CSE 373 Homework 4 Project Description

Assigned:	Wednesday, April 24, 2002
Due:	Wednesday, May 1, 2002
	At the start of class

Introduction

For this homework project, you will develop two code modules that implement character string hashing and answer some questions. All the source files that we are providing for this assignment are in a zip file on the web site. You should download and unzip the file. The questions were handed out in class; you can also get a copy from the web site.

Hash. The mainHash program reads the familiar symbol table files. Using a test definition array in one of your modules, mainHash calls your routines to build and analyze hash tables one after another. This process repeats until all the defined test cases are completed.

The main program for the homework is supplied in ADT/Hash/mainHash.c, and you are to write the individual hash table management functions as defined in ADT/include/hash.h and the actual hash functions which are kept in symbol_hash.c. As usual, the functions are based very closely on the discussion in the textbook, however, they are not exactly the same.

Grading

The 7 homework assignments of the quarter will count for a total of 50% of your class grade, and each individual homework assignment will count for about 7% of the total class grade. The grading for this project is as follows.

Questions:	10 points
Implementation:	10 points
Total	20 points

NOTE: You need to turn in several things:

1. the paper copy of your answer sheet
2. a copy of the receipt you got when you did the web turn in
3. do a web turnin of your implementation of hash.c, symbol_hash.c
4. do a web turnin of your hashtv.txt, hashpid.txt, hashbin.txt, and hashperfect.txt

Staple the answer sheet and the receipt together and turn them in on Wednesday.

Directory Structure

All of the homework projects are implementations of one of the Abstract Data Types that we discuss in class. The directory structure of the files you receive is as follows:

ADT/	top level directory
ADT/include	header files
ADT/symbols	example symbol table files
ADT/Hash	directory for the homework project
ADT/Lecture	examples from the lectures, if any
ADT/lib	precompiled binaries, if any

Program: Hash

The purpose of this program is to read a symbol table file and create a Hash table from it, then print some information about the shape of the table. Input is taken from stdin or a named symbol table file, output is to stdout (which can be redirected to a disk file).

This program requires you to implement two code modules, hash.c and symbol_hash.c. The main program and the header files are supplied. The implementation file “hash.c” is where the general purpose hash table management functions go. The implementation file symbol_hash.c is where the specific hashers for Symbols go.

Remember that most of the code you need to implement is outlined in the book or on the web site associated with the book. There is a link to the code from our syllabus page. You need to modify it for our data types and comparison strategies and integrate it into the application.

To do:

1. Create the Hash project just as you have done in the previous homeworks. For the Program arguments entry, use “..\symbols\tvsymbols.txt”.
2. Note that although the files “hash.c” and “symbol_hash.c” are provided to you, they are very incomplete, and so the project will not link correctly. All the routines that you will implement are missing.
3. Change the name that is included in the function getHashAuthor. As delivered to you, it says “Anonymous Author.” You should change that to be your own name.
4. Write the routines that are needed by mainHash as described below, rebuild the project, and run it. Debug until done. You need to add procedures to hash.c and symbol_hash.c.
5. If you run the project from Visual C, the output gets displayed in a console window. This may not be very understandable. I have supplied a batch file called winrunit.bat that runs your program (Debug\hash.exe) from a command line four times and stores the output in hashtv.txt, hashpid.txt, hashbin.txt, and hashperfect.txt. To use this batch file, get a command prompt (Start->Programs->Accessories->Command Prompt) and

use the cd command to move to the ADT\Hash directory. Then type “winrunit”. There is a similar file called runit for running the program in Linux.

6. Review the code and the output files as needed in order to answer the questions in the homework.

The function headers for the routines you need to write are in ADT/include/hash.h. Note that the functions are very similar, but NOT IDENTICAL, to the ones defined in the textbook. The functions are as follows.

File hash.c:

char *getHashAuthor(void)

Returns a text string to the caller naming the person who wrote the program. Change “Anonymous Author” to your name.

HashTable CreateHashTable(int n, Hasher F, Comparator C)

Create a new hash table data structure and return a pointer to it. The data structure is of type HashTbl, the pointer to it is of type HashTable. In addition to the table size and a pointer to the data area for the table, our hash table data structure holds a function pointer to the hash function that is being used (of type Hasher) and a comparison function (of type Comparator). These pointers are initialized in CreateHashTable, using the values passed to it by the calling program (mainHash).

