACK_EMPTY(S)
if top[S] = 0
    then return TRUE
    else return FALSE
```



СТЕК: Удаление

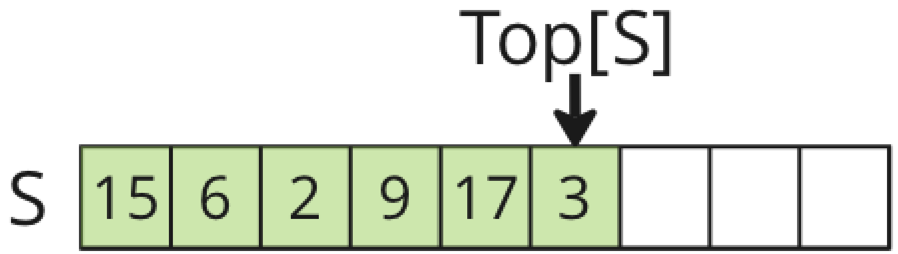
```
STACK_EMPTY(S)
if top[S] = 0
    then return TRUE
    else return FALSE
```



Top[S]-1 !

а вот значение само не удаляем!

СТЕК: **Удаление**




```
STACK_EMPTY(S)
if top[S] = 0
    then return TRUE
    else return FALSE
```

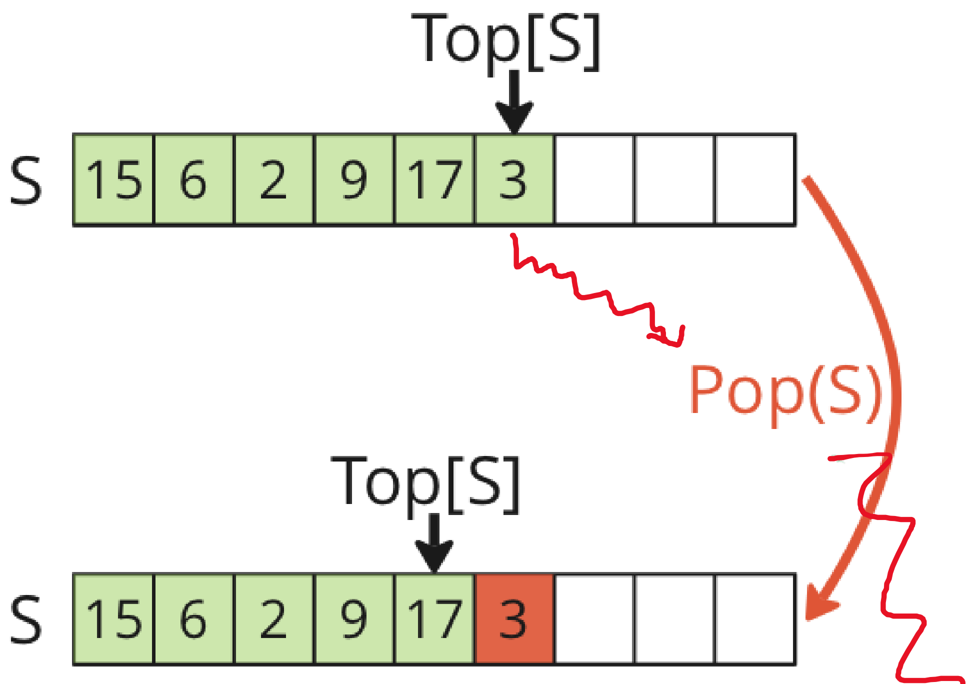
```
POP(S)
if STACK_EMPTY(S)
    then error "underflow"
    else top[S] ← top[S] − 1      ← переместим top
         return S[top[S] + 1]     ← вернём значение
```

СТЕК: Удаление

```
STACK_EMPTY(S)
  if top[S] = 0
     then return TRUE
     else return FALSE
```

```
POP(S)
  if STACK_EMPTY(S)
     then error "underflow"
     else top[S] ← top[S] − 1
          return S[top[S] + 1]
```

Возвращаем значение снятого элемента



Данные в ячейке оставляем, при первой неоходимости данные будут перезаписаны

Указатель верхушки стека показывает какие сначала адресов стека заняты = их трогать нельзя

Все остальные адреса, занятые данными, могут быть перезаписаны = адреса свободны

return

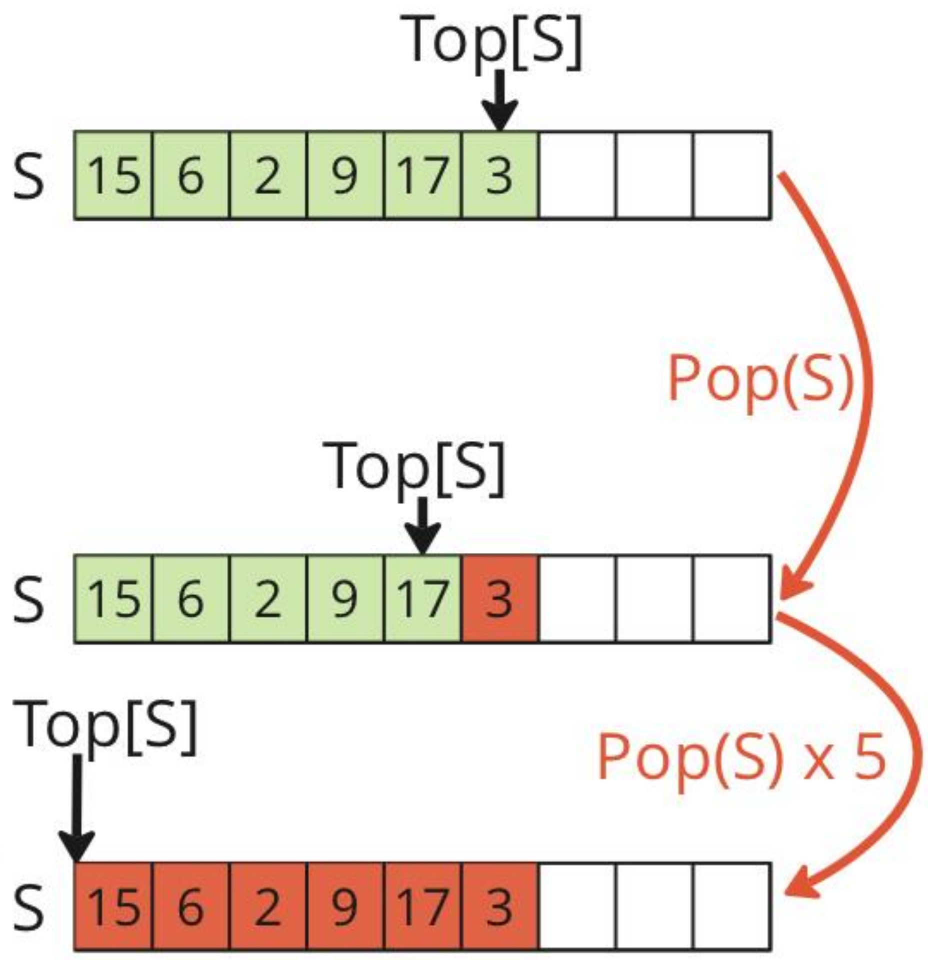
3

СТЕК: **Удаление**

```
STACK_EMPTY(S)
if top[S] = 0
    then return TRUE
    else return FALSE
```

```
POP(S)
if STACK_EMPTY(S)
    then error "underflow"
    else top[S] ← top[S] − 1
         return S[top[S] + 1]
```

Возвращаем значение снятого элемента



Данные в ячейке оставляем, при первой неоходимости данные будут перезаписаны

Указатель верхушки стека показывает какие сначала адресов стека заняты = их трогать нельзя

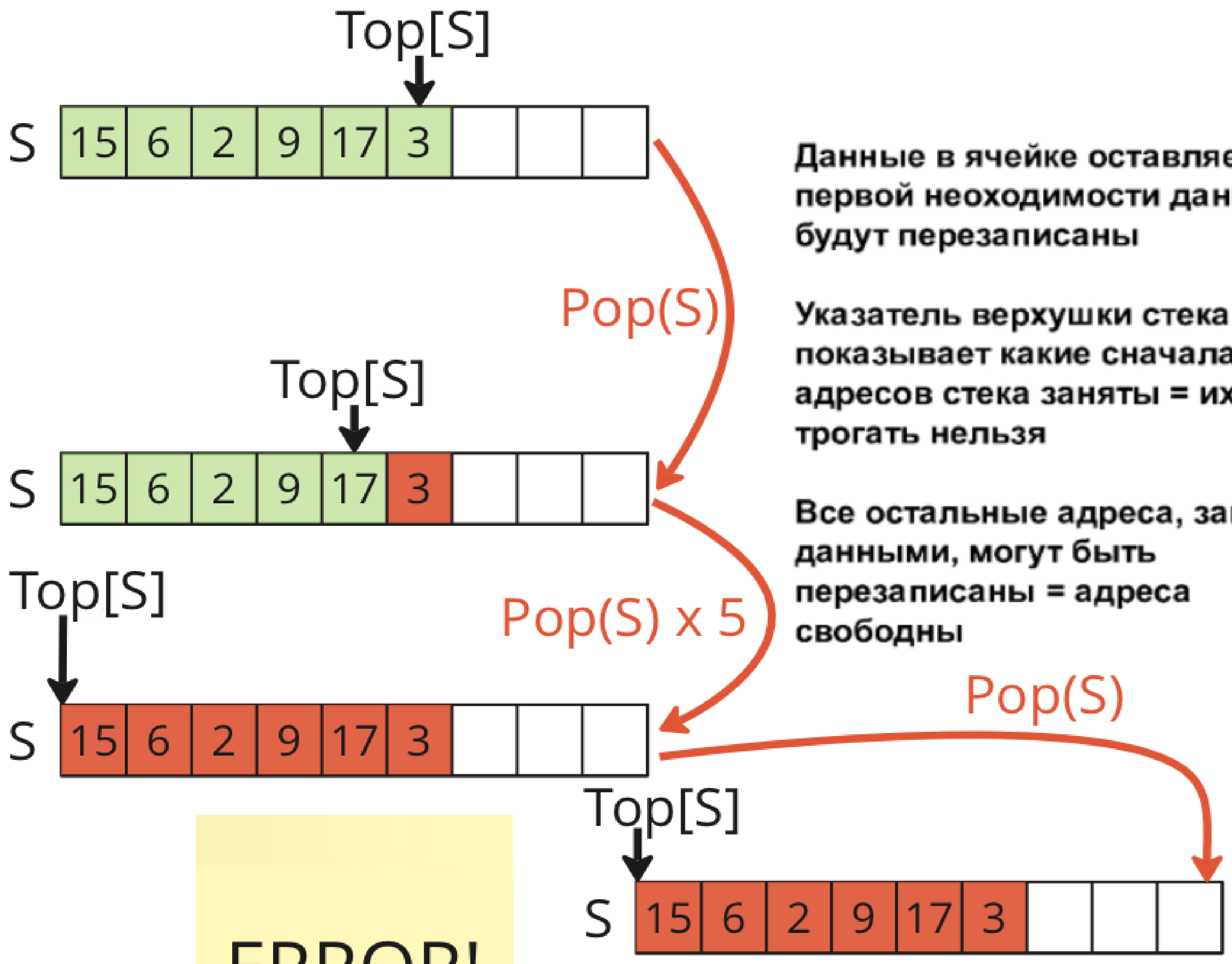
Все остальные адреса, занятые данными, могут быть перезаписаны = адреса свободны

СТЕК: Удаление

```
STACK_EMPTY(S)
  if top[S] = 0
     then return TRUE
     else return FALSE
```

