141. Linked List Cycle

Easy Hash Table Linked List Two Pointers

Problem Description

The problem provides us with the head of a singly-linked list and asks us to determine whether there is a cycle within this linked list. A cycle exists if a node's `next` pointer points to a previously traversed node, meaning one could loop indefinitely within the linked list. We do not have access to the `pos` variable which indicates the node that the last node is connected to (forming the cycle). Our goal is to figure out whether such a cycle exists by returning `true` if it does or `false` otherwise.

Intuition

The intuition behind the solution relies on the "tortoise and the hare" algorithm, or Floyd's cycle-finding algorithm. The core idea is to use two pointers, a slow pointer and a fast pointer. The slow pointer moves one step at a time, while the fast pointer moves two steps at a time.

If the linked list has no cycle, the fast pointer will eventually reach the end of the list (null).

If the linked list has a cycle, the fast pointer will start looping within the cycle and eventually both pointers will be inside the cycle. Given that the fast pointer moves twice as fast as the slow pointer, it will catch up to the slow pointer from behind, meeting at some node within the cycle. This is reminiscent of a track where a faster runner laps a slower runner.

Once both pointers occupy the same node (they meet), we can confirm that a cycle exists and return `true`. If the while loop terminates (fast pointer reaches the list's end), we return `false` as no cycle is present.

Solution Approach

The solution uses the Floyd's cycle-finding algorithm. We initialize two pointers, `slow` and `fast`, and both of them start at the head of the linked list.

While traversing the list:

- The `slow` pointer is moved by one node at a time using `slow.next`.
- The `fast` pointer is moved by two nodes at a time using `fast.next.next`.

This means after each iteration through our loop, `fast` is two nodes ahead of `slow` (assuming they don't yet point to the same node and the `fast` pointer has not encountered the end of the list).

- If the linked list has no cycle, the `fast` pointer will reach a node that has a `null next` pointer and the loop will end, hence the condition `while fast and fast.next:` in our loop.
- If there is a cycle, `fast` will eventually meet `slow` inside the cycle, since it moves twice as fast and will thus close the gap between them by one node per step inside the cycle.

The loop continues until either `fast` reaches the end of the list (indicating there is no cycle), or `fast` and `slow` meet (indicating there is a cycle). If `fast` and `slow` meet (i.e., `slow == fast`), we return `true`. If the loop ends without them meeting, we return `false`.

Here is a pseudo-code breakdown of the algorithm:

```
initialize slow and fast pointers at head
while fast is not null and fast.next is not null
    move slow pointer to slow.next
    move fast pointer to fast.next.next
    if slow is the same as fast
        return true (cycle detected)
return false (no cycle since fast reached the end)
```

The crux of the method is that the existence of a cycle is exposed by the movement of the `fast` pointer in relation to the `slow` pointer. If they ever point to the same node, it means there is a cycle because the `fast` pointer must have lapped the `slow` pointer somewhere within the cycle.

Example Walkthrough

Let's say we have the following linked list where the last node points back to the second node, forming a cycle:

`1 -> 2 -> 3 -> 4 -> 5 ^ | |_________|`

To illustrate the solution approach with this example:

1. We initialize two pointers at the head:
   - `slow` is at node `1`
   - `fast` is at node `1`
2. We then start iterating through the list:
   - Move `slow` to the next node (`2`), move `fast` two nodes ahead (`3`).
3. On the next iteration:
   - Move `slow` to `3`, move `fast` to `5`.
4. Next iteration:
   - Move `slow` to `4`, move `fast` two nodes ahead but because of the cycle, it lands on `2`.
5. On the following iteration:
   - Move `slow` to `5`, `fast` moves to `4`.

At this point the loop continues:

6. `slow` moves to node `2` (following the cycle) and `fast` to node `3`.
7. Next, `slow` moves to `3` and `fast` jumps two steps, landing on `5` again.
8. Finally, `slow` goes to `4` and `fast` which is at `5` now makes a two-step jump and lands on `4`, meeting the `slow` pointer.

Since `slow` and `fast` are both pointing to `4`, this is evidence of a cycle. Thus, we return `true`.

If at any point `fast` or `fast.next` were null, it would mean that `fast` has reached the end of the linked list and there is no cycle, in which case we would return `false`. However, in our example, as `slow` and `fast` meet, we have confirmed the presence of a cycle.

