CS 2420 Program 4 – 20 points  
Spring 2019

Déjà vu – Hash Table

**Part 1: Becoming familiar with the code**

HashTable code has been given to you. No testing program has been provided. To become familiar with how the code works, try reading in a small input file and make sure you can create a hash table of those entries.

**Testing:** Make sure the following works:

a. Insert values

b. Delete values

c. Find values

d. Printing the contents of the hash table.

e. Control the size of the hash table.

What happens if you attempt to delete an item that isn’t there?

What if you add more things than can fit into the hash table?

**Modification:** You have been given the hash table code from your text for doing quadratic probing. Modify the code by using double hashing instead of quadratic probing.

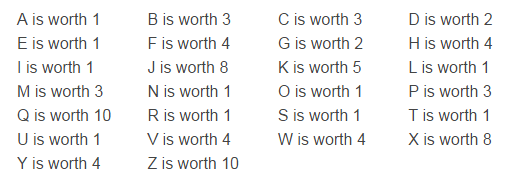
Please retain the generic structure of the hash table. Be careful not to add anything to the hash table that only makes sense for this specific problem.

Part 1 is for your benefit. The code you turn in does not need to show the results of this experimentation.

**Part 2 Using the code to solve a bigger problem**

The code does need to use quadratic probing.

**Your favorite word game is getting a little old. You decide to create “house rules” to reward creativity. This is what you decide on. You will score each word you generate based on (1) length of word (2) value of each letter in the word (3) bonus (associated with infrequently used words).**



|  |  |
| --- | --- |
| **Length** | **lengthValue** |
| **1-2** | **0** |
| **3** | **1** |
| **4** | **2** |
| **5** | **3** |
| **6** | **4** |
| **7** | **5** |
| **8 or more** | **6** |

|  |  |
| --- | --- |
| **Times Word is Used Before in game** | **bonus** |
| **0** | **5** |
| **1-5** | **4** |
| **6-10** | **3** |
| **11-15** | **2** |
| **more than 15** | **1** |

**The formula to compute the score for a word is:**

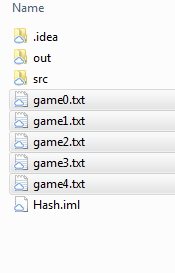
**Word score = (sum of letter values) \* (lengthValue)\*bonus**

**So suppose you use the word “ozone” for the first time. It would have the score of (1+10+1+1+1)\*3\*5 = 210**

**If you used “via” for the 12th time, it would have the score of (4+1+1)\*1\*2 = 12 points.**

**Input**

|  |
| --- |
| **Files** |
| game0.txt |
| game1.txt |
| game2.txt |
| game3.txt |
| game4.txt |

Each input file consists of a list of words which you are to score. The score for the game is the sum of all the word scores. If you are using Intellij, the files need to be located in the same directory as src (not within src).

**Output**

For each input file, print the following:

1. The score for the game (which is the sum of the word scores).
2. Hash Table Statistics:
3. The total number finds done on the hash table
4. The number of probes required in those finds. (This will help us determine how good the hash function is.) You can add a count inside the findPos method.
5. Number of items stored in hash table.
6. The physical length of the hash table. This code changes the size of the hash table dynamically.
7. Contents of the first 20 non-null entries in the hash table. This includes the word and the occurrence count.

**Hints:**

Using a hashtable in a generic fashion is a bit tricky.

Use a hash table to record how many times each word in the input file has been seen before (on previous lines). You will need to create a class which stores both the word and the count (of times it has been used previously). Suppose the class is called WordInfo. Then you will create a hash table of WordInfo.

**H** = **new** DoubleHashTable<WordInfo>();

**To see if a word is already in the table, the code will look something like:**

WordInfo w = **new** WordInfo(word,0);  
WordInfo f = **H**.find(w);

**if** (f==**null**) { // Item is not already there

You can’t just ask if the word is in the hashtable as the hashtable only knows about operations on the item stored in the hash table.

find works by checking if two “WordInfo” items are “equal” in terms of having the same key.

You will need to override “equals” in WordInfo, so you see if the word is there (without expecting the count to be the same). Using @Override is helpful as it makes sure your new function is seen as a replacement for the build-in method of the same name.

You will need to define “hashCode” for WordInfo as well.

My equals function looked like:

@Override  
**public boolean** equals(Object w2){  
 **if** (w2==**this**){  
 **return true**;  
 }  
 **if** (!(w2 **instanceof** WordInfo)){  
 **return false**;  
 }  
 WordInfo w= (WordInfo)w2;  
 **return** (**this**.**word**.compareTo(w.**word**) ==0);  
}

To compute the value of a character (a is 1, b is 3, c is 3...), there are a variety of ways of doing this.  While a nested if or switch works, you should realize that you can easily turn the character into an int (to be used as a subscript into a table of values). The trick is subtract ‘a’ from the character.

**for** (**int** i=0; i < w.length(); i++)  
 val+= letterValue[w.charAt(i)-**'a'**];

Make sure you understand the difference between the hash function and the collision resolution method. The hash function tells you how to pick your “first choice” for where an entry will be stored. The collision resolution method tells you what to do when your first choice isn’t available.

When you use double hashing, you need to make sure the step for two different keys (that originally hashed to the same location) is something different. Look at the class notes for good hash and step functions.