BST-AVL Tree Checker Documentation

# Description

There were two parts to this assignment. Part one was, given an AVL Tree class with methods for rotation but nothing else, build a valid AVL tree, detecting imbalances with each insertion, calling the proper rotation method, and reporting it to standard output. Additional requirements included reading AVL node data from a text file (not given) and creating a Book class for the data and an AVL Node class to wrap it in. Null nodes were considered to have a height of -1.

I added the method readBookRecords to read from the input file and modified AVLTree’s insert method to call the proper rotation method if an imbalance was detected. When the main method is called, an AVLTree object is instantiated and an ArrayList of AVLNodes is made from the return value of a call to readBookRecords. All nodes in the ArrayList are then added to the root of the AVLTree object.

The second part involved creating a random binary tree from twenty-five randomly generated ISBN numbers and validating if it was a binary search tree. If it was, validate that it was also an AVL tree. Fixing the tree was not required, only reporting the problems (e.g. The random binary tree is/is not a BST/AVL tree).

I created a BinaryNode class that houses an ISBN key and left child, right child, and parent references. These nodes are also comparable to each other by implementing the Comparable interface. I also made a BinaryTree class with an insert method that places a node in a random null available node and a bstChecker method. For testing, I also made a method that inserts nodes in BST order. When the main method in this class is called, a random list of 25 ints are made and inserted into a BinaryTree as BinaryNodes. If the tree is not a binary search tree, it reports this as well as how many instances of incorrect parent-child relationships exist within the tree. Otherwise, the tree is checked if it’s an AVL tree by calling a static method I made in the AVLTree class called avlTreeChecker that works with BinaryNodes instead of AVLNodes. If the tree is not an AVL tree, this is reported, along with the number of unbalanced heights the tree has. If both checks are passed, this is reported.

# Input

Two input text files are included. The first, “book data.txt,” contains a list of 25 actual ISBNs, titles, and authors. Since the ISBNs required the long data type to store them, “book data 2.txt” has the ISBN portion replaced with ints that correspond to that entry’s actual ISBN sort order.

# Output

All output goes to standard output, including imbalance reports for part one and tree validation reports for part two.

# Specific Method Explanations

## BinaryTree

insert

This method handles three case types: an null tree, a tree with a single node, and all other cases. The last is the most complicated to implement a randomized insert among all possible insertion points in the tree. Null nodes themselves cannot be stored, so their parents must be used instead. First, all nodes with null child nodes are identified and stored in an ArrayList with the helper method parentsOfNullsFinder (explanation below). Only finding leaves ignores valid cases of a leaf whose parent’s other child is null. To differentiate between parents with an null left node, an null right node, and two null child nodes, two ArrayLists are used (in hindsight, I should’ve made an class instead). The first ArrayList simply stores BinaryNodes, while the second stores Boolean values whose indices correspond to the first array. A false value means the left child is null and a true value means the right child is null. If both are null, the node is added twice in the first list. With all possible insertion points stored, a random null node is chosen and the node given in the method parameters is placed there.

parentsOfNullsFinder

This is a recursive void method that modifies an ArrayList of BinaryNodes given in the method parameters. A BinaryNode is the second parameter. If the current node (second parameter) is not already in the ArrayList storing parents (first parameter), then it checks if either the left or right child is null, adding it to the list if it is. Then if the opposite child (left if right, right if left) is not null, the method calls itself again for that child. If both children are not null, the method calls itself for both. The first call should start with the root as the second parameter.