

## Абстрактные типы данных

Приоритетная очередь  
(`priority queue`)

## Структуры данных

Бинарная куча (`binary heap`)

Биномиальная куча (`binomial heap`)

Куча Фибоначчи (`Fibonacci heap`)



# Абстрактные типы данных

Приоритетная очередь  
(`priority queue`)

# Приоритетная очередь (англ. priority queue)

Предположим, что для каждого элемента определён некоторый приоритет. В простейшем случае значение приоритета может совпадать со значением элемента. В общем случае соотношение элемента и приоритета может быть произвольным.

**Приоритетной очередью** называется такой абстрактный тип данных, интерфейс которого включает в себя следующие операции:

`PullHighestPriorityElement()` — поиск и удаление элемента с самым высоким приоритетом;

`InsertWithPriority(x, prior(x))` — добавление элемента  $x$  с указанным приоритетом

C++	Java	Python
контейнер-адаптер <b>std::priority_queue</b> ,  представляющий приоритетную очередь, основанную на <u>бинарной куче</u>	класс <b>PriorityQueue</b> ,  содержащий внутри <u>бинарную кучу</u>	нет абстрактного интерфейса приоритетной очереди, есть лишь модуль <b>heapq</b> ,  в котором реализована <u>бинарная куча</u>

Хотя приоритетные очереди часто ассоциируются с кучами, они концептуально отличаются от куч.

**Приоритетная очередь — это абстрактное понятие.**

По аналогии с тем, как список (list) может быть реализован с помощью связанного списка (linked list) или массива (array), приоритетная очередь (priority queue) может быть реализована с помощью кучи (heap) или другими способами (stack, queue, deque ...)



# Структуры данных

Бинарная куча (`binary heap`)

Биномиальная куча (`binomial heap`)

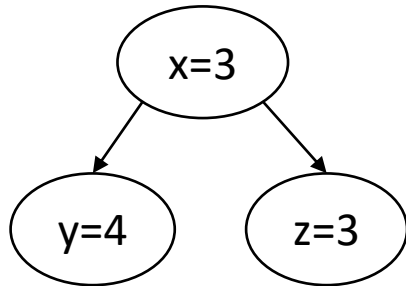
Куча Фибоначчи (`Fibonacci heap` )

**Куча** (англ. *heap*) — специализированная древовидная структура данных, которая удовлетворяет свойству кучи.

В вершинах древовидной структуры хранятся ключи.

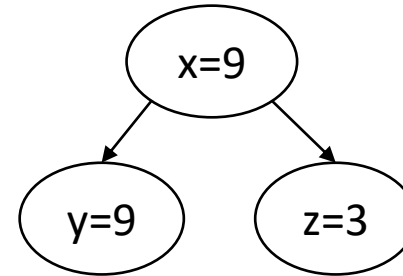
Различают два варианта куч: **min-heap** и **max-heap**.

### Свойство кучи для **min-heap**



если вершина с ключом  $y$  является потомком вершины с ключом  $x$ , то  $x \leq y$ .

### Свойство кучи для **max-heap**

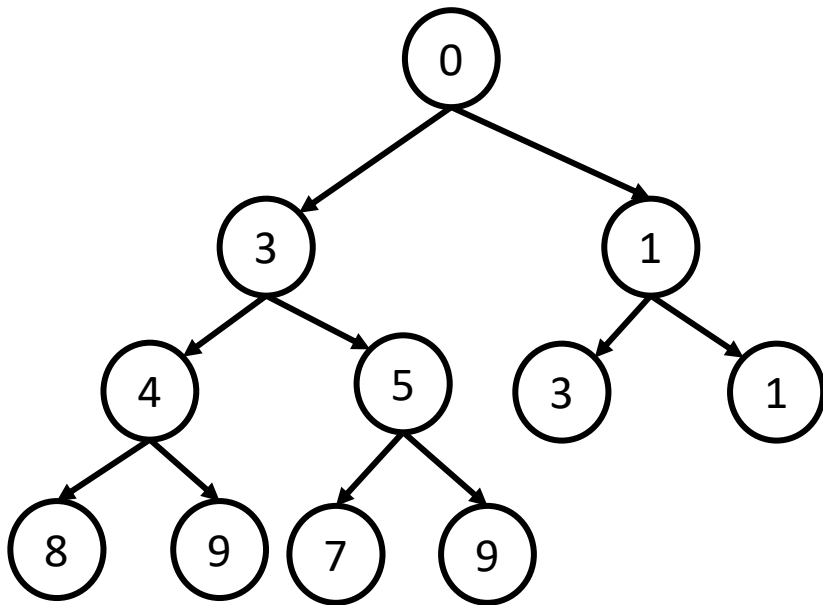


если вершина с ключом  $y$  является потомком вершины с ключом  $x$ , то  $x \geq y$ .

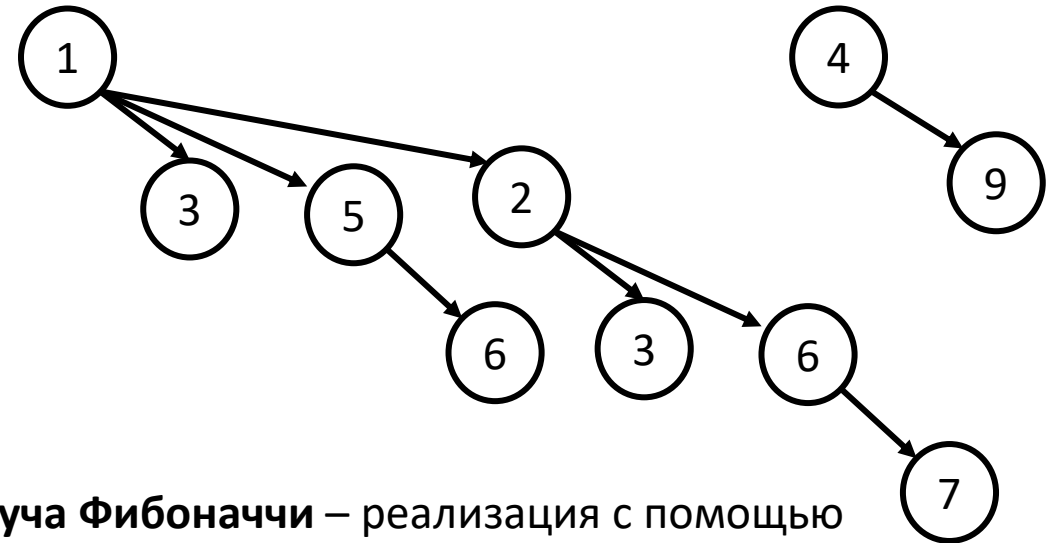
В дальнейшем, если не оговорено иное, будем считать, что при работе с кучей у нас вариант **min-heap**.

Существует много способов реализации структуры данных «куча» с помощью корневых деревьев:

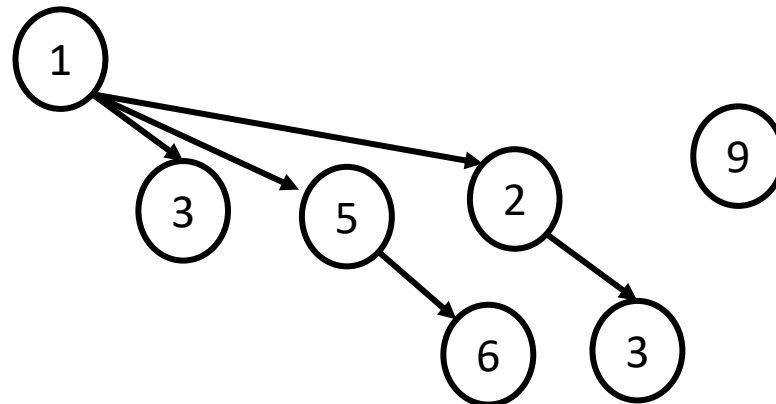
**1. Бинарная куча** (англ. *binary heap*), или **пирамида** – реализация кучи с помощью полного бинарного дерева.



**2. Биномиальная куча** – реализация кучи с помощью семейства биномиальных деревьев



**3. Куча Фибоначчи** – реализация с помощью семейства корневых деревьев



## Базовый набор операций:

**GetMin()** — поиск минимального ключа;

**ExtractMin()** — удаление минимального ключа;

**Insert(x)** — добавление ключа x.

## Расширенный набор операций:

**IncreaseKey**

**DecreaseKey**

— модификация ключа вершины на заданную величину  
(предполагается, что известна позиция вершины внутри структуры данных);

**Heapify** — построение кучи для последовательности из n ключей.



# Бинарная куча (англ. *binary heap*)

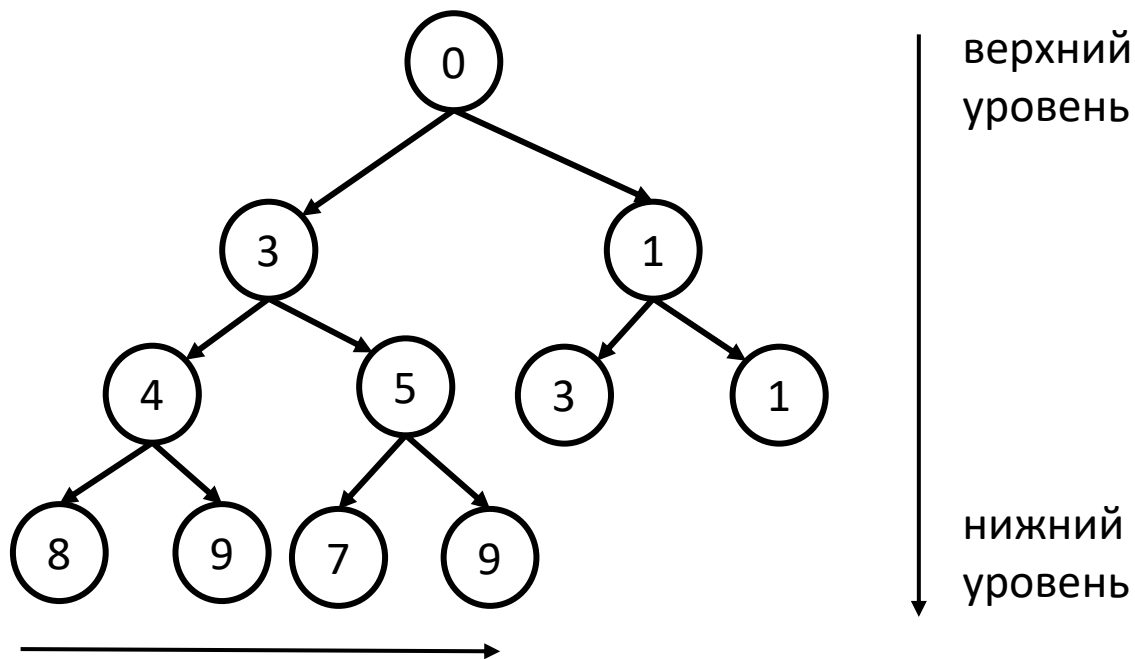
## Бинарная куча, или пирамида –

реализация кучи с помощью полного бинарного дерева.

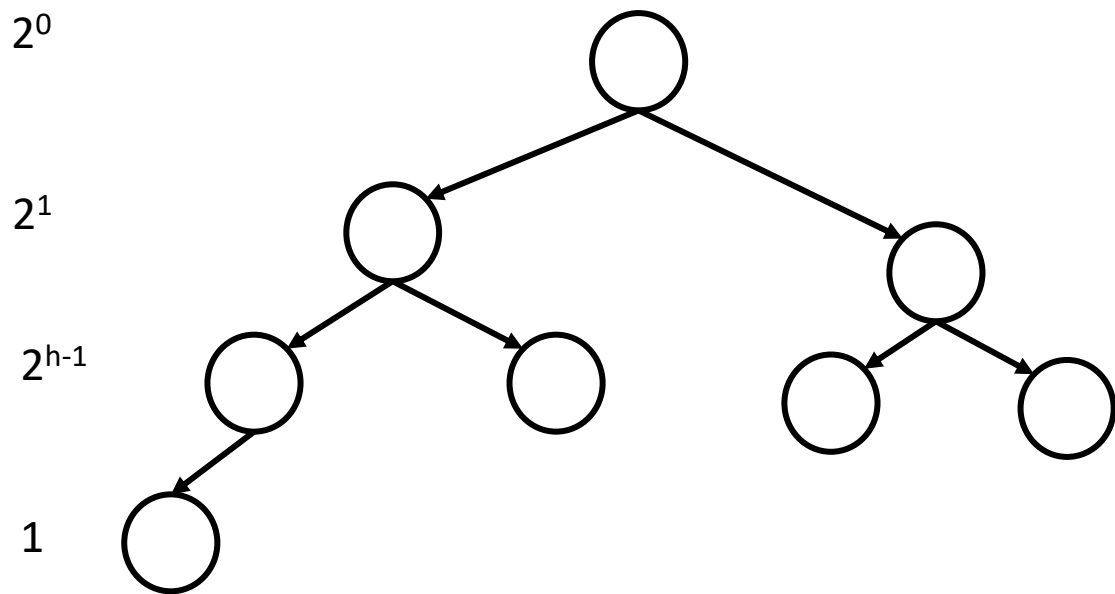
### Полное бинарное дерево –

это такое корневое дерево, в котором каждая вершина имеет не более двух сыновей, а заполнение вершин осуществляется в порядке от верхних уровней к нижним, причём на одном уровне заполнение вершинами производится слева направо. Пока уровень полностью не заполнен, к следующему уровню не переходят.

Последний уровень в полном бинарном дереве может быть заполнен не полностью.

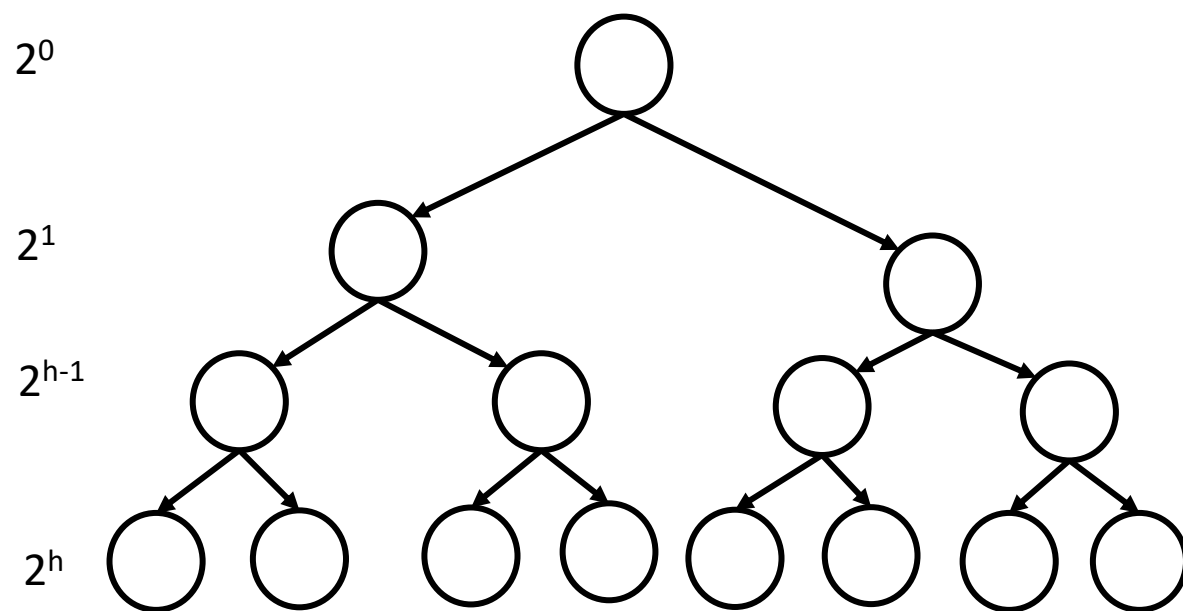


Минимальное число вершин в полном  
бинарном дереве высоты  $h$



$$2^0 + 2^1 + \dots + 2^{h-1} + 1 = \frac{2^h - 1}{2 - 1} + 1 = 2^h$$

Максимальное число вершин в  
полном бинарном дереве высоты  $h$



$$2^0 + 2^1 + \dots + 2^h = \frac{2^{h+1} - 1}{2 - 1} = 2^{h+1} - 1$$

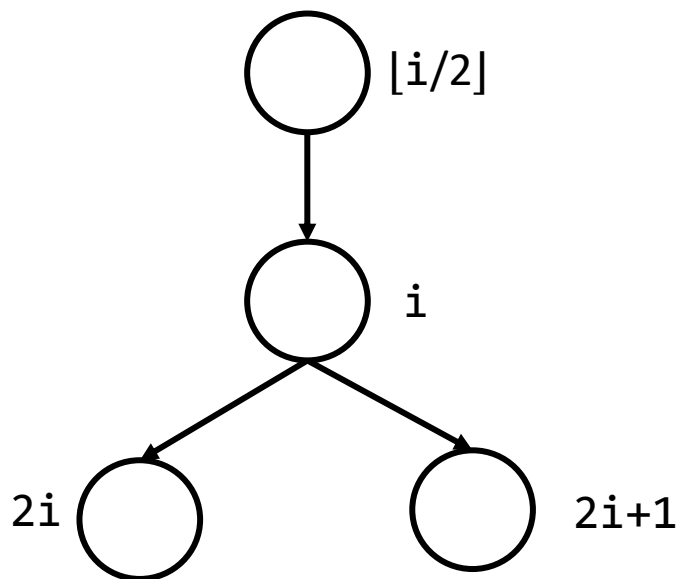
$$2^h \leq n \leq 2^{h+1} - 1$$

$$\underbrace{2^h \leq n \leq 2^{h+1} - 1}$$

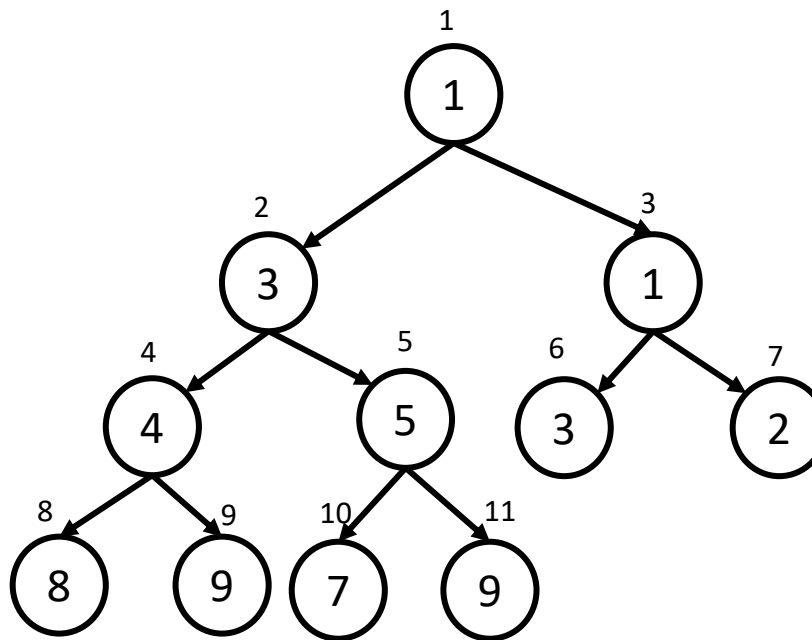
Высота  $h$  полного бинарного дерева, содержащего  $n$  вершин, —  $O(\log n)$ .

