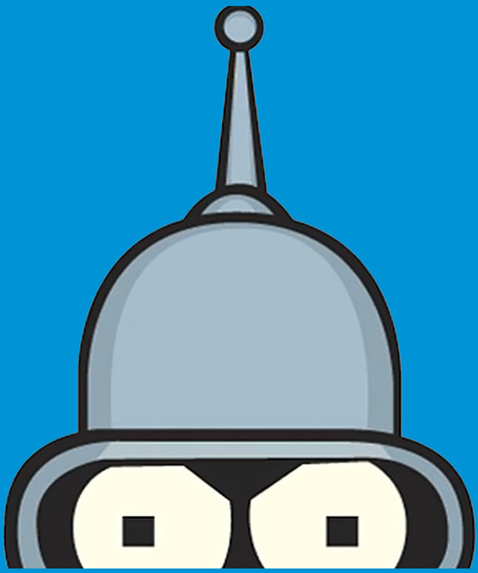
{EPITECH}

SECURED

ELEMENTARY PROGRAMMING IN C



SECURED

binary name: libhashtable.a
language: C
compilation: via Makefile, including re, clean and fclean rules
Authorized functions: write, malloc, free

- ✓ The totality of your source files, except all useless files (binary, temp files, objfiles,...), must be included in your delivery.
- ✓ All the bonus files (including a potential specific Makefile) should be in a directory named bonus.
- ✓ Error messages have to be written on the error output, and the program should then exit with the 84 error code (0 if there is no error).

You're standing in the middle of your impeccably organized laboratory at Epitech. Tools are precisely aligned, electronics gleam under the neon lights, and your once chaotic workspace is now a model of efficiency. But as you contemplate your robot plans for the big robot battle tournament, a sense of foreboding comes over you.

In this technology-driven world, your plans aren't just sheets of paper or digital files. They are the keys to your future success, coveted secrets that could lead you to victory in the robot arena. The question then becomes: how do you protect them from prying eyes or ill-intentioned rivals?

This is where your new quest begins: "Secured". A project not only crucial to your progress at Epitech, but essential to ensuring the security of your most precious creations.

Your first task is to build a hash table, similar to a high-tech digital safe, where every robot blueprint is stored with cryptographic precision. Imagine a labyrinth of data where only those with the right key can navigate.

Your second challenge is to learn the art of hashing. Like a digital spell, you'll turn your plans into encrypted enigmas, indestructible and incomprehensible to anyone who doesn't have the magic password.

By mastering these skills, you not only prepare for the tournament, but also forge the digital armor that will keep your treasures safe in the cyber-danger-filled real world.

— Narrator —

The project

Objectives

Now that you've got a tidy workbench, you're going to want to start **securing** and preciously storing certain information, like your robot plans for example. The aim of this project is to introduce you to hash tables.

✓ Developing your own **hash function**
✓ Creating your own **hash table**

As with the first 2 module projects, you'll need to be able to handle a large number of entries in your hash table!

Skills

You've already seen a lot through the Setting Up and Organized projects alone!

✓ 1-2d arrays
✓ Linked lists
✓ File handling
✓ Library handling
✓ void * pointers
✓ Dynamic programming algorithm
✓ Sorting algorithm
✓ Argument handling

As you'll have noticed, the idea of the **"Elementary Programming in C"** module is to retrace all the fundamental mechanics of C, some of which we've already seen in CPool! Just wait until the 2nd semester!

Through this last project, the idea is to conclude the semester with 4 new skills:

✓ Merging arrays and linked lists
✓ Pointer to functions
✓ Library compilation (it's been a long time since CPool, hasn't it?)
✓ Hash algorithm

Delivery

Hash Function

The hash function is the central element of a hash table, defining which table index should be used to store the data. A hash function is a mathematical function that converts any numerical data into an output string comprising a fixed number of characters or a positive integer.

Hash functions are the basic tools of modern cryptography that are used in information security to authenticate transactions, messages, and digital signatures !

One of your first goals will be to develop your own hash function, which you can then use for your hash table.

