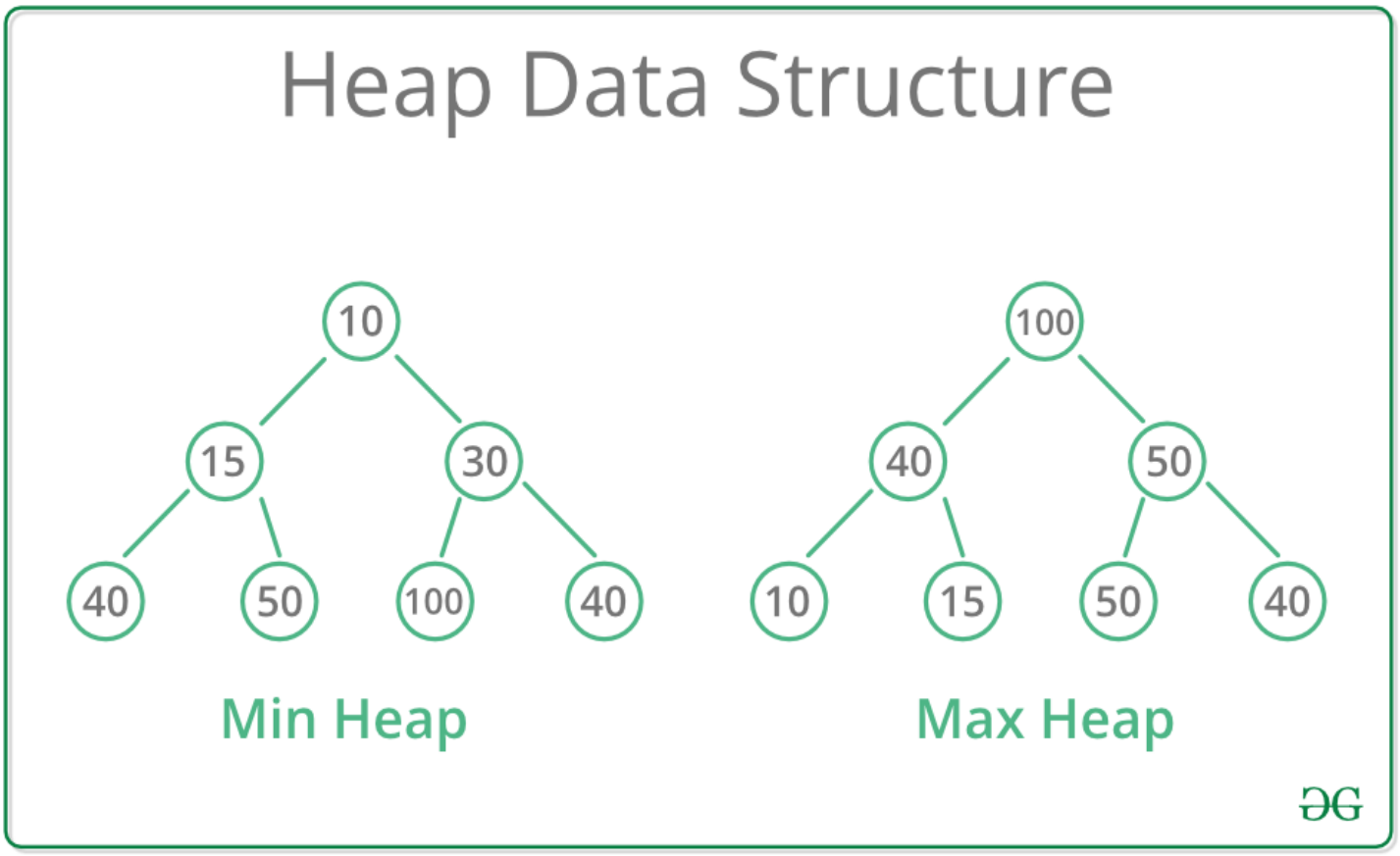
HEAP

- Theory of Construction of Heap
- Problems on Heap – using heapq

We have so far covered, **Hierarchical Data Structure**, like trees & BST **and Linear Data Structures** like array, queue, stack & linked list.

