STACK

Q) Design a stack which supports push, pop, peek, isEmpty & minEle. And the time complexity of each of these operations should be O(1)

Ex:

push(5)
push(1)
push(3)
peek() →3
minEle → 1
pop()
minEle → 1
pop()
minEle → 5

We have a stack, A = [5 2 1 7] & B = []

Taking a variable, minele = infinity, we will pop (7) from A and push (7) to B. We will check if the minimum value between infinity and 7. So the value of minele = 7 as 7 < infinity.

Now, we will pop (1) from A and push (1) to B. We will check for minimum element between 1 & 7. The value of minele = 1.

This process will be done for remaining two elements. For this, the time complexity would be O(2N), as we empty A to B and then return the value from B to A. So for A to B it is 'n' and from B to A it will be 'n'.

How to get this done in a time complexity of O (1)?

We will have 2 stacks. For A we will push (5) and will do the same for B as well.

When 1 is pushed in A and B, minEle will become 1 as 5 > 1.

We will push(3) in A. now the minEle = 1 in B, as 3 > 1, so we will push (1) again in stack B.

For the peek value, we will check stack A and for minEle we will check for stack B.

To find minEle, all we have to do is check the peek() of B.

LeetCode:155. Min Stack

Input

["MinStack", "push", "push", "push", "getMin", "pop", "top", "getMin"]

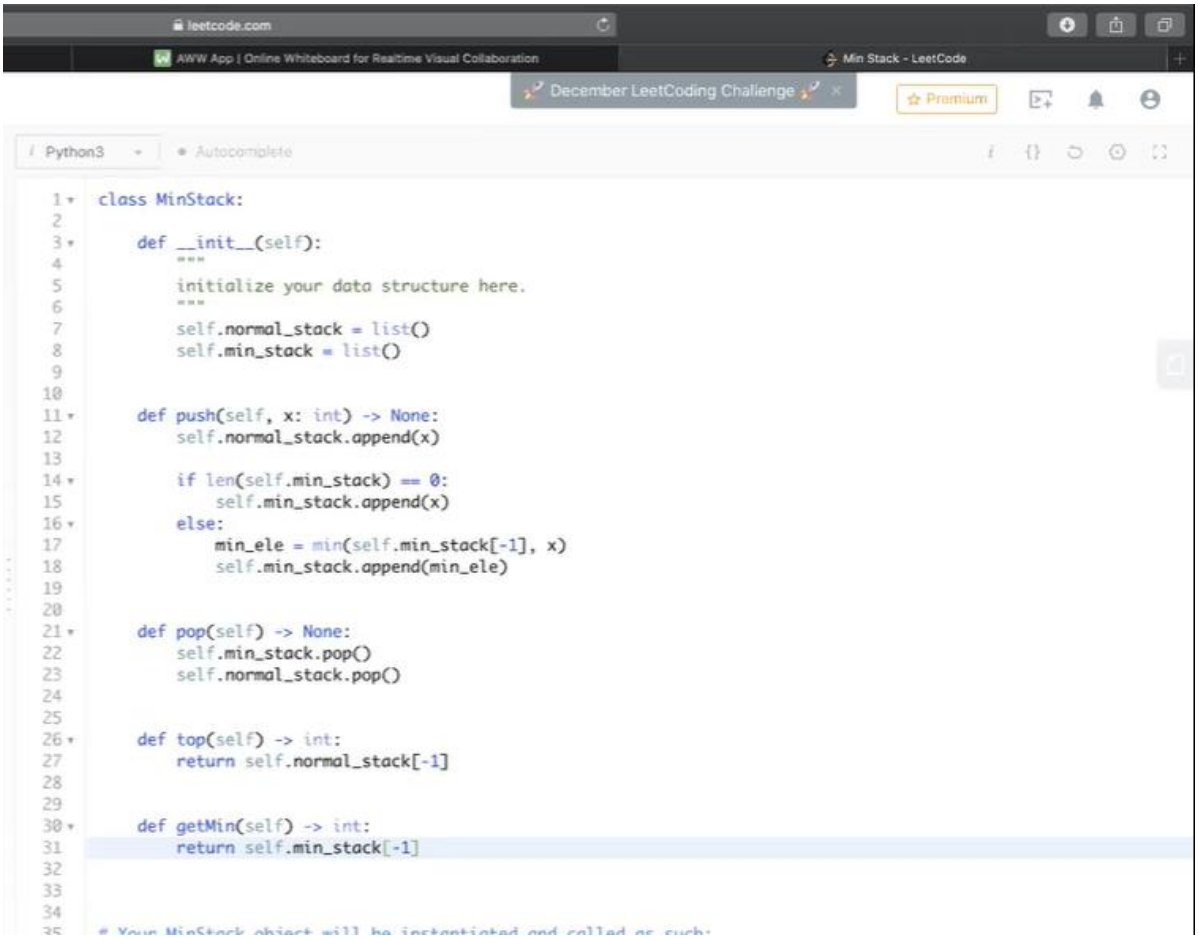
[[], [-2], [0], [-3], [], [], [], []]

