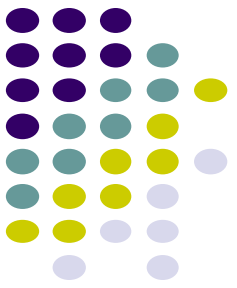


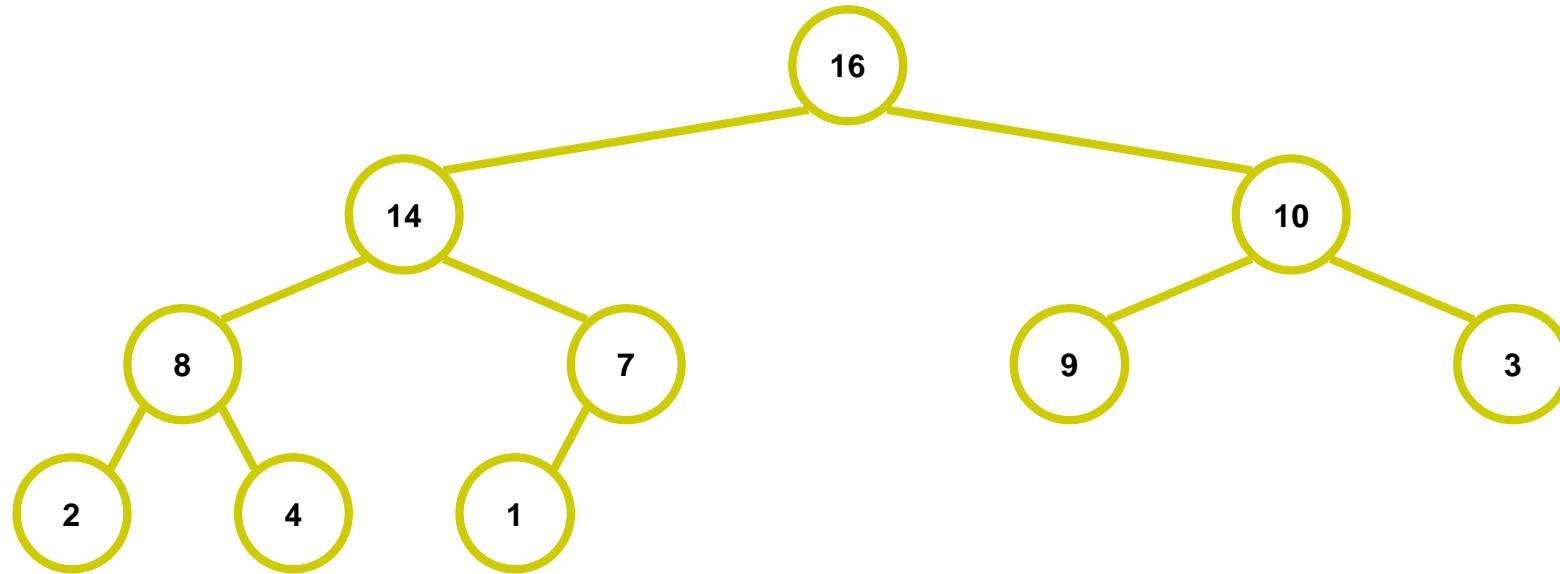
# Lecture # 7

- Binary Heaps
- Heap Sort

# Heaps

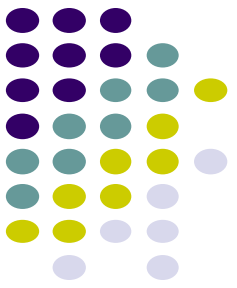


- A *heap* can be seen as a complete binary tree:

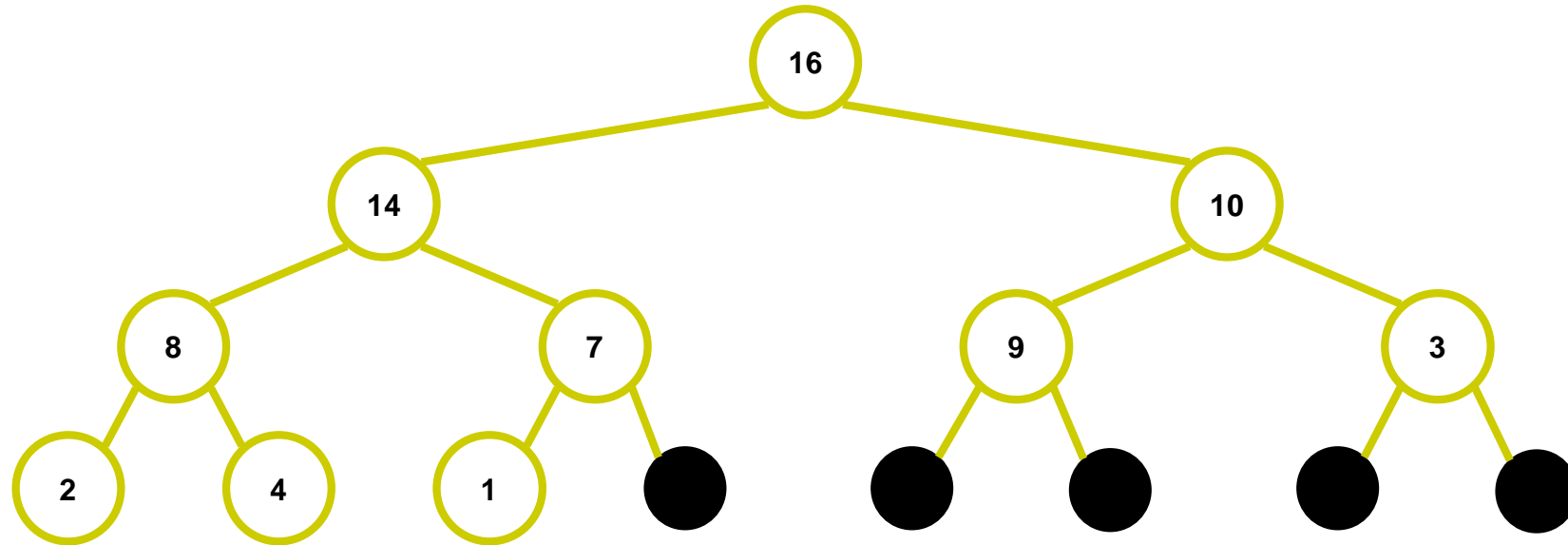


- *What makes a binary tree complete?*
- *Is the example above complete?*

# Heaps

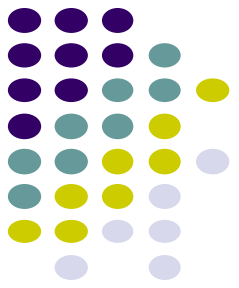


- A *heap* can be seen as a complete binary tree:

