Lab 11 – Creating and Using Maps

In this lab, you will implement a *BookCollection* application using Java's *HashMap* class. You'll then create and test your own tree-based map and use it in your *BookCollection* application.

Copy the lab directory into your *CS200/labs*; it contains a data file called *sampleLibrary.BookCollection* along with four java files: *BookCollection.java*, *BookDescription.java*, *TreeMapTest.java* and *InnerIteratorForTreeMap.java*. You'll use the code in the latter class as an inner class in your map class. For now, keep all files in your lab folder.

Problem 1: Building the BookCollection application

The core of the book collection will be a *HashMap<String, HashMap<String, BookDescription>>* which will associate an author (a *String* key of the outer map) with a map of titles (a *String* key of the inner map) and descriptions (a *BookDescription* value of the inner map). Notice that we're dealing with nested maps here (a map used as the value of another map) ... and you thought you've seen it all!

First, you'll develop and test the *BookCollection* class; the next step would be to develop an application that manages the collection. For your maps, you'll use Java's *Map* interface and *HashMap* class, which are documented here on the official Java API pages.

Part 1.1: The BookDescription class

Complete the *BookDescription* class which should contain, at a minimum, the following instance fields:

author–the author

title –the title of the book

publisher–the name of the publisher

date–the year of publication

rating–your own single-character rating system for the books

pages–number of pages

description–a description of the book

You will want at least one constructor as well as accessor (`get`) methods for *every* instance fields, a `toString` method, a `setDescription` method and a `setRating` method. The main constructor will have parameters for all the book fields; you may add other constructors with fewer parameters if you decide they will be useful. The `toString` method should format the string to include newline characters so that the string format matches the format of the *BookCollection* file (see below). You may choose to have more than the specified mutator (`set`) methods, but only for fields that might change over time; the author and title of a book presumably never change, so those fields should not have mutator methods.

Part 1.2: The BookCollection class

Once you have the *BookDescription* class complete, move on to the *BookCollection* class which contains the following instance variable:

```
private HashMap< String, HashMap <String, BookDescription>> library;
```

BookCollection objects consist of *BookDescription* objects that are accessed by *author* and *title*. That is, the `library` instance variable maps authors to their books map; and for each author, the books map maps titles to their descriptions. Consequently, to get the description of a particular book, you first have to get the books written by the author and then get the title from among those books.

Your *BookCollection* class should have methods that enable you to:

- create a new empty collection–you need a default constructor

- create a new collection by reading books from a specified file formatted as described below–you need another constructor with a *String* filename as a parameter
- iterate over all authors–the method may return an *Iterator<String>* that yields author names, one-by-one (note that the outer map already has such a method)
- iterate over all titles for a given author–the method may return an *Iterator<String>* that yields title names, one-by-one, for the given author (again, note that the inner maps already have such a method; if there is no entry for the specified author, the method should return *null*)
- get a particular book description for a given author and title
- add and remove books
- update the description of a book for a given author and title
- write the collection to a file formatted as described below
- check if a book collection is empty
- get the size a book collection (along with a size instance variable that counts the total number of books in the collection)

In file *sampleLibrary.BookCollection*, you are given a sample of text book descriptions in the format appropriate for this lab. Your file constructor should be able to read this file and should read any file with that format. The *write* method should produce files in the same format. The *toString* method of the *BookDescription* class should facilitate this process.

Part 1.3: The driver program

Create a driver program to test your class thoroughly. You may use driver programs from earlier labs as a model BUT YOU ARE EXPCTED TO BE VERY THOROUGH IN YOUR TESTING. Specifically, I want to see adequate output from your driver program to demonstrate the following functionalities:

- create a collection by reading books from the provided file *sampleLibrary.BookCollection*
- use the collection to iterate over and display all author names
- use the collection to iterate over and display the books for each author
- update the description of one of the books for any of the authors–then, display all books for that author to show the updated value
- add a new book to your collection–then, use the updated collection to iterate over and display the books for all authors
- write the collection to a new file–check that file has been written properly
- remove a book from your collection–then, use the updated collection to iterate over and display the books for all authors

Problem 2: Implementing and testing a tree-based map class

Recall that a map is a set of <*key, value*> pairs; every key in the map is unique, but the same value may be associated with more than one key. In this problem you will implement a tree-based map class using your *<FOO>LinkedBinarySearchTree* class from Lab 09. In other words, you'll make a binary search tree object store <*key, value*> pair elements and behave like a map (but not a hash map).

