Linked List Mutation

Updated October 5, 3:30pm

Last lecture, we talked about the basic `LinkedList` class, and spent a lot of time traversing a linked list. Here's a reminder of the basic traversal pattern using a `while` loop: understanding this is critical before moving on!

```
curr = my_linked_list.first
while curr is not None:
    print(curr.item)
    curr = curr.next
```

All of the methods we looked at least time were **non-mutating**, meaning they didn't change the linked list. We started with them because they're generally easier to understand than their mutating cousins. Today, we're going to look at the two major **mutating** operations on linked lists: deleting and inserting nodes.

Deleting a Node

Suppose we want to *remove* the `i`-th node from a Linked List. We can accomplish this by performing the following sequence of steps:

1. Iterate to the (`i-1`)-th node (same type of traversal as above)
2. Update the `next` attribute of that node to skip over the `i`-th node, directly to the (`i+1`)-th node.

Here's an implementation of the above idea:

```
def remove(self, index):
    """ (LinkedList, int) -> NoneType
    Remove node at position index from this list.
    Raise IndexError if index is >= the length of self.
    """
    # Use the __len__ method from last class
    # to detect an index error
    if len(self) <= index:
        raise IndexError

    if index == 0:
        # Update self.first to remove the first node
        self.first = self.first.next
    else:
        # Iterate to (index-1)-th node
        curr = self.first
        for i in range(index - 1):
            curr = curr.next

        # Update link to skip over index-th node
        curr.next = curr.next.next
```

Note that we used a slightly different strategy for removing the first node; why was this necessary?

Also, you may remember from last lecture that `__len__` iterates through the whole list to determine its length. Thus, this method loops through the list twice: once to calculate the length of the list to check for an index error, and again to get to the `(index-1)`-th node. Can you use a `while` loop instead of a `for` loop to check for an index error directly, looping just once?

Inserting a Node

Now suppose we want to insert a new node with a specific item at a position `i`. Like deletion, we can accomplish this just by changing a few links.

0. Create a new node containing the item.
1. Iterate to the (`i-1`)-th node.
2. Update the links of the (`i-1`)-th and new node to insert the new node in the list.

```
def insert(self, index, item):
    """ (LinkedList, int, object) -> NoneType

    Insert a new node containing item at position index.
    Raise IndexError if index is > the length of self.
    Note that adding to the end of a linked list is okay.
    """
    if index > len(self):
        raise IndexError

    # Create new node containing the item
    new_node = Node(item)

    if index == 0:
        new_node.next = self.first
        self.first = new_node
    else:
        # Iterate to (index-1)-th node
        curr = self.first
        for i in range(index - 1):
            curr = curr.next

        # Update links to insert new node
        new_node.next = curr.next
        curr.next = new_node
```

Once you understand how this method works, try making it more efficient in the same way as suggested for `remove` above.

David Liu (liudavid at cs dot toronto dot edu)
Come find me in BA4260
Site design by Deyu Wang (dyw999 at gmail dot com)