STACK,QUEUES, HEAP

Implement stack using array:

STACK: LIFO (Last in first out)
Operation on stack:

- push()
- pop()
- top()
- size()
- isempty()

Take a pointer ‘top’ which initially points to -1

Implementation of above operation: ARR= [2] , TOP=-1

- **push(x) :** Increment the top pointer and put the element at top+1
  def push(x):
  top++
  arr [top] = x

- **pop() :** return the element at arr[top] and remove it and decrement the top pointer.
  Def pop():
  x= arr[top]
  top - -
  Return x

- **top():** returns the element on the top without removing it Def top():
  x= arr[top]
  Return x

- **size():** returns the size of stack, return the index+1 where top is pointing to.
  Def size():
  Return top+1

- **isempty():** check whether the stack is empty or not
  Def empty():
    If top==-1:
      Return true
    else false

Reference to read more about it and its practical implementation: [Introduction to Stack - Data Structure and Algorithm Tutorials - GeeksforGeeks]

Time complexity of all the operation is O(1)

Q: Balanced parentheses:

```
class Solution:
    def isValid(self, s: str) -> bool:
        stack=[]
        for i in s:
            if i=='{' or i=='(' or i=='[':
                stack.append(i)
            elif i=='}' and len(stack)>0 and stack[-1]=='{':
                stack.pop()
            elif i==')' and len(stack)>0 and stack[-1]=='(':
                stack.pop()
            elif i==']' and len(stack)>0 and stack[-1]=='[':
                stack.pop()
            else:
                return False

        if stack:
            return False
        else:
            return True
```

Q: Implement stack using linklist:
INTUTION: as we know stack is LIFO so when push push from front of list and when pop, pop from front so the element which is last pushed get popped first , hence following the policy of LIFO. so Overall, the time complexity of the push, pop, and isEmpty operations in a stack implemented using a linked list is **O(1)** or constant time.

```
#Function to push an integer into the stack.
def __init__(self):
    self.head=None

def push(self, data):
    newnode= StackNode(data)
    if self.head==None:
        self.head=newnode
    else:
        newnode.next= self.head
        self.head=newnode


#Function to remove an item from top of the stack.
def pop(self):
    if self.head==None:
        return -1
    else:
        x= self.head.data
        self.head=self.head.next
        return x
```

Implement queues using array:

Queue : FIFO (First in first out)

Let say the size of array is n.
Take two pointer ‘front’ and ‘rear’ and a variable count to keep track of how many elements are present in array/queue:

Operations on queue:

- **enqueue():** Inserts an element at the end of the queue i.e. at the rear end.
- **dequeue():** This operation removes and returns an element that is at the front end of the queue.
- **front():** This operation returns the element at the front end without removing it.
- **rear():** This operation returns the element at the rear end without removing it.

- **isEmpty():** This operation indicates whether the queue is empty or not.
- **isFull():** This operation indicates whether the queue is full or not.
- **size():** This operation returns the size of the queue i.e. the total number of elements it contains.

Time complexity of all the operation is O(1)

Reference to read more about it and its practical implementation: [Introduction and Array Implementation of Queue - GeeksforGeeks]

Array implementation Of Queue:

For implementing queue, we need to keep track of two indices, front and rear. We enqueue an item at the rear and dequeue an item from the front. If we simply increment front and rear indices, then there may be problems, the front may reach the end of the array. The solution to this problem is to increase front and rear in circular manner.

Initially count=0, front=0, rear=n-1

```
Def enque(x):
    If count==n:
        Not able to push element , queue is full
    rear= (rear+1)%n
    Arr[rear]= x
    Count++
```

