Design and Analysis of Algorithms

CSE 5311

Lecture 6 Heap Sort

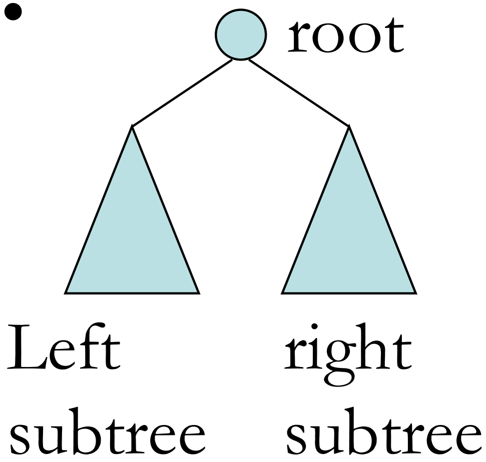
Junzhou Huang, Ph.D.

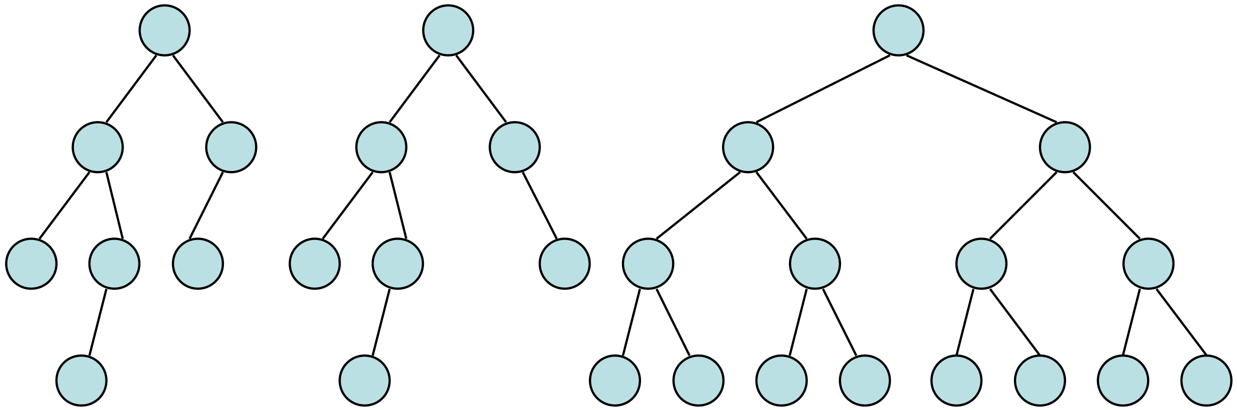
Department of Computer Science and Engineering

Heap Data Structure

- **Definition**
  - (binary) heap data structure is an array object that we can view as a nearly complete binary tree
- **A node of the tree corresponds to an element of the array *A[1…n]***
  - n: heap size
  - A[1]: root
  - [*i*/2]: parent of node *i*
  - 2*i*: left child of node *i*
  - 2*i*+1: right child of node *i*

