CS601-01: HEAP DATA STRUCTURE

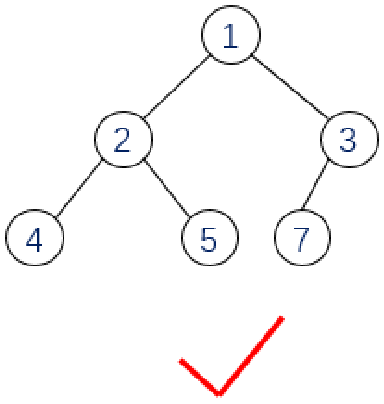
NEHA HATTIHOLI & SWETHA BANAGIRI

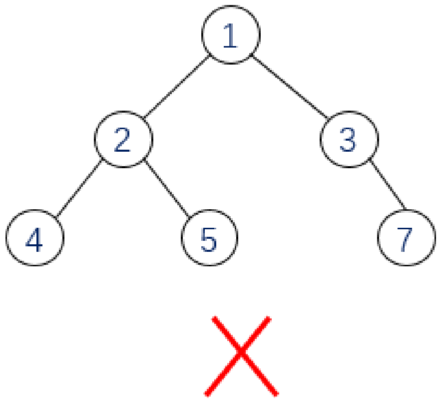
AGENDA

- Overview
- Properties
- Heap Implementation using Array
- Types of Heap – Min Heap/ Max Heap
- Heapify Algorithms
- Insert/Delete Operations
- Time/Space Complexity
- Applications of Heap

OVERVIEW

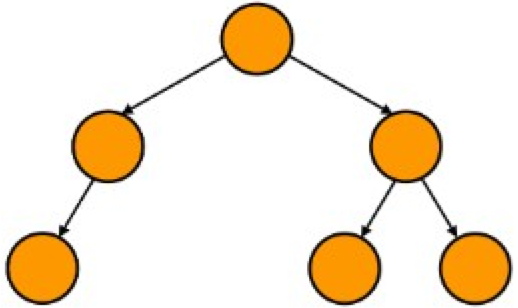
- Binary Tree based Data Structure
- A complete tree that satisfies heap property
- Efficient Implementation of Priority Queue
- Implemented using arrays
- If node A is the parent of node B, then node A is ordered with respect to node B with the same ordering applied to the whole heap



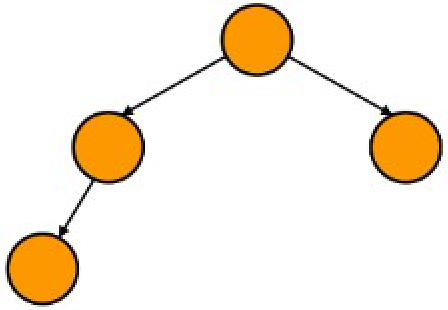


THINK TONK 

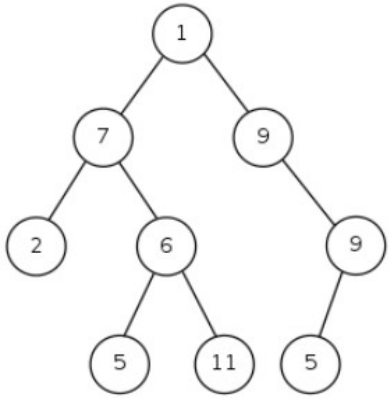
From the following images, select the ones that follows the rules of Complete Binary Tree



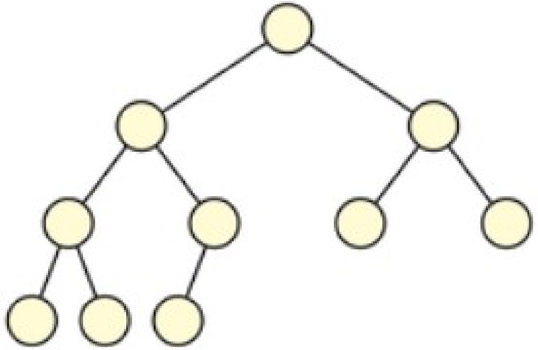
(1)



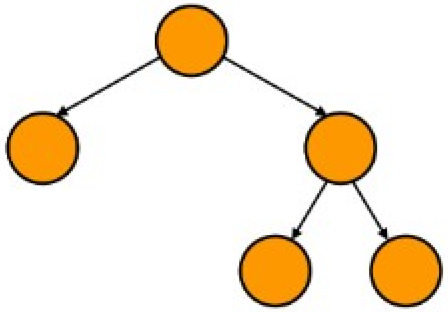
(2)



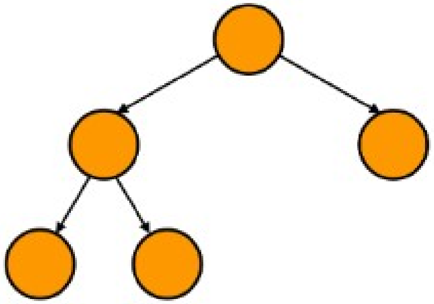
(3)



(4)

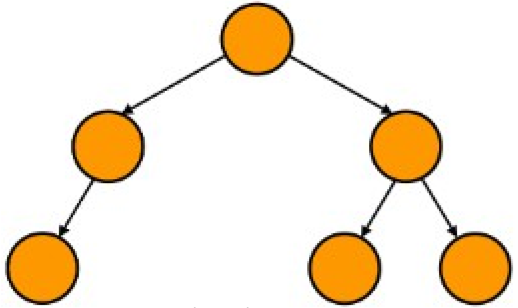


(5)



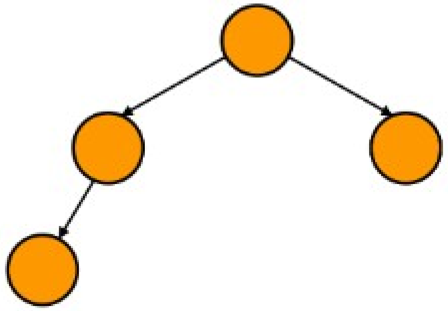
(6)

CHECK POINT



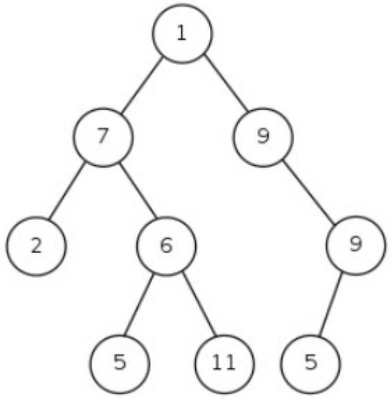
✗

(1)



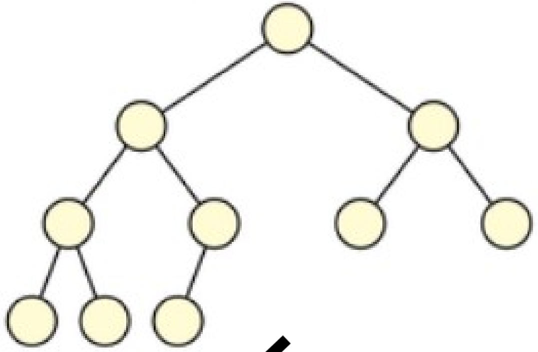
✓

(2)



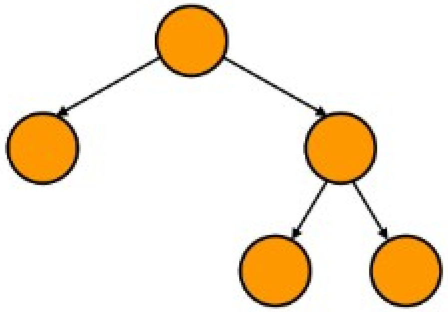
✗

(3)



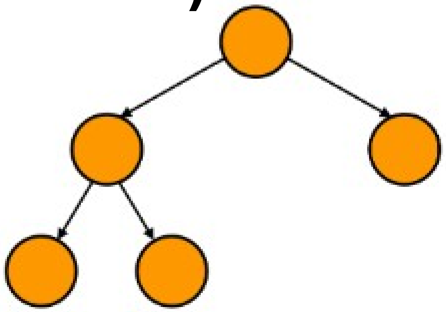
✓

(4)



