2181. Merge Nodes in Between Zeros

Medium Linked List Simulation Leetcode Link

Problem Description

The task is to transform a given linked list, which has integer values with sequences of non-zero numbers separated by zeros. Important to consider is that the linked list both begins and ends with a `0`. The transformation process is to take every sequence of numbers between two `0`s, calculate their sum, and then create a new node with that sum. Essentially, you are merging all nodes between two `0`s into a single node with their sum as its value. The requirement is to do this for every sequence of numbers between `0`s within the linked list. In the end, the returned linked list should not contain any `0`s, except those demarcating the start and end of sequences.

Intuition

To solve this problem, we are walking through the linked list node by node while keeping a running sum of the values between `0`s. This process begins after skipping the initial `0` node at the head of the list. When we encounter a `0`, we know we've reached the end of a sequence, so we create a new node with the running sum and append it to the result list, and reset the sum to `0` to start for the next sequence. We use a dummy node to help us easily return the head of the new transformed list at the end of the process, and a tail node to keep track of the last node in our result list as we are appending new sum nodes.

The dummy and tail nodes initially point to the same dummy node. As we iterate through the original list, whenever we hit a `0`, we create a new node (with the current sum), link it to `tail.next`, and then update `tail` to point to the new node we just added. This continues until we have gone through the entire input list. In the end, `dummy.next` points to the head of the new list without the leading `0`, having all intermediate `0`s removed and replaced by the sum of values between them.

Solution Approach

The implementation uses a single pass through the linked list, utilizing a `dummy` node to simplify the final return and a `tail` node to keep track of the end of the new linked list being constructed. The algorithm underpinning this solution is a simple traversal of a singly-linked list paired with elementary accumulation of values.

Here's a step-by-step breakdown of the solution implementation:

1. Initialize a `dummy` node that will ultimately act as the predecessor of our new list. Initialize `tail` to also refer to this dummy node. This node is a sentinel to make appending to the list easier.
2. Initialize a running sum `s` to `0`. This will accumulate the values of the nodes between each pair of `0`s in the input list.
3. Skip the initial `0` of the input list by setting `cur` to `head.next`. We know the list starts with a `0`, so there's no need to include it in our sum.
4. Begin iterating through the list using a while loop that continues until `cur` is `None`, which will indicate we've reached the end of the list.
5. Inside the loop, check if the current node's value is not `0`. If it is not, add the value to the running sum `s`. If the node's value is `0`, it signifies we've reached the end of a sequence of numbers, and it's time to create a new node with the accumulated sum.
   - Create a new node with the sum `s` as its value.
   - Append the new node to the list by setting `tail.next` to this new node.
   - Update `tail` to refer to the new node, effectively moving our end-of-list marker to the right position.
   - Reset `s` back to `0` in preparation to start summing the next sequence.
6. Move `cur` forward in the list to process the next node.
7. The loop exits when we've examined every node. At that point, `dummy.next` will point to the head of our newly constructed list with summed values. Return `dummy.next` to adhere to the problem's requirement of returning the head of the modified linked list.

The critical data structure used here is the singly-linked list, and the patterns utilized include the runner technique (iterating over the list) and the sentinel node pattern (`dummy` node for easy return). The solution is efficient, running in O(n) time complexity with O(1) additional space complexity, where n is the number of nodes in the original linked list, as it involves only a single pass through the list and does not allocate any additional data structures that grow with the size of the input.

Example Walkthrough

Let's assume we have the following linked list:

```
0 -> 1 -> 2 -> 3 -> 0 -> 4 -> 5 -> 0
```

We want to transform it to:

```
0 -> 6 -> 9 -> 0
```

where each node between two `0`s now contains the sum of all original nodes in that range.

We start by creating a `dummy` node and making both `dummy` and `tail` point to it. We'll also initialize our running sum `s` to `0`. Our linked list initially looks like this (with `dummy` and `tail` pointing to the first `0`):

```
dummy -> 0
   |
tail
```

Our running sum `s` is `0`.

Step 1: We initialize our `dummy` and `tail` nodes.

```
dummy -> 0
   |
tail
```

Step 2: Initialize running sum `s` to `0`.

Step 3: Set `cur` to `head.next` to skip the initial `0`.

Step 4-6: We iterate through the list, updating the running sum `s` and creating new nodes as we encounter zeros. The process is as follows:

`cur` initially points to `1`. Since `cur.val` is not `0`, we add it to `s`.

```
s = 1
```

Move `cur` to the next node (`2`). Since `cur.val` is not `0`, we add it to `s`.

```
s = 3
```

Move `cur` to the next node (`3`). Since `cur.val` is not `0`, we add it to `s`.

```
s = 6
```

`cur` now points to `0`, so we've reached the end of a sequence. We create a new node with value `s` (which is `6`), attach it to `tail.next`, and update `tail` to point to the new node.

```
dummy -> 0 -> 6
              |
             tail
```

Reset `s` to `0` since we're starting a new sequence.

Move `cur` to the next node (`4`) and repeat the process. Add `4` to `s`.

```
s = 4
```

Move `cur` to `5`, add to `s`.

```
s = 9
```

`cur` points to `0` again, so end of a sequence. Create a new node with value `9`, attach it to `tail`, update `tail`, and reset `s`.

```
dummy -> 0 -> 6 -> 9
                   |
                  tail
```

Since there are no more nodes after this final `0`, we've finished iterating through the list.

