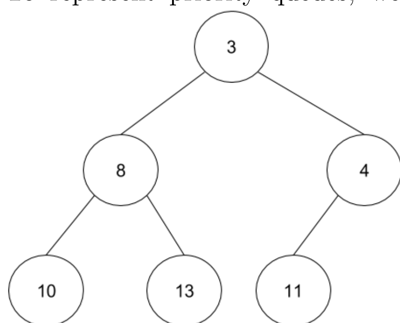
Priority Queues

1 Priority Queues: Heaps

Sometimes instead of just storing data, we want to be able to get them in a certain order. To do this, we can use a ***priority queue***. The API for a minimum priority queue is as follows:

```
1 public interface MinPQ<Item> {
2         public void add(Item x);
3         public Item getSmallest();
4         public Item removeSmallest();
5         public int size();
6 }
```

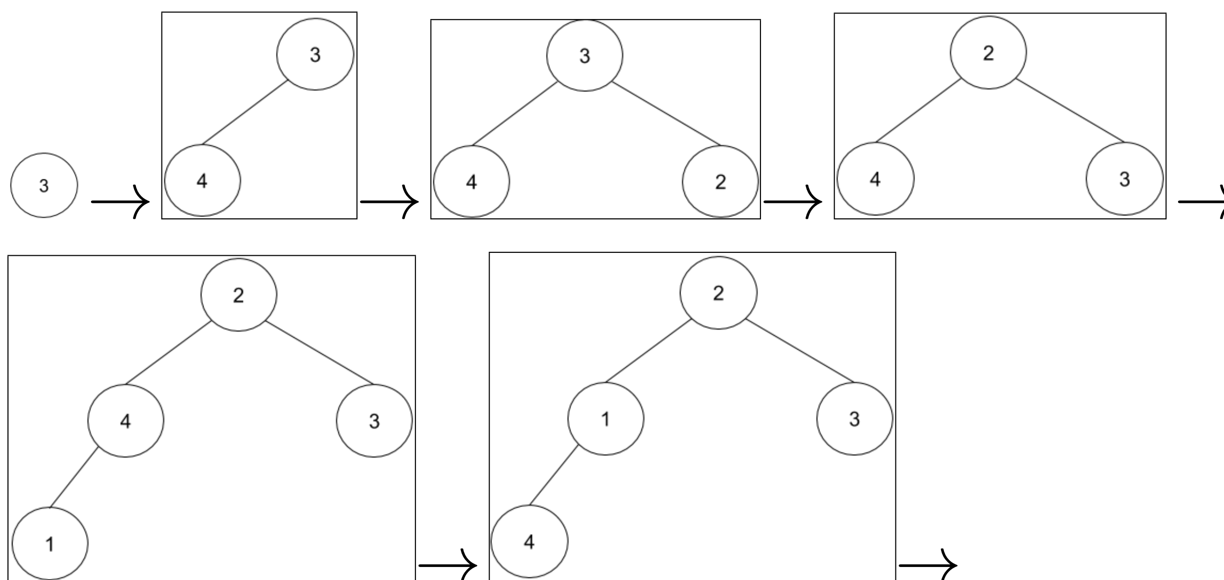
To represent priority queues, we can use a tree like structure called a heap. A heap looks like this:

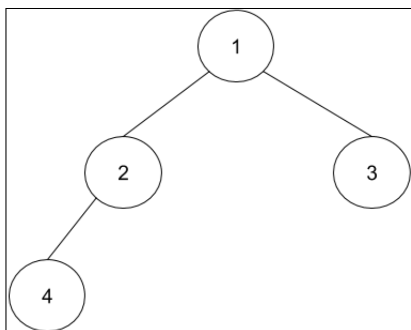


This above heap is a min-heap, that is the minimum element is the item on the top. There are a few rules for min-heaps.

- Any child must be greater than its parent.
- The "tree" structure must always be filled from top to bottom and left to right.
- Any node can have a maximum of two children.

