NAME – AMEY HUSHAR PATKAR

ASSIGNMENTS NO – 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.

ANS –

To find the next greater element for each element in the given array, you can use a stack-based approach. Here's the step-by-step algorithm:

Initialize an empty stack and create an array to store the result.

Traverse the given array from right to left.

For each element arr[i], repeat steps 4 and 5 until the stack is empty or the top element of the stack is greater than arr[i].

Pop elements from the stack as long as they are smaller than or equal to arr[i].

If the stack is empty, there is no greater element to the right of arr[i], so store -1 in the result array at index i. Otherwise, store the top element of the stack in the result array at index i.

Push arr[i] onto the stack.

Repeat steps 3 to 6 for all elements of the given array.

Return the result array.

Here's the implementation of the above algorithm in Python:

def findNextGreaterElement(arr):

stack = []

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

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

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

stack.pop()

if stack:

result[i] = stack[-1]

stack.append(arr[i])

return result

# Example 1

arr1 = [1, 3, 2, 4]

print(findNextGreaterElement(arr1)) # Output: [3, 4, 4, -1]

# Example 2

arr2 = [6, 8, 0, 1, 3]

print(findNextGreaterElement(arr2)) # Output: [8, -1, 1, 3, -1]

The time complexity of this algorithm is O(N), where N is the size of the given array.

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.

ANS –

To find the nearest smaller number on the left for each element in the given array, you can use a stack-based approach similar to the previous question. Here's the step-by-step algorithm:

Initialize an empty stack and create an array to store the result.

Traverse the given array from left to right.

For each element arr[i], repeat steps 4 and 5 until the stack is empty or the top element of the stack is smaller than arr[i].

Pop elements from the stack as long as they are greater than or equal to arr[i].

If the stack is empty, there is no smaller element to the left of arr[i], so store -1 in the result array at index i. Otherwise, store the top element of the stack in the result array at index i.

Push arr[i] onto the stack.

Repeat steps 3 to 6 for all elements of the given array.

Return the result array.

Here's the implementation of the above algorithm in Python:

def findNearestSmallerNumber(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

# Example 1

arr1 = [1, 6, 2]

print(findNearestSmallerNumber(arr1)) # Output: [-1, 1, 1]

# Example 2

arr2 = [1, 5, 0, 3, 4, 5]

print(findNearestSmallerNumber(arr2)) # Output: [-1, 1, -1, 0, 3, 4]

The time complexity of this algorithm is O(N), where N is the length of the given array.

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

ANS –

To implement a stack using two queues, you can use the following approach:

Create two queues, q1 and q2.

Implement the push operation:

Add the new element to the empty queue (either q1 or q2).

If any elements exist in the other queue, dequeue all elements from that queue and enqueue them into the newly populated queue.

Swap the names of the two queues (q1 and q2).

Implement the pop operation:

If the current queue (q1) is not empty, dequeue the front element and return it as the popped element.

If the current queue (q1) is empty, return -1 as there are no elements in the stack.

Here's the implementation of the stack using two queues in Python:

from queue import Queue

class Stack:

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

self.q1 = Queue()

self.q2 = Queue()

def push(self, x):

# Add the new element to an empty queue (either q1 or q2)

if self.q1.empty():

self.q1.put(x)

else:

self.q2.put(x)

# Dequeue all elements from the other queue and enqueue them into the newly populated queue

while not self.q1.empty():

self.q2.put(self.q1.get())

# Swap the names of the two queues

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

def pop(self):

# If the current queue (q1) is not empty, dequeue the front element and return it as the popped element

if not self.q1.empty():

return self.q1.get()

# If the current queue (q1) is empty, return -1 as there are no elements in the stack

return -1

# Example 1

stack1 = Stack()

stack1.push(2)

stack1.push(3)

print(stack1.pop()) # Output: 3

stack1.push(4)

print(stack1.pop()) # Output: 4

# Example 2

stack2 = Stack()

stack2.push(2)

print(stack2.pop()) # Output: 2

print(stack2.pop()) # Output: -1

stack2.push(3)

print(stack2.pop()) # Output: 3

This implementation provides the desired functionality of a stack using two queues. The time complexity of both push and pop operations is O(N) in the worst case, where N is the number of elements in the stack.

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}

ANS –

To reverse a stack using recursion, you can follow these steps:

Implement a recursive function, let's call it reverseStack, that takes the stack as a parameter.

In the reverseStack function:

If the stack is empty or contains only one element, return the stack as it is.

Otherwise, pop an element from the stack and store it in a variable, let's call it top.

Recursively call reverseStack on the remaining stack.

After the recursive call, insert top at the bottom of the reversed stack.

Return the reversed stack.

Call the reverseStack function on the given stack and store the result in a new stack variable.

The new stack variable will now contain the reversed stack.

Here's the implementation of reversing a stack using recursion in Python:

class Stack:

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

self.items = []

def push(self, item):

self.items.append(item)

def pop(self):

if not self.is\_empty():

return self.items.pop()

def is\_empty(self):

return len(self.items) == 0

def reverse(self):

if not self.is\_empty():

top = self.pop()

self.reverse()

self.insert\_at\_bottom(top)

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

if self.is\_empty():

self.push(item)

else:

top = self.pop()

self.insert\_at\_bottom(item)

self.push(top)

# Example 1

stack1 = Stack()

stack1.push(3)

stack1.push(2)

stack1.push(1)

stack1.push(7)

stack1.push(6)

stack1.reverse()

print(stack1.items) # Output: [6, 7, 1, 2, 3]

# Example 2

stack2 = Stack()

stack2.push(4)

stack2.push(3)

stack2.push(9)

stack2.push(6)

stack2.reverse()

print(stack2.items) # Output: [6, 9, 3, 4]

In the above implementation, we have a Stack class that provides the necessary methods for stack operations. The reverse method is used to reverse the stack using recursion. The time complexity of reversing the stack is O(N^2), where N is the number of elements in the stack, because for each element, we perform a recursive call on the remaining stack elements.

