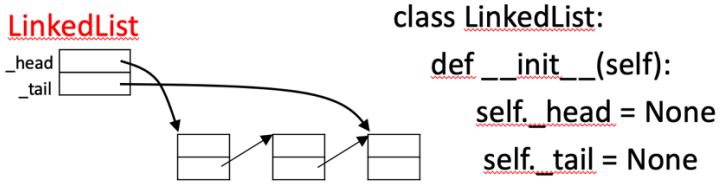
Work with your neighbor. (This will be graded for participation only.)

1. The following definition for the `LinkedList` class includes an attribute for a reference to the tail of the linked list, as shown in the diagram:




```
class LinkedList:
    def __init__(self):
        self._head = None
        self._tail = None
```

To properly maintain the tail reference whenever a linked list is modified, the linked list methods have to be revisited and possibly modified. For example, we just saw in lecture the changes needed two concatenate two lists.

The method `append(self, new)`, appends Node `new` to the end of a linked list. Rewrite the method to incorporate the changes needed to update the tail reference.

ANS:

```
def append(self, new):
    if self._head == None:
        self._head = new
        self._tail = new
    else:
        self._tail._next = new
        self._tail = new
```

Why is the complexity of `append()` using a linked list with a tail reference O(1)?

ANS:

The linked list class maintains a reference to the tail of the list; you do not need to traverse the linked list from the beginning in order to find the end (tail) of the list. The operations in the if and else clauses above are all O(1), making the method `append()` O(1).

2. Using the `LinkedList` class above with a tail reference, answer these questions:

   a) What is the complexity of a method `remove_last(self)` that removes the last element of the list? Explain your answer.

   ANS:

   To remove the last element of the list, we need to set `self._tail` to reference of the second-to-last node in the list. You need to traverse the entire list (except the last node) in order to find the second-to-last node, which makes this method O(n).

   b) Could we somehow change the `LinkedList` and/or `Node` classes to make `remove_last(self)` O(1)? Explain your answer.

   ANS:

   To make `remove_last(self)` O(1), we would need to know—for every node—what the previous node is. We would need to modify the `Node` class to have an attribute for the previous node and change all of the `Node` methods to maintain the reference to the previous node.

   NOTE: Linked lists with next and previous attributes are called Doubly Linked Lists.

3. Another famous sorting algorithm is Quicksort. This sorting algorithm first partitions the list to be sorted into two separate lists by comparing each element to a "pivot" value. Values less than or equal to the pivot value are placed in one list, and values greater than the pivot are placed in a second list.

   Write a function `partition(alist, pivot)` that partitions a *built-in Python* list `alist` according to the integer pivot value `pivot`. Create two new lists and return them in a tuple. What is the complexity of your partition function?

   ANS:

```
def partition(alist, pivot):

    smaller = []
    larger = []

    for elem in alist:
        if elem <= pivot:
            smaller.append(elem)
        else:
            larger.append(elem)

    return (smaller, larger)
```

   The for loop will iterate n times, where n is the size of the list. The statements in the body of the loop are all O(1); as a result, this function is O(n).

Wait until we do Hashing before continuing.

4. Implement the Dictionary ADT as described in the lecture slides, with this simplification: the dictionary is created as a fixed size, which is given when the Dictionary ADT is created. See the Usage and Hint below to determine how to implement the ADT.

```
Usage:
>>> d = Dictionary(7)
>>>
>>> d.put('five', 5)
>>> d.put('three', 3)
```

Hint 1:

```
>>> d._pairs
[['five', 5], ['three', 3], None, None, None, None, None]
```

Hint 2:
You will need an attribute to keep track of the next available slot in the underlying list.

Hint 3:
If the Dictionary is created with capacity 7, the underlying list will look like this:

```
[None,None,None,None,None,None,None]
```

ANS:

```
class Dictionary:
    def __init__(self,capacity):
        # each element will be a key/value pair
        self._pairs = [None] * capacity
        self._nextempty = 0

    def put(self, k, v):
        self._pairs[self._nextempty] = [k,v]
        self._nextempty += 1

    def get(self, k):
        for pair in self._pairs[0:self._nextempty]:
            if pair[0] == k:
                return pair[1]
        return None
```