

# *Heap Sort*

# *Why sorting*

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

# *Heapsort*

☞ Heapsort combines the better attributes of sorting algorithms:

- Like merge sort, it's running time is  $O(n \lg n)$ .
- Like insertion sort, it sorts in place.

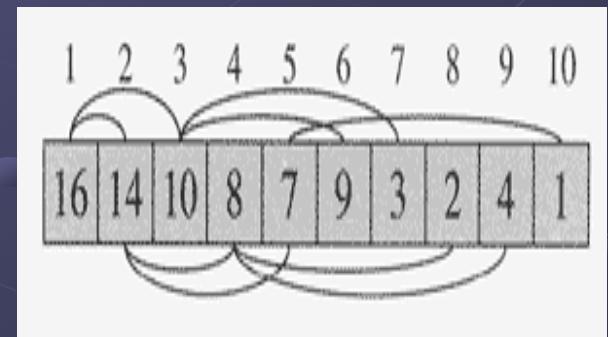
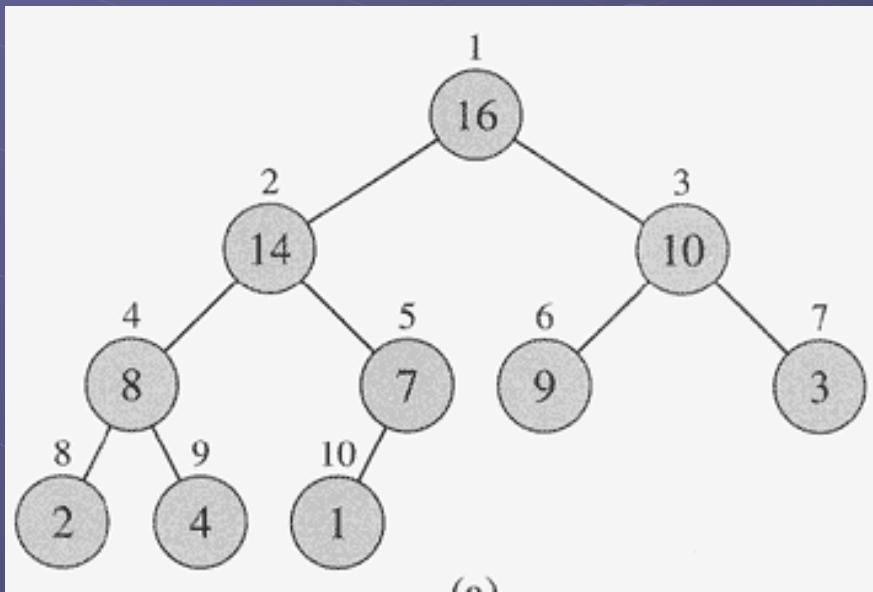
☞ Heapsort introduce another algorithm design technique:

- The use of a data structure to manage information during the execution of the algorithm.

# Heaps<sup>1</sup>

## Heap:

- An array object that can be viewed as a nearly complete binary tree.



# Heaps<sup>2</sup>

## ☞ Max-heap (Commonly used in heapsort)

- The *max-heap property* is that for every node  $i$  other than the root,

$$A[\text{PARENT}(i)] \geq A[i]$$

- The largest element in a max-heap is stored at the root.

## ☞ Min-heap

- The *min-heap property* is that for every node  $i$  other than the root,

$$A[\text{PARENT}(i)] \leq A[i]$$

- The smallest element in a min-heap is stored at the root.
- Commonly used in priority queues.

# Heaps<sup>3</sup>

## ☞ Basic procedures:

- MAX-HEAPIFY: Maintaining the max-heap property, runs in  $O(\lg n)$  time.
- BUILD-MAX-HEAP: Produce a max-heap from an unordered input array, runs in linear time.
- HEAPSORT: Sorts an array in place, runs in  $O(n \lg n)$  time.
- MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE\_KEY, HEAP-MAXIMUM: Allow the heap to be used as a priority queue, run in  $O(\lg n)$  time.

