

Informe TDA ~ hash abierto

Padrón: 108951

Nombre: Luciano Andrés Lorenzo

TDA HASH

Un hash es una estructura que contiene valores a los cuales accedemos a partir de una clave **única e inmodificable**.

A partir de la clave, lo que hacemos es pasarla por una función hash que definimos, con la cual podremos averiguar en que posición esta guardado el valor asociado a esa clave. Esta función hash debe depender solo de los parámetros que recibe y debe siempre devolver el mismo valor para la misma clave.

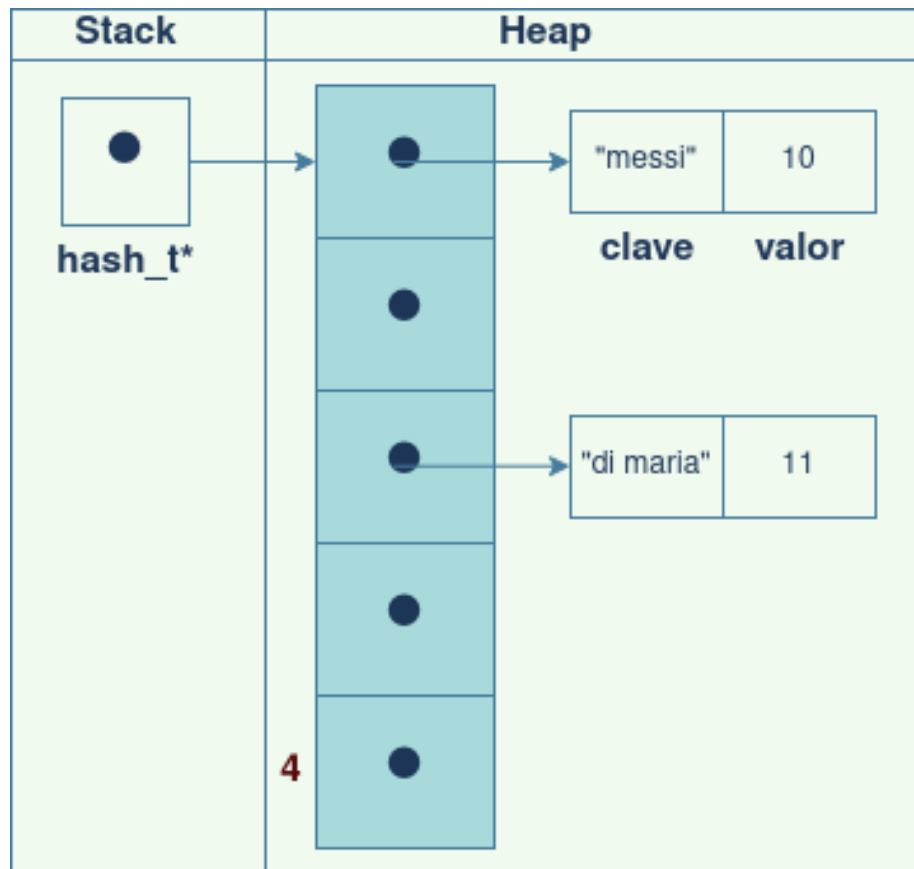
Casi cualquier función es valida como función de hash, pero muy pocas son en verdad útiles. Necesitamos una función que disperse lo máximo posible las posiciones de las claves, para que haya la menor cantidad de colisiones posibles (que dos claves guarden su valor en la misma posición)

Hay dos tipos principales de tablas de hash que se diferencian claramente en como almacenan los valores y manejan las colisiones, hash abierto y cerrado.

