Documentation

The problem involves reordering a singly linked list such that all nodes with odd indices (1-based) appear first, followed by the nodes with even indices. The order of nodes within the odd and even groups should remain the same as in the input list. For example, given the list [1, 2, 3, 4, 5], the odd-indexed nodes are [1, 3, 5] and the even-indexed nodes are [2, 4]. After reordering, the result should be [1, 3, 5, 2, 4]. This problem requires solving in linear time complexity, O(n), and constant space complexity, O(1), meaning that we canot use extra data structures and must rearrange the list in place.

The solution approach involves maintaining two pointers, odd and even, which will help separate the nodes into odd and even indexed groups. The odd pointer begins at the head of the list, and the even pointer starts at the second node. The goal is to iterate through the list, updating the next pointers of the odd and even nodes to group them accordingly. We also need to keep a reference to the head of the even list so that we can link the last odd node to the head of the even list once the traversal is complete. This step ensures that the reordered list starts with the odd nodes followed by the even nodes.

The process begins by checking if the list is empty or contains just one node. In such cases, the list does not need any reordering, and we return it as is. If there are more nodes, we initialize the two pointers, odd and even. The odd pointer starts at the head of the list, and the even pointer begins at the second node. We also store the head of the even list to reconnect it to the end of the odd list once we finish separating the nodes. As we traverse the list, the odd pointer will move through the odd-indexed nodes, and the even pointer will move through the even-indexed nodes. We adjust their next pointers to point to the appropriate nodes, ensuring the separation of the odd and even groups.

The traversal continues until the even pointer reaches the end of the list or there are no more even nodes to process. When the traversal ends, the odd pointer will be at the last odd node. We then link the next pointer of the last odd node to the head of the even list, effectively combining the two groups into the final reordered list. This solution handles all edge cases, such as empty lists, lists with a single node, and lists with only odd or only even-indexed nodes, by performing the appropriate checks and returning the list as is when no reordering is necessary.

The time complexity of the solution is O(n), where n is the number of nodes in the list. This is because we process each node exactly once during the traversal. The space complexity is O(1) since we only use a constant amount of extra space, namely the two pointers (odd and even), and do not rely on any additional data structures. The in-place reordering ensures that the solution is both space-efficient and time-efficient, meeting the problem's requirements.

This problem provides an excellent opportunity to practice efficiently manipulating linked lists. The solution relies on the careful management of pointers to rearrange nodes in a way that avoids using extra space and minimizes the time complexity. By keeping track of the odd and even nodes and conecting them in the right order, we can achieve the desired result with minimal overhead. The key takeaway is that understanding pointer manipulation and ensuring correct handling of node conections can solve such problems efficiently.

In conclusion, the solution effectively addresses the task of reordering a singly linked list by separating nodes into odd and even indexed groups and then reattaching the even group to the end of the odd group. This approach ensures that the operation is performed in linear time while adhering to the space complexity constraints. The simplicity and efficiency of this approach make it a valuable technique for solving linked list problems, especially in scenarios where space is a premium.