Linked Lists

Announcements

- Lab 11 is due tomorrow
- Quiz 6 is on Monday
  - Emphasis on sorting and recursion
- Ninja session tonight, 7-9pm
- The final is in two weeks!
  - Will cover class definitions and linked lists

Today's Plan

- Highlight quiz 6 topics
- Recap of defining classes
- Intro to linked lists

Quiz 6 Topics

- Understand sorting algorithms, especially bubble sort and selection sort
    - Know the order in which swaps will happen
    - Understand what the code is doing
- Know run times for various algorithms

Algorithm run times

- *O(1)*: indexing a list, arithmetic
- *O(log n)*: binary search
- *O(n)*: linear search
- *O(n log n)*: merge sort
- $O(n^2)$: bubble sort, selection sort, insertion sort

Quiz 6 topics

- Drawing the stack for a recursive function call
- Writing recursive functions
  - The “leap of faith” approach.
  - Look at examples of recursion over ints, lists, and strings.

Recursion rules of thumb

Type	Base case	General case
int	`n == 0` or `n == 1`	Combine `n` and a recursive call on `n-1`
sequence	`len(seq) == 0` or `len(seq) == 1`	Combine `seq[0]` and a recursive call on `seq[1:]`

Classes recap

- The methods for a class create an **interface** for interacting with instances of that class.

- There are **two perspectives** on a class: that of the person implementing the class and that of the person using the class. (Sometimes the same person.)

- The “bookkeeping” is hidden behind the class interface from the person using the class. This makes the class more convenient to use. Bookkeeping involves creating and maintaining instance variables.

Classes recap

- Other than `__init__`, which is called first, methods may be called in any order. Make sure your instance variables stay current regardless.

- When you implement a method ask yourself: *did this method change any of my instance variables?*

- Choose the **minimal** set of instance variables that allows you to implement the methods listed in the class interface. If you hang on to extra instance variables you'll have to do extra work to keep these current as well.

Classes recap

- Instance variables are kept in the self parameter: `self.x`, `self.y`, `self.bug`, `self.heading`, etc.

- To call a method from within another method the syntax is `self.method_name()` not `method_name(self)`.

Linked lists

- Can we use classes to create something like a list?

  - Yes, linked lists are an alternative **data structure** to Python's built-in lists.

  - Linked lists and Python lists share a similar interface.

  - "Beneath the hood" they are different; some operations are faster with linked lists, others with Python lists.

  - Choose which data structure you want based on the features of a particular application.

List interface

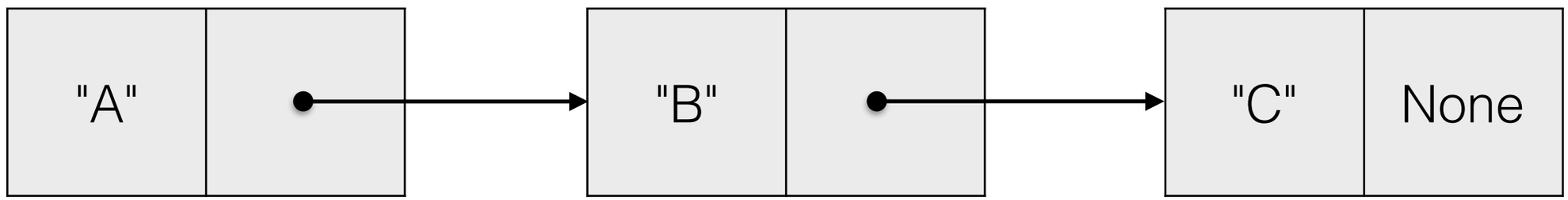
- Operations:
  - Create an empty list
  - Append a new item onto end of list
  - Get the length of a list
  - Access values in a list
  - Remove an item from a list
  - Search through a list
  - Sort a list
  - Change an item in a list

Linked list structure

- Each item in a linked list corresponds to a “node”
- A node is a bicameral (two-chambered) cell:
  - One chamber holds the item
  - Other chamber points to the next node
- If we have the first node, we can access every node in the linked list by following the **chain of links**.

