Trabajo Práctico — HASH

[7541/9515] Algoritmos y Programación II
Primer cuatrimestre de 2022

Alumno:	Dall Acqua, Denise
Número de padrón:	108645
Email:	Denisedallacqua10@gmail.com

Índice

Enunciado

Se pide implementar una Tabla de Hash abierta (direccionamiento cerrado) en C. Para ello se brindan las firmas de las funciones públicas a implementar y se deja a criterio del alumno la creación de las estructuras y funciones privadas del TDA para el correcto funcionamiento de la Tabla de Hash cumpliendo con las buenas prácticas de programación. La función de Hash a utilizar será interna de la implementación y también debe ser decidida por el alumno. Para esta implementación las claves admitidas por la tabla serán solamente strings. Adicionalmente se pide la creación de un iterador interno que sea capaz de recorrer las claves almacenadas en la tabla.
El TDA entregado deberá compilar y pasar las pruebas dispuestas por la cátedra sin errores, adicionalmente estas pruebas deberán ser ejecutadas sin perdida de memoria.
A modo de ejemplo, se brindara al alumno un archivo “ejemplo.c”. Este archivo no es mas que un programa mínimo de ejemplo de utilización del TDA a implementar y es provisto sólo a fines ilustrativos como una ayuda extra para entender el funcionamiento del mismo. No es necesario modificar ni entregar el archivo de ejemplo, pero si la implementación es correcta, debería correr con valgrind sin errores de memoria.
Para la resolución de este trabajo se recomienda utilizar una metodología orientada a pruebas. A tal fin, se incluye un archivo pruebas.c que debe ser completado con las pruebas pertinentes de cada una de las diferentes primitivas del TDA. El archivo de pruebas forma parte de la entrega y por lo tanto de la nota final. Aún más importante, las pruebas van a resultar fundamentales para lograr no solamente una implementación correcta, sino también una experiencia de desarrollo menos turbulenta.

También, se deberá redactar un informe que elabore los siguientes puntos:

- Explique teóricamente qué es una tabla de hash abierta y cerrada e ilustre las operaciones de inserción, eliminación y búsqueda para los dos casos. Ayúdese con diagramas para explicar.
- Explique su implementación y decisiones de diseño (por ejemplo, la estructura utilizada, el criterio metodología de rehash, función de hashing, etc).

1. Introducción

Para este informe desarrollaré todo lo visto he implementado sobre el TDA hash, diferenciando un hash abierto de un cerrado, en relación a su estructura y como se desarrollan las primitivas en cada una de ellas; y sobre la implementación que he llevado a cabo.

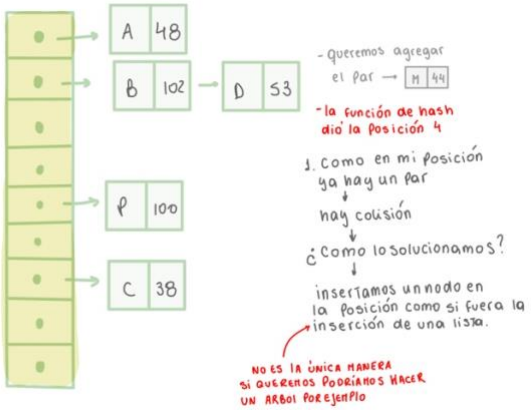
2. Teoría

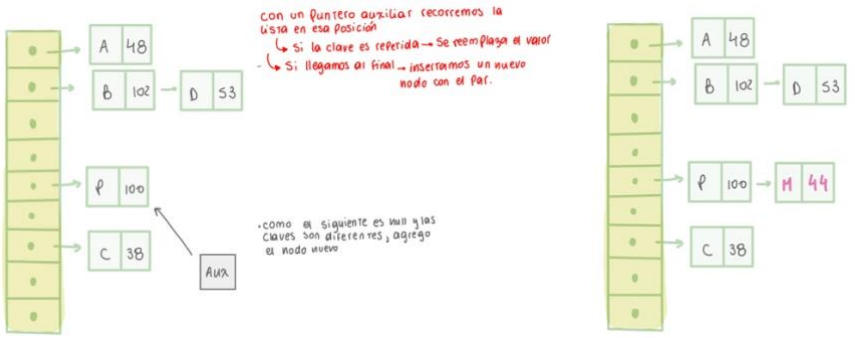
¿QUE ES UN DICCIONARIO?

Un diccionario es una colección de pares "clave, valor" en el cual puedo acceder a su valor por medio de esa clave. Podríamos decir que la clave funciona como un índice en su búsqueda. Es utilizado gracias a la buena performance a la hora de acceder al dato y ya que no hay "duplicación de entradas", ósea la clave en un diccionario es "única". La estructura que almacena estos pares se la llama TABLA DE HASH y cada elemento se va insertan en la posición que le corresponde gracias a una función que dada la clave elegida, realiza algunas operaciones y nos da un numero asociado que sería la posición donde estaría este par en nuestra tabla.

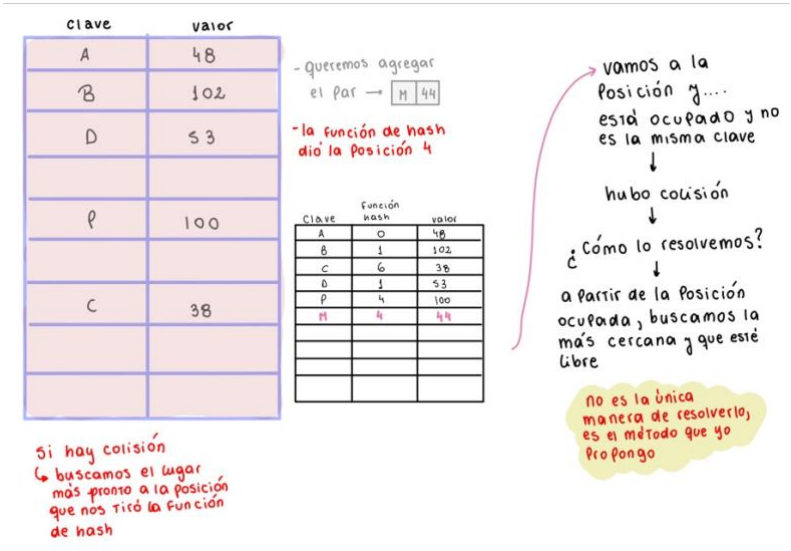
DIFERENCIAS ENTRE HASH CERRADO Y HASH ABIERTO

