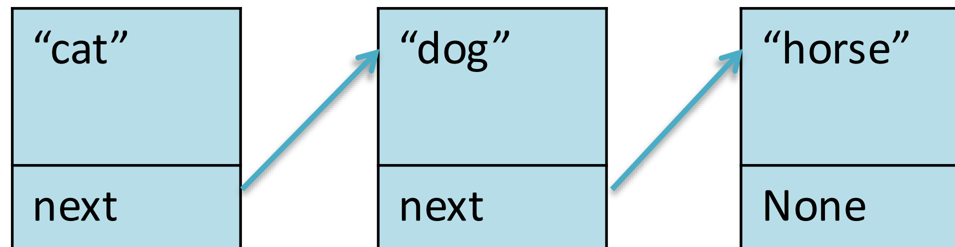
COSC 2306
Data Programming

Linked Lists

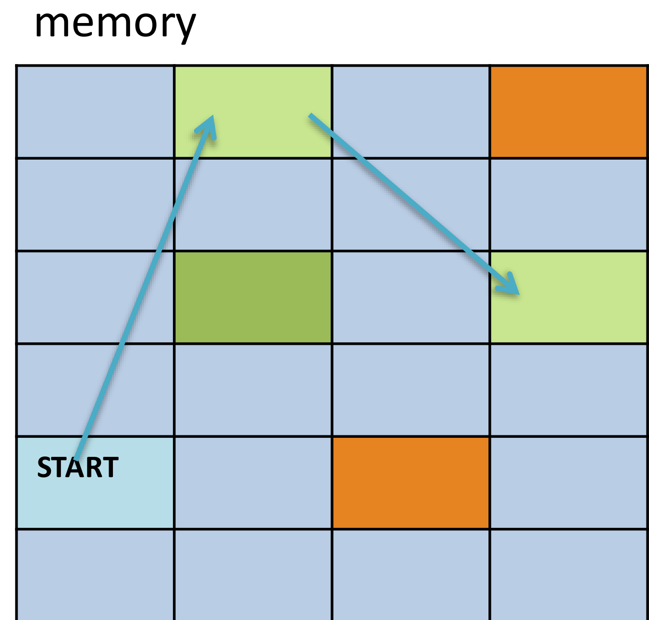
Linked Lists

- Collection of components (nodes)
- Every node (except last) contains access to next node
- Node components
  - **Data**: stores the actual data
  - **Link**: stores reference of next node



Pros and cons

- Pros: dynamic
  - No fixed limit to list size
  - No waste of space
- Cons:
  - **No random access**



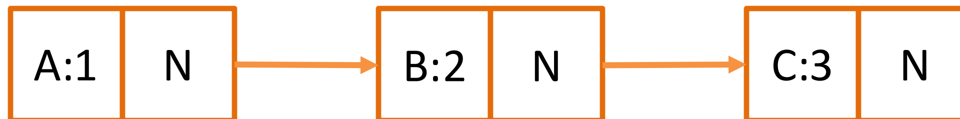
For one-way Linked List:
Start from the **head**
Can not go backwards

List node

```
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
```

Example of list node

- A = Node(1)
- B = Node(2)
- C = Node(3)
- A.next = B
- B.next = C
- print(A.next.data) #2
- print(A.next.next.data) #3



List class

When created, the list contains only the head (empty)

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

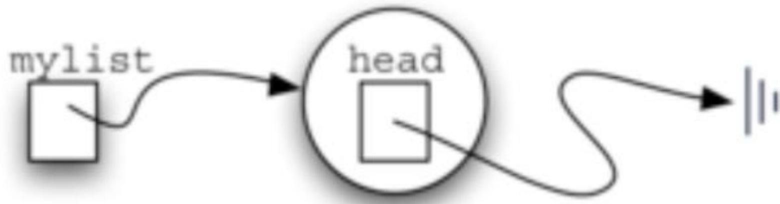


Figure 5: An Empty List

LinkedList class does not contain any node object, either empty or reference to the first node

Building a list (backward)

```
def add(self,item):
    temp = Node(item)
    temp.setNext(self.head)
    self.head = temp
```

Order very important! Why?

