© 2007 Sudarshan S. Chawathe, with contributions from Mark Royer.

Name: ______________________________

Please note that much of the programming component of this assignment is devoted to getting familiar with the class accounts and submission procedure; thus, it is much simpler than the programming assignments that will follow. You should submit

1. a hard-copy of pages 1–6 of this assignment with your answers filled in, and
2. an electronic package that contains the source files for your work on the programming component. (Details will follow.)

You are welcome, and encouraged, to use any resources (e.g., Web sites) to help you with your work. However, *all such help must be clearly noted* in your submissions. Further, no matter what you use, *you must be able to explain* how and why it works. Refer to the class policy for details, and ask for clarifications if you are unsure if something is allowed.

Questions marked with ⋆ are optional and may be answered for extra credit.

1. (1 pt.) Write your name in the space provided above.

2. (2 pt.) Read the material on the class Web site. Sign your name here to indicate that you have read this material: ____________

3. (2 pt.) Change the passwords on your Gandalf (Unix) and PC cluster accounts (when available). Fill in the following information. (Change your passwords and note *yes* in the last column.)

Account	User Name	Password Changed?
Gandalf	____________	____________
PC cluster	____________	____________

4. (5 pts.) True or false: A binary tree may be defined as a tree in which each node has at most two children. Justify your answer.

5. (10 pts.) The textbook provides both recursive and nonrecursive definitions of trees.[1] Prove that these two definitions are equivalent.

[1] Mark Allen Weiss, *Data Structures and Problem Solving Using Java*, 3rd edition (Addison-Wesley, 2006), pp. 596–597.

6. (10 pts.) Consider the generic implementation of binary trees described in the textbook.[2] Provide a few lines of code illustrating how the binary tree of integers (Java type `int`) suggested by Figure 1 may be created using this implementation. Indicate the points, if any, where your code uses the *autoboxing* and *auto-unboxing* features of Java.

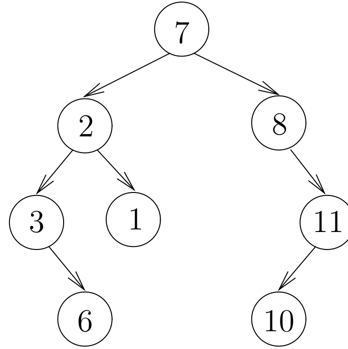

Figure 1: A binary tree of `ints`

[2] *Idem*, p. 605.

7. (10 pts.) Describe a scheme to store binary trees of `int`s without using any explicit pointers. Here the term pointer is used in a general sense, and includes object references. Hint: Use arrays and arithmetic on array indices. Illustrate your scheme by indicating how it may be used to store the binary tree suggested by Figure 1 on page 3.

8. (10 pts.) Quantify the space efficiency of the scheme you provide for Question 7 by providing expressions for the minimum and maximum amount of space (in bytes) required to store a binary tree of n `int`s. Justify your answer, and list any assumptions you need to make.

9. $\star$ (10 pts.) We say two trees are *isomorphic* if there exists a bijection between their sets of nodes that preserves labels and parent-child relationships. In more detail, let T_1 denote a rooted, labeled tree with node-set N_1 and root r_1, with T_2, N_2, and r_2 defined analogously. For a node $n \in N_1 \cup N_2$, let $p(n)$ denote the parent of n in the appropriate tree. We assume N_1 and N_2 are disjoint and define $p(r_1) = r_1$ and $p(r_2) = r_2$ for convenience. Similarly, let $l(n)$ denote the label of node n. We say T_1 is isomorphic to T_2 iff there is a 1-1, onto function $f : N_1 \rightarrow N_2$ such that,

   - for all $n \in N_1$, $l(n) = l(f(n))$ and
   - for all $m, n \in N_1$, $m = p(n)$ if and only if $f(m) = p(f(n))$.

   Describe an efficient algorithm that determines if two trees, given as input, are isomorphic. Analyze the running time of your algorithm as a function of the input size.

10. (50 pts.) In a nutshell, your task here is to implement the interfaces `Tree` and `TreeNode` presented below. You should submit your work following the submission procedure that will be posted on the class Web site.

    We illustrate the behavior of some of the methods in the interface using Figure 2.

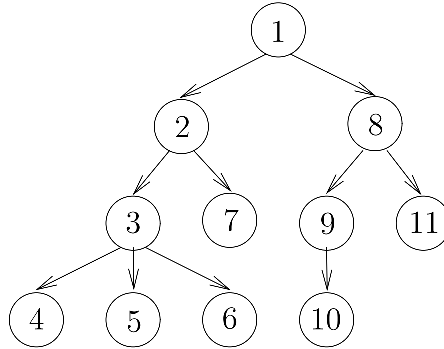

Figure 2: A rooted, labeled tree with 11 nodes.