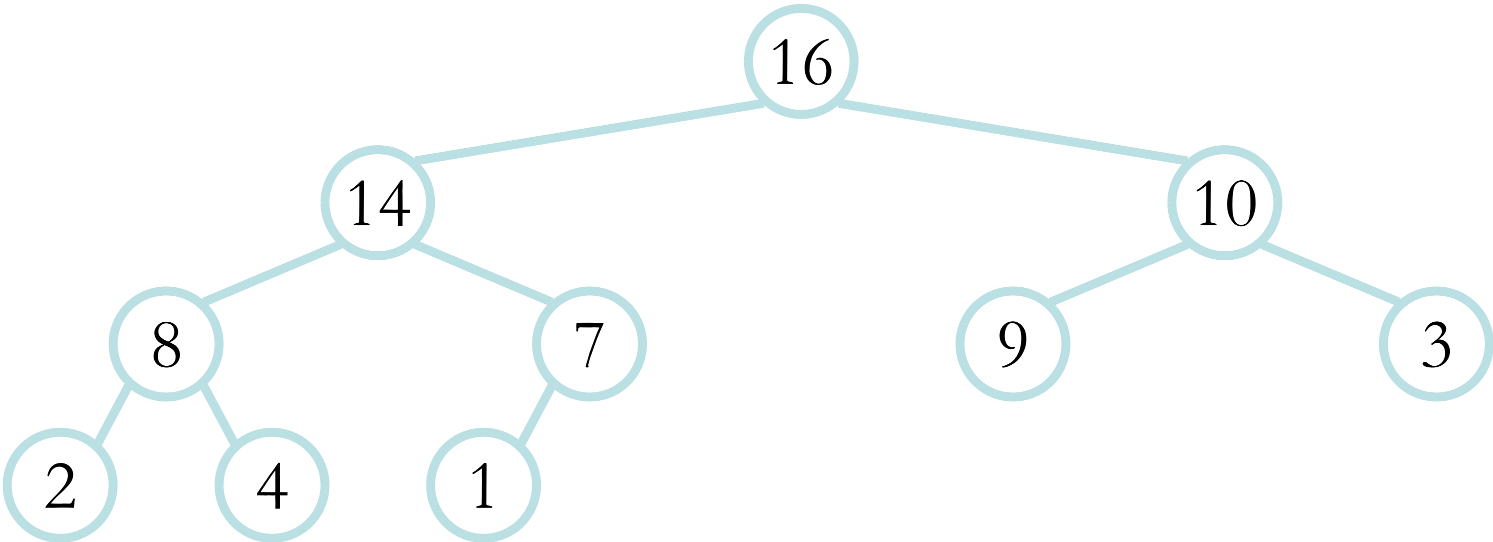
Binary Tree

- Contains no node, or
- 

eg.

- A node without subtree is called a leaf.
- In a full binary tree, each node has 2 or NO children.
- A complete binary tree has all leaves with the same depth and all internal nodes have 2 children.

Heaps

- A *heap* is a “complete” binary tree, usually represented as an array:

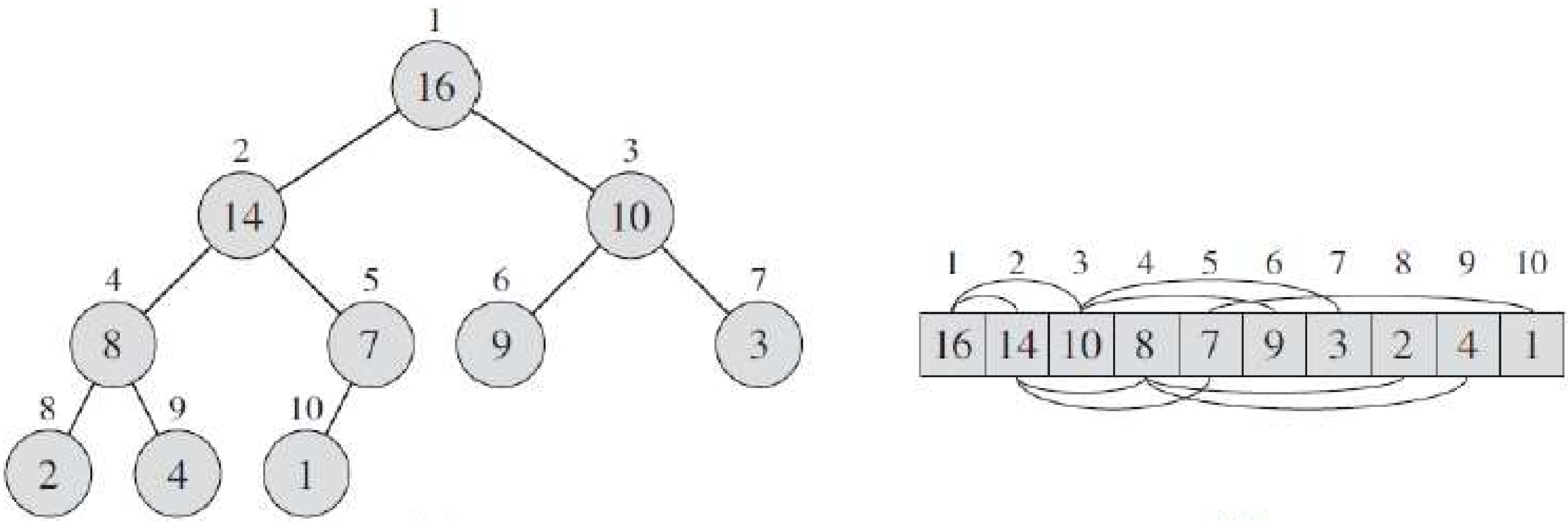


A =

16	14	10	8	7	9	3	2	4	1

Heap Data Structure

- Height of node = # of edges on a longest simple path from the node down to a leaf
- Height of heap = height of root



(a) (b)

Heaps

- To represent a heap as an array:

```
Parent(i) { return ⌊i/2⌋; }
Left(i) { return 2*i; }
right(i) { return 2*i + 1; }
```

Properties of Heap

- **Height of heap:** $\Theta(\lg n)$
- **Max-heap:**
  - $A[\text{PARENT}(i)] \geq A[i]$ for all node i except the root
  - Root store the largest value
- **Min-heap:**
  - $A[\text{PARENT}(i)] \leq A[i]$ for all node i except the root
  - Root store the smallest value

The Heap Property

- **Heaps also satisfy the *heap property*:** **Max-heap!**

  $A[Parent(i)] \geq A[i]$ for all nodes $i > 1$

  - In other words, the value of a node is at most the value of its parent
  - The largest value is thus stored at the root (A[1])
- **Because the heap is a binary tree, the height of any node is at most $\Theta(\lg n)$**

