Problem Set 5:

Trees

Note: You will be graded in part on your coding style. Your code should be easy to read, well organized, and concise. You should avoid duplicate code.

Homework Source: http://www.cs.cmu.edu/~mrmiller/

Background: The Code

Download hw9.zip. Inside you'll find:

- TreeNode.java - A generic `TreeNode<E>` class with public fields. Open this file and familiarize yourself with it.
- TreeDisplay.java - A user interface for displaying the contents of a binary tree. Use this class to test your code. For example:

```
TreeDisplay<Integer> display = new TreeDisplay<Integer>();
//creates a new window for showing a binary tree of integer
values  display.setRoot(t); //shows the contents of t in the
window, //where t refers to some binary tree of integer values
```

  Note: Modifying a tree that is already being displayed will change what is displayed. If you wish to simultaneously display the tree as it was before and after a modification, then you will need to display the tree and a copy of that tree. To do so, you are encouraged to write a method to make a copy of a tree.

- HW9.java - Contains the 10 static methods that you will be completing. All of the code you write for this assignment should appear in this file. Note that a number of these methods are generic static methods. For example, to declare a static method that determines if a tree contains a given value, we might write the following signature:

```
public static <E> boolean contains(TreeNode<E> t, E obj)
```

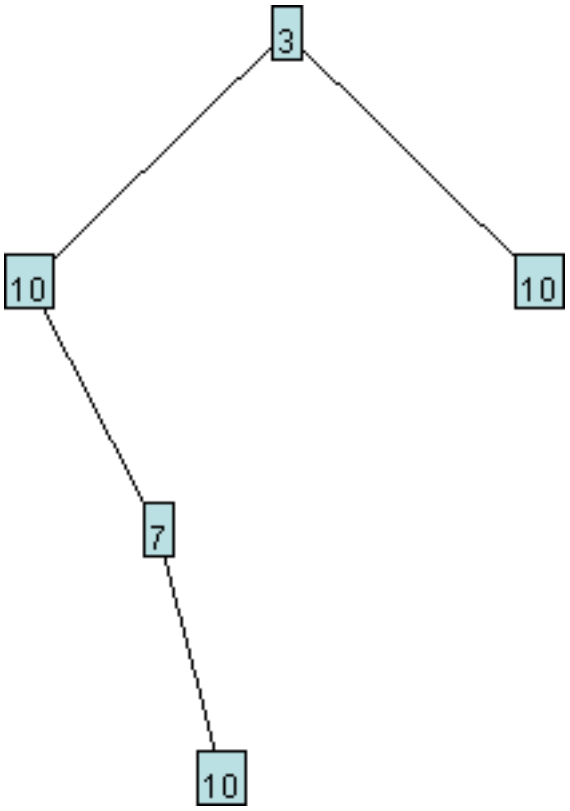
  The return value of this method is `boolean`--not `E`. The `<E>` near the beginning allows us to refer to the type `E` throughout the method. This let's us use the same `contains`method to determine if a tree of integers

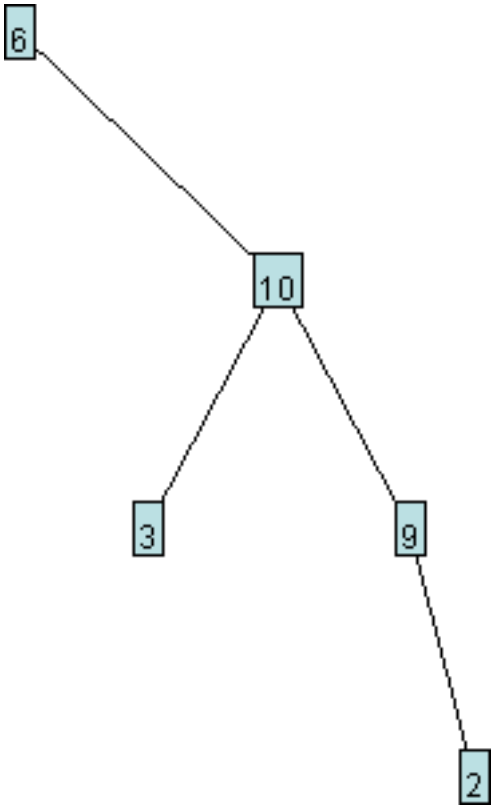
contains a particular integer, and to determine if a tree of strings contains a particular string, and so on.