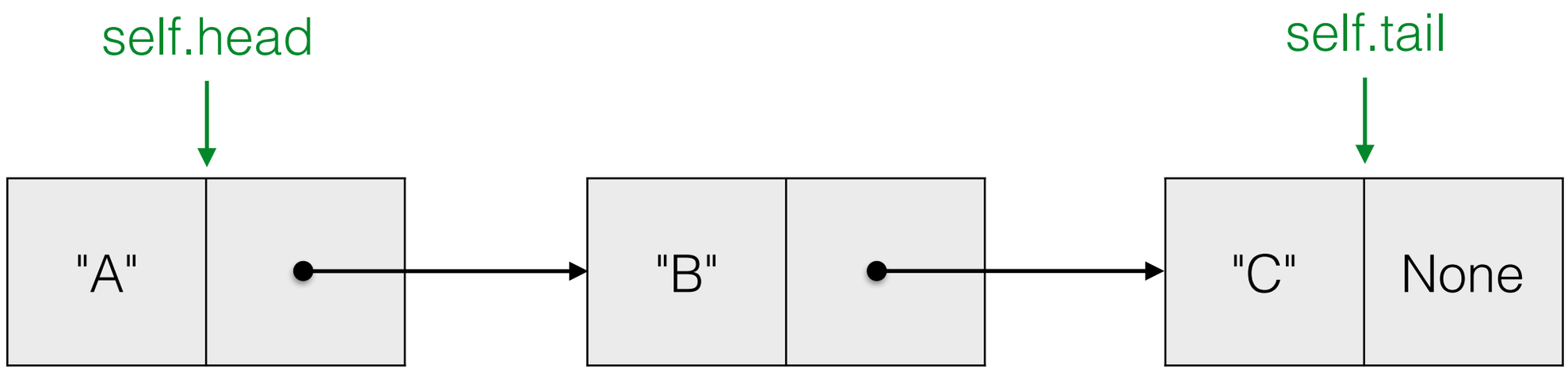
Linked list structure

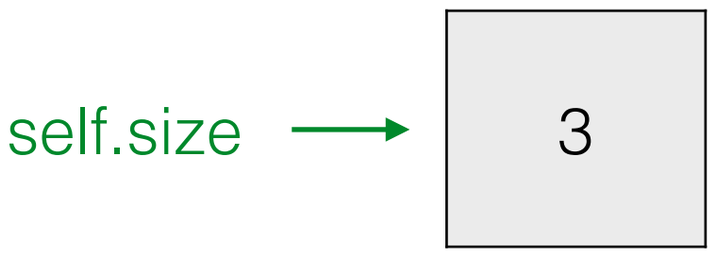
- Linked list of strings “A”, “B”, “C”



Linked list structure

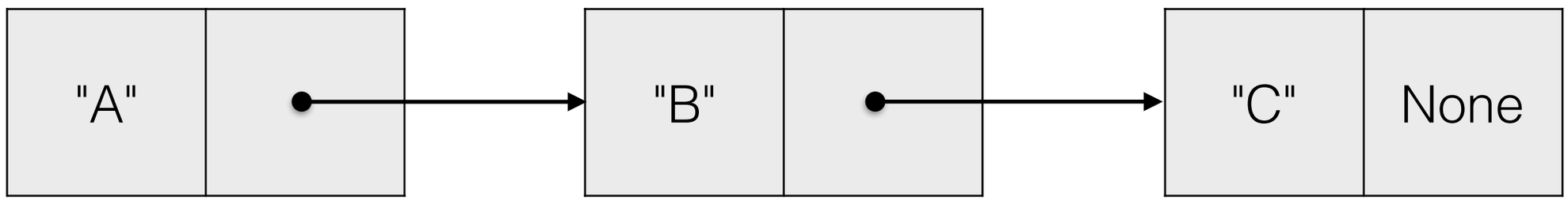
- Linked list of strings “A”, “B”, “C”





```
class Node(object):

    def __init__(self, data):
        self.data = data
        self.next = None

    def getData(self):
        return self.data

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

    def getNext(self):
        return self.next

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

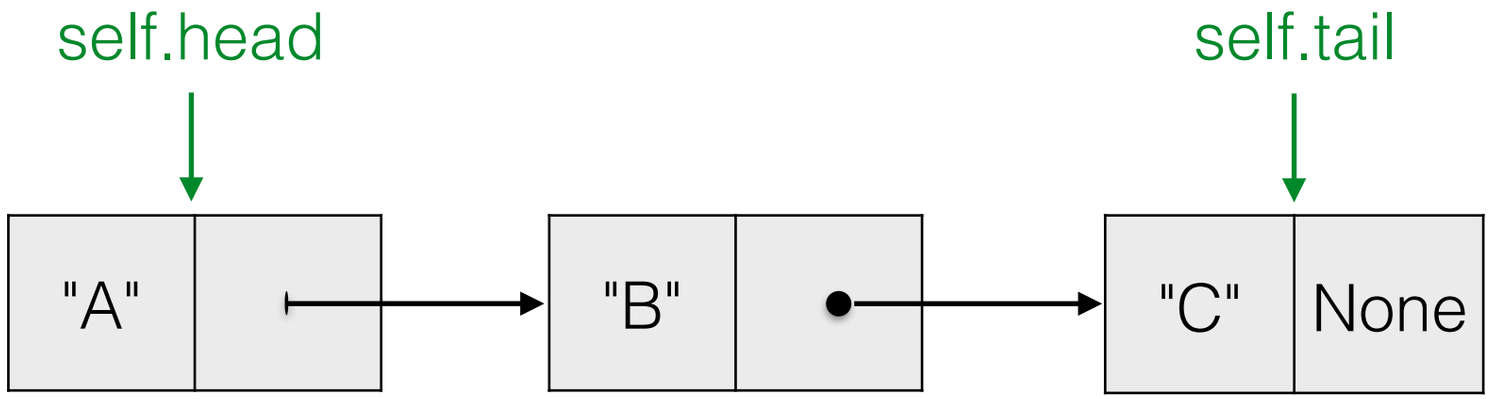
Construction

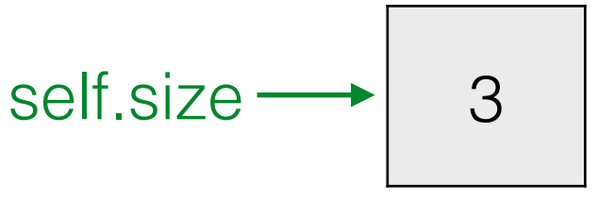



