

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.

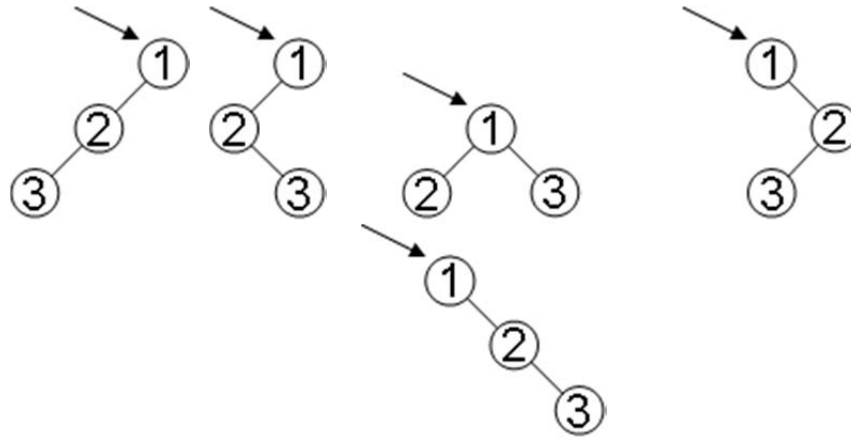
Exercise: Traversals

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.

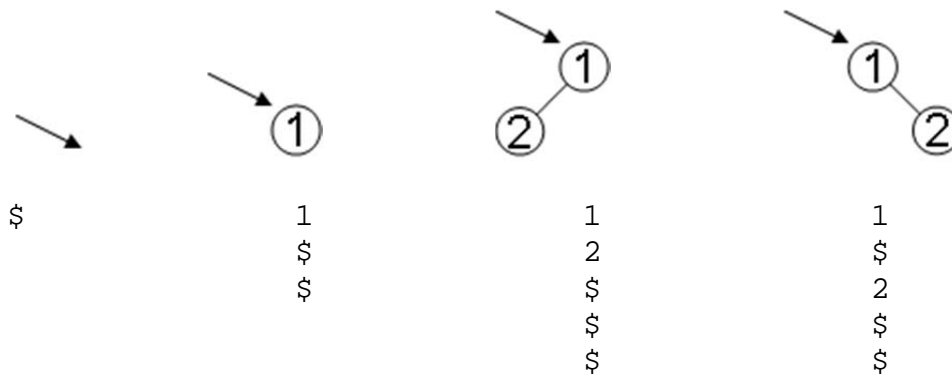
```
public static void preOrder(TreeNode t, TreeDisplay display)
public static void inOrder(TreeNode t, TreeDisplay display)
public static void postOrder(TreeNode t, TreeDisplay display)
```

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.



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 `Strings` 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 `Strings` (including `$` markers) and returns the tree represented by the `Strings` 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: Twenty Questions

In the game of "Twenty Questions", you think of a person or thing, and then answer the computer's yes/no questions. (Traditionally, you would only allow the computer to ask 20 questions, but we'll ignore this limit.) The following shows a transcript of *three rounds* of a game of Twenty Questions.

***Think of a person or thing, and press enter or type
'quit'***

***Is it Barack Obama? no
I give up. Who/what is it? a binary tree
What distinguishes a binary tree from Barack Obama?
a binary tree is a data structure.***

***Think of a person or thing, and press enter or type
'quit'***

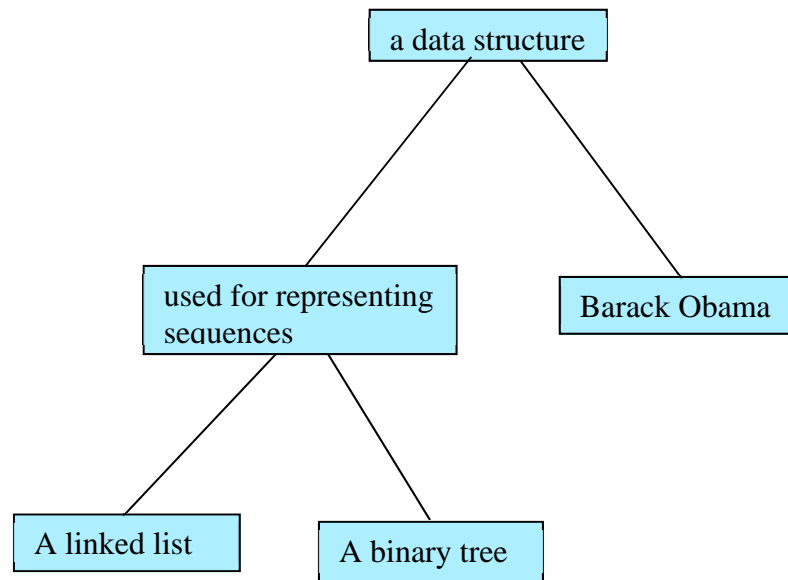
***Is it a data structure? yes
Is it a binary tree? no
I give up. Who/what is it? a linked list
What distinguishes a linked list from a binary tree?
a linked list is used to represent sequences***

