94. Binary Tree Inorder Traversal

# Definition for a binary tree node.

# class TreeNode:

#     def \_\_init\_\_(self, val=0, left=None, right=None):

#         self.val = val

#         self.left = left

#         self.right = right

class Solution:

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

        # res = []

        # def inOrder(root : Optional[TreeNode]):

        #     if root is None:

        #         return None

        #     #inorder Traversal

        #     inOrder(root.left)

        #     res.append(root.val)

        #     inOrder(root.right)

            # preorder Traversal

            # res.append(root.val)

            # inOrder(root.left)

            # inOrder(root.right)

            # post Order Traversal

            # inOrder(root.left)

            # inOrder(root.right)

            # res.append(root.val)

        # inOrder(root)

        # return res

        #inorder Traversal using iteration

        stack = []

        res = []

        cur = root

        # while stack or cur:

        #     while cur:

        #         stack.append(cur)

        #         cur = cur.left

        #     cur = stack.pop()

        #     res.append(cur.val)

        #     cur = cur.right

        while stack or cur:

            if cur is not None:

                stack.append(cur)

                cur = cur.left

            elif stack:

                cur = stack.pop()

                res.append(cur.val)

                cur = cur.right

        return res

        # #preorder

        # if root is None:

        #     return None

        # stack.append(root)

        # while stack:

        #     cur = stack.pop()

        #     res.append(cur.val)

        #     if cur.right is not None:

        #         stack.append(cur.right)

        #     if cur.left is not None:

        #         stack.append(cur.left)

        # return res

104. Maximum Depth of Binary Tree

# Definition for a binary tree node.

# class TreeNode:

#     def \_\_init\_\_(self, val=0, left=None, right=None):

#         self.val = val

#         self.left = left

#         self.right = right

class Solution:

    def maxDepth(self, root: Optional[TreeNode]) -> int:

        if root is None:

            return 0

        # left = self.maxDepth(root.left)

        # right = self.maxDepth(root.right)

        # return 1 + max(left, right)

        # stack = [[root, 1]]

        # res = 1

        # while stack:

        #     root, depth = stack.pop()

        #     if root:

        #         res = max(res, depth)

        #         stack.append([root.right, depth + 1])

        #         stack.append([root.left, depth + 1])

        # return res

        cur = root

        stack = []

        stack.append([root, 1])

        res = 1

        while stack:

            cur , depth = stack.pop()

            if cur.right is not None:

                stack.append([cur.right, depth + 1])

            if cur.left is not None:

                stack.append([cur.left, depth + 1])

            res = max(res, depth)

        return res

        # if root is None:

        #     return 0

        # res = []

        # cur = root

        # stack = []

        # stack.append(root)

        # depth = 1

        # while stack:

        #     found = False

        #     cur = stack.pop()

        #     res.append(cur.val)

        #     if cur.right is not None:

        #         stack.append(cur.right)

        #         found = True

        #     if cur.left is not None:

        #         stack.append(cur.left)

        #         found = True

        #     if found:

        #         depth += 1

        # return depth

101. Symmetric Tree

**Example 1:**



# Definition for a binary tree node.

# class TreeNode:

#     def \_\_init\_\_(self, val=0, left=None, right=None):

#         self.val = val

#         self.left = left

#         self.right = right

class Solution:

    def isSymmetric(self, root: Optional[TreeNode]) -> bool:

        # def symmetric(left : Optional[TreeNode], right: Optional[TreeNode]) -> bool:

            # if left is None or right is None:

            #     return left == right

            # elif left.val != right.val:

            #     return False

        #     if left is None and right is None:

        #         return True

        #     if left is None or right is None or left.val != right.val:

        #         return False

        #     # if left.val != right.val:

        #     #     return False

        #     return symmetric(left.right, right.left) and symmetric(left.left, right.right)

        # if root is None:

        #     return True

        # else:

        #     return symmetric(root.left, root.right)

        #iterative approach

        if root is None:

            return True

        stack = [[root.left, root.right]]

