数据结构与算法设计

周　可

Mail：zhke@hust.edu.cn

华中科技大学，武汉光电国家研究中心

3.1 堆排序

3.2 Priority queues

Heaps

The *(binary) heap* data structure is an array object that we can view as a nearly complete binary tree

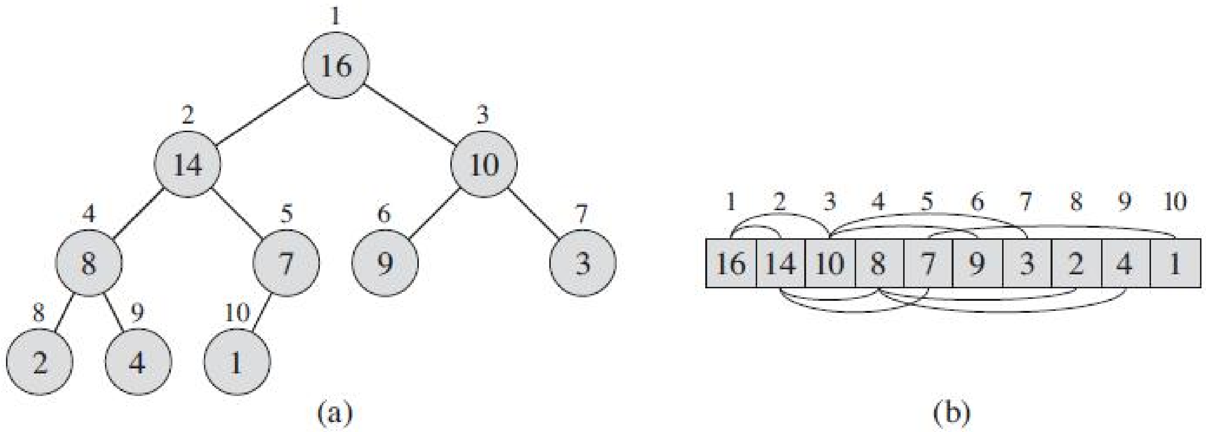


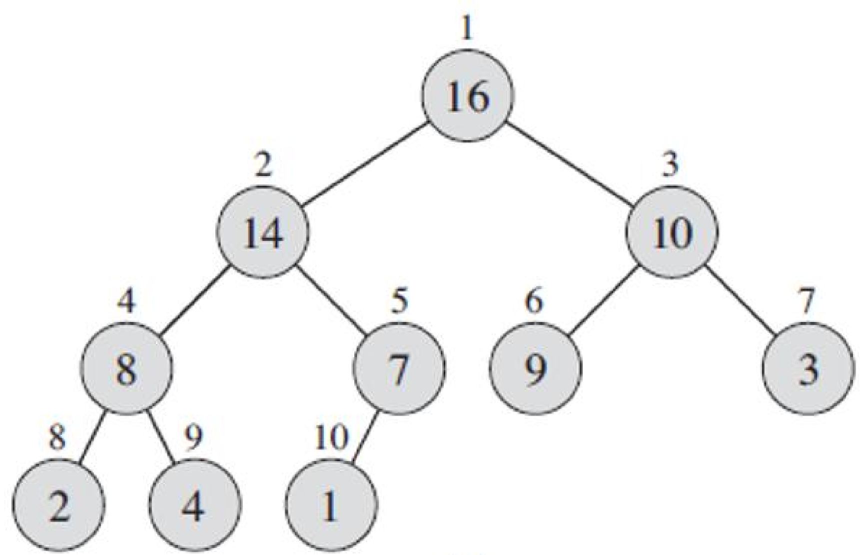
Figure 2.1 A max-heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children. The tree has height three; the node at index 4 (with value 8) has height one.

The root of the tree is A[1], and given the index i of a node, we can easily compute the indices of its parent, left child, and right child:

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

LEFT(i)
return $2i$

RIGHT(i)
return $2i + 1$



There are two kinds of binary heaps: max-heaps and min-heaps.

