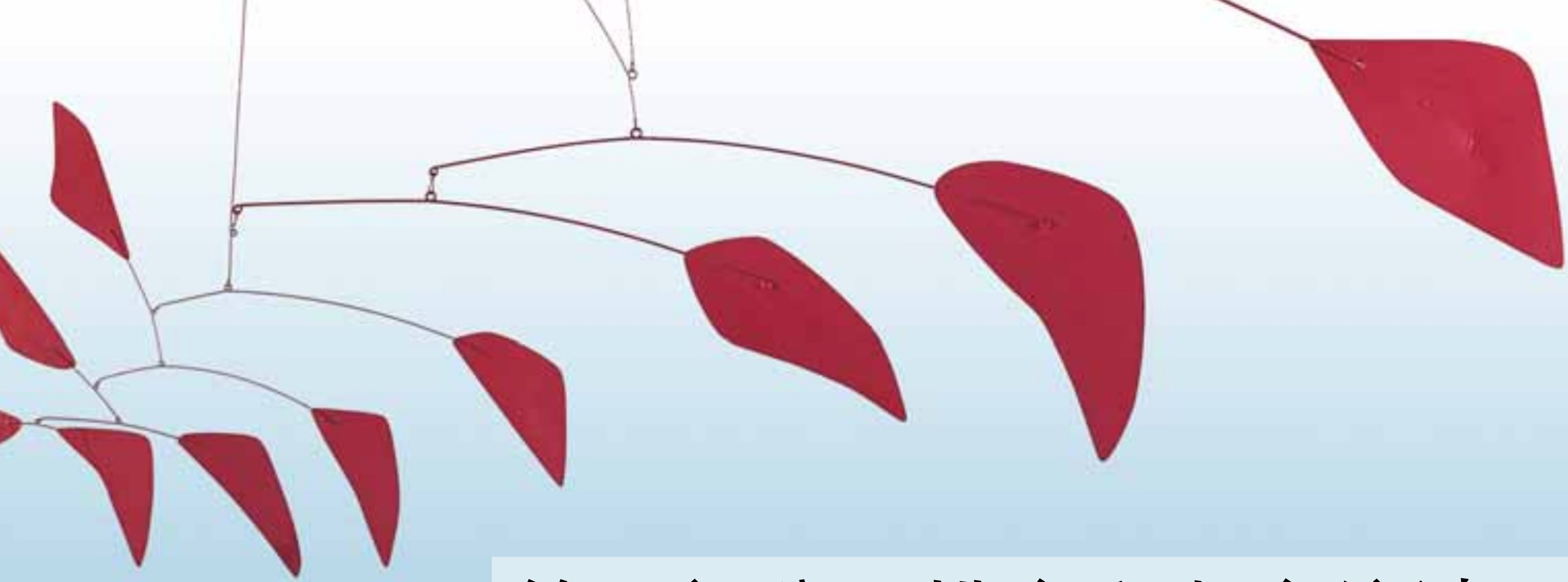


第二部分：排序和顺序统计量

INTRODUCTION TO

ALGORITHMS

THIRD EDITION

堆排序

《算法导论》—— 第5讲

jiacaicui@163.com

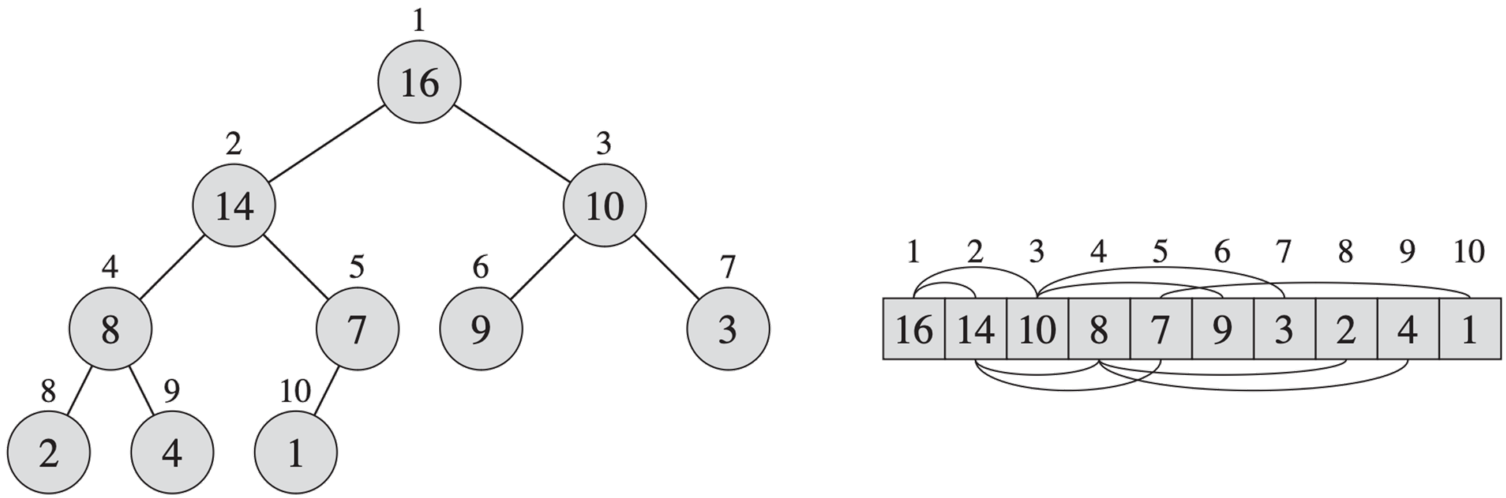
内容提要

- 学习“堆”数据结构
  - 最大堆与最小堆
  - 维护堆的性质：MAX-HEAPIFY
  - 建堆：BUILD-MAX-HEAP
- 堆排序算法
  - 时间：最坏情况 $O(n \log n)$ ——归并排序。
  - 空间：原址（in place）排序，只需常数项额外空间——插入排序。
  - 兼备“归并排序”和“插入排序”之长。
- 优先队列及其实现
  - 同一数据结构（抽象数据类型）的不同实现具有不同的复杂度。

堆

简单而又强大的数据结构

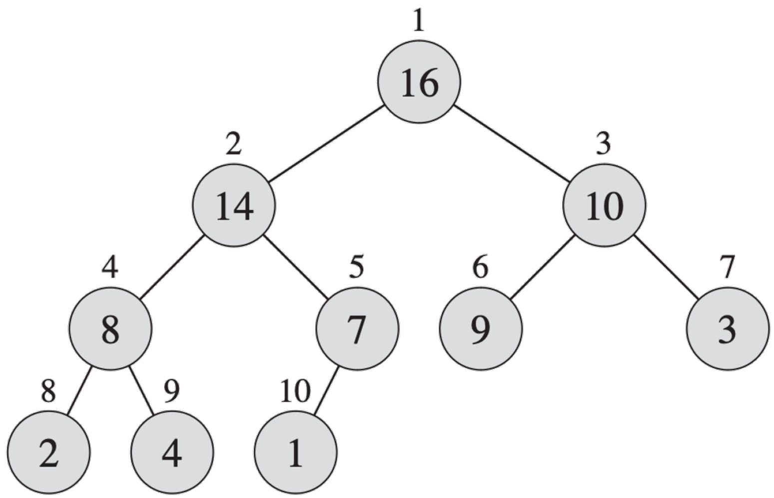
堆



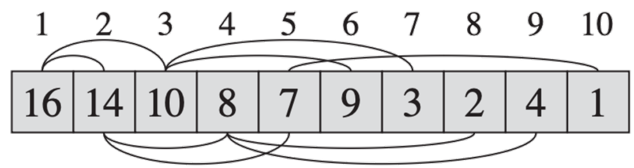
(a) (b)

- 堆是一个完全二叉树（complete binary tree）。
  - 完全二叉树是一个二叉树，它除了最后一层以外的每一层都是满的，最后一层的叶子结点集中在最左边。
  - 满二叉树是一个二叉树，它的每一层都是满的。
  - 第 k 层“满”等价于这一层有 2^{k-1} 个结点。

堆



(a)



(b)

