MASSEY UNIVERSITY
TE KUNENGA KI PŪREHUROA
UNIVERSITY OF NEW ZEALAND

Computational Thinking and Algorithms
159.172
Linked Lists

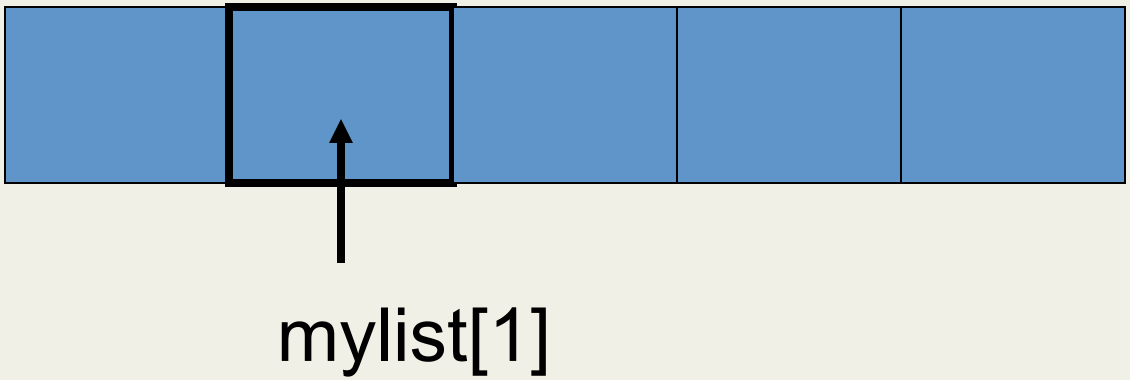
Amjed Tahir

a.tahir@massey.ac.nz

Previous contributors: Catherine McCartin

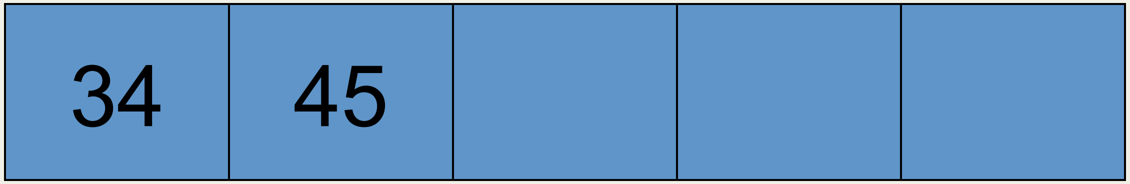
Linked lists

A Python list, implemented using an array, provides **constant time** access to a cell by specifying its index.



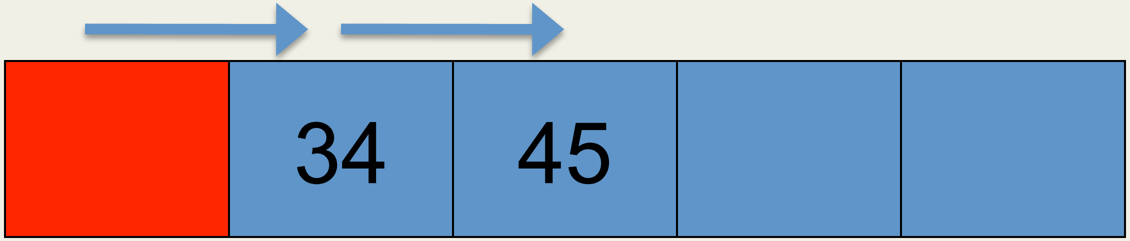
Linked lists

Inserting at the beginning of an array with n elements takes time **O(n)**, all elements must move over by one.



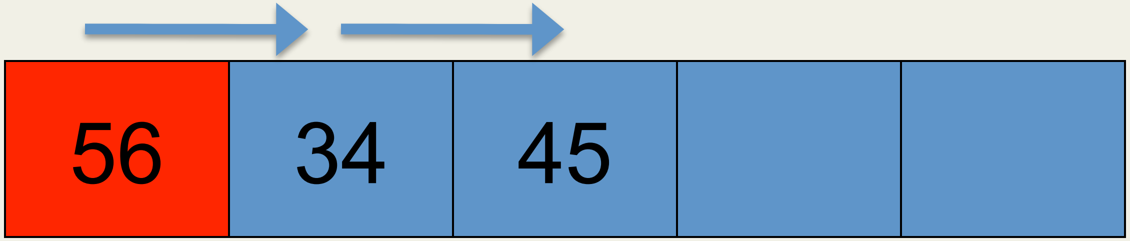
Linked lists

Inserting at the beginning of an array with n elements takes time **O(n)**, all elements must move over by one.



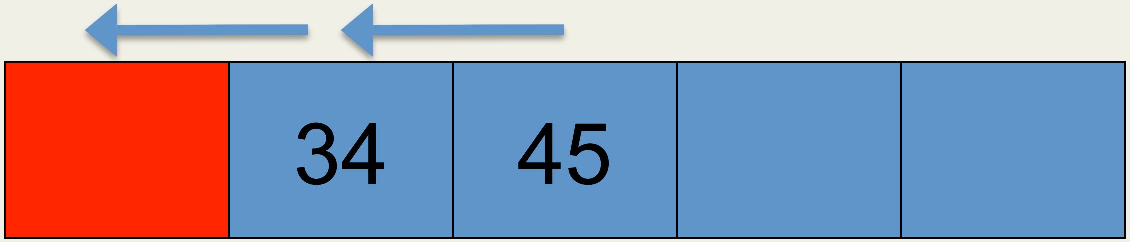
Linked lists

Inserting at the beginning of an array with n elements takes time **O(n)**, all elements must move over by one.



Linked lists

Deleting the first element of an array with n elements also takes time **O(n)**, all elements must move **back** by one.

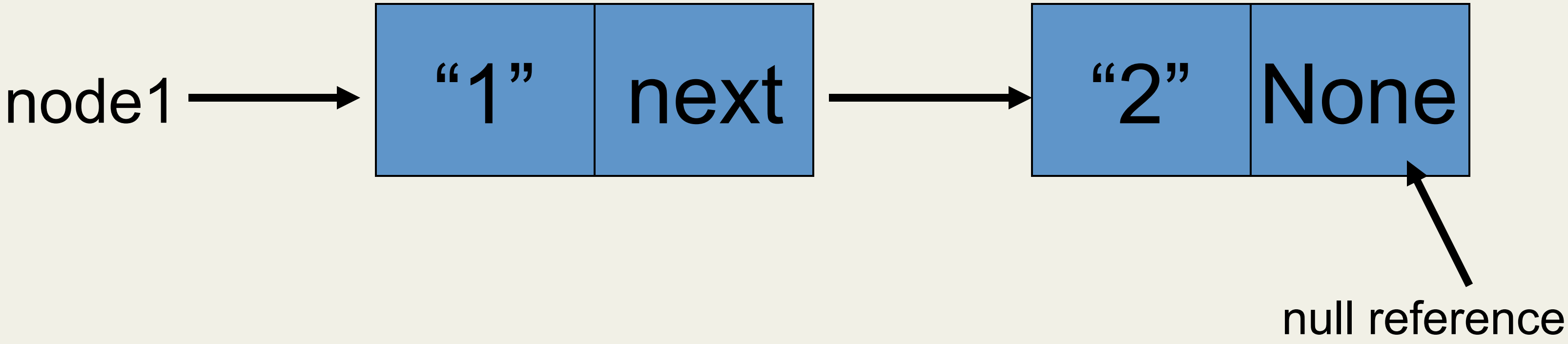


Linked lists

A linked list provides **constant time** insertions and deletions, but accessing an element can take **O(n)** time in the worst case.

We use a node structure with a data field (sometimes called the data/cargo) and a **next** field that references the next node in the list.

The next field of the last node has the special value **null** (None).



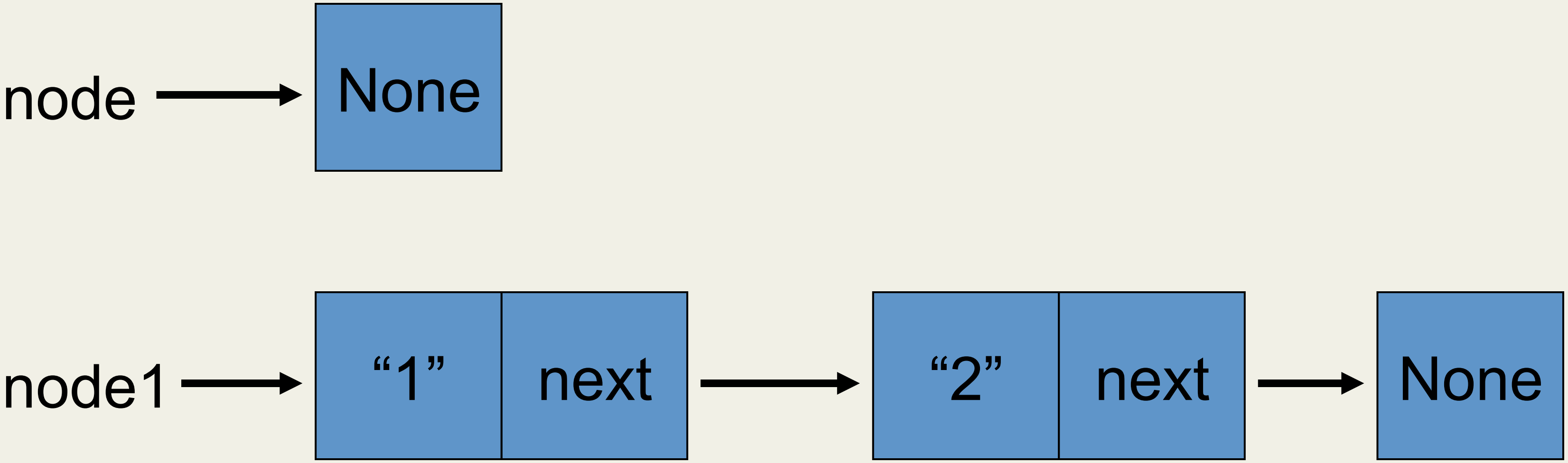
Linked lists

A linked list is a **recursive data structure**.

