算法设计与分析

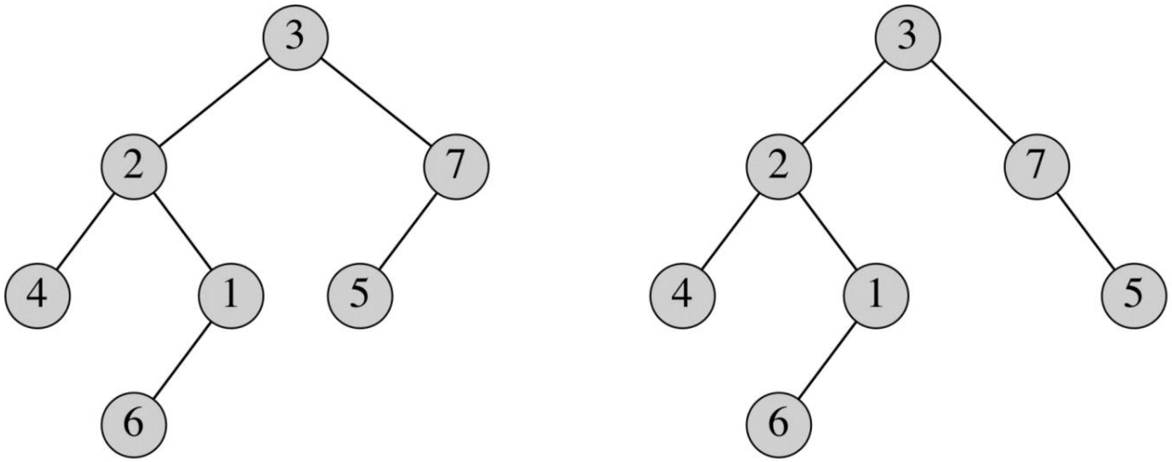
堆排序

主要内容

- 快速找到最小的值——Top K问题，最小生成树的Prime算法，Huffman编码
- 堆——优先队列
- 堆排序
  - O(n lg n) 最坏运行时间—像归并排序。
  - Sorts in place—像插入排序。
  - 结合了两个算法的优点。
  - 一个使用一种数据结构（堆）来排序的排序算法。

二叉树 (1)

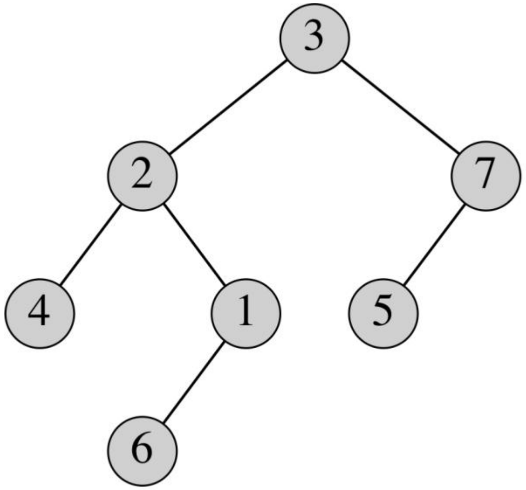
- *二叉树* 是一个有根结点的有序树，其中每个结点最多有两个孩子结点，并且左孩子结点和右孩子结点可区分 (也就是说他们有不同属性)。
- *有序树* 是一个有根结点的树，其中每个结点的孩子结点都是有序的 (第一个孩子结点，第二个孩子结点，等等)。



两个不同的二叉树。

二叉树 (2)

- 在一个二叉树中，
- *一个结点的深度*=从这个结点到根结点的简单路径的边数。
- *一个结点的高度*= 从该结点到一个叶子结点的最长简单路径的边数。
- *一颗树 T的深度* 是树中所有结点最大的深度。
- *一棵树 T 的高度*= 树的根结点的高度= 树的深度

