CODING INTERVIEW PREPARATION

Day 3: Linked Lists & Stacks

Complete Study Guide with Multiple Solutions

TABLE OF CONTENTS

Page

DAY 3: LINKED LISTS & STACKS

Key Concepts for Today:

- **Singly Linked List** manipulation and traversal
- **Dummy nodes** for edge case handling
- **Stack** for validation and expression evaluation
- **Monotonic Stack** for maintaining order properties
- **Two-pointer techniques** for linked list problems
- **Iterative vs Recursive** approaches

Problems Index:

CORE CONCEPTS EXPLANATION

Singly Linked List Fundamentals

Definition: A linear data structure where elements (nodes) are stored in sequence, with each node containing data and a reference to the next node.

Node Structure:

```
public class ListNode
{
    public int val;
    public ListNode next;
    public ListNode(int val=0, ListNode next=null)
    {
        this.val = val;
        this.next = next;
    }
}
```

Key Operations:

- **Traversal**: Moving through nodes using `next` pointers
- **Insertion**: Creating new nodes and adjusting pointers
- **Deletion**: Removing nodes by updating pointer references
- **Reversal**: Changing direction of `next` pointers

Dummy Node Pattern

Purpose: Simplifies edge case handling by providing a consistent starting point.

When to Use:

- When the head node might be removed or changed
- When merging or creating new linked lists

- When operations need to handle empty lists uniformly

Implementation Pattern:

```
ListNode dummy = new ListNode(0);
ListNode current = dummy;
// Perform operations...
return dummy.next; // Return actual head
```

Stack Data Structure

LIFO Principle: Last In, First Out - elements are added and removed from the top.

Core Operations:

- **Push**: Add element to top - **O(1)**
- **Pop**: Remove element from top - **O(1)**
- **Peek/Top**: View top element without removing - **O(1)**
- **IsEmpty**: Check if stack is empty - **O(1)**

Common Applications:

- **Expression evaluation**: Parentheses matching, postfix notation
- **Function call management**: Call stack in programming
- **Undo operations**: Browser history, text editors
- **Parsing**: Syntax analysis, balanced brackets

Monotonic Stack

Definition: A stack where elements are maintained in either increasing or decreasing order.

Types:

- **Monotonic Increasing**: Elements increase from bottom to top
- **Monotonic Decreasing**: Elements decrease from bottom to top

Applications:

- Finding next greater/smaller elements
- Histogram problems (largest rectangle)
- Temperature problems (next warmer day)

Two-Pointer Techniques for Linked Lists

Fast and Slow Pointers:

- **Cycle Detection**: Fast moves 2 steps, slow moves 1 step
- **Finding Middle**: When fast reaches end, slow is at middle
- **Kth from End**: Start second pointer k steps behind first

Two Separate Pointers:

- **Merging**: One pointer for each list
- **Comparison**: Moving through lists simultaneously

Why These Concepts Matter in Interviews

Problem-Solving Patterns:

- **Linked Lists**: Test pointer manipulation and edge case handling
- **Stacks**: Demonstrate understanding of LIFO and recursion simulation
- **Memory Management**: Show awareness of object references and cleanup

Real-world Applications:

- **Operating Systems**: Process scheduling, memory allocation
- **Databases**: Index structures, query processing
- **Compilers**: Expression parsing, syntax tree construction
- **Web Browsers**: History management, DOM manipulation

PROBLEM 1: REVERSE LINKED LIST

Problem Statement:

Given the head of a singly linked list, reverse the list, and return the reversed list.[1]

Test Cases:

```
Example 1:
Input: head = [1,2,3,4,5]
Output: [5,4,3,2,1]
Explanation: The linked list is reversed so 5 points to 4, 4 to 3, etc.

Example 2:
Input: head = [1,2]
Output: [2,1]
Explanation: Simple two-node reversal.

Example 3:
Input: head = []
```

```
Output: []
Explanation: Empty list remains empty.
```

Interview Questions & Answers:

1. **Q:** "Can you reverse the list in-place without extra space?"
   A: Yes, using iterative approach with three pointers: previous, current, and next. This uses O(1) space.
2. **Q:** "What's the difference between iterative and recursive approaches?"
   A: Iterative uses O(1) space and explicit pointers. Recursive uses O(n) space due to call stack but can be more intuitive.
3. **Q:** "How do you handle edge cases like empty list or single node?"
   A: Empty list returns null, single node returns itself unchanged. Both are natural base cases.
4. **Q:** "Can you reverse only a portion of the linked list?"
   A: Yes, that's "Reverse Linked List II" - you'd need start and end positions and more complex pointer management.
5. **Q:** "What happens if you don't properly manage the pointers?"
   A: You'll lose references to nodes, causing memory leaks, or create infinite loops, or lose parts of the list.

Approach 1: Recursive

Idea: Recursively reverse the rest of the list, then adjust current node's pointer.

Time Complexity: O(n) - visit each node once
Space Complexity: O(n) - recursion call stack

```
using System;

public class Solution
{
    public ListNode ReverseList(ListNode head)
    {
        // Base cases: empty list or single node
        if (head == null || head.next == null)
        {
            return head;
        }

        // Recursively reverse the rest of the list
        ListNode reversedHead = ReverseList(head.next);

        // Reverse the current connection
        head.next.next = head;
        head.next = null;

        return reversedHead;
    }
}
```

Approach 2: Iterative with Stack

Explanation: Push all nodes onto stack, then pop them off to rebuild list in reverse order.

Time Complexity: O(n) - two passes through the list
Space Complexity: O(n) - stack storage

```
using System;
using System.Collections.Generic;

public class Solution
{
    public ListNode ReverseList(ListNode head)
    {
        if (head == null) return null;

        // Push all nodes onto stack
        var stack = new Stack<ListNode>();
        ListNode current = head;

        while (current != null)
        {
            stack.Push(current);
            current = current.next;
        }

        // Pop nodes to create reversed list
        ListNode newHead = stack.Pop();
        current = newHead;

        while (stack.Count > 0)
        {
            current.next = stack.Pop();
            current = current.next;
        }

        current.next = null; // Important: terminate the list
        return newHead;
    }
}
```

Approach 3: Optimal - Iterative Three Pointers

Detailed Reasoning: Use three pointers to track previous, current, and next nodes. This allows us to reverse links while maintaining references to continue traversal. This is optimal because it achieves reversal in single pass with constant space.

