Binary Heap & Priority Queue

Binary Heap

A Binary heap is a heap data structure that takes the form of a binary tree.

- We can also call this as 'heap' only. Since, it is a ~~cap~~ complete Binary Tree, we use to call it Binary Heap.

* Difference Between Complete & Full Binary Tree

Conditions of a Complete Binary Tree :-

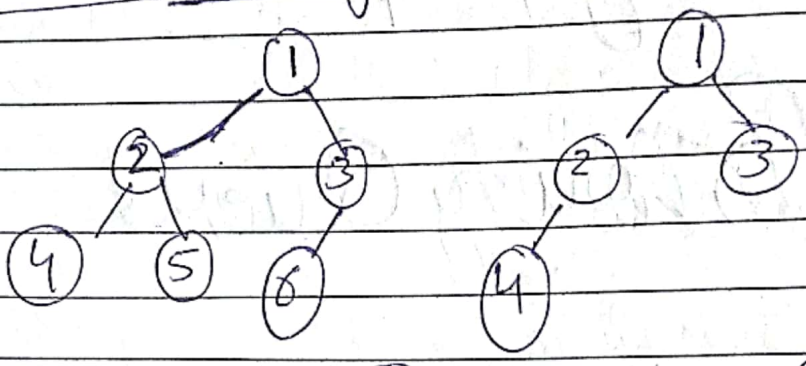
- All the leaf element / node must lean towards the left.
- The last leaf element / node might not have a right sibling i.e. a complete Binary Tree need not to be a full Binary Tree.

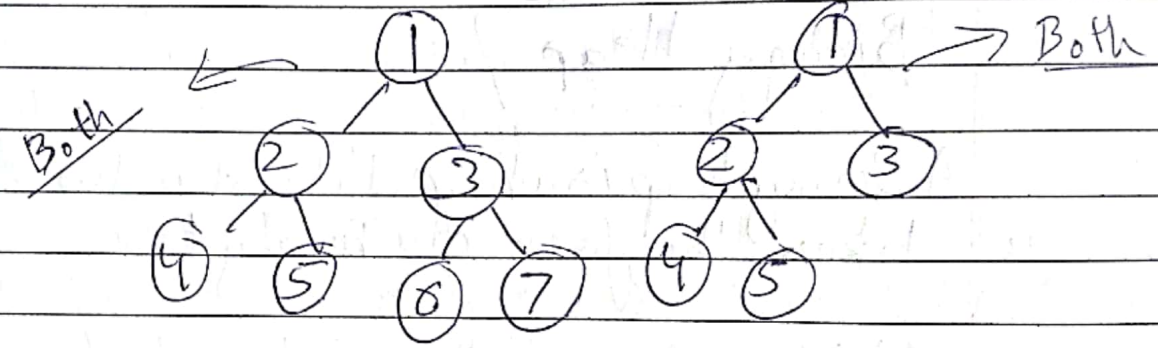
Conditions of a Full Binary Tree :-

- Every Parent node ~~either~~ either have Full (two) Child node or Zero (0) child node.
- A full Binary tree of a given height h has $2^h - 1$ nodes.

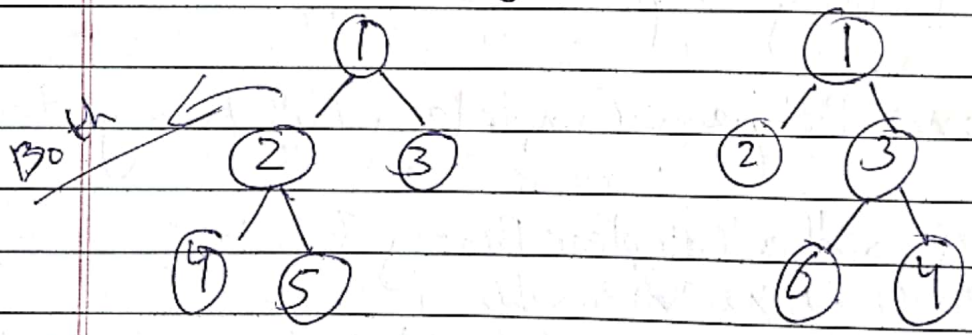
Eu:

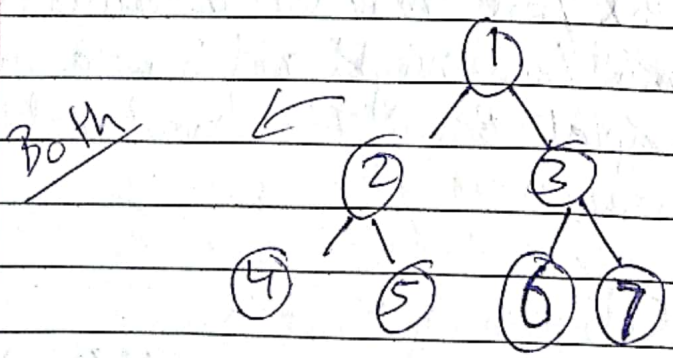
Complete Binary Tree:

