Chap 6. Heapsort 孫光天

Combine the better attributes of two sorting algorithms (merge sort & insertion sort) into heapsort. Its running time is O(n log n).

6.1 Heaps

The (binary) heap data structure is an array object that can be viewed as a nearly complete binary tree except the lowest (see Figure 6.1). The heap-size[A]: the number of elements in the heap stored within array A.

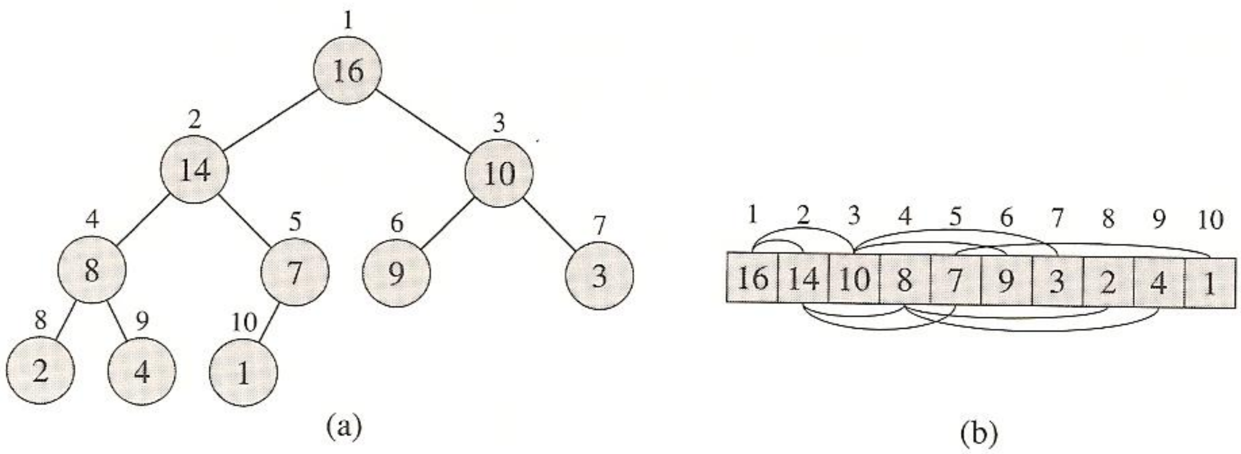

Figure 6.1 A max-heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; 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.

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

LEFT(i)
  return $2i$

RIGHT(i)
  return $2i + 1$

Two kinds of heaps: max-heaps and min-heaps.

The largest element in a max-heap is stored at the root, and the substree rooted at a node contains values no larger than that contained at the node itself.（最大元素放在 **root** 所形成的二元樹：**max-heaps**）

(min-heaps in the opposite way)

The height of a node in a heap to be the number of edges on the longest simple downward path from the node to a leaf. (從 **node** 到 **leaf**，非到 **root)**。**The tree height is $\Theta(\lg n)$.**

Four procedure are used for heap sort:

1. **MAX-HEAPIFY runs in $O(\lg n)$ time, maintains the max-heap property. (**大的元素放在樹(或子樹)的 **root** 上**)**
2. **BUILD-MAX-HEAP runs in linear time, produces a max-heap from an unsorted input array.**
3. **HEAPSORT runs in $O(n \log n)$, sorts an array in place.**
4. **MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY, and HEAP-MAX run in $O(\lg n)$, allow the heap data structure to be used as a priority queue.**

6.2 Maintaining the heap property

Manipulate max-heap (處理一個 **heap): Let A[i] is larger than LEFT[i] & RIGHT[i](node *i*** 比其子樹 **LEFT[*i*] & RIGHT[*i*]**大**)**

