

# Computational Problems & Algorithms to Solve Them

Debswapna Bhattacharya, PhD  
Assistant Professor  
Computer Science and Software Engineering  
Auburn University

# Sorting

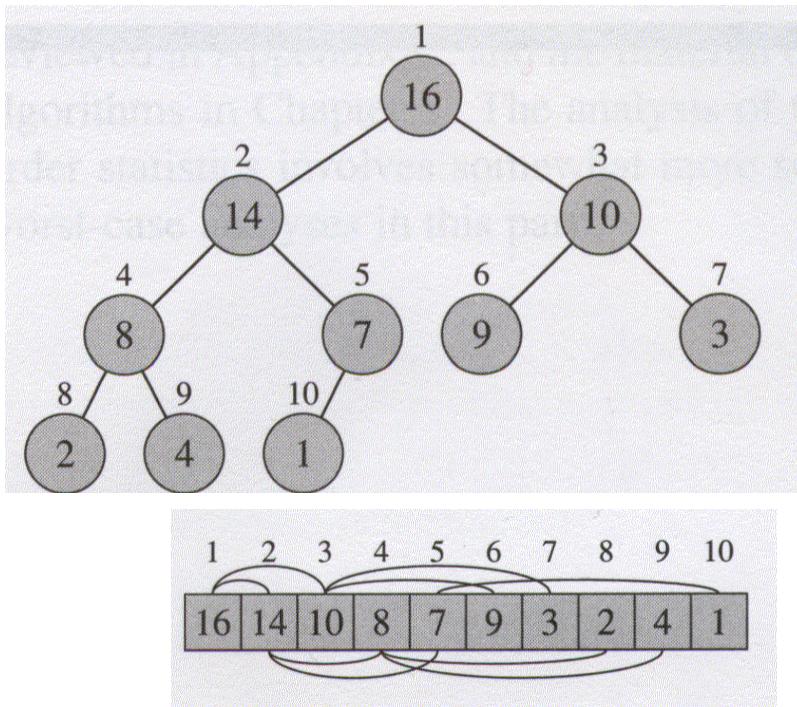
# Sorting Algorithms

- We have already discussed three  $O(n^2)$  Sorting Algorithms
  - Bubble
  - Selection
  - Insertion
- Thinking Assignment: Though both are  $O(n^2)$  algorithms, Selection will always run as or more efficiently than Bubble. Why?
- And one  $O(n\log n)$  sorting algorithm
  - Merge
- Next we will discuss two more  $O(n\log n)$  sorting algorithms
  - Heap
  - Quick
- And three linear sorting algorithms
  - Counting
  - Radix
  - Bucket

# The Binary Heap Data Structure and Heapsort

# Binary Heap

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



Parent( $i$ )

**return**  $[i / 2]$   
LEFT( $i$ )

**return**  $2i$

Right( $i$ )

**return**  $2i+1$

# Heap property

- **Max-heap** (the largest data item in the root):  
 $A[\text{parent}(i)] \geq A[i]$
- **Min-heap** (the smallest data item in the root):  
 $A[\text{parent}(i)] \leq A[i]$
- The *height of a node* in a tree: the number of edges on the longest downward path from the node to a leaf.
- The *height of a tree*: the height of the root
- *The height of a heap*:  
 $O(\lg n)$ .

# Priority Queues

A Binary Heap is an instance of a data structure called a Priority Queue that allows efficient detection and extraction of the data item with the largest or smallest “key” in a set of items without having to sort them.

# Priority Queue Operations

1. Build a priority queue of n data items
2. Locate the data item with the max/min key
3. Locate and extract the data item with the max/min key
4. Insert a new data item with its key into an existing Priority Queue
5. Increase or decrease the key value of a data item already in the Priority Queue

