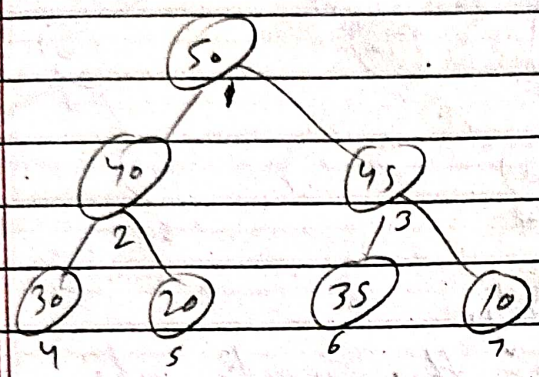
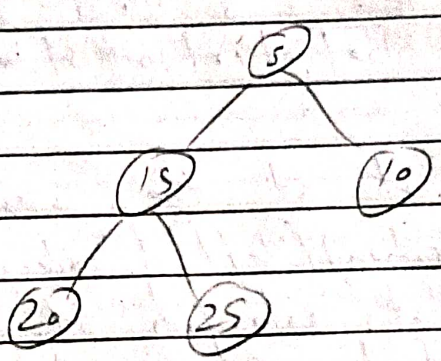
DATE : / 20
PAGE No

Heap

a) Max Heap



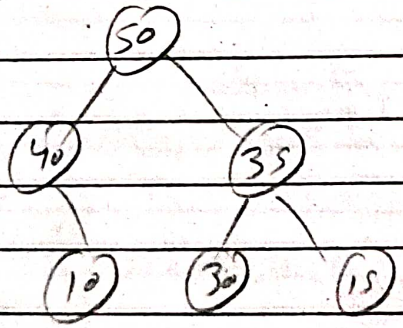
b) Min Heap



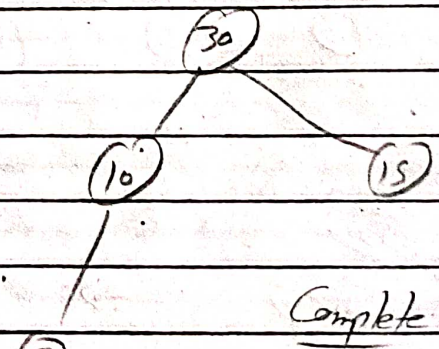
Heap is a complete Binary tree that satisfies a heap property.

↳ Don't have compulsion to the leaf node, that they must be have 2 children.

↳ Element positioning start from left.