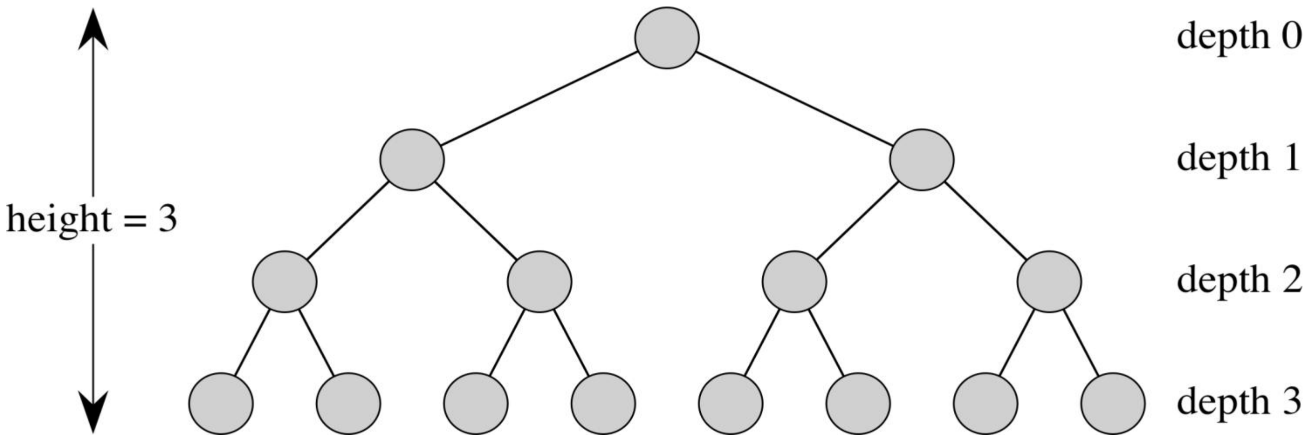


结点2的深度 = 1
树*T* 的深度 = 3
结点2的高度 = 2
树*T* 的高度 = 3

完全二叉树

- *完全二叉树*是一个所有叶子结点在同样深度，而且每个非叶节点都有两个孩子结点的二叉树。

高度为**3**
的完全
二叉树

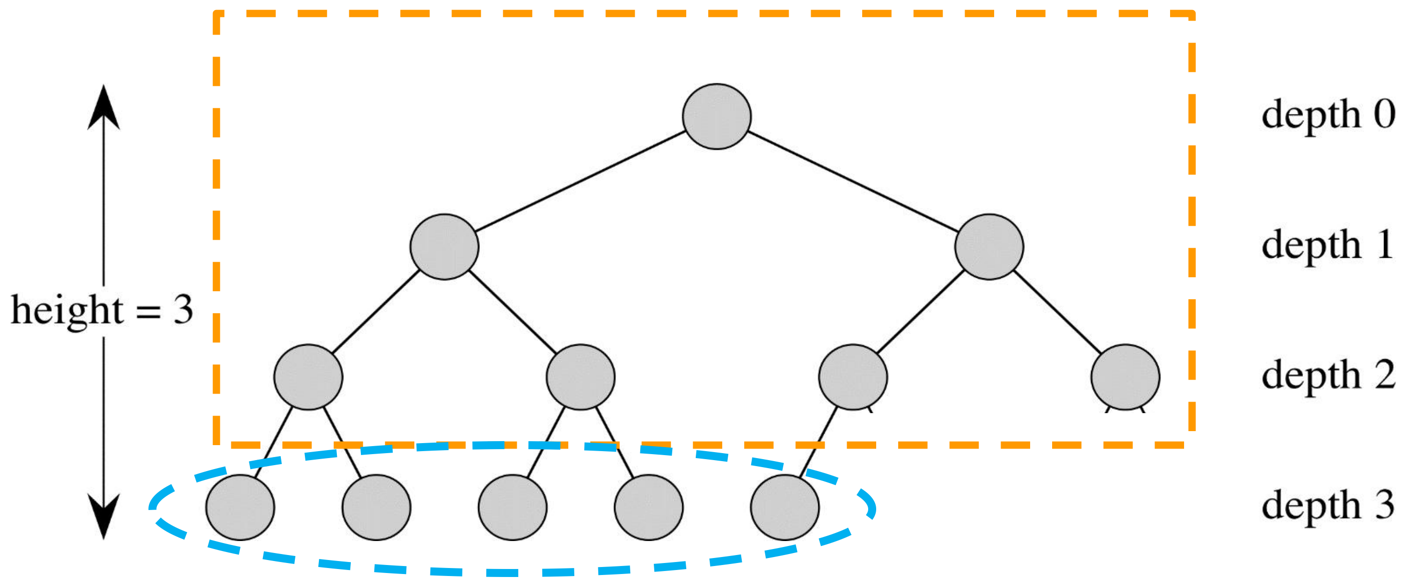


在完全二叉树中，高度为 h 的结点数？ $\mathbf{2^{h+1}-1}$

有 n 个结点的完全二叉树的高度？ $\mathbf{\lg(n+1)-1}$

近似完全二叉树 (1)

- 深度为 d 的 *近似完全二叉树* 满足下面两个条件:
  - 只考虑深度为 $d-1$ 时是完全二叉树
  - 深度为 d 的结点都在靠左部分



高度为 3 的近似完全二叉树

近似完全二叉树 (2)

- 有 n 个结点的近似完全二叉树 T 的高度是多少?
  - 假设 *T* 不是一个完全二叉树。
  - 假设高度是 *h*。
  - *T* 包含一个深度为 $h-1$ 的完全二叉树，而且有一些深度为 *h* 的结点，因此,

$$2^h - 1 < n < 2^{h+1} - 1 \Rightarrow 2^h \le n < 2^{h+1}$$

$$\Rightarrow \lg n - 1 < h \le \lg n \Rightarrow h = \Theta(\lg n)$$

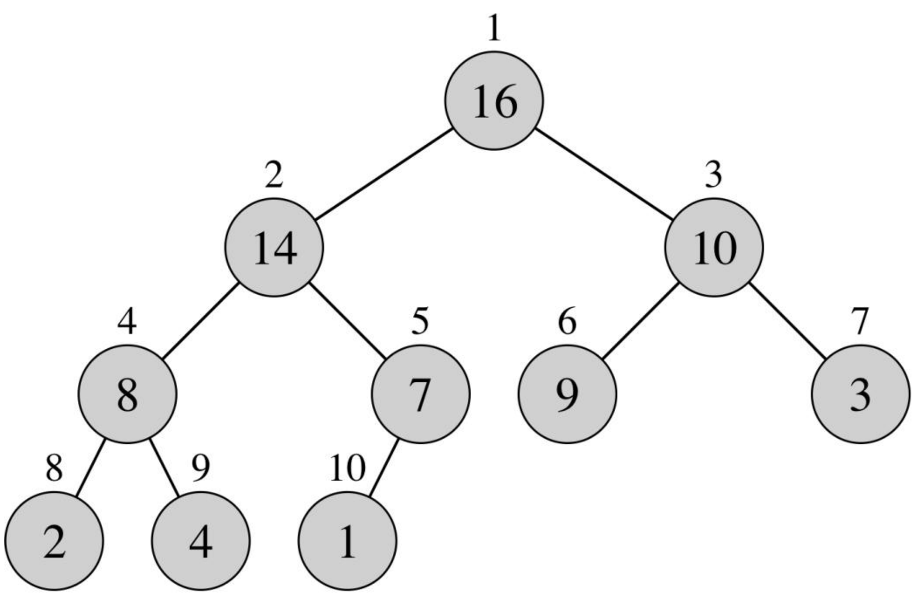
- 高度为 h 的近似完全二叉树有多少结点?
  - 假设有 *n* 个结点， 那么$2^h \le n < 2^{h+1} \Rightarrow n = \Theta(2^h)$

