Lab 5: Hash Tables in C

In this lab we're making hashtables in C. We've been using C arrays for the past two labs and now that Lab 4 has given us a working linked list in C, we can implement a hash table.

You will be doing the following:

- TODO1 implement the hash table
- (EXTRA CREDIT) TODO2 No memory leaks
    - valgrind will display the following message:

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

Hash Table Refresher

Put simply, a hash table is just an array that is able to store data associated with a key. In Assignment 4 we were storing data using a `String` for our keys and an `Object` for our value, which would eventually be a list. The reason we want to make an array of lists instead of just using an array is it is much more efficient with storage space since we don't need to make space in an array for every possible hash value. Additionally we don't have to worry about running out of space in the array to deal with *collisions*, two keys being turned into the same array index.

Recall that in Assignment 4 we needed to call the Java function `hashCode()` on our `String` key and then turned the hash code (named `hashCode` in the PDF) into an index into out hash table (making it between `0` and `tableSize - 1`.

```
String key = "hello";
int hashCode = key.hashCode();
int arrayIndex = Math.abs(hashCode) % tableSize;
```

At a high level this code is converting a `String` into a valid array index. What `hashCode()` is doing behind the scenes involves, at least in C, getting a bit closer to the actual data representation in memory.

Hashing in C

Our `hash()` will actually be made up of three functions:

- `int hash(Dictionary D, char* key)`
    - The function that will return an array index
- `unsigned int pre_hash(char* input)`
    - This function converts a string into an `unsigned int`
- `unsigned int rotate_left(unsigned int value, int shift)`
    - The function responsible for moving bits to semi-randomize the resulting key

Note: You will only need to call `hash(Dictionary D, char* key)` when you want to compute a hash value.

These functions, included in the `Dictionary.c` file are defined as follows:

```
unsigned int rotate_left(unsigned int value, int shift) {
      int sizeInBits = 8 * sizeof(unsigned int);
      shift = shift & (sizeInBits - 1);
      if(shift == 0){
            return value;
      }
      return (value << shift) | (value >> (sizeInBits - shift));
}
```

```
unsigned int pre_hash(char* input) {
      unsigned int result = 0xBAE86554 //this may correspond to an
address
      while (*input) {
            result ^= *input++;
            result = rotate_left(result, 5);
      }
      return result;
}
```

```
int hash(Dictionary D, char* key) {
      return pre_hash(key) % D->tableSize;
}
```

Note: You will only need to call `hash(Dictionary D, char* key)` when you want to compute a hash value.

A few things to note about these functions are that they are manipulating the actual bits of our data in memory, specifically the `&`, `<<`, `>>`, and `^=` operators which are defined as follows:

- `&` is *bitwise* AND
    - Instead of looking at a truth value it looks at the binary representations of its arguments and creates a new binary number using a logical AND for each bit (if you have taken CE12 you will be familiar with this).
- `<<` and `>>` are bit shifts
    - All data is represented as some sequence of bits in memory. Doing a left or right shift moves the bit sequence stored in a given variable (on the left) the specified number of bits (on the right) and fills in 0s when there is no more of the original value to shift into a bit.
- `^=` is a bitwise exclusive or assignment
    - This operator performs the XOR operation between its two arguments and assigns the result to the left hand side.

Wikipedia has a more in depth explanations of bitwise operators if you're curious (https://en.wikipedia.org/wiki/Bitwise_operations_in_C)

We have also included a program called `HashDemo.c` to show how the above three functions work. To compile it, simply enter the following:

```
bash$ gcc -std=c99 -o HashDemo HashDemo.c
```

To run the program:

```
./HashDemo
```

It will show you a breakdown of strings and their indices as created by the above functions, the functionality of `rotate_left()` on a single unsigned int, and the number of bits a single character takes up.

A Note on typedefs

In this lab we'll also be seeing more of these statements and it's one of the built in abstractions in C. We've done `typedef struct node_type {...} Node;` to allow us to refer to a `Node` without having to repeatedly say `struct`. Inside of `Dictionary.h` and `Dictionary.c` are two other `typedef`s, one for a `struct DictionaryObj*` and one for a `struct EntryObj*`. These allow you to refer to pointers by whatever name was given after the `typedef`, in this case `Dictionary` is another name for a pointer to a `DictionaryObj` and `Entry` is another name for a pointer to an `EntryObj`.

