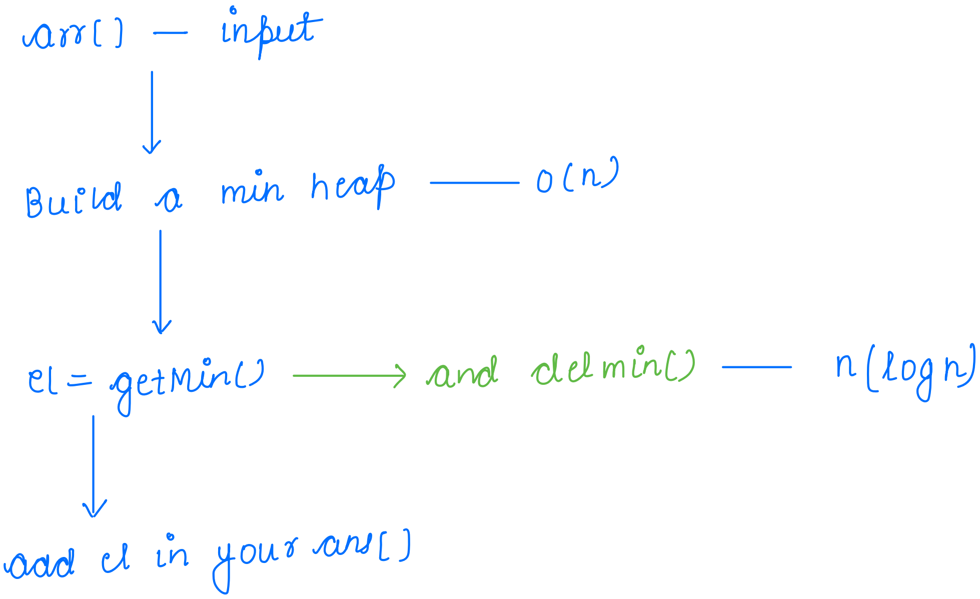
Lecture : Heaps 2

Agenda

- Heap sort
- Kth largest element
- Kth largest element in all windows
- Nearly sorted array.
- Running Median

Qu Sort an array in inc^r order using a heap.

Approach1



arr[] — input

↓

Build a min heap —— O(n)

↓

el = getMin() ⟶ and delmin() —— n(logn)

↓

add el in your ans[]

TC: O(nlogn)

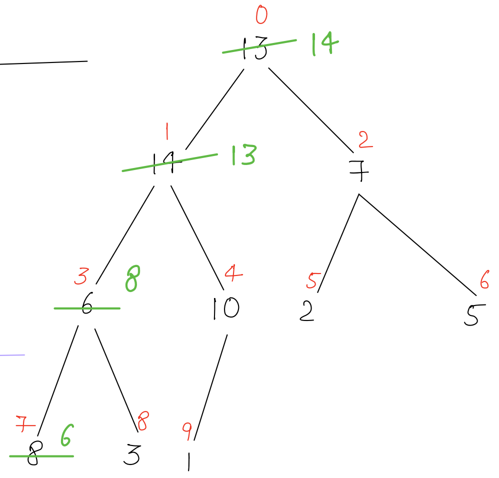
SC: O(n)

Approach2 Max heap

0	1	2	3	4	5	6	7	8	9
~~13~~ 14	~~14~~ 13	7	~~6~~ 8	10	2	5	~~8~~ 6	3	1

Dry run : Non-leaf nodes [4th - 0th]

idx	Left child 2*idx +1	Right child 2*idx+2	swap
4 A[4] = 10	9 A[9] = 1		No
3 A[3] = 6	7 A[7] = 8	8 A[8] = 3	swap(3, 7)
2			no
1			no
0			swap(0,1)
1			do not swap



Logic

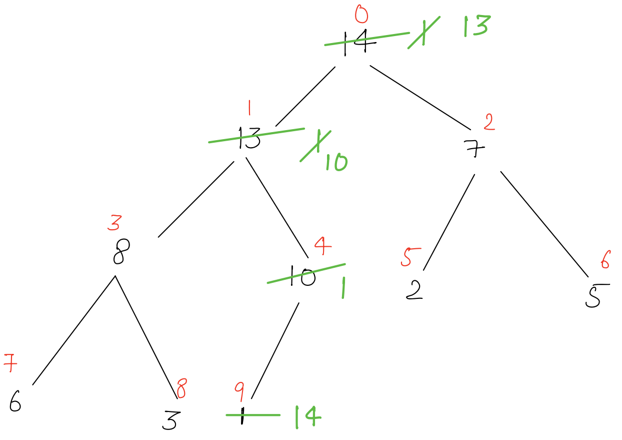
0	1	2	3	4	5	6	7	8	9
~~14~~ ~~1~~ 13	~~13~~ ~~1~~ 10	7	8	~~10~~ 1	2	5	6	3	~~1~~ 14

