**DSA – ASSIGNMENT 16**

💡 **Question 1** Given an array, for each element find the value of the nearest element to the right which is having a frequency greater than that of the current element. If there does not exist an answer for a position, then make the value ‘-1’.

**Examples:**

Input: a[] = [1, 1, 2, 3, 4, 2, 1]

Output : [-1, -1, 1, 2, 2, 1, -1]

Explanation:

Given array a[] = [1, 1, 2, 3, 4, 2, 1]

Frequency of each element is: 3, 3, 2, 1, 1, 2, 3

Lets calls Next Greater Frequency element as NGF

1. For element a[0] = 1 which has a frequency = 3,

As it has frequency of 3 and no other next element

has frequency more than 3 so '-1'

2. For element a[1] = 1 it will be -1 same logic

like a[0]

3. For element a[2] = 2 which has frequency = 2,

NGF element is 1 at position = 6 with frequency

of 3 > 2

4. For element a[3] = 3 which has frequency = 1,

NGF element is 2 at position = 5 with frequency

of 2 > 1

5. For element a[4] = 4 which has frequency = 1,

NGF element is 2 at position = 5 with frequency

of 2 > 1

6. For element a[5] = 2 which has frequency = 2,

NGF element is 1 at position = 6 with frequency

of 3 > 2

7. For element a[6] = 1 there is no element to its

right, hence -1

Input : a[] = [1, 1, 1, 2, 2, 2, 2, 11, 3, 3]

Output : [2, 2, 2, -1, -1, -1, -1, 3, -1, -1]

**Solution. :-**

* Initialize an empty dictionary called frequency to store the frequency of each element in the array.
* Initialize an empty stack called stack to store the elements encountered so far.
* Initialize an empty dictionary called nextGreaterFrequency to store the nearest element to the right with a greater frequency.
* Iterate through the array in reverse order (from right to left):
  + While the stack is not empty and the frequency of the element at the top of the stack is less than or equal to the frequency of the current element:
    - Pop the element from the stack.
  + If the stack is empty, set the value of nextGreaterFrequency for the current element as -1.
  + If the stack is not empty, set the value of nextGreaterFrequency for the current element as the element at the top of the stack.
  + Push the current element onto the stack.
  + Increment the frequency of the current element in the frequency dictionary.
* Create an empty list called result to store the output.
* Iterate through the original array:
  + Append the value of nextGreaterFrequency for each element to the result list.
* Return the result list as the output.

**def findNearestGreaterFrequency(arr):**

**frequency = {}**

**stack = []**

**nextGreaterFrequency = {}**

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

**while stack and frequency.get(stack[-1], 0) <= frequency.get(arr[i], 0):**

**stack.pop()**

**if not stack:**

**nextGreaterFrequency[arr[i]] = -1**

**else:**

**nextGreaterFrequency[arr[i]] = stack[-1]**

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

**frequency[arr[i]] = frequency.get(arr[i], 0) + 1**

**result = [nextGreaterFrequency.get(num, -1) for num in arr]**

**return result**

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

**result = findNearestGreaterFrequency(arr)**

**print(result)**

💡 **Question 2** Given a stack of integers, sort it in ascending order using another temporary stack.

**Examples:**

Input : [34, 3, 31, 98, 92, 23]

Output : [3, 23, 31, 34, 92, 98]

Input : [3, 5, 1, 4, 2, 8]

Output : [1, 2, 3, 4, 5, 8]

**Solution. :-**

* Create a temporary stack to store the sorted elements.
* While the original stack is not empty:
  + Pop the top element from the original stack and store it in a variable called temp.
  + While the temporary stack is not empty and the top element of the temporary stack is greater than temp:
    - Pop the top element from the temporary stack and push it back into the original stack.
  + Push temp onto the temporary stack.
* After the loop, the temporary stack will contain the elements of the original stack in ascending order.
* Transfer the elements from the temporary stack back to the original stack to obtain the sorted stack.

