369. Plus One Linked List

Medium | Linked List | Math | Leetcode Link

Problem Description

The problem is about incrementing a given non-negative integer by one. However, the integer isn't represented in the traditional numerical form; it's represented as a linked list, with each node containing a single digit. The head of the linked list contains the most significant digit (MSD), while the tail contains the least significant digit (LSD). For example, the integer `123` would be represented as a linked list `1 -> 2 -> 3`.

The goal is to add one to this integer and return the resulting linked list following the same MSD to LSD format. The problem must be solved in such a way that the linked list structure is mantained, without converting the entire list to an integer or a series of strings.

Intuition

To intuitively approach this solution, we need to think about how we generally add one to a number. Starting from the LSD, we add one; if this causes a digit to exceed `9`, it rolls over to `0`, and we add `1` to the next significant digit, continuing this process until there are no more rollovers or we've reached the MSD.

Following this idea in a linked list, we first need to locate the right-most digit that is not a `9`. This digit is important because it's the one that will potentially increase by one. All digits to the right of it will become zero if there's a rollover. If all digits are `9`, we'll need a new node to accommodate the extra digit generated from the rollover (from `999` to `1000`, for instance).

Here's a step-by-step breakdown of the approach:

1. Use a sentinel (dummy) node at the start of the list to handle cases when a new digit must be added (a new MSD).
2. Traverse the linked list to find the rightmost node whose value is not `9`. This node is named `target`. If all digits are `9`, `target` will remain as the dummy node.
3. Increment the `target` node's value by `1`.
4. Set all nodes to the right of `target` (if any) to `0`, as these have been "rolled over".
5. If the dummy node's value is `0`, it means no new MSD was added and we can return the original head. Otherwise, return the dummy node as it now contains the new MSD.

This approach works due to the linked list's inability to be accessed randomly (we can't go back once we pass a node), making it necessary to mark the last non-`9` whilst initially traversing the list. This will prevent us from needing a second full traversal.

Solution Approach

The solution makes use of a simple linked list traversal and manipulation approach. We follow the below steps algorithmically:

1. **Initialization**:
   - Create a dummy node (`dummy`) with `0` as its value, which will precede our original linked list. This helps in scenarios where a carry over might add an additional digit to the front. `dummy.next` is pointed to the head of the input list.
   - Initialize a variable `target` to point to the dummy node. This `target` variable will be used to remember the position in the list where we might later add one.
2. **Traverse the List**:
   - Using a `while` loop, we start to traverse the list starting from the head while checking for a condition, `head != None`. This condition ensures we stop at the end of the list.
   - During traversal, we look for the right-most node that is not a `9`. We update `target` to point to this node each time we encounter a non-`9` value.
3. **Increment the Target Value**:
   - After the traversal, we know the `target` node is the right-most node that isn't `9`. We increment `target.val` by `1`.
4. **Handle Rollover**:
   - Move `target` to its next node.
   - Continue another loop to set all the following values to `0`, turning all `9`s that come after the incremented value into `0`s (as they have been rolled over).
5. **Return the Modified List**:
   - If the dummy node's value is still `0`, it implies the increment did not add a new digit, so we return `dummy.next` as the head of the updated list.
   - If the dummy node's value is `1`, it implies a new digit has been added due to a carry (for example, from `999` to `1000`), so we return the `dummy` node itself as it is now the head of the updated list.

In terms of data structures, we simply use the given linked list nodes and a single additional node for the dummy. The space complexity of the algorithm is O(1) since we are modifying the input linked list in place and only using a fixed number of extra variables. The time complexity of the algorithm is O(n), where n is the number of nodes in the linked list, since we are potentially traversing the entire list.

The patterns used in the solution include the two-pointer technique as well as the sentinel node pattern. The two-pointer technique, in this case, involves the `target` and `head` pointers to traverse and modify the input list. The sentinel node pattern is used to simplify operations at the head of the list, allowing us to handle edge cases more gracefully.

Example Walkthrough

Let's consider a linked list that represents the integer `129`:

```
1 -> 2 -> 9
```

Here's a step-by-step walkthrough of the solution approach with this example:

1. **Initialization**:
   - Create a dummy node with a value `0` and connect it to the head of the list. The list now looks like `0 -> 1 -> 2 -> 9`. Initialize the `target` to point to `dummy`.
2. **Traverse the List**:
   - Start traversing the list: Move to `1`. Since `1` is not `9`, update `target` to this node.
   - Move to `2`. Since `2` is not `9`, update `target` to this node.
   - Move to the final node `9`. Keep `target` at `2` as the final node is indeed a `9`.
3. **Increment the Target Value**:
   - `target` is pointing to the node with value `2`. Increment this node's value by `1`. The list now temporarily looks like `0 -> 1 -> 3 -> 9`.
4. **Handle Rollover**:
   - Set all nodes to the right of `target` to `0`. This is just the final node in this example. Now the list looks like `0 -> 1 -> 3 -> 0`.
5. **Return the Modified List**:
   - Check the dummy node. It still has the value `0`, so no new MSD has been added. Thus, the head of the updated list is `dummy.next`.
   - The final resulting list is `1 -> 3 -> 0`, which represents the integer `130`.

The example illustrates the given steps, showing how to navigate and modify the list without turning it into another data type. This results in a linked list that represents the initial number incremented by `1`.