Time Complexity: O(n) - single pass through list
Space Complexity: O(1) - only using pointer variables

```
using System;

public class Solution
{
    /// <summary>
    /// Reverses a singly linked list using iterative three-pointer technique
    /// Time: O(n), Space: O(1)
    /// </summary>
    public ListNode ReverseList(ListNode head)
    {
        // Initialize three pointers
        ListNode prev = null;    // Previous node (will become next)
        ListNode current = head; // Current node being processed

        while (current != null)
        {
            // Store next node before we lose reference
            ListNode nextTemp = current.next;

            // Reverse the link: current now points to previous
            current.next = prev;

            // Move pointers forward for next iteration
            prev = current;      // Previous becomes current
            current = nextTemp;  // Current becomes next
        }

        // prev is now the new head of reversed list
        return prev;
    }
}
```

Key Takeaways:

- **Three-pointer pattern**: Essential technique for linked list manipulation
- **Pointer management**: Always store next reference before modifying current.next
- **Edge case handling**: Empty and single-node lists are natural base cases

PROBLEM 2: MERGE TWO SORTED LISTS

Problem Statement:

You are given the heads of two sorted linked lists `list1` and `list2`. Merge the two lists into one sorted list. The list should be made by splicing together the nodes of the first two lists. Return the head of the merged linked list.[2]

Test Cases:

```
Example 1:
Input: list1 = [1,2,4], list2 = [1,3,4]
Output: [1,1,2,3,4,4]
Explanation: Merge by comparing values and maintaining sorted order.

Example 2:
Input: list1 = [], list2 = []
Output: []
Explanation: Both lists empty, result is empty.

Example 3:
Input: list1 = [], list2 = [^0]
Output: [^0]
Explanation: One empty list, return the other list.
```

Interview Questions & Answers:

1. **Q:** "How do you handle lists of different lengths?"
   A: After one list is exhausted, append the remaining nodes from the other list directly since they're already sorted.
2. **Q:** "Can you merge the lists in-place without creating new nodes?"
   A: Yes, by adjusting the next pointers of existing nodes instead of creating new nodes.
3. **Q:** "What if the lists are not sorted?"
   A: The problem assumes sorted input. For unsorted lists, you'd need to sort first or use a different merging strategy.
4. **Q:** "How would you extend this to merge k sorted lists?"
   A: Use divide-and-conquer approach, merging pairs of lists until only one remains, or use a min-heap.
5. **Q:** "What's the advantage of using a dummy node?"
   A: Dummy node eliminates edge case handling for the first element and simplifies pointer management.

Approach 1: Recursive

Idea: Compare heads of both lists, choose smaller one, recursively merge the rest.

Time Complexity: O(m + n) where m, n are list lengths
Space Complexity: O(m + n) - recursion call stack

```
using System;

public class Solution
{
    public ListNode MergeTwoLists(ListNode list1, ListNode list2)
    {
        // Base cases: one list is empty
        if (list1 == null) return list2;
        if (list2 == null) return list1;

        // Choose smaller head and recursively merge rest
        if (list1.val <= list2.val)
        {
            list1.next = MergeTwoLists(list1.next, list2);
            return list1;
        }
        else
        {
            list2.next = MergeTwoLists(list1, list2.next);
            return list2;
        }
    }
}
```

Approach 2: Iterative without Dummy Node

Explanation: Compare heads to determine the starting node, then iterate through both lists merging nodes.

Time Complexity: O(m + n) - single pass through both lists
Space Complexity: O(1) - only using pointer variables

```
using System;

public class Solution
{
    public ListNode MergeTwoLists(ListNode list1, ListNode list2)
    {
        if (list1 == null) return list2;
        if (list2 == null) return list1;

        // Determine the head of merged list
        ListNode head, current;
        if (list1.val <= list2.val)
        {
            head = current = list1;
            list1 = list1.next;
        }
        else
        {
            head = current = list2;
            list2 = list2.next;
        }

        // Merge remaining nodes
        while (list1 != null && list2 != null)
        {
            if (list1.val <= list2.val)
            {
                current.next = list1;
                list1 = list1.next;
            }
            else
            {
                current.next = list2;
                list2 = list2.next;
            }
            current = current.next;
        }

        // Append remaining nodes
        current.next = list1 ?? list2;

        return head;
    }
}
```

Approach 3: Optimal - Dummy Node

Detailed Reasoning: Using a dummy node eliminates the need to handle the first element specially and makes the code cleaner and more uniform. This is the preferred approach in most implementations.

Time Complexity: O(m + n) - visit each node exactly once
Space Complexity: O(1) - constant extra space

```
using System;

public class Solution
{
    /// <summary>
    /// Merges two sorted linked lists using dummy node technique
    /// Time: O(m + n), Space: O(1)
    /// </summary>
    public ListNode MergeTwoLists(ListNode list1, ListNode list2)
    {
        // Create dummy node to simplify edge cases
        ListNode dummy = new ListNode(0);
        ListNode current = dummy;

        // Merge nodes while both lists have elements
        while (list1 != null && list2 != null)
        {
            if (list1.val <= list2.val)
            {
                current.next = list1;
                list1 = list1.next;
            }
            else
            {
                current.next = list2;
                list2 = list2.next;
            }
            current = current.next;
        }
        // Append remaining nodes from non-empty list
        // One of these will be null, so this handles both cases
        current.next = list1 ?? list2;

        // Return merged list (skip dummy node)
        return dummy.next;
    }
}
```

Key Takeaways:

- **Dummy node pattern**: Simplifies edge case handling for list construction
- **Two-pointer traversal**: Advance pointers based on comparison results
- **Efficient appending**: Remaining sorted elements can be appended directly