TODO1: Implementing the Hash Table

Similar to Assignment 4 you'll be implementing something resembling the `MyHashtable` class but not quite because, once again, C does not have classes. The basic blueprint for our C version should look very familiar:

- A `struct` defined in the `Dictionary.c` called `Entry` (a lot like the `Node` struct you implemented in Lab 4). It will once again store Key/Value pairs:
    - `char* key`
    - `char* value`
    - As before it should have an associated `Entry newEntry(char* key, char* value)` function
    - As we're in C it should also have an associated `void freeEntry(Entry* pE)` function (more details about this in TODO2).
- Again like in Assignment 4 your hash table struct will have properties that aren't directly accessible outside of the `Dictionary.c` file:
    - `int tableSize` - the size of the array being used by the hash table
    - `int size` - the number of key/value entries currently in the hash table
    - `List** table` - an array of pointers to `List`s. As in Assignment 4, the reason for this is to handle cases where two keys have the same hash value.
- You'll be implementing the following functions in `Dictionary.c`:
    - `Dictionary newDictionary(int tableSize)`
        - Allocates space for a new `Dictionary`, initializes the new `Dictionary`'s variables, and returns it
    - `void freeDictionary(Dictionary* pD)`
        - Frees all data associated with the `Dictionary` pointed at by `pD` (more detail about this in TODO2)

- `int isEmpty(Dictionary D)`
  - Returns an `int`; booleans in C are integers. Traditionally a `1` represents `true` and `0` represents `false`
- `int size(Dictionary D)`
  - Returns the number of key/value pairs in D
- `void insert(Dictionary D, char* key, char* value)`
  - Adds a new key/value pair into the dictionary using a linked list to deal with collisions (translating Assignment 4's `put()` is a good place to start)
- `char* lookup(Dictionary D, char* key)`
  - Returns the value in `Dictionary D` associated with `key` (translating Assignment 4's `get()` is a good place to start)
- `void delete(Dictionary D, char* key)`
  - Removes the `Entry` associated with `key` in `Dictionary D` (translating Assignment 4's `remove()` is a good place to start)
  - Remember that because we're in C we need to free whatever was deleted here.
- `void makeEmpty(Dictionary D)`
  - Removes all Entries from Dictionary D (translating Assignment 4's `clear()` is a good place to start)
  - Remember that because we're in C we need to free memory we're not using anymore ourselves
- `void printDictionary(FILE* out, Dictionary D)`
  - Prints the contents of `Dictionary D` to a file. More on this below.

Each of the above functions has a brief description in `Dictionary.h` including specifics about return values and preconditions (what each function assumes to be true when it's called).

Making an Array of Lists and Notes About calloc

Because we're in C, we don't get to use the nice Java array syntax:

```
MyLinkedList[] table = new MyLinkedList[tableSize];
```

instead we need to do what we did in Lab 3 and use `calloc()`:

- First we need a pointer to a `List`:

```
List** table
```

- Next, inside `newDictionary()` we'll need to set our newly instantiated Dictionary's table pointer to point to something on the heap:

```
calloc(tableSize, sizeof(List*))
```

The reason we need table to be of type List** is because it's an array of pointers to `List`s. Even though table looks a little strange compared to a Java array declaration and initialization, we can still access whatever is in it using the same array indexing we've been working with (assume D is of type Dictionary):

```
D->table[0] = make_list();
```

To make a new instance of a `DictionaryObj` we still need to call `malloc()`, which requires a pointer:

```
Dictionary D = malloc(sizeof(DictionaryObj));
```

Note: One important thing to remember about `sizeof()` is you want to give it the `struct` not the pointer. So when we want to make a new dictionary, we don't want to give `sizeof()` `Dictionary`, we want to give it `DictionaryObj` because remember that `Dictionary` is really just another name for a `DictionaryObj*`.

