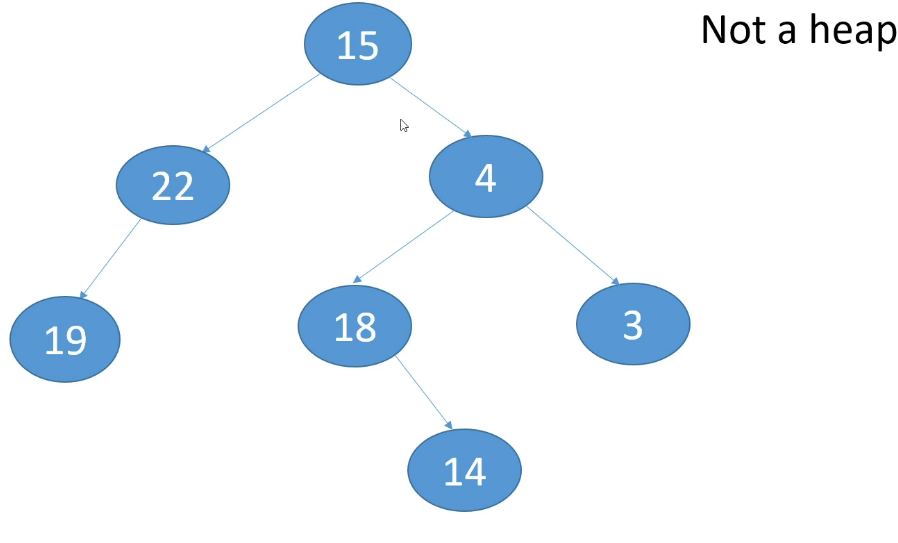
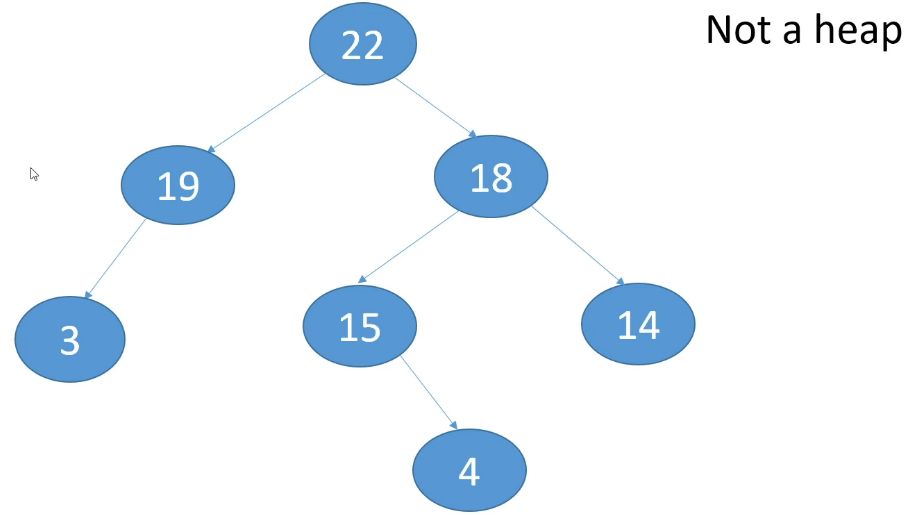
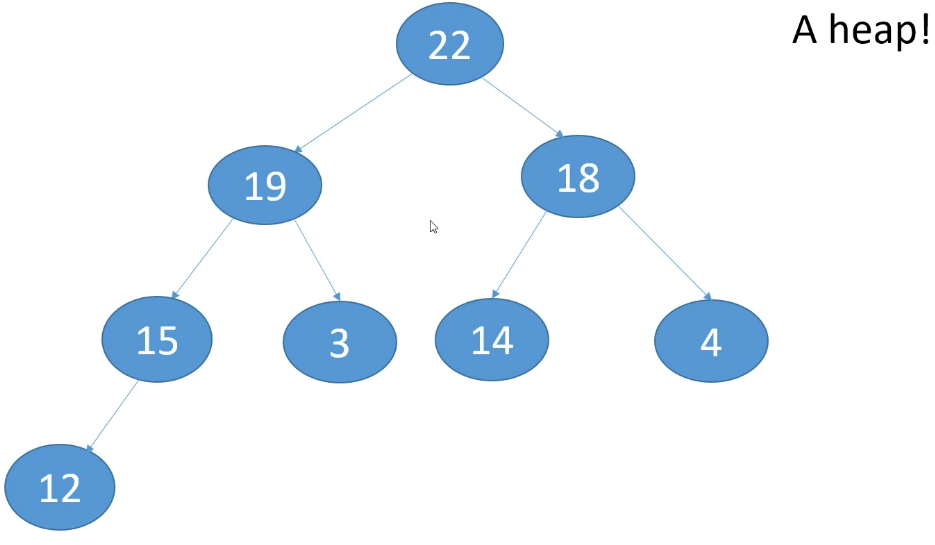
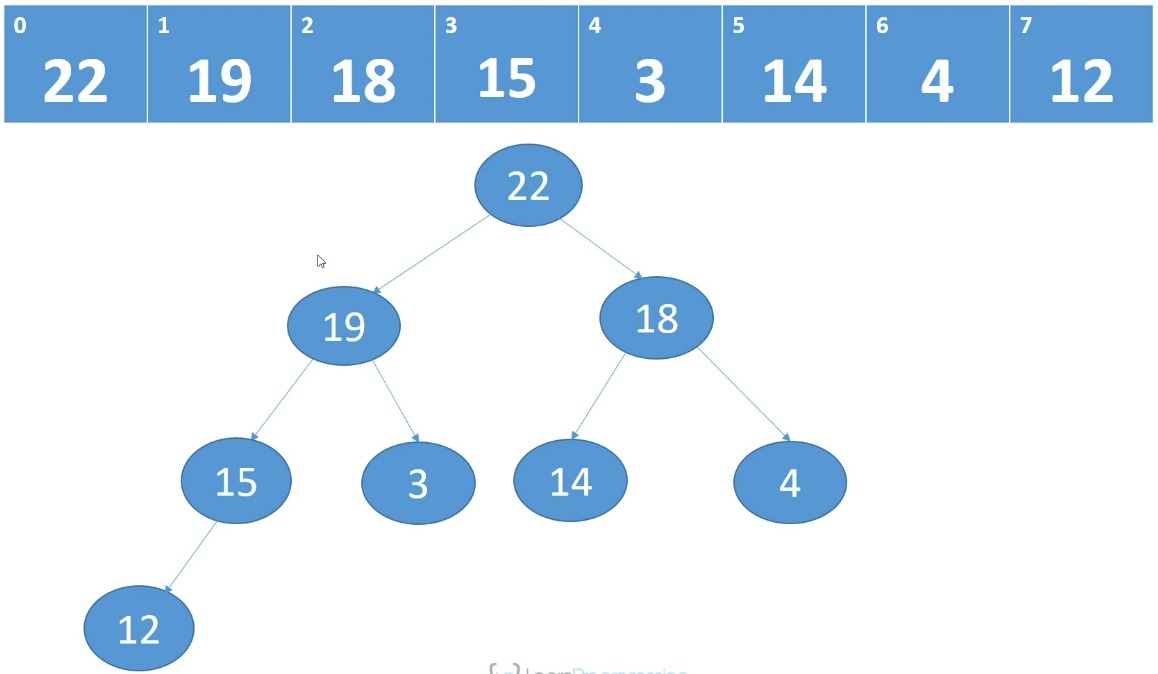
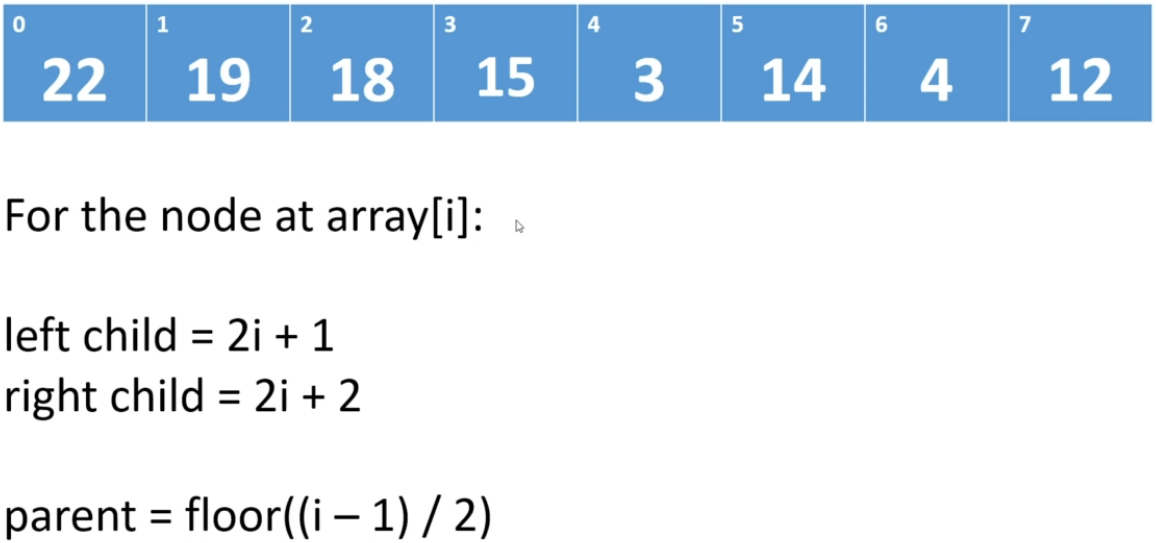
**Introduction to Heaps**  
\* **Heaps have nothing to do with memory**.  
\* You may have heard the term Heap discussed in relation to memory but this has absolutely no relation to that.  
\* **Heap** is a **special type of** **Binary Tree**.