A linked list is either:

– the empty list, represented by None, or

– a node that contains a cargo object and a reference to a linked list.

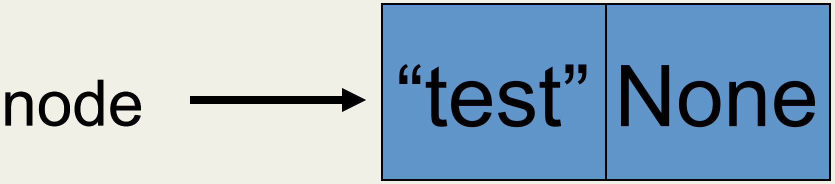


Python Node class

```
class Node:
  def __init__(self, cargo=None, next=None):
  # optional parameters
    self.cargo = cargo
    self.next  = next

  def __str__(self):
  #defines a string representation of a Node object
    return str(self.cargo)
```

Python Node class

```
>>> node = Node("test")
>>> print (node)
test
```



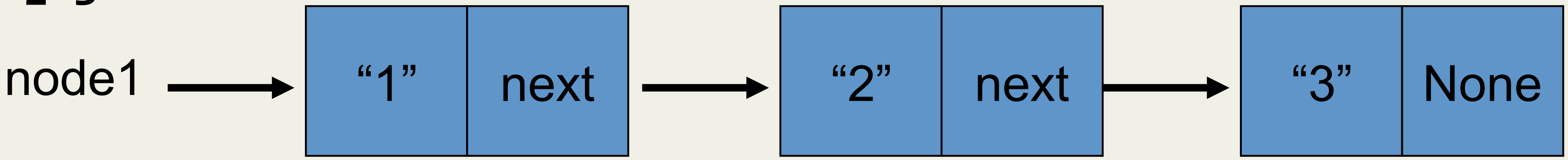
traversing a list

start at the "head" of the list, traverse it until we reach a nil reference

```
def traverseList(node):
  current_node = node
  while current_node is not None:
    print (current_node)
    current_node = current_node.next
```

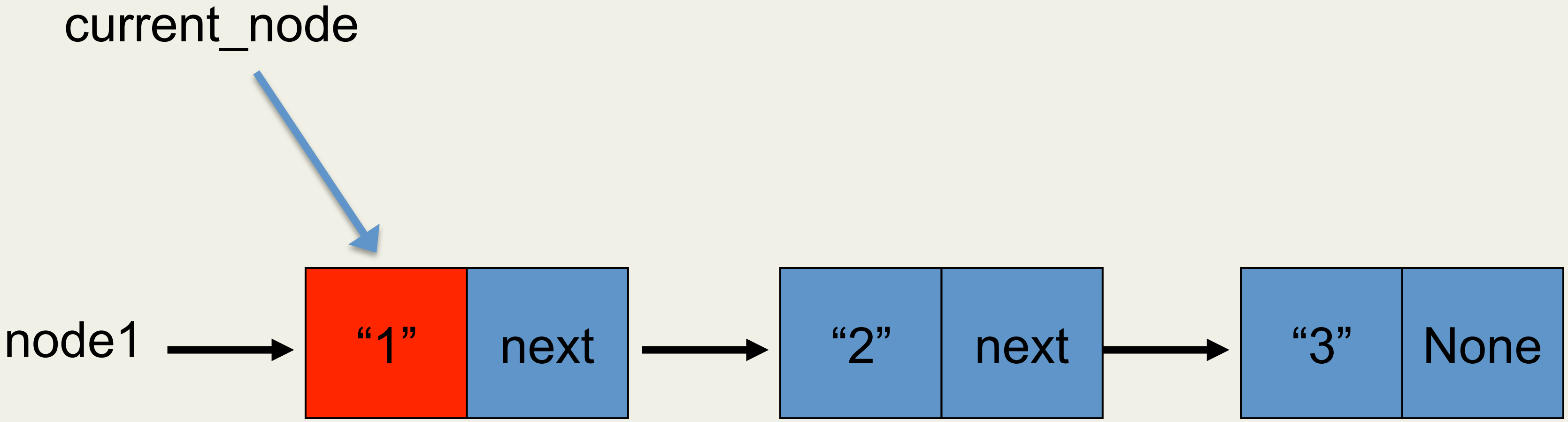
to invoke this function, we pass a reference to the first node:

```
>>> traverseList(node1)
1 2 3
```



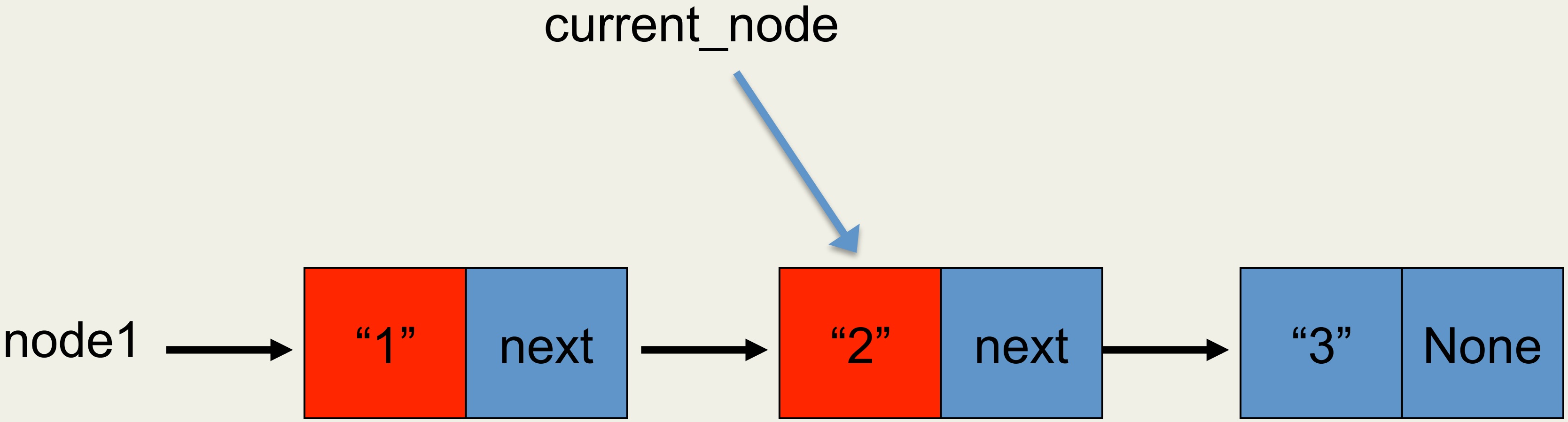
traversing a list

```
def traverseList(node):
  current_node = node
  while current_node is not None:
      print(current_node)
      current_node = current_node.next
```



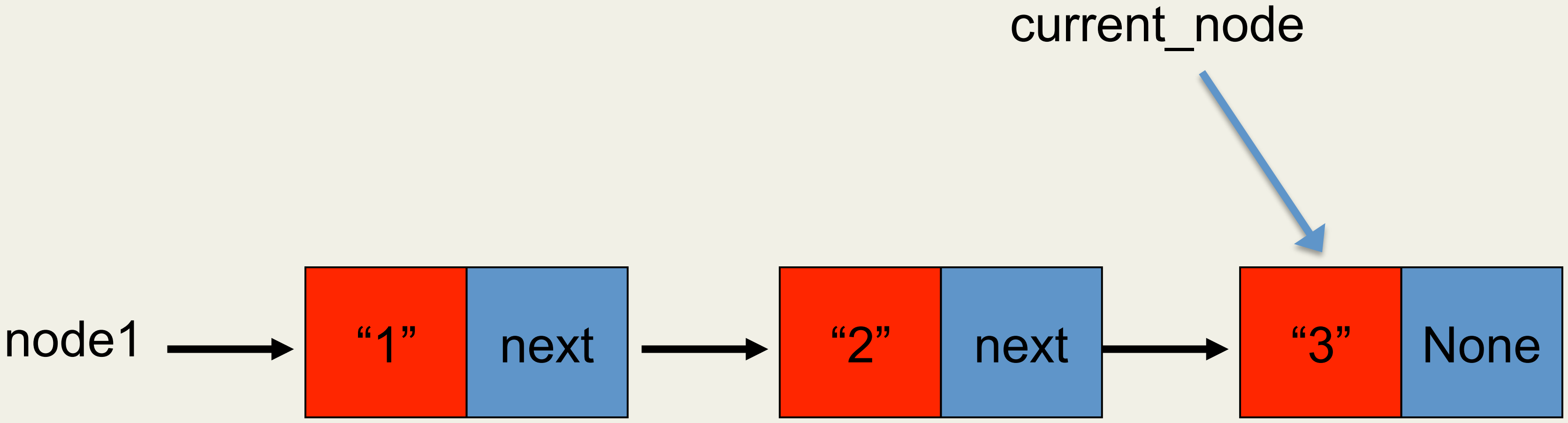
traversing a list

```
def traverseList(node):
   current_node = node
   while current_node is not None:
       print(current_node)
       current_node = current_node.next
```



