Date 28/07/24
Page

* HEAPS

- Why? Let's say find the smallest no. → | 3 | 8 | 4 | 19 | 20 | 2 | 36 |

O(N) gives you smallest no.

✓ you can reduce it to O(1) by inserting by sorted → | 2 | 3 | 4 | 8 | 19 | 20 |

O(1) ∵ min. at 0th index

✓ But it takes O(N log N) to insert item

How to reduce it?

using

Heaps

✓ Heaps are generally stored in arrays but represented as Trees.

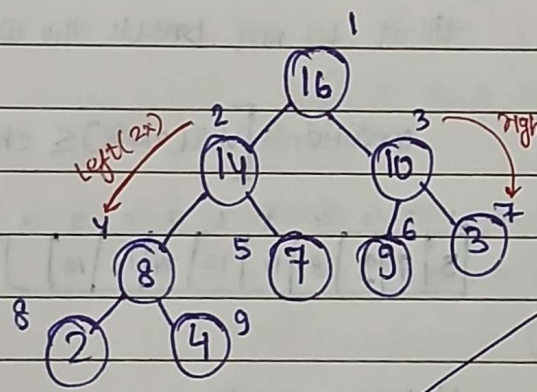
- **How Heaps work?**

Level order Traversal

may not be sorted

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

↳ Heaps stored in array



How it is represented?

① complete tree

② Every node value ≥ All of it's children

★ Two imp. conditions in minHeap Data structures

root → i = 1

for complete binary Tree

- Parent of (i) → i/2 eg. (14) → index(2) → parent of → 4 ∴ 4/2 = 2, 5 ∴ 5/2 = [2.5] = 2
- Left of (i) → 2 × i
- Right of (i) → 2 × i + 1

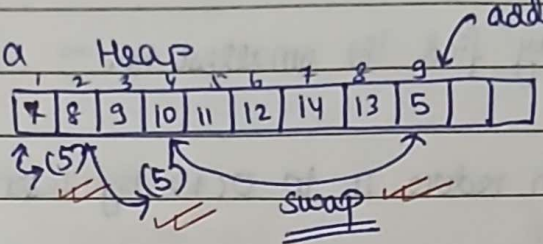
eg. Parent of 7 → 7/2 = 3.5 = 3 → (10) ✗ (3)

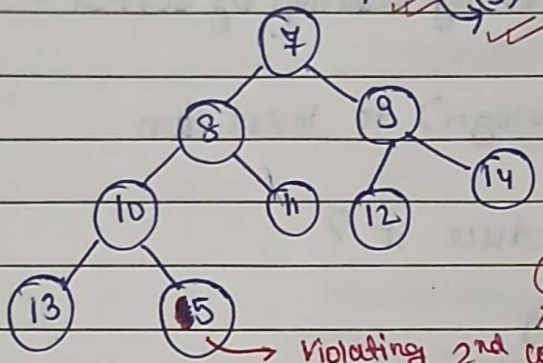
∴ No pointers required because of this formula.

[height = log(N)]

- How to solve problem of delete & insert?

→ Inserting items in a Heap

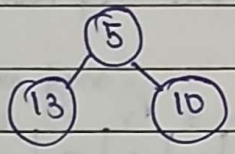




(no children < parent)

violating 2nd cond^{n} of Heap ∴ swap

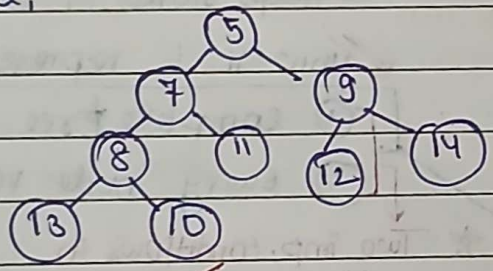
(no need to check with 13) ∵ if 13>10, 5<10, then obviously 5<13

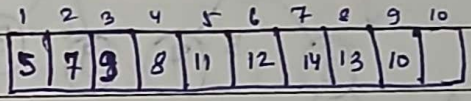


Illy, for 5 & 7 then 5 & 7 ... loop on checking till it not break the laws.

Hence,

condition → [val(node) ≤ childrens]





This is known as UpHeap method

Since, we're checking one at every level & levels are log n

∴ complexity → [O(log(N))]

→ Removing items

→ from here, if we remove 5 (remove from 1st index) ★

∴ put the element which is last of index (there it is 10) → put on 1st index in place of 5.

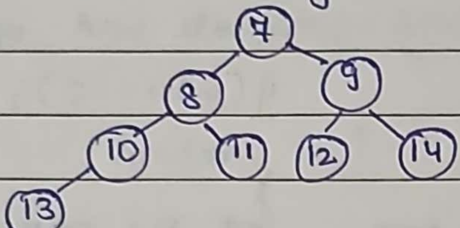
arr[i] = 10;

(updated)

Date
Page

→ Now check if left is smaller or right is smaller
→ left ∵ 7 ∴ swap with 7 (automatically adjusted with 9 → obvious)
→ Now for 10 again check left or right which is smaller
→ left ∵ 8 ∴ swap with 8
→ ~~Ag~~ Again now 13 (larger) ∴ stop.

