Sorting

A universal problem!

Many applications often incorporate sorting

There is a wide variety of sorting algorithms, and they use rich set of techniques.

Sorting algorithm

- Insertion, Bubble, Selection sorts:
  - Non-recursive
  - In place: only a constant number of additional storage locations (for local variables) are used outside the array.
- Merge sort :
  - Recursive; Not in place.
- Heap sort : (Chapter 6)
  - Non-recursive
  - Sorts n numbers in place in O(nlgn)

Sorting algorithm

- Quick sort : (chapter 7)
  - Recursive
  - In place
  - Worst time complexity O(n^2); Average time complexity O(n log$n$)
  - Fastest general purpose sorting algorithm
- Linear sorting algorithms : (chapter 8)
  - Counting sort
  - Radix sort
  - Bucket Sort

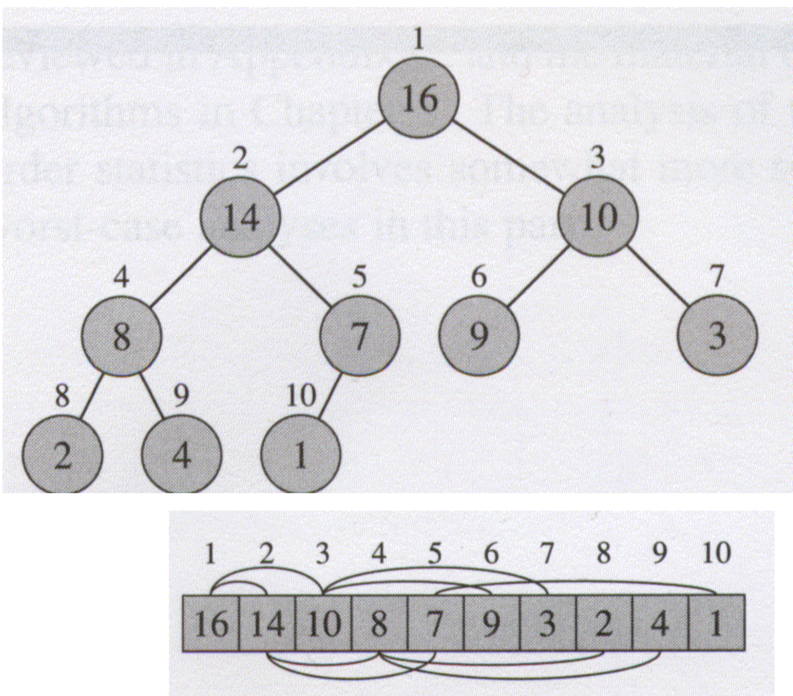
Heapsort

Ch. 6 Reading Assignments

<u>All of the chapter</u> (make sure you understand the Loop Invariant correctness proof of BUILD-MAX-HEAP on p. 157 but omit its mathematical proof of complexity on p.159)

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 : $A[\text{parent}(i)] \geq A[i]$
- Min-heap : $A[\text{parent}(i)] \leq A[i]$
- 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***: floor(lg n)=O(lg *n*).

