C track: assignment 7

Goals

This week, we are going to move beyond a discussion of built-in features of the C language (we've covered most of them already) and continue discussing how to implement fundamental data structures in C. The data structure we will implement for this assignment is called a hash table. Hash tables are among the most common and most useful general-purpose data structures. Many scripting languages, including perl and python, include hash tables as fundamental data structures, but in C you have to make them yourself. We will see that the combination of C's structs, arrays and pointers will make this relatively easy, if not completely painless.

Language concepts covered this week

None.

Other concepts covered this week

Hash tables.

Hash tables

A hash table is a generalization of an array. An array can be thought of as a data structure that maps integers to items of a given type. A hash table can map arbitrary (constant) items of one type (known as "keys") to items (not necessarily constant) of another type (known as "values"). Hash tables are most often used to map strings to data, and that's what we'll be doing here. You can think of this as if you had an array which was indexed by a string value instead of by an integer. Note that hash tables are not the only way to build such a mapping, but they are one of the most efficient.

Hash functions

The basic idea behind a hash table is this. You take your key and process it with a special function called a "hash function". This function will transform any key into a non-negative integer with a bound which is proportional to the size of the table. In our case it will transform a string into a value between 0 and 127. Ideally, a hash function will map different keys to different hash values so that a random population

of keys will give a uniform distribution of hash values.

The data structure

The hash table itself is essentially an array of linked lists. In fact, we define it as a struct which contains an array of linked lists as its only field (in a real hash table implementation, there would be more fields, so this is realistic; also, it gives you experience working with more complex data structures). The array indices are the same as the hash values (so in our case the array length is exactly 128), and the linked lists are initially empty (set to `NULL`). Note that the linked lists are represented as pointers to a node struct, just like in lab 6. We'll refer to this array below as the "node pointer array".

Adding items to a hash table

We add items to a hash table as a "key-value pair". Let's say our keys represent movie names and our values represent the rating of the movie in stars (0-5). A typical key-value pair might be ("The Matrix", 5). First, we compute the hash value for "The Matrix" (you'll see how below) and get (say) 47. The 47th location in the node pointer array will initially be empty (*i.e.* it will contain a `NULL` pointer). We then create a new linked list node. The node's type will be different from what it was last lab; here it will have *three* fields: a field called `key` (a `char *` which holds the address of the string used as a key), a field called `value` which holds the `int` value, and a "next" pointer as before. We put the (address of the) key string into the `key` field (here, "The Matrix"), and the value into the `value` field (here, 5). We then link it to location 47 in the hash table array by putting a pointer to the node in the node pointer array at location 47. The "next" field of the linked list node itself will be `NULL`. Later, we might want to add another entry, say ("The Matrix Revolutions", 0). First, we compute the hash value of "The Matrix Revolutions". Now one of two things could happen:

- If the hash value is not 47, we just create another linked list node and add it to the place in the array corresponding to the hash value, just as we did for the first item.

- If the hash value is 47 again (which ideally it won't be), we create a new linked list node. Then we have to add it to the old linked list at array position 47. We do this by inserting the new node into the old linked list at any point (the order doesn't matter). Usually we either put it at the front or the back of the list (putting it at the front is actually faster and easier to program). To do this we'll have to adjust some pointer values.

Now you can see why hash values should be distributed randomly over the set of all possible keys: if all keys hashed to 47, we would end up with just a single linked list at one location in the array and nothing anywhere else. In that case, looking up

elements would take time proportional to the number of entries (as it does with linked lists). If most of the linked lists are either empty or have only one or two values, lookup will require a constant (small) amount of time. Of course, as the number of elements in the hash table increases in size eventually all the slots will be occupied even with a perfect hash function. At this point you should probably create a bigger hash table, but we won't worry about this in this lab.

Retrieving items from a hash table

To retrieve an item from a hash table means to retrieve the value given the key. The way this is done is as follows:

- Compute the hash value for the key.
- Find the location in the node pointer array corresponding to the hash value.
- Search the linked list at that location for the key. In general, there will be a very small number of items in the linked list (usually one or none), so the search will not take long.
- If the key is found, return the value. If not, return a "not found" value (this will become clearer later). If there was no linked list at that location in the array, also return the "not found" value.

As we said above, this means that our linked list nodes have to have three elements:

1. the key
2. the value associated with the key
3. a pointer to the next node, which may be `NULL` if there is no next node.

Program to write

Your program will use a hash table to count the number of occurrences of words in a text file. The basic algorithm goes as follows. For each word in the file:

1. Look up the word in the hash table.
2. If it isn't there, add it to the hash table with a value of 1. In this case you'll have to create a new node and link it to the hash table as described above.
3. If it is there, add 1 to its value. In this case you don't create a new node.

