6. One-to-One : $\log_b a = \log_b c$ iff $a = c$

9. Change of Base : $\log_b a = \frac{\log_c a}{\log_c b} = \frac{\log a}{\log b} = \frac{\ln a}{\ln b}$

Other Logarithmic Definitions :-

Common Logarithms :-

① Logarithms with a based '10' are called "common logarithms". It is customary to write $\log_{10} x$ as "$\log x$".

Natural Logarithms :-

Logarithms with the base of 'e' are called "natural" logarithms. It is customary to write $\log_e x$ as "$\ln x$"

Logarithms

Def:- If $x > 0$ and 'b' is a constant ($b \neq 1$), then $y = \log_b x$ if and only if "$b^y = x$".

In the equation $y = \log_b x$, 'y' is referred to as the logarithm, 'b' is the base, and 'x' is the argument.

→ $y = \log_b x$ — logarithmic form $\quad b^y = x$

$b^y = x$ — exponential form of $y = \log_b x$

Properties of logarithms:-

If b, a and c are positive real numbers, $b \neq 1$ and 'n' is a real number, then:

1. Product : $\log_b (a \cdot c) = \log_b a + \log_b c$

2. Quotient : $\log_b \frac{a}{c} = \log_b a - \log_b c$

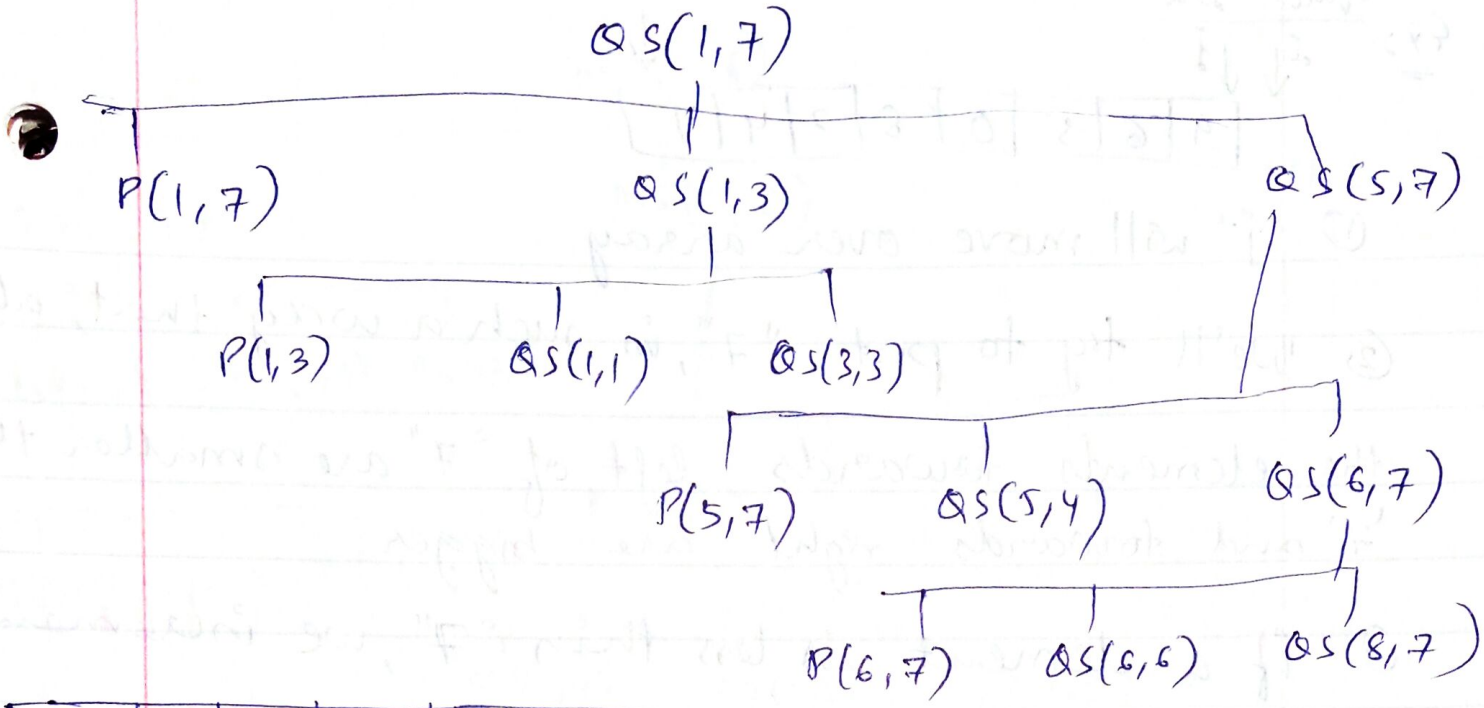
3. Power : $\log_b a^n = n \log_b a$

4. $\log_b 1 = 0$

5. $\log_b b = 1$

6. Inverse 1 : $\log_b b^n = n$

7. Inverse 2 : $b^{\log_b n} = n ; \quad n > 0$



5	7	6	1	3	2	4

Interesting Inputs for Quick-Sort:-

① Ascending order input.