# Algorithms that implement these operations on a Max Binary Heap

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

# Maintaining the heap property

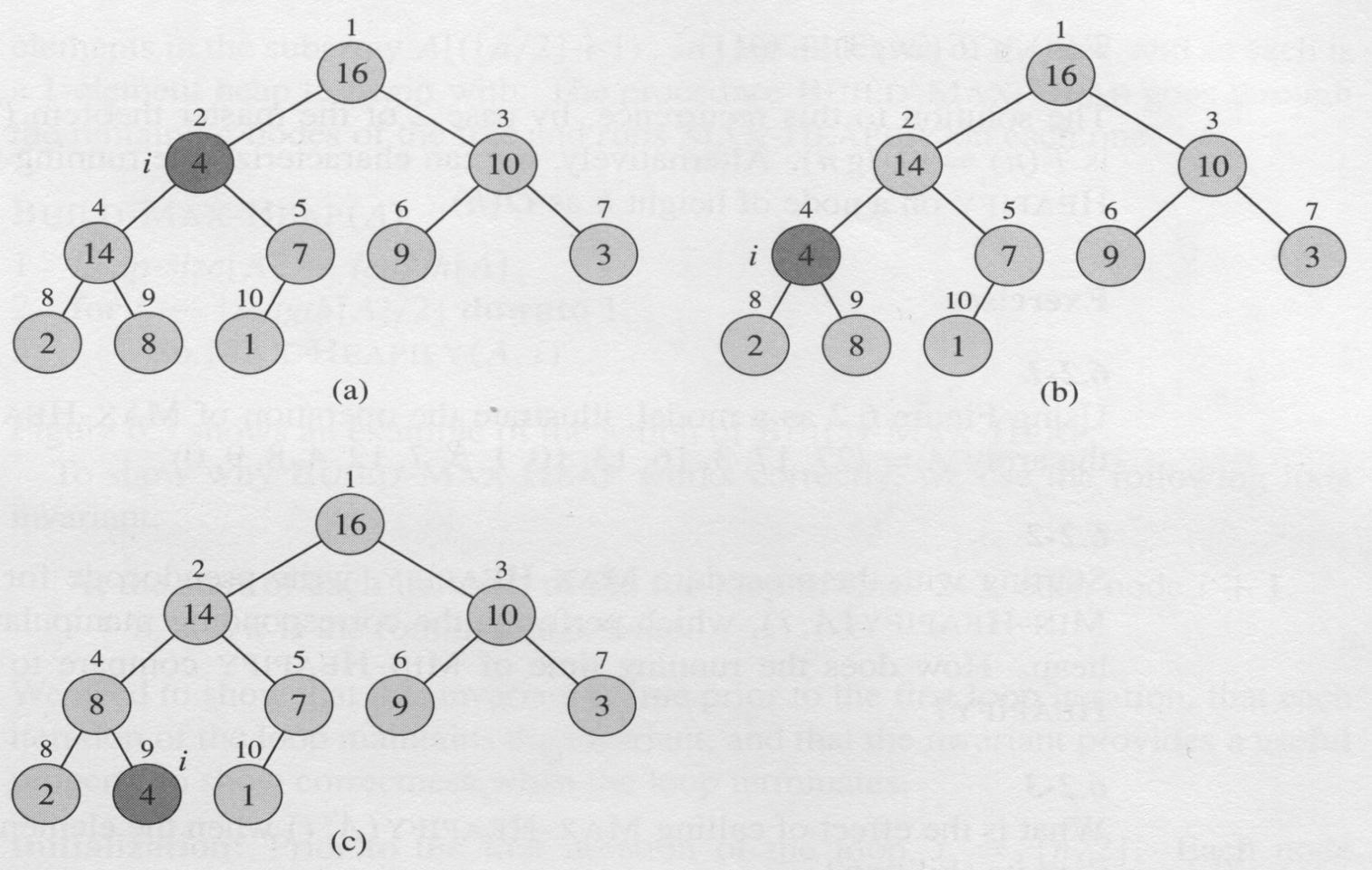
Max-Heapify is an important algorithm for manipulating heaps. Its inputs are an array  $A$  and an index  $i$  in the array. When Max-Heapify is called, it is expected that the binary trees rooted at  $\text{LEFT}(i)$  and  $\text{RIGHT}(i)$  are correct max heaps, but that  $A[i]$  may be smaller than its children, thus violating the max heap property at node  $i$ .