Exercises

These exercises can be completed in any order. **You should not expect to receive partial credit for methods that do not always work correctly, so please be sure to test your code thoroughly.**

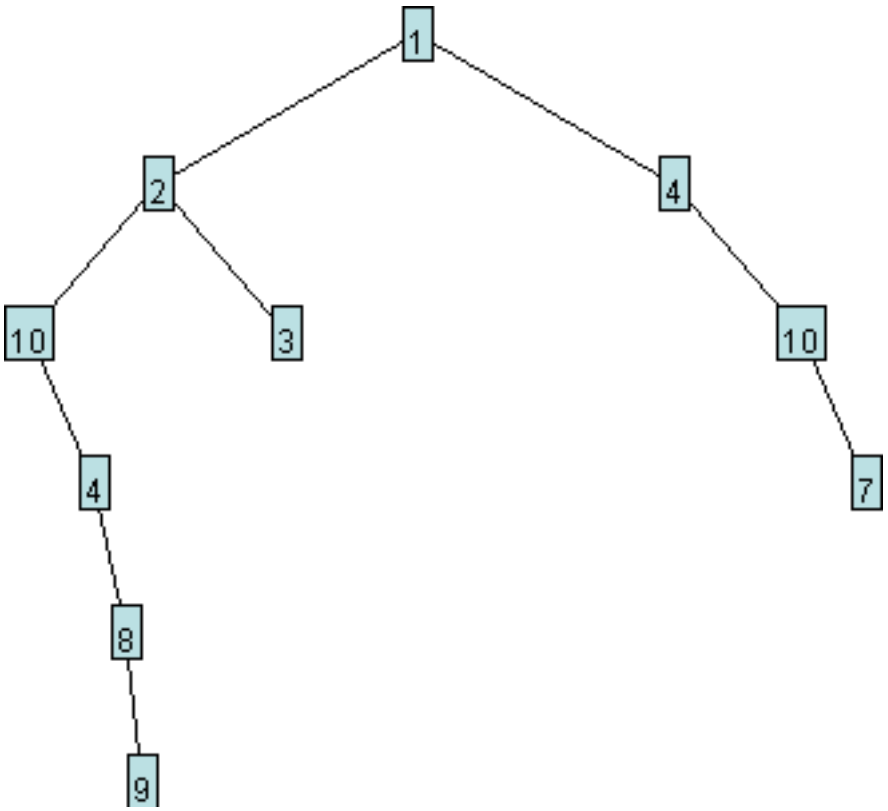
1. Complete the **`randomTree`** method, which should build a randomly shaped tree of integers, with the specified number of nodes, where each node contains a random integer in the range from 1 to `n` inclusive. For example, the following trees were both created by calling `randomTree(5, 10)`:



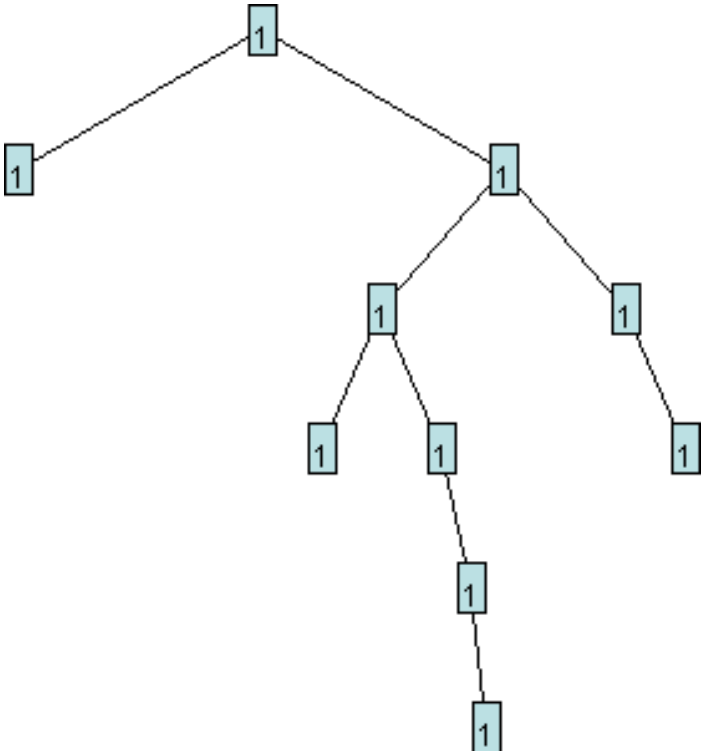


For each node, your method should randomly choose how many of the nodes in the tree should appear to the left of the current node, and how many should appear to the right.

