6.Heapsort

Yu-Shuen Wang, CS, NCTU

Why sorting

1. Sometimes the need to sort information is inherent in an application.
2. Algorithms often use sorting as a key subroutine.
3. There is a wide variety of sorting algorithms, and they use rich set of techniques.
4. Sorting problem has a nontrivial lower bound
5. Many engineering issues come to fore when implementing sorting algorithms.

Sorting algorithm

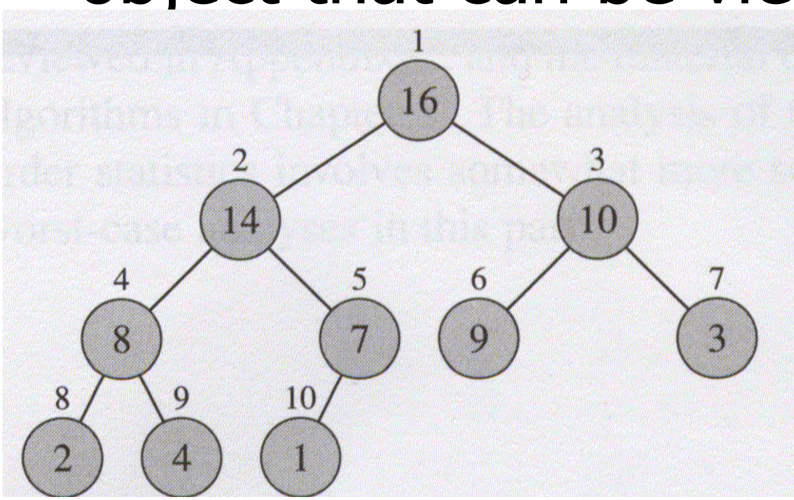
- Insertion sort :
  - In place: only a constant number of elements of the input array are even sorted outside the array.
- Merge sort :
  - not in place.
- Heap sort : (Chapter 6)
  - Sorts n numbers in place in $O(n \lg n)$

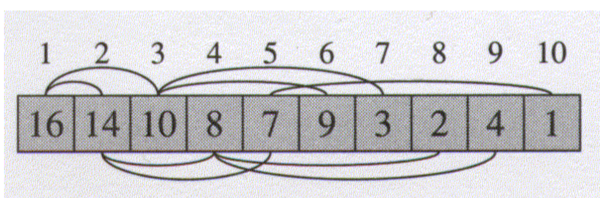
Sorting algorithm

- Quick sort : (chapter 7)
  - worst time complexity $O(n^2)$
  - Average time complexity $O(n \log n)$
- Decision tree model : (chapter 8)
  - Lower bound $O(n \log n)$
  - Counting sort
  - Radix sort
- Order statistics

6.1 Heaps (Binary heap)

- The ***binary heap*** data structure is an array object that can be viewed as a complete tree.





Parent(i)

return $\lfloor i/2 \rfloor$

LEFT(i)

return $2i$

Right(i)

return $2i+1$

Heap property

- Max-heap : A [parent(i)] $\geq$ A[i]
- Min-heap : A [parent(i)] $\leq$ A[i]
- The ***height of a node*** in a tree: the number of edges on the longest simple downward path from the node to a leaf.
- The ***height of a tree***: the height of the root
- ***The height of a heap***: O(log n).

