COSC 222 Data Structure

Priority Queue and Heaps

Prioritization problems

- **Print jobs:** printers constantly accept and complete jobs from all over the SCI building. We may want to print department chair's jobs before other professors, etc.

- **Emergency Room (ER) scheduling:** Scheduling patients for treatment in the ER. A heart attack victim should be treated sooner than a patient with a cold, regardless of arrival time.

Priority Queues versus Queues

- A **queue** stores items by arrival time, and the item that arrived first is removed first.
- A **priority queue** stores items with priorities, and the item with the highest priority is removed first.

Priority Queue ADT

- **Priority queue**: A collection of ordered elements that provides fast access to the **minimum** (or **maximum**) element.

  - `add` adds in order
  - `remove` removes/returns minimum value
  - `peek` returns minimum or "highest priority" value
  - `isEmpty`, `size`, `clear`

Implementation ideas: Unsorted array

- Consider using an unsorted array to implement a priority queue.
  - **add**: Store it in the next available index, as in a list.
  - **peek**: Loop over elements to find minimum element.
  - **remove**: Loop over elements to find min. Shift to remove.

```
queue.add(9);
queue.add(23);
queue.add(8);
queue.add(-3);
queue.add(49);
queue.add(12);
queue.remove();
```

index	0	1	2	3	4	5	6	7	8	9
value	9	23	8	**-3**	49	12	0	0	0	0
size	6									

Implementation ideas: Unsorted array

index	0	1	2	3	4	5	6	7	8	9
value	9	23	8	**-3**	49	12	0	0	0	0
size	6									

- How efficient is `add`? `peek`? `remove`?
  - O(1), O(N), O(N)
  - (peek must loop over the array; remove must shift elements)

Implementation ideas: Sorted array

- Consider using a sorted array to implement a priority queue.
  - add: Store it in the proper index to maintain sorted order.
  - peek: Minimum element is in index [0].
  - remove: Shift elements to remove min from index [0].

```
queue.add(9);
queue.add(23);
queue.add(8);
queue.add(-3);
queue.add(49);
queue.add(12);
queue.remove();
```

index	0	1	2	3	4	5	6	7	8	9
value	**-3**	8	9	12	23	49	0	0	0	0
size	6									

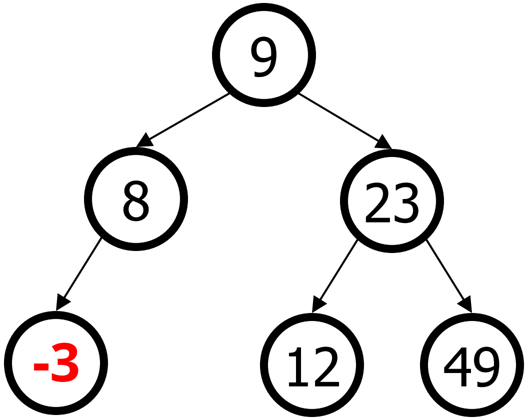
Implementation ideas: Sorted array

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	**-3**	8	9	12	23	49	0	0	0	0
size	6									

- How efficient is add? peek? remove?
  - O(N), O(1), O(N)
  - (add and remove must shift elements)

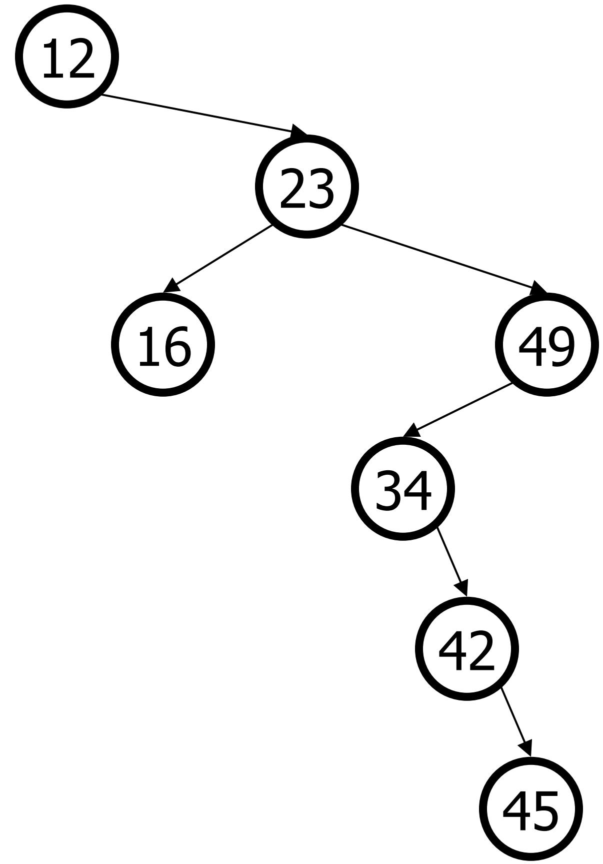
Implementation ideas: Binary search tree

- Consider using a binary search tree to implement a PQ.
  - `add`: Store it in the proper BST L/R - ordered spot.
  - `peek`: Minimum element is at the far left edge of the tree.
  - `remove`: Unlink far left element to remove.



Binary search tree?

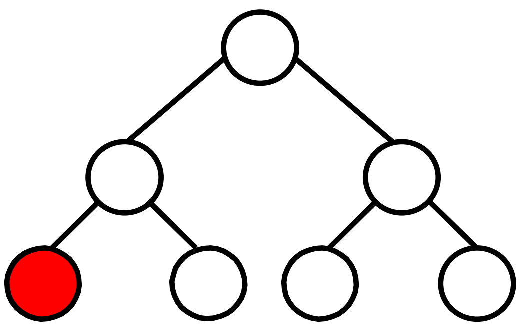
- How efficient is add? peek? remove?
  - O(N), O(N), O(N)...?
- A tree that is unbalanced has a height close to N rather than log N, which breaks the expected runtime of many operations.



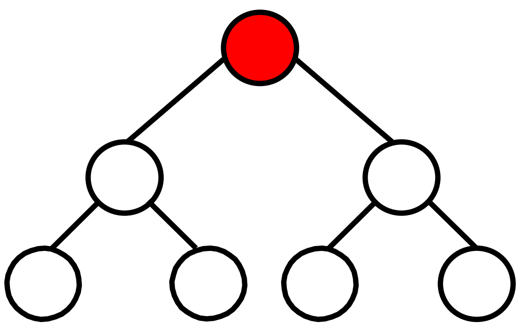
Binary heap invariants

- **Idea**: adapt the tree-based method
- **Insight**: in a tree, finding the min is expensive!
  - Rather then having it to the left, **have it on the top**!

