Programming Assignment #4: HashyHash

COP 3502, Fall 2016

Due: Sunday, December 4, *before* 11:59 PM

Abstract

In this program, you will implement hash tables with linear and quadratic probing. Some of the functions in this assignment count the number of collisions that occur over a hash table's lifetime and are designed to help show that the amortized cost of your hash table operations (insertion, search, and deletion) is really low when you have a good hash function.

By completing this assignment, you will also learn a bit about function pointers, and you will gain additional practice with dynamic memory management.

One of the very practical takeaways from this assignment is that you will learn how to use `valgrind` to test your programs for memory leaks.

When you're finished, you will have a very solid implementation of hash tables that you can use for all kinds of things.

Attachments

HashyHash.h, primes.c, testcase*.c, output*.c, test-all.sh

Deliverables

HashyHash.c

(Note: Capitalization of your filename matters!)

1. Overview

In this assignment, you will implement hash tables that use linear and quadratic probing and support insertion, search, and deletion operations. The hash tables will expand dynamically when they start to get too full, and they will keep track of how many operations have been performed on them, as well as how many collisions have occurred over the course of their lifetimes.

By tracking the number of operations performed on each hash table (insert, search, and delete), as well as the total number of collisions encountered while executing those operations, you can gather empirical data to demonstrate whether you have a good hash function that leads to few collisions on average. You can also use this data to convince yourself that, with very few collisions per operation on average, hash tables actually have the ability to yield very good runtimes for all the operations listed.

Within each hash table, you will also store an indication of what probing mechanism is being used (linear or quadratic probing), as well as a pointer to the hash function that drives the hash table. That way, by passing a single hash table pointer to a function, you give that function everything it needs to know in order to perform a variety of operations on the hash table.

To facilitate all these goals, we have defined a powerful `HashTable` struct for you. (See Section 3, "HashyHash.h," below.)

2. Function Pointers

In this assignment, each of your hash tables will be tightly coupled with a hash function that travels everywhere your hash table goes, ensuring that you never get stuck in a situation where you want to perform an operation (insertion, search, or deletion) but have no idea what hash function was being used with that hash table.

These hash functions will be written outside of your hash table code, in our test case files. However, each of your hash tables will contain a pointer to one of those hash functions.

I know what you're thinking: "Whaattttt?! A pointer to a *function*?!" Yes, this is really happening. And it's not really any more complex than having a pointer to, say, an *int* or a *double*. Here's how it works:

First, suppose you have the following function in some source file:

```
char mystery(int m, int n)
{
    return 'a' + (m * n * n * n) % 26;
}
```

Don't worry too much about what that *mystery()* function does, because it's total nonsense. The point is, it takes two integers as its only input parameters, and it returns a single character (which happens to be a character on the range 'a' through 'z').

Inside any other function, you can declare a variable that is capable of pointing to *mystery()* like so:

```
int main(void)
{
    // This just creates a function pointer. It is currently uninitialized.
    char (*foo)(int, int);

    return 0;
}
```

In general, the basic syntax for declaring a function pointer is:

```
<return_type> (*<ptr_var_name>)(<input1_datatype>, <input2_datatype>, ...);
```

Setting up your function pointer to point to a function is super easy. The name of a function is basically already a pointer (just like the name of an array is already a pointer to the base of that array). So, to set up our *foo* pointer to point to our *mystery()* function, we'd do this:

```
int main(void)
{
    // This creates a function pointer and initializes it.
    char (*foo)(int, int) = mystery;

    return 0;
}
```

Voila! That *foo* variable is now a pointer to the *mystery()* function. We can still call *mystery()* directly, or we can call it by dereferencing *foo* – just like we can set the value of an integer variable directly or change its value by dereferencing a pointer to that variable. The syntax for calling a function via a function pointer is as follows:

```
(*<ptr_var_name>)(<input1_value>, <input2_value>, ...);
```

For example:

```
int main(void)
{
    char (*foo)(int, int) = mystery;
    printf("Calling mystery indirectly: %c\n", (*foo)(5, 99));
    printf("Calling mystery directly: %c\n", mystery(5, 99));

    return 0;
}
```

You're probably asking, "Why would I ever need a function pointer?" or "Why is this happening to me?" This assignment provides one compelling example of the utility of function pointers: any time you need to strongly associate a function with a particular instance of some data structure (such as a hash table being strongly coupled with the hash function you've been using to insert keys into it), tucking a function pointer inside that data structure can help you achieve that.

