Delete a node from a singly-linked list,↴

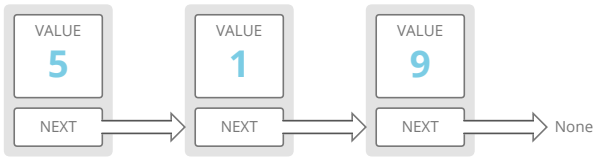
A **linked list** is a low-level data structure. It stores an *ordered* sequence of items in individual "node" objects that have pointers to other nodes.

In a **singly-linked list**, the nodes each have one pointer to the next node.

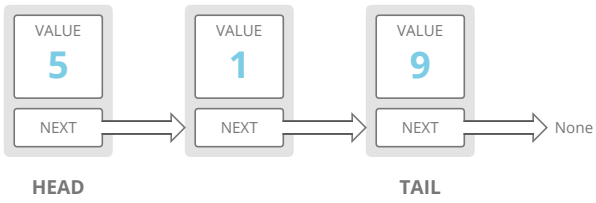
```ruby
class LinkedListNode

    attr_accessor :value, :next

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

So we could build a singly-linked list like this:

```ruby
a = LinkedListNode.new(5)
b = LinkedListNode.new(1)
c = LinkedListNode.new(9)

a.next = b
b.next = c
```



In a linked list, the first node is called the **head** and the last node is called the **tail**.



Often, our only connection to *the list itself* is a variable pointing to the head. From there we can walk down the list to all the other nodes.

Like linked lists, **arrays** also store ordered lists of items, so you usually have a choice of which one to use.

Advantages of linked lists:

1. Linked lists have **constant-time insertions and deletions** in any position (you just change some pointers). Arrays require $O(n)$ time to do the same thing, because you'd have to "shift" all the subsequent items over 1 index.
2. Linked lists can **continue to expand** as long as there is space on the machine. Arrays (in low-level languages) must have their size specified ahead of time. Even in languages with "dynamic arrays" that automatically resize themselves when they run out of space (like Python, Ruby and JavaScript), the operation to resize a dynamic array has a large cost which can make a single insertion unexpectedly expensive.

Disadvantages of linked lists:

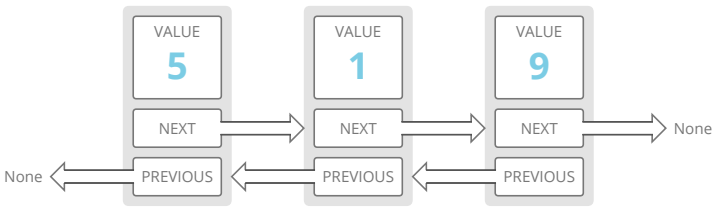
1. To access or edit an item in a linked list, you have to **take $O(i)$ time to walk from the head of the list to the ith item** (unless of course you already have a pointer directly to that item). Arrays have *constant-time* lookups and edits to the ith item.

Another type of linked list is a **doubly linked list**, which has pointers to the next *and the previous* nodes.

```ruby
class LinkedListNode

    attr_accessor :value, :next, :previous

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

So we could build a doubly linked list like this:

```ruby
a = LinkedListNode.new(5)
b = LinkedListNode.new(1)
c = LinkedListNode.new(9)

# put b after a
a.next = b
b.previous = a

# put c after b
b.next = c
c.previous = b
```



Doubly linked lists allow us to traverse our list *backwards*. In a *singly* linked list, if you just had a pointer to a node in the *middle* of a list, there would be *no way* to know what its previous node was. Not a problem in a doubly 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

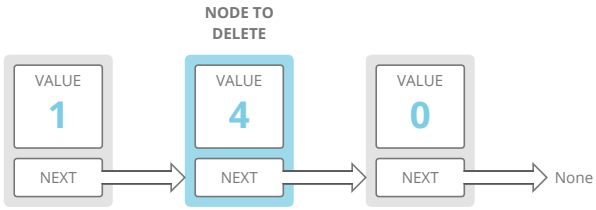
We can do this in $O(1)$ time and space! But our answer is tricky, and it *could* have some side effects...

