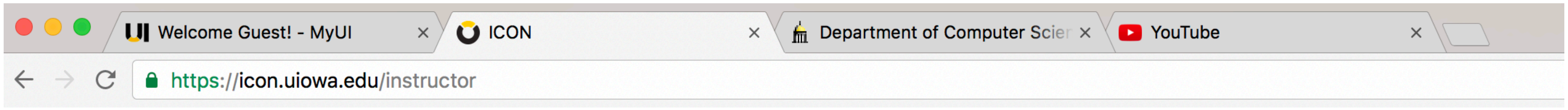
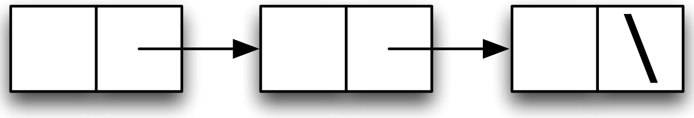
# Tabs in your browser...


Welcome Guest! - MyUI
ICON
Department of Computer Scie
YouTube
https://icon.uiowa.edu/instructor

A linked list is helpful here!!

...why?

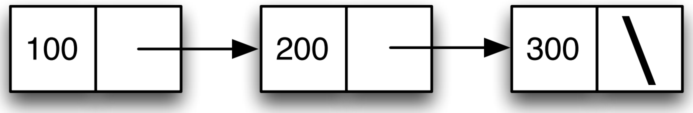


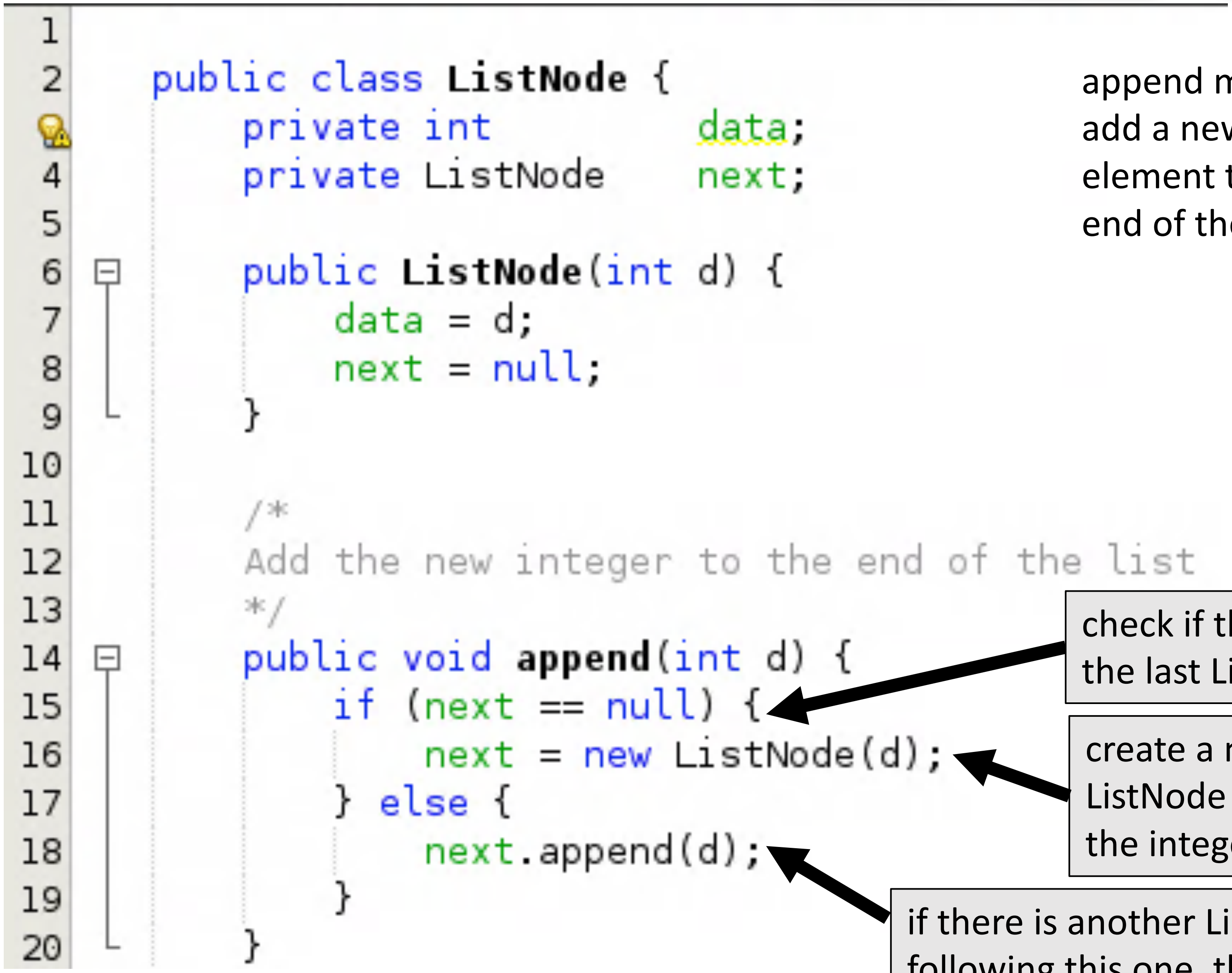
## Today’s big ideas 2

- write some methods for ListNode using iteration (loops) or recursion
- Don’t use dot ( . ) if your reference could be `null`!
- ***encapsulate*** ListNodes inside of a ***LinkedList*** class so we can try different implementations of a linked list
- LinkedLists be empty, so we have to check for this case. A ***sentinel node*** provides a useful invariant (header!=null) that simplifies code

# The *append* method

example linked list





```
public class ListNode {
    private int          data;
    private ListNode     next;

    public ListNode(int d) {
        data = d;
        next = null;
    }

    /*
    Add the new integer to the end of the list
    */
    public void append(int d) {
        if (next == null) {
            next = new ListNode(d);
        } else {
            next.append(d);
        }
    }
```

