OCT 2024	S	M	T	W	T	F	S
			1	2	3	4	5
	6	7	8	9	10	11	12
	13	14	15	16	17	18	19
	20	21	22	23	24	25	26
	27	28	29	30	31		

Heap

'24 SEPTEMBER 03 TUESDAY

36th Week 247-119

→ A Heap is a tree based Data structure which satisfies below properties:

(1) A Heap is a complete binary tree (all levels are completly filled except the last level and last level is filling from left to right)

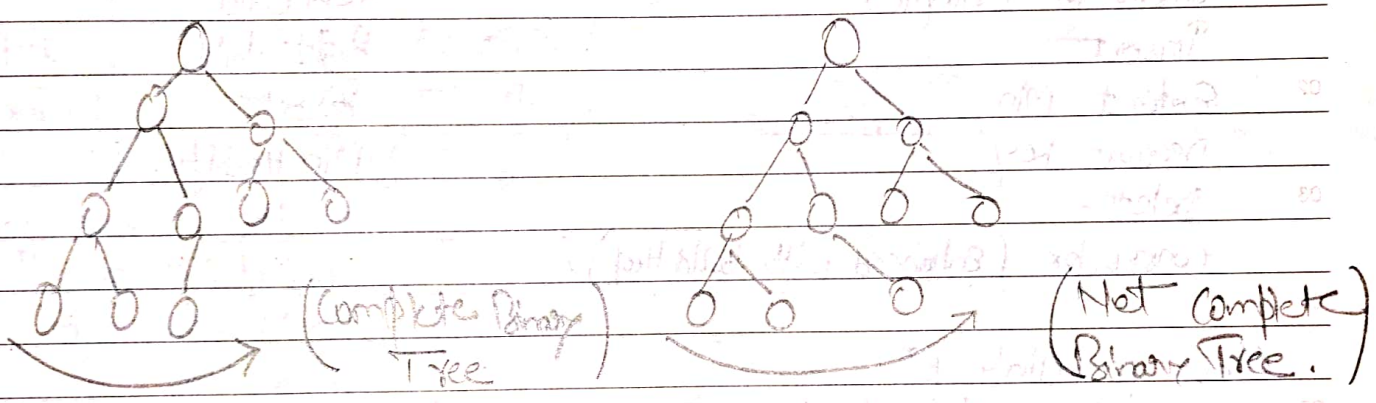
(2) A Heap is either Min Heap (or) max Heap.

In min Heap, the data at root must be minimum among all keys present in the binary Heap.

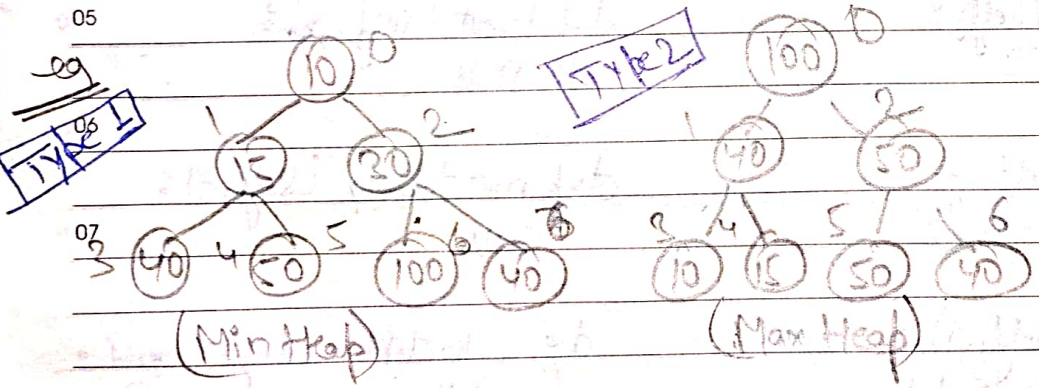
The same property must be recursively true for all Nodes in the Tree.

Max Heap is similar to Min Heap i.e, maximum among all at root.

Binary Heap: A Binary Heap is a heap where each node can have atmost two children. In other words, a binary Heap is a complete binary Tree satisfying the above-mentioned properties.



eg



Array representation

0	1	2	3	4	5	6
10	15	30	40	50	100	40

0	1	2	3	4	5	6
100	40	50	10	15	50	40

The root element will be at arr[0].