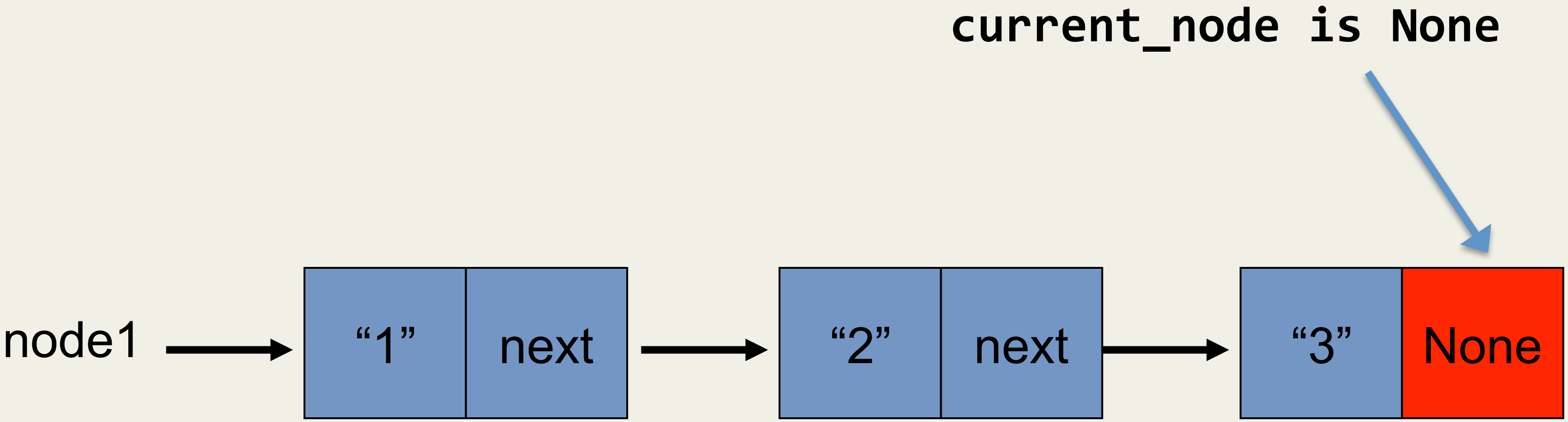
traversing a list

```
def traverseList(node):
  current_node = node
  while current_node is not None:
      print(current_node)
      current_node = current_node.next
```



traversing a list

```
def traverseList(node):
  current_node = node
  while current_node is not None:
    print(current_node)
    current_node = current_node.next
```



Lists and recursion

To print a list backwards:

1. separate the list into two pieces:

   the first node (called the head); and the rest (called the tail).

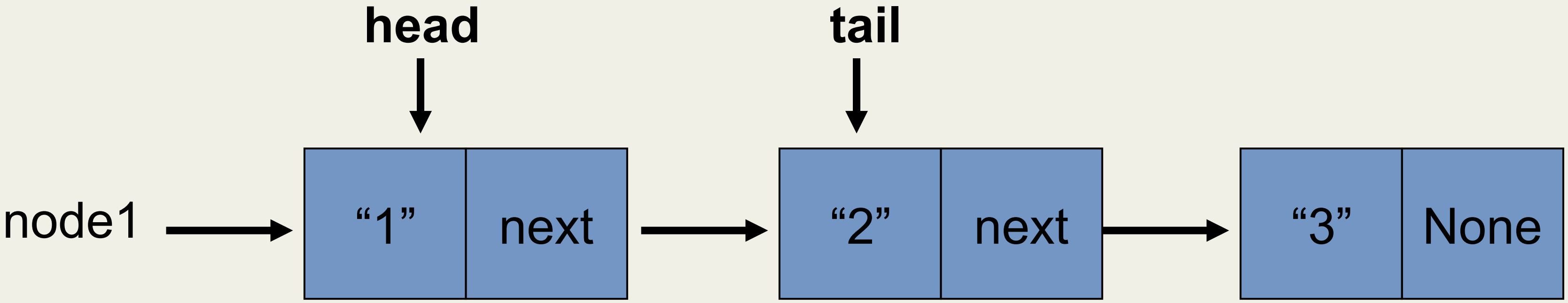
1. print the tail backward.

2. print the head.

```
def printBackward(thelist):
    if thelist == None:    # base case
       return
    head = thelist
    tail = thelist.next
    printBackward(tail)    # recursive call
    print (head)
```

Lists and recursion

```
def printBackward(thelist):
  if thelist == None:   # base case
    return
  head = thelist
  tail = thelist.next
  printBackward(tail)   # recursive call
  print (head, end= '')

>>> printBackward(node1)
3 2 1
```



head

tail

node1

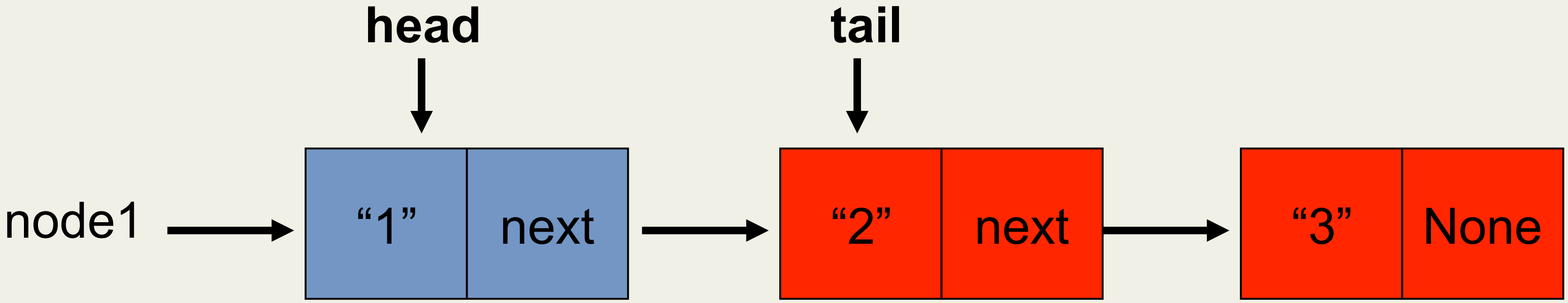
“1” next

“2” next

“3” None

Lists and recursion

```
def printBackward(thelist):
  if thelist == None:    # base case
     return
  head = thelist
  tail = thelist.next
  printBackward(tail)    # recursive call
  print(head, end= ' ')

>>> printBackward(node1)
3 2 1
```



head

tail

node1

“1” next

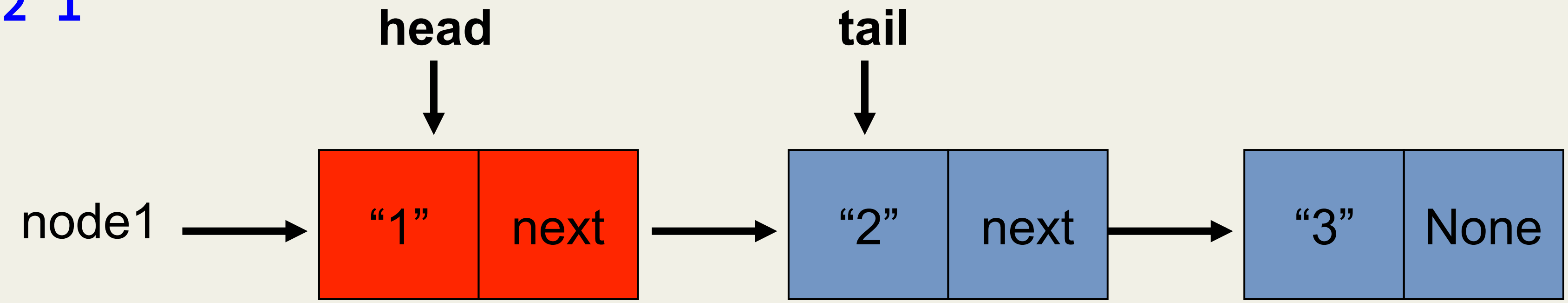
“2” next

“3” None

Lists and recursion

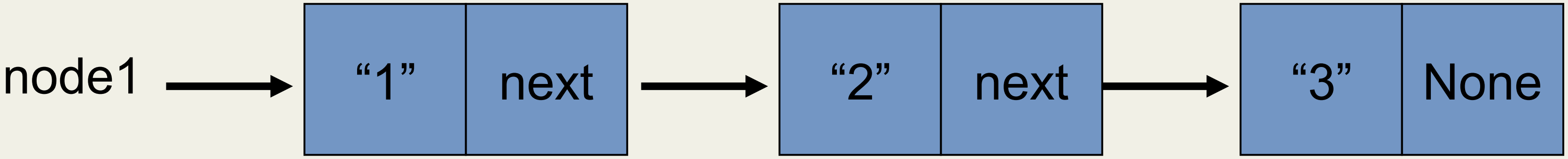
```
def printBackward(thelist):
  if thelist == None:   # base case
     return
  head = thelist
  tail = thelist.next
  printBackward(tail)   # recursive call
  print(head, end = ' ')
```

```
>>> printBackward(node1)
3 2 1
```



