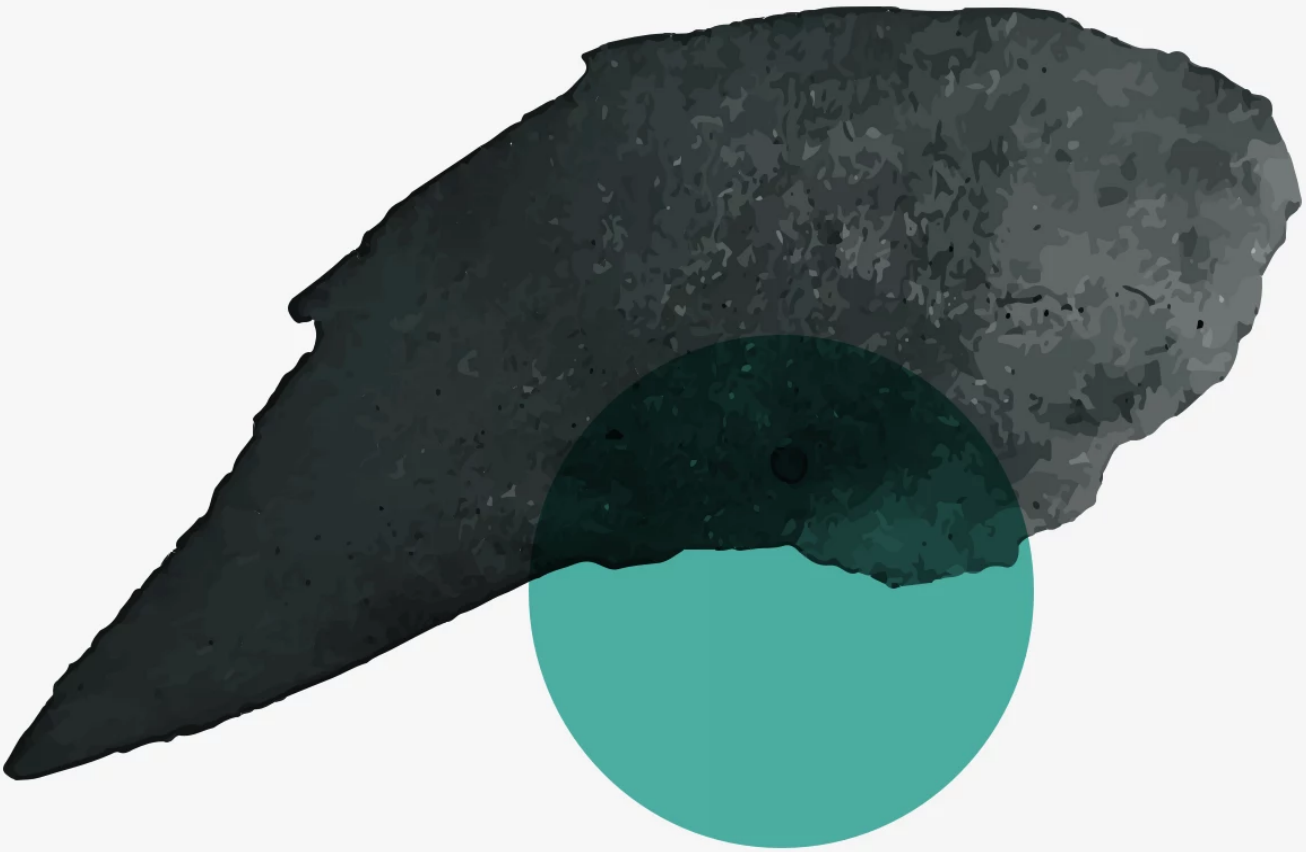


Introduction to Heap Data Structure

* Finding smallest, largest or any query type conditions in data structures heap method is used. It gives the result in constant time (i.e O(1)).

→ For example finding the smallest element in an array or largest element in an array etc. It can be done in O(1) time

→ O(N) element find smallest number

3	8	4	19	20	2	36	...

→ First sort the element while entering the number in the array smallest would be in 0th index and largest would be at last index

2	3	4	8	19	20	36

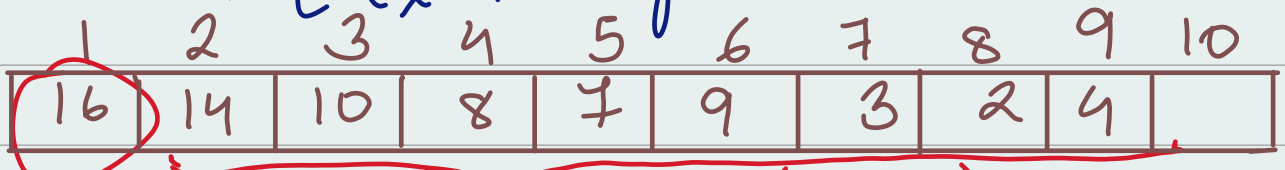
→ O(1)

