

Лекция 4

Очереди с приоритетами, кучи

Очередь с приоритетом и кольцевая очередь

Мы продолжаем изучать динамические структуры данных. На прошлой лекции мы познакомились с простой очередью сегодня занятие у нас будет посвящено чуть более сложным вариантам той самой очереди.

Что такое очередь с приоритетом и что такое кольцевая очередь? Начнем с более простой структуры - это кольцевая очередь. Принцип её работы точно такой же самый как и у обычной простой очереди, то есть первый элемент который попадает в нашу очередь – первым её и покидает. Но с одним маленьким нюансом, если в случае с простой очередью мы извлекаем данные из элемента который находятся в начале очереди, и затем этот элемент исключаем из очереди, то в случае кольцевой очереди мы этот элемент не исключаем. Мы получаем данные этого элемента, а затем сам этот элемент перемещаем в конец очереди. По кругу. Поэтому, собственно такая очередь и называется кольцевой. Далее мы извлекаем следующий элемент, получаем его данные и опять помещаем его в конец очереди. Таким образом эта очередь имеет как бы зацикленный вид.

Очередь с

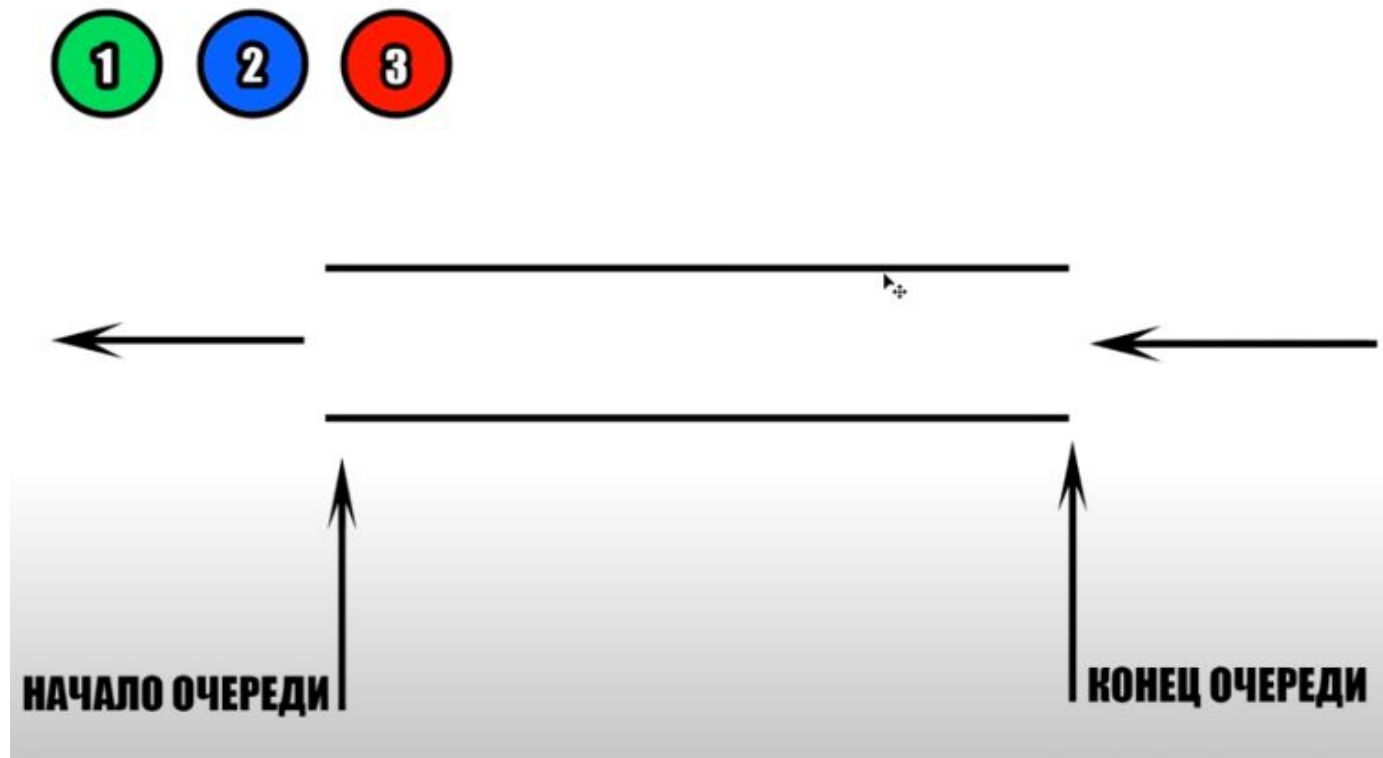
приоритетом

Несколько более сложным вариантом очереди является **очередь с приоритетом**. Порядок извлечения элементов из такой очереди зависит от приоритета обработки такого элемента. Критерием, по которому будет оцениваться приоритетность необходимой обработки элементов очереди, то есть, в каком порядке они будут извлекаться

может быть всё что угодно:

- данные
- структуры
- списки
- классы и т.д.

Главное, что бы у Вас был **определён приоритетный параметр** по которому и будет совершаться выборка. В нашем, конкретном примере, в качестве приоритета выбраны просто обычные целые числа. Вы видите слева вверху на рисунке изображения трех элементов. Логика их обработки будет такая: **чем меньше** значение числа элемента - **тем**

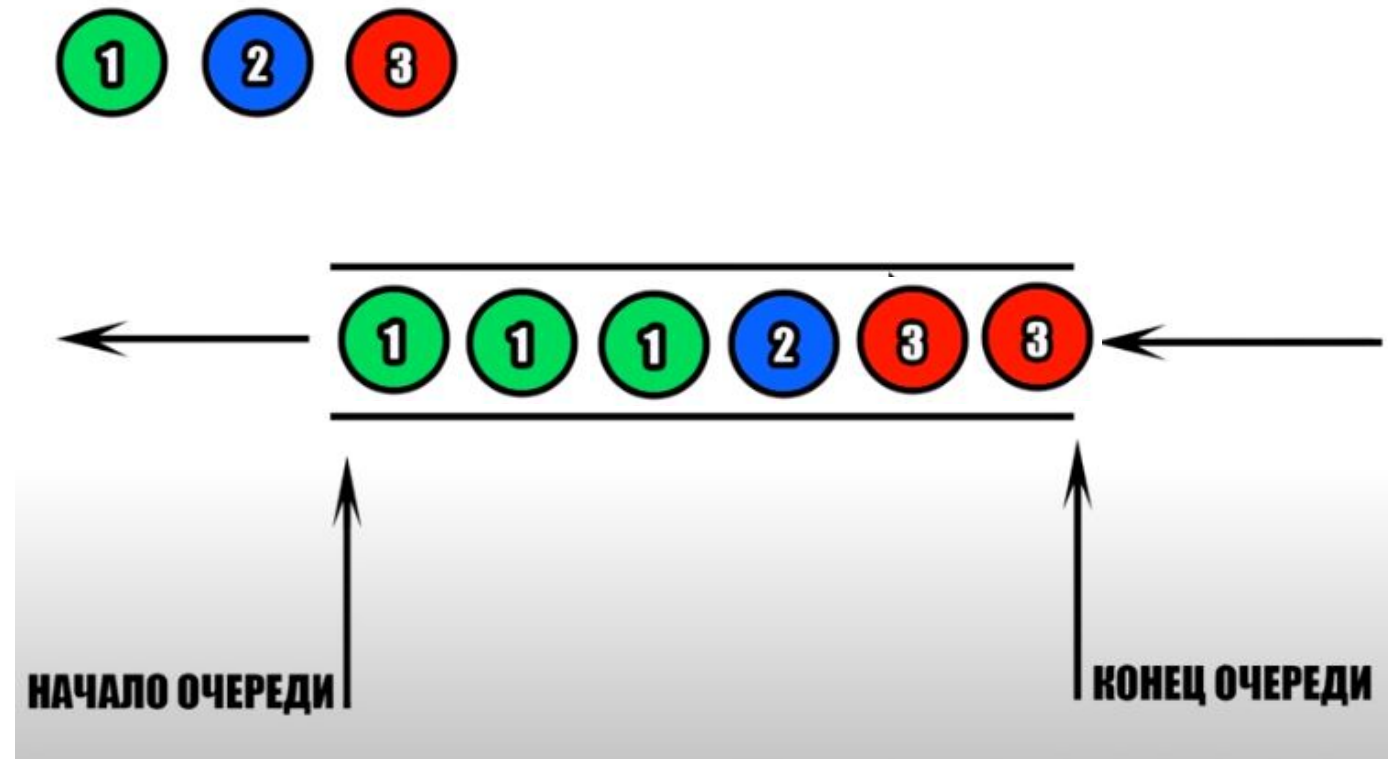


То есть у зеленого элемента будет **первый приоритет**, у синего – **второй**, ну и у красного – **третий**. Самым высокоприоритетным, для обработки, элементом, в данной схеме, и является зеленый элемент со значением «1».

Рассмотрим варианты, каким образом может быть организована работа приоритетной очереди. Один из этих вариантов, гласит нам о том, что элементы в очереди должны быть упорядоченный по приоритетам **в момент поступления элемента в очередь**. Это означает, что когда мы добавляем элемент в очередь, он сразу же становится в нужном порядке в соответствии со своим приоритетом. Такой вариант организации очереди с приоритетом называется **очередь с приоритетным включением**.

Существует еще один вариант очереди с приоритетом - это **очередь с приоритетным исключением**. При таком варианте организации очереди, у нас элементы добавляются в очередь просто в порядке их поступления, и никак не сортируются. Но в момент извлечения из очереди мы сначала выбираем элемент который имеет наивысший приоритет для обработки.

То есть разница в том, где выполняется обработка постановки задач согласно их приоритетам.

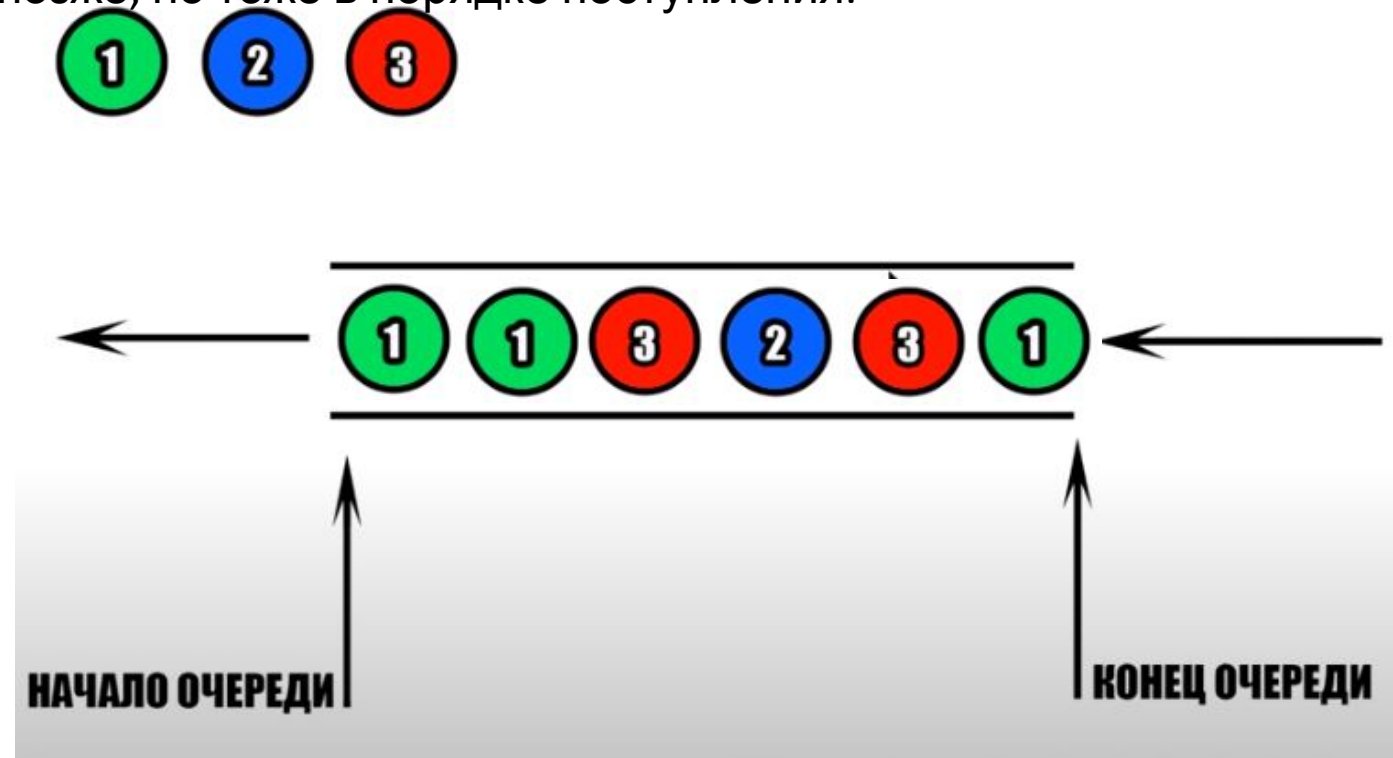


Очередь с приоритетным

Таким образом, обработка положения элемента в очереди, согласно его приоритету, в первом варианте (с приоритетным включением) сводится к тому, что у нас **сначала ставится элемент в нужное положение**, и затем мы уже не разбираемся, где он находится. Просто все подряд извлекаем, так как все элементы у нас будут уже с одной стороны отсортированы по мере важности, в другой стороны – они будут стоять в очереди по мере их постановки. На рисунке, на прошлом слайде, зелёные элементы с номером «1» стоят друг за другом, но это **не обязательно значит**, что они в очередь поступили друг за другом! Вполне возможны любые варианты: сначала в очередь встали синий и красный элементы, потом зелёный, потом красный, потом снова два зелёных. Но **сам принцип формирования очереди с включением расставил их** в том порядке, что на рисунке, при этом порядок поступления зелёных элементов тоже сохранён. Сначала стоит зелёный который пришёл первым, потом – другие два, что пришли позже, но тоже в порядке поступления.

