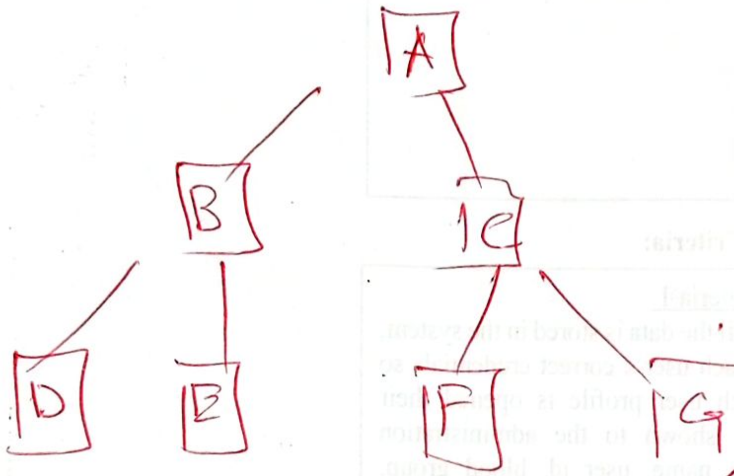
Heap Sort

1

Q: What is Binary Tree

example of BT



Array Representation = Arr =

1	2	3	4	5	6	7
A	B	C	D	E	F	G

e.g Assume array from zero

Parent = i

left child = $2*i$

right child = $i*2+1$

when code:

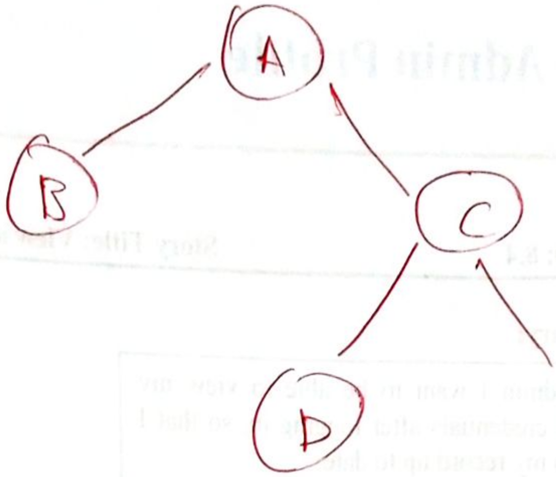
$2*i+1$

$2*i+2$

parent = $\lfloor \frac{i}{2} \rfloor$ → floor value

Parent of F = $\frac{6}{2}$ = 3

Another Example



1	2	3	4	5	6	7
A	B	C	-	-	D	E

~~A B C~~

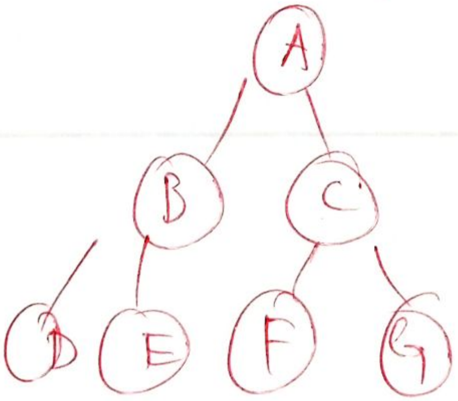
~~Full~~

Q Full Binary Tree vs Complete Binary Tree.

Full Tree

Root

$h=0$



- All levels are filled in tree

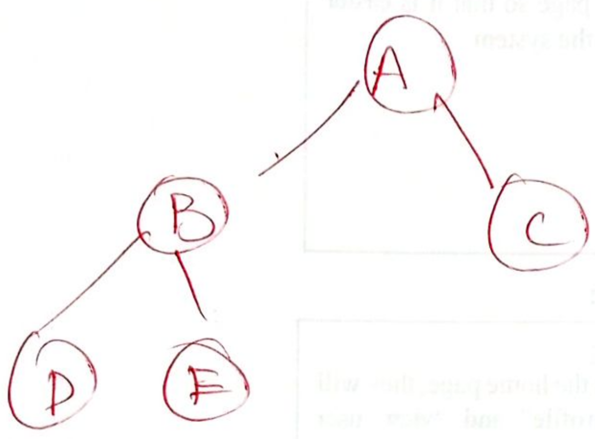
h = heigh. of tree

Full tree here $2^{h+1}-1$ number of node

Complete Binary Tree

→ Fill left to right
→ height difference = h-1

Also no empty element b/w (in array)

