

Лекция 9

Бинарные кучи (пирамиды)

Курносов Михаил Георгиевич

E-mail: mkurnosov@gmail.com

WWW: www.mkurnosov.net

Курс «Структуры и алгоритмы обработки данных»

Сибирский государственный университет телекоммуникаций и информатики (Новосибирск)

Весенний семестр, 2016

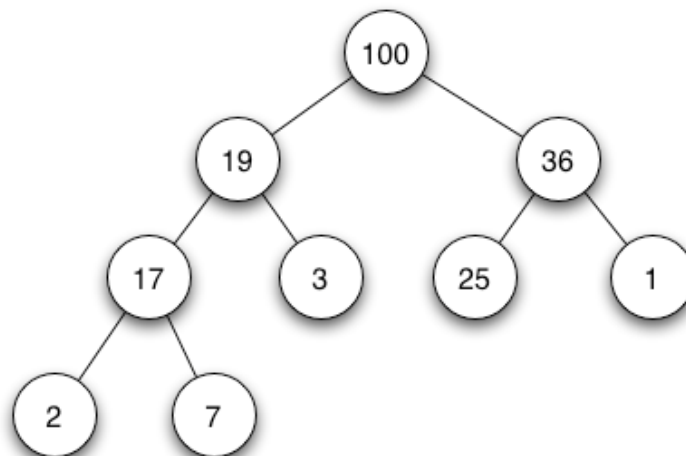
Очередь с приоритетом (priority queue)

- **Очередь с приоритетом (priority queue)** – очередь, в которой элементы имеют приоритет (вес); первым извлекается элемент с наибольшим приоритетом (ключом)
- **Поддерживаемые операции:**
 - ☐ **Insert** – добавление элемента в очередь
 - ☐ **Max** – возвращает элемент с максимальным приоритетом
 - ☐ **ExtractMax** – удаляет из очереди элемент с максимальным приоритетом
 - ☐ **IncreaseKey** – изменяет значение приоритета заданного элемента
 - ☐ **Merge** – сливает две очереди в одну

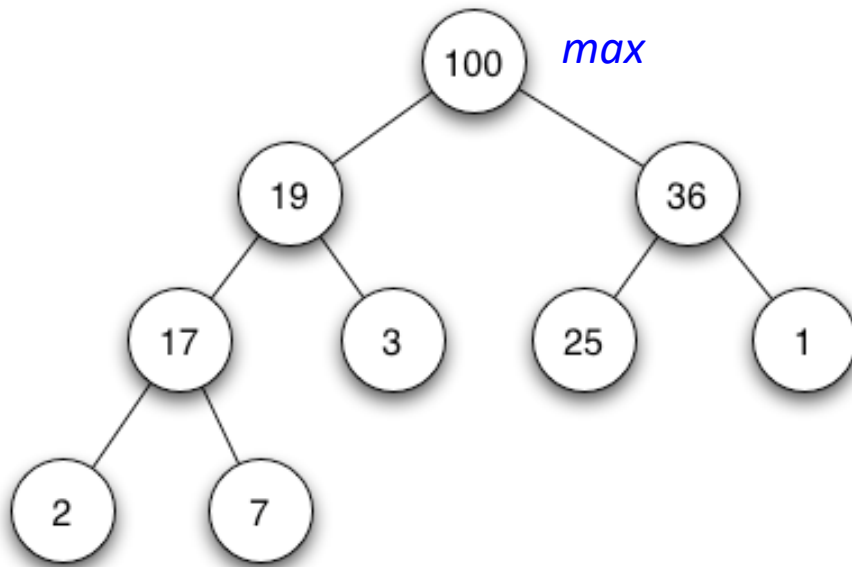
Значение (value)	Приоритет (priority)
Слон	3
Кит	1
Лев	15

Бинарная куча – пирамида (binary heap)

- Бинарная куча (пирамида, сортирующее дерево, binary heap) – это двоичное дерево, удовлетворяющее следующим условиям:
 - Приоритет любой вершины не меньше (\geq), приоритета ее потомков
 - Дерево является *полным двоичным деревом* (complete binary tree) – все уровни заполнены слева направо (возможно за исключением последнего)