- **max-heap**: The largest element in a max-heap is stored at the root, and the subtree rooted at a node contains values no larger than that contained at the node itself.
- **min-heap**: The smallest element in a min-heap is at the root
- the ***height*** of a heap is the height of the binary tree. That is O(lg$n$)

- The procedure MAX-HEAPIFY will maintain the max-heap property.
- Assume the binary trees rooted at LEFT[i] and RIGHT[i] are submaxheaps

```
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)
```

The running time of MAX-HEAPIFY is $O(\lg n)$

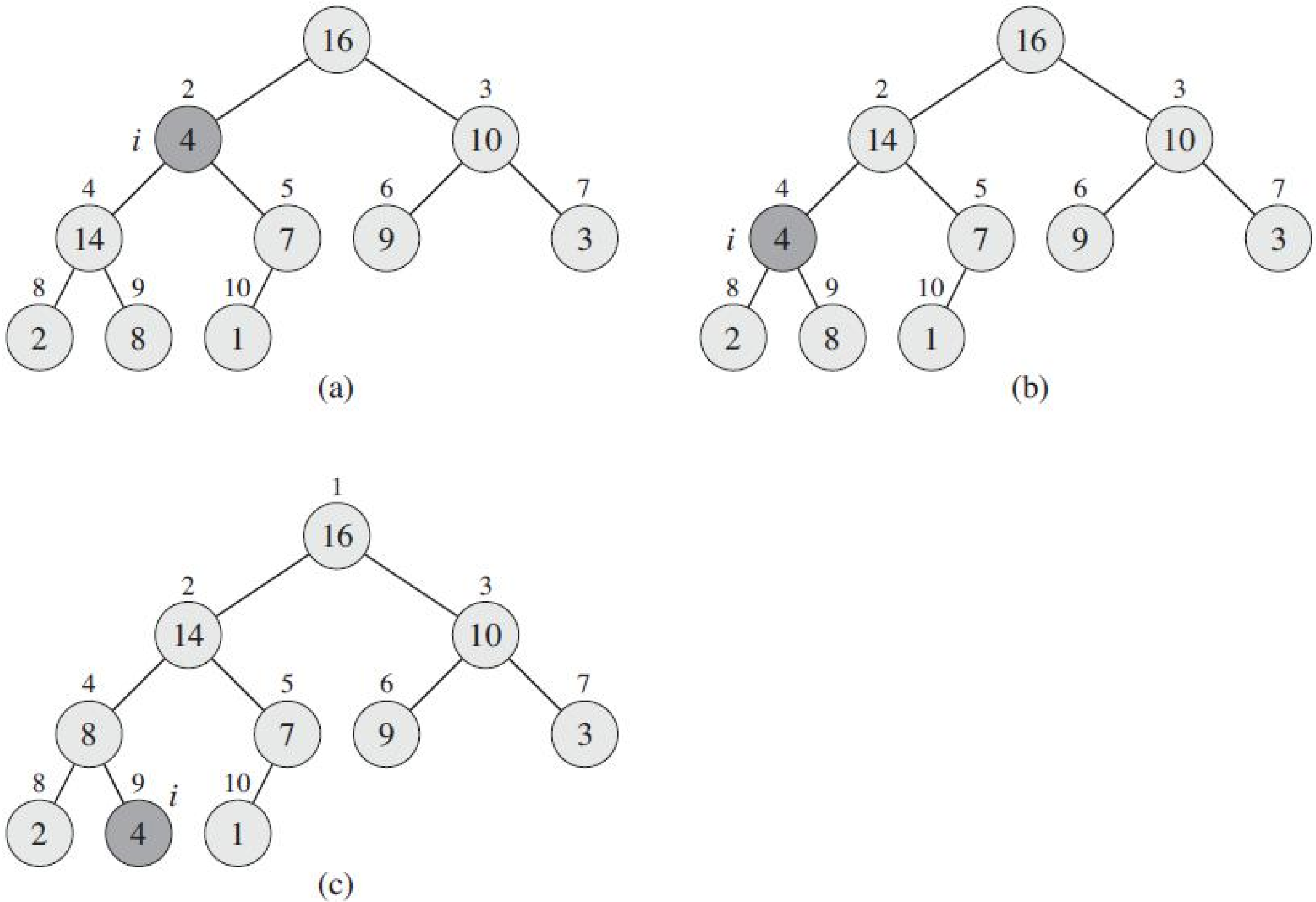


Figure 6.2 The action of MAX-HEAPIFY$(A, 2)$, where $A.heap\text{-}size = 10$. **(a)** The initial configuration, with $A[2]$ at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in **(b)** by exchanging $A[2]$ with $A[4]$, which destroys the max-heap property for node 4. The recursive call MAX-HEAPIFY$(A, 4)$ now has $i = 4$. After swapping $A[4]$ with $A[9]$, as shown in **(c)**, node 4 is fixed up, and the recursive call MAX-HEAPIFY$(A, 9)$ yields no further change to the data structure.