(1 2 3 4 5 6)

$T(n) = O(n) + T(n-1)$

$= O(n^2)$

② Descending Order

(6 5 4 3 2 1) ⇒ (1 5 4 3 2 6)

$T(n) = O(n) + T(n-1) = O(n^2)$

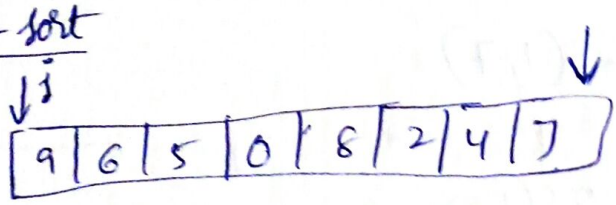
③ Entire array has same elements

(2 2 2 2 2 2) ⇒ $T(n) = O(n^2)$

④

quick-sort

Ex:



① 'j' will move over array

② We'll try to put "7", in such a way that, all the elements towards left of '7' are smaller than '7' and towards right are bigger

③ if a element is less than "7", we increment 'i' and swap, a[i] with a[j]

④ In the algorithm, ~~till~~ from 0^{th} index to i^{th} element elements will be less than '7' & i^{th} to j^{th} element, greater than '7', and after 'j', yet to be examined

→ QUICKSORT(A, p, r)

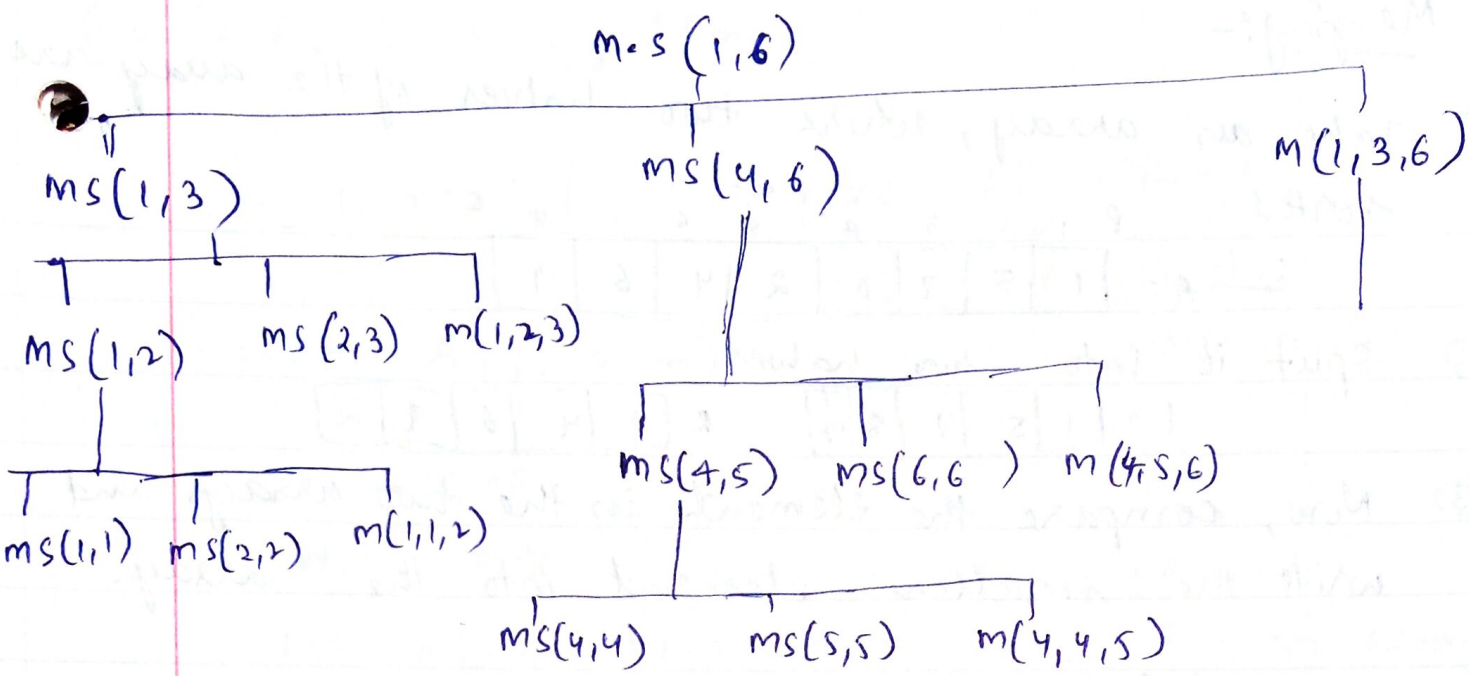
```
{ if (p<r)
  { q = PARTITION(A, p, r)
    QUICKSORT(A, p, q-1)
    QUICKSORT(A, q+1, r)
  }
}
```

	best case, partition at n/2	worst case
	T(n)	T(n)
	O(n)	O(n)
	T(n/2)	T(0)
	T(n/2)	T(n-1)
	$T(n) = 2*T(n/2) + O(n)$	$T(n) = T(n-1) + O(n)$
	$= O(n \log n)$	$= T(n-1) + cn$

$T(n) = T(n-1) + cn$

$= T(n-2) + c(n-1) + cn$

$= T(n-3) + c(n-2) + c(n-1) + cn \quad = O(n^2)$



QUICK - SORT ALGORITHM :-

$\Omega(n \log n)$

$- T(n) = O(n) + 2T(n/2)$

$O(n^2)$

```
PARTITION (A, p, r)
{
    x = A[r]
    i = p - 1
    for (j = p to r - 1)
    {
        if (A[j] ≤ x)
        {
            i = i + 1
            exchange A[i] with A[j]
        }
    }
    exchange A[i+1] with A[r]
    return i+1
}
```

