CSE 373

Priority queue implementation; Intro to heaps

read: Weiss Ch. 6

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

Prioritization problems

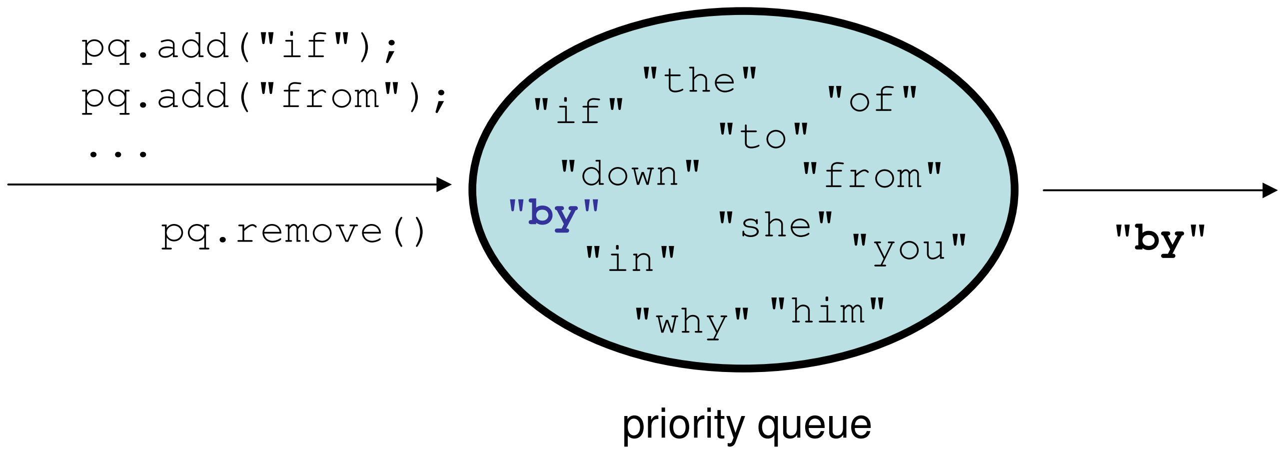
- **print jobs:** CSE lab printers constantly accept and complete jobs from all over the building. We want to print faculty jobs before staff before student jobs, and grad students before undergrad, etc.

- **ER scheduling:** Scheduling patients for treatment in the ER. A gunshot victim should be treated sooner than a guy with a cold, regardless of arrival time. How do we always choose the most urgent case when new patients continue to arrive?

- *key operations we want:*
  - ***add*** *an element (print job, patient, etc.)*
  - ***get/remove*** *the* ***most "important"*** *or "urgent" element

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)



Unfilled array?

- Consider using an unfilled 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									

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

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();
```

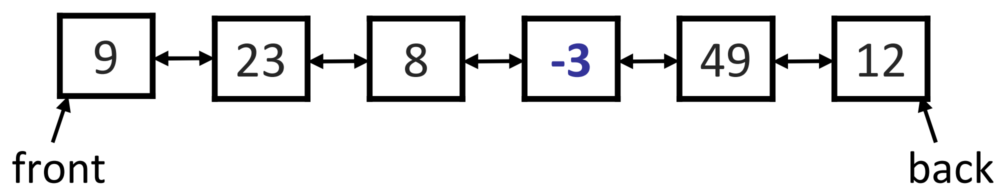
index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	**-3**	8	9	12	23	49	0	0	0	0
size	6									

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

Linked list?

- Consider using a doubly linked list to implement a priority queue.
  - `add`: Store it at the end of the linked list.
  - `peek`: Loop over elements to find minimum element.
  - `remove`: Loop over elements to find min. Unlink to remove.

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

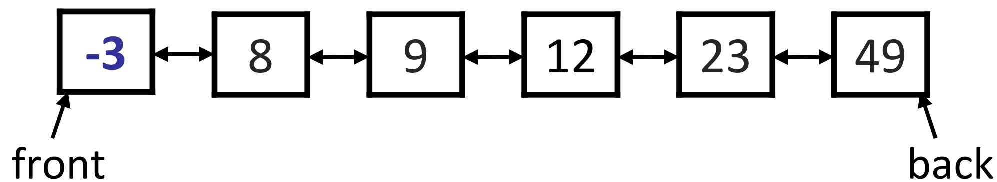


  - How efficient is `add`? `peek`? `remove`?
    - O(1), O(*N*), O(*N*)
    - (`peek` and `remove` must loop over the linked list)

Sorted linked list?

- Consider using a *sorted* linked list to implement a priority queue.
  - `add`: Store it in the proper place to maintain sorted order.
  - `peek`: Minimum element is at the front.
  - `remove`: Unlink front element to remove.

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

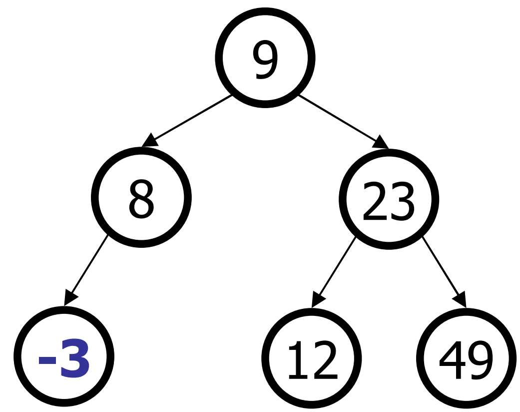


  - How efficient is `add`? `peek`? `remove`?
