KEY

FINAL EXAMINATION
MAY 2011
CSC 112 ♦ SECTION 01
INSTRUCTOR: NICHOLAS R. HOWE

YOU MAY USE TWO 8.5"x11" SHEETS OF NOTES ON THIS EXAM.
YOU MAY NOT USE THE TEXTBOOK, A COMPUTER, OR ANY OTHER INFORMATION SOURCE BESIDES YOUR TWO PAGES OF NOTES.

All work should be written in the exam booklet. Partial credit will be granted where appropriate if intermediate steps are shown. When you are done, please sign the statement below. Good luck!

1. **Abstract Data Types** (12 points)

Consider the class defined below in answering the questions that follow.

```
import java.util.*;

public class Tape {
    int position = 0;;
    Stack<Integer> left = new Stack<Integer>();
    Stack<Integer> right = new Stack<Integer>();

    public int goLeft(int input) {
        int value;
        right.push(input);
        if (left.empty()) {
            value = 0;
        } else {
            value = left.pop();
        }
        return value;
    }

    public int goRight(int input) {
        int value;
        left.push(input);
        if (right.empty()) {
            value = 0;
        } else {
            value = right.pop();
        }
        return value;
    }
}
```

a.) Describe abstractly the data structure embodied in this class. (Try to avoid describing how it operates and instead state what it means, perhaps with a physical analogy.)

The class is inspired by the infinite tape on a Turing machine. Effectively, it is an infinite array that extends to both positive and negative indices. The current position can be moved either left or right. In doing so, it returns the value of the array element passed over, and writes a new value to the tape behind it.

b.) The interface to this class works differently from the list iterators provided by the Java collections framework. State the differences, and describe how the interface to this class might be changed to make it operate more like the collections iterators. Propose specific methods that should be added, including possible call signatures, or changes to the operation of existing methods.

The list iterators defined for the Java collections framework don't modify the data structure by default as they move. To make this class interface more like standard iterators, the goLeft and goRight methods should store the same value on the tape as they just read. For modification, there should be a remove method that will delete the last value passed over, an insert method that will put in a new value, and a set method that will change the last value passed. To implement remove and insert, the class will have to keep track of the last move made.

2. **Data Structure Selection** (12 points)

"When all you have is a hammer, everything looks like a nail."

In the very first homework assignment for this course, we wrote a program that carried on an AI-like conversation. Consider the following summary of the program's behavior:

1. Ask the user to predict the number of interactions she wants
2. Print a random greeting.
3. For the specified number of interactions:
   a. Read the user's input
   b. Scan the input word by word, searching for mirror words. If one is found, replace it with the appropriate mirror.
   c. If any mirror words were found, print the mirrored phrase
   d. Otherwise, pick one of several possible canned phrases and say it instead
4. Say goodbye
5. Recap by printing the entire conversation one more time.

At the time we wrote this, the only data structure we knew how to use was the array. Now we have many more tools in our toolbox. What data structures would you recommend using for the tasks in this assignment, and why? How could the program's behavior be improved by using different data structures? If no other data structure improves upon an array, you can also say that. Justify each of your choices, under the assumption that you plan to scale up this program to handle much longer and more complex conversations.

If the conversation is stored in an ArrayList or similar data structure instead of a simple array, then it will not be necessary to predict in advance how long the conversation should last. The mirror words should be stored in a HashMap or similar structure so that they can quickly be checked, and the appropriate mirror selected.

3. **Hash Functions** (12 points)

Consider the following implementation of `.equals`, which is to be added to the `BinaryTree` class we used in lecture and lab.

```
public static boolean equal(BinaryTree tree1, BinaryTree tree2) {
    return ((tree1 == null)&&(tree2==null))||
        ((tree1!=null)&&(tree2!=null)
         &&equal(tree1.left,tree2.left)
         &&equal(tree1.right,tree2.right));
}

public boolean equals(Object tree) {
    return (tree!=null)&&
        (this.getClass() == tree.getClass())&&
        equal(this,(BinaryTree)tree);
}
```

We have seen previously that when .equals is redefined, it is customary to redefine .hashCode as well. Please comment on the advantages and/or disadvantages of the following treatments of .hashCode:

a.) Leave it as is (i.e., accept the inherited default)