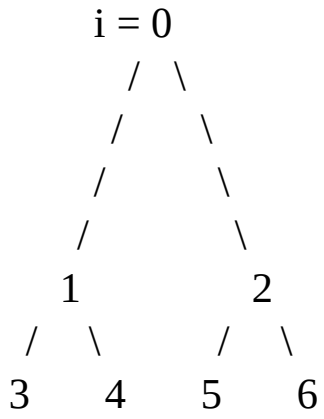
A HEAP is a special tree-based data structure in which the tree is a complete binary tree. Generally, heaps can be to two types:

1. **Max-Heap**: In a max heap the key present at the root node must be greatest among the keys present at all it's children. The same property must be recursively true for all sub-trees in that Binary Tree.
2. **Min-Heap**: In a min heap the key present at the root node must be minimum among the keys present at all of its children. The same property must be recursively true for all sub-trees in the Binary Tree.

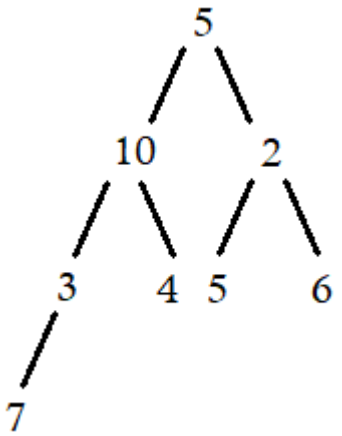


Properties:

1) It is a complete tree; all the levels are filled and the last level is filled in left to right order.
2) For a node ‘i’, 2*i + 1 (left-child) and 2*i + 2 (right-child)



Q) How will you create a max heap or min heap.



Heap is represented using an array. The above heap has 8 nodes, so we need to create 8 boxes.

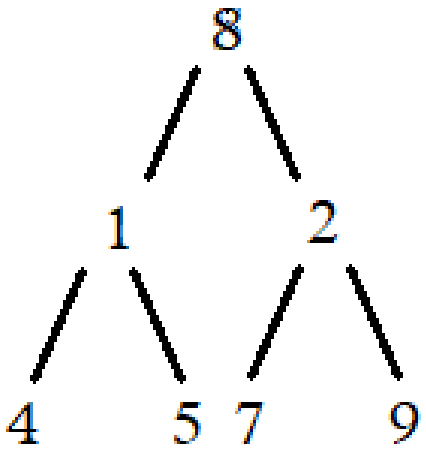
0	1	2	3	4	5	6	7
5	**10**	**2**	**3**	**4**	**5**	**6**	**7**

We have to write the heap in a level order traverse. We create a heap using array.

The index of 5 in the array is 0. The index of the left child will be 2*i + 1 = 1 and 2*I + 2 = 2. So, 10 is at 1st index and 2 is at the 2nd index. Using the same property, we can find out the left and right child for a particular index.

