Interview Cake

You have a linked list.

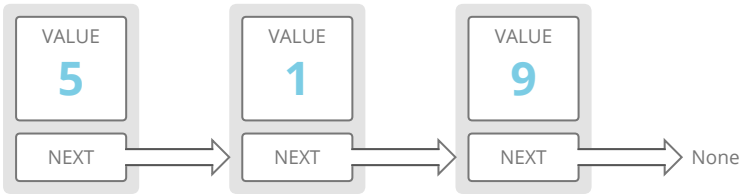
A **linked list** is a low-level data structure. It stores an *ordered* sequence of items in individual "node" objects that have pointers to other nodes.

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

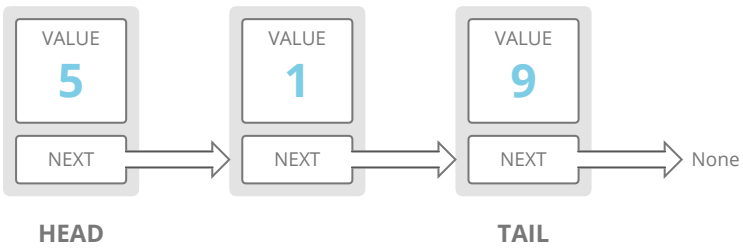
```python
class LinkedListNode:

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

So we could build a singly linked list like this:

```python
a = LinkedListNode(5)
b = LinkedListNode(1)
c = LinkedListNode(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.

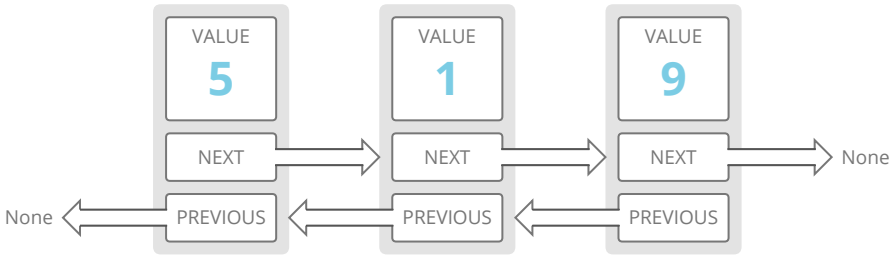
Another type of linked list is a **doubly linked list**, which has pointers to the next *and the previous* nodes.

Python

```
class LinkedListNode:


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

So we could build a doubly linked list like this:

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

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:

Python

```
class LinkedListNode:

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

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