With this option, the definition of equals will not match the definition of hashCode. Two objects that are .equals will probably not generate the same hashCode. This means that lookup in a HashMap or similar data structure will not retrieve the stored values properly.

b.) Define as shown below.

```
    public int hashCode() {
        int result = data.hashCode();
        if (left!=null) {
            result += 3*left.hashCode();
        }
        if (right!=null) {
            result += 7*right.hashCode();
        }
        return result;
    }
```

This is the best option of the three shown. It is asymmetric in the left and right subtrees, but that doesn't matter so much. It will still return the same value whenever two trees are .equals, and is unlikely to do so under other circumstances except by rare coincidence. Unlike option c it will return different values if you swap the locations of the node data.

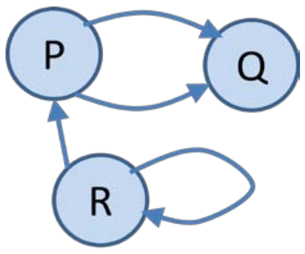
c.) Define as shown below.

```
    public int hashCode1() {
        int result = data.hashCode();
        if (left!=null) {
            result *= left.hashCode();
        }
        if (right!=null) {
            result *= right.hashCode();
        }
        return result;
    }
```

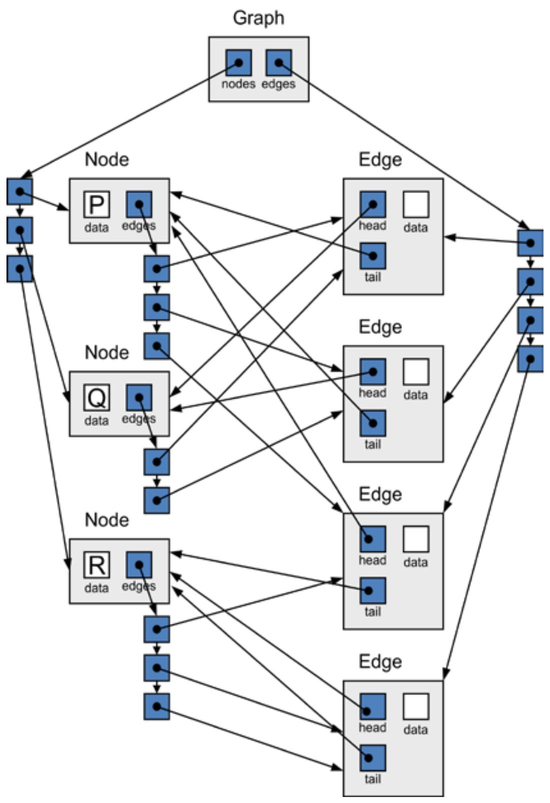
This option returns the same value for two trees that are .equals, but also for other trees that are not .equals. In fact, it is guaranteed to return the same .hashCode for any two trees with the same set of data values in the nodes, even if the trees are different shapes and/or the data values are distributed differently. This can cause hash tables to operate inefficiently, since multiple keys will map to the same slot in the table.

4. **Graphs** (16 points)

Consider the graph implementation used in assignment 10 and the final project. A portion is reproduced below, showing the names of fields but not the methods or javadoc comments. Assume that there is no prohibition in place against duplicate or self edges. Diagram the data structures that would be created in memory to represent the graph shown at right. Show all instances of `Graph`, `Node`, and `Edge`, the fields they contain, and the reference links between them, in the style we have been using in class.

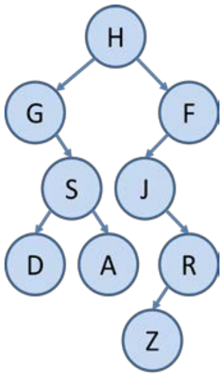


```
public class Graph<V,E> {
  protected LinkedList<Node> nodes;
  protected LinkedList<Edge> edges;

  public class Node {
    private V data;
    private LinkedList<Edge> edges;
  }

  public class Edge {
    private E data;
    private Node head;
    private Node tail;
  }
}
```



5. **Tree Traversal** (8 points)

Consider the unbalanced binary tree at right. In what order would the nodes be visited for the following types of traversals?



