# 11.3 Mutating Linked Lists

All of the linked list methods we have looked at so far are *non-mutating*, meaning they did not change the linked list upon which they were called. We started with these methods because they're generally easier to understand than their mutating cousins. Now, we're going to look at the two major **mutating** operations on linked lists: inserting into and deleting items from a linked list.

## *Warm-up:* `LinkedList.append`

We'll start with the simplest version of this: appending a new item to the end of a linked list. Here's a method header, with a docstring to illustrate its use.

```
class LinkedList:
    def append(self, item: Any) -> None:
        """Add the given item to the end of this linked list.

        >>> linky = LinkedList()  # LinkedLists start empty
        >>> linky.append(111)
        >>> linky.append(-5)
        >>> linky.append(9000)
        >>> linky.to_list()
        [111, -5, 9000]
        """
```

Recall that a `LinkedList` object has only one attribute, a reference to the *first* node in the list. Unfortunately, this means that we have some work to do to implement `LinkedList.append`: before adding the item, we need to find the current last node in the linked list, and then add a new node to the end of that. Let's start by creating a new node containing the given item, and then using our linked list traversal code template:[1]

[1] We're following our own advice from the previous section!

```
    def append(self, item: Any) -> None:
        """..."""
        new_node = _Node(item)  # The node to insert into the linked list

        curr = self._first
        while curr is not None:
            ... curr.item ...
            curr = curr.next
```

This template is a good start, but now our thinking must begin. First: what do we do with `curr.item`? The answer is "Nothing!"—we don't need to actually use any of the existing items in the list, and instead are just going through the list to get to the last node. Unfortunately, there's a problem with the loop: this loop is designed to keep going until we've processed all of the elements of the list, and `curr` becomes `None`. But this is actually going too far for our purposes: we want to stop the loop when it reaches the last node.[2]

[2] This is actually a subtle instance of the classic "off-by-one" error in computer science: our loop iterates one too many times.

How do we know when the loop has reached the last node? We can tell by looking at the `next` attribute: *a `_Node` object's `next` attribute refers to the next `_Node` in the linked list, or `None` if there are no more nodes.* So we can modify our loop condition to check for using `curr.next is None` instead:

```
def append(self, item: Any) -> None:
    """..."""
    new_node = _Node(item)  # The node to insert into the linked list

    curr = self._first
    while curr.next is not None:
        curr = curr.next

    # After the loop, curr is the last node in the LinkedList.
    assert curr is not None and curr.next is None
```

At this point, you might notice a flaw in this change: we aren't guaranteed that `curr` starts off as a node—it could be `None`.[3] But because we don't want to get bogged down with

[3] Exercises: when would this happen? What error would be raised by our code if `curr` were `None`?

handling that case right now, we'll add an if statement along with a TODO comment in our code and keep going for now.

```
def append(self, item: Any) -> None:
    """..."""
    new_node = _Node(item)  # The node to insert into the linked list

    if self._first is None:
        ...  # TODO: Handle this case
    else:
        curr = self._first
        while curr.next is not None:
            curr = curr.next

        # After the loop, curr is the last node in the LinkedList.
        assert curr is not None and curr.next is None
```

So then after the loop ends, we know that `curr` refers to the last node in the linked list, and we are finally in a position to add the given item to the linked list. To do so, we need to link the current last node to the new node we created.

```
def append(self, item: Any) -> None:
    """..."""
    new_node = _Node(item)  # The node to insert into the linked list

    if self._first is None:
        ...  # TODO: Handle this case
    else:
        curr = self._first
        while curr.next is not None:
            curr = curr.next

        # After the loop, curr is the last node in the LinkedList.
        assert curr is not None and curr.next is None
        curr.next = new_node
```

And finally, let's handle that TODO. By writing an if statement, we know exactly what condition we need to handle: when `self._first is None`. We know from the documentation of our `LinkedList` class that `self._first` can only be `None` if `self` refers to an empty linked list. In this case, the new node becomes the first node in the linked list.

```
def append(self, item: Any) -> None:
    """..."""
    new_node = _Node(item)

    if self._first is None:
        self._first = new_node
    else:
        curr = self._first
        while curr.next is not None:
            curr = curr.next

        # After the loop, curr is the last node in the LinkedList.
        assert curr is not None and curr.next is None
        curr.next = new_node
```

## *A more general initializer*

With our `append` method in place, we can now stop creating linked lists by manually assigning attributes, and instead modify our linked list initializer to take in an iterable collection of values, which we'll then append one at a time to the linked list:[4]

[4] Exercise: Try removing the `self._first = None` statement and running the code. What goes wrong, and why?

```
class LinkedList:
    def __init__(self, items: Iterable) -> None:
        """Initialize a new linked list containing the given items.
        """
        self._first = None
        for item in items:
            self.append(item)
```

While this code is perfectly correct, it turns out that it is rather inefficient. We'll leave it as an exercise for you to analyse the running time of `LinkedList.append` and this version of `LinkedList.__init__` (for the latter, let $n$ be the size of the input iterable). You'll explore a faster way of implementing this initializer a bit later in the course.