**def sortStack(stack):**

**tempStack = []**

**while stack:**

**temp = stack.pop()**

**while tempStack and tempStack[-1] > temp:**

**stack.append(tempStack.pop())**

**tempStack.append(temp)**

**while tempStack:**

**stack.append(tempStack.pop())**

**return stack**

**stack = [34, 3, 31, 98, 92, 23]**

**sortedStack = sortStack(stack)**

**print(sortedStack)**

💡 **Question 3** Given a stack with **push()**, **pop()**, and **empty()** operations, The task is to delete the **middle** element \*\*\*\*of it without using any additional data structure.

**Examples:**

Input  : Stack[] = [1, 2, 3, 4, 5]

Output : Stack[] = [1, 2, 4, 5]

Input  : Stack[] = [1, 2, 3, 4, 5, 6]

Output : Stack[] = [1, 2, 4, 5, 6]

**Solution. :-**

* Create a recursive function called deleteMiddle that takes the stack as input.
* If the stack is empty or contains only one element, return the stack as it is (base case).
* Pop the top element from the stack and store it in a variable called mid.
* Recursively call the deleteMiddle function on the remaining elements of the stack.
* If the size of the stack is odd, discard the middle element (mid).
* If the size of the stack is even, push the mid element back onto the stack.

**def deleteMiddle(stack):**

**if len(stack) == 0 or len(stack) == 1:**

**return stack**

**mid = stack.pop()**

**stack = deleteMiddle(stack)**

**size = len(stack)**

**if size % 2 == 0:**

**stack.append(mid)**

**return stack**

**stack = [1, 2, 3, 4, 5]**

**updatedStack = deleteMiddle(stack)**

**print(updatedStack)**

💡 **Question 4** Given a Queue consisting of first **n** natural numbers (in random order). The task is to check whether the given Queue elements can be arranged in increasing order in another Queue using a stack. The operation allowed are:

1. Push and pop elements from the stack
2. Pop (Or Dequeue) from the given Queue.
3. Push (Or Enqueue) in the another Queue.

**Examples :**

Input : Queue[] = { 5, 1, 2, 3, 4 }

Output : Yes

Pop the first element of the given Queue

i.e 5. Push 5 into the stack.

Now, pop all the elements of the given Queue and push them to second Queue.

Now, pop element 5 in the stack and push it to the second Queue.

Input : Queue[] = { 5, 1, 2, 6, 3, 4 }

Output : No

Push 5 to stack.

Pop 1, 2 from given Queue and push it to another Queue.

Pop 6 from given Queue and push to stack.

Pop 3, 4 from given Queue and push to second Queue.

Now, from using any of above operation, we cannot push 5 into the second Queue because it is below the 6 in the stack.

**Solution. :-**

* Initialize an empty stack and an empty result queue.
* Initialize a variable expectedElement to 1, which represents the expected next element.
* Iterate through the given queue:
  + If the front element of the queue is equal to expectedElement, enqueue it into the result queue, and increment expectedElement by 1.
  + Otherwise, check if the stack is not empty and the top element of the stack is equal to expectedElement. If true, pop the top element from the stack, enqueue it into the result queue, and increment expectedElement by 1.
  + If none of the above conditions are met, push the front element of the queue into the stack and dequeue it from the queue.
* After the loop, check if the remaining elements in the stack match the expected elements. If not, return "No".
* Enqueue the remaining elements in the stack into the result queue.
* Check if the elements in the result queue are in increasing order. If true, return "Yes". Otherwise, return "No".