3. HashyHash.h

We have included a `HashyHash.h` header file that you must `#include` in your `HashyHash.c` source file like so:

```
#include "HashyHash.h"
```

When including this header file, the capitalization of `HashyHash.h` matters! Filenames are case sensitive in Linux, and that is of course the operating system we'll be using to test your code. You also must use quotes instead of angled brackets when including that header file, and you must not modify `HashyHash.h` under any circumstances. If you're using Code::Blocks, you will need to add the `HashyHash.h` header file to your project. See the instructions in Section 10, "Compilation and Testing (Code::Blocks)."

In addition to some basic functional prototypes, the header file contains the following `#define` directives. These identifiers (`UNUSED`, `DIRTY`, `HASH_ERR`, and `HASH_OK`) will be used throughout your code. You should never hard-code a value like 0 or 1 in place of these identifiers. We might change the underlying numeric values of these to something else when writing test cases to grade your program.

```
// Use this to mark a cell unused. Do NOT use INT_MIN directly.
#define UNUSED INT_MIN

// Use this to mark a cell dirty. Do NOT use INT_MAX directly.
#define DIRTY INT_MAX

// Several functions require you to return this in the event of an error.
#define HASH_ERR 0

// Several functions require you to return this upon success.
#define HASH_OK 1

// Default capacity for new hash tables. See description of makeHashTable().
#define DEFAULT_CAPACITY 5
```

The header file also contains a `ProbingType` enum, which you will use to indicate whether each of your hash tables uses linear or quadratic probing. We used enums back in Program #1 (ChessMoves), but if you're feeling a bit rusty on the syntax, please be sure to consult a C programming reference on this topic.

The header file also contains two struct definitions that you will use in this assignment: Firstly, there's a `HashStats` struct. Each hash table has its own `HashStats` struct that contains two members, as described below:

```
typedef struct HashStats
{
   // This field keeps track of how many insert, search, or delete operations
   // take place over this hash table's lifetime.
   int opCount;

   // This field keeps track of how many collisions occur when performing
   // insert, search, or delete operations on this hash table.
   int collisions;
} HashStats;
```

After several operations have been performed on a hash table, dividing the number of collisions by the number of operations will tell you how many collisions took place per operation on average, which will give you a clearer window into the average runtimes for these operations.

Finally, `HashyHash.h` contains your `HashTable` struct:

```
typedef struct HashTable
{
   // Your hash table will store integer keys in this array.
   int *array;

   // The current capacity of your hash table (the length of 'array').
   int capacity;

   // The size of your hash table (the number of elements it contains).
   int size;

   // A pointer to the hash function for this hash table (initially NULL).
   unsigned int (*hashFunction)(int);

   // Probing type: LINEAR or QUADRATIC. Initialize to LINEAR by default.
   ProbingType probing;

   // A struct within a struct for maintaining stats on this hash table:
   // number of operations performed and number of collisions encountered.
   HashStats stats;

} HashTable;
```

4. Test Cases and the test-all.sh Script

As always, we've included multiple test cases with this assignment, which show some ways in which we might test your code. These test cases are not comprehensive. You should also create your own test cases if you want to test your code comprehensively.

We've also included a script, `test-all.sh`, that will compile and run all test cases for you. You can run it on Eustis by placing it in a directory with `HashyHash.c`, `HashyHash.h`, and all the test case files, and typing:

```
bash test-all.sh
```

5. Output

The functions you write for this assignment should not produce any output. If your functions cause anything to print to the screen, it will interfere with our test case evaluation.

Please be sure to disable or remove any `printf()` statements you have in your code before submitting this assignment.

6. Special Requirement: Using Valgrind to Test for Memory Leaks

Part of the credit for this assignment will be awarded based on your ability to implement the program without memory leaks. To test for memory leaks, you can use a program called `valgrind`, which is installed on Eustis.

To run your program through `valgrind` at the command line, compile your code with the -g flag and then run `valgrind`, like so:

```
gcc HashyHash.c testcase01.c -lm -g
valgrind --leak-check=yes ./a.out
```

You'll want to repeat the above process for all the test cases released with this assignment.