Not Complete Binary tree



Complete Binary tree
Max heap

	0	1	2	3	4	5	6	7
A =	X	50	40	45	30	20	35	10

Node = i

Parent (i) = $\lfloor \frac{i}{2} \rfloor$

left child = 2 * i

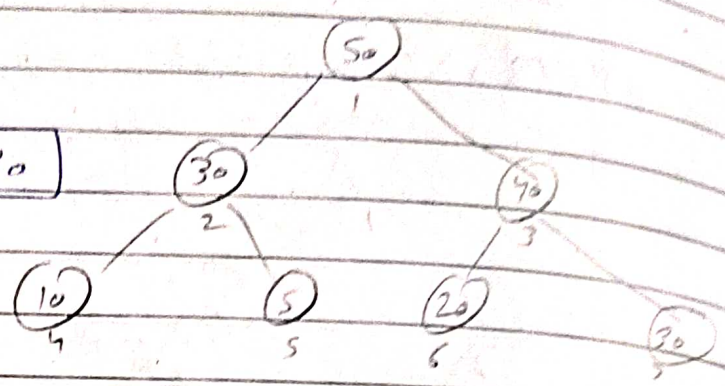
Right child = 2 * i + 1

DATE : / 20
PAGE No

- **Insert (Max Heap)**

Insert : 60

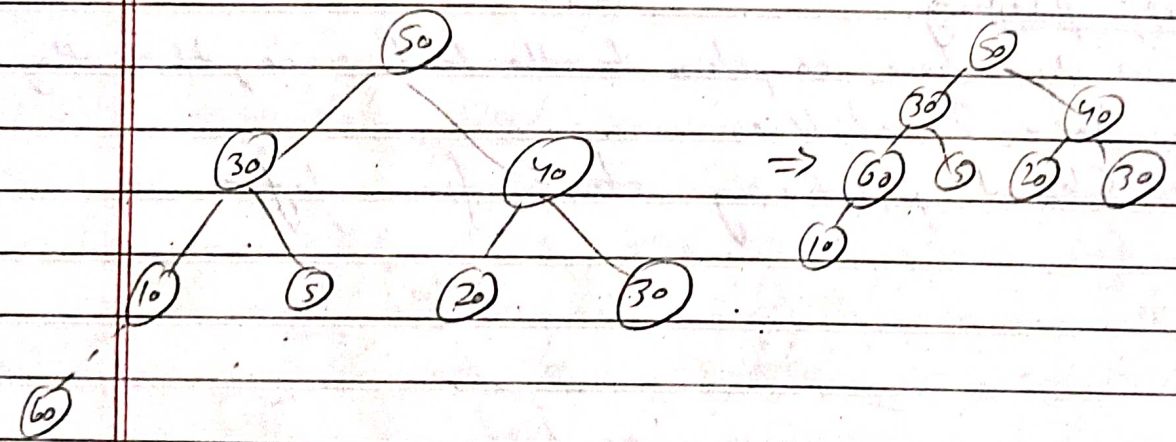
	1	2	3	4	5	6	7
A :	50	30	40	10	5	20	30

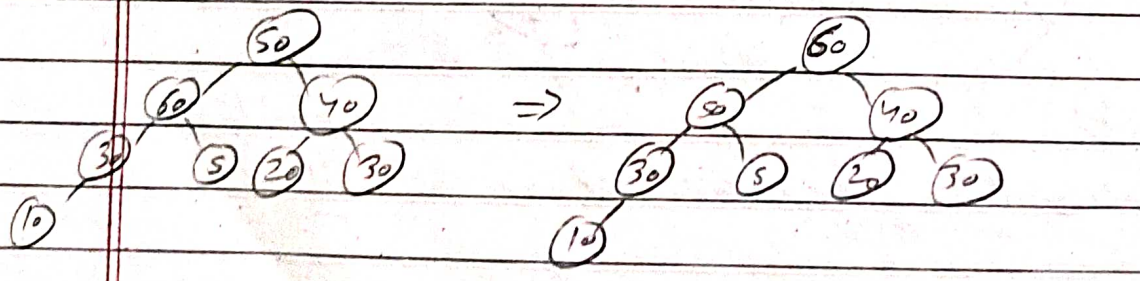


Step 1: Insert at the (insertion position) (left of node side) that satisfies Heap property.

Step 2: check if parent is smaller than insertion value; swap

Step 3: Repeat until reaches at index 1.





	1	2	3	4	5	6	7	8
A :	60	50	40	30	5	20	30	10

Time Complexity = $O(\log(n))$

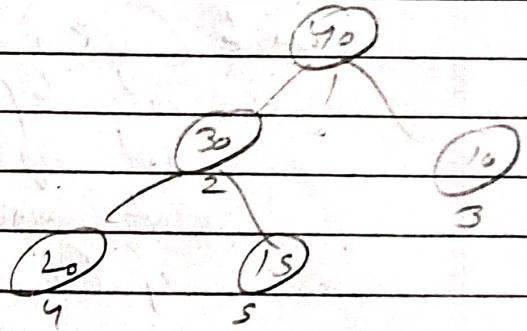
Traverse = Bottom to Top

DATE: / 20
PAGE No

```
void insert (A[], n, value) {
    n = n+1  // we are adding new element
    A[n] = value;
    int i = n;
    while (i > 1) {
        int parent = i/2;
        if (a[parent] < a[i]) {
            swap (A, parent, i);
            i = parent;
        } else {
            return;
        }
    }
}
```