    - O(*N*), O(1), O(1)
    - (`add` must loop over the linked list to find the proper insertion point)

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.

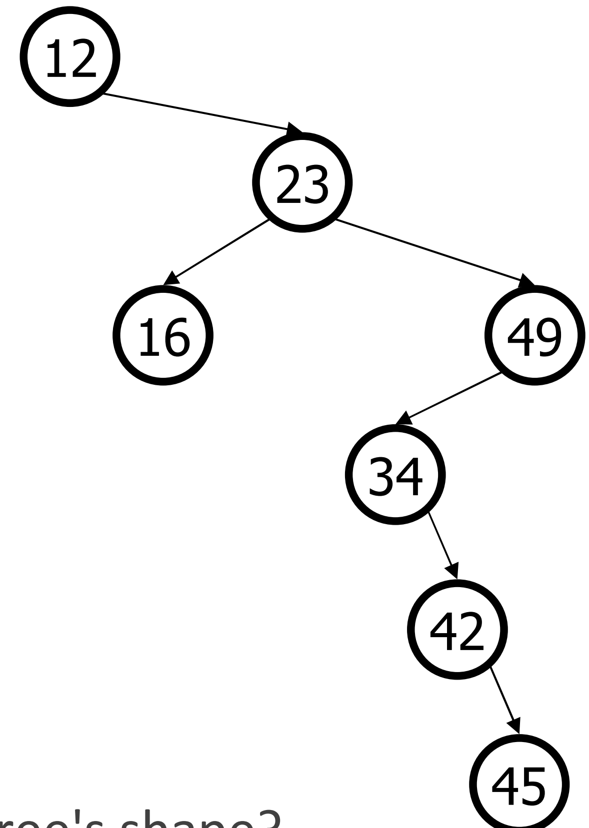
```
queue.add(9);
queue.add(23);
queue.add(8);
queue.add(-3);
queue.add(49);
queue.add(12);
queue.remove();
```



  - How efficient is `add`? `peek`? `remove`?
    - O(log *N*), O(log *N*), O(log *N*)...?
    - (good in theory, but the tree tends to become unbalanced to the right)

Unbalanced binary tree

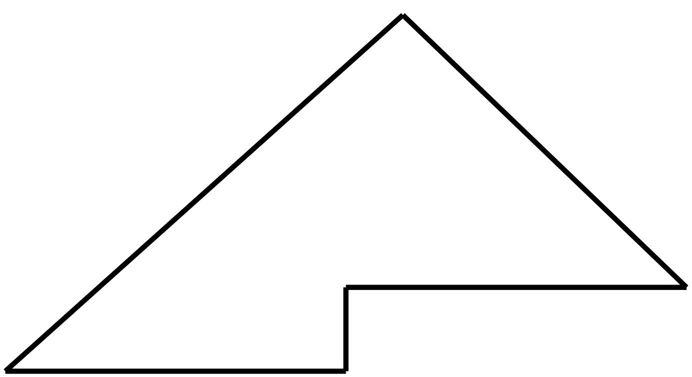
```
queue.add(9);
queue.add(23);
queue.add(8);
queue.add(-3);
queue.add(49);
queue.add(12);
queue.remove();
queue.add(16);
queue.add(34);
queue.remove();
queue.remove();
queue.add(42);
queue.add(45);
queue.remove();
```

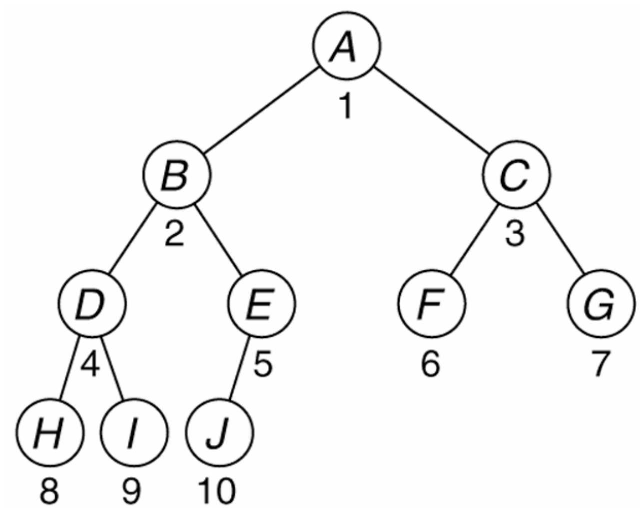


- Simulate these operations. What is the tree's shape?
- A tree that is *unbalanced* has a height close to *N* rather than log *N*, which breaks the expected runtime of many operations.

Heaps

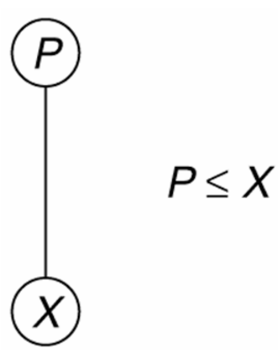
- **heap**: A *complete* binary tree with *vertical* ordering.