Dry run

swap(0, 9)

downheapify (0, 8)

idx	lc 2idx+1	rc 2idx+2	swap
0 A[0]=1	1 A[1]=13	2 A[2] = 7	swap(0,1)
1 A[1]=1	3 A[3]=8	4 A[4]=10	swap(1,4)
4			

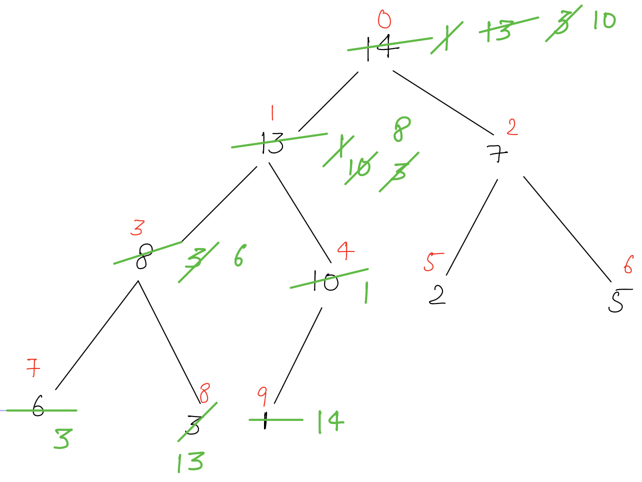


0	1	2	3	4	5	6	7	8	9
~~14~~ ~~1~~ ~~13~~ ~~3~~ 10	~~13~~ ~~1~~ ~~10~~ ~~3~~ 8	7	~~8~~ ~~3~~ 6	~~10~~ 1	2	5	~~6~~ 3	~~3~~ 13	~~1~~ 14

swap(0, 8)

downheapify (0, 7)

idx	lc 2idx+1	rc 2idx+2	swap
0 A[0] =3	1 A[1]=10	2 A[2] =7	swap(0, 1)
1			swap(1,3)
3			swap(3,7)



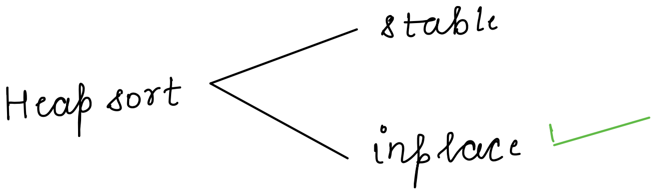
To be contn.~

Algorithm

Build a max heap —— O(n), O(1)

```
j = n-1;
while( j > 0 ) {
    swap(0, j);
    j--
    downheapify(0, j); —— O(logn)
}
```

TC: O(nlogn)
SC: O(1)



Qu Given arr[n], find Kth largest element —— min heap.

0	1	2	3	4	5	6
8	5	1	2	4	9	7

K	ans
1	9
2	8
3	7

Approach1 Arrays. sort(arr)

Return arr[n-k]

TC: O(nlogn)
SC: O(1)

Approach2 Binary search. —— h/w

Max
min } state space search.

Approach3 Using heap sort — h/w

Build Max heap

↓

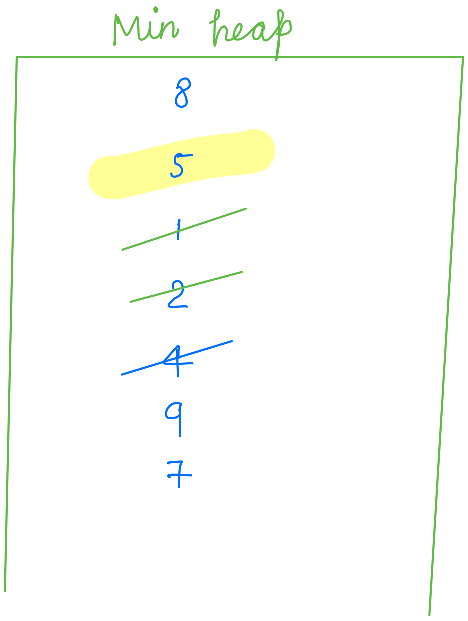
getMax (k-1) times & delete max.