- 完全二叉树可以储存为一个数组。
  - 根结点为 $A[1]$，下标为 i 的结点的父结点、左子结点、右子结点的下标可以如下计算：

PARENT(i)
1 **return** $\lfloor i/2 \rfloor$

LEFT(i)
1 **return** $2i$

RIGHT(i)
1 **return** $2i + 1$

完全二叉树的“下标”计算

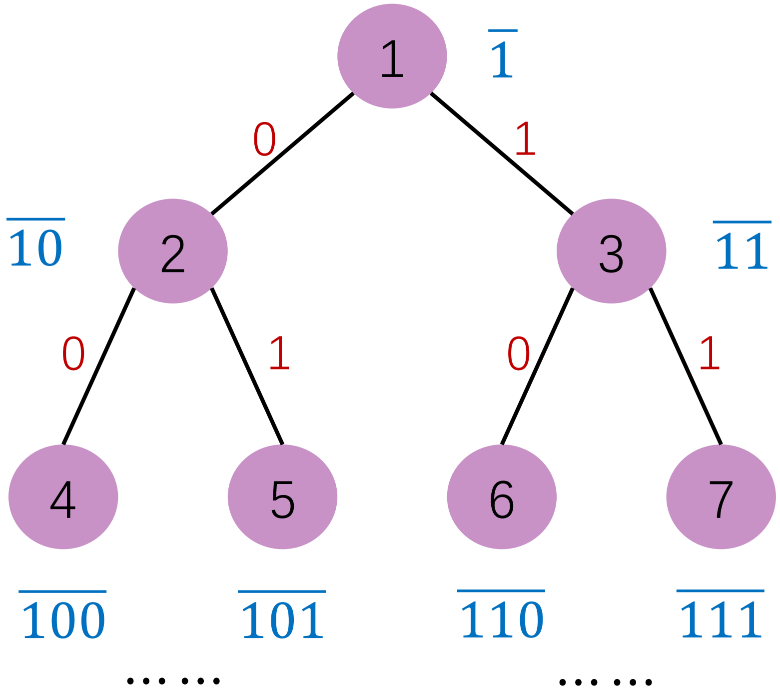
$\text{PARENT}(i)$
1 **return** $\lfloor i/2 \rfloor$

$\text{LEFT}(i)$
1 **return** $2i$

$\text{RIGHT}(i)$
1 **return** $2i + 1$

- 显然是对的，但为什么？
  - 所有的正整数可以用形如 $\overline{1a_1a_2\cdots a_k}$ $(k \geq 0)$ 的二进制数编码表示，构造如下的完全二叉树：
    - 对于任意的 $i = \overline{1a_1a_2\cdots a_k}$，从根结点 1 到结点 i 的路径长度为 k，其中：
    - 如果 $a_j = 0$，则第 j 步“走”左子结点；
    - 如果 $a_j = 1$，则第 j 步“走”右子结点。
  - 我们会得到一棵怎样的二叉树呢？

完全二叉树的“下标”计算



LEFT(i)

return $2i$

RIGHT(i)

return $2i+1$

PARENT(i)

return $\lfloor i/2 \rfloor$

- 基于数组的完全二叉树的“好”的计算机实现：
  - 使用左移代替乘2运算，右移代替除以2运算；
  - 使用宏或者内联函数代替函数。

一些记号和说法上的约定

- 表示堆的数组 A 包括两个属性：
  - $A.length$ 通常给出的是数组元素的个数，$A.heap-size$ 表示该数组中有效堆元素的个数，$0 \le A.heap-size \le A.length$ 。
  - $A[1..A.length]$ 可能都有存放数据，但只有 $A[1..A.heap-size]$ 是堆的有效元素。
- 高度：
  - 结点的高度：该结点到叶结点最长简单路径上边的数目；
  - 堆的高度：根结点 $A[1]$ 的高度，n 个元素的堆的高度为 $\lfloor \log n \rfloor$ 。(详见练习5-1问题1)

堆的性质

- 最大堆：除了根结点以外的所有结点 i 满足

$$A[PARENT(i)] \geq A[i]$$

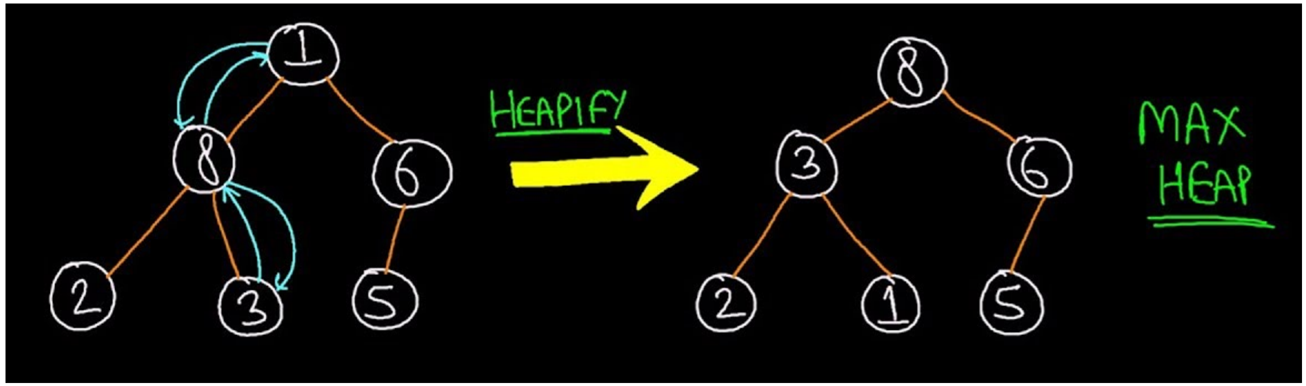
  - 每个结点至多与其父结点一样大，根结点是最大结点。
  - 堆排序使用的是最大堆。

- 最小堆：除了根结点以外的所有结点 i 满足

$$A[PARENT(i)] \leq A[i]$$

  - 每个节点至少与其父结点一样大，根结点是最小结点。
  - 最小堆常用于构造优先队列。

- 对于某个特定的应用，必须明确是最大堆还是最小堆；除非某个属性既适用于最大堆，也适用于最小堆。



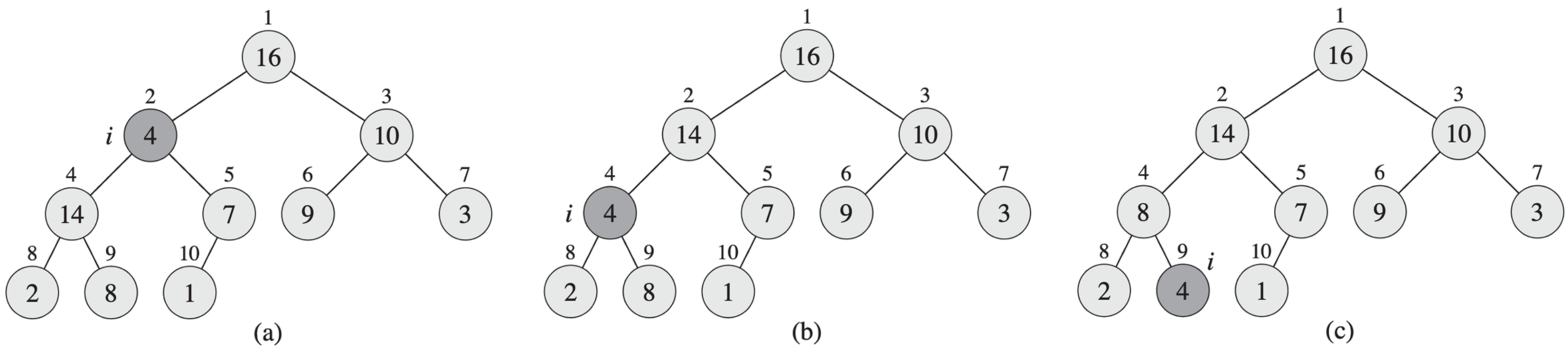
维护堆的性质

使得给定子树满足最大堆的性质

维护最大堆的性质

输入：结点 i， 满足根结点为 $LEFT(i)$ 和 $RIGHT(i)$ 子树都满足最大堆性质。

输出：根结点为 i 的子树也满足最大堆性质。

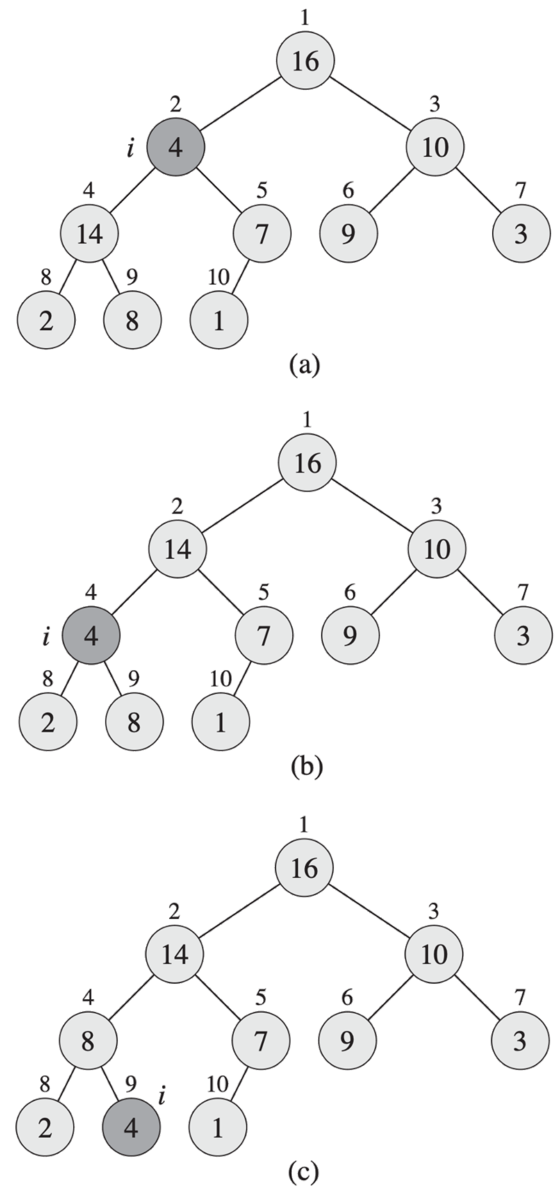


核心思路：让 $A[i]$ 在最大堆中 “逐级下降” 。

- 如果 $A[i]$ 比 $A[LEFT(i)]$ 和 $A[RIGHT(i)]$ 都大，则已经满足最大堆性质，算法终止。
- 否则，将 $A[i]$ 与 $A[LEFT(i)]$ 和 $A[RIGHT(i)]$ 中较大的那个交换，然后递归地对被改动的子树维护最大堆的性质。