***Think of a person or thing, and press enter or type
'quit'***

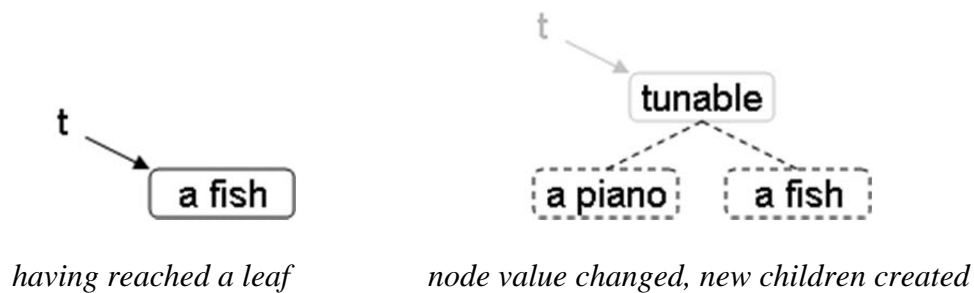
***Is it a data structure? yes
Is it used to represent sequences? no
Is it a binary tree? yes
I win!***

***Think of a person or thing, and press enter or type
'quit'
quit***

You'll be implementing this program using a binary tree like the following to represent the computer's knowledge of the world. The leaves of the tree represent the final guesses, while the interior nodes represent the questions along the way. Notice that an answer of "yes" leads us down the left subtree, while an answer of "no" leads us down the right one.



We will first write the method `twentyQuestionsRound`, which should play only a *single round* of twenty questions, asking the user questions as it walks down the given knowledge tree (lighting up the display as it goes, just for fun). If your program reaches a leaf and does not answer the user's question correctly, you will need to modify the tree. Here is the easiest way to do so. Suppose, having reached a leaf, your program guesses that the user is thinking of "a fish", but the user responds that they were actually thinking of "a piano", and that a piano is "tunable". Your program should *change the value of the original node* to be "tunable", and then hang two new nodes from it, as shown below.



Notice that `twentyQuestionsRound` walks the tree. A recursive implementation will likely be the easiest.

In writing `twentyQuestionsRound`, you should call the method `getUserInput` provided for you in `TreeUtilities`.

```

private static void twentyQuestionsRound(
    TreeNode t, TreeDisplay display)
  
```

Next, download the file knowledge.txt. Complete the `twentyQuestions` method, so that it loads the knowledge tree from that file, displays the tree in a new `TreeDisplay`, and then repeatedly (1) asks the user to think of a person or thing, (2) waits for the user to press enter, or type 'quit' (3) plays a round of twenty questions, and at the end of the game (when the user types 'quit') (4) saves the modified knowledge tree back to knowledge.txt. Note that a single game may involve several rounds and that it you do not store the tree and re-load it after each round. The tree is stored back to `knowledge.txt` only at the end of the game (when the user types 'quit').

```
public static void twentyQuestions()
```

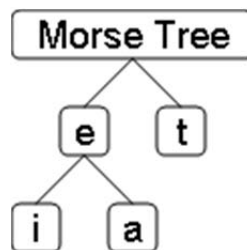
Go test that your game works correctly.

Exercise: The shape of things to come

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.

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)
```

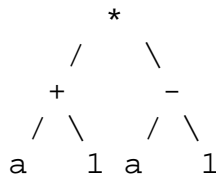
You'll then need to write the method decodeMorse, which uses the decodingTree you built to decipher a cipherText Morse code message (such as ". . - -"), again lighting up the display as it walks down the decodingTree.

```
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.



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).