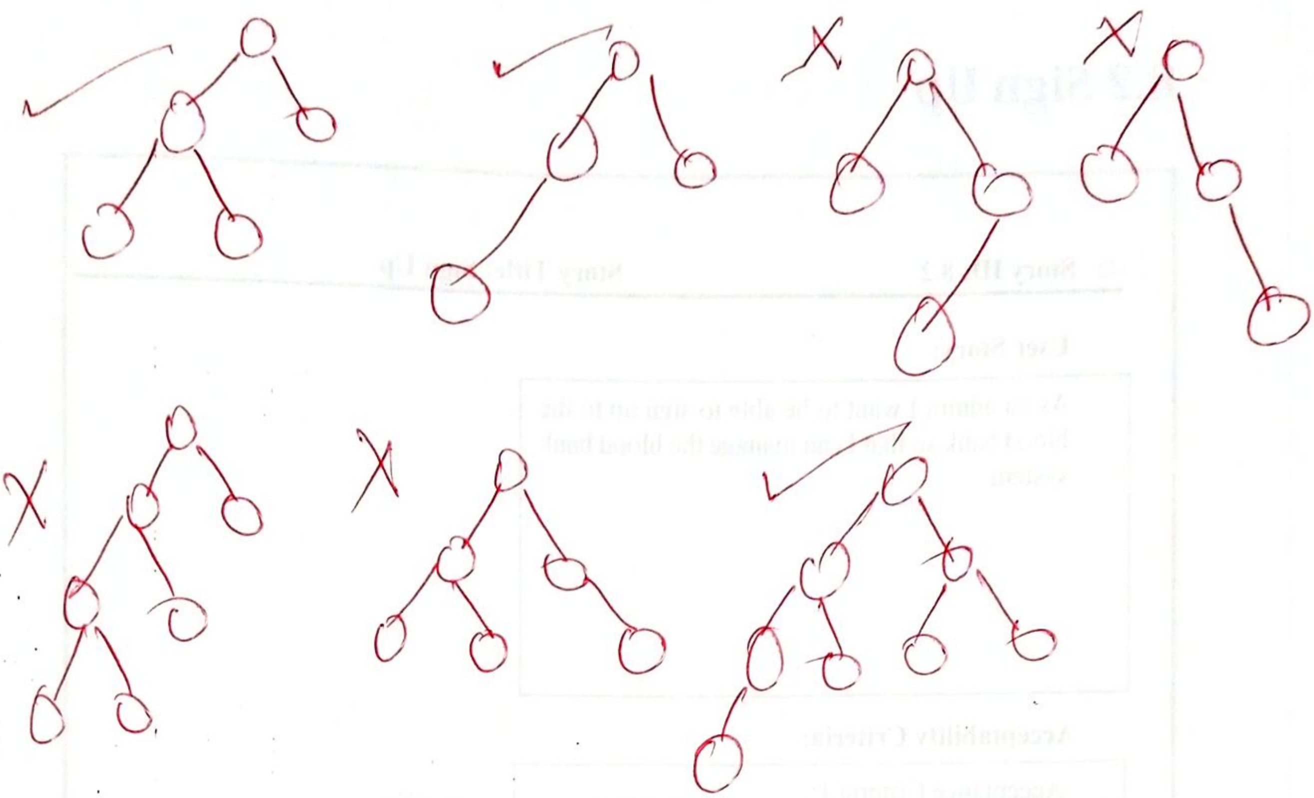


A	B	C	D	E	

So its complete but not full

All full binary tree are complete tree.

