Reverse Doubly Linked List HackerRank

1. **Node Class:**
   The **Node** class represents individual nodes in the doubly linked list. Each node contains pointers to the next and previous nodes, as well as an integer value. The class constructor initializes these pointers and sets the node's value.

2. **DoublyLinkedList Class:**
   The **DoublyLinkedList** class manages the overall operations and behaviors of a doubly linked list. It provides functions to add elements to the list, reverse the list using different approaches, display the list, and handle memory cleanup.

3. **Constructor DoublyLinkedList()**
   - Initializes an empty doubly linked list with head, tail, and length

4. **bool isEmpty() Function:**
   - This function checks whether the linked list is empty or not.
   - Time Complexity: O(1)
   - Space Complexity: O(1)

5. **void insertWhileEmpty(Node* newNode) Function:**
   - Inserts an element into the linked list when it's initially empty.
   - Time Complexity: O(1)
   - Space Complexity: O(1)

6. **Node* pushBack(int value) Function:**
   - Adds a new node with the given value to the end of the linked list.
   - Time Complexity: O(1) for inserting at the tail
   - Space Complexity: O(1)

7. **void display(Node* &head) Function:**
   - Displays the elements of the linked list from the given head node to the end.
   - Time Complexity: O(n), where n is the number of nodes in the list
   - Space Complexity: O(1)

8. **Destructor ~DoublyLinkedList()**
   - Frees memory by deleting all nodes in the doubly linked list.
   - Time Complexity: O(n)
   - Space Complexity: O(1)

Approach 1: Reverse the linked list iteratively

- In this approach, the linked list is reversed by iteratively updating the **next** and **prev** pointers of each node.
- A **currNode** pointer is used to traverse the list.
- The **nextNode** pointer is used to temporarily store the next node in the list before updating the **next** pointer of the current node.
- The **prevNode** pointer is used to temporarily store the previous node in the list before updating the **prev** pointer of the current node.
- The loop continues until **currNode** becomes **nullptr**, which means the end of the list has been reached.
- At each step, the **next** and **prev** pointers of the **currNode** are swapped, and the **currNode** is moved to the **nextNode**.
- After the loop, the **head** pointer is updated to point to the new head (which was previously the tail of the original list).
- **Time Complexity: O(n), where n is the number of nodes in the list**
- **Space Complexity: O(1)**

Approach 2: Reverse the linked list recursively

- In this approach, the linked list is reversed using a recursive process.
- The base case is when the **head** is **nullptr** or when the list has only one element (i.e., **head->next** is **nullptr**).
- In the recursive call, the remaining list starting from **head->next** is reversed, and the **newHead** of the reversed part is returned.
- Once the base case is reached, the **head->next** pointer is set to **head**, and the **prev** pointer of the **head** is updated to **head->next**.
- Finally, the **next** pointer of the **head** is set to **nullptr**, and the **newHead** is returned.
- **Time Complexity: O(n), where n is the number of nodes in the list**
- **Space Complexity: O(n) due to the recursion stack space**

Approach 3: Reverse the linked list using a stack

- This approach uses a stack to reverse the linked list.
- It starts by pushing all nodes onto the stack while traversing the list using a loop.

- After all nodes are pushed onto the stack, the **head** pointer is updated to point to the last node (top of the stack).
- Then, a new traversal begins, where nodes are popped from the stack and their pointers are adjusted to reverse the list.
- The **currNode** pointer is used to keep track of the current node being modified.
- **Time Complexity: O(n), where n is the number of nodes in the list**
- **Space Complexity: O(n) for the stack**

Approach 4: Reverse the linked list in-place using two pointers

- This approach reverses the linked list by iteratively swapping the next and prev pointers of each node.
- The currNode pointer is used to traverse the list.
- In each iteration, the next and prev pointers of the currNode are swapped using the swap function.
- The head pointer is updated to point to the current node (which is now the new head).
- The currNode is then moved to its previous node (which was the next node before swapping), and the process continues.
- **Time Complexity: O(n), where n is the number of nodes in the list**
- **Space Complexity: O(1)**