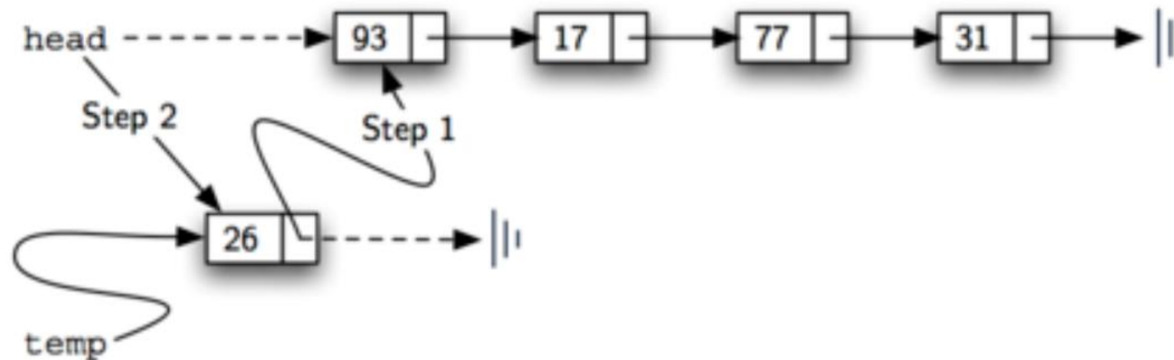


Figure 7: Adding a New Node is a Two-Step Process

Building a list (backward)

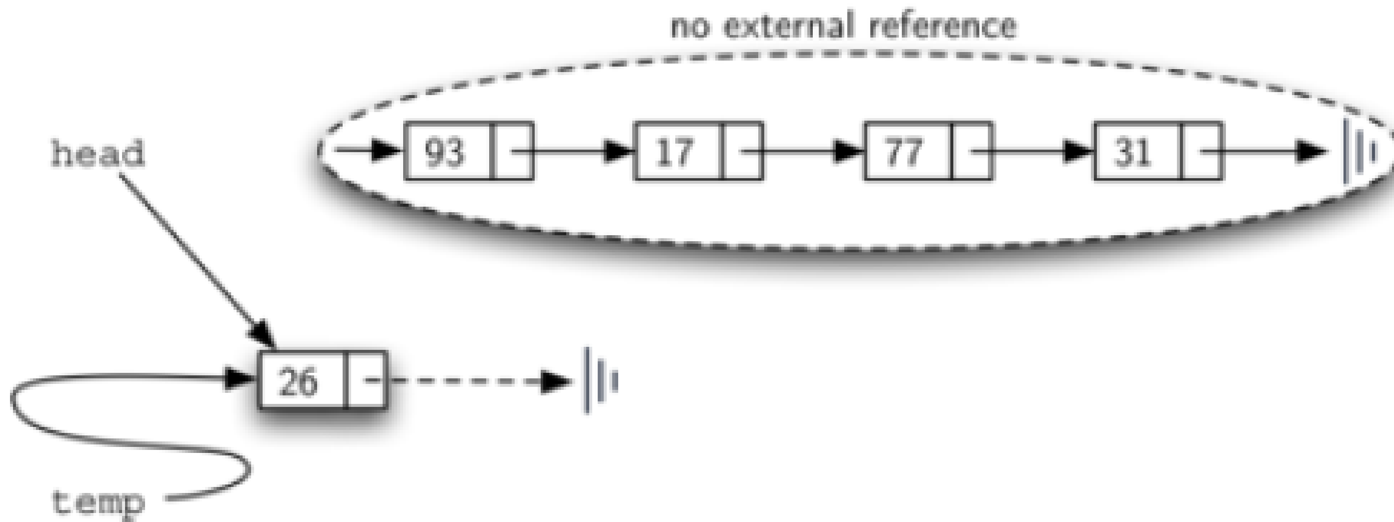


Figure 3.25: Result of Reversing the Order of the Two Steps.