Q) Given a Heap convert into Max Heap

8	1	2	4	5	7	9



Percolate UP

Is 9 a max heap in itself? Yes

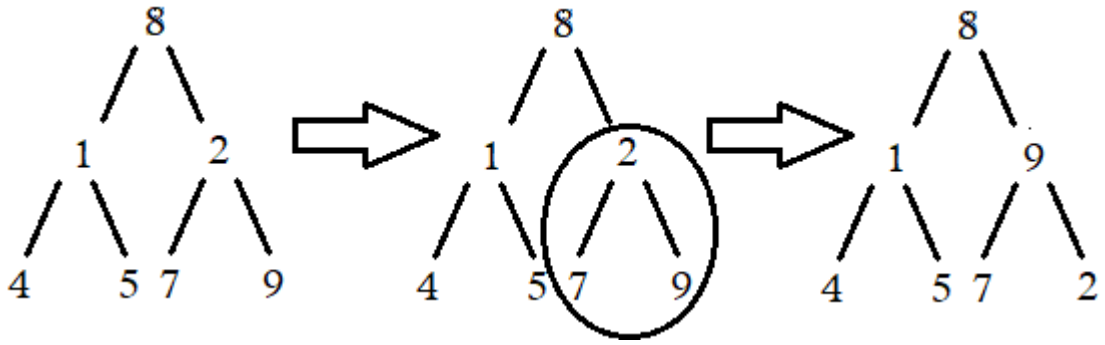
Is 5 a max heap in itself? Yes

Is 7 a max heap in itself? Yes

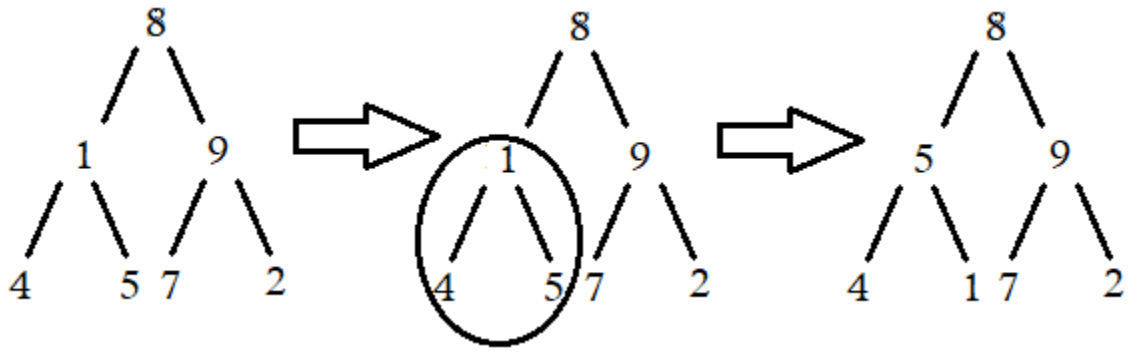
Is 4 a max heap in itself? Yes

The leaf node on itself is a max heap.

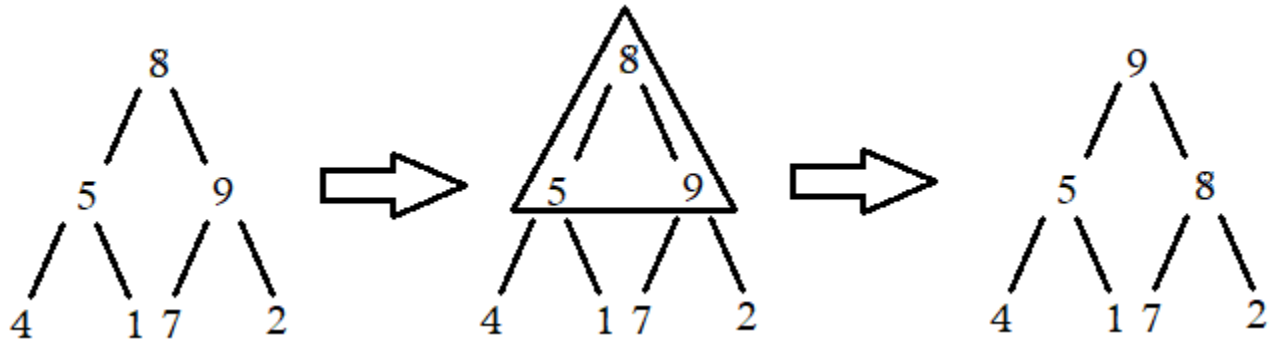
For 2 to be max heap, 2 should be greater than the elements below it. So, for I = 2, 2*i+1 = 5 and 2*i+2 = 6. The element at 6th index is greatest, so if we replace the 2nd index value with the 6th index value, then it would become a max heap.



For 1 to be max heap, 1 should be greater than the elements below it. So, for I = 1, 2*i+1 = 3 and 2*i+2 = 4. The element at the 4th index is greatest, so if we swap the 1st index value with the 4th index value, then it would become a max heap.



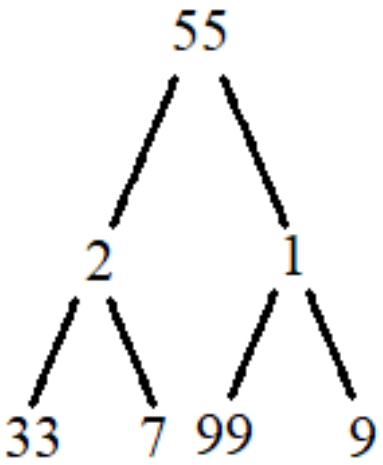
For 8, to be max heap, 8 should be the greatest element that all the elements below it. Since we have already swapped the elements at index 1 & 2, if we only need to check for index 0, 1 & 2.



