Implementation of Heaps:

- → CBT
- → Heap Intro
- → insert() / deletemin()
- → Heapify() / Heapsort()

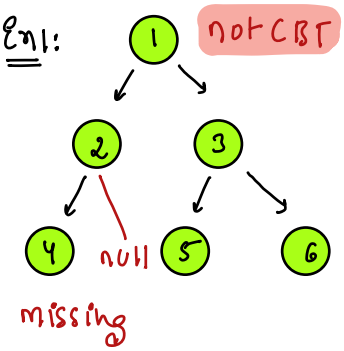
Today's { 1hr → 10mins, 1hr → 10mins, 1hr class → }

Complete binary Tree (CBT) → { Pre-requisite }

A BT is said to be CBT if

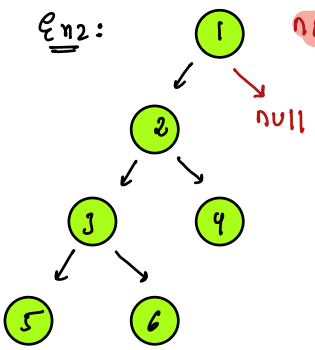
1) All nodes have to be present level by level from left → right
2) All level should be completely filled except last level
   { At last level it can be either completely filled or not }

Eg1:



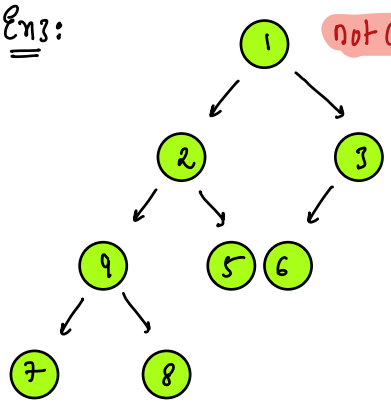
not CBT

Eg2:



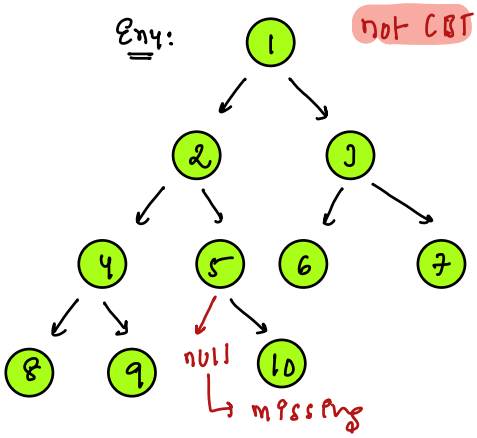
not CBT

Eg3:



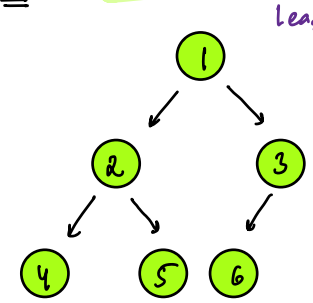
not CBT

Eg4:



not CBT

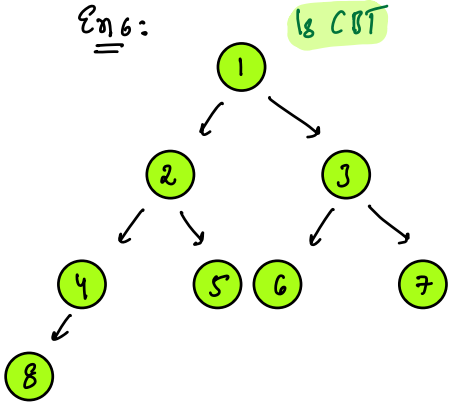
Eg5:



Is CBT

N = 6
leaf = 3

Eg6:



Is CBT

N = 8
leaf = 4

Height of CBT:

Height of CBT	Min Nodes	Max Nodes
1	2 : 2^1	3 : 2^2-1
2	4 : 2^2	7 : 2^3-1
3	8 : 2^3	15 : 2^4-1
4	16 : 2^4	31 : 2^5-1

h ⟶ : min : 2^h max: $2^{h+1}-1$

// Say N Nodes, height of CBT = h ≈ log N

min Nodes : $2^h = N$

$h = \log_2 N$

max Nodes : $2^{h+1}-1 = N$

$2^{h+1} = N+1$

Apply log on both sides

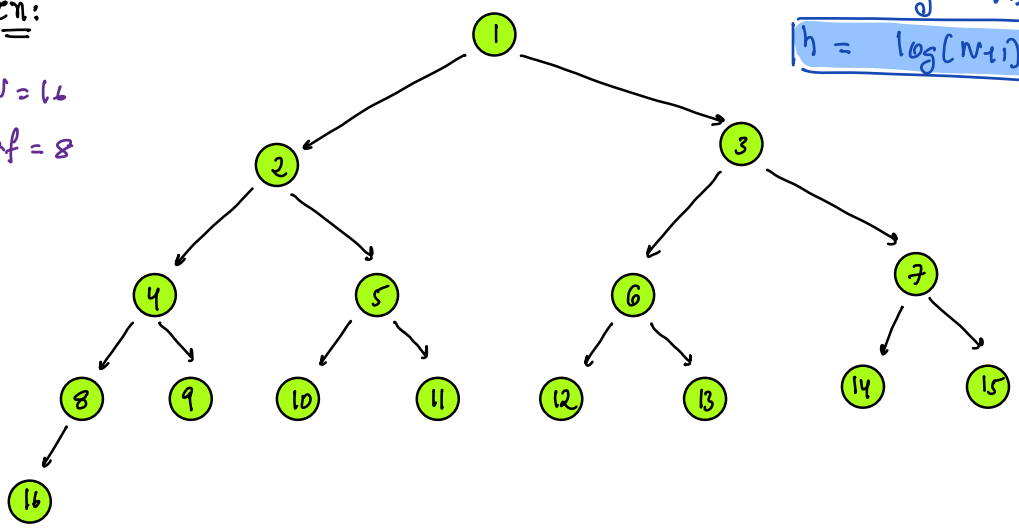
$h+1 = \log(N+1)$

$h = \log(N+1) - 1$

Eg:

N = 16

leaf = 8



Implementation of CBT, using nodes → {It's possible}

```
class Node {
    int data
    Node left
    Node right
    Node(n) {
        data = n
        left = right = null
    }
}
```

Implementation of CBT using arrays / arraylist / dynamic array

insert: 3 2 9 6 8 7 2 14 16

insert all in list ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓

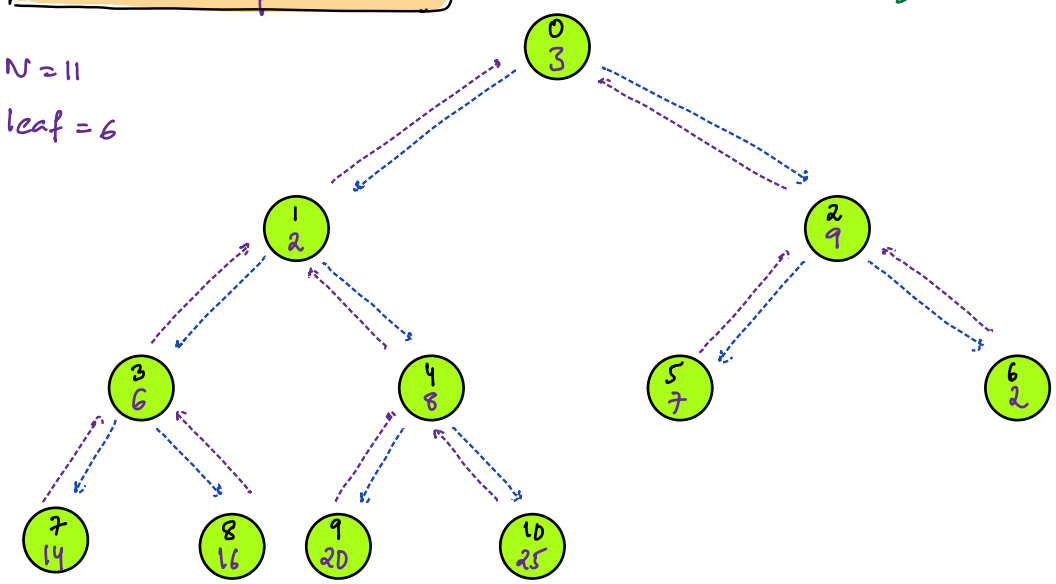
	0	1	2	3	4	5	6	7	8	9	10
li :	3	2	9	6	8	7	2	14	16	20	25