A BST or AVL tree

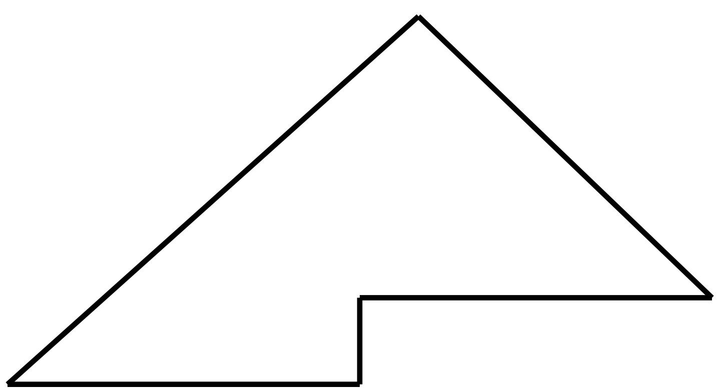


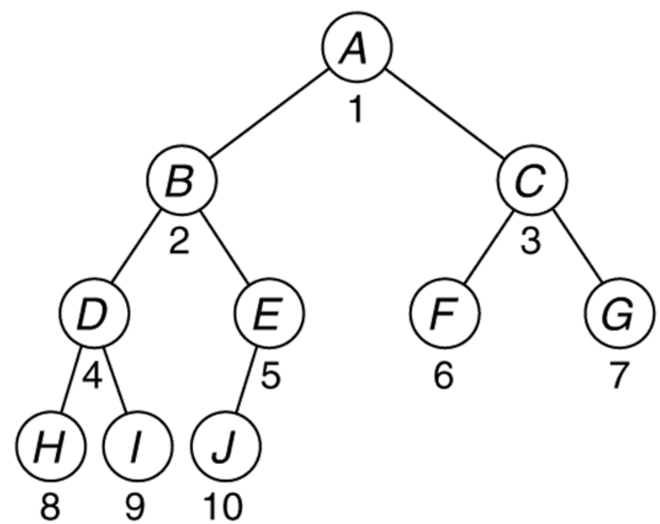
A binary heap



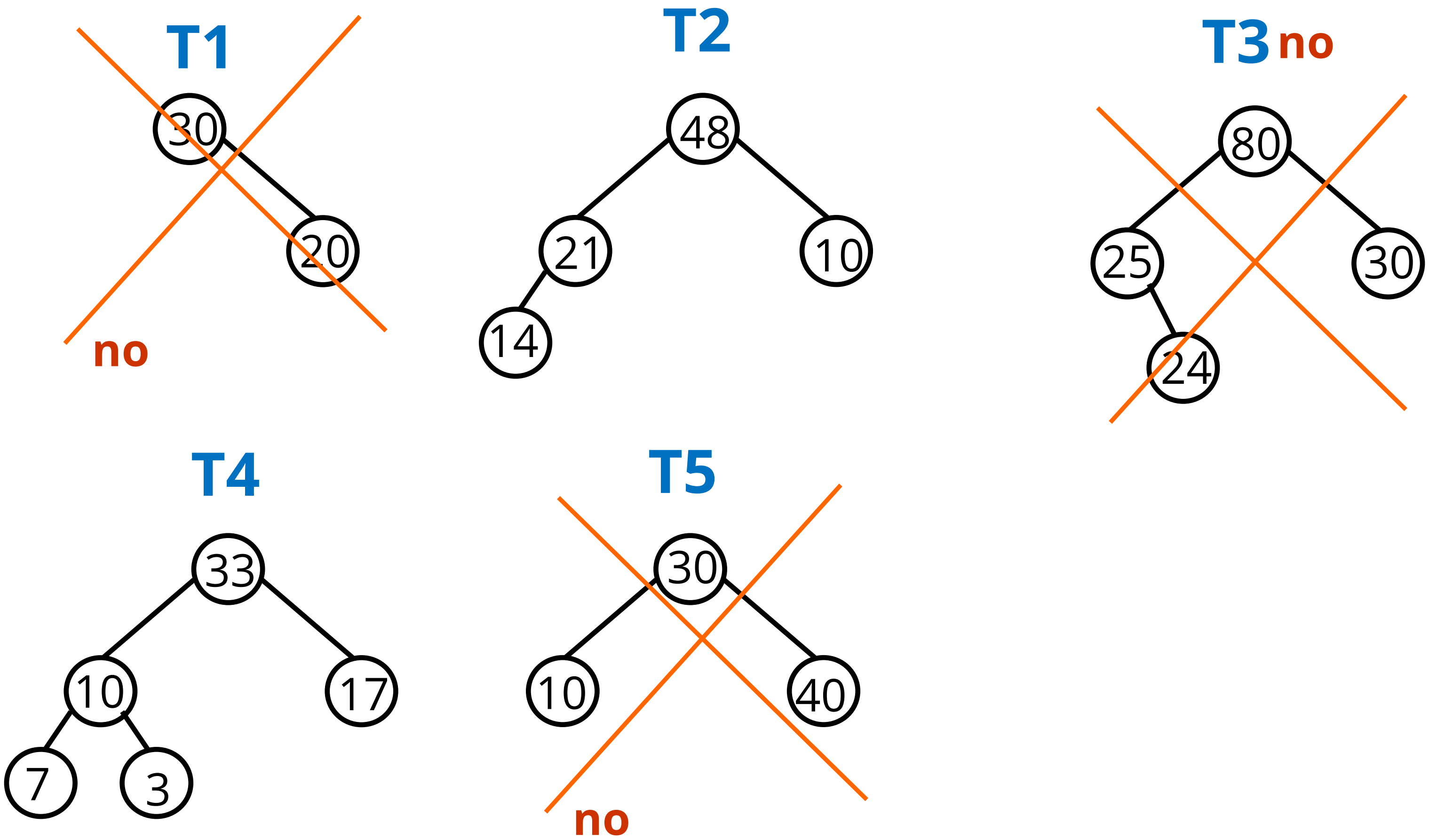
Heaps

- A heap is a binary tree satisfying 2 properties:
- **Completeness:** Every level of the tree (except last) is completely filled, and on last level nodes are as far left as possible.
- **Heap-order:**
  - Max-Heap: every element in tree is <= its parent (**max on top**)
  - Min-Heap: every element in tree is >= its parent (**min on top**)





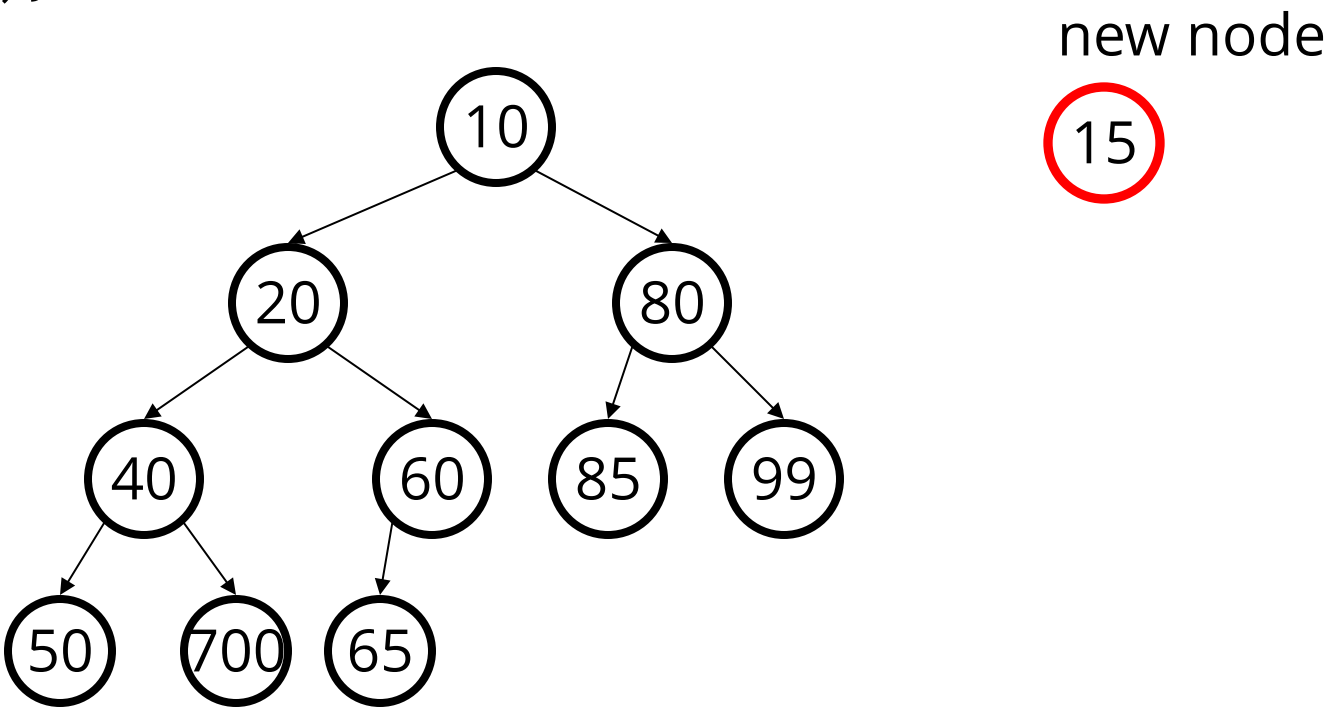
Which of the following are examples of max-heaps?



