CSE 373

Priority queue implementation using a heap

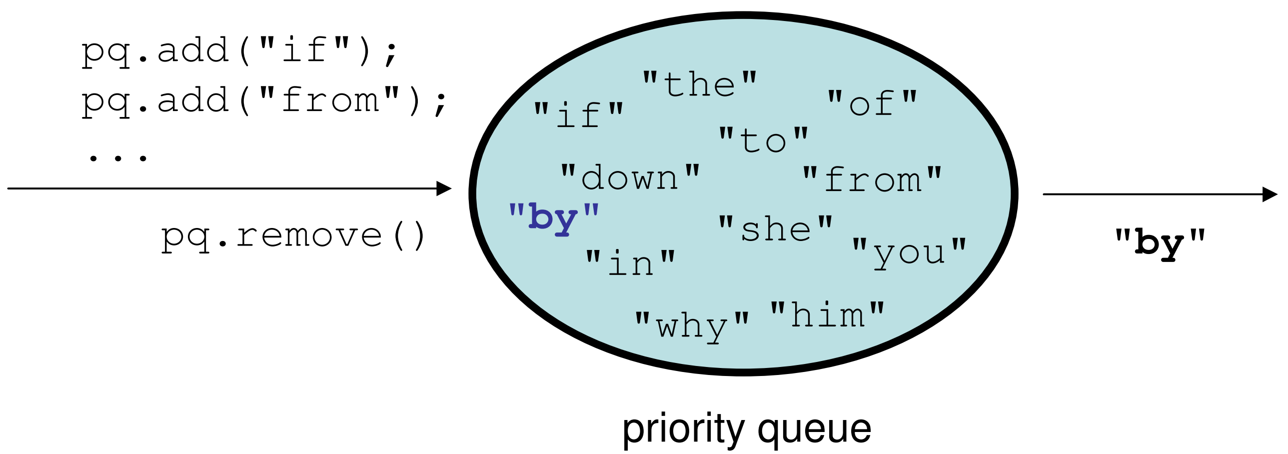
read: Weiss Ch. 6

slides created by Marty Stepp
http://www.cs.washington.edu/373/

© University of Washington, all rights reserved.

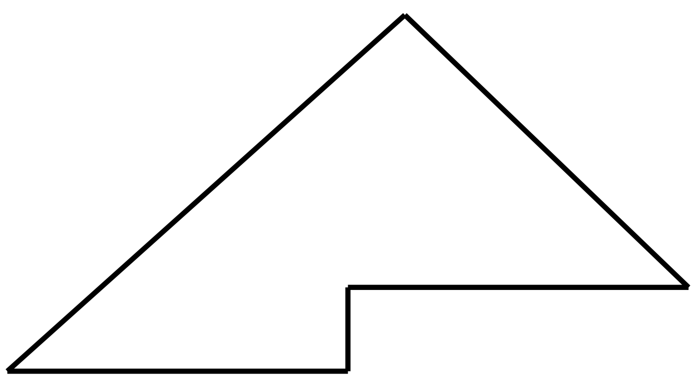
Priority Queue ADT

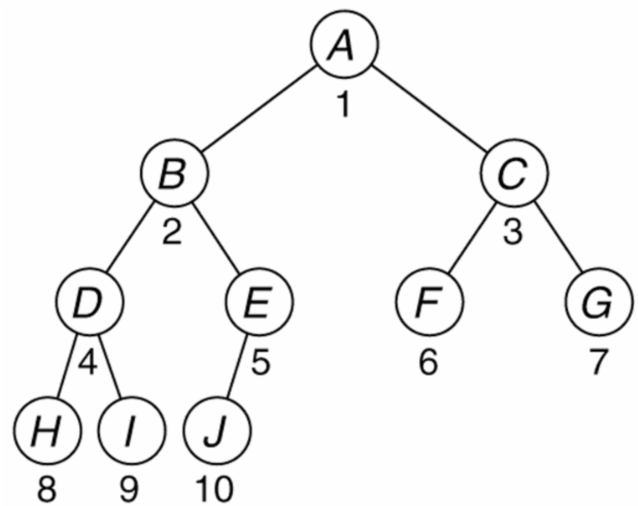
- **priority queue**: A collection of ordered elements that provides fast access to the minimum (or maximum) element.
  - `add` adds in order
  - `peek` returns **minimum** or "highest priority" value
  - `remove` removes/returns **minimum** value
  - `isEmpty`, `clear`, `size`, `iterator` O(1)