```
POP(S)
  if STACK_EMPTY(S)
     then error "underflow"
     else top[S] ← top[S] − 1
          return S[top[S] + 1]
```

Возвращаем значение снятого элемента



Данные в ячейке оставляем, при первой неоходимости данные будут перезаписаны

Указатель верхушки стека показывает какие сначала адресов стека заняты = их трогать нельзя

Все остальные адреса, занятые данными, могут быть перезаписаны = адреса свободны

ERROR!

снять с пустого стека НЕЛЬЗЯ

ОЦЕНКА ОПЕРАЦИЙ С СТРУКТУРОЙ ПО ВРЕМЕНИ

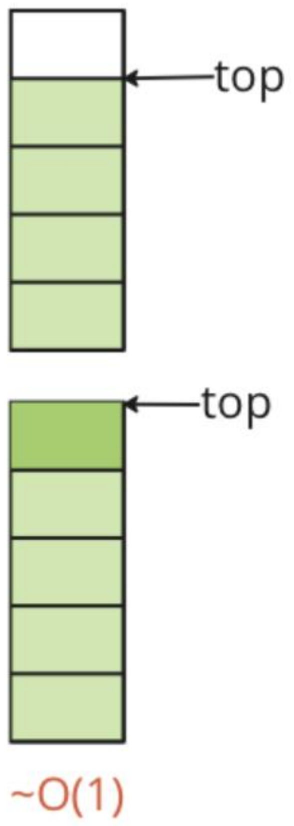
Удаление	Добавление	Поиск
O(1)	**O(1)**	**O(n)**

ОЦЕНКА ОПЕРАЦИЙ С СТРУКТУРОЙ ПО ВРЕМЕНИ

Удаление	Добавление	Поиск
O(1)	O(1)	O(n)

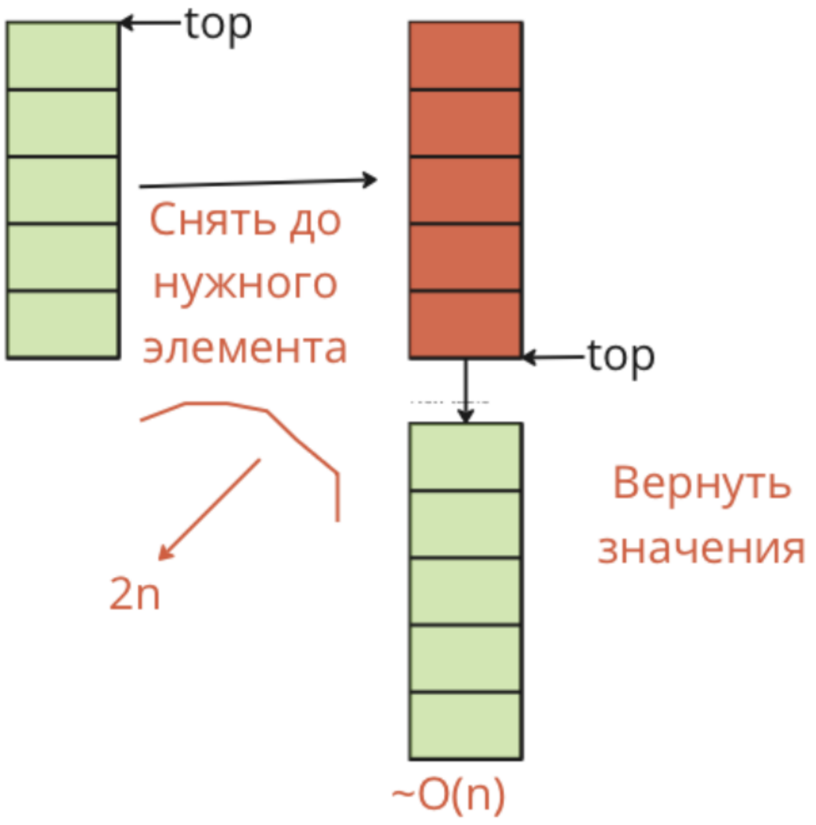
ОЦЕНКА ОПЕРАЦИЙ С СТРУКТУРОЙ ПО ВРЕМЕНИ

Удаление	Добавление	Поиск
O(1)	**O(1)**	**O(n)**



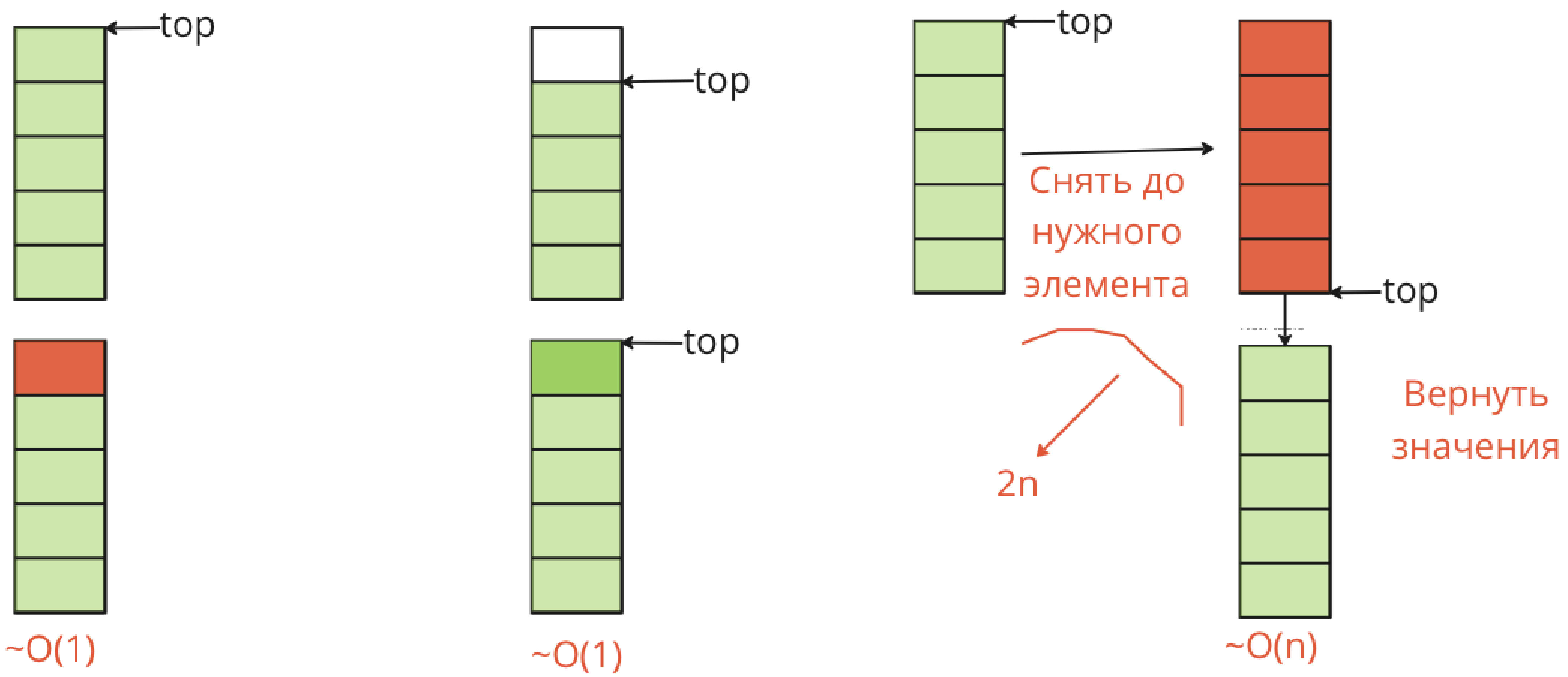
ОЦЕНКА ОПЕРАЦИЙ С СТРУКТУРОЙ ПО ВРЕМЕНИ

Удаление	Добавление	Поиск
O(1)	O(1)	O(n)



ОЦЕНКА ОПЕРАЦИЙ С СТРУКТУРОЙ ПО ВРЕМЕНИ

Удаление	Добавление	Поиск
O(1)	O(1)	O(n)




ATTENTION, DEAR GUESTS!

Тут бывал каждый!

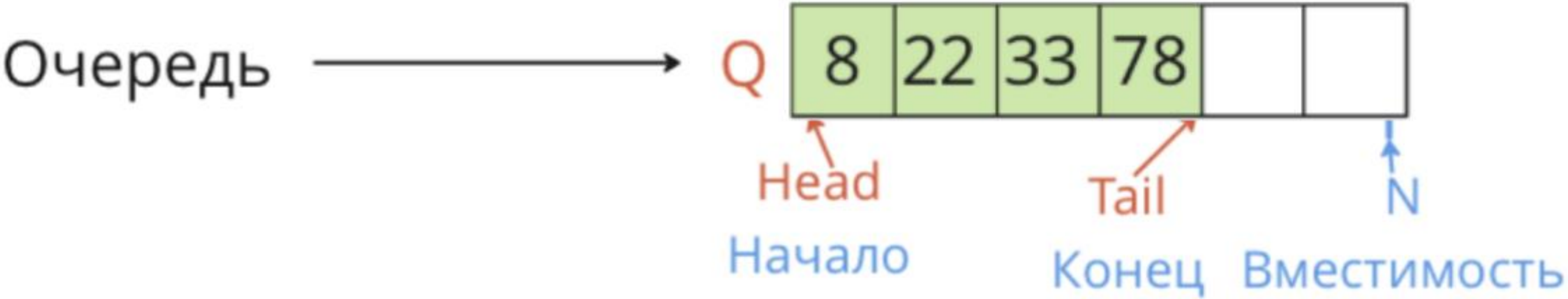
Есть начало очереди

Есть конец очереди

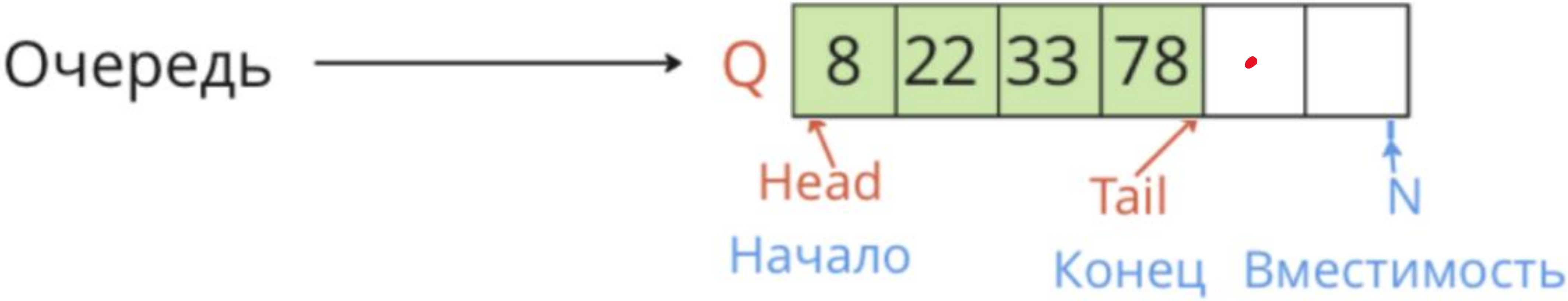
Очердь может быть пустой

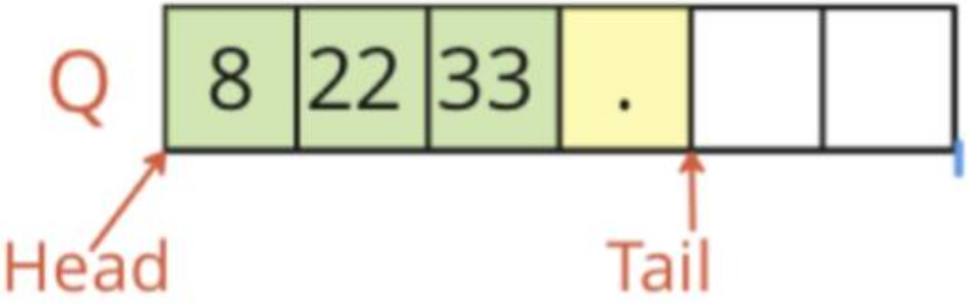
Рано пришел, быстрее попал в музей

Очередь



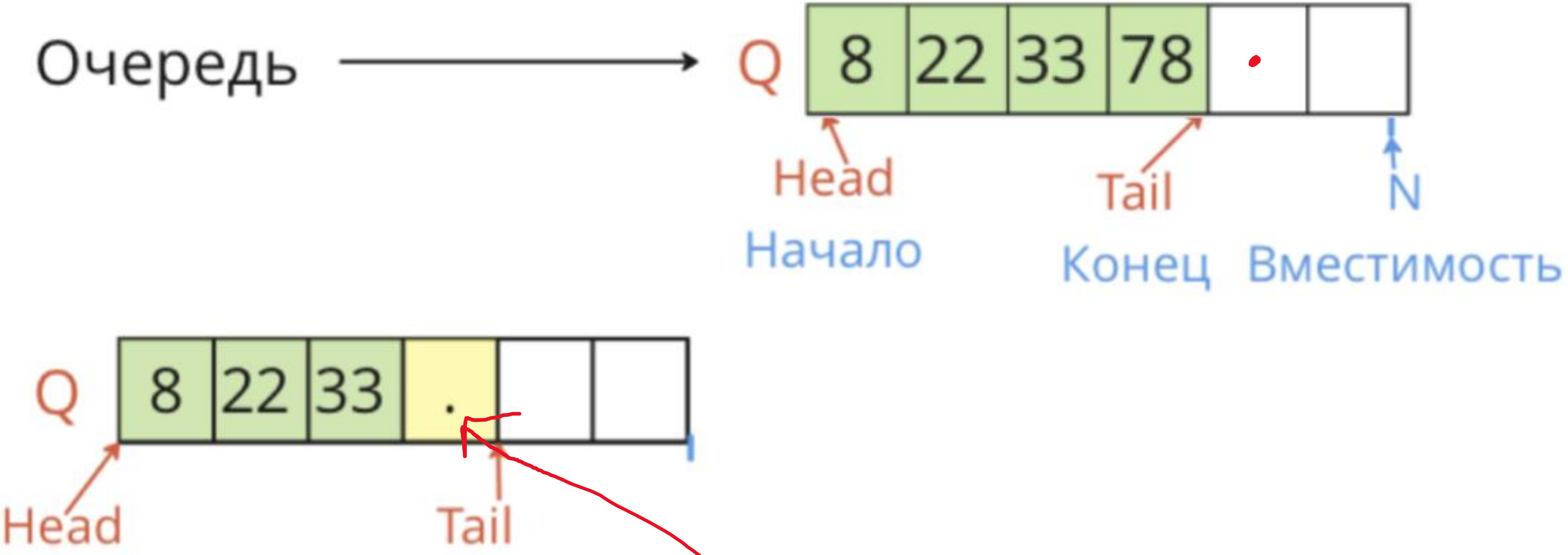
Очередь: Добавление





1) Новый пришел: добавим(78)

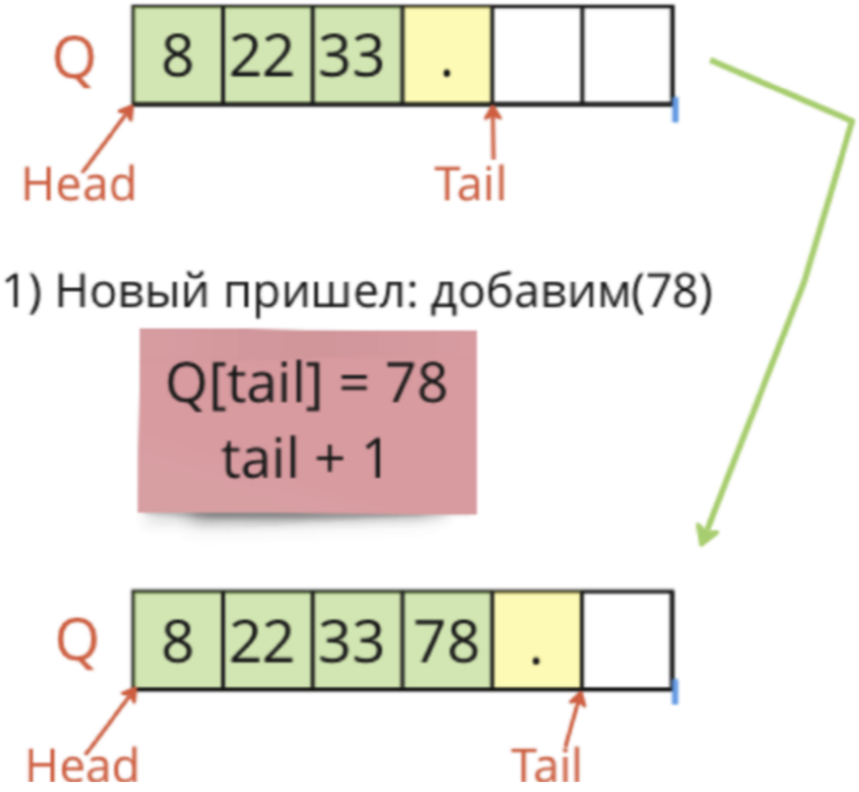
Очередь: Добавление



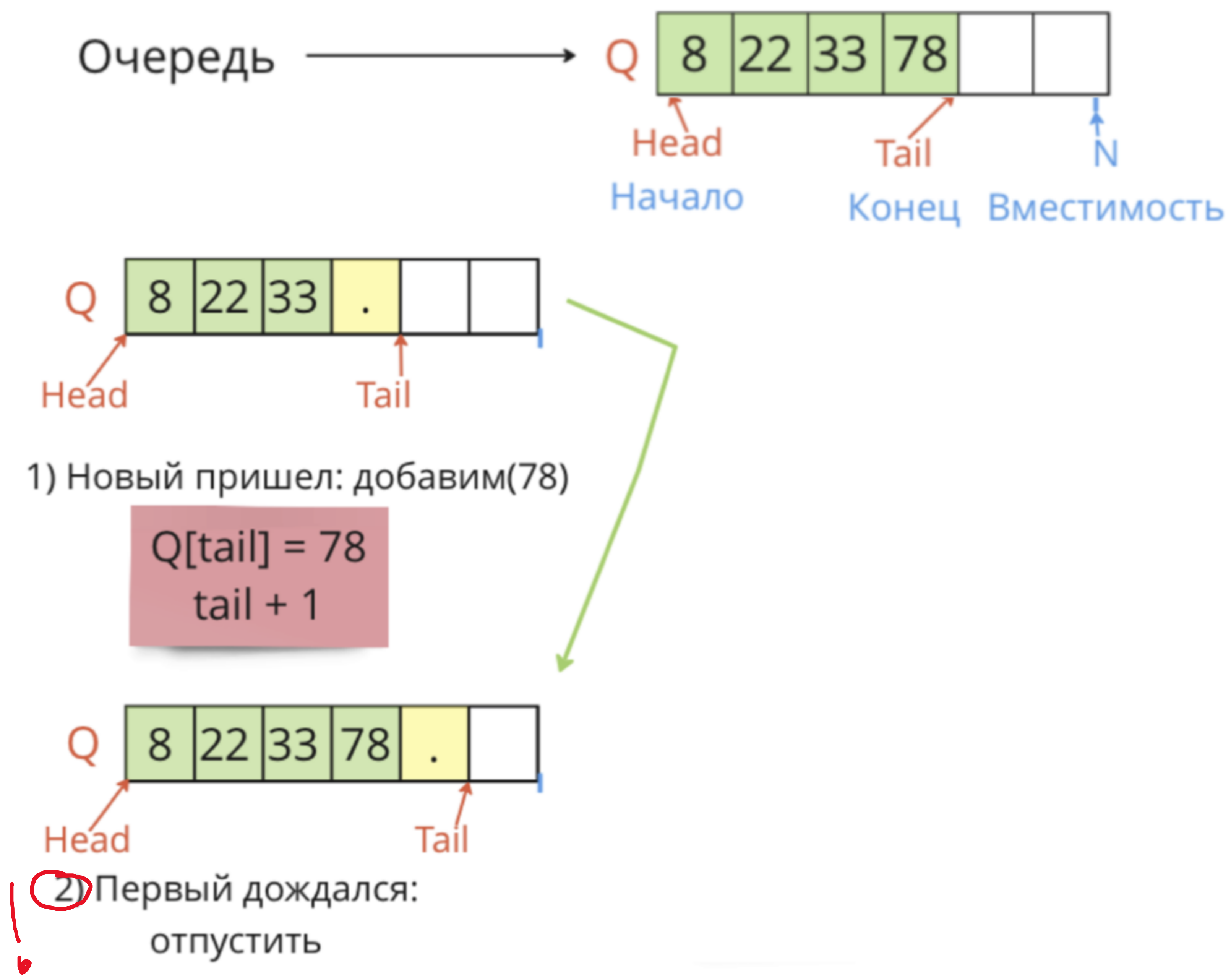
1) Новый пришел: добавим(78)

в конец !!!

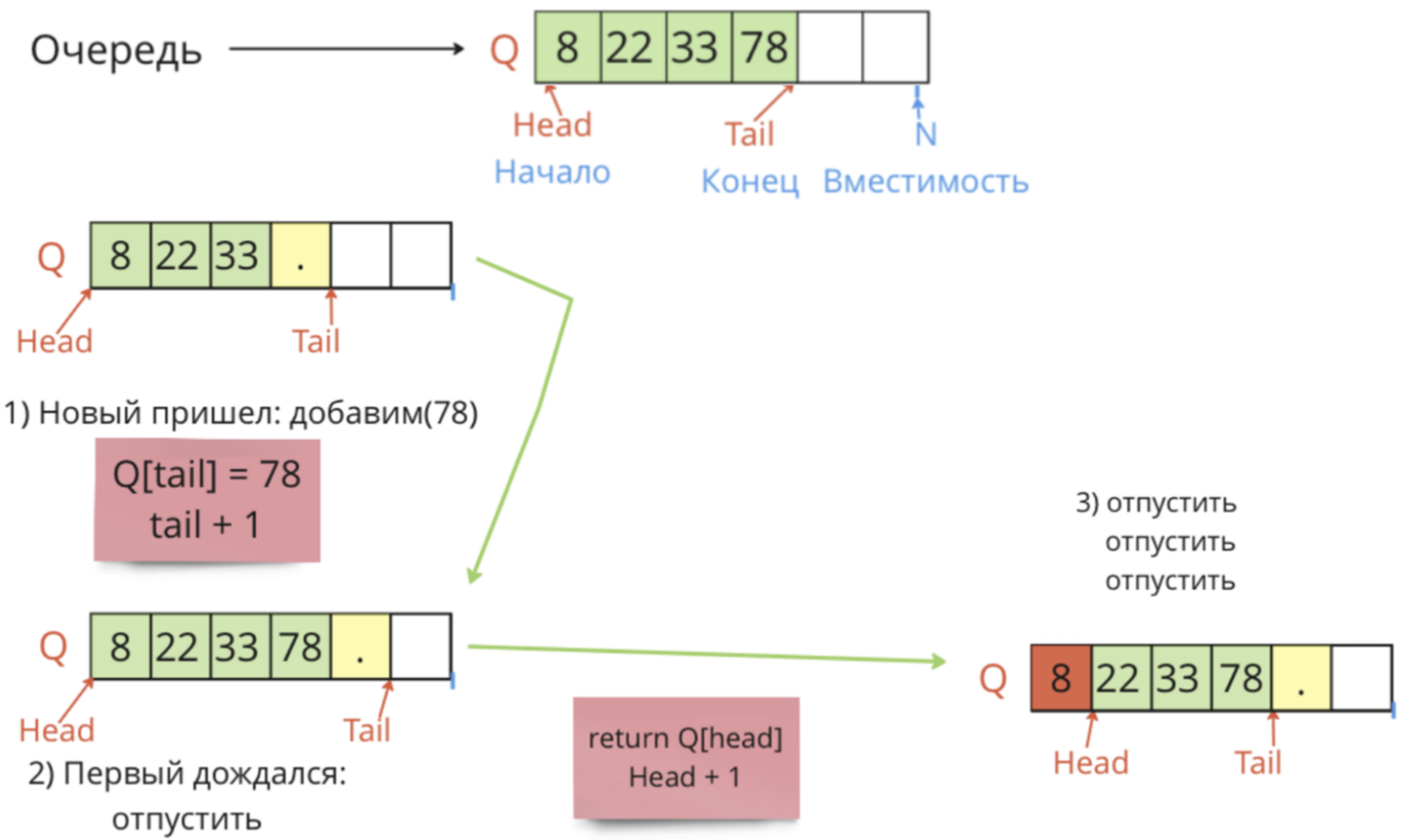
Очередь: Добавление



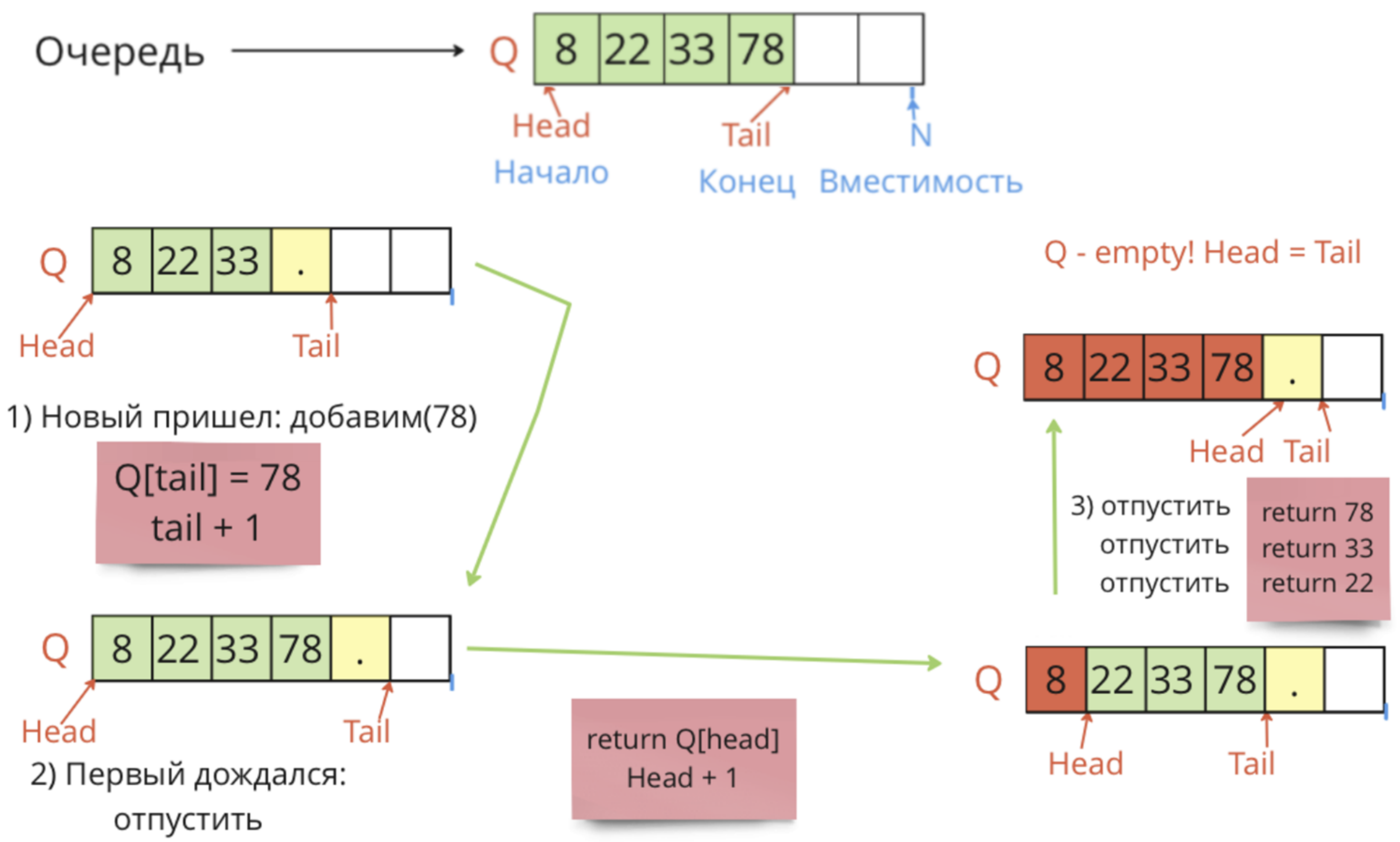
Очередь: Добавление



Очередь: Добавление



Очередь


Очередь
Q
8 22 33 78
Head
Начало
Tail
Конец
N
Вместимость
Q
8 22 33 .
Head
Tail
1) Новый пришел: добавим(78)
Q[tail] = 78
tail + 1
Q
8 22 33 78 .
Head
Tail
2) Первый дождался:
отпустить
return Q[head]
Head + 1
Q - empty! Head = Tail
Q
8 22 33 78 .
Head
Tail
3) отпустить
отпустить
отпустить
return 78
return 33
return 22
Q
8 22 33 78 .
Head
Tail

ОЧЕРЕДЬ: **FIFO** (first in, first out) первый добавленный, будет удален первым

Реализация:

Массив **Q** – это ячейки памяти ограниченного размера N,

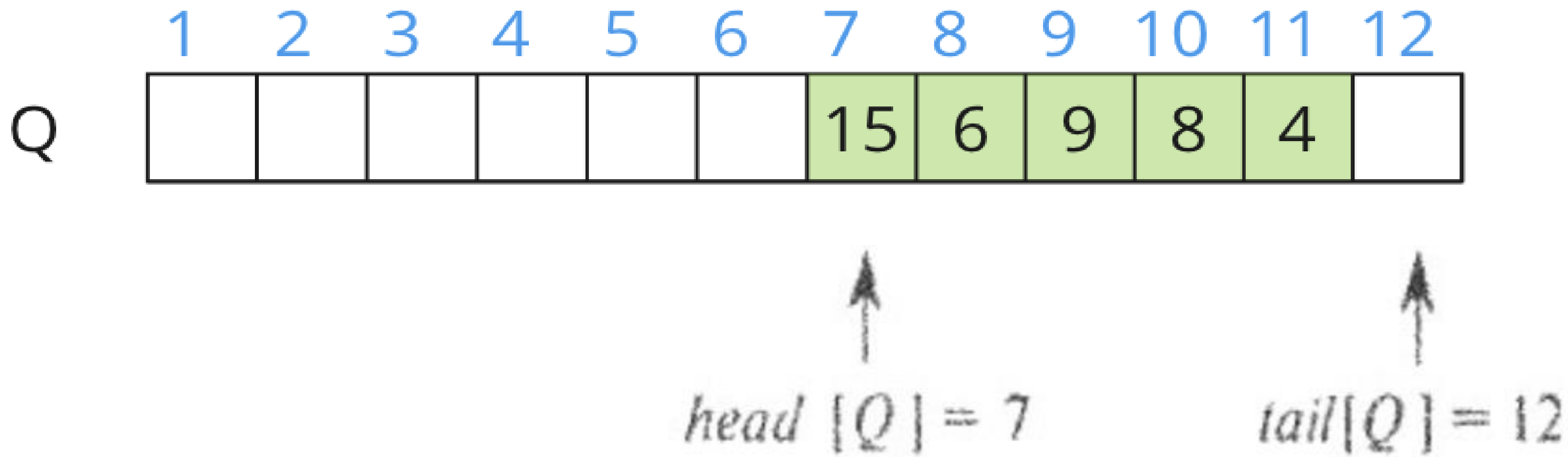
Индексы первого **Head[Q]** и последнего элементов **Tail[Q]** – граница используемой памяти

ОЧЕРЕДЬ: **FIFO** (first in, first out) первый добавленный, будет удален первым

Реализация:

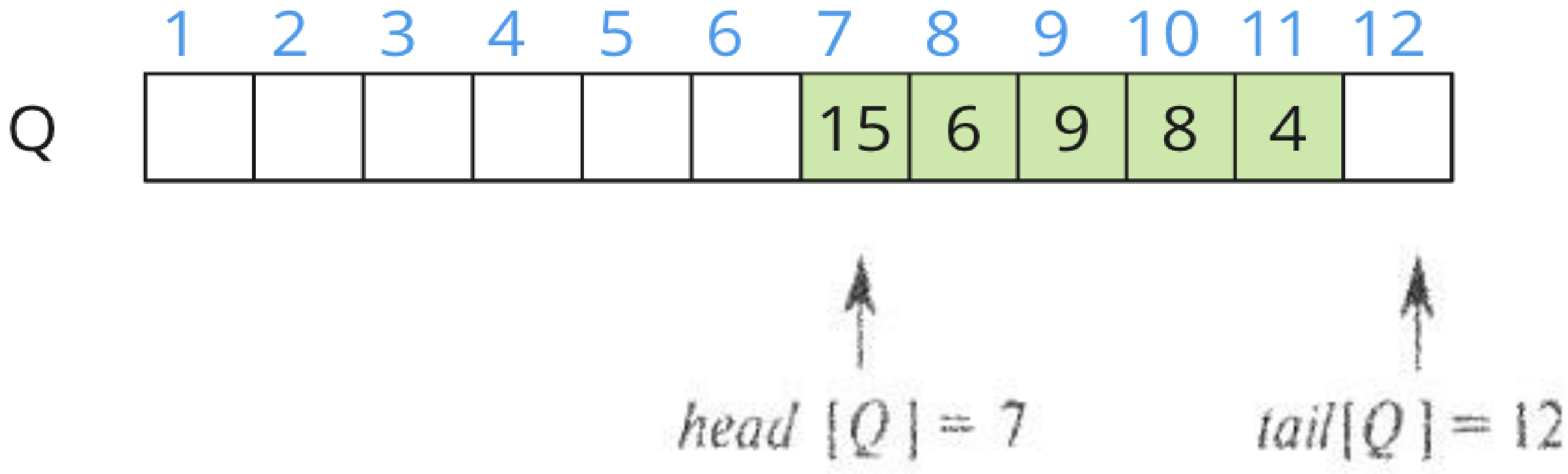
Массив **Q** – это ячейки памяти ограниченного размера N,

Индексы первого **Head[Q]** и последнего элементов **Tail[Q]** – граница используемой памяти



Вставка в ОЧЕРЕДЬ:

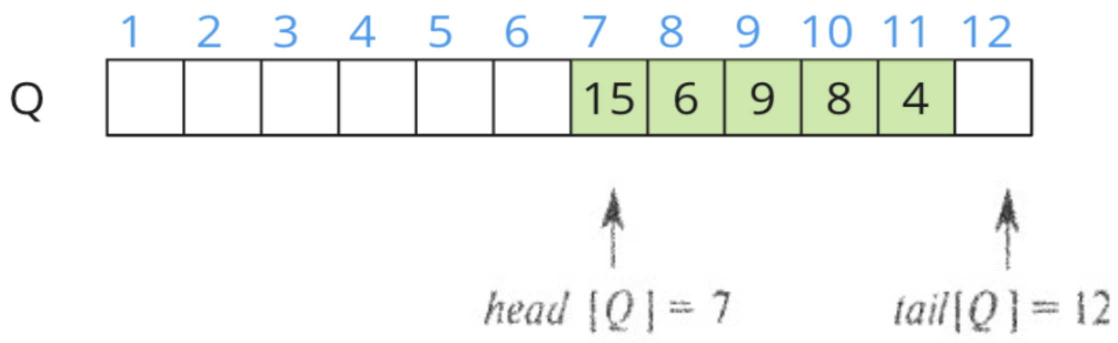
FIFO (first in, first out) первый добавленный, будет удален первым



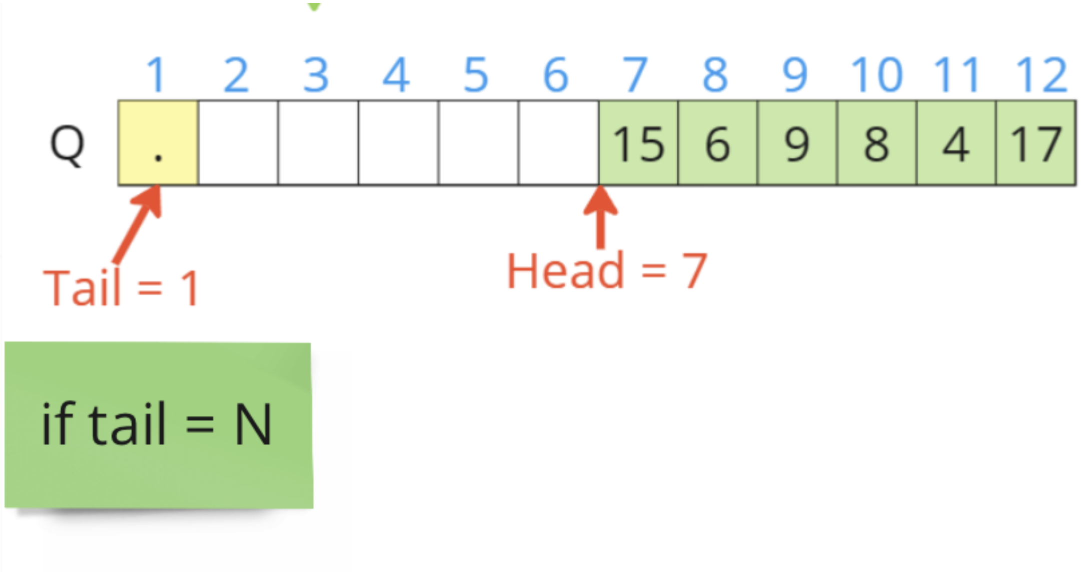
Вставим 17

Вставка в ОЧЕРЕДЬ:

FIFO (first in, first out) первый добавленный, будет удален первым

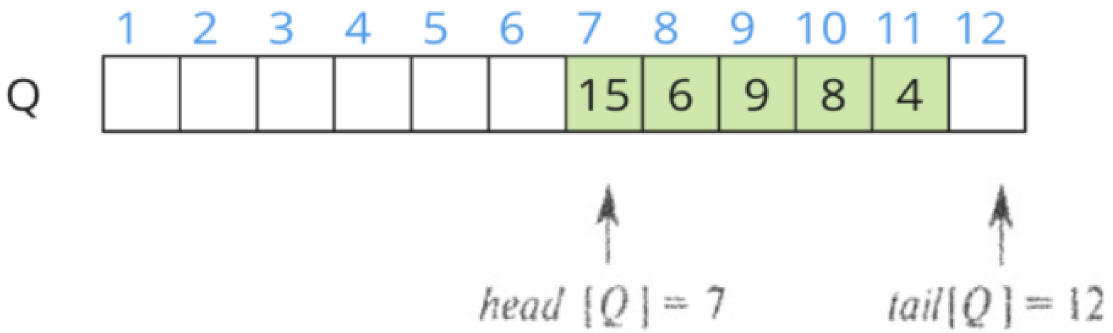


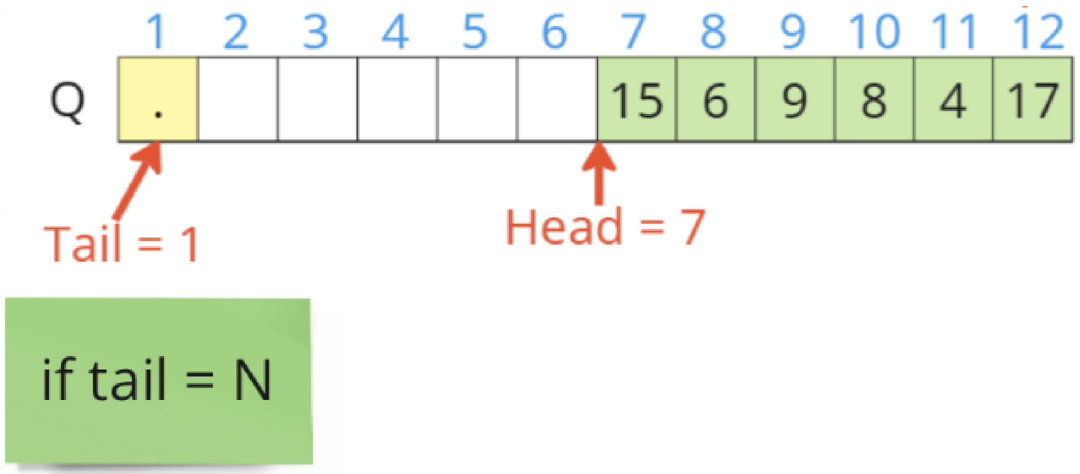
Вставим 17



Вставка в ОЧЕРЕДЬ:

FIFO (first in, first out) первый добавленный, будет удален первым

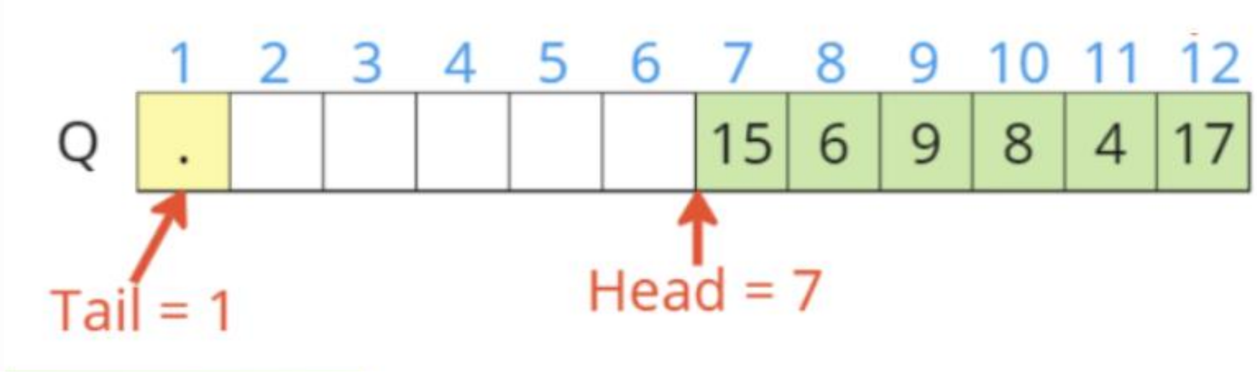




считаем, что блок памяти цикличен и
при достижении края переходим
вначало выделенных адресов

Вставка в ОЧЕРЕДЬ:

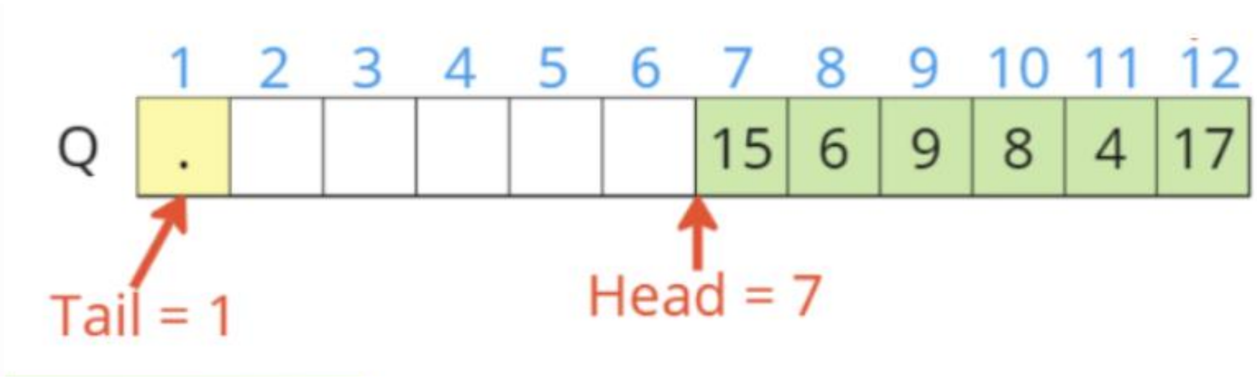
FIFO (first in, first out) первый добавленный, будет удален первым



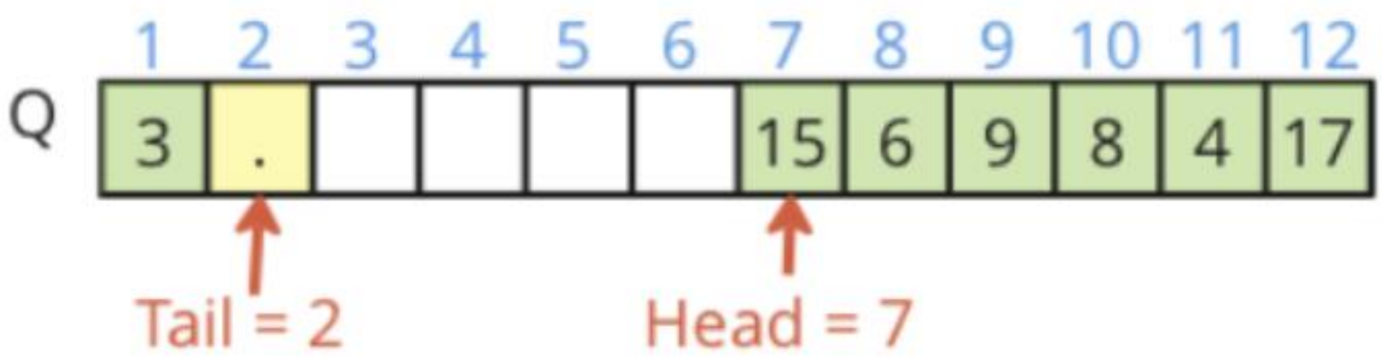
Добавим 3

Вставка в ОЧЕРЕДЬ:

FIFO (first in, first out) первый добавленный, будет удален первым



Добавим 3

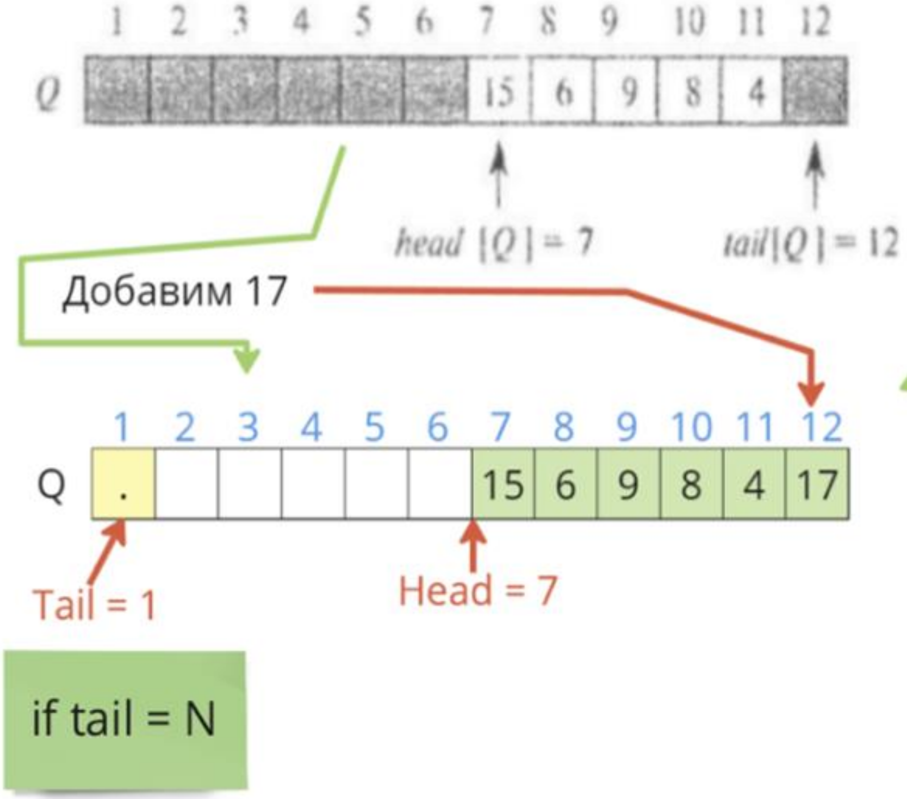


Вставка в ОЧЕРЕДЬ:

FIFO (first in, first out) первый добавленный, будет удален первым

Вставка в ОЧЕРЕДЬ:

FIFO (first in, first out) первый добавленный, будет удален первым



ENQUEUE(Q, x)

$Q[tail[Q]] \leftarrow x$ Запись данных

if $tail[Q] = length[Q]$

  then $tail[Q] \leftarrow 1$

  else $tail[Q] \leftarrow tail[Q] + 1$

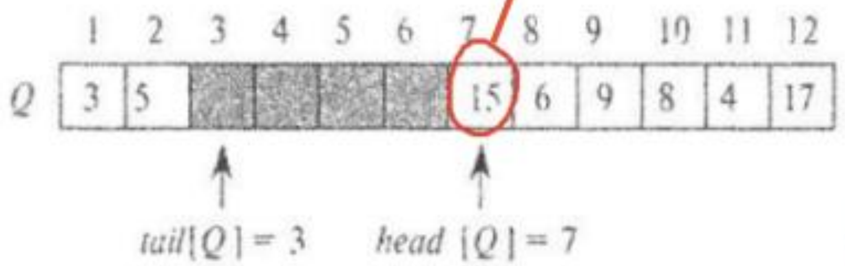
ОЧЕРЕДЬ: **FIFO** (first in, first out) первый добавленный, будет удален первым

Реализация:

Массив **Q** – это ячейки памяти ограниченного размера N,

Индексы первого **Head[Q]** и последнего элементов **Tail[Q]** – граница используемой памяти

return 15

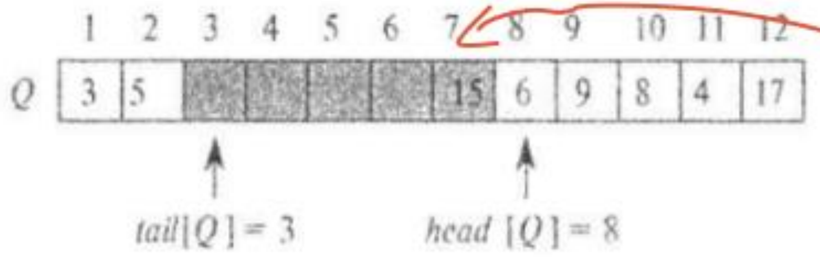


DEQUEUE(Q)