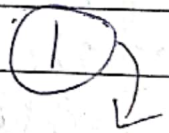




Full Binary Tree:

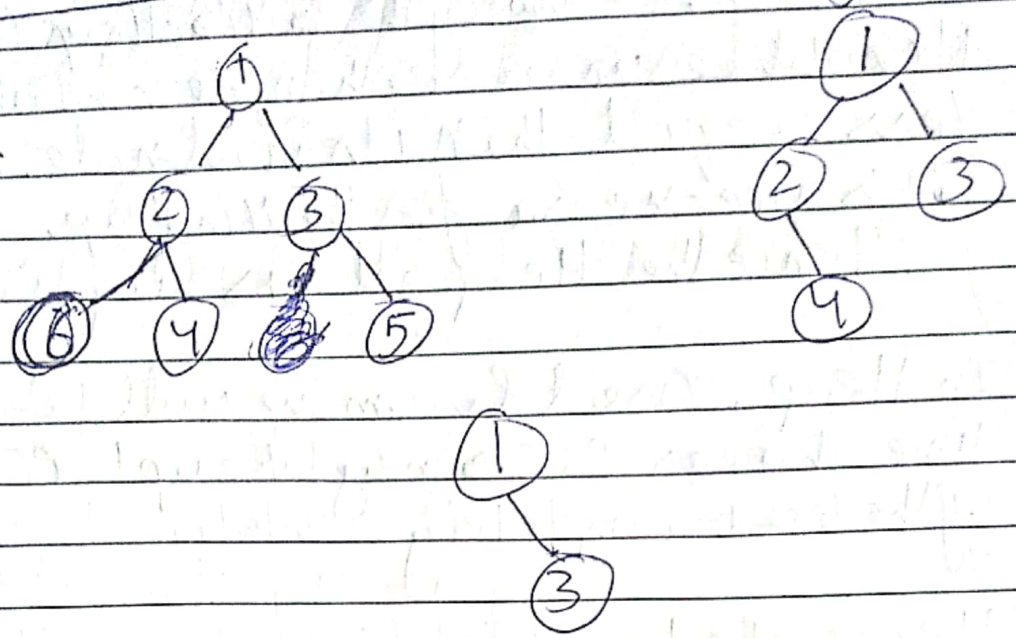






(Full also a perfect Binary tree at h=0)

Neither Complete Nor Full Binary Tree :-



- In Binary heap, a node at level k of the tree if the distance between this node and the root is k. The level of the root is 0. The maximum possible number of nodes at level k is 2^{k}.

* Difference b/w BST & Binary Heap.

- BST is an ordered data structure, however the Heap is not. So, if order matters, then it is better to use BST.

- In BST, every node's value is strictly greater than the value of its left child and strictly lower than the value of its right child. Hence, this data structure doesn't allow duplicate value. and makes easy to iterate all the values of the BST in sorted order.

On the other hand, Binary heap can either be Min-heap or Max-Heap. The rule of the Min-Heap is that the subtree under each node contains values less or equal than its root node, whereas it's vice-versa for the Min-Heap.
Hence the Heap allows duplicates.

- In Heap, Insert & remove will take O(log(n)) time whereas BST may take upto O(n) time, if the tree is completely unbalanced.
- Heap can be built in linear time, However the BST needs O(n log(n)) to be created.

Binary Heap

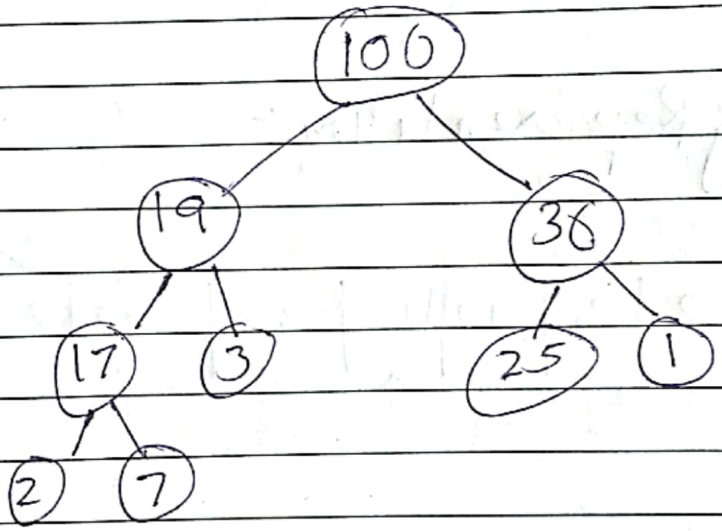
- Since, Heap is an unordered DS (data structure). The only possible way to get all its element in sorted order is to remove the root of the tree n times.
This algorithm is also called Heap Sort and takes O(n log(n)) time.
But BST is ordered DS. So we can easily get all the values of the BST in sorted order by using InOrder Traversal throughout the tree.
- The heap can be either Min-Heap or Max-heap.
- In computer memory, the heap is usually represented as an array of numbers.

