203. Remove Linked List Elements

Easy Recursion Linked List

Problem Description

The problem requires us to modify a linked list by removing all nodes that contain a specified value. Given the head of a linked list and an integer `val`, we must iterate through the linked list and delete any node where the `Node.val` is equal to `val`. The function should then return the head of the modified linked list, which might be different from the initial head if the head node itself needed to be removed.

Intuition

To solve this problem, we need to iterate through the linked list and keep track of the previous node (`pre`) so we can modify its `next` pointer when we find a node that needs to be removed. The challenge is that the nodes to be removed can be anywhere in the list, including the head. To handle the case where the head itself needs to be removed, we introduce a dummy node with an arbitrary value that points to the head of the linked list. This dummy node acts as a placeholder that makes it easier to handle edge cases, particularly when we need to remove the head node.

Our approach involves iterating over the linked list with a pointer `pre` initialized to the dummy node. For each node, we check if the `next` node has a value equal to `val`. If it does not, we move `pre` to the next node. If it does, instead of moving to the next node, we change the `next` pointer of `pre` to skip over the node with the `val` we want to remove, effectively removing it from the linked list. We continue this process until we have checked all nodes. Finally, we return the `next` node of the dummy, which represents the new head of the modified linked list.

Solution Approach

The implementation begins with creating a dummy node. This dummy node is a commonly used pattern when dealing with linked list modifications, as it simplifies edge case handling when the head of the list might be removed. We can say that the dummy node acts as a fake head (or a sentinel) to the list. The dummy node is linked to the actual head of the list.

After initializing the dummy node, we introduce a pointer `pre` that is used to traverse the list. This pointer starts at the dummy node.

The core of the solution is a `while` loop that continues until `pre.next` is `None` (meaning we have reached the end of the list). Inside the loop, the condition `if pre.next.val != val` is checked. If this condition is true, it means the current `pre.next` node does not need to be removed, and we can safely move `pre` to point to this node (`pre = pre.next`). If the condition is false, we have encountered a node with the value `val`, which must be removed. To remove this node, we alter `pre.next` to skip over the current `pre.next` node and point to `pre.next.next`. This effectively removes the unwanted node from the list.

After the loop terminates, we return `dummy.next`, which is the new head of the modified list. This works even if the original head was removed because `dummy.next` will now point to the first node with a value different from `val`.

The aforementioned approach leverages basic linked list traversal and node skipping techniques. The dummy node pattern helps to avoid special case handling for the head of the list. This pattern is also known for its utility in simplifying the code and making the algorithm cleaner and easy to understand.

The overall time complexity of the solution is `O(n)`, where `n` is the number of nodes in the list, since we may have to traverse the entire list in the worst case. The space complexity is `O(1)` as we only use a fixed amount of extra space.

Example Walkthrough

Let's assume we have a linked list `1 -> 2 -> 6 -> 3 -> 4 -> 5 -> 6` and we want to remove all nodes with the value `6`.

Here's a step-by-step illustration:

1. Create a dummy node with an arbitrary value (can be anything as it will not be used) and point its `next` to the head of the list. So, now we have **dummy** -> 1 -> 2 -> 6 -> 3 -> 4 -> 5 -> 6.
2. Initialize a pointer `pre` to point at the dummy node. So, `pre` is now pointing to **dummy**.
3. Begin iterating over the list. Since `pre.next` (the first `1` node) does not have the value `6`, we move `pre` to point to the `1` node.
4. Continue to the next node with value `2`. The value `2` is not `6` either, so we move `pre` to this node.
5. The next node has the value `6`, which is the value we want to remove. We change `pre.next` to skip this `6` node and point to its next node, which is `3`. So now `pre` (which points to `2`) has its `next` pointing to `3`. The list now effectively looks like this: **dummy** -> 1 -> 2 -> 3 -> 4 -> 5 -> 6.
6. Move to the next nodes with values `3`, `4`, and `5` in the same manner since their values are not `6`.
7. The next node with value `6` is again the node we want to remove. We change `pre.next` to skip this node and point to the next, which is `null`. The modified list now looks like this: **dummy** -> 1 -> 2 -> 3 -> 4 -> 5 -> `null`.
8. Since we've reached the end of the list (`pre.next` is `null`), we stop iterating.
9. Finally, we return the node right after our dummy, which is the new head of the modified list. The returned list is `1 -> 2 -> 3 -> 4 -> 5`, as all `6` nodes have been removed.

