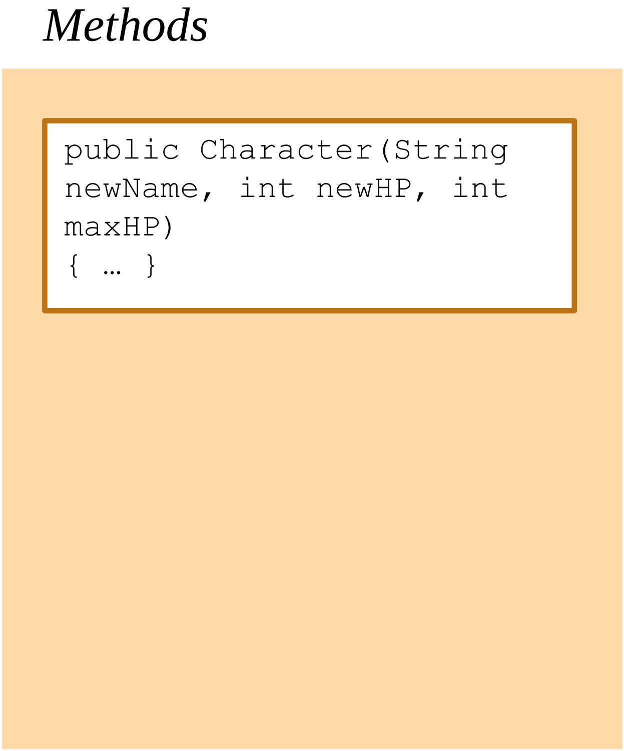
References and Linked Lists

Object References

Linked Lists

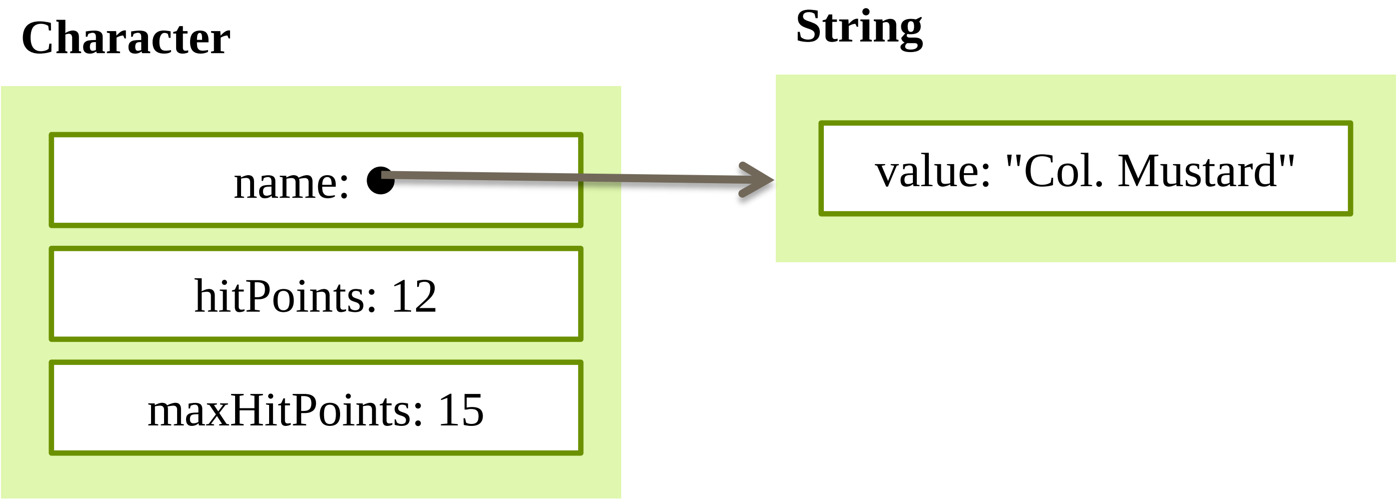
Object References

```
public class Character
{
    String name;
    int hitPoints;
    int maxHitPoints;

    public Character(String newName,
                     int newHP,
                     int maxHP)
    {
        name = newName;
        hitPoints = newHP;
        maxHitPoints = maxHP;
    }
}
```



Methods

```
public Character(String
newName, int newHP, int
maxHP)
{ … }
```



Character

name:

hitPoints: 12

maxHitPoints: 15

String

value: "Col. Mustard"

```
Character c = new Character("Col. Mustard", 12, 15);
```

Why == Doesn't Work for Strings

```
String magicWord =
    new String ("abacadabra");
String otherMagicWord =
    new String("abacadabra");

// this expression is NOT true
if (magicWord == otherMagicWord)
{
}
```

Why == Doesn't Work for Strings

```
String magicWord = "abacadabra";
String otherMagicWord = "abacadabra";

// this expression is NOT true
if (magicWord == otherMagicWord)
{
}
```

This is a reference to one `String` object in memory…

Why == Doesn't Work for Strings

```
String magicWord = "abacadabra";
String otherMagicWord = "abacadabra";

// this expression is NOT true
if (magicWord == otherMagicWord)
{
}
```

…and this is a reference to a different `String` object in memory…

Why == Doesn't Work for Strings

```
String magicWord = "abacadabra";
String otherMagicWord = "abacadabra";

// this expression is NOT true
if (magicWord == otherMagicWord)
{
}
```

…so the two references are not equal – they point to different locations in memory

Why == Doesn't Work for Strings

```
String magicWord = "abacadabra";
String otherMagicWord = "abacadabra";

// this expression IS true
if (magicWord == magicWord)
{
}
```

Why == Doesn't Work for Strings

```
String magicWord = "abacadabra";
String otherMagicWord = "abacadabra";

// this expression IS true
if (magicWord == magicWord)
{
}
```

A reference is equal to itself (they clearly point to the same memory location)

Why == Doesn't Work for Strings

```
String magicWord = "abacadabra";
String otherMagicWord = magicWord;

// this expression IS true
if (magicWord == otherMagicWord)
{
}
```

Why == Doesn't Work for Strings

```
String magicWord = "abacadabra";
String otherMagicWord = magicWord;

// this expression IS true
if (magicWord == otherMagicWord)
{
}
```

Both variables point to the same location in memory

Why == Doesn't Work for Strings

```
String magicWord = "abacadabra";
String otherMagicWord = "abacadabra";

// this expression IS true
if (magicWord.equals(otherMagicWord))
{
}
```

Why == Doesn't Work for Strings

```
String magicWord = "abacadabra";
String otherMagicWord = "abacadabra";

// this expression IS true
if (magicWord.equals(otherMagicWord))
{
}
```

The `String` `equals` method checks whether the `String`s are logically equivalent, character-by-character

Why == Doesn't Work for Strings

```
String magicWord = "abacadabra";
String otherMagicWord = "abacadabra";

// this expression is also true
// (wait… what?)
if ("abacadabra" == "abacadabra")
{
}
```

Why == Doesn't Work for Strings

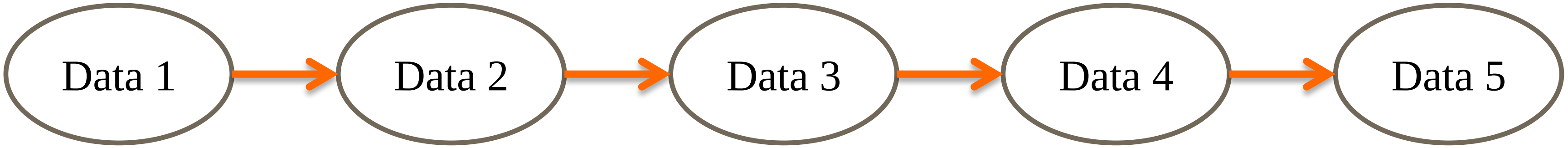
```
String magicWord = "abacadabra";
String otherMagicWord = "abacadabra";

// this expression is also true
// (wait… what?)
if ("abacadabra" == "abacadabra")
{
}
```

Java is clever about constants and doesn't store them as two separate objects!

