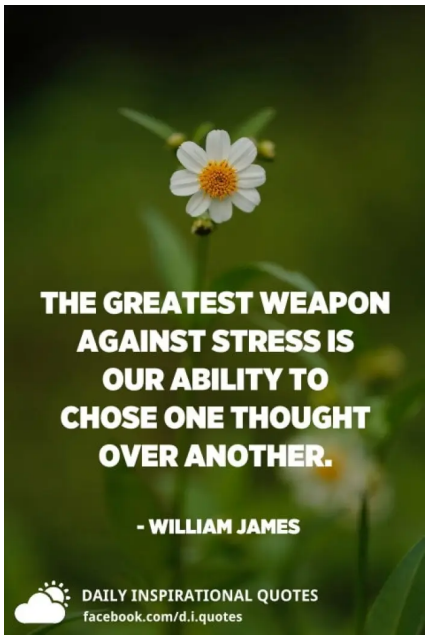
Heaps 1



Good Evening

Content → Minimise the cost of connecting ropes

→ Heap & its property

→ Array implementation of Trees

→ Min heap

→ insertion

→ extractmin

→ Build a min heap

→ Merge K sorted lists

Minimum Cost

Given with some ropes, pick 2 ropes & connect them together. we have to keep on doing this until we have one single rope left.

If you connect 2 ropes, you have to pay some **cost**

sum of length of ropes

The task is to connect all the ropes with **minimum cost**

ar[ ] : 2 5 2 6 3

			cost
2	6	= 8	8
2	8	= 10	10
5	3	= 8	8
10	8	= 18	18
			44

			cost
2	6	= 8	8
5	8	= 13	13
2	13	= 15	15
3	15	= 18	18
			54

ar[ ] : 2 2 3 5 6 4 7 11

			cost
2	2	= 4	4
3	4	= 7	7
5	6	= 11	11
7	11	= 18	18
			40

Idea → Pick 2 smallest rope for the combination

Issue → Sort the array → Pick 2 ele from front

Create a new rope

Insert this rope again

Sort again

TC = $N \log n * N^2$

Requirements
- Insert
- Delete min()
- Get min()

Data Structure = Heap
- insertion — O(log n)
- Extract min() / Extract max — O(log n)

TC = O(n log n)

Heap Data Structure

↳ Nothing but a Binary Tree with two properties

Structure (CBT)

