Lab 9

The goal of the lab is to develop your own strategy for filling up a hash table and minimize the number of collisions that result.

You can use any hashing strategy you like. The main method that reads in the data is provided (see Lab9 code below).

In the code provided you will see a HashTable class – your code will use the check() method to check for a collision when you are finding a word. Every time you make an unsuccessful check() call (i.e. a collision), a counter ticks up. You have to keep this as low as possible.

You should just **complete the fill() and find() methods**. When you run the code, it will output the number of collisions you have. If you see the word "error!" it means that your find() method is not finding the words properly in the hash table.

The input is already taken care of. All you need to do is complete the fill() method, which fills the words into the hash table array, and the find() method, which should return the slot in the hash table where the word is to be found. You have to come up with some system for mapping words to slot number, based on the word itself (for example, adding all the letters together to create a number, and then moduloing by the size of the hash table etc.)

There are 90,000 items being inserted into a hash table of size 99,991, so the load factor is 90%, which is very high. If you can get the number of collisions below 200,000 that is very good. Below 150,000 is really excellent.

```
import java.util.*;
import java.io.*;

public class Lab9{

    public static void main (String[] args){
        File file = new File("C:\\ HashTable.txt");

        int inputSize = 90000;
        String[] input = new String[inputSize];
        try {
            Scanner scan = new Scanner(file);

            for(int i = 0; i < inputSize; i++) {
                input[i] = scan.nextLine();
            }
            scan.close();
        } catch (Exception e) {
            System.err.println(e);
        }
        int size=99991;
        Solution mysolution = new Solution();
        String[] hashtable=mysolution.fill(size, input);
        HashTable mytable = new HashTable(hashtable);

        Solution mysolution2 = new Solution(); //prevents cheating
by using memory
        for(int i=0;i<inputSize;i++){
            int rand = (int)(Math.random()*inputSize);
            String temp = input[i];
            input[i]=input[rand];
            input[rand]=temp;
        }
```

```
        int total=0;
        for(int i=0;i<inputSize;i++){
            int slot = mysolution2.find(size, mytable, input[i]);
            if(!hashtable[slot].equals(input[i])){
                System.out.println("error!");
            }
        }
        long status=mytable.gettotal();
        System.out.println("Collisions: " + status);
    }

}


class HashTable{

    private String[] hashTable;
    private long total=0;

    public HashTable(String[] input){
        hashTable = input;
    }

    public boolean check(int slot, String check){
        if(hashTable[slot].equals(check)){
            return true;
        }else{
            total++;
            return false;
        }
    }
```

```
    public long gettotal(){
        return total;
    }
}
```

```
class Solution{

    public int find(int size, HashTable mytable, String word){

        //fill this in so as to minimize collisions
        //takes in the HashTable object and the word to be found
        //the only thing you can do with the HashTable object is call "check"
        //this method should return the slot in the hashtable where the word is

        return 0;
    }

    public String[] fill(int size, String[] array){

        //takes in the size of the hashtable, and the array of Strings to be placed in the hashtable
        //this should add all the words into the hashtable using some system
        //then it should return the hashtable array
        String[] hashtable = new String[size];
        for(int i=0;i<size;i++){
            hashtable[i]="";
        }
```

```
        return hashtable;

    }
}
```