

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?)

```
1  class LinkedList:
2      """A linked list implementation of the List ADT.
3      """
4      # === Private Attributes ===
5      # _first:
6      #     The first node in the linked list, or None if the list
7      #     is empty.
8      # _length:
9      #     The number of nodes in the linked list.
10     #
```

```

10     # === Representational Invariants ===
11     # - _length >= 0
12

```

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?)

```

1  class LinkedList:
2      """A linked list implementation of the List ADT.
3      """
4      # === Private Attributes ===
5      # _first:
6      #     The first node in the linked list, or None if the list
7      #     is empty.
8      # _length:
9      #     The number of nodes in the linked list.
10     #
11     # === Representational Invariants ===
12     # - _length >= 0
13     _first: Optional[_Node]
14
15     def __init__(self, items: list) -> None:
16         """ Initialize a new linked list containing the given
17         items.
18
19         The first node in the linked list contains the first
20         item in <items>
21         """
22
23         index = 0
24         while index < len(items):
25             items[index] = _Node(items[index])
26
27             if index > 0:
28                 items[index-1].next = items[index]
29             index += 1
30
31         self._first = items[0]
32         # ===== (Task 3, Step 2) =====
33         self._length = len(items)
34         # =====
35         ...
36
37     def insert(self, index: int, item: Any) -> None:
38         """Insert a the given item at the given index in this
39         list.
40
41         Raise IndexError if index > len(self) or index < 0.
42         Note that adding to the end of the list is okay.

```

```

40     # >>> lst = LinkedList([1, 2, 10, 200])
41     # >>> lst.insert(2, 300)
42     # >>> str(lst)
43     # '[1 -> 2 -> 300 -> 10 -> 200] '
44     # >>> lst.insert(5, -1)
45     # >>> str(lst)
46     # '[1 -> 2 -> 300 -> 10 -> 200 -> -1] '
47     # >>> lst.insert(100, 2)
48     # Traceback (most recent call last):
49     # IndexError
50     """
51     # Create new node containing the item
52     new_node = _Node(item)
53
54     if index == 0:
55         self._first, new_node.next = new_node, self._first
56     else:
57         # Iterate to (index-1)-th node.
58         curr = self._first
59         curr_index = 0
60         while curr is not None and curr_index < index - 1:
61             curr = curr.next
62             curr_index += 1
63
64         if curr is None:
65             raise IndexError
66         else:
67             # Update links to insert new node
68             curr.next, new_node.next = new_node, curr.next
69             # ===== (Task 3, Step 2) =====
70             self._length += 1
71             # =====
72
73

```

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.

```

1     class LinkedList:
2         """A linked list implementation of the List ADT.
3         """
4         # === Private Attributes ===

```

```

5         # _first:
6         #     The first node in the linked list, or None if the list
is empty.
7         # _length:
8         #     The number of nodes in the linked list.
9         #
10        # === Representational Invariants ===
11        # - _length >= 0
12        _first: Optional[_Node]
13
14        def __init__(self, items: list) -> None:
15            """ Initialize a new linked list containing the given
items.
16
17            The first node in the linked list contains the first
item in <items>
18            """
19
20            index = 0
21            while index < len(items):
22                items[index] = _Node(items[index])
23
24                if index > 0:
25                    items[index-1].next = items[index]
26                index += 1
27            # ===== (Task 3, Step 3) =====
28            self._first = items[0] if len(items) > 0 else None
29            # =====
30            self._length = len(items)
31
32        ...
33        def insert(self, index: int, item: Any) -> None:
34            """Insert a the given item at the given index in this
list.
35
36            Raise IndexError if index > len(self) or index < 0.
37            Note that adding to the end of the list is okay.
38
39            # >>> lst = LinkedList([1, 2, 10, 200])
40            # >>> lst.insert(2, 300)
41            # >>> str(lst)
42            # '[1 -> 2 -> 300 -> 10 -> 200]'
43            # >>> lst.insert(5, -1)
44            # >>> str(lst)
45            # '[1 -> 2 -> 300 -> 10 -> 200 -> -1]'
46            # >>> lst.insert(100, 2)
47            # Traceback (most recent call last):
48            # IndexError
49            """
50            # Create new node containing the item
51            new_node = _Node(item)
52
53            if index == 0:

```

```

54         self._first, new_node.next = new_node, self._first
55     else:
56         # Iterate to (index-1)-th node.
57         curr = self._first
58         curr_index = 0
59         while curr is not None and curr_index < index - 1:
60             curr = curr.next
61             curr_index += 1
62
63         if curr is None:
64             raise IndexError
65         else:
66             # Update links to insert new node
67             curr.next, new_node.next = new_node, curr.next
68             self._length += 1
69
70
71     #
-----
72     # Lab Task 1
73     #
-----
74
75     # TODO: implement this method
76     # NOTE: The doctest will not run until task 3
77     # ===== (Task 3, Step 3) =====
78     def __len__(self) -> int:
79         """Return the number of elements in this list.
80
81         >>> lst = LinkedList([])
82         >>> len(lst)           # Equivalent to lst.__len__()
83         0
84         >>> lst = LinkedList([1, 2, 3])
85         >>> len(lst)
86         3
87         """
88         return self._length
89     # =====
90
91     ...
92     # TODO: implement this method
93     # ===== (Task 3, Step 3) =====
94     def __setitem__(self, index: int, item: Any) -> None:
95         """Store item at position <index> in this list.
96
97         Raise IndexError if index >= len(self).
98
99         >>> lst = LinkedList([1, 2, 3])
100         >>> lst[0] = 100 # Equivalent to lst.__setitem__(0, 100)
101         >>> lst[1] = 200
102         >>> lst[2] = 300
103         >>> str(lst)

```

```

103         '[100 -> 200 -> 300]',
104         """
105
106         curr = self._first
107         i = 0
108
109         while (curr is not None) and (i <= index):
110             if index != i:
111                 curr = curr.next
112                 i += 1
113                 continue
114
115             curr.item = item
116             return
117
118         raise IndexError
119     # =====
120

```

Listing 3: task_3_step_3_solution.py

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

In comparison to previous result of `__len__`, time taken stays constant over input sizes.

This is a significant improvement.

```

1  LinkedList] Size    1000: 3.918999999998063e-06
2  [LinkedList] Size   2000: 2.3170000000001679e-06
3  [LinkedList] Size   4000: 2.640999999997673e-06
4  [LinkedList] Size   8000: 2.5440000000001572e-06
5  [LinkedList] Size  16000: 2.63400000000013575e-06
6

```