Во втором же случае (с приоритетным исключением) мы просто добавляем в очередь все элементы подряд, а когда нам нужно извлечь элемент – тогда мы уже и разбираемся у какого из наших элементов наивысший приоритет для обработки и для извлечения его данных и так далее. При этом снова важнее всего – приоритет извлекаемого элемента, потом момент его постановки в очередь.

На рисунке справа сначала извлекаются, последовательно, два зелёных элемента, потом очередь проверяется на наличие ещё «зелёных». У нас такой есть, следовательно - извлекаем его. Потом синий, потом первый красный, потом – второй.



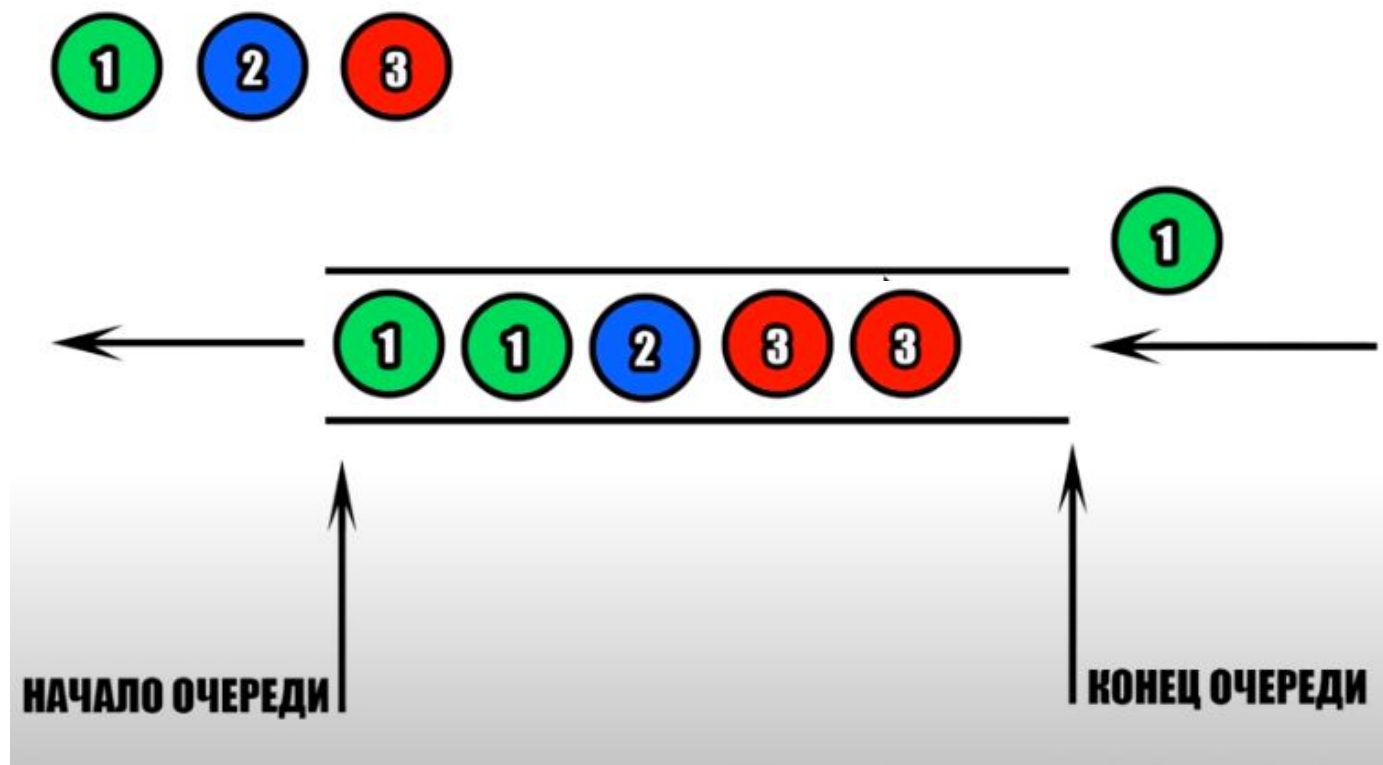
Очередь с приоритетным

Рассмотрим добавления нового элемента в очередь с приоритетным включением. При таком варианте, элементы в очереди находятся в упорядоченном состоянии (см. рисунок) согласно их приоритетам. Представим, что мы хотим добавить еще какой-то элемент в нашу очередь. Пускай это будет зелёный элемент с приоритетом 1, то есть, с наивысшим приоритетом.

Согласно правилам обычной очереди мы должны были бы поставить такой элемент просто в конец очереди, но так как он имеет приоритет 1, то есть наивысший, а очередь у нас **с приоритетным включением**, то мы должны выполнить такие шаги: мы должны наш новый элемент расположить согласно его приоритету.

Получается, что в нашей очереди для каждого приоритета есть своя очередь. Элементы с приоритетом 1 находится в своей очереди они выстраиваются друг за другом в порядке их поступления.

Элементы под индексом 2 выстраиваются в свою очередь согласно их поступлению и далее элементы под индексом 3. Мы **не можем** поместить новый элемент ни в начало очереди, ни между уже имеющимися зелёными элементами, так как они пришли раньше. В данном случае, нам нужно поместить наш новый элемент между последним зелёным и первым синим элементами. Это нужно учесть при разработке алгоритма добавления элемента в очередь с приоритетным включением.



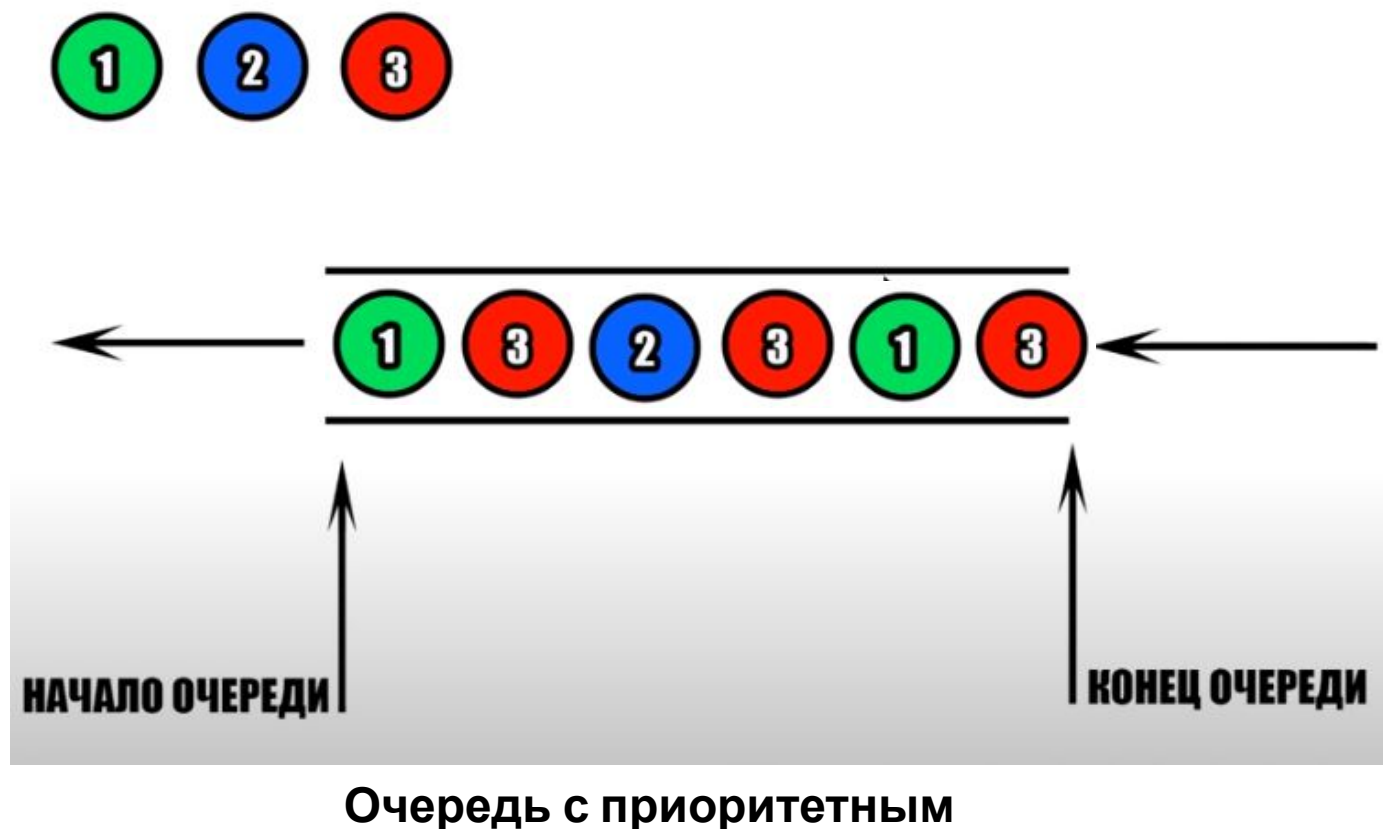
Очередь с приоритетным

В случае с очередью с приоритетным исключением – у нас гораздо проще процесс постановки элементов в очередь – просто записываем их по мере поступления. А вот процесс извлечения должен идти согласно приоритетов. Например, очередь на рисунке. Сначала выходит/забираем зелёный (с высшим приоритетом), что стоит ближе к выходу (началу очереди). Потом проверяем все элементы на наличие ещё «зелёных». Если находим (как у нас) – забираем. Потом ищем и выводим элементы с приоритетом «2» (синие), а когда кончатся они – выводим красные, в порядке их постановки в очередь.

Чисто технически, поиск и вынимание приоритетных элементов, лежащих в глубине очереди, удобно реализовывать на стеке. Разумеется размер стека должен быть **не меньше, чем длина очереди минус 1** (почему?). Применительно к нашему рисунку, при поиске второго зелёного элемента, было бы удобно откинуть на стек сначала красный, потом синий, потом снова красный элементы (считаем их по порядку, от выхода), а после обнаружения и извлечения зелёного – снова всё вернуть из стека на своё место в очереди. При этом сохраняется порядок поступления элементов.

Потом этот же алгоритм применяется для поиска синих, а потом и красных элементов.

Замечание. Если в очереди остаются только элементы с самым низким приоритетом, выводить их можно уже быстрее, без использования стека. Это нужно помнить и



Как Вам уже должно быть понятно, у обеих очередей, при таком «лобовом» алгоритме организации, нет выигрыша в использовании, так как и там и там есть операции за $O(1)$ и $O(n^m)$ (где n – количество элементов в очереди, а m – количество приоритетов).

Есть ли способ уменьшить затраты времени? В программировании (и в Ваших лабораторных работах) есть прямой ответ на этот вопрос. Если нужно выиграть время – нужно менять алгоритм. Даже если Ваш прежний алгоритм давал правильное решение, но **за большее время**.

В прошлом семестре мы уже знакомились с одной структурой, которая всегда была организована так, что её элементы упорядочивались как в очереди с приоритетами. В частности – с приоритетным включением.

Это была – неубывающая двоичная Куча.

Двоичная куча (binary heap) – просто реализуемая структура данных, позволяющая быстро (за логарифмическое время) добавлять элементы и извлекать элемент с максимальным приоритетом (например, максимальный по значению).

