143. Reorder List

Medium | Stack | Recursion | Linked List | Two Pointers | Leetcode Link

Problem Description

The problem provides a singly linked list with nodes labeled from `L0` to `Ln`. The task is to reorder the list in a specific manner without changing the node values but only by rearranging the nodes. The reordered list should follow a pattern where the first node is followed by the last node, then the second node is followed by the second to last node, and this pattern continues until all nodes have been reordered. This results in a list that starts at the head, alternates nodes from the start and end of the list, and meets in the middle.

Intuition

The solution can be broken down into several logical steps:

1. **Finding the middle of the list**: We use the fast and slow pointer technique to find the middle of the linked list. The slow pointer moves one step at a time, while the fast pointer moves two steps at a time. When the fast pointer reaches the end, the slow pointer will be at the middle of the list.

2. **Reversing the second half of the list**: Once we have the middle of the list, we reverse the second half. This is done iteratively by initializing pointers and rearranging the links between nodes in the second half of the list.

3. **Merging the two halves**: With the second half reversed, we now have two lists: the first half in the original order and the second half in reverse order. We merge the two lists by alternating nodes from each, starting with the first node of the first half and inserting the first node of the second half after it, followed by the second node of the first half, and so on, until all the nodes from the second half have been inserted into the first half.

The provided solution follows these steps to achieve the desired list reordering.

Solution Approach

The implementation of the solution can be outlined in the below steps corresponding to the intuition described earlier:

1. **Using Two Pointers to Find the Middle**: We initialize two pointers, both starting at the head of the list. The `slow` pointer advances one node at a time, while the `fast` pointer advances two nodes at a time. When the `fast` pointer either reaches the end of the list or the node before the end, the `slow` pointer will be at the middle of the list. The loop `while fast.next and fast.next.next:` ensures we stop at the correct position in the list.

2. **Reversing the Second Half**: In order to reverse the second half of the list, we first set `slow.next` to `None` to mark the end of the first half of the list. We then use three pointer variables (`cur`, `pre`, and `t`) to reverse the second half of the list. `cur` starts at the first node of the second half, while `pre` is set to `None` to mark the new end of the list. We then iterate through the second half using `while cur:`, in each iteration, we temporarily store the next node using `t = cur.next`, point the current node to `pre`, and then move `pre` and `cur` forward.

3. **Merging the Two Halves**: We now have two lists: the first half, starting at `head`, and the second half, starting at `pre`, which is the reverse of the original second half. We merge these two half-lists by iterating through them, taking one node from each list and adjusting the pointers to merge them into a single list. The loop `while pre:` allows us to do just that. During each iteration, we store the next node of `pre` in `t`, then we link `pre` to the next of the current node in the first list (`cur.next`). After updating `cur.next` to `pre`, we advance `cur` and `pre` using the stored values.

The final result of these steps is a reordered list in the desired pattern: `L0 -> Ln -> L1 -> Ln-1 -> L2 -> Ln-2 -> ...` until all nodes are repositioned accordingly.

Example Walkthrough

Consider a linked list with nodes L0 → L1 → L2 → L3 → L4. We want to reorder this list following the specific pattern described in the problem: L0 → L4 → L1 → L3 → L2.

Step 1: Finding the Middle of the List We use two pointers, `slow` and `fast`. Initially, both pointers start at L0. As we iterate through the list:

- In the first step, `slow` is at L0, and `fast` is at L1.
- In the second step, `slow` is at L1, and `fast` is at L3.
- In the third step, `slow` is at L2, and `fast` is at the end (null), so `slow` is now at the middle of the list.

The list is now considered in two parts: the first half is L0 → L1 → L2, and the second half, starting at L3, needs to be reversed.

Step 2: Reversing the Second Half Starting from L3, we reverse the second half of the list. We set `slow.next` to `None` to mark the end of the first half to get L0 → L1 → L2 and L3 → L4.

