<u>Heap Data Structures:</u>

Binary Tree: A type of tree where each node can have a maximum of two children. A heap is a binary tree where the value of a node is always greater or equal to the values of its children nodes. The first tier of the tree contains just one node, the root node. From the next tiers nodes are added filling up empty positions from left to right. Although a heap is a tree containing vertices and edges, the data structure it uses is just a regular array. Vertices and edges are used for a better understanding but the background data structure is a simple array, not a 2d array or an array list.

X	T	O	G	S	M	N	E	R	Y	Z	C

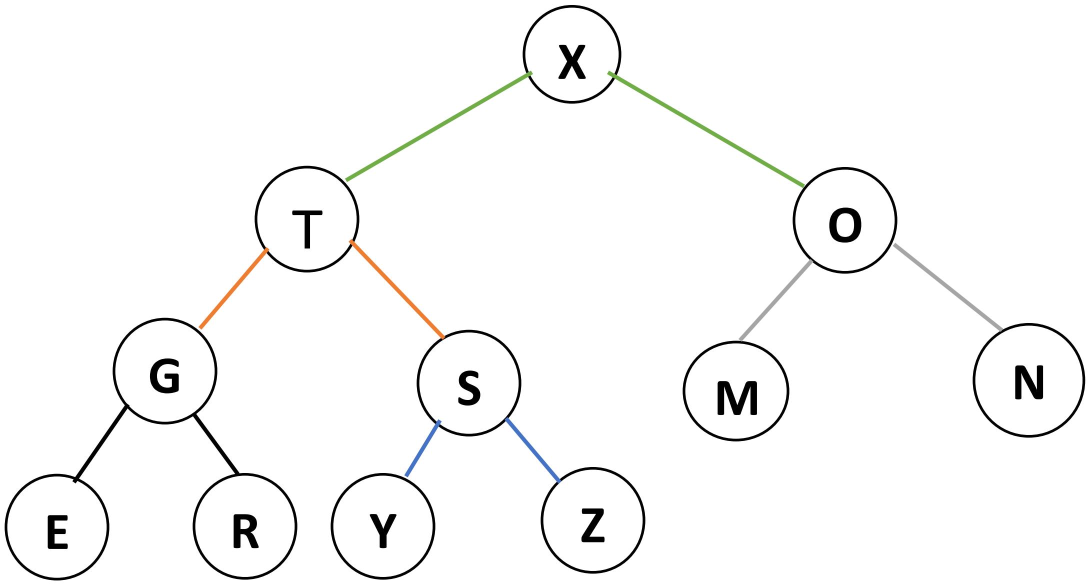
The above array is a heap. To satisfy heap properties:

X is greater T and O

T is greater than G and S

O is greater than M and N

In heap one parent can have two children at max(It can have one child). One item falls under only one parent. The sequence gets simpler if we arrange it in Binary Tree manner. Consider the above set of letters as a Binary Tree.



We call a Tree heap-ordered if the following property is held. KEY of the parent NODE is equal or greater than its children NODES. Note that the array starts from position 1.

1	2	3	4	5	6	7	8	9	10	11	12
X	T	O	G	S	M	N	E	R	Y	Z	C

For ith NODE, its parent NODE is i/2.

For ith NODE, its children are 2i and 2i+1.

Note: 2i is the LEFT child and 2i+1 is the RIGHT child.

Try to find parents and its children of the above elements.