Heaps

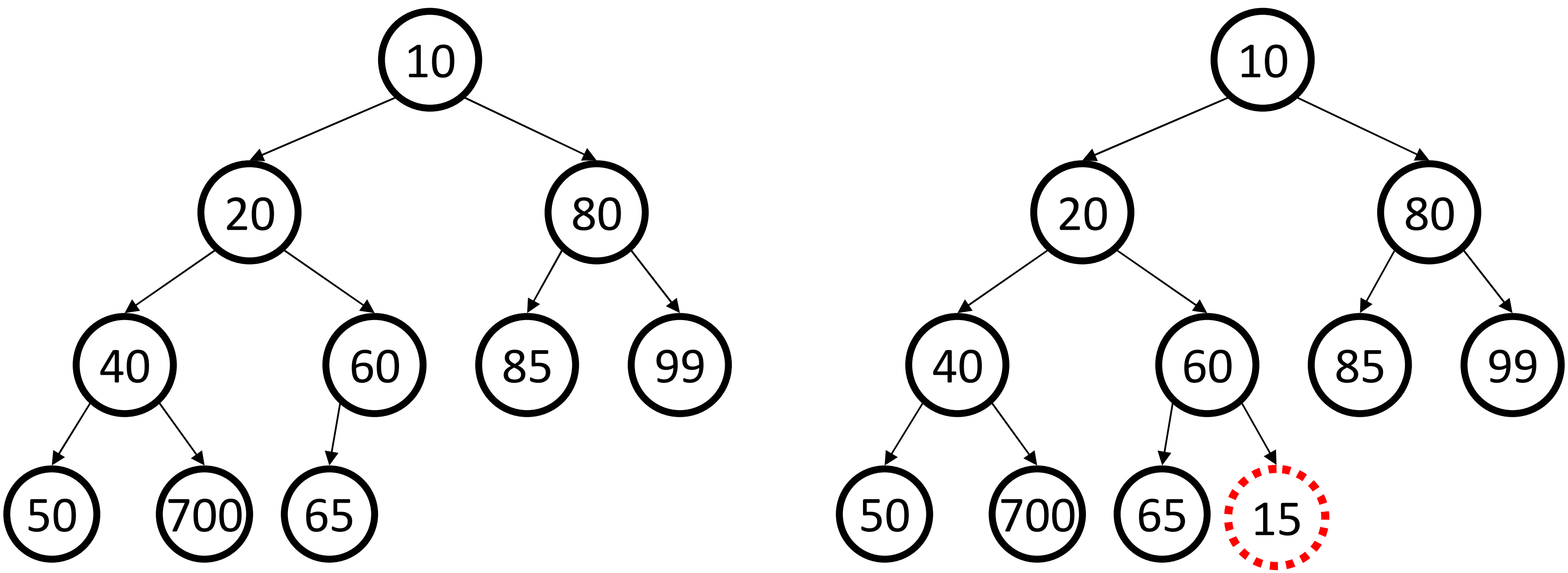
- **heap**: A *complete* binary tree with *vertical* ordering.
  - **complete tree**: Every level is full except possibly the lowest level, which must be filled from left to right
    - (i.e., a node may not have any children until all possible siblings exist)





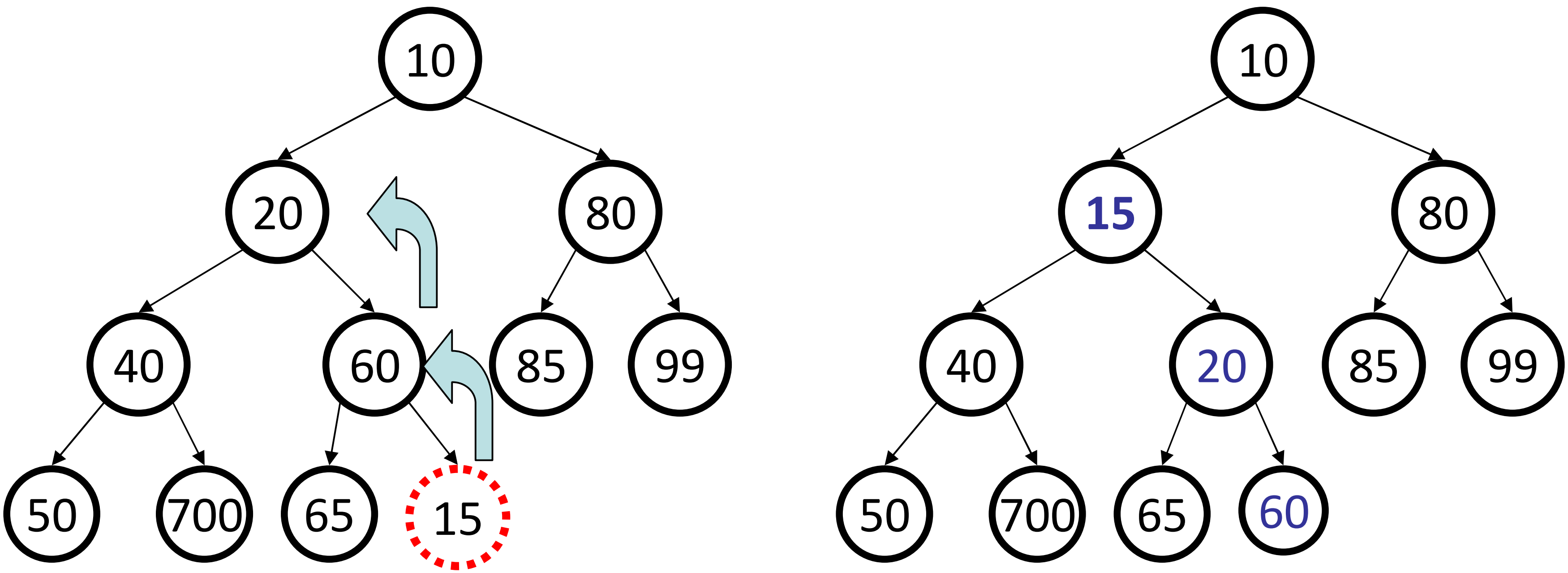
The add operation

- When an element is added to a heap, it should be initially placed as the *rightmost leaf* (to maintain the completeness property).
  - But the heap ordering property becomes broken!



