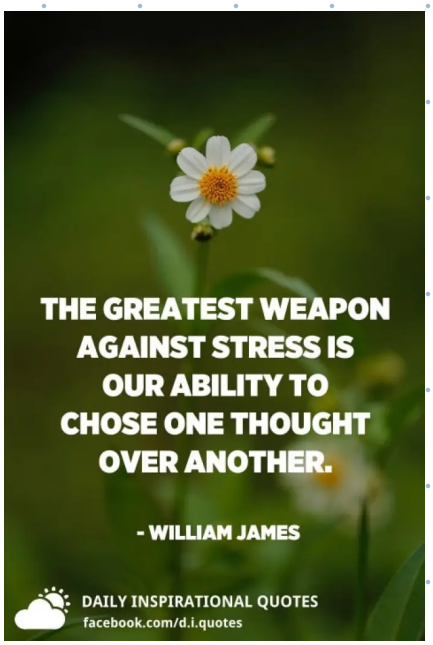
Heaps 1



Good Morning

Content

- Minimise the cost of connecting ropes
- Heap & its property
- Array implementation of Trees
- Min heap
  - Insertion
  - Extract min
- Build a min heap
- Merge N sorted arrays

* **Connecting ropes**

Given an array that represents size of diff ropes. In a single operation, you can connect two ropes. Cost of connecting two ropes is sum of length of ropes you are connecting. Find **minimum cost** of connecting all ropes.

A[] = {2 5 2 6 3}

cost

2 6 = 8

2 8 = 10

5 3 = 8

10 8 = 18

cost 44

A[] = { 5 2 8 3}

A[] = { 5 3 10}

A[] = {10 8}

A[] = {18}

A[] = {2 5 2 6 3}

cost

2 5 = 7

7 2 = 9

9 6 = 15

15 3 = 18

cost = 49

A[] = {7 2 6 3}

A[] = {9 6 3}

A[] = {15 3}

A[] = {18}

* Idea → Pick two ropes of smallest lengths

A[] = { 2 5 2 6 3 }

cost

2 2 = 4

3 4 = 7

5 6 = 11

11 7 = 18

Cost = 40

A[] = { 5 6 3 4 }

A[] = { 5 6 7 }

A[] = { 11 7 }

A[] = { 18 }

Idea → Sort the array → Pick 2 elements from front

Create a new rope

Insert new rope

↓

Sort array again

TC: O(N * NlogN)

Idea 2 → Use insertion sort → TC: $O(N^2)$, SC: O(1)

cost

1 2 = 3

3 3 = 6

4 6 = 10

cost = 19

A[] = { 1 2 3 4 }

A[] = { 3 3 4 }

A[] = { 4 6 }

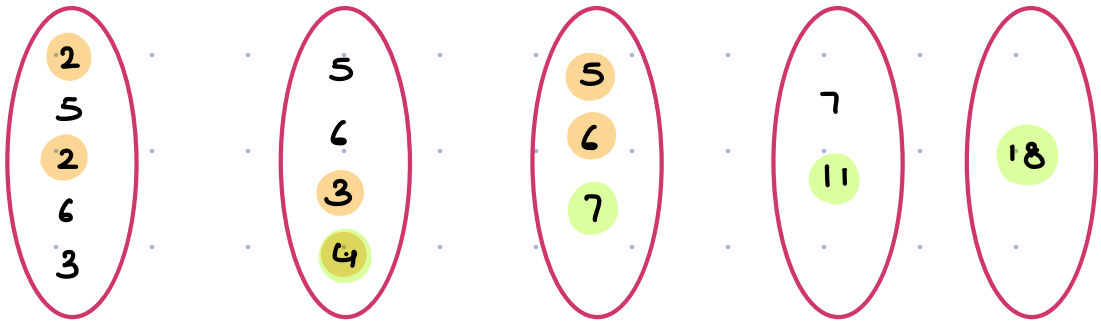
A[] = { 10 }

Requirement → Insert
Requirement → Delete min

Heaps / Priority Queue → DS that is used whenever we have to do frequent extraction of min / max & insertion.