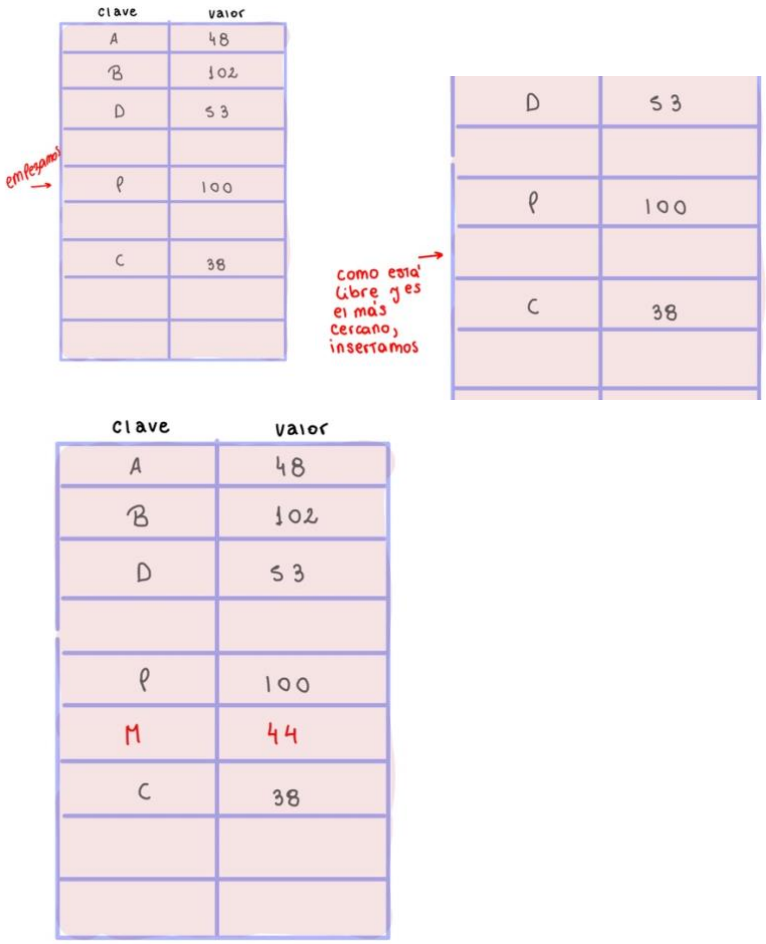
Existen dos tipos de hash: Cerrado y abierto. El **hash abierto** tiene la característica de que su tabla de hash tiene en cada posición un TDA que elijamos para almacenar los pares, por ejemplo una lista, donde vamos a ir guardando cada uno en la correspondiente lista según la posición de nuestra tabla de hash, que podríamos decir que es un vector de punteros a listas. Si hay una colisión, ósea que alguna clave nos dé una posición donde ya hay un elemento, simplemente insertamos el nodo enlazándolo con el nodo que ya existía; por eso se lo denomina "direccionamiento cerrado" ya que la posición del par no va a cambiar aunque ya haya uno previo ocupando el lugar. La complejidad de este tipo de hash depende del TDA que lleguemos a utilizar para guardar los valores, en el caso de la lista seria O(n).



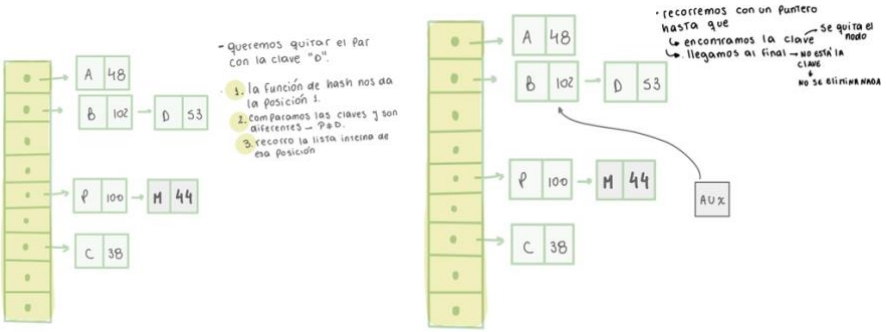


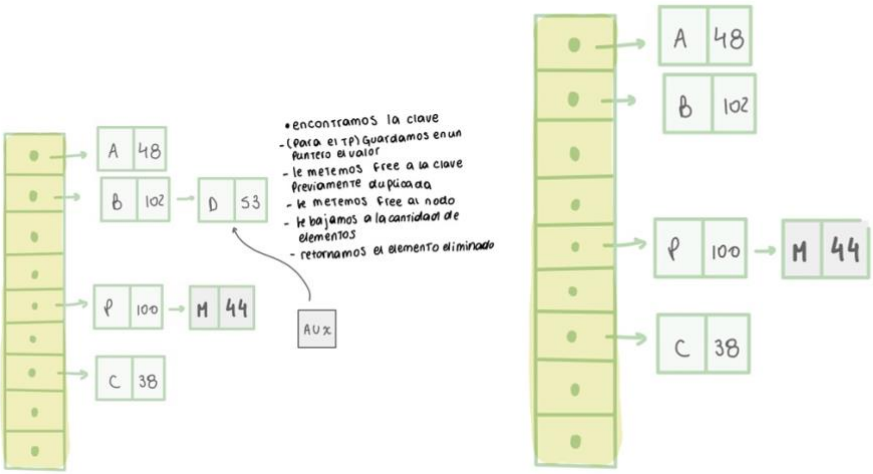
El **hash cerrado** tiene la característica de que los valores se guardan dentro de la misma tabla de hash, ósea no hay un TDA de por medio entre la clave y el dato. Cuando queremos insertar un elemento, debemos meter en la función de hash para que nos dé su posición dentro de ella y pueden pasar dos cosas: Que ese lugar este libre, entonces procedemos a insertar el par; o que ya haya un valor ocupando ese espacio, entonces lo que hacemos es: a partir de ese lugar buscar el más cercano para poder insertarlo; por eso este tipo de hash tiene un "direccionamiento abierto" ya que la clave puede estar en otra posición de la que me dio la función de hash.



