Day 23 (07/07/2025)

---

1. Kth from End of Linked List

A problem that introduces advanced linked list traversal techniques and teaches how to find elements from the end efficiently using the two-pointer approach.

Given the **head of a linked list** and the number **k**, your task is to find the **kth node from the end**. If k is more than the number of nodes, then the output should be **-1**. This problem appears frequently in **interviews** and **real-world applications** like implementing undo operations, recent history tracking, or buffer management systems. You can solve this by first counting nodes and then traversing again, but try to think of **more efficient approaches** using the two-pointer technique that requires only one pass.

This teaches **two-pointer traversal** and **reverse indexing techniques** that are essential for **efficient linked list processing and single-pass algorithms**.

Your task: Find the kth node from the end of a linked list using efficient single-pass traversal techniques.

Examples

Input:

LinkedList: 1->2->3->4->5->6->7->8->9, k = 2

Output:

8

Input:

LinkedList: 10->5->100->5, k = 5

Output:

-1

---

2. Remove Duplicates from a Sorted Linked List

A problem that demonstrates linked list modification techniques and teaches how to remove duplicate elements while maintaining list integrity.

Given a **singly linked list** that is already **sorted**, the task is to remove **duplicates** (nodes with duplicate values) from the given list if they exist. This operation is fundamental in **data cleaning** and **deduplication** applications where you need to **eliminate redundant entries** while preserving the original order. The challenge involves understanding how to properly link nodes while skipping duplicates and managing memory efficiently.

This introduces **linked list modification** and **duplicate removal techniques** that are crucial for **data preprocessing and memory-efficient list operations**.

Your task: Remove duplicate nodes from a sorted linked list while maintaining proper node connections.

Examples

Input:

LinkedList: 2->2->4->5

Output:

2 -> 4 -> 5

Input:

LinkedList: 2->2->2->2->2

Output:

2

---

3. Delete Middle of Linked List

A problem that teaches linked list deletion techniques and demonstrates how to remove specific nodes while maintaining list structure.

Given a **singly linked list**, delete the **middle of the linked list**. If there are **even nodes**, then there would be two middle nodes, and we need to delete the **second middle element**. If the input linked list has a **single node**, then it should return **NULL**. This problem is commonly used in **memory management** and **list optimization** where you need to remove elements based on position while maintaining list integrity.

This teaches **node deletion techniques** and **position-based removal** that are essential for **dynamic list management and memory optimization**.

Your task: Delete the middle node of a linked list while properly handling edge cases and maintaining list connections.

Examples

Input:

LinkedList: 1->2->3->4->5

Output:

1->2->4->5

Input:

LinkedList: 2->4->6->7->5->1

Output:

2->4->6->5->1