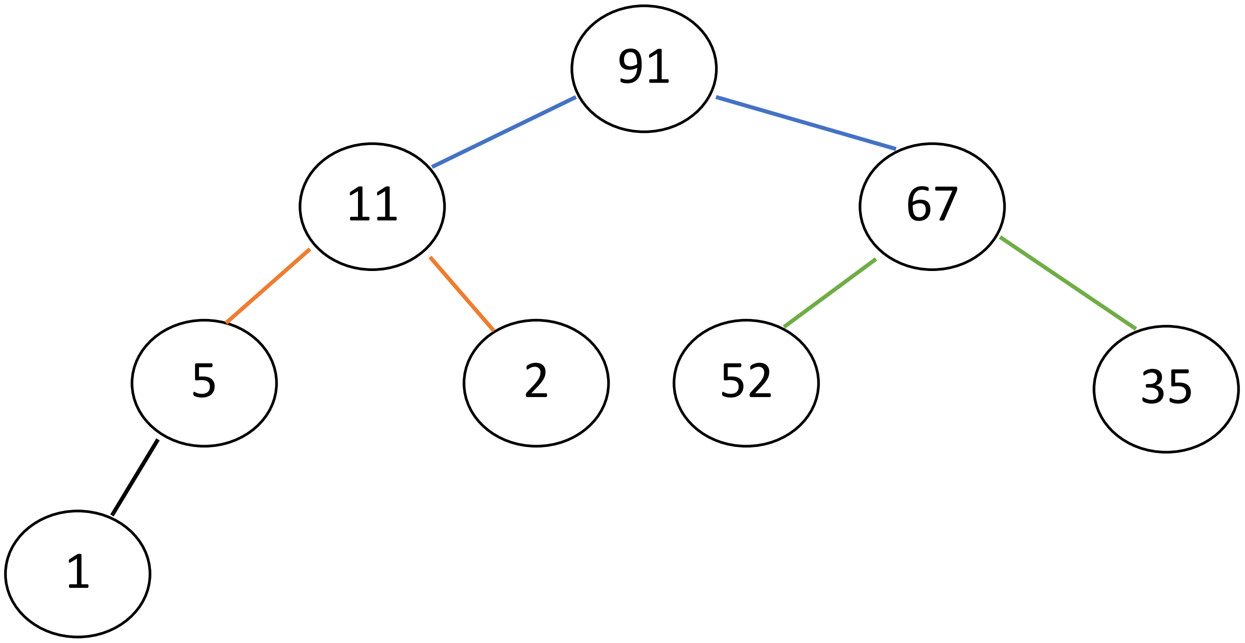
<u>Benefit of using ARRAY for Heap rather than Linked List</u>

ARRAYS give you random access to its elements by indices. You can just pick any element from the ARRAY by just calling corresponding index. Finding parent and its children is trivial.

Linked List is sequential. This means you need to keep visiting elements in the linked list unless you find the element you are looking for. Linked List does not allow random access as ARRAY does. On the other hand each linked list must have three (3) references to traverse the whole Tree (Parent, left, Right).

