FACULTAD DE INGENIERIA
Universidad de Buenos Aires

TDA — Hash

[7541/9515] Algoritmos y Programación II

Segundo cuatrimestre de 2021

Alumno:	Hurtado, Daniel
Número de padrón:	107404
Email:	dhurtado@fi.uba.ar

1. Introducción

Este trabajo práctico tiene como objetivo la continuación en las implementaciones de TDA (Tipo de dato abstracto), proceso iniciado con el TDA Lista y TDA ABB anteriores. Recordando las anterior explicaciones, podemos retomar la definición de un TDA como la creación de estructuras y funcionalidades definidas que no hagan referencia a datos específicos o contenido irrelevante para su propósito (abstracción), brindando la posibilidad de ser reutilizados en infinitud de situaciones que requieran dichas funcionalidades, sin importar la índole de la información, datos o procesos manejados en donde se requiera implementarla. En este TDA, se implementará las definiciones de Hash, teniendo también como objetivo el análisis y posibilidad de uso de TDAs externos/previamente creados con la intención de profundizar el conocimiento en la reutilización de TDAs (uno de sus propósito principales) dentro de sí.

Siguiendo de igual manera el desarrollo enfocado en TDD (Test Development Drive, o en español, desarrollo guiado por pruebas), cuya finalidad es realizar la implementación conociendo de

antemano cual sería su correcto y esperado funcionando, a la vez que dicha implementación seguirá los criterios establecidos por la cátedra y el contrato facilitado (tda_hash.pdf y hash.h).

2. Teoría

Previo a profundizar en el Hash a implementar y recordatorio de las definiciones de lista (a ser utilizado en la implementación en el punto siguiente), podemos permitirnos un recordario del por qué de su existencia, en general de los TDA. En un lenguaje como C, cuyos tipos de datos y funcionalidades son limitadas con respecto a otros lenguajes más completos en estos aspectos como Python o Javascript; los programadores carecemos de opciones nativas de utilizar, o en este caso, organizar de mejor manera gran cantidad de información de manera accesible y ordenada, sin depender a su vez de las limitaciones de memoria de estos lenguajes fuertemente tipados. Debido a esta razón, se realizan implementaciones de nuevos tipos de datos que nos permitan organizar, agrupar y manejar datos de diferentes tipos.

1. [Recordatorio] Lista - Dinámica/Simplemente enlazada

Una lista consiste en una sucesión de datos (que se encuentran en posiciones determinadas), que según la implementación, pueden ser relacionados entre si (su posición con respecto a otro), por contener la información de cada uno de sus elementos en un mismo bloque de memoria (tanto estática o dinámica), o mediante nodos.

Tiene como finalidad ser capaces de albergar datos de manera ilimitada (con la limitación del hardware) que se puedan encontran en diferentes ubicaciones de la memoria, permitiendo así un mejor aprovechamiento de la misma y mejor rendimiento a la hora de agregar elementos por no tener la necesidad redimensionar nuestra lista de manera constante.

En este caso, hablando de listas dinámicas y simplemente enlazadas debemos conocer primero lo que es un nodo. Son tipos datos estructurados que contienen además del dato o elemento guardado, una referencia o relación entre otros nodos, los cuales en conjunto formarían la lista. Pueden verse como structs de la siguiente manera, variando según la implementación requerida:

```
typedef struct nodo {
  void* elemento;
  nodo_t* siguiente;
} nodo_t;
```

De esta manera podemos definir una lista de nodos como un tipo de dato que contiene una referencia al primer elemento de la misma, pudiendo contener (según su implementación) a una referencia a su último elemento o la cantidad de elementos que contiene, un ejemplo de esta podría ser la siguiente:

```
typedef struct lista {
  int cantidad_elementos;
  nodo_t* primero;
} lista_t;
```

Así mismo, podemos definir lo que podemos hacer con sus datos, de manera abstracta:

```
- Agregar elementos a la lista (en cualquier posición).
- Obtener un elemento de la lista (cualquiera sea su posición).
- Sacar un elemento de la lista (cualquiera sea su posición).
- Iterar el contenido de la lista, tanto interna como externamente.
```

Además claro, de la creación y destrucción de la misma, y funciones utilitarias como tamaño, está vacía, etc.