**Heaps (Theory)**  
\* **Binary Heap** is a **Complete Binary Tree that satisfies the heap property**.  
**1) Requirement**  
**Complete** **Binary** **Tree** => **every level of the tree is full except potentially the last level where if it’s not complete - if there’s any space at the bottom level, then the existing Leaves all have to be as far to the left as possible.  
2) The Heap property requirement**  
**Max heap**: **every parent is >= children**  
**Min heap**: **every parent is <= children**  
\* We call them Binary Heaps because they are Binary Trees.  
\* **Children are added at each level from left to right**.  
 => If we were adding into an empty Heap, the first value would become the root, the next value would become the left child, the 3rd value would become the right child of the root, and then we would move down to the next level and so the 4th value would be come the left child of the of the left child of the root. And the 5th value would become the right child of the left child of the root.  
\* **Usually implemented as arrays**.  
=> We saw an implementatin of a Binary Tree that uses classes, that uses a Tree class and a TreeNode class.  
=> When you have a Complete Binary Tree, you can actually back them by arrays.  
=> That’s how Heaps are usually implemented.  
\* **The maximum or minimum value will always be at the root of the tree - the advantage of using a heap**.  
=> **That’s why Heaps exist**.  
=> You can get the min() or max() in **O(1) because accessing the root is a constant time operation.**  
**Heapify - the process of converting a Binary Tree into a Heap - this often has to be done after an insertion or deletion**.  
\* When we insert a node into a tree, we generally add it to the bottom level because when you’re building a tree, you start at the top and then you move to the next level and add nodes left to right. And so when we add a node to an already existing tree, we add it at the first available spot at the bottom level.  
=> But once we’ve done that, **the tree might no longer meet the Heap property**.  
=> So we have to fix the tree.  
\* **No required ordering between siblings**.  
=> So when you have nodes at the same level, they don’t have to be in ascending or descending order.  
\* The important relationship when it comes to heaps is the relative values between parents and children.

   
1) It’s not a Complete Binary Tree, 22 is missing a child.  
2) Doesn’t meet the heap property, the parent-child relationships.  
  
1) It’s not a Complete Binary Tree, 19 is missing a child.  
  
\* Heap => it’s a Complete Binary Tree and it meets the **Max Heap** property.  
\* One interesting characteristic of a Map Heap is if you travel **from the Root down to all the leaves, all the values along the path are in descending order**.  
\* The opposite would be true if you started at a Leaf and traveled up to the Root.  
=> **If you want to check whether something is a Heap, try to find a path for which that isn’t true**.  
\* The Min Heap would work similarly but the orders would be reversed.  
\* **You can back Complete Binary Trees using an array**.