PROBLEM 3: REORDER LIST

Problem Statement:

You are given the head of a singly linked-list. The list can be represented as: $L_0 \to L_1 \to \ldots \to L_{n-1} \to L_n$. Reorder the list to be on the following form: $L_0 \to L_n \to L_1 \to L_{n-1} \to L_2 \to L_{n-2} \to \ldots$ You may not modify the values in the list's nodes. Only nodes themselves may be changed.[3]

Test Cases:

```
Example 1:
Input: head = [1,2,3,4]
Output: [1,4,2,3]
Explanation: Reorder as 1 -> 4 -> 2 -> 3

Example 2:
Input: head = [1,2,3,4,5]
Output: [1,5,2,4,3]
Explanation: Reorder as 1 -> 5 -> 2 -> 4 -> 3

Example 3:
Input: head = [^1]
Output: [^1]
Explanation: Single node remains unchanged.
```

Interview Questions & Answers:

1. **Q:** "How do you find the middle of the linked list?"
   A: Use fast and slow pointers. When fast reaches the end, slow will be at the middle (or just before middle for even-length lists).
2. **Q:** "Why do you need to reverse the second half?"
   A: To access elements from the end of the list easily. After reversing, the last element

becomes the first element of the second half.

3. **Q:** "How do you merge the two halves in alternating fashion?"
   A: Use two pointers, one for each half. Alternate between picking nodes from first half and second half.
4. **Q:** "What if the list has even vs odd number of nodes?"
   A: The algorithm works for both. For odd length, the middle element stays in the first half.
5. **Q:** "Can you do this without modifying the original list structure?"
   A: The problem requires in-place modification. You could use extra space to store values and rebuild, but that's less efficient.

Approach 1: Using Extra Space (Array)

Idea: Store all nodes in an array, then reorder by reconstructing the list with alternating indices.

Time Complexity: O(n) - traverse list and reconstruct
Space Complexity: O(n) - array to store all nodes

```
using System;
using System.Collections.Generic;

public class Solution
{
    public void ReorderList(ListNode head)
    {
        if (head == null || head.next == null) return;

        // Store all nodes in a list
        var nodes = new List<ListNode>();
        ListNode current = head;

        while (current != null)
        {
            nodes.Add(current);
            current = current.next;
        }

        // Reorder by alternating between start and end
        int left = 0, right = nodes.Count - 1;

        while (left < right)
        {
            nodes[left].next = nodes[right];
            left++;

            if (left < right)
            {
                nodes[right].next = nodes[left];
                right--;
            }
        }

        // Terminate the list
        nodes[left].next = null;
    }
}
```

Approach 2: Stack for Second Half

Explanation: Find the middle, push second half onto stack, then merge first half with popped elements from stack.

Time Complexity: O(n) - multiple passes through the list
Space Complexity: O(n/2) = O(n) - stack for second half

```
using System;
using System.Collections.Generic;

public class Solution
{
    public void ReorderList(ListNode head)
    {
        if (head == null || head.next == null) return;

        // Find the middle using slow and fast pointers
        ListNode slow = head, fast = head;
        while (fast.next != null && fast.next.next != null)
        {
            slow = slow.next;
            fast = fast.next.next;
        }

        // Push second half onto stack
        var stack = new Stack<ListNode>();
        ListNode current = slow.next;
        while (current != null)
        {
            stack.Push(current);
            current = current.next;
        }

        // Cut the list in half
        slow.next = null;

        // Merge first half with stack elements
        current = head;
        while (stack.Count > 0)
        {
            ListNode next = current.next;
            ListNode stackNode = stack.Pop();

            current.next = stackNode;
            stackNode.next = next;
            current = next;
        }
    }
}
```

Approach 3: Optimal - Find Middle, Reverse, Merge

Detailed Reasoning: This three-step approach is optimal because it achieves the reordering in-place with O(1) extra space. The key insight is that reordering is equivalent to merging the first half with the reversed second half.

Time Complexity: O(n) - three linear passes
Space Complexity: O(1) - only using pointer variables

```
using System;

public class Solution
{
    /// <summary>
    /// Reorders linked list using find middle, reverse, and merge technique
    /// Time: O(n), Space: O(1)
    /// </summary>
    public void ReorderList(ListNode head)
    {
        if (head == null || head.next == null) return;

        // Step 1: Find the middle of the list
        ListNode slow = head, fast = head;
        while (fast.next != null && fast.next.next != null)
        {
            slow = slow.next;
            fast = fast.next.next;
        }

        // Step 2: Reverse the second half
        ListNode secondHalf = ReverseList(slow.next);
        slow.next = null; // Cut the list in half

        // Step 3: Merge the two halves alternately
        ListNode first = head;
        ListNode second = secondHalf;

        while (second != null)
        {
            ListNode firstNext = first.next;
            ListNode secondNext = second.next;

            first.next = second;
            second.next = firstNext;

            first = firstNext;
            second = secondNext;
        }
    }

    private ListNode ReverseList(ListNode head)
    {
        ListNode prev = null;
        ListNode current = head;

        while (current != null)
        {
            ListNode next = current.next;
            current.next = prev;
            prev = current;
            current = next;
        }

        return prev;
    }
}
```

Key Takeaways:

- **Multi-step approach**: Complex problems often break down into simpler sub-problems
- **Fast/slow pointer technique**: Standard method for finding middle of linked list
- **In-place reversal**: Reuse existing list reversal technique for efficiency

PROBLEM 4: REMOVE NTH NODE FROM END OF LIST

Problem Statement:

Given the head of a linked list, remove the nth node from the end of the list and return its head. [4]

Test Cases:

```
Example 1:
Input: head = [1,2,3,4,5], n = 2
Output: [1,2,3,5]
Explanation: Remove the 2nd node from end (value 4).

Example 2:
Input: head = [^1], n = 1
Output: []
Explanation: Remove the only node, list becomes empty.

Example 3:
Input: head = [1,2], n = 1
Output: [^1]
Explanation: Remove last node (value 2).
```

Interview Questions & Answers:

1. **Q:** "What if n is greater than the length of the list?"
   A: Problem assumes valid input, but in practice, you should validate and handle this edge case appropriately.
2. **Q:** "How do you remove the head node efficiently?"
   A: Use a dummy node so that removing the head is handled the same way as removing any other node.
3. **Q:** "Can you solve this in one pass without knowing the list length?"
   A: Yes, use two pointers separated by n positions. When the front pointer reaches the end, the back pointer is at the node to remove.
4. **Q:** "What's the key insight for the two-pointer approach?"
   A: Position the pointers so that when the first reaches the end, the second is exactly at the position we need to modify.
5. **Q:** "How do you handle removing the last node?"
   A: The algorithm handles this naturally - the second pointer will be at the second-to-last node, which should point to null.