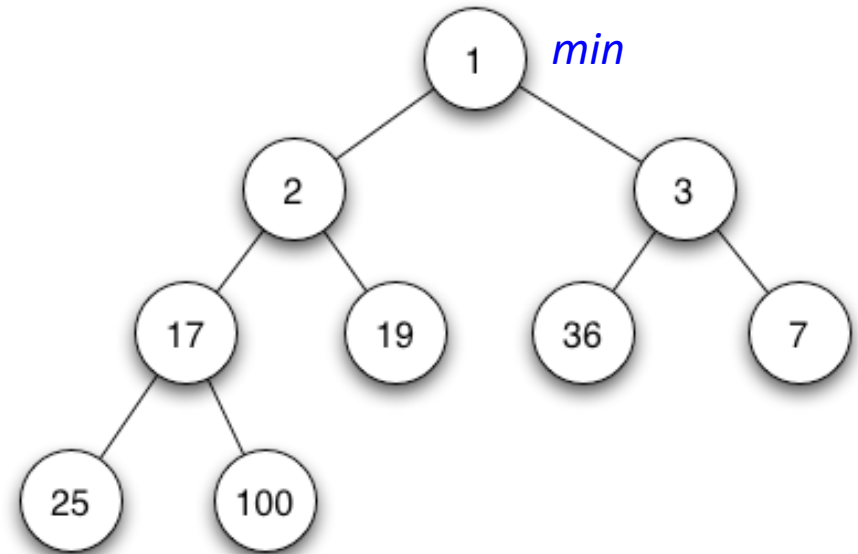


Бинарная куча – пирамида (binary heap)



Невозрастающая пирамида
max-heap

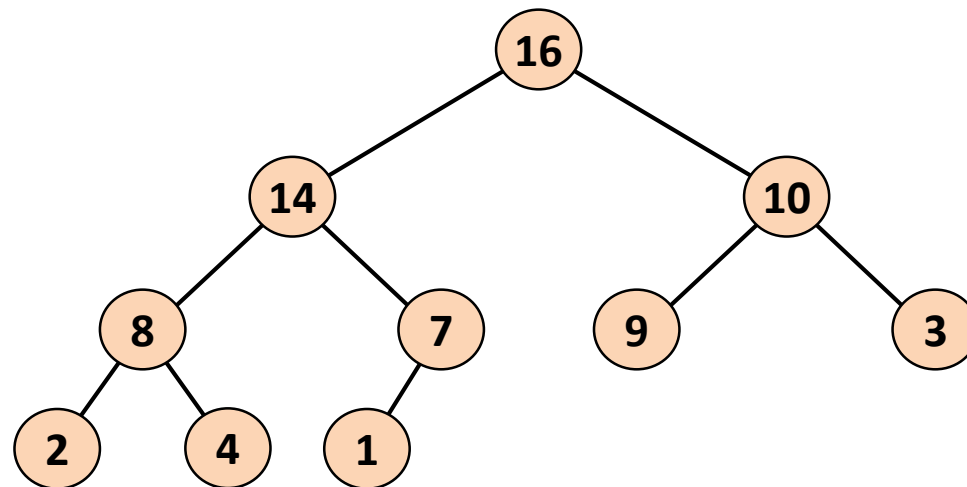
Приоритет любой вершины **не меньше (\geq)**,
приоритета потомков



Неубывающая пирамида
min-heap

Приоритет любой вершины **не больше (\leq)**,
приоритета потомков

Реализация бинарной кучи на основе массива



max-heap (10 элементов)

Массив $H[1..14]$ приоритетов (ключей):

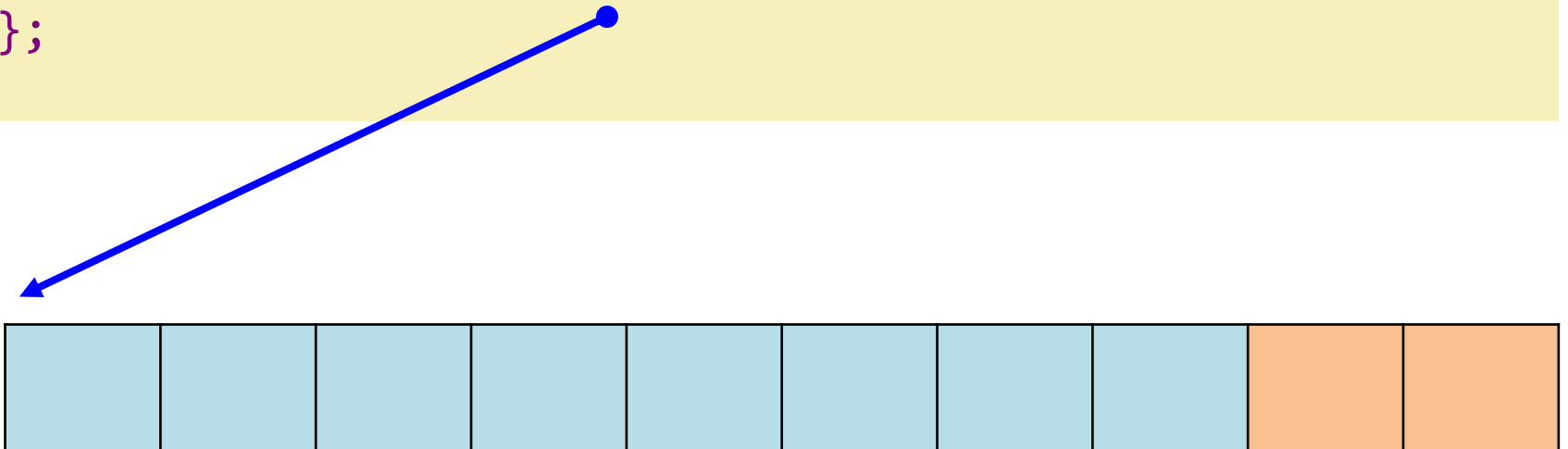
16	14	10	8	7	9	3	2	4	1				
----	----	----	---	---	---	---	---	---	---	--	--	--	--