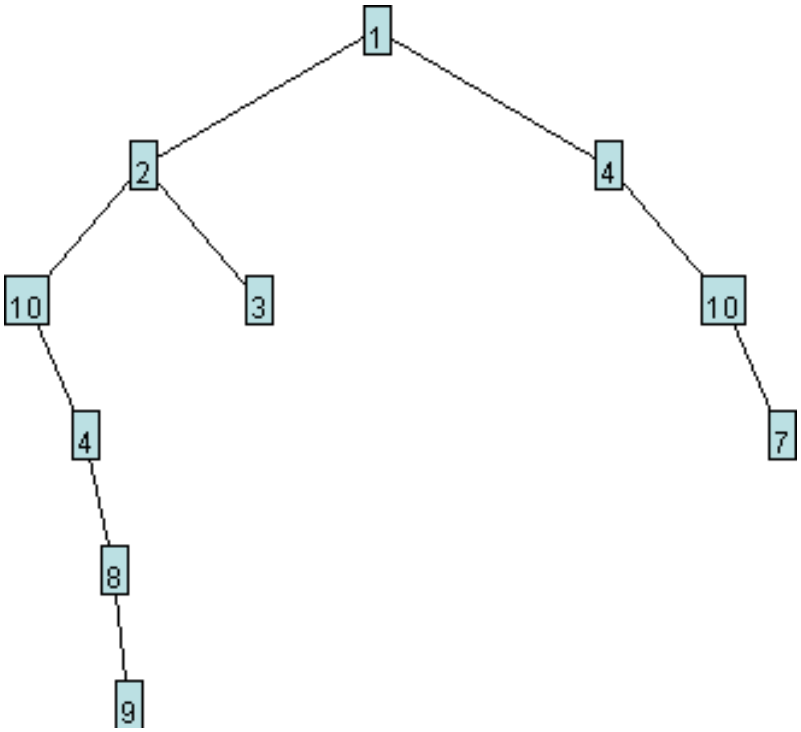
2. Complete the **`replace`** method, which should replace all occurrences of `oldValue` with `newValue` in the given tree, and return the number of occurrences of values that were replaced. This method should work for various data types, including strings, so be sure to use the proper test for equality. Do not create any new nodes.
3. Complete the **`countNodesAtDepth`** method, which should return the number of nodes at the given depth in the given tree. The root node has a depth of 0. In the following tree, there are 2 nodes at depth 1 ("2" and "4"), 3 nodes at depth 2 ("10", "3", and "10"), 1 node at depth 4 ("8"), and no nodes at depth 6.



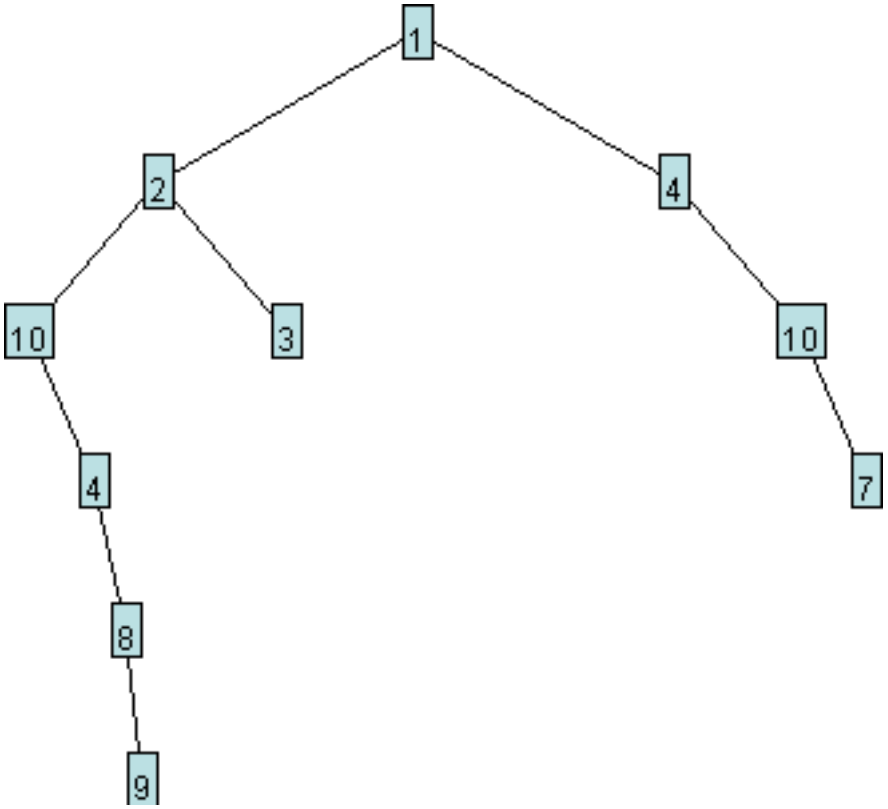
4. Complete the **`allSame`** method, which should return true if and only if every value in the tree is the same. This method should work for various data types, including strings, so be sure to use the proper test for equality. For example, `allSame` should return true for a tree like the following, and for any tree with fewer than 2 elements:



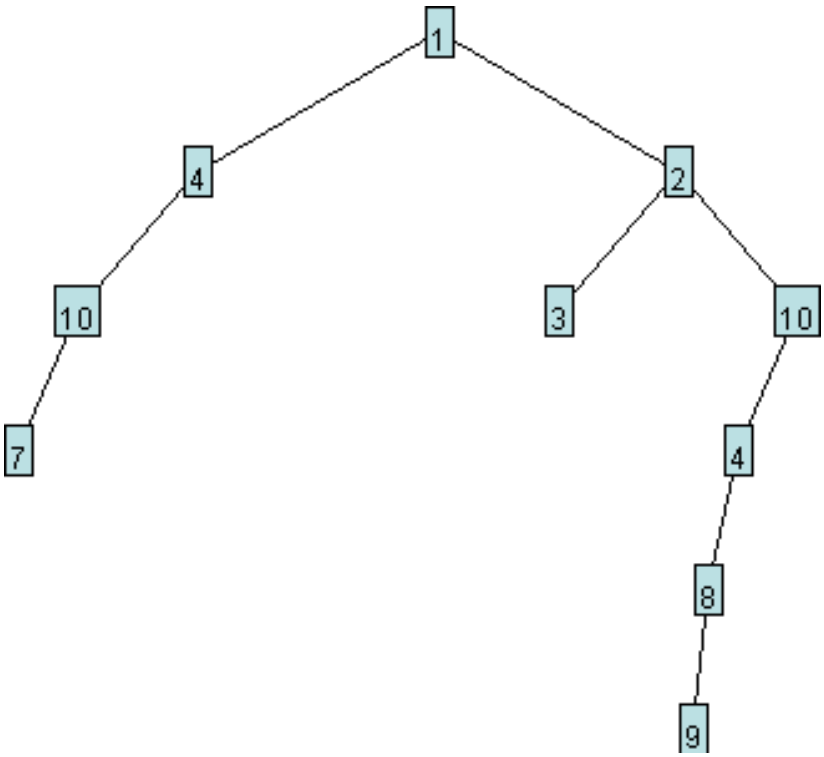
5. Complete the **`leafList`** method, which should return a `List` of the data values that appear in the leaves of the given tree. For the following tree, `leafList` should return the list [9, 3, 7], in that order.



6. Complete the **`reflect`** method, which should modify the given tree so that it is reflected horizontally. For example, if `myTree` originally refers to the following tree

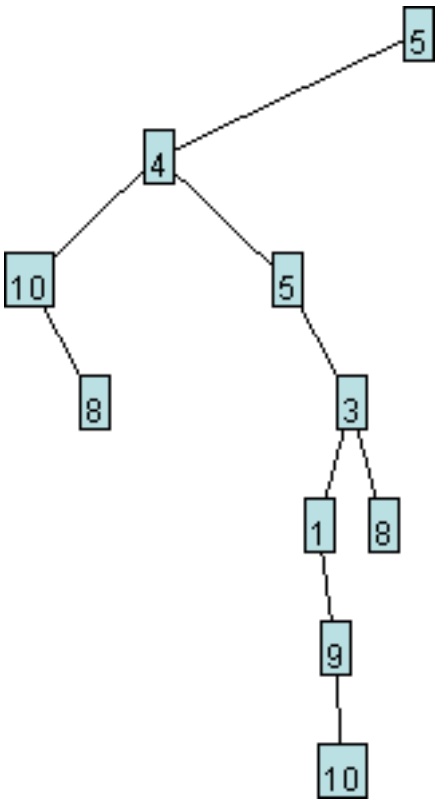


and we now call `reflect(myTree)`, then `myTree` should now appear as follows:

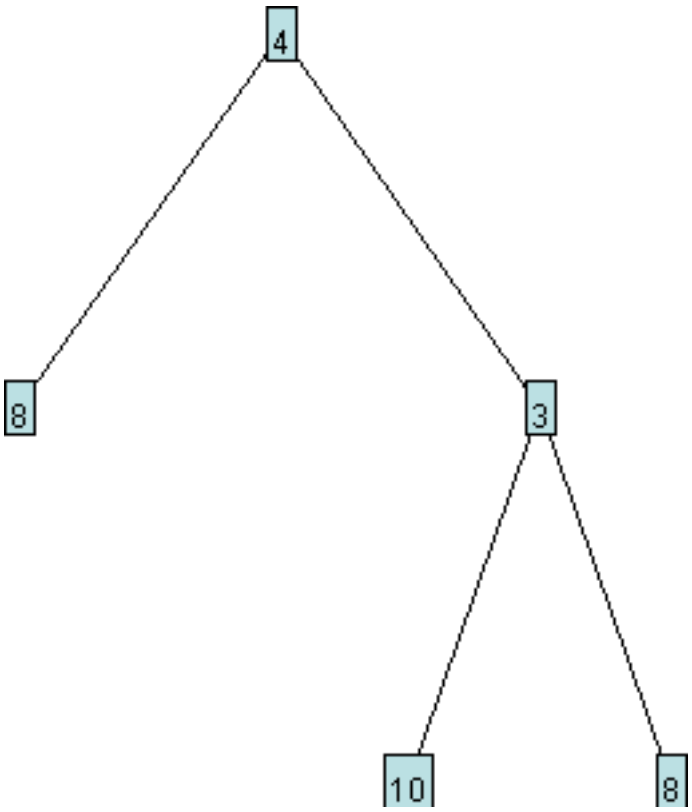


Your method should not create any new nodes.

7. Complete the **`condense`** method, which should remove all nodes that have exactly one child. This method should modify the tree passed to it, and return a reference to the resulting tree. For example, if `myTree` originally refers to the following tree

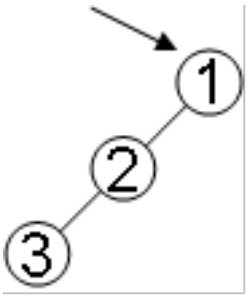


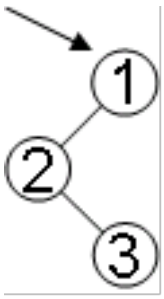
and we now call `condense(myTree)`, then the method will return a reference to the root of the condensed tree, which should appear as follows:

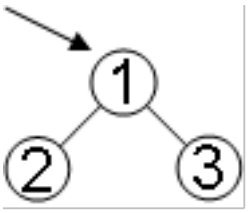


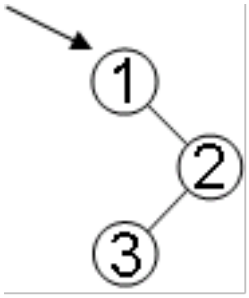
Your method should not create any new nodes.

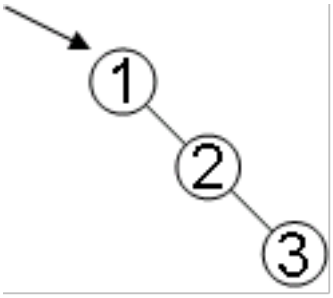
8. We'd like to be able to save the contents of a tree to a text file. Suppose we want to save any of the following trees to a file.