堆

- 一个 (二叉) *堆* 是一个 *近似完全二叉树*:
  - 结点中存储的数值来自一个有序的集合。
  - 每个结点存储的数值满足一种 *堆的性质*.
- 两种堆的性质:
  - *最大堆性质* : 每个结点存储的数值 ≥ 该结点的孩子节点存储的数值。
    - 最大值存储在根结点
  - *最小堆性质* :每个结点存储的数值 ≤该结点的孩子节点存储的数值。
    - 最小值存储在根结点

堆 (续)

- 两种类型的堆:
  - *最大堆*满足*最大堆性质*
  - *最小堆*满足 *最小堆性质*
- *最大堆*举例



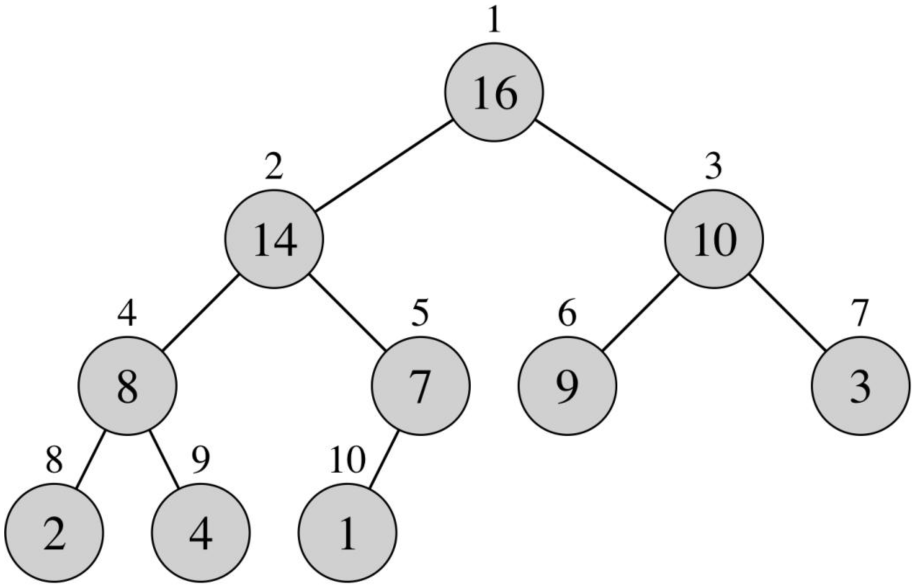
- 如何实现一个堆?
- 如果用数组存储堆，结点外的数字是结点的数组下标。
- 结点里的数字是每个结点存储的值，也叫 ***keys*** 。

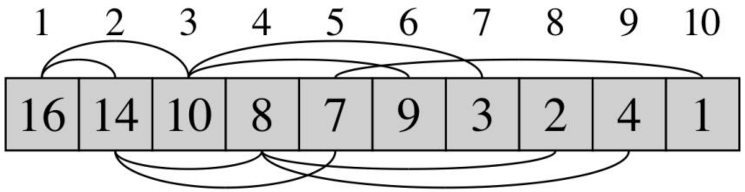
用数组实现堆

- 一个堆可以用一个数组 A来实现。
  - 根结点是 $A[1]$.
  - $A[i]$ 的左孩子结点 = $A[2i]$.
  - $A[i]$ 的右孩子结点 = $A[2i + 1]$.
  - $A[i]$ 的父节点 = $A[\lfloor i/2 \rfloor]$.
- 使用数组, 找父节点和孩子结点的操作可以很快计算。

数组实现 (续)

- 用数组实现最大堆





Arcs go between parents and children.

堆的基本操作

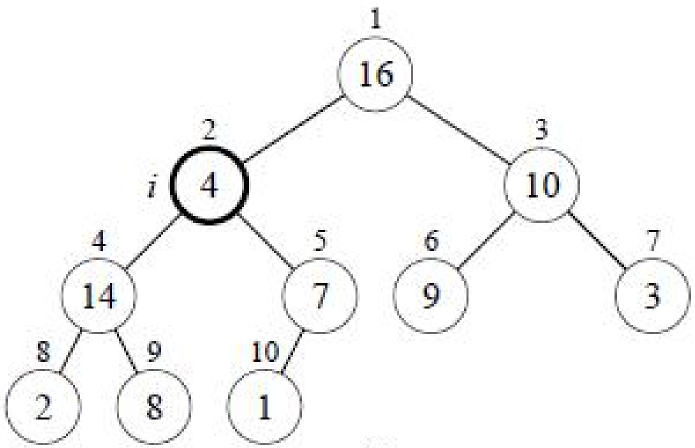
- ***Max-Heapify***: 维护最大堆性质; 代价 $O(\lg n)$ 时间
- ***Build-Max-Heap***: 从一个无序数组建成一个最大堆; 代价 $\Theta(n)$ 时间
- ***Heapsort***: **in place**排序一个数组；代价 $O(n \lg n)$
- ***Max-Heap-Insert***, ***Heap-Extract-Max***, ***Heap-Increase-Key***, **and** ***Heap-Maximum***: 这些操作可用堆实现 *优先队列* 。

维护堆的性质

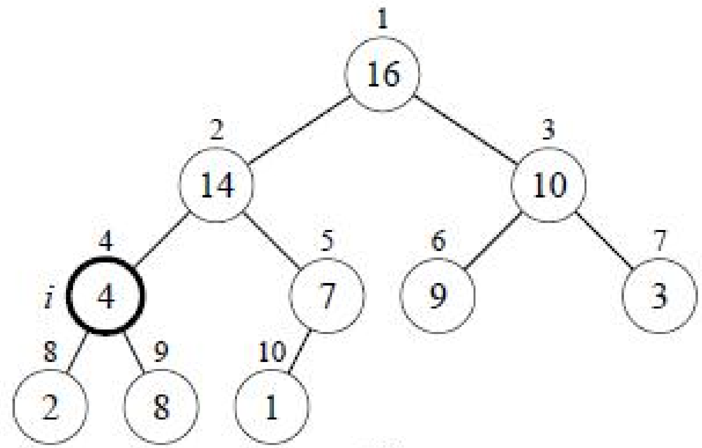
- ***Max-Heapify*** 维护最大堆的性质。
  - 调用 **Max-Heapify** 之前: $A[i]$, 可能比它的孩子结点小。
  - 条件: i 的左和右子树已经是最大堆。
  - 调用 **Max-Heapify** 之后: 以 i 为根的子树是一个最大堆。