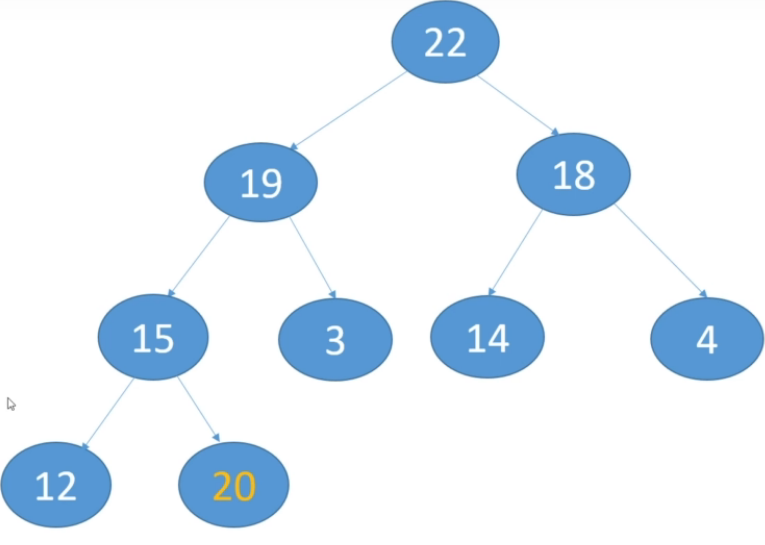
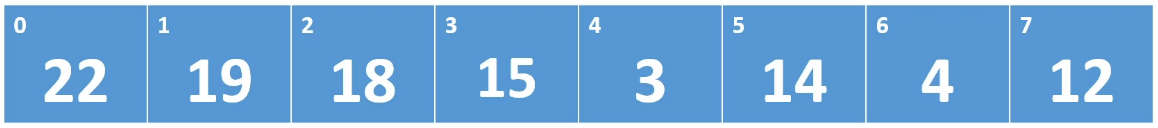
**Storing Heaps as Arrays**  
\* **You can store any Complete Binary Tree as an array**.  
=> **We put the root at array[0]**  
=> We then traverse each level from **left to right**  
=> **left child** => **array[1]**  
=> **right child** => **array[2]**

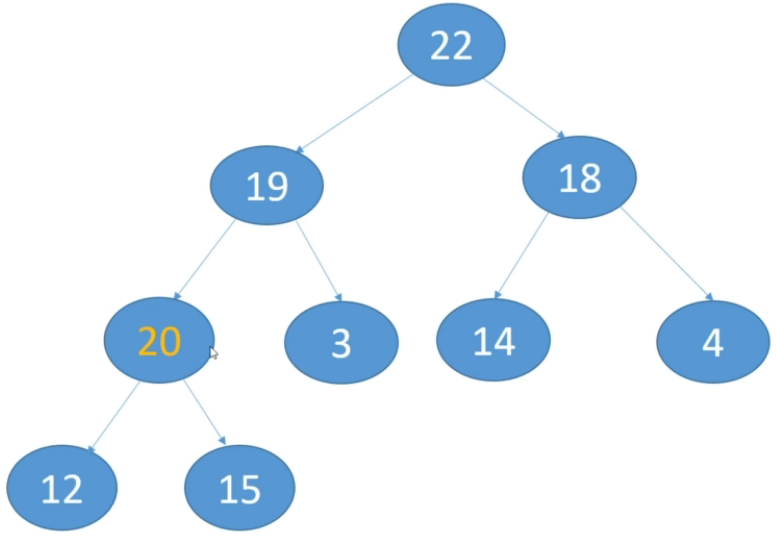
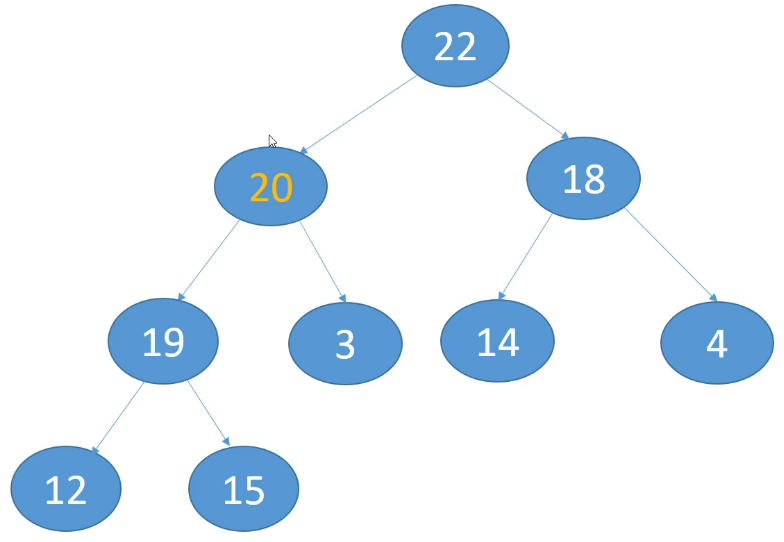
=> the left child of the left child => array[3]  
=> right child of the left child => array[4]

  
\* This is how we figure out the parent and the children for any slot in the array:  


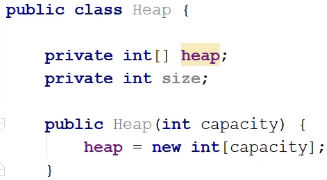
\* So for 15:  
left child = 2i + 1 = 2 \* 3 + 1 = 7  
right child = 2i + 2 = 2 \* 3 + 2 = 8, out of bounds so it doesn’t have a right child  
parent = floor((i - 1) / 2) = floor((3 - 1) / 2) = floor(2 / 2) = floor(1)

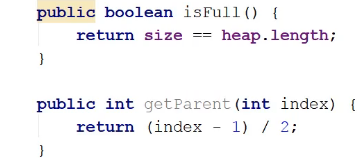
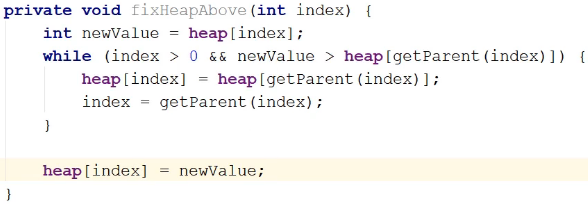
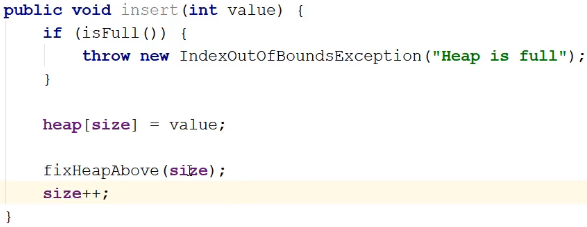
\* **This is how we store Complete Binary Trees in an array**.  
\* **It’s very important to understand and remember that we can only do this with Complete Binary Trees**. Because otherwise this wouldn’t work.  
=> **This works because there are no empty spots in the Tree**.  
=> **And so we can just traverse each level in turn and fill out the array**.  
**Insert into Heap**  
=> **Always add new items to the end of the array**  
=> Then we have to fix the heap (heapify)  
=> We compare the new item against its parent  
=> If the item is greater than its parent, we swap it with its parent  
=> We then rinse and repeat

\* Let’s add 20:  
 

