COMP249/2 – Section D – Fall 2015

ASSIGNMENT #3
Due: December 1st

Purpose: The purpose of this assignment is to allow you practice working with different data structure such as linked lists and hash tables.

General Guidelines When Writing Programs:

- Include the following comments at the top of your source codes
```
// ------------------------------------------------------------
// Assignment (include number)
// Question: (include question number)
// Written by: (include you name and student id)
// For COMP249 Section: (Substitute your section letter(s))
// ------------------------------------------------------------
```
- In a comment, give a general explanation of what your program does. As the programming questions get more complex, the explanations will get lengthier.
- Include comments in your program describing the main steps in your program.
- Display a welcome message which includes your name(s).
- Display clear prompts for users when you are expecting the user to enter data from the keyboard.
- All output should be displayed with clear messages and in an easy to read format.
- End your program with a closing message so that the user knows that the program has terminated.

Part 1

A) The **CellPhone** class has the following attributes: a *serialNum* (**long** type), a *brand* (**String** type), a *year* (**int** type, which indicates manufacturing year) and a *price* (**double** type). It is assumed that brand name is always recorded as a single word (i.e. Motorola, Sony, Ericsson, Panasonic, etc.). It is also assumed that all cellular phones follow one system of assigning serial numbers, regardless of their different brands, so no two cell phones may have the same serial number.

You are required to write the implementation of the **CellPhone** class. Besides the usual mutator and accessor methods (i.e. ***getPrice()***, ***setYear()***) the class must have the following:

- Parameterized constructor that accepts four values and initializes *serialNum*, *brand*, *year* and *price* to these passed values ;

- Copy constructor, which takes two parameters, a **CellPhone** object and a **long** value. The newly created object will be assigned all the attributes of the passed object, with the exception of the serial number. *serialNum* is assigned the value passed as the second parameter to the constructor. It is always assumed that this value will correspond to the unique serial number rule;
- ***clone()*** method. This method will prompt the user to enter a new serial number, then creates and returns a clone of the calling object with the exception of the serial number, which is assigned the value entered by the user;
- Additionally, the class should have a ***toString()*** and ***equals()*** methods. Two cell phones are equal if they have the same attributes, with the exception of the serial number, which could be different.

B) The file ***Cell_Info.txt***, which one of its versions is provided with this assignment, has the information of various cell phone objects. Generally, the file may have zero or more records. The information stored in this file is always assumed to be correct and following the unique serial number rule.

C) The **CellList** class has the following:

a. An inner class called **CellNode**. This class has the following:

   i. Two private attributes: an object of **CellPhone** and a pointer to a **CellNode** object;

   ii. Default constructor, which assigns both attributes to null;

   iii. Parameterized constructor that accepts two parameters, a **CellPhone** object and a **CellNode** object, then initializes the attributes accordingly;

   iv. Copy constructor;

   v. *clone()* method;

   vi. Other mutator and accessor methods.

b. A private attribute called *head*, which should point to the first node in this list object;

c. A private attribute called *size*, which always indicate the current size of the list (how many nodes are in the list);

d. Default constructor, which creates an empty list;

e. Copy constructor, which accepts a **CellList** object and create a copy of it;

f. A method called ***addToStart()***, which accepts one parameter, an object from **CellPhone** class. The method then creates a node with that passed object and insert this node at the head of the list;

g. A method called ***insertAtIndex()***, which accepts two parameters, an object from **CellPhone** class, and an integer representing an index. If the index is not a valid index (a valid index must have a value between *0* and *size-1*), the method must throw a ***NoSuchElementException*** and terminate the program. If index is valid then the method creates a node with the passed **CellPhone** object and inserts this node at the given index. The method must properly handle all special cases;

h. A method called ***deleteFromIndex()***, which accepts one integer parameter representing an index. Again, if the index is not valid, the method must throw a ***NoSuchElementException*** and terminates the program. Otherwise; the node pointed by that index is deleted from the list. The method must properly handle all special cases;

i. A method called ***deleteFromStart()***, which deletes the first node in the list (the one pointed by head). All special cases must be properly handled.