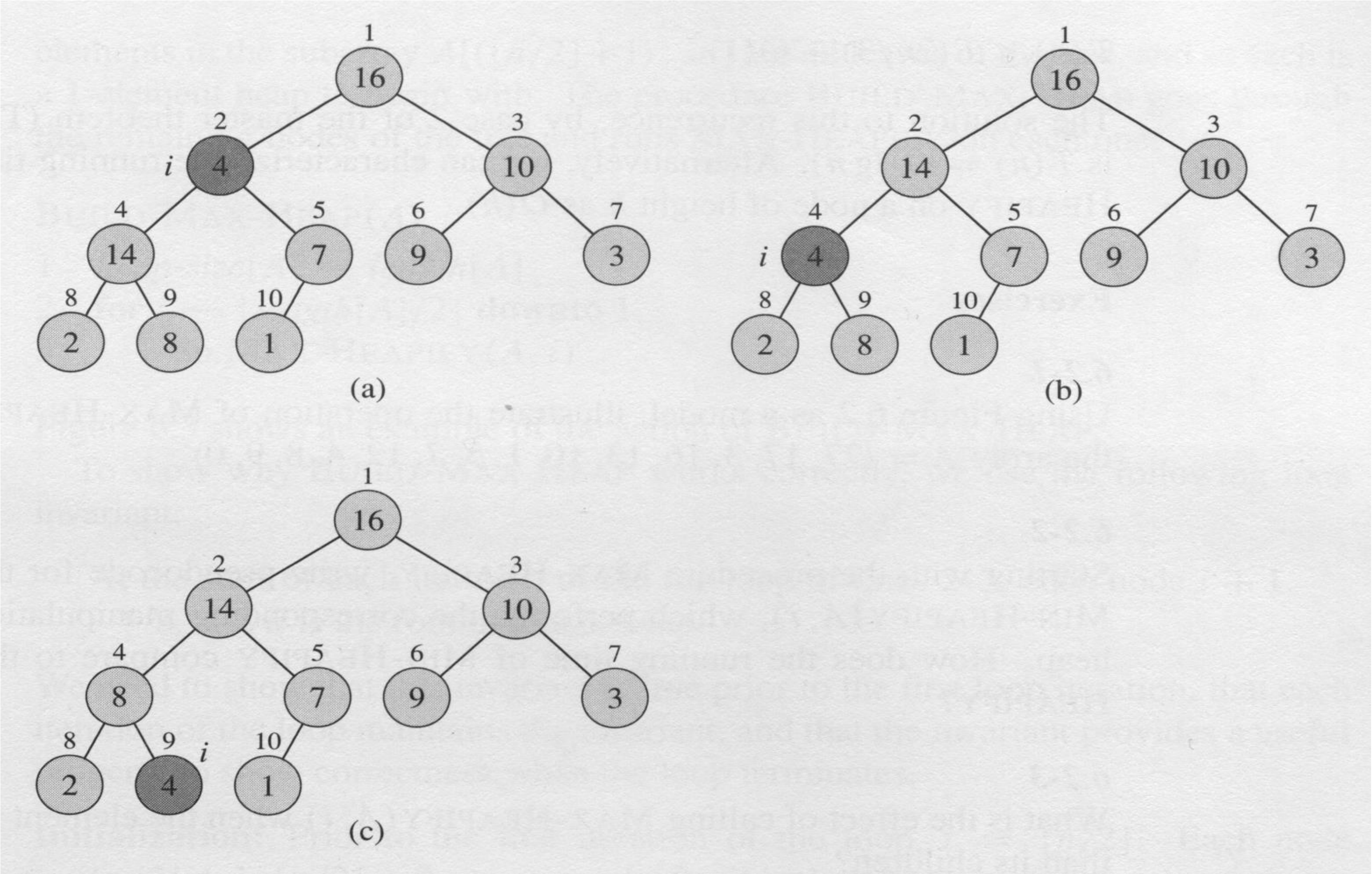
Basic procedures on heap

- Max-Heapify procedure
- Build-Max-Heap procedure
- Heapsort procedure
- Max-Heap-Insert procedure
- Heap-Extract-Max procedure
- Heap-Increase-Key procedure
- Heap-Maximum procedure

6.2 Maintaining the heap property

- Heapify is an important subroutine for manipulating heaps. Its inputs are an array A and an index i in the array. When Heapify is called, it is assume that the binary trees rooted at LEFT(i) and RIGHT(i) are heaps, but that $A[i]$ may be smaller than its children, thus violating the heap property.