* Max Binary Heap :-

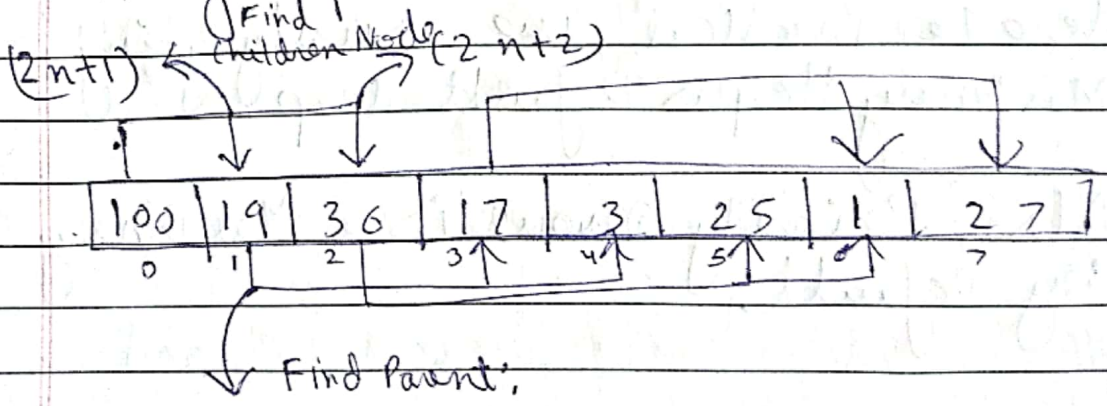
- Parent Nodes are always Greater than child Nodes but there are no guarantees between sibling nodes. ~~Because Heap is a complete Binary tree.~~

- All the children of each node are as full as they can be and left children are filled out first. Because Heap is a Complete Binary tree.

Tree Representation :



Array Representation



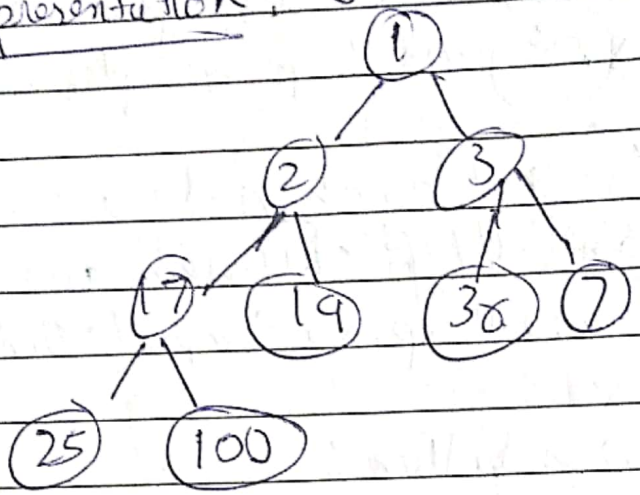
Find Parent :

Math.floor(n-1/2)

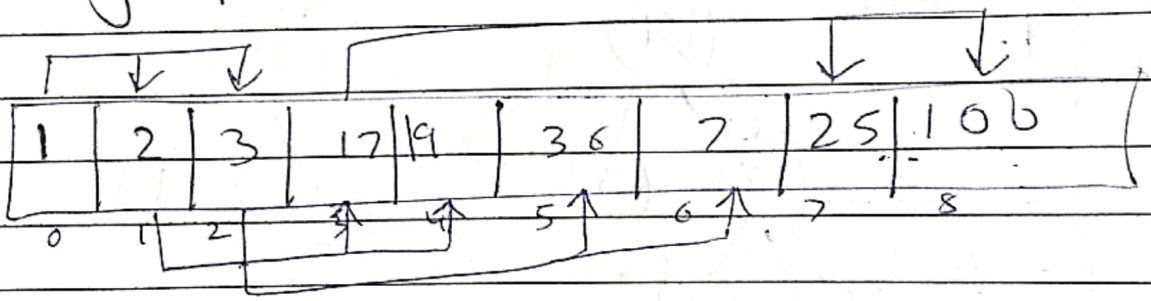
∵ n is the index of array for children & Parent Node)

* Min Binary Heap :-
Parent Node always less than child Nodes

Tree Representation :



Array Representation :



Note :