```
x ← Q[head[Q]]
if head[Q] = length[Q]
   then head[Q] ← 1
   else head[Q] ← head[Q] + 1
return x
```

переход ! на начало адресов

Отпустить



не очищаем ячейку, помечаем ка свободную

ПЕРЕПОЛНЕНИЕ/ПУСТО

ПУСТО и ПЕРЕПОЛНЕНИЕ:

if Head[Q] == Tail[Q]

then “overflowing” / «empty»

ПЕРЕПОЛНЕНИЕ/ПУСТО

ПУСТО и ПЕРЕПОЛНЕНИЕ:

if Head[Q] == Tail[Q]

then “overflowing” / «empty»

ПЕРЕПОЛНЕНИЕ/ПУСТО

ПУСТО и ПЕРЕПОЛНЕНИЕ:

if Head[Q] == Tail[Q]

then “overflowing” / «empty»

ОЦЕНКА ОПЕРАЦИЙ С СТРУКТУРОЙ ПО ВРЕМЕНИ

Удаление	Добавление	Поиск
O(1)	O(1)	O(n)

ОЦЕНКА ОПЕРАЦИЙ С СТРУКТУРОЙ ПО ВРЕМЕНИ

Удаление	Добавление	Поиск
O(1)	O(1)	O(n)

Head + 1

OR

Head = 1

+ return

Q[head]

~O(1)

ОЦЕНКА ОПЕРАЦИЙ С СТРУКТУРОЙ ПО ВРЕМЕНИ

Удаление	Добавление	Поиск
O(1)	O(1)	O(n)

Tail + 1

OR

Tail = 1

+

Q[Tail] = x

~O(1)

ОЦЕНКА ОПЕРАЦИЙ С СТРУКТУРОЙ ПО ВРЕМЕНИ

Удаление	Добавление	Поиск
O(1)	O(1)	O(n)

1) Пройти всю очередь от Head до Tail ~ O(n)
2) Через переход от N к 1 ~O(n)

ОЦЕНКА ОПЕРАЦИЙ С СТРУКТУРОЙ ПО ВРЕМЕНИ

Удаление	Добавление	Поиск
O(1)	O(1)	O(n)

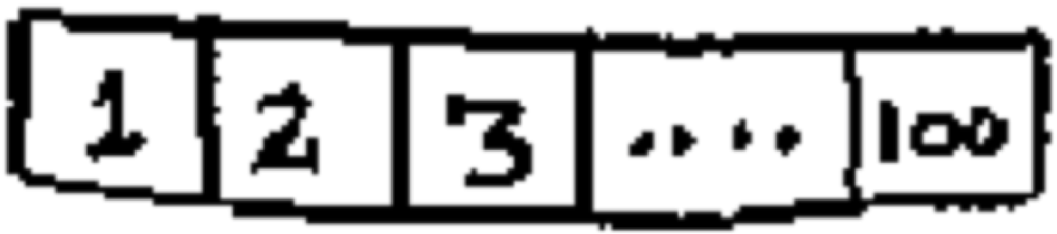
Head + 1
OR
Head = 1
+ return
Q[head]
~O(1)

Tail + 1
OR
Tail = 1
+
Q[Tail] = x
~O(1)

1) Пройти всю очередь от Head до Tail ~ O(n)
2) Через переход от N к 1 ~O(n)

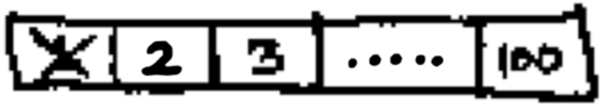
Обычный поиск

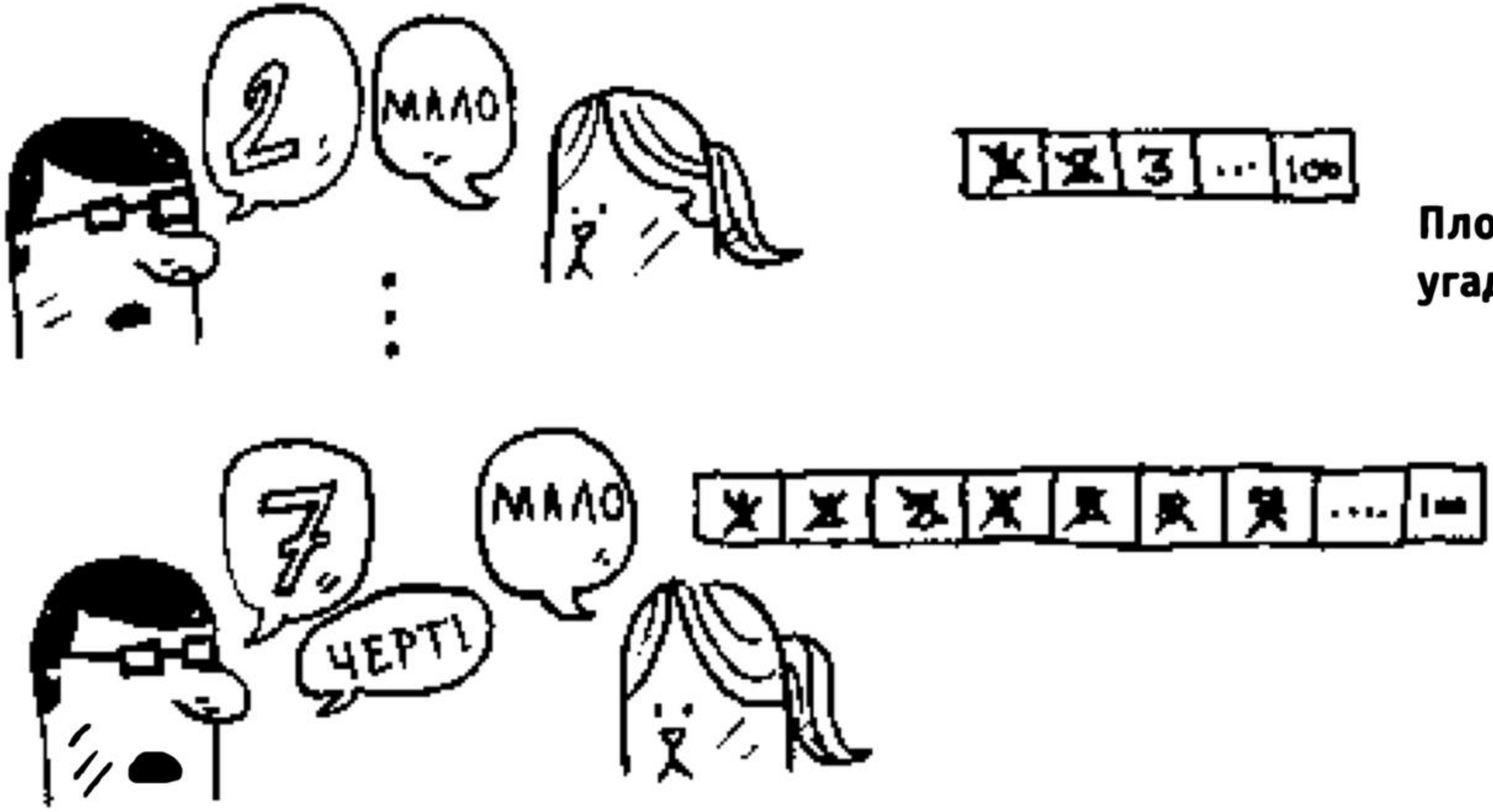
Сыграем в игру: Отгадайте какое число загадано от 1 до 100



Отгадайте число используя как меньше попыток. При каждой попытке будет даваться один из трех вариантов: “мало”, “много”, “угадал”

Предположим, вы начинаете перебирать все варианты подряд: 1, 2, 3, 4 ....
Вот как это будет выглядеть.





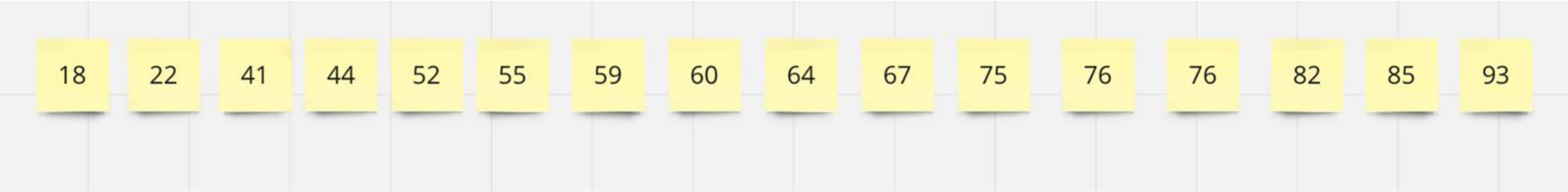
Плохой способ угадать число

Это пример “тупого поиска”. Если загадать число 99, понадобиться 99 попыток! Получается обычный поиск работает за O(n)

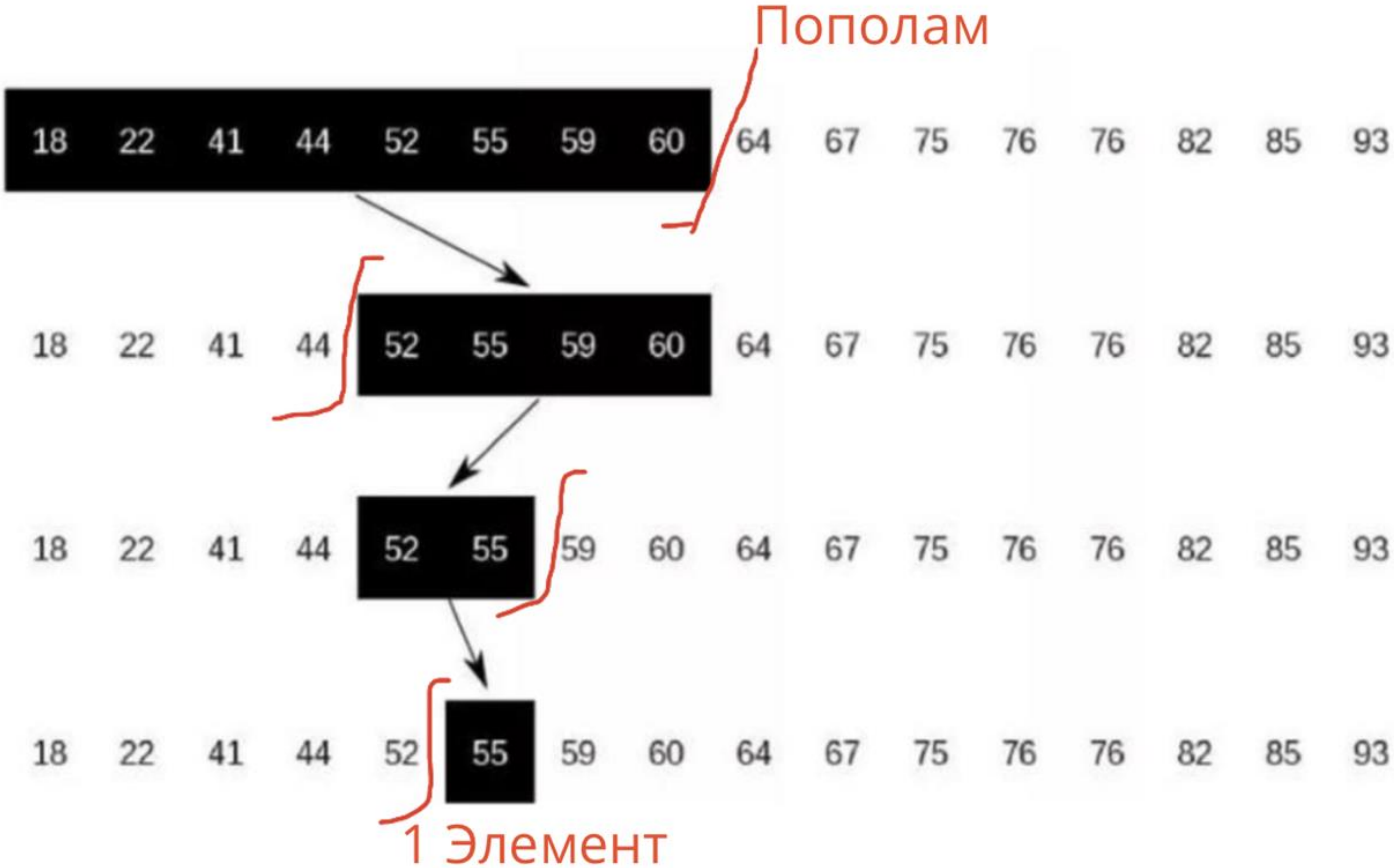
Как сделать эффективнее?

Бинарный поиск

Давайте найдем число 55



Бинарный поиск

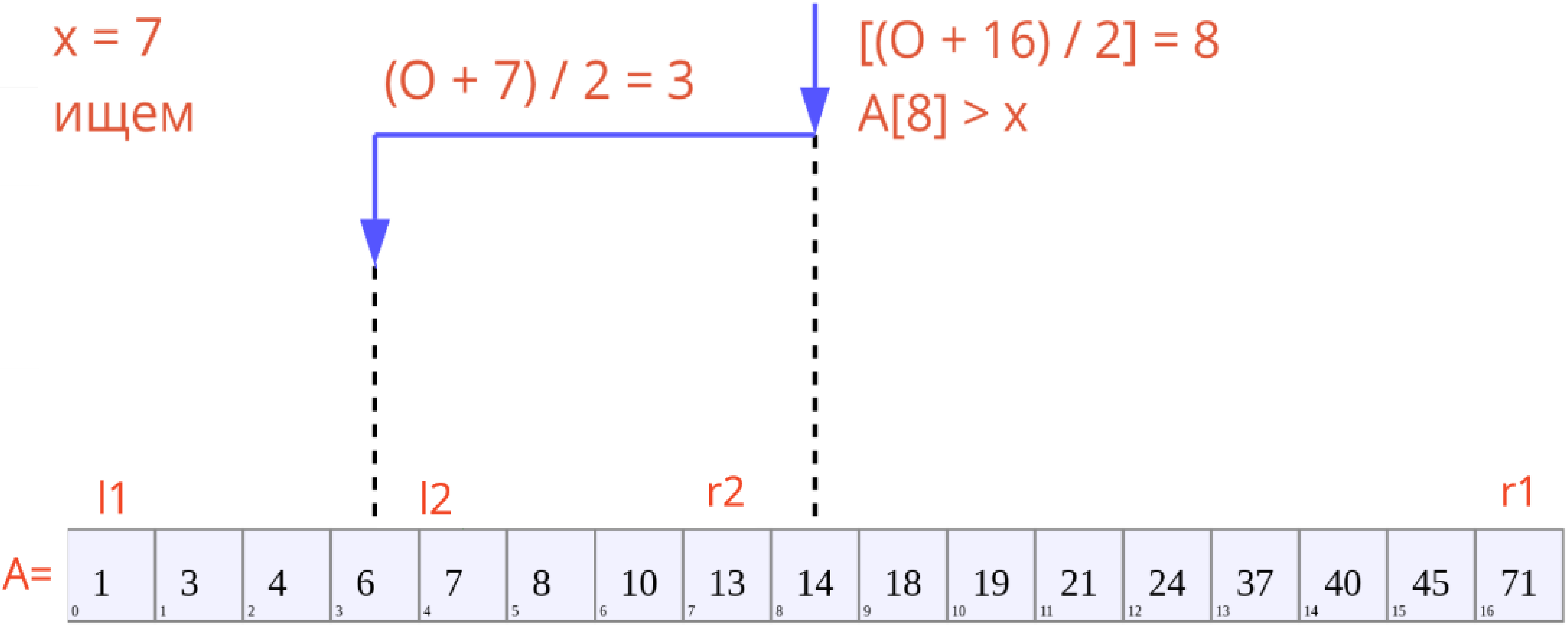


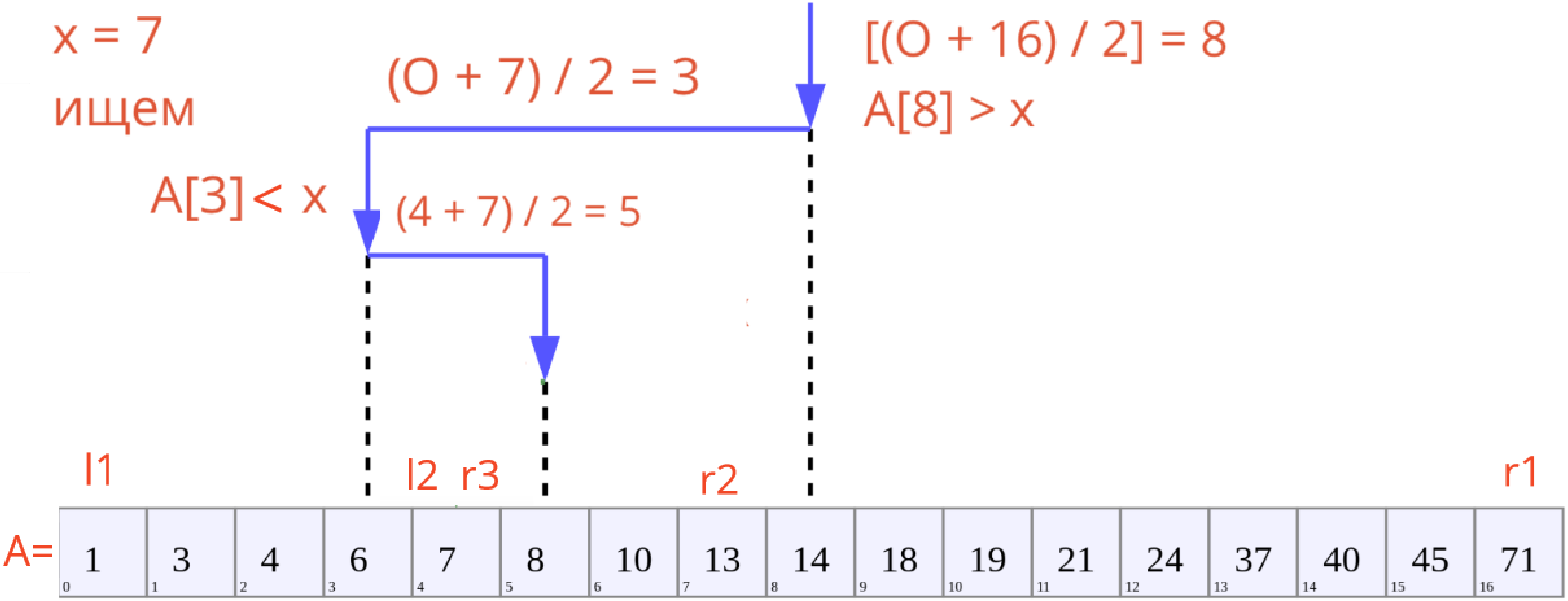
x = 7

ищем

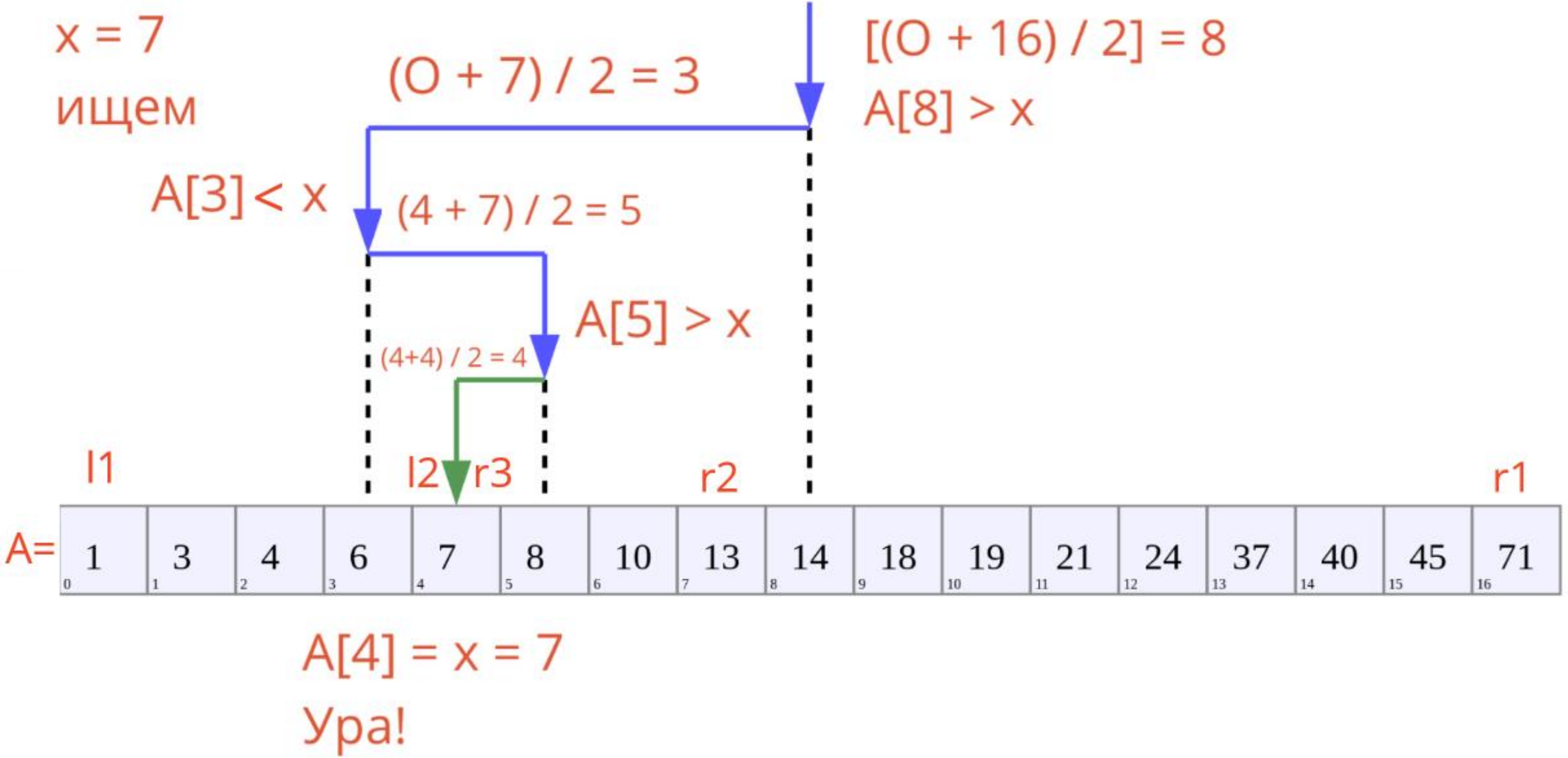
A=

l1																r1
1	3	4	6	7	8	10	13	14	18	19	21	24	37	40	45	71
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16


x = 7
ищем
(0 + 7) / 2 = 3
[(0 + 16) / 2] = 8
A[8] > x
l1
l2
r2
r1
A=
1 3 4 6 7 8 10 13 14 18 19 21 24 37 40 45 71
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16



x = 7
ищем
(O + 7) / 2 = 3
[(O + 16) / 2] = 8
A[8] > x
A[3]< x
(4 + 7) / 2 = 5
l1
l2
r3
r2
r1
A=
1 3 4 6 7 8 10 13 14 18 19 21 24 37 40 45 71
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16


x = 7
ищем
[(0 + 16) / 2] = 8
A[8] > x
(0 + 7) / 2 = 3
A[3] < x
(4 + 7) / 2 = 5
A[5] > x
(4+4) / 2 = 4
l1
l2
r3
r2
r1
A=
1 3 4 6 7 8 10 13 14 18 19 21 24 37 40 45 71
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
A[4] = x = 7
Ура!

Бинарный поиск

// l, r - левая и права границы

// m - середина области поиска

// Запускаем цикл

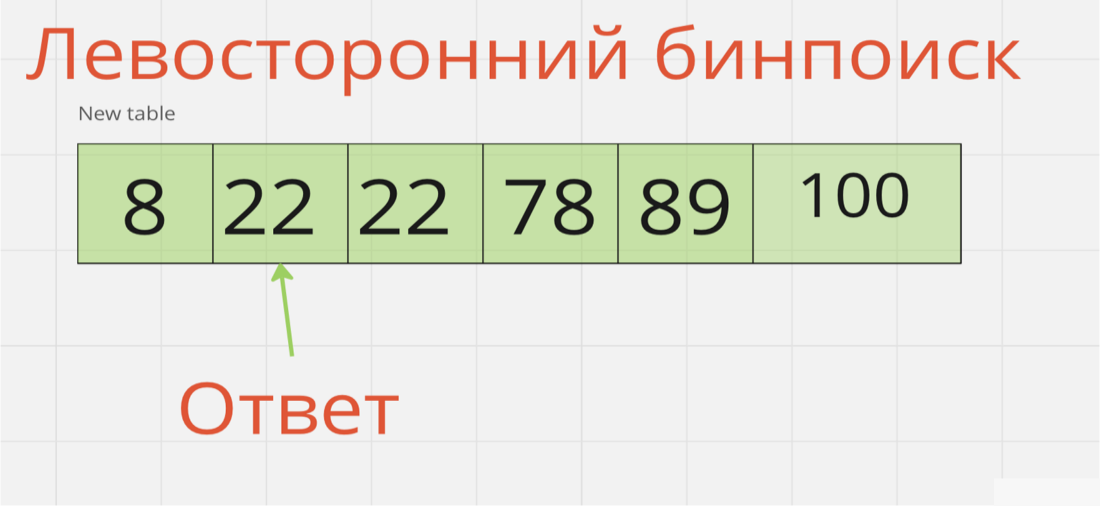
Сужение границ

```
BianarySearch(A, x)
  l = -1
  r = A.length
  while r > l+1
      m = [(l + r) / 2]
      if A[m] < x
              then l = m
              else r = m

  if r < A.length и A[r] == x
     then return r
     else return -1
```

~O(log(n))

Левый и правый бинпоиск




```python
#левый бинпоиск, который ищет левую границу
while right - left > 1:
    middle = (right + left) // 2
    if a[middle] < x:
        left = middle
    else:
        right = middle
```

Правостостороний бинпоиск

New table

8	22	22	22	78	89

Ответ

```
#правый бинпоиск, который ищет правый границу
while right_1 - left_1 > 1:
    middle = (right_1 + left_1) // 2
    if a[middle] <= x:
        left_1 = middle
    else:
        right_1 = middle
```

Циклический дек на динамическом массиве

Ключевые поля:

- $\mathtt{n}$ — размер массива,
- $\mathtt{d}[0 \ldots \mathtt{n} - 1]$ — массив, в котором хранится дек,
- $\mathtt{newDeque}[0 \ldots \mathtt{newSize}]$ — временный массив, где хранятся элементы после перекопирования,
- $\mathtt{head}$ — индекс головы дека,
- $\mathtt{tail}$ — индекс хвоста.

Дек состоит из элементов $\mathtt{d}[\mathtt{head} \ldots \mathtt{tail} - 1]$ или $\mathtt{d}[0 \ldots \mathtt{tail} - 1]$ и $\mathtt{d}[\mathtt{head} \ldots \mathtt{n} - 1]$. Если реализовывать дек на динамическом массиве, то мы можем избежать ошибки переполнения. При выполнении операций $\mathtt{pushBack}$ и $\mathtt{pushFront}$ происходит проверка на переполнение и, если нужно, выделяется большее количество памяти под массив. Также происходит проверка на избыточность памяти, выделенной под дек при выполнении операций $\mathtt{popBack}$ и $\mathtt{popFront}$. Если памяти под дек выделено в четыре раза больше размера дека, то массив сокращается в два раза. Для удобства выделим в отдельную функцию $\mathtt{size}$ получение текущего размера дека.

```
int size()
  if tail > head
    return n − head + tail
  else
    return tail − head
```

```
function pushBack(x : T):
  if (head == (tail + 1) % n)
    T newDeque[n * 2]
    for i = 0 to n − 2
      newDeque[i] = d[head]
      head = (head + 1) % n
    d = newDeque
    head = 0
    tail = n − 1
    n *= 2
  d[tail] = x
  tail = (tail + 1) % n
```

```
T popBack():
  if (empty())
    return error "underflow"
  if (size() < n / 4)
    T newDeque[n / 2]
    int dequeSize = size()
    for i = 0 to dequeSize − 1
      newDeque[i] = d[head]
      head = (head + 1) % n
    d = newDeque
    head = 0
    tail = dequeSize
    n /= 2
  tail = (tail − 1 + n) % n
  return d[tail]
```

```
function pushFront(x : T):
  if (head == (tail + 1) % n)
    T newDeque[n ∗ 2]
    for i = 0 to n − 2
      newDeque[i] = d[head]
      head = (head + 1) % n
    d = newDeque
    head = 0
    tail = n − 1
    n ∗= 2
  head = (head − 1 + n) % n
  d[head] = x
```