Below Tree is Representation

11 : 4B : 44B

N = 11

leaf = 6



parent :	left	right
0	1	2
1	3	4
2	5	6
3	7	8
4	9	10

Obs:

parent(i) → left : 2i+1

parent(i) → right : 2i+2

child(i) → parent : $\frac{i-1}{2}$

i = 9 ⟶ parent: $\frac{9-1}{2} = 4$

i = 8 ⟶ parent: $\frac{8-1}{2} = 3$

i = 6 ⟶ parent: $\frac{6-1}{2} = 2$

Advantage:

: 2 way traversal

: Space saving when compared Tree based n Nodes

Heaps:

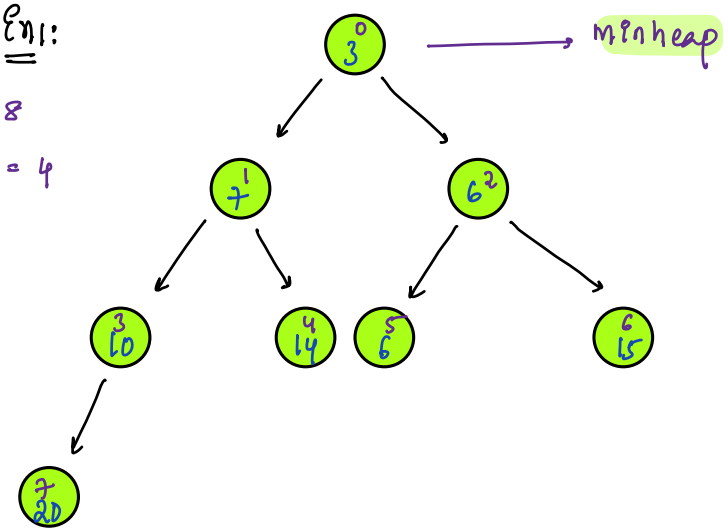
- BT should be a CBT fist
- Every node >= child nodes : Max heap
- Every node <= child nodes : Min heap

Note:

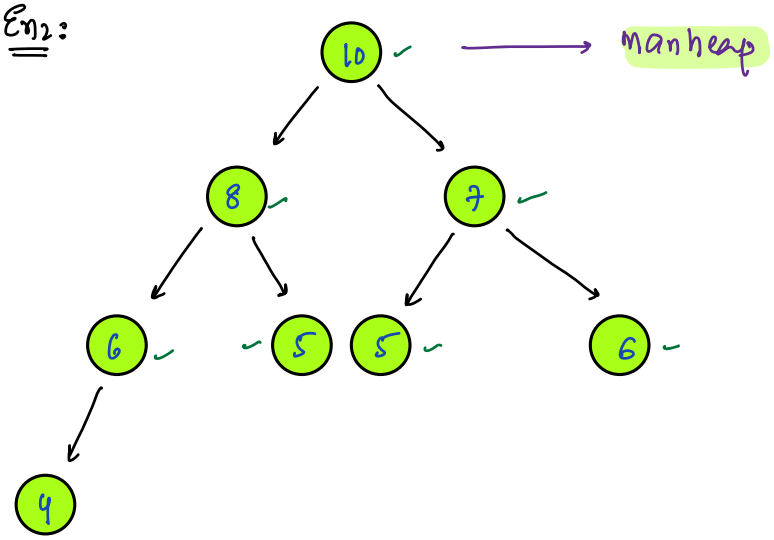
Ex1:

N = 8

leaf = 4



Ex2:



Min heap Operations:

- → insert() : $\log N$
- → getMin() : $O(1)$
- → deleteMin() : $\log N$
- → size() : $O(1)$

→ search() : $O(N)$, not useful

→ delete() : $O(N)$, not useful

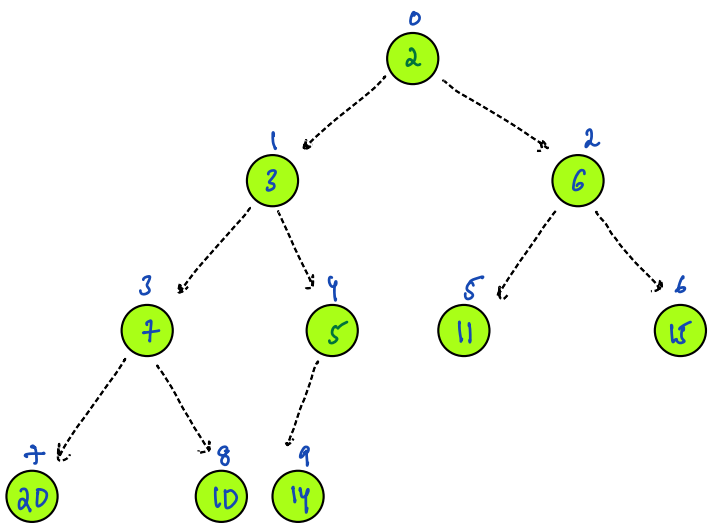
→ search() & delete

// Given min heap:

	0	1	2	3	4	5	6	7	8	9
list<int> ar:	2	3	6	7	5	11	15	20	10	14

N = 10

leaf = 5



insert(5):

ind	parent: (ind-1)/2	Ideally: ar[par] <= a[ind]
8	(8-1)/2 : 3	ar[3] <= ar[8] * swap (ar[3] ↔ ar[8])
3	(3-1)/2 : 1	ar[1] <= ar[3] * swap (ar[1] ↔ ar[3])
1	(1-1)/2 : 0	ar[0] <= ar[1] ✓ Break

insert(2):

ind	parent: (ind-1)/2	Ideally: ar[par] <= a[ind]
9	(9-1)/2 : 4	ar[4] <= ar[9] * swap (ar[4] ↔ ar[9])
4	(4-1)/2 : 1	ar[1] <= ar[4] * swap (ar[1] ↔ ar[4])
1	(1-1)/2 : 0	ar[0] <= ar[1] * swap (ar[0] ↔ ar[1])

