Reminders and Topics

- **Project 8 due this Friday 4pm**
- Topics of this lecture:
  - **Priority Queue**
  - **The Heap Data Structure**
  - **Heap Implementation**

Priority Queue

- So far we've learned **Queue is a FIFO structure**.
  - Element that is enqueued first is dequeued first.

- In a **Priority Queue**, each element is given a priority (or importance) value.
  - When dequeuing, the element with **the highest priority is dequeued first**.
  - Similarly, peek() returns the highest priority element.

- Examples of Priority Queue:
  - Processes scheduling in a multitask OS.
  - Bandwidth management.

Sorted Array as Priority Queue

- One way to implement Priority Queue is by using a **Sorted Array**.
  - Keep the elements in ascending order, thus the highest priority element is always at the end.
  - Enqueue and dequeue maintain the order.
    - What's the cost of enqueue and dequeue here?

      Enqueue: O(N)

      Dequeue: O(1)

- What if we use unsorted array?

  Enqueue: O(1)

  Dequeue: O(N)

Clicker Question #1

What's the output? Assume higher value means higher priority.

```
PriorityQueue q = new PriorityQueue();
q.enqueue(25);
q.enqueue(35);
System.out.println(q.dequeue());
q.enqueue(15);
System.out.println(q.peek());
q.enqueue(45);
System.out.println(q.dequeue());
System.out.println(q.dequeue());
```

(a) 35 25 45 25

(b) 25 35 45 35

(c) 35 45 25 15

(d) 25 35 45 15

(e) 35 15 45 25

Answer on next slide

Clicker Question #1

What's the output? Assume higher value means higher priority.

```
PriorityQueue q = new PriorityQueue();
q.enqueue(25);
q.enqueue(35);
System.out.println(q.dequeue());
q.enqueue(15);
System.out.println(q.peek());
q.enqueue(45);
System.out.println(q.dequeue());
System.out.println(q.dequeue());
```

(a) 35 25 45 25

(b) 25 35 45 35

(c) 35 45 25 15

(d) 25 35 45 15

(e) 35 15 45 25

Priority Queue Interface

```
public interface PriorityQueueInterface<T extends
                                        Comparable<T>> {
  boolean isEmpty();
  boolean isFull();
  T peek();

  void enqueue(T element);
  // Throws PriorityQueueOverflowException if full;
  // otherwise, adds element to this priority queue.

  T dequeue();
  // Throws PriorityQueueUnderflowException if empty;
  // otherwise, removes element with highest priority
  // and returns it.
}
```

BST as Priority Queue

- You can certainly also implement Priority Queue by using a **Binary Search Tree (BST)**.
  - If the BST is balanced at all times, the enqueue and dequeue (equivalent to inserting a new node and removing the max from the tree) are both O(log N)
  - However, this requires self-balancing trees.

- Both BST and Sorted Array enforce strong ordering. If we are only concerned with finding and removing the largest element, it's sufficient to keep the elements **weakly ordered**.

Heaps

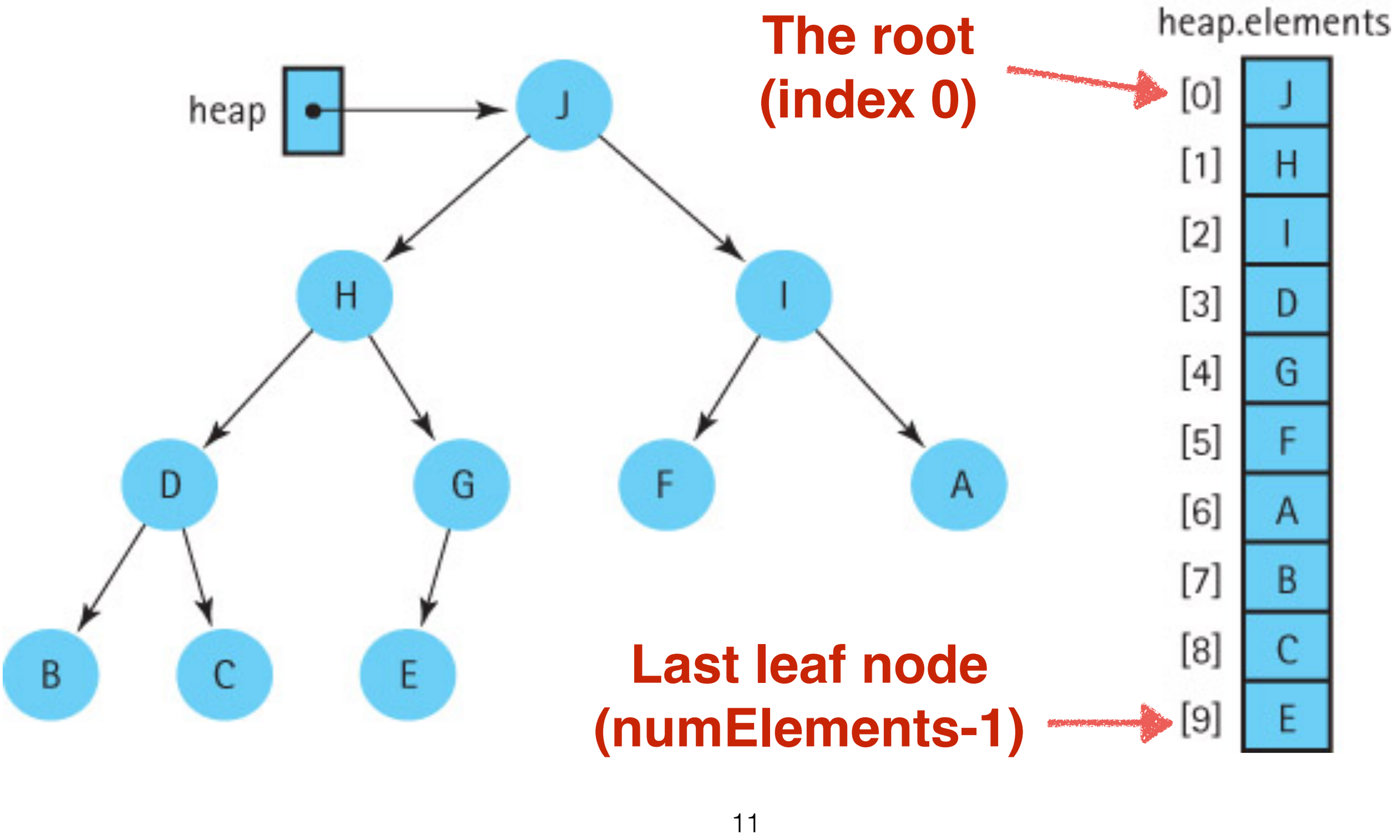
- **A Heap is a binary tree** with the following properties:
  - **Shape property**: it's a complete binary tree.
  - **Order property**: the value of every node is greater than or equal to (>=) its children's values.
    - This property defines max-heap. You can similarly define min-heap.

- In a **complete** binary tree, every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible.

