```java
package DoubleHashTable;

import List.LNode;

/**
 * DoubleHashTable class is an implementation of a double hashing table that implements insert, find,
 * delete, and print. If an item is added to the table more than once; a counter, c, of HNode is
 * incremented by one. Implemented operations are:
 *
 * insert       Insert a String into the table.
 * find         Locate a String in the table and return it's index.
 * delete       Find a String in the table and decrement it's counter by one. If counter is 0, flag as
 *              deleted.
 *
 * When the table is printed, the indexes from 0 to 112 will be printed along with the word's count,
 * and the word. At the end of the table, the percent full the table is will be printed.
 *
 * ****************** Hash functions *****************
 *
 * Primary Hash function for initial index:     key % TABLE_SIZE
 * Secondary Hash function for probe sequence:  (key+43)/43
 *
 * ******************* Valid Input *******************
 *
 * words        An LNode linear linked-list of Strings.
 *
 * *************** Public Operations *****************
 *
 * insert       Insert a given String into the hash table.
 * find         Locate a given String in the table and return it's index.
 * delete       Find and remove a given string from the table.
 * print        Print a visual representation of the table to the console.
 *
 * ***************** Global Variables *****************
 *
 * TABLE_SIZE   Private variable for size of table (113)
 * table        Private variable for table HNode array.
 *
 * @author Matt Laidman (5199807)
 * @version 1.0 (November 24, 2014)
 */
public class DoubleHashTable {

    private static int TABLE_SIZE = 113;                        // Private int for table size
    private HNode[] table = new HNode[TABLE_SIZE];              // Private HNode array for table

    /**
     * Public constructor to call private hashIt function with words list.
     *
     * @param words     The words to add to the table.
     */

    public DoubleHashTable(LNode words) {

        hashIt(words);                                          // Call private hashIt with words
    }

    /**
     * Public insert function to call private insert function with String and integer representation of
     * String from getInt function.
     *
     * @param word      The words to insert into the table.
     */

    public void insert (String word) {

        insert(word, getInt(word));                             // Call private insert with word/int
    }

    /**
     * Public find function to call private find function with String and integer representation of
```

```
 * String from getInt function.
 *
 * @param word      The word to find in the table.
 * @return          The index of the word in the array.
 */

public int find (String word) {

    return find(word, getInt(word));                        // Return index of the word in the array
}

/**
 * Public delete function to call private delete function with the index of the word from the getInt
 * function.
 *
 * @param word      The word to delete from the function.
 */

public void delete (String word) {

    delete(find(word));                                     // Find the word's index and delete it
}

/**
 * Private delete function decrements the counter of the HNode at the given index. If the counter
 * reaches 0, the deleted boolean flag, del, is set.
 *
 * @param index     The index of the word to delete.
 */

private void delete (int index) {
    if (index != -1 && table[index] != null) {
        table[index].c--;                                   // Decrement counter
        if (table[index].c == 0) {                          // If counter reaches 0
            table[index].del = true;                        // Flag as deleted
        }
    }
}

/**
 * Private find function takes a given String and integer representation pair, and performs the
 * hash and probe sequence until it either finds a null HNode, or the word in the table. If the word
 * is found, the index in the array is returned. If the word is found the index of the word in the
 * array is returned, otherwise -1 is returned.
 *
 * @param word      The String to find in the table
 * @param key       The integer representation of word from getInt.
 * @return          The index of the word if found, -1 otherwise.
 */

private int find (String word, int key) {
    int hash = primaryHash(key);                            // Get initial hash
    while (table[hash] != null && !isDuplicateNode(table[hash], word)) {
        hash = (hash + secondaryHash(hash)) % TABLE_SIZE;   // Probe if necessary until found or null
    }
    if (table[hash] != null) {                              // If found return index
        return hash;
    } else {                                                // Otherwise return -1
        return -1;
    }
}

/**
 * Private insert function gets an initial hash value from the integer and then if necessary probes
 * the table with a value from a second hash function until an empty index in the array is found or
 * the word itself is found. If the word already exists in the table, it's counter, c, is
 * incremented, otherwise the HNode is created with the String.
