24. Swap Nodes in Pairs

Medium Refactor Linked List

Problem Description

Given a singly linked list, the objective is to swap every two adjacent nodes and return the modified linked list. Unlike simple value swapping, this task requires changing the actual node connections. Importantly, the modification must be carried out without altering the values within the nodes—in essence, only the node linkages can be reshuffled. The problem also asks for the function to accommodate linked lists with an odd number of nodes, in which case the final node remains in its original position since there's no pair to swap with.

Intuition

Transforming the structure of a linked list often points towards manipulating node pointers. To swap nodes in pairs, we can employ two main strategies: recursion or iteration.

- **Recursive Approach**: We perform a depth-first traversal, recursively swapping pairs of nodes. In each recursive call, we handle a pair and then delve deeper, assuming that the rest of the list beyond this pair will be solved in the same recursive manner. When the recursion unwinds, the entire list is thus reordered in pairs.
- **Iterative Approach**: Iteration is the alternative strategy that seems natural for this problem, as linked list manipulation often involves loop constructs. The provided code snippet uses an iterative approach with two pointers, which sidesteps the complexities that can come with recursion, such as stack space concerns. We introduce a `dummy` node that precedes the head of the list for simplicity, allowing us to standardize the swapping process without special-casing the head of the list. Two pointers `pre` and `cur` are established to maintain references to the nodes being operated on. `cur` points at the current node under consideration and `pre` trails behind, enabling us to properly link the swapped pairs to the rest of the list. By looping through the list and updating these pointers step-by-step, we systematically exchange the adjacent nodes, while preserving the original node values and ensuring all connections are accurately reestablished post-swap.

Solution Approach

This problem can be resolved by two approaches, recursion or iteration, using the fundamental concepts of linked list traversal and pointer manipulation.

Solution 1: Recursion

The recursive approach to the problem often provides a clean and intuitive solution. Here, we swap each pair of nodes and recursively call the function on the sublist starting with the third node. Here's how the process works:

1. The base condition checks if the current node (`head`) is null or if it's the last node with no pair to swap (`head.next` is null). In either case, the `head` is returned as-is, since no more swapping is possible.
2. For any nodes at least one adjacent node to swap, we store the next node in `t` and perform the swap by setting `head.next` to the recursive call on `head.next.next`. This effectively links the current `head` to the result of swapping the rest of the list.
3. The second node of the pair (`t`) now needs to point to the first (`head`), forming the swapped pair, and `t` is returned as the new head of this swapped section.

Each step handles two nodes and links the swapped pair to the result of the rest of the list, which is computed recursively.

The time complexity of this approach is $O(n)$, where "n" is the number of nodes in the linked list, as each node pair is visited once. The space complexity is also $O(n)$ on account of the recursive call stack.

Solution 2: Iteration

The iterative solution approach is generally more space-efficient for linked lists as it avoids the overhead of the recursive call stack.

1. We start by creating a dummy node that acts as a placeholder prior to the head of the list. This allows us to handle the head of the list the same as any other node in the swapping process, which simplifies the code.
2. We then set two pointers, `pre` starting at the dummy node, and `cur` starting at the list's actual head.
3. Within a loop, as long as `cur` and its next node `cur.next` are not null, we perform the swap. We first store the next node in `t`, then link `cur.next` to `t.next`.
4. After that, `t.next` is updated to point to `cur`, effectively placing `t` before `cur`.
5. We now reset the pointer `pre.next` to `t` to connect the previous part of the list to our newly swapped pair.
6. Finally, we update `pre` to `cur` (the first node of the swapped pair) and move `cur` to `cur.next` to process the next pair.

With this pattern, we successively swap adjacent nodes and move forward in the list until all pairs are swapped or we reach the end of the list.

The time complexity for this iterative approach is also $O(n)$ since each node is visited exactly once, while the space complexity is improved to $O(1)$ because we aren't making any recursive calls, just reassigning pointers.

Both approaches effectively change node pointers to swap adjacent nodes in pairs without altering the node values, successfully solving the problem at hand.

Example Walkthrough

Let's assume we have a linked list with the following values:

[1 -> 2 -> 3 -> 4 -> 5]

Our goal is to swap every pair of adjacent nodes. We'll walk through the iterative approach since it's mentioned in the content provided.

1. We introduce a dummy node to simplify our process. The list now starts with a dummy node followed by the original nodes:

   [dummy -> 1 -> 2 -> 3 -> 4 -> 5]

   Here, `dummy` is a standalone node that we use as a starting point, `pre` starts as the dummy node, and `cur` starts at the head of the real list (node with value 1).

2. We check if `cur` and `cur.next` (nodes 1 and 2) are not null, to proceed with the swap. Since they are both not null, we move onto swapping the nodes. Let's denote `t` as the node after `cur`, which is node 2.

3. We swap the nodes by reassigning pointers:

   - First, we make `cur.next` point to `t.next` (which is node 3):

     [dummy -> 1 ---
     3 -> 4 -> 5] [2 ---]

   - Then we update `t.next` to point to `cur`, placing `t` before `cur`:

     [dummy ---
     2 -> 1 ---
     3 -> 4 -> 5]

4. Now we link the previous part of the list (the dummy node) to our swapped pair by setting `pre.next` to `t`:

   [dummy -> 2 -> 1 -> 3 -> 4 -> 5]

5. We then advance our pointers: `pre` moves to `cur` (node 1), and `cur` moves to `cur.next` (node 3).

6. We continue with the loop, checking if there are more pairs to swap. Again, `cur` and `cur.next` are not null, so we store the node after `cur` (`t`, which is node 4) and perform a similar set of pointer reassignments:

   [dummy -> 2 -> 1 ---
   4 -> 3 ---
   5] [3 ---]

   After swapping, we reconnect the list:

   [dummy -> 2 -> 1 -> 4 -> 3 -> 5]

7. We update `pre` to `cur` (node 3) and `cur` to `cur.next` (node 5).

8. Lastly, we find that `cur.next` is null because node 5 has no pair to swap with. Therefore, we stop the loop and return the head of the modified list, which is the node immediately following our dummy node.