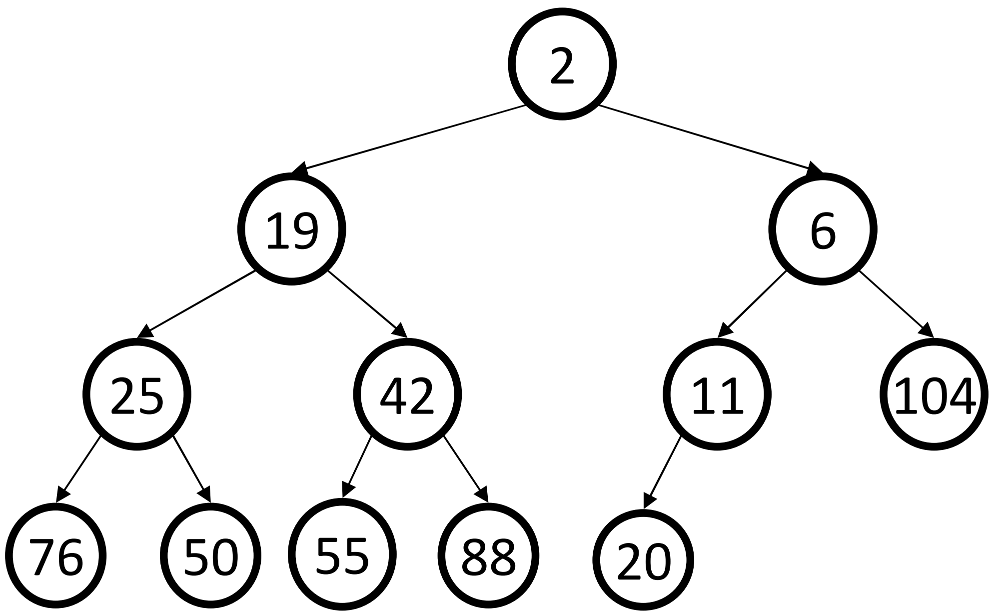
"Bubbling up" a node

- **bubble up**: To restore heap ordering, the newly added element is shifted ("bubbled") up the tree until it reaches its proper place.
  - Weiss: *"percolate up"* by swapping with its parent
  - How many bubble-ups are necessary, at most?



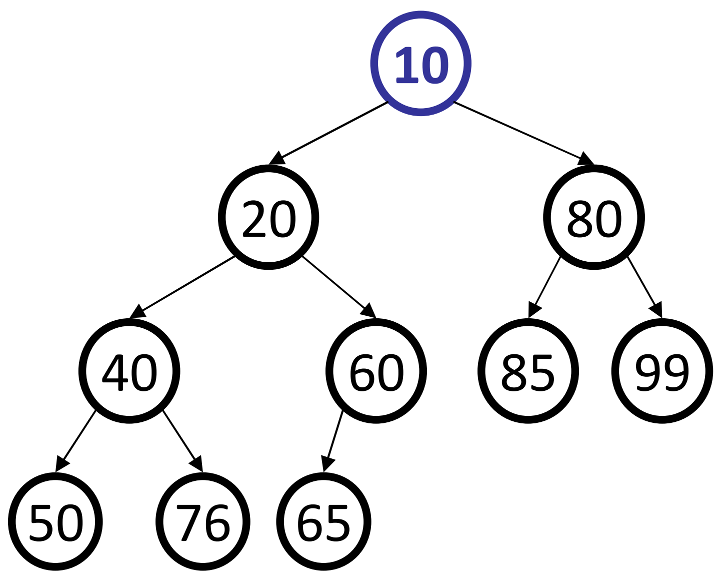
Bubble-up add exercise

- Draw the tree state of a min-heap after adding these elements:
  - 6, 50, 11, 25, 42, 20, 104, 76, 19, 55, 88, 2



