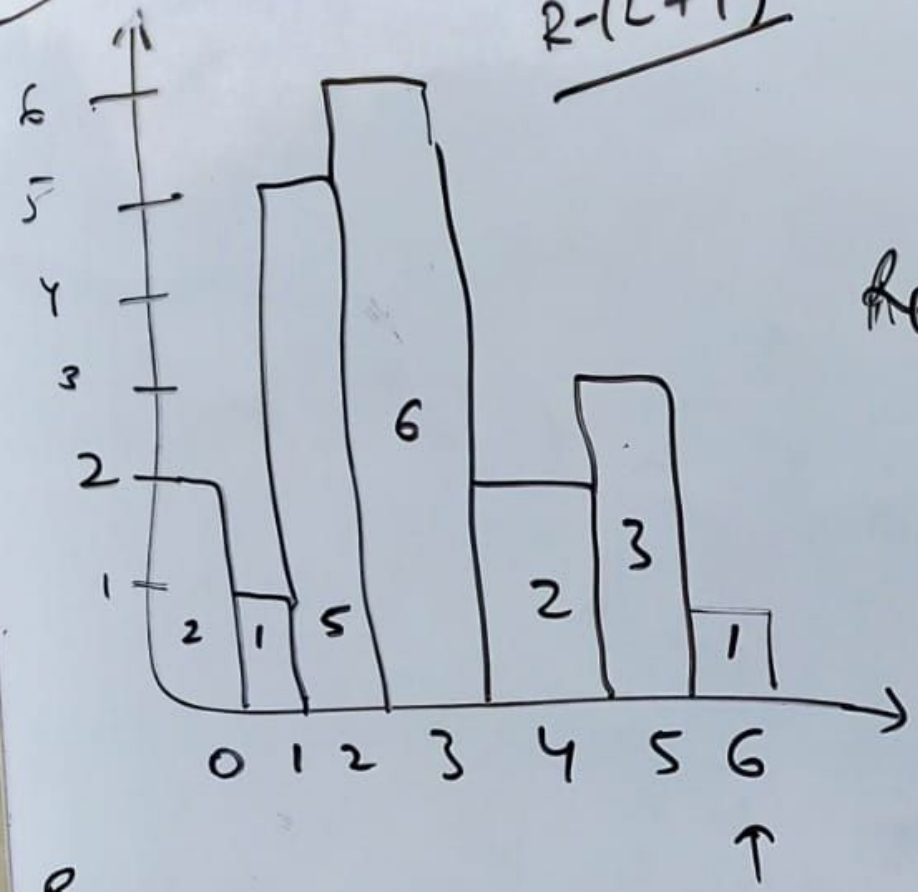
```
class MyQueue {

    int front, rear;
    int arr[] = new int[100005];

    MyQueue()
    {
        front=0;
        rear=0;
    }
    void push(int x)
    {
        arr[rear++] = x;
    }
    int pop()
    {
        if(front >= rear)return -1;
        return arr[front++];
    }
}
```

```
class MyStack
{
    int top;
    int arr[] = new int[1000];

    MyStack()
    {
        top = -1;
    }

    void push(int a)
    {
        arr[++top] = a;
    }
    int pop()
    {
        if(top==-1)return -1;
        return arr[top--];
    }
}
```

```
63
64  /* Structure of the class is
65  class TwoStack
66  {
67
68      int size;
69      int top1,top2;
70      int arr[] = new int[100];
71
72      TwoStack()
73      {
74          size = 100;
75          top1 = -1;
76          top2 = size;
77      }
78  }*/
79
80  class Stacks
81  {
82      //Function to push an integer into the stack1.
83      void push1(int x, TwoStack sq)
84      {
85          if(sq.top1+1==sq.top2)return;
86          sq.arr[++sq.top1] = x;
87      }
88
89      //Function to push an integer into the stack2.
90      void push2(int x, TwoStack sq)
91      {
92          if(sq.top2-1==sq.top1)return;
93          sq.arr[--sq.top2] = x;
94      }
95
96      //Function to remove an element from top of the stack1.
97      int pop1(TwoStack sq)
98      {
99          if(sq.top1==-1)return -1;
100         return sq.arr[sq.top1--];
101     }
102
103     //Function to remove an element from top of the stack2.
104     int pop2(TwoStack sq)
105     {
106         if(sq.top2==sq.size)return -1;
107         return sq.arr[sq.top2++];
108     }
109 }
110
111
```