append means add a new element to the end of the list

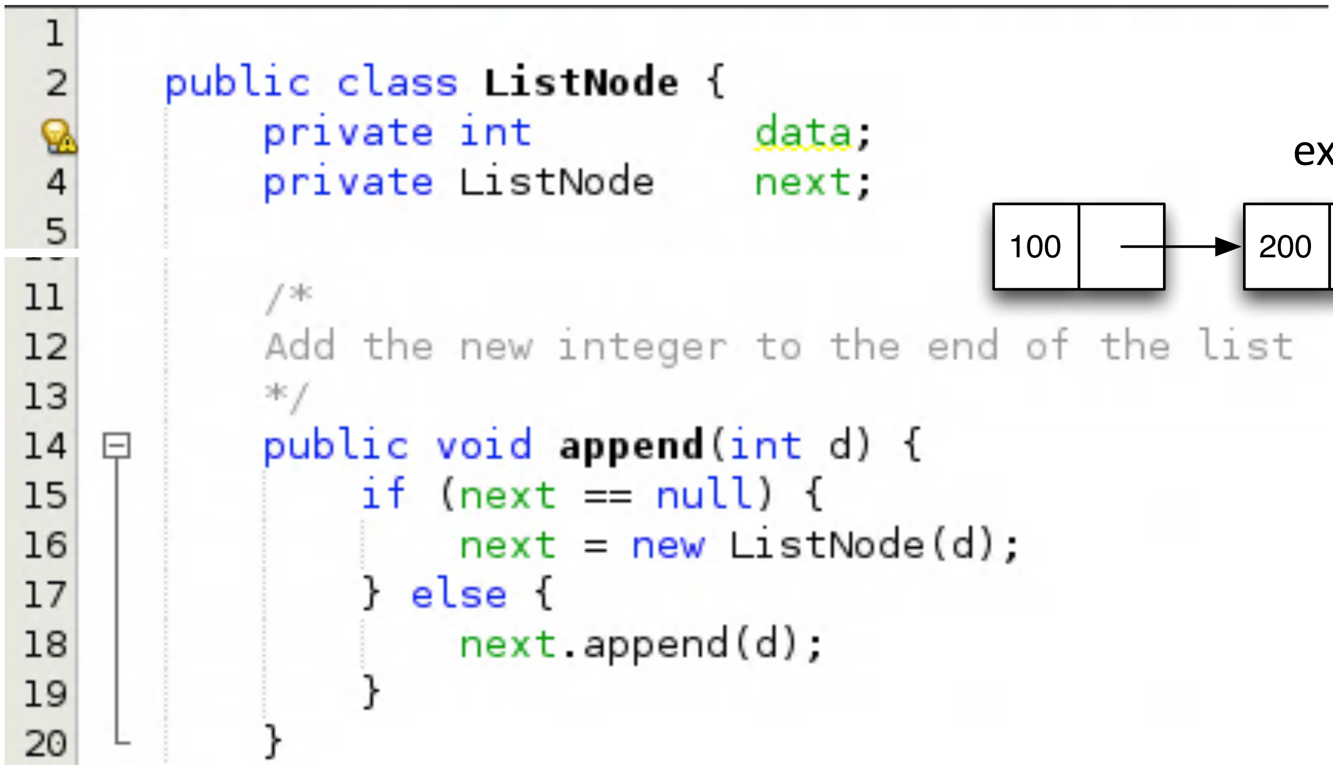
check if this is the last ListNode

create a new ListNode to hold the integer

if there is another ListNode following this one, then append to that one

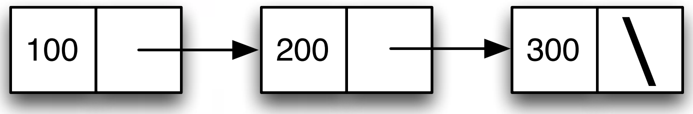
# The *append* method

https://b.socrative.com/login/student/
CS2230X – 1000-2999
CS2230Y – 3000-5999



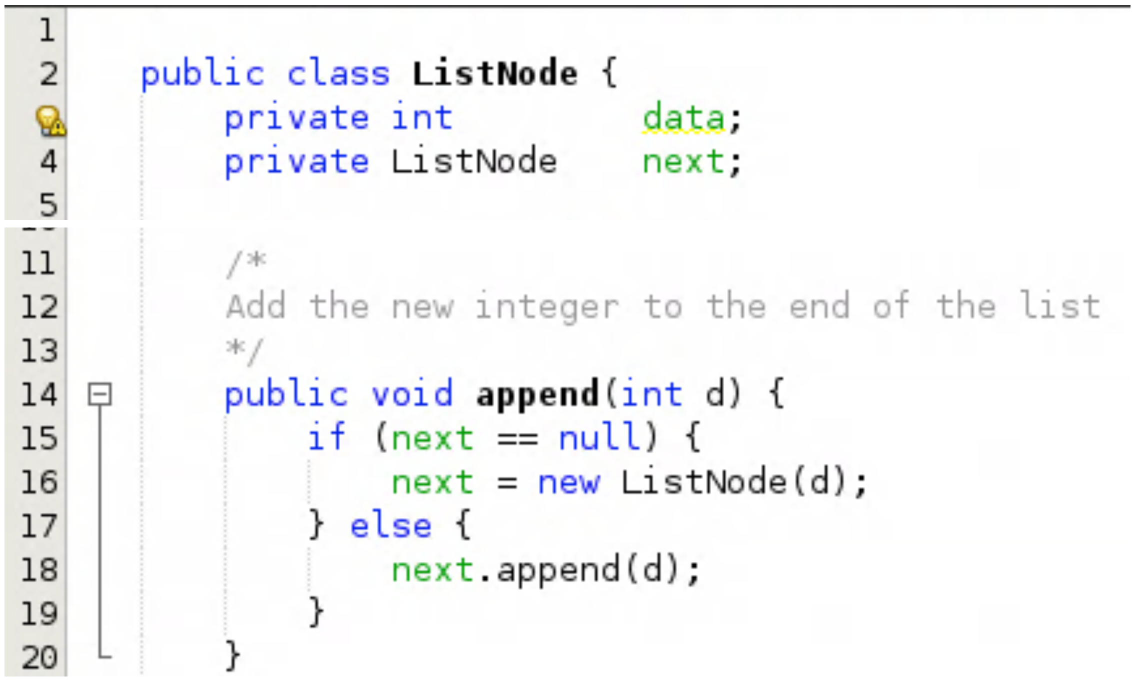
```
 1
 2   public class ListNode {
         private int          data;
 4       private ListNode     next;
 5
11       /*
12       Add the new integer to the end of the list
13       */
14       public void append(int d) {
15           if (next == null) {
16               next = new ListNode(d);
17           } else {
18               next.append(d);
19           }
20       }
```

example linked list



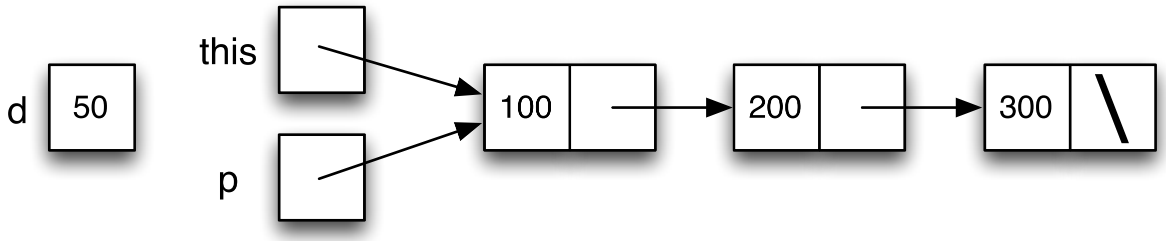
How does the append method traverse (i.e. walk node to node) the linked list?

a) line 18: the Java keyword “next” takes us to the following node in a linked list
b) line 18: by calling append again, it will affect a different ListNode than before
c) line 18: calling append on a different value of d
d) line 18: next looks at the reference to the following ListNode, the dot follows the reference to the actual ListNode object, then we call append on it
e) line 16: assigning next to a new ListNode brings us to the following ListNode



```
 1
 2  public class ListNode {
 3      private int             data;
 4      private ListNode        next;
 5

11      /*
12      Add the new integer to the end of the list
13      */
14      public void append(int d) {
15          if (next == null) {
16              next = new ListNode(d);
17          } else {
18              next.append(d);
19          }
20      }
```

Here is the boxes and arrows diagram right after p.append(50) is called and we are on line 15.

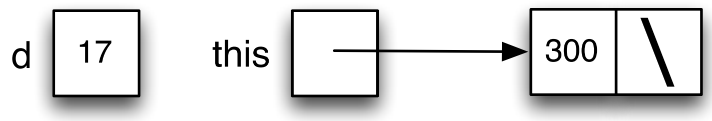


Draw the boxes and arrows diagram when we get to line 15 *again*.