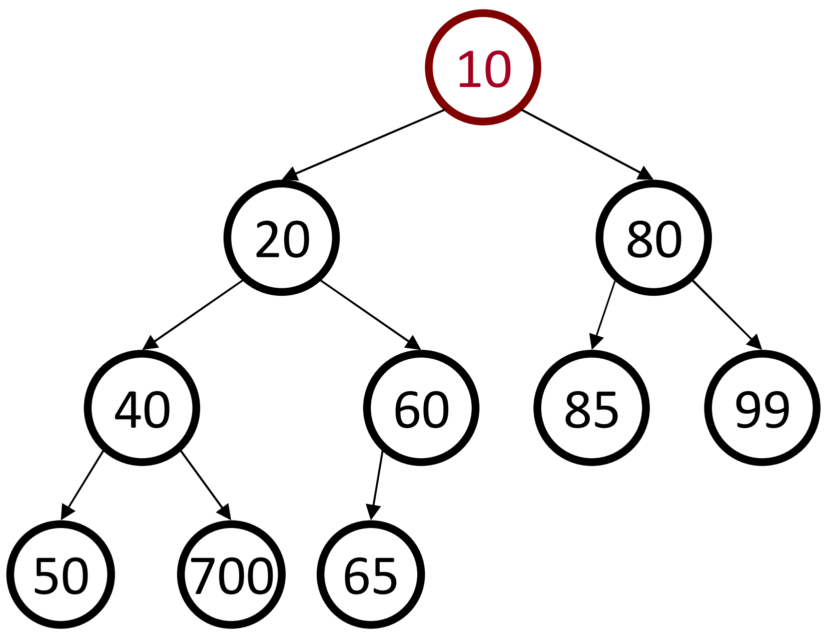
The peek operation

- A peek on a min-heap is trivial to perform.
  - because of heap properties, minimum element is always the root
  - O(1) runtime
- Peek on a max-heap would be O(1) as well (return max, not min)



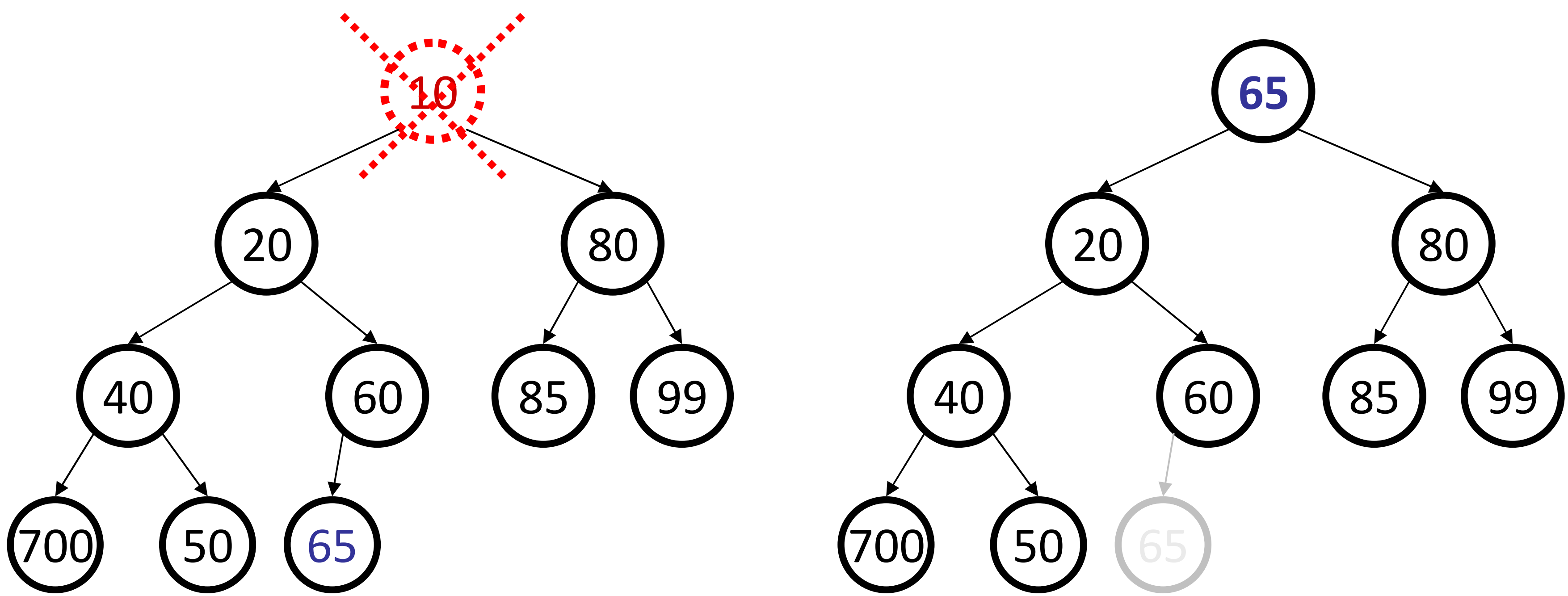
The remove operation

- When an element is removed from a heap, what should we do?
  - The root is the node to remove. How do we alter the tree?
  - `queue.remove();`