MAX-HEAPIFY

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



结点数为 n，高度为 $h=\Theta(\log n)$ 的子树的运行时间：

$$T(h) \le T(h-1)+\Theta(1)$$

$$\Rightarrow T(h)=O(h)=O(\log n)$$

迭代版本的 MAX-HEAPIFY

- 尾递归（tail-recursion）可以很容易得转化成只需要常数项额外空间的迭代。
- 本质：递归调用后本次调用栈帧中的数据无需再次使用。

```
procedure MaxHeapify(A, i)
    largest = i
    while true do
        l = Left(i)
        r = Right(i)
        if l ≤ A.heap-size and A[l] > A[largest] then
            largest = l
        end if
        if r ≤ A.heap-size and A[r] > A[largest] then
            largest = r
        end if
        if largest = i then
            break
        else
            Swap(A[i], A[largest])
            i = largest
        end if
    end while
end procedure
```

注：尾递归指的是递归函数的递归调用发生在“尾部”。

建堆

根据给定元素构建最大堆

建立最大堆

输入：长度为 n 的数组 $A[1..n]$ 。

输出：大小为 n 的堆 $A[1..n]$ 。

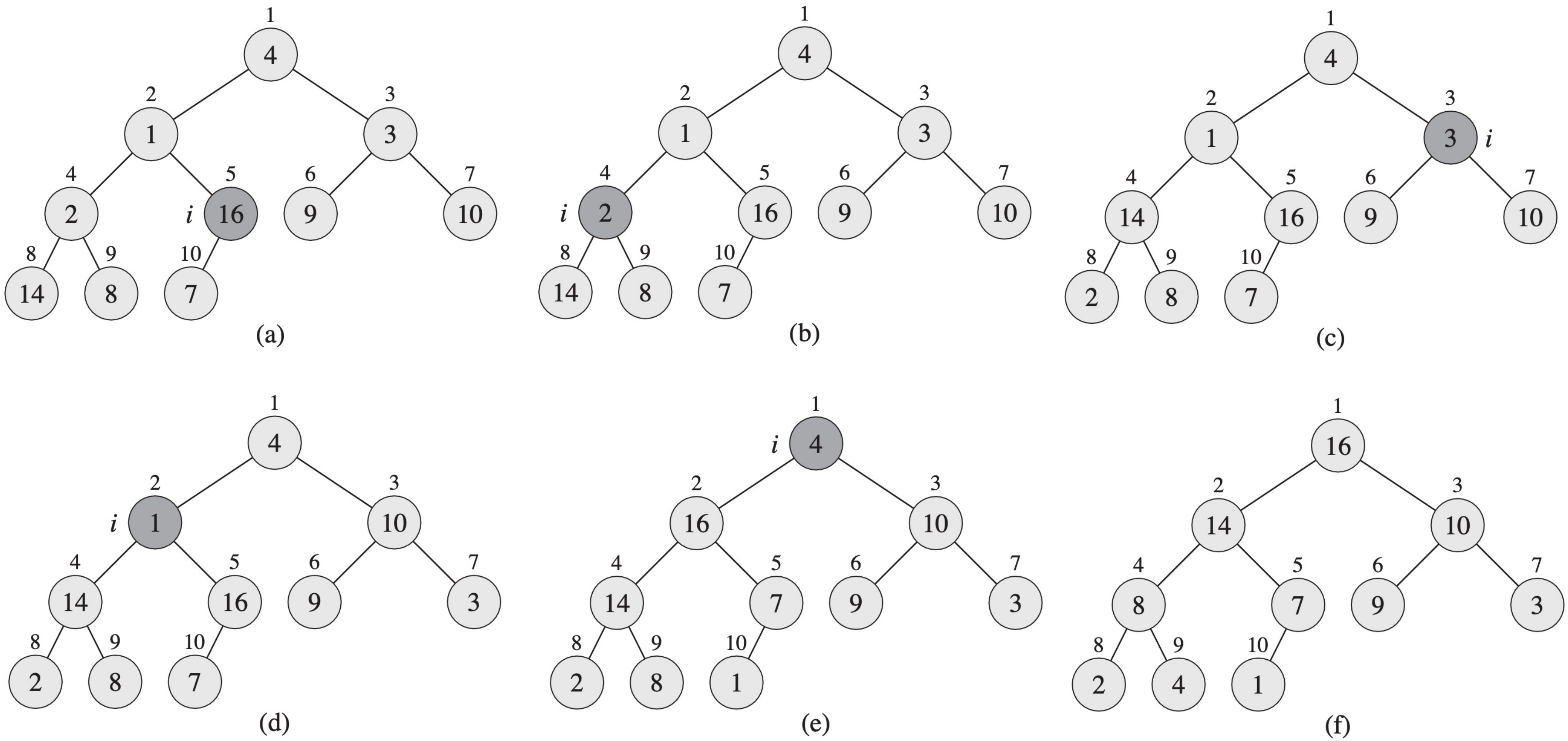
```
BUILD-MAX-HEAP(A)
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
    MAX-HEAPIFY(A, i)
```

核心思路：对于每一个非叶结点调用 MAX-HEAPIFY 来维护对应子树的最大堆性质。

注：子数组 $A\left[\left\lfloor\frac{n}{2}\right\rfloor+1..n\right]$中的元素都是堆的叶结点（详见练习5-1问题2）。

BUILD-MAX-HEAP 算法过程

A										
	4	1	3	2	16	9	10	14	8	7



(a) (b) (c) (d) (e) (f)

- 核心思路：对于每一个非叶结点调用 MAX-HEAPIFY 来维护对应子树的最大堆性质。

BUILD-MAX-HEAP 的正确性

```
2    for i = ⌊A.length/2⌋ downto 1
3        MAX-HEAPIFY(A, i)
```

循环不变式：第 2 到 3 行中每一次 for 循环的开始，结点 $i+1, i+2, \cdots, n$ 都是一个最大堆的根结点。

- 初始化：$i = \left\lfloor \frac{n}{2} \right\rfloor$， $\left\lfloor \frac{n}{2} \right\rfloor + 1, \left\lfloor \frac{n}{2} \right\rfloor + 2, \cdots, n$ 都是叶结点，是平凡的最大堆根结点。
- 保持：若每次迭代开始前 $i+1, i+2, \cdots, n$ 都是一个最大堆的根结点，
  - $RIGHT(i) > LEFT(i) > i$， $RIGHT(i)$ 和 $LEFT(i)$ 都是最大堆根结点；
  - 从而 $MAX-HEAPIFY(A, i)$ 维护了最大堆的性质，i 也是最大堆的根结点；
  - 则 $i, i+1, i+2, \cdots, n$ 都是最大堆的根结点，循环保持了不变式。
- 终止：$i = 0$，根据循环不变式，结点 $0 + 1 = 1$ 是一个最大堆的根结点，则 A 是一个最大堆。

BUILD-MAX-HEAP 的运行时间

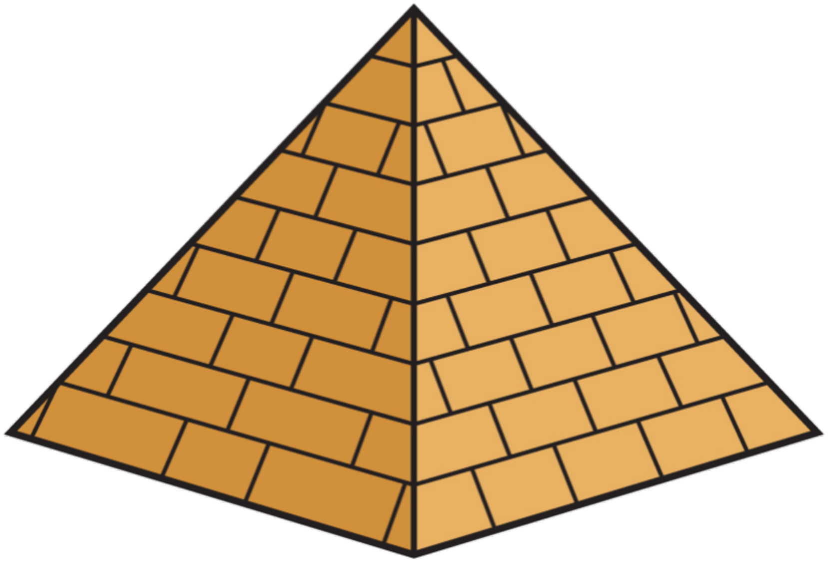
```
BUILD-MAX-HEAP(A)
    A.heap-size = A.length
    for i = ⌊A.length/2⌋ downto 1
        MAX-HEAPIFY(A, i)
```

