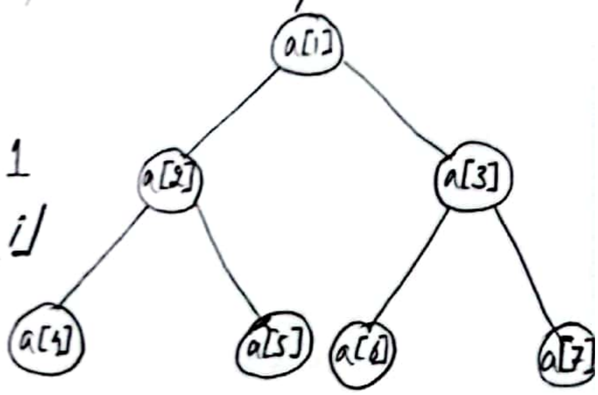
Trees stored in Arrays

As seen earlier trees can be stored with the help of pointer-like structures, where each item contains reference to its children. If the tree in question is a complete binary tree, there is a useful array based alternative.

A binary tree is <u>complete</u> if every level, except possibly the last, is completely filled, and all the leaves on the last level are placed as far left as possible.

Complete binary trees always have minimal height for their size n, i.e. $\log_2 n$, and are always perfectly balanced. Moreover they can be stored in arrays directly.



* starting index chosen 1 by design
* Nodes at level i have indices $2^i, \dots, 2^{i+1}-1$
* Level of nodes with index i is $\lfloor \log_2 i \rfloor$
* Children of a node with index i, if exists have indices $2i$, & $2i+1$
* Parent of a child with index i has index $i/2$ (using integer division)

This allows following simple algorithm.

boolean isRoot(int i) return i==1	int left(int i) return 2*1
int level(int i) return log(i)	int right(int i) return 2*i+1
int parent(int i) ...	

* which makes the processing of these trees much easier.

DSA

01-02-2023
Fall 2022 (2)

Observations

* This way of storing a binary tree as an array is not efficient if tree is not complete. i.e. reserve space in array for every possible node.
* Keeping BST balanced is a difficult problem, Moreover array based representations are inefficient due to involvement of shifting arrays for insertion & Deletion.

Priority Queues

* Every day Queues → 1^{st} come 1^{st} serve
* Special cases Priority base Queues, e.g. Hospitals
* Priority Queues $\xrightarrow{\text{Representation}}$ Complete Binary tree in array form
* Priority Queues can be efficiently implemented by binary heap tree (special type of complete binary tree)

Binary heap tree

* Node labels → search keys in BST $\xrightarrow{\text{Now}}$ represent priority
* Insert delete w/o having to keep the whole tree sorted like BST
* Because we only have to remove one element at a time, i.e. one with the highest priority, & that always lies at the root.

A

01-02-2023
Fall 2022 (3)

Definition A BHT is a complete binary tree either empty or satisfies the following:

* The priority of the root is higher than or equal to that of its children.
* The LST & RST of root are heap trees.

- Binary tree Array representation already done.