**Max-Heapify** (A[1...n]: array of number, i: int  $1 \leq i \leq n$ )

```
1 l = 2i
2 r = 2i + 1
3 if 1 ≤ A.heap-size and A[l] > A[i]
4     then largest = l
5     else largest = i
6 if r ≤ A.heap-size and A[r] > A[largest]
7     then largest = r
8 if largest ≠ i
9     then swap A[i] and A[largest]
10    Max-Heapify (A, largest)
```

Thinking Assignment: Modify Max-Heapify to get Min-Heapify for min heaps

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



# Max-Heapify Complexity

- It has to be  $O(\text{height})$  – why?
- Height of a heap is  $O(\lg n)$
- So Max-Heapify is  $O(\lg n)$

Alternately

- What is the base case?
- What are the recurrence relations? The size of the tree reduces by at least one third in each recursive call. So:
  - $T(\text{base case}) = c$
  - $T(n) = T(2n/3) + c$
- These can be solved to show  $T(n)=O(\lg n)$
- Thinking Assignment: Try it using the Master Theorem or Recursion Tree to solve these

# Building a heap

Build-Max-Heap( $A[1\dots n]$ : array of number)

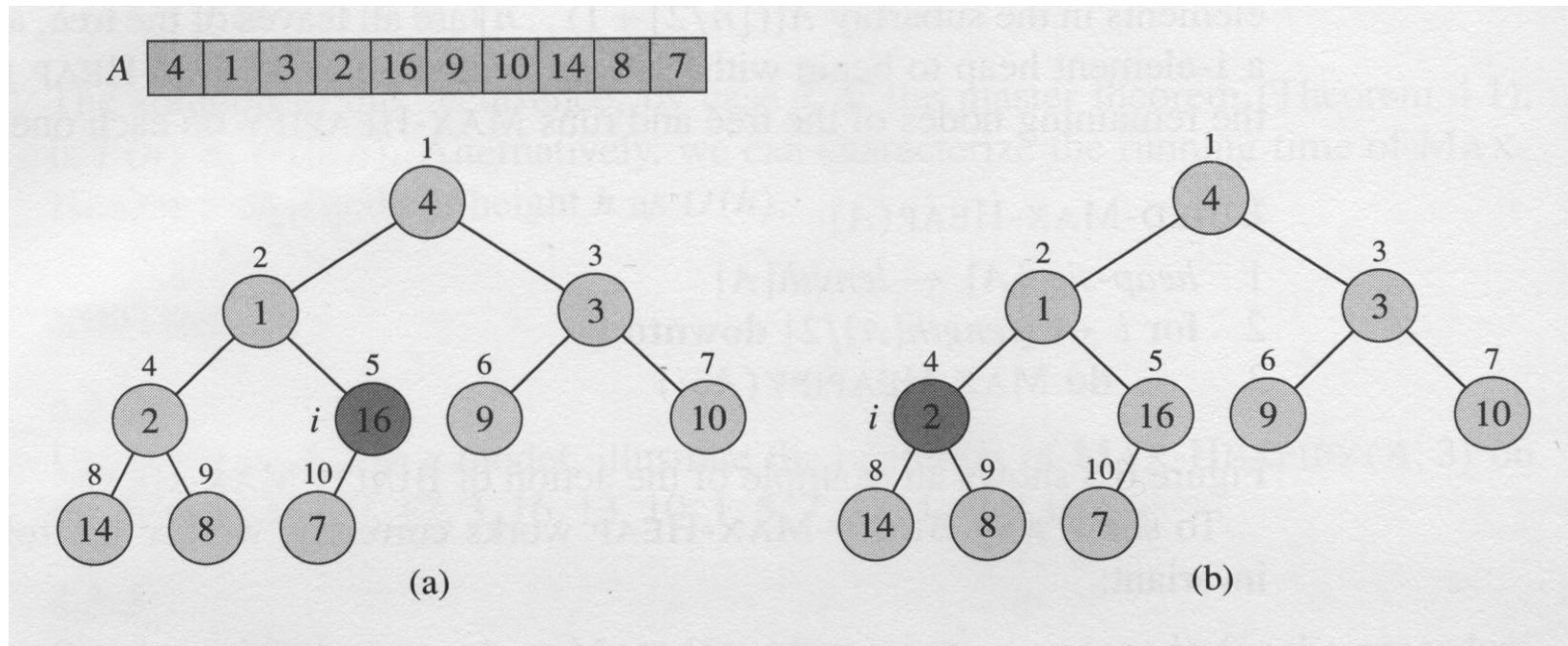
- 1  $A.\text{heap-size} = A.\text{length}$  { $A.\text{heap-size}$  used by Max-Heapify}
- 2 **for**  $i = \lfloor A.\text{length}/2 \rfloor$  **downto** 1
- 3     Max-Heapify( $A, i$ )