Priority Queue → insert — TC : O(log n)
Priority Queue → getmin() / getmax() — TC : O(log n)

A[] = { 2 5 2 6 3 }



MinPQ

* Priority Queue

PriorityQueue < Integer > pq = new PriorityQueue <>(); → min PQ

PriorityQueue < Integer > pq = new PriorityQueue <>(Collections.reverseOrder()) → max PQ

Ans -

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

for (i=0; i<n; i++){
    pq.add(A[i]);
}

cost = 0
while (pq.size() > 1){
    int val1 = pq.remove();
    int val2 = pq.remove();

    cost += val1 + val2

    pq.add(val1 + val2);
}

return cost;
```

TC : O(N log N)

SC : O(N)

* **Priority Queue** → Binary Tree with two properties

Structure

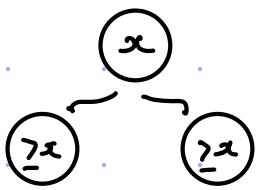
Complete Binary Tree

→ All levels should be completely filled.

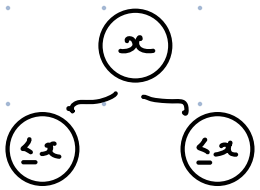
→ Last level can have exception of not entirely filled but it should also be filled from **L to R**

Order of ele

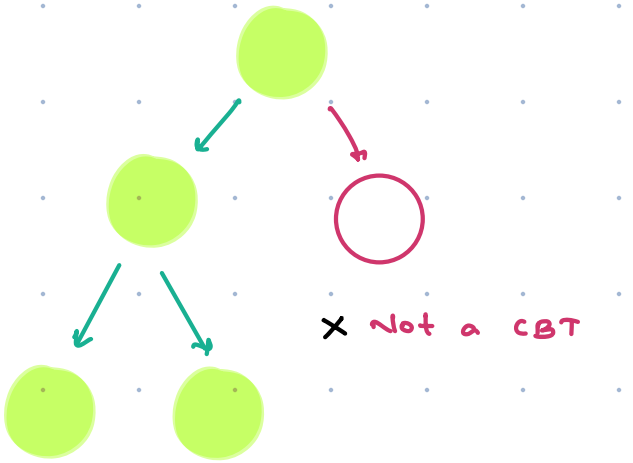