Heapify()

- **Heapify()**: maintain the heap property
  - Given: a node *i* in the heap with children *l* and *r*
  - Given: two subtrees rooted at *l* and *r*, assumed to be heaps
  - Action: let the value of the parent node "float down" so subtree at *i* satisfies the heap property
    - ➢If A[i] < A[l] or A[i] < A[r], swap A[i] with the largest of A[l] and A[r]
    - ➢Recurse on that subtree
  - Running time: O(*h*), h = height of heap = O(lg *n*)

Pseudocode Heapify(A,i)

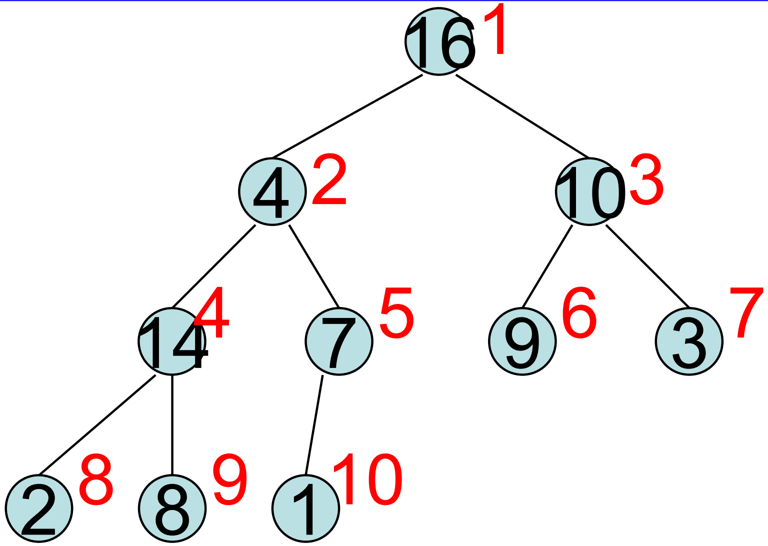
```
MAX-HEAPIFY(A, i)
l ← LEFT(i)
r ← RIGHT(i)
if l ≤ heap-size[A] and A[l] > A[i]
    then largest ← l
    else largest ← i
if r ≤ heap-size[A] and A[r] > A[largest]
    then largest ← r
if largest ≠ i
    then exchange A[i] ↔ A[largest]
         MAX-HEAPIFY(A, largest)
```

Heapify(A,i)

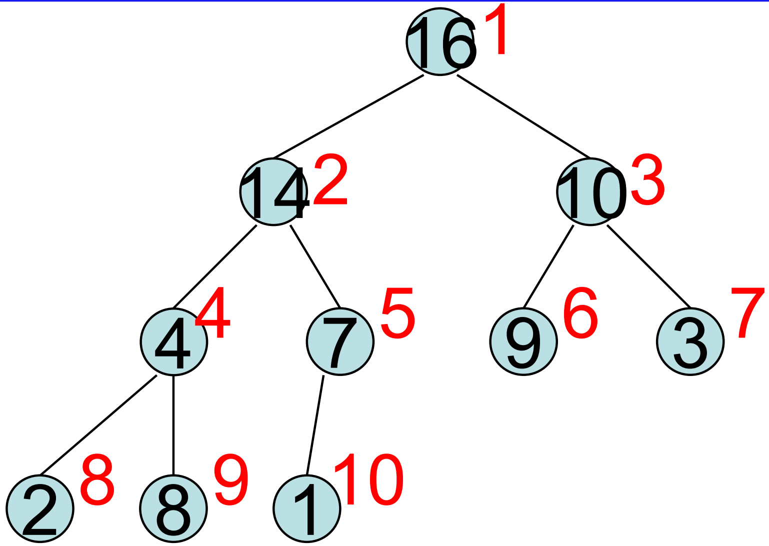
- **Running Time:**
  - The running time of HEAPIFY on a subtree of size *n* rooted at a given node *i* is the $\Theta(1)$ time to fix up the relationships among the elements A[i], A[LEFT(i)], and A[RIGHT(i)]
  - Plus the time to run HEAPIFY on a subtree rooted at one of the children of node i (assuming that the recursive call occurs).
- **Formula:**
  - The children's subtrees each have size at most 2n/3
  - The worst case occurs when the bottom level of the tree is exactly half full

