**DSA – ASSIGNMENT 15**

💡 **Question 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.

**Example 1:**

Input:

N = 4, arr[] = [1 3 2 4]

Output:

3 4 4 -1

Explanation:

In the array, the next larger element

to 1 is 3 , 3 is 4 , 2 is 4 and for 4 ?

since it doesn't exist, it is -1.

**Example 2:**

Input:

N = 5, arr[] [6 8 0 1 3]

Output:

8 -1 1 3 -1

Explanation:

In the array, the next larger element to

6 is 8, for 8 there is no larger elements

hence it is -1, for 0 it is 1 , for 1 it

is 3 and then for 3 there is no larger

element on right and hence -1.

**Solution. :-**

* Create an empty stack and initialize it.
* Iterate through the array from right to left.
  + Pop elements from the stack while they are smaller than or equal to the current element.
  + The popped elements do not have a next greater element.
* If the stack is empty after the previous step, assign -1 as the next greater element for the current element.
* If the stack is not empty, the top element of the stack is the next greater element for the current element.
* Push the current element onto the stack.
* Repeat steps 2-5 until all elements in the array have been processed.
* If there are any remaining elements in the stack, they do not have a next greater element. Assign -1 as the next greater element for each of these elements.

**def findNextGreaterElements(arr):**

**stack = []**

**result = []**

**for i in range(len(arr)-1, -1, -1):**

**while stack and stack[-1] <= arr[i]:**

**stack.pop()**

**if not stack:**

**result.append(-1)**

**else:**

**result.append(stack[-1])**

**stack.append(arr[i])\**

**result.reverse()**

**return result**

**arr = [1, 3, 2, 4]**

**result = findNextGreaterElements(arr)**

**print(result)**

💡 **Question 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.

**Example 1:**

Input: n = 3

a = {1, 6, 2}

Output: -1 1 1

Explaination: There is no number at the

left of 1. Smaller number than 6 and 2 is 1.

**Example 2:**

Input: n = 6

a = {1, 5, 0, 3, 4, 5}

Output: -1 1 -1 0 3 4

Explaination: Upto 3 it is easy to see

the smaller numbers. But for 4 the smaller

numbers are 1, 0 and 3. But among them 3

is closest. Similary for 5 it is 4.

**Solution. :-**

* Create an empty stack and initialize it.
* Iterate through the array from left to right.
  + Pop elements from the stack while they are greater than or equal to the current element.
  + The popped elements are not the nearest smaller numbers.
* If the stack is empty after the previous step, assign -1 as the nearest smaller number for the current element.
* If the stack is not empty, the top element of the stack is the nearest smaller number for the current element.
* Push the current element onto the stack.
* Repeat steps 2-5 until all elements in the array have been processed.

**def findNearestSmallerElements(arr):**

**stack = []**

**result = []**

**for i in range(len(arr)):**

**while stack and stack[-1] >= arr[i]:**

**stack.pop()**

**if not stack:**

**result.append(-1)**

**else:**

**result.append(stack[-1])**

**stack.append(arr[i])**

**return result**

**arr = [1, 6, 2]**

**result = findNearestSmallerElements(arr)**

**print(result)**

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

**Example 1:**

Input:

push(2)

push(3)

pop()

push(4)

pop()

Output:3 4

Explanation:

push(2) the stack will be {2}

push(3) the stack will be {2 3}

pop() poped element will be 3 the

  stack will be {2}

push(4) the stack will be {2 4}

pop()   poped element will be 4

**Example 2:**

Input:

push(2)

pop()

pop()

push(3)

Output:2 -1

**Solution. :-**

* Create two queues, q1 and q2.
* The push operation will be performed on q1. Add the element to q1.
* The pop operation will be performed by transferring elements from q1 to q2 until there is only one element left in q1.
* Remove and return the last element from q1.
* Swap the names of q1 and q2 so that q2 becomes the new empty queue for future push operations.

**class Stack:**

**def \_\_init\_\_(self):**

**self.q1 = []**

**self.q2 = []**

**def push(self, element):**

**self.q1.append(element)**

**def pop(self):**

**if not self.q1:**

**return None**

**while len(self.q1) > 1:**

**self.q2.append(self.q1.pop(0))**

**popped\_element = self.q1.pop(0)**

**self.q1, self.q2 = self.q2, self.q1**

**return popped\_element**

**stack = Stack()**

**stack.push(2)**

**stack.push(3)**

**print(stack.pop())**

**stack.push(4)**

**print(stack.pop())**

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

**Example 1:**

Input:St = {3,2,1,7,6}

Output:{6,7,1,2,3}

**Example 2:**

Input:St = {4,3,9,6}

Output:{6,9,3,4}

**Solution. :-**

* Create a recursive function called reverseStack that takes the stack St as a parameter.
* If the stack St is empty or contains only one element, return.
* Otherwise, pop the top element from the stack and store it in a variable called top\_element.
* Call the reverseStack function recursively on the remaining stack St.
* After the recursive call, insert the top\_element at the bottom of the reversed stack by calling another recursive function called insertAtBottom.
* In the insertAtBottom function, if the stack is empty, push the element into the stack.
* Otherwise, pop the top element from the stack and recursively call insertAtBottom to insert the element at the bottom of the stack.
* Once the insertAtBottom function returns, the stack will be reversed.