→ arr[(i-1)/2] # Returns the Parent Node. eg Parent(5)=2

→ arr[(2×i)+1] # Returns the left child Node. eg left(1)=3

→ arr[(2×i)+2] # Returns the right child Node. eg right(1)=4

SEPTEMBER
04 '24
36th Week
248-118
WEDNESDAY

S	M	T	W	T	F	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30					

SEP 2024

Advantages of this structure:—

08 (1) Random Access is Fast.
(2) Cache friendly [due to contiguous location].
09 (3) Complete binary Tree → height [min. possible height].
(4) Used in Heap sort.
10 (5) Used to Implement Priority Queue.

11 Note that Min Heap is internally represented as an Array Only.

12 **Heap Implementation.**

01 Constructor (Simple)
Insert
02 Extract Min
Decrease key
03 Delete
Constructor (Enhanced with build Heap)

Main operation

Left child
Right child
Parent
Min Heapify

Utility Function.

04

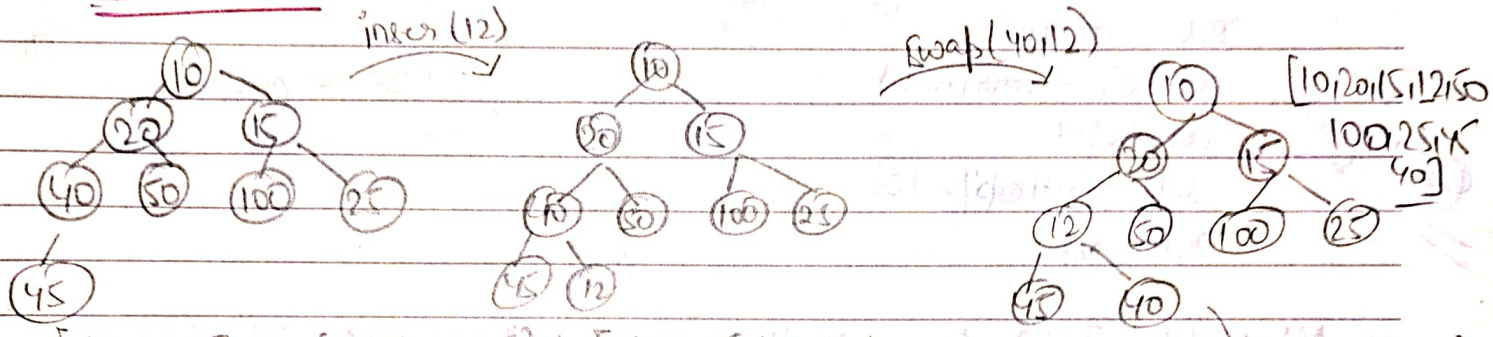
```
Class myMinHeap:
05  (1) def __init__(self):
        self.arr = []

06  (2) def parent(self, i):
07      return (i-1)//2

    (3) def lchild(self, i):
        return 2*i+1

    (4) def rchild(self, i):
        return 2*i+2
```

```
def insert(self, n):
    pass

def minHeapify(self, i):
    pass

def extractMin(self, i):
    pass

def decrease(self, i, n):
    pass

def delete(self, i):
    pass
```

Operations on Heap

1. Heapify — A process of creating a heap from an array.
2. Insertion — Process to insert an element in existing Heap [O(logn)]
3. deletion — Process of delition of element in Heap [O(logn)]
4. Peek — to find max or min element for max & min Heap.

Insertion



Class implementation (1) (2) (3) (4)

```
def insert(self, n)
    arr = self.arr
    arr.append(n)
    i = len(arr)-1
    while i>0 and arr[self.Parent(i)] > arr[i]:
        p = self.Parent(i)
        arr[i], arr[p] = arr[p], arr[i]
        i = p.
```

Process of Insertion:

- First Increase the heap size by 1, so that it can store the new element.
- Insert the new element at the end of Heap.
- This newly inserted element may disrot the properties of Heap property. So in order to keep the properties of Heap, heapify this newly inserted element following a bottom up approach.

