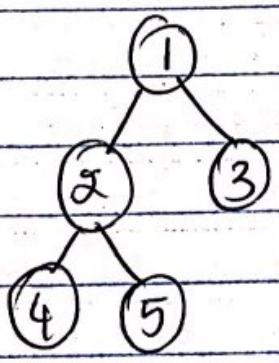
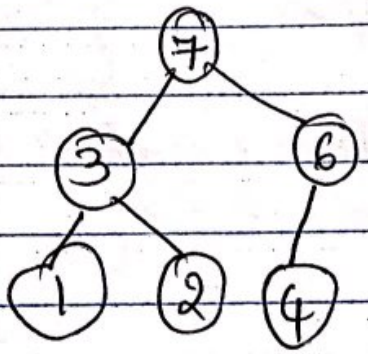
Heaps

- It is a tree with some special properties.
- Basic requirements of a heap is
  - ① value of a node must be ≥ (or ≤) to the values of its children (applies to all nodes)
    - this is also called heap property.
  - ② All leaves should be at h or h-1 levels
    - i.e., complete binary trees.
    - *heap should form a Complete BT

Valid heaps

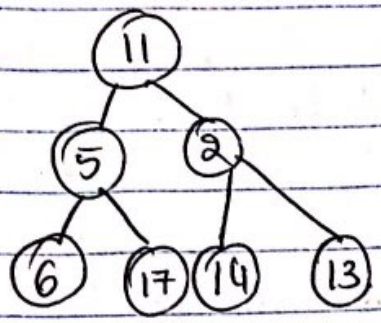
① 

1 < 2, 3.
2 < 4, 5.
leaves at h & h-1.

② 

7 > 3, 6
3 > 1, 2.
6 > 4
leaves at h & h-1

Invalid heap



diff operations (should be same).

11 > 5, 2 .
5 < 6, 17 .
2 < 14, 13

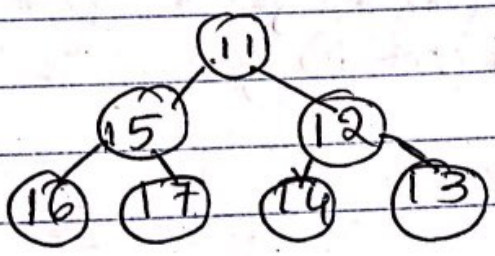
All parent nodes must follow the same property.

∴ Not a valid heap.

Types of heaps.

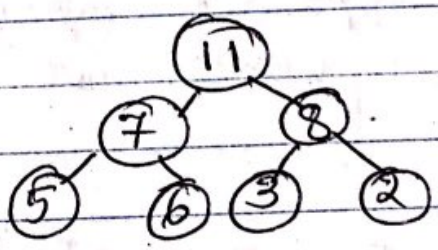
1) Min heap.

Value of a node must be ≤ to values of child ren



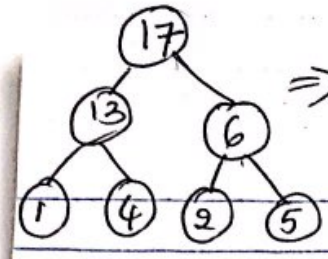
2) Max heap.

parent node ≥ child ren



Binary Heaps -

- Each node can have up to 2 children.
- Implemented using arrays [unlike BT's where BT's are impld using list [stacks]].
- Heaps are complete BT's, so there won't be any wastage of locations.



⇒ In a heap, this is represented as

0	1	2	3	4	5	6
17	13	6	1	4	2	5

Declaration of Heap -

```
Public class Heap {
  public int[] arr;
  -"- int Count;
  -"- int Capacity;
  -"- int heap_type
  public Heap (int Capacity, int heap_type);
  public Parent (int Capacity, int heap_type);
  public int LeftChild (int i);
  public int RightChild (int i);
  public int GetMax (int i);
  public int getMin (int i);
}
```

Parent of a Node.