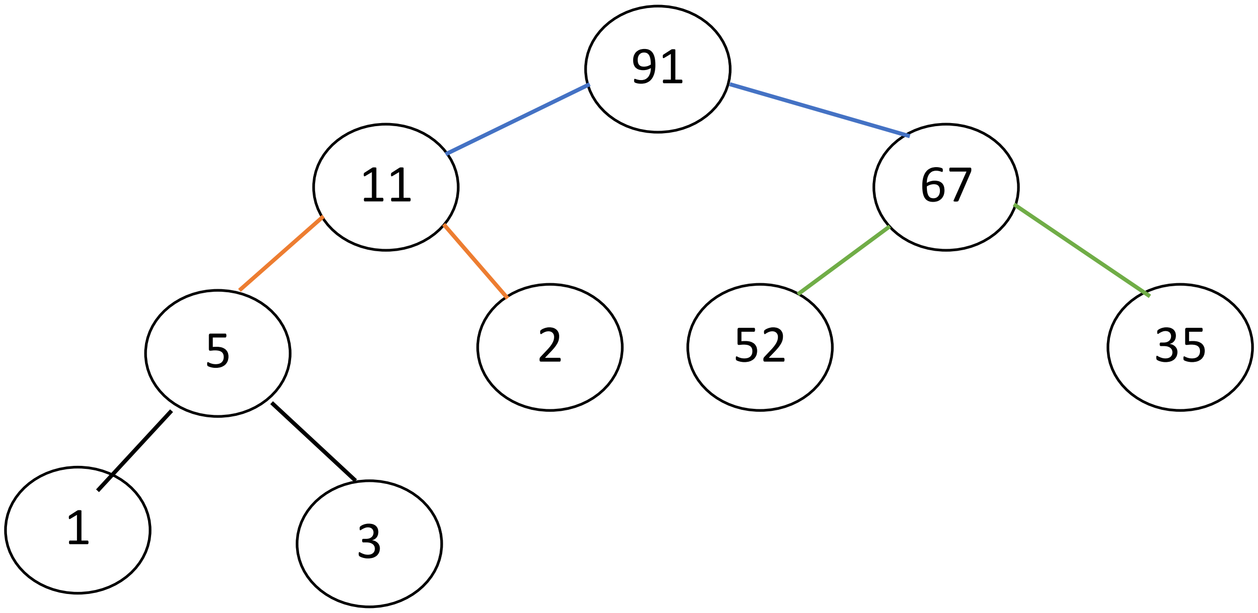
<u>Operations on Heap</u>

(a) **Insert ():**Inserts an element at the bottom of the Heap. Then we must make sure that the Heap property remains unchanged. When inserting an element in the Heap, we start from the left available position to the right.



Consider the above Heap. If we want to insert an element 3, we start left to right at the bottom level. Therefore 3 will be added as a child of 5.

Then the new Heap will look like this:

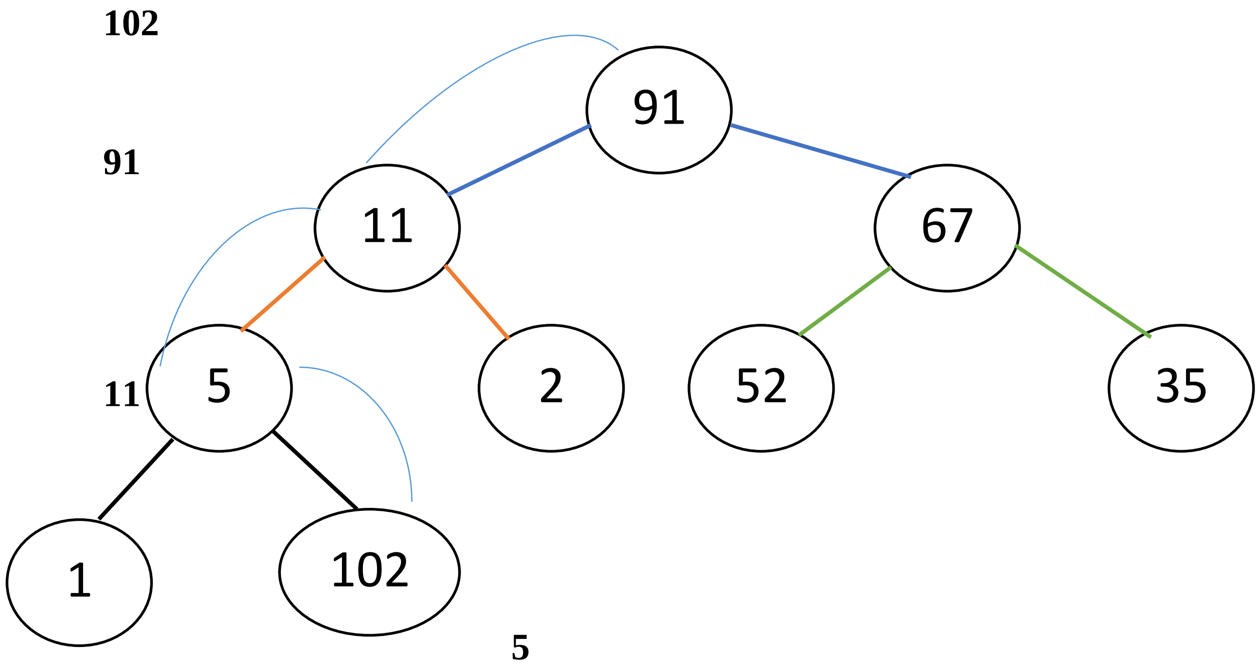


Look carefully five (5) is 3's parent and it is larger. Hence Heap property is kept intact.