Interface `Tree`: Refer to the listing `Tree.java` below.

getRoot Returns the root of the tree. For the tree in Figure 2, this returns the node containing 1.

size Returns the number of nodes in the tree. For the tree in Figure 2, this returns 11.

height Returns the length of the path from the root node to the deepest leaf. For the tree in Figure 2, this returns 3.

printPreOrder When invoked on the tree in Figure 2, the following should be displayed: `1,2,3,4,5,6,7,8,9,10,11`. Additional examples and definitions can be found in the textbook.[3]

printPostOrder When invoked on the tree in Figure 2, the following should be displayed: `4,5,6,3,7,2,10,9,11,8,1`.

makeEmpty Removes all of the nodes from the tree.

isEmpty Returns true if the tree contains no nodes.

height This method takes a node as input and returns its height in the tree. For nodes 7 and 8 in Figure 2, this method returns 0 and 2, respectively.

depth This method takes a node as input and returns its depth in the tree. For nodes 4 and 11 in Figure 2, this method returns 3 and 2, respectively.

You should also create a public constructor for the tree that takes a `TreeNode` which will be the root node of the tree.

[3] *Idem*, pp. 600-601.

Tree.java:

```
/**
 * A rooted tree with the following properties.
 *
 * <ul>
 * <li>One node is distinguished as the root.</li>
 * <li>Every node c, except the root, is connected by an edge from exactly one
 * other node p. Node p is c's parent, and c is one of p's children.</li>
 * <li>A unique path traverses from the root to each node. The number of edges
 * that must be followed is the path length.</li>
 * </ul>
 *
 * This can be found on page 596 in Chapter 18 of "Data Structures & Problem
 * Solving Using Java" by Mark Allen Weiss.
 */
public interface Tree {

   /**
    * @return The root of this tree, if the tree is nonempty and null
    * otherwise.
    */
   public TreeNode getRoot();

   /**
    * @return The number of nodes in this tree.
    */
   public int size();

   /**
    * @return The length of the path from the root node to the deepest
    * leaf, if the tree is nonempty, and 0 otherwise.  See the
    * height(TreeNode) and depth(TreeNode) methods below.
    */
   public int height();

   /**
    * Displays all the elements in the tree in an order that ensures
    * parents are displayed before any of their children have been
    * displayed.
    */
   public void printPreOrder();

   /**
    * Displays all the elements in the tree in an order that ensures
    * parents are displayed after all their children have been
```

```
     * displayed.
     */
    public void printPostOrder();

    /**
     * Removes all nodes from this tree.
     */
    public void makeEmpty();

    /**
     * @return true iff this tree has no nodes.
     */
    public boolean isEmpty();

    /**
     * The height of a node in the tree is the length of the path from the node
     * to the deepest leaf in its subtree. See page 596 of the textbook.
     *
     * @param node
     *            The node whose height is desired; it must not be null.
     * @return The height of the node in the tree or -1 if the node is not in
     *         the tree.
     */
    public int height(TreeNode node);

    /**
     * The depth of a node in the tree is the length of the path from the root
     * to the node. See page 596 of the textbook.
     *
     * @param node
     *            The node whose depth is desired; it must not be null.
     * @return The depth of the node in the tree or -1 if the node is not in the
     *         tree.
     */
    public int depth(TreeNode node);
}
```

Interface `TreeNode`: Refer to the listing `TreeNode.java` below.

getElement Returns the element contained by the node. For the root node in Figure 2 this would return 1.

setElement This method takes a non-null element and sets it as this node's element.

getChildren Returns this node's children in list form. (The order is arbitrary.)

addChild This method takes a non-null `TreeNode` and adds it as a child.

size Returns the number of nodes belonging to this node's subtree. If this node has no children then the size is 1.

height Returns the length of the path from this node to its subtrees deepest leaf.

printPreOrder Similar to `printPreOrder` in `Tree`, but only displays this node's subtree in preorder. For instance, invoking this method on node 8 in Figure 2 would display 8,9,10,11.

printPostOrder Similarly, this method displays this node's subtree in postorder. Invoking this method on node 8 from Figure 2 would display 10,9,11,8.

toString This method is not actually specified in `TreeNode`, since every Object in Java has a `toString` method specified in the class Object. You should redefine `toString` in `MyTreeNode` to display the node's element.

In addition, you should create a constructor that takes an integer value and sets the value as the `TreeNode`'s element.

