19. Remove Nth Node From End of List

Medium Linked List Two Pointers

Leetcode Link

Problem Description

The problem presents us with a linked list and requires us to remove the nth node from the end of the list. A linked list is a linear collection of elements called nodes, where each node contains data and a reference (or link) to the next node in the sequence. The `head` of a linked list refers to the first node in this sequence. Our goal is to remove the nth node from the last node and return the updated list's `head`.

In simple terms, if we were to count nodes starting from the last node back to the first, we need to find and remove the node that sits at position 'n'. For example, if the linked list is `1->2->3->4->5` and `n` is 2, the resulting linked list after the operation should be `1->2->3->5`, since the `4` is the 2nd node from the end.

The challenge with this task is that we can't navigate backwards in a singly linked list, so we need to figure out a way to reach the nth node from the end only by moving forward.

Intuition

To address the problem, we apply a two-pointer approach, which is a common technique in linked list problems. The intuition here lies in maintaining two pointers that we'll call `fast` and `slow`. Initially, both of these pointers will start at a dummy node that we create at the beginning of the list, which is an auxiliary node to simplify edge cases, such as when the list contains only one node, or when we need to remove the first node of the list.

The `fast` pointer advances `n` nodes into the list first. Then, both `slow` and `fast` pointers move at the same pace until `fast` reaches the end of the list. At this point, `slow` will be right before the node we want to remove. By updating the `next` reference of the `slow` pointer to skip the target node, we effectively remove the nth node from the end of the list.

The use of a dummy node is a clever trick to avoid having to separately handle the special case where the node to remove is the first node of the list. By ensuring that our `slow` pointer starts before the `head` of the actual list, we can use the same logic for removing the nth node, regardless of its position in the list.

After we've completed the removal, we return `dummy.next`, which is the updated list's `head`, after the dummy node.

Solution Approach

The solution implements an efficient approach to solving the problem by using two pointers and a dummy node. Here's a step-by-step walkthrough of the implementation:

1. **Create a Dummy Node:** A dummy node is created and its `next` pointer is set to the `head` of the list. This dummy node serves as an anchor and helps to simplify the removal process, especially if the `head` needs to be removed.

   ```
   dummy = ListNode(next=head)
   ```

2. **Initialize Two Pointers:** Two pointers `fast` and `slow` are introduced and both are set to the dummy node. This allows for them to start from the very beginning of the augmented list (which includes the dummy node).

   ```
   fast = slow = dummy
   ```

3. **Advance Fast Pointer:** The `fast` pointer advances `n` steps into the list.

   ```
   for _ in range(n):
       fast = fast.next
   ```

4. **Move Both Pointers:** Both pointers then move together one step at a time until the `fast` pointer reaches the end of the list. By this time, `slow` is at the position right before the nth node from the end.

   ```
   while fast.next:
       slow, fast = slow.next, fast.next
   ```

5. **Remove the Target Node:** Once `slow` is in position, its `next` pointer is changed to skip the target node which effectively removes it from the list.

   ```
   slow.next = slow.next.next
   ```

6. **Return Updated List:** Finally, the next node after the dummy is returned as the new head of the updated list.

   ```
   return dummy.next
   ```

The algorithm fundamentally relies on the two-pointer technique to find the nth node from the end. The use of a dummy node facilitates the removal of nodes, including the edge case of the `head` node, with a consistent method. The space complexity is O(1) since no additional space is required aside from the pointers, and the time complexity is O(L), where L is the length of the list, as we traverse the list with two pointers in a single pass.

Example Walkthrough

Let's use the linked list `1->2->3->4->5` as an example to illustrate the solution approach described above, where `n` is 2, meaning we want to remove the 2nd node from the end, which is the node with the value `4`. Here's how the algorithm would be applied:

1. **Create a Dummy Node:** We create a dummy node with `None` as its value and link it to the head of our list. Our list now looks like `dummy->1->2->3->4->5`.
2. **Initialize Two Pointers:** We set both `fast` and `slow` pointers to the dummy node. They both point to `dummy` initially.
3. **Advance Fast Pointer:** As `n` is 2, we advance the `fast` pointer `n` steps into the list. `fast` becomes:
   - After 1st step: `fast` points to `1`
   - After 2nd step: `fast` points to `2`
4. **Move Both Pointers:** Now we move both `slow` and `fast` pointers together one step at a time until `fast` reaches the last node:
   - `slow` points to `dummy` and `fast` points to `2` (Start)
   - `slow` points to `1` and `fast` points to `3`
   - `slow` points to `2` and `fast` points to `4`
   - `slow` points to `3` and `fast` points to `5` (End)

   At the end of this step, `slow` points to the `3` and `fast` points to the last node `5`.
5. **Remove the Target Node:** Now, `slow.next` is pointing to the node `4`, which we want to remove. We update `slow.next` to point to `slow.next.next` (which is `5`), effectively removing the `4` from the list.
6. **Return Updated List:** Finally, we return the next node after the dummy, which is the `head` of our updated list (`1->2->3->5`). The updated list after removal is as follows: `1->2->3->5`.

The problem is solved in one pass and the node is successfully removed according to the given constraints.

