206. Reverse Linked List

Easy Recursion Linked List

Problem Description

The task is to reverse a singly linked list. A linked list is a data structure where each element (often called a 'node') contains a value and a pointer/reference to the next node in the sequence. A singly linked list means that each node points to the next node and there is no reference to previous nodes. The problem provides a pointer to the head of the linked list, where the 'head' represents the first node in the list. Our goal is to take this linked list and return it in the reversed order. For instance, if the linked list is `1 -> 2 -> 3 -> null`, the reversed list should be `3 -> 2 -> 1 -> null`.

Intuition

To reverse the linked list, we iterate over the original list and rearrange the `next` pointers without creating a new list. The intuition behind this solution is to take each node and move it to the beginning of the new reversed list as we traverse through the original list. We maintain a temporary node, often referred to as a 'dummy' node, which initially points to `null`, as it will eventually become the tail of the reversed list once all nodes are reversed.

We iterate from the head towards the end of the list, and with each iteration, we do the following:

- Temporarily store the next node (since we are going to disrupt the `next` reference of the current node).
- Set the `next` reference of the current node to point to what is currently the first node of the reversed list (initially, this is `null` or `dummy.next`).
- Move the dummy's next reference to the current node, effectively placing the current node at the beginning of the reversed list.
- Move to the next node in the original list using the reference we stored earlier.

This process ensures that we do not lose track of the remaining parts of the original list while building the reversed list. After we have iterated through the entire original list, the `dummy.next` will point to the new head of the reversed list, which we then return as the result.

Solution Approach

The provided solution employs an iterative approach to go through each node in the linked list and reverse the links. Here's a step-by-step walk-through of the algorithm used:

1. A new `ListNode` called `dummy` is created, which acts as the placeholder before the new reversed list's head.
2. A pointer called `curr` is initialized to point to the `head` of the original list. This pointer is used to iterate over the list.
3. The iteration starts with a `while` loop which continues as long as `curr` is not `null`. This ensures we process all nodes in the list.
4. Inside the loop, `next` temporarily stores `curr.next`, which is the pointer to the next node in the original list. This is crucial since we are going to change `curr.next` to point to the new list and we don't want to lose the reference to the rest of the original list.
5. We then set `curr.next` to point to `dummy.next`. Since `dummy.next` represents the start of the new list, or `null` in the first iteration, the current node now points to the head of the reversed list.
6. `dummy.next` is updated to `curr` to move the starting point of the reversed list to the current node. At this point, `curr` is effectively inserted at the beginning of the new reversed list.
7. `curr` is updated to `next` to move to the next node in the original list, using the pointer we saved earlier.
8. Once all nodes have been processed and the loop exits, `dummy.next` will be the head of the new reversed list.
9. The new reversed list referenced by `dummy.next` is returned.

By updating the `next` pointers of each node, the solution reverses the direction of the list without allocating any additional nodes, which makes it an in-place reversal with a space complexity of O(1). Each node is visited once, resulting in a time complexity of O(n), where n is the number of nodes in the list.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we have the following linked list:

```
1 -> 2 -> 3 -> null
```

We want to reverse it to become:

```
3 -> 2 -> 1 -> null
```

Here's the step-by-step process to achieve that using the provided algorithm:

1. We create a `ListNode` called `dummy` that will initially serve as a placeholder for the reversed list. At the beginning, `dummy.next` is set to `null`.
2. We initialize a pointer `curr` to point to the head of the original list which is the node with the value `1`.

```
dummy -> null
curr -> 1 -> 2 -> 3 -> null
```

3. Starting the iteration, we enter the `while` loop since `curr` is not `null`.
4. We store `curr.next` in `next`, so `next` points to `2`. `next` will help us move forward in the list after we've altered `curr.next`.

```
next -> 2 -> 3 -> null
```

5. We update `curr.next` to point to `dummy.next`, which is currently `null`. Now the first node (`1`) points to `null`, the start of our new reversed list.

```
dummy -> null <- 1     2 -> 3 -> null
curr ----------^      next ----^
```

6. We move the start of the reversed list to `curr` by setting `dummy.next` to `curr`. The reversed list now starts with `1`.

```
dummy -> 1 -> null
         ^
curr ----|
```

7. We update `curr` to `next`, moving forward in the original list. `curr` now points to `2`.

```
dummy -> 1 -> null
curr -> 2 -> 3 -> null
```

8. The loop continues. Again, we save `curr.next` to `next`, and update `curr.next` to point to `dummy.next`. Then we shift the start of the reversed list by setting `dummy.next` to the current node and update `curr` to `next`. After this iteration, `dummy` points to the new head `2`, and our reversed list grows:

```
dummy -> 2 -> 1 -> null

curr ----|    3 -> null
       next --^
```

9. In the final iteration, we perform similar steps. We save `curr.next` to `next`, set `curr.next` to `dummy.next`, and move `dummy.next` to `curr`. `curr` is then updated to the `null` we saved in `next`:

```
dummy -> 3 -> 2 -> 1 -> null

curr ----|
```

10. Once `curr` is `null`, the `while` loop terminates, and we find that `dummy.next` points to `3`, which is the new head of the reversed list.
11. Lastly, we return the reversed list starting from `dummy.next`, which is `3 -> 2 -> 1 -> null`.