Complete Binary Tree

→ All levels should be completely filled

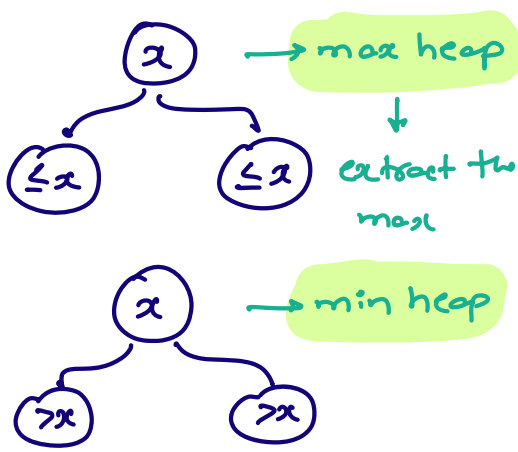
Exception is for last level

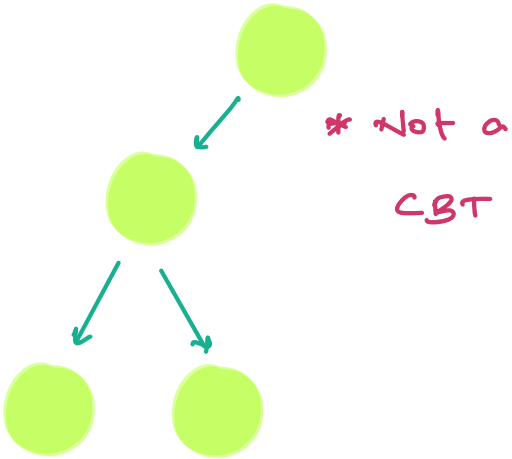
↳ but for last level as well

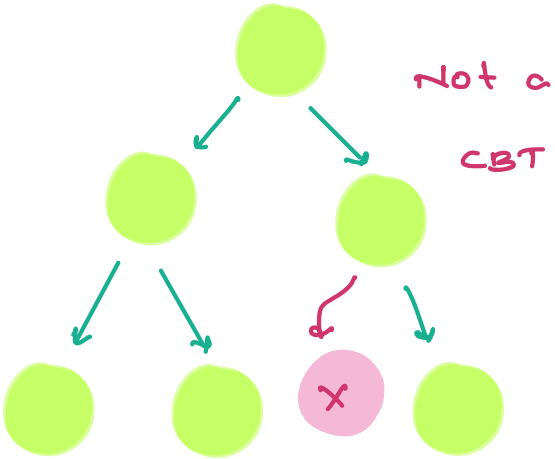
↳ children added left to Right

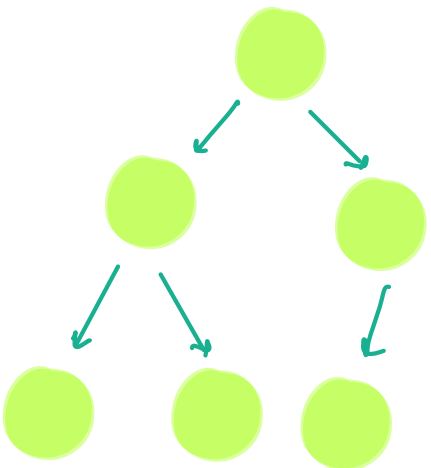
Order of Ele (HOP)