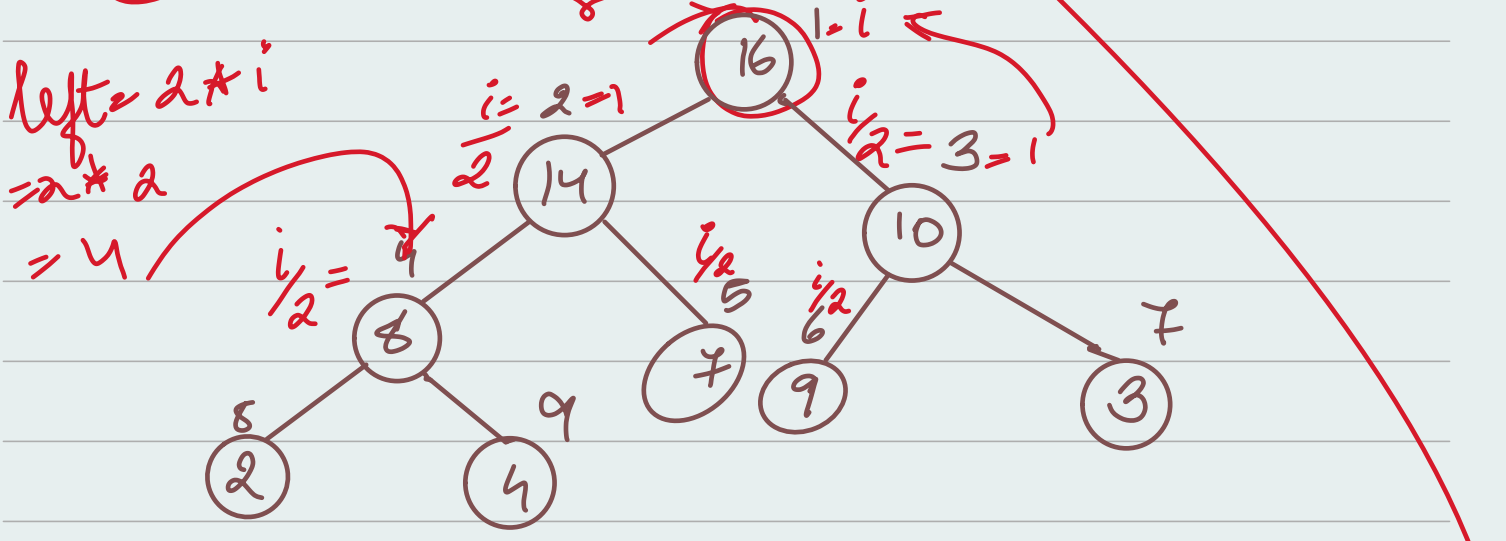
→ But here is a problem if we want to insert an element in an array it will take an average of O(N log N) to insert one element. Can it be reduced to less time complexity ?
Yes, answere is heaps

→ Given a condition on collection of(n) items The result will be in constant times. Heap data structure is used Inserting an element would take log(N)

→ Heaps are stored internally as an array only But represented as a tree

* Max Heap :-





① Complete Binary Tree
② Every Node value ≥ All of its children

may not be Sorted

root ⇒ i = 1
parent (i) = i/2
left (i) = 2 * i
right (i) = 2 * i + 1

no pointers required.

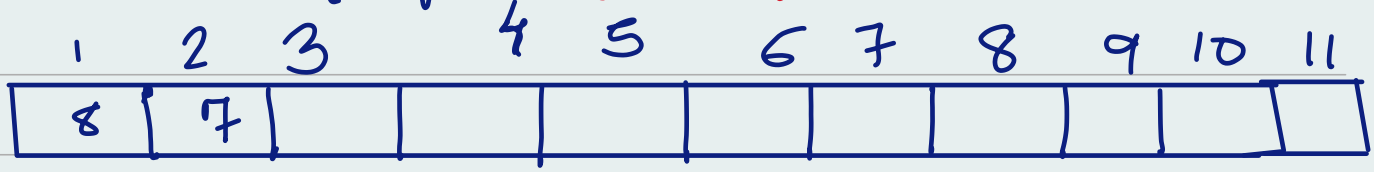
height of the binary tree = log (N)

Because it does not have node left or node right.

we will use this formula to get into every node

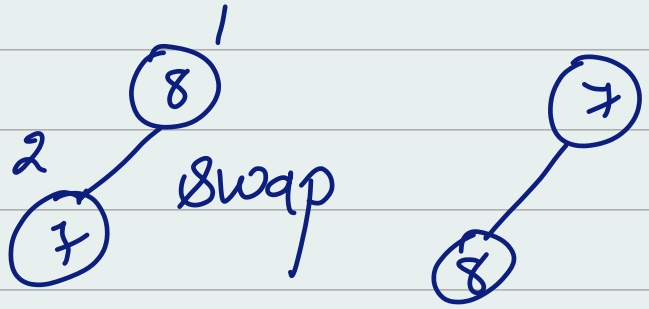
→ Suppose we want to insert 37 at very first index and remove 16 from there.
→ How to insert and how to delete