Therefore, by using a dummy node and the pointer `pre`, we managed to remove all the nodes with the value `6` without having to deal with special cases like the head of the list being removed. This demonstrates the elegance and effectiveness of the solution approach.

Solution Implementation

Python

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

class Solution:
    def removeElements(self, head: ListNode, val: int) -> ListNode:
        # Create a dummy node that acts as the new head of the list with a sentinel value
        dummy_node = ListNode(-1)
        dummy_node.next = head

        # Initialize a pointer to track the current node we're examining
        current_node = dummy_node

        # Iterate over the list
        while current_node.next is not None:
            # If the value of the next node is the one we want to remove
            if current_node.next.val == val:
                # Remove it by skipping the node
                current_node.next = current_node.next.next
            else:
                # Move to the next node otherwise
                current_node = current_node.next

        # Return the modified list, starting from the node after the dummy node
        return dummy_node.next
```

Java

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

    ListNode() {}

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

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

class Solution {
    public ListNode removeElements(ListNode head, int val) {
        // Create a dummy head to simplify edge cases such as removing the first element
        ListNode dummyHead = new ListNode(-1);
        dummyHead.next = head;

        // Initialize the previous node as the dummy node
        ListNode previousNode = dummyHead;

        // Iterate through the list
        while (previousNode.next != null) {
            // Check if the current node has the value we need to remove
            if (previousNode.next.val == val) {
                // Remove the current node from the list
                previousNode.next = previousNode.next.next;
            } else {
                // Move to the next node
                previousNode = previousNode.next;
            }
        }

        // Return the new head of the list, which is the next of dummy node
        return dummyHead.next;
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to remove all nodes with a specific value from a linked list.
    ListNode* removeElements(ListNode* head, int val) {
        // Create a dummy node that serves the purpose of an anchor for the new list without the given value.
        ListNode* dummyNode = new ListNode(0);
        dummyNode->next = head;

        // Initialize a pointer that will traverse the list, starting from the dummy node.
        ListNode* current = dummyNode;

        // Iterate through the list until the end is reached.
        while (current->next) {
            // Check if the next node's value matches the value to be removed.
            if (current->next->val == val) {
                // If a match is found, skip over the node with the matching value.
                current->next = current->next->next;
            } else {
                // Otherwise, move to the next node.
                current = current->next;
            }
        }

        // Return the next node of dummy as it is the new head of the modified list.
        return dummyNode->next;
    }
};
```

TypeScript

```typescript
// Function to remove elements from a linked list that have a specific value.
// head: The first node of the linked list.
// val: The value to remove from the linked list.
// Returns the new head of the linked list after removals.
function removeElements(head: ListNode | null, val: number): ListNode | null {
    // Initialize a dummy node with the next pointer pointing to the head of the list.
    // This simplifies edge cases such as trying to delete the head of the list.
    const dummy: ListNode = new ListNode(0, head);

    // This points to the current node we're examining, starting from the dummy.
    let current: ListNode = dummy;

    // Iterate through the list as long as the next node exists.
    while (current.next !== null) {
        // If the value of the next node is the target value, bypass the next node.
        if (current.next.val === val) {
            current.next = current.next.next; // Skip the node to remove it.
        } else {
            current = current.next; // Move to the next node if it's not to be removed.
        }
    }

    // Return the next node of the dummy, which represents the new head after removals.
    return dummy.next;
}
```

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

class Solution:
    def removeElements(self, head: ListNode, val: int) -> ListNode:
        # Create a dummy node that acts as the new head of the list with a sentinel value
        dummy_node = ListNode(-1)
        dummy_node.next = head

        # Initialize a pointer to track the current node we're examining
        current_node = dummy_node

        # Iterate over the list
        while current_node.next is not None:
            # If the value of the next node is the one we want to remove
            if current_node.next.val == val:
                # Remove it by skipping the node
                current_node.next = current_node.next.next
            else:
                # Move to the next node otherwise
                current_node = current_node.next

        # Return the modified list, starting from the node after the dummy node
        return dummy_node.next
```

Time and Space Complexity

Time Complexity

The time complexity of the provided code is `O(n)`, where `n` is the number of elements in the linked list. The while loop iterates through each node in the linked list a single time, checking if the node's value matches the target value `val` and removing it if necessary.

Space Complexity

The space complexity of the provided code is `O(1)`. This is because the code only uses a constant amount of extra space: one dummy node (`dummy`) and one pointer (`pre`). The removal of elements is done in-place, not requiring any additional data structures that would scale with the size of the input.