Some Examples (if complete)



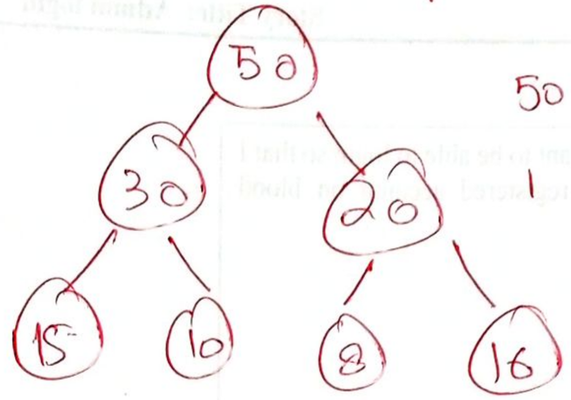
Height of Tree $= \log(n)$

Heap

BST ordered
Heap is not

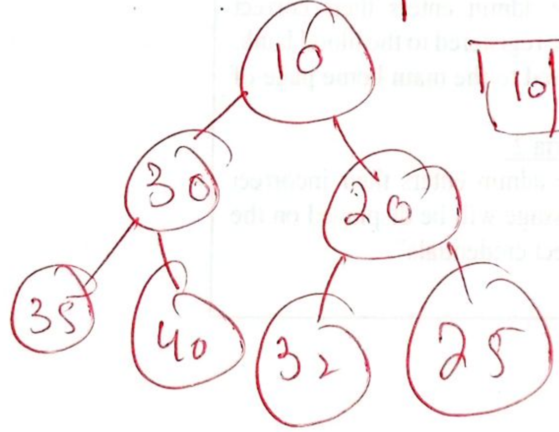
Condition
① → Complete Binary Tree
② → Root will have min or max value (all descendants)

max heap

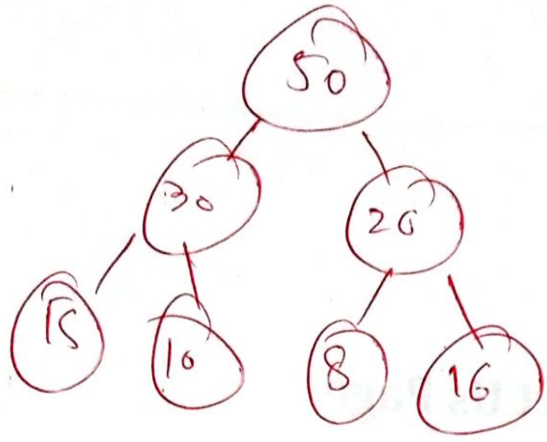


50	30	20	15	10	8	16
1	2	3	4	5	6	7

min heap



10	30	20	35	40	32	25



Lets add 60

So

50	30	20	15	10	8	16

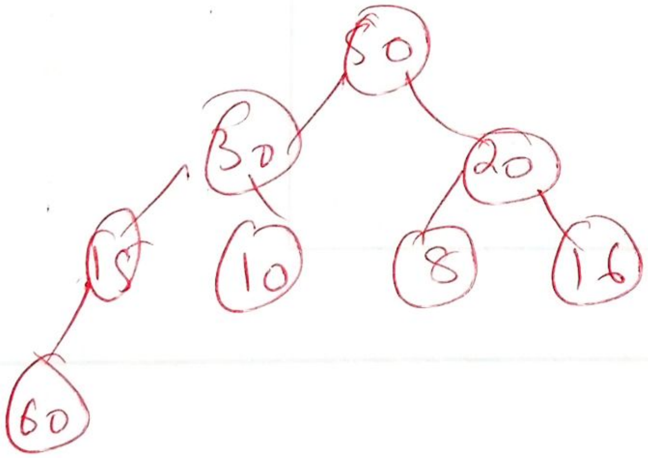
maintain the complete quality of BT.

so

insert in the end.