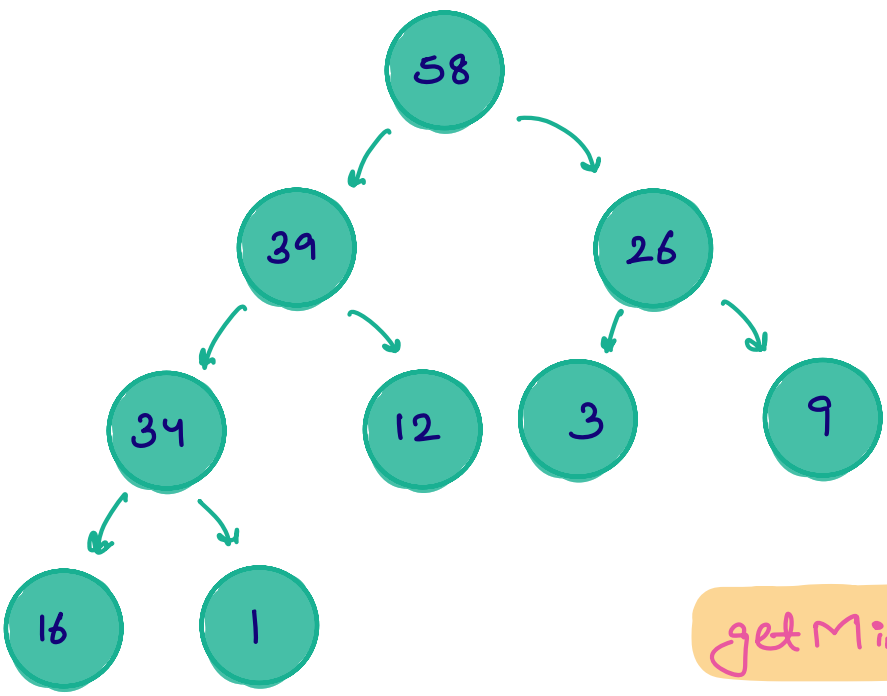
Heap order property

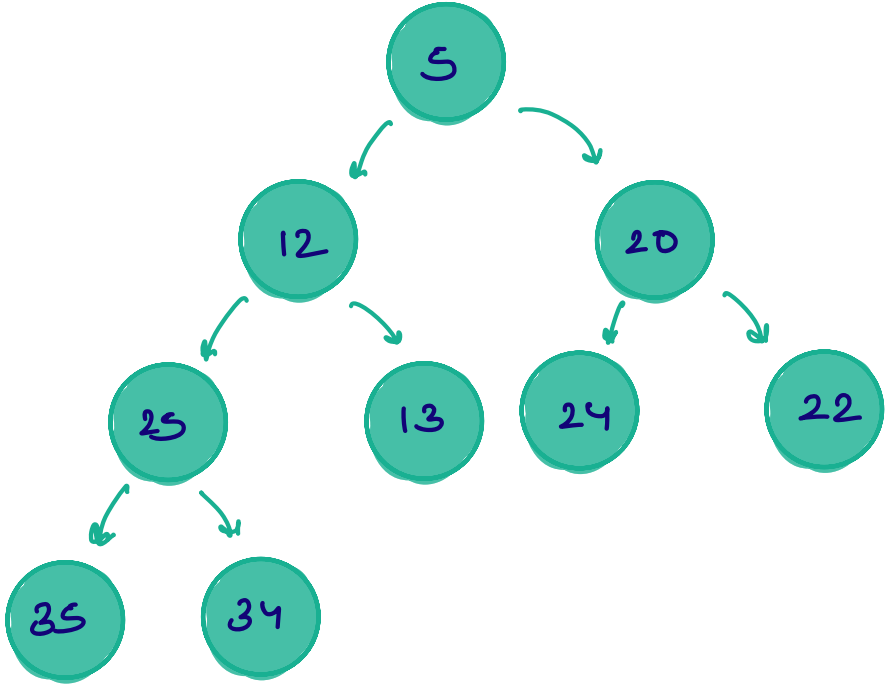



* Not a
CBT


Not a
CBT
X


CBT

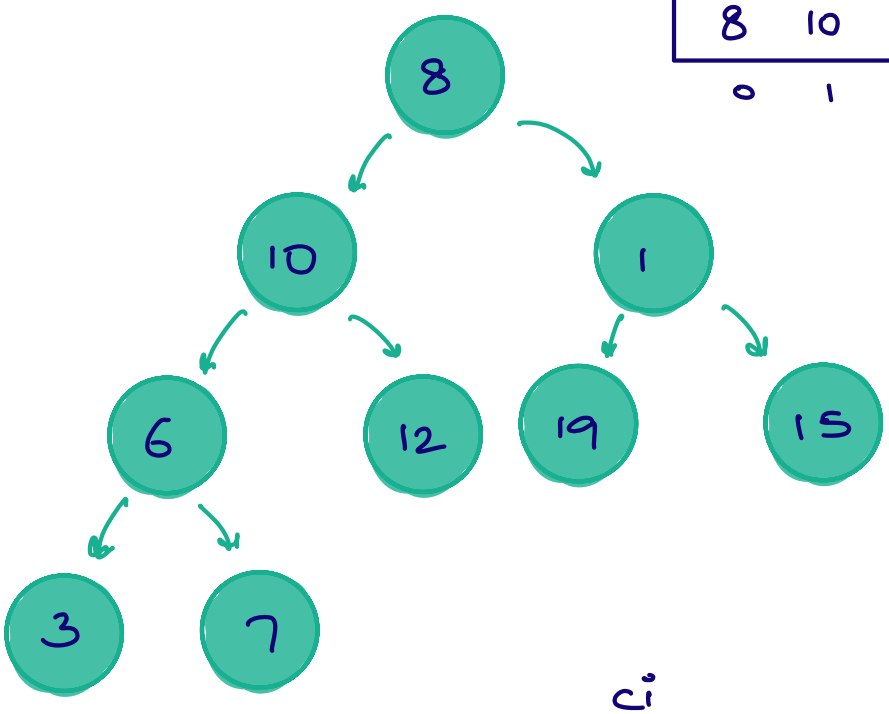

58
39
26
34
12
3
9
16
1
Max heap
getMin()/getMax() → O(1)



