Merge Sorted Linked Lists

Problem Statement

The LeetCode problem "Merge Two Sorted Lists" involves merging two sorted linked lists into a single sorted linked list. In this scenario, you are provided with two sorted linked lists, and your objective is to combine them into one merged list that is sorted in non-decreasing order. The resulting list should maintain the order of elements from the original lists while ensuring that the final list is also sorted in ascending order.

Examples

To further illustrate the process of merging two sorted linked lists, let's consider a few examples showcasing different scenarios:

Example 1: Both Lists Non-Empty

Input Lists: List 1: 1 -> 3 -> 5
List 2: 2 -> 4 -> 6

Merged List: 1 -> 2 -> 3 -> 4 -> 5 -> 6

Example 2: One List Empty

Input Lists: List 1: 2 -> 4 -> 6
List 2: (empty)

Merged List: 2 -> 4 -> 6

Example 3: Both Lists Empty

Input Lists: List 1: (empty)
List 2: (empty)

Merged List: (empty)

These examples demonstrate the versatility of the solution in handling different cases with varying input scenarios. By following the prescribed approach of merging two sorted linked lists, the resulting merged list maintains the sorted order of elements from the original lists.

Constraints

When approaching the problem of merging two sorted linked lists, it is essential to consider certain constraints that govern the input and expected output. These

constraints play a crucial role in defining the boundaries within which the solution must operate effectively.

Constraints to Consider:

1. **Maximum Number of Nodes:** There may be a constraint on the maximum number of nodes that each linked list can contain. This limitation impacts the scalability and performance of the merging algorithm.
2. **Values Range:** Another significant constraint is the range within which the values of the nodes in the linked lists fall. Understanding this range helps in designing an efficient algorithm that can handle different value distributions.
3. **Input Assumptions:** It is important to specify any assumptions regarding the input data, such as the format in which the linked lists are provided and the presence of any null or empty lists.
4. **Edge Cases Handling:** Consideration of edge cases, such as when one or both input lists are empty, can influence the implementation strategy and ensure robustness in handling diverse scenarios.

By acknowledging and addressing these constraints upfront, developers can devise a more robust solution for merging two sorted linked lists while accounting for potential limitations and ensuring the algorithm's reliability across various input scenarios.

Explanation

To effectively merge two sorted linked lists into a single sorted list, a systematic approach can be followed to ensure the correct order of elements while optimizing time and space complexity. The general algorithmic steps for merging two sorted linked lists are as follows:

1. **Initialization:** Begin by creating a new linked list to store the merged result. Initialize pointers to the heads of the two input linked lists and a pointer to the current node in the merged list.
2. **Comparison and Merging:** While both input lists are not empty, compare the values of the nodes at the heads of the two lists. Append the smaller value to the merged list and move the corresponding pointer forward. Repeat this process until one of the input lists becomes empty.
3. **Appending Remaining Elements:** If one of the input lists still has remaining elements after the previous step, simply append the remaining nodes to the merged list since they are already sorted.
4. **Finalization:** Once all elements have been merged, return the head of the merged list as the final result.

This approach works effectively because it leverages the inherent sorted nature of the input lists to merge them efficiently into a single sorted list. By consistently selecting the

smallest element from the heads of the input lists, the algorithm ensures that the final merged list remains sorted in ascending order.

The time complexity of this merging algorithm is O(n), where n is the total number of elements in both input lists. The space complexity is also O(n) since a new linked list is created to store the merged result. This algorithm provides a balanced trade-off between efficiency and simplicity, making it a practical solution for merging two sorted linked lists.

Simplified Explanation

When we talk about merging two sorted linked lists, it's like organizing two decks of cards in ascending order. Imagine you have two decks of cards, and each card is a number. Your job is to combine these decks into one big deck, making sure the cards are in numerical order from smallest to largest.

Here's how you can do it step by step:

1. **Start Merging:** Take one card from each deck. Compare the numbers on the cards. Pick the smaller number and put it in a new deck. Repeat this process until you have used all the cards from both decks.
2. **Finishing Touches:** If one deck still has cards left after the first step, just add them directly to the new deck because they are already sorted.
3. **Voila! Final Result:** Your new deck is now a combination of the two original decks, with all the numbers in order from smallest to largest.

