Binary Search Tree

This assignment involves implementing a standard binary search tree as a Java generic. The primary purpose of the assignment is to ensure that you have experience with some of the issues that arise with Java generics, and with the common design patterns that arise in implementing tree structures in Java.

Design and Implementation Requirements

There are some explicit requirements, stated in the header comments for `BST<>` and in the header comments for some of the specified **`public`** methods, in addition to those on the *Programming Standards* page of the course website. Your implementation must conform to those requirements. In addition, none of the specified `BST<>` member functions should write output.

You may safely add features to the given interface, but if you omit or modify members of the given interface you will almost certainly face compilation errors when you submit your implementation for testing.

You must implement all tree traversals recursively, not iteratively. You will certainly need to add a number of **`private`** recursive helper functions that are not shown above. Since those will never be called directly by the test code, the interfaces are up to you.

Be sure to read and follow the instructions for how to implement deletion in your BST; the testing of your solution will be picky about that.

You must place the declaration of your binary search tree generic in a package named `CS3114.J2.DS` and specify **`public`** and package access for members as indicated in the interface specification, or compilation will fail when you submit it.

Design and Implementation Suggestions

In most cases, the **`public`** functions shown in the BST interface will be little more than stubs that call recursive **`private`** helper functions. This is the standard pattern for implementation, and it is natural, since the recursive traversals require a parameter that points to a tree node (and the client who makes calls to the **`public`** function will not possess any such pointer).

There are ways to avoid this. For example, the **`public`** function could take a node pointer, and then call itself to perform the traversal. In that case, the caller would presumably pass **`null`** for that parameter, and the implementation of the **`public`** function would interpret that as a sign to begin at the root node. This can be made to work, but it's fairly clumsy since the client is exposed to a seemingly meaningless parameter, and may misinterpret your intent.

The traversal logic can be designed in several ways. My solution uses what might be called the "fall off the end of a branch" approach. That is, I will make recursive calls passing a **`null`** node pointer, and check for that at the beginning of the function body. Other solutions use what might be called the "one level look-ahead" approach. That is, they will check to see if the pointer in the parent node is **`null`** before making the recursive call. I find the former approach leads to simpler, cleaner code, but that is merely a matter of personal taste in design.

A similar dichotomy arises when designing search logic, which also plays a role in insertion and deletion. One approach is to proceed with the traversal until you arrive at the desired node (or fall off the end of a branch). Another is to look down from the parent node, and examine the child node before actually going there. Again, I find the former approach leads to cleaner code, and again that is merely a matter of taste.

Some operations can fail. The client may attempt to find or delete a value that doesn't occur in the tree, or may attempt to insert a value that already occurs in the tree (our version of the BST will not insert duplicate values). When an operation fails, it should not do so silently, and so the **`public`** functions will return an indicator of success or failure. If the **`public`** function returns a pointer (Java reference), it could simply return **`null`** on failure. Otherwise, the **`public`** function will return a Boolean value.

Note that it is never acceptable to perform a needless traversal of the tree; that wastes time for no reason. For example, insertion and deletion logic should never call the **`public`** search function, or its **`private`** helper, to determine whether the target value is in the tree, and then perform a second search traversal if the value is absent (insertion) or present (deletion). Such implementations seem to be fairly common, perhaps due to a misunderstanding of the purpose of the software engineering goal of avoiding code duplication.

