CMSC 132 Project #3 Fall 2012

Date due: Monday, November 12, 10:00:00 p.m.

1 Introduction

In this project you will implement a generic binary search tree; in particular it will be a polymorphic binary search tree, consisting of the classes `EmptyTree` and `NonEmptyTree`. We provided an example of a (nongeneric) polymorphic linked list when trees were covered, which you should first study carefully. It is also important to study "regular" (non–polymorphic) binary search trees to ensure you have a good understanding of them before starting to code. Note that the `EmptyTree` class you will implement must use the singleton design pattern, discussed in class, while the `EmptyList` class in the polymorphic linked list example did not.

One purpose of the project is to get experience implementing a binary search tree, but note that various methods of the tree will need to be implemented recursively, so another purpose is to get more experience using recursion with data structures. Some of the methods may not be that useful or applicable in a practical program; in other words there is no real reason for a few of them other than to get more practice with the concepts just mentioned. Because the concepts involved in using a polymorphic data structure are important, we need to ensure that you code the project correctly, otherwise you won't be using and learning the things you should be. That means that there are a larger than usual number of things that you aren't allowed to do listed in Section 3. Be sure to read these carefully in advance, and if you have any questions be sure to ask, so you will be able to do things right.

2 The binary search tree

The binary search tree classes are generic classes with two type parameters; `K` is used for keys, which must be of a type that is comparable with itself, and `V` is for values. In particular, the tree associates each value with a key, so a value will be looked up using that key (the key may be of the same type, or even have the same value as the value, or the types of keys and values for any tree may be different). Note that the methods that have to search in trees (for keys) have no idea whether the element type that the key parameter `K` has been instantiated with has overridden `equals()` or not. If not, then comparing keys using `equals()` when searching will just do reference comparison, which probably isn't what's needed. But all of the methods know that the key type **must** implement the `Comparable` interface, due to the type parameter `<K extends Comparable<K>>`.

Some methods should throw exceptions in certain situations, as described below. It doesn't matter what string or message the exceptions are constructed with, or even if they have a message string at all.

2.1 The `Tree` interface

This interface describes the functionality of both empty and nonempty search trees. Since `Tree` is an interface it has no methods to be written, however the effects of the methods are explained here since they must be written in both classes that implement `Tree`. Note that trees in this project are binary search trees, so all methods must be sure that the binary search tree property is maintained at all times.

Since this is a polymorphic tree, there is no inner `Node` class as there was with the list classes in your prior project. Instead, trees will consist of objects of the classes `EmptyTree` and `NonEmptyTree`, which both implement the `Tree` interface. Methods that return a value of type `Tree` (or `NonEmptyTree`) will usually (except where noted) return a reference to their current object (which will in many cases have been modified by the method), to allow chained method calls. But in some cases they will have to return a reference to a different object. See the discussion below, and the polymorphic list example for illustration.

2.1.1 `NonEmptyTree<K, V> add(K key, V value)`

This method should add the key `key` to the current object tree, associated with the value `value`. However, if the key `key` is already present in the tree the method should instead **replace** the value that was previously associated with it with the new value `value`; the rest of the tree should be unchanged. Note that this method

should always succeed, unless the program encounters an error due to running out of memory, which in normal usage should not occur.

This method returns a reference to a `Tree`. In many cases– but not all– it may just return a reference to the (modified) current object tree. However, in some cases it can't. For example, calling `add()` to add a key/value pair to an `EmptyTree` object must return a `NonEmptyTree` containing the key associated with the value. Note that when using or calling this method you have to pay attention to the return value– if you simply invoke `add()` on a `Tree` and ignore the return value your code is wrong.

2.1.2 `V lookup(K key)`

This method should find the value that the key `key` is associated with (meaning the key in the tree that compares identically to `key`. It should return a reference to that value, or `null` if `key` is not associated with any value in the current object tree. (This is like a search or find operation, but we named it differently because it's not just looking for a key, it's looking up that key and then returning the associated data item if the key was found.)

2.1.3 `int size()`

This method should return the number of key/value pairs that are present in the tree it's called upon; the result will be zero (for an empty tree) or more.

2.1.4 `K max() throws EmptyTreeException`

This method should return a reference to the key that is largest in the tree it's called upon. If the tree it's called upon is empty it should throw an `EmptyTreeException`; this is a checked exception that we defined, which you should not modify (it is in fact empty), that will be used to signal that a tree has no maximum or no minimum element because it has no contents.

2.1.5 `K min() throws EmptyTreeException`

This method should return a reference to the key that is smallest in its current object tree, but if the tree it's called upon it empty it should throw an `EmptyTreeException`.