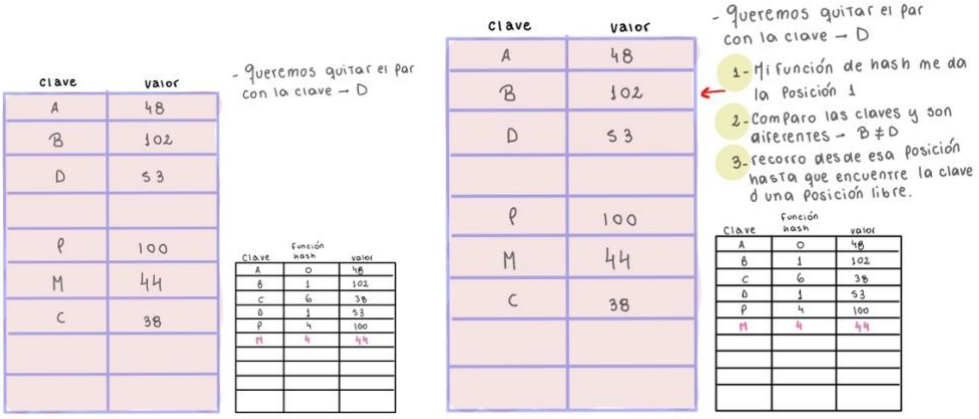


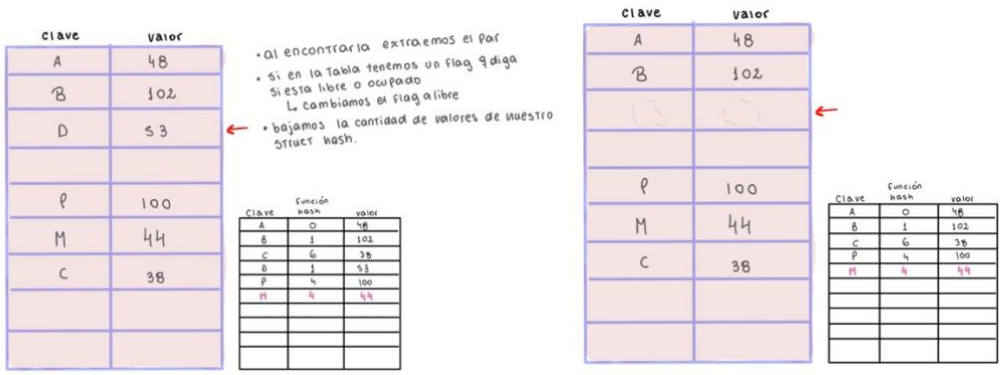
Para la función de quitar, el **hash abierto** “hashea” la clave que le pasamos y va a ir comparando con las claves de los nodos de la lista en esa posición, si la encuentra libera el nodo y no va a retornar NULL, ya que la clave no existe. Hay que recordar que si le insertamos la misma clave a nuestra función de hash SIEMPRE debe darnos el mismo número.



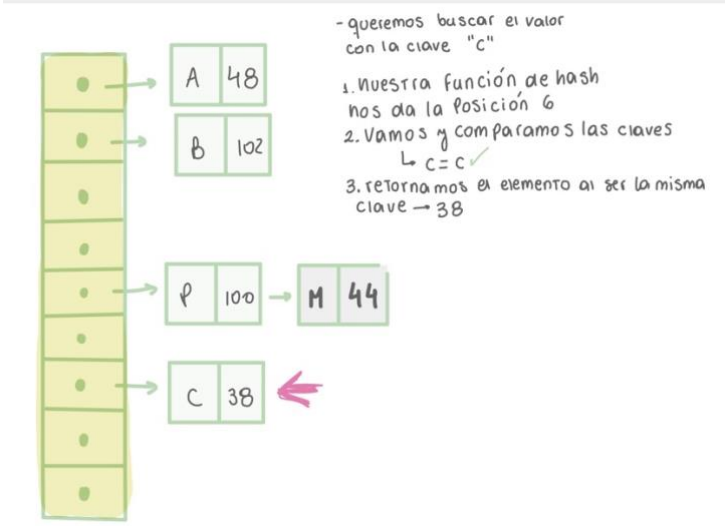


En un **hash cerrado** el quitar actúa de manera diferente. Hasheamos la clave y vamos a la posición que nos dicen, comparamos las claves: si es la misma, se quita el par de la tabla; pero si son diferentes se recorre la tabla hasta que la halláramos o si nos topamos con un espacio libre (acordémonos que nunca va a quedar un espacio libre entre la posición que nos dio la función de hash y el elemento insertado en otra posición, ya que al insertarlo buscamos el lugar libre mas próximo). Cuando hayamos quitado, debemos reordenar los valores y vamos recorriendo desde la posición que sacamos si hay un valor que previamente haya colisionado, lo metemos en la posición que quitamos el elemento. Si no encontramos entonces dejamos así como está la tabla. Esto se realiza, ya que sino el método de "probing lineal" (también hay otros métodos de búsqueda, es para dar un ejemplo) no va a tener efecto, y puede que cuando encuentre un espacio vacío va a dejar de buscar y puede que la clave quedo más abajo en la tabla.