- 主要思想:
  - 比较 $A[i]$, $A[\text{Left}(i)]$, and $A[\text{Right}(i)]$
  - 如果有需要, 把 $A[i]$ 与其较大的一个孩子结点交换
  - 在堆中继续向下比较和交换，直到以 i 为根的子树是一个最大堆。

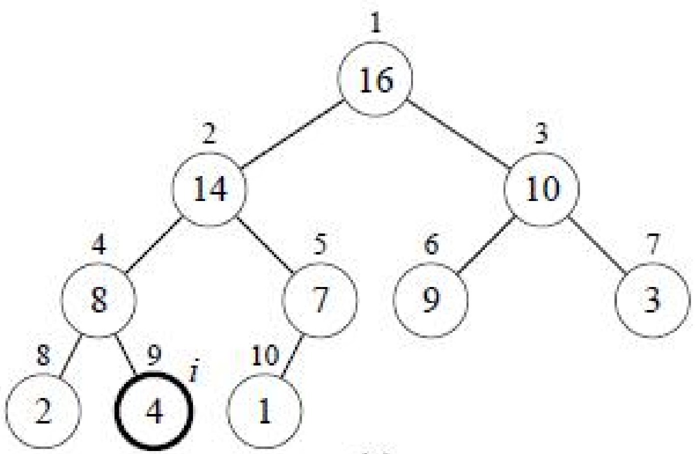
演示 Max-Heapify



(a)



(b)



(c)

- 结点 2 违反最大堆性质。
- 比较结点 2 和其孩子结点, 将结点2与其较大的孩子交换。
- 继续向下比较交换, 直到以存储4的结点为根结点的子树成为一个最大堆。此时，最大堆就是一个叶子结点。

算法 Max-Heapify

```
MAX-HEAPIFY(A, i, n)
  l = LEFT(i)
  r = RIGHT(i)
  if l ≤ n and A[l] > A[i]
        largest = l
  else largest = i
  if r ≤ n and A[r] > A[largest]
        largest = r
  if largest ≠ i
        exchange A[i] with A[largest]
        MAX-HEAPIFY(A, largest, n)
```

- 运行时间: **$O(\lg n)$**
  - 树的高度是 **$\lg n$**
  - 将 **$A[i]$** 向下移动一层需要常数时间

建堆

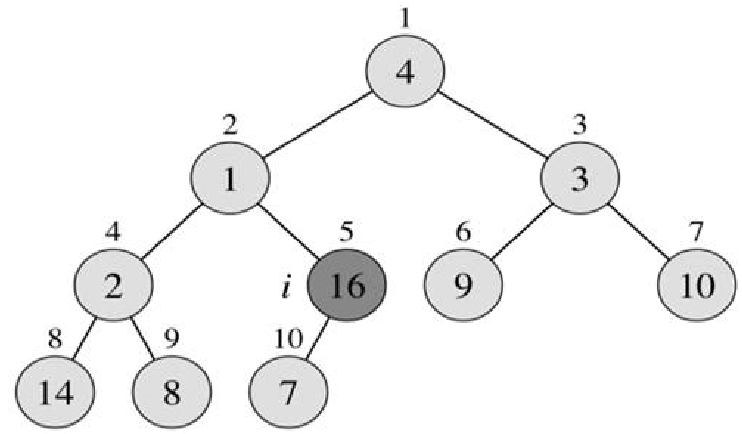
- 自底向上的过程把一个无序的数组 *A* 建成一个最大堆

为什么不从 n 开始?

BUILD-MAX-HEAP(A, n)

for $i = \lfloor n/2 \rfloor$ **downto** 1

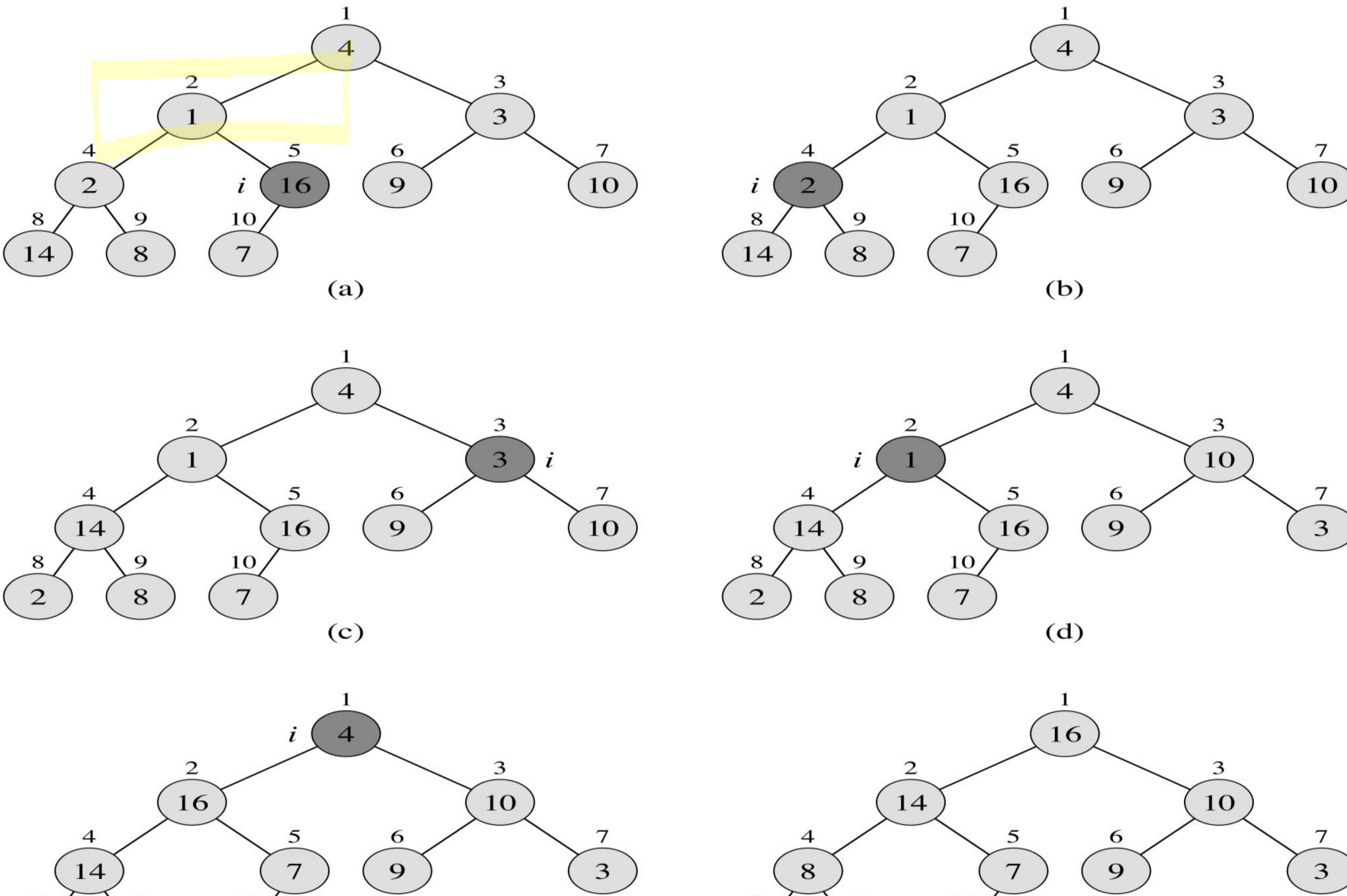
  MAX-HEAPIFY(A, i, n)