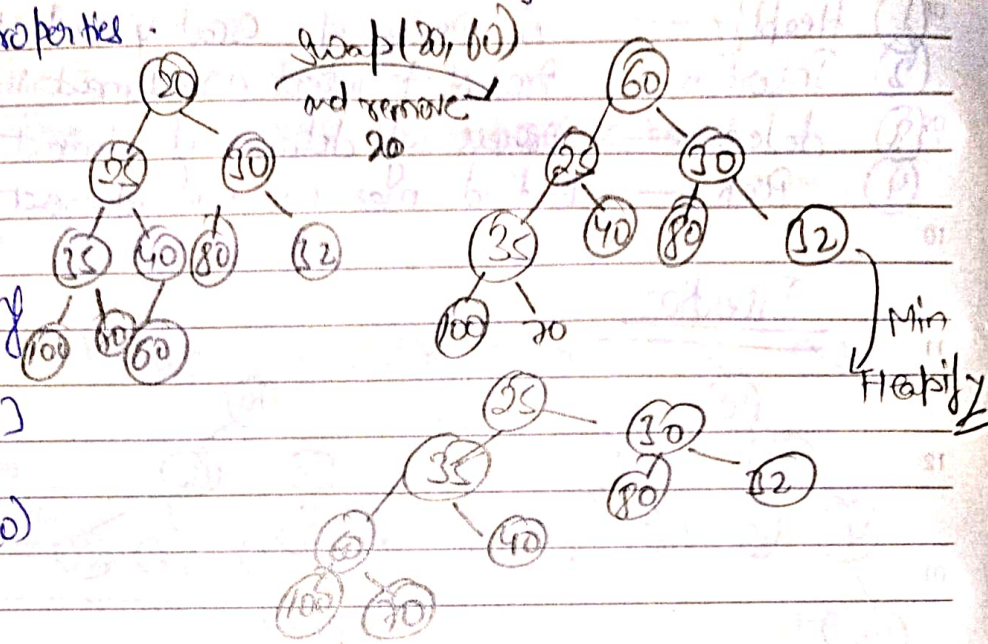
S	M	T	W	T	F	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30					

SEP 2024

Extract min Element should be removed and the remaining element should follow min Heap properties.

```
def extractMin(self):
    arr=self.arr
    n= len(arr)
    if n==0:
        return math.inf
    res = arr[0]
    arr[0] =arr[n-1]
    arr.pop()
    self.minHeapify(0)
    return res
```