# Maintaining the Heap Property<sup>1</sup>

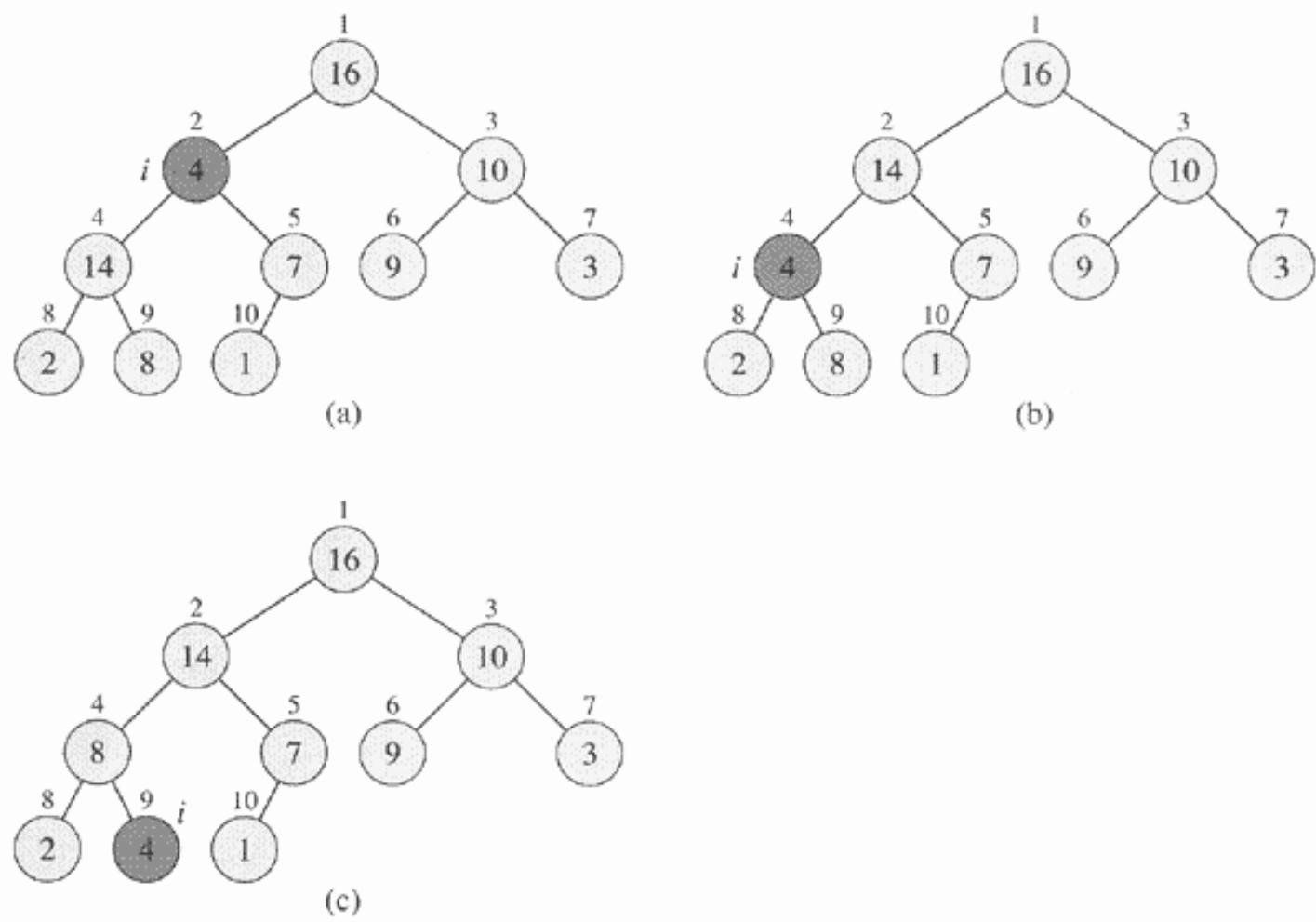
## ☞ MAX-HEAPIFY

- Let the value at  $A[i]$  “float down” in the max-heap so that the subtree rooted at index  $i$  become a max-heap.

MAX-HEAPIFY( $A, i$ )

```
1   $l = \text{LEFT}(i)$ 
2   $r = \text{RIGHT}(i)$ 
3  if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$ 
4       $\text{largest} = l$ 
5  else  $\text{largest} = i$ 
6  if  $r \leq A.\text{heap-size}$  and  $A[r] > A[\text{largest}]$ 
7       $\text{largest} = r$ 
8  if  $\text{largest} \neq i$ 
9      exchange  $A[i]$  with  $A[\text{largest}]$ 
10     MAX-HEAPIFY( $A, \text{largest}$ )
```