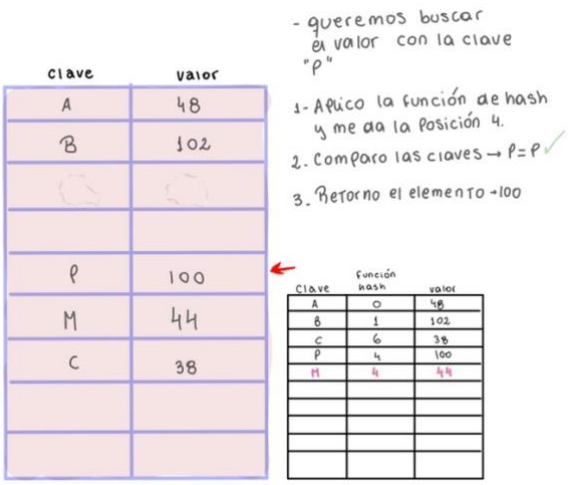




Para la búsqueda por medio de una clave, el **hash abierto** realiza el mismo algoritmo para buscar la clave como era un el quitar. Hasheamos la clave con la función de hash, vamos a esa posición y si no es la misma clave, vamos buscando en la lista que tenemos en esa posición hasta que la encontremos, entonces devolvemos el elemento; o si llegamos al final de la lista , eso quiero decir que no está la clave entonces se devuelve NULL.



En el **hash cerrado**, la búsqueda de la clave es el mismo algoritmo que en la operación de quitar. Hasheamos la clave, y si la clave no es la misma, vamos buscando hasta que la encontramos(entonces devolvemos el valor) o haya un espacio libre (que significa que la clave no se encuentra en la tabla).



3. Detalles de implementación

Para mi TDA hash, he utilizado un conjunto de nodos enlazados tipo “lista” para resolver las colisiones con las siguientes estructuras:

```c
typedef struct casillero{
    void* elemento;
    char* clave;
    struct casillero *siguiente;
}casillero_t;

struct hash {
    casillero_t **tabla;
    size_t ocupado;
    size_t cant_elementos;
    size_t capacidad;
};
```

El **struct casillero_t** es como está compuesto el nodo del par “clave-valor” y la **estructura del hash** que contiene la información necesaria para su implementación: el vector dinámico “*tabla*” es la tabla del hash, “*ocupado*” nos va a decir cuántos lugares ocupados hay en la tabla(va a ayudar para el rehash), “*capacidad*” es la capacidad de toda nuestra tabla y “*cant_elementos*” nos dice cuántos elementos hay.

FUNCIÓN DE HASH

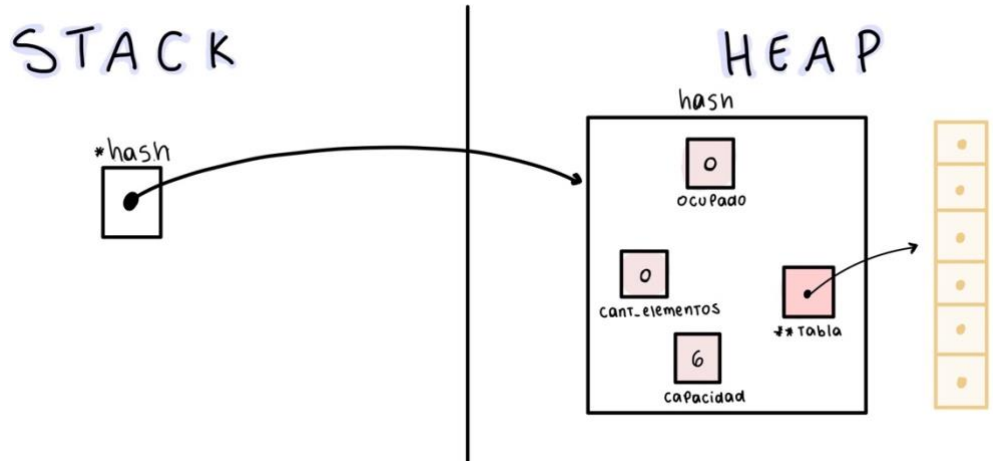
```c
/*
*post: me devuelve un numero en base a la clave
*/
uint32_t funcion_hash(const char *key)
{
    uint32_t hash, i;
    for(hash = i = 0; i < strlen(key); ++i){
        hash += key[i];
        hash += (hash << 10);
        hash ^= (hash >> 6);
    }

    hash += (hash << 3);
    hash ^= (hash >> 11);
    hash += (hash << 15);
    return hash;
}
```

Esta función va hacer algunas operaciones con desplazamientos bit a bit y sumándole lo que nos dé en la posición i de la clave (se puede observar en la imagen hash += key[i]) mientras se itera hasta llegar a la cantidad que nos dé el *strlen*(), ósea el largo de nuestro string. Y nos va a devolver un valor del tipo uint32_t , que sería aun número entero sin signo de 4 bytes. En mi implementación se va a ver que luego para poder tomar la posición como size_t lo casteo y hago modulo de la capacidad para que no me de una posición fuera de la tabla.

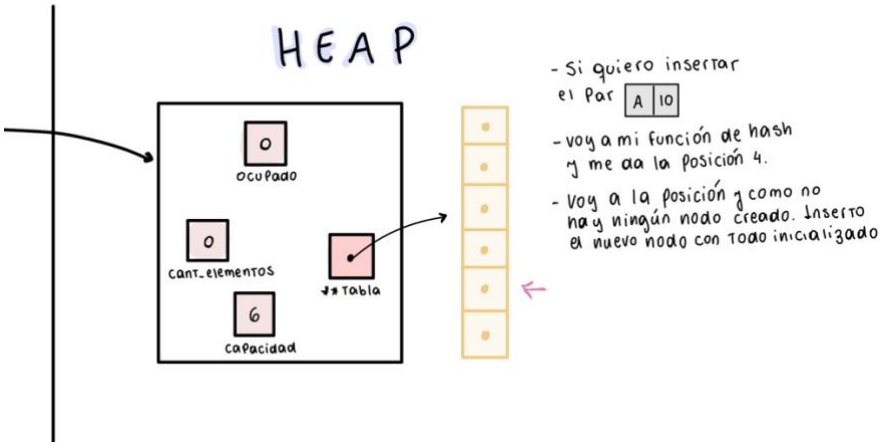
```c
size_t posicion = ((size_t)funcion_hash(clave))%hash->capacidad;
```

