Lab 4

CSE 274

A company has just started operating. As time goes by, the company might start doing business in different states of the country, or might stop doing business in some states. We use a Hash Table data structure to keep a record of which states of the country the company is doing business in. In this Lab, the keyword "states" refers to the states of the country.

Import the `Application.java` in Eclipse, run the Application and make sure that you receive no error.
Erase the body of the main method, copy the following lines of code into the body of the main method, run the application and compare the output with the expected output.

```
HashTable myhashtable = new HashTable(52);
myhashtable.insert("AK");
myhashtable.insert("FL");
myhashtable.insert("IA");
myhashtable.insert("MN");
myhashtable.insert("NV");
myhashtable.insert("VT");
myhashtable.displayTable();
```

The expected output is printed below:

```
** ** ** ** ** ** ** ** IA ** AK ** ** ** ** ** FL ** ** ** ** **
   ** ** ** MN ** ** ** ** ** ** ** ** NV ** ** ** ** ** VT ** **
   ** ** ** ** ** ** ** ** **
```

From above, some of the states are inserted successfully in the `InternalArray`.

I. A Buggy hashFunc

Erase the body of the main method, copy the following lines of code into the body of the main method, run the application and see the output:

```
HashTable myhashtable = new HashTable(52);
myhashtable.insert("AK");
myhashtable.insert("FL");
myhashtable.insert("IA");
myhashtable.insert("MN");
myhashtable.insert("NV");
myhashtable.insert("VT");
myhashtable.insert("ND");
myhashtable.displayTable();
```

The expected output should include the following states and only the following states: `VT IA AK FL MN NV ND`. However, as it can be seen from the output of the above code `FL` is missing. It seems that the Hash Function `hashFunc` is Buggy.

Let's review the expected performance of an ideal hash function. The ideal hash function produces a unique `hashIndex` for each state. Look into the body of the `hashFunc` method in the `Application.java` file. Is the developed `hashFunc` one of such ideal ones? To answer this question examine the output of the `hashFunc` for all the 52 states. Erase the body of the main method, copy the following lines of code into the body of the main method, run the application and see the output:

```
HashTable myhashtable = new HashTable(52);
System.out.print("AL: "+myhashtable.hashFunc("AL")+" ");
System.out.print("AK: "+myhashtable.hashFunc("AK")+" ");
System.out.print("AZ: "+myhashtable.hashFunc("AZ")+" ");
System.out.print("AR: "+myhashtable.hashFunc("AR")+" ");
System.out.print("CA: "+myhashtable.hashFunc("CA")+" ");
System.out.print("CO: "+myhashtable.hashFunc("CO")+" ");
System.out.print("CT: "+myhashtable.hashFunc("CT")+" ");
System.out.print("DE: "+myhashtable.hashFunc("DE")+" ");
System.out.print("DC: "+myhashtable.hashFunc("DC")+" ");
System.out.print("FL: "+myhashtable.hashFunc("FL")+" ");
System.out.print("GA: "+myhashtable.hashFunc("GA")+" ");
System.out.print("HI: "+myhashtable.hashFunc("HI")+" ");
System.out.print("ID: "+myhashtable.hashFunc("ID")+" ");
System.out.print("IL: "+myhashtable.hashFunc("IL")+" ");
System.out.print("IN: "+myhashtable.hashFunc("IN")+" ");
System.out.print("IA: "+myhashtable.hashFunc("IA")+" ");
System.out.print("KS: "+myhashtable.hashFunc("KS")+" ");
System.out.print("KY: "+myhashtable.hashFunc("KY")+" ");
System.out.print("LA: "+myhashtable.hashFunc("LA")+" ");
System.out.print("ME: "+myhashtable.hashFunc("ME")+" ");
System.out.print("MD: "+myhashtable.hashFunc("MD")+" ");
System.out.print("MA: "+myhashtable.hashFunc("MA")+" ");
System.out.print("MI: "+myhashtable.hashFunc("MI")+" ");
System.out.print("MN: "+myhashtable.hashFunc("MN")+" ");
System.out.print("MS: "+myhashtable.hashFunc("MS")+" ");
System.out.print("MO: "+myhashtable.hashFunc("MO")+" ");
System.out.println("");
System.out.print("MT: "+myhashtable.hashFunc("MT")+" ");
System.out.print("NE: "+myhashtable.hashFunc("NE")+" ");
System.out.print("NV: "+myhashtable.hashFunc("NV")+" ");
System.out.print("NH: "+myhashtable.hashFunc("NH")+" ");
System.out.print("NJ: "+myhashtable.hashFunc("NJ")+" ");
System.out.print("NM: "+myhashtable.hashFunc("NM")+" ");
System.out.print("NY: "+myhashtable.hashFunc("NY")+" ");
System.out.print("NC: "+myhashtable.hashFunc("NC")+" ");
System.out.print("ND: "+myhashtable.hashFunc("ND")+" ");
System.out.print("OH: "+myhashtable.hashFunc("OH")+" ");
System.out.print("OK: "+myhashtable.hashFunc("OK")+" ");
System.out.print("OR: "+myhashtable.hashFunc("OR")+" ");
System.out.print("PA: "+myhashtable.hashFunc("PA")+" ");
System.out.print("PR: "+myhashtable.hashFunc("PR")+" ");
System.out.print("RI: "+myhashtable.hashFunc("RI")+" ");
System.out.print("SC: "+myhashtable.hashFunc("SC")+" ");
System.out.print("SD: "+myhashtable.hashFunc("SD")+" ");
System.out.print("TN: "+myhashtable.hashFunc("TN")+" ");
System.out.print("TX: "+myhashtable.hashFunc("TX")+" ");
System.out.print("UT: "+myhashtable.hashFunc("UT")+" ");
System.out.print("VT: "+myhashtable.hashFunc("VT")+" ");
System.out.print("VA: "+myhashtable.hashFunc("VA")+" ");
System.out.print("WA: "+myhashtable.hashFunc("WA")+" ");
System.out.print("WV: "+myhashtable.hashFunc("WV")+" ");
System.out.print("WI: "+myhashtable.hashFunc("WI")+" ");
```