- n 个元素的堆最多有 $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ 个高度为 h 的结点（详见练习5-1问题6）；
- 对于高度为 h 的结点 i 调用 MAX-HEAPIFY 的时间为 $O(h)$；
- 总代价为：

$$\sum_{h=1}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot O(h) = O\left(n \sum_{h=1}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right) = O\left(n \sum_{h=1}^{\infty} \frac{h}{2^h}\right) = O\left(n \cdot \frac{\frac{1}{2}}{\left(1 - \frac{1}{2}\right)^2}\right) = O(n)$$

- 因此，BUILD-MAX-HEAP可以在线性时间内把一个无序数组构造成为一个最大堆。

堆排序算法



兼采插入和归并之长的排序算法

堆排序及其正确性

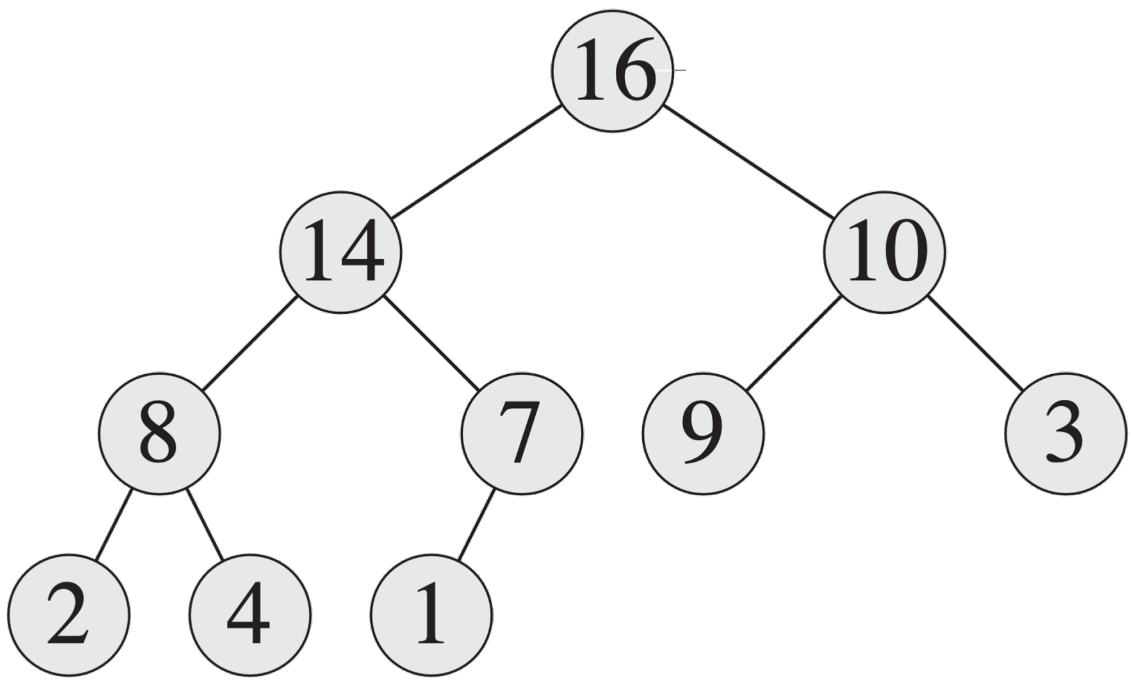
核心思路：先建堆，然后不断地将堆顶元素和堆尾元素互换，减小堆的大小，并调用 MAX-HEAPIFY(A, 1) 维护最大堆的性质。

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

循环不变式：算法第 2 到 5 行的 **for** 循环每次迭代开始时，子数组 $A[1..i]$ 是一个包含了 $A[1..n]$ 中前 i 小的元素的最大堆；子数组 $A[i+1..n]$包含了 $A[1..n]$ 中已经排好序的前 $n-i$ 大元素。（证明见*练习5-1问题7*）

堆排序的过程

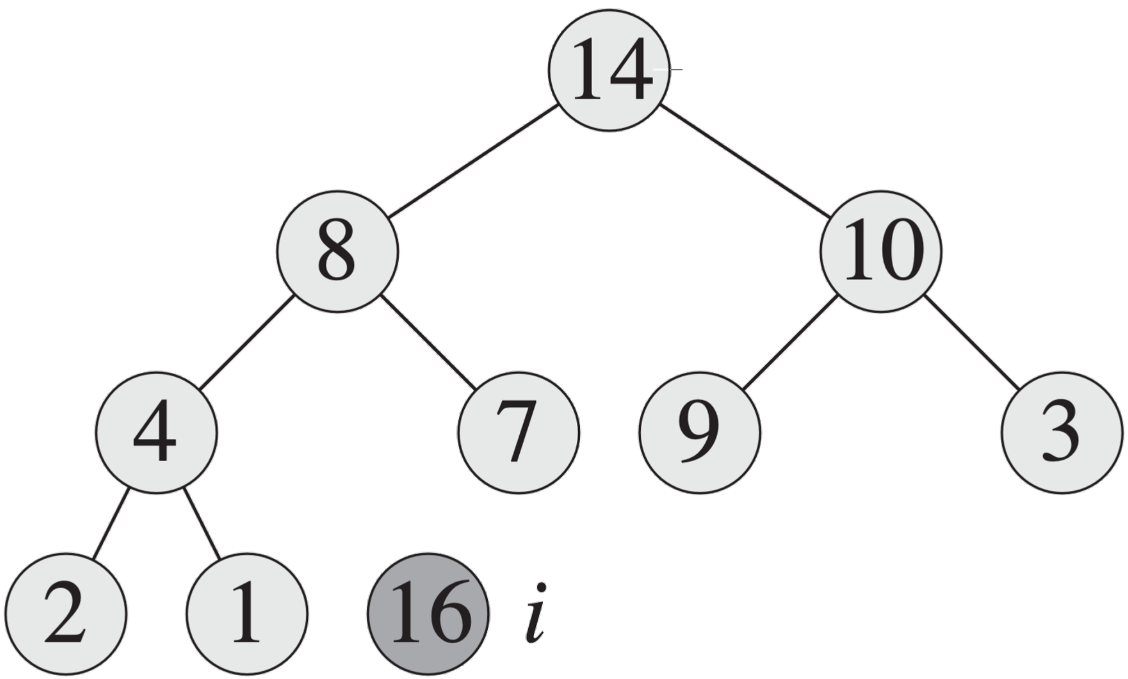
核心思路：先建堆，然后不断地将堆顶元素和堆尾元素互换，减小堆的大小，并调用 MAX-HEAPIFY(A, 1) 维护最大堆的性质。



(a)

堆排序的过程

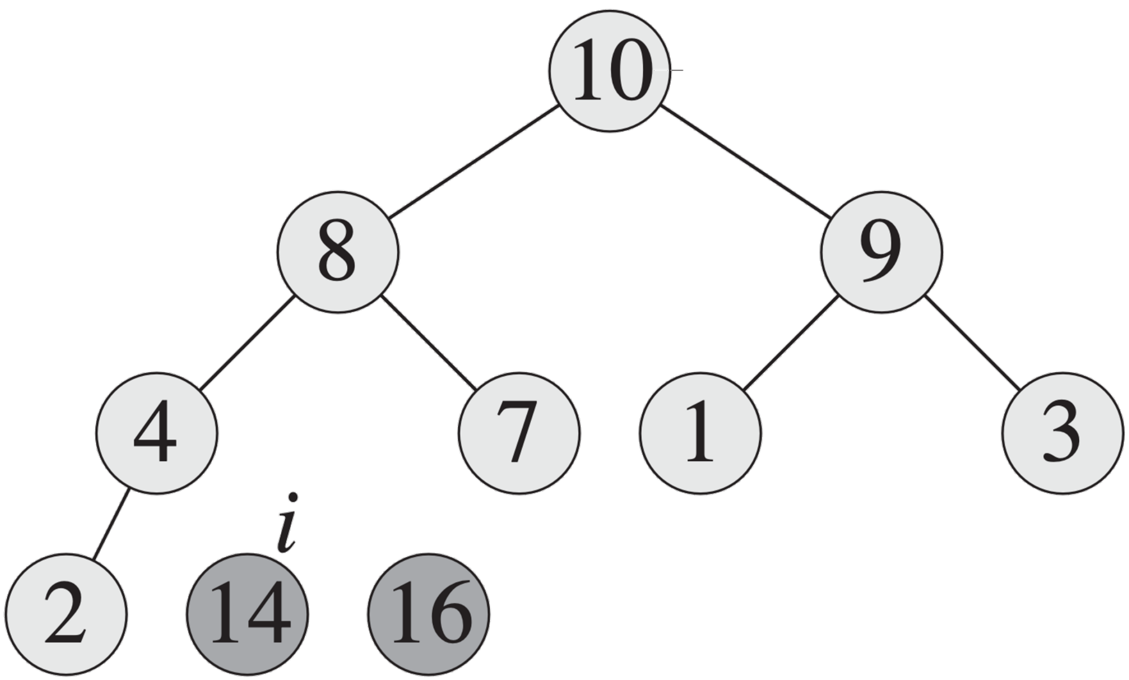
核心思路：先建堆，然后不断地将堆顶元素和堆尾元素互换，减小堆的大小，并调用 MAX-HEAPIFY(A, 1) 维护最大堆的性质。



