PROBLEM 1:  
(Oversees, Refugees, Guarantees, Search-trees (30 points))

The tree below is a search tree.

Part A (4 points)
What is the pre-order traversal of the tree? (The first value printed is *macaque*)

Part B (2 points)
Provide a word (it doesn’t have to be a real word, it must contain at least four letters) that could be inserted as a left-child of *monkey* so that the tree is still a search tree.

Part C (2 points)
Show by drawing where a node with *gibbon* would be inserted into the tree.

Part D (4 points)
The tree above has a height of four and has three leaves. Draw a search tree with the same values, a height of four, and which has four leaves. Draw the tree on the previous page.

*continued →*
Part E (4 points)
The method `printQ` below prints one line for every node in a tree. The first value printed when called with the tree above is the string `macaque`. What is the complete output?

```java
public static void printQ(TreeNode root){
    Queue<TreeNode> q = new LinkedList<TreeNode>();
    if (root != null){
        q.add(root);
    }
    while (q.size() != 0){
        root = q.remove();
        System.out.println(root.info);
        if (root.left != null) q.add(root.left);
        if (root.right != null) q.add(root.right);
    }
}
```

Part F (4 points)
The method `printS` below prints one line for every node in a tree. The first value printed when called with the tree above is the string `macaque`. What is the complete output?

```java
public static void printS(TreeNode root){
    Stack<TreeNode> st = new Stack<TreeNode>();
    if (root != null){
        st.push(root);
    }
    while (st.size() != 0){
        root = st.pop();
        System.out.println(root.info);
        if (root.right != null) st.push(root.right);
        if (root.left != null) st.push(root.left);
    }
}
```
PROBLEM 2: (Trees (22 points))

For the purposes of this problem, a full, complete binary tree with \( n \) levels has \( 2^{n-1} \) leaf nodes and, more generally, \( 2^{k-1} \) nodes at level \( k \) where the root is at level 1, the root’s two children are at level 2, and so on. The diagram below shows two such trees, the tree on the left is a level-3 full, complete tree and the tree on the right is a level-2 full, complete tree.

In this problem tree nodes have parent pointers. The declaration for such tree nodes follows.

```java
public static class TreeNode {
    String info;
    TreeNode left, right, parent;
    TreeNode(String s, TreeNode lptr, TreeNode rptr, TreeNode pptr) {
        info = s;
        left = lptr; right = rptr;
        parent = pptr;
    }
}
```

Part A (6 points)

Write the method `makeComplete` that returns a full-complete binary tree with the specified number of levels. The call `makeComplete(3,null)` should return a tree such as the one above on the left; `makeComplete(1,null)` should return a single-node tree. The root of the tree has a null parent; all other tree nodes should have correct parent pointers. Use the empty string `"` for the `info` value when creating nodes.

```java
/**
 * Return root of a full complete binary tree with # levels specified,
 * returning null when level == 0.
 * @param level is the level of the full-complete tree
 * @param parent is the parent of the root being created and returned
 */
TreeNode makeComplete(int level, TreeNode parent) {
```

PROBLEM 4: (Family Trees (22 points))

In this problem assume all values in trees are unique, no value appears more than once in a tree. **In this problem the tree is not necessarily a search tree.**

The code in the function `leastAncestor` shown below returns a pointer to the least ancestor of two strings in a tree.

The least ancestor of two string values `p` and `q` is the node furthest from the root (deepest) which is an ancestor of both `p` and `q` (there is a path from the least ancestor to both `p` and `q`).

For example, the tree diagrammed below on the right yields the values shown in the table on the left.

