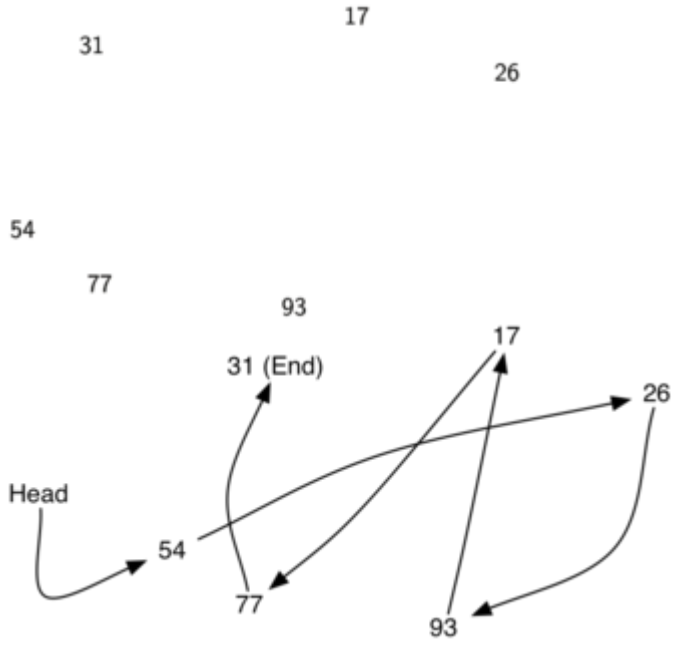
# Unordered Lists

In order to implement an unordered list, we will construct what is commonly known as a **linked list**. Recall that we need to be sure that we can maintain the relative positioning of the items. However, there is no requirement that we maintain that positioning in contiguous memory. For example, consider the collection of items shown below. It appears that these values have been placed randomly. If we can maintain some explicit information in each item, namely the location of the next item, then the relative position of each item can be expressed by simply following the link from one item to the next.




It is important to note that the location of the first item of the list must be explicitly specified. Once we know where the first item is, the first item can tell us where the second is, and so on. The external reference is often referred to as the **head** of the list. Similarly, the last item needs to know that there is no next item.

## The Node Class

The basic building block for the linked list implementation is the **node**. Each node object must hold at least two pieces of information. First, the node must contain the list item itself. We will call

this the **data field** of the node. In addition, each node must hold a reference to the next node. The code below shows the Python implementation. To construct a node, you need to supply the initial data value for the node. Evaluating the assignment statement below will yield a node object containing the value 93.

```
class Node:
    def __init__(self,initdata):
        self.data = initdata
        self.next = None

    def getData(self):
        return self.data

    def getNext(self):
        return self.next

    def setData(self,newdata):
        self.data = newdata

    def setNext(self,newnext):
        self.next = newnext
```

The special Python reference value `None` will play an important role in the `Node` class and later in the linked list itself. A reference to `None` will denote the fact that there is no next node. Note in the constructor that a node is initially created with `next` set to `None`. Since this is sometimes referred to as “grounding the node,” we will use the standard ground symbol to denote a reference that is referring to `None`. It is always a good idea to explicitly assign `None` to your initial next reference values.

# The Unordered List Class

As we suggested above, the unordered list will be built from a collection of nodes, each linked to the next by explicit references. As long as we know where to find the first node (containing the first item), each item after that can be found by successively following the next links. With this in mind, the `UnorderedList` class must maintain a reference to the first node. The code below shows the constructor. Note that each list object will maintain a single reference to the head of the list.

```
class UnorderedList:

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

Initially, when we construct a list, there are no items. As we discussed in the `Node` class, the special reference `None` will again be used to state that the head of the list does not refer to anything. Eventually, the example list given earlier will be represented by a linked list as shown below. The head of the list refers to the first node which contains the first item of the list. In turn, that node holds a reference to the next node (the next item) and so on. It is very important to note that the list class itself does not contain any node objects. Instead, it contains a single reference to only the first node in the linked structure.


The `isEmpty` method simply checks to see if the head of the list is a reference to `None`. The result of the boolean expression `self.head==None` will only be true if there are no nodes in the linked list. Since a new list is empty, the constructor and the check for empty must be consistent with one another. This shows the advantage to using the reference `None` to denote the “end” of the linked structure. In Python, `None` can be compared to any reference. Two references are equal if they both refer to the same object. We will use this often in our remaining methods.