Merging:-

① Take an array, where two halves of the array are sorted

Ex:- A =

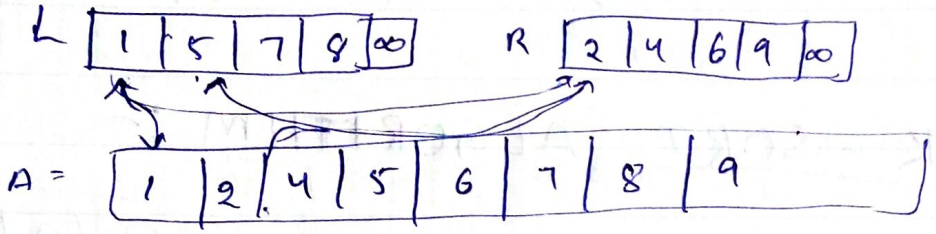
p	1	2	3	q	q+1	6	7	8	r
	1	5	7	8	2	4	6	9	

② Split it into two halves

L: | 1 | 5 | 7 | 8 | ∞ |

R | 2 | 4 | 6 | 9 | ∞ |

③ Now, compare the elements in the two arrays and write the smaller element into the array.



→ Recursive function :-

```
merge_sort (A, p, r)                    — T(n)
{  if (p < r)
       q = ⌊(p+r)/2⌋
       merge_sort (A, p, q)             — T(n/2)
       merge_sort (A, q+1, r)           — T(n/2)
       merge (A, p, q, r)               — O(n)
}
```

$$T(n) = 2 * T(n/2) + O(n)$$

$$= \Theta(n \log n)$$

Divide & Conquer Algorithms:

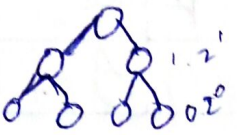
Merge Sort:- (out of place Algorithm)

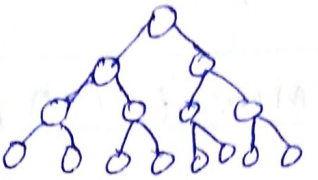
(we don't sort array in the same array)

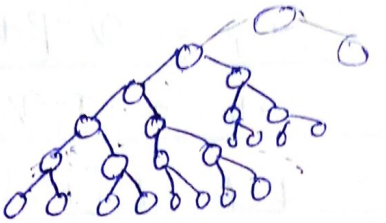
```
MERGE (A, p, q, r)
{
    n1 = q - p + 1
    n2 = r - q
    let L[1 ... n1+1] and R[1 to n2 + 1] be new arrays
    for (i = 1 to n1)
        L[i] = A[p + i - 1]
    for (j = 1 to n2)
        R[j] = A[q + j]
    L[n1 + 1] = ∞
    R[n2 + 1] = ∞
    i = 1, j = 1

    for (k = p to r)
        if (L(i) ≤ R[j])
            A[k] = L[i]
            i = i + 1
        else A[k] = R[j]
            j = j + 1
```

$S(0) = 0$ 0

$S(1) = 1$ 

$S(2) = 1+1 = 2$ 

$S(3) = (1+2+1) + (1+2+1) = 8$ 

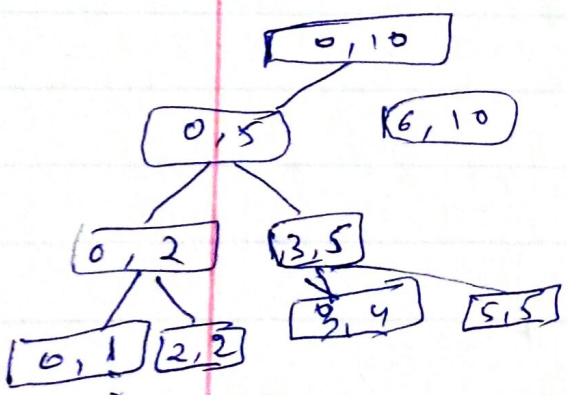
$S(4) = ((1+2+1) + 3 + (1+2+1)) + ((1+2+1) + 3 + (1+2+1))$ 

$= 22$

$S(5) = ((1+2+1) + (3) + (1+2+1) + 4 + (3) + (1+2+1) + (1+2+1)) + ((1+2+1) + (3) + (1+2+1) + 4 + (3) + (1+2+1) + (1+2+1))$

$= 52$

$S(6) = 114$



omsairam omsrivenkateswarayanamaha ommahaganadhipatayenamaha omnamaha shivaya

HEAPS

→ Time complexities of operations on different types of arrays

	insert	search	find min	Delete min
unsorted Array	O(1)	O(n)	O(n)	O(n)
sorted Array	O(n)	O(logn)	O(1)	O(n)
linked list	O(1)	O(n)	O(n)	O(n+1) = O(n)

If some problem needs, insert, find-min and delete-min to be optimal, then we have

HEAP - Data structure

	insert	find-min	delete min
min Heap	O(logn)	O(1)	O(log n)