Heaps

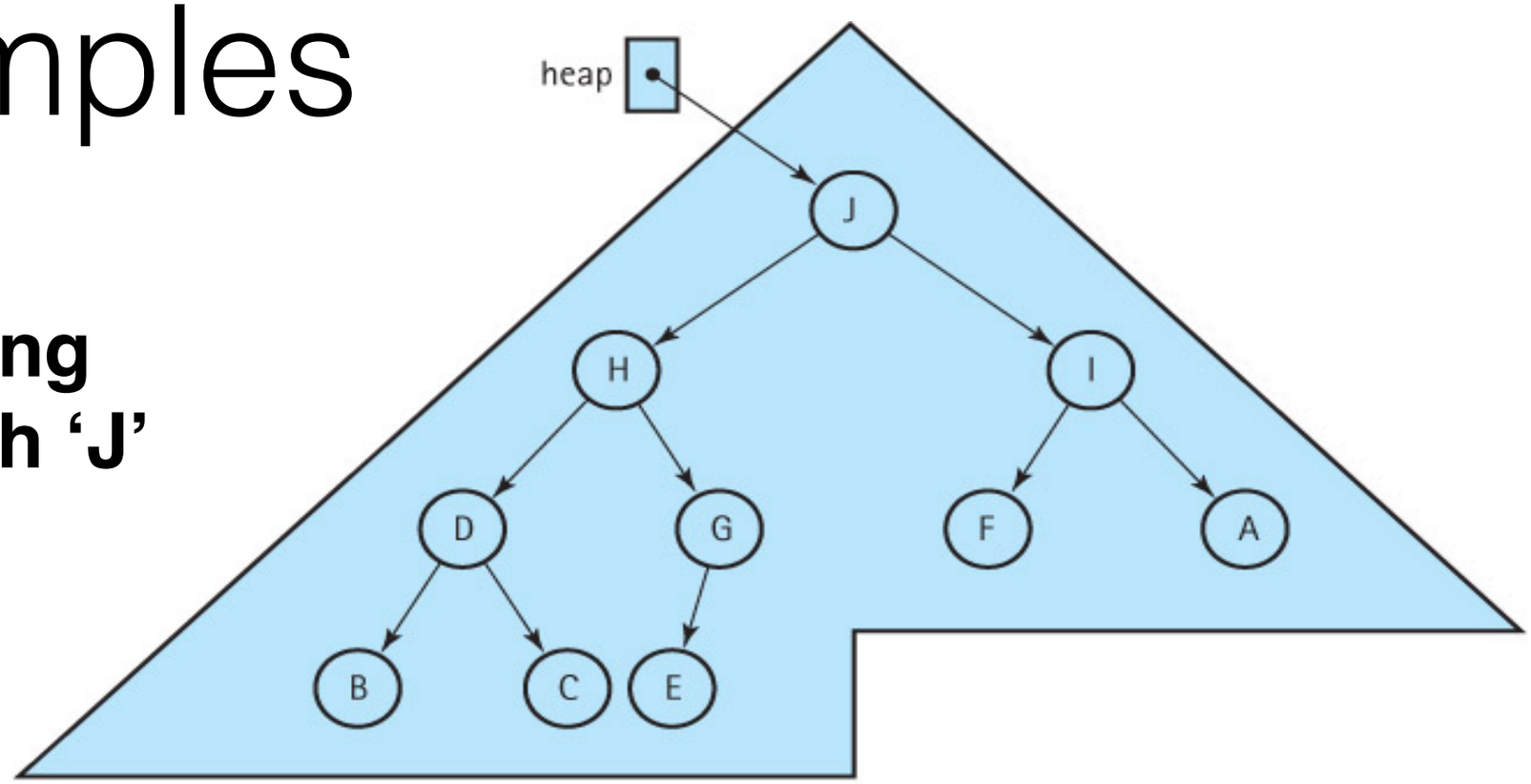
- A heap is **often stored as an array** (recall last lecture).
  - As a heap is a complete binary tree, the array representation has no holes/gaps. So it's very efficient.
  - As we've learned, for any node at index `i`
    - The children are at indices `(2*i+1)` and `(2*i+2)`
    - The parent is at `(i-1)/2`

Heap Stored as Array



Heap Examples

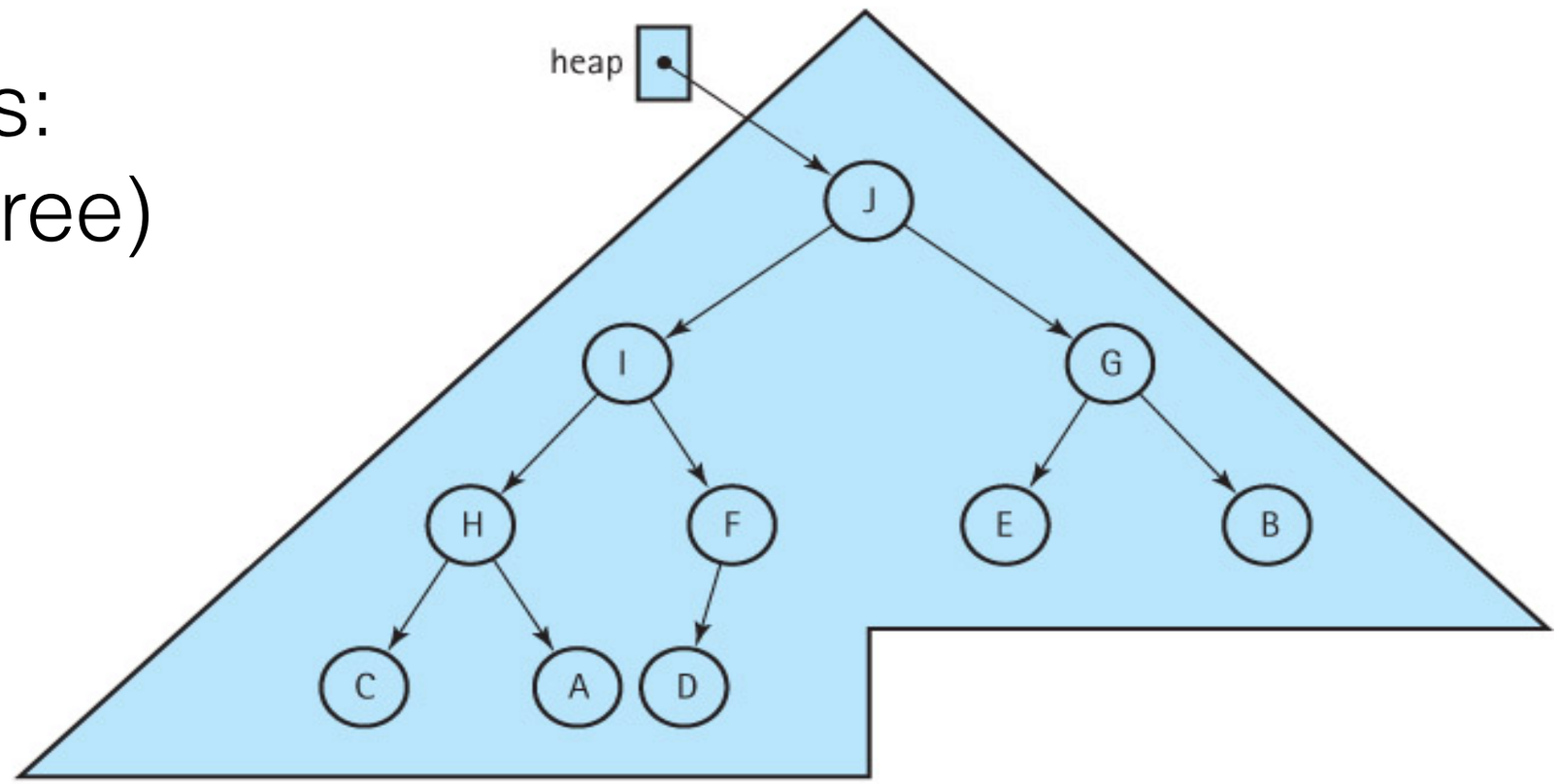
Two heaps containing the letters ‘A’ through ‘J’



(a)

Verify both properties:

- Shape (complete tree)
- Order



(b)

Heaps vs. BSTs

- The order property in a heap is different from a BST:
  - BST is strongly ordering (node's value is in between):

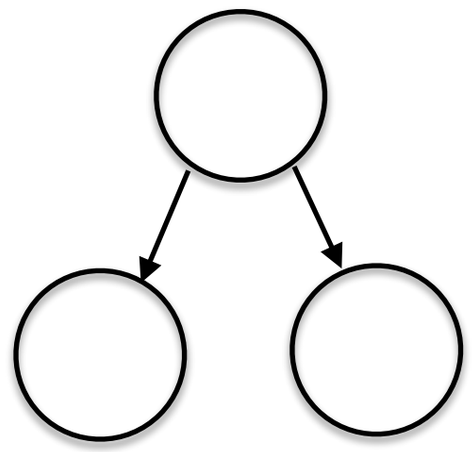
    `left subtree <= node < right subtree`

  - Heap is weakly ordered:

    `(node >= left child) && (node >= right child)`

    node's value is the largest among the three

- Pop quiz: is it possible for a 3-node tree shown on the right to be simultaneously a heap and a BST?



No! Heap requires root >= right. BST requires root < right.

Heaps vs. BSTs

- The heap's order property implies:
  - **The largest value is always at the root**
  - At a node, there is no particular ordering of its two children nodes. Therefore you can't search for an element using the BST search method.
  - Similarly, there is no easy way (other than traversing the entire tree) to find the smallest element in a max-heap.
- This is why we say it's **weakly ordered**. But it's just enough to support priority queue operations very efficiently.

Clicker Question #2

Which nodes in a heap could possibly contain the **third-largest** element in the heap, assuming that no two elements have the same value?

(a) anywhere except the root

(b) anywhere in level 1

(c) anywhere in level 1 or 2

(d) anywhere in level 1, 2, or 3.

(d) in level 1 or 2, or in the left half of level 3

Answer on next slide

Clicker Question #2

Which nodes in a heap could possibly contain the **third-largest** element in the heap, assuming that no two elements have the same value?