Approach 1: Two-Pass Algorithm

Idea: First pass to count nodes, second pass to remove the (length - n + 1)th node from the beginning.

Time Complexity: O(n) - two passes through the list
Space Complexity: O(1) - only using counter and pointers

```
using System;

public class Solution
{
    public ListNode RemoveNthFromEnd(ListNode head, int n)
    {
        // First pass: count the total number of nodes
        int length = 0;
        ListNode current = head;
        while (current != null)
        {
            length++;
            current = current.next;
        }

        // Handle edge case: removing the head
        if (length == n)
        {
            return head.next;
        }

        // Second pass: find the node before the one to remove
        current = head;
        for (int i = 0; i < length - n - 1; i++)
        {
            current = current.next;
        }

        // Remove the nth node from end
        current.next = current.next.next;

        return head;
    }
}
```

Approach 2: Recursive Approach

Explanation: Use recursion to traverse to the end, then count backwards to find and remove the nth node.

Time Complexity: O(n) - single traversal
Space Complexity: O(n) - recursion call stack

```
using System;

public class Solution
{
    private int count = 0;

    public ListNode RemoveNthFromEnd(ListNode head, int n)
    {
        // Reset count for each function call
        count = 0;
        return RemoveHelper(head, n);
    }

    private ListNode RemoveHelper(ListNode head, int n)
    {
        if (head == null) return null;

        // Recursively process the rest of the list
        head.next = RemoveHelper(head.next, n);

        // Count nodes from the end
        count++;

        // If this is the nth node from end, remove it
        if (count == n)
        {
            return head.next;
        }

        return head;
    }
}
```

Approach 3: Optimal - One Pass with Two Pointers

Detailed Reasoning: Use two pointers separated by n positions. When the first pointer reaches the end, the second pointer will be at the node just before the one we need to remove. Using a dummy node eliminates edge case handling.

Time Complexity: O(n) - single pass through the list
Space Complexity: O(1) - only using pointer variables

```
using System;

public class Solution
{
    /// <summary>
    /// Removes nth node from end using one-pass two-pointer technique
    /// Time: O(n), Space: O(1)
    /// </summary>
    public ListNode RemoveNthFromEnd(ListNode head, int n)
    {
        // Use dummy node to handle edge case of removing head
        ListNode dummy = new ListNode(0);
        dummy.next = head;

        ListNode first = dummy;  // First pointer
        ListNode second = dummy; // Second pointer

        // Move first pointer n+1 steps ahead
        // This ensures second pointer will be at node before target
        for (int i = 0; i <= n; i++)
        {
            first = first.next;
        }

        // Move both pointers until first reaches the end
        while (first != null)
        {
            first = first.next;
            second = second.next;
        }

        // Remove the nth node from end
        second.next = second.next.next;

        return dummy.next;
    }
}
```

Key Takeaways:

- **Two-pointer separation**: Position pointers to achieve desired relative positioning
- **Dummy node advantage**: Eliminates special handling for head node removal
- **One-pass optimization**: Achieve result without knowing list length in advance

PROBLEM 5: LINKED LIST CYCLE

Problem Statement:

Given `head`, the head of a linked list, determine if the linked list has a cycle in it. There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the `next` pointer.[5]

Test Cases:

```
Example 1:
Input: head = [3,2,0,-4], pos = 1
Output: true
Explanation: There is a cycle where the tail connects to the 1st node (0-indexed).

Example 2:
Input: head = [1,2], pos = 0
Output: true
Explanation: There is a cycle where the tail connects to the 0th node.

Example 3:
Input: head = [^1], pos = -1
Output: false
Explanation: There is no cycle in the linked list.
```

Interview Questions & Answers:

1. **Q:** "How does the Floyd's Cycle Detection (tortoise and hare) algorithm work?"
   A: Fast pointer moves 2 steps, slow moves 1 step. If there's a cycle, fast will eventually "lap" slow and they'll meet.
2. **Q:** "Why does the fast pointer eventually catch the slow pointer in a cycle?"
   A: In each iteration, the distance between pointers decreases by 1. Eventually, the distance becomes 0 and they meet.
3. **Q:** "What if you need to find where the cycle starts?"
   A: That's "Linked List Cycle II" - after detecting cycle, reset one pointer to head and move both at same speed until they meet.
4. **Q:** "Can you solve this with constant space without modifying the list?"
   A: Yes, Floyd's algorithm uses O(1) space. Alternative O(n) space solution uses HashSet to track visited nodes.

5. **Q:** "What happens if the fast pointer moves 3 steps instead of 2?"
   A: Algorithm still works, but might be less efficient. The key is that fast moves more steps than slow.

Approach 1: HashSet to Track Visited Nodes

Idea: Keep track of all visited nodes in a HashSet. If we encounter a node we've seen before, there's a cycle.