j. A method called ***replaceAtIndex()***, which accepts two parameters, an object from **CellPhone** class, and an integer representing an index. If index is not valid, the method simply returns; otherwise the object in the node at the passed index is to be replaced by the passed object;

k. A method called ***find()***, which accepts one parameter of type long representing a serial number. The method then searches the list for a node with a cell phone with that serial number. If such object is found, then the method returns a pointer to that node where the object is found; otherwise, the method returns null. The method must keep track of how many iterations were made before the search finally finds the phone or concludes that it is not in the list; this value indicates the complexity of the algorithm, *O()*;

l. A method called ***contains()***, which accepts one parameter of type long representing a serial number. The method returns true if the an object with that serial number is in the list; otherwise, the method returns false;

m. A method called ***showContents()***, which displays the contents of the list, in a similar fashion to what is shown in Figure 1 below.

n. A method called ***equals()***, which accepts one parameter, an object of **CellList**. The method returns true if the two lists contain similar objects; otherwise the method returns false. Recall that two **CellPhone** objects are similar if they have the same values with the exception of the serial number, which can, and actually is expected to be, different;

➔ Finally, here are some general rules that you must consider when implementing the above methods:

- Whenever a node is added or deleted, the list size must be adjusted accordingly;
- All special cases must be handled, whether or not the method description explicitly states that;

- Whenever an attempt is made to access illegal location in the list, and unless otherwise has clearly been specified, an exception must be thrown and the program terminates;
- All ***clone()*** and copy constructors must perform a deep copy; no shallow copies are allowed;
- If any of your methods allows privacy leak, you must clearly place a comment at the very beginning of this method indicating that this method may result in privacy leak, the reason behind that, and propose other ways that can be used to avoid/eliminate privacy leak. Please keep in mind that you are not required to implement these proposals;

D) Now, you are required to write a public class called **CellListUtilization**. In the *main()* method, you must do the following:

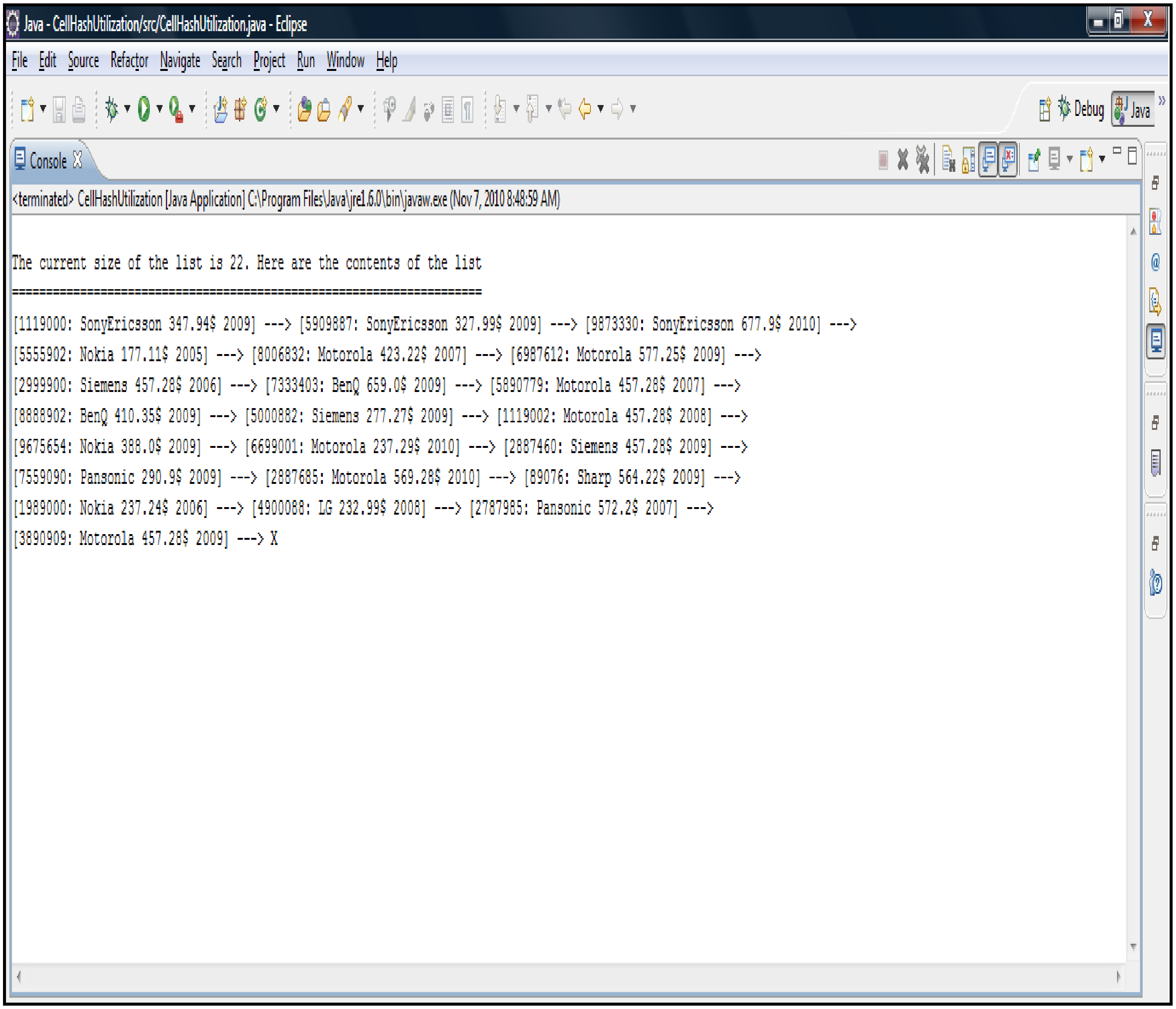
a. Create at least two empty lists from the **CellList** class;