(a) anywhere except the root

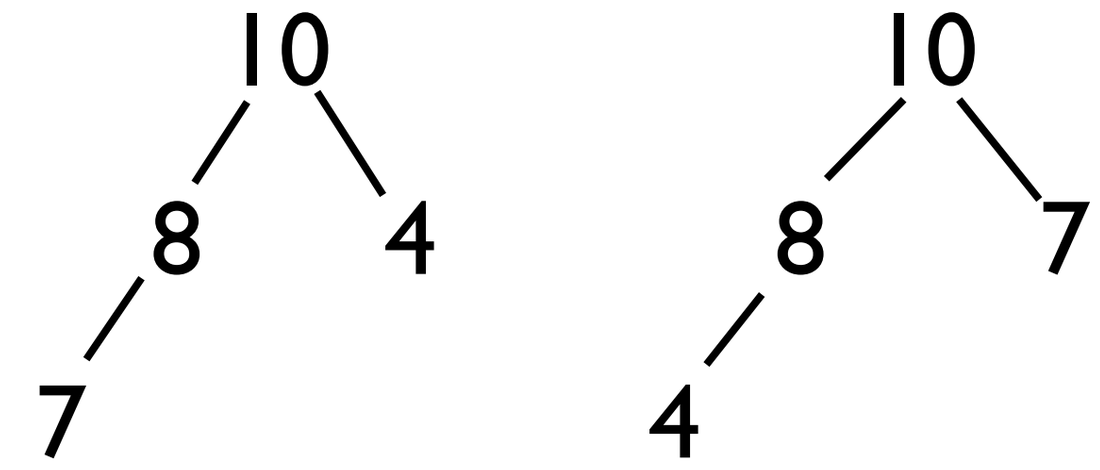
(b) anywhere in level 1

(c) anywhere in level 1 or 2

(d) anywhere in level 1, 2, or 3.

(d) in level 1 or 2, or in the left half of level 3

Clicker #2 Examples

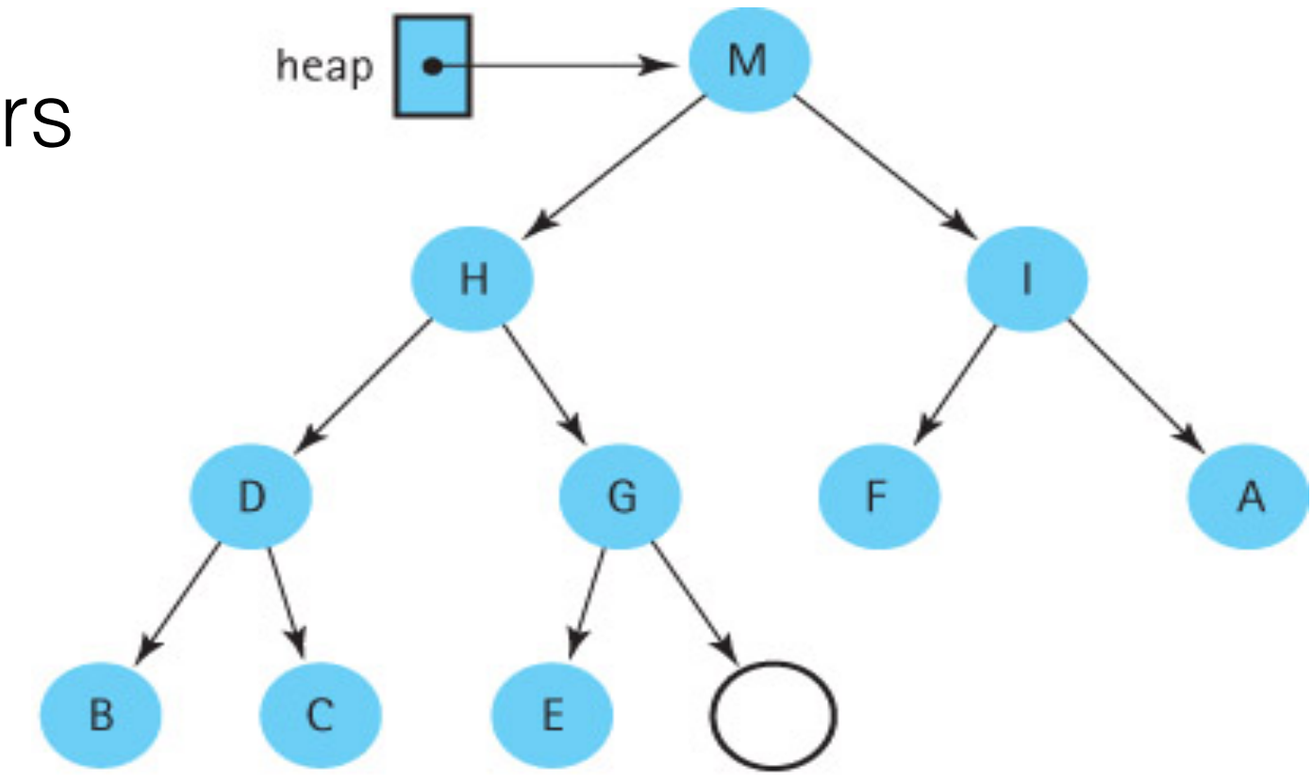


Level 2 Level 1

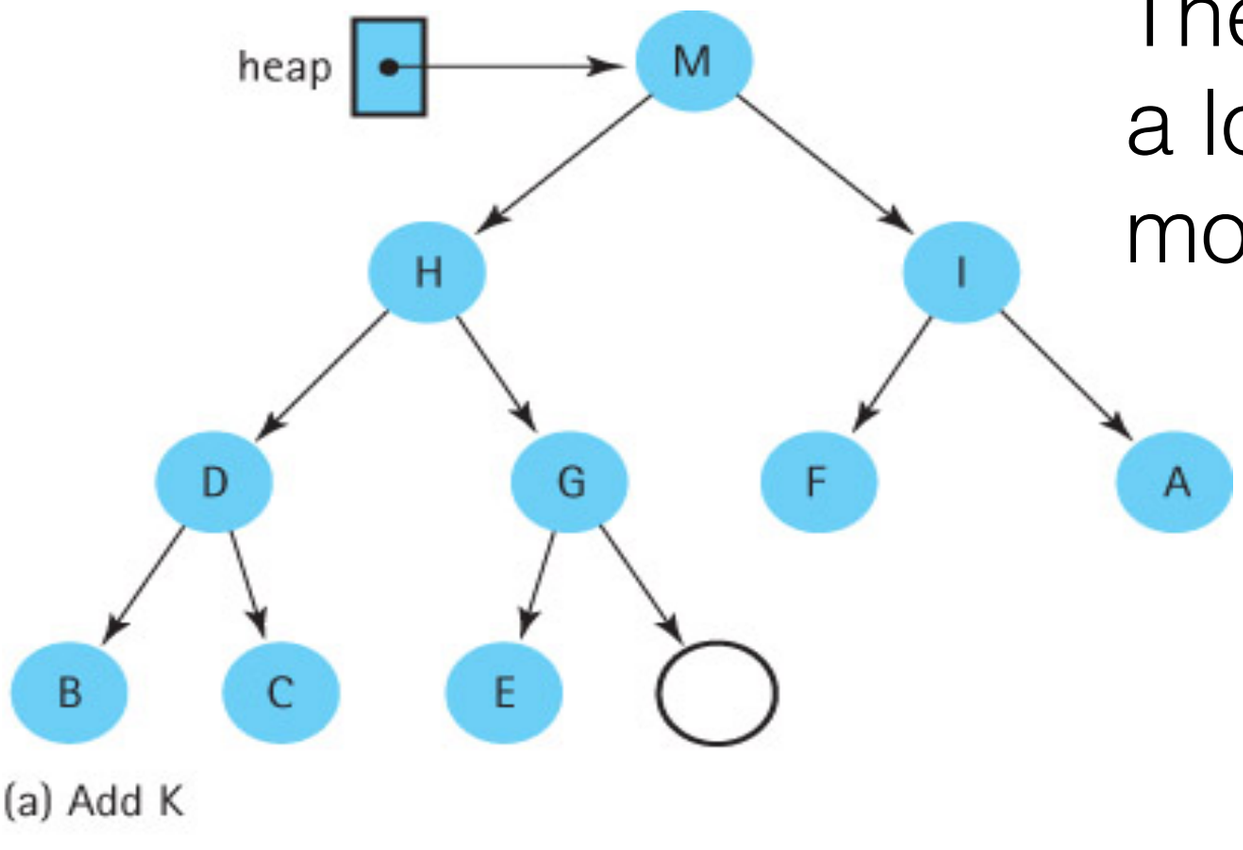
Any node on one of these levels could be third-largest.

