82. Remove Duplicates from Sorted List II

Medium Linked List Two Pointers

Leetcode Link

Problem Description

In this problem, we are given the head of a linked list that is already sorted. Our task is to delete all nodes that have duplicate numbers, leaving only those numbers that appear exactly once in the original list. It is important to note that the returned linked list must be sorted as well, which is naturally maintained as we only remove duplicates from the already sorted list. For example, given `1->2->3->3->4->4->5`, the output should be `1->2->5` since `3` and `4` are duplicates and should be removed.

Intuition

To solve this problem, we aim to traverse the linked list and remove all the duplicates. One common approach to solve such problems is to use a two-pointer technique where one pointer is used to keep track of the current node of interest ('cur') and another to track the node before the current 'group' of duplicates ('pre').

By iterating through the linked list, we move the 'cur' pointer whenever we find that the current node has the same value as the next node (i.e., a duplicate). Once we're past any duplicates, we check if 'pre.next' is equal to 'cur', which would mean that 'cur' has no duplicates. If that's the case, we move 'pre' to 'cur'; otherwise, we set 'pre.next' to 'cur.next', effectively removing the duplicate nodes from the list.

This way, we traverse the list only once and maintain the sorted order of the non-duplicate elements. The use of a dummy node (which 'pre' points to initially) makes sure we also handle the case where the first element(s) of the list are duplicates and need to be removed.

Solution Approach

The implementation of the solution follows a simple yet effective approach to eliminate duplicates from a sorted linked list:

1. **Initialization:** We initialize a dummy node that points to the head of the list. This dummy node acts as a placeholder for the start of our new list without duplicates. We also create two pointers, `pre` and `cur`.
   - `pre` is initially set to the dummy node. It will eventually point to the last node in the list that has no duplicates.
   - `cur` is set to the head of the list and is used to iterate through and identify duplicates.
2. **Iteration:** We use a `while` loop to iterate over the list with the `cur` pointer until we reach the end of the list. Within this loop:
   - We use another `while` loop to skip all the consecutive nodes with the same value. This is done by checking if `cur.next` exists and if `cur.next.val` is the same as `cur.val`. If so, we move `cur` to `cur.next`.
   - After skipping all nodes with the same value, we check whether `pre.next` is the same as `cur`. If it is, this means `cur` is distinct, and thus we simply move `pre` to be `cur`. If not, it means we have skipped some duplicates, so we remove them by linking `pre.next` to `cur.next`.
3. **Link Adjustment:** If duplicates were removed, `pre.next` is assigned to `cur.next`. This effectively removes all the duplicates we have just skipped over because we link the last non-duplicate node to the node right after the last duplicate node.
4. **Move to Next Node:** We move `cur` to `cur.next` to proceed with the next group of nodes.
5. **Result:** After the loop finishes, `dummy.next` points to the head of the final list with all duplicates removed. Since we maintained the relative order of the non-duplicate elements and the list was initially sorted, the result is a sorted linked list with all duplicates removed.

By following this approach, using a dummy node, two pointers, and careful link adjustments as we iterate through the list, we ensure:

- The original sorted order of the list is preserved.
- All duplicates are removed in one pass, making the solution efficient.

The solution exploits the sorted property of the input list, allowing us to detect duplicates easily by comparing adjacent nodes. The use of a dummy node allows for a uniform deletion process, including the edge case where the head of the list contains duplicates and must be deleted.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Consider the sorted linked list: `1->2->2->3`.

1. **Initialization:**
   - Create a dummy node. Let's say `dummy.val` is `0`, and `dummy.next` points to the head of our list, so now we have `0->1->2->2->3`.
   - Set the `pre` pointer to `dummy` and `cur` pointer to `head` (which is `1`).
2. **Iteration:**
   - The list is now `0->1->2->2->3` with `pre` at `0` and `cur` at `1`.
   - `cur.next` is `2`. Since `cur.val` is not equal to `cur.next.val`, we move `pre` to `cur`, and `cur` to `cur.next`.
   - Now `pre` is at `1` and `cur` is at the first `2`.
3. **Identifying Duplicates:**
   - `cur.next` is also `2`. Since `cur.val` equals `cur.next.val`, we move `cur` to `cur.next`. Now, `cur` is at the second `2`.
   - After the inner loop, `cur.next` is `3`, which is not equal to `cur.val`. Since `pre.next` (which is the first `2`) is not equal to `cur` (which is the second `2`), we found duplicates.
4. **Link Adjustment:**
   - We link `pre.next` to `cur.next`, effectively removing both `2`s. The list now becomes `0->1->3`.
5. **Move to Next Node:**
   - Move `cur` to `cur.next`, which is `3`.
   - `cur.next` is `null` now, ending the iteration.
6. **Result:**
   - The final list is pointed to by `dummy.next`, which is `1->3`, with all duplicates (`2->2`) removed.

