### Portfolio Function Title: BinarySearchTree

Version number: 1

### Function Description:

Allows for a user to build a linked list of comparable objects of the same type. Users can add object to the tree and can access the all objects in four different ordered queues. They can also check the number or objects. They can also access the maximum height. Users can check if there is a certain object in the tree. Users can remove a certain object from the tree and get the root object of the tree. The users can also check if the tree is empty or balanced.

### Author:

Nathan Mankovich

### Date Written/Last Modified:

04-27-2015 / 04-30-2015

### How to use the Function:

To use the BinarySearchTree program just instantiate a BinarySearchTree<obj> with an object of your choice and then you can use the different methods in your java code.

The program is stored in the portfolio.jar file, put that file in the same folder as YourProgram.java.

Open a Windows command window and enter:

To compile the program with YourProgram.java enter:

javac -cp .;portfolio.jar YourProgram.java

pause

Then, to run the program with the provided sample-mailing-list.csv enter:

java -cp .;portfolio.jar YourProgram

pause

User Interface: Java code and command line only.

### How the Function works:

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 left child is always less than node pointing to it. The right child is always greater than the node pointing to it.

### Supported Methods (including Inputs, Outputs, Features and Results by method):

* GetRoot gets the root of the tree.
  + Input: none
  + Output: the element that is the root of the binary search tree
* Contains checks if a certain element is in the tree
  + Input: an element
  + Output: Boolean that is true if the element is in the tree
* Insert inserts an element in the tree
  + Input: element
  + Output: none
* Remove removes an element from the tree
  + Input: element
  + Output: none
* CheckIfEmpty checks if the tree is empty
  + Input: nonde
  + Output: Boolean that is true if the tree is empty
* GetSize gets the number of elements in the tree
  + Input: none
  + Output: integer that is the number of elements in the tree
* GetHeight gets the maximum number of edges from the root to the furthest node
  + Input: none
  + Output: integer that is the maximum height of the tree
* IsBalanced checks if the minimum height and maximum heights differ by 1 (aka. Balanced)
  + Input: none
  + Output: Boolean if the tree is balanced
* TravTreeIn traverses the tree in order
  + Input: none
  + Output: queue of the elements in the tree in order.
* TravTreeLevel traverses the tree in level-order
  + Input: none
  + Output: queue of the elements in order by level (1st level is 1st).
* TravTreePre traverses the tree in pre-order
  + Input: none
  + Output: queue of the elements in the order they were traversed: from node, then left, then the right
* TravTreePost traverses the tree in post-order
  + Input: none
  + Output: queue of the elements in the order they were traversed: from left, then right, then the node

### Known problems and limitations:

A user cannot add elements that return a zero when using the compareTo() class on the elements. For example: if one wanted to make a binary search tree and tried to add two “Joe” strings, only one Joe string would be added because Joe.compareTo(Joe) evaluates to zero.

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 all the data of the instantiated type in the tree.

Since there is no-auto balancing or balance tree methods, the elements must be input in a “random order.” If they are in perfect ascending or descending order according to the element type’s compareTo() method, the tree might as well be replaced with a LinkedList.