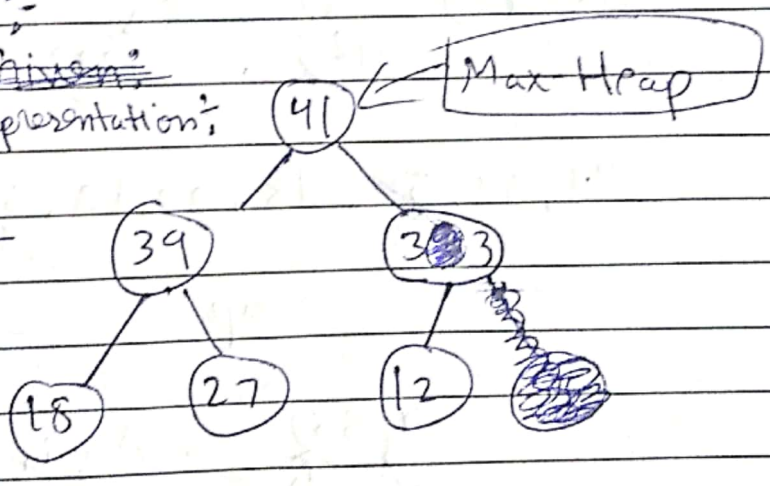
- Here For Practical, we usually going to use Max Binary Heap as Default Heap.
- Also, Priority Queue is a Max Binary Heap by Default.

* Adding to a Heap (By Default Max-Heap):

- Push the value into the values property on the Heap.
  Values property is would be an array of numbers.
  As we push means adding number at the beginning of the values array or say at the ~~rear~~ end of the Binary Heap.
- Bubble the value up to its correct spot as per the property of max or min heap that we're using.

Steps:-

(i) Tree Representation:

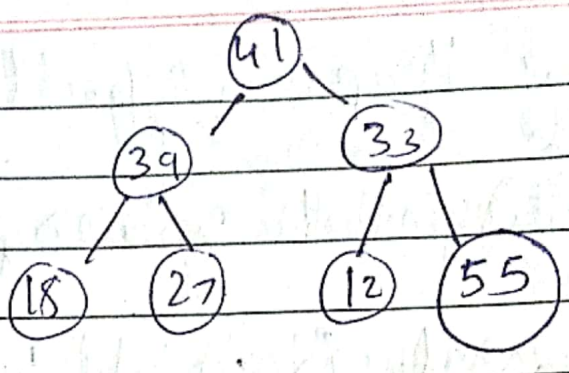


Given

Array Representation:-

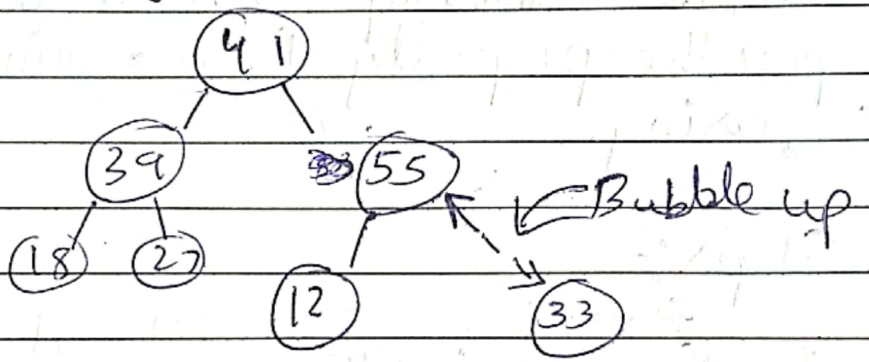
[41, 39, 33, 18, 27, 12]

(ii). We want to add 55 as value.



- [41, 39, 33, 18, 27, 12, 55] ← Push

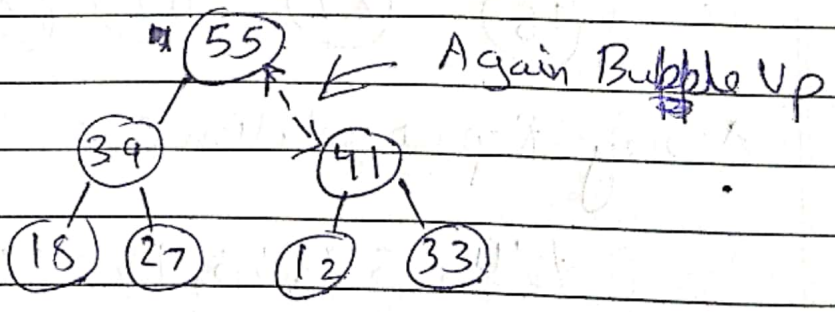
(iii) Bubble up the value (if the value is not arranged as per the rule of max-Heap)



[41, 39, 55, 18, 27, 12, 33]

Swap

(iv)



[55, 39, 41, 18, 27, 12, 33]

Now, the value is completely arranged as per Max-Binary Heap.