Heap-based Priority Queue

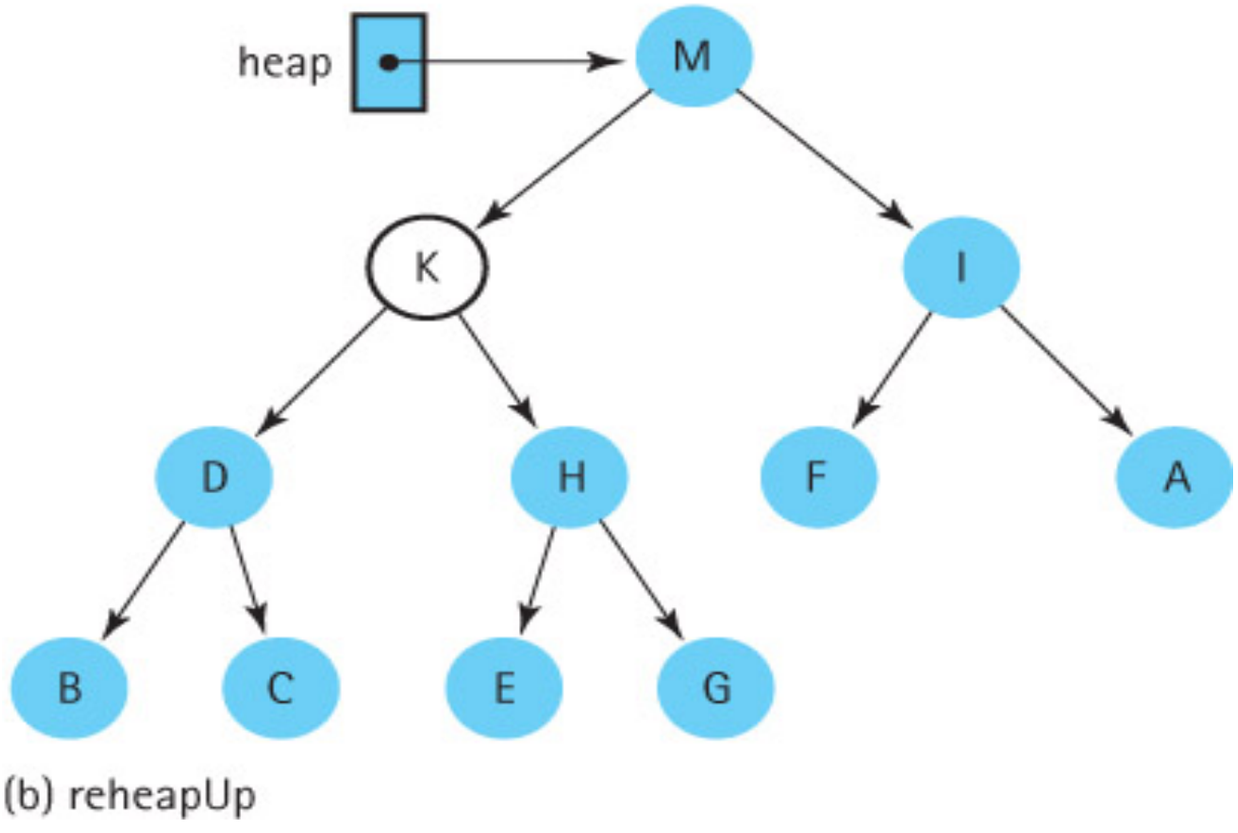
- **Enqueue**
  - To ensure complete tree property, we insert the new element at the next spot in the tree (simply the end of the array in our array representation of the heap).
  - To ensure order property, we '**bubble up**' the new element until it encounters a parent that's no longer smaller than it.
- The 'bubble up' is a.k.a. **reheapUp**



(a) Add K



(a) Add K



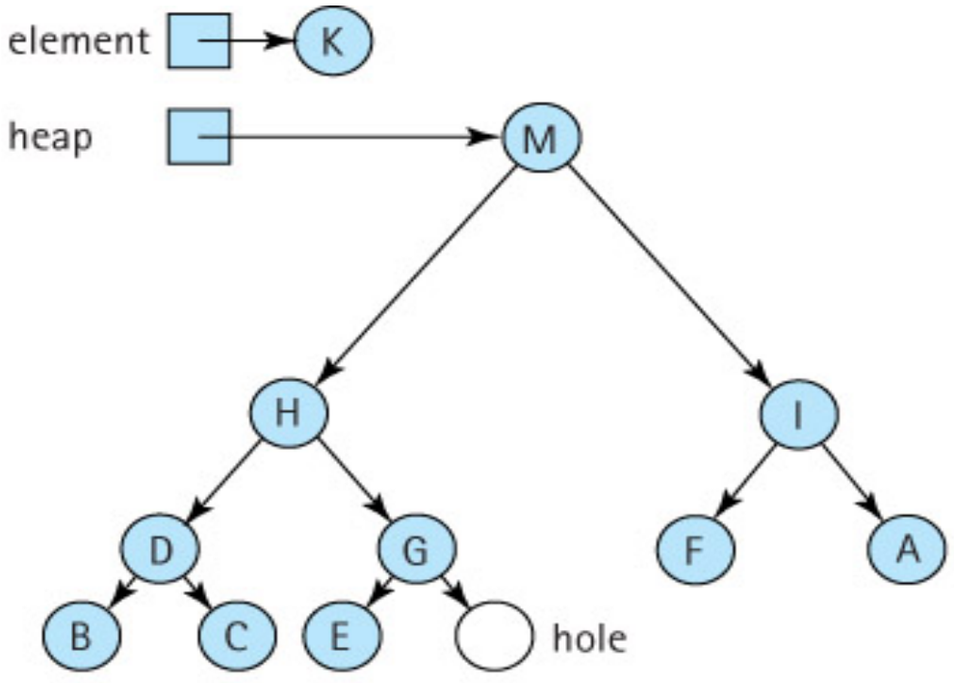
(b) reheapUp

The ‘bubble up’ normally involves a lot of ‘swapping’. To make it more efficient, here is what we do:

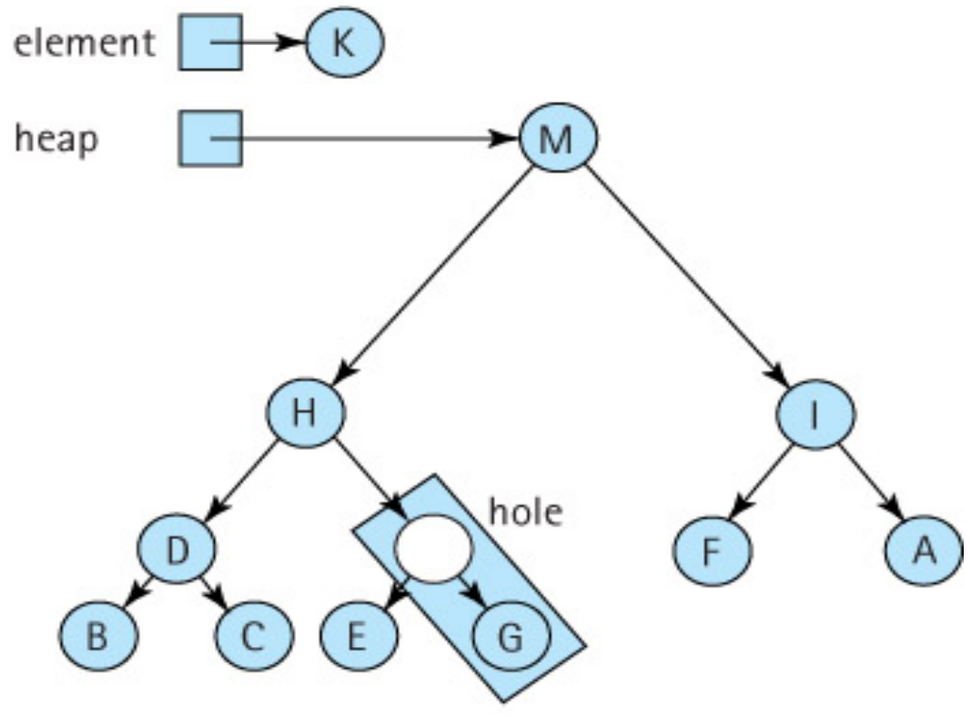
(a) To enqueue ‘K’, we create a ‘hole’ in the bottom right-most position in the tree (i.e. end of array).

(b) Next, we bubble up the hole until we reach the place in the tree where the new element satisfies the order property

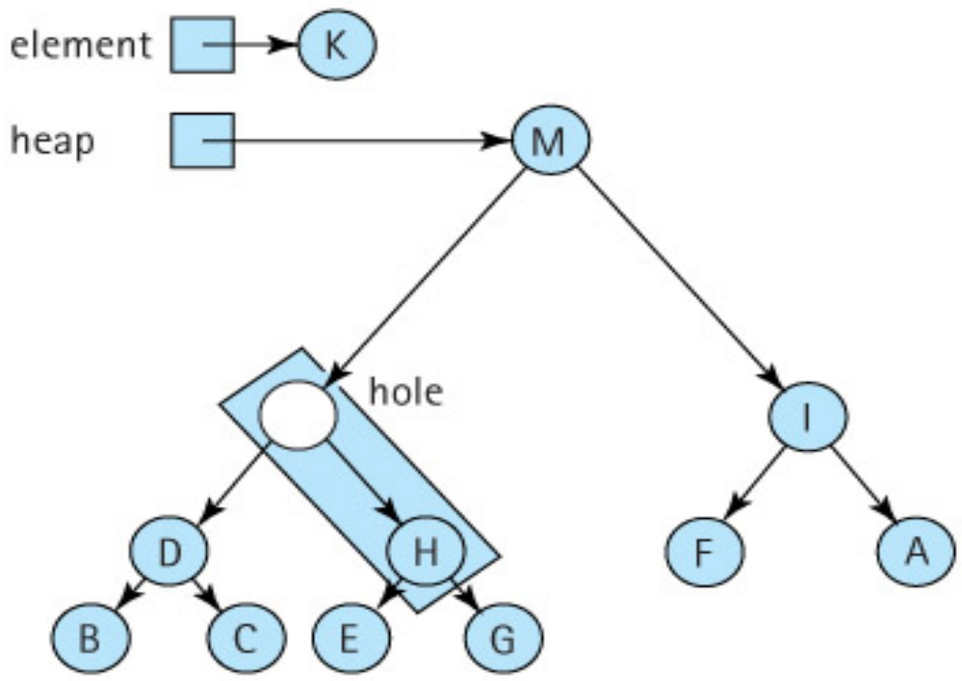
(c) Copy the new element there.