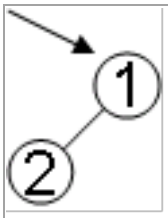


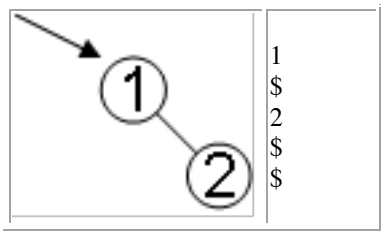






We might be tempted to simply follow a preorder traversal and output the sequence of visited nodes to a file. Unfortunately, however, every one of the trees shown above would result in the same preorder traversal: 1, 2, 3. Therefore, if we stored 1, 2, 3 in a file, we would not have enough information to recover the original shape of the tree. One solution to our problem is to output an extra marker every time we reach a null. We'll use $ for this purpose. Below are several simple examples of a tree and the text file produced when that tree is saved to a file.

Tree	Text File
	$
	1 $ $
	1 2 $ $ $



Complete the **`save`** method, which should save the given tree to the file with the given name, in the format specified above.

Correction: Please modify the method declaration so that the `save` method is `static` (if it wasn't already).

In case you are unfamiliar with file saving in Java, here is some sample code for saving some text to a file:

```
PrintWriter out = new PrintWriter(new FileWriter("somefile.txt"));
out.println("some"); out.println("text"); out.close();
```

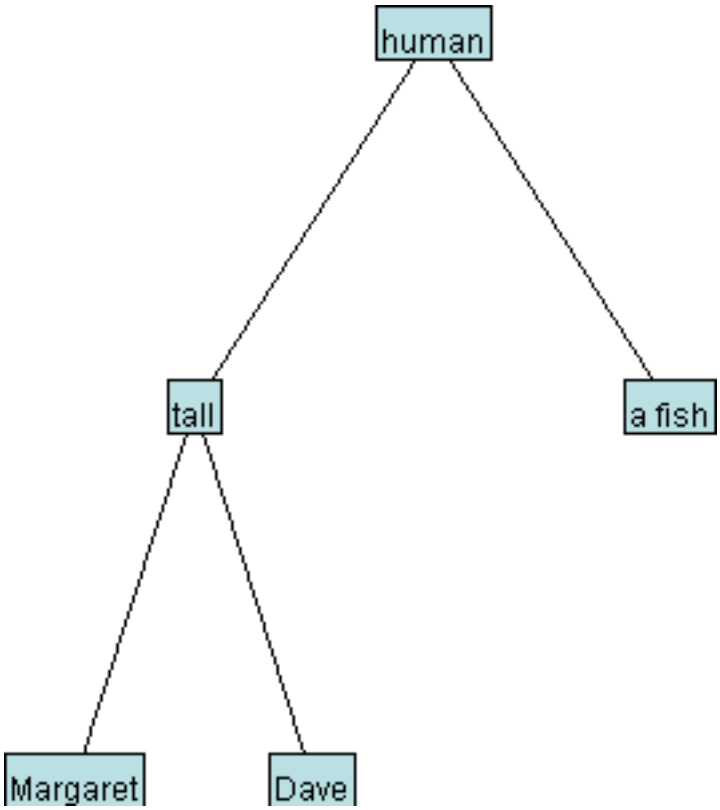
9. Complete the **`load`** method, which takes in the name of a file in the format shown above, and returns the tree represented by that file. In case you are unfamiliar with file loading in Java, here is some sample code for loading some text from a file:

   ```
   Scanner in = new Scanner(new File("somefile.txt")); String
   firstLine = in.nextLine(); String secondLine = in.nextLine();
   in.close();
   ```

10. In this exercise, we'll be programming the computer to play a game of *20 Questions*. You will think of a person or thing, and then you will call the twentyQuestions method, which will direct the computer to attempt to guess what you are thinking of by asking you a series of yes/no questions. (Traditionally, the computer would be limited to 20 questions, but we will ignore that limit for this exercise.) In order for the computer to ask semi-intelligent questions, it must have some knowledge of the world. In this exercise, that knowledge will take the form of a decision tree.

    Our decision tree will contain one or more strings, and every node will have zero or two children. The strings in the leaves are the people and things that the computer knows about. The strings in the other nodes are questions the computer can ask to help identify which person or thing you are thinking of. All of the people/things in the left subtree of a

question correspond to an answer of *yes*, while the people/things in the right subtree correspond to an answer of *no*. Consider the following decision tree:



This tree has 3 people/things: Margaret, Dave, and a fish. And it has 2 questions: Is it human? and Is it tall? All of the humans appear to the left of "human", and all of the non-humans appear to the right. Likewise, all of the tall humans appear to the left of both "human" and "tall".