Python Solution

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

class Solution:
    def remove_nth_from_end(self, head: ListNode, n: int) -> ListNode:
        # Create a dummy node that points to the head of the list. This helps simplify edge cases.
        dummy_node = ListNode(next_node=head)

        # Initialize two pointers to the dummy node.
        fast_pointer = slow_pointer = dummy_node

        # Move fast pointer n steps ahead so it maintains a gap of n between slow pointer.
        for _ in range(n):
            fast_pointer = fast_pointer.next

        # Move both pointers until fast pointer reaches the end of the list.
        # Keeping the gap of 'n' between them, slow pointer will point to the node
        # just before the one to be removed.
        while fast_pointer.next:
            slow_pointer, fast_pointer = slow_pointer.next, fast_pointer.next

        # Remove the nth node from the end by skipping over it with the slow pointer.
        slow_pointer.next = slow_pointer.next.next

        # Return the new head, which is the next node of dummy node.
        return dummy_node.next
```

Java Solution

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

    // Constructor to initialize the node without a next node
    ListNode() {}

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

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

public class Solution {

    // Removes the nth node from the end of the list
    public ListNode removeNthFromEnd(ListNode head, int n) {
        // Create a dummy node that precedes the head of the list
        ListNode dummyNode = new ListNode(0, head);

        // Initialize two pointers, starting at the dummy node
        ListNode fastPointer = dummyNode;
        ListNode slowPointer = dummyNode;

        // Move the fast pointer n steps ahead
        while (n-- > 0) {
            fastPointer = fastPointer.next;
        }

        // Move both pointers until the fast pointer reaches the end of the list
        while (fastPointer.next != null) {
            slowPointer = slowPointer.next;
            fastPointer = fastPointer.next;
        }

        // Skip the node that is nth from the end
        slowPointer.next = slowPointer.next.next;

        // Return the head of the modified list, which is the next node of dummy node
        return dummyNode.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:
    // Removes the n-th node from the end of the list and returns the new head
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        // Create a dummy node pointing to the head to handle edge cases easily
        ListNode* dummyNode = new ListNode(0, head);
        // These pointers will help find the node to remove
        ListNode* fastPointer = dummyNode;
        ListNode* slowPointer = dummyNode;

        // Move fastPointer n steps ahead
        for(int i = 0; i < n; i++) {
            fastPointer = fastPointer->next;
        }

        // Move both pointers until fastPointer reaches the last node
        while (fastPointer->next != nullptr) {
            slowPointer = slowPointer->next;
            fastPointer = fastPointer->next;
        }

        // slowPointer will be just before the node we want to remove
        slowPointer->next = slowPointer->next->next;

        // The returned head might not be the original head if the first node was removed
        ListNode* newHead = dummyNode->next;
        delete dummyNode; // Clean up memory used by dummyNode
        return newHead;
    }
};
```

Typescript Solution

```typescript
// Type definition for a single node of a singly linked list.
type ListNode = {
    val: number;
    next: ListNode | null;
};

// Function to create a new ListNode with given value and next node.
function createListNode(val: number = 0, next: ListNode | null = null): ListNode {
    return { val, next };
}

/**
 * Removes the n-th node from the end of the list and returns the head of the modified list.
 * @param {ListNode | null} head - The head of the linked list.
 * @param {number} n - The position from the end of the list to remove the node.
 * @returns {ListNode | null} - The head of the list after removal.
 */
function removeNthFromEnd(head: ListNode | null, n: number): ListNode | null {
    // Create a dummy node that will help in edge cases, like removing the first node.
    const dummyNode = createListNode(0, head);
    // Initialize two pointers to start at the dummy node.
    let fastPointer: ListNode | null = dummyNode;
    let slowPointer: ListNode | null = dummyNode;

    // Advance the fast pointer n steps ahead of the slow pointer.
    for (let i = 0; i < n; ++i) {
        fastPointer = fastPointer.next;
    }

    // Move both pointers until the fast pointer reaches the end of the list
    while (fastPointer.next !== null) {
        slowPointer = slowPointer.next;
        fastPointer = fastPointer.next;
    }

    // Skip the target node by assigning its next node to the next of the slow pointer.
    // This effectively removes the target node from the list.
    slowPointer.next = slowPointer.next.next;

    // Return the new head of the list. The dummy node's next pointer points to the list head.
    return dummyNode.next;
}
```

Time and Space Complexity

The given code is designed to remove the nth node from the end of a singly linked list. It employs the two-pointer technique with `fast` and `slow` pointers. Let's analyze the time and space complexity:

Time Complexity

The time complexity is determined by the number of operations required to traverse the linked list. There are two main steps in this algorithm:

1. The `fast` pointer moves `n` steps ahead – this takes `O(n)` time.
2. Both `fast` and `slow` pointers move together until `fast` reaches the last node. The worst-case scenario is when `n` is 1, which would cause the `slow` pointer to traverse the entire list of size `L` (list length), taking `O(L)` time.

Since the list is traversed at most once, the overall time complexity is `O(L)`.

Space Complexity

The space complexity is determined by the amount of additional space used by the algorithm, which does not depend on the input size:

- The `dummy` node and the pointers (`fast` and `slow`) use constant extra space, regardless of the linked list's size.
- Therefore, the space complexity is `O(1)` for the extra space used by the pointers and the dummy node.