4.4 Linked Lists and Running Time

MOTIVATION TO STUDYING LINKED LISTS:

- Improving the efficiency of some of the basic list operations

RUNNING TIME OF THREE OPERATIONS OF ARRAY-BASED LISTS:

- Looking up an element of the list by the index:
    - Takes constant time (independent of length of list/index we are looking up)
    - O(1)
- Inserting/removing an element at index (0 <= i <= n):
    - List of length n takes time proportional to n - i
    - O(n - i)
    - Inserting/Removing at the front of a list: O(n) -> time linear in length of the list
    - Inserting/Removing at the end of a list: O(1)

LINKED LISTS:

```
def insert(self, index: int, item: Any) -> None:
    # Create a new node
    new_node = _Node(item)

    # Need to do something special if we insert into the first position.
    # In this case, self._first *must* be updated.
    if index == 0:
        new_node.next = self._first
        self._first = new_node
    else:
        # Get the node at position (index - 1)
        curr_index = 0
        curr = self._first
        while curr is not None and curr_index < index - 1:
            curr = curr.next
            curr_index = curr_index + 1

        if curr is None:
            raise IndexError
        else:
            # At this point, curr refers to the node at position (index - 1)
            curr.next, new_node.next = new_node curr.next
```

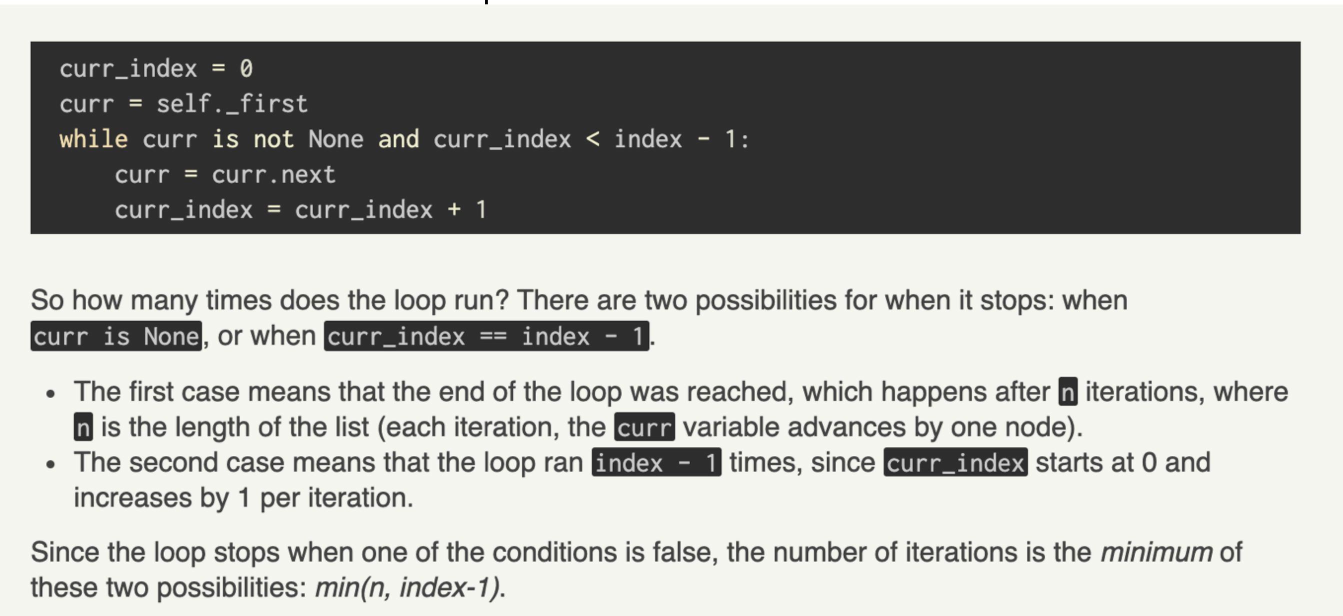
- When index == 0:
    - The branch that executes takes constant time as *both assignments are*

  independent of list's length

- Else (Inserting item at the end of the linked list):
  - The *loop must iterate till it reaches the end* of the list -> linear time
- **Linked lists have the exact opposite running-times as array_based lists !!!**
- Inserting into **front** of linked list : O(1) time
- Inserting into the **back** of linked list: O(n) time

CONSTANT TIME:

- Overall running time depends on the number of lines that execute -> depends on the number of times the loop runs



```
curr_index = 0
curr = self._first
while curr is not None and curr_index < index - 1:
    curr = curr.next
    curr_index = curr_index + 1
```

So how many times does the loop run? There are two possibilities for when it stops: when `curr is None`, or when `curr_index == index - 1`.

- The first case means that the end of the loop was reached, which happens after `n` iterations, where `n` is the length of the list (each iteration, the `curr` variable advances by one node).
- The second case means that the loop ran `index - 1` times, since `curr_index` starts at 0 and increases by 1 per iteration.

Since the loop stops when one of the conditions is false, the number of iterations is the *minimum* of these two possibilities: *min(n, index-1)*.

CONSIDER:

- Whenever we write a Big-Oh expression to capture the fact that the running time can't drop below 1
- The body of the loop takes assignment statements -> constant time