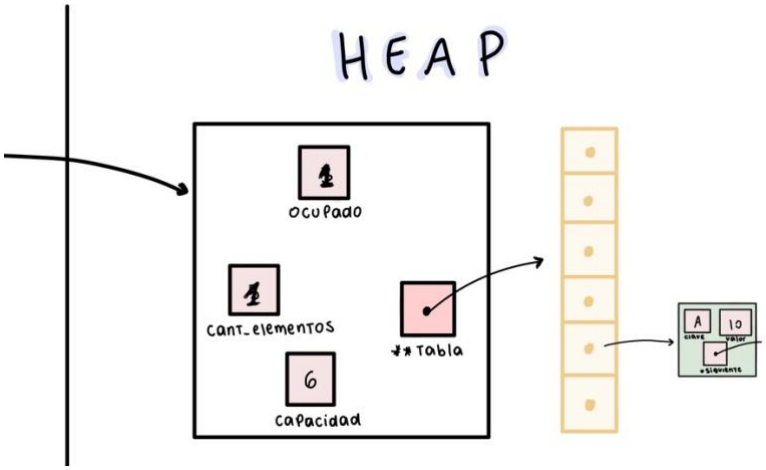
3.1. *Hash_crear()*

En esta función se inicializa el hash dependiendo de la capacidad que le paso el usuario, aunque si la capacidad es menor a 3; se le asigna 3.



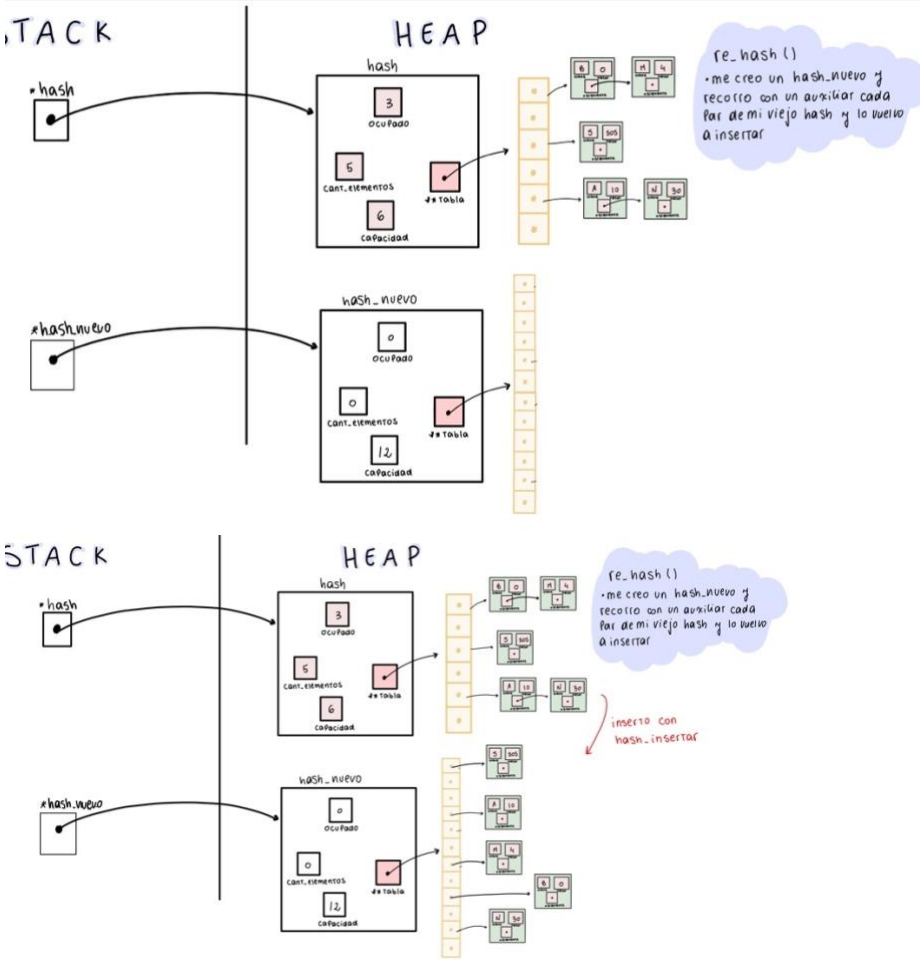
3.2 *Hash_insertar*()