Your own hash function:

```c
int hash(char *key, int len);
```

There are **4 criteria** for determining whether a hash function is good:

1. The output string is entirely determined by the hashed data (constants are allowed but must be justified!)
2. All input data is used in hashing
3. Two almost identical inputs give very different output values
   And most complicated of all:
4. The hash function evenly distributes the data in the hash table!

In the library you render, there will be a hash function like this one. `key` being the value to be hashed and `len` being the size of the hash table (some hash methods take table size into account).

Usually, the hash function gives us the index directly, modulated by the size of the table, but in this case we'll return the hashed value and do the modulation ourselves later.

There are already many well-known hashing methods, such as MD5, SHA-1 or SHA-256, but in our case, in order to obtain an index, methods such as the **mid-square method** or the **binning method** may be of interest!

Here's an example of use with a hash function quickly developed for the occasion. We're trying to hash dummy folder names from our work for the robot tournament:

```c
int main(void)
{
    int size = 5;

    printf("Vision: %d\n", hash("Vision", size));
    printf("Perception: %d\n", hash("Perception", size));
    printf("Fight: %d\n", hash("Fight", size));
    printf("Pathfinding %d\n", hash("Pathfinding", size));
    return 0;
}
```

Terminal

```
~/B-CPE-110> ./a.out
Vision: 1769199557
Perception: 1290224677
Fight: 1751607738
Pathfinding: 1296709920
```

In this case, our hash function is not good enough.
Why not? Because it doesn't meet criteria **3.** and **4.**!

Once you've developed your first hash function, as simple as it is, and your hash table, the next step is to spend some time finding the right method!

Hash tables are highly appreciated by developers for the speed with which they can access their data. Other programming languages also have usable hash tables, such as C++'s `unordered_map` (formerly called `hash_map`).

In the library you're rendering, there will be two functions for creating and destroying a hash table:

Creating and deleting the hash table:

```
hashtable_t *new_hashtable(int (*hash)(char *, int), int len);
void delete_hashtable(hashtable_t *ht);
```

The `new_hashtable` function returns a pointer to a structure imposed on you, a `hashtable_t` structure. What's more, you'll notice that the `new_hashtable` takes a `hash` parameter, which is nothing other than a pointer to a function! This allows each hash table to be assigned its own function.

It's up to you to decide what to put in the structure, and what type of data structure to use for the table... !

4 other hash table manipulation functions are also requested!

Handling the hash table:

```
int ht_insert(hashtable_t *ht, char *key, char *value);
int ht_delete(hashtable_t *ht, char *key);
char *ht_search(hashtable_t *ht, char *key);
void ht_dump(hashtable_t *ht);
```

With the `ht_insert`, `ht_delete` and `ht_search` functions, don't forget to do some error handling, the return value types aren't there for nothing!

The hashtable.h file attached to the project looks like this:

```
~/B-CPE-110> cat hashtable.h
#ifndef HASHTABLE_H
        #define HASHTABLE_H

typedef struct hashtable_s {
        // Your code here
} hashtable_t;

// Hash function
int hash(char *key, int len);

// Create & destroy table
hashtable_t *new_hashtable(int (*hash)(char *, int), int len);
void delete_hashtable(hashtable_t *ht);

// Handle table
int ht_insert(hashtable_t *ht, char *key, char *value);
int ht_delete(hashtable_t *ht, char *key);
char *ht_search(hashtable_t *ht, char *key);
void ht_dump(hashtable_t *ht);

#endif /* HASHTABLE_H */
```

Dump

The ht_dump function displays the state of the hash table. As a reminder, a hash table is an array in which we store indexed information.

```c
int main(void)
{
    hashtable_t *ht = new_hashtable(&hash, 4); // We create a table of size 4

    ht_dump(ht); // We display the current state of the table
    return 0;
}
```

Terminal

```
~/B-CPE-110> ./a.out
[0]:
[1]:
[2]:
[3]:
```

Insert

The `ht_insert` function is used to insert a new element into the hash table. It's the `key` parameter that needs to be hashed to find out the index in which to store the `value`.

```c
int main(void)
{
    hashtable_t *ht = new_hashtable(&hash, 4);

    ht_insert(ht, "Vision", "./Documents/Tournament/Modules/Vision");
    ht_insert(ht, "Kratos", "./Trash/Hollidays_Pics/.secret_folder/kratos.ai");
    ht_insert(ht, "<3", "+33 6 31 45 61 23 71");
    ht_dump(ht);
    return 0;
}
```