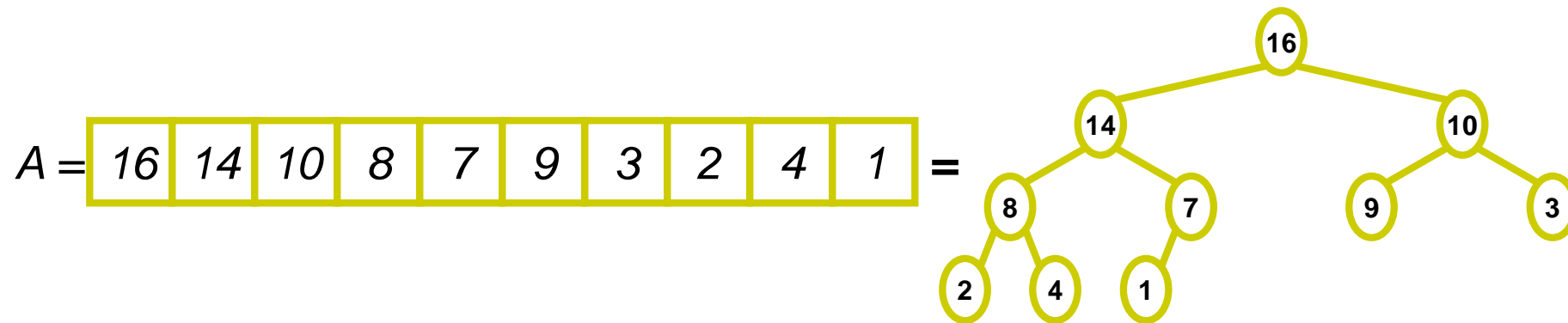


- Some books call them “***nearly complete***” binary trees; can think of unfilled slots as null pointers

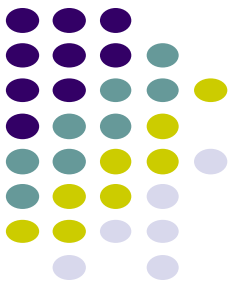
# Heaps



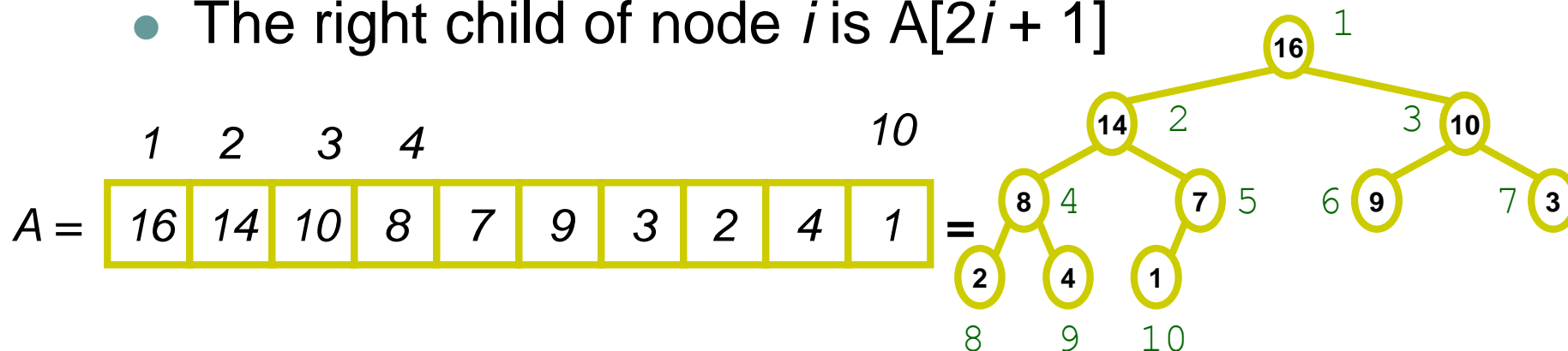
- In practice, heaps are usually implemented as arrays:



# Heaps



- To represent a complete binary tree as an array:
  - The root node is  $A[1]$
  - Node  $i$  is  $A[i]$
  - The parent of node  $i$  is  $A[i/2]$  (note: integer divide)
  - The left child of node  $i$  is  $A[2i]$
  - The right child of node  $i$  is  $A[2i + 1]$



# Referencing Heap Elements

■ So...

Parent(i) { return  $\lfloor i/2 \rfloor$ ; }

Left(i) { return  $2*i$ ; }

right(i) { return  $2*i + 1$ ; }

# The *max-Heap* Property

- *max-heap property says:*

$$A[\text{Parent}(i)] \geq A[i] \quad \text{for all nodes } i > 1$$

- In other words, the value of a node is at most the value of its parent
- Where is the largest element in a heap stored?

- Definitions:

- The *height* of a node in the tree = the number of edges on the longest downward path to a leaf
- The height of a tree = the height of its root

# The *min-Heap* Property

- *min-heap property says:*

$$A[\text{Parent}(i)] \leq A[i] \quad \text{for all nodes } i > 1$$

- Where is the smallest element in a heap stored?

