**More Binary Tree examples**

NOTE: Your homework asks you to write static methods using a public TreeNode class. All the methods here are for the BinaryTree class with an inner BTnode class, for contrast.

**Perfect Binary Trees**

**Definition:** A binary tree is *perfect* if all the leaves are at the same level and every non-leaf has exactly two children. An empty tree is considered perfect.

---

*What is a recursive definition of a perfect tree?*

A tree is perfect if either of the following is true. (WRONG)
- An empty tree.
- Both the left and right subtrees are perfect.

THE ABOVE IS INSUFFICIENT!

*What else needs to be true?*

Can the left subtree be any perfect tree, and right subtree be any perfect tree?
- No. Both must have the same height.

---

**Perfect binary trees recursively:**
A binary tree is perfect if either of the following is true.
- An empty tree.
- Both the left and right subtrees are perfect and have the same height (or size).

---

Given the recursive definition above, write a method `isPerfect` for the BinaryTree class that returns true if the binary tree is perfect or false otherwise.
/ A common solution, but slow (why?)
public boolean isPerfect() {
    return isPerfect(root);
}

private boolean isPerfect(BTNode<E> node) {
    if (node == null) return true;
    else {
        int leftHeight = height(node.left);
        int rightHeight = height(node.right);
        return (leftHeight == rightHeight
            && isPerfect(node.left)
            && isPerfect(node.right));
    }

What is the runtime of the above method in the best case? When?

The best case is when the root has two subtrees of unequal height. \( O(n) \)

What is the runtime of the above method in the worst case? When?

The worst case is when the tree is perfect. \( O(n \log n) \)

Reason:
At level 0: The root does \( O(n) \) work to find the height of the two subtrees.
At level 1: There are two subtrees, each with \( O(n/2) \) nodes. Each subtree does \( O(n/2) \) work to find the heights of its two subtrees, for a total of \( O(n) \) work.
At level 2: There are four subtrees, each with \( O(n/4) \) nodes. Each subtree does \( O(n/4) \) work to find the heights of its two subtrees, for a total of \( O(n) \) work.
....
At level \( \log n \): \( O(n) \) leaves, each doing \( O(1) \) work, for a total of \( O(n) \) work.

Overall: \( O(n \log n) \)

Exercise: Write a method \texttt{write a method isPerfect} for the \texttt{BinaryTree} class that returns true if the binary tree is perfect or false otherwise, and has a worst case runtime of \( O(n) \) time.

Hint: The helper method should return an \texttt{int} that indicates whether the tree is perfect or not.
In the BinaryTrees class:

// Correct, but poorly implemented method
public int countLeaves() {
    return myCountNodes(root);
}

public static boolean isLeaf(BTNode node){
    if(node.left == null && node.right == null){
        return true;
    }
    else{
        return false;
    }
}

public static int myCountNodes(BTNode node){
    int counter = 0;

    if(node == null){
    }
    else if(node.left == null && node.right == null){
        return 1;
    }
    else{
        if(node.left!=null){
            if(isLeaf(node.left)==true){
                counter++;
            }
            else{
                counter += myCountNodes(node.left);
            }
        }
        if(node.right!=null){
            if(isLeaf(node.right)==true){
                counter ++;
            }
            else{
                counter += myCountNodes(node.right);
            }
        }
    }
    return counter;
}
Problems with the above code:

countLeaves:
- The helper method is called myCountNodes. But it is counting leaves not nodes.

isLeaf:
- It public static, when it should be private. A client does not have access to any BTNodes, since BTNode is an inner class.
- The method could be more concise: Simply return the boolean expression
- It should have a precondition: node != null

myCountNodes:
- The (node == null) conditional has an empty body, which makes it hard to understand why it is there.
  - Either set the counter to 0 or return 0
- When is (node == null) test needed? Since the method tests if a subtree is null before making a recursive call, the only time the method could be called with a null argument is the first time, when the root is null.
  - Either move the test to the wrapper so that the test is done only once on the root, or don’t test before the recursive calls.
  - In the first case, you should add a precondition that the node != null
- It tests if the node is a leaf without using the helper method isLeaf.
- But it also tests if a subtree is a leaf before make a recursive call.
  - As with the null case above, either move the test whether the node is a leaf to the wrapper method or don’t test before the recursive calls and let the recursive call test if a node is a leaf.

// Better solution
public int countLeaves() {
    return countLeaves(root);
}

// precondition: node != null
private boolean isLeaf(BTNode node){
    return (node.left == null && node.right == null);
}

private int countLeaves(BTNode node){
    if (node == null) {
        return 0;
    }
    else if (isLeaf(node)){
        return 1;
    }
    else {
        return countLeaves (node.left) +
                countLeaves (node.right);
    }
}
In the BinaryTree class:

```java
// Initialize the root to refer to a perfect binary tree
// with height n and every node's data is element.

class BinaryTree {
    E element;
    int n;
    public BinaryTree(E element, int n) {
        root = build(element, n);
    }

    private BTNode build(E element, int n) {
        if (n < 0) {
            return null;
        } else {
            return new BTNode(element, build(element, n-1), build(element, n-1));
        }
    }
}
```

**Runtime:** $O(2^n)$ (How many nodes will be in the final tree? $O(1)$ work for each node)

The following solution does not work. Why?
```
private BTNode build(E element, int n) {
    if (n == 0) {
        return null;
    } else {
        BTNode t = build(element, n-1);
        return new BTNode(element, t, t);
    }
}
```

It doesn’t create a tree, since left and right subtrees refer to the same tree object!

DON’T write a recursive constructor. Silly. For every node, it creates two BinaryTree objects, extracts the root out of each tree, and then throws away the BinaryTree objects.

```java
public BinaryTree(int n, E element) {
    if (n == 0) {
        root = null;
    } else {
        root = new BTNode(element,
                          new BinaryTree(n-1, element).root,
                          new BinaryTree(n-1, element).root);
    }
}
```
Exercise: Write a method for the BinaryTree class that removes nodes that are originally the leaves of this binary tree.

Note: The recursive helper method can either return the root of the resulting tree, or it can be a void method. The former is much easier to write. Can you see why?

// Removes all nodes that were originally leaf nodes
public void pruneLeaves() {

}

// Removes all nodes that were originally leaf nodes.
// Returns a reference to the root of the resulting tree.
private BTNode<E> pruneLeaves(BTNode<E> node) {

}

Exercise: Write a method for the BinaryTree class that returns true if this tree has the same shape as the other tree specified in the parameter.

// Returns true if this tree has the same tree as //otherTree (regardless of their values).
public <E2> boolean sameShape(BinaryTree<E2> otherTree) {

}