# Iterative (for-loop) implementation of append

```
1 // where is the mistake?
2 public void append(int d) {
3     ListNode current = this;
4     while (current != null) {
5         current = current.next;
6     }
7     current.next = new ListNode(d);
8 }
```

HINT: here is the boxes-and-arrows for an example list, when we are on line 3...



...what is the boxes-and-arrows when we've reached line 7 (not yet executed line 7)?

## The bug in iterative append

```
1 // where is the mistake?
2 public void append(int d) {
3     ListNode current = this;
4     while (current.next != null) {
5         current = current.next;
6     }
7     current.next = new ListNode(d);
8 }
```

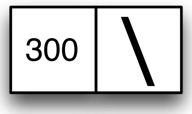
current=null
after line 6

you can’t dereference a
reference that is null

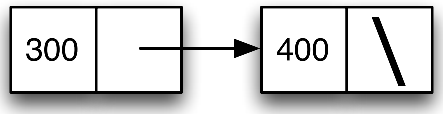
# The fix for iterative append

```
// where is the mistake?
public void append(int d) {
    ListNode current = this;
    while (current.next != null) {
        current = current.next;
    }
    current.next = new ListNode(d);
}
```

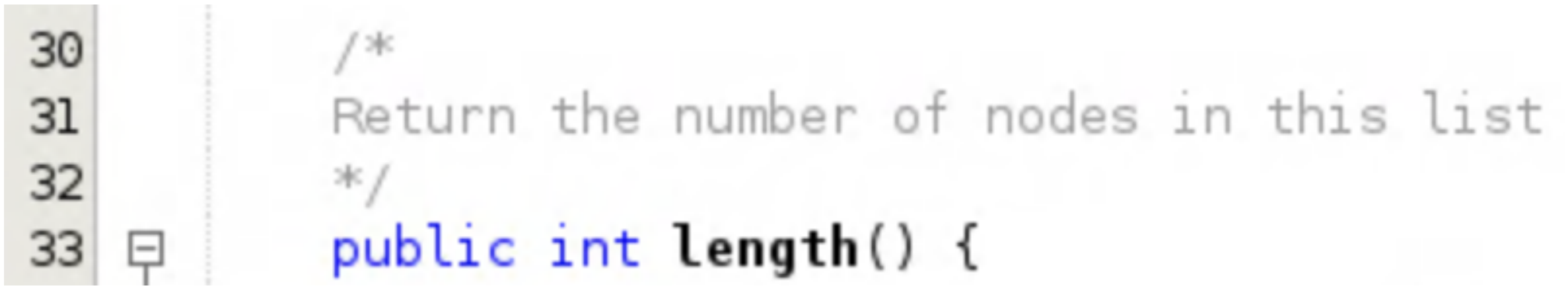
we've found the ListNode that looks like



now set its next field

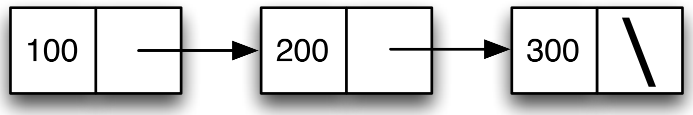


## Method to get length of the list



```
/*
Return the number of nodes in this list
*/
public int length() {
```

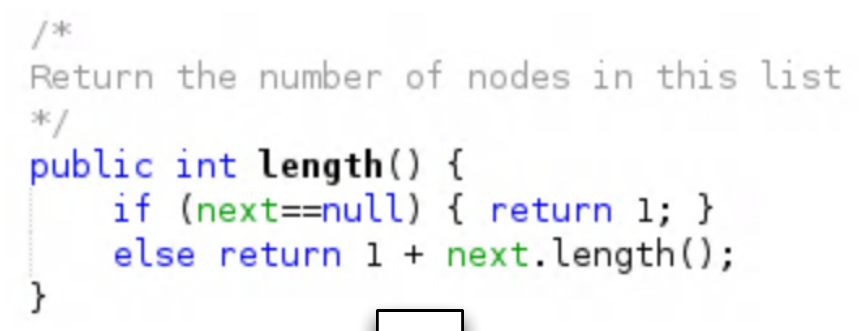
What should be the algorithm for our implementation of length()?



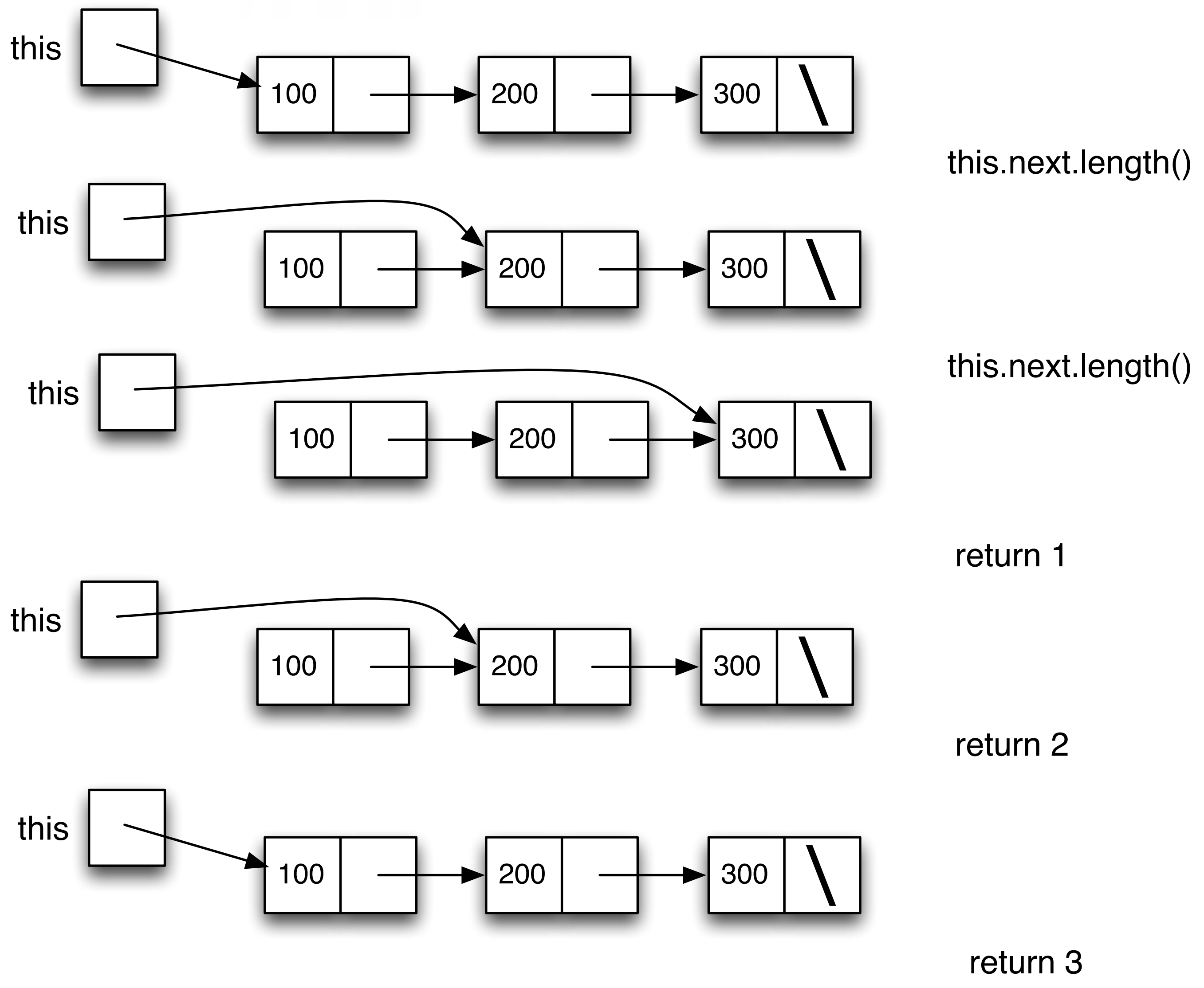
length() returns 3

- *answer in words*
- *then give an example of calling length() on the above list by illustrating in terms of some boxes-and-arrows diagrams*

