Wednesday 2/23/22

Infix2postfix expression using stacks:
https://www.youtube.com/watch?v=OVFwgYrMShw&ab_channel=ShivamFrequency
Doubly linked list: https://www.youtube.com/watch?v=JdQeNxWCguQ
DFS, BFS: https://www.youtube.com/watch?v=zaBhtODEL0w&t=76s

HW5 (solve at minimum 5 of the following 6 questions):

1	A valid parentheses string is either empty "", "(" + A + ")", or A + B, where A and B are valid parentheses strings, and + represents string concatenation. ● For example, "", "()", "(())()", and "((()))" are all valid parentheses strings. A valid parentheses string s is primitive if it is nonempty, and there does not exist a way to split it into s = A + B, with A and B nonempty valid parentheses strings. Given a valid parentheses string s, consider its primitive decomposition: s = P1 + P2 + ... + Pk, where Pi are primitive valid parentheses strings. Return s after removing the outermost parentheses of every primitive string in the primitive decomposition of s. Example 1: Input: s = "(()())(())" Output: "()()()" Explanation: The input string is "(()())(())", with primitive decomposition "(()())" + "(())". After removing outer parentheses of each part, this is "()()" + "()" = "()()()". Example 2: Input: s = "(()())(())(()(()))" Output: "()()()()(())" Explanation: The input string is "(()())(())(()(()))", with primitive decomposition "(()())" + "(())" + "(()(()))". After removing outer parentheses of each part, this is "()()" + "()" + "()(())" = "()()()()(())". Example 3: Input: s = "()()" Output: "" Explanation: The input string is "()()", with primitive decomposition "()" + "()". After removing outer parentheses of each part, this is "" + "" = "". Constraints: ● 1 <= s.length <= 105 ● s[i] is either '(' or ')'. ● s is a valid parentheses string. —----------

	class Solution {
	public:
	  string removeOuterParentheses(string s) {
	
	  }
	};

2	You are given a string s consisting only of characters 'a' and 'b'.
	You can delete any number of characters in s to make s balanced. s is balanced if there is no pair of indices (i,j) such that i < j and s[i] = 'b' and s[j]= 'a'.
	Return the minimum number of deletions needed to make s balanced.
	
	Example 1:
	Input: s = "aababbab"
	Output: 2
	Explanation: You can either:
	Delete the characters at 0-indexed positions 2 and 6 ("aababbab" -> "aaabbb"), or
	Delete the characters at 0-indexed positions 3 and 6 ("aababbab" -> "aabbbb").
	
	Example 2:
	Input: s = "bbaaaaabb"
	Output: 2
	Explanation: The only solution is to delete the first two characters.
	
	Constraints:
	● 1 <= s.length <= 105
	● s[i] is 'a' or 'b'.
	—----------
	class Solution {
	public:
	  int minimumDeletions(string s) {
	
	  }
	};

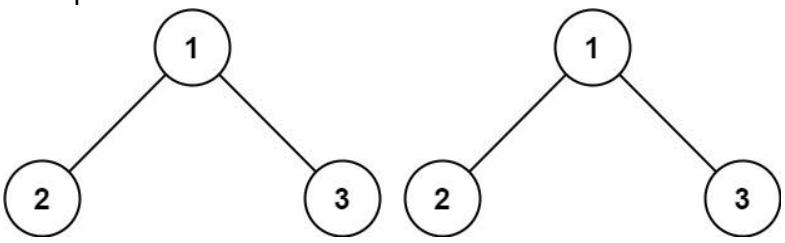
3	Design a stack-like data structure to push elements to the stack and pop the most frequent element from the stack. Implement the FreqStack class: ● FreqStack() constructs an empty frequency stack. ● void push(int val) pushes an integer val onto the top of the stack. ● int pop() removes and returns the most frequent element in the stack. ○ If there is a tie for the most frequent element, the element closest to the stack's top is removed and returned. Example 1: Input ["FreqStack", "push", "push", "push", "push", "push", "push", "pop", "pop", "pop", "pop"] [[], [5], [7], [5], [7], [4], [5], [], [], [], []]

	Output
	[null, null, null, null, null, null, null, 5, 7, 5, 4]
	
	Explanation
	FreqStack freqStack = new FreqStack();
	freqStack.push(5); // The stack is [5]
	freqStack.push(7); // The stack is [5,7]
	freqStack.push(5); // The stack is [5,7,5]
	freqStack.push(7); // The stack is [5,7,5,7]
	freqStack.push(4); // The stack is [5,7,5,7,4]
	freqStack.push(5); // The stack is [5,7,5,7,4,5]
	freqStack.pop(); // return 5, as 5 is the most frequent. The stack becomes [5,7,5,7,4].
	freqStack.pop(); // return 7, as 5 and 7 is the most frequent, but 7 is closest to the top. The stack becomes [5,7,5,4].
	freqStack.pop(); // return 5, as 5 is the most frequent. The stack becomes [5,7,4].
	freqStack.pop(); // return 4, as 4, 5 and 7 is the most frequent, but 4 is closest to the top. The stack becomes [5,7].
	