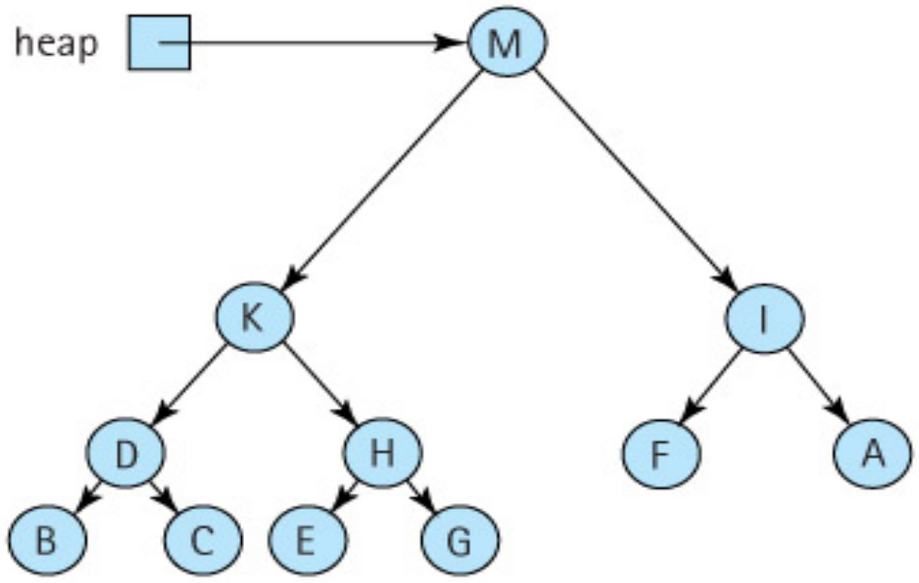
(a) Add K



(b) Move hole up



(c) Move hole up



(d) Place element into hole

reheapUp

```
// assume elements are stored in array ‘heap’
// last element is at index lastIndex
// recall that node i’s parent is at (i-1)/2
private void reheapUp (T elem) {
  int hole = lastIndex;
  int parent = (hole-1)/2;
  while ( (hole > 0) &&
          elem.compareTo(heap[parent]) > 0 ) {
    heap[hole] = heap[parent];
    hole = parent;
    parent = (parent-1)/2;
  }
  heap[hole] = elem;
}
```

Clicker Question #3

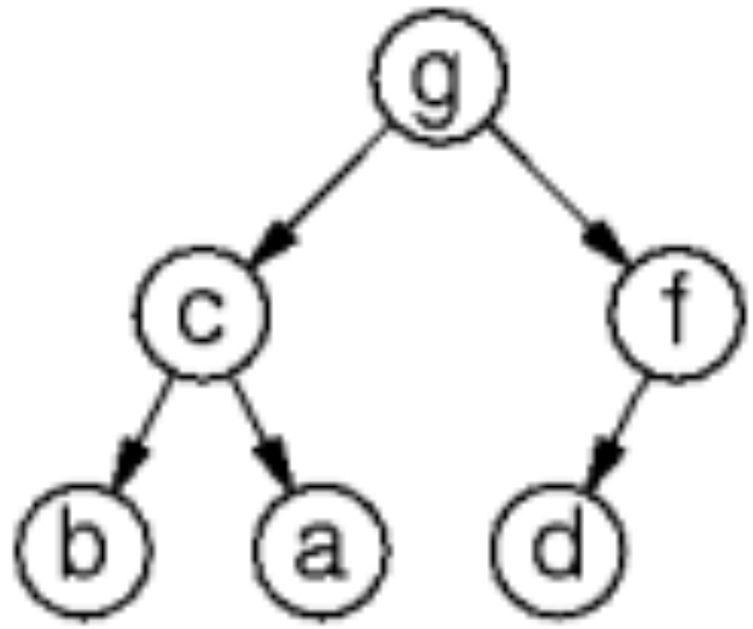
If I add the value “k” to this heap, what does the heap look like afterwards?

a) g c f b a d k

b) k c g b a d f

c) g c k b a d f

d) k c f b a d g



Answer on next slide

Clicker Question #3

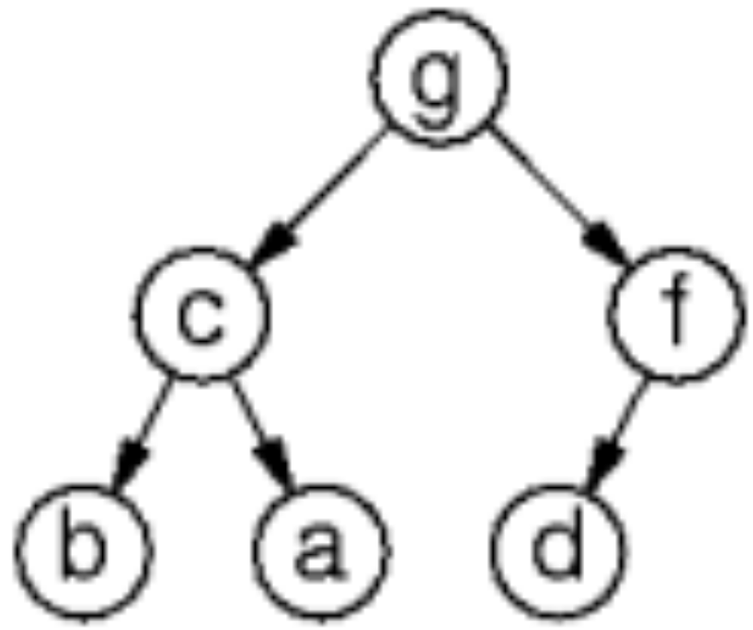
If I add the value “k” to this heap, what does the heap look like afterwards?