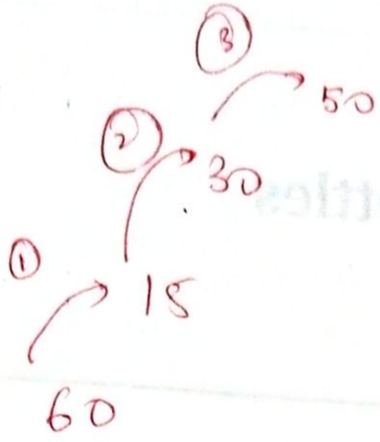
50	30	20	15	10	8	16	60

so

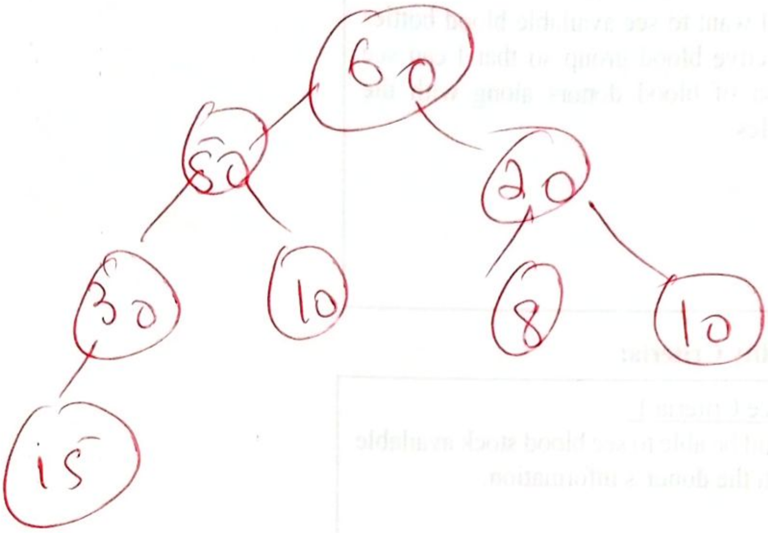


It is not heep!

Now adjust, compare with parent.



50



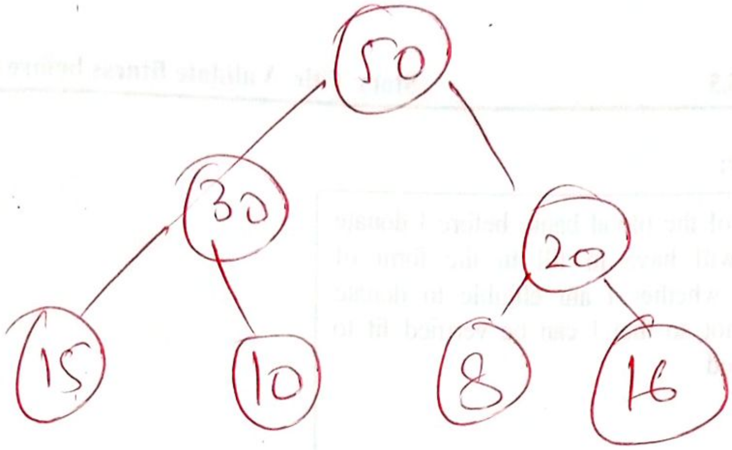
Time for insertion = $\log(n)$

As the height of tree is $\log n$.

if no swaps, then $O(1)$.

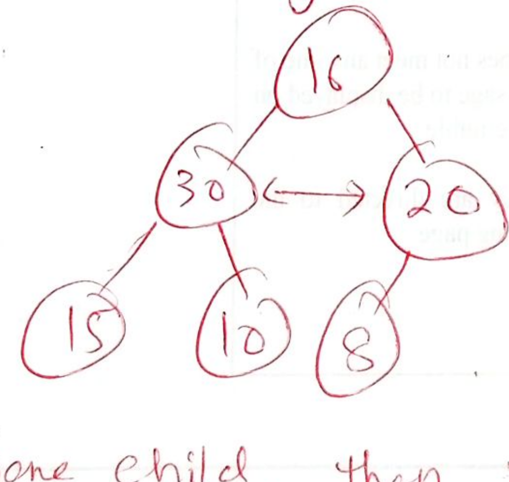
Delete
(Max heap.)

⇒ We can only delete root element



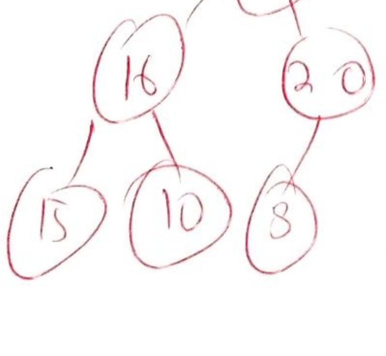
So remove 50, move last to Top

So 16 will go as root



→ complete but no heap.

Compare child, then replace larger child with 16.



→ check with all remaining decendents

maximum adjustment = (log n) for deletion (a)

Q: So what happens when we delete another element?

So we will get the next big element!

In end, we will have the smallest one.