(兩個子樹中，最大往上移，小的往下移動**)**

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

內容互換
繼續往下

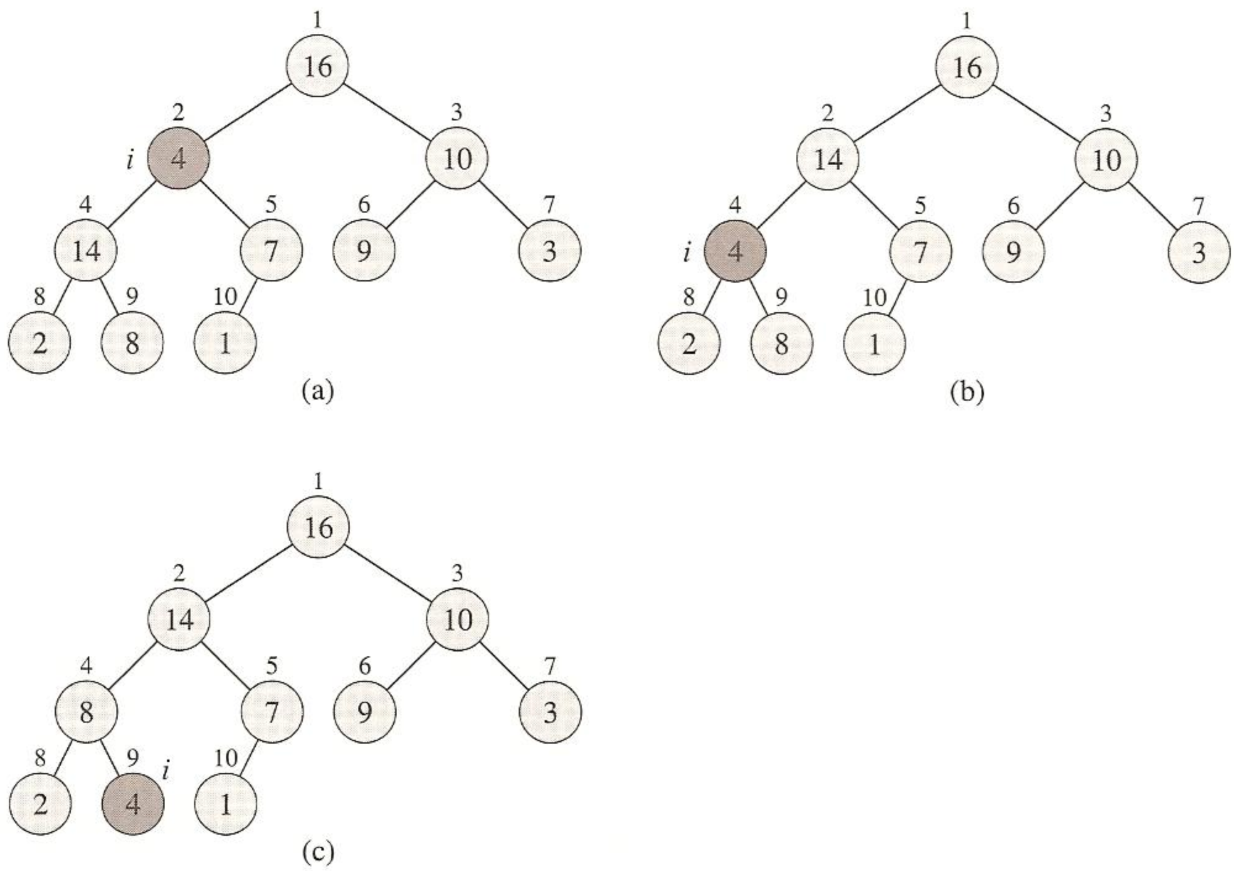

Figure 6.2 The action of MAX-HEAPIFY$(A, 2)$, where *heap-size*$[A] = 10$. **(a)** The initial configuration, with $A[2]$ at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in **(b)** by exchanging $A[2]$ with $A[4]$, 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.

Lines 1~ 2: *l*, *r* 爲 node *i* 左右子樹的指標(index).

The children's substree each have size at most 2n/3. The worst case occurs when the last row of the tree is exactly half full – and the running time of MAX-HEAPIFY can therefore be described by the recurrence:

$$T(n) \le T(2n/3) + \Theta(1)$$
$$= O(\lg n)$$

6.3 Building a heap

Elements in the subarray $A[(\lfloor n/2 \rfloor + 1) .. n]$ are all leaves of the tree. The remaining nodes of the tree and runs MAX-HEAPIFY on each one. (heap 元素，後半段均爲 葉(leaves)節點，所以，要建 heap，只需考慮的 root(包含子樹)元素爲 $1 \sim \lfloor n/2 \rfloor$)

```
BUILD-MAX-HEAP(A)
1  heap-size[A] ← length[A]
2  for i ← ⌊length[A]/2⌋ downto 1
3       do MAX-HEAPIFY(A, i)
```

動動腦：
***i* 可以從 1 to *n*/2 嗎？**
" for *i* ← 1 to *length*[A]/2" ?

