Solution Review: Reverse a Linked List

This review provides a detailed analysis of the different ways to solve the Reverse a Linked List challenge.

We'll cover the following

Solution: Iterative Pointer Manipulation

main.py

LinkedList.py

Node.py

```
from LinkedList import LinkedList
from Node import Node
def reverse(lst):
  # To reverse linked, we need to keep track of three things
  previous = None # Maintain track of the previous node
  current = lst.get_head() # The current node
  next = None # The next node in the list

  #Reversal
  while current:
    next = current.next_element
    current.next_element = previous
    previous = current
```

```
14      current = next
15
16      #Set the last element as the new head node
17      lst.head_node = previous
18    return lst
19
20  lst = LinkedList()
21  lst.insert_at_head(6)
22  lst.insert_at_head(4)
23  lst.insert_at_head(9)
24  lst.insert_at_head(10)
25  lst.print_list()
26
27  reverse(lst)
28  lst.print_list()
```

The brain of this solution lies in the loop which iterates through the list. For any `current` node, its link with the `previous` node is reversed and `next` stores the next node in the list:

- Store the `current` node's `next_element` in `next`
- Set `current` node's `next_element` to `previous` (reversal)
- Make the `current` node the new `previous` so that it can be used for the next iteration
- Use `next` to move on to the next node

In the end, we simply point the `head` to the last node in our loop.

Time Complexity

The algorithm runs in *O(n)* since the list is traversed once.

Hopefully, you've got a good idea of pointer manipulation by now. The next challenge will be a little trickier, so don't be afraid to test yourself.

Back

Challenge 5: Reverse a Linked List

Next

Challenge 6: Detect Loop in a Linked L...