As you examine the output of `valgrind`, all you really need to know is that the magic phrase you're looking for to indicate that you have no memory leaks is:

```
All heap blocks were freed -- no leaks are possible
```

For help deciphering the output of `valgrind`, see the following link:

http://valgrind.org/docs/manual/quick-start.html

7. Prime Numbers and the primes.c Source File

The primes.c file included with this assignment has a handy function that you're welcome to copy and paste into your `HashyHash.c` file. It uses the `sqrt()` function from math.h, so you'll need to use the -lm flag to compile your source code:

```
gcc HashyHash.c testcase01.c -lm
```

8. Function Requirements

Function descriptions for this assignment are included on the following several pages. Please do not include a `main()` function in your submission.

```
HashTable *makeHashTable(int capacity);
```

Description: This function creates a hash table. Start by dynamically allocating space for a new hash table, and then properly initializing all the members of that struct. Note that by default, *hashFunction* should be initialized to `NULL`, *probing* should be initialized to `LINEAR`, and *array* should be initialized to a dynamically allocated integer array of length *capacity*. If the *capacity* parameter passed to this function is less than or equal to 0, you should use `DEFAULT_CAPACITY` (defined in `HashyHash.h`) to set the initial length of the array. All cells within the array should be initialized to `UNUSED` (which is – you guessed it! – defined in `HashyHash.h`).

Within this function, if any call to a memory allocation function fails, you should free any memory that was dynamically allocated within this function up until that point and then return `NULL`.

Return Value: Return a pointer to the new hash table (or `NULL` if any call to a dynamic memory allocation function fails).

```
HashTable *destroyHashTable(HashTable *h);
```

Description: Free all dynamically allocated memory associated with this hash table. Avoid segmentation faults in the event that *h* is `NULL`.

Return Value: `NULL`

```
int setProbingMechanism(HashTable *h, ProbingType probing);
```

Description: Within the hash table, set the *probing* field to either `LINEAR` or `QUADRATIC`, depending on the value of the *probing* parameter passed to this function.

Return Value: If the function is successful, return `HASH_OK` (defined in `HashyHash.h`). If the function fails (either because *h* is `NULL` or because the function received a *probing* value other than `LINEAR` or `QUADRATIC`), return `HASH_ERR` (also defined in `HashyHash.h`).

```
int setHashFunction(HashTable *h, unsigned int (*hashFunction)(int));
```

Description: Within the hash table, set the *hashFunction* field to the hash function that is passed as the second parameter to this function.

Return Value: Return `HASH_ERR` if *h* is `NULL`. Otherwise, return `HASH_OK` (even if *hashFunction* is `NULL`).

```
int isAtLeastHalfEmpty(HashTable *h);
```

Description: Determines whether the hash table is at least half empty.

Runtime Restriction: This must be an O(1) function.

Return Value: Return 1 if the hash table is at least half empty. If the hash table is less than half empty (more than half full), or if *h* is `NULL`, return 0. (You may assume that the hash table's *capacity* member will not be zero, although it's good practice to guard against this, just in case.)

```
int expandHashTable(HashTable *h);
```

Description: Within the hash table, replace the array (whose length is currently *capacity*) with a longer array. If the hash table is set to use linear probing, the length of the new array should be (*capacity* * 2 + 1). If the hash table is set to use quadratic probing, the length of the new array should be the first prime number that is greater than or equal to (*capacity* * 2 + 1). (Note that there is a handy function in `primes.c` that will help you generate prime numbers, and you're welcome to copy and paste it into your source code for this assignment.)

After creating an expanded array, you should re-hash all the values from the old array (starting from index 0 and working your way up to index *capacity* - 1) into the new array. In doing so, you should increment the hash table's *collisions* counter for each new collision that takes place as you re-hash the old values into the new array. However, you should *not* increase the hash table's *opCount* counter for each of those insertion operations. (If our goal is to count the average number of collisions per operation performed on this hash table, then to be fair, if inserting an element causes the table to expand, all the collisions that result should really be attributed to that *one* insertion operation – not all of the insertion operations that occur as a side effect when the hash table is expanded.) So, this function should affect the *collisions* count, but it should not affect the *opCount* of the hash table at all.

After the array is expanded, all of its empty cells should be initialized to `UNUSED`, and it should not have any cells marked as `DIRTY`.

Within this function, be sure to free memory as needed in order to avoid memory leaks.

Return Value: Return `HASH_OK` if the expansion of the hash table is successful. If it fails for any reason (such as the failure of a memory allocation function), return `HASH_ERR`.

```
int insert(HashTable *h, int key);
```

Description: First and foremost, before even inserting the new element into the hash table, check that the hash table is still at least half empty – regardless of whether the table is using linear or quadratic probing. (You have a handy function that can do that for you.) If not, expand the hash table as described above. (You have a handy function to do that for you, as well!)

