### Portfolio Function Title: BinarySearchTree

Version number: 1

### Function Description:

Allows for a user to build a binary search tree of generic objects of comparable type E.

Public method overview:

Can return the data of the root Node of the binary search tree.

Can check if an element comparable type E is in the tree.

Can insert an element of comparable type E into the tree.

Removes an element of comparable type E from the tree.

Can check if the tree is empty.

Can get the number of elements in the tree.

Gets the maximum height of the tree.

Checks if the tree is balanced- the minimum height and maximum heights differ by 1.

Traveses the tree in order of comparable type E and returns a queue of the elements in the tree in order.

Traverses the tree in level order and returns a queue of the elements in order by level (1st level is 1st).

Traverses the tree from node, then left, then the right and returns a queue of the elements in the order they were traversed.

Traverses the tree from left, then right, then the node and returns a queue of the elements in the order they were traversed.

### Author:

Nathan Mankovich

### Date Written/Last Modified:

05-05-2015 / 05-07-2015

### Design overview:

The binary search tree stores elements in nodes (in the node class) that have a data field for elements of generic comparable type E, a left pointer to the left child and a right pointer to the right child.

The tree is ordered according to the compareTo class of the elements of generic comparable type E. If an element is “smaller” than the current node’s data it goes to the left child. If the element is “bigger” than the current node’s data it goes to the right child. If the element is the same as the current node’s data it is not added to the binary search tree. Thus, this binary tree does not accept elements that are duplicates of one another according to their compareTo class.

The Remove method basically traverses the binary tree twice. The first time it traverses the tree it find the node that contains the element to be removed. Then, there are three possible cases: the node containing the element to be deleted has no children, one child or two children. If the node has no children, the node’s parent is pointed to null. If the node has two children, its parent node now points to the smallest right child of the removed node. If the node has one child, its child takes its place.

The CheckSize method returns a public variable that keeps track of the number of elements in the tree by incrementing (+1) every time an element is added to the tree and decrementing (-1) every time an element is removed from the tree.

The GetHeight method calls a recursive helper method that goes through the tree and ultimately returns the maximum number of edges in the tree. If the tree is empty or only has one node, the GetHeight method will return 0.

The CheckIfEmpty method checks to see if the root of the tree is null. If it is, the method returns a true Boolean. Otherwise, the method returns a false Boolean.

The IsBalanced uses two recursive helper methods to find the max and min number of edges in the tree. If these two integers differ by more than 1, the method returns a false boolean. Otherwise, the method returns a true boolean.

The tree can be traversed four different ways and return queues of the elements of type E in the tree in four different orders. The first way, TravTreeIn uses a recursive helper method to traverse the tree in order by going to the left child, the node, then the right child. TravTreeLevel does not use a recursive helper method. It just uses loops to traverse the tree by level starting at the root and working its way down. TravTreePre uses a recursive helper method like TravTreeIn but goes node, left child, the right child. The last tree traverse is the method TravTreePost which uses a recursive helper method that goes from left child to right child then to the node.

### Feature Specifications:

CheckIfEmpty checks if the tree is empty by looking at the root of the tree.

Contains checks if the tree contains the input element by using loops to traverse the tree.

Insert inserts an element of type E into the tree either by calling the recursive method InsertNode or inserting the element at the root.

Remove removes the node with data field that is the element from the tree.

GetRoot returns the data at the root node.

GetSize returns the size of the tree.

GetHeight gets the height of the tree as the largest number of edges by calling the getHeightNode method.

IsBalanced determines if the tree is balanced by getting the maximum and minimum heights of the tree in number of edges by calling the getHeightNode and getMinHeightNode method.

TravTreeIn calls the TravTreeInNode method to traverse the tree in order.

TravTreeLevel does a level traversal of the tree and returns a queue of strings of the elements of the tree in level order.

TravTreePre calls the TravTreePreNode method to traverse the tree in pre order.

TravTreePost calls the TravTreePostNode method to traverse the tree in post order.

Nodes are created in the Node class.

### Programmer User Interface:

User Interface: Java code and command line only.

