Programming Assignment #2: GenericBST

COP 3503, Fall 2020

Due: Sunday, September 20, *before* 11:59 PM

Abstract

In this assignment, you will gain experience working with generics in Java. In particular, you will extend my binary search tree (BST) code from a data structure that just holds integers to a powerful container that can hold any kind of Comparable object under the sun.

A tangential benefit of this program is that you get to spend some time looking through some fairly straight-forward and well-structured BST code. It is always good to revisit how common data structures can be implemented as you ramp up your skills in different programming languages. You will also gain experience modifying someone else's code (something you will be doing a lot if you become a software developer) and adding comments to code.

Deliverables

GenericBST.java

Note! The capitalization and spelling of your filename matter!

Note! Code must be tested on Eustis, but submitted via Webcourses.

1. Problem Statement

Included with this assignment is a source file named *BST.java*. You must modify that file as follows:

1. Change the class name to *GenericBST* and the file name to *GenericBST.java*.
2. Modify the source code so that the class is generic (i.e., so that the binary search trees will be able to hold any type of data) with the restriction that only classes that implement *Comparable* may be stored in the BSTs. As with *Cluster.java* (posted in Webcourses), you do *not* need to write a *compareTo()* method.
3. The comments in *BST.java* are practically non-existent. After you have read and absorbed the program's structure and functionality, add comments for the entire program. (See the commenting guidelines below.)
4. Write a *public static double difficultyRating()* method that returns a double on the range of 1.0 (ridiculously easy) through 5.0 (insanely difficult) indicating how difficult you found this assignment.
5. Write a *public static double hoursSpent()* method that returns a realistic and reasonable estimate (greater than zero) of the number of hours you spent working on this assignment.

To make this class generic, you will have to change the return types of some of the methods I have implemented. However, please do not change static methods to non-static methods, or private methods to public methods, and vice-versa.

2. Special Requirement: No Compile-Time Warnings (*Super Important!*)

Here are some special restrictions regarding compile-time warnings:

- ★ Your code must not produce any warnings when compiled on Eustis. For this particular assignment, it's especially important not to have any *-Xlint:unchecked* warnings. Please note that some systems don't produce *-Xlint:unchecked* warnings! The safest way to check whether your code is producing such warnings is to compile and run your program on Eustis with the *test-all.sh* script.
- ★ You cannot use the *@SuppressWarnings* annotation (or any similar annotations) to suppress warnings in this assignment.

Please do not give your code to classmates and ask them to check whether their compilers generate compile-time warnings for your code. Remember, sharing code in this course is out of bounds for assignments.

3. General Commenting Guidelines

Here are some guidelines to follow when adding comments to *GenericBST.java* or when commenting your own code this semester:

- ★ **Keep comments brief!** Don't comment every line of code. Also, try to avoid comments that exceed two or three lines. Note that the comments in my code in Webcourses are designed for educational reading. They're far more lengthy and verbose than anything you should be including in your own code.

- ★ In general, above each method signature, give a *brief*, high-level overview of what that method does. Say something about its input parameters, expected output, and return value (if applicable).
- ★ Within your methods, give high-level comments that tell what different chunks of code do (rather than commenting every single line of code). However:
  - ☆ Avoid simply repeating things that are "obvious" in your code, thereby making someone read your code twice (once in English and once in Java). That is, avoid transliterating your code from Java to English. (Again: You shouldn't be commenting every single line of code.)
  - ☆ Prefer comments that tell *how* or *why* something works, rather than simply saying *what* something does, unless you're explaining the behavior of a chunk of code in a way that makes your code easier for a reader to digest.
- ★ When writing comments, a good rule of thumb is this: Suppose you load up your code for an assignment five years from now, and you've lost the PDF for that assignment. What comments would help you quickly understand what you were doing and how each of the methods plays into your program's overall functionality?
- ★ Try to use method names that are verb phrases, which therefore act as documentation of what your code is doing. (Avoid obscure method and variable names that would require us to add clarifying comments everywhere they appear.)
- ★ Don't copy and paste method descriptions from the assignment PDFs, as they sometimes contain non-standard characters that will prevent your code from compiling. (Also, the method descriptions in the PDFs are often way too verbose. You should write shorter overviews for each of your methods.)

4. Style Restrictions (Same as in Program #1) (*Super Important!*)

Please conform as closely as possible to the style I use while coding in class. To encourage everyone to develop a commitment to writing consistent and readable code, the following restrictions will be strictly enforced:

- ★ Capitalize the first letter of all class names. Use lowercase for the first letter of all method names.
- ★ Any time you open a curly brace, that curly brace should start on a new line.
- ★ Any time you open a new code block, indent all the code within that code block one level deeper than you were already indenting.
- ★ Be consistent with the amount of indentation you're using, and be consistent in using either spaces or tabs for indentation throughout your source file. If you're using spaces for indentation, please use at least two spaces for each new level of indentation, because trying to read code that uses just a single space for each level of indentation is downright painful.
- ★ Please avoid block-style comments: /* *comment* */
- ★ Instead, please use inline-style comments: // *comment*
- ★ Always include a space after the "//" in your comments: "// *comment*" instead of "//*comment*"