BST Interface

Because this assignment will be auto-graded using a test harness we will provide, your implementation must conform to the `public` interface below, and include at least all of the `public` and package access members that are shown:

```
// The test harness will belong to the following package; the BST
// implementation will belong to it as well.  In addition, the BST
// implementation will specify package access for the inner node class
// and all data members in order that the test harness may have access
// to them.
//
package CS3114.J2.DS;

// BST<> provides a generic implementation of a binary search tree
//
// BST<> implementation constraints:
//    - The tree uses package access for root, and for the node type.
//    - The node type uses package access for its data members.
//    - The tree never stores two objects for which compareTo() returns 0.
//    - All tree traversals are performed recursively.
//    - Optionally, the BST<> employs a pool of deleted nodes.
//      If so, when an insertion is performed, a node from the pool is used
//      unless the pool is empty, and when a deletion is performed, the
//      (cleaned) deleted node is added to the pool, unless the pool is
//      full.  The maximum size of the pool is set via the constructor.
//
// User data type (T) constraints:
//    - T implements compareTo() and equals() appropriately
//    - compareTo() and equals() are consistent; that is, compareTo()
//      returns 0 in exactly the same situations equals() returns true
//
public class BST<T extends Comparable<? super T>> {

    class BinaryNode {
       // Initialize a childless binary node.
       // Pre:    elem is not null
       // Post:   (in the new node)
       //         element == elem
       //         left == right == null
       public BinaryNode( T elem ) { . . . }

       // Initialize a binary node with children.
       // Pre:    elem is not null
       // Post:   (in the new node)
       //         element == elem
       //         left == lt, right == rt
       public BinaryNode( T elem, BinaryNode lt, BinaryNode rt ) { . . . }

       T          element;  // the data in the node
       BinaryNode left;     // pointer to the left child
       BinaryNode right;    // pointer to the right child
    }
```

```
      BinaryNode root;        // pointer to root node, if present
      BinaryNode pool;        // pointer to first node in the pool
      int        pSize;       // size limit for node pool

      // Initialize empty BST with no node pool.
      // Pre:   none
      // Post:  (in the new tree)
      //        root == null, pool == null, pSize = 0
      public BST( ) { . . . }

      // Initialize empty BST with a node pool of up to pSize nodes.
      // Pre:   none
      // Post:  (in the new tree)
      //        root == null, pool = null, pSize == Sz
      public BST( int Sz ) { . . . }

      // Return true iff BST contains no nodes.
      // Pre:   none
      // Post:  the binary tree is unchanged
      public boolean isEmpty( ) { . . . }

      // Return pointer to matching data element, or null if no matching
      // element exists in the BST.  "Matching" should be tested using the
      // data object's compareTo() method.
      // Pre:  x is null or points to a valid object of type T
      // Post: the binary tree is unchanged
      public T find( T x ) { . . . }

      // Insert element x into BST, unless it is already stored.  Return true
      // if insertion is performed and false otherwise.
      // Pre:   x is null or points to a valid object of type T
      // Post:  the binary tree contains x
      public boolean insert( T x ) { . . . }

      // Delete element matching x from the BST, if present.  Return true if
      // matching element is removed from the tree and false otherwise.
      // Pre:   x is null or points to a valid object of type T
      // Post:  the binary tree does not contain x
      public boolean remove( T x ) { . . . }

      // Remove from the tree all values y such that y > x, according to
      // compareTo().
      // Pre:   x is null or points to a valid object of type T
      // Post:  the tree contains no value y such that compareTo()
      //           indicates y > x
      public void capWith( T x ) { . . . }

      // Return true iff other is a BST that has the same physical structure
      // and stores equal data values in corresponding nodes.  "Equal" should
      // be tested using the data object's equals() method.
      // Pre:   other is null or points to a valid BST<> object, instantiated
      //           on the same data type as the tree on which equals() is invoked
      // Post:  both binary trees are unchanged
      public boolean equals(Object other) { . . . }
   }
```

Notes on Deletion

The test harness will expect you to handle deletion of a node in a precisely-specified manner. Deletion of a leaf simply requires setting the pointer to that node to **`null`**. Deletion of a node with one non-empty subtree simply requires setting the pointer to that node to point to the node's subtree.

Deletion of a node with two non-empty subtrees must be handled in the following manner. First, locate the node rMin that holds the minimum value in the right subtree of the node to be deleted. Then, detach rMin from the right subtree using the appropriate logic for a leaf or a one-subtree node. Next, replace the node to be deleted with rMin, by resetting pointers as necessary. Do not simply copy the data reference from rMin to the node containing the value to be deleted.