The add operation

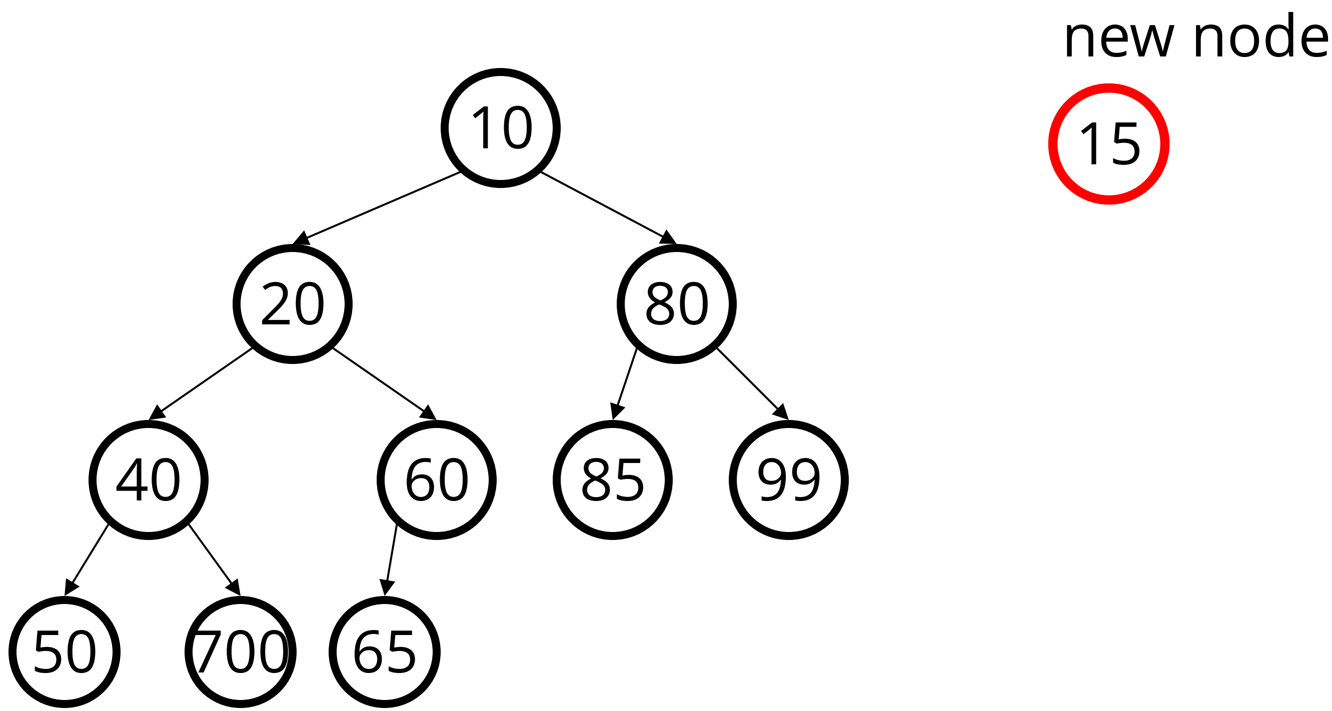
- When an element is added to a heap, where should it go?
  - Must insert a new node while maintaining heap properties.

```
queue.add(15);
```



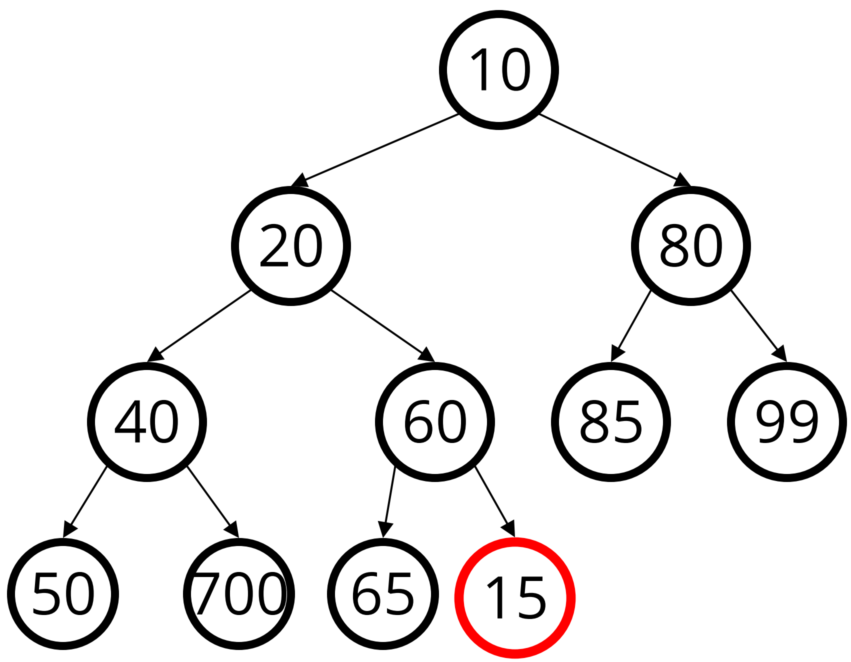
The add operation

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



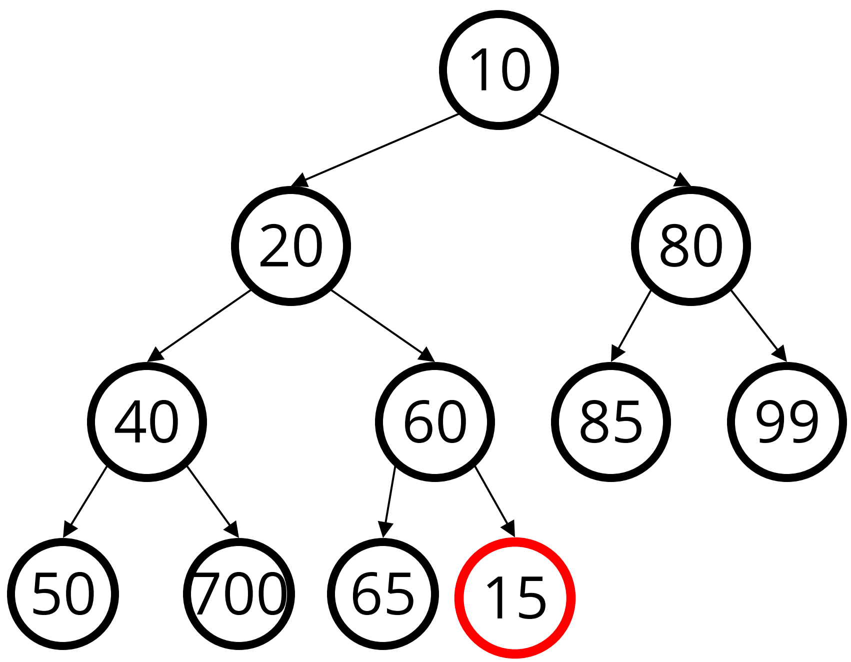
The add operation

- 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!



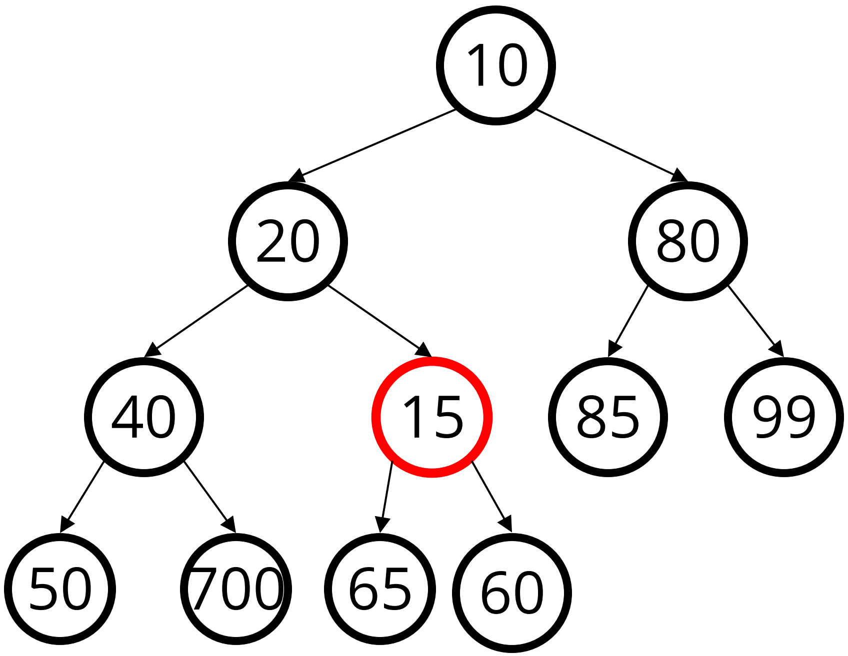
The add operation

- **bubble up**: To restore heap ordering, the newly added element is shifted ("bubbled") up the tree until it reaches its proper place.
  - "bubble up" by swapping with its parent



