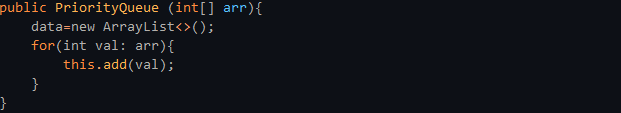
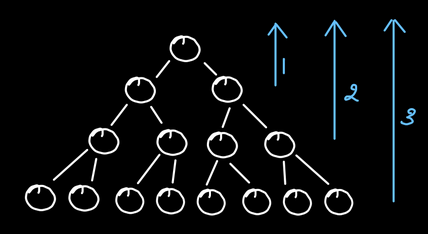
**1. Understanding Problem:**

If the code had been something like shown below, then a priority queue would have been constructed of complexity O(nlogn).



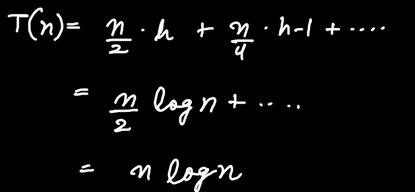
The reason for O(nlogn) complexity is by supposing we have 15 elements and by first adding the root node. Then 2 more elements are added to it which get upheapified. Again each of these elements add 2 more elements which again get upheapified. This goes on and can be depicted by figure 1



The 8 elements in the last level are upheapified by a height of 3. The 4 elements in the level previous to it are heapified by height of 2 and the 2 elements before that are upheapified by a height of 1.

The element in the first level is not upheapified.

Here, we can see that out of the 15 elements, 8 elements (or almost half of 15 elements) are stored in the last level with a time of height, "h". And so on the rest of the elements take subsequently lesser time (see figure 2).



Hence the above figure proves that the complexity for the given code is O(nlogn).

However we are required to write a constructor which achieves this in O(n) time.

**2. Approach:**

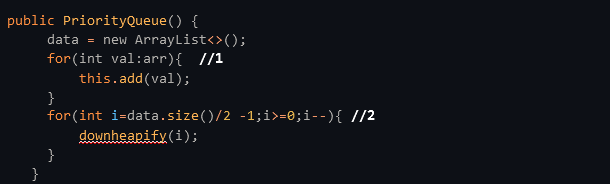
**Optimised Method**

We are going to build a constructor for O(n) complexity.

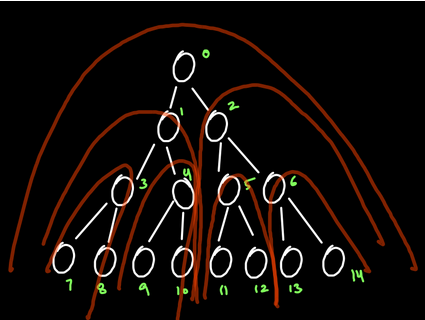
In this approach, we are not going to add the nodes in heap by not calling "upheapify" on any of them. We simply add them in an arraylist.

Now you must be thinking, on adding in arraylist, Complete Binary Tree property has been honored but since "upheapify" has not been called therefore heap won't get formed, and hence, Heap Order property is not honored.

So we apply "downheapify" for this purpose. Let's see how, in the code given below.



Let's now discuss the above code using figure 3.



Here, when //1 is executed, all the nodes of the tree get added one by one.

Moving to line //2, we see that according to the code, initially i= data.size()/2 -1=15/2 -1=6.

Hence, we start calling downheapify from node 6.

As you can see in the figure, downheapify is called on the subtree with root as the 6th node. This subtree then becomes heap.

As we go further in the loop, we call downheapify in the given order of the roots: root node 5, root node 4, root node 3, root node 2, root node 1 and finally root node 0.

At this moment, the entire tree has been formed as a heap.

**ADVANTAGE OF DOWNHEAPIFY:**

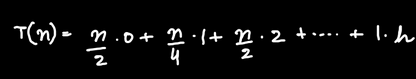
In the last level, where there were the maximum number of nodes ( i.e. n/2 nodes), no work was done on them.

In the level previous to it, where there were a lesser number of nodes i.e. n/4 nodes, downheapify was called for a height 1.

Further, in the previous level, where there were an even lesser number of nodes i.e. n/2 nodes, downheapify was called for a height 2.

Also, for the first level of the tree, of only 1 node, downheapify was called for the entire length of the tree, h.

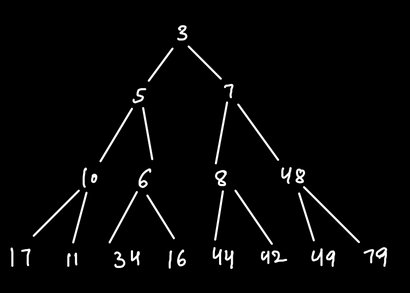
According to this, the time complexity will be as shown in figure 4.



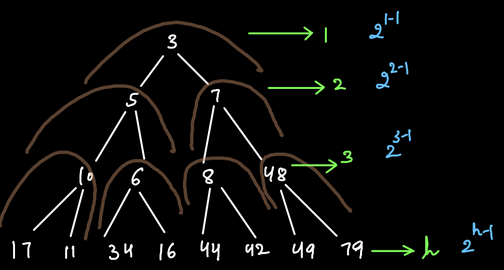
Comparing figure 4 with figure 2 we notice that in figure 2, a maximum number of nodes are required for most work which isn't desirable. However, in Figure 4, the maximum number of nodes have to do the least work (no work) which is quite advantageous.

Let's now understand the concept of downheapify using an example.

Say, the given tree is as shown in Figure 5.



As we have seen in the optimized approach, we call downheapify on its nodes as shown in figure 6.



As seen in //2 of the code, initially i= data.size()/2 -1= 15/2 -1= 6.

Hence, we start calling downheapify from node at 6th position, i.e. node 48.

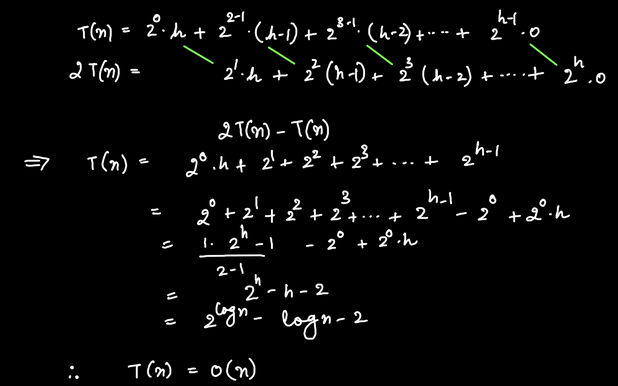
As you can see in the figure, downheapify is called on the subtree with root as the 6th node. This subtree then becomes heap.

As we go further in the loop, we call downheapify in the given order of the roots: root node 8, root node 6, root node 10, root node 7, root node 5 and finally root node 3.

At this position, the entire tree has been formed as a heap.

At each level of the tree, the corresponding number of nodes in that level is written.

Hence, the time complexity of this tree will be as shown below.



From the above calculations, we get the time complexity as O(n).