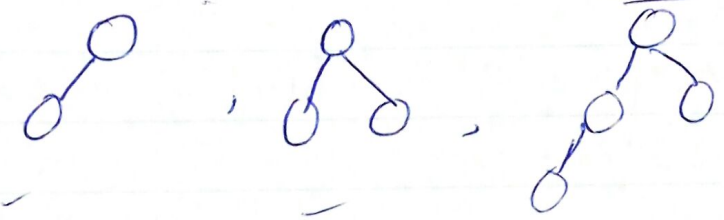
→ "Heap" is a "binary tree" or 3-ary (or) n-ary tree

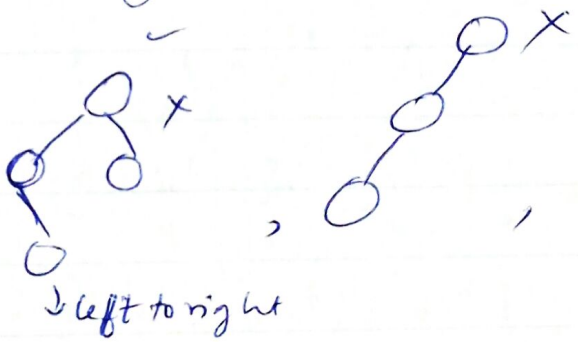
→ Binary Tree: Every node can have atmost '2' children

→ Every Heap is a "almost complete binary tree"

→ Complete B-T : All nodes are filled in the last level

→ Almost-Complete B.T: leaves should be in either last (or) last but one "level & filled from left to right





MAX HEAP: "ROOT", should be greater than left and right sub-tree nodes

MIN-HEAP: "ROOT", should be less than left and right sub-tree nodes

HEAP IMPLEMENTATION:

→ Heap implementation is done using the "Array"

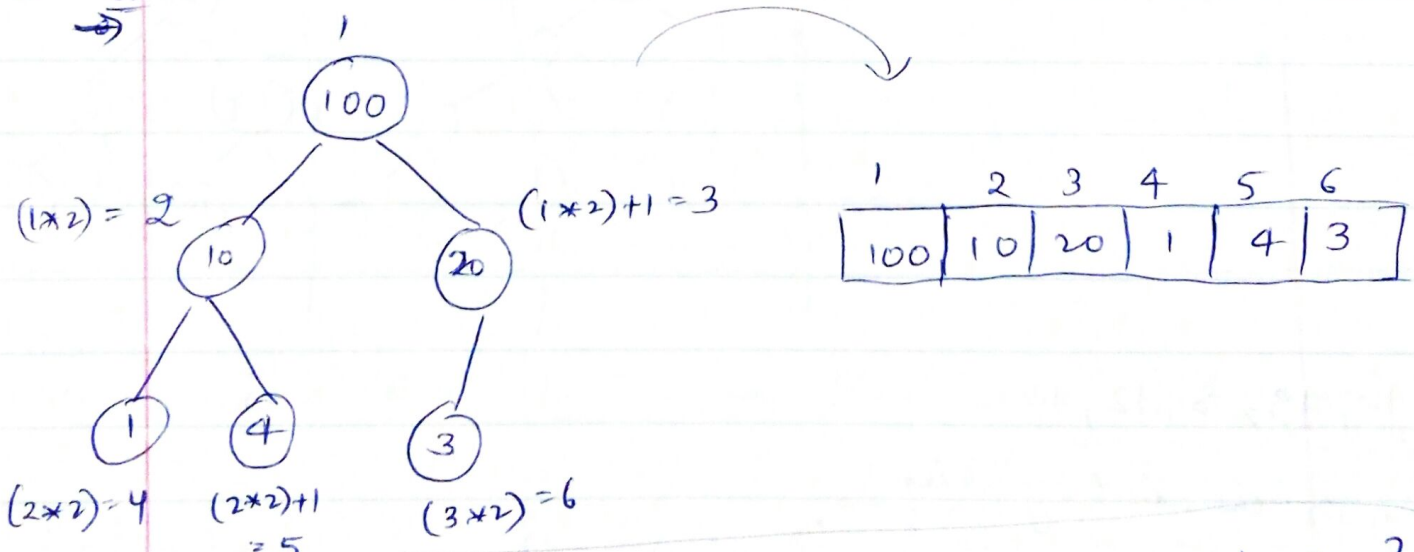
Points

① "Root" will be placed at 1st element

② "left child" = (index of root) * 2

③ "Right child" = ((index of root) * 2 + 1)

Ex:



Why heap Complete or Almost Complete B.T.?

*) If you have Complete (or) Almost Complete array will be completely filled (or) there space (or) gaps is left between elements.

Finding elements :-

(1) Left Child (i) = $2 * i$ ⇒ multiplying with '2' → left shift

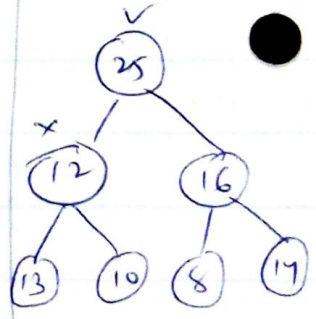
(i → index)

(2) Right Child (i) = $2 * i + 1$

(3) Parent (i) = $\lfloor i/2 \rfloor$ ⇒ right shift ⇒ dividing by '2'

Ex:

	array length A.length	MAX-HEAP no. of elements following heap property A.heapsize
25, 12, 16, 13, 10, 8, 14	7	1