from queue import Queue

**def checkQueueOrder(queue):**

**stack = []**

**resultQueue = Queue()**

**expectedElement = 1**

**while not queue.empty():**

**frontElement = queue.queue[0]**

**if frontElement == expectedElement:**

**resultQueue.put(frontElement)**

**expectedElement += 1**

**queue.get()**

**elif stack and stack[-1] == expectedElement:**

**resultQueue.put(stack.pop())**

**expectedElement += 1**

**else:**

**stack.append(queue.get())**

**while stack:**

**resultQueue.put(stack.pop())**

**while not resultQueue.empty():**

**if resultQueue.get() != expectedElement - 1:**

**return "No"**

**expectedElement += 1**

**return "Yes"**

**queue = Queue()**

**queue.put(5)**

**queue.put(1)**

**queue.put(2)**

**queue.put(3)**

**queue.put(4)**

**result = checkQueueOrder(queue)**

**print(result)**

💡 **Question 5** Given a number , write a program to reverse this number using stack.

**Examples:**

Input : 365

Output : 563

Input : 6899

Output : 9986

**Solution. :-**

* Convert the given number into a string.
* Initialize an empty stack.
* Iterate through each digit in the string representation of the number:
  + Push each digit onto the stack.
* Initialize an empty string to store the reversed number.
* Pop each digit from the stack and append it to the reversed number string.
* Convert the reversed number string back to an integer.
* Return the reversed number.

**def reverseNumber(num):**

**num\_str = str(num)**

**stack = []**

**# Push each digit onto the stack**

**for digit in num\_str:**

**stack.append(digit)**

**reversed\_num\_str = ''**

**# Pop each digit from the stack and append it to the reversed number string**

**while stack:**

**reversed\_num\_str += stack.pop()**

**# Convert the reversed number string back to an integer**

**reversed\_num = int(reversed\_num\_str)**

**return reversed\_num**

**number = 365**

**reversed\_number = reverseNumber(number)**

**print(reversed\_number)**

💡 **Question 6** Given an integer k and a [**queue**](https://www.geeksforgeeks.org/queue-data-structure/) of integers, The task is to reverse the order of the first **k** elements of the queue, leaving the other elements in the same relative order.

Only following standard operations are allowed on queue.

* **enqueue(x) :** Add an item x to rear of queue
* **dequeue() :** Remove an item from front of queue
* **size() :** Returns number of elements in queue.
* **front() :** Finds front item.

**Solution. :-**

* Create an empty stack and push the first k elements from the front of the queue into the stack.
* Dequeue the first k elements from the queue.
* While the stack is not empty, pop each element from the stack and enqueue it back into the queue.
* Now the first k elements of the queue will be in reverse order. However, the remaining elements will still be in the original order.
* Dequeue the first k elements from the front of the queue and enqueue them back into the queue in the same order.
* The queue now contains the reversed order of the first k elements, while the other elements remain in the same relative order.

**from queue import Queue**

**def reverseFirstK(queue, k):**

**if queue.empty() or k <= 0 or k > queue.qsize():**

**return**

**stack = []**

**count = 0**

**# Push the first k elements from the queue into the stack**

**while count < k:**

**stack.append(queue.queue[0])**

**queue.get()**

**count += 1**

**# Pop each element from the stack and enqueue it back into the queue**

**while stack:**

**queue.put(stack.pop())**

**# Dequeue the first k elements from the front of the queue and enqueue them back in the same order**

**for i in range(queue.qsize() - k):**

**queue.put(queue.queue[0])**

**queue.get()**

**return queue**

**queue = Queue()**

**queue.put(1)**

**queue.put(2)**

**queue.put(3)**

**queue.put(4)**

**queue.put(5)**

**k = 3**

**reversed\_queue = reverseFirstK(queue, k)**

**while not reversed\_queue.empty():**

**print(reversed\_queue.queue[0], end=' ')**

**reversed\_queue.get()**

💡 **Question 7** Given a sequence of n strings, the task is to check if any two similar words come together and then destroy each other then print the number of words left in the sequence after this pairwise destruction.

**Examples:**

Input : ab aa aa bcd ab

Output : 3

As aa, aa destroys each other so,

ab bcd ab is the new sequence.