## One implementation of length()

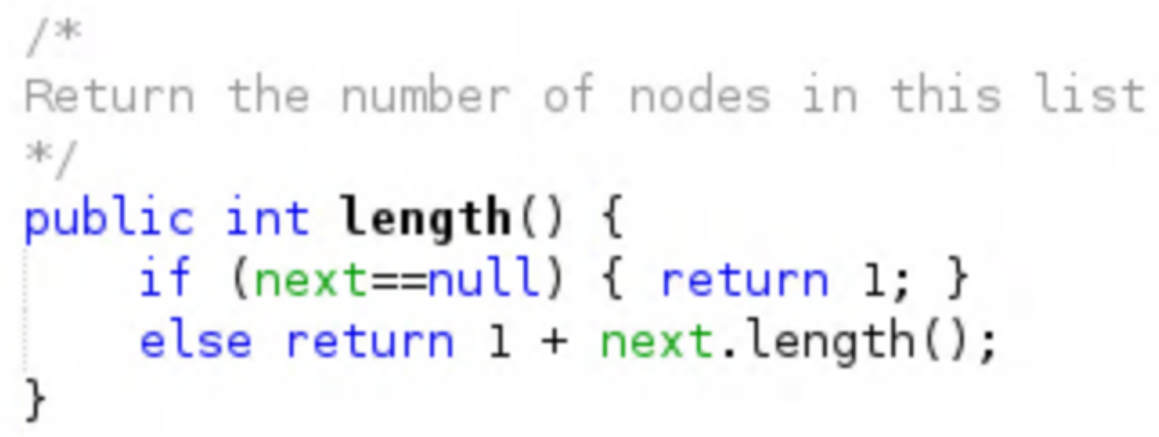


```
/*
Return the number of nodes in this list
*/
public int length() {
    if (next==null) { return 1; }
    else return 1 + next.length();
}
```



https://b.socrative.com/login/student/
CS2230X – 1000-2999
CS2230Y – 3000-5999

If it takes 1ms to find the length of a list length 10, how long for a list of size 10,000?



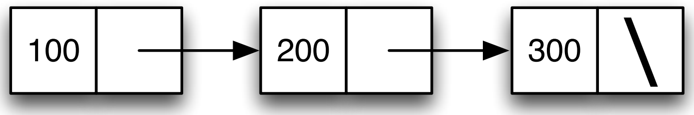
```
/*
Return the number of nodes in this list
*/
public int length() {
    if (next==null) { return 1; }
    else return 1 + next.length();
}
```

a) 1ms
b) 1,000ms
c) 2,000ms
d) 10,000ms
e) 20,000ms

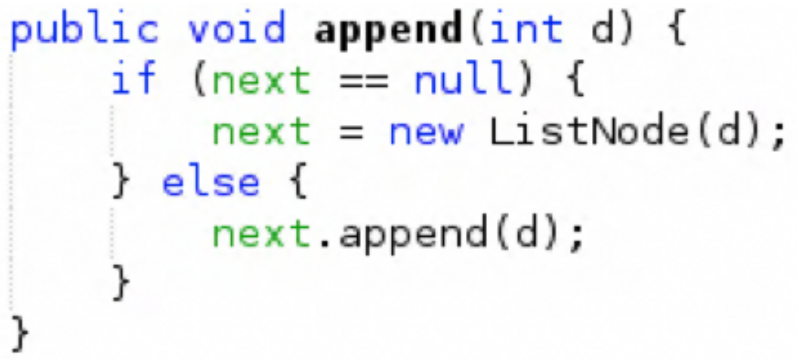
## What to do now

- HW2 out today
- Quiz 2 upcoming
- Pre-lab 2 posted today
- announcement: Debug Your Brain will again be Tu 3pm, due to Labor Day

# Some problems with ListNode

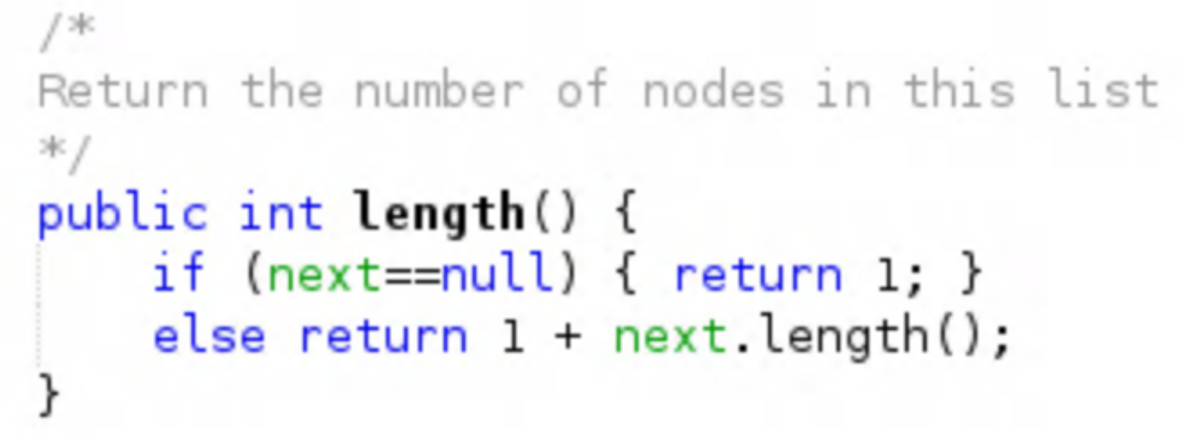


- We have to go through the *whole list* to **append** a new element



```
public void append(int d) {
    if (next == null) {
        next = new ListNode(d);
    } else {
        next.append(d);
    }
}
```