Linked Lists

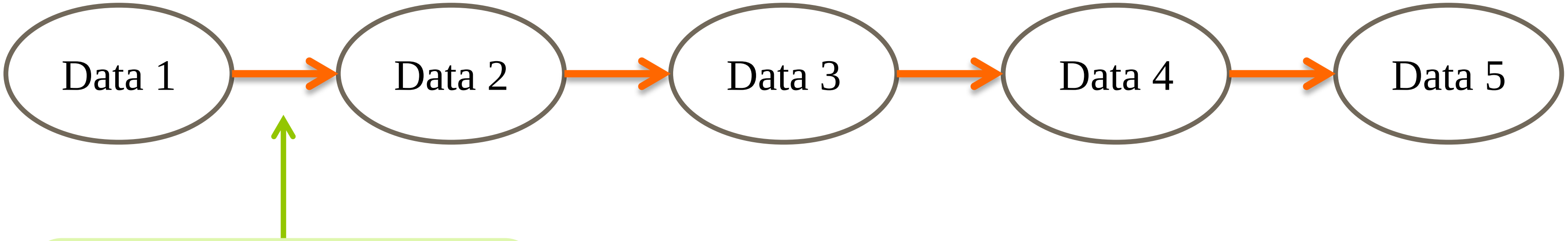
Linked Lists

Conceptually:



Linked Lists

Conceptually:



Data 1

Data 2

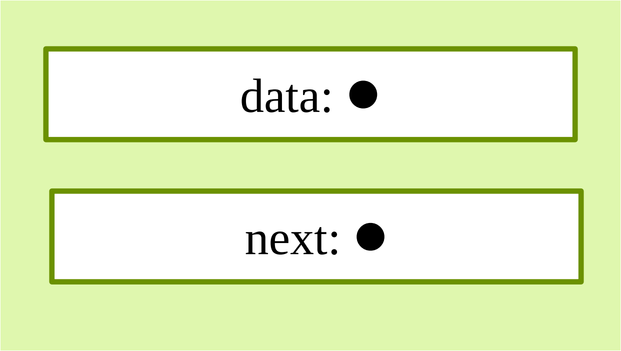
Data 3

Data 4

Data 5

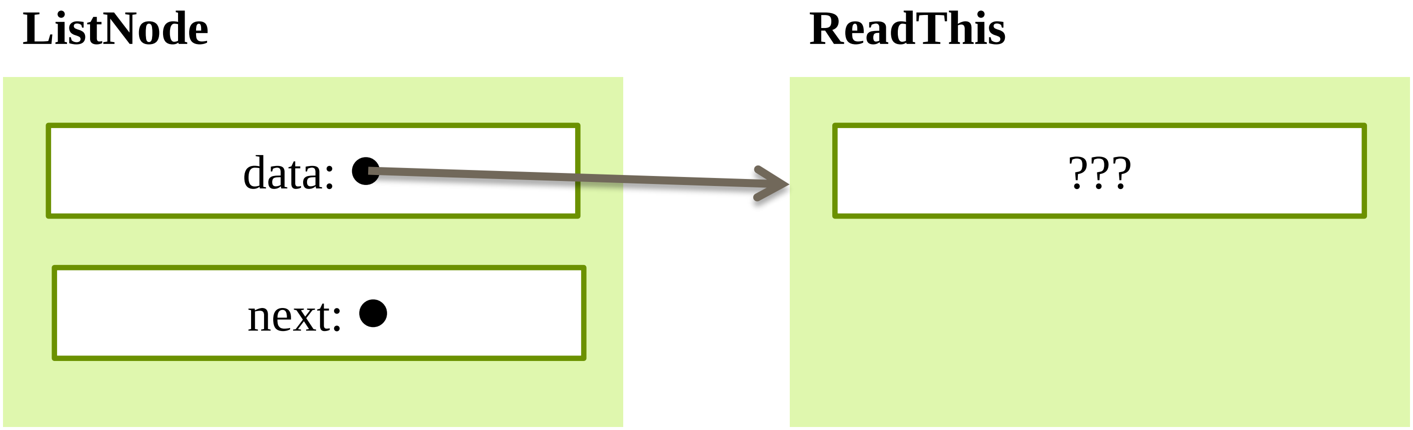
We want to be able to break and rearrange the links easily

ListNode



data: ●

next: ●



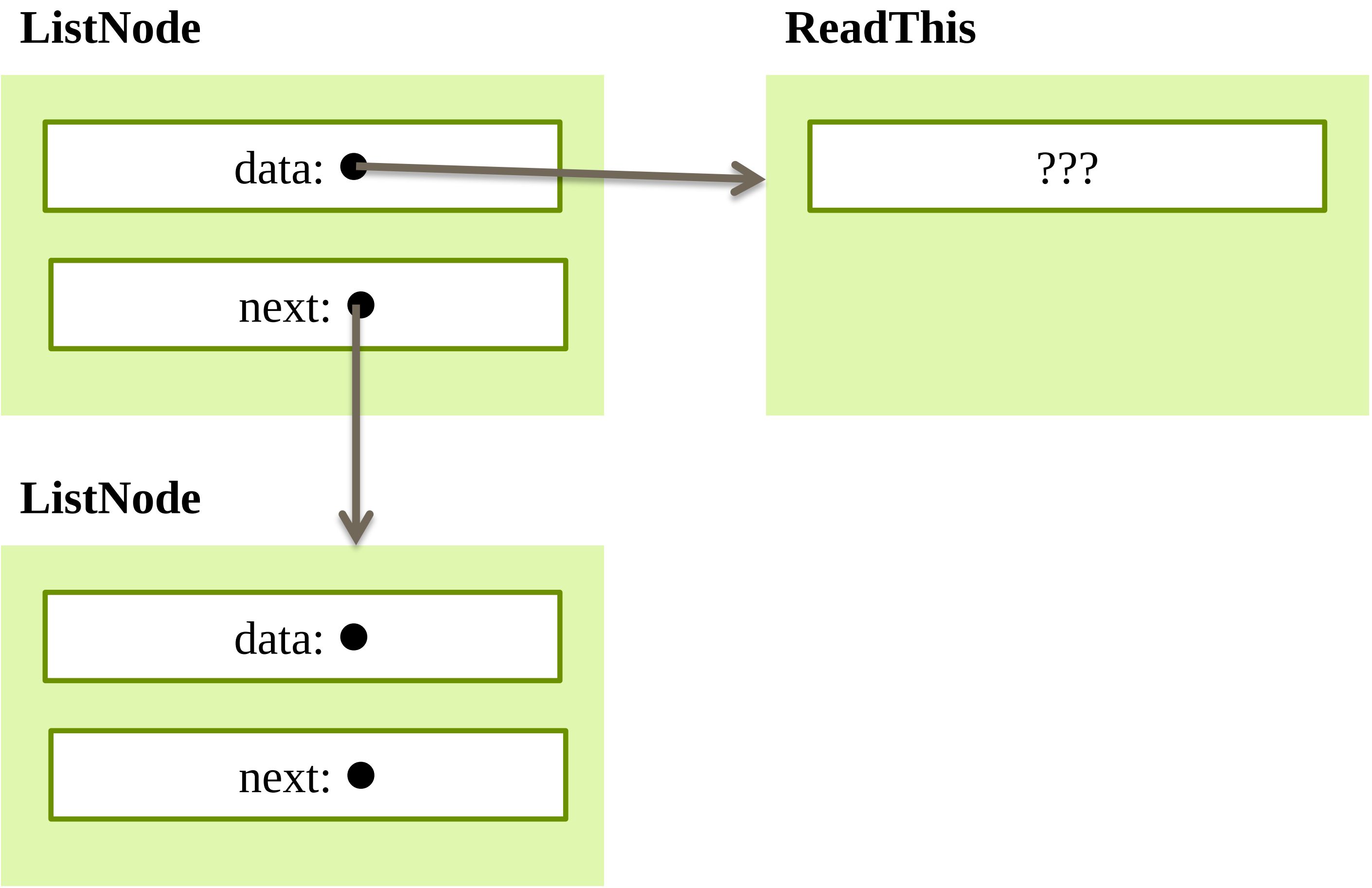
ListNode

data: ●

next: ●

ReadThis

???