Max-Heapify(A,2) heap-size[A] = 10

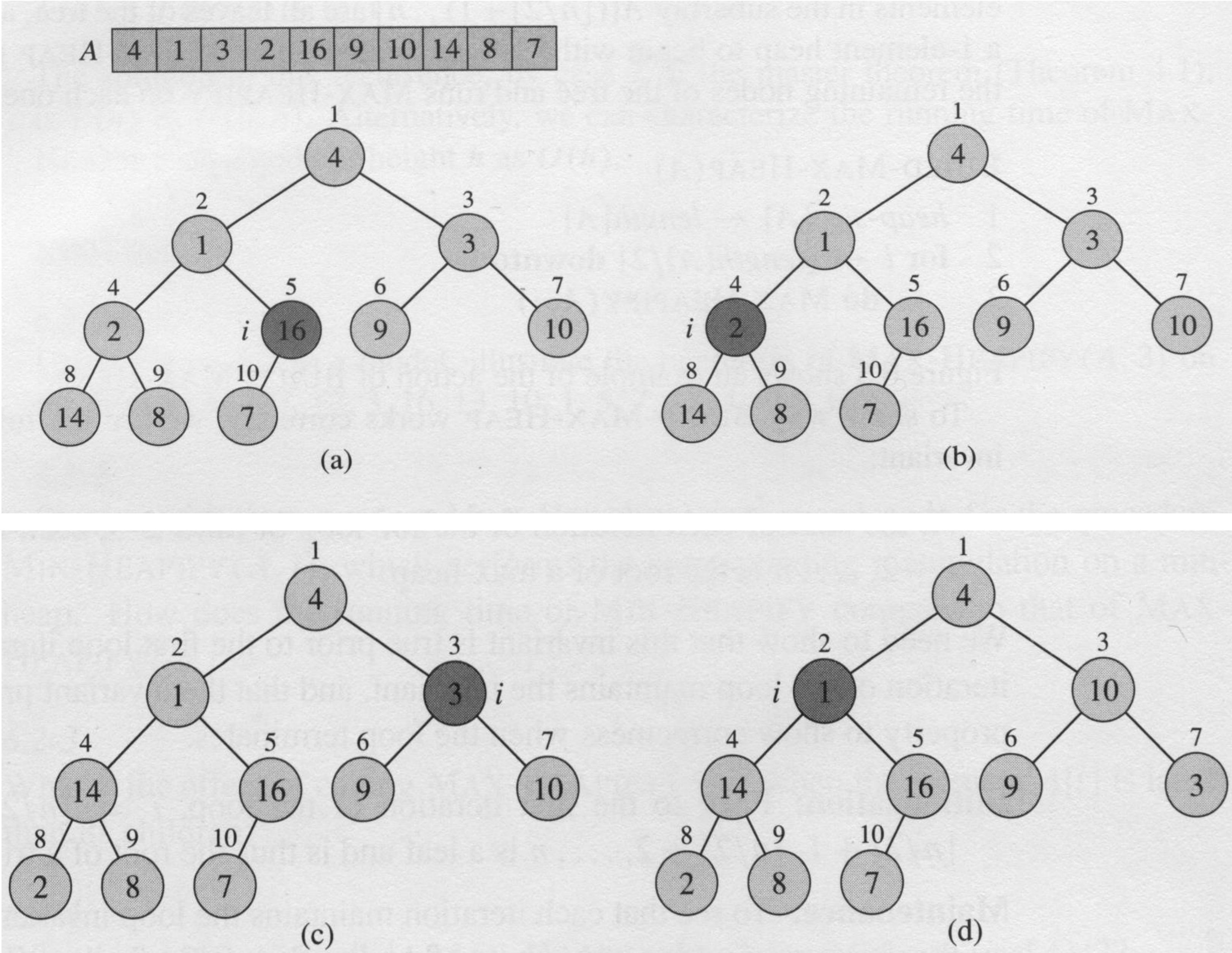


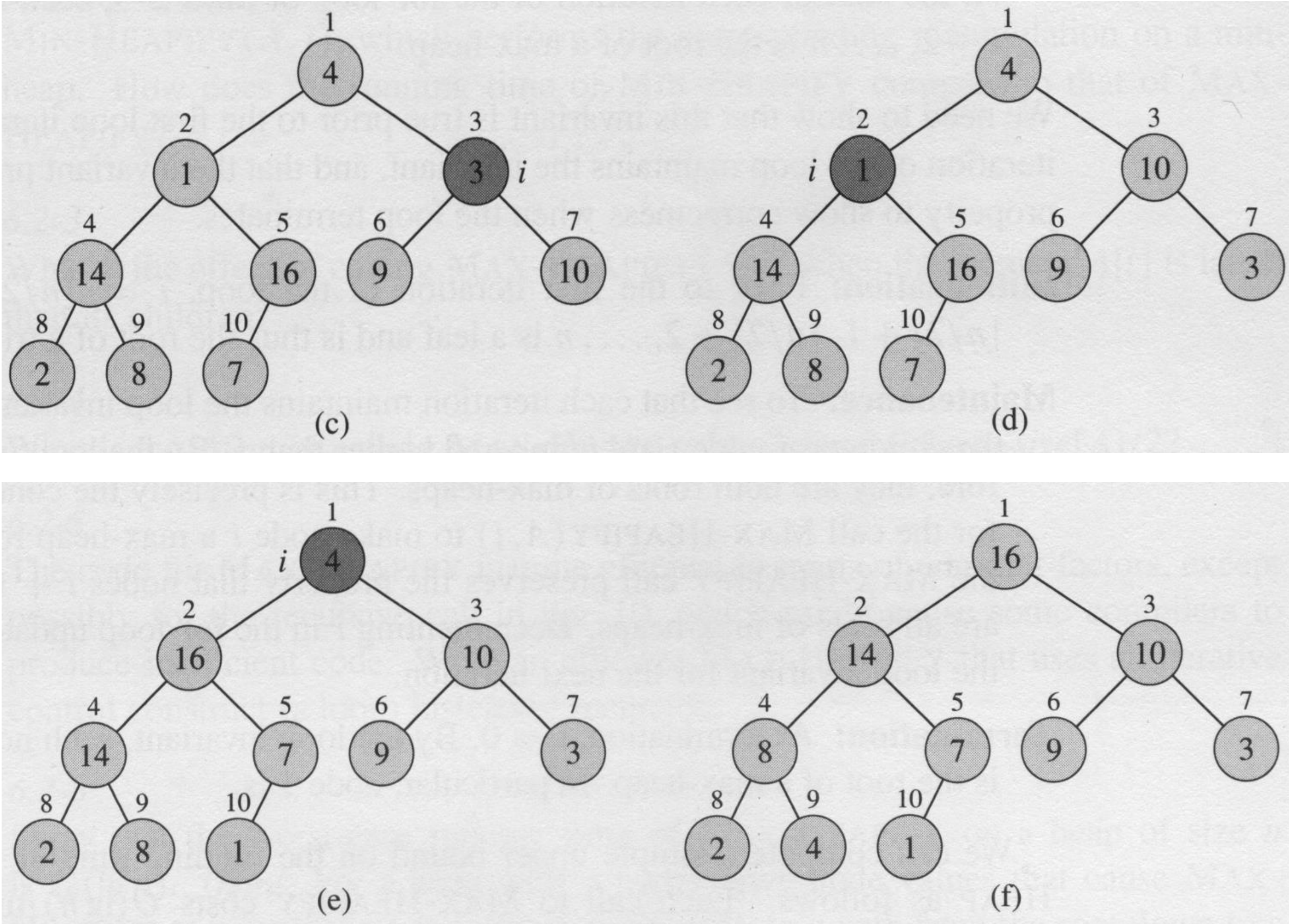
```
Max-Heapify (A, i)
l → Left (i)
r → Right(i)
if l ≤ heap-size[A] and A[l] > A[i]
    then largest ← l
    else largest ← i
if r ≤ heap-size[A] and A[r] > A[largest]
    then largest ← r
if largest ≠ i
    then exchange A[i] ↔ A[largest]
            Max-Heapify (A, largest)
```

Alternatively O(h)(h: height)

6.3 ***Building a heap***

```
Build-Max-Heap(A)
heap-size[A] ← length[A]
for i ← ⌊length[A]/2⌋ downto 1
    do Max-Heapify(A, i)
```


A
4 1 3 2 16 9 10 14 8 7
(a)
(b)
(c)
(d)
i



(c) (d) (e) (f)

Complexity

- Upper bound: O(n log n)
- A tighter bound:

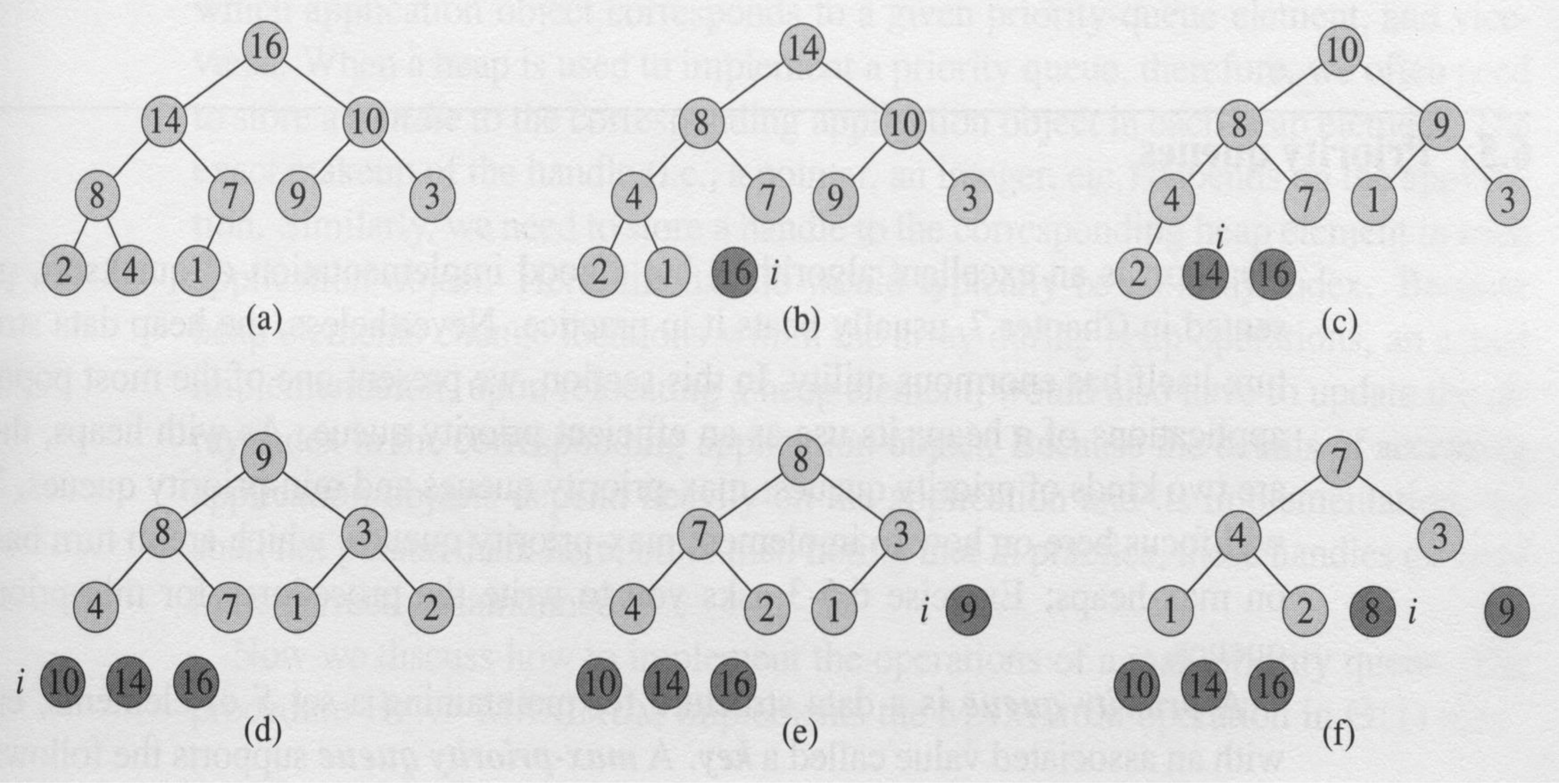
$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{h}{2^h} \right\rceil \right) = O(n)$$