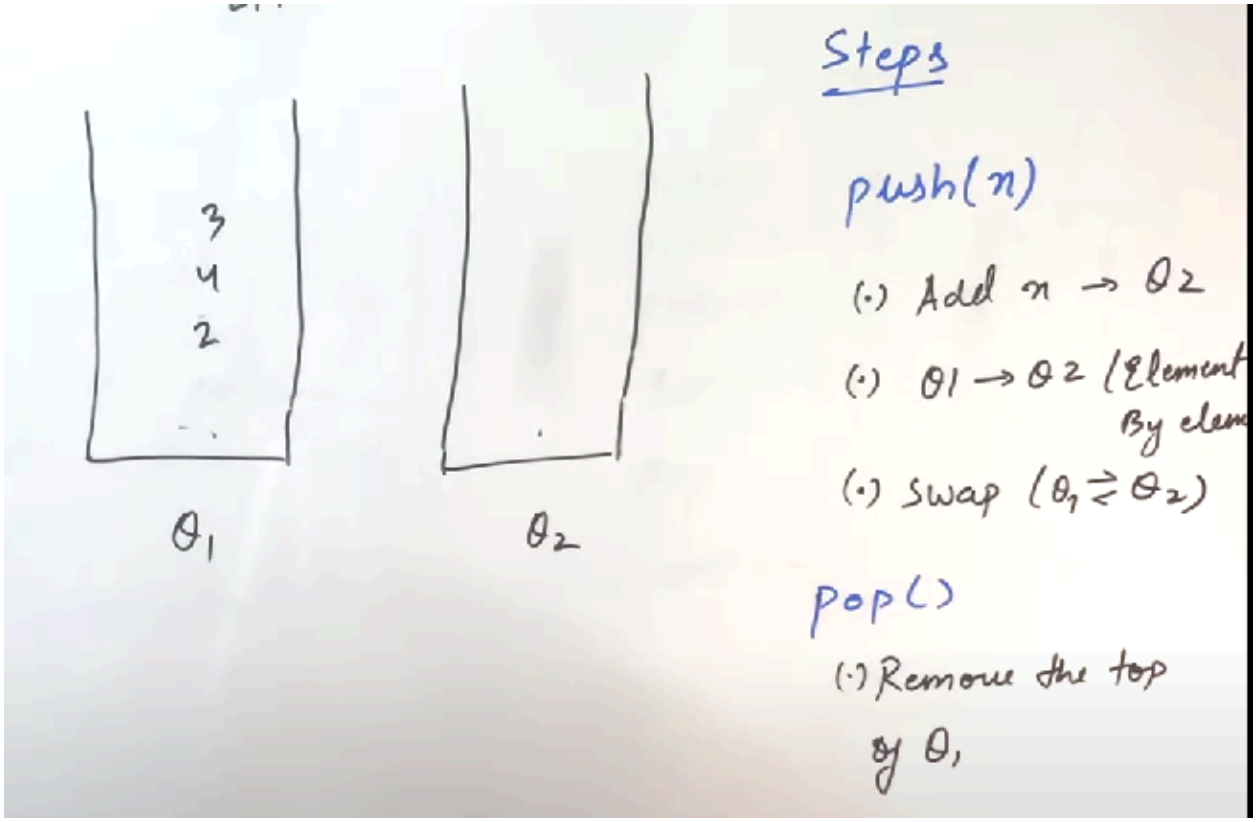
—----------

```
Def deque():
    If count==0:
        Nothing to pop, queue is empty

    x=arr[front]
    front = (front +1) %n;
```

```
    Count- -
    Return x
—----------------------------------------------------------------------------

Def isempty():
     If count==0:
          Return true
     Else:
          Return false
—----------------------------------------------------------------------------

Def front():
     If isempty():
            Queue is empty
     Return arr[front]
—----------------------------------------------------------------------------

Def rear():
    If isempty():
            Queue is empty
     Return arr[rear]
—----------------------------------------------------------------------------

Def isfull():
    If count==n:
          Return queue is full
—----------------------------------------------------------------------------
```

Implementing stack using queues:

Stack can be implemented by two queues



Take two queues Q1 and Q2 ,

Time Complexity: O(N), Space Complexity: O(2N)

- **push(x) encountered:**
  1. enque x to Q2
  2. Deque element by element from Q1 and enque in Q2
  3. Swap Q1 and Q2(names during implementation)

- **pop() encountered:**
  1. Deque from Q1

```
class MyStack:

    def __init__(self):
        self.q1=[]
        self.q2=[]

    def push(self, x: int) -> None:
        #to push element in the stack , push it in q2 and then copy all element from q1
        to q2 and then swap names of q1 and q2 so that q2 is now empty queue
        self.q2.append(x)
        while(len(self.q1)!=0):
            x1= self.q1.pop(0)
            self.q2.append(x1)
        self.q1, self.q2= self.q2,self.q1

    def pop(self) -> int:
        #to pop element we pop from q1 which is having the elements
        return self.q1.pop(0)


    def top(self) -> int:
        return self.q1[0]


    def empty(self) -> bool:
        if len(self.q1)==0:
            return True
        return False
```

Implementing stack using single queue:

Time Complexity: O(N) Space Complexity: O(N)