Solution Implementation

Python

```python
class ListNode:
    def __init__(self, value):
        self.value = value
        self.next = None


class Solution:
    def hasCycle(self, head: ListNode) -> bool:
        # Initialize two pointers, slow and fast. Both start at the head of the list.
        slow_pointer = fast_pointer = head

        # Loop until fast_pointer reaches the end of the list
        while fast_pointer and fast_pointer.next:
            # Move slow_pointer by one step
            slow_pointer = slow_pointer.next
            # Move fast_pointer by two steps
            fast_pointer = fast_pointer.next.next

            # If slow_pointer and fast_pointer meet, there's a cycle
            if slow_pointer == fast_pointer:
                return True

        # If fast_pointer reaches the end, there is no cycle
        return False
```

Java

```java
/**
 * Definition for singly-linked list.
 */
class ListNode {
    int value; // The value of the node.
    ListNode next; // Reference to the next node in the list.

    // Constructor to initialize the node with a specific value.
    ListNode(int value) {
        this.value = value;
        this.next = null;
    }
}

public class Solution {
    /**
     * Detects if there is a cycle in the linked list.
     *
     * @param head The head of the singly-linked list.
     * @return true if there is a cycle, false otherwise.
     */
    public boolean hasCycle(ListNode head) {
        // Initialize two pointers, the slow pointer moves one step at a time.
        ListNode slow = head;
        // The fast pointer moves two steps at a time.
        ListNode fast = head;

        // Keep traversing the list as long as the fast pointer and its next are not null.
        while (fast != null && fast.next != null) {
            // Move the slow pointer one step.
            slow = slow.next;
            // Move the fast pointer two steps.
            fast = fast.next.next;

            // If the slow and fast pointers meet, a cycle exists.
            if (slow == fast) {
                return true;
            }
        }
        // If the loop ends without the pointers meeting, there is no cycle.
        return false;
    }
}
```

C++

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int value;
 *     ListNode *next;
 *     ListNode(int x) : value(x), next(nullptr) {}
 * };
 */
class Solution {
public:
    // Checks if the linked list has a cycle
    bool hasCycle(ListNode *head) {
        ListNode *slowPointer = head; // Initialize slow pointer
        ListNode *fastPointer = head; // Initialize fast pointer

        // Loop until the fast pointer reaches the end of the list
        while (fastPointer && fastPointer->next) {
            slowPointer = slowPointer->next;          // Move slow pointer by 1 node
            fastPointer = fastPointer->next->next;    // Move fast pointer by 2 nodes

            // If both pointers meet at the same node, there is a cycle
            if (slowPointer == fastPointer) {
                return true;
            }
        }

        // If the fast pointer reaches the end of the list, there is no cycle
        return false;
    }
};
```

TypeScript

```typescript
// Function to detect whether a singly-linked list has a cycle.
// This uses Floyd's Tortoise and Hare algorithm.
function hasCycle(head: ListNode | null): boolean {
    // Initialize two pointers, 'slowPointer' and 'fastPointer' at the head of the list.
    let slowPointer = head;
    let fastPointer = head;

    // Traverse the list with both pointers.
    while (fastPointer !== null && fastPointer.next !== null) {
        // Move 'slowPointer' one step.
        slowPointer = slowPointer.next;
        // Move 'fastPointer' two steps.
        fastPointer = fastPointer.next.next;

        // If 'slowPointer' and 'fastPointer' meet, a cycle is detected.
        if (slowPointer === fastPointer) {
            return true;
        }
    }

    // If 'fastPointer' reaches the end of the list, no cycle is present.
    return false;
}
```

```python
class ListNode:
    def __init__(self, value):
        self.value = value
        self.next = None


class Solution:
    def hasCycle(self, head: ListNode) -> bool:
        # Initialize two pointers, slow and fast. Both start at the head of the list.
        slow_pointer = fast_pointer = head

        # Loop until fast_pointer reaches the end of the list
        while fast_pointer and fast_pointer.next:
            # Move slow_pointer by one step
            slow_pointer = slow_pointer.next
            # Move fast_pointer by two steps
            fast_pointer = fast_pointer.next.next

            # If slow_pointer and fast_pointer meet, there's a cycle
            if slow_pointer == fast_pointer:
                return True

        # If fast_pointer reaches the end, there is no cycle
        return False
```

Time and Space Complexity

The given Python code is using the Floyd's Tortoise and Hare algorithm to find a cycle in a linked list.

Time Complexity

The time complexity of the code is `O(N)`, where `N` is the number of nodes in the linked list. In the worst-case scenario, the fast pointer will meet the slow pointer in one pass through the list, if there is a cycle.

Space Complexity

The space complexity of the code is `O(1)`. This is because the algorithm uses only two pointers, regardless of the size of the linked list, which means it only requires a constant amount of extra space.