- We have to go through the *whole list* to get the **length**

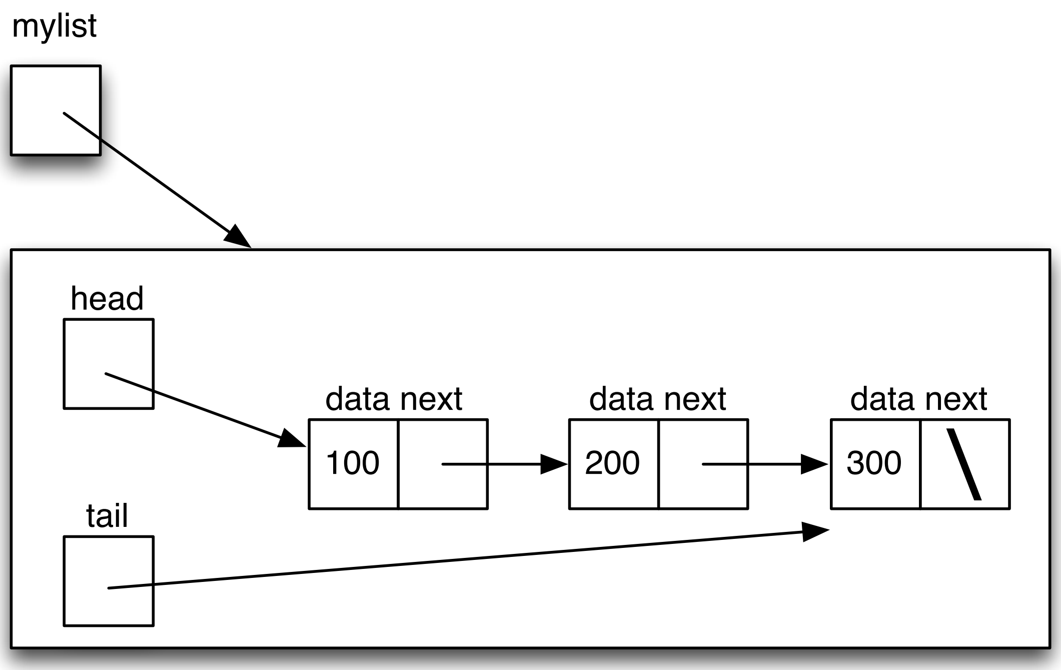


```
/*
Return the number of nodes in this list
*/
public int length() {
    if (next==null) { return 1; }
    else return 1 + next.length();
}
```

# A new class, LinkedList

LinkedList uses the `ListNode` class in its *implementation*

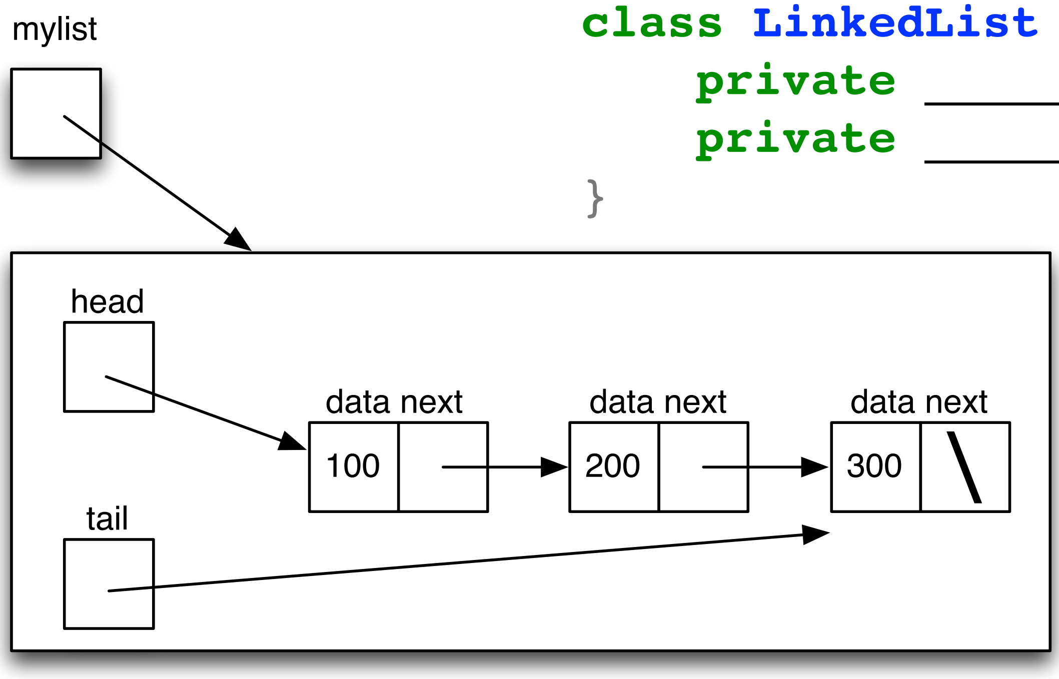
Inside LinkedList, we can privately keep a reference to the front (head) **and** the back (tail)



```
LinkedList mylist = new LinkedList();
mylist.append(100); mylist.append(200); mylist.append(300);
```

**IMPORTANT: the append method on this slide is LinkedList.append *not* ListNode.append!**

## What should be the type for head and tail?

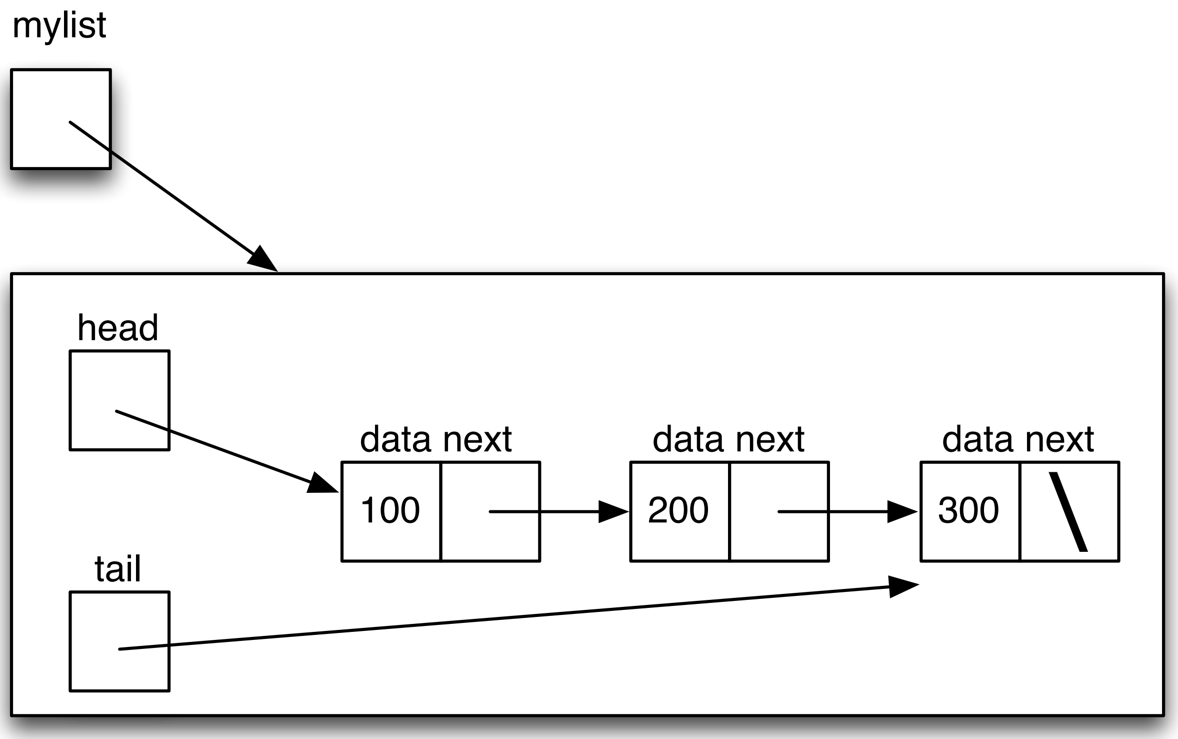


```
class LinkedList {
    private ________ head;
    private ________ tail;
}
```