```
T popFront():
  if (empty())
    return error "underflow"
  if (size() < n / 4)
    T newDeque[n / 2]
    int dequeSize = size()
    for i = 0 to dequeSize − 1
      newDeque[i] = d[head]
      head = (head + 1) % n
    d = newDeque
    head = 0
    tail = dequeSize
    n /= 2
  T ret = d[head]
  head = (head + 1) % n
  return ret
```

АиСД
Лекция 8

- **Логическая структура пирамида (куча)**

- **Пирамидальная сортировка**

Сортировка выбором

В чём идея сортировки выбором?

Сортировка выбором

1) В не отсортированном подмассиве ищется локальный максимум (минимум)

Сортировка выбором

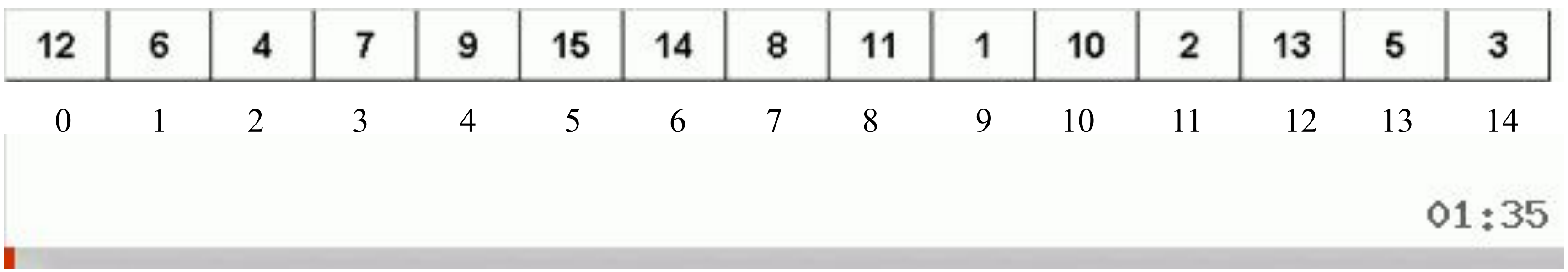
1) В не отсортированном подмассиве ищется локальный максимум (минимум)
2) Найденный максимум (минимум) меняется местами с последним (первым) элементом в подмассиве.

Сортировка выбором

1) В не отсортированном подмассиве ищется локальный минимум
2) Найденный максимум (минимум) меняется местами с последним (первым) элементом в подмассиве.
3) Если в массиве остались неотсортированные подмассивы — смотри пункт 1.

Сортировка выбором

Каждый раз мы находим в массиве наименьший элемент за n, n - 1, n - 2 и так далее => асимптотика O(n^2)



ПИРАМИДАЛЬНАЯ СОРТИРОВКА

Улучшение сортировки выбором -

Пирамидальная сортировка

Пирамидальная сортировка

- **Пирамидальная сортировка, представляющая собой улучшение метода прямого выбора, была предложена Джоном Уильямсом в 1964, а затем улучшена Робертом Флойдом**.

Пирамидальная сортировка

- **Пирамидальная сортировка**, представляющая собой **улучшение метода прямого выбора**, была предложена Джоном Уильямсом в 1964, а затем улучшена Робертом Флойдом.

- **В сортировке прямым выбором наименьший элемент на рассматриваемом участке массива фактически отыскивается путём полного просмотра участка, то есть также как в процедуре линейного поиска.**

Пирамидальная сортировка

- **Пирамидальная сортировка, представляющая собой улучшение метода прямого выбора, была предложена Джоном Уильямсом в 1964, а затем улучшена Робертом Флойдом**.

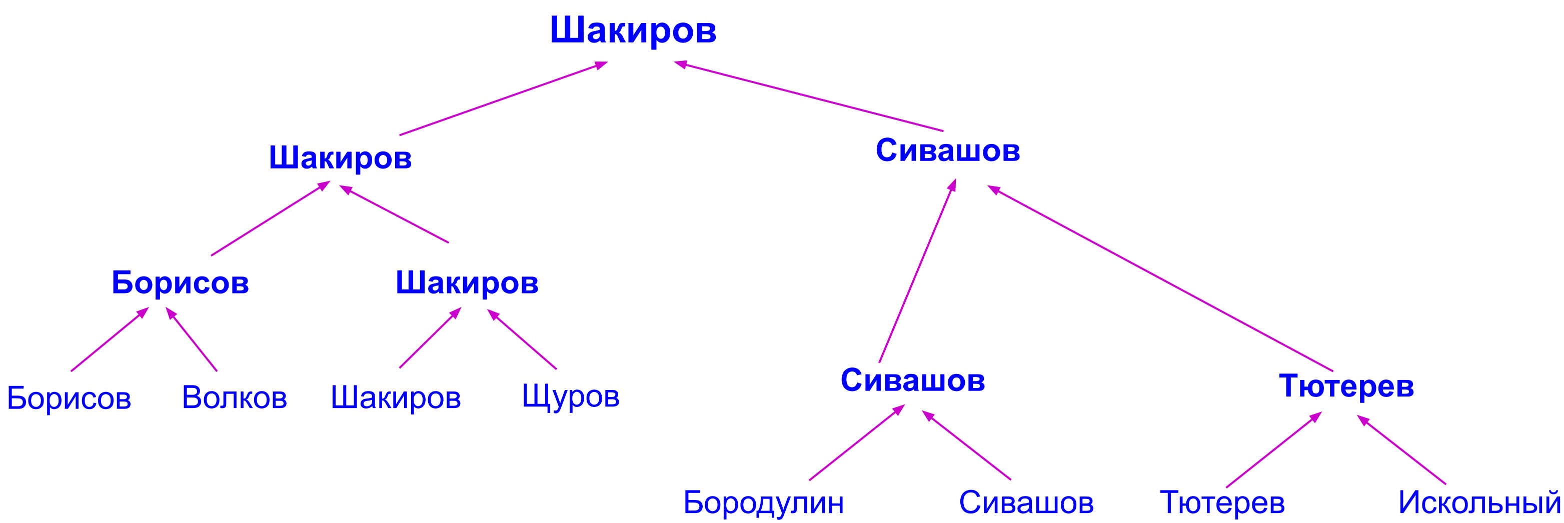
- **В сортировке прямым выбором наименьший элемент на рассматриваемом участке массива фактически отыскивается путём полного просмотра участка, то есть также как в процедуре линейного поиска.**

- **Идея улучшения этой сортировки состоит в *переходе от линейного выбора* со сложностью O(N) *к выбору в дереве*, с помощью которого можно сохранить и использовать гораздо больше информации о процессе выбора и уменьшить сложность до $O(\log_2 N)$.**

Пирамидальная сортировка

Допустим проводятся соревнования по какому-либо виду спорта (шахматы, теннис и т.д.) среди восьми участников. Можно провести чемпионат, когда каждый из участников встречается с каждым, но это требует много времени и затрат.

Пирамидальная сортировка



Рассмотрим сначала бытовой аналог пирамидальной сортировки

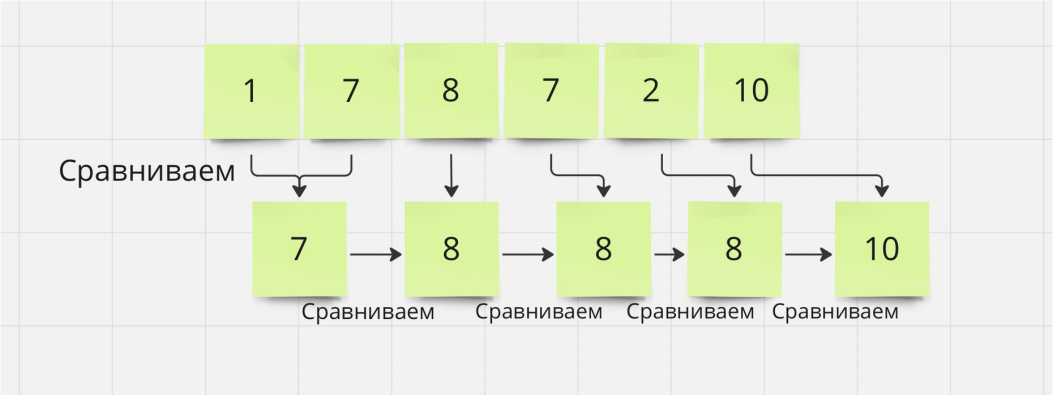
Есть более простой и быстрый способ организации соревнований - <u>кубковая система.</u> Участников по какому-либо принципу разделяют на пары.

Победитель каждой пары выходит в следующий круг, а проигравший выбывает из соревнования.

Пирамидальная сортировка

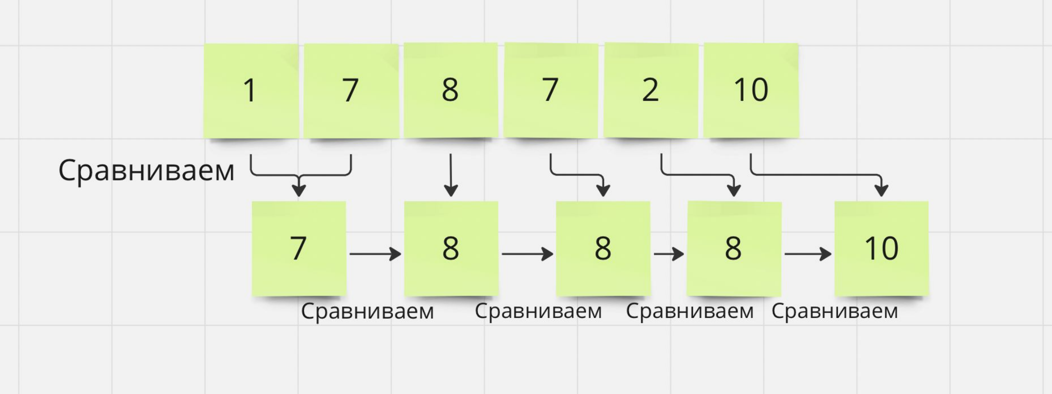
Сравниваем попарно элементы при поиске минимального. Потом снова попарно, так вместо n операций мы получаем log(n)

Два способа сравнения

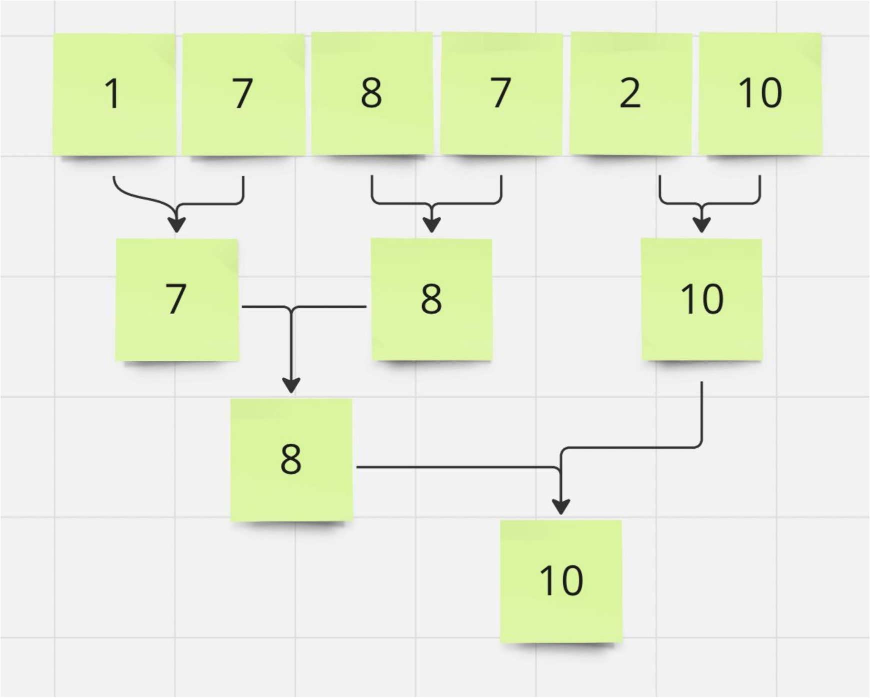


O(n)

Два способа сравнения



O(n)



O(log(n))

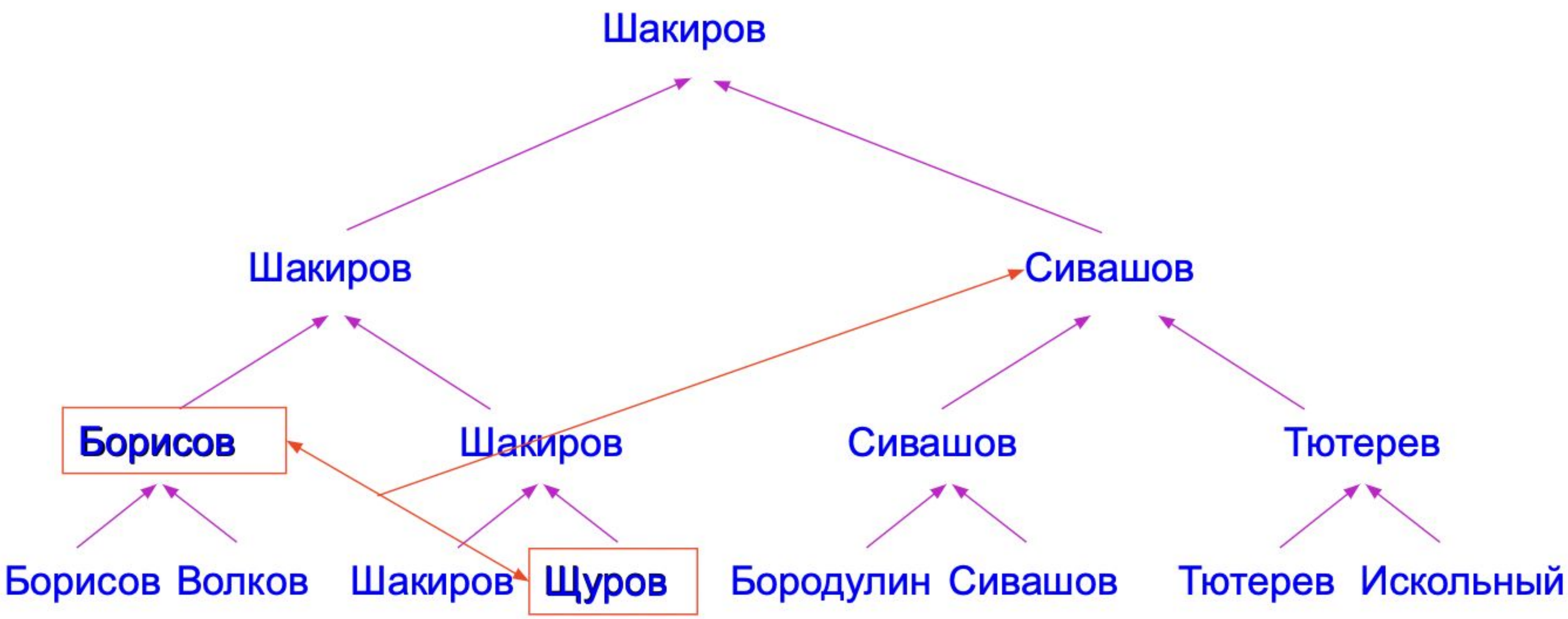
ПИРАМИДАЛЬНАЯ СОРТИРОВКА

Таким образом очень быстро в три этапа определяется сильнейший игрок.

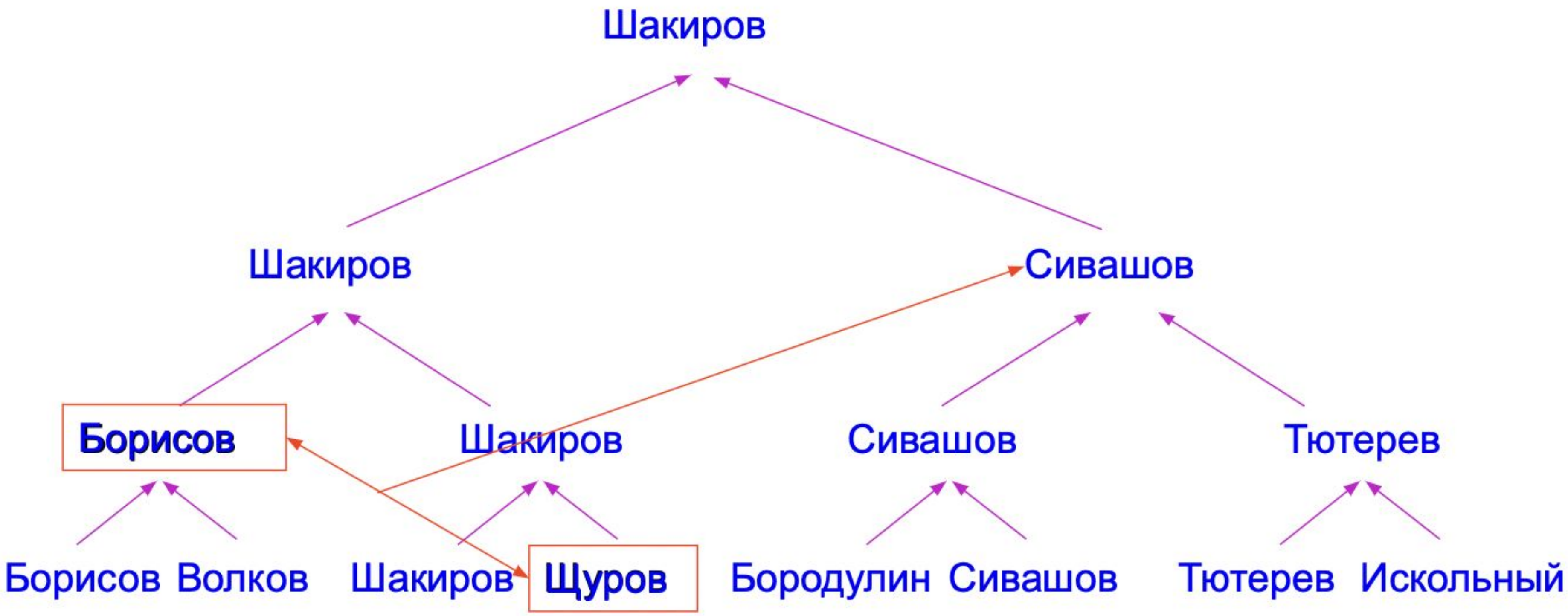
ПИРАМИДАЛЬНАЯ СОРТИРОВКА

Однако, кубковая система с выбыванием имеет недостаток: сложно определить второго, третьего и т.д. по силе игрока, в то время как чемпионат распределяет всех по своим местам.

Пирамидальная сортировка

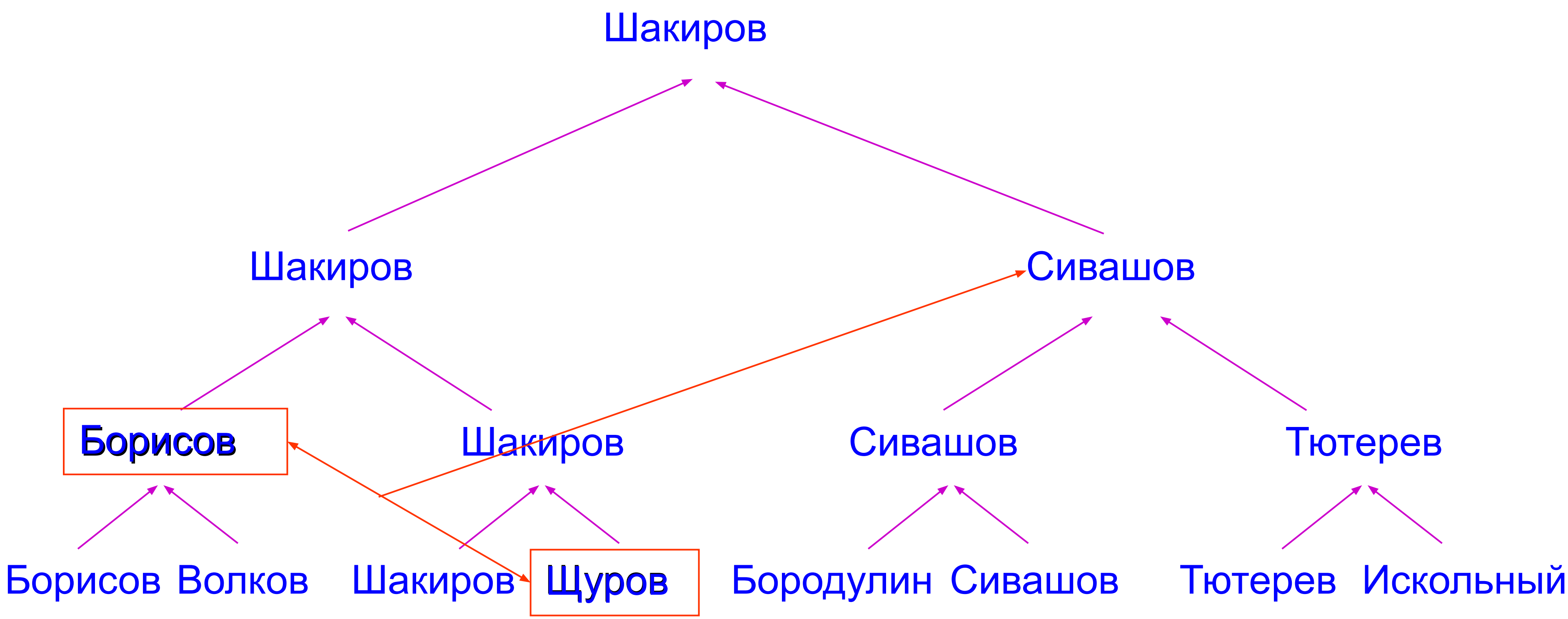


Пирамидальная сортировка



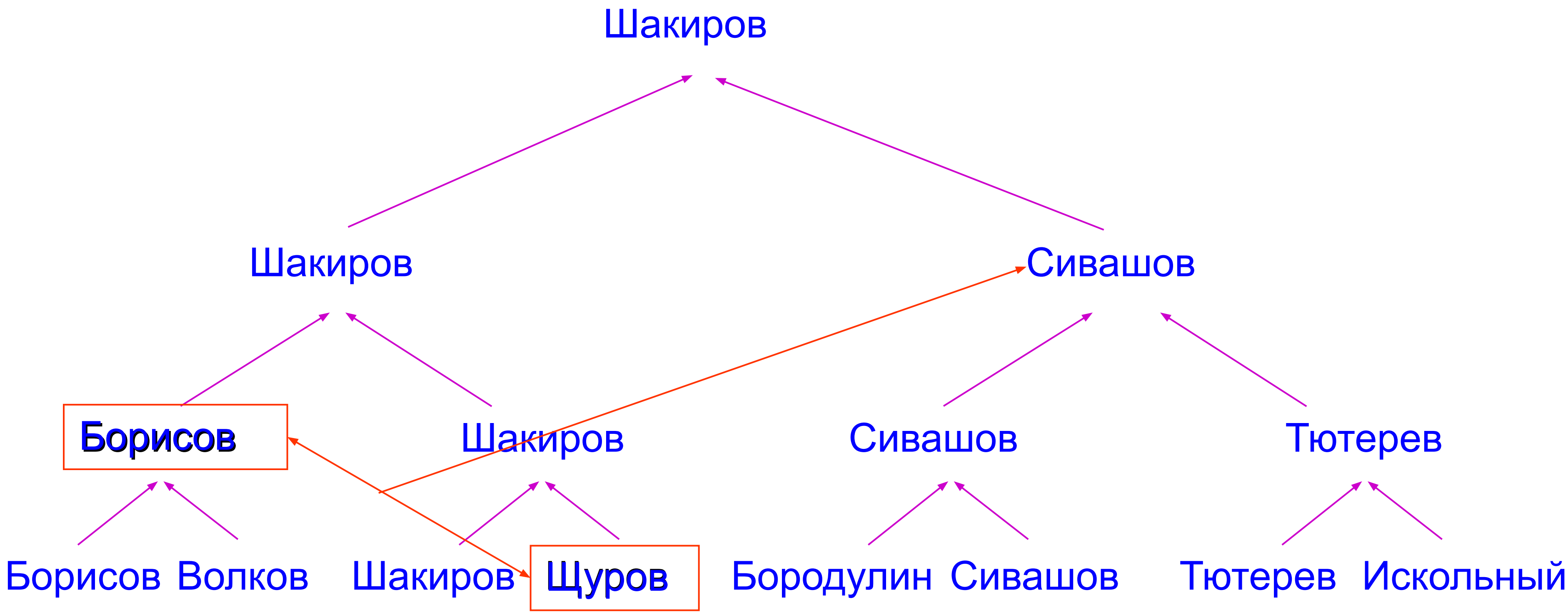
Сортировка по возрастанию = сортировка спортсменов по силе

Пирамидальная сортировка



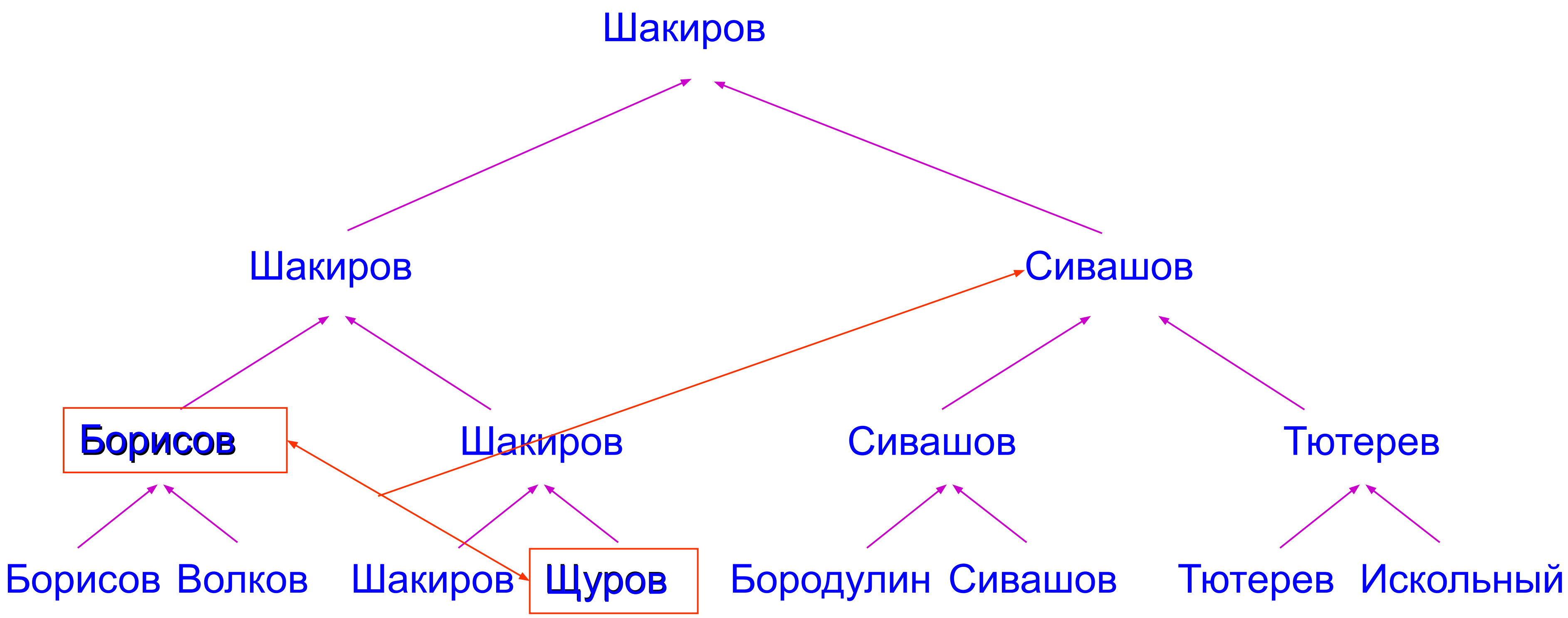
Дело в том, что Сивашов, проигравший в финале Шакирову, совсем не обязательно второй по силам участник соревнований.

Пирамидальная сортировка



Сильнее Сивашова может быть Борисов, сильнее может быть и Щуров, которые проиграли на ранних этапах Шакирову.

Пирамидальная сортировка



Чтобы выявить второго по силе нужно пройти по дереву соревнования вниз, выбрать соперников Шакирова, организовать между ними матч, а затем победитель сыграет с Сивашовым.

Пирамидальная сортировка

Когда мы нашли минимальный элемент при "сортировке выбором" нам потребуется снова искать минимальный элемент.

Пирамидальная сортировка

Когда мы нашли минимальный элемент при "сортировке выбором" нам потребуется снова искать минимальный элемент.

Из оставшихся участников каждый раз выбираем **сильнейшего** по силе.

Пирамидальная сортировка

Когда мы нашли минимальный элемент при "сортировке выбором" нам потребуется снова искать минимальный элемент.

Из оставшихся участников каждый раз выбираем **сильнейшего** по силе.

Если мы каждый раз после выбора текущего лидера будем перестраивать дерево, то мы сможем выбирать текущего лидера за **O(1)**!

Пирамидальная сортировка

Сколько операций необходимо выполнить, чтобы сохранить свойства "кубкового дерева" после удаления очередного лидера?

Пирамидальная сортировка

Видно, что для выявления второго участника потребуется организовать всего два дополнительных матча.

Это оказалось возможным за счёт того, что в структуре дерева существует дополнительная информация о выполненных ранее сравнениях и её использование приведет к уменьшению количества сравнений для выбора второго участника

Пирамидальная сортировка

Однако построение подобного дерева из массива требует дополнительных затрат памяти,

по крайней мере, за счет дублирования одних и тех же элементов на разных уровнях дерева:

вместо хранения N элементов нужно хранить 2N - 1 элемент, что фактически эквивалентно использованию вспомогательного массива

Пирамидальная сортировка

В основе предложенной Р. Флойдом эффективной реализации сортировки выбором из дерева лежит специальная разновидность бинарного дерева, которую принято называть ***пирамидальным деревом*** или просто ***пирамидой***. Иногда такое дерево называют ***деревом сортировки***.

Пирамидальная сортировка

В основе предложенной Р. Флойдом эффективной реализации сортировки выбором из дерева лежит специальная разновидность бинарного дерева, которую принято называть ***пирамидальным деревом*** или просто ***пирамидой***. Иногда такое дерево называют ***деревом сортировки***.

Пирамидальным называется ***полное бинарное*** **дерево**, у которого ключ корня любого поддерева ***не меньше***, чем ключи двух его дочерних вершин.

Пирамидальная сортировка

В основе предложенной Р. Флойдом эффективной реализации сортировки выбором из дерева лежит специальная разновидность бинарного дерева, которую принято называть ***пирамидальным деревом*** или просто ***пирамидой***. Иногда такое дерево называют ***деревом сортировки***.

Пирамидальным называется ***полное бинарное*** **дерево**, у которого ключ корня любого поддерева ***не меньше***, чем ключи двух его дочерних вершин.

В отличие ***от дерева поиска*** ключи ***дочерних*** вершин пирамидального дерева считаются ***неупорядоченными***: ключ левой дочерней вершины может быть как больше, так и меньше ключа правой дочерней вершины

Пирамидальное дерево

Пирамида (Heap) – это бинарное дерево высоты *k*, удовлетворяющая следующим требованиям:

Пирамидальное дерево

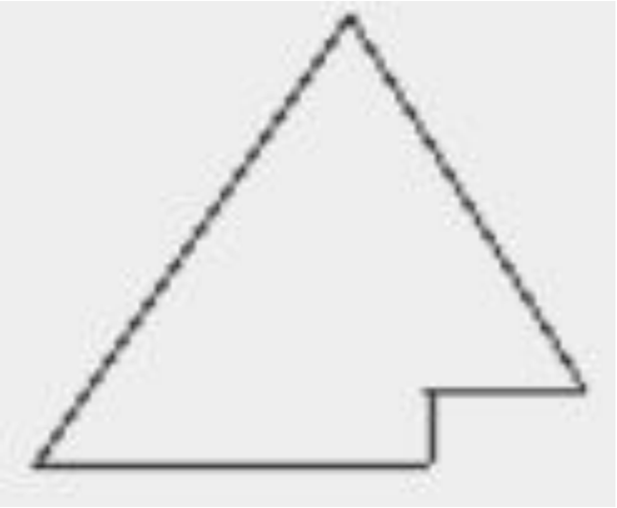
Пирамида (Heap) – это бинарное дерево высоты *k*, удовлетворяющая следующим требованиям:

- **узлами дерева являются элементы массива;**

Пирамидальное дерево

Пирамида (Heap) – это бинарное дерево высоты *k*, удовлетворяющая следующим требованиям:

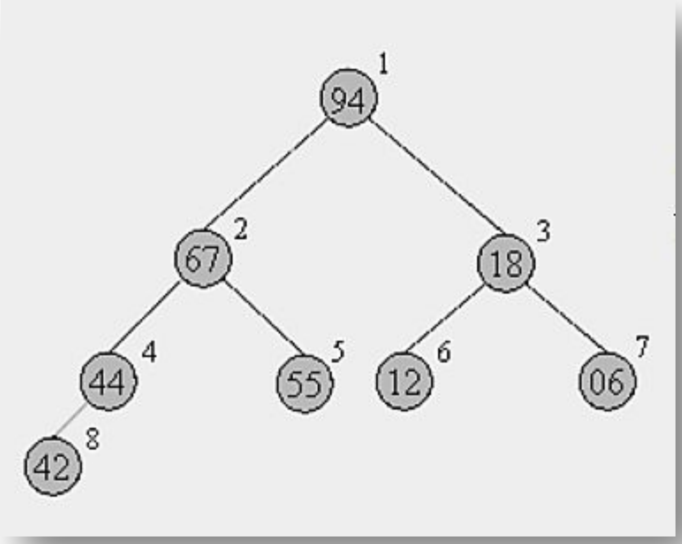
- **дерево является сбалансированным, т.е. все листья имеют глубину *k* или *k – 1*, при этом уровень *k – 1* полностью заполнен, а уровень *k* заполнен слева направо, т.е. форма пирамиды имеет приблизительно такой вид:**



Пирамидальное дерево

Пирамида (Heap) – это бинарное дерево высоты *k*, удовлетворяющая следующим требованиям:

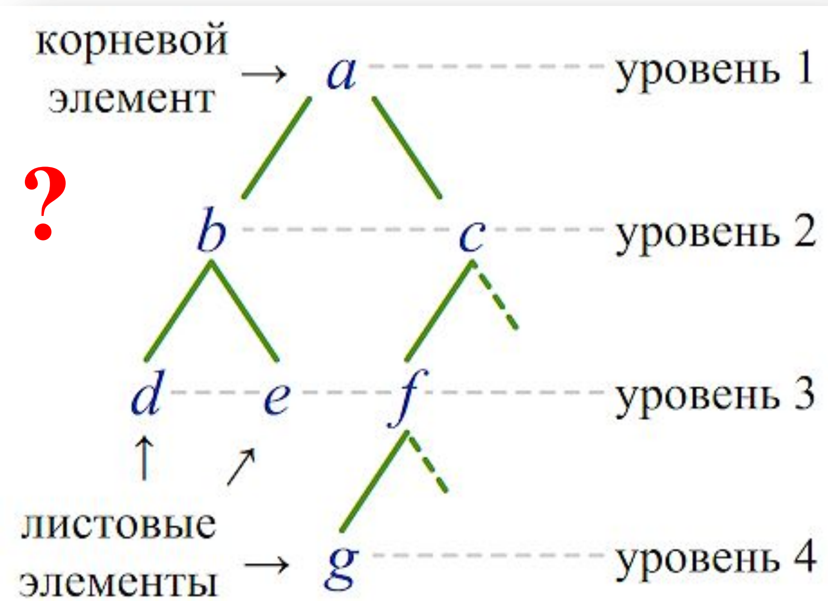
- **выполняется "свойство пирамиды":**
  - **каждый элемент меньше либо равен родителю,**
  - **корнем дерева является максимальный элемент массива**



Пример пирамиды, имеющей глубину 4

Пирамидальное дерево

Для представления пирамиды не требуется создавать специальную структуру данных – ее можно хранить прямо в сортируемом массиве



?

Соответствие между геометрической структурой пирамиды как дерева и массивом:

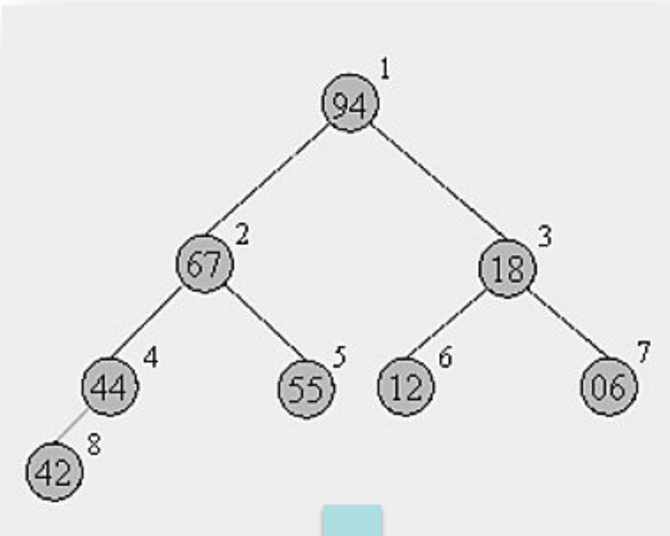
► **в *a[0]* хранится корень дерева**

► **левый и правый сыновья элемента *a[i]* хранятся в *a[2i+1]* и *a[2i+2]* соответственно**

Здесь – массив из *n* элементов, нумерация от *0* до *n-1*

Пирамидальное дерево

Для представления пирамиды не требуется создавать специальную структуру данных – ее можно хранить прямо в сортируемом массиве



Массив

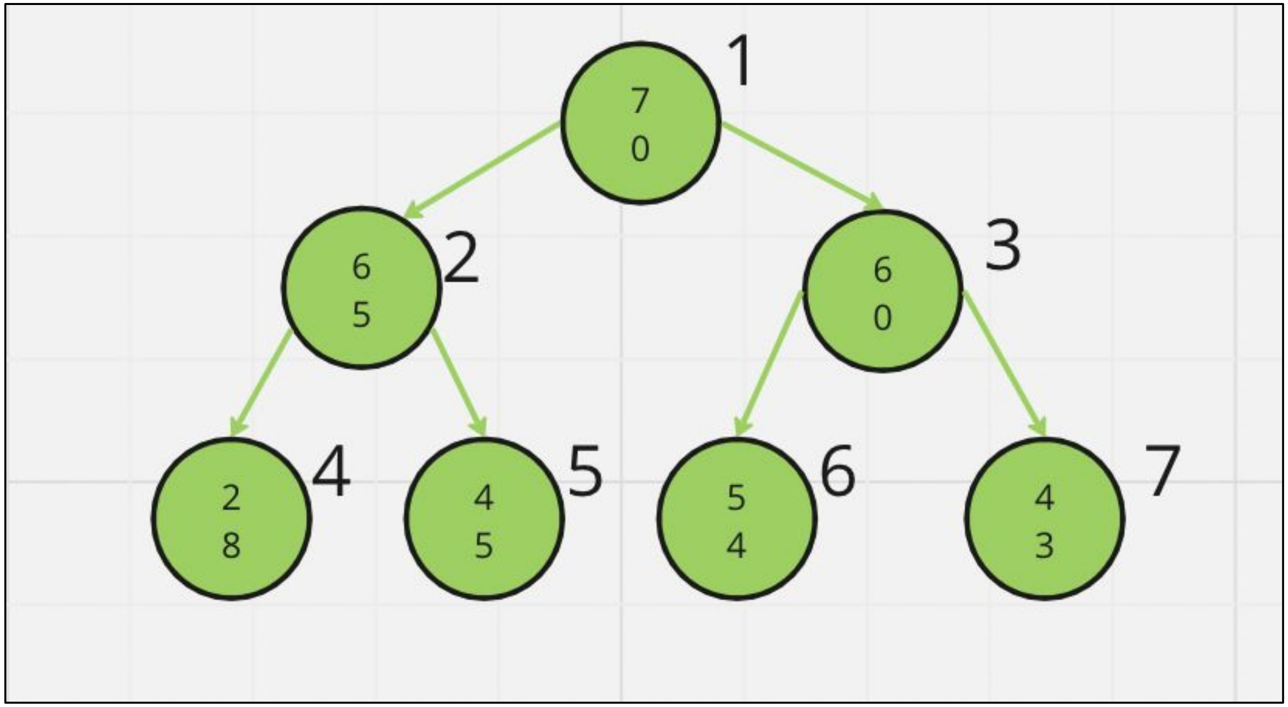
94, 67, 18, 44, 55, 12, 06, 42

Слева - направо, сверху - вниз

Характеристическое свойство для массива, хранящего в себе пирамиду:

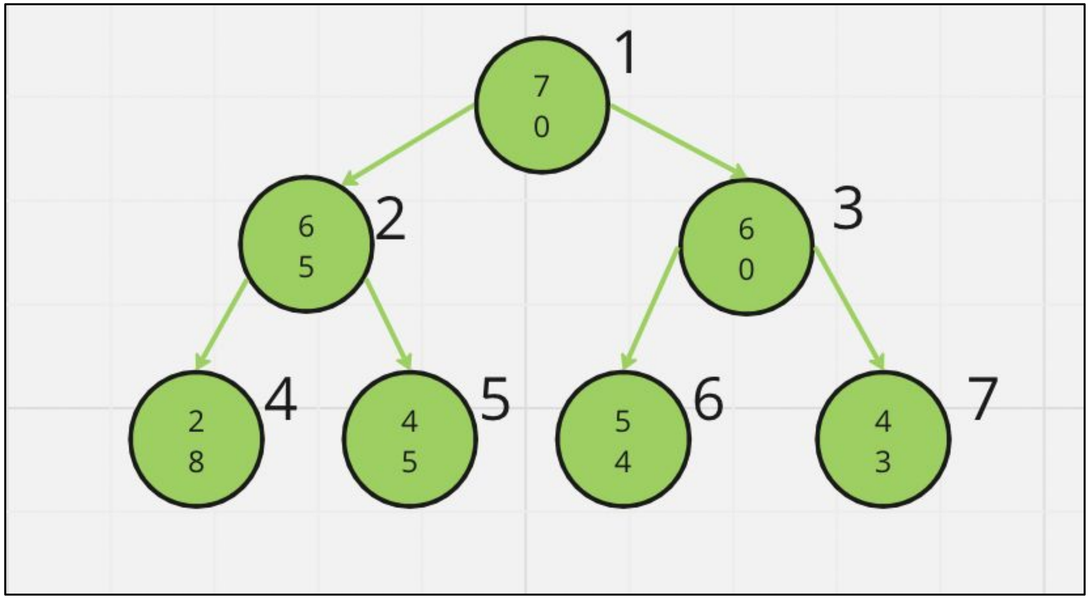
- $a[i] \geq a[2i+1]$
- $a[i] \geq a[2i+2]$

Пирамидальное дерево



Ключ корня 70 больше, чем ключи двух его дочерних узлов 65 и 60. Ключ левого поддерева 65 больше ключей 28 и 45, также как и ключ правого поддерева 60 больше, чем ключи 54 и 43 их дочерних узлов. Изображённое дерево *является пирамидой*

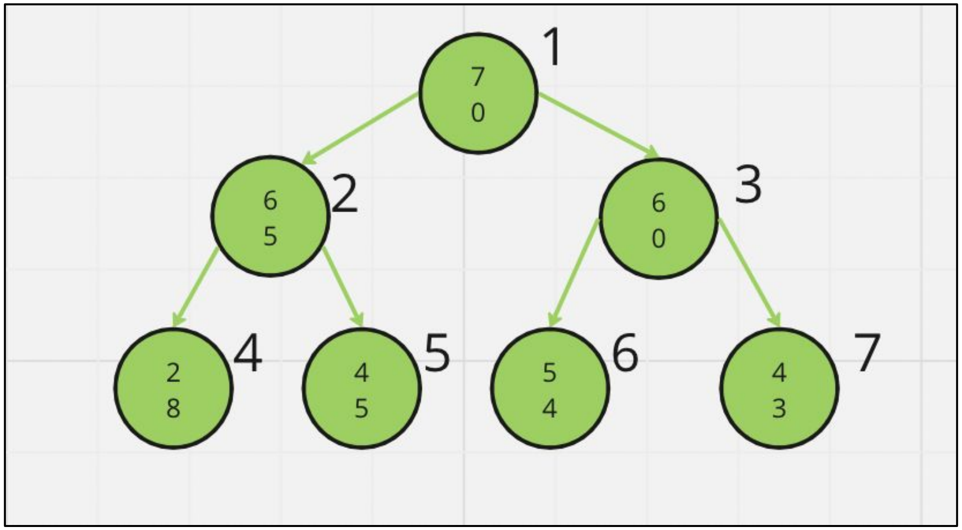
Пирамидальное дерево



Для сортировки в порядке ***убывания*** используется другой вариант пирамидального дерева, в котором ключ корня любого поддерева ***не больше***, чем ключи двух его дочерних вершин

Пирамидальное дерево

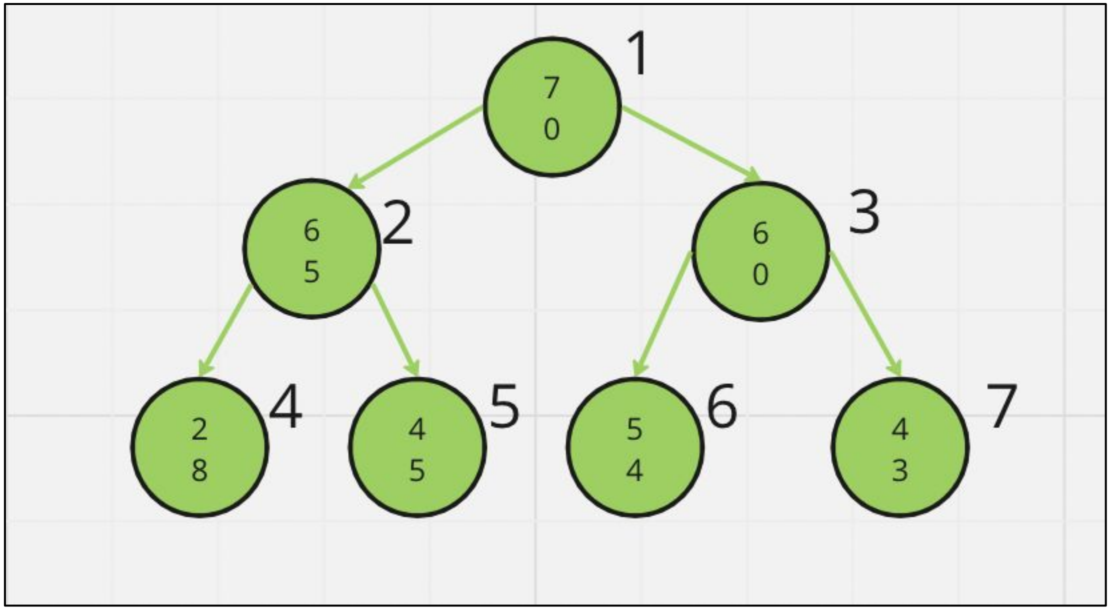
Включение новых узлов в пирамидальное дерево осуществляется *строго слева направо* и только *после полного заполнения* предшествующего уровня



Пирамидальное дерево

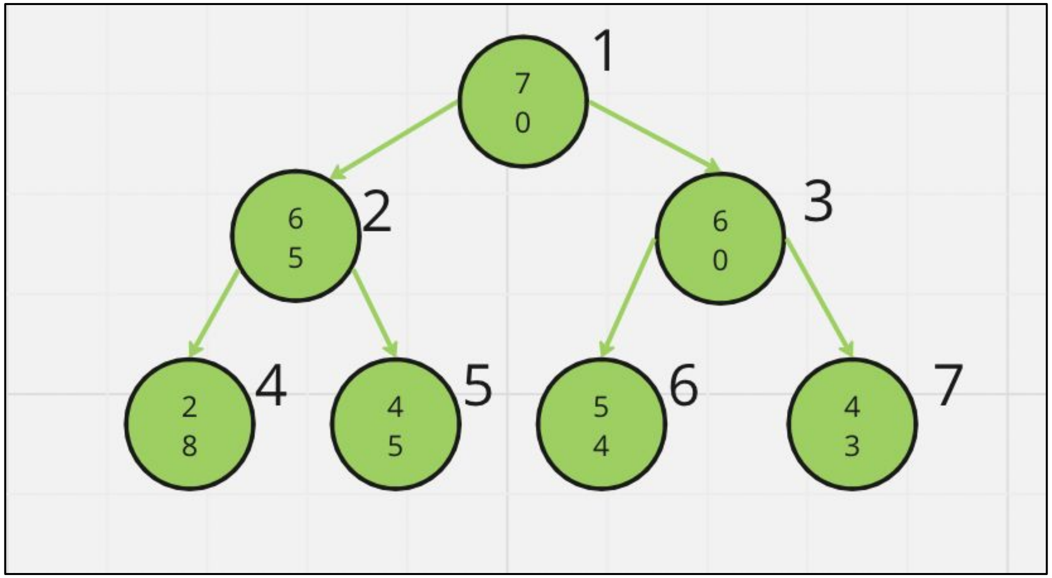
Заполненным **считается уровень дерева, содержащий максимально возможное количество узлов**

Частично незаполненным может быть только *самый нижний* уровень дерева



Пирамидальное дерево

```
function insert(key : int):
    a.heapSize = a.heapSize + 1
    a[a.heapSize - 1] = key
    siftUp(a.heapSize - 1)
```



Пирамидальное дерево

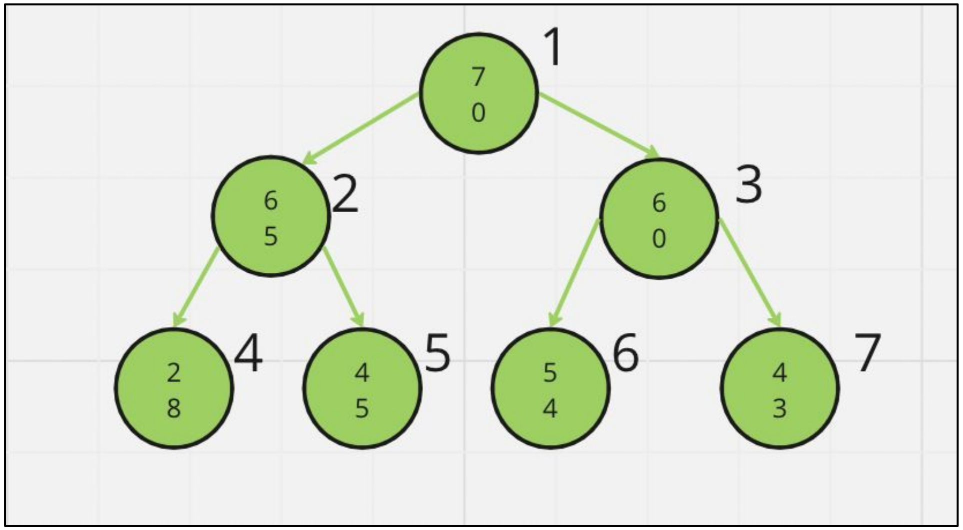
Пирамидальное дерево
Операция siftUp

Если элемент больше своего отца, условие 1 соблюдено для всего дерева, и больше ничего делать не нужно. Иначе, мы меняем местами его с отцом. После чего выполняем **siftUp** для этого отца. Иными словами, слишком маленький элемент всплывает наверх. Процедура выполняется за время **O(logn)**.