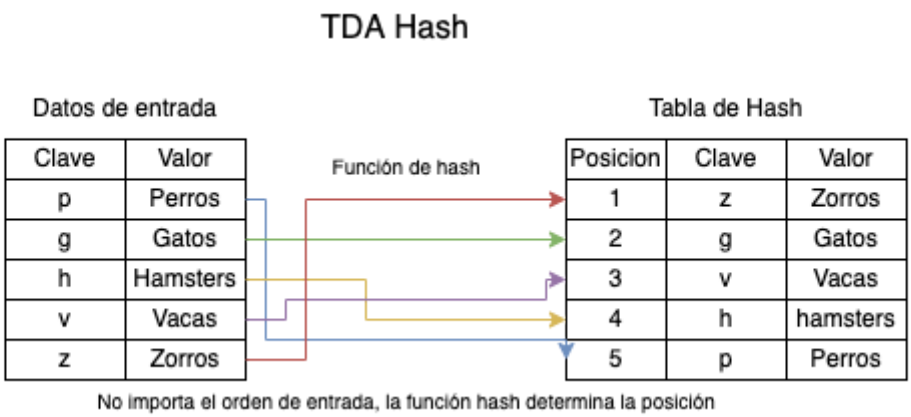
2. Hash

Un hash es un tipo de dato abstracto utilizado para almacenar datos/valores/elementos en sí, identificados cada uno de ellos a su vez mediante una clave o key, la cual es utilizada para establecer su posición dentro del hash, e identificarlo en una búsqueda, obtención, eliminación. La clave de un dato contenido en un hash es única, por lo que, en caso de insertar un nuevo elemento con la misma clave, en lugar de insertarlo, actualizará el elemento/valor/dato contenido en clave.

El hash cuenta con una tabla o hashtable de posiciones donde se albergan los datos almacenados (par clave/valor), cuya posición de los elementos es determinada mediante una función de hash, la cual determina inequívocamente y de manera consistente, la posición del dato según algún criterio tomando en base a la clave del mismo. Por ejemplo: la clave 123 puede ser dirigida a la posición en la tabla 50, y siempre que dicha función vuelva a ser ejecutada con la misma clave, deberá volver a apuntar hacia la posición 50.

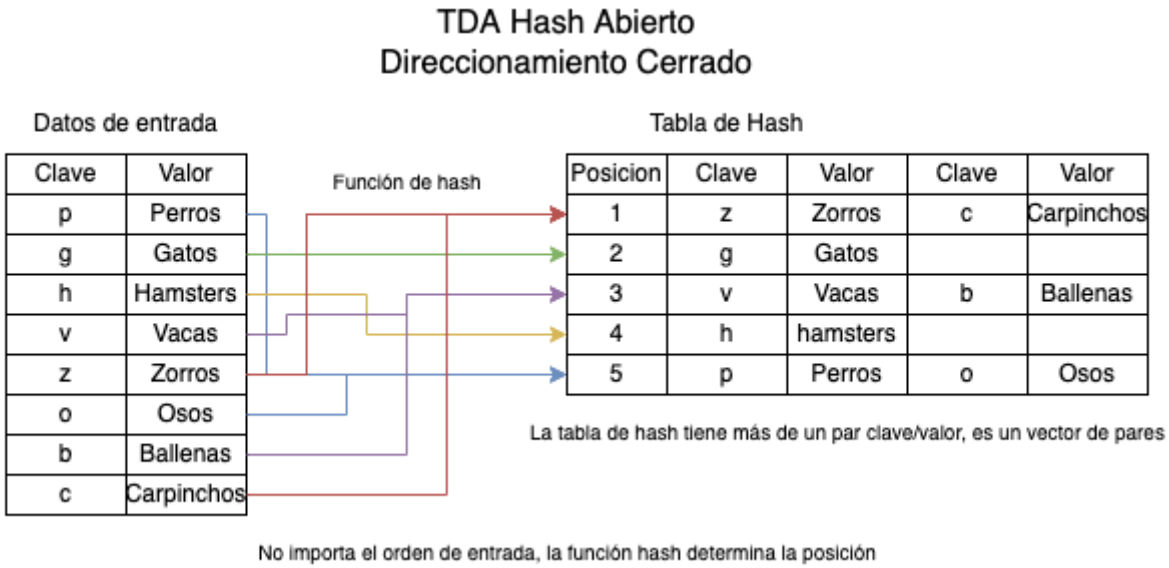
Este criterio de posicionamiento de elementos, permite optimizar la insersión, búsqueda y eliminación de elementos en un aproximado (casos de uso promedios reales) de O(1). Sin embargo, al igual que ocurría con el Tda ABB, existe la posibilidad de mutar su forma dependiendo de casos extremos o mal aprovechamiento del mismo hasta llegar a ser O(n).

