Lab 5: Linked Lists Solution

3) Augmenting our linked list implementation

It makes sense that our implementation of *LinkedList.__len__* is so slow; but how is the built-in *list.__len__* so much faster?

It turns out that built-in Python lists use an additional attribute to store their length, so that whenever *list.__len__* is called, it simply returns the value of this attribute.

The process of adding an extra attribute to an existing data structure is known as augmentation, and is very common in computer science. Every data structure augmentation poses a question of trade-offs:

- The benefit of augmenting is that the extra attribute makes certain operations simpler and/or more efficient to implement.
- The cost of augmenting is that this extra attribute increases the complexity of the data structure implementation.

  In particular, such attributes often have representation invariants associated with them that must be maintained every time the data structure is mutated.

1. Create a copy of your *LinkedList* class (you can pick a name for the copy), and add a new private attribute *_length* to the class documentation and initializer.

   Write down a representation invariant for this new attribute; you can use English here, but try to be precise without using the word "length" in your description. (Hint: how do we define length in terms of the nodes of a list?)

```
    class LinkedList:
        """A linked list implementation of the List ADT.
        """
        # === Private Attributes ===
        # _first:
        #     The first node in the linked list, or None if the list
    is empty.
        # _length:
        #     The number of nodes in the linked list.
        #
```

```
        # === Representational Invariants ===
        # - _length >= 0

```

Listing 1: task_3_step_1_solution.py

2. Update each mutating method to preserve your representation invariant for this new attribute. (Why don't we need to worry about the non-mutating methods?)

```
    class LinkedList:
        """A linked list implementation of the List ADT.
        """
        # === Private Attributes ===
        # _first:
        #     The first node in the linked list, or None if the list is empty.
        # _length:
        #     The number of nodes in the linked list.
        #
        # === Representational Invariants ===
        # - _length >= 0
        _first: Optional[_Node]

        def __init__(self, items: list) -> None:
            """ Initialize a new linked list containing the given items.

                The first node in the linked list contains the first item in <items>
            """

            index = 0
            while index < len(items):
                items[index] = _Node(items[index])

                if index > 0:
                    items[index-1].next = items[index]
                index += 1

            self._first = items[0]
            # ======= (Task 3, Step 2) =======
            self._length = len(items)
            # ================================
        ...

        def insert(self, index: int, item: Any) -> None:
            """Insert a the given item at the given index in this list.

            Raise IndexError if index > len(self) or index < 0.
            Note that adding to the end of the list is okay.

```

```
            # >>> lst = LinkedList([1, 2, 10, 200])
            # >>> lst.insert(2, 300)
            # >>> str(lst)
            # '[1 -> 2 -> 300 -> 10 -> 200]'
            # >>> lst.insert(5, -1)
            # >>> str(lst)
            # '[1 -> 2 -> 300 -> 10 -> 200 -> -1]'
            # >>> lst.insert(100, 2)
            # Traceback (most recent call last):
            # IndexError
            """
            # Create new node containing the item
            new_node = _Node(item)

            if index == 0:
                self._first, new_node.next = new_node, self._first
            else:
                # Iterate to (index-1)-th node.
                curr = self._first
                curr_index = 0
                while curr is not None and curr_index < index - 1:
                    curr = curr.next
                    curr_index += 1

                if curr is None:
                    raise IndexError
                else:
                    # Update links to insert new node
                    curr.next, new_node.next = new_node, curr.next
                    # ======= (Task 3, Step 2) =======
                    self._length += 1
                    # ================================


```

Listing 2: task_3_step_2_solution.py

There is no need to worry about non-mutating methods because the number of nodes in linked list doesn't change.

3. Now let's enjoy the benefit of this augmentation!

   Modify your new class' *__len__* method to simply return this new attribute.

   Use doctests wisely to ensure you've made the correct changes for this and the previous step.

4. Finally, perform some additional timing tests to demonstrate that you really have improved the efficiency of *__len__*.