```
node1 = Node("A")
node2 = Node("B")
node3 = Node("C")
node1.setNext(node2)
node2.setNext(node3)
```

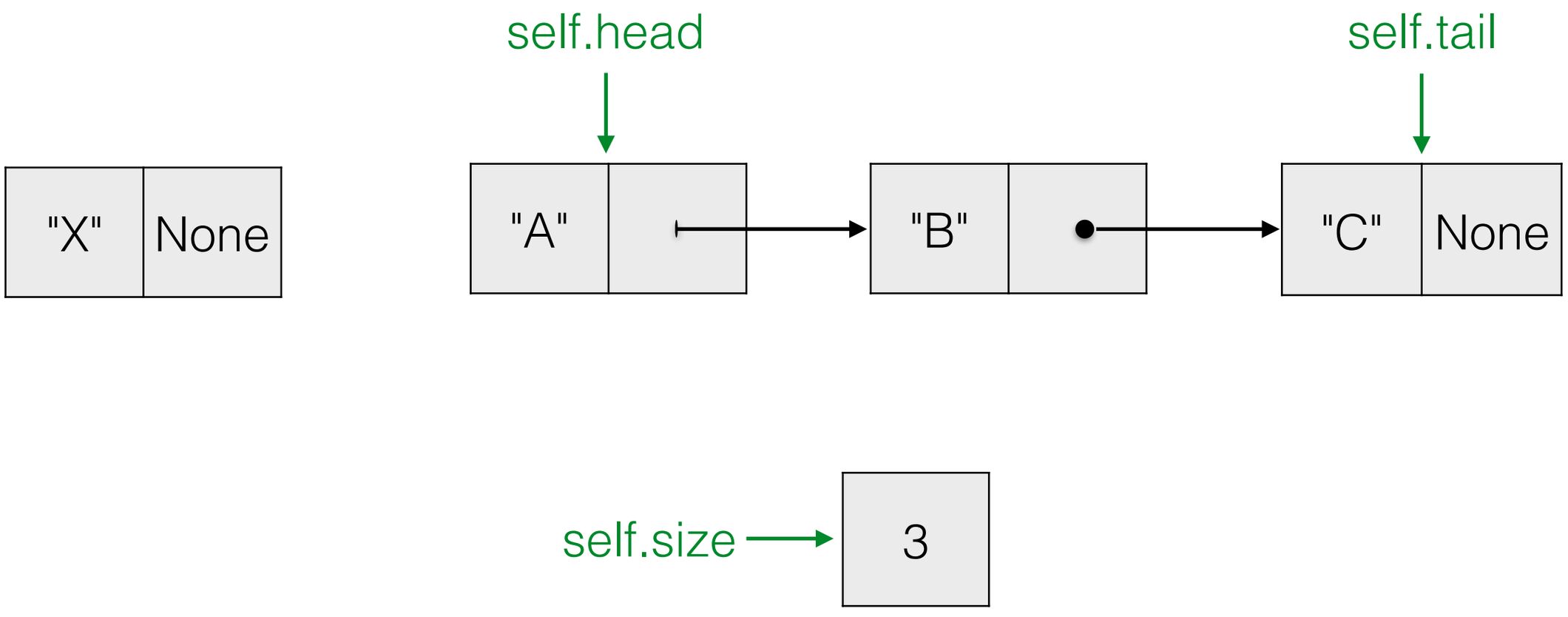
```
from node import *

class LinkedList(object):