14, 13, 8, 12, 10	5	5

array in descending order will be a max heap

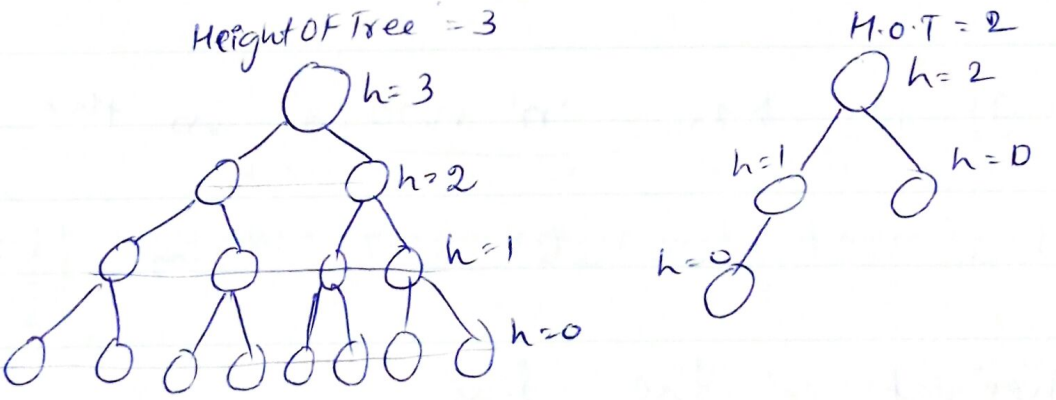
Array in ascending order ⇒ Min Heap

→ If 'you' have to construct a heap, no need to sort, which takes $O(n \log n)$ time, Instead, we can construct a heap in $O(n)$ time complexity.

Complete Binary Tree Properties :-

⊙ **Height of a Node** :- The longest possible no. of edges from that node to any leaf

Ex :-



⊙ **Height of a Tree** :- = Height of the root

⊙ Given, height of the complete Binary tree, what are maximum of Nodes in the Tree ?

max. no. of nodes : $(2^0 + 2^1 + 2^2 + \dots 2^h)$

$$= \frac{1(2^{h+1} - 1)}{2 - 1} \quad \therefore \; 2^{h+1} - 1$$

⊙ max. no of nodes of 3-ary tree = $3^0 + 3^1 + 3^2 + \dots$

$$= \frac{1\left(3^{h+1}-1\right)}{3-1}$$

$$= \frac{3^{h+1}-1}{2}$$

⊙ max. no of nodes for n-ary tree =

$$n^0 + n^1 + n^2 + n^3 + \dots n^h$$

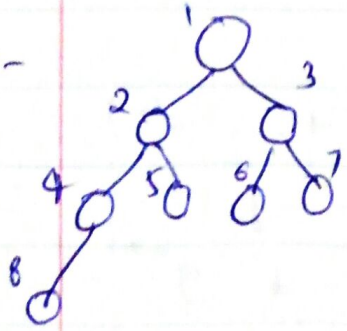
$$= \frac{1\left(n^{h+1}-1\right)}{n-1} = \frac{n^{h+1}-1}{n-1}$$

⇒ If we have 'n' nodes in the complete B.T (or) almost complete B.T, then $\lfloor \log n \rfloor$ is the height of the tree.

⊙ In a complete B.T (or) almost complete B.T, range of leaves nodes ~~start from index~~ is

$$\left[\lfloor n/2 \rfloor + 1 \text{ to } n\right]$$

Ex:-



leaves = [5, 6, 7, 8]

nodes from 5 to 8

MAX-HEAPIFY:

```
MAX-HEAPIFY(A, i):
{
    l = 2i ;
    r = 2i+1 ;
    if (l ≤ A.heapsize and A[l] > A[i])
        largest = l ;
    else largest = i ;

    if (r ≤ heap-size and A[r] > A[largest])
        largest = r ;
    if (largest ≠ i)
        exchange A[i] with A[largest]
        MAX-HEAPIFY (A, largest)
}
```

- One way to create a ~~Ma~~ Max-Heap is to sort elements in the descending order. But it takes $O(n \log n)$ time.

① Every leaf is a heap, as it is the maximum & it is the minimum and everything.

② So, Building blocks of heaps are leaf nodes,

Ex:- Consider arr = [1, 5, 6, 8, 12, 14, 16]

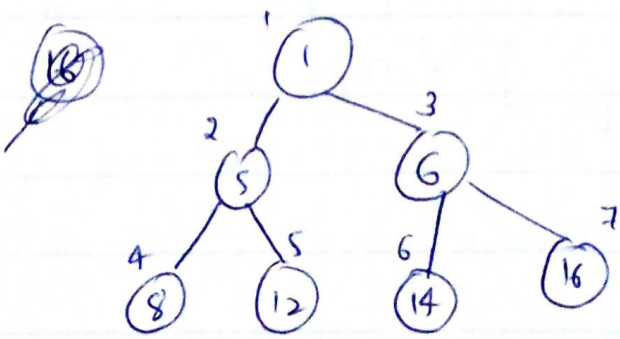
1. while constructing a heap, we consider that leaf nodes are heaps, as ~~it is the m~~

2. We know that leaf nodes start from the indices

$\left\lfloor \frac{n}{2} \right\rfloor + 1$

3. So, we will start heapifying from the largest non-leaf node index and moves towards root.

If you consider above example, C.B.T is



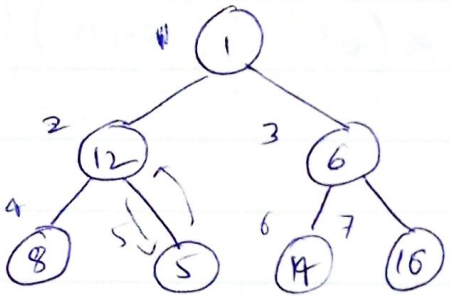
① leaf nodes start from, index $\lfloor \frac{7}{2} \rfloor + 1 = 4$.