And that completes the reversal of our linked list using the iterative approach described in the solution.

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 reverseList(self, head: ListNode) -> ListNode:
        # Initialize a dummy node, which will be the new head after reversal
        dummy_node = ListNode()

        # Start from the head of the list
        current_node = head

        # Iterate over the linked list
        while current_node is not None:
            # Save the next node
            next_node = current_node.next

            # Reverse the link so that current_node.next points to the node before it
            current_node.next = dummy_node.next
            dummy_node.next = current_node

            # Move to the next node in the original list
            current_node = next_node

        # The dummy node's next now points to the head of the reversed list
        return dummy_node.next
```

Java

```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; }
}

class Solution {

    /**
     * Reverses the given linked list.
     *
     * @param head The head of the original singly-linked list.
     * @return The head of the reversed singly-linked list.
     */
    public ListNode reverseList(ListNode head) {
        // Dummy node that will help in reversing the list.
        ListNode dummy = new ListNode();

        // Pointer to traverse the original list.
        ListNode current = head;

        // Iterating through each node in the list.
        while (current != null) {
            // Temporary store the next node.
            ListNode nextTemp = current.next;

            // Reversing the link so that current.next points to the new head (dummy.next).
            current.next = dummy.next;

            // Move the dummy's next to the current node making it the new head of the reversed list.
            dummy.next = current;

            // Move to the next node in the original list.
            current = nextTemp;
        }

        // Return the reversed linked list which is pointed by dummy's next.
        return dummy.next;
    }
}
```

C++

```cpp
// Definition for singly-linked list node.
struct ListNode {
    int val;        // The value of the node.
    ListNode *next;  // Pointer to the next node in the list.

    // Default constructor initializes with default values.
    ListNode() : val(0), next(nullptr) {}

    // Constructor initializes with a given value and next pointer set to nullptr.
    ListNode(int x) : val(x), next(nullptr) {}

    // Constructor initializes with a given value and a given next node pointer.
    ListNode(int x, ListNode *next) : val(x), next(next) {}
};

class Solution {
public:
    // Function to reverse a singly-linked list.
    ListNode* reverseList(ListNode* head) {
        // The 'dummy' node acts as the new head of the reversed list.
        ListNode* dummy = new ListNode();

        // 'current' node will traverse the original list.
        ListNode* current = head;

        // Iterate through the list until we reach the end.
        while (current != nullptr) {
            // 'nextNode' temporarily stores the next node.
            ListNode* nextNode = current->next;

            // Reverse the 'current' node's pointer to point to the new list.
            current->next = dummy->next;

            // The 'current' node is prepended to the new list.
            dummy->next = current;

            // Move to the next node in the original list.
            current = nextNode;
        }

        // The head of the new reversed list is 'dummy->next.'
        return dummy->next;
    }
};
```

TypeScript

```typescript
// Definition for a node in a singly-linked list
interface ListNode {
    val: number;
    next: ListNode | null;
}

/**
 * Reverses a singly linked list.
 * @param {ListNode | null} head - The head node of the linked list to be reversed
 * @return {ListNode | null} The new head of the reversed linked list
 */
function reverseList(head: ListNode | null): ListNode | null {
    // Return immediately if the list is empty
    if (head === null) {
        return head;
    }

    // Initialize pointers
    let previousNode: ListNode | null = null; // Previous node in the list
    let currentNode: ListNode | null = head;  // Current node in the list

    // Iterate through the list
    while (currentNode !== null) {
        const nextNode: ListNode | null = currentNode.next; // Next node in the list

        // Reverse the current node's pointer
        currentNode.next = previousNode;

        // Move the previous and current pointers one step forward
        previousNode = currentNode;
        currentNode = nextNode;
    }

    // By the end, previousNode is the new head of the reversed linked list
    return previousNode;
}
```

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

class Solution:
    def reverseList(self, head: ListNode) -> ListNode:
        # Initialize a dummy node, which will be the new head after reversal
        dummy_node = ListNode()

        # Start from the head of the list
        current_node = head

        # Iterate over the linked list
        while current_node is not None:
            # Save the next node
            next_node = current_node.next

            # Reverse the link so that current node.next points to the node before it
            current_node.next = dummy_node.next
            dummy_node.next = current_node

            # Move to the next node in the original list
            current_node = next_node

        # The dummy node's next now points to the head of the reversed list
        return dummy_node.next
```

Time and Space Complexity

The time complexity of the provided code is `O(n)`, where `n` is the number of nodes in the linked list. This is because the code iterates through all the nodes in the list a single time.

The space complexity of the code is `O(1)`. The space used does not depend on the size of the input list, since only a finite number of pointers (`dummy`, `curr`, `next`) are used, which occupy constant space.