✗

(5)



✓

(6)

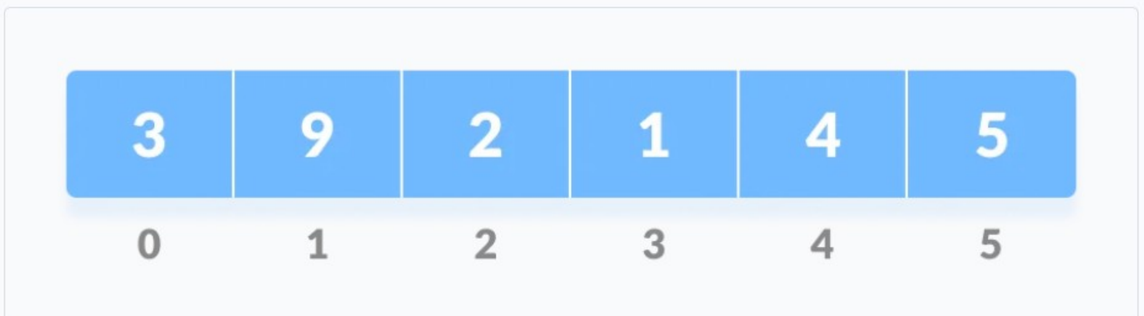
PROPERTIES

Heap Data Structure has the following rules:

- The tree must satisfy the rules of complete binary tree
- Must satisfy Heap Ordering
  - Max Heap : Value of each node is <= Value of its parent
  - Min heap : Value of each node is >= Value of its parent

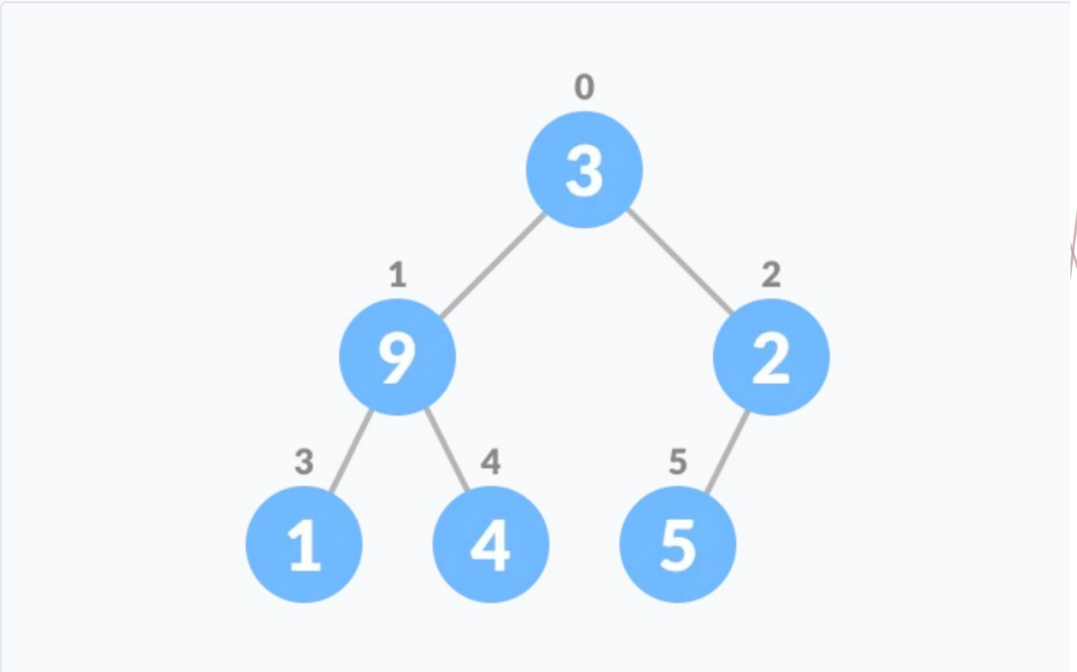
HEAP IMPLEMENTATION USING ARRAY

ARRAY TO HEAP REPRESENTATION



Initial Array

For a root element at index 'i'
Index of LeftChild = 2i + 1
Index of RightChild = 2i+2

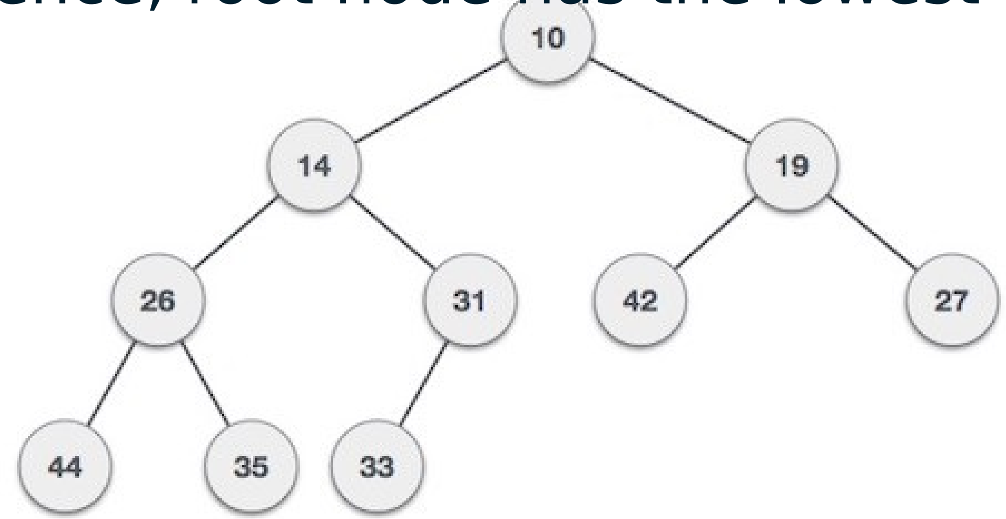


Complete binary tree

TYPES OF HEAP

MIN HEAP

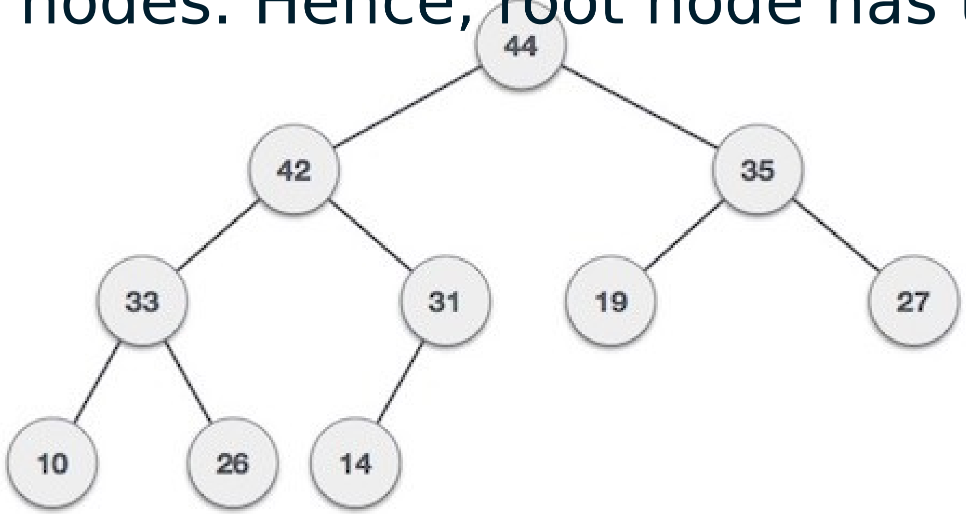
- Value of every node is less than or equal to its decedent nodes. Hence, root node has the lowest value



10	14	19	26	31	42	27	44	35	33
0	1	2	3	4	5	6	7	8	9

MAX HEAP

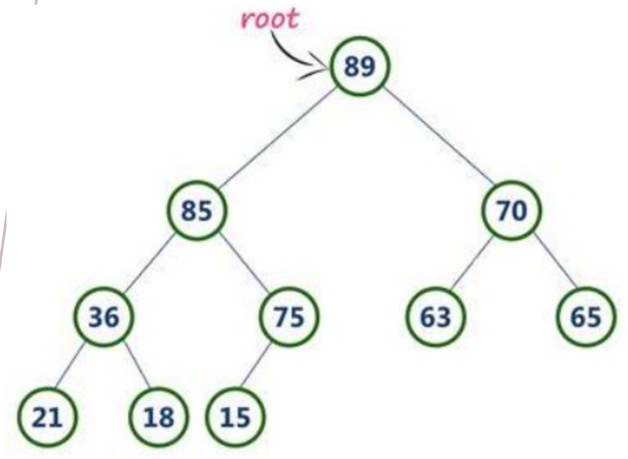
- Value of every node is greater than or equal to its decedent nodes. Hence, root node has the highest value