```
function siftUp(i : int):
    while a[i] < a[(i - 1) / 2]     // i  0 — мы в корне
        swap(a[i], a[(i - 1) / 2])
        i = (i - 1) / 2
```

Пирамидальное дерево

Узлы пирамидального дерева принято нумеровать в соответствии с порядком поступления новых узлов в дерево. Эти номера принято называть *индексами узлов пирамидального дерева*



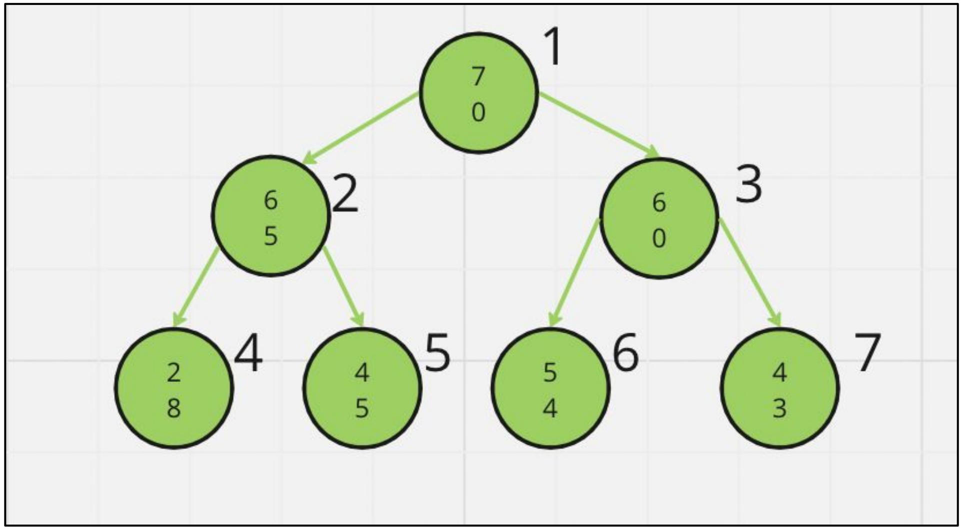
В этой системе нумерации корневой узел всегда имеет индекс 1.

Два его дочерних узла имеют индексы 2 и 3, а, например, дочерние для узла с индексом 3 имеют индексы 6 и 7 и т.д.

Пирамидальное дерево

Узлы пирамидального дерева принято нумеровать в соответствии с порядком поступления новых узлов в дерево. Эти номера принято называть *индексами узлов пирамидального дерева*

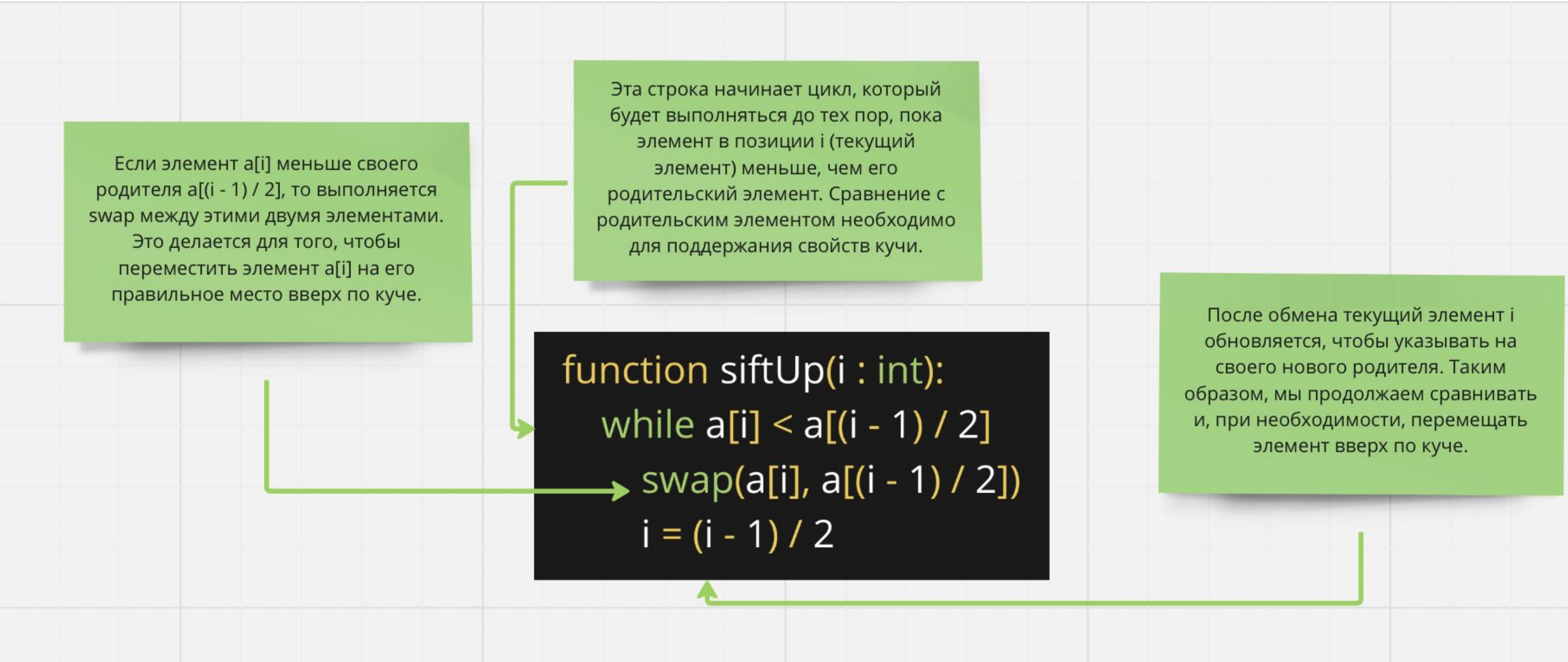
```
function siftUp(i : int):
   while a[i] < a[(i - 1) / 2]
      swap(a[i], a[(i - 1) / 2])
      i = (i - 1) / 2
```



В этой системе нумерации корневой узел всегда имеет индекс 1.

Два его дочерних узла имеют индексы 2 и 3, а, например, дочерние для узла с индексом 3 имеют индексы 6 и 7 и т.д.

Пирамидальное дерево



Асимптотика

Shift-up

Работа процедуры: если элемент больше своего отца, условие 1 соблюдено для всего дерева, и больше ничего делать не нужно. Иначе, мы меняем местами его с отцом. После чего выполняем siftUp для этого отца. Иными словами, слишком маленький элемент всплывает наверх. Процедура выполняется за время O(logn)

Асимптотика

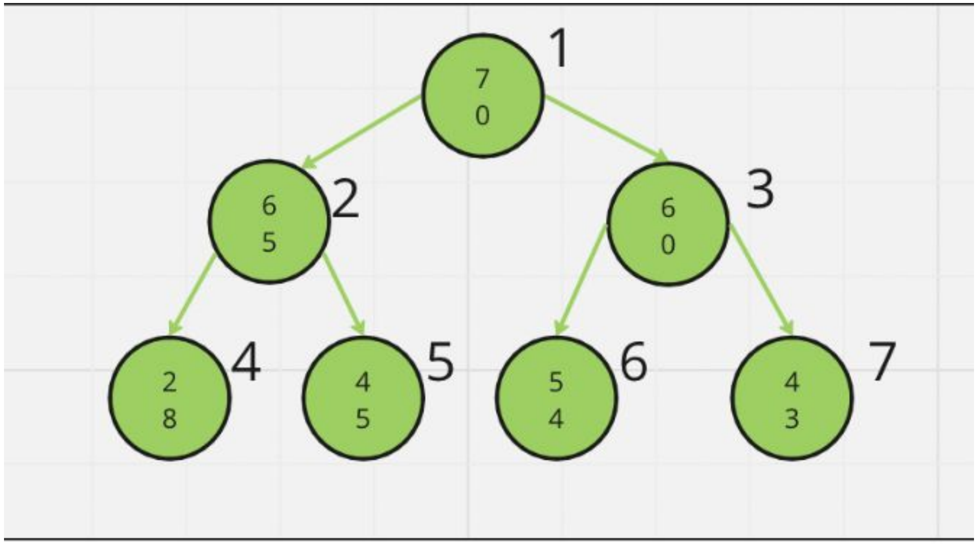
Insert

Выполняет добавление элемента в кучу за время $O(\log(n))$.

Так как сначала мы добавляем элемент в конец, а затем восстанавливаем свойства упорядоченности с помощью процедуры siftUp.

Пирамидальное дерево

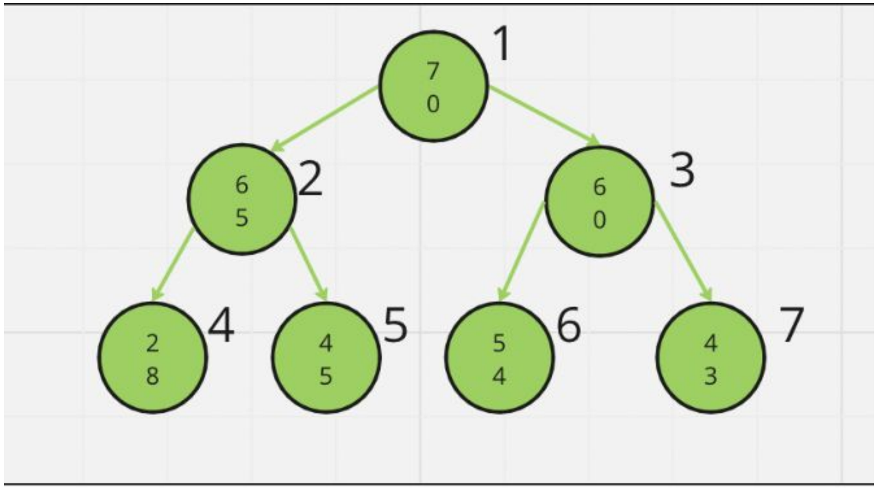
```
int extractMax():
    int max = a[0]
    a[0] = a[a.heapSize - 1]
    a.heapSize = a.heapSize - 1
    siftDown(0)
    return max
```



В общем случае для узла с индексом k дочерние узлы (если они есть) всегда имеют индексы 2k и 2k+1. Для узла с индексом m (m≠1) родительский всегда имеет индекс int(m/2).

Пирамидальное дерево

```
int extractMax():
    int max = a[0]
    a[0] = a[a.heapSize - 1]
    a.heapSize = a.heapSize - 1
    siftDown(0)
    return max
```



В общем случае для узла с индексом k дочерние узлы (если они есть) всегда имеют индексы 2k и 2k+1. Для узла с индексом m (m ≠ 1) родительский всегда имеет индекс int(m / 2).

Эти простые соотношения между индексами родительского и дочерних узлов обеспечивают высокую эффективность адресной арифметики на машинном уровне

С использованием индексов свойство ключей узлов пирамидального дерева можно записать в виде

$$(x[k] \geq x[2k]) \wedge (x[k] \geq x[2k+1]), \quad k=1, 2, 3,\ldots, \mathrm{int}(N/2)$$

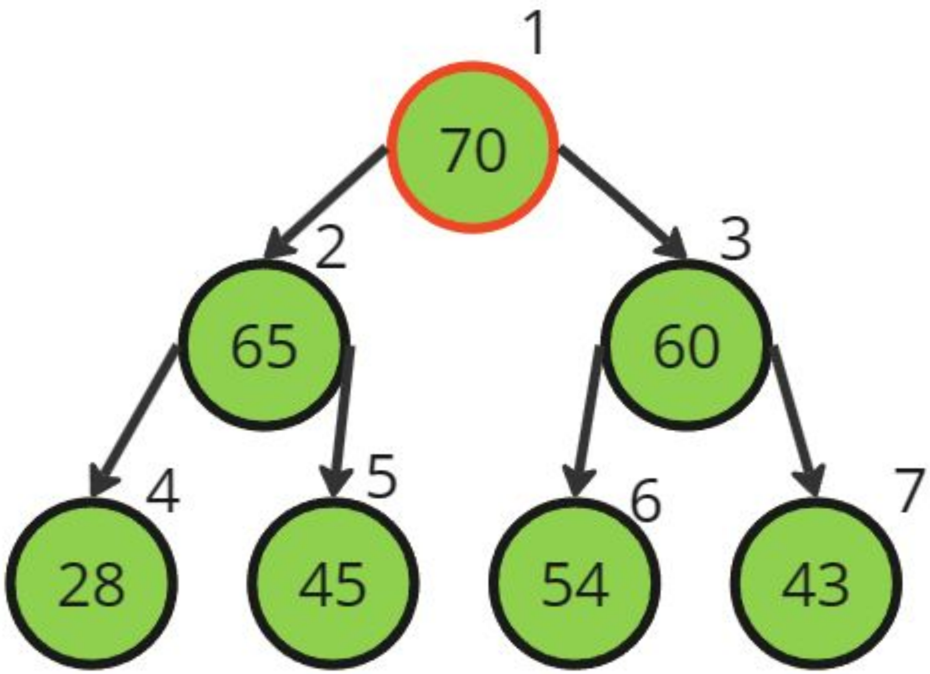
Пирамидальное дерево

Корневой элемент заменяется последним элементом в куче. После извлечения максимального элемента, последний элемент временно занял позицию корня. Это позволяет сохранить свойства кучи, поскольку после этой замены нарушается только свойство кучи на корне, и остальная часть кучи остается валидной.

```
int extractMax():
  int max = a[0]
  a[0] = a[a.heapSize - 1]
  a.heapSize = a.heapSize - 1
  siftDown(0)
  return max
```

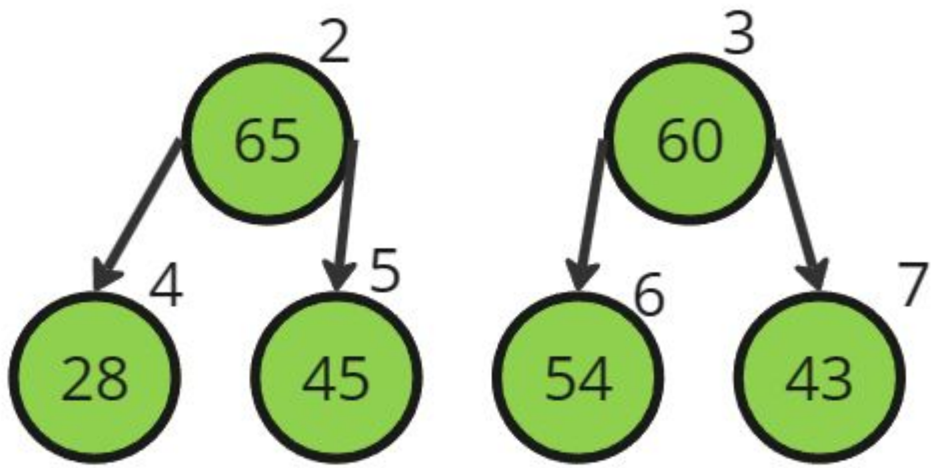
После замены корневого элемента нужно восстановить свойство кучи, просеивая новый корневой элемент вниз по дереву.

Пирамидальное дерево: extractMax



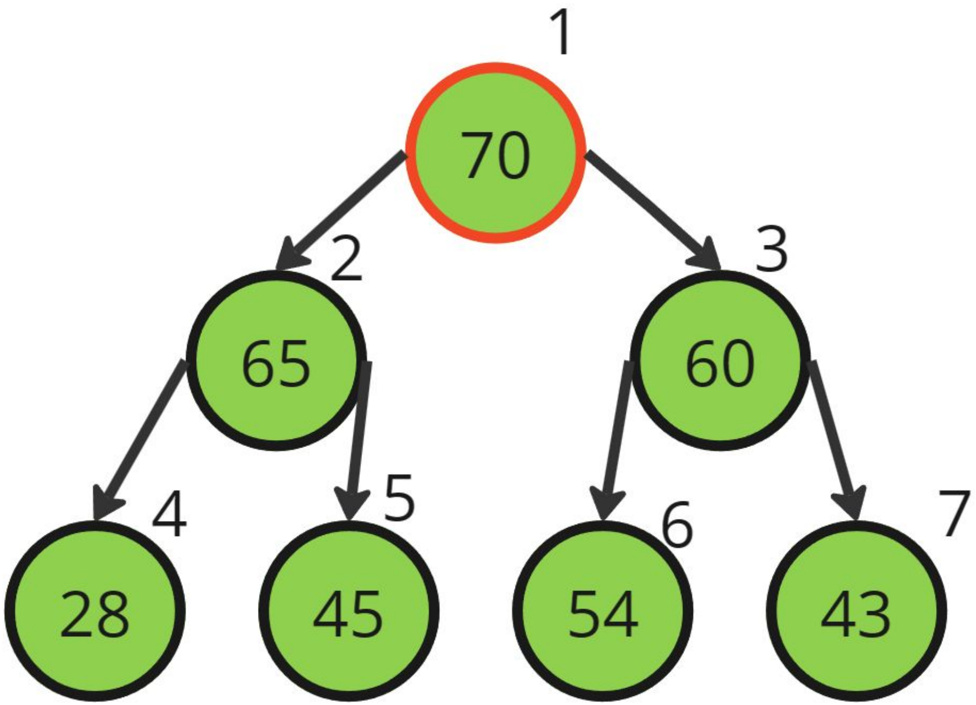
Пирамидальное дерево: extractMax

Если удалим сразу корень, то дерево развалится



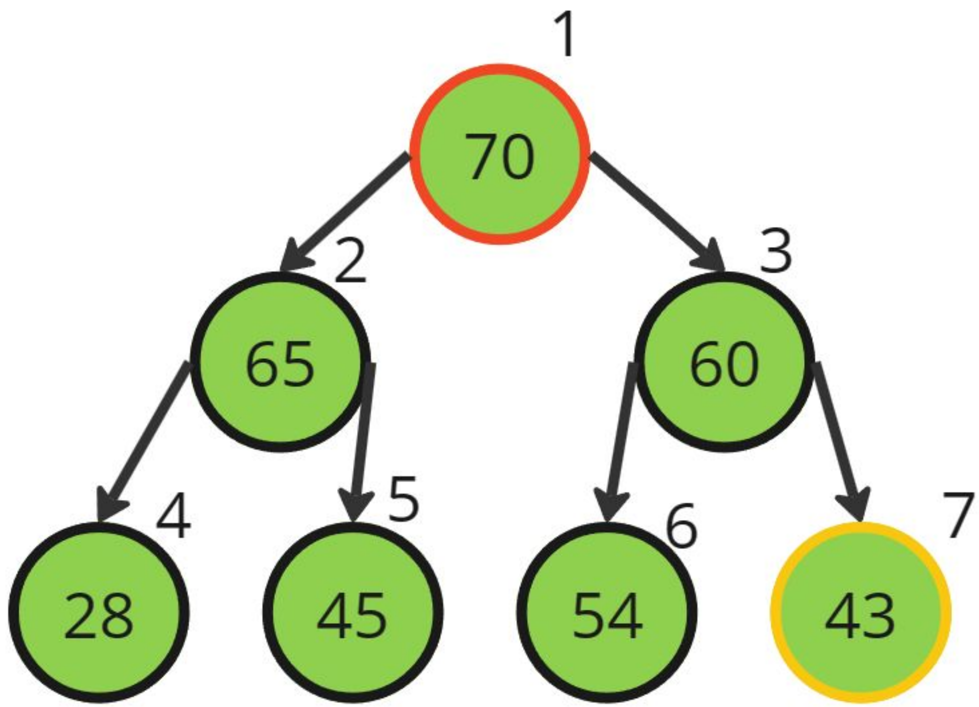
Пирамидальное дерево: extractMax

Чтобы сохранить структуру дерева, поменяем значение корня со значением последнего листа



Пирамидальное дерево: extractMax

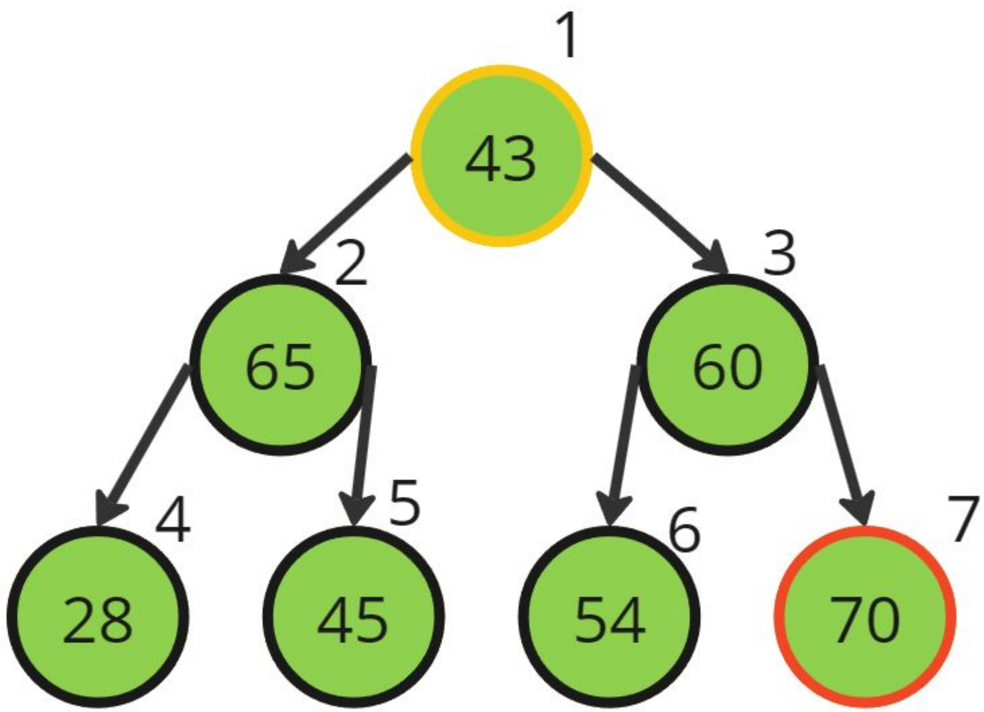
Чтобы сохранить структуру дерева, поменяем значение корня со значением последнего листа



Пирамидальное дерево: extractMax

Чтобы сохранить структуру дерева, поменяем значение корня со значением последнего листа

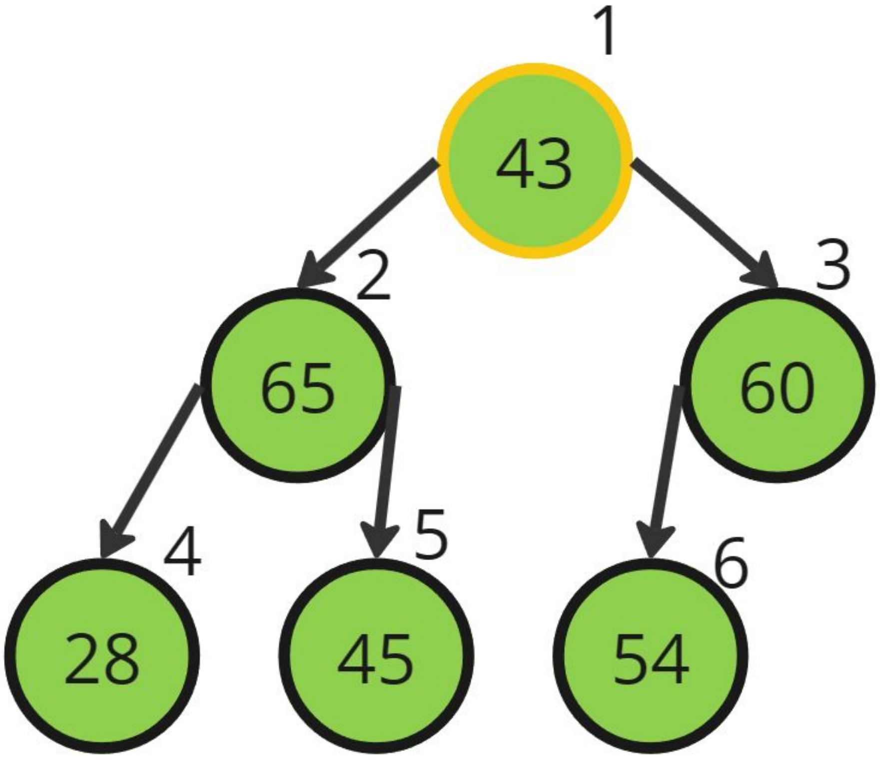
```
a[0] = a[a.heapSize - 1]
```



Пирамидальное дерево: extractMax

Теперь безболезненно удалим лист

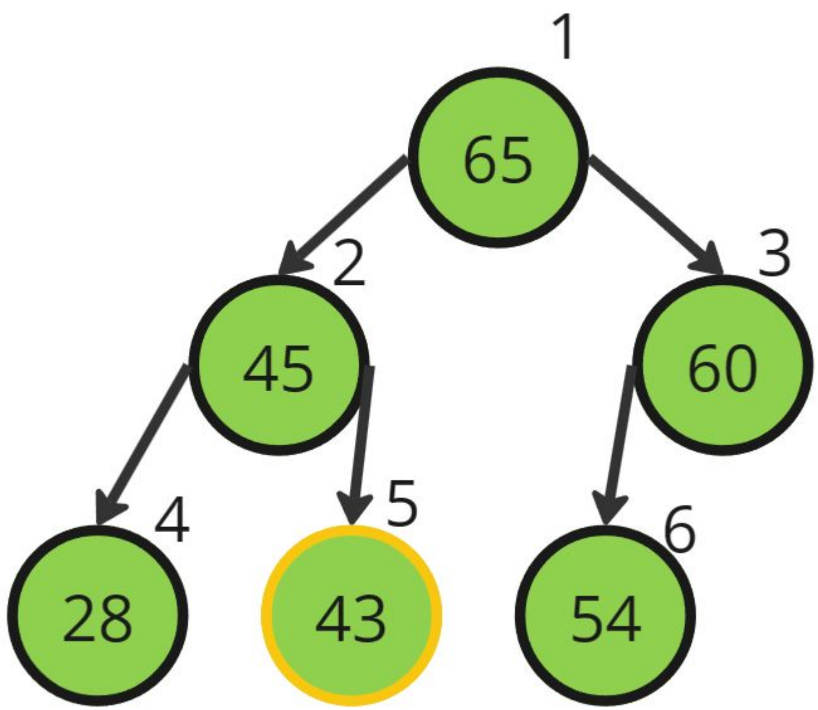
```
a.heapSize = a.heapSize - 1
```



Пирамидальное дерево: extractMax

Нужно просеять вниз корень, чтобы элемент нашел свое место

```
siftDown(0)
```

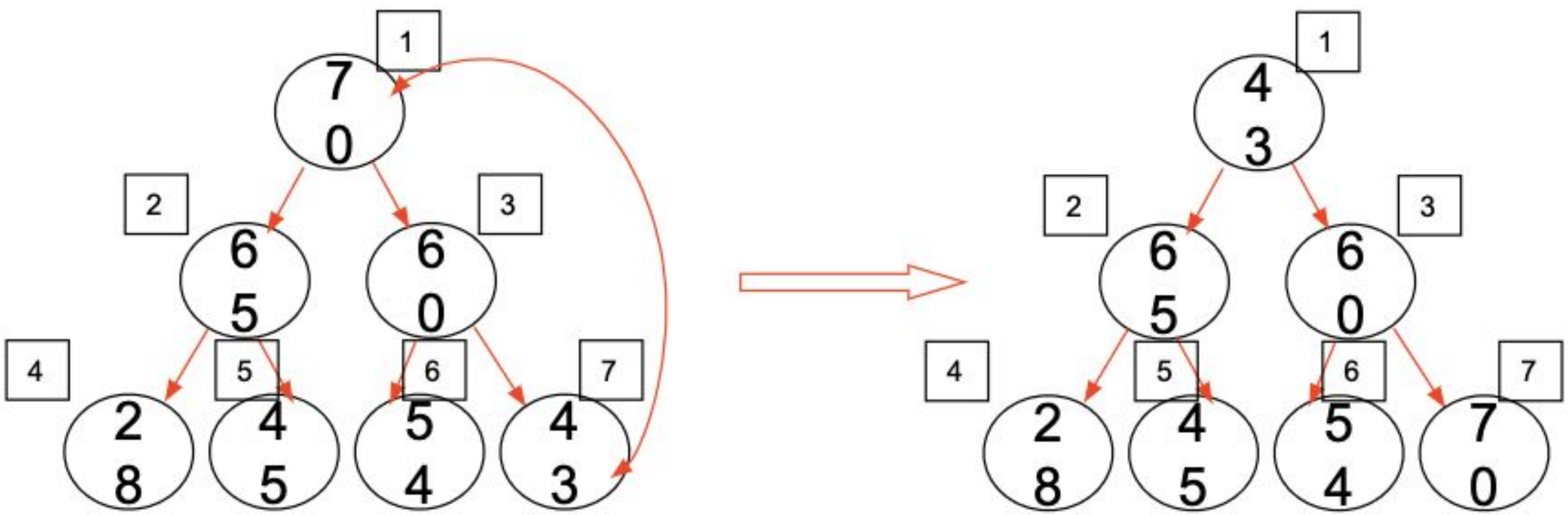


Пирамидальная сортировка

Использовать пирамидальное дерево для сортировки массива можно следующим образом. Допустим из элементов сортируемого массива {45, 28, 54, 43, 70, 60, 65} удалось некоторым образом построить рассматриваемое пирамидальное дерево

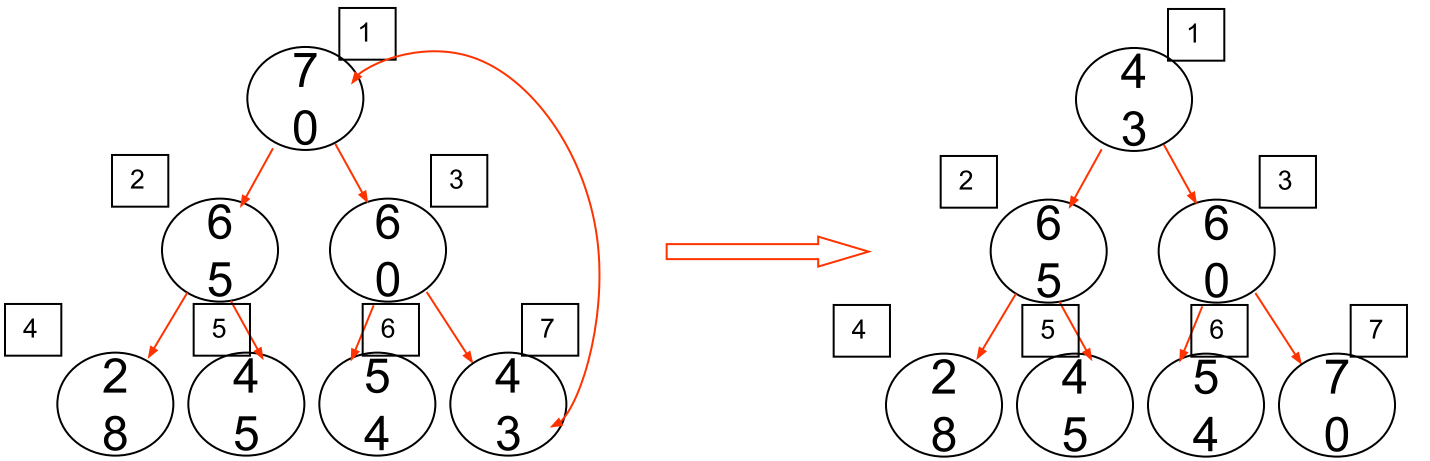
Пирамидальная сортировка

Использовать пирамидальное дерево для сортировки массива можно следующим образом. Допустим из элементов сортируемого массива
{45, 28, 54, 43, 70, 60, 65} удалось некоторым образом построить рассматриваемое пирамидальное дерево

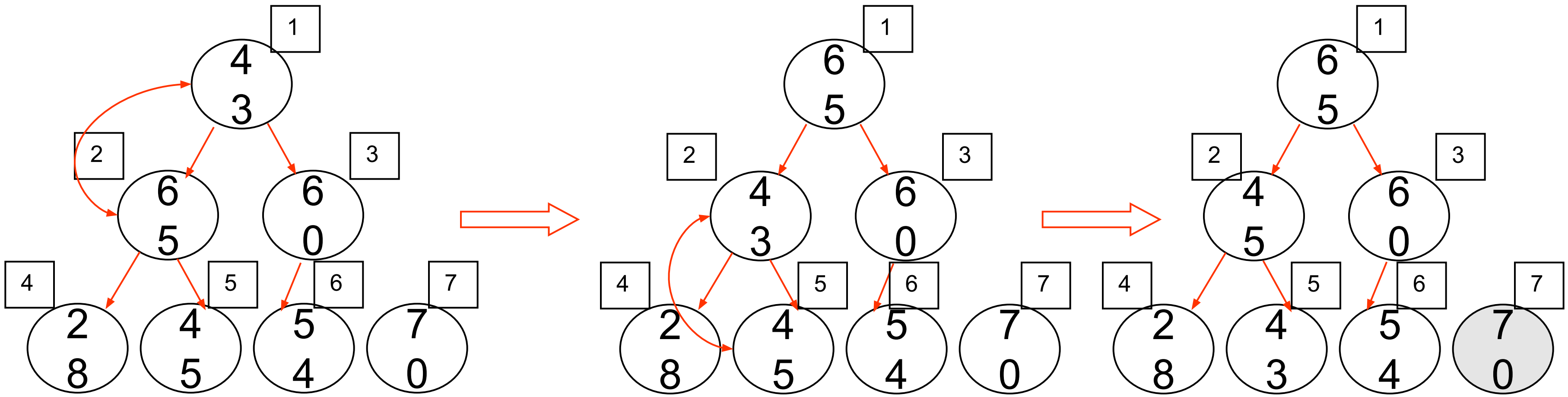


Пирамидальная сортировка

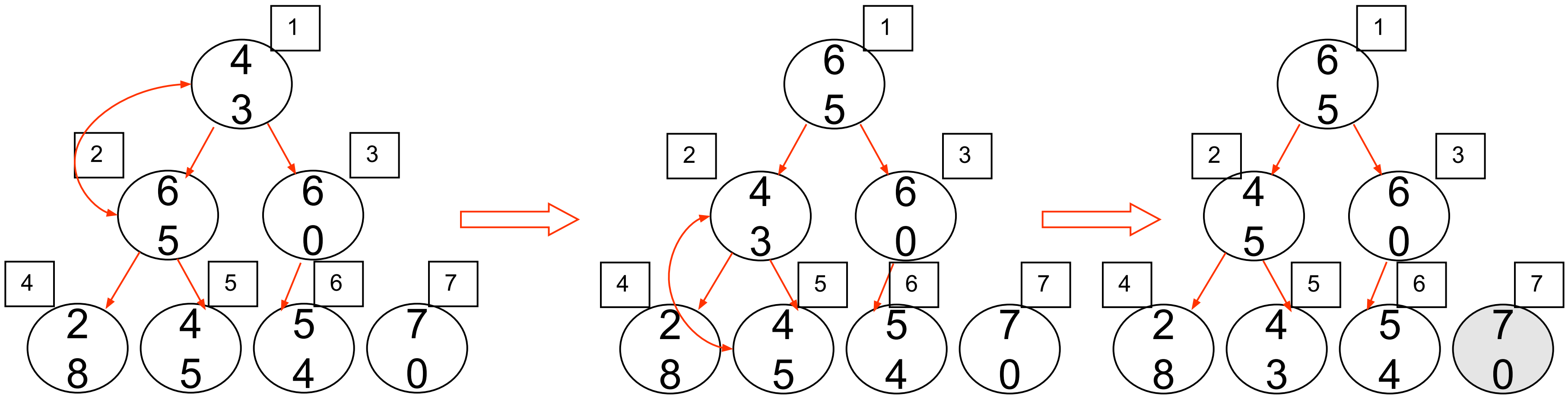
Использовать пирамидальное дерево для сортировки массива можно следующим образом. Допустим из элементов сортируемого массива {45, 28, 54, 43, 70, 60, 65} удалось некоторым образом построить рассматриваемое пирамидальное дерево



По основному свойству пирамиды максимальный элемент (70) находится в её корне и имеет индекс 1. При сортировке в порядке возрастания максимальный элемент должен находиться в конце массива, быть последним, поэтому узел с индексом 1 можно поменять местами с последним узлом (43), имеющим индекс 7, после чего исключить его из дальнейшей сортировки

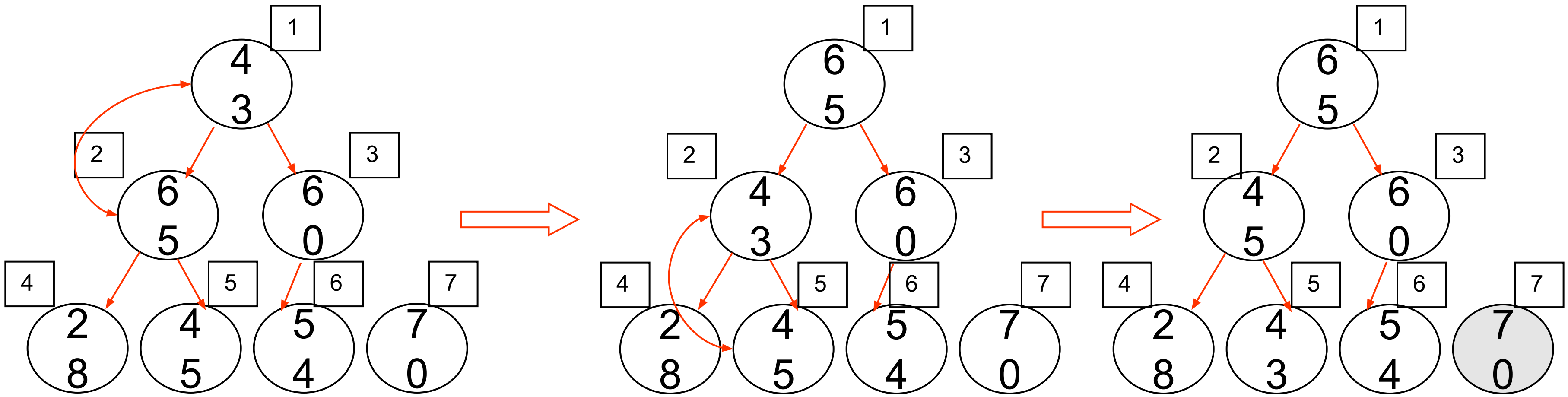


Однако в результате получится бинарное дерево, которое имеет нарушение пирамидальности в корневом узле



Однако в результате получится бинарное дерево, которое имеет нарушение пирамидальности в корневом узле

Дерево, у которого свойство упорядоченности ($x[k] \geq x[2k]$) $\wedge$ ($x[k] \geq x[2k+1]$) выполняется для всех узлов, *кроме корневого* называется *частично упорядоченным пирамидальным деревом*



Дерево, у которого свойство упорядоченности $(x[k] \ge x[2k]) \wedge (x[k] \ge x[2k+1])$ **выполняется для всех узлов,** ***кроме корневого*** **называется** ***частично упорядоченным пирамидальным деревом***

Оказывается, что восстановление пирамидальности выполняется очень простым способом: нужно поменять местами корень и *большший* из его двух дочерних узлов, а затем рекурсивно применить эту процедуру к выбранному поддереву

В результате такого преобразования вновь получится пирамидальное дерево, в корне которого находится максимальный ключ

Просеивание вниз: рекурсивная реализация