Min heap

* Implementation of Heaps Using Array

8	10	1	6	12	19	15	3	7
0	1	2	3	4	5	6	7	8



0 → 1 → 2*0+1
0 → 2 → 2*0+2

1 → 3 → 2*1+1
1 → 4 → 2*1+2

2 → 5 → 2*2+1
2 → 6 → 2*2+2

ci

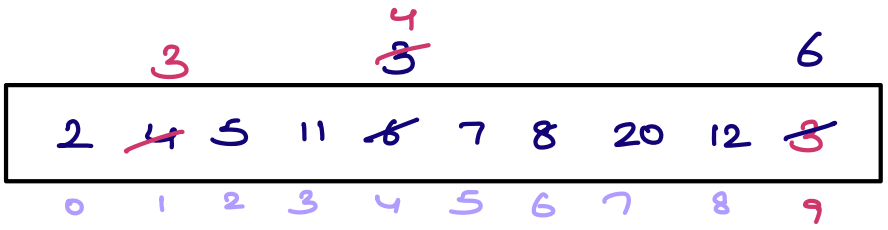
$i \rightarrow 2i+1$ & $2i+2$

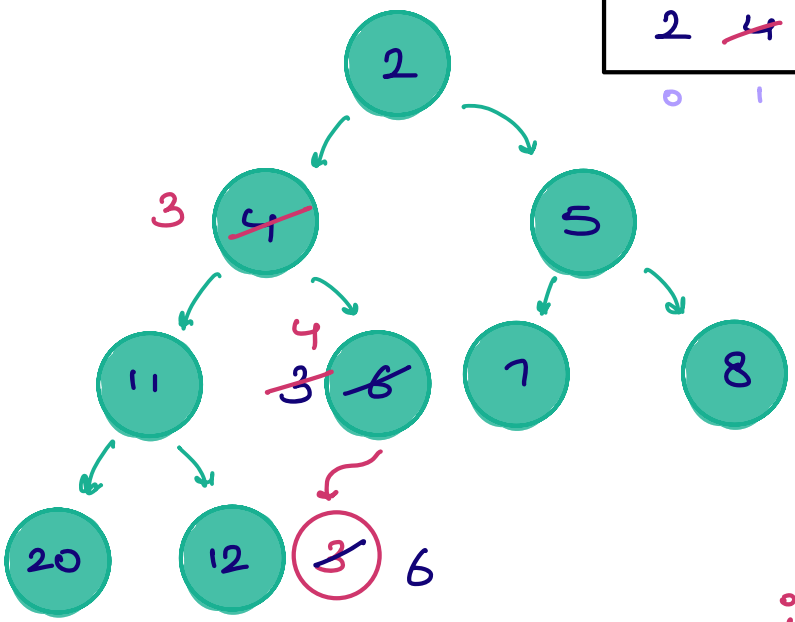
$i \rightarrow (i-1)/2$

pi

* Min Heap

Insertion in min heap → insert 3





i	par	ar[par] ≤ ar[i]
9	$\frac{9-1}{2} = 4$	No → swap
4	$\frac{4-1}{2} = 1$	No → swap
1	$\frac{1-1}{2} = 0$	Yes → break

TC : O(log n)

SC : O(1)

Upheafify()

↓

Insert k in heap

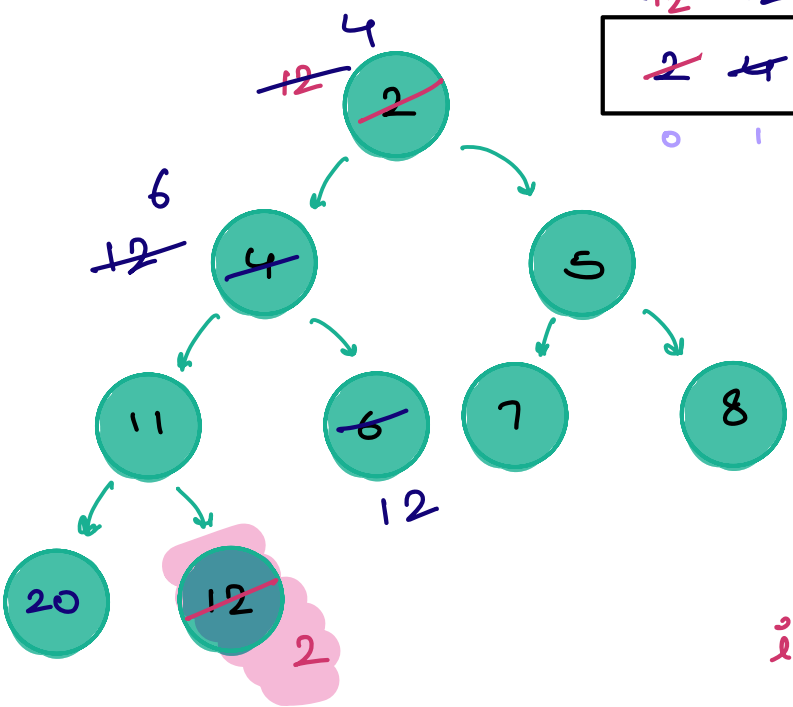
heap[ ] ; ← Dynamic Array

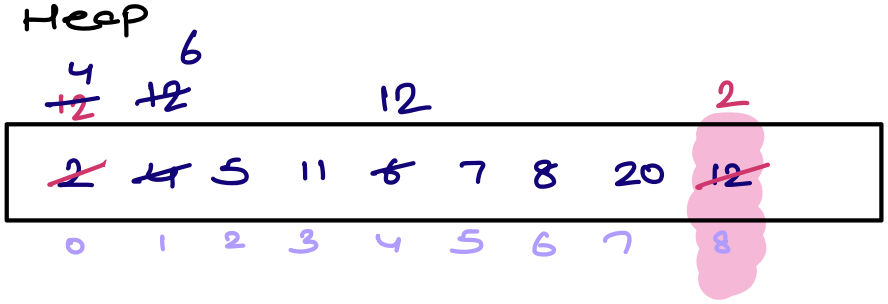
heap.insert(k)

```
i = heap.size() - 1 ;
while (i != 0){
    par = (i-1)/2
    if (heap[par] > heap[i])
        swap(heap[par], heap[i]);
        i = par;
    else{
        break
    }
}
```

