Cycle Detection in LL

Give Hint to solve this question. Given the head of a linked list, return the node where the cycle begins. If there is no cycle, return null.

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. Internally, pos is used to denote the index of the node that tail's next pointer is connected to (0-indexed). It is -1 if there is no cycle. Note that pos is not passed as a parameter.

Do not modify the linked list.

To solve this problem, you can use **Floyd's Cycle Detection Algorithm** (also known as the **Tortoise and Hare Algorithm**). Here's a structured hint to approach the problem:

1. **Detect if there is a cycle**:
   - Use two pointers, `slow` and `fast`.
   - Initialize both pointers to the head of the linked list.
   - Move `slow` by one step and `fast` by two steps at a time.
   - If `slow` and `fast` meet, a cycle exists. Otherwise, if `fast` or `fast.next` becomes `null`, there is no cycle.
2. **Find the starting node of the cycle**:
   - When `slow` and `fast` meet, reset one of the pointers (say `slow`) to the head of the linked list.
   - Move both pointers one step at a time.
   - The point where they meet again is the start of the cycle.

This works because the distance between the meeting point and the start of the cycle is equal to the distance between the head and the start of the cycle.

3. **Return the result**:
   - If the cycle is detected, return the node where the two pointers meet after the second phase.
   - If no cycle is detected in step 1, return `null`.

Would you like a step-by-step example to clarify further?

> This works because the distance between the meeting point and the start of the cycle is equal to the distance between the head and the start of the cycle. Explain why it works

This behavior is due to the mathematical properties of a linked list with a cycle and can be explained with the following steps:

Key Definitions

- Let the **head** of the linked list be A.
- Let the **start of the cycle** be C.
- Let the **meeting point** of the slow and fast pointers within the cycle be M.
- Define:
  - $d_{\text{head to cycle}}$: Distance from A to C.
  - d_{cycle}: Length of the cycle.
  - $d_{\text{meeting in cycle}}$: Distance from C to M.

Step-by-Step Analysis

1. **Distances Traveled by Slow and Fast Pointers**:
   - The slow pointer moves 1 step at a time.
   - The fast pointer moves 2 steps at a time.
   - If the slow and fast pointers meet at M, then the fast pointer must have traversed the cycle multiple times more than the slow pointer.

   Mathematically:

   $$\text{Distance traveled by fast pointer} = 2 \times \text{Distance traveled by slow pointer}$$

   Let the distance traveled by the slow pointer be X (from A to M), then:

   $$2X = X + k \cdot d_{\text{cycle}}$$

   where k is the number of complete cycles traversed by the fast pointer.

   Simplify:

$$X = k \cdot d_{\text{cycle}}$$

2. **Breaking X Into Components**: The distance X can be split into two parts:

   $$X = d_{\text{head to cycle}} + d_{\text{meeting in cycle}}$$

   Substituting into the earlier equation:

   $$d_{\text{head to cycle}} + d_{\text{meeting in cycle}} = k \cdot d_{\text{cycle}}$$

3. **Distance to the Start of the Cycle**: The key insight is that the portion $d_{\text{meeting in cycle}}$ (from C to M) is equal to the remainder when the slow pointer first enters the cycle. This means the slow pointer has effectively traveled the same distance as the fast pointer, modulo the cycle length.

   Now, when you reset one pointer to A and leave the other at M:

   - The pointer at A travels $d_{\text{head to cycle}}$ steps.
   - The pointer at M travels $d_{\text{cycle}} - d_{\text{meeting in cycle}}$ steps (to complete the cycle and meet at C).

   Since $d_{\text{meeting in cycle}} = d_{\text{cycle}} - d_{\text{head to cycle}}$, they meet at C, the start of the cycle.

Intuition

The algorithm works because resetting one pointer to the head splits the traversal into two phases:

- Traversing the distance from A to C (head to cycle).
- Completing the remaining distance in the cycle to meet at C.

This guarantees convergence at the start of the cycle.

I am unable to understand. Explain in better way

Let’s break it down with a simpler explanation and a step-by-step walkthrough to help clarify:

Problem Setup

1. You have a linked list with a cycle.
2. You want to find where the cycle begins.

Understanding the Two-Pointer Approach

We use two pointers:

- **Slow pointer** moves one step at a time.
- **Fast pointer** moves two steps at a time.

Step 1: Detect if a Cycle Exists

- If there's no cycle, the fast pointer will reach the end (`null`).
- If there's a cycle, the fast pointer will eventually catch up to the slow pointer. Why? Because the fast pointer moves faster and "laps" the slow pointer within the cycle.

Step 2: Why Do We Reset One Pointer to the Head?

When the slow and fast pointers meet, they meet at some point inside the cycle (let's call it `M`). Now, we want to find the start of the cycle (let's call it `C`).

Step 3: Understanding the Key Distances

1. **Let's define the distances:**
   - $L1$: Distance from the head of the list to the start of the cycle (`A` to `C`).
   - $L2$: Distance from the start of the cycle to the meeting point inside the cycle (`C` to `M`).
   - C: Total length of the cycle.

   So, the total distance traveled by the slow pointer is:

   $$\text{Distance traveled by slow pointer} = L1 + L2$$

   The fast pointer travels **twice** as far as the slow pointer:

   $$\text{Distance traveled by fast pointer} = 2 \times (L1 + L2)$$

2. **Fast pointer laps the cycle:** The fast pointer's distance can also be written as:

   $$\text{Distance traveled by fast pointer} = L1 + L2 + n \cdot C$$

(where n is the number of full laps around the cycle).

Equating the two:

$$2 \times (L1 + L2) = L1 + L2 + n \cdot C$$

3. **Simplify the equation:** Subtract $L1 + L2$ from both sides:

$$L1 + L2 = n \cdot C$$

This means:

- The distance the slow pointer has traveled is equal to n full cycles plus the distance from the head to the start of the cycle.

Step 4: Why Do the Two Pointers Meet at the Start of the Cycle?

- After the slow and fast pointers meet, reset one pointer (say, `slow`) to the **head** of the linked list.
- Keep the other pointer (`fast`) at the meeting point.
- Now, both pointers move **one step at a time**. Let's see what happens:
  - The pointer starting at the head (`A`) travels a distance of $L1$.
  - The pointer starting at the meeting point (`M`) travels the remaining cycle distance $C - L2$ to reach the cycle's start.

Since $L1 = C - L2$, **both pointers meet at the start of the cycle (`C`)**.

Simplified Visual Representation

- **Head to start of cycle**: $L1$
- **Cycle length**: C
- **Meeting point inside cycle**: $L2$

When one pointer is reset to the head:

- Head pointer travels $L1$.

- Meeting point pointer travels $C - L2$.

Since $L1 = C - L2$, **they meet at the cycle's start**.

Does this explanation make it clearer?