(b)

堆排序的过程

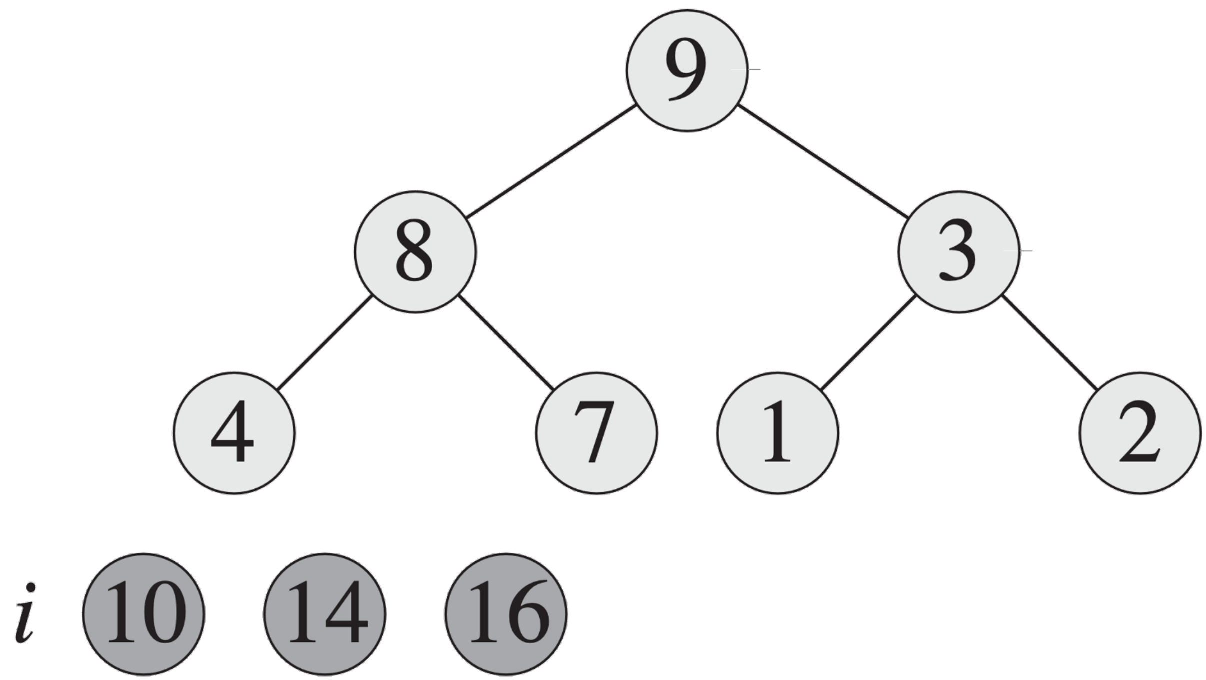
核心思路：先建堆，然后不断地将堆顶元素和堆尾元素互换，减小堆的大小，并调用 MAX-HEAPIFY(A, 1) 维护最大堆的性质。



(c)

堆排序的过程

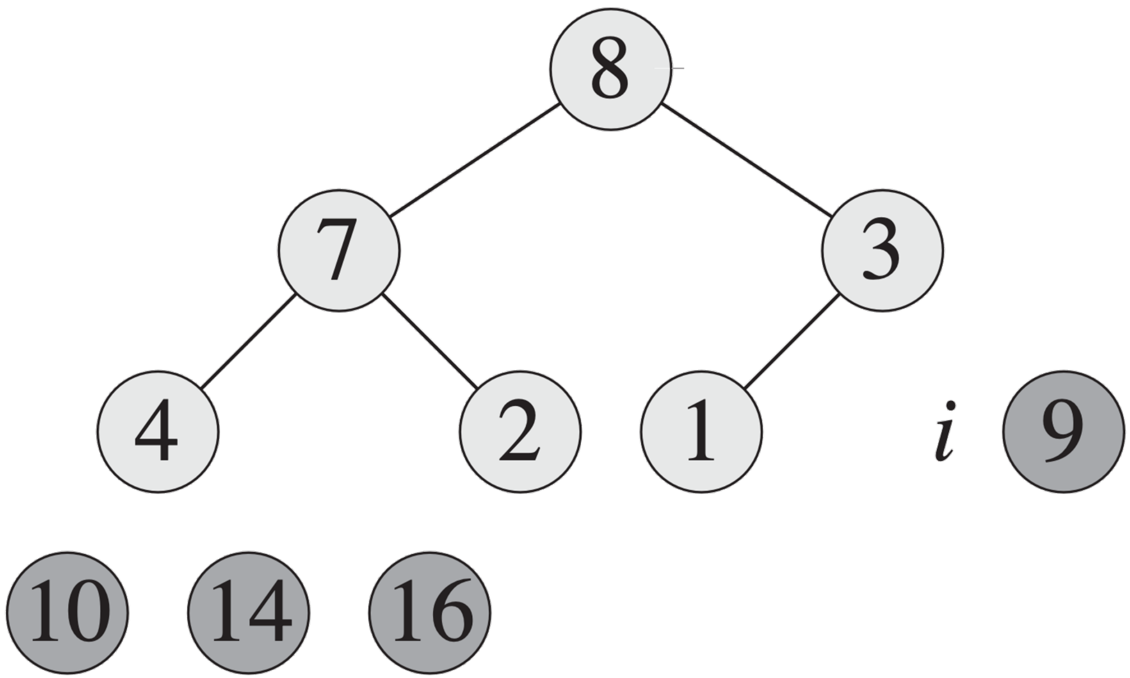
核心思路：先建堆，然后不断地将堆顶元素和堆尾元素互换，减小堆的大小，并调用 MAX-HEAPIFY(A, 1) 维护最大堆的性质。



(d)

堆排序的过程

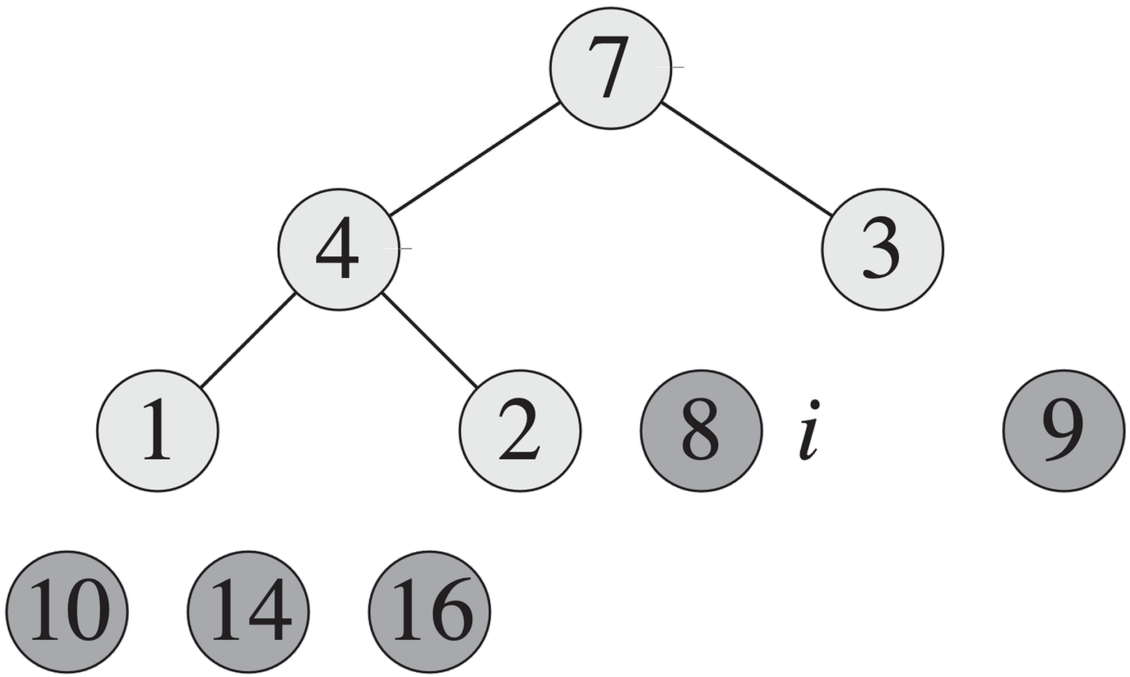
核心思路：先建堆，然后不断地将堆顶元素和堆尾元素互换，减小堆的大小，并调用 MAX-HEAPIFY(A, 1) 维护最大堆的性质。