- **push():**

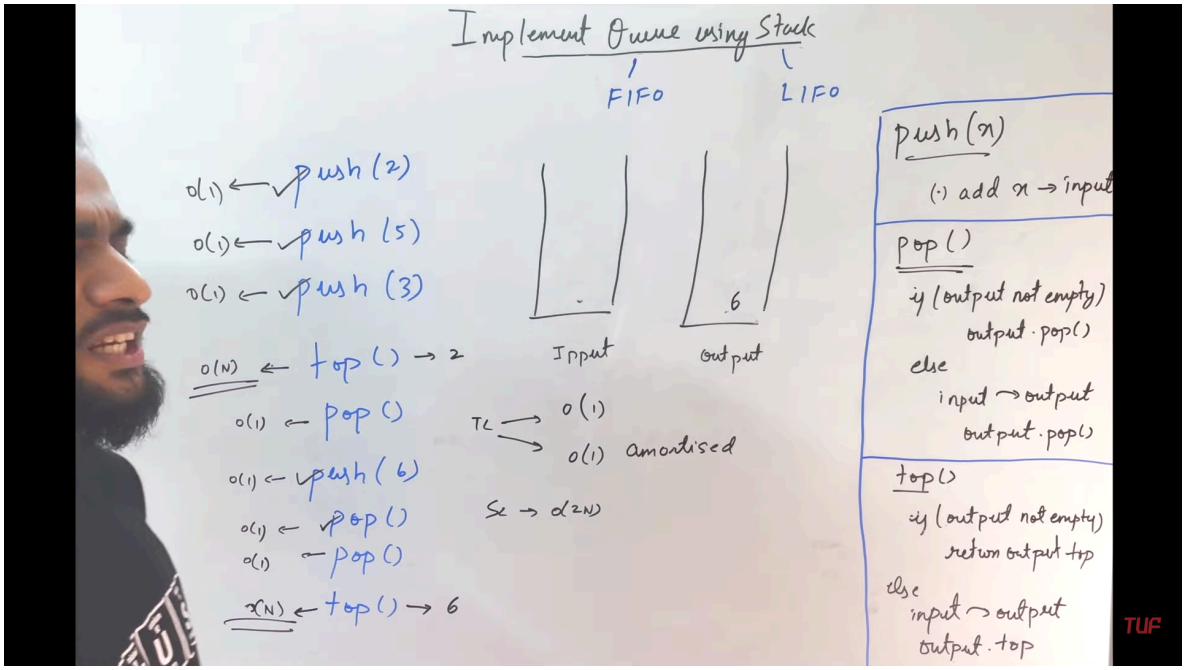
To implement it using single queue, first take a variable, named c=0
Enqueue the element in queue, as q.append(x)
And to ensure that if we dequeue the element is pop from q.pop(0) be the element who is inserted last . as stack is Last in first out.
So after appending element to queue we run a while loop which pop all elements before that element from front and append it to rear of queue which maintains it order for pop .

- **pop():**
  Simply dequeue element from front . pop(0)

```python
class MyStack:

    def __init__(self):
        self.q=[]

    def push(self, x: int) -> None:
        self.q.append(x)
        c=len(self.q)
        while(c>1):
            x1=self.q.pop(0)
            self.q.append(x1)
            c-=1

    def pop(self) -> int:
        return self.q.pop(0)


    def top(self) -> int:
        return self.q[0]


    def empty(self) -> bool:
        if len(self.q)==0:
            return True
        return False
```

Implementing queue using 2 stack:

A method by which we get **TC of pop as well as top in O(1) amotized. And push O(1)**




```python
class MyQueue:
    def __init__(self):
        self.stack1=[]
        self.stack2=[]

    def push(self, x: int) -> None:
        self.stack1.append(x)

    def pop(self) -> int:
        if(len(self.stack2)!=0):
            return self.stack2.pop()
        while(len(self.stack1)!=0):
            x= self.stack1.pop()
            self.stack2.append(x)
        return self.stack2.pop()

    def peek(self) -> int:
        if(len(self.stack2)!=0):
            return self.stack2[-1]
        while(len(self.stack1)!=0):
            x= self.stack1.pop()
            self.stack2.append(x)
        return self.stack2[-1]

    def empty(self) -> bool:
        if len(self.stack1)==0 and len(self.stack2)==0:
            return True
        return False
```

Sort a Stack

```
 def sortStack(stack):
    if len(stack) <= 1:
        return stack

    temp = stack.pop()
    stack = sortStack(stack)
    stack = insertInSortedOrder(stack, temp)

    return stack

def insertInSortedOrder(stack, element):
    if len(stack) == 0 or element >= stack[-1]:
        stack.append(element)
        return stack

    temp = stack.pop()
    stack = insertInSortedOrder(stack, element)
    stack.append(temp)

    return stack
```

Q: Previous smaller element than current. /Nearest smaller element.

we find out the smaller element than current which is nearest to it in O(n) time and O(n) space.

For Example

```
Input 1:
    A = [4, 5, 2, 10, 8]
Output 1:
    G = [-1, 4, -1, 2, 2]
Explaination 1:
    index 1: No element less than 4 in left of 4, G[1] =
-1
    index 2: A[1] is only element less than A[2], G[2] =
A[1]
    index 3: No element less than 2 in left of 2, G[3] =
-1
    index 4: A[3] is nearest element which is less than
A[4], G[4] = A[3]
    index 4: A[3] is nearest element which is less than
A[5], G[5] = A[3]
```

```python
class Solution:
    # @param A : list of integers
    # @return a list of integers
    def prevSmaller(self, A):
        stack=[]
        ans=[]
        for i in range(len(A)):
                while(len(stack)!=0 and stack[-1]>=A[i]):
                    stack.pop()
                if len(stack)==0:
                    ans.append(-1)
                else:
                    ans.append(stack[-1])
                stack.append(A[i])
        return ans
```

Q: Next smaller element than current.

```
Input: 'a' = [4, 7, 8, 2, 3, 1]

Output: Modified array 'a' = [-1, -1,
2, -1, 1, -1]
```

```python
def nextSmallerElement(arr,n):
    stack=[]
    ans=[0]*n
    for i in range(n-1,-1,-1):
            while(len(stack)!=0 and stack[-1]>=arr[i]):
                stack.pop()
            if len(stack)==0:
                ans[i]=-1
            else:
                ans[i]=stack[-1]
            stack.append(arr[i])
    return ans
```

