328. Odd Even Linked List

Medium Linked List

Problem Description

Given the head of a singly linked list, we are tasked to reorder it in a specific way. We have to rearrange the nodes such that those at the odd positions (1st, 3rd, 5th, etc.) come before those at the even positions (2nd, 4th, 6th, etc.). It's important to maintain the original relative order of the nodes within the odd group and within the even group. For example, if the original linked list is 1 → 2 → 3 → 4 → 5, after reordering it should look like 1 → 3 → 5 → 2 → 4. This reordering needs to be achieved with certain efficiency constraints: we can't use extra storage space, meaning we must rearrange the nodes in-place, and we must do it with a time complexity of O(n), where n is the number of nodes in the linked list.

Intuition

To solve this problem efficiently, we exploit the in-place nature of linked list reordering. Instead of creating new lists or arrays which would require extra space, we can carefully rearrange the nodes by modifying their `next` pointers.

The approach is to create two pointers, one that will traverse the odd nodes (`a` in the solution code) and another for the even nodes (`b`). The pointers essentially alternate, where `a` progresses to the node pointed to by `b.next` and `b` then moves to the node pointed to by the updated `a.next`. This way, `a` is skipping over the even nodes to link together all the odd nodes, and `b` does the same among the even nodes. This also maintains the original relative order within each subgroup of nodes.

After the loop, all odd-indexed nodes and even-indexed nodes are grouped separately, but we need to link the last node of the odd nodes to the first of the even nodes. To facilitate this, before starting the reordering process, we keep a separate pointer, `c`, at the first even node. When the reordering is complete (which is when `b` or `b.next` is `None`, indicating the end of the list), we simply link the last node of the odd-indexed group (pointed to by `a`) to the first of the even-indexed group (pointed to by `c`). At this point, our reordering is complete and we can return the `head` of the modified list as the final output.

Solution Approach

The implementation of the solution follows a specific pattern where the nodes are reordered in-place using only a few pointers, without the need for additional data structures:

1. First, we initialize three pointers:
   - `a`, which starts at the head of the list and will be used to iterate over odd indexed nodes.
   - `b`, which starts at the second node (head.next) and will be used to iterate over even indexed nodes.
   - `c`, which keeps track of the head of the even indexed sublist (also starts at head.next).
2. We enter a loop that continues until either `b` is `None` (meaning we've reached the end of the list) or `b.next` is `None` (meaning there are no more odd nodes to process since odd nodes are followed by even nodes).
3. Inside the loop:
   - We link `a.next` to `b.next`, which is effectively connecting the current odd-indexed node to the next odd-indexed node (`a.next` skips an even node).
   - We update `a` to be `a.next` so that it now points to the next odd-indexed node.
   - We link `b.next` to `a.next`, since `a` has advanced one step, `a.next` is now an even-indexed node, and we need to skip an odd node.
   - We update `b` to be `b.next` to move to the next even-indexed node.
4. After the loop, all odd nodes are now connected, and `a` points to the last odd node. We set `a.next` to `c`, which links the end of the odd-indexed sublist to the head of the even-indexed sublist we tracked earlier.
5. Finally, we return `head`, which now points to the reordered list where odd-indexed nodes are grouped at the beginning, followed by even-indexed nodes.

The above steps conform to the given constraints of `O(1)` extra space complexity (since we're not using any additional data structures) and `O(n)` time complexity (we go through the list at most twice, once for odd nodes and once for even nodes).

The successful execution of this pattern depends heavily on the understanding of linked list structure and careful manipulation of `next` pointers, which allows us to make changes in place.

Example Walkthrough

Let's walk through the provided solution approach with a small example to illustrate the method. Suppose we have a linked list:

`1 -> 2 -> 3 -> 4`

1. We set up our three pointers:
   - `a` starts at 1 (head of the list).
   - `b` starts at 2 (second node, head.next).
   - `c` also starts at 2 (head of the even sublist).
2. Enter the loop. Currently, `b` is not `None` nor is `b.next` `None`, so we proceed.
3. First iteration:
   - Link `a.next` (1's next) to `b.next` (3). Now the list looks like: `1 -> 3 -> 4`
   - Move `a` to `a.next` (now `a` is at 3).
   - Since `a` moved, we now update `b.next` (2's next) to `a.next` which does not change anything since `a.next` is still 4.
   - Move `b` to `b.next` (now `b` is at 4).

The list now effectively looks like this, with the pointers at their respective positions (a = 3, b = 4, c = 2):

`1 -> 3 -> 4 -> 2 -> NULL`

4. Continue the loop. Now, `b.next` is `None`, we exit the loop.
5. Connect the last odd node (`a`, now at 3) to the head of the even sublist (`c`, which is still at 2): `3 -> 2`. The list now becomes:

`1 -> 3 -> 2 -> 4 -> NULL`

6. Finally, return the head of the reordered list, which is still at 1. The fully reordered list, illustrating odd-positioned nodes followed by even-positioned ones, is now:

`1 -> 3 -> 2 -> 4`

Each step in this approach closely adheres to the specified pattern, ensuring the final output is achieved with an `O(n)` time complexity and without using any extra space. The list provided has been reordered in place effectively.