TC:
SC:

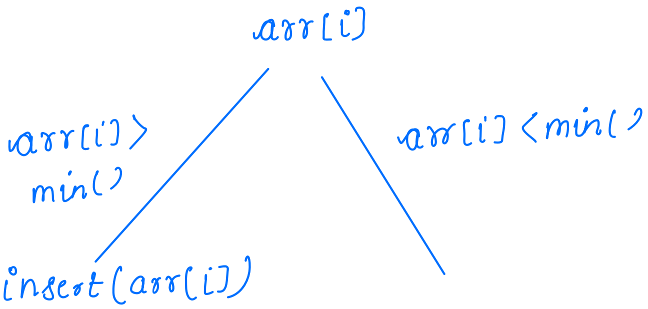
Approach

0	1	2	3	4	5	6
8	5	1	2	4	9	7

K = 4



Min heap
↓
K elements
↓
min() will be K th largest.



Pseudocode

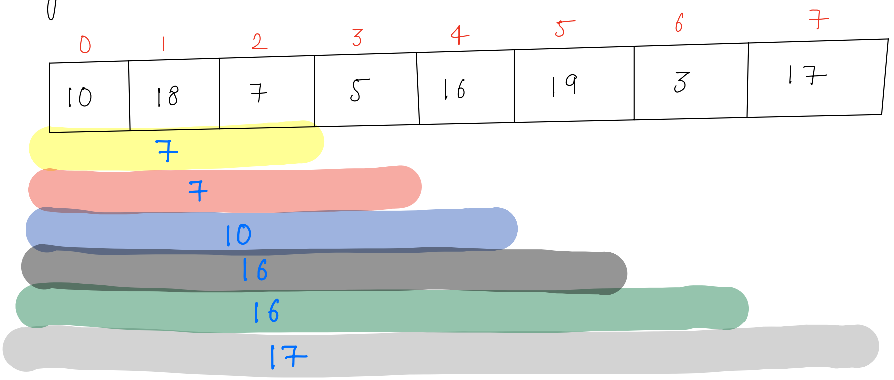
```
int kthLargestElement(arr[], k) {
    PriorityQueue<Integer> pq = new PriorityQueue<>();
    for(i=0; i<k; i++) {
        pq.add(arr[i]);
    }

    for(i=k; i<n; i++) {
        if (arr[i] > pq.peek()) {
            pq.poll();
            pq.add(arr[i]);
        }
    }
    return pq.peek();
}
```

TC: $(n-k) * \log k$
SC: $O(k)$

Kth smallest element ——— max heap.

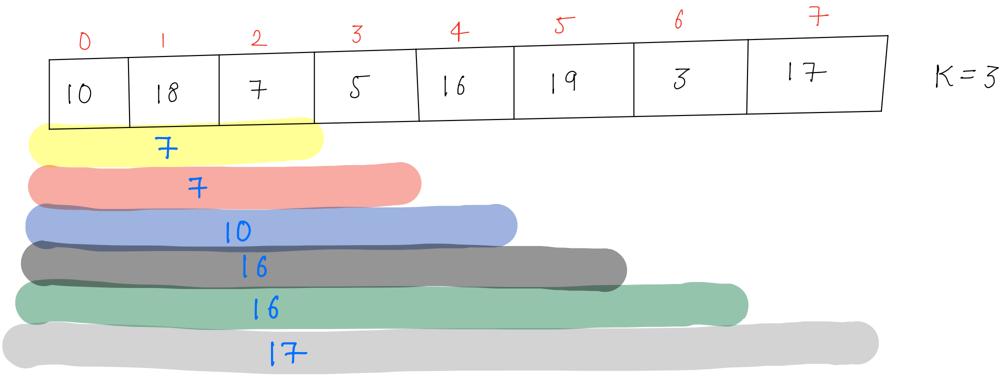
Qu find Kth largest element for all windows of an array starting from 0th index.



