

# 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 ( $O(\log n)$ ). Unfortunately, fast  $O(\log n)$  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 BlackBoard.

## 2 Requirements [43 points]

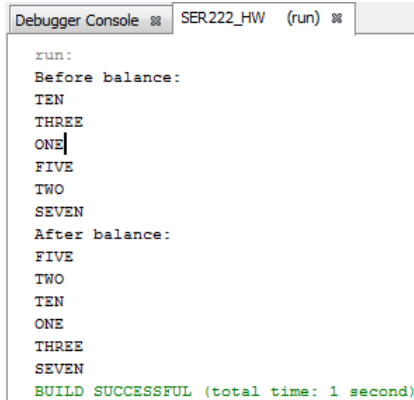
In this assignment you will practice implementing symbol tables using BSTs. Download the attached base file and then rename it, and the class inside, to include your last name instead of "Base". You will only need to change the base file. The purpose of OrderedSymbolTable and SymbolTable are only to define the ADTs which the BST data structure supports.

- Implement `contains()`, `isEmpty()`, `ceiling()`, `deleteMax()`, and `size(Key lo, Key hi)`. [7 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()`. If the tree is balanced, then searching for keys will take act like binary search and require only  $\log n$  comparisons. (Come up with a way yourself - don’t skip to 3.3.) [18 points]
- Sedgewick 3.2.37: Write a method `printLevel(Key key)` that takes a Key as argument and prints the keys 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]

## 3 Testing

For most of the methods that you write, you should be able to tell if they are correct. 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.



```
run:
Before balance:
TEN
THREE
ONE
FIVE
TWO
SEVEN
After balance:
FIVE
TWO
TEN
ONE
THREE
SEVEN
BUILD SUCCESSFUL (total time: 1 second)
```

## 4 Submission

The submission for this assignment has one part: a source code submission. The file should be attached to the homework submission link on BlackBoard.

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

**Source Code:** Please name your main class as "LastNameBSTST.java" (e.g. "AcunaBSTST.java"). The class must be in the default package.