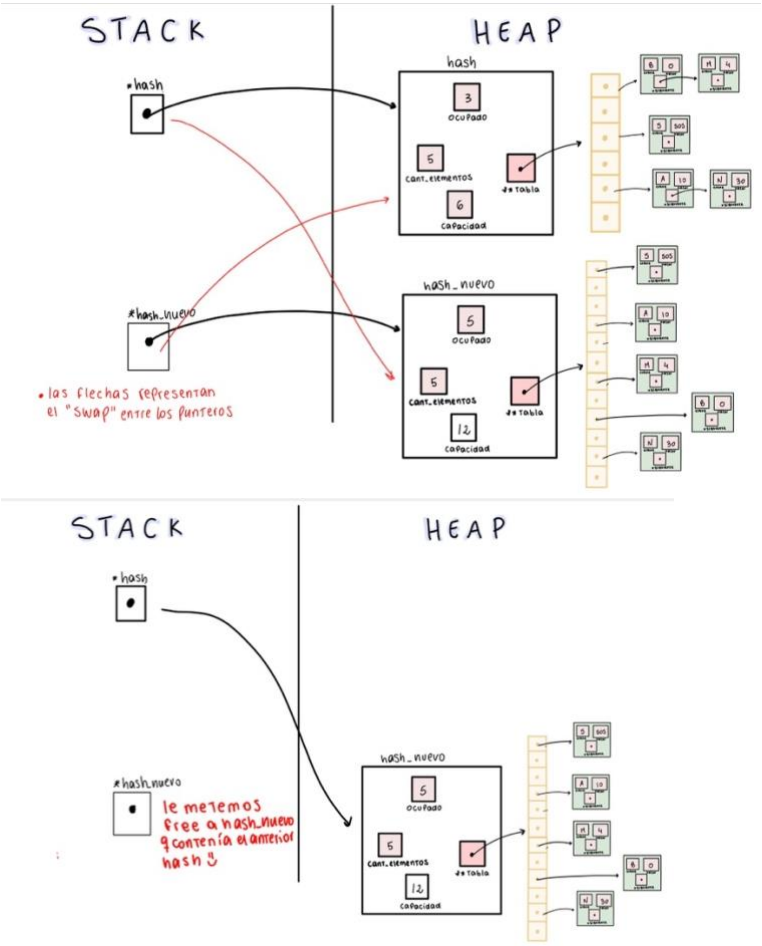




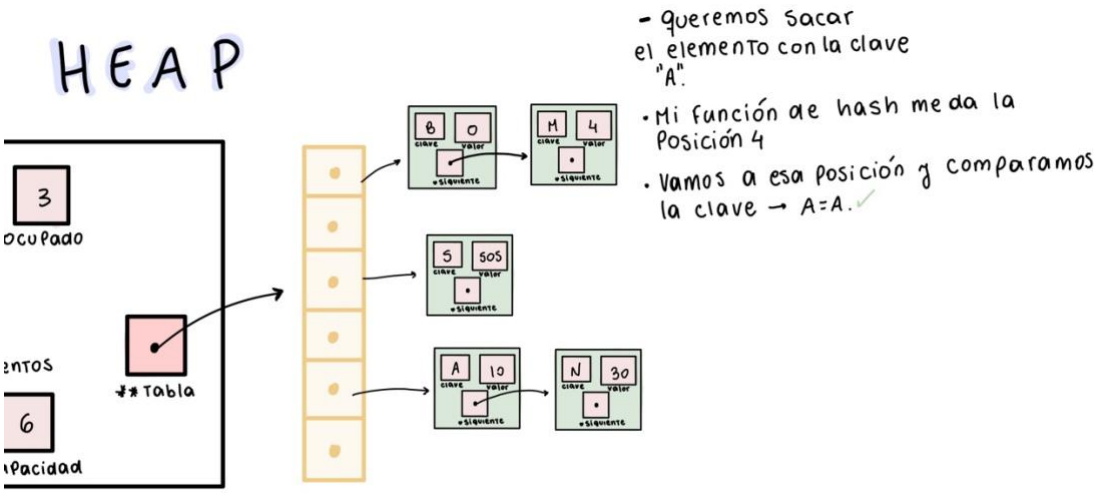
3.2.1 re_hash()

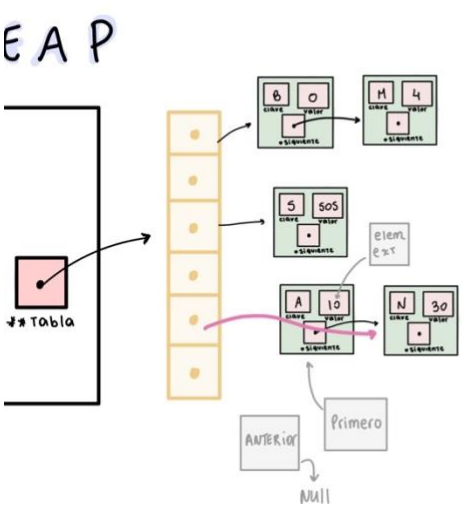
Para el *rehash*, solo me fijo en la capacidad de ocupado que tiene la tabla. Calculo el factor de carga dividiendo los lugares ocupados con la capacidad total, si esta me da mayor o igual a 0.75 entonces es hora de rehashear. Para poder lograrlo, primero me creo un hash nuevo con la funciones *hash_crear()* con el doble de la capacidad del anterior hash; luego, voy recorriendo todos los nodos del viejo hash e insertando en el nuevo con la función *hash_insertar(),* cuando ya haya recorrido todo, hago un respectivo "swap" para que la tabla que he hecho en el nuevo le quede al puntero de hash que utilizamos, la tabla vieja al puntero nuevo. Y finalmente, utilizo la función *hash_destruir()* para liberar la memoria de la tabla vieja, quedándome así con la tabla nueva con el doble de capacidad.