```
def isEmpty(self):
    return self.head == None
```

## Adding Items to the List

So, how do we get items into our list? We need to implement the `add` method. However, before we can do that, we need to address the important question of where in the linked list to place the new item. Since this list is unordered, the specific location of the new item with respect to the

other items already in the list is not important. The new item can go anywhere. With that in mind, it makes sense to place the new item in the easiest location possible.

Recall that the linked list structure provides us with only one entry point, the head of the list. All of the other nodes can only be reached by accessing the first node and then following `next` links. This means that the easiest place to add the new node is right at the head, or beginning, of the list. In other words, we will make the new item the first item of the list and the existing items will need to be linked to this new first item so that they follow.

The linked list shown below was built by calling the `add` method a number of times.

```
>>> mylist.add(31)
>>> mylist.add(77)
>>> mylist.add(17)
>>> mylist.add(93)
>>> mylist.add(26)
>>> mylist.add(54)
```

Note that since 31 is the first item added to the list, it will eventually be the last node on the linked list as every other item is added ahead of it. Also, since 54 is the last item added, it will become the data value in the first node of the linked list.

## Removing Items from the List

The `remove` method requires two logical steps. First, we need to traverse the list looking for the item we want to remove. Once we find the item (recall that we assume it is present), we must remove it. The first step is very similar to `search`. Starting with an external reference set to the head of the list, we traverse the links until we discover the item we are looking for. Since we assume that item is present, we know that the iteration will stop before `current` gets to `None`. This means that we can simply use the boolean `found` in the condition.

When `found` becomes `True`, `current` will be a reference to the node containing the item to be removed. But how do we remove it? One possibility would be to replace the value of the item with some marker that suggests that the item is no longer present. The problem with this approach is the number of nodes will no longer match the number of items. It would be much better to remove the item by removing the entire node.

In order to remove the node containing the item, we need to modify the link in the previous node so that it refers to the node that comes after `current`. Unfortunately, there is no way to go

