Computer Science 62
Lab 7

Wednesday, March 11, 2015

Introduction

In this lab, we will experiment with Bailey's `BinaryTree` class. Begin looking at the documentation for the `BinaryTree` class[1]. For today, pay special attention to the constructors and these methods:

```
value()                              isEmpty()
left()                               iterator()
right()                              toString()
setLeft(BinaryTree<E> newLeft)       height()
setRight(BinaryTree<E> newRight)
```

We will use this class to construct some examples of binary *search* trees whose values are integers. For a binary search tree the value contained in a node n is

- greater than all the values contained in nodes of the left subtree rooted at n, and
- less than or equal to all the values contained in nodes of the right subtree rooted at n.

By saying "less than or equal to," we are implicitly allowing duplicate values. It is a decision that can be changed easily.

Notice that Bailey has a class called `BinarySearchTree` which automates most of the operations on binary search trees. We will not use it today because we want you to experience the joy of manipulating trees directly.

A simple example

Draw by hand a binary search tree with 2 at the root and values 1, 3, and 4 in the subtrees. How many different tree structures are possible?

Next, start Eclipse and create a new project for this laboratory (note that you will need to include the BAILEY variable for this lab). Create a class called `Lab07`. Inside this class, write a method

[1] You can find the documentation for the `BinaryTree` class under the "Bailey materials" heading on the "Documentation and Handouts" page on the course website

to construct the tree you just drew and print it out using the `toString` method. You can call this method, `constructSimpleTree`.

Recall that there are three constructors for the `BinaryTree<E>` class: no data, one data item, and one data item with two children. If you need a BinaryTree with only one child (with a data value) you'll need to specify the other child as an empty tree.

Notice how the `toString` methods prints out the trees and in particular, the empty trees.

Inserting values

A more realistic exercise is to start with an empty tree and insert many new nodes. Write a method `insert` that takes a tree and a new value, and inserts a node with that value into the tree. The method returns the new binary tree (We need to return a tree to handle the case when we construct a new tree.)

```
public static BinaryTree<Integer> insert(BinaryTree<Integer> tree,
                                         Integer value)
```

There are two cases. If the original tree is empty, then we simply create a new one-element tree and return it, effectively discarding the old tree. If the original tree is not empty, then we navigate through it until we get to an empty subtree and replace that empty subtree with a one-element tree. We return the original, recently modified, tree.

Test your `insert` method by writing a different method that (1) creates an empty tree, (2) inserts the integers 2, 4, 1, and 3 (in that order) by calling `insert`, and (3) prints the resulting tree. What happens if you insert in a different order, for example, 2, 3, 1, 4 or 2, 1, 3, 4?

Bigger trees

Now that everything is working, write a method `biggerTrees` that inserts 128 random integers. Printing such a tree would not be very enlightening. Instead, use the `iterator` method of the `BinaryTree<E>` class and print the sequence of values generated; they should appear in non-decreasing order. The `java.util.Random` class will be useful for this.

Tree heights

Finally, conduct some experiments on heights of trees. Create a method called `randomTreeHeights`. Inside this method, you should determine the heights of several randomly-constructed 128-node trees and see how they compare. What is the theoretical minimum height? Do your trees ever come close? Can you artificially create a tree with the theoretical *maximum* height?

Calculate the average height over 100 random trees. Is the average height closer to the minimum or maximum?

Write the answer to these questions in the JavaDoc comments for the `randomTreeHeights` method.

What to hand in

You should export your Eclipse project, rename the folder appropriately, and then drag it into the dropbox. If you are working with a partner, please make sure that both names are in the JavaDoc comments at the top of the `Lab07` class.

Philosophical reflection.

The implementation of the `insert` method reveals a serious shortcoming of the `BinaryTree` class in this context. The object-oriented paradigm tells us that we ought to be able to insert an element into a tree object by modifying the internal state of the object. However, it is not obvious how to change an empty `BinaryTree` into a non-empty one. Therefore, we end up writing an assignment statement—and changing a reference instead of an object: `tree = insert(tree, value);` instead of just `insert(tree, value);` Later, we will discuss ways to overcome the defect and to make the insertion process more uniform.

One CAN transform an empty tree into a non-empty one: set the value to a non-null object, and create new empty trees for the subtrees. However, that requires one to have distinct empty trees everywhere. Bailey's original (non-generic) device of using only one empty tree won't work, and his compromise of using the same tree for empty left and right subtrees won't work either.