4

Lab 2: Hashtables in C

In this assignment, we will explore `hsearch`, a hash table library for C that comes with Linux.

The `hsearch(3)` library is a hash table implementation for C. Because `hsearch` is a part of the POSIX standard,[1] you can find it preinstalled on many operating systems.

The fact that `hsearch` is written in C means that it has a few quirks you should be aware of. The fact that its designers apparently thought that nobody would ever need more than one hash table is also a little strange. Nevertheless, `hsearch` is a fast hash table implementation, and after you understand its idiosyncrasies, it is relatively easy to use.

`hsearch(3)` only has three function calls: `hcreate`, which creates the table, `hsearch` which both searches the table and adds entries, and `hdestroy`, which deallocates the table. For a list of the quirks, be sure to see section 4.6 at the end of this handout.

[1] POSIX stands for "Portable Operating Systems Interface." It defines what C APIs, libraries, and other standard components an operating system must have in order to be UNIX-like. Standards like POSIX are extremely important in a world that has many operating systems: Linux, macOS, FreeBSD, Solaris, and so on. If you write a program with the POSIX standards in mind, then it is likely that your code will run on many operating systems with little to no customization.

4.1 *Learning Goals*

In this lab, you will learn:

- how to work with the `hsearch(3)` hash table;
- get some practice manually allocating and deallocating data.

4.2 Requirements

Collaboration. This is an ungraded assignment. You may work with whomever you wish.

Language and Libraries. Your solution must be written using C. Only use the built-in C libraries. Do not download any additional libraries.

4.3 Inputs and Outputs

The file, `passwords.db`, is a (real-ish) leaked password database of the following form:

$$\texttt{username}_1\texttt{,password}_1$$
$$\texttt{username}_2\texttt{,password}_2$$
$$\ldots$$
$$\texttt{username}_n\texttt{,password}_n$$

We want to answer the question: how often are passwords reused? To do this, you will build a hash table that maps each unique password you encounter to a simple count. These counts should then be printed out as follows:

For this lab, the order of the output is not important.

$$\texttt{password}_1\texttt{: count}_1$$
$$\texttt{password}_2\texttt{: count}_2$$
$$\ldots$$
$$\texttt{password}_n\texttt{: count}_n$$

You can check that your implementation did the right thing by saving the program's outputs to a file and then comparing them against the database. For example,

```
$ ./password_counter > outputs.txt
$ head outputs.txt
pipik53: 1
Pohled267: 2
AbpoHuQEpp: 3
martinstraka17: 2
271987: 1
pamela: 1
HB65FeScow: 1
cacuvo39: 2
tulen777: 2
JLurRn9F6F: 2
$ grep AbpoHuQEpp passwords.db
Finochio,AbpoHuQEpp
Antonio_Crespo,AbpoHuQEpp
Blahonovsky6,AbpoHuQEpp
```

Three users share the password `AbpoHuQEpp`, so you can see that our `password_counter` program correctly counted them.

4.4 *Starter Code*

Starter code is provided for this lab. Download the starter, unzip it, and then you can work with it.

```
$ wget https://csci331.s3.amazonaws.com/hashtable-starter.zip
$ unzip hashtable-starter.zip
```

If you don't have `wget` or `unzip` installed on your computer, you can install them with `apt`.

4.5 *How to Start*

The starter code includes a number of sections marked `TODO` in the comments. You should replace these `TODO`s with your own code.

I also provide a small set of `#define` statements at the top of the starter code. These should provide some important clues about how to work with your table.

You can learn about the `hsearch(3)` hash table by typing `$ man 3 hsearch`. Note that the `man` page includes sample code.

4.6 *Gotchas*

There are some important caveats about the `hsearch` implementation that you should be aware of.

- You can have at most *one* hash table at a time. Consequently, you are never given the ability to save a pointer to this data structure.
- Create a new database using the `hcreate` function. `hcreate` takes a parameter for the maximum size of the table. For performance reasons, you should set it to be 25% larger than the maximum number of elements than you expect to store in the table.
- Storing and retrieving from the table uses the same function, `hsearch`. The behavior of this function depends on the `action` argument, either `ENTER` or `FIND`.

- Elements cannot be deleted from the table, but they can be updated. Because this is C, you should think carefully about you really do need to reinsert elements when updating. An alternative approach is to modify the data value through a pointer.
- When you store in the hash table, what is stored is a *copy* of a pointer to a `key` and a *copy* of a pointer to a `data` item.
- The type of the `key` is always a string pointer, namely a `char *`.
- The type of the `data` is always a `void *`, which essentially means "a pointer to something." For example, if you store a string pointer in the `data` field, you will need to cast it like `(char *)` when you read it out.
- `hdestroy` only deallocates the keys in the table, not the data. You will need to manually deallocate the data yourself.

  When storing data in your table, I suggest that you store copies of key and data values. For example, supposing `char* key` and `char* value` are initialized elsewhere,

  ```
  ENTRY e, *ep;
  e.key = strdup(key);
  e.data = strdup(value);
  ep = hsearch(e, ENTER);
  ```

  `strdup` makes a copy of a string by calling `malloc` and then copying the string data into the new location.

  Be aware that your own `e.data` value in this lab will not be a `char*` as in the example. It should be a `int*`.

- Finally, remember to verify that the `hsearch` function is successful. The `man` page explains how to check for success.