2.1.6 `Tree<K, V> delete(K key)`

This method should remove the key `key` from the tree that it's called upon, along with the value that is associated with it. Note that due to the effects of the `add()` method described above trees will never have duplicate keys, so there will be at most one key matching `key`.

This method returns a reference to a `Tree`. In many cases– but not all– it may just return a reference to the (modified) current object tree. However, in some cases it can't. Note that when using or calling this method you have to pay attention to the return value– if you simply invoke `delete()` on a `Tree` and ignore the return value your code is wrong. If `key` isn't present in the tree the method should have no effect, meaning that it should return a reference to its unmodified current object.

This method **may not** remove the indicated element by creating a new tree and inserting all the key/value pairs from the original tree except the one with key `key`. Although there are several deletion algorithms that could be used to remove an element from a binary search tree only one of them was explained in lecture, so it's recommended that you use that one.

2.1.7 `boolean haveSameKeys(Tree<K, V> otherTree)`

This method should return true if its current object tree and `otherTree` have exactly the same keys, and false otherwise. In other words, the two trees must have **all** the same keys, and **only** the same keys. It doesn't matter whether their values are the same or not. **Note**: the keys (and values) do **not** have to have been added to the trees in the same order, and the trees do **not** have to have the same shape in order for the method to return true– all that matters is whether they have the same keys. For example, one tree could have three

keys 20, 30, and 10 added in that order (with some associated values), and another tree could have the same keys added in the order 30, 10, 20 (with some associated values), and another tree could have the same keys added in the order 10, 20, and 30 (again with some associated values)– if you draw the trees all three will look different, but this method should return true if called on any pair of them. But if another tree has the keys 10, 20 and 40, then this method would return false if called on it and any of the ones above.

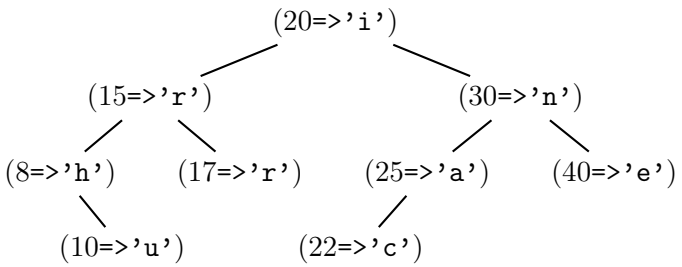
2.1.8 `Tree<K, V> removeSubTree(K key)`

This method should remove the key `key` with its associated value, **and** its entire subtree, from the tree that it's called upon. If `key` is the key at the root of the tree then the tree will become empty as a result. This method returns a reference to a `Tree`. In many cases– but not all– it may return a reference to a modified version of the current object tree (with the subtree having been removed), but in some cases it must return a reference to something different. If `key` isn't present in the current object tree then the method should have no effect, meaning that it should return a reference to its unmodified current object.

2.1.9 `void pathFromRoot(K key, List<K> list)`

If the key `key` is present in its current object tree this method should **remove** any current contents of its parameter `list` (which will be some type implementing Java's `List` interface, such as an `ArrayList` or a `LinkedList`), and cause it to contain the list of all of the keys in the tree between the root and `key`, in that order. If `key` is the key at the root of the tree then the list will just contain that single key. If `key` is not present in the tree at all then the original contents of `list` should **not** be modified at all.

For instance, consider the following example tree, where the key is the first element of each key, value pair shown (in this tree keys are integers and values are characters). Note that keys are unique in a tree, but values need not be.



If `pathFromRoot(17, list)` was called on this tree then `list` would subsequently contain the keys 20, 15, and 17, in this order. After `pathFromRoot(22, list)`, `list` would contain 20, 30, 25, and 22, while after `pathFromRoot(20, list)`, `list` would just contain 20.

2.1.10 `void pathToRoot(K key, List<K> list)`

This method should return the reverse of the list that `pathFromRoot()` would return. As above, if `key` is in its current object tree then `list` should be cleared first. For example, if `pathFromRoot(22, list)` was called on the tree above then `list` would subsequently contain 22, 25, 30, and 20, in this order.

This method **cannot** just call `pathFromRoot()` and reverse the list that it produces. It must be written from scratch to return the list described. It should not be that different from `pathFromRoot()`, once you understand how to write that method.

2.1.11 `Tree<K, V> clone()`

The `Tree` interface implements the `Cloneable` interface (when one interface implements another in Java, the word `extends` is used). The `clone()` method must create and return a reference to a new `Tree` object whose **contents and shape** are exactly the same as its parameter object's (see below what we mean by that). It must do this **without** calling a copy constructor, firstly because the Java API for `Cloneable` says that no constructor is called, and, secondly, because the tree classes don't have copy constructors (if you were to try to write copy constructors for the tree classes you might have difficulty).

