TDA HASH

[7541/9515] Algoritmos y Programación II

Primer cuatrimestre de 2022

Alumno:	Subero, Andrés
Número de padrón:	109114
Email:	asubero@fi.uba.ar

1. Introducción

El trabajo práctico tiene como finalidad poner en práctica un diccionario, utilizando el TDA HASH y sus primitivas. Existen dos tipos de HASH, abierto y cerrado. El HASH abierto tiene los elementos fuera de la tabla, es decir, guarda los elementos o los punteros a los elementos en una estructura aparte, y la tabla guarda el puntero a la estructura y a su vez esta estructura puede tener un puntero a otra estructura que contiene otro elemento, esto hace que la posición generada por la función HASH sea definitiva para encontrar el elemento, si no se encuentra en la primera estructura, se busca en la siguiente. Esto es lo que se conoce como direccionamiento cerrado (la dirección es la que da la función).

Por otro lado, el HASH cerrado es aquel que contiene el elemento o el puntero al elemento dentro de la tabla. Como solo se puede tener un elemento por posición, cuando se generan las colisiones se busca la siguiente posición libre más cercana, haciendo que la posición dada por la función hash no sea definitiva, a esto se le conoce direccionamiento abierto.

Para la implementación se utilizó un hash abierto con direccionamiento cerrado, siendo la tabla un vector de punteros a nodos. Cada nodo contiene un puntero a una clave (string en este caso), un puntero a un elemento(valor) y un puntero al nodo siguiente, que puede ser null si no existe otro nodo. La estructura que contiene el vector, llamada hash, contiene además dos int, uno para guardar la cantidad de elementos guardados y uno para la capacidad del hash.

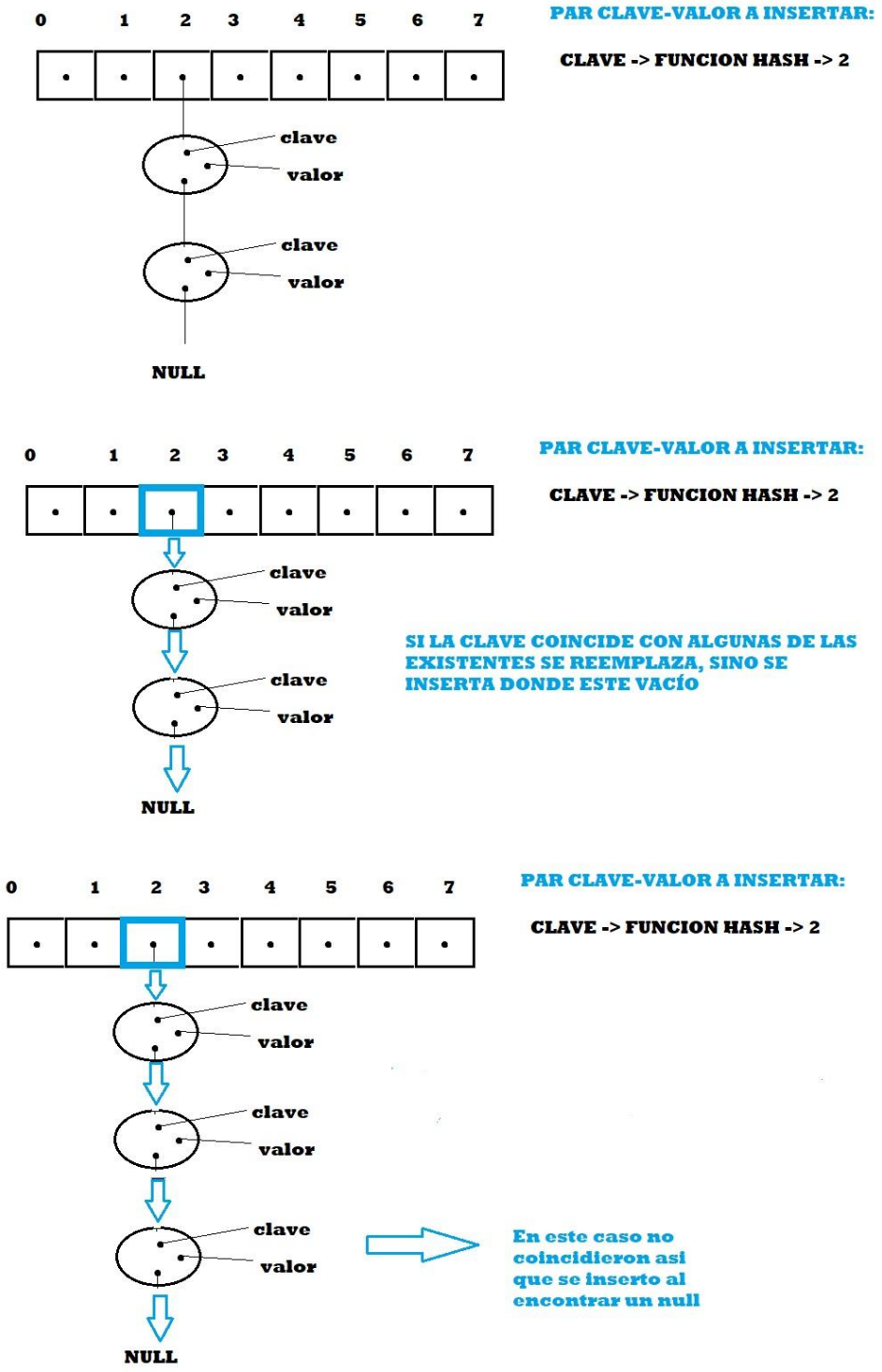
Para confirmar la correcta implementación del TDA, se realizaron pruebas unitarias, las cuales simulan la mayor cantidad de casos posibles (en este caso son muy importante los casos bordes) y con cada prueba se asegura que ese caso está cubierto y no fallará en el futuro.Para la realización de las pruebas, se intentó realizar primero la prueba y luego implementar la mínima solución (TDD).