Break → 9:56 → 10:06 pm

Extractmin() in Min Heap





ci

$i=0$ → $2*0+1 = 1$

$2*0+2 = 2$

swap(12,4)

$i=1$ → $2*1+1 = 3$

$2*1+2 = 4$

swap(12,6)

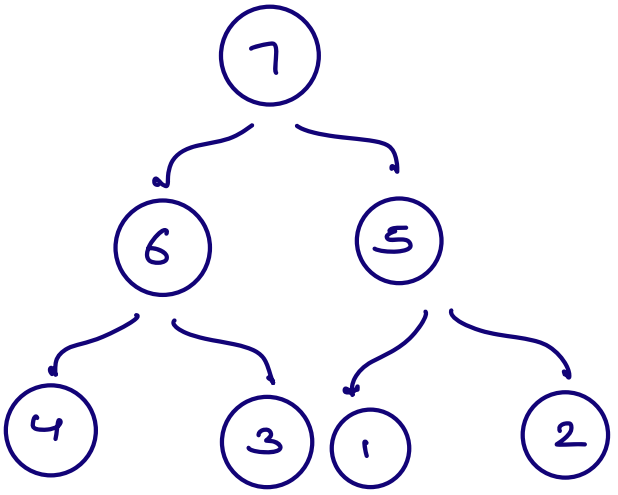
TC: O(log n)

$i = 4$ ⟶ stop

Downheapify

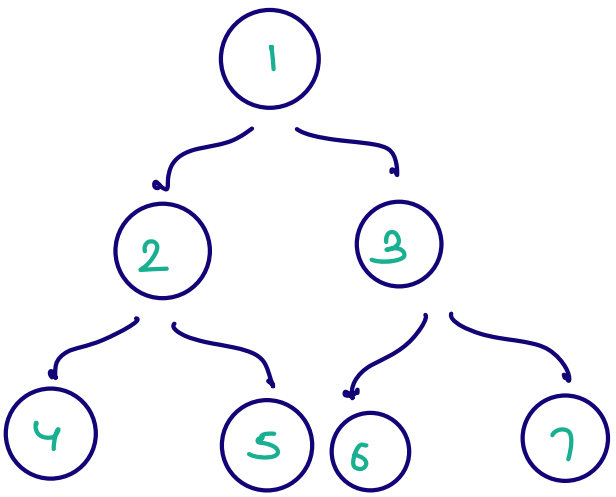
{TO DO}

* Build the Minheap



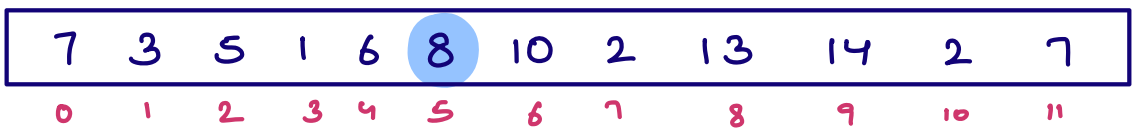
sort the array

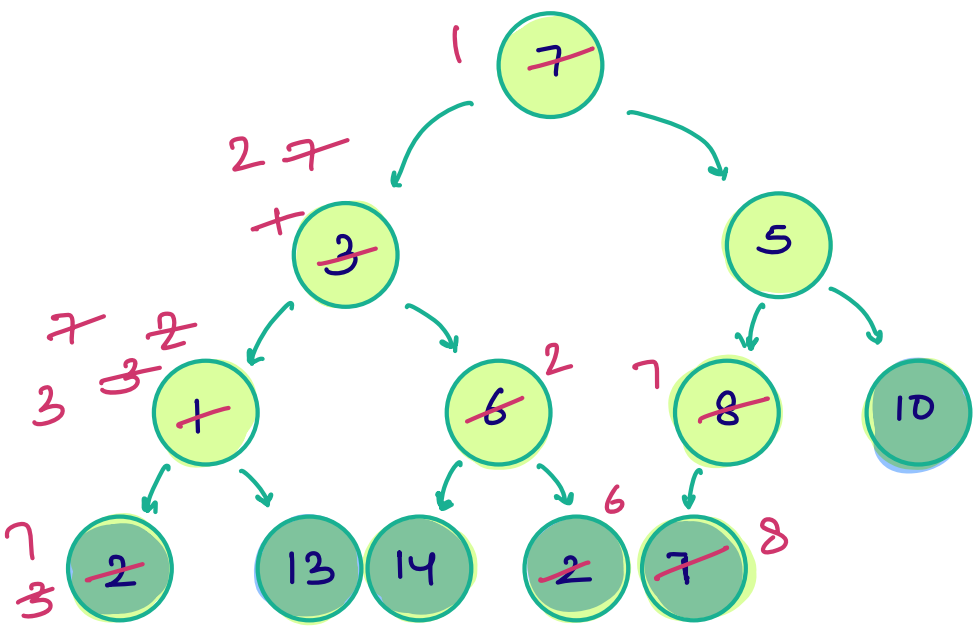
TC: O(nlogn)