# Maintaining the Heap Property<sup>2</sup>



# Maintaining the Heap Property<sup>3</sup>

## ☞ The running time of MAX-HEAPIFY

- Worst case: The children's subtree each have size at most  $2n/3$  when the last row of the tree is exactly half full.

$$T(n) \leq T(2n/3) + \Theta(1) \quad (\text{Case 2 of Master theorem})$$

$$\therefore T(n) = O(\lg n)$$

# Building a Heap<sup>1</sup>

## Algorithm

```
BUILD-MAX-HEAP( $A$ )
```

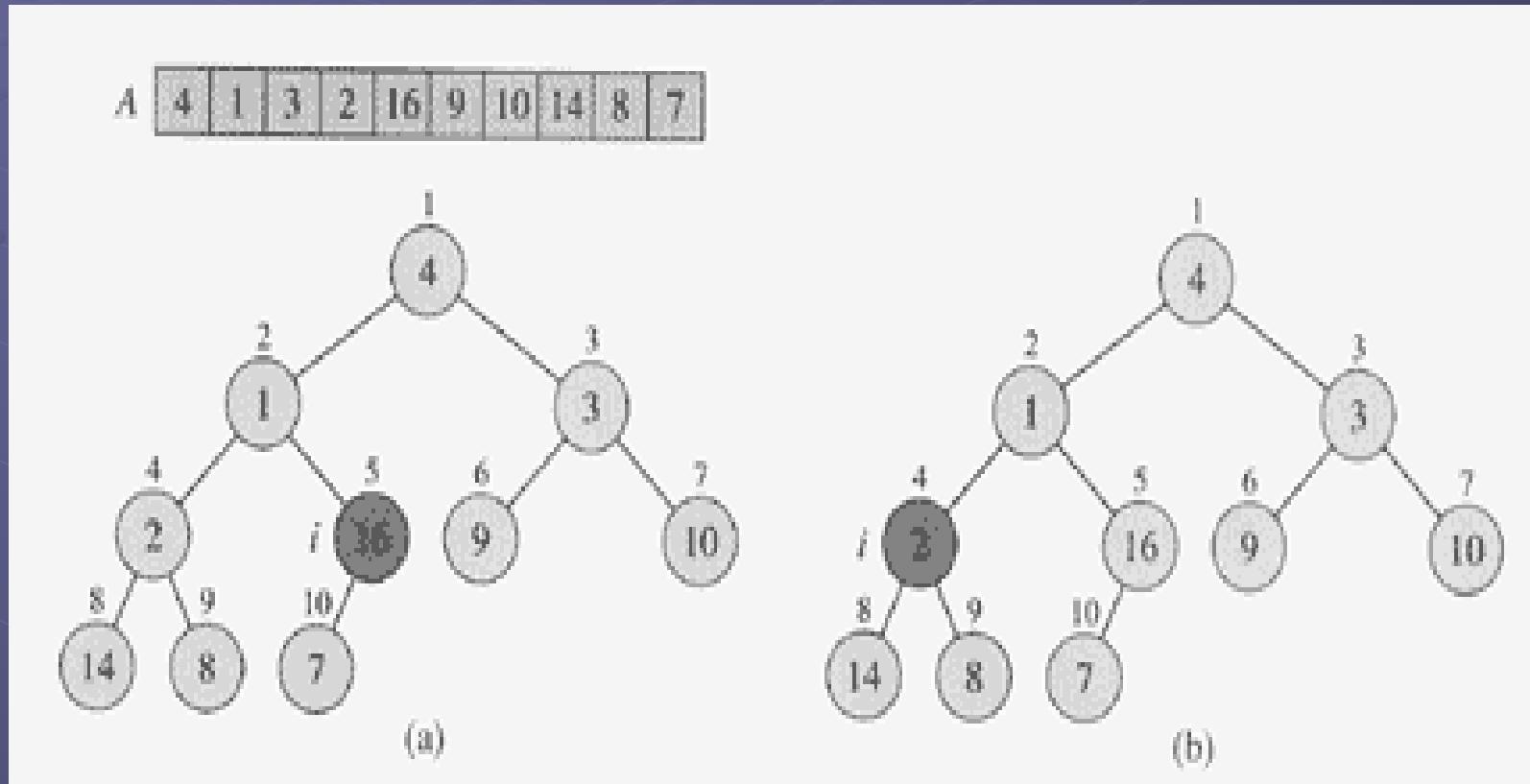
- 1  $A.\text{heap-size} = A.\text{length}$
- 2 **for**  $i = \lfloor A.\text{length}/2 \rfloor$  **downto** 1
- 3     MAX-HEAPIFY( $A, i$ )