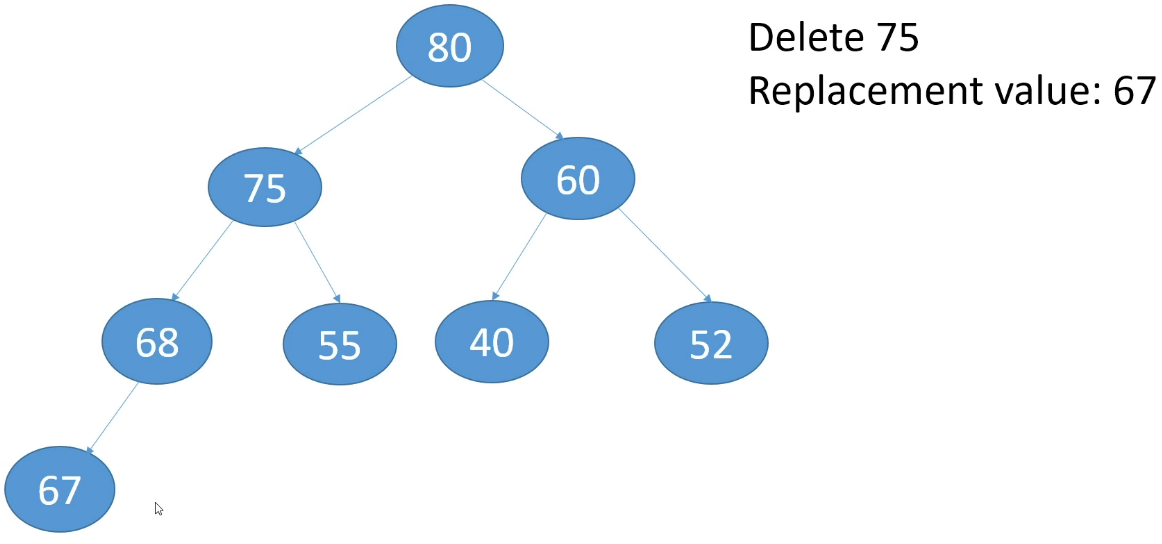
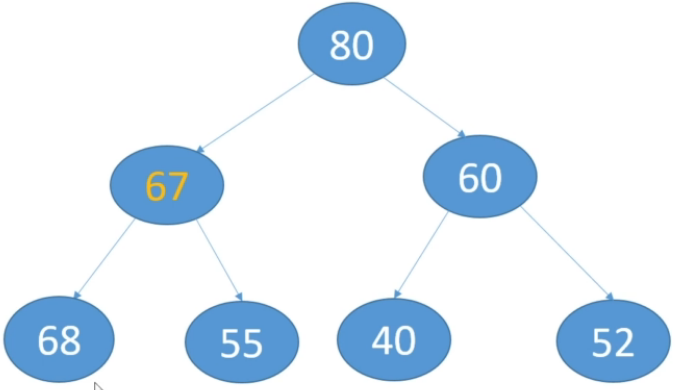
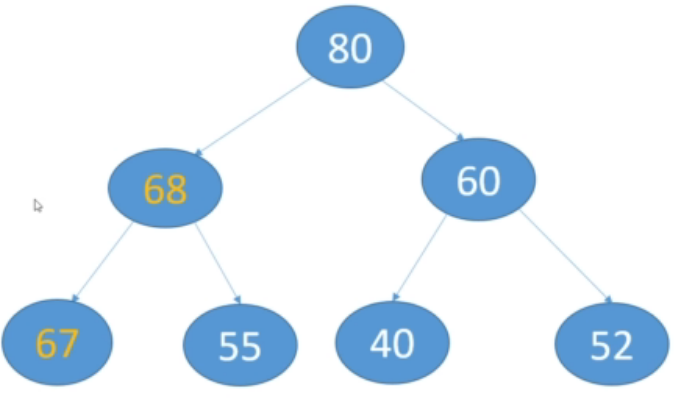
**1) Compare the new value against its parent**.  
=> This is not a Heap anymore because 20 > 15 parent.  
**2) If the value > parent, swap them.**=> 20 > 15, swap them.  
**3) Repeat.**  
=> 20 > 19, swap them.  
 

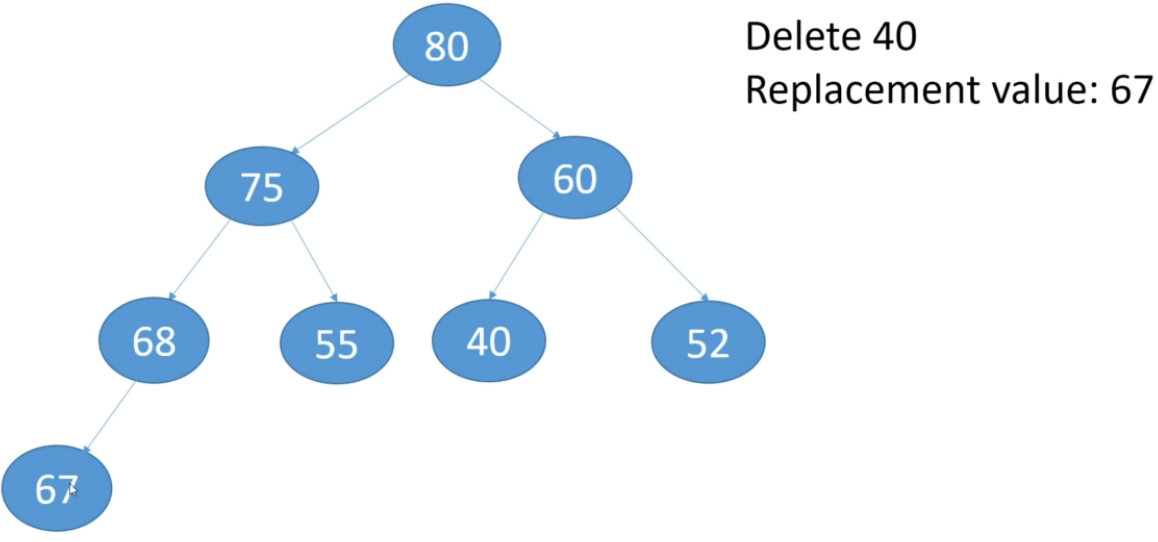
=> This is now a Heap again.  
\* The characteristic of the min/max Heap is maintained when we swap because every time we’re swapping a greater value with its parent and so we know that after the swap, the parent will have a greater value than the child.  
\* We also have to heapify after deleting a value and in that case, it would be a little bit more involved because it depends on which node we’re deleting - where it’s located in the tree.