Building a heap

- We can use the procedure MAX-HEAPIFY in a bottom-up manner to convert an array A[1.. n], where n= A.length, into a max-heap.

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

- The Subarray A[|n/2|+1…n] are all leave of the tree. The procedure BUILD-MAX-HEAP goes through the remaining nodes of the tree and runs MAX-HEAPIFY on each one.

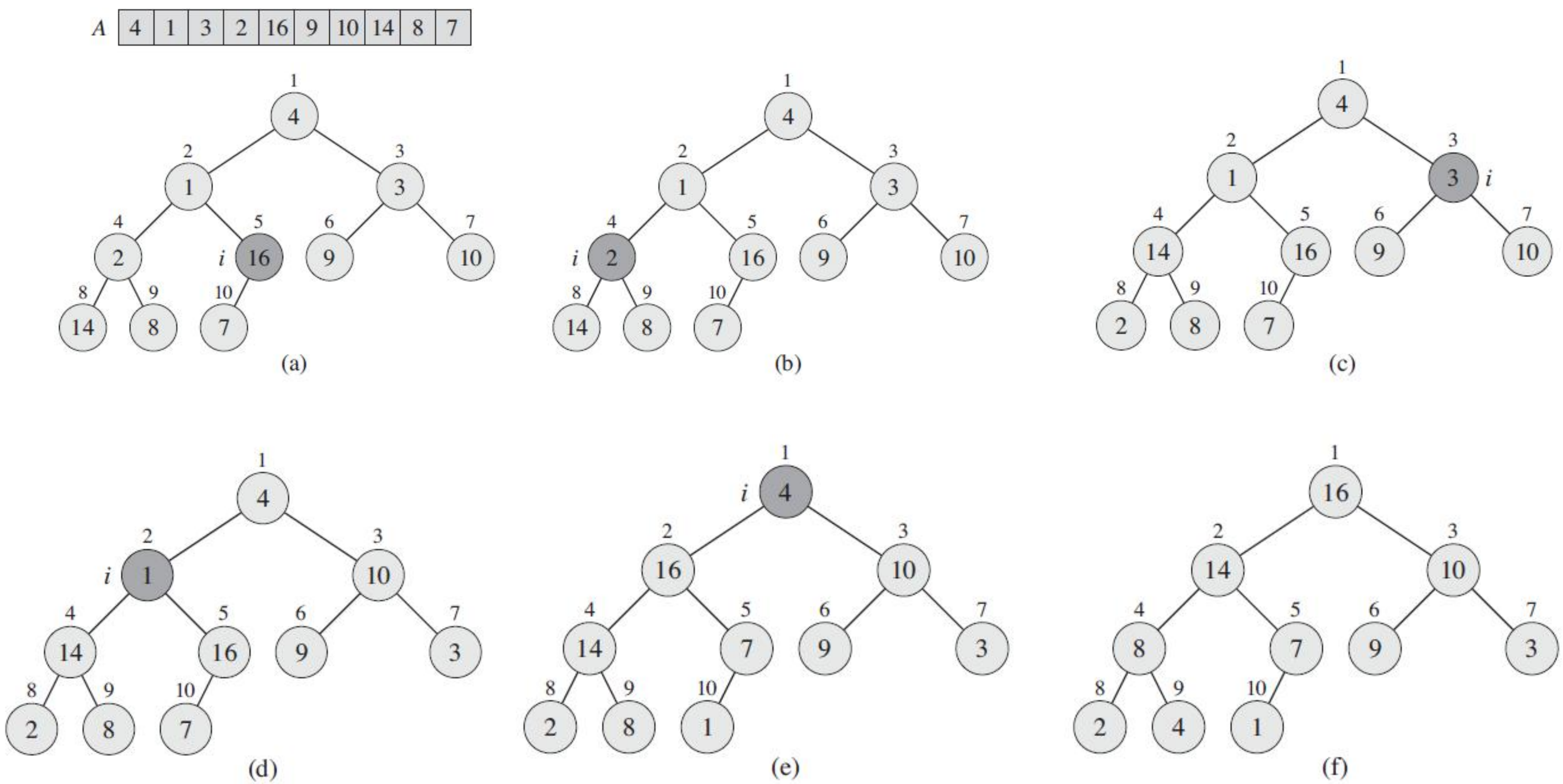


Figure 6.3 The operation of BUILD-MAX-HEAP, showing the data structure before the call to MAX-HEAPIFY in line 3 of BUILD-MAX-HEAP. **(a)** A 10-element input array A and the binary tree it represents. The figure shows that the loop index i refers to node 5 before the call MAX-HEAPIFY(A, i). **(b)** The data structure that results. The loop index i for the next iteration refers to node 4. **(c)–(e)** Subsequent iterations of the **for** loop in BUILD-MAX-HEAP. Observe that whenever MAX-HEAPIFY is called on a node, the two subtrees of that node are both max-heaps. **(f)** The max-heap after BUILD-MAX-HEAP finishes.

- we can build a max-heap from an unordered array in **linear time**.
    - The time required by **MAX-HEAPIFY** when called on a node of height h is **O(h)**, and
    - so we can express the total cost of BUILD-MAX-HEAP as being bounded from above by

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

We evalaute the last summation by substituting $x = 1/2$ in the formula (A.8), yielding

$$\begin{aligned} \sum_{h=0}^{\infty} \frac{h}{2^h} &= \frac{1/2}{(1-1/2)^2} \\ &= 2 . \end{aligned}$$

Thus, we can bound the running time of BUILD-MAX-HEAP as

$$\begin{aligned} O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) &= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) \\ &= O(n) . \end{aligned}$$

- How to build a min-heap?

The heapsort algorithm

① First, using BUILD-MAX-HEAP to build a max-heap on the input array A[1..n], where n =A.*length*.

② Then, put the root, the maximum element, into its correct final position A[n].

③ And then call MAX-HEAPIFY(A,1) to rebuild a max-heap in A[1..n-1].

④ Repeats this process for the max-heap of size n-1 down to a heap of size 2.