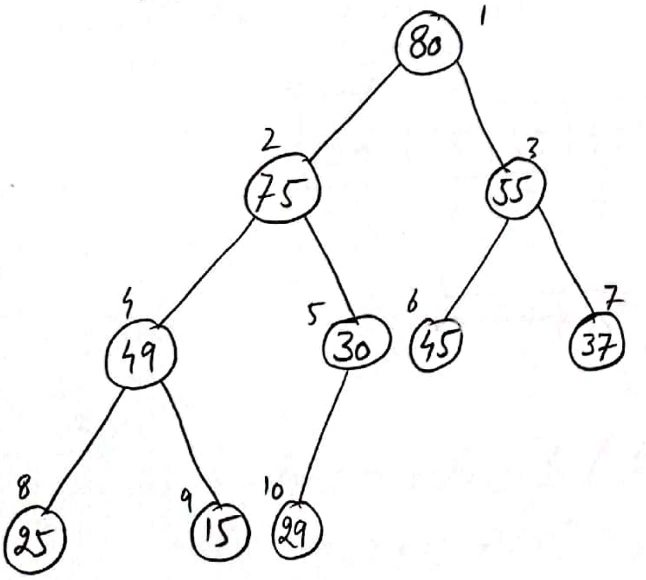
Heap

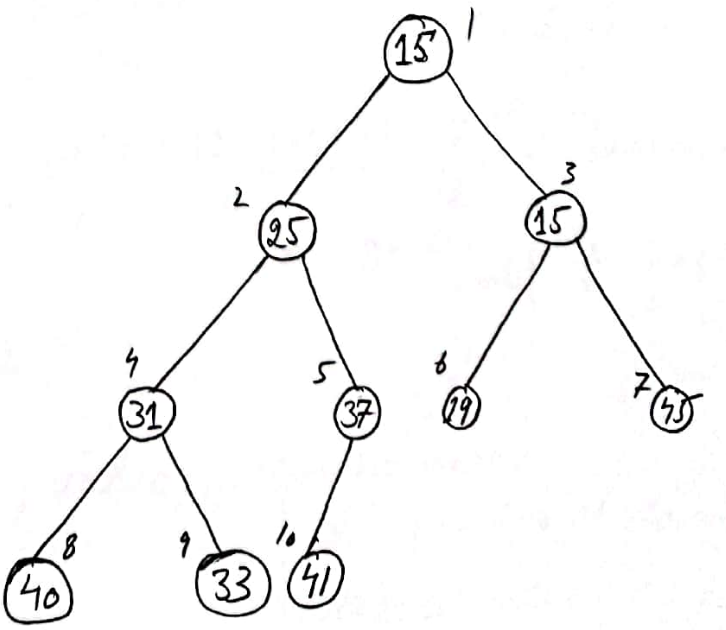
Max heap

for every node i, the value of node is less than or equal to its parent's value, except root node.

Min heap

for every node i, the value of node is greater than or equal to its parent's value.



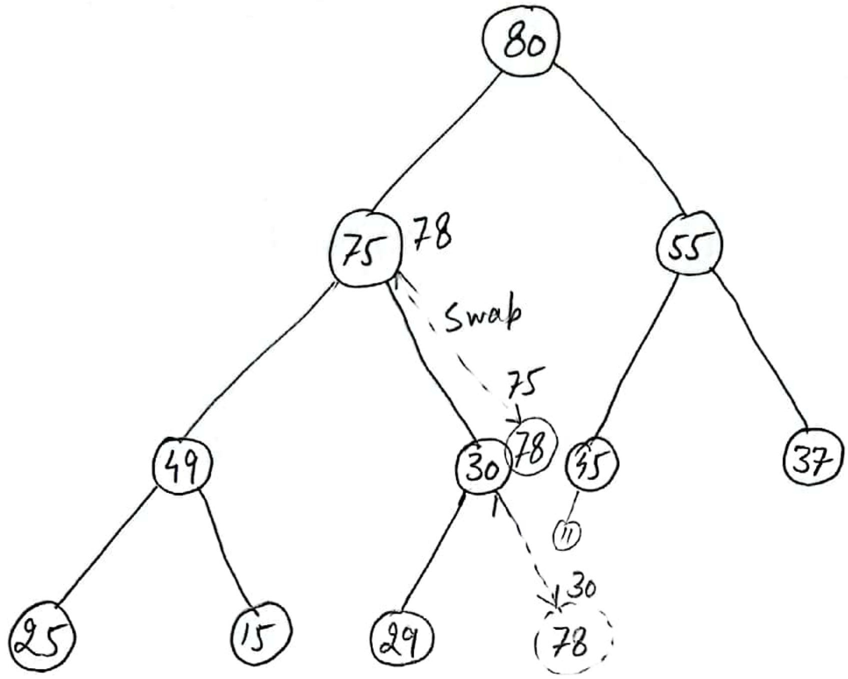


80	75	55	49	30	45	37	25	15	29
1	2	3	4	5	6	7	8	9	10

15	25	15	31	37	19	45	40	33	41
1	2	3	4	5	6	7	8	9	10

Insert in Max heap

Insert (78)
then (11)



(1) * Insert from leaf as left as left & vacant location

(2) * Check with its parent, swap if greater than parent

(3) * Repeat till less than or equal to parent or root node is reached.