Basic algorithms on max 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

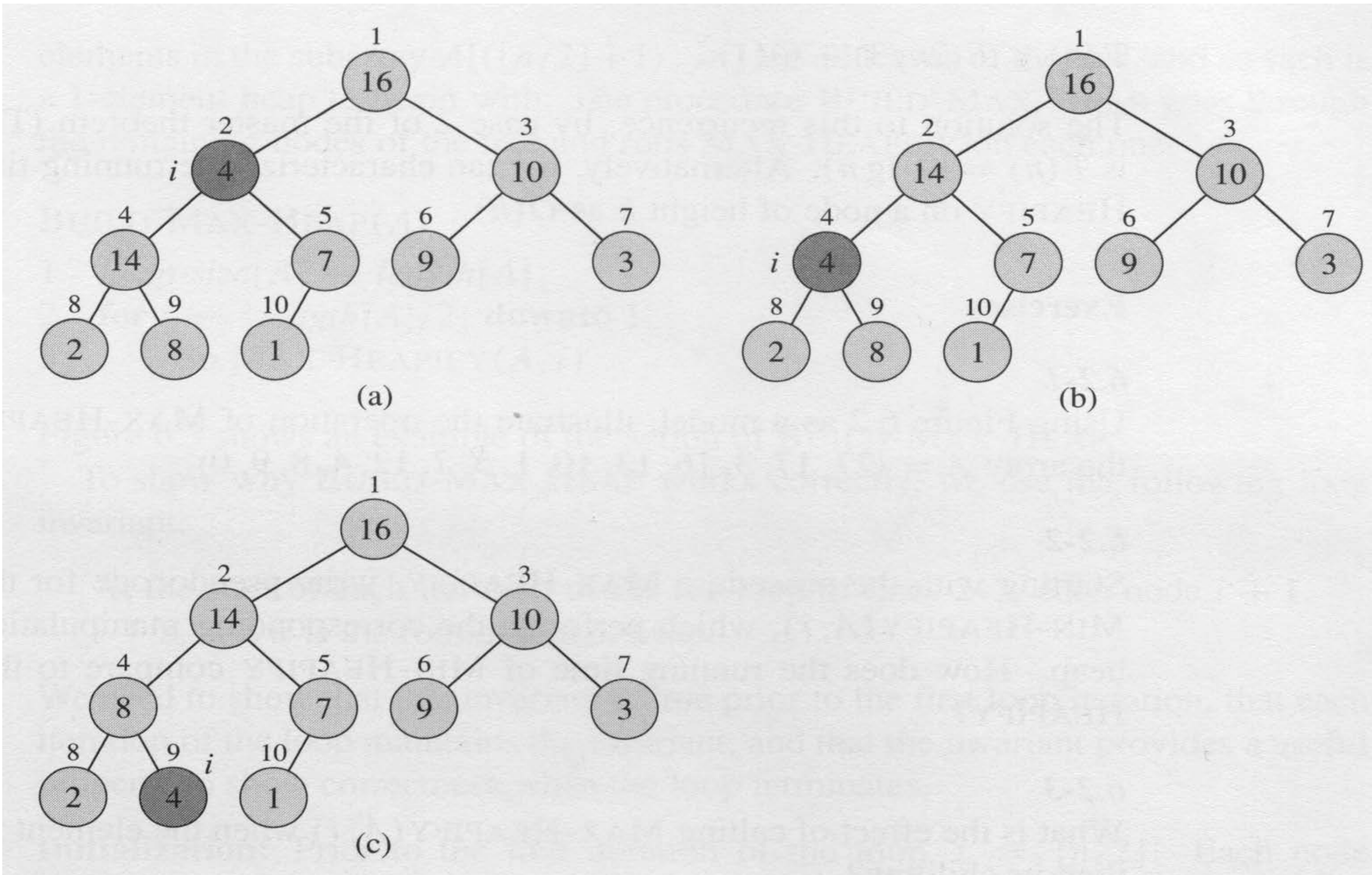
6.2 Maintaining the heap property

- Max-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 assumed 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 max heap property.

```
Max-Heapify (A, i)
l = 2i
r = 2i+1
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 swap A[i] and A[largest]
                Max-Heapify (A, largest)
```

Thinking Assignment: How about Min-Heapify?

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



(a) (b) (c)

Max-Heapify Complexity

- It has to be O(height) – why?
- Height of a heap is O(lgn)
- So Max-Heapify is O(lgn)

Alternately

- What is the base case?
- What are the recurrence relations?

$$T(base\,case) = \Theta(1) = c$$

$$T(n) \leq T(\frac{2n}{3}) + \Theta(1)\ or\ T(\frac{2n}{3}) + c$$

- These can be solved to show T(n)=O(lgn)
- Thinking Assignment: Try it!