Min Heap : Insert



left = i/2 i = 2

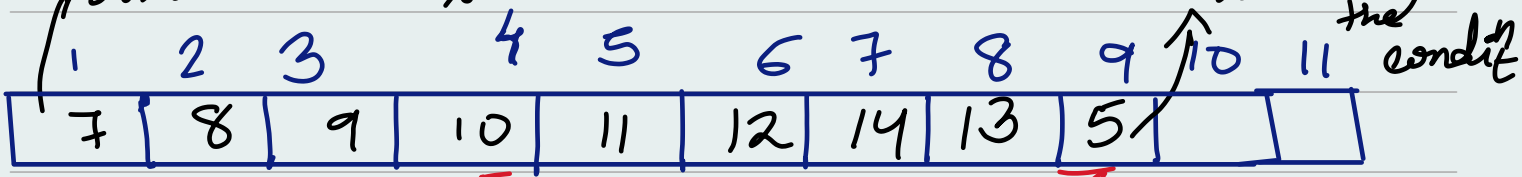
= 2/2 = 1

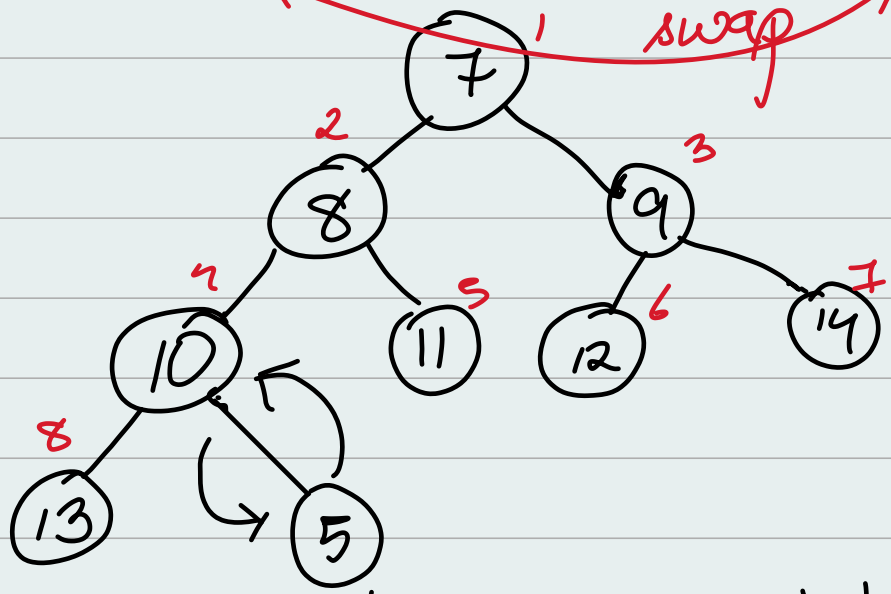


Here val (node) ≤ all of its children

7 ≤ 8

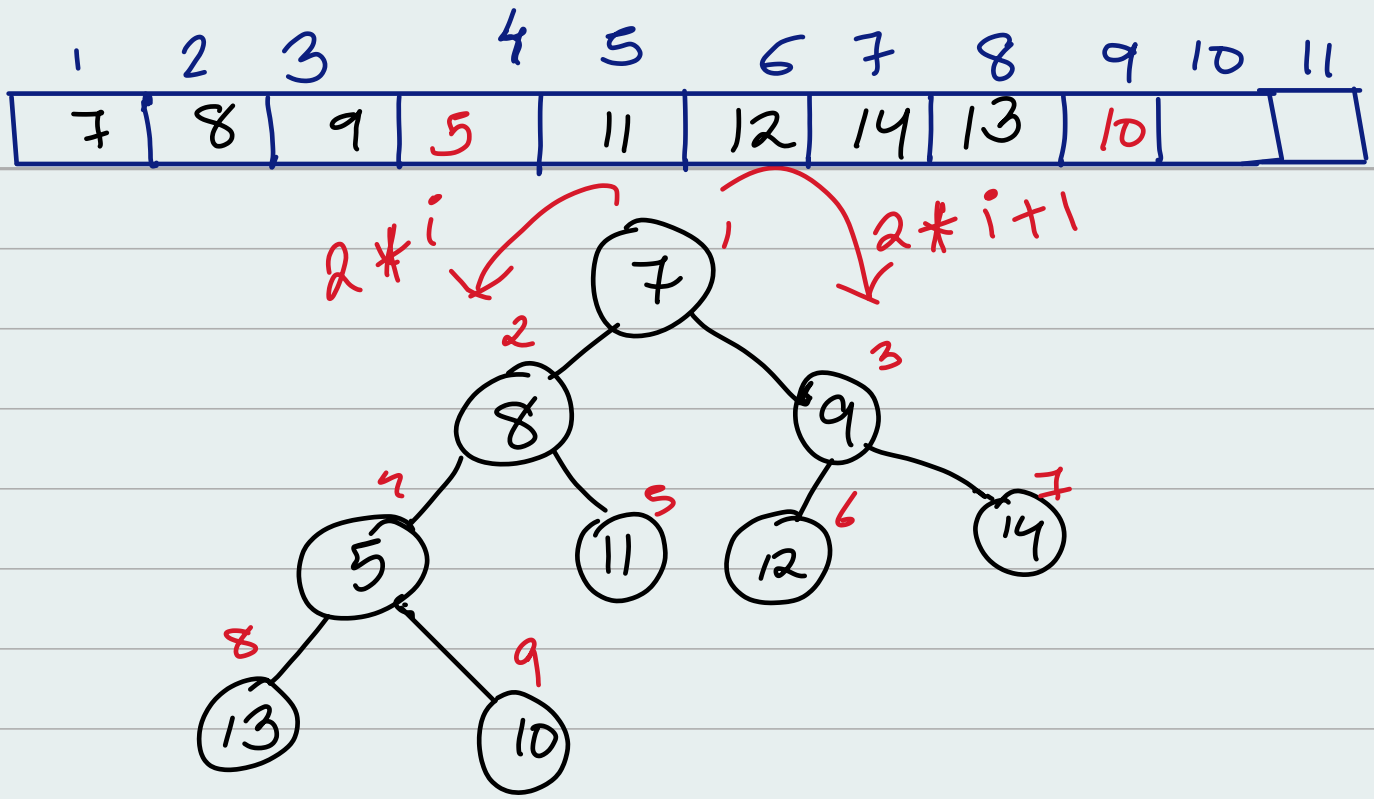
First item should be minimum





↳ condition violated

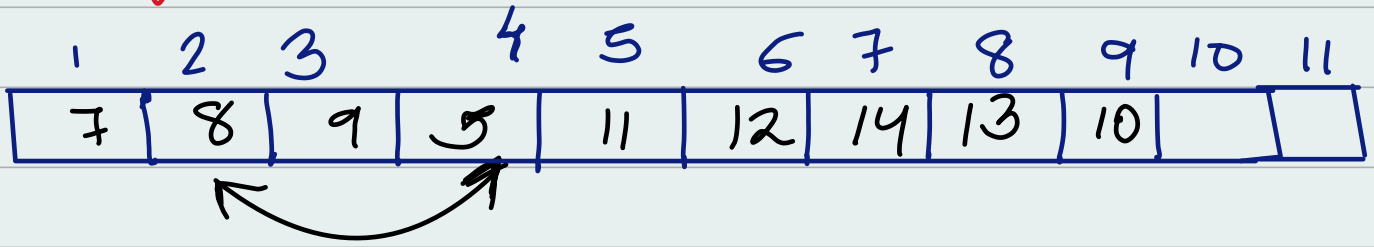