Question 5

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

Example 1:

Input: S="GeeksforGeeks"

Output: skeeGrofskeeG

ANS –

To reverse a string using a stack, you can follow these steps:

Initialize an empty stack.

Iterate through each character in the string from left to right.

Push each character onto the stack.

After iterating through all the characters, pop each character from the stack and append it to a new string.

The new string will now contain the reversed version of the original string.

Here's the implementation of reversing a string using a stack in Python:

class Stack:

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

self.items = []

def push(self, item):

self.items.append(item)

def pop(self):

if not self.is\_empty():

return self.items.pop()

def is\_empty(self):

return len(self.items) == 0

def reverse\_string(string):

stack = Stack()

# Push each character onto the stack

for char in string:

stack.push(char)

reversed\_string = ""

# Pop each character from the stack and append it to the reversed string

while not stack.is\_empty():

reversed\_string += stack.pop()

return reversed\_string

# Example

S = "GeeksforGeeks"

reversed\_S = reverse\_string(S)

print(reversed\_S) # Output: skeeGrofskeeG

In the above implementation, we have a Stack class that provides the necessary methods for stack operations. The reverse\_string function takes a string as input and uses a stack to reverse the string. The time complexity of reversing the string is O(N), where N is the length of the string.

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.

ANS –

To evaluate a postfix expression and find the final value, you can use a stack-based approach. Here's the step-by-step algorithm:

Create an empty stack.

Iterate through each character in the postfix expression from left to right.

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 top two elements from the stack.

Perform the corresponding operation on the popped elements.

Push the result back onto the stack.

After iterating through all the characters, the stack will contain the final value of the postfix expression.

Pop the top element from the stack and return it as the final result.

Here's the implementation of evaluating a postfix expression in Python:

class Stack:

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

self.items = []

def push(self, item):

self.items.append(item)

def pop(self):

if not self.is\_empty():

return self.items.pop()

def is\_empty(self):

return len(self.items) == 0

def evaluate\_postfix(expression):

stack = Stack()

for char in expression:

if char.isdigit():

stack.push(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.push(result)

return stack.pop()

# Example 1

S1 = "231\*+9-"

result1 = evaluate\_postfix(S1)

print(result1) # Output: -4

# Example 2

S2 = "123+\*8-"

result2 = evaluate\_postfix(S2)

print(result2) # Output: -3

In the above implementation, we have a Stack class that provides the necessary methods for stack operations. The evaluate\_postfix function takes a postfix expression as input and uses a stack to evaluate the expression. The time complexity of evaluating the postfix expression is O(N), where N is the length of the expression.

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

ANS –

To design a stack that supports push, pop, top, and retrieving the minimum element in constant time, we can utilize an additional stack to keep track of the minimum values at each step.

Here's the implementation of the MinStack class in Python:

class MinStack:

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

self.stack = []

self.min\_stack = []

def push(self, val: int) -> None:

self.stack.append(val)

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

self.min\_stack.append(val)

def pop(self) -> None:

if self.stack:

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

self.min\_stack.pop()

self.stack.pop()

def top(self) -> int:

if self.stack:

return self.stack[-1]

return None

def getMin(self) -> int:

if self.min\_stack:

return self.min\_stack[-1]

return None

This implementation maintains two stacks: stack and min\_stack. The stack stores all the elements pushed onto it, and the min\_stack keeps track of the minimum values. Whenever an element is pushed onto the stack, we compare it with the top element of the min\_stack. If it is less than or equal to the top element, we push it onto the min\_stack. When an element is popped from the stack, we check if it is equal to the top element of the min\_stack. If it is, we also remove the top element from the min\_stack.

Let's run the given example to verify the implementation:

minStack = MinStack()

minStack.push(-2)

minStack.push(0)

minStack.push(-3)

print(minStack.getMin()) # Output: -3

minStack.pop()

print(minStack.top()) # Output: 0

print(minStack.getMin()) # Output: -2

The output matches the expected result [null,null,null,null,-3,null,0,-2]. The getMin function retrieves the minimum element in constant time, as required.

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

ANS –

To compute the amount of water that can be trapped after raining in an elevation map, we can use the Two-Pointer approach.

Here's the step-by-step algorithm:

Initialize two pointers, left and right, pointing to the start and end of the elevation map.

Initialize two variables, left\_max and right\_max, to keep track of the maximum height encountered from the left and right sides, respectively. Set both variables to 0.

Initialize a variable water to store the total amount of trapped water. Set it to 0.

While left is less than or equal to right:

If height[left] is less than or equal to height[right]:

If height[left] is greater than left\_max, update left\_max to height[left].

Otherwise, calculate the amount of water that can be trapped at the left pointer position: left\_max - height[left]. Add this value to water.

Increment left by 1.

Otherwise, if height[left] is greater than height[right]:

If height[right] is greater than right\_max, update right\_max to height[right].

Otherwise, calculate the amount of water that can be trapped at the right pointer position: right\_max - height[right]. Add this value to water.

Decrement right by 1.

Return the total amount of trapped water.

Here's the implementation of the trap function in Python:

def trap(height):

left = 0

right = len(height) - 1

left\_max = 0

right\_max = 0

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

Let's run the given examples to test the trap function:

height1 = [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]

print(trap(height1)) # Output: 6

height2 = [4, 2, 0, 3, 2, 5]

print(trap(height2)) # Output: 9

The output matches the expected results. The function correctly calculates the amount of water that can be trapped in the given elevation maps.