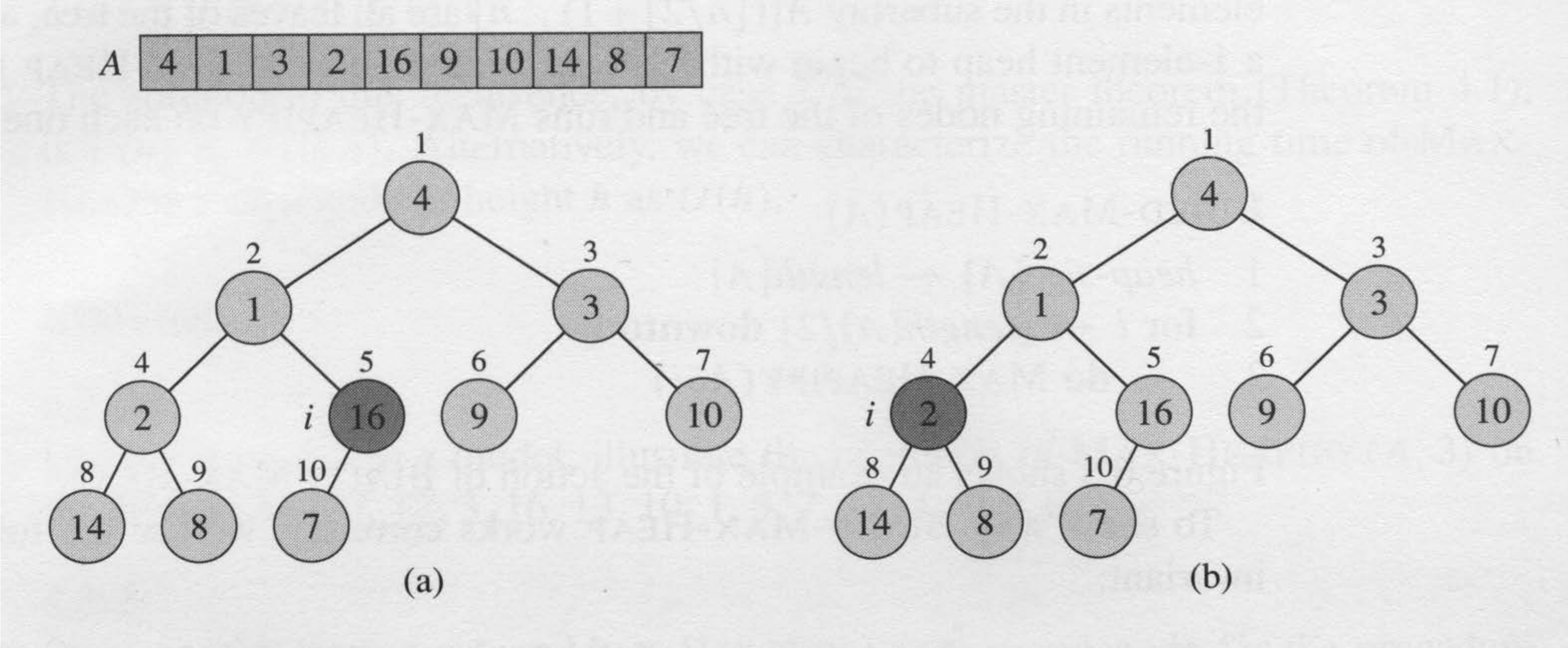
6.3 ***Building a heap***

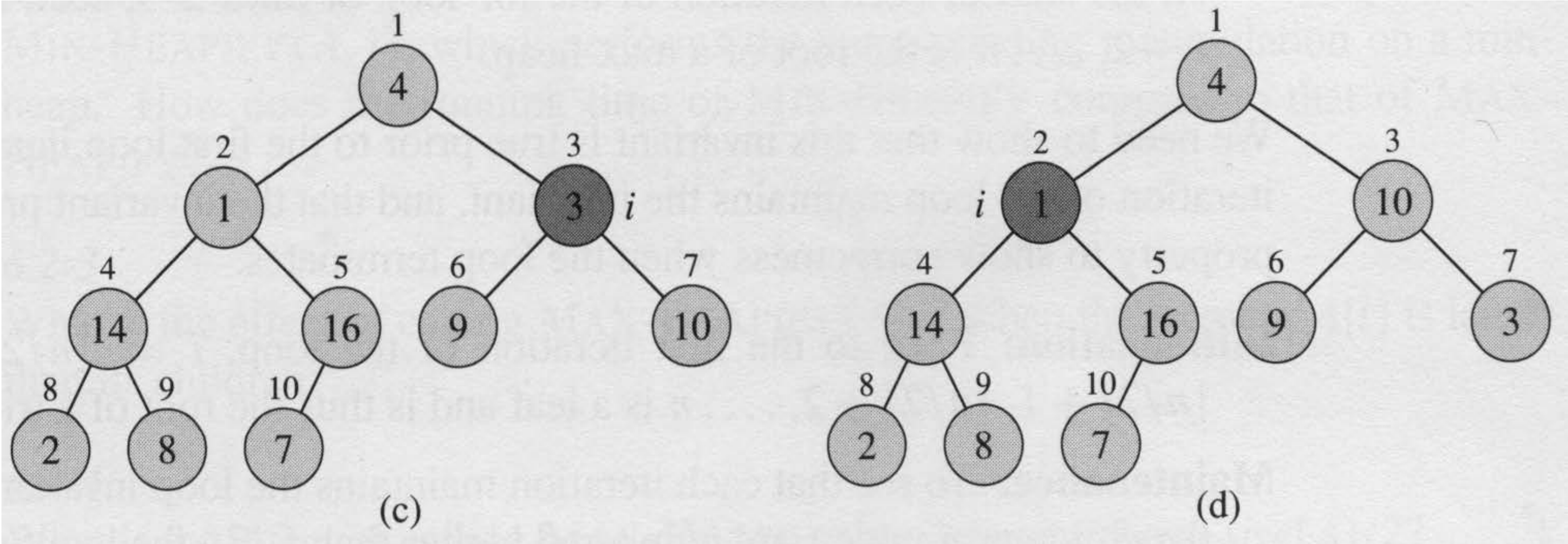
```
Build-Max-Heap(A)
1   heap-size(A) = length(A)
2   for i = ⌊heap-size(A)/2⌋ downto 1
3       Max-Heapify(A, i)
```