- Корень дерева храниться в ячейке $H[1]$ – максимальный элемент
- Индекс родителя узла i : $Parent(i) = \lfloor i/2 \rfloor$
- Индекс левого дочернего узла: $Left(i) = 2i$
- Индекс правого дочернего узла: $Right(i) = 2i + 1$

$$H[Parent(i)] \geq H[i]$$

Реализация бинарной кучи на основе массива

```
struct heapnode {  
    int key;           /* Priority (key) */  
    char *value;       /* Data */  
};  
  
struct heap {  
    int maxsize;       /* Array size */  
    int nnodes;        /* Number of keys */  
    struct heapnode *nodes; /* Nodes: [0..maxsize] */  
};
```



Создание пустой кучи

```
struct heap *heap_create(int maxsize)
{
    struct heap *h;

    h = malloc(sizeof(*h));
    if (h != NULL) {
        h->maxsize = maxsize;
        h->nnodes = 0;
        /* Heap nodes [0, 1, ..., maxsize] */
        h->nodes = malloc(sizeof(*h->nodes) * (maxsize + 1));
        if (h->nodes == NULL) {
            free(h);
            return NULL;
        }
    }
    return h;
}
```

$$T_{Create} = O(1)$$

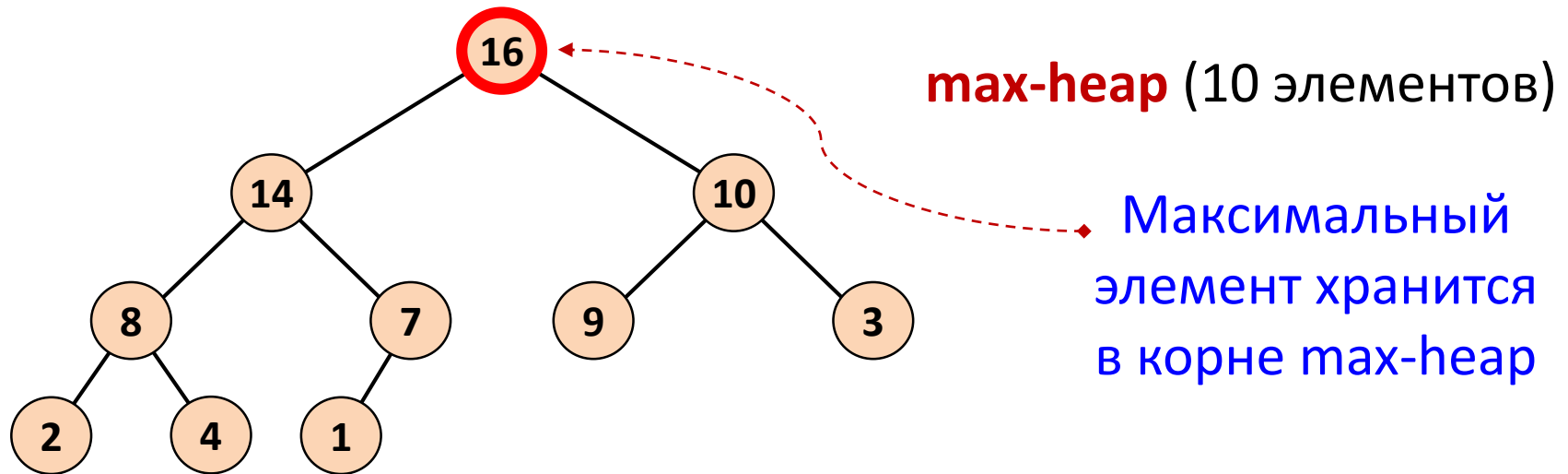
Удаление кучи

```
void heap_free(struct heap *h)
{
    free(h->nodes);
    free(h);
}

void heap_swap(struct heapnode *a,
               struct heapnode *b)
{
    struct heapnode temp;

    temp = *a;
    *a = *b;
    *b = temp;
}
```


Поиск максимального элемента



Массив $H[1..14]$ приоритетов (ключей):

