Chapter 4

Heaps

4.1 Introduction

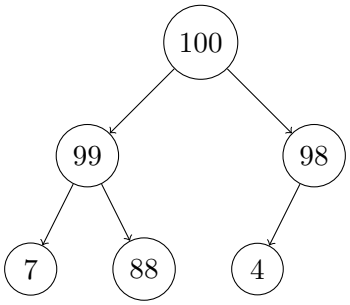
The **heap** is a data structure typically used to keep track of the max or minimum of a set of elements. They are typically used when implementing priority queues because they can keep track of the highest (or lowest) priority easily and efficiently. As such, the only real similarity I can draw between the data structure and its name is the appearance of its representation.

4.2 Description

Because the **heap** is typically drawn as a tree, it will use similar terminology that can be found in the binary search tree pdf.

The one relationship of a **heap** is that the parent node is always greater (or less depending on the kind of **heap**) than its children. There is no strict relationship between any other nodes aside from the fact that the relationship is transitive and it could be said that a descendant node will be less than its ancestor nodes.

When drawn as a tree, a **heap** should always be a complete binary tree. If you don't remember what that means, it means everything is filled out except for the last level although that level must be filled starting from the left. This structure is something that will remaing even after insertion and deletion. For example, a **max heap** could look like this:



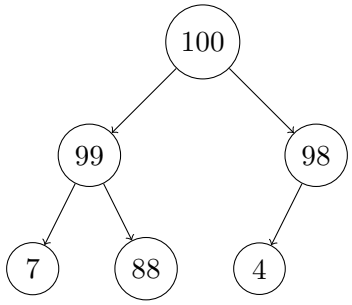
As we can see, for any pair of parent-child nodes, the parent's value is always greater than the child's. For all the future examples, we'll be using a **max heap**.

4.3 Searching

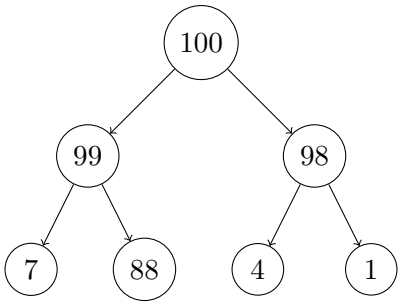
Remember that a the only relationship that exists in a **heap** is the one between a parent and child node. How does this affect searching? At a glance, we can easily find the max because it is the root, but what about other values? You will actually have to traverse most of the tree to determine whether a specific value exists or not, so there is not a benefit to searching for a specific value.

4.4 Insertion

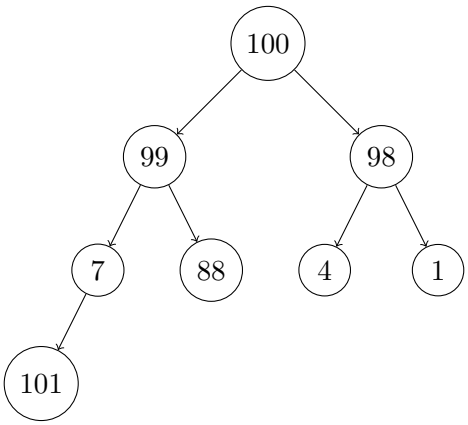
Ok, remember that we want our **heap** to stay as a complete binary tree. So where should we add a node to maintain this? At the lowest level as far left as we can. Consider the previously shown **heap**:



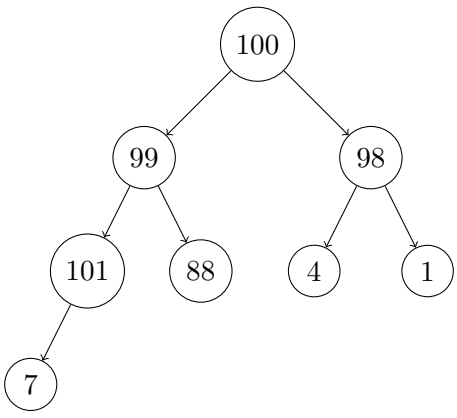
Let's insert 1 into the **heap**. It will go on the lowest level in the first free spot.