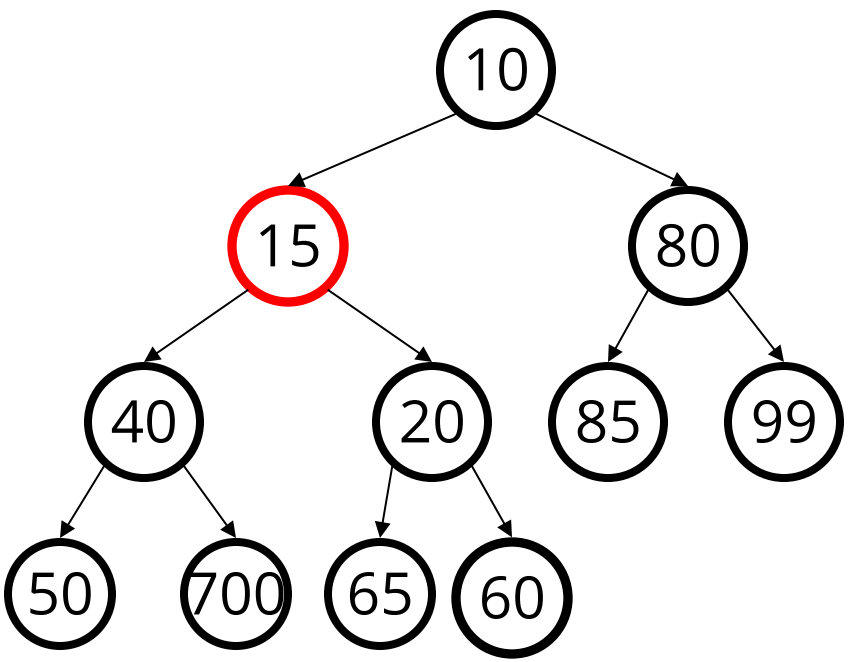
The add operation

- **bubble up**: To restore heap ordering, the newly added element is shifted ("bubbled") up the tree until it reaches its proper place.
  - "bubble up" by swapping with its parent



The add operation

- **bubble up**: To restore heap ordering, the newly added element is shifted ("bubbled") up the tree until it reaches its proper place.
  - "bubble up" by swapping with its parent
  - All OK!



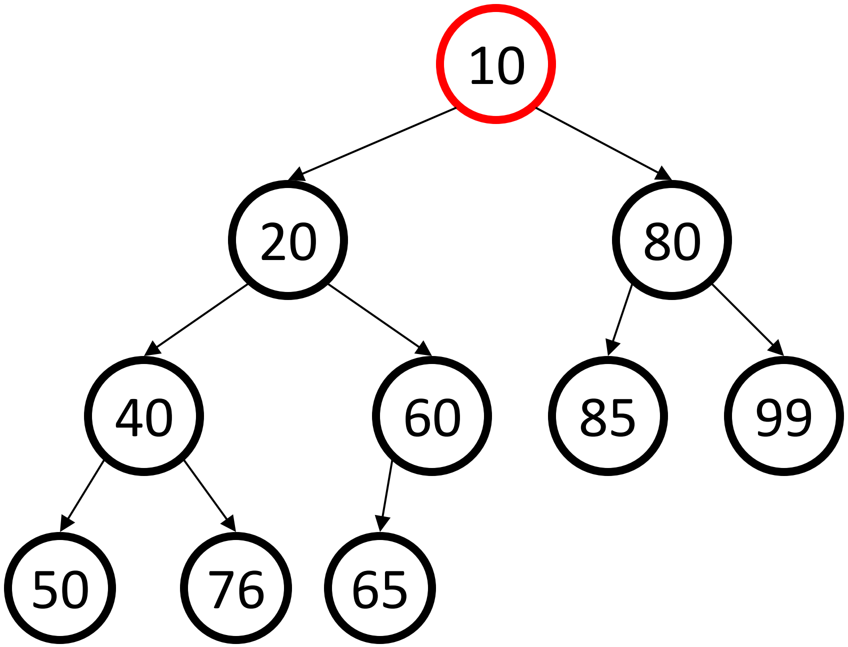
Analyzing insert

- We usually need to bubble up a few times! So, number of swaps ≈ height ≈ log(n) in the worst case!

The remove operation

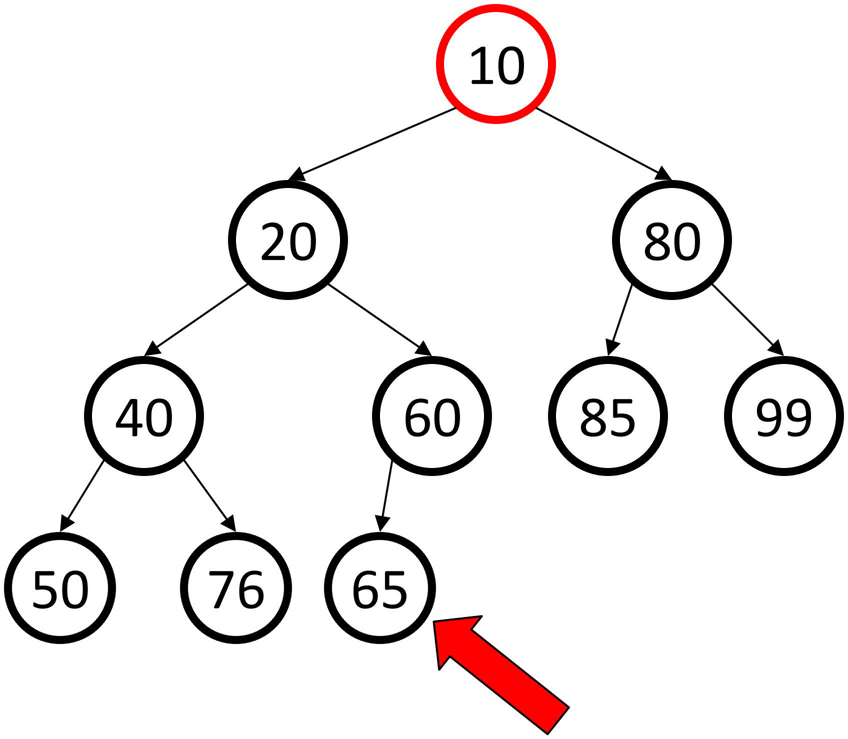
- When an element is removed from a heap, what should we do?
  - The root is the node to remove. How do we alter the tree?

```
queue.remove();
```



