Heaps -2

Question: Heap Sort

A = 2 1 7 4 5

Sort(A) = 1 2 4 5 7

Approach 1:

→ Build a min-heap from the array $O(n)$

→ Extract the min element one by 1.

ExtractMin() ⇒ $O(\log n)$

$\boxed{\cancel{2},\ 7,\ \cancel{4},\ 5}$

Min-heap

$\log n$
$\log n$
$\log n$
⋮

Arr : $\boxed{1,\ 2,\ 4}$

T.C : $n \log n$
S.C : $O(n)$ ⇒ Created one extra array

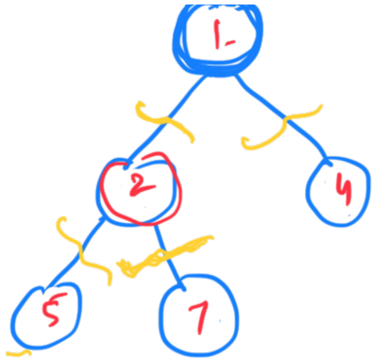
Approach 2:

Sort(A) =	1	2	4	5	7
A =	2	1	7	4	5
Maxheap :	1	2	4	5	7
	0	1	2	3	4

→ Sorted in increasing order

1) Build a max heap : $O(n)$

$O(1)$



Max-heap

-1 1) Swap 1st & last element

2) Percolate-down

(logn)

n times

T.C: $O(n) + O(n \log n) \approx O(n \log n)$

S.C: $O(1)$

Question: k^{th} Smallest element in a stream of numbers.

K = 3

	0	1	2	3	4	5	6	7	8	9	10	11
A =	10	12	8	14	4	20	16	7	2	9	1	3
	-1	-1	12	12	10	10	10	8	7	7	4	3

Brute Force:

-) Everytime, sort the array and return k^{th} element.

T.C: $(n \log n) \times n$

: $\boxed{O(n^2 \log n)}$

Approach:

Maintain array in sorted.

A = 10 12 8 14 4 20

1

A' = 1 3 4 5 6 ...

T.C for sorted insert : $\boxed{O(n)}$

For 1 element : $O(n)$

↓

n times

$\boxed{T.C : O(n^2)}$

Efficient Approach :

→ Find the smallest element / Largest

k^{th} smallest / k^{th} largest

↓

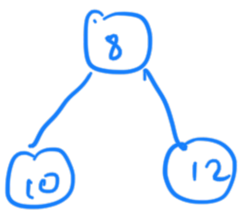
$\boxed{\text{Heaps}}$

→ Heap of size k.

→ Min-Heap / Max-heap $\quad k = 3$

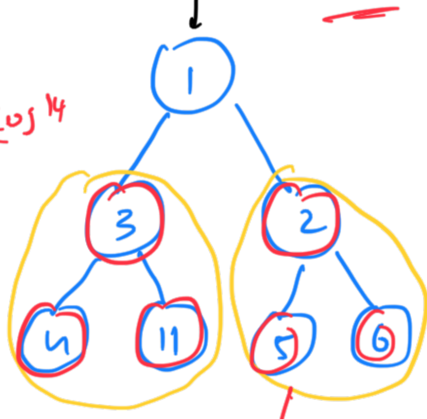
A =	10	12	8	14	4	20	16	7	2	9	1	3
	-1	-1	12	12	10	10	10	8				

$\boxed{\text{Min-Heap}}$



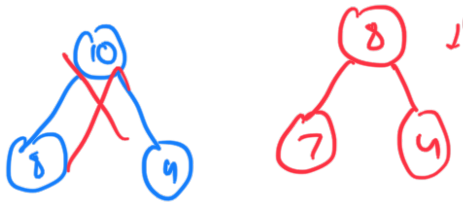
T.C to access max ele in min-heap ?

Min-heap



↓ $O(n)$

Max-heap



↓ $O(\log k)$

1) ExtractMax() => $O(\log k)$

2) Insert new => $O(\log k)$

When a new number comes : log k
↓
N times
T.C : O(N log K)
S.C : O(K)
↓
Heap of size K.

Question : Sort a K-sorted array (Nearly sorted Array)

→ Given an array, every element is at max K distance away from its position in sorted order

→ sort the array

K+1 element

K = 3

A =	10	18	7	5	6	22	19	
				10	18	19	22	
Sort(A) =	5	6	7	10,	18	19	22	
	5	6	7					

Brute - Force :

Apply any sorting Algo on array A.