ListNode

data: ●

next: ●

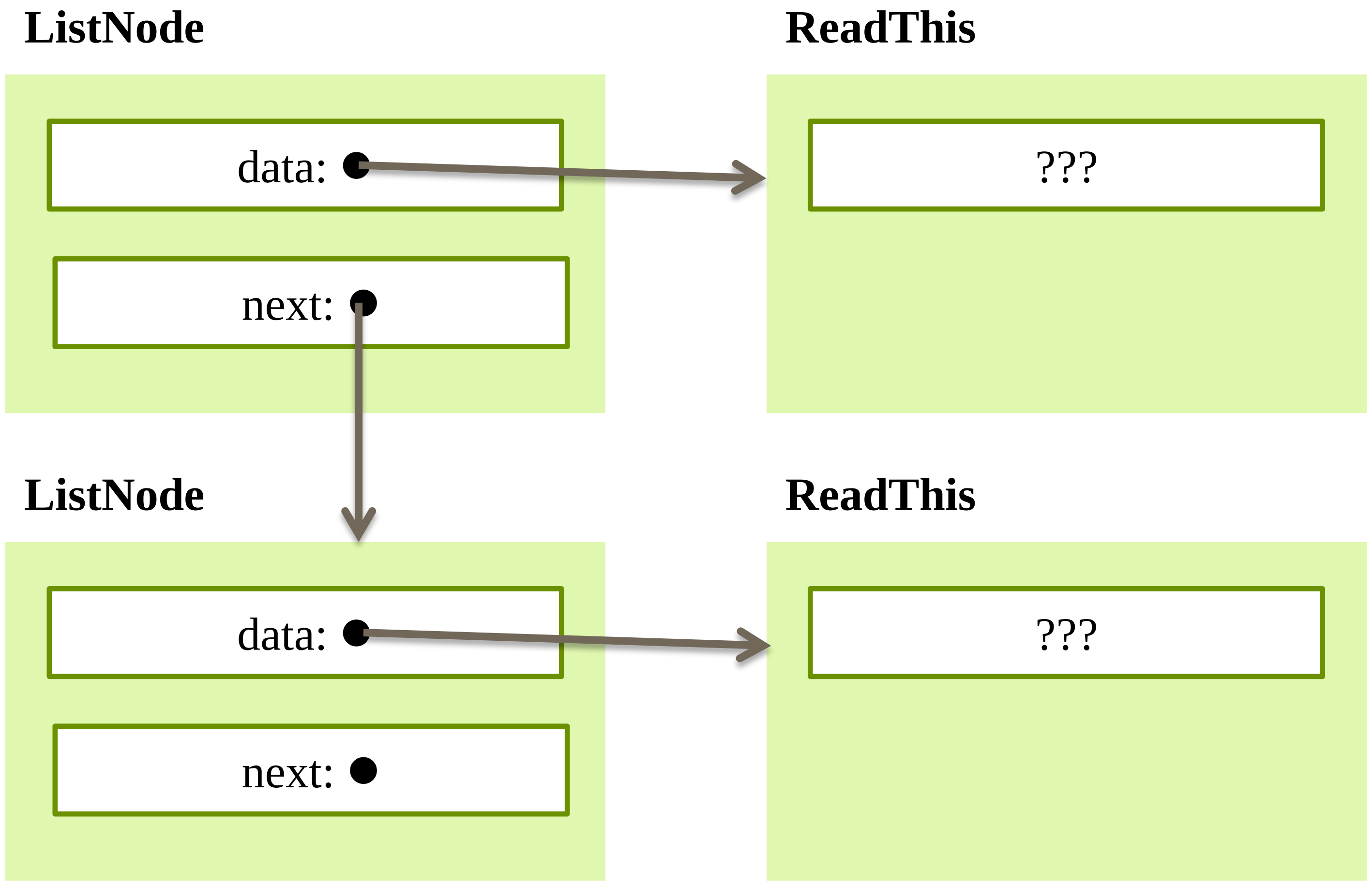
ReadThis

???

ListNode

data: ●

next: ●



ListNode

data:

next:

ReadThis

???

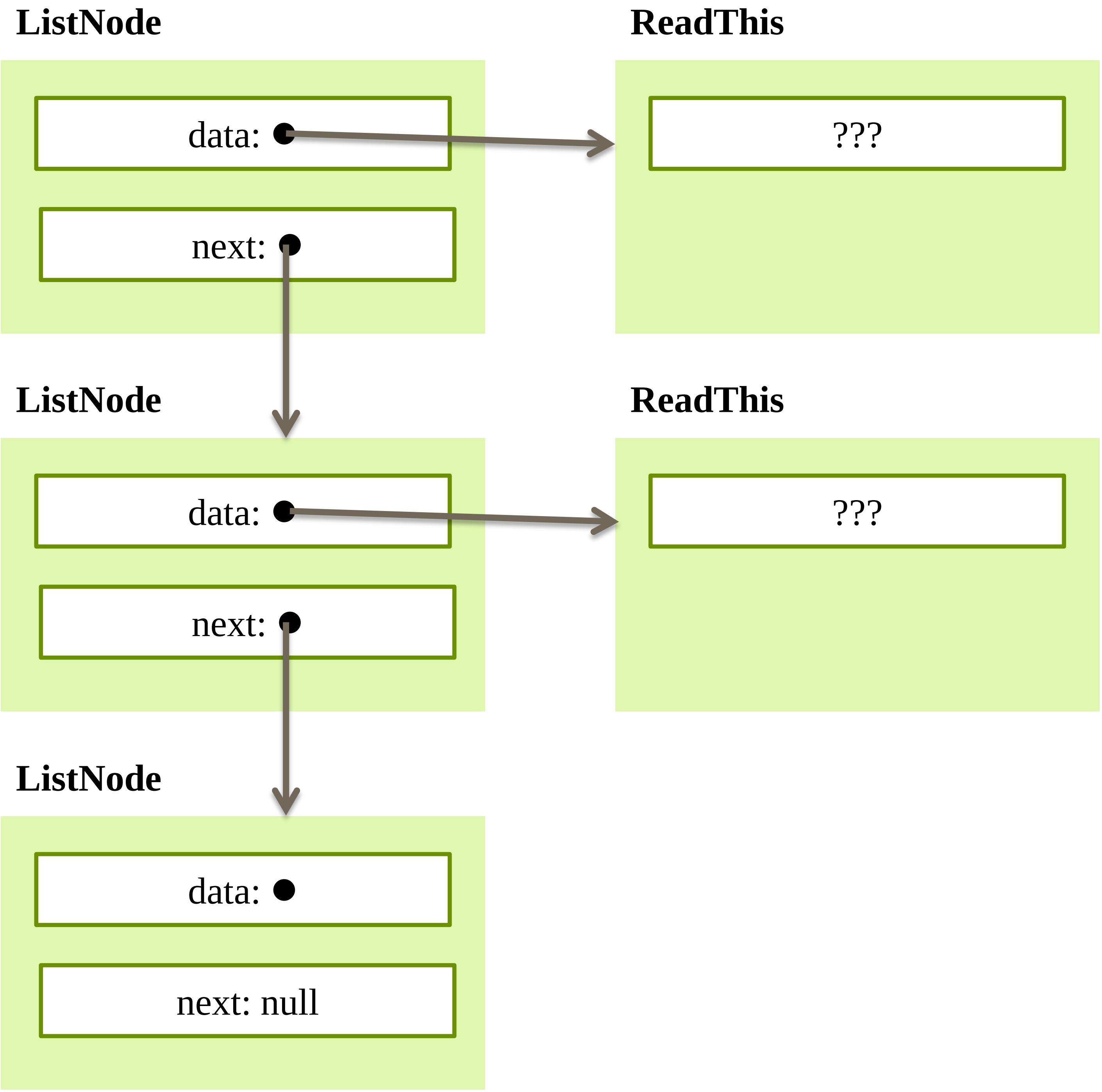
ListNode

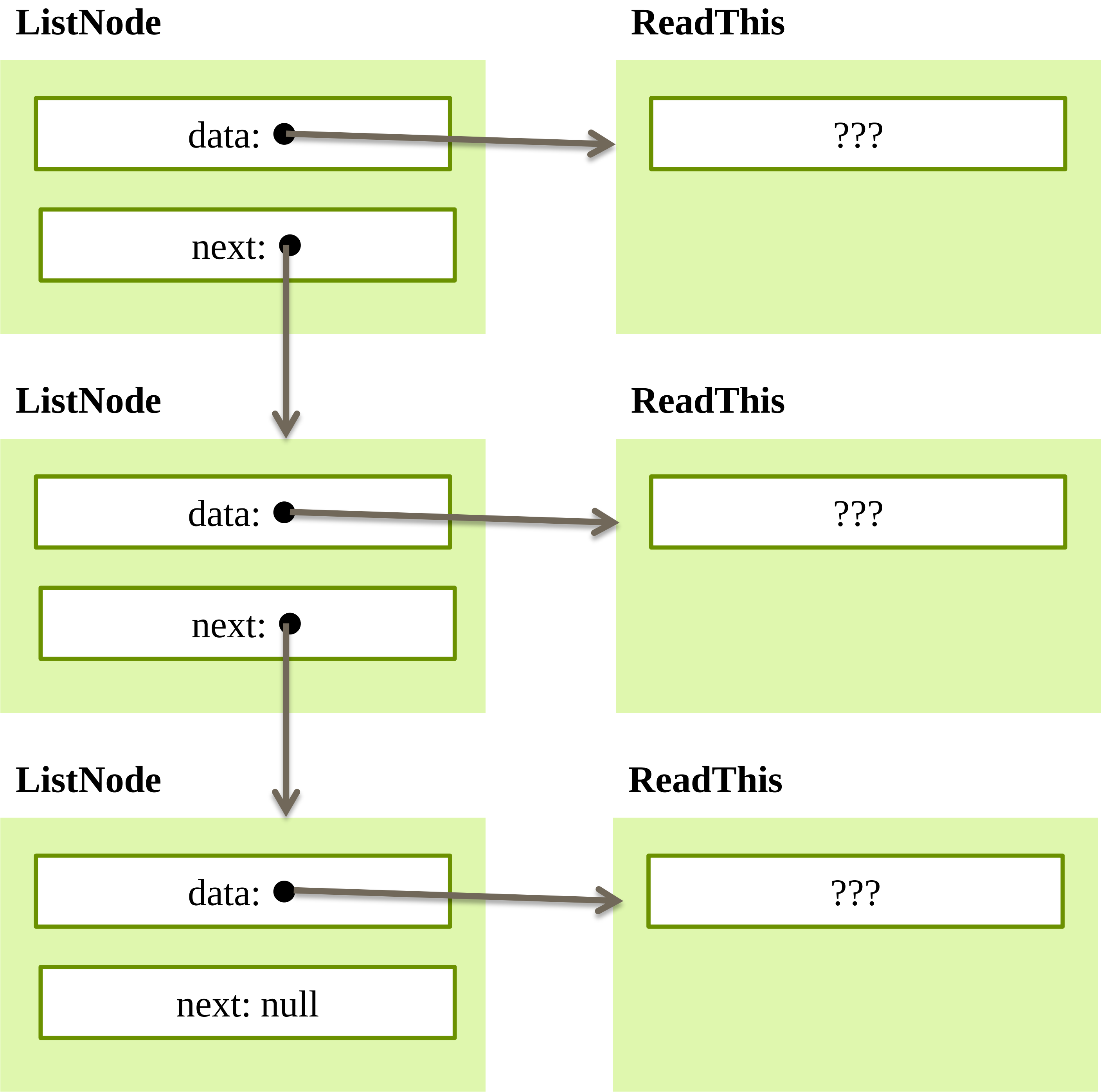
data:

next:

ReadThis

???