Heap order property

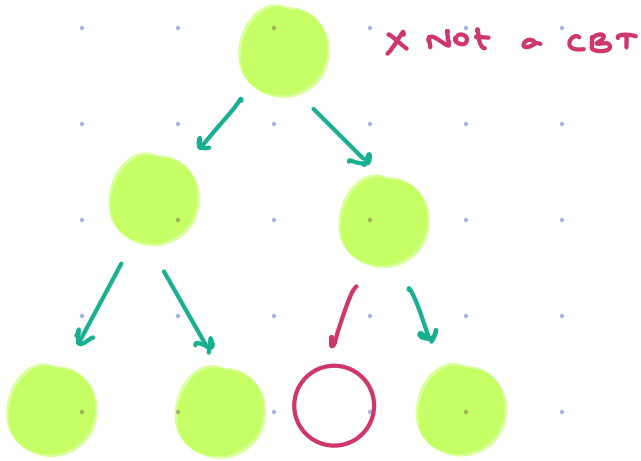


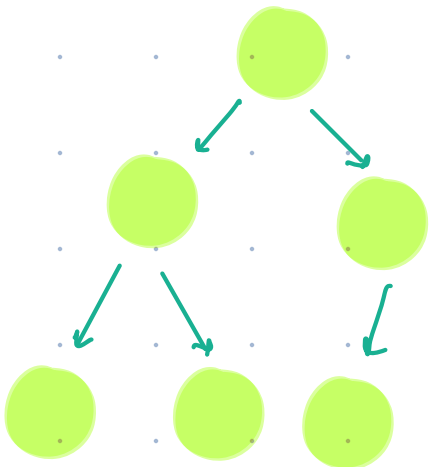
Min PQ



Max PQ

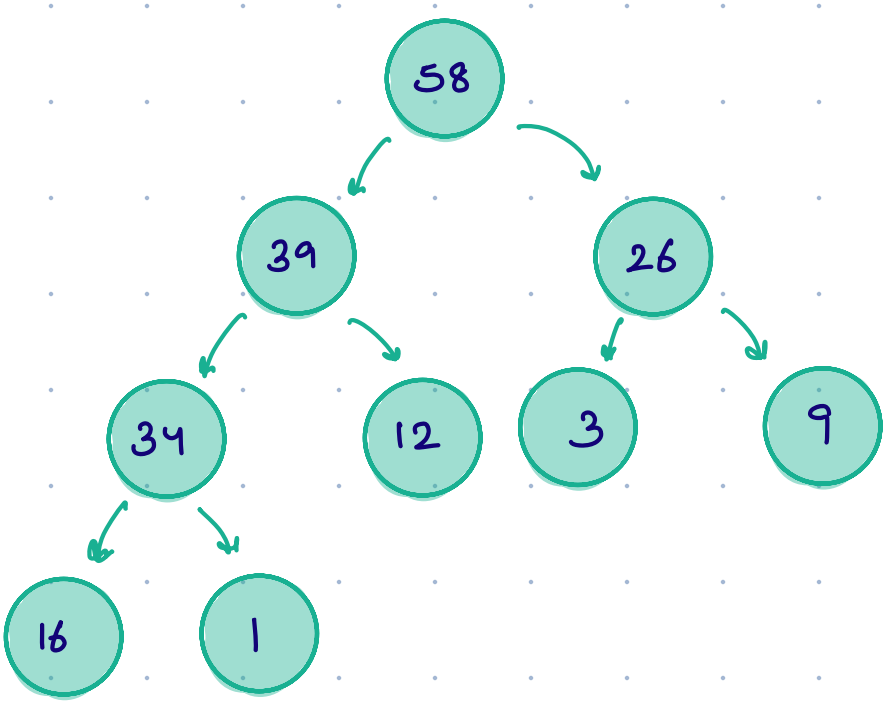






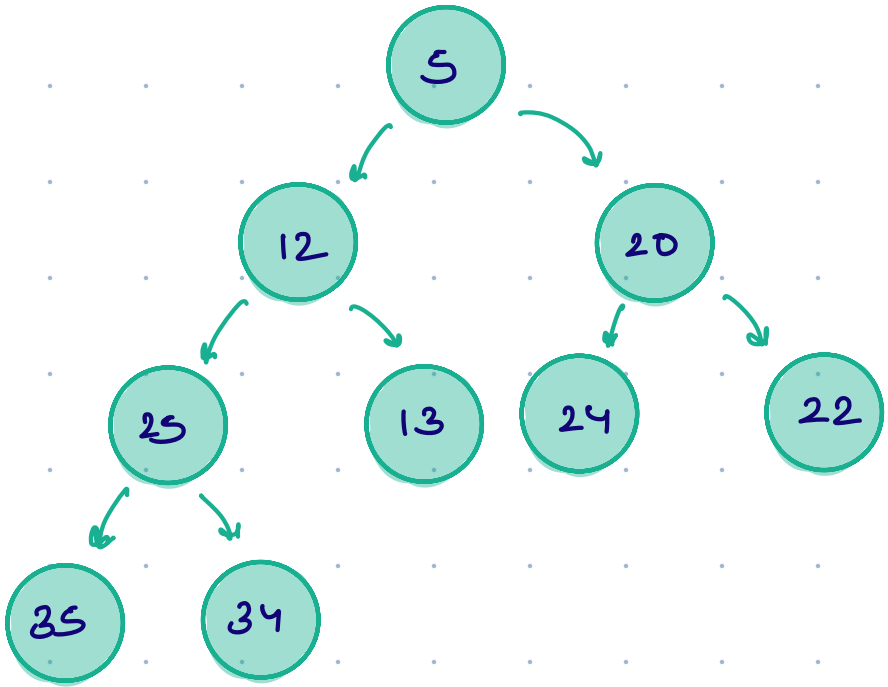
✓

Valid CBT



Max PQ

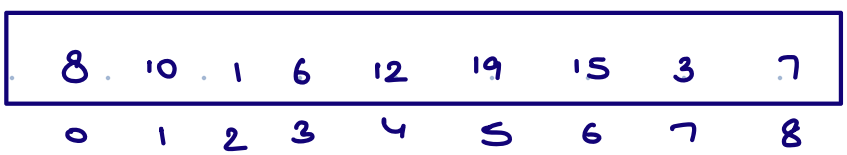
Parent val will always be greater or = than the children

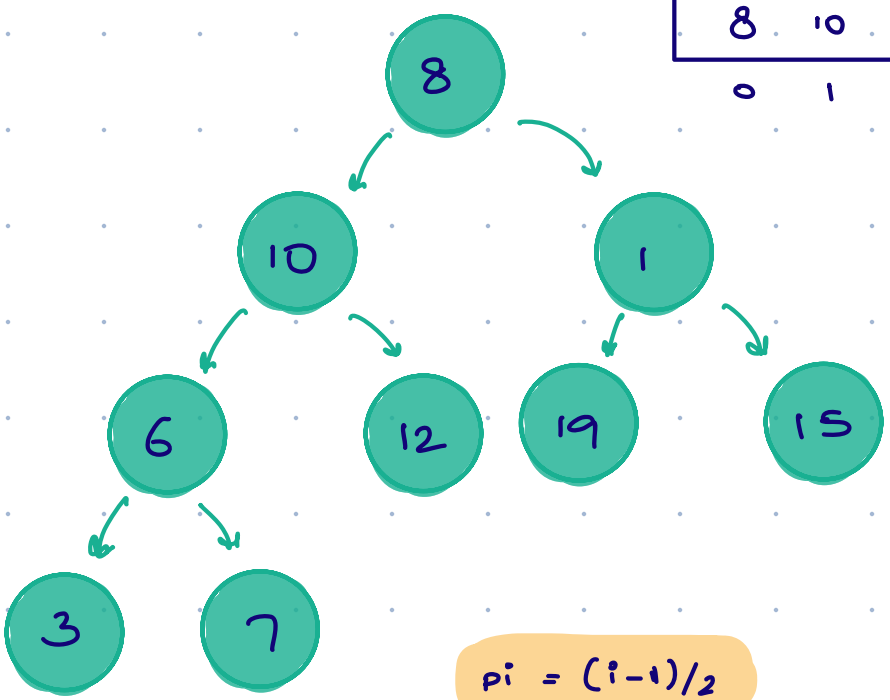


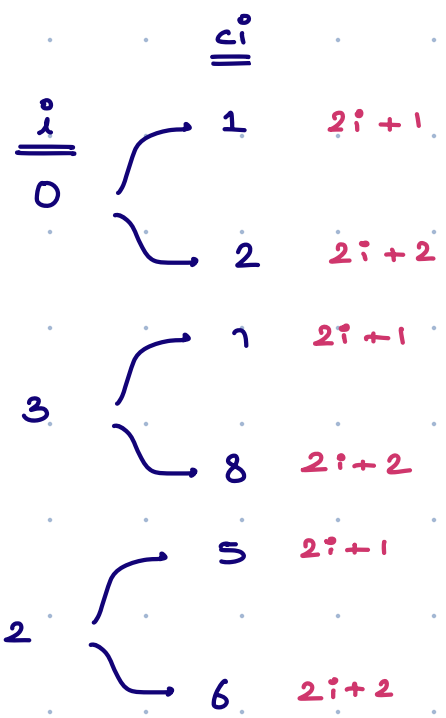
Min PQ

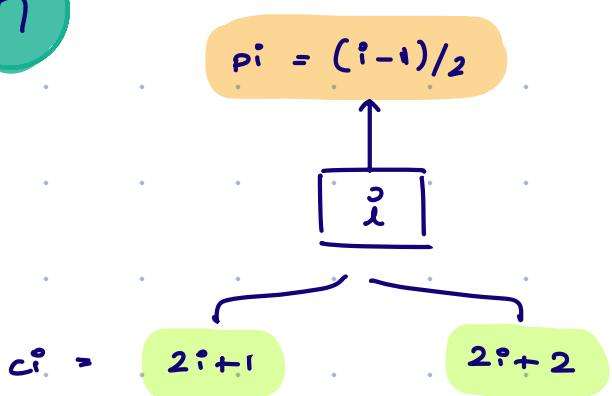