16	14	10	8	7	9	3	2	4	1				
----	----	----	---	---	---	---	---	---	---	--	--	--	--

- Корень дерева храниться в ячейке $H[1]$ – максимальный элемент
- Индекс родителя узла i : $Parent(i) = \lfloor i/2 \rfloor$
- Индекс левого дочернего узла: $Left(i) = 2i$
- Индекс правого дочернего узла: $Right(i) = 2i + 1$

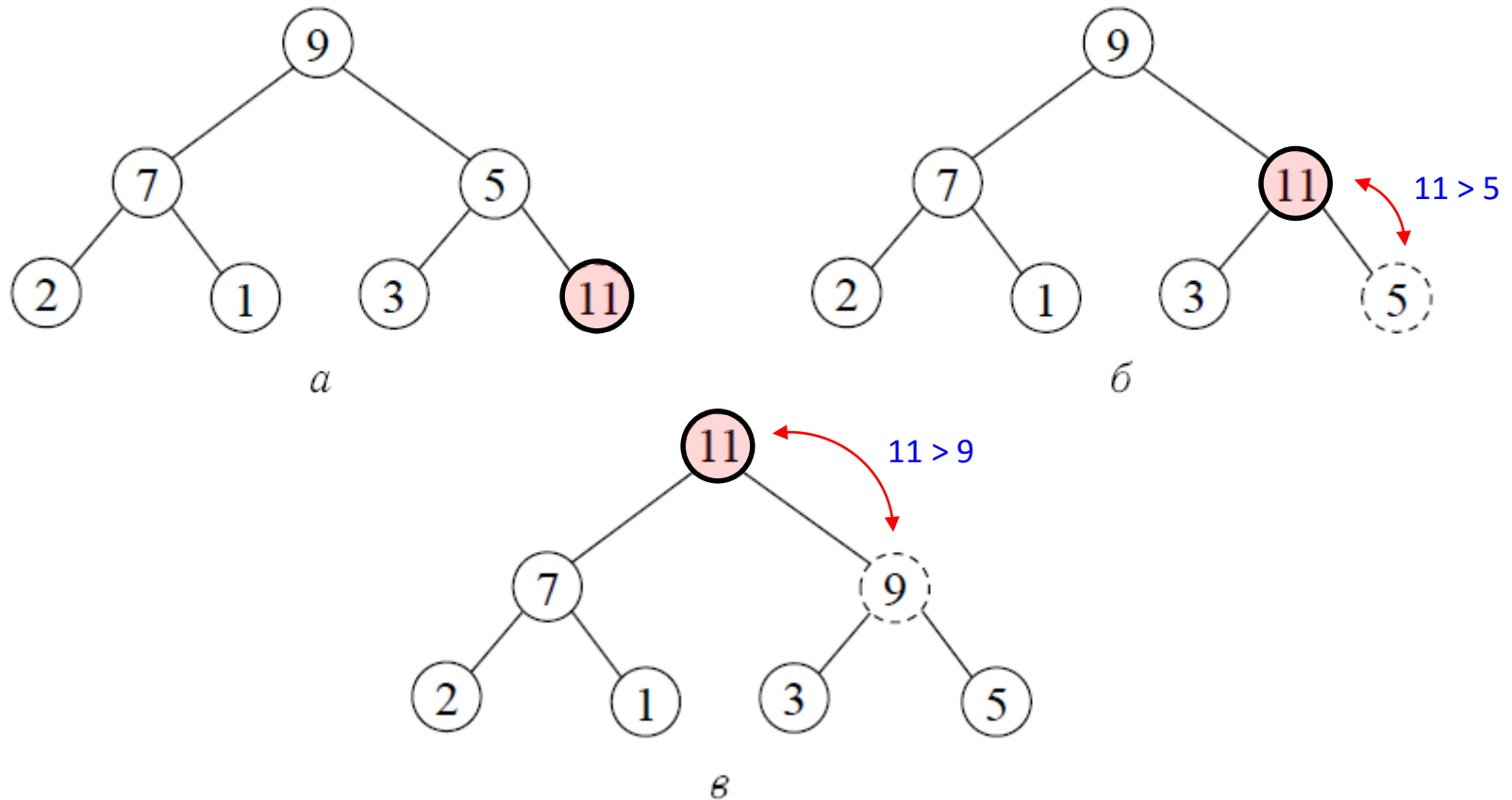
$$H[Parent(i)] \geq H[i]$$

Поиск максимального элемента

```
struct heapnode *heap_max(struct heap *h)
{
    if (h->nnodes == 0)
        return NULL;
    return &h->nodes[1];
}
```