Q: Next greater element: Bruteforce approach:

Input: nums1 = [4,1,2], nums2 = [1,3,4,2]

Output: [-1,3,-1]

```python
class Solution:
    def nextGreaterElement(self, nums1: List[int], nums2: List[int]) -> List[int]:
        stack=[]
        d={}
        for i in range(len(nums2)-1,-1,-1):
            while(len(stack)!=0 and stack[-1]<nums2[i]):
                stack.pop()
            if len(stack)!=0 and stack[-1]>nums2[i]:
                d[nums2[i]]=stack[-1]
            else:
                d[nums2[i]]=-1
            stack.append(nums2[i])

        print(d)
        ans=[]
        for i in nums1:
            if i in d:
                ans.append(d[i])
        return ans
```

Q: Implementing min stack in O(1) time but O(2n) space:

```
class MinStack:
    def __init__(self):
        self.st=[]

    def push(self, val: int) -> None:
        if len(self.st)==0:
            self.st.append([val,val])
        else:
            x=self.st[-1][1]
            if x<val:
                self.st.append([val,x])
            else:
                self.st.append([val,val])

    def pop(self) -> None:
        self.st.pop()

    def top(self) -> int:
        return self.st[-1][0]

    def getMin(self) -> int:
        return self.st[-1][1]
```

There exist more **modified version of it in O(1) time and O(n) space** .
Go and watch video.

Q: Next greater element II:
There are two variant for it. One is simple array and another is circular array:
(1) Next Greater Element II - LeetCode

For circular array

Input: nums = [1,2,1]
Output: [2,-1,2]
Explanation: The first 1's next greater number is 2;
The number 2 can't find next greater number.
The second 1's next greater number needs to search circularly, which is also 2.

```
class Solution:
    def nextGreaterElements(self, nums: List[int]) -> List[int]:
        n=len(nums)
        st=[]
        ans=[0]*n
        for i in range(2*n-1,-1,-1):
            while(len(st)!=0 and st[-1]<=nums[i%n]):
                st.pop()
            if i<n:
                if (len(st)!=0):
                    ans[i] = st[-1]
                else:
                    ans[i]=-1

            st.append(nums[i%n])
        return ans
```

Q: Sliding Window maximum:

Intuition: We take a deque in which we keep track of indexes which is less than i-k+1. The deque contains the index of element in decreasing order of the value. Whenever the index goes out of range we pop element from front and whenever we need to chck that the element at rear first to pop all indexes which has elements less than the current element so that latex max value index is at front of deque.

TC: O(N) +O(N) = O(N)

SC: O(k) = Window size

```
class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        n=len(nums)
        ans=[]
        deque=[]
        if n<=k:
            return [max(nums)]
        for i in range(n):
            if len(deque)==0:
                deque.append(i)
            while(len(deque)!=0 and deque[0]<i-k+1):
                deque.pop(0)
            while(len(deque)!=0 and nums[deque[-1]]<=nums[i]):
                deque.pop()
            deque.append(i)
            if i>=k-1:
                ans.append(nums[deque[0]])
        return ans
```

Q: Rotten oranges:

Time Complexity: O (n x n) x 4

Reason: Worst-case – We will be making each fresh orange rotten in the grid and for each rotten orange will check in 4 directions

Space Complexity: O (n x n)

Reason: worst-case – If all oranges are Rotten, we will end up pushing all rotten oranges into the Queue data structure

```
class Solution:
    def orangesRotting(self, grid: List[List[int]]) -> int:
        m= len(grid)
        n= len(grid[0])
        visited= [[-1]*n for i in range(m)]
        queue=[]
        fresh=0              # Keep track of the number of fresh oranges

        # Find initial rotten oranges and mark them as visited and append it in queue with time stamp 0
        for i in range(m):
            for j in range(n):
                if grid[i][j]==2:
                    queue.append([i,j,0])
                    visited[i][j]=2
                elif grid[i][j]==1:
                    fresh+=1
        # No fresh oranges initially, so no time needed for rotting
        if fresh==0:
            return 0
        tm=0
        drow=[-1,0,1,0]
        dcol=[0,1,0,-1]
        while(len(queue)!=0):
            r= queue[0][0]
            c= queue[0][1]
            t= queue[0][2]
            tm= max(t,tm)
            queue.pop(0)
            for i in range(4):
                nrow= r+drow[i]
                ncol= c+dcol[i]
                if nrow>=0 and nrow<m and ncol>=0 and ncol<n and visited[nrow][ncol]!=2 and grid[nrow][ncol]==1:
                    queue.append([nrow,ncol,tm+1])
                    visited[nrow][ncol]=2
                    fresh-=1

         # There are fresh oranges left, impossible to rot all
        if fresh>0:
            return -1
        return tm
```

Q: Celebrity problem using stack .

PS: There is also a solution exists of graph.

