142. Linked List Cycle II

Medium | Hash Table | Linked List | Two Pointers

Problem Description

The problem presents a linked list and asks us to determine where a cycle begins within it. A cycle in a linked list happens when a node's `next` reference points back to a previous node in the list, causing a portion of the list to be traversed endlessly. We are given the head of the linked list, and we must find the node at which this cycle starts. If there is no cycle, our function should return `null`.

A linked list cycle is conceptually akin to a running track, where the entry point of the cycle is the "gate" to the track, and the cycle itself is the loop. Our goal is to figure out where this "gate" is located within the list.

Intuition

To resolve the problem of finding out the cycle's starting point, we can use the two-pointer technique, which is efficient and doesn't require extra memory for storage.

The intuition behind this algorithm involves a faster runner (the `fast` pointer) and a slower runner (the `slow` pointer), both starting at the head of the linked list. The `fast` pointer moves two steps at a time while the `slow` pointer moves only one. If a cycle exists, the `fast` pointer will eventually lap the `slow` pointer within the cycle, indicating that a cycle is present.

Once they meet, we can find the start of the cycle. To do this, we set up another pointer, called `ans`, at the head of the list and move it at the same pace as the `slow` pointer. The place where `ans` and the `slow` pointer meet again will be the starting node of the cycle.

Why does this work? If we consider that the distance from the list head to the cycle entrance is `x`, and the distance from the entrance to the meeting point is `y`, with the remaining distance back to the entrance being `z`, we can make an equation. Since the `fast` pointer travels the distance of `x + y + n * (y + z)` (where `n` is the number of laps made) and `slow` travels `x + y`, and `fast` is twice as fast as `slow`, then we can deduce that `x = n * (y + z) - y`, which simplifies to `x = (n - 1) * (y + z) + z`. This shows that starting a pointer at the head (`x` distance to the entrance) and one at the meeting point (`z` distance to the entrance) and moving them at the same speed will cause them to meet exactly at the entrance of the cycle.

Solution Approach

In this solution, we use the two-pointer technique, which involves having two iterators moving through the linked list at different speeds: `slow` and `fast`. `slow` moves one node at a time, while `fast` moves two nodes at a time.

The algorithm is divided into two main phases:

1. **Detecting the cycle**: Initially, both `slow` and `fast` are set to start at the head of the list. We then enter a loop in which `fast` advances two nodes and `slow` advances one node at a time. If there is no cycle, the `fast` pointer will eventually reach the end of the list and we can return `null` at this point, as there is no cycle to find the entrance of.

   However, if there is a cycle, `fast` is guaranteed to meet `slow` at some point within the cycle. The meeting point is not necessarily the entrance to the cycle, but it indicates that a cycle does exist.

2. **Finding the cycle starting node**: When `fast` and `slow` meet, we introduce a new pointer called `ans` and set it to the head of the list. Now, we move both `ans` and `slow` one node at a time. The node at which they conjoin is the start of the cycle.

Why does the above approach lead us to the start of the cycle? We derive this from the fact that:

- The distance from the list's head to the cycle's entrance is denoted as `x`.
- The distance from the cycle's entrance to the first meeting point is `y`.
- The remaining distance from the meeting point back to the entrance is `z`.

Using these variables, we know that when `fast` and `slow` meet, `fast` has traveled `x + y + n * (y + z)` which is the distance to the meeting point plus `n` laps of the cycle.

Since `fast` travels at twice the speed of `slow`, the distance `slow` has traveled (`x + y`) is half that of `fast`, leading us to the equation `2(x + y) = x + y + n * (y + z)`. Simplifying this, we find `x = (n - 1)(y + z) + z`.

This equation essentially states that the distance from the head to the cycle entrance (`x`) is equal to the distance from the meeting point to the entrance (`z`) plus some multiple of the cycle's perimeter (`y + z`). This is why moving the `ans` pointer from the head and `slow` from the meeting point at the same pace will lead them to meet at the cycle's entrance.

The Python code provided implements this approach efficiently, using only two extra pointers (`fast` and `slow`) for detection and one extra (`ans`) for locating the cycle's entrance.

Example Walkthrough

Let's consider a simple linked list example to walk through the solution approach:

Suppose we have the linked list `1 -> 2 -> 3 -> 4 -> 5 -> 2` (the last node points back to the second one, creating a cycle). Here, the node with the value `2` is the start of the cycle.

1. **Detecting the cycle**:
   - Initially, both `slow` and `fast` pointers are at the head of the list (node with value `1`).
   - Move `slow` to the next node (`2`) and `fast` two nodes forward (`3`).
   - Move `slow` to the next node (`3`) and `fast` two nodes forward (`5`).
   - Continue this process until `fast` and `slow` meet. In our case, after few iterations, `fast` and `slow` both point to one of the nodes inside the cycle (let's say they meet at node with value `4`).
2. **Finding the cycle starting node**:
   - Place the `ans` pointer at the head of the list (node with value `1`).
   - Move `ans` to the next node (`2`) and `slow` to the next node (`5`).
   - Continue moving both `ans` and `slow` one node at a time. As the pointers move one step each turn, they will meet at the node that is the start of the cycle.
   - In our case, `ans` and `slow` will both meet at the node with value `2`, which is the correct entrance to the cycle.