В памяти компьютера полное бинарное дерево легко реализуется с помощью массива.

Если предположить, что индексы массива начинаются с единицы, то для элемента с индексом  $i$  сыновьями являются элементы с индексами  $2i$  и  $2i + 1$ , а родителем является элемент массива по индексу  $[i/2]$ .



Пример.



В памяти компьютера бинарная куча будет храниться в массиве следующим образом:

1	2	3	4	5	6	7	8	9	10	11
1	3	1	4	5	3	2	8	9	7	9

$n=11$

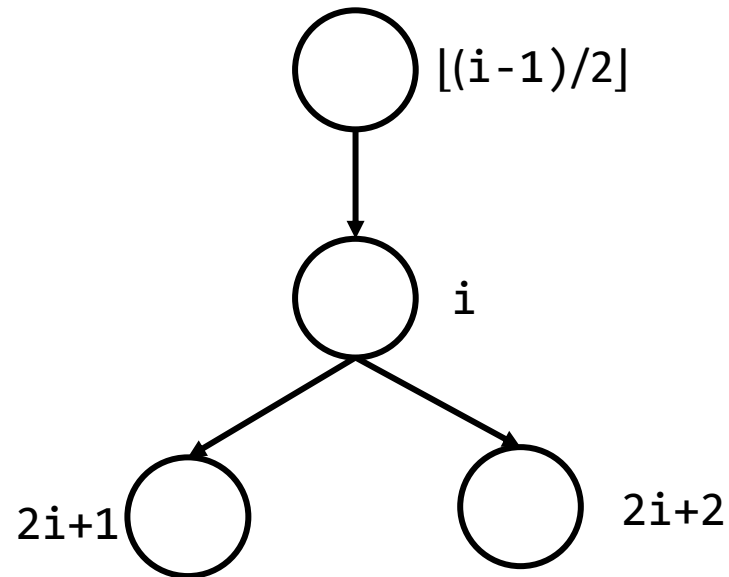
(число элементов в куче)

Если предположить, что индексы массива начинаются с нуля, то для перехода от 1-индексации к 0-индексации:

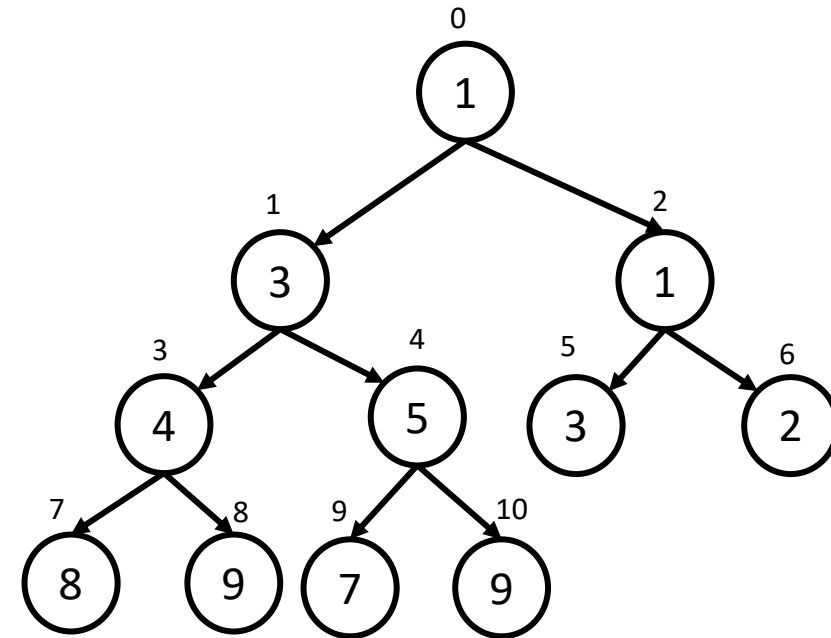
вместо  $i$  подставим  $i' = i + 1$ ,  
затем из результата вычтем 1.

**Сыновьями** элемента  $i$  являются элементы с индексами  
 $2(i+1)-1 = 2i+1$ ,  
 $[2(i+1)+1]-1 = 2i+2$ .

**Родителем** элемента  $i$  является элемент  
 $\lfloor (i+1)/2 \rfloor - 1 = \lfloor (i-1)/2 \rfloor$ .



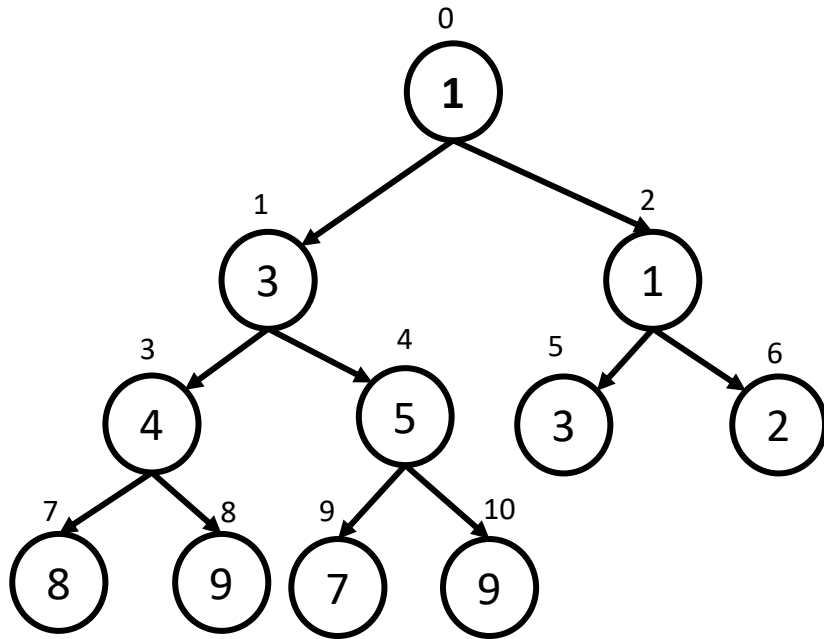
Пример.



В памяти компьютера указанное бинарная куча  
будет храниться в массиве следующим образом:

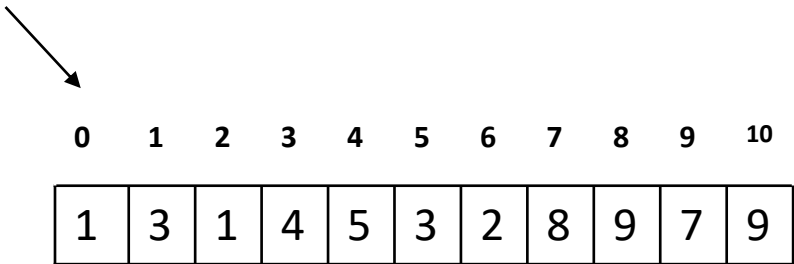
0	1	2	3	4	5	6	7	8	9	10
1	3	1	4	5	3	2	8	9	7	9

## GetMin() — поиск минимального ключа

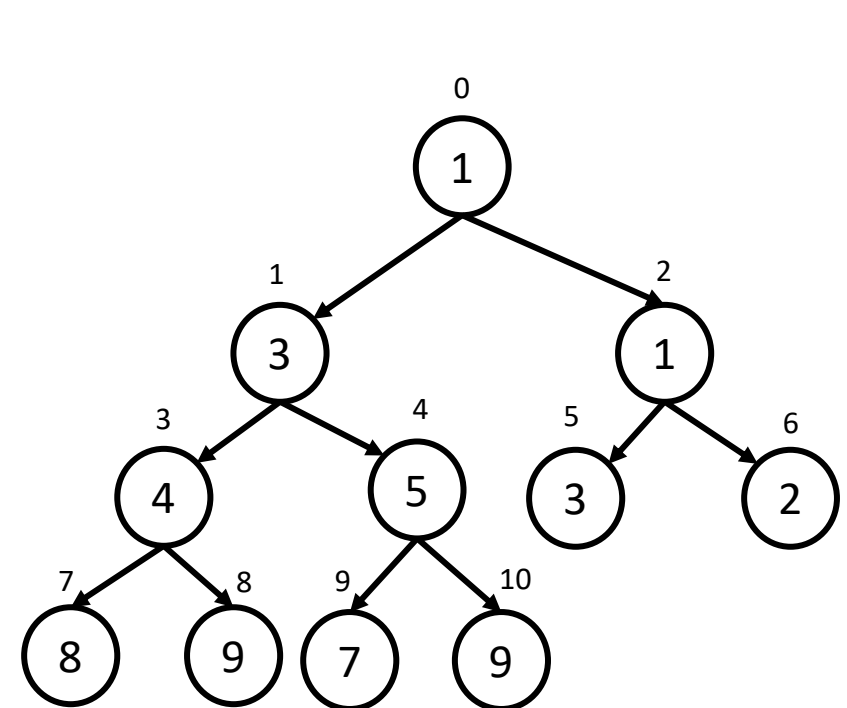


```
def GetMin(a):  
    return a[0]
```

$O(1)$

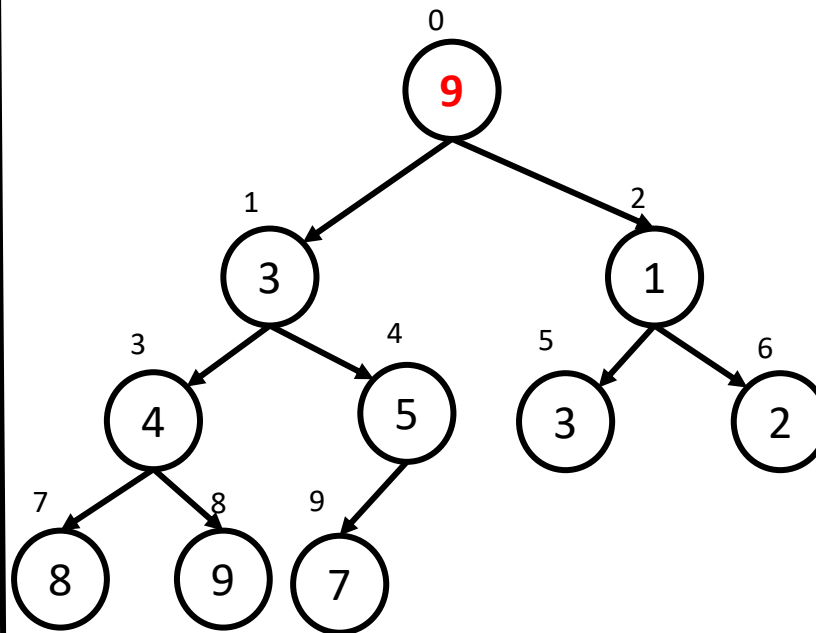


# ExtractMin() — удаление минимального ключа



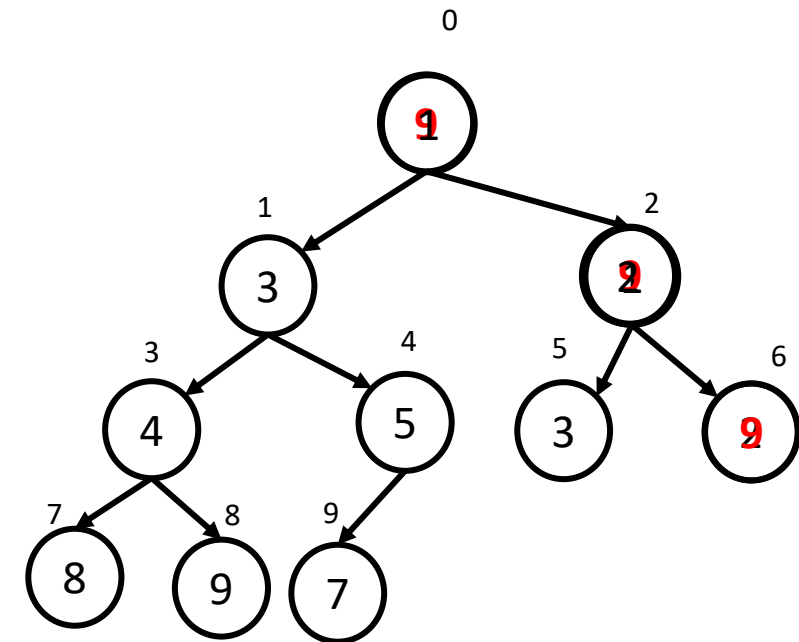
i	0	1	2	3	4	5	6	7	8	9	10
a	1	3	1	4	5	3	2	8	9	7	9

$n = 11$



0	1	2	3	4	5	6	7	8	9
9	3	1	4	5	3	2	8	9	7

$n = 10$



0	1	2	3	4	5	6	7	8	9
1	3	2	4	5	3	9	8	9	7

$n = 10$

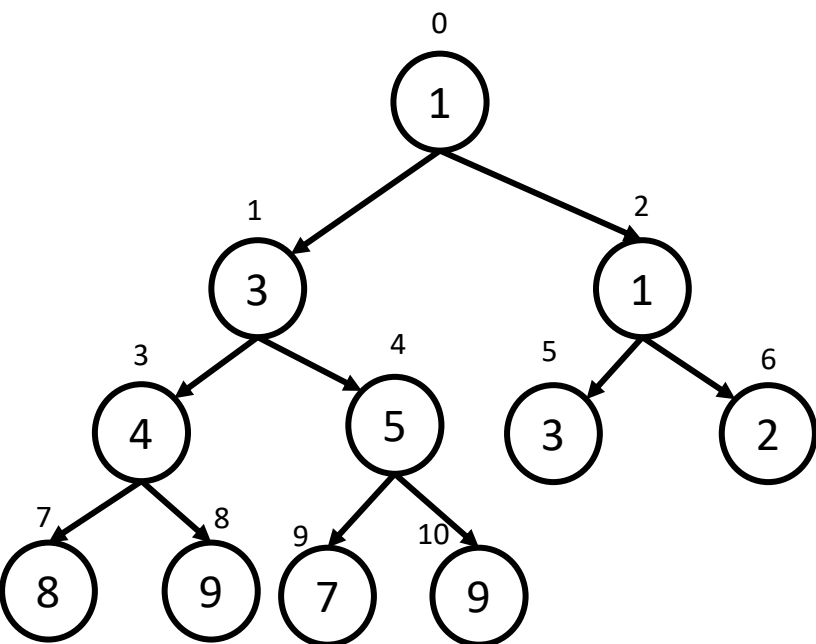
```
def ExtractMin(a):  
    a[0] = a[len(a) - 1]  
    a.pop()  
  
    i = 0  
    while 2 * i + 1 < len(a):  
        if (2 * i + 2 == len(a)) or (a[2 * i + 1] < a[2 * i + 2]):  
            j = 2 * i + 1 # left child  
        else:  
            j = 2 * i + 2 # right child  
        if a[i] <= a[j]:  
            break  
        a[i], a[j] = a[j], a[i] # swap  
        i = j
```