parent of (5) greater than 5, swap



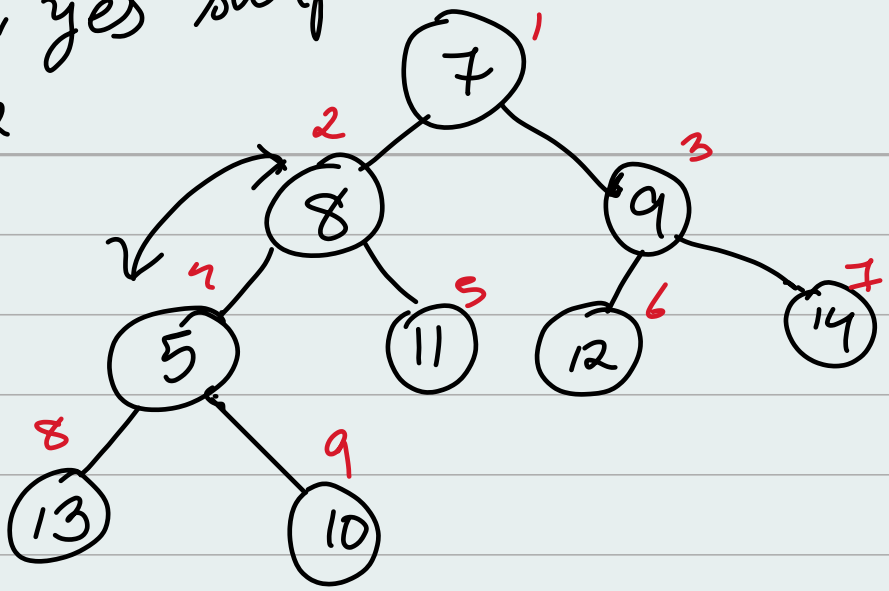
→ we don't have to check whether 5 is greater than 13 or left side of the node. Cause value of the node is ≤ all of its children node.

For 10,
13 > 10 and 5 < 10 then obviously 5 < 13 Hence no need to check.

// keep checking until it not breaking any laws.

