CMPS 12B
Introduction to Data Structures
Programming Assignment 3

The goal of this project is to implement a *Dictionary* ADT based on the linked list data structure. The elements of the Dictionary will be pairs of Strings called *key* and *value* respectively. Keys will be distinct, whereas values may be repeated. Thus any two (*key*, *value*) pairs must have different keys, but may possibly have identical values. You can think of a key as being an account number, and a value as say an account balance, both represented by Strings. Recall that an ADT consists of two things: (1) a collection of states, and (2) a collection of operations which act on states. In the Dictionary ADT a state is simply a set of (*key*, *value*) pairs. There are seven ADT operations which are to be implemented by the methods below.

`public boolean isEmpty()`
Returns true if the Dictionary contains no pairs and returns false otherwise.

`public int size()`
Returns the number of (*key*, *value*) pairs in the Dictionary.

`public String lookup(String key)`
If the Dictionary contains a pair whose key field matches the argument key, lookup returns the associated value field. If no such pair exists in the Dictionary a null reference is returned.

`public void insert(String key, String value)`
If the Dictionary does not currently contain a pair whose key matches the argument key, then the pair (*key*, *value*) is added to the Dictionary. If such a pair does exist, a `DuplicateKeyException` will be thrown with the message: "`cannot insert duplicate keys`". Thus `insert()` has the precondition that the Dictionary does not currently contain the argument key. This precondition can be tested by the client module by doing `lookup(key)==null`.

`public void delete(String key)`
If the Dictionary currently contains a pair whose key field matches the argument key, then that pair is removed form the Dictionary. If no such pair exists, then a `KeyNotFoundException` is thrown with the message: "`cannot delete non-existent key`". Thus `delete()` has the precondition that the Dictionary currently contains the argument key. This precondition can be tested by the client module by doing `lookup(key)!=null`.

`public void makeEmpty()`
Resets the Dictionary to the empty state.

`public String toString()`
Returns a String representation of the current state of the Dictionary. Keys will be separated from values by a single space, and consecutive pairs will be separated by newline characters. The return String will be terminated by a newline character. Pairs will occur in the return String in the same order they were inserted into the Dictionary.

Implementation of the Dictionary ADT
An interface file for the Dictionary ADT (`DictionaryInterface.java`) with prototypes for the above methods will be provided on the class webpage. The implementation file for the Dictionary ADT, which you will write, will be called `Dictionary.java`. In it you will define the Dictionary class and explicitly implement the interface, i.e. the class heading will be:

```
public class Dictionary implements DictionaryInterface
```

You will turn in the interface file with your project, but you are not to alter the contents of that file in any way (don't even put in the customary comment block with your name). In addition to the implementation file you will also write files `DuplicateKeyException.java` and `KeyNotFoundException.java` which define the two types of exception classes to be thrown. Make both of these exceptions to be subclasses of RuntimeException, and follow the examples given in lecture and on the webpage.

Your implementation of the Dictionary ADT will utilize a linked list data structure. The linked list may be any of the variations discussed in class (e.g. singly linked with head reference, singly linked with both head and tail reference, circular, doubly linked, with or without dummy node(s), or any combination of the preceding types.) It is recommended that you take the linked list representation of the IntegerList ADT as a starting point for this project. Just rename that file and start making changes to it. In particular your `Node` class must be a private inner class to the Dictionary class. However, your `Node` class will no longer contain an `int` field since the data stored at each node will be a pair of Strings. You have two options on how to store pairs. The first option is to let the Dictionary class contain another private inner class called `Pair` that encapsulates the two Strings with fields called `key` and `value` respectively. The data field in your `Node` will then contain a reference to `Pair`. The second (simpler) option is to define your `Node` class to contain two String fields in addition to its `Node` reference field(s). It is recommended that your Dictionary class contain a private method with the following heading.

```
private Node findKey(String key)
```

This method should return a reference to the `Node` containing its argument `key`, or return `null` if no such `Node` exists. Such a method will be helpful in implementing the methods `insert()`, `delete()` and `lookup()`.

Testing

Create another file called DictionaryTest.java whose purpose is to serve as a test client for the Dictionary ADT while it is under construction. This file will define the class DictionaryTest, which need not contain any more than a `main()` method (although you may add other static methods at your discretion.) The design philosophy here is that an ADT should be thoroughly tested in isolation before it is used in any application. Build your Dictionary ADT one method at a time, calling each operation from within DictionaryTest.java to wring out any bugs before going on to the next method. The idea is to add functionality to the Dictionary in small bits, compiling and testing as you go. This way you are never very far from having something that compiles, and errors that arise are likely to be confined to recently written code. You will submit the file DictionaryTest.java with this project. It is expected that it will change significantly during the testing phases of your project. As that happens, comment out the old test code as you insert tests of more recently written operations. The final version of DictionaryTest.java should contain enough test code (possibly all in comments) to convince the grader that you did in fact test your ADT in isolation before proceeding.

Once you believe all ADT operations are working properly, copy the files `DictionaryClient.java` and `Makefile` to your working directory (both provided on the webpage). At this point `%make` will create an executable jar file in your working directory called `DictionaryClient`. This program will have the following output. (Recall that `%` here represents the Unix prompt.)

```
%DictionaryClient
1 a
2 b
3 c
4 d
5 e
6 f
7 g

key=1 value=a
key=3 value=c
key=7 value=g
key=8 not found

2 b
4 d
5 e
6 f

false
4
true

%
```

If your Dictionary ADT behaves according to specs, your output should look exactly as above. You can check this by doing `% DictionaryClient > out`, copy the file `model-out` from the webpage to your working directory, then do `% diff out model-out`. If `diff` gives no output, then the files are exactly the same, which is good. Note the unix operator `>` redirects program output to a file. In other words it associates the data stream `stdout` with the file on its right hand side, instead of the screen. Similarly the unix operator `<` associates the file on its right hand side with the data stream `stdin`, instead of the keyboard. See the man pages for a description of the `diff` command.

What to turn in

You may alter the provided Makefile to include submit and test utilities, or alter it in any other way you see fit, as long as it creates an executable jar file called `DictionaryClient`, and includes a clean utility. Do not however alter the files `DictionaryInterface.java` or `DictionaryClient.java`. Thus your `pa3` directory of your repository should contain these 8 files:

`README`	table of contents, notes to grader
`Dictionary.java`	created by you
`DictionaryTest.java`	created by you
`DuplicateKeyException.java`	created by you
`KeyNotFoundException.java`	created by you
`DictionaryInterface.java`	unchanged
`DictionaryClient.java`	unchanged
`Makefile`	alter at your discretion

In ecommons submit a pair programming log using one of the templates from https://classes.soe.ucsc.edu/cmps012b/Fall16/assignments/logTemplates.txt being sure to insert the git commit id for your final commit.

As always, start early and ask plenty of questions.