Hence →



∵ checking each level wise
∴ complexity ⇒ O(log N)

[revise Generics in OOPs]

⇒ Code :-

type T is extending comparable so we compare 2 objects

```
class Heap<T extends Comparable<T>> {
    private ArrayList<T> list;
    public Heap(){                         → Heap constructor
        list = new ArrayList<>();
    }

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

Since in the theory we discussed index starting from 1 but in code we'll start from 0 & adjust others similarly.

```
    private int parent(int index){
        return (index - 1)/2 ;            ∵ parent was i/2
    }
         "      left      "
        return index * 2 + 1 ;            ∵ left was i * 2
    }                                     (soch ke dekho piche tree se)
         "      right     "
        return index * 2 + 2 ;            ∵ i*2+1
```

Inserting → public void insert(T value){

this will add at last index ← list.add(value);

now swapping bottom to up (upheap) ← upheap(list.size()-1);
}

private void upheap(int index){

if(index==0){

return;
}

start checking from parent of last index ← int p = parent(index);

if this index < parent →

if(list.get(index).compareTo(list.get(p))<0){

swap if yes ← swap(index, p);

repeat till above ← upheap(p);

}
}

Removing → public T remove() throws Exception {

if(list.isEmpty()){

throw new Exception("removing from an empty heap!");

first remove item at 0th index → T temp = list.get(0);

for placing last index item to its place ← T last = list.remove(list.size()-1);

~~taking last & removing from there~~

if(!list.isEmpty()){

list.set(0, last);

starting from 1st index to bottom ← downheap(0);
}

return temp;
}

```java
private void downheap(int index){
    int min = index;
    int left = left(index);
    "   right = right( " );
```

(if left exists) → if (left< list.size && list.get(min).compareTo(list.get(left)) >0){ — if min > left then swap both

```java
        min = left;
    }
```

lly for right

Now calling for next →

```java
    if (min != index){
        swap (min, index);
        downheap (min);
    }
}
```

- <u>Heap Sort</u>

↳ You have a heap, till the heap is empty keep removing items & put it in a list.

∴ removing n-items till last level of logN

∴ complexity → O(nlogN).

∴ ⇒

since already sorted hai

```java
public ArrayList <T> heapsort() throws Exception{
    ArrayList <T> data = new ArrayList<>();
    while (! list.isEmpty()) {
        data.add (this.remove());
    }
    return data;
}
```

}

```
import java.util.ArrayList;
psvm ( ) throws Exception {
HeapInteger> heap = new Heap<>();
```

Main func. →
(storing into Arraylist)

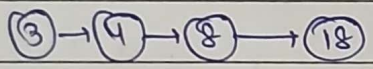
```
heap.insert (34);
:
:
:
ArrayList list = heap.heapSort();
sout (list);
}
}
```

"This is also called Priority Queue."

eg. getting min. item is the priority ∴ ↑

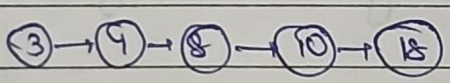
eg. " max " " " " ∴ ↓

→ Priority Queue as a Linked List



To insert 10 (while checking) at correct position
as in linked list we cannot jump in btw ∴ start from head & end till tail.

∴ checks 3 ✓ checks 4 ✓ checks 8 ✓ checks 18 ✗ ∴ place 10 before 18.

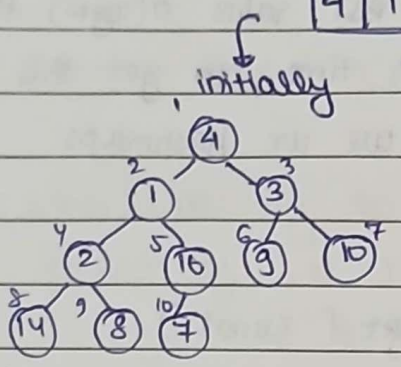


Time complexity → O(N) ∴ Not good for LL

∴ use Heaps.

Q. Create a max Heap from an unsorted Array.

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



→ Property condition of max Heap Val(node) ≥ childrens

Bottom starting from N/2 → because all after $\frac{N}{2}$ i.e $\frac{N}{2}+1, \ldots n$ are leaf nodes (they don't have any child to check).
(so start from 5th)

code → for (i = N/2 till 1)
↓ downheap(i) ✓
write on own ✓

[Time complexity = O(N)] not O(N/2 * log(N))

$\frac{N}{2}, \frac{N}{4}, \frac{N}{16}, \ldots$ } Harmonic Progression sum *