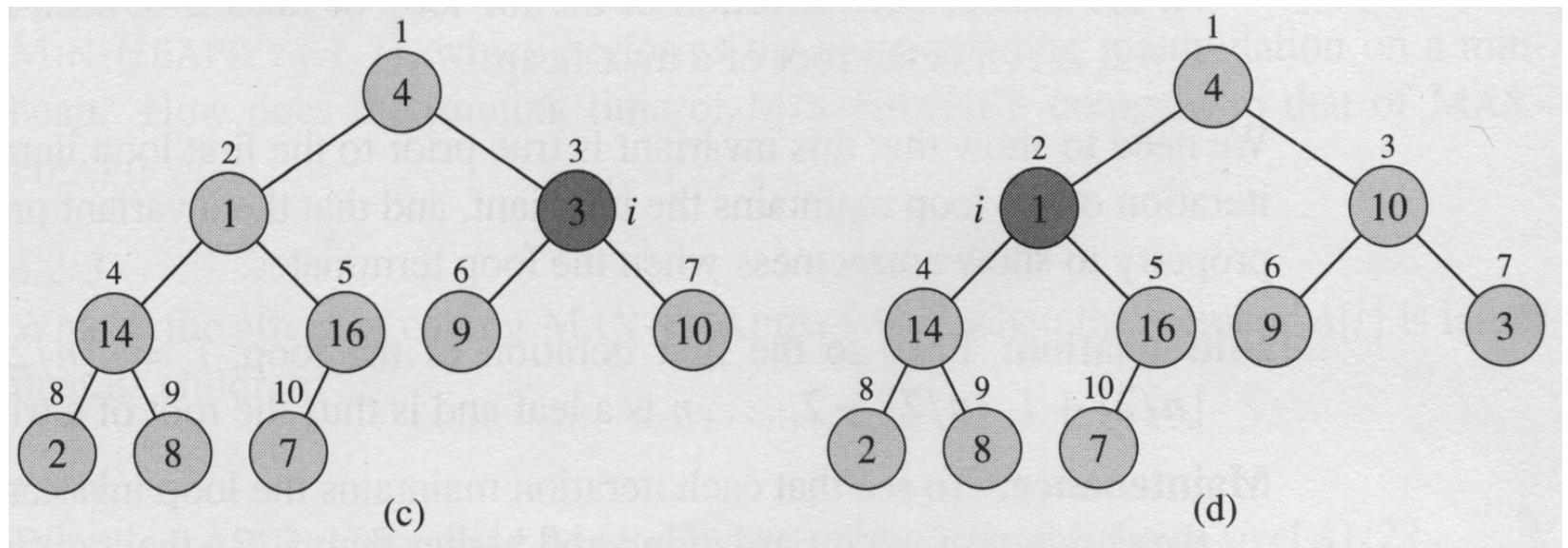
Thinking Assignments:

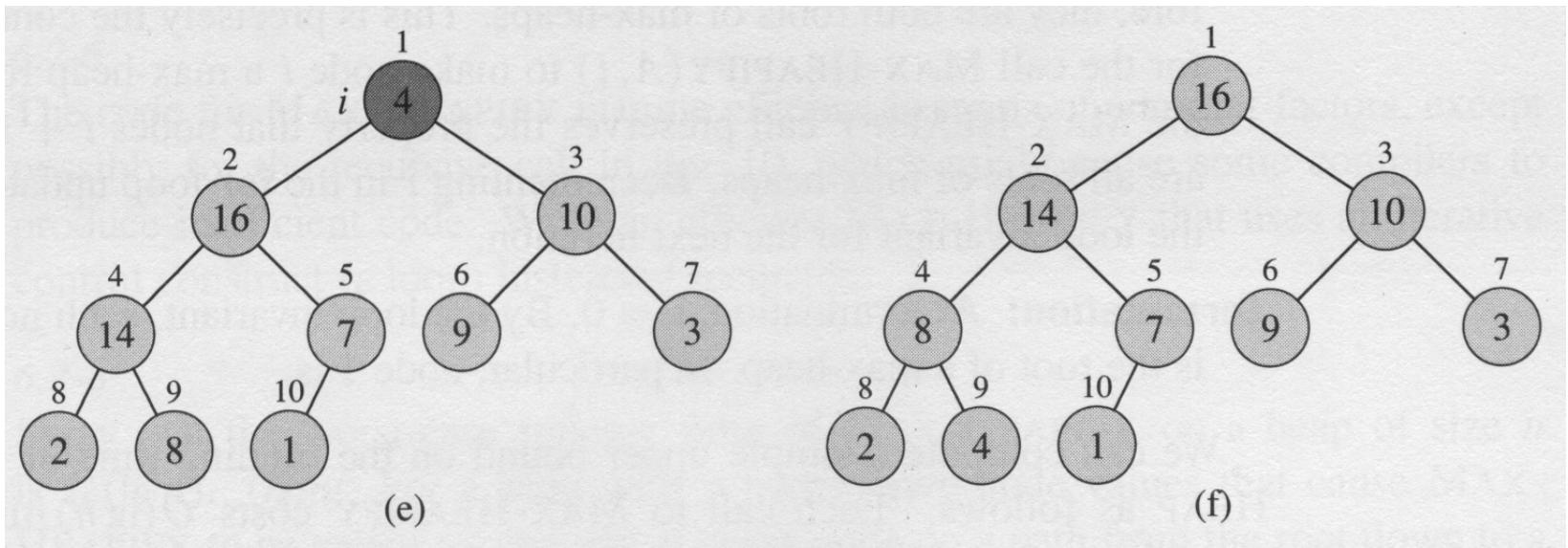
Why does this loop start from  $\lfloor \text{length}(A)/2 \rfloor$  ?

Why does the loop not go from 1 to  $\lfloor \text{heap-size}(A)/2 \rfloor$  ?

# Build-Max-Heap







# What is Build-Max-Heap's complexity?

$O(nlgn)$  is a good estimate – why?

But a tighter upper bound can be found: Build-Max-Heap is actually  $O(n)$  – linear!

Optional Reading Assignment: If you want to know why, read text p. 159

# The Heapsort algorithm

Heapsort( $A[1\dots n]$ : array of number)

- 1 Build-Max-Heap( $A$ )
- 2 **for**  $i = A.length$  **down to** 2
- 3     swap  $A[1]$  and  $A[i]$
- 4      $A.heap-size = A.heap-size - 1$
5.     Max-Heapify( $A, 1$ )

