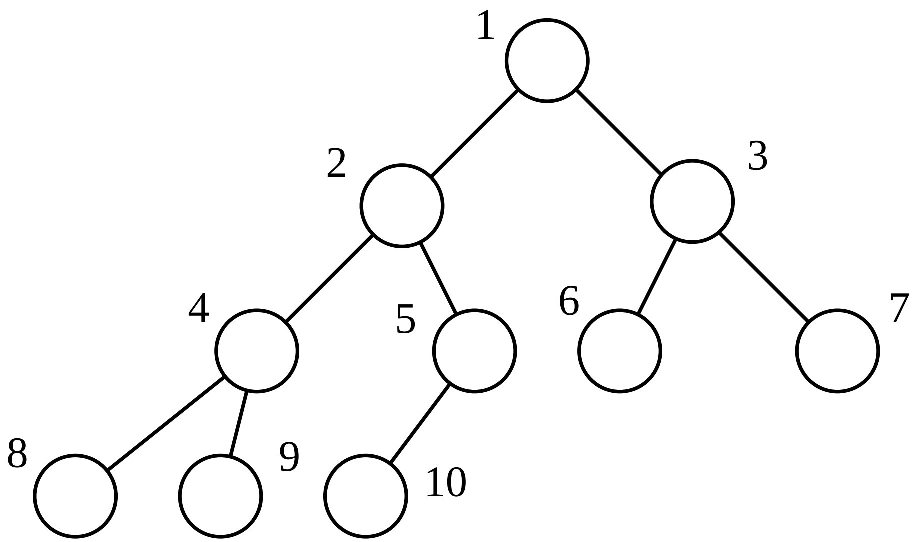
6. Heapsort and Priority Queues

Heapsort is an in place sorting algorithm that runs in $O(n \log n)$.

- It achieves asymptotically optimal running time
- It only takes a constant amount of space outside the input array
- The heapsort algorithm uses the **heap data structure**, which can also be used for **minimum/maximum priority queues**
- It is an optimal sorting algorithm

6.1 Heaps

A binary heap is a nearly complete binary tree **without child-parent pointers**, usually stored as an array. Each node (array element) contains a value (key) and perhaps data associated with the key. The tree is completely filled on **all levels** except possibly the lowest (bottom) level, which is filled from the left. Note that the tree root has the highest level number while leaves have the lowest level numbers.