In the output, you should see the following `hashIndex`-es:

```
AL: 11 AK: 10 AZ: 25 AR: 17 CA: 2 CO: 16 CT: 21 DE: 7 DC: 5 FL: 16
GA: 6 HI: 15 ID: 11 IL: 19 IN: 21 IA: 8 KS: 28 KY: 34 LA: 11 ME: 16
MD: 15 MA: 12 MI: 20 MN: 25 MS: 30 MO: 26 MT: 31 NE: 17 NV: 34
NH: 20 NJ: 22 NM: 25 NY: 37 NC: 15 ND: 16 OH: 21 OK: 24 OR: 31
PA: 15 PR: 32 RI: 25 SC: 20 SD: 21 TN: 32 TX: 42 UT: 39 VT: 40
VA: 21 WA: 22 WV: 43 WI: 30 WY: 46
```

Are the above 52 `hashIndex`-es unique? Please notice that:
The `hashFunc` gives a `hashIndex` of 21 for the states of CT, OH, SD, VA and IN.
The `hashFunc` gives a `hashIndex` of 25 for the states of AZ, MN, NM and RI.
The `hashFunc` gives a `hashIndex` of 16 for the states of CO, FL, ME and ND.
The `hashFunc` gives a `hashIndex` of 20 for the states of NH, SC and MI.
The `hashFunc` gives a `hashIndex` of 11 for the states of AL, ID and LA.
The `hashFunc` gives a `hashIndex` of 15 for the states of HI and PA.
The `hashFunc` gives a `hashIndex` of 34 for the states of KY and NV.
The `hashFunc` gives a `hashIndex` of 30 for the states of MS and WI.
The `hashFunc` gives a `hashIndex` of 31 for the states of MT and OR.
The `hashFunc` gives a `hashIndex` of 32 for the states of PR and TN.
The `hashFunc` gives a `hashIndex` of 17 for the states of AR and NE.
The `hashFunc` gives a `hashIndex` of 22 for the states of NJ and WA.

