AP Computer Science with Data Structures

Lab: Trees

Exercise: Displaying Trees

Download TreeNode.java, TreeUtil.java, and TreeDisplay.java. `TreeNode` is the class you'll be using to represent binary trees in this lab (and on quizzes). In the TreeUtil.java file, you'll be implementing a number of helpful methods (and some fun ones) for operating on binary trees. Later in the lab, you'll use the `TreeDisplay` class to display traversals of binary trees. First, however, you'll need to implement the three methods in `TreeUtil` that `TreeDisplay` calls.

```
public static Object leftmost(TreeNode t)
public static Object rightmost(TreeNode t)
public static int maxDepth(TreeNode t)
```

Be sure to test these methods on some simple trees. Then, go ahead and try out the `TreeDisplay` class. (You can use the `createRandom` method to build a randomly shaped tree of a given maximum depth.)

```
TreeNode tree = TreeUtil.createRandom(6);
TreeDisplay display = new TreeDisplay();
display.displayTree(tree);
```

Exercise: Counting

Go ahead and implement the following standard binary tree methods.

```
public static int countNodes(TreeNode t)
public static int countLeaves(TreeNode t)
```

Test that each of these works correctly by displaying a random tree and printing the output of each of these methods.

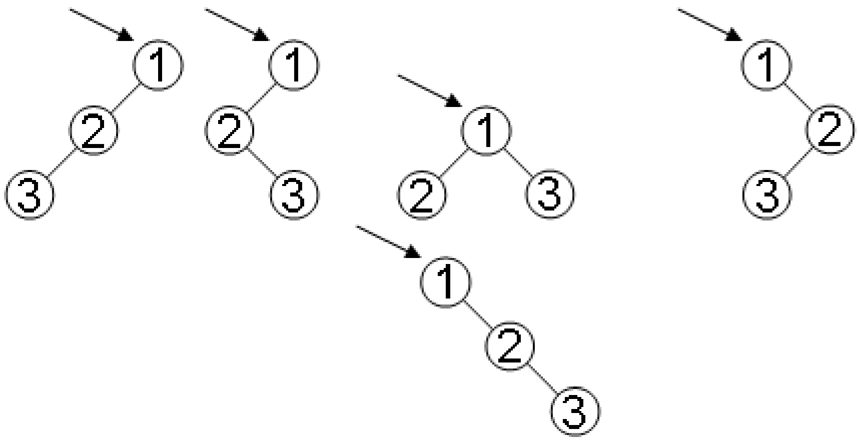
Exercise: Traversals

Implement the three binary tree traversal methods. Notice that each takes in a `TreeDisplay` object, so that we can see the traversal graphically. To light up a node, be sure to call `display.visit(t)` at the appropriate time in your implementation.

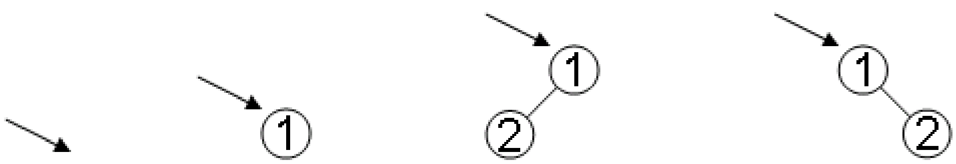
```
public static void preOrder(TreeNode t, TreeDisplay display)
public static void inOrder(TreeNode t, TreeDisplay display)
public static void postOrder(TreeNode t, TreeDisplay display)
```

Exercise: Save the Trees

We'd like to be able to save the contents of a tree to a text file, which is essentially a sequence of text. 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 single one of these trees results 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.



$	1	1	1
	$	2	$
	$	$	2
		$	$
		$	$

Before going on, make sure you know what sequence should be written to a text file when saving any of the trees whose preorder traversal gave 1, 2, 3.