No Swap 78 75 Swap No Swap

80	75	55	49	30	45	37	25	15	29	78	11
1	2	3	4	5	6	7	8	9	10	11	12

Parent $\lfloor \frac{12}{2} \rfloor = 6$

78 Swap $\lfloor \frac{5}{2} \rfloor = 2$

30 Parent $\lfloor \frac{11}{2} \rfloor = 5$

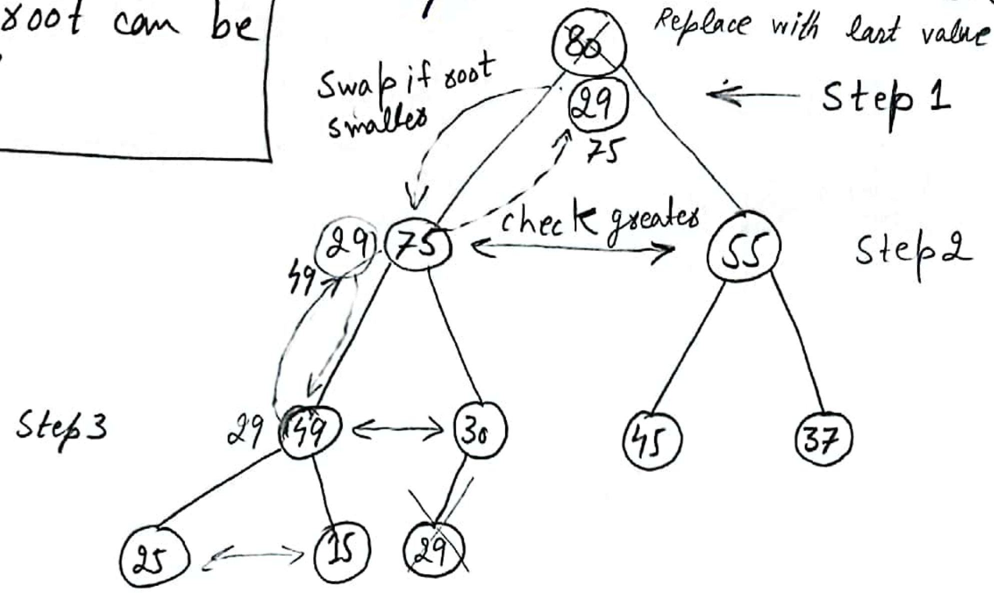
```
insertHeap (A, n, value)
{ n = n+1; A[n] = value;
  i = n;
while(i>1) { Parent = ⌊i/2⌋;
  if (A[Parent] < A[i]) [swap (A[Parent], A[i]); i = parent;
  else return
}
```

A = array
n = number of elements
value = value to be inserted.

Delete in Max heap

Only root can be deleted

Delete 80



	29									—
A	80	75	55	49	30	45	37	25	15	29 ← Free
	1	2	3	4	5	6	7	8	9	10

Step 1 – Delete 80, replace root with last value.

