Marcos Bianchi - 108921

Tabla de Hash

Una tabla de hash es una estructura de datos que relaciona claves con valores. A diferencia de otros TDAs la tabla tiene una ventaja a la hora de buscar elementos en ella, la complejidad de una búsqueda tiende a ser de O(1). Esto es porque la tabla está acompañada de una función de hashing, esta función determina dónde se va a guardar un elemento en la tabla y en resumen sabemos donde “viven” los elementos y solo tenemos que ir a visitar esa dirección del vector para encontrarlos.

Tabla de Hash Cerrada

Una tabla de hash cerrada es una tabla donde los elementos se guardan dentro de la tabla, esto quiere decir que cada elemento ocupa una posición de la tabla y no se comparten. El problema que tiene esta implementación es que se introducen las colisiones.

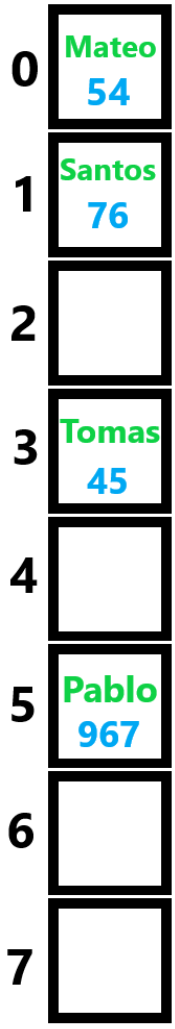
Las colisiones se producen cuando 2 claves distintas se hashean al mismo número, por ejemplo: Si mi función de hash devuelve el valor del primer carácter de la clave enviada y esta recibiera “Manuel” y “Mateo” en ambos casos devuelve 77. Luego de hacer 77 % capacidad_del_hash (en este caso es 10), nos quedamos en que queremos insertar ambos elementos en la posición 7.

En una tabla cerrada tenemos que direccionar el segundo elemento a otra posición de la tabla, existen varios métodos pero el más simple es la prueba lineal que consiste en seguir de largo y buscar más adelante en la tabla hasta encontrarle un espacio vacío a nuestro elemento.

Se llama direccionamiento abierto porque el programa debe cambiar la dirección que le devuelve la función de hash para acomodar al nuevo elemento.

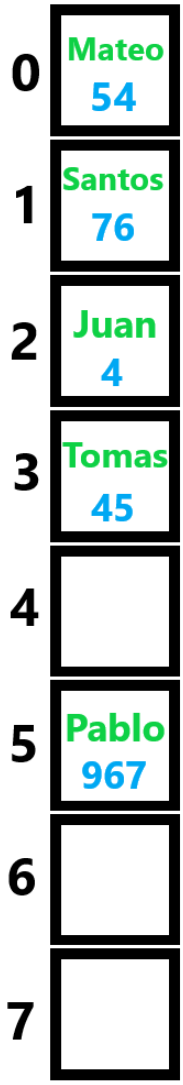
Insertar:

Para insertar un elemento en una tabla primero hay que hashear su clave, para el ejemplo voy a usar una tabla con claves de tipo string y valores de tipo int.



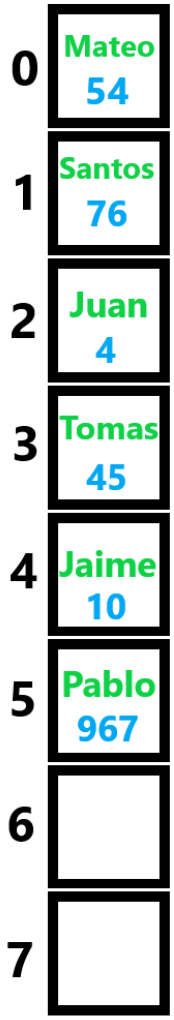
(Aclaro que no se si los demás elementos estan donde deberian estar)

Supongamos que la clave es “Juan” y el valor es 4, vamos a hashear “Juan” y ver donde queda. Si usamos la función de hash que explique antes, la función va a devolver 74, para saber donde hay que insertar tenemos que hacer “hash(“Juan”) % hash->capacidad” (para el ejemplo uso una tabla de capacidad == 8). Entonces quedamos en que hay que insertar en la posición 2.



Entonces la tabla queda así después de insertar <clave / valor> <Juan / 4>. En caso de que ya exista un elemento con clave == “Juan” se va a actualizar el valor anterior con el nuevo valor. En caso de haber una colisión con otro elemento va a pasar lo siguiente.