A note on `printDictionary()`

The function `void printDictionary(FILE* out, Dictionary D)` is a little more flexible than what we've been doing in C up to this point since it's able to take any arbitrary file and write to it. Instead of using `printf()` which only lets us print to `stdout`, we can use `fprintf()` which functions almost identically to `printf()` with one difference. A general call to `printf` looks like:

```
printf("<format string>", <argument1>, <argument2>,..., <argumentn>);
```

while a general call to `fprintf` looks like:

```
fprintf(<file name>, "<format string>", <argument1>,...,
<argumentn>);
```

Similar to Lab 2 in Java, C treats standard in (`stdin`) and standard out (`stdout`) as files and you'll notice that the calls to `printDictionary()` in `DictionaryClient.c` look like this:

```
printDictionary(stdout, A);
```

Compiling and Running

As in Lab 4, we have a few different files we need to compile at once to have our object file actually work as intended. We also now have two different files for you to use to test your hash table: `DictionaryClient.c` and `DictionaryClient2.c`. To compile them you'll need to run the following:

```
bash$ gcc DictionaryClient.c Dictionary.c list.c --std=c99 -o
DictionaryClient
```

to make the object file for `DictionaryClient` and

```
bash$ gcc DictionaryClient2.c Dictionary.c list.c --std=c99 -o
DictionaryClient2
```

To make the object file for `DictionaryClient2`.

DictionaryClient is fairly simple and should print the following to the console:

```
two bar
five happy
one foo
three blah
seven blue
four galumph
```

```
six sad
key="one" value="foo"
key="two" value="bar"
key="three" value="blah"
key="four" value="galumph"
key="five" value="happy"
key="six" value="sad"
key="seven" value="blue"
two bar
five happy
four galumph
six sad
key="one" not found "(null)"
key="two" value="bar"
key="three" not found "(null)"
key="four" value="galumph"
key="five" value="happy"
key="six" value="sad"
key="seven" not found "(null)"
false
4
true
```

`DictionaryClient2` does something a little more fun and loads each line of its own .c file into the hash table and prints the hashtable to a file named `DictionaryClient2-out`, which should have the following contents:

```
line 53:         key = &keyBuffer[keyBufferOffset];
line 36:         value = &valBuffer[valBufferOffset];
line 63:      freeDictionary(&A);
line 18:      char* value;
line 45:         key = &keyBuffer[keyBufferOffset];
line 11:   #define MAX_LEN 180
line 28:      while( fgets(line, MAX_LEN, in)!=NULL ){
line 55:         insert(A, key, value);
line 21:      int keyBufferOffset = 0, valBufferOffset = 0;
line 38:         valBufferOffset = valBufferLength;
line 9:    #include"Dictionary.h"
line 65:       free(valBuffer);
line 47:          keyBufferOffset = keyBufferLength;
line 31:          lineNumber++;
line 2:     // DictionaryClient2.c
line 13:    int main(int argc, char* argv[]){
line 40:          // put label in keyBuffer
line 57:          valBufferOffset += (strlen(value) + 1);
line 23:       char line[MAX_LEN+1];
line 50:       // put keys and values in dictionary A
line 67:       fclose(out);
line 49:
line 33:          line[lineLength] = '\0';  // overwrite newline '\n'
with null '\0'
line 60:       printDictionary(out, A);
line 15:       FILE* in = fopen("DictionaryClient2.c", "r");
line 42:          labelLength = strlen(label);
line 59:
line 5:
line 70:    }
line 25:       int i, labelLength, lineLength, lineNumber = 0;
line 52:       for(i=0; i<lineNumber; i++){
line 69:       return(EXIT_SUCCESS);
line 35:          valBuffer = realloc(valBuffer,
valBufferLength*sizeof(char) );
line 62:       // free memory and close files
line 17:       char* key;
line 44:          keyBuffer = realloc(keyBuffer,
keyBufferLength*sizeof(char) );
line 7:     #include<stdlib.h>
line 10:
line 27:       // read input files
line 54:          value = &valBuffer[valBufferOffset];
line 20:       char* valBuffer = NULL;
line 37:          strcpy(value, line);
line 8:     #include<string.h>
line 64:       free(keyBuffer);
line 19:       char* keyBuffer = NULL;
line 46:          strcpy(key, label);
line 30:          // put line in valBuffer
line 1:
//-------------------------------------------------------------------
----------
line 12:
line 29:
line 56:          keyBufferOffset += (strlen(key) + 1);
line 22:       int keyBufferLength = 0, valBufferLength = 0;
line 39:
line 66:       fclose(in);
line 48:       }
line 32:          lineLength = strlen(line)-1;
line 3:     // Another test client for the Dictionary ADT
line 14:       Dictionary A = newDictionary();
line 41:          sprintf(label, "line %d:\t", lineNumber);
line 58:       }
line 4:
//-------------------------------------------------------------------
----------
line 24:       char label[MAX_LEN+1];
line 51:       keyBufferOffset = valBufferOffset = 0;
line 68:
line 34:          valBufferLength += (lineLength+1);
line 61:
line 16:       FILE* out = fopen("DictionaryClient2-out", "w");
line 43:          keyBufferLength += (labelLength+1);
line 6:     #include<stdio.h>
line 26:
```

