Connection to 127.0.0.1:30030 was lost. Please make sure you're connected to the Internet and try again.
(Development Only)
details... Close

p2 BST and Balanced Search Tree AVL

Search Trees: BST and AVL

Announcements | Overview | Files | Steps | Submission

Announcements (and answers to FAQs)

Corrections, clarifications, and other announcements regarding this programming assignment will be found below.

- **2/15: removed blue highlights from Exception hierarchy diagram (do not edit or submit exception types)**
- **2/12: Added BSTNode.java to the list of files provided (this type may be used, edited as you wish)**
- **2/11: Program assigned.**
- **This is an individual assignment.**
- **Students may not work together to complete any part of this programming assignment.**
- Use Piazza (Links to an external site.) to ask questions about the assignment.
  Please search for related posts before posting new questions.
- TAs will be available in the CS Labs (See lab hours schedule (Links to an external site.))

REMINDER: Students caught posting coursework online (i.e. for seeking help or a paid programmer) will get a zero on the assignment for a first offense and may fail course for multiple offense. (Student Conduct (Links to an external site.))

Learning Outcomes p2 (Links to an external site.)

Overview

For this assignment, you are required to implement the BST (Binary Search Tree) class, the AVL balanced search tree class, and a JUnit test class that shows that your implementations are complete and correct.

The provided starter source files establish the inheritance hierarchy that we require for this assignment.

Students may modify the BSTNode class and create an AVLNode class in any way that you wish.

Do not add public fields, methods, constructors to other interface or class types.

UML Class Diagrams

Fig 1: UML Class Diagram showing relationships between:
DataStructureADT, SearchTreeADT, BSTADT, BST, and AVL

BST implements BSTADT which extends SearchTreeADT which extends DataStructureADT, and AVL inherits from BST

Fig 2: UML Class Diagrams showing required exception types. (DO NOT EDIT OR SUBMIT exception types)

p2_201901_exceptions-3.png

Files

Note: updates to these files are possible, and will be posted in announcements on this page.

1. **p2.zip (Links to an external site.) (all of the files below in one zip file)**
2. DataStructureADT (Links to an external site.) - (provided interface - do NOT EDIT and do NOT SUBMIT)
3. SearchTreeADT (Links to an external site.) - (provided interface - do NOT EDIT and do NOT SUBMIT)
4. BSTADT (Links to an external site.) (provided interface - do NOT EDIT and do NOT SUBMIT)
5. IllegalNullKeyException (Links to an external site.) extends Exception (provided - do NOT EDIT and do NOT SUBMIT)
6. KeyNotFoundException (Links to an external site.) extends Exception (provided - do NOT EDIT and do NOT SUBMIT)
7. DuplicateKeyException (Links to an external site.) extends Exception (provided - do NOT EDIT and do NOT SUBMIT)
8. **BST (Links to an external site.) implements BSTADT (do edit and submit)**
9. **AVL (Links to an external site.) extends BST (do edit and submit)**
10. **BSTTest (Links to an external site.) - a JUnit test class for the BST class (do edit and submit)**
11. **AVLTest (Links to an external site.) - a JUnit test class for the AVL class (do edit and submit)**
12. **BSTNode (Links to an external site.) (if any of your classes use this type)**

Requirements

1. **Follow the information documented in the provided interfaces and class starters.**
2. **Follow the comments in each starter class. For example:**
   - **getKeyAtRoot()** should return a K type (not a node type)
   - **getKeyOfLeftChildOf(K)** should return a K type that is the value of the key that is in the left child of the node that contains the specified key. Or, it returns null, if there is no left child of the specified key.
   - Note: All traversals methods return a **List<K>** and not a List<KeyValuePairs> or nodes.
3. **Define (code) BST<K extends Comparable<K>>**
   - Must be generic type
   - Must implement the **BSTADT<K extends Comparable<K>>** interface
   - Must implement the binary search tree operations without re-balancing
4. **Define (code) AVL<K extends Comparable<K>>**
   - Must be generic type
   - Must extends **BST<K extends Comparable<K>>**
   - Must implement an AVL operations and maintain balance.
5. **Expected Exceptions: (must be thrown as indicated in ADT comments)**
   - **IllegalNullKeyException**
   - **KeyNotFoundException**
   - **DuplicateKeyException**
6. **Define a class named BSTTest.java as a JUnit test class.**
   1. Add JUnit tests to it to fully test your BST class.
   2. Test method names must be consistent and descriptive; see provided tests and p1 for simple test examples, i.e.:
      1. test01_isEmpty
      2. test03_insertManyDeleteOne
   3. Test all boundary conditions, i.e.:
      - Test for correct operations on an empty tree
      - Trees with one item
      - Trees with a few to several items added in an order that is balanced
      - Trees with a few to several items added in an order that is **not** balanced
      - Trees with a few to several items added and deleted
      - Trees with many items added and deleted
      - Trees with items added, deleted, and new items added again
      - etc...
7. **Define a class named AVLTest.java as a JUnit test class.**
   1. Add JUnit tests to it to fully test your AVL class.
   2. Note: **AVL** is a **BST;** so implementing BST and extending it via inheritance should mean that your AVL compiles and runs and passes all basic BST tests. However, it will not maintain balance as is required by the AVL insert and remove operations. You will need to override some parts of BST in the AVL class to restore balance after inserts and removes.
8. Save and submit screen shot of JUnit test results (results.png or results.jpg)
9. We provided the skeleton of two test classes to get you started, and we recommend trying Test-Driven Development where you write a test first and then add code to pass your test.

Note: Part of your score for this assignment will depend upon if you are able to test the balancing of the AVL operations as well as the general functionality of both classes. Test the easy stuff and several known insert and delete sequences that will allow you to check if the re-balancing has worked for your AVL. Test that your structures throw the correct exception types.

Although not required, we **highly** recommend the use of a code coverage tool. Eclipse offers a free tool called EclEmma (Links to an external site.). This will help you in finding places in your code which you have not yet tested.

Windows Users: may wish to download and install this Snipping Tool (Links to an external site.)

Steps

1. Read assignment and review grading rubric.
   Note: The rubric is subject to some changes.
2. You may use the Java development environment of your choice. *However, all programs must compile and run using JUnit5 and Java 8 from the Linux workstations in the lab computers for grading.* If you are going to use the CS lab computers, we recommend that you use Eclipse. You may want to review the Eclipse tutorial (Links to an external site.) to learn the basics. Note that on the Linux lab computers, you should enter "eclipse&" at the prompt instead of what is described in the tutorial.
3. Create a **p2_project** folder for your work and copy the provided source files into your project folder and refresh.

   project_folder.png
4. Review the starter classes and complete some of the missing test bodies
5. Identify (write on paper or text file - not in code yet) additional tests that will determine if your structure works as advertised (required)
   1. Name and briefly describe each test
   2. What is the "setup" for the test
   3. What operations will be called
   4. What is expected result of each operation
   5. How will you get the actual result for comparison
   6. Follow test pattern given for all tests
6. Add your own tests to the test class
   1. Do they pass?
   2. If not, add code in your data structure to implement the operation being tested and rerun tests
   3. Repeat this step as you...
7. Incrementally develop your solution:
   1. Write some code.
   2. Test your partial implementation.
   3. **Submit your partially complete solution files.**
8. Repeat until implementation is complete and correct
9. (Links to an external site.)

Submit your files to p2 (Links to an external site.)