3.3 Hash_quitar()

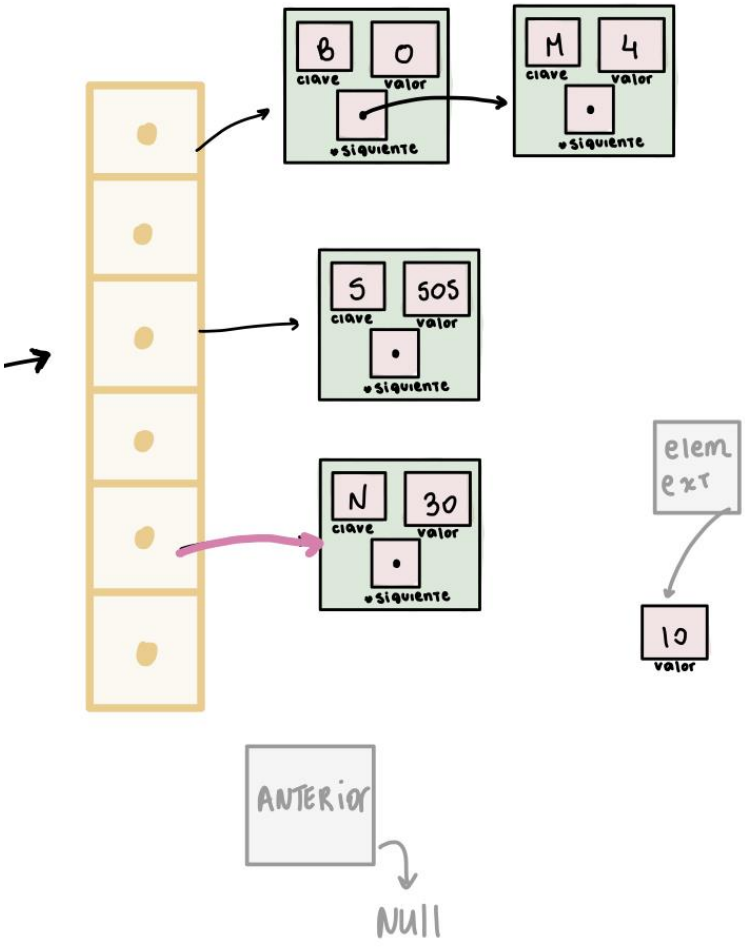




- como es la primera posición → el puntero anterior queda en Null

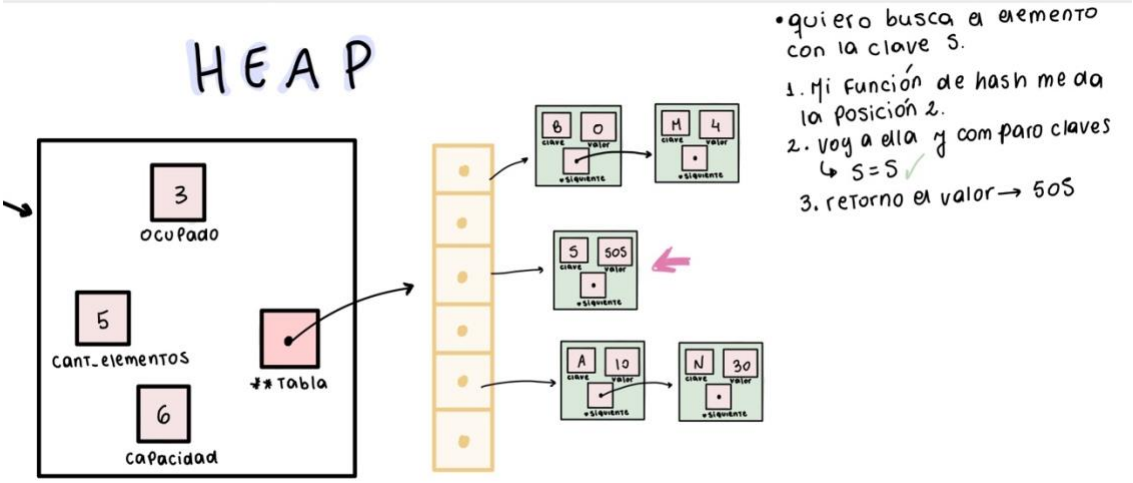
Pasos:

1. Apunto al elemento con el puntero "elemento_extraído"
2. Apunto la posición de la tabla al siguiente
3. Hago free a la clave
4. Hago free al nodo → por medio del puntero "primero"



3.4 Hash_obtener() y Hash_contiene()

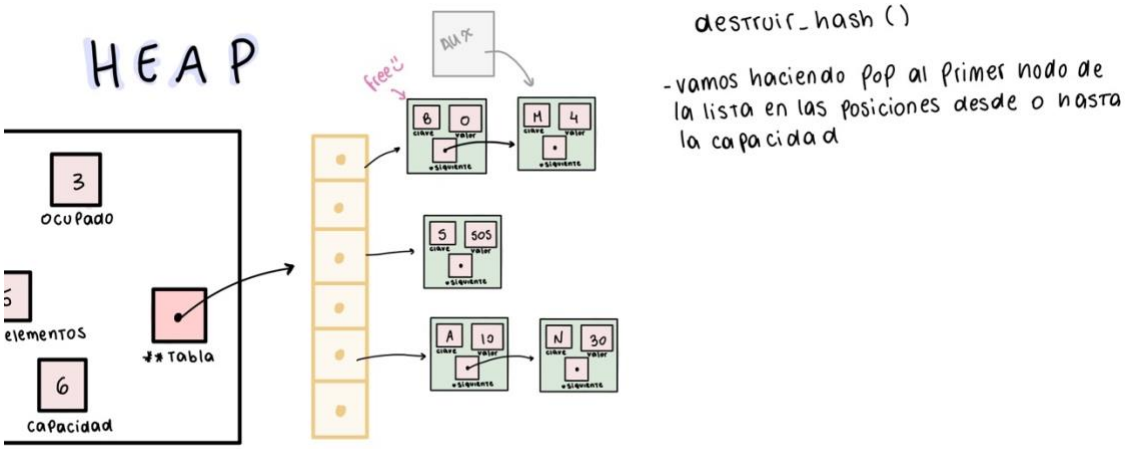
Estas funciones son bastantes parecidas, las dos van a buscar por medio de la clave que se les pase por parámetro*, hash_obtener()* si encuentra la clave nos va a devolver el elemento hallado, en cambio *hash_contiene()* si la encuentra nos va a devolver true , si no false.