*

arr[] =





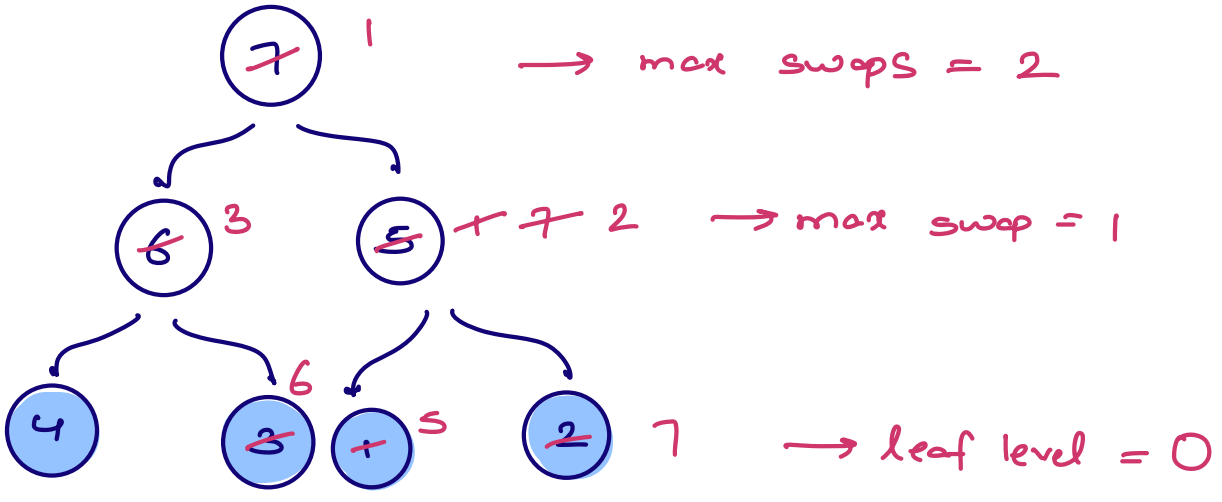
* Working from first non leaf node

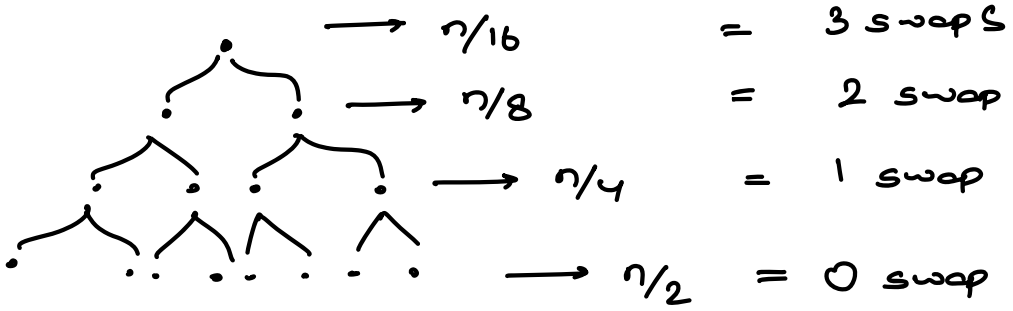
$i = 11\ (n-1)$

$par = \frac{(i-1)}{2} = \frac{(11-1)}{2} = 5$

$= \frac{n-1-1}{2} = \frac{n-2}{2} = \frac{n}{2} - 1$

7	6	5	4	3	1	2





Total work = $\frac{n}{2} * 0 + \frac{n}{4} * 1 + \frac{n}{8} * 2 + \frac{n}{16} * 3 \cdots$

$= \frac{n}{2} \left(0 + \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \cdots\right)_{AGP}$

$S = \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \cdots$

$\frac{S}{2} = \quad \frac{1}{4} + \frac{2}{8} + \frac{3}{16} + \cdots$

$-$

$\frac{S}{2} = \left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots\right)_{GP}$