Building a list (backward)

```
def add(self,item):
    temp = Node(item)
    temp.setNext(self.head)
    self.head = temp
```

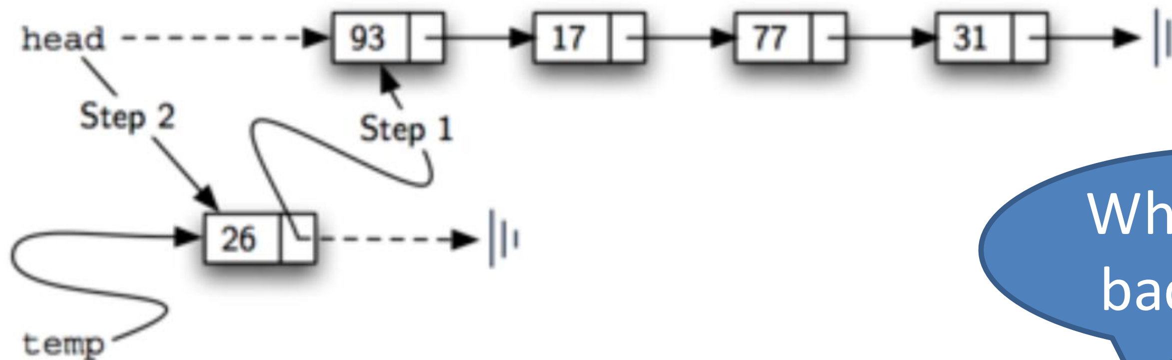


Figure 7: Adding a New Node is a Two-Step Process

Why adding backward?

Adding at head: O(1)
Adding at tail (traverse): O(n)

Linked list class

- Insert a new node
  - The new node always becomes the head

```
def insert(self, value):
    newnode = Node(value)
    newnode.next = self.head
    self.head = newnode
```

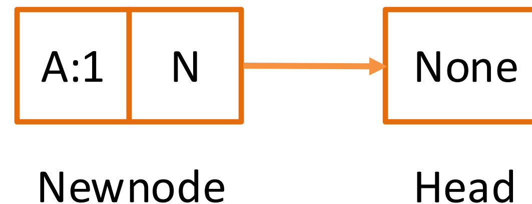
None

Head

Linked list class

- Insert a new node
  - The new node always becomes the head

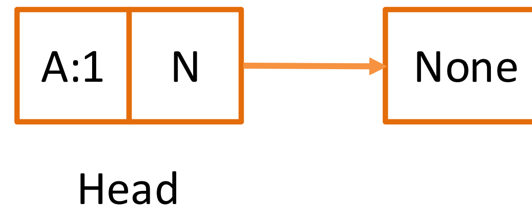
```
def insert(self, value):
    newnode = Node(value)
    newnode.next = self.head
    self.head = newnode
```



Linked list class

- Insert a new node
  - The new node always becomes the head

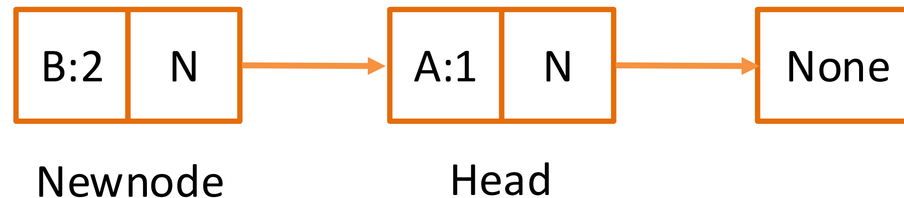
```
def insert(self, value):
    newnode = Node(value)
    newnode.next = self.head
    self.head = newnode
```