**ExtractMin()** — удаление минимального ключа

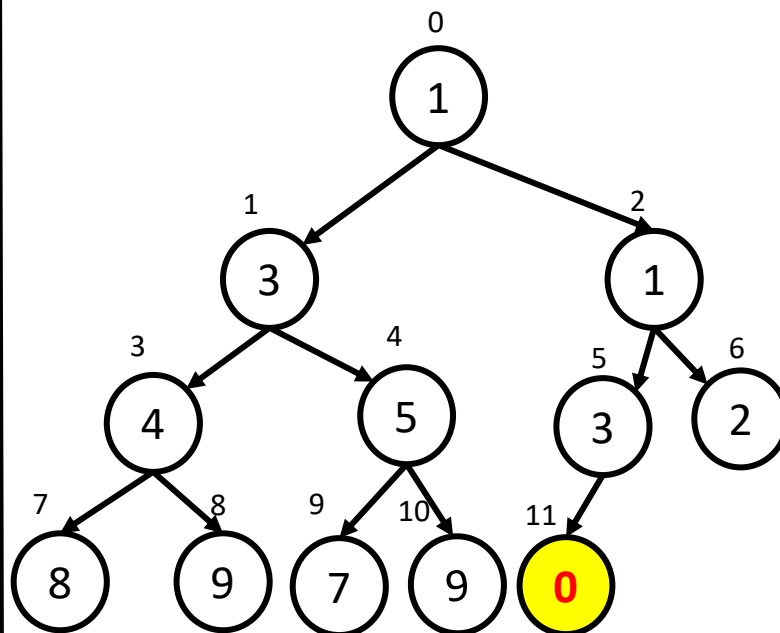
$O(\log n)$

# Insert(x) — добавление ключа x



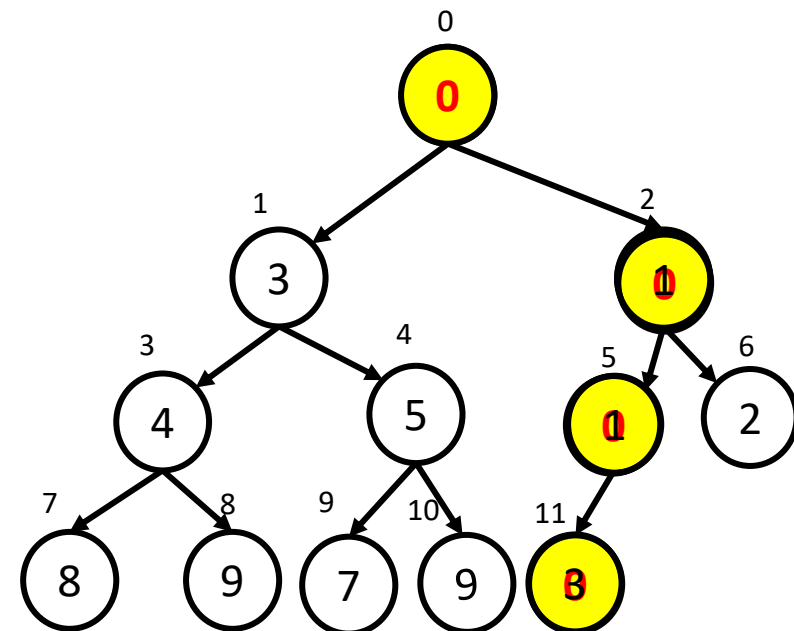
i	0	1	2	3	4	5	6	7	8	9	10
a	1	3	1	4	5	3	2	8	9	7	9

$n = 11$



0	1	2	3	4	5	6	7	8	9	10	11
1	3	1	4	5	3	2	8	9	7	9	0

$n = 12$



0	1	2	3	4	5	6	7	8	9	10	11
0	3	1	4	5	1	2	8	9	7	9	3

$n = 12$

```
def Insert(a, x):  
    a.append(x)  
  
    i = len(a) - 1  
    while i > 0:  
        j = (i - 1) // 2 # a[j] is the parent of a[i]  
        if a[j] <= a[i]:  
            break  
        a[i], a[j] = a[j], a[i] # swap  
        i = j
```

**Insert( $x$ )** — добавление ключа  $x$

$$O(\log n)$$

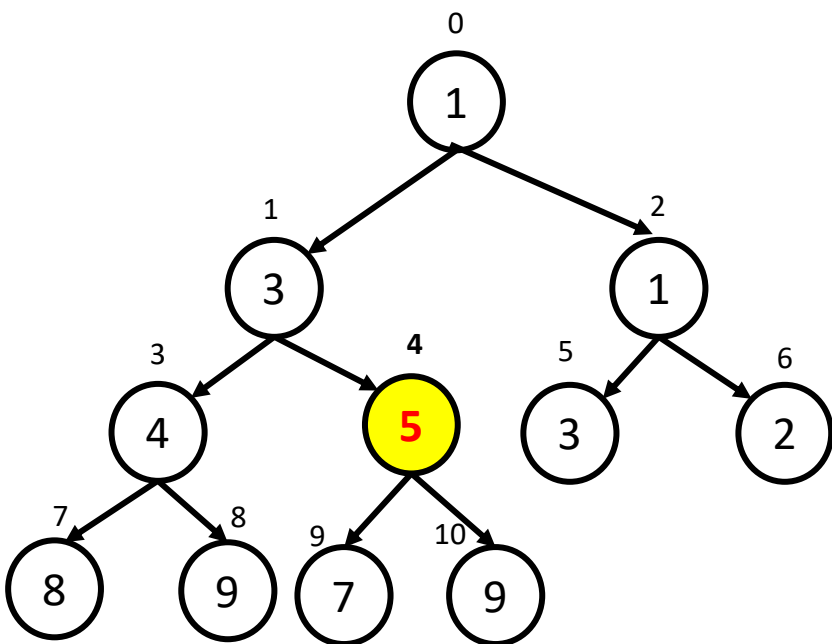
# DecreaseKey

уменьшение ключа вершины на заданную величину

(предполагается, что известна позиция вершины внутри структуры данных);

$O(\log n)$

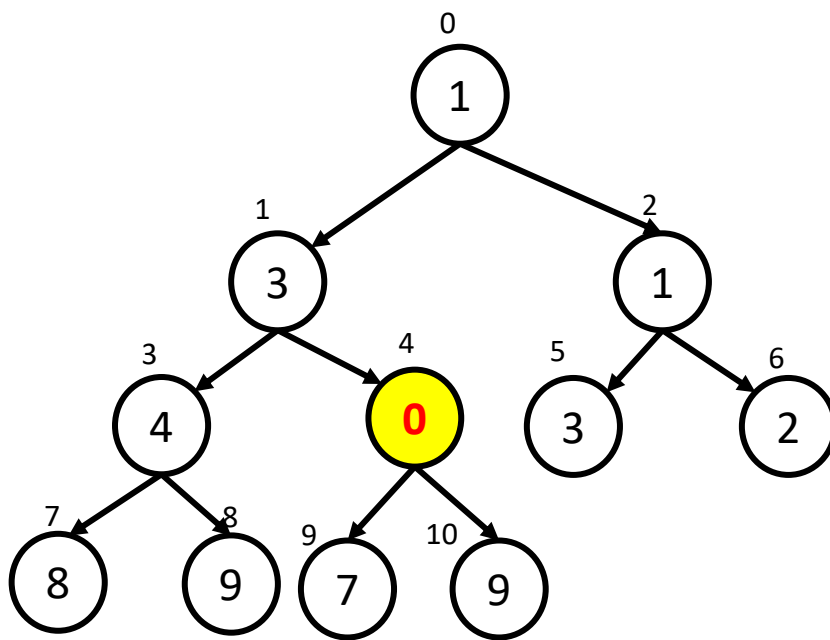
до модификации



0	1	2	3	4	5	6	7	8	9	10
1	3	1	4	5	3	2	8	9	7	9

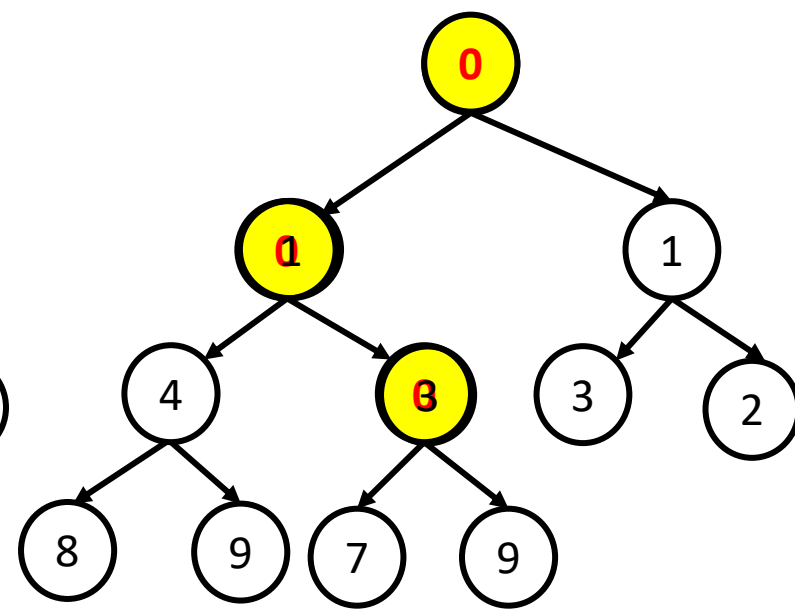
в момент модификации

(элемент по индексу 4 уменьшили на число 5)



0	1	2	3	4	5	6	7	8	9	10
1	3	1	4	0	3	2	8	9	7	9

после модификации



0	1	2	3	4	5	6	7	8	9	10
0	1	1	4	3	3	2	8	9	7	9

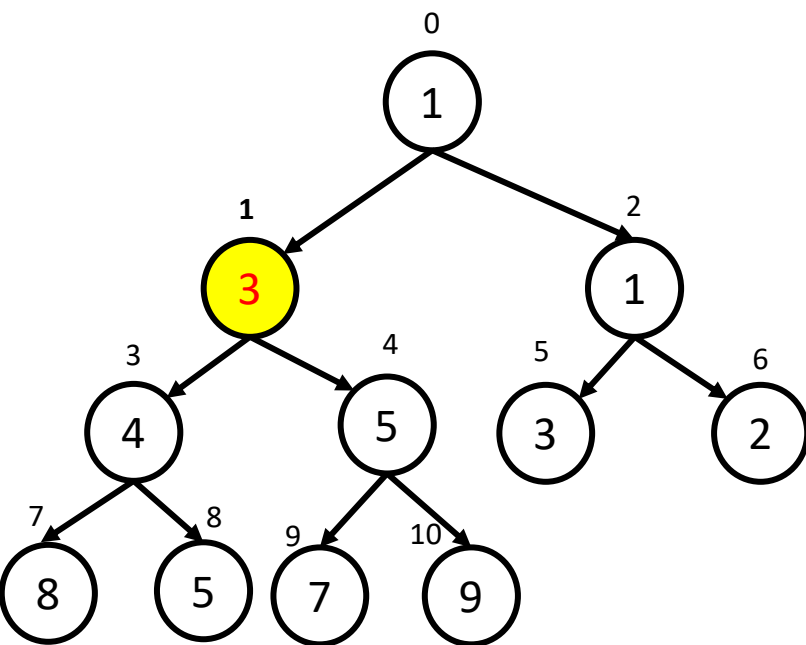
# IncreaseKey

увеличение ключа вершины на заданную величину

(предполагается, что известна позиция вершины внутри структуры данных);

$$O(\log n)$$

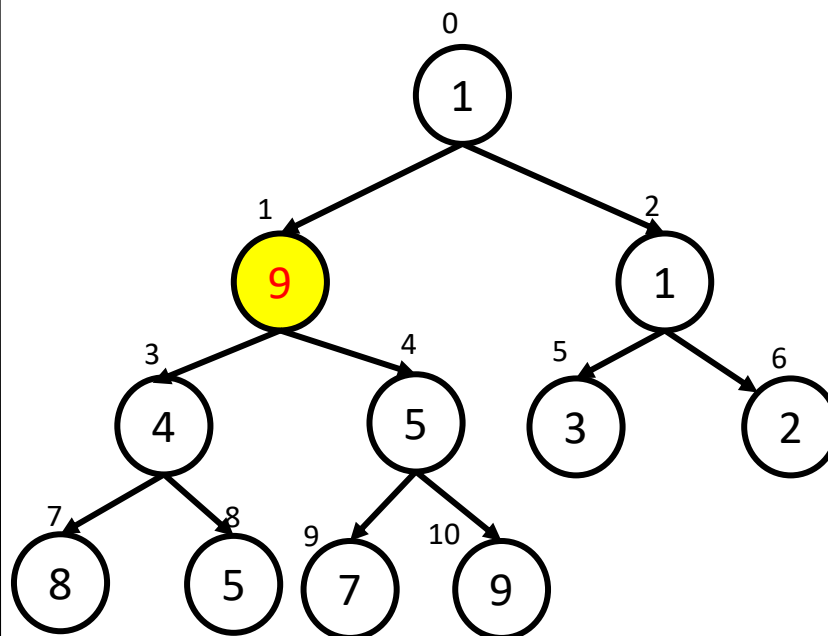
до модификации



0	1	2	3	4	5	6	7	8	9	10
1	3	1	4	5	3	2	8	5	7	9

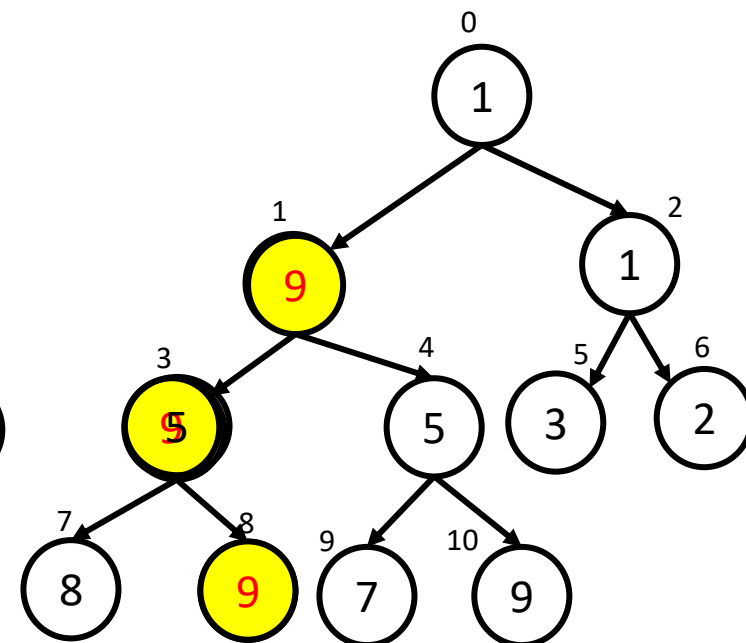
в момент модификации

(элемент по индексу 1 увеличили на число 6)



0	1	2	3	4	5	6	7	8	9	10
1	9	1	4	5	3	2	8	5	7	9

после модификации



0	1	2	3	4	5	6	7	8	9	10
1	4	1	5	5	3	2	8	9	7	9

**DecreaseKey**

**уменьшение** ключа вершины на заданную величину

**IncreaseKey**

**увеличение** ключа вершины на заданную величину

предполагается, что известна позиция вершины внутри структуры данных

$$O(\log n)$$

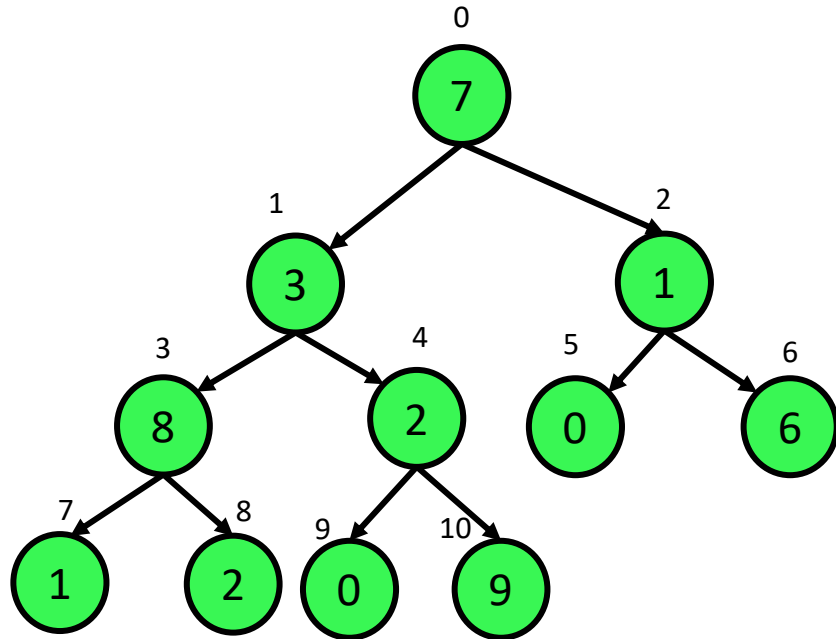
# Heapify построение кучи для последовательности из $n$ ключей.

Пример.