(e)

堆排序的过程

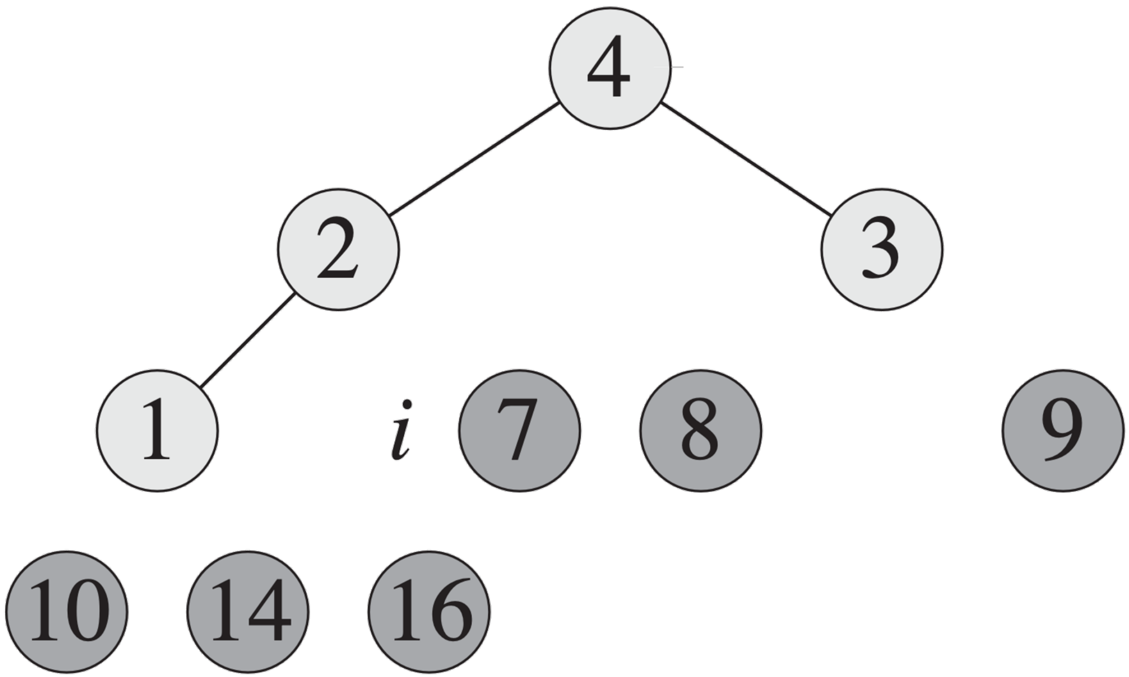
核心思路：先建堆，然后不断地将堆顶元素和堆尾元素互换，减小堆的大小，并调用 MAX-HEAPIFY(A, 1) 维护最大堆的性质。



(f)

堆排序的过程

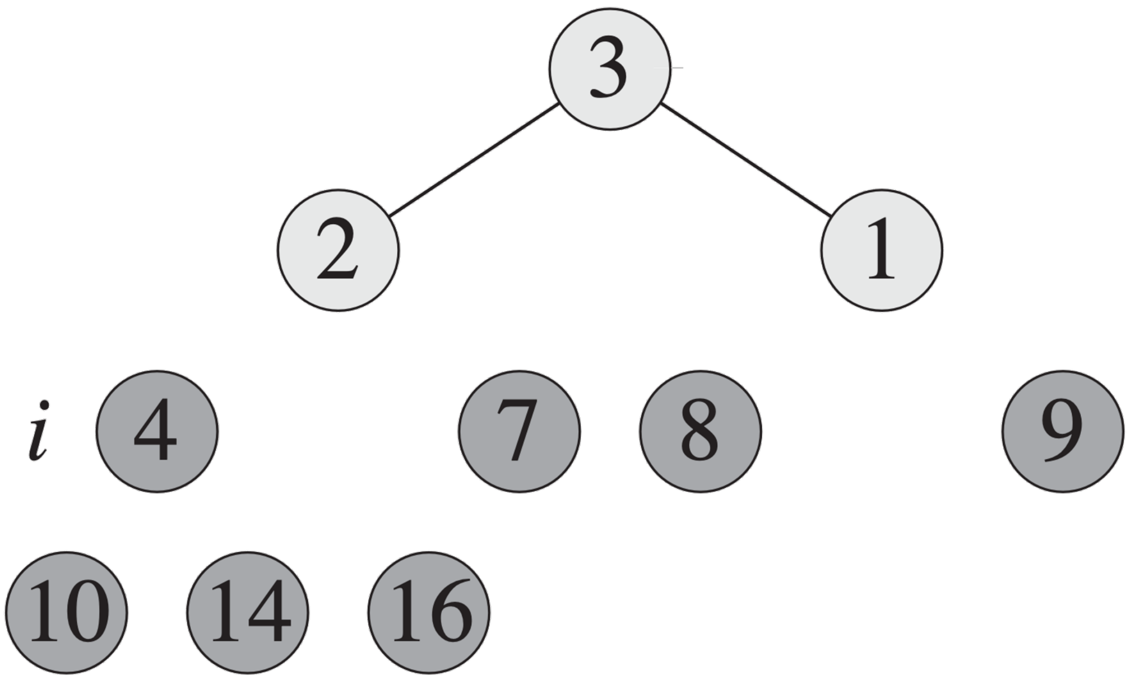
核心思路：先建堆，然后不断地将堆顶元素和堆尾元素互换，减小堆的大小，并调用 MAX-HEAPIFY(A, 1) 维护最大堆的性质。



(g)

堆排序的过程

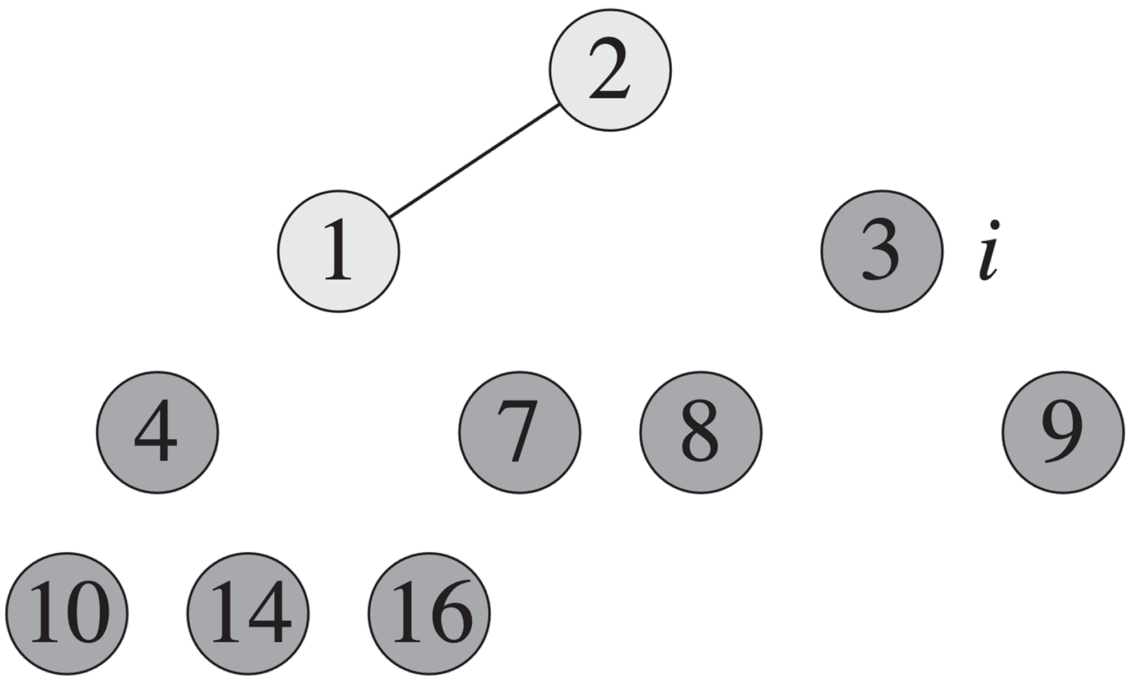
核心思路：先建堆，然后不断地将堆顶元素和堆尾元素互换，减小堆的大小，并调用 MAX-HEAPIFY(A, 1) 维护最大堆的性质。



(h)