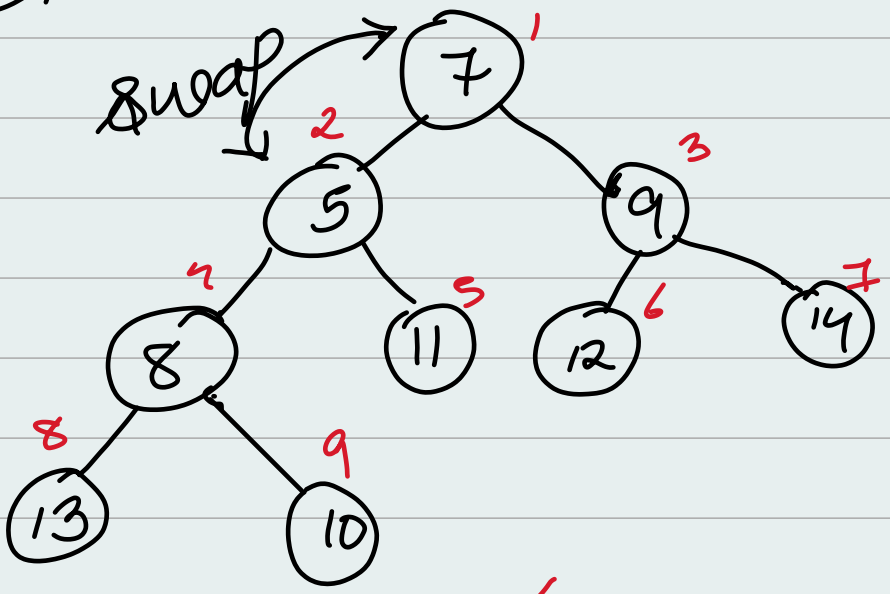


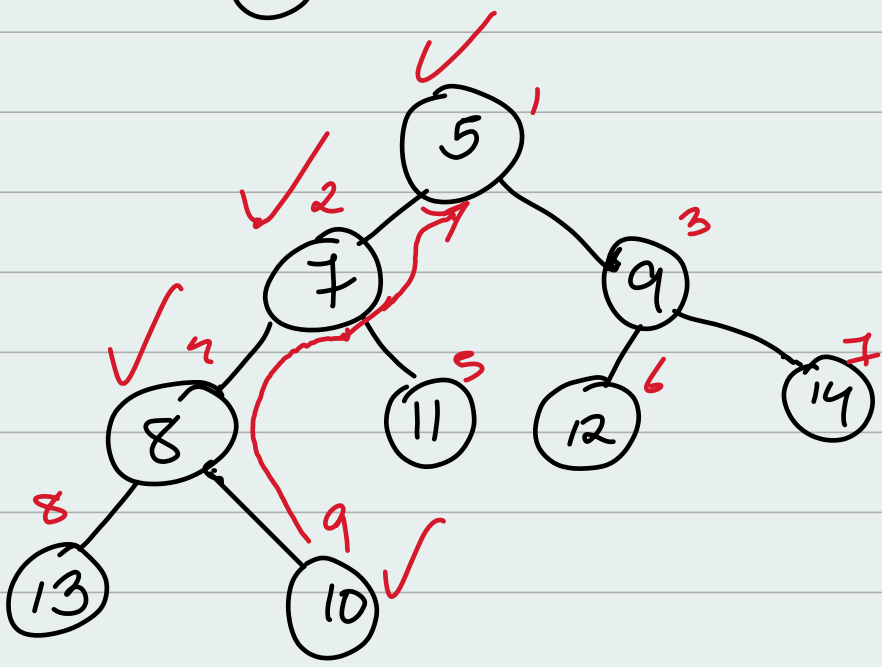
is 8 > 5, yes swap
left = i/2



1	2	3	4	5	6	7	8	9	10	11
7	5	9	8	11	12	14	13	10		

swap



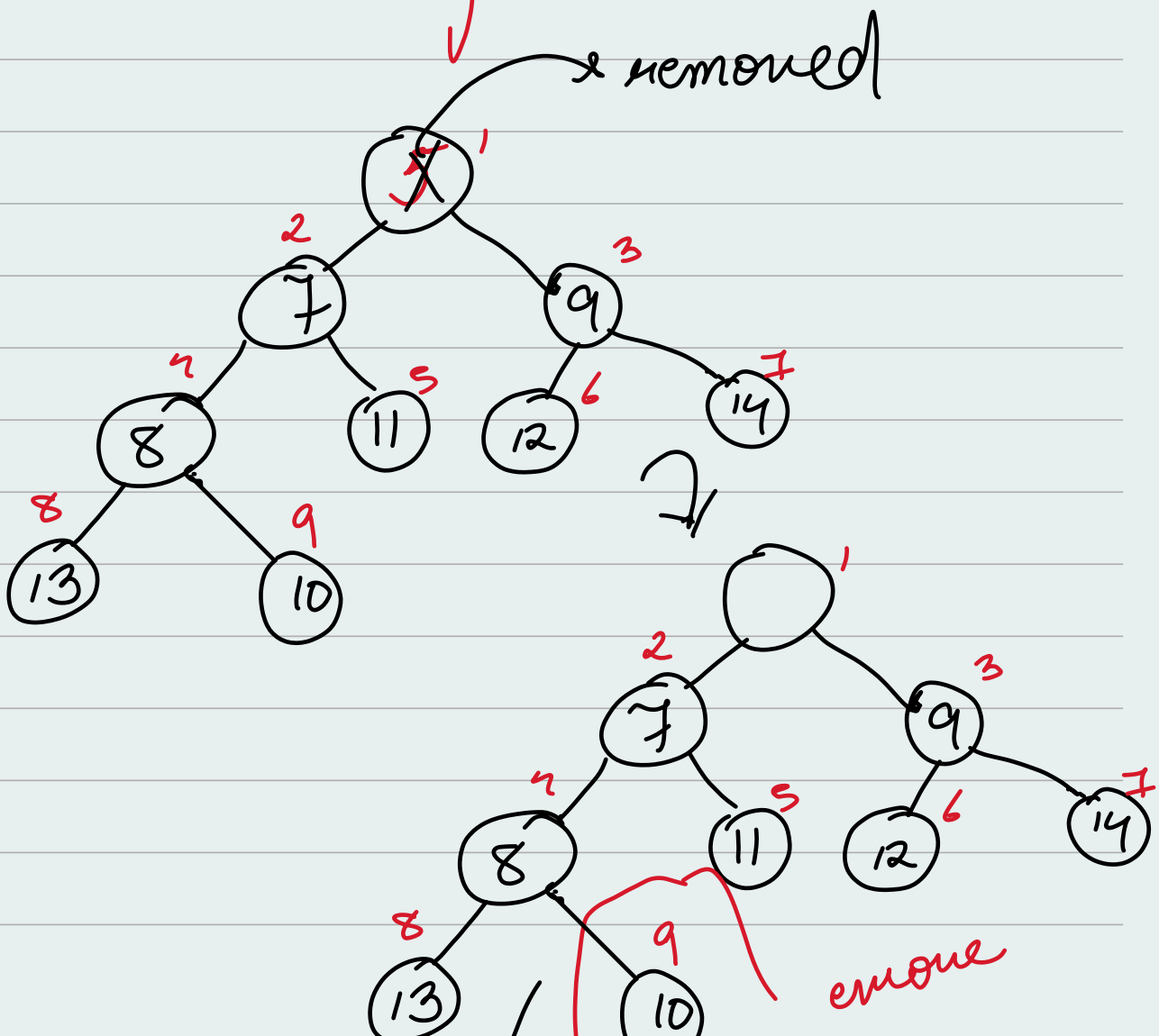


1	2	3	4	5	6	7	8	9	10
5	7	9	8	11	12	14	13	10	

→ This is know as up heap method.

→ Checking once every level. Total height is log(N) so inserting is log(N).