 *
 * @param word      The word to add to the table.
 * @param key       The Integer representation of the word.
 */
```

```
private void insert (String word, int key) {

    int hash = primaryHash(key);                              // Get initial hash
    if (table[hash] == null) {                                // If empty index
        table[hash] = new HNode(word);                        // Create new HNode with word
    } else {                                                  // Otherwise
        if (table[hash].key.equals(word)) {                   // If word already in index
            table[hash].c++;                                  // Increment counter
            if (table[hash].del) {                            // If word was flagged as deleted,
                table[hash].del = false;                      // Set as del to false
            }
        } else {                                              // Otherwise probe until found or empty
            hash = (hash + secondaryHash(hash)) % TABLE_SIZE; // index
            while (table[hash] != null && !isDuplicateNode(table[hash], word)) {
                hash = (hash + secondaryHash(hash)) % TABLE_SIZE;
            }
            if (table[hash] != null) {                        // If found, increment counter
                table[hash].c++;
                if (table[hash].del) {                        // If word was flagged as deleted,
                    table[hash].del = false;                  // Set as del to false
                }
            } else {                                          // Otherwise create HNode with word
                table[hash] = new HNode(word);
            }
        }
    }
}

/**
 * Private isDuplicateNode function returns true if the given HNode has a key equal to the given
 * String.
 *
 * @param node      The HNode to compare.
 * @param word      The String to compare.
 * @return          True if HNode is duplicate, false otherwise.
 */

private boolean isDuplicateNode (HNode node, String word) {

    return node != null && node.key.compareTo(word) == 0;
}

/**
 * Private secondaryHash function returns the probe sequence for a given key.
 *
 * Hash Function: (key+43)/43
 *
 * @param key       The key to get probe sequence for.
 * @return          The probe sequence for the key.
 */

private int secondaryHash (int key) {

    return (key+43)/43;
}

/**
 * Private primaryHash function returns the initial hash index for a given integer representation of
 * a String.
 *
 * @param key       The integer representation of the String.
 * @return          The index value for the key.
 */

private int primaryHash (int key) {

    return key % TABLE_SIZE;
}

/**
```

```
     * Private getInt function returns an integer mostly unique to a given String. The function
     * implemented to do this is:
     *
     * (Sum((Index+1)*ASCIICharValue))*PI
     *
     * @param word      The word to get integer for.
     * @return          The integer representation of the word.
     */

    private int getInt (String word) {

        double key=0;
        char[] chars = word.toCharArray();                      // Get char array.
        for (int i = 0 ; i < chars.length ; i++) {              // For each character in array
            key += (i+1)*chars[i];                              // Add to sum (index+1)*ASCIICharValue
        }
        return (int)(key*Math.PI);                              // Return sum*PI
    }

    /**
     * Private hashIt function adds each word in a given list to the table by calling the insert
     * function.
     *
     * @param words     The list of words to add to the table
     */

    private void hashIt (LNode words) {

        while (words != null) {                                 // While there are words in list
            insert(words.key);                                  // Insert word
            words = words.next;                                 // Get next word
        }
    }

    /**
     * Public print function prints a visual representation of the Hash table to the console.
     * The index, followed by the count of the word in the table, followed by the word itself will be
     * printed. Following the table, the percentage full the table is will be printed.
     */

    public void print ( ) {
        int c = 0;
        double p;
        System.out.println("\n Index  | Count | Word");
        System.out.println("------------------------");
        for (int i = 0 ; i < TABLE_SIZE ; i++) {                // For each index value
            System.out.print (i + "\t\t| ");                     // Print index
            if (table[i] != null && !table[i].del) {            // If index is not null or deleted
                c++;
                System.out.print(table[i].c);                   // Print count
                if (table[i].c >= 10) {
                    System.out.print("\t| ");
                }else {
                    System.out.print("\t\t| ");
                }
                System.out.println(table[i].key);               // Print word
            } else {
                System.out.println("\t\t|");
            }
        }
        p = ((double)c)/TABLE_SIZE;
        System.out.print("\nTable is ");                        // Print percentage full
        System.out.format("%.3f", p);
        System.out.println("% full.");
    }
}
```