→ Code Implementation :-

```
class ~~Max~~ Binary Heap {
    Constructor() {
      this.maxHeapValues = [41, 39, 33, 18, 27, 12]
     This.minHeapValues = [1, 2, 3, 19, 36, 7, 25, 100]
}
addingToMaxHeap(ele) {
     this.~~val~~maxHeapValues.push(ele);
    Let idx = this.maxHeapValues.length - 1,
      element = this.maxHeapValues[idx];

  while(idx > 0) {
    let parenIdx = Math.floor((idx-1)/2);
      parenEle = this.maxHeapValues[parenIdx];

      if(parenEle > element) break;

      this.maxHeapValues[idx] = parenEle;
      This.maxHeapValues[parenIdx] = element;
        idx = parentIdx;
  }
    Return this.maxHeapValues;
}

addingToMinHeap(ele) {
```

Everything would be same as it is in addingtoMaxHeap().
Just change maxHeapValues to minHeapValues & change if(parenEle > element) to if(parenEle < element)

```
}  }
```

```
const heap = new BinaryHeap();
heap.addingToMaxHeap(55);
heap.addingToMinHeap(17);
```

Output:

[55, 39, 41, 18, 27, 12, 33]

[1, 2, 3, 17, 36, 7, 25, 100, 19]

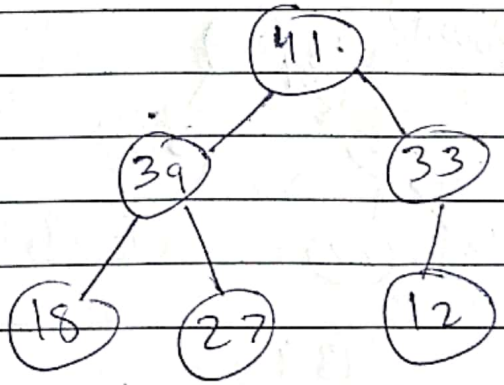
* Removing from a Heap (By default Max-heap) (also known as Extract Max) or Extract Min)

- Sink Down:
The procedure for deleting the root from the heap (effectively extracting the maximum element in a max-heap or the minimum element in a min-heap) and restoring the properties is called down-hep (also known as bubble down).

- Procedures:
  - Remove the root
  - Replace with the most recently added or say the last element from the Heap or say the very first element from the array.
  - Adjust (sink down)

- Steps:-

Given :

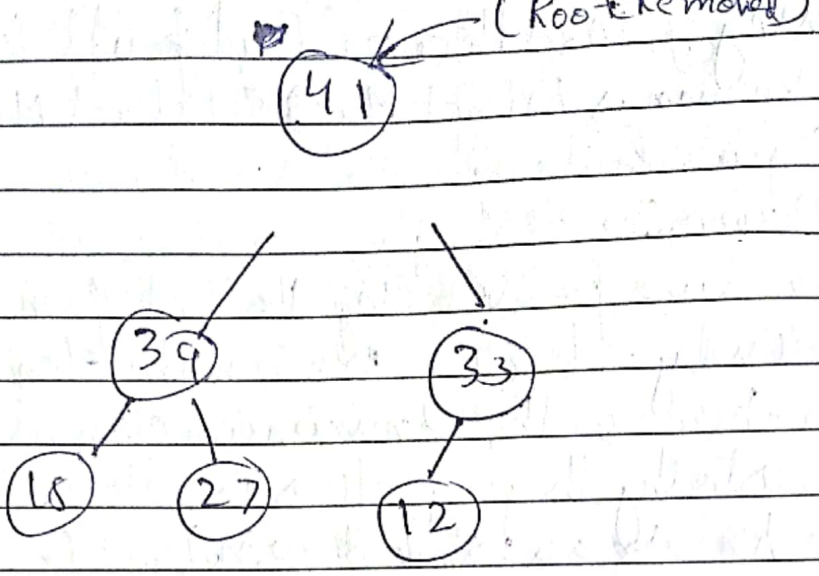


Tree Representation

Array Representation

[41, 39, 33, 18, 27, 12]

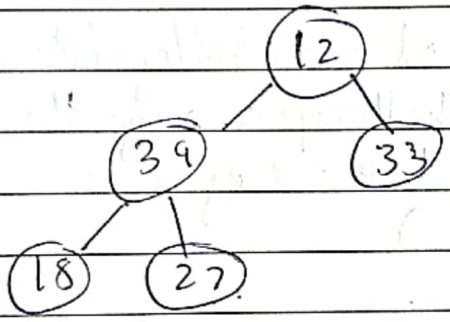
(i)



[41, 39, 33, 18, 27, 12]