At the end of the program, you should output all the words in the file, one per line, with the count next to the word *e.g.*

```
cat 2
hat 4
green 12
eggs 3
ham 5
algorithmic 14
```

```
deoxyribonucleic 3
dodecahedron 400
```

Also, make sure that you explicitly free all memory that you've allocated during the run of the program before your program exits. This includes:

1. all the nodes in the linked lists that the hash table's node pointer array points to,
2. the hash table's node pointer array itself,
3. the hash table struct itself,
4. the string keys used in the linked list nodes (allocated in the `main()` function (see below)).

The memory leak checker will help ensure that you get this right ;-)

The hash function

The hash function you will use is extremely simple: it will go through the string a character at a time, convert each character to an integer, add up the integer values, and return the sum modulo 128 (but don't use a magic number for this). This will give an integer in the range of [0, 127]. This is a poor hash function; for one thing, anagrams will always hash to the same value. However, it's very simple to implement. Note that a value of type `char` in a C program can also be treated as if it were an `int`, because internally it's a one-byte integer represented by the ASCII character encoding. If you like, you can convert the `char` to an `int` explicitly using a type cast. However, a string of characters is **not** an `int`, and you can't use `atoi` to convert it into one (mainly because most of the keys will be words, not string representations of numbers). You also shouldn't try to type cast the string to an `int` (it'll just cast the address into an integer, which might work but it's not what you want). A string is an array of `chars`, which is not the same as a single `char`; keep that in mind. So you'll need a loop to go through the string character-by-character.

Other details

Since a value associated with a key will always be positive, if you search for a key and don't find it in a hash table, just return 0. That's the special "not found" value for this hash table.

For simplicity, the program assumes that the input file has one word per line (we've written this code for you). The order of the words you output isn't important, since the test script will sort them.

Things to watch out for

You should design your hash table functions to be generic *i.e.* independent of the

purpose those functions are used for. Specifically, **don't** assume that particular values to be assigned mean anything special (for instance, don't assume that assigning a value of 1 means that a key doesn't exist in the table, just because that's how it will be used in the rest of the program). You should think of the hash table functions as being part of a hash table library that you are writing. Functions in a library could be used in many different programs, so the fewer the assumptions you make about how the functions will be used, the better. The only exception to this is the rule that says if you are searching for a key and it isn't found, you should return zero. That's not very generic (you can imagine a hash table in which it would be OK to store a value of zero), but it makes the program simpler.

There is one memory leak which may be hard to get rid of which involves the `set_value` function. You should assume that the key value that is passed to `set_value` is newly-allocated, and so it either has to go into the hash table or it has to be freed.

Supporting files

To simplify your task, we're supplying you with:

- a header file. **Please read this file!** It describes the data structures for the hash table and also a useful `#define`. A lot of the problems that students have with this lab are a direct result of not reading this file carefully. In particular, make sure you understand the `hash_table` struct, because if you don't, the program will be impossible to get right.
- a template file for your code; this file will be called "hash_table.c".
- a main file
- a Makefile
- a test input file
- a test output file
- a test script
- The memory leak checker: memcheck.c and memcheck.h.

All you have to do is to fill in the definitions in the template file. Don't change anything in the other files.

Testing your program

The program name is `test_hash_table`. It takes one command-line argument, which is an input file of words. Here, the test input file is called `test.in`, so to run the program you would type:

```
% test_hash_table test.in
```

This will print a list of word/number pairs to `stdout`. To check if this is the right list,

type `make test` to run the test script on your program. The test script will run your program on `test.in` and will compare the output of the program to the correct output. Don't hand in the program until it passes the test. Also make sure that there are no memory leaks (the memory leak checker will report these).

For extra credit...

As we mentioned above, you aren't supposed to change anything other than the definitions in the template file. However, the resulting design is not the cleanest possible design from the standpoint of managing memory. If you *were* allowed to change something in the file `main.c` as well as in the template file, how would you do this in order to create a cleaner design? Why would this design be better than the existing design? You can put your answer in comments in the `main.c` file. Don't change any of the code in the `main.c` file, however; just say what you *would* do. Also, leave a note in the comments at the top of the `hash_table.c` file that you've modified `main.c`, so we know we should look at it.

To hand in

The file `hash_table.c`, and the `main.c` file if you have done the extra credit question (and *only* if you have done that!).

References

- Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Cliff Stein
- Algorithms in C by Robert Sedgewick