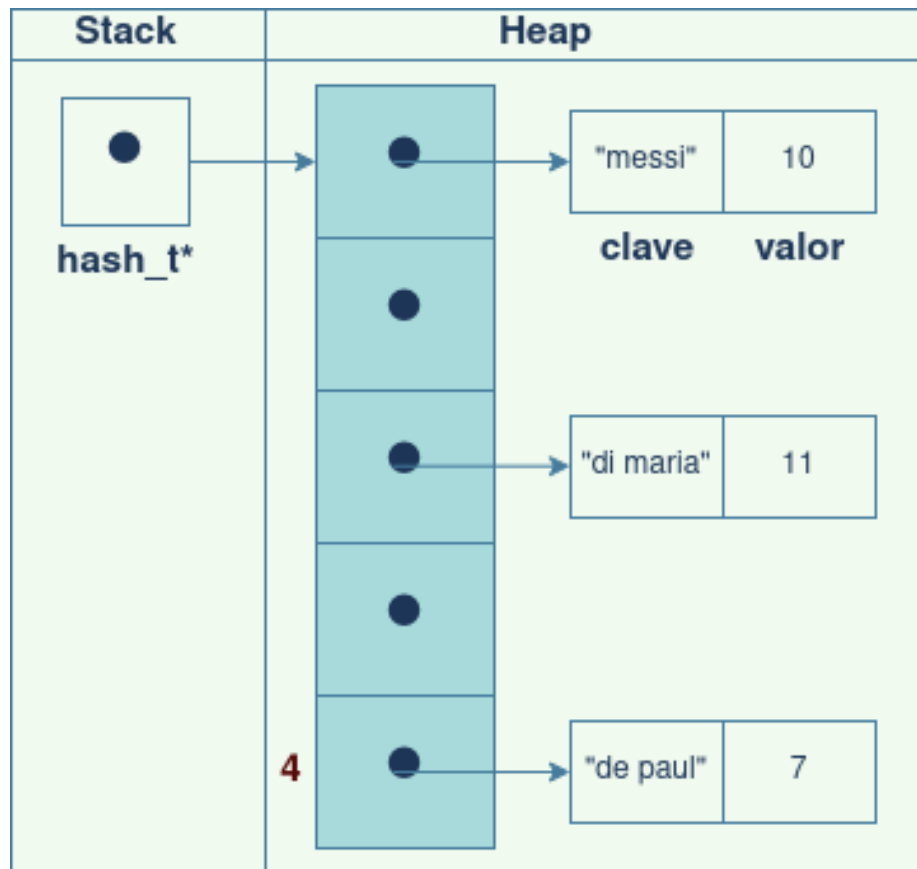
Hash abierto **Hash abierto:** Cada posición que nos devuelve la función hash es una lista, en la cual están todos los pares clave/valor que colisionan enlazados.

Dada una clave busco su lista asociada (el puntero a la lista asociada) con su hash, y desde ahí son operaciones de una lista. Para **insertar** inserto al final de la lista. Para **eliminar**, elimino de la lista. Y en el caso de **buscar** devuelvo el valor asociado a la clave.

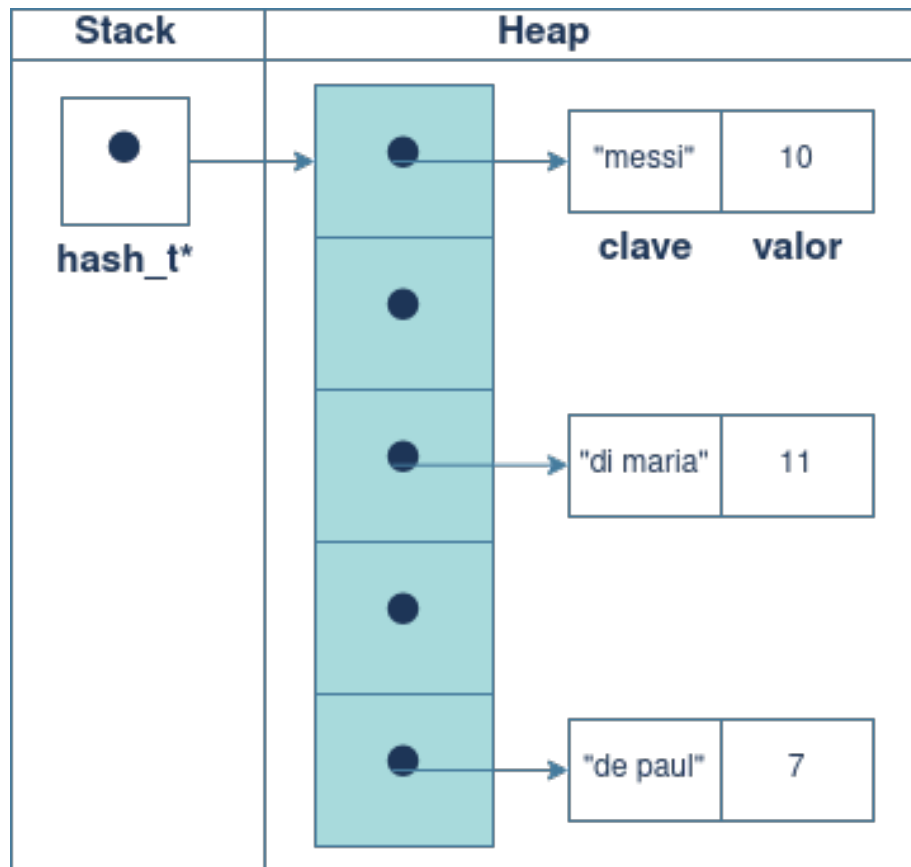
En este ejemplo para ilustrar, quiero insertar la clave "de paul" con valor 7. La función hash de la clave me devuelve la posición 4.