Searching for an item

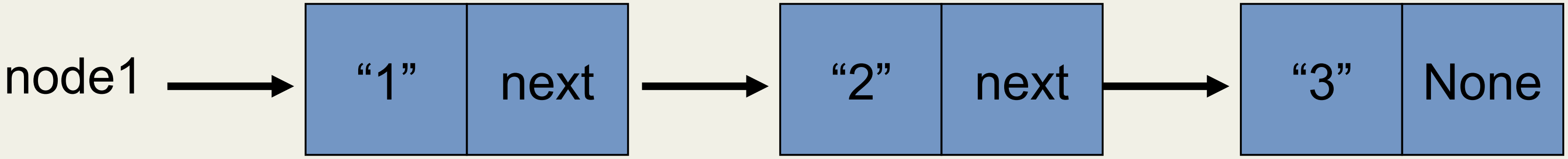
```
def searchList(node, target):
  current_node = node
  while current_node is not None and current_node.cargo != target:
    current_node = current_node.next
  if current_node is not None:  # WHAT'S A BETTER WAY TO WRITE THIS
    return True
  else:
    return False

>>> searchList(node1, “3”)
True
```



node1 → “1” next → “2” next → “3” None

Searching for an item

```
def searchList(node, target):
  current_node = node
  while current_node is not None and current_node.cargo != target:
    current_node = current_node.next
  return current_node is not None

>>> searchList(node1, “3”)
True
```



recursive version

```
def searchList(node, target):
  if node is None:          # base case
    return False
  if node.cargo == target:   # base case
    return True
  return searchList(node.next, target)   # recursive case

# BUILD A LIST
node1 = Node(1)
node2 = Node(2)
node3 = Node("dog")
node1.next = node2
node2.next = node3
```

```
>>> print searchList(node1, "dog")
True
>>> print searchList(node1, 2)
True
>>> print searchList(node1, 5)
False
```

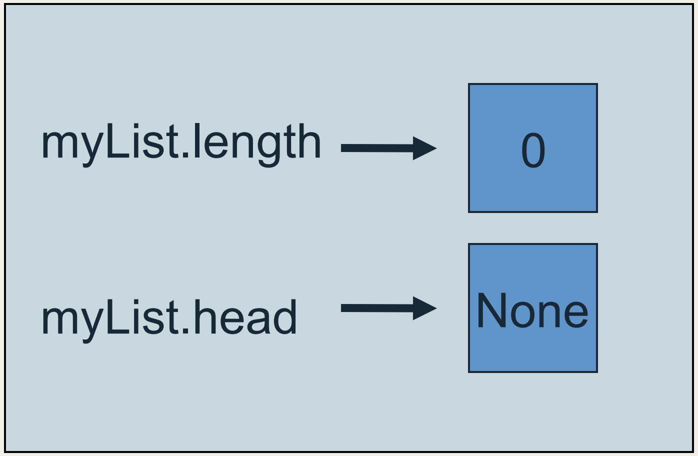
The Linked List class

How can we easily:

- add items to a linked list?
- delete items from a linked list?

We **really** need is a class that creates and manipulates lists of linked Node objects.

```
class LinkedList:
  def __init__(self):
    self.length = 0
    self.head   = None

>>> myList = LinkedList()
```



The Linked List class

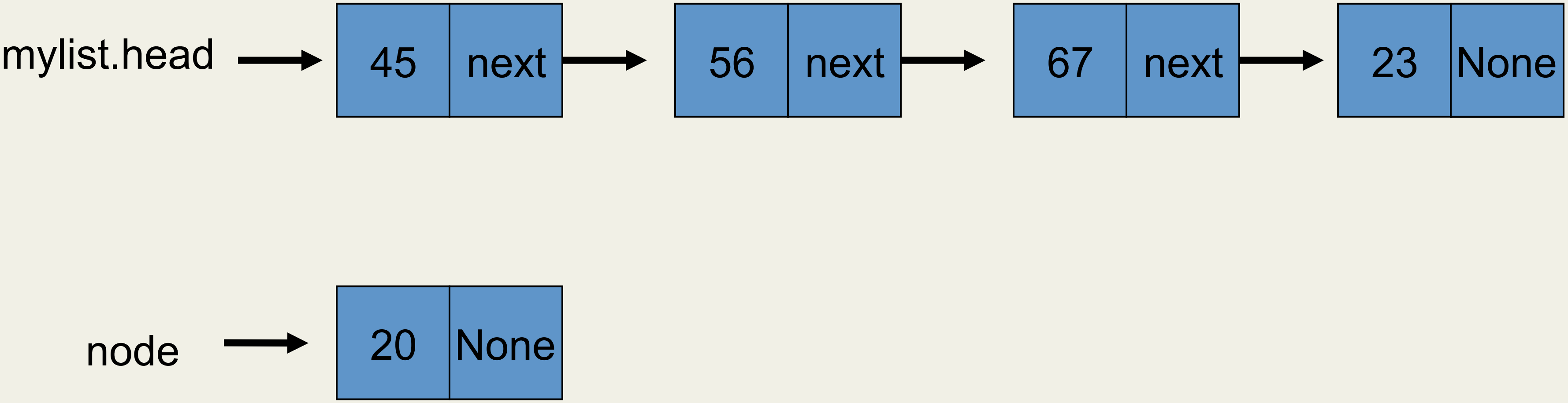
addFirst(item) takes an item of cargo as an argument and puts it in a node at the beginning of the list:

```
class LinkedList:
  def __init__(self):
    self.length = 0
    self.head   = None

  def addFirst(self, cargo):
    node = Node(cargo)
    node.next = self.head
    self.head = node
    self.length = self.length + 1
```

The Linked List class

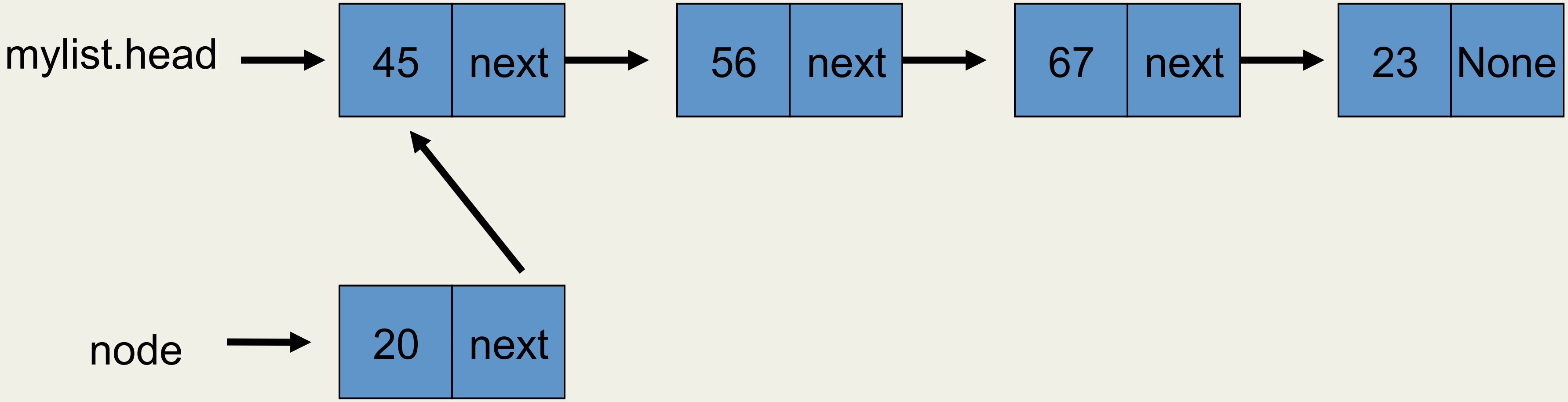
```
def addFirst(self, cargo):
    node = Node(cargo)
    node.next = self.head
    self.head = node
    self.length = self.length + 1
```

```
>>> myList.addFirst(20)
```