2. Teoría

2.1) Primitivas de un HASH abierto

- Insertar: Para insertar un par clave-valor, se genera una posición con la función hash, pasándole como parámetro la clave. La función hash devuelve la posición, con ella buscamos la posición en el vector o se puede encontrar con las siguientes situaciones: ¿ está vacía la posición? se inserta; ¿hay par clave-valor? se revisa si la clave es la misma clave del par que queremos insertar, en el caso de ser la misma solo se cambia el elemento existente por el nuevo, si no es la misma clave se pasa al siguiente nodo (o la estructura que se utilice) y se repite el proceso hasta lograr insertar o actualizar.

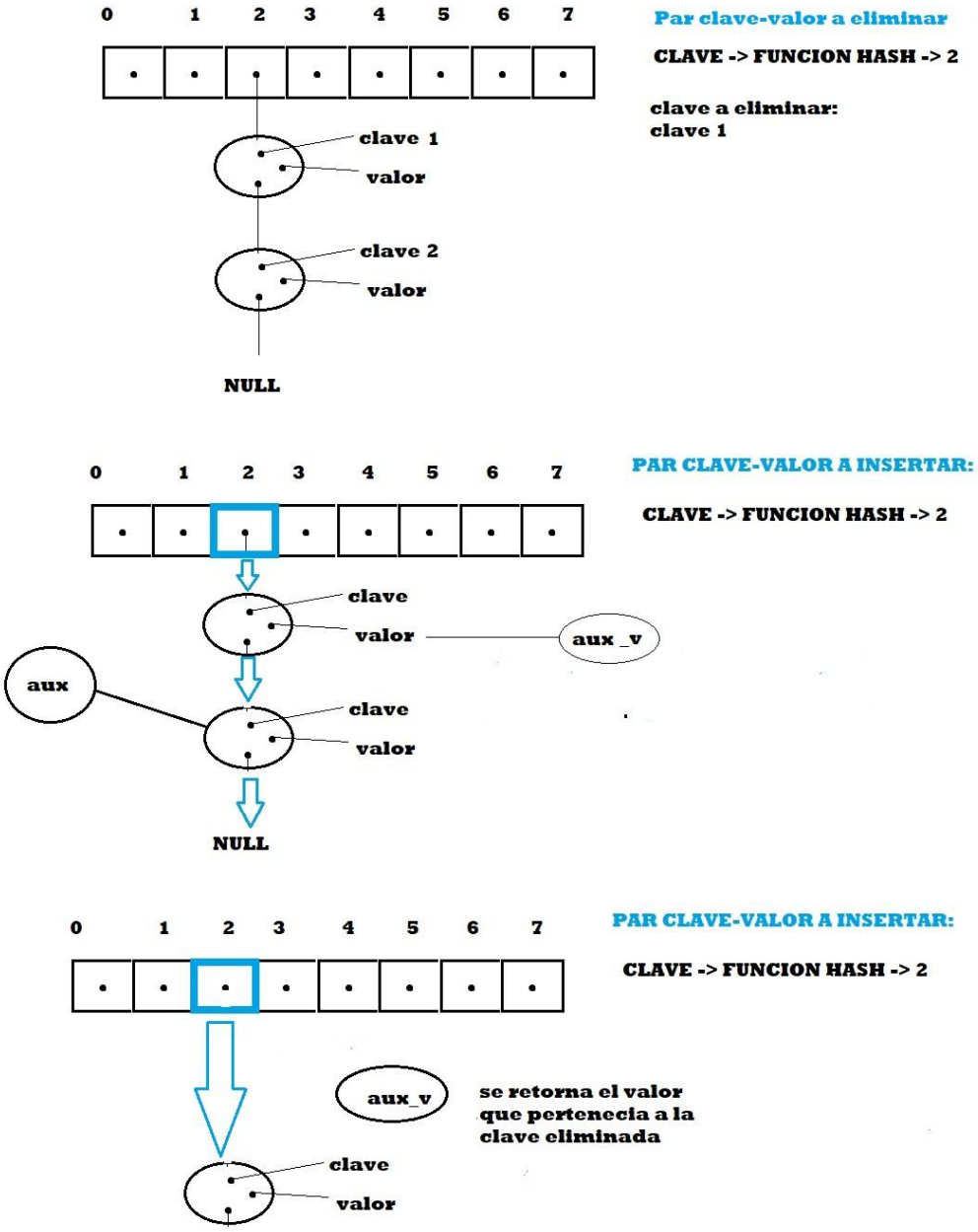
  Insertar implica lo siguiente: se reserva memoria para la nueva estructura que contendrá al par clave-valor, si se tiene éxito al reservar, se reserva memoria para la clave, si se tiene éxito se le asigna la copia de la clave y el elemento o su puntero a la nueva estructura creada. En el caso de no tener éxito al reservar memoria, se libera la memoria que se haya reservado (en el caso de que falle la clave se libera la del nodo) y se retorna null (o lo que diga la convención). Se aumenta 1 a la cantidad de elementos.



- Eliminar: Se obtiene la posición con la clave y la función hash, se busca en el vector la posición y al igual que al insertar pueden ocurrir las mismas 3 situaciones: ¿está vacía la posición? no hay nada que eliminar así que se devuelve lo que se tenga por convención (null en este caso), ¿hay par clave-valor ? Se compara la clave buscada con la que existe, si son iguales,

se elimina, si no son iguales se pasa al siguiente nodo y se repiten los mismos pasos.

Eliminar implica lo siguiente: se guarda en un auxiliar el puntero al siguiente nodo, en otro auxiliar se guarda el elemento o su puntero, y luego se elimina el nodo (se libera la memoria ocupada por la estructura que contiene a la clave y valor buscada) y se retorna el auxiliar que apunta al siguiente del nodo eliminado para poder conectarlo con el nodo predecesor del eliminado. Se disminuye en 1 la cantidad de elementos.