a) g c f b a d k

b) k c g b a d f

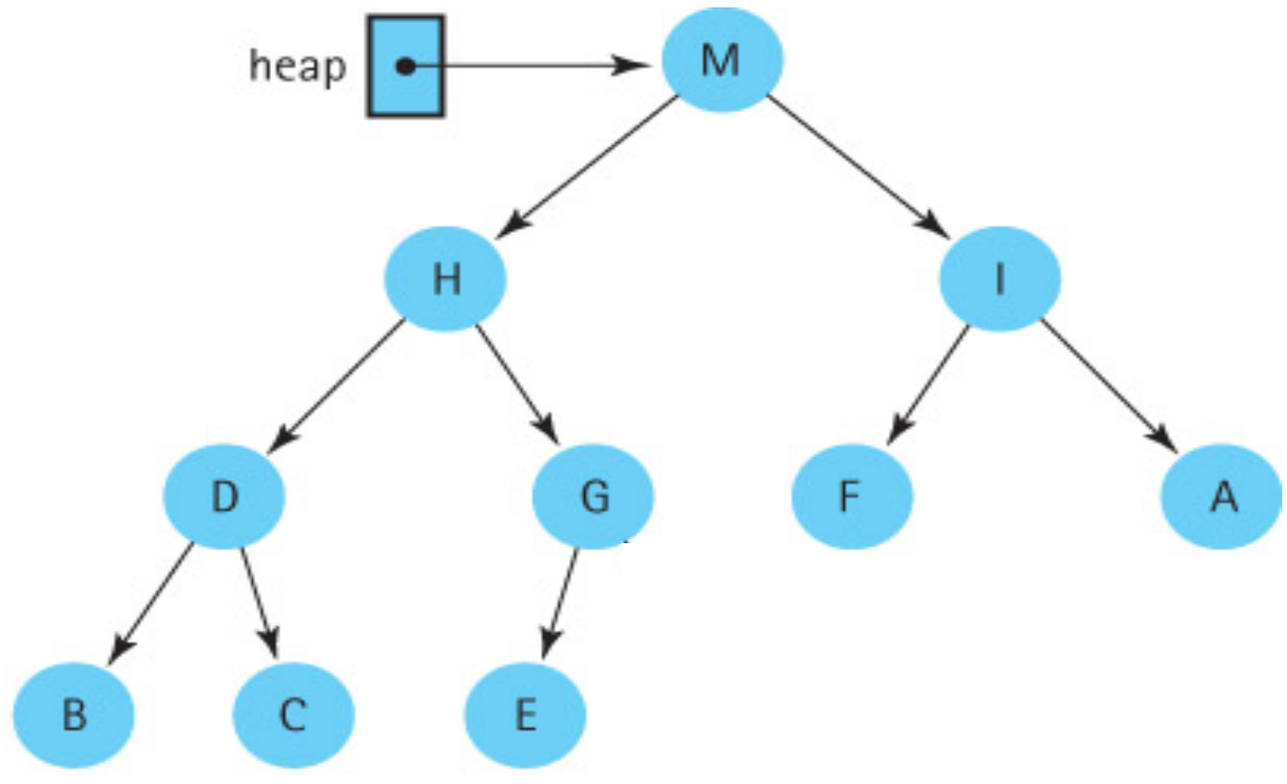
c) g c k b a d f

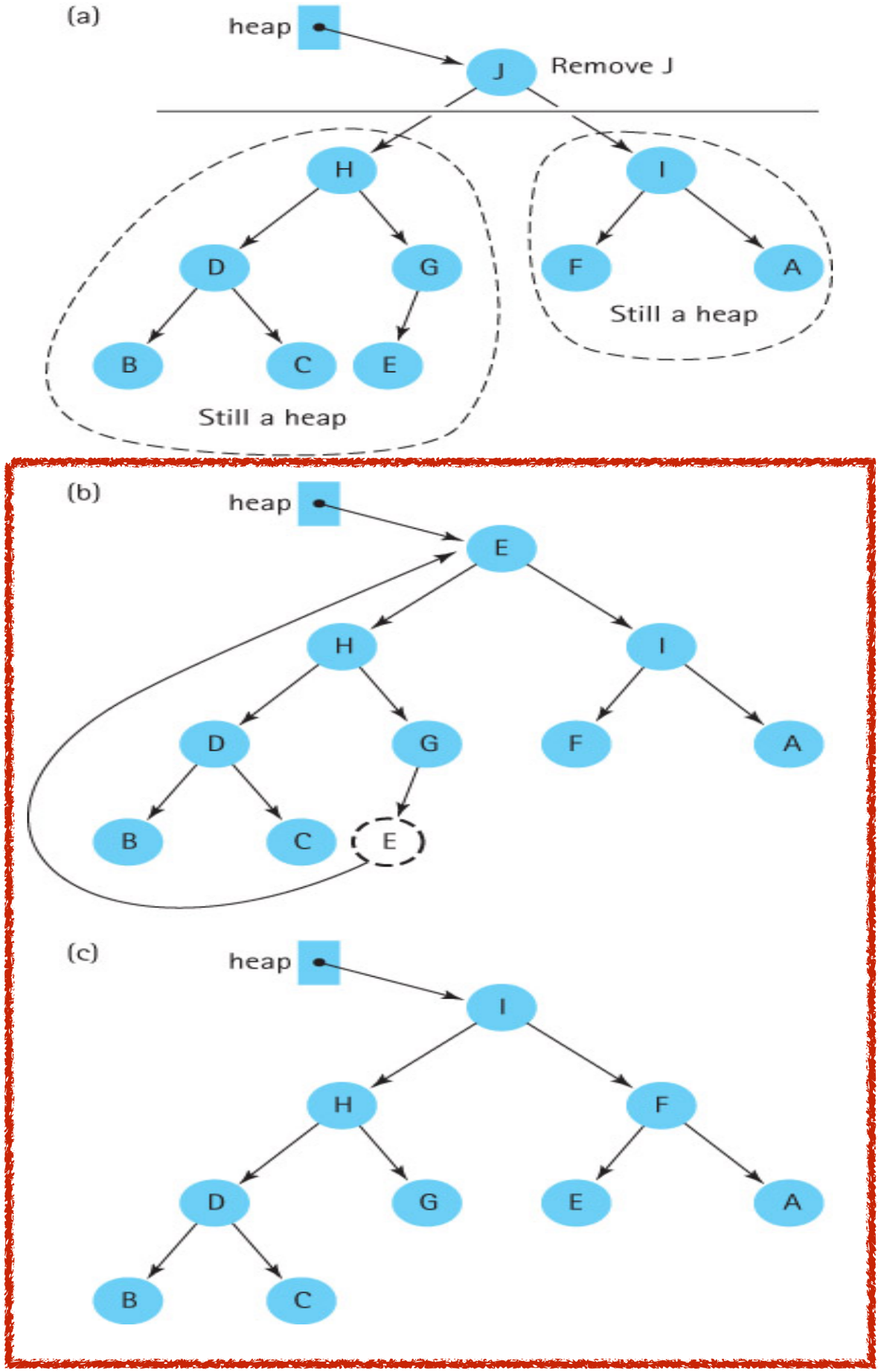
d) k c f b a d g



Heap-based Priority Queue

- **Dequeue**
  - To dequeue, we remove the root node (to be returned), but need to restructure the heap to ensure properties.
  - To ensure shape property, we substitute the root with the last node, so that the tree is still complete.
  - To ensure order property, we '**bubble down**' (aka **reheapDown**) the new root until it's no longer smaller than either of its children.





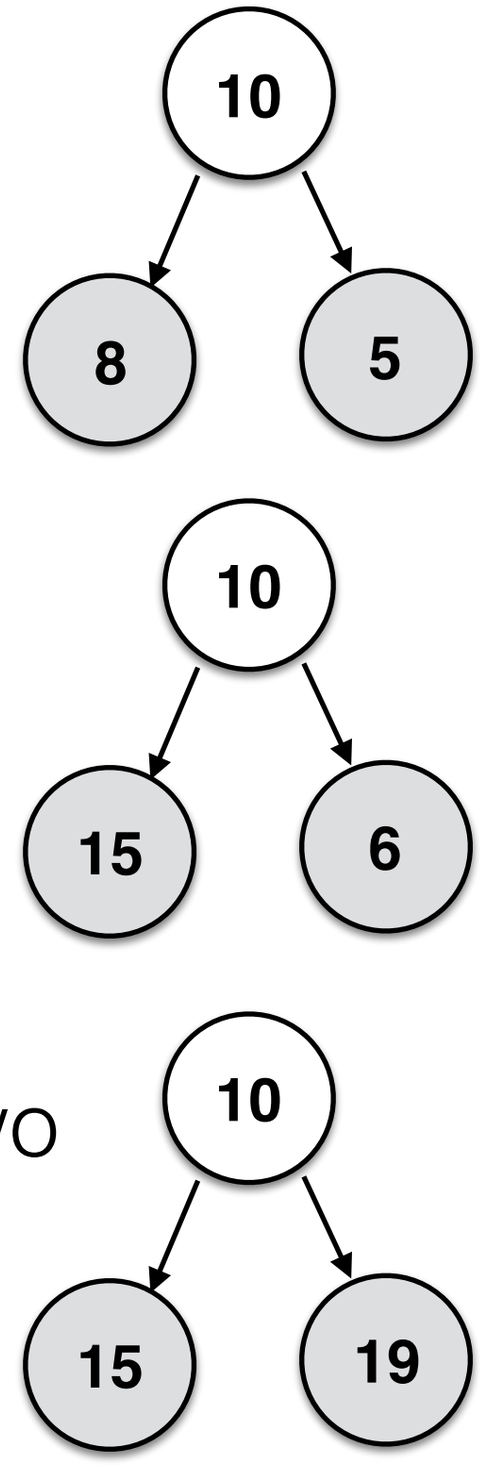
(a) We remove the root element and substitute it with the bottom right-most element to satisfy completeness.

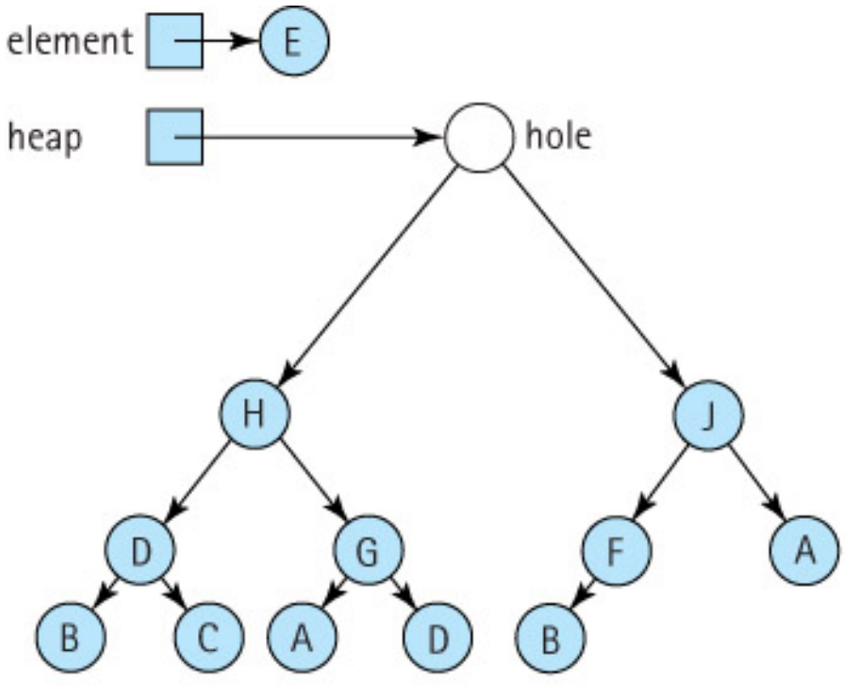
(b) Next, we bubble down the root node to satisfy the order property.

