1. class Solution:

    def levelOrder(self, root: TreeNode) -> List[List[int]]:

        res = []

        if not root:

            return res

        q = []

        q.append(root)

        while q:

            sz = len(q)

            level = []

            for i in range(sz):

                cur = q.pop(0)

                level.append(cur.val)

                if cur.left:

                    q.append(cur.left)

                if cur.right:

                    q.append(cur.right)

            res.append(level)

        return res

2. class Solution:

    def topKFrequent(self, nums: List[int], k: int) -> List[int]:

        valToFreq = {}

        for v in nums:

            valToFreq[v] = valToFreq.get(v, 0) + 1

        pq = []

        for entry in valToFreq.items():

            heapq.heappush(pq, (entry[1], entry[0]))

            if len(pq) > k:

                heapq.heappop(pq)

        res = []

        for i in range(k - 1, -1, -1):

            res.append(heapq.heappop(pq)[1])

        return res[::-1]

class Solution2:

    def topKFrequent(self, nums: List[int], k: int) -> List[int]:

        valToFreq = {}

        for v in nums:

            valToFreq[v] = valToFreq.get(v, 0) + 1

        freqToVals = [[] for \_ in range(len(nums) + 1)]

        for val, freq in valToFreq.items():

            freqToVals[freq].append(val)

        res = []

        for i in range(len(freqToVals) - 1, 0, -1):

            valList = freqToVals[i]

            for j in range(len(valList)):

                res.append(valList[j])

                if len(res) == k:

                    return res

        return res

3. class TrieNode:

    def \_\_init\_\_(self):

        self.children = {}

        self.isWord = False

        self.refs = 0

    def addWord(self, word):

        cur = self

        cur.refs += 1

        for c in word:

            if c not in cur.children:

                cur.children[c] = TrieNode()

            cur = cur.children[c]

            cur.refs += 1

        cur.isWord = True

    def removeWord(self, word):

        cur = self

        cur.refs -= 1

        for c in word:

            if c in cur.children:

                cur = cur.children[c]

                cur.refs -= 1

class Solution:

    def findWords(self, board: List[List[str]], words: List[str]) -> List[str]:

        root = TrieNode()

        for w in words:

            root.addWord(w)

        ROWS, COLS = len(board), len(board[0])

        res, visit = set(), set()

        def dfs(r, c, node, word):

            if (

                r < 0

                or c < 0

                or r == ROWS

                or c == COLS

                or board[r][c] not in node.children

                or node.children[board[r][c]].refs < 1

                or (r, c) in visit

            ):

                return

            visit.add((r, c))

            node = node.children[board[r][c]]

            word += board[r][c]

            if node.isWord:

                node.isWord = False

                res.add(word)

                root.removeWord(word)

            dfs(r + 1, c, node, word)

            dfs(r - 1, c, node, word)

            dfs(r, c + 1, node, word)

            dfs(r, c - 1, node, word)

            visit.remove((r, c))

        for r in range(ROWS):

            for c in range(COLS):

                dfs(r, c, root, "")

        return list(res)