Version 1 above runs slowly since it traverses over some parts of the tree many times. A better solution looks at each node only once. The trick is to write a utility helper function isBSTRecur(struct node* node, int min, int max) that traverses down the tree keeping track of the narrowing min and max allowed values as it goes, looking at each node only once. The initial values for min and max should be INT_MIN and INT_MAX -- they narrow from there.

```
/*
 Returns true if the given tree is a binary search tree
 (efficient version).
*/
int isBST2(struct node* node) {
  return(isBSTRecur(node, INT_MIN, INT_MAX));
}

/*
 Returns true if the given tree is a BST and its
 values are >= min and <= max.
*/
int isBSTRecur(struct node* node, int min, int max) {
```

15. Tree-List

The Tree-List problem is one of the greatest recursive pointer problems ever devised, and it happens to use binary trees as well. CLibarary #109 http://cslibrary.stanford.edu/109/ works through the Tree-List problem in detail and includes solution code in C and Java. The problem requires an understanding of binary trees, linked lists, recursion, and pointers. It's a great problem, but it's complex.

Section 3 -- C/C++ Solutions

Make an attempt to solve each problem before looking at the solution -- it's the best way to learn.

1. Build123() Solution (C/C++)

```
// call newNode() three times
struct node* build123a() {
  struct node* root = newNode(2);
  struct node* lChild = newNode(1);
  struct node* rChild = newNode(3);

  root->left = lChild;
  root->right= rChild;

  return(root);
}

// call newNode() three times, and use only one local variable
struct node* build123b() {
  struct node* root = newNode(2);
  root->left = newNode(1);
  root->right = newNode(3);

  return(root);
}
```

14. isBST2() Solution (C/C++)

```
/*
 Returns true if the given tree is a binary search tree
 (efficient version).
*/
int isBST2(struct node* node) {
  return(isBSTUtil(node, INT_MIN, INT_MAX));
}

/*
 Returns true if the given tree is a BST and its
 values are >= min and <= max.
*/
int isBSTUtil(struct node* node, int min, int max) {
  if (node==NULL) return(true);

  // false if this node violates the min/max constraint
  if (node->data<min || node->data>max) return(false);

  // otherwise check the subtrees recursively,
  // tightening the min or max constraint
  return
    isBSTUtil(node->left, min, node->data) &&
    isBSTUtil(node->right, node->data+1, max)
  );
}
```


15. TreeList Solution (C/C++)

The solution code in C and Java to the great Tree-List recursion problem is in CSLibrary #109
http://cslibrary.stanford.edu/109/

Section 4 -- Java Binary Trees and Solutions

In Java, the key points in the recursion are exactly the same as in C or C++. In fact, I created the Java solutions by just copying the C solutions, and then making the syntactic changes. The recursion is the same, however the outer structure is slightly different.

In Java, we will have a BinaryTree object that contains a single root pointer. The root pointer points to an internal Node class that behaves just like the node struct in the C/C++ version. The Node class is private -- it is used only for internal storage inside the BinaryTree and is not exposed to clients. With this OOP structure, almost every operation has two methods: a one-line method on the BinaryTree that starts the computation, and a recursive method that works on the Node objects. For the lookup() operation, there is a BinaryTree.lookup() method that the client uses to start a lookup operation. Internal to the BinaryTree class, there is a private recursive lookup(Node) method that implements the recursion down the Node structure. This second, private recursive method is basically the same as the recursive C/C++ functions above -- it takes a Node argument and uses recursion to iterate over the pointer structure.

Java Binary Tree Structure

To get started, here are the basic definitions for the Java BinaryTree class, and the lookup() and insert() methods as examples...

```
// BinaryTree.java
public class BinaryTree {
  // Root node pointer. Will be null for an empty tree.
  private Node root;


  /*
   --Node--
   The binary tree is built using this nested node class.
   Each node stores one data element, and has left and right
   sub-tree pointer which may be null.
   The node is a "dumb" nested class -- we just use it for
   storage; it does not have any methods.
  */
  private static class Node {
    Node left;
    Node right;
    int data;

    Node(int newData) {
      left = null;
      right = null;
      data = newData;
    }
  }

  /**
   Creates an empty binary tree -- a null root pointer.
  */
  public void BinaryTree() {
    root = null;
  }


  /**
   Returns true if the given target is in the binary tree.
   Uses a recursive helper.
  */
  public boolean lookup(int data) {
    return(lookup(root, data));
  }


  /**
   Recursive lookup  -- given a node, recur
   down searching for the given data.
  */
  private boolean lookup(Node node, int data) {
    if (node==null) {
      return(false);
    }

    if (data==node.data) {
      return(true);
    }
    else if (data<node.data) {
```

```
    return(lookup(node.left, data));
  }
  else {
    return(lookup(node.right, data));
  }
}


/**
 Inserts the given data into the binary tree.
 Uses a recursive helper.
*/
public void insert(int data) {
  root = insert(root, data);
}


/**
 Recursive insert -- given a node pointer, recur down and
 insert the given data into the tree. Returns the new
 node pointer (the standard way to communicate
 a changed pointer back to the caller).
*/
private Node insert(Node node, int data) {
  if (node==null) {
    node = new Node(data);
  }
  else {
    if (data <= node.data) {
      node.left = insert(node.left, data);
    }
    else {
      node.right = insert(node.right, data);
    }
  }

  return(node); // in any case, return the new pointer to the caller
}
```


OOP Style vs. Recursive Style

From the client point of view, the BinaryTree class demonstrates good OOP style -- it encapsulates the binary tree state, and the client sends messages like lookup() and insert() to operate on that state. Internally, the Node class and the recursive methods **do not** demonstrate OOP style. The recursive methods like insert(Node) and lookup (Node, int) basically look like recursive functions in any language. In particular, they do not operate against a "receiver" in any special way. Instead, the recursive methods operate on the arguments that are passed in which is the classical way to write recursion. My sense is that the OOP style and the recursive style do not be combined nicely for binary trees, so I have left them separate. Merging the two styles would be especially awkward for the "empty" tree (null) case, since you can't send a message to the null pointer. It's possible to get around that by having a special object to represent the null tree, but that seems like a distraction to me. I prefer to keep the recursive methods simple, and use different examples to teach OOP.

Java Solutions

Here are the Java solutions to the 14 binary tree problems. Most of the solutions use two methods:a one-line OOP