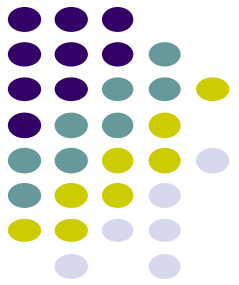


# Heap Height

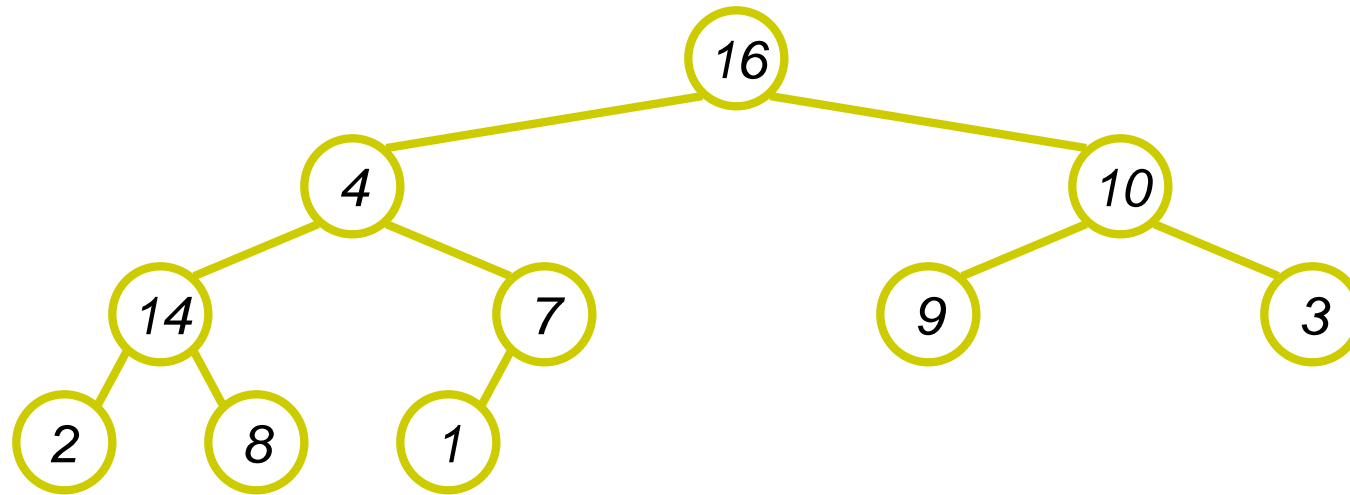
- What is the height of an  $n$ -element heap? Why?
- Basic heap operations take at most time proportional to the height of the heap

# Heap Operations: Max-Heapify(A,i)

- Max-Heapify(): maintain the heap property
  - Its input is an Array A and an index i of array
  - Left and right sub trees of node i are assumed to be max-heaps.
  - **Problem**: The sub tree rooted at i may violate the heap property (*How?*)
  - **Action**: let the value of the parent node “float down” so sub tree at i satisfies the heap property



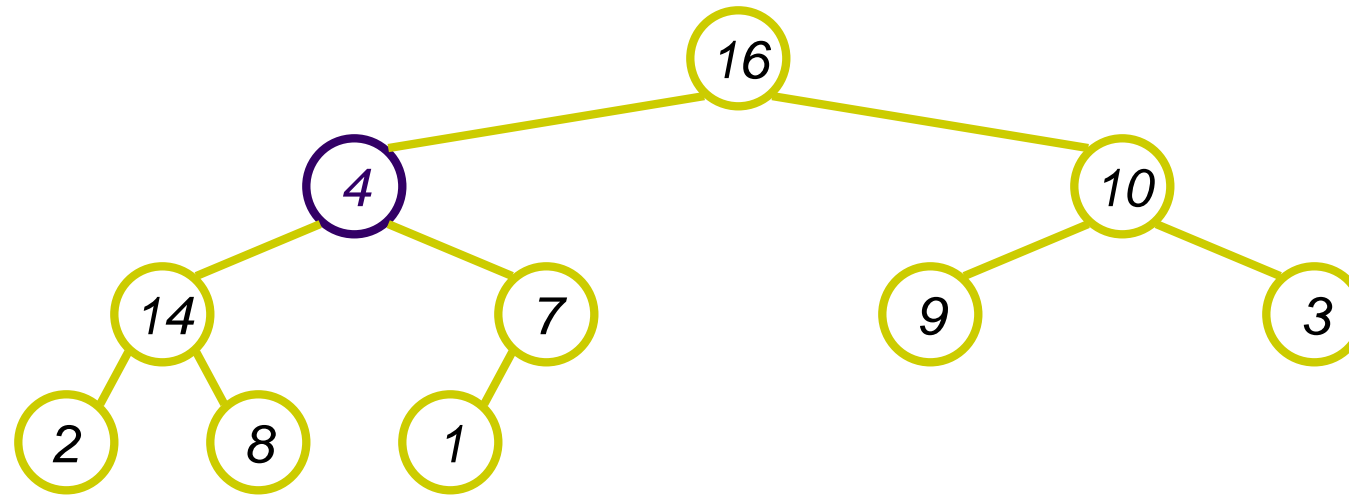
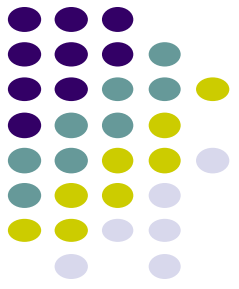
# Max-Heapify( ) Example