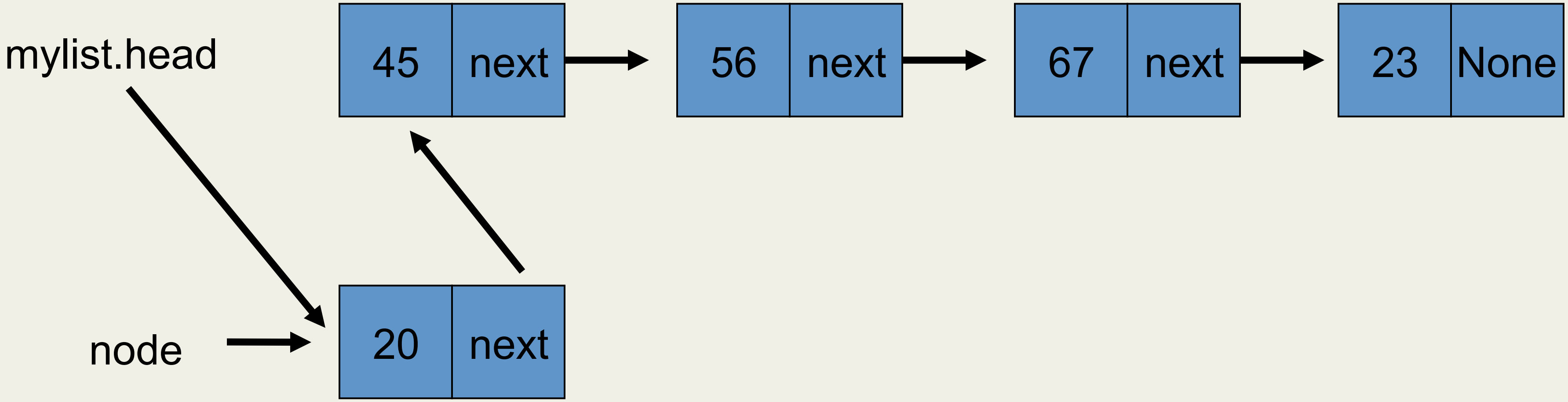
The Linked List class

```
def addFirst(self, cargo):
    node = Node(cargo)
    node.next = self.head
    self.head = node
    self.length = self.length + 1

>>> myList.addFirst(20)
```



The Linked List class

```
def addFirst(self, cargo):
   node = Node(cargo)
   node.next = self.head
   self.head = node
   self.length = self.length + 1

>>> myList.addFirst(20)
```

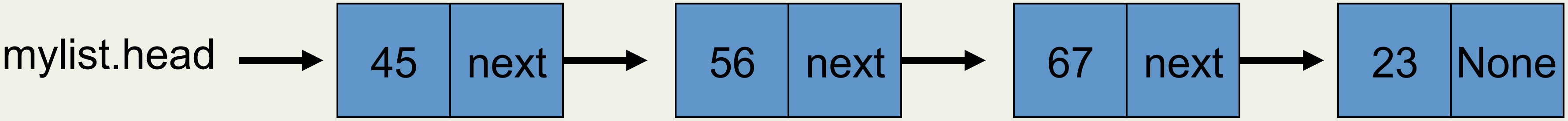


The Linked List class

Remove a node from the beginning of linked list, return cargo value:

```
def removeFirst(self):
  cargo = self.head.cargo
  self.head = self.head.next
  self.length = self.length - 1
  return cargo
```

```
>>> listval = myList.removeFirst()
```

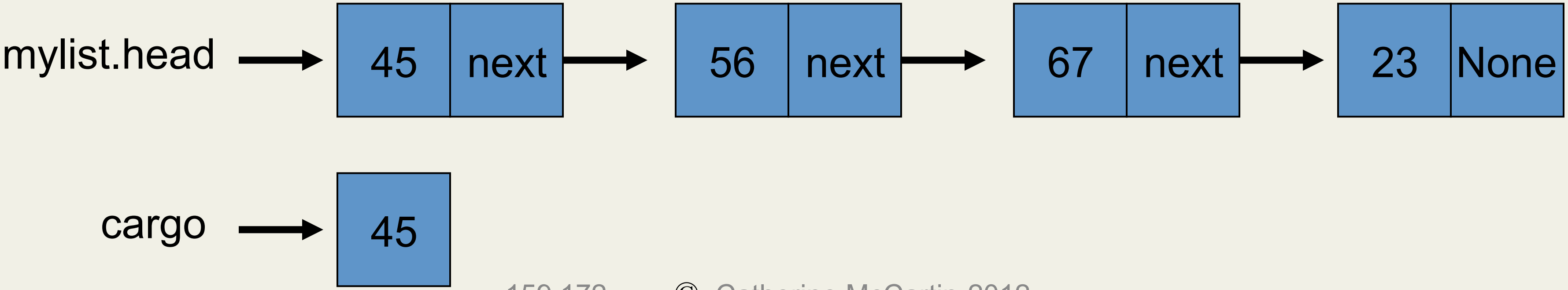


The Linked List class

Remove a node from the beginning of linked list, return cargo value:

```
def removeFirst(self):
  cargo = self.head.cargo
  self.head = self.head.next
  self.length = self.length - 1
  return cargo
```

```
>>> listval = myList.removeFirst()
```

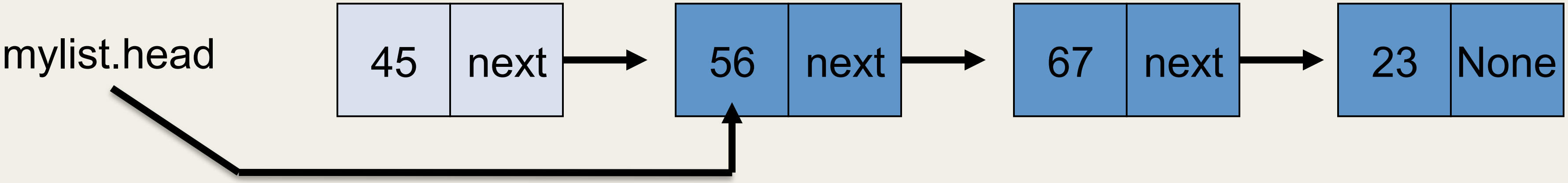


The Linked List class

Remove a node from the beginning of linked list, return cargo value:

```
def removeFirst(self):
  cargo = self.head.cargo
  self.head = self.head.next
  self.length = self.length - 1
  return cargo
```

```
>>> listval = myList.removeFirst()
```

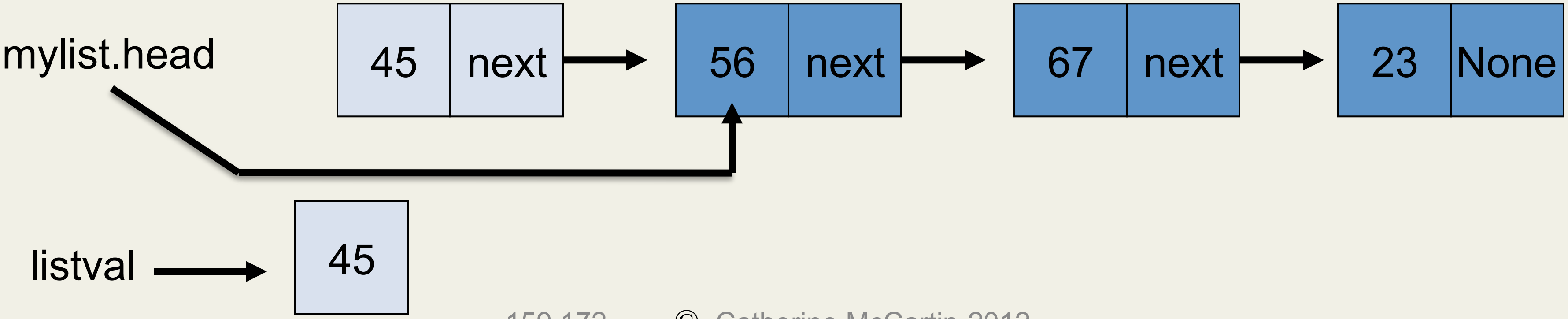


The Linked List class

Remove a node from the beginning of linked list, return cargo value:

```
def removeFirst(self):
  cargo = self.head.cargo
  self.head = self.head.next
  self.length = self.length - 1
  return cargo
```

```
>>> listval = myList.removeFirst()
```



The Linked List class

Adding an item to the end of a linked list?

