11.5 Linked List Running-Time Analysis

To wrap up the discussion of linked lists, we return to our original motivation to studying linked lists: improving the efficiency of some of the basic list operations. To begin, let us review the running time of three key operations for the built-in Python `list` class (where n is the size of the list):

Operation (assuming $0 \leq i < n$)	**Running time**
Indexing (`lst[i]`)	$\Theta(1)$
Insert into index i	$\Theta(n - i)$
Remove item at index i	$\Theta(n - i)$

What about the corresponding operations for `LinkedList`?

Running-time analysis of `LinkedList.insert`

Let's study our code for `LinkedList.insert`.

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

        if i == 0:
            self._first, new_node.next = new_node, self._first
        else:
            curr = self._first
            curr_index = 0

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

            if curr is None:
                raise IndexError
            else:
                curr.next, new_node.next = new_node, curr.next
```

Running-time analysis. Let n be the length (i.e., number of items) of `self`.

Because the code is split into two different cases with the outer if statement, we'll organize our running-time analysis into cases as well.

Case 1: Assume `i == 0`. In this case, the if branch executes, which takes constant time, so we'll count it as one step.

Case 2: Assume `i > 0`. In this case:

- The first two statements in the else branch (`curr = self._first`, `curr_index = 0`) take constant time, so we'll count them as 1 step.
- The statements after the while loop all take constant time, so we'll count them as 1 step.
- The while loop iterates until either it reaches the end of the list (`curr is None`) or until it reaches the correct index (`curr_index == i - 1`).
    - The first case happens after n iterations, since `curr` advances by one `_Node` each iteration.
    - The second case happens after $i - 1$ iterations, since `curr_index` starts at 0 and increases by 1 each iteration.

  So the number of iterations taken is $\min(n, i - 1)$.

  Each iteration takes 1 steps, for a total of $\min(n, i - 1)$ steps.

This gives us a total running time of $1 + \min(n, i - 1) + 1 = \min(n, i - 1) + 2$ steps.

Putting it all together. So in the first case (when $i = 0$), we have a running time of $\Theta(1)$. In the second case (when $i > 0$), we have a running time of $\Theta(\min(n, i))$.[1] The second

[1] Note that $\min(n, i - 1) \in \Theta(\min(n, i))$, since $i - 1 \in \Theta(i)$.

expression also becomes $\Theta(1)$ when $i = 0$,[2] and so we can say that the overall running time

[2] You might notice that $\min(n, i) = 0$ and $\Theta(0)$ is not the same thing as $\Theta(1)$. This is a technical detail that's not too important, but essentially for running time analysis we assume that every piece of code takes at least 1 step, and so our Theta expressions always have an implicit "$\max(\ldots, 1)$" to ensure they are greater than 0.

of `LinkedList.insert` is $\Theta(\min(n, i))$.

Comparing `LinkedList` and `list` running times

If we want to compare this analysis against the above running time for the built-in `list.insert` method, we can assume that $0 \leq i < n$, in which case $\min(n, i) = i$, and we get that the running time of `LinkedList.insert` is $\Theta(i)$. This is a bit subtle, since the n (our traditional variable for "input size") has disappeared from the Theta expression! The Theta expression $\Theta(\min(n, i))$ is the most general expression we could write, without assuimng

any relationship between the length of the linked list and the given index i. Our expression $\Theta(i)$ is a simplification *under an additional assumption* that $i < n$. Essentially, we say that *if* we treat i as small with respect to the size of the list, then the running time of the algorithm does not depend on the size of the list.[3]

[3] The most extreme case of this is when `i == 0`, so we're inserting into the front of the linked list. This takes *constant time*, meaning it does not depend on the length of the linked list.

The running times of `LinkedList.__getitem__` and `LinkedList.pop` follow the same analysis, since they also involve using a loop to reach the node at the correct index. So we can now add to our table from the start of this section:

Operation (assuming $0 \leq i < n$)	**Running time (`list`)**	**Running time (`LinkedList`)**
Indexing (`lst[i]`)	$\Theta(1)$	$\Theta(i)$
Insert into index i	$\Theta(n - i)$	$\Theta(i)$
Remove item at index i	$\Theta(n - i)$	$\Theta(i)$

So for list indexing, linked lists have a worse performance: the running time is proportional to the index being accessed, rather than being constant time.[4] For insertion

[4] Remember that constant-time indexing is the main benefit of array-based list implementations, taking advantage of references to list elements being stored in continguous memory locations.

and deletion, linked lists have the exact opposite running times as array-based lists! In particular, inserting into the *front* of a linked list takes $\Theta(1)$ time, and inserting into the *back* of a linked list takes $\Theta(n)$ time, where n is the length of the list.

This may seem disappointing, because now it isn't clear which list implementation is "better." But in fact this is pretty typical of computer science: when creating multiple implementations of a public interface, each implementation will often be good at some operations, but worse at others. In practice, it is up to the programmer who is acting as a *client* of the interface to decide which implementation to use, based on how they prioritize the efficiency of certain operations over others. If you want constant time indexing and mainly will add/remove elements at the end of the list, prefer a `list`. If you plan to mainly add/remove elements at the front of the list, prefer a `LinkedList`.[5]

[5] In your tutorial, you'll learn about a more complex form of linked list which allows for fast updates at both their front and end. Stay tuned for that!