$S_\infty = \frac{a}{1-r}$

$= \frac{1/2}{1 - \frac{1}{2}} = 1$

$\frac{S}{2} = 1$

$\boxed{S = 2}$

TC : $\frac{n}{2} * 2 = n \simeq O(n)$

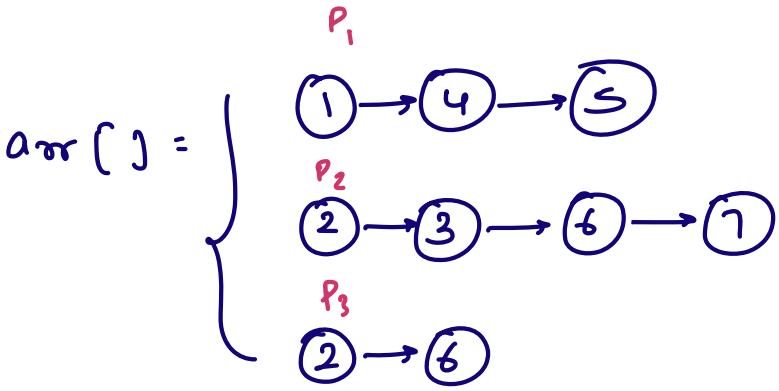
SC : $O(n)$

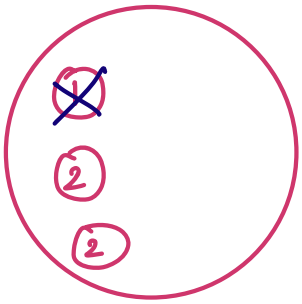
```
* for ( i = n/2 - 1 ; i ≥ 0 ; i-- )
  |
  |   downheapify (heap, i)
  |
  }
```

```
downheapify ( int [] heap, int i )
|
|   while (2i+1 < n) {
|   |
|   |   x = min ( heap[i], heap[2i+1], heap[2i+2])
|   |
|   |   if (x == heap[i]) { return; }   // parent is smallest
|   |
|   |   else if ( x == heap[2i+1] ) {
|   |   |
|   |   |   swap ( heap[i], heap[2i+1]) ;  // leaf child is
|   |   |                                     smallest
|   |   |   i = 2i+1
|   |   }
|   |   else {
|   |   |
|   |   |   swap ( heap[i], heap[2i+2] ;  // right child
|   |   |                                    is smallest
|   |   |   i = 2i+2 ;
|   |   }
|   |
|   }
|
}
```

* Merge K sorted list

Given an arr of linkedlist, each linkedlist is sorted in ascending order. Merge all the linkedlist in one sorted list.






```
PriorityQueue<Node> pq = new PriorityQueue<>(
                                (a, b) -> a.val - b.val)

for (Node x : arr){
    pq.add(x)
}
```

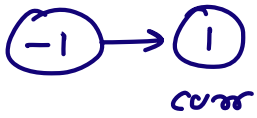
} first Node of every list in PQ

```
ListNode ans = new ListNode(-1)

ListNode curr = ans

while (pq.size()>0)
    curr.next = pq.remove();
    curr = curr.next;
    if (curr.next != null){
        pq.add(curr.next);
    }
}

return ans.next;
```



TC: O(n*k) log k

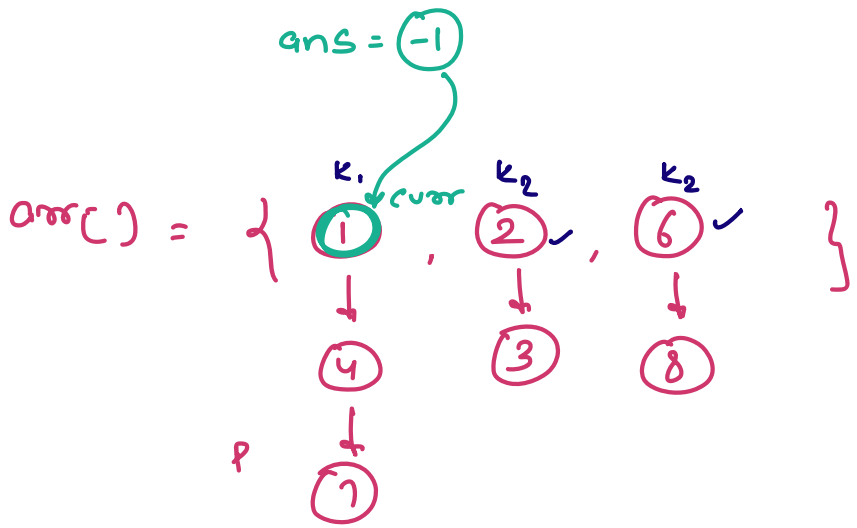
SC: O(k)