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

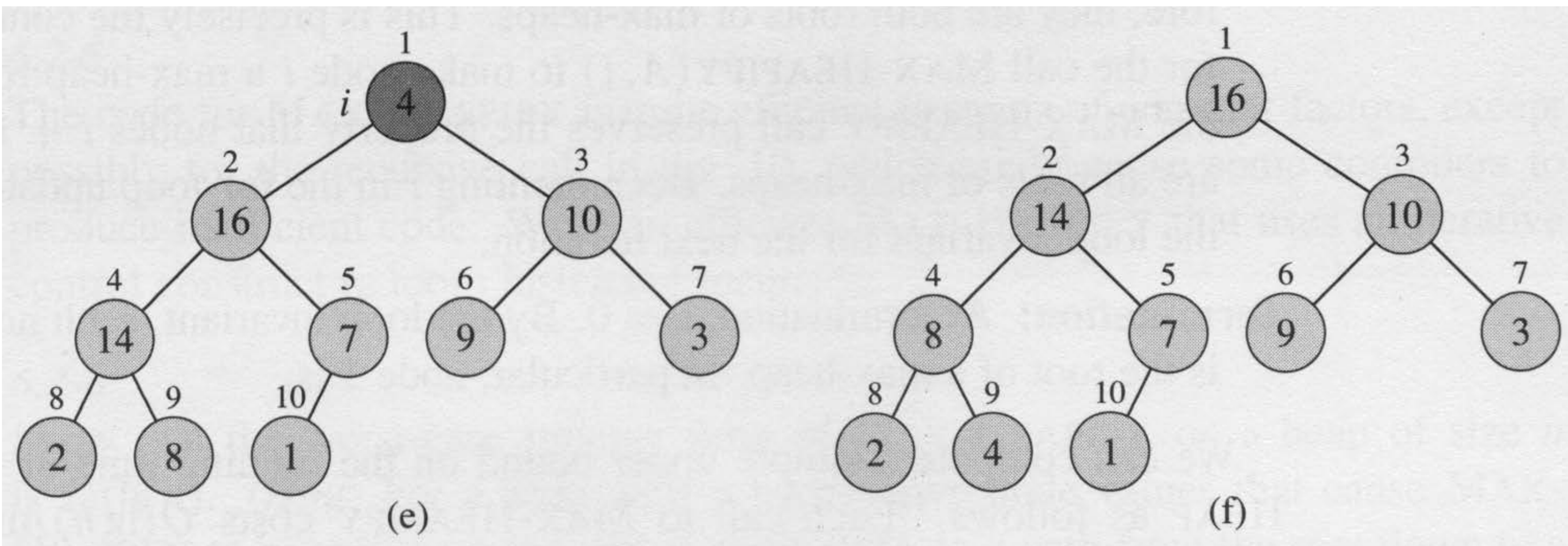
Why does the loop not go from 1 to $\lfloor heap\text{-}size(A)/2 \rfloor$?

Build-Max-Heap





1
4
2
1
3
3
i
4
14
5
16
6
9
7
10
8
2
9
8
10
7
(c)
1
4
2
1
i
3
10
4
14
5
16
6
9
7
3
8
2
9
8
10
7
(d)


1
i
4
2
16
3
10
4
14
5
7
6
9
7
3
8
2
9
8
10
1
(e)
1
16
2
14
3
10
4
8
5
7
6
9
7
3
8
2
9
4
10
1
(f)

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!

If you want to know why, look at the next slide and read the text p. 159 (optional)

- $O(n \log n)$? We can find a tighter upper bound!

$$n-element\ heap\ has\ height \lfloor \lg n \rfloor$$

$$and \left\lceil \frac{n}{2^{h+1}} \right\rceil nodes\ at\ height\ h$$

$$T(n)\ for\ the\ a\lg orithm =$$

$$\sum_{h=1}^{\lfloor \lg n \rfloor} ((\#of\ \ nodes\ at\ height\ h) * O(h)) + c$$

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

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

$$and \sum_{h=0}^{\infty} \frac{h}{2^h} = 2\ (\,because \sum_{k=0}^{\infty} kx^k\ = \frac{x}{(1-x)^2})$$

$$So\, O(n \sum_{h=0}^{\infty} \frac{h}{2^h}) = O(n * 2) = O(n)$$

$$T(n) < \sum_{h=0}^{\infty} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O(n),\, so\, T(n) = o(n)$$

6.4 The Heapsort algorithm

Heapsort(A)

1 Build-Max-Heap(A)

2 **for** i = length(A) **down to** 2

3 swap A[1] and A[i]