Time: $O(\lg n)$, $T(n) \le T(2n/3) + \Theta(1)$

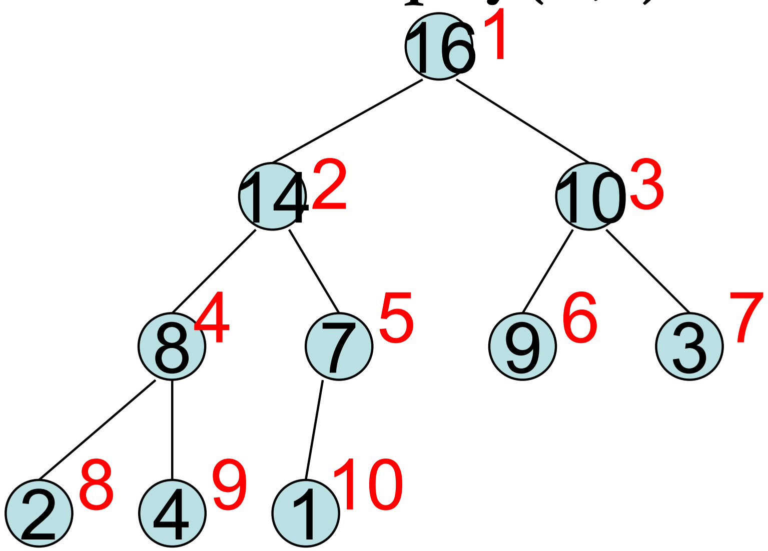
Heapify(A,2):



Heapify(A,4):



Heapify(A,9):

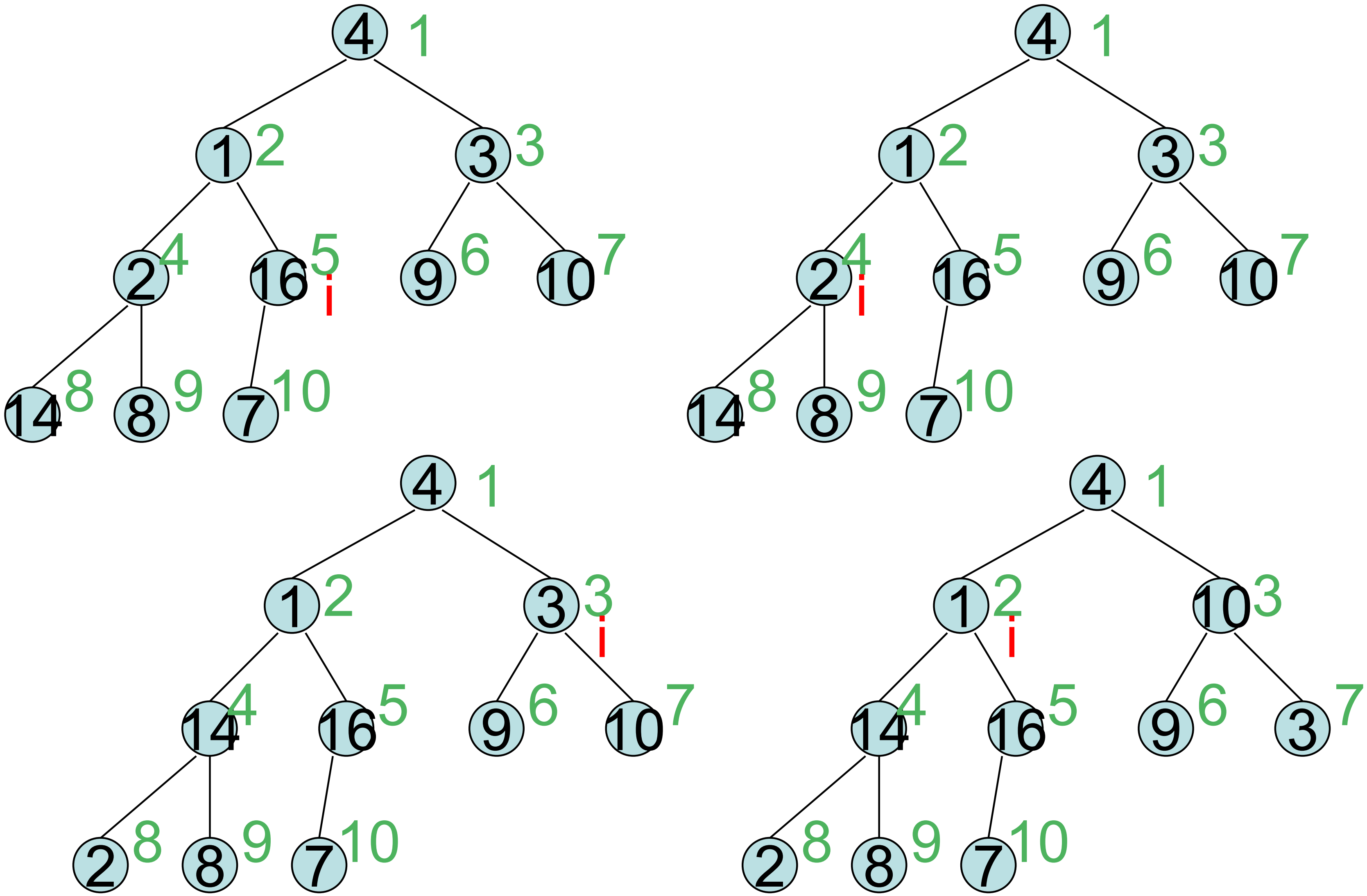


BuildHeap()