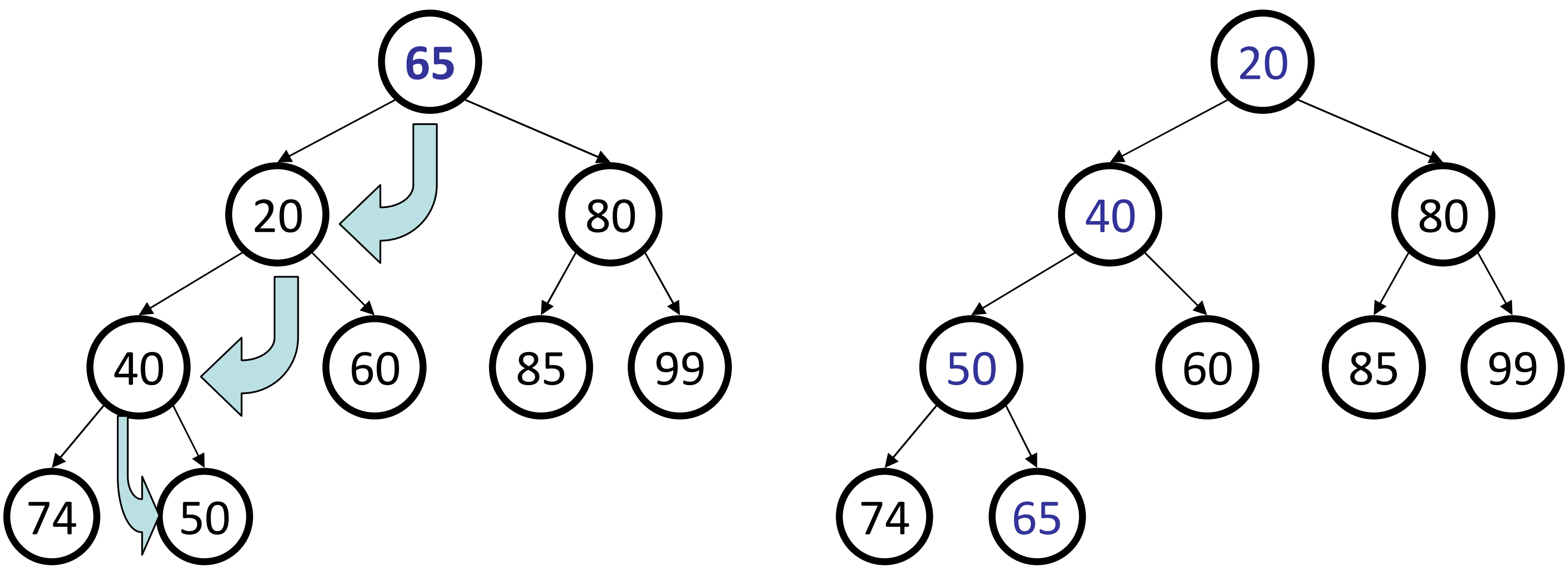
The remove operation

- When the root is removed from a heap, it should be initially replaced by the *rightmost leaf* (to maintain completeness).
  - But the heap ordering property becomes broken!



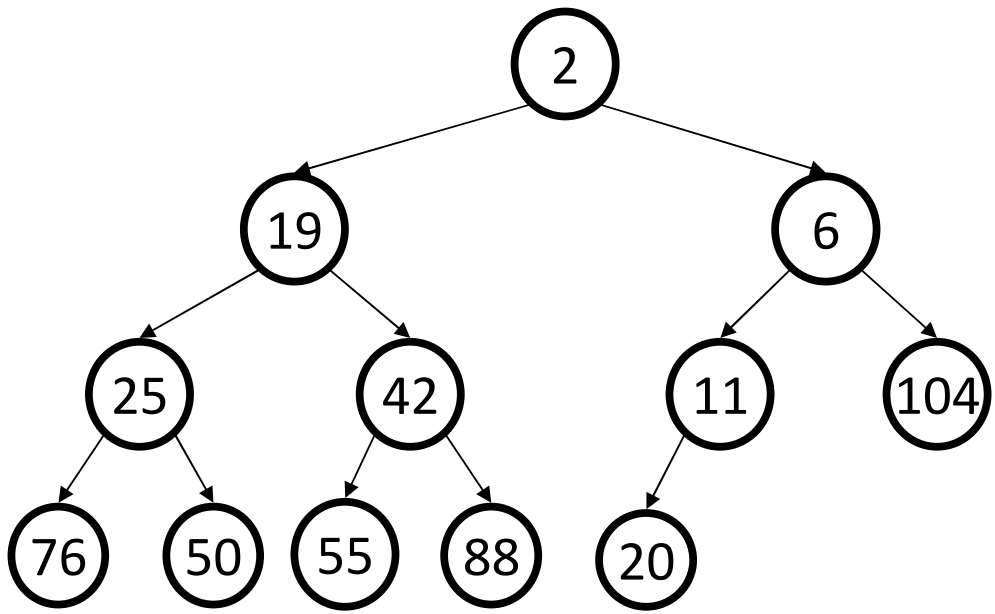
"Bubbling down" a node

- **bubble down**: To restore heap ordering, the new improper root is shifted ("bubbled") down the tree until it reaches its proper place.
  - Weiss: *"percolate down"* by swapping with its *smaller* child (why?)
  - How many bubble-down are necessary, at most?