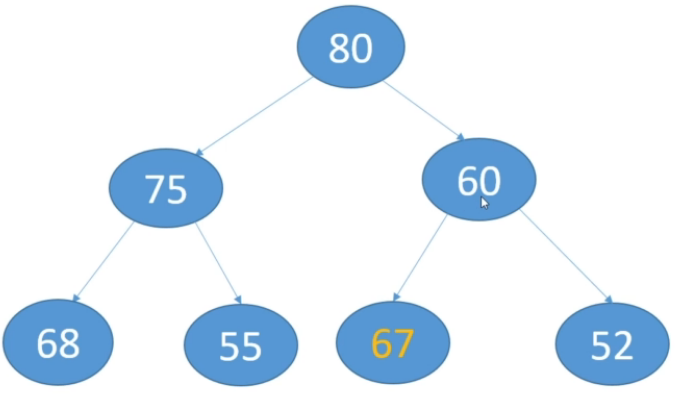
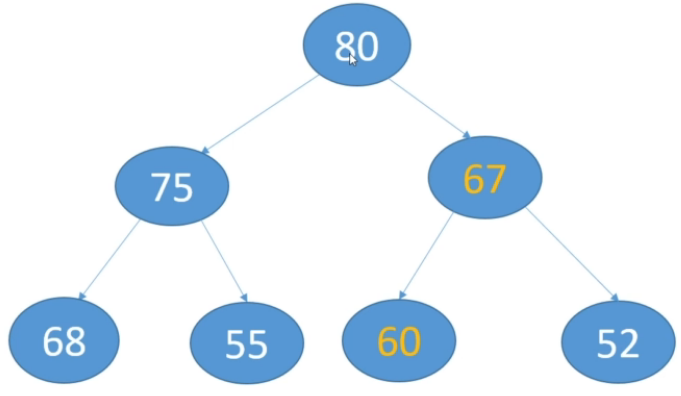
**Heaps (Insert)**  
\* We’re going to implement a **Max Heap**.  
=> **parents >= children**.  
\* **Heaps are usually implemented as arrays** and that’s how we’re going to do it.  
\* If a heap is full, we can either **throw an exception** or we could **resize the array**.  
\* We don’t need to use floor() to get the parent because we’re using integers and when you do division with **integers**, it will be **rounded down automatically to the lowest whole number**.  


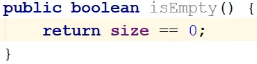
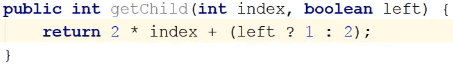
  
\* When we **insert**, we heapify by **looking up the heap** and comparing/swapping parents.  
\* When we **delete**, we heapify by **looking up the heap or** **looking down the heap** and comparing/swapping parents.  
\* **We’re going to assign the new value at the final step, when we have found its corrent position**.  
\* So we’re bubbling it up and we’re shifting parents down.  
  
\* We keep pushing parents down and **when we drop out of the loop, whatever index it’s pointing to will be the spot where we’re supposed to assign the new value**.  


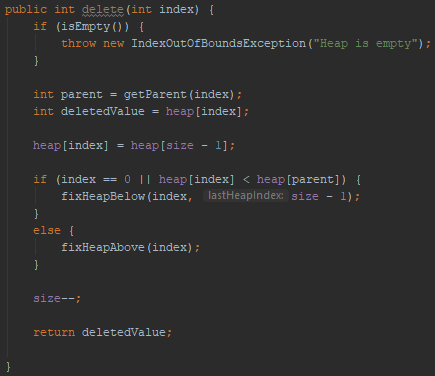
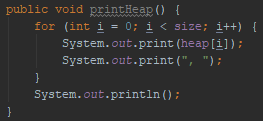
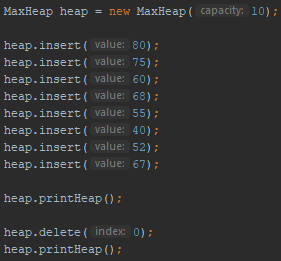
**Heaps (Delete Theory)**  
**Delete**  
=> Must choose a replacement value  
=> Will take the rightmost value, so that the tree remains complete  
=> Then we must heapify the heap  
=> When replacement value is greater than parent, fix heap above. Otherwise, fix heap below.  
\* Just as with BST, when we delete an item, we need to find a replacement value for it, but t his is a lot easier than when we delete a node with 2 children from a BST, because we always use the same replacement node - the rightmost value in the heap, so the rightmost Leaf in the Tree.  
\* The reason we do that is because we want the Tree to remain Complete. And so if we’re going to delete a node from somewhere in the tree and we want to replace that node, if we take the rightmost Leaf from the last level of the tree and we take that node and we delete it and we use it as the replacement node, after we’ve done that, the tree will still be Complete.  
\* Of course after we’ve replaced the deleted value with the new value, we may not have a Heap anymore because the heap property might be violated.  
\* And so after we’ve done that, we need to heapify the Heap.  
\* It’s more common to look down the heap because when you’re taking the very last Leaf and moving it to the left in the array, it’s more likely that you’re going to end up putting it somewhere where it turns out to be < less than its children but it is quite possible that you’re going to put it somewhere and it’s actually > greater than its parent and in that case we need to fix the heap upwards, not downwards.

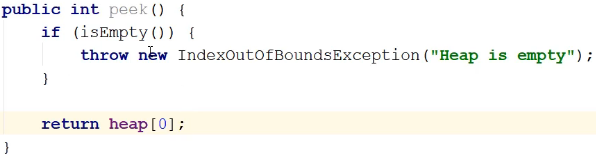
\* The way we decide which direction to look is after we moved the rightmost Leaf to its new position and that position would be the position of the node we deleted, we have to look up the heap if that value is > greater than its parent. If it’s < less than its parent, then we compare against its children and if it’s < less than one of its children, then we have to look down the heap.  
\* If we’re looking up the heap, we’re going to do the same we did for insert().  
\* If we’re looking down the heap, we’re going to swap the replacement value with the larger of its   
2 children and the reason we do that is if we swap it with the larger of its 2 children - so the larger child will become the parent of the replacement value and its old sibling, then we know for sure at that point, that that larger child is larget than both the replacement value and its old sibling and so the heap property is preserved.  
\* In both cases we rinse and repeat until the replacement value is in its correct position.  
  
\* If we took the replacement value from somewhere else, we would no longer have a Complete Binary Tree.  
=> We replace 75 with 67 and delete the original 67 position.  
  
=> Heapify.  
=> 67 < parent 80 so we know that we don’t have to fix the heap above.  
=> **67 < left child => we need to fix the heap downwards**  
=> 67 > right child  
\* **If it were < both of its children, we would want to swap it with the largest child**.  
  
=> We have a Max Heap again.