Por ejemplo, en el siguiente diagrama podemos observar un hash simple como una serie de datos con clave/valor son insertados en el hash, en el cual no importa el orden de inserción, ya que es la función de hash que determina la posición a la que van los objetos insertados:

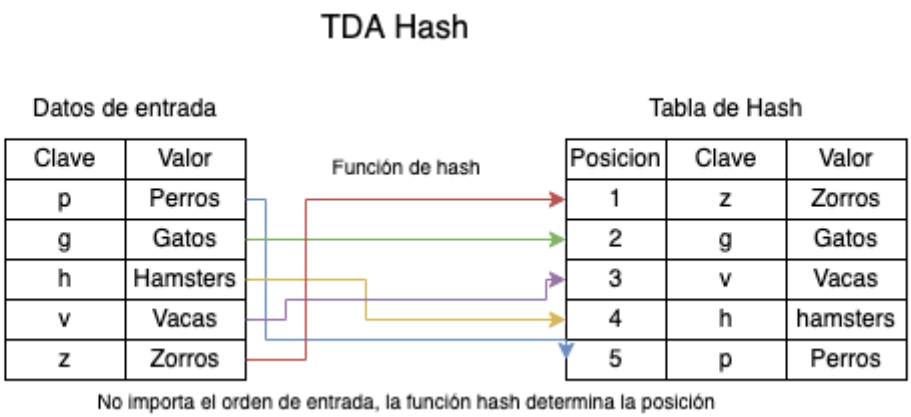


A su ves, los hash pueden dividirse en diversos tipos, tales como:

- Hash de Redireccionamiento Cerrado o Hash Abierto: En este caso de hash, el tamaño de la tabla no coincide con la cantidad máxima de datos permitidos para almacenar, debido a que cada una de estas posiciones a su vez admite más de un dato, ya sea mediante uso de vectores estáticos o dinámicos, o nodos enlazados (nativos o a su vez mediante implementaciones/tdas externos). Esta variante permite garantizar el conocimiento de donde insertar/obtener/quitar un elemento dado. Como ejemplo de esta implementación podemos ver el siguiente diagrama representado con vectores, donde tenemos más vectores que posiciones, y algunos de ellos comparten una misma posición, guardándose juntos en el vector:

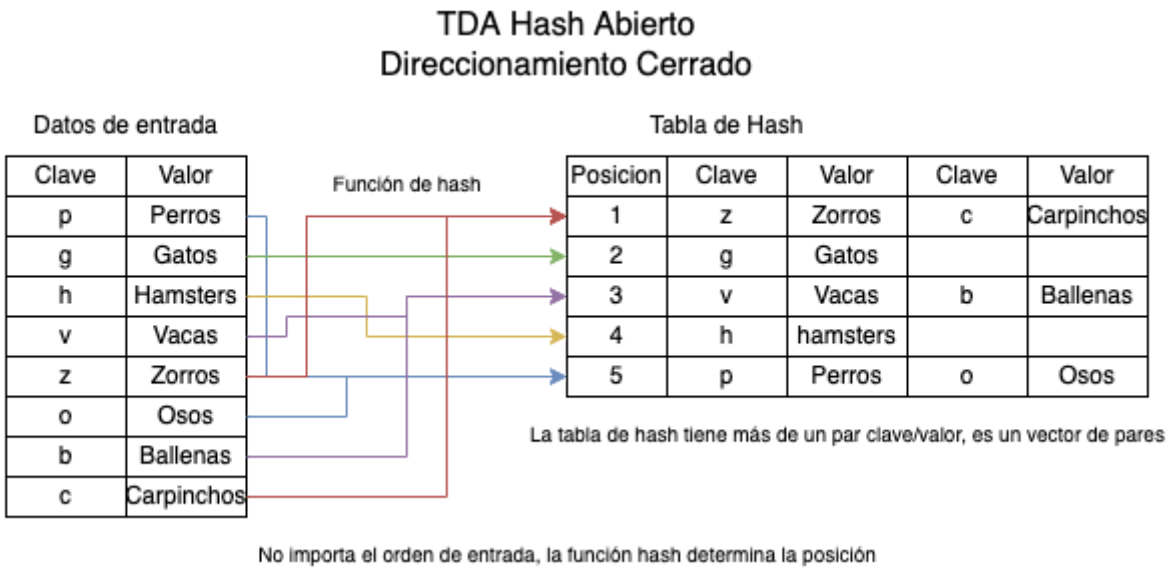


- Hash de Redireccionamiento Abierto o Hash Cerrado: En este caso de hash, el tamaño de la tabla coincide con el tamaño de la cantidad de elementos que puede albergar, puesto que cada una de las posiciones de la tabla admite un único par clave/valor. En este tipo de hash, podemos volver a observar el diagrama inicial, en donde cada elemento va a una posición específica que puede contener un único elemento:



Durante la insersión de elementos, es posible que ocurran casos en los que la posición en la tabla correspondiente a una clave coincida con la posición en la tabla de otra clave diferente, estos casos son llamados colisiones, según sea el tipo de hash, pueden occurir diversas situaciones:

- Colisiones en Hash de redireccionamiento cerrado o hash abierto: En este tipo de hash, debido a que una posición dentro de la hashtable permite almacenar más de un elemento, la nueva clave es guardada en conjunto a las anteriores que ocupaban la misma posición, sin perder ninguna (exceptuando el caso de inserción con misma clave, en donde se actualiza en lugar de insertar). En la anterior definición de hash abierto pudimos ver un caso en donde esto ocurría, guardándose el nuevo dato en conjunto al anterior:



- Colisiones en Hash de redireccionamiento abierto o hash cerrado: En este tipo de hash, debido a que una posición solo puede tener un dato, se busca una posición disponible en el hash para poder insertar el elemento solicitado. Puede ser realizado mediante diversas técnicas como Probing lineal (buscar el siguiente espacio disponible inmediato), Probing cuadrático (buscar una posición disponible según los intentos fallidos al cuadrado), o hash doble (aplicar una segunda función de hash para determinar otra posición).

Cómo últimos concepto importantes podemos mencionar al factor de carga y rehash:

- El factor de carga es calculado como numero de claves almacenadas / tamaño de la tabla de hash, nos sirve para determinar el momento adecuado para realizar un rehash. Es un dato

importante debido a que a mayor número de elementos con el mismo tamaño de tabla, peor es el performance (y en el caso de un hash de redireccionamiento abierto / hash cerrado, nos podemos quedar sin espacio).

- El rehash por su parte, es un procedimiento de agrandamiento de la tabla de hash, en donde cada uno de los elementos actualmente contenidos en el hash son movidos a una nueva tabla más grande, continuando la misma lógica de hasheado. Tiene el objetivo de escalar conforme nuestro hash crece en cantidad, para asegurar un performance constantemente óptimo. Es un procedimiento realizado usualmente sin límite de ocasiones, por lo que es posible empezar con un hash muy pequeño y terminar con uno muy grande dependiendo del uso. Sin embargo, es una operación muy costosa a nivel de procesamiento, por lo que resulta más importante conocer de antemano un aproximado del tamaño que requerimos para no realizar rehash en exceso.

3. Detalles de implementación

La implementación de este TDA, consiste en la implementación de un TDA Hash con redireccionamiento cerrado (hash abierto), y debido al funcionamiento teórico del mismo, se hace énfasis en la modularización, utilización y aprovechamiento de TDAs previamente creados, fortaleciendo el conocimiento y aprendizaje en caso de usos reales de los mismos.

