Delete a node from a singly-linked list, given *only* a *variable pointing to that node.*

The input could, for example, be the variable `b` below:

```ruby
class LinkedListNode

    attr_accessor :value, :next

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

a = LinkedListNode.new('A')
b = LinkedListNode.new('B')
c = LinkedListNode.new('C')

a.next = b
b.next = c

delete_node(b)
```

Gotchas

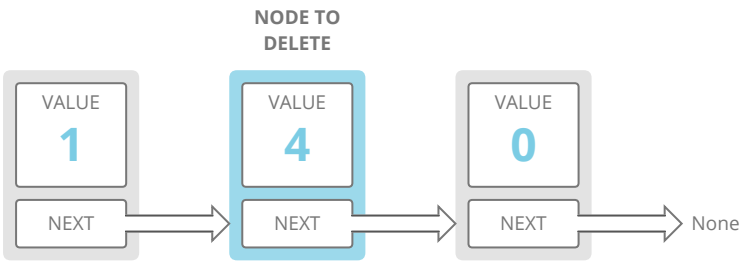
We can do this in $O(1)$ time and space! But our answer is tricky, and it *could* have some side effects...

Breakdown

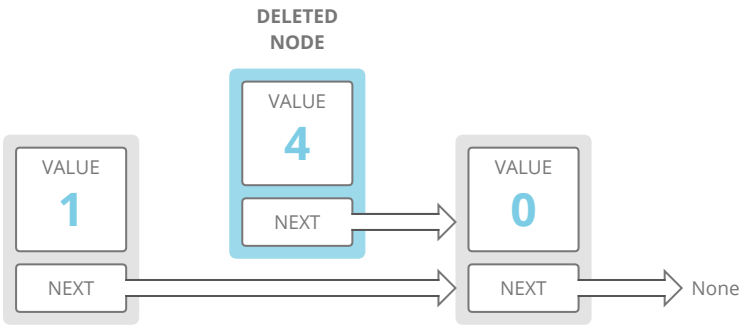
It might be tempting to try to traverse the list from the beginning until we encounter the node we want to delete. But in this situation, we don't know where the head of the list is—we *only* have a reference to the node we want to delete.

But hold on—how do we even delete a node from a linked list in general, when we *do* have a reference to the first node?

We'd change the *previous* node's pointer to *skip* the node we want to delete, so it just points straight to the node *after* it. So if these were our nodes **before** deleting a node:



These would be our nodes **after** our deletion:



So we need a way to skip over the `current` node and go straight to the next node. But we don't even have *access* to the previous node!

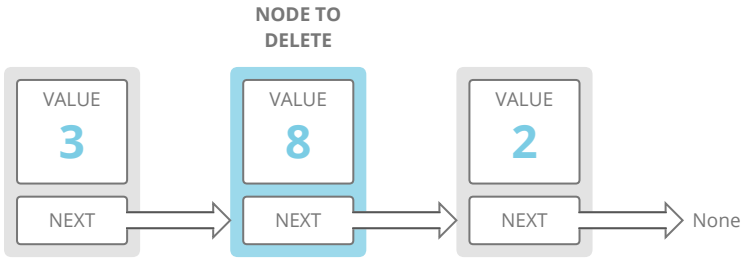
Other than rerouting the previous node's pointer, **is there *another way* to skip from the previous pointer's *value* to the next pointer's *value*?**

What if we *modify* the current node instead of deleting it?

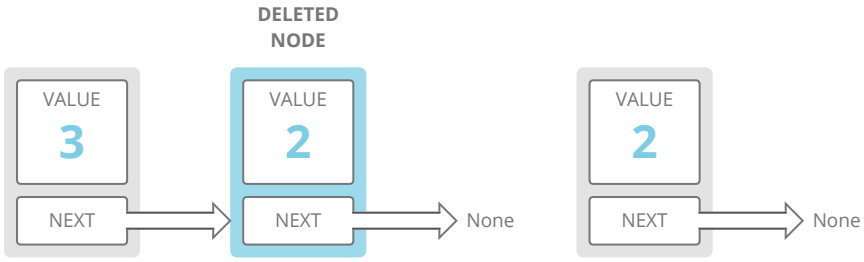
Solution

We take `@value` and `@next` from *the input node's* ***next*** *node* and copy them *into the* ***input node***. Now the input node's *previous* node effectively skips the input node's old value!

So for example, if this was our linked list **before** we called our function:



This would be our list **after** we called our function:



In some languages, like C, we'd have to manually delete the node we copied from, since we won't be using that node anymore. Here, we'll let Ruby's garbage collector take care of it.

```ruby
def delete_node(node_to_delete)

    # get the input node's next node, the one we want to skip to
    next_node = node_to_delete.next

    if next_node

        # replace the input node's value and pointer with the next
        # node's value and pointer. the previous node now effectively
        # skips over the input node
        node_to_delete.value = next_node.value
        node_to_delete.next  = next_node.next

    else

        # eep, we're trying to delete the last node!
        raise "Can't delete the last node with this method!"
    end
end
```

But be careful—there are some potential problems with this implementation:

First, it doesn't work for deleting the *last* node in the list. We *could* change the node we're deleting to have a value of `nil`, but the second-to-last node's `@next` pointer would still point to a node, even though it should be `nil`. This *could* work—we could treat this last, "deleted" node with value `nil` as a "dead node" or a "sentinel node," and adjust any node *traversing* code to stop traversing when it hits such a node. The trade-off there is we couldn't have non-dead nodes with values set to `nil`. Instead we chose to throw an exception in this case.

Second, this method can cause some unexpected side-effects. For example, let's say we call:

```ruby
a = LinkedListNode.new(3)
b = LinkedListNode.new(8)
c = LinkedListNode.new(2)

a.next = b
b.next = c

delete_node(b)
```

There are two potential side-effects:

1. **Any references to the input node have now effectively been reassigned to its `@next` node.** In our example, we "deleted" the node assigned to the variable `b`, but in actuality we just gave it a new value (2) and a new `@next`! If we had another pointer to `b` *somewhere else* in our code and we were assuming it still had its old value (8), that could cause bugs.
2. **If there are pointers to the input node's *original next node*, those pointers now point to a "dangling" node** (a node that's no longer reachable by walking down our list). In our example above, `c` is now dangling. If we changed `c`, we'd never encounter that new value by walking down our list from the head to the tail.

Complexity

$O(1)$ time and $O(1)$ space.

What We Learned

My favorite part of this problem is how imperfect the solution is. Because it modifies the list "in place" it can cause other parts of the surrounding system to break. This is called a "side effect."

In-place operations like this can save time and/or space, but they're risky. If you ever make in-place modifications in an interview, make sure you tell your interviewer that in a real system you'd carefully check for side effects in the rest of the code base.