The below solution gives TLE error:

```
def findCelebrity(n, knows):

    arr= [[0]*n for i in range(n)]
    for i in range(n):
        for j in range(n):
            if i==j:
                continue
            if(knows(i,j)):
                arr[i][j]=1
    st=[]
    for i in range(n):
        st.append(i)

    while(len(st)>1):
        a= st.pop()
        b=st.pop()
        if knows(a,b):
            st.append(b)
        else:
            st.append(a)
    if not st:
        return -1
    potential= st.pop()
    for i in range(n):
        if arr[potential][i]==1:
            return -1
    for i in range(n):
        if i!= potential and arr[i][potential]==0:
            return -1
    return potential
```

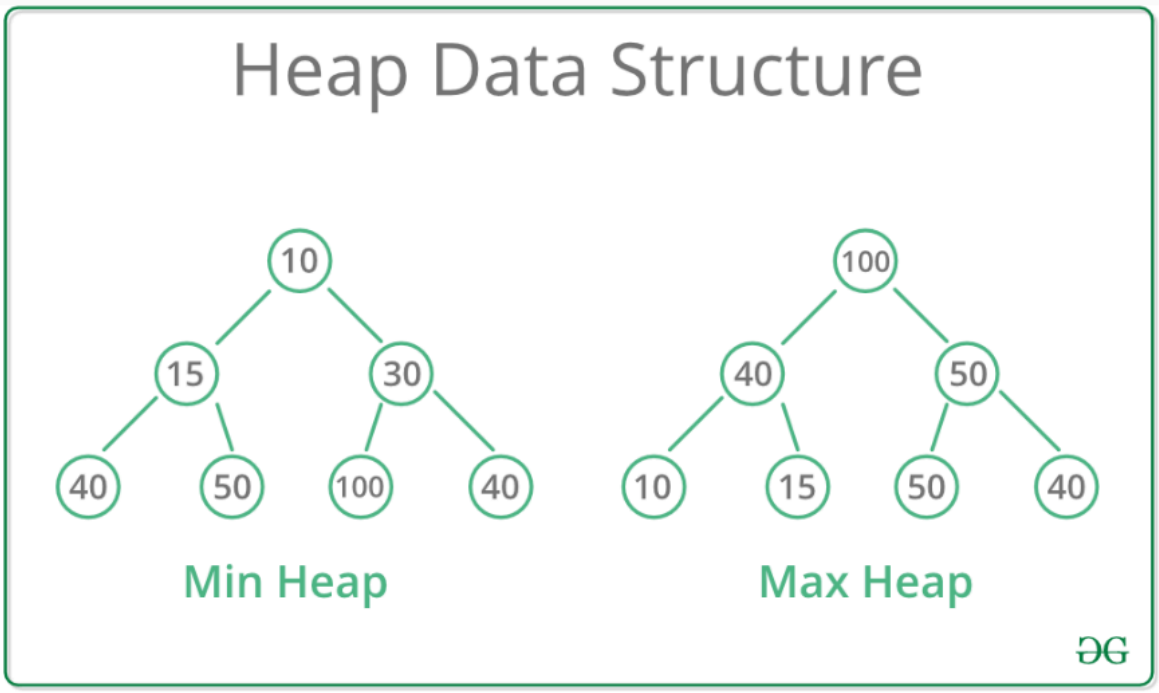
Improved solution in O(2n) time and O(1) space complexity.

```python
def findCelebrity(n, knows):
    potential = 0
    for i in range(1, n):
        if knows(potential, i):
            potential = i

    for i in range(n):
        if i != potential and (knows(potential, i) or not knows(i, potential)):
            return -1

    return potential
```

HEAP:

What is Heap Data Structure?

> *A Heap is a special Tree-based data structure in which the tree is a complete binary tree.*



Heap Data Structure

Operations of Heap Data Structure:

- **Heapify:** a process of creating a heap from an array.
- **Insertion:** process to insert an element in existing heap time complexity O(log N).
- **Deletion:** deleting the top element of the heap or the highest priority element, and then organizing the heap and returning the element with time complexity O(log N).
- **Peek:** to check or find the first (or can say the top) element of the heap.

Types of Heap Data Structure

Generally, Heaps can be of two types:

1. **Max-Heap**: In a Max-Heap the key present at the root node must be greatest among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.
2. **Min-Heap**: In a Min-Heap the key present at the root node must be minimum among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.

Type of questions:

- Kth smallest element/ k smallest element
- Kth largest element
- Sort the k sorted array, or nearly sorted array/ Merge k sorted array.
- K closest elements etc
- K frequent element
- Frequent sort
- K closest point to origin

Q: K largest element:

As we can do sorting and find the kth element by its index but it requires O(nlogn) time, so we can reduce it to O(nlogk) to only sort the k element at a time. We iterate through n element and only sort k element so Tc here is O(nlogk) . We can implement it by heap.
Sc: O(k)

As we know the property of min heap that min heap top/root having smallest value. So we insert element one by one into min heap upto k size, when size exceeds pop element from root as it is min element so not required in finding kth largest element.at last when all element traversed and the min heap contain the root as min element which is our kth largest element. And if we print all the left element in min_heap, that is our k largest elements.

```python
#User function Template for python3
import heapq
class Solution:

    def kLargest(self,arr, n, k):
        min_heap=[]
        for i in range(n):
            heapq.heappush(min_heap,arr[i])
            if len(min_heap)>k:
                heapq.heappop(min_heap)
        ans=[]
        while(len(min_heap)!=0):
            ans.append(heapq.heappop(min_heap))
        ans.reverse()
        return ans
```

Q: Kth smallest: Similar approach as above, we use max_heap here as we need to find out the kth smallest element.