By following these steps and the reasonings behind the solution approach, we are able to find that the node with the value `2` is the entry point of the cycle in the linked list without using any extra memory for storage, only the two pointer variables `fast`, `slow`, and later the `ans` pointer.

Python Solution

```python
# Definition for singly-linked list.
class ListNode:
    def __init__(self, value=0, next=None):
        self.value = value
        self.next = next

class Solution:
    def detectCycle(self, head: Optional[ListNode]) -> Optional[ListNode]:
        # Initialize two pointers, slow and fast
        slow = fast = head

        # Traverse the linked list with two pointers moving at different speeds
        while fast and fast.next:
            slow = slow.next  # Slow pointer moves one step
            fast = fast.next.next  # Fast pointer moves two steps

            # Check if the slow and fast pointers meet, indicating a cycle
            if slow == fast:
                # Initialize another pointer to the head of the linked list
                start = head

                # Move both pointers at the same speed until they meet at the cycle's start node
                while start != slow:
                    start = start.next
                    slow = slow.next

                # Return the node where the cycle begins
                return start

        # If no cycle is detected, return None
        return None
```

Java Solution

```java
// Definition for singly-linked list.
class ListNode {
    int val;
    ListNode next;

    ListNode(int x) {
        val = x;
        next = null;
    }
}

public class Solution {
    // This method detects the node where the cycle begins in a linked list
    public ListNode detectCycle(ListNode head) {
        // Two pointers initialized to the start of the list
        ListNode fast = head;
        ListNode slow = head;

        // Loop until the fast pointer reaches the end of the list
        while (fast != null && fast.next != null) {
            // Move the slow pointer by one step
            slow = slow.next;
            // Move the fast pointer by two steps
            fast = fast.next.next;
            // If they meet, a cycle is detected
            if (slow == fast) {
                // Initialize another pointer to the start of the list
                ListNode start = head;
                // Move both pointers at the same pace
                while (start != slow) {
                    // Move each pointer by one step
                    start = start.next;
                    slow = slow.next;
                }
                // When they meet again, it's the start of the cycle
                return start;
            }
        }
        // If we reach here, no cycle was detected
        return null;
    }
}
```

C++ Solution

```cpp
// Definition for singly-linked list.
struct ListNode {
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(nullptr) {}
};

class Solution {
public:
    // This function detects a cycle in a linked list and returns the node
    // where the cycle begins. If there is no cycle, it returns nullptr.
    ListNode* detectCycle(ListNode* head) {
        ListNode* fastPointer = head; // Fast pointer will move two steps at a time
        ListNode* slowPointer = head; // Slow pointer will move one step at a time

        // Loop until the fast pointer reaches the end of the list,
        // or until the fast and slow pointers meet, indicating a cycle.
        while (fastPointer && fastPointer->next) {
            slowPointer = slowPointer->next;            // Move slow pointer one step
            fastPointer = fastPointer->next->next;      // Move fast pointer two steps

            // Check if the slow and fast pointers have met, indicating a cycle.
            if (slowPointer == fastPointer) {
                ListNode* entryPoint = head; // Start from the head

                // Loop until the entry point of the cycle is found.
                while (entryPoint != slowPointer) {
                    entryPoint = entryPoint->next;       // Move entry point one step
                    slowPointer = slowPointer->next;     // Move slow pointer one step
                }

                // The entry point is where the slow pointer and entry point meet.
                return entryPoint;
            }
        }

        // If the loop exits without the pointers meeting, there is no cycle.
        return nullptr;
    }
};
```

Typescript Solution

```typescript
// Definition for singly-linked list node.
interface ListNode {
    val: number;
    next: ListNode | null;
}

/**
 * Detects a cycle in a linked list and returns the node where the cycle begins.
 * If there is no cycle, it returns null.
 * @param head - The head of the singly-linked list.
 * @returns The node where the cycle begins or null if no cycle exists.
 */
function detectCycle(head: ListNode | null): ListNode | null {
    // Initialize two pointers, slow and fast.
    let slow: ListNode | null = head;
    let fast: ListNode | null = head;

    // Traverse the list with two pointers moving at different speeds.
    while (fast !== null && fast.next !== null) {
        slow = slow!.next; // Move slow pointer one step.
        fast = fast.next.next; // Move fast pointer two steps.

        // If slow and fast pointers meet, a cycle exists.
        if (slow === fast) {
            // Initialize another pointer to the beginning of the list.
            let startPoint: ListNode | null = head;
            // Move the startPoint and slow pointer at the same speed.
            while (startPoint !== slow) {
                startPoint = startPoint!.next;
                slow = slow!.next;
            }
            // The node where both pointers meet is the start of the cycle.
            return startPoint;
        }
    }

    // If no cycle is detected, return null.
    return null;
}
```

Time and Space Complexity

The time complexity of the code is `O(n)`, where `n` is the number of nodes in the linked list. This is because in the worst case, both `fast` and `slow` pointers traverse the entire list to detect a cycle, and then a second pass is made from the `head` to the point of intersection which is also linear in time.

The space complexity of the code is `O(1)`. This is due to the fact that no additional space proportional to the size of the input linked list is being allocated; only a fixed number of pointer variables `fast`, `slow`, and `ans` are used, irrespective of the size of the linked list.