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

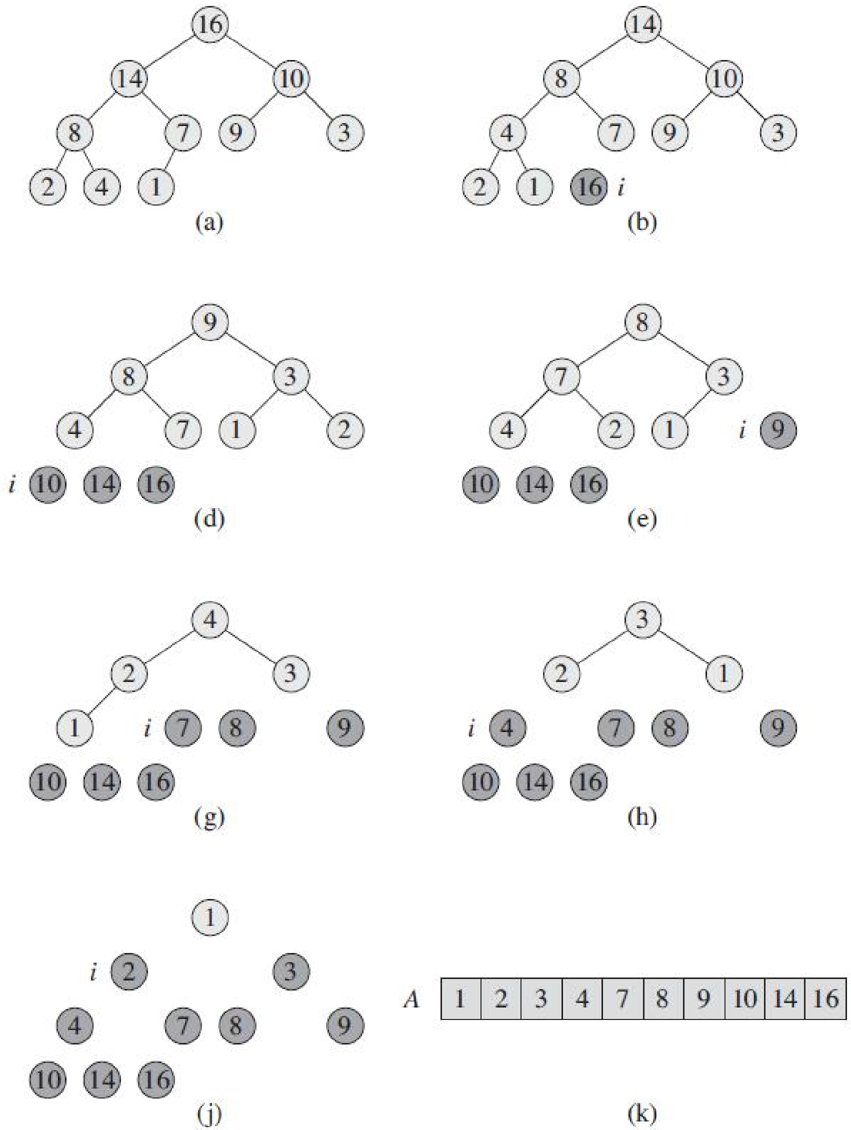
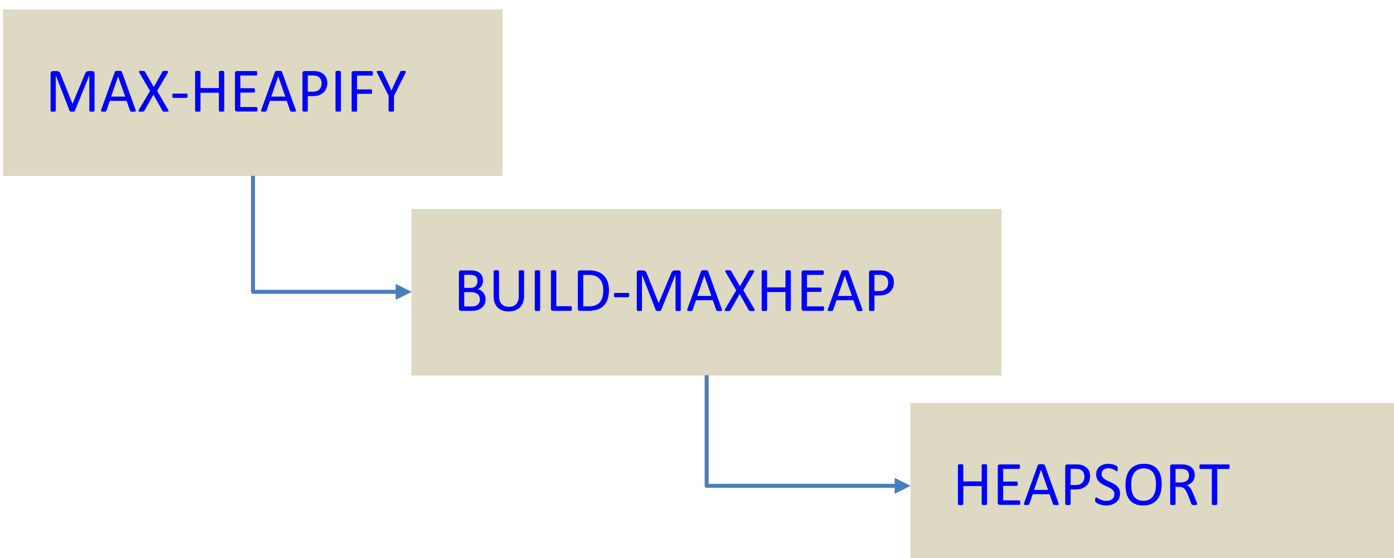


Figure 6.4 The operation of HEAPSORT. **(a)** The max-heap data structure just after BUILD-MAX-HEAP has built it in line 1. **(b)–(j)** The max-heap just after each call of MAX-HEAPIFY in line 5, showing the value of i at that time. Only lightly shaded nodes remain in the heap. **(k)** The resulting sorted array A.