- 在**heapification**的过程中, 只需要考虑非叶节点。
- 子数组 $A[\lfloor n/2 \rfloor+1 .. n]$ 中的元素对应的所有结点都是叶子结点，因为 $A[\lfloor n/2 \rfloor]$ 是非叶节点中数组下标最大的。
  - $A[\lfloor n/2 \rfloor]$ 的左孩子是 $A[2\lfloor n/2 \rfloor]$, **which is** $A[n]$ 如果 n 是偶数 **or** $n-1$ 如果 n 是奇数。

建堆: 举例

A	4	1	3	2	16	9	10	14	8	7

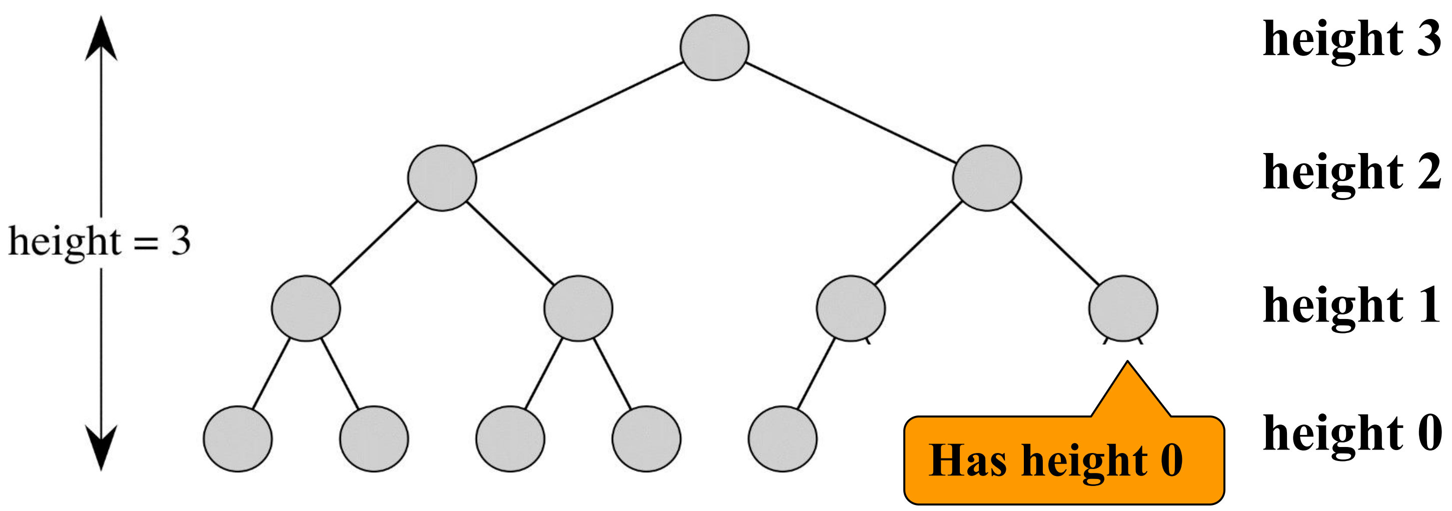


建堆: 正确性

- *循环不变*: 每一次for循环的开始，结点 $i+1, i+2, \ldots, n$ 都是一个最大堆的根结点。
- *初始化*: 结点 $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ 都是叶子结点, 他们都是一个最大堆的根结点。 循环开始时 $i = \lfloor n/2 \rfloor$, 上述循环不变为真
- *保持*: 结点 i 的孩子结点的数组下标比 i 大，因此，根据循环不变，它们都是最大堆的根。因此，调用 **Max-Heapify**(A, i, n) 的条件被满足，该过程使得结点 i 成为一个最大堆的根。递减 i 的值为下一次循环重新建立循环不变。
- *中止*: 当 $i = 0$, 循环中止。根据循环不变, 每个结点都是最大堆的根。结点1就是最大的那个堆的根。

建堆: 分析(1)