A) int
B) int[]
C) ListNode
D) ListNode[]
E) LinkedList

https://b.socrative.com/login/student/
CS2230X – 1000-2999
CS2230Y – 3000-5999

# Algorithm for LinkedList's append() ?

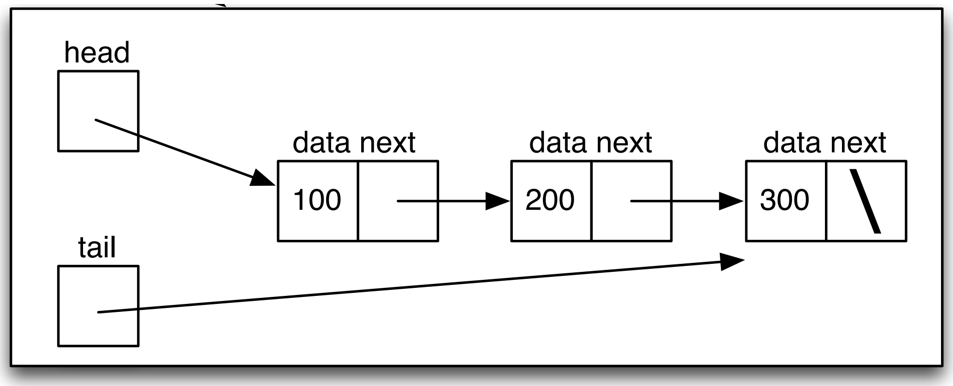


*1 sentence answer*

example usage

```
LinkedList mylist = new LinkedList();
...
mylist.append(400);
```

# Proposed append implementation

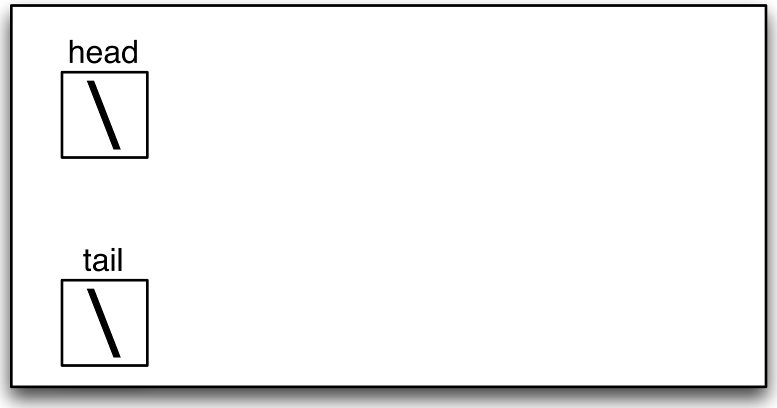


```
public class LinkedList {
     private ListNode head;
     private ListNode tail;



      public void append(int d) {
            ListNode n = new ListNode(d);
            tail.next = n;
            tail = n;
       }
}
```

We have a bug! What is it?
(find a LinkedList for which it fails)

```
public void append(int d) {
        ListNode n = new ListNode(d);
        tail.next = n;
        tail = n;
    }
```

empty list case



UH OH...

```java
public void append(int d) {
  ListNode n = new ListNode(d);
  if (tail == null) {
    // list is empty
    head = n;
    tail = n;
  } else {
    tail.next = n;
    tail = n;
  }
}
```

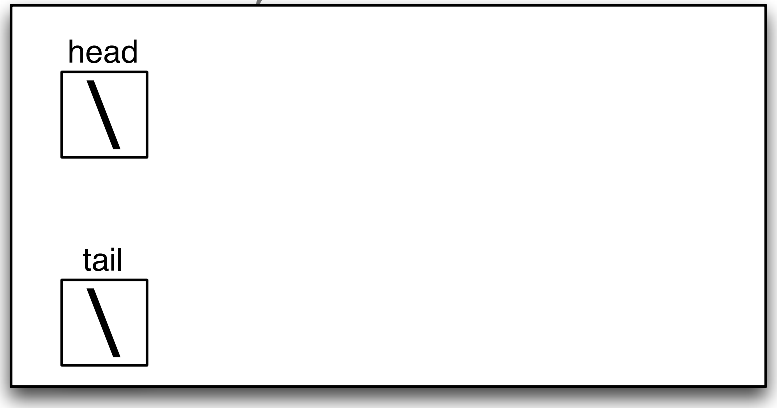
empty list case

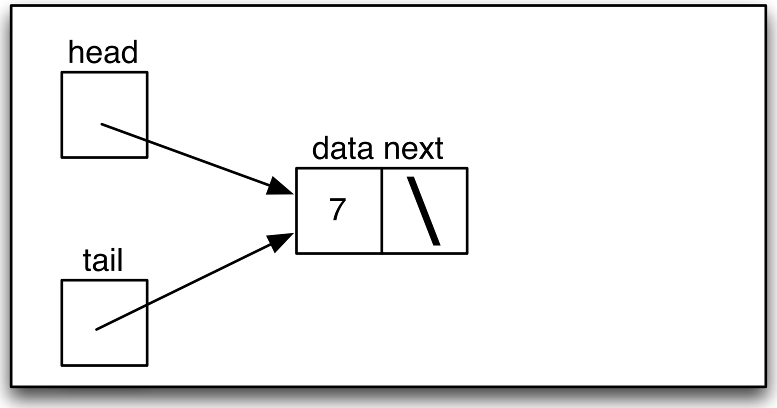
non empty list case

BEFORE

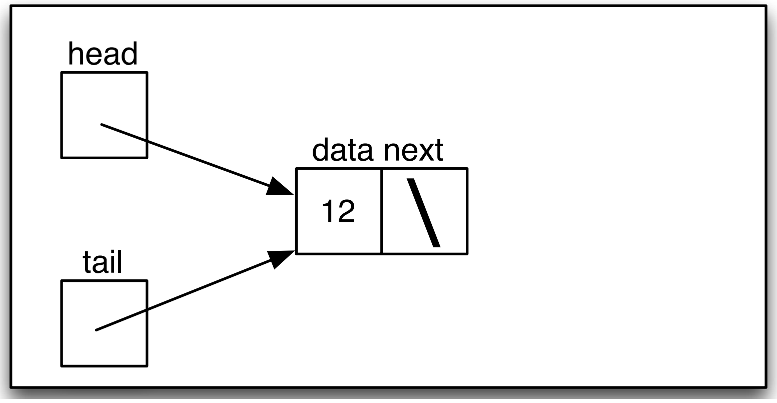
AFTER

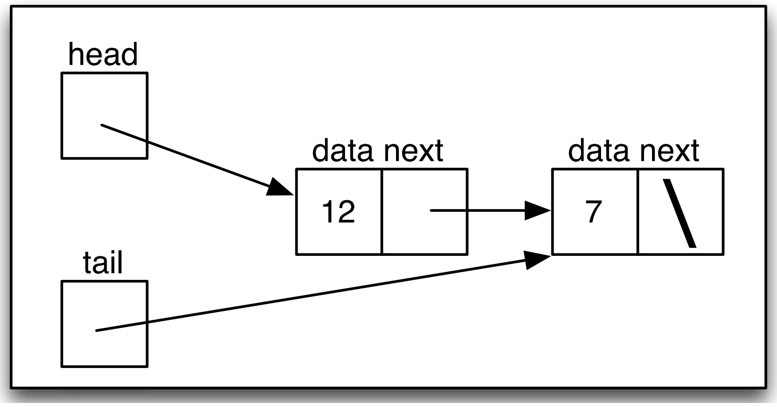
empty list case





non empty list case





# The potential for an ***invariant***!

Wouldn't it be nice if we didn't need a special case for $tail\ == \ null$?

Rephrased version of this question: can we design LinkedList to ensure the following invariant?