LAH

R-(L+1)

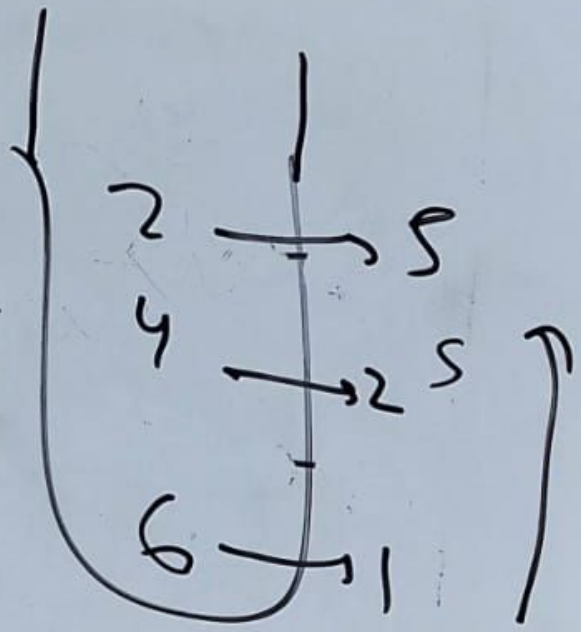



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| L | -1 | -1 | 1 | 2 | 1 | 4 | -1 |
| R | 1 | 7 | 4 | 4 | 6 | 6 | 7 |

| res | | | | | | | |
|---|---|---|---|---|---|---|---|
| | 2 | 7 | 10 | 6 | 8 | 3 | 7 |