堆排序的过程

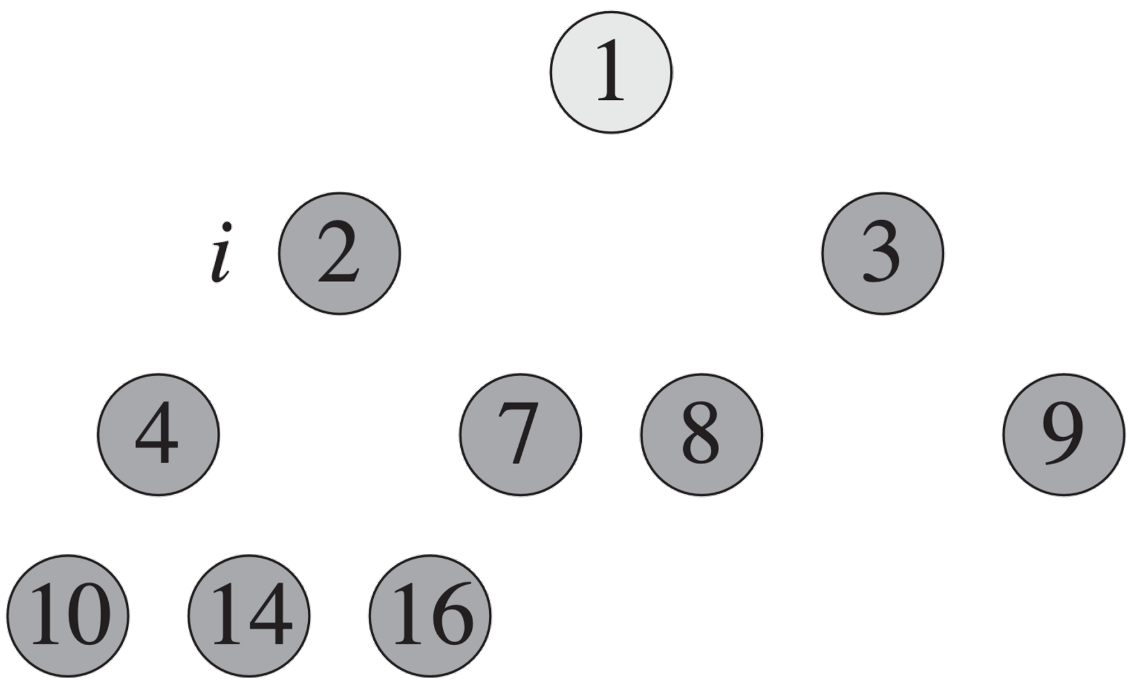
核心思路：先建堆，然后不断地将堆顶元素和堆尾元素互换，减小堆的大小，并调用 MAX-HEAPIFY(A, 1) 维护最大堆的性质。



(i)

堆排序的过程

核心思路：先建堆，然后不断地将堆顶元素和堆尾元素互换，减小堆的大小，并调用 MAX-HEAPIFY(A, 1) 维护最大堆的性质。



(j)

堆排序的过程

核心思路：先建堆，然后不断地将堆顶元素和堆尾元素互换，减小堆的大小，并调用 MAX-HEAPIFY(A, 1) 维护最大堆的性质。

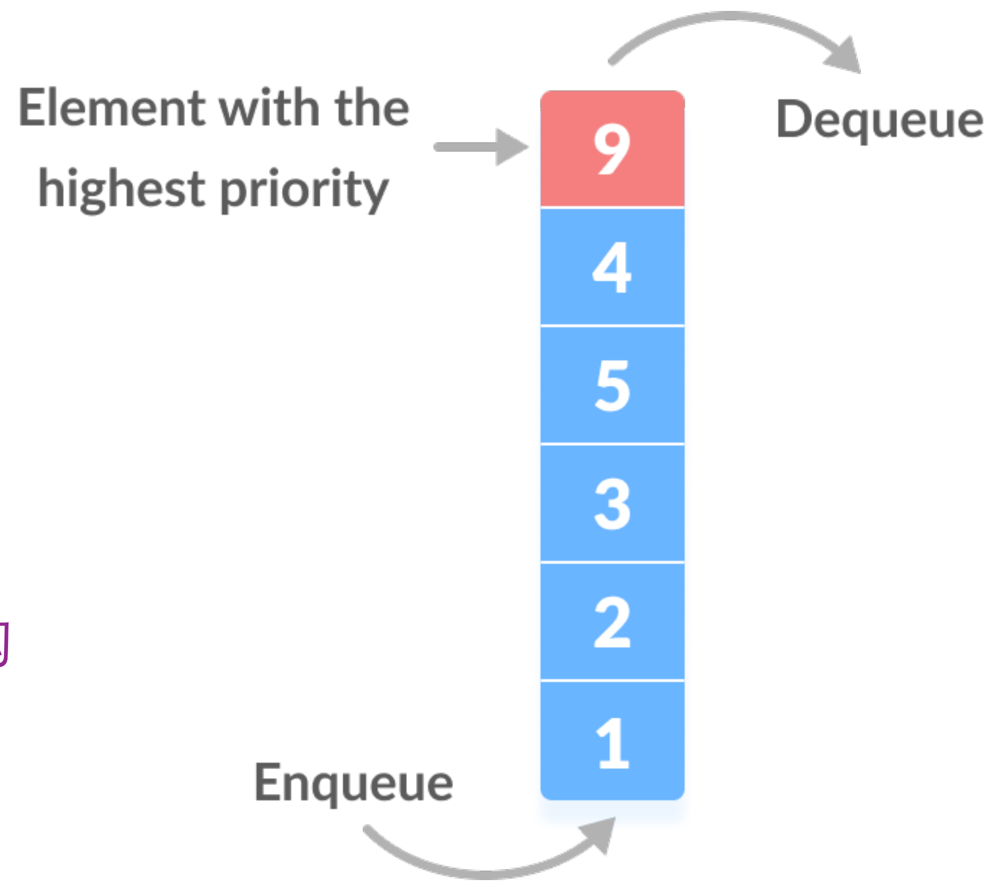
A	1	2	3	4	7	8	9	10	14	16

(k)

堆排序算法的运行时间

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

- HEAPSORT 算法的时间复杂度为 $O(n \log n)$ 。
    - 第 1 行调用 BUILD-MAX-HEAP 的时间复杂度为 $O(n)$ 。
    - 第 2 到 5 行的 **for** 循环调用了 $n-1$ 次 MAX-HEAPIFY， 每次的代价是 $O(\log n)$， 总代价为 $O(n \log n)$ 。



优先队列

一种超好用的数据结构

优先队列（priority queue）

优先队列是一种用来维护由一组元素构成的集合 S 的数据结构，其中的每一个元素都有一个相关的值，称为关键字（key）。

一个最大优先队列支持以下操作（最小优先队列与之是对偶的）：

- $INSERT(S,x)$：把元素 x 插入集合 S 中，即 $S = S \cup \{x\}$。
- $MAXIMUM(S)$：返回 S 中具有最大关键字的元素。
- $EXTRACT-MAX(S)$：去掉并返回具有最大关键字的元素。
- $INCREASE-KEY(S,x,k)$：将元素 x 的关键字值增加到 k，这里假设 k 的值不小于原关键字值。

最大优先队列是一种抽象数据类型（abstract data type），其实现方式是未定的，只有在指定实现方式之后，才可以讨论各个操作的运行时间。

优先队列的应用

- 最大优先队列
  - INSERT, MAXIMUM, EXTRACT-MAX, INCREASE-KEY
  - 例：共享计算机系统的作业调度——抢占式
- 最小优先队列
  - INSERT, MINIMUM, EXTRACE-MIN, DECRASE-KEY
  - 例：基于事件驱动的模拟器，最小生成树，单源点最短路径
- 一些编程细节
  - 在编程实践中，卫星数据较多的时候，我们通常是操作数据对象的句柄（handle），而不是数据对象本身。
  - 一个对象的句柄是可以访问该对象的“轻量”的入口。

使用最大堆实现最大优先队列

- 实现 $O(1)$ 的 MAXIMUM 操作：

```
HEAP-MAXIMUM(A)
return A[1]
```

- 实现 $O(\log n)$ 的 EXTRACT-MAX 操作：
  - 思路同 HEAPSORT 第 3 到 5 行的循环体部分。

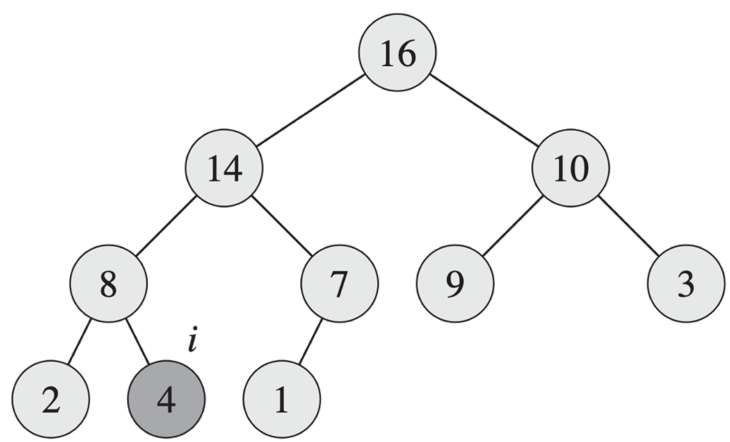
```
HEAP-EXTRACT-MAX(A)
if A.heap-size < 1
    error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size - 1
MAX-HEAPIFY(A, 1)
return max
```