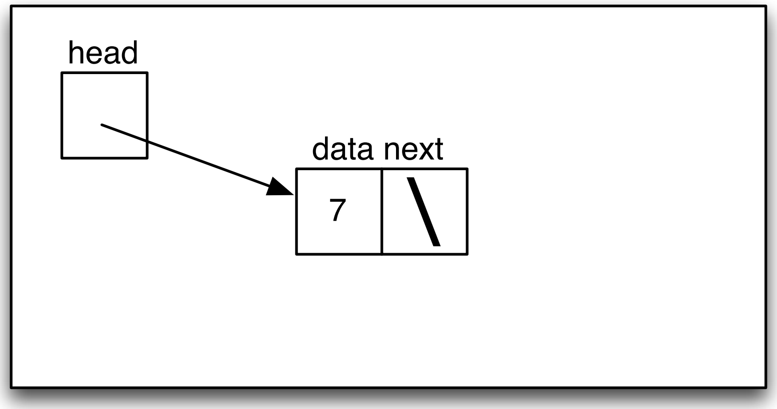
$tail\ != \ null$

```java
public void append(int d) {
  ListNode n = new ListNode(d);
  if (tail == null) {
    // list is empty
    head = n;
    tail = n;
  } else {
    tail.next = n;
    tail = n;
  }
}
```

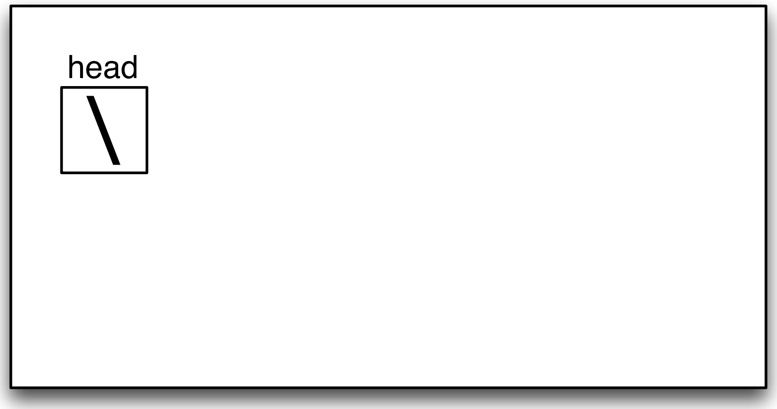
can we eliminate the need for this check?