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

5 Max-Heapify(A,1)

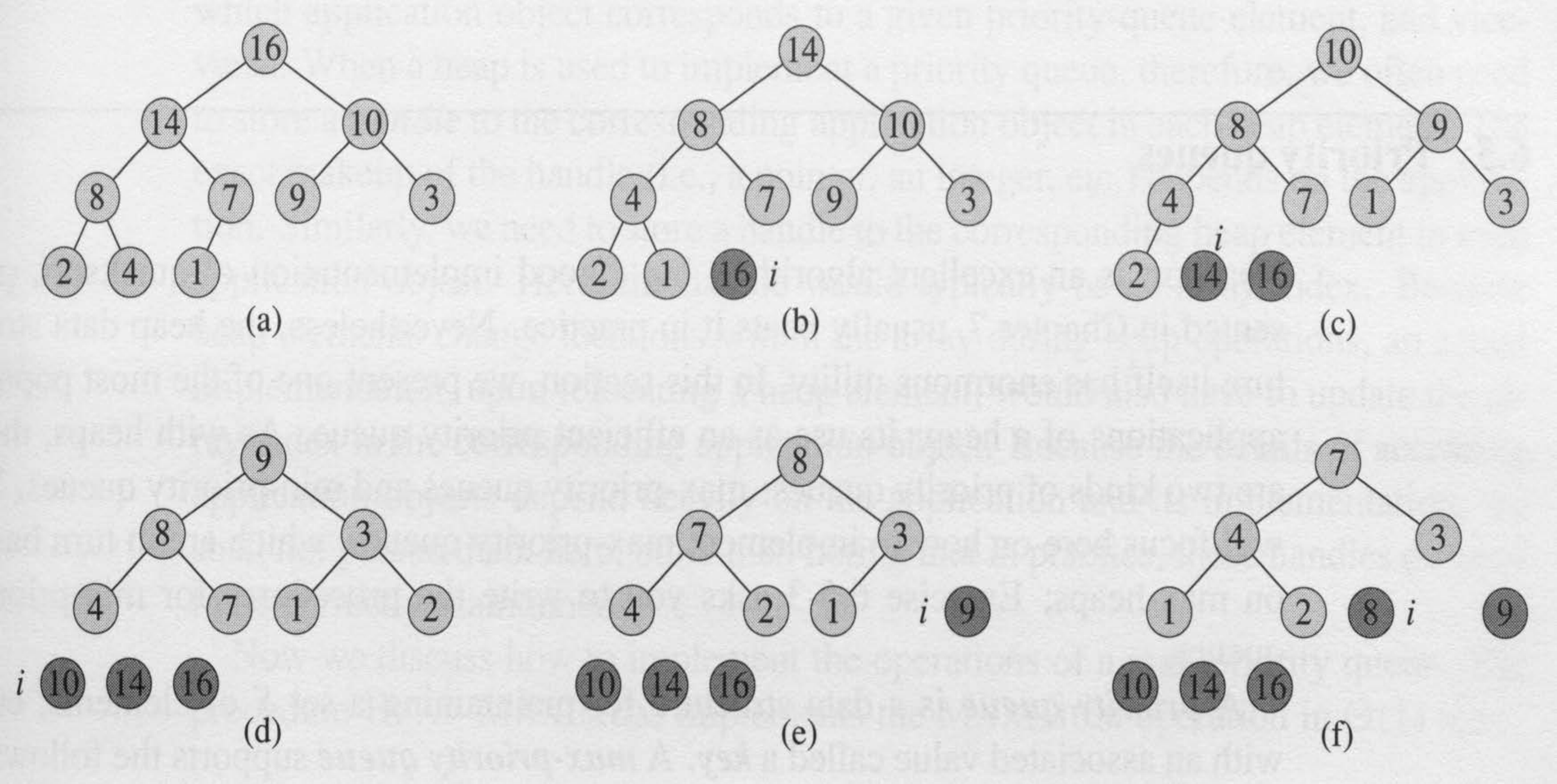
Thinking Assignment:

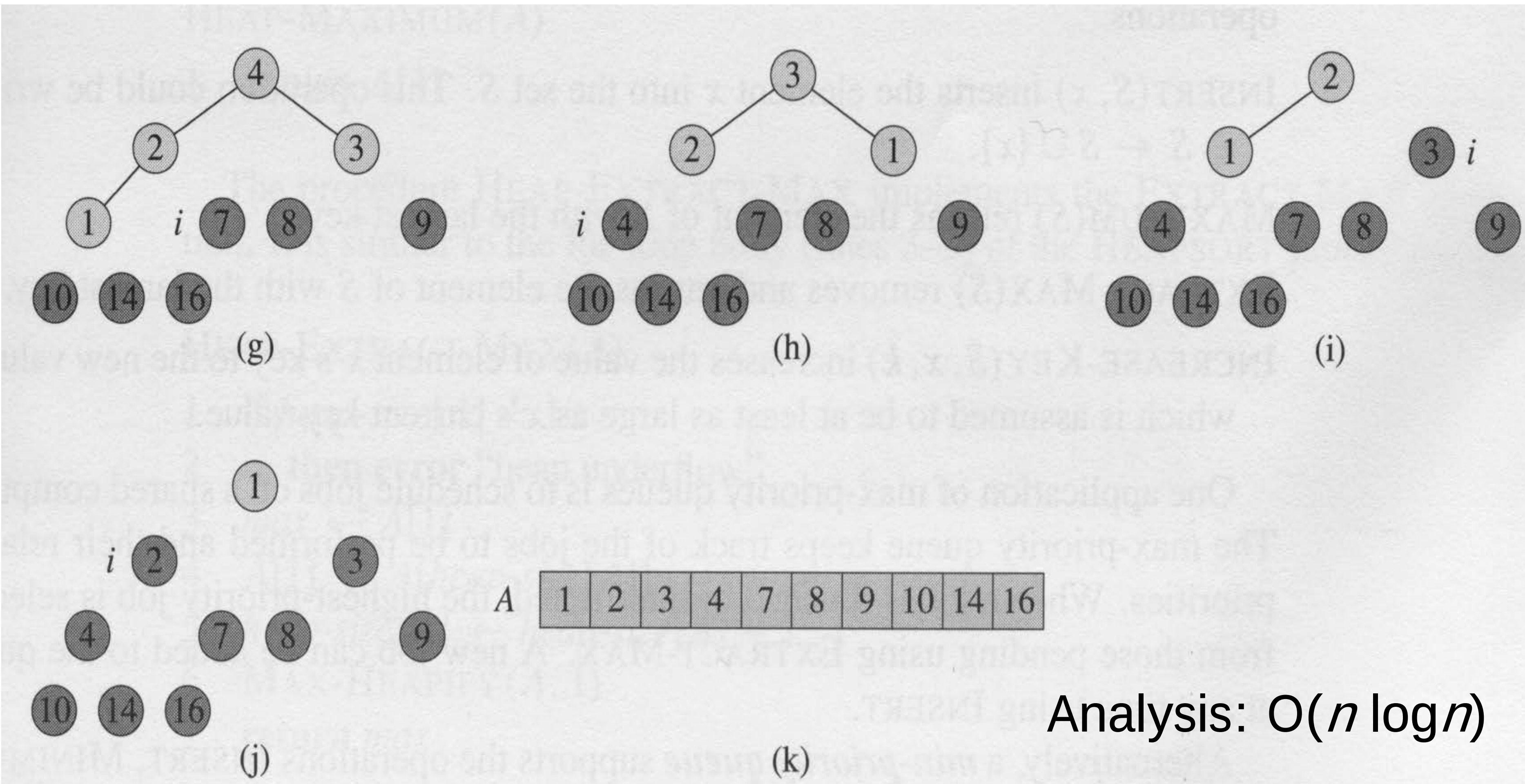
Does this algorithm sort in ascending order?

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

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

The operation of Heapsort





Analysis: $O(n \log n)$

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 maintains 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-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-Maximum(A)

```
return A[1]
```


Thinking Assignment: Write Heap-Extract-Min(A)

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]

swap A[i] and A[Parent(i)]

i = Parent(i)

Thinking Assignment:

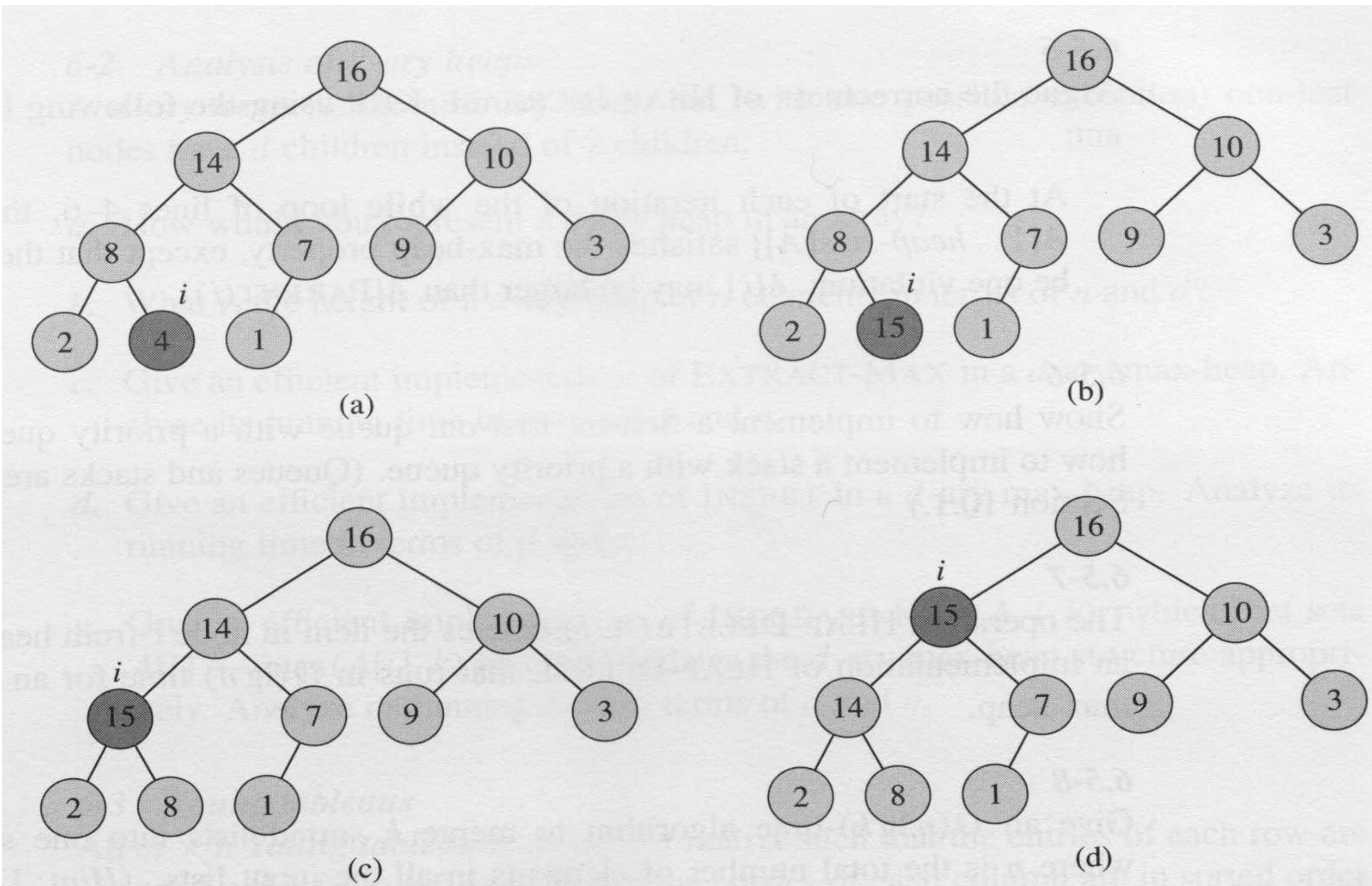
What is this algorithm’s complexity?

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

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

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

Heap-Increase-Key



Max-Heap-Insert(A, *key*)

heap-size(A) = heap-size(A) + 1
A[heap-size(A)] = $-\infty$
    Max-Heap-Increase-Key (A, heap-size(A), key)

Thinking Assignment: Write Min-Heap-Insert(A, key)

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