[In Class] Lesson 13 (10/13)

Meet Binary Trees

In the pre-class work you were introduced to a new data structure called the **binary tree**. Like Linked Lists, Binary Trees are a structure made up of a collection of connected nodes, however instead of each node holding the reference to one other node, "next", binary tree nodes hold two references, "left" and "right". You can see we are using a static inner node class just like we did previously for LinkedLists.

Since each node can have up to two references we get a branching structure like so:

```
               +---+
               | 6 |
           ___ +---+ ___
         /               \
     +---+               +---+
     | 3 |               | 2 |
     +---+               +---+
    /     \             /     \
+---+     +---+     +---+     +---+
| 1 |     | 4 |     | 6 |     | 4 |
+---+     +---+     +---+     +---+
```

We will refer to the top level node as the **overallRoot**. In our example here the overallRoot has two **child** nodes, one to the **left** and one to the **right**. These nodes are called **branch** nodes as they themselves have children. The bottom level of nodes who have no children are called **leaf** nodes. Computer scientists really love the tree analogy, even though these trees are upside down.

In the pre-class work you saw a constructor that directly set the left and right references, but this approach won't easily scale or adapt to trees of varying size or shape. It might be difficult to see how to use a loop to construct a branching structure like this, but we can see how each tree is made up of smaller sub-trees. This repeating pattern of sub trees within sub trees lends itself well to recursion!

Traversing a tree

Again like Linked Lists, because Binary Trees are made of nodes, we need a way to traverse them without using indices and only by following references.

In the preclass work you wrote the method `size` to traverse the tree and count the number of nodes. Note the structure of this solution: public/private pair with an `IntTreeNode` parameter, a base case when the `IntTreeNode` parameter is `null`, a recursive case that traverses down `node.left` and `node.right`.

How can you adapt this code to complete the `traversal` method?