Interview Cake

Hooray! It's opposite day. Linked lists go the opposite way today.

Write a function for reversing a linked list. Do it in-place.

Your function will have one input: the head of the list.

Your function should return the new head of the list.

Here's a sample linked list node class:

```javascript
function LinkedListNode(value) {
    this.value = value;
    this.next = null;
}
```

Gotchas

We can do this in $O(1)$ space. So don't make a new list; use the existing list nodes!

We can do this is in $O(n)$ time.

Careful—even the right *approach* will fail if done in the wrong *order*.

Try drawing a picture of a small linked list and running your function by hand. Does it actually work?

The most obvious edge cases are:

1. the list has 0 elements
2. the list has 1 element

Does your function correctly handle those cases?

Breakdown

Our first thought might be to build our reversed list "from the beginning," starting with the head of the final *reversed* linked list.

The head of the reversed list will be the *tail* of the input list. To get to that node we'll have to walk through the whole list once ($O(n)$ time). And that's just to get started.

That seems inefficient. **Can we reverse the list while making just one walk from head to tail of the input list?**

We can reverse the list by changing the `next` pointer of each node. Where should each node's `next` pointer...point?

Each node's `next` pointer should point to the *previous* node.

How can we move each node's `next` pointer to its *previous* node in one pass from head to tail of our current list?

Solution

In one pass from head to tail of our input list, we point each node's `next` pointer to the previous item.

The order of operations is important here! We're careful to copy `current.next` into `next` *before* setting `current.next` to `previous`. Otherwise "stepping forward" at the end could actually mean stepping *back* to `previous`!

```javascript
function reverse(headOfList) {
    var current  = headOfList;
    var previous = null;
    var nextNode = null;

    // until we have 'fallen off' the end of the list
    while (current) {

        // copy a pointer to the next element
        // before we overwrite current.next
        nextNode = current.next;

        // reverse the 'next' pointer
        current.next = previous;

        // step forward in the list
        previous = current;
        current = nextNode;
    }

    return previous;
}
```

We return `previous` because when we exit the list, `current` is `null`. Which means that the last node we visited—`previous`—was the tail of the *original* list, and thus the head of our *reversed* list.

Complexity

$O(n)$ time and $O(1)$ space. We pass over the list only once, and maintain a constant number of variables in memory.

Bonus

This in-place reversal destroys the input linked list. What if we wanted to keep a copy of the original linked list? Write a function for reversing a linked list out-of-place.

What We Learned

It's one of those problems where, even once you know the procedure, it's hard to write a bug-free solution. Drawing it out helps a lot. Write out a sample linked list and walk through your code by hand, step by step, running each operation on your sample input to see if the final output is what you expect. This is a great strategy for *any* coding interview question.