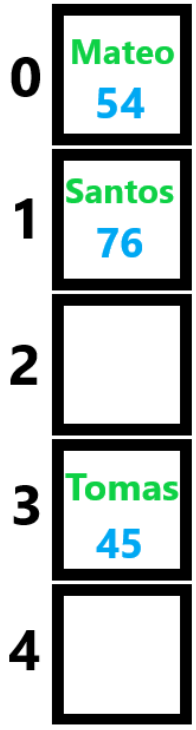
En caso de una colisión, por ejemplo queremos insertar un elemento con clave == “Jaime”, vamos a tener que encontrarle una nueva posición en la tabla. Después de hashear la clave y utilizar la fórmula del módulo determinamos que hay que guardarlo en la posición 2 pero esta ya está ocupada. Vamos a aplicar el probing lineal y vamos a visitar la posición 3 en busca de un espacio vacío, no lo encontramos así que visitamos la posición 4 y como está vacía insertamos <clave / valor> <Jaime / 10> en la posición 4.



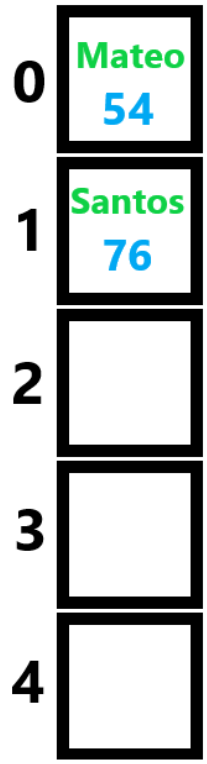
A medida que se va llenando la tabla se va perdiendo eficiencia, por ejemplo si quisiéramos insertar “Juan Pablo”, habría que recorrer casi toda la lista hasta llegar a la posición 6 e insertar el elemento ahí. Entonces habría que rehashear la tabla. Consiste en agrandar la tabla e ir elemento por elemento hasheando sus claves para encontrarles un nuevo lugar, en una tabla cerrada este procedimiento es distinto que en una tabla abierta porque los elementos se guardan dentro pero si la tabla fuera de punteros a par_t se podrían traspasar esos pares a la nueva tabla. Esto va a cambiar permanentemente las posiciones de los elementos en ella.

Quitar:

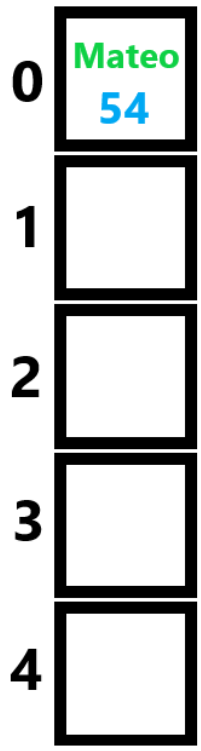
Para quitar un elemento de la tabla vamos a tener que recibir una clave del elemento que queremos borrar, luego hashear la clave para saber donde empezar a buscar dentro de la tabla, quitar el elemento y devolverlo.



Si el usuario quisiera borrar clave == “Tomas”, se hashea la clave luego de aplicar la función de hash determinamos que el elemento con clave “Tomas” vive en la posición 3. Accedemos a la posición 3 y reemplazamos los valores por los valores predeterminados que tengamos para representar que quedó vacío.



Ahora supongamos que queremos borrar a Santos, esa clave debería estar en la posición 0 pero antes no teníamos espacio para almacenarla ahí entonces la movimos una posición más abajo. Entonces vamos a visitar la posición 0 y no vamos a encontrar el elemento que estábamos buscando, hay que seguir buscando en las posiciones del vector. Buscamos una posición más abajo y encontramos el elemento que buscábamos, lo quitamos y lo devolvemos su valor al usuario.



Si quisiéramos borrar un elemento con clave inexistente no podemos conformarnos con ir a su posición hasheada y no encontrarlo porque quizá esas posiciones estaban ocupadas cuando insertamos ese elemento o no, la computadora eso no lo sabe. Vamos a tener que tener en cuenta que posiciones borramos o reorganizar un poco el hash luego de borrar elementos para de esa manera buscar hasta encontrar un espacio vacío que no haya sido borrado y saber ahí si existe o no el elemento.

Búsqueda:

La búsqueda es simplemente hashear la clave que recibimos del usuario y en caso de no estar ahí aplicar probing lineal hasta encontrar el elemento y devolverlo o un espacio vacío que no haya sido borrado que determina que el elemento no existe.

