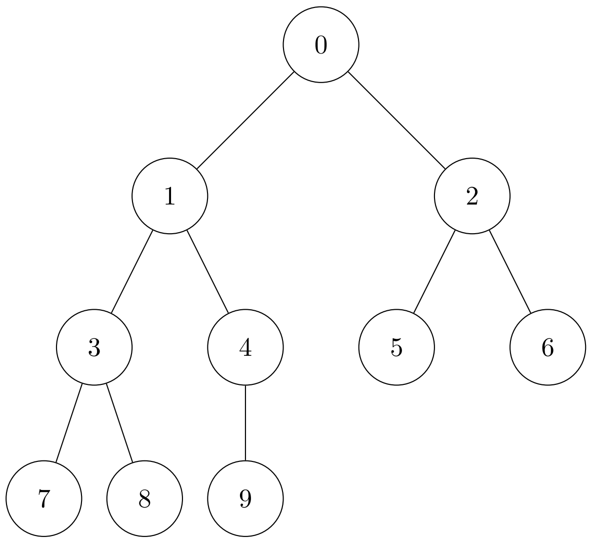
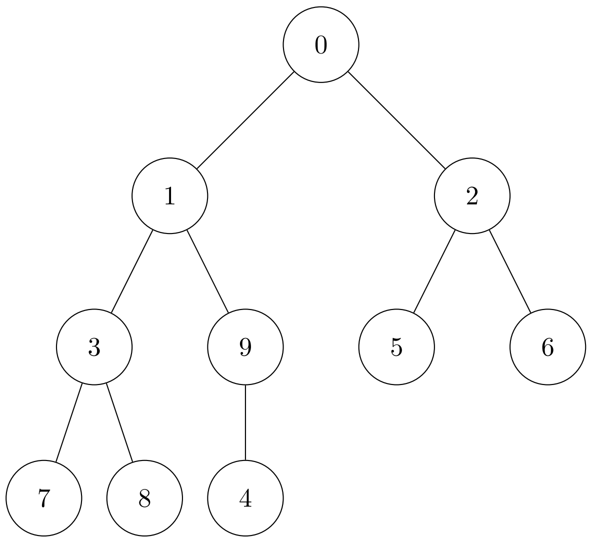
Varsha Baisane, DS Quiz 1

To represent a heapify program using trees, we build a tree with the elements of the array inserted into it.
The heapify program begins the loop from $n/2-1$ leaf nodes of the tree and then compares the current node with its children. If the parent node happens to be smaller than the child node, it swaps the parent node with the corresponding children node. The loop goes on for all the n elements within an array. And once, all n elements have been heapified according to these conditions, we now have our new heaped tree.

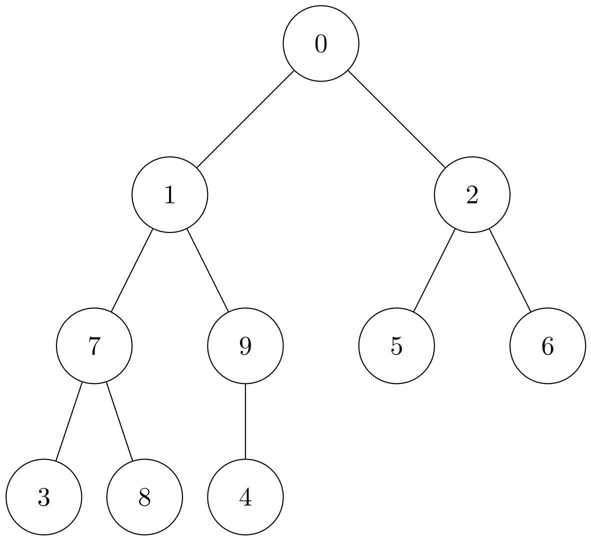
The heapify program can be represented using binary trees as follows:



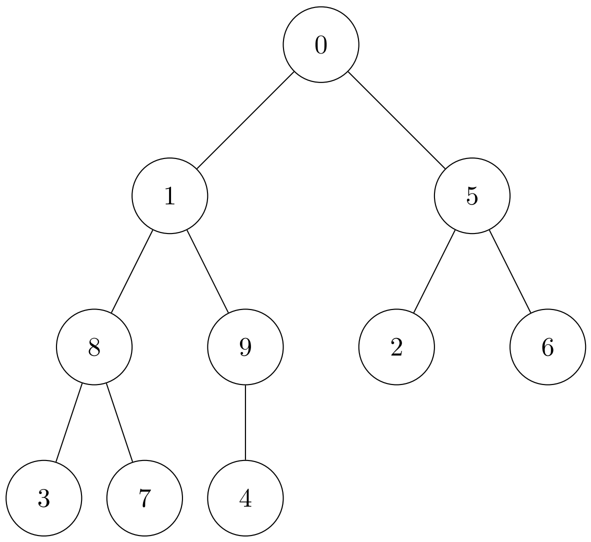
I begin at $n/2-1$ leaf nodes, i.e. 4. Comparing with its children 9, the heap property is violated as $9 > 4$ and thus there's a swap between the parent node and the child.



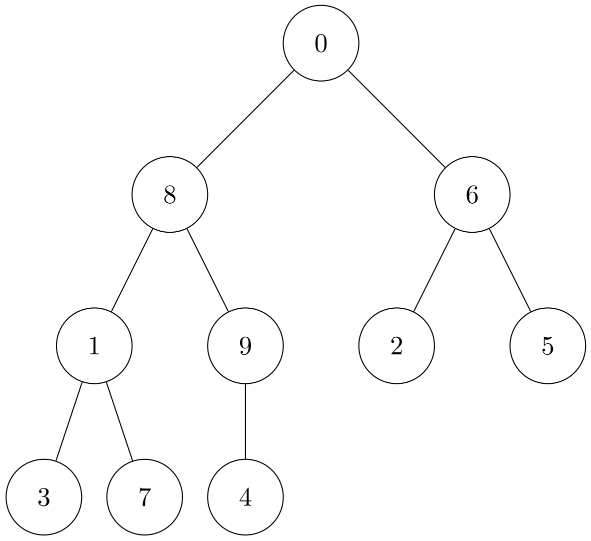
Now, 9 holds the heap property so move on to 3 as this node's children violates the heap property.



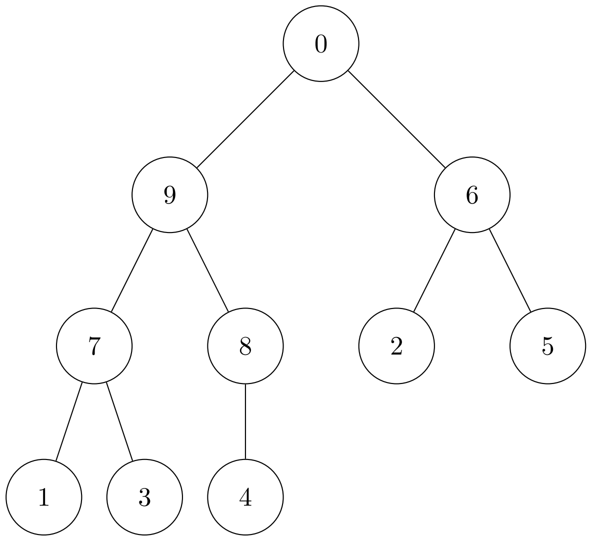
Swapping 7 with its parent 3 as it again violates the heap property.