  - **complete tree**: Every level is full except possibly the lowest level, which must be filled from left to right
    - (i.e., a node may not have any children until all possible siblings exist)

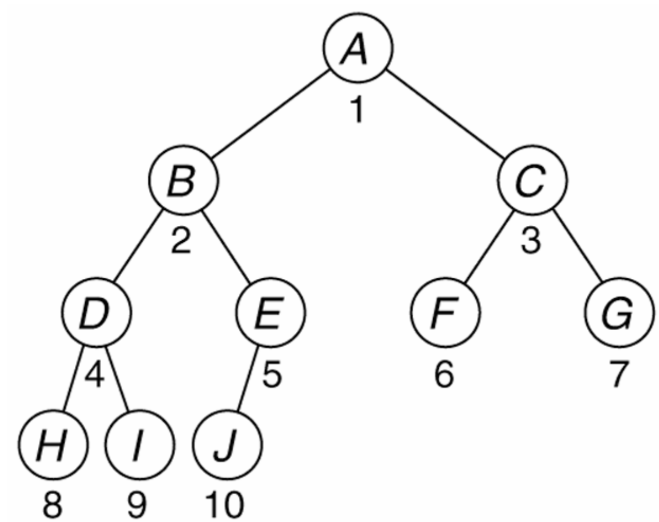




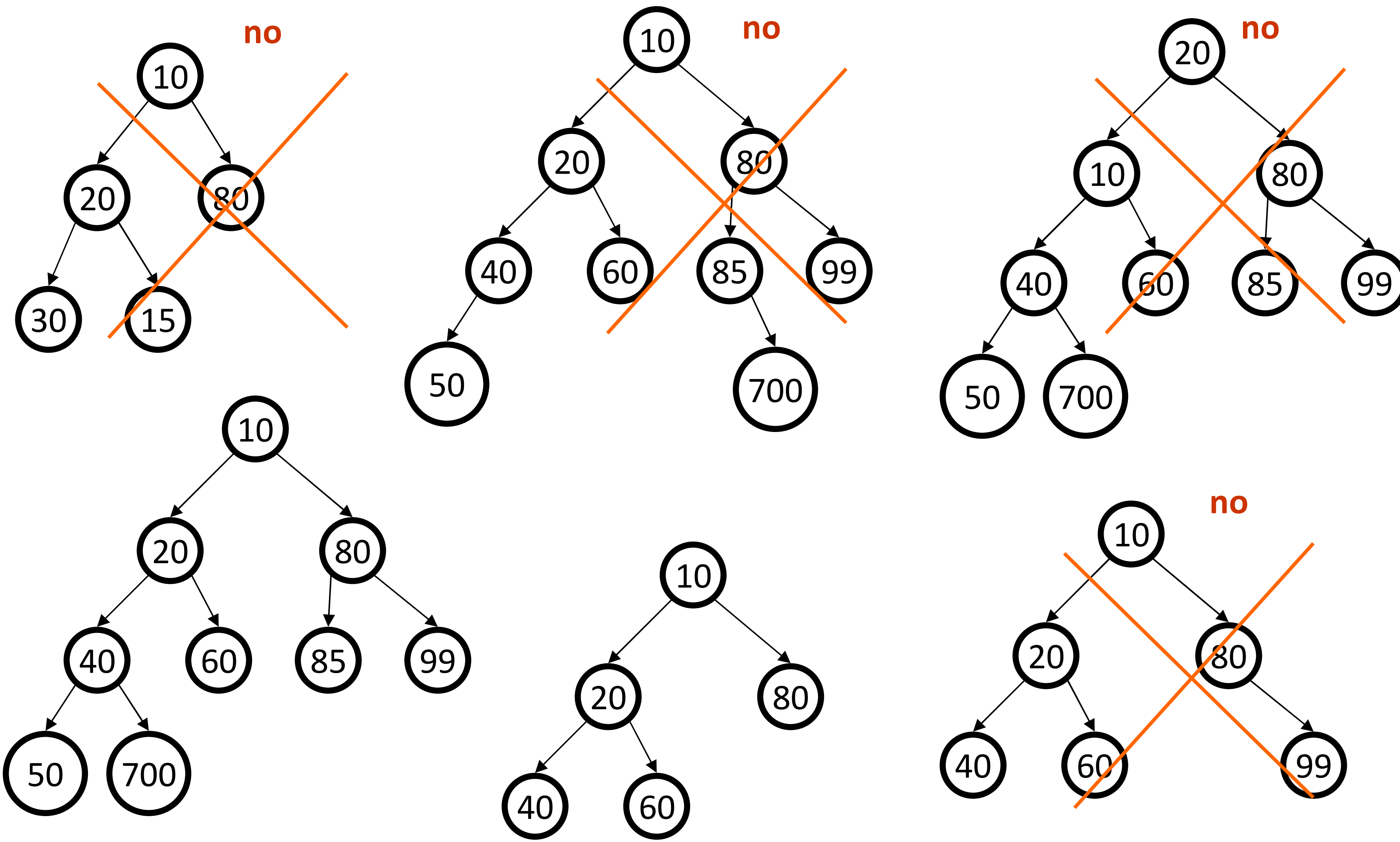
Heap ordering

- **heap ordering**: If P ≤ X for every element X with parent P.
  - Parents' values are always smaller than those of their children.
  - Implies that minimum element is always the root (a "min-heap").
    - variation: "max-heap" stores largest element at root, reverses ordering
  - Is a heap a BST? How are they related?

