Daily Coding Problem #258

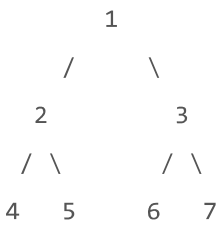
Problem

This problem was asked by Morgan Stanley.

In Ancient Greece, it was common to write text with the first line going left to right, the second line going right to left, and continuing to go back and forth. This style was called "boustrophedon".

Given a binary tree, write an algorithm to print the nodes in boustrophedon order.

For example, given the following tree:



You should return `[1, 3, 2, 4, 5, 6, 7]`.

Solution

At first glance this looks similar to in-order traversal, which we typically implement with a queue. Perhaps we could just keep track of what level of the tree we are on, and append

queue. Perhaps we could just keep track of what level of the tree we are on, and append the right node to the queue before the left on only odd levels.

However, this doesn't quite work: in the example above, if node `3` is visited before node `2`, we cannot stop `6` and `7` from being added to the queue before `4` and `5`, resulting in `[1, 3, 2, 6, 7, 4, 5]`.

Instead, we can keep the idea of appending the right node before the left on certain levels, but use this in a function that is applied to each row of the tree. If the level is zero, our base case, we can print out the value of the node. Otherwise, we should recursively apply our function to the left and right subtrees, in the proper order, decrementing the level count at each step.

```
class Node:
    def __init__(self, data, left=None, right=None):
        self.data = data
        self.left = left
        self.right = right


def print_level(node, level, forward):
    if level == 0:
        print(node.data)
    else:
        if forward:
            print_level(node.left, level - 1, forward)
            print_level(node.right, level - 1, forward)
        else:

            print_level(node.right, level - 1, forward)
            print_level(node.left, level - 1, forward)
```

The `forward` variable determines whether or not a given row should be printed left to right or right to left, and is flipped at each level. To determine how many levels we must start out with, we can use a helper function to find the depth of the tree.

```
def get_height(tree, height=0):
    if not tree:
        return height
    return max(get_height(tree.left, height + 1), get_height(tree.right, height + 1))


def boustrophedon(tree):
    n = get_height(tree)
```

```
    forward = True

    for level in range(n):
        print_level(tree, level, forward)
        forward = not forward
```

If the binary tree is very skewed, the height would be `O(N)`. Since printing each level is also `O(N)`, this algorithm runs in $O(N^2)$ time.

As an alternative, we can solve this iteratively in `O(N)` time by maintaining a stack for the nodes that should be printed forward, and another stack for the nodes that should be printed backward. Each time we pop an element off one stack, we add its child nodes to the other stack. If the element comes from the forward stack, we append the left child before the right; if the element comes from the backward stack, we do the opposite.

For example, applying this to the tree above, we would take the following steps:

- Starting with just the root element in `forward`, pop it and print. Append `2` and then `3` to `backward`.
- Pop `3` from `backward` and print it. Append `7` and then `6` to `forward`.
- Pop `2` from `backward` and print it. Append `5` and then `4` to `forward`.
- Pop and print the elements `4`, `5`, `6`, and `7` from the forward stack.

Here is how we could implement this:

```
def boustrophedon(root):
    forward = [root]
    backward = []

    while backward or forward:
        while forward:
            tmp = forward.pop()
            print(tmp.data)

            if tmp.left:
                backward.append(tmp.left)
            if tmp.right:
                backward.append(tmp.right)
```

```
    while backward:
        tmp = backward.pop()
        print(tmp.data)

        if tmp.right:
            forward.append(tmp.right)
        if tmp.left:
            forward.append(tmp.left)
```

Since we pop each element exactly once, this indeed runs in `O(N)` time, with `O(N)` additional space for the stacks.