The following transcript shows the result of a call to `twentyQuestions` using the decision tree shown above. The bold text is printed to the console by the `twentyQuestions`method, and the yes/no answers are typed into the console by the user.

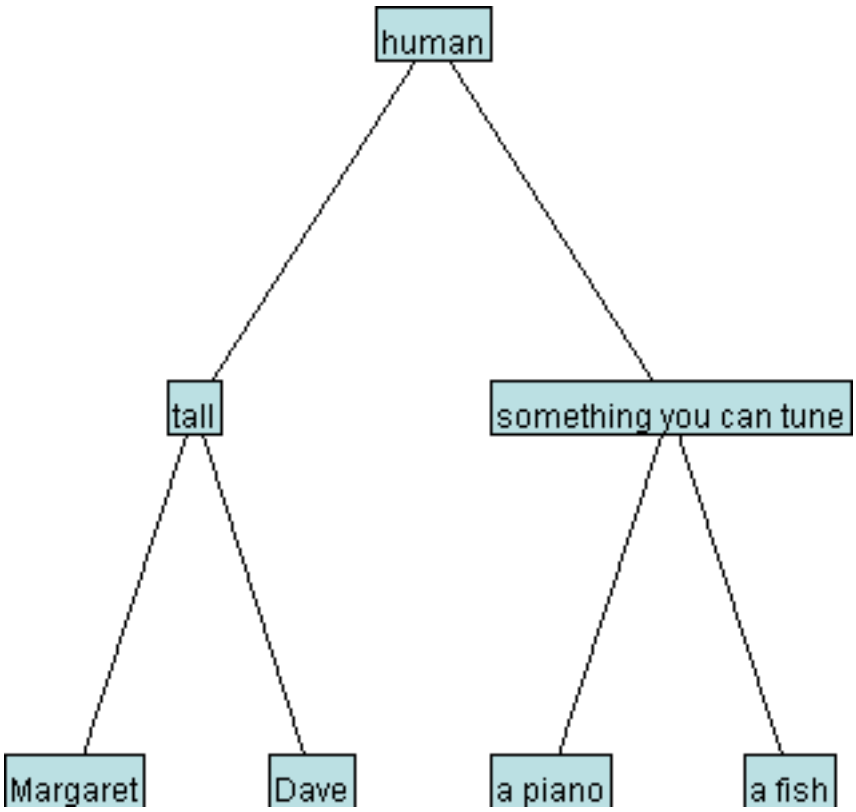
```
Is it human? yes
Is it tall? no
Is it Dave? yes
I win!
```

On the other hand, if the computer's final guess is incorrect, the computer will ask for the correct answer and a characteristic to distinguish that correct answer from the computer's incorrect answer,

and will modify the decision tree to reflect this new knowledge. This is illustrated in the transcript below for a different call to `twentyQuestions`, using the same decision tree:

```
Is it human? no
Is it a fish? no
I give up. Who/what is it? a piano
What distinguishes a piano from a fish?
a piano is something you can tune
```

After this call to `twentyQuestions`, the decision tree will now appear as follows:



Go ahead and complete the `twentyQuestions` method. The printed text should be formatted **exactly** as shown in the sample transcripts above. When new knowledge must be added to the tree, your code should create **only two nodes**.

Your code cannot change which node is considered to be the root of the decision tree. Your code should work even if there is only one node in the tree.

The static variable `userInput` has been initialized to refer to a `Scanner` for reading user input from the console. The following line of code will allow the user to enter a line of text and store the result in `s`:

```
String s = userInput.nextLine();
```

Optional: When you have finished this, you might consider creating a little game for yourself, where the computer loads its knowledge from a file, and plays multiple rounds of twenty questions, saving the modified tree whenever it learns something new.

Submitting Your Work

When you have completed this assignment and tested your code thoroughly, create a .zip file with your work (including HW9.java). Our online handin system only accepts .zip files. (To create a .zip file on a lab computer, right-click on the folder with your code and choose "create archive", making sure to choose ".zip" from the dropdown box on the upper right of the dialog.

Make sure to keep a copy of your work just in case!