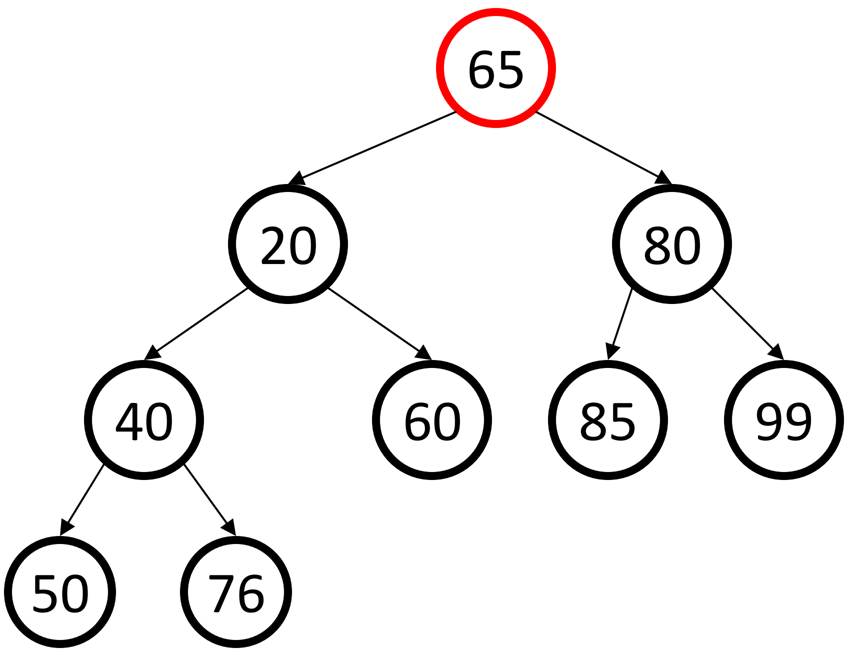
The remove operation

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



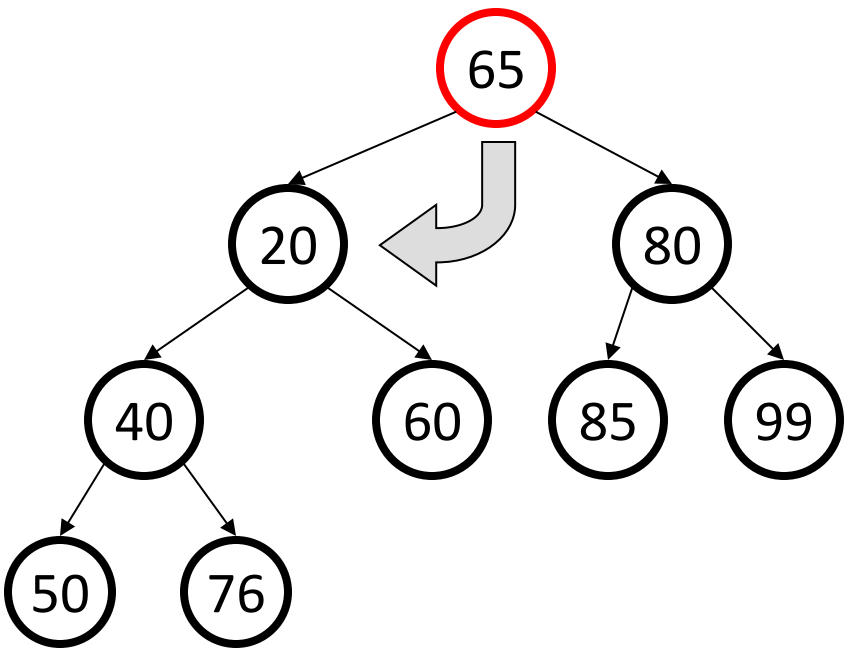
The remove operation

- 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!



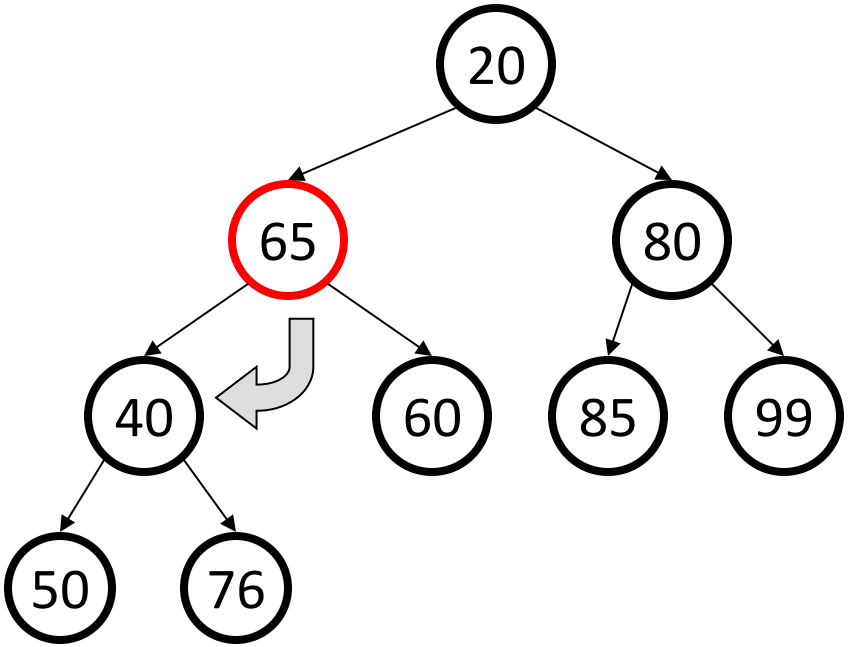
The remove operation

- **Bubble down**: To restore heap ordering, the new improper root is shifted ("bubbled") down the tree until it reaches its proper place.
  - "bubble down" by swapping with its **smaller child**



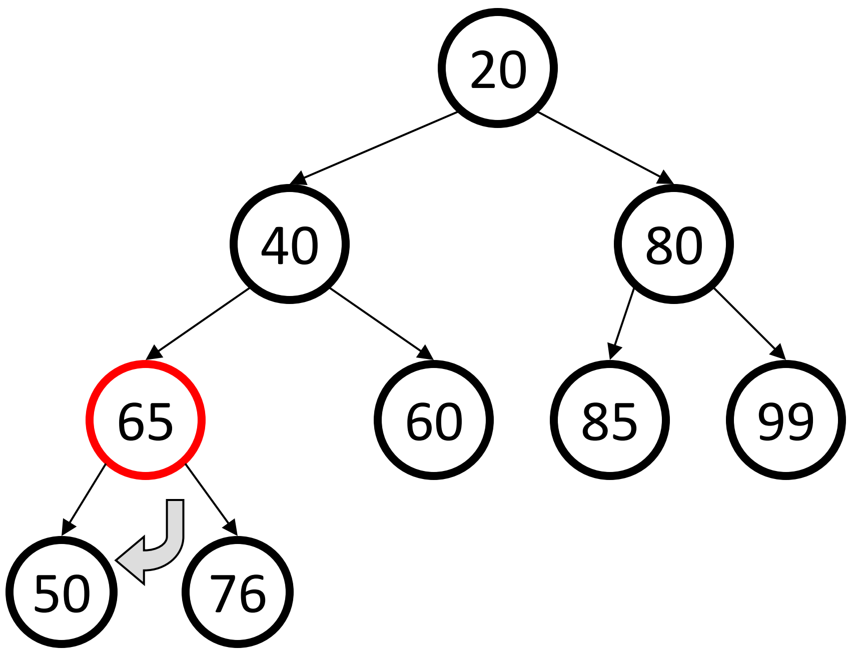
The remove operation

- **Bubble down**: To restore heap ordering, the new improper root is shifted ("bubbled") down the tree until it reaches its proper place.
  - "bubble down" by swapping with its **smaller child**



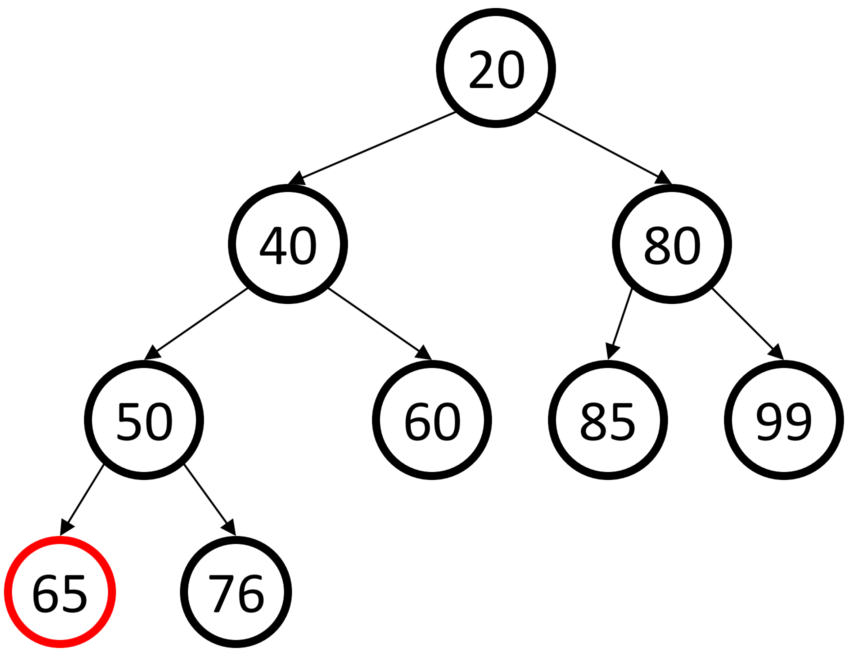
The remove operation

- **Bubble down**: To restore heap ordering, the new improper root is shifted ("bubbled") down the tree until it reaches its proper place.
  - "bubble down" by swapping with its **smaller child**



The remove operation

- **Bubble down**: To restore heap ordering, the new improper root is shifted ("bubbled") down the tree until it reaches its proper place.
  - "bubble down" by swapping with its **smaller child**

