CSC111 Lecture 4: Mutating Linked Lists, Part 2

Hisbaan Noorani

January 20, 2021

Contents

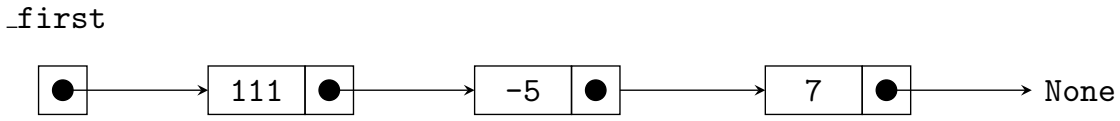
1 Exercise 1: Index-based deletion

Last class, we studied *index-based insertion* into a linked list, and implemented the `LinkedList.insert` method. Now, we're going to study *index-based deletion*, implementing the following method:

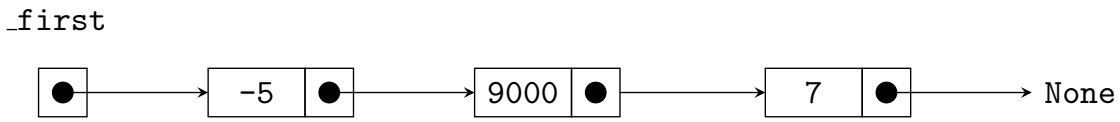
```
class LinkedList:
    def pop(self, i: int) -> Any:
        """Remove and return the item at index i.

        Raise IndexError if i >= len(self).

        Preconditions:
            - i >= 0

        >>> lst = LinkedList([1, 2, 10, 200])
        >>> lst.pop(1)
        2
        >>> lst.to_list()
        [1, 10, 200]
        >>> lst.pop(2)
        200
        >>> lst.pop(0)
        1
        >>> lst.to_list()
        [10]
        """
```

1. As we did last class, before implementing this method we'll look at some diagrams representing example inputs.

(a) Modify the linked list diagram below to show what happens when we call `LinkedList.pop` on it with `i == 2`.



(b) Repeat, but with `i == 0` instead.



2. Implement `LinkedList.pop`. As with `LinkedList.insert`, use the linked list traversal pattern with a parallel `curr_index` variable. You should have two cases based on whether `self._first` needs to be mutated or not.

```
class LinkedList:
    def pop(self, i: int) -> Any:
        """Remove and return the item at index i.

        Raise IndexError if i >= len(self).

        Preconditions:
            - i >= 0

        >>> lst = LinkedList([1, 2, 10, 200])
        >>> lst.pop(1)
        2
        >>> lst.to_list()
        [1, 10, 200]
        >>> lst.pop(2)
        200
        >>> lst.pop(0)
        1
        >>> lst.to_list()
        [10]
        """
        if i == 0:
            if self._first is None:
                # The list is empty. There is no 0th element
                raise IndexError
            else:
                # self._first is a _Node
                curr = self._first
                self._first = self._first.next
        else:
            curr == self._first
            curr_index = 0

            while not (curr is None or curr_index == i - 1)
                curr = curr.next
                curr_index += 1

```

```
            if curr is None:
                raise IndexError
            else:
                if curr.next is None:
                    # There is no node at index i
                    raise IndexError
                else:
                    # curr_index == i - 1; curr is the node at index i - 1
                    item = curr.next.item
                    curr.next = curr.next.next
                    return item
```

2 Exercise 2: Value-based deletion

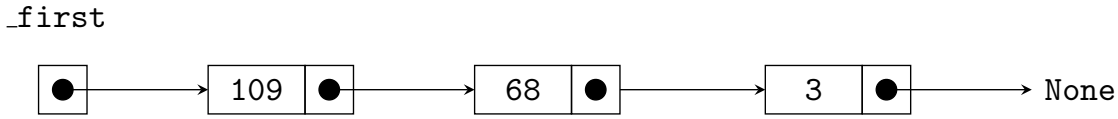
The built-in `list` data type has another type of deletion: `list.remove`, which removes the first occurrence of an item. This is known as *value-based deletion*, since the item that's removed is based on its value rather than its index.