- `cur` begins at L3, and `pre` is `None`.
- We swap the next of `cur` (which is L4) to point to `pre` and advance `pre` to be L3 and `cur` to be L4.
- Now `cur` is L4, and we point L4 to the new `pre` (which is L3), making `pre` equal to L4 and `cur` to `None`.

After completing this process, the second half is reversed, and our lists look like this: L0 → L1 → L2 and L4 → L3.

Step 3: Merging the Two Halves We have two sublists, and now we merge them in the alternate sequence:

- Start with `head` at L0 and `pre` at L4.
- Save the next of `head` (L1) and link `head` to `pre` (L4). List after this step: L0 → L4.
- Advance `head` to saved next (L1) and `pre` to next in the reversed list (L3).
- Save next of `head` again (L2) and link `head` to `pre` (L3). List after this step: L0 → L4 → L1 → L3.
- Advance `head` to saved next (L2), and since `pre` has no next (null), we've finished merging.

The final reordered list is L0 → L4 → L1 → L3 → L2, which matches the required pattern.

Python Solution

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

class Solution:
    def reorderList(self, head: ListNode) -> None:
        """
        This function takes the head of a singly linked list and reorders it in-place
        so that the nodes are in a specific sequence: L0 → Ln → L1 → Ln - 1 → L2 → Ln - 2 → …
        You must do this without altering the values in the list's nodes, i.e., only nodes themselves may be changed.
        """

        # Use the fast and slow pointer technique to find the middle of the linked list
        fast = slow = head
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next

        # Split the linked list into two halves
        second_half = slow.next
        slow.next = None

        # Reverse the second half of the linked list
        previous = None
        current = second_half
        while current:
            temp = current.next
            current.next = previous
            previous, current = current, temp

        # Merge the two halves, inserting nodes from the second half into the first
        first_half = head
        second_half = previous # This is now the head of the reversed second half.
        while second_half:
            temp1 = first_half.next
            temp2 = second_half.next

            first_half.next = second_half
            second_half.next = temp1

            first_half, second_half = temp1, temp2

        # The linked list is now re-ordered in the required pattern

```

Java Solution

```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 void reorderList(ListNode head) {
        // Step 1: Use two-pointers to find the middle of the linked list
        ListNode fastPointer = head, slowPointer = head;
        // fastPointer moves twice as fast as the slowPointer
        while (fastPointer.next != null && fastPointer.next.next != null) {
            slowPointer = slowPointer.next;
            fastPointer = fastPointer.next.next;
        }

        // Step 2: Split the list into two and reverse the second half
        // Now, slowPointer is at the middle of the list
        ListNode current = slowPointer.next; // This is the start of the second half
        slowPointer.next = null; // Split the list into two

        ListNode previous = null;
        // Reverse the second half of the list
        while (current != null) {
            ListNode temp = current.next;
            current.next = previous;
            previous = current;
            current = temp;
        }

        // Step 3: Merge the two halves back together
        current = head; // Reset current to the start of the first half

        // Traverse the first and the reversed second half together
        while (previous != null) {
            // 'previous' traverses the reversed list
            ListNode temp = previous.next;
            // Link the current node of the first half to the current node of the reversed second half
            previous.next = current.next;
            // Link the current node of the reversed second half to the next node in the first half
            current.next = previous;