Figure 6.3 shows an example of the action of BUILD-MAX-HEAP.

Each call to MAX-HEAPIFY costs O(lg n) and there are O(n) such call. Thus, the running time is O(n lg n). But it is not asymptotically tight. (order 估算太高，因爲不是每個 node 高度都 O(lg n)). Each node has various heights, and the heights of most nodes are small. For an n-element heap has height $\lfloor \lg n \rfloor$ and at most $\lceil n/2^{h+1} \rceil$ nodes of any height h. (高度是由底往上數，所以，leaves 葉節點高度爲 0，個數有 $\lceil n/2 \rceil$ 一半)。

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

The last summation can be evaluated by substituting $x = 1/2$ in the formula (A.8), which yields

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

$\boxed{(\sum k X^k = X / (1-X)^2) \quad X=1/2\text{，p.1148 (A.8)}}$

Thus, the running time of BUILD-MAX-HEAP can be bounded as

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

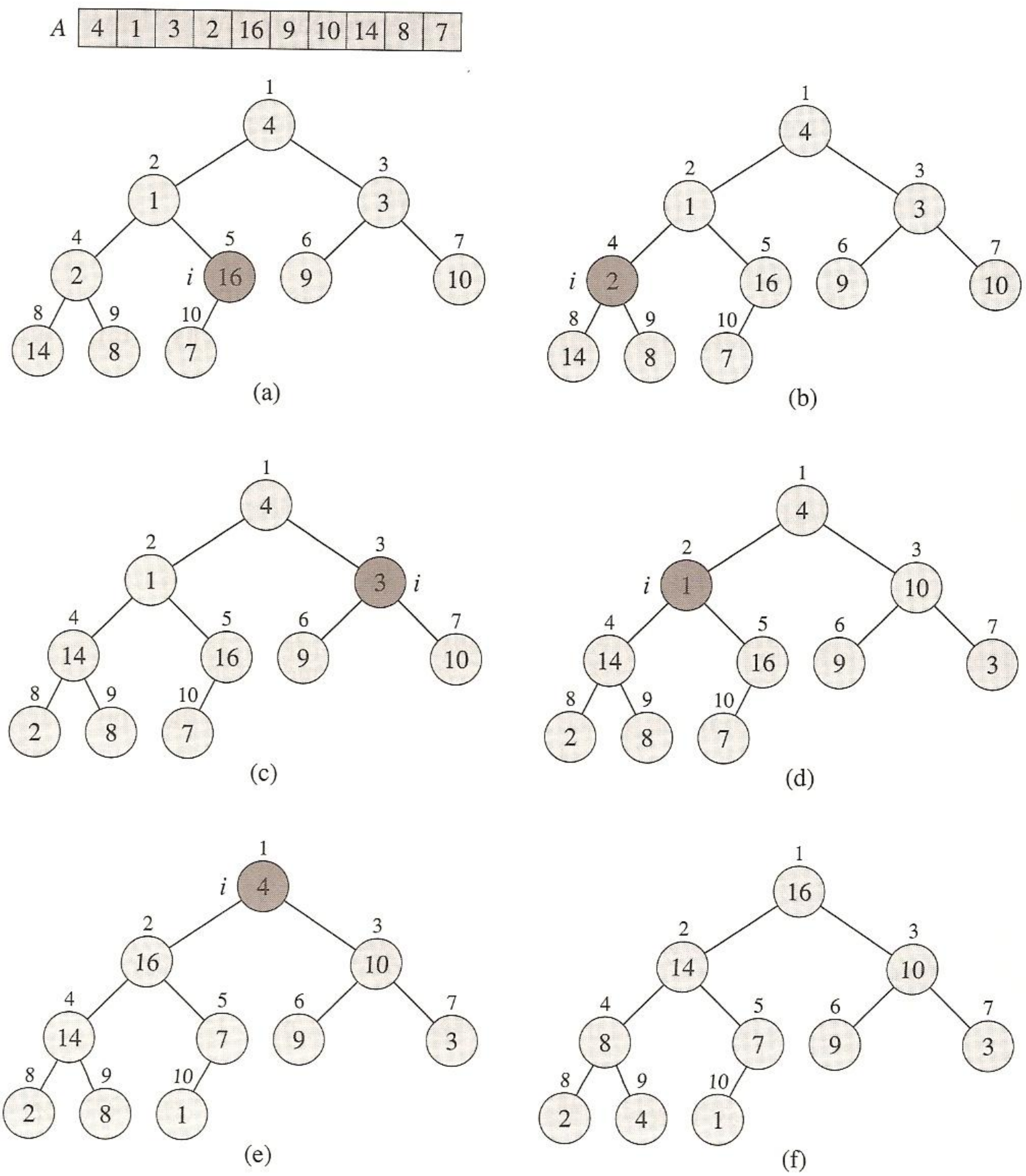

Figure 6.3 The operation of BUILD-MAX-HEAP, showing the data structure before the call to MAX-HEAPIFY in line 3 of BUILD-MAX-HEAP. **(a)** A 10-element input array A and the binary tree it represents. The figure shows that the loop index i refers to node 5 before the call MAX-HEAPIFY(A, i). **(b)** The 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.

6.4 The heapsort algorithm

We select the root of heap and move to the last element *i*,(從小排到大，所以，最大 element (root A[1]) 移到最後 A[i]，heap-size 減一) and one call to MAX-HEAPIFY after extract the root of heap. Then, the heap-size is decreased by one.

```
HEAPSORT(A)
  BUILD-MAX-HEAP(A)
  for i ← length[A] downto 2
       do exchange A[1] ↔ A[i]
          heap-size[A] ← heap-size[A] − 1
          MAX-HEAPIFY(A, 1)
```

Figure 6.4 shows the procedure. (1 與 16 互換，1 到 root，16 移到最後 A[i]，呼叫 MAX-HEAPIFY，1 往下移：1 與 14 互換，1 與 8 互換，1 與 4 互換，1 到達 leaf)