Time Complexity: O(n) - visit each node at most once
Space Complexity: O(n) - HashSet storage

```
using System;
using System.Collections.Generic;

public class Solution
{
    public bool HasCycle(ListNode head)
    {
        var visited = new HashSet<ListNode>();
        ListNode current = head;

        while (current != null)
        {
            // If we've seen this node before, there's a cycle
            if (visited.Contains(current))
            {
                return true;
            }

            // Mark current node as visited
            visited.Add(current);
            current = current.next;
        }

        // Reached end of list without finding cycle
        return false;
    }
}
```

Approach 2: Modify Nodes (Destructive)

Explanation: Mark visited nodes by modifying their values to a special marker. If we encounter a marked node, there's a cycle.

Time Complexity: O(n) - single pass through list
Space Complexity: O(1) - no extra space
Note: This modifies the original list structure

```
using System;

public class Solution
```

```
{
    public bool HasCycle(ListNode head)
    {
        ListNode current = head;

        while (current != null)
        {
            // Use a special value to mark visited nodes
            if (current.val == int.MinValue)
            {
                return true; // Found a marked node = cycle
            }

            // Mark current node as visited
            current.val = int.MinValue;
            current = current.next;
        }

        return false;
    }
}
```

Approach 3: Optimal - Floyd's Cycle Detection (Two Pointers)

Detailed Reasoning: Use two pointers moving at different speeds. If there's a cycle, the faster pointer will eventually "lap" the slower one within the cycle. This is optimal because it uses constant space and doesn't modify the original list.

Time Complexity: O(n) - in worst case, traverse entire list
Space Complexity: O(1) - only using two pointers

```
using System;

public class Solution
{
    /// <summary>
    /// Detects cycle in linked list using Floyd's cycle detection algorithm
    /// Time: O(n), Space: O(1)
    /// </summary>
    public bool HasCycle(ListNode head)
    {
        // Edge case: empty list or single node without cycle
        if (head == null || head.next == null) return false;

        // Initialize two pointers
        ListNode slow = head;      // Moves 1 step at a time
        ListNode fast = head.next; // Moves 2 steps at a time

        // Continue until fast pointer reaches end or pointers meet
        while (fast != null && fast.next != null)
        {
            // If pointers meet, there's a cycle
            if (slow == fast)
            {
                return true;
            }

            // Move pointers
            slow = slow.next;           // Move 1 step
            fast = fast.next.next;      // Move 2 steps
        }

        // Fast pointer reached end, no cycle
        return false;
    }
}
```

Key Takeaways:

- **Floyd's algorithm**: Classic two-pointer technique for cycle detection
- **Speed differential**: Different speeds ensure meeting if cycle exists
- **Space efficiency**: O(1) space solution preferred over O(n) HashSet approach

PROBLEM 6: MIN STACK

Problem Statement:

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time. Implement the MinStack class:[6]

- `MinStack()` initializes the stack object.
- `void push(int val)` pushes the element val onto the stack.
- `void pop()` removes the element on the top of the stack.
- `int top()` gets the top element of the stack.
- `int getMin()` retrieves the minimum element in the stack.

You must implement a solution with **O(1)** time complexity for each function.

Test Cases:

```
Example 1:
Input: ["MinStack","push","push","push","getMin","pop","top","getMin"]
       [[],[-2],[^0],[-3],[],[],[],[]]
Output: [null,null,null,null,-3,null,0,-2]
Explanation:
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin(); // return -3
minStack.pop();
minStack.top();    // return 0
minStack.getMin(); // return -2
```

```
Example 2:
Input: ["MinStack","push","push","getMin","getMin","pop","getMin"]
       [[],[^1],[^2],[],[],[],[]]
Output: [null,null,null,1,1,null,1]

Example 3:
Input: ["MinStack","push","push","push","pop","pop","pop","getMin"]
       [[],[^1],[^2],[^0],[],[],[],[]]
Output: [null,null,null,null,null,null,null,error]
Explanation: Stack becomes empty, getMin() on empty stack should handle gracefully.
```

Interview Questions & Answers:

1. **Q:** "How do you maintain the minimum in O(1) time when elements are popped?"
   A: Use auxiliary stack to track minimums, or encode minimum information with each element.
2. **Q:** "What happens when the minimum element is popped?"
   A: The previous minimum becomes the new minimum. Auxiliary stack automatically handles this by maintaining minimum at each level.
3. **Q:** "Can you implement this with a single stack?"
   A: Yes, using encoding technique where you store the difference from minimum, but it's more complex and has limitations.
4. **Q:** "How do you handle duplicate minimum values?"
   A: Push onto min stack whenever new element is less than or equal to current minimum (not just less than).
5. **Q:** "What's the space complexity trade-off?"
   A: Using auxiliary stack takes O(n) extra space but keeps all operations O(1). Single stack approaches are more complex.

Approach 1: Two Stacks - Simple

Idea: Use main stack for elements and auxiliary stack for tracking minimums.

Time Complexity: O(1) for all operations
Space Complexity: O(n) - worst case, auxiliary stack same size as main stack

```
using System;
using System.Collections.Generic;

public class MinStack
{
    private Stack<int> stack;
    private Stack<int> minStack;

    public MinStack()
    {
        stack = new Stack<int>();
        minStack = new Stack<int>();
    }
    public void Push(int val)
    {
        stack.Push(val);

        // Push to minStack if it's empty or val is new minimum
        if (minStack.Count == 0 || val <= minStack.Peek())
        {
            minStack.Push(val);
        }
    }

    public void Pop()
    {
        if (stack.Count == 0) return;

        int popped = stack.Pop();

        // If popped element was minimum, remove from minStack too
        if (popped == minStack.Peek())
        {
            minStack.Pop();
        }
    }

    public int Top()
    {
        return stack.Peek();
    }

    public int GetMin()
    {
        return minStack.Peek();
    }
}
```

Approach 2: Single Stack with Pairs

Explanation: Store pairs of (value, current_minimum) in single stack.