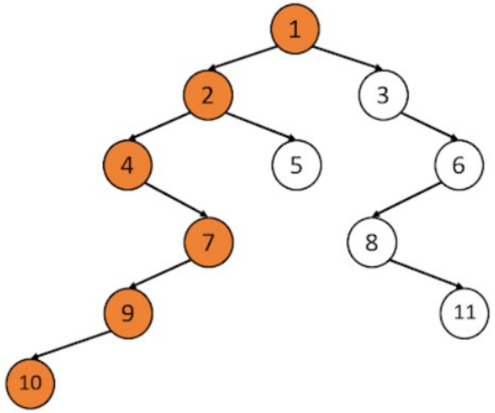
44	**42**	**35**	**33**	**31**	**19**	**27**	**10**	**26**	**14**
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

THINK TONK

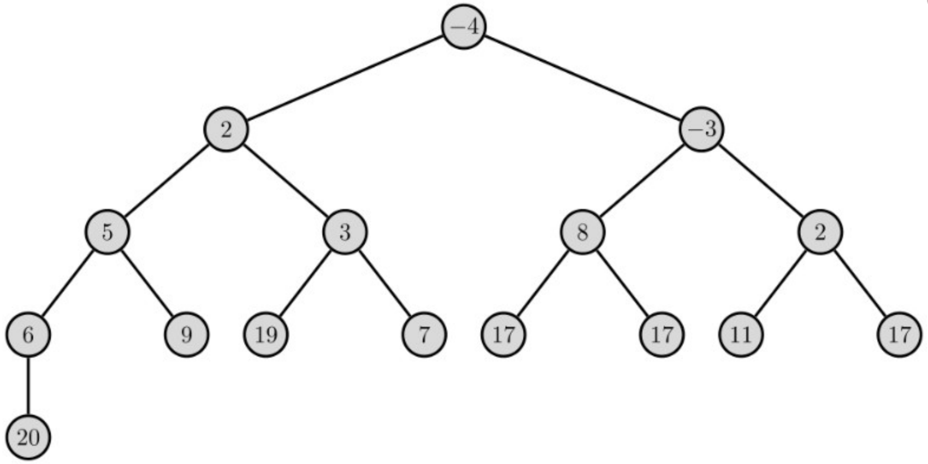
From the following images, validate if it's a heap and if yes, mark them as min/max heaps



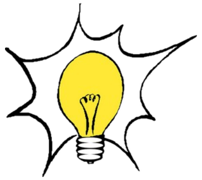
(1)

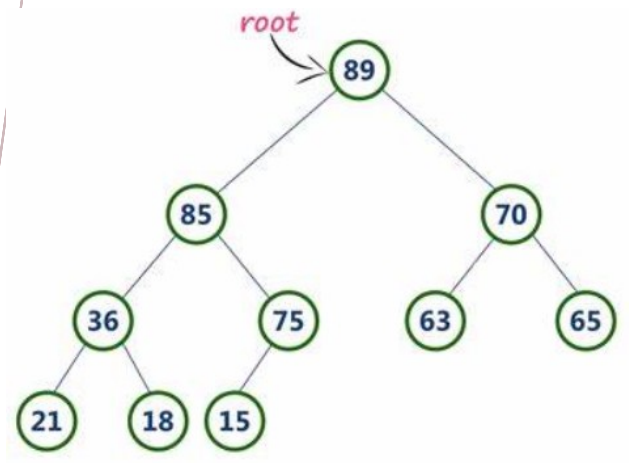


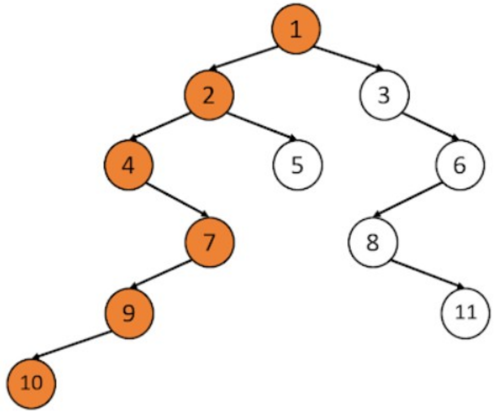
(2)

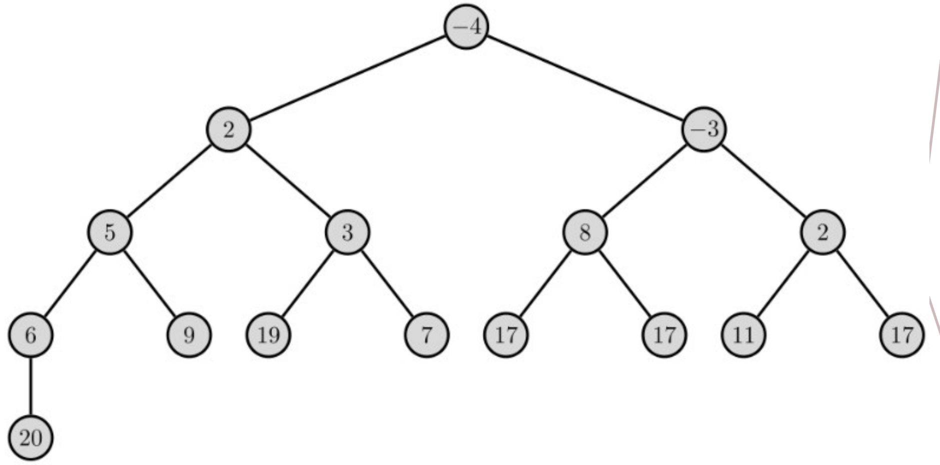


(3)

CHECK POINT 







MAX HEAP (1)

✕ (2)

MIN HEAP (3)

HEAP OPERTIONS

- HEAPIFY
- INSERT ELEMENT INTO HEAP
- DELETE ELEMENT FROM THE HEAP

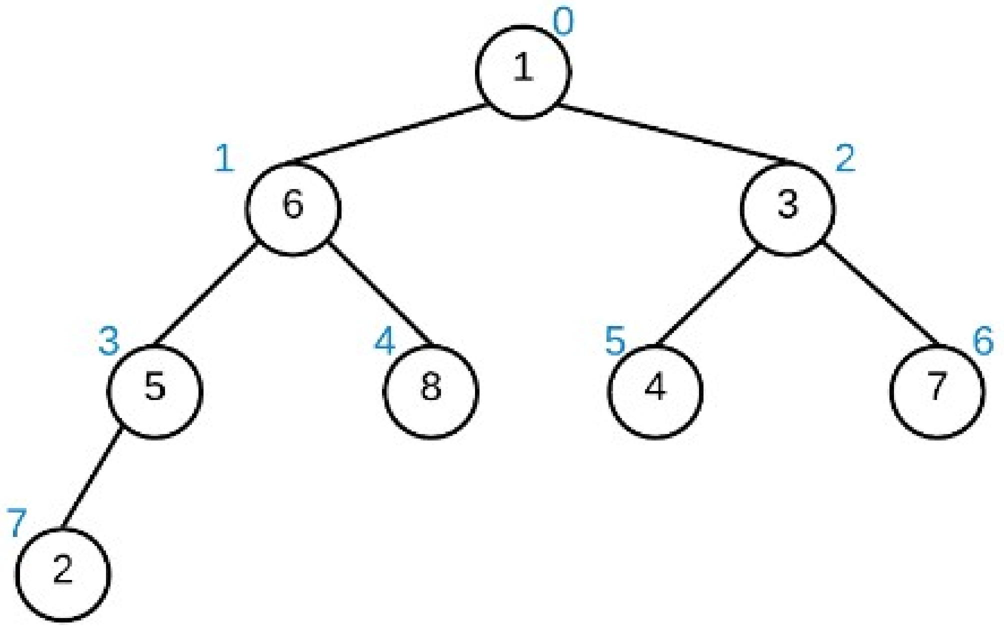
HEAPIFY

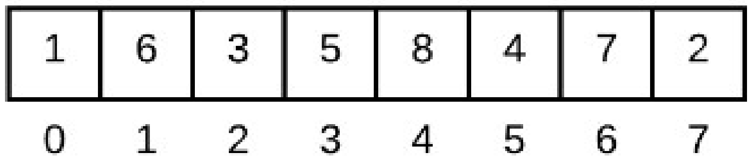
- The process of maintaining Heap ordering given a non-heap structure