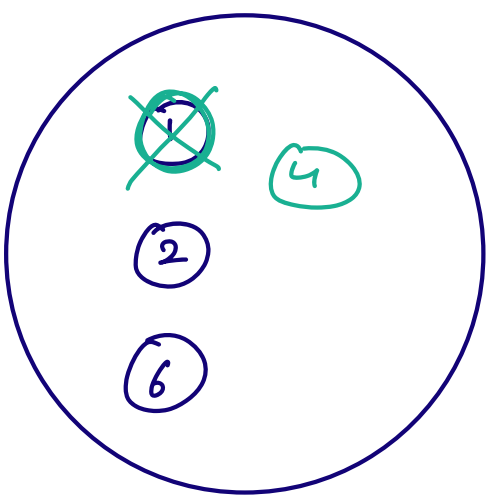
Inbuild heap

```
PriorityQueue<Integer> pq = new PriorityQueue<>();
```

min heap

```
PriorityQueue<Integer> pq = new PriorityQueue<>(Collections.reverseOrder());
```






```
while (pq.size()>0)

    curr.next = pq.remove()

    curr= curr.next

    if (curr.next != null)

        pq.add(curr.next);
```

04. **Recover sorted ar[]**

↳ Given an arr[], which is formed by **swapping 2 distinct index** positions in a increasing sorted ar[]. Get original sorted ar[]

Sorted ar[] = { 4 10 12 14 18 19 25 28 }

* Input ar[] = { 4 10 19 14 18 12 25 28 }

0 1 2 3 4 5 6 7

< < > < > < <

ar[] = { 2 6 23 10 14 19 8 40 }

< < > < < > <

ar[] = { 3 6 10 15 12 17 20 33 }

< < < > < < <

ar[] = { 3 6 9 31 14 18 24 12 35 }

< < < > < < > <

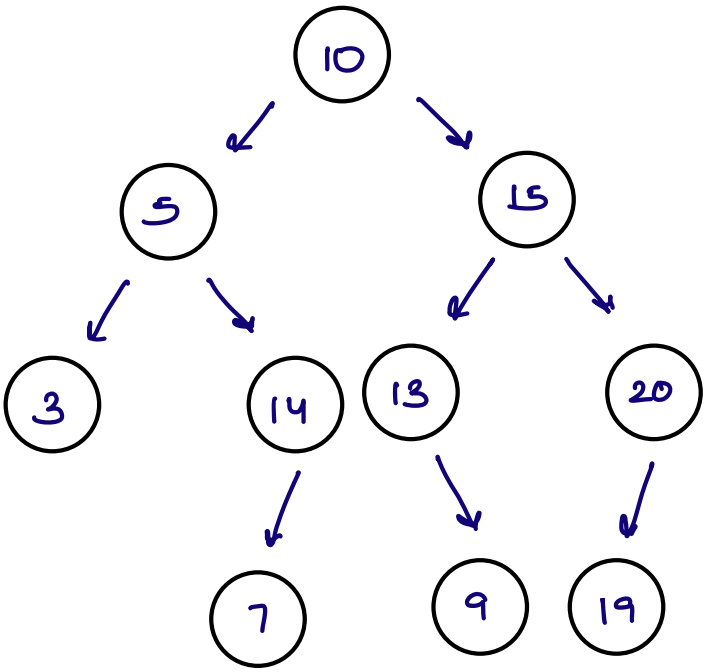
Idea → Iterate on ar[] & compare 2 adj ele

01 1st time comparison fail → 1st val = ar[i]

2nd val = ar[i+1]

02 2nd time comparison fails → 2nd val = ar[i+1]

Q5. Given a BST, which is formed by swapping 2 distinct nodes, recover original BST.



Inorder = { 3 5 7 14 10 13 9 15

Idea 1 → Do the inorder & store every ek in AL

Now, we can follow the above question approach

TC: O(n)

SC: O(n)

```
Node first = null , second = null
Node prev = null;
void recover ( root )
{
    if (root == null) return ;
    recover (root.left)
    if ( prev != null && prev.data > root.data && first == null)
    {
        first = prev
        second = root
    }
    else if ( prev != null && prev.data > root.data)
    {
        second = root;
    }
    Prev = root ;
    recover (root.right)
}
```