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

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

For example:

```ruby
class LinkedListNode

    attr_accessor :value, :next

    def initialize(value)
        @value = value
        @next  = nil
    end
end

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

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

kth_to_last_node(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!

> The **call stack** is what a program uses to keep track of what function it's currently running and what to do with that function's return value.

Whenever you call a function, a new **frame** gets pushed onto the call stack, which is popped off when the function returns. As functions call other functions, the stack gets taller. In recursive functions, the stack can get as tall as the number of times the function calls itself. This can cause a problem: the stack has a limited amount of space, and if it gets too big you can get a **stack overflow** error.

Breakdown

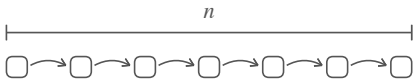
It might be tempting to iterate through the list until we reach the end and then walk backwards k nodes.

But we have a *singly* linked list! We can't go backwards. What else can we do?

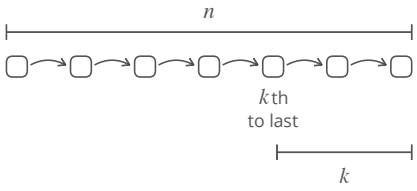
What if we had the *length* of the list?

Then we could calculate how far to walk, starting from the head, to reach the kth to last node.

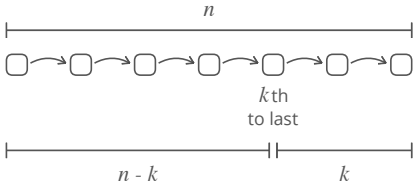
If the list has n nodes:



And our target is the kth to last node:



The distance from the head to the target is $n - k$:



Well, we *don't* know the length of the list (n). But can we *figure it out*?

Yes, we could iterate from the head to the tail and count the nodes!

Can you implement this approach in code?

```ruby
def kth_to_last_node(k, head)

    # STEP 1: get the length of the list
    # start at 1, not 0
    # else we'd fail to count the head node!
    list_length = 1
    current_node = head

    # traverse the whole list,
    # counting all the nodes
    while current_node.next
        current_node = current_node.next
        list_length += 1
    end

    # STEP 2: walk to the target node
    # calculate how far to go, from the head,
    # to get to the kth to last node
    how_far_to_go = list_length - k

    current_node = head
    (0...how_far_to_go).each do |i|
        current_node = current_node.next
    end

    return current_node
end
```

What are our time and space costs?

$O(n)$ time and $O(1)$ space, where n is the length of the list.

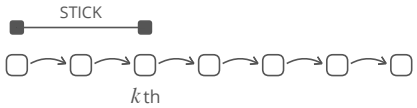
More precisely, it takes n steps to get the length of the list, and another $n - k$ steps to reach the target node. In the worst case $k = 1$, so we have to walk all the way from head to tail *again* to reach the target node. This is a total of $2n$ steps, which is $O(n)$.

Can we do better?

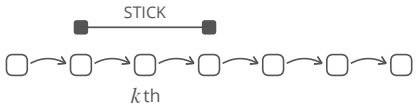
Mmmmaaaaaaybe.

The fact that we walk through our whole list once just to get the length, then walk through the list *again* to get to the kth to last element sounds like a lot of work. Perhaps we can do this in just one pass?

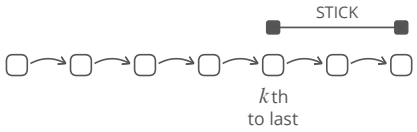
What if we had like a "stick" that was k nodes wide. We could start it at the beginning of the list, so that the left side of the stick was on the head and the right side was on the kth node.



Then we could slide the stick down the list...



until the right side hit the end. At that point, the left side of the stick would be on the *k*th to last node!



How can we implement this? Maybe it'll help to keep *two* pointers?

We can allocate two variables that'll be references to the nodes at the left and right sides of the "stick"!

```ruby
def kth_to_last_node(k, head)

    left_node  = head
    right_node = head

    # move right_node to the kth node
    (0...k - 1).each do |x|
        right_node = right_node.next
    end

    # starting with left_node on the head,
    # move left_node and right_node down the list,
    # maintaining a distance of k between them,
    # until right_node hits the end of the list
    while right_node.next
        left_node  = left_node.next
        right_node = right_node.next
    end

    # since left_node is k nodes behind right_node,
    # left_node is now the kth to last node!
    return left_node
end
```

This'll work, but **does it actually save us any time**?

Solution

We can think of this two ways.

First approach: use the length of the list.

1. walk down the whole list, counting nodes, to get the total `list_length`.
2. subtract k from the `list_length` to get the distance from the head node to the target node (the kth to last node).
3. walk that distance from the head to arrive at the target node.

```ruby
def kth_to_last_node(k, head)

    if k < 1
        raise ArgumentError, "Impossible to find less than first to last node: #{k}"
    end

    # STEP 1: get the length of the list
    # start at 1, not 0
    # else we'd fail to count the head node!
    list_length = 1
    current_node = head

    # traverse the whole list,
    # counting all the nodes
    while current_node.next
        current_node = current_node.next
        list_length += 1
    end

    # if k is greater than the length of the list, there can't
    # be a kth-to-last node, so we'll return an error!
    if k > list_length
        raise ArgumentError, "k is larger than the length of the linked list: #{k}"
    end

    # STEP 2: walk to the target node
    # calculate how far to go, from the head,
    # to get to the kth to last node
    how_far_to_go = list_length - k

    current_node = head
    (0...how_far_to_go).each do |i|
        current_node = current_node.next
    end

    return current_node
end
```

Second approach: maintain a k-wide "stick" in one walk down the list.

1. Walk one pointer k nodes from the head. Call it `right_node`.
2. Put another pointer at the head. Call it `left_node`.
3. Walk both pointers, at the same speed, towards the tail. This keeps a distance of k between them.
4. When `right_node` hits the tail, `left_node` is on the target (since it's k nodes from the end of the list).

Ruby

```ruby
def kth_to_last_node(k, head)

    if k < 1
        raise ArgumentError, "Impossible to find less than first to last node: #{k}"
    end

    left_node  = head
    right_node = head

    # move right_node to the kth node
    (0...k - 1).each do |x|

        # but along the way, if a right_node doesn't have a next,
        # then k is greater than the length of the list and there
        # can't be a kth-to-last node! we'll raise an error
        if !right_node.next
            raise ArgumentError, "k is larger than the length of the linked list: #{k}"
        end

        right_node = right_node.next
    end

    # starting with left_node on the head,
    # move left_node and right_node down the list,
    # maintaining a distance of k between them,
    # until right_node hits the end of the list
    while right_node.next
        left_node  = left_node.next
        right_node = right_node.next
    end

    # since left_node is k nodes behind right_node,
    # left_node is now the kth to last node!
    return left_node
end
```

In either approach, make sure to check if `k` is greater than the length of the linked list! That's bad input, and we'll want to raise an error.

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, `right_node` *also* walks all the way from head to tail, and `left_node` *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

We listed two good solutions. One seemed to solve the problem in one pass, while the other took two passes. But the single-pass approach didn't take half as many steps, it just took the *same* steps in a different *order*.

So don't be fooled: "one pass" isn't always fewer steps than "two passes." Always ask yourself, "Have I actually changed the *number* of steps?"
