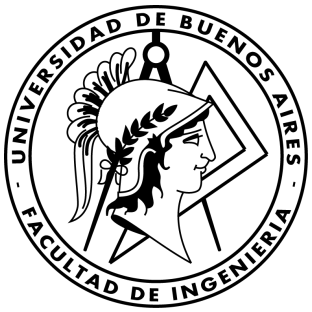

TDA - Hash

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

Alumno:	Bohórquez, Rubén
Número de padrón:	109442
Email:	rbohorquez@fi.uba.ar

Índice

1. Introducción

En este trabajo, se implementó una tabla de hash, que es una de las implementaciones más comunes de diccionario. Su principal utilidad es que el acceso a los elementos de una tabla de hash es prácticamente *O(1)*, por lo que se puede acceder a grandes cantidades de datos de manera muy rápida. En este reporte se hablará de los distintos tipos de hash y se precisará cómo fue realizada la implementación para este trabajo.

2. Teoría

2.1. Diccionarios

Un diccionario, de los que conocemos habitualmente, es un libro lleno de *palabras* ordenadas alfabéticamente donde a cada palabra se le asigna una *descripción* que le corresponda. Esta descripción depende del tipo de diccionario que sea. Por ejemplo, a la palabra *alto* se le puede asignar *De gran estatura, bajo, o tall* si es un diccionario de significados, de antónimos, o de español a inglés. Este concepto lo podemos generalizar, porque ya vimos que no importa qué le asignemos a cada *palabra*, ya que lo que nos interesa es el *contenido* de la descripción asociada. La palabra simplemente nos ayuda a *buscar* dicho contenido. Un diccionario es, entonces, en términos más generales, una colección de *pares*: por un lado tenemos el *valor o contenido*, que es lo que queremos almacenar, y por otro lado tenemos la *clave*, generalmente una string, que le asociamos para buscar dicho contenido. Una de las ventajas que tiene esta estructura es que le asigna una especie de *etiqueta* a todos los elementos, por lo que nos da un modo universal y sencillo de identificar todos los elementos del diccionario, sean del tipo que sean.

En un principio, parecería que esto no hace nada para ayudarnos con la velocidad de acceso a los datos. Al fin y al cabo, lo único que hace esta idea de asignarle una clave a cada elemento es cambiar el problema de acceder rápidamente a cualquier tipo de dato que teníamos a un problema de acceder rápidamente a claves. Ahí es cuando entra en juego la idea de las *funciones de hash*.

2.2. Función de Hash

El problema que tenemos ahora es que tenemos una serie de claves, y queremos una forma de ordenar sus datos asociados tal que podamos acceder a ellos con la clave en el menor tiempo posible (idealmente constante). Lo primero que se nos puede ocurrir es ordenar los datos según las claves en algún tipo de estructura, como una lista o un abb, que en el mejor de los casos (un abb bien balanceado) tiene una complejidad de *O(log(n))*. Pero hay una operación muy básica que es *O(1)*: acceder a una posición *conocida* de un vector. Por lo general la parte que toma más tiempo al acceder a un elemento en una estructura es, precisamente, *buscarlo*. Una vez sabemos dónde está el elemento, es cuestión de simplemente acceder a él. Entonces, lo más natural sería utilizar de alguna forma la *clave* (que es precisamente la información que necesitamos para empezar siquiera a buscar un elemento) como si fuera el *índice* del valor asociado en la estructura donde los guardamos. Lo único que nos falta es una manera de asociar a las *claves* un valor numérico que podamos usar como, por ejemplo, el índice de un vector. Este es el rol que cumple la función de hash. Una función de hash recibe una clave (generalmente una string) y devuelve un número dentro de un rango fijo (un número de x cifras binarias). Es evidente que lo mejor sería que a cada clave le asigne un número distinto, pero, como esto es imposible en la práctica debido a que hay *muchas* claves posibles, lo que se busca es que sea *muy poco probable* que dos claves cualesquiera den el mismo resultado. Esto se traduce a que las claves tienen que estar *bien distribuidas* entre todos los posibles valores que devuelve la función. Los detalles se escapan del alcance de este reporte, pero, en resumidas cuentas, esto lo logra *mezclando* distintas partes de la clave varias veces para introducir *entropía*, de modo que pequeños cambios a la clave resultan en hashes completamente distintos. Existen muchas funciones de hash para distintas aplicaciones, pero en este caso nos conviene una que sea rápida (al fin y al cabo, el punto de todo esto es acceder rápido a un conjunto de datos).