By applying this approach, we have effectively removed all nodes with duplicate numbers from our sorted linked list by only traversing the list once, and the sorted property of the list remained intact.

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 deleteDuplicates(self, head: Optional[ListNode]) -> Optional[ListNode]:
        # Create a dummy node which acts as the new head of the list
        dummy = prev_node = ListNode(next_node=head)

        # Initialize the current pointer to the head of the list
        current = head

        # Iterate through the list
        while current:
            # Skip all duplicate nodes
            while current.next and current.next.val == current.val:
                current = current.next


            # Check if duplicates were found
            # If prev_node.next is the same as current, no duplicates were immediately following
            # If not, duplicates were found and they are skipped
            if prev_node.next == current:
                prev_node = current
            else:
                prev_node.next = current.next

            # Move to the next node in the list
            current = current.next

        # After filtering duplicates, return the next node of dummy since it stands before the new head of list
        return dummy.next

```

Java Solution

```java
class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        // Dummy node to act as a fake head of the list
        ListNode dummyNode = new ListNode(0, head);

        // This pointer will lag behind the current pointer and point to the last
        // node of the processed list without duplicates
        ListNode precedingNode = dummyNode;

        // This pointer will traverse the original list
        ListNode currentNode = head;

        // Iterate through the list
        while (currentNode != null) {
            // Skip all nodes that have the same value as the current node
            while (currentNode.next != null && currentNode.next.val == currentNode.val) {
                currentNode = currentNode.next;
            }

            // If there are no duplicates for the current value,
            // then the precedingNode should now point to the currentNode
            if (precedingNode.next == currentNode) {
                precedingNode = currentNode;
            } else {
                // Otherwise, bypass all duplicates by linking precedingNode to the node
                // following the last duplicate
                precedingNode.next = currentNode.next;
            }
            // Move to the next node in the list
            currentNode = currentNode.next;
        }

        // Return the processed list without duplicates
        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:
    ListNode* deleteDuplicates(ListNode* head) {
        // Create a dummy node that points to the head of the list
        ListNode* dummyNode = new ListNode(0, head);
        ListNode* pred = dummyNode; // Predecessor node that trails behind the current node
        ListNode* curr = head; // Current node we're examining

        // Traverse the list
        while (curr) {
            // Skip duplicate nodes
            while (curr->next && curr->next->val == curr->val) {
                curr = curr->next;
            }
            // If the predecessor's next is the current node, no duplicates were found
            // Move the predecessor to the current node
            if (pred->next == curr) {
                pred = curr;
            } else {
                // Skip the duplicate nodes by linking the predecessor's next to the current's next
                pred->next = curr->next;
            }
            // Move to the next node
            curr = curr->next;
        }
        // Return the modified list, excluding the dummy node
        return dummyNode->next;
    }
};
```

Typescript Solution

```typescript
// Type definition for ListNode
type ListNode = {
  val: number;
  next: ListNode | null;
};

// Function to create a new ListNode
const createListNode = (val: number, next: ListNode | null = null): ListNode => {
  return { val, next };
};

/**
 * Deletes all duplicates such that each element appears only once
 *
 * @param {ListNode | null} head - The head of the input linked list
 * @return {ListNode | null} The modified list with duplicates removed
 */
const deleteDuplicates = (head: ListNode | null): ListNode | null => {
  // Create a dummy head to ease the deletion process by avoiding edge cases at the list start
  const dummy: ListNode = createListNode(0, head);

  // Pointers for the previous and current nodes, starting with dummy and head
  let previous: ListNode = dummy;
  let current: ListNode | null = head;

  // Iterate through the list
  while (current) {
    // Skip all nodes that have the same value as the current node
    while (current.next && current.val === current.next.val) {
      current = current.next;
    }

    // If the next node of previous is current, we did not find duplicates
    // Move pre to cur
    if (previous.next === current) {
      previous = current;
    } else {
      // If there were duplicates, skip them by pointing pre.next to cur.next
      previous.next = current.next;
    }

    // Move to the next node
    current = current.next;
  }

  // Return the modified list without the initial dummy node
  return dummy.next;
};

// Note: To execute this code, a ListNode type declaration as well as createListNode()
// function must exist in the global scope, The deleteDuplicates() function will also
// be globally accessible.
```

Time and Space Complexity

The time complexity of the provided code to delete duplicates in a sorted linked list 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, and while there can be inner loops to skip duplicate values, each node is visited only once due to the fact that the code moves the pointer `cur` forward to skip all duplicates of the current node in one go.

The space complexity of the code is $O(1)$. It uses a fixed amount of extra space: the `dummy` and `pre` variables, which do not depend on the size of the input linked list. No additional data structures are utilized that would scale with the input size, hence the constant space complexity.