For Node at i^{th} location,
its parent is at $\left(\frac{i-1}{2}\right)^{th}$ location

Node at i = parent at $\left(\frac{i-1}{2}\right)$ location

In the above eg, Node 6 is at location 2,
∴ its parent (17) is at $\frac{2-1}{2} = \frac{1}{2} \approx 0$
0^{th} location

Main operations : 1) getMin() / getMax()
2) extractMin() / extractMax()
3) insert() delete

Children of a Node :

For parent at i^{th} location,
* Left child will be at $(2i+1)^{th}$ location
* Right child will be at $(2i+2)^{th}$ location

In the above eg:,
Node 6 at location 2,
→ Left child (2) is at $2 \times 2 + 1 = 5$
→ Right child (5) is at $2 \times 2 + 2 = 6$

Getting the Max element :

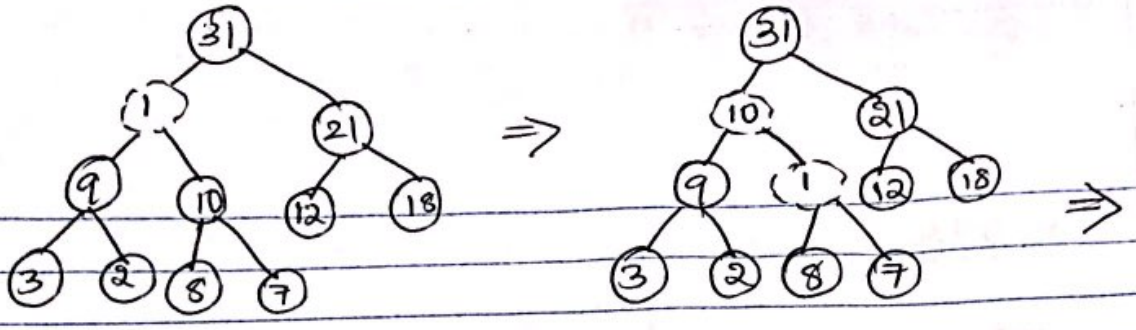
* Max element in max heap is always at root, i.e, at array[0].

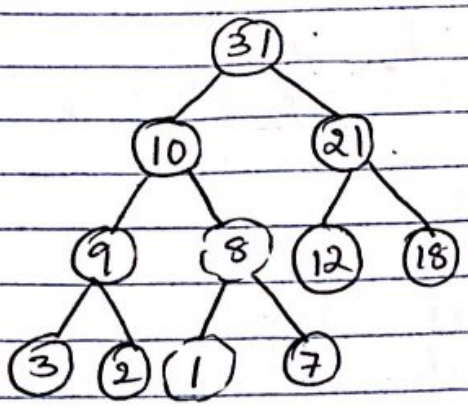
Heapifying an Element :

- The process of adjusting the location of the inserted element into heap & make it a heap again is called heapifying.

In max heap,

To heapify an element,
① find the maximum of its children & swap it with current element
② Continue until heap property is satisfied at every node in the heap.





In heapifying process,

1) Moving from top to bottom is called Percolate down.

2) Moving from bottom to top is called Percolate ~~top~~ up

TC of heapifying is O(log n)

SC O(1)

Deleting an element

1) Copy first element into some variable
2) Copy last element into first element location
3) Percolate down [top to bottom heapify] the first element.

TC => O(log n)

~~1) takes~~

1st point above takes O(1) TC.

2nd point —11— O(1) TC

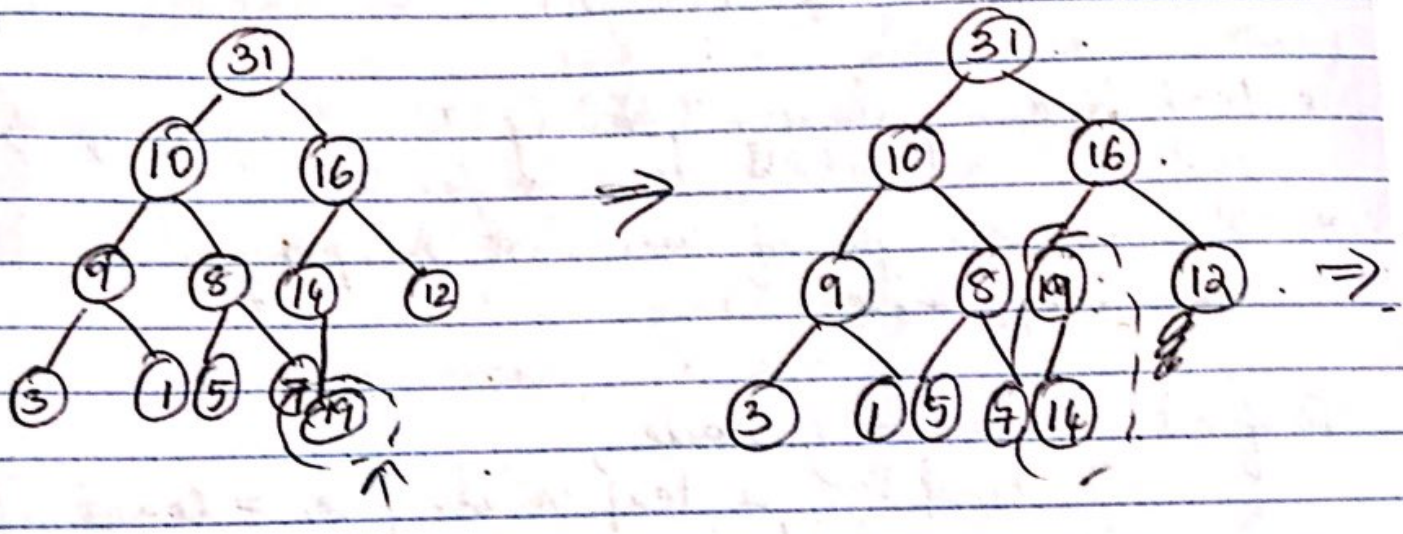
3rd point —11— O(log n) TC.