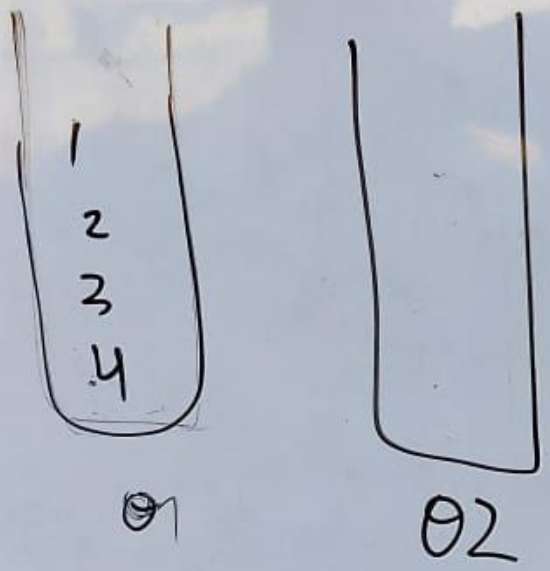
R

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 7 | 4 | 5 | 6 | 6 | 7 |




next SMALLER ELE

# IMPLEMENT STACK USING QUEUE




1, 2, 3, 4

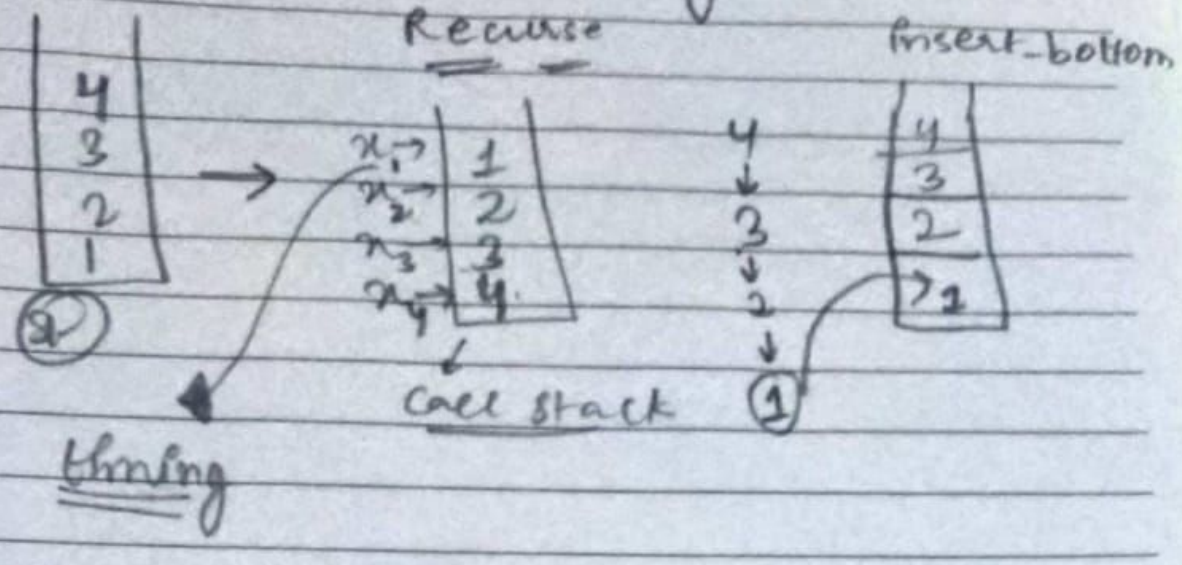
**PUSH (x)**

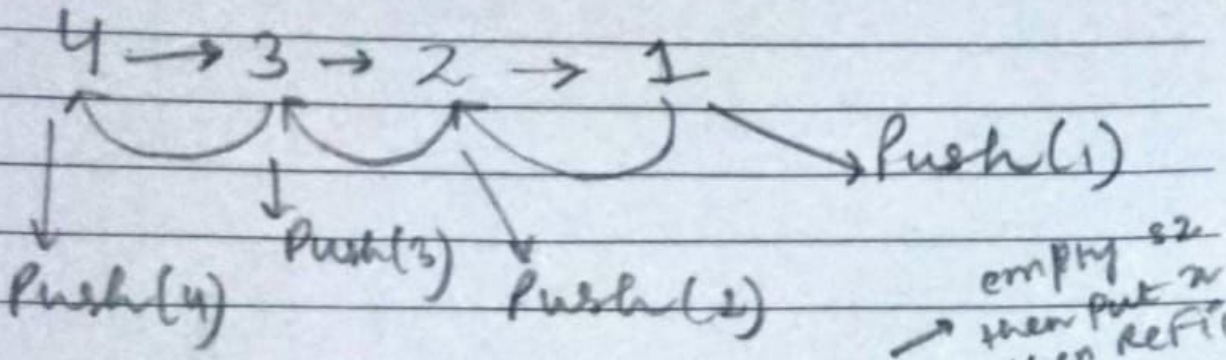
1) Q1 → Q2 (ALL)
2) Add x → Q1
3) Q2 → Q1 (ALL)

**POP ( )**

1) Q1.POP()