We will now implement an equivalent `LinkedList.remove` method:

```
class LinkedList:
    def remove(self, item: Any) -> None:
        """Remove the first occurence of item from the list.

        Raise ValueError if the item is not found in the list.
        """
```

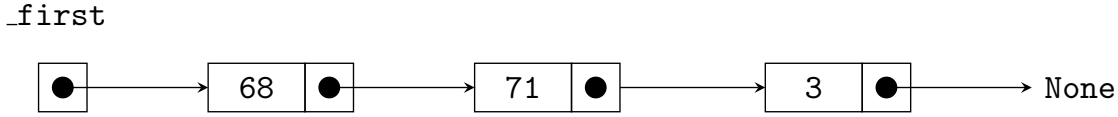
1. Suppose you want to remove `71` from the linked list below.

   (a) Modify the diagram above to show how the linked list would change when `71` is removed.



   (b) Which node needed to be mutated?
   The node at index `1` needs to be mutated (i.e. the node before `71`)

   (c) Here is the same linked list. Suppose we now wanted to remove `109`. Modify the diagram to show how the linked list needs to change.



2. Using these ideas, implement the `LinkedList.remove` method.

```python
class LinkedList:
    def remove(self, item: Any) -> None:
        """Remove the first occurence of item from the list.

        Raise ValueError if the item is not found in the list.
        """
        prev, curr = None, self._first

        while curr is not None:
            if curr.item == item:
                if prev is None:  # curr is self._first
                    self._first = curr.next
                    return
                else:  # curr is not self._first
                    prev.next = curr.next
                    return

            prev, curr = curr, curr.next

        raise ValueError
```

3 Exercise 3: Running time of linked list operations

1. Suppose we have a Python `list` of length 1,000,000, and a `LinkedList` of length 1,000,000.

   (a) If we want to *access* the item at index 500,000 in each list, would it be significantly faster for the `list`, the `LinkedList`, or about the same amount of time for both? Explain.
   Our `LinkedList` traversal pattern that we use for indexing has a running time of $\Theta(i)$ where i is the requested index. This is much slower than the build in `list` indexing method which is $\Theta(1)$. In this case, `list` would be 500,000 times faster than `LinkedList`.

   (b) If we want to *delete* the item at index 0 in each list, would it be significantly faster for the `list`, the `LinkedList`, or about the same amount of time for both? Explain.
   It would be about the same amount of time. Removing the first index does not require the use of our `LinkedList` traversal pattern. This means that the operation would have a running time of $\Theta(1)$. For the build in `list`, the running time is $\Theta(n)$. In this case, `LinkedList` would be 1,000,000 times faster than `list`.

   (c) If we want to *insert* an item at index 500,000 in each list, would it be significantly faster for the `list`, the `LinkedList`, or about the same amount of time for both? Explain.
   In this case, running time would be very similar. The `LinkedList` traversal pattern would need to be used meaning the running time would be $\Theta(\frac{n}{2})$. The `list.pop()` method would also be $\Theta(\frac{n}{2})$.

2. Consider our current implementation of `LinkedList.__init__`, which uses the `append` method:

```python
class LinkedList:
    def __init__(self, items: Iterable) -> None:
        self._first = None
        for item in items:
            self.append(item)
```

   (a) What is the running time of `LinkedList.append`, in terms of `n`, the length of the linked list?
   $RT_{\text{append}} \in \Theta(n)$. This is because we need to use the linked list traversal pattern to get to the end of the list then set the last `_Node.next` to be the new item that we are appending.

(b) Analyze the running time of this implementation of `LinkedList.__init__`, in terms of n, the length if `items`.

Remember that when taking into account the running time of a helper function, you can drop the "Theta", e.g. count a $\Theta(n)$ function as n steps.

The for loop iterates n times.

The loop body takes m steps where m is the current length of self which is changing over time. It increases by 1 each iteration.

So at iteration i, loop body takes i steps.

So then finally, $RT_{\_\_\text{init}\_\_} = \sum_{i=0}^{n-1} i = \frac{n \cdot (n-1)}{2} \in \Theta(n^2)$