- **Delete (Max Heap)**

	1	2	3	4	5
A :	40	30	10	20	15



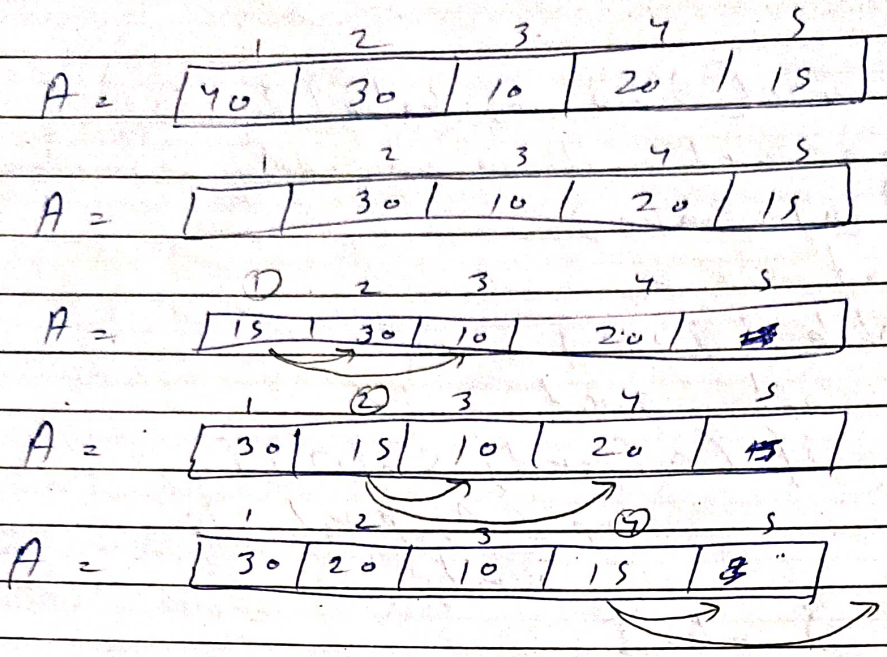
We care about topmost ele.

Step 1: Remove Root
Step 2: Remove last node and insert it into root.
Step 3: Check for childs (Take Max) if parent smaller; swap
Step 4: Repeat until n

Time Complexity = log(n)

Traverser = Top to Bottom

DATE : / 20
PAGE No




```
void delete(A[], n) {
    A[1] = A[n];
    n = n-1;
    i = 1;
    while (i < n) {
        int left = A[2*i];
        int right = A[2*i+1];
        int larger = left > right ? 2*i : 2*i+1;
        if (A[i] < A[larger]) {
            swap(A, i, larger);
            i = larger;
        }
        else {
            return;
        }
    }
}
```

Time Complexity = $O(\log(n))$

DATE : / 20
PAGE No

- Heapify

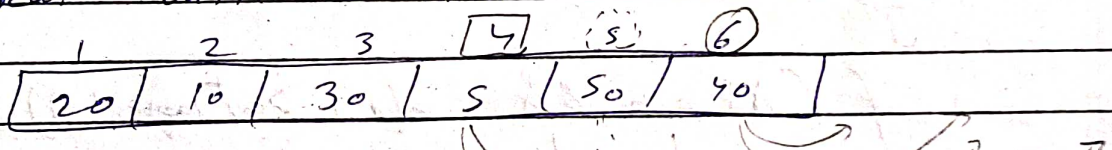
Way 1:
If we add by our method, it will take $O(n \log n)$

Way 2:
Heapify $O(n)$

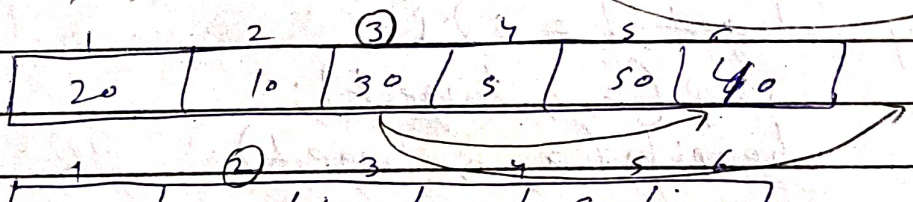
~~Theat~~ There can be many configuration of a heap.