Tabla de Hash Abierta

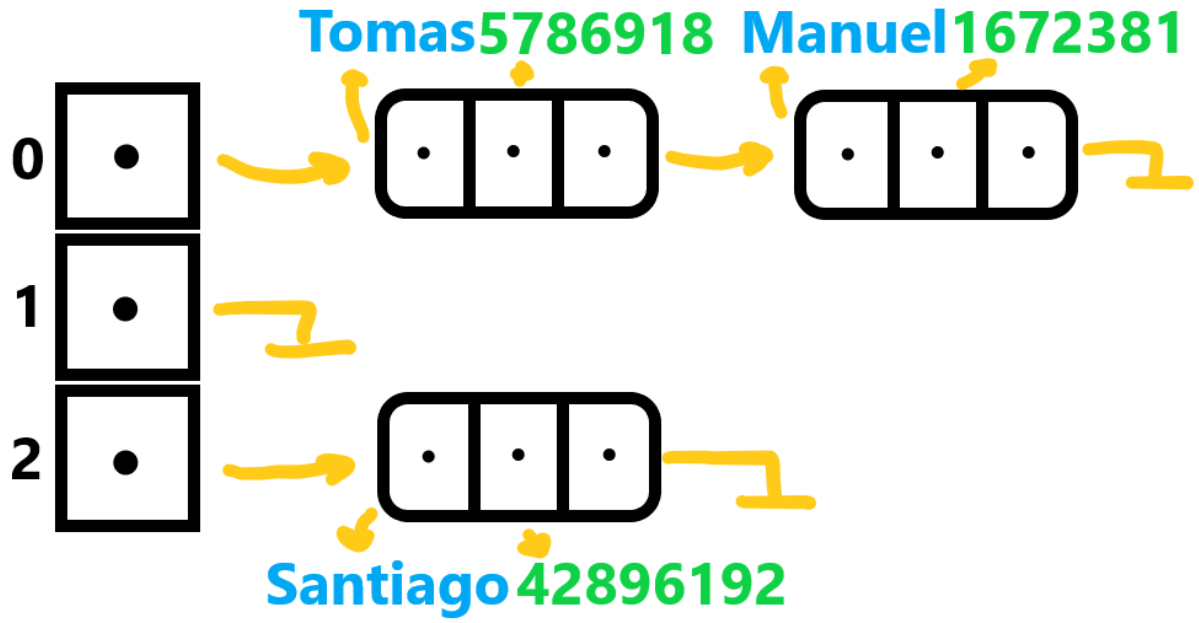
Una tabla de hash abierta es una tabla donde los elementos se guardan fuera de la tabla, esto quiere decir que se va a utilizar otro tipo de estructura para almacenar los

elementos y que varios elementos pueden compartir posiciones del vector de la tabla. Esto quiere decir que las colisiones no son un problema cuando se implementa este tipo de tabla.

En una tabla abierta tenemos que acomodar los elementos en otra estructura que puede ser un árbol, una lista o quizá una tabla cerrada, el direccionamiento es cerrado, esto quiere decir que no se modifica la posición que me devuelve la función de hash y que si el usuario quiere buscar un elemento x, siempre va a estar en la posición que me devuelva la función de hash con respecto a su clave.

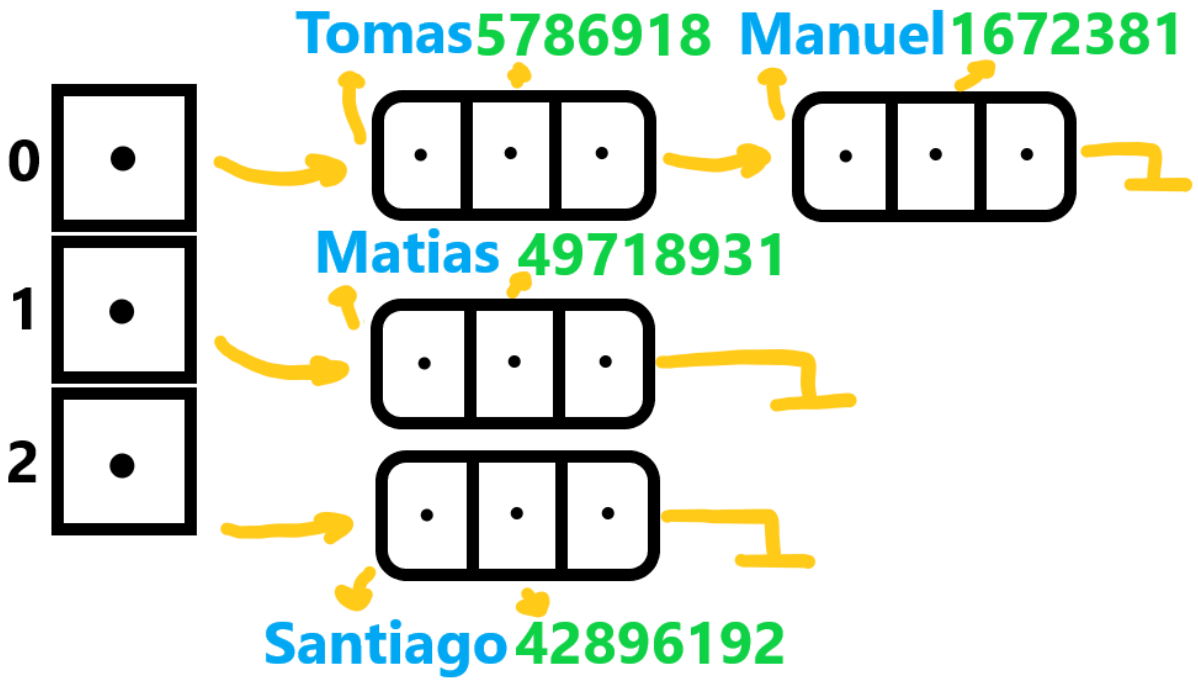
Insertar:

Para insertar en una tabla abierta primero vamos a hashear la clave y ver en qué posición de la tabla habría que guardar el elemento. Luego se recorre la estructura que para este ejemplo será una lista y de encontrar una coincidencia en cuanto a la clave se reemplaza su valor con el nuevo y de recorrer toda la lista sin hallarla solo se inserta el nodo.



En esta tabla hay 3 elementos y vamos a insertar un cuarto con clave “Matias”, luego de aplicar la fórmula utilizando el módulo de lo que devuelva la función de hash nos queda que hay que insertar el elemento en la posición 1. Así que el programa va a reservar memoria y va a dirigirse a la posición 1 y primero chequear que no haya ningún elemento insertado, en este caso no lo hay asi que solo apunta el puntero al nuevo elemento.

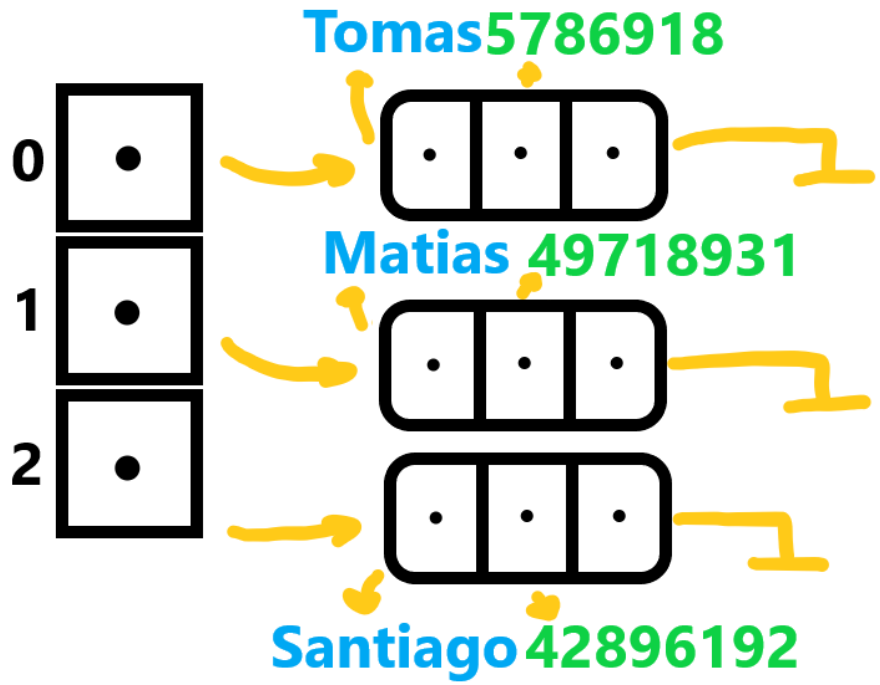
Una vez que la tabla se va llenando pasa lo mismo que con la tabla cerrada, se pierde eficiencia en todos sus procesos. Entonces se debe rehashear la tabla, se deben reacomodar todos sus elementos en una tabla más grande.



En caso de querer actualizar el valor del elemento con clave “Matias” se introduce clave:valor Matias y su nuevo valor, el programa va a cambiar el valor anterior por el nuevo. En caso de querer agregar un elemento en una posición donde ya la hay, se inserta en la lista correspondiente.