## Correctness

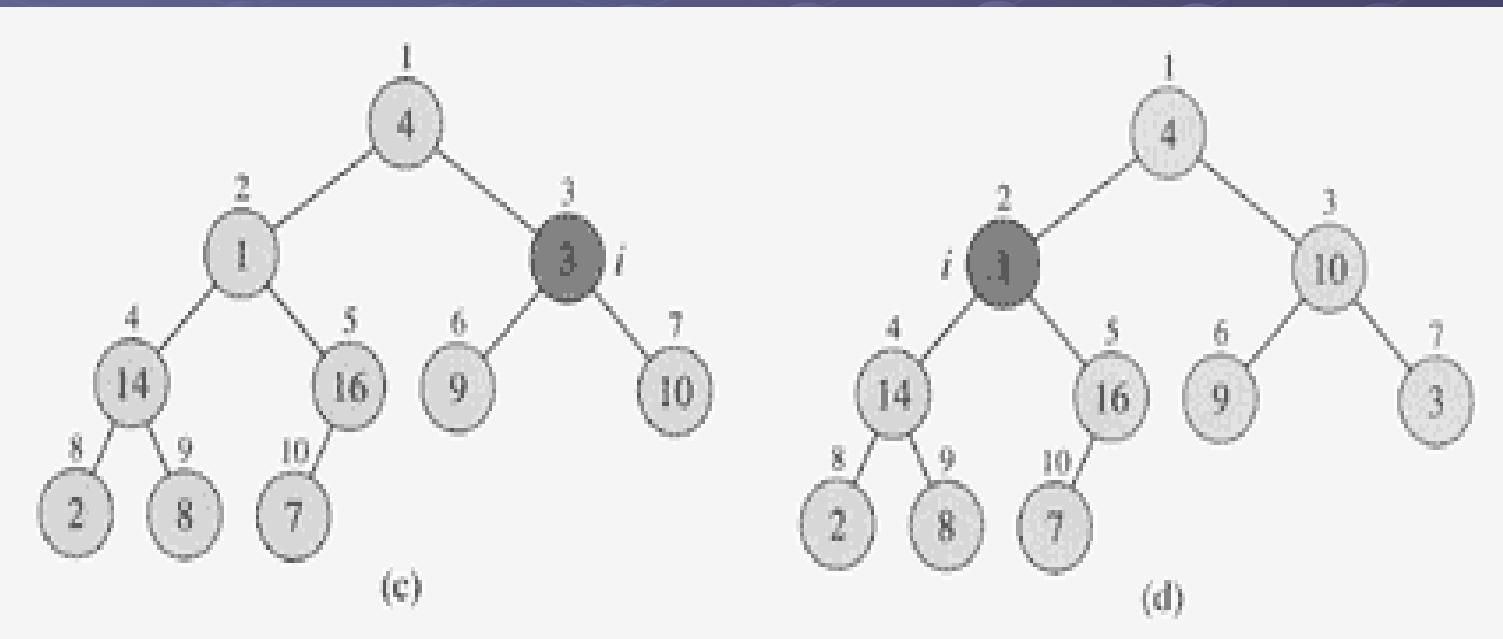
### ■ Loop invariant

At the start of each iteration of the for loop, each node  $i + 1, i + 2, \dots, n$  is the root of a max-heap.