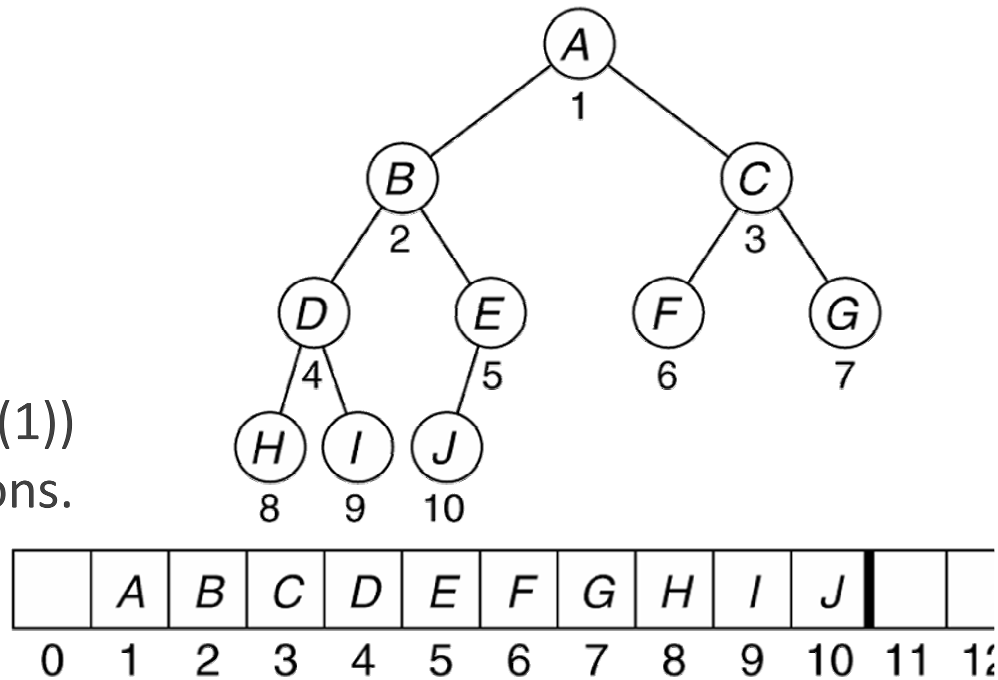
Bubble-down exercise

- Suppose we have the min-heap shown below.
- Show the state of the heap tree after remove has been called 3 times, and which elements are returned by the removal.



Array heap implementation

- Though a heap is conceptually a binary tree, since it is a *complete* tree, when implementing it we actually can "cheat" and just *use an array*!
  - index of root = 1 (leave 0 empty to simplify the math)
  - for any node *n* at index *i* :
    - index of *n*.left = 2*i*
    - index of *n*.right = 2*i* + 1
    - parent index of *n*?
  - This array representation is elegant and efficient (O(1)) for common tree operations.



Implementing HeapPQ

- Let's implement an `int` priority queue using a min-heap array.

```
public class HeapIntPriorityQueue
        implements IntPriorityQueue {
    private int[] elements;
    private int size;

    // constructs a new empty priority queue
    public HeapIntPriorityQueue() {
        elements = new int[10];
        size = 0;
    }

    ...
}
```

Helper methods

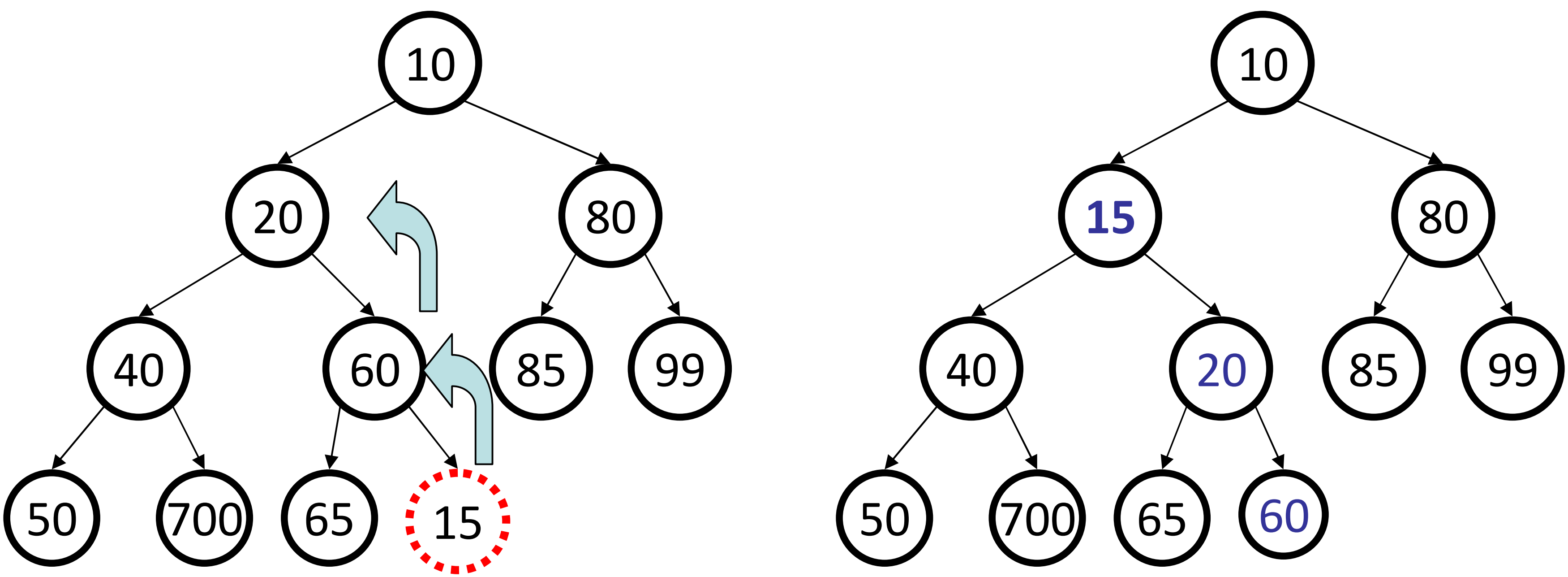
- Since we will treat the array as a complete tree/heap, and walk up/down between parents/children, these methods are helpful:

```
// helpers for navigating indexes up/down the tree
private int parent(int index)        { return index/2; }
private int leftChild(int index)     { return index*2; }
private int rightChild(int index)    { return index*2 + 1; }
private boolean hasParent(int index) { return index > 1; }
private boolean hasLeftChild(int index) {
    return leftChild(index) <= size;
}
private boolean hasRightChild(int index) {
    return rightChild(index) <= size;
}
private void swap(int[] a, int index1, int index2) {
    int temp = a[index1];
    a[index1] = a[index2];
    a[index2] = temp;
}
```