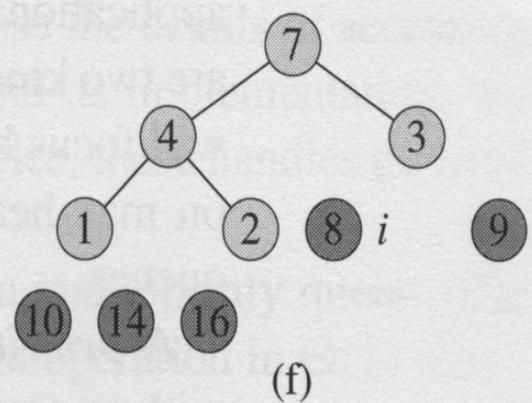
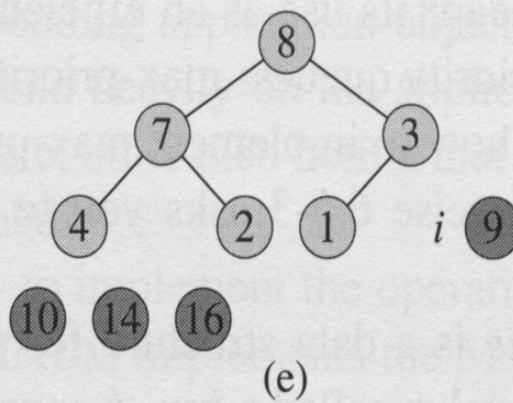
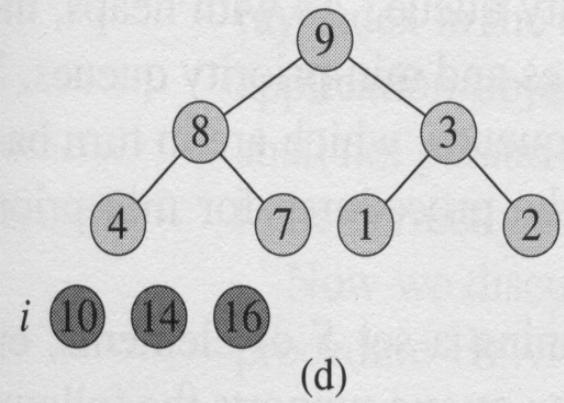
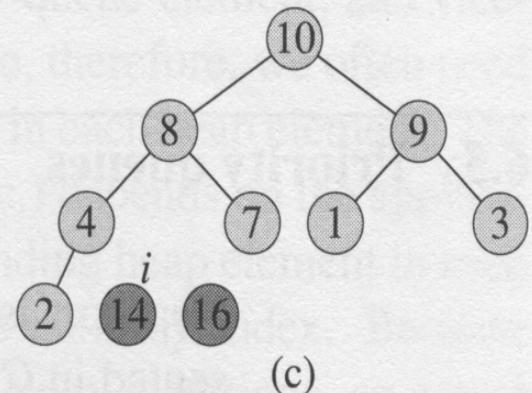
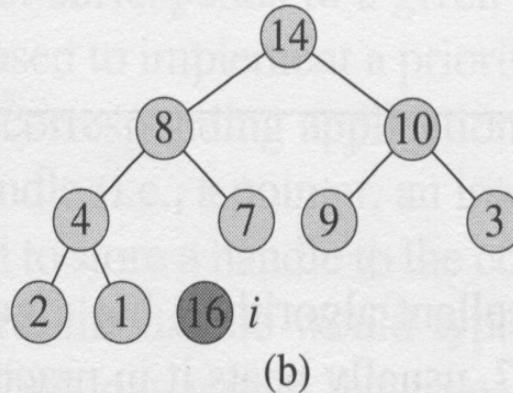
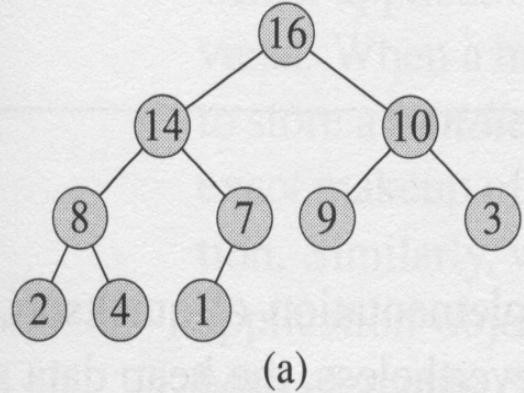
Thinking Assignments:

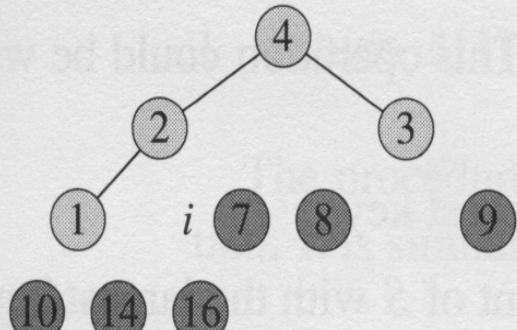
Does this algorithm sort in ascending order?

Why is its worst-case complexity  $O(nlgn)$ ?