Go ahead and implement the method `fillList`, which takes in a `List` of `String`s and fills that list with the contents of the tree `t`, including `$` markers.

```
public static void fillList(TreeNode t, List<String> list)
```

Now download `FileUtil.java`, and use its `saveFile` method to implement the `saveTree` method below.

```
public static void saveTree(String fileName, TreeNode t)
```

Test that you can correctly save a randomly shaped tree by opening the text file created when you call `saveTree`.

Exercise: Load the Trees

Implement the method `buildTree`, which takes in an `Iterator` of `String`s (including `$` markers) and returns the tree represented by the `String`s returned from the `Iterator`.

```
public static TreeNode buildTree(Iterator<String> it)
```

Finally, use `FileUtil`'s `loadFile` method to implement the `loadTree` method below.

```
public static TreeNode loadTree(String fileName)
```

Test that you can now correctly load back and display a saved tree.

Exercise: The shape of things to come

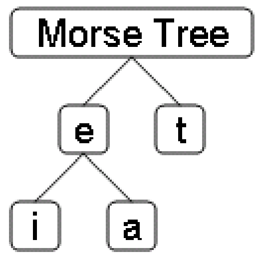
Given a binary tree `t`, a copy of `t` is a new tree with all new `TreeNode` objects pointing to the same values as `t` (in the same order, shape, etc.) Two trees have the same shape if they have `TreeNode` objects (not necessarily the values in the TreeNodes) in the same locations relative to the root.

Complete and test the methods `copy` and `sameShape` in `TreeUtilities`. These methods must be implemented *correctly* to receive credit. Converting the tree to a list in order to copy it or determine its shape will not receive credit.

```
public static TreeNode copy(TreeNode t)
public static boolean sameShape(TreeNode t1, TreeNode t2)
```

Exercise: Coding Theory

Write a program to decode Morse code messages into plain text. The decoding is implemented with the help of a binary "decoding" tree. Morse code for each letter represents a path from the root of the tree to some node: a dot means "go left", while a dash means "go right". The node at the end of the path contains the symbol corresponding to the code. Try decoding the Morse code message `". .- -"` using the following decoding tree. (Data compression algorithms make use of such decoding trees.)



The following program decodes this message.

```
TreeNode decodingTree = createDecodingTree(display);
System.out.println(decodeMorse(decodingTree, ". .- -", display));
```

Although you've been given the `createDecodingTree` method, you'll need to write the `insertMorse` method it calls to add a `letter` to the appropriate position in the `decodingTree` (as determined by the letter's `code`), lighting up the `display` as it walks down the `decodingTree`.

```
private static void insertMorse(TreeNode decodingTree,
           String letter, String code, TreeDisplay display)
```

You'll then need to write the method `decodeMorse`, which uses the `decodingTree` you built to decipher a `cipherText` Morse code message (such as `". .- -"`), again lighting up the `display` as it walks down the `decodingTree`. When you are done implementing this method correctly remove the call to display.visit() while traversing down the tree and retain it only when you reach a character and append it to the decoded message.

```
public static String decodeMorse(TreeNode decodingTree,
                   String cipherText, TreeDisplay display)
```

If you finish early

Level of difficulty: 10/10

An algebraic expression with parentheses and defined precedence of operators can be represented by a binary tree, called an **expression tree**. For example, we can represent the expression "$(a + 1)(a - 1)$" with the following tree.

```
          *
        /   \
       +     -
      / \   / \
     a   1 a   1
```

Try drawing an expression tree for the expression "$2 / (1/x + 1/y)$".

