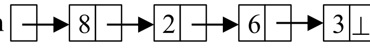
Recursion on linked lists: Reversing a linked list

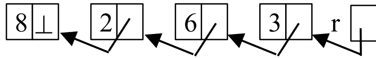
```
/** An instance is a node of a singly linked list*/
class Node {
   int val;     // the value in the node
   Node next; // ptr to next node
                  // (null if no more nodes)
}
```

An instance of class `Node` is a node of a singly linked list. Field `val` contains the value of the node, and field `next` contains either a pointer to the next node or null (represented by ⊥) if there are no more nodes. Class `Node` has methods, but we don't need to know about them for this video.

We show a linked list pointed to be `h`; it represents the list (8, 2, 6, 3).



We develop a recursive function to reverse linked list `h`. Here's its specification —note that only the `next` fields should be changed. As an example, the assignment `r= rev(h.next);` should produce this change in the list, with `r` pointing at the result.




```
/** Reverse list h —by changing only the next fields—
  * and return a pointer to the reversed list. */
public static Node rev(Node h) {


}
```

Base cases

The first step in writing this recursive function is to identify the base cases and write code to take care of them. It is in your best interest to pause the video and perform this step yourself! Please do that!

When you are finished, proceed with the video and see how we perform this step.

©David Gries, 2018

Recursion on linked lists: Reversing a linked list

The two simplest cases are:

1. A list of length 0, indicated by `h` == **null** h[⊥]
2. A list of length 1, indicated by `h.next` == **null** h[ ]→[3|⊥]

In both of these cases, the reverse of linked list h is itself, so there is no work to be done.

We emphatically do *not* consider a list of length 2 as a base case because processing it would cause us to reverse the linked list, and we do *not* want extra work!

The two base cases are easily implemented. The reverse of a list with one or two elements is the same list, so `h` is returned.

```
/** Reverse list h —by changing only the next fields—
  * and return a pointer to the reversed list. */
public static Node rev(Node h) {
   if (h == null || h.next == null) return h;
   // h [ ]→[8| ]→[2| ]→[6| ]→[3|⊥]

   …
}
```

We now introduce a recursive call

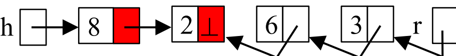
After the code for the base case, we place a comment that shows an example of linked list `h`. We handle the recursive case in three steps.

(1) Write a recursive call on function `rev`. Its argument must be a shorter linked list than parameter `h`, so that progress toward termination is made.
(2) Since `rev` is a function, assign the value of the call to a new local variable, call it, say, `r`.
(3) This next step is important! Look at the *specification* of `rev`. Based on it, write a comment after the assignment statement showing the linked list and variables `h` and `r` after the assignment statement. You see how careful we were in drawing the comment? Be just as careful as precise as we were.

Pause the video and carry out the three steps shown above yourelf, carefully. Proceed with the video after you are done to see how we do these three steps.

©David Gries, 2018

According to the specification, the recursive call `rev (h.next)` reverses all but the first node of the list, and we assign its value to a new local variable `r`. Here is what the variables and linked list are after the call:



Variable `r` points at the first node of the reversed list, which points at the second node of the reversed list, which points at the third node, which has **null** in its next field.

Variable `h` still points at the first node of the original list, and the first node still points at the second node of the original list.

To complete the reversal of linked list `h`, the two red next fields have to be changed. Let's copy the comment.

First, change the next field of the second node to point to the first node, `h`.

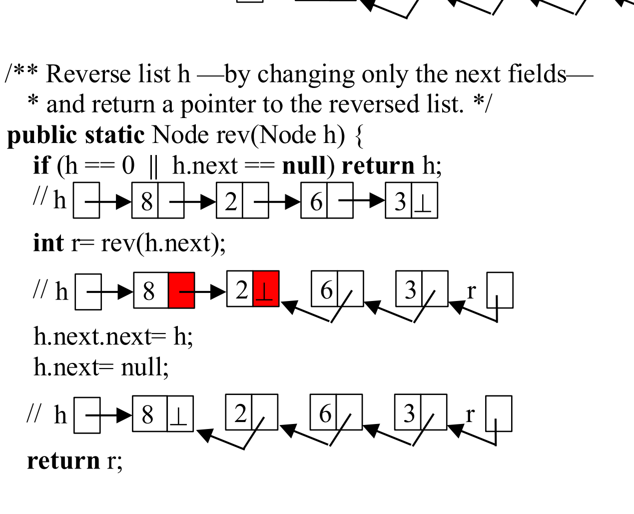
```
h.next.next= h;
```

Second, since node `h` is now the last node of the reversed list, change its next field to **null**.

```
h.next= null;
```

Now the whole list has been reversed, and `r` can be returned:

h 8 ⊥ 2 6 3 r



Discussion

What helped us *develop* this function? First, we wrote an example of a linked list as a precondition of the assignment of the recursive call to variable `r`. Second, we wrote a corresponding postcondition of that assignment statement that also serves as a precondition to the following lines. The postcondition was based on the *specification* of the function. Finally, we wrote a postcondition of the two statements that completed reversing the list. This made ithe development *easy*!

©David Gries, 2018