ListNode
data: ●
next: ●
ReadThis
???
ListNode
data: ●
next: ●
ReadThis
???
ListNode
data: ●
next: null



ListNode

data: ●

next: ●

ReadThis

???

ListNode

data: ●

next: ●

ReadThis

???

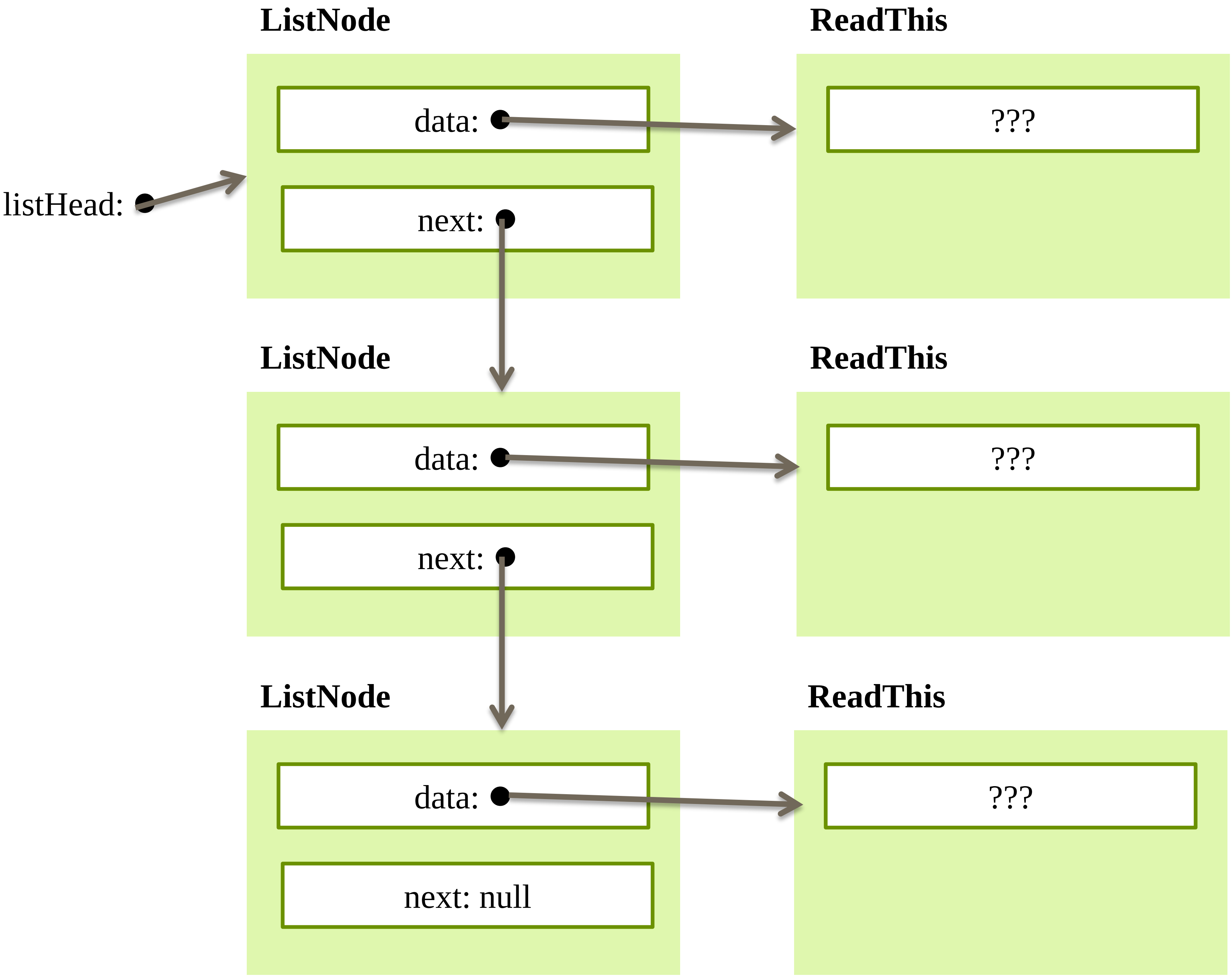
ListNode

data: ●

next: null

ReadThis

???



ListNode

ReadThis

data:

???

listHead:

next:

ListNode

ReadThis

data:

???

next:

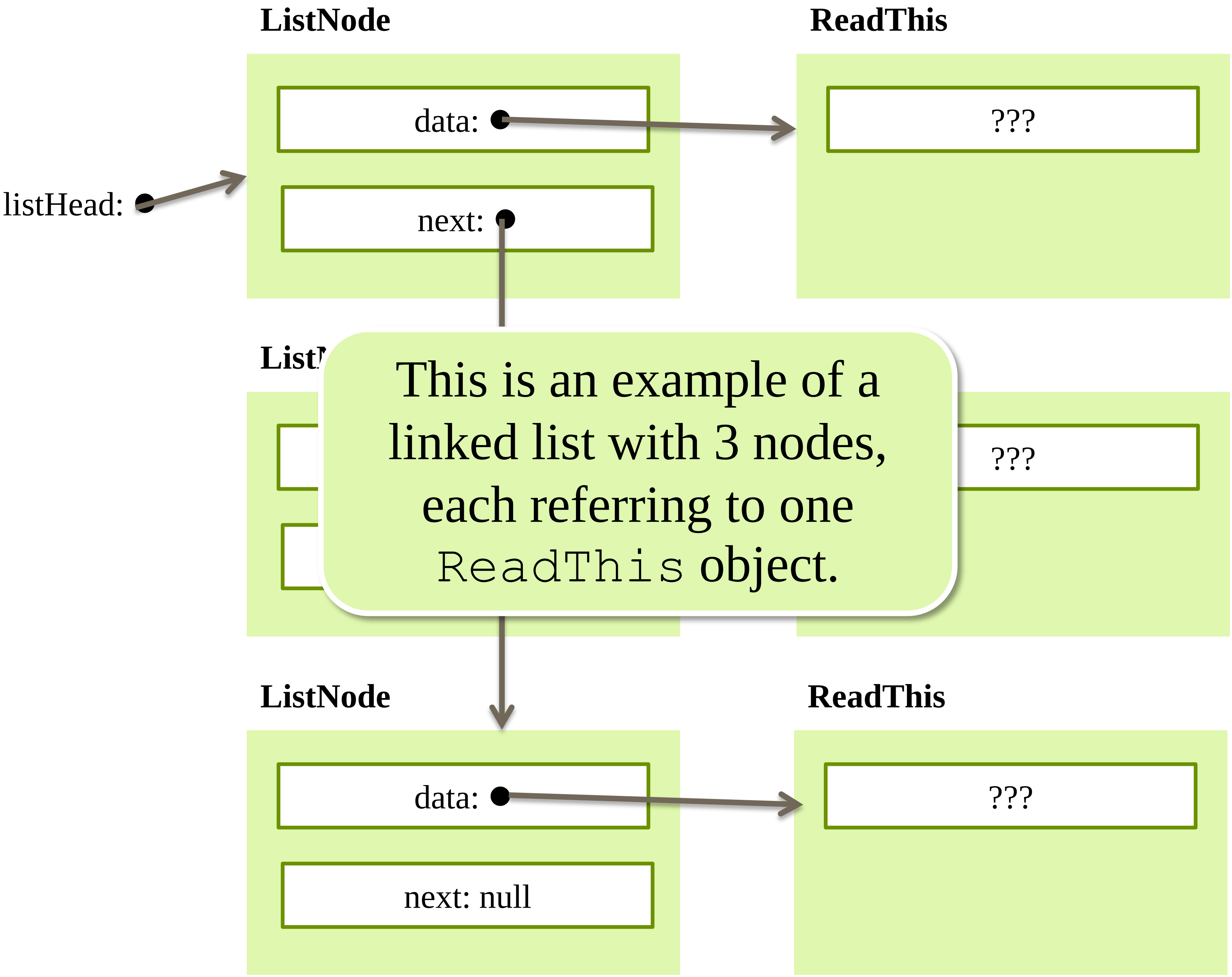
ListNode

ReadThis

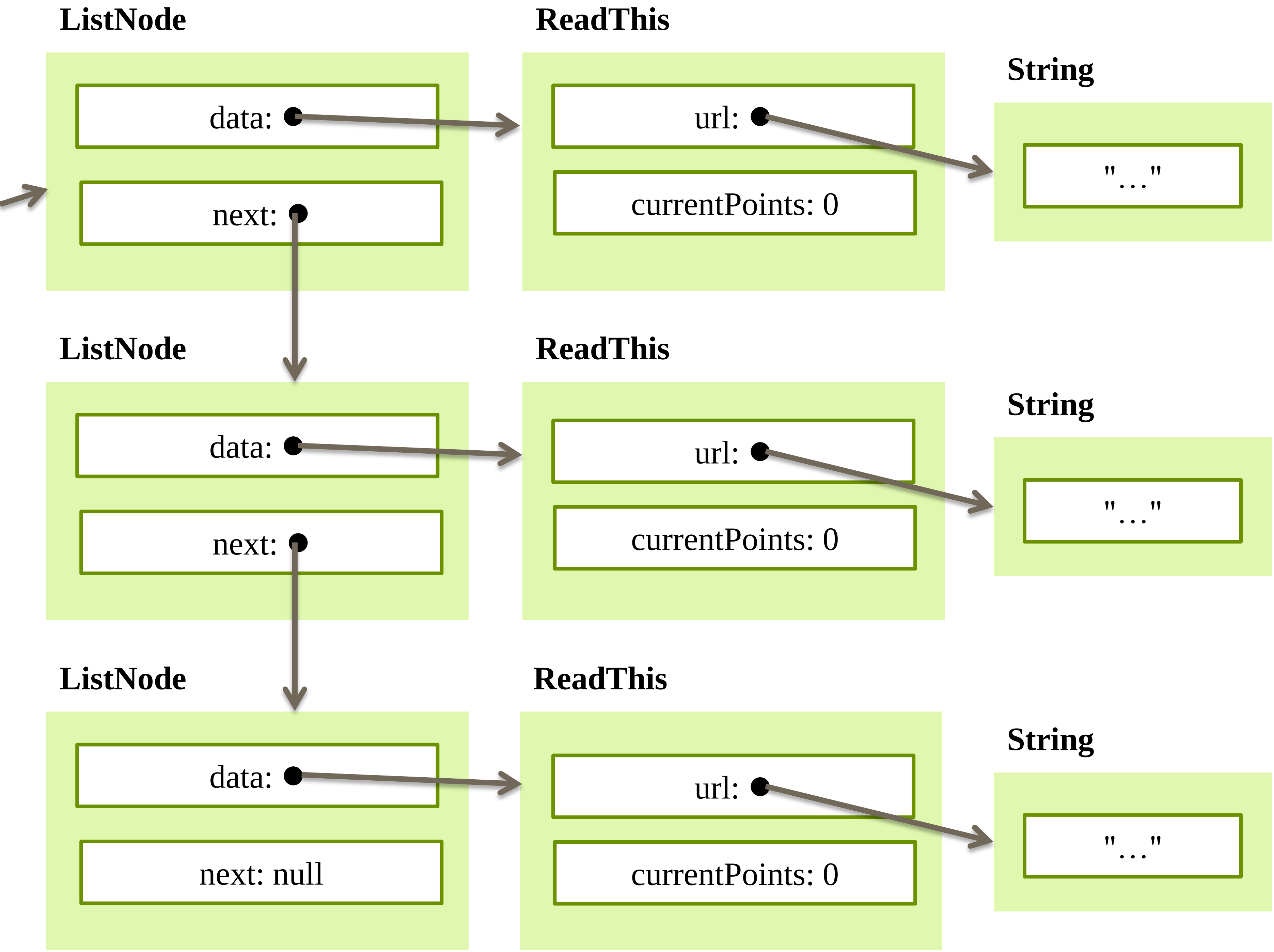
data:

???

next: null



This is an example of a linked list with 3 nodes, each referring to one `ReadThis` object.


ListNode
data: ●
next: ●
ReadThis
url: ●
currentPoints: 0
String
"..."
ListNode
data: ●
next: ●
ReadThis
url: ●
currentPoints: 0
String
"..."
ListNode
data: ●
next: null
ReadThis
url: ●
currentPoints: 0
String
"..."

Linked List Operations

We want to be able to:

1. Add new nodes to the beginning or end of the list.
2. Add new nodes in the middle.
3. Remove nodes from the beginning or end.
4. Remove nodes from the middle.
5. Get the size of the list.

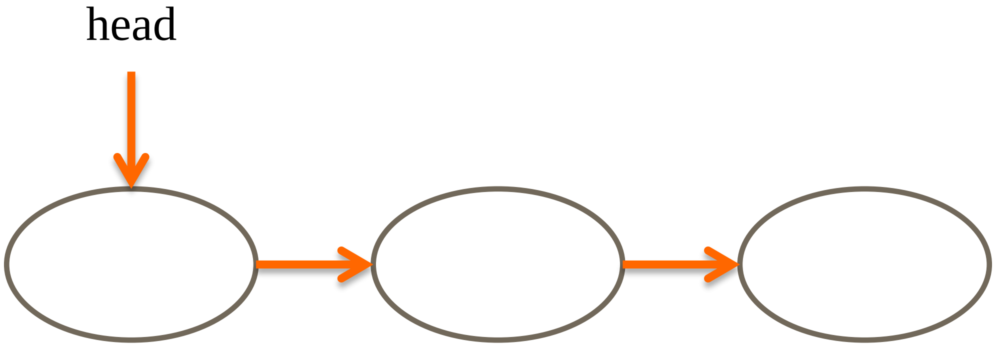
Linked List Operations

We want to be able to:

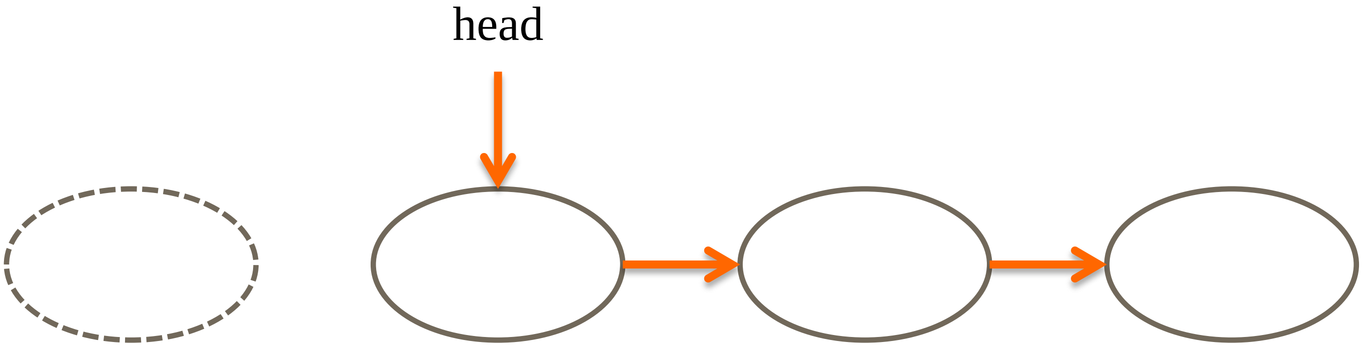
1. Add new nodes to the beginning or end of the list.
2. Add new nodes in the middle.
3. Remove nodes from the beginning or end.
4. Remove nodes from the middle.
5. Get the size of the list.

These will be illustrated here, all will be implemented in code.

Add Nodes to Beginning

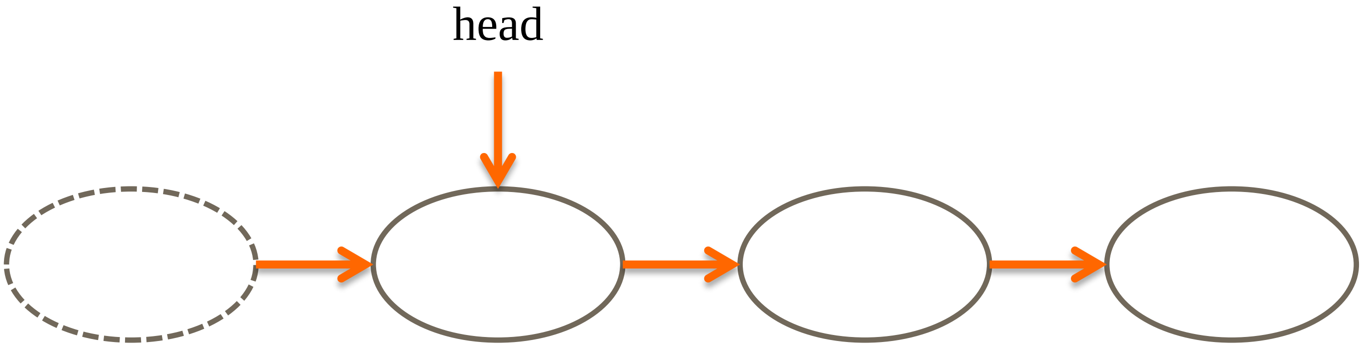


Add Nodes to Beginning



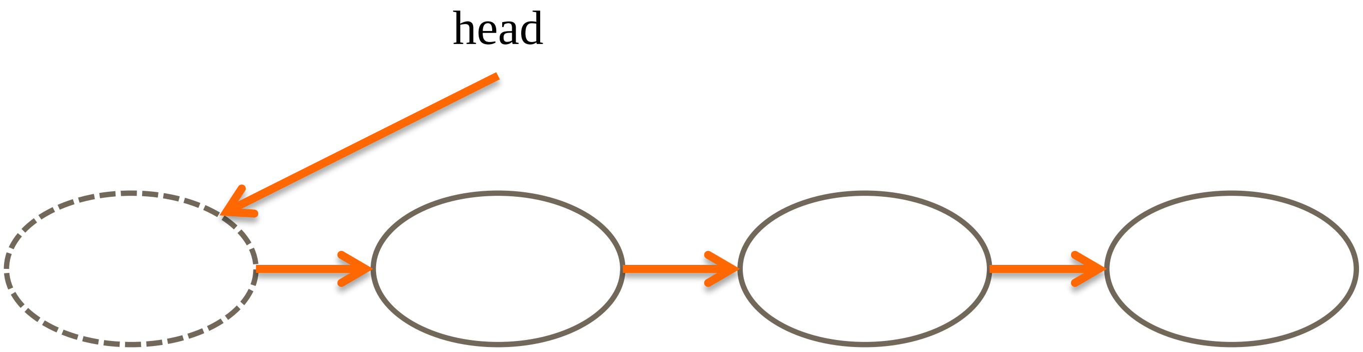
```
ListNode newNode = new ListNode();
```