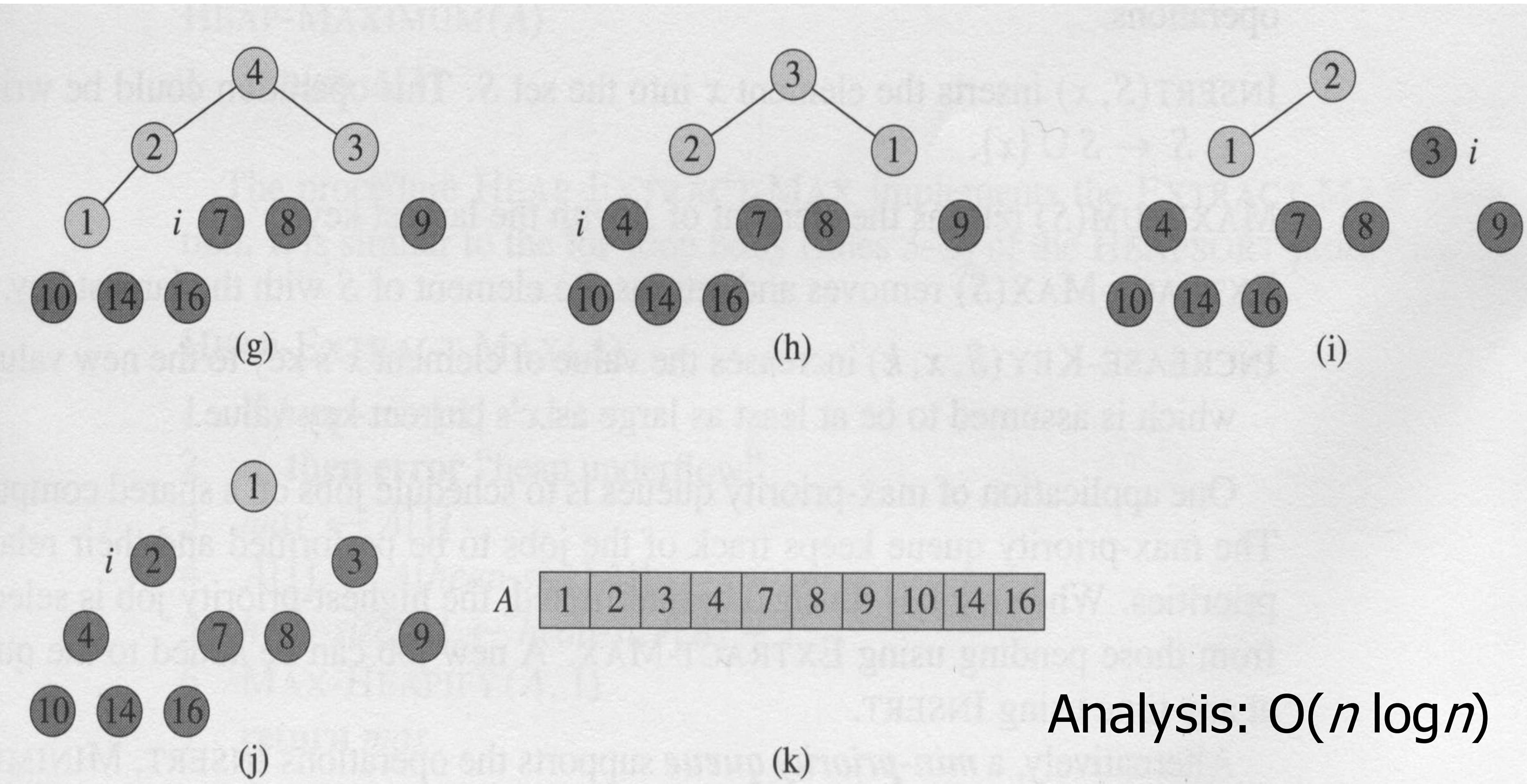
$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \Rightarrow \sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2} \Rightarrow \sum_{h=0}^{\infty} \frac{h}{2^h} = \sum_{h=0}^{\infty} \frac{1/2}{(1/2)^2} = 2$$

6.4 The Heapsort algorithm

```
Heapsort(A)
Build-Max-Heap(A)
for i ← length[A] down to 2
    do exchange A[1]↔A[i]
        heap-size[A] ← heap-size[A] -1
        Max-Heapify(A.1)
```

The operation of Heapsort





Analysis: $O(n \log n)$

7.5 Priority queues

A **priority queue** is a data structure that maintain a set S of elements, each with an associated value call a **key**. A **max-priority queue** support the following operations:

- **Insert** (S, x) — $O(\log n)$
- **Maximum** (S) — $O(1)$
- **Extract-Max** (S) — $O(\log n)$
- **Increase-Key** (S, x, k) — $O(\log n)$

