Assignment 5

Version 2.01 (last update: Oct. 29, 19:37)
Changes highlighted in yellow
Due date: Thu, Nov 5, 11:59 PM

This is a major revision of the assignment. Please disregard the previous version.

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;
  void **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 number of elements in the table, and **`nel`** the number of elements currently in the table.

data is an array of **capacity** number of pointers to the data accessed by the hash table. **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. Finally, **compar** is a function which returns a value of 0 if the data stored at the two pointer arguments match.

Additionally, you will be using the following structure to measure the performance of your code and count the number of memory read, memory write, malloc and free operations.

```
struct Performance
{
  unsigned int reads;
  unsigned int writes;
  unsigned int mallocs;
  unsigned int frees;
};
```

Basic function prototypes and descriptions for your assignment

```
struct Performance *newPerformance();
```

This function will allocate sufficient memory for a **Performance** structure, set **reads**, **writes, mallocs,** and **frees** to zero (yes, I realize there is technically one malloc in this function) and return the address of the structure. Your function should print an error message to the standard error stream and **exit** if the **malloc** function fails.

```
struct HashTable *createTable( struct Performance *performance,
                               unsigned int capacity,
                               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, and copy over the values of **capacity**, **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 pointers. The function will set the value of each pointer in the data array equal to **NULL**, and increment the **reads** variable in the **performance** structure by **capacity** (one for each pointer). The address of the structure will be returned. **mallocs** in the **performance** structure should be incremented by 1. If either **malloc** fails it will print a message to the standard error and **exit**.

```
void addElement( struct Performance *performance, struct HashTable *table,
                 void *src );
```

(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 pass the value of **src** and the value of **capacity** in the structure pointed to by **table**, and use the function **hash** in the structure pointed to by **table** to calculate an index in the pointer array **data**. Beginning at that index, it will increment the index until a **NULL** pointer is found (if the initial index is **NULL**, the index remains the same). If the index reaches **capacity** in table, it should be set to zero instead and the search for a **NULL** should continue. Each time it increments the index, **reads** in the **performance** structure should be incremented by 1. Once a **NULL** pointer is found (and there's guaranteed to be one if **nel<capacity**) at the current index in the array, it will copy the pointer value of **src** to the index in the array and increment **nel**. **writes** in the **performance** structure should (also) be incremented by 1.

```
int getIdx( struct Performance *performance, struct HashTable *table,
                  void *src );
```

(Find an element in the **HashTable**, return its index.) This function will pass the value of **src** and the value of **capacity** in the structure pointed to by **table**, and use the function **hash** in the structure pointed to by **table** to calculate an index in the pointer array **data**. Beginning at that index, it will use the **compar** function in table to determine whether the **src** pointer's data matches the pointer's data at the current index within **data** until:

(a) there is a match (**compar** returns 0), in which case it will return the index; or
(b) the **index** reaches **capacity**, in which case it is set to 0 and the search continues; or
(c) it gets to the original index that was returned by the **hash** function, in which case it will return **-1**.

This function will increment **reads** in the **performance** structure for each time it uses the **compar** function.

```
void freeTable( struct Performance *performance, struct HashTable *table );
```

(Free the **HashTable**.) This function will free the **data** pointer in **table** and also free **table** itself. It will increment **frees** in the **performance** structure by 1.

Derived function prototypes and descriptions for your assignment

```
void *getElement( struct Performance *performance, struct HashTable *table,
                  void *src );
```

(Find an element in the **HashTable**, return its pointer.) This function will return the pointer in the **data** array of the **table** at the index calculated by **getIdx** (above). If **getIdx** returns -1, it will return **NULL**. You will need to access **table** to implement this function.

```
void removeElement( struct Performance *performance, struct HashTable *table,
                    void *target );
```

(Remove an element from the **HashTable**.) This function will set the pointer in the **data** array of the **table** at the index calculated by **getIdx** (above) equal to **NULL**. It will increment **writes** in the **performance** structure by 1. **nel** should be decreased by one.

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 functions:

`int hashAccuracy( 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`**.

`void rehash( struct HashTable *table );`
(Move the pointers in the **`HashTable`** to get a better **`hashAccuracy`**.) This function should move the pointers in the **`HashTable`** to improve the **`hashAccuracy`**. After running this function, the table must contain all the same pointers as before, just at different indices. Your grade for this part will be based on how much you improve upon the **`hashAccuracy`**.

You can write additional helper functions as necessary to make sure your code is modular, readable, and easy to modify.

Header File

Use the #ifndef…#define…#endif construct (Lecture 02) 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 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

newPerformance	1
createTable	1
addElement	3
getIdx	4
freeTable	1
getElement	1
removeElement	1
style	2
makefile	2
hashAccuracy	2
rehash	2
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.