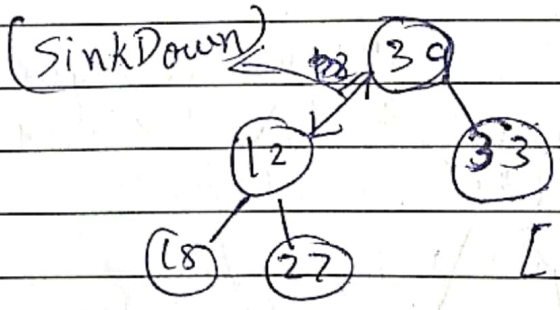
swap

(ii)



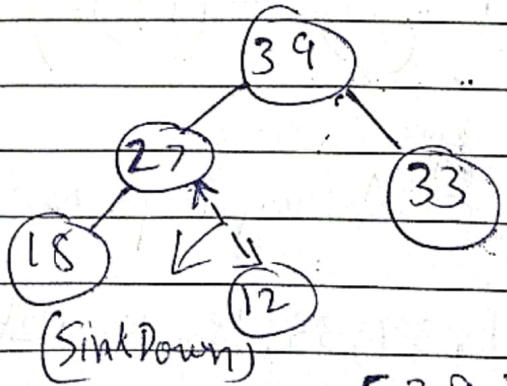
[12, 39, 33, 18, 27]

(iii)



[39, 12, 33, 18, 27]

(iv)



[39, 27, 33, 18, 12]

→ Code Implementation :-

```
class MaxBinaryHeap {
    constructor() {
        this.values = [41, 39, 33, 18, 27, 12];
    }
    removing() {
        const max = this.values[0];
        let end = this.values.pop();
        const sinkDown = this.sink_down(end);
        return sinkDown;
    }
    sink_down(end) {
        let idx = 0;
        while (true) {
            let element = end,
                leftIdx = (2*idx + 1),
                rightIdx = (2*idx + 2),
                leftChild = this.values[leftIdx],
                rightChild = this.values[rightIdx],
                swapIdx = null;

            if (leftIdx < this.values.length) {
                if (element < leftChild && leftchild > rightchild) {
                    swapIdx = leftIdx;
                }
            }
            if (rightIdx < this.values.length) {
                if (element < rightChild && leftchild < rightchild)
                    swapIdx = rightIdx;
            }
        }
```

```
    if(swap===null) break;
    this.values[idx] = this.values[swapIdx];
    this.values[swapIdx] = element;
    idx = swapIdx;
  }
  return this.values;
}
const maxHeap = new MaxBinaryHeap();
maxHeap.removing();
```

Output :

[39, 27, 33, 18, 12]

* Why do we need to know Binary Heap?

Binary Heaps are use to implement Priority Queues, which are very commonly used data structures.

By default, Priority Queues is a maxBinary Heap.

They are also used quite a bit, with graph traversal Algorithms.

Priority Queue

- It's a data structure where each element has a priority.
- Elements with higher priorities are served before elements with lower priorities.
- Here, value doesn't matter, Heap is constructed using Priority.
- Lower Number means Higher priority. Thus, we're going to use MinBinary Heap.
- Priority Queue is an abstract data type that is similar to a queue, and every element has some priority value associated with it.
- Typically stacks and queues are techically types of priority queues. The highest priority element in a stack is the most recently inserted one (LIFO), while the highest priority element in a queue is the least inserted one (FIFO).

* Our Priority Queue

- Write a Min Binary Heap - Lower number means higher priority.
- Each node has a val and a priority. Use the priority to build the heap.
- Enqueue method accepts a value and ~~prioriting~~ priority, makes a new node, and puts it in the right spot based off of its priority.
- Dequeue method removes root element, returns it, and rearranges heap using priority. (having minimum value)

* NOTE !

Till now, we first add Element in the Heap then we performed Bubbling Up to arrange elements to form MinHeap.

But in Priority Queue, while adding particular Element / Node, we'll perform bubbling Up operation respectively for particular Node.

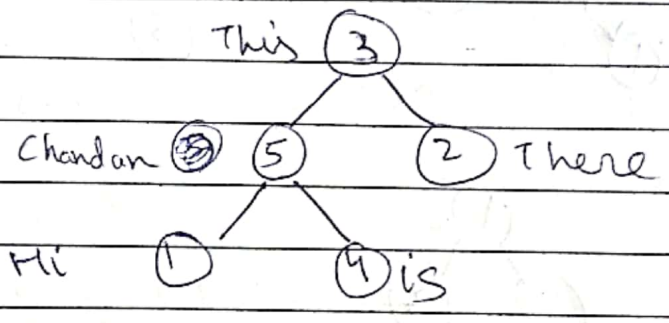
* Steps of Enqueue :-

Steps of Enqueuing is simple, we just put the Node (consisting value associated with priority) into the ~~values~~ values array.