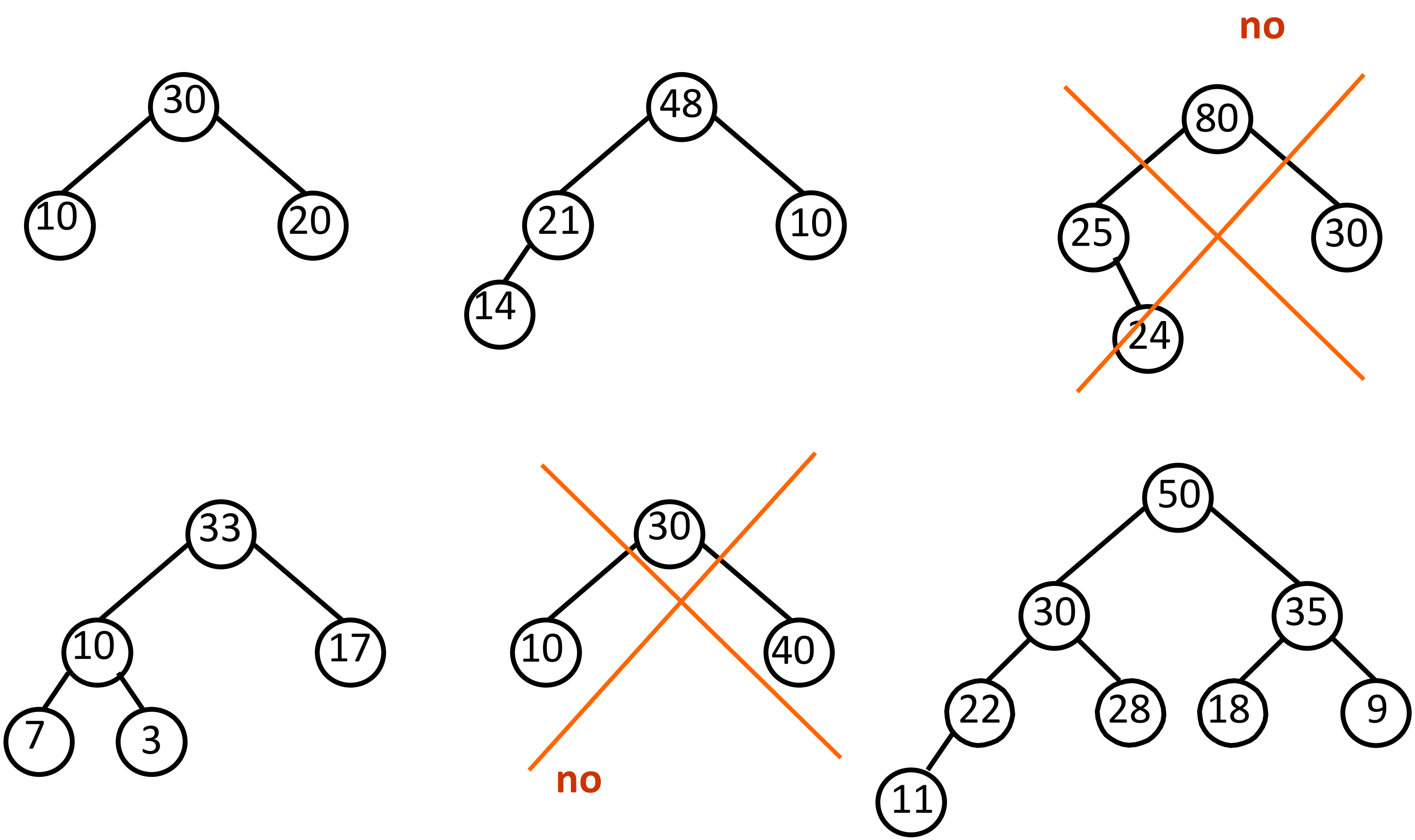




Which are min-heaps?



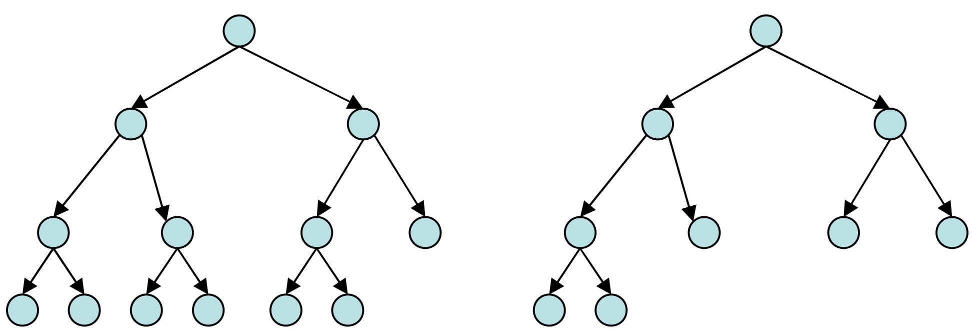
Which are max-heaps?



Heap height and runtime

- The height of a complete tree is always log *N*.
  - How do we know this for sure?