Add Nodes to Beginning



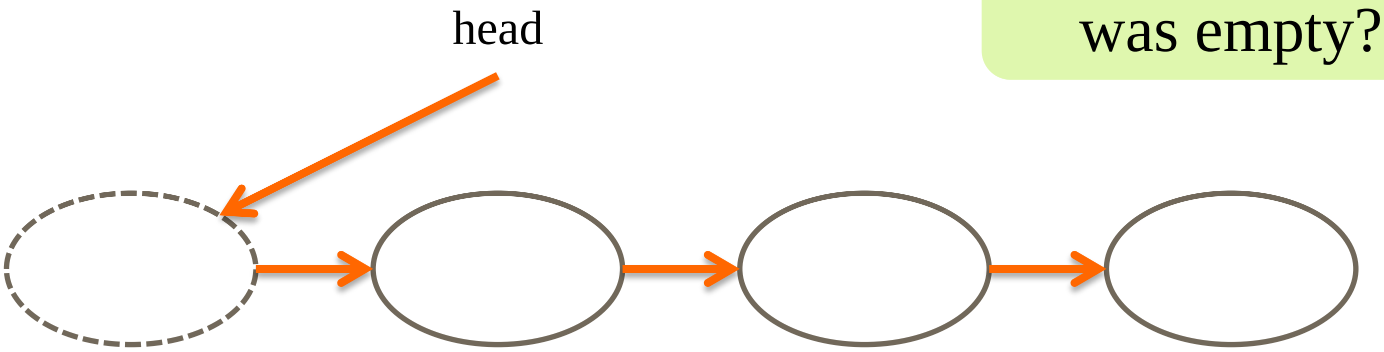
```
ListNode newNode = new ListNode();
newNode.next = head;
```

Add Nodes to Beginning



```
ListNode newNode = new ListNode();
newNode.next = head;
head = newNode;
```

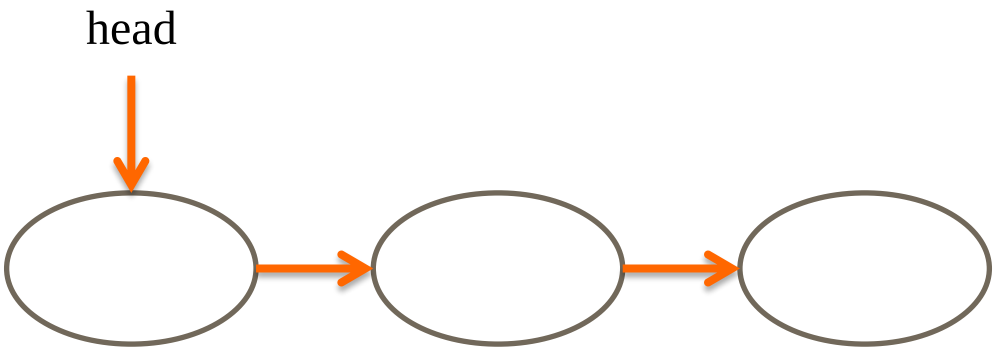
Add Nodes to Beginning



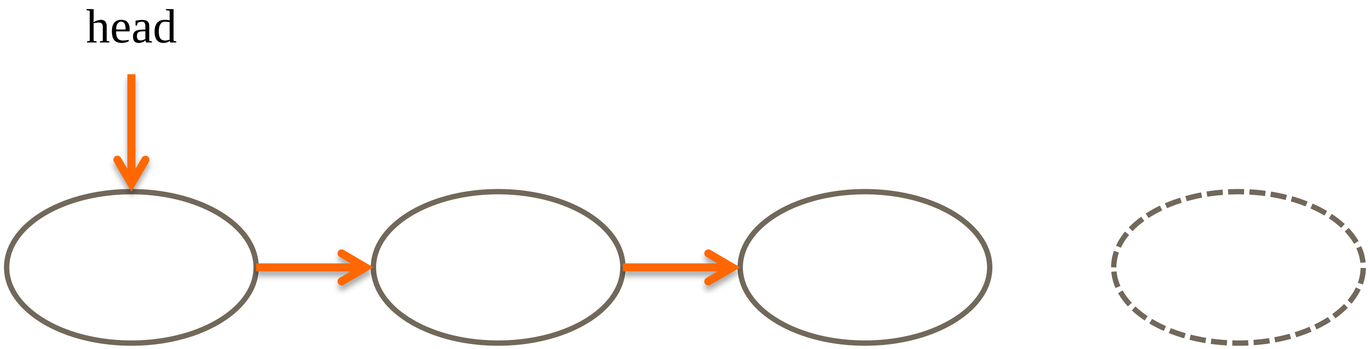
What if the list was empty?

```
ListNode newNode = new ListNode();
newNode.next = head;
head = newNode;
```

Add Nodes to End

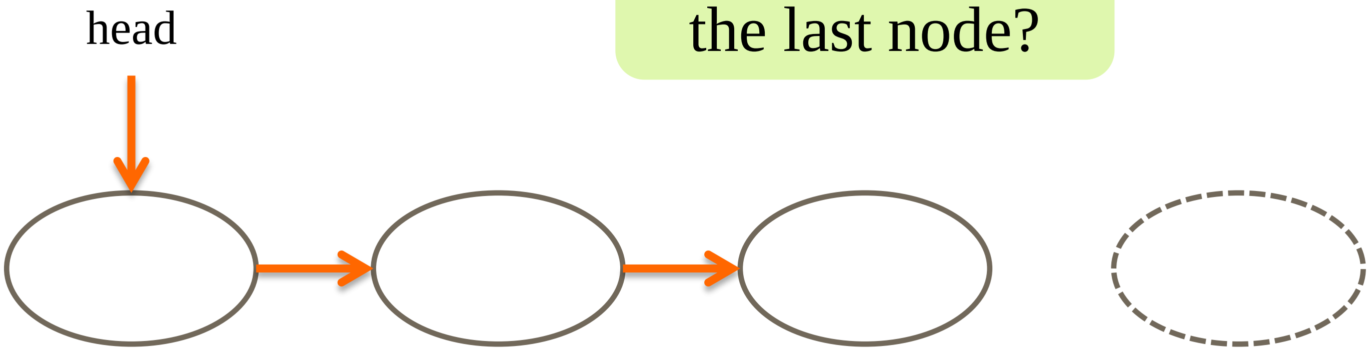


Add Nodes to End



```
ListNode newNode = new ListNode();
```

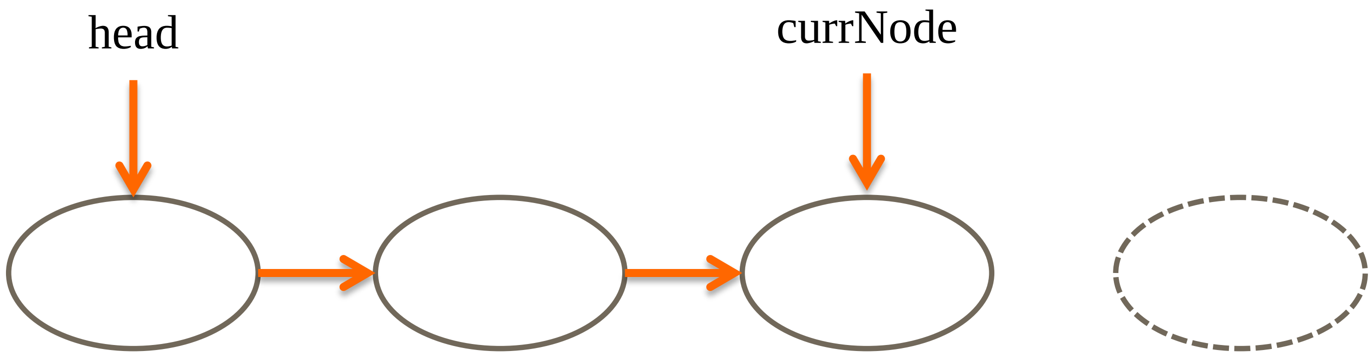
Add Nodes to End



How do we find the last node?