→ Removing the first item in the min heap ③



Rebalancing the tree by putting the next smallest item in the min heap tree

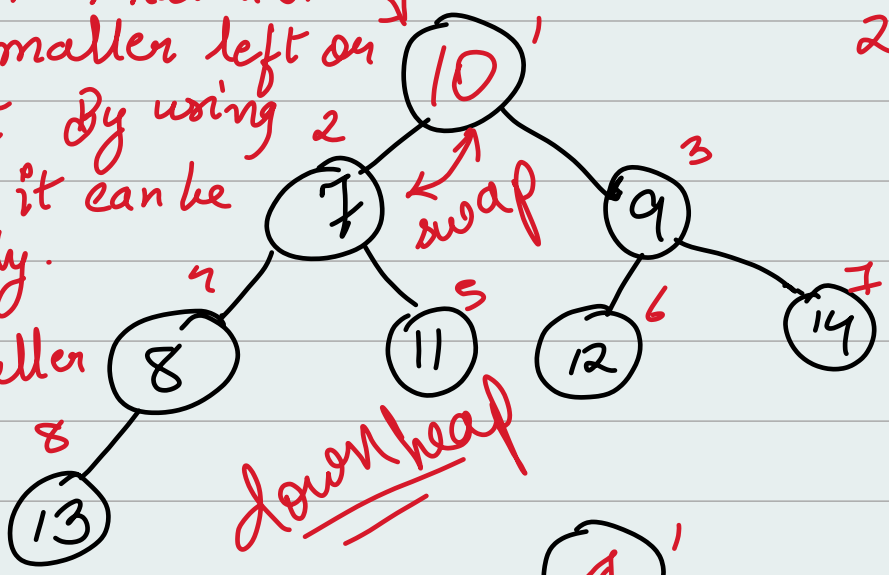
→ Take the last element (i.e 10 in the above array) and put it in the first index coz it can be done easily

1	2	3	4	5	6	7	8	9	10	11
10	7	9	8	11	12	14	13			

check what item is smaller left or right By using formula it can be done easily.