    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0

    def getFirstItem(self):
        return self.head.getData()

    def getLastItem(self):
        return self.tail.getData()

    def getSize(self):
        return self.size

    ...
```

Adding “X” to the front

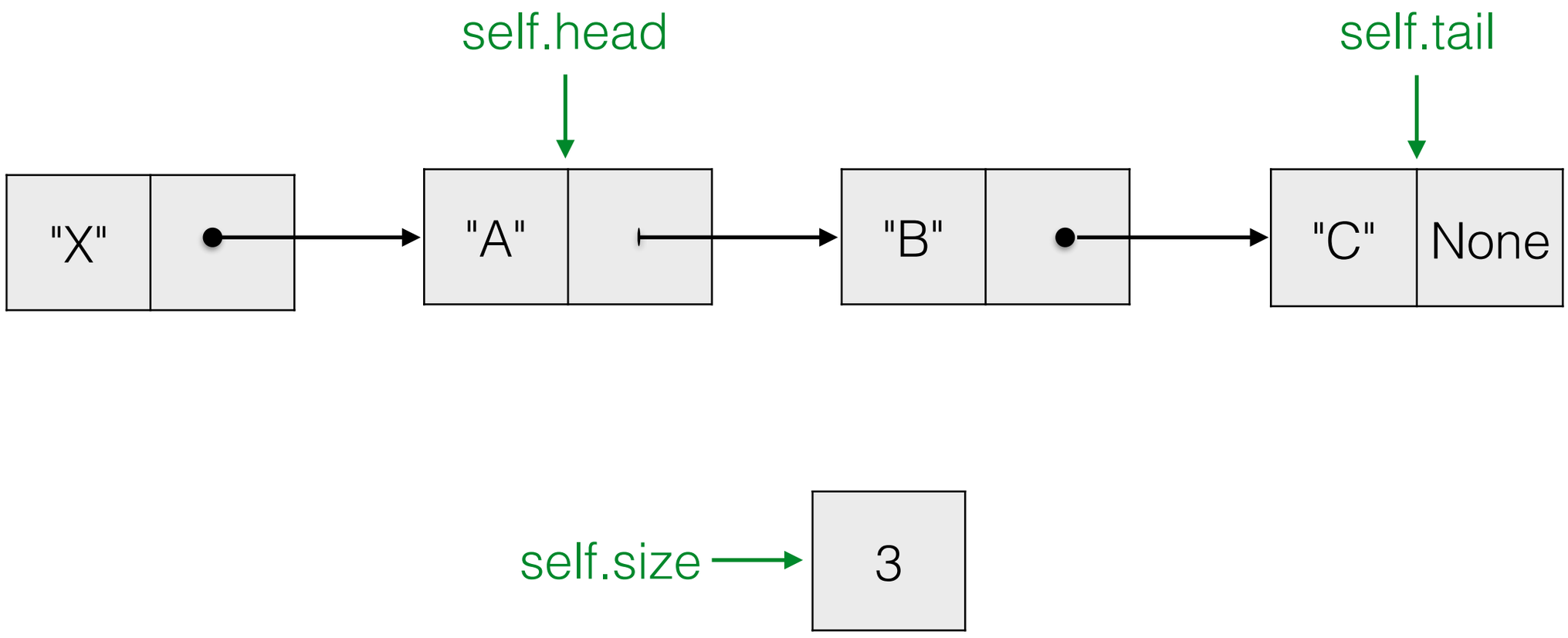




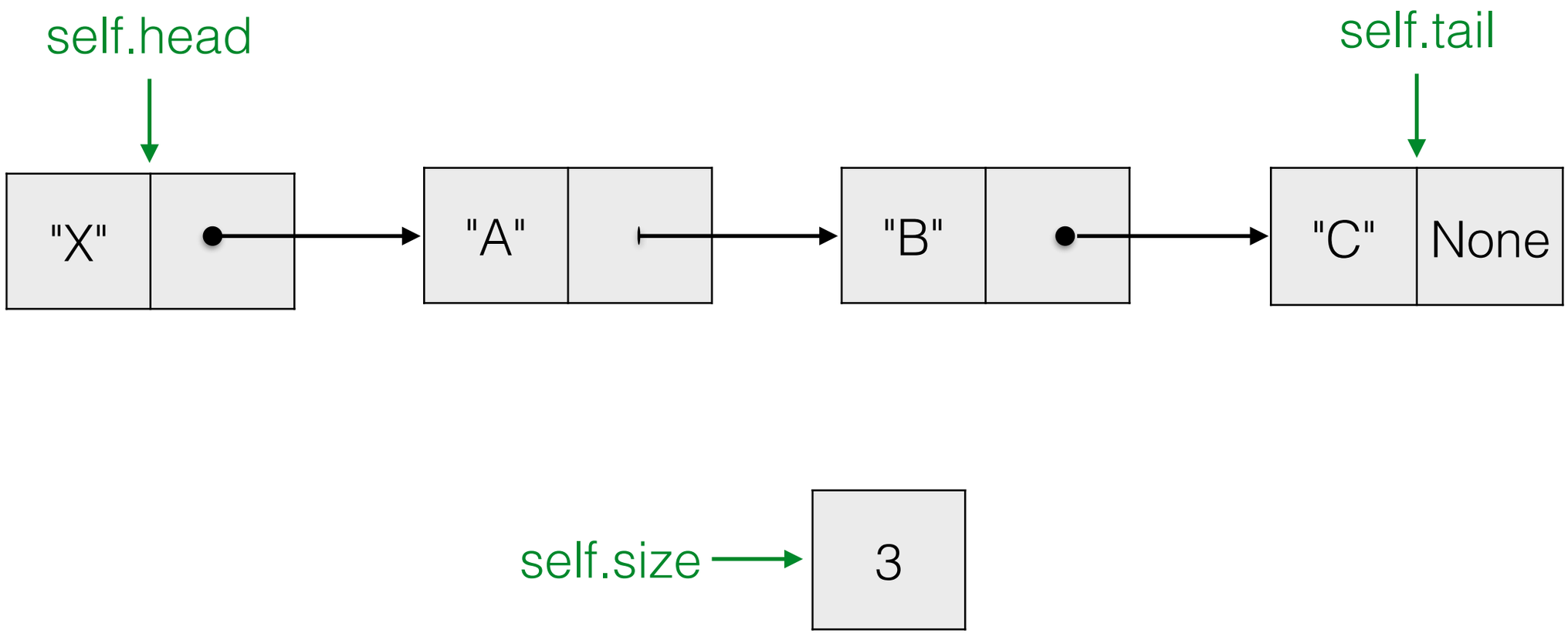
Adding “X” to the front



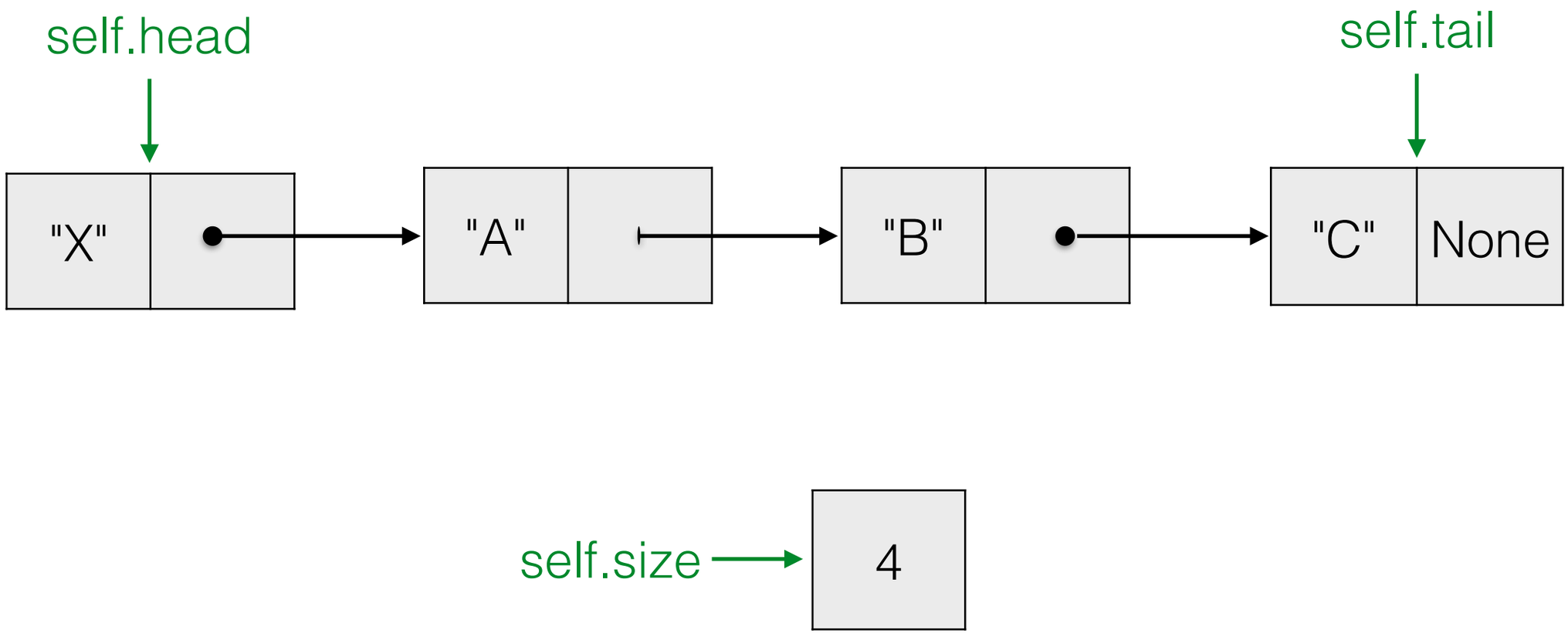
Adding “X” to the front



Adding “X” to the front



Adding “X” to the front



Adding to the front

- To insert a new item:

    1. Create a new node with the item in its first chamber.
    2. Point new node's second chamber to the former first node.
    3. Update `self.head` to point to new node
    4. Increment `self.size`

- If the list is empty update `self.tail` also

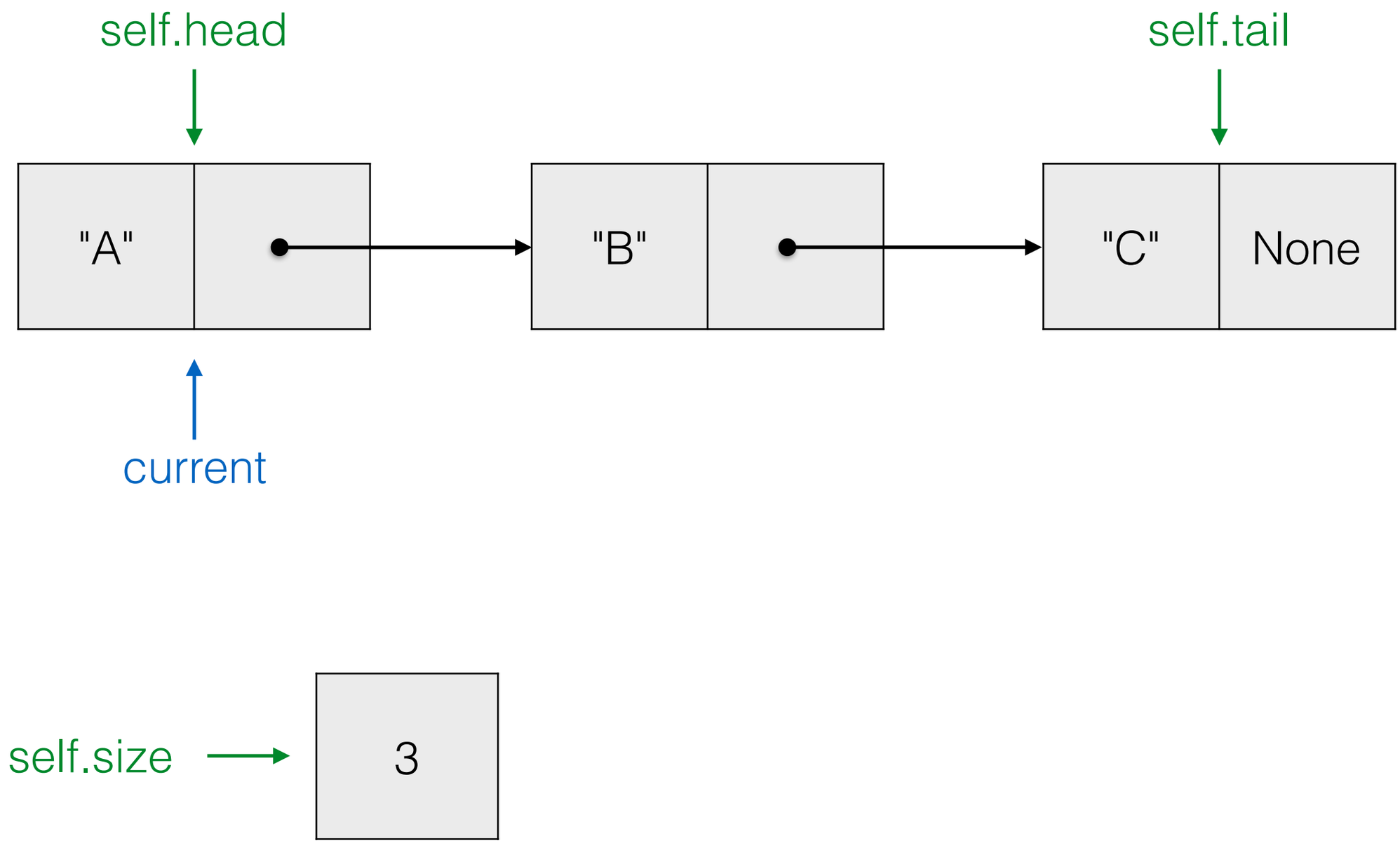
Adding to the front