- *简单界*: $O(n)$ 调用 **Max-Heapify**, 每次调用需要 $O(\lg n)$ 时间 ➔ 建堆需要 $O(n \lg n)$ 时间。
- 能找到更准确的界吗?
- *准确界*: 一个结点上 **Max-Heapify** 的运行时间是该结点高度的线性函数, 大多数结点的高度很小。堆的高度是 $\lg n$。
  - 最多有 $\lceil n/2^{h+1} \rceil$ 个高度为 h 的结点。



建堆: 分析 (2)

- *准确界* (续):
- 最多有 $\lceil n/2^{h+1} \rceil$ 个高度为 h 的结点。
- 在高度为 h 的结点上运行 **Max-Heapify** 的时间是 $O(h)$, 因此建堆总的代价是

$$\sum_{h=0}^{\lg n} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lg n} \frac{h}{2^h} \right) \le O\left(n \sum_{h=0}^{\infty} h \left(\frac{1}{2} \right)^h \right)$$

- 因为 $\sum_{i=0}^{\infty} x^k = \frac{1}{1-x}$ **for** $|x| < 1$,

$$\sum_{i=0}^{\infty} k x^{k-1} = \frac{1}{(1-x)^2} \Rightarrow \sum_{i=0}^{\infty} k x^k = \frac{x}{(1-x)^2} \Rightarrow O\left(n \sum_{h=0}^{\infty} h \left(\frac{1}{2} \right)^h \right) = O(2n)$$

- 建堆的代价为 $O(n)$。

堆排序算法: 思想

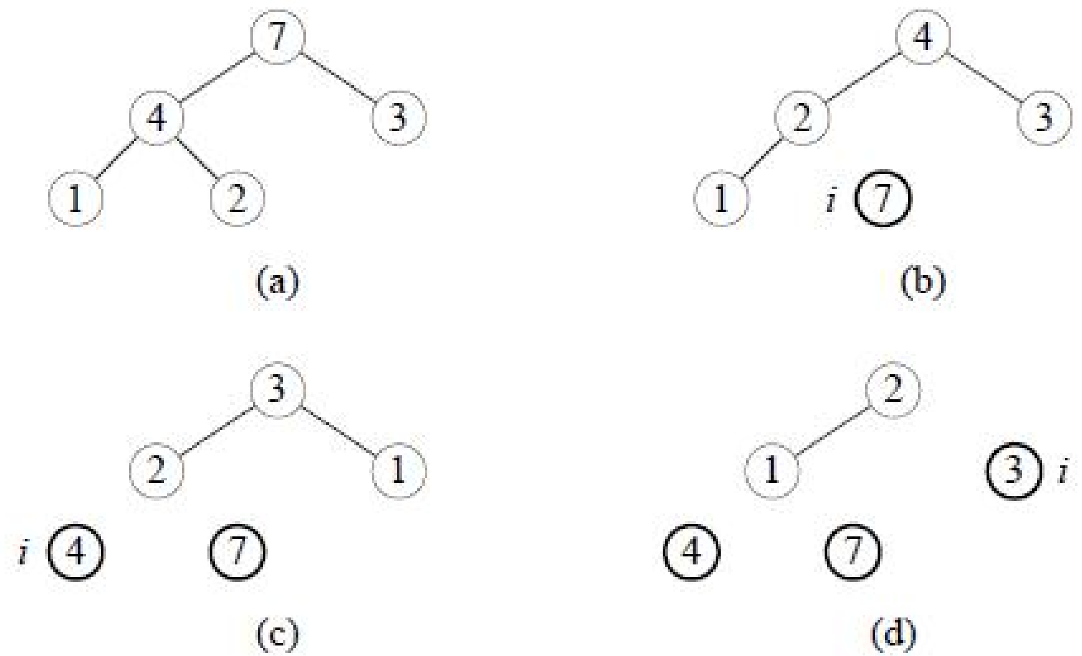
给定一个数组, *堆排序* 算法如下:

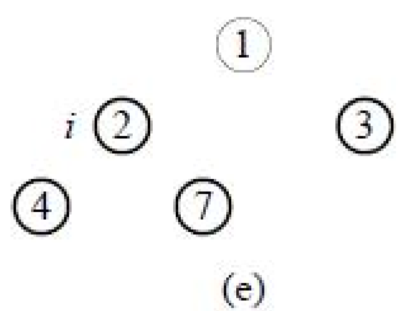
- 在数组上建一个最大堆。
- 从根结点开始 (它的值最大), 算法将最大值放到数组中正确的地方，也就是将它与数组中最后一个元素交换位置。
- “去掉” 数组中最后一个元素 (它已经在正确的位置)， 在新的根结点上调用 **Max-Heapify**，新的根结点有可能违反堆的性质。
- 重复“去掉” 操作直到只剩一个结点 (也就是最小值), 这是数组已经排序完成。

堆排序: 举例

初始数组:

A					
	7	**4**	**3**	**1**	**2**





排序后的数组:

A					
	1	2	3	4	7

堆排序: 伪代码

$\text{HEAPSORT}(A, n)$

$\quad\text{BUILD-MAX-HEAP}(A, n)$
$\quad$**for** $i = n$ **downto** 2
$\qquad$exchange $A[1]$ with $A[i]$
$\qquad\text{MAX-HEAPIFY}(A, 1, i - 1)$

Heapsort Algorithm: Analysis

```
HEAPSORT(A, n)
  BUILD-MAX-HEAP(A, n)
  for i = n downto 2
      exchange A[1] with A[i]
      MAX-HEAPIFY(A, 1, i − 1)
```

- **Build-Max-Heap: $O(n)$**
- **for loop: $n - 1$ 次**
  - 交换值**: $O(1)$**
  - **Max-Heapify: $O(\lg n)$**
- *总时间* **: $O(n \lg n)$**
  - 与归并排序一样，而且是**in place**排序。

优先队列

- 堆的应用，实现一个高效的 *优先队列。*
- 优先队列是一个维护动态集合 **S** 数据结构, 其中每一个元素都有一个值 (也称 ***key***), 这个值表示该元素的 *优先级* 。
- 类比最大堆和最小堆, 也有 *最大优先队列* **and** *最小优先队列。*
- 最大优先队列的应用: 共享计算机系统的作业调度 – 在将要执行的所有作业中, 选择优先级最高的执行。