Implementing add

- Let's write the code to add an element to the heap:

```
public void add(int value) {
    ...
}
```



Implementing add

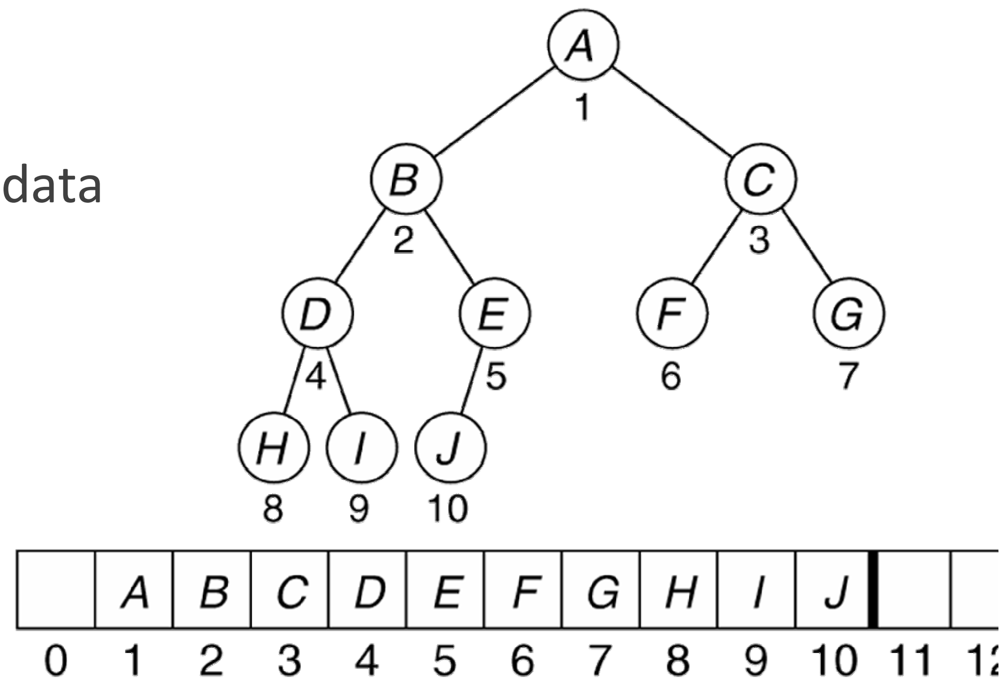
```
// Adds the given value to this priority queue in order.
public void add(int value) {
    elements[size + 1] = value;  // add as rightmost leaf

    // "bubble up" as necessary to fix ordering
    int index = size + 1;
    boolean found = false;
    while (!found && hasParent(index)) {
        int parent = parent(index);
        if (elements[index] < elements[parent]) {
            swap(elements, index, parent(index));
            index = parent(index);
        } else {
            found = true;  // found proper location; stop
        }
    }

    size++;
}
```

Resizing a heap

- What if our array heap runs out of space?
  - We must enlarge it.
  - When enlarging hash sets, we needed to carefully rehash the data.
  - What must we do here?
  - (We can simply copy the data into a larger array.)