Time Complexity: O(1) for all operations
Space Complexity: O(n) - storing pairs instead of single values

```
using System;
using System.Collections.Generic;

public class MinStack
{
    private Stack<(int val, int min)> stack;

    public MinStack()
    {
        stack = new Stack<(int, int)>();
    }
    public void Push(int val)
    {
        int currentMin = stack.Count == 0 ? val : Math.Min(val, stack.Peek().min);
        stack.Push((val, currentMin));
    }

    public void Pop()
    {
        if (stack.Count > 0)
        {
            stack.Pop();
        }
    }

    public int Top()
    {
        return stack.Peek().val;
    }

    public int GetMin()
    {
        return stack.Peek().min;
    }
}
```

Approach 3: Optimal - Auxiliary Stack with Space Optimization

Detailed Reasoning: Use auxiliary stack but only push when necessary. Push to min stack when element is less than or equal to current minimum, and track count of equal minimums to handle duplicates correctly.

Time Complexity: O(1) for all operations
Space Complexity: O(n) in worst case, but typically much better

```
using System;
using System.Collections.Generic;

/// <summary>
/// MinStack implementation using auxiliary stack with space optimization
/// All operations: O(1) time, O(n) space
/// </summary>
public class MinStack
{
    private Stack<int> dataStack;
    private Stack<int> minStack;

    public MinStack()
    {
        dataStack = new Stack<int>();
        minStack = new Stack<int>();
    }

    public void Push(int val)
    {
        // Always push to data stack
        dataStack.Push(val);

        // Push to min stack if it's empty or val is <= current minimum
        // Use <= (not <) to handle duplicate minimums correctly
        if (minStack.Count == 0 || val <= minStack.Peek())
        {
            minStack.Push(val);
        }
    }

    public void Pop()
    {
        if (dataStack.Count == 0) return;

        int poppedValue = dataStack.Pop();

        // If the popped value was the minimum, remove from min stack
        if (minStack.Count > 0 && poppedValue == minStack.Peek())
        {
            minStack.Pop();
        }
    }

    public int Top()
    {
        if (dataStack.Count == 0)
            throw new InvalidOperationException("Stack is empty");

        return dataStack.Peek();
    }

    public int GetMin()
    {
        if (minStack.Count == 0)
            throw new InvalidOperationException("Stack is empty");

        return minStack.Peek();
    }
}
```

Key Takeaways:

- **Auxiliary data structure**: Use second stack to maintain additional information
- **Handle duplicates**: Use `<=` comparison to properly track duplicate minimum values
- **Space-time trade-off**: O(n) space for O(1) time operations

PROBLEM 7: VALID PARENTHESES

Problem Statement:

Given a string `s` containing just the characters `'('`, `')'`, `'{'`, `'}'`, `'['` and `']'`, determine if the input string is valid. An input string is valid if: 1) Open brackets must be closed by the same type of brackets. 2) Open brackets must be closed in the correct order. 3) Every close bracket has a corresponding open bracket of the same type.[7]

Test Cases:

```
Example 1:
Input: s = "()"
Output: true
Explanation: Single pair of matching parentheses.

Example 2:
Input: s = "()[]{}"
Output: true
Explanation: Multiple types of brackets, all properly matched.

Example 3:
Input: s = "(]"
Output: false
Explanation: Wrong type of closing bracket for opening parenthesis.

Example 4:
Input: s = "([)]"
Output: false
Explanation: Brackets are interleaved incorrectly.

Example 5:
Input: s = "{[]}"
Output: true
Explanation: Nested brackets are properly matched.
```

Interview Questions & Answers:

1. **Q:** "Why is a stack the ideal data structure for this problem?"
   A: Stack's LIFO property matches the nested nature of valid parentheses - the most recently opened bracket should be the first to close.
2. **Q:** "How do you handle different types of brackets efficiently?"
   A: Use a HashMap to map closing brackets to their corresponding opening brackets for quick lookup.
3. **Q:** "What if the string contains other characters besides brackets?"
   A: The problem states only bracket characters, but in general, you'd skip non-bracket characters.
4. **Q:** "How do you detect mismatched bracket types?"
   A: When encountering a closing bracket, check if the top of stack has the corresponding

opening bracket.

5. **Q:** "What indicates an invalid string?"
   A: Three cases: 1) Closing bracket with no opening bracket, 2) Mismatched bracket types, 3) Unmatched opening brackets at the end.

Approach 1: Stack with Multiple If Statements

Idea: Use stack to track opening brackets, check each closing bracket against stack top.

Time Complexity: O(n) - single pass through string
Space Complexity: O(n) - worst case, all opening brackets

```
using System;
using System.Collections.Generic;

public class Solution
{
    public bool IsValid(string s)
    {
        var stack = new Stack<char>();

        foreach (char c in s)
        {
            // Push opening brackets onto stack
            if (c == '(' || c == '[' || c == '{')
            {
                stack.Push(c);
            }
            // Check closing brackets
            else if (c == ')' || c == ']' || c == '}')
            {
                if (stack.Count == 0) return false;

                char top = stack.Pop();

                if ((c == ')' && top != '(') ||
                    (c == ']' && top != '[') ||
                    (c == '}' && top != '{'))
                {
                    return false;
                }
            }
        }

        // Valid if no unmatched opening brackets remain
        return stack.Count == 0;
    }
}
```

Approach 2: Stack with HashMap Lookup

Explanation: Use HashMap to map closing brackets to opening brackets for cleaner code and easier extension.

Time Complexity: O(n) - single pass through string
Space Complexity: O(n) - stack plus small HashMap

```
using System;
using System.Collections.Generic;

public class Solution
{
    public bool IsValid(string s)
    {
        var stack = new Stack<char>();
        var map = new Dictionary<char, char>
        {
            { ')', '(' },
            { ']', '[' },
            { '}', '{' }
        };

        foreach (char c in s)
        {
            if (map.ContainsKey(c))
            {
                // Closing bracket: check if it matches top of stack
                if (stack.Count == 0 || stack.Pop() != map[c])
                {
                    return false;
                }
            }
            else
            {
                // Opening bracket: push onto stack
                stack.Push(c);
            }
        }

        return stack.Count == 0;
    }
}
```

Approach 3: Optimal - Stack with Direct Character Matching

Detailed Reasoning: Most efficient approach pushes corresponding closing bracket onto stack for each opening bracket. This eliminates HashMap lookup and simplifies matching logic.