Step 7: Our new list is ready. Return `dummy.next`, which is the head of the new transformed list:

```
0 -> 6 -> 9 -> 0
```

The resulting list starts and ends with `0`, and all nodes in between represent the sum of original sequences, with intermediate `0`s removed.

Python Solution

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

class Solution:
    def mergeNodes(self, head: Optional[ListNode]) -> Optional[ListNode]:
        # Create a dummy node which will be the placeholder for the new list's start.
        dummy_node = tail = ListNode()
        # Initialize the sum to keep track of the non-zero node values.
        node_sum = 0

        # Move the cursor to the next node as we want to skip the initial zero node.
        current_node = head.next

        while current_node:
            # If the current node's value is not zero, add it to the sum.
            if current_node.value != 0:
                node_sum += current_node.value
            else:
                # If a zero node is encountered, this signals the end
                # of the segment. We then add a new node with the
                # segment's sum to the resultant list.
                tail.next = ListNode(node_sum)
                # Move the tail to the new last node.
                tail = tail.next
                # Reset the sum for the next segment.
                node_sum = 0
            # Move to the next node in the original list.
            current_node = current_node.next

        # Return the new list without the initial dummy node.
        return dummy_node.next
```

Java Solution

```java
class Solution {
    public ListNode mergeNodes(ListNode head) {
        // Create a dummy node to act as the starting point of the new list
        ListNode dummyNode = new ListNode();
        // 'sum' will be used to calculate the sum of values between two zeros
        int sum = 0;
        // 'currentTail' refers to the current end of the new linked list
        ListNode currentTail = dummyNode;

        // Start processing nodes after the initial zero node
        for (ListNode currentNode = head.next; currentNode != null; currentNode = currentNode.next) {
            if (currentNode.val != 0) {
                // Add the current node's value to 'sum'
                sum += currentNode.val;
            } else {
                // If a zero node is found, create a new node with the sum and reset 'sum'
                currentTail.next = new ListNode(sum);
                // Move 'currentTail' to the new end node
                currentTail = currentTail.next;
                sum = 0; // Reset 'sum' as we started a new sum calculation
            }
        }

        // Return the next of dummyNode as it the head of the newly formed list
        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:
    /**
     * The function takes the head of a linked list where each node contains an integer.
     * It merges consecutive nodes with non-zero values by summing their values,
     * and separates sums with zero-valued nodes. It returns the new list without the initial zero node and trailing zeros.
     *
     * @param head The head of the provided linked list.
     * @return The head of the modified linked list.
     */
    ListNode* mergeNodes(ListNode* head) {
        // Dummy node preceding the result linked list
        ListNode* dummy = new ListNode();

        // Tail keeps track of the last node of the result list
        ListNode* tail = dummy;

        // Variable to accumulate values of nodes until a zero node encountered
        int sum = 0;

        // Iterate over nodes starting after the first (dummy) one
        for (ListNode* current = head->next; current != nullptr; current = current->next) {
            // If current value is not zero, add to sum
            if (current->val != 0) {
                sum += current->val;
            } else {
                // If a zero node is reached, add a new node with the accumulated sum to the result list
                tail->next = new ListNode(sum);
                tail = tail->next; // Move the tail forward
                sum = 0; // Reset sum for next segment
            }
        }
        // Return the head of the resulting list
        return dummy->next;
    }
};
```

Typescript Solution

```typescript
// Global function to merge nodes based on the sum between zeros in a linked list
function mergeNodes(head: ListNode | null): ListNode | null {
    // Create a dummy node to serve as the start of the new list
    const dummyNode: ListNode = new ListNode();
    // This will be the current node we are adding new sums to
    let currentNode: ListNode = dummyNode;
    // Initialize the sum
    let currentNodeSum: number = 0;

    // Iterate through the entire linked list
    while (head !== null) {
        // If the current value is zero and we have a non-zero sum
        if (head.val === 0 && currentNodeSum !== 0) {
            // Add new node to our list
            currentNode.next = new ListNode(currentNodeSum);
            // Move to the new node
            currentNode = currentNode.next;
            // Reset sum for the next set of values
            currentNodeSum = 0;
        }
        // Add the current value to our sum
        currentNodeSum += head.val;
        // Move to the next node in the list
        head = head.next;
    }

    // Return the next node as the dummy node is a placeholder
    return dummyNode.next;
}
```

Time and Space Complexity

The time complexity of the given code is `O(n)`, where `n` is the number of nodes in the linked list. This is because the code iterates through each node exactly once. Within each iteration, it performs a constant time operation of adding the node's value to the sum `s`, or creating a new node when a zero value is encountered. The construction of the new linked list is done in tandem with the traversal of the original list, ensuring that there are no additional passes required.

The space complexity of the code is `O(m)`, where `m` is the number of non-zero segments separated by zeros in the original linked list. This is due to the fact that a new list is being created where each node represents the sum of a non-zero segment. The space used by the new list is directly proportional to the number of these segments. No additional space is used except for the temporary variables `dummy`, `tail`, `s`, and `cur`, which do not depend on the size of the input list.