```
void siftdown(int * a, int n, int i)
{
    int j = i;
    if (2 * i + 1 < n && a[j] < a[2 * i + 1]) {
        j = 2 * i + 1;
    }
    if (2 * i + 2 < n && a[j] < a[2 * i + 2]) {
        j = 2 * i + 2;
    }
    if (i != j) {
        swap(a[i], a[j]);
        siftdown(a, n, j);
    }
}
```

выбираем наибольшее значение из узла и его детей, если таковые имеются

Если просеиваемый узел есть наибольший, то завершаем алгоритм

- Оценка сложности: $O(\log N)$
- Дополнительная память: $O(\log N)$

Требуется доп. память O(logN), так как алгоритм реализован рекурсивно и необходимо хранить точки возврата. Максимальная глубина рекурсии - высота кучи (logN)

Просеивание вниз: итеративная реализация

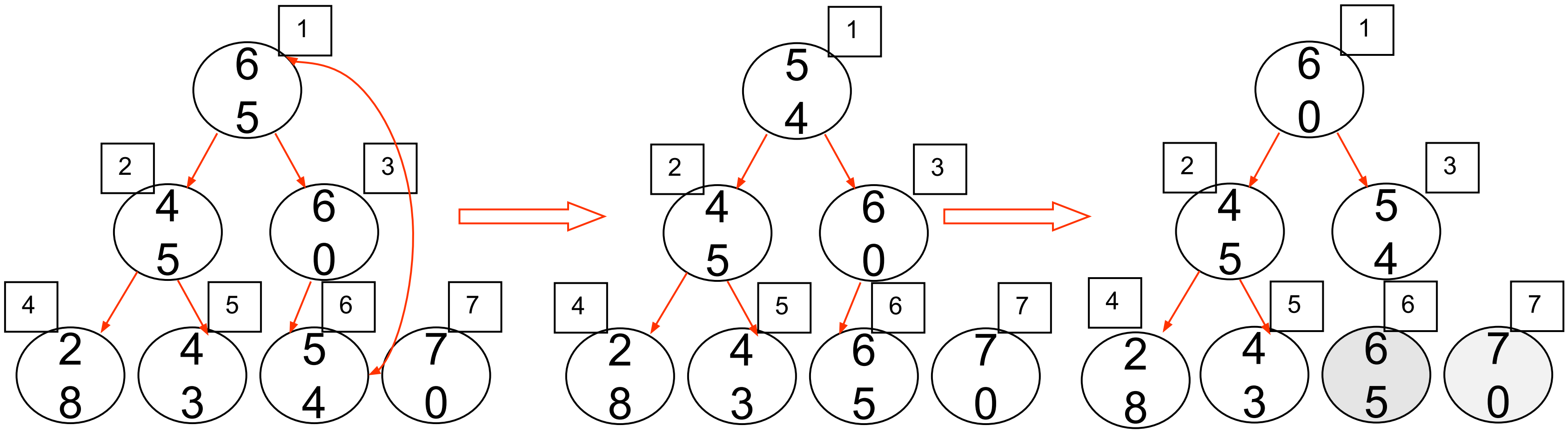
```
function siftDown(i : int):
    while 2 * i + 1 < a.heapSize
        left = 2 * i + 1
        right = 2 * i + 2
        j = left
        if   right < a.heapSize and a[right] > a[left]
            j = right
        if   a[i] >= a[j]
            break
        swap(a[i], a[j])
        i = j
```

Вычисляем номер левого ребенка, пока такой существует

Сравниваем левого и правого ребенка

Если узел больше либо равен чем максимальный из его детей, то завершаем алгоритм

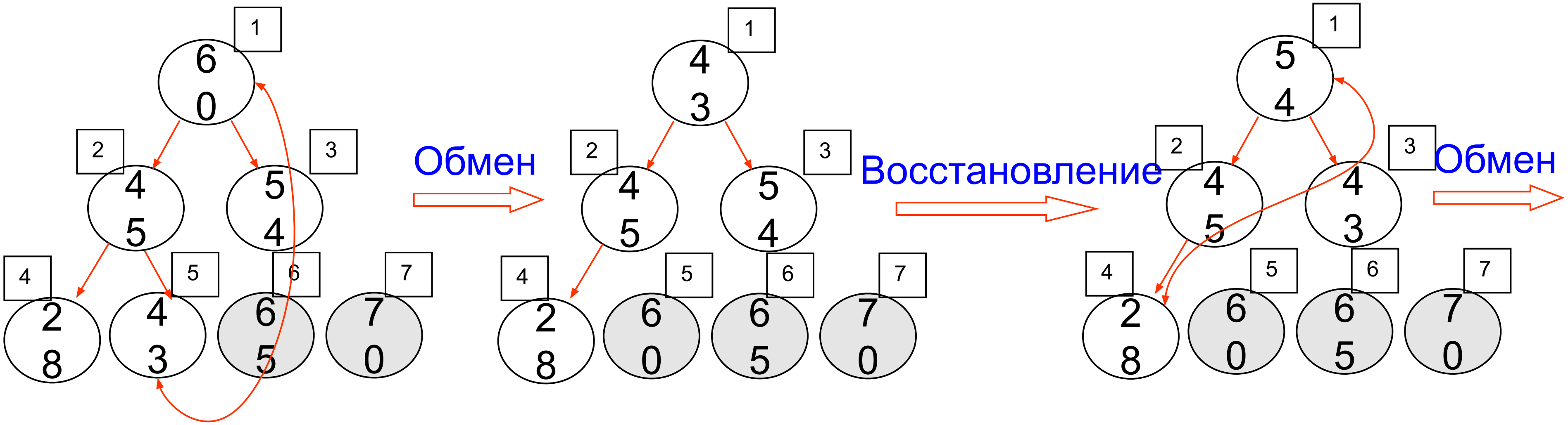
Эта реализации требует O(1) доп. памяти, так как алгоритм состоит из одного цикла.

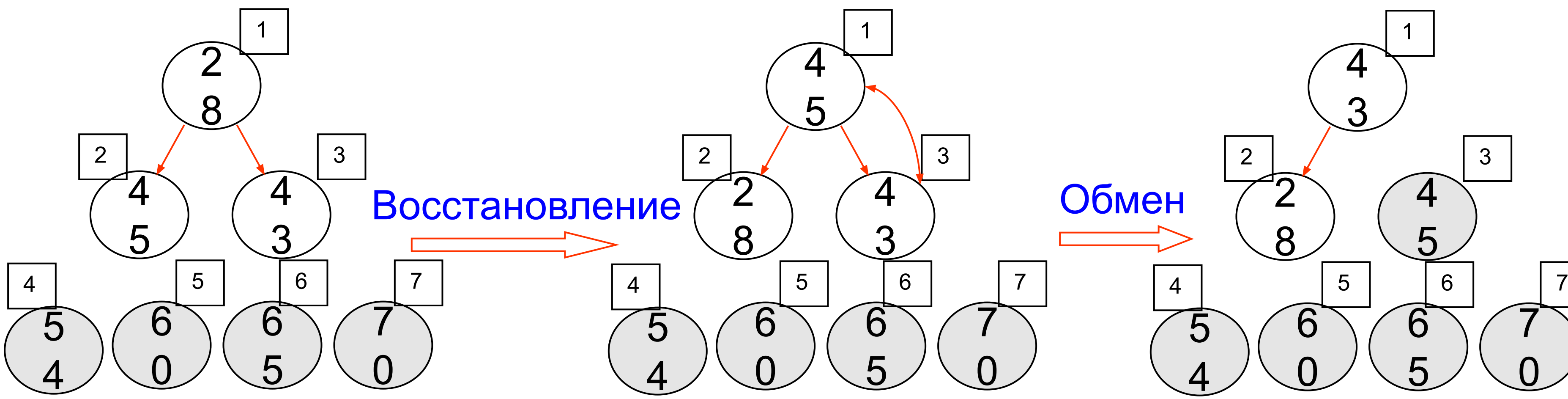


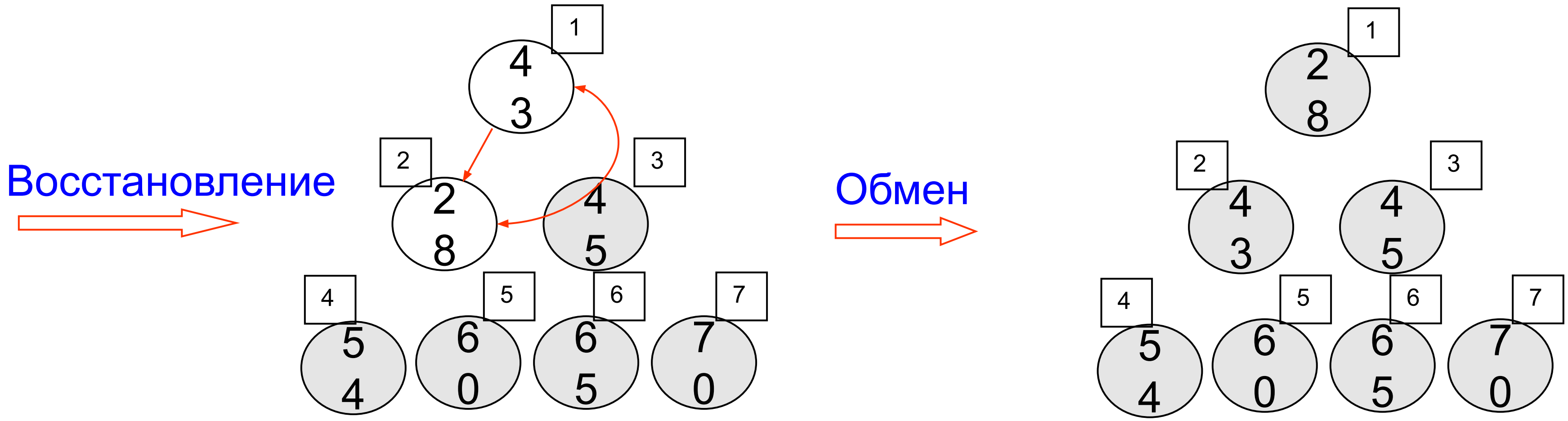
Можно сказать, что произошёл выбор наибольшего элемента в неупорядоченной части массива, который теперь можно присоединить к уже упорядоченной части, поменяв местами с *предпоследним* элементом

Вновь получено частично упорядоченное пирамидальное дерево, которое можно тем же способом преобразовать в пирамидальное

Этот процесс следует продолжать до тех пор, пока все элементы массива не окажутся на своих местах


Обмен
Восстановление
Обмен


Восстановление
Обмен



В результате получена структура, в которой ключи узлов упорядочены по возрастанию

Но для применения этого способа нужно уметь представлять сортируемый массив в виде пирамидального дерева

В принципе, для *начального построения* дерева из заданного сортируемого массива применяется практически тот же самый подход, что и для *восстановления* дерева

Вначале из заданного массива, исходя из соответствия индексов массива индексам узлов пирамидального дерева, строится дерево, в котором условие пирамидальности *ещё не выполняется*

Построение пирамиды через последовательный insert

Составить пирамиду можно просто добавив все элементы из исходного массива в кучу

Будем использовать функцию insert()

Построение пирамиды через последовательный insert

Составить пирамиду можно просто добавив все элементы из исходного массива в кучу

Будем использовать функцию insert()

insert(5)

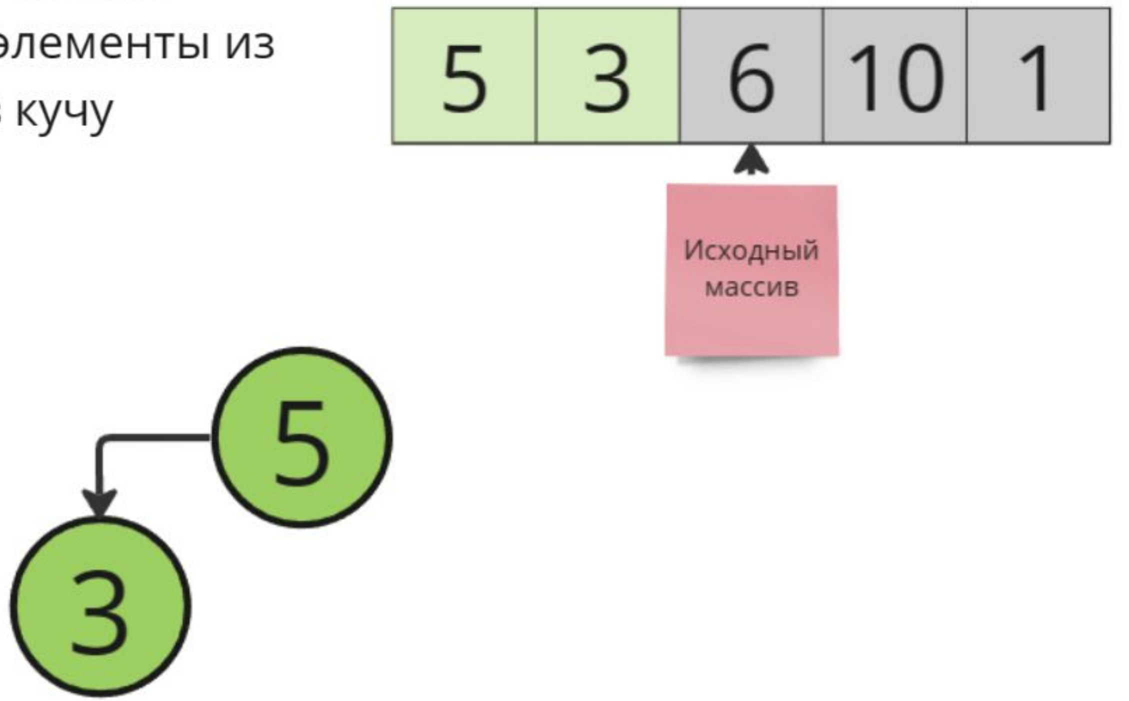
Построение пирамиды через последовательный insert

Составить пирамиду можно просто добавив все элементы из исходного массива в кучу

Будем использовать функцию insert()

insert(5)

insert(3)



Построение пирамиды через последовательный insert

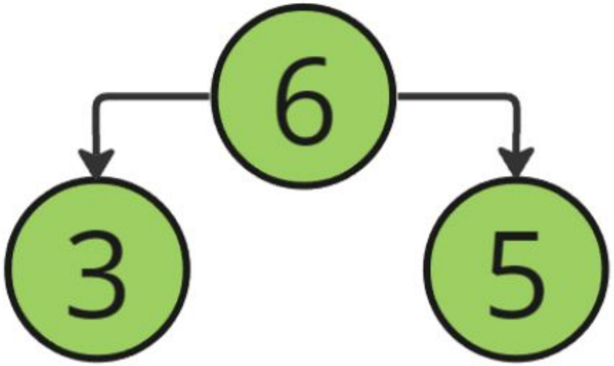
Составить пирамиду можно просто добавив все элементы из исходного массива в кучу

Будем использовать функцию insert()

insert(5)

insert(3)

insert(6)



Построение пирамиды через последовательный insert

Составить пирамиду можно просто добавив все элементы из исходного массива в кучу

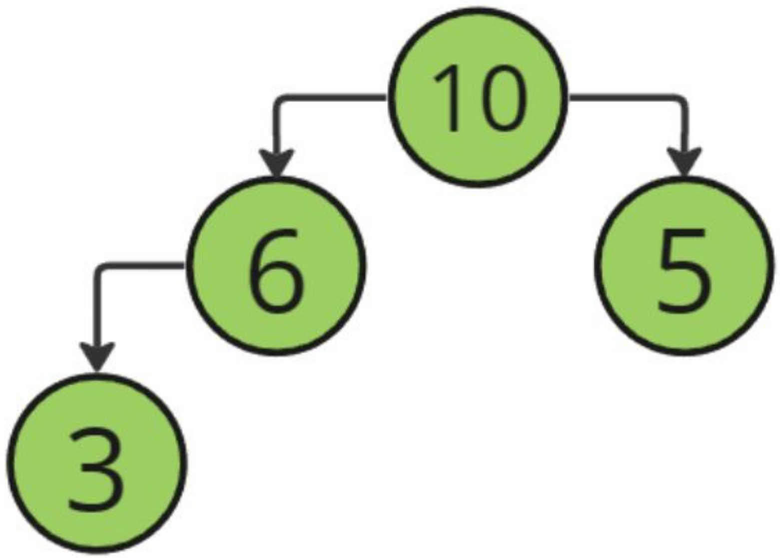
Будем использовать функцию insert()

insert(5)

insert(3)

insert(6)

insert(10)



Построение пирамиды через последовательный insert

Составить пирамиду можно просто добавив все элементы из исходного массива в кучу

Будем использовать функцию insert()

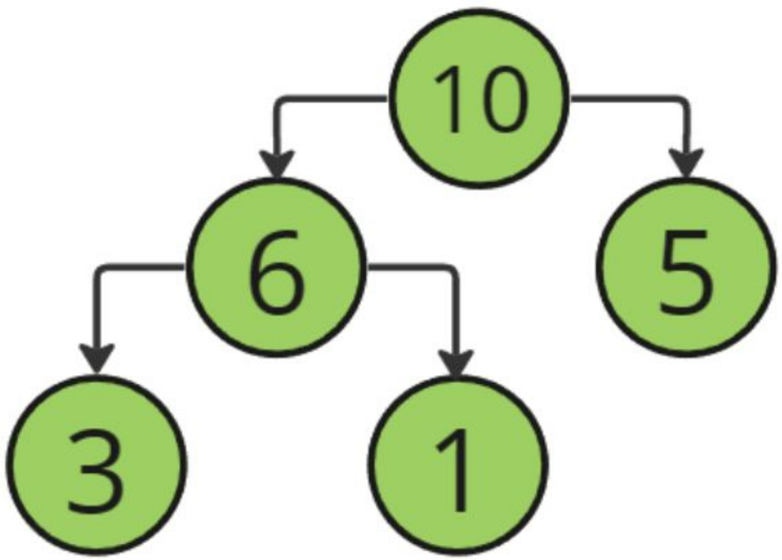
insert(5)

insert(3)

insert(6)

insert(10)

insert(1)



Асимптотика построения пирамиды через последовательный insert

Дан массив $a[0..n-1]$.

Строим пирамиду с минимумом в корне.

Делаем нулевой элемент массива корнем, а дальше по очереди добавить все его элементы в конец кучи и запускать от каждого добавленного элемента siftUp.

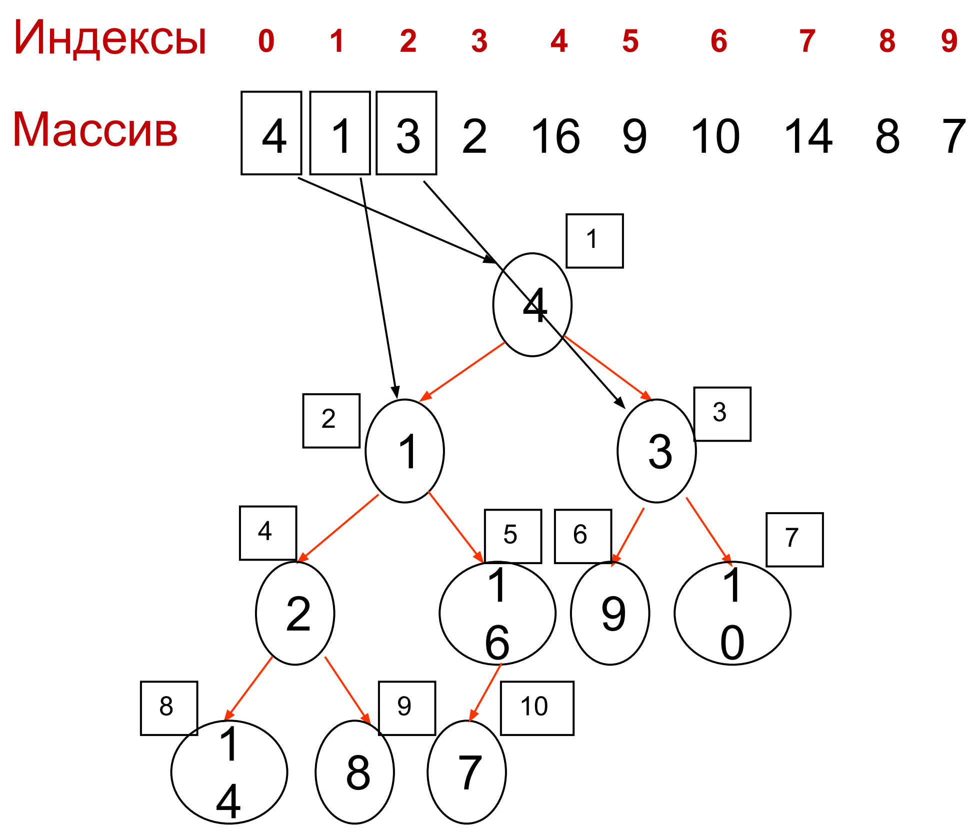
siftUp работает за O(log(n))

Тогда

Временная оценка такого алгоритма - O(nlog(n)).

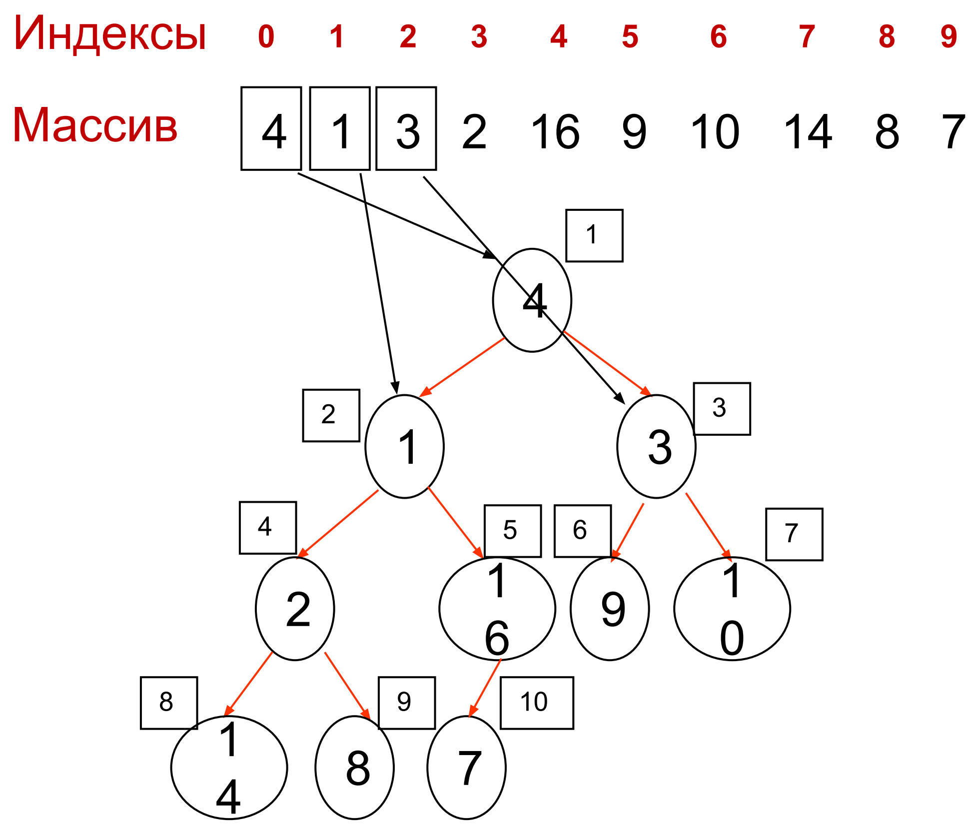
Построение пирамиды

Возьмем для сортировки , например, такой 10-элементный массив: {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}. Элементы по одному выбираем из массива и включаем в пирамидальное дерево в соответствии с правилами его заполнения, ***не обращая при этом внимания на значения элементов***



Построение пирамиды

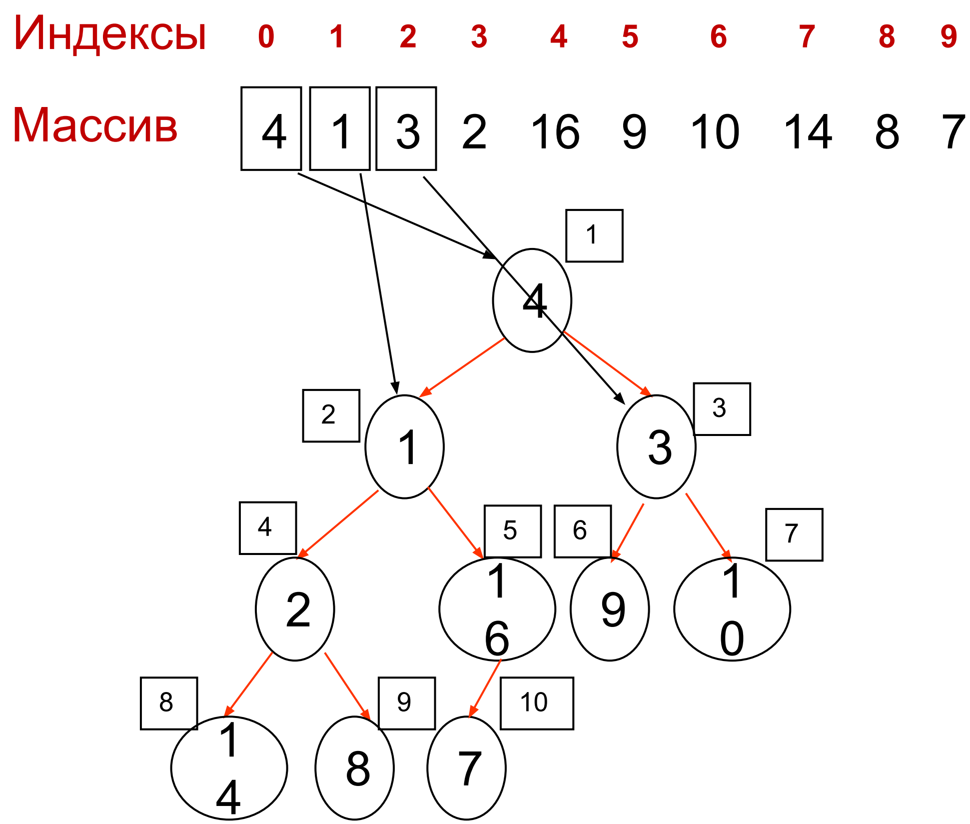
Возьмем для сортировки , например, такой 10-элементный массив: {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}. Элементы по одному выбираем из массива и включаем в пирамидальное дерево в соответствии с правилами его заполнения, ***не обращая при этом внимания на значения элементов***



Получилось бинарное дерево общего вида, ключи узлов которого являются соответствующими элементами сортируемого массива

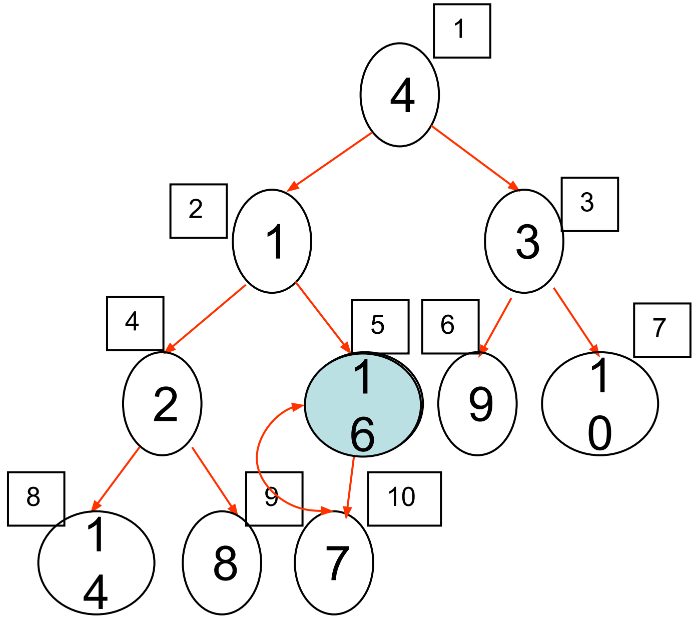
Построение пирамиды

Возьмем для сортировки , например, такой 10-элементный массив: {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}. Элементы по одному выбираем из массива и включаем в пирамидальное дерево в соответствии с правилами его заполнения, ***не обращая при этом внимания на значения элементов***

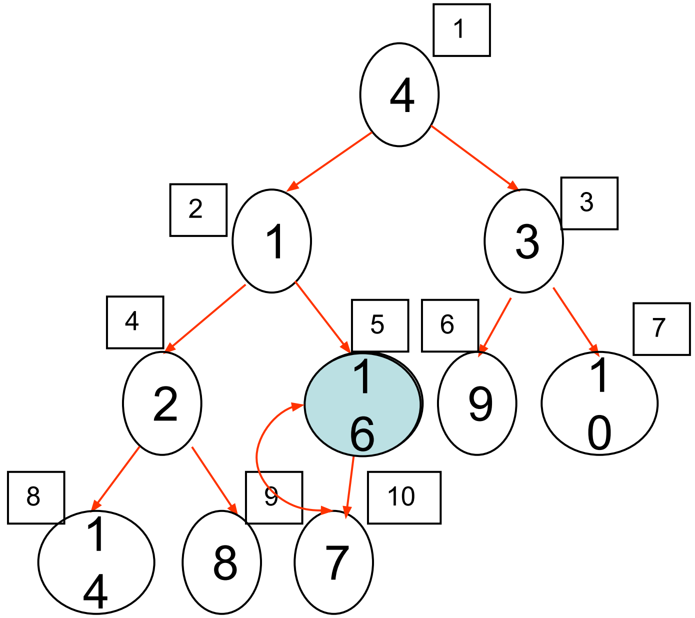


Получилось бинарное дерево общего вида, ключи узлов которого являются соответствующими элементами сортируемого массива

Это дерево *не является пирамидальным*, но его довольно просто можно преобразовать в пирамидальное

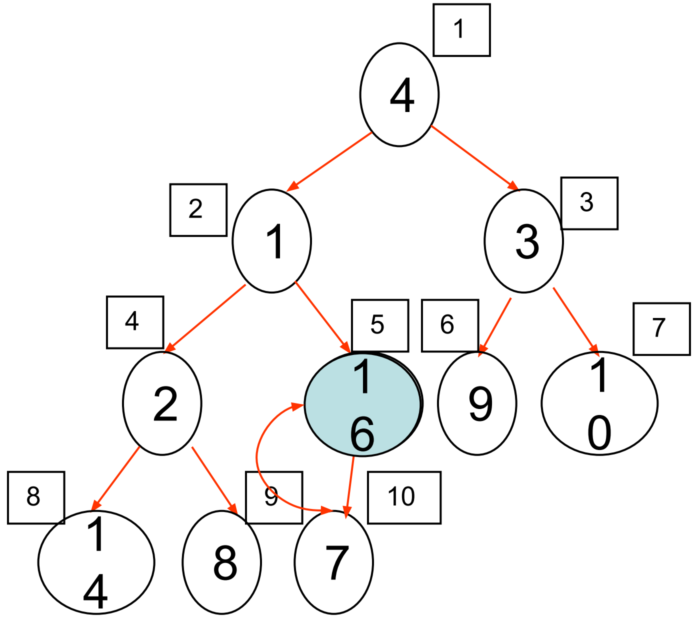


Итак, выбираем узел с индексом 5 и преобразуем поддерево в корне которого он находится , так чтобы выполнялось свойство пирамидальности



Итак, выбираем узел с индексом 5 и преобразуем поддерево в корне которого он находится , так чтобы выполнялось свойство пирамидальности

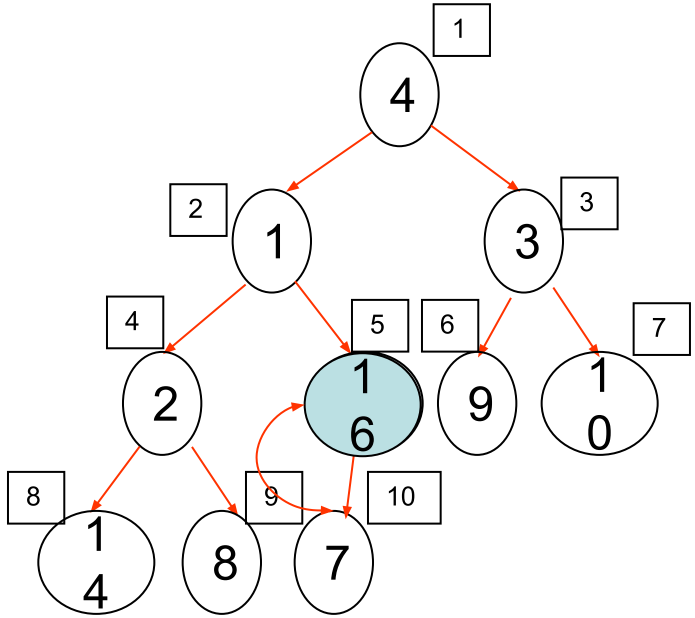
Можно считать, что *листья* этого дерева уже являются пирамидальными поддеревьями. ***Листья образуют нижний уровень дерева, его нижний слой***