# Reverse a stack using Recursion →

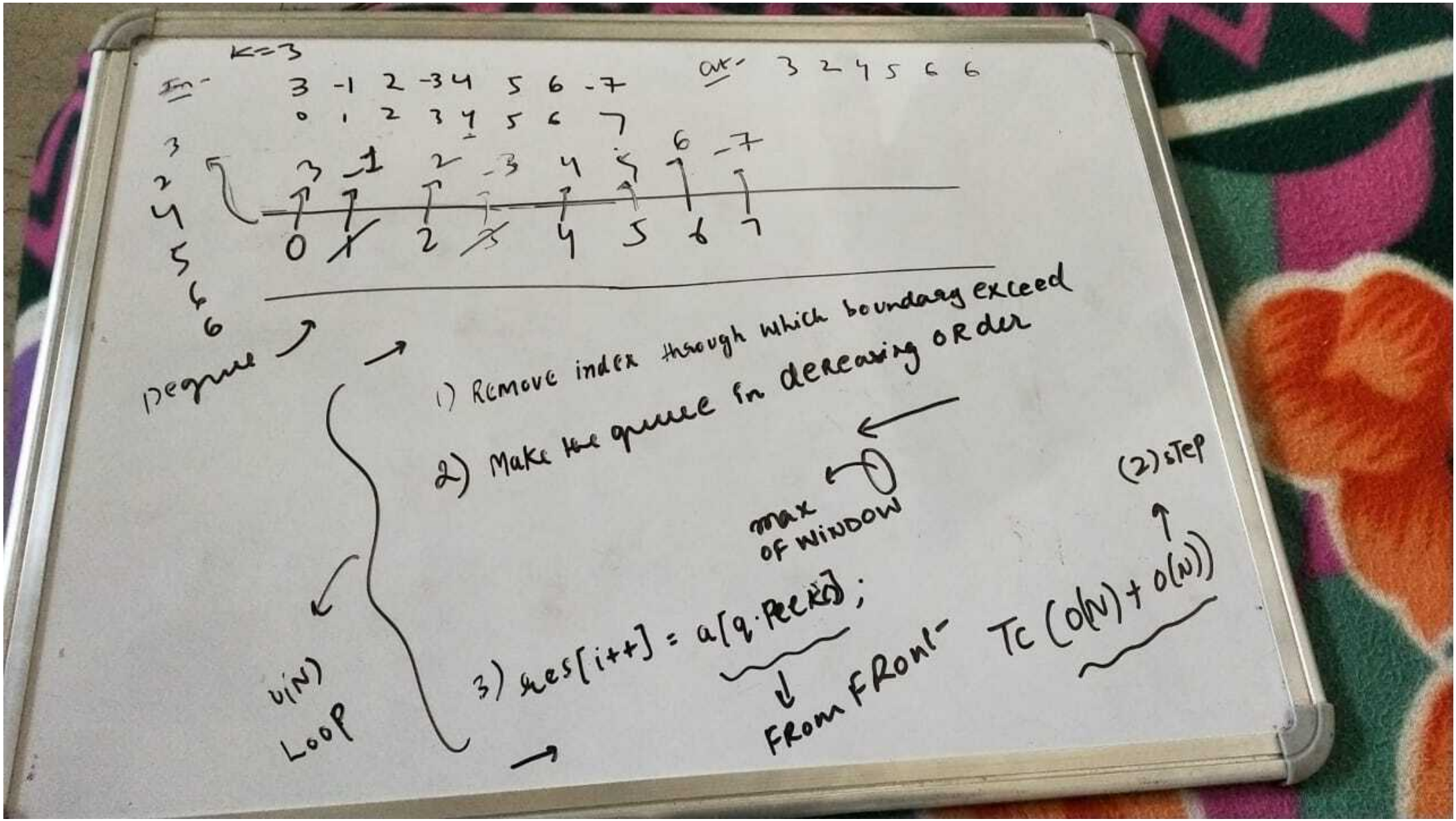



```
void reverse(){
if(!st.isEmpty()){
int x = st.pop();
reverse();
insert(x);
}
}
```

```
void insert(int x){
if(st.isEmpty())
    st.push(x);
else{
int a = st.pop();
insert(x);
st.push(a);
}
}
```

ERD
The Power of Innovation


K=3
In- 3 -1 2 -3 4 5 6 -7
0 1 2 3 4 5 6 7
Out- 3 2 4 5 6 6
3 -1 2 -3 4 5 6 -7
0 1 2 3 4 5 6 7
3
2
4
5
6
6
Deque
1) Remove index through which boundary exceed
2) Make the queue in decreasing order
max of window
(2) step
O(N) Loop
3) res[i++] = a[q.peek()];
From FRONT
TC (O(N) + O(N))

Get Min STACK

APP 1 → O(2N) SPACE
O(1) TIME

StackNode {

int val, min;

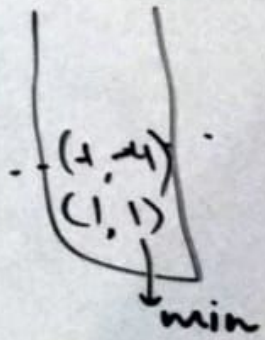
}

Push(1)

Push(-4)

Pop() → 1

getMin → Top.min (1)