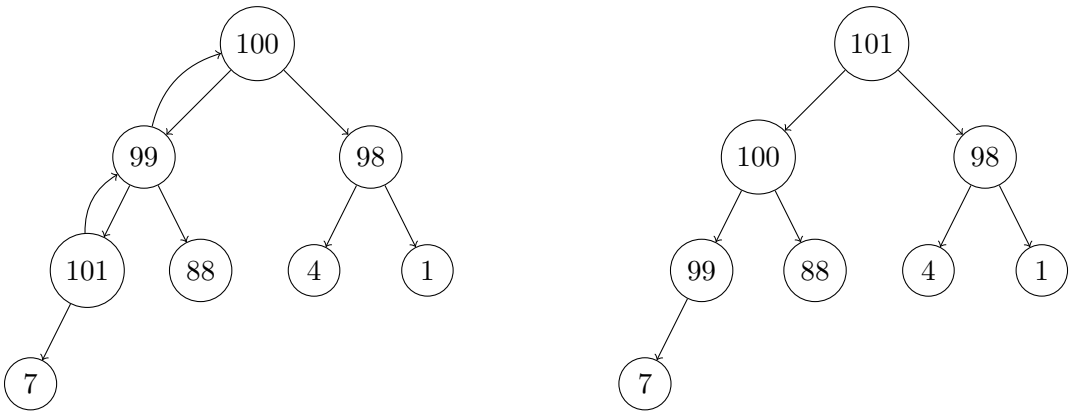
Well that was easy. But wait, what if we had to insert a number that will end up breaking the parent child relationship? We just have to do a few extra steps; let's try inserting 101 this time.



Alright, so now we notice the relationship is broken so let's just swap 101 and 7. What an easy fix!



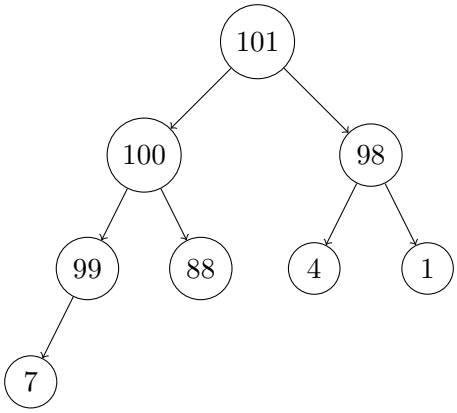
Uh oh, the relationship is still not satisfied! ($99 < 101$). Just keep swapping upwards until you're good!



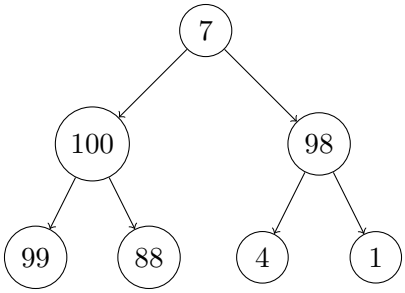
This movement of the node upwards is commonly called **bubbling up**.

4.5 Deletion

Deletion is only a tad more difficult. Unlike other data structures, when you delete from a **heap** (remove would be a better word), you always delete the root. Let's try deleting from the previous example:

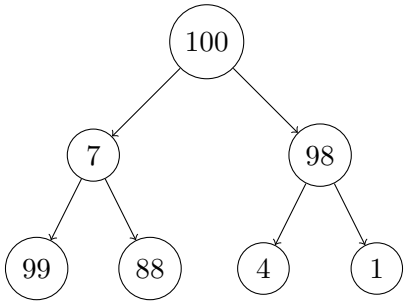


So we'll be deleting 101. But remember we want to keep the completeness of the tree! So let's take that last node and put it at the root.



We have a complete tree but something seems off... the relationship is broken again! This time, instead of **bubbling up**, we will **bubble down**. 7 is obviously out of place so we should move it down, but which way should it go? We want to satisfy the relationship for these 3 nodes so which of the following arrangements looks good?

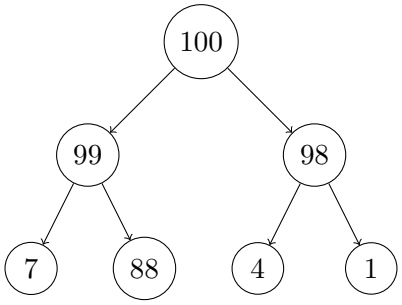
The left one does fixes everything, so it looks like the rule is swap with the maximum of the two children in this case!



We end up with this but we're still not done; I'll leave the rest as an exercise to the reader. Haha. Well obviously you swap 7 and 99 and then you're finished.

4.6 Array-Based Heap

Arrays... what? Interestingly enough, **heaps** are typically implemented in arrays instead of having node objects and linking them together with pointers! Let's go back to our previous **heap** (because it's easier to copy and paste than to make a new example):



All we have to do is dump these contents into an array left to right, top to bottom. We'll end up getting

$$100 \rightarrow 99 \rightarrow 98 \rightarrow 7 \rightarrow 88 \rightarrow 4 \rightarrow 1$$

Let's try to find a way to determine the index of a child if we have the index of the parent. The index of 100 is 0, it's children have indicies 1 and 2; The index of 99 is 1, it's children have indicies 3 and 4; and so on After staring at the numbers multiple times, you might notice the following:

$$parent = i; children = 2i + 1, 2i + 2$$

So if you know the parent's index, you can get the children with this handy formula. And likewise

$$child = i; parent = (i - 1)/2$$

But what about decimals? Use integer division!

4.7 Examples

Removed for now.