237. Delete Node in a Linked List

Medium Linked List

Problem Description

In this problem, we're tasked with deleting a specific node from a singly-linked list, but with a twist: we're only given access to that particular node, not the head of the list. Moreover, the given node will not be the list's tail, and all node values in the list are unique. What makes this task unique is that typically, to remove a node from a linked list, we would need access to the node before the one we want to delete to adjust the `next` pointer. However, since we're only given the target node itself (and guaranteed it's not the last one), we need a different strategy. Our goal is to remove the node from the list while preserving the order of remaining nodes. It's important to understand that "deleting the node" here means ensuring the value of the said node doesn't appear in the linked list sequence, and the length of the linked list is decreased by one.

Intuition

Understanding the constraints and the standard operations on a linked list is key to solving this problem. Given that we cannot directly remove the target node by changing its previous node's `next` pointer (because we don't have access to it), we can use a clever trick: copy the value from the next node into the target node, and then remove the next node instead. This effectively overwrites the target node's value with its successor's value, and then deleting the successor (which we have access to) achieves the goal of removing the target node's value from the list. The key realization here is that we're not actually deleting the given node, but rather, we're making it a clone of the next node in terms of value, then skipping over the next node, effectively removing its presence from the list.

Solution Approach

The solution takes advantage of the properties of a singly-linked list and uses the constraint that is provided: We are given access to the node that must be deleted, and it is not the last node of the list.

Here's the step-by-step approach that is used in the solution:

1. Copy the value from the next node into the current node. This is done by using the statement `node.val = node.next.val`. After this step, the current node (`node`) and the next node (`node.next`) have the same value, the one that was originally in `node.next`.
2. Delete the next node from the linked list. This doesn't mean removing it from memory, since Python's garbage collector will take care of that eventually. We simply need to change the reference of the current node's `next` to skip the next node and point to the node following it. We accomplish this with `node.next = node.next.next`.

By performing these two steps, we've essentially shifted the value from the `node.next` to `node`, and then we unlink `node.next` from the chain. Visually, if our list was `A -> B -> C -> D` and we wanted to delete `B`, we make `B` take on the value and position of `C`, resulting in `A -> C -> D`. Then `B` (with the value of `C`) is no longer part of the list. This fulfills all the requirements for the deletion operation as per the problem statement.

Example Walkthrough

Let's assume we have a singly-linked list that looks as follows:

```
4 -> 5 -> 1 -> 9
```

And we're asked to delete the node with value `5` from it. We do not have access to the head of the list, only to the target node. Here's how we would solve it using the solution approach:

1. We start with the node we want to delete, which has the value `5`. The list looks like this:

   ```
   4 -> 5 -> 1 -> 9
   ```

2. According to our 1st step in the solution approach, we copy the value of the next node (`1`) into the current node (`5`). Our list will then look like this:

   ```
   4 -> 1 -> 1 -> 9
   ```

   Now, the node we wanted to delete (`5`) has been overwritten with the value of `1`.

3. Next, we perform the 2nd step in our solution where we unlink the node after the current node (the second `1`). This is done by setting the `next` pointer of our current node to the `next` of the next node. After this step, the list now looks like this:

   ```
   4 -> 1 -> 9
   ```

4. The node originally containing `5` has now been removed (or rather its value copied over and its successor unlinked), and our linked list preserves the original order with the target node no longer present.

The deletion is successful and has been performed using the given access constraints. The linked list now correctly contains the sequence `4 -> 1 -> 9`.

Solution Implementation

Python

```python
# Definition for a singly-linked list node.
class ListNode:
    def __init__(self, value):
        self.value = value  # Initialize node's value
        self.next = None    # Next node reference


class Solution:
    def deleteNode(self, node):
        """
        Delete a node from a singly-linked list, given only access to that node.

        :param node: The node to be deleted from the linked list.
        :type node: ListNode
        """
        node.value = node.next.value  # Copy the value from the next node to the current node
        node.next = node.next.next    # Bypass the next node by pointing to the node after next
```

Java

```java
// Class definition for a singly-linked list node.
class ListNode {
    int val; // The value of the node.
    ListNode next; // Reference to the next node in the list.

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

// Class containing the solution method.
class Solution {
    /**
     * Deletes a node (except the tail) from a singly-linked list, given only access to that node.
     * The input node should not be the tail, and the list should have at least two elements.
     * @param node the node to be deleted
     */
    public void deleteNode(ListNode node) {
        // Copy the value of the next node into the current node.
        node.val = node.next.val;
        // Set the current node's next pointer to skip the next node, effectively deleting it.
        node.next = node.next.next;
    }
}
```

C++

```cpp
// Definition for singly-linked list node.
struct ListNode {
    int val;        // The value of the node.
    ListNode *next;   // Pointer to the next node in the list.

    // Constructor to initialize a ListNode with a value and optional link to the next node.
    ListNode(int x) : val(x), next(nullptr) {} // Using nullptr instead of NULL
};

class Solution {
public:
    // Deletes the given node (except the tail) from the linked list.
    // node represents the node to be deleted.
    void deleteNode(ListNode* node) {
        // This function assumes that the node to delete is not the tail
        // and that we are not handling edge cases where node could be null.

        // Copy the value from the next node to the current node.
        node->val = node->next->val;

        // Save the next next node, which will be the new next after deletion.
        ListNode* next_next = node->next->next;

        // Delete the next node, effectively removing it from the list.
        delete node->next;

        // Make the next pointer of the current node point to the new next node (next_next).
        node->next = next_next;
    }
};
```

TypeScript

```typescript
// Given the node to be deleted (except the tail), this function will delete the node in place.
function deleteNode(node: ListNode | null): void {
    if (node === null || node.next === null) {
        // If the node is null or it is the tail node, do nothing since deletion is not possible.
        return;
    }
    // Copy the value from the next node to the current node.
    node.val = node.next.val;
    // Bypass the next node by pointing the current node's `next` reference to the next node's `next`.
    node.next = node.next.next;
}


# Definition for a singly-linked list node.
class ListNode:
    def __init__(self, value):
        self.value = value  # Initialize node's value
        self.next = None    # Next node reference


class Solution:
    def deleteNode(self, node):
        """
        Delete a node from a singly-linked list, given only access to that node.

        :param node: The node to be deleted from the linked list.
        :type node: ListNode
        """
        node.value = node.next.value  # Copy the value from the next node to the current node
        node.next = node.next.next    # Bypass the next node by pointing to the node after next
```

Time and Space Complexity

The time complexity of the code is `O(1)`. This is because it takes a constant amount of time to copy the value from the next node to the given node and to update the next pointer of the given node. No loops or recursive calls are involved.

The space complexity of the code is `O(1)`. No extra space is used that is dependent on the size of the input. Only pointers are reassigned which does not require additional space.