Quitar:

Para quitar en una tabla abierta se debe recibir una clave que se utilizara para primero hashearla y saber por dónde empezar a buscar y segundo para comparar esa clave a la clave por la que estará pasando nuestro iterador. Así que si quisiéramos quitar de la lista a Manuel deberíamos primero hashearla y buscar en esa lista (en este caso sabemos que Manuel está en la posición 0). Luego iterar la lista hasta encontrarlo y eliminarlo, de no encontrar el elemento con clave idéntica no se hace nada.



Búsqueda:

Para buscar un elemento en una tabla abierta se debe recibir una clave, hashearla e iterar una lista hasta encontrar una igualdad con la clave recibida. Si la clave no existe en esa lista significa que el elemento no existe en toda la tabla ya que una tabla abierta tiene direccionamiento cerrado y eso significa que no podría estar en ningún otro lado.

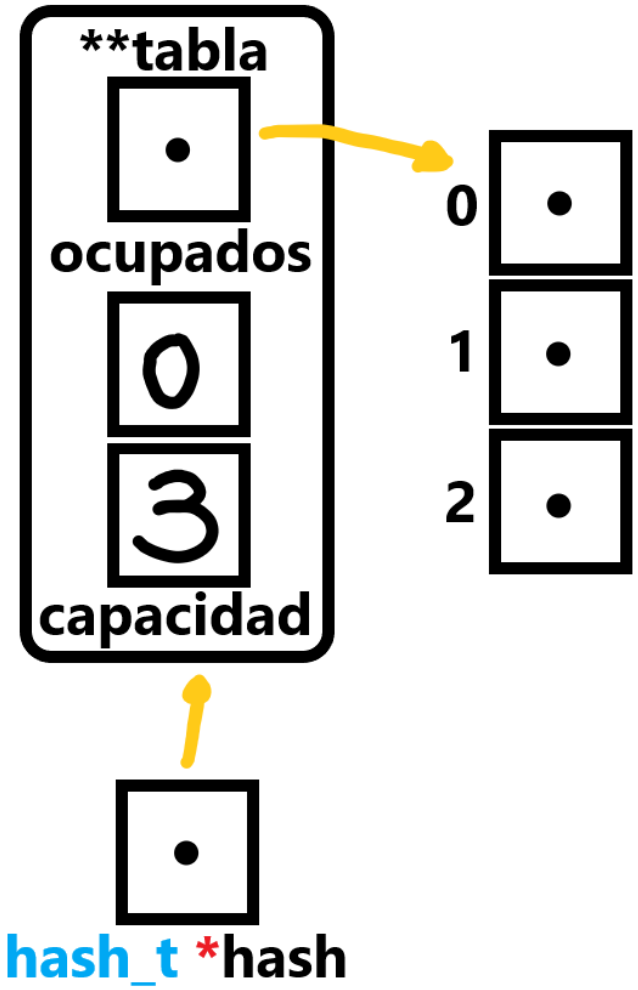
Complejidad:

En ambas tablas la complejidad de todos estos procesos tiende a ser o se apunta a que sean de O(1). Esto tiende a ser así en ciertos casos que serían que la función de hash sea buena y distribuya bien los elementos, que se le haga rehash a la tabla con un factor de carga razonable. Si una tabla abierta con 3 posiciones guarda 300 elementos el diccionario pierde mucha eficiencia, por eso se hace rehash y por eso se utiliza una buena función de hash. Aunque hiciéramos rehash con un factor de carga super bajo si nuestra función de hash es mala (en el peor de los casos) devuelve el mismo número siempre y todos los elementos insertados apuntan a la misma posición de la tabla se generan colisiones constantemente.

Implementación

```c
hash_t *hash_crear(size_t capacidad);
```

hash_crear reserva espacio en memoria del tamaño de un hash_t y en el inicializa su capacidad y su cantidad de ocupados en 0. Reserva memoria para una tabla del tamaño de la capacidad pasada por parámetro * sizeof(puntero), en caso de que algún malloc devuelva NULL se libera la memoria reservada previamente y se devuelve NULL.



La tabla se inicializa con calloc para que queden todos apuntando a NULL.

```c
size_t hashify(const char *clave);
```

Mi función de hash recibe una clave y primero inicializa una variable int llamada hash que almacena el número 5381, va a iterar toda la clave y a guardar el valor de un caracter en otra variable llamada c. Por último a nuestra variable hash la multiplica por 2^5 (utiliza un operador de bits), le suma el valor anterior de hash y le suma el valor de c. Se repite el proceso hasta que se itere toda la clave y se devuelve el valor de hash casteado a size_t.

