Lecture: Remove N-th Node from End of Linked List — From Zero to Advanced

1. Introduction

In this problem, you are given a **singly linked list** and a number n.
Your goal is to **remove the N-th node from the end** of the list in **one pass** (efficiently).

This is a **two-pointer technique** problem — very popular in interviews at companies like **Google, Microsoft, and Amazon**.

2. Real-Life Analogy

Imagine a **line of people** waiting at a counter.
You're asked to remove the **3rd person from the end**.

Instead of counting all people each time, you use **two markers** —

- one moves ahead by 3 steps,
- then both move together until the first marker reaches the end.
  The second marker now stands right **before the 3rd from the end** — ready to remove that person.

That's exactly how the **two-pointer approach** works.

3. Problem Statement

Given the head of a linked list and an integer n, remove the N-th node from the end of the list and return its head.

4. Example

Input:

head = [10 -> 20 -> 30 -> 40 -> 50]

n = 2

Output:

10 -> 20 -> 30 -> 50

✅ The 2nd node from the end (40) is removed.

5. Key Concepts

Concept	Explanation
Data Structure	Singly Linked List
Technique	Two Pointer / Fast & Slow
Goal	Remove N-th node from end
Constraints	Must handle head deletion too

6. Naive Approach

- Traverse list once to count total nodes length.
- Then traverse again to remove (length - n)th node.

Drawback: Requires **two passes (O(2n))**.

7. Optimized Approach — Two Pointers (One Pass)

Logic

1. Use two pointers: fast and slow.
2. Move fast ahead by n nodes.
3. Then move both pointers together until fast.next == None.
4. slow will now be **just before** the node to delete.
5. Adjust link:
   slow.next = slow.next.next

8. Visual Explanation

For list:

10 -> 20 -> 30 -> 40 -> 50

n = 2

Step 1: Move fast 2 steps ahead

fast = 30, slow = 10

Step 2: Move both until fast.next is None

fast = 50, slow = 30

Step 3: Remove slow.next (which is 40)

Result: 10 -> 20 -> 30 -> 50

9. Pseudocode

```
def remove_nth_from_end(head, n):
    dummy = Node(0)
    dummy.next = head
    slow = fast = dummy

    # Move fast ahead by n steps
    for _ in range(n):
        fast = fast.next

    # Move both until fast reaches end
    while fast.next:
        slow = slow.next
        fast = fast.next

    # Remove the target node
    slow.next = slow.next.next

    return dummy.next
```

10. Example Dry Run

Input:
head = 10 -> 20 -> 30 -> 40 -> 50, n = 2

Step	fast	slow	Operation
1	Move fast 2 ahead → 30	0(dummy)	Setup done
2	fast = 50	slow = 30	Move both together
3	fast.next = None	slow.next = 40	Remove 40

✅ **Output:** 10 -> 20 -> 30 -> 50

11. Handling Edge Cases

Case	Input	Output	Description
Delete head	[10, 20, 30], n=3	[20, 30]	Head removed
Single node	[10], n=1	[]	Empty list
n = 1	[1, 2, 3], n=1	[1, 2]	Remove last node
Empty list	[]	[]	No operation

✅ Using a **dummy node** handles all these cases cleanly.

12. Complete Code Example

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None


def print_list(head):
    temp = head
    while temp:
        print(temp.data, end=" -> ")
        temp = temp.next
```

```
print("None")


def remove_nth_from_end(head, n):
    dummy = Node(0)
    dummy.next = head
    slow = fast = dummy

    # Move fast ahead by n nodes
    for _ in range(n):
        fast = fast.next

    # Move both until fast reaches end
    while fast.next:
        fast = fast.next
        slow = slow.next

    # Remove target node
    slow.next = slow.next.next

    return dummy.next
```

13. Example Usage

```
# Create linked list: 10 -> 20 -> 30 -> 40 -> 50
head = Node(10)
head.next = Node(20)
head.next.next = Node(30)
head.next.next.next = Node(40)
```

```
head.next.next.next.next = Node(50)


print("Original List:")
print_list(head)


head = remove_nth_from_end(head, 2)


print("After Removing 2nd Node from End:")
print_list(head)
```

Output

```
Original List:
10 -> 20 -> 30 -> 40 -> 50 -> None
After Removing 2nd Node from End:
10 -> 20 -> 30 -> 50 -> None
```

14. Complexity Analysis

Operation	Time	Space
Remove Nth Node (One Pass)	O(n)	O(1)
Two-Pass Approach	O(2n)	O(1)

✅ The one-pass two-pointer method is the most efficient and optimal.

15. Variations

1. **Remove K-th node from Start**
2. **Delete all nodes with given value**
3. **Find N-th node from End (without deleting)**
4. **Delete middle node**

16. When to Use

- Problems where you need to **find a node relative to the end**
- When minimizing **traversal time** is important
- When implementing **linked list editors or task queues**

17. Real-World Applications

- **Task scheduling:** remove N-th most recent job
- **Version control systems:** delete last N commits
- **Music playlists:** remove N-th last played song

18. Summary Table

Step	Description
1	Create dummy node
2	Move fast ahead by n
3	Move both until fast reaches end
4	Skip target node
5	Return new head

19. Practice Tasks

1. Delete 1st node from end.
2. Delete last node from list.
3. Delete N-th node from end in a single-element list.
4. Modify to return **deleted node's value**.
5. Try deleting N-th node using **stack** instead of two pointers.

20. Closing Thought

The **Remove N-th Node from End** problem is a brilliant exercise in the **two-pointer technique**.
It builds strong logic for problems like **cycle detection, middle node finding, and k-group reversals**.