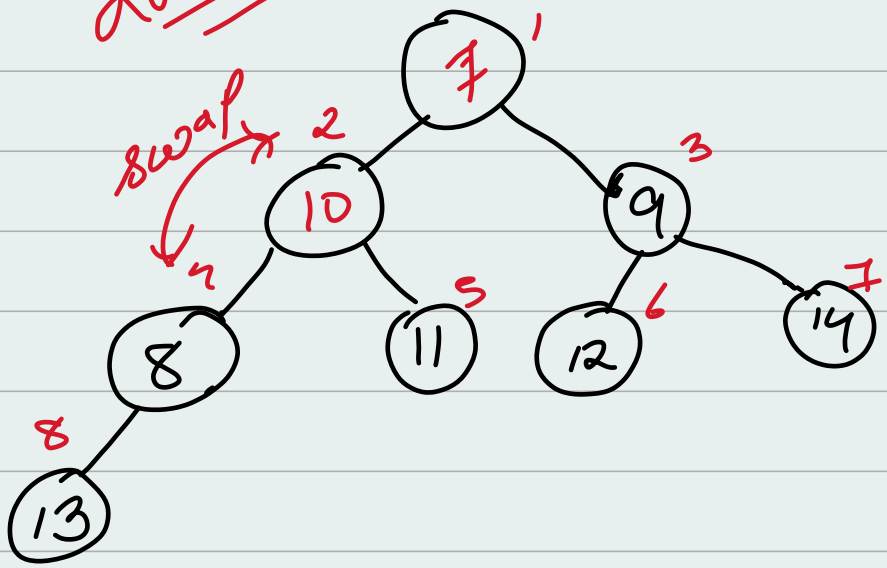
$2 * i$ = left

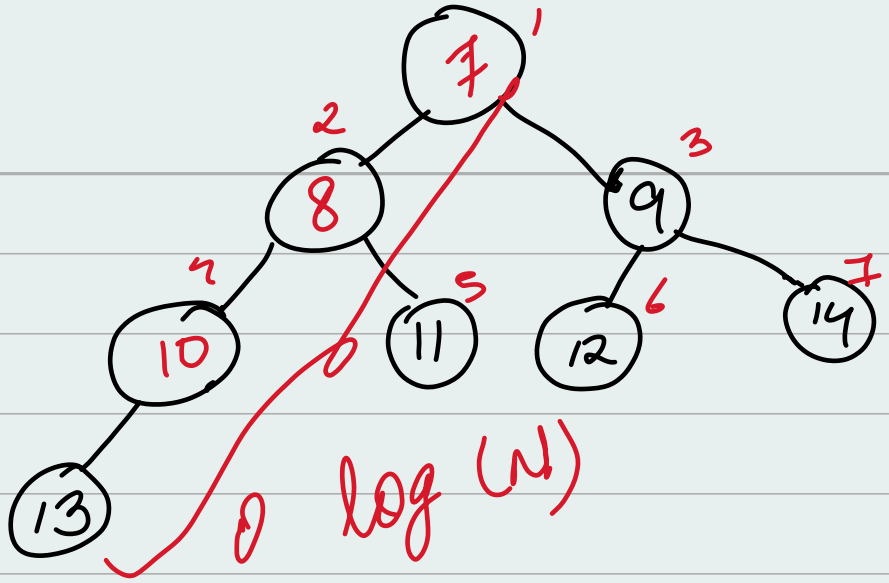
7 is smaller the left side So swap

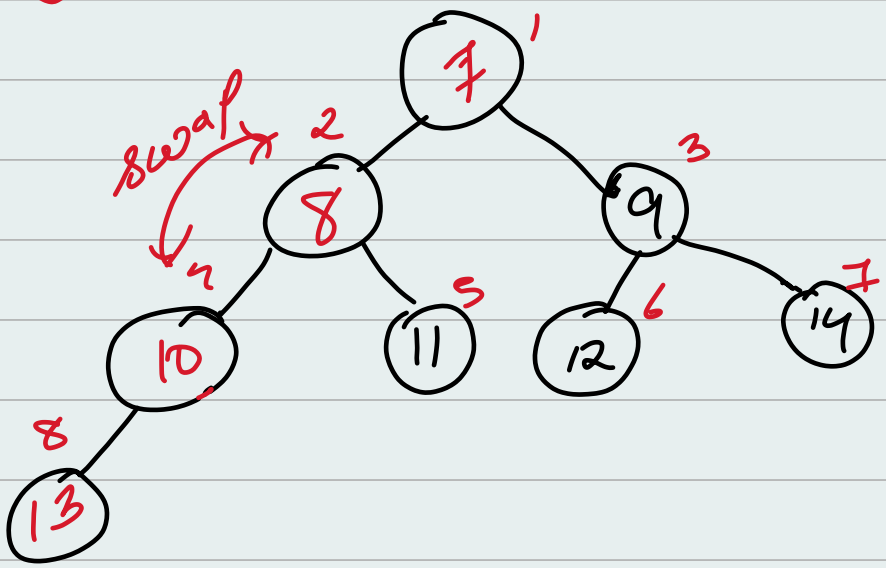


swap

down heap







* Priority Queue using Linked List

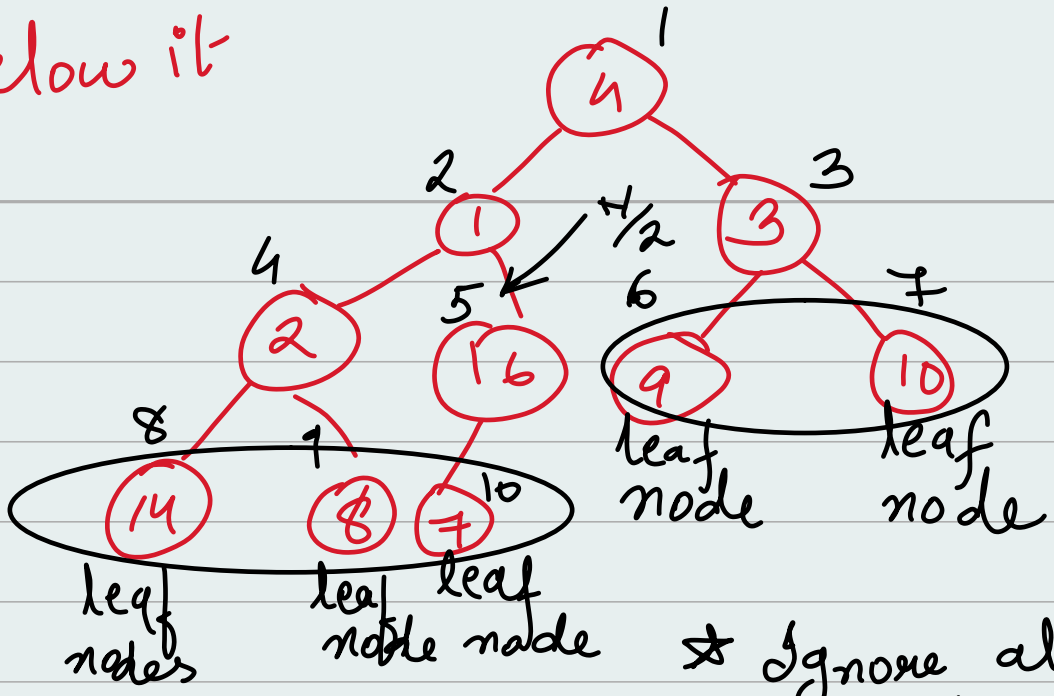
* given an unsorted array create a max heap from that

1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7

→ Make it into a binary tree.

→ Max heap property: every node should be greater than equal to all the children

below it



* Ignore all the leaf nodes

Why start from $N/2$?

$\left[\frac{N}{2} + 1 \ldots N\right]$ → leaf nodes

→ We are skipping all the leaf nodes 6, 7, 8, 9, 10 index elements because every element's down 14, 8, 7, 9, 10 will all be greater than children nodes

→ for (index = $\frac{N}{2}$ till 1) → {1, 2, 3, 4, 5}

do downheap (index)

→ outer loop running $\frac{N}{2}$ times

and downheap is log (N)

but time complexities N log (N)

Because, for all the nodes that are one above level leaf nodes For leaf nodes it's taking constant time complexities

leaf nodes $= \frac{N}{2}$ = 0 times comparison

1 level above leaf nodes $= \frac{\frac{N}{2}}{2} = \frac{N}{4} = C$

level 2 $= \frac{N}{4/2} = \frac{N}{8} = 2C$

level 3 $= \frac{N}{16} = 3C$

:

root $= 1 = \log N \times (C)$

Total time $= \frac{N}{4} * C + \frac{N}{8}(2C) + \frac{N}{16} \times 3(C)$

$+ \ldots + 1 (C * \log N)$

$\frac{N}{4} = 2^k$

$$= C \cdot 2^k \underbrace{\left(\frac{1}{2^0} + \frac{2}{2^1} + \frac{3}{2^2} + \frac{4}{2^3} + \cdots + \frac{k+1}{2^k} \right)}$$

Bounded by a constant.

Sums of ix^i to $\infty = \frac{x}{(1-x)^2}$

Sums in mathematics

$$\boxed{O(2^k) = O(N)}$$

Code :-

```
public class Heap <T extends Comparable<T>>{

    private ArrayList<T> list; // list to
