21. Merge Two Sorted Lists

Easy Recursion Linked List Leetcode Link

Problem Description

You are given two sorted linked lists `list1` and `list2`. Your task is to merge these two linked lists into a single sorted linked list. This new list should retain all the nodes from both `list1` and `list2`, and it must be sorted in ascending order. The final linked list should be constructed by connecting the nodes from the two lists directly, which is similar to splicing. The function should return the new list's head node.

Intuition

The intuition behind the given solution approach comes from the fact that both `list1` and `list2` are already sorted. The algorithm can take advantage of this by comparing the nodes from both lists and choosing the smaller one to be the next node in the merged list. This selection process is repeated until one of the lists is exhausted.

We start by creating a dummy node, which is a typical technique used in linked list operations where the head of the result list is unknown. The `dummy` node acts as a non-invasive placeholder to build our solution without knowing the head in advance; after the merge operation, the actual head of the merged list will be `dummy.next`.

A pointer named `curr` is assigned to keep track of the end of the already built part of the merged list; it starts at the `dummy` node. We use a `while` loop to iterate as long as there are elements in both `list1` and `list2`. During each iteration, we compare the values of the current heads of both lists and append the smaller node to the `next` of `curr`, moving `list1` or `list2` and `curr` forward.

When the loop ends because one of the lists is empty, we know that all the remaining elements in the non-empty list must be larger than all elements already merged because the lists were sorted initially. So, we can simply point the `next` of `curr` to the non-empty list to complete the merge. Finally, we return the actual start of the merged list, which is `dummy.next`.

Solution Approach

The solution is implemented using a simple iterative approach which traverses both input linked lists simultaneously, always taking the next smallest element to add it to the merged linked list. The main data structure used here is the singly-linked list. The algorithm employs the classic two-pointer technique to merge the lists.

Here are the step-by-step details of the algorithm:

1. A dummy node is created; this node does not hold any meaningful value but serves as the starting point of the merged linked list.
2. A `curr` pointer is initialized to point at the dummy node. This pointer moves along the new list as nodes are added.
3. A `while` loop continues as long as there are elements in both `list1` and `list2`. Inside the loop, a comparison is made:
   - If the value of the current node in `list1` is less than or equal to the value of the current node in `list2`, the `next` pointer of `curr` is linked to the current node of `list1`, and `list1` is advanced to its next node.
   - Otherwise, `curr next` is linked to the current node of `list2`, and `list2` is advanced to its next node.
4. After each iteration, `curr` is moved to its `next` node, effectively growing the merged list.
5. If one of the lists is exhausted before the other, the loop ends. Since one of the lists (either `list1` or `list2`) will be `None`, the `or` operator in Python ensures that `curr.next` will be linked to the remaining non-empty list.
6. Finally, the head of the merged list, which is immediately after the dummy node, is returned (`dummy.next`), thus omitting the dummy node which was just a placeholder.

This algorithm provides a clean and efficient way to traverse through two sorted lists, merging them without requiring additional space for the new list, as it reuses the existing nodes from both lists. It effectively adheres to the O(n) time complexity, where n is the total number of nodes in both lists combined, because each node is visited exactly once.

Example Walkthrough

Let's consider the following small example to illustrate the solution approach. Suppose we have two linked lists:

```
List1: 1 -> 3 -> 5
List2: 2 -> 4 -> 6
```

We want to merge `List1` and `List2` into a single sorted linked list.

1. First, we create a dummy node. It doesn't store any data but will serve as an anchor point for our new list.

2. We initialize a `curr` pointer to the dummy node. This `curr` will be used to keep track of the last node in our merged list.

3. Now we enter our `while` loop. Since both `List1` and `List2` have elements, we make comparisons:

   - We compare `List1's` head (1) with `List2's` head (2). Since 1 is less than 2, we link `curr` to `List1's` head and advance `List1` to the next element (which makes it 3 now).

4. The list starts to form: Dummy → 1. `Curr` now moves to 1.

5. In the next comparison, `List1's` value (3) is greater than `List2's` value (2). So we link `curr` to `List2's` head and advance `List2` to the next element (which makes it 4 now).

6. The list now is: Dummy → 1 → 2. `Curr` moves forward to 2.

7. As we continue this process, the next comparison has `List1's` value (3) less than `List2's` value (4), so we link `curr` to `List1's` head and move `List1` forward.

8. The list now looks like: Dummy → 1 → 2 → 3. `Curr` moves to 3.

9. This process of comparing and moving forward continues until we reach the end of one list. In this example, the merged list sequence goes: Dummy → 1 → 2 → 3 → 4 → 5 → 6, after exhaustively comparing:

   - 3 is less than 4
   - 5 is greater than 4
   - 5 is less than 6

10. Since we've reached the end of `List1` (there are no more elements to compare), we link `curr` to the remaining `List2` (which is just 6 now).

11. We've reached the end of both lists, and our merged list is: Dummy → 1 → 2 → 3 → 4 → 5 → 6.

12. Finally, we return `dummy.next` as the head of the new list, which omits the dummy node. So our final merged list is 1 → 2 → 3 → 4 → 5 → 6.