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:
    # Function to rearrange nodes of a given linked list such that all
    # the odd indexed nodes appear before all the even indexed nodes.
    def oddEvenList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        # If the list is empty, nothing needs to be done.
        if head is None:
            return None

        # Initialize two pointers, one for odd indexed nodes (odd_ptr)
        # and one for even indexed nodes (even_ptr), and also keep track of
        # the start of even indexed nodes (even_head).
        odd_ptr = head
        even_ptr = even_head = head.next

        # Traverse the list, rearranging the nodes.
        while even_ptr and even_ptr.next:
            # Make the next of odd_ptr link to the node after the next
            # (effectively the next odd indexed node).
            odd_ptr.next = even_ptr.next
            # Move the odd_ptr to its new next.
            odd_ptr = odd_ptr.next

            # Connect the next of even_ptr to the node after the next
            # (effectively the next even indexed node).
            even_ptr.next = odd_ptr.next
            # Move the even_ptr to its new next.
            even_ptr = even_ptr.next

        # After all the odd indexed nodes have been linked, append
        # the even indexed nodes to the end of the list formed by odd indexed nodes.
        odd_ptr.next = even_head

        # Return the rearranged list.
        return head
```

Java Solution

```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 {
    public ListNode oddEvenList(ListNode head) {
        // If the list is empty, return null.
        if (head == null) {
            return null;
        }

        // Initialize pointers for manipulation.
        // 'odd' points to the last node in the odd-indexed list.
        // 'even' points to the last node in the even-indexed list.
        // 'evenHead' points to the first node of the even-indexed list.
        ListNode odd = head;
        ListNode even = head.next;
        ListNode evenHead = even;

        // Iterate over the list to rewire nodes.
        while (even != null && even.next != null) {
            // Link the next odd node.
            odd.next = even.next;
            // Move the 'odd' pointer to the next odd node.
            odd = odd.next;

            // Link the next even node.
            even.next = odd.next;
            // Move the 'even' pointer to the next even node.
            even = even.next;
        }

        // After reordering, attach the even-indexed list to the end of the odd-indexed list.
        odd.next = evenHead;

        // Return the head of the modified list.
        return head;
    }
}
```

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:
    ListNode* oddEvenList(ListNode* head) {
        // If the list is empty, just return an empty list
        if (!head) {
            return nullptr;
        }

        // Use 'oddTail' to keep track of the last node in the odd-indexed nodes
        ListNode* oddTail = head;
        // Use 'evenHead' and 'evenTail' to keep track of the even-indexed nodes
        ListNode *evenHead = head->next, *evenTail = evenHead;

        // Iterate as long as there are even nodes to process
        while (evenTail && evenTail->next) {
            // Connect the odd nodes
            oddTail->next = evenTail->next;
            oddTail = oddTail->next;

            // Connect the even nodes
            evenTail->next = oddTail->next;
            evenTail = evenTail->next;
        }

        // Attach the even nodes to the end of the odd nodes
        oddTail->next = evenHead;

        // Return the reorganized list
        return head;
    }
};
```

Typescript Solution

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

/**
 * This function reorders a singly-linked list by separating it into odd-indexed
 * nodes and even-indexed nodes and then concatenating the even-indexed nodes
 * to the end of the odd-indexed nodes.
 * @param {ListNode | null} head - The head node of the linked list.
 * @return {ListNode | null} - The head of the modified list with odd and even nodes separated.
 */
function oddEvenList(head: ListNode | null): ListNode | null {
  // Return null if the list is empty.
  if (!head) {
    return null;
  }

  // Initialize pointers to manage the odd and even parts of the list.
  let odd: ListNode = head; // Points to the last node in the odd list.
  let even: ListNode | null = head.next; // Points to the first node in the even list.
  let evenHead: ListNode | null = head.next; // Keeps track of the head of the even list.

  // Iterate over the list to separate nodes into odd and even lists.
  while (even && even.next) {
    odd.next = even.next; // Link next odd node.
    odd = odd.next; // Move odd pointer to the next node.
    even.next = odd.next; // Link next even node.
    even = even.next; // Move even pointer to the next node.
  }

  // Attach the even list to the end of the odd list.
  odd.next = evenHead;

  // Return the head of the modified list.
  return head;
}
```

Time and Space Complexity

The given Python code reorders a linked list so that all the odd nodes are listed before the even nodes, maintaining their relative order. To analyze the time complexity and space complexity of the function `oddEvenList`, let's go through it step by step:

Time Complexity:

- The while loop continues until it has traversed all the nodes of the linked list because it stops only when `b` is `None` (meaning the end of the list) or `b.next` is `None` (meaning the last node is reached).
- Inside the loop, operations have a constant time complexity (`O(1)`), which includes assigning `next` pointers for `a` and `b`.
- Since each node is visited once during the traversal, the total time complexity is based on the number of nodes in the linked list (`n`). Therefore, the time complexity is `O(n)` where `n` is the number of nodes in the linked list.

Space Complexity:

- The algorithm only uses a constant amount of extra space for pointers `a`, `b`, and `c` irrespective of the size of the input linked list.
- No additional data structures are used that grow with the input size.
- Therefore, the space complexity is `O(1)` constant space.