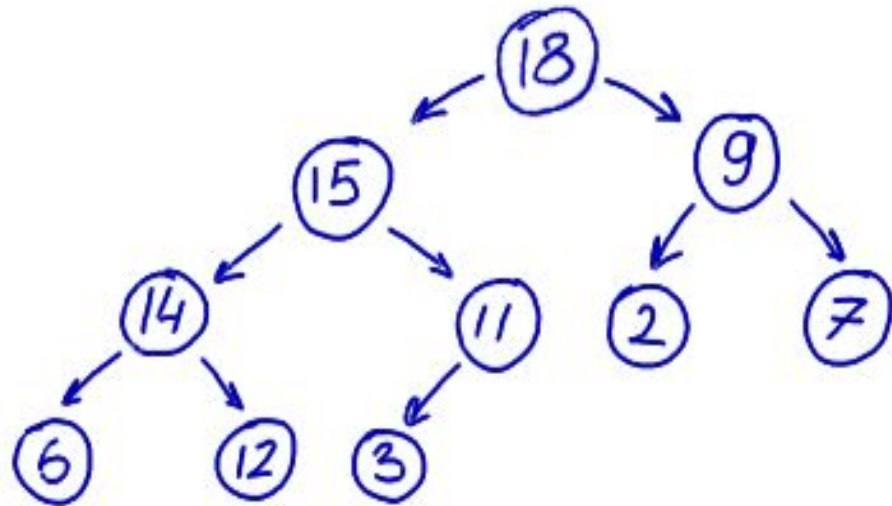
Двоичная куча представляет собой полное бинарное дерево (complete binary tree), для которого выполняется основное свойство кучи: все уровни заполнены, возможно, за исключением последнего и приоритет каждой вершины **больше приоритетов её потомков**. В простейшем случае приоритет каждой вершины можно считать равным её значению. В таком случае структура называется max-heap, поскольку корень поддерева является максимумом из значений элементов поддерева.

- Значение в любой вершине не меньше, чем значения её потомков
- Глубина всех листьев (расстояние до корня) отличается не более чем на 1 слой
- Последний слой заполняется слева направо без «дырок»

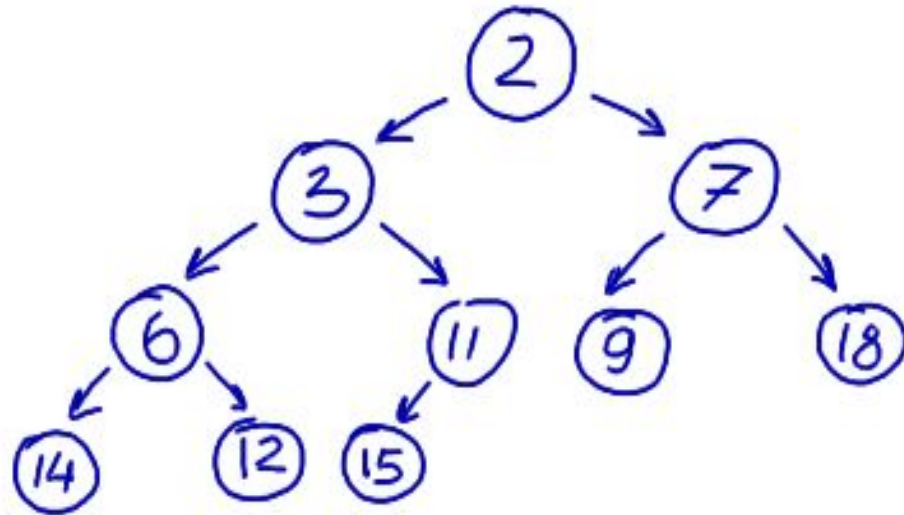
Свойство порядка размещения вершин (heap order property) заключается в том, что двоичная куча может быть максимальной (maximum heap) или минимальной (minimum heap):

maximum heap - если значение (приоритет) в каждом узле больше или равно значениям в узлах потомков;

minimum heap - если значение (приоритет) в каждом узле меньше или равно значениям в узлах потомков.

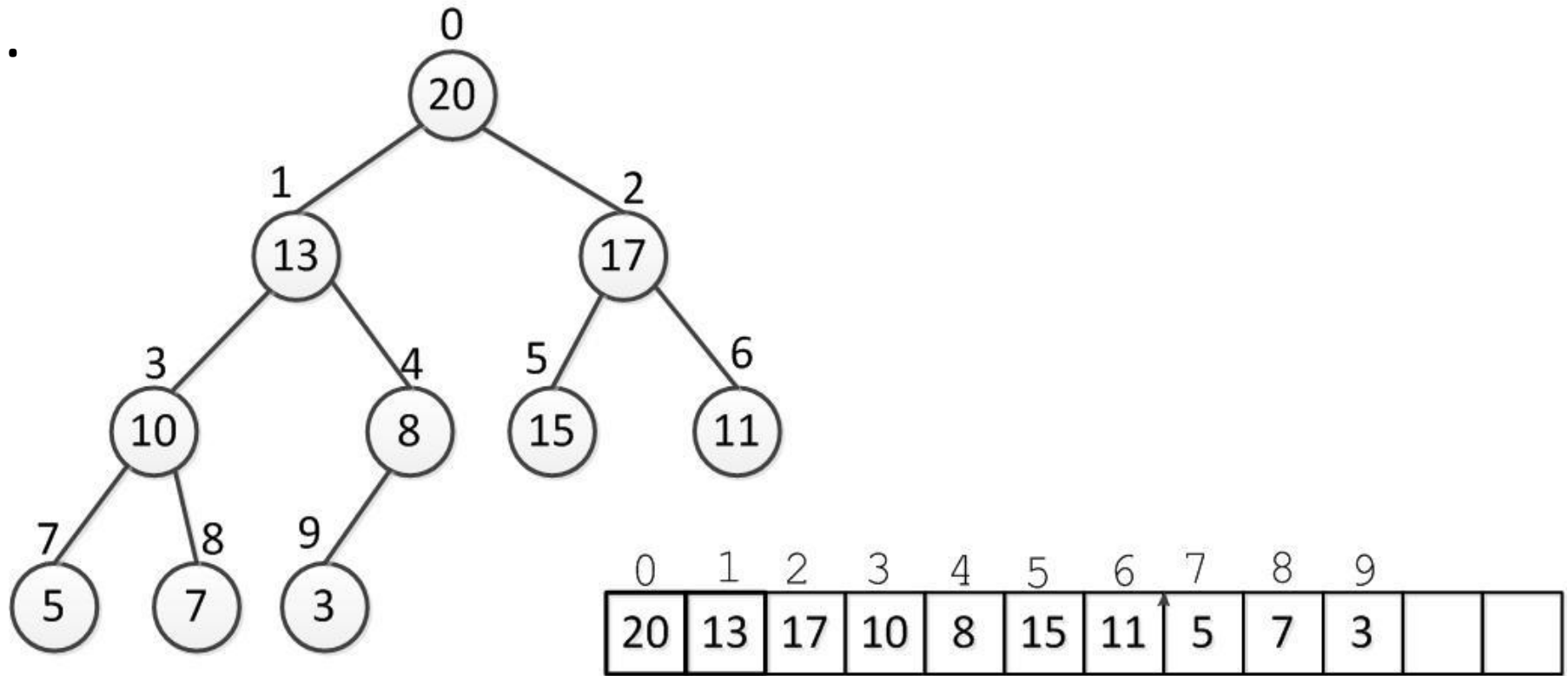


max heap



min heap

Элементы бинарной кучи организованы таким образом, что приоритет вершины не ниже приоритета каждого из ее сыновей.



невозрастающая пирамида (*max-heapproperty*)

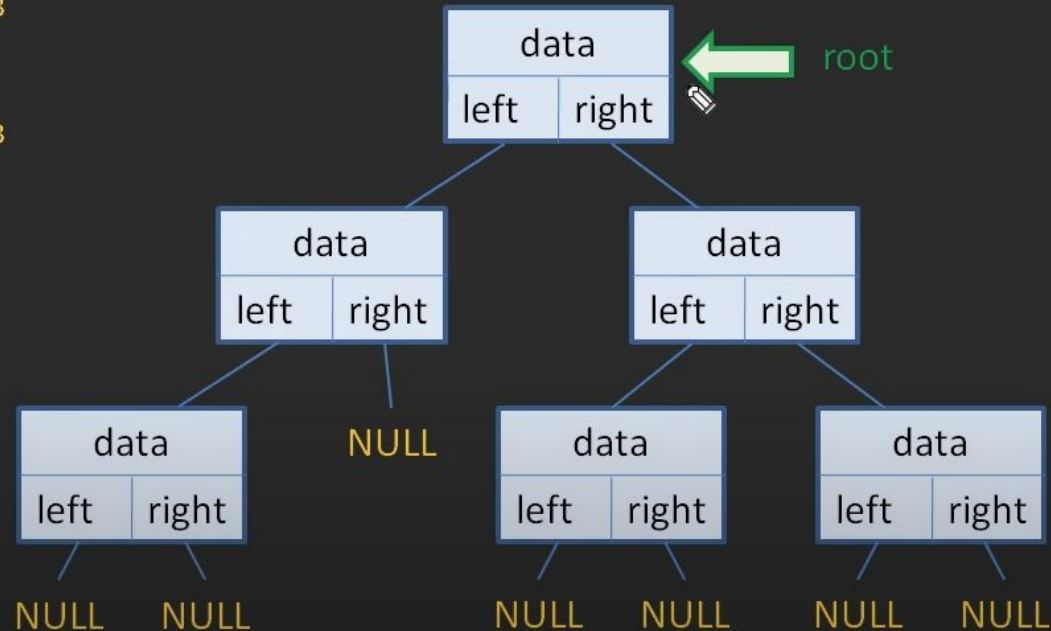
Сыновьями вершины с индексом i являются вершины с индексами $2 \cdot i + 1$ и $2 \cdot i + 2$. А родитель имеет индекс $(i-1)/2$, где i – индекс «ребёнка».

Бинарные кучи (пирамиды)

Пирамида – это структура данных, представляющая собой объект-массив, который можно рассматривать как почти полное бинарное дерево. Его можно реализовывать не только на массивах, но и списках.

Способы обхода вершин двоичного дерева

- обход вершин дерева в ширину (breadth-first)
- обход вершин дерева в глубину

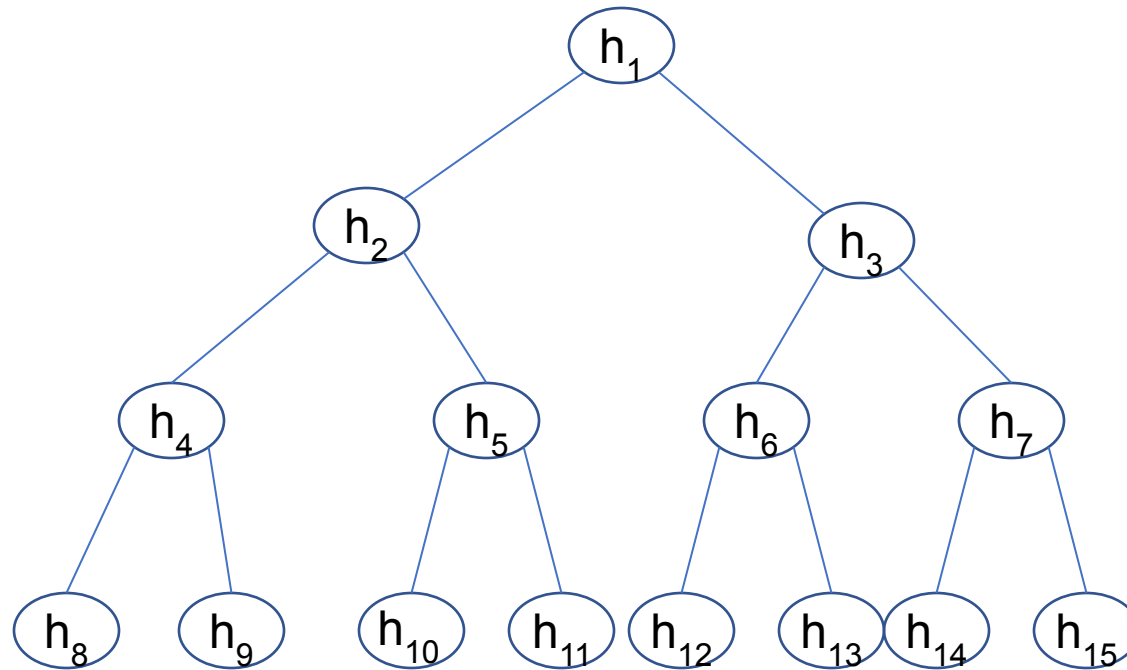


Полная пирамида при $n=15$

Пусть дан набор элементов $h[1..15]$

Полная пирамида может быть изображена в виде корневого бинарного дерева, в котором элементы h_{2i+1} и h_{2i+2} являются сыновьями элемента h_i .

