Lab 9 – Implementing and Using a Binary Search Tree (BST)

Preliminaries

In this lab you will implement a binary search tree and use it in the *WorkerManager* program from Lab 03.

Problem 1: Building and testing your own linked binary search tree

Start by copying this lab folder to your *CS200/labs* directory. Keep test class *BSTTest.java* in your lab folder. Open the file and change all instances of *<FOO> and <foo>* to the appropriate initials.

Now, create an appropriately named (*<YOUR CAPITALIZED INITIALS>LinkedBinarySearchTree*) class in 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).

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

Notice how the generic type `ElementType` specified after the class name has the restriction that it must extend `Comparable<ElementType>`. You should know why this is the case.

Study the online documentation for interface *ZHBinarySearchTree*, found on http://www.users.csbsju.edu/~csweb/ZHStructures/index.html. Next, create copies of all methods from interface *ZHBinarySearchTree* in your newly created class along with a default constructor, and implement them as stub methods. At this point, your class should compile without errors.

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

```
protected class BSTNode extends ZHBinaryTreeNode<ElementType, BSTNode>
```

Again, take some time to study the *ZHBinaryTreeNode* class, which is extended by the *BSTNode* class. (Refer to its documentation at http://www.users.csbsju.edu/~csweb/ZHStructures/index.html.) It defines a generic tree node structure suitable for any binary tree, but not specifically for a binary *search* 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* node of the tree. Most methods in the outer class will makes 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. Consequently, your outer class should contain the following two *private* instances variables: *root* and *size*. *Don't modify these or add any other instance variables.*

Add three constructors to inner class *BSTNode* (a default constructor, one that takes a single `ElementType` parameter and one that takes three parameters). Each of the constructors corresponds exactly to a constructor of the super class *ZHBinaryTreeNode*, having the same parameters with the same types and names. The inner class constructors simply call the corresponding superclass constructors.

Methods of the outer class are all straightforward: The constructor initializes the *root* instance variable to a new empty *BSTNode;* the `isEmpty` and `size` methods use the `size` instance variable in the usual way; the `contains`, `get`, and `iterator` methods just call the corresponding inner class methods on the *root*. The `add` and `remove` methods

are almost as simple, but they need to check the result of the inner class call and increment or decrement `size` respectively if the result is *true*.

There are five *public* methods that need to be implemented in the inner class *BSTNode*: `iterator`, `contains`, `get`, `add` and `remove`; in addition, there are two *protected* helper methods `copyNodeToThis` and `removeAndReturnLeftmost` that are useful for implementing the other methods, but are not part of the *public* interface. Since these methods are recursive and somewhat complex, we'll consider each of them separately later on.

The `iterator` method

The `iterator` method should simply call any one of the iterator methods from super class *ZHBinaryTreeNode*. You should use the in-order iterator because it iterates over the items in increasing order.

The `contains` method

Super class *ZHBinaryTreeNode* has a working generic `contains` method, but it doesn't take advantage of the ordering in the binary search tree, so we shouldn't use it; instead, we need to override it for efficiency. The overridden method should be recursive with two base cases: if this node is empty, it returns *false*; if this 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`.

The `get` method

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

The `add` method

Consider the `add` method next. It has the same recursive case structure as `contains` and `get` methods, but what you do with the cases is different. If this node is empty, it means you've reached the point where the new element should be added. To do this, we convert this node into a non-empty node containing the new element by copying the new element into the node's element field and putting new empty trees 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.

The `remove` method

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 variables 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.
  */

protected void copyNodeToThis(BSTNode otherNode) {
  // fill in your code
}
```

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; it may be this node. 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 auxiliary 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.
  */
protected ElementType removeAndReturnLeftmost() {
  // fill in your code here
}
```

The algorithm for the `remove` method has the same overall case structure as the other methods. If this node is empty, there's nothing to remove, so the method should return *false*. If the comparison shows that the element we're trying to remove is not equal to the element in this node, we recursively remove it from the appropriate left or right sub-tree.

If this node contains an element equal to the element we want to remove, we have found the element we want to remove from the tree. In this case, there are three sub-cases. 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 the `copyNodeToThis` method; similarly, if the right sub-tree is empty, we remove the element by copying the fields of *leftChild* into this node. If neither sub-tree is empty, we have to 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 left sub-tree, and removing it from the sub-tree. We can do this in one step by assigning the value returned by calling `removeAndReturnLeftmost` on the *rightChild* to `this.element`. In all of these three sub-cases, the method returns *true*.

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: Modifying the WorkerManager program to use a tree of Workers

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 all four manager programs from lab 03 into today's lab folder: *WorkerManager.java*, *HourlyEmployeeManager.java*, *SalariedEmployeeManager.java* and *VolunteerManager.java*

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 (i.e. implement) interface *Comparable<ElementType>*. We must therefore make the *Worker* hierarchy implement this interface. There are two changes required: First, change the *Worker* interface so that it extends *Serializable* as well as *Comparable<Worker>*. Second, add *public equals* and *compareTo* methods to the *HourlyWorker* and *SalariedEmployee* classes, such that two workers are equal if they have the same name, and the *compareTo* method returns the result of comparing the two names.

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.

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.