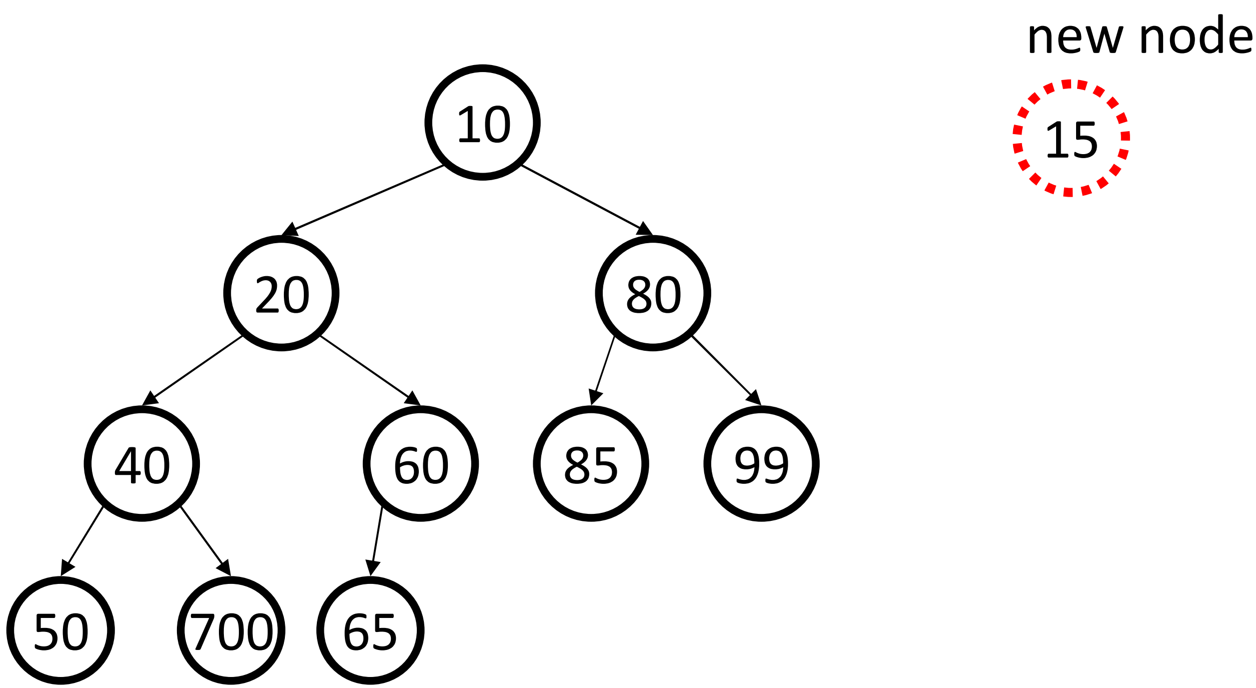
- Because of this, if we implement a priority queue using a heap, we can provide the following runtime guarantees:
  - `add`: O(log *N*)
  - `peek`: O(1)
  - `remove`: O(log *N*)



n-node complete tree
of height h:
$2^h \leq n \leq 2^{h+1} - 1$
$h = \lfloor \log n \rfloor$

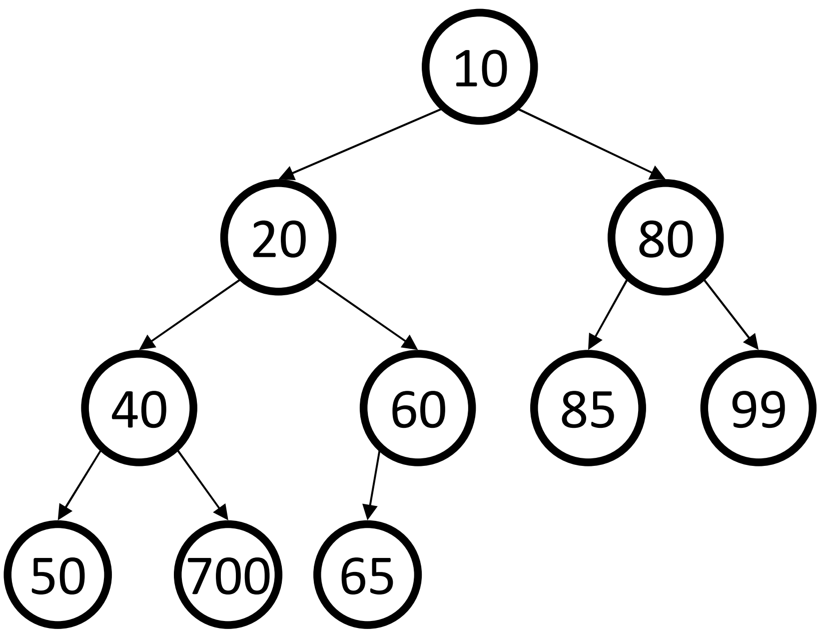
The add operation

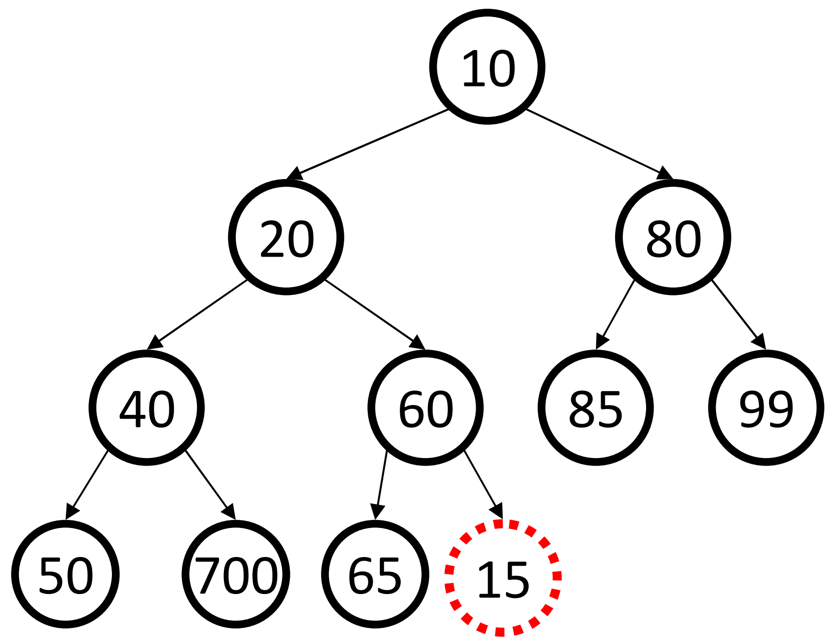
- When an element is added to a heap, where should it go?