backward in the linked list. Since `current` refers to the node ahead of the node where we would like to make the change, it is too late to make the necessary modification.

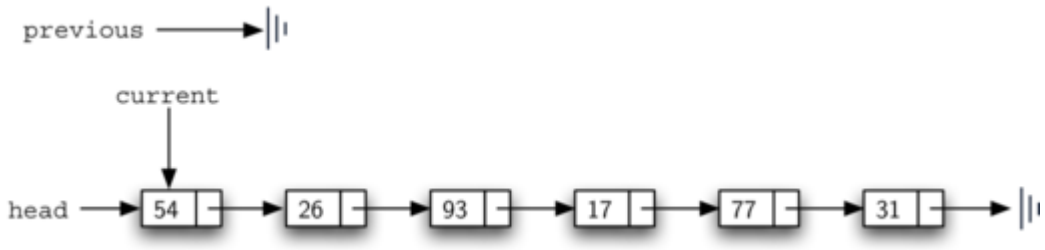
The solution to this dilemma is to use two external references as we traverse down the linked list. `current` will behave just as it did before, marking the current location of the traverse. The new reference, which we will call `previous`, will always travel one node behind `current`. That way, when `current` stops at the node to be removed, `previous` will be referring to the proper place in the linked list for the modification.

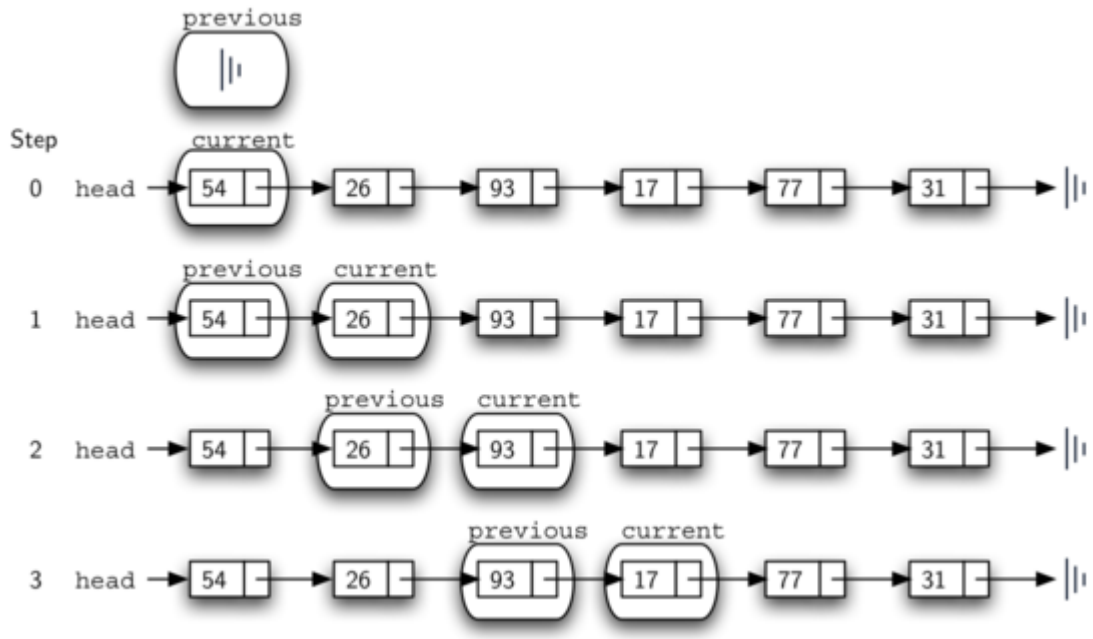
Let's look at the complete `remove` method. Lines 2–3 assign initial values to the two references. Note that `current` starts out at the list head as in the other traversal examples. `previous`, however, is assumed to always travel one node behind current. For this reason, `previous` starts out with a value of `None` since there is no node before the head. The boolean variable `found` will again be used to control the iteration.

In lines 6–7 we ask whether the item stored in the current node is the item we wish to remove. If so, `found` can be set to `True`. If we do not find the item, `previous` and `current` must both be moved one node ahead. Again, the order of these two statements is crucial. `previous` must first be moved one node ahead to the location of `current`. At that point, `current` can be moved. This process is often referred to as “inch-worming” as `previous` must catch up to `current` before `current` moves ahead.

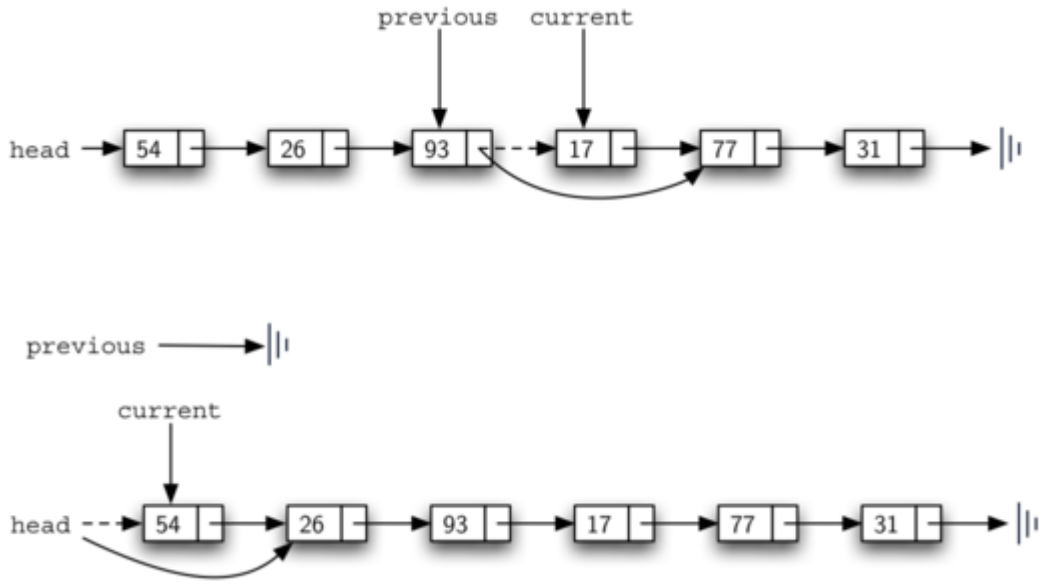
```
def remove(self,item):
    current = self.head
    previous = None
    found = False
    while not found:
        if current.getData() == item:
            found = True
        else:
            previous = current
            current = current.getNext()

    if previous == None:
        self.head = current.getNext()
    else:
        previous.setNext(current.getNext())
```






Once the searching step of the `remove` has been completed, we need to remove the node from the linked list. The graph below shows the link that must be modified. However, there is a special case that needs to be addressed. If the item to be removed happens to be the first item in the list, then `current` will reference the first node in the linked list. This also means that `previous` will be `None`. We said earlier that `previous` would be referring to the node whose next reference needs to be modified in order to complete the removal. In this case, it is not `previous` but rather the head of the list that needs to be changed.



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#).