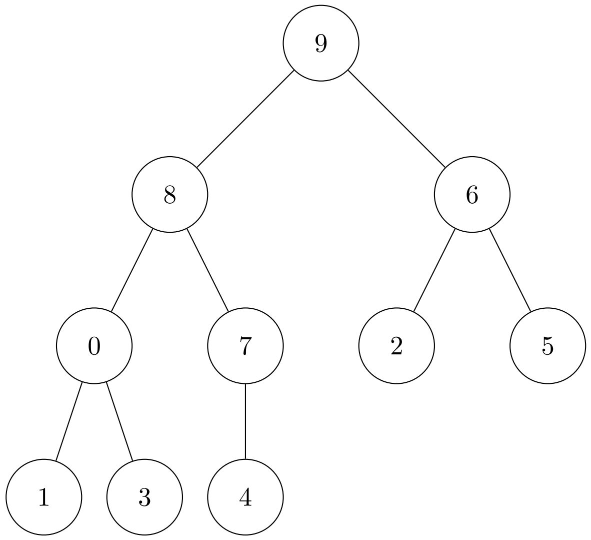
Swapping 8 with its parent 7 and 5 with 2 as it's greater



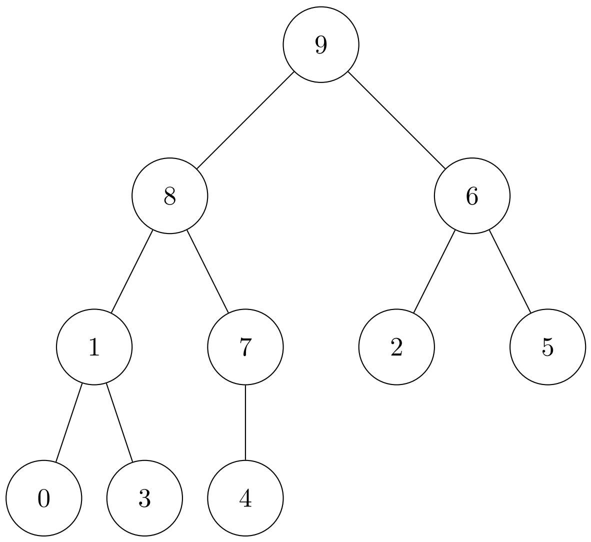
Swapping 6 with its parent 5 and 8 with 1.



swapping 3 with 1 and 7 with 3



Swapping 9 with 0 and 7 with 0 and 8 with 7.



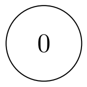
Swapping 1 with 0.

Thus, we get a max-heap tree.

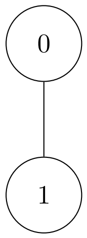
Time Complexity: O(n)
The number of comparisons for heapify is approximately O(n * $\log_2$(n)) in the worst case. As in each iteration of the loop, the program does two comparisons (arr[left] > arr[largest] and arr[right] > arr[largest]). The number of iterations is proportional to the height of the binary tree, which is $\log_2$(n) for 'n' number of elements. Therefore, the total number of comparisons is O(n * $\log_2$(n)) in the worst case.

HeapInsert

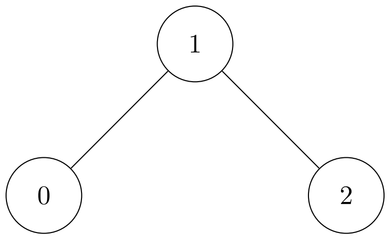
Representing HeapIsert using binary tree representations can be given as follows:



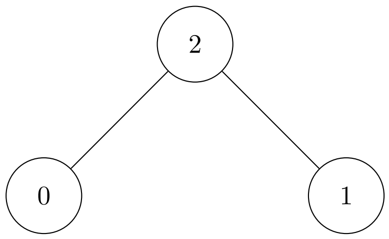
The heap is now inserted with the first element of the array, 0.



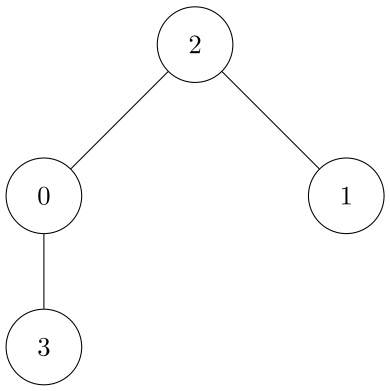
Inserting the next element 1 and now we compare the inserted node with its parent node and if parent node is smaller than the child, swap and go on with it with the newly inserted element.
Therefore, we have