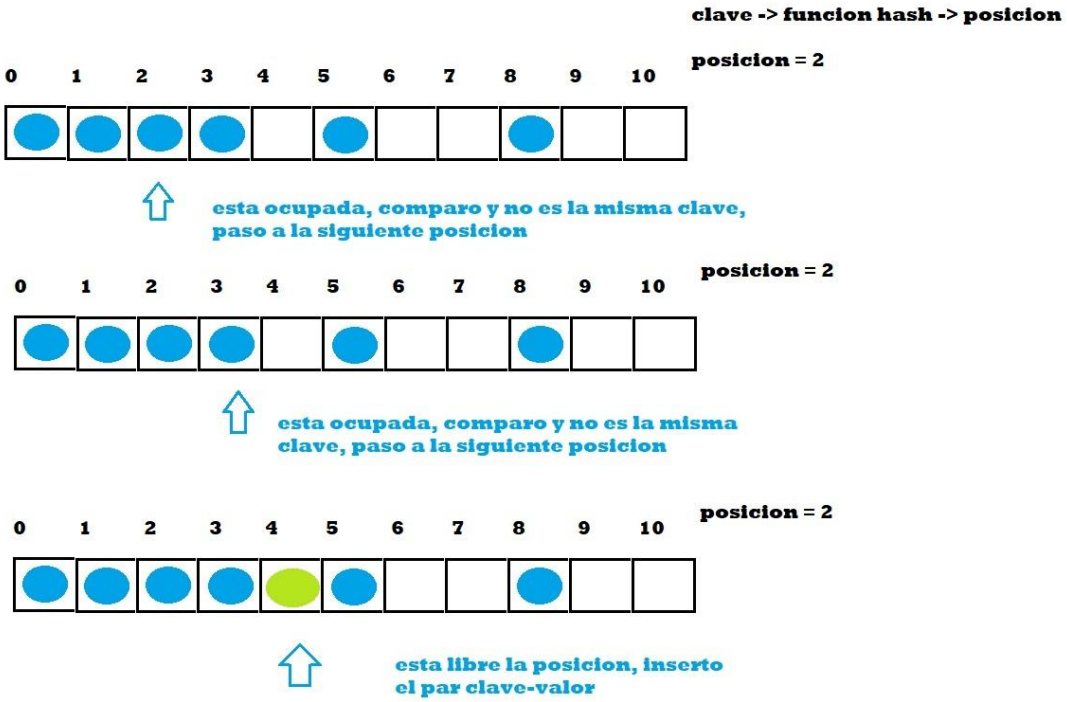
- Buscar: Se obtiene la posición con la clave y la función hash, se busca en el vector la posición y al igual que al insertar y eliminar pueden ocurrir las mismas 3 situaciones: ¿está vacía la posición? no existe el elemento valor buscado así que se devuelve lo que se tenga por convención (null en este caso), ¿hay par clave-valor ? Se compara la clave buscada con la que existe,

si son iguales, se retorna el puntero al elemento, si no son iguales se pasa al siguiente nodo y se repiten los mismos pasos.

