FACULTAD
DE INGENIERIA
Universidad de Buenos Aires

TDA N°3 — Tabla Hash

[7541/9515] Algoritmos y Programación II

Primer cuatrimestre de 2023

Alumno:	Salluzzi, Luca
Número de padrón:	108088
Email:	lsalluzzi@fi.uba.ar

1. Introducción

En este trabajo se buscaba afianzar en el alumno los conocimientos sobre tablas hash. Para esto, se le pidió implementar una tabla hash abierta con primitivas otorgadas por la cátedra.

2. Teoría, tipos de tablas hash

Un hash es una implementación de un TDA diccionario. Este está compuesto por una tabla en la cual se asignan elementos según su clave. Para encontrar su posición en la tabla, se pasa la clave por una función hash la cual devuelve un número. Al aplicar la funcion módulo entre ese número obtenido y la capacidad de nuestra tabla, obtenemos la posición a la que asignaremos el elemento.
Si dos elementos son asignados a la misma posición, se genera una colisión, la forma de resolverla depende del tipo de hash a utilizar.

Hash abierto

En un hash abierto, los elementos se guardan fuera de la estructura original (véase, por ejemplo, en una lista). Las colisiones se resuelven concatenando los elementos que se encuentran en la misma posición dentro de otra estructura. De esta forma, nos ahorramos el tener que reasignarlos a una nueva posición.
El tipo de direccionamiento de este hash es cerrado, ya que la posición que se le asigne al elemento en su primer hasheo, se mantendrá siempre y no variará.

Su complejidad en el peor caso es O(n), ya que puede terminar degenerándose a lista si no se lo rehashea al alcanzar una cantidad específica de elementos.
Sus complejidades son:

crear	Destruir	Agregar un elemento	Quitar un elemento	Buscar un elemento
O(1)	O(n)	O(1)	O(n)	O(n)

A la hora de agregar, simplemente se hashea la clave para buscar la posición y luego se carga el par <clave;valor> a la última posición de la lista correspondiente. Siempre el proceso es el mismo, ya que, utilizando el puntero al último par no es necesario recorrer todos los elementos de la lista.
Luego, para quitar o buscar, la justificación es la misma, puesto que, en el peor caso, debemos recorrer los elementos de la lista de la posición correspondiente hasta encontrar el par buscado o el par a eliminar (pudiéndose este encontrarse al final).
Finalmente, a la hora de destruir el hash, dependemos de la cantidad de elementos en el mismo, por lo que la complejidad será O(n).