Using linear or quadratic probing (depending on the value of the hash table's *probing* field) in conjunction with the hash table's *hashFunction*, insert *key* into the hash table's array. Be aware that *hashFunction* might return very large values. You are responsible for modding by the table length to prevent array index out-of-bounds errors with the hash values produced by the hash function.

While probing for a spot to insert *key*, increment the hash table's *collisions* counter every time a collision occurs. (Note that finding an empty spot for the key should *not* count as a collision.)

After you've inserted the key, increment the *size* and *opCount* for this hash table. (Each of those should get incremented by 1.)

Return Value: Return `HASH_OK` if the operation is successful. Otherwise, return `HASH_ERR`. (Some potential causes of failure include: *h* is `NULL`; within *h*, the *hashFunction* is `NULL`; or expanding the hash table fails. As a failsafe measure, I also coded up my version of this function to return `HASH_ERR` if it goes through *capacity* number of iterations without finding an open spot to insert the key. That should not be possible if the table is expanding properly, but it's a solid backup plan to help avoid an infinite loop just in case there's a bug in the hash table expansion function.)

```
int search(HashTable *h, int key);
```

Description: Using linear or quadratic probing (depending on the value of the hash table's *probing* field) in conjunction with the hash table's *hashFunction*, search for *key* in the hash table's array. Be aware that *hashFunction* might return very large values. You are responsible for modding by the table length to prevent array index out-of-bounds errors with the hash values produced by the hash function.

While probing for the *key*, increment the hash table's *collisions* counter every time a collision occurs with an element that is not the key. (Note that hitting an `UNUSED` cell should *not* count as a collision, and should allow us to return from the function right away. However, finding a `DIRTY` cell should count as a collision.) Note: The number of cells probed by this function should not exceed *capacity*. If you perform *capacity* iterations of your probing loop without finding the key, break out and return.

Be sure to increment the *opCount* for this hash table by 1 regardless of whether you find the key or not.

Return Value: If the key is found, return the index where it resides in the hash table. Otherwise, return -1. If *h* is `NULL` or the *hashFunction* member of the hash table is `NULL`, return -1. If there are multiple instances of *key* in the hash table, simply return the index of the first one you encounter while probing.

```
int delete(HashTable *h, int key);
```

Description: Using linear or quadratic probing (depending on the value of the hash table's *probing* field) in conjunction with the hash table's *hashFunction*, search for *key* in the hash table's array. If the key is found, delete it by setting the cell's value to `DIRTY`.

Be aware that *hashFunction* might return very large values. You are responsible for modding by the table length to prevent array index out-of-bounds errors with the hash values produced by the hash function.

While probing for the *key*, increment the hash table's *collisions* counter every time a collision occurs with an element that is not the key. (Note that hitting an `UNUSED` cell should *not* count as a collision, and should allow us to return from the function right away. However, finding a `DIRTY` cell should count as a collision and also requires that we continue searching through the table.) Note: The number of cells probed by this function should not exceed *capacity*. If you perform *capacity* iterations of your probing loop without finding the key, break out and return.

Be sure to increment the *opCount* for this hash table by 1 regardless of whether you find the key or not, and decrement the *size* member if the key is successfully deleted.

If you get a bit crafty, you can outsource most of the work for this function to your *search()* function. However, this is not strictly necessary.

Side Note: A solid implementation of hash tables might shrink the array if it starts to get too sparse (in order to cut back on wasted memory), but we will avoid doing that in this assignment (simply to make our lives a bit easier).

Return Value: If *key* is deleted successfully, return the index where it resided in the hash table. Otherwise, return -1. If *h* is `NULL` or the *hashFunction* member of the hash table is `NULL`, return -1. If there are multiple instances of *key* in the hash table, simply delete the first one you encounter while probing.

```
double difficultyRating(void);
```

Return Value: Return a double indicating how difficult you found this assignment on a scale of 1.0 (ridiculously easy) through 5.0 (insanely difficult).

```
double hoursSpent(void);
```

Return Value: Return an estimate (greater than zero) of the number of hours you spent on this assignment.