Output

[null, null, null, null, -3, null, 0, -2]

Explanation

MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin(); // return -3
minStack.pop();
minStack.top();   // return 0
minStack.getMin(); // return -2



```
class MinStack:

    def __init__(self):
        """
        initialize your data structure here.
        """
        self.normal_stack = list()
        self.min_stack = list()


    def push(self, x):
        self.normal_stack.append(x)

        if len(self.min_stack) == 0:
            self.min_stack.append(x)
        else:
            min_ele = min(self.min_stack[-1], x)
            self.min_stack.append(min_ele)


    def pop(self):
        self.normal_stack.pop()
        self.min_stack.pop()


    def top(self):
        return self.normal_stack[-1]


    def getMin(self):
        return self.min_stack[-1]
```

We take two variables ‘normal_stack’ and ‘min_stack’. In normal stack we will append all elements, but in min_stack we will only append the minimum value.

Find the next greater element in right for each element.

[2, 1, 7, 4, 6, 8, 1, 9]

For

2 → 7

1 → 7

7 → 8

4 → 6

8 → 9

1 → 9

9 → None

[7, 7, 8, 6, 8, 9, 9, None]

Brute Force:

```
for i in rang(n-1):
        for j in range(i+1,n):
                if A[j] > A[i]:
                        print(A[j])
                        break
```

2nd Approach:

We are 2 and stack is also empty, so we will add 2 to the stack. At 1, we have not yet found the greater element for 2 and 1 so we will push 1 as well into the stack.

At 7, we see that 7 is the next greater element of 1. So we will pop 1 and push 7. As 7 is also the next greater element of 2, we can pop 2 and push 7.

At 4, as 7 > 4, we push 4 to stack as we need to find the next greater element for 4.

At 6. 6 > 4, so we can pop 4 and push 6. Since 6 < 7 and we have to find the next greater element for 7 as well, we will push 6.

At 8, 8 > 6, so we pop 6 and push 8. 8 > 7 so we will pop 7 and push 8.

The next element 1 is less that 8. So we need to find the next greater element for 8 and for 1.

At 9, 9 > 1, so we pop 1 and push 9. As 9 > 8, we pop 8 and push 9.

As there is no other element present, we will push None for 9.

```
def solve(A):
    n = len(A)
    stack = list()

    next_greater_element = [0] * len(A)
    # this give index and values as well.

    for idx, val in enumerate(A):
        if len(stack) == 0:
            stack.append(idx)
        else:
            cur = val

            while len(stack) != 0 and stack[-1] < cur:
                x = stack[-1]
                stack.pop()
                next_greater_element[x] = cur
            stack.append(idx)

    return next_greater_element


if __name__ == "__main__":
    A = [2, 1, 7, 4, 6, 8, 1, 9]
    ans = solve(A)
    print(ans)
```