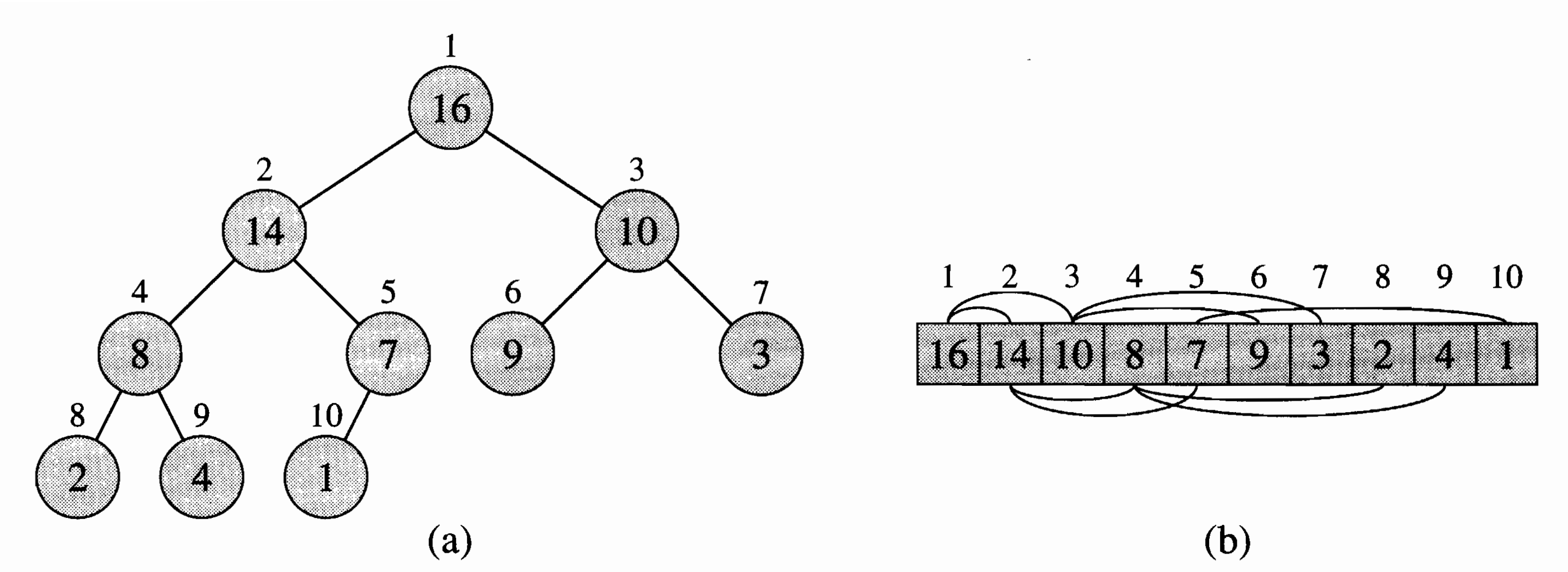


Array **indexes** are assigned top to bottom, left to right, as shown. Notice that the index of each element is used to derive its parent index and its children indices.

6.1 Heaps (continued)

There are two types of heaps: the max-heap and the min-heap.

- For a **max-heap**, every node other than the root has a key less than or equal to the key in its parent.
- For a **min-heap**, every node other than the root has a key greater than or equal to the key in its parent.



A max-heap (Cormen et al., p128)

6.1 Heaps (continued)

The tree structure of each binary heap does not need pointers, since the relatives of a node are stored in array positions that can be calculated:

- The root of the tree is $A[1]$.
- Given the index i of a node, the indices of its relatives (the parent, the left child and right child if they exist) are as follows.
  - $\mathrm{PARENT}(i) = \lfloor i/2 \rfloor$ except for $i = 1$.
  - $\mathrm{LEFT}(i) = 2i$ if the left child exists (or $2i \leq n$).
  - $\mathrm{RIGHT}(i) = 2i + 1$ if the right child exists (or $(2i + 1) \leq n$.

Basic properties of a max-heap:

- The key in the root is the largest key.
- Given any node in a max-heap, the node and all its descendants form a max-heap, (i.e., the subtree rooted at the node is a maximum heap, too).

6.1 Heaps (continued)

Let A be an array and i the index of an element in it. To transform A into a maximum (or minimum) heap, we introduce the following operations.

- MAX-HEAPIFY(A,i) – Make the subtree rooted at i into a max-heap, assuming that the subtrees rooted at LEFT(i) and RIGHT(i) are max-heaps.
- BUILD-MAX-HEAP(A) – Make the unordered array A into a max-heap.
- HEAP-MAXIMUM(A) – Return the largest key in a max-heap A.
- MAX-HEAP-INSERT(A,key) – Insert a new key into a max-heap A.
- HEAP-EXTRACT-MAX(A) – Delete the largest key from a max-heap A.
- HEAP-INCREASE-KEY(A,i,key) – Increase the key of element i in a max-heap A.

6.2 Maintaining the heap property

When an element of a max-heap **violates the max-heap property**, the error can be fixed by repeatedly using one of the following operations.

- *Swap up.* If a key is greater than the key of its parent, swap them.
- *Swap down.* If a key is less than the key of one of its two children (or a child), swap the key with the larger key of the two children.

6.2 MAX-HEAPIFY operation

MAX-HEAPIFY(A, i) – Make the subtree rooted at i into a max-heap, assuming that the subtrees rooted at LEFT(i) and RIGHT(i) are max-heaps already.