Linked list class

- Insert a new node
  - The new node always becomes the head

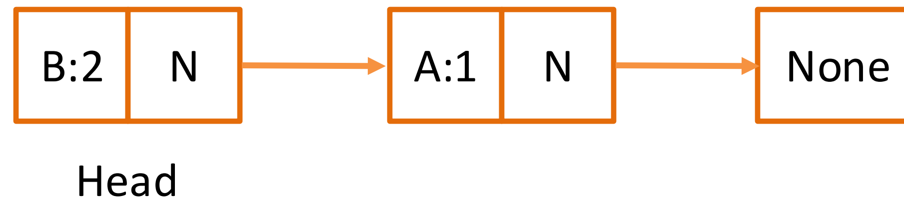
```
def insert(self, value):
    newnode = Node(value)
    newnode.next = self.head
    self.head = newnode
```



Linked list class

- Insert a new node
  - The new node always becomes the head

```
def insert(self, value):
    newnode = Node(value)
    newnode.next = self.head
    self.head = newnode
```



List – size (traversal)

Listing 5

```
1  def size(self):
2      current = self.head
3      count = 0
4      while current != None:
5          count = count + 1
6          current = current.getNext()
7
8      return count
```

External reference

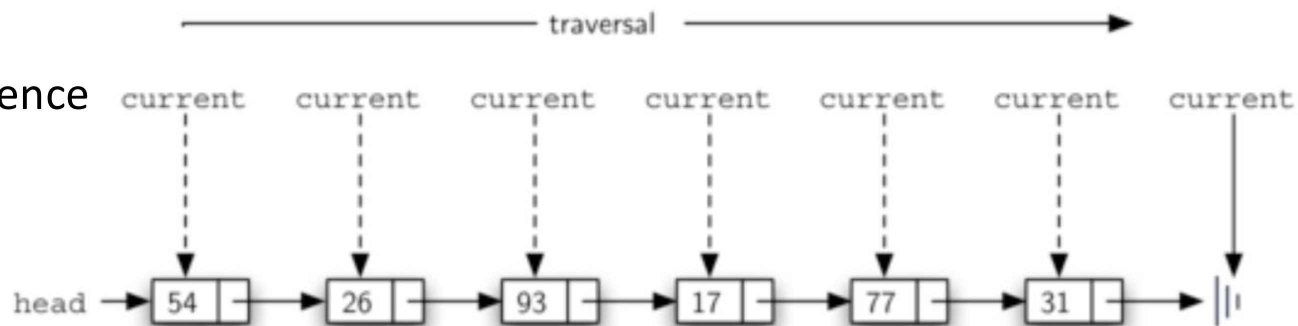


Figure 9: Traversing the Linked List from the Head to the End

Linked list with count

- Maintain a count for nodes

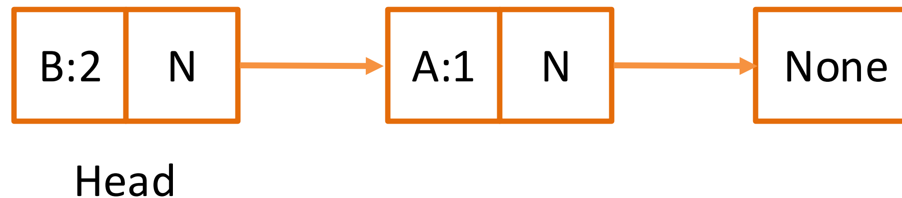
```
class LinkedList:

    def __init__(self):
        self.head = None
        self.count = 0

    def isEmpty(self):
        return self.count == 0

    def insert(self, value):
        newnode = Node(value)
        newnode.next = self.head
        self.head = newnode
        self.count += 1
```

```
    def size(self):
        return self.count
```

List – remove the head node

- Pop (remove) a node
  - Only remove the head node

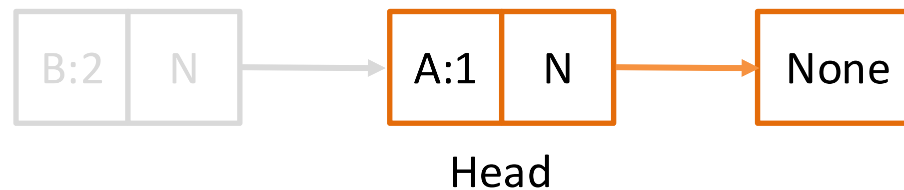
```
def pop(self):
    if self.isEmpty()
        return
    self.head = self.head.next
```