Parent value will always be smaller than its children

* Array Implementation of Trees





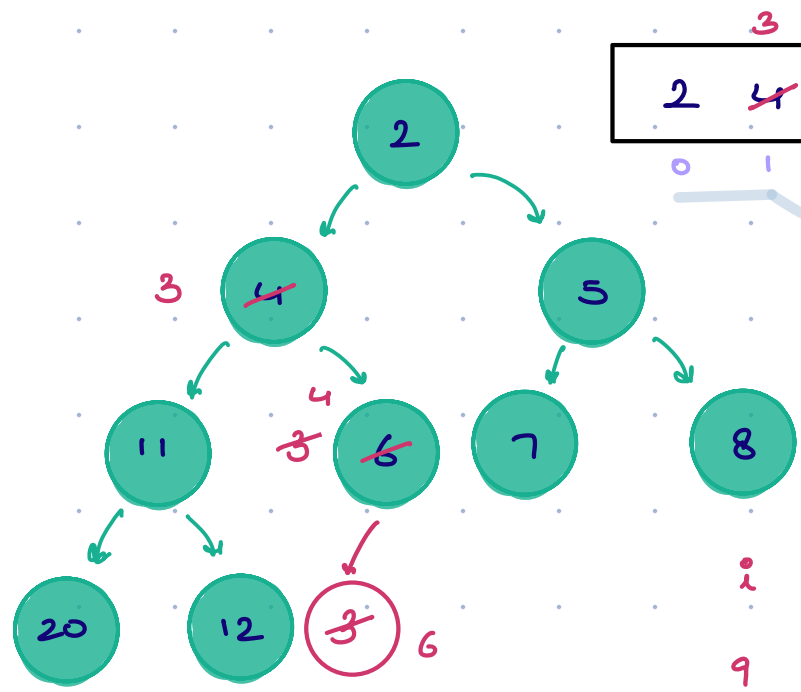


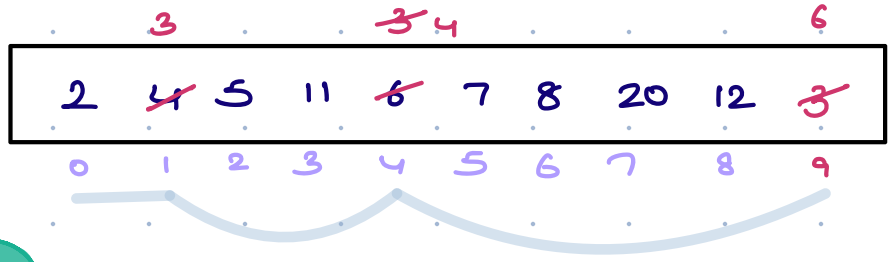


* Min PQ

Tc: O(log n)

* Insertion in Min heap → insert 3





i	par	A[p] ≤ A[i]
9	$\frac{9-1}{2} = 4$	No → swap
4	$\frac{4-1}{2} = 1$	No → swap
1	$\frac{1-1}{2} = 0$	Yes, stop

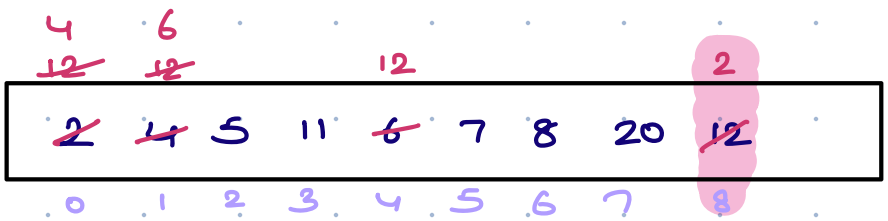
Tc: O(log n)

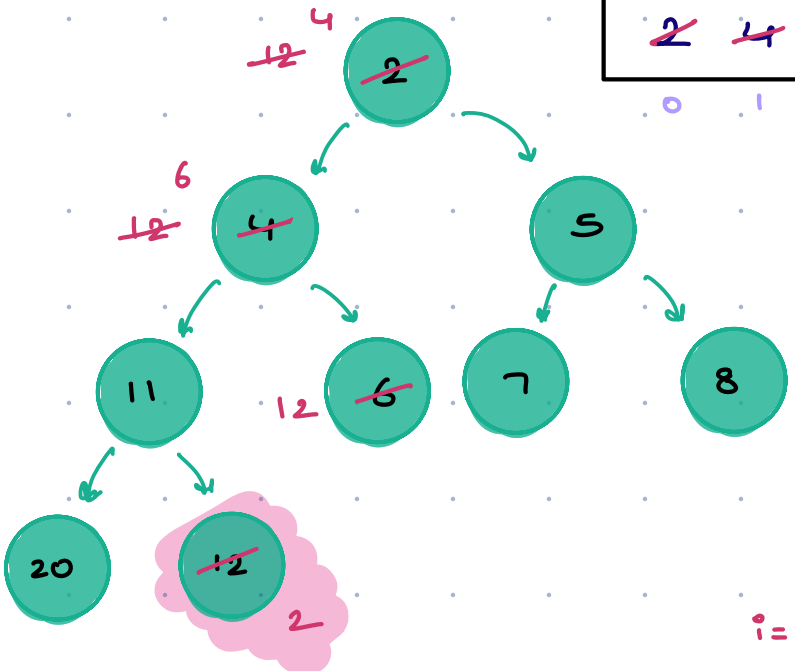
Sc: O(1)