2.3. Tablas de Hash

Ahora tenemos (casi) todo lo que necesitamos para construir nuestra estructura de acceso rápido a datos. Primero, creamos una tabla del tamaño que veamos conveniente para almacenar nuestros datos. Ahora, en vez de insertar cada elemento en la tabla en orden, le asignamos una *clave*, le aplicamos nuestra función de hash a dicha clave, y el valor que nos devuelva (módulo el tamaño de la tabla, para ajustar el rango de valores que devuelve la función de hash al rango de índices válidos) es la posición en la que tenemos que guardar ese elemento en la tabla, y lo guardamos junto con su clave como un *par clave-valor*. Luego para buscar un elemento, simplemente usamos la clave, aplicamos la función de hash, y vamos a ese índice en la tabla. Pensemos un poco en la complejidad algorítmica de algunas operaciones en esta tabla: para insertar un elemento, necesitas aplicar la función de hash, dirigirte a esa posición en el vector, y luego aplicar las operaciones necesarias para insertar un sólo elemento (copiar y mover memoria, etc.). Lo mismo puede ser dicho de las funciones de obtener y eliminar elementos. Todas estas operaciones son *O(1)* (la función de hash *técnicamente* depende del tamaño de la clave, pero esto es despreciable a menos que usemos claves de varios miles de caracteres en longitud). Esto es precisamente lo que queríamos, una estructura cuyo tiempo de acceso a datos es constante. Sólo hace falta encargarse de un problema más: ¿Qué pasa si dos claves distintas nos devuelven el mismo hash?

2.4. Colisiones de Hash

Como ya mencionamos antes, existen muchas más claves posibles que valores que puede devolver nuestra función de hash. Además, se nos suma el problema de que también hay más valores de hash que espacios en el vector donde guardamos elementos. Es casi seguro que nos vamos a encontrar con claves que tengan el mismo valor de hash y que tengan que ir en la misma posición del vector. A esto lo llamamos *colisiones de hash*, y es precisamente lo que buscamos minimizar con una función de hash con claves bien distribuidas. Las dos soluciones más obvias que se le pueden llegar a ocurrir a alguien son las dos soluciones que usamos para este problema: o bien guardamos todos los elementos con el mismo hash *en la misma posición*, o bien buscamos guardarlos *en otra posición distinta*. La primera solución nos da lo que se conoce como **Hash abierto, de direccionamiento cerrado**, mientras que la segunda corresponde a un **Hash cerrado, de direccionamiento abierto**.

2.4.1. Hash abierto, direccionamiento cerrado:

Si queremos guardar todos los elementos con el mismo hash en la misma posición, está claro que lo que queremos que vaya en cada posición ya no es uno sólo de nuestros elementos, sino una *colección*, o *lista* de elementos, que contenga a todos los que les corresponde esa posición. Podemos reutilizar entonces el concepto de lista enlazada, y guardar en vez de sólo pares clave-elemento, pares *enlazados* clave-elemento, de modo que si intentamos almacenar algo en una posición ocupada, simplemente añadimos el nuevo par al final de la lista enlazada. Para obtener elementos, podemos usar la función de hash para encontrar la posición del elemento y recorrer la lista enlazada hasta encontrar el que tenga la misma clave. Este método de resolver colisiones se llama *encadenamiento*. Se le dice hash *abierto* porque no todos los elementos están almacenados *dentro* de la tabla de hash, sino que hay algunos por fuera en las listas enlazadas. Se le dice de direccionamiento *cerrado* porque la *dirección* que nos da la función de hash es definitiva, el elemento que estamos buscando dentro de la tabla se encuentra sí o sí dentro de algún lugar de la lista en esa posición o no está en la tabla.

2.4.2. Hash cerrado, direccionamiento abierto:

La otra opción ante una colisión es tratar de encontrar otro lugar donde guardar el elemento dentro de la misma tabla. Existen varios métodos de decidir cuál debe ser esa posición, pero aquí explicaremos dos. La primera es muy sencilla: guardar el elemento en la posición *inmediatamente siguiente*. Si esa posición también está ocupada, se va a la siguiente a esa. Así hasta encontrar