② So, largest non-leaf node = 3

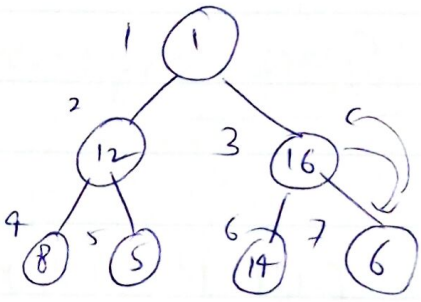
③ Call the max_heapify on the node with index = 3.

=>

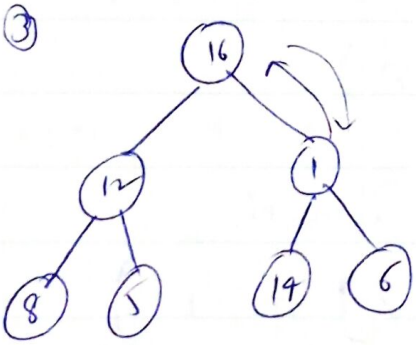
(1)



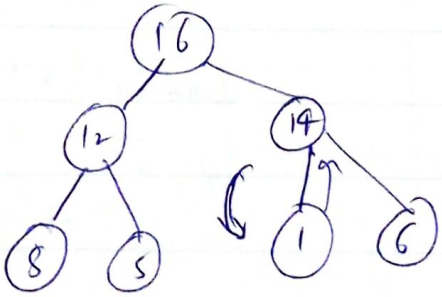
(2)



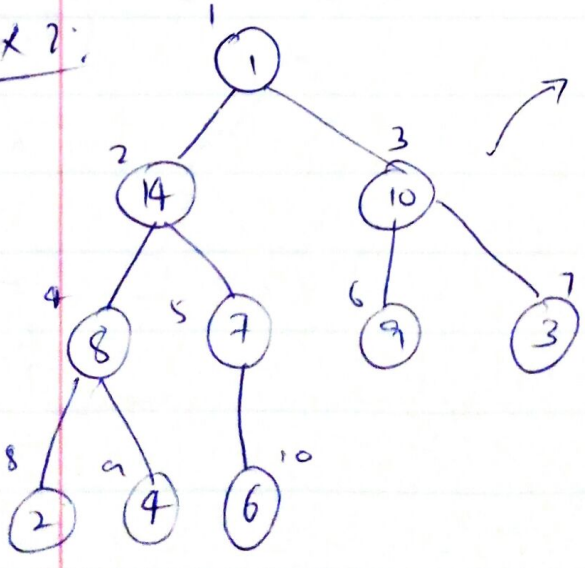
(3)



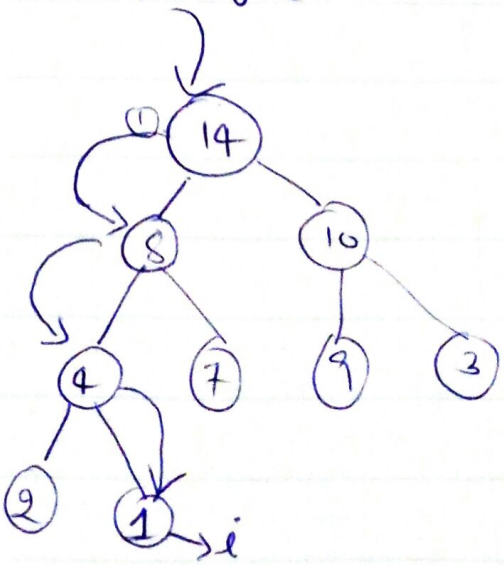
(4)



Ex 2:



=> max_heapify(1)



Time Complexity:

1. In order to do one swapping, we need to do "2" comparisons at each level and at max "log n" levels. So

$$T(n) := 2 * \log n = O(\log n)$$

Space complexity : O(log n).

→ BUILD_MAX_HEAP

```
Build-Max-Heap (A)
{
    A.heapsize = A.length
    for (i = ⌊A.length/2⌋ to 1)
        MAX-HEAPIFY(A, i)
}
```

Time Complexity Analysis :-

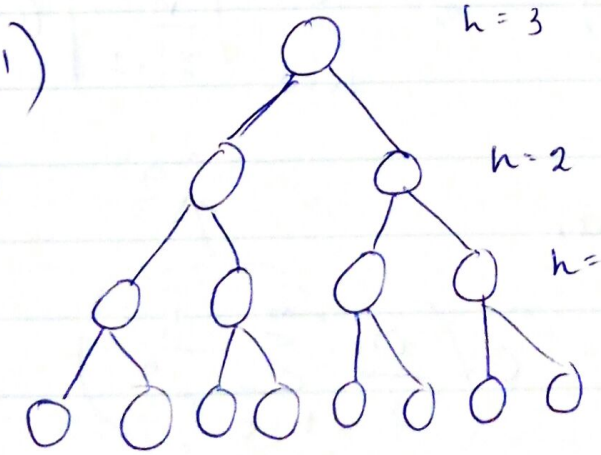
⊛ We might think, if we call heapify on a node, it takes $O(\log n)$ time and as we have 'n' nodes, time complexity would be $O(n \log n)$.