```
void upheapify ( ArrayList<Integer> heap, int k)
    heap.add(k);
    i = heap.size() - 1;
    while ( i > 0) {
        par = (i-1)/2;
        if ( heap.get(par) > heap.get(i)) {
```

```
        swap ( heap , par, i );
        i = par;
    }
}
else {
    break;
}
}
```

* Extract min





swap(first idx & last idx)

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)

i=4 ⟶ stop

```
int smallest ( heap)
    swap ( heap , 0 , heap.size() -1 )
    int rv = heap.remove ( heap.size()-1 );
    downheapify( heap, 0 );
    return rv;
}
```

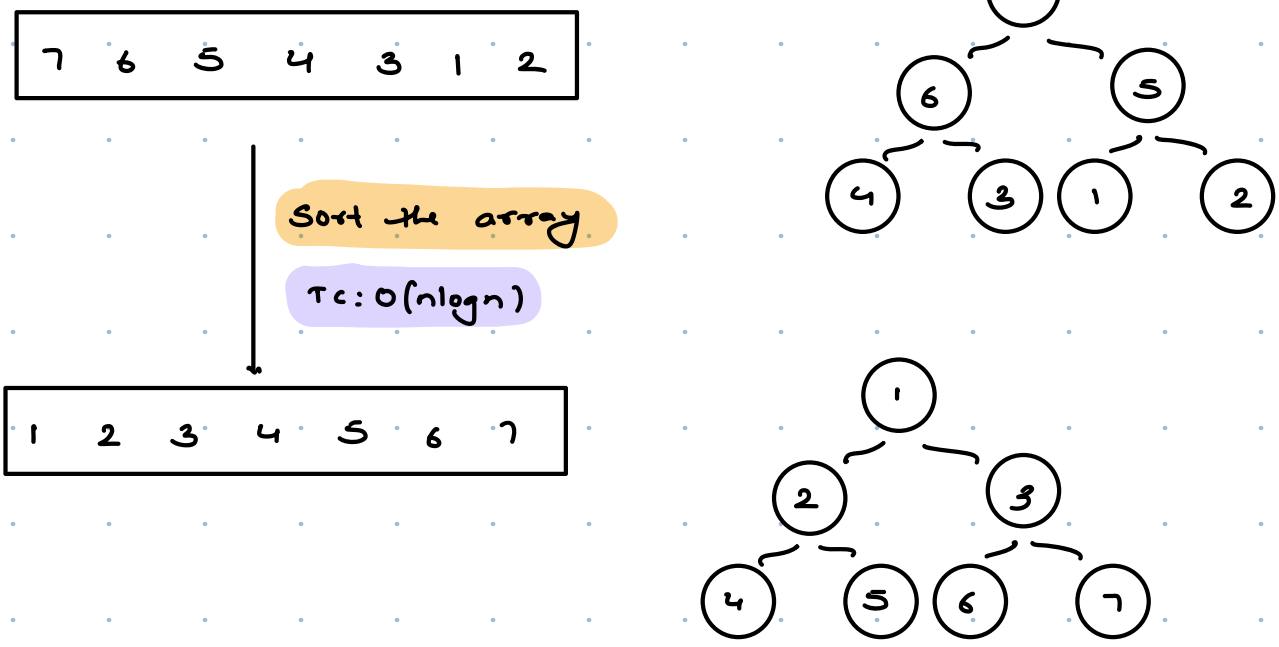
Tc: O(log n)
Sc: O(1)