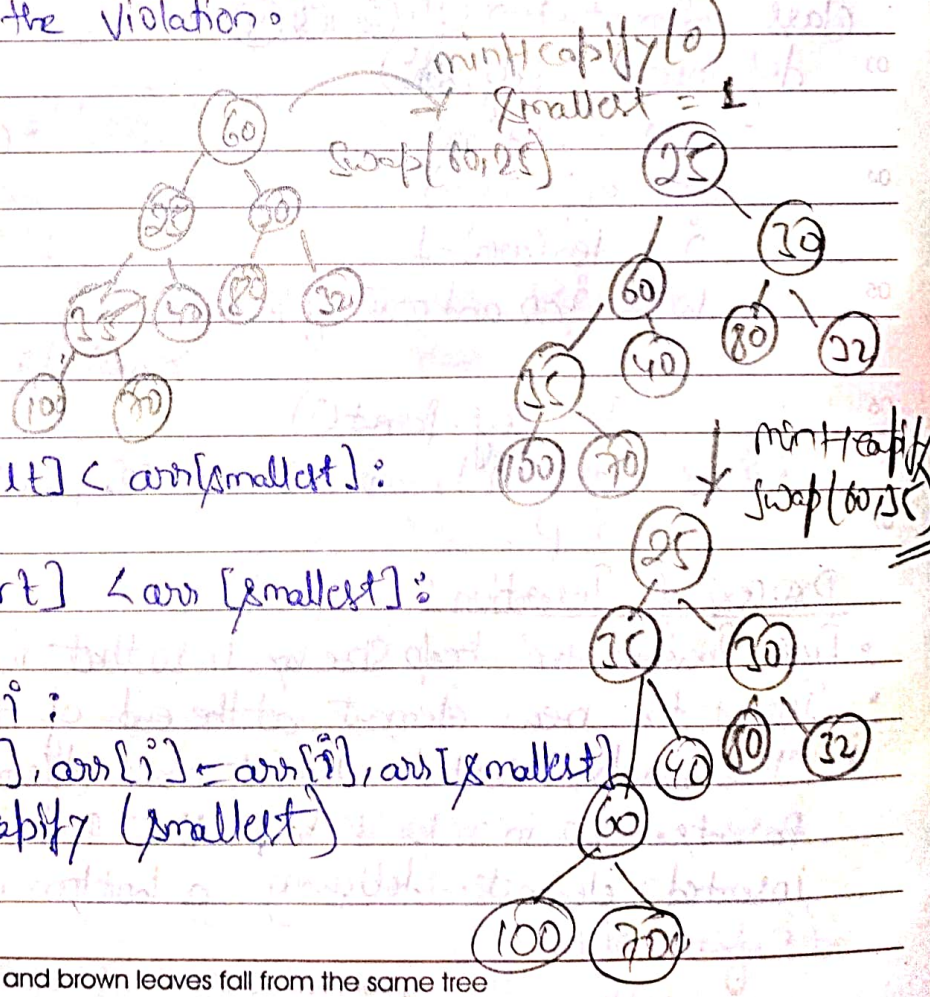
$\theta(\log n)$



Min Heapify Input is a min Heap where only the root is violating. We need to fix the violation.

```
def minHeapify(self,i):
    arr =self.arr
    lt = self.lchild(i)
    rt = self.rchild(i)
    smallest = i
    n= len(arr)
    if lt < n and arr[lt] < arr[smallest]:
        smallest = lt
    if rt <n and arr[rt] < arr[smallest]:
        smallest = rt
    if smallest != i:
        arr[smallest], arr[i] = arr[i], arr[smallest]
        self.minHeapify(smallest)
```

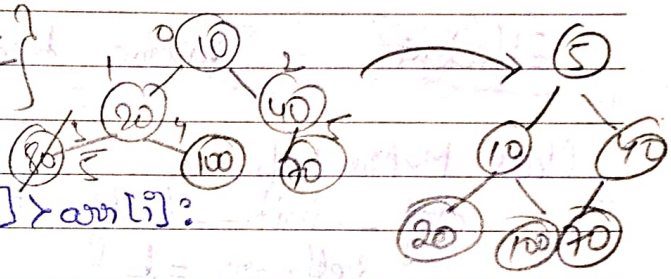
$\theta(\log n)$



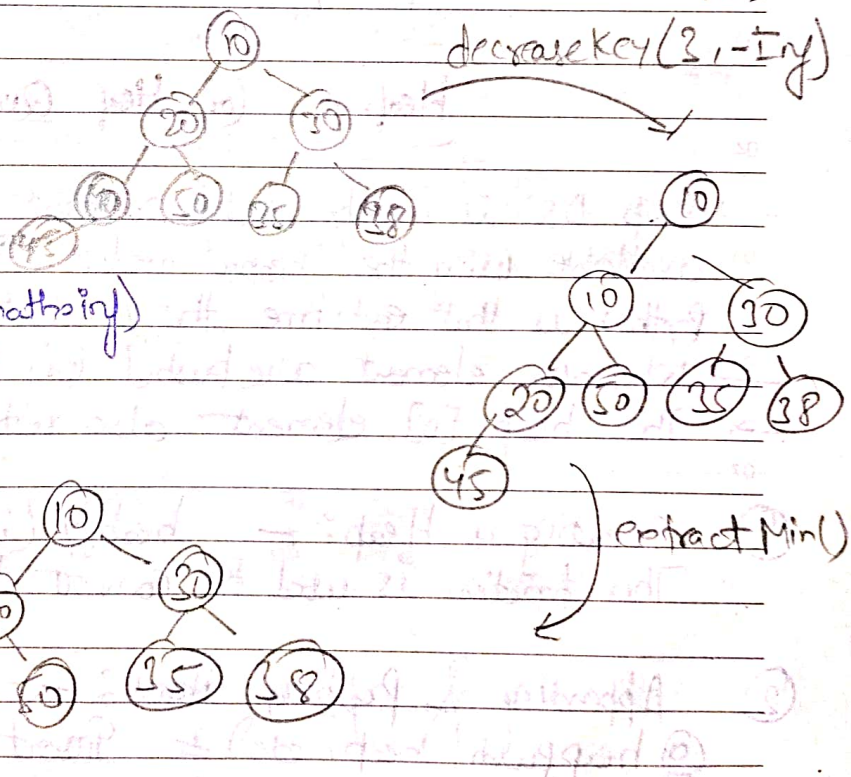
Deletion

1. Replace the root (or) element to be deleted by last element.
2. Delete the last element from the heap.
3. Since the last element is now placed at the position of the root node. So, it may not follow the heap property. Therefore heapify the last node placed at the position of root.

```
def decreaseKey(self,i,x):
    arr=self.arr
    arr[i]=x
    while i!=0 and arr[self.parent[i]]>arr[i]:
        p=self.parent(i)
        arr[i],arr[p]=arr[p],arr[i]
        i=p
```




```
def deleteKey(self,i):
    n=len(self.arr)
    if i>=n:
        return
    else:
        self.decreaseKey(i,-maths.inf)
        self.extractMin()
```



SEPTEMBER
08 '24
37th Week
252-114
SUNDAY

S	M	T	W	T	F	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30					

SEP 2024

Build Heap

Given N elements. The task is to build a Binary Heap from the given array. The Heap can be either max Heap (or) min Heap.

