Q.2 Reverse a Doubly Linked List without creating any new node.

Since a doubly linked list allows traversal in both directions using next and prev pointers, an in-place reversal can be achieved by swapping these pointers for each node. The algorithm will consist of the following steps:

Step 1: Pointer Initialization
Start with a pointer current at the head of the doubly linked list. Also, maintain a pointer temp to assist in swapping.
Step 2: Iteration (Swapping Pointers)
For each node in the list:

- Swap its next and prev pointers.
- Move the current pointer to the previous node (which is now stored in prev after swapping).

Step 3: Updating Head
Once the traversal is complete, update the head of the linked list to the last node encountered.
This approach ensures that we traverse the list only once, making the time complexity **O(n)**.

Pseudo Code:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

def reverseDoublyLinkedList(head):
    current = head
    temp = None

    # Traverse the list and swap next and prev for each node
    while current is not None:
        temp = current.prev
        current.prev = current.next
        current.next = temp
        current = current.prev  # Move to the next node (which was previously 'next')

    # After the loop, temp will be at the last node (new head)
    if temp is not None:
        head = temp.prev  # Set new head of the list

    return head
```

Time Complexity Analysis (O(n)):

- We traverse the doubly linked list **once**, swapping the next and prev pointers for each node.
- Since each node is visited exactly **once**, the total number of operations is proportional to **n** (where n is the number of nodes in the list).
- Hence, the time complexity is **O(n)**.

Space Complexity Analysis (O(1)):

- We **do not** use any extra data structures (like arrays, stacks, or recursion).
- The algorithm only uses a few **temporary pointer variables (current, temp)**, which consume constant space **O(1)**.
- Since the reversal is done **in place**, no additional memory is required.
- Thus, the space complexity is **O(1)**.

Q. Check for anagrams

Algorithm: Hashing Approach for Checking Anagrams

To determine if two words are anagrams, we can use a **hashing (frequency count) approach** as follows:

Steps:

1. Convert both words to lowercase (to ensure case insensitivity).
2. Check if lengths match—if not, they can't be anagrams.
3. Create a frequency dictionary manually for both words:
    - Traverse each word and count occurrences of each character using a dictionary.
4. Compare both dictionaries:
    - If they match, the words are anagrams.
    - Otherwise, they are not.

Pseudo Code:

```
def areAnagrams(word1, word2)
  # Convert words to lowercase to make comparison case insensitive
  word1, word2 = word1.lower(), word2.lower()

  # If lengths are different, they cannot be anagrams
  if len(word1) != len(word2):
    return False

  # Create frequency dictionaries manually
  freq_count1 = {}
  freq_count2 = {}

  # Count character frequencies for word1
```

```
    for char in word1:
        if char in freq_count1:
            freq_count1[char] += 1
        else:
            freq_count1[char] = 1

    # Count character frequencies for word2
    for char in word2:
        if char in freq_count2:
            freq_count2[char] += 1
        else:
            freq_count2[char] = 1

    # Compare both frequency dictionaries
    return freq_count1 == freq_count2
```

Time Complexity: O(n)

- Counting characters in a word takes **O(n)**.
- Comparing two dictionaries takes **O(1)** (since at most **26** lowercase letters exist).
- Overall complexity: **O(n)**.

Space Complexity: O(1)

- Since the maximum number of unique letters is **26** (in the English alphabet), the hash table has a constant size.
- Thus, the space complexity is **O(1)**.