优先队列操作

- 最大优先队列支持如下操作:
  - **Insert(*S*, *x*): 将元素 *x* 插入集合 *S*。**
  - **Maximum(*S*): 返回集合 *S* 中 key 最大的元素。**
  - **Extract-Max(*S*):去掉并返回集合 *S* 中 key 最大的元素。**
  - **Increase-Key(*S*, *x*, *k*): 增加元素 *x* 的key 到 *k*。假定 $k \geq x$ 当前的key。**
- 最小优先队列支持的操作:
  - **Insert(*S*, *x*):将元素 *x* 插入集合 *S*.**
  - **Minimum(*S*):返回集合 *S* 中 key 最小的元素.**
  - **Extract-Min(*S*):去掉并返回集合 *S* 中 key 最小的元素.**
  - **Decrease-Key(*S*, *x*, *k*):减少元素 *x* 的key 到 *k*。假定 $k \leq x$ 当前的key。**

用堆实现优先队列的操作

- 用最大堆和它的操作实现最大优先队列。
  - **Max-Heap-Insert(*A*, *x*)**
  - **Heap-Maximum(*A*): return *A*[1].**
  - **Heap-Extract-Max(*A*, *n*)**
  - **Heap-Increase-Key(*A*, *x*, *k*)**

- 用最小堆和它的操作实现最小优先队列。

Heap-Extract-Max

给定数组 *A*:

- 确保堆不为空。
- 复制最大元素(根结点).
- 把树中最后一个结点变成新的根结点。
- **Re-heapify** 减少一个结点的堆。
- 返回复制的最大元素。

```
HEAP-EXTRACT-MAX(A, n)
 if n < 1
     error "heap underflow"
 max = A[1]
 A[1] = A[n]
 n = n - 1
 MAX-HEAPIFY(A, 1, n)        // remakes heap
 return max
```

Running time: Constant-time assignments plus time for Max-Heapify: $\Theta(\lg n)$.

Heap-Increase-Key

给定数组 A, 元素 $A[i]$, 和新的 *key*:

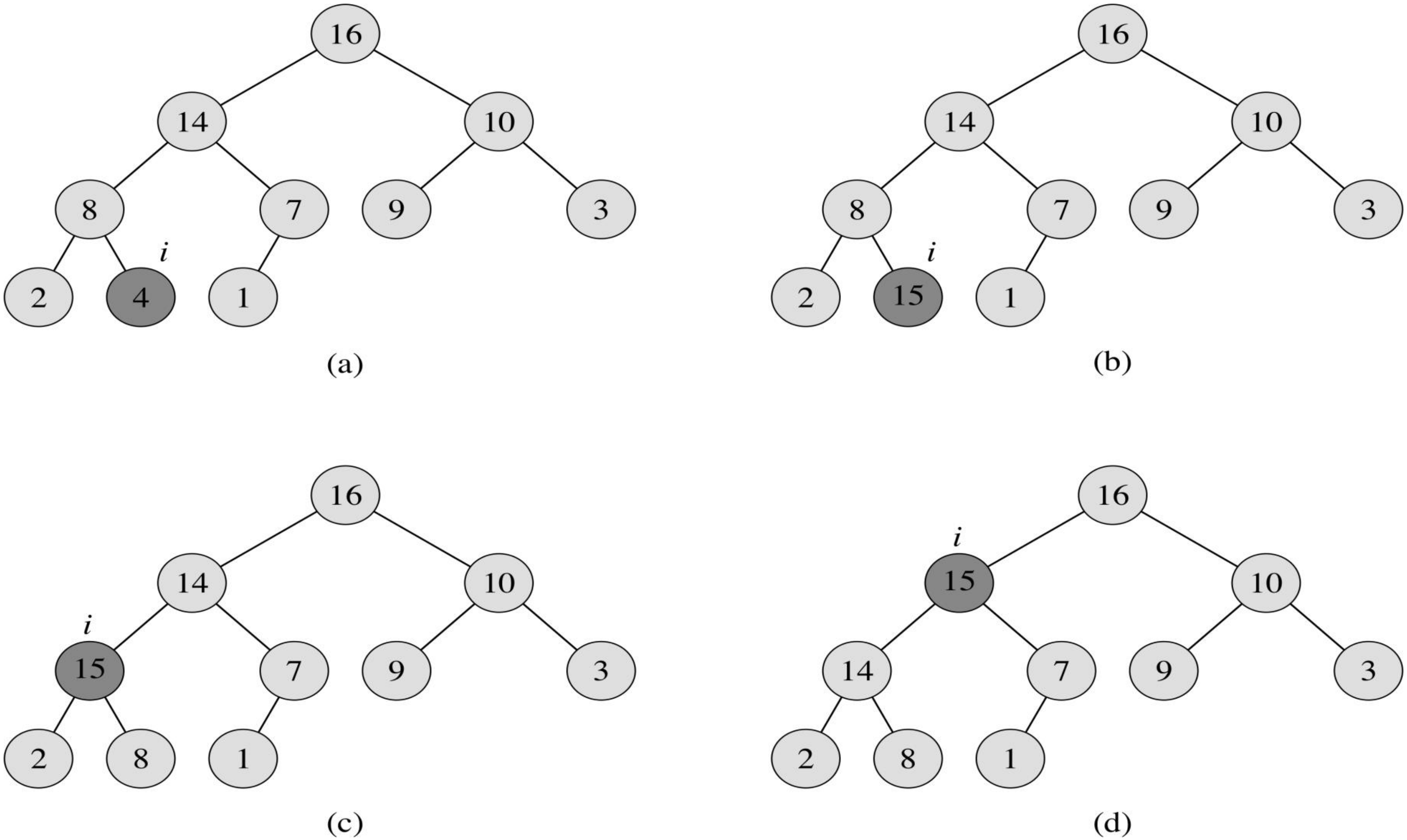
- 确保 $key \geq A[i]$.
- 更新 $A[i]$的 key。
- 向上遍历树，比较 $A[i]$ 和它的父结点，有需要就交换值, 直到 $A[i]$的 key 比它的父结点的 key小。