- **Algorithm for maintaining Max Heap**
  - if **currentNode** is greater than children, then the heap properties are satisfied
  - Else
    - Swap **currentNode** with **Max** child
    - Heapify() that subtree

HEAPIFY ALGORITHM

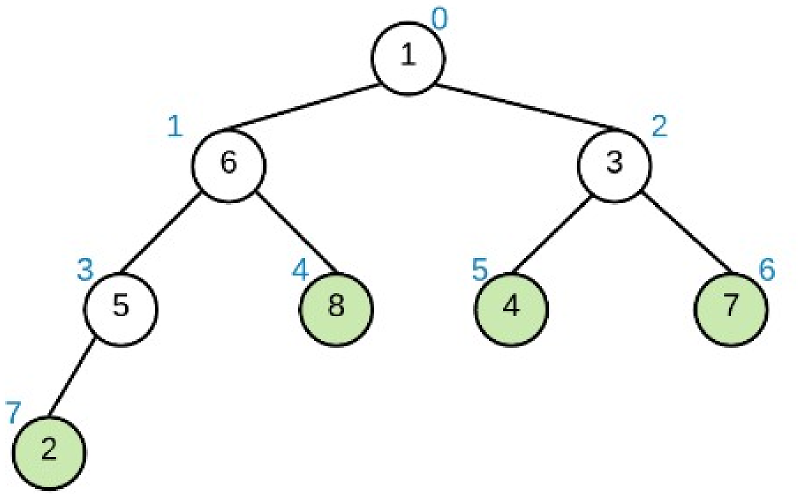
- Following shows the step by step heapify() recursive algorithm on the non-heap array that does not satisfy





HEAPIFY ALGORITHM

- All leaf nodes are valid heaps

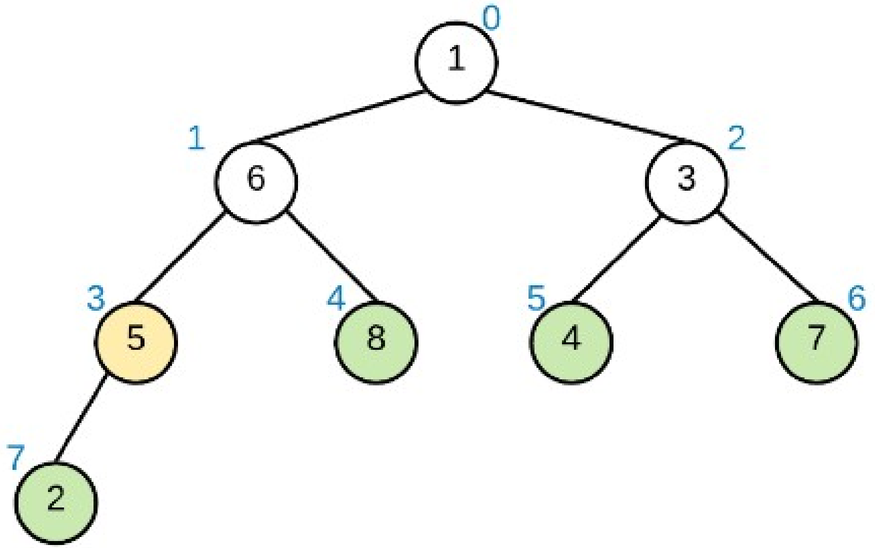


1	6	3	5	8	4	7	2
0	1	2	3	4	5	6	7

- Heapify() loops from first index of non-leaf node to index zero.
  - Index of first non-leaf node : floor[n/2] -1 where n is the total number of elements in the array

HEAPIFY ALGORITHM

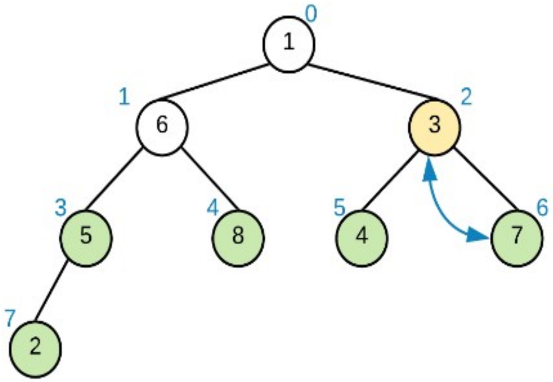
- Heapify(3)
  - First subtree is root element 5 at index 3
  - No swap is needed as the subtree follows heap properties



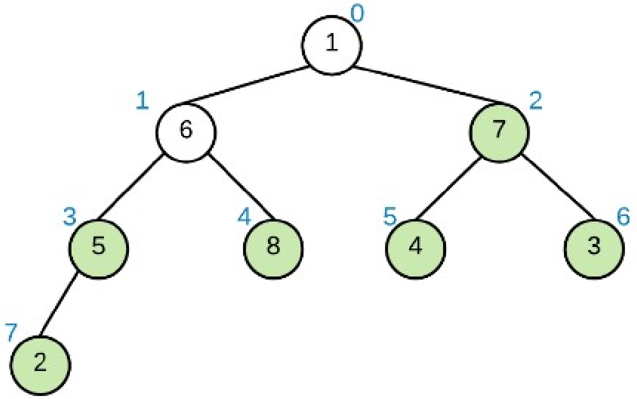
1	6	3	5	8	4	7	2
0	1	2	3	4	5	6	7

HEAPIFY ALGORITHM

- Heapify(2)
  - Second subtree is root element 3 at index 2
  - Children elements 4 and 7 > Parent element 3
  - Swap Parent 3 with element 7



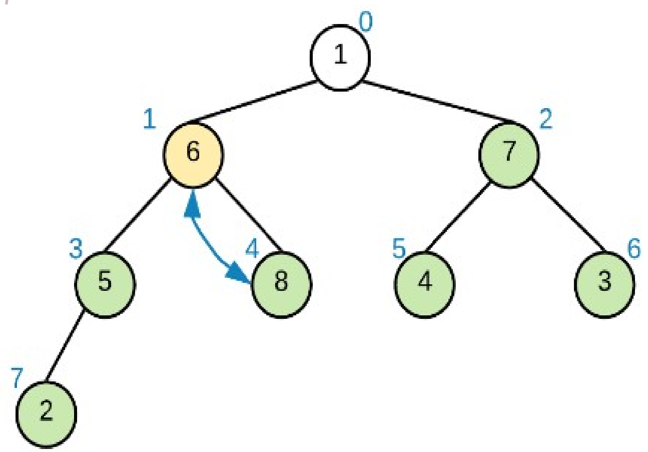
1	6	3	5	8	4	7	2
0	1	2	3	4	5	6	7

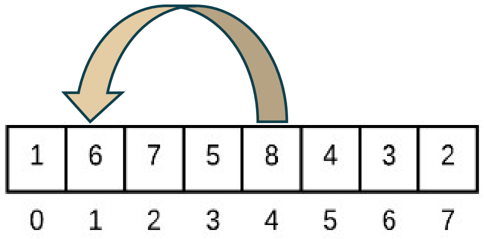


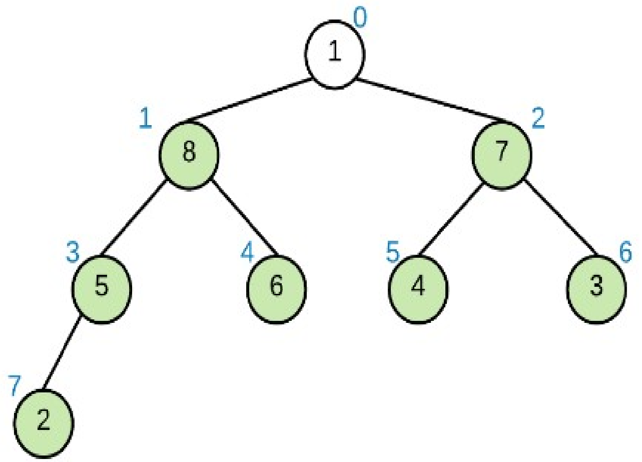
1	6	7	5	8	4	3	2
0	1	2	3	4	5	6	7

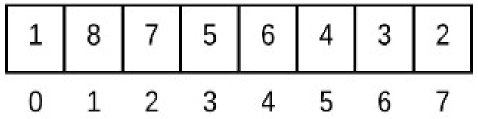
HEAPIFY ALGORITHM

- Heapify(1)
  - Third subtree is root element 6 at index 1
  - Child element 8 > Parent element 6 (Swap)