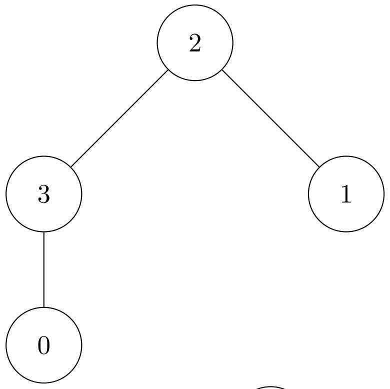
Comparison and swap

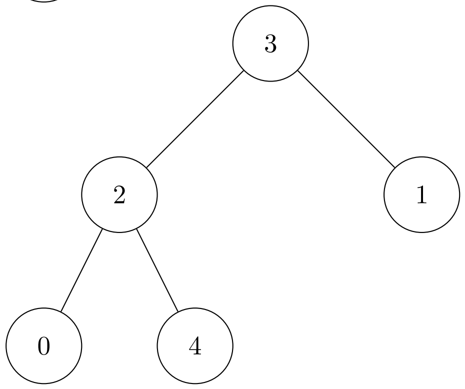


c++

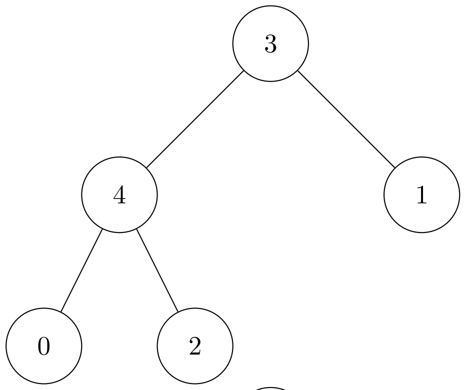


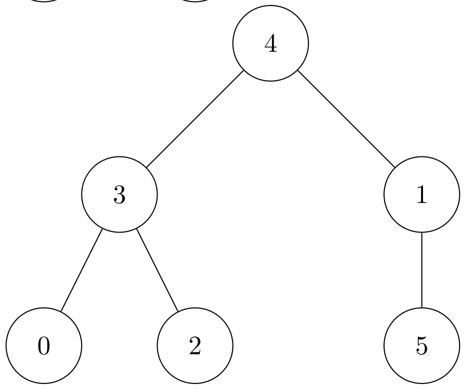
Thus, comparing and swapping along with insertion of the newly added element, the trees goes on as follows:



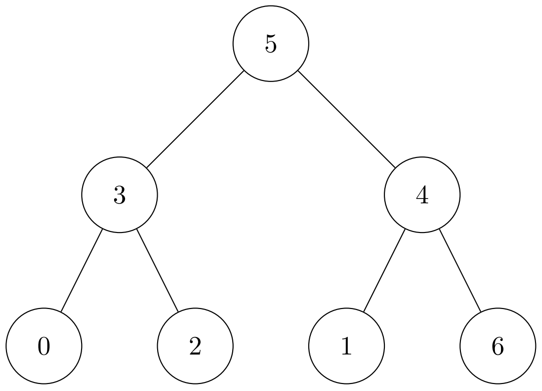


Swapping 3 and inserting 4.