9. Compilation and Testing (Linux/Mac Command Line)

To compile multiple source files (`.c` files) at the command line:

```
gcc HashyHash.c testcase01.c -lm
```

By default, this will produce an executable file called `a.out`, which you can run by typing:

```
./a.out
```

If you want to name the executable file something else, use:

```
gcc HashyHash.c testcase01.c -lm -o HashyHash.exe
```

...and then run the program using:

```
./HashyHash.exe
```

Running the program could potentially dump a lot of output to the screen. If you want to redirect your output to a text file in Linux, it's easy. Just run the program using the following command, which will create a file called `whatever.txt` that contains the output from your program:

```
./HashyHash.exe > whatever.txt
```

Linux has a helpful command called `diff` for comparing the contents of two files, which is really helpful here since we've provided several sample output files. You can see whether your output matches ours exactly by typing, e.g.:

```
diff whatever.txt output01.txt
```

If the contents of `whatever.txt` and `output01.txt` are exactly the same, `diff` won't have any output. It will just look like this:

```
seansz@eustis:~$ diff whatever.txt output01.txt
seansz@eustis:~$ _
```

If the files differ, it will spit out some information about the lines that aren't the same. For example:

```
seansz@eustis:~$ diff whatever.txt output01.txt
1c1
< fail whale :(
---
> Success!
seansz@eustis:~$ _
```

10. Compilation and Testing (Code::Blocks)

The key to getting your program to include a custom header file in Code::Blocks (or any IDE) is to create a project. Here are the step-by-step instructions for creating a project in Code::Blocks and importing `HashyHash.h` and the `HashyHash.c` file you've created (even if it's just an empty file so far).

1. Start Code::Blocks.

2. Create a New Project (*File → New → Project*).

3. Choose "Empty Project" and click "Go."

4. In the Project Wizard that opens, click "Next."

5. Input a title for your project (e.g., "HashyHash").

6. Pause to reflect on how much you've learned this semester. You are an impressive human being.

7. Choose a folder (e.g., Desktop) where Code::Blocks can create a subdirectory for the project.

8. Click "Finish."

Now you need to import your files. You have two options:

1. Drag your source and header files into Code::Blocks. Then right click the tab for **each** file and choose "Add file to active project."

– or –

2. Go to *Project → Add Files…*. Then browse to the directory with the source and header files you want to import. Select the files from the list (using CTRL-click to select multiple files). Click "Open." In the dialog box that pops up, click "OK."

You should now be good to go. Try to build and run the project (F9). As a friendly reminder: Even if you develop your code with Code::Blocks on Windows, you ultimately have to transfer it to the Eustis server to compile and test it there. See the following page (Section 9, "Compilation and Testing (Linux/Mac Command Line)") for instructions on command line compilation in Linux.

11. Grading Criteria and Submission

11.1. Deliverables

Submit a single source file, named `HashyHash.c`, via Webcourses. The source file should contain definitions for all the required functions (listed above), as well as any auxiliary functions you need to make them work. Don't forget to `#include "HashyHash.h"` in your source code. Your program should compile on Eustis with both of the following:

```
gcc -c HashyHash.c
gcc HashyHash.c testcase01.c -lm
```

11.2. Special Restrictions

1. ***Important!*** The use of global variables might result in an automatic zero. Embrace the given function parameters and struct definitions, please. As always, you must also avoid the use of mid-function variable declarations and system calls (such as `system("pause")`).

2. ***Important!*** Your `HashyHash.c` file **must not** include a `main()` function. If it does, your code will fail to compile during testing, and you will not receive credit for this assignment.

3. Do not submit the `HashyHash.h` header file with your code. You should only submit `HashyHash.c`. We will use our own version of the header file when testing your program.

4. Be sure you don't write anything in `HashyHash.c` that conflicts with what's given in `HashyHash.h`. Namely, do not try to define a `HashTable` struct in `HashyHash.c`, since your source file will already be importing the definition of that struct from `HashyHash.h`.

5. Be sure to include your name and NID as a comment at the top of your source file.

11.3. Grading

The *expected* scoring breakdown for this programming assignment is:

80%	Correct output for test cases used in grading
10%	No memory leaks; passes `valgrind` tests
10%	Comments and whitespace

Note! Your program must be submitted via Webcourses, and it must compile and run on Eustis to receive credit. Programs that do not compile will receive an automatic zero.

Your grade will be based primarily on your program's ability to compile and produce the *exact* results expected. Even minor deviations will cause your program's output to be marked as incorrect, resulting in severe point deductions. The same is true of how you name your functions and their parameters. Please be sure to follow all requirements carefully and test your program thoroughly.

Note also that your functions should not print anything to the screen. If they do, it will interfere with the output we generate while testing, resulting in incorrect test case results and an unfortunate loss of points.

11.4. Closing Remarks

Start early. Work hard. Ask questions. Good luck!