TC: O(nlogk)
SC: O(k)

```python
#User function Template for python3
import heapq
class Solution:
    def kthSmallest(self,arr, l, r, k):
        '''
        arr : given array
        l : starting index of the array i.e 0
        r : ending index of the array i.e size-1
        k : find kth smallest element and return using this function
        '''

        #in maxheap we push element in negative order and when pop negate it to make positive

        max_heap=[]
        for i in range(l,r+1):
            heapq.heappush(max_heap,-arr[i])
            if len(max_heap)>k:
                heapq.heappop(max_heap)
        ans= -heapq.heappop(max_heap)
        return ans
```

Q: Sort k sorted array:

Tc: O(nlogk)
SC: O(k)

Given an array of n elements, where each element is at most k away from its target position, you need to sort the array optimally.

As it is already given that any element is k away, [-k, +k] index from its target position.

```python
#User function Template for python3
import heapq
class Solution:

    #Function to return the sorted array.
    def nearlySorted(self,a,n,k):
        min_heap=[]
        ans=[]
        for i in range(n):
            heapq.heappush(min_heap,a[i])
            if len(min_heap)>k:
                ans.append(heapq.heappop(min_heap))

        while(len(min_heap)!=0):
            ans.append(heapq.heappop(min_heap))
        return ans
```

Q: Top Frequent Element:

TC: O(nlogk+n)

SC: O(n)

```python
import heapq
class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        ans=[]
        map={}
        min_heap=[]
        for i in range(len(nums)):
            if nums[i] in map:
                map[nums[i]]+=1
            else:
                map[nums[i]]=1
        for key, val in map.items():
            heapq.heappush(min_heap,[val,key])
            if len(min_heap)>k:
                heapq.heappop(min_heap)
        while(len(min_heap)!=0):
            temp=heapq.heappop(min_heap)
            ans.append(temp[1])
        return ans
```

Q: K closest element:

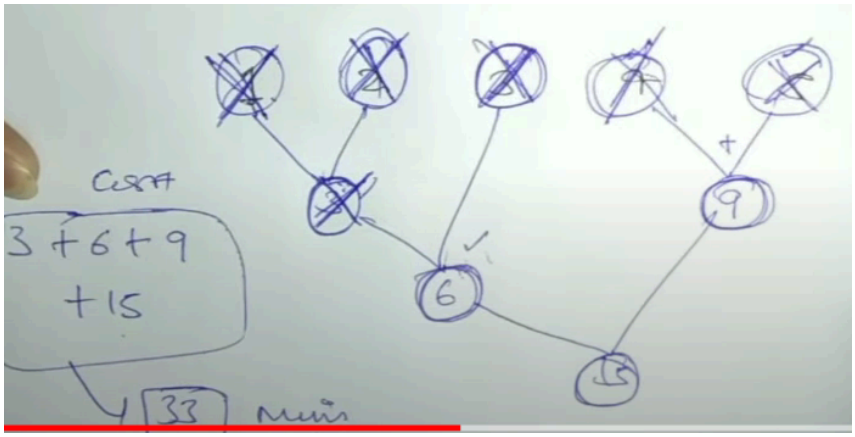
Time complexity: O(nlogk)

Space complexity: O(k)

```python
import heapq
from typing import List

class Solution:
    def findClosestElements(self, arr: List[int], k: int, x: int) -> List[int]:
        max_heap = []
        for num in arr:
            diff = abs(num - x)
            heapq.heappush(max_heap, (-diff, -num))
            if len(max_heap) > k:
                heapq.heappop(max_heap)

        closest = []
        while max_heap:
            closest.append(-heapq.heappop(max_heap)[1])

        closest.sort()
        return closest
```

Q: Connect all given rope into one rope to minimize the cost. As cost is same as length of rope.