Python Solution

```python
class ListNode:
    def __init__(self, val=0, next_node=None):
        self.val = val
        self.next = next_node

class Solution:
    def plusOne(self, head: ListNode) -> ListNode:
        # Create a dummy node before the head to handle edge cases easily
        dummy = ListNode(0)
        dummy.next = head

        # This variable will keep track of the last node before the sequence of 9's
        non_nine_node = dummy

        # Traverse the linked list to find the last node that is not a '9'
        while head:
            if head.val != 9:
                non_nine_node = head
            head = head.next

        # Increase the value of the last non-nine node by 1
        non_nine_node.val += 1

        # Make 'current' point to the node right after the incremented node
        current = non_nine_node.next

        # Set all the nodes after the last non-nine node to '0'
        while current:
            current.val = 0
            current = current.next

        # If the dummy node's value is '0', it means the linked list doesn't have leading zeros
        # If the dummy node's value is not '0', the list starts with a '1' followed by zeros
        return dummy if dummy.val != 0 else dummy.next

```

Java Solution

```java
/**
 * Definition for singly-linked list.
 */
public class ListNode {
    int val; // Value of the node
    ListNode next; // Reference to the next node

    ListNode() {}
    ListNode(int val) { this.val = val; }
    ListNode(int val, ListNode next) {
        this.val = val;
        this.next = next;
    }
}

class Solution {
    public ListNode plusOne(ListNode head) {
        // Create a dummy node which initially points to the head of the list
        ListNode dummy = new ListNode(0);
        dummy.next = head; // Connect the dummy node to the head of the list
        ListNode notNine = dummy; // This will point to the last node not equal to 9

        // Traverse the list to find the rightmost not-nine node
        while (head != null) {
            if (head.val != 9) {
                notNine = head; // Update the rightmost not-nine node
            }
            head = head.next; // Move to the next node
        }

        // Increment the value of the rightmost not-nine node
        notNine.val += 1;

        // Set all the nodes right to the increased node to 0
        ListNode current = notNine.next;
        while (current != null) {
            current.val = 0;
            current = current.next; // Move to the next node
        }

        // Check if dummy node has the incremented value (meaning carry was there)
        // If dummy val is 1, return the dummy node, else return the original list without the dummy
        return dummy.val == 1 ? dummy : dummy.next;
    }
}

```

C++ Solution

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    // Function to add one to a number represented as a linked list.
    ListNode* plusOne(ListNode* head) {
        // Create a dummy head in case we need to add a new head (e.g., 999 + 1 = 1000).
        ListNode* dummyHead = new ListNode(0);
        dummyHead->next = head;

        // 'nonNineNode' will point to the rightmost node that is not a 9 or to the dummy head.
        ListNode* nonNineNode = dummyHead;

        // Traverse the list to find the rightmost node that is not a 9.
        while (head != nullptr) {
            if (head->val != 9) {
                nonNineNode = head;
            }
            head = head->next;
        }

        // Increment the value of the rightmost non-nine node.
        ++nonNineNode->val;

        // Move to the next node, which is the first in the sequence of 9's.
        nonNineNode = nonNineNode->next;

        // Reset all the following 9's to 0's because we've already added one to the preceding digit.
        while (nonNineNode != nullptr) {
            nonNineNode->val = 0;
            nonNineNode = nonNineNode->next;
        }

        // If the dummy head's value was changed, it means we added a new digit, so we return the dummy head.
        // Otherwise, we return the original head of the list.
        return dummyHead->val == 1 ? dummyHead : dummyHead->next;
    }
};

```

Typescript Solution

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

// Function to create a new ListNode.
function createListNode(val: number, next: ListNode | null = null): ListNode {
  return { val: val, next: next };
}

// Function to add one to a number represented as a linked list.
function plusOne(head: ListNode | null): ListNode | null {
  // Create a dummy head in case we need to add a new head (e.g., from 999 + 1 = 1000).
  let dummyHead = createListNode(0);
  dummyHead.next = head;

  // 'nonNineNode' will point to the rightmost node that is not a 9, or to the dummy head if all are 9s.
  let nonNineNode: ListNode = dummyHead;

  // Traverse the list to find the rightmost node that is not a 9.
  while (head !== null) {
    if (head.val !== 9) {
      nonNineNode = head;
    }
    head = head.next;
  }

  // Increment the value of the rightmost non-nine node.
  nonNineNode.val++;

  // Move to the next node, which is the first in the sequence of 9s after the incremented digit.
  nonNineNode = nonNineNode.next;

  // Reset all the following 9s to 0s because we have added one to the preceding digit.
  while (nonNineNode !== null) {
    nonNineNode.val = 0;
    nonNineNode = nonNineNode.next;
  }

  // If the dummy head's value was changed, it means we added a new digit at the start, so return the dummy head.
  // Otherwise, return the original head of the list.
  return dummyHead.val === 1 ? dummyHead : dummyHead.next;
}

```

Time and Space Complexity

Time Complexity

The given Python code traverses the linked list twice. In the first traversal, it looks through all the nodes to find the last node before a sequence of '9's that needs to be incremented. In the worst case, this traversal looks at every node exactly once, resulting in a time complexity of `O(n)`, where `n` is the length of the linked list.

The second traversal occurs after the increment and only traverses the portion of the list that consists of '9's, turning them into '0's. In the worst case, this could again traverse the entire list (in the case that all nodes are '9's except the first), giving this traversal a time complexity of `O(n)` as well.

The total worst-case time complexity, combining both traversals, remains `O(n)` because the constants do not matter for Big O notation and the list is only traversed a constant number of times (specifically, twice).

Space Complexity

The space complexity of the algorithm is dependent on the additional variables defined in the method and not on the input size. The method uses a few constant extra space for pointers (`dummy`, `target`, and `head`), and does not create any additional data structures that grow with the input size. Therefore, the space complexity is `O(1)`. The dummy node created does not count as extra space since it's only a fixed-size pointer to the existing list (and not extra nodes being created).