```
class LinkedList(object):
    ...
    # Add item to front
    def prepend(self, item):
        newNode = Node(item)
        if self.size == 0:
            self.head = newNode
            self.tail = newNode
        else:
            newNode.setNext(self.head)
            self.head = newNode
        self.size += 1
    ...
```

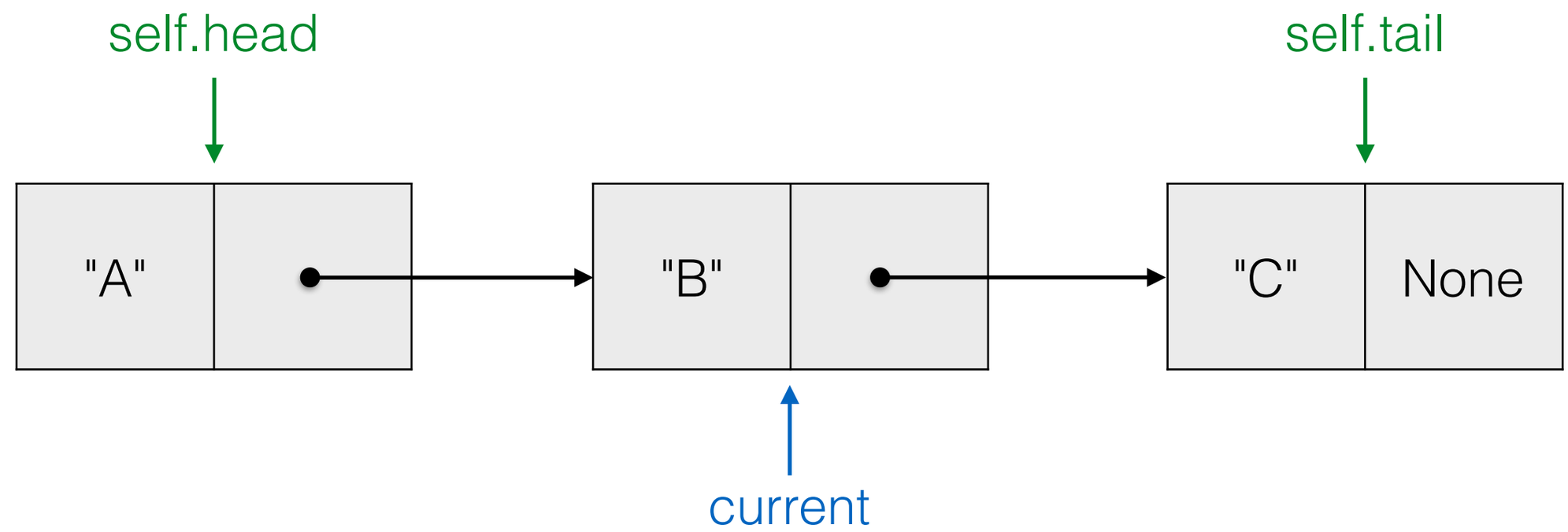
Adding to the back

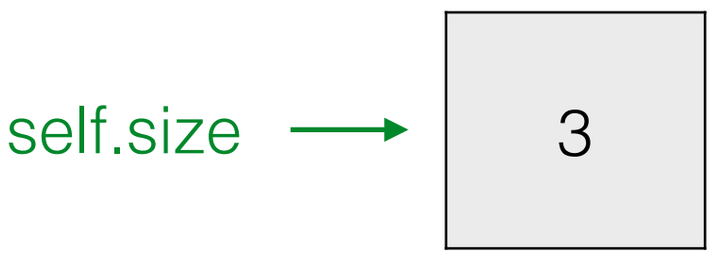
- It’s like adding to the front, except now we need to update ‘next’ for the last node (`self.tail`)