Элемент в любом узле численно не меньше всех своих потомков, а вершина полной пирамиды h_1 содержит максимальный элемент всей последовательности.



Пример полной пирамиды при $n = 12$

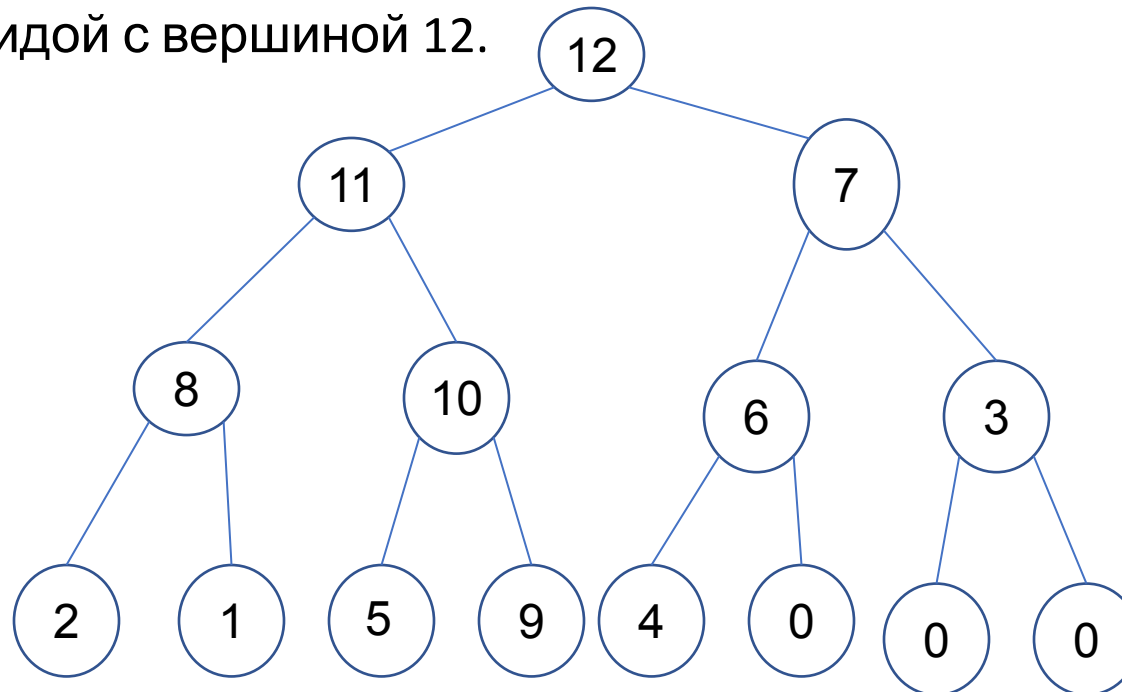
Если число элементов в полной пирамиде не равно $2^k - 1$, самый нижний уровень дерева будет неполным: недостающих сыновей можно достроить, добавив в пирамиду несколько заключительных «минимальных» элементов «0», не нарушающих условия пирамиды.

Последовательность, упорядоченная по убыванию/возрастанию, является полной пирамидой.

Например, последовательность из 12 элементов

12, 11, 7, 8, 10, 6, 3, 2, 1, 5, 9, 4

является полной пирамидой с вершиной 12.



Основные операции над элементами пирамиды

Пусть A – массив, на котором построена куча (пирамида)

length $[A]$ – количество элементов массива

heap_size $[A]$ – количество элементов пирамиды,
содержащиеся в куче A

Parent (i)
 return $(i-1) / 2$

Left (i)
 return $2 \cdot i + 1$

Right (i)
 return $2 \cdot i + 2$

Реализация функций на языке Си.

Для вершины i вычисляются позиции левого и правого сыновей.

```
int left(int i)          | левый сын вершины i  
{  
    return i << 1 + 1;  
}
```

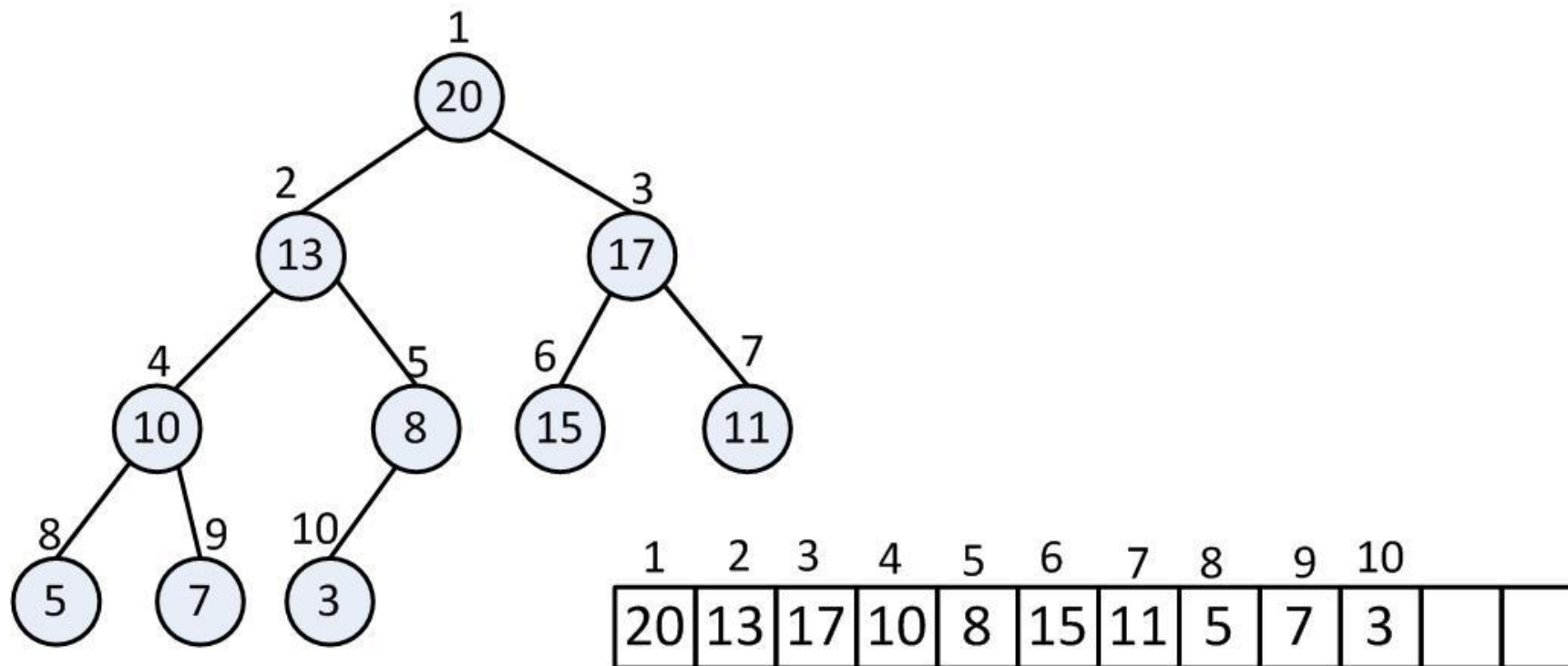
```
int right(int i)         | правый сын вершины i  
{  
    return (i + 1) << 1;  
}
```


Отцом вершины с индексом i является вершина с индексом $(i-1)/2$, у корневой вершины отца нет.

```
int parent(int i)    | отец вершины i  
{  
    return (i - 1) >> 1;  
}
```

Высота пирамиды, h , определяется как высота его корня и равна $O(\log n)$, где n – количество элементов в пирамиде. Время исполнения основных операций в пирамиде пропорционально высоте дерева.

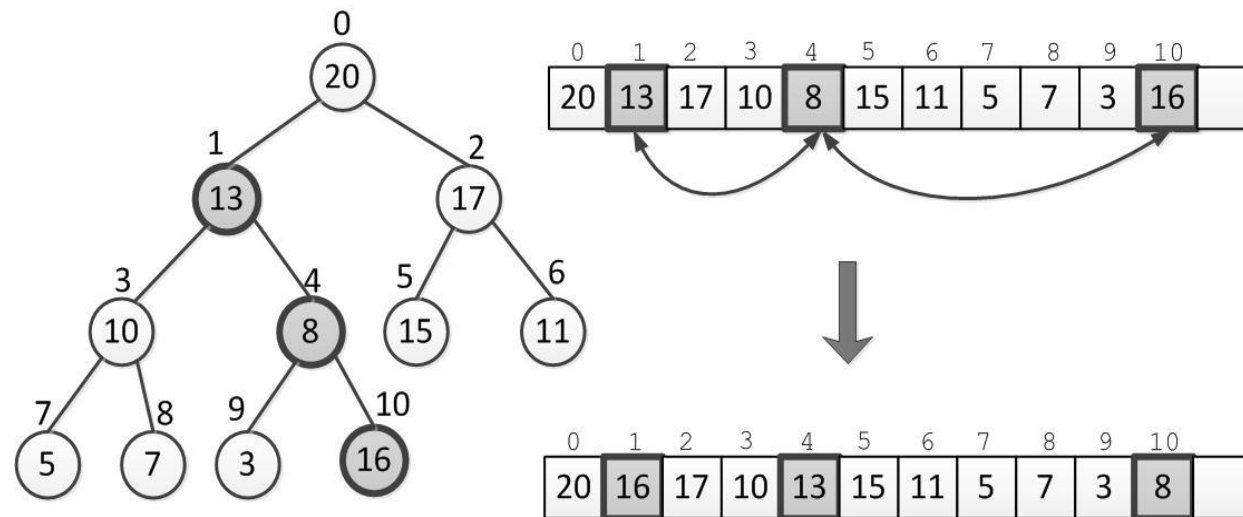
Свойство кучи (пирамиды)



- Свойство **невозрастающих пирамид** (max-heap property)
 $A[\text{Parent}(i)] \geq A[i]$
- Свойство **неубывающих пирамид** (min-heap property)
 $A[\text{Parent}(i)] \leq A[i]$
- Нижний ряд заполняется слева на право!

Основные методы

Insert (вставка) элемента в пирамиду



Процедура *Heap_Insert* (*x*)

$i \leftarrow \text{heap_size}[a]$

$a[i] \leftarrow x$

$\text{heap_size}[a] \leftarrow \text{heap_size}[a] + 1$

Sift_Up(*i*)

конец процедуры

$\text{Parent} = (i - 1) / 2 = 4$

Sift_Up(*i*)

if $i = 1$ return

if $A[i] \geq A[\text{Parent}(i)]$

Swap(*i*, Parent(*i*))

Sift_Up (Parent(*i*))

Процедура *Heap_Insert* вызывает в процессе своей работы процедуру *Sift_Up(i)*, которая «просеивает вверх» по пирамиде элемент из вершины с номером *i*, если он не удовлетворяет свойству пирамиды.

Процедура *Sift_Up(i)*

Если $i = 0$ то выход

**Если $a[i] \geq a[parent(i)]$ то
начало**

$a[i] \leftrightarrow a[parent(i)]$ *// обмен значениями*

Sift_Up(parent(i))

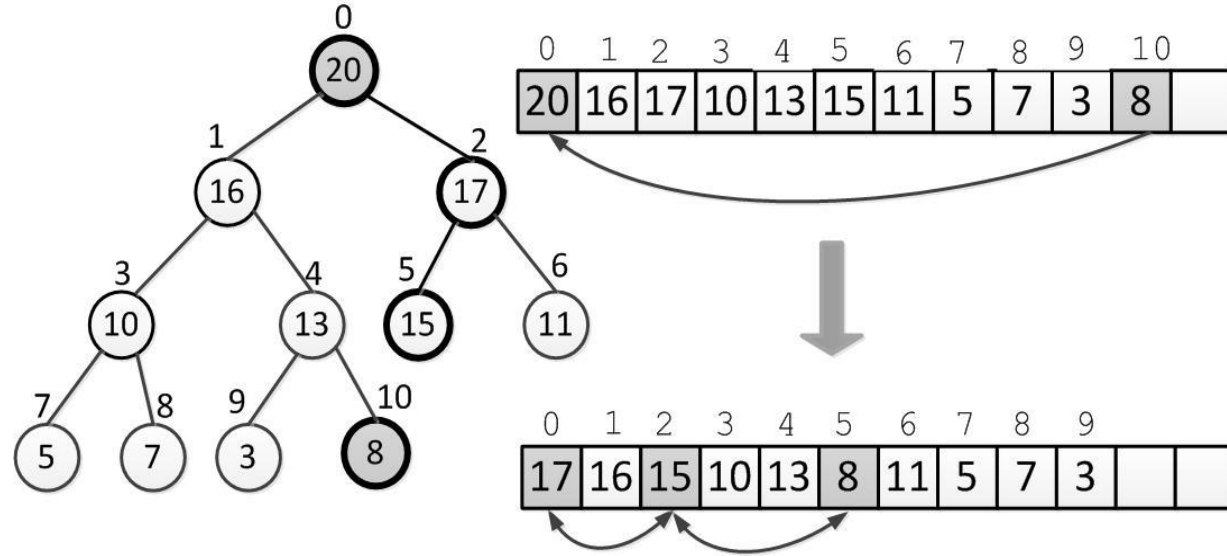
конец

Конец процедуры

Удаление максимального элемента из

ПЯРАМІДІ

$O(\log_2 N)$



функция *Heap_Extract*(*a*) : ключ

| *извлечение максимального элемента из кучи*

Если *heap_size*[*a*] < 1 **то**

выдать -1 | *куча пуста*

max ← *a*[0]

a[0] ← *a*[*heap_size*[*a*]]

heap_size[*a*] ← *heap_size*[*a*] - 1

Sift_Down(*a*, 0)

Выдать *max*

Конец функции

«Просеивание вниз» для обеспечения сохранения свойств кучи.

