CS 2420 Program 4 – 20 points
Fall 2015

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.

Make sure the following works:

a. Insert values

b. Delete values

c. Find values

d. Printing the contents of the hash table.

e. How can you 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?

Part 1 is for your benefit. Nothing needs to be turned in from this part.

Part 1 exposes a couple of "gotchas" in the code.

a. The code to print the contents relies on a toString. Obviously, this won't work for built-in types. If you are testing the code with strings, you'll see this. It would be better to overload << for any types you want to use as HashRecords with the hash table.

b. The hash table is expecting HashRecord pointers. This has some advantages in that you don't have to copy whole records and can easily return a NULL for "not found". However, you will need to be careful. Consider what happens when you do something like:

```
Pair wordFrequency;
cin >> wordFrequency.word;
wordFrequency.ct = 0;
hashTab.insert(wordFrequency.word, &wordFreqency)
```

Depending on where wordFreqency is defined you could have one of the following problems:

a. Every entry in the hash table shares the same address, so if you change one Pair, you change all Pairs.

b. You gave the hashTable the address of a local variable (which was destroyed upon exit from the method).

Instead, you need to do something like:

```
cin >> word;
Pair *wordFrequency = new Pair(word,0);
hashTab.insert(word, wordFreqency)
```

Part 2 Using the code to solve a bigger problem

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 word).

A is worth 1	B is worth 3	C is worth 3	D is worth 2
E is worth 1	F is worth 4	G is worth 2	H is worth 4
I is worth 1	J is worth 8	K is worth 5	L is worth 1
M is worth 3	N is worth 1	O is worth 1	P is worth 3
Q is worth 10	R is worth 1	S is worth 1	T is worth 1
U is worth 1	V is worth 4	W is worth 4	X is worth 8
Y is worth 4	Z is worth 10		

Length	LengthValue
1-2	0
3	1
4	2
5	3
6	4
7	5
8 or more	6

Times 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:

score(word) = (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.

Use a hash table to record how many times each word in the input file has been seen before.

Specifics

You have been given the hash table code from your text for doing quadratic probing. Modify the code in at least two ways:

a. Use your own hash function, patterned after code we used in class rather than the built-in hash function that is used in the code.
b. Instead of quadratic probing, use double hashing.

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

Input

Each input file will consist of a list of words which you are to score.

Output

For each input file, generate the score for the game. The score would typically be updated after every word is input. For simplicity, print the score after every OutputFrequency words and at the end of the game. (Allow the user to set OutputFrequency.)

For debugging/grading purposes, allow the user to easily request a copy of the first few (50 or so) items of the hash table be output. No user interface is required. The user can just change the main program.

The hash table print should contain the following information:

1. The hash table should keep track of the total number finds done on the hash table AND the number of probes required in those finds. (This will help us determine how good the hash function is.)
2. Number of items stored in hash table.
3. Size of hash table. This code changes the size of the code dynamically.
4. Contents of entries in the hash table. This includes the word and the occurrence count.

Files
game0.txt
game1.txt
game2.txt
game3.txt
game4.txt

Hints:

1. 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 case 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):

```
char c;
int sub = c-'a';
```

This is classic C coding. You should know it works.

2. There is an elegant way of converting length to the length value as well.
3. 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.
4. 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.

   One way to do this is to create a different hash function for the step. For example, if your regular hash function is

```
unsigned int  hash(string key)
{unsigned int hashVal=0;
  for (i=0; i < key.length(); i++)
    hashVal =  (hashVal << 5) ^ key[i]  ^ hashVal;
  return hashVal %TABLESIZE;
}
```

   You could compute step as

```
unsigned int step(string key)
{unsigned int stepVal=0;
  for (i=0; i < key.length(); i++)
    stepVal =  (stepVal << 7) ^ key[i]  ^ stepVal;
  return 1+ stepVal %(TABLESIZE-2);
}
```