Terminal

```
~/B-CPE-110> ./a.out
[0]:
> 13116 - +33 6 31 45 61 23 71
[1]:
> 1769199557 - ./Documents/Tournament/Modules/Vision
[2]:
> 1952572858 - ./Trash/Hollidays_Pics/.secret_folder/kratos.ai
[3]:
```

In this example, the hash of the `key` "Vision" gave `1769199557` then `% len`, the size of the array, to know where to store the `value`.

It's important to point out that you won't get the same results with the same `key`, since everyone develops their own hash function!

Delete

The `ht_delete` function will delete a value from the table. We hash the `key`, go to the given index and then delete the value associated with this hash.

```c
int main(void)
{
    hashtable_t *ht = new_hashtable(&hash, 4);

    ht_insert(ht, "Vision", "./Documents/Tournament/Modules/Vision");
    ht_insert(ht, "Kratos", "./Trash/Hollidays_Pics/.secret_folder/kratos.ai");
    ht_insert(ht, "<3", "+33 6 31 45 61 23 71");
    ht_delete(ht, "Vision");
    ht_dump(ht);
    return 0;
}
```

Terminal

```
~/B-CPE-110> ./a.out
[0]:
> 13116 - +33 6 31 45 61 23 71
[1]:
[2]:
> 1952572858 - ./Trash/Hollidays_Pics/.secret_folder/kratos.ai
[3]:
```

Search

The `ht_search` function takes a `key` as a parameter, then hashes it to access the index. If the value matches the hash, then the `value` is returned.

```c
int main(void)
{
    hashtable_t *ht = new_hashtable(&hash, 4);

    ht_insert(ht, "Vision", "./Documents/Tournament/Modules/Vision");
    ht_insert(ht, "Kratos", "./Trash/Hollidays_Pics/.secret_folder/kratos.ai");
    ht_insert(ht, "<3", "+33 6 31 45 61 23 71");
    ht_dump(ht);

    printf("Looking for \"%s\": \"%s\"\n", "Kratos", ht_search(ht, "Kratos"));
    return 0;
}
```

```
Terminal
~/B-CPE-110> ./a.out
[0]:
> 13116 - +33 6 31 45 61 23 71
[1]:
> 1769199557 - ./Documents/Tournament/Modules/Vision
[2]:
> 1952572858 - ./Trash/Hollidays_Pics/.secret_folder/kratos.ai
[3]:
Looking for “Kratos”: “./Trash/Hollidays_Pics/.secret_folder/kratos.ai”
```

Collisions

What happens if we use the `ht_insert` function but the hash that is generated once modulo is an index that is already filled? We call this a collision!

Well, there are generally 2 ways of resolving a collision:

- ✓ We can insert our value in the next available index
- ✓ **Or we can create a list at the index to store multiple elements.**

This is what we're going to do here, and it's called a **Separate Chaining Hash Tables**.

```c
int main(void)
{
    hashtable_t *ht = new_hashtable(&hash, 4);

    ht_insert(ht, "Vision", "./Documents/Tournament/Modules/Vision");
    ht_insert(ht, "Kratos", "./Trash/Hollidays_Pics/.secret_folder/kratos.ai");
    ht_insert(ht, "<3", "+33 6 31 45 61 23 71");
    ht_insert(ht, "</3", "+33 7 51 49 01 38 11"); // The same index as "<3"
    ht_dump(ht);
    return 0;
}
```

Terminal

```
~/B-CPE-110> ./a.out
[0]:
> 3354428 - +33 7 51 49 01 38 11
> 13116 - +33 6 31 45 61 23 71
[1]:
> 1769199557 - ./Documents/Tournament/Modules/Vision
[2]:
> 1952572858 - ./Trash/Hollidays_Pics/.secret_folder/kratos.ai
[3]:
```

If you try to store a value to a hash already present, you have to update the value!

v1

{EPITECH}