Lab 8 – Implementing and Using a Binary Search Tree (BST)

Preliminaries

In this lab you will implement a binary search tree and then use it instead of Java's Set ADT in the ***WorkerManager*** program from ***Lab 03***. Recall that BSTs can behave as sets because they are efficient and can easily detect duplicates.

So far, we've done several linked implementations during lab; consequently, you are expected to complete this lab WITH MINIMAL to NO assistance from your lab instructor or TA. That time has finally arrived, my good friends!

Problem 1 (of 2): Building and testing your own linked binary search tree, using *structural* recursion

Part 1.1: Creating a shell for class <FOO>LinkedBinarySearchTree

Keep classes ***BSTTest.java*** and ***TestClassStats.java*** in your lab folder. Open file ***BSTTest.java*** and change all instances of ***<FOO>*** *and* ***<foo>*** to the appropriate initials. Next, create an appropriately named (***<YOUR CAPITALIZED INITIALS>LinkedBinarySearchTree***) class inside your ***JavaPackages/<your initials>structures*** package. This class should implement interface ***ZHBinarySearchTree*** available in package ***zhstructures*** and should have the class header shown below (replace ***<FOO>*** with your initials). Notice how the generic type `E` specified after the class name has the restriction that it must extend `Comparable<E>`; by now, you should know what this means and why this is the case:

```
public class <FOO>LinkedBinarySearchTree<E extends Comparable<E>> implements ZHBinarySearchTree<E>
```

Please visit https://faculty.csbsju.edu/irahal/SW/ZHStructures/ to locate the complete documentation for interface ***ZHBinarySearchTree*** and study it carefully. Next, create copies of all methods from this interface in your newly created class along with a default constructor, and implement them as stub methods: methods that return integers, Boolean or objects should simply return -1, ***false*** and ***null***, respectively. This should allow your class to compile without errors. Do not forget to complete the documentation for your class as you build it; use `{@inheritDoc}` when possible.

Like previous linked ADT implementations, your ***<FOO>LinkedBinarySearchTree*** class will require an inner class which we'll name ***BSTNode***. Include an inner class called ***BSTNode*** which extends the ***ZHBinaryTreeNode*** `abstract` class from package ***zhstructures*** and has the following class header:

```
private class BSTNode extends ZHBinaryTreeNode<E, BSTNode>
```

Notice how cool class ***ZHBinaryTreeNode*** is? It uses TWO generic types instead of just one! This shouldn't have much of an impact on your implementation, however. Take enough time to study class ***ZHBinaryTreeNode*** extended by your ***BSTNode*** inner class and the methods available in it: https://faculty.csbsju.edu/irahal/SW/ZHStructures/. It defines a generic tree node suitable for any binary tree.

Inner class ***BSTNode*** is similar to class ***ListNode*** from the indexed linked list implementation lab but supports a tree node instead. As with our ***IndexedList*** ADT, we will include in the outer class a reference to a node of the inner class. In this case, the node reference will be to the ***root*** of the tree. Most methods in the outer class will make calls on the ***root*** instance variable for recursive auxiliary methods to be defined in inner class ***BSTNode***. In addition, you'll need to maintain a ***size*** integer instance variable. Therefore, your outer class should only contain the following two `private` instance variables: ***root*** and ***size***. Don't change names of fields and methods; otherwise, you're likely to encounter failed JUnit tests.

Include three constructors in inner class ***BSTNode***: a default constructor, a second that takes an element parameter and a third one that takes three parameters: element, left child node and right child node. Each of the constructors corresponds exactly to a constructor of the super class ***ZHBinaryTreeNode***, having the same parameters with the same types and order. As before, the inner class constructors simply call the corresponding superclass constructors. Remember `super(…)`?