The returned object and the current object must be completely **independent**, so that subsequently changing either one of them will have no effect on the other one. However, notice that when we say "a new `Tree` object" we mean that the newly–created tree should not share any `Tree` objects with the current object, but it will not actually be a full deep copy– the actual keys and values will in fact be aliased. This is because any references in trees to the stored keys and values will be declared using generic type parameters, and it's not possible in a generic class to instantiate an object of a generic type parameter. Therefore, after making a clone of a tree using this method, it should be possible to add or remove elements from one tree without affecting the other tree's elements, but if mutable objects are being stored in trees, and after this method is called a method is called on an object in either the original tree or the cloned tree that changes its state, it will be changed in both trees.

(Note that this method uses covariant return types, so the return type of `clone()` in the two tree subclasses are their own class type, which are both subclasses of the superclass method's return type.)

2.2 The `EmptyTree` class

This class is used to represent the empty binary search tree, meaning the tree that contains no elements or key/value pairs. This class is a singleton class; since all empty search trees are the same, there is no need for multiple instances of it. Instead, your class **must** create a **single, shared instance** of it, and any constructors that you write in this class must be **private**, to prevent other code from mistakenly creating additional instances of the class.

Because of the way that generics are implemented in Java a generic class' type parameters cannot be used in any of its static fields, methods, static nested types, or static initializers, because static methods are shared among all instances of a class– even ones that use different types to instantiate the type parameter with. Therefore, when you instantiate the single instance of the class, it cannot use the type parameters `K` and `V`. Although omitting them will generate a compilation warning, you may suppress it using the annotation `@SuppressWarnings("unchecked")`,

Note that since an `EmptyTree` doesn't actually contain any data, the same singleton can be used for any kind of empty tree, regardless of the actual types used for `K` and `V` in instantiating it.

2.2.1 `static <K extends Comparable<K>, V> EmptyTree<K, V> getInstance()`

This method should return a reference to the one and only unique instance of this class. Note that `<K extends Comparable<K>, V>` means that this method is a generic method (like the ones in Homework #2). As above, static methods of a generic class can't use the class' type parameters, But since we need to use the type parameters for the method's return value, the solution is to make the method a generic method (that happens to be inside a generic class).

2.2.2 `String toString()`

This method will override `toString()` in the object class. What it will need to do is implied by the description of `toString()` in the `NonEmptyTree` class explained below.

2.3 The `NonEmptyTree` class

This class represents a nonempty binary search tree. An instance of this class should contain:

- a key,
- a value (that the key is associated with),
- a reference to a left `Tree` that contains key/value pairs such that all keys in the left `Tree` are less than the key stored in this tree object, and
- a reference to a right `Tree` that contains key/value pairs such that all keys in the right `Tree` are greater than the key stored in this tree object.

2.3.1 `String toString()`

The string returned by this method must consist of all of the key/value pairs that are stored in its current object tree (which will be zero or more), with the key and value of each pair separated by the symbols `=>` (with no internal or intervening space). The key/value pairs must appear in the string in **increasing order** according to their keys. Each key/value pair containing `=>` must be separated from the next one in the sting by a single blank space, but no space should appear before the first pair or after the last pair. (**Note**: your method should **not** add a blank space after the string representation for every key/value pair and then remove the last one; it should instead just not add a blank space after the last one.) For example, the string returned for the tree above would be (blank spaces are shown as ␣): "`8=>h␣10=>u␣15=>r␣17=>r␣20=>i␣22=>c␣25=>a␣30=>n␣40=>e`".

3 Project requirements and related issues

1. **Carefully read** the following design and implementation restrictions prior to coding. Not following these requirements may result in losing credit on the project. If you're not sure how to code any of the methods given these limitations, ask in office hours.
   - You **must** use a polymorphic tree. This means that there will **not be any `null` references anywhere in a tree**. Trees will be composed entirely of `EmptyTree` and `NonEmptyTree` objects linked together.
   - Your `EmptyTree` class **must** be a singleton class.
   - You may **not** explicitly check a `Tree` to see whether it is an `EmptyTree` or `NonEmptyTree`. This means that you **may not use `instanceof` or `getClass()` anywhere in your code**. Instead. you must use polymorphism (and exception handling, where appropriate) to handle the differences between empty and nonempty trees. You will need to write versions of every method in both the `EmptyTree` and `NonEmptyTree` classes, that each have the necessary behavior for the type of tree that that class represents.
   - You **may not** perform any casting anywhere in your code, other than the `clone` methods. If you are casting you are not using generic programming correctly.
   - You **may not** check whether a tree is empty by using comparisons like the following:
     - `if (left == EmptyTree.getInstance())`
     - `if (tree.size() == 0)`
     - or other comparisons similar to these.

