Assignment 4
Binary Search Tree
4/8/19

Submit ZIP file with code (with directories for Java packages as needed).

Reference: “Part C: Tree Data Structures” and “Chapter S4 – Comparing”.

150 Points Total

Problem 1 of 2 – BinarySearchTree1 (80% of Scoring)

Add a class named “BinarySearchTree1”. Note that duplicate keys are allowed.

The tree should include the following methods (five methods total – of course you'll have more methods like helpers, etc).

Core Methods

Method Name: ***add***
Method Parameter: *Comparable*
Return Type: *No return value*
Description: *Add the Comparable (data) into the correct position in the tree*

Method Name: ***search***
Method Parameter: *A “Comparable” type, which is the data to search for*
Return Type: *An “Object”, which is the matching data*
Description: *Return the first data found that matches the method parameter value – Otherwise (if none found) return null.*

Traversal Methods

Three methods names as follows that collect the data in the specified order

- **collectPreOrder**
- **collectInOrder**
- **collectPostOrder**

For each of the three methods:

> Method Name: As shown above
> Method Parameter: *Each method has one method parameter: a List type*
> Return Type: *None*
> Purpose: *To traverse the tree in the order per the method name (PreOrder, InOrder and PostOrder). As it traverses it should add data to the method parameter (List) in the specified traversal order.*

Testing Suggestions (Information Only)

Simple types may be used to test your tree (String, Integer, etc) or more interesting classes that you could design. If you implement a class, you would want it to implement the Java "Comparable" interface. And of course you can easily generate large collections of your sample data.
There is simpler sample test data at the end of this document ("Problem 1 Sample Test Data") to help get started and for demonstration of the expected traversal order.

Problem 2 of 2 – BinarySearchTree (20% of Scoring)

This problem is similar to "Problem 1" except will add some practical searching features. Note that duplicate keys are allowed.

New Features

The tree in "Problem 1" had a couple **limitations**:

1. Comparable supports only one type of comparison for a given data type. Thus, if loading our tree with "Employee" objects, we would have to decide to compare based on employeeId OR firstName OR lastName. But we could choose only one. This would often be too limiting in practice.

2. The "search" method does not support a simple key. And a simple key would likely be required in practice. For example, if we have a tree of " Employee" objects compared by "firstName", it would naturally follow that we may want to search for a search key like: 1023 (id), "Zooster" (Last Name), etc. However, our tree (from Problem 1) does not support this – it would require that you have an employee object to search for an employee object.

So, for this problem (BinarySearchTree), we are going to eliminate both limitation #1 and #2 above.

Thus, the new tree will have these **features**:

1. The tree will be dynamic and support multiple comparison strategies for any given data type.

2. The tree will support searching be a proper search "key" (generally a simple type like String, Integer, etc)

BinarySearchTree

Name the new class named "BinarySearchTree".

Design and implement the tree per "New Features" section above.

The tree should be general – i.e. it should work for an infinite number of data types.

Core Methods

It should have the following methods:

Method Name: ***add***
Method Parameter: *The data to add into the tree. The method param type should be the same as the data type that the given tree contains*
Return Type: *No return value*
Description: *Add the data into the correct position in the tree*

Method Name: ***search***
Method Parameter: *One parameter "searchKey" (the type should be dynamic – you should design how to code the searchKey).*
Return Type: *The data type that a given tree contains (e.g. this is controlled by the tree user), Employee, Lake, etc.*
Description: *Return the first data found that matches the method parameter "searchKey", if none found, return null. The method parameter type should be dynamic (controlled by tree user). E.g. for a tree of "Employees", one tree instance may have a search key of Integer type (e.g. for id), and another tree instance may have a search key of String" type (e.g. for first or last name).*

Assume we have variable "tree" which is a tree of Employee objects sorted by employee name of type String. We should be able to do this without any compile errors/warnings:

```
Employee match = tree.search("Bob Achooz");
```

Assume we have another variable "tree2" which is a tree of Employee objects sorted by employee id of type Integer. We should be able to do this without any compile errors/warnings:

```
Employee match2 = tree2.search(1023);
```

And this (for either "tree" or "tree2" in the previous examples:

```
//given variable "newEmployee" of type Employee
tree.add( newEmployee);
```

Traversal

Three methods names as follows that collect the data in the specified order

- collectPreOrder
- collectInOrder
- collectPostOrder

These methods should be similar to Problem 1 except that the method parameter (a List type) should also declare the list element type. The list element type should match the type of data in the tree.

Extra Credit #1 (Up to 15 Points)

For Class: BinarySearchTree (from "Problem 2")
Add Method Named: ***selectBetween***
Method Parameters:

- *method parameter 1: A search key to find the "first" element in the "select"*
- *method parameter 2: A search key to find the "last" element in the "select"*
  The search key type should be the same as you use in the "search" method.

Return type: *A list with data (elements) of the same type as the tree data*
Description:
Select a set of elements per the search keys (inclusive)
For example if you have a tree of lakes sorted by max depth (shallowest to deepest), we might use this method like "lakes = selectBetween(50, 100):" which would return a list of all the lakes (Lake objects) with a max depth of greater or equal to 50 and less than or equal to 100). The returned lakes would be in the same order as the tree (i.e. for this example – shallowest to deepest). Part of the points will be based on devising an optimized approach that would not need to traverse the entire tree to solve the problem.

Extra Credit #2 (Up to 10 Points)

Design and implement a binary search tree iterator named "BSTIterator"

- The iterator should work on BinarySearchTree (from "Problem 2")
- The iterator should iterate in order (e.g. same order as "in-order" traversal)
- Use your Stack (not Java's Stack) for the implementation
- Do **not** use a Java List for any part of the iterator implementation
- The tree should support the method "tree.toIterator()" that would return a BSTIterator that would start it's iteration on the tree root node.
- The tree should also support the (overloaded) method "tree.toIterator(searchKey)" that would return the BSTIterator. The parameter "searchKey" should have the same type as declared for the tree's "search" method.
    - The tree should search for the first match
        - If found the returned iterator should be initialized on the matching node (i.e. when the iterator user begins iteration it would start iteration on that node).
        - if not found return null

Testing Suggestions (Information Only)

Sample test data is available on GitHub – "Lakes.zip". This data could be sorted and searched by a variety of keys such as lake name, max depth nearby town, and surface area.

Gather/generate your own data sets of interest to you.

Appendices

Problem 1 Sample Test Data

If the following elements are added to the tree:

50, 70, 20, 10, 60, 40, 30

Then traversal would be:

(preorder)
50
20
10
40
30
70
60

(inorder)
10
20
30
40
50
60
70

(postorder)
10
30
40
20
60
70
50

Problem 2 Sample Test Data

See “Lakes.zip” on GitHub...