Daily Coding Problem #6

Problem

This problem was asked by Google.

An XOR linked list is a more memory efficient doubly linked list. Instead of each node holding `next` and `prev` fields, it holds a field named `both`, which is an XOR of the next node and the previous node. Implement an XOR linked list; it has an `add(element)` which adds the element to the end, and a `get(index)` which returns the node at index.

If using a language that has no pointers (such as Python), you can assume you have access to `get_pointer` and `dereference_pointer` functions that converts between nodes and memory addresses.

Solution

For the head, `both` will just be the address of next, and if it's the tail, it should just be the address of prev. And intermediate nodes should have an XOR of `next` and `prev`.

Here's an example XOR linked list which meets the above conditions:

```
A     <->     B     <->     C     <->     D

B           A ⊕ C         B ⊕ D           C
```

Let's work through `get` first, assuming that the above conditions are maintained. Then, given a node, to go to the next node, we have to XOR the current node's `both` with the previous node's address. And to handle getting the next node from the head, we would

initialize the previous node's address as 0.

So in the above example, A's `both` is B which when XOR'd with `0` would become B. Then B's `both` is `A ⊕ C`, which when XOR'd with A becomes C, etc.

To implement `add`, we would need to update current tail's `both` to be XOR'd by its current `both` the new node's memory address. Then the new node's `both` would just point to the memory address of the current tail. Finally, we'd update the current tail to be equal to the new node.

```
import ctypes


# This is hacky. It's a data structure for C, not python.
class Node(object):
    def __init__(self, val):
        self.val = val
        self.both = 0


class XorLinkedList(object):
    def __init__(self):
        self.head = self.tail = None
        self.__nodes = [] # This is to prevent garbage collection

    def add(self, node):
        if self.head is None:
            self.head = self.tail = node
        else:
            self.tail.both = id(node) ^ self.tail.both
            node.both = id(self.tail)
            self.tail = node

        # Without this line, Python thinks there is no way to reach nodes between
        # head and tail.
        self.__nodes.append(node)


    def get(self, index):
        prev_id = 0
        node = self.head
```

```
        for i in range(index):
            next_id = prev_id ^ node.both

            if next_id:
                prev_id = id(node)
                node = _get_obj(next_id)
            else:
                raise IndexError('Linked list index out of range')
        return node


def _get_obj(id):
    return ctypes.cast(id, ctypes.py_object).value
```

add runs in O(1) time and `get` runs in O(N) time.