     As above, you should have versions of methods in both the `EmptyTree` and `NonEmptyTree` classes, that work properly for their respective type of tree.
   - Your `EmptyTree` and `NonEmptyTree` implementations **may not** perform any comparisons against `null` for the purpose of determining whether a tree or subtree is empty or not.
   - You **may not** use `Collections.sort()` anywhere in this project.
   - Recursive methods **may not** use fields to communicate values or information between recursive calls. They should use parameters if this is necessary.
2. Other than the following you **may not** use any arrays, or any Java library classes:
   - You may declare `String` objects in the `toString()` methods, perform string concatenation on them, and do comparisons on `String`s, but no other `String` class methods may be used.
   - In the `pathFromRoot()` and `pathToRoot()` methods you may use methods of the `List` interface to remove the elements of their `List` parameters, and to add new elements to them, but you may not call any Java library sorting algorithm, nor write a sorting algorithm yourself.

3. You may add fields to the classes in the project, and methods to the interface and classes, including constructors (subject to the constraint that the `EmptyTree` class must be a singleton class), but new interfaces or classes should not be added.

   If you want or need to use helper methods, they're probably going to be called upon one or both children (subtrees) of an element in a tree. The two children of a `NonEmptyTree` must both be `Tree` references, since they may be `EmptyTree`s or `NonEmptyTree`s. Therefore any methods that are added probably must be added to the `Tree` interface and written in both the `EmptyTree` and `NonEmptyTree` classes, so they may be called upon the children of a tree regardless of whether they are `EmptyTree`s or `NonEmptyTree`s. Note that as a result you **are** allowed to add new methods to the `Tree` interface, but do not change the signatures of any of the existing methods, otherwise your program will compile for you but not for the public tests or secret tests when it's submitted.

4. Although in lecture (and on the exam) we said that exceptions should not be used for "normal" processing, and only for handling errors, this project is an exception; exceptions may (and must) be used in certain places (for instance, see the provided polymorphic list example).

5. Methods that have objects as parameters should throw a `NullPointerException` if an object parameter is `null`. It doesn't matter what string or message the exception is constructed with, or even if it has a message string at all. (Note that you may not really have to do anything special to cause this to happen.)

6. Your methods should be **efficient** in that they should use the binary search tree property to **avoid** traversing parts of trees unnecessarily. For example, the `lookup()` method should **not** traverse an entire tree looking for the indicated key– this would work but is no more efficient than using an unsorted list.

7. Grades for this project will be based on:

public tests	50 points
secret tests	30 points
student tests	10 points
source code and programming style	10 points

8. As in the earlier programming coursework your student tests must be in the Java class file `StudentTests.java`, which must be in the `tests` package. Your submission **must** have the exact same package structure as in the code distribution initially given, and use the class names specified.

9. None of the restrictions above apply to your student tests; you may use any Java classes or language features that you like in writing them. Do not rely on the public tests to test your methods exhaustively, and don't wait until you've finished coding to write your student tests. Be sure to test each method **as soon as you write it**, or as soon as possible (some methods need others to be written in order to be able to be tested).

10. The course project grading policy describes how projects will be graded, what good style consists of, the late policy, and other important concepts. Be sure to carefully read this information.

11. Do **not** use the submit server's web submission procedure to submit a jarfile or zip archive; you must submit your project directly from Eclipse.

12. **Do not wait** until the last minute to finish and submit your project. Start working right away!

13. After you submit you must **log into the submit server** and verify that your submission compiled and worked correctly there.

14. Recall that the course project grading policy says that all your projects must work on **at least half of the public tests** (by the end of the semester) in order for you to be eligible to pass the course. See the project grading policy for full details.

4 Academic integrity statement

Please **carefully read** the academic honesty section of the course syllabus. **Any evidence** of impermissible cooperation on projects, use of disallowed materials or resources, or unauthorized use of computer accounts, **will be submitted** to the Student Honor Council, which could result in an XF for the course, or suspension or expulsion from the University. Be sure you understand what you are and what you are not permitted to do in regards to academic integrity when it comes to project assignments. These policies apply to all students, and the Student Honor Council does not consider lack of knowledge of the policies to be a defense for violating them. Full information is found in the course syllabus– please review it at this time.