CIT 5940 - Module 6 Programming Assignment
Binary Search Trees

CIT5940 Staff

Contents

Assignment Overview

In this module, we learned that a Binary Search Tree (BST) must remain balanced in order to guarantee $O(\log_2 n)$ operations. This assignment asks you to use and modify a Binary Search Tree implementation in order to determine whether the tree is balanced.

Learning Objectives

- Apply what you have learned about how Binary Search Trees represent and store data
- Implement tree traversal algorithms for determining the structure of a tree
- Modify an existing Binary Search Tree implementation

Advice

We highly encourage you to utilize the course discussion forum to discuss the project with classmates and TAs. Before asking the question, please search in the forum to see if there are any similar questions asked before. Please check *all* suggestions listed there.

For this and all other assignments in this course, we recommend that you work on your local computer using an integrated development environment (IDE) such as Eclipse, IntelliJ, Visual Studio Code, or whatever IDE you used for any prior Java courses that you are familiar with. Although you will need to upload your solution to Codio for grading (see the Submission Section below) and could develop your solution on that platform, we recommend using industry standard tools such as Eclipse or IntelliJ so that you become more used to them and can take advantage of their features.

If you have trouble setting up your IDE or cannot get the starter code to compile, please post a note in the discussion forum and a member of the course staff will try to help you get started.

We expect that you will create your own JUnit tests to validate your solution before submission.

1 Setup

1. Begin by downloading the `CIT594-BSTs.zip` archive from Canvas, then extract the contents. The main files you will work in are `BinarySearchTree.java`. This contains the unimplemented methods for the code that you will write in this assignment.
2. Compile this code before making any changes. This will ensure your local environment is configured correctly.
3. Review the current implementation. You'll want to understand the various parts of the code and how they work together before making changes!
4. The implementation here constitutes an ordered set of values. Because uniqueness of values is maintained by calling each value's `compareTo` method, the implementation will reject any attempt at adding a `null` value to the set.

2 Requirements

2.1 Structure and Compiling

1. You MUST use a JDK version of 17 or higher.
2. You MAY use a JDK version higher than 17, but you MUST set the Java language level to 17 for compatibility with Codio.
3. You MUST NOT change any method signatures for any of the provided methods that you must implement. This includes the parameter lists, names, and return value types, etc.
4. You MUST leave all classes in the default package.
5. You MUST NOT create additional `.java` files for your solution.
6. You SHOULD create JUnit tests to help validate your code. These tests SHOULD NOT be a part of your solution. We do expect you to test your code thoroughly with JUnit tests before submission.
7. You MUST follow our directives in the starter code comments. This means if we ask you to not change a variable's value, you MUST NOT change that variable's value.
8. You MAY create additional helper functions, as long as they meet the other requirements of the assignment (e.g. doesn't crash with invalid inputs, etc.).
9. You MAY add to the `BinarySearchTree` class or the inner `Node` class if desired
10. You MAY add additional classes, but you MUST define them in `BinarySearchTree.java`
11. You MUST fill out the required Academic Integrity signature in the comment block at the top of your submission files.

2.2 Functionality Specifications

2.2.1 General Requirements

1. Your method MUST NOT crash if a user enters an invalid input. Remember you do not have control over what a user inputs as arguments. For example, all methods should handle `null` inputs gracefully.
2. You MUST adhere to these definitions:
   (a) A **leaf** `Node` is a `Node` with no successor `Node`.
   (b) A **parent** `Node` is a `Node` with successors.
   (c) A **child** `Node` is a `Node` that is a successor of a parent `Node`.
   (d) The **root** `Node` is the highest `Node` in the structure and has no parent `Node`.