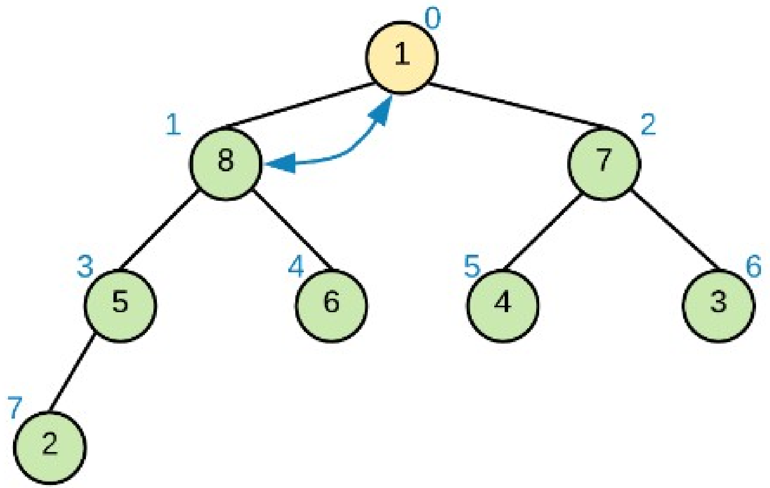


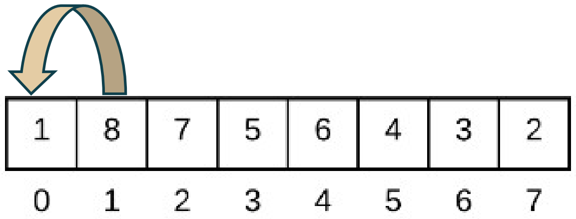




HEAPIFY ALGORITHM

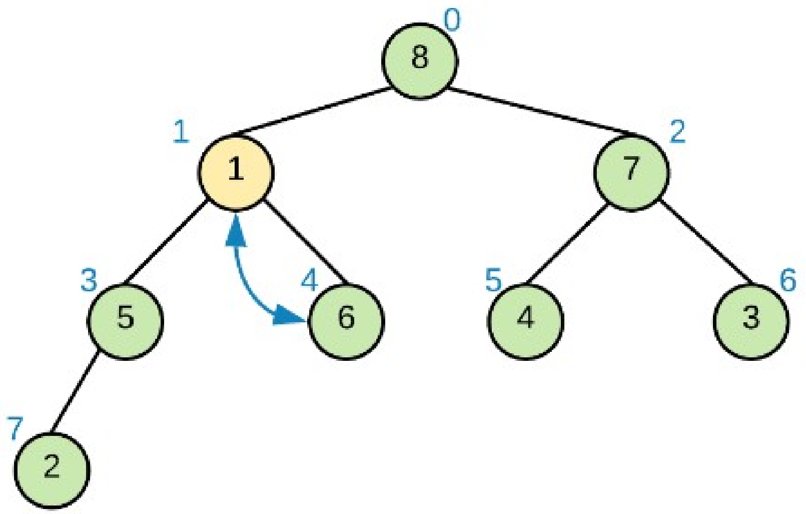
- Heapify(0)
  - Fourth subtree is root element 1 at index 0
  - Child element 8 > Parent element 1 (Swap)

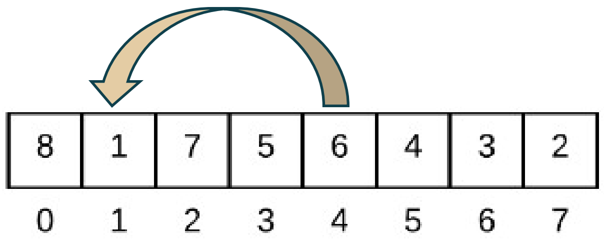




HEAPIFY ALGORITHM

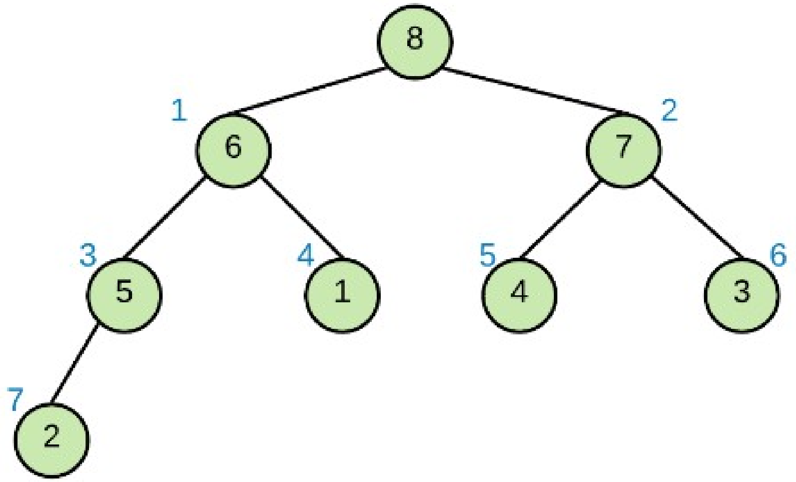
- Heapify(0)
  - Subtree with root element 7 is already heapified
  - Subtree with root element 1 needs heapification
  - Children elements 6 and 5 > Parent element 1 (Swap)





HEAPIFY ALGORITHM

- Final Max Heap after Heapification
  - Subtree with element 7 is already heapified
  - Subtree with element 1 needs heapification
  - Children elements 6 and 5 > Parent element 1 (Swap)