$A =$ 

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

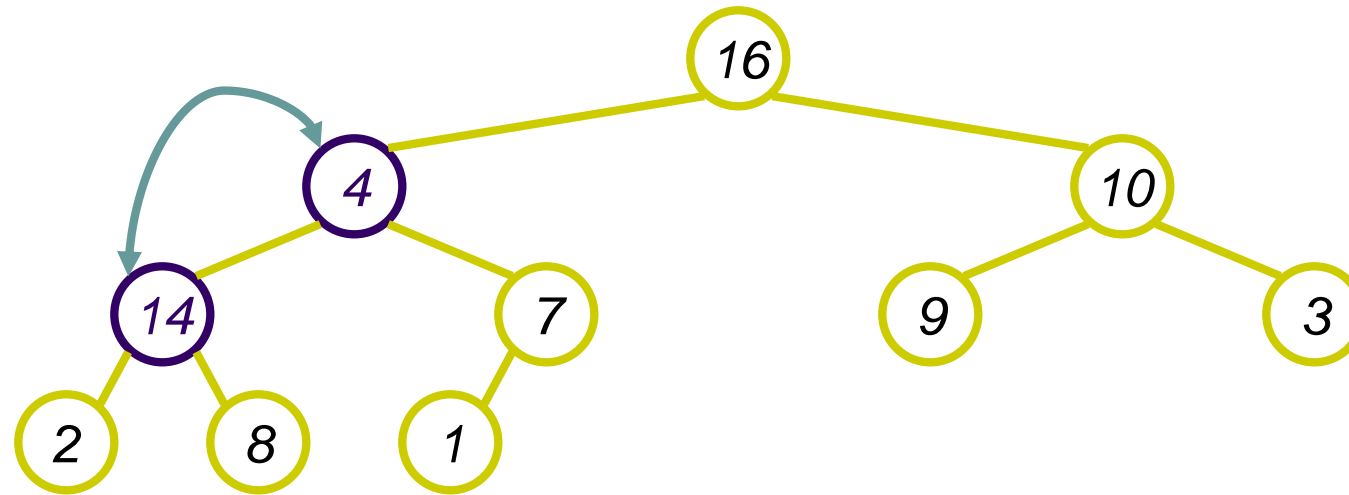
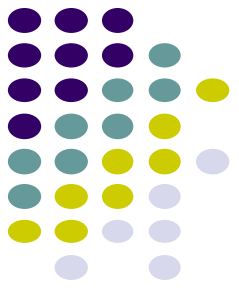
# Heapify() Example



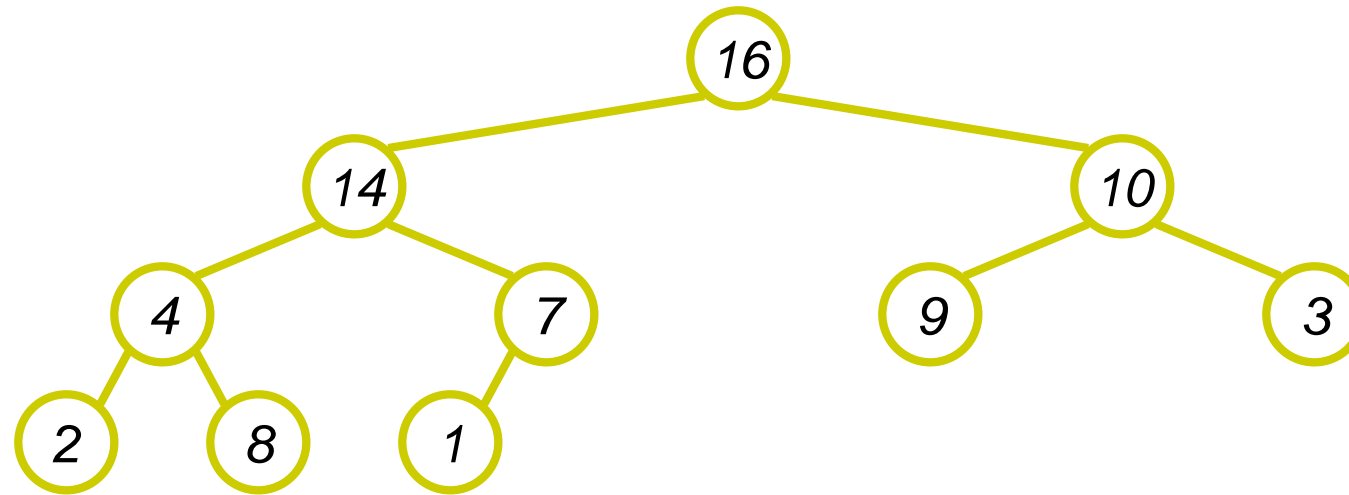
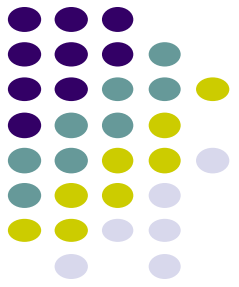
$A =$ 

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

# Heapify() Example



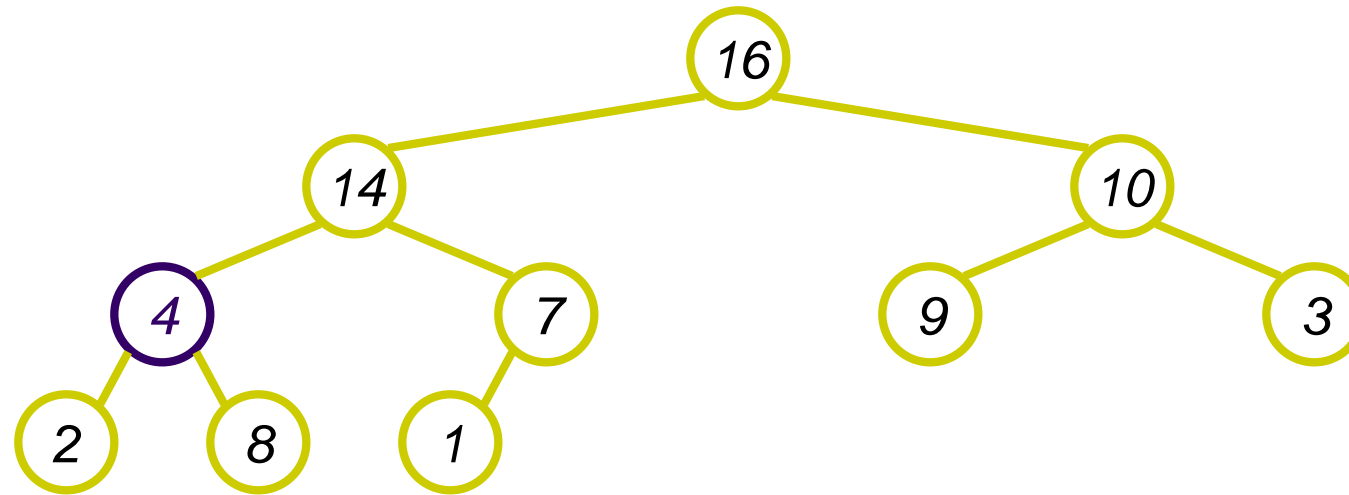
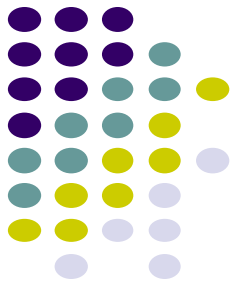
# Heapify() Example