BUT, we can prove that, time complexity to build a Heap is $O(n)$.

Ex :- In a complete B-T, the no. of nodes of a tree at height 'h' = $\left\lceil \frac{n}{2^{h+1}} \right\rceil$

Ex :-

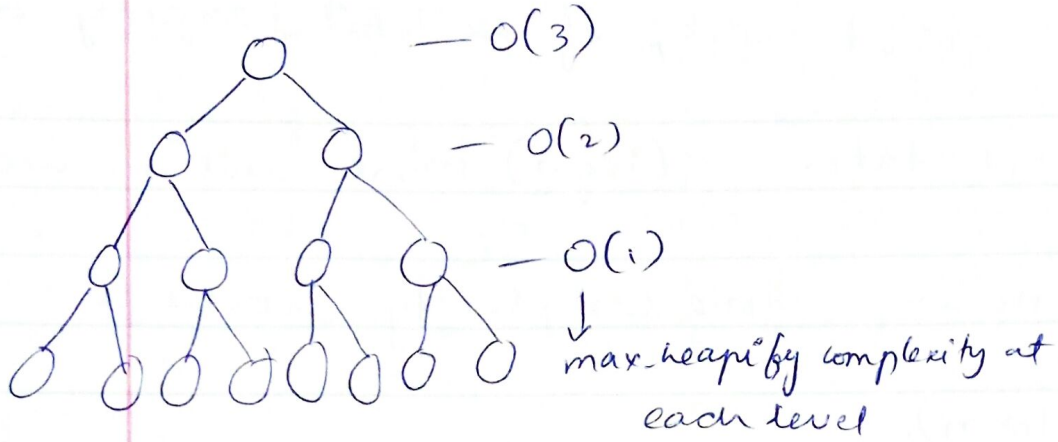
(3, 1)
(2, 2)
(1, 4)
(0, 8)
↓ height ↓ no. of nodes



$h = 0 ; \left\lceil \frac{15}{2^{0+1}} \right\rceil = 8$

$h = 1 ; \left\lceil \frac{15}{2^2} \right\rceil = 4$

So, if we apply max_heapify at each level



→ Work done, at height 'h', to apply max_heapify

$$= \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$$

→ In a tree, height can range from 0, log n

$$= \sum_{h=0}^{\log n} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) \Rightarrow \sum_{h=0}^{\log n} \left\lceil \frac{n}{2^{h}\cdot 2} \right\rceil (c \cdot h)$$

Pull constants out,

$$= \frac{cn}{2} \sum_{h=0}^{\log n} \frac{h}{2^h} \leq O\left(\frac{cn}{2} \sum_{h=0}^{\infty} \frac{h}{2^h} \right)$$

"2" =

$$\Downarrow$$

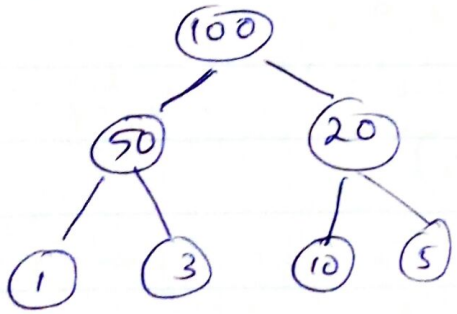
$$O(n).$$

Given a max-heap, delete maximum from it, ?

To delete maximum element from heap,

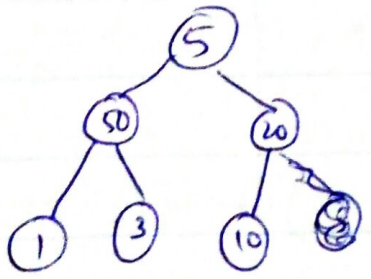
- Take the root node i.e. maximum
- Replace the maximum i.e root node with last element of the heap
- Call the max_heapify function on the root node (It'll again make the entire tree to heap)

Ex:- 100, 50, 20, 1, 3, 10, 5

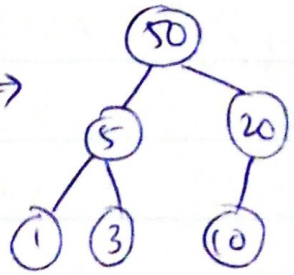


Step 1:-

Take '100' out and replace with '5'



⇒



100

Step 2: If you continue this for 'n' times, you get the elements in the descending order

Pseudo Code:-

```
Heap_Extract_Max (A)
{
    if A.heap_size < 1
        error "heap underflow"
    max = A[1]
    A[i] = A[A.heap_size]
    A.heap_size = A.heap_size - 1
    MAX_HEAPIFY (A, 1)
    return max
}
```