T.C : $\boxed{O(n \log n)}$

1) Deleting Min → O(log k)
2) Insert → O(log k)

K+1

Size of Heap : (K+1)

Approach 2 :

T.C : O(N · log K)
S.C : O(K)

→ $\boxed{\text{Simple}}$

Question: Running Median

→ Given a stream of numbers, you have to return the median of numbers received so far.

Median : Middle element in sorted array

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

Med = 5.

Case 1 : odd Elements.

median = A[n/2] ✓

Case 2 : Even Elements. → 0 - indexing

$(\frac{6}{2}-1)$ arr$[\frac{6}{2}]$

A =	1	2	3	4	5	6
	0	1	2	3	4	5

Med = $\frac{3+4}{2}$

N = 6

$$\frac{\left(A\left[\frac{n}{2}\right] + A\left[\frac{n}{2}-1\right]\right)}{2} \checkmark$$

$A[\frac{n}{2}], A[\frac{n}{2}-1]$

A =	9	8	7	3	6	4
Med :	9	8.5	8	7.5	7	6.5

$\frac{8+9}{2}$

7 8 9

3 7 8 9

3 6 7 8 9

3 4 6 7 8 9

Brute Force :

$(n \log n) \times N = O(N^2 \log N)$

Approach 2:

Sorted Insert ⟹ $O(n)$

↓

N times

T.C: $O(n^2)$

Efficient Approach:

Observations

1) We're only interested in the middle elements of array

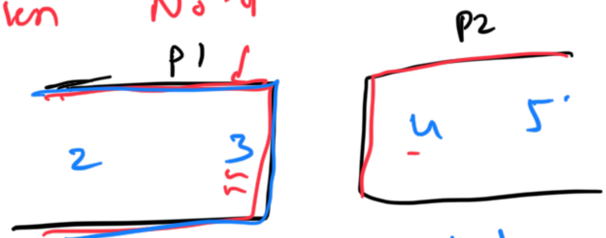
2) Do we need array in sorted order?

NO

3) Finding Median is $O(1)$ if array is sorted

4) T.C: $\boxed{O(N)}$ → (for sorted input)

Case 1: Even No. of elements. (3.5)

A = 

→ Divide into equal halves ($\frac{n}{2}$ eles in each)

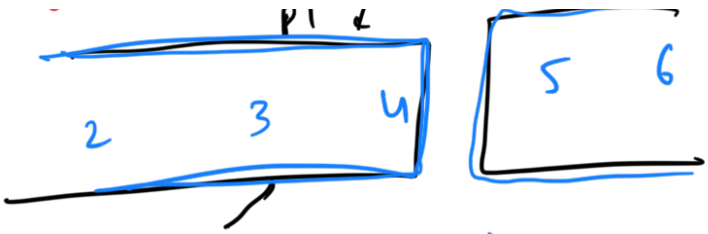
$$\text{Median} = \frac{\text{Max}(P1) + \text{Min}(P2)}{2}$$

⟹ All elements in P1 HAS TO BE LESS than elements in P2

odd No. of elements   P2   Med = 4

Case2:

A =



Median = Max(p1)

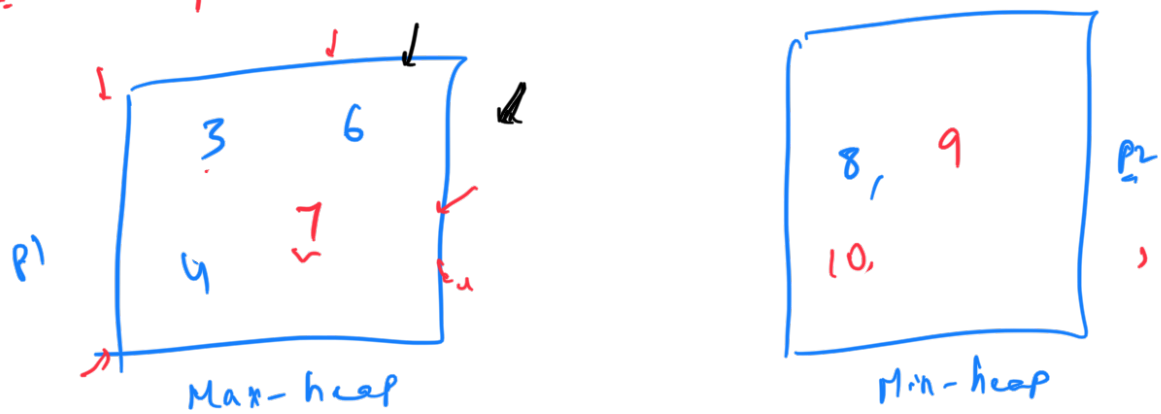
p1 => Max heap

p2 => Min Heap

size(Max Heap) - size(Min Heap) = {0, 1}

A =	9	8	7	3	6	4	10
Med =	9	8.5	8	7.5	7	6.5	7

(ele < root)



size(p1) - size(p2)

