Assignment 5

Version 1.10 (last update: Nov. 1, 23:00)

Changes highlighted in yellow

Due date: Thu, Nov 4, 11:59 PM

Added a description of the **freeTable** function and corrected the rubric.

Summary and Purpose

For this assignment, you will be writing a collection of C functions that operate on hash tables. You will use these to understand the underlying properties of hash tables, their strengths and weaknesses. In particular, you can compare the performance of the hash tables in this assignment to the arrays of Assignment 2, the lists of Assignment 3, and the trees of Assignment 4.

Deliverables

You will be submitting:

1) A file called **hash.h** that contains your function prototypes (see below).
2) A file called **hash.c** that contains your function definitions.
3) A **makefile** that contains the instructions to compile your code.

You will submit all of your work via git to the School's gitlab server. (As per instructions in the labs.)

This is an individual assignment. Any evidence of code sharing will be investigated and, if appropriate adjudicated using the University's Academic Integrity rules.

Structures for your assignment

You will be working with variables having the following structures which you must declare in your header file.

```
struct HashTable
{
  unsigned int capacity;
  unsigned int nel;
  unsigned int width;
  int data;
```

```
  int (*hash)( void *, int );
  int (*compar)(const void *, const void *);
}
```

This structure represents a hash table (our fourth data structure used in this course). **capacity** is the maximum number of elements in the hash table, **nel** the number of elements currently in the table, and width is the **width** each element. **data** is an address in **memsys** where an array of **capacity** number of integers are stored. Each integer in the **memsys** array is the address of a data element. **hash** is a function pointer to the hashing function used to decide where to store the data. **hash** takes two arguments: a pointer to the data to be stored or searched, and an integer which is one more than the maximum value that the hash function is allowed to produce (i.e. **capacity**). Finally, **compar** is a function which returns a value of 0 if the data stored at the two pointer arguments match.

Malloc vs memmalloc

Unless otherwise noted, you should use **memmalloc** to request all the memory required for the functions below from the **memsys** system. All **malloc** and **memmalloc** calls should be **free/memfree**'d at the appropriate time, and definitely before the program ends.

Functions on **HashTable**s

```
struct HashTable *createTable( struct memsys *memsys,
                               unsigned int capacity,
                               unsigned int width,
                               int (*hash)( void *, int ),
                               int (*compar)(const void *, const void *) );
```

(Allocate the memory for a **HashTable** and initialize the parameters.) This function will allocate sufficient memory for a **HashTable** structure using **malloc**, and copy over the values of **capacity**, **width**, **hash**, and **compar**. It will set **nel** to zero, and **data** to a newly allocated block of memory sufficiently large enough to store **capacity** many integers. The function will **memalloc** enough memory for an array of **capacity** number of integers. The address in **memsys** of this array will be stored in the data variable of the **HashTable** structure. The function will use **setval** to set the value of each element in the **memsys** array to **MEMNULL**. The address of the structure will be returned. If either **malloc** or **memmalloc** fails it will print a message to the standard error and **exit**.

`void addElement( struct memsys *memsys, struct HashTable *table, int addr );`

(Add an element to the **`HashTable`** with linear probing when a collision occurs.) If **`nel`** is equal to **`capacity`**, this function will print an error message to the standard error stream and **`exit`**. Otherwise, this function will retrieve the data located at **`addr`** into a local variable and pass the local variable and **`capacity`** to the function **`hash`** to calculate an index in the address array stored in **`memsys`** at the location **`data`**. (Hint, because we don't know the size of the data at compile time, you will need to use **`malloc`** to allocate enough memory for the data based on the **`width`** value in **`table`**.) Beginning at that index, it retrieves an integer address from the array stored at location **`data`**, and will increment the index until a **`MEMNULL`** address is found in the **`memsys`** array (if the initial index in the array is **`MEMNULL`**, the index remains the same). If the index reaches **`capacity`** in table, it should be set to zero instead and the search for a **`MEMNULL`** should continue until it gets back to the original index. Once a **`MEMNULL`** address is found (and there's guaranteed to be one if **`nel`**<**`capacity`**) at the current index in the array, it will copy the **`addr`** into the **`memsys`** array and increment **`nel`**. The memory that was **`malloc`**'ed must be **`free`**'d no matter where the new element is added.