# Let's consider a similar case with the LinkedList with no tail

LinkedList

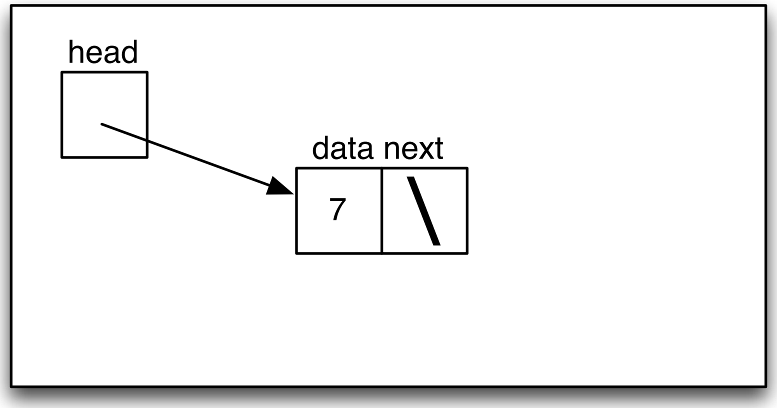


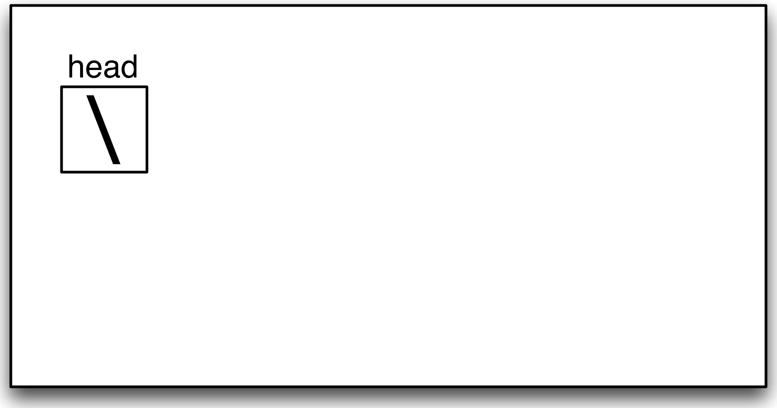
non-empty list



empty list

LinkedList

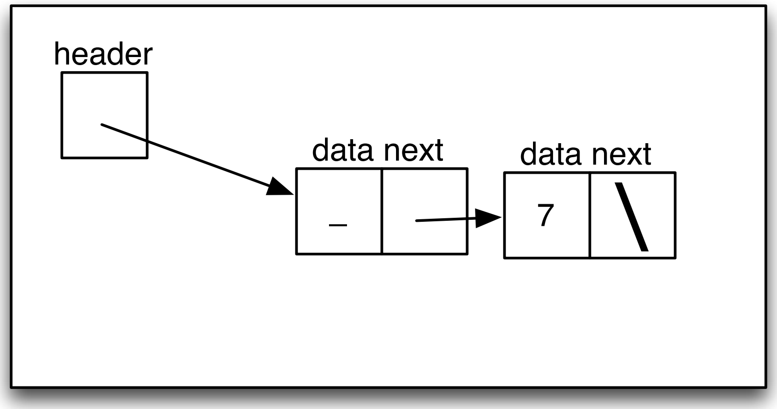


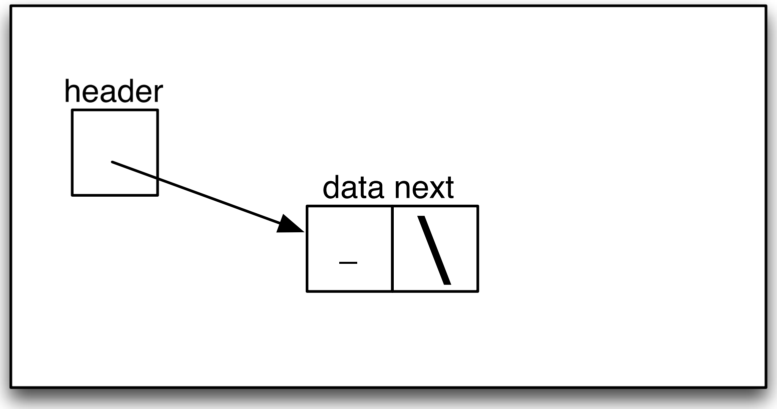


non-empty list

empty list

SLinkedList



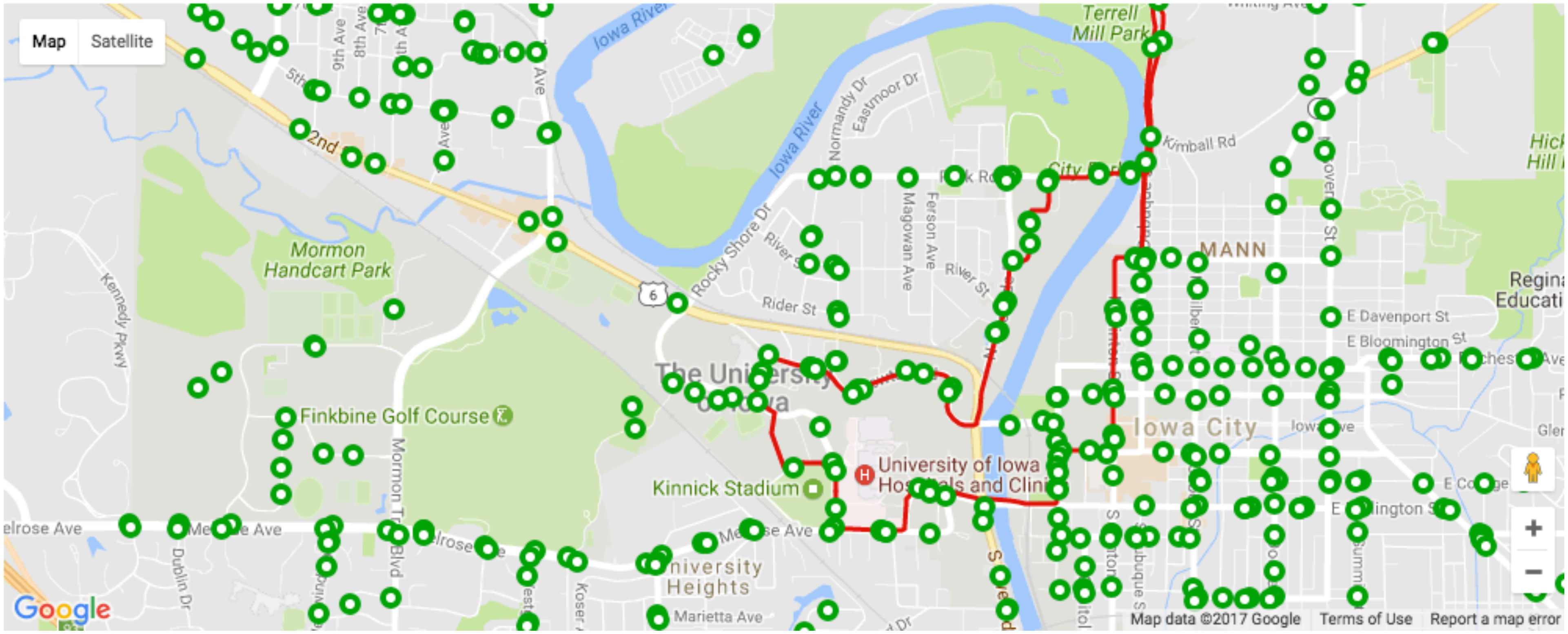


an invariant in SLinkedList: header != null

# Linked lists: other variants!

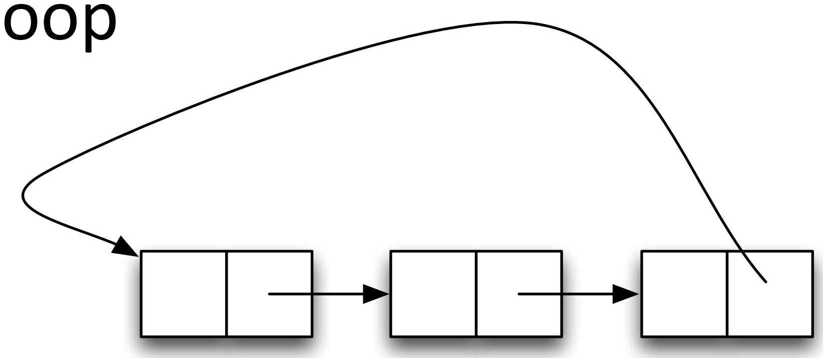
(also in Chapter 3)

# storing Bus stops in a linked list

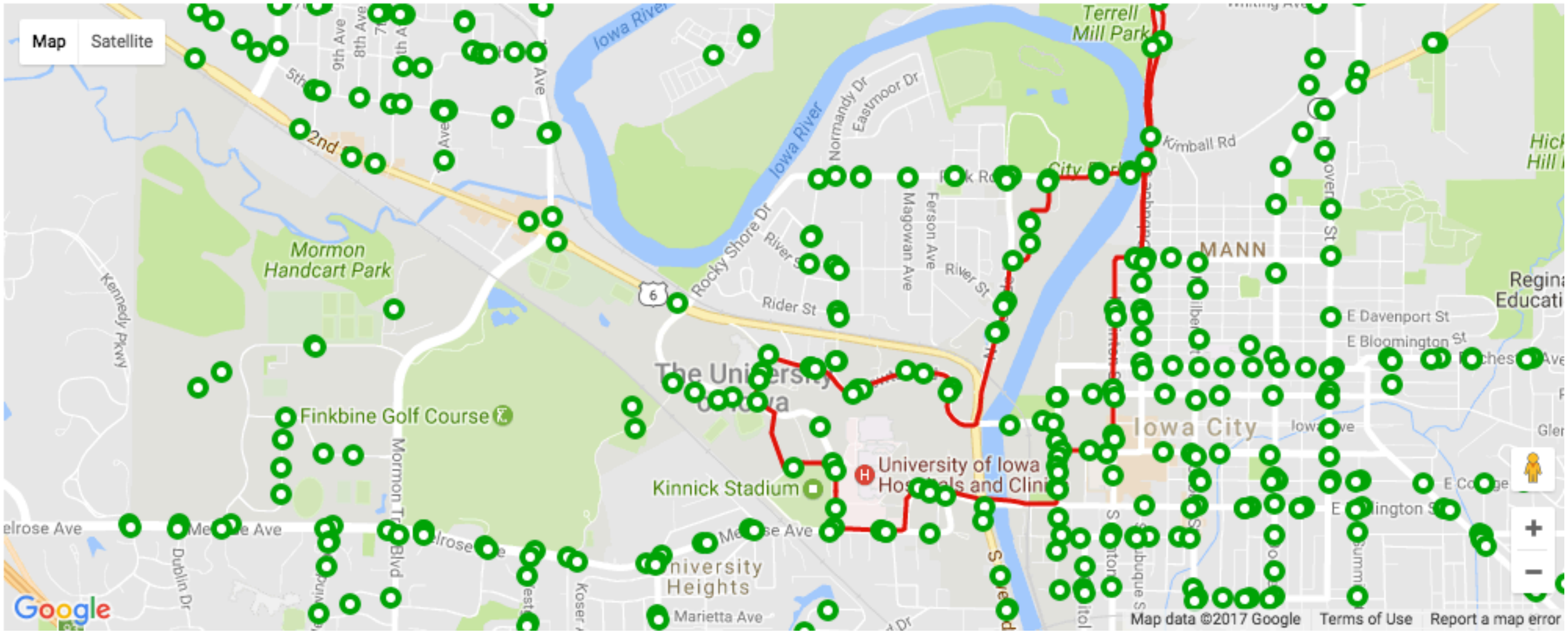


Cambus Red route – goes in a loop

Circularly linked list of bus stops

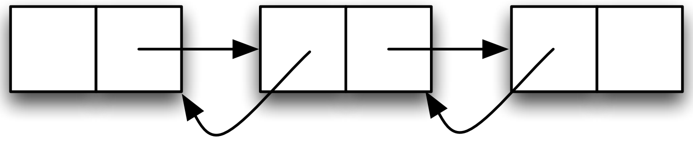


# storing Bus stops in a linked list



Cambus Red route – goes in a loop **either clockwise or counter clockwise**

doubly linked list of bus stops



# Today’s big ideas 2

- write some methods for ListNode using iteration (loops) or recursion
- Don’t use dot ( . ) if your reference could be `null`!
- ***encapsulate*** ListNodes inside of a ***LinkedList*** class so we can try different implementations of a linked list
- LinkedLists be empty, so we have to check for this case. A ***sentinel node*** provides a useful invariant (header!=null) that simplifies code