Построить бинарную кучу для последовательности элементов: 7,3,1,8,2,0,6,1,2,0,9

1. Строим полное бинарное дерево

$O(n)$



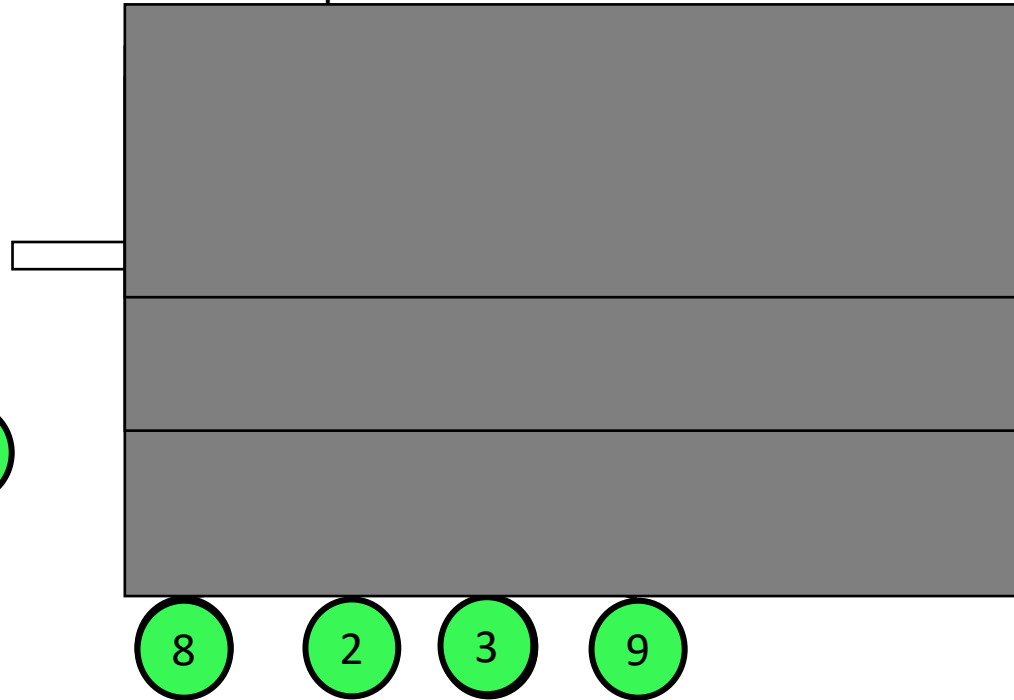
0 1 2 3 4 5 6 7 8 9 10

7	3	1	8	2	0	6	1	2	0	9
---	---	---	---	---	---	---	---	---	---	---



$n = 11$

2. Просеивание



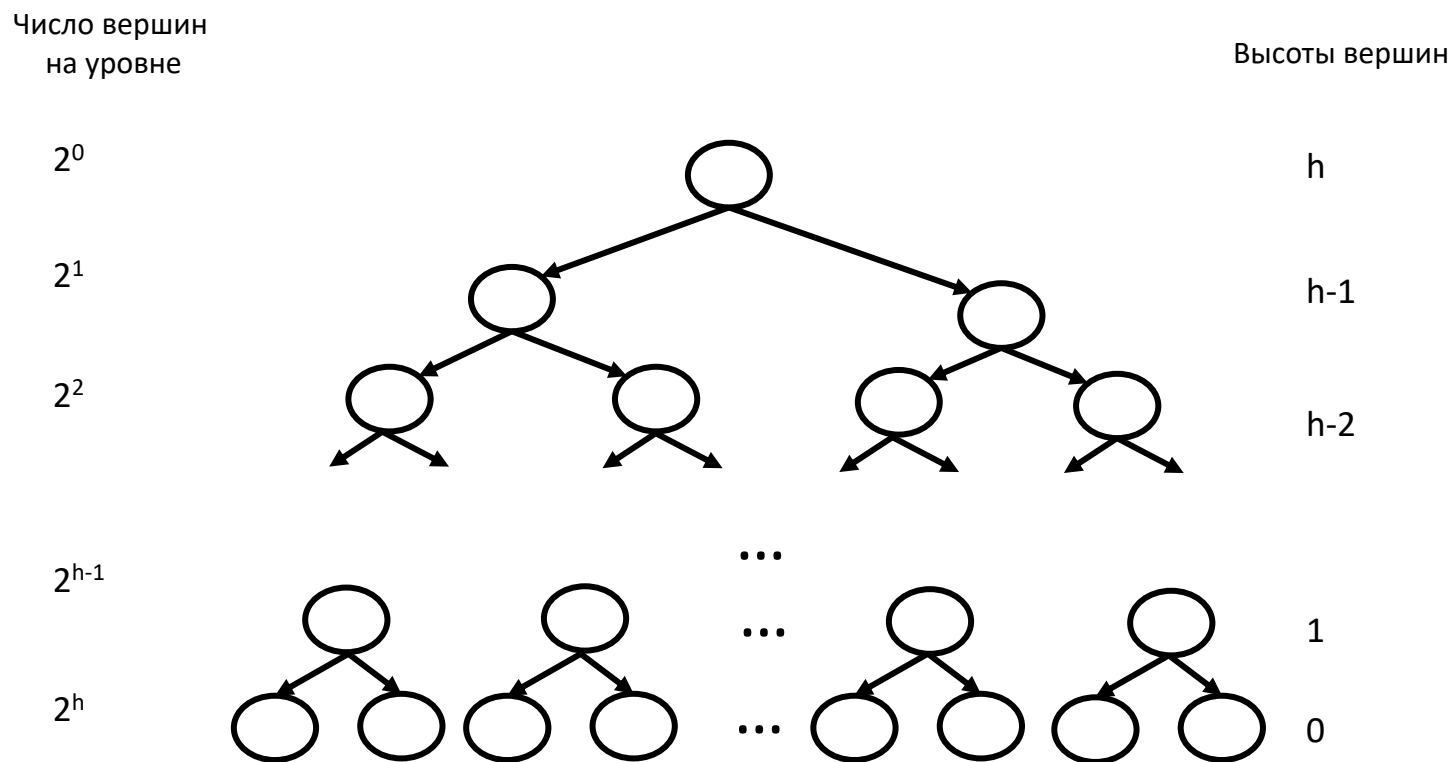
0 1 2 3 4 5 6 7 8 9 10

0	0	1	1	2	7	6	8	2	3	9
---	---	---	---	---	---	---	---	---	---	---

$n = 11$



Для того, чтобы оценить время работы построения бинарной кучи для последовательности из  $n$  элементов, необходимо оценить суммарное число всех просеиваний. Число просеиваний равно сумме высот всех вершин дерева.



$$S = 2^0 \cdot h + 2^1 \cdot (h-1) + 2^2 \cdot (h-2) + 2^3 \cdot (h-3) + 2^4 \cdot (h-4) + \dots + 2^{h-2} \cdot 2 + 2^{h-1} \cdot 1 + 2^h \cdot 0$$

$$\begin{array}{rcl}
 S & = & 2^0 \cdot h + 2^1 \cdot (h-1) + 2^2 \cdot (h-2) + 2^3 \cdot (h-3) + 2^4 \cdot (h-4) + \dots + 2^{h-2} \cdot 2 + 2^{h-1} \cdot 1 \\
 2 \cdot S & = & \phantom{2^0 \cdot h + } 2^1 \cdot h + 2^2 \cdot (h-1) + 2^3 \cdot (h-2) + 2^4 \cdot (h-3) + \dots + 2^{h-1} \cdot 2 + 2^h \cdot 1
 \end{array}$$


---

$$2S - S = -2^0 \cdot h + (2^1 + 2^2 + 2^3 + \dots + 2^{h-1} + 2^h) = -h + \frac{2 \cdot (2^h - 1)}{2 - 1} = 2^{h+1} - h - 2$$

Так как число вершин полного бинарного дерева высоты  $h$  удовлетворяет неравенствам:

$$2^h \leq n \leq 2^{h+1} - 1$$

Получаем оценку сверху на число просеиваний:

$$S = 2^{h+1} - h - 2 \leq 2 \cdot n$$

Время работы алгоритма построения бинарной кучи:  $O(n) + O(n) = O(n)$

**Heapify** построение кучи для последовательности из  $n$  ключей:

$$O(n)$$

Время выполнения базовых операций для бинарной кучи, содержащей  $n$  вершин:

### Базовый набор операций:

**GetMin()**

поиск  
минимального  
ключа;

$O(1)$

**ExtractMin()** —

удаление  
минимального  
ключа;

$O(\log n)$

**Insert(x)** —

добавление  
ключа  $x$ .

$O(\log n)$

### Расширенный набор операций:

**IncreaseKey**

**DecreaseKey**

$O(\log n)$

модификация ключа вершины на  
заданную величину  
(предполагается, что известна позиция  
вершины внутри структуры данных);

**Heapify** —

построение кучи для  
последовательности  
из  $n$  ключей.

$O(n)$

На практике бинарную кучу редко приходится реализовывать самостоятельно, поскольку готовые решения есть в стандартных библиотеках многих языков программирования. Однако важно понимать, как именно устроена эта структура данных.

C++	Java	Python
<p>контейнер-адаптер <b>std::priority_queue</b>, представляющий приоритетную очередь, основанную на бинарной куче</p> <p>Кроме того, в C++ STL доступна серия алгоритмов <b>std::make_heap</b>, <b>std::push_heap</b>, <b>std::pop_heap</b> и др.</p> <p>Эти функции позволяют построить кучу на базе любой последовательности элементов.</p>	<p>класс <b>PriorityQueue</b>, содержащий бинарную кучу</p> <p>внутри</p>	<p>нет абстрактного интерфейса приоритетной очереди, есть лишь модуль <b>heapq</b>, в котором реализована бинарная куча</p>



# **Применение на практике**

# Пирамидальная сортировка («сортировка кучей», англ. *heapsort*)

## 1. Heapify —

строим бинарную кучу для последовательности из  $n$  ключей.

## 2. Пока куча не станет пустой:

**ExtractMin()** — удаление минимального ключа;

Время работы сортировки кучей в худшем случае:

$$O(n) + O(n \cdot \log n) = O(n \cdot \log n)$$

## C++ std::sort()

Основой служит алгоритм быстрой сортировки – модифицированный QuickSort, он же **IntroSort** (*интроспективная сортировка*) разработанный специально для **STL** (1997 г., Дэвид Мюссер).

В качестве опорного элемента выбирается «**медиана и трёх**»: средний по значению элемент из первого, последнего и центрального элемента сортируемой области.

Если в сортируемом фрагменте число элементов  $< 16$ , то

фрагмент сортируется методом вставки **InsertionSort**

(сортировка вставками устойчива, работает в худшем случае за  $O(n^2)$  и для больших массивов не используется, но на малых длинах эффективна ввиду простоты реализации).

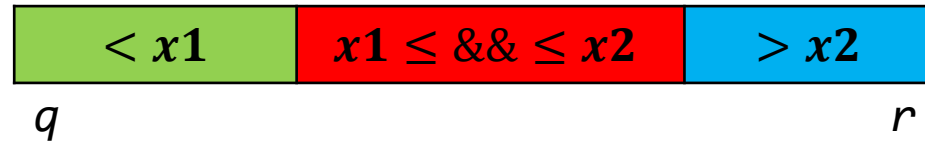

Если в сортируемом фрагменте число элементов  $\geq 16$ , то выполняется **модифицированный QuickSort**:

если глубина рекурсии превысила некоторое пороговое значение, например,  $1.5 \cdot \log_2(n)$ , где  $n$  — длина всего массива, то рекурсивные операции прекращаются и данный фрагмент сортируется пирамидальным методом **HeapSort** в чистом его виде (сортировка кучей в худшем случае работает за  $O(n \cdot \log n)$ , не устойчива).

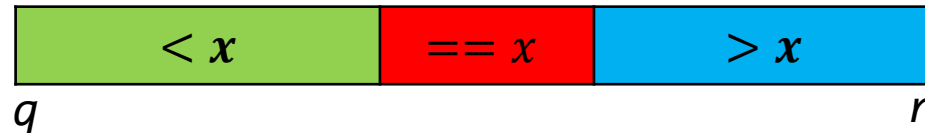


## Java `java.util.Arrays.sort()` для примитивных типов данных

Основой служит модифицированный **QuickSort** , для его реализации используется разбиение с двумя опорными элементами, которые выбираются при помощи нахождения золотого сечения, после чего выбираются 5 базовых элементов из числа которых специальным алгоритмом выбирают два опорных. Разбиение в общем виде выглядит вот так:



В случае, когда в массиве наблюдается много одинаковых элементов, то применяется Dutch Flag partitioning, Предложенный Эдсгером Дейкстрой.



Если в сортируемом фрагменте число элементов  $< 44$ , то фрагмент сортируется методом вставки **InsertionSort**

Также для ситуаций, когда глубина рекурсии превышает некоторое пороговое значение предусмотрен переход к алгоритму сортировки **HeapSort** в чистом его виде.

## Java `java.util.Arrays.sort()` для пользовательских типов данных

До Java 7 использовался алгоритм сортировки слиянием **MergeSort**, однако в дальнейшем было решено отказаться от него в пользу алгоритма сортировки **TimSort**, которая была выбрана разработчиками из-за её устойчивости, а также была оптимизирована по использованию дополнительной памяти до  $n/2$  (показывает лучшую производительность по сравнению с другими устойчивыми алгоритмами сортировки, например, таким алгоритмом, как «пузырёк»).

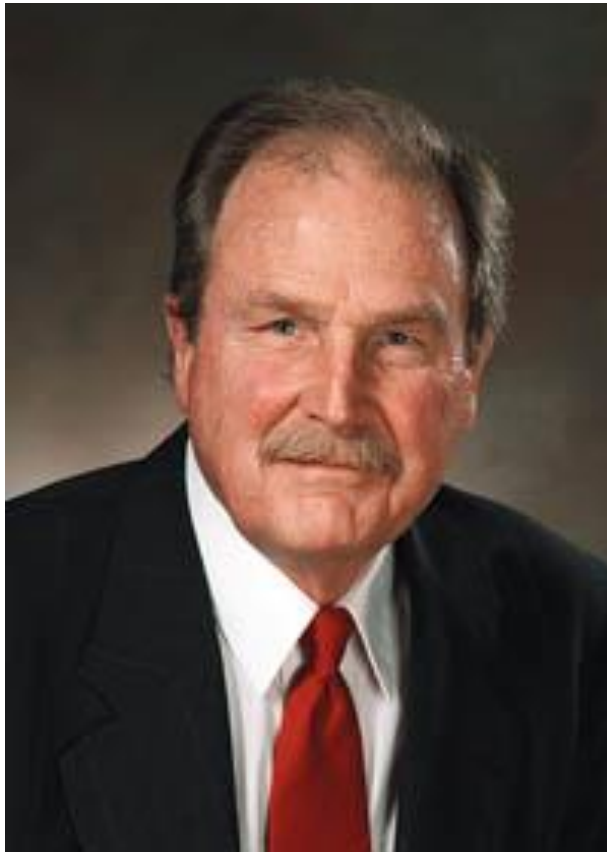
## Java java.util.Collections.sort()

Использует внутри `java.util.Arrays.sort()` для пользовательских типов данных

Галькевич Виктор, 2024 год

**Сжатие информации.  
Алгоритм префиксного кодирования Хаффмана**

Метод разработан в 1952 году  
аспирантом Массачусетского технологического института  
Дэвидом Хаффманом при написании им курсовой работы



<b>Дэвид А. Хаффман</b> <i>David Albert Huffman</i>	
Родился	9 августа 1925 г. <u>Огайо</u>
Умер	7 октября 1999 г. (74 года) <u>Санта-Крус, Калифорния</u>
Альма-матер	<u>Университет штата Огайо</u> , <u>Массачусетский технологический институт</u>
Известен	<u>Кодирование Хаффмана</u>
Награды	<u>Медаль Ричарда У. Хэмминга IEEE (1999)</u>
<b>Научная карьера</b>	
Поля	<u>Теория информации</u> , <u>Теория кодирования</u>

На вход поступает текст. По тексту строится таблица частот встречаемости символов.

Строится дерево кодирования Хаффмана (H-дерево).

По H-дереву символам текста ставится в соответствие код - последовательность бит:

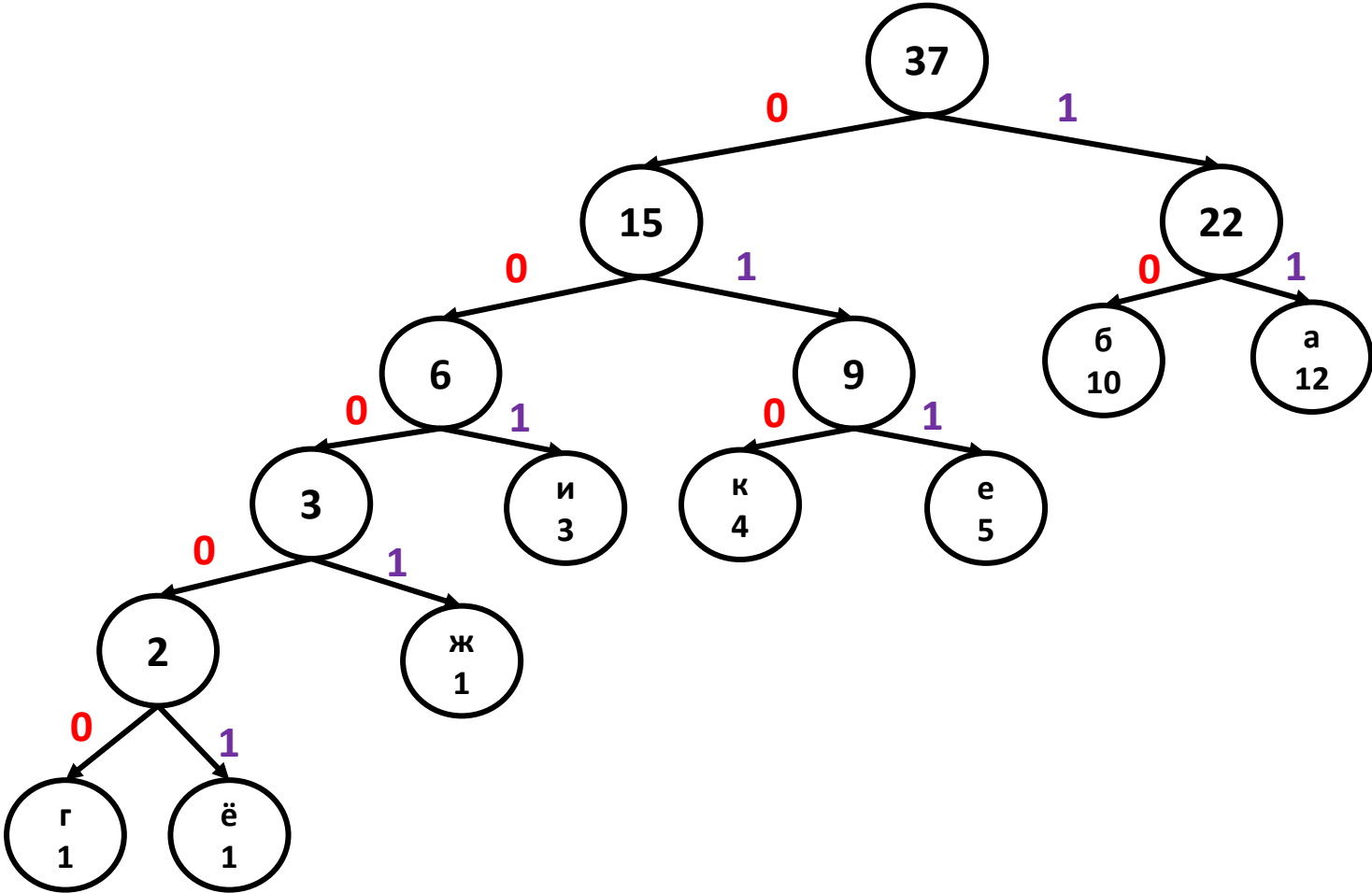
- ✓ код - **переменной длины**, т.е. символам, которые встречаются чаще, соответствует битовый код меньшей длины;
- ✓ код - **префиксный**, т.е. ни один из полученных кодов не является префиксом другого, что позволяет однозначно выполнять декодирование).