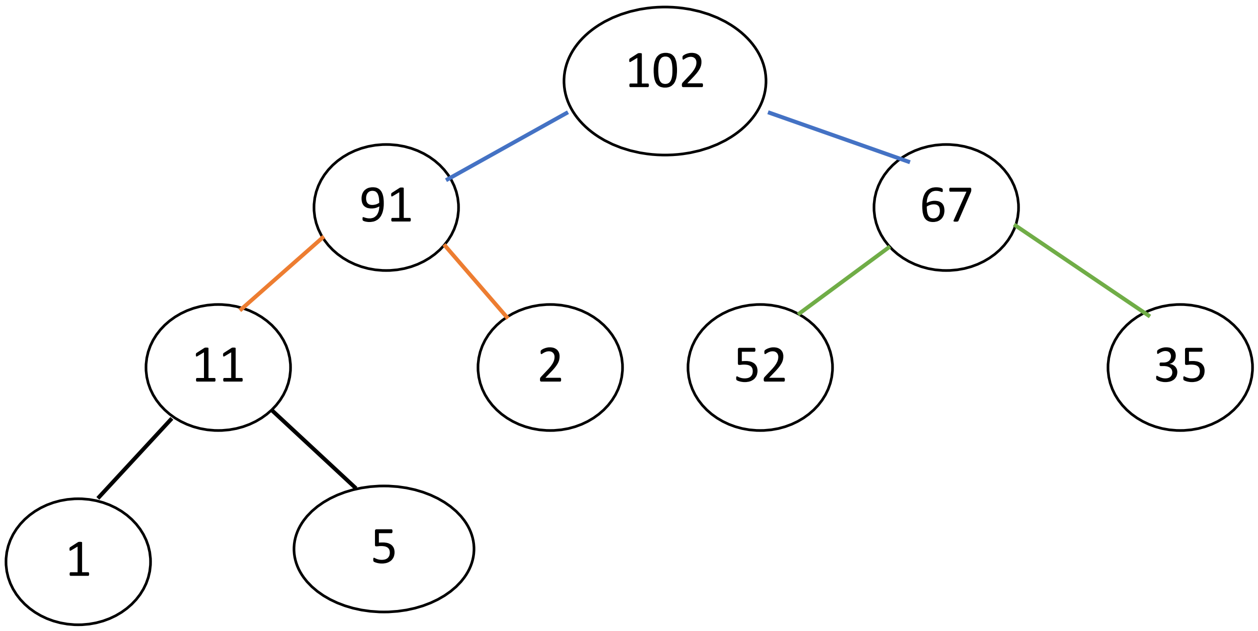
What if we want to insert 102 instead of 3?

Let's say we want to insert 102 at the existing Heap. So where to add? Obviously 102 will be added as a child of 5. Now is the Heap property hold intact? Therefore we need to put 102 in its correct position. How we gonna do it? The methodology is called HeapifyUp or MaxHeapify ().

(b) HeapifyUp(): Let the new NODE be 'n' (in this case it is the node that contains 102). Check 'n' with its parent. If the parent is smaller (**n > parent)** than the node 'n', **replace** 'n' with the parent. Continue this process until n is its correct position.

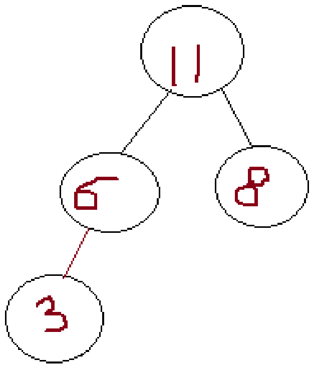


After the HeapifyUp operation the Heap will look like this:

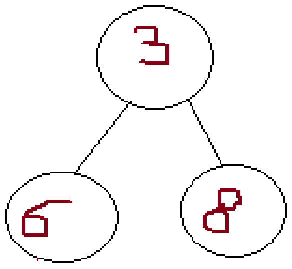


Time Complexity: Best case O(1) when a key is inserted in the correct position. Else O(lgn) in the worst case.