$A =$ 

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

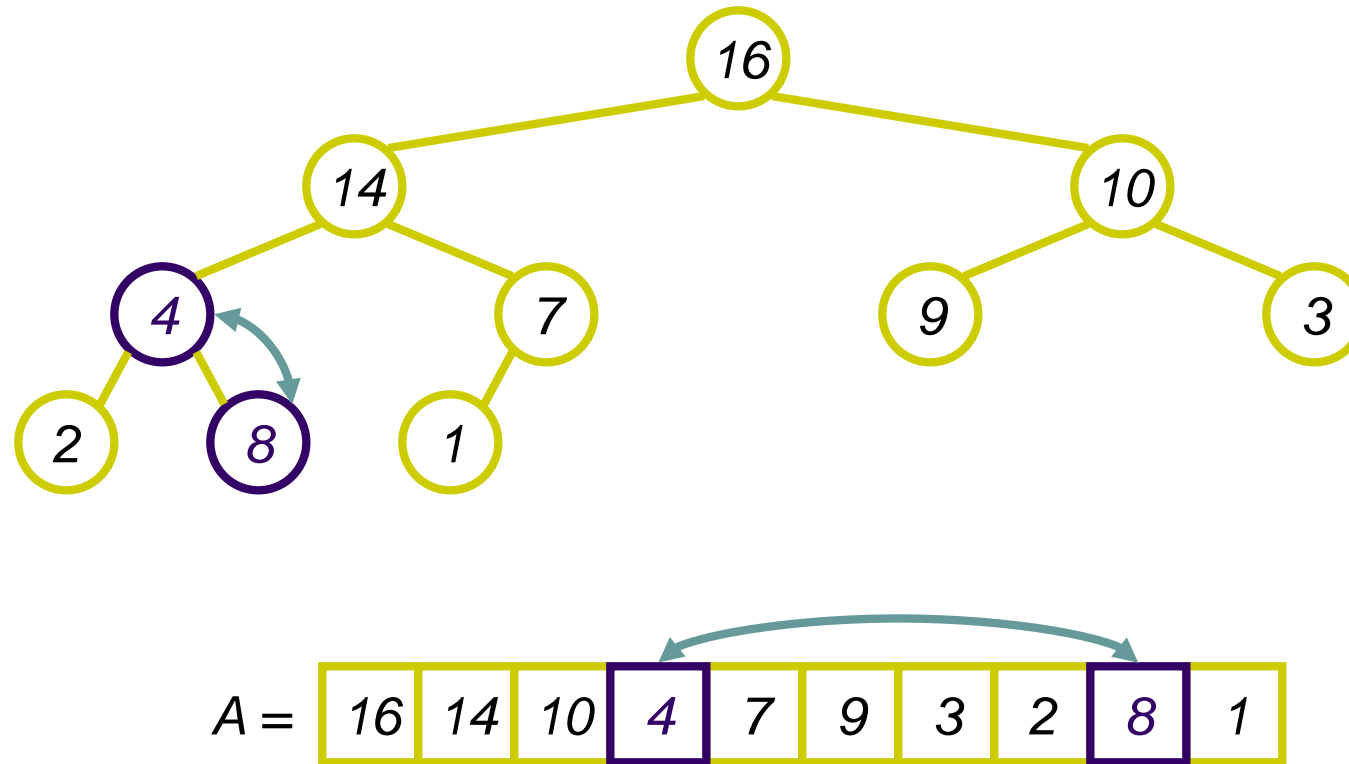
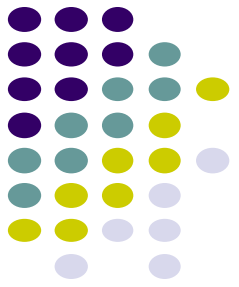
# Heapify() Example