Naive Approach — Sorting — $O(n \log n)$

Efficient — Traverse from Bottom — $O(n)$

```
Class MyMinHeap:
    def __init__(self, l=[]):
        self.arr = l
        i = (len(l) - 2) // 2
        while i >= 0:
            self.minHeapify(i)
            i = i - 1
```

Maximum Node at height $h = \left\lceil \frac{n}{2^{h+1}} \right\rceil$

TC: $\Theta / O(n)$

Heapq (or) Heap Queue

→ Heap DS is mainly used to represent a priority queue. In Python, It is available using the 'heapq' module. The property of this data structure in Python is that each time the smallest heap element is popped (min-heap).

→ Whenever element are pushed (or) popped, heap structure is maintained.

→ The heap [0] element also return the smallest element each time.

① Creating a Heap: — heapify (iterable)
This function is used to convert the Iterable into heap data structure.

② Appending & Popping items: —
(a) heappush (heap, ele) : Insert the element.
(b) heappop (heap) : Remove and return the smallest element from Heap.

(3) Appending and popping simultaneously :-

08 (a) heappushpop (heap, ele) :- Combine the functions both push & pop.

(b) heapreplace (heap, ele) :- also insert and pops elements but the

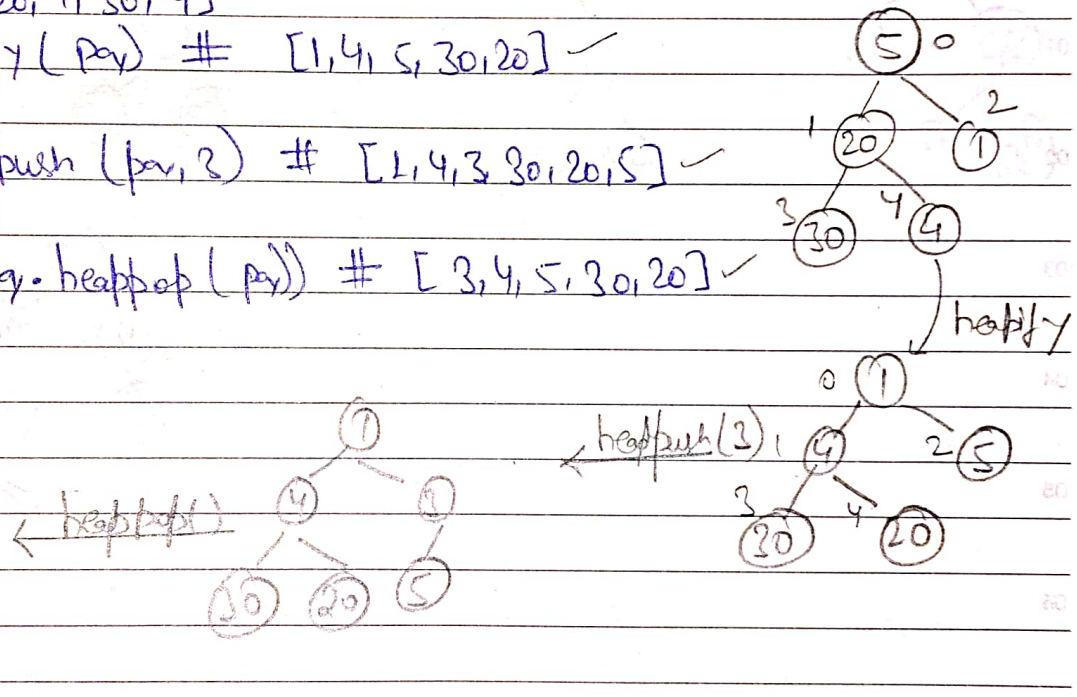
09 element is first popped then the element is pushed.

10 (4) Find the largest & smallest element from Heap :-

(a) nlargest (k, iterable, key = fun) : return the k^{th} largest element.