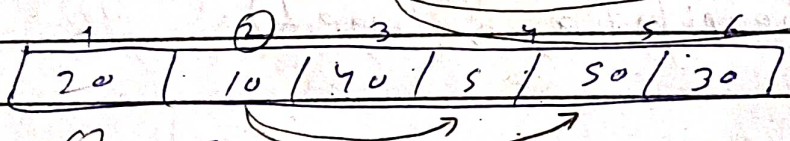
Step 1: Start from last element and chck if its child is greater; swap

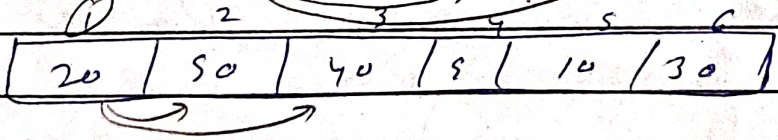
Step 2: Repeat until 1 index

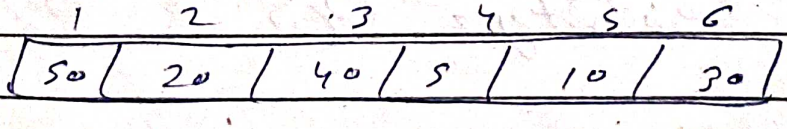


Check = 6, 5, 4.









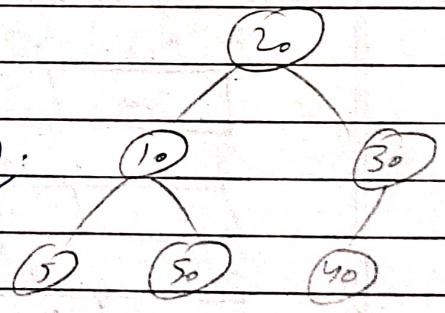
Max Heap

Improvement - We can start from $\frac{n}{2}$ instead of n. Bcz for leaf node we are not swapping at all.

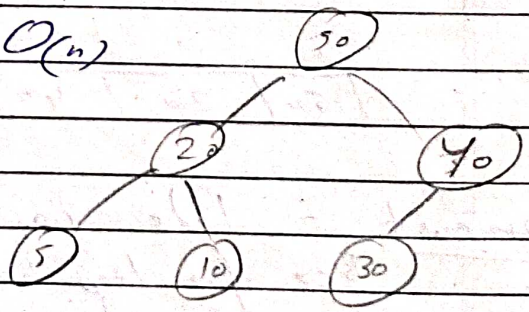
DATE : / 20
PAGE No

```
buildheap (int a[], int n) {
    for (int i = n/2 ; i>0 ; i--) {
        heapify(a, n, i);
    }
}

void heapify (int a[], int n, int i) {
    int largest = i;
    int l = 2*i;
    int r = 2*i + 1;
    if (l<=n && a[l] > a[largest]) {
        largest = l;
    }
    if (r<=n && a[r] > a[largest]) {
        largest = r;
    }
    if (largest != i) {
        swap(a, i, largest);
        heapify(a, n, largest);
    }
}
```



Time Complexity = O(n)



DATE : / 20
PAGE No

- Heap Sort

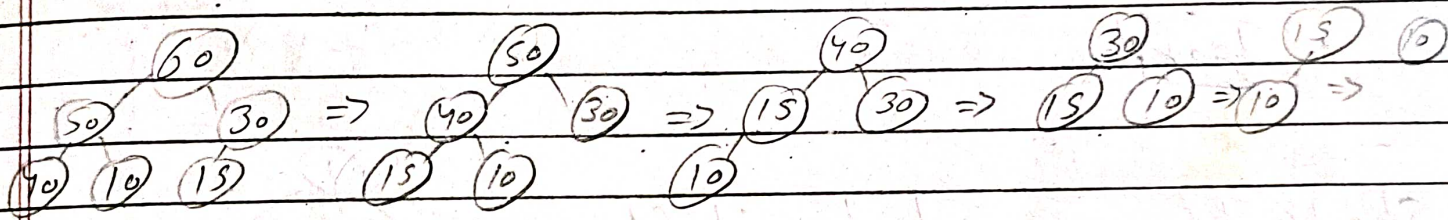
1	2	3	4	5	6
40	10	30	50	60	15

O(n) ↓ Convert into a heap

1	2	3	4	5	6
60	50	30	40	10	15

O(n log n) ↓ Delete ele one by one

1	2	3	4	5	6
10	20	30	40	50	60



n (log n)

n elements, Deletion

```
void heapSort(int a[], int n) {
O(n)    buildheap(a, n); // build heap
O(n log n)  for(int i = n; i > 1; i--) {
                print(a[i]);
                swap(a, 1, i); // swap and making one
                heapify(a, i-1, 1); // sort remaining
            }                   size sort from
}
```