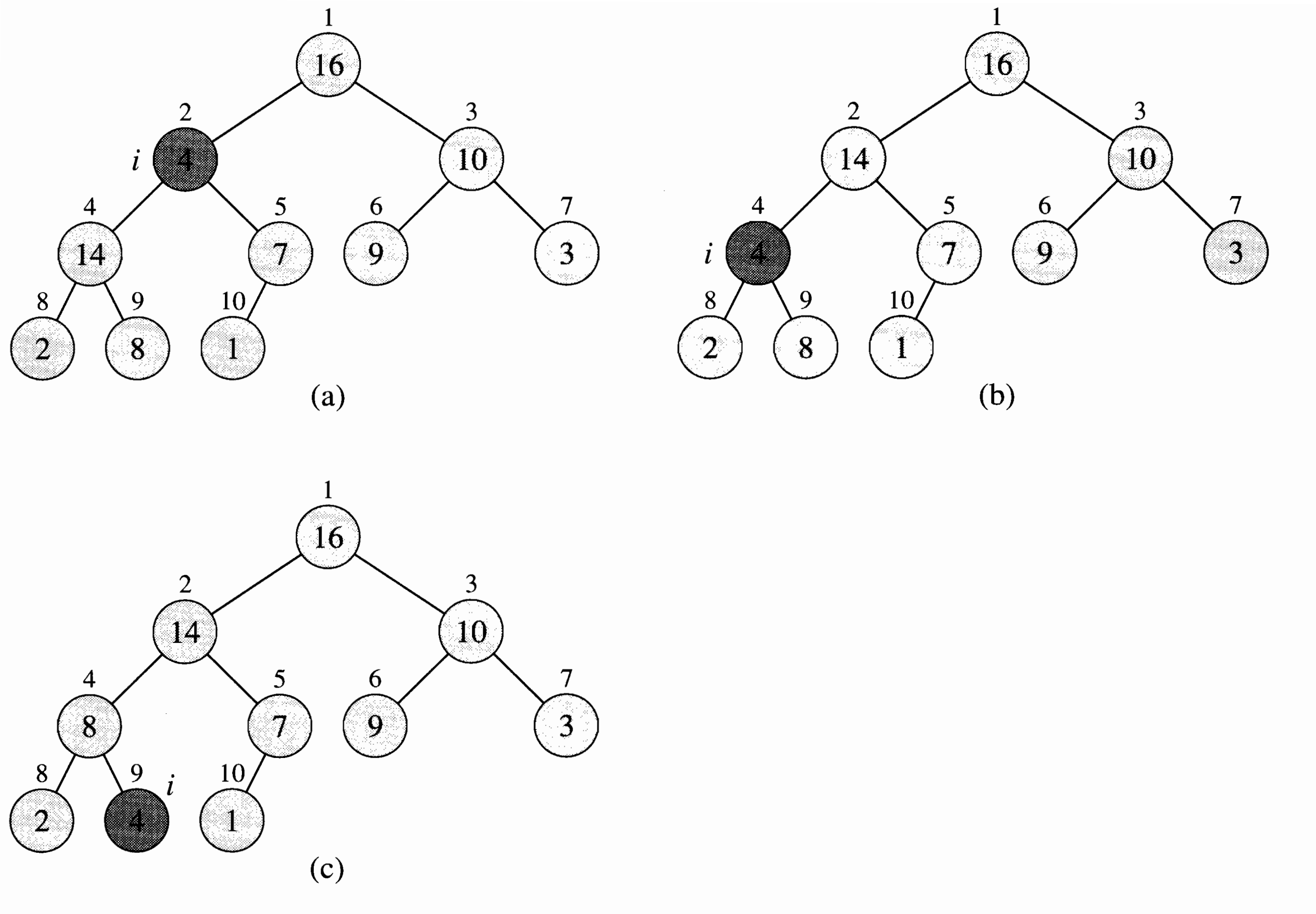
Method: swap down as necessary.

Algorithm MAX-HEAPIFY(A, i)

$l \leftarrow LEFT(i)$;
$r \leftarrow RIGHT(i)$;
IF $(l \leq heap\_size[A])$ & $(A[l] > A[i])$
    THEN $largest \leftarrow l$
    ELSE $largest \leftarrow i$;
IF $(r \leq heap\_size[A])$ & $(A[r] > A[largest])$
    THEN $largest \leftarrow r$;
IF $largest \neq i$
    THEN exchange $A[i] \leftrightarrow A[largest]$;
    MAX-HEAPIFY$(A, largest)$;

6.2 MAX-HEAPIFY operation (example)

The invisible value is 4 in the following figures, which violates the max-heap property.