So array size was ?

~~16~~	~~30~~	~~20~~	~~15~~	~~10~~	~~8~~	

30	16	20	15	10	8	

Free

what we deleted, add it in free space, maintain it, not part of heap

If we delete again,

30 will be deleted

So

20	16	10	15	8	30	50

So this is the idea of heap sort

Heap Sort

2 Step

for given list

(1) Create heap

(2) Delete all elements

result = sorted list

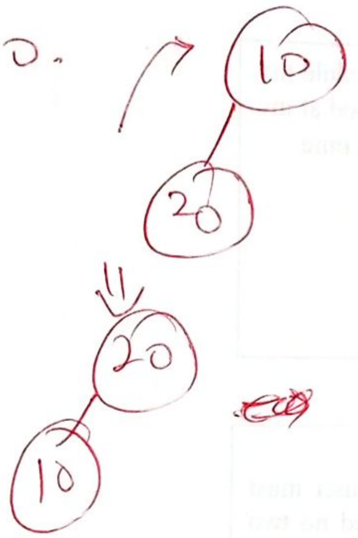
Assume (Initial Array).

10 20 15 30 40

1 2 3 4 5

10 HEAP

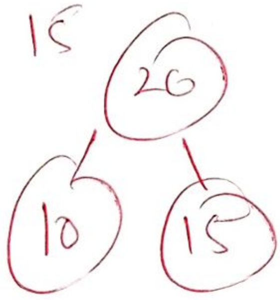
Insert 20.