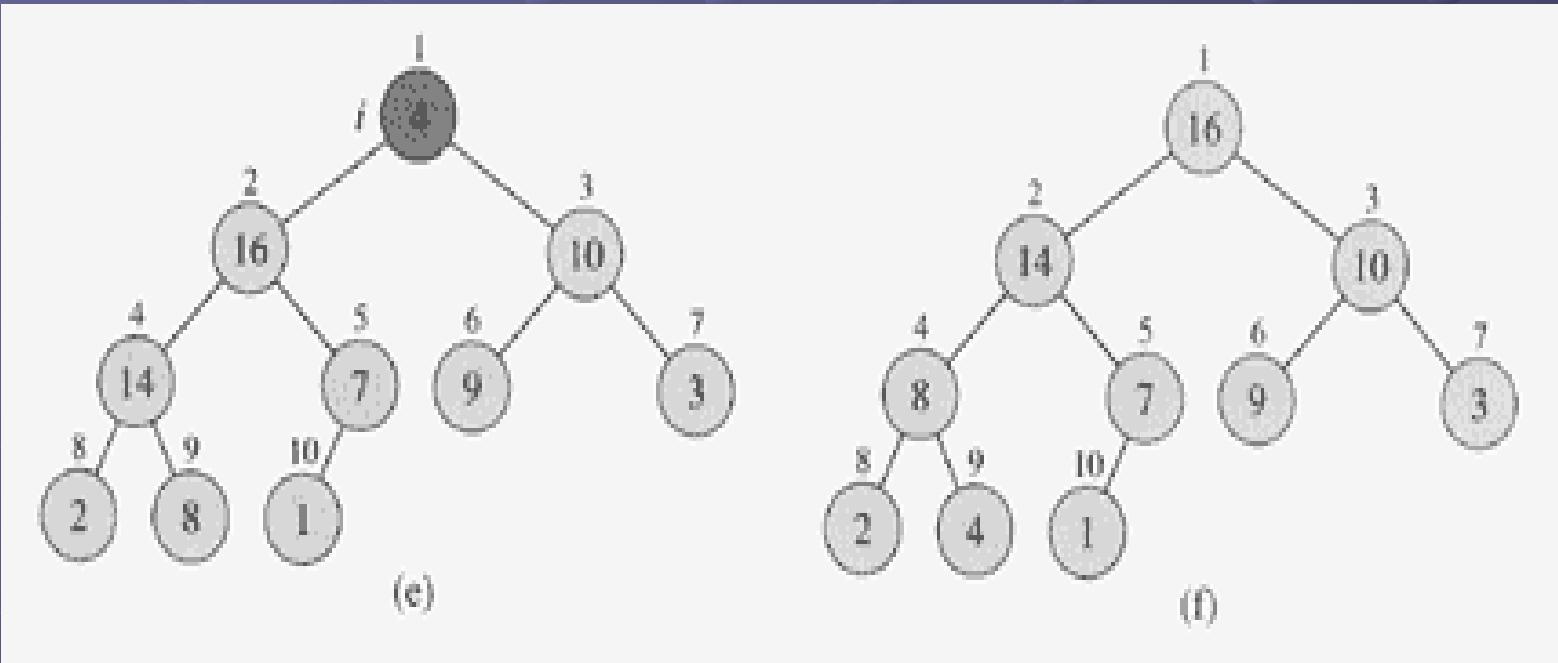
# Building a Heap<sup>2</sup>



# Building a Heap<sup>3</sup>



# *Building a Heap<sup>4</sup>*



# Building a Heap<sup>5</sup>

■ Initialization: Prior to the first iteration of the loop,  $i = \lfloor n/2 \rfloor$ .

■ Maintenance

- The children of node  $i$  are both roots of max-heaps.
- MAX-HEAPIFY( $A, i$ ) makes node  $i$  a max-heap root.

■ Termination: At termination  $i = 0$ .

☞ Running time:  $O(n \lg n)$

- Each call to MAX-HEAPIFY cost  $O(\lg n)$  time.
- There are  $O(n)$  such call.
- Not asymptotic tight.

# *Building a Heap*<sup>6</sup>

## ☞ Tighter analysis

- An  $n$ -element heap has height  $\lfloor \lg n \rfloor$  and at most  $\lceil n/2^{h+1} \rceil$  nodes of any height  $h$ .
- The time required by MAX-HEAPIFY when called on a node of height  $h$  is  $O(h)$ .

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

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = 2 \left( \Theta \sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2} \right)$$

$$O(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}) = O(n \sum_{h=0}^{\infty} \frac{h}{2^h}) = O(n)$$