Breakdown

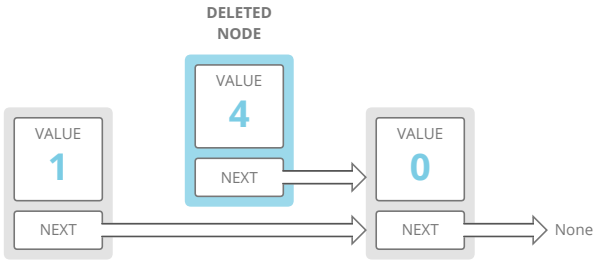
It might be tempting to try to traverse the list from the beginning until we encounter the node we want to delete. But in this situation, we don't know where the head of the list is—we *only* have a reference to the node we want to delete.

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!

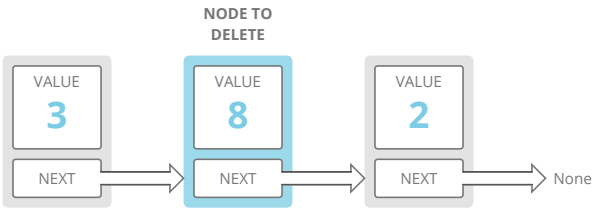
Other than rerouting the previous node's pointer, **is there *another way* to skip from the previous pointer's *value* to the next pointer's *value*?**

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

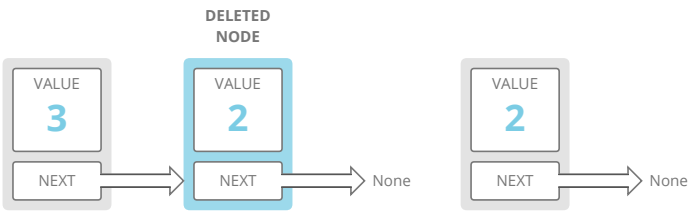
Solution

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

A **garbage collector** automatically frees up memory that a program isn't using anymore.

For example, say we did this in Ruby:

```ruby
def get_min(nums)
    # NOTE: this is NOT the fastest way to get the min!

    nums_sorted = nums.sort
    return nums_sorted[0]
end

my_nums = [5, 3, 1, 4, 6]
puts get_min(my_nums)
```

Look at `nums_sorted` in `get_min()`. We allocate that whole array inside our function, and once the function returns we don't need the array anymore. In fact, once the function returns we *don't have any references to it anymore*!

What happens to that array in memory? The Ruby garbage collector will notice we don't need it anymore and free up that space.

> How does the garbage collector know when something can be freed?
>
> One option is to start by figuring out what we *can't* free. For example, we definitely can't free local variables that we're going to need later on. And, if we have an array, then we also shouldn't free any of the array's elements.
>
> This is the main intuition behind one garbage collector strategy:
>
> 1. Carefully figure out what things in memory we might still be using or need later on.
> 2. Free everything else.
>
> This strategy is often called **tracing garbage collection**, since we usually implement the first step by tracing references from one object (say, the array) to the next (an element within the array).
>
> A different option is to have each object keep track of the number of things that reference it—like a variable holding the location of an array or multiple edges pointing to the same node in a graph. We'll call this number an object's **reference count**.
>
> In this case, a garbage collector can free anything with a reference count of zero.
> This strategy is called **reference counting**, since we are *counting* the number of times each object is *referenced*.

Some languages, like C, don't have a garbage collector. So we need to manually free up any memory we're not using anymore:

```c
// make a string that can hold 15 characters
// including the terminating null byte ('\0')
str = malloc(15);

// ... do some stuff with it ...

// we're done. free that memory!
free(str);
```

We sometimes call this **manual memory management**.

Some languages, like C++, have both manual and automatic memory management.

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.
