CS 361 – Lab #3 – Euler tour traversal and some applications

Today we will experiment with binary tree traversals: inorder, preorder and postorder. Because the code for these traversals is so similar, our text recommends the Euler tour strategy. If we first implement an Euler tour traversal, then very little additional work is required to specify exactly which way we want to visit the vertices in the tree. The Euler tour traversal is also flexible enough to allow for other applications, such as printing out a fully parenthesized arithmetic expression.

1. To begin, in your account create a new directory called lab03. Copy my code from http://cs.furman.edu/~chealy/cs361/lab03/. You will only need to modify Tree.java and Driver.java.

2. Run the driver program. The program should run smoothly. It creates a tree with 7 nodes, and it uses a preorder traversal (implemented as a Java iterator inside the tree's toString() method) to print out the contents of the tree. The program later creates a second tree corresponding to some arithmetic expression, but for now we aren't yet doing anything with this second tree. We'll look at that one later.

3. Next, notice that at the beginning of Driver.java, where we define the nodes of our tree and slowly build the tree bottom-up, there are two alternate definitions of the subtree at Q. Un-comment the line that says Tree q = new Tree(new Item("Q", j, null)); and comment-out the other definition of q. Compile and run the driver program. Uh-oh! Something is wrong with our Tree constructor. Unfortunately, it doesn't handle the case where either child is null. Please make the appropriate corrections to the second constructor in Tree.java. After you have made the changes, make sure the program prints all of the nodes and the correct total number of nodes in the tree.

   Hint: Notice that the initial-value constructor takes 3 parameters, including Tree references to the left and right subtree. You need to add code that checks to see if each of these subtrees is null. If not null, you need to do 3 things: Tell root who its child is, tell the child that its parent is root, and finally update the numNodes of the tree.

4. Now we are ready to attack our iterators. I have provided you an example traversal iterator, one that just does a preorder traversal of the tree. Currently, in Tree.java there is an inner class called PreorderIterator, and then at the bottom you should see corresponding access methods called preorderIterator that can return this iterator object. Refer to the code in the PreorderIterator inner class as you proceed to implement the EulerTourIterator class. In other words, you need certain attributes, constructor and required methods to satisfy the Iterator interface. I have provided a stub class declaration for your EulerTourIterator, and a comment above it giving some hints as to how to start.

Hint: The purpose of the iterator class is to create a linear list of nodes that would be encountered during a traversal. This linear list is an ArrayList attribute called list. Notice that we create an empty list in the constructor. Then, we have a second method called addNext() that will actually put nodes into the list.

5. After you have implemented a skeleton of the EulerTourIterator inner class, you need to turn your attention to carefully implementing its addNext() method. This method is important because it will encapsulate our 3 classic traversals of inorder, preorder and postorder. Because the Euler tour can handle any type of traversal, it is not necessary for us to define further inner classes for these 3 other traversals. All we need to do is define the access methods (given at the bottom of Tree.java). The way I've written these access methods should give you a clue as to a possible implementation for addNext(). Feel free to implement addNext() as you see fit… but one way to do it is to encode the left-action, underneath-action and right-action as strings. See the comment above the inner class for details.

   Hint: Notice that addNext() is recursive. This is because all of the traversals are recursive. It takes 4 parameters: the Node n that we are concerned with, and then 3 strings indicating our left action, below action, and right action. Each of these action strings has 3 possible values: the word "myself", the empty string "", or something else. Each time you consider an action, you need to write an if-else statement encompassing these 3 possibilities. The action "myself" means that the Node n needs to go directly into the list. The action "" means do nothing. And if the action string is anything else, that literal string gets put into a new Node to go into the list.

6. When you are ready to test your traversals, be sure to un-comment the iterator access methods at the bottom of Tree.java. Also, please deprecate (i.e. comment-out) the old PreorderIterator inner class and its two original access methods. In Driver.java, un-comment the code that tests these traversals. Run Driver.java to verify that the 3 traversals of the tree are correct.

7. Finally, let's perform one more application of the Euler tour traversal: printing out a fully parenthesized arithmetic expression. The latter part of Driver.java defines a second tree called total, which is a full binary tree representing a numerical expression. After defining the tree, we print it out using a postorder iterator object. To create parenthesized output, we need to write a similar segment of code underneath that creates an (Euler tour) iterator object and traverses it. This time, you need to think about what information you would pass to the Euler tour inner class so that it knows when to print the parentheses and when to print the number or operator itself. Compile and run your program… Do you know why the program prints parentheses around everything, including single numbers and the entire tree itself?

   _____

   _____