$$T_{Max} = O(1)$$

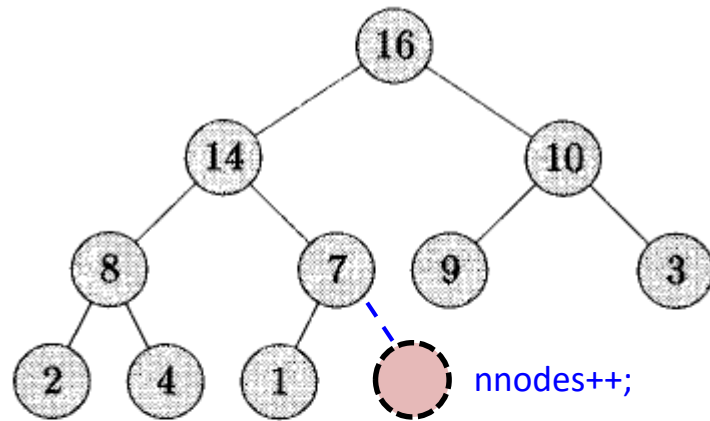
Вставка элемента в бинарную кучу (maxheap)



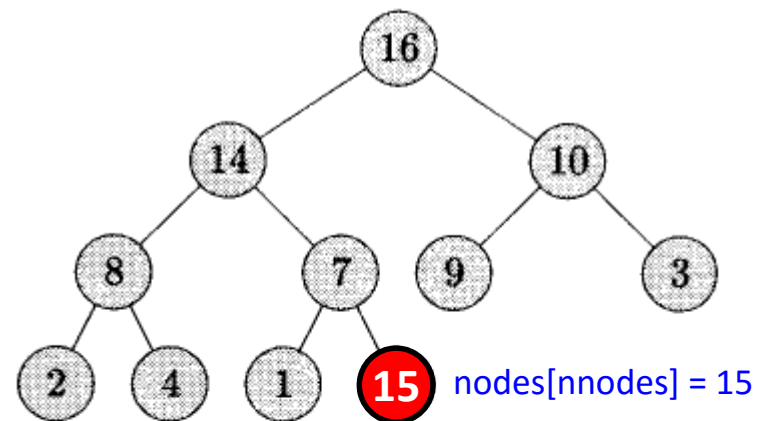
Вставка элемента с приоритетом 11
[DSABook, Глава 12]

$$T_{Insert} = O(\log n)$$

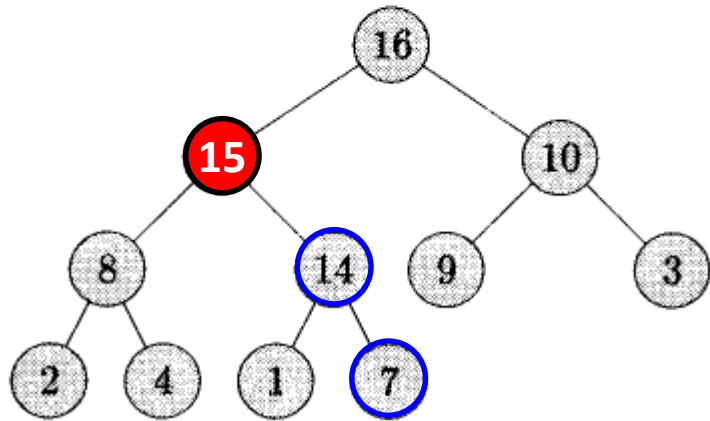
Вставка элемента в бинарную кучу



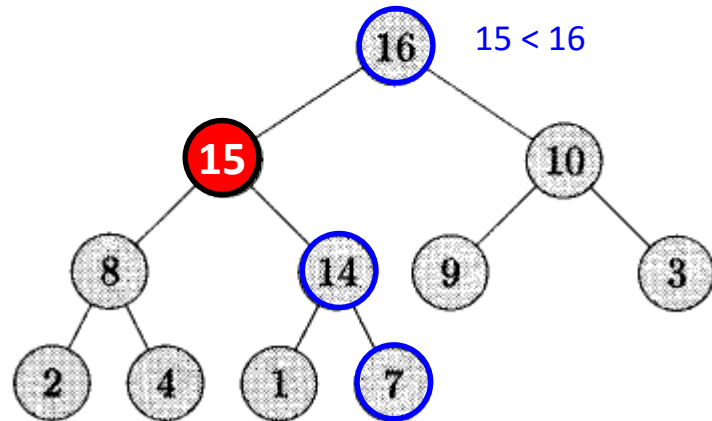
(a)



(б)



(в)



(г)

Вставка элемента с приоритетом 15 [CLRS, Глава 6]