(c) This is more complex than bubble up because a node can have up to two children (while it has only one parent).

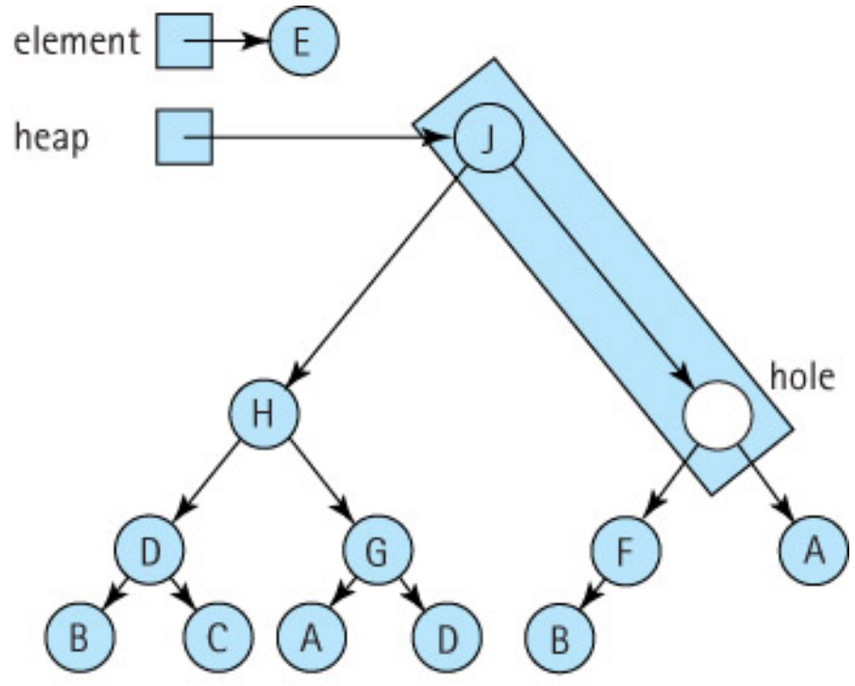
Bubble Down (reheapDown)

- During bubble-down, to ensure order property you may have several cases:
  - The node has no children with a value greater than it. —> You are done.
  - The node has 1 child whose value is greater
    —> bubble down along that child
  - Both children have values that are greater
    —> bubble down along the bigger of the two
    (if you pick the smaller of the two,
    it would violate the order property!)

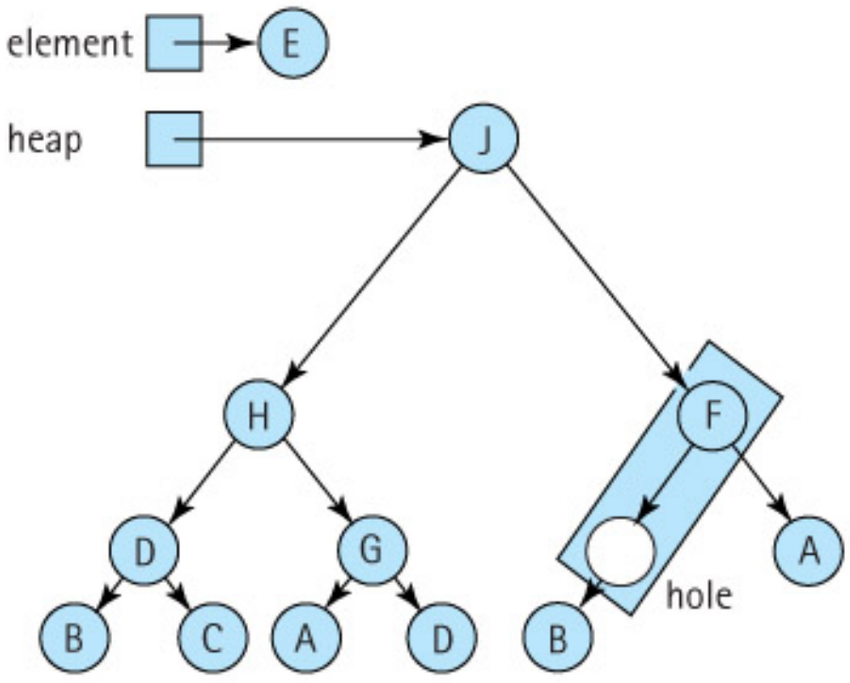




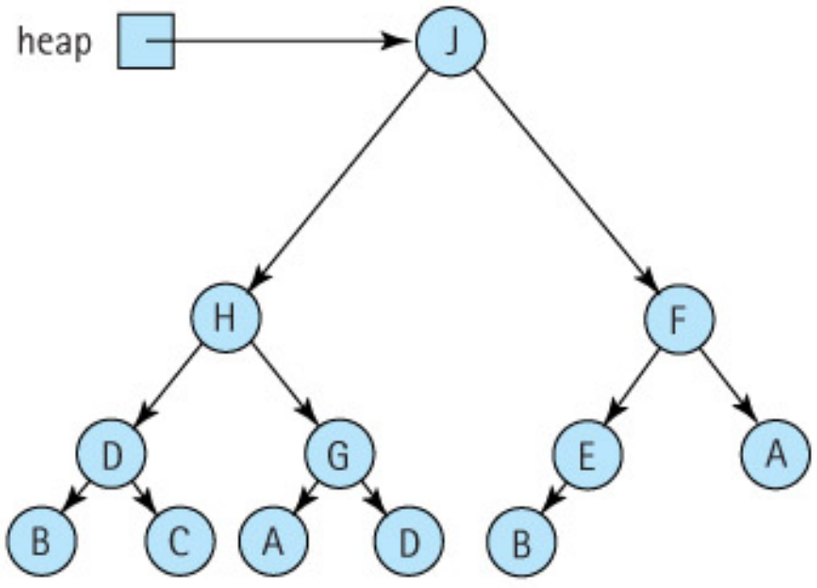
(a) reheapDown (E);



