Q.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.

Solution :

def find\_next\_greater\_elements(arr):

stack = []

result = [-1] \* len(arr)

for i in range(len(arr)):

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

index = stack.pop()

result[index] = arr[i]

stack.append(i)

return result

Q.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.

Solution :

def find\_nearest\_smaller\_elements(arr):

stack = []

result = [-1] \* len(arr)

for i in range(len(arr)):

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

stack.pop()

if stack:

result[i] = stack[-1]

stack.append(arr[i])

return result

Q.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

Solution :

class Stack:

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

self.q1 = []

self.q2 = []

def push(self, value):

self.q1.append(value)

def pop(self):

if not self.q1 and not self.q2:

return "Stack is empty"

while len(self.q1) > 1:

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

return self.q1.pop(0) if self.q1 else self.q2.pop(0)

def top(self):

if not self.q1 and not self.q2:

return "Stack is empty"

while len(self.q1) > 1:

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

top\_element = self.q1[0] if self.q1 else self.q2[0]

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

return top\_element

def isEmpty(self):

return len(self.q1) == 0 and len(self.q2) == 0

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

Solution :

def reverse\_stack(St):

if len(St) <= 1:

return

item = St.pop()

reverse\_stack(St)

insert\_at\_bottom(St, item)

def insert\_at\_bottom(St, item):

if len(St) == 0:

St.append(item)

else:

top\_item = St.pop()

insert\_at\_bottom(St, item)

St.append(top\_item)

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

Solution :

def reverse\_string(S):

stack = []

for char in S:

stack.append(char)

reversed\_string = ""

while stack:

reversed\_string += stack.pop()

return reversed\_string

Q6] 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 -**.

Solution :

def evaluate\_postfix\_expression(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()

Q.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.

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 :

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]:

self.min\_stack.append(val)

def pop(self):

if self.stack:

popped = self.stack.pop()

if popped == self.min\_stack[-1]:

self.min\_stack.pop()

def top(self):

if self.stack:

return self.stack[-1]

return None

def getMin(self):

if self.min\_stack:

return self.min\_stack[-1]

return None

Q.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.

Solution :

def trap\_water(height):

left = 0

right = len(height) - 1

left\_max = right\_max = total\_water = 0

while left <= right:

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

left\_max = max(left\_max, height[left])

total\_water += left\_max - height[left]

left += 1

else:

right\_max = max(right\_max, height[right])

total\_water += right\_max - height[right]

right -= 1

return total\_water