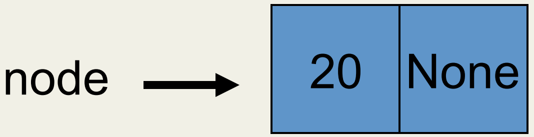
```
def insertLast(self, cargo):
    node = Node(cargo)
    node.next = None
    # if list is empty the new node goes first
    if self.head == None:
            self.head = node
    else:
        last = self.head      # find the last node in the list
        while last.next is not None:
            last = last.next
        # append the new node
        last.next = node
    self.length = self.length + 1
```

The Linked List class

Adding an item to the end of a linked list?

```
>>> mylist.insertLast(20)
```

```
node = Node(cargo)
node.next = None
if self.head == None:
    # if list is empty the new node goes first
    self.head = node
```

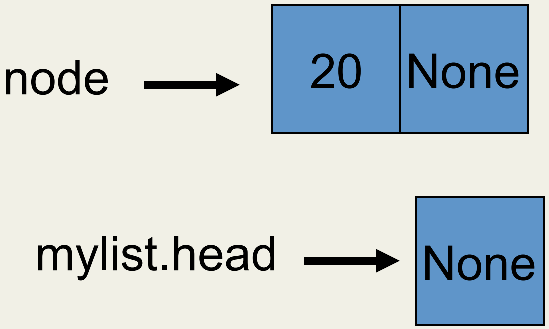


The Linked List class

Adding an item to the end of a linked list?

```
>>> mylist.insertLast(20)
```

```
node = Node(cargo)
node.next = None
if self.head == None:
  # if list is empty the new node goes first
  self.head = node
```

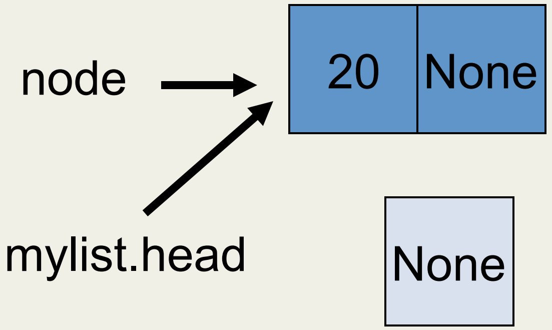


The Linked List class

Adding an item to the end of a linked list?

```
>>> mylist.insertLast(20)
```

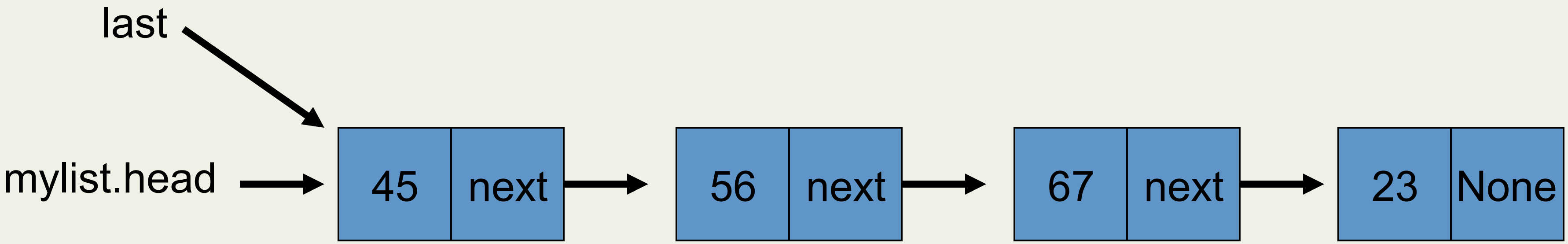
```
node = Node(cargo)
node.next = None
if self.head == None:
  # if list is empty the new node goes first
  self.head = node
```



The Linked List class

Adding an item to the end of a linked list?

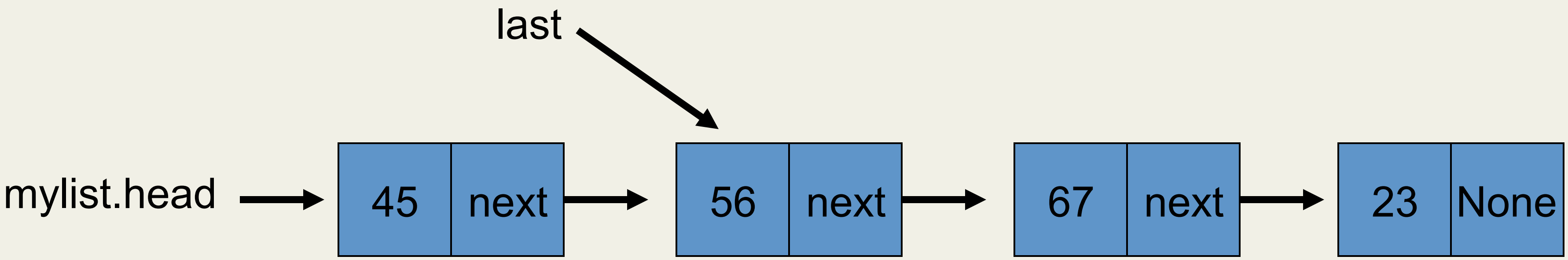
```
else:
  # find the last node in the list
  last = self.head
  while last.next is not None:
      last = last.next
  # append the new node
  last.next = node
 self.length = self.length + 1
```



The Linked List class

Adding an item to the end of a linked list?

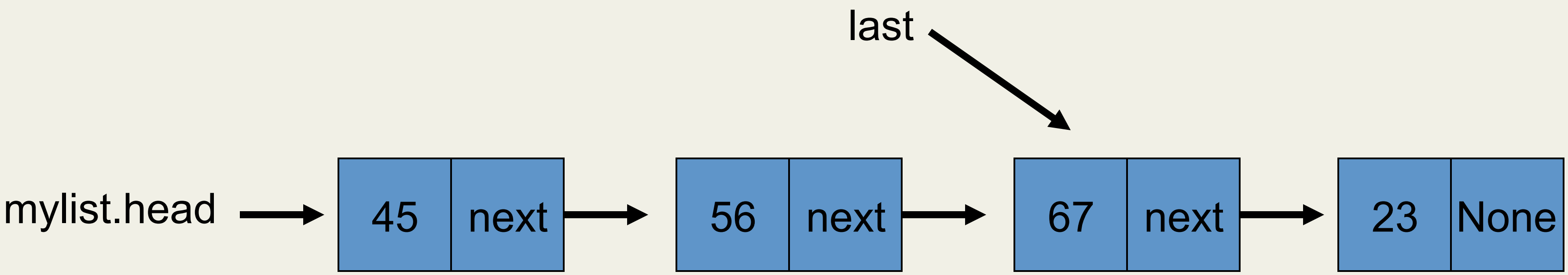
```
else:
  # find the last node in the list
  last = self.head
  while last.next is not None:
        last = last.next
  # append the new node
  last.next = node
 self.length = self.length + 1
```



The Linked List class

Adding an item to the end of a linked list?

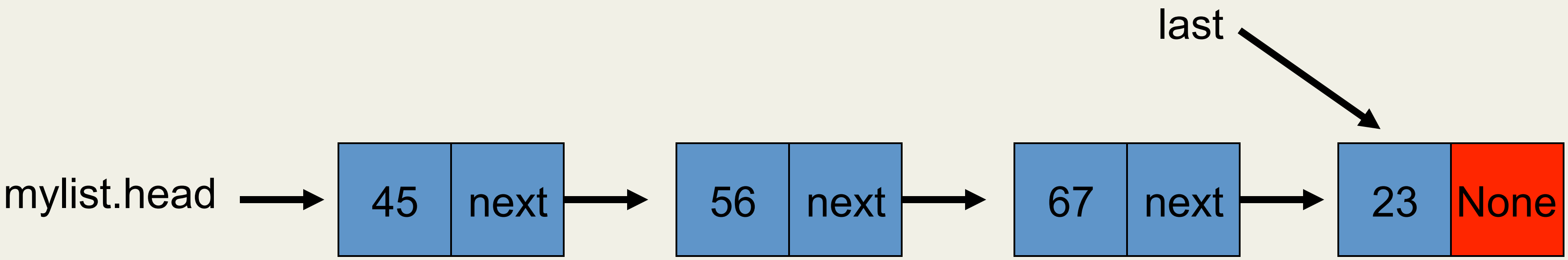
```
else:
  # find the last node in the list
  last = self.head
  while last.next is not None:
        last = last.next
  # append the new node
  last.next = node
 self.length = self.length + 1
```



The Linked List class

Adding an item to the end of a linked list?

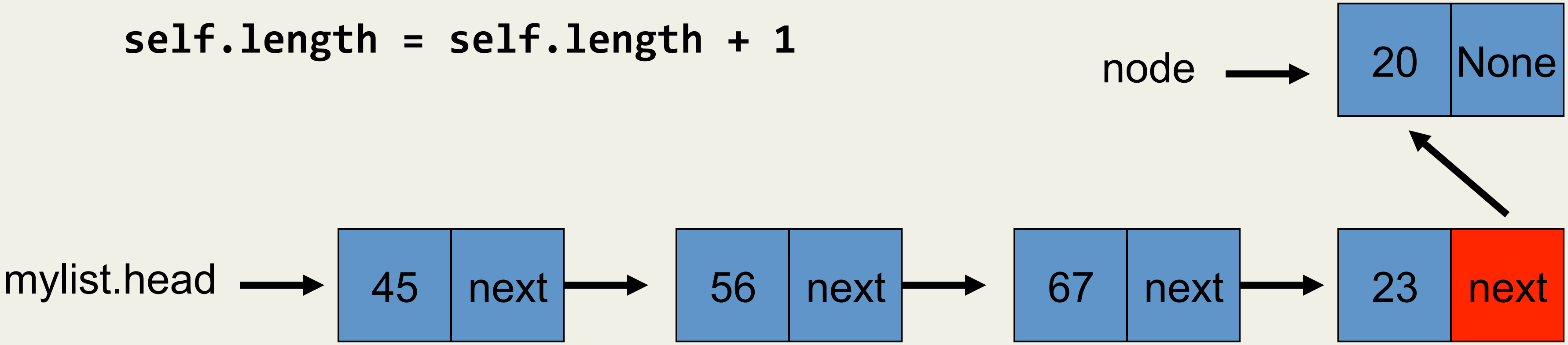
```
else:
  # find the last node in the list
  last = self.head
  while last.next is not None:
        last = last.next
  # append the new node
  last.next = node
 self.length = self.length + 1
```



The Linked List class

Adding an item to the end of a linked list?

```
else:
  # find the last node in the list
  last = self.head
  while last.next is not None:
        last = last.next
  # append the new node
  last.next = node
 self.length = self.length + 1
```



The Queue ADT

The Queue ADT is defined by the following operations:

__init__()

Initialize a new empty queue.

insert(new_item) or sometimes **enqueue(new_item)**

Add a new item to the queue.

remove() or sometimes **dequeue**()

Remove and return an item from the queue.

