Mastering Linked List Reversal Patterns for Coding Interviews

Welcome to your daily dose of coding interview prep! Today, we'll dive into a powerful and popular topic: **Linked List Reversal**. This concept isn't just a one-trick pony. Mastering it unlocks a whole class of problems and helps you level up your understanding of pointer manipulation—a skill that is essential for many coding interviews.

Linked lists are everywhere in interviews because they test your ability to work with pointers, understand edge cases, and handle in-place modifications. Reversing a linked list is one of the most fundamental operations, but it comes in a variety of flavors: reverse the whole list, reverse a part of it, reverse in groups, and so on.

In this article, we'll explore three related problems, each a twist on the basic reversal pattern:

- **Reverse Linked List**: Flip the whole list.
- **Reverse Linked List II**: Flip only a subpart, between two positions.
- **Reverse Nodes in k-Group**: Flip every group of `k` nodes.

By the end, you'll see how a core technique adapts to different scenarios—and you'll be ready for whatever linked list twist an interviewer throws at you!

Topic Introduction: Linked List Reversal

What is Linked List Reversal?

Linked list reversal is the process of changing the direction of all the pointers in a singly linked list so that the head becomes the tail and vice versa. This operation is a classic test of pointer manipulation skills.

How does it work?

Imagine you have a list: `A -> B -> C -> D`. To reverse it, you need to make the links go backward: `D -> C -> B -> A`. The core idea is to walk through the list, and for each node, point its `next` to the previous node.

When to use it?

- When asked to reverse the entire list.
- To reverse only a portion of a list (between two nodes).
- To reverse nodes in groups of k.
- To solve problems where you need to process a list from tail to head.

Simple Example:

Suppose you have `1 -> 2 -> 3 -> 4 -> 5`, and you want to reverse the whole thing. The result should be `5 -> 4 -> 3 -> 2 -> 1`.

Let's start simple and build up!

Problem 1: Reverse Linked List

LeetCode #206

Problem Statement (in plain English):

Given the head of a singly linked list, reverse the list and return the new head.

Example:

- Input: `1 -> 2 -> 3 -> 4 -> 5`
- Output: `5 -> 4 -> 3 -> 2 -> 1`

Let's break this down:

- We need to reverse the pointers so each node points to its predecessor instead of its successor.
- The process ends when we reach the end (`None`), which becomes the new head.

Thought Process:

At each step, keep track of three things:

- The current node.
- The previous node (to which you'll point the `next`).
- The next node (so you don't lose the rest of the list).

Let's walk through the above example with pen and paper!

Try this variant: What happens if your list is just a single node? Or empty?

- Input: `[]` (empty list) — Output: `[]`
- Input: `1` — Output: `1`

Take a moment to try solving this on your own before reading the solution.

Solution: Iterative Approach

We'll use the iterative approach because it's the most direct way to see pointer manipulation in action (and sets the foundation for the next problems).

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def reverseList(head):
    prev = None
    curr = head

    while curr:
        next_temp = curr.next    # Save next node
        curr.next = prev         # Reverse the link
        prev = curr              # Move prev forward
```