Running time: Upward path from node *i* has length $O(\lg n)$ in an *n*-element heap: $O(\lg n)$.

```
HEAP-INCREASE-KEY(A, i, key)
  if key < A[i]
      error "new key is smaller than current key"
  A[i] = key
  while i > 1 and A[PARENT(i)] < A[i]
      exchange A[i] with A[PARENT(i)]
      i = PARENT(i)
```

Heap-Increase-Key: 举例

Heap-Increase-Key(*A*, *i*, 15)



Max-Heap-Insert

将 *key* 插入到堆中:

- 增加堆的大小。
- 在堆的最后一个位置增加一个 key 为 – ∞的结点。
- 增加 – ∞ 到 *key*， 调用 Heap-Increase-Key 。

$$\begin{aligned}&\text{MAX-HEAP-INSERT}(A, key, n)\\&\quad n = n + 1\\&\quad A[n] = -\infty\\&\quad \text{HEAP-INCREASE-KEY}(A, n, key)\end{aligned}$$

Running time: Constant time assignments + time for Heap-Increase-Key: $O(\lg n)$.

用堆实现优先队列: 总结

- 优先队列操作的运行时间 $O(\lg n)$.
  - **Max-Heap-Insert(A, x): $O(\lg n)$**
  - **Heap-Maximum(A): return $A[1]$: $O(1)$**
  - **Heap-Extract-Max(A, n): $O(\lg n)$**
  - **Heap-Increase-Key(A, x, k): $O(\lg n)$**
- 除了 **Heap-Maximum(A)**, 其他操作的运行时间以堆的高度为界。
  - 有些操作向上执行。
  - 有些操作向下执行。

优先队列的其他操作

- 假定一个集合 S 中，每个元素 e 有两个属性：
  - **id** : 唯一定义 e
  - **priority**: e 的优先级
- 操作:
  - **Find**(S, x): 在S中找到 $id = x$ 元素的优先级。
  - **ChangePriority**(S, x, p): 将 id = x 元素的优先级变为 p，可变大，也可变小。
- *问题* : 用堆实现**Find**(S, x)的运行时间?
- *答案* : $O(n)$. 堆中的元素不按*id*排序。

改进Find(S, x)

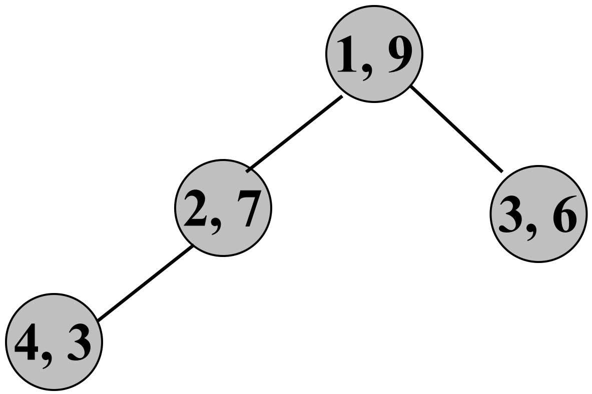
- 如何使Find(S, x) 的运行时间成为 $O(1)$ ？
- 用另一个数组“handle”追踪堆中每个元素的位置，如果 id x 元素不在堆中， “handle” 中的值为 “impossible value”。
- 假定:
  - 优先队列最多有 n 个元素
  - id 是1 至n之间的整数
  - 没有多次出现的具有相同id 的元素

改进Find(S, x) (续)

- 引入一个新的数组$L[1 .. n]$ 追踪元素的位置: $L[i]$ 存储id = i的元素的位置。
- 两个数组 $A[1 .. n]$ 和 $L[1 .. n]$
  - $A[1 .. n]$ 存储元素的ids 和优先级, $A[1 .. n]$ 是堆。
  - $L[1 .. n]$ 存储$A[1 .. n]$中元素的位置。
- $L[1 .. n]$ 可以在$O(1)$时间找到任何给定 id = x 的元素: 该元素是 $A[L[x]]$。
- 如果A中有元素移动，他们的位置也需要在 L 中更新，这是用 $O(1)$ 时间实现Find(S, x)的代价。
- 元素有多个属性，用 $O(1)$ 时间实现Find(S, x)： 扩展堆。

改进Find(S, x) (续)

举例: 最大堆， $n = 5$。

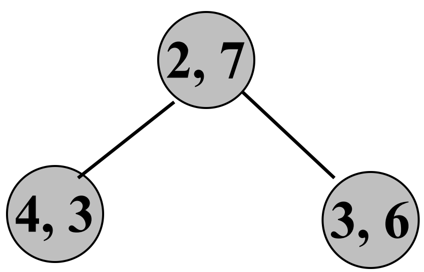


	1	2	3	4	5
A	1,9	2,7	3,6	4,3	

	1	2	3	4	5
L	1	2	3	4	-1

Note: -1 indicates missing element.

- After Heap-Extract-Max(A, 4):



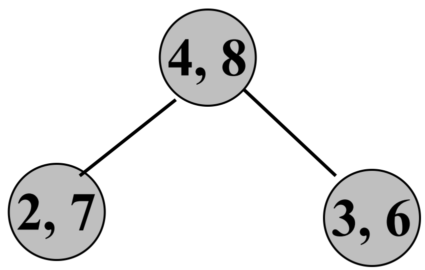
	1	2	3	4	5
A	2,7	4,3	3,6		

	1	2	3	4	5
L	-1	1	3	2	-1

Note how L is updated.

改进Find(S, x) (续)

- **After ChangePriority(A, 4, 8):**



	1	2	3	4	5
A	4,8	2,7	3,6		

	1	2	3	4	5
L	-1	2	3	1	-1