```
void downheapify ( heap, i , n )
    while ( 2i+1 < n ){
        x = min( heap.get(i) , heap.get(2i+1); heap.get(2i+2))
        if ( x == heap.get(i) ) return ;
        else if ( x == heap.get(2i+1)) {
            swap ( heap, i, 2i+1)
            i = 2i+1
        }
        else {
            swap ( heap, i, 2i+2);
            i= 2i+2;
        }
    }
}
```

8:42 AM → 8:52 AM

* Build a Min PQ

Idea 1



* Idea 2

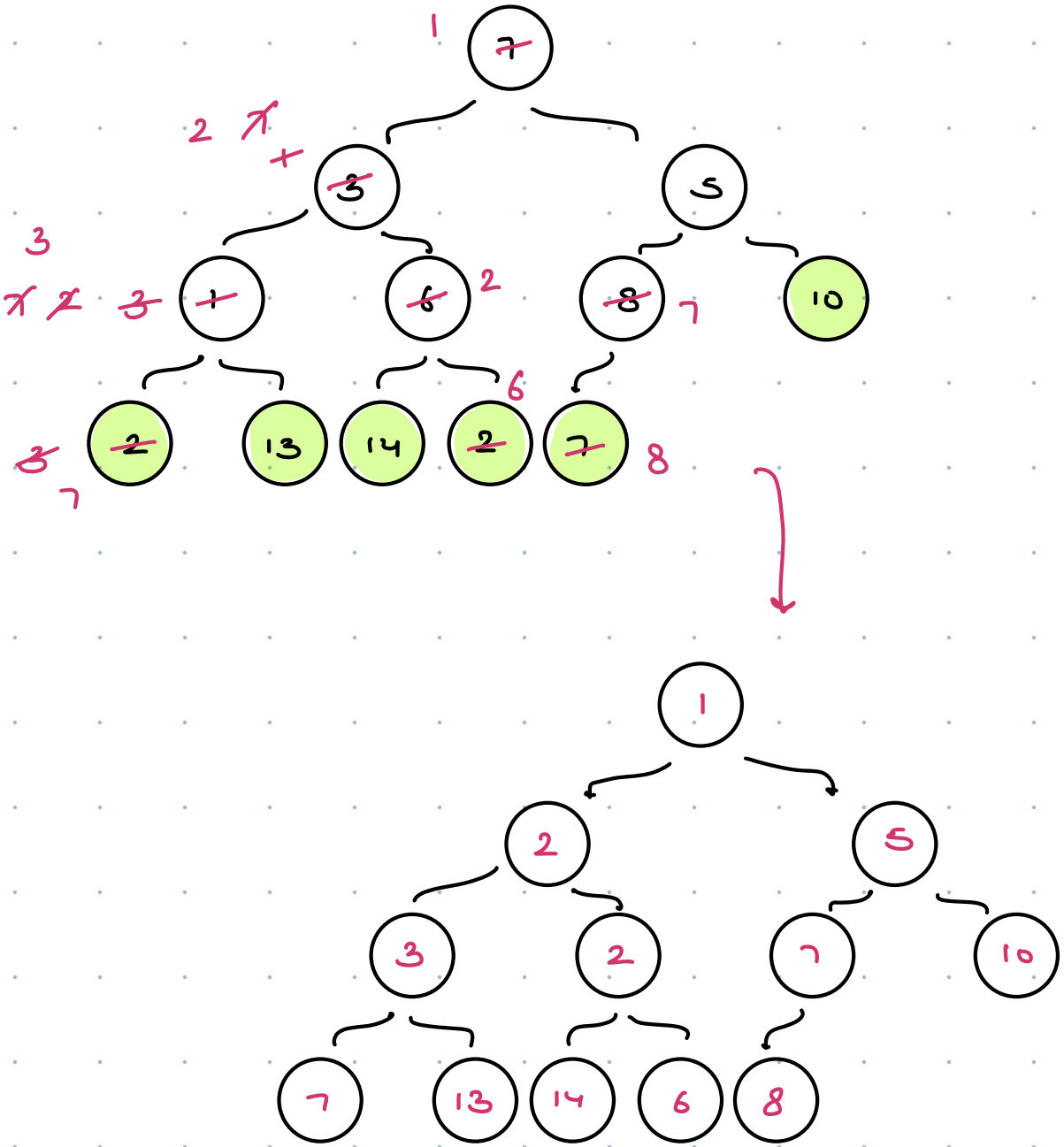
To call upheapify function for every element

TC: O(NlogN)
SC: O(N)

* Idea 3

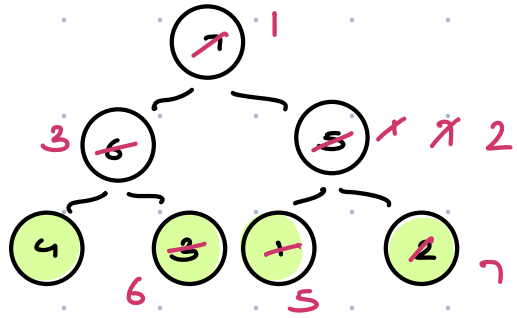
We call downheapify coming from last non leaf node & build the heap in O(n) time.

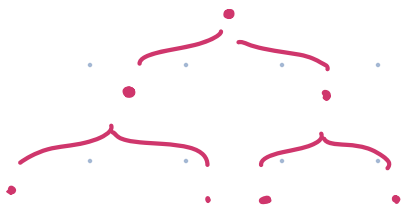
7	3	5	1	6	8	10	2	13	14	2	7


7
3
5
1
6
8
10
2
13
14
2
7

1
2
5
3
2
7
10
7
13
14
6
8

7	6	5	4	3	1	2





→ n/8 → 2 swaps

→ n/4 → 1 swap

→ $\frac{n}{2}$ → 0 swaps

Total swaps $= \frac{n}{2} * 0 + \frac{n}{4} * 1 + \frac{n}{8} * 2 + \dots\dots)$

$= n \left(\frac{0}{2} + \frac{1}{4} + \frac{2}{8} + \frac{3}{16} + \dots \right)$

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

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

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

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

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

a = first term

r = common ratio

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

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

$$\boxed{S = 2}$$

Total swaps $= \frac{n}{2} * 2$

$= n$

TC : O(n)

