CS-260	Lab #5: Hash Tables (Spellchecker)

General

For this lab you will further your studies of Abstract Data Types (ADT's) by coding a hash table structure using open hashing (separate chaining). The basic assignment for this lab is to implement a hash table ADT, and then a version of the spell checker that uses the hash table for the dictionary. You are to do this with as minimal changes as possible to the existing code by replacing the use of the "TestList" class with your own implementation of a hash table class. This can be done with only changing the changing the dictionary field from "TestList" to your container class.

As with all of my labs, you can enhance this in any way (maybe add a GUI interface) for extra credit.

Turn in your complete source, project files, directory structure, etc. of your ECLIPSE project directory (use the EXPORT tool in Eclipse to export the entire project directory). This zipped file should be turned in using the same method and done for previous labs.

General lab requirements:

At the beginning of your .java files use comments that include your name, creation date, Project #5, and purpose of this .java file.

Each public class and method should have a JavaDoc formatted comment specifying at least what the method will do. These comments can be fairly short at this point (one sentence). Methods that are implementations of interfaces that are already documented only need the @Override tag as they will get the JavaDoc from their interface definition. You should additionally have inline comments at every point in your program when the purpose is not obvious from the code itself.

Your indentation should be consistent. You should indent once for each block level (code between braces), and reduce the amount of indent used after the end of each code block. Use the Eclipse code formatter to clean up your code!

You should use EXCEPTION HANDLING when ever it makes sense to do so. For classes that implement standard java interfaces, use the javaDoc's for the interface as a guide for deciding which exceptions should be handled for each interface method.

Concepts

This lab will introduce you to Java programming techniques for a hash table ADT and gives you additional experience with parts of the Java Class Hierarchy. **Take a look at the Java *HashSet* class. You are to write a new class that behaves similarly to this class.**

Background

This Read the entire "Hash Tables" section of the online text. Study closely the sections on "Open Hashing (separate chaining)".

Assignment

5a. Do the following:

Write a new **Hash table** class that IMPLEMENTS the Java ***Set*** interface. Further, your container must be implemented as a generic container that can accommodate any object type by using Java generics. Your implementation must be a hash table that resolves collisions using separate chaining. To get full credit you must use your own double-linked list class to implement the "buckets" for each table entry. You do not have to implement all of the public methods, but you must implement at least the following:

add(O), clear(), contains(O), remove(0), isEmpty(), iterator(), and size()

Consider the other methods a "challenge" exercise; but notice I am requiring the remove method to be implemented for this lab.

Important Notes:

1. Name your class "OpenChainHashSet" so that it can be tested with the jUnit tests that I have provided. It will need to implement both the java "Set" interface, and my provided "CompareCount" interface. As hashing is NOT a comparison ordered structure, the parameterized generic type can be any class type and does not need to extend Comparable. Here is the class definition header that you must use:

   ```
   public class OpenChainHashSet<E> implements Set<E>, CompareCount
   ```

2. Your OpenChainHashSet class must have a constructor that has an int parameter "tableSize" that will define the initial number of buckets to be used in your hash table. "tableSize" is the number of individual buckets (array elements), NOT the number of items in the hash table. Use a separate field to track the current number of items in the table.

   Since a load factor of about 75% is a good place to start, and we have a dictionary with about 110,000 words, a prime number "size" value that you could try would be 150,001. You should also test some smaller sizes. Here are some addition prime tableSizes to try: 7,919; 25,013; 49,999; 150,001

3. Create an array with "tableSize" number of separate lists (the buckets) using your double linked list ADT (if your list is not working, you can use LinkedList, but this will be a deduction). **You are creating an array of linked lists for your open chained hash table.**

Java versions 1.5 – 1.7 have an odd **restriction on creating Arrays of generic types** even thought these are just references. Even Sun/Oracle the java library code has to do casting around this restriction. So you create an array of the non-parameterized type and then cast it. Here is how you do this:

```
//In your field declaration section
private DLList<E>[] hashTable;

//Then in the constructor is the memory allocation for the array
hashTable = ( DLList<E>[]) new DLList[ tableSize];
```

4. You are not required to do dynamic resizing of your hashtable, this is an optional challenge exercise (not for any credit) if you wish to tackle this. Your individual buckets are not size limited.

5. Your OpenChainHashSet class must use the hashCode( ) method that is already defined for each class in Java. Since the hashCode( ) value will generally include negative values, you should take the absolute value of this number. To find the bucket that the "item" should be stored in, you would use:

```
int index = Math.abs( item.hashCode( )) % tableSize;
```

   Add the item to the bucket list at location "index" using the add method of your linked list class. Note that the list object may not yet exist at any given location, so you need to check for a null at the location, and create a new list object if the array location is null.

6. Your contains method would use the same logic as shown in #5 to find which bucket that item should be in; then it will use linear search on that list using your contains method that is already written for your list ADT.

7. Remove will work similarly, find the bucket (list) that the item should be located in, and then call your DLList remove method to actually remove the item.

8. Your OpenChainHashSet "contains" method must set a field variable called something like compareCount to 0 on each call to contains. The only compares that are done are in the linear search done on the hash table bucket. So simply have the "contains" method set the compareCount field to the number of compares done by the list "contains" method.

9. The iterator is very challenging to write correctly since you have MULTIPLE lists to walk through correctly and track when you are done and what is next to return. Think through your structure closely to understand the issues.

As long as you have a good start on implementation of your hash table structure and the above methods, you will get full credit the 5a lab grade.

5b. Do the following:

1. Get your hash table fully functional and tested. Have your hash table class also implement the "CompareCount" interface so that it will work with the SpellChecker project files. Comparison counting should be done at minimum on the "contains(0)" method. You should have previously implemented this for your double-linked list class.

2. Now that you have a functional hash table class, edit the Dictionary class of the spellchecker and replace the use of the "TestList" class with your hash table ADT. The program should produce the same results as with the AVL tree version. It should be interesting to compare the run times of the AVL version and your hash table version. Try setting the tableSize to several of the different prime number values that I suggested.

Submission

Turn in your complete source, project files, directory structure, etc. of your ECLIPSE project directory. Please **export** the entire project using the Eclipse export utility). This zipped file should be turned in using the same method and done for previous labs.