For i = 0, 2*i+1 = 1 and 2*i+2 = 2. Since the 2nd index has the max value, we need to swap the 0th index with the 2nd index value.

Percolate DOWN

Heap → [55, 2, 1, 33, 7, 99, 57]
0 1 2 3 4 5 6



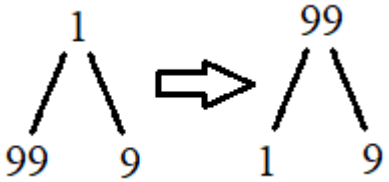
indexing,

At 0 → 55
At 1 → 2
At 2 → 1
At 3 → 33
At 4 → 7
At 5 → 99
At 6 → 9

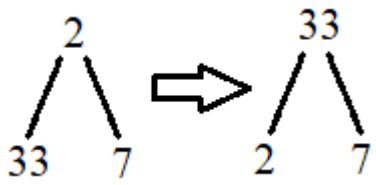
33 in itself is a max heap
7 in itself is a max heap
99 in itself is a max heap
9 in itself is a max heap

We will check from bottom. 99, 9, 7 and 33 are max heap in itself.

At 2nd node, 1 should be the greatest among all its child nodes. So, the left value (5th node) = 99 and the right value (6th index) = 9. Since 99 is the greatest value (99 > 1), we will swap 2nd index with 5th index.