Methods of the outer class are all straightforward:

(1) The constructor initializes the ***root*** instance variable to a new empty ***BSTNode***

(2) Methods `isEmpty` and `size` use the ***size*** instance variable, as you'd expect (not recursive)

(3) Methods `contains`, `get`, and `iterator` simply call the corresponding inner class methods on the ***root*** (recall that we're doing structural recursion)

(4) Methods `add` and `remove` are almost as simple as (3), but they need to check the result of the inner class call and increment or decrement ***size*** respectively if the result is ***true***

We'll need to implement the following methods in inner class ***BSTNode***:

(1) Methods `iterator` and `contains`–for efficiency, we'll override these from superclass ***ZHBinaryTreeNode*** where they are declared `public`; we can't lower the visibility, so they'll have to be declared `public` in ***BSTNode***

(2) Methods `get`, `add`, and `remove`–must be declared `private` and implemented recursively

(3) Methods `copyNodeToThis` and `removeAndReturnLeftmost`–these two additional `private` helper methods are useful for implementing the `remove` method but are not part of the outer class public interface

After creating all necessary methods described above and implementing them as stub methods, for now, you should be able to compile your class without errors. In addition, you should be able to run the test class and get the two structure test methods (`testOuterClassStructure` and `testInnerClassStructure`) to pass. Since some of these methods are recursive and somewhat complex, we'll consider each of them separately later.

Part 1.2: Studying the visualization

We have created a visualization to help you understand the logic needed to complete your ADT, specifically methods `add`, `contains`, and `remove`. Please visit Binary Search Trees under **Lab08_BinarySearchTrees** on *https://faculty.csbsju.edu/irahal/SW/DataStructuresVisualizations/Algorithms.html*. Study these carefully.

Part 1.3: Completing and testing your methods

To implement the methods in your ADT while simultaneously testing them, please follow the same approach from previous labs.

`iterator` method

The `iterator` method may simply call any one of the iterator methods from superclass ***ZHBinaryTreeNode***; check them out on *https://faculty.csbsju.edu/irahal/SW/ZHStructures/*. You should use the in-order iterator because it iterates over the items in increasing order.

`contains` method *(refer to the visualization to understand the logic)*

Super class ***ZHBinaryTreeNode*** includes a generic public `contains` method that works for ANY binary tree, but it doesn't take advantage of the ordering in the binary search tree, so you *shouldn't* use it; instead, we need to override it for efficiency. The overridden method should be recursive with two base cases: if the current node is empty, it returns ***false***; if current node is not empty and contains the element we're looking for, it returns ***true***. There are also two recursive cases which you should be able to work out on your own. To determine the second base case and the two recursive cases, you need to perform a comparison between the element in this node and the element you're looking for. The following statement is a convenient way to do the comparison:

```
int comp = this.element.compareTo(element)
```

The value of `comp` will be zero if the two elements are equal, less than zero if `this.element` is less than parameter `element` and greater than zero if `this.element` is greater than parameter `element`.

`add` method (*refer to the visualization to understand the logic*)

The `add` method has a similar recursive structure as `contains`, but what you do in each case is different. If you arrive at a terminal node, it means you've reached the point where the new element should be added. To do this, we convert this node into a non-terminal node containing the new element by copying parameter `element` into the node's ***element*** field and putting new terminal nodes into the ***leftChild*** and ***rightChild*** fields. This case then returns ***true***. The next three cases depend on the same comparison we made in the `contains` method so you should be able to figure them out on your own.

****TEST: When you reach this point, make sure corresponding tests (for ALL methods completed so far) in the provided test class are successful before moving on.*

`get` method

The `get` method is also very similar to `contains`; it follows the same process but returns a ***null*** and `this.element` (NOT parameter `element`) in place of ***false*** and ***true***, respectively.

****TEST: Again, make sure corresponding tests (for ALL methods completed so far) work correctly before moving on.*

`remove` method (*refer to the visualization to understand the logic*)

The `remove` method is the most complicated method, so we'll include two auxiliary methods to make it easier to code: `copyNodeToThis` and `removeAndReturnLeftmost`.