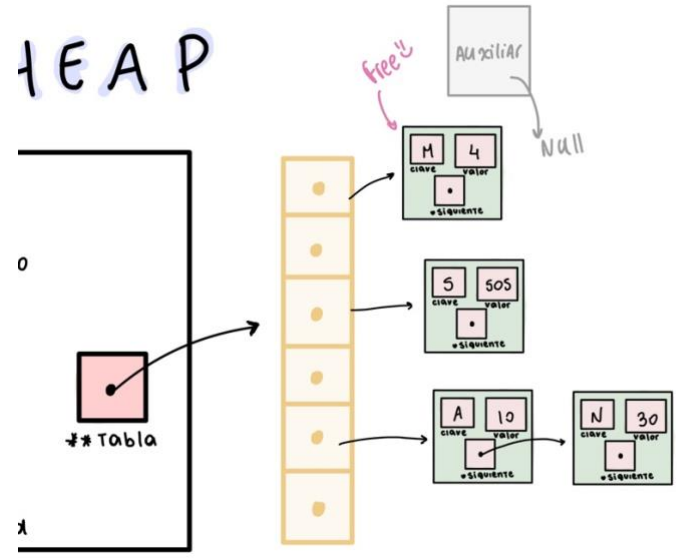


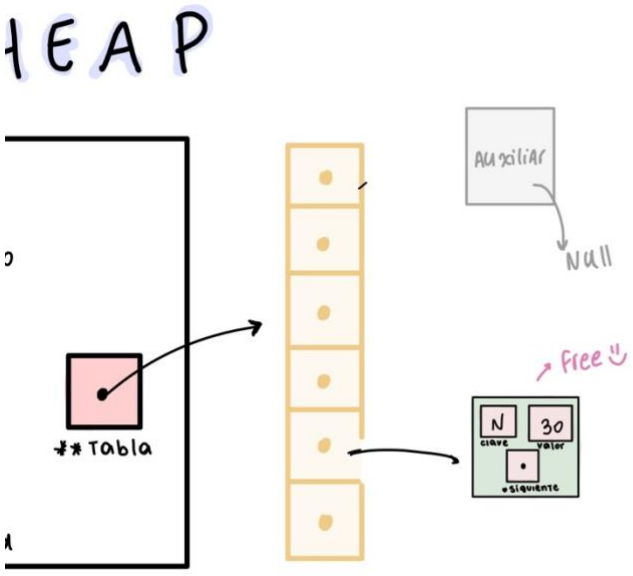
Para este ejemplo si hubiéramos utilizado la función *hash_contiene*() nos habría dado **TRUE**.

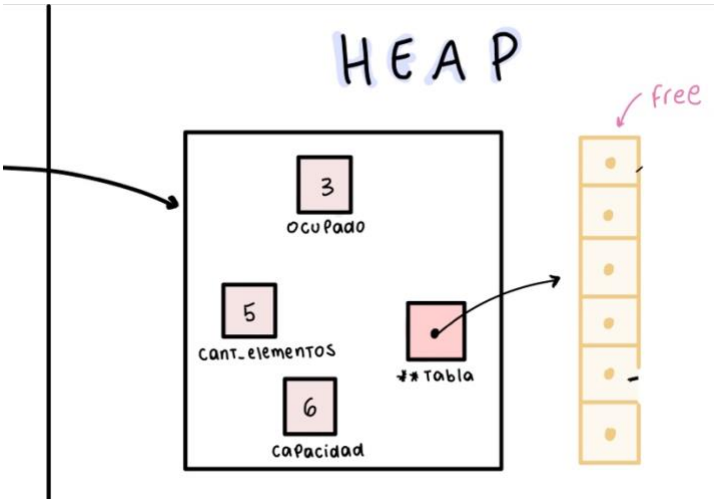
3.5 Hash_destruir() y Hash_destruir_todo()

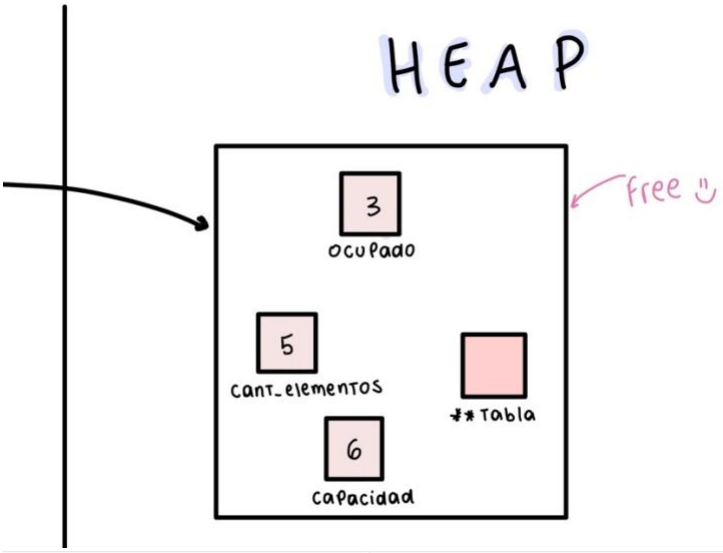
Estas funciones son parecidas entre sí, digamos que tiene el mismo objetivo: liberar la memoria utilizada por el hash. La diferencia es que *hash_destruir_todo()* va a destruir también el valor si lo habíamos reservado espacio en la memoria previamente.

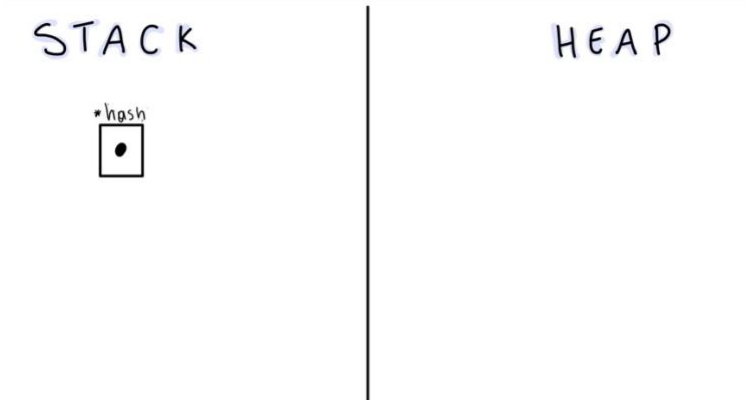



HEAP
Free!!
Auxiliar
Null
M
4
clave
valor
•siguiente
5
505
A
10
N
30
** Tabla


HEAP
Auxiliar
Null
Free!!
N
30
clave
valor
•siguiente
** Tabla


HEAP
free
3
ocupado
5
cant_elementos
** tabla
6
capacidad


HEAP
free
3
ocupado
5
cant_elementos
** tabla
6
capacidad


STACK
HEAP
*hash