процедура *Sift_Down* (*a*, *i*) | просеивание элемента с
номером *i* по пирамиде *a*

l ← *left*(*i*)

r ← *right*(*i*)

Если *l* < *heap_size*[*a*] **и** *a*[*l*] > *a*[*i*] **то**

largest ← *l*

иначе

largest ← *i*

Если *r* < *heap_size*[*a*] **и** *a*[*r*] > *a*[*largest*] **то**

largest ← *r*

Если *largest* ≠ *i* **то**

начало

a[*i*] ↔ *a*[*largest*] | обмен значениями

Sift_Down(*a*, *largest*)

конец

конец процедуры

```
template <class T>
void Heap<T>:: heapify(int i) {
    int left, right;
    int temp;
    left = 2*i+1;
    right = 2*i+2;
    if(left < heapSize) {
        if(h[i] < h[left]) {
            temp = h[i];
            h[i] = h[left];
            h[left] = temp;
            heapify(left);
        }
    }
    if(right < heapSize) {
        if(h[i] < h[right]) {
            temp = h[i];
            h[i] = h[right];
            h[right] = temp;
            heapify(right);
        }
    }
}
```

Операции над кучей

```
Heap_Maximum(A)  
    return A[1]
```

```
Heap_Extract_Max(A)  
    if heap_size[A] < 1  
        then error "нет элементов"  
    max ← A[1]  
    A[1] ← A[heap_size[A]]  
    heap_size[A] ← heap_size[A] - 1  
    Sift_Down(A, 1)  
    return max
```


Sift_Down(A, i)

$l \leftarrow \text{Left}(i)$

$r \leftarrow \text{Right}(i)$

if $l \leq \text{heap_size}[A]$ и $A[l] > A[i]$

 then $\text{largest} \leftarrow l$

 else $\text{largest} \leftarrow i$

if $r \leq \text{heap_size}[A]$ и $A[r] > A[\text{largest}]$

 then $\text{largest} \leftarrow r$

if $\text{largest} \neq i$

 then Обменять $A[i] \leftrightarrow A[\text{largest}]$

 Sift_Down(A, largest)

Структуры

Ассоциативные таблицы

Каждый элемент – (k, v)

k – ключ

v – значение

Операции:

$i \leftarrow \text{Insert}(k)$ - вставить ключ k

$\text{Remove}(i)$ - удалить ключ с индексом i

$k \leftarrow \text{Get_Min}()$ - взять миним. ключ

$k \leftarrow \text{Extract_Min}()$ - удалить мин. ключ

$i' \leftarrow \text{Decrease_Key}(i, k)$ - уменьшить ключ

В

массивах:

$O(1)$

$O(1)$

$O(N)$

$O(N)$

$O(1)$

Очереди с приоритетами

Очередь с приоритетами (priority queue) – это абстрактный тип данных, предназначенный для обслуживания множества S , с каждым элементом которого связано определенное значение, называемое ключом (key).

В **невозрастающей** очереди с приоритетами поддерживаются следующие операции:

- $\text{Insert}(S, x)$
- $\text{Maximum}(S)$
- $\text{Extract_Max}(S)$
- $\text{Increase_Key}(S, x, k)$

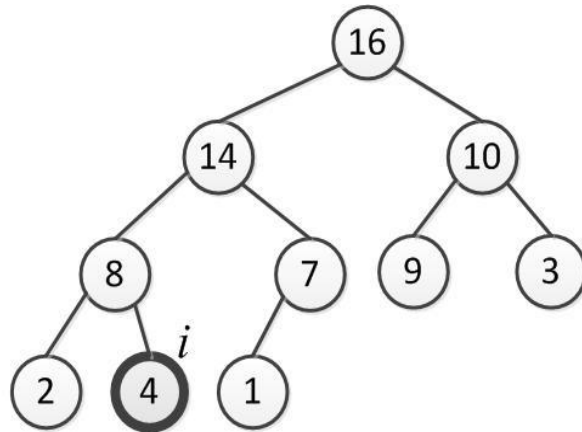
В **неубывающей** очереди с приоритетами поддерживаются следующие операции:

- $\text{Insert}(S, x)$
- $\text{Minimum}(S)$
- $\text{Extract_Min}(S)$
- $\text{Decrease_Key}(S, x, k)$

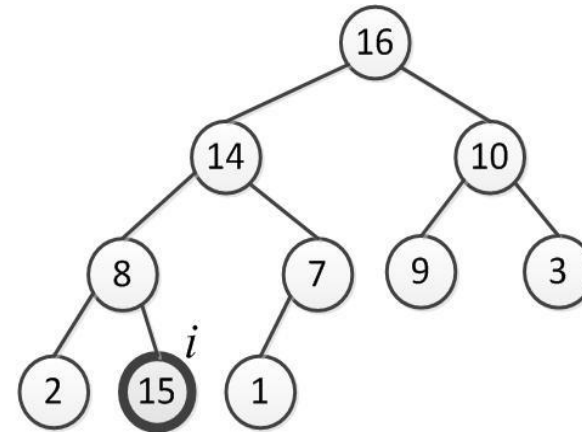
Реализация очереди с приоритетами с помощью пирамиды

```
Heap_Increase_Key (a, i, key)  
{  
  if key < a[ i ]  
    return error | Новый ключ меньше текущего  
  a[ i ]  $\leftarrow$  key; | обновление ключа  
  while i > 1 & a[ parent( i ) ] < a[ i ]  
    do a[ i ]  $\leftrightarrow$  a[ parent( i ) ]  
    i  $\leftarrow$  parent( i )  
}
```

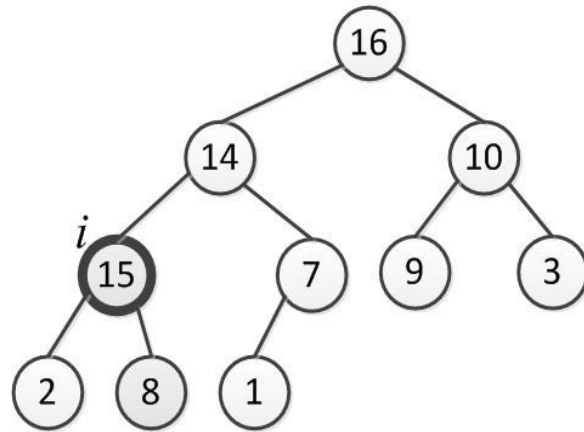
Пример: увеличение ключа



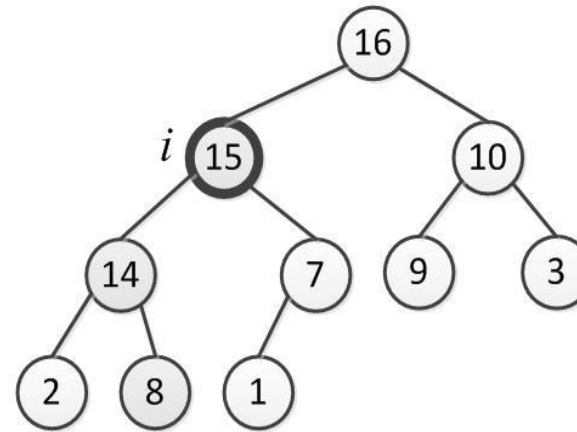
a)



б)



в)



г)

Вставка элемента в очередь

Max_Heap_Insert(a, key)

heap_size[a] \leftarrow heap_size[a] + 1;

a[heap_size[a]] \leftarrow MINVALUE

Heap_Increase_Key (a, heap_size [a], key);

Пятиминутка

Построить невозрастающую кучу,
добавляя в нее элементы в указанной
последовательности:

18 1 19 5 2 16 0 14 9 4 11

Выполнить действия:

Удалить максимальный

Вставить число 3

Увеличить ключ: новое значение =

17

старый индекс = 4

Напечатать результат:

Куча = _____

Значение максимального = _____


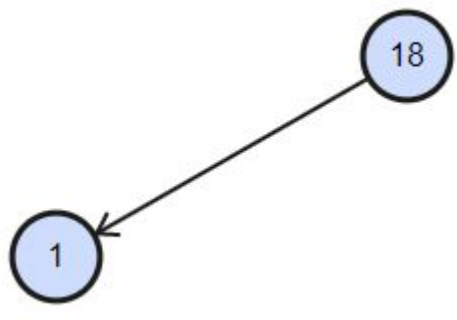
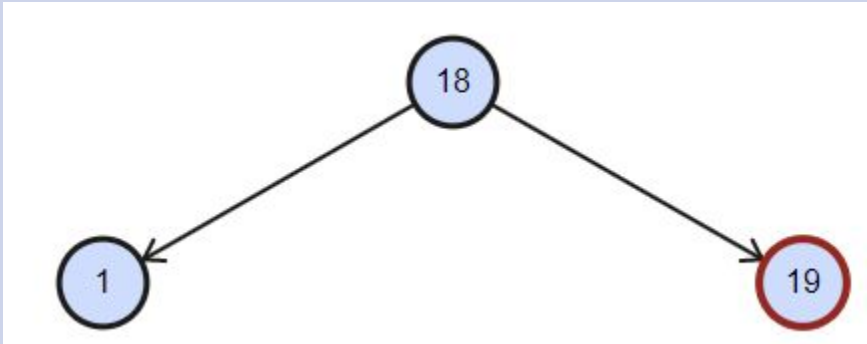
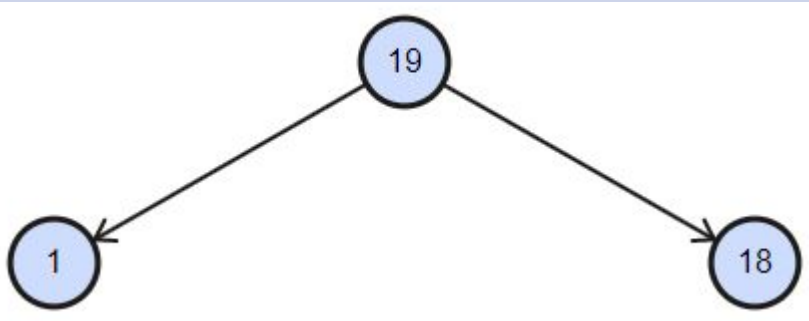
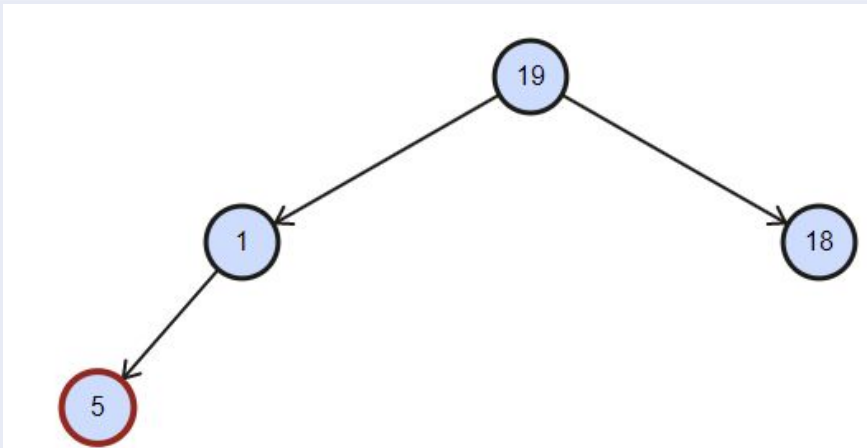
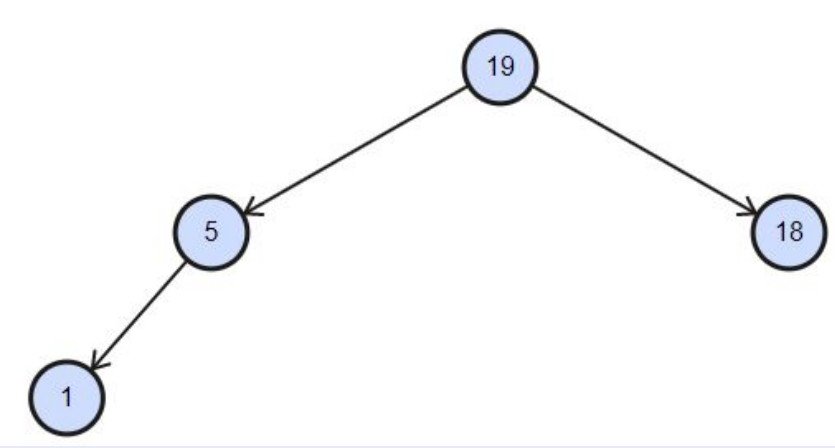
Куча = _____