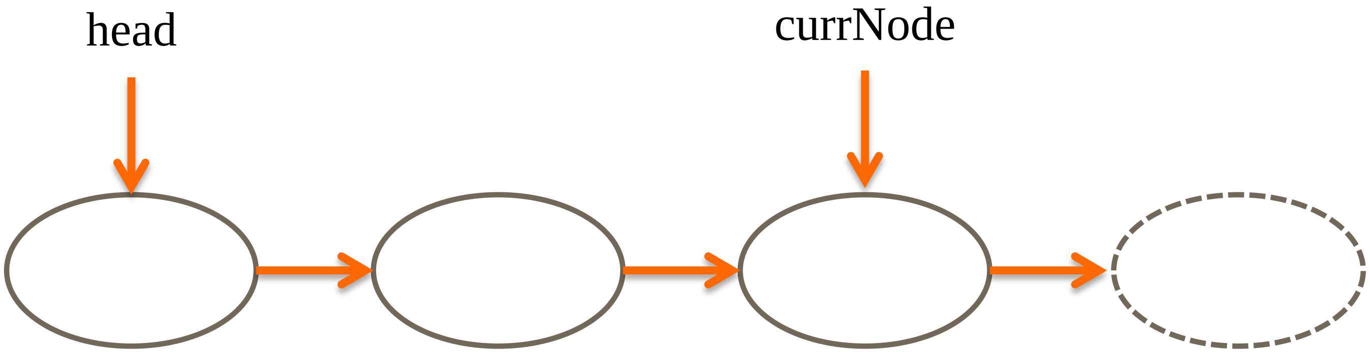
```
ListNode newNode = new ListNode();
```

Add Nodes to End



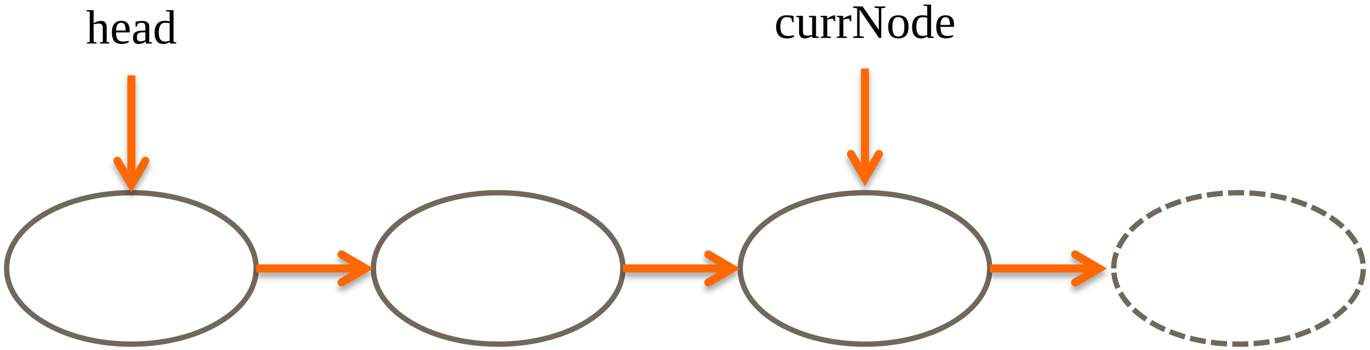
```
ListNode newNode = new ListNode();
ListNode currNode = head;
while (currNode.next != null)
{
   currNode = currNode.next;
}
```

Add Nodes to End



```
ListNode newNode = new ListNode();
ListNode currNode = head;
while (currNode.next != null)
{
   currNode = currNode.next;
}
currNode.next = newNode;
```

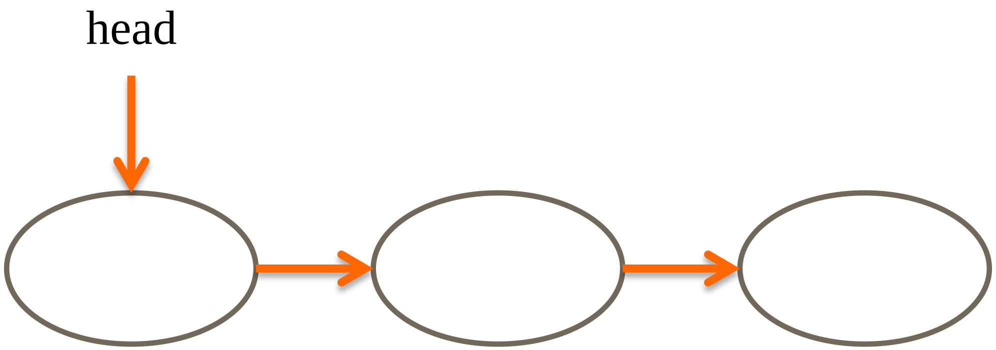
Add Nodes to End



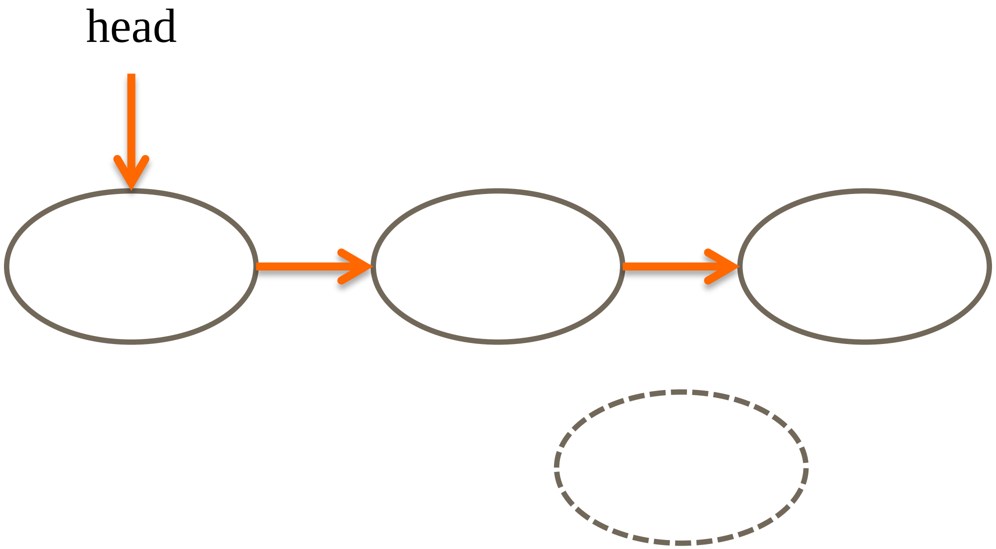
```
ListNode newNode = new ListNode();
ListNode currNode = head;
while (currNode.next != null)
{
   currNode = currNode.next;
}
currNode.next = newNode;
```

Again, what if the list was empty?

Add Nodes to Middle

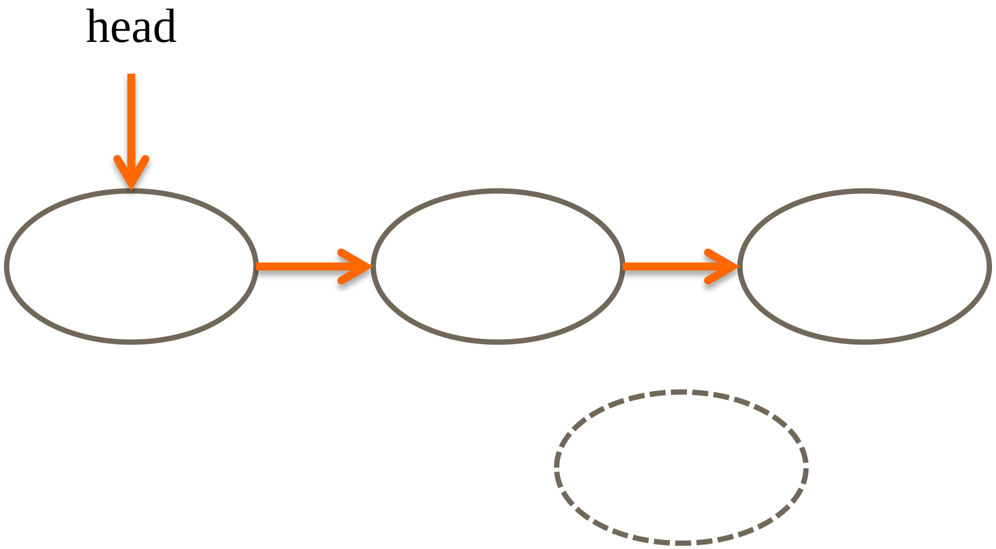


Add Nodes to Middle



```
ListNode newNode = new ListNode();
```

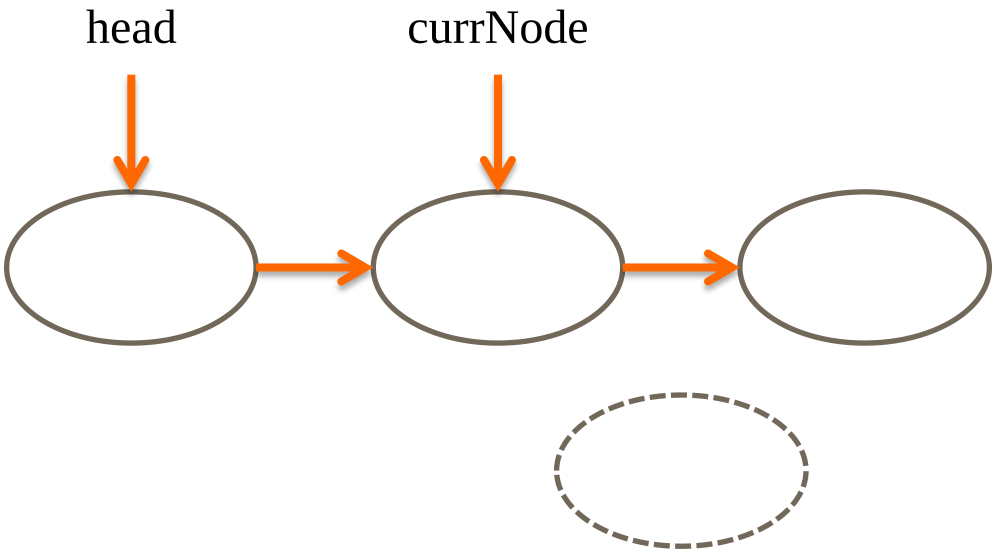
Add Nodes to Middle



```
ListNode newNode = new ListNode();
```