Вставка элемента в бинарную кучу

```
1  function INSERT( $A[1..m]$ ,  $key$ ,  $value$ )
2      if  $n = m$  then
3          return HeapOverflow
4      end if
5       $n = n + 1$ 
6       $A[n].key = key$ 
7       $A[n].value = value$ 
8      HEAPIFYUP( $A$ ,  $n$ )
9  end function

10 function HEAPIFYUP( $A[1..m]$ ,  $i$ )
11     while  $i > 1$  and  $A[PARENT(i)].key < A[i].key$  do
12         SWAP( $A[i]$ ,  $A[PARENT(i)]$ )           /* Обмен значений узлов */
13          $i = PARENT(i)$ 
14     end while
15 end function
```

Вставка элемента в бинарную кучу

```
int heap_insert(struct heap *h, int key, char *value)
{
    if (h->nnodes >= h->maxsize) {
        /* Heap overflow */
        return -1;
    }

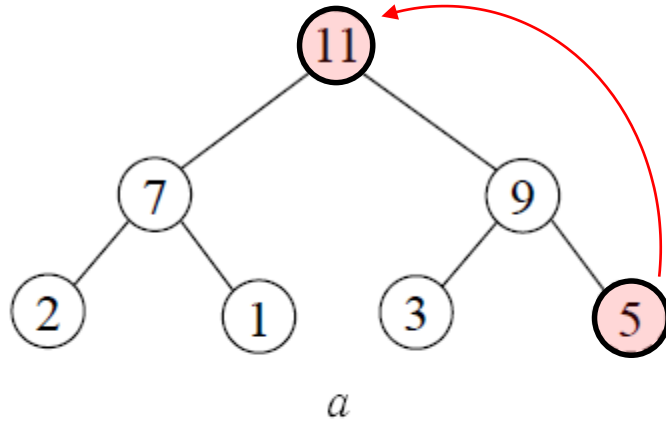
    h->nnodes++;
    h->nodes[h->nnodes].key = key;
    h->nodes[h->nnodes].value = value;

    // HeapifyUp
    for (int i = h->nnodes; i > 1 &&
        h->nodes[i].key > h->nodes[i / 2].key; i = i / 2)
    {
        heap_swap(&h->nodes[i], &h->nodes[i / 2]);
    }
    return 0;
}
```

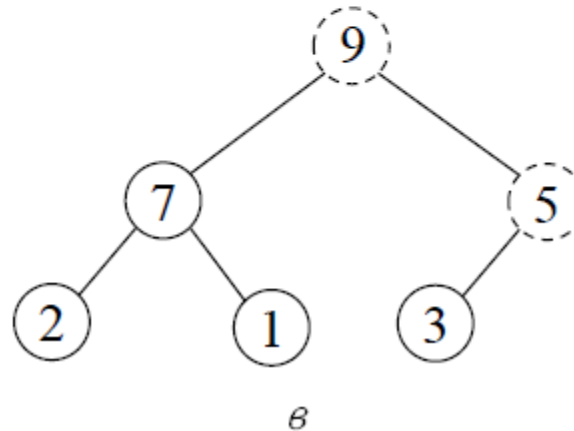
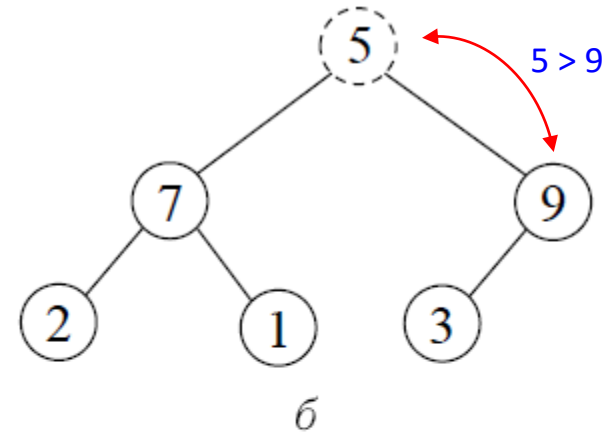
$$T_{Insert} = O(\log n)$$

Удаление максимального элемента

Заменяем корень $A[1]$ листом $A[n]$



HeapifyDown($A, 1$)



Удаление элемента с максимальным приоритетом 11
[DSABook, Глава 12]