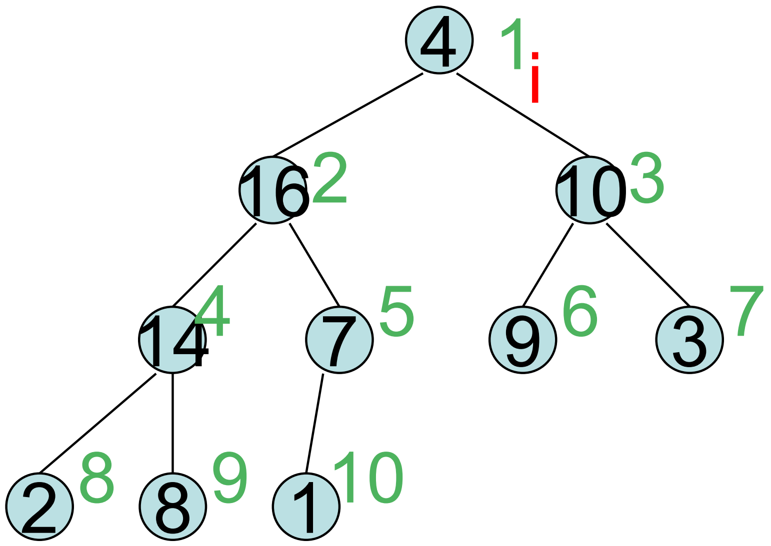
- We can build a heap in a bottom-up manner by running **Heapify()** on successive subarrays
  - Fact: for array of length *n*, all elements in range $A[\lfloor n/2 \rfloor + 1 .. n]$ are heaps (***Why?***) **single node**
  - So:
    - Walk backwards through the array from n/2 to 1, calling **Heapify()** on each node.
    - Order of processing guarantees that the children of node *i* are heaps when *i* is processed

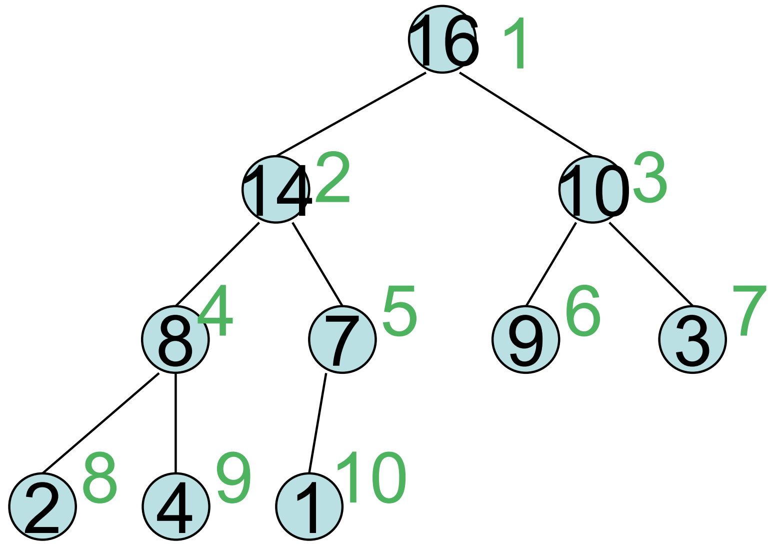
BuildHeap()

	1	2	3	4	5	6	7	8	9	10
A	4	1	3	2	16	9	10	14	8	7



BuildHeap()






```
// given an unsorted array A, make A a heap
BuildHeap(A)
{
   heap_size(A) = length(A);
   for (i = ⌊length[A]/2⌋ downto 1)
        Heapify(A, i);
}
```

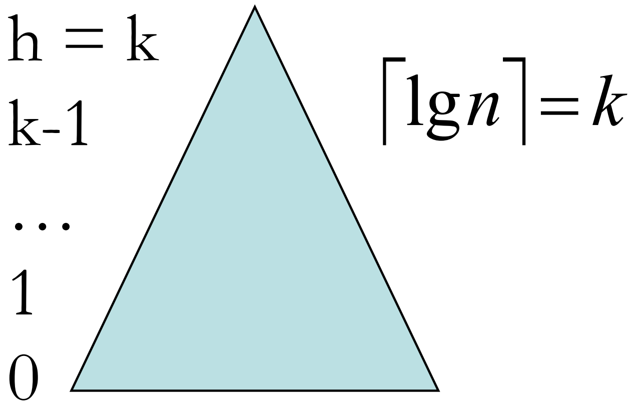
Analyzing BuildHeap()