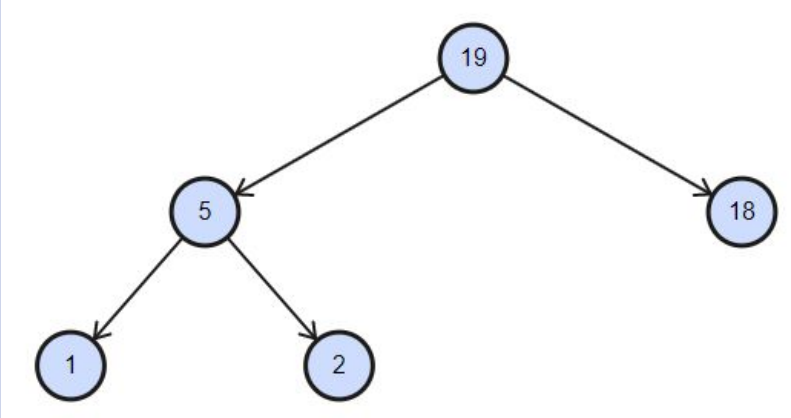
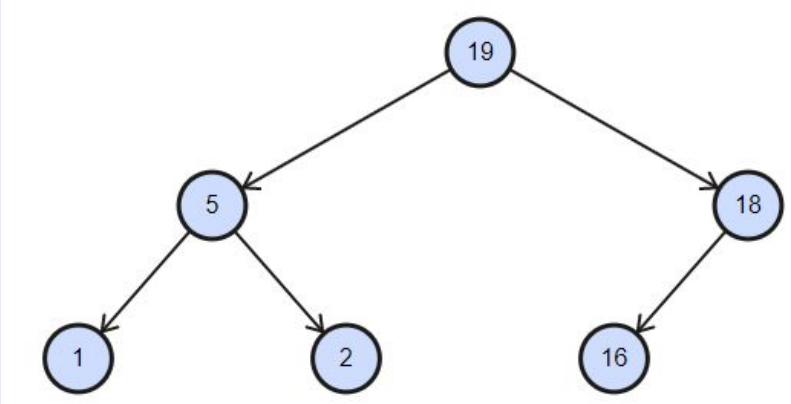
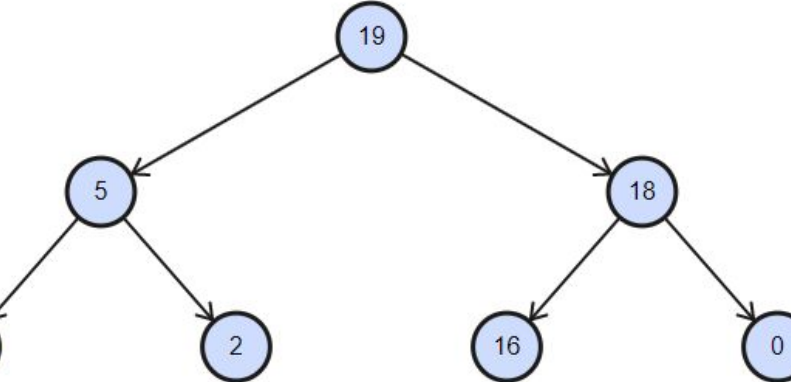
Индекс (приоритет) его = _____

Куча = _____

Новый приоритет (индекс) = _____

Куча = _____

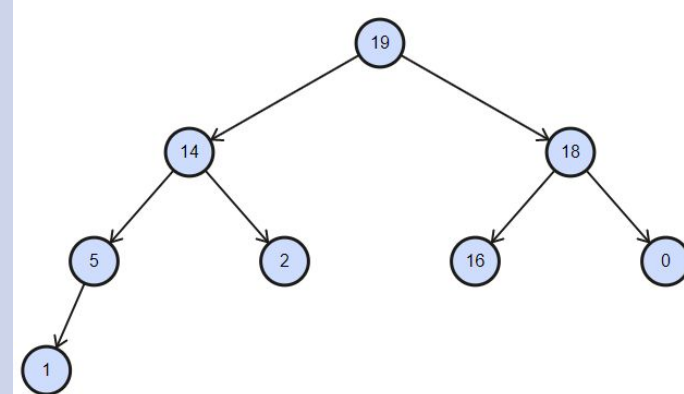
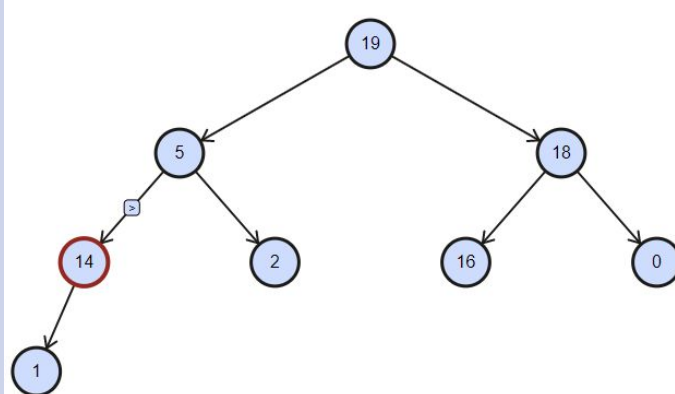
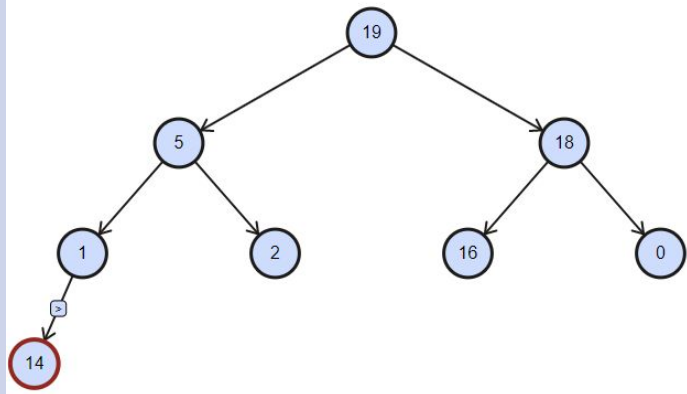
Элемент	КУЧА	
18		
1		
19	 <p>просеиваем</p>	
5	 <p>просеиваем</p>	

Элемент	КУЧА		
2		 <pre>graph TD; 19((19)) --> 5((5)); 19 --> 18((18)); 5 --> 1((1)); 5 --> 2((2));</pre>	
16		 <pre>graph TD; 19((19)) --> 5((5)); 19 --> 18((18)); 5 --> 1((1)); 5 --> 2((2)); 18 --> 16((16));</pre>	
0		 <pre>graph TD; 19((19)) --> 5((5)); 19 --> 18((18)); 5 --> 1((1)); 5 --> 2((2)); 18 --> 16((16)); 18 --> 0((0));</pre>	

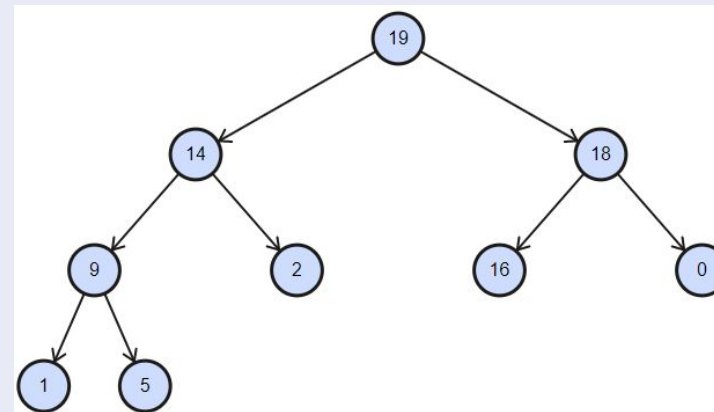
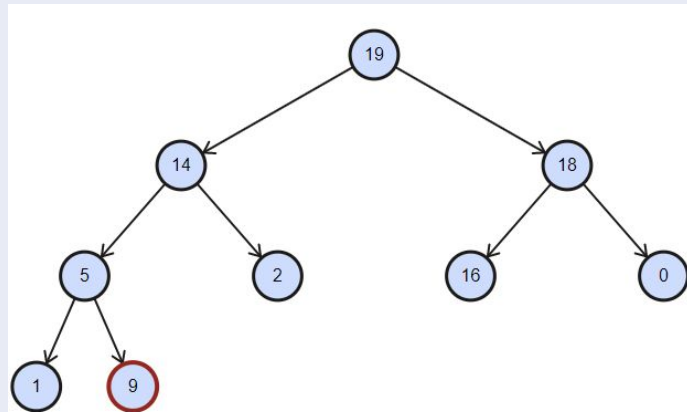
Элемент

КУЧА

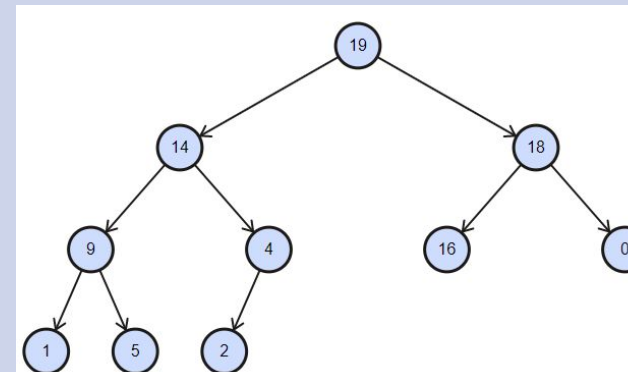
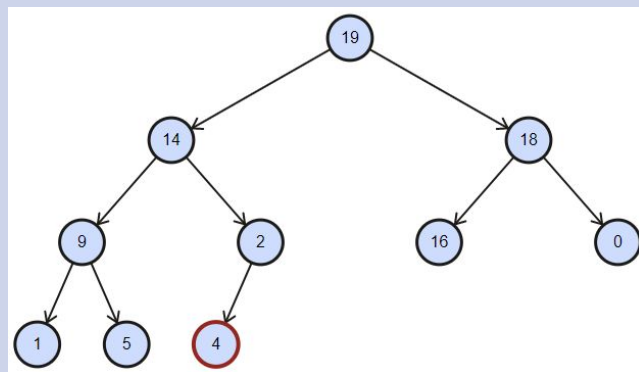
14