11 (b) nsmallest (k, iterable, key = fun) : return the k^{th} smallest element.

```
import heapq
pq = [5, 20, 1, 30, 4]
heapq.heapify(pq)  # [1, 4, 5, 30, 20]
print(pq)
heapq.heappush(pq, 3)  # [1, 4, 3, 30, 20, 5]
print(pq)
print(heapq.heappop(pq))  # [3, 4, 5, 30, 20]
print(pq)
```



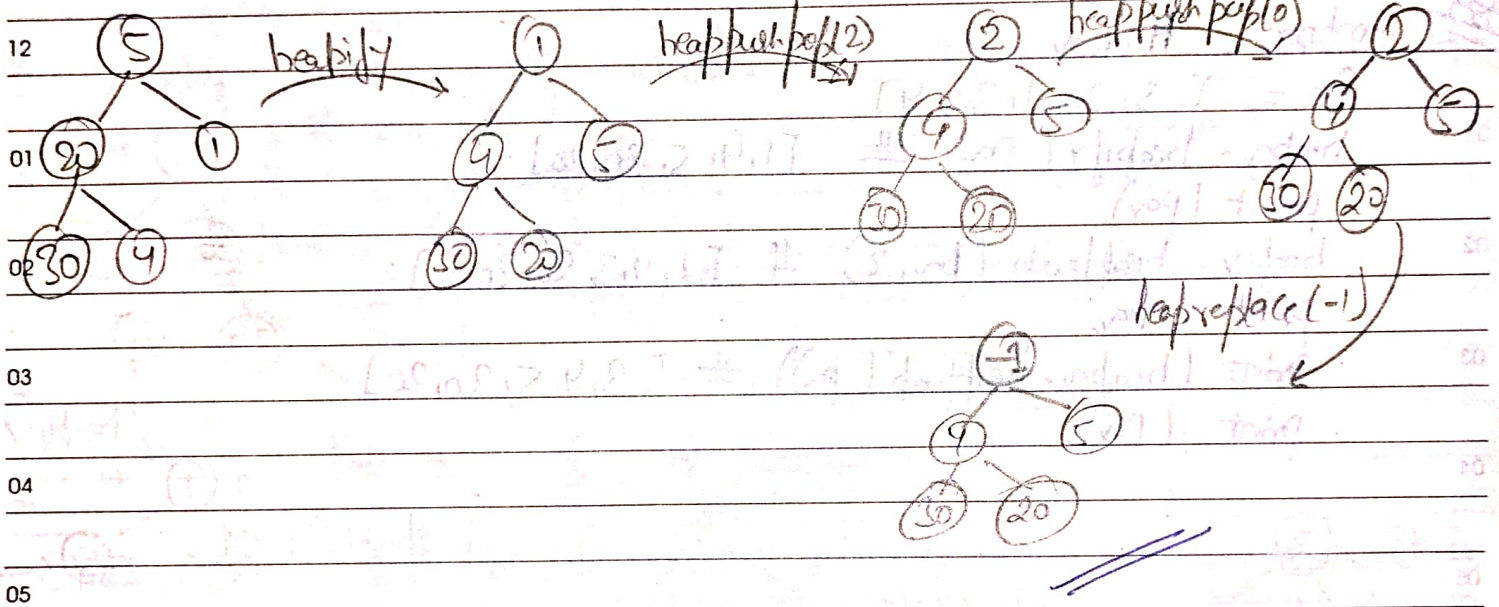

```
print(heapq.nlargest(2, pq))  # [30, 20]
print(heapq.nsmallest(2, pq))  # [1, 4]
```

SEPTEMBER
10
'24
37th Week
254-112
TUESDAY

S	M	T	W	T	F	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30					

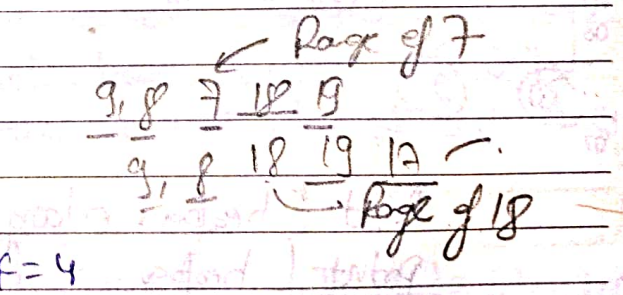
SEP 2024

```
import heapq
pq = [5, 20, 1, 30, 4]
heapq.heapify(pq)
print(heapq.heappushpop(pq, 2)) # 1
print(pq)  # [2, 4, 5, 20, 20]
print(heapq.heappushpop(pq, 0)) # 0
print(pq)  # [2, 4, 5, 30, 20]
print(heapq.heapreplace(pq, -1)) # [-1, 4, 5, 30, 20]
```