Now write the recursive method `eval`, which evaluates the expression represented by `expTree`. Assume that the nodes contain `Integer` operands and the `String` operators "`+`" and "`*`".

```
public static int eval(TreeNode expTree)
```

For example, the following code prints the value of the expression "4 * (2 + 3)".

```
tree = new TreeNode("*",
     new TreeNode(new Integer(4)),
     new TreeNode("+",
          new TreeNode(new Integer(2)),
          new TreeNode(new Integer(3))));
System.out.println(eval(tree));
```

Now implement the method `createExpressionTree`, which takes in a string expression (consisting of integers, "`+`", "`*`", and parentheses) and returns an expression tree that represents it (again consisting of `Integer` objects and `String` operators).

```
public static TreeNode createExpressionTree(String exp)
```

For example, the following code creates an expression tree and evaluates it.

```
TreeNode tree = createExpressionTree("(5+6)*(4*(2+3))");
display.displayTree(tree);
System.out.println(eval(tree));
```

The `createExpressionTree` method is fairly difficult to implement, and there are many ways to write it. One way is to sweep across the string from left to right, keeping track of the number of parentheses still open (open parentheses that have not yet been matched by a closing parenthesis). If there are no parentheses still open when you come to an operator, you can recursively create expression trees for the substrings to the left and right of the operator, and join these trees with a new `TreeNode`. If you don't find such an operator, then you've either got an integer (which you'll parse into an `Integer` using the constructor that takes in a string) or an expression in parentheses (which you'll need to extract from the parentheses and recursively create an expression tree from it).

Level of difficulty: 9/10

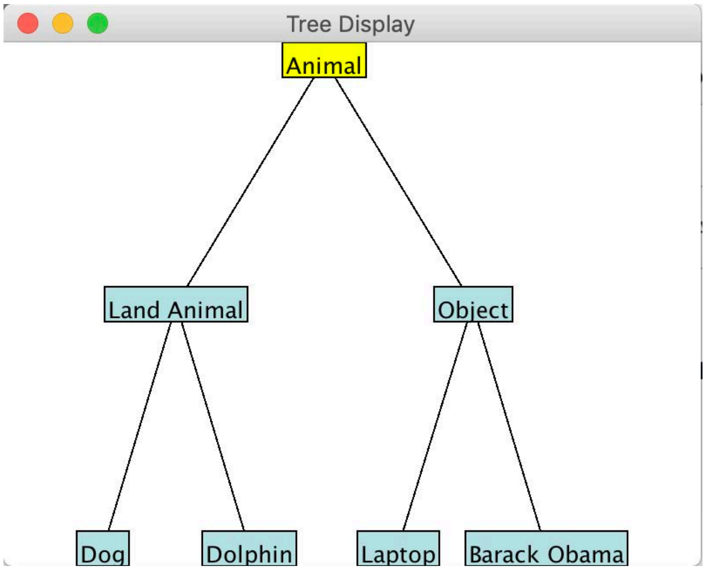
Decision Trees and Game Theory - Let's play a game anyone?
A binary tree can be used to implement various games such as the guessing game or a game of twenty questions.
A game of twenty questions poses yes/no questions to the user traversing down a decision tree to eventually end up at a leaf. The leaves of the decision tree represent answers and the root and internal nodes represent the questions that help determine the answer. A round of twenty questions includes asking a question to the user and recursively repeating the same on the left subtree or right subtree depending on whether the user responds with a yes or a no.

For implementing the game of twenty questions start with a simple knowledge.txt. You may choose to go with any questions of your own choice. Some interesting possiblities are sports, books etc.

Here is an example to demonstrate the working. A simple knowledge.txt could be as follows

```
Animal
Land Animal
Dog
$
$
Dolphin
$
$
Object
Laptop
$
$
Barack Obama
$
$
```

Here is the corresponding decision tree



Let's make a more intelligent game. Machine Learning anyone?
The game will begin by posing the question
Is is an Animal?
If the user responds with a yes the next question posed would be
Is it a Land Animal?
If the user again responds with a yes the answer is displayed as
Is it a Dog?
If the user responds with yes the game responds with
I win and restarts
If at the last step instead of responding with a yes the user responds with a no the game responds with
I give up. What is it?
If the user responds with lion the game asks for the differentiator between a dog and a lion by saying
What distinguishes a lion from Dog?
If the user responds with Wild Animal
the Tree is modified to the following tree. The the game has a chance to learn everytime it is played.