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

Write a function for reversing a 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.

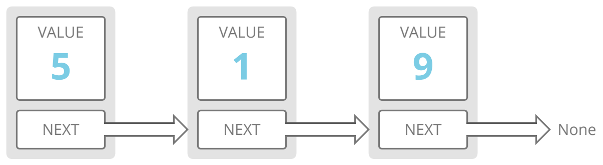
Ruby

```ruby
class LinkedListNode

    attr_accessor :value, :next

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

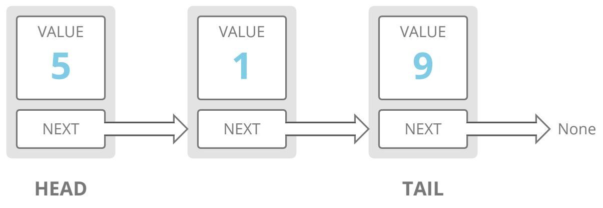
So we could build a singly-linked list like this:

Ruby

```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.

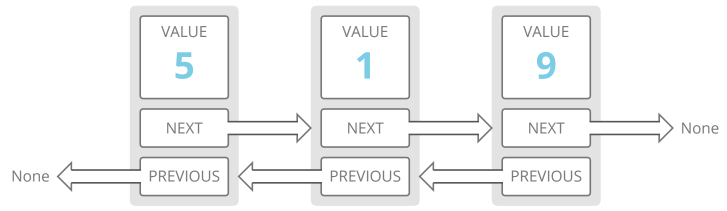
Ruby ▾

```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 ▾

```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.

Do it in-place.

An **in-place** algorithm operates *directly* on its input and *changes* it, instead of creating and returning a *new* object. This is sometimes called **destructive**, since the original input is "destroyed" when it's edited to create the new output.

Careful: "In-place" does *not* mean "without creating any additional variables!" Rather, it means "without creating a new copy of the input." In general, an in-place function will only create additional variables that are $O(1)$ space.

Here are two functions that do the same operation, except one is in-place and the other is out-of-place:

Ruby

```
def square_array_in_place(int_array)

    int_array.each_with_index do |element, index|
        int_array[index] *= element
    end

    # NOTE: we could make this function return nil,
    # since we modify int_array in place.
    return int_array
end

def square_array_out_of_place(int_array)

    # we allocate a new array with the length of the input array
    squared_array = [nil] * int_array.length

    int_array.each_with_index do |element, index|
        squared_array[index] = element ** 2
    end

    return squared_array
end
```

Working in-place is a good way to save space. An in-place algorithm will generally have $O(1)$ space cost.

But be careful: an in-place algorithm can cause side effects. Your input is "destroyed" or "altered," which can affect code *outside* of your function. For example:

```ruby
original_array = [2, 3, 4, 5]
squared_array  = square_array_in_place(original_array)

puts "squared: #{squared_array}"
# prints: squared: [4, 9, 16, 25]

puts "original array: #{original_array}"
# prints: original array: [4, 9, 16, 25], confusingly!

# and if square_array_in_place() didn't return anything,
# which it could reasonably do, squared_array would be nil!
```

Generally, out-of-place algorithms are considered safer because they avoid side effects. You should only use an in-place algorithm if you're very space constrained or you're *positive* you don't need the original input anymore, even for debugging.

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:

```ruby
class LinkedListNode

    attr_accessor :value, :next

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

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`!

Ruby ▾

```ruby
def reverse(head_of_list)
    current = head_of_list
    previous = nil
    next_node = nil

    # until we have 'fallen off' the end of the list
    while current

        # copy a pointer to the next element
        # before we overwrite current.next
        next_node = current.next

        # reverse the 'next' pointer
        current.next = previous

        # step forward in the list
        previous = current
        current = next_node
    end

    return previous
end
```

We return `previous` because when we exit the list, `current` is `nil`. 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.