10	20

20	10

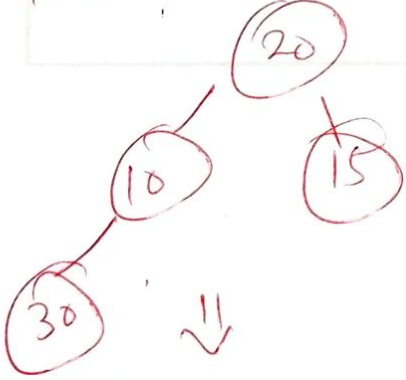
Insert 15



20	10	15

Compare with parent

Insert 30

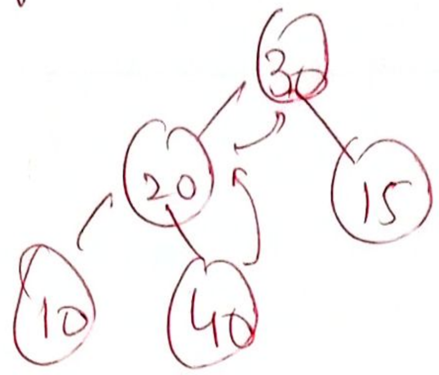


20	10	15	30

20 30 15 10 =>

30	20	15	10

Insert 40

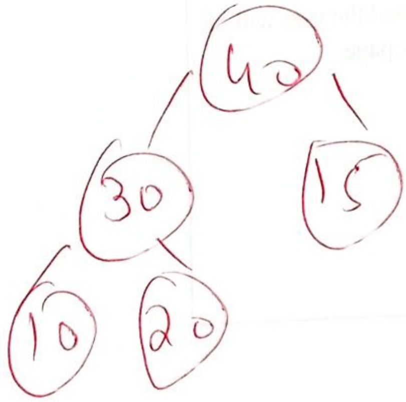


Now compare with parent

30	20	15	10	40

30	40	15	10	20

40	30	15	10	20
1	2	3	4	5



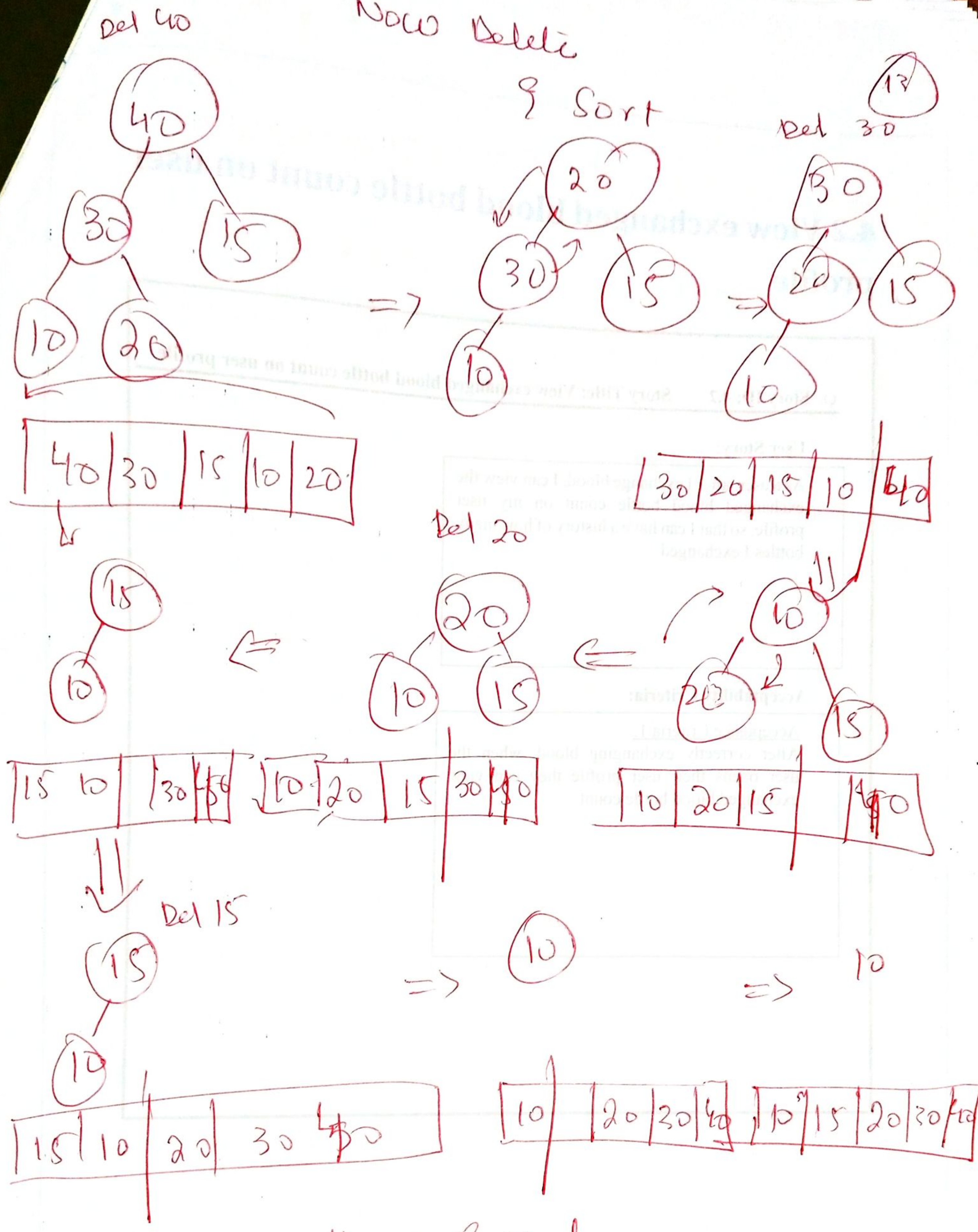
How much time taken

we inserted n elements

height of tree (to move & adjust)