- ★ The header comments introducing your source file (including the comment(s) with your name, course number, semester, NID, and so on), should always be placed ***above*** your import statements.
- ★ Use end-of-line comments sparingly. Comments longer than three words should always be placed ***above*** the lines of code to which they refer. Furthermore, such comments should be indented to properly align with the code to which they refer. For example, if line 16 of your code is indented with two tabs, and line 15 contains a comment referring to line 16, then line 15 should also be intended with two tabs.
- ★ Please do not write excessively long lines of code. Lines must be no longer than 100 characters wide.
- ★ Avoid excessive consecutive blank lines. In general, you should never have more than one or two consecutive blank lines.
- ★ Please leave a space on both sides of any binary operators you use in your code (i.e., operators that take two operands). For example, use *(a + b) - c* instead of *(a+b)-c*. (The only place you do ***not*** have to follow this restriction is within the square brackets used to access an array index, as in: *array[i+j]*.)
- ★ When defining or calling a method, do not leave a space before its opening parenthesis. For example: use *System.out.println("Hi!")* instead of *System.out.println ("Hi!")*.
- ★ Do leave a space before the opening parenthesis in an *if* statement or a loop. For example, use use *for (i = 0; i < n; i++)* instead of *for(i = 0; i < n; i++)*, and use *if (condition)* instead of *if(condition)* or *if(condition)*.
- ★ Use meaningful variable names that convey the purpose of your variables. (The exceptions here are when using variables like *i*, *j*, and *k* for looping variables or *m* and *n* for the sizes of some inputs.)
- ★ Do not use *var* to declare variables.

5. Compiling and Testing GenericBST on Eustis (and the *test-all.sh* Script!)

Recall that your code must compile, run, and produce precisely the correct output on Eustis in order to receive full credit. Here's how to make that happen:

1. To compile your program with one of my test cases:

   ```
   javac GenericBST.java TestCase01.java
   ```

2. To run this test case and redirect your output to a text file:

   ```
   java TestCase01 > myoutput01.txt
   ```

3. To compare your program's output against the sample output file I've provided for this test case:

   ```
   diff myoutput01.txt sample_output/TestCase01-output.txt
   ```

   If the contents of *myoutput01.txt* and *TestCase01-output.txt* are exactly the same, *diff* won't print anything to the screen. It will just look like this:

```
seansz@eustis:~$ diff myoutput01.txt sample_output/TestCase01-output.txt
seansz@eustis:~$ _
```

Otherwise, if the files differ, *diff* will spit out some information about the lines that aren't the same.

4. I've also included a script, *test-all.sh*, that will compile and run all test cases for you. You can run it on Eustis by placing it in a directory with *GenericBST.java* and all the test case files and typing:

```
bash test-all.sh
```

Super Important: Using the *test-all.sh* script to test your code on Eustis is the safest, most sure-fire way to make sure your code is working properly before submitting.

6. Grading Criteria and Miscellaneous Requirements

> ***Important Note:*** When grading your programs, we will use different test cases from the ones we've released with this assignment, to ensure that no one can game the system and earn credit by simply hard-coding the expected output for the test cases we've released to you. You should create additional test cases of your own in order to thoroughly test your code. In creating your own test cases, you should always ask yourself, "What kinds of inputs could be passed to this program that don't violate any of the input specifications, but which haven't already been covered in the test cases included with the assignment?"

The *tentative* scoring breakdown (not set in stone) for this programming assignment is:

70%	Passes test cases with 100% correct output formatting. In testing, I'll throw some home-brewed Comparable objects into GenericBST containers and see whether the trees are properly constructed and the traversal outputs are correct. This portion of the grade includes tests of the *difficultyRating()* and *hoursSpent()* methods.
10%	Program compiles without warnings. To be eligible for these points, generics with Comparable must be implemented properly, and the code must not use any warning suppression annotations.
20%	Adequate comments and whitespace. To earn these points, you must adhere to the style restrictions set forth above. We will likely impose huge penalties for small deviations, because we really want you to develop good style habits in this class. Please include a header comment with your name and NID, and please be sure to name your file correctly.

Your program must be submitted via Webcourses.

Please be sure to submit your *.java* file, not a *.class* file (and certainly not a *.doc* or *.pdf* file). Your best bet is to submit your program in advance of the deadline, then download the source code from Webcourses, re-compile, and re-test your code in order to ensure that you uploaded the correct version of your source code.

Important! **Programs that do not compile on Eustis will receive zero credit.** When testing your code, you

should ensure that you place *GenericBST.java* alone in a directory with the test case files (source files, the *sample_output* directory, and the *test-all.sh* script), and no other files. That will help ensure that your *GenericBST.java* is not relying on external support classes that you've written in separate *.java* files but won't be including with your program submission.

Important! You might want to remove *main()* and then double check that your program compiles without it before submitting. Including a *main()* method can cause compilation issues if it includes references to home-brewed classes that you are not submitting with the assignment. Please remove.

Important! You should not print anything to the screen. Extraneous output will result in severe point deductions. The only output for this program should come from the pre-written tree traversal methods. Please do not make any changes to the print statements in the tree traversal methods.

Important! No file I/O. Please do not read or write to any files.

Important! Please do not create a java package. Articulating a *package* in your source code could prevent it from compiling with our test cases, resulting in severe point deductions.

Important! Name your source file, class(es), and method(s) correctly. Minor errors in spelling and/or capitalization could be hugely disruptive to the grading process and may result in severe point deductions. Similarly, failing to implement a required method, or failing to make certain methods *public*, *private*, *static*, and/or non-static (as required), may cause test case failure. Please double check your work!

Test your code thoroughly. Please be sure to create your own test cases and thoroughly test your code. You're welcome to share test cases with each other, as long as your test cases don't include any solution code for the assignment itself.

Start early! Work hard! Ask questions! Good luck!