Debido a lo antes mencionado, esta implementación hace uso del TDA Lista correspondiente al primer TDA evaluado de la cátedra. El cual es utilizado en esta implementación con la importante premisa de "utilizar y sacar provecho de las primitivas genéricas de lista para lograr una implementación limpia del hash concentrándonos únicamente en las funcionalidades de hash", con lo cual, ciertas restricciones propias fueron seguidas en la implementación, tales como: no interactuar directamente con los structs de nodo_t y lista_t en si mísmos dentro de hash.c, puesto que las primitivas de lista permiten operar sobre ellas sin conocer como está hecho. Lo cual, permite que nuestro hash no dependa de la implementación de lista, sino únicamente de sus primitivas públicas. Más sin embargo, esto no hubiera sido posible en todos los casos o requerimientos de nuestro hash, debido a que nuestra lista (implementación del primer TDA de la cátedra) no abarca todos los posibles casos de uso, con lo cual, antes de empezar a trabajar sobre la implementación de hash, fueron agregadas dos nuevas primitivas a lista.h/lista.c, de forma genérica y siguiendo la lógica de un TDA (Pensado similarmente al Open/Closed Principle), brindando la posibilidad además de ser usadas en casos no correspondientes a un hash, estas son los siguientes puntos:

1. Función: lista_t* lista_insertar_o_actualizar(lista_t* lista, void* elemento, bool (*comparador)(void* elemento_actual, void* elemento_nuevo), void** elemento_anterior)

   Esta función es una extensión abstracta del tda lista, la cual tiene como objetivo recorrer la lista en búsqueda de un elemento, para la cual se utilizará la función comparadora recibida por parámetro junto con el elemento actual y elemento a insertar recibido también por parámetro para determinar si será actualizado este elemento actual será reemplazado por el nuevo. De ser reemplazado, también se devolverá el elemento anterior en la referencia void**

elemento_anterior. En caso de no encontrarse un elemento igual/equivalente según la funcion comparadora, se insertará el nuevo elemento al final de la lista.

La implementación de esta nueva primitiva en lista, permite optimizar el recorrido de la lista para la actualización de un elemento, que en caso contrario de utilizar unicamente las primitivas anteriores, hubiera sido requerido realizar dos recorridos o un reccorido muy engorroso haciendo uso del iterador externo haciendo un código más extenso y quizás dificil de entender.

2. Función: void* lista_quitar_especifico(lista_t* lista, bool (*comparador)(void* elemento, void* aux), void* aux)

   Esta función al igual que la anterior, es una extensión abstracta del tda lista, la cual permite el reccorido de la lista con el objetivo de quitar un elemento específico sin conocer su posición, haciendo uso de la función comparadora recibida, con el elemento actual en la iteración y el aux recibido. La cual devolverá el elemento quitado (de encontrarse), o NULL en caso de no existir.

   De igual manera a la anterior, fue implementada con la intención de hacer eficiente el recorrido y eliminación del dato específico, puesto que en caso contrario haciendo uso de las primitivas únicamente, era requerida una iteración hasta encontrar el dato y posteriormente la eliminación de la posición específica, realizando en total dos recorridos.

Adicionalmente, se describen algunos detalles más técnicos de la implementación, más allá de los requerimientos establecidos por la cátedra:

3. Función: hash_t* hash_crear(hash_destruir_dato_t destruir_elemento, size_t capacidad_inicial)

   Esta función es encargada de crear una instancia de nuestro tda hash para hacer uso de ella, además los requerimientos de implementación como el mínimo de capacidad_inicial para la creación de la hashtable establecido en 3, y el guardado de destruir_elemento para su posterior uso en inserciones (actualizaciones), eliminaciones y destrucciones, se tomó la decisión de implementación de no ser la responsable de la creación de cada una de las listas contenidas en la hash_table. Por el contrario, se pensó en optimizar dicho proceso para ser ejecutado únicamente de requerirse en sus respectivas inserciones/iteraciones/eliminaciones/obtenciones, cómo ahorro de procesamiento en casos de uso de tablas grandes con un posible poco uso.

