# Memoria de la tarea 1 - Hashing

El ordenador es Windows 10. El IDE es CodeBlocks 16.01 con el compilador por defecto “GNU GCC Compiler”.

Empiezo el proyecto copiando el prototipo que deja el profesor en el campus (main.c, hashing.c, hashing.h).

Primero creo el proyecto vacío en lenguaje C. Luego creo los archivos anteriores vacíos y copio y pego el código que deja el profesor en cada uno de ellos, respectivamente. Formateo un poco el código (con herramientas automáticas) por deformación profesional estoy acostumbrado a otro tipo…

Cambio el nombre de las funciones por “h\_<nombre función>” para que no me den problemas al compilar el proyecto en CodeBlocks por utilizar nombres reservados.

Arreglo un pequeño fallo en el código, en la función “show”, que mostraba un warning en el printf.

Intento dejar la funcionalidad de “buscar” a la función “H”, que es la que hará la iteración de las colisiones. La función “search” (“h\_search” ahora) se dedica solamente a llamar a la función Hashing “H” que es la que devolverá la posición verdadera del elemento si existe. Dejo la función “h\_search” por si en un futuro necesitase más funcionalidad que no se deba poner en la función Hashing.

Las funciones ahora devuelven el entero 1 si ha ido bien y el -1 si ha ido mal (pensando que lo mismo en un futuro puedan devolver -2, -3… si hay varios errores posibles).

La función H ahora llama a la nueva función G, que implementa la prueba lineal finalmente.

Para que todo funcione bien, a la función H le paso un código dependiendo si quiero insertar o buscar, para que se comporte de forma diferente al encontrar la clave que se quiere. Lo mismo se podría mejorar el rendimiento poniendo enteros en vez de strings, pero de momento lo dejo así para comodidad mía.

Como pensé antes, en la función de inserción devuelvo ahora varios códigos. Un 1 cuando se ha insertado, un -1 cuando no se ha insertado por falta de espacio, y un -2 cuando ya existe ese elemento en la tabla. No pongo estos códigos en DEFINE (como el de LIBRE o BORRADO) porque solo valdrían para esta función, en “h\_search” no tendría sentido el mismo código de “Ya estaba insertado”. Dejo comentado qué es qué y así no hay problemas.

Compruebo que funciona y añado más elementos, cambio los números para ver que borra e inserta bien.

Ahora modularizo el código para poder implementar el ejercicio 2: prueba dependiente de clave.

Creo los archivos “h\_clave”, “h\_lineal”, “h\_utils” (tanto .c como .h). Separo la función H y G de las demás (insert, search…) y creo una función “H\_lineal”, otra “H\_clave” y las G de la misma forma. Las funciones insert, search y demás que no sean “H” o “G” las pongo en “utils.h”.

A las funciones de “utils” que utilicen la función “H” les agrego un parámetro nuevo llamado “prueba”, que indicará si se utiliza la H\_lineal o H\_clave. Ya que se supone que estas funciones no dependen del algoritmo.

Como al hacer la modularización se llaman a las mismas cabeceras desde diferentes sitios, pongo los llamados “guards” en los “.h” para no repetir definiciones.

Cambio el nombre a los archivos, les quito la parte “h\_” del inicio, para no confundirme con la extensión “.h”.

Arreglando la prueba lineal para que al sumar la “i” en la función G se haga el módulo.

Implemento la función G\_clave de forma correcta. Creo la variable “d” que sea la división, para que no haya problemas pongo el tamaño de la tabla a 8 y condiciono la “d” para que sea impar y mayor que cero siempre.

Una vez he terminado de hacer el algoritmo, prueba a hacerlo con los mismos valores a papel y comparar. Todo está bien en ambos, lineal y clave.

Ahora me dispongo a cambiar el struct para que contenta la información de los coches, según la tarea. Además de hacer la función que convierta la matrícula en un número.

Como para hacer la función he hecho el campo “unsigned int” la constante LIBRE y BORRADO ya no me sirven con ese valor negativo. Por suerte, como esta función siempre va a dar un valor numérico bastante grande, puedo poner LIBRE como un 1 y BORRADO como un 0, con seguridad de que nunca la clave va a valer eso.

Modifico las funciones insert, search, etc. para que en vez de pasar el “key” se pase la matrícula.

Termino de darle los retoques a las funciones, probar que todo vaya bien, etc.

Una vez hecho esto, aumento el tamaño de la tabla según la tarea y leo los coches del CSV.

Pongo la función de leer fichero que dio el profesor en el fichero “hashing”. Lo renombro a “h\_insert\_fichero”. Creo un registro por cada línea del CSV y lo relleno, luego llamo a la función de inserción con este registro para no repetir código.

Después de un rato intentando ver por qué el factor de la tabla lineal es 1 y el factor de la dependiente de clave no es 1, me doy cuenta de que incrementé el valor de la dependiente de clave en 6 elementos (los que hay libres).

Cambio los mensajes de debug al fichero “hashing”, según el código que devuelva cada función y quito los mensajes del “main”.

Después de unos retoques al código y varias pruebas, funciona bien la inserción de la prueba lineal y la de clave.

Mejoro los logs y las llamadas a las funciones, para que todo se haga en su sitio.

Implemento el contador de intentos para cada función. Cambio el INT que devolvía antes la función H y G, para que sea un struct de dos INTs. La posición (como antes) y los intentos.

Implemento el cálculo de tiempo para todas las funciones, ya que estoy. El código lo tomo de la página:

<https://stackoverflow.com/questions/459691/best-timing-method-in-c>

Hago pruebas, y parece que el de clave es más rápido, pero con esta cantidad de coches a penas se nota en mi ordenador.

Veo que todo funciona bien y luego hago el menú de la aplicación. Hasta ahora he estado probando cada cosa comentando lo que no quería ejecutar. Para dejar la aplicación decente hago un menú para poder elegir el tipo de hashing y la acción a realizar (insertar, buscar, borrar…).

Pruebo que todo funcione para entregar la práctica.