$A =$ 

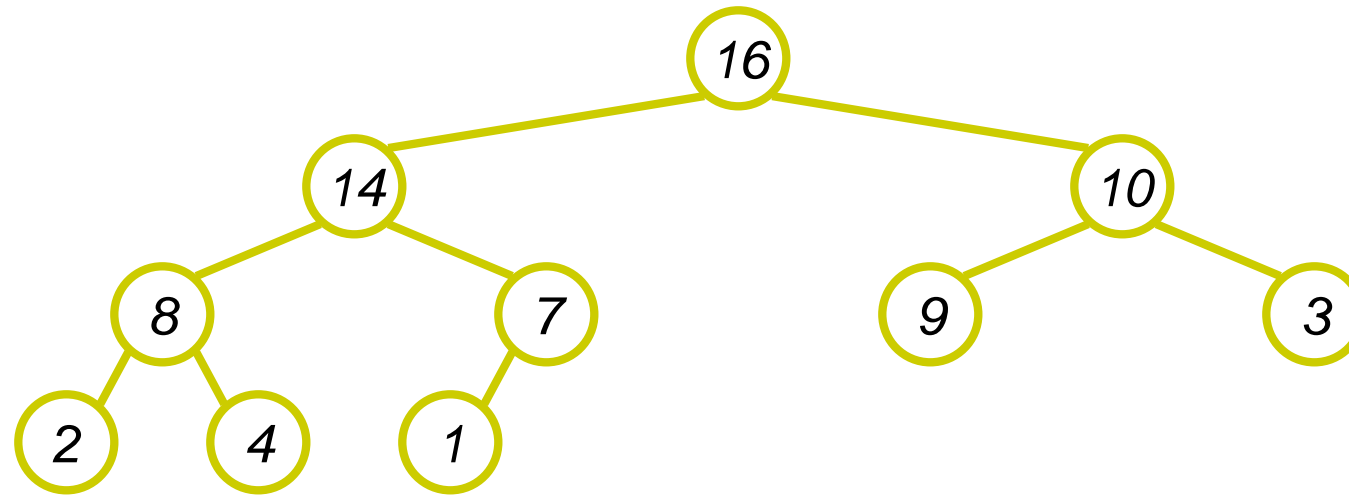
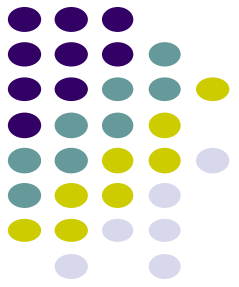
16	14	10	4	7	9	3	2	8	1
----	----	----	---	---	---	---	---	---	---

# Heapify() Example





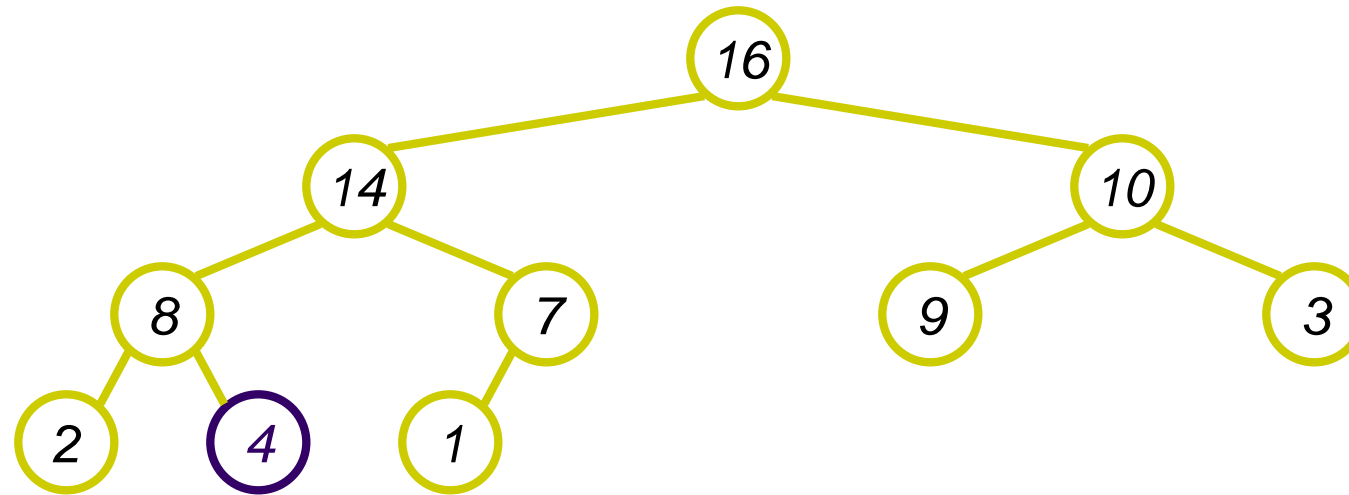
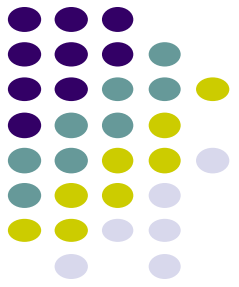
# Heapify() Example



$A =$ 

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

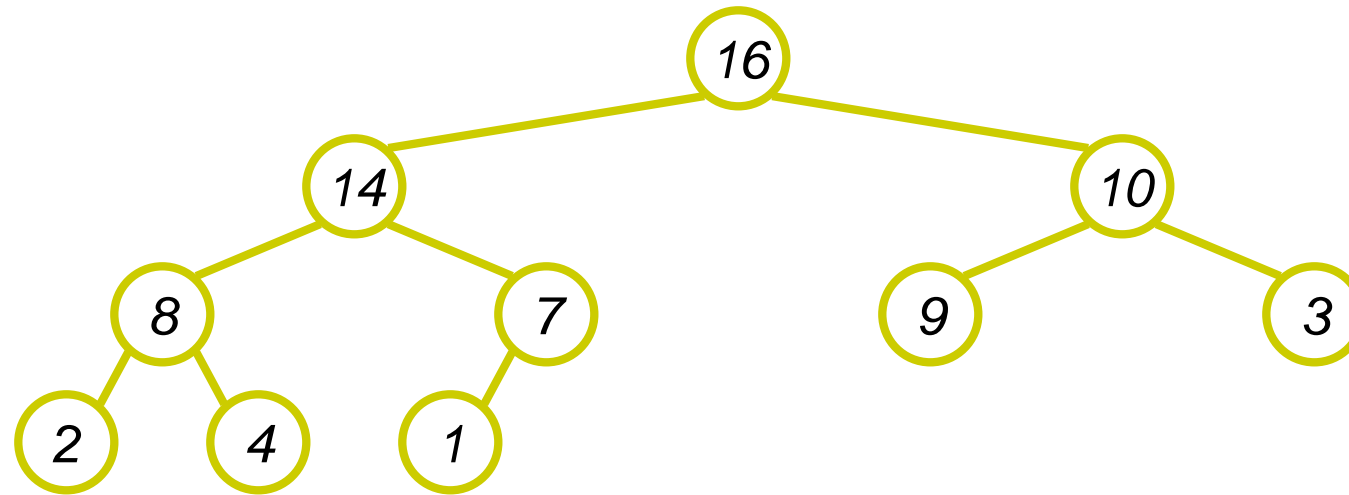
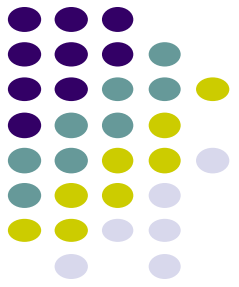
# Heapify() Example