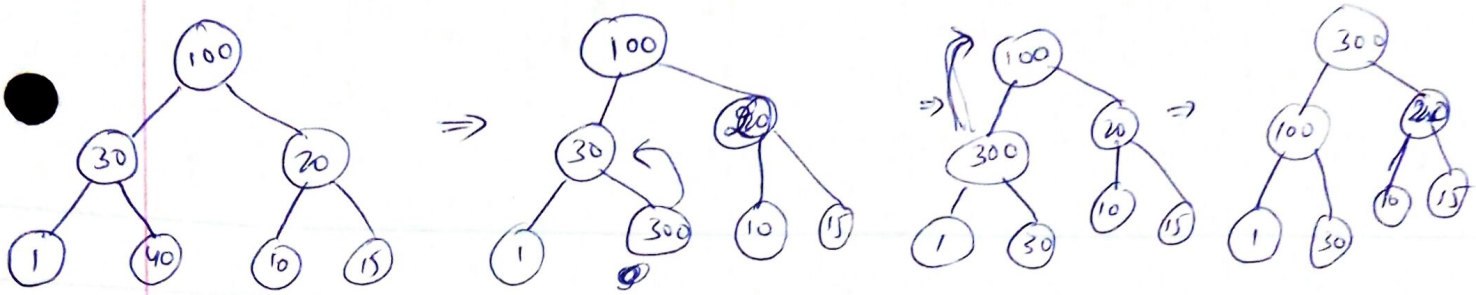
→ Time Complexity: $O(\log n)$

(x) Increase Key Value in a Heap:

```
Heap_Increase_Key (A, i, key):
{
    if (key < A[i])
        error
    A[i] = key
    while (i > 1 and A[i/2] < A[i])
        exchange A(i) and A[i/2]
        i = i/2
}
```

Time Complexity $= O(\log n)$

Heap-increase-key (A, 5, 300)



→ Decreasing a Key?

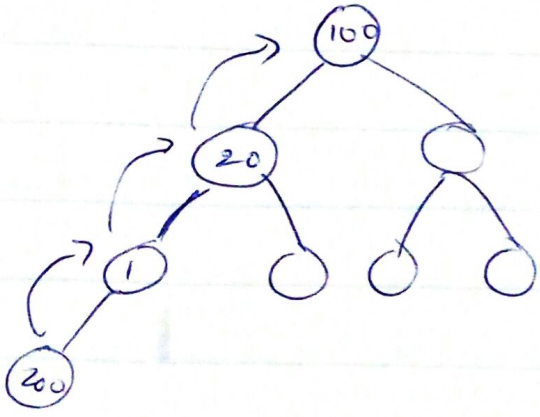
- Decrease a Key
- Call max-heapify on the Key.

$T(n) = O(\log n)$,,

→ Insertion in MAX. Heap:

- Insert element at the end
- Compare it with 'parent'; if parent > Key, swap;
- Continue till $i >= 0$.

$T(n) = O(\log n)$



	Find max	Delete max	insert	increase_key	decrease key
max-heap	O(1)	O(log n)	O(log n)	O(log n)	O(log n)

	Find min	delete random element	search random element
not optimal	O(n)	O(n)	O(n)

⑥ HEAP_SORT :-

```
Heap Sort (A)
{
    BUILD_MAX_HEAP (A)
    for (i=A.length down to 2)
        exchange A[i] with A(i)
        A.heapsize = A.heapsize - 1
        MAX_HEAPIFY (A, 1)
}
```

STEPS :-

- Build HEAP
- Take maximum i.e a[0] and put it at the end of the array & Reduce the size of the array(heap size) by "1".
- Call max_heapify on "first" element

Heap sort:

Time Complexity:

- Always max-heapify is applied at root,

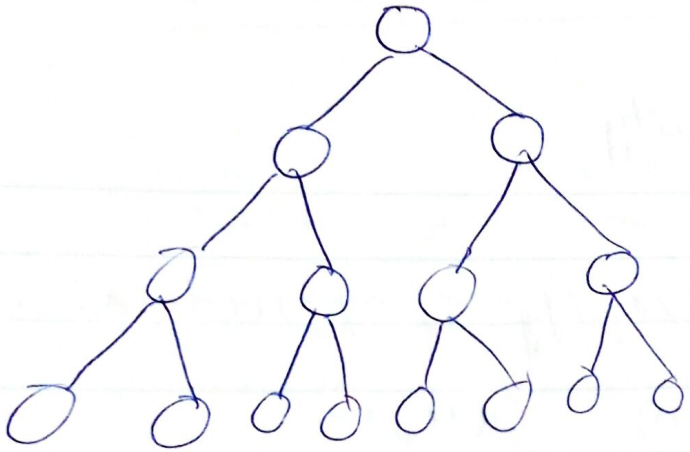
  So, $T(n) = O(\log n)$

- If 'n' elements are there then,

$$\boxed{T(n) = O(n \log n)}$$

→ we cannot argue, as we have done in "BUILD HEAP", because, As we are swapping first element with last element (mostly leaf), and running MAX-HEAPIFY on root. We will be having "$\log n$" as the height of the tree, till (or) atleast "$n/2$" times (~~leaves~~ leaves in a CBT of 'n' nodes).

⇒ So, $T(n) = O(n/2 \log n) \Rightarrow O(n \log n)$



$\}$ $n/2$ nodes

$\Downarrow$

i.e, " $n/2$ " times tree height will be "$\log n$".