	частота	битовый код
а	12	11
б	10	10
г	1	00000
е	5	011
ё	1	00001
ж	1	0001
з	4	010
и	3	001

# Н-дерево

- 1) Каждому символу ставим в соответствие узел дерева, вес узла – частота встречаемости символа в тексте.
- 2) Полагаем все узлы - свободными.
- 3) Пока не останется 1 свободный узел, выполняем следующие действия:
  - ✓ находим 2 свободных узла  $v$  и  $w$  с минимальным весом и исключаем их из множества свободных узлов;
  - ✓ формируем новый свободный узел  $r$ , полагая  $v$  и  $w$  сыновьями  $r$ ;
  - ✓ вес узла  $r$  определяем как сумму весов  $v$  и  $w$ .
- 4) Обходим дерево, ставя метки дугам дерева «0» или «1» (например, «0» – левому сыну, а «1» – правому).

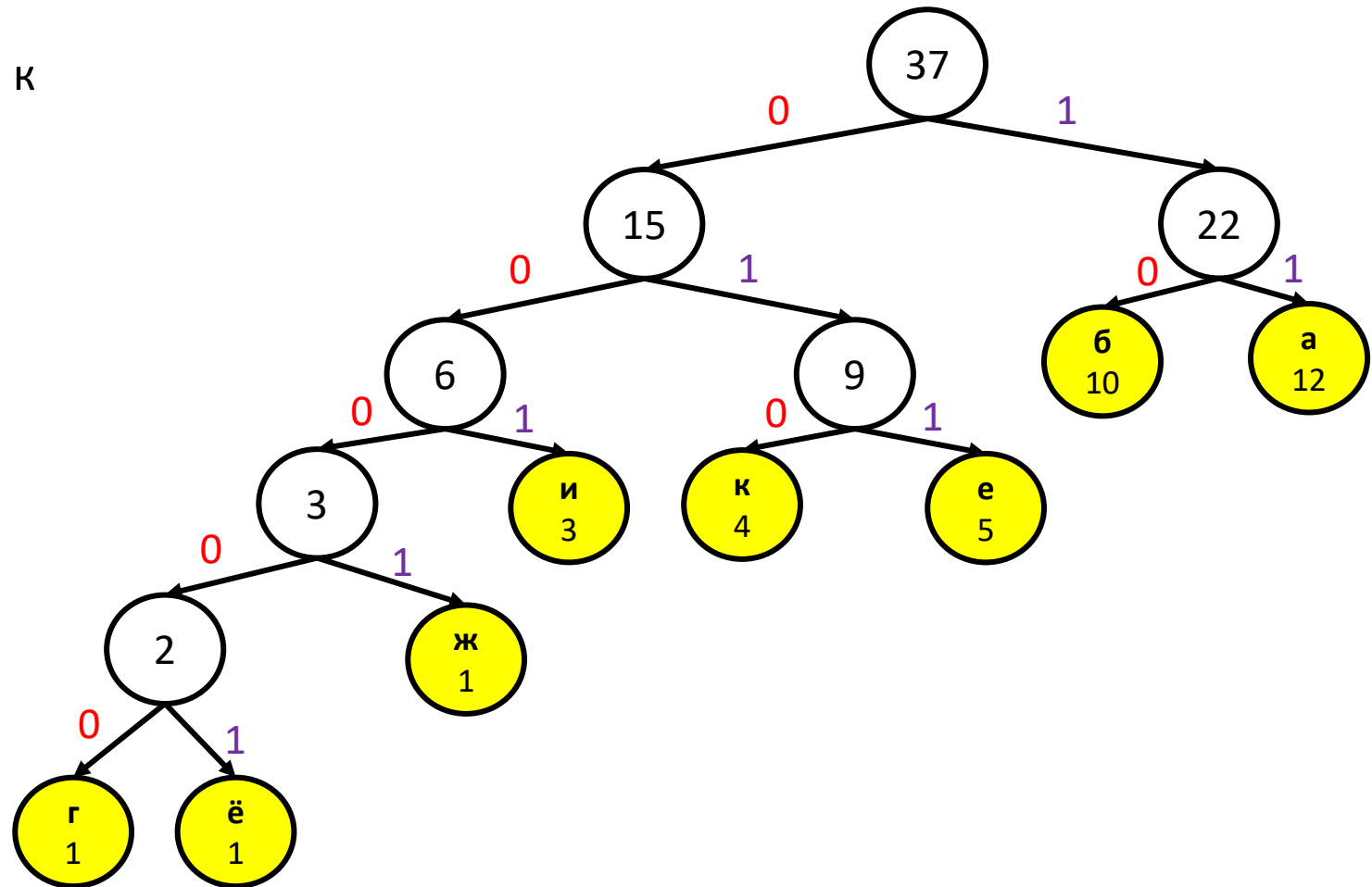
# Н-дерево



	частота
а	12
б	10
г	1
е	5
ё	1
ж	1
к	4
и	3

**Битовый код символа –**  
строка бит на пути от корня к  
этому символу.

	частота	битовый код
а	12	11
б	10	10
г	1	00000
е	5	011
ё	1	00001
ж	1	0001
к	4	010
и	3	001



## Кодирование:

Текст :

**кажжекаа ...**

Закодированный текст:

(010)(11 )( 0001)(0001)(011)(010)(11)(11)  
к а ж ж е к а а

	частота	битовый код
а	12	11
б	10	10
г	1	00000
е	5	011
ё	1	00001
ж	1	0001
к	4	010
и	3	001

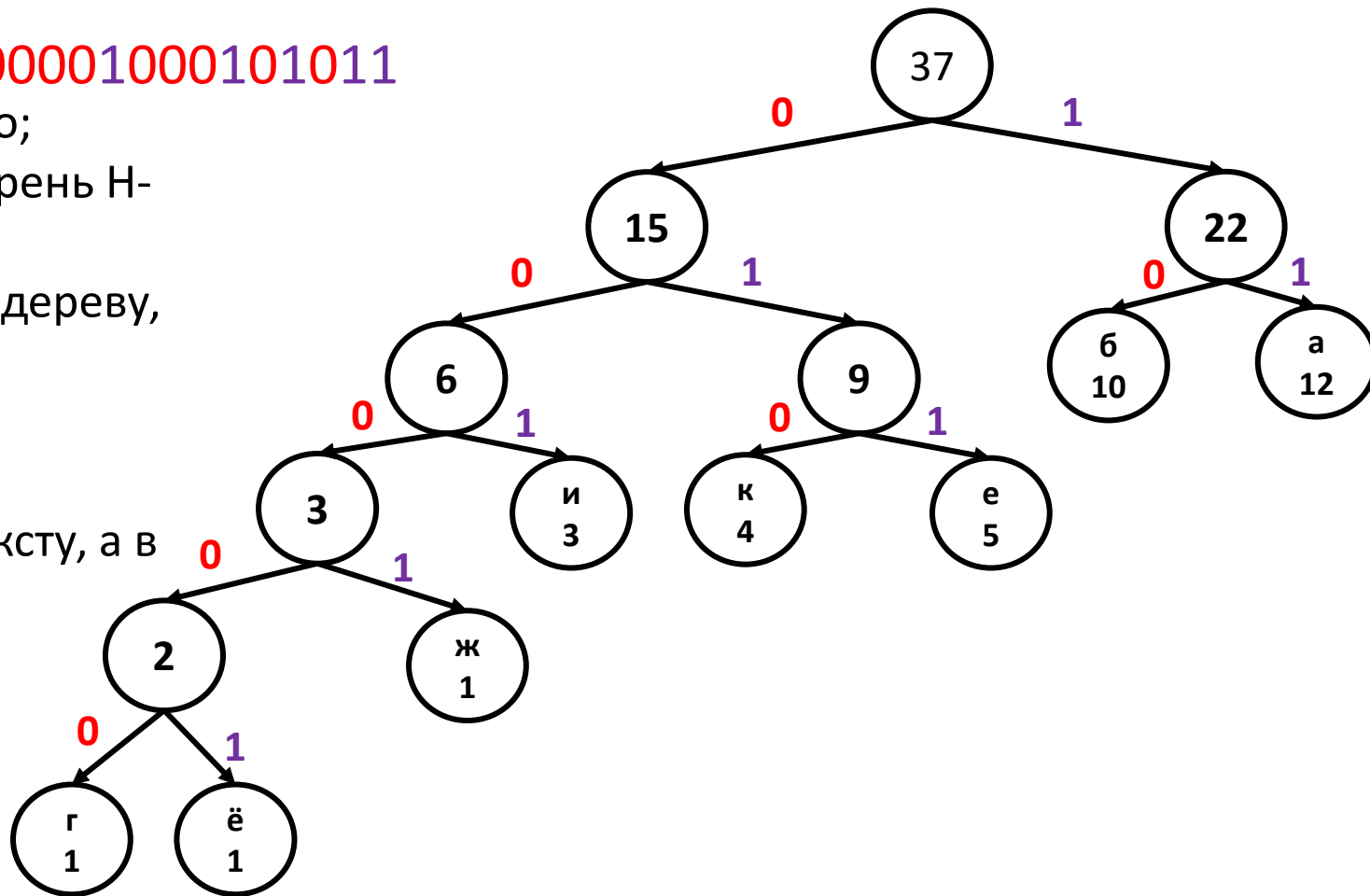


## Декодирование: 1011100101100001000101011

для декодирования требуется Н-дерево;

- ✓ становимся на начало текста и в корень Н-дерева;
- ✓ двигаемся параллельно по тексту и дереву, пока не дойдём до листа дерева;
- ✓ выписываем символ, который соответствует листу;
- ✓ продолжаем далее движение по тексту, а в дереве становимся снова в корень;

Что закодировано в сообщении?



## ЗАДАЧА

На вход поступает таблица частот встречаемости символов текста, который будет закодирован классическим алгоритмом Хаффмана.

Вам дали эту таблицу, упорядочив символы в соответствии с их частотой встречаемости (сначала идут символы, которые реже всего встречаются в тексте).

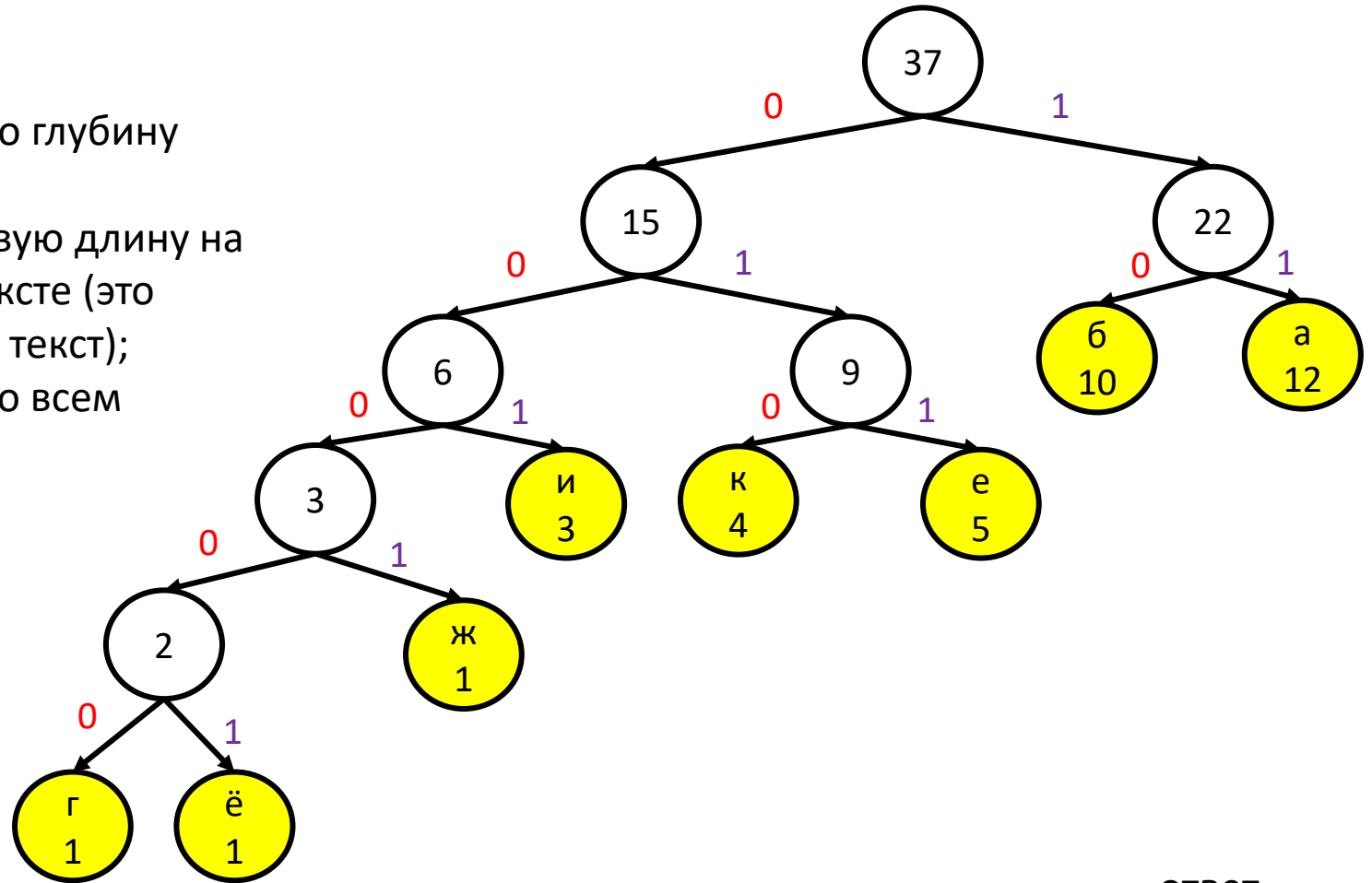
**Необходимо разработать эффективный! алгоритм,** который определяет длину в битах текста после сжатия его методом Хаффмана (само сжатие выполнять не нужно) и **оценить его время работы,** указав используемые структуры данных.

	частота
ё	1
ж	1
г	1
и	3
к	4
е	5
б	10
а	12

# Наивный алгоритм

- (1) по таблице частот строим H-дерево;
- (2) находим для каждого листа (=символа) его глубину (битовую длину символа);
- (3) перемножаем для каждого символа битовую длину на частоту встречаемости этого символа в тексте (это битовая длина всех вхождений символа в текст);
- (4) суммируем значения, полученные в (3), по всем символам текста;

	частот а	битовый код	битовая длина символа
а	12	11	2
б	10	10	2
г	1	00000	5
е	5	011	3
ё	1	00001	5
ж	1	0001	4
к	4	010	3
и	3	001	3



ОТВЕТ

$$(12*2) + (10*2) + (1*5) + (5*3) + (1*5) + (1*4) + (4*3) + (3*3) = 94 \text{ бита}$$

если не сжимать текст, то получили:  $37 * 8 \text{ бит} = 296 \text{ бит}$

Какое время работы у Вашего «наивного алгоритма»?

Разработайте более эффективный алгоритм и проверьте себя, решив эту задачу в iRunner: [Кодирование Хаффмана](#)

Имя входного файла: huffman.in  
 Имя выходного файла: huffman.out  
 Ограничение по времени: 1 с  
 Ограничение по памяти: 256 МБ

Кодирование Хаффмана (D. A. Huffman) относится к префиксному кодированию, позволяющему минимизировать длину текста за счёт того, что различные символы кодируются различным числом битов.

Напомним процесс построения кода. Вначале строится дерево кода Хаффмана. Пусть исходный алфавит состоит из  $n$  символов,  $i$ -й из которых встречается  $p_i$  раз во входном тексте. Изначально все символы считаются активными вершинами будущего дерева,  $i$ -я вершина помечена значением  $p_i$ . На каждом шаге мы берём две активных вершины с наименьшими метками, создаём новую вершину, помечая её суммой меток этих вершин, и делаем её их родителем. Новая вершина становится активной, а двое её сыновей из списка активных вершин удаляются. Процесс многократно повторяется, пока не останется только одна активная вершина, которая полагается корнем дерева.

Заметим, что символы алфавита представлены листьями этого дерева. Для каждого листа (символа) длина его кода Хаффмана равна длине пути от корня дерева до него. Сам код строится следующим образом: для каждой внутренней вершины дерева рассмотрим две дуги, идущие от неё к сыновьям. Одной из дуг присвоим метку 0, другой — 1. Код каждого символа — последовательность из нулей и единиц на пути от корня к листу.

