COMPUTER SCIENCE 21A (SUMMER TERM, 2017) DATA STRUCTURES

PROGRAMMING ASSIGNMENT 2

Due Thursday, July 27 @ 11:30pm

VERY IMPORTANT

- Your code should be well commented:
  - Add your name and email address at the beginning of each .java file.
  - Write comments within your code when needed.
    - ✓ You must use Javadoc comments for your classes and methods.
  - In each method's Javadoc header, include its running time.
- Before submitting your assignment you **must** zip all your files.
  - The ZIP file must have the naming convention LastnameFirstname-PA2.zip
- Submit your assignment on LATTE using the assignment submission link.
- You may only use data structures that you implement yourself.
  - **You may not use arrays.**
  - Your data structure should not have hardcoded data types. You should use generics.

The Secret Message

You are a secret agent working for the infamous Brandeis Spy Network.
You've received a secret message from your leader, *the Professor*, in the format of a series of insert statements into an AVL binary search tree.

Your mission is to implement a structure that will perform the specified insertions, so that you can print the postorder traversal of the final, balanced tree, and read the secret message.

You must implement an AVL Binary Search Tree in which each node stores a value. The structure of the tree should be determined by the natural ordering of the values. Natural ordering is when a series of elements are sorted automatically according to their compareTo( ) method. Thus, your data structure should be of type: BinarySearchTree<E **extends** Comparable<E>>.

The insertions in the secret message will be of type DataElement<String>, but you should use generics so that a user can instantiate your tree with any type that extends Comparable.

Besides implementing the following methods, implement any helper methods necessary for your program.

hasLeft( ): returns true if the tree has a left child.

hasRight( ): returns true is the tree has a right child.

isLeaf( ): returns true if the tree is a leaf (has no children).

isEmpty( ): returns true if the tree is empty (doesn't have a root).

isRoot( ): returns true if the tree has no parent.

isLeftChild( ): returns true if the tree is a left child of its parent.

isRightChild( ): returns true if the tree is a right child of its parent.

hasParent( ): returns true if the tree has a parent.

findNode(E e): returns the node that contains e, or null.

findMin( ): returns the node with the minimum value in the tree.

findSuccessor( ): returns the node that is the successor to the current node.

addRoot(E e): add a root containing e to the tree if the tree is empty (throw exception if it isn't).

insert(E e): Starting from the root, walk down the tree, searching for the correct location for e. If the tree already contains a node in that location, throw an exception. Otherwise, insert a node containing into the tree. Remember to preserve the balanced binary search tree property.

search(E e): Returns the node containing e from the tree. Returns null if nothing is stored at that location.

delete(E e): Removes the node containing e from the tree. Throws an exception if there is no node containing e. (NOTE: adjusting the tree to preserve the balanced binary search tree property is extra credit). You may wish to write one or more auxiliary methods. If so, describe the choices you made in comments or in a readme file.

size( ): returns an int that is the number of nodes currently in the tree. Do not use a field, this must be a recursive method.

balanceFactor( ): Each Tree in a binary search tree has a balance factor, which is equal to the height of its left subtree minus the height of its right subtree. A binary tree is balanced if every balance factor is either 0, 1, or -1. Write a method that calculates the balance factor of your tree.

height( ): return the height of the tree. Your method should be recursive.

depth( ): return the depth of the node. Your method should be recursive.

balance( ): write a method that balances this binary search tree using the algorithm described in class. You may wish to write one or more auxiliary methods. If so, describe the choices you made in the comments or in a readme file. Note: remember that keeping the tree unbalanced, and then attempting to balance it after multiple insertions or deletions will not work.

rightRotation( ): rotates the tree to the right around the calling node.

leftRotation( ): rotates the tree to the left around the calling node.

postorder(): returns the String generated from a postorder traversal.