Interview Cake

You have a linked list and want to find the kth to last node.

Write a function `kthToLastNode()` that takes an integer k and the `headNode` of a singly-linked list, and returns the kth to last node in the list.

For example:

Java

```java
public static class LinkedListNode {

    public String value;
    public LinkedListNode next;

    public LinkedListNode(String value) {
        this.value = value;
    }
}

LinkedListNode a = new LinkedListNode("Angel Food");
LinkedListNode b = new LinkedListNode("Bundt");
LinkedListNode c = new LinkedListNode("Cheese");
LinkedListNode d = new LinkedListNode("Devil's Food");
LinkedListNode e = new LinkedListNode("Eccles");

a.next = b;
b.next = c;
c.next = d;
d.next = e;

kthToLastNode(2, a);
// returns the node with value "Devil's Food" (the 2nd to last node)
```

Gotchas

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

We can do this in $O(1)$ space. If you're recursing, you're probably taking $O(n)$ space on the call stack!

Breakdown

Solution

Upgrade now ➜ (/upgrade)

Complexity

Both approaches use $O(n)$ time and $O(1)$ space.

But the second approach is fewer steps, since it gets the answer "in one pass," right? *Wrong.*

In the first approach, we walk one pointer from head to tail (to get the list's length), then walk another pointer from the head node to the target node (the kth to last node).

In the second approach, `rightNode` *also* walks all the way from head to tail, and `leftNode` *also* walks from the head to the target node.

So in both cases we have two pointers taking the same steps through our list. The only difference is the *order* in which the steps are taken. The number of steps is the same either way.

However, the second approach *might* still be slightly *faster*, due to some caching and other optimizations that modern processors and memory have.

Let's focus on caching. Usually when we grab some data from memory (for example, info about a linked list node), we also store that data in a small cache right on the processor. If we need to use that same data again soon after, we can quickly grab it from the cache. But if we don't use that data for a while, we're likely to replace it with other stuff we've used more recently (this is called a "least recently used" replacement policy).

Both of our algorithms access a lot of nodes in our list twice, so they *could* exploit this caching. But notice that in our second algorithm there's a much shorter time between the first and second times that we access a given node (this is sometimes called "temporal locality of reference"). Thus it seems more likely that our second algorithm will save time by using the processor's cache! But this assumes our processor's cache uses something like a "least recently used" replacement policy —it might use something else. Ultimately the *best* way to really know which algorithm is faster is to implement both and time them on a few different inputs!

Bonus

Can we do better? What if we expect n to be huge and k to be pretty small? In this case our target node will be close to the end of the list...so it seems a waste that we have to walk all the way from the beginning *twice*.

Can we trim down the number of steps in the "second trip"? One pointer will certainly have to travel all the way from head to tail in the list to get the total length...but can we store some "checkpoints" as we go so that the second pointer doesn't have to start all the way at the beginning? Can we store these "checkpoints" in constant space? Note: this approach only saves time if we know that our target node is towards the *end* of the list (in other words, n is much larger than k).

What We Learned