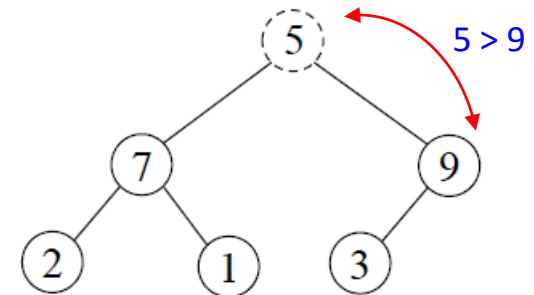
Удаление максимального элемента

```
1 function DELETEMAX(A[1..m])  
2   if  $n < 1$  then  
3     return HeapEmpty  
4   end if  
5    $max = A[1]$   
6    $A[1] = A[n]$   
7    $n = n - 1$   
8   HEAPIFYDOWN(A, 1)  
9   return  $max$   
10 end function
```

```
11 function HEAPIFYDOWN(A[1..m], i)  
12   while  $i \leq n$  do  
13      $left = LEFT(i)$   
14      $right = RIGHT(i)$   
15      $largest = i$   
16     if  $left \leq n$  and  $A[left].key > A[i].key$  then  
17        $largest = left$   
18     else if  $right \leq n$  and  $A[right].key > A[i].key$  then  
19        $largest = right$   
20     end if  
21     if  $largest \neq i$  then  
22       SWAP( $A[i]$ ,  $A[largest]$ )  
23        $i = largest$   
24     else  
25       break  
26     end if  
27   end while  
28 end function
```

/* Завершаем проход по дереву */

HeapifyDown(A, 1)



Удаление максимального элемента

```
struct heapnode heap_extract_max(struct heap *h)
{
    if (h->nnodes == 0)
        return (struct heapnode){0, NULL};

    struct heapnode maxnode = h->nodes[1];
    h->nodes[1] = h->nodes[h->nnodes];
    h->nnodes--;
    heap_heapify(h, 1);

    return maxnode;
}
```

Восстановление свойств кучи (max-heap)

```
void heap_heapify(struct heap *h, int index)
{
    for (;;) {
        int left = 2 * index;
        int right = 2 * index + 1;

        // Find largest key: A[index], A[left] and A[right]
        int largest = index;
        if (left <= h->nnodes &&
            h->nodes[left].key > h->nodes[index].key)
        { largest = left; }
        if (right <= h->nnodes &&
            h->nodes[right].key > h->nodes[largest].key)
        { largest = right; }

        if (largest == index)
            break;

        heap_swap(&h->nodes[index], &h->nodes[largest]);
        index = largest;
    }
}
```

$$T_{Heapify} = O(\log n)$$

Увеличение ключа в maxheap

```
1 function INCREASEKEY( $A[1..m], i, key$ )  
2   if  $A[i].key > key$  then  
3     return HeapInvalidKey    /* Новый ключ меньше текущего */  
4   end if  
5    $A[i].key = key$   
6   HEAPIFYUP( $A, i$ )              /* Восстанавливаем свойства кучи */  
7 end function
```

$$T_{Increase} = O(\log n)$$

Увеличение ключа в maxheap

```
int heap_increase_key(struct heap *h, int index, int key)
{
    if (h->nodes[index].key > key)
        return -1;

    h->nodes[index].key = key;
    for ( ; index > 1 &&
          h->nodes[index].key > h->nodes[index / 2].key;
          index = index / 2)
    {
        heap_swap(&h->nodes[index], &h->nodes[index / 2]);
    }
    return index;
}
```

$$T_{Increase} = O(\log n)$$

Построение бинарной кучи

- Дан неупорядоченный массив A длины n
- Требуется построить из его элементов бинарную кучу

Построение бинарной кучи из массива

- Дан неупорядоченный массив A длины n
- Требуется построить из его элементов бинарную кучу

```
function BuildHeap(A[1:n])  
    h = CreateBinaryHeap(n)           //  $O(1)$   
    for i = 1 to n do  
        HeapInsert(h, A[i], A[i])    //  $O(\log n)$   
    end for  
end function
```

$$T_{BuildHeap} = O(\textcolor{red}{n} \log n)$$

Построение кучи из массива за время $O(n)$

- Задан $A[1..m]$ массив элементов
- Требуется построить бинарную кучу

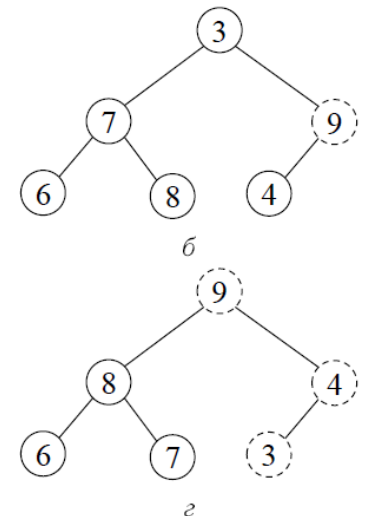
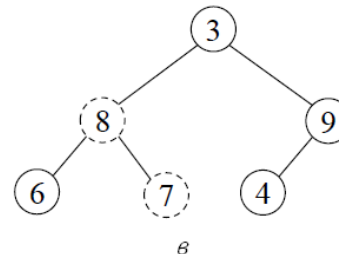
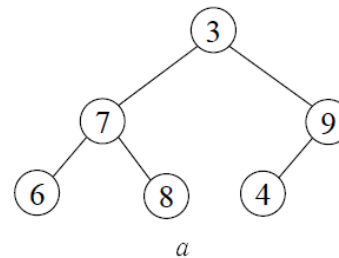
```
1 function BUILD_MAX_HEAP( $A[1..m], n$ )
2    $i = \lfloor n/2 \rfloor$ 
3   while  $i \geq 1$  do
4     HEAPIFY_DOWN( $A, i$ )
5      $i = i - 1$ 
6   end while
7 end function
```