By following this simple method, you can merge two sorted linked lists efficiently. Think of it as sorting papers with numbers written on them into a neat pile from smallest to largest. This way, you can easily combine two lists without losing track of the order.

This merging process is like putting together puzzle pieces. You start with the corner pieces (the smallest numbers) and gradually fit the rest in place until you have a complete picture (a fully merged list).

This method ensures that the final merged list is sorted correctly without mixing up the numbers. It's like organizing a messy room by putting everything in its proper place, making it easier to find things later on.

Code

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

def mergeTwoLists(l1, l2):
    # Initialize a dummy node and a pointer to it
    dummy = ListNode(0)
```

```
    current = dummy

    # Traverse both lists while they are not empty
    while l1 and l2:
        # Compare values of the two lists
        if l1.val < l2.val:
            current.next = l1
            l1 = l1.next
        else:
            current.next = l2
            l2 = l2.next
        current = current.next

    # Append remaining nodes from either list
    current.next = l1 or l2

    # Return the merged list starting from the next node of the dummy node
    return dummy.next

# Example usage
# l1 = 1 -> 3 -> 5
# l2 = 2 -> 4 -> 6
# Merged list: 1 -> 2 -> 3 -> 4 -> 5 -> 6
```

This Python code snippet demonstrates the implementation of merging two sorted linked lists. By creating a ListNode class to represent the nodes of the linked list and defining a function mergeTwoLists to merge two input lists, the code follows a structured approach to combine the lists while maintaining the sorted order.

The mergeTwoLists function initializes a dummy node to facilitate the merging process. It then iterates through both input lists, comparing node values and appropriately linking them in ascending order in the merged list. Any remaining nodes from either list are appended at the end to ensure a complete merged list.

This well-commented code provides a clear understanding of how the merging algorithm operates and how it handles the sorting logic for the linked lists. By following this code structure, developers can efficiently merge two sorted linked lists in Python while ensuring the resulting list maintains the desired order.

Explanation of Each Line of Code

Let's break down the Python code snippet provided for merging two sorted linked lists and delve into the functionality of each line to understand how the merging process is implemented:

1. **class ListNode:**
    - This line defines a class named ListNode that represents a node in a singly-linked list. Each node has two attributes: val to store the node's value and next to reference the next node in the list.
2. **def __init__(self, val=0, next=None):**

- The __init__ method is the constructor for the ListNode class. It initializes a node with a default value of 0 and a default next node of None.

3. **self.val = val**
   - Assigns the provided value to the val attribute of the current node.
4. **self.next = next**
   - Assigns the provided next node to the next attribute of the current node.
5. **def mergeTwoLists(l1, l2):**
   - Defines a function named mergeTwoLists that takes two input parameters representing the heads of two sorted linked lists.
6. **dummy = ListNode(0)**
   - Creates a dummy node with a value of 0 to serve as the starting point for the merged list.
7. **current = dummy**
   - Initializes a pointer current to the dummy node, which will be used to build the merged list.
8. **while l1 and l2:**
   - Enters a while loop that continues as long as both input lists l1 and l2 are not empty.
9. **if l1.val < l2.val:**
   - Compares the values of the nodes at the heads of the two input lists.
10. **current.next = l1**
    - Links the current node to the node with the smaller value (l1) and moves the pointer in list l1 to the next node.
11. **current.next = l2**
    - Links the current node to the node with the smaller value (l2) and moves the pointer in list l2 to the next node.
12. **current = current.next**
    - Moves the current pointer to the next node in the merged list.
13. **current.next = l1 or l2**
    - Appends the remaining nodes from either list (l1 or l2) to the merged list.
14. **return dummy.next**
    - Returns the head of the merged list (excluding the dummy node).

This code efficiently merges two sorted linked lists by iterating through the input lists, comparing values, and linking nodes in ascending order to create a single merged list.

The use of a dummy node simplifies the merging process, ensuring a clean and sorted result.