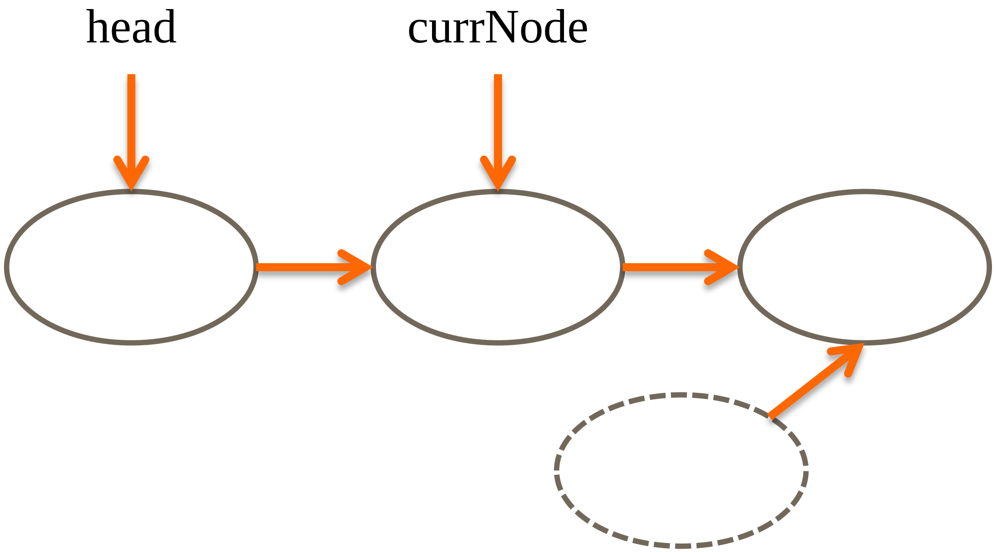
How do we find the node we want to insert after?

Add Nodes to Middle



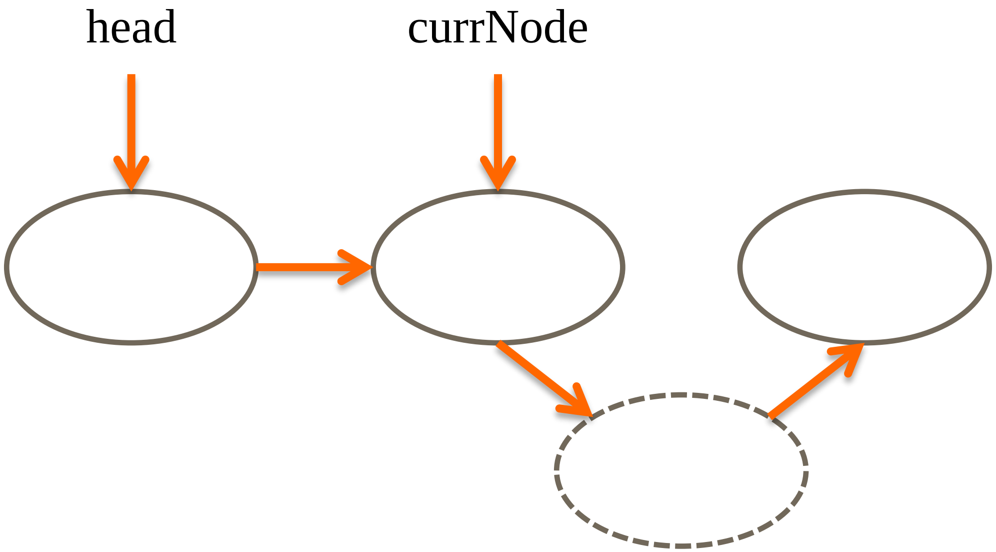
```
ListNode newNode = new ListNode();
ListNode currNode = head;
while (currNode != null &&
       !currNode.data.equals(insertAfter.data))
{
   currNode = currNode.next;
}
```

Add Nodes to Middle



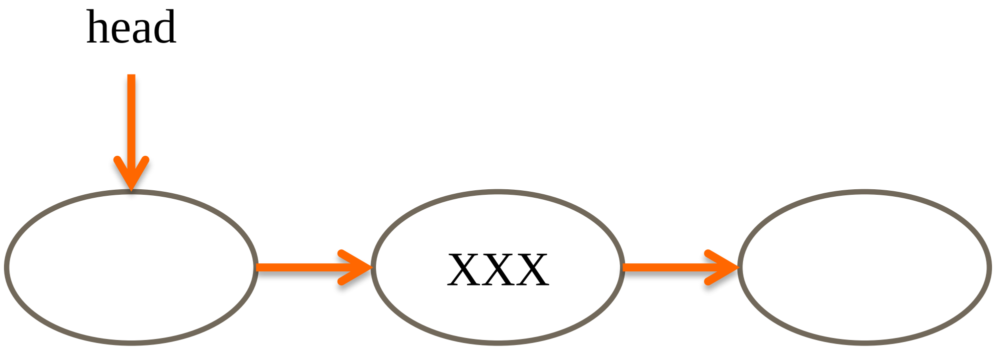
```
if (currNode != null)
{
    newNode.next = currNode.next;
}
```

Add Nodes to Middle

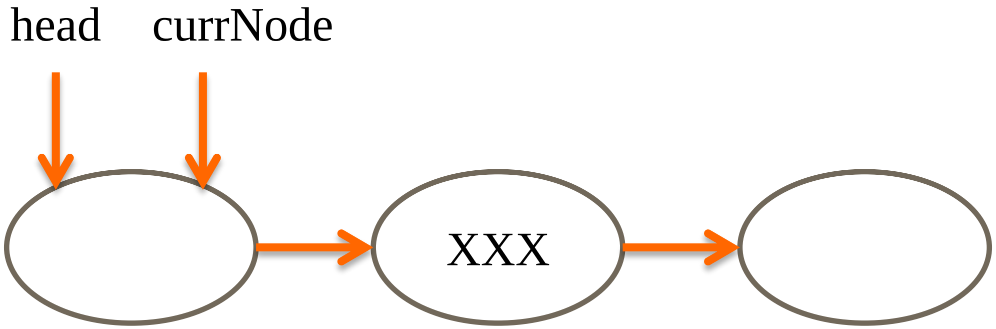


```
if (currNode != null)
{
    newNode.next = currNode.next;
    currNode.next = newNode;
}
```

Remove Nodes from Middle

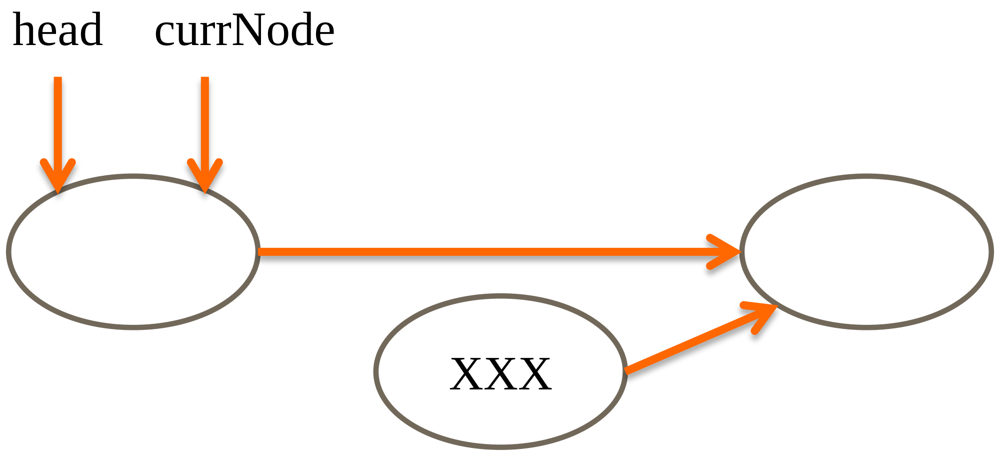


Remove Nodes from Middle



```
ListNode currNode = head;
while (currNode.next != null &&
      !currNode.next.data.equals(deleteNode.data))
{
  currNode = currNode.next;
}
```

Remove Nodes from Middle



```
ListNode currNode = head;
while (currNode.next != null &&
      !currNode.next.data.equals(deleteNode.data))
{
  currNode = currNode.next;
}
```

`currNode.next = currNode.next.next;`