Interview Cake

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

Write a function for reversing a linked list.

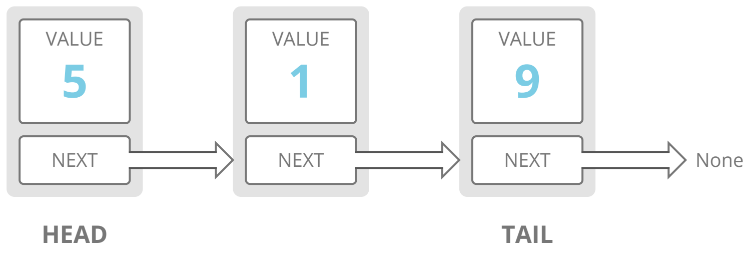
A **linked list** organizes items sequentially, with each item storing a pointer to the next one.

Picture a linked list like a chain of paperclips linked together. It's quick to add another paperclip to the top or bottom. It's even quick to insert one in the middle—just disconnect the chain at the middle link, add the new paperclip, then reconnect the other half.

An item in a linked list is called a **node**. The first node is called the **head**. The last node is called the **tail**.

	Worst Case
space	$O(n)$
prepend	$O(1)$
append	$O(1)$
lookup	$O(n)$
insert	$O(n)$
delete	$O(n)$

> Confusingly, *sometimes* people use the word **tail** to refer to "the whole rest of the list after the head."



> Unlike an array, consecutive items in a linked list are not necessarily next to each other in memory.

Strengths:

- **Fast operations on the ends**. Adding elements at either end of a linked list is $O(1)$. Removing the first element is also $O(1)$.

- **Flexible size**. There's no need to specify how many elements you're going to store ahead of time. You can keep adding elements as long as there's enough space on the machine.

Weaknesses:

- **Costly lookups**. 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.

Uses:

- **Stacks (/concept/stack) and queues (/concept/queue)** only need fast operations on the ends, so linked lists are ideal.

In Python 3.6

Most languages (including Python 3.6) don't provide a linked list implementation. Assuming we've already implemented our own, here's how we'd construct the linked list above:

Python 3.6 ▼

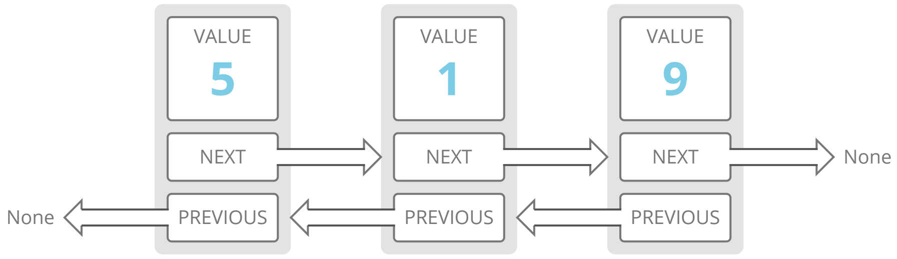
```
a = LinkedListNode(5)
b = LinkedListNode(1)
c = LinkedListNode(9)

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

Doubly Linked Lists

In a basic linked list, each item stores a single pointer to the next element.

In a **doubly linked list**, items have pointers to the next *and the previous* nodes.



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 nodes came before it. Not a problem in a doubly linked list.

Not cache-friendly

Most computers have caching systems that make reading from sequential addresses in memory faster than reading from scattered addresses (/article/data-structures-coding-interview#ram).

Array (/concept/array) items are always located right next to each other in computer memory, but linked list nodes can be scattered all over.

So iterating through a linked list is usually quite a bit slower than iterating through the items in an array, even though they're both theoretically $O(n)$ time.

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:

Python 3.6

```
def square_list_in_place(int_list):
    for index, element in enumerate(int_list):
        int_list[index] *= element

    # NOTE: We could make this function just return, since
    # we modify int_list in place.
    return int_list


def square_list_out_of_place(int_list):
    # We allocate a new list with the length of the input list
    squared_list = [None] * len(int_list)

    for index, element in enumerate(int_list):
        squared_list[index] = element ** 2

    return squared_list
```

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:

Python 3.6

```
original_list = [2, 3, 4, 5]
squared_list  = square_list_in_place(original_list)

print("squared: %s" % squared_list)
# Prints: squared: [4, 9, 16, 25]

print("original list: %s" % original_list)
# Prints: original list: [4, 9, 16, 25], confusingly!

# And if square_list_in_place() didn't return anything,
# which it could reasonably do, squared_list would be None!
```

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:

Python 3.6 ▾

```
class LinkedListNode(object):

    def __init__(self, value):
        self.value = value
        self.next  = None
```

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_node.next` into `next` *before* setting `current_node.next` to `previous_node`. Otherwise "stepping forward" at the end could actually mean stepping *back* to `previous_node`!

Python 3.6 ▾

```
def reverse(head_of_list):
    current_node = head_of_list
    previous_node = None
    next_node = None

    # Until we have 'fallen off' the end of the list
    while current_node:
        # Copy a pointer to the next element
        # before we overwrite current_node.next
        next_node = current_node.next

        # Reverse the 'next' pointer
        current_node.next = previous_node

        # Step forward in the list
        previous_node = current_node
        current_node = next_node

    return previous_node
```

We return `previous_node` because when we exit the list, `current_node` is `None`. Which means that the last node we visited—`previous_node`—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.