Intuition: We always connect those rope which is having min value and then push back to min heap.




```python
#User function Template for python3
import heapq
class Solution:
    #Function to return the minimum cost of connecting the ropes.
    def minCost(self,arr,n):
        sum=0
        min_heap=[]
        for i in range(n):
            heapq.heappush(min_heap,arr[i])

        while(len(min_heap)!=1):
            temp1=heapq.heappop(min_heap)
            temp2=heapq.heappop(min_heap)
            temp= temp1+temp2
            sum+=temp
            heapq.heappush(min_heap,temp)

        return sum
```

Q: Maximum sum combination:

By brute force approach we can easily solve it in O(n^2 log n) time as it requires sum of every element in A with every element in B and then placing all the sum into another array and then sort it which take n^2logn time and then return the k largest by iterating k element. Overall the tc of it is O(n^2logn + k)

Lets see optimal approach which we can do it by use of maxheap. As here first we sort the array A and B , from our intuition we know that if we sum the maximum element the sum is also the max. With this intuition we add last two element and push it in max_heap. And then try other cross element like last of A and second last of B or last of B and second last of A. By this we do it k times as we need k largest sum. We keep track of indices in set as when any index encountered again then it will again not be calculated.

```python
# } Driver Code Ends
#User function Template for python3
import heapq
class Solution:
    def maxCombinations(self, N, K, A, B):
        A.sort()
        B.sort()
        s=set()
        max_heap=[]
        ans=[]
        heapq.heappush(max_heap,(-(A[N-1]+B[N-1]),N-1,N-1))
        while K:
            sum, x, y = heapq.heappop(max_heap)
            ans.append(-sum)
            if (x-1,y) not in s:
                heapq.heappush(max_heap,(-(A[x-1]+B[y]),x-1,y))
                s.add((x-1,y))
            if (x,y-1) not in s:
                heapq.heappush(max_heap,(-(A[x]+B[y-1]),x,y-1))
                s.add((x,y-1))
            K=K-1

        return ans
```

Time complexity:
Sorting A and B: The sorting operation takes O(N log N) time for each array, which is the dominant factor. So the total time complexity for sorting is O(N log N + N log N) = O(N log N).

Generating K combinations: The while loop runs K times. In each iteration, we perform two heap operations: heappop and heappush, both of which take O(log N) time. Therefore, the time complexity for generating K combinations is O(K log N).

Hence, the overall time complexity of the maxCombinations method is O(N log N + K log N), which can be simplified to O((N + K) log N).
Space complexity:
Sorting A and B: The sorting operation doesn't require any additional space since it is performed in-place.

Additional data structures: We use a set s to store visited indices and a max heap max_heap to keep track of the maximum combinations. The size of both s and max_heap can grow up to N elements. Therefore, the space complexity for additional data structures is O(N).

Hence, the overall space complexity is O(N).

Q: Find median from data stream:

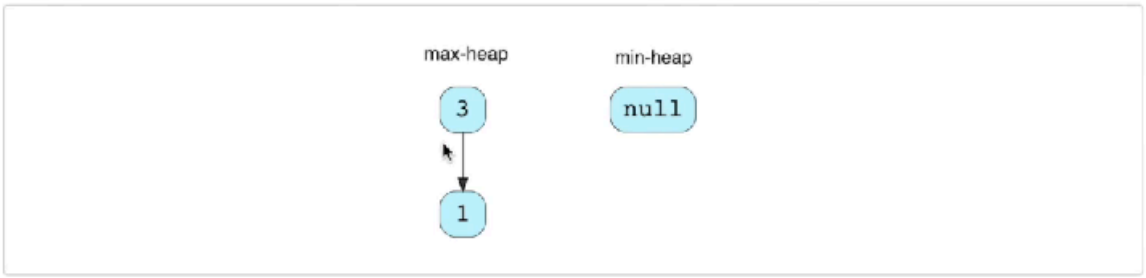
The best data structure that comes to mind to find the smallest or largest number among a list of numbers is a Heap. Let's see how we can use a heap to find a better algorithm.

1. We can store the first half of numbers (i.e., `smallNumList`) in a **Max Heap**. We should use a **Max Heap** as we are interested in knowing the largest number in the first half.
2. We can store the second half of numbers (i.e., `largeNumList`) in a **Min Heap**, as we are interested in knowing the smallest number in the second half.
3. Inserting a number in a heap will take $O(logN)$, which is better than the brute force approach.
4. At any time, the median of the current list of numbers can be calculated from the top element of the two heaps.

Let's take the Example-1 mentioned above to go through each step of our algorithm:

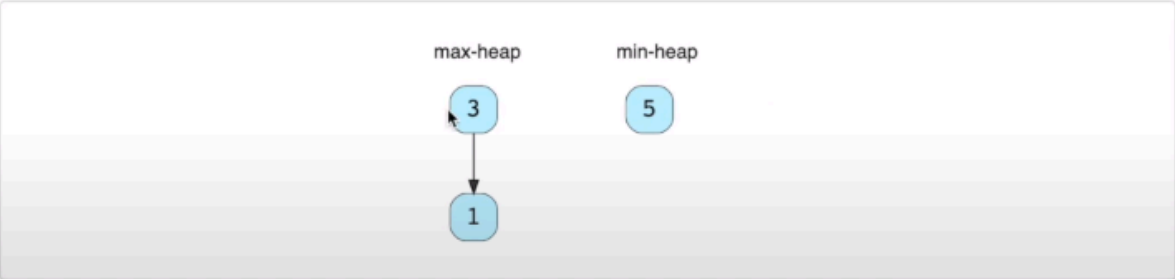
1. `insertNum(3)`: We can insert a number in the **Max Heap** (i.e. first half) if the number is smaller than the top (largest) number of the heap. After every insertion, we will balance the number of elements in both heaps, so that they have an equal number of elements. If the count of numbers is odd, let's decide to have more numbers in max-heap than the **Min Heap**.