The test harness will expect deletion to be handled in precisely this manner, and will penalize you if it is not.

Notes on the Node Pool

If the node pool is enabled (by calling the second constructor):

- The pool will hold up to `pSize` nodes, which will not contain data elements (i.e., `null` data reference).
- The nodes in the pool will be linked into a linear list via their `right` child pointers. Failure to do so will defeat the grading code and result in no credit for your node pool.
- When `remove()` is called, the node will be added to the pool unless the pool is full. Whether you update the pool on calls to `capWith()` and `clear()` is up to you; our testing of the pool will not consider those cases.
- When an insertion is performed, a node from the pool will be used unless the pool is empty.

Test Harness and Grading

We will be testing your implementation with our own test code, which will be posted in a zip file on the course website. Create a test directory on a Linux system, and unpack the test code zip file there.

`gradeJ2.sh`	bash shell script that runs the grading tools
`readme.txt`	explains how this all works
`tools.zip, which contains:`	
`   BSTGenerator.jar`	test case generator
`   LogComparator.jar`	grading comparator tool
`   testDriver.java`	test driver for BST
`   CS3114/J2/DS/BST.java`	shell file for BST soln
`   CS3114/J2/DS/Monk.class`	test class for BST

The supplied test code is thorough, but some edge cases may not occur on every execution of the test code, since it randomizes the generated test data. You should be sure to run the test code a number of times, using the randomization feature.

Evaluation

You should document your implementation in accordance with the *Programming Standards* page on the course website. It is possible that your implementation will be evaluated for documentation and design, as well as for correctness of results. If so, your submission that achieved the highest score will be evaluated by one of the TAs, who will assess a deduction (ideally zero) against your score from the Curator.

Note that the evaluation of your project may depend substantially on the quality of your code and documentation.

What to Submit

This assignment will be auto-graded under CentOS (which should not matter at all) using Java version 1.8u20 or later, and the posted testing/grading code.

Submit your `BST.java` file (not a zip or jar) containing your BST generic to the Curator System. Submit nothing else. Your solution should not write anything to standard output.

Your submitted source file will be placed in the appropriate subdirectory with the packaged test code, and will then be evaluated by running the supplied testing script. We will make no accommodations for submissions that do not work with that script.

Warning: the requirement here is for a plain, uncompressed Java source file. We will not accept files in any other format (e.g., tar files, 7-zip'd files, gzip'd files, jar files, rar files, …). Such submissions will NOT work with the supplied script, and will be discarded when we run the grading code ourselves.

You will be allowed to submit your solution multiple times, in order to make corrections. Your score will be determined by testing your last submission. The Curator will not be used to grade your submissions in real time, but you will have already done the grading using the posted code.

Instructions, and the appropriate link, for submitting to the Curator are given in the *Student Guide* at the Curator website:

http://www.cs.vt.edu/curator/.

Pledge

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the following pledge statement at the beginning of the file that contains `main()`:

```
//    On my honor:
//
//    - I have not discussed the Java language code in my program with
//      anyone other than my instructor or the teaching assistants
//      assigned to this course.
//
//    - I have not used Java language code obtained from another student,
//      or any other unauthorized source, including the Internet, either
//      modified or unmodified.
//
//    - If any Java language code or documentation used in my program
//      was obtained from another source, such as a text book or course
//      notes, that has been clearly noted with a proper citation in
//      the comments of my program.
//
//    - I have not designed this program in such a way as to defeat or
//      interfere with the normal operation of the grading code.
//
//    <Student's Name>
//    <Student's VT email PID>
```

We reserve the option of assigning a score of zero to any submission that is undocumented or does not contain this statement.

Change Log

Version	Date	Change(s)
3.00	Jan 15	Base document