  - Must insert a new node while maintaining heap properties.
  - `queue.add(15);`



The add operation

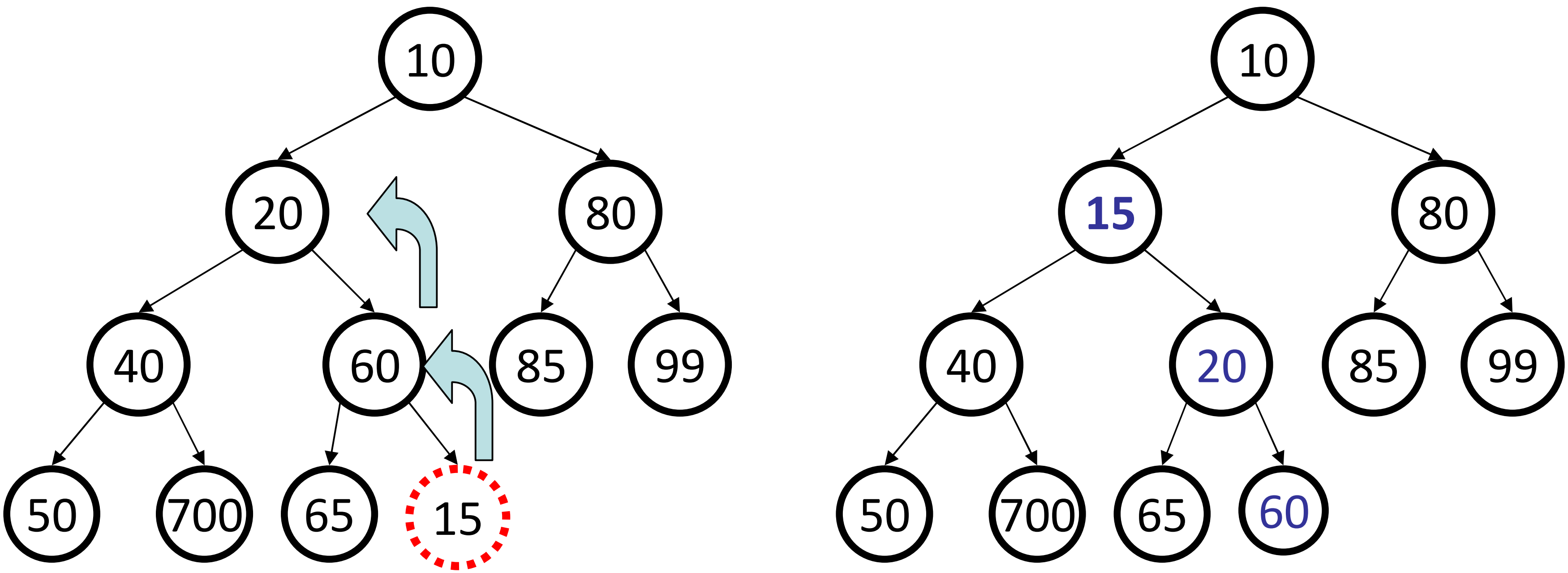
- When an element is added to a heap, it should be initially placed as the *rightmost leaf* (to maintain the completeness property).
  - But the heap ordering property becomes broken!





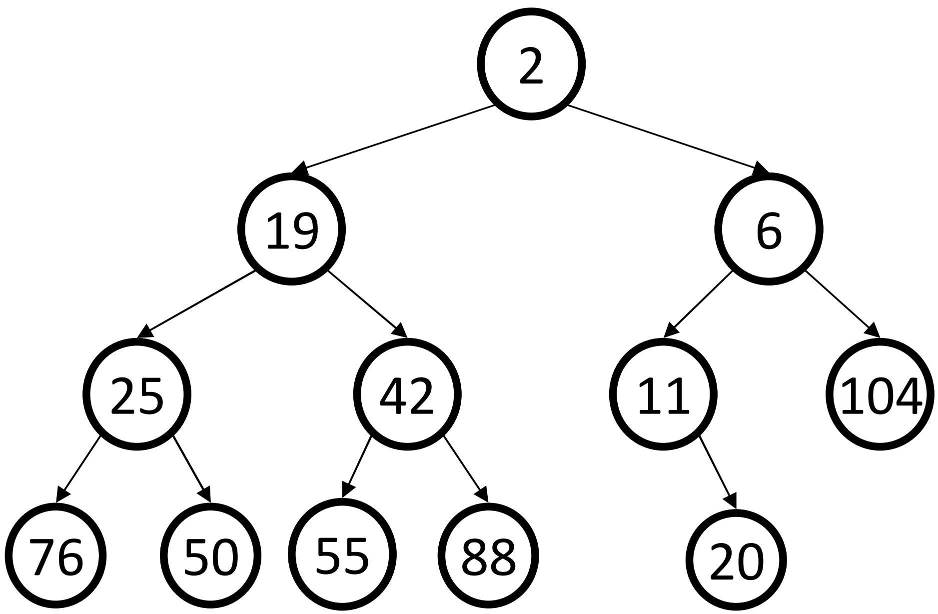
"Bubbling up" a node

- **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?