4. Función: int hash_insertar(hash_t* hash, const char* clave, void* elemento)

   Esta función es encargada de insertar (o actualizar datos con la misma clave) elementos en el hash, hace uso de la nueva primitiva abstracta de lista (lista_insertar_o_actualizar) junto con un comparador de datos para un óptimo recorrido de la lista. En caso de actualizar un dato equivalente (con misma clave) previamente almacenado, hace uso de la función destructora (en caso de existir) para destruir el elemento contenido, y establece el nuevo dato.

Al finalizar cada inserción, esta función también se encarga de comprobar el factor de carga del hash excede el mínimo requerido para la realización de un rehash, y de ser el caso, ejecuta la función de rehash descrita más adelante.

5. Función: int hash_quitar(hash_t* hash, const char* clave)

   Esta función es encarga de la eliminación de un elemento contenido en el hash según la clave recibida, hace uso de la nueva primitiva abstracta de lista (lista_quitar_especifico) junto con un comparador del dato contenido con la clave solicitada para eliminación, con la intención de optimizar el recorrido de la lista. En caso de encontrar y quitar el elemento solicitado de la lista, aplica la función destructora del hash al elemento.

6. Función: void* hash_obtener(hash_t* hash, const char* clave)

   Esta función devuelve en caso de encontrarse, el elemento contenido en el hash con la clave solicitada. Debido a que en este recorrido en busca del elemento no es necesaria realizar ninguna operación adicional más que devolver el elemento, es utilizado el iterador externo del tda lista al no suponer una desventaja o problema en cuando a performance se refiere. Sin embargo, también hubiera sido factible implementar una tercera primitiva nueva que cumpliera dicha necesidad.

7. Función: size_t hash_con_cada_clave(hash_t* hash, bool (*funcion)(hash_t* hash, const char* clave, void* aux), void* aux)

   Esta función realiza un recorrido del hash aplicando la función recibida con el hash, las clave y el aux recibido en cada iteración. Continuando en caso de devolver false, y deteniendose en caso de devolver true. Devolviendo la cantidad total de elementos recorridos. La implementación de la misma hace uso de un transformador de recorridos (descrito en el siguiente punto), con el objetivo de poder reutilizar la primitiva de iteración interna de la lista (lista_con_cada_elemento) y no tener la necesidad de reescribir dicha funcionalidad.

8. Función y struct aux: bool transformador_de_recorrido(void* elemento, void* aux) y struct transformador_de_recorrido_aux

   Esta función tiene la estructura requerida por el iterador interno del tda lista (lista_con_cada_elemento), con la intención de aprovechar dicho iterador en la primitiva de hash bool transformador_de_recorrido(void* elemento, void* aux), haciendo una transformación entre ambas funciones gracias al struct de auxiliar que debe recibir conteniendo los datos del hash, función solicitada en hash_con_cada_clave y su respectivo auxiliar.

9. Función interna: int rehashear_tabla(hash_t* hash)

   Esta función se encarga de crear una nueva tabla de hash con el doble de capacidad a la anterior, y mueve todos los elementos contenidos en la tabla actual a la nueva bajo los mismos criterios de hasheado, para posteriormente eliminar la tabla anterior y reemplazarla por la nueva de mayor tamaño. Internamente, hace uso del iterador interno del tda lista

(lista_con_cada_elemento) con el objetivo de no reescribir las iteraciones de la lista, haciendo uso de una función auxiliar para la inserción de cada uno de los mismos datos contenidos hacia la nueva lista.

El resto de funciones continuan la misma lógica, aplicando los requisitos solicitados por la cátedra.

4. Pruebas y ejemplo

Como se mencionaba en la introducción, el enfoque de desarrollo utilizado en la implementación con respecto a las pruebas, fue TDD o Test Development Driven (Desarrollo Guiado por Pruebas), las cuales forman parte de la entrega y no son provistas por la cátedra. Las mismas se encontran en ./pruebas.c y pueden ser ejecutadas haciendo uso del comando `make` con el archivo makefile, o en su defecto con `gcc src/*.c pruebas.c -o pruebas 2>&1` . Dichas pruebas abarcan en conjunto a cada uno de los aspectos o funcionalidades esperadas del tda hash.