Accordingly, the developed `hashFunc` does not provide unique `hashIndex`-es. Rather the developed `hashFunc` produces `hashIndex`-es that sometime collide.

II. ProbeANDinsert Method

Developing an ideal `hashFunc` is a hard task. Therefore, in this section we use a `ProbeANDinsert` method that can successfully insert all the states in the `InternalArray` using the non-ideal `hashFunc` that is available in `HashTable` class. Add the following method to the `HashTable` class:

```
public void ProbeANDinsert(String key)
{
      int hashIndex = hashFunc(key);

      while (!InternalArray[hashIndex].equals("**"))
      {
            if (InternalArray[hashIndex].equals(key))
                  return;
            ++hashIndex;
            hashIndex = hashIndex % arraySize;
      }
      InternalArray[hashIndex] = key;
}
```

Erase the body of the main method, copy the following lines of code into the body of the main method, run the application and compare the output with the expected output.

```
HashTable myhashtable = new HashTable(52);
myhashtable.ProbeANDinsert("AK");
myhashtable.ProbeANDinsert("FL");
myhashtable.ProbeANDinsert("IA");
myhashtable.ProbeANDinsert("MN");
myhashtable.ProbeANDinsert("NV");
myhashtable.ProbeANDinsert("VT");
myhashtable.ProbeANDinsert("ND");
myhashtable.displayTable();
```

The expected output is printed below:

```
** ** ** ** ** ** ** ** IA ** AK ** ** ** ** ** FL ND ** ** ** **
   ** ** ** MN ** ** ** ** ** ** ** ** NV ** ** ** ** ** VT ** **
   ** ** ** ** ** ** ** ** **
```

Notice that hashFunc gives a hashIndex of 16 for both FL and ND. Still, the ProbeANDinsert method successfully inserts both the FL and ND in the InternalArray. Also, notice that ND is placed in the InternalArray right after FL.

From Section I, the hashFunc gives a hashIndex of 16 for FL, ND, CO and ME. Let's call the ProbeANDinsert method for all these states to see where they are placed in the InternalArray. Erase the body of the main method, copy the following lines of code into the body of the main method, run the application and compare the output with the expected output.

```
HashTable myhashtable = new HashTable(52);
myhashtable.ProbeANDinsert("AK");
myhashtable.ProbeANDinsert("FL");
myhashtable.ProbeANDinsert("IA");
myhashtable.ProbeANDinsert("MN");
myhashtable.ProbeANDinsert("NV");
myhashtable.ProbeANDinsert("VT");
myhashtable.ProbeANDinsert("ND");
myhashtable.ProbeANDinsert("CO");
myhashtable.ProbeANDinsert("ME");
myhashtable.displayTable();
```

The expected output is printed below:

```
** ** ** ** ** ** ** ** IA ** AK ** ** ** ** ** FL ND CO ME ** **
   ** ** ** MN ** ** ** ** ** ** ** ** NV ** ** ** ** ** VT ** **
   ** ** ** ** ** ** ** ** **
```

Notice that ProbeANDinsert method places FL, ND, CO and ME in a block of data in the InternalArray.

Section I discussed which states have a same hashInteger given by hashFunc. Review that discussion and use the ProbeANDinsert method to create a block of four data items right after MN, including MN itself.

