CPE 111 – Programming with Data Structures
International Sections - January 2021
Laboratory Exercise 8

Objective

This lab is intended to give you the opportunity to practice using a hash table data structure

Instructions

Summary

Create a program called **hashDictionary.c** that stores multiple words and definitions. After building the dictionary, allow the user to request the definitions of words. Each time the user enters a word, look it up in the hash table to find its definition and print it. Do some experiments to see how the size of the hash table affects the retrieval time.

Details

1. Download the contents of the **Lecture8** directory under **demos**. (The fastest way to do this is to download and unpack **lecture8Demos.zip.**) You should have the following files:

chainHashTable.c
abstractHashTable.h
hashTester.c
Makefile
wordlist.txt

2. Open or type **wordlist.txt** so you can see its contents. Each line has a single word plus its definition, with a "|" separating the two. Your program will read this file and build a dictionary using a hash table. This is similar to Lab 7. In the current lab, instead of creating your own data structure, you will use my chained hash table to store and retrieve the information in the list. Note that there are about 5000 words and definitions in this file.

3. Start with your Lab 8 program, **dictionary.c**. Copy this file to **hashDictionary.c.** Include the usual system header files and also include **abstractHashTable.h** and **timeFunctions.h**.

4. Copy the function ***bitwiseOpHash()*** from **hashTester.c** to **hashDictionary.c**. You will be using this hash function to initialize the hash table where you will store words and definitions. The word will be the key to the hash.

5. Define a constant to hold the number of buckets in your hash table.

```
#define BUCKET_COUNT 503
```

6. In your main function, call the function ***hashTableInit()*** to initialize the hash table. Pass **BUCKET_COUNT** as the first argument (for the size of the hash table) and **&bitwiseOpHash** as the second argument (the hash function to use). Be sure to check the return value to make sure that the initialization succeeds.

7. Modify the original ***buildDictionary()*** function from **dictionary.c**. This function should still read and parse the word list. *Be sure to change your code to expect a vertical bar instead of a colon between word and the definition.* Each time you read a word and definition, create a dictionary entry as you did in the Lab 7 code. However, once you have created and initialized the dictionary entry structure, just call the ***hashTableInsert()*** function to put the entry into the dictionary. Pass the word as the key argument. Pass the pointer to your allocated *DICT_ENTRY_T* as the data argument. For the **pCollision** argument, just declare a dummy integer variable and pass a pointer to it. You will not need to look at this argument.

8. Modify the ***printDefinition*** function to use the ***hashTableLookup()*** function. The function should look up the word in the dictionary and print out the definition it finds for the word (if any). (Note that in this word list, no word has more than one definition.) The ***hashTableLookup()*** function will return either the full dictionary entry structure associated with the word, or NULL if the word is not in the dictionary.

Immediately before you call ***hashTableLookup()***, call the function **recordTime()** (in **timeFunctions.h/c**) passing a 1 for the **bStart** argument. Immediately after the call to ***hashTableLookup()***, call the ***recordTimefunction()*** again with a 0 argument, saving the result in a long integer variable. This will tell you how long it took to look up the word. Print this value to the terminal after the definition, e.g.

```
recordTime(1);
/* code to call function to lookup the word */
long microseconds = recordTime(0);
/* print definition... or message about not finding word */
printf("Lookup required %d microseconds\n\n", microseconds);
```

9. Remove the ***getDictIndex()***, ***printFullDictionary()*** and ***freeDictionary()*** functions, since you will not use them. Instead of calling ***freeDictionary()***, call ***hashTableFree()***, before you exit from the main function.

10. Add the necessary commands to **Makefile** to compile and link your program. You will need to link in **chainedHashTable.o** and **timeFunctions.o**.

11. Test your program with the following words. Write down the time it takes to look up each one in the columns of the table below that are labeled "Test 1".

In dictionary	***Time Test1***	***Time Test 2***	***Time Test 3***	***Not in dictionary***	***Time Test 1***	***Time Test 2***	***Time Test 3***
aspire				**ant**			
corpse				**antibiotic**			
plural				**costly**			
positive				**plant**			
salient				**seek**			
senile				**toast**			
tranquil				**vine**			
venom				**water**			
zodiac				**zoo**			

12. Now edit your program and change the value of BUCKET_COUNT to 53. Recompile, relink, and test with the same words. Write the times in the "Test 2" columns. Do you see any change in the times?

13. Change the value of BUCKET_COUNT TO 5003. Rebuild and run the tests again. Write the times in the "Test 3" columns. Do you see any changes in the times? Do you understand the pattern of information you see?

14. Upload **hashDictionary.c,** the **Makefile** and **timetable.txt** with the time values above.

15. **Extra challenge #1**. Add code that will compute and print the percentage of words that cause collisions. Print this information just before you return from ***buildDictionary()***.

16. **Extra challenge #2**. Modify the data structures and the code in **hashDictionary.c** to allow up to three definitions for any single word. The file **multiwordlist.txt** in the demo directory has some examples of words with multiple definitions. Note that the ***hashTableLookup()*** function will only find the first match of a word. So if you want to handle multiple definitions, you will need to modify some other aspect of the data structure.