0	1	2	3	4	5	6	7
10	18	7	5	16	19	3	17

K=3

Approach



min-heap

~~10~~
18
~~7~~
~~16~~
19
17

yellow	heap.peek() = 7
red	〃
blue	10
black	16
green	16
grey	17

Pseudocode

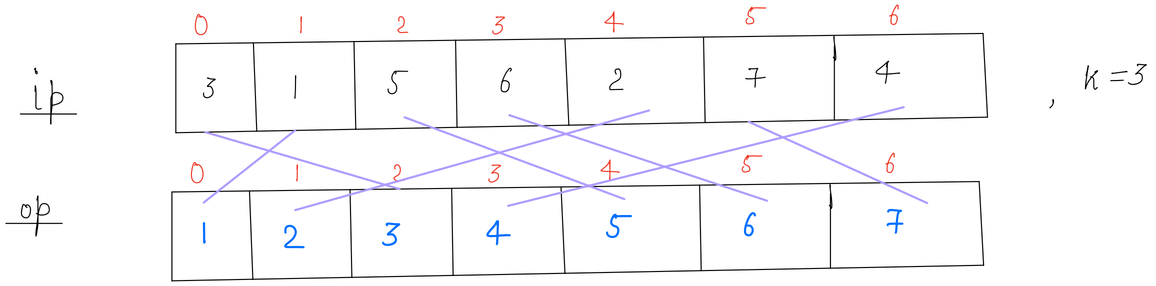
```
void  kthLargestForAllWindow( int[] arr, int k) {
        PriorityQueue<Integer> pq = new PriorityQueue<>();

        for(i=0; i<k; i++) {
                pq.add( arr[i]);
        }
        print(pq.peek());
        for( i=k; i<n; i++) {
              if ( arr[i] > pq.peek()) {
                        pq.poll();
                        pq.add( arr[i]);
              }
        }     print(pq.peek());

}
```

Qu Given arr[n] and k. Every element is at max K distance away from its sorted position. Sort the array.

Note: k is very small w.r.t n.



Approach

0	1	2	3	4	5	6
3	1	5	6	2	7	4

, k =3

output

0	1	2	3	4	5	6
1	2	3	4	5	6	7

Min -heap

~~3~~
~~1~~
~~5~~
~~6~~
~~2~~
~~7~~
~~4~~

idx.	
0th	(0 -3) idx

Pseudocode

```
void sort( int[] arr, int k) {
    PriorityQueue<Integer> pq = new PriorityQueue<>();

    for(i=0; i<=k; i++) {
        pq.add(arr[i]);
    }
    idx = 0;

    for(i=k+1; i<n; i++) {
        arr[idx] = pq.poll();
        idx++;
        pq.add(arr[i]);
    }
    while(!pq.isEmpty()) {
        arr[idx] = pq.poll();
        idx++;
    }
}
```

TC: O(n logk)
SC: O(k)

Break: 8:40 – 8:50

Qu Given a running stream of integers, find median for all inputs [Hard]

Median Ex1

5	10	2	1	4

↓ sort

1	2	4	5	10

Median ⇒ 4

Ex2

5	10	2	3	1	4

↓ sort

1	2	3	4	5	10

Median ⇒ $\frac{3+4}{2} = 3.5$

ip

9	8	17	20	25	10	5	3

9

8.5

[8 9 17] = 9

[8 9 17 20] = $\frac{9+17}{2}$ = 13

[8 9 17 20 25] = 17

[8 9 10 17 20 25] = $\frac{10+17}{2}$ = 13.5

⋮

Approach

Case1:

5	7	4	3	6	2	1

↓ sort

1	2	3	4	5	6	7

Part1 Part2

Median ⇒ Max of part 1.

Case2

1	2	3	4	5	6	7	8

Part1 part2.

Median ⇒ $\dfrac{\text{Max of part1} + \text{Min of part2}}{2}$

Observation

1.) if no of elements are odd.

$$\underbrace{\boxed{1}\boxed{2}\boxed{3}\boxed{4}}_{P1}\underbrace{\boxed{5}\boxed{6}\boxed{7}}_{P2}$$

Part1 $\Rightarrow \frac{n+1}{2}$ el.

Part2 $\Rightarrow \frac{n}{2}$ el.

Ans = max of part1. [achieve it using max heap]

2.) if no of elements are even.

$$\underbrace{\boxed{1}\boxed{2}\boxed{3}\boxed{4}}_{P1}\underbrace{\boxed{5}\boxed{6}\boxed{7}}_{P2}\boxed{8}$$

Part1 $\Rightarrow \frac{n}{2}$ el.

Part2 $\Rightarrow \frac{n}{2}$ el.

