**DAY-4 PRACTICE PROBLEMS**

**STACK AND QUEUES**

|  |  |
| --- | --- |
| 1 | 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* |
| 2 | Implement a last-in-first-out (LIFO) using only two queues. The implemented data structure should support all the functions of a normal LIFO (push, top, pop, and empty).  Implement the MyS class:   * void push(int x) Pushes element x to the top of the stack. * int pop() Removes the element on the top of the stack and returns it. * int top() Returns the element on the top of the stack. * boolean empty() Returns true if the stack is empty, false otherwise.   Notes:   * You must use only standard operations of a queue, which means that only push to back, peek/pop from front, size and empty operations are valid. * Depending on your language, the queue may not be supported natively. You may simulate a queue using a list or deque (double-ended queue) as long as you use only a queue's standard operations.     **Example 1:**  Input  ["MyS", "push", "push", "top", "pop", "empty"]  [[], [1], [2], [], [], []]  Output  [null, null, null, 2, 2, false]  Explanation  MyS myStack = new MyS();  myS.push(1);  myS.push(2);  myS.top(); // return 2  myS.pop(); // return 2  myS.empty(); // return False    Constraints:   * 1 <= x <= 9 * At most 100 calls will be made to push, pop, top, and empty. * All the calls to pop and top are valid. |
| 3 | Implement a first in first out (FIFO) using only two stacks. The implemented data structure should support all the functions of a normal queue (push, peek, pop, and empty).  Implement the MyQ class:   * void push(int x) Pushes element x to the back of the queue. * int pop() Removes the element from the front of the queue and returns it. * int peek() Returns the element at the front of the queue. * boolean empty() Returns true if the queue is empty, false otherwise.   Notes:   * You must use only standard operations of a stack, which means only push to top, peek/pop from top, size, and empty operations are valid. * Depending on your language, the stack may not be supported natively. You may simulate a stack using a list or deque (double-ended queue) as long as you use only a stack's standard operations.     Example 1:  Input  ["MyQ", "push", "push", "peek", "pop", "empty"]  [[], [1], [2], [], [], []]  Output  [null, null, null, 1, 1, false]  Explanation  MyQ myQueue = new MyQ();  myQ.push(1); // queue is: [1]  myQ.push(2); // queue is: [1, 2] (leftmost is front of the queue)  myQ.peek(); // return 1  myQ.pop(); // return 1, queue is [2]  myQ.empty(); // return false    Constraints:   * 1 <= x <= 9 * At most 100 calls will be made to push, pop, peek, and empty. * All the calls to pop and peek are valid. |
| 4 | Given a string s, find the first non-repeating character in it and return its index. If it does not exist, return -1.    Example 1:  Input: s = "leetcode"  Output: 0  Example 2:  Input: s = "loveleetcode"  Output: 2  Example 3:  Input: s = "aabb"  Output: -1  Constraints:   * 1 <= s.length <= 105 * s consists of only lowercase English letters. |
| 5 | You have a RecentCounter class which counts the number of recent requests within a certain time frame.  Implement the RecentCounter class:   * RecentCounter() Initializes the counter with zero recent requests. * int ping(int t) Adds a new request at time t, where t represents some time in milliseconds, and returns the number of requests that has happened in the past 3000 milliseconds (including the new request). Specifically, return the number of requests that have happened in the inclusive range [t - 3000, t].   It is guaranteed that every call to ping uses a strictly larger value of t than the previous call.    Example 1:  Input  ["RecentCounter", "ping", "ping", "ping", "ping"]  [[], [1], [100], [3001], [3002]]  Output  [null, 1, 2, 3, 3]  Explanation  RecentCounter recentCounter = new RecentCounter();  recentCounter.ping(1); // requests = [1], range is [-2999,1], return 1  recentCounter.ping(100); // requests = [1, 100], range is [-2900,100], return 2  recentCounter.ping(3001); // requests = [1, 100, 3001], range is [1,3001], return 3  recentCounter.ping(3002); // requests = [1, 100, 3001, 3002], range is [2,3002], return 3    Constraints:   * 1 <= t <= 109 * Each test case will call ping with strictly increasing values of t. * At most 104 calls will be made to ping. |