6.2 MAX-HEAPIFY operation (running time)

Time required: $O(\log n)$. **Why?**

The *height* of an element is the distance to the *bottom level*.

For an element on level ℓ, the time required by MAX-HEAPIFY is $O(\ell)$.

6.3 BUILD-MAX-HEAP operation

Given an unordered array A, turn it into a max-heap.

The idea is to fix one element at a time, working from the leaves up to the root. As each element's turn comes, its children are already fixed.

Entries indexed by $\lceil length[A]/2 \rceil$ and later are leaves, so they are already fixed.

BUILD-MAX-HEAP(A)

1 $heap\_size[A] \leftarrow length[A]$
2 for $i \leftarrow \lfloor length[A]/2 \rfloor$ downto 1 do
3 MAX-HEAPIFY(A, i)

Time required: $O(n)$. **Why?**

6.3 BUILD-MAX-HEAP operation

Let the bottom level of the max-heap be 0. Then, the level of the root of the max-heap is $\lfloor \log n \rfloor$.

Theorem: It takes $O(n)$ for `BUILD-MAX-HEAP` operation.

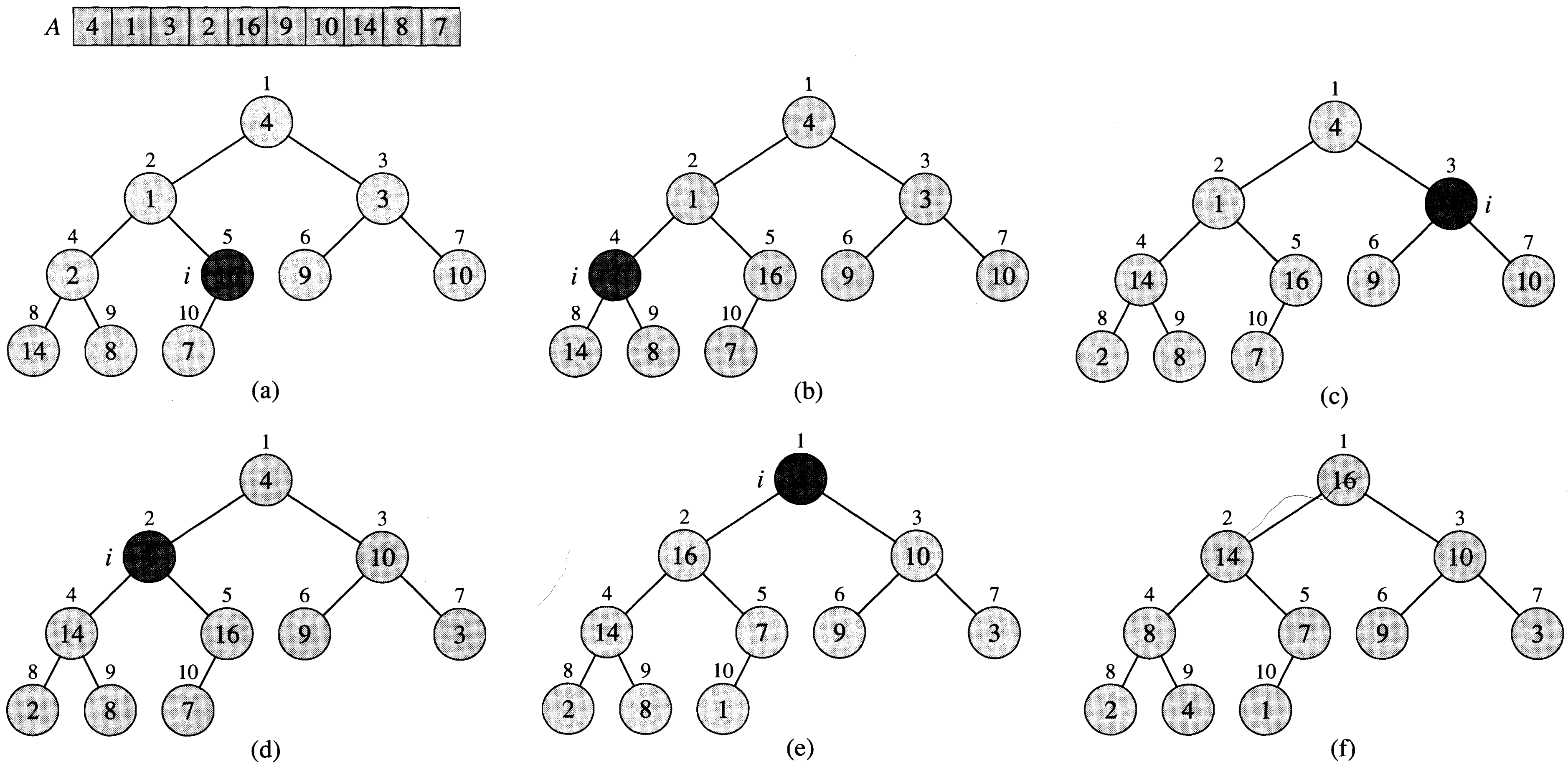
Proof

$$\sum_{h=0}^{\lfloor \log n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h) = O(n \sum_{h=0}^{\lfloor \log n \rfloor} \lceil \frac{h}{2^{h+1}} \rceil) \quad \text{since there are } \lceil \frac{n}{2^{h+1}} \rceil \text{ nodes at level } h \text{ with } 0 \leq h$$

we then have

$$\begin{aligned}
\sum_{h=0}^{\lfloor \log n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h) &= O(n \sum_{h=0}^{\lfloor \log n \rfloor} \lceil \frac{h}{2^{h+1}} \rceil) = O(n \sum_{h=0}^{\infty} \lceil \frac{h}{2^{h+1}} \rceil) \\
&= O(n \cdot 2), \quad \text{since } \textstyle\sum_{h=0}^{\infty} \frac{h}{2^h} = 2 \\
&= O(n). \qquad (2)
\end{aligned}$$

6.3 BUILD-MAX-HEAP operation (example)



Cormen et al., p158, where $n = 10$ elements.