### Input and Output Requirements and Restrictions:

Public methods:

* GetRoot
  + Input: none
  + Output: the element of type E that is the root of the binary search tree
* Contains
  + Input: an element of comparable type E
  + Output: Boolean that is true if the element is in the tree
* Insert
  + Input: element of comparable type E
  + Output: none
* Remove:
  + Input: element of comparable type
  + Output: none
* CheckIfEmpty
  + Input: nonde
  + Output: Boolean that is true if the tree is empty
* GetSize
  + Input: none
  + Output: integer that is the number of elements in the tree
* GetHeight
  + Input: none
  + Output: integer that is the maximum height of the tree
* IsBalanced
  + Input: none
  + Output: Boolean if the tree is balanced- the minimum height and maximum heights differ by 1
* TravTreeIn
  + Input: none
  + Output: queue of the elements in the tree in order.
* TravTreeLevel
  + Input: none
  + Output: queue of the elements in order by level (1st level is 1st).
* TravTreePre
  + Input: none
  + Output: queue of the elements in the order they were traversed: from node, then left, then the right
* TravTreePost
  + Input: none
  + Output: queue of the elements in the order they were traversed: from left, then right, then the node

Private Methods (helper recursive methods):

* GetHeightNode
  + Input: the current node
  + Output: the max height (# edges)
* GetMinHeightNode
  + Input: the current node
  + Output: the height
* TravTreeNodeIn
  + Input: the current node and the queue as it gets filled with the elements in order
  + Output: the final queue with all the elements of the tree in order
* TravTreeNodePre
  + Input: the current node and the queue as it gets filled with the elements in order they were traversed: from node, then left, then the right
  + Output: the final queue with all the elements of the tree in order they were traversed: from node, then left, then the right
* TravTreeNodePost
  + Input: the current node and the queue as it gets filled with the elements in order they were traversed: from left, then right, then the node
  + Output: the final queue with all the elements of the tree in order they were traversed: from left, then right, then the node
* InsertNode (in the node class)
  + Input: the current node
  + Output: none

### Assumptions and Dependencies:

The Binary Search Tree assumes that no duplicate objects (according to the object’s compareTo method) are added to the tree.

Also whenever a user calls a method that inputs an element into the tree they must input an element of the type that the tree was declared for. For example: if the binary search tree was instantiated with elements of type String then the tree can only be filled with Strings.

It also assumes that users will only want to traverse the entire tree every time they call a traverse tree method.

### Known problems and limitations:

The tree only returns queues from the traverse tree methods which limits the user to traversing the whole tree every time the any tree traversal method is called. This may be useful. For example, if the user wants all the elements in the tree in order the user can call the TravTreeIn() method and get a queue with the front of the queue with all the elements from the tree in order from head to tail.

Since there is not a balance tree method, the tree can be built so that it is almost a linked list where every element only has one right or left child. So, if the tree gets out of balance the benefits of a fast element access time are greatly decreased.

### Use Cases:

This binary search tree is best used as a data structure for elements that are input in a “random” order. For example, if the input elements are input such that each element is “larger” than the previous element according to the element’s compareTo() method, this data structure will just function like a linked list.

Also, this is best used for storing data and retrieving all the data again in one of the few traversal types because all the traversal methods (ex: TravTreeIn) return a queue of the data.

This binary search tree can be used to store data that needs to be accessed and sorted without much work on the “front end”.

### Testing Methodology:

Be sure to put the tBinarySearchTree.java and the portfolio.jar file in the same folder.

To compile:

javac -cp .;portfolio.jar BinarySearchTree.java

To run:

java -cp .;portfolio.jar tStack > LogBinSchTreeTest.txt

All command line outputs will be in the new LogStackTest.txt file in the folder that contains the tBinarySearchTree and the portfolio.jar files.

### Modification history:

Version 1: 05-07-2015

### Design detail and/or Diagrams:

When the program starts, an empty binary search tree is created with a null root node and a 0 integer length value. There is an inner node class that contains the public variables: data (the element of comparable type E), left Node (left child pointer) and right Node (right child pointer). There is one recursive method InsertNode in the Node class that is a private recursive method which helps the Insert method insert a new donor. All other methods are in the BinarySearchTree class and outside the Node class. Any methods that are implemented using recursion utilize helper methods to recurse in order to have input parameters of type Node.