Step 2 – Check children of 29 i.e new root

Left child = $2*i = 2\times 1 = 2$

right child = $2*i+1 = 2\times 1+1 = 3$

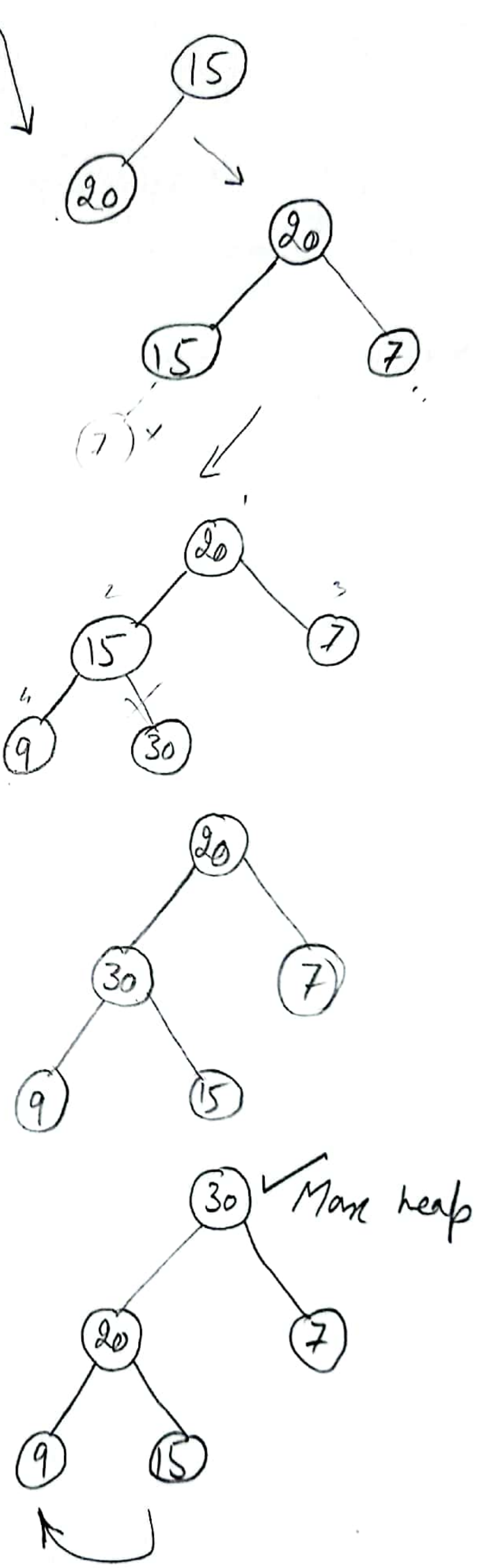
```
if (A[LC] ≥ A[RC]) && (A[i] < A[LC])
    { swap (A[i], A[LC])
      i = LC
    }
else if (A[LC] < A[RC]) && (A[i] < A[RC])
    { swap (A[i], A[RC])
      i = RC
    }
```

* Delete Another root & demonstrate sorting in Ascending order by Max Heap.
* Descending order by Min Heap.

A Fall 2022

01-02-2023

	1	2	3	4	5
A	15	20	7	9	30



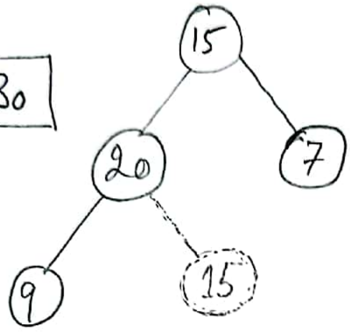
	1	2	3	4	5
A	30	20	7	9	15

* Delete 30 = temp

15	20	7	9	30

↓

20	15	7	9	30

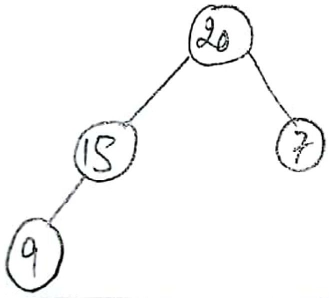


* Delete 20

9	15	7	20	30

↓

15	9	7	20	30

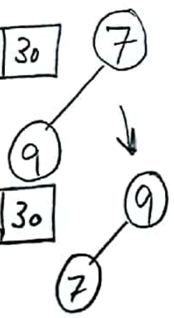


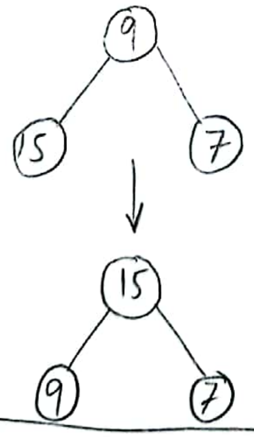
* Delete 15

7	9	15	20	30

↓

9	7	15	20	30





* Delete 9

7

7	9	15	20	30

Ascending Order

Time Complexity

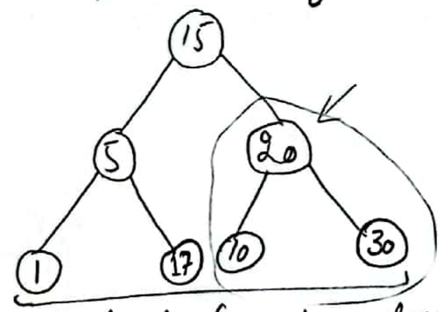
Insertion of 1 element $O(\log n)$
" " n " $O(n \times \log n)$