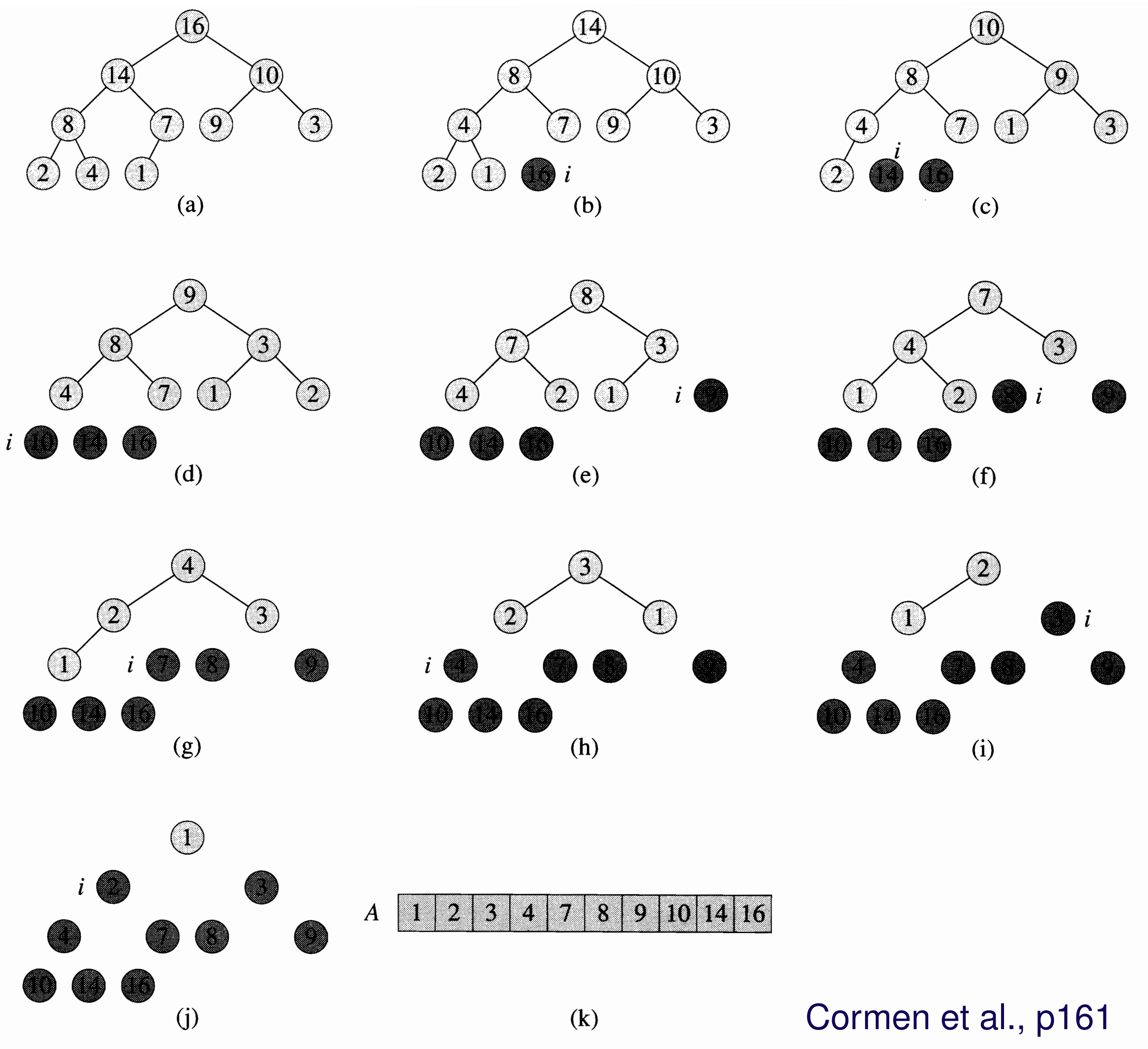
6.4 Heapsort

Heapsort is an algorithm for sorting an array.

Method: Turn the array into a max-heap. Then repeatedly remove the maximum element of the heap into the proper place in the array.

```
Algorithm HEAPSORT(A)
    BUILD-MAX-HEAP(A)
    for  i ← length[A] downto 2 do
        A[1] ⇔ A[i] /* exchange the keys in these two cells */
        heap_size[A] ← heap_size[A] − 1
        MAX-HEAPIFY(A, 1)
```

Time required: $O(n \log n)$. **Why?**



Cormen et al., p161

6.5 Priority Queues

A priority queue is a data structure for maintaining a set S of elements, each with an associated value called the **key**. There are two types of priority queues: the max-priority queue and the min-priority queue.

A max-priority queue supports the following operations.

- INSERT(S, x) inserts the element x into the set S, i.e., $S \leftarrow S \cup \{x\}$.
- 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, which is assumed to be **at least as large as** x's current key value.

6.5 Operations in a max-priority queue

We can implement a max-priority queue S, using a max-heap A.

HEAP-MAXIMUM(A)

  return $A[1]$

Time required: $O(1)$.

HEAP-EXTRACT-MAX(A)

  if $heap\_size[A] < 1$
    then error: “heap underflow”
  $max \leftarrow A[1]$
  $A[1] \leftarrow A[heap\_size[A]]$
  $heap\_size[A] \leftarrow heap\_size[A] - 1$
  MAX-HEAPIFY($A, 1$)
  return max