Sort a K sorted Array

i/p: arr = [9, 8, 7, 18, 19, 17] K=2
o/p: → [7, 8, 9, 17, 18, 19]



i/p: arr = [10, 9, 7, 8, 4, 30, 50, 60] K=4
o/p: → [4, 7, 8, 9, 10, 30, 50, 60]

Naive arr.sort() → O(nlogn)

Efficient

```
import heapq
def sortk(arr,k):
    n= len(arr)
    hp= arr[:k+1]
    heapq.heapify(hp)
    index = 0
    for i in range(k+1, n):
        arr[index] = heapq.heappop(hp)
        index +=1
        heapq.heappush(hp, arr[i])
    while hp:
        arr[index] = heapq.heappop(hp)
        index +=1
```

```
def PrintArray(arr):
    for i in arr:
        print(i, end=" ")
```

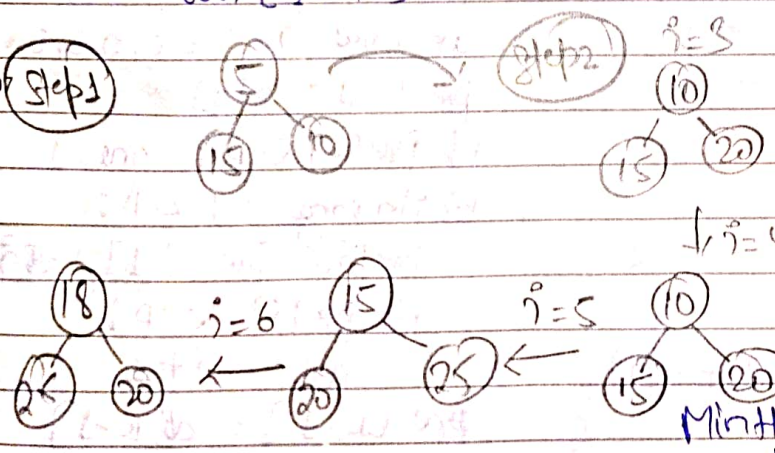
TC= O(n+klogk)

K largest Element

Given array is [1, 23, 12, 9, 20, 2, 50] and you are asked for the largest 3 element K= 3 then your program should print 50, 23, 20.

Naive
arr.sort()
arr[-1:-K]

Efficient
1. Build a Min Heap of first k items.
2. Traverse from (k+1)th element.
3. Compare current element with top of heap if smaller than top, ignore it.
4. Else remove the top element and insert the current element in the Min Heap.
5. Print contents of min Heap.



MinHeap: TC= O(k+(n-k)xlogk)
max Heap: O(n+klogn)

SEPTEMBER
12
'24
37th Week
256-110
THURSDAY

S	M	T	W	T	F	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30					

SEP 2024

```
import heapq as hq
def firstKElement(arr, size, k):
    minHeap = []
    for i in range(k):
        minHeap.append(arr[i])
    hq.heapify(minHeap)
    for i in range(k, size):
        if minHeap[0] > arr[i]:
            continue
        else:
            minHeap[0] = minHeap[-1]
            minHeap.pop()
            minHeap.append(arr[i])
            hq.heapify(minHeap)
    for i in minHeap:
        print(i, end=" ")
```

```
Sum = 10
res = 0
const.sort()                    O(nlogn)
for i in const:
    if i <= Sum:
        Sum -= i
        res += 1
    else:
        break
print(res)
```

M-2

M-1

```
def number(a, n, p, k):
    a.sort()
    pre = []
    for i in range(n): pre.append(0)
    ans, val, i, j = 0, 0, 0, 0
    pre[0] = a[0]
    if pre[0] <= p: ans = 1
    for i in range(1, k-1):
        pre[i] = pre[i-1] + a[i]
        if pre[i] <= p:
            ans = i+1
    pre[k-1] = a[k-1]
    for i in range(k-1, n):
        if i >= k:
            pre[i] += pre[i-k] + a[i]
        if pre[i] <= p:
            ans = i+1
    return ans
```

Purchase maximum items.