List – remove the head node

- Remove a node
  - First only remove (pop) the head node

```
def pop(self):
    if self.isEmpty()
        return
    self.head = self.head.next
```



Linked list with count

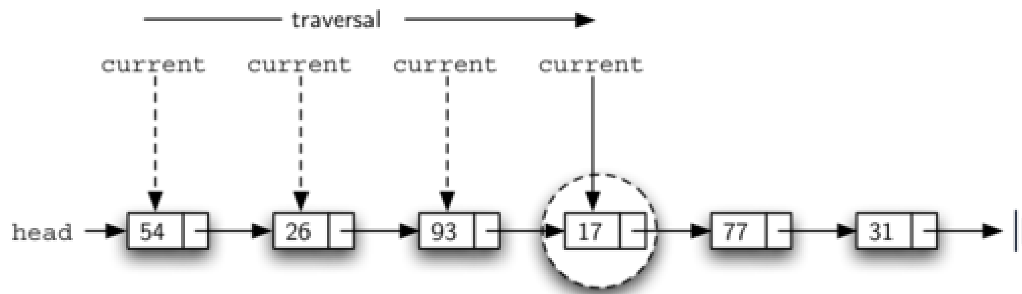
- Maintain a count for nodes

```
class LinkedList:


    def __init__(self):
        self.head = None
        self.count = 0

    def isEmpty(self):
        return self.count == 0

    def insert(self, value):
        newnode = Node(value)
        newnode.next = self.head
        self.head = newnode
        self.count += 1
```

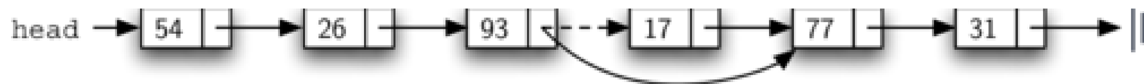
```
    def pop(self):
        if self.isEmpty():
            return
        self.head = self.head.next
        self.count -= 1

    def size(self):
        return self.count
```

List – remove a node

Step1: search the item

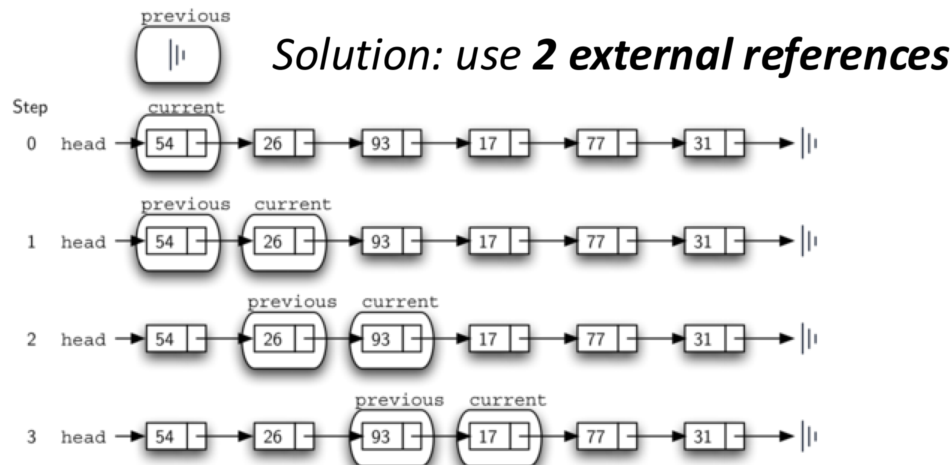


Step 2: remove the item



Challenge: cannot go backwards!