# Get Min STACK

APP 2 → O(N.) SPACE
O(1) TIME

OPTIMAL

Push(2×val - min)
Pop(2×min - val)

if less than min

val + val < min + val → 2val - min < val

2 × min - (2 × val - Prev_min)

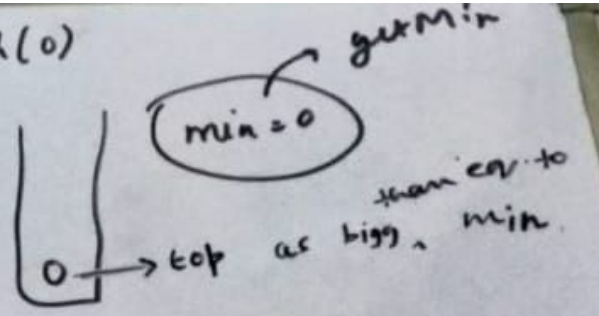
2 × min - (2 × min) - Prev_min

min = Prev_min

Rollback

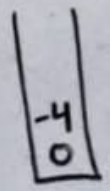
st.top → current min

Push(0)




Push(-2)

min = 0 - 2




Insert → 2 × (-2) - 0 = -4

top = -2 (as st.top < min)

getMin → -2

pop() as top < min

1) val = st.pop() // -4
2) min = 2 × (-2) - (-4) = 0

↓

ROLLBACK

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |

HEAP

ans $\{-\infty, \infty\}$

max → ~~5~~ 9 } 9-1 ⇒ 8 < ∞

1 } {1, 9}

1, 5, 9
2

→ 1
→ 2
→ 3
→ 4

9-2 ⇒ 7 < 8

{2, 9}

MIN Heap

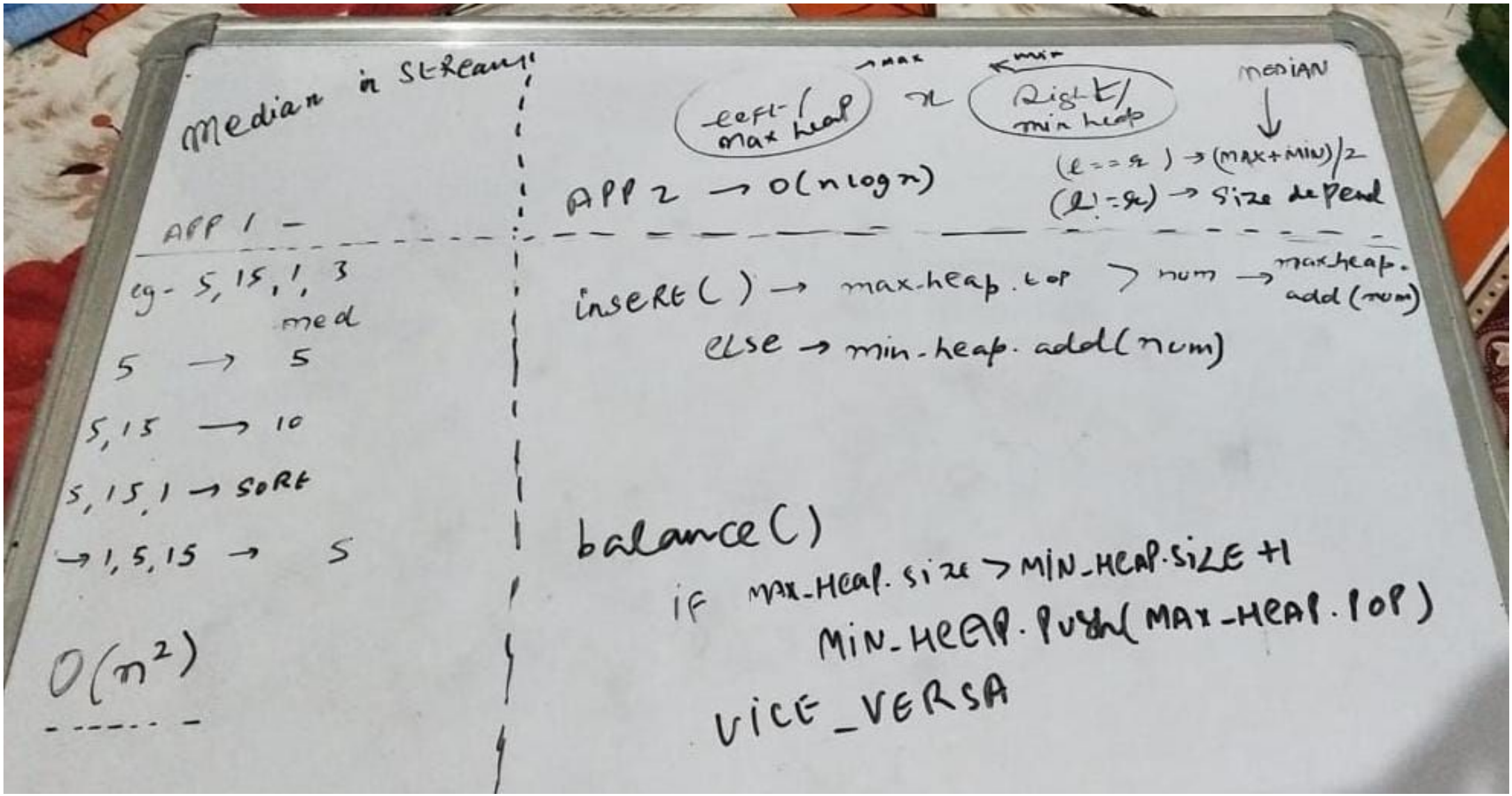
9-4 ⇒ 5, 5 < 6

① Min heap
② Track max element {4, 9} minimum Range

MINIMUM RANGE IN K LIST

ERD The Power of Innovation

O(n * K * log K)


Median in Stream
APP 1 –
eg- 5, 15, 1, 3
med
5 → 5
5, 15 → 10