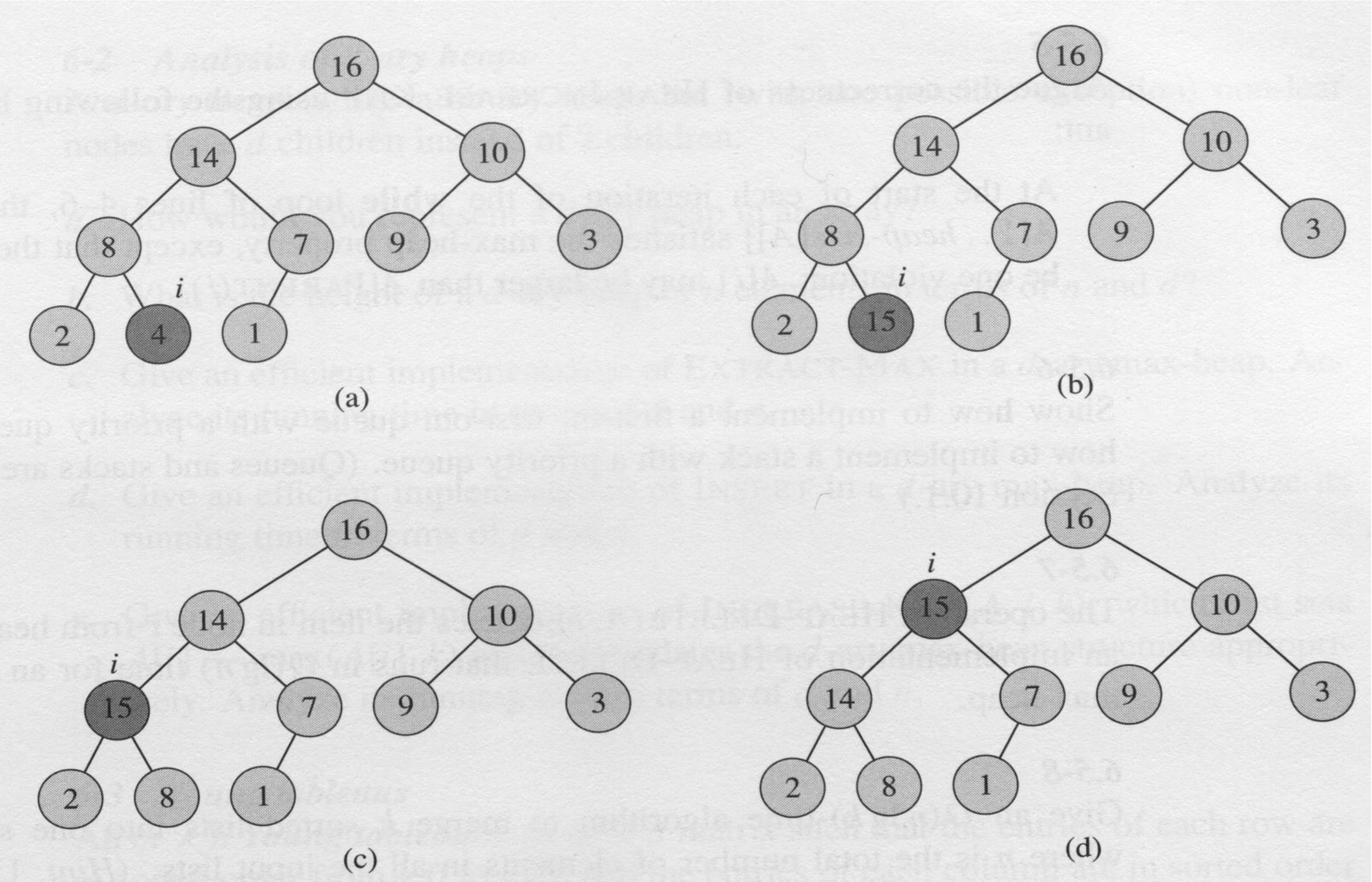
Heap_Extract-Max(A)

```
if heap-size[A] < 1
   then error "heap underflow"
max ← A[1]
A[1] ←A[heap-size[A]]
heap-size[A] ← heap-size[A] - 1
Max-Heapify (A, 1)
return max
```

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)
```

Heap-Increase-Key



Heap_Insert(A, *key*)

1 heap-size[A] $\leftarrow$ heap-size[A] + 1

2 A[heap-size[A]] $\leftarrow -\infty$

3 Heap-Increase-Key (A, heap-size[A], key)