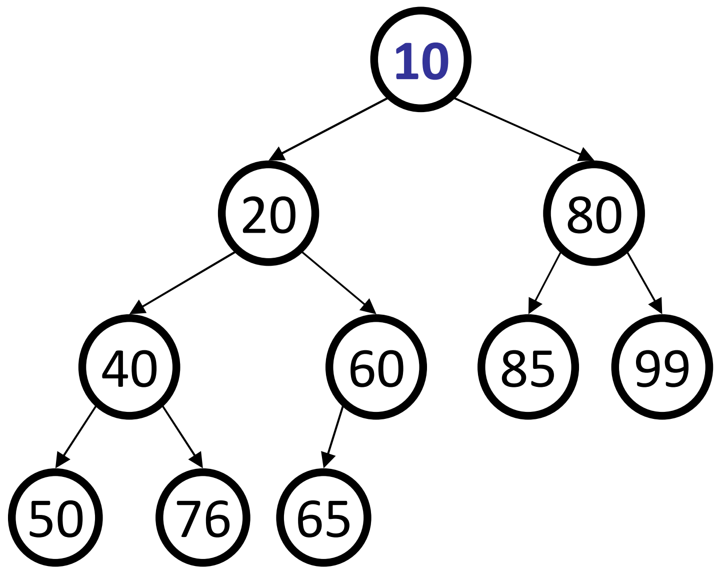
Bubble-up exercise

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



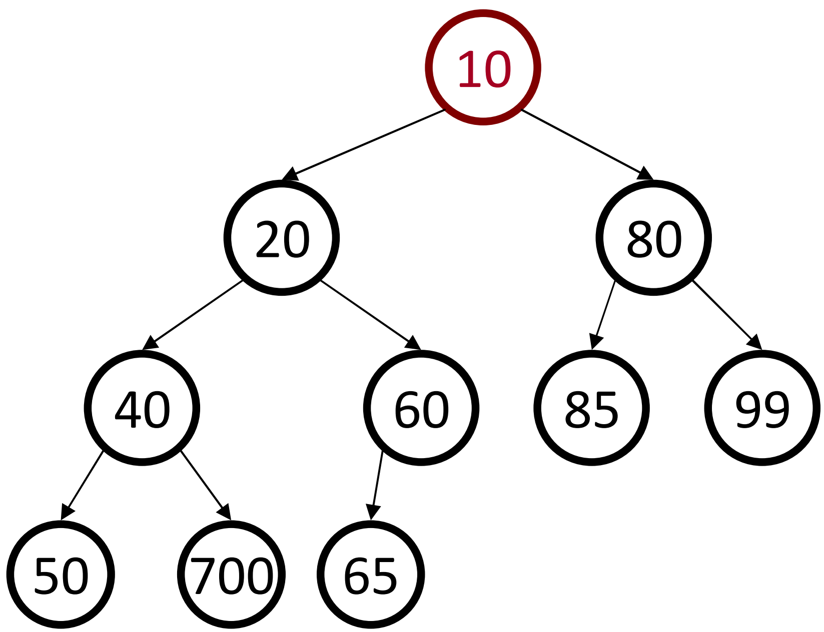
The peek operation

- 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)



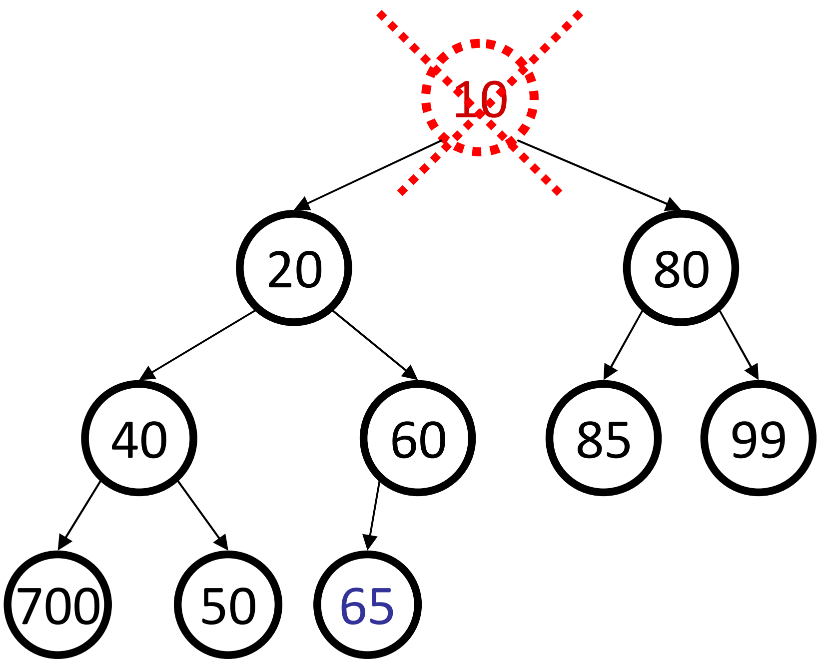
The remove operation

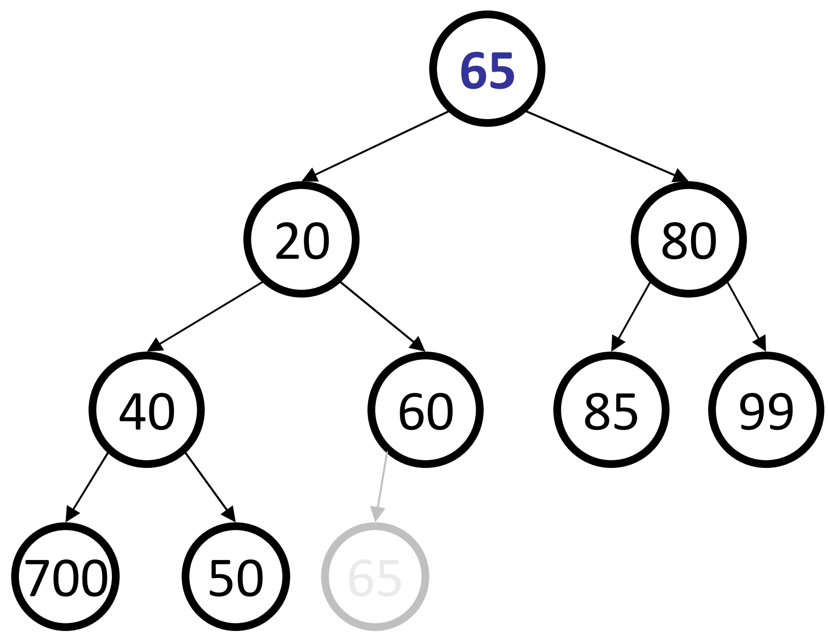
- When an element is removed from a heap, what should we do?