$A =$ 

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

# Heapify() Example



$A =$ 

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

# Analyzing Heapify( )

- The number of nodes in a complete binary tree of height  $h$  is

$$n = 2^0 + 2^1 + 2^2 + \dots + 2^h = \sum_{i=0}^h 2^i = 2^{h+1} - 1$$

- So  $h$  in terms of  $n$  is:

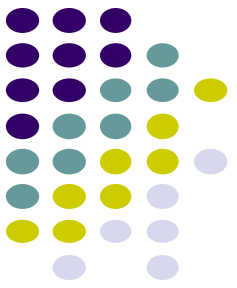
$$h = ( \log(n + 1) ) - 1 \approx \log n \in \Theta(\log n)$$

- Running time of Max-Heapify( ) on a node of height  $h$  will be  $O(\lg n)$  or  $O(h)$ .



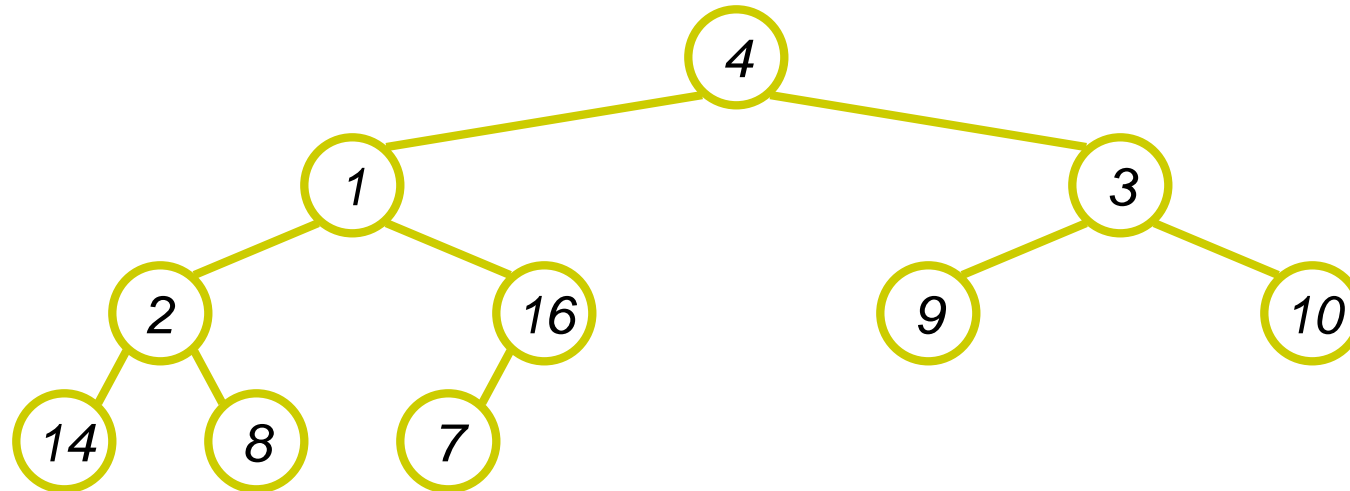
# Heap Operations: BuildHeap()

- 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?*)
  - **Ans:** Because sub array  $A[\lfloor n/2 \rfloor + 1 .. n]$  are all leaves.
  - 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() Example

- Work through example  
 $A = \{4, 1, 3, 2, 16, 9, 10, 14, 8, 7\}$



# BuildHeap( )

- For array of length  $n$ , all elements in range  $A[\lfloor n/2 \rfloor + 1 .. n]$  are leaves of the tree.
- The procedure **BuildHeap( )** goes through the remaining each and every node of the tree and apply **Heapify( )** on every node so that the whole tree is in the form of heap, having all **heap** properties.



# Analyzing BuildHeap( )

- Each call to Max-Heapify() costs  $O(\lg n)$  or  $O(h)$  time, and
- There would be total of  $O(n)$  such calls ( specifically,  $\lfloor n/2 \rfloor$  ) for complete set of nodes in a heap.
- Thus the running time is  $O(n \lg n)$ . This upper bound is correct but not *asymptotically tight*.

# Analyzing BuildHeap(): Tight bound

- The time required by max-heapify() when called on single node of height  $h$  is  $O(h)$ .

- **Fact:** an  $n$ -element heap has height  $\lfloor \lg n \rfloor$  and at most  $\lceil n/2^{h+1} \rceil$  nodes of any height  $h$ .  
( leaves are at height 0 )

- We have the formula:

$$\sum_{k=0}^{\infty} k \cdot x^k = x / (1-x)^2 \text{ for } |x| < 1$$

$$\text{Put } x = 1/2 < 1$$

$$\sum_{k=0}^{\infty} k/2^k = 2$$



- So we can express the total cost of BuildHeap() as follows.

