CS 430 Programming Languages
Spring 2017
Module 7 Lab

JAMES MADISON UNIVERSITY

1. This lab is about classes and unit testing in Ruby. Make a new Ruby script called lab7.rb and open an editor window for it. Open another window for executing this script.

2. You will make three classes: two are classes we use to illustrate classes in Ruby, and the third is a class to hold unit tests. Lets set up the unit testing class first. After the comment with your name at the start of the file, put the line `require "test/unit"`. This will make Ruby read files containing the unit testing framework.

3. Now we make a class to hold unit tests. Put in the lines
   ```
   class BST_Test < Test::Unit::TestCase
   end
   ```
   This declares a class BST_Test that is a sub-class of Test::Unit::TestCase (the < sign indicates a sub-class and the double-colons are for modules, which we are not messing with). It is called BST_Test because it will test a binary search tree (BST) class. The end is the delimiter marking the end of the class. We will put test code inside this class. Note that class names must be capitalized in Ruby.

4. A binary search tree needs nodes, so lets start with a Node class. Declare a Node class above the BST_Test class. Node is not an explicit sub-class of any class, so you don't need to use the < symbol (all Ruby classes are implicit sub-classes of Object, however, just like in Java).

5. Class constructors in Ruby are always called initialize(). As you will recall, new binary search tree nodes are always created at the bottom of the tree, so new nodes always have empty left and right sub-trees and the value stored at the node. Hence our Node class initialize() function can take the value at the node as its parameter, store it at the node, and set the left and right sub-tree instance variables to nil (the null pointer in Ruby), thus creating a new node with the correct instance variables. So make a method inside the class with the signature initialize(value).

6. The initialize method needs to set three instance variables. In Ruby, instance variables always start with the character @, but they don't exist until we assign values to them (like other variables in Ruby). So in the body of initialize, set the instance variable @value to value and the instance variables @left and @right to nil. Now when we create a new node with Node.new(v), we will get a new Node instance with its variables set appropriately by initialize.

7. In Ruby, instance variables are always private, so we can't read or write them without writing access methods. This is a pain, so Ruby will do this for us if we ask, making certain instance variables into *attributes*, which are instance variables with accessor methods. All we have to do is include in our class attr_reader (for read-only attributes), attr_writer (for write-only attributes), or attr_accessor (for read-write attributes), followed by the names of the instance variables preceded by a colon. For example: attr_accessor :value, :left, :right will make all our instance variables into readable and writeable attributes. Thus if n is a Node

instance, n.left = t will set n.@left to t, and v = v.value will retrieve the value of n.@value and assign it to v. Make accessors for all the instance variables of the Node class. Put the attr_accessor specification after the initialize method but inside the Node class.

8. Our Node class is done, so lets test it. In the test class, write a method called test_node. Ruby looks for methods whose name begins "test," so we are constrained a little in naming. Make a node with the value "abc" (or whatever you like). We check that the node has the right values in it using *assert* methods. assert_equal(expected, actual) tests that an actual computed value is what is expected. For example, we should have a line in our test method assert_equal("abc", node.value). Write the other assert method calls, save the program, and then run this as a regular Ruby script (that is, type ruby lab7.rb in your execution window). You should get a report showing how the testing went. Fix any problems and rerun the script until all tests pass.

9. Add a BST class to the file. You will recall from data structures that you only really need a root node instance variable in this class, but that it is useful to also have a size instance variable. Make an initialize method that sets up an empty tree. We want the size instance variable to be readable, so make an attribute reader for it.

10. Add a test_BST method to the test class at the end of the file. Make a new tree and assert that its size is 0. Run your tests.

11. Now lets write insert and find methods for the BST class. Lets do this using test-driven development, which means we will write the tests first and then write the code to pass the tests.

12. The first tests we write insert a single value into the tree, test that the tree has size one, that we can find that value, and that we cannot find some other value. The insert method takes one argument, the value to be inserted; the find method takes one value and returns a boolean. (The assert(t) method just checks that t is true.) So write these tests and run the test script, which should fail.

13. Lets write the code to pass the tests. If you recall from data structures, the empty tree is a special case during insertion because the root just refers to the new node holding the inserted value. So write this case and run the tests. The insert method test should pass now.

14. Write the find method. For practice, lets user recursion. The find method takes only one parameter, but a recursive search requires that we keep track of where we are in the tree. Lets make a helper function search(node, value) that searches the sub-tree whose root is node. If node is nil, then the tree is empty so the result is false. Otherwise, we can look for the value at the node, and if we find it, we return true. Otherwise, if the value sought is less than the value at the node, then we call search on the left sub-tree; otherwise we call it on the right sub-tree. We kick off the search by calling search(@root, value) from find. We can make search private by adding the line private :search somewhere in the class after search is defined. Test your code. Fix it until it passes all tests.

15. Now lets add tests for a bigger tree. Add a few more items to the tree and make an assertion about the size of the tree. Make assertions about values in and not in the tree. Try

to add values to make the tree an interesting and challenging shape to stress your algorithm (which you have not written yet). Run your tests, which will fail.

16. Recall that in a BST, values are inserted at the bottom of the tree after searching down the tree as if looking for the value to be inserted—where the search would end in failure at a `nil` child, the insertion algorithm adds a new node with the inserted value. Remember to adjust the `size` attribute. And write your code so that duplicate values are not inserted. Write your code and test it, fixing it until it works.

17. We could have put the test code in a different file (and usually we would), and include the file with the tested code at the top using `require` the way we included the unit testing framework code. If a required filel is in the same directory as the file we are writing, then we put `require_relative "filename.rb";` this avoids having to specify the entire path to the file. Alternatively, we could comment out the testing code so it won't interfere with using the tested code. The easiest way to comment out a block of code in Ruby is the insert a line before the block with `=begin` in the first column, and a line after the block with `=end` in the first column.

18. We could write more methods, but at this point you should be able to write classes and tests, so you are done.