una posición vacía. Esto es lo que se conoce como *probing*, y no necesariamente tiene que ser revisando las posiciones siguientes una a una (lo que se conoce como *probing lineal*), se puede usar otra secuencia para ir buscando posiciones a partir de la original (por ejemplo, una sucesión cuadrática). Para encontrar un elemento simplemente se busca en esa sucesión a partir de la posición que indique la función de hash hasta encontrar un elemento con la misma clave. La otra solución sería asignar una zona específica de la tabla reservada exclusivamente para elementos que colisionaron. Esta zona se la conoce como *zona de desborde*, que por lo general tiene un tamaño predefinido en función del tamaño total de la tabla. En esta zona simplemente vamos insertando los elementos por orden de llegada, y si tenemos que buscar algo en la zona de desborde hay que recorrer los elementos uno por uno. Este tipo de hash se lo conoce como *cerrado* porque todos los elementos se encuentran dentro de la tabla, pero su direccionamiento es *abierto* porque la dirección que devuelve la función de hash *puede o no* contener el elemento con dicha clave. Esto genera un problema al eliminar elementos: si al buscar una clave el hash devuelve una posición vacía, puede ser porque no hay nada almacenado en la tabla con esa clave, o porque *había otra cosa en esa posición* que fue borrada, entonces lo que estamos buscando fue almacenado *en otra parte de la tabla*. Para resolver este problema, a cada posición de la tabla hay que añadirle una especie de marca que indique si una posición vacía *siempre* ha estado vacía, o si contenía algo que *fue eliminado* de la tabla en algún punto.

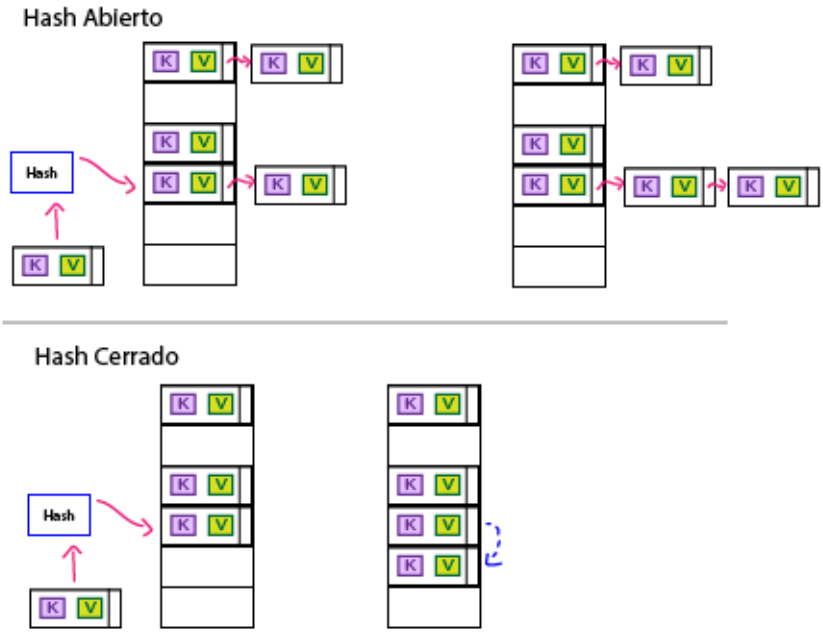

Figura 1: Colisiones en un hash abierto vs. cerrado.

2.4.3. Rehasheo:

Ambos métodos tienen en común que, si tenemos la mala suerte de que hay muchas colisiones entre nuestros elementos, la complejidad algorítmica va aumentando ya que tenemos que recorrer varios elementos para llegar al que queremos. Es evidente que, mientras más grande sea la tabla, menos probable es que haya dos elementos con el mismo índice. Pero tampoco conviene crear directamente una tabla con una capacidad muy grande para guardar pocos elementos porque sería muy ineficiente en recursos. La forma de solucionar esto es, como en los vectores dinámicos, agrandar la tabla según vayamos necesitando más espacio. Este proceso es conocido como *rehash*.

Al agrandar la tabla, como la posición de cada elemento era el hash de la clave *módulo el tamaño de la tabla*, se tienen que reinsertar todos los elementos otra vez, ya que muy probablemente su posición dentro de la tabla tenga que cambiar para ser consistente con la función de hash. Para saber si es necesario agrandar la tabla de hash o no, se suele definir lo que se conoce como *factor de carga* del hash, que es la cantidad de elementos en la tabla en proporción a su tamaño total. Para evitar que se degrade el rendimiento, es común rehashear una tabla cuando su factor de carga supere un máximo permitido (alrededor de 0.7 - 0.8 normalmente). También se puede rehashear a una tabla *más pequeña* si el factor de carga baja de un cierto mínimo para ahorrar recursos (especialmente espacio en memoria).