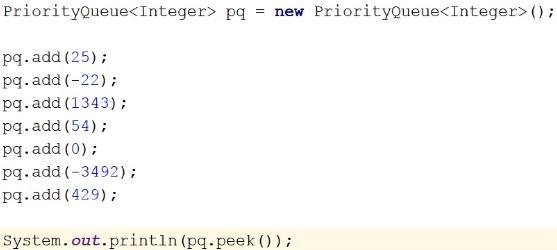
\* Replace 40 with 67.  
  
\* **67 > parent 60 => we need to look up the heap**.  
\* It has no children.  
  
=> 67 > parent 60, swap them.  
=> 67 < parent 80.  
\* That’s it, it’s a little easier than BST because the replacement value is always the same and because we usually are backing a heap with an array, basically the replacement value is always at position   
size - 1 so the last value in the array.

**Heaps (Delete)**  
  
  
\* We’ll take the index for the delete() - if we take the value, then we’re going to have to search the heap for the index of the value, and we can do that of course, but we’d have to use a linear search. We can’t use a binary search because you can only do a binary search on a sorted array and we can’t sort the array becausae that would blow away our heap structure.  
\* So I could accept a value and then the first thing we do in the method is a linear search, but instead, I’m just going to take the index.  
\* One thing to keep in mind is when you’re working with heaps, you pretty much never are going to want to just take some random value off the heap, you always want to take the root off the heap, and so you just have a delete method that doesn’t take anything because it’s understood that you want the root. But some implementations will support deleting or removing any element in the heap, and so that’s what we’re going to do here.  
  
\* We don’t need the lastHeapIndex for the delete but we’re going to need it for Heap Sort later, this basically says this is the last index of the heap. Right now the heap goes from position 0 - (size - 1), this tells us the last position of the heap in the array. For delete, we don’t really need it.  
\* In the WHILE loop, what we’re doing is we’re getting the 2 children of the element at position index, and when we first enter the loop, that will be the index of where we’ve put the replacement value, so the index of where the value we deleted was, and then if the element in that position doesn’t have any children, we don’t do anything.   
\* And so as long as the index we’re looking at is part of the heap, we get the 2 children indices and then we chec kt osee if those indices are actually part of the heap, if they’re not, then we’re done because we’re actually dealing with a leaf, if they are or if at least the left child is, we check for a right child.  
\* If we have a left child but no right child, then by default we’re going to swap with the left child.  
\* If we have 2 children, then we’re going to want to swap with the larger of the 2.  
\* After we’ve decided which child we’d want to swap with, we then compare the value at index with the child we’d swap with, and if the value is < child, we do have to swap them.  
\* If the value at heap[index] >= child, we’re done, the heap is once again satisfying the heap propery.  
\* Otherwise once we’ve done the swapping, we have to compare the replacement value with its new children. We’ve put the replacement value into childToSwap at this point - that’s the new index and so we’re going to update index to that and loop back around.  
\* What if the node has a right child but not a left child?  
=> That’s not possible because a heap has to be a Complete Tree, and so the only possibilities are the node has 2 children or the node has a left child. In a heap we can’t have a node that has no left child but a right child because that would mean that we don’t have a Complete Tree and that would mean we don’t have a heap.

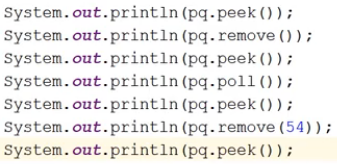
  
\* If we’re dealing with the root, so if the node we deleted was the root and so the replacement value has gone into the root, then obviously we can only look down the tree.  
\* If the replacement value is < parent, that means we don’t have to look up the tree because we know that the replacement value is going to be less than everything above the tree.  
  
\* Now let’s test it.  
  
  
\* If we delete index 1 (which is 75) instead:  
\* That’s it for delete, it’s a matter of replacing the deleted value with the rightmost value and then fixing the heap, either by looking up the heap or down the heap.

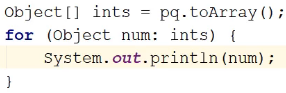
**Heaps (Peek)**  
\* When we call the peek() method, we want to **look at what’s at the root**.  
\* **If we wanted to remove what’s at the root, we’d just call our delete() method with an index of 0**.  
  
\* And that’s pretty much our simple implementation of a heap.  
\* Let’s talk about the Time Complexity.  
**O(logn)** => **insert()** - it’s a constant time operation but then we have to potentially fix the heap and to do that, in the worst case we may have so swap the new item all the way up to the root and that would be logn swaps.  
**O(nlogn)** => **delete()** => **If you want to delete some random value and you don’t have the index.**  
\* For deleting, the first thing we have to do is find the item we want to remove. I didn’t show that, but I explained that - if we’re going to allow deletion of some random object from the heap, we’d have to find it first. And that’s going to be in the worst case Linear **O(n)** - because we’re going to use a linear search to do it.   
\* And then once we’ve found the item we want to delete, we’re going to have to fix the heap potentially and once again, the worst case for that is that we either have to take the replacement value all the way from the root down to the bottom of the tree, or the opposite - take the replacement value from the bottom of the tree and take it all the way up to the root. And so once again, the worst case for that will be **O(logn)**.  
\* However, when you’re working with a heap, you’re really usually only interested in the root. You’re only ever going to remove the root, you’re not going to remove anything else.  
**O(logn)** => **remove the root**  
\* You don’t have to do the linear search.  
\* Finding the root is **O(1)** because it’s always at index 0. And then to fix the heap will be **O(logn).**  
\* So essentially, it will take you a lot longer - on average - to remove a random item than it will to remove the root. Because you know exactly where the root is. You don’t have to do the Linear search which would cost you N steps in the worst case.  
\* When we use heaps, we usually only want to work with the root because   
**we’re generally using a heap because we’re always interested in the minimum or maximum value in the data set**.  
\* If you’re interested in doing random access operations and you’re going to be doing a lot of them, then Heap is not going to be your data structure of choice.  
**O(1)** => **find the minimum or maximum value (Min Heap or Max Heap)**  
\* That’s because it’s always at the root.  
\* That’s different then when you’re working with a BST, because in a BST you may have to travel all the way down the left or right edges to find the minimum or maximum value.

**Priority Queues**  
**Priority Queue** => **A really common use of Heaps**.  
It’s an **Abstract Data Type** that commonly implemented as a **Max Heap**.  
\* Queues are usually FIFO, items are removed from the Queue in the order they were added.  
\* But what if we wanted to change that slightly and say that we always want to access the highest priority item.  
\* And so rather than always removing the items in the order they were added, when we add an item, we assign it a priority and when we go to remove an item, the highest priority item is the one that’s removed.  
\* So it’s not going to be FIFO anymore.  
\* It’s whichever item has the highest priority.  
\* **If the values in the nodes indicate the priority, then a Max Heap is an ideal data structure for it**.  
\* Because the value with the **highest priority is always at the root of the Heap**.  
**O(1)** => **get the highest priority item**.

