Georgia Gwinnett College
School of Science and Technology
ITEC 3150: Advanced Programming
Homework Assignment 7

In this assignment you are to implement BST, a generic Binary Search Tree class. You are provided with two files: **BSTMapNode.java** which contains the node class BSTMapNode for implementing BST, and **BSTMap.java** which contains the BSTMap class whose methods you are required to complete. Both classes are generic, where type parameter K is the generic type for the key and type parameter V is the generic type for the value. Two other files are also provided for GUI interface.

Comparable

Since a binary search tree stores data by key and its methods need to compare the keys of data, the requirement for the generic type K is that it implements the Comparable interface, like Integer and String, as specified in the definitions of the classes. Recall that to compare two objects $key1$ and $key2$ of type K that implements Comparable, you may call the method $key1.compareTo(key2)$, and there are three possible outcomes:

- $key1.compareTo(key2) = 0$ means $key1 = key2$
- $key1.compareTo(key2) < 0$ means $key1 < key2$
- $key1.compareTo(key2) > 0$ means $key1 > key2$

BSTMapNode

BSTMapNode is the node class for implementing the BST class. It contains the following instance variables as well as their getters and setters:

- key – the key stored at the node
- $value$ – the value associated with key
- $left$ – the BSTNode object representing the left child of the node
- $right$ – the BSTNode object representing the right child of the node
- $height$ – the height of the node in the tree
- $balanceFactor$ – the balance factor of the node (see Slide 30 of the BST lectures)
- $size$ – the size of the subtree rooted at the node

It is very similar to the SimpleBSTNode class you saw in the lectures. Other than the generics, the only thing new is the $size$ instance variable, which is initialized to 1 when a new BSTMapNode object is created. See the constructor for BSTMapNode. As a new node is added to the tree, you will need to update the $size$ for all affected nodes, as you do for $height$ and $balanceFactor$.

***The only file you modify is* BSTMap.java!**

BSTMap

BSTMap is the class whose methods you are required to complete. Complete the methods following their requirements and hints in their Javadoc. Below are some important notes.

Recursion

A binary search tree is recursive in nature – every node induces a subtree consisting of the node and all its descendants, and every subtree preserves the BST property and is therefore a BST itself. Consequently, the cleanest and most elegant implementations of BST methods are recursive.

For this assignment ***you are encouraged to implement all the methods by recursion***. Because you recurse on the nodes of the tree and the public method stubs do not take a node as an input parameter, you will need to write private recursive helper methods as you saw in the SimpleBST class. Be sure to Javadoc all your helper methods.

Put, put helper and the helper's helpers

By far the most complicated method required for this assignment is put(). However, it is essentially the same as its counterpart in SimpleBST. Therefore, you can use the code for put(), putHelper() and the helper's helpers – balance(), rotateLeft(), rotateRight(), update() and height(), from the SimpleBST class. The only new items you need to incorporate are as follows:

1. Modify the code properly to handle the generic types K and V for the key and value.
2. Provide the code for rotateLeft() which mirrors rotateRight(). You can get lots of help from Slide 33 for the BST lectures when completing this method.
3. In update(), update the $size$ instance variable in addition to $height$ and $balanceFactor$. You need to think of the relation between the size of a node and the sizes of its left and right children, and how to define the size of a null node.
4. Handle the exceptions properly.

Time Complexity

Both **put()** and **get()** should have time complexity $O(\log n)$ as in SimpleBST.

The time complexity of **reverseOrder()** should be $O(n)$. However, you would ***not*** receive credit if you perform an in-order traversal of the tree and then reverse the order, as this is unnecessary. Instead your method should directly obtain the list of all keys in *descending* order. **Hint**: Modify in-order traversal by changing the order in which the nodes are visited.

The time complexity of **kSmallest($\boldsymbol{k}$)** should be $O(\log n + k)$. Note that the required time complexity does ***not*** allow you to perform an in-order traversal on the entire tree and then return the k smallest keys. Instead you should only traverse the branches of the tree necessary to get the data you need.

Grading

- [25 Points] put()
- [25 Points] get()
- [25 Points] reverseOrder()
- [25 Points] kSmallest()