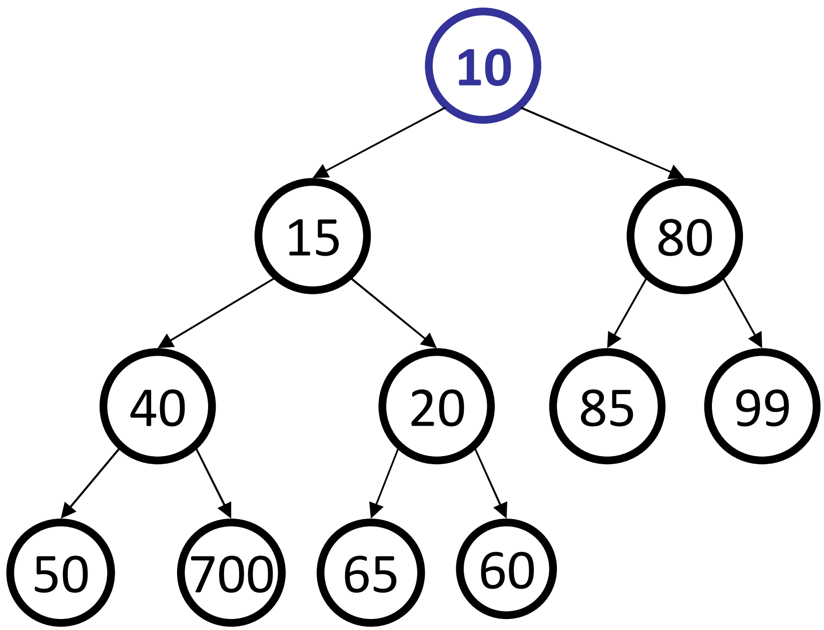
Modified add code

```
// Adds the given value to this priority queue in order.
public void add(int value) {
    // resize to enlarge the heap if necessary
    if (size == elements.length - 1) {
        elements = Arrays.copyOf(elements,
                                 2 * elements.length);
    }
    ...
}
```

Implementing peek

- Let's write code to retrieve the minimum element in the heap:

```
public int peek() {
    ...
}
```



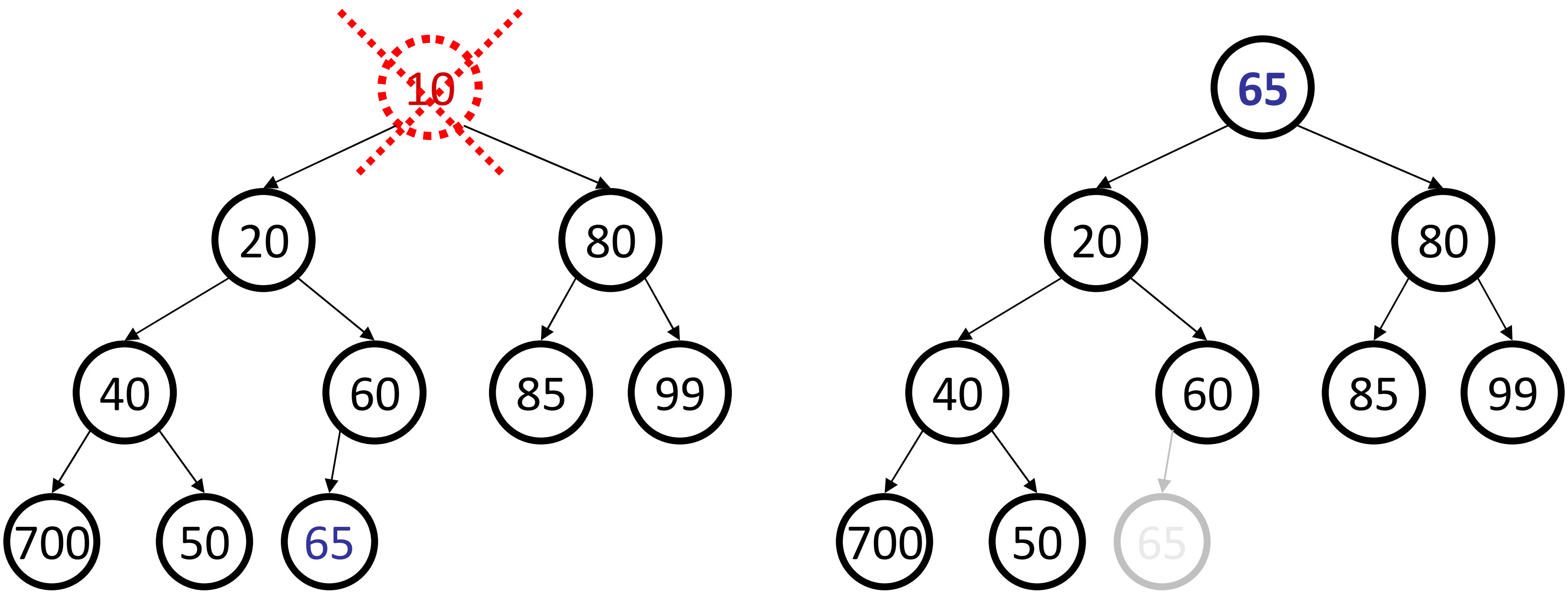
Implementing peek

```
// Returns the minimum element in this priority queue.
// precondition: queue is not empty
public int peek() {
    return elements[1];
}
```

Implementing remove

- Let's write code to remove the minimum element in the heap:

```
public int remove() {
    ...
}
```



Implementing remove

```
public int remove() {    // precondition: queue is not empty
    int result = elements[1];      // last leaf -> root
    elements[1] = elements[size];
    size--;
    int index = 1;    // "bubble down" to fix ordering
    boolean found = false;
    while (!found && hasLeftChild(index)) {
        int left = leftChild(index);
        int right = rightChild(index);
        int child = left;
        if (hasRightChild(index) &&
                elements[right] < elements[left]) {
            child = right;
        }
        if (elements[index] > elements[child]) {
            swap(elements, index, child);
            index = child;
        } else {
            found = true;  // found proper location; stop
        }
    }
    return result;
}
```