`TreeNode.java`:

```
import java.util.List;

/**
 * Represents a single node in a tree.
 */
public interface TreeNode {

   /**
    * @return This node's element
    */
   public int getElement();

   /**
    * Set this node's element to the argument.
    * @param i The element encapsulated by this node.
    */
   public void setElement(int i);

   /**
    * @return This node's parent.
    */
   public TreeNode getParent();

   /**
    * @return All of this node's direct descendents
    */
```

```
    public List<TreeNode> getChildren();

    /**
     * Makes the given node a child of this node.
     * @param child  The node to add as a child.
     */
    public void addChild(TreeNode child);

    /**
     * @return The number of nodes belonging to this node's subtree, including
     *         this node.
     */
    public int size();

    /**
     * @return The height of this node in the tree.
     *
     * The height of any node is 1 more than the height of its maximum-height
     * child.
     */
    public int height();

    /**
     * Displays all the elements in the subtree rooted at this node in
     * an order that ensures parents are displayed before any of their
     * children have been displayed.
     */
    public void printPreOrder();

    /**
     * Displays all the elements in the subtree rooted at this node in
     * an order that ensures parents are displayed after all their
     * children have been displayed.
     */
    public void printPostOrder();
}
```

Driver `TestMyTree`: A simple test program has been included to serve as a template for code you should write to test your implementation of the above interfaces. Refer to the listing `TestMyTree.java` below. The driver prints the expected and actual output for a few invocations of the interface methods. For the tree of Figure 2, the following output is expected:

```
Preorder tree traversal: 1,2,3,4,5,6,7,8,9,10,11 - 1,2,3,4,5,6,7,8,9,10,11
```

```
Postorder tree traversal: 4,5,6,3,7,2,10,9,11,8,1 - 4,5,6,3,7,2,10,9,11,8,1
Tree root: 1,1
Tree size: 11,11
Size of 10: 1,1
Size of 8: 4,4
Tree height: 3,3
Height of 1: 3,3
Height of 7: 0,0
Height of 8: 2,2
Height of 10: 0,0
Depth of 20: -1,-1
Depth of 1: 0,0
Depth of 4: 3,3
Depth of 11: 2,2
Depth of 10: 3,3
Depth of 20: -1,-1
Is empty: false,false
Is empty: true,true
```

TestMyTree.java:

```
/**
 * A simple example of using the Tree interface described earlier.  We
 * create the tree of Figure 1 and then compare the expected and
 * actual results of some of the methods applied to that tree.  The
 * tests are not exhaustive and are only meant to provide a brief example
 * of the use of the interface.
 */
public class TestMyTree {

   // We use the following for accesses from the main method.
   static TreeNode four, seven, eight, ten, eleven, twenty;

   public static void main(String[] args) {
      Tree tree = createFigure1Tree();
      // Ouput format: Expected - Actual
      System.out.print("Preorder tree traversal: 1,2,3,4,5,6,7,8,9,10,11 - ");
      tree.printPreOrder();
      System.out
            .print("Postorder tree traversal: 4,5,6,3,7,2,10,9,11,8,1 - ");
      tree.printPostOrder();
      // Ouput format: Expected,Actual
      System.out.println("Tree root: 1," + tree.getRoot());
      System.out.println("Tree size: 11," + tree.size());
      System.out.println("Size of 10: 1," + ten.size());
```

```
        System.out.println("Size of 8: 4," + eight.size());
        System.out.println("Tree height: 3," + tree.height());
        System.out.println("Height of 1: 3," + tree.getRoot().height());
        System.out.println("Height of 7: 0," + tree.height(seven));
        System.out.println("Height of 8: 2," + tree.height(eight));
        System.out.println("Height of 10: 0," + tree.height(ten));
        System.out.println("Depth of 20: -1," + tree.depth(twenty));
        System.out.println("Depth of 1: 0," + tree.depth(tree.getRoot()));
        System.out.println("Depth of 4: 3," + tree.depth(four));
        System.out.println("Depth of 11: 2," + tree.depth(eleven));
        System.out.println("Depth of 10: 3," + tree.depth(ten));
        System.out.println("Depth of 20: -1," + tree.depth(twenty));
        System.out.println("Is empty: false," + tree.isEmpty());
        tree.makeEmpty();
        System.out.println("Is empty: true," + tree.isEmpty());
    }

    /**
     * Create the tree from Figure 1.
     *
     * @return new Figure 1 tree.
     */
    private static Tree createFigure1Tree() {
        TreeNode three = new MyTreeNode(3);
        four = new MyTreeNode(4);
        three.addChild(four);
        three.addChild(new MyTreeNode(5));
        three.addChild(new MyTreeNode(6));
        TreeNode two = new MyTreeNode(2);
        two.addChild(three);
        seven = new MyTreeNode(7);
        two.addChild(seven);
        eight = new MyTreeNode(8);
        eight.addChild(new MyTreeNode(9));
        ten = new MyTreeNode(10);
        eight.getChildren().get(0).addChild(ten);
        eleven = new MyTreeNode(11);
        eight.addChild(eleven);
        TreeNode root = new MyTreeNode(1);
        root.addChild(two);
        root.addChild(eight);
        return new MyTree(root);
    }
}
```

11. ⋆ (10 pts.) Implement the method you provide for Question 9 and quantify its running time experimentally. Include the source files and experimental results as part of your electronic submission for Question 10, including the necessary details in the README file.

    Your experimental results should include the running time and memory required for a variety of input conditions, with varying input sizes and other relevant parameters.