Traversing a linked list

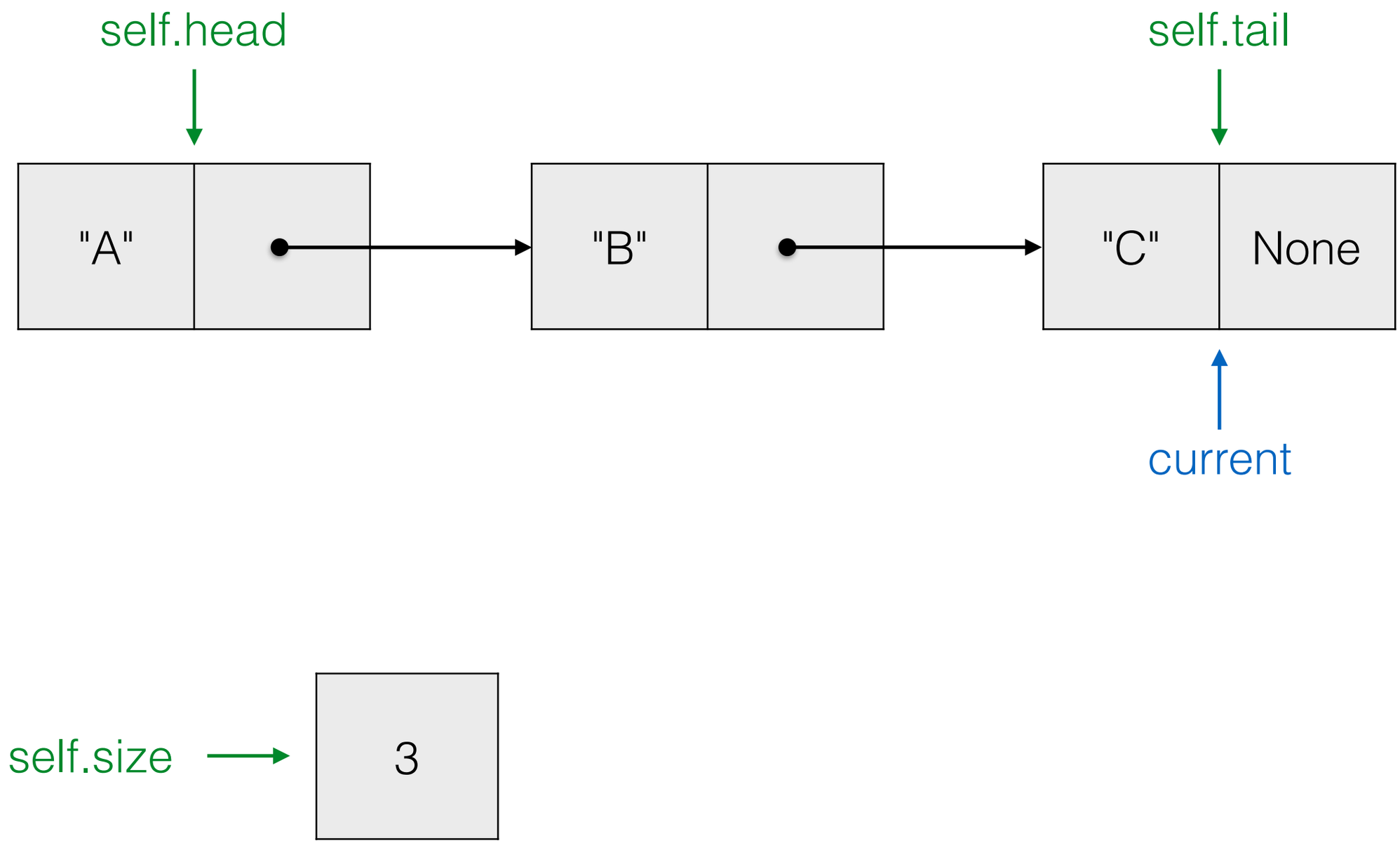


Traversing a linked list

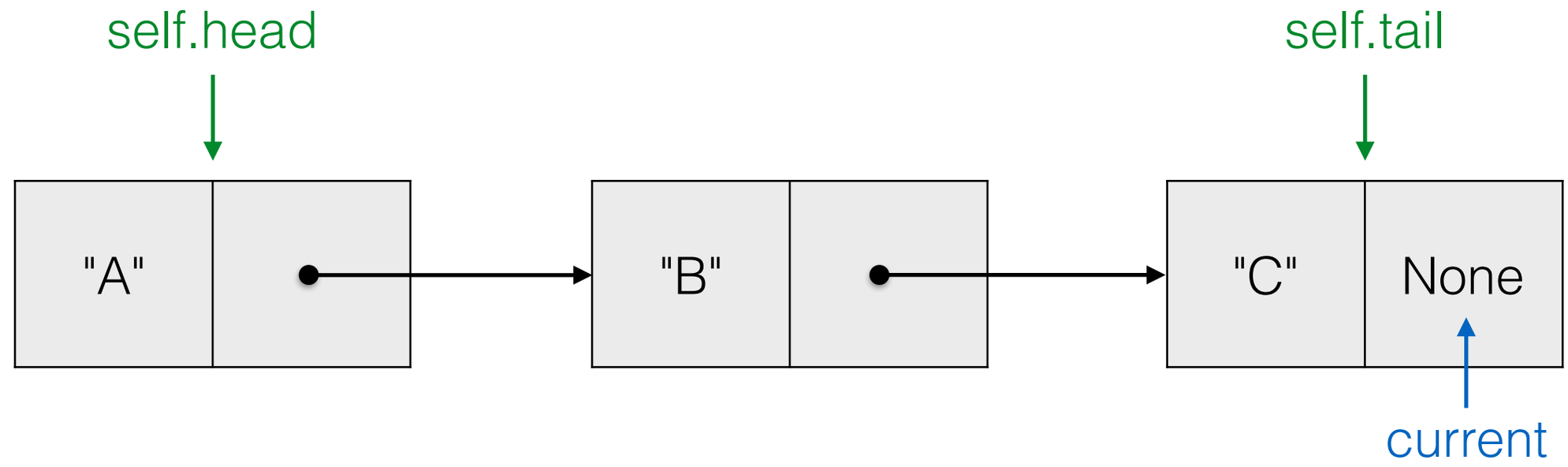


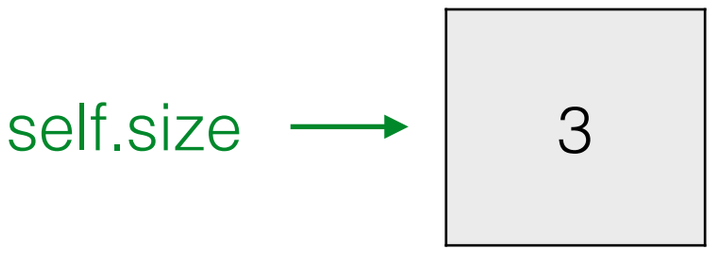


Traversing a linked list



Traversing a linked list





Traversing a linked list

- Use a while loop that continues until the current `Node` is equal to `None`
- (Can also use a for loop since we know the number of nodes)
- Traversal will be used in a number of methods:
  - `__str__`, searching, indexing, etc.