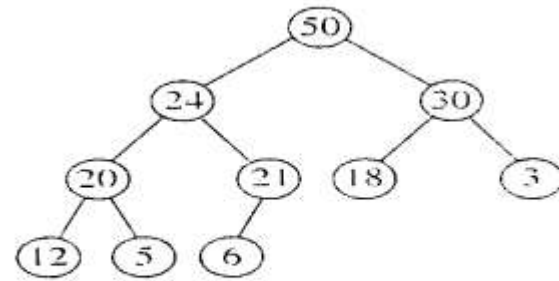
$$\sum_{h=0}^{\lfloor \lg n \rfloor} \lceil n / 2^{h+1} \rceil \cdot O(h) = O \left( n \cdot \sum_{h=0}^{\lfloor \lg n \rfloor} \lceil h / 2^h \rceil \right)$$

As  $\sum_{h=0}^{\infty} h/2^h = 2$

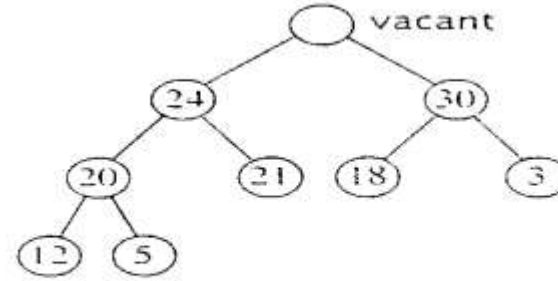
- Thus running time of BuildHeap() can be  $O(n)$ .

# Heapsort

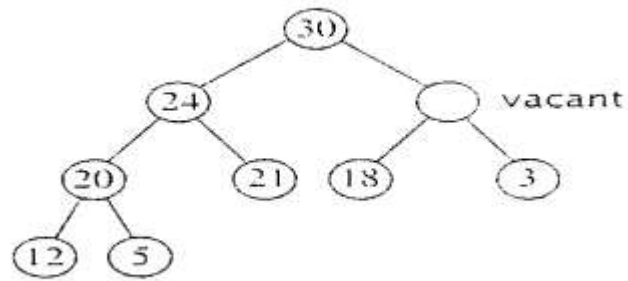
- Given **BuildHeap()**, an in-place sorting algorithm is easily constructed:
  - Maximum element is at **A[1]**
  - Discard by swapping with element at **A[n]**
    - Decrement heap\_size[A]
    - A[n] now contains correct value
  - Restore heap property at A[1] by calling **Heapify()**
  - Repeat,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



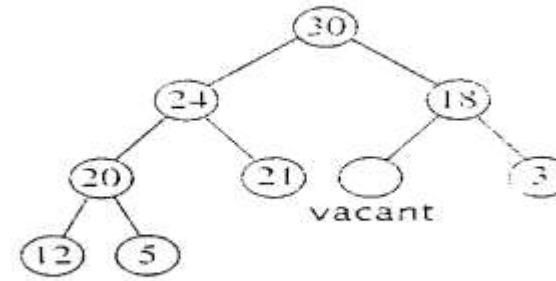
(a) The heap



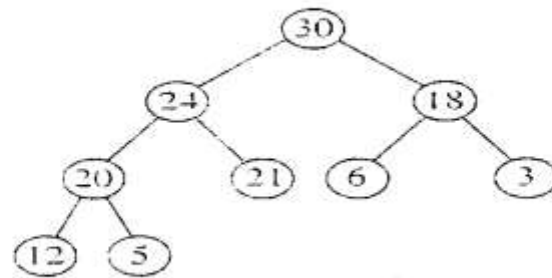
(b) The key at the root has been removed; the rightmost leaf at the bottom level has been removed.  $K = 6$  must be reinserted.



(c) The larger child of vacant, 30, is greater than  $K$ , so it moves up and vacant moves down.

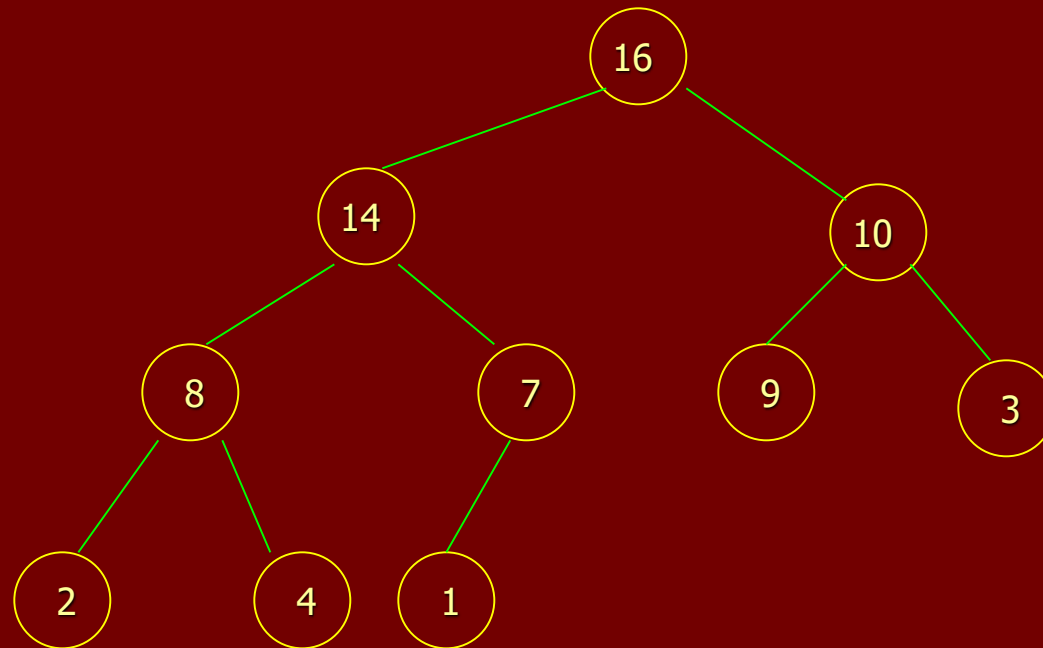


(d) The larger child of vacant, 18, is greater than  $K$ , so it moves up and vacant moves down.



(e) Finally, since vacant is a leaf,  $K = 6$  is inserted.

- View book (Cormen) page number 137 figure 6.4 for better understanding of heap sort mechanism for following Tree.



# 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()**
  - $= O(n) + (n - 1) O(\lg n)$
  - $= O(n) + O(n \lg n)$
  - $= O(n \lg n)$