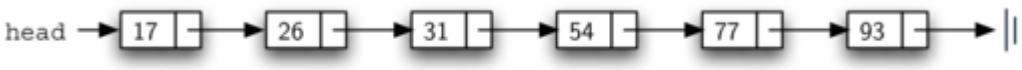
Ordered Lists

In order to implement the ordered list, we must remember that the relative positions of the items are based on some underlying characteristic. The ordered list of integers given above (17, 26, 31, 54, 77, and 93) can be represented by a linked structure as shown below. Again, the node and link structure is ideal for representing the relative positioning of the items.



To implement the `OrderedList` class, we will use the same technique as seen previously with unordered lists. Once again, an empty list will be denoted by a `head` reference to `None`

```
class OrderedList:
    def __init__(self):
        self.head = None
```

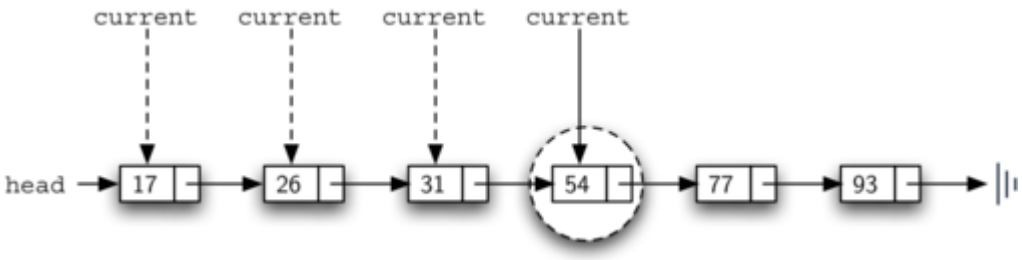
As we consider the operations for the ordered list, we should note that the `isEmpty` and `size` methods can be implemented the same as with **unordered lists (https://maryville.instructure.com/courses/43640/pages/unordered-lists)** since they deal only with the number of nodes in the list without regard to the actual item values. Likewise, the `remove` method will work just fine since we still need to find the item and then link around the node to remove it. The two remaining methods, `search` and `add`, will require some modification.

The Search Method for Ordered Lists

The search of an unordered linked list required that we traverse the nodes one at a time until we either find the item we are looking for or run out of nodes (`None`). It turns out that the same approach would actually work with the ordered list and in fact in the case where we find the item it is exactly what we need. However, in the case where the item is not in the list, we can take advantage of the ordering to stop the search as soon as possible.

For example, the picture below shows the ordered linked list as a search is looking for the value 45. As we traverse, starting at the head of the list, we first compare against 17. Since 17 is not

the item we are looking for, we move to the next node, in this case 26. Again, this is not what we want, so we move on to 31 and then on to 54. Now, at this point, something is different. Since 54 is not the item we are looking for, our former strategy would be to move forward. However, due to the fact that this is an ordered list, that will not be necessary. Once the value in the node becomes greater than the item we are searching for, the search can stop and return `False`. There is no way the item could exist further out in the linked list.



The code below shows the complete `search` method. It is easy to incorporate the new condition discussed above by adding another boolean variable, `stop`, and initializing it to `False` (line 4). While `stop` is `False` (not `stop`) we can continue to look forward in the list (line 5). If any node is ever discovered that contains data greater than the item we are looking for, we will set `stop` to `True` (lines 9–10). The remaining lines are identical to the unordered list search.

```
def search(self,item):
    current = self.head
    found = False
    stop = False
    while current != None and not found and not stop:
        if current.getData() == item:
            found = True
        else:
            if current.getData() > item:
                stop = True
            else:
                current = current.getNext()

    return found
```

Adding Items to an Ordered List

The most significant method modification will take place in `add`. Recall that for unordered lists, the `add` method could simply place a new node at the head of the list. It was the easiest point of

access. Unfortunately, this will no longer work with ordered lists. It is now necessary that we discover the specific place where a new item belongs in the existing ordered list.

Source: Problem Solving and Algorithms in Python (http://interactivepython.org/runestone/static/pythonds/index.html#) from Bradley Miller on www.interactivepython.org (http://interactivepython.org/runestone/static/pythonds/index.html#).