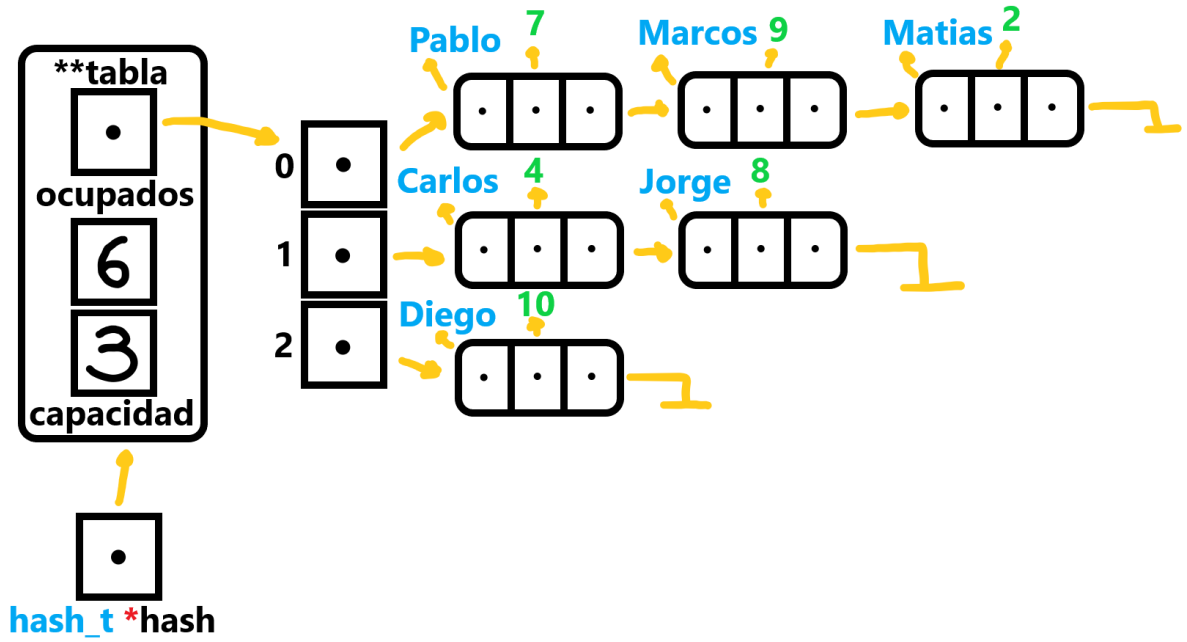
Esta función la saque de internet obviamente pero la que use para todo el desarrollo del TDA fue una que devuelve la suma de todos los caracteres. Cuando no me pasó las pruebas tuve que cambiarla porque generaba muchas colisiones.

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

Mi hash insertar primero calcula el factor de carga y de exceder o ser igual a COTA_MIN (una variable constante que tengo en el .h) rehashea la tabla. Luego calcula utilizando la función de hash en que posición de la tabla hay que insertar el elemento y si esa posición se encuentra vacía, se invoca a nodo_crear() y apunta el puntero de la tabla al nodo. En caso de que no esté vacía se invoca a la función nodo_a_lista.

nodo_crear() reserva espacio en la memoria para un nodo_t guarda una copia de la clave pasada por parámetro, guarda el elemento e incrementa la cantidad de ocupados del hash en 1, luego devuelve ese puntero.

nodo_a_lista() recibe un puntero al hash, un puntero al primer nodo de la lista actual, un puntero a la clave, un puntero al elemento y un puntero al puntero anterior. Básicamente itera con un puntero la lista actual hasta que se termine o hasta encontrar una clave idéntica. En caso de hallarla se reemplaza su valor y en caso de llegar al final se crea un nodo con nodo_crear() y se conecta al final de la lista.



Utilizando mi implementación esta tabla ya se hubiera rehasheado pero voy a explicarla con esta tabla.

