ASSIGNMENT 15

1. Given an array **arr[ ]** of size **N** having elements, the task is to find the next greater element for each element of the array in order of their appearance in the array.Next greater element of an element in the array is the nearest element on the right which is greater than the current element.If there does not exist next greater of current element, then next greater element for current element is -1. For example, next greater of the last element is always -1.

```
def next_greater_element(arr):
    stack = []  # Stack to store the indices of potential next greater elements
    result = [-1] * len(arr)

    for i in range(len(arr) - 1, -1, -1):
        while stack and arr[stack[-1]] <= arr[i]:
            stack.pop()

        if stack:
            result[i] = arr[stack[-1]]

        stack.append(i)

    return result
```

2. Given an array **a** of integers of length **n**, find the nearest smaller number for every element such that the smaller element is on left side.If no small element present on the left print -1.

```
def nearest_smaller_number(arr):
    stack = []  # Stack to store the indices of potential nearest smaller elements
    result = [-1] * len(arr)

    for i in range(len(arr)):
        while stack and arr[stack[-1]] >= arr[i]:
            stack.pop()

        if stack:
            result[i] = arr[stack[-1]]

        stack.append(i)

    return result
```

3. Implement a Stack using two queues **q1** and **q2**.

```
from collections import deque

class Stack:
    def __init__(self):
        self.q1 = deque()
        self.q2 = deque()

    def push(self, x):
        # Push the element to the back of q1
        self.q1.append(x)

    def pop(self):
        # Move all elements from q1 to q2 except the last element
        while len(self.q1) > 1:
            self.q2.append(self.q1.popleft())

        # Remove and return the last element from q1
        element = self.q1.popleft()

        # Swap q1 and q2 to maintain the order of the remaining elements
        self.q1, self.q2 = self.q2, self.q1

        return element

    def top(self):
        # Move all elements from q1 to q2 except the last element
        while len(self.q1) > 1:
            self.q2.append(self.q1.popleft())

        # Get the last element from q1
        element = self.q1[0]

        # Move the last element to q2
        self.q2.append(self.q1.popleft())

        # Swap q1 and q2 to maintain the order of the remaining elements
        self.q1, self.q2 = self.q2, self.q1

        return element

    def empty(self):
        return len(self.q1) == 0
```

4. You are given a stack **St**. You have to reverse the stack using recursion.

```
def reverse_stack(stack):
    if not stack:
        return

    top_element = stack.pop()

    reverse_stack(stack)

    insert_at_bottom(stack, top_element)
def insert_at_bottom(stack, item):
    if not stack:
        stack.append(item)
        return
    top_element = stack.pop()

    insert_at_bottom(stack, item)

    stack.append(top_element)
```

5. You are given a string **S**, the task is to reverse the string using stack.

```
def reverse_string(s):
    stack = []
    reversed_string = ""

    # Push each character onto the stack
    for char in s:
        stack.append(char)

    # Pop each character from the stack and append it to the reversed string
    while stack:
        reversed_string += stack.pop()

    return reversed_string
```

6. Given string **S** representing a postfix expression, the task is to evaluate the expression and find the final value. Operators will only include the basic arithmetic operators like ***, /, + and -**.

```
def evaluate_postfix(expression):
    stack = []

    # Iterate through each character in the postfix expression
    for char in expression:
        if char.isdigit():
            # Push operand onto the stack
            stack.append(int(char))
        else:
            # Pop two operands from the stack and perform the operation
            operand2 = stack.pop()
            operand1 = stack.pop()

            # Perform the corresponding operation
            if char == '+':
                result = operand1 + operand2
            elif char == '-':
                result = operand1 - operand2
            elif char == '*':
                result = operand1 * operand2
            elif char == '/':
                result = operand1 / operand2

            # Push the result back onto the stack
            stack.append(result)

    # The final result will be the only element left on the stack
    return stack.pop()
```

7. Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

Implement the `MinStack` class:

- `MinStack()` initializes the stack object.
- `void push(int val)` pushes the element `val` onto the stack.
- `void pop()` removes the element on the top of the stack.
- `int top()` gets the top element of the stack.
- `int getMin()` retrieves the minimum element in the stack.

You must implement a solution with `O(1)` time complexity for each function

```
class MinStack:
    def __init__(self):
        self.stack = []  # Main stack to store elements
        self.min_stack = []  # Stack to store minimum elements

    def push(self, val):
        self.stack.append(val)  # Push the element onto the main stack

        # If the minimum stack is empty or the new element is smaller than the current minimum,
        # push the new element onto the minimum stack
        if not self.min_stack or val <= self.min_stack[-1]:
            self.min_stack.append(val)

    def pop(self):
        # If the top element of the main stack is equal to the top element of the minimum stack,
        # pop the top element from the minimum stack as well
        if self.stack[-1] == self.min_stack[-1]:
            self.min_stack.pop()

        self.stack.pop()  # Pop the top element from the main stack

    def top(self):
        return self.stack[-1]  # Return the top element of the main stack

    def getMin(self):
        return self.min_stack[-1]  # Return the top element of the minimum stack
```

8 Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it can trap after raining.

```
def trap_water(elevation_map):
    stack = []
    water = 0

    for current_index, current_bar_height in enumerate(elevation_map):
        while stack and current_bar_height > elevation_map[stack[-1]]:
            popped_bar_height = elevation_map[stack.pop()]

            if not stack:
                break
            distance = current_index - stack[-1] - 1
            trapped_water = distance * (min(current_bar_height, elevation_map[stack[-1]]) - popped_bar_height)
            water += trapped_water
        stack.append(current_index)

    return water
```