Editorial : Linked List Advance Assignment

Question : Rotate Linked List

```
# Definition of Linked List Node

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

# Complete the function below

class Solution:

    def rotateRight(self,head,k):

        n = 0;
        temp = head;
        while(temp):
            temp = temp.next;
            n += 1;

        # Reduce k so it doesn't exceed the length of the list
        k = k % n;

        # Perform the rotation k times
        for i in range(k):
            prev = None;
            curr = head;

            # Traverse to the end of the list
            while(curr.next != None):
                prev = curr;
                curr = curr.next;

            # 'curr' is now the last node, 'prev' is the node before the last
            prev.next = None;  # Break the link before the last node
            curr.next = head;  # Link the last node to the front
            head = curr;       # Update the head to be the last node

        return head;
```

Explanation

1. **Finding the Length of the List**

   ```
   n = 0;
   temp = head;
   while(temp):
       temp = temp.next;
       n += 1;
   ```

   - We initialize a counter `n` to `0`.
   - We traverse the linked list using a temporary pointer `temp`, incrementing `n` for each node until we reach the end.
   - After this loop, `n` holds the total number of nodes in the linked list.

2. **Adjusting `k`**

   ```
   k = k % n;
   ```

   - If `k` is larger than the length of the list, rotating the list more than `n` times would be redundant (every `n` rotations brings the list back to its original state).
   - We therefore take `k` modulo `n` to handle cases where `k >= n`.

3. **Rotating the List `k` Times**

   ```
   for i in range(k):
       prev = None;
       curr = head;
       while(curr.next != None):
           prev = curr;
           curr = curr.next;

       prev.next = None;
       curr.next = head;
       head = curr;
   ```

   - **Outer Loop** (`for i in range(k)`): We will perform the 1-step rotation process exactly `k` times.
   - **Inner Loop**: We start from the head (`curr = head`) and move forward until `curr.next` is `None`.
     - At the end of this traversal, `curr` points to the **last node** in the list, and `prev` points to the node just before the last node.
   - **Re-linking**:

- `prev.next = None;` removes the last node from its current position by making the second-to-last node the new tail.
- `curr.next = head;` places the last node at the **front** of the list.
- `head = curr;` updates the list's head pointer to this newly moved node.

4. **Return the New Head**

```
return head;
```

- After `k` such single-step rotations, we return the updated `head` of the list.

Time Complexity Note

- This approach performs a **1-step rotation** `k` times. Each 1-step rotation requires traversing the list (in the worst case), leading to a time complexity of approximately **O(n*k)**.
- There is a more optimal solution that runs in **O(n)** by connecting the tail to the head to form a cycle, then breaking at the correct spot. However, the above code correctly implements a simpler, step-by-step rotation approach.

Question : Palindrome List

```
'''
Node {
    int data,
    Node* next
}
'''
'''
This function should return true/false if linked list is palindrome/not palindrome
'''
def isPalindrome(head):
    if not head or not head.next:
        return True

    slow = head
    fast = head

    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
```

```python
    prev = None
    curr = slow
    while curr:
        next_node = curr.next
        curr.next = prev
        prev = curr
        curr = next_node

    first_half = head
    second_half = prev

    while second_half:
        if first_half.data != second_half.data:
            return False
        first_half = first_half.next
        second_half = second_half.next

    return True
```

Explanation

1. **Base Cases**

```python
if not head or not head.next:
    return True
```

   - If the list is empty (`head` is `None`) or has only one node (`head.next` is `None`), it is trivially a palindrome, so we return `True`.

2. **Finding the Middle of the List**

```python
slow = head
fast = head
while fast and fast.next:
    slow = slow.next
    fast = fast.next.next
```

   - We use two pointers, `slow` and `fast`.
   - `slow` moves one step at a time, `fast` moves two steps at a time.
   - By the time `fast` reaches the end (or just before it), `slow` will be at the **middle** of the linked list.

3. **Reversing the Second Half**

```
prev = None
curr = slow
while curr:
    next_node = curr.next
    curr.next = prev
    prev = curr
    curr = next_node
```

   - Starting from the middle (`slow`), we reverse the linked list from this point to the end.
   - `prev` becomes the new head of the reversed second half by the time the loop finishes.