Throughout the traversal, we only moved forward, directly connecting the smaller node from either list to the merged list, until we completely traversed both lists and combined them into one sorted list.

Python Solution

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

class Solution:
    def mergeTwoLists(self, list1: ListNode, list2: ListNode) -> ListNode:
        # Creating a sentinel node which helps to easily return the head of the merged list
        sentinel = ListNode()

        # Current node is used to keep track of the end of the merged list
        current = sentinel

        # Iterate while both lists have nodes
        while list1 and list2:
            # Choose the smaller value from either list1 or list2
            if list1.val <= list2.val:
                current.next = list1   # Append list1 node to merged list
                list1 = list1.next     # Move to the next node in list1
            else:
                current.next = list2   # Append list2 node to merged list
                list2 = list2.next     # Move to the next node in list2

            # Move the current pointer forward in the merged list
            current = current.next

        # Add any remaining nodes from list1 or list2 to the merged list
        # If one list is fully traversed, append the rest of the other list
        current.next = list1 if list1 else list2

        # The sentinel node's next pointer points to the head of the merged list
        return sentinel.next

```

Java Solution

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

    ListNode() {}

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

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

public class Solution {

    /**
     * Merge two sorted linked lists and return it as a new sorted list.
     * The new list should be made by splicing together the nodes of the first two lists.
     *
     * @param list1 First sorted linked list.
     * @param list2 Second sorted linked list.
     * @return The head of the merged sorted linked list.
     */
    public ListNode mergeTwoLists(ListNode list1, ListNode list2) {
        // Initialize a dummy node to act as the head of the merged list.
        ListNode dummyHead = new ListNode();
        // This pointer will be used to add new elements to the merged list.
        ListNode current = dummyHead;

        // As long as both lists have elements, keep iterating.
        while (list1 != null && list2 != null) {
            if (list1.val <= list2.val) { // If list1's value is less or equal, add it next.
                current.next = list1;
                list1 = list1.next; // Move to the next element in list1.
            } else { // If list2's value is less, add it next.
                current.next = list2;
                list2 = list2.next; // Move to the next element in list2.
            }
            current = current.next; // Move forward in the merged list.
        }

        // In case one of the lists has remaining elements, link them to the end.
        current.next = (list1 == null) ? list2 : list1;

        // dummyHead.next points to the head of the merged list.
        return dummyHead.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:
    // Merges two sorted linked lists into one sorted linked list
    ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) {
        // Creates a dummy head node to facilitate easy return and manipulation of the merged list.
        ListNode* dummyHead = new ListNode();
        // Maintains the current node pointer in the merged list.
        ListNode* current = dummyHead;

        // Traverse both lists while both have elements.
        while (list1 && list2) {
            // Compare the current values from both lists to determine which node to take next.
            if (list1->val <= list2->val) {
                // Take the node from list1 and advance the list1 pointer.
                current->next = list1;
                list1 = list1->next;
            } else {
                // Take the node from list2 and advance the list2 pointer.
                current->next = list2;
                list2 = list2->next;
            }
            // Advance the pointer in the merged list.
            current = current->next;
        }

        // Append the non-empty remainder of the list to the merged list.
        // If list1 is not empty, append it; otherwise, append list2.
        current->next = list1 ? list1 : list2;

        // The dummy head's next points to the start of the merged list,
        // so return dummyHead->next.
        return dummyHead->next;
    }
};

```

Typescript Solution

```typescript
// Definition for singly-linked list node
interface ListNode {
    val: number;            // The value of the node
    next: ListNode | null;  // The reference to the next node
}

/**
 * Merge two sorted linked lists and return it as a new sorted list.
 * The new list should be made by splicing together the nodes of the first two lists.
 *
 * @param list1 - The head node of the first linked list.
 * @param list2 - The head node of the second linked list.
 * @returns The head node of the merged linked list.
 */
function mergeTwoLists(list1: ListNode | null, list2: ListNode | null): ListNode | null {
    // If one of the lists is null, return the other list since there's nothing to merge
    if (list1 === null || list2 === null) {
        return list1 || list2;
    }

    // Compare the values of the two list heads and recursively merge the rest of the lists
    if (list1.val < list2.val) {
        // If the value of the first list head is less,
        // link that node to the result of merging the rest of the lists
        list1.next = mergeTwoLists(list1.next, list2);
        return list1;
    } else {
        // If the value of the second list head is less or equal,
        // link that node to the result of merging the rest of the lists
        list2.next = mergeTwoLists(list1, list2.next);
        return list2;
    }
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(n + m)`, where `n` and `m` are the lengths of `list1` and `list2` respectively. This is because the while loop continues until we reach the end of one of the input lists and during each iteration of the loop, it processes one element from either `list1` or `list2`. Therefore, in the worst case, the loop runs for the combined length of `list1` and `list2`.

Space Complexity

The space complexity of the code is `O(1)`. While we are creating a `dummy` node and a `curr` pointer, the space they use does not scale with the input size, as they are just pointers used to form the new linked list. No additional data structures are used that depend on the input size, so the space used by the algorithm does not grow with the size of the input lists.