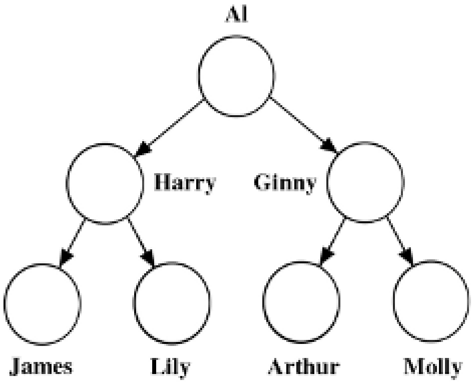
Computer Laboratory 12
CSci 1913: Introduction to Algorithms, Data Structures, and Program Development
April 18/19, 2017

0. Introduction.

This laboratory exercise asks you to experiment with building and traversing binary trees. You will write some code that is based on the depth-first traversal algorithms which were discussed in class, such as `inorder`, `postorder`, and `preorder`. The trees used in this assignment are *not* binary search trees!

1. Theory.

Since each human has two biological parents, a mother and a father, we can use a binary tree to represent his or her ancestors. For example, the following binary tree shows ancestors of someone named Al. Al's father is Harry, and his mother is Ginny. Similarly, Harry's father is James, and Harry's mother is Lily. Ginny's father is Arthur, and Ginny's mother is Molly.



We can answer questions about Al's ancestry by traversing the binary tree. For example, we might ask if Al is descended from Molly. He is, because there is a path from the node labeled Al to the one labeled Molly. Similarly, we might ask if Ginny is descended from James. She isn't, because there is no path from the node labeled Ginny to the one labeled James. And we might ask if Harry is descended from Salazar. He isn't, because there is no node labeled Salazar in the tree. (But for all we know, he might be, if the tree included more ancestors of his parents.)

2. Implementation.

You must implement a Java class called `FamilyTree` that uses a binary tree to represent someone's ancestors. It must have a `private` class called `Node` whose instances represent nodes in the tree. The `Node` class must have a `String` slot called `name`, and `Node` slots called `father` and `mother`. These act like the `left` and `right` slots discussed in class.

The `FamilyTree` class must also have at least the following methods. Note that Java allows two or more methods to have the same name if they take different arguments. Also note that private methods are used to help implement public ones.

```
public FamilyTree(String ego)
```

Constructor. Make a new instance of `FamilyTree`. It contains a binary tree with one `Node`. The `name` slot of this `Node` must be `ego`. Its `father` and `mother` slots must be `null`.

```
private Node find(String name)
```

Call find with the arguments name, and the Node at the root of the tree. Return whatever the two-argument version of find returns, either a Node or null.

```
private Node find(String name, Node root)
```

Do a recursive, depth-first traversal of the tree whose root is root. If the traversal visits a Node whose name slot is name, then return that Node. If the traversal never visits such a Node, then return null.

```
public void addParents(String ego, String father, String mother)
```

Find the Node whose name slot is ego. If there is no such Node, then throw an IllegalArgumentException. Otherwise, set the father slot of the Node to be a new Node, whose name slot is father. Set the mother slot of the Node to be a new Node, whose name slot is mother. The father and mother slots of both new Node's must be null.

```
public boolean isDescendant(String ego, String ancestor)
```

Test if a person named ego is descended from a person named ancestor. Use find to obtain the Node's from the tree that correspond to ego and ancestor. If either of these Node's do not exist in the tree, then return false. Otherwise call isDescendant with the Node's as its arguments, and return its result.

```
private boolean isDescendant(Node root, Node ancestor)
```

Do a recursive, depth-first traversal of the tree whose root is root. If the traversal visits ancestor, then return true. If the traversal never visits ancestor, then return false.

Here are some hints. First, if you want to compare Node's, then you must use ==. This is because we aren't interested in the contents of the Node's: we just want to test if two pointers reference the same Node. Of course you must still use equals methods to compare String's.

Second, you may assume that each person is his or her own descendant, and you may assume that no two persons in the tree have the same name. You may also assume that no String is ever null. These assumptions will make your code easier to write.

Third, the "hard" part of writing find and isDescendant may be in figuring out how to return values from deep recursions. Try this: make two recursive calls, one to visit the father subtree, and the other to visit the mother subtree. Save the values returned by each call, and then decide which one to return. (This advice may not make sense until you start to write code.)

3. Deliverables.

The file **tests.java** on Moodle contains Java code that performs a series of tests. The tests call methods from the FamilyTree class. Some of them print what those methods return. Each test is followed by a comment that tells how many points it is worth, and optionally what must be printed if it works correctly.

Run the tests, then turn in the Java source code for your FamilyTree class. Your lab TA will tell you how and where to turn it in. If your lab is on **Tuesday, April 18, 2017,** then your work must be turned in by **11:55 PM** on **Tuesday, April 25, 2017**. If your lab is on **Wednesday, April 19, 2017,** then your work must be turned in by **11:55 PM** on **Wednesday, April 26, 2017**. To avoid late penalties, do not confuse these two dates.