COMPUTER SCIENCE 21A (SUMMER, 2015)
DATA STRUCTURES AND ALGORITHMS

PROGRAMMING ASSIGNMENT 2

Due Fri, June 26 @ 11:55pm.

REMEMBER

- Your code should be well commented:
  - Add your name and email address at the beginning of each .java file.
  - Write a clear description of each class.
  - Write comments within your code when needed.
- Before submitting your assignment, zip all your files.
- To submit, upload your zip file on Latte.

A. Implement an AVL Binary Search tree in which each node stores both a key and a value. Call the class **BinarySearchTree**. Keys are Strings and Values are Objects. Besides implementing the following methods, implement any helper method necessary for your program. Don't forget to write (and submit) the test file as well.

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

hasRight( ): returns true if 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(String key): returns the node with the given key.

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

findSuccessor(String key): returns the node that is the successor to this node in the tree.

addRoot(String key, Object value): add a root to the tree if the tree is empty (throw exception if it isn't).

insert (String key, Object value): Starting from the root, walk down the tree, searching for the correct location for *key*. If the tree already contains a node with *key* as its key, throw an exception. Otherwise, insert a node into the tree with *key* and *value* as its key and value. Remember to preserve the balanced binary search tree property.

search(String key): Returns the Object stored in the node with *key* as its key, and returns null if the tree does not have a node with that key.

delete(String key): Removes the node containing *key* from the tree. Throws an exception if there is no node with *key*. (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, the number of nodes currently in the tree.

balanceFactor( ): Each Tree in a binary search tree has a balance factor, which is equal to the height of the 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.

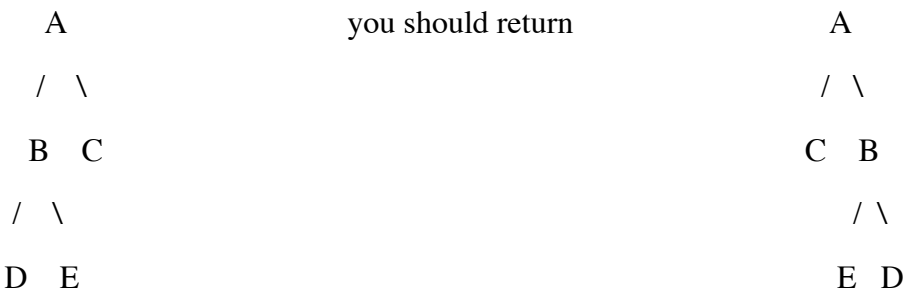
depth(): return the depth of the node. Your method should be recursive.

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 comments or in a readme file. Note: Remember keeping the tree totally unbalanced (as you are inserting and deleting nodes) and then try to fix it afterwards won't work.

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

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

mirrorTree(): return a new tree which looks like the mirror image of the given tree (your method should be recursive) for example:



printInorder(): prints the tree using an inorder traversal.

B. Implement a class called HashTable. It should also store key-value pairs with Strings as keys and Objects as values. Use chaining to handle collisions, but instead of using a linked list as buckets, use the binary search tree you created in part 1 **(*note that this is not a typically used strategy in the real world*)**.
Implement the following methods:

HashTable(int size): This constructor creates a hash table whose array length is equal to *size*.

hash(String string): Perform some hash function on *string* and return an int. Don't use the same function that we implement in class; make up your own. Make sure that the range of your hash function matches the size of the hash table's underlying array.

put(String key, Object value): Perform the hash function on *key*, and add a record containing *key* and *value* at the position in the array indicated by the result of the hash. Return true if the pair is successfully added and false if it is not.

get(String key): Returns the Object that is associated with *key*, and null if none exists.

hasKey(String key): Returns true if the table is currently storing a record with *key* as its key, and false if not.

remove(String key): Removes the key-value pair associated with *key* from the table.

size( ): Returns an int, the total number of key – value entries currently in the hash table.

distribution( ): Returns an int[ ] whose length is the same as the length of the hash table. The value of the *n*th cell in this array should be equal to the number of entries stored in the bucket corresponding to the *n*th position in the hash table.

keys( ): Return a String[ ] containing all Strings that are keys currently associated with Objects in the hash table. You will likely need to add a method to your BinarySearchTree

class to accomplish this.

Play around with several different hash functions, and use the **distribution( )** method to evaluate how well they do at spreading values out evenly across the hash table. Also, experiment with different sizes for the hash table (try both prime and non-prime values) to see what effect it has. Write a short description of your results in a readme file, and include it with your code.