Итак, выбираем узел с индексом 5 и преобразуем поддерево в корне которого он находится , так чтобы выполнялось свойство пирамидальности

Можно считать, что *листья* этого дерева уже являются пирамидальными поддеревьями. ***Листья образуют нижний уровень дерева, его нижний слой***

По построению дерева количество листьев всегда равно N-int(N/2). Причём в массиве они занимают последние позиции: x[i], i=int(N/2)+1,…,N

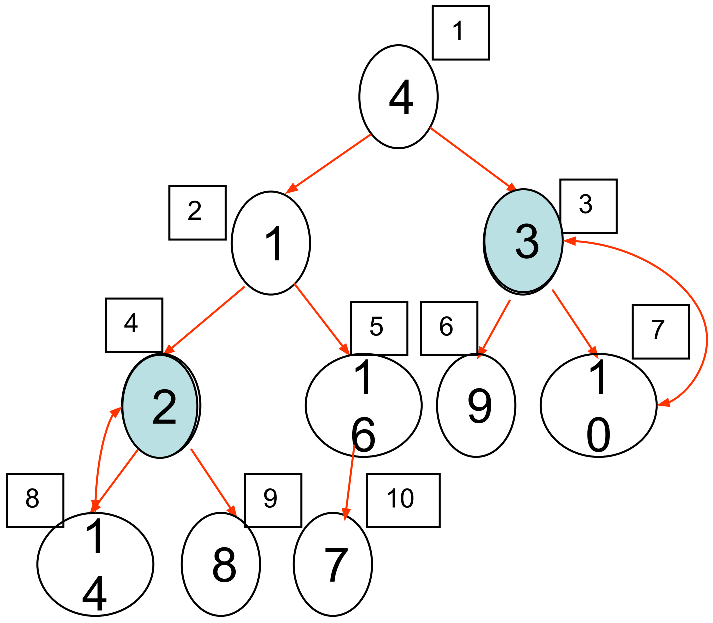


Итак, выбираем узел с индексом 5 и преобразуем поддерево в корне которого он находится , так чтобы выполнялось свойство пирамидальности

Можно считать, что *листья* этого дерева уже являются пирамидальными поддеревьями. ***Листья образуют нижний уровень дерева, его нижний слой***

По построению дерева количество листьев всегда равно N-int(N/2). Причём в массиве они занимают последние позиции: x[i], i=int(N/2)+1,…,N

<u>Процедура преобразования полученного дерева</u> сводится к последовательному перебору *в обратном порядке* всех ещё не упорядоченных узлов (в рассматриваемом примере узлов с индексами 5, 4, …,1) и применению описанного выше приёма приведения частично упорядоченного дерева к пирамидальному, *к совокупности узлов, состоящей из вновь рассматриваемого и уже пирамидальных*



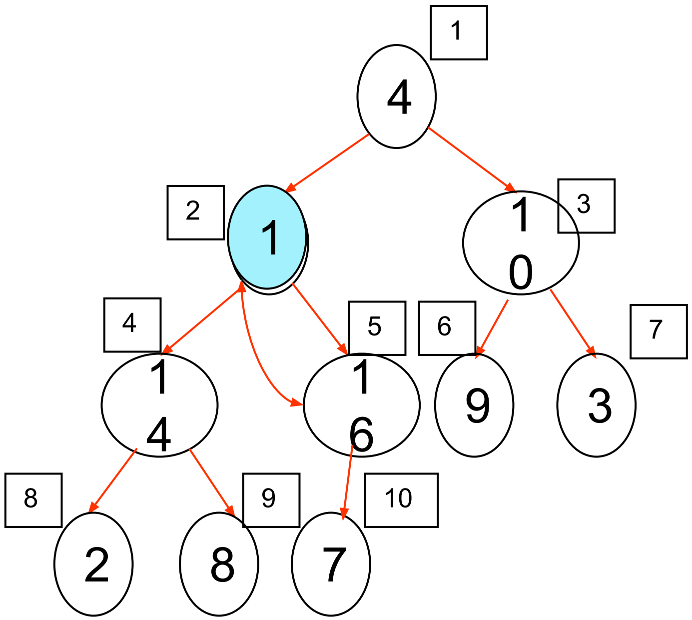
Так как узел 5, содержащий ключ 16, имеет единственный дочерний, ключ которого меньше, чем у узла 5, можно утверждать, что поддерево с корнем 16 уже является пирамидальным

Следующим по порядку рассматривается узел с индексом 4

Он имеет два дочерних с ключами 8 и 14. Большим из них является узел с ключом 14 и его нужно поменять местами с корневым

Следующим по порядку рассматривается узел с индексом 3

Он имеет два дочерних с ключами 9 и 10. Большим из них является узел с ключом 10 и его нужно поменять местами с корневым



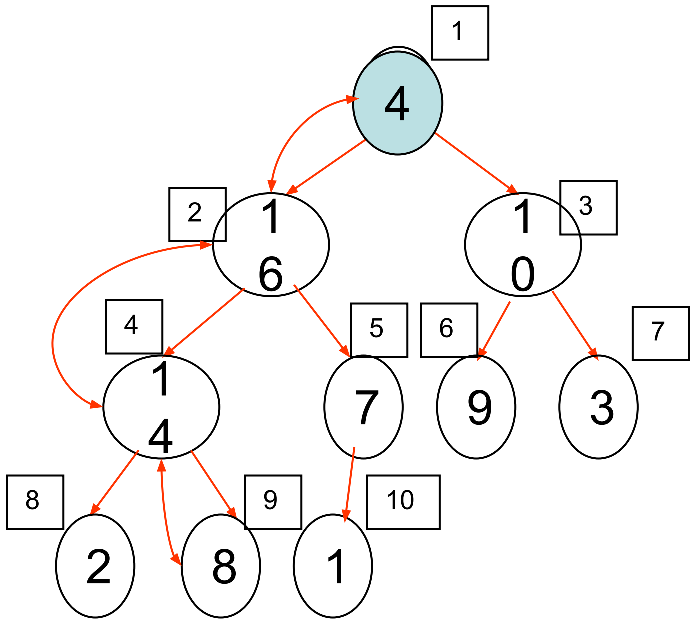
Далее рассматривается узел с индексом **2**

Два его дочерних узла имеют ключи **14** и **16**

Больший из них **16** должен поменяться местами с корневым узлом

Но такой обмен нарушил свойство пирамидальности для поддерева, имеющего в качестве корня узел с индексом **5**

Следовательно, нужно восстановить пирамидальность, выполнив обмен для нового корня и его дочернего узла с индексом **10**



На последнем этапе выбираем узел с индексом 1

Вначале меняем его местами с большим дочерним (16)

Теперь работаем в поддереве с узлом, имеющим индекс 2. Его нужно поменять местами с большим дочерним (14)

И последним преобразуется поддерево, имеющее корнем узел с индексом 4. Он меняется местами с дочерним узлом, с индексом 9

В результате из сортируемого массива получено пирамидальное дерево, к которому уже можно применять описанный выше способ сортировки.

Заметим, что это пирамидальное дерево эквивалентно массиву {16, 14, 10, 8, 7, 9, 3, 2, 4, 1}, он существенно отличается от исходного {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}

Сравнение асимптотики

Вариант 1: последовательное добавление	O(n*logn)	n раз выполняем добавление в кучу, которое выполняется за O(logn)

Сравнение асимптотики

Вариант 1: последовательное добавление	O(n*logn)	n раз выполняем добавление в кучу, которое выполняется за O(logn)
Вариант 2: просеивание вниз, начиная от первых родителей	O(n)	

Пирамидальная сортировка

Фактически для реализации алгоритма пирамидальной сортировки само дерево строить необязательно. И алгоритм построения пирамидального дерева и последующее его преобразование сводятся к перемещениям соответствующих элементов в сортируемом массиве

- **Вначале нужно преобразовать сортируемый массив к виду, эквивалентному пирамидальному дереву. В результате на первое место попадет максимальный по всему массиву элемент x[1]=max {x[i], i=1,2,…, N}**

Пирамидальная сортировка

Сводя все предыдущие рассуждения вместе получим следующий алгоритм пирамидальной сортировки:

Пирамидальная сортировка

Сводя все предыдущие рассуждения вместе получим следующий алгоритм пирамидальной сортировки:

- **Вначале нужно преобразовать сортируемый массив к виду, эквивалентному пирамидальному дереву. В результате на первое место попадет максимальный по всему массиву элемент** $x[1]=\max\{x[i], i=1,2,\ldots, N\}$

Пирамидальная сортировка

- **Затем нужно выполнить N-1 шаг сортировки:**

Пирамидальная сортировка

- **Затем нужно выполнить N-1 шаг сортировки:**
  - **На первом шаге меняются местами находящийся на первом месте максимальный и N элементы. После чего последний элемент образует уже упорядоченную часть, а элементы с 1-го по N-1-й — неупорядоченную**

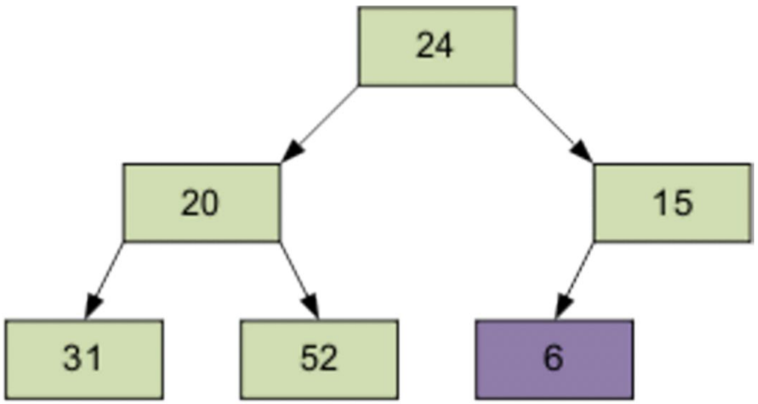
Пирамидальная сортировка

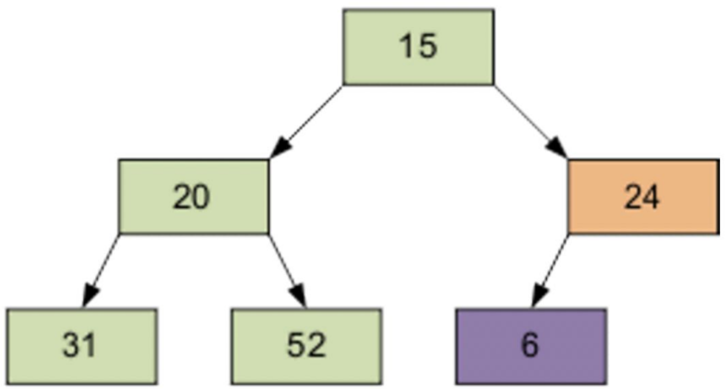
- **Затем нужно выполнить N-1 шаг сортировки:**
  - **На первом шаге меняются местами находящийся на первом месте максимальный и N элементы. После чего последний элемент образует уже упорядоченную часть, а элементы с 1-го по N-1-й — неупорядоченную**
  - **Получившаяся совокупность элементов неупорядоченной части массива эквивалента частично упорядоченному пирамидальному дереву, которое восстанавливается до полностью пирамидального. При этом максимальный из неупорядоченной части вновь попадает в начало массива**

ПИРАМИДАЛЬНАЯ СОРТИРОВКА

- **На втором шаге меняются местами 1 и N-1 элементы. После чего на «своих» местах уже находятся N-1-й и N-й элементы, образуя упорядоченную часть**

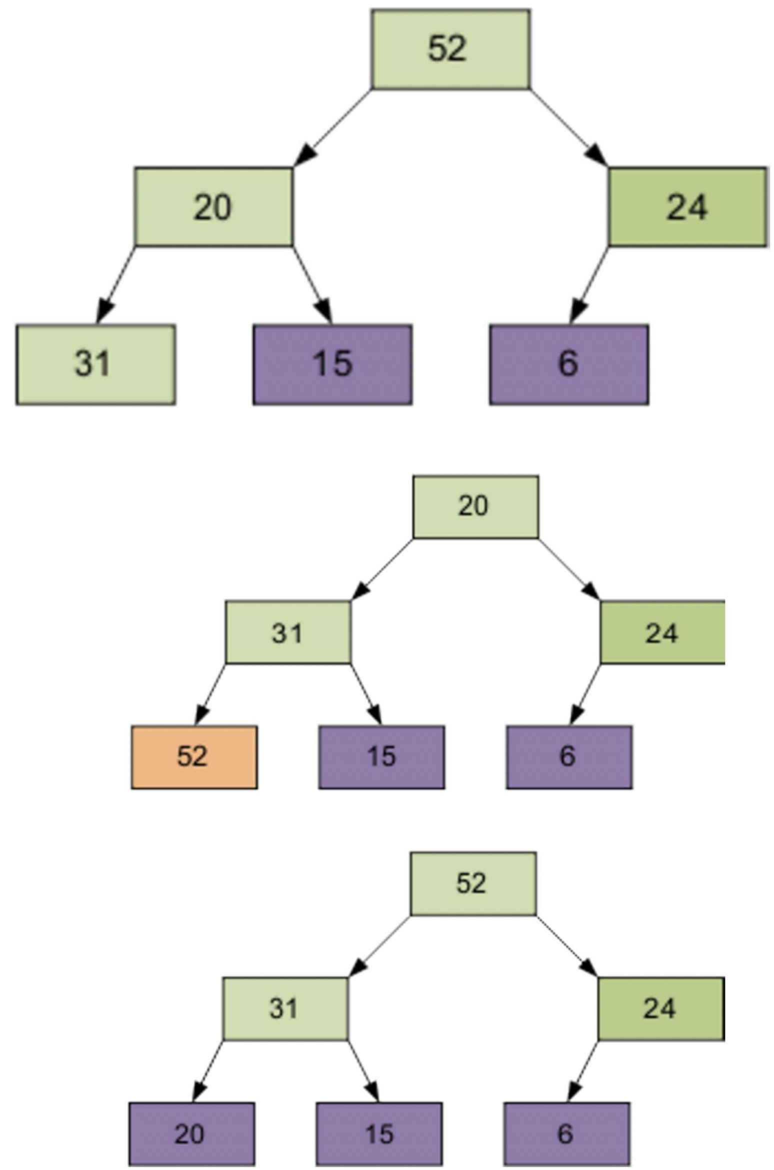
- **Первые N-2 элемента, образующие неупорядоченную часть, вновь преобразуются и максимальный из них перемещается на первое место**



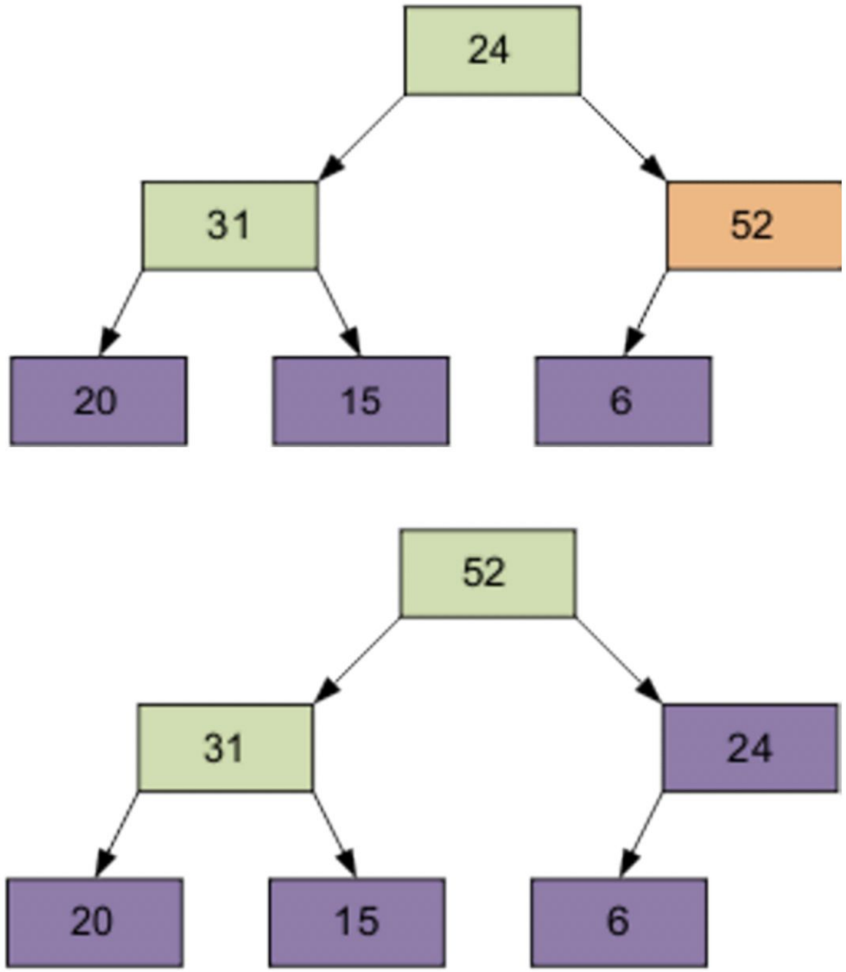


ПИРАМИДАЛЬНАЯ СОРТИРОВКА

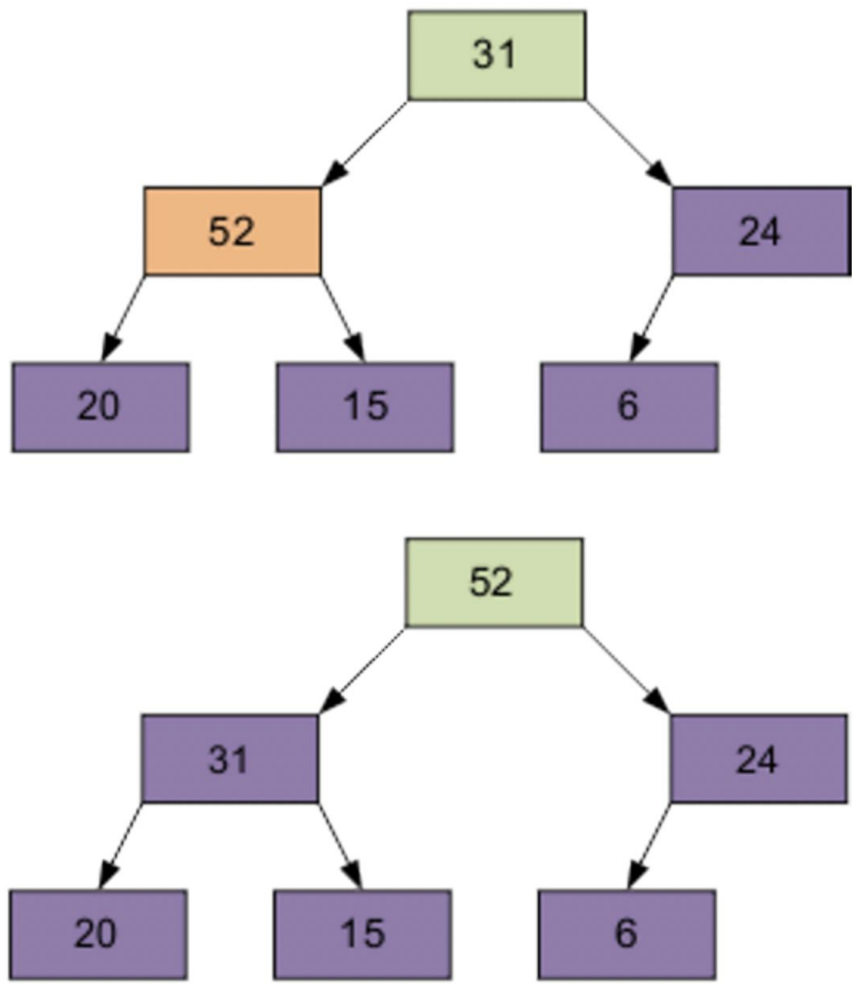
- **Таким образом, на i-ом шаге *всегда находящийся на первом* месте *максимальный* элемент x[1] меняется местами с *последним* элементом *неупорядоченной* части x[N-i+1]**
- **Затем, из уменьшенной на один элемент неупорядоченной части при восстановлении её пирамидальности выбирается наибольший и перемещается на первое место в массиве**



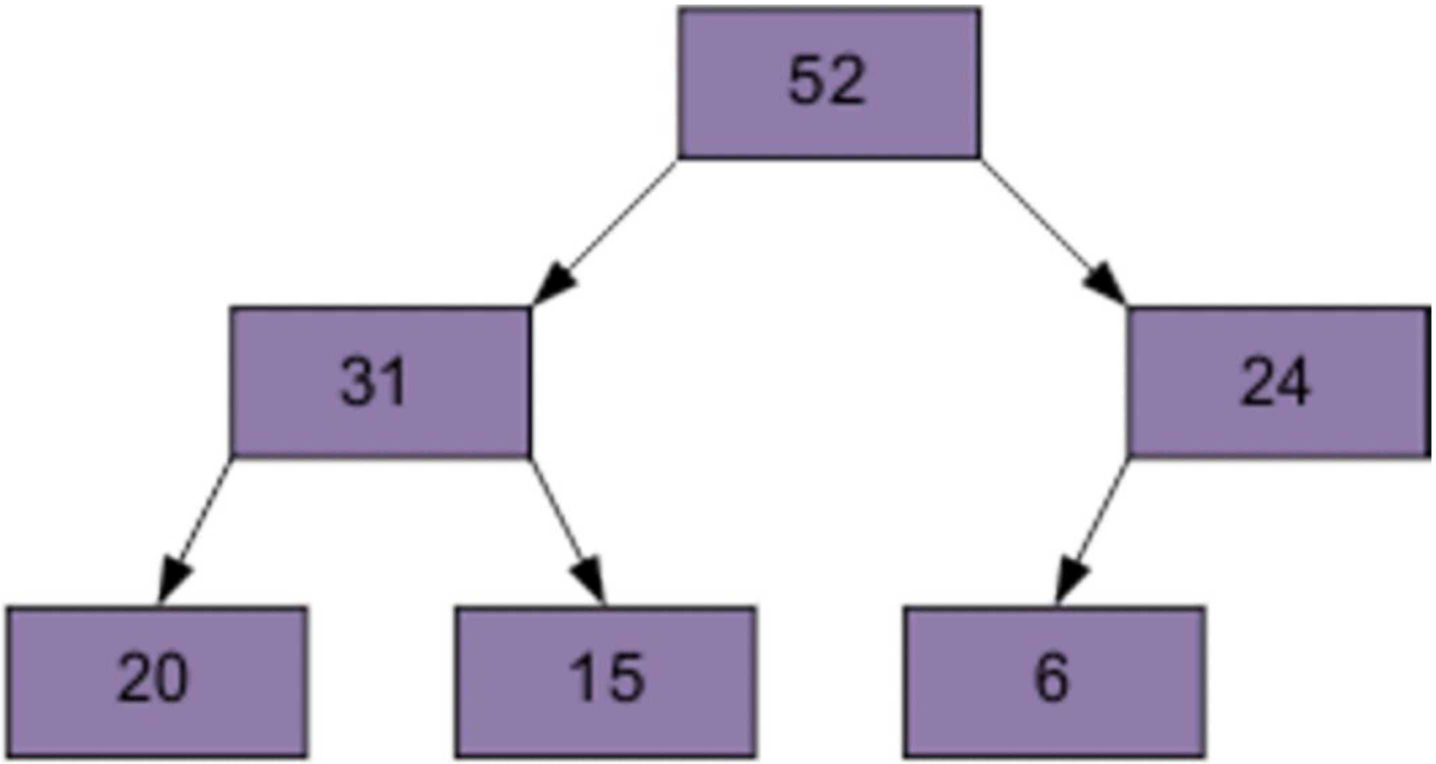
ПИРАМИДАЛЬНАЯ СОРТИРОВКА



ПИРАМИДАЛЬНАЯ СОРТИРОВКА

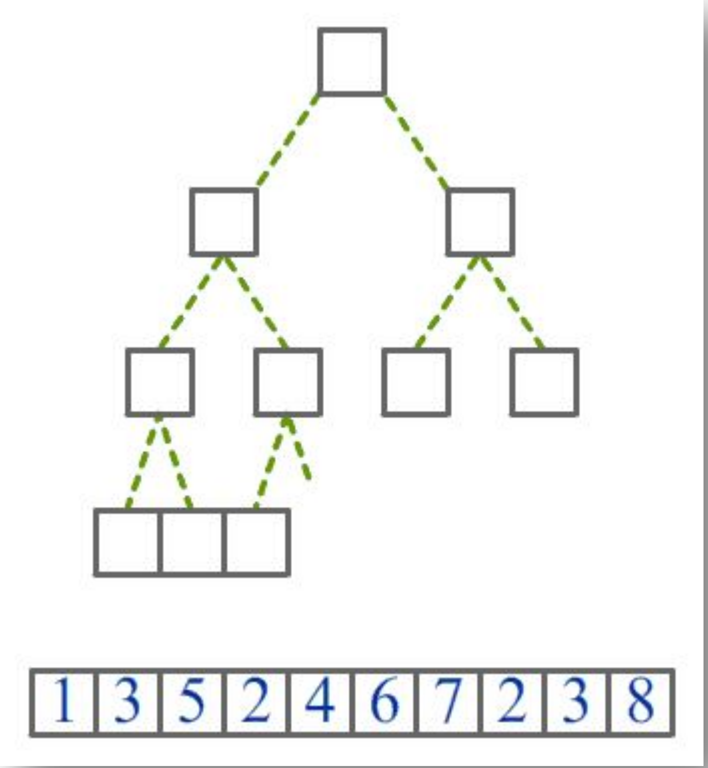


ПИРАМИДАЛЬНАЯ СОРТИРОВКА



ПИРАМИДАЛЬНАЯ СОРТИРОВКА, ПОСТРОЕНИЕ ПИРАМИДЫ