  - The root is the node to remove. How do we alter the tree?
  - `queue.remove();`



The remove operation

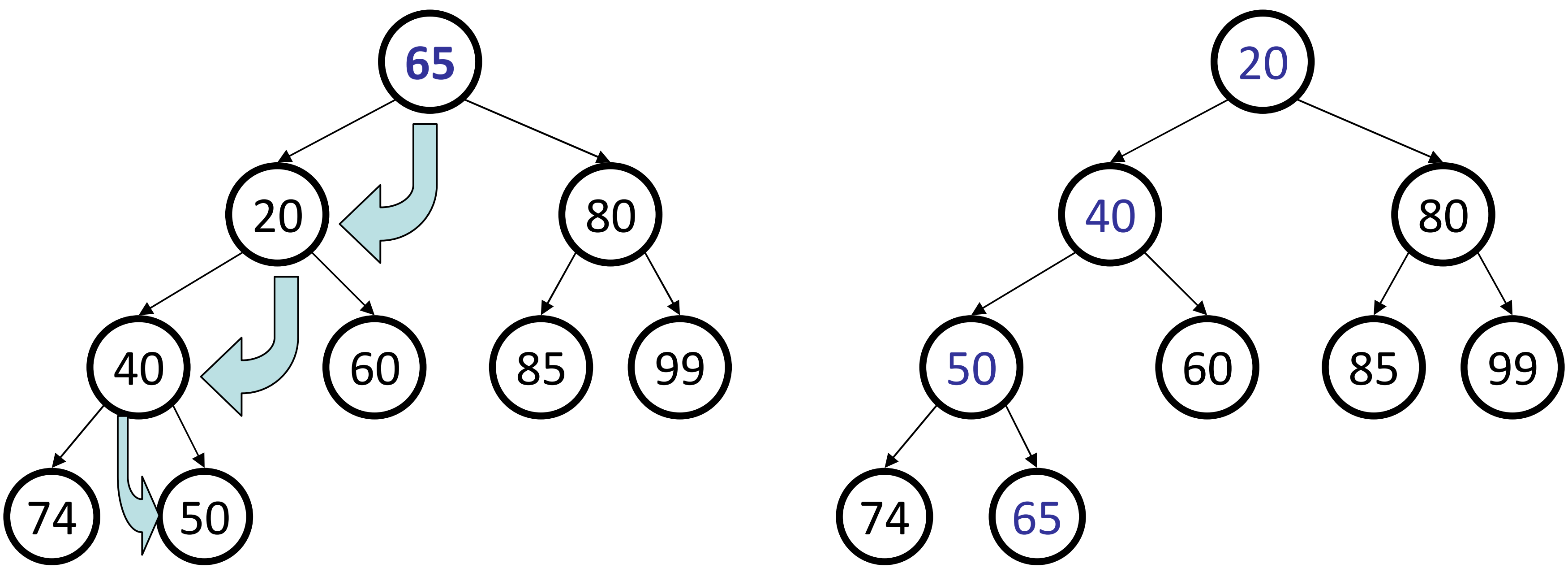
- When the root is removed from a heap, it should be initially replaced by the *rightmost leaf* (to maintain completeness).
  - But the heap ordering property becomes broken!





"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.