Задача состоит в том, чтобы вычислить длину текста после его кодирования методом Хаффмана. Сам текст не дан, известно лишь, сколько раз каждый символ встречается в тексте. Этого достаточно для решения задачи, поскольку длина кода зависит только от частоты появления символов.

Разработайте алгоритм, работающий за  $n$ , где  $n$  — количество частот появления символов.

### Формат входных данных

Первая строка содержит целое число  $n$  ( $2 \leq n \leq 2\,500\,000$ ).

Вторая строка содержит  $n$  чисел  $p_i$  — частоты появления символов в тексте ( $1 \leq p_i \leq 10^9$ ,  $p_i \leq p_{i+1}$  для каждого  $i$  от 1 до  $n - 1$ ).

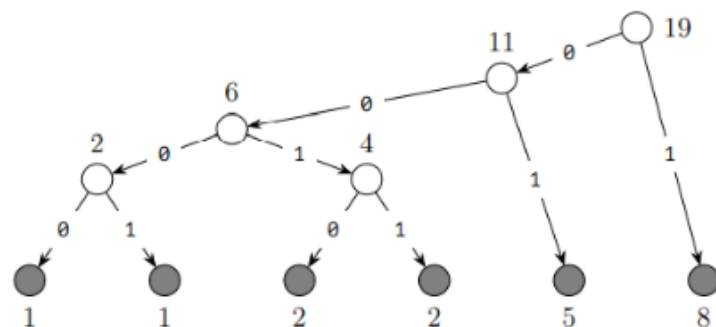
### Формат выходных данных

Выведите единственное число — длину (в битах) закодированного текста.

### Пример

huffman.in	huffman.out
6 1 1 2 2 5 8	42

Один из вариантов дерева кодирования Хаффмана:



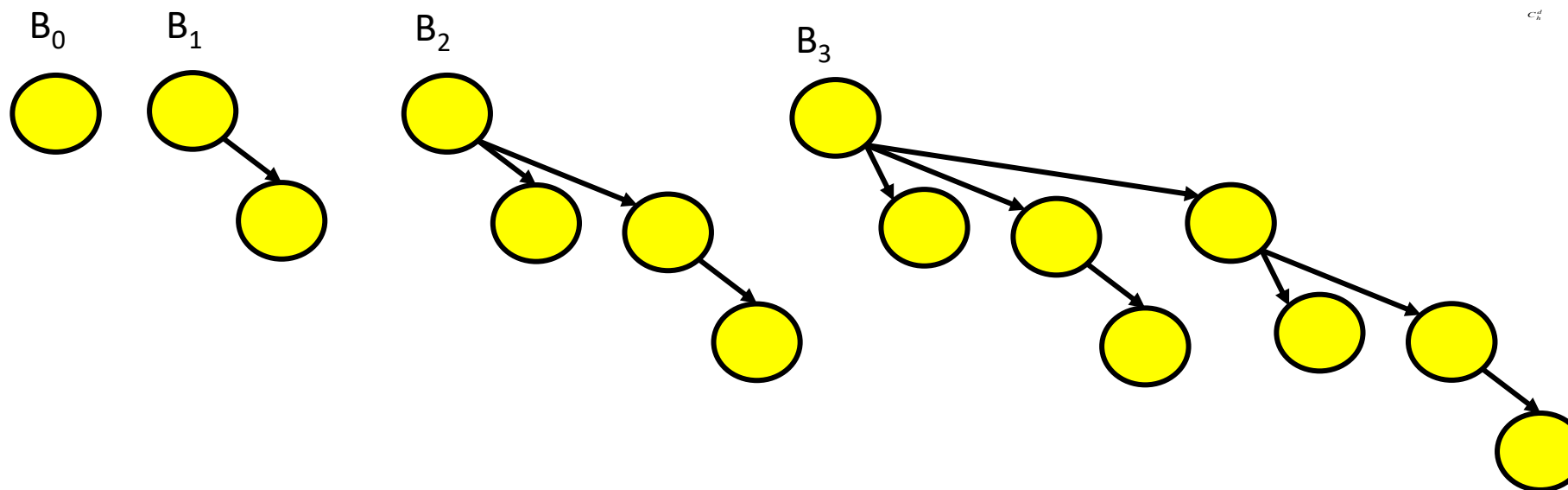
# Биномиальная куча

**Биномиальная куча** – это биномиальный лес, для которого выполняются следующие свойства:

**Инвариант 1:** каждая вершина удовлетворяет основному свойству кучи: приоритет отца не ниже приоритета каждого из его сыновей;

**Инвариант 2:** в семействе биномиальных деревьев нет двух деревьев с корнями одинакового ранга (ранг вершины – количество её сыновей, ранг дерева – ранг корня).

Семейство биномиальных деревьев:

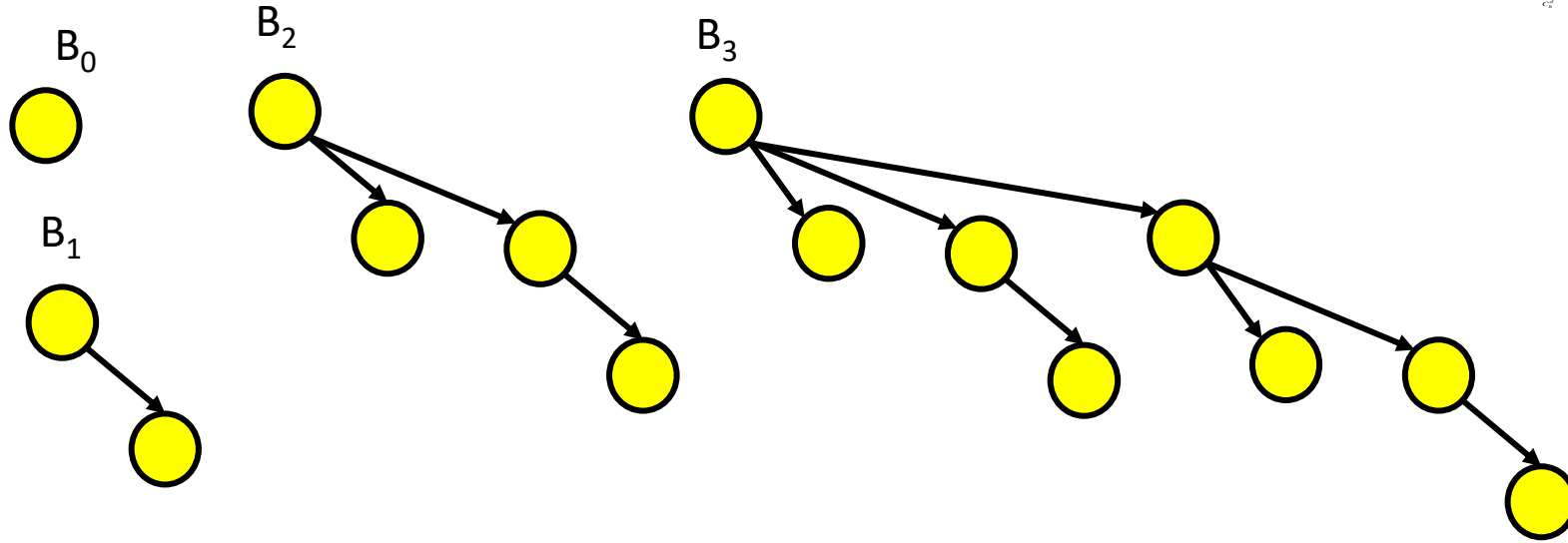


у биномиального  
дерева высоты  $h$  на  
глубине  $d$  находится  
ровно  $C_h^d$  вершин

в биномиальном дереве у  
вершины высоты  $h$   
сыновья – биномиальные  
деревья  $B_0, B_1, \dots, B_{h-1}$



## Свойства семейства биномиальных деревьев:



- для биномиального дерева **ранг** любой **вершины** совпадает с её **высотой**;
- по построению биномиальное дерево  $B_h$  содержит  $2^h$  вершин;
- если в дереве  $B_h$  содержится  $n$  вершин, то его высота  $h = \log_2 n$ ;
- так как ранг дерева равен его высоте, то **ранг** дерева  $B_h$  равен  $\log_2 n$ , где  $n$  — число вершин дерева;

- любая последовательность из  $n$  элементов может быть представлена единственным образом как семейство биномиальных деревьев, в котором не более одного дерева каждого ранга:

разложим число  $n$  по степеням 2, например, если  $n = 13 = 2^3 + 2^2 + 2^0$ , то семейство биномиальных деревьев состоит из деревьев  $B_3, B_2, B_0$

- для семейства из  $k$  уникальных биномиальных деревьев, в котором суммарно  $n$  вершин справедливо неравенство:

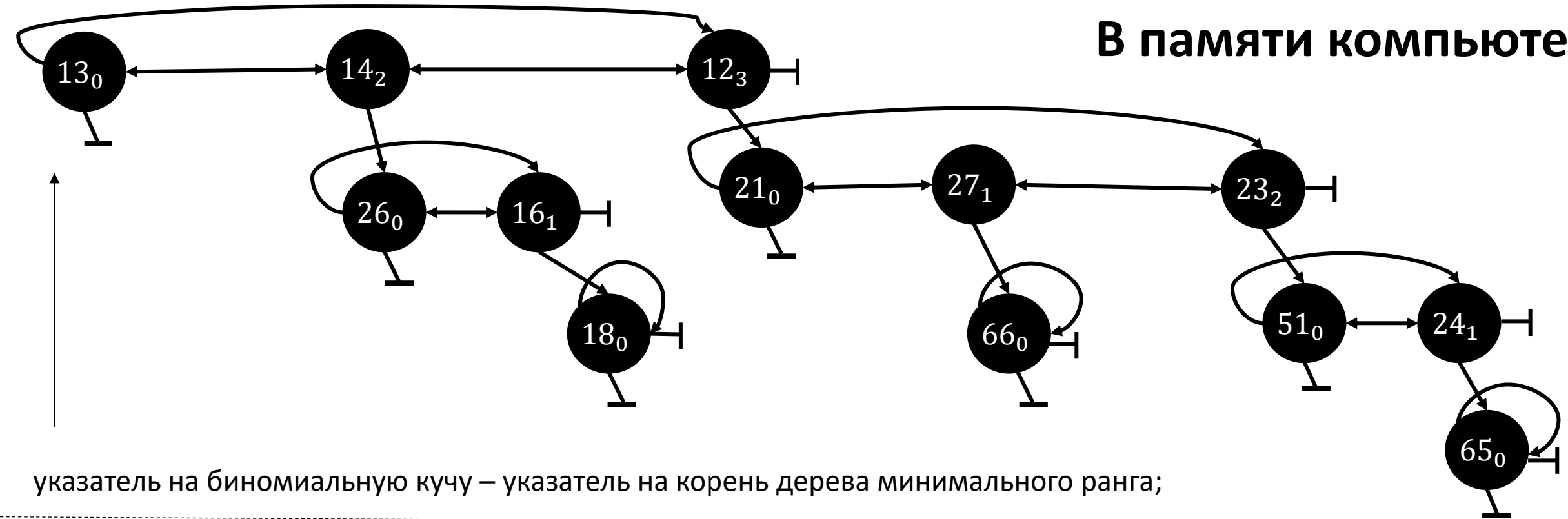
$$n \geq n^{\min} = 2^0 + 2^1 + \dots + 2^{k-1} = 2^k - 1$$

$$k \leq \log_2(n + 1)$$

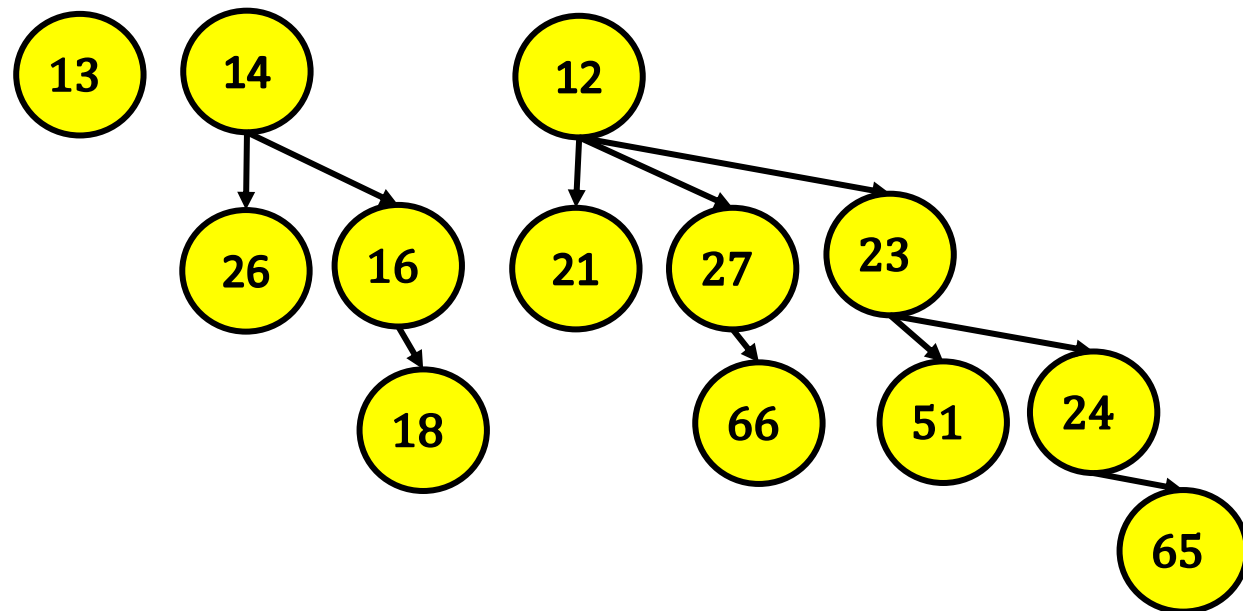
$n^{\min}$  - минимально возможное число вершин в семействе



# В памяти компьютера



указатель на биномиальную кучу – указатель на корень дерева минимального ранга;



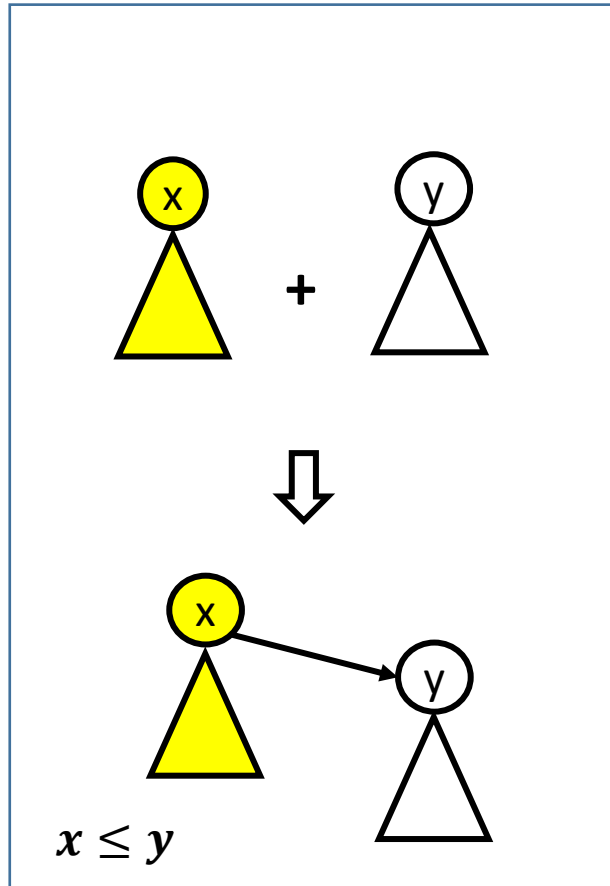
каждая вершина содержит информацию о ранге;  
деревья упорядочены по рангам;  
вершина имеет указатель на первого сына (null, если нет сыновей);  
вершина имеет указатель на левого и правого брата;  
сыновья каждой вершины представлены в виде двунаправленного списка, сыновья упорядочены по рангам, указатель левого брата первого сына – последний сын, а указатель правого брата null, если правого брата нет;



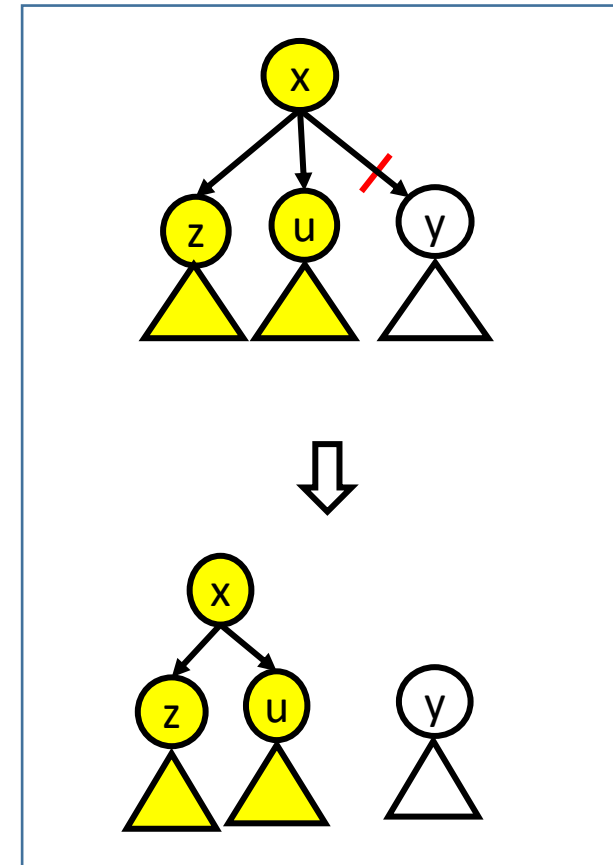


Дополнительные вспомогательные операции **link** и **cut**,  
которые нужны для выполнения базовых операций