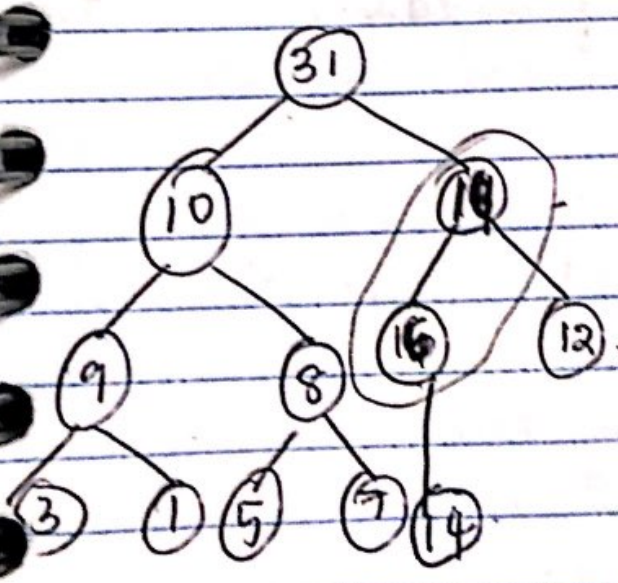
for heapifying

Deleting an element uses Percolate down
Inserting an element uses percolate up.

Inserting an element

1) Increase heap size.
2) Keep new element at the end of heap.
3) Heapify the element from bottom to top (Percolate up).





Insertion takes

(a) $O(1) + O(1) + O(\log n)$

$= O(\log n)$.

Destroying heap:

Set, count = 0 & array = null.

Heapifying the array

1) Take n input items & place them into an empty heap.
2) Takes O(nlogn) time in worst case, ∵ n inserts at O(logn) time each.

- Leaf nodes always satisfy the heap property and do not need to care for them.
- It is enough if we just heapify the non-leaf nodes.

To find non-leaf node,
=> Find the a leaf node [h = count - 1]
=> parent of that leaf node.
$= \frac{h-1}{2}$

TC:

HeapSort

- HeapSort algo, inserts all elements (from an unsorted array) into a heap, then removes them from the root of that heap until the heap is empty.
- Heap Sort can be done in place with the array to be sorted.
- Alternative - Instead of deleting an element,
  - * exchange the first element (max) with last element & reduce heap size (array size)
  - * Then we heapify 1st element
  - * Continue until no. of remaining elements is one.

TC : O(logn)

As we remove elements from the heap, values become sorted

* Why heaps are implemented using arrays & not trees? [like BT using stacks]
  1) Lower mem usage (no need to store 3 pointers)
  2) Easier mem management (just one object allocated, rather than N)
  3) Better locality of reference (items in heap are relatively closer together in mem)

~~But for~~ Using arrays is a good choice if we know that the data is constant.

- But in reality, not all BT's are completely full & perfectly balanced.

→ If array-bound BT is mostly empty, array space is wasted

→ If the tree needs to grow deeper, then we ~~to~~ this array has to be copied to a larger array, this is time expensive.