Method `copyNodeToThis` simply copies all the instance variables from another parameter node into this node. The method header for this method is given below:

```
/**
 * Replaces values of instance fields element, leftChild and rightChild in this node with those from
 * parameter otherNode.
 * @param otherNode the node from which values of instance variables are to be copied into this node.
 */
private void copyNodeToThis(BSTNode otherNode)
```

Method `removeAndReturnLeftmost` recursively finds the leftmost node in this sub-tree, which is the node with the smallest ***element*** in the right sub-tree of the node with the element to be removed. The method remembers the ***element*** in the leftmost node, removes that node and returns the remembered value. There are two base cases: if this node is empty (which shouldn't happen because we're not going to call this method on an empty node), throw a `NoSuchElementException`*;* if this node's left sub-tree is empty, then this node is the leftmost node, in this case, it copies the ***rightChild*** into this node using the `copyNodeToThis` method and returns the old element that was in the node. The recursive case is when the left sub-tree is not empty; in this case, return the value produced by recursively calling `removeAndReturnLeftmost` on the ***leftChild****.* The header for this method is shown next:

```
/**
 * Removes and returns the leftmost element in this sub-tree.
 * @return the former leftmost element in this sub-tree.
 * @throws NoSuchElementException if this sub-tree is empty.
 */
private E removeAndReturnLeftmost()
```

The algorithm for `remove` has the same overall case structure as the other methods. There are three sub-cases (shown next) to consider when (and if) we find a node that contains an ***element*** equal to the element we want to remove. In all these cases, the method returns ***true***. As you might expect, ***false*** is returned instead if we never find the ***element*** in the tree.

(1) if the left sub-tree of this node is empty, we remove the element by copying the fields of ***rightChild*** (which may also be empty) into this node using `copyNodeToThis`

(2) similarly, if the right sub-tree is empty, we remove the element by copying the fields of ***leftChild*** into this node

(3) if neither sub-tree is empty, we remove the element by replacing the ***element*** in this node by an ***element*** in one of the sub-trees, either the leftmost element in the right sub-tree (which is what you'll do for this lab) or the rightmost element in the left sub-tree, and removing the latter–we can do this in one step by assigning the element value returned by calling `removeAndReturnLeftmost` on the ***rightChild*** to the current ***element***.

When you have completed these methods, use the supplied test program to test your code. Show your *fully documented* code and testing results to the lab instructor or TA before you move on.

Problem 2 (of 2): Modifying the WorkerManager program to use a tree of Workers instead of Java's Set ADT

If you have not already done so, complete the worker manager from ***Lab 03***. If neither you nor your programming partner are close to finishing that lab, arrange to get a copy from another student who has it completed; be sure that every file you copy from another person includes that person's name as author. Copy the four manager programs from ***Lab 03*** into today's lab folder: ***WorkerManager***, ***HourlyEmployeeManager***, ***SalariedEmployeeManager*** and ***VolunteerManager***

In this problem, we will be replacing the set of workers in the ***WorkerManager*** program with a binary search tree of workers. To do this, we have to take into account the generic-type restriction of the binary search tree: the generic element type must extend interface ***Comparable<E>***. We must therefore make the ***Worker*** hierarchy implement this interface. There are two changes required:

First, change the ***Worker*** interface so that it extends ***Comparable<Worker>*** (as well as ***Serializable*** if not already done). Second, add method ***public compareTo*** to the ***HourlyWorker*** and ***SalariedEmployee*** classes, such that the ***compareTo*** method returns the result of comparing the two names. Please note that a ***Worker's*** name is of type String and class ***String*** already has a ***compareTo*** method for you to use (similar to how we did the ***equals*** method).

In the ***WorkerManager*** class change the data type of the *workers* instance field to ***ZHBinarySearchTree<Worker>*** and replace the call to Java's ***HashSet*** constructor in the ***WorkerManager's*** constructor with your linked binary search tree constructor. Recompile the ***WorkerManager*** and, if necessary, the binary search tree. You may need to make additional changes.

Using the same testing strategy as before, retest your ***WorkerManager*** program. If it no longer works correctly, correct the code so that it does so. If it no longer adds and removes elements correctly, the problem may be in your binary search tree rather than in the ***WorkerManager***. In what ways, if any, does it behave differently than before? Demo your finished program to the TA or lab instructor when done and explain the changes that you've made.