# The Heapsort Algorithm<sup>I</sup>

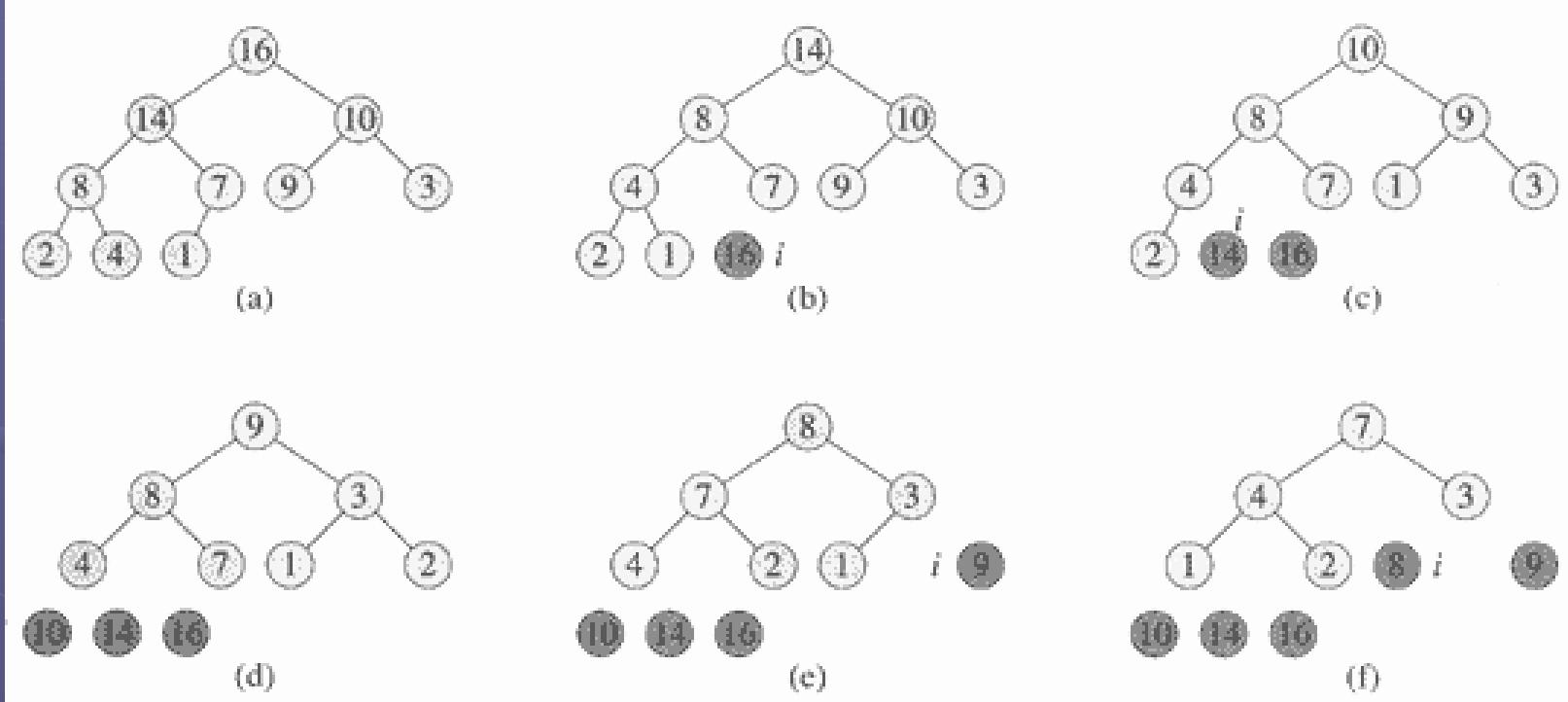
## ☞ Algorithm

```
HEAPSORT( $A$ )
```

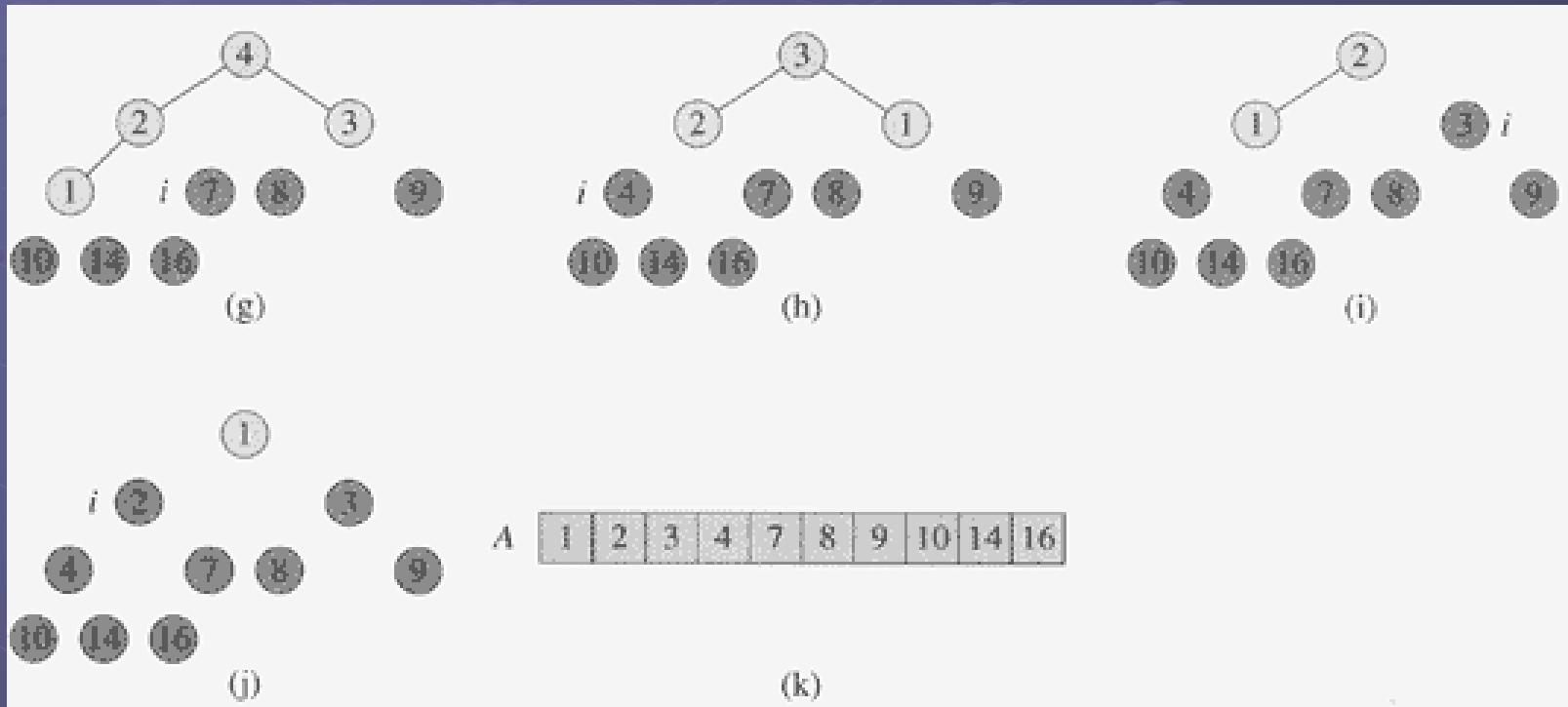
- 1 BUILD-MAX-HEAP( $A$ )
- 2 **for**  $i = A.length$  **downto** 2
- 3     exchange  $A[1]$  with  $A[i]$
- 4      $A.heap-size = A.heap-size - 1$
- 5     MAX-HEAPIFY( $A, 1$ )

☞ Running time:  $O(n \lg n)$ .

# The Heapsort Algorithm<sup>2</sup>



# The Heapsort Algorithm<sup>3</sup>



# Priority Queues<sup>1</sup>

## ☞ Priority queue:

- A data structure for maintaining a set  $S$  of elements, each with an associated value called a *key*.

## ☞ Max-priority queue

- $\text{INSERT}(S, x)$
- $\text{MAXIMUM}(S)$
- $\text{EXTRACT-MAX}(S)$
- $\text{INCREASE-KEY}(S, x, k)$

## ☞ Min-priority queue

# Priority Queues<sup>2</sup>

☞ **HEAP-MAXIMUM(A)**       $\Theta(1)$

1 **return**  $A[1]$

☞ **HEAP-EXTRACT-MAX(A)**       $O(\lg n)$

```
1 if heap-size < 1
2 then error “heap underflow”
3 max =  $A[1]$ 
4  $A[1] = A[A.\text{heap-size}]$ 
5  $A.\text{heap-size} = A.\text{heap-size} - 1$ 
6 MAX-HEAPIFY ( $A, 1$ )
7 return max
```

# Priority Queues<sup>3</sup>

## ☞ HEAP-INCREASE-KEY ( $A, i, key$ )      $O(\lg n)$

- 1 **if**  $key < A[i]$
- 2 **then error** “new key is smaller than current key”
- 3  $A[i] = key$
- 4 **while**  $i > 1$  and  $A[\text{PARENT}(i)] < A[i]$
- 5 exchange  $A[i]$  with  $A[\text{PARENT}(i)]$
- 6  $i = \text{PARENT}(i)$

## ☞ MAX-HEAP-INSERT( $A, key$ )      $O(\lg n)$

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

# Priority Queues<sup>4</sup>