2. `insertNum(1)`: As '1' is smaller than '3', let's insert it into the **Max Heap**.



Now, we have two elements in the **Max Heap** and no elements in **Min Heap**. Let's take the largest element from the **Max Heap** and insert it into the **Min Heap**, to balance the number of elements in both heaps.

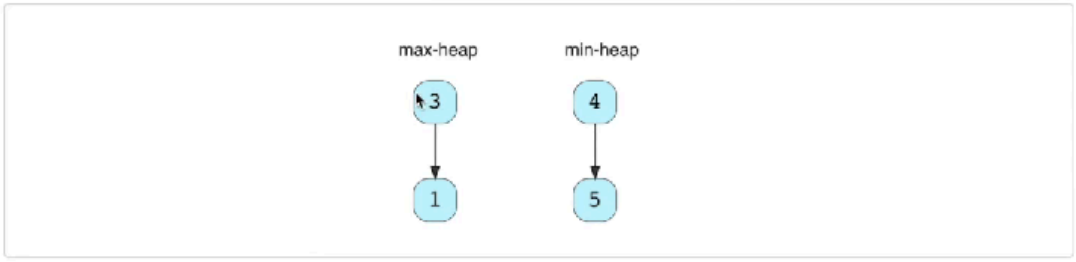
3. `findMedian()` : As we have an even number of elements, the median will be the average of the top element of both the heaps -> $(1 + 3)/2 = 2.0$
4. `insertNum(5)` : As '5' is greater than the top element of the **Max Heap**, we can insert it into the **Min Heap**. After the insertion, the total count of elements will be odd. As we had decided to have more numbers in the **Max Heap** than the **Min Heap**, we can take the top (smallest) number from the **Min Heap** and insert it into the **Max Heap**.



5. `findMedian()` : Since we have an odd number of elements, the median will be the top element of **Max Heap** -> 3. An odd number of elements also means that the **Max Heap** will have one extra element than the **Min Heap**.

---

5. `findMedian()` : Since we have an odd number of elements, the median will be the top element of **Max Heap** -> 3. An odd number of elements also means that the **Max Heap** will have one extra element than the **Min Heap**.
6. `insertNum(4)` : Insert '4' into **Min Heap**.



7. `findMedian()` : As we have an even number of elements, the median will be the average of the top element of both the heaps -> $(3 + 4)/2 = 3.5$

The time complexity of the addNum method in the MedianFinder class is O(log N), where N is the total number of elements inserted so far. This is because the method involves pushing elements into the max heap and the min heap, and the heapq.heappush operation has a time complexity of O(log N) for maintaining the heap property.

The findMedian method has a constant time complexity of O(1) since it only involves accessing the top elements of the max heap and the min heap.

Overall, the time complexity of adding a number and finding the median using the MedianFinder class is O(log N), where N is the total number of elements inserted so far.

```python
import heapq
class MedianFinder:
    def __init__(self):
        self.max_heap=[]    # to store the smaller half of elements
        self.min_heap=[]    # to store the larger half of elements

    def addNum(self, num: int) -> None:
        if len(self.max_heap)==0 or -self.max_heap[0]>=num:
            heapq.heappush(self.max_heap,-num)
        else:
            heapq.heappush(self.min_heap,num)

        #either both the heap will have equal no of element
        #or max-heap will have one more element that min-heap

        #so now check if max-heap contains more element than min-heap+1 then need to balance it
        #to balance it we pop element from max-heap and push it to minheap so that
        #no of elment in max-heap is equal or only 1 greater than minheap
        if len(self.max_heap)>len(self.min_heap)+1:
            heapq.heappush(self.min_heap, -heapq.heappop(self.max_heap))
        elif(len(self.max_heap)<len(self.min_heap)):
            heapq.heappush(self.max_heap,-heapq.heappop(self.min_heap))

    def findMedian(self) -> float:

        if len(self.min_heap)==len(self.max_heap):  #ie,even no of element
            a = -self.max_heap[0]
            b = self.min_heap[0]
            return (a+b)/2


        else:
            return -self.max_heap[0]
```

Merge K Sorted Arrays

```python
def mergeKSortedArrays(kArrays, k:int):
    heap = []  # min-heap to store elements
    result = []  # merged sorted array

    # Initialize the min-heap with the first element from each array
    for i, array in enumerate(kArrays):
        if len(array) > 0:
            heapq.heappush(heap, (array[0], i, 0))

    while heap:
        value, array_index, element_index = heapq.heappop(heap)
        result.append(value)

        # Check if there are more elements in the same array
        if element_index + 1 < len(kArrays[array_index]):
            next_element = kArrays[array_index][element_index + 1]
            heapq.heappush(heap, (next_element, array_index, element_index + 1))

    return result
```