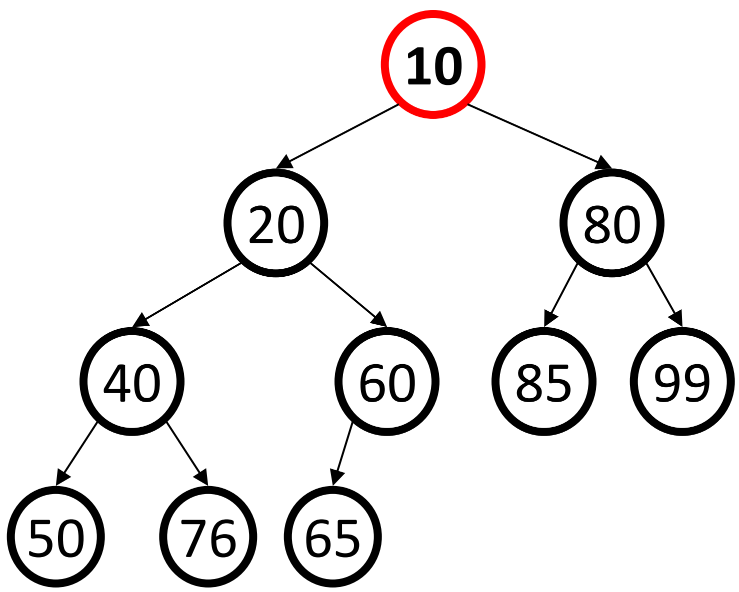


Analyzing remove

- Usually must bubble all the way down. So number of swaps ≈ height ≈ log(n).

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)



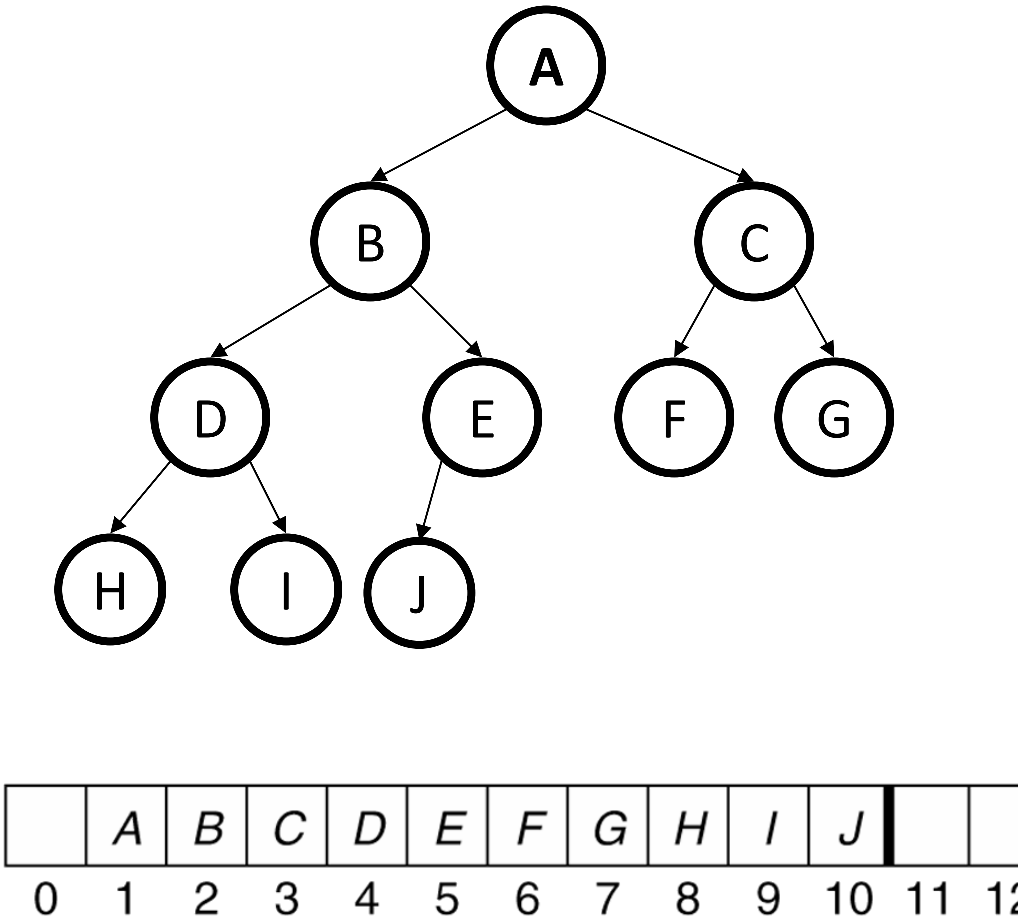
Heap height and runtime

- The height of a complete tree is always log N.

- Because of this, if we implement a priority queue using a heap, we can provide the following runtime guarantees:
  - `add`: O(log N)
  - `peek`: O(1)
  - `remove`: O(log N)

Array heap implementation

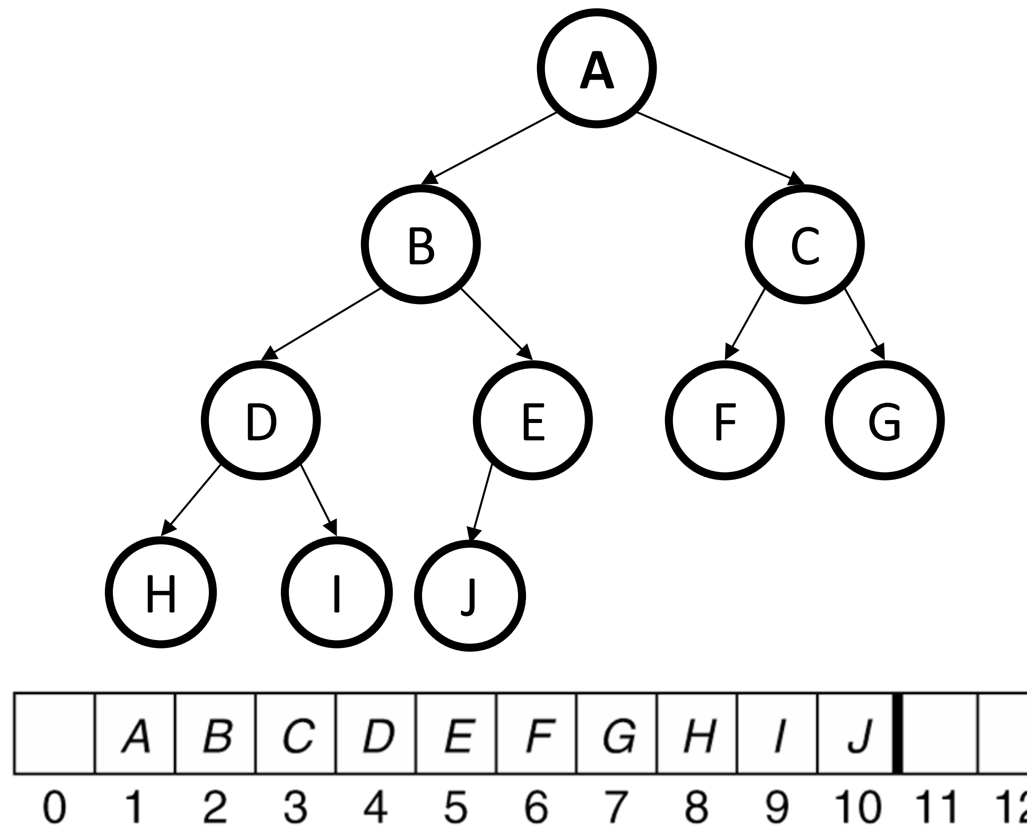
- Though a heap is conceptually a binary tree, since it is a complete tree, when implementing it we actually just use an array!



Array heap implementation

- index of root = 1 (leave 0 empty to simplify the math)
- for any node n at index i :
  - index of n.left   = 2i
  - index of n.right = 2i + 1
  - parent index of n?