Time required: $O(\log n)$. **Why?**

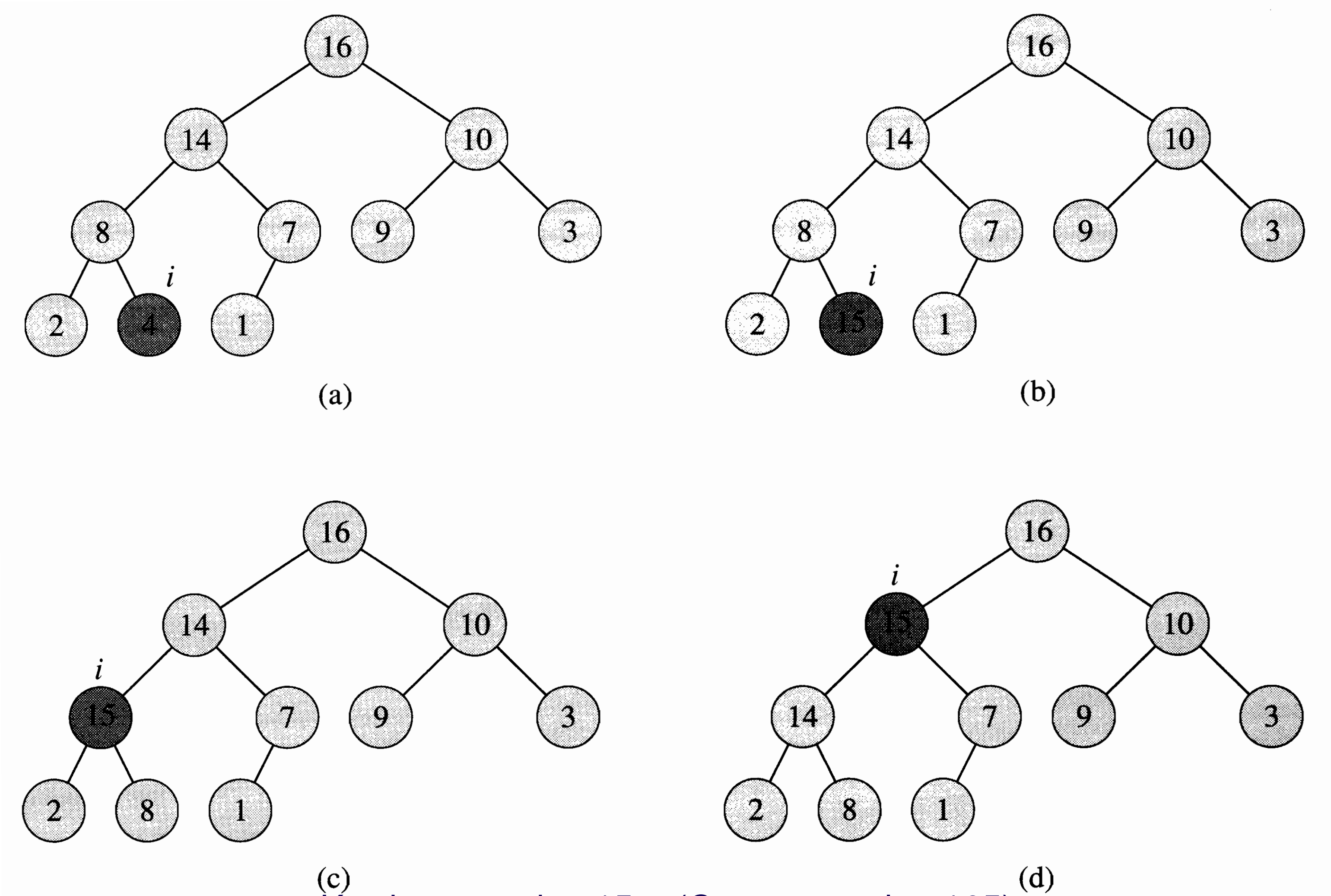
6.5 Operations in a max-priority queue (continued)

If increasing a key causes the max-heap property to be violated (since the new key exceeds the key of the parent), then swap up until the error is fixed.

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

Time required: $O(\log n)$. **Why?**

6.5 HEAP-INCREASE-KEY (example)



Key increased to 15. (Cormen et al., p165)

6.5 Operations in a max-priority queue (continued)

To add a new element, just put it in the next array position and fix the heap if necessary.

MAX-HEAP-INSERT(A, key)

1. $heap\_size \leftarrow heap\_size[A] + 1$
2. $A[heap\_size] \leftarrow -\infty$
3. HEAP-INCREASE-KEY$(A, heap_size[A], key)$

In other words, We make use of two existing operations to implement a new element insertion into a maximum priority queue. That is, insert the element with a minimum key ($-\infty$), then increase the key value to its actual value.