**def reverseStack(St):**

**if len(St) <= 1:**

**return**

**top\_element = St.pop()**

**reverseStack(St)**

**insertAtBottom(St, top\_element)**

**def insertAtBottom(St, element):**

**if len(St) == 0:**

**St.append(element)**

**return**

**top = St.pop()**

**insertAtBottom(St, element)**

**St.append(top)**

**St = [3, 2, 1, 7, 6]**

**reverseStack(St)**

**print(St)**

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

**Example 1:**

Input: S="GeeksforGeeks"

Output: skeeGrofskeeG

**Solution. :-**

* Create an empty stack.
* Iterate through each character in the input string S.
* Push each character onto the stack.
* After iterating through all the characters, create an empty string called reversed\_string.
* Pop each character from the stack and append it to the reversed\_string.
* The reversed\_string will be the reversed version of the input string.

**def reverseString(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**

**S = "GeeksforGeeks"**

**reversed\_str = reverseString(S)**

**print(reversed\_str)**

💡 **Question 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 -**.

**Example 1:**

Input: S = "231\*+9-"

Output: -4

Explanation:

After solving the given expression,

we have -4 as result.

**Example 2:**

Input: S = "123+\*8-"

Output: -3

Explanation:

After solving the given postfix

expression, we have -3 as result.

**Solution. :-**

* Create an empty stack to store operands.
* Iterate through each character in the input string S.
  + If the character is an operand (a digit), convert it to an integer and push it onto the stack.
  + If the character is an operator, pop the last two operands from the stack.
  + Perform the corresponding arithmetic operation on the operands based on the operator.
  + Push the result back onto the stack.
* After iterating through all the characters, the final value will be the top element of the stack.
* Pop the top element from the stack and return it as the result.

**def evaluatePostfixExpression(S):**

**stack = []**

**for char in S:**

**if char.isdigit():**

**stack.append(int(char))**

**else:**

**operand2 = stack.pop()**

**operand1 = stack.pop()**

**if char == '+':**

**result = operand1 + operand2**

**elif char == '-':**

**result = operand1 - operand2**

**elif char == '\*':**

**result = operand1 \* operand2**

**elif char == '/':**

**result = operand1 / operand2**

**stack.append(result)**

**return stack.pop()**

**S = "231\*+9-"**

**result = evaluatePostfixExpression(S)**

**print(result)**

💡 **Question 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.

**Example 1:**

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

**Solution. :-**

* Initialize two stacks: stack and min\_stack. The stack will store the actual elements, and min\_stack will store the minimum elements.
* When pushing an element, check if the min\_stack is empty or if the new element is smaller than or equal to the top element of the min\_stack.
  + If true, push the new element onto both the stack and the min\_stack.
  + If false, push the new element onto the stack, but duplicate the top element of the min\_stack and push it onto the min\_stack. This ensures that the min\_stack always has the minimum element at the top.
* When popping an element, pop the top element from both the stack and the min\_stack.
* To get the top element, return the top element from the stack without modifying the stack.
* To retrieve the minimum element, return the top element from the min\_stack without modifying the stack.

class MinStack:

**def \_\_init\_\_(self):**

**self.stack = []**

**self.min\_stack = []**

**def push(self, val):**

**self.stack.append(val)**

**if not self.min\_stack or val <= self.min\_stack[-1][0]:**

**self.min\_stack.append((val, 1))**

**else:**

**self.min\_stack[-1] = (self.min\_stack[-1][0], self.min\_stack[-1][1] + 1)**

**def pop(self):**

**if self.stack:**

**if self.min\_stack[-1][1] > 1:**

**self.min\_stack[-1] = (self.min\_stack[-1][0], self.min\_stack[-1][1] - 1)**

**else:**

**self.min\_stack.pop()**

**self.stack.pop()**

**def top(self):**

**if self.stack:**

**return self.stack[-1]**

**def getMin(self):**

**if self.min\_stack:**

**return self.min\_stack[-1][0]**

**minStack = MinStack()**

**minStack.push(-2)**

**minStack.push(0)**

**minStack.push(-3)**

**print(minStack.getMin())**

**minStack.pop()**

**print(minStack.top())**

**print(minStack.getMin())**

💡 **Question 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.

**Example 1:**



Input: height = [0,1,0,2,1,0,1,3,2,1,2,1]

Output: 6

Explanation: The above elevation map (black section) is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped.

**Example 2:**

Input: height = [4,2,0,3,2,5]

Output: 9

**Solution. :-**

* Initialize two pointers, left and right, at the beginning and end of the elevation map.
* Initialize variables left\_max and right\_max to keep track of the maximum height encountered on the left and right side, respectively. Set both variables to 0 initially.
* Initialize a variable water to store the total amount of water trapped, set it to 0.
* While the left pointer is less than the right pointer:
  + If the height at the left pointer is less than or equal to the height at the right pointer:
    - Check if the height at the left pointer is greater than left\_max. If true, update left\_max to the current height.
    - Add the difference between left\_max and the height at the left pointer to water.
    - Increment the left pointer.
  + If the height at the left pointer is greater than the height at the right pointer:
    - Check if the height at the right pointer is greater than right\_max. If true, update right\_max to the current height.
    - Add the difference between right\_max and the height at the right pointer to water.
    - Decrement the right pointer.
* After the loop, return the water variable, which represents the total amount of water trapped.

**def trapWater(height):**

**left = 0**

**right = len(height) - 1**

**left\_max = right\_max = water = 0**

**while left < right:**

**if height[left] <= height[right]:**

**if height[left] > left\_max:**

**left\_max = height[left]**

**else:**

**water += left\_max - height[left]**

**left += 1**

**else:**

**if height[right] > right\_max:**

**right\_max = height[right]**

**else:**

**water += right\_max - height[right]**

**right -= 1**

**return water**

**height = [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]**

**water\_trapped = trapWater(height)**

**print(water\_trapped)**