ADS1 Assignment 2

Binary search tree with a rebalance option

In this assignment you will implement a binary search tree with one special operation: rebalance.

Before you start coding consider the ADT and make sure you know exactly what to do. You are going to make a generic tree, that is a tree that can store any kind of type (but for the BST it must implement the comparable interface). The traversals return an ArrayList of the objects in the tree, you don't need to return an iterator. You don't need to support more than one occurrence of each element either.

It's recommended to build the tree in phases:

1. A binary tree
2. A binary search tree with add and remove
3. A binary search tree with rebalance (not to be mistaken with a self-balancing tree such as an AVL or red-black tree)

Test driven development is well suited for the assignment since it can be broken into smaller bits that can be test on their own. Besides, **it is a requirement that all parts of the code are well tested**. When you encounter errors, it can be useful to see a graphical representation of the tree. To that end I have uploaded the source code for a simple ascii code printout of a tree, but you are welcome to make a nicer graphical presentation (it is not a requirement).

Remember that in phase one there are no add or delete methods. This is because a general binary tree does not have a specific ordering. In order to create a tree, you must manually do it by creating and combining nodes.

You are welcome to work in groups of three to four students but you must hand in individually and indicate in the code who you worked with.

The assignment must be handed in no later than March 24 at 23.55.

Feedback:

- Feedback will be peer assessment. In practice, this means that you will give feedback on two of your fellow students' work and receive feedback from two of your fellow students.
- Remember to be constructive.
- Feedback must be given no later than March 31 at 23.55.

Class BinaryTreeNode ADT:

Operation	Description
void setElement(E element)	Store the element in the Node
E element getElement()	Returns the element from the Node
void addLeftChild(BinaryTreeNode)	Add a left child to the Node
void addRightChild(BinaryTreeNode)	Add a right child to the Node
BinaryTreeNode getLeftChild()	Returns a reference to the left child or null if there is no left child
BinaryTreeNode getRightChild()	Returns a reference to the right child or null if there is no right child

Class BinaryTree ADT:

Operation	Description
BinaryTreeNode getRoot()	Returns a reference to the root or null if tree is empty
void setRoot(BinaryTreeNode)	Set the root of the tree
boolean isEmpty()	Determines whether the tree is empty
int size()	Returns the number of elements in the tree
boolean contains(E element)	Determines if an element is present in the tree
ArrayList <E> inOrder()	Returns a inOrder representation of the tree or null if the tree is empty
ArrayList <E> preorder()	Returns a preOrder representation of the tree or null if the tree is empty
ArrayList <E> postOrder()	Returns a postOrder representation of the tree or null if the tree is empty
ArrayList <E> levelOrder()	Returns a level Order representation of the tree or null if the tree is empty
int height()	Returns the height of the tree or -1 if the tree is empty

Class BinarySearchTree ADT (extends Binary Tree):

Operation	Description
boolean insert(E element)	Add an element to the tree. Return true if successfully inserted, false if already present
boolean removeElement(E element)	Remove an element from the tree Return true if successfully removed, false if not present
E element findMin()	Returns the minimum element of the tree
E element findMax()	Returns the maximum element of the tree
boolean contains(Element)	Determines if an element is present in the tree Return true if present else false.
void rebalance()	Rebalance the entire tree, the outcome must be a balanced tree.