5, 15, 1 → SORT
→ 1, 5, 15 → 5
O(n²)
left / max heap
max
right / min heap
min
MEDIAN
APP 2 → O(n log n)
(L == R) → (MAX+MIN)/2
(L != R) → size depend
insERT( ) → max-heap.top > num → maxheap.add(num)
else → min-heap.add(num)
balance( )
if MAX-HEAP.size > MIN_HEAP.SIZE +1
MIN_HEAP.push(MAX-HEAP.top)
VICE_VERSA

Kth largest sum contigous subArray → 2020

a[] → {20, -5, -1}  (K = 3)

ALL sum → {20, 15, 14, -5, -6, -1}

K = 3 } → (14)

| 0 | 20 | 15 | 14 | → Prefix sum |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | |

```
FOR( int i=1; i<=n; i++) {
  FOR( int j=i; j<=n; j++) {
    int x= sum[j] - sum[i-1];
    if (q.size() < K) q.add(x);
    else if(q.peek < x)
    {
      q.Pole();
      q.add(x);
    }
  }
}
return q.pole();
}
```

MIN HEAP

NlogN

N

# HEAP (Complete binary Tree)

↳ Full binary Tree upto (n-1)

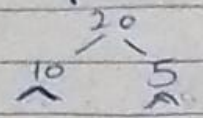
→ Array Representation

→ Node at i { left child (2*i+1), right child (2*i+2) }

↙

Parent [i/2]