3. Detalles de implementación

El tipo de hash que se implementó en esta ocasión fue un hash abierto de direccionamiento cerrado. La estructura en sí es prácticamente lo discutido en la sección teórica: un vector dinámico que hace las veces de tabla, y además dos enteros que guardan la capacidad total de la tabla y el número de elementos actualmente almacenados. El vector dinámico almacena *pares enlazados*, que contienen la clave, el elemento en sí que se está almacenado, y una referencia al siguiente par de la lista en caso de que haya colisiones que encadenar. Específicamente, el vector dinámico es de *punteros* a pares, debido a que fue más fácil implementar algunas funciones de esa forma. No se vio la necesidad de incluir la función de hash dentro de la estructura porque no se le dio al usuario la opción de proporcionar una por su lado. La función de hash escogida fue *djb2*. A continuación se explica brevemente lo que hace cada función:

- **Crear:** funcionamiento bastante trivial. Reserva la memoria necesaria para una tabla de hash del tamaño que el usuario especifique. Hay un tamaño mínimo para la tabla de hash, así que si se intenta crear una más pequeña se crea en su lugar una con el tamaño mínimo. Lo único notable para aclarar es que el vector dinámico donde se almacenan los pares se inicializa *todo en 0's*, ya que almacena punteros y el resto de las funciones interpretan un puntero nulo como un espacio vacío.

- **Insertar:** Esta función inserta un elemento a la tabla, y si ya había otro elemento con la misma clave almacenado, lo reemplaza. Lo primero que hace es crear el par en la memoria dinámica, guardando el elemento directamente, y *copiando* la clave a otra dirección de memoria antes de guardarla. Esto es para evitar que la tabla se rompa si el usuario elimina o modifica la dirección de memoria con la clave original. Lo siguiente es buscar la posición que le corresponde al par en el vector. Se aplica la función de hash módulo la capacidad para obtener la posición. Si esa posición está vacía, se coloca el par ahí y ya. Si está ocupada, se itera sobre los pares enlazados en esa posición hasta que se llegue al final de la lista o se encuentre un par con clave la duplicada. Si se encuentra un par con la misma clave, se almacena el elemento en un puntero auxiliar por si lo necesita el usuario, luego se reemplaza por el elemento nuevo, y se libera toda la memoria reservada por el nuevo par (ya que no se insertó) antes de volver. Por último, se actualiza la cantidad de elementos almacenados (si se insertó un nuevo par en lugar de reemplazar un elemento existente), se rehashea en caso de ser necesario (explicado más adelante), y se devuelve el hash.

- **Obtener:** Probablemente la función más sencilla después de crear. Se dirige a la posición del vector que le indique el hash de la clave recibida, e itera la lista enlazada en esa posición hasta que encuentre un elemento con la misma clave o llegue al final, en cuyo caso sabe que no hay un elemento con esa clave. La función *contiene* funciona exactamente igual, solo que devuelve un booleano en lugar del elemento.

- **Quitar:** La eliminación también funciona de manera parecida a *Obtener*. Se dirige a la posición que le indique la función de hash, y recorre la lista enlazada en esa posición hasta encontrar el elemento a eliminar. La eliminación en sí es exactamente igual a la eliminación

en una lista enlazada: mientras se busca el elemento a eliminar se va guardando el elemento anterior, de modo que cuando se consiga, se guarda el elemento a eliminar en una variable auxiliar, y se asigna el elemento que le sigue como el siguiente de su anterior. Una vez quitado el par se destruye, liberando la memoria de la clave, la memoria del par en sí, y devolviendo el elemento eliminado.

- **Destruir/Recorrer/Rehashear:** Estas tres funciones fueron agrupadas porque su funcionamiento es prácticamente idéntico. Las tres recorren todos los pares de la tabla, para eliminarlos, pasarlos a una función, o añadirlos a otro hash el doble de grande, respectivamente. El modo en que hacen esto es recorriendo todas las listas enlazadas del vector dinámico una por una, ya que es la forma más sencilla de recorrer *todos* los elementos (o hasta que la función lo indique en el caso de *recorrer*). Al finalizar el recorrido, recorrer devuelve la cuenta de los elementos iterados, destruir libera la memoria del vector y del hash, y rehash intercambia el hash original por el rehasheado para devolverlo al usuario y destruye el que ya no se va a usar.z