The final swapped list is:

[2 -> 1 -> 4 -> 3 -> 5]

By following these steps iteratively, we've managed to swap the adjacent nodes in pairs throughout the list while maintaining the original values, just as required by the problem statement. The time complexity of this operation is $O(n)$ since we visited each node once, and the space complexity is $O(1)$ due to the constant number of pointers used.

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 swapPairs(self, head: Optional[ListNode]) -> Optional[ListNode]:
        # Create a dummy node that points to the head of the list
        dummy = ListNode(0, head)
        prev_node, current_node = dummy, head

        # Iterate through the list while there are pairs to swap
        while current_node and current_node.next:
            # Store the node that will be swapped with the current node
            temp = current_node.next
            # Adjust 'current_node.next' to the node after 'temp'
            current_node.next = temp.next
            # The 'temp' node now should point to 'current_node' after swapping
            temp.next = current_node
            # 'prev_node.next' is adjusted to point to 'temp' after swapping
            prev_node.next = temp
            # Move 'prev_node' and 'current_node' forward in the list
            prev_node, current_node = current_node, current_node.next

        # Return the new head of the swapped list
        return dummy.next
```

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 ListNode swapPairs(ListNode head) {
        // The dummy node is used to simplify the edge case where the list might contain only one node.
        ListNode dummyNode = new ListNode(0);
        dummyNode.next = head;

        // 'previousNode' always points to the node before the pair that needs to be swapped.
        ListNode previousNode = dummyNode;
        // 'currentNode' is the first node in the pair that needs to be swapped.
        ListNode currentNode = head;

        // Iterate over the list in steps of two nodes at a time.
        while (currentNode != null && currentNode.next != null) {
            // 'nextNode' is the second node in the pair that needs to be swapped.
            ListNode nextNode = currentNode.next;
            // Swap the pair by adjusting the pointers.
            currentNode.next = nextNode.next;
            nextNode.next = currentNode;
            previousNode.next = nextNode;

            // Move 'previousNode' pointer two nodes ahead to the last node of the swapped pair.
            previousNode = currentNode;
            // Advance 'currentNode' to the next pair of nodes to swap.
            currentNode = currentNode.next;
        }

        // The 'next' of dummy points to the new head after swapping pairs.
        return dummyNode.next;
    }
}
```

C++ Solution

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 *     ListNode() : val(0), next(nullptr) {}
 * };
 */
class Solution {
public:
    ListNode* swapPairs(ListNode* head) {
        // Create a dummy node to anchor the modified list and simplify edge cases
        ListNode* dummyNode = new ListNode(0);
        dummyNode->next = head;

        // Use 'previousNode' to keep track of the last node of the previous pair
        ListNode* previousNode = dummyNode;

        // 'currentNode' will be used to iterate through the original list
        ListNode* currentNode = head;

        // Proceed with swapping pairs while there are at least two nodes left to process
        while (currentNode && currentNode->next) {
            // Identify the node to be swapped with the current node
            ListNode* nextNode = currentNode->next;

            // Swap the pair by reassigning pointers
            currentNode->next = nextNode->next;
            nextNode->next = currentNode;
            previousNode->next = nextNode;

            // Move 'previousNode' pointer forward to the current node after swap
            previousNode = currentNode;

            // Move 'currentNode' pointer forward to the next pair
            currentNode = currentNode->next;
        }

        // The 'dummyNode.next' now points to the head of the modified list
        ListNode* swappedHead = dummyNode->next;

        // Clean up memory by deleting the dummy node
        delete dummyNode;

        // Return the head of the modified list with pairs swapped
        return swappedHead;
    }
};
```

Typescript Solution

```typescript
// Type definition for a singly-linked list node.
type ListNode = {
    val: number;
    next: ListNode | null;
};

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

/**
 * Swaps every two adjacent nodes and returns its head.
 * Note: This function assumes the existence of a ListNode type.
 * @param {ListNode | null} head - The head of the linked list.
 * @return {ListNode | null} - The new head of the modified list.
 */
function swapPairs(head: ListNode | null): ListNode | null {
    // Create a dummy node to simplify edge cases.
    let dummy = createListNode(0, head);

    // Initialize pointers for the previous and current nodes.
    let prev = dummy;
    let current = head;

    // Traverse the list in pairs.
    while (current && current.next) {
        // Store the node following the current node.
        const nextPair = current.next;
        // Skip the next node to point to the one after.
        current.next = nextPair.next;
        // Point the next node back to the current node, effecting the swap.
        nextPair.next = current;
        // Link the previous node to the new head of the swapped pair.
        prev.next = nextPair;

        // Move the pointers forward to the next pair.
        prev = current;
        current = current.next;
    }

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

Time and Space Complexity

The time complexity of the code is $O(n)$ where n is the number of nodes in the given linked list. This complexity arises because the algorithm must visit each node to swap the pairs, traversing the entire length of the list once.

The space complexity of the code is $O(1)$ because it only uses a constant amount of extra space. The variables `dummy`, `pre`, `cur`, and `t` are used for manipulation, but the space occupied by these variables does not scale with the size of the input list, thus constituting a constant space complexity.