Como en este caso esta vacio, hago que el puntero de la posición 4 del vector de punteros apunte al nodo clave/valor "de paul"/7.



Y de esta forma esta insertado.



Hash cerrado Hash cerrado (con Probing lineal): Guarda los pares en un vector directamente.

Dada una clave busco su la posición del par clave/valor con su hash. De aquí todo depende si esa posición esta ocupada o no:

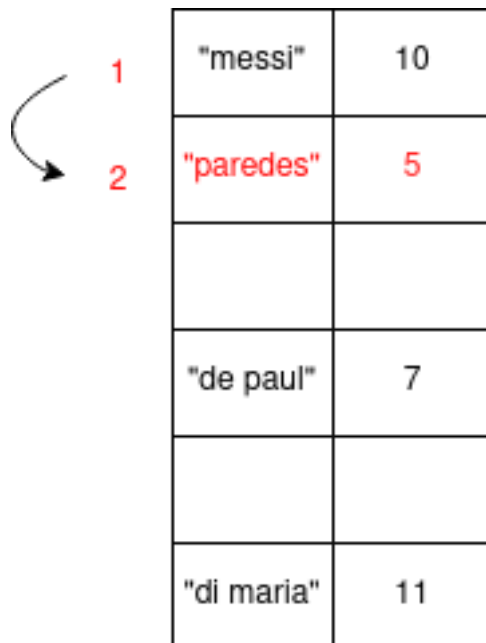
En el caso de que esté vacía, inserto el par clave/valor allí. En caso de eliminar o buscar, entonces no existe esa clave en la tabla.

En el caso de que este ocupada esa posición, es el caso de una colisión. Si queremos insertar, lo que haremos sera buscar el próximo lugar libre e insertar ahí.

En este ejemplo quiero insertar la clave "paredes" con valor 5, su hash me da la posición 0 de la tabla, pero esta se encuentra ocupada con la clave "messi" (hay una colisión en la posición 0)

"messi"	10
"de paul"	7
"di maria"	11

Por lo que simplemente inserto el par clave/valor en el siguiente espacio libre, en este caso es la siguiente posición (1).



1	"messi"	10
2	"paredes"	5
	"de paul"	7
	"di maria"	11

Para buscar, tenemos que ir comparando clave a clave avanzando en el vector

hasta que encontremos la que estamos buscando, en el caso de haber llegado a una lugar libre quiere decir que no existe la clave en la tabla. **Quitar** es la operación más complicada, ya que vaciando la posición donde esta la clave es un claro problema, porque estoy dejando un espacio vacío que puede llegar a confundir en una colisión para el caso de inserción y buscar. Por lo que siempre tengo que quitar y reacomodar de alguna manera la tabla. Avanzo hasta encontrar el próximo lugar vacío o una clave que pueda ocupar ese mismo espacio que vaciamos (en este último caso no hago nada).

Decisiones de diseño implementación hash abierto

Los pares clave valor se guardan en un struct que llame `nodo_t`, y que ademas contengan un puntero a un siguiente `nodo_t`. De esta forma tengo listas enlazadas que se ocupan de las colisiones, no me pareció necesario utilizar una estructura aparte para la lista enlazada.

La estructura del hash es simple, ademas del contador de los elementos actuales y capacidad máxima, tiene una vector de punteros a `nodo_t` (es decir un puntero a lista enlazada).

Si la capacidad sobrepasa el factor de carga, el hash realiza un rehash. Para implementarlo decidí solo 'realocar' la memoria para el vector de punteros a `nodo_t`, y doblar su capacidad en memoria. Para ello se crea una nuevo vector con esa doble capacidad, y sin eliminar de la memoria a los todos los nodos, es decir insertando los punteros, se van reconectando a la nueva tabla. Luego libero la memoria para la vieja tabla y la reemplazo con la nueva.

Para la función de hash, decidí utilizar una que dada una clave string, sume una cantidad especifica de caracteres y los multiplique por la cantidad de caracteres. Y teniendo ese numero, le aplique el operador módulo con la cantidad máxima de elementos en el hash, y devuelva el valor obtenido, el cual va a ser siempre una posición válida dentro del vector de listas.