b. Open the ***Cell_Info.txt*** file, and read its contents (line by line). Use these records to initialize one of the **CellList** objects you created above; Simply, you need to use the ***addToStart()*** method to insert the read objects into the list. However, the list should not have any duplicate records, so if the input file has duplicate entries, which is the case in the file provided with the assignment for instance, your code must handle this case so each record is inserted in the list at most once;

c. Show the contents of the list you just initialized;

d. Prompt the user to enter few serial numbers and search the list that you created from the input file for these values. Make sure to display the number of conducted searches;

e. Following that, you must create enough objects to test each of the constructors/methods of your classes. The details of this part are left as an open-ended for you. You can do whatever you wish as long as your methods are being tested including some of the special cases.


Java - CellHashUtilization/src/CellHashUtilization.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Debug
Java
Console

```
<terminated> CellHashUtilization [Java Application] C:\Program Files\Java\jre1.6.0\bin\javaw.exe (Nov 7, 2010 8:48:59 AM)

The current size of the list is 22. Here are the contents of the list
======================================================================
[1119000: SonyEricsson 347.94$ 2009] ---> [5909887: SonyEricsson 327.99$ 2009] ---> [9873330: SonyEricsson 677.9$ 2010] --->
[5555902: Nokia 177.11$ 2005] ---> [8006832: Motorola 423.22$ 2007] ---> [6987612: Motorola 577.25$ 2009] --->
[2999900: Siemens 457.28$ 2006] ---> [7333403: BenQ 659.0$ 2009] ---> [5890779: Motorola 457.28$ 2007] --->
[8888902: BenQ 410.35$ 2009] ---> [5000882: Siemens 277.27$ 2009] ---> [1119002: Motorola 457.28$ 2008] --->
[9675654: Nokia 388.0$ 2009] ---> [6699001: Motorola 237.29$ 2010] ---> [2887460: Siemens 457.28$ 2009] --->
[7559090: Pansonic 290.9$ 2009] ---> [2887685: Motorola 569.28$ 2010] ---> [89076: Sharp 564.22$ 2009] --->
[1989000: Nokia 237.24$ 2006] ---> [4900088: LG 232.99$ 2008] ---> [2787985: Pansonic 572.2$ 2007] --->
[3890909: Motorola 457.28$ 2009] ---> X
```

Figure 1: Sample of Displaying the Contents of a CellList

Part 2

In this part, you will still utilize the same **CellPhone** and the **CellList** classes from Part 1. The following classes are then to be added:

A) A class called **CellHash**, which has the following:

a. A private array, called *hashArr*, of **CellList** objects;

b. A default constructor that initializes the array to have 12 empty CellList list objects. If you wish, you can add a static integer constant in the **CellHash** class to have that size (12);

c. A method, called ***computeHashValue()***, which accepts and uses the serial number of a cell phone object to calculate a hash value between 0 and 11. This computed hash value will determine the array entry (actually the list at this entry) where a **CellNode** (which has that cell phone object) will be added. The details of how this hash value can be computed are left totally up to you, and you should design it in a way that allows such calculation to be made very quickly;

d. A method called ***addToHashTable()***, which reads the contents of the ***Cell_Info.txt***, calculates the hash value based on the serial number, then inserts a CellNode object with the read record into the list at the proper array entry. The addition to the list must not allow any duplicate nodes to be inserted in any of the lists. That is, if the ***Cell_Info.txt*** file includes duplicate entries, which is the case in the given file, only one of these entries is added to the list. Effectively, lists will always include nodes with distinct values.

e. A method called ***displayHashContents()***, which display the contents of the array; that is the contents of each of the lists in the array. Your display should be in a similar fashion to what is shown in Figure 2;

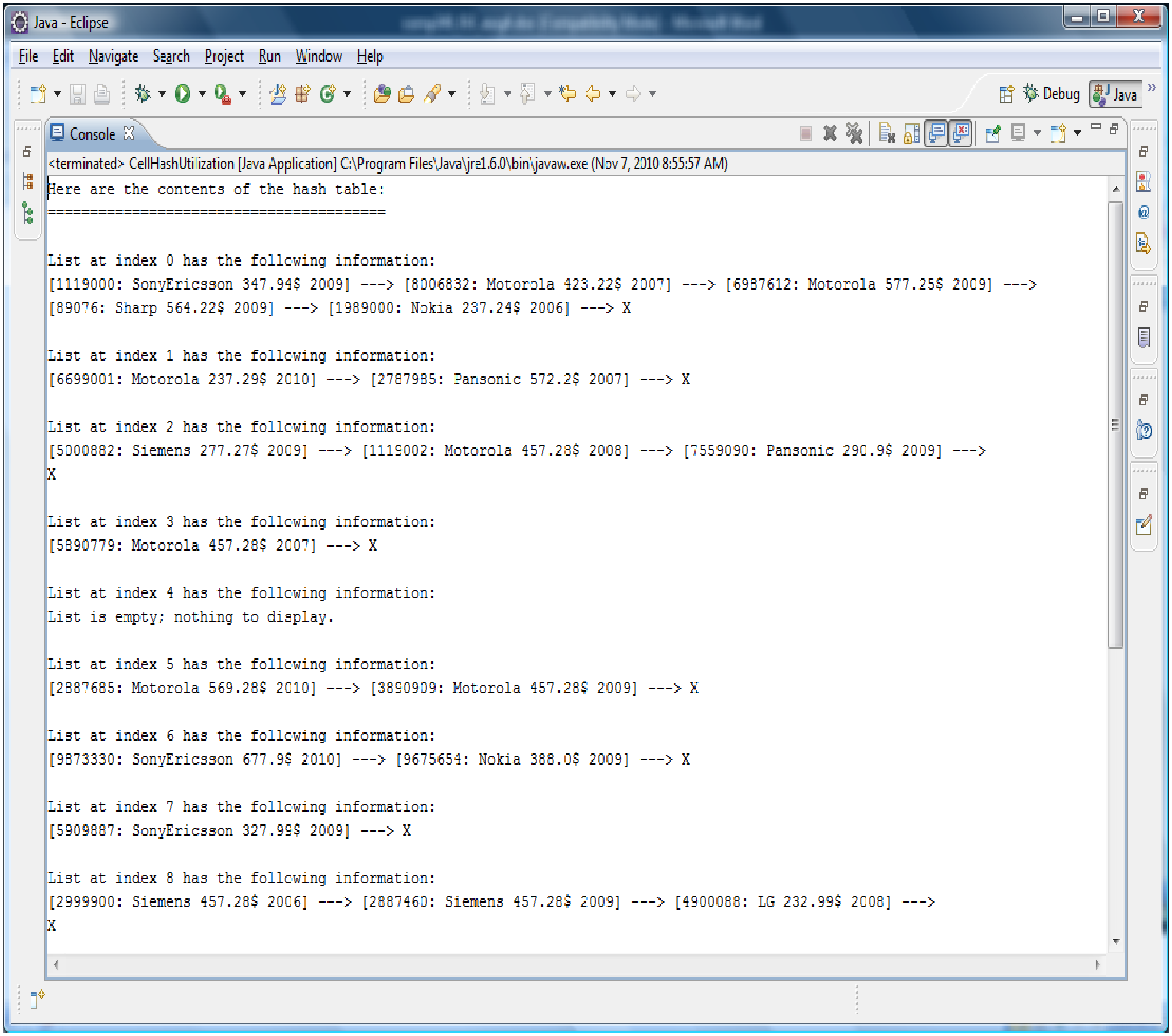
f. A method called ***findCell()***, which accepts a cell phone serial number and finds out if this phone (that is the node that has this phone) is in the *hashArr* table. The method must keep track and display the number of searches conducted before the cell phone is found, or search concludes that no such cell phone is in the table. Effectively, this value indicates the complexity of the algorithm; this is the *O()*.