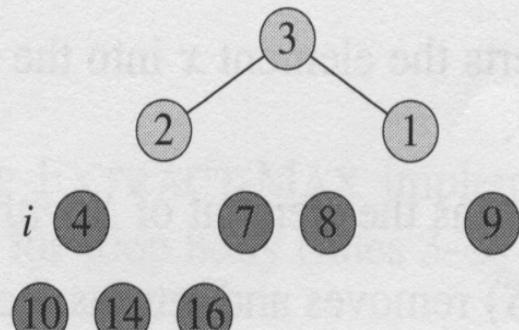
What is its best case complexity? For what kind of input?

# The operation of Heapsort

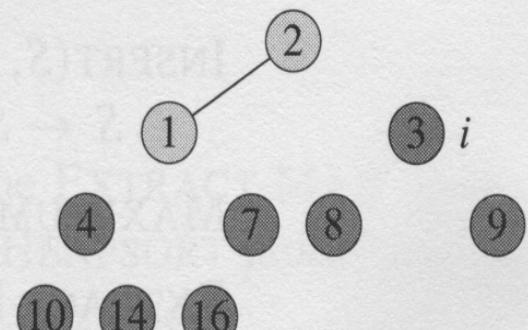




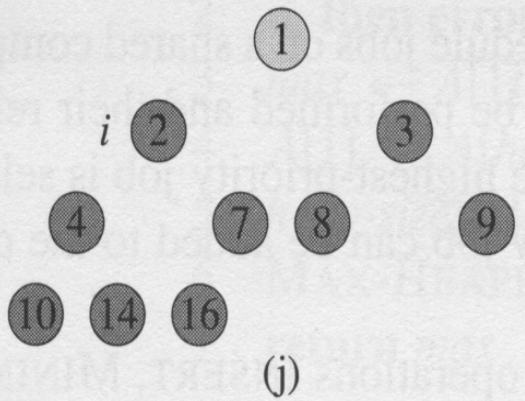
(g)



(h)



(i)



(j)

$A$	1	2	3	4	7	8	9	10	14	16
-----	---	---	---	---	---	---	---	----	----	----

(k)

Complexity:  $O(n \log n)$

# Recap: Priority Queues

The heap data structure is not only used in sorting but also in priority queues. A **priority queue** is a data structure that maintain a set  $S$  of elements, each with an associated value call a **key**. Priority queues has many applications. A **max (or min) priority queue** should at least support the following operations:

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

# Two Heap Operations

**Heap-Maximum(A[1...n]: array of number)**

1 return A[1]

**Heap-Extract-Max(A[1...n]: array of number)**

1 if A.heap-size < 1

2 then error “heap underflow”

3 max = A[1]

4 A[1] = A[A.heap-size]

5 A.heap-size = A.heap-size - 1

6 Max-Heapify (A, 1)

7. return max

**Thinking Assignment: Write Heap-Extract-Min**

**Max-Heap-Increase-Key**(A[1...n]: array of number, i: int  $1 \leq i \leq n$ , 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]    {Parent(i) returns  $\lfloor i/2 \rfloor$  }
- 5    swap A[i] and A[Parent(i)]
6.     i = Parent(i)

Thinking Assignments:

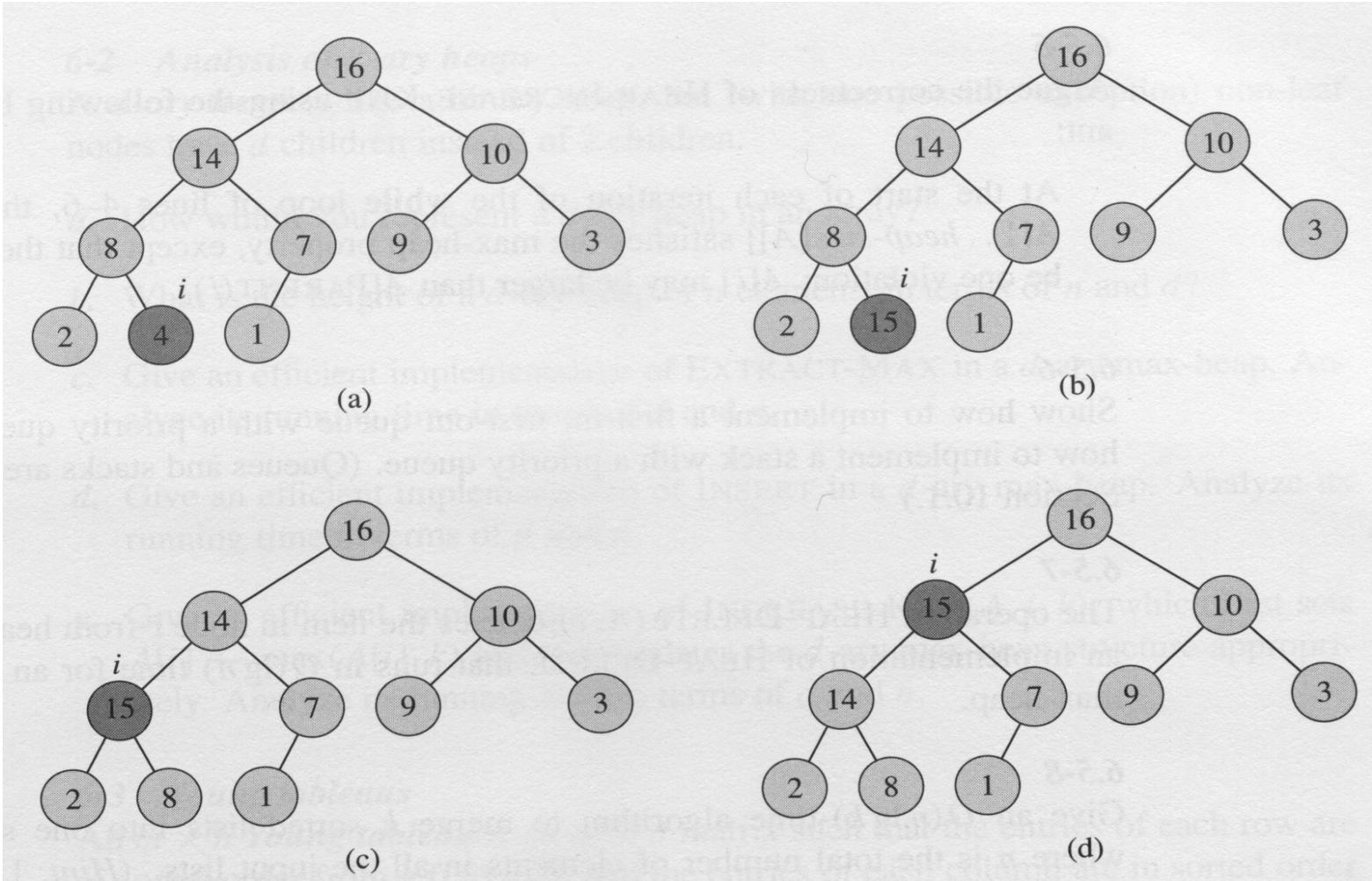
What is this algorithm's complexity?

Write Max-Heap-Decrease-Key(A, i, key)

Write Min-Heap-Increase-Key(A, i, key)

Write Min-Heap-Decrease-Key(A, i, key)

# Max-Heap-Increase-Key



Max-Heap-Insert( $A[1\dots n]$ : array of number, key)

- 1  $A.\text{heap-size} = A.\text{heap-size} + 1$
- 2  $A[A.\text{heap-size}] = -\infty$
3. Max-Heap-Increase-Key ( $A, A.\text{heap-size}, \text{key}$ )

Thinking Assignment: Write Min-Heap-Insert( $A, \text{key}$ )

# Algorithm Design Thinking Assignments

- Min-Heapify
- Build-Min-Heap
- Heapsort procedure using a Min heap
- Heap-Extract-Min
- Heap-Minimum
- Min-Heap-Decrease-Key
- Min-Heap-Insert
- Max-Heap-Decrease-Key
- Min-Heap-Increase-Key

## Ch. 6 Reading Assignments

All of the chapter (make sure you understand the Loop Invariant correctness proof of BUILD-MAX-HEAP on p. 157)

## Ch. 6 Thinking Assignments

Exercises: 6.1-1:6.1-7, 6.2-1:6.2-6,  
6.3-1:6.3-2, 6.4-1:6.4-4, 6.5-1:6.5-8