```
for (i = n/2 - 1; i >= 0; i--) {
    downheapify(heap, i);
}
```

TC : O(N)
SC : O(1)

```
void swap(AL<I> arr, i, j) {
    int temp = arr.get(i);
    arr.set(i, arr.get(j));
    arr.set(j, temp);
}
```

Merge N-sorted arrays

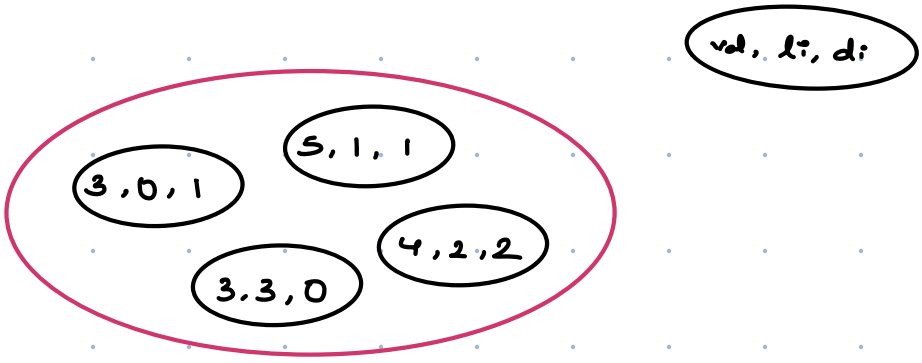
AL<AL<I>>

AL = {
0 → {2, 3, 5, 11, 15, 20} (indices 0 1 2 3 4 5)
1 → {1, 5, 7, 9} (indices 0 1 2 3)
2 → {0, 2, 4} (indices 0 1 2)
3 → {3, 4, 5, 6, 7, 8} (indices 0 1 2 3 4 5)
}

Ans = {0 1 2 2 3 3 4 4 5 5 5 .... }

BF Idea → Make N pointers to keep track of N arraylists & then compare their values again & again.

Idea 2 Use Min priority Queu to get minimum element first



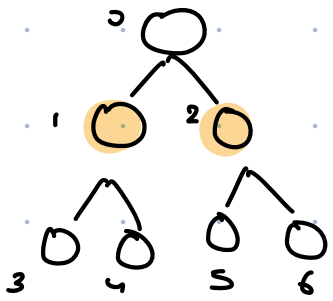
Ans = {0, 1, 2, 2

```
class triplet implements Comparable<I> {
    int val;
    int li;   // list idx
    int di;   // data idx

    triplet ( int v, int l, int d){
        this.val = v
        this.li = l
        this.di = d;
    }

    public int compareTo ( triplet o){
        if (this.val < o.val) return -1;
        else if (this.val > o.val) return 1;
        else return 0;
    }
}
```

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

for ( i=0; i< lists.size(); i++){
    triplet tp = new triplet ( list.get(i).get(0), i , 0)

    pq.add (tp);

}
```

```
AL <I> ans = new AL<>();

while (pq.size() > 0){

    triplet rp = pq.remove();

    ans.add(rp.val);

    if (rp.di + 1 < lists.get(rp.li).size()){

        int v = lists.get(rp.li).get(rp.di+1);

        int l = rp.li

        int d = rp.di + 1;

        pq.add(new triplet(v, l, d);
    }

}
```

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




```
void downheapify ( heap, i, j )
{
    while ( 2i + 1 < j ) {
        x = min( heap.get (i) , heap.get (2i+1); heap.get(2i+2))
        if ( x == heap.get (i) ) return ;
        else if ( x == heap.get (2i+1)) {
            swap ( heap, i, 2i+1)
            i = 2i + 1
        }
        else {
            swap ( heap, i, 2i+2);
            i = 2i+2 ;
        }
    }
}
```