(b) Move hole down

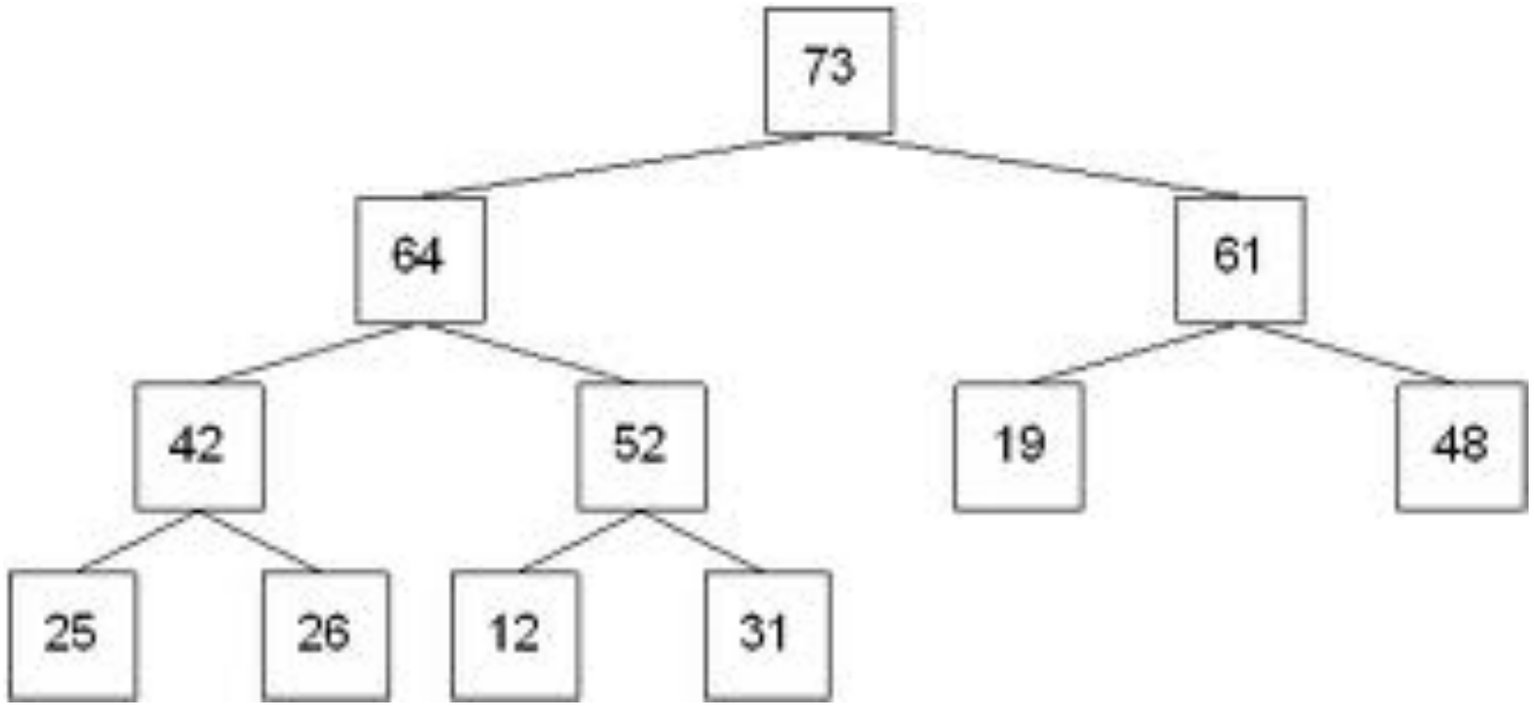


(c) Move hole down



(d) Fill in final hole

Clicker Question #4



If we dequeue the root element from this heap, what will be the values of the first seven nodes in the new heap?

(a) 73, 64, 61, 42, 52, 19, 48

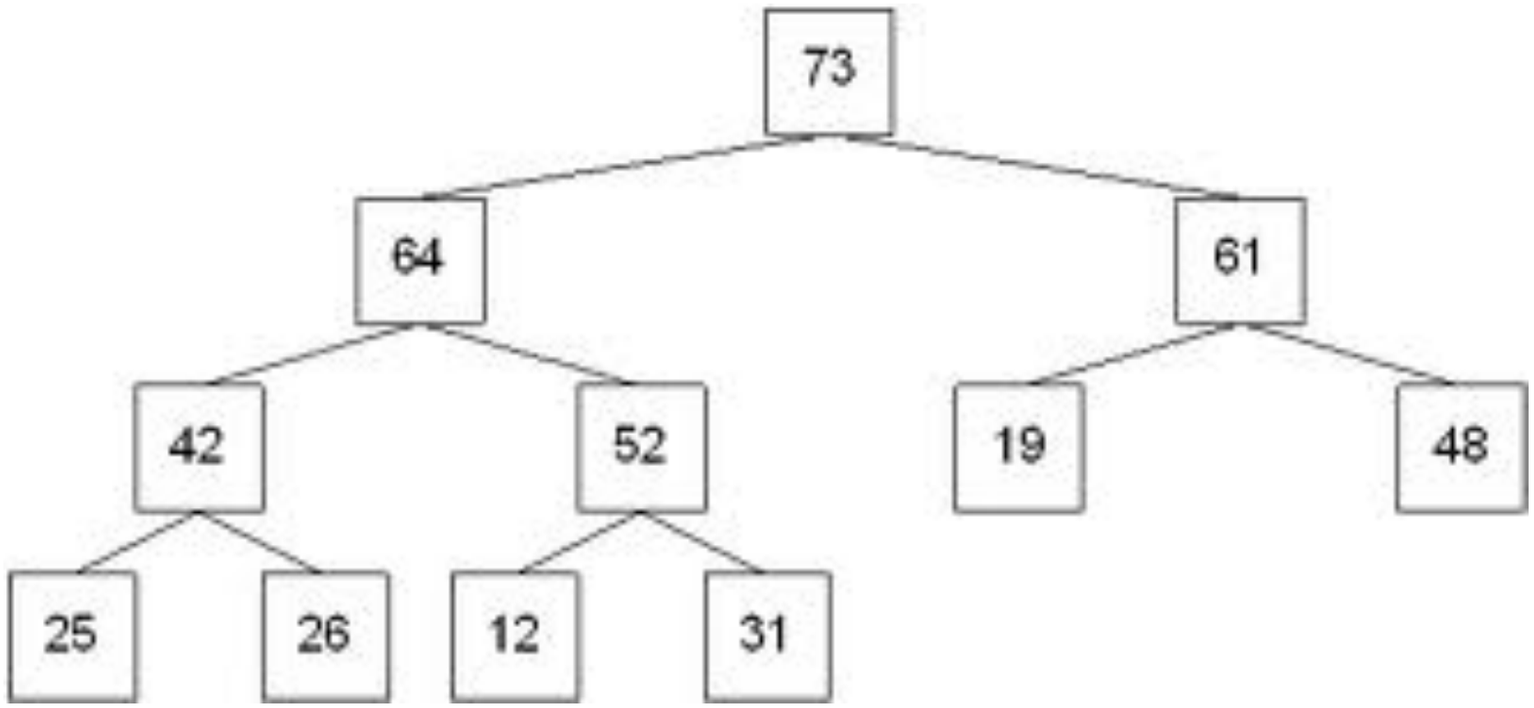
(b) 31, 64, 61, 42, 52, 19, 48

(c) 64, 61, 52, 48, 42, 31, 19

(d) 64, 52, 61, 42, 31, 19, 48

Answer on next slide

Clicker Question #4



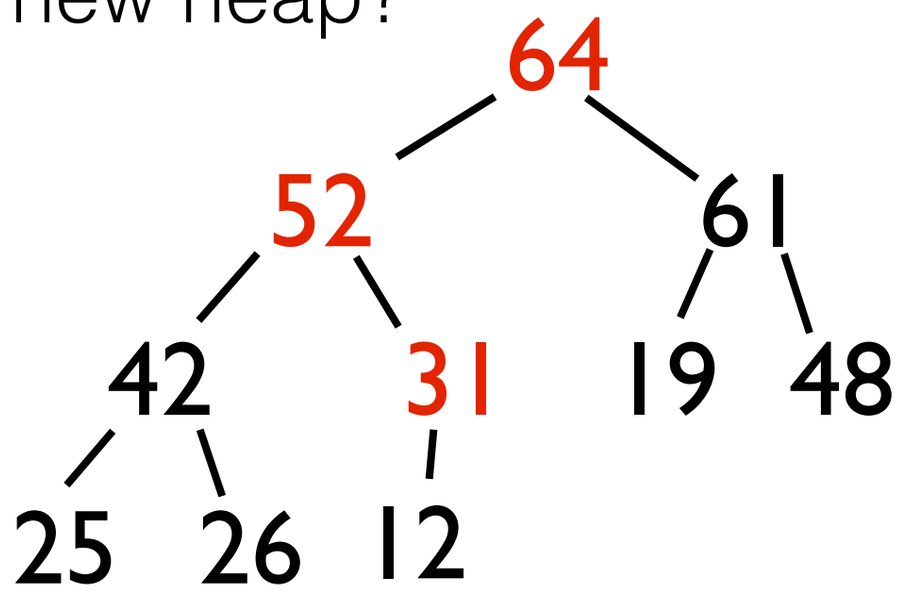
If we dequeue the root element from this heap, what will be the values of the first seven nodes in the new heap?

(a) 73, 64, 61, 42, 52, 19, 48

(b) 31, 64, 61, 42, 52, 19, 48

(c) 64, 61, 52, 48, 42, 31, 19

(d) 64, 52, 61, 42, 31, 19, 48



dequeue Code

```
// assume elements are stored in array ‘heap’
// last element is at index lastIndex
public T dequeue( ) throws PriQUnderflowException {
    T hold, toMove;
    if (lastIndex == -1)
        throw new PriQUnderflowException(“underflow”);
    else {
        hold = heap[0];
        toMove = heap[lastIndex];
        lastIndex--;
        if (lastIndex != -1)
            reheapDown(toMove);
        return hold;
    }
}
```

```
private void reheapDown (T elem) {
  int largerChild, hole=0, left=1, right=2;
  while (left <= lastIndex) { // at least 1 child
    if (right <= lastIndex && // right child exists
        heap[left].compareTo(heap[right]) < 0)
      largerChild = right;
    else
      largerChild = left;
    if (elem.compareTo(heap[largerChild]) >= 0)
      break;
    heap[hole] = heap[largerChild];
    hole = largerChild;
    left = (hole*2)+1, right = (hole*2)+2;
  }
  heap[hole] = elem;
}
```

Heaps vs other implementations

	enqueue	dequeue
Heap	$O(\log_2 N)$	$O(\log_2 N)$
Linked List	$O(N)$	$O(1)$
Binary Search Tree		
Balanced	$O(\log_2 N)$	$O(\log_2 N)$
Skewed	$O(N)$	$O(N)$