```
BuildHeap (A,n)
     Heapsize(A) ← n
     i ← Heapsize
     for i = ⌊n/2⌋ down to 1
          Heapify(A,i)
```



Пирамидальная сортировка

Анализ пирамидальной сортировки показывает, что её сложность $O(N*\log_2 N)$

```
fun heapSort(A : list <T>):
  buildHeap(A)
  heapSize = A.size
  for i = 0 to n - 1
    swap(A[0], A[n - 1 - i])
    heapSize--
    siftDown(A, 0, heapSize)
```

Пирамидальная сортировка

Анализ пирамидальной сортировки показывает, что её сложность $O(N*\log_2 N)$

```
fun heapSort(A : list <T>):
  buildHeap(A)
  heapSize = A.size
  for i = 0 to n - 1
    swap(A[0], A[n - 1 - i])
    heapSize--
    siftDown(A, 0, heapSize)
```

Операция siftDown работает за $O(\log(n))$. Всего цикл выполняется $(n-1)$ раз. Таким образом сложность сортировки кучей является $O(n\log(n))$.

Пирамидальная сортировка

Анализ пирамидальной сортировки показывает, что её сложность $O(N*\log_2 N)$

```
fun heapSort(A : list <T>):
   buildHeap(A)
   heapSize = A.size
   for i = 0 to n - 1
     swap(A[0], A[n - 1 - i])
     heapSize--
     siftDown(A, 0, heapSize)
```

Операция siftDown работает за $O(\log(n))$. Всего цикл выполняется $(n-1)$ раз. Таким образом сложность сортировки кучей является $O(n\log(n))$.

Эту сортировку (впрочем, как и все улучшенные варианты сортировок) не рекомендуется применять для небольших массивов, так как, например, для N=1000 даже прямые вставки окажутся примерно вдвое быстрее пирамидальной

ПИРАМИДАЛЬНАЯ СОРТИРОВКА

Приоритетная очередь

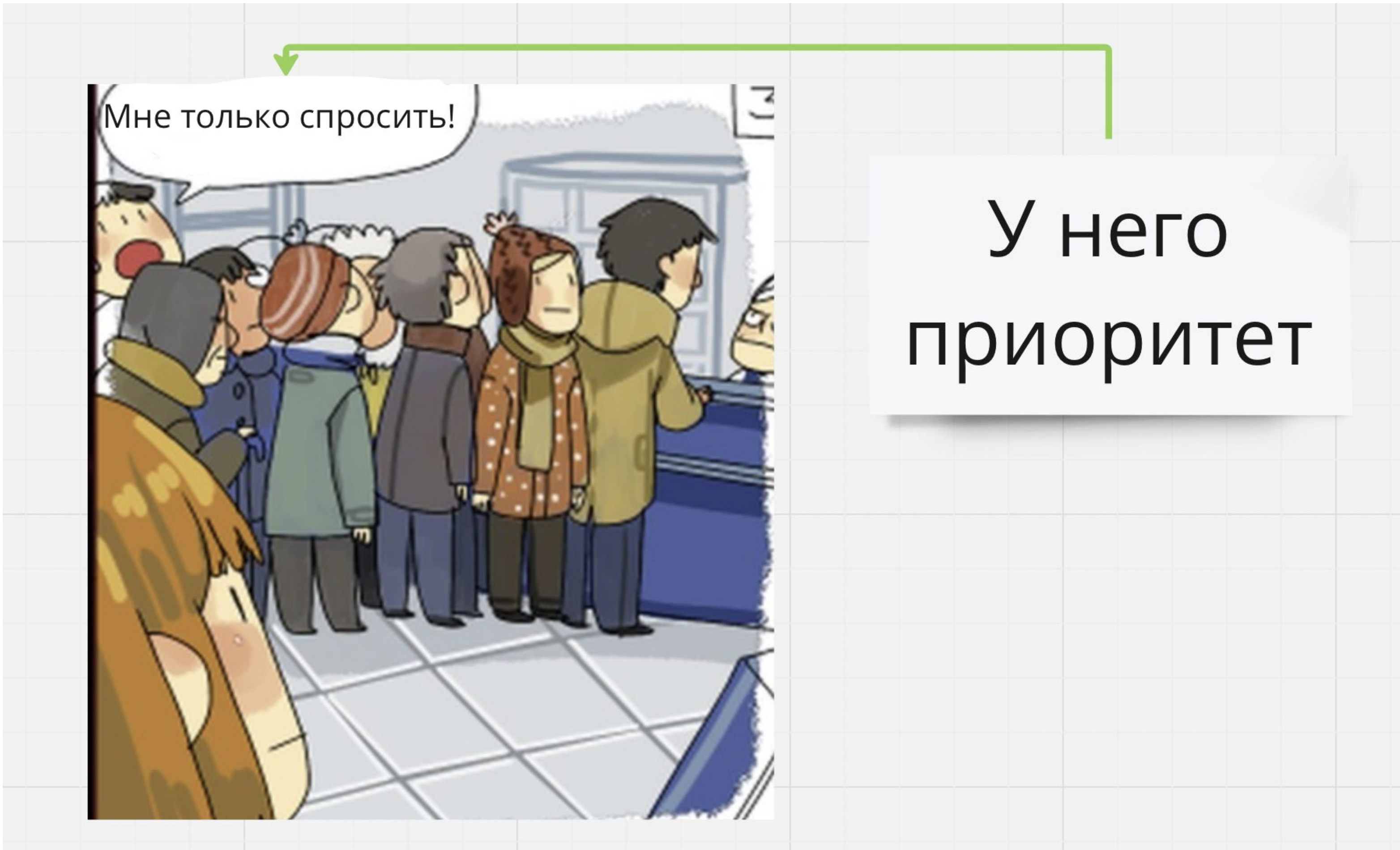
Абстрактная структура данных наподобие стека или очереди, где у каждого элемента есть приоритет.

Приоритетная очередь

Абстрактная структура данных наподобие стека или очереди, где у каждого элемента есть приоритет.

Элемент с более высоким приоритетом находится перед элементом с более низким приоритетом. Если у элементов одинаковые приоритеты, они располагаются в зависимости от своей позиции в очереди.

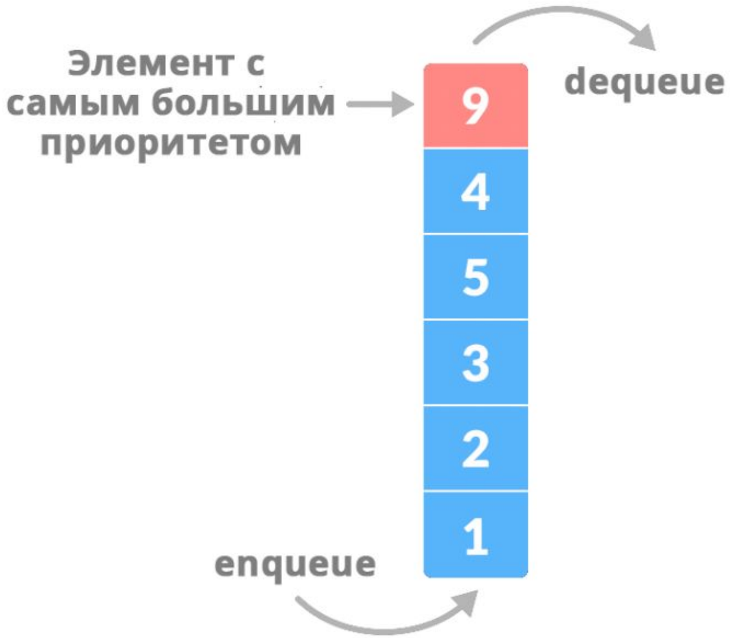
Приоритетная очередь



Приоритетная очередь

Приоритетные очереди поддерживают следующие операции:

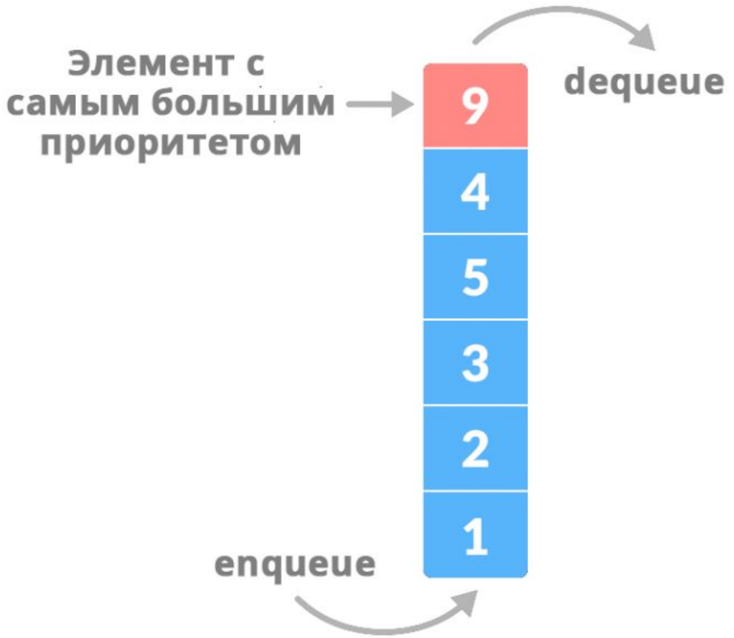
- findMin или findMax — поиск элемента с наибольшим приоритетом



Приоритетная очередь

Приоритетные очереди поддерживают следующие операции:

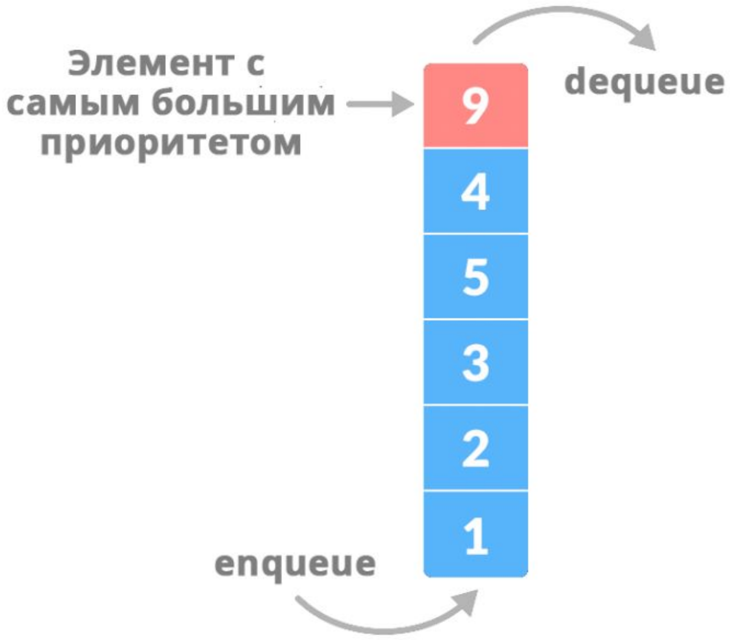
- insert или push — вставка нового элемента



Приоритетная очередь

Приоритетные очереди поддерживают следующие операции:

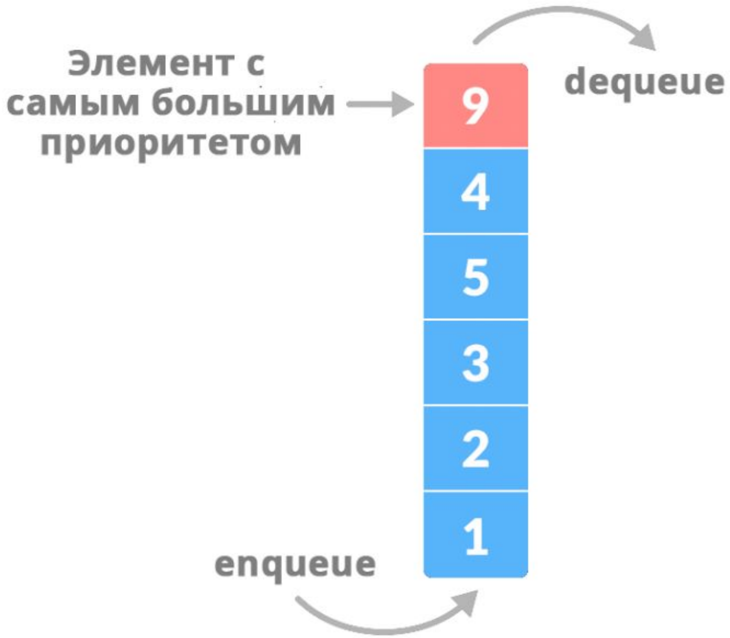
- extractMin или extractMax — извлечь элемент с наибольшим приоритетом



Приоритетная очередь

Приоритетные очереди поддерживают следующие операции:

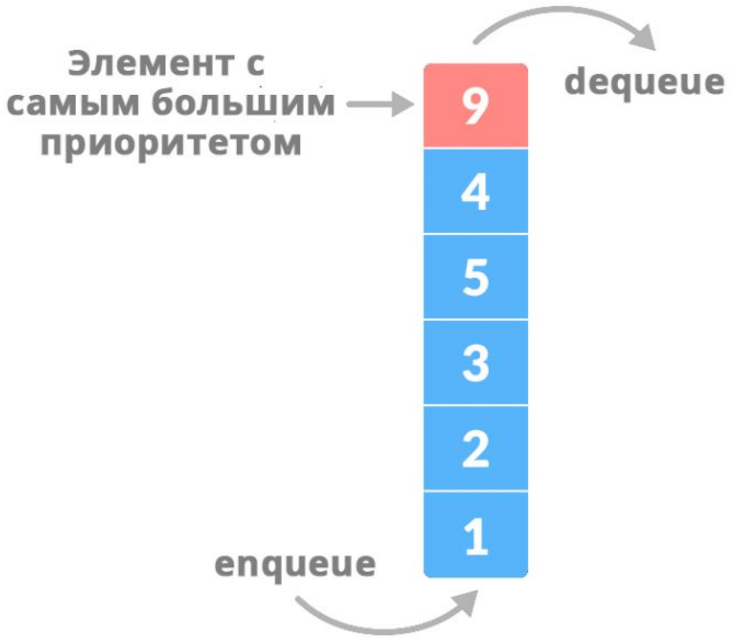
- deleteMin или deleteMax — удалить элемент с наибольшим приоритетом,



Приоритетная очередь

Приоритетные очереди поддерживают следующие операции:

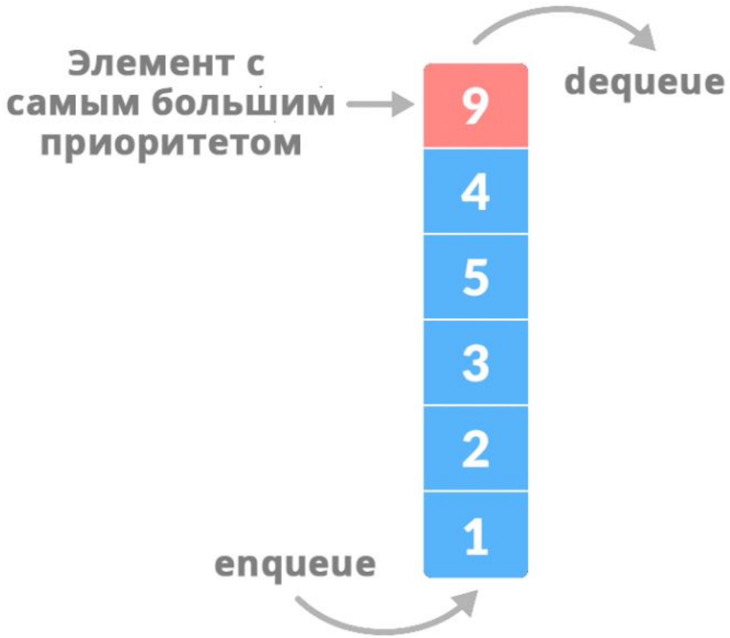
- increaseKey или decreaseKey — обновить значение элемента



Приоритетная очередь

Приоритетные очереди поддерживают следующие операции:

- findMin или findMax — поиск элемента с наибольшим приоритетом,
- insert или push — вставка нового элемента,
- extractMin или extractMax — извлечь элемент с наибольшим приоритетом,
- deleteMin или deleteMax — удалить элемент с наибольшим приоритетом,
- increaseKey или decreaseKey — обновить значение элемента



Приоритетная очередь

Почему неудобно на списке:

Чтобы вставить элемент, мы должны пройти по списку и найти правильное положение для вставки узла, чтобы сохранить общий порядок приоритетной очереди.

Это приводит к тому, что операция push() занимает O(N) времени.

Название	Операции				Описание
	insert	extractMin	decreaseKey	merge	
Наивная реализация (неотсортированный список)	$O(1)$	$O(n)$	$O(n)$	$O(1)$	Наивная реализация с использованием списка.

Приоритетная очередь

Почему неудобно на массиве:

Элемент с наивысшим приоритетом всегда приходится искать за O(n).

Необходимо выделять непрерывный участок памяти.

Название	Операции			
	insert	extractMin	decreaseKey	merge
Наивная реализация (отсортированный массив)	$O(n)$	$O(1)$	$O(\log n)$	$O(n+m)$

Приоритетная очередь

Почему удобно на куче:

Преимущество кучи заключается в поиске наименьшего элемента в куче по сложности **O(1)**

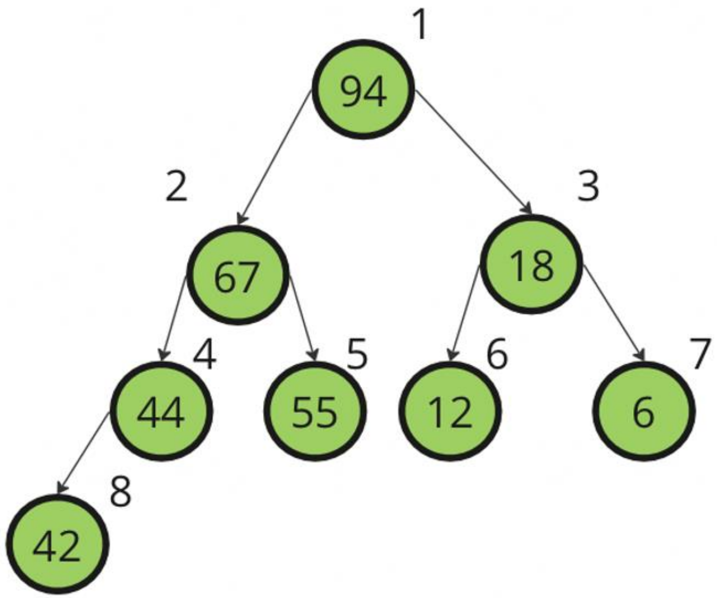
Название	Операции			
	insert	extractMin	decreaseKey	merge
Двоичная куча	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n+m)$

Асимптотика

Название	Операции			
	insert	extractMin	decreaseKey	merge
Наивная реализация (неотсортированный список)	$O(1)$	$O(n)$	$O(n)$	$O(1)$
Наивная реализация (отсортированный массив)	$O(n)$	$O(1)$	$O(\log n)$	$O(n+m)$
Двоичная куча	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n+m)$

Удаление из пирамиды

Двоичная куча



Применить операции:

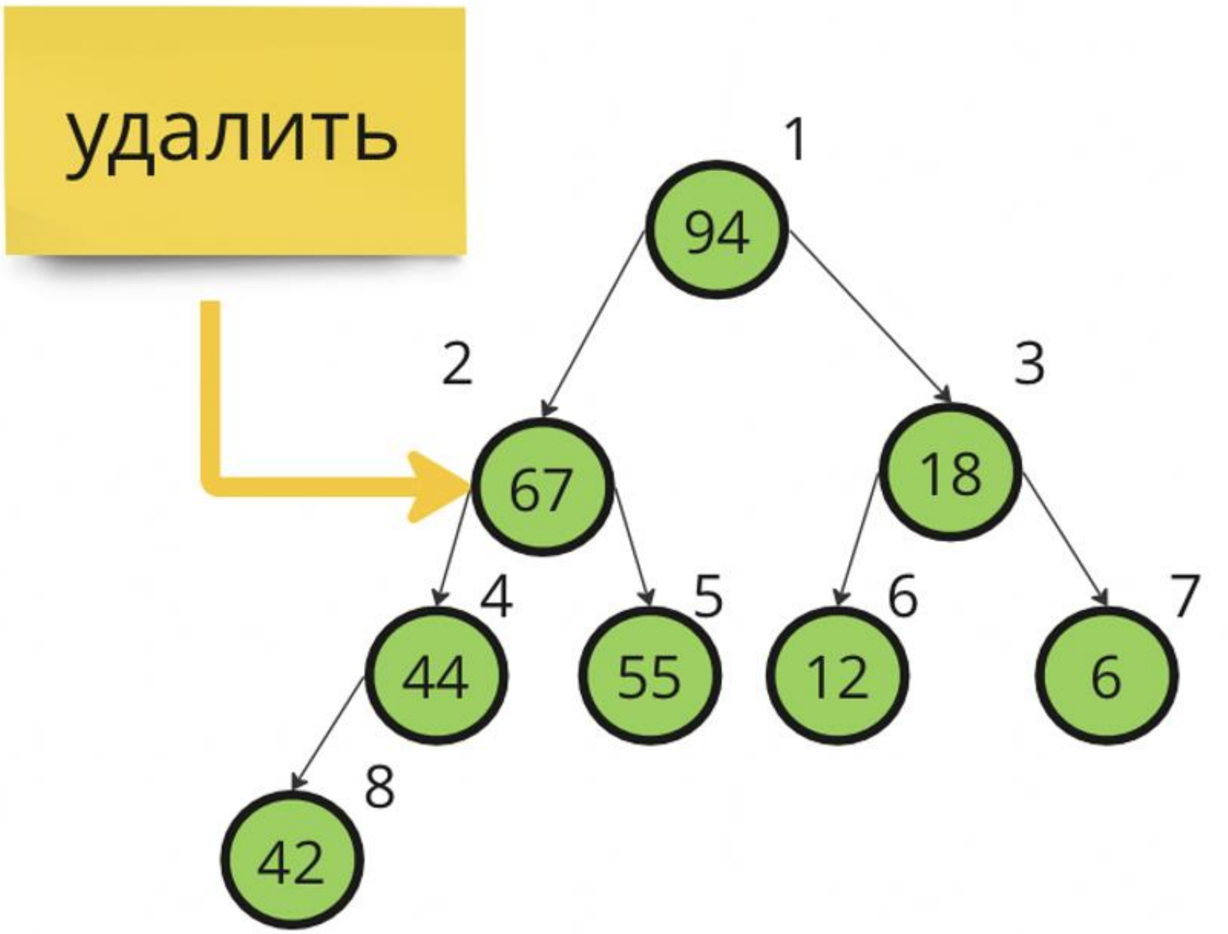
1) Уменьшить значение ключа до минимального (нужно искать минимум – это долго)
2) Увеличить значение до максимального (присвоим значение A[0] + 1 и далее поднять вверх на позицию A[0] и уже

Выполнить удаление – это долго сперва, потом A[n] просеивать вниз)

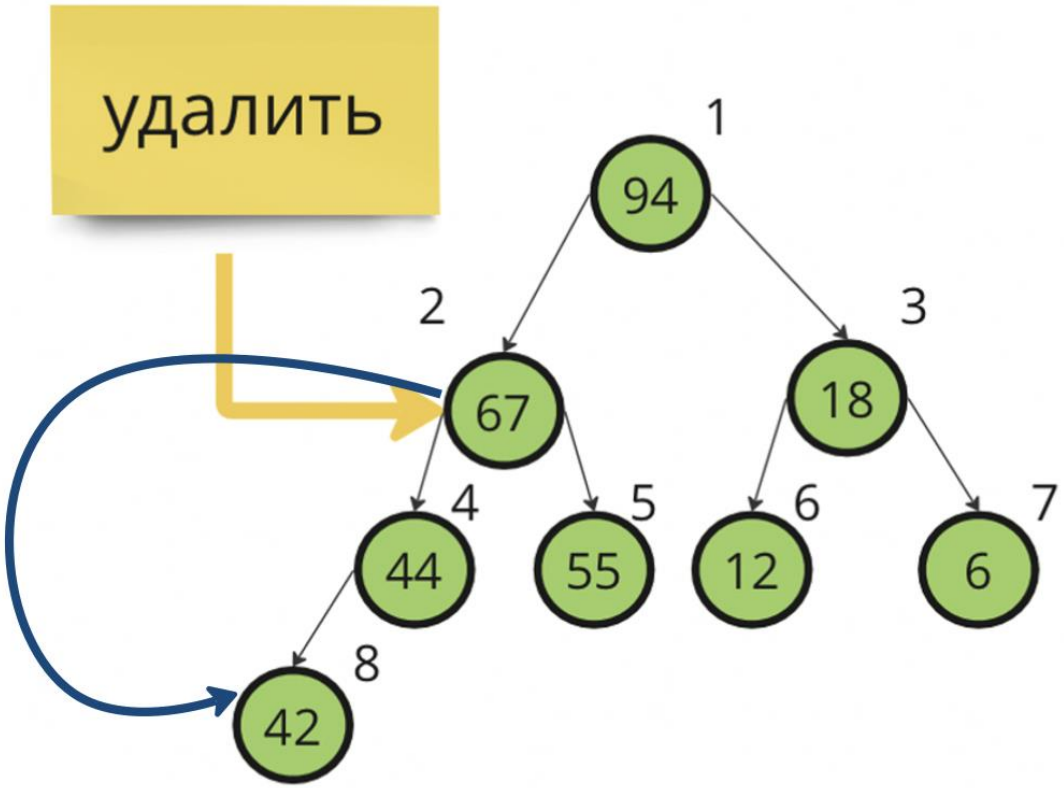
В целом мы сохранить элемент со значением ключа!

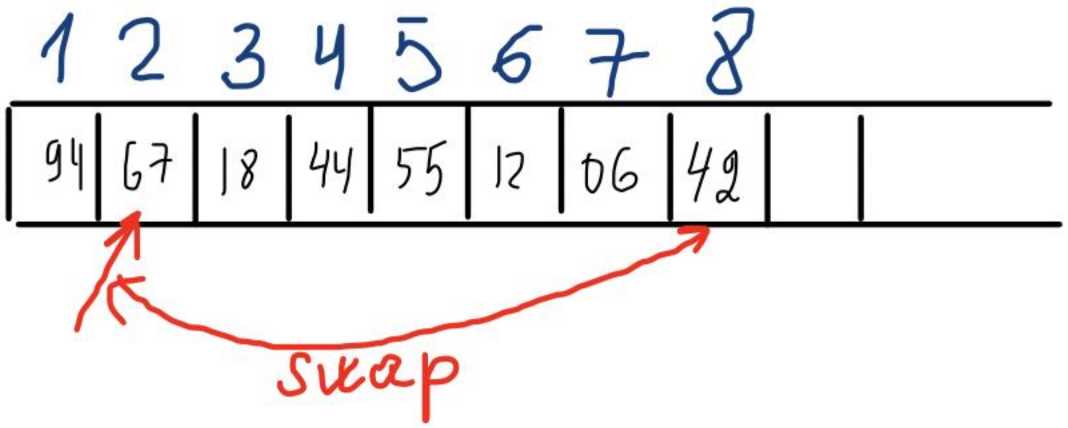
1	2	3	4	5	6	7	8		
94	67	18	44	55	12	06	42		

Удаление

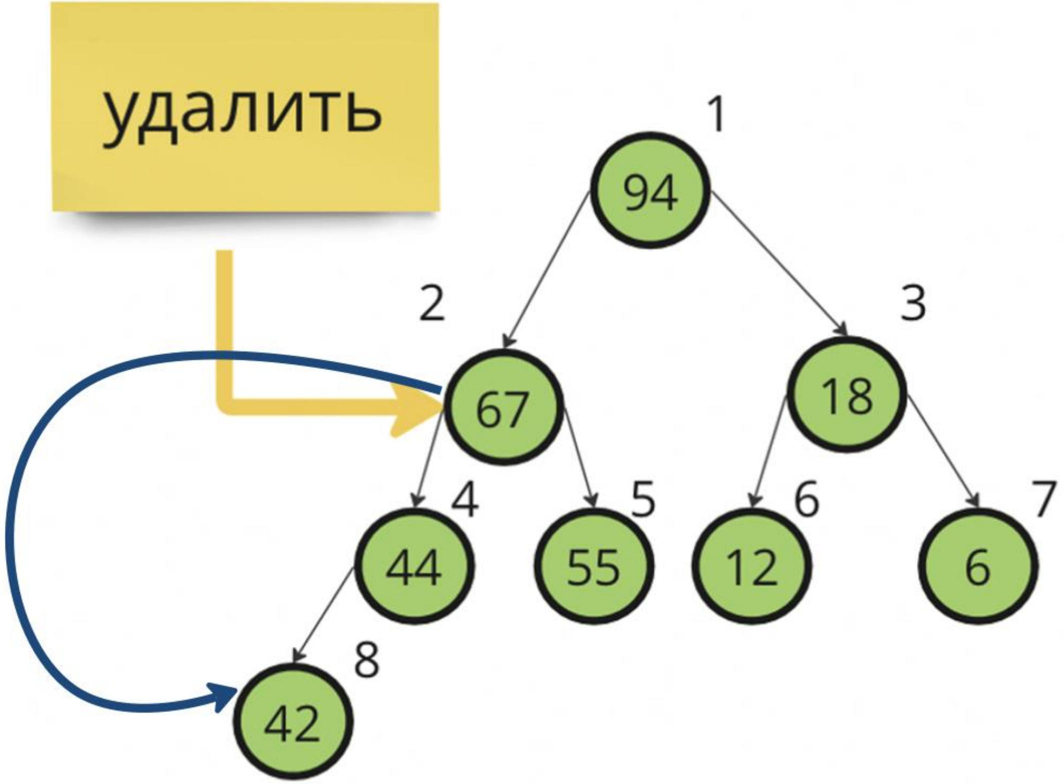


Удаление


удалить
1
94
2
67
3
18
4
44
5
55
6
12
7
6
8
42

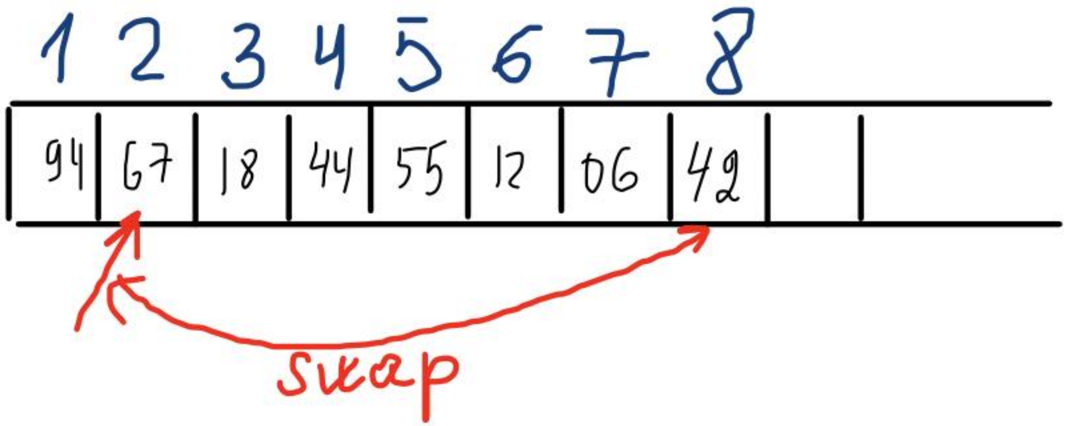

1 2 3 4 5 6 7 8
94 67 18 44 55 12 06 42
swap

Удаление

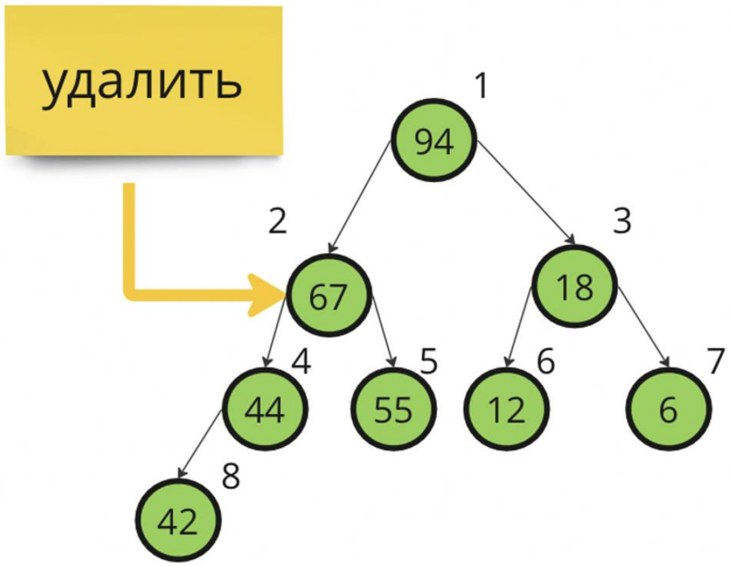


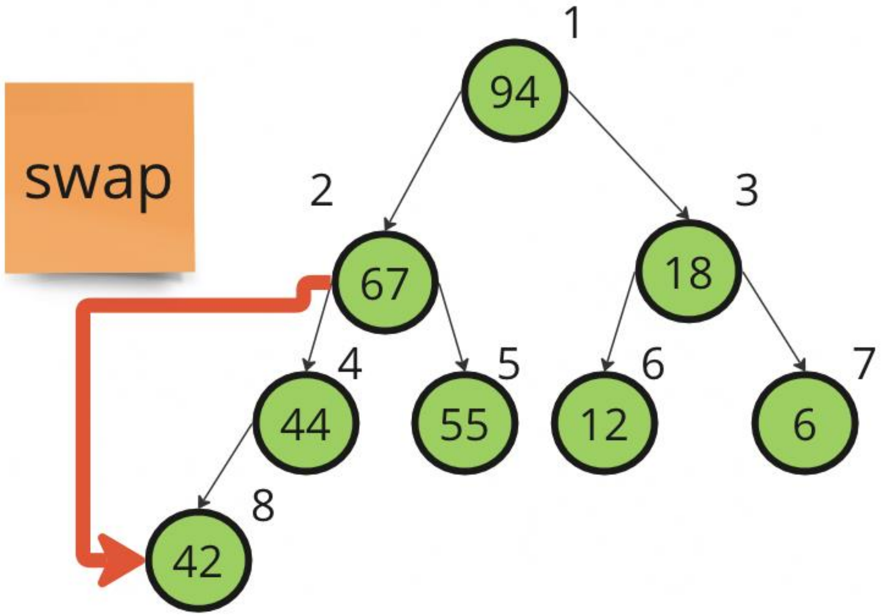
Либо :

1) Просеять вниз
2) Просеять вверх

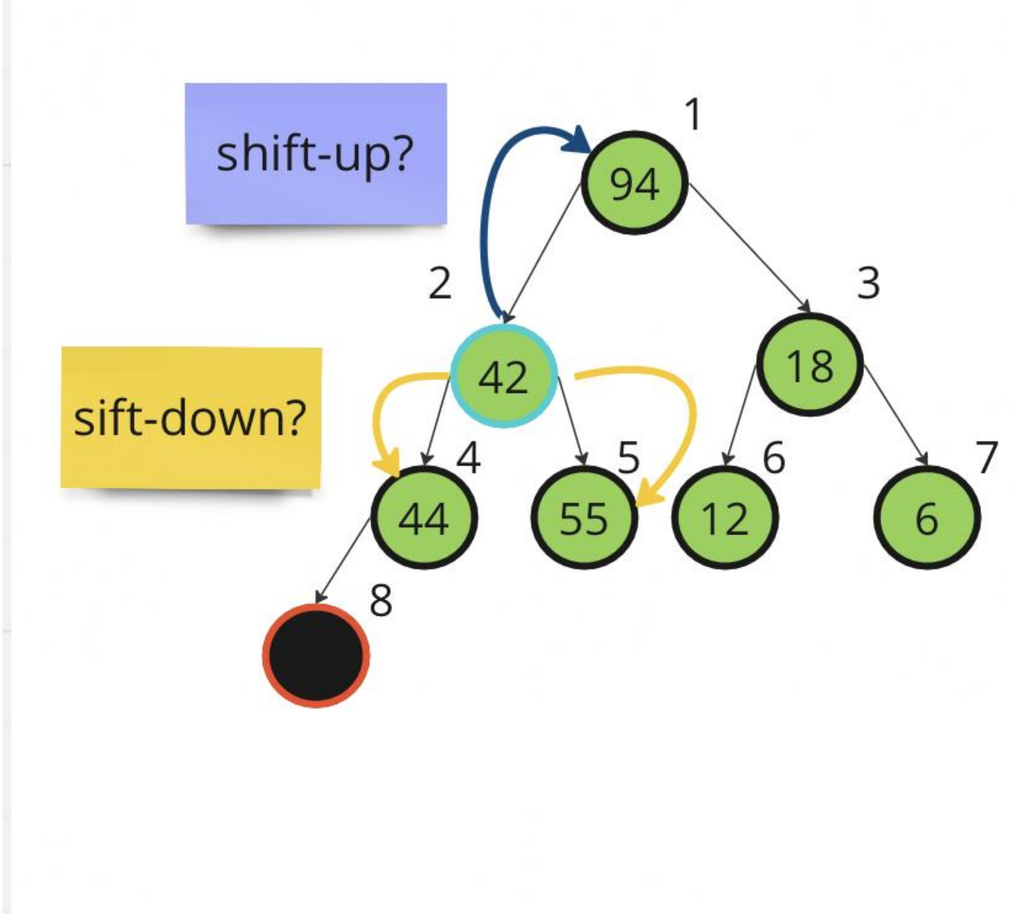


Удаление


удалить
1
94
2
3
67
18
4
5
6
7
44
55
12
6
8
42

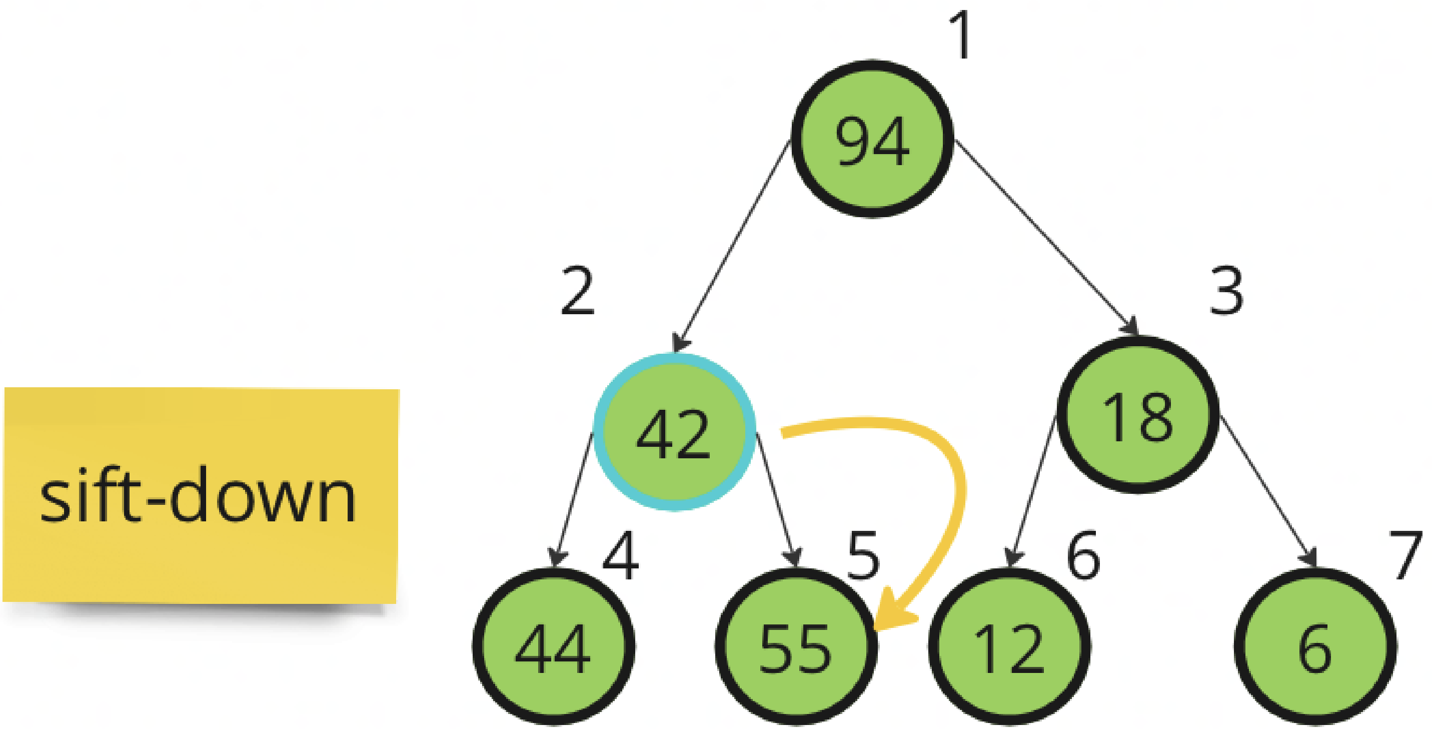

swap
1
94
2
3
67
18
4
5
6
7
44
55
12
6
8
42

Удаление



Удаление

Удаление



Удаление