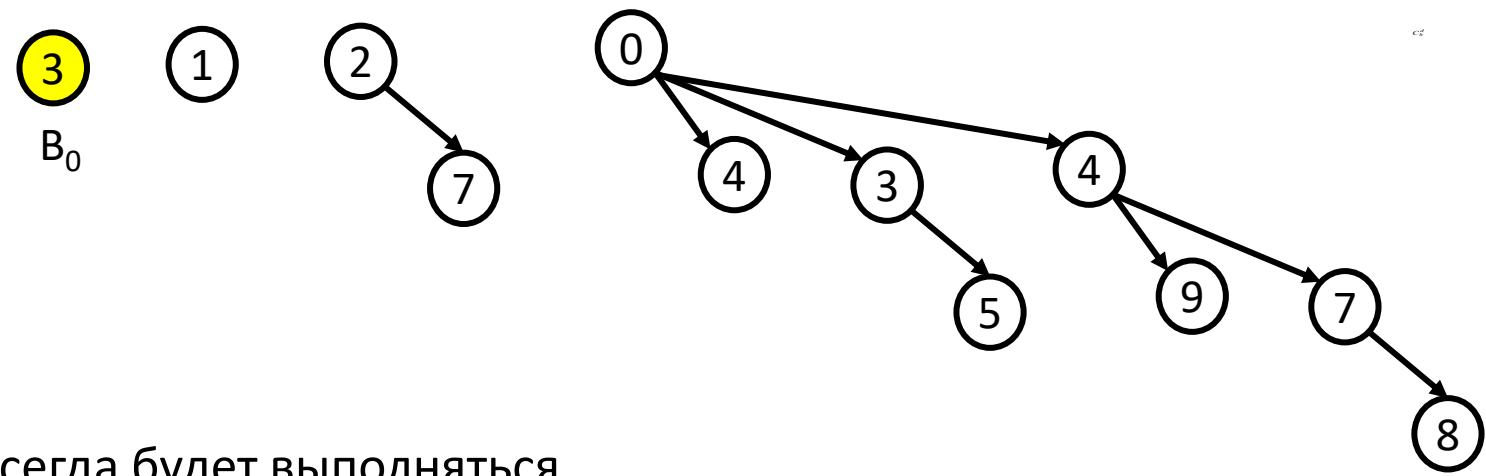
**link**( $x, y$ )



**cut**( $y$ )

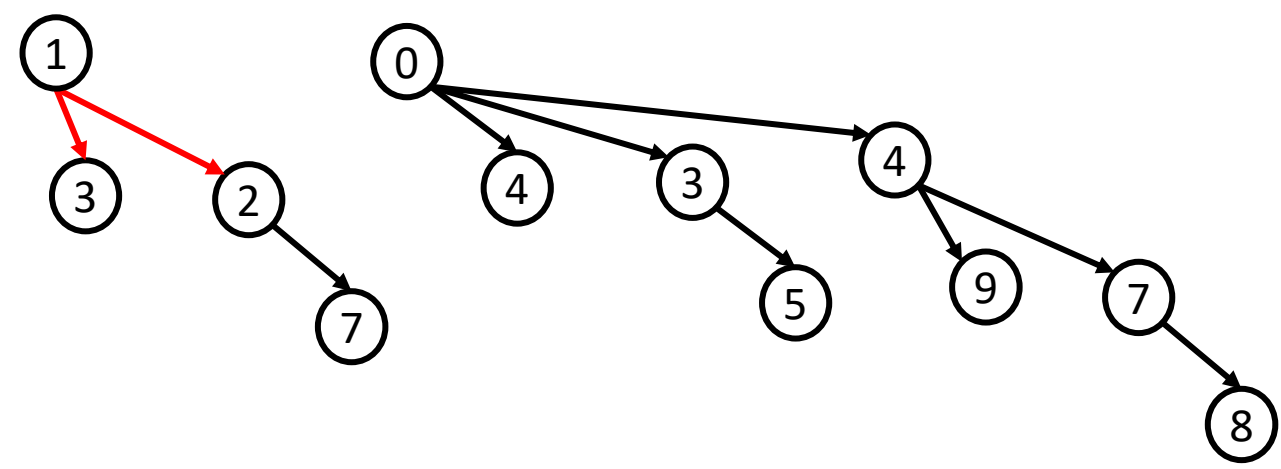


**Insert(x)** — добавление ключа  $x$ .



**Инвариант 1** всегда будет выполняться.

**Инварианта 2** - выполним серию операций **link** над деревьями одного ранга:

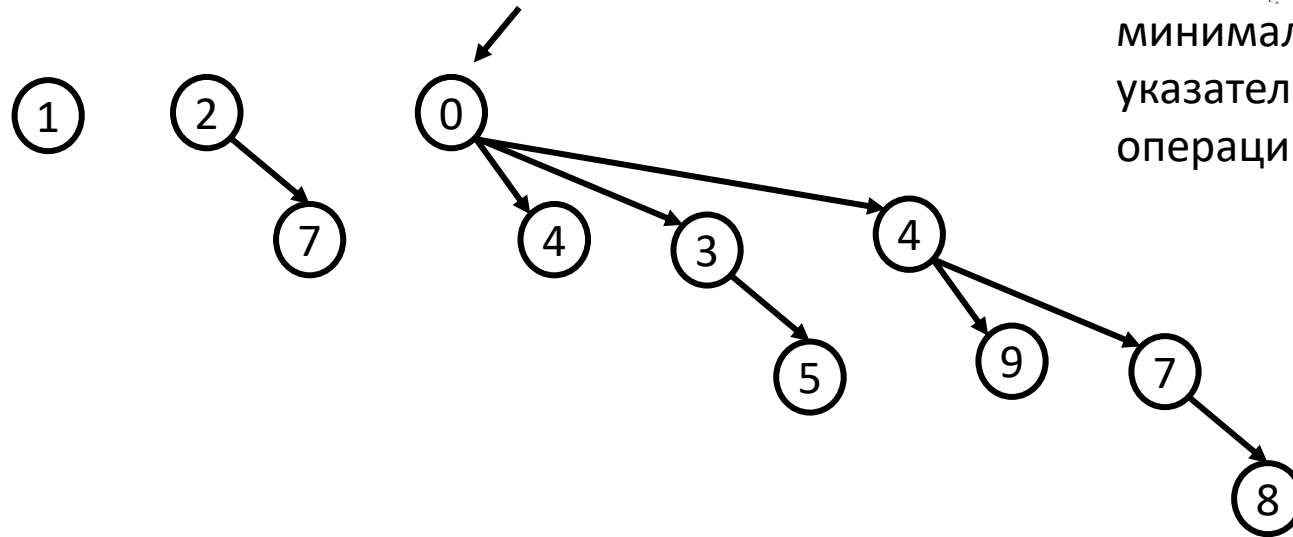


Так как каждый **link** уменьшает число деревьев на 1, а число деревьев в биномиальном семействе из  $n$  вершин -  $O(\log n)$ , то время работы операции добавления ключа:

$O(\log n)$ .

**GetMin()** — поиск минимального ключа;

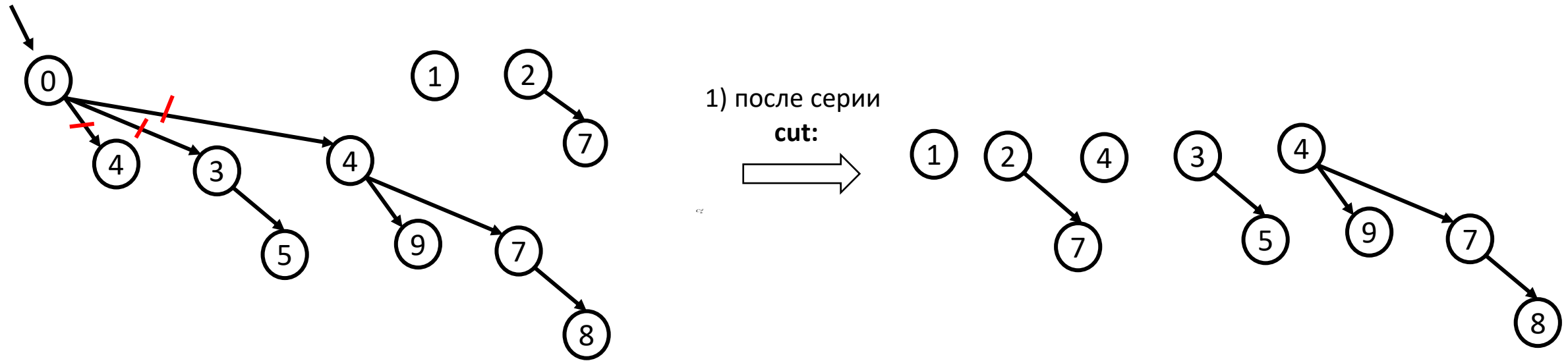
$O(1)$



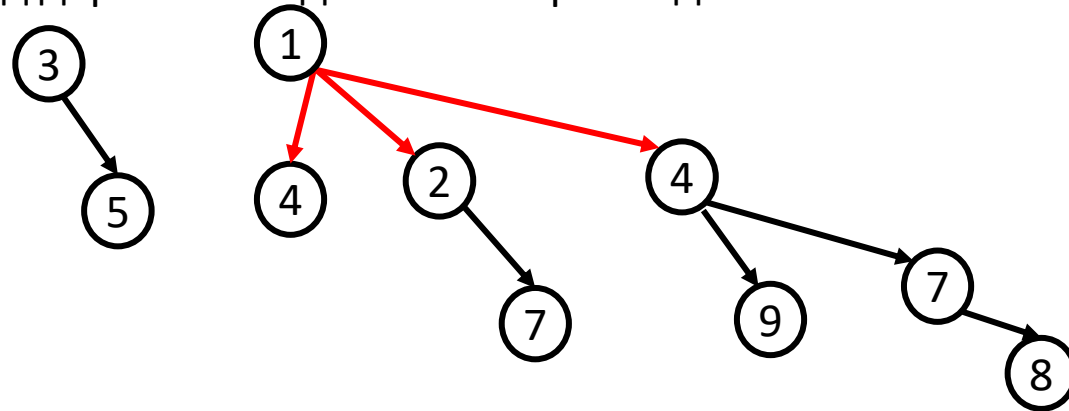
хранят указатель на корень дерева с минимальным ключом и поддерживают этот указатель в процессе выполнения других операций;



**ExtractMin()** — удаление минимального ключа;



2) выполним серию операций **link** над деревьями одинакового ранга для восстановления инварианта 2:



Так как каждый **link** уменьшает число деревьев на 1, а число деревьев в семействе есть  $O(\log n)$ , то время работы операции удаления минимального элемента:

$$O(\log n)$$



**Heapify** — построение кучи для последовательности из  $n$  ключей

Биномиальную кучу будем строить вызовом  $n$  раз функции **Insert**( $x$ ) .

*Оценим число биномиальных деревьев  
после выполнения каждой операции **Insert** ( $x$ )*

пусть  $t_i$  — число операций **link**, которые были выполнены при добавлении элемента  $x$

$$\text{1-й элемент:} \quad 0 + 1 - t_1$$

$$\text{2-й элемент:} \quad (1 - t_1) + 1 - t_2 = 2 - (t_1 + t_2)$$

$$\text{3-й элемент:} \quad 2 - (t_1 + t_2) + 1 - t_3 = 3 - (t_1 + t_2 + t_3)$$

...

$$\text{\textit{n}-й элемент:} \quad n - (t_1 + t_2 + t_3 + \dots + t_n)$$

---

$$n - \sum_{i=1}^n t_i \geq 1$$

$$\sum_{i=1}^n t_i \leq n - 1$$



Так как  $\sum_{i=1}^n t_i \leq n - 1$ , то время работы алгоритма **Heapify** построения кучи для последовательности из  $n$  ключей в худшем случае:

$$O(n)$$

Усреднённая оценка операции добавления одного элемента в биномиальную кучу:

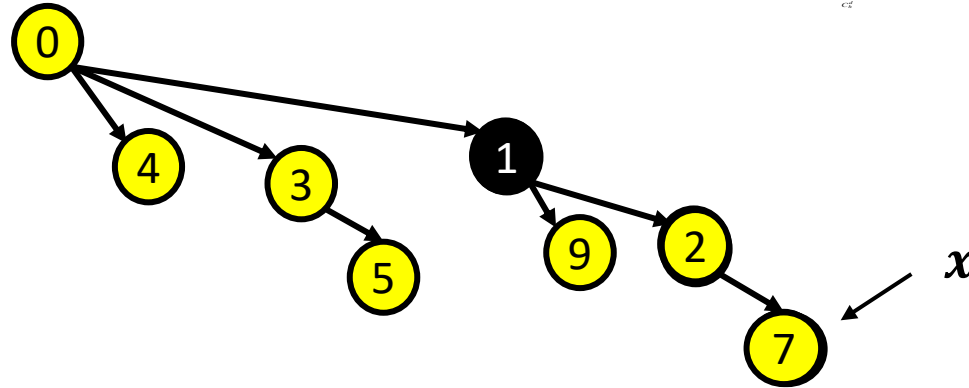
- 1) предположим, что в биномиальной куче было изначально  $z_0$  деревьев;
- 2) выполним  $k$  раз операцию **Insert**( $x$ );
- 3) просуммируем затраченное в худшем случае время;
- 4) разделим полученное значение на число выполненных операций.

$$\frac{z_0 + O(k) + O(k)}{k} = O(1)$$



## DecreaseKey (уменьшение ключа)

Предполагается, что задана позиция модифицируемой вершины .

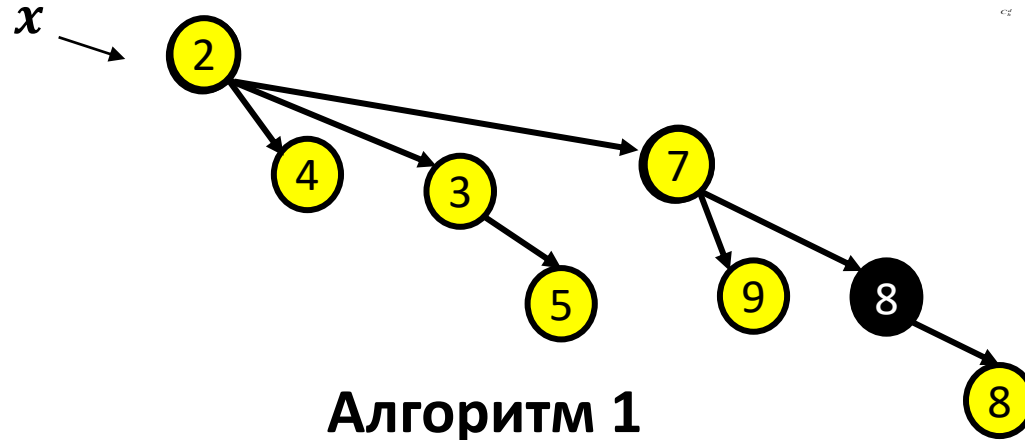


Уменьшаем ключ вершины  $x$  и просеиваем (обменами с отцом) элемент  $x$  до тех пор, пока для него не выполнится свойство кучи.

Так как один обмен выполняется за  $O(1)$ , а количество обменов ограничено высотой дерева  $h = O(\log n)$ , то описанный алгоритм выполнит операцию уменьшения ключа за время:

$$\Theta(\log n)$$

## IncreaseKey(увеличение ключа)



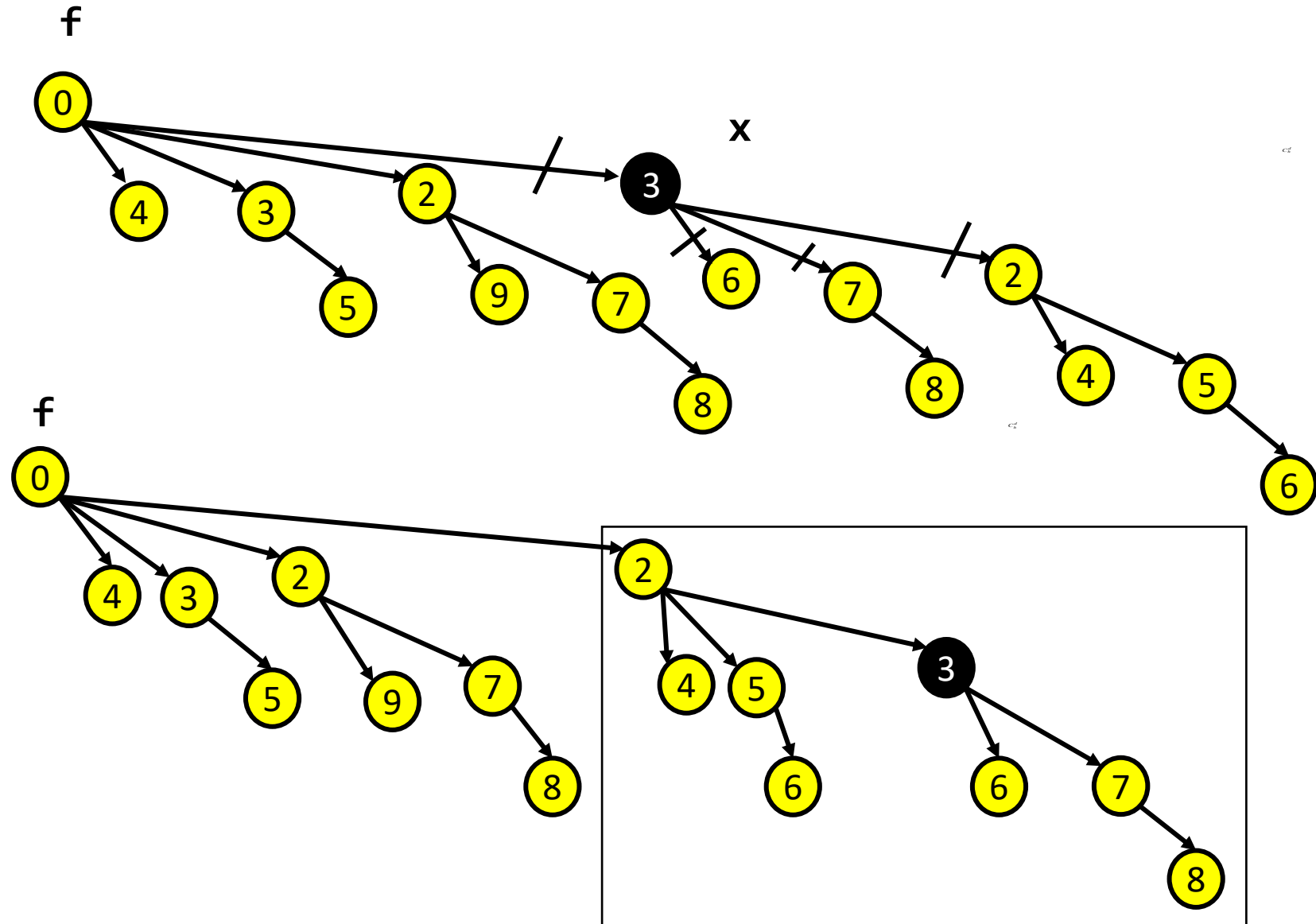
1. Увеличиваем ключ вершины  $x$ .
2. Если после этого для  $x$  нарушается свойство кучи, то просеиваем её (обменами с наименьшим из сыновей) тех пор, пока не выполнится инвариант 1.