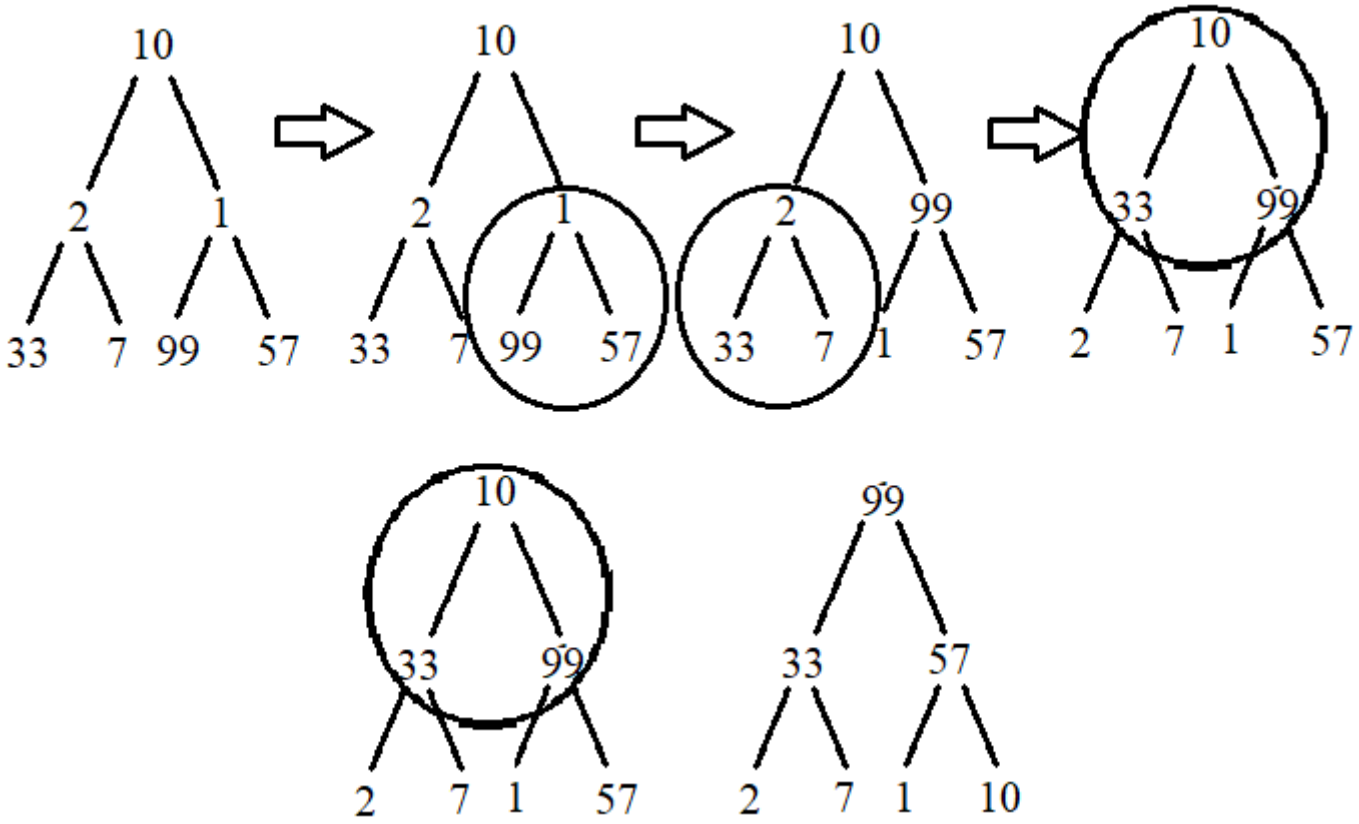


At 1st node, 2 should be the greatest among all its child nodes. But, the left value (3rd index) = 33 and the right value (4th index) = 7. Since 33 is the greatest value (33 > 2), we will swap 1st index with 3rd index.



Among the 55, 33. 99 the max value is 99. So, the 2nd index value will be swapped with 0th index value and we will get a max heap.

For a heap [10, 2, 1, 33, 7, 99, 57]



Time Complexity for Percolate Up is **O(log n)**.

https://www.geeksforgeeks.org/time-complexity-of-building-a-heap/

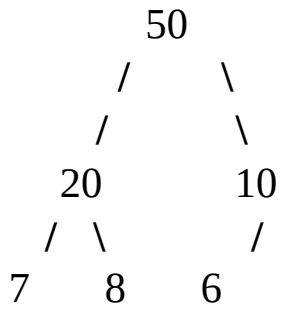
Heap Implementation:

```
heap = [1,2,3,4,5,6,7,8,9]


def heapify(i):
    global heap

    left_idx = 2*i + 1
    right_idx = 2*i + 2

    # leaf node since they are already max heap
    if left_idx > len(heap) - 1 or right_idx > len(heap) - 1:
        return

    max_idx = 1
    if heap[i] < heap[left_idx]:
        max_idx = left_idx

    if heap[max_idx] < heap[right_idx]:
        max_idx = right_idx

    if max_idx != i:
        heap[max_idx], heap[i] = heap[i], heap[max_idx]
        heapify(max_idx)


if __name__ == "__main__":
    n = len(heap)

    for i in range (n - 1, -1, -1):
        heapify(i)

    print(heap)
```

OUTPUT:

[9, 8, 7, 4, 5, 6, 3, 2, 1]

Sort – Heap



We will swap the last value with the root value. So, [50, 20, 10, 7, 8, 6] will become [6, 20, 10, 7, 8, 50]. The delete the last node (50). Since 20 is biggr will swap it with 6 and then swap 6 with 8.

So, [6, 20, 10, 7, 8] → [**20**, **6**, 10, 7, 8] →[20, **8**, 10, 7, **6**] →[**6**, 8, 10, 7, **20**]

CODE:

```
heap = [1,2,3,4,5,6,7,8,9]

def heapsort():
    global heap

    while len(heap) != 0:
        print(heap[0])
        heap[0], heap[-1] = heap[-1], heap[0]
        heap.pop()
        heapify(0)

def heapify(i):
    global heap

    left_idx = 2*i + 1
    right_idx = 2*i + 2

    # leaf node since they are already max heap
    if left_idx > len(heap) - 1 or right_idx > len(heap) - 1:
        return

    max_idx = 1
    if heap[i] < heap[left_idx]:
        max_idx = left_idx

    if heap[max_idx] < heap[right_idx]:
        max_idx = right_idx

    if max_idx != i:
        heap[max_idx], heap[i] = heap[i], heap[max_idx]
        heapify(max_idx)
```

```
if __name__ == "__main__":
    n = len(heap)

    # Creating a max heap
    for i in range (n - 1, -1, -1):
        heapify(i)

    heapsort()
```

OUTPUT:

9
8
7
6
5
4
3
2
1