```
        curr = next_temp         # Move curr forward

    return prev  # prev is the new head at the end
```

Time Complexity: O(n) — Each node is visited once.

Space Complexity: O(1) — We only use a few extra pointers.

Code Explanation:

- We initialize `prev` to None, and `curr` to the head.
- At each iteration, we:
  - Save `curr.next` (so we don't lose the rest of the list).
  - Point `curr.next` backward to `prev`.
  - Move `prev` and `curr` forward.
- When `curr` becomes None, `prev` is the new head.

Let's walk through a quick test case:

Suppose `head = 1 -> 2 -> 3`.

- Before first iteration: `prev=None`, `curr=1`
- After first iteration: `curr.next` points to `None` (prev), `prev=1`, `curr=2`
- After second iteration: `curr.next` points to `1`, `prev=2`, `curr=3`
- After third iteration: `curr.next` points to `2`, `prev=3`, `curr=None`

Try this input on paper: `1 -> 2`

What will the reversed list be? *(Answer: `2 -> 1`)*

Did you know?

You can also solve this recursively! After finishing this article, try writing a recursive solution.

Now, let's see how this basic pattern is adapted when we are asked to reverse only part of a list.

Problem 2: Reverse Linked List II

LeetCode #92

Problem Statement (rephrased):

Given the head of a singly linked list and two positions, `left` and `right`, reverse the nodes of the list from position `left` to `right`, and return the modified list.

- Only the nodes between `left` and `right` (inclusive) should be reversed.
- The rest of the list should remain as is.

Example:

- Input: `1 -> 2 -> 3 -> 4 -> 5`, `left = 2`, `right = 4`
- Output: `1 -> 4 -> 3 -> 2 -> 5`

How is this different from Problem 1?

- Instead of reversing the whole list, we only reverse a sublist.
- We need to carefully reconnect the reversed sublist back into the main list.

Let's walk through the example:

- The list before reversal: `1 -> 2 -> 3 -> 4 -> 5`
- We reverse nodes at positions 2, 3, 4: `2 -> 3 -> 4` becomes `4 -> 3 -> 2`
- After reversal: `1 -> 4 -> 3 -> 2 -> 5`

Try this input:

- Input: `1 -> 2 -> 3 -> 4 -> 5`, `left = 3`, `right = 5`
- What should the output be? *(Answer: `1 -> 2 -> 5 -> 4 -> 3`)*

Take a moment to try solving this on your own before reading the solution.

Solution: Iterative, In-Place Reversal

We can use the same logic as in Problem 1, but only for the sublist between `left` and `right`. We'll need to:

- Find the node just before `left` position (`pre`).
- Reverse the part between `left` and `right`.
- Reconnect the reversed sublist.

Here's how we can do it:

```
def reverseBetween(head, left, right):
    if not head or left == right:
        return head

    dummy = ListNode(0)
    dummy.next = head
    pre = dummy

    # Step 1: Move pre to the node before the reversal starts
    for _ in range(left - 1):
        pre = pre.next

    # Step 2: Reverse the sublist
    curr = pre.next
    prev = None
    for _ in range(right - left + 1):
        next_temp = curr.next
        curr.next = prev
        prev = curr
        curr = next_temp
```

```
    # Step 3: Connect reversed sublist back to the main list
    pre.next.next = curr
    pre.next = prev

    return dummy.next
```

Time Complexity: O(n) — We traverse the list once.
Space Complexity: O(1) — Only pointer manipulation.

Code Explanation:

- We use a dummy node to simplify edge cases (like reversing from the head).
- We move `pre` to the node just before the sublist to reverse.
- We reverse the sublist just as in Problem 1, but only for `right - left + 1` steps.
- `pre.next` is the first node in the sublist (which, after reversal, will become the tail), so we reattach its `next` to the node after the reversed sublist (`curr`).
- Finally, we reattach `pre.next` to the new head of the reversed sublist (`prev`).

Let's see a test case in action:

Input: `1 -> 2 -> 3 -> 4 -> 5`, left=2, right=4

- After step 1: `pre` points to `1`
- Step 2 reverses `2 -> 3 -> 4` to `4 -> 3 -> 2`
- Step 3 reconnects: `1 -> 4 -> 3 -> 2 -> 5`

Try this one:
Input: `3 -> 5`, left=1, right=2
What is the output? *(Answer: `5 -> 3`)*

Did you realize?
You can also use recursion to reverse a sublist, though it gets a bit trickier. Give it a shot after mastering the iterative approach!

Now that you're comfortable reversing arbitrary portions of a list, let's kick it up a notch: what if you need to reverse multiple groups, each of size `k`?

Problem 3: Reverse Nodes in k-Group

LeetCode #25

Problem Statement (in your own words):

Given a singly linked list, reverse the nodes in every group of `k` and return the modified list. If the number of nodes is not a multiple of `k`, leave the last group as is.

- Reverse the first `k` nodes, then the next `k`, etc.

Example:

- Input: `1 -> 2 -> 3 -> 4 -> 5`, `k = 3`
- Output: `3 -> 2 -> 1 -> 4 -> 5`

How is this different from the previous problems?

- Instead of reversing a single sublist, you reverse *every* group of `k` nodes.
- Groups of fewer than `k` nodes at the end stay as is.
- You need to repeatedly identify sublists of size `k` and apply reversal.

Let's break down the example:

- First group: `1 -> 2 -> 3` reversed to `3 -> 2 -> 1`
- Remaining: `4 -> 5` (less than `k=3`, so left as is)
- Final result: `3 -> 2 -> 1 -> 4 -> 5`

Try this example on paper:

Input: `1 -> 2 -> 3 -> 4 -> 5 -> 6`, k=2

What should the output be? *(Answer: `2 -> 1 -> 4 -> 3 -> 6 -> 5`)*

Take a moment to try solving this on your own before reading the solution.

Solution: Iterative, Group-wise Reversal

This problem builds upon the sublist reversal pattern from Problem 2. For each group of size `k`, we:

- Check if there are at least `k` nodes left.
- Reverse the next `k` nodes.
- Connect the reversed group back into the list.

Here's how you can do it:

```
def reverseKGroup(head, k):
    dummy = ListNode(0)
    dummy.next = head
    group_prev = dummy

    while True:
        kth = group_prev
        # Step 1: Find the kth node after group_prev
        for _ in range(k):
            kth = kth.next
            if not kth:
                return dummy.next

        group_next = kth.next
        # Step 2: Reverse k nodes
```

```
        prev, curr = kth.next, group_prev.next
        for _ in range(k):
            temp = curr.next
            curr.next = prev
            prev = curr
            curr = temp
        # Step 3: Reconnect reversed group
        temp = group_prev.next
        group_prev.next = prev
        group_prev = temp

    return dummy.next
```

Time Complexity: O(n) — Each node is visited once.

Space Complexity: O(1) — We only use pointers.

Code Explanation:

- `dummy` is a helper node to manage edge cases.
- `group_prev` points to the node before the current group.
- We look ahead `k` nodes to ensure a full group is available.
- We reverse the group (using the familiar pattern from Problems 1 and 2).
- After reversal, we reconnect the group and move `group_prev` forward.

Let's step through a test case:

Input: `1 -> 2 -> 3 -> 4 -> 5`, k=2

- First group: `1 -> 2` reversed to `2 -> 1`
- Second group: `3 -> 4` reversed to `4 -> 3`
- Remaining: `5` (less than k=2, not reversed)
- Result: `2 -> 1 -> 4 -> 3 -> 5`

Try this yourself:

Input: `1 -> 2 -> 3 -> 4 -> 5`, k=1

What will the output be? *(Answer: `1 -> 2 -> 3 -> 4 -> 5` — no change)*

Did you guess?

This problem can also be solved recursively! Once you're comfortable with the iterative version, try implementing the recursion.

Summary and Next Steps

Congratulations! You've just journeyed through three classic linked list reversal problems:

- **Reverse Linked List:** The backbone of pointer manipulation.
- **Reverse Linked List II:** Focused reversal between two positions, requiring careful reconnection.

• **Reverse Nodes in k-Group:** Generalizes the pattern to many sublists, practicing repetition of pointer tricks.

Key takeaways:

• All three problems use the same core reversal logic, just applied to different portions of the list.
• Dummy nodes make handling edge cases (like reversing from the head) much easier.
• Carefully reconnecting reversed sections is essential—missing a pointer update is a common source of bugs.
• For problems with partial or repeated reversal, isolating the sublist and using the reversal pattern from Problem 1 is a solid approach.

Common mistakes to watch for:

• Forgetting to reconnect the reversed sublist to the rest of the list.
• Losing track of where the current, previous, and next pointers should be after each reversal.
• Not handling edge cases (e.g., reversal at head, k greater than list length, single-node lists).

Action List for You:

• **Solve all three problems yourself**—without looking at the solutions!
• **Try the recursive versions** for each, once you're comfortable with iteration.
• **Explore other linked list reversal variations**, like palindrome checking or odd-even reordering.
• **Read other people's solutions** to see creative pointer handling and edge case management.
• **Practice drawing pointer diagrams** for each operation, especially when debugging.
• **Don't get frustrated by mistakes**—linked list manipulation takes practice. Every bug is a learning opportunity!

Remember, building intuition with pointers takes time, and these problems are a rite of passage in the world of coding interviews. Keep practicing, stay curious, and you'll soon be reversing lists like a pro!