    At index $\lfloor n/2 \rfloor$



Implementing HeapPQ

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

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

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

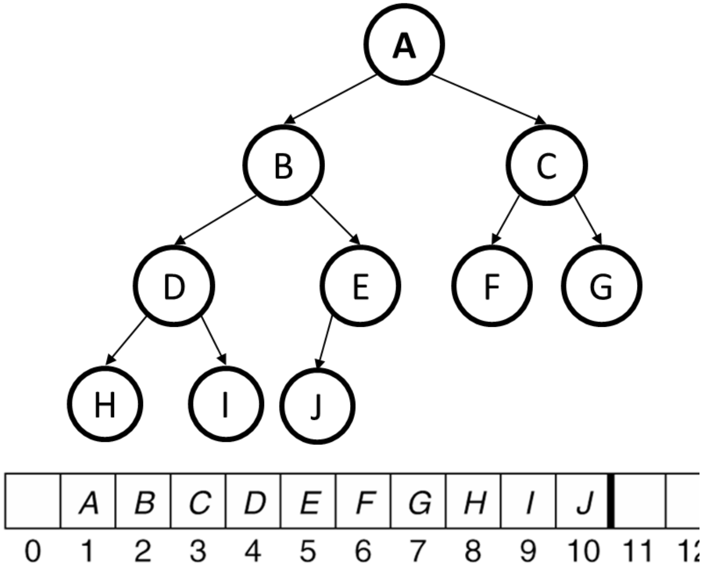
Helper methods

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

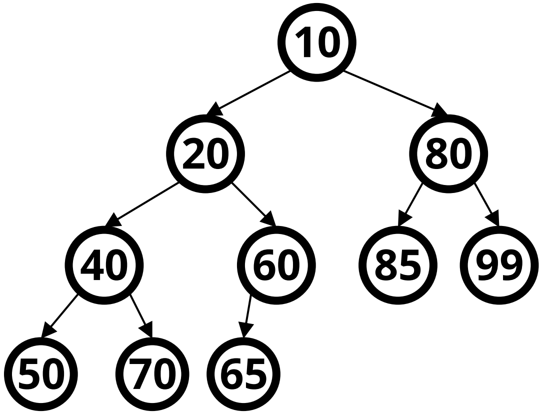
```
// helpers for navigating indexes up/down the tree

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; }
```

Helper methods

```
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;
}
```

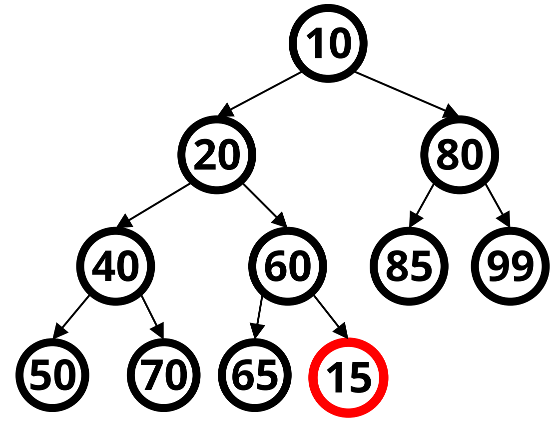


Implementing add




```
public void add(int value) {
    elements[size + 1] = value;
    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
        }
    }
    size++;
}
```

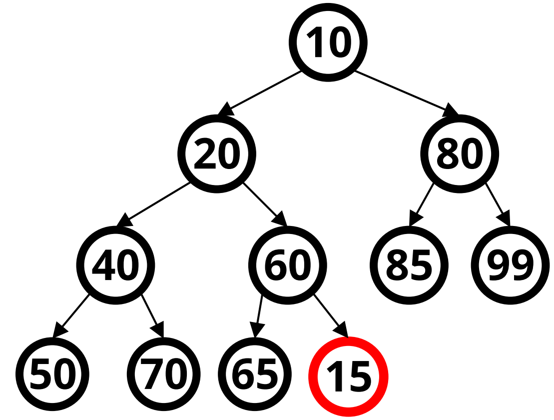
Implementing add



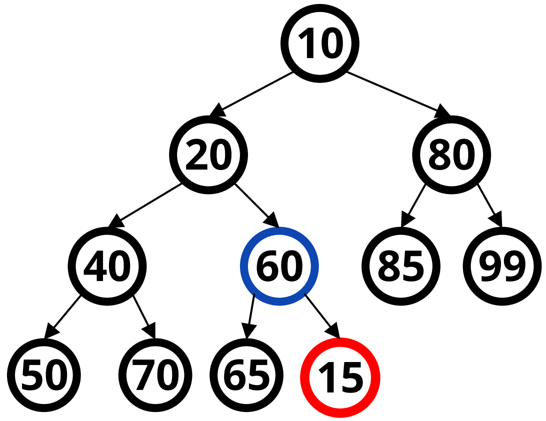

```
public void add(int value) {
    elements[size + 1] = value;
    int index = size + 1;
    boolean found = false;
    while (!found && hasParent(index)) {
        int parent = parent(index);
        if (elements[index] < elements[parent]) {
            swap(elements, index, parent(index));
            index = parent(index);
        } else {
            found = true;  // found proper location
        }
    }
    size++;
}
```

Implementing add

```
public void add(int value) {
    elements[size + 1] = value;
    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
        }
    }
    size++;
}
```

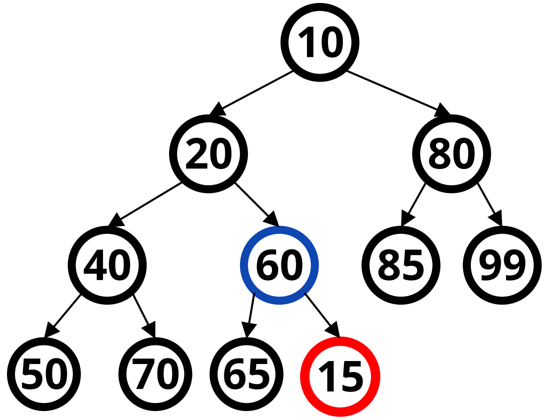


Implementing add



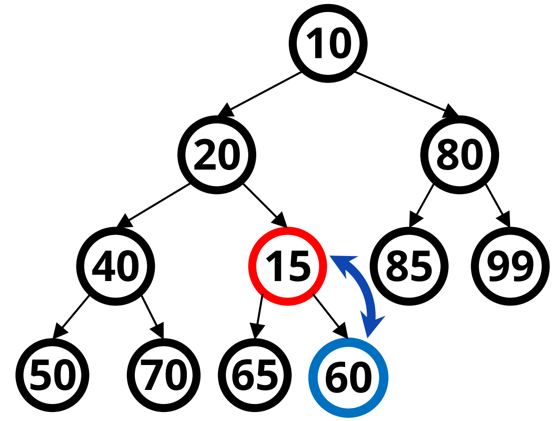

```
public void add(int value) {
    elements[size + 1] = value;
    int index = size + 1;
    boolean found = false;
    while (!found && hasParent(index)) {
        int parent = parent(index);
        if (elements[index] < elements[parent]) {
            swap(elements, index, parent(index));
            index = parent(index);
        } else {
            found = true;   // found proper location
        }
    }
    size++;
}
```

Implementing add



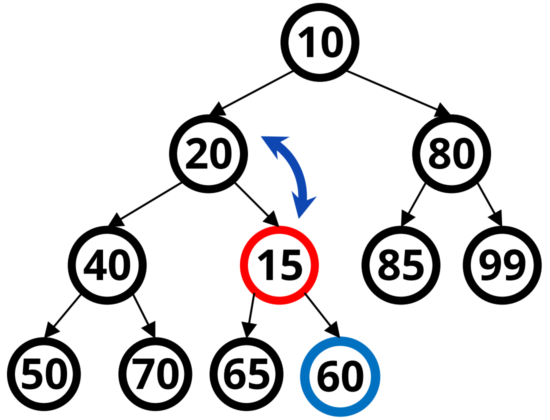

```
public void add(int value) {
    elements[size + 1] = value;
    int index = size + 1;
    boolean found = false;
    while (!found && hasParent(index)) {
        int parent = parent(index);
        if (elements[index] < elements[parent]) {
            swap(elements, index, parent(index));
            index = parent(index);
        } else {
            found = true;   // found proper location
        }
    }
    size++;
}
```