- Each call to **`Heapify()`** takes O(lg *n*) time
- There are O(*n*) such calls (specifically, $\lfloor n/2 \rfloor$)
- Thus the running time is O(*n* lg *n*)
  - *Is this a correct asymptotic upper bound?*
  - *Is this an asymptotically tight bound?*
- A tighter bound is O(*n*)
  - *How can this be? Is there a flaw in the above reasoning?*

Analyzing BuildHeap(): Tight

- To **Heapify()** a subtree takes O(h) time where h is the height of the subtree
  - h = O(lg m), m = # nodes in subtree
  - The height of most subtrees is small
- Fact: an n-element heap has at most $\lceil n/2^{h+1} \rceil$ nodes of height h
- Using this fact, it can be proved that **BuildHeap()** takes O(n) time

Analysis



- **Tighter analysis: O(n)**
  - Assume $n = 2^k - 1$, a complete binary tree. The time required by Heapify when called on a node of height h is O(h).
  - Total cost = $$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) = O(n)$$

  by exercise: $$\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$$

Heapsort

- Given **BuildHeap()**, an in-place sorting algorithm is easily constructed:
  - Maximum element is at A[1]
  - Discard by swapping with element at A[n]
    - Decrement heap_size[A]
    - A[n] now contains correct value
  - Restore heap property at A[1] by calling **Heapify()**
  - Repeat, always swapping A[1] for A[heap_size(A)]

Heapsort

```
HEAPSORT(A)
    BUILD-MAX-HEAP(A)
    for i = A.length downto 2
        exchange A[1] with A[i]
        A.heap-size = A.heap-size − 1
        MAX-HEAPIFY(A, 1)
```

- Build-Max-Heap: $O(n)$
- For loop: n - 1 times
- Exchange element: $O(1)$
- Max-Heapify: $O(\lg n)$

Total time: $O(n \lg n)$

Heapsort

```
Heapsort(A)
{
    BuildHeap(A);
    for (i = length(A) downto 2)
    {
        Swap(A[1], A[i]);
        heap_size(A) -= 1;
        Heapify(A, 1);
    }
}
```

Analyzing Heapsort

- The call to **BuildHeap()** takes O(n) time
- Each of the n - 1 calls to **Heapify()** takes O(lg n) time
- Thus the total time taken by **HeapSort()**
  = O(n) + (n - 1) O(lg n)
  = O(n) + O(n lg n)
  = O(n lg n)

Priority Queues

- Heapsort is a nice algorithm, but in practice Quicksort (next lecture) usually wins
- But the heap data structure is incredibly useful for implementing *priority queues*
  - A data structure for maintaining a set S of elements, each with an associated value or *key*
  - Supports the operations **Insert()**, **Maximum()**, and **ExtractMax()**
  - ***What might a priority queue be useful for?***

Priority Queue Operations

- **Insert(S, x)** inserts the element x into set S
- **Maximum(S)** returns the element of S with the maximum key
- **ExtractMax(S)** removes and returns the element of S with the maximum key
- *How could we implement these operations using a heap?*

O(lg *n*)

Implementing Priority Queues

```
HeapInsert(A, key)    // what's running time?
{
    heap_size[A] ++;
    i = heap_size[A];
    while (i > 1  AND  A[Parent(i)] < key)
    {
        A[i] = A[Parent(i)];
        i = Parent(i);
    }
    A[i] = key;
}
```

O(lg *n*)

Implementing Priority Queues

```
HeapMaximum(A)
{
    // This one is really tricky:

    return A[i];
}
```