a.) Inorder: *GDSAHJZRF*

b.) Preorder: *HGSDAFJRZ*

c.) Breadth-first: *HGFSJDARZ*

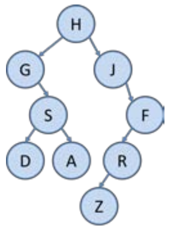
d.) Postorder: *DASGZRJFH*

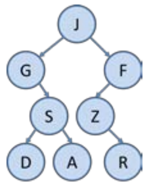
6. **Tree Operations** (8 points)

Starting in each case from the tree shown in the previous question, draw the tree that would result from the following operations:

a.) Right rotation on the subtree rooted at F

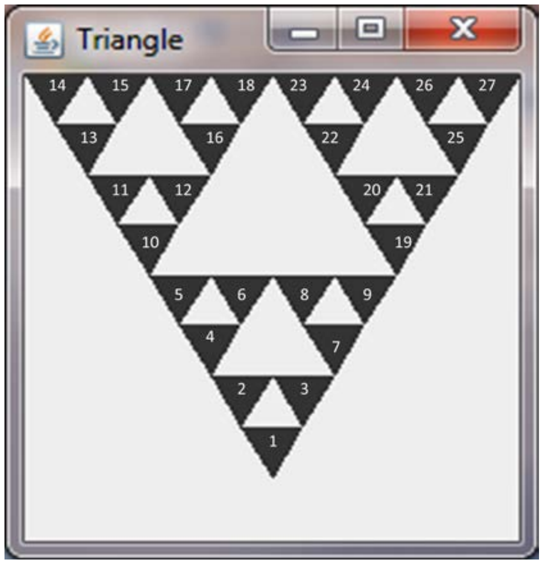
b.) Deletion of H, using copy-left to fill holes.





7. **Recursion** (8 points)

Consider the methods defined below. The figure shown at right was created by a call to paintComponent as shown. There are 27 small triangles that make up the shape. In your answer booklet, label these small triangles from 1 to 27 in the order that they will be painted.



```
    public static Point midpoint(Point p1, Point p2) {
        return new Point((p1.x+p2.x)/2,(p1.y+p2.y)/2);
    }

    public void drawTriangle(Graphics g, int order, Point
top, Point left, Point right) {
        if (order <= 0) {
            int xcoords[] = {top.x, left.x, right.x};
            int ycoords[] = {top.y, left.y, right.y};
            g.fillPolygon(xcoords,ycoords,3);
        } else {
            drawTriangle(g,order-
1,top,midpoint(top,left),midpoint(top,right));
            drawTriangle(g,order-1,midpoint(top,left),left,midpoint(left,right));
            drawTriangle(g,order-1,midpoint(top,right),midpoint(left,right),right);
        }
    }

    public void paintComponent(Graphics g) {
        drawTriangle(g,3, new Point(100,173),new Point(0,0), new Point(200,0));
    }
```

8. **Programming Practice** (12 points)

For each of the following scenarios, decide whether the datum described (in bold) would be best implemented as a local variable, an argument to a method, a class field, a static class field, or a static final constant. If there are several possibilities, pick the one you think is best and justify your conclusion.

a.) A GUI-based program must keep track of the **current interaction mode**, as the event handler methods for several user actions have different behavior depending upon the mode.

This should be a field in the GUI manager class.

b.) The user interface for a program will include a map, which is an image of a known fixed size. The **dimensions** of this image must be stored somewhere so that they can be supplied when needed as the GUI is drawn.

This should be a static final constant.

c.) A method processes a String to create some data structure. In some cases it may be passed a String that has already been partially processed. It needs to know the **index of the beginning of the unprocessed portion of the string**.

This should be an argument to the method.

d.) A method is designed to read an entire file and search for duplicate lines. To do this, it employs a **hash table to keep track of lines it has already seen**. When finished, it returns the number of duplicate lines encountered.

This should be a local variable.

e.) A **BufferedReader** is created that reads from System.in. Several different event handlers must read from this input, which may be redirected from a file.

This should be a static field.

f.) A particular program uses many instances of a particular class, which are modified from time to time. The program must keep a record of **which instance was last to be modified**, so that the change can be reversed later if desired.

This should be a static field.

9. **Programming Methodology** (12 points)

Write a short essay describing what you believe good computer programming is. You should draw on your experiences in labs and on homework assignments, on the lectures, and on your Moodle discussions about programming. Answers will be evaluated for comprehensiveness, thoughtfulness, and writing quality.