4. **Comparing Both Halves**

```
first_half = head
second_half = prev
while second_half:
    if first_half.data != second_half.data:
        return False
    first_half = first_half.next
    second_half = second_half.next
```

   - We set `first_half` to the original head of the list and `second_half` to the head of the reversed second half.
   - We then traverse both halves in tandem:
     - If at any point the data in `first_half` does not match the data in `second_half`, we know the list is not a palindrome, and we return `False`.
   - If we finish this loop without finding a mismatch, it means the list **is** a palindrome.

5. **Return True if All Match**

```
return True
```

   - If all corresponding nodes in the first and second halves match, the function returns `True`, indicating the linked list is a palindrome.

How It Works Overall

- We find the middle of the list using fast and slow pointers.

- We reverse the second half of the list in-place.
- We compare the first half and the reversed second half node by node.
- If all nodes match, the list is a palindrome; otherwise, it is not.

Question : Remove Maximum Element in Linked List

```
class Solution:
    def deleteMaximum(self, head):
        # Write your code here
        if(head == None):
            return -1;

        if(head.next==None):
            return None;

        i = 0;
        pos = -1;
        curr = head;
        mx = float("-inf");
        while(curr!=None):
            if(curr.data>=mx):
                mx = curr.data;
                pos = i;
            curr = curr.next;
            i+=1;

        prev = None;
        temp = head;
        for i in range(pos):
            prev = temp;
            temp = temp.next;

        if(prev == None):
            return temp.next;

        prev.next = temp.next;
        return head;
```

Explanation

1. **Check for Empty or Single-Node List**

```python
if head == None:
    return -1  # Indicate an empty list
if head.next == None:
    return None # Removing the only node leaves the list empty
```

- If the list is empty, return `-1`.
- If there's only one node, removing it returns `None`.

2. **Find the Maximum Value and Its Position**

```python
i = 0
pos = -1
curr = head
mx = float("-inf")
while curr != None:
    if curr.data >= mx:
        mx = curr.data
        pos = i
    curr = curr.next
    i += 1
```

- We track the **largest value** `mx` and the **position** `pos` where it occurs.
- If multiple nodes share the maximum value, we update `pos` to the **last** occurrence (because `>=` is used).

3. **Traverse to the Node Before the Maximum**

```python
prev = None
temp = head
for i in range(pos):
    prev = temp
    temp = temp.next
```

- We move `temp` to the node **at** position `pos`.
- `prev` will end up as the node **before** `temp`.

4. **Delete the Maximum Node**

```python
if prev == None:
    return temp.next  # If the max node is at the head
prev.next = temp.next
return head
```

- If `prev` is `None`, it means the max node was at the head, so we skip it by returning `temp.next`.
- Otherwise, we link `prev.next` to `temp.next`, removing the maximum node from the chain.

---

6) Add 1 to Linked List

```
def addOne(head):
    def reverseList(node):
        prev = None
        current = node
        while current:
            next_node = current.next
            current.next = prev
            prev = current
            current = next_node
        return prev

    head = reverseList(head)

    carry = 1
    current = head
    while current:
        current.data += carry
        if current.data < 10:
            carry = 0
            break
        current.data = 0
        if not current.next:
            current.next = Node(0)
        current = current.next

    head = reverseList(head)
    return head
```

Explanation

1. **Reverse the List**

```
def reverseList(node):
    ...
head = reverseList(head)
```

- We reverse the linked list so the **least significant digit** is at the front.

2. **Add One**

```
carry = 1
current = head
while current:
    current.data += carry
    if current.data < 10:
        carry = 0
        break
    current.data = 0
    if not current.next:
        current.next = Node(0)
    current = current.next
```

   - Start with a carry of 1 (since we're adding 1).
   - Add `carry` to `current.data`.
   - If `current.data` is now 10 or more, set it to 0 and keep carry = 1.
   - If `current.data` is less than 10, set carry = 0 and break out of the loop (no further carry to propagate).
   - If we reach the end of the list and still have carry = 1, we create a new node (e.g., going from 999 to 1000).

3. **Reverse Again**

```
head = reverseList(head)
return head
```

   - After adding 1, we reverse the list back to its original order.
   - Return the updated head.