`int getElement( struct memsys *memsys, struct HashTable *table, void *key );`
(Find an element in the **`HashTable`**, return its **`memsys`** address.) This function will pass **`key`** and **`capacity`** to the function **`hash`** to calculate an index in the address array stored in **`memsys`** at the location **`data`**. Beginning at that index, it retrieves an integer address from the array stored at location **`data.`** If the address was **`MEMNULL`** the function should return **`MEMNULL`**. Otherwise, it will retrieve the data at that address into a local variable. (See previous hint.) Next it will use the **`compar`** function to compare the local variable to the key. If they match, the function should return the address of the data that was retrieved into the local variable. If there is no match, it will increment the index in the array and try again. If the index reaches **`capacity`** in table, it should be set to zero instead and the search for a **`MEMNULL`** should continue until it gets back to the original index. The memory that was **`malloc`**'ed must be **`free`**'d no matter whether/where the element is found.

`void freeTable( struct memsys *memsys, struct HashTable *table );`
(Free the table.) This function should **`memfree`** the hash table stored in **`memsys`**, and free the **`table`** structure. It should ***not*** free the data records whose addresses are stored in the hash table.

The Last 20%

The above, constitutes 80% of the assignment. If you complete it, you can get a grade up to 80% (Good). The rest of the assignment is more challenging and will allow you to get a grade of 80-90% (Excellent) or 90-100% (Outstanding). Make sure you complete the first part well, before proceeding to the following additional part.

Write the following function:

```
int hashAccuracy( struct memsys *memsys, struct HashTable *table );
```

(Compute the hash accuracy of the contents of the **`HashTable`**.) This function will calculate the difference between every entry in the **`HashTable`**'s index and the value computed by the **`hash`** function. If the index is less than the **`hash`** function's value, it will add the index to the difference between the **`hash`** function's value and **`capacity`** because linear probing has to loop around.

You can write additional helper functions as necessary to make sure your code is modular, readable, and easy to modify. You can re-use your array.c code if you find it helpful.

Header File

Use the #ifndef…#define…#endif construct in your header file to prevent problems if your header file is included multiple times.

Testing

You are responsible for testing your code to make sure that it works as required. The CourseLink web-site will contain some test programs to get you started. However, we will use a different set of test programs to grade your code, so you need to make sure that your code performs according to the instructions above by writing more test code.

Your assignment will be tested on the standard SoCS Virtualbox VM (http://socs.uoguelph.ca/SoCSVM.zip) which will be run using the Oracle Virtualbox software (https://www.virtualbox.org/wiki/Downloads). If you are developing in a different environment, you will need to allow yourself enough time to test and debug your code on the target machine. We will NOT test your code on YOUR machine/environment.

Full instructions for using the SoCS Virtualbox VM can be found at: https://wiki.socs.uoguelph.ca/students/socsvm.

Makefile

You will create a makefile that supports the following targets:

`all:` this target should generate hash.o.

All programs and .o files must be compiled with **`clang`** and the –std=**`c99 -Wall -pedantic`** options and compile without any errors or warning.

`clean:` this target should delete all **`.o`** files.

`hash.o:` this target should create the object file, **`hash.o`**, by compiling the **`hash.c`** file.

All compilations and linking must be done with the **`-Wall -pedantic -std=c99`** flags and compile and link **without any warnings or errors**.

Git

You must submit your .c, .h and makefile using git to the School's git server. Only code submitted to the server will be graded. Do ***not*** e-mail your assignment to the instructor. We will only grade one submission; we will only grade the last submission that you make to the server and apply any late penalty based on the last submission. So once your code is complete and you have tested it and you are ready to have it graded make sure to commit and push all of your changes to the server, and then do not make any more changes to the A2 files on the server.

Academic Integrity

Throughout the entire time that you are working on this assignment. You must not look at another student's code, now allow your code to be accessible to any other student. You can share additional test cases (beyond those supplied by the instructor) or discuss what the correct outputs of the test programs should be, but do not share ANY code with your classmates.

Also, do your own work, do not hire someone to do the work for you.

Grading Rubric

createTable	1
addElement	5
getElement	5

freeTable	1
style	2
makefile	2
hashAccuracy	4
Total	20

Ask Questions

The instructions above are intended to be as complete and clear as possible. However, it is YOUR responsibility to resolve any ambiguities or confusion about the instructions by asking questions in class, via the discussion forums, or by e-mailing the course e-mail.