So, values array would look like this after enqueue.

priority	(3)	(5)	(2)	(1)	(4)
values →	'This'	'Chandan'	'There'	'Hi'	'is'
indices →	0	1	3	4	5

Tree Representation :-



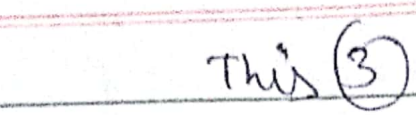
But, the resultant values[] should look like this,
['Hi', 'There', 'This', 'Chandan', 'is']
This would create a Min Binary Heap.

* Steps of enqueue along with Bubbling Up Operation :-

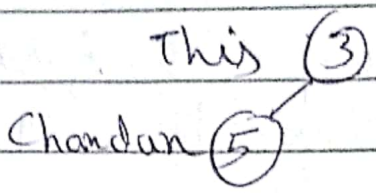
Steps to Enqueue :
(i) 'This' (3)
(ii) Chandan (5)
(iii) There (2)
(iv) Hi (1)
(v) 'is' (4)

values ↙ ↘ Priority

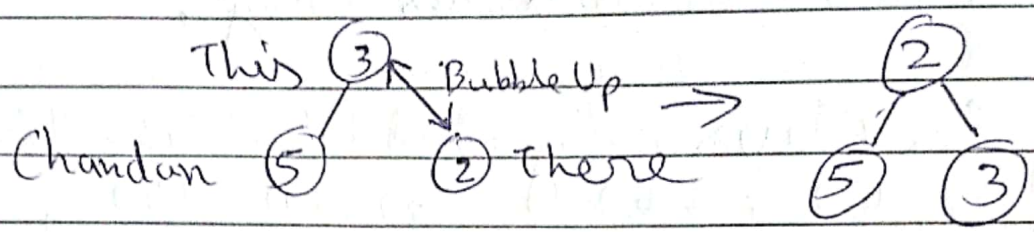
(i)



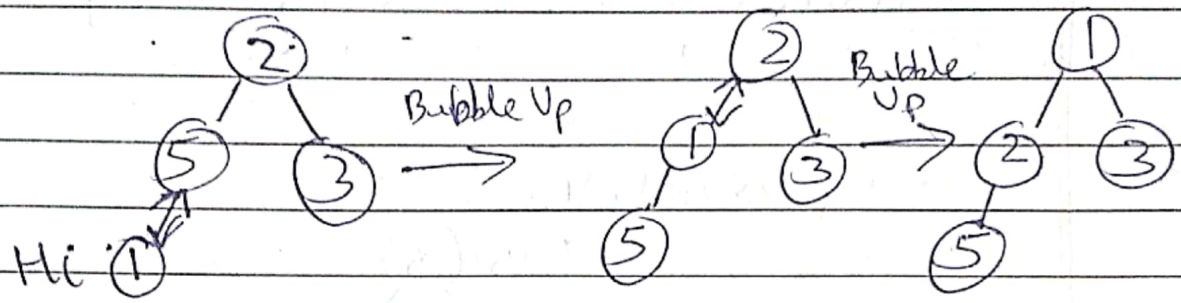
(ii)



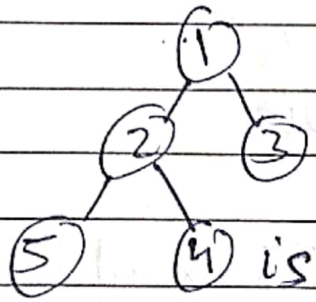
(iii)



(iv)



(v)



Resultant Values [] after Enqueue & Bubble Up:

Priority → (1) (2) (3) (5) (4)

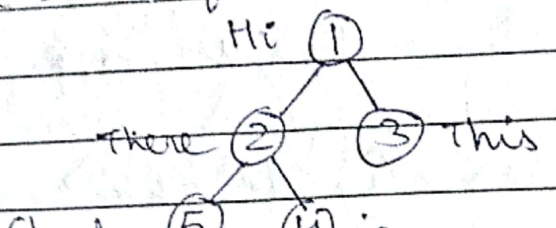
values → ['Hi', 'There', 'This', 'Chandan', 'is']

If we look at the Priority, it is still not sorted. So to sort this we will perform Dequeue & sink-down or bubble down operation.