使用最大堆实现最大优先队列

- 实现 $O(\log n)$ 的 INCREASE-KEY 操作：

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 $A[\text{PARENT}(i)] < A[i]$
    exchange $A[i]$ with $A[\text{PARENT}(i)]$
    $i = \text{PARENT}(i)$

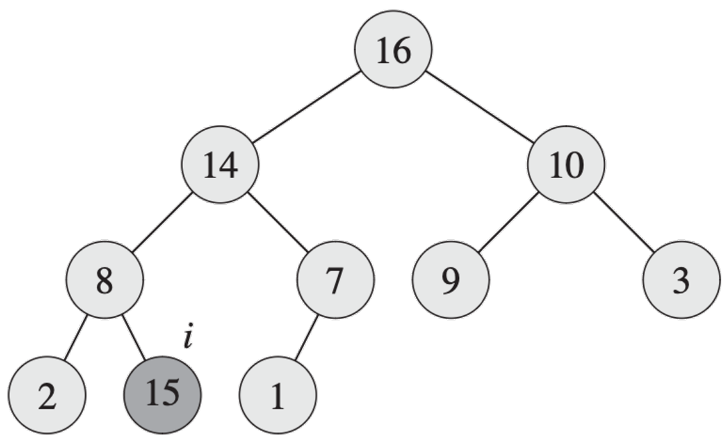


(a)

使用最大堆实现最大优先队列

- 实现 $O(\log n)$ 的 INCREASE-KEY 操作：

```
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)
```

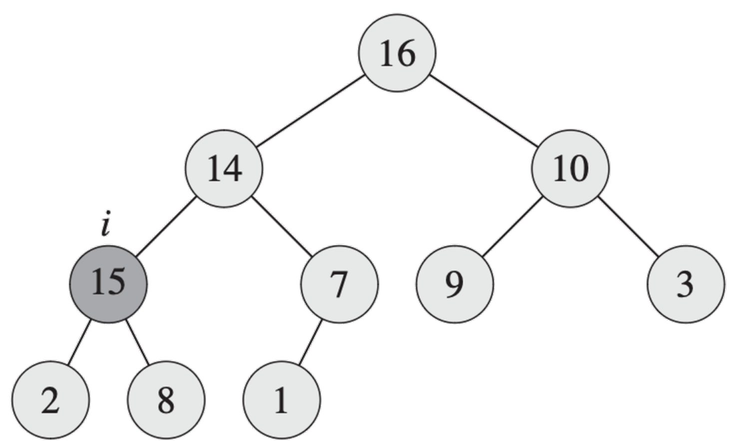


(b)

使用最大堆实现最大优先队列

- 实现 $O(\log n)$ 的 INCREASE-KEY 操作：

```
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)
```



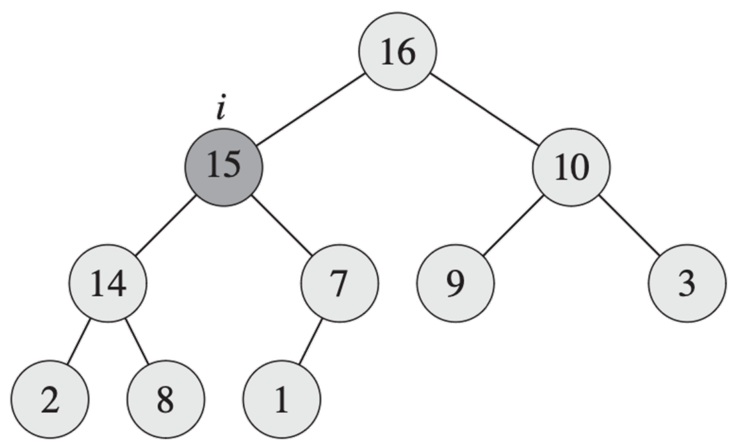
(c)

使用最大堆实现最大优先队列

- 实现 $O(\log n)$ 的 INCREASE-KEY 操作：

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 $A[\text{PARENT}(i)] < A[i]$
    exchange $A[i]$ with $A[\text{PARENT}(i)]$
    $i = \text{PARENT}(i)$



(d)

使用最大堆实现最大优先队列

- 实现 $O(\log n)$ 的 INCREASE-KEY 操作：

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

- 4 到 6 行 **while** 循环的思路同插入排序内部的 **while** 循环。
- 循环不变式：每次迭代开始时，子数组$A[1..A.heap-size]$要满足最大堆的性质。如有违背，只有一个可能：$A[i] > A[PARENT(i)]$ 。

(详见练习5-2问题2)

使用最大堆实现最大优先队列

- 实现 $O(\log n)$ 的 INSERT 操作：
  - 思路：先通过关键字为 $-\infty$ 的结点来扩展最大堆，然后调用 INCREASE-KEY 将其关键字增加到待插入的值即可。

```
MAX-HEAP-INSERT(A, key)
A.heap-size = A.heap-size + 1
A[A.heap-size] = −∞
HEAP-INCREASE-KEY(A, A.heap-size, key)
```

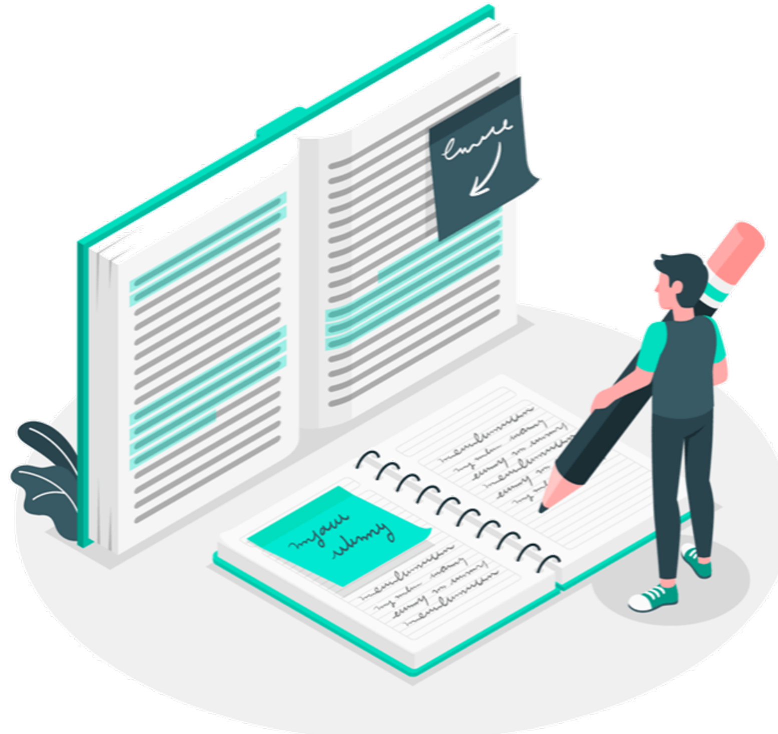
- 在一个用堆实现的优先队列中，所有的操作都可以在 $O(\log n)$ 以内完成。
  - 具体地，MAXIMUN / MINIMUM 复杂度为 $O(1)$；
  - EXTRACT-MAX/MIN，INCREASE/DECREASE-KEY，INSERT 复杂度为 $O(\log n)$。

另一种实现

- 使用顺序表来实现优先队列的算法：
  - INSERT 操作：直接插在末尾，复杂度为 $O(1)$。
  - MAXIMUM 操作：遍历取最大值，复杂的为 $O(n)$ 。
  - EXTRACT-MAX 操作：清除最大值元素，复杂度为 $O(n)$ 。
  - INCREASE-KEY 操作：直接修改元素，复杂度为 $O(1)$ 。

操作	堆实现	顺序表实现
INSERT	$O(\log n)$	$O(1)$
MAXIMUM	$O(1)$	$O(n)$
EXTRACT-MAX	$O(\log n)$	$O(n)$
INCREASE-KEY	$O(\log n)$	$O(1)$

本讲小结



内容提要

- 学习“堆”数据结构
  - 最大堆与最小堆
  - 维护堆的性质：MAX-HEAPIFY
  - 建堆：BUILD-MAX-HEAP
- 堆排序算法
  - 时间：最坏情况 $O(n\log n)$ ——归并排序。
  - 空间：原址（in place）排序，只需常数项额外空间——插入排序。
  - 兼备“归并排序”和“插入排序”之长。
- 优先队列及其实现
  - 同一数据结构（抽象数据类型）的不同实现具有不同的复杂度。

The End!