Así mismo, y si provisto por la cátedra, se cuenta con una pequeño programa (ejemplo.c) que hace uso del hash implementado, a forma de ejemplo de funcionalidad, el cual además puede ser tomado en cuenta como guía de utilización. Si bien, el mismo tiene como objetivo comprobar el correcto funcionamiento del TDA, es una implementación pequeña que no contempla todos los casos de uso, por lo que es complementaria a las pruebas de la entrega, y no un reemplazo a las mismas.

5. Importación, utilización y compilación

Para utilizar este tda, necesitamos importar la librería "src/hash.h" en nuestro programa, agregando los archivos hash_struct.h, hash.h, hash.c, lista.c, lista.h a nuestro script de compilación, de la siguiente manera:

Script de compilación de ejemplo previo a importación de hash y dependencias: `gcc ./miprograma.c -o miprograma` Script de compilación de ejemplo posterior a importación de hash y dependencias: `gcc ./miprograma.c ./src/*.c -o miprograma`

Con esta modificación a nuestro script de gcc, permitiremos la importación en la compilación de la compilación de las dependencias.

1- Creación: Una vez realizado, podemos empezar a crear un hash para ser utilizado en nuestro programa, mediante:

```
hash_t* hash = hash_crear(hash_destruir_dato_t destruir_elemento, size_t capacidad_inicial);
```

Donde destruir_elemento es una función de tipo bool (*destructor)(void) opcional, que permite destruir los elementos insertados en el hash al momento de ser eliminados o actualizados de la tabla, y capacidad_inicial es el tamaño de la tabla de hash con la que se iniciará el hash (Con un mínimo establecido de 3).

2- Inserción: Una vez contamos con nuestro hash creado, podemos utilizarlo para empezar a insertar datos mediante la primitiva de hash_insertar:

```
int hash_insertar(hash_t* hash, const char* clave, void* elemento);
```

Enviando el hash previamente creado, la clave y el elemento correspondiente. Recibiremos como retorno 0 en caso de éxito, o -1 en caso de error.

3- Obtención/Búsqueda: Podemos buscar los elementos (en el caso de hash_obtener) o simplemente saber si existen (con hash_contiene) dentro de nuestro hash, de la siguiente manera:

```
void* hash_obtener(hash_t* hash, const char* clave); y bool hash_contiene(hash_t* hash, const char* clave);
```

Enviando el hash creado y la clave a buscar, y obteniendo el elemento en caso de encontrarse en hash_obtener o NULL en caso de no existir. En el caso de hash_contiene, true si existe y false en caso contrario.

4- Obtener cantidad: Podemos saber cuantos elementos están actualmente almacenados en nuestro hash mediante:

```
size_t hash_cantidad(hash_t* hash);
```

Simplemente enviando nuestro hash, y obteniendo como retorno la cantidad, o 0 en caso de error.

5- Quitar elemento: Podemos eliminar un elemento en específico (conociendo su clave) de nuestro hash mediante:

```
int hash_quitar(hash_t* hash, const char* clave);
```

Enviando nuestro hash y la clave a quitar, devolviendo 0 en caso de éxito, o -1 en caso de error o no encontrarse la clave.

6- Iterar: Podemos recorrer nuestro hash clave a clave mediante el interador interno:

```
size_t hash_con_cada_clave(hash_t* hash, bool (*funcion)(hash_t* hash, const char* clave, void* aux), void* aux);
```

Enviando nuestro hash, una función que reciba el hash, clave y un auxiliar (que podría ser opcional) la cual será ejecutada en cada una de las claves iteradas y devolverá false en caso de quere continuar iterando, o true para deter la iteración con dicho elemento.

7- Destruir: Al terminar de usar nuestro hash podemos destruirlo y liberar la memoria:

```
void hash_destruir(hash_t* hash);
```

Únicamente enviando nuestro hash.

Cómo anteriormente, mencionabamos, podemos tomar como ejemplo la implementación provista por la cátedra encontrada en ./ejemplo.c.