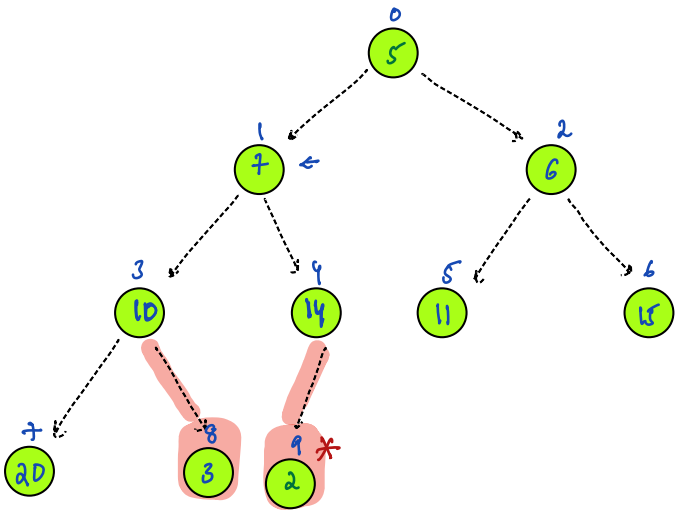
0 ⟶ no parent : { Break }

```
                                    min heap    ele to insert
list<int> minheap_insert( list<int> ar, int n) {   TC: O(logN)
                                                   SC: O(1)
    int ind = ar.size()
    ar.add(n) // In a list new ele is added at last index
    int par = (ind-1)/2
    while( ind > 0 && ar[par] > ar[ind] ) {   // ideally minheap
                                              ar[par] <= ar[ind]
                                              since condition fails we swap.
        swap ar[par] & ar[ind] // new ele is going to par
        ind = par
        par = (ind-1)/2 // updating par for index
    }
    return ar;
}
                                    min heap
list<int> minheap_delete ( list<int> ar) {   TC: O(logN) SC: O(1)
    int n = ar.size()
    swap (ar[0] & ar[n-1]) // first & last index ele
    ar.delete() // Delete last index element
    n = n-1 // size decrease by 1
    int i=0
    while (i<n) {
        int min_ind;
        int l = 2i+1, r = 2i+2 // i -> left: 2i+1, right: 2i+2
        if(l>=n) { //no leftchild break }
        if( l < n ){ // left exists
            min_ind = l
        }
        if( r<n && ar[r] < ar[e]){ //right exist & itf min
            min_ind = r
        }
        if (ar[i] <= ar[min_ind]) { break }   //already satisfying
        else{ swap ar[i] <-> ar[min_ind] ; i = min_ind }
    }
}
```

delete_min:

// Given min heap:

	0	1	2	3	4	5	6	7	8 *	9 *
list < int > ar:	5	7	6	10	14	11	15	20	3	2



Delete: Copy last element to 0th ind & delete last element

ind	left: (2i+1)	right (2i+2)	min_ind (ind with min value)	ar[ind] > ar[min_ind]
0	1	2	1	swap (ar[0] ↔ ar[1])
1	3	4	4	swap (ar[1] ↔ ar[4])
4	no children			break

Delete: Copy last element to 0th ind & delete last element

ind	left: (2i+1)	right (2i+2)	min_ind	ar[ind] > ar[min_ind]
0	1	2	1	swap (ar[0] ↔ ar[1])
1	3	4	3	swap (ar[1] ↔ ar[3])
3	7	=	7	if (ar[ind] <= ar[min_ind]) : no swap, break it

// Heapify:

1) Given a list of ints → convert into → min heap / max heap

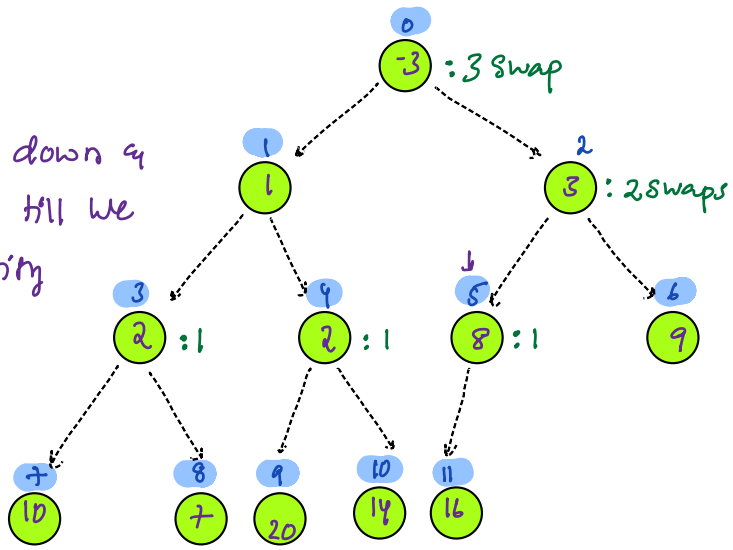
min_heapify max_heapify

	0	1	2	3	4	5	6	7	8	9	10	11
ar[12] = {	16	2	8	10	14	3	9	2	7	20	1	-3 }

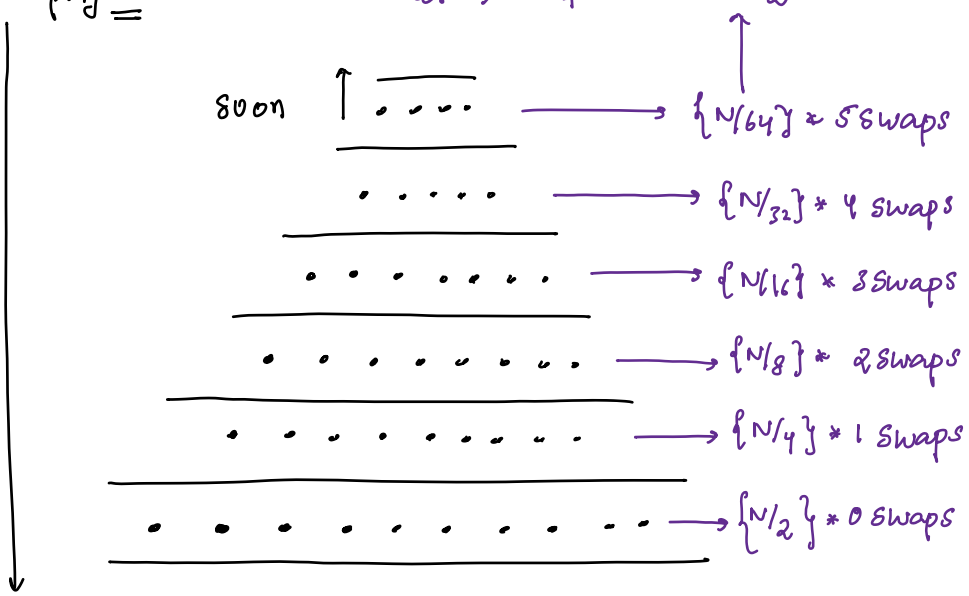
1. sort ar[] : nlogn

2. Heapify : O(N)

At every node going down & swapping with child till we reach leaf or we satisfy min heap condition



Heapify TC: Assume N Nodes ⇒ leaf Nodes = N/2



TC: $S = \frac{N}{4} + \frac{2N}{8} + \frac{3N}{16} + \frac{4N}{32} + \frac{5N}{64} + \dots$

TC: O(N)

TC: $S = \frac{N}{4} + \frac{2N}{8} + \frac{3N}{16} + \frac{4N}{32} + \frac{5N}{64} + \dots\dots$

Multiply & minus steps

$2S = \frac{N}{2} + \frac{2N}{4} + \frac{3N}{8} + \frac{4N}{16} + \frac{5N}{32} + \dots\dots$

$-$

$S = \frac{N}{4} + \frac{2N}{8} + \frac{3N}{16} + \frac{4N}{32} + \frac{5N}{64} + \dots\dots$

$S = N/2 + N/4 + N/8 + N/16 + N/32 + N/64 \dots\dots$

$S = N\{ 1/2 + 1/4 + 1/8 + 1/16 + 1/32 + \dots\dots \}$

Sum of gp ≈ 1

$S = N \times 1 \Rightarrow O(N)$

list<int> ar min_heapify (list<int> ar) TC: O(N) SC: O(1)

```
int n = ar.size()
k = n-1 ; k >= 0; k --) {
    int i = k;
    while (i < n) {
        int min_ind;
        int l = 2i+1, r = 2i+2 // i → left: 2i+1, right: 2i+2
        if (l >= n) { //no left child break }
        if ( l < n ) { // left exists
            min_ind = l
        }
        if ( r < n  && ar[r] < ar[l]) { //right exist & it's min
            min_ind = r
        }
        if (ar[i] <= ar[min_ind]) { break } //already satisfying
        else{ swap ar[i] ↔ ar[min_ind] ; i = min_ind }
    } }
} return ar;
```