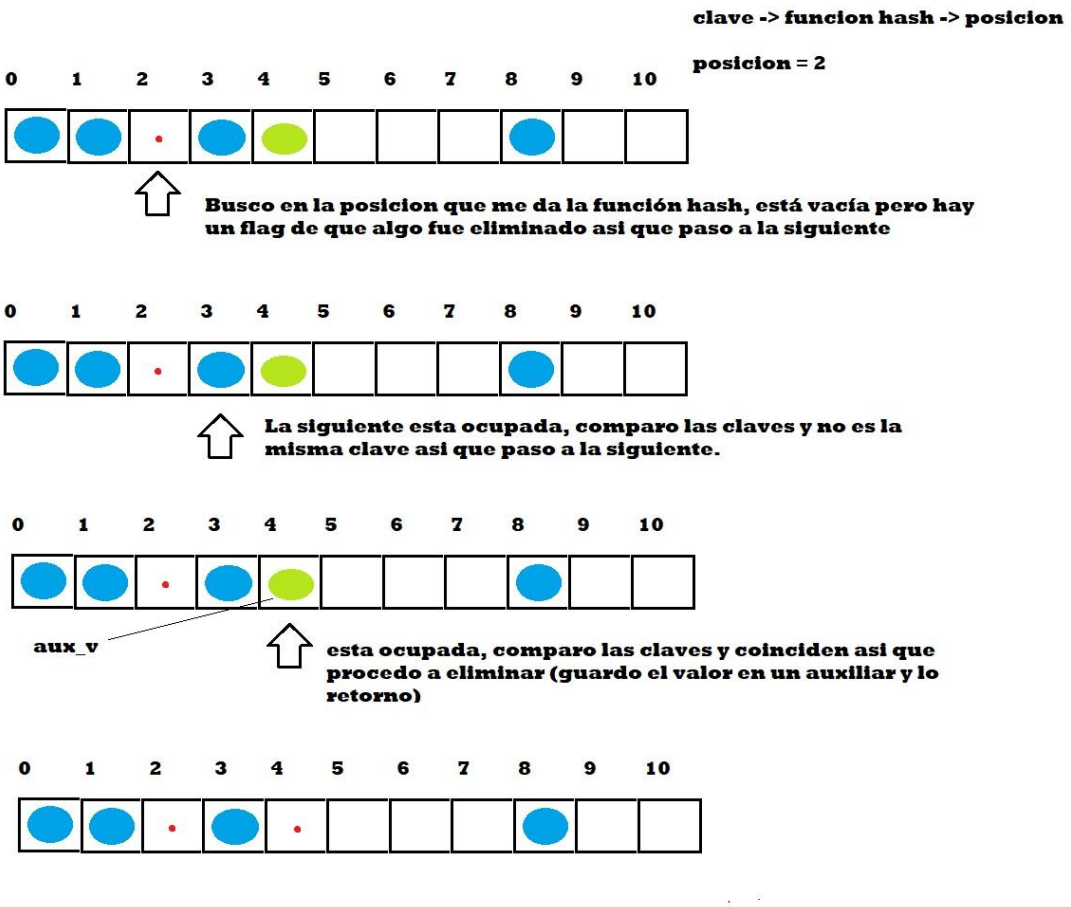
2.2) Primitivas de un HASH cerrado

- **Insertar:** Se obtiene la posición con la función hash, se busca la posición en el vector, pueden ocurrir las mismas 3 situaciones pero se procede de distinta manera al ser de direccionamiento abierto: ¿está vacía la posición? se inserta, ¿existe un par clave-valor? Se comparan las claves, si son iguales se reemplaza el elemento con el elemento nuevo, si son distintas se pasa la siguiente posición del vector y verifica lo mismo, esto se va a repetir las veces necesarias para poder insertar.

  Insertar implica: reservar memoria para la clave, asignarle la copia de la clave y el elemento a la estructura del vector. Se aumenta 1 a la cantidad de elementos.



- Eliminar: Se obtiene la posición con la función hash, se busca la posición en el vector, nos encontramos con las mismas 3 situaciones pero cabe aclarar que a la hora de eliminar se deja una flag de que algo se eliminó. Entonces al encontrar la posición se revisa si es la clave buscada, en el caso de serlo se elimina y se deja el flag en true; si no es la misma clave se pasa a la siguiente posición, ya que pudo haber una colisión al insertarla. Este proceso se repite hasta encontrar la clave buscada o hasta encontrar un lugar vacío en el vector sin el flag de eliminado, ya que si hay un lugar vacío sin que hubiese algo antes, significa que el par clave valor estuvo en el primer lugar con el flag encontrado, o nunca se insertó, ya que al colisionar se hubiese buscado la siguiente posición libre, entonces no hay posibilidad de que exista dicha clave buscada con el espacio vacío encontrado.



