Reverse Singly Linked List LeetCode

1. **Node Class:**
    - This class represents a node in the linked list.
    - It has two attributes: **value** to store the value of the node, and **next** to store the pointer to the next node.
2. **LinkedList Class:**
    - This class represents a singly linked list with basic operations like insertion, display, and reversal.
    - **Constructor LinkedList():**
        - Initializes an empty linked list by setting **head** and **tail** pointers to **NULL** and **length** to 0.
        - Time Complexity: O(1)
        - Space Complexity: O(1)
    - **Function isEmpty():**
        - Checks if the linked list is empty.
        - Time Complexity: O(1)
        - Space Complexity: O(1)
    - **Function insertWhileEmpty(Node* newNode):**
        - Inserts a node into an empty linked list.
        - Time Complexity: O(1)
        - Space Complexity: O(1)
    - **Function pushBack(int value):**
        - Inserts a new node with the given value at the end of the linked list.
        - Time Complexity: O(1)
        - Space Complexity: O(1)
    - **Function display(Node* &head):**
        - Displays the elements of the linked list.
        - Time Complexity: O(n) where n is the number of nodes in the linked list.
        - Space Complexity: O(1)

- **Destructor ~LinkedList():**
    - Frees the memory for all nodes in the linked list.
    - Time Complexity: O(n) where n is the number of nodes in the linked list.
    - Space Complexity: O(1)

3. **Main Function:**
    - Creates a **LinkedList** object and performs various operations on the linked list, including insertion and reversal.

Approach 1: Reverse the linked list iteratively

In this approach, the linked list is reversed by iteratively changing the direction of the pointers. The basic idea is to traverse the list while maintaining three pointers: **prevNode**, **currNode**, and **nextNode**. At each step, the **currNode**'s **next** pointer is directed towards the **prevNode**. This effectively reverses the direction of the pointers. Here's how the process works:

1. Initialize **prevNode** as **NULL**, **currNode** as the **head**, and **nextNode** as **NULL**.
2. While **currNode** is not **NULL**: a. Save **currNode**'s **next** pointer in **nextNode**. b. Update **currNode**'s **next** pointer to point to **prevNode**. c. Move **prevNode** to **currNode** and **currNode** to **nextNode**.
3. Update the **head** to point to the last node, which is now the new starting point.
4. **Time Complexity: O(n) where n is the number of nodes in the linked list.**
5. **Space Complexity: O(1)**

Approach 2: Reverse the linked list recursively

In this approach, the linked list is reversed using recursion. The idea is to reverse the sublist starting from the second node onward, then adjust the pointers for each node as the recursion unwinds. Here's how it works:

1. Base case: If the list is empty or has only one element, return the **head** itself.
2. Recursively reverse the sublist starting from the second node (**head->next**) using the **reverseRecursively** function.
3. Adjust the pointers to reverse the current node: a. Set **head->next->next** to point to the **head**. b. Set **head->next** to **NULL**.
4. Return the new head of the reversed list, which is the **newHead**.

5. **Time Complexity: O(n) where n is the number of nodes in the linked list.**
6. **Space Complexity: O(n) due to recursive function calls.**

Approach 3: Reverse the linked list using a stack

In this approach, the linked list is reversed by utilizing a stack to store the nodes temporarily. The process involves the following steps:

1. Create an empty stack.
2. Traverse the linked list and push each node onto the stack.
3. Update the **head** to point to the last node (top of the stack) as it will be the new starting point.
4. Pop elements from the stack and update the **next** pointers to reverse the list while maintaining the order.
5. **Time Complexity: O(n) where n is the number of nodes in the linked list.**
6. **Space Complexity: O(n) due to the stack used for storage.**