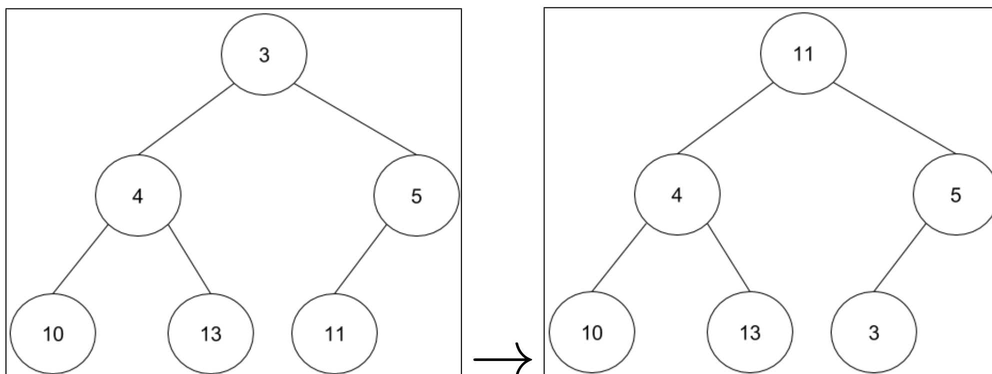
Now we will build a heap from scratch for the input as follows: 3, 4, 2, 1.



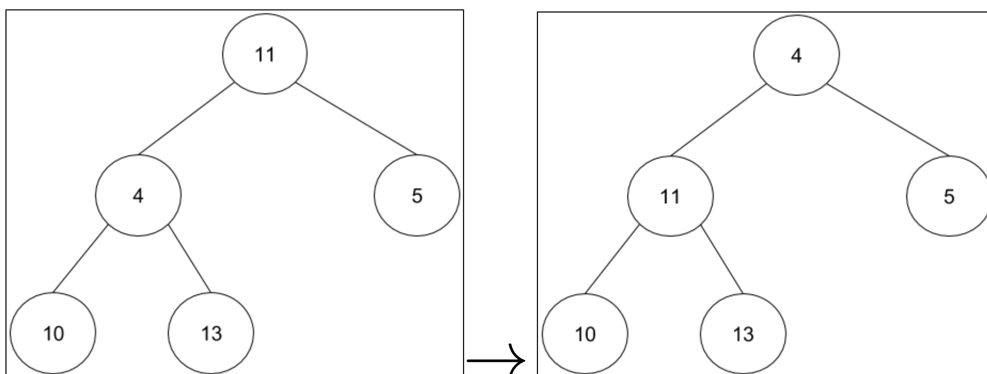


Note how when we find an element that is less than its parent, we "swim" it up and switch it with its parent. Now, let's go over how we can delete from a min-heap. For priority queues in general, we only remove the minimum element, or the element with the highest priority.

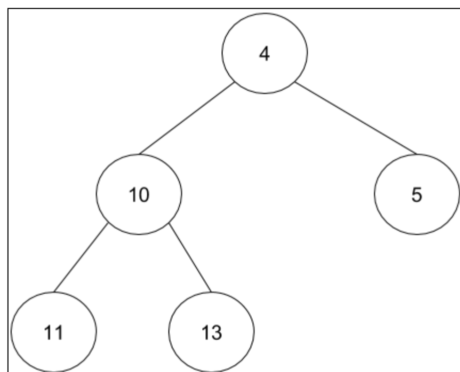
Now that we know how to add to a heap, we will go over how to delete from them. In priority queues, it is only sensible to delete the item with the maximum priority. In order to do so, we do a somewhat unintuitive approach, but one that must be done. We will start with the following heap:



You may be asking yourself, how did we get to this point? Well, when deleting the item with the highest priority, we switch the root with the item that is the rightmost item on the lowest level.



At this point, we delete the item with the highest priority (which we know is the rightmost item on the lowest level). Following this, we sink the root node. We check to see if the root is less than one, or both, of its children, if so, we choose the smallest child and swap it with the root. We then repeat this step with the same node until we cannot swap the item anymore.



We finally swap 10 and 11 because 10 is the only child of the node 11 that is less than it.

Since heaps have a consistent ordering. We can represent them in an arraylike fashion using a formula: specifically, the child of a node will always be either $2k$ or $2k + 1$ where k is the parent index. The assumption being made is that the root is always at index 1. This means that 1's children will be 2 and 3, 2's children will be 4 and 5, 3's children will be 6 and 7, and so forth. *getSmallest* takes $\Theta(1)$, as it will always be the root. *add* and *removeSmallest* however, take the height of the tree- which is $\Theta(log(n))$.