Question : Insert Node in Doubly Linked List at the End

```
def addNode(A, B):
    X = Node(B)
    if A is None:
        return X
    head = A
    while A.next != None:
        A = A.next
    A.next = X
```

```
    X.prev = A
    return head
```

Explanation

1. **Function Parameters**
   - `A`: The head of the doubly linked list.
   - `B`: The value to be inserted into a new node.
2. **Create the New Node**

   ```
   X = Node(B)
   ```

   - We instantiate a new `Node` with the data `B`.
3. **Check if the List is Empty**

   ```
   if A is None:
       return X
   ```

   - If there is no existing list, the new node `X` becomes the head.
4. **Traverse to the End**

   ```
   head = A
   while A.next != None:
       A = A.next
   ```

   - We move through the list until we find the last node (`A.next == None`).
5. **Link the New Node**

   ```
   A.next = X
   X.prev = A
   return head
   ```

   - We attach `X` as the last node by setting `A.next = X`.
   - We set the `prev` pointer of `X` to `A`.
   - We return the original `head` of the list.

Question : Remove Duplicates

```python
class Solution:
    def deleteDuplicates(self, head):
        temp = head
        if (temp is None):
            return head
        while(temp.next):
            if (temp.data == temp.next.data):
                new1 = temp.next.next
                temp.next = new1
            else:
                temp = temp.next

        return head
```

Explanation

1. **Initial Check**

   ```python
   if (temp is None):
       return head
   ```

   - If the linked list is empty (`head` is `None`), we just return `head` .

2. **Traversing the List**

   ```python
   while (temp.next):
       ...
   ```

   - We use `temp` to iterate through the list until we reach the second-to-last node (`temp.next` must exist).

3. **Checking for Duplicates**

   ```python
   if (temp.data == temp.next.data):
       new1 = temp.next.next
       temp.next = new1
   else:
       temp = temp.next
   ```

- If `temp.data == temp.next.data`:
  - We've found a duplicate. We skip over the next node by linking `temp.next` to `temp.next.next`.
  - This effectively removes the duplicate node from the list.
- **Otherwise**:
  - We move `temp` forward by one node (i.e., `temp = temp.next`).

4. **Return the Updated Head**

```
return head
```

- The head of the list remains the same (unless it was `None` initially).
- By the end of this loop, all adjacent duplicates have been removed.

Question : Reverse in Pair

```
class Solution:
    def reversepair(self, head: ListNode) -> ListNode:
        # Write your code here
        if not head or not head.next:
            return head

        # Create a dummy node to act as the new head of the modified list
        dummy = ListNode(0)
        dummy.next = head

        # Initialize prev to the dummy node and curr to the head
        prev = dummy
        curr = head

        # Traverse the list while there are at least two nodes to swap
        while curr and curr.next:
            nextNode = curr.next  # The node to be swapped with curr

            # Perform the swap
            curr.next = nextNode.next
            nextNode.next = curr
            prev.next = nextNode

            # Move prev and curr pointers forward
            prev = curr
            curr = curr.next
```

```
    # The new head of the list is the node after the dummy node
    head = dummy.next
    return head
```

Explanation

1. **Edge Case**

   ```
   if not head or not head.next:
       return head
   ```

   - If the list has fewer than two nodes, no swaps are needed, so we return the original `head`.

2. **Dummy Node**

   ```
   dummy = ListNode(0)
   dummy.next = head
   ```

   - We create a temporary “dummy” node that points to the original head.
   - This simplifies edge cases (like swapping the first two nodes).

3. **Pointers**

   ```
   prev = dummy
   curr = head
   ```

   - `prev` initially points to the dummy node.
   - `curr` points to the first node in the list.

4. **Swap in Pairs**

   ```
   while curr and curr.next:
       nextNode = curr.next
       curr.next = nextNode.next
       nextNode.next = curr
       prev.next = nextNode
       ...
   ```

   - We keep swapping adjacent pairs as long as `curr` and `curr.next` are valid.
   - **Swap Steps**:

   a. `nextNode = curr.next`: Identify the node to swap with `curr`.
   b. `curr.next = nextNode.next`: Link `curr` to the node after `nextNode`.
   c. `nextNode.next = curr`: Make `nextNode` point back to `curr`.
   d. `prev.next = nextNode`: Connect `prev` to `nextNode`, effectively placing `nextNode` in front of `curr`.

5. **Advance Pointers**

   ```
   prev = curr
   curr = curr.next
   ```

   - We move `prev` to the node `curr` (which is now the second node in the swapped pair).
   - We move `curr` forward to continue the process on the next pair.

6. **Return the New Head**

   ```
   head = dummy.next
   return head
   ```

   - The new head is `dummy.next` after all swaps are complete.
   - This could be different from the original head if at least one swap occurred.