                                  store the heap.

    public Heap (){ // constructor to initialize
list = new Arraylist<>(); the list
}

private void swap (int first, int second ){
    T temp = list.get (first),
    list.set (first, list.get (second));
    list.set (second, temp );
```

```
}

private int parent (int index){
    return (index - 1) / 2;

private int left (int index){
    return index * 2 + 1;

private int right (int index){
    return index * 2 + 2;

}

public void insert (T value){
    list add (value);
    upHeap ( list size() - 1);
}
private void upHeap (int index){
    if (index == 0){
        return;
    }
    int p = parent (index);
    if (list get(index) compareTo(list.
                                get(p)) < 0){
        swap (index, p);
        upHeap (p);
    }
}
```

```
public T remove () throws Exception {
    if (list isEmpty ()){
        throw new Exception ("Removing
            from an Empty Heap!");
    }
    T temp = list get (0);
    T last = list. remove (list. size()
                -1);

    if (! list. is empty ()){
        list. set (0, last);
        downHeap (0);
    }

    return temp;
}

private void downHeap (int index){
    int min = index,
    int left = left (index);
    int right = right (index);
```

```
        if (left < list.size() && list.get
(min).compareTo(list.get(left))<0){
            min = left;
        }

        if (right < list.size() && list.get
(min).compareTo(list.get(right))<0){
            min = right;
        }

        if (min != index){
            swap(min, index);
            downHeap(min);
        }
    }
    public ArrayList<T> heapSort()
        throws Exception {
        ArrayList<T> data = new
                    ArrayList<>();
```

```
while (! list. isEmpty() ){
    data. add (this. remone);
}
return data;
}
}
```