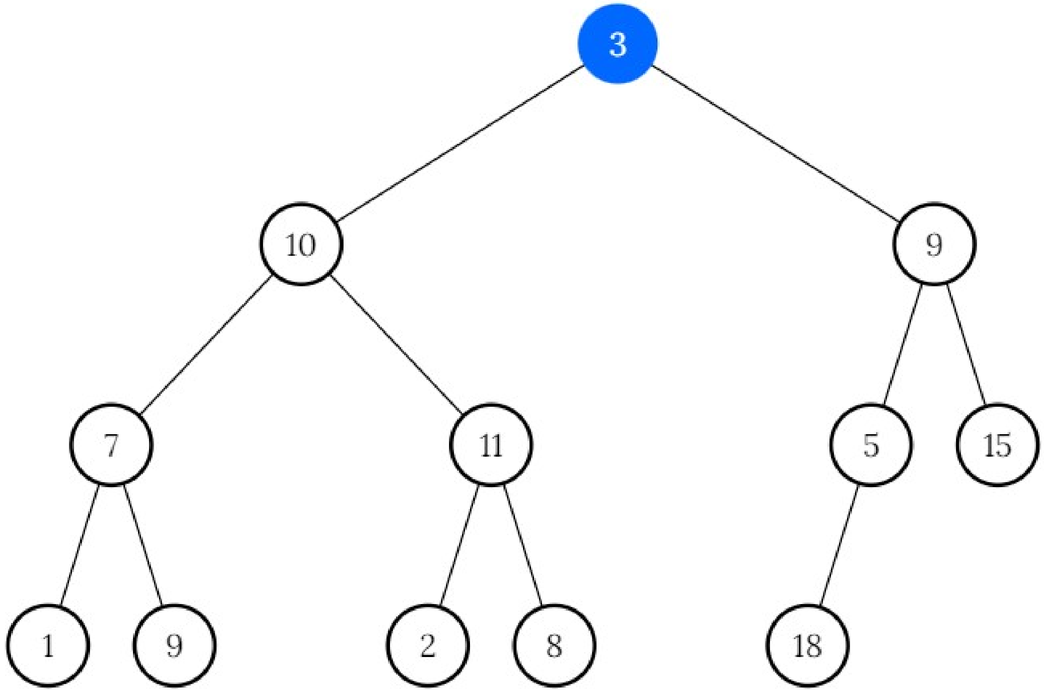
8	6	7	5	1	4	3	2
0	1	2	3	4	5	6	7

HEAPIFY ALGORITHM

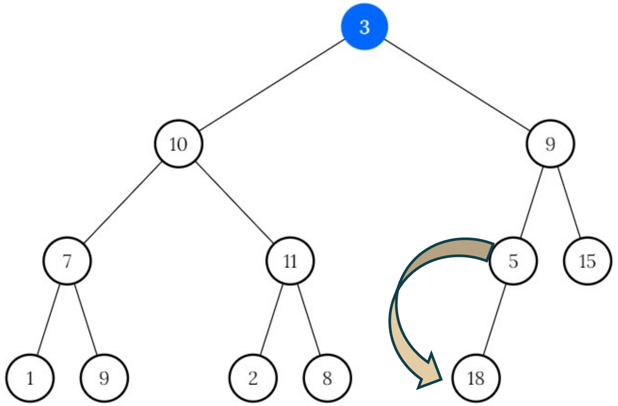
```
Heapify(array, i) {
  largest = i
  leftChild = 2i + 1
  rightChild = 2i + 2

  if array[leftChild] > array[largest] {
    if array[rightChild] > array[leftChild] {
      set rightChild as largest }
    else
      set leftChild as largest }}
  else if array[rightChild] > array[largest] {
    set rightChild as largest  }

  if largest != I {
    swap array[i] and array[largest]
    Heapify(array, largest) } }

MaxHeap(array, size) {
  loop from the i = (n/2)-1 to zero {
    Heapify(array, i) }}
```

LET'S HEAPIFY..

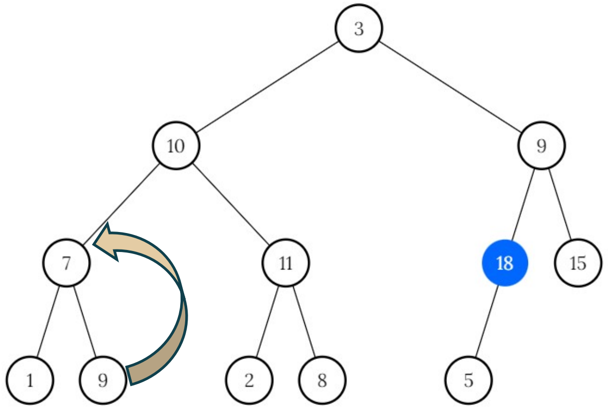
- Heapify the following non-heap structure that follows max heap property



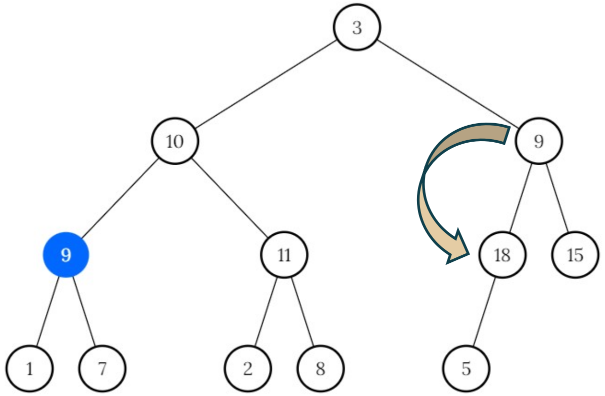
HEAPIFICATION



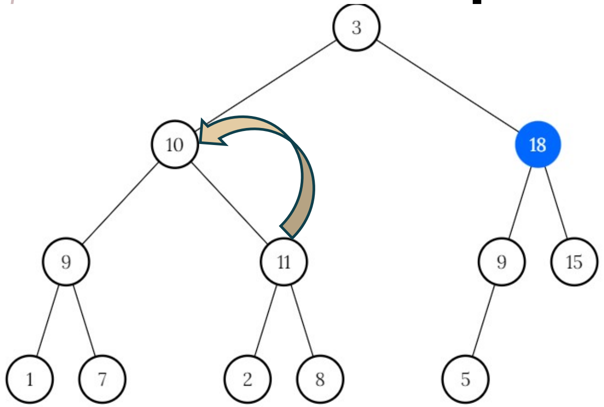
(1



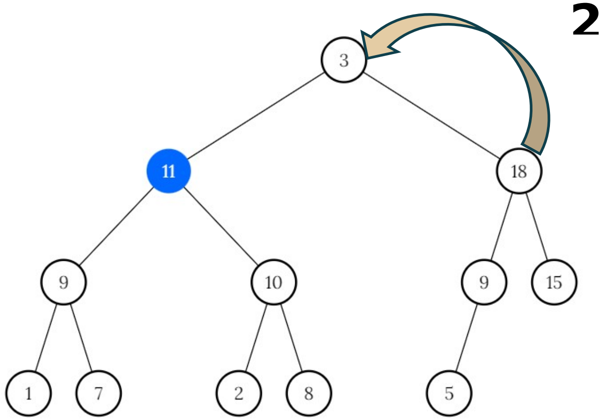
(2



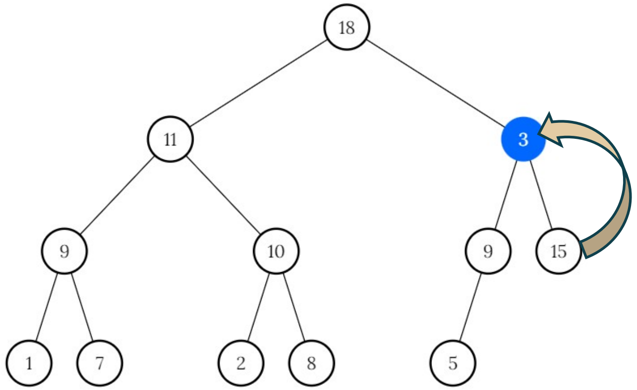
(3



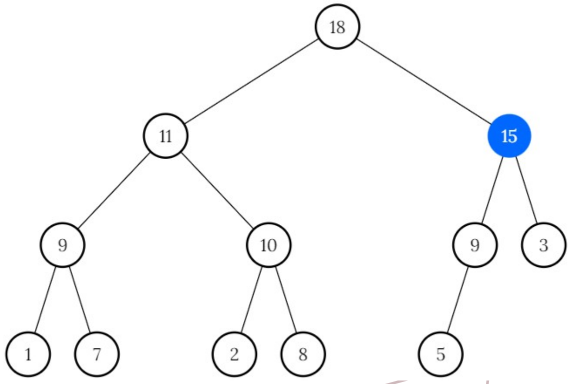
(4



(5



(6

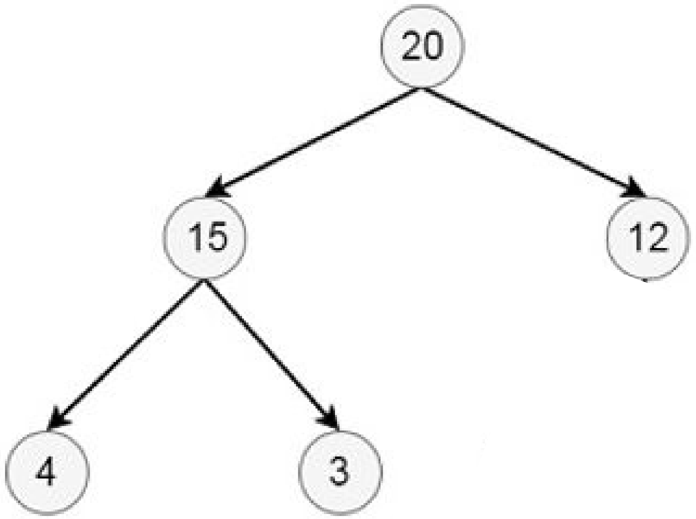


(7

INSERT OPERATION

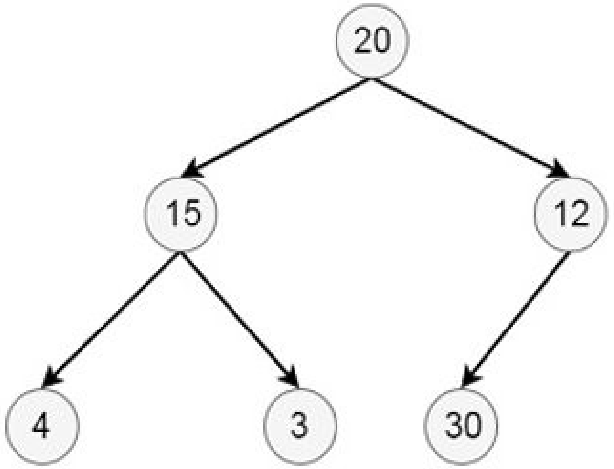
Insert new element at the end of the heap and heapify