	Constraints:
	● 0 <= val <= 109
	● At most 2 * 104 calls will be made to push and pop.
	● It is guaranteed that there will be at least one element in the stack before calling pop.
	—----------
	class FreqStack {
	public:
	FreqStack() {
	
	}
	
	void push(int val) {
	
	}
	
	int pop() {
	
	}
	};
	
	/**
	* Your FreqStack object will be instantiated and called as such:
	* FreqStack* obj = new FreqStack();
	* obj->push(val);
	* int param_2 = obj->pop();
	*/

4	Given the roots of two binary trees p and q, write a function to check if they are the same or not.

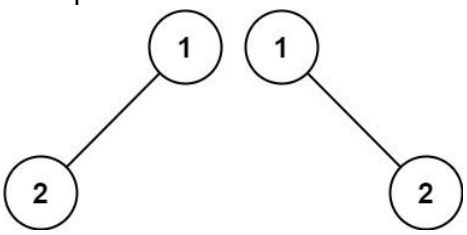
Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.

Example 1:



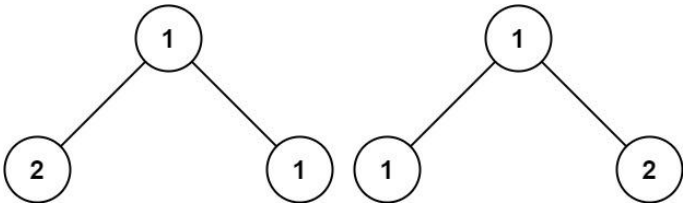
Input: p = [1,2,3], q = [1,2,3]
Output: true

Example 2:



Input: p = [1,2], q = [1,null,2]
Output: false

Example 3:



Input: p = [1,2,1], q = [1,1,2]
Output: false

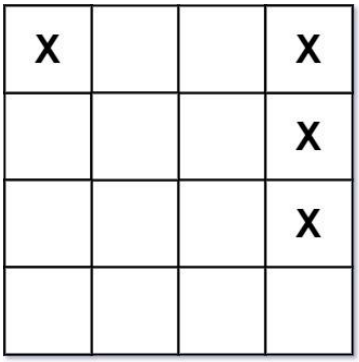
Constraints:

- The number of nodes in both trees is in the range [0, 100].
- -104 <= Node.val <= 104

—----------

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right)
{}
```

	* };
	*/
	class Solution {
	public:
	bool isSameTree(TreeNode* p, TreeNode* q) {
	
	}
	};

5	Given an m x n matrix board where each cell is a battleship 'X' or empty '.', return the number of the battleships on board.
	
	Battleships can only be placed horizontally or vertically on board. In other words, they can only be made of the shape 1 x k (1 row, k columns) or k x 1 (k rows, 1 column), where k can be of any size. At least one horizontal or vertical cell separates between two battleships (i.e., there are no adjacent battleships).
	
	Example 1:
	 
	Input: board = [["X",".",".","X"],[".",".",".","X"],[".",".",".","X"]]
	Output: 2
	
	Example 2:
	Input: board = [["."]]
	Output: 0
	
	Constraints:
	● m == board.length
	● n == board[i].length
	● 1 <= m, n <= 200
	● board[i][j] is either '.' or 'X'.
	
	Follow up: Could you do it in one-pass, using only O(1) extra memory and without modifying the values board?
	—----------
	class Solution {
	public:
	int countBattleships(vector<vector<char>>& board) {

	} };

6	Given an array of strings words (without duplicates), return all the concatenated words in the given list of words.
	A concatenated word is defined as a string that is comprised entirely of at least two shorter words in the given array.
	
	Example 1:
	Input: words = ["cat","cats","catsdogcats","dog","dogcatsdog","hippopotamuses","rat","ratcatdogcat"]
	Output: ["catsdogcats","dogcatsdog","ratcatdogcat"]
	Explanation: "catsdogcats" can be concatenated by "cats", "dog" and "cats";
	"dogcatsdog" can be concatenated by "dog", "cats" and "dog";
	"ratcatdogcat" can be concatenated by "rat", "cat", "dog" and "cat".
	Example 2:
	Input: words = ["cat","dog","catdog"]
	Output: ["catdog"]
	
	Constraints:
	● 1 <= words.length <= 104
	● 0 <= words[i].length <= 30
	● words[i] consists of only lowercase English letters.
	● 0 <= sum(words[i].length) <= 105
	—----------
	class Solution {
	public:
	vector<string> findAllConcatenatedWordsInADict(vector<string>& words) {
	
	}
	};