Lab Report 07

Nathan Bickel

Problem

The problem was to, given a file containing descriptions of certain fruits, create a binary search tree that contains representations of these fruits. The fruits were to be sorted first by weight and then by alphabetical order as a tiebreaker. There was an example dialogue as a guide for how the program should respond to the user and what steps it should do.

Solution

The solution used three classes to accomplish the goal:

The first class was a Fruit class with attributes type and weight. type is a String representing the name of the fruit, and it could take values of “apple”, “orange”, “banana”, “kiwi”, or “tomato”; these values were represented in an array and the setter checked to see if the parameter string matched any of the values in the array. weight is a double representing the weight of the fruit, and it was restricted to values strictly greater than zero. The constructors called the setter methods, and if no values were input or if they were invalid the default values of “apple” and 1.0 were used for type and weight. The **toString** method returned a String with the format “Type: <type> Weight: <weight>”. The **compareTo** took in a Fruit other and compared it to the current Fruit (this); it returned 1 if the weight of this was greater than the weight of other (or if they were equal and the type of this was second alphabetically), -1 if the weight of this was less than the weight of other (or if they were equal and the type of this was first alphabetically), or 0 if both type and weight were equal.

The second class was a LinkedBST class, where the functionality of a generic binary search tree was implemented with type T that implements the interface Comparable. It had an internal class called Node where the data T could be combined with links to other Nodes. Each Node has a leftChild, which must be smaller than the parent (using the **compareTo** method from the Comparable interface), and a rightChild, which must be greater. The LinkedBST class had one attribute, a Node called root, that contained links to all other Nodes in the tree. Its **add** method either sets the root to a new Node containing the data, or recursively calls the private version of the **add** method on the root and subsequent children until it gets to a null child (a leaf) and adds it there. Its

printPreOrder prints out the given Node, recursively calls itself on the Node's leftChild, and then recursively calls itself on the Node's right child. Its **printPostOrder** method recursively calls itself on the Node's leftChild, recursively calls itself on the Node's right child, and then prints out the given Node. Finally, its **printInOrder** method recursively calls itself on the left child, prints out the given Node, and then recursively calls itself on the right child. Thus, this method has the effect of printing every value from smallest to greatest, since the leftmost value on the tree will be the smallest and the rightmost will be the largest. Its **search** method uses recursion in a similar way and the properties of the tree to iterate through until either a match is found and true is returned, or a leaf is reached and false is returned. Its **remove** method uses a private method called **findMinInTree.** This method uses recursion to start at the root and keep going to the left until the node's left child is null, which means the smallest value has been found, and then return that node. The **remove** method uses recursion in much the same way as the other methods. First, it searches through the tree until it finds the Node that needs to be removed. Then, if the node has no children, it is set to null, and if it has only one child, that node is simply replaced with the child. However, if it has two children, the node is switched with the smallest value in the right subtree (using the **findMinInTree** method) and then the node is removed. In this way, the structure of the tree is preserved.

The third class, FruitTreeTester, was the front end of the LinkedBST. It had a LinkedBST of type Fruit called fruitTree, a LinkedList of type Fruit called fruitList, a Scanner to take user input called keyboard, and a Scanner to read in from a file called fileReader. The class's **constructFileReader** method asks the user for a text file name, and continues to ask them until the file could be found or they enter "quit". Then, fileReader was constructed with this name. Its **populateTree** method takes in each line from the text file, splits it into a tab-delimited array, checks if it followed the format, and then adds it to fruitTree and fruitList. Its **removeRandomFruit** method takes a random value from fruitList, prints it out, removes it from fruitTree as well, and then prints out what's left of the tree in order. The **main** method calls **constructFileReader**, **populateTree**, fruitTree's **printInOrder**, **printPreOrder,** and **printPostOrder** methods, and finally **removeRandomFruit**. In this way, the example dialogue was essentially matched.

Implementation Problems Encountered

I wasn't sure whether to use the toString method when printing out Fruits (since doing so wouldn't match the example dialogue), but I used it anyway. I also didn't know if I should worry about handling duplicates, but since the weights are reported with such precision, it seemed very unlikely that there would be any exact matches so I neglected to handle this edge case. Other than that, I didn't really have any problems.

Lab Report Questions

1. A self-balancing tree makes the necessary rotations if there are significantly more or fewer Nodes on the right side of the root than the left, while a non-self-balancing tree does not make these rotations.
2. In a balanced binary search tree, the Big O complexity for searching is $\log_2 n$, because only a fraction of values need to be checked as one knows whether to go right or left at any node by comparing the value being searched for to the node.
3. In a non-balanced binary search tree, the Big O complexity for search is n, because the worst case scenario is that either every Node will only have a right Child or every Node will only have a left child. In this case, the tree is functionally the same as a linked list, and every value needs to be checked.