Linked List Deletion Episode- Kill them all!

Welcome to your daily PrepLetter! Today, we're diving into a classic interview topic: performing deletions in linked lists. If you've ever been asked about removing "nodes" from a linked list, congratulations — you've touched one of the most popular coding interview patterns.

Deleting nodes in a linked list isn't just about erasing data; it's about manipulating pointers, understanding edge cases, and sometimes using clever techniques like two pointers or in-place updates. These skills are essential for linked list mastery and will power you through a big chunk of tech interviews.

In this article, we'll explore three linked list deletion problems, each teaching you a crucial variant of this pattern. We'll build from the simplest to the most nuanced — so by the end, you'll confidently handle linked list deletions of any flavor.

Ready? Let's get started!

Topic Introduction

What's the big deal about deleting in linked lists?

Unlike arrays, linked lists store data in nodes connected by pointers. Removing a node means you need to carefully update these pointers so you don't accidentally lose access to the rest of the list. The tricky part? You often can't go backward, and you might not have a direct reference to the node before the one you want to delete.

When do you use these techniques?

- Whenever you need to remove elements from a linked list (single or multiple, by value or position).
- When you're asked to do the operation in-place, with constant extra space.
- When time and efficiency matter (as they always do in interviews!).

Simple Example:
Suppose you have a list: `1 -> 2 -> 3 -> 4`, and you want to remove `3`.
You need to make `2.next` point to `4`, effectively skipping `3`.

Now, let's see how this plays out in three real-world problems.

Problem 1: Remove Duplicates from Sorted List

Link: LeetCode 83 - Remove Duplicates from Sorted List

Problem Statement (In Simple Words)

Given a sorted linked list, remove all duplicate elements such that each element appears only once. Return the modified list.

Example:
Input: `1 -> 1 -> 2 -> 3 -> 3`

Output: `1 -> 2 -> 3`

Step-by-Step Walkthrough

Because the list is sorted, all duplicates are next to each other. This makes it a great starting point: we can solve it with a single pass and simple pointer adjustment.

Let's walk through the example:

- Start at `1`. Next node is also `1` (duplicate), so we skip it by pointing the current node's `next` to the next-next node.
- Now at `1` (which points to `2`). Move to `2`.
- Next is `3`, then another `3` (duplicate). Skip the duplicate by updating pointers.

Try this example with pen and paper:
Input: `1 -> 1 -> 1 -> 2 -> 3 -> 3 -> 4 -> 4`
Expected Output: `1 -> 2 -> 3 -> 4`

Challenge Yourself

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

Solution Breakdown

Since the list is sorted and we only need to remove consecutive duplicates, a single pointer (let's call it `current`) is enough:

- Start at the head.
- While `current` and `current.next` exist:
  - If `current.val == current.next.val`, skip the next node.
  - Else, move `current` forward.

Here's the Python code:

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

def deleteDuplicates(head):
    current = head
    while current and current.next:
        if current.val == current.next.val:
            # Skip the duplicate node
            current.next = current.next.next
        else:
            # Move to the next distinct element
```

```
            current = current.next
    return head
```

Time and Space Complexity

- **Time:** O(N), where N is the number of nodes. We visit each node once.
- **Space:** O(1), since we do this in-place.

Code Explanation

- We use a single pointer, `current`, to traverse the list.
- When we find two nodes with the same value, we skip the duplicate by changing the `next` pointer.
- We only move `current` forward when there's no duplicate.

Try another example:
Input: `1 -> 2 -> 2 -> 2 -> 3 -> 4 -> 4 -> 5`
Can you walk through the code and see how it removes duplicates?

Problem 2: Delete Node in a Linked List

Link: LeetCode 237 - Delete Node in a Linked List

Problem Statement (In Simple Words)

You're given a reference to a node in the middle (not the first or last) of a singly linked list. Delete this node from the list — but you don't have access to the head or previous node!

Example:
Input: List: `4 -> 5 -> 1 -> 9`, node to delete: `5`
Output: `4 -> 1 -> 9` (since `5` is removed)

What's Different Here?

Unlike the previous problem, you can't traverse from the head or know the previous node. Instead, you have to work with only the given node. This forces us to get creative!

Try this with pen and paper:
List: `7 -> 8 -> 10 -> 12`, node to delete: `10`
Expected Output: `7 -> 8 -> 12`

Challenge Yourself

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

Solution Breakdown

Since we don't have access to the previous node, we can't just change the previous node's `next` pointer. Instead, we copy the value from the next node into the current node, then delete the next node.

- Copy the value of the next node into the current node.
- Change the current node's `next` to point to the node after next.
- The next node is now skipped — effectively "deleted".

Here's the Python code:

```
def deleteNode(node):
    # Copy value from next node
    node.val = node.next.val
    # Remove next node by skipping it
    node.next = node.next.next
```

Time and Space Complexity

- **Time:** O(1), because we do a constant number of operations.
- **Space:** O(1), no extra space used.

Code Explanation

- The key trick is to overwrite the current node's value with the next node's value.
- Then, we skip over the next node by changing the pointer.
- This works only if the node to delete is **not the last node**.

Try another test case on your own:
List: `6 -> 7 -> 8 -> 9 -> 10`, node to delete: `9`
Expected Output: `6 -> 7 -> 8 -> 10`

Did you notice?
Compared to Problem 1, here we do not traverse the list or use two pointers — just local manipulation. This is a unique pattern for this specific kind of problem!

Problem 3: Remove Nth Node From End of List

Link: LeetCode 19 - Remove Nth Node From End of List

Problem Statement (In Simple Words)

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

Example:

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

Output: `1 -> 2 -> 3 -> 5` (removed the 4th node, which is 2nd from end)

How Is This Different?

Here, you need to delete a node, but you don't know its value or have a direct reference — you only know its position from the end. This means you need to find the right node first, then delete it.

This is a classic scenario for the **two-pointer technique** (sometimes called "fast and slow pointers"). You've seen pointer manipulation in the previous problems, but this time, you'll use two pointers to handle the "from the end" challenge.

Try this with pen and paper:

Input: `1 -> 2 -> 3 -> 4`, n = 1

Expected Output: `1 -> 2 -> 3`

Challenge Yourself

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

Solution Breakdown

To delete the nth node from the end, you need to:

- Set up two pointers, `fast` and `slow`, both starting at a dummy node before the head.
- Move `fast` n+1 steps ahead (so there's a gap of n between `fast` and `slow`).
- Move both pointers forward until `fast` reaches the end. Now, `slow` is right before the node to delete.
- Skip the node by updating `slow.next`.

Here's the Python code:

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

def removeNthFromEnd(head, n):
    dummy = ListNode(0)
    dummy.next = head
    fast = slow = dummy
    # Move fast n+1 steps ahead to maintain the gap
    for _ in range(n + 1):
        fast = fast.next
```

```
    # Move both pointers until fast reaches the end
    while fast:
        fast = fast.next
        slow = slow.next
    # Skip the nth node from end
    slow.next = slow.next.next
    return dummy.next
```

Time and Space Complexity

- **Time:** O(N), because we traverse the list at most twice.
- **Space:** O(1), constant extra space.

Code Explanation

- **Dummy node** handles edge cases like deleting the first node.
- **Two pointers**: `fast` is advanced so that the gap between `fast` and `slow` is `n`.
- When `fast` hits the end, `slow` is just before the node we want to delete.
- We skip the node by changing `slow.next`.

Dry-run example:

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

- After moving `fast` 3 steps: `fast` at node 3, `slow` at dummy.
- Move both until `fast` is None:
  - `fast` at 4, `slow` at 1
  - `fast` at 5, `slow` at 2
  - `fast` at None, `slow` at 3
- `slow.next` is 4 (to be removed). Skip it: `3.next = 5`.

Your turn:

Try input: `1 -> 2`, n = 2

What happens? (Hint: This is a good test for deleting the head.)

Did you know?

This problem can also be solved in two passes: first, find the length; second, find and delete the node. But the two-pointer approach is more elegant and efficient. Give the two-pass method a try after you finish this article!

Summary and Next Steps

Why group these problems?

All three are about deleting nodes from a singly linked list — but each highlights a different twist:

- Removing consecutive duplicates (easy pointer update).
- Deleting a node with only a reference to that node (in-place overwrite).

- Removing the nth node from the end (two-pointer technique).

Key Patterns and Variations:

- **Pointer manipulation:** Update `next` pointers carefully to avoid losing nodes.
- **Dummy node usage:** Simplifies edge cases, especially when deleting the head.
- **Two-pointer technique:** Essential for "from the end" operations.
- **In-place operations:** Sometimes you must work with very limited references.

Common Mistakes:

- Forgetting to handle edge cases (like deleting the head).
- Losing reference to parts of the list (accidentally breaking the chain).
- Not updating all pointers when removing duplicates.
- Trying to delete the last node with only a reference to it (not possible in Problem 2).

Action List for You

- **Solve all 3 problems yourself** (LeetCode links above).
- **Try the extra test cases** given in each problem section.
- **Explore similar problems:** Look for "Reverse Linked List", "Partition List", and "Merge Two Sorted Lists".
- **Review other solutions:** See how others handle edge cases or use different techniques (like recursion for Problem 1).
- **Practice dry-running your code:** Use pen and paper to track pointer updates.
- **Don’t stress over tough problems:** If you get stuck, come back later. Each attempt helps you learn!

Keep practicing, and remember: every pointer you move is one step closer to acing that interview. See you tomorrow for the next PrepLetter!