$\boxed{= n\log n}$

Now Delete & Sort



Heap Sorted

Now complexity analysis

~~No of deleted element = n log n~~

~~No of insertie~~

insertion complexity = n log n

no of elements — height adjustment

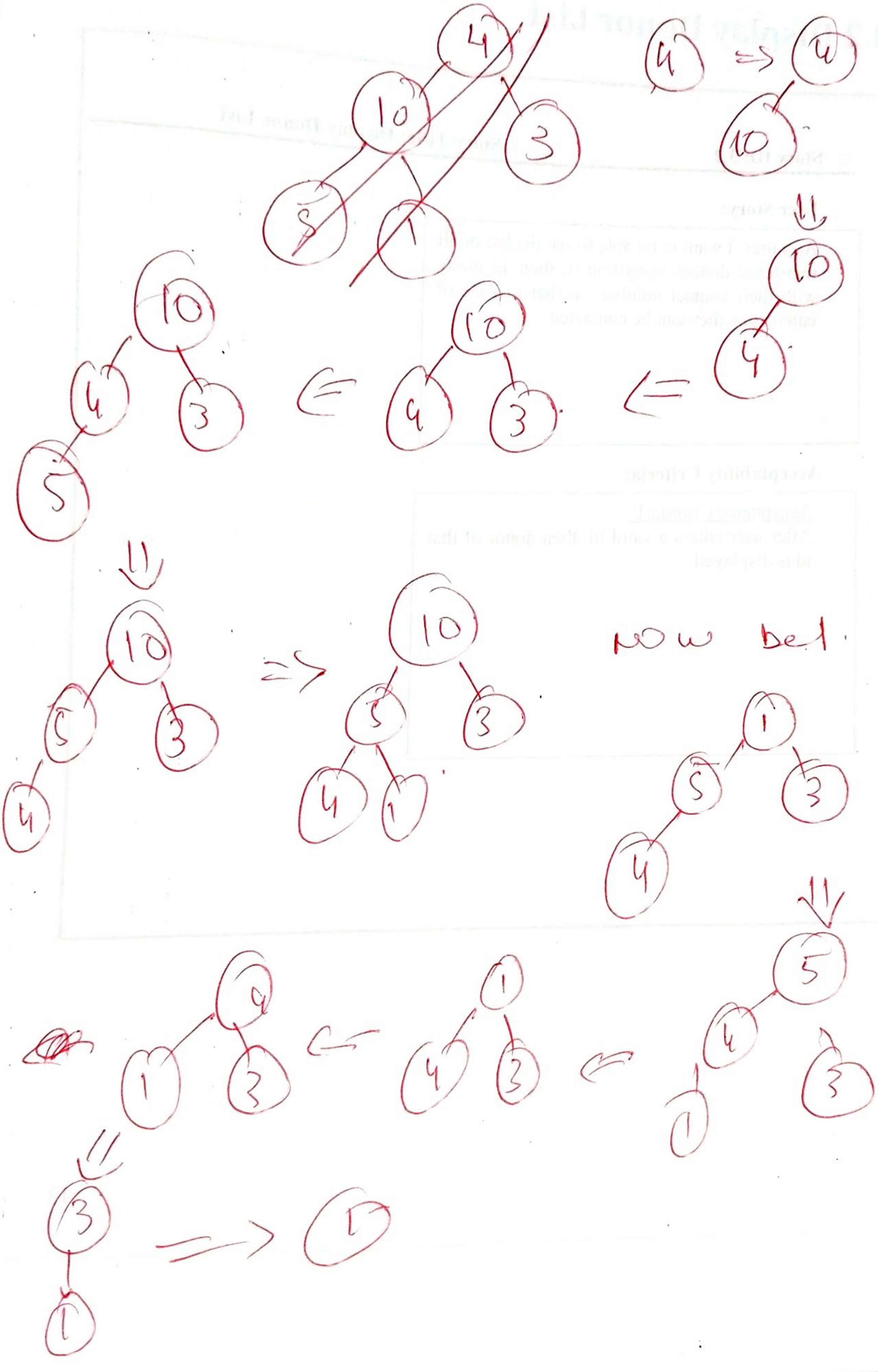
deletion complexity = n log n

= n log n + n log n

= 2 n log n = O(n log n)

Example

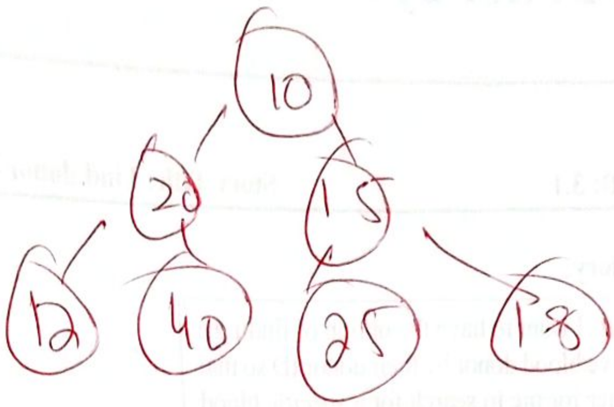
4, 10, 3, 5, 1



Heapify

10 20 15 12 40 25 18

Now make b/t.