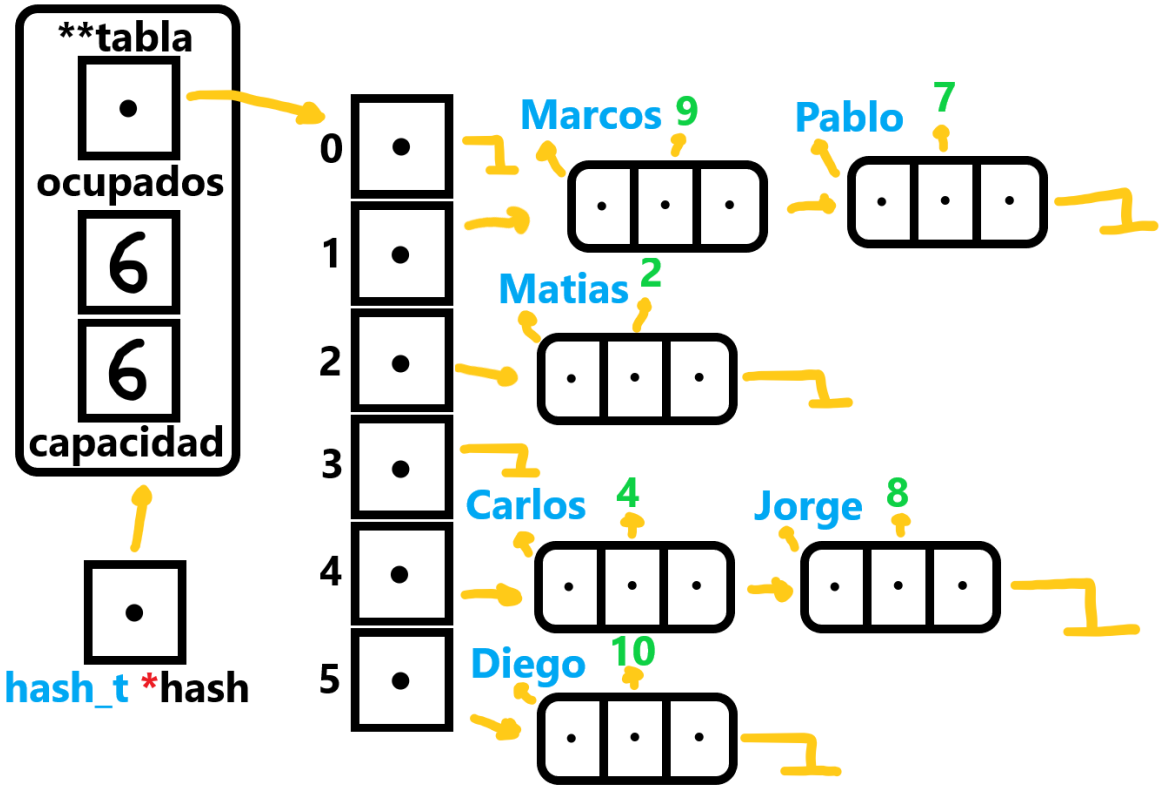
```
void rehash(hash_t *hash);
```

Mi función de rehash recibe un puntero al hash, crea un puntero a punteros llamado tabla_aux que va a servir para no perder la referencia a nuestra tabla anterior, duplica la capacidad del hash y reserva espacio en memoria para una tabla de esta capacidad. Luego itera las listas de tabla_aux con la ayuda de un iterador, hashea las claves e invoca a traspasar _nodo() enviando un puntero a la tabla nueva, el puntero al nodo que queremos

traspasar y la posición en la tabla donde hay que reinsertarlo. Por último se libera la memoria reservada por la tabla vieja.

traspasar_nodo() recibe estos datos y va a guardarse la posición siguiente del nodo, va a apuntar su puntero siguiente a donde apunte la posición x de la tabla y va a apuntar este mismo puntero al nodo. Por último devuelve el puntero sig para que el iterador en rehash() pueda avanzar una posición y evaluar el siguiente puntero. Esta función agrega los nodos al principio de la tabla, va a alterar totalmente el orden anterior (Una tabla de hash es una estructura desordenada así que no hay problema).

Luego de aplicar rehash sobre la tabla de arriba se debería ver algo parecido a esto:



Esta implementación de rehash no reinserta nodos sino que los redirecciona. El rehash no garantiza que no vayan a haber colisiones.

