Solution Review: Insertion at Tail

This review provides a detailed analysis of the solution to the Insertion at Tail challenge.

We'll cover the following

Solution: List Traversal

main.py

LinkedList.py

Node.py

```
from LinkedList import LinkedList
from Node import Node
# Access HeadNode => list.getHead()
# Check if list is empty => list.isEmpty()
# Node class  { int data ; Node nextElement;}

# Inserts a value at the end of the list


def insert_at_tail(lst, value):
    # Creating a new node
    new_node = Node(value)

    # Check if the list is empty, if it is simply point head t
    if lst.get_head() is None:
        lst.head_node = new_node
        return

    # if list not empty, traverse the list to the last node
    temp = lst.get_head()

    while temp.next_element:
        temp = temp.next_element

    # Set the nextElement of the previous node to new node
    temp.next_element = new_node
    return


lst = LinkedList()
lst.print_list()
insert_at_tail(lst, 0)
lst.print_list()
insert_at_tail(lst, 1)
lst.print_list()
insert_at_tail(lst, 2)
lst.print_list()
insert_at_tail(lst, 3)
lst.print_list()

```

If you grasped the logic behind insertion at the head of a linked list, this shouldn't be much of a challenge. If the list is empty, the situation is exactly like insertion at head. Otherwise, we can simply use a loop to reach the tail of the list and set our new node as the `next_element` of the last node (**line 26**).

Time Complexity

This algorithm traverses the entire linked list and hence, works in *O(n)* time.

At this point, we have covered the first two types of insertions. The last one, **Insertion at the k^{th} Position**, is just an extension of these two. If you need to insert a node at a specific index in the list, simply iterate to that position and appropriately switch pointers. Try it out on your own.

In the next lesson, we will discuss the search algorithm for linked lists.

Back

Challenge 1: Insertion at Tail

Next

Challenge 2: Search in a Singly Linked ...

Completed