III. Developing ProbeANDdelete

Since hashFunc is not ideal we cannot use delete method of the HashTable class for deleting data from the InternalArray. Develop the following method for the HashTable class:

```
public void ProbeANDdelete(String key)
```

The above method takes a key from the user and deletes the key from the InternalArray using the non-ideal hashFunc in Application.java file. Use the following logic/pseudocode for developing ProbeAnddelete method:

```
int hashIndex = hashFunc(key)
while (InternalArray[hashIndex] is not empty)

      if (key is in InternalArray[hashIndex])

            delete InternalArray[hashIndex]
            return;

      increase hashIndex by one.
      hashIndex = hashIndex % arraySize
return
```

To test the developed ProbeANDdelete method erase the body of the main method, copy the following lines of code into the body of the main method, run the application and compare the output with the expected output.

```
HashTable myhashtable = new HashTable(52);
myhashtable.ProbeANDinsert("AK");
myhashtable.ProbeANDinsert("FL");
myhashtable.ProbeANDinsert("IA");
myhashtable.ProbeANDinsert("MN");
myhashtable.ProbeANDinsert("NV");
myhashtable.ProbeANDinsert("VT");
myhashtable.ProbeANDinsert("ND");
myhashtable.ProbeANDinsert("CO");
myhashtable.ProbeANDinsert("ME");
myhashtable.ProbeANDinsert("AZ");

myhashtable.ProbeANDdelete("AK");
myhashtable.ProbeANDdelete("CO");
myhashtable.ProbeANDdelete("AZ");

myhashtable.displayTable();
```

The expected output is printed below:

```
** ** ** ** ** ** ** ** IA ** ** ** ** ** ** ** FL ND ** ME ** **
   ** ** ** MN ** ** ** ** ** ** ** ** NV ** ** ** ** ** VT ** **
   ** ** ** ** ** ** ** ** **
```

IV. A BUG IN PROBEANDDELETE METHOD

There is a bug in ProbeANDdelete method. To see this bug, erase the body of the main method, copy the following lines of code into the body of the main method, run the application and compare the output with the expected output.

```
HashTable myhashtable = new HashTable(52);
myhashtable.ProbeANDinsert("AK");
myhashtable.ProbeANDinsert("FL");
myhashtable.ProbeANDinsert("IA");
myhashtable.ProbeANDinsert("MN");
myhashtable.ProbeANDinsert("NV");
myhashtable.ProbeANDinsert("VT");
myhashtable.ProbeANDinsert("ND");
myhashtable.ProbeANDinsert("CO");
myhashtable.ProbeANDinsert("ME");
myhashtable.ProbeANDinsert("AZ");
myhashtable.ProbeANDdelete("CO");
myhashtable.ProbeANDdelete("ME");
myhashtable.displayTable();
```

The expected output is printed below:

```
** ** ** ** ** ** ** ** IA ** ** ** ** ** ** ** FL ND ** ** ** **
   ** ** ** MN ** ** ** ** ** ** ** ** NV ** ** ** ** ** VT ** **
   ** ** ** ** ** ** ** ** **
```

As you have noticed, the `ProbeANDdelete` method is unable to delete ME. That is because, when `ProbeANDdelete` is called to delete CO, `ProbeANDdelete` creates a gap between ME and the block of data that ME belonged to. When such a gap exists, if `ProbeANDdelete` method is called to delete ME, `ProbeANDdelete` method terminates the act of probing once `ProbeANDdelete` reaches the gap, and never checks the `InternalArray` at the specific index where ME is residing.

To fix this bug, develop the following method:

```
public void ProbeAnddeleteButNoGap(String key);
```

The above method deletes a key without creating a gap in any block of data. In fact, the above method shifts the data items in blocks of data when needed to prevent creation of gaps. Use the following logic/pseudocode for developing the above method:

```
int hashIndex = hashFunc(key)
while (InternalArray[hashIndex] is not empty)

   if (key is in InternalArray[hashIndex])

        while (InternalArray[hashIndex] is not empty)

           int NextIndex = (hashIndex+1)%arraySize
           InternalArray[hashIndex] = InternalArray[NextIndex]
           hashIndex = hashIndex+1
           hashIndex = hashIndex % arraySize

        return;

   Increase hashIndex by one.

   hashIndex = hashIndex % arraySize

return
```