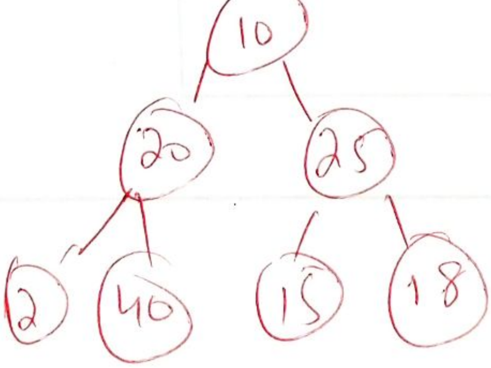


Now

We will adjust element from down to up & from leaf

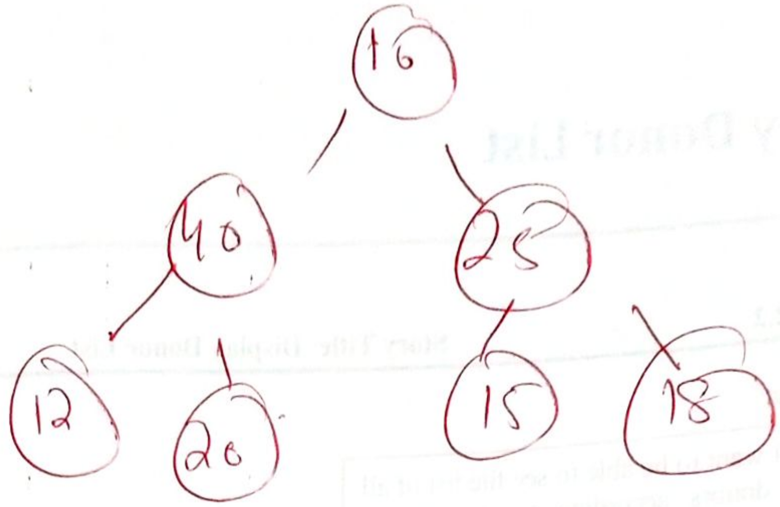
so check 18, no child so heap do it till 12.

Now check 15 → have child, adjust

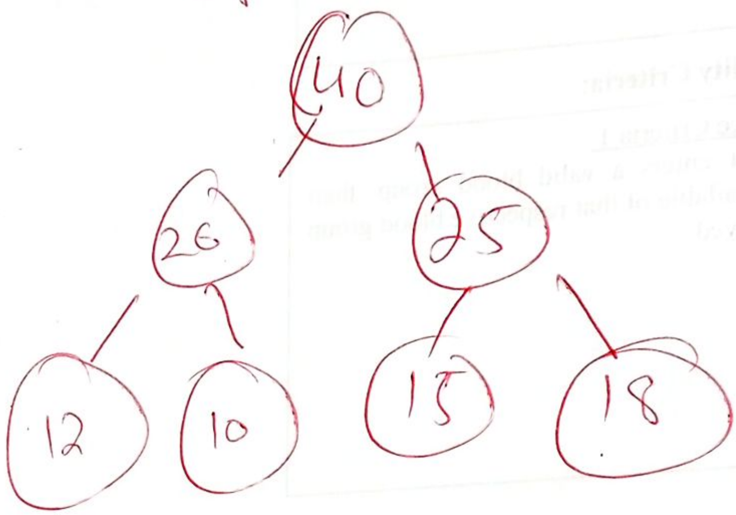


only 1 adjust

Now Check 20



check 10 (for all decendu).



so analytically, time is $O(n)$

why $\frac{7}{2} \Rightarrow \underline{3}$

Algo

```
sort( arr )
{   ~~i=0~~
    N = arr.length

for (int i = N/2 - 1 ; i >= 0 ; i--)

  heapify (arr, N, i);

for ( i = N - 1 ; i > 0 ; i--)
{     temp = arr [0];

      arr[0] = arr [i];

      arr [i] = temp;

      heapify (arr, N-i, i)

}
}
```

Algo

Heapify

```
heapify (int arr[], N, i)

largest = i

int l = 2 * i ;
right = 2 * i + 1 ;

if ( l < N && arr[l] > arr[largest]

    largest = l

if ( r < N && arr[r] > arr[largest]
    largest = r.

if ( largest != i )
{
    swap arr[i] , arr[largest]
}
heapify (arr, N, largest);
```