Input :  tom jerry jerry tom

Output : 0

As first both jerry will destroy each other.

Then sequence will be tom, tom they will also destroy

each other. So, the final sequence doesn’t contain any

word.

**Solution. :-**

* Initialize an empty stack.
* Iterate through each word in the sequence:
  + If the stack is empty or the top word of the stack is not equal to the current word, push the current word onto the stack.
  + Otherwise, if the top word of the stack is equal to the current word, pop the top word from the stack as they are similar and will destroy each other.
* After the iteration, the stack will contain the remaining words in the sequence after pairwise destruction.
* Return the size of the stack, which represents the number of words left in the sequence.

**def countWordsAfterDestruction(sequence):**

**stack = []**

**for word in sequence:**

**if not stack or stack[-1] != word:**

**stack.append(word)**

**else:**

**stack.pop()**

**return len(stack)**

**sequence = ["ab", "aa", "aa", "bcd", "ab"]**

**words\_left = countWordsAfterDestruction(sequence)**

**print(words\_left)**

💡 **Question 8** Given an array of integers, the task is to find the maximum absolute difference between the nearest left and the right smaller element of every element in the array.

**Note:** If there is no smaller element on right side or left side of any element then we take zero as the smaller element. For example for the leftmost element, the nearest smaller element on the left side is considered as 0. Similarly, for rightmost elements, the smaller element on the right side is considered as 0.

**Examples:**

Input : arr[] = {2, 1, 8}

Output : 1

Left smaller LS[] {0, 0, 1}

Right smaller RS[] {1, 0, 0}

Maximum Diff of abs(LS[i] - RS[i]) = 1

Input : arr[] = {2, 4, 8, 7, 7, 9, 3}

Output : 4

Left smaller LS[] = {0, 2, 4, 4, 4, 7, 2}

Right smaller RS[] = {0, 3, 7, 3, 3, 3, 0}

Maximum Diff of abs(LS[i] - RS[i]) = 7 - 3 = 4

Input : arr[] = {5, 1, 9, 2, 5, 1, 7}

Output : 1

**Solution. :-**

* Create two empty stacks: leftStack and rightStack.
* Initialize two arrays: leftSmaller and rightSmaller, both filled with zeros, having the same size as the input array.
* Iterate through the array from left to right:
  + Pop elements from the leftStack until the top of the stack is smaller than the current element. For each popped element, update the corresponding index in the leftSmaller array with the popped element.
  + Push the current element onto the leftStack.
* Iterate through the array from right to left:
  + Pop elements from the rightStack until the top of the stack is smaller than the current element. For each popped element, update the corresponding index in the rightSmaller array with the popped element.
  + Push the current element onto the rightStack.
* Calculate the maximum absolute difference by iterating through the array:
  + Compute the absolute difference between leftSmaller[i] and rightSmaller[i].
  + Update the maximum absolute difference if the current difference is greater than the maximum.
* Return the maximum absolute difference.

**def findMaxAbsDifference(arr):**

**n = len(arr)**

**leftStack = []**

**rightStack = []**

**leftSmaller = [0] \* n**

**rightSmaller = [0] \* n**

**maxDiff = 0**

**# Find the nearest left smaller element for each element**

**for i in range(n):**

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

**leftStack.pop()**

**if leftStack:**

**leftSmaller[i] = leftStack[-1]**

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

**# Find the nearest right smaller element for each element**

**for i in range(n-1, -1, -1):**

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

**rightStack.pop()**

**if rightStack:**

**rightSmaller[i] = rightStack[-1]**

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

**# Calculate the maximum absolute difference**

**for i in range(n):**

**diff = abs(leftSmaller[i] - rightSmaller[i])**

**maxDiff = max(maxDiff, diff)**

**return maxDiff**

**arr = [2, 1, 8]**

**maxDiff = findMaxAbsDifference(arr)**

**print(maxDiff)**