Deletion of 1 " $O(\log n)$
" " n " $O(n\log n)$

Total $O(2n\log n)$
→ $O(n\log n)$ in all cases

Previous method has time complexity $O(n \log n)$

Heapify Method



	1	2	3	4	5	6	7
A	15	5	20	1	17	10	30

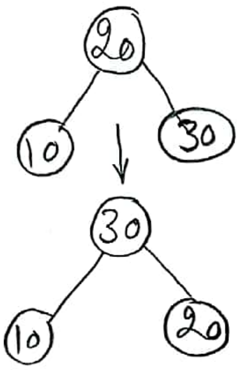
Leaf Nodes from $\lfloor \frac{n}{2} \rfloor + 1$ to n

$\lfloor \frac{7}{2} \rfloor + 1 = 4$ to 7

* All leaf nodes already heaps, as no child
* Only check non-leaf nodes.

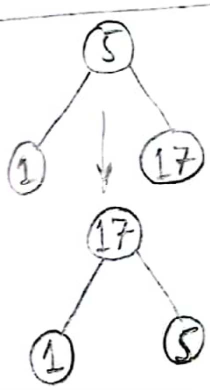
* Start Heapify from largest index of Non-leaf nodes that is start from $\lfloor \frac{n}{2} \rfloor$
* Start from Index 3

15	5	30	1	17	10	20



* Check index 2

15	17	30	1	5	10	20

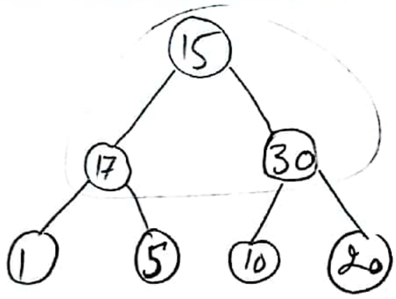


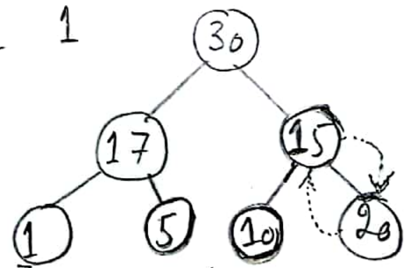
* Check index 1

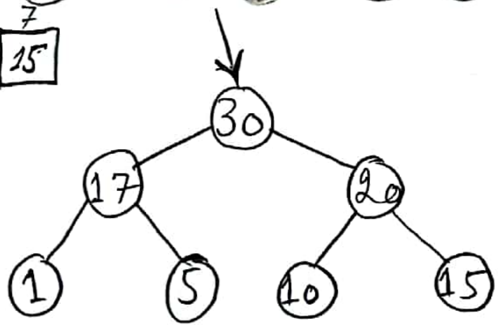
30	17	15	1	5	10	20

↓

1	2	3	4	5	6	7
30	17	20	1	5	10	15







```
maxHeapify (A, n, i)          // i= largest non-leaf node location
{
    int largest = i
    int l = 2*i
    int r = 2*i+1
   while (l ≤ n && A[l] > A[largest])
            largest = i
   while( r ≤ n && A[r] > A[largest])
            largest = r
   if (largest != i)
     {   swap (A[largest], A[i])
         maxHeapify (A, n, largest)
     }
}
```

```
heapSort (A, n)
{
   for (i = n/2 ; i>=1; i--)                 } Create Heap
          maxHeapify (A, n, i)
   for(i=n , i >=1, i--)                      } sort by deleting.
     { swap(A[1], A[i])
       maxHeapify (A,n,1)
     }
```