\* Common operations:  
**insert()** => with priority  
**poll()** => get and remove the highest priority item  
**peek()** => return the highest priority item but don’t remove it  
**PriorityQueue** => **unbounded** priority queue based on a **Priority** **Heap**.  
=> That means there’s no limit to the number of items it can hold.  
\* One interesting thing about this PriorityQueue is that **it’s actually a Min Heap**.  
=> **So the lower the number, the higher the priority**.  
\* It’s an **array implementation**.  
\* They’re resizing the array if you try to add something to a PriorityQueue that’s already full.  
\* It’s **not synchronized**.  
\* If you want to use it **for multiple Threads**, you should use **PriorityBlockingQueue** instead.  
**add()  
remove() => returns the item and removes it  
peek()  
poll() => returns the item and removes it**  
**\* poll() and remove() both remove the highest priority item and return that.**

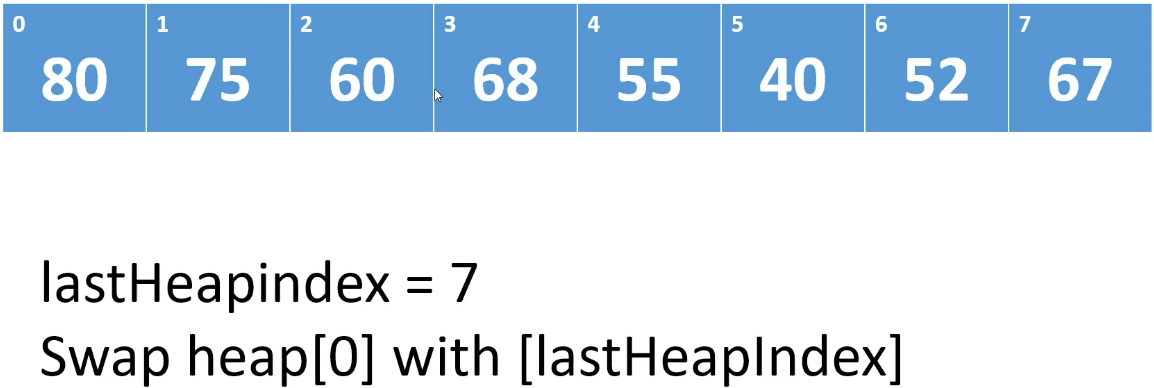


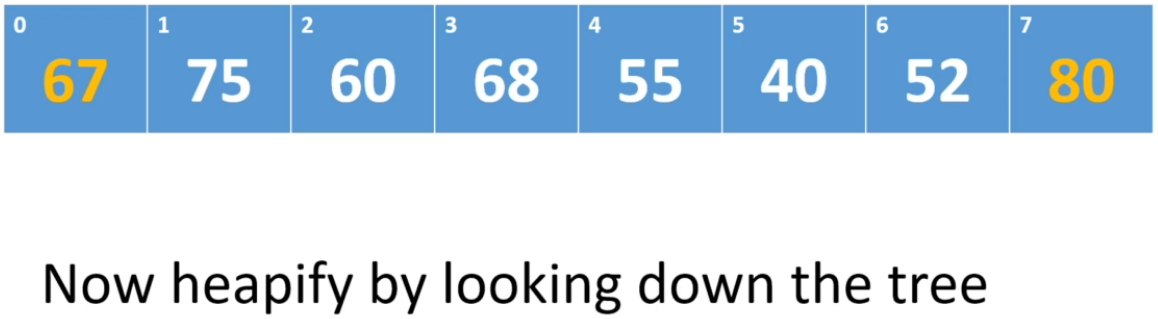
\* You can remove a random item too.  
  
\* If you give the remove() method the priority to delete, it doesn’t return the object you removed because you’re actually passing what you want to remove so it would kind of be redundant.  
\* The difference between poll() and remove() when you pass it a value and in fact remove when you don’t is, if you call:  
remove() => without a value = poll() => removes the root  
remove() => with a specific value => removes the value  
poll() => with a specific value => removes the value  
\* If you want to build a PriorityQueue **using instances that aren’t integers, of course that would be fine, as long as the class implements** the **Comparable interface so that the PriorityQueue can compare instances**.  
\* **You can also provide a Comparator when you construct the PriorityQueue**.  
=> There is a version of the constructor that lets you pass a Comparator.  
**toArray()** => get the array from the queue

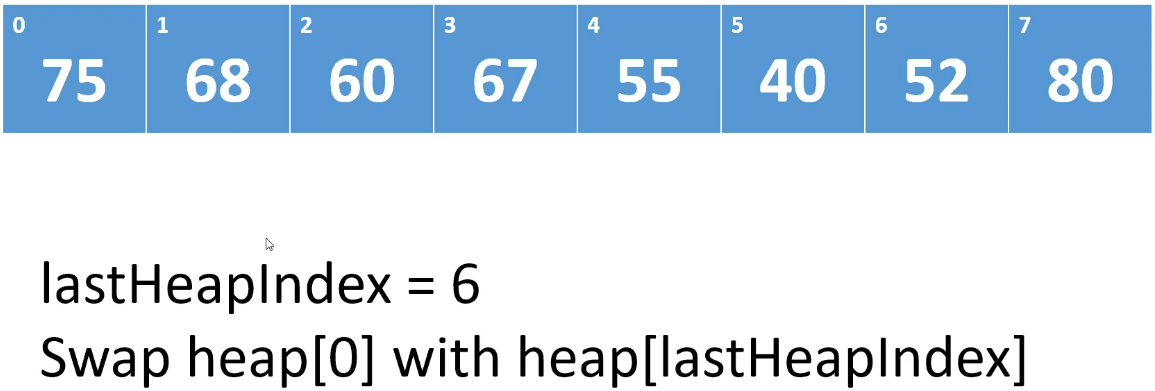
  

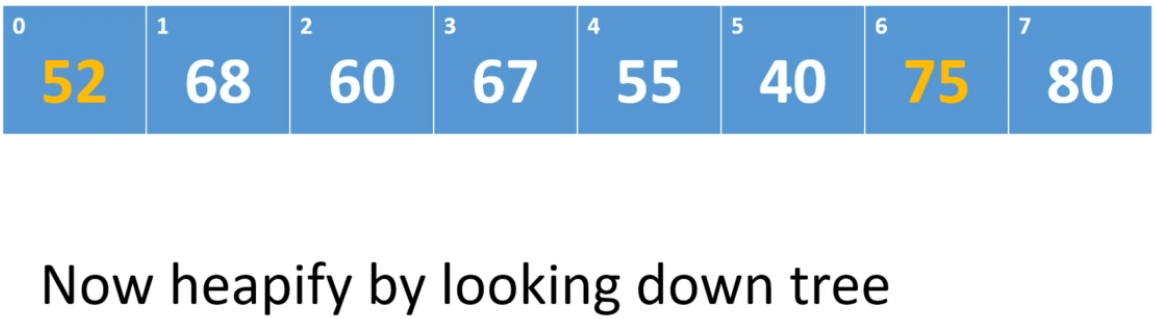

-3492 => children: 0 22  
0 => children: 54 25  
-22 => children: 1343 429  
\* **What if you need a Max Heap?**  
=> Don’t forget that you can provide a Comparator to the class, so you could provide a Comparator that will get you the behavior you want and builds a Max Heal instead of a Min Heap.  
=> And so **you’d need a Comparator that will look at the 2 values and whenever you have one value greater than the other, you in fact want to return that that value’s less**, you’d want to flip things around so that this class would actually be building a Max Heap.