            // Move to the next node in the first half
            current = previous.next;
            // Proceed to the next node in the reversed second half
            previous = temp;
        }
    }
}

```

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:
    void reorderList(ListNode* head) {
        if (!head || !(head->next) || !(head->next->next)) {
            // If the list has 0, 1, or 2 nodes, no reordering is needed.
            return;
        }

        // Use the fast and slow pointer technique to find the middle of the list.
        ListNode* fast = head;
        ListNode* slow = head;
        while (fast->next && fast->next->next) {
            slow = slow->next; // Move one step.
            fast = fast->next->next; // Move two steps.
        }

        // Split the list into two halves.
        ListNode* secondHalf = slow->next;
        slow->next = nullptr; // Terminate first half.

        // Reverse the second half of the list.
        ListNode* prev = nullptr;
        while (secondHalf) {
            ListNode* temp = secondHalf->next;
            secondHalf->next = prev;
            prev = secondHalf;
            secondHalf = temp;
        }

        // Start merging the first and second halves one node at a time.
        ListNode* firstHalf = head;
        ListNode* secondHalfHead = prev; // Points to the head of the reversed second half.
        while (secondHalfHead) {
            ListNode* temp = secondHalfHead->next;
            secondHalfHead->next = firstHalf->next;
            firstHalf->next = secondHalfHead;

            // Move pointers ahead.
            firstHalf = secondHalfHead->next; // Moved to the next of the newly added node.
            secondHalfHead = temp; // Moving to the next node in the reversed half.
        }
    }
};

```

Typescript Solution

```typescript
// Function to reorder a linked list in-place such that the nodes are in a specific order
function reorderList(head: ListNode | null): void {
    let slowPointer = head; // This will be used to find the middle of the list
    let fastPointer = head; // This will go twice as fast to find the end quickly

    // First, split the list into two halves. The slowPointer will end up at the midpoint
    while (fastPointer != null && fastPointer.next != null) {
        slowPointer = slowPointer.next;
        fastPointer = fastPointer.next.next;
    }

    // Reverse the second half of the list using the standard three-pointer approach
    let nextNode = slowPointer.next;
    slowPointer.next = null; // This null will be the new end of the first half
    while (nextNode != null) {
        [nextNode.next, slowPointer, nextNode] = [slowPointer, nextNode, nextNode.next]; // Reverse the pointers in the second half
    }

    // Now merge the two halves, weaving them together one by one
    let leftPointer = head; // This will traverse the first half
    let rightPointer = slowPointer; // This will traverse the reversed second half

    // Weaving the two halves together
    while (rightPointer.next != null) {
        let leftNext = leftPointer.next; // Store the next node in first half
        leftPointer.next = rightPointer; // Link the first node from second half
        rightPointer = rightPointer.next; // Move to the next node in second half
        leftPointer.next.next = leftNext; // Connect the next node from first half
        leftPointer = leftPointer.next.next; // Move to the next node in merged list
    }
}

```

Time and Space Complexity

The given Python code performs a reordering of a singly-linked list such that the node from the end is alternated with the node from the beginning. Here's the breakdown of its computational complexity:

Time Complexity:

The time complexity of the code is determined by several sequential operations:

1. **Finding the Middle of the List**: The first `while` loop uses the fast and slow pointers to find the middle of the list. Since the fast pointer moves at twice the speed of the slow pointer, the loop runs in `O(n/2)` where `n` is the number of nodes in the list. This simplifies to `O(n)`.

2. **Reversing the Second Half of the List**: The second `while` loop reverses the second half of the list, from the node after the middle to the end of the list. This portion of the list has `n/2` nodes, so the loop runs in `O(n/2)`, which also simplifies to `O(n)`.

3. **Merging Two Halves**: The third `while` loop merges the two halves of the list. Since it processes each node exactly once, its time complexity is `O(n)`.

Overall, the time complexity is `O(n) + O(n) + O(n)` which simplifies to `O(n)` because constants are dropped in Big O notation.

Space Complexity:

The space complexity refers to the additional space used by the algorithm, not including the input itself:

1. **Pointers**: The algorithm uses a constant number of pointers (`fast`, `slow`, `cur`, `pre`, `t`) whose space usage does not depend on the input size.

2. **In-Place Operations**: The list is modified in-place, with nodes' `next` pointers being changed to reorder it, but no additional data structures are used that grow with the input size.

Thus, the space complexity is `O(1)` since only a constant amount of extra space is used aside from the input.