JavaScript Odd Even Linked List

Challenge

Given the `head` of a singly linked list, group all the nodes with odd indices together followed by the nodes with even indices, and return the reordered list.

The first node is considered odd, and the second node is even, and so on.

Note that the relative order inside both the even and odd groups should remain as it was in the input.

You must solve the problem in `O(1)` extra space complexity and `O(n)` time complexity.

1st Example

```
Input: head = [1,2,3,4,5]
Output: [1,3,5,2,4]
```

2nd Example

```
Input: head = [2,1,3,5,6,4,7]
Output: [2,3,6,7,1,5,4]
```

Constraints

- `-10⁶ <= Node.val <= 10⁶`
- The number of nodes in the linked list is in the range `[0, 10⁴]`.

Solution

```
const oddEvenList = (head) => {
    if (!head || !head.next) return head;

    let odd       = head,
        even      = head.next,
        firstEven = even;

    while (even && even.next) {
        odd.next  = even.next;
        odd       = odd.next;
        even.next = odd.next;
        even      = even.next;
    }

    odd.next = firstEven;

    return head;
};
```

Explanation

I've coded a function called `oddEvenList` that takes a linked list, represented by the `head` node, as input. The purpose of this function is to rearrange the linked list such that all odd-indexed nodes come before even-indexed nodes.

The function first checks if the `head` node is null or if it doesn't have a next node. If either of these conditions is true, it immediately returns the `head` as it is, since no rearrangement is needed.

Three variables, `odd`, `even`, and `firstEven`, are declared. `odd` is initially set to the `head` node, `even` is set to the next node of `head`, and `firstEven` is set to the `even` node.

A while loop is used to iterate through the linked list. The loop continues as long as `even` is not null and `even.next` is not null, ensuring that there are at least two more nodes to rearrange.

Inside the loop, the next node of `odd` is set to the next node of `even`, effectively skipping over the `even` node. Then, `odd` is updated to the node that was just set as the next node of `odd`.

Similarly, the next node of `even` is set to the next node of `odd`, effectively skipping over the next `odd` node. Then, `even` is updated to the node that was just set as the next node of `even`.

After the loop ends, the next node of the last `odd` node is set to `firstEven`, effectively linking the odd-indexed nodes to the even-indexed nodes.

Finally, the function returns the `head` of the rearranged linked list.

In summary, this function rearranges a linked list such that all odd-indexed nodes come before even-indexed nodes. It achieves this by iterating through the linked list, skipping over even-indexed nodes and linking the odd-indexed nodes to the even-indexed nodes. The function then returns the head of the rearranged linked list.