① cost = [1, 12, 5, 111, 200], sum = 10
o/p = 2 ✓

② cost = [20, 10, 5, 30, 100], sum = 35
o/p = 3

TC <= O(n) + O(res x logn)

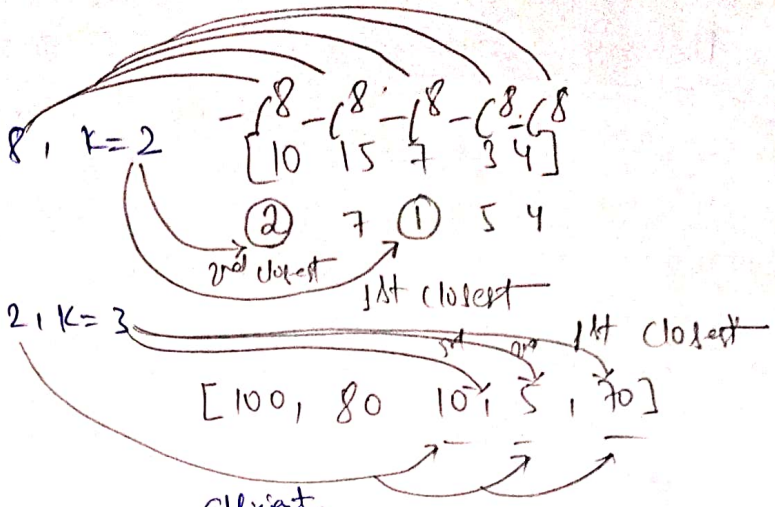
K Closest Element

i/p : arr = [10, 15, 7, 3, 4] , n = 8, k = 2

o/p : 7 10

i/p : arr = [100, 80, 10, 5, 70] , n = 2, k = 3

o/p : 5 10 70



Naive

```
def kclosest(arr, k, n):
    for i in range(k):
        mi = 0
        for i in range(1, len(arr)):
            if (abs(arr[mi] - n) > abs(arr[i] - n)):
                mi = i
        print(arr[mi], end = " ")
        arr.pop(mi)
```

O(nk)

Efficient

```
def kclosest(arr, n, k):
    h = []
    for i in range(k):
        heapq.heappush(h, (-abs(arr[i] - n), i))
    for i in range(k, n):
        curr = -abs(arr[i] - n)
        p, pi = h[0]
        if curr > p:
            heapq.heappop(h)
            heapq.heappush(h, (curr, i))
    while h:
        p, pi = heapq.heappop(h)
        print(arr[pi], end = " ")
```

TC = (n + klogk)

Merge K sorted array

i/p : arr[][] = {{10, 20, 30}, {5, 15}, {1, 9, 11, 18}}

o/p : res[] = {1, 5, 9, 10, 11, 15, 18, 20, 30}

Naive soln ①

① Put all element in res[]

② Sort res[]

TC = O(nk log nk)

where k is No. of input arrays
n is maximum No. of element in an array.

Naive soln ②

① Copy first array to res[]

② Do following for remaining arrays starting from the second array merge current arr into res[].

TC = O(nk²)

Efficient ①

```
def mergeK(arr):
    res = []
    h = []
    for i in range(len(arr)):                          # O(klogk)
        heapq.heappush(h, (arr[i][0], i, 0))
    while h:                                           # O(n x log k)
        val, ap, vp = heapq.heappop(h)
        res.append(val)
        if (vp + 1 < len(arr[ap])):
            heapq.heappush(h, (arr[ap][vp + 1], ap, vp + 1))
    return res
```

O = (nklogk)

Median of a Stream

i/p: arr[] = {25, 7, 10, 15, 20}

25 16 10 12.5 15

Sequence	median
{25}	25
{25, 7}	(25+7)/2 = 16
{25, 7, 10}	10
{25, 7, 10, 15}	(10+15)/2 = 12.5
{25, 7, 10, 15, 20}	15

i/p: arr[] = {20, 10, 30, 7}

20 15 20 15

Median means the element, which is smaller than half of the element and greater than half of the element.

```
def streamMedian(arr):
    s, g = [], []
    for i in range(len(arr)):
        heappush(s, -arr[i])
        heappush(g, -heappop(s))
        if len(g) > len(s):
            heappush(s, -heappop(g))
        if len(g) < len(s):
            print(-s[0])
        else:
            print((g[0] - s[0])/2)
```

DO the following for every item n:

(a) s.push(n)

(b) g.push(s.pop())

(c) if size(g) > size(s)
        s.push(g.pop())

(d) if size(s) > size(g)
        Print(s.top())
    else Print((s.top() + g.top())/2)