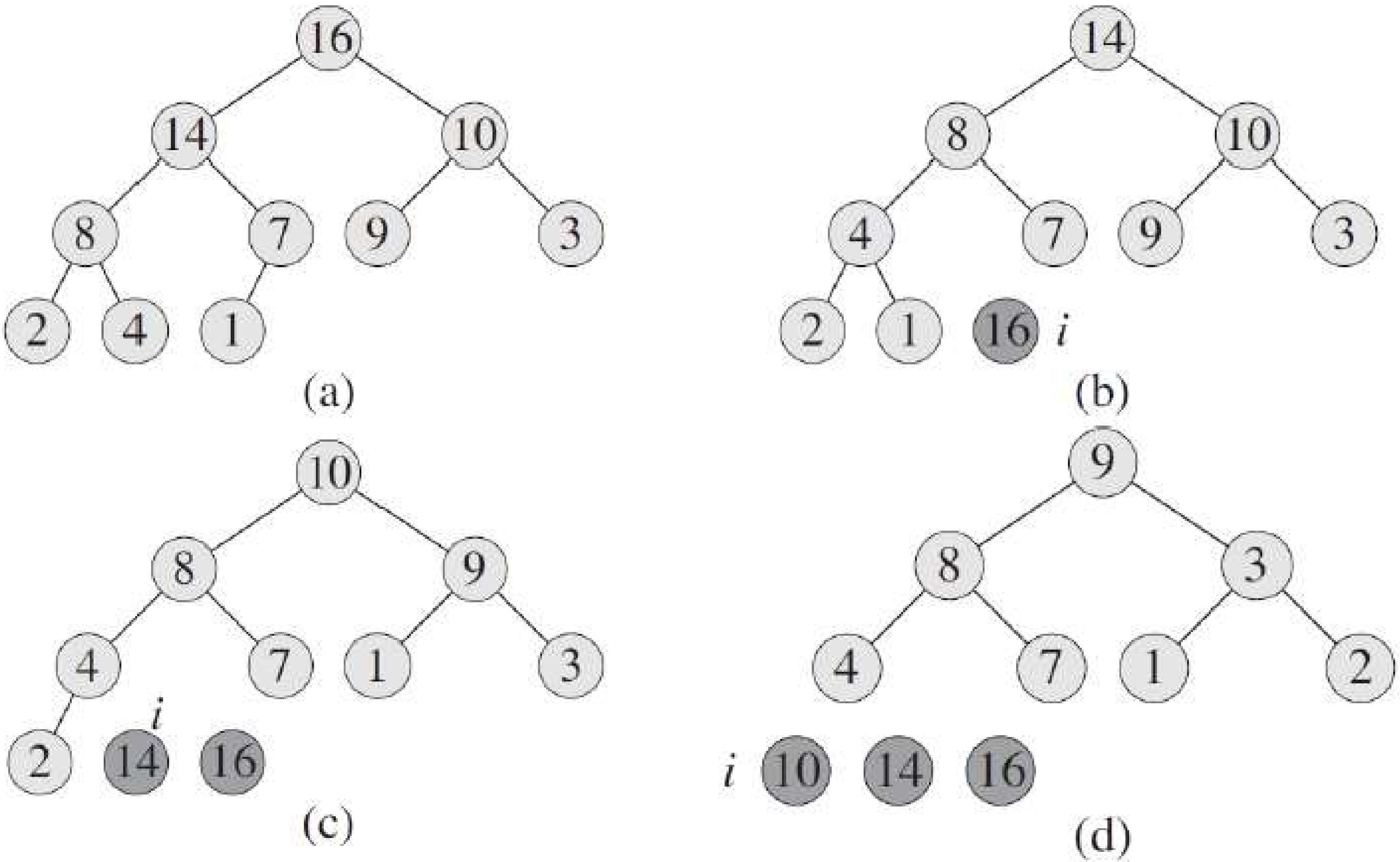
$\Theta(1)$

Implementing Priority Queues

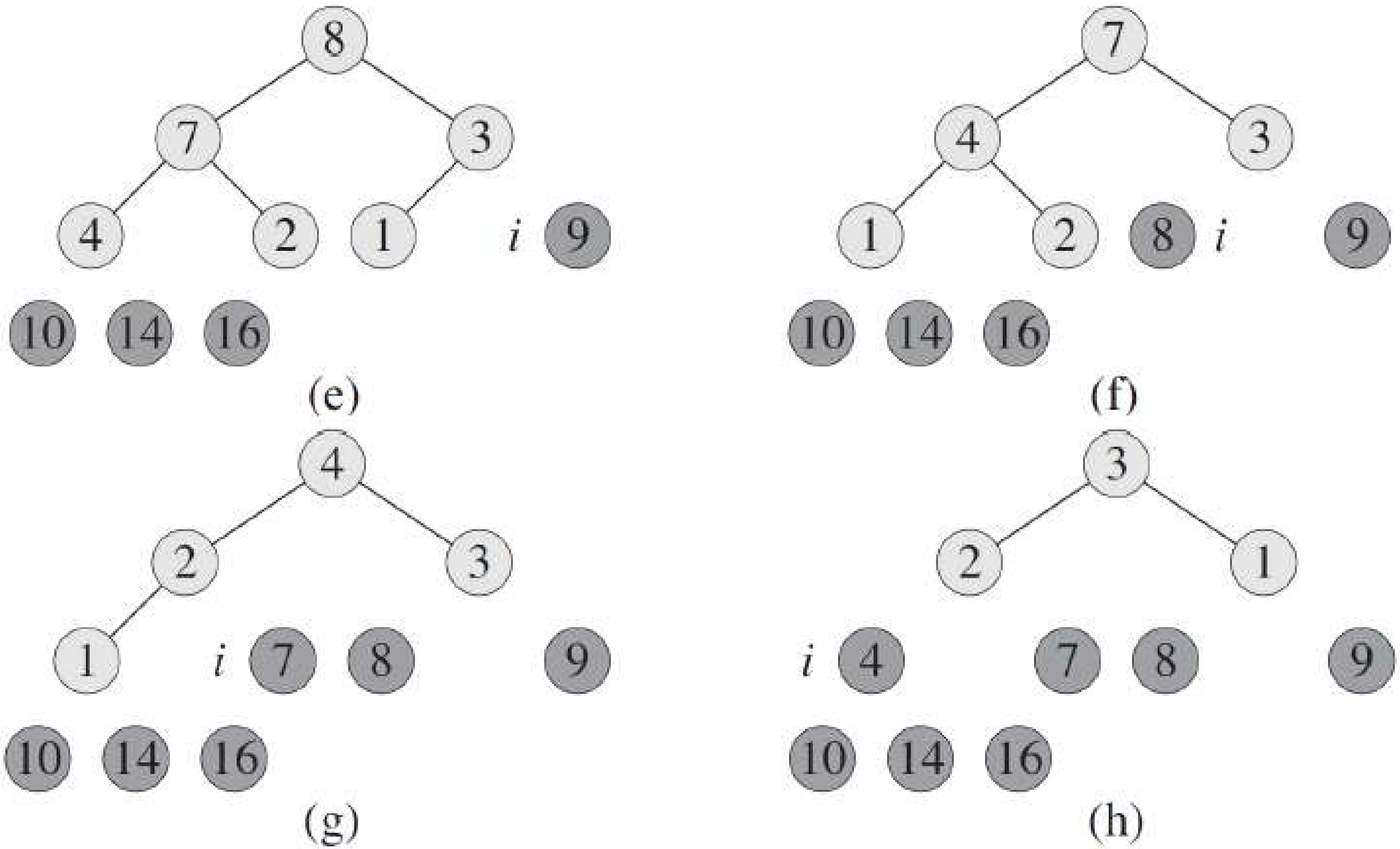
```
HeapExtractMax(A)
{
    if (heap_size[A] < 1) { error; }
    max = A[1];
    A[1] = A[heap_size[A]]
    heap_size[A] --;
    Heapify(A, 1);
    return max;
}
```