The item that is returned is the first one that was added.

isEmpty()

Check whether the queue is empty.

Linked list implementation

```
class Queue:
  def __init__(self):
    self.length = 0
    self.head = None

  def isEmpty(self):
    return (self.length == 0)
```

Queue Linked list implementation

```
def insert(self, cargo):
    # same as insertLast() for Linked Lists
    node = Node(cargo)
    node.next = None
    if self.head == None:
       # if queue is empty the new node goes first
       self.head = node
    else:
       # find the last node in the queue
       last = self.head
       while last.next is not None:    # MIGHT TAKE A LONG TIME
         last = last.next
       # append the new node
       last.next = node
    self.length = self.length + 1
```

Queue Linked list implementation

```
def remove(self):
# same as removeFirst() for Linked Lists
  cargo = self.head.cargo
  self.head = self.head.next
  self.length = self.length - 1
  return cargo
```

Improved linked Queue

We can improve the performance of the insert method if we modify the Queue class so that it maintains a reference to both the first and the **last** node.

```
class ImprovedQueue:
  def __init__(self):
    self.length = 0
    self.head   = None
    self.last   = None

  def isEmpty(self):
    return (self.length == 0)
```

Improved linked Queue

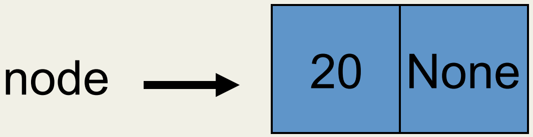
```
def insert(self, cargo):
    node = Node(cargo)
    node.next = None
    if self.length == 0:
        # if list is empty, the new node is head and last
        self.head = self.last = node
    else:
        # find the last node
        last = self.last
        # append the new node
        last.next = node
        self.last = node
    self.length = self.length + 1
```

Improved linked Queue

Adding an item to the end of a Queue?

```
>>> myQueue.insert(20)
```

```
node = Node(cargo)
node.next = None
if self.length == 0:
  # if list is empty, the new node is head and last
  self.head = self.last = node
```

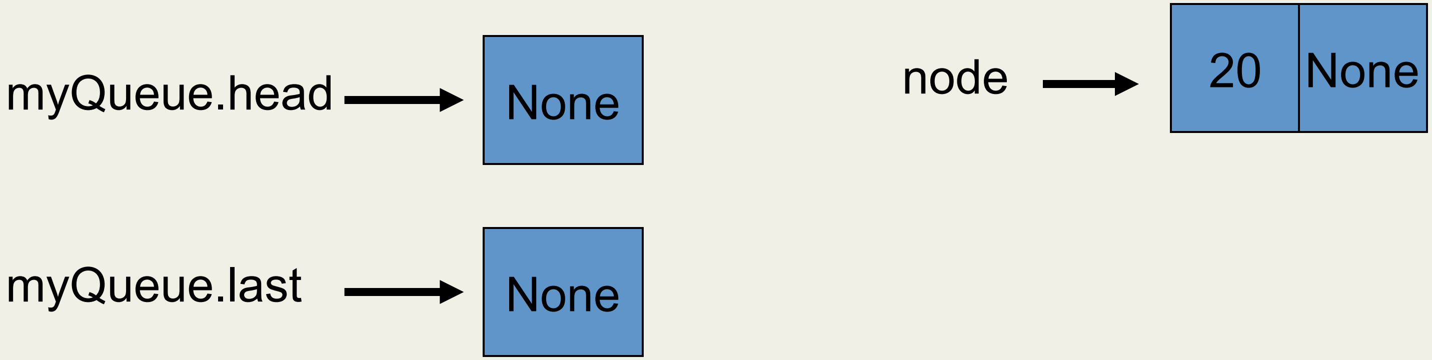


Improved linked Queue

Adding an item to the end of a Queue?

```
>>> myQueue.insert(20)
```

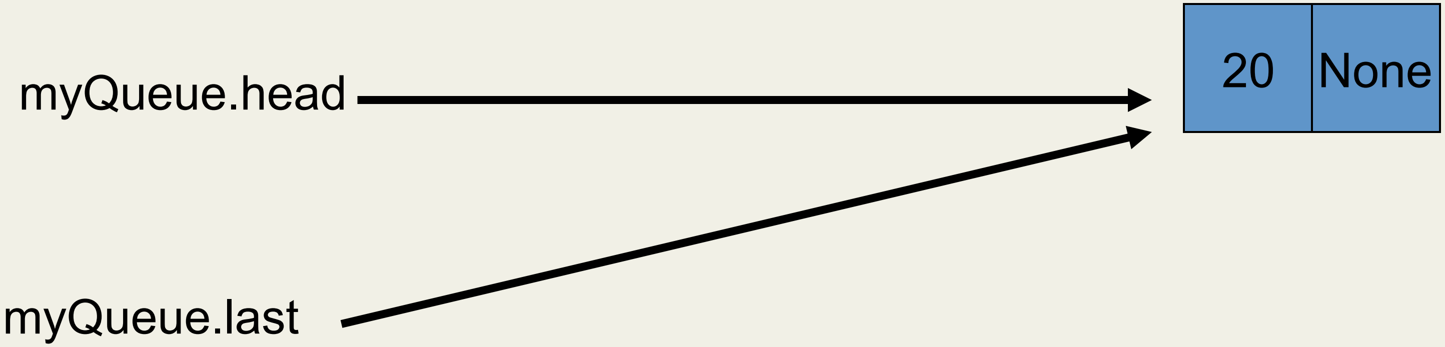
```
node = Node(cargo)
node.next = None
if self.length == 0:
  # if list is empty, the new node is head and last
  self.head = self.last = node
```



Improved linked Queue

Adding an item to the end of a Queue?

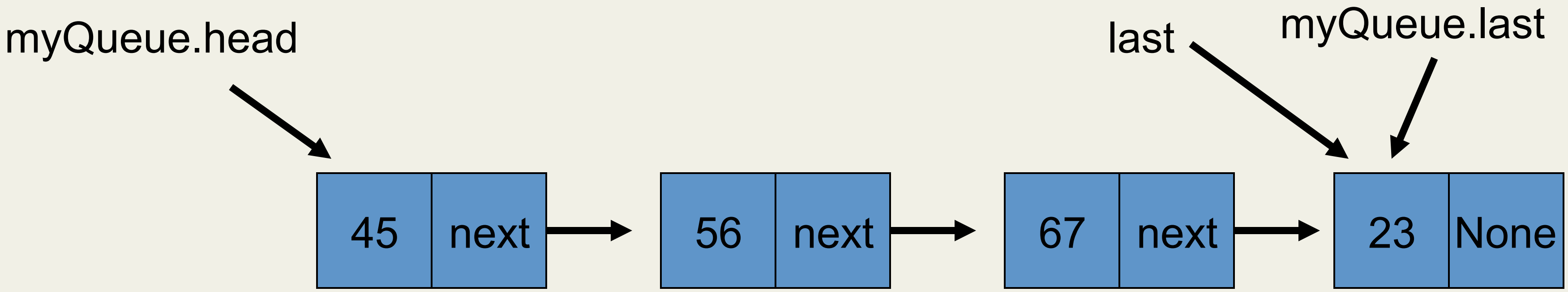
```
>>> myQueue.insert(20)

  node = Node(cargo)
  node.next = None
  if self.length == 0:
    # if list is empty, the new node is head and last
    self.head = self.last = node
```



Improved linked Queue

Adding an item to the end of a Queue?