Time Complexity: O(n) - single pass through string
Space Complexity: O(n) - stack storage only

```csharp
using System;
using System.Collections.Generic;

/// <summary>
/// Validates parentheses using stack with direct character matching
/// Time: O(n), Space: O(n)
/// </summary>
public class Solution
{
    public bool IsValid(string s)
    {
        var stack = new Stack<char>();

        foreach (char c in s)
        {
            // For opening brackets, push corresponding closing bracket
            switch (c)
            {
                case '(':
                    stack.Push(')');
                    break;
                case '[':
                    stack.Push(']');
                    break;
                case '{':
                    stack.Push('}');
                    break;
                default:
                    // For closing brackets, check if it matches expected
                    if (stack.Count == 0 || stack.Pop() != c)
                    {
                        return false;
                    }
                    break;
            }
        }

        // Valid if all brackets are matched (stack is empty)
        return stack.Count == 0;
    }
}
```

Key Takeaways:

- **Stack for nesting**: LIFO property naturally handles nested bracket structures
- **Early termination**: Return false immediately when mismatch is detected
- **Complete matching**: Valid string requires empty stack at the end

PROBLEM 8: EVALUATE REVERSE POLISH NOTATION

Problem Statement:

You are given an array of strings `tokens` that represents an arithmetic expression in a Reverse Polish Notation. Evaluate the expression. Return an integer that represents the value of the expression.[8]

Note that:

- The valid operators are `'+'`, `'-'`, `'*'`, and `'/'`.
- Each operand may be an integer or another expression.
- The division between two integers always truncates toward zero.
- There will not be any division by zero.
- The input represents a valid arithmetic expression in a reverse polish notation.

Test Cases:

```
Example 1:
Input: tokens = ["2","1","+","3","*"]
Output: 9
Explanation: ((2 + 1) * 3) = 9

Example 2:
Input: tokens = ["4","13","5","/","+"]
Output: 6
Explanation: (4 + (13 / 5)) = 6

Example 3:
Input: tokens = ["10","6","9","3","+","-11","*","/","*","17","+","5","+"]
Output: 22
Explanation:
((10 * (6 / ((9 + 3) * -11))) + 17) + 5
= ((10 * (6 / (12 * -11))) + 17) + 5
= ((10 * (6 / -132)) + 17) + 5
= ((10 * 0) + 17) + 5 = 22
```

Interview Questions & Answers:

1. **Q:** "How does Reverse Polish Notation work?"
   A: Operators come after operands. When you encounter an operator, apply it to the most recent operands (which are on top of the stack).
2. **Q:** "Why is stack perfect for this problem?"
   A: RPN naturally follows LIFO order - most recent operands are used first when an operator is encountered.
3. **Q:** "How do you handle operator precedence?"
   A: RPN eliminates precedence concerns - the notation itself dictates the order of operations.

4. **Q:** "What about division truncation toward zero?"
   A: In C#, integer division automatically truncates toward zero for positive results, but be careful with negative numbers.
5. **Q:** "How do you differentiate between numbers and operators?"
   A: Check if the token is one of the four operators (+, -, *, /). Everything else is treated as a number.

Approach 1: Stack with String Comparison

Idea: Use stack to store operands, when operator is encountered, pop two operands and apply operation.

Time Complexity: O(n) - single pass through tokens
Space Complexity: O(n) - stack storage for operands

```
using System;
using System.Collections.Generic;

public class Solution
{
    public int EvalRPN(string[] tokens)
    {
        var stack = new Stack<int>();

        foreach (string token in tokens)
        {
            if (token == "+" || token == "-" || token == "*" || token == "/")
            {
                // Pop two operands (order matters for - and /)
                int operand2 = stack.Pop();
                int operand1 = stack.Pop();
                int result = 0;

                if (token == "+") result = operand1 + operand2;
                else if (token == "-") result = operand1 - operand2;
                else if (token == "*") result = operand1 * operand2;
                else if (token == "/") result = operand1 / operand2;

                stack.Push(result);
            }
            else
            {
                // Token is a number, push onto stack
                stack.Push(int.Parse(token));
            }
        }

        return stack.Pop();
    }
}
```

Approach 2: Stack with Switch Statement

Explanation: Use switch statement for cleaner operator handling and better performance.

Time Complexity: O(n) - single pass through tokens
Space Complexity: O(n) - stack storage

```
using System;
using System.Collections.Generic;

public class Solution
{
    public int EvalRPN(string[] tokens)
    {
        var stack = new Stack<int>();

        foreach (string token in tokens)
        {
            switch (token)
            {
                case "+":
                    stack.Push(stack.Pop() + stack.Pop());
                    break;
                case "-":
                    int subtrahend = stack.Pop();
                    stack.Push(stack.Pop() - subtrahend);
                    break;
                case "*":
                    stack.Push(stack.Pop() * stack.Pop());
                    break;
                case "/":
                    int divisor = stack.Pop();
                    stack.Push(stack.Pop() / divisor);
                    break;
                default:
                    stack.Push(int.Parse(token));
                    break;
            }
        }

        return stack.Pop();
    }
}
```

Approach 3: Optimal - Stack with HashSet and Delegates

Detailed Reasoning: Use HashSet for O(1) operator checking and function delegates for operation dispatch. This approach is most extensible and has best performance characteristics.