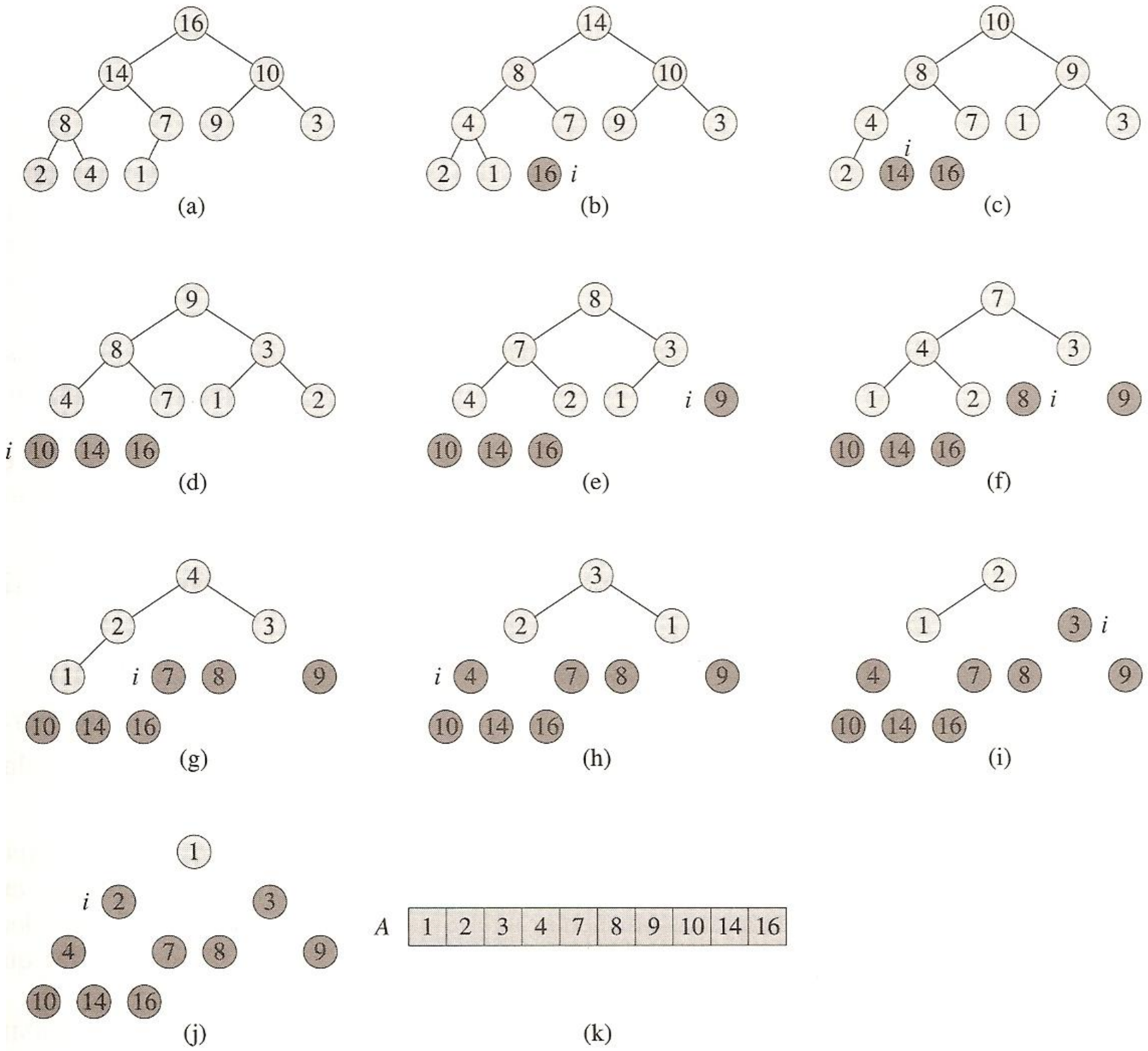

Figure 6.4 The operation of HEAPSORT. **(a)** The max-heap data structure just after it has been built by BUILD-MAX-HEAP. **(b)–(j)** The max-heap just after each call of MAX-HEAPIFY in line 5. The value of i at that time is shown. Only lightly shaded nodes remain in the heap. **(k)** The resulting sorted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

The heap sort procedure takes time O(n lg n), since the call to BUILD-MAX-HEAP takes O(n) and each the n-1 calls to MAX-HEAPIFY takes time O(lg n) (合起來就是 O(n lg n)).

6.5 Priority queues

One of the popular applications of a heap: priority queues (max-priority queue and min-priority queue). 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:

INSERT(S, x) inserts the element x into the set S.

MAXIMUM(S) returns the element of S with the largest key

EXTRACT-MAX(S) removes and returns the element of S with the largest key

INCREASE-KEY(S, *x*, *k*) increases the value of element *x*'s key to the new value *k*.

One application of max-priority queues is to schedule jobs (job-schedule in OS) on a shared computer. When a heap is used to implement a priority queue, therefore, we often need to store a *handle* (類似 pointer) to the corresponding application object in each heap element.

The procedure HEAP-EXTRACT-MAX implements the EXTRACT-MAX operation:

```
HEAP-EXTRACT-MAX(A)
if heap-size[A] < 1
    then error "heap underflow"
max ← A[1]                          ⇨ 將最大值 A[1] 取出
A[1] ← A[heap-size[A]]
heap-size[A] ← heap-size[A] − 1
MAX-HEAPIFY(A, 1)
return max
```

The running time is O(lg n).

The HEAP-INCREASE-KEY operation (將內容 A[i]增為 key 值 (若 A[i]比 key 值大就不增加)):

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

(步驟 5 是當 *key* 值放入 A[i]中，卻比其原先 parent(i)值(A[parent(i)])大，就互換，以符合 heap-tree)，see figure 6.5

The running time is O(lg n).