Part 2.1: Creating the tree map class

Keep file *TreeMapTest.java* file in your lab folder. Open it and change all instances of *<FOO>* and *<foo>* to the appropriate uppercase or lowercase initials you are using. Rename the iterator file (*InnerIteratorForTreeMap.java*) to *<FOO>TreeMap.java* using your initials in place of *<FOO>* and move the file in your *<foo>structures* directory. Inside this file, create an outer class with the following header (place the outer class above the *MapIterator* inner class):

```
public class <FOO>TreeMap<KeyType extends Comparable<KeyType>, ValueType>>
      implements ZHMap<KeyType, ValueType>
```

Study the *ZHMap* interface available online at *http://www.users.csbsju.edu/~csweb/ZHStructures/index.html* . Your class should have one instance variable called `innerTree` of type *ZHBinarySearchTree< ZHComparableKeyPair <KeyType, ValueType>>*. Its only constructor should create an empty map, so it doesn't need any parameters, but it should initialize the `innerTree` to be an empty *<FOO>LinkedBinarySearchTre* (use your initials instead of *<FOO>*).

Add all the methods needed to satisfy the *ZHMap* interface. The documentation for the *ZHMap* interface shows these methods.

Part 2.2: Completing the tree map class

Now, you will complete the code for the tree map class so that it correctly implements the specified methods.

Several of the methods need to deal explicitly with the *ZHComparableKeyPair* class; please study the *ZHComparableKeyPair* documentation (http://www.users.csbsju.edu/~csweb/ZHStructures/index.html). In a nutshell, this class allows us to treat a (*key,value*) pair as a single object which we'll store as elements in the `innerTree` instance variable.

Here are brief descriptions of how these methods should work:

- `isEmpty` and `size`: These methods should call corresponding methods on the `innerTree` instance variable.
- `iterator`: This method should just return a new instance of the inner *MapIterator* class that was supplied with this lab. You can see that the code for the iterator is straightforward; it just wraps a *KeyType* iterator around a *ZHComparableKeyPair* iterator associated with the `innerTree` instance variable allowing us to iterate over the keys in the map.
- `contains`, `get` and `remove`: These methods should all construct a *ZHComparableKeyPair* object with the desired *key* and a *null value*, and then call corresponding methods on the `innerTree` instance variable with the created *ZHComparableKeyPair* object as parameter; values returned by calls on the `innerTree` instance variable ultimately decide what these methods return.
- `put`: This method first constructs a *ZHComparableKeyPair* with the specified *key* and *value*, and then calls the `innerTree's` *get* method; depending on the value returned by the *get* method, the method should either (1) call the tree's *add* method and returns *null* OR (2) call the returned pair's *setValue* method with the specified new value and returns the old value.

Part 2.3: Testing the tree map class

Once you have a complete tree map class, use the supplied test program to test your code. Show your fully-documented code and the testing to the lab instructor or TA before you leave the lab or at the beginning of the next lab.

Problem 3: Using your map in the BookCollection application

Make copies of the files from the *BookCollection* application from *Problem 1* and rename them in order to distinguish them from the previous version. Open the renamed files and make the necessary changes so that your application now uses your map class instead of java's *HashMap*. In general, you'll need to import the *zhstructures* and *<foo>structures* packages, change the interface names to *ZH* interfaces and change the class names to *<FOO>* classes.

Test your changes to make sure that your new *BookCollection* application still works. It should be able to read the same files that the original read or wrote and to write files that are readable by the original version.

Have fun.