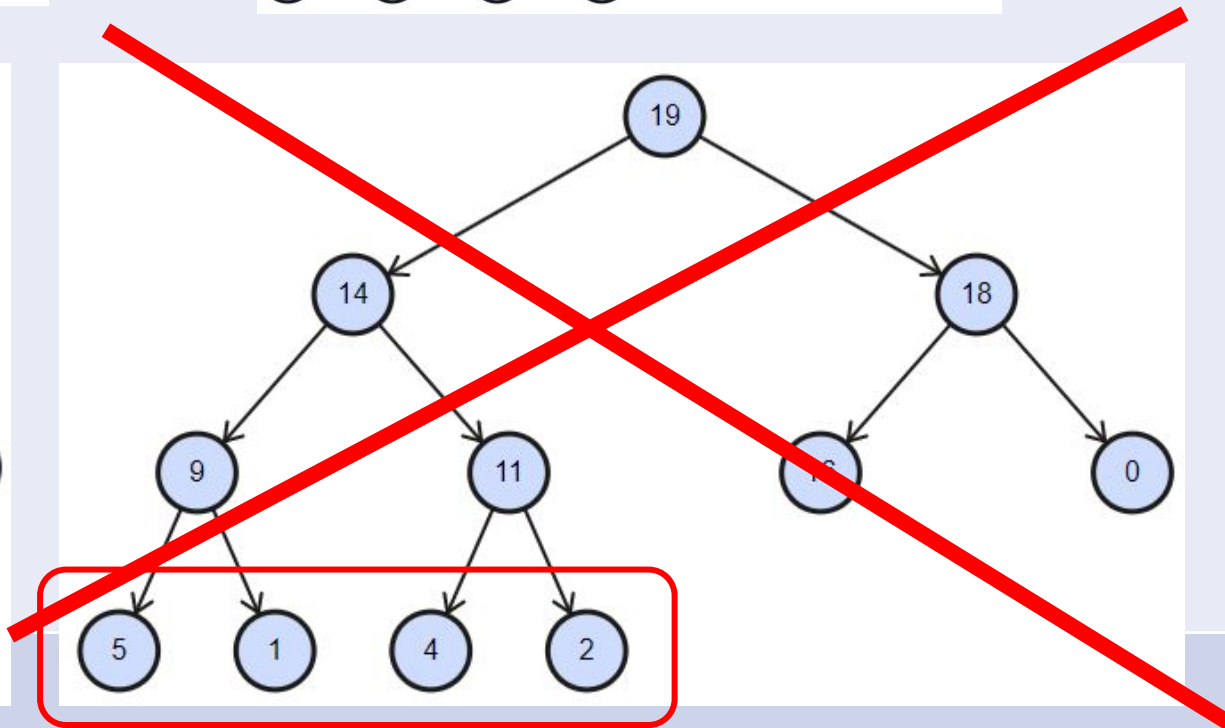
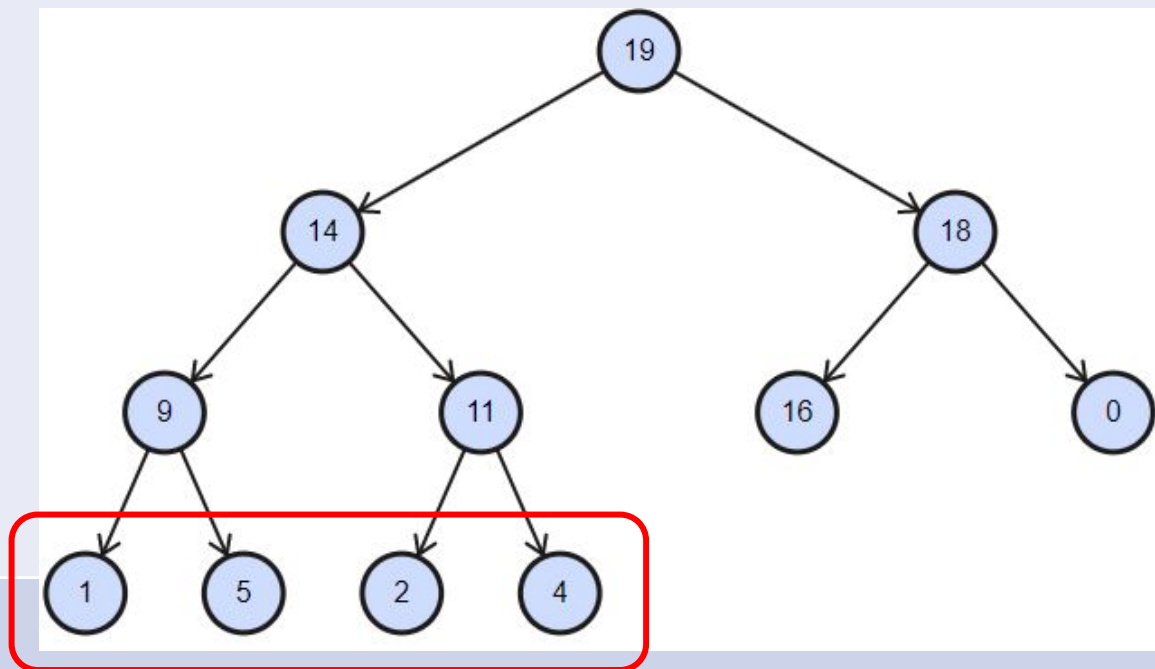
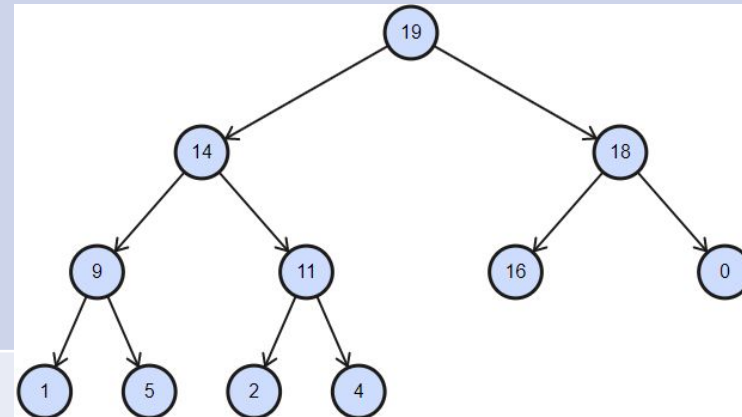
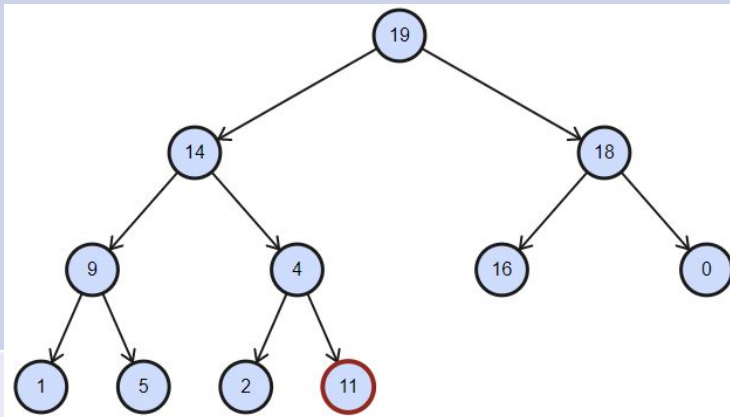
9



4



11



ОТВЕТ: 19 14 18 9 11 16 0 1 5 2 4; Максимальный элемент 19

Пятиминутка

Построить невозрастающую кучу,
добавляя в нее элементы в указанной
последовательности:

18 1 19 5 2 16 0 14 9 4 11

Выполнить действия:

Удалить максимальный

Вставить число 3

Увеличить ключ: новое значение =

17

старый индекс = 4

Напечатать результат:

Куча = 19 14 18 9 11 16 0 1 5 2 4

Значение максимального = 19

Куча = _____

Индекс (приоритет) его = _____

Куча = _____

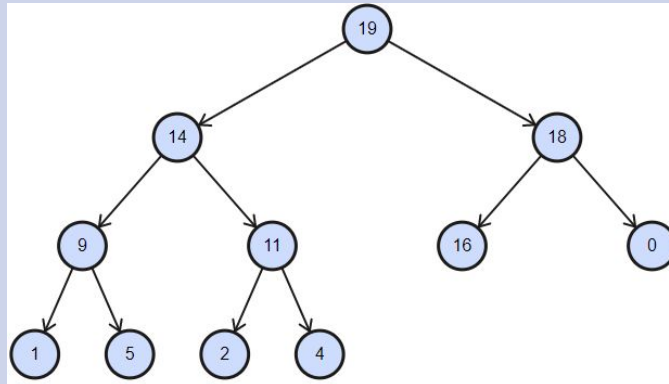
Новый приоритет (индекс) = _____

Куча = _____

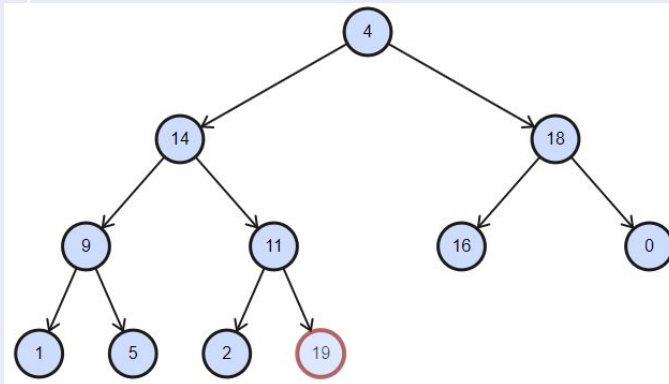
Действие

КУЧА

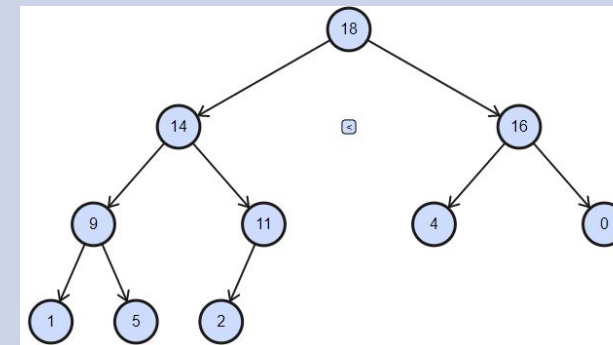
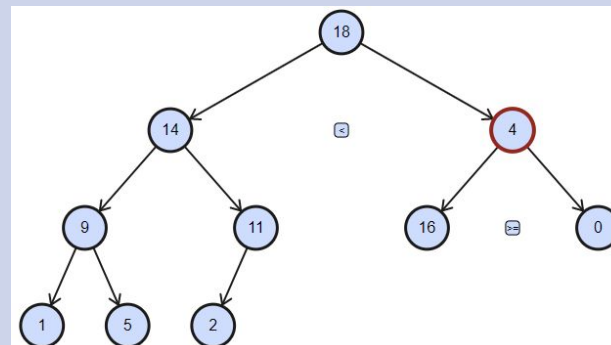
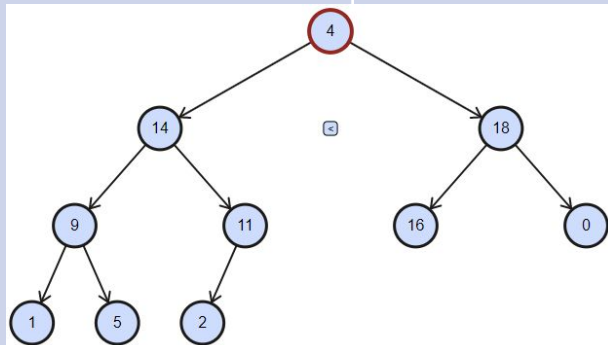
Удалить
максимальный
элемент



Исходная «Куча»



Меняем местами корневой и последний элементы,
старый корневой, потом, исключаем, «Кучу» - снова
упорядочиваем



ОТВЕТ:
18 14 16 9 11 4 0 1 5 2

Пятиминутка

Построить невозрастающую кучу,
добавляя в нее элементы в указанной
последовательности:

18 1 19 5 2 16 0 14 9 4 11

Выполнить действия:

Удалить максимальный

Вставить число 3

Увеличить ключ: новое значение =

17

старый индекс = 4

Напечатать результат:

Куча = 19 14 18 9 11 16 0 1 5 2 4

Значение максимального = 19

Куча = 18 14 16 9 11 4 0 1 5 2

Индекс (приоритет) его = _____

Куча = _____

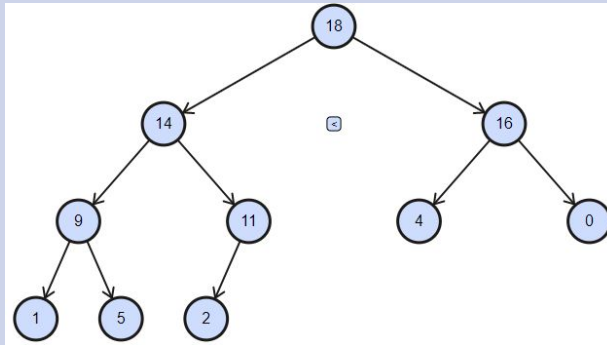
Новый приоритет (индекс) = _____

Куча = _____

Действие

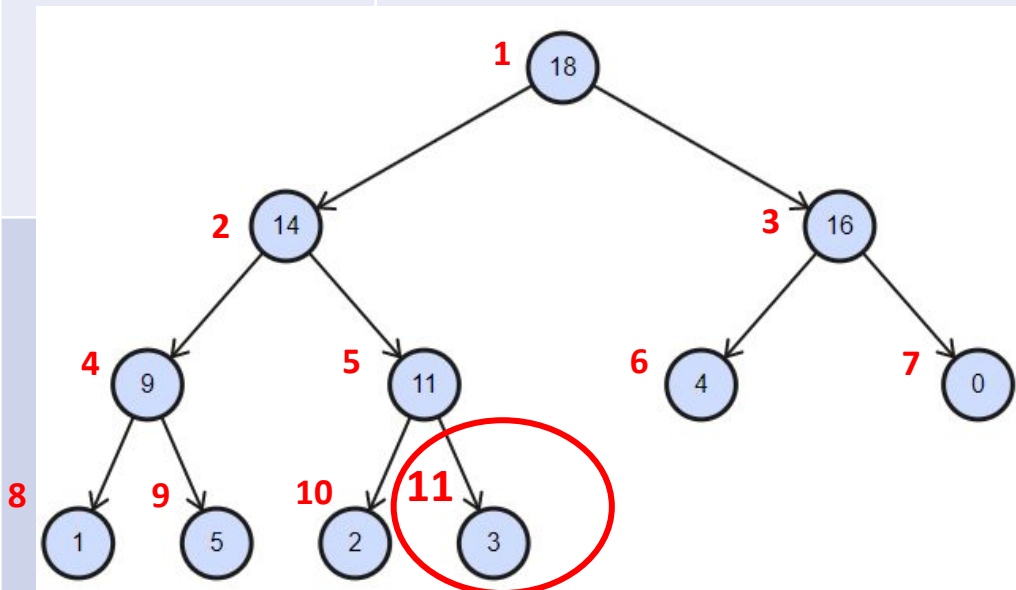
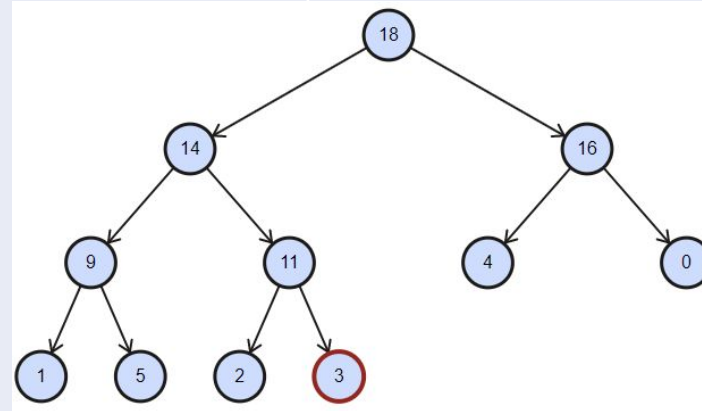
КУЧА

Вставить
новый
элемент,
число
3



Исходная «Куча»

Вставляем элемент,
«Кучу» - снова упорядочиваем



ОТВЕТ: 18 14 16 9 11 4 0 1 5 2 3, индекс = 11

Пятиминутка

Построить невозрастающую кучу,
добавляя в нее элементы в указанной
последовательности:

18 1 19 5 2 16 0 14 9 4 11

Выполнить действия:

Удалить максимальный

Вставить число 3

Увеличить ключ: новое значение =

17

старый индекс = 4

Напечатать результат:

Куча = 19 14 18 9 11 16 0 1 5 2 4

Значение максимального = 19

Куча = 18 14 16 9 11 4 0 1 5 2

Индекс (приоритет) его 11

Куча = 18 14 16 9 11 4 0 1 5 2 3

Новый приоритет (индекс) = _____

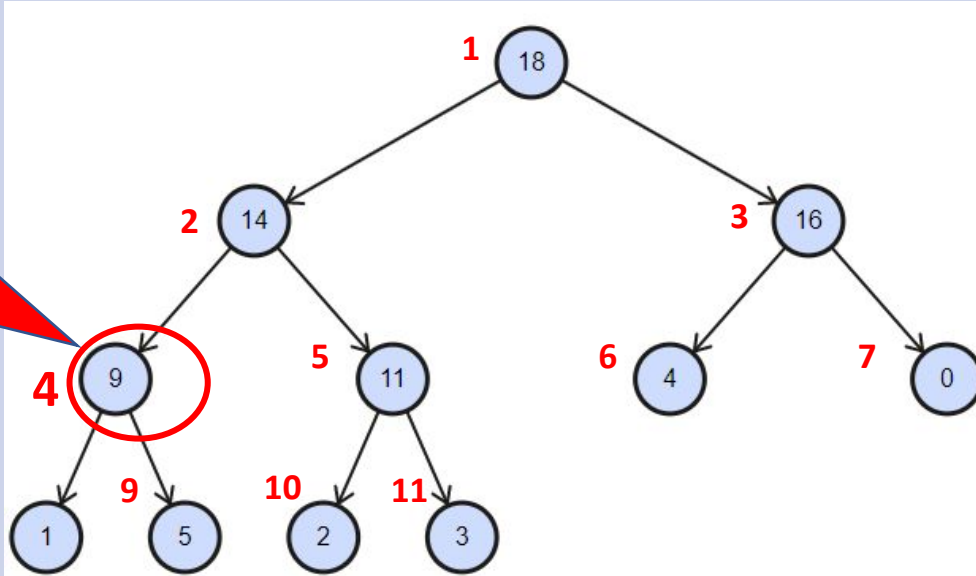
Куча = _____

Действие

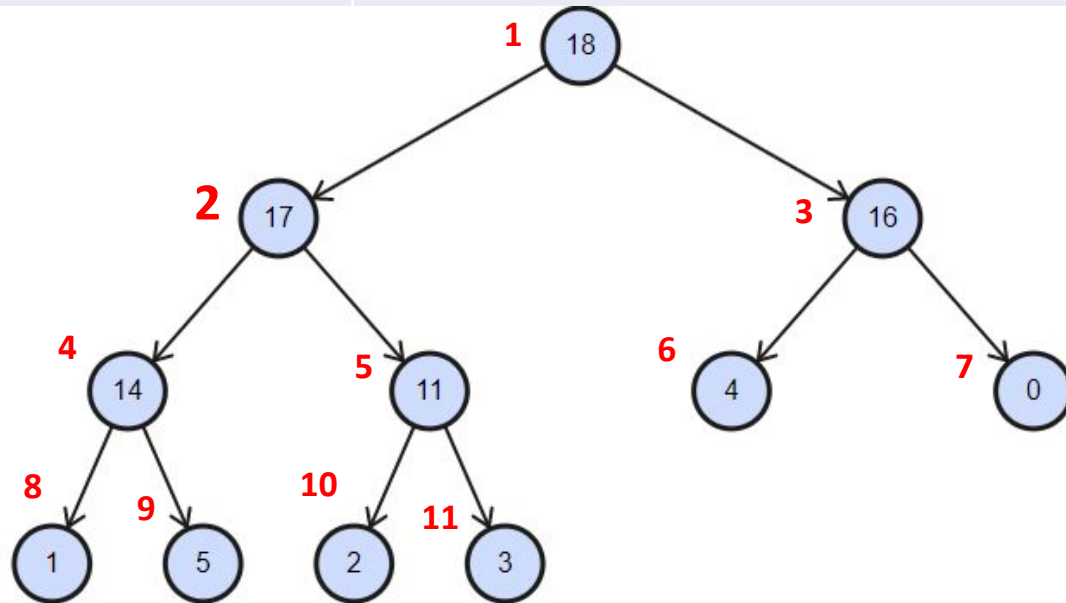
КУЧА

Увеличить
ключ: новое
значение

17,
старый
индекс **4**



Исходная «Куча»



Заменяем элемент,
«Кучу» - снова упорядочиваем

ОТВЕТ: 18 17 16 14 11 4 0 1 5 2 3, индекс = 2

Пятиминутка

Построить невозрастающую кучу,
добавляя в нее элементы в указанной
последовательности:

18 1 19 5 2 16 0 14 9 4 11

Выполнить действия:

Удалить максимальный

Вставить число 3

Увеличить ключ: новое значение =

17

старый индекс = 4

Напечатать результат:

Куча = 19 14 18 9 11 16 0 1 5 2 4

Значение максимального = 19

Куча = 18 14 16 9 11 4 0 1 5 2

Индекс (приоритет) его ¹¹ =

Куча = 18 14 16 9 11 4 0 1 5 2 3

Новый приоритет (индекс)² =

Куча = 18 17 16 14 11 4 0 1 5 2 3

```
#ifndef HEAP_H
#define HEAP_H

#include <iostream>
using namespace std;

class Heap {
    static const int SIZE = 100; // максимальный размер кучи
    int *h; // указатель на массив кучи
    int HeapSize; // размер кучи
public:
    Heap(); // конструктор кучи
    ~Heap();
    void insert(int); // добавление элемента кучи
    void outHeap(); // вывод элементов кучи в форме кучи
    void out(); // вывод элементов кучи в форме массива
    int getmax(); // удаление вершины (максимального элемента)
    void heapify(int); // упорядочение кучи
};

#endif
```

```
template <class T>
class Heap {
    int size;
    T *h;           // указатель на массив кучи
    int heapSize; // размер кучи
public:
    Heap(int); // конструктор кучи
    ~Heap();
    void insert(const T &n); // добавление элемента кучи
    void outHeap(); // вывод элементов кучи в форме кучи
    void out(); // вывод элементов кучи в форме массива
    int getMax(); // удаление вершины (максимального элемента)
    void heapify(int); // упорядочение кучи
    T & operator [](int) {
        return h[i];
    }
    const T & operator [](int i) const {
        return h[i];
    }
};
```

```
template <class T>
Heap<T>::Heap(int size) {
    heapSize = 0;
    h = new T[this -> size = size];
}
```

```
template <class T>
Heap<T> :: ~Heap() {
    if(h) delete h;
    cout << "distructor" << endl;
}
```

```
typedef struct {  
    int value;  
    int key;  
} ITEM;
```

```
class HEAP {  
    ITEM *h;  
    int size;  
public:  
    ....  
};
```



```

]void Heap :: insert(int n) {
    int i, parent;
    i = HeapSize;
    h[i] = n;
    parent = (i-1)/2;
    while(parent >= 0 && i > 0) {
        if(h[i] > h[parent]) {
            int temp = h[i];
            h[i] = h[parent];
            h[parent] = temp;
        }
        i = parent;
        parent = (i-1)/2;
    }
    HeapSize++;
}

```

```

template <class T>
void Heap<T> :: insert(const T &n) {
    if (heapSize >= size) {
        throw NoMoreSpase();
    }
    int i = heapSize;
    int parent;
    while(i > 0 && n > h[parent = (i-1)/2]) {
        h[i] = h[parent];
        i = parent;
    }
    h[i] = n;
    heapSize++;
}

```

```
template <class T>
int Heap<T>:: getmax() {
    int x;
    x = h[0];
    h[0] = h[heapSize-1];
    heapSize--;
    heapify(0);
    return(x);
}
```


Обработка исключений

```
#ifndef NOMORESPASE_H
#define NOMORESPASE_H

#include <exception>
#include <iostream>
using namespace std;

class NoMoreSpase : public exception {
    char * msg;
public:
    NoMoreSpase() : msg("No more spase") {}
    const char * what() const throw() {
        return msg;
    }
};

#endif
```

```
#include "heap.h"

int main() {
    int count;
    cout << "Input count element ";
    cin >> count;
    Heap<int> heap(count);
    int k;
    for(int i=0; i<=count; i++) {
        cout << "Input element " << i+1 << ": ";
        cin >> k;
        try {
            heap.insert(k);
        } catch (NoMoreSpace &ex) {
            cout << "Got an exception: " << ex.what() << endl;
        }
    }
    cout << endl << "Heap:" << endl;
    heap.outHeap();
    cout << endl;
    heap.out();
    cout << endl << "Max element: " << heap.getmax();
    cout << endl << "New heap:" << endl;
    heap.outHeap();
    cout << endl;
    heap.out();
    cout << endl << "Max element: " << heap.getmax();
    cout << endl << "New heap:" << endl;
    heap.outHeap();
    cout << endl;
    heap.out();
    return 0;
}
```

Вывод кучи

```
template <class T>
void Heap<T>:: outHeap() {
    int i = 0;
    int k = 1;
    while(i < heapSize) {
        while((i < k) && (i < heapSize)) {
            cout << h[i] << " ";
            i++;
        }
        cout << endl;
        k = k * 2 + 1;
    }
}

template <class T>
void Heap<T>:: out() {
    for(int i=0; i< heapSize; i++) {
        cout << h[i] << " ";
    }
    cout << endl;
}
```

C:\Windows\system32\cmd.exe

```
Input count element 6
Input element 1: 43
Input element 2: 6
Input element 3: 4
Input element 4: 90
Input element 5: 674
Input element 6:
3
Input element 7: 5
Got an exception: No more space
```

Heap:

674

90 4

6 43 3

674 90 4 6 43 3

Max element: 674

New heap:

90

43 4

3 6

90 43 4 3 6