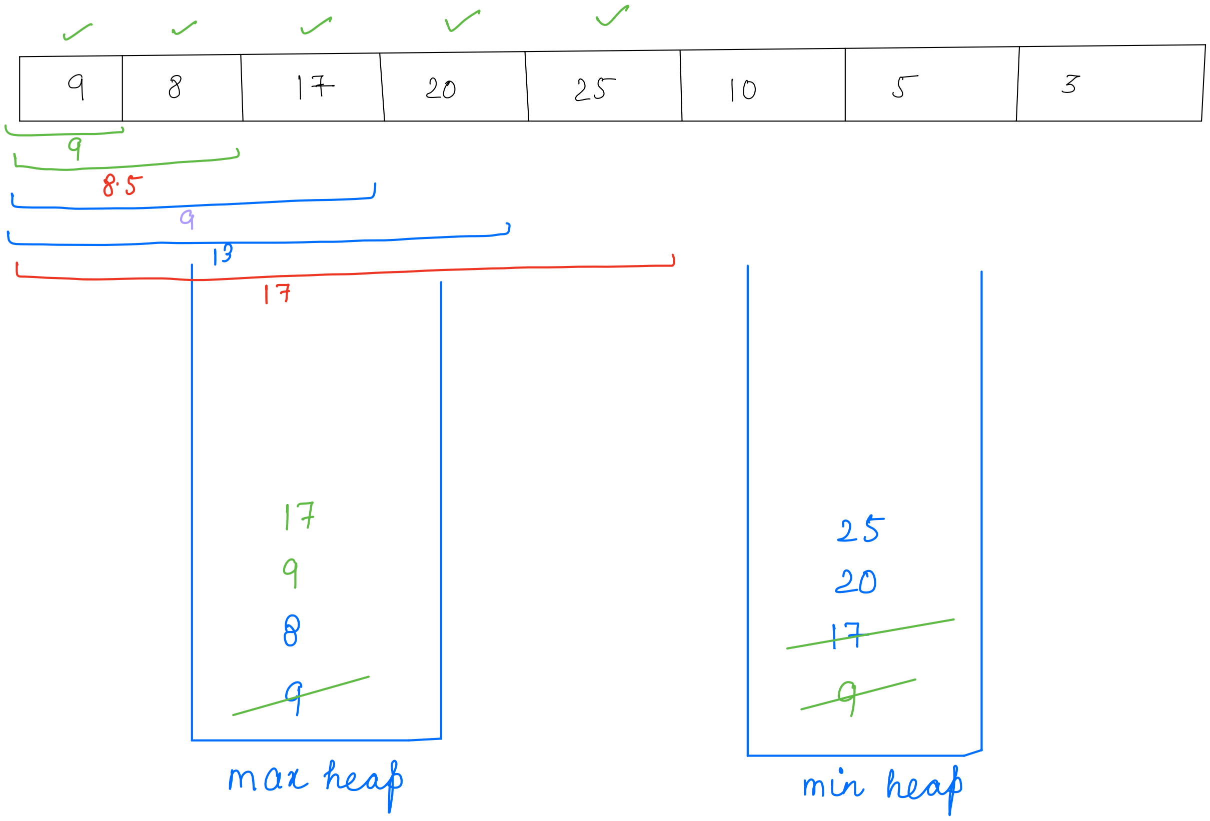
Max heap

Min heap

ans = $\dfrac{\text{Max of part1} + \text{Min of part2}}{2}$

3.) size(maxheap) − size(minHeap) <= 1

Dryrun



Balancing

size(maxHeap) − size(Minheap)

0 — $\frac{\max(\text{maxHeap}) + \min(\text{minHeap})}{2}$

1 — max(maxHeap)

2 — minHeap.add(maxHeap.poll())

−1 — maxHeap.add(minHeap.poll())

Pseudocode

h/w

```
void runningMedian( arr[] ) {
        // Max heap ——— pq1
        // Min Heap ——— pq2

        pq1.add( arr[0] );
        print ( pq1.peek() );

        for( i=1; i<n; i++ ) {
            curr = arr[i];
            if ( curr < pq1.peek() ) {
                pq1.add ( curr );
            } else {
                pq2.add ( curr );
            }

            // check for balance
            if( pq1.size() - pq2.size() > 1 ) {
                int el = pq1.poll();
                pq2.add (el);
            }

            if( pq1.size() - pq2.size() < 0 ) {
                int el = pq2.poll();
                pq1.add (el);
            }
            int totalsize = pq1.size() + pq2.size();
            if ( totalsize % 2 == 0 ) {
                print ( $\frac{pq1.peek() + pq2.peek()}{2}$ )

            } else {
                print ( pq1.peek() );
            }
}
```

Thankyou ☺