Так как одно просеивание выполняется за  $O(\log n)$ , а число просеиваний ограничено высотой дерева  $h = O(\log n)$ , то алгоритм 1 выполнит операцию увеличения ключа:

$$\Theta(\log^2 n)$$



## Алгоритм 2



1. Увеличиваем ключ вершине  $x$ .
2. Если инвариант 1 выполняется, то процедура увеличения ключа завершена.
3. Если инвариант 1 НЕ выполняется, то
  - 3.1. Применяем операцию **cut** к самой вершине  $x$  и ко всем её сыновьям.  
Пусть  $f$  – отец вершины  $x$ .
  - 3.2. Восстанавливаем инвариант 2: серия операций **link** над «отрезанными» деревьями одного ранга (каждое из этих деревьев – биномиальное).  
Суммарное число link -  $O(\log n)$ .
  - 3.3. Полученное дерево «прикрепляем» к  $f$ .

Время работы алгоритма 2:  $\Theta(\log n)$

Время выполнения базовых операций для биномиальной кучи, содержащей  $n$  вершин:

## Базовый набор операций:

**GetMin()** —

поиск  
минимального  
ключа;

$O(1)$

**ExtractMin()** —

удаление  
минимального ключа;

$O(\log n)$

**Insert(x)** —

добавление ключа  $x$ .

$O(\log n)$

## Расширенный набор операций:

**IncreaseKey**  $O(\log n)$

**DecreaseKey**

модификация ключа вершины на  
заданную величину

(предполагается, что известна позиция вершины  
внутри структуры данных);

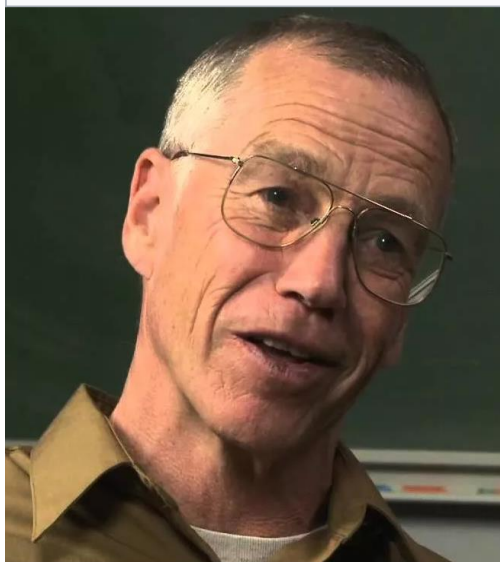
**Heapify** —

построение кучи для  
последовательности  
из  $n$  ключей.

$O(n)$

Майкл Фридман

англ. *Michael Hartley  
Freedman*



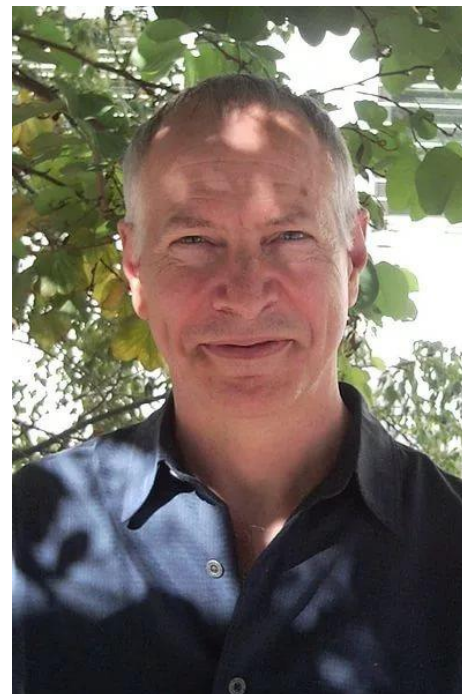
Дата рождения: 21.04.1951

Место рождения: Лос-Анджелес

# Куча Фибоначчи (англ. *Fibonacci heap*)

была предложена  
Майклом Фридманом  
и  
Робертом Тарьяном  
в **1984** году.

Роберт Андре Тарьян  
англ. Robert Endre Tarjan



Дата рождения: 30.04.1948

Место рождения: Помона (штат  
Калифорния, США)

**Куча Фибоначчи** – это семейство корневых деревьев, для которого выполняются следующие свойства (инварианты):

**Инвариант 1.** Каждая вершина в куче Фибоначчи удовлетворяет основному свойству кучи: приоритет отца не ниже приоритета каждого из его сыновей.

**Инвариант 2.** В семействе корневых деревьев нет двух деревьев с корнями одинакового ранга.

**Инвариант 3.** Каждая некорневая вершина в куче Фибоначчи может потерять не более одного сына при выполнении процедуры *cut*.

Название «кучи Фибоначчи» обусловлено тем, что для доказательства оценок трудоемкости операций используются числа Фибоначчи.

ранг любого узла в куче Фибоначчи не превосходит:

$$2 \cdot \log_2 n + 1$$

если в куче  $n$  вершин, то число деревьев в ней:

$$2 \cdot \log_2 n + 2$$

# DecreaseKey (уменьшение ключа)

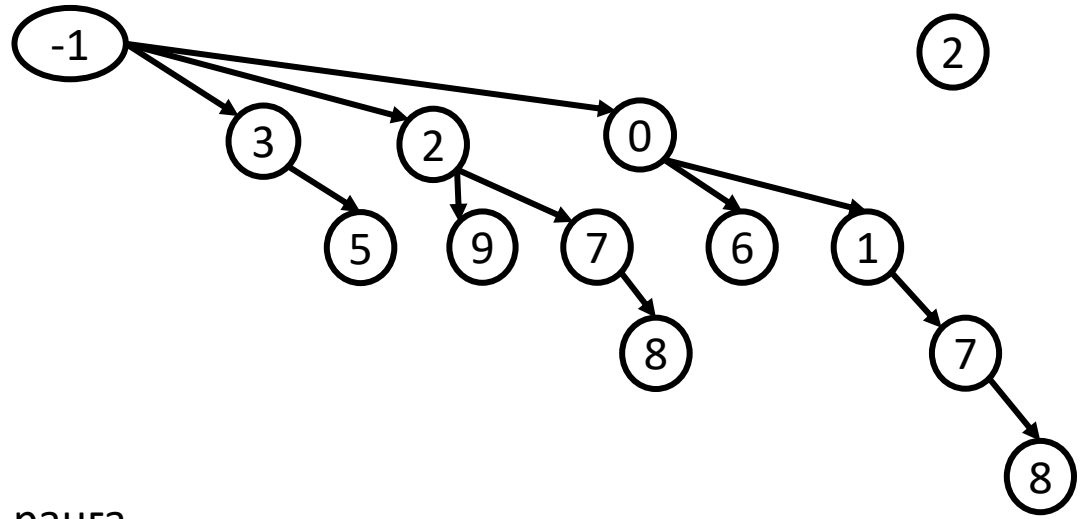
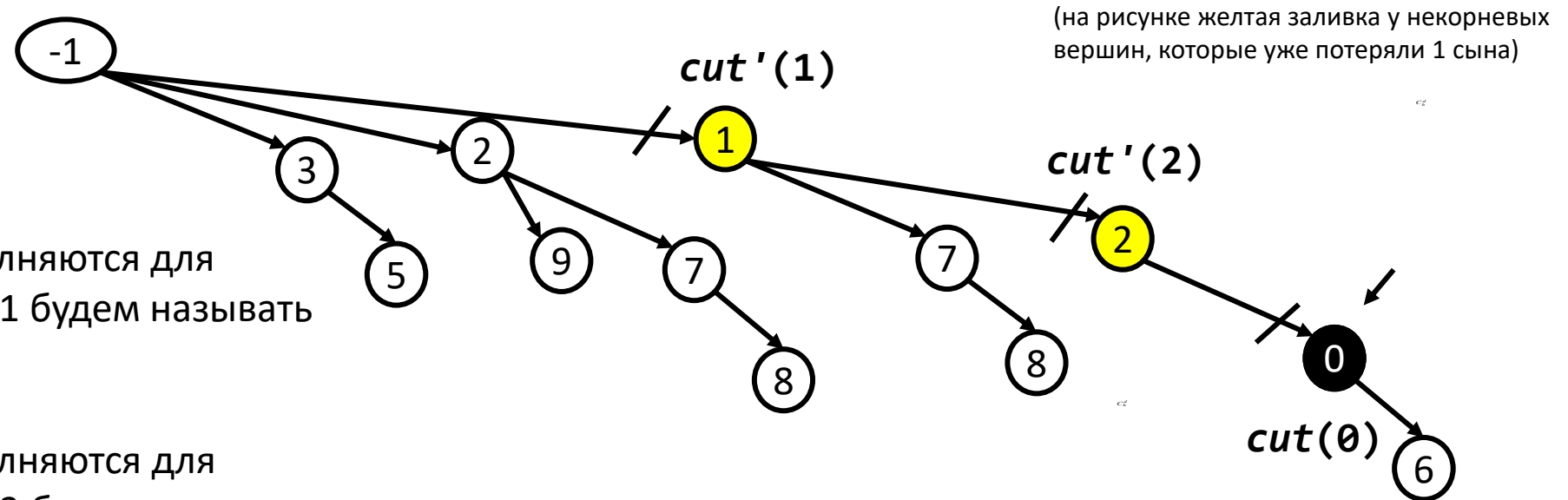
- ✓ операции **cut**, которые выполняются для восстановления инварианта 1 будем называть **исходными cut (cut)**
  - ✓ операции **cut**, которые выполняются для восстановления инварианта 3 будем называть **порождёнными cut (cut')**
- (на рисунке желтая заливка у некорневых вершин, которые ранее уже теряли сына)

Выполнены:

Восстановление инварианта 1:  
одна исходная операция **cut**

Восстановление инварианта 3:  
серия порожденных **cut'**

Восстановление инварианта 2:  
серия операций **Link** над деревьями одного ранга



# IncreaseKey (увеличение ключа)

Выполнены:

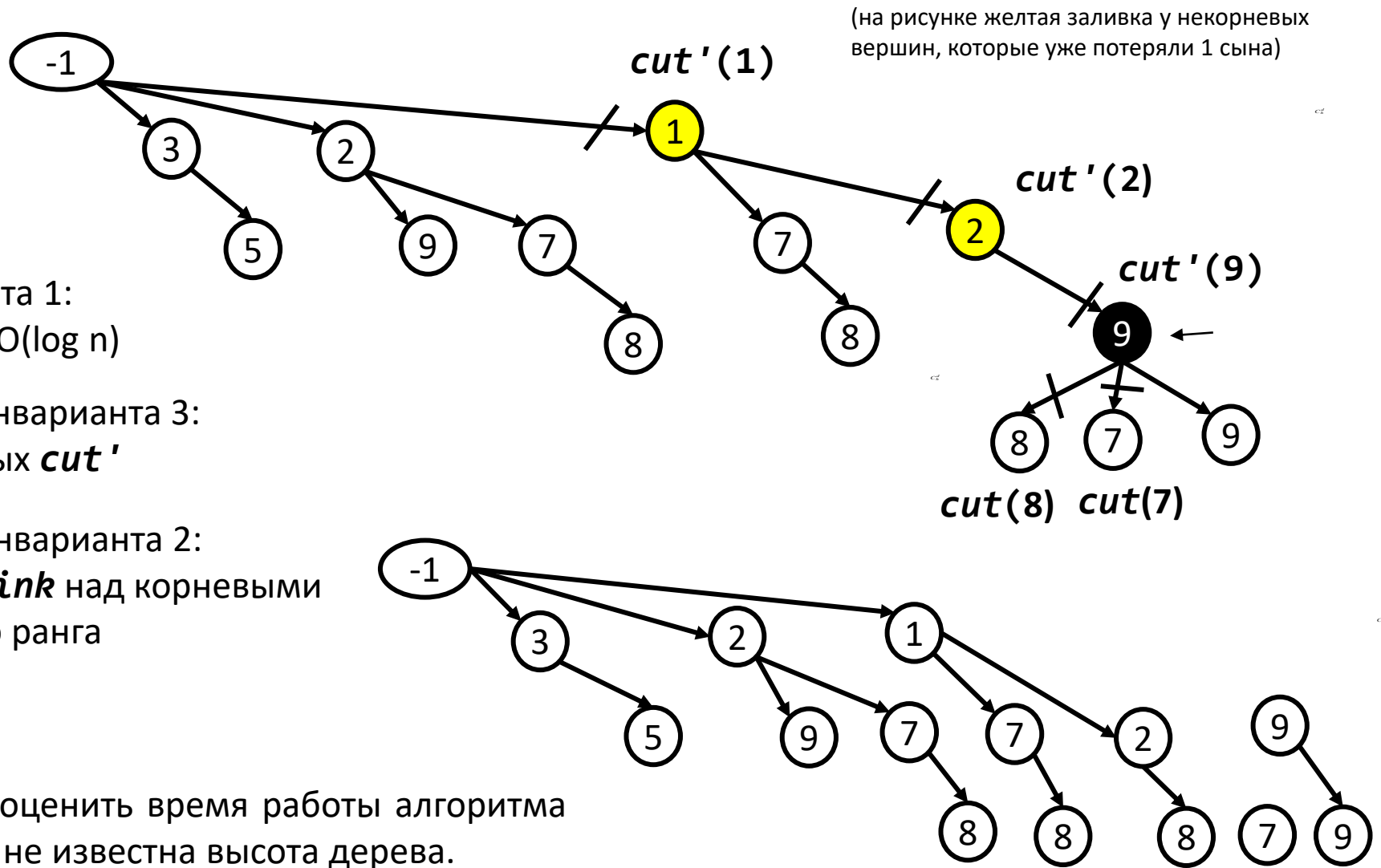
Восстановление инварианта 1:  
исходные операция **cut** -  $O(\log n)$

Восстановление инварианта 3:  
серия порожденных **cut'**

Восстановление инварианта 2:  
серия операций **Link** над корневыми  
деревьями одного ранга

В худшем случае не можем оценить время работы алгоритма  
модификации ключа, так как не известна высота дерева.

Будем оценивать **усреднённое** время работы операции.



Предположим, что мы выполнили некоторое число *исходных операций cut*, а они привели к выполнению серии *порождённых операций cut'* и *link*.

**Справедливы следующие утверждения:**

1. Общее число *порожденных операций cut'* не превышает общего числа *исходных cut*

$$n(cut') \leq n(cut)$$

2. Число процедур *link* равно, как максимум, *m* плюс число всех процедур *cut*, где *m* — начальное число корневых деревьев:

$$n(link) \leq m + n(cut') + n(cut)$$



# Куча Фибоначчи

Усреднённая оценка трудоемкости  
операции добавления нового элемента:

$$O(1)$$

$$\frac{O(k) + O(k)}{k} = O(1)$$

Усреднённая оценка трудоемкости  
операции уменьшения ключа (задана  
ссылка на элемент в структуре):

$$O(1)$$

$$\frac{O(k) + O(k) + O(k)}{k} = O(1)$$

Усреднённая оценка трудоемкости  
операции увеличения ключа (задана  
ссылка на элемент в структуре):

$$O(\log n)$$

$$\frac{O(k \cdot \log n) + O(k \cdot \log n) + O(k \cdot \log n)}{k} = O(\log n)$$

Усреднённая оценка трудоемкости  
операции удаления минимального  
элемента:

$$O(\log n)$$

$$\frac{O(k \cdot \log n) + O(k \cdot \log n)}{k} = O(\log n)$$

## Тема 3. Структуры данных

0.3. Бинарная куча (проверка на соответствие структуре)

0.4. Биномиальная куча (понимание структуры)

43. 2 Кодирование Хаффмана



БЕЛОРУССКИЙ  
ГОСУДАРСТВЕННЫЙ  
УНИВЕРСИТЕТ

Спасибо за внимание!