* Steps to Dequeue & Sink-Down / Bubble Down :-

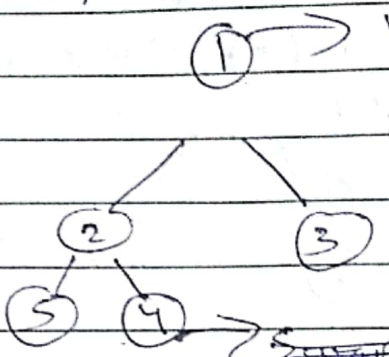
→ Calling for 1st Dequeue :-

Given :-

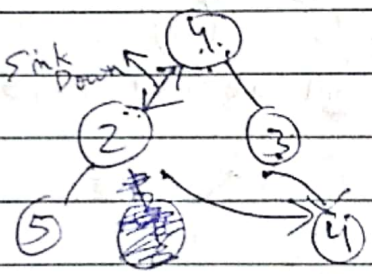


['Hi', 'there', 'This', 'Chandan', 'is']

(i)



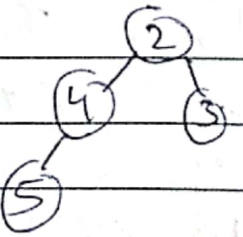
(ii)



(4) is smaller to both (2) & (3).
Out of both (2) is smaller, so (4) swap with (2).

~~(ii)~~

(iii)



Result : Minimum Element/Node :- Hi (1)

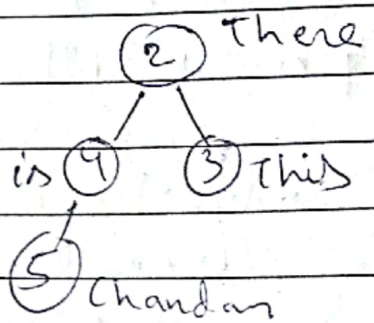
(2) (4) (3) (5)

Values :- ['there', 'is', 'This', 'Chandan']

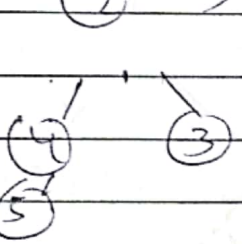
Note :- This is the First Dequeue.

→ Calling for 2nd Dequeue :

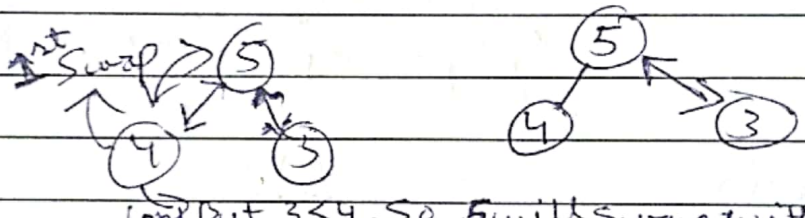
Given :



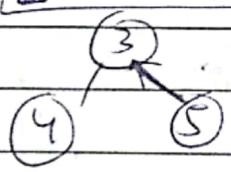
(i)



(ii)



(iii)



Result :

Min : There (2)

Value : [This, is, Chandan]
(3) (4) (5)

Similarly, for Calling 3rd, 4th & 5th Dequeue.

Min : This (3), is (4) & Chandan (5)

Code Implementation :-

```
class Node {
    constructor(val, priority) {
        this.value = val;
        this.priority = priority;
    }
}

class PriorityQueue {
    constructor() {
        this.values = [];
    }

    enqueue(val, priority) {
        let newNode = new Node(val, priority);
        this.values.push(newNode);
        this.bubbleUp();
    }
    bubbleUp() {
        let idx = this.values.length - 1;
        while (idx > 0) {
            let element = this.values[idx];
            let parentIdx = Math.floor((idx - 1) / 2);
            let parent = this.values[parentIdx];

            if (element.priority >= parent.priority) break;

            this.values[parentIdx] = element;
            this.values[idx] = parent;
            idx = parentIdx;
        }
    }
}
```

```
dequeue(){
  const min = this.values[0];
  let end = this.values.pop();

  if (this.values.length > 0) {
      this.values[0] = end;
      this.sinkDown();
  }

  if(min).
    return 'Highest Priority Value: ${min.value} (${min.priority})';
  else
    return 'Queue is Empty';
}

sinkDown() {
  let element = this.values[0];
      idx = 0;

  while (true) {

    let leftIdx = (2*idx + 1),
        rightIdx = (2*idx + 2),
        leftChild = this.values[leftIdx],
        rightChild = this.values[rightIdx],
        swapIdx = null;

    if (leftIdx < this.values.length)
      if(element.priority > leftChild.priority) swapIdx = leftIdx;
```

```
if (rightIdx < this.values.length)
  if ((swap === null && element.priority < rightChild.priority)
    || (swap !== null && rightChild.priority < leftChild.
      priority))
      swapIdx = rightIdx;

  if (swap === null) break;

  this.values[idx] = this.values[swapIdx];
  this.values[swapIdx] = element;
      idx = swapIdx;
    }
  }
}

const priorityQueue = new PriorityQueue();
// For Input
priorityQueue.enqueue('This', 3);
priorityQueue.enqueue('Chandan', 5);
  ,,          ,,        ('There', 2);
  ,,          ,,        ('Hi', 1);
  ,,          ,,        ('is', 4);

// For Output
priorityQueue.dequeue();
```

Output :-

'Highest Priority Value: Hi(1)'