- Buscar: Es el mismo proceso de eliminar solo que sin “eliminar”, si se encuentra la clave buscada se retorna el valor, en caso de no encontrarla se sigue buscando en las siguientes posiciones hasta encontrarla o encontrar un espacio vacío sin flag.

Para ambos casos, HASH abierto o cerrado, se tiene que realizar el rehash, que es el agrandar la tabla o vector, para ello se establece un porcentaje de la tabla que debe estar lleno para saber cuándo realizarlo, entonces antes de insertar un elemento (preferiblemente) se revisa si sumando 1 a la cantidad de pares guardados se sobrepasa el porcentaje establecido, de ser así se hace el rehash y luego se inserta, sino solamente se inserta. El rehash se puede hacer de varias maneras, se puede agrandar solo la tabla o se puede crear un hash nuevo. De cualquier forma se tiene que recorrer la tabla anterior e ir almacenando los pares en la nueva tabla, usando la función hash y la capacidad de la nueva tabla, ya que al usar la capacidad en la división, si esta cambia la posición puede cambiar también.

3. Detalles de implementación

Se definieron 2 estructuras para la implementación, la estructura nodo, que contiene 3 punteros, uno para el nodo siguiente, uno para la clave y el último para el valor. La segunda estructura es la del hash, contiene 2 int para almacenar guardar la cantidad de pares y la capacidad, y un vector de punteros a nodos. El recorrido de las posiciones del vector del hash se hizo de forma iterativa, ya que era simplemente recorrer un vector común. Y a la hora de recorrer los nodos pertenecientes a una posición se usaron funciones recursivas ya que eran iguales a las del tda lista. En el caso del rehash si se recorrieron los nodos del vector antiguo de manera iterativa con un while para que sea más simple, pero se usó insertar_nodo con el nuevo vector en cada iteración.

3.1) Insertar:

Para la inserción se utilizó una función auxiliar llamada insertar_nodo, la cual es recursiva, va recorriendo los nodos pertenecientes a una posición del vector (se forma una lista de nodos), es la encargada de reservar memoria y verificar que haya salido correctamente y asignar el valor y la clave al nodo creado y el string creado. Para el rehash se creó una función llamada rehash, lo que hace es crear un nuevo vector de punteros a nodos con la nueva capacidad, luego recorre el antiguo vector de forma iterativa, a medida que va recorriendo el vector y las listas formadas en cada posición, hace uso de la función insertar_nodo en el nuevo vector, luego de insertar, libera la memoria del nodo que está en el antiguo vector (la de la clave y la del nodo). Cuando termina de recorrer el vector, el vector está vacío porque se fueron eliminando las estructuras, así que solo resta liberar la memoria del antiguo vector y asignarle el nuevo vector al hash, se actualiza la capacidad del vector y se retorna el hash actualizado.

3.2) Eliminar:

Para eliminar el nodo se creó una función llamada eliminar_nodo, la cual es recursiva, va comparando la clave dada con la clave del nodo actual, si son iguales, guarda en un auxiliar el nodo siguiente, libera la memoria reservada por el nodo a eliminar, retorna el siguiente que estaba en el auxiliar. En el caso de no coincidir las claves sigue buscando pero si encuentra un NULL sale de la función ya que no existe el nodo con la clave buscada. En el caso de eliminar satisfactoriamente,

cambia el valor de un flag pasado por referencia, esto hace que la función quitar disminuya en 1 el tope del hash.

3.3) Destruir y destruir todo:

La función destruir todo recorrer el vector de forma iterativa, se el destructor no es null, aplica el destructor al elemento, luego libera la memoria reservada por la clave y el nodo, finalmente libera la memoria reservada por el vector y luego por el hash. La función destruir usa la función destruir todo con un destructor NULL.