The HEAPSORT procedure takes time O(nlg*n*), since the call to BUILD-MAXHEAP takes time O(n) and each of the n -1 calls to MAX-HEAPIFY takes time O(lg*n*).

<u>What is the LOGIC CHAIN of above three procedures?</u>



3.1 堆排序

3.2 Priority queues

- A ***priority queue is a application of heap as a data structure.***
- A ***priority queue*** is a data structure for maintaining a set S of elements, each with an associated value called a ***key***.
- A ***max-priority queue*** supports the following operations:

$\boxed{\text{INSERT}(S, x)}$ inserts the element x into the set S, which is equivalent to the operation $S = S \cup \{x\}$.

$\boxed{\text{MAXIMUM}(S)}$ returns the element of S with the largest key.

$\boxed{\text{EXTRACT-MAX}(S)}$ removes and returns the element of S with the largest key.

$\boxed{\text{INCREASE-KEY}(S, x, k)}$ increases the value of element x's key to the new value k, which is assumed to be at least as large as x's current key value.

A max-priority queue can be implemented by max-heap.

1) HEAP-MAXIMUM implements the MAXIMUM operation in Θ(1) time.

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

2) The procedure HEAP-EXTRACT-MAX implements the EXTRACT-MAX operation in O(lg*n*) time.

```
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
```

The procedure HEAP-INCREASE-KEY implements the INCREASE-KEY operation in O(lgn) time.

```
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)
```

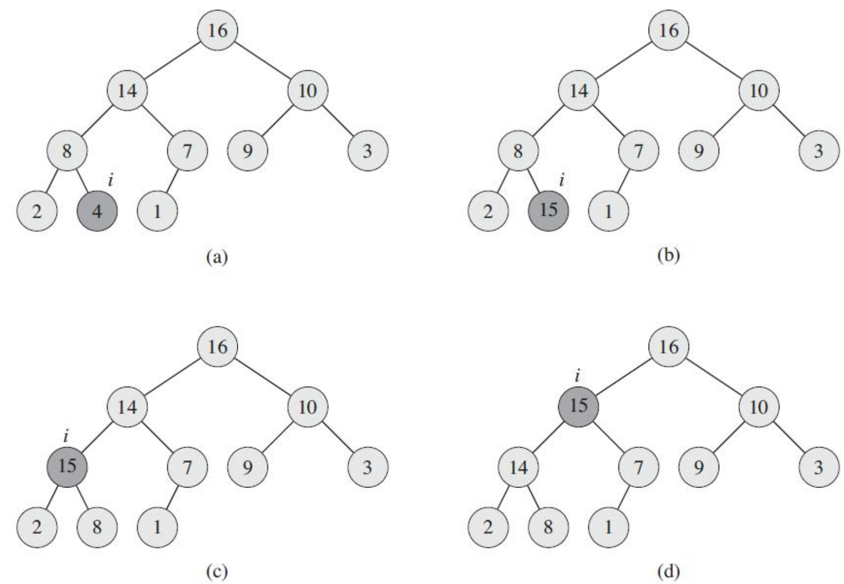


Figure 6.5 The operation of HEAP-INCREASE-KEY. **(a)** The max-heap of Figure 6.4(a) with a node whose index is i heavily shaded. **(b)** This node has its key increased to 15. **(c)** After one iteration of the **while** loop of lines 4–6, the node and its parent have exchanged keys, and the index i moves up to the parent. **(d)** The max-heap after one more iteration of the **while** loop. At this point, $A[\text{PARENT}(i)] \geq A[i]$. The max-heap property now holds and the procedure terminates.

The procedure MAX-HEAP-INSERT implements the INSERT operation

MAX-HEAP-INSERT(A, key)

1 $A.heap\text{-}size = A.heap\text{-}size + 1$
2 $A[A.heap\text{-}size] = -\infty$
3 HEAP-INCREASE-KEY$(A, A.heap\text{-}size, key)$

- The procedure first expands the max-heap by adding to the tree a new leaf whose key is -∞. Then it calls HEAP-INCREASE-KEY to set the key of this new node to its correct value and maintain the max-heap property.

作业：6.4-1

Thank You!

Q&A