3 - 4 = {-1}

T.C: (log n) + log n

(Min)

push -> Insert
pop -> Delete Min

Pseudocode:

```
priority_queue<int> maxHeap, minHeap;

int median(int x) {
    if(maxHeap.size() == 0 || x < maxHeap.top()) {
        maxHeap.push(x);
```

A = 5 4 6

MaxHeap MinHeap

}
else
minHeap.push(x);
}
if(maxHeap.size() - minHeap.size() > 1) {
int temp = maxHeap.top();
maxHeap.pop();
minHeap.push(temp);
}
if(maxHeap.size() - minHeap.size() < 0) {
int temp = minHeap.top();
minHeap.pop();
maxHeap.push(temp);
}
if(maxHeap.size() == minHeap.size()) {
return (maxHeap.top() + minHeap.top())/2
}
else {
return maxHeap.top();
}
}

→ Deletion
→ Insertion
O(logn)

T.C : O(logn × N)
= O(N logN)

S.C: O(N)
{
2 Heaps of size $\frac{N}{2}$ each

Question: Maximum array sum after K negations

A =	24	-68	-29	-9	84
		68	-29	-9	84
	24			-9	84
	24	68	29		
	24	68	29	9	84
	24	68	29	9	84
→ 24 ⇒	-24	68	29	-9	84
9 ⇒		68	29	-9	84

B = 4 24

1) -68 → 68

2) -29 → 29

3) -9 → 9

4)

A = 24 68 29 9 84 4

Min

→ Maintain a min Heap

(B times)

→ for (i=0 ; i< B ; i++) {

(logn) min_ele = minHeap.top();

minHeap.pop();

minHeap.push(-min_ele);

}

Iterate & find the sum

→ Insert all element into min-heap ⇒ O(1)

T.C :

1) Building a min Heap ⇒ O(n)

2) For all B steps, ⇒ O(B logn)

3) Find sum of element of heap ⇒ O(n)

$$\boxed{\begin{array}{ll} T.C: & O(n + B\log n) \\ S.C: & O(1) \end{array}}$$

Element

→ Deleting

→ Insert new Element

Kth Smallest Element

```
priority_queue<int> pq;
vector<int> ans;
for(int i = 0; i < k; i ++){
      pq.push(a[i]);

      if(i == k-1) ans.push_back(pq.top());
      else   ans.push_back(-1);
}
for(int i = k; i < n; i++){
      if(a[i] < pq.top()){
            pq.pop();
            pq.push(a[i]);
      }
      ans.push_back(pq.top());
}
return ans;
```

Sort a nearly sorted array

```
vector<int> kPlaces(vector<int> a, int k){
      n = a.size();
      // Initialize a min heap
      priority_queue<int, vector<int>, greater<int>> pq;

      // Push first k+1 elements
      for(int i = 0; i <= min(k, n-1); i++){
            pq.push(a[i]);
      }
      for(i = k+1; i<n; i++){
            ans.push_back(pq.top());
            pq.pop();
            pq.push(a[i]);
      }

      while(!pq.empty()){
            ans.push_back(pq.top());
            pq.pop();
      }
      return ans;
}
```

Running Median

```
priority_queue<int, vector<int>, greater<int>> minHeap;
priority_queue<int> maxHeap;

// Median after the element x has been added to the stream
int median(int x){
	if(!maxHeap.size() || x <= maxHeap.top())
		maxHeap.push(x);
	else
		minHeap.push(x);

	if(maxHeap.size() - minHeap.size() > 1){
		temp = maxHeap.top();
		maxHeap.pop();
		minHeap.push(temp);
	}
	else if(maxHeap.size() - minHeap.size() < 0){
		temp = minHeap.top();
		minHeap.pop();
		maxHeap.push(temp);
	}

	if(maxHeap.size() == minHeap.size())
		return maxHeap.top() + minHeap.top() / 2;
	else
		return maxHeap.top();

}
```