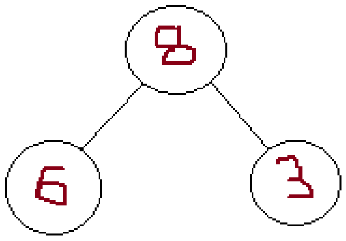
(c) Delete (): While deletion, delete the root of the Heap Tree. How? Deletion is done by replacing the root with the last element. The Heap property will be broken 100%. Small value will be at the top (root) of the Heap. Therefore we must put it in a right place which is definitely somewhere down the Tree.



Replace the root (11) with the last node (3)

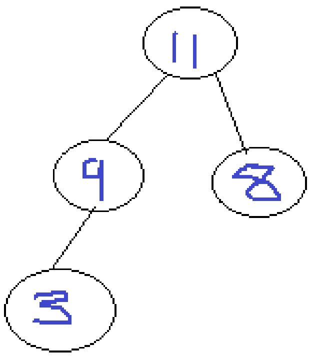


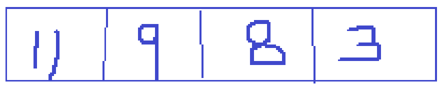
The root is now smaller than its children. So replace 3 with the children with the greater key



This process of putting a node in its correct place by traveling downward is called heapify down.

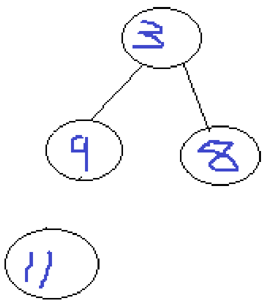
Heap Sort: Combination of delete and heapify down. One thing we know for sure which is the maximum valued/key node is the root node.

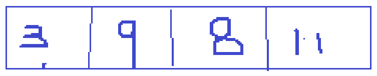


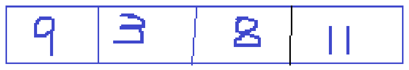


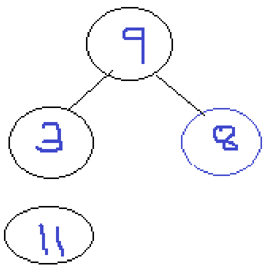
Steps:

Replace the root with the last node, which is basically delete and then heapify down.



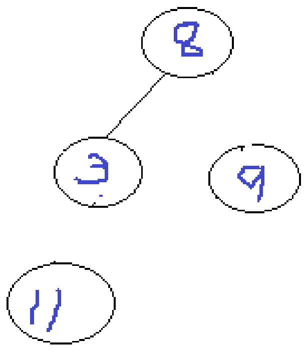


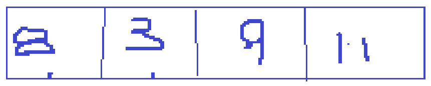




After heapify down

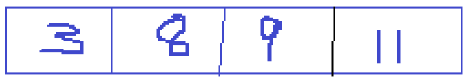
Now replace 9 with 8, which is basically delete again.

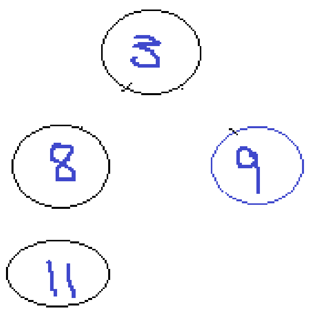




Heapy down not required because the root is bigger than child

Now replace 8 with 3.





After heapify down

This is heap sort. Time complexity O(nlgn). Heap sort is in place, that is, no extra array is required and not stable.