It performs only a constant amount of work on top of the **O(lg *n*)** time for HEAPIFY

Example

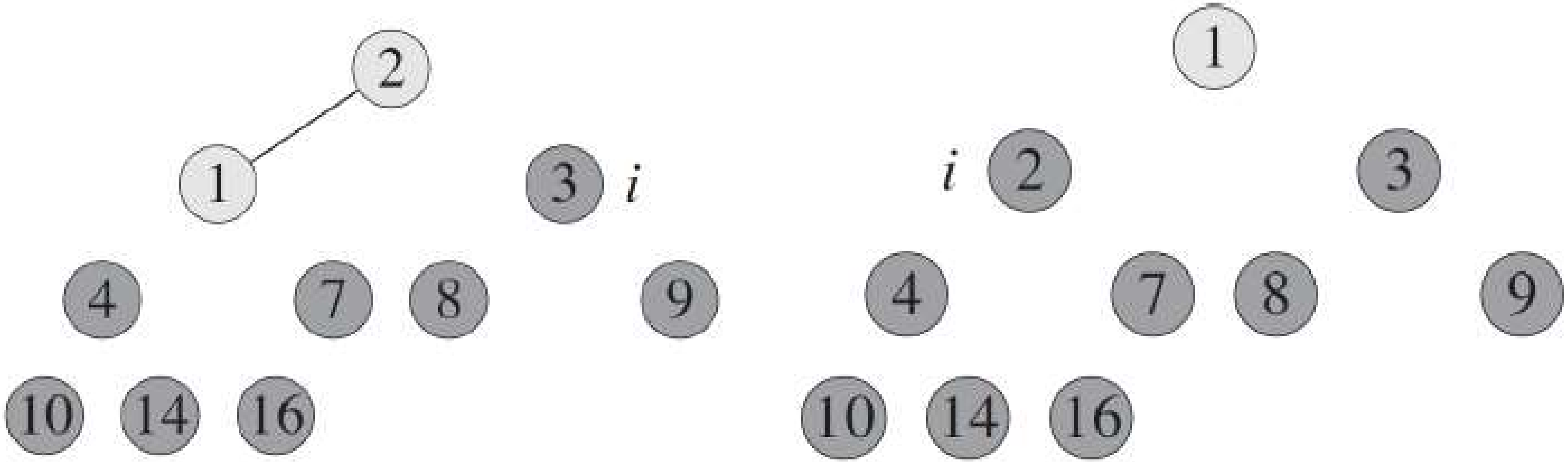

16
14
10
8
7
9
3
2
4
1
(a)
14
8
10
4
7
9
3
2
1
16
i
(b)
10
8
9
4
7
1
3
2
i
14
16
(c)
9
8
3
4
7
1
2
i
10
14
16
(d)

Example



8
7
3
4
2
1
i
9
10
14
16
(e)
7
4
3
1
2
8
i
9
10
14
16
(f)
4
2
3
1
i
7
8
9
10
14
16
(g)
3
2
1
i
4
7
8
9
10
14
16
(h)

Example



(i) (j)

A									
1	2	3	4	7	8	9	10	14	16