Implementing a Binary Search Tree

Summary: In this assignment, you will complete the implementation of a Binary Search Tree, and write several methods that manipulate it.

1 Background

In order to practice implementing symbol tables, you will implement a binary search tree. A BST can actually be considered to be another type of data structure that we can use to implement an ADT. Since symbol tables are about finding values based on keys (the "search" problem), it's appropriate to use a BST to implement them. BSTs are nice structures for searching, because they mimic the process of a binary search ($O(logn)$). Unfortunately, fast $O(logn)$ look ups only work in BSTs that are balanced. Any time we remove or add nodes, there is a chance the tree will become stilted, which can degrade search to look like $O(n)$. After we finish the implementation of a BST, we will go on to address this problem.

This document is separated into four sections: Background, Requirements, Testing, and Submission. You have almost finished reading the Background section already. In Requirements, we will discuss what is expected of you in this homework. In Testing, we give some basic suggestions on how the tree additions should be tested. Lastly, Submission discusses how your source code should be submitted on Canvas.

2 Requirements [40 points]

In this assignment you will practice implementing symbol tables using BSTs. Provided for you is a base file (CompletedBST.java) that you will complete and submit. The purpose of OrderedSymbolTable and SymbolTable are only to define the ADTs which the BST data structure supports. The BST interface defines some additional pieces of functionality specific to BST implementations of symbol tables like balance(). (Note: the BST interface also contains a getRoot() method. This method exists only for testing, in practice we would avoid have something that so immediately exposes the internal representation.)

- Implement contains(), isEmpty(), deleteMax(), and size(Key lo, Key hi). [4 points]
- Recursive methods are nice since it is easy to tell they work, however, they tend to be slower than non-recursive methods. Give **non-recursive** implementations of get() and put(). (Hint: the book contains the solution for one of these.) Include them in your BST class as new methods called getFast() and putFast(). [6 points]
- Write a method that balances an existing BST, call it balance(). A BST is balanced if the height of its left and right sub-trees are different by at most one. Recursively applied. If the tree is balanced, then searching for keys will take act like binary search and require only logn comparisons. No performance requirements on your balancing algorithm. (Come up with a way yourself - don't skip to 3.3. That section is really complicated and meant for the harder case where you need to do it in log time.) [18 points]
- Sedgewick 3.2.37: Write a method displayLevel(Key key) that takes a Key as argument and prints the values in the subtree rooted at that node in level order (in order of their distance from the root, with nodes on each level in order from left to right). Hint: Use a Queue. [12 points]

2.1 Packages

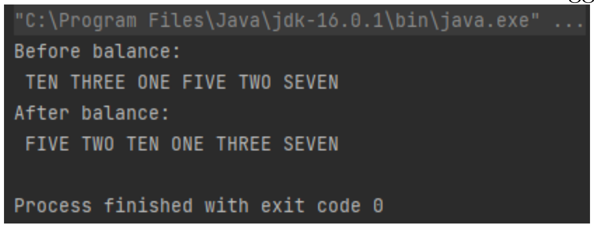
Do not import any packages other than java.util.Collections, java.util.LinkedList, java.util.NoSuchElementException, or java.util.Queue. (Do not use any star imports.)

Required Files	Optional Files
CompletedBST.java	(none)

Table 1: Submission ZIP file contents.

3 Testing

For most of the methods that you write, you should be able to tell if they are correct by inspecting the output. However, the balance and printLevel methods are more tricky. In the base file, there is sample code that builds a BST, displays it (using printLevel), balances it, and displays it again. The output is shown below. You may want to start by implementing printLevel and seeing if the output matches the screen shot for the "before balance" state of the tree. Ideally you would write a couple of additional tests to verify that level-wise display is working. After that is done, then you can move on to checking balance. Your answer may or may not exactly match the output below. If you want to be absolutely sure it is balanced, you'll need to take a look at the tree's structure in a debugger.



```
"C:\Program Files\Java\jdk-16.0.1\bin\java.exe" ...
Before balance:
 TEN THREE ONE FIVE TWO SEVEN
After balance:
 FIVE TWO TEN ONE THREE SEVEN

Process finished with exit code 0
```

4 Submission

The submission for this assignment has one part: a source code submission.

Writeup: For this assignment, no write up is required.

Source Code: The source file must be named as "CompletedBST.java", and then added to a ZIP file (which can be called anything). The class must be in the "edu.ser222.m03_02" package, as already done in the provided base file (do not change it!). You will submit the ZIP on Gradescope.

4.1 Gradescope

This assignment will be graded using the Gradescope platform. Gradescope enables cloud-based assessment of your programming assignments. Our implementation of Gradescope works by downloading your assignment to a virtual machine in the cloud, running a suite of test cases, and then computing a tentative grade for your assignment. A few key points:

- **Grades computed after uploading a submission to Gradescope are NOT FINAL.** We have final say over the grade, and may adjust it upwards or downwards.
- **Additional information on the test cases used for grading is not available, all the information you need (plus some commonsense and attention to detail) is provided in the assignment specification.** Note that each test case will show a small hint in its title about what it is testing that can help you to target what needs to be investigated.

If you have a hard time passing a test case in Gradescope, you should consider if you have made any additional assumptions during development that were not listed in the assignment, and then try to make your submission more general to remove those assumptions.

Protip: the Gradescope tests should be seen as a way to get immediate feedback on your program. This helps you both to make sure you are meeting the assignments requirements, and to check that you are applying your knowledge correctly. Food for thought: if you start on the assignment early, check against our suite often, and use its feedback, there's no reason why you can't both get full credit and know that you'll get full credit even before the deadline.

4.1.1 Standard Programming Deductions

Due to the nature of cloud grading, we use a different policy for the standard deductions:

- **If your submission does not compile, or is missing the file mentioned above, you will receive a zero grade.**
- **Following the file submission standards (e.g., the submission contains project files, lacks a proper header) is optional, and will not be enforced.** (We would appreciate if you at least included the header though!)