Heap Sort:

a Given ar[] —min_heapify→ min_heap —n-1 del→ dec order

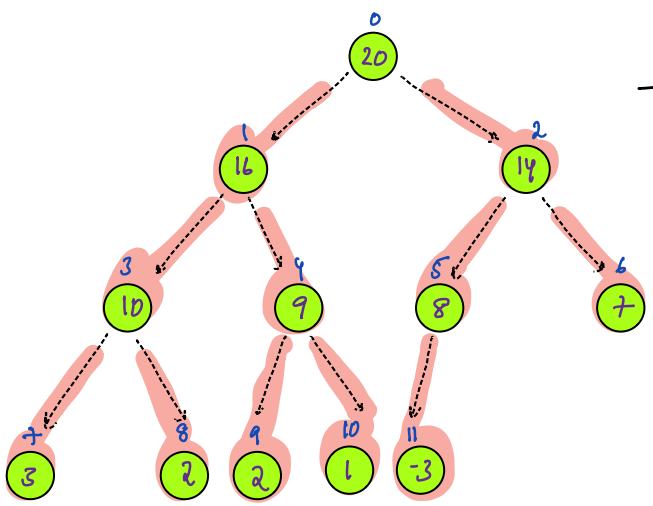
TC: $O(N) + (n-1)\log N \Rightarrow O(N \log N)$ SC: $O(1)$

b Given ar[] —max_heap→ max-heap —n-1 del→ inc order

	0	1	2	3	4	5	6	7	8	9	10	11
ar[12] = {	16	2	8	10	14	3	9	2	7	20	1	-3 }

min heap ↓

	0	1	2	3	4	5	6	7	8	9	10	11
ar[12] = {	-3	1	3	2	2	8	9	10	7	20	14	16 }



delete:

N: ar[0] & ar[N-1] N = N-1

12 : ar[0] & ar[11] 11 : swap from root
11 : ar[0] & ar[10] 10 : swap from root
10 : ar[0] & ar[9] 9 : swap from root
9 : ar[0] & ar[8] 8 : swap from root
8 : ar[0] & ar[7] 7 : swap from root
7 : ar[0] & ar[6] 6 : swap from root
6 : ar[0] & ar[5] 5 : swap from root
5 : ar[0] & ar[4] 4 : swap from root
4 : ar[0] & ar[3] 3 : swap from root
3 : ar[0] & ar[2] 2 : swap from root
2 : ar[0] & ar[1] 1 : swap from root
1 : Break

After deletions & modifications on ar[]

	0	1	2	3	4	5	6	7	8	9	10	11
ar[12] : {	20	16	14	10	9	8	7	3	2	2	1	-3 }

```
List<int> heapSort(List<int> ar){   TC: NlogN  SC: O(1)

    // Convert ar() → min heap
    ar = minHeapify(ar) // return min heap of ar
    int n = ar.length()
    while(N > 1){
        swap(ar[0] & ar[n-1])
        n = n-1 //assume last ele deleted, reduce size
        i = 0 // swap from root to leaf, till minheap condition satisfies
        while(i < n){
            int min_ind;
            int l = 2i+1, r = 2i+2 // i → left: 2i+1, right: 2i+2
            if(l >= n){ //no left child break }
            if( l < n ){ // left exists
                min_ind = l
            }
            if( r < n  &&  ar[r] < ar[l]){ //right exist & it's min
                min_ind = r
            }
            if(ar[i] <= ar[min_ind]){ break }   → //already satisfying
            else{ swap ar[i] ↔ ar[min_ind] ; i = min_ind }
        }
    }
    return ar
}
```