To test the developed ProbeANDdeleteNoGap method, erase the body of the main method, copy the following lines of code into the body of the main method, run the application and compare the output with the expected output.

```
HashTable myhashtable = new HashTable(52);
myhashtable.ProbeANDinsert("AK");
myhashtable.ProbeANDinsert("FL");
myhashtable.ProbeANDinsert("IA");
myhashtable.ProbeANDinsert("MN");
myhashtable.ProbeANDinsert("NV");
myhashtable.ProbeANDinsert("VT");
myhashtable.ProbeANDinsert("ND");
myhashtable.ProbeANDinsert("CO");
myhashtable.ProbeANDinsert("ME");
myhashtable.ProbeANDinsert("AZ");

myhashtable.ProbeANDdeleteButNoGap("CO");
myhashtable.ProbeANDdeleteButNoGap("ME");

myhashtable.displayTable();
```

The expected output is printed below:

```
** ** ** ** ** ** ** ** IA ** AK ** ** ** ** ** FL ND ** ** ** **
   ** ** ** MN AZ ** ** ** ** ** ** ** NV ** ** ** ** ** VT ** **
   ** ** ** ** ** ** ** ** **
```

V. Iterating Forever

The ProbeANDdeleteNoGap still has a bug. If the InternalArray is full and this method is called, a while loop inside the method will iterate forever. This bug can be fixed by preventing the InternalArray from becoming full. Add a new private data type of int to the HashTable class and name it num. Develop the following methods:

```
public void ProbeANDinsertNotForever(String key)
public void ProbeANDdeleteNoGapNotForever(String key)
```

The ProbeANDinsertNotForever method increases num by one when inserting a new key into the InternalArray. The ProbeANDdeleteNoGapNotForever method decreases num by one when deleting a key from the InternalArray. The ProbeANDinsertNotForever does not insert the key into the InternalArray if the size of InternalArray is arraySize-1.

To test the developed methods, erase the body of the main method, copy the content of test.txt file into the body of the main method, run the application and compare the output with the expected output. The expected output is printed below:

```
WA CA WV WI DC GA DE IA ** ** AK AL ID LA MA HI CO AR FL IL ME CT
   IN MD MI AZ MN MO KS NE MS MT NH NJ KY NV NM NY NC ND OH OK OR
   PA PR RI SC SD TN TX UT VA
```

VI. ProbeANDfind

Develop the following method for the HashTable class:

```
public boolean ProbeANDfind(String key)
```

The above method searches for a `key` in the `InternalArray` and `return true` only when the `key` is found in the `InternalArray`. When `Key` exists in the `InternalArray` the number of steps that the method executes should not be larger than the number of data elements in the data block that `key` belongs to.
To test the developed methods, erase the body of the main method, copy the following lines of code into the body of the main method, run the application and compare the output with the expected output.

```
HashTable myhashtable = new HashTable(52);
myhashtable.ProbeANDinsertNotForever("AL");
myhashtable.ProbeANDinsertNotForever("AK");
myhashtable.ProbeANDinsertNotForever("AZ");
myhashtable.ProbeANDinsertNotForever("AR");
myhashtable.ProbeANDinsertNotForever("FL");
myhashtable.ProbeANDinsertNotForever("CO");
myhashtable.ProbeANDinsertNotForever("CT");
myhashtable.ProbeANDinsertNotForever("DE");
myhashtable.ProbeANDinsertNotForever("ME");
System.out.println(myhashtable.ProbeANDfind("ME"));
myhashtable.ProbeANDdeleteButNoGapNotForever("ME");
System.out.println(myhashtable.ProbeANDfind("ME"));
System.out.println(myhashtable.ProbeANDfind("FL"));
System.out.println(myhashtable.ProbeANDfind("CA"));
```

The expected output is printed below:

```
true
false
true
false
```

VII. Submitting the Assignment

The mechanism that we used in this lab to implement a hash table data structure using a non-ideal hash function is called linear probing.

Before submitting the assignment, erase the body of the main method and copy the lines of code in Section VI into the body of the main method.