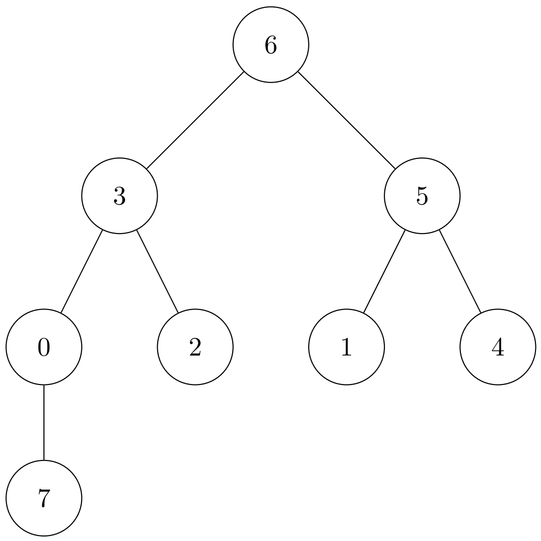




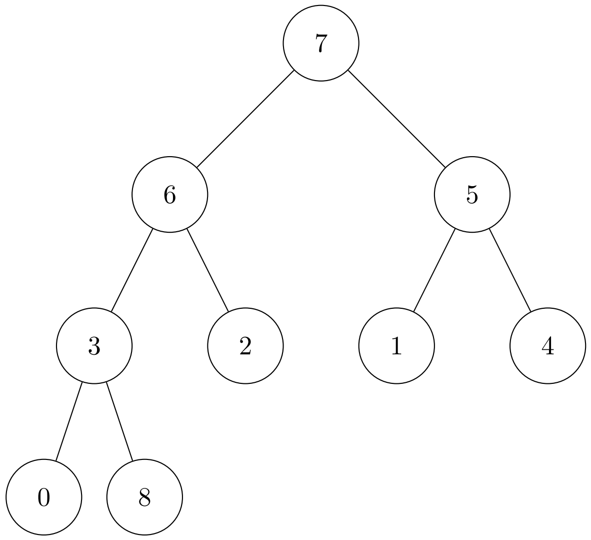
Swap 4 and insert 5,



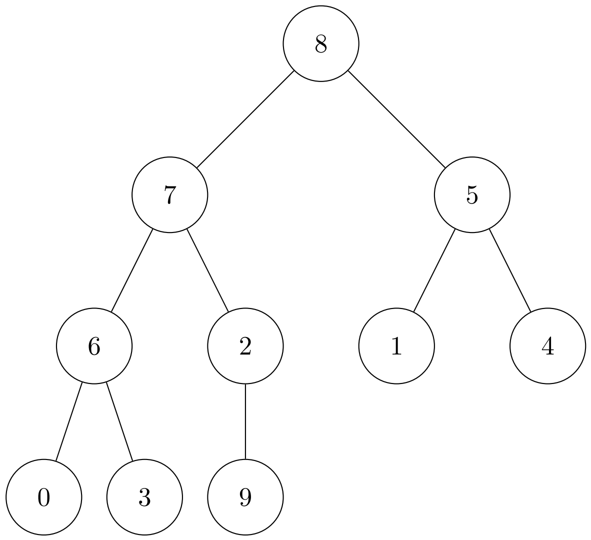
Swap 5 and insert 6.



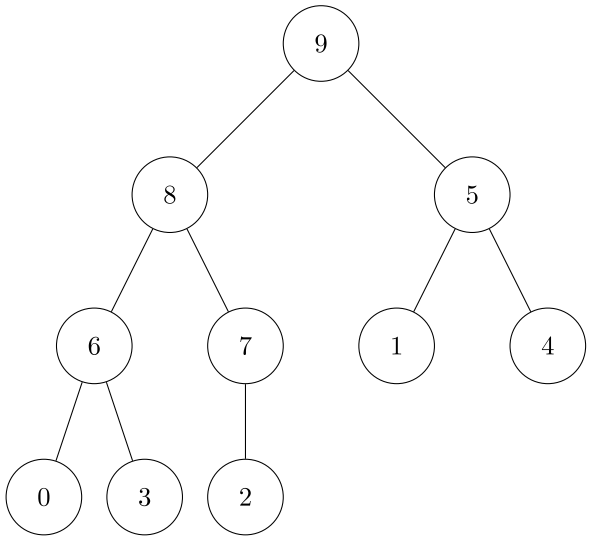
Swap 6 and insert 7



Swapping 7 and inserting 8.



Swap 8 and finally insert 9 the adjustment of this element will finally give us the desired tree.



Thereby, giving us our completely inserted and sorted heap.

Time Complexity: O(log n) which for n elements is O(n log n)
The number of comparisons for heap insertion is approximately O($\log_2$(n)) in the worst case. This is because the element being inserted may need to be compared and swapped with its parent at each level of the binary heap, and the number of levels in a binary heap is $\log_2$(n) for 'n' elements.
Therefore, O($\log_2$(n)) comparisons.