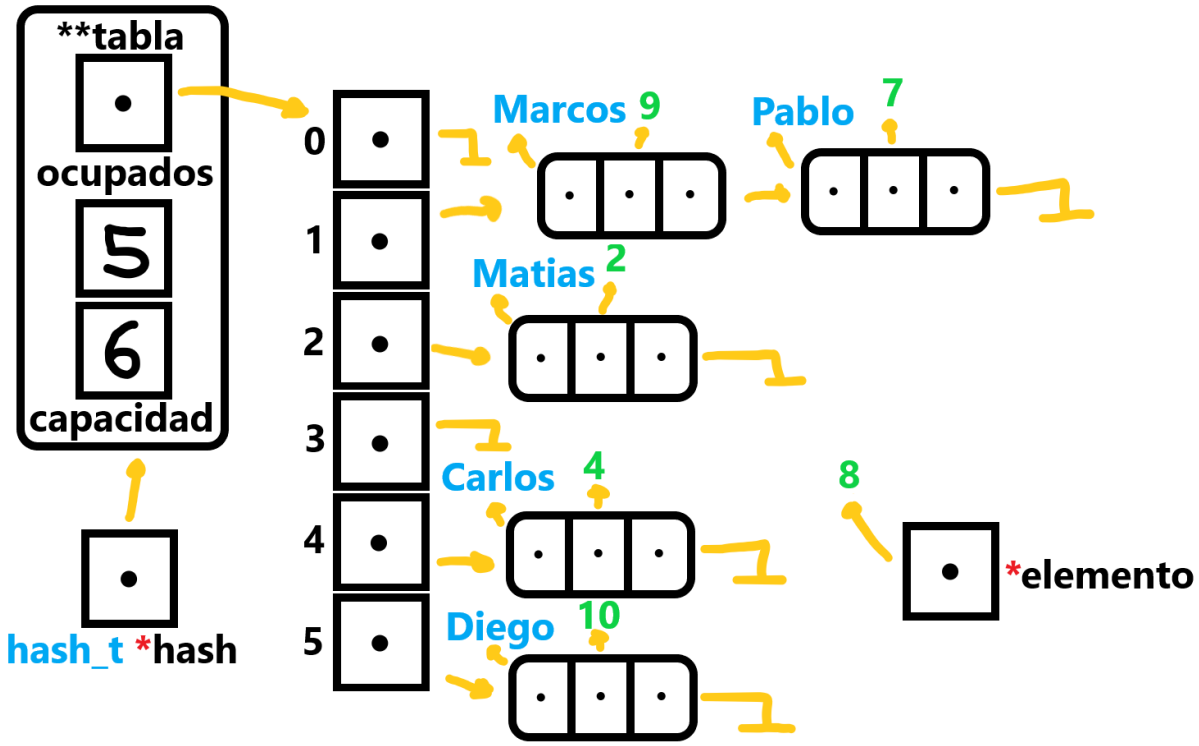
```c
void *hash_quitar(hash_t *hash, const char *clave);
```

hash_quitar() primero calcula la posición de la tabla en la que se encuentra la clave por eliminar y comienza a iterar la lista junto a un puntero aux que apunta al nodo anterior al iterador. Si se encuentra la clave se invoca a nodo_quitar(), de no ser así se devuelve NULL.

nodo_quitar() recibe un puntero al hash, un puntero a un nodo, un puntero al nodo anterior y una posición. Primero guarda el valor inicial del nodo con un puntero llamado elemento, luego apunta al nodo anterior al siguiente (en caso de que el anterior sea NULL,

apunta al puntero inicial de la tabla al nodo siguiente), resta por uno la cantidad de ocupados en el hash y devuelve un puntero al valor del nodo.

En el ejemplo de abajo se eliminó a Jorge, se resta en 1 la cantidad de ocupados y se devuelve el elemento.




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

hash_obtener() calcula la posición de la tabla hasheando la clave e itera la lista utilizando un iterador interno. En caso de encontrar al elemento lo devuelve y en caso de no encontrarlo devuelve NULL.

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

hash_contiene() invoca a hash_obtener y devuelve su resultado, en caso de ser un puntero válido es true y en caso de ser NULL devuelve true.

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

hash_cantidad() devuelve la capacidad de la tabla.

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

Invoca a hash_destruir_todo() pasando destructor NULL.

```
void hash_destruir_todo(hash_t *hash, void (*destructor)(void *));
```

hash_destruir_todo() itera toda la tabla liberando la memoria reservada por ella y sus elementos, por último libera la memoria de la estructura hash.

Comienza por la primera posición de la tabla y utiliza un iterador interno para invocar a la función nodo_destruir() con cada nodo.

nodo_destruir() se guarda la posición siguiente al nodo pasado por parámetro, en caso de tener un puntero a la función de destrucción válido destruye el valor dentro del nodo, libera la memoria reservada por la clave, libera la memoria del nodo y devuelve un puntero al siguiente.

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

hash_con_cada_clave() itera toda la tabla aplicando la función pasada por parametro a todos los nodos visitados, aparte cuenta por cuantos nodos se itero y devuelve ese valor. En caso de que en algún momento de la ejecución la función devuelve false se devuelve el valor de iteradas en ese momento cortando el proceso.