2.2.2 `Node findNode(E)`

1. this method MUST take a generic value as the argument
2. it MUST return the corresponding `Node` that holds the value
3. if the value is `null`, it MUST return `null`
4. if the value does not exist in the binary search tree, it MUST return `null`

2.2.3 `int depth(E)`

1. this method MUST take a generic value as the argument
2. it MUST return the **depth** of the corresponding `Node` that holds the value
3. the **depth** is the number of edges from that `Node` to the root `Node`
4. the **depth** of the root `Node` is `0`
5. the **depth** of any immediate child of the root `Node` is `1`
6. you MUST handle the following edge cases:
   (a) if the value is `null`, it MUST return `-1`
   (b) if the value does not exist in the binary search tree, it MUST return `-1`

2.2.4 `static int height(BinarySearchTree<?>.Node)`

1. this method MUST take a `Node` as the argument
2. it MUST return the **height** of the subtree rooted at the input `Node`
3. the **height** of a subtree is the maximum number of edges between the root of the subtree and any descendant leaf
4. the **height** of a leaf is `0`
5. the **height** of an empty subtree (represented as a `null Node`) is `-1`

2.2.5 `static boolean isBalancedNode(BinarySearchTree<?>.Node)`

1. this method MUST take a `Node` as the argument
2. it MUST return `true` if and only if the absolute value of the difference between the **height** of the left subtree and the **height** of the right subtree is less than `2`
3. it MUST return `true` if the input is `null`
4. it MUST return `false` in all other scenarios

2.2.6 `boolean isBalanced()`

1. this method MUST NOT take any arguments
2. it MUST return false if and only if there exists any `Node n` in the tree for which `isBalancedNode(n)` returns `false`; otherwise return `true`

2.3 Writing Test Cases

Upon completion of each functional requirement, always conduct thorough testing of the corresponding functions. Consult the guidelines on composing JUnit tests and creating test cases in Module 1. You MUST develop JUnit tests to cover all corner cases you can think of for each of your implemented function in order to get full credit. This testing component carries a weight of approximately 5% towards the total assignment score.

3 Submission

3.1 Pre-submission Check

Before you submit, please double check that you followed all the instructions, especially the items in the Structure and Compiling section above.

3.2 Codio Submission

1. When you are ready to submit the assignment, go to the Module X Programming Assignment Submission item and click the button to go to the Codio platform.
2. Once you are logged into Codio, read the submission instructions in the `README` file. This should be the first thing that appears in the window.
3. Upload your solution and any JUnit test files to the "submit" folder.
4. In the menu bar, select Run JUnit Tests. This will run any unit test files that you have uploaded with your submission. Note that there are no pre-submission checks provided. However you can use this feature as a basic validation check to be sure your code compiles.
5. When you're ready to submit, go to Education and select Mark As Completed. Confirm at the prompt.
6. You will see quite a bit of output, even if all the tests pass. At the bottom of the output, starting at `YOUR AUTOGRADING RESULTS BELOW` you will see the number of successful and failed test cases.
7. If you want to update your code and resubmit, for example if you did not pass all of the tests, un-mark the assignment as complete. You will then be able to edit your code, run your JUnit tests, and resubmit.

4 Grading

4.1 Grading Overview

1. There is no peer review for this assignment. You will not be marked on code style. However, we do expect you to use best practices (comments, indentation, etc), especially if you want to review with a TA.
2. There are no hidden tests. You will be able to see the name of each test, if it passed or failed, and the associated point value.
3. Failed tests will have brief feedback about the specifics of the failure.
4. We will not provide the exact test cases.
5. You have unlimited submissions, until the due date of the assignment as noted in the Syllabus (or your approved extension).
6. The score at the due date is final. Scores will sync to Canvas relatively immediately.

4.2 Rubric Items

This assignment will be marked on 5 rubric items.

- **`findNode` method** : 14 points
- **`depth` method** : 13 points
- **`height` method** : 9 points
- **`isBalancedNode` method** : 17 points
- **`isBalanced` method** : 18 points

Total : 71 points (+ 5 points for JUnit test cases)

Good Luck!