Time Complexity: O(n) - single pass with O(1) operations
Space Complexity: O(n) - stack storage plus small constant for operators

```
using System;
using System.Collections.Generic;

/// <summary>
/// Evaluates Reverse Polish Notation using stack and function delegates
/// Time: O(n), Space: O(n)
/// </summary>
public class Solution
{
    public int EvalRPN(string[] tokens)
    {
        var stack = new Stack<int>();
        var operators = new HashSet<string> { "+", "-", "*", "/" };

        // Function to perform operations
        Func<int, int, string, int> operate = (a, b, op) => op switch
        {
            "+" => a + b,
            "-" => a - b,
            "*" => a * b,
            "/" => a / b,
            _ => throw new ArgumentException("Invalid operator")
        };

        foreach (string token in tokens)
        {
            if (operators.Contains(token))
            {
                // Pop operands in correct order (b is top of stack)
                int b = stack.Pop();
                int a = stack.Pop();

                // Perform operation and push result
                int result = operate(a, b, token);
                stack.Push(result);
            }
            else
            {
                // Parse number and push onto stack
                stack.Push(int.Parse(token));
            }
        }

        // Final result is the only element left in stack
        return stack.Pop();
    }
}
```

Key Takeaways:

- **RPN evaluation**: Stack naturally handles the postfix order of operations
- **Operand order**: When popping for binary operations, second pop is first operand
- **Single result**: Valid RPN expression leaves exactly one value on stack

DAY 3 SUMMARY

Summary of Concepts Covered

Linked List Manipulation:

- **Pointer management**: Tracking previous, current, and next nodes
- **Dummy node pattern**: Simplifying edge cases in list construction
- **Two-pointer techniques**: Fast/slow for middle finding, cycle detection
- **In-place operations**: Modifying list structure without extra space

Stack Applications:

- **LIFO processing**: Last-in-first-out for nested structures
- **Expression evaluation**: Parentheses matching, postfix notation
- **Auxiliary data structures**: Supporting O(1) operations with extra stack
- **Validation problems**: Checking balanced structures

Key Algorithms:

- **Floyd's Cycle Detection**: Two pointers for cycle finding
- **List Reversal**: Three-pointer iterative technique
- **Stack-based parsing**: Processing nested or sequential structures

Time & Space Complexity Summary

Problem Type	Brute Force	Optimized	Key Technique
Reverse Linked List	**O(n)** recursive	**O(1)** iterative	Three-pointer manipulation
Merge Sorted Lists	**O(n)** recursive	**O(1)** iterative	Dummy node + two pointers
Reorder List	**O(n)** space	**O(1)** space	Find middle + reverse + merge
Remove Nth from End	**O(n)** two-pass	**O(n)** one-pass	Two pointers with gap
Cycle Detection	**O(n)** space HashSet	**O(1)** space	Floyd's fast/slow pointers
Min Stack	**O(1)** all ops	**O(1)** all ops	Auxiliary stack
Valid Parentheses	**O(n)** time/space	**O(n)** time/space	Stack for LIFO matching
RPN Evaluation	**O(n)** time/space	**O(n)** time/space	Stack for operand storage

Pattern Recognition Guide

Use Linked List Techniques When:

- Modifying list structure (reversal, reordering)
- Finding specific positions (middle, nth from end)
- Detecting cycles or loops
- Merging or splitting lists

Use Stack When:

- Processing nested structures (parentheses, expressions)
- Need LIFO access pattern
- Reversing order of operations
- Maintaining auxiliary information (like minimum values)

Use Two Pointers When:

- Finding middle of linked list
- Detecting cycles
- Maintaining relative positions
- One-pass algorithms for list problems

Common Patterns and Tricks

Dummy Node Benefits:

- Eliminates special handling for head node
- Simplifies list construction code
- Consistent handling of empty lists
- Easier pointer management

Floyd's Algorithm Applications:

- Cycle detection in any sequence
- Finding middle element
- Detecting meeting points
- Loop-related problems

Stack Design Patterns:

- Auxiliary stack for O(1) queries
- Stack for expression parsing
- LIFO for nested structure validation
- Stack for undo/redo operations

Review Tips

- **Practice pointer visualization**: Draw linked list diagrams while coding
- **Master dummy node pattern**: Use consistently for list construction problems
- **Understand stack applications**: Recognize when LIFO property is beneficial
- **Edge case handling**: Empty lists, single nodes, cycles, stack overflow

How to Practice Mock Interviews

1. **Problem Classification** (3 min): Identify if it's linked list manipulation or stack application
2. **Approach Discussion** (7 min): Explain brute force, then optimal approach with complexity
3. **Implementation** (25 min): Code while explaining pointer movements and stack operations
4. **Testing** (10 min): Walk through examples, especially edge cases like empty structures

Common Mistakes to Avoid

- **Null pointer exceptions**: Always check `node != null` before accessing `node.next`
- **Lost references**: Store `next` pointer before modifying `current.next`
- **Infinite loops**: Ensure pointers always make progress in cycles
- **Stack underflow**: Check stack emptiness before `pop()` operations
- **Operator precedence**: Remember RPN eliminates precedence concerns
- **Integer overflow**: Consider overflow in arithmetic operations

Advanced Topics for Further Study

- **Linked List Cycle II**: Finding where cycle starts
- **Copy List with Random Pointer**: Handling multiple pointer types
- **LRU Cache**: Combining linked list with hash map
- **Expression Tree**: Converting infix to tree structure
- **Monotonic Stack**: For next greater/smaller element problems
- **Doubly Linked List**: Bidirectional traversal capabilities

Real-world Applications

- **Memory Management**: Linked lists in malloc/free implementations
- **Undo/Redo Systems**: Stack-based operation history
- **Function Call Stack**: How programming languages handle recursion
- **Browser History**: Stack-like navigation with back/forward
- **Expression Compilers**: Stack-based parsing and evaluation
- **Database B+ Trees**: Linked structures for efficient data access

This completes Day 3: Linked Lists & Stacks with comprehensive coverage of all 8 problems, following the same detailed structure and educational approach as previous days.

⁂

1. https://leetcode.com/problems/reverse-linked-list/
2. https://leetcode.com/problems/merge-two-sorted-lists/
3. https://github.com/doocs/leetcode/blob/main/solution/0100-0199/0143.Reorder List/README_EN.md
4. https://leetcode.com/problems/remove-nth-node-from-end-of-list/
5. https://leetcode.com/problems/linked-list-cycle/
6. https://leetcode.com/problems/min-stack/
7. https://leetcode.com/problems/valid-parentheses/
8. https://leetcode.com/problems/evaluate-reverse-polish-notation/