B) A public class called **CellHashUtilization**. In the main() method of this class, you are required to do the following:

a. Create one **CellHash** object, and read the contents of the ***Cell_Info.txt*** file (using ***addToHashTable()***) into the *hashArr* of that object;

b. Prompt the user to enter few serial numbers to search for, use the ***findCell()*** method to find whether these phone are in the table. Make sure to display this number of conducted searches.

➔ While you do not need to write any further code to compare the results to the one you had in **Part 1**, You should compare these results to the results of the searches you conducted in **Part 1**.



```
<terminated> CellHashUtilization [Java Application] C:\Program Files\Java\jre1.6.0\bin\javaw.exe (Nov 7, 2010 8:55:57 AM)
Here are the contents of the hash table:
========================================

List at index 0 has the following information:
[1119000: SonyEricsson 347.94$ 2009] ---> [8006832: Motorola 423.22$ 2007] ---> [6987612: Motorola 577.25$ 2009] --->
[89076: Sharp 564.22$ 2009] ---> [1989000: Nokia 237.24$ 2006] ---> X

List at index 1 has the following information:
[6699001: Motorola 237.29$ 2010] ---> [2787985: Pansonic 572.2$ 2007] ---> X

List at index 2 has the following information:
[5000882: Siemens 277.27$ 2009] ---> [1119002: Motorola 457.28$ 2008] ---> [7559090: Pansonic 290.9$ 2009] --->
X

List at index 3 has the following information:
[5890779: Motorola 457.28$ 2007] ---> X

List at index 4 has the following information:
List is empty; nothing to display.

List at index 5 has the following information:
[2887685: Motorola 569.28$ 2010] ---> [3890909: Motorola 457.28$ 2009] ---> X

List at index 6 has the following information:
[9873330: SonyEricsson 677.9$ 2010] ---> [9675654: Nokia 388.0$ 2009] ---> X

List at index 7 has the following information:
[5909887: SonyEricsson 327.99$ 2009] ---> X

List at index 8 has the following information:
[2999900: Siemens 457.28$ 2006] ---> [2887460: Siemens 457.28$ 2009] ---> [4900088: LG 232.99$ 2008] --->
X
```

Figure 2: Sample of Displaying the Contents of the Hash Table

Submitting Assignment #3 (5 points)

- Zip together the source codes. (Please use ARCHIVE tool).
- Naming convention for zip file:

  The zip file should be called *a#_studentID*, where # is the number of the assignment *studentID* is your student ID number. For example, for the first assignment, student 123456 would submit a zip file named *a1_123456.zip*
- Submit your zip file at: https://fis.encs.concordia.ca/eas/ as **Programming Assignment** and submission **#3**. Assignments submitted to the wrong directory would be discarded and no replacement submission will be allowed.
- Submit only **ONE version** of an assignment. If more than one version is submitted the first one will be graded and all others will be disregarded.

Evaluation Criteria or Assignment #3 (5 points)

Part 1 (3 points)	
Design and Correctness of Classes	2 pt
Proper and Sufficient Testing of Your Methods	0.5 pt
Privacy Leak Comments and Proposals to Avoid It	0.5 pt
Part 2 (2 points)	
Design and Correctness of Classes	1.5 pts
Proper and Sufficient Testing of Your Methods	0.5 pt