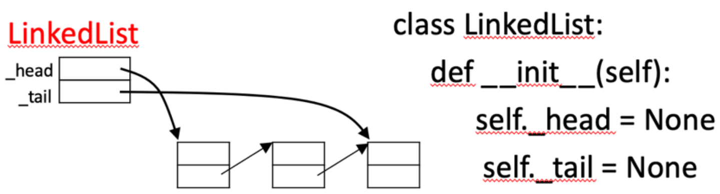
CSC 120 **ICA-32**

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.

What is the complexity of `append()` using a linked list with a tail reference? Explain your answer.

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.

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

3. We have seen the Mergesort algorithm. 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?

Wait until we have moved onto Hashing before doing this problem.

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.

   Use a Python list of a fixed size for the dictionary in your `init()` method. Use **`[None]*size`** to create the list.

   Write the `put(key,value)` and `get(key)` methods.

   See the Usage and Hints 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]`

```
class Dictionary:
    def __init__(self,capacity):
        # each element will be a key/value pair




    def put(self, k, v):








    def get(self, k):
```