        # stack = []

        # stack.append([root.left, root.right])

        while stack:

            l, r = stack.pop()

            if l is None and r is None:

                continue

            if l is None or r is None or (l.val != r.val):

                return False

            stack.append([l.left, r.right])

            stack.append([l.right, r.left])

        return True

    #same tree

# class Solution:

#     def isSameTree(self, p : TreeNode, q: TreeNode) -> bool:

#         if p is None and q is None:

#             return True

#         if p is None or q is None:

#             return False

#         if p.val != q.val:

#             return False

#         return isSameTree(p.left, q.left) and isSameTree(p.right, q.right)

141. Linked List Cycle

**Example 1:**



# Definition for singly-linked list.

# class ListNode:

#     def \_\_init\_\_(self, x):

#         self.val = x

#         self.next = None

class Solution:

    def hasCycle(self, head: Optional[ListNode]) -> bool:

        # if head is None:

        #     return False

        # if head.next is None:

        #     return False

        # cur = head

        # s = set()

        # while cur:

        #     if cur in s:

        #         return True

        #     s.add(cur)

        #     cur = cur.next

        # return False

        #to use less memory

        fast, slow = head, head

        while fast and fast.next:

            slow = slow.next

            fast = fast.next.next

            if fast == slow:

                return True

        return False

        # f , s= head, head

        # while f:

        #     if s is not None:

        #         s = s.next

        #     if f is None:

        #         return False

        #     else:

        #         f = f.next

        #         if f is None:

        #             return False

        #         else:

        #             f = f.next

        #     if f == s:

        #         return True

206. Reverse Linked List

**Example 1:**



# Definition for singly-linked list.

# class ListNode:

#     def \_\_init\_\_(self, val=0, next=None):

#         self.val = val

#         self.next = next

class Solution:

    def reverseList(self, head: Optional[ListNode]) -> Optional[ListNode]:

        # if head is None:

        #     return None

        # if head.next is None:

        #     return head

        #iterative approach

        prev, cur = None, head

        while cur:

            temp = cur.next

            cur.next = prev

            prev = cur

            cur = temp

        return prev

        #recursion

        # def reverse(cur, prev):

        #     if cur is None:

        #         return prev

        #     next = cur.next

        #     cur.next = prev

        #     return reverse(next, cur)

        # # if head is None:

        # #     return None

        # return reverse(head, None)

160. Intersection of Two Linked Lists

For example, the following two linked lists begin to intersect at node c1:



# Definition for singly-linked list.

# class ListNode:

#     def \_\_init\_\_(self, x):

#         self.val = x

#         self.next = None

class Solution:

    def getIntersectionNode(self, headA: ListNode, headB: ListNode) -> Optional[ListNode]:

        if headA is None or headB is None:

            return False

        # s = set()

        # while headA or headB:

        #     if headA is not None:

        #         if headA in s:

        #             return headA

        #         s.add(headA)

        #         headA = headA.next

        #     if headB is not None:

        #         if headB in s:

        #             return headB

        #         s.add(headB)

        #         headB = headB.next

        # return None

        # s = set()

        # while headA:

        #     s.add(headA)

        #     headA = headA.next

        # while headB:

        #     if headB in s:

        #         return headB

        #     headB = headB.next

        # return None

        #using less memory

        A, B = headA, headB

        while A != B:

            if A is not None:

                A = A.next

            else:

                A = headB

            if B is not None:

                B = B.next

            else:

                B = headA

        return A

        # A, B = headA, headB

        # while A or B:

        #     if A == B:t

        #         return A

        #     if A is not None:

        #         A = A.next

        #     else:

        #         A = headB

        #     if B is not None:

        #         B = B.next

        #     else:

        #         B = headA

        # return A

        # A, B = headA, headB

        # while A != B:

        #     A = A.next if A else headB

        #     B = B.next if B else headA

        # return A

        # l, r = headA, headB

        # A, B = headA, headB

        # a, b = 1, 1

        # while headA:

        #     headA = headA.next

        #     a += 1

        # while headB:

        #     headB = headB.next

        #     b += 1

        # if a < b:

        #     for i in range(b-a):

        #         B = B.next

        #     while B:

        #         if B == l:

        #             return B

        #         B = B.next

        #         l = l.next

        #     return None

        # else:

        #     for i in range(a-b):

        #         A = A.next

        #     while A:

        #         if A == r:

        #             return A

        #         A = A.next

        #         r = r.next

        #     return None

234. Palindrome Linked List

**Example 1:**



**Input:** head = [1,2,2,1]

**Output:** true

# Definition for singly-linked list.

# class ListNode:

#     def \_\_init\_\_(self, val=0, next=None):

#         self.val = val

#         self.next = next

class Solution:

    def isPalindrome(self, head: Optional[ListNode]) -> bool:

        # using array so requires extra memory

        # nums = []

        # while head:

        #     nums.append(head.val)

        #     head = head.next

        # l, r = 0, len(nums) - 1

        # while l < r:

        #     if nums[l] != nums[r]:

        #         return False

        #     l, r = l + 1, r - 1

        # return True

        #using less memory

        #finding mid point - slow

        fast, slow = head, head

        while fast and fast.next:

            slow = slow.next

            fast = fast.next.next

        #reversing second half of the list

        prev = None

        while slow:

            temp = slow.next

            slow.next = prev

            prev = slow

            slow = temp

        #now find if palindrome or not

        left, right = head, prev

        # while prev:

        #     if left.val != prev.val:

        #         return False

        #     left = left.next

        #     prev = prev.next

        # return True

        while right:

            if left.val != right.val:

                return False

            left = left.next

            right = right.next

        return True

2. Add Two Numbers (Medium)

**Example 1:**



**Input:** l1 = [2,4,3], l2 = [5,6,4]

**Output:** [7,0,8]

**Explanation:** 342 + 465 = 807.

# Definition for singly-linked list.

# class ListNode:

#     def \_\_init\_\_(self, val=0, next=None):

#         self.val = val

#         self.next = next

class Solution:

    def addTwoNumbers(self, l1: Optional[ListNode], l2: Optional[ListNode]) -> Optional[ListNode]:

        temp = cur = ListNode()

        carry = 0

        while l1 or l2 or carry:

            if l1 is not None:

                v1 = l1.val

            else:

                v1 = 0

            v2 = l2.val if l2 else 0

            #new digit

            val = v1 + v2 + carry

            carry = val // 10

            val = val % 10

            cur.next = ListNode(val)

            #update pointers

            cur = cur.next

            l1 = l1.next if l1 else None

            if l2:

                l2 = l2.next

            else:

                l2 = None

        return temp.next

19. Remove Nth Node from End of List (Medium)

**Example 1:**



**Input:** head = [1,2,3,4,5], n = 2

**Output:** [1,2,3,5]

# Definition for singly-linked list.

# class ListNode:

#     def \_\_init\_\_(self, val=0, next=None):

#         self.val = val

#         self.next = next

class Solution:

    def removeNthFromEnd(self, head: Optional[ListNode], n: int) -> Optional[ListNode]:

        temp = ListNode(0, head)

        left = temp

        right = head

        for i in range(n):

            print(right.val)

            right = right.next

        # if right:

        #     print(right.val)

        while right:

            left = left.next

            right = right.next

        print(left.val)

        left.next = left.next.next

        return temp.next

        # if head is None:

        #     return None

        # if head.next is None:

        #     return None

        # a = head; b = head

        # x = 0

        # print(head.val)

        # while a:

        #     x += 1

        #     a = a.next

        # print(x)

        # if n == x:

        #     b = b.next

        # for i in range(x - n - 1):

        #     head = head.next

        # print(head.val)

        # temp = head.next

        # head.next = temp.next

        # return b