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>least ancestor</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>E</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>G</td>
<td>B</td>
</tr>
<tr>
<td>K</td>
<td>H</td>
<td>C</td>
</tr>
<tr>
<td>H</td>
<td>J</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>G</td>
<td>A</td>
</tr>
<tr>
<td>G</td>
<td>A</td>
<td>A</td>
</tr>
</tbody>
</table>

**Part A (8 points)**

The complexity of `leastAncestor` shown on the next page is not $O(n)$ for an n-node tree. What is the complexity and why? Justify using recurrence relations and an explanation of each part of the recurrence. Provide big-Oh complexities in both the **average case** (assume trees are roughly balanced) and the **worst case**.
Here's code to find the least ancestor, the helper method `inTree` is called from method `leastAncestor`. Note (again) that in this problem trees are not search trees.

```java
public static boolean inTree(TreeNode root, String s) {
    if (root == null) return false;
    if (root.info.equals(s)) return true;
    return inTree(root.left, s) || inTree(root.right, s);
}

public static TreeNode leastAncestor(TreeNode t, String p, String q) {
    if (t == null) return null;

    // first check subtrees (lower than me) for ancestor
    TreeNode result = leastAncestor(t.left, p, q);
    if (result != null) return result;
    result = leastAncestor(t.right, p, q);
    if (result != null) return result;

    // didn't find in subtrees, am I the least ancestor? check
    // me and left/right subtrees for p/q (vice versa)
    if ((t.info.equals(p) || inTree(t.left, p)) &&
        (t.info.equals(q) || inTree(t.right, q))) {
        return t;
    }
    if ((t.info.equals(q) || inTree(t.left, q)) &&
        (t.info.equals(p) || inTree(t.right, p))) {
        return t;
    }

    return null;
}
```
Part B (6 points)

Write the method `findPath` whose header is given below. The method stores values in the `ArrayList` parameter representing the path from the root of `t` to the node containing `target` if there is a path (in which case it returns true). If there is no path from the root to `target` then `list` is empty and the method should return false.

For example, given the tree on the previous page we have:

<table>
<thead>
<tr>
<th>call</th>
<th>ArrayList list</th>
<th>return</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>findPath(t,&quot;F&quot;, list)</code></td>
<td>(A,B,C,F)</td>
<td>true</td>
</tr>
<tr>
<td><code>findPath(t,&quot;A&quot;, list)</code></td>
<td>(A)</td>
<td>true</td>
</tr>
<tr>
<td><code>findPath(t,&quot;J&quot;, list)</code></td>
<td>(A,D,J)</td>
<td>true</td>
</tr>
<tr>
<td><code>findPath(t,&quot;G&quot;, list)</code></td>
<td>(A,B,E,G)</td>
<td>true</td>
</tr>
<tr>
<td><code>findPath(t,&quot;B&quot;, list)</code></td>
<td>(A,B)</td>
<td>true</td>
</tr>
<tr>
<td><code>findPath(t,&quot;X&quot;, list)</code></td>
<td>()</td>
<td>false</td>
</tr>
</tbody>
</table>

Hint: the value of the current node is tentatively on the path before recursive call(s) are made, and is removed from the path if the recursive call(s) fail.

```java
/**
 * Add values to list so that they represent strings
 * on path from t to node containing target. If target
 * not in the tree then no values added to list.
 * @return true if path found, return false otherwise
 */
public static boolean findPath(TreeNode t, String target, ArrayList<String> list) {
    if (t == null) return false;
    if (t.info.equals(target)) {
        list.add(t.info);
        return true;
    }
    // add code here
}
```
Part C (8 points)

Write a version of `leastAncestor` that runs in $O(n)$ time for an $n$-node tree. There are at least two approaches to consider (you can code any algorithm that runs in $O(n)$ time, minimal credit will be given for algorithms that do not run in $O(n)$ time.)

One approach is to call `findPath` twice (from the previous part, assume it works as specified) and find the last value that’s the same in the lists returned, this is the least common ancestor.

Another approach is write an auxiliary function that returns three values (e.g., in an array of Objects or a class with three fields): an ancestor-pointer and a boolean that tells if $p$ is in the tree and a boolean that tells if $q$ is in the tree.

Write the code and justify that it runs in $O(n)$ time.

```java
/**
 * Should run in $O(n)$ time for an $n$-node tree.
 */
public static TreeNode leastAncestor(TreeNode t, String p, String q){
```