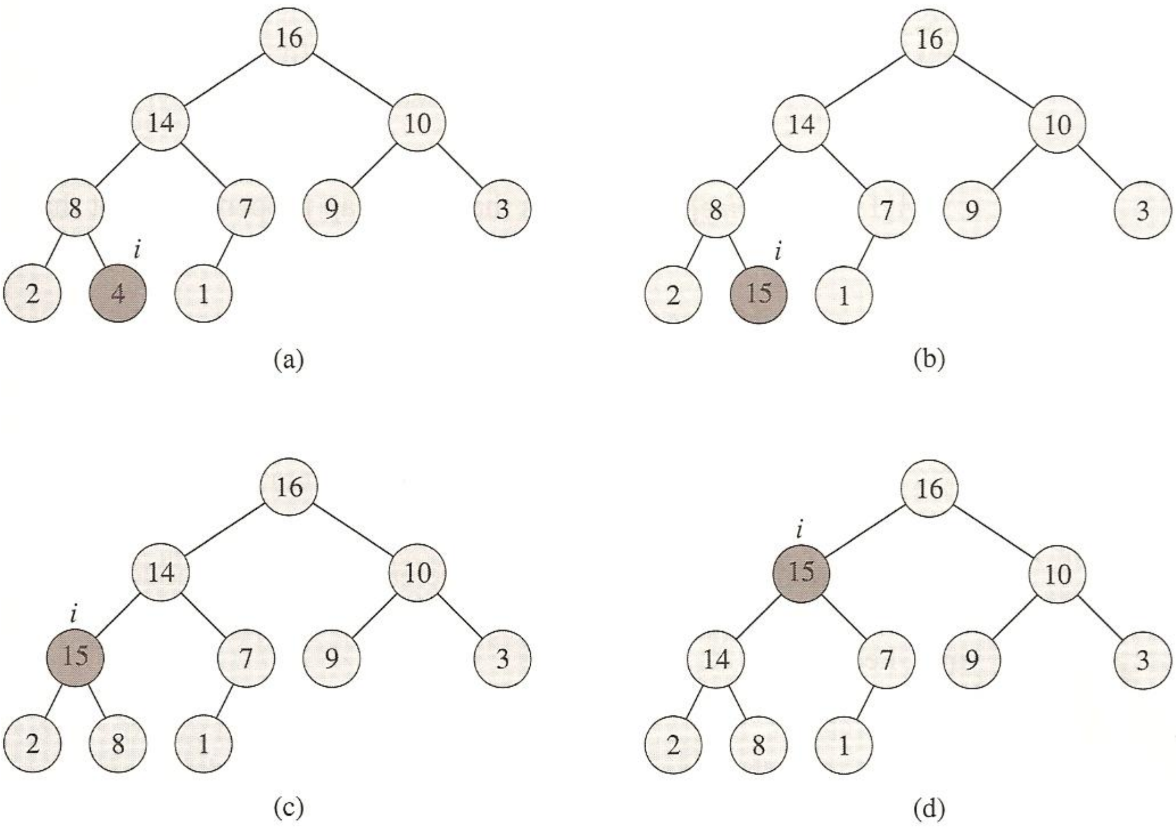

Figure 6.5 The operation of HEAP-INCREASE-KEY. **(a)** The max-heap of Figure 6.4(a) with a node whose index is i heavily shaded. **(b)** This node has its key increased to 15. **(c)** After one iteration of the **while** loop of lines 4–6, the node and its parent have exchanged keys, and the index i 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 MAX-HEAP-INSERT (插入一個新值) operation:

MAX-HEAP-INSERT(A, *key*)
1 $heap\text{-}size[A] \leftarrow heap\text{-}size[A] + 1$
2 $A[heap\text{-}size[A]] \leftarrow -\infty$
3 HEAP-INCREASE-KEY(A, $heap\text{-}size[A]$, *key*)

Insert an element with value *key* into heap, and put the A[n+1] 位置 (the last element + 1)，如同將最後一個值 **A[n+1]**改爲 **key** 值，再重新排成 **heap.**

The running time is O(lg n).

EXERCISE: Coding "Heapsort (*A*)" (上面 副程式 方式)
Data: 1, 8, 4, 9, 7, 21, 33, 32, 6, 5, 55, 22, 17, 26, 36, 24, 13, 11

(下週 報告: 1. 演算法 與 Source code ; 2. 執行過程(要印出)； 3.結果)