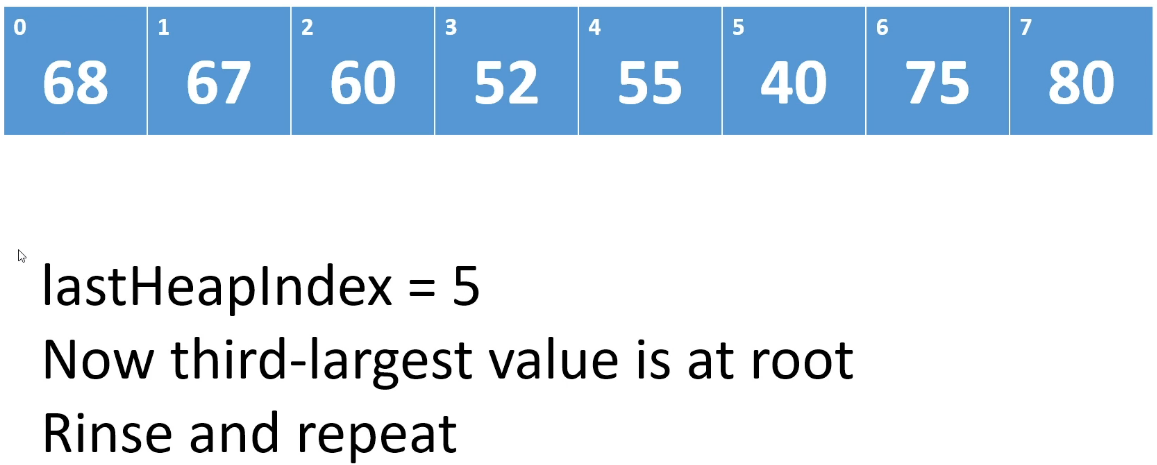
**Heapsort (Theory)**  
**Heap Sort** => **Requires a Heap**  
\* **Max Heap**:  
=> We know the root has the largest value  
=> Swap the root with last element in the array  
=> Heapify the tree, but exclude the last node  
=> After heapify, second largest element is at the root  
=> Rinse and repeat  
\* The implementation would be slightly different for a Min Heap but the concepts would be the same.  
\* If we have an array that is a Max Heap and we want to sort that array, we swap the root with the last element in the array (heap). And then that means that the root is in its correct sorted position, relataive to everything else in the heap. And the we have to heapify the tree but we exclude the last node, so we exclude the root now because the root is now in the last node and we don’t include it when we heapify the heap because if we did, we’d end up moving the root back up to the root and we don’t want to do that.  
\* After we’ve done that, the new root will be the 2nd largest value element in the heap.  
\* And then we rinse and repeat.  
\* And so we take the new root and swap it with the last value in the new heap, which we have reduced by 1 because when we swapped the root to the end, we then reduced the size of the heap by 1 and we heapify that reduced heap and after we’ve done that, the new root will be the 2nd largest element that was in the original heap, and so we rinse and repeat.  
\* When we’re finished, the root will now be the 3rd largest element in the data set.  
\* **This is why we need the lastHeapIndex parameter in the heapify methods**.  
\* **On every iteration, the heap is reduced by 1. So that lastHeapIndex gets decremented by 1 on each iteration**.



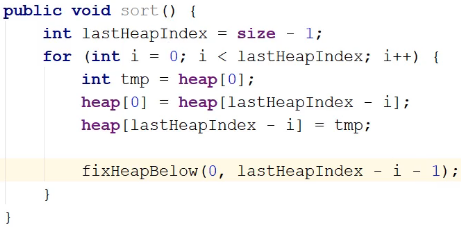


\* Now heapify with lastHeapIndex = 6.  


\* Now heapify with lastHeapIndex = 5.  




\* It’s possible there’s more array beyond index 7 but that part of the array was never part of the heap.

**Heapsort (Implementation)**  
  
\* We keep doing that until we’ve looked at every single item on the heap.  
\* The length of the array is 10 but the size is 8, so there are 2 more positions but they were never part of the heap.  
**O(nlogn)** => in the worst case, the reason for that is we **swap N elements** **O(n)** because we go through the loop N times and then on each iteration of the loop, we also have to fix the heap by calling **heapify** and that is **O(logn)**, so O(n) \* O(logn) = O(nlogn).  
\* And so if you have a Heap and you want to sort it, Heapsort can be a decent sort algorithm.  
**In-place** => **we don’t use any extra memory that relies on the number of items in the Heap**.  
\* Once you’ve sorted the Heap, it’s no longer a Heap. And so you wouldn’t want to sort a Heap that you want to continue using as a Heap.  
\* And so if you want to continue using a Heap as a Heap, you don’t want to sort it because the moment you sort it, just like when we were doing the delete and I said that if you have to find the value, you’re going to have to do a Linear search, you can’t use a binary search because that would require sorting the array, and once you sort the array, you no longer have a Heap, the whole structure has been blown away.  
\* **And so you wouldn’t want to sort a Heap that you want to continue using**.  
\* **If you want to use a Heapsort, your motivation for building a Heap would be because you’re going to use Heapsort on the data, not because you’re going to use the Heap as a Heap**.  
\* If you do it that way though, then the Time Complexity will obviously not be O(nlogn) because you’re going to have to build the Heap first, and that will involved heapifying at each step. Even so, the worst case for doing that can be better than some of the quadratic sort algorithms, so it’s something you might want to consider, but depending on the data of course, there are all the other sort algorithms that we looked at.  
\* **Building a Heap just to sort it might not be the best way to go**.  
\* And that sort of brings us full circle to the beginning of the course where we say **it depends**.  
\* It depends on what you’re going to do.

**Resources**  
PriorityQueue class javadoc  
<https://docs.oracle.com/javase/9/docs/api/java/util/PriorityQueue.html>