- Add element 30 in the following heap



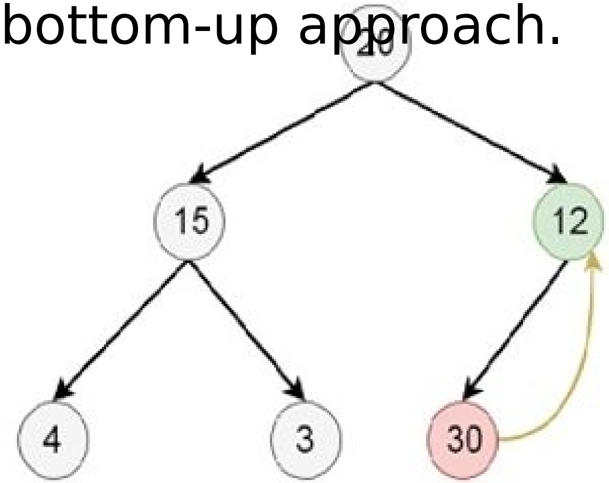
20	**15**	**12**	4	**3**
0	**1**	**2**	3	**4**

- Increase the heap size by 1
- Insert the new element at the end of the Heap.

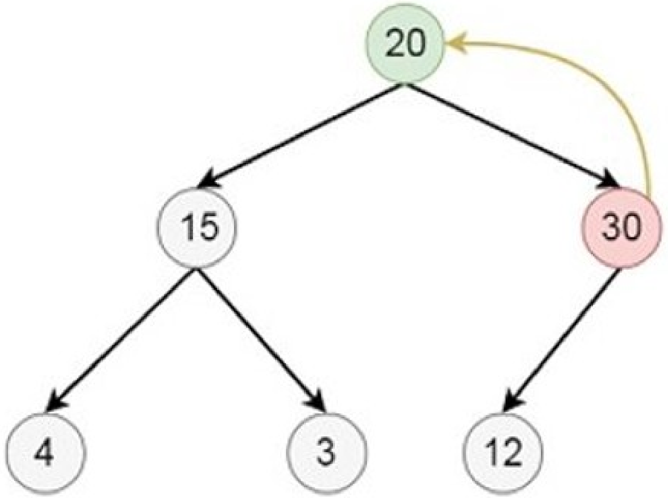


20	15	12	4	3	**30**
0	1	2	3	4	**5**

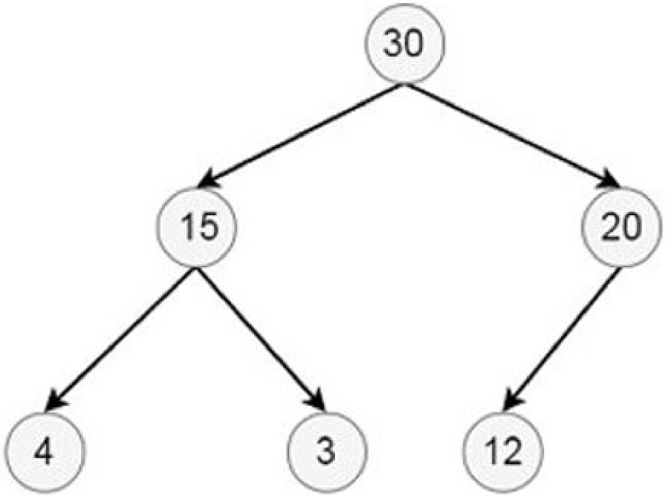
- Heapify this newly inserted element following a bottom-up approach.



20	15	**12**	4	3	**30**
0	1	2	3	4	5



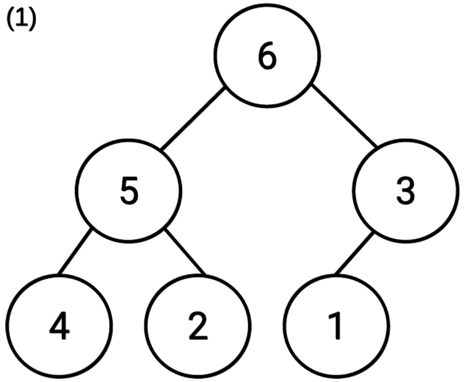
20	15	**30**	4	3	12
0	1	2	3	4	5



30	**15**	**20**	**4**	**3**	**12**
0	**1**	**2**	**3**	**4**	**5**

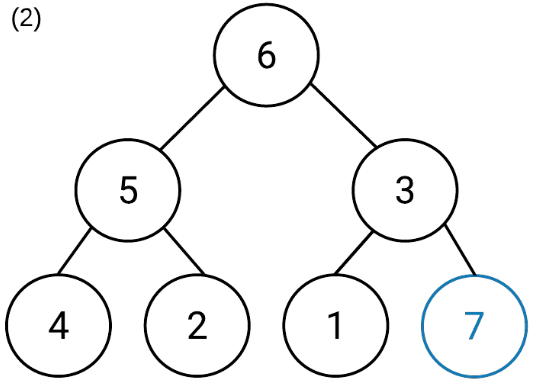
Inserting 7 into this heap

(1)



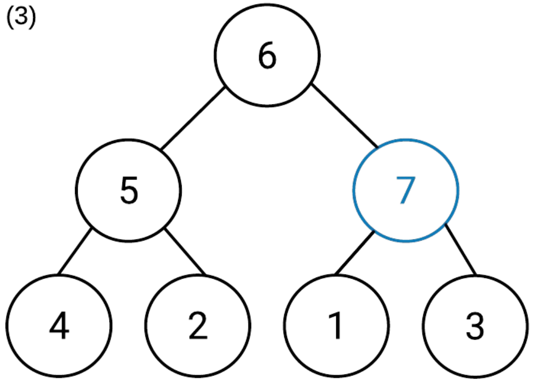
Starting with this max heap

(2)



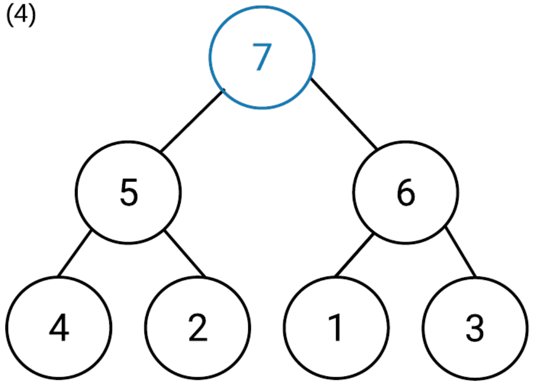
Step 1: 7 is inserted at the bottom most, right most position

(3)



Step 2: Because 7 is bigger than its parent, the 3 node, it gets swapped

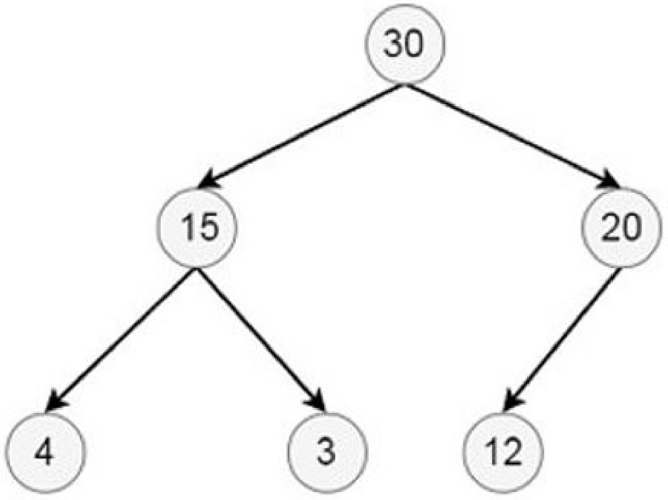
(4)