Solution: use ***2 external references***

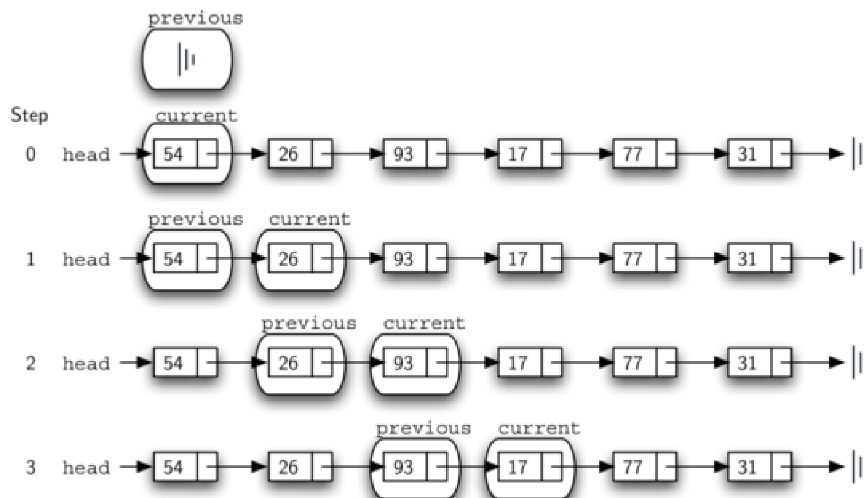


List – remove a node

```
1   def remove(self,item):
2       current = self.head
3       previous = None
4       found = False
5       while not found:
6           if current.getData() == item:
7               found = True
8           else:
9               previous = current
10              current = current.getNext()
11
12      if previous == None:
13          self.head = current.getNext()
14      else:
15          previous.setNext(current.getNext())
```

Step 1: search the item

Step 2: remove the item



Exercise -- append

Create append method for linked list

Hint: add to the tail of linked list

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


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

Exercise -- append

```
def append(self,item):
    current = self.head
    if current == None:
        self.head = Node(item)
    else:
        while current.next != None:
            current = current.next
        current.next = Node(item)
```

Exercise -- iter

- Create method to print data in linked list: __iter__ (hint: use generator)

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


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

Exercise -- iter

```
def __iter__(self):
        element = self.head
        while element is not None:
            yield element
            element = element.next

newlist = LinkedList()
newlist.append(1)
newlist.append(2)

iterlist = iter(newlist)
for item in iterlist:
   print(item.data)
>1
>2
```

Stack with Linked list

- Implement a stack with the Linked list class
  - tips: head is the stack top
  - push: insert
  - pop: pop (remove the head node)
  - peek: check the head node
  - size: check the count

Stack with Linked list

```
class StackwithLinkedList:

    def __init__(self):
        self.items = LinkedList()

    def isEmpty(self):
        return self.items.isEmpty()

    def push(self, item):
        self.items.insert(item)

    def pop(self):
        if self.isEmpty():
            return None
        return self.items.pop()

    def peek(self):
        if self.isEmpty():
            return None
        return self.items.head.data

    def size(self):
        return self.items.size()
```

Stack with Linked list

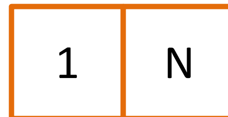
- st = StackwithLinkedList()

None

Head (top)

Stack with Linked list

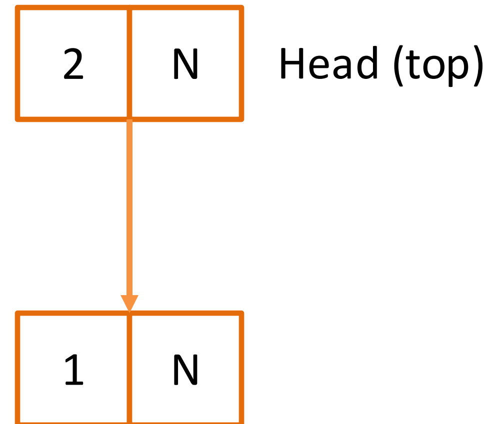
- st.push(1)
  - Add Node(1) to the head
- st.push(2)
  - Add Node(2) to the head



Head (top)

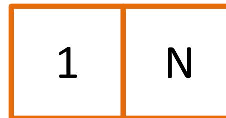
Stack with Linked list

- st.push(1)
  - Add Node(1) to the head
- st.push(2)
  - Add Node(2) to the head



Stack with Linked list

- st.pop()
  - Remove the head and return its data



Head (top)