I have supplied an implementation of the List management routines as part of this homework in list.c. You can use the procedure CreateListArray to create all of the lists needed by the hash table at once instead of creating each list individually, if you like. Or you can create the array and store individual list pointers in it yourself using CreateList. Either way, make sure you use the corresponding DestroyListArray or DestroyList when you destroy the hash table.

void DestroyHashTable(HashTable H)

Release all memory allocated for this hash table. Notice that this function does not release memory associated with any items that might be pointed to by entries in the table. It is the responsibility of the calling program to manage the memory associated with objects other than the table itself.

void InsertHashEntry(HashTable H, ElementType key)

Add a new list node somewhere in the hash table. Try to find the element. If it is already in the table, then return without doing anything. If not, hash the key to decide which list it should go in, then use InsertListEntry to add it to that list.

Position FindHashEntry(HashTable H, ElementType X)

Find the list node containing the given element. The function works by hashing the Element key, then looking through the appropriate list using the FindListEntry function. Recall that you can call a function using a function pointer with something that looks like (*(H->eHash))(key,H->TableSize). Also, the Comparator function pointer needed by FindListEntry was stored when the HashTable was created. FindHashEntry returns a pointer to a ListNode or NULL if not found.

ElementType GetHashEntryElement(Position P)

Retrieves the element pointer from a ListNode for use by the caller.

void PrintHashTableStatistics(HashTable H, ElementPrintLabel P)

This function prints out a summary of the state of a hash table. You shouldn't need to change it, although you can certainly add to it if you like.

File symbol_hash.c

This file contains the definition of the test cases to run and the actual hash functions that are applied to the names of the Symbols we are hashing. Each test case is defined in a simple struct called a TestCase. There is an array of these structs defining the basic test cases to run.

The entries in a TestCase are:
a. The name of the test case
b. the size of the hash table
c. the name of the function that does the actual string to HashIndex hash function
d. the Comparator to use. In this case, we always use symbolCompareByName.

If you want to implement additional test cases for your code, just add a row to the table of test cases. The main program executes all test cases before the row that starts with a NULL name field.

You need to write the hash functions hashAdder, hashShifter, hashRadix, and hashPerfect.

hashConstant is a simple minded hash function that always returns the same value. It is provided so that you can see how the header looks and the general configuration of a Hasher function.

hashAdder is the character adder given in figure 5.3 in the book.

hashShifter is given in figure 5.5.

hashRadix treats the character string as a very large base 256 number and finds the remainder when that number is divided by the table size. This is not hard to do if you think about the division algorithm. To divide the dividend by the divisor and save the remainder, you start with the left most digit of the dividend (in this case, the first 8-bit character in the string). Using the mod operator, find the remainder when that number is divided by the divisor. Multiply the remainder by the base (256), add the next character, and use the mod operator again. Repeat this process until you run out of characters. The last remainder is the remainder for the entire division.

You can check your understanding of this by doing a long division example by hand and noticing that what I have described is exactly what you do when you do a division, using 10 in place of 256.

hashPerfect is a perfect hash function for the symbols given in perfectsymbols.txt. Any function that you can come up with that maps those symbols to unique HashIndex values is okay, however, don't use any of the above three functions.

Sample output

The main program produces a small block of output for each hash table generated in the run. You need to run the program at least four times using a different symbol table each time. This is done for you in the batch files.

A typical output block is

```
Test 1: Character Add (Fig 5.3)
   19 Number of elements in hash table
  100 Number of slots in hash table
 1.00 Expected average list length
   17 Number of lists with entries
    0 Minimum list length (83 occurrences)
    2 Maximum list length (2 occurrences)
 1.12 Actual average list length
      Max length list: FindTreeNodeMin symbolPrintLabel
```

This is taken from the file hashtv.txt, the result of running the program with tvsymbols.txt as input. There are 19 elements total in the table, and the table has 100 slots (index locations) in it. Since the table is larger than the number of entries, the best possible list length is 1 (expected average). There are 17 lists that have entries in them, and 83 that do not have any entries. There are 2 lists that have two entries in them (as a result of collisions) and so the actual average list length is 1.12. The symbols in the last 2-entry list are given in the last line of the block. This can be helpful in understanding why your hash functions are hashing symbols to the same index.