Implementing add



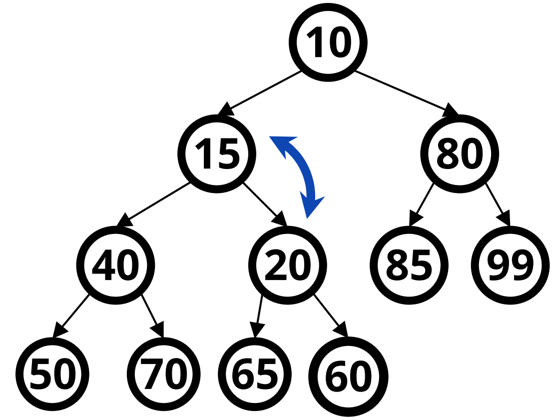

```
public void add(int value) {
    elements[size + 1] = value;
    int index = size + 1;
    boolean found = false;
    while (!found && hasParent(index)) {
        int parent = parent(index);
        if (elements[index] < elements[parent]) {
            swap(elements, index, parent(index));
            index = parent(index);
        } else {
            found = true;  // found proper location
        }
    }
    size++;
}
```

Implementing add




```
public void add(int value) {
    elements[size + 1] = value;
    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
        }
    }
    size++;
}
```

Implementing add

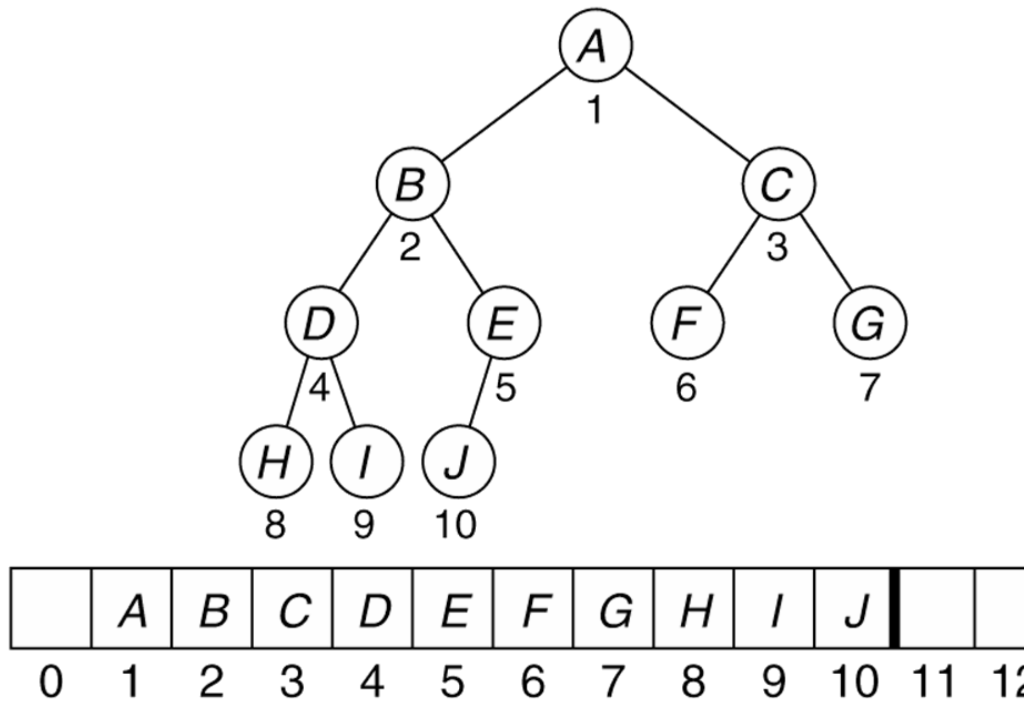



```
public void add(int value) {
    elements[size + 1] = value;
    int index = size + 1;
    boolean found = false;
    while (!found && hasParent(index)) {
        int parent = parent(index);
        if (elements[index] < elements[parent]) {
            swap(elements, index, parent(index));
            index = parent(index);
        } else {
            found = true;  // found proper location
        }
    }
    size++;
}
```

Resizing a heap

- What if our array heap runs out of space?
  - We must enlarge it.
- We can simply copy the

data into a larger array.



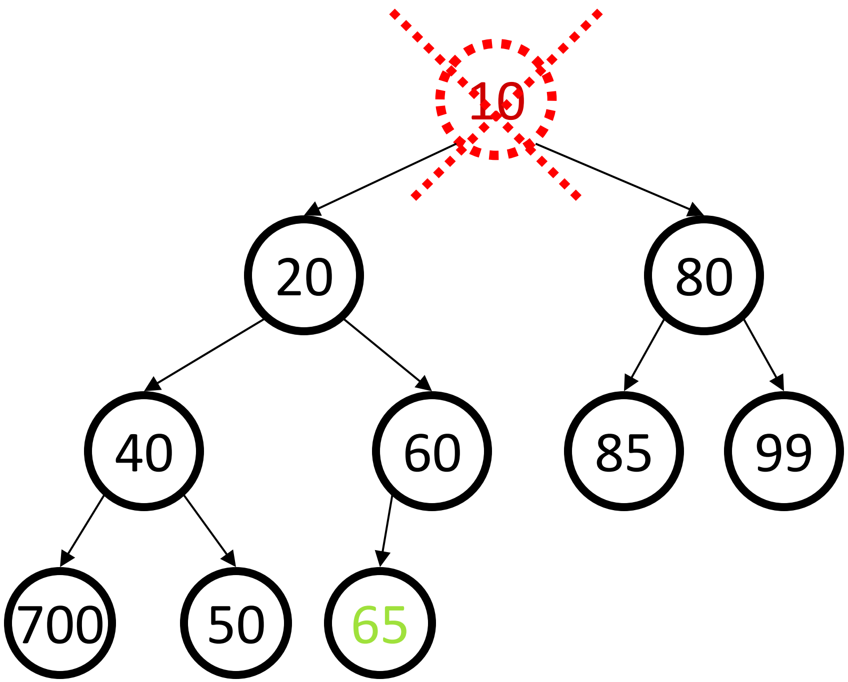
Modified add code

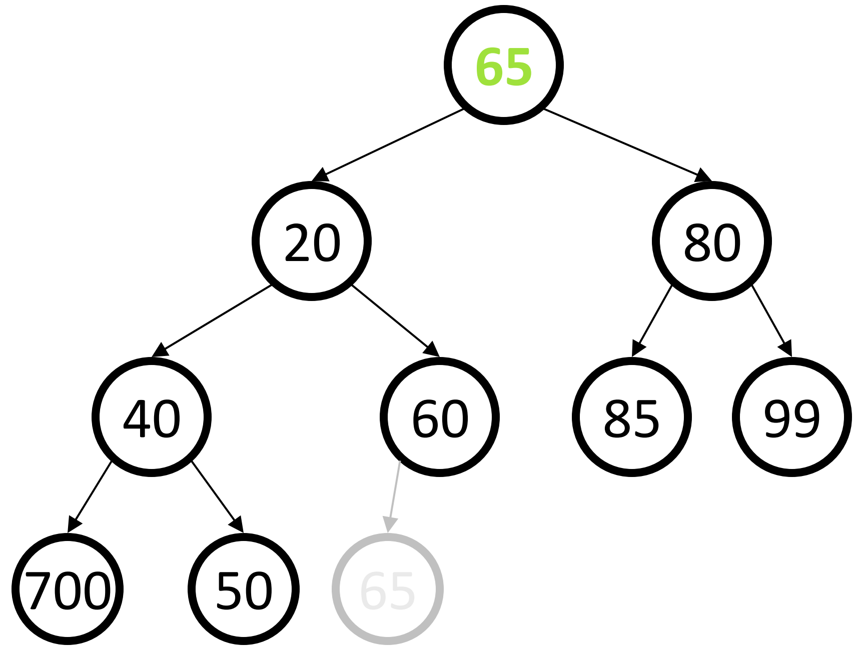
```
// Adds the given value to this priority queue in order.
public void add(int value) {
    // resize to enlarge the heap if necessary
    if (size == elements.length - 1) {
        elements = Arrays.copyOf(elements,
                                 2 * elements.length);
    }
    ...
}
```

Implementing 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)) {










    }
    return result;
}
```

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 smallerChild = left;
        if (hasRightChild(index) &&
                elements[right] < elements[left]) {
            smallerChild = right;
        }
        if (elements[index] > elements[smallerChild ]) {
            swap(elements, index, smallerChild );
            index = smallerChild ;
        } else {
            found = true;  // found proper location; stop
        }
    }
    return result;
}
```

Other Methods

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

// Returns true if the heap has no elements; false
 otherwise.
 public boolean isEmpty() {
   return size == 0;
 }
```

Questions?