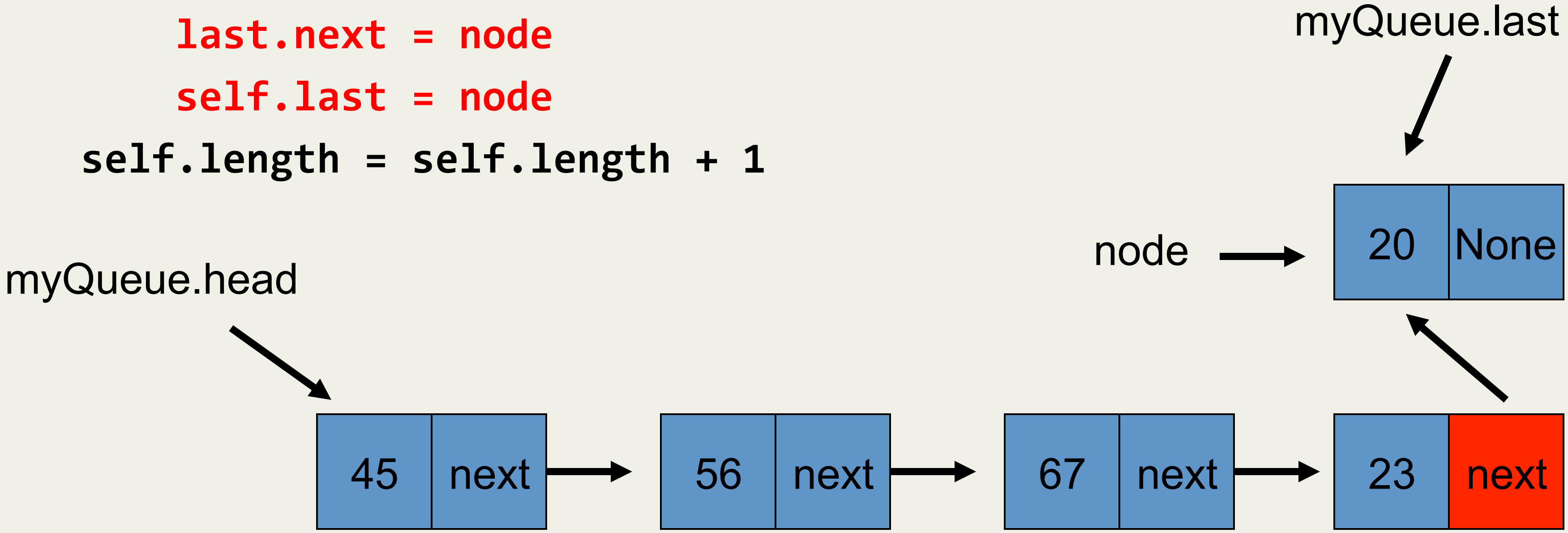
```
else:
    # find the last node
    last = self.last
    # append the new node
    last.next = node
    self.last = node
self.length = self.length + 1
```



Improved linked Queue

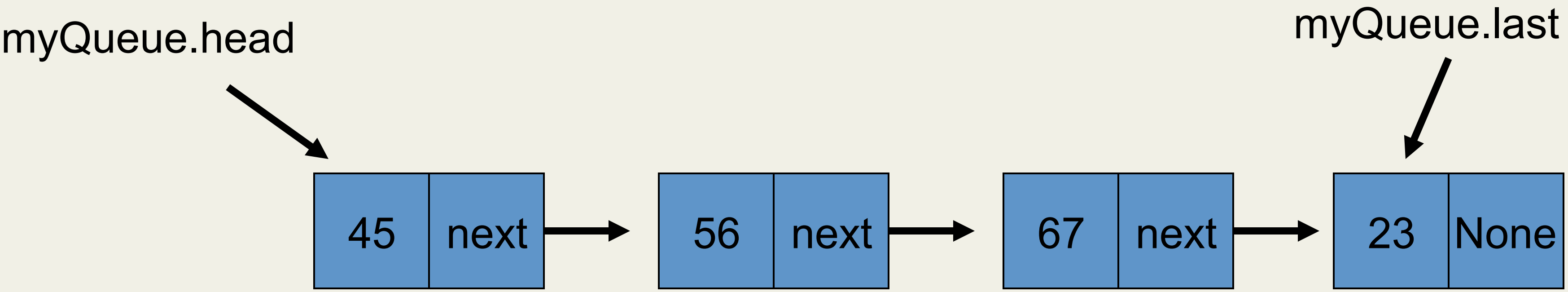
Adding an item to the end of a Queue?

```
else:
    # find the last node
    last = self.last
    # append the new node
    last.next = node
    self.last = node
self.length = self.length + 1
```



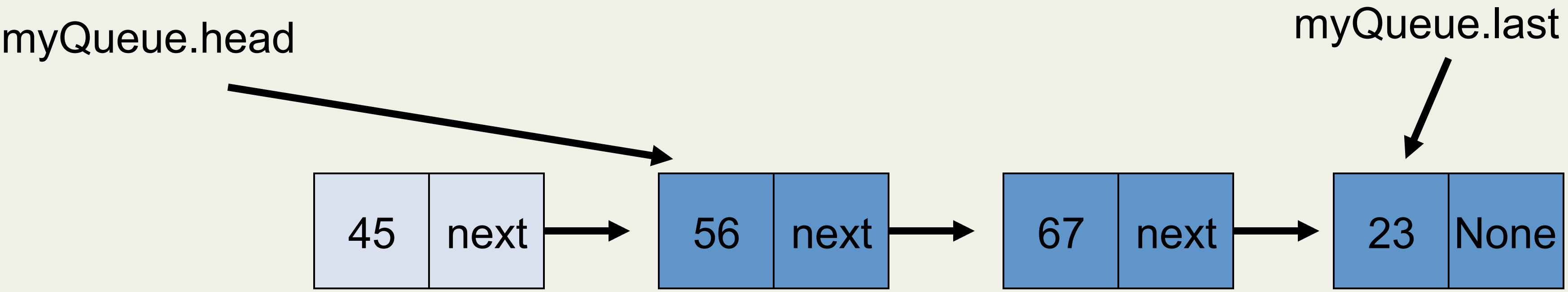
Improved linked Queue

```
def remove(self):
    cargo     = self.head.cargo
    self.head = self.head.next
    self.length = self.length – 1
    # if list becomes empty, last must be set to None
    if self.length == 0:
      self.last = None
    return cargo
```



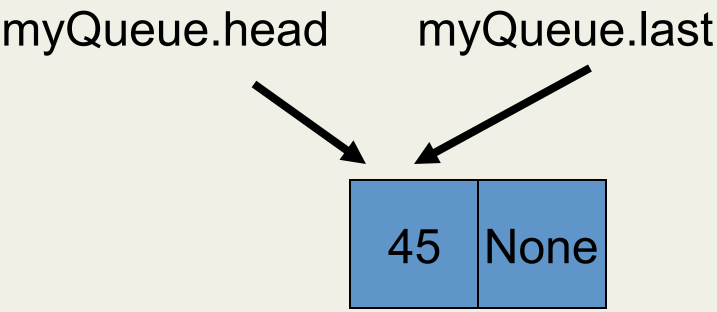
Improved linked Queue

```
def remove(self):
    cargo       = self.head.cargo
    self.head = self.head.next
    self.length = self.length – 1
    # if list becomes empty, last must be set to None
    if self.length == 0:
      self.last = None
    return cargo
```



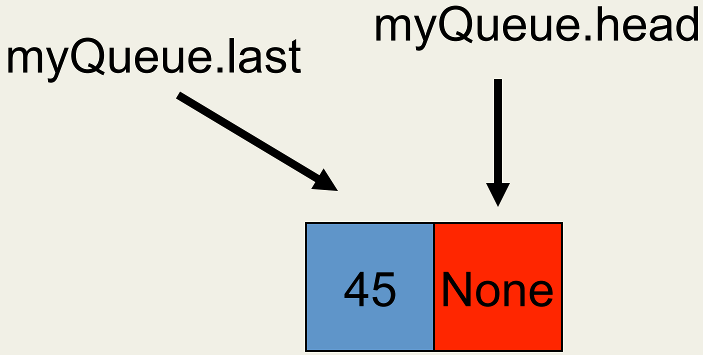
Improved linked Queue

```
def remove(self):
    cargo     = self.head.cargo
    self.head = self.head.next
    self.length = self.length – 1
    # if list becomes empty, last must be set to None
    if self.length == 0:
      self.last = None
    return cargo
```



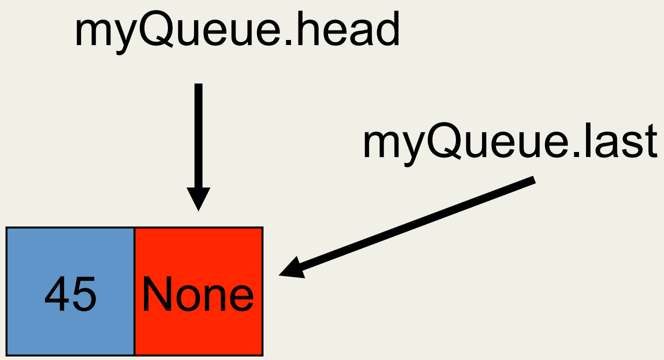
Improved linked Queue

```
def remove(self):
    cargo     = self.head.cargo
    self.head = self.head.next
    self.length = self.length – 1
    # if list becomes empty, last must be set to None
    if self.length == 0:
      self.last = None
    return cargo
```



Improved linked Queue

```
def remove(self):
    cargo     = self.head.cargo
    self.head = self.head.next
    self.length = self.length – 1
    # if list becomes empty, last must be set to None
    if self.length == 0:
      self.last = None
    return cargo
```


myQueue.head
myQueue.last
45
None