Memory Management

We've now done two labs in C and have a bit of experience doing manual memory management so you'll notice that our `freeDictionary()` function is asking for something a little odd, a `Dictionary*` (remember that `Dictionary` is another name for `DictionaryObj*`). The same is true of `freeEntry()` wanting an `Entry*` rather than an `Entry` (another name for `EntryObj*`). The reason for this is **to allow the function that is responsible for freeing memory to also be able to set the pointer that is**

pointing to the freed memory to NULL. Put more simply, it's an emulation of the way deleting something in Java works since in Java you don't have to worry about a reference pointing to deallocated memory.

What does this mean for you? If you look at either of the `DictionaryClient.c` files you'll notice that their calls to `freeDictionary()` use `&A`. `&A` means we're getting the address of `A`, which is of type `Dictionary` (really a `DictionaryObj*`). So when we say `freeDictionary(&A)`, we're passing it the pointer to a pointer, which is what `freeDictionary()` wants.

When we're actually writing our destructor method, instead of simply freeing what we were passed as we have done before with List:

```
void free_node(Node* node){
    free(node->data);
    free(node);
}
```

as an example, we need to dereference the pointer to the pointer. In `freeDictionary()`'s case `pD` is a pointer to a pointer to an instance of a DictionaryObj:

```
&pD->pD->instance of DictionaryObj
```

If all we have is `&pD`, how do we get to our instance? We can use the `*` operator to dereference `&pD` like this:

```
(*pD)
```

Once we have (`*pD`), we can treat it just like a regular pointer! But because we also have `&pd` (the pointer to the pointer), this allows us to set `pD` to `NULL` *inside the `freeDictionary()` function*.

Note: Remember that the `List` we built in Lab 4 uses a `void*` for data and since we'll be storing an `Entry` in a `Node` struct, we'll need to loop through the `List` calling `freeEntry()` on each Node's data pointer. Once we finish freeing each `Entry`, **then** we're allowed to call `free_list()`.

Reminders About valgrind

Make sure you're running valgrind as follows for both DictionaryClient and DictionaryClient2:

```
valgrind --leak-check=full -v ./DictionaryClient

valgrind --leak-check=full -v ./DictionaryClient2
```

The important things to look out for (after segmentation faults) is that the number of allocations and the number of frees should be the same. Generally when these are equal you won't have any memory leaks, though the *in use* report at exit part of the HEAP SUMMARY will give you a more detailed breakdown.

Grading

TODO 1: 85 points

- `Entry` and `Dictionary` structs completed (10 points)
- `newDictionary()`, `newEntry()`, `freeDictionary()`, `freeEntry()` implemented (15 points)
- `size()` and `isEmpty()` implemented (5 points)
- `delete()` and `makeEmpty()` implemented (20 points)
- `insert()` and `lookup()` implemented (25 points)
- `printDictionary()` implemented (10 points)

TODO 2: (EXTRA CREDIT) 10 points (no memory leaks)
Style: 15 points

As a note, it is far more important to turn in code that compiles and is a TODO or only partially does a TODO, than it is to attempt a TODO and make your code uncompilable. If you code doesn't compile, you will receive 40 points off, so make sure your code compiles before you submit it (even if you are missing a TODO).

Turning in the Lab

- Create a directory with the following name: `<student ID>_lab5` where you replace `<student ID>` with your actual student ID. For example, if your student ID is 1234567, then the directory name is `1234567_lab5`.

- Put a copy of your edited `Dictionary.h, Dictionary.c , list.h, list.c, DictionaryClient.c, and DictionaryClient2.c` files in the directory.
- Compress the folder using zip. Zip is a compression utility available on mac, linux and windows that can compress a directory into a single file. This should result in a file named `<student ID>_lab5.zip` (with `<student ID>` replaced with your real ID of course).
- Upload the zip file through the page for Lab 5 in canvas.