Step 3: Once again, 7 is bigger than its parent, the 6 node, so it gets swapped

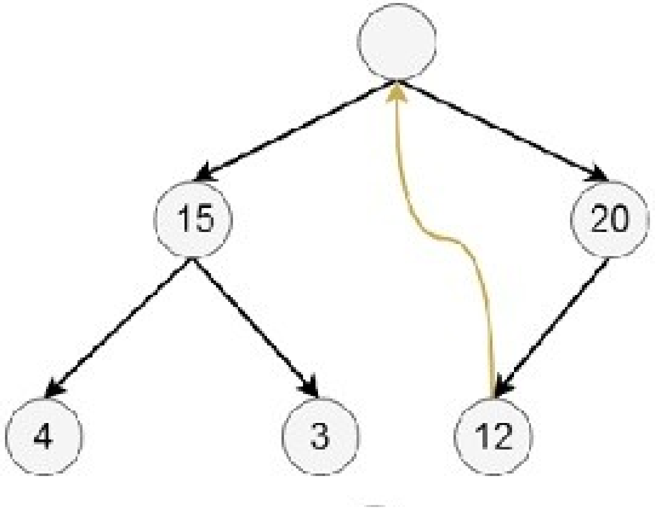
DELETE OPERATION

- Let us delete root element in the below heap



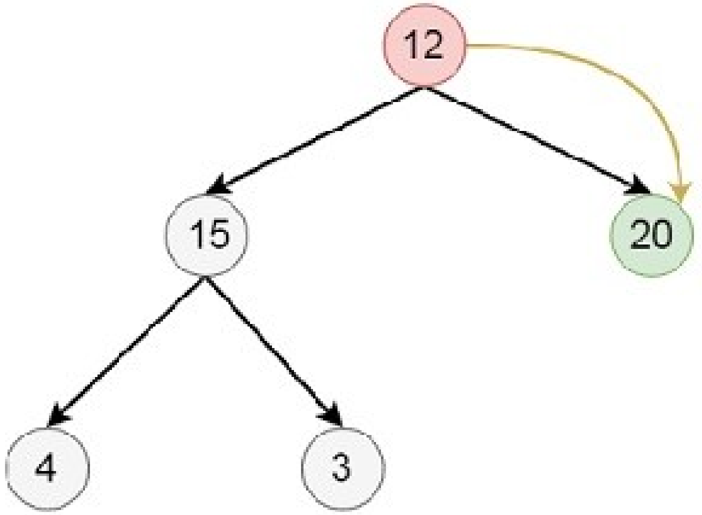
30	15	20	4	3	12
0	1	2	3	4	5

- Replace the root or element to be deleted by the last element.
- Delete the last element from the Heap.

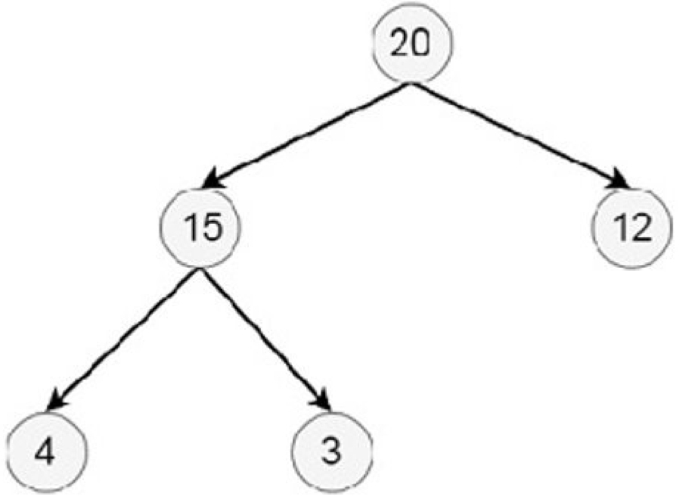


30	15	20	4	3	**12**
0	1	2	3	4	5

- Heapify the last node placed at the new position.



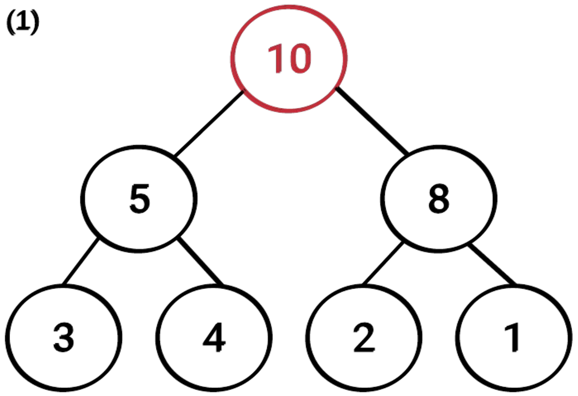
12	15	**20**	4	3
0	1	2	3	4



20	**15**	**12**	**4**	**3**
0	**1**	**2**	**3**	**4**

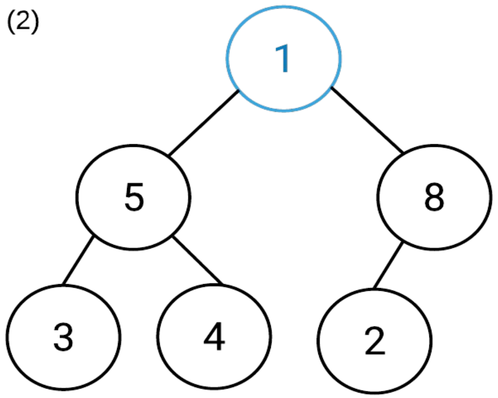
Deleting from this heap

(1)



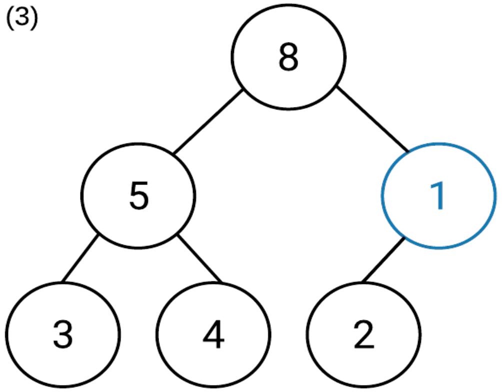
Starting with this max heap

(2)



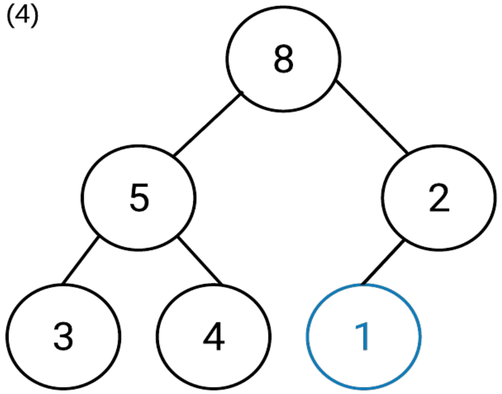
Step 1: the bottom most, left most node, the 1 node, gets placed at the root

(3)



Step 2: Because 1 is less than both of its children, it swaps with the larger element, the 8 node

(4)



Step 3: Once again, 7 is bigger than its parent, the 6 node, so it gets swapped

TIME COMPLEXITY OF HEAP OPERATIONS

- Time Complexity Using Binary Tree

	Best Case	**Worst Case**
Insert	O(1)	$O(\log_2 n)$
Delete	O(1)	$O(\log_2 n)$

APPLICATIONS OF HEAP

- **Heap Sort** - Uses binary heap to sort an array in O(nlogn) time
- **Priority Queue –** Uses binary heap to implement insert/delete/update operations in O(logn) time
- **Graph Algorithms** - Dijkstra's algorithm and Minimum Spanning Tree use heap as internal traversal data structures to reduce time complexity

THANK YOU!