- $A[6] = (3, 7, 9, 6, 8, 4)$
- BuildMaxHeap($A, 6$)

$$T(n) = \sum_{h=1}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=1}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h}\right)$$

$$\sum_{h=1}^{\lfloor \log_2 n \rfloor} h \left(\frac{1}{2}\right)^h < \sum_{h=0}^{\infty} h \left(\frac{1}{2}\right)^h = \frac{1/2}{(1 - 1/2)^2} = 2.$$

$$T(n) = O\left(n \sum_{h=1}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h}\right) = O(n).$$



Работа с бинарной кучей

```
int main()
{
    struct heap *h;
    struct heapnode node;

    h = heap_create(100);
    heap_insert(h, 16, "16");
    heap_insert(h, 14, "14");
    heap_insert(h, 10, "10");
    heap_insert(h, 8, "8");
    heap_insert(h, 7, "7");
    heap_insert(h, 9, "9");
    heap_insert(h, 3, "3");
    heap_insert(h, 2, "2");
    heap_insert(h, 4, "4");
    heap_insert(h, 1, "1");

    node = heap_extract_max(h);
    printf("Item: %d\n", node.key);

    int i = heap_increase_key(h, 9, 100);

    heap_free(h);
    return 0;
}
```


Сортировка на базе бинарной кучи

- На основе бинарной кучи можно реализовать алгоритм сортировки с вычислительной сложностью $O(n \log n)$ в худшем случае

Как ?

Сортировка на базе бинарной кучи

- На основе бинарной кучи можно реализовать алгоритм сортировки с вычислительной сложностью $O(n \log n)$ в худшем случае

Как ?

```
function HeapSort(v[1:n])
```

```
  h = CreateBinaryHeap(n)
```

```
  for i = 1 to n do
```

```
    HeapInsert(h, v[i], v[i])
```


```
  end for
```


```
  for i = 1 to n do
```


```
    v[i] = HeapRemoveMax(h)
```

```
  end for
```

```
end function
```


$$T_1 = O(1)$$


$$T_2 = O(\log n)$$


$$T_3 = O(\log n)$$

Сортировка на базе бинарной кучи

- На основе бинарной кучи можно реализовать алгоритм сортировки с вычислительной сложностью $O(n \log n)$ в худшем случае

Как ?

```
function HeapSort(v[1:n])
```

```
  h = CreateBinaryHeap(n)
```

```
  for i = 1 to n do
```

```
    HeapInsert(h, v[i], v[i])
```

```
  end for
```

```
  for i = 1 to n do
```

```
    v[i] = HeapRemoveMax(h)
```

```
  end for
```

```
end function
```

$$T_1 = O(1)$$

$$T_2 = O(\log n)$$

$$T_3 = O(\log n)$$

$$T_{\text{HeapSort}} = 1 + n \log n + n \log n = O(n \log n)$$

Сортировка на базе бинарной кучи

- На основе бинарной кучи можно реализовать алгоритм сортировки с вычислительной сложностью $O(n \log n)$ в худшем случае

Как ?

Алгоритм 12.1. Пирамидальная сортировка

```
1  function HEAPSORT( $A[1..n], n$ )
2      BUILDMAXHEAP( $A, n$ )
3       $i = n$ 
4      while  $i \geq 2$  do
5          SWAP( $A[1], A[i]$ )
6           $n = n - 1$ 
7          HEAPIFYDOWN( $A, 1$ )
8      end while
9  end function
```

$$T_{HeapSort} = O(n) + O((n - 1) \log n) = O(n \log n)$$

Очередь с приоритетом (priority queue)

- В таблице приведены трудоемкости операций очереди с приоритетом (в худшем случае, worst case)
- Символом '*' отмечена амортизированная сложность операций

Операция	Binary heap	Binomial heap	Fibonacci heap	Pairing heap	Brodal heap
FindMin	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
DeleteMin	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)^*$	$O(\log n)^*$	$O(\log n)$
Insert	$\Theta(\log n)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
DecreaseKey	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)^*$	$O(\log n)^*$	$\Theta(1)$
Merge/Union	$\Theta(n)$	$\Omega(\log n)$	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$

Домашнее чтение

- [DSABook, Глава 12]
- Анализа вычислительной сложности построения бинарной кучи (Build-Max-Heap) за время $O(n)$
 - [CLRS, Глава 6]