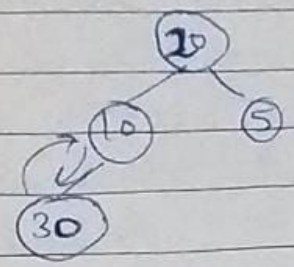
Max heap / min heap



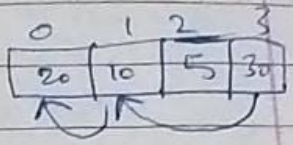

## Insert element into heap

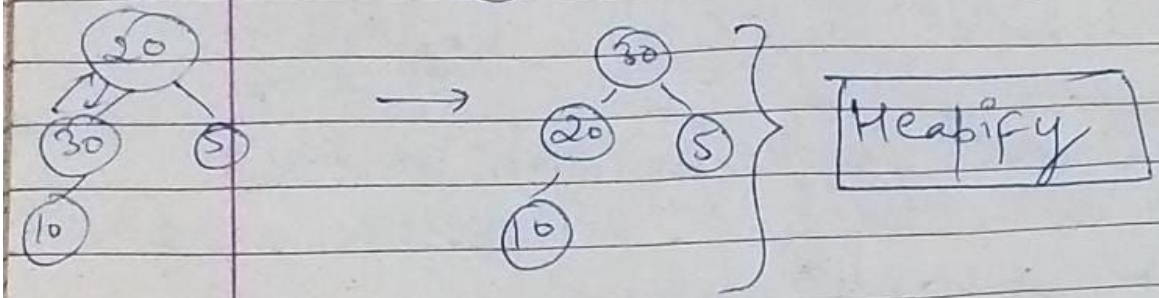
1) Put it into last of array
2) compare with parent & swap if violating

TC: O(logn)




Insert → 30



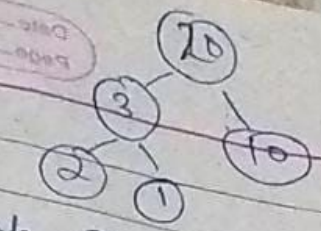




Insert take O(1) & Heapify Take O(logn)

## Delete element → 1) just swap the first and last element of heap

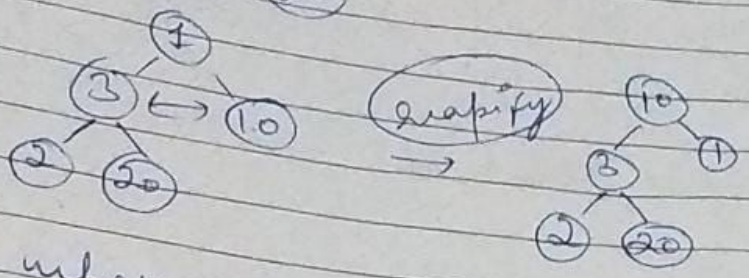
2) now Heapify by comparing the child of first element and it's parent

O(logn) → heapify



Delete will give 20

Swap 1 & 20



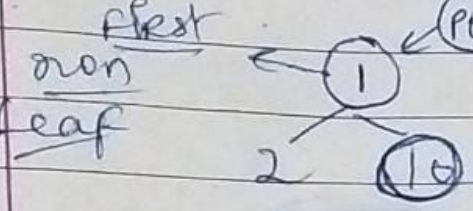

So when we delete all we will get the increasing order that's why for kth Largest element we use min heap. & vice versa.

Heap SORT → create heap & Delete all

Heapify → start from last of array & follow a heap Property to create a heap.

Takes O(n) : (optimal)

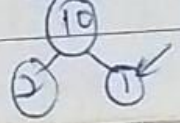
→ 1) Level (Reverse Level order) & at each node check top-down approach.




FIRST-non-Leaf-node → (3/2) - 1 ⇒ 0

10 > 2 so, i=1 largest = 10

swap them & move Ptr to Largest index



```
class Solution {
    public int countNodes(Node root){
        if(root==null)return 0;
        return 1+countNodes(root.left)+countNodes(root.right);
    }
    public boolean isComplete(Node root,int index,int num_nodes){
        if(root==null)return true;
        if(index>=num_nodes)return false;
        return isComplete(root.left,2*index+1,num_nodes) && isComplete(root.right,2*index+2,num_nodes);
    }
    public boolean isCompleteTree(Node root) {
        return isComplete(root,0,countNodes(root));
    }
    boolean isHeapUtil(Node root)
    {
        if (root.left == null && root.right == null)
            return true;

        if (root.right == null) {
            return root.data >= root.left.data;
        }
        else {
            if (root.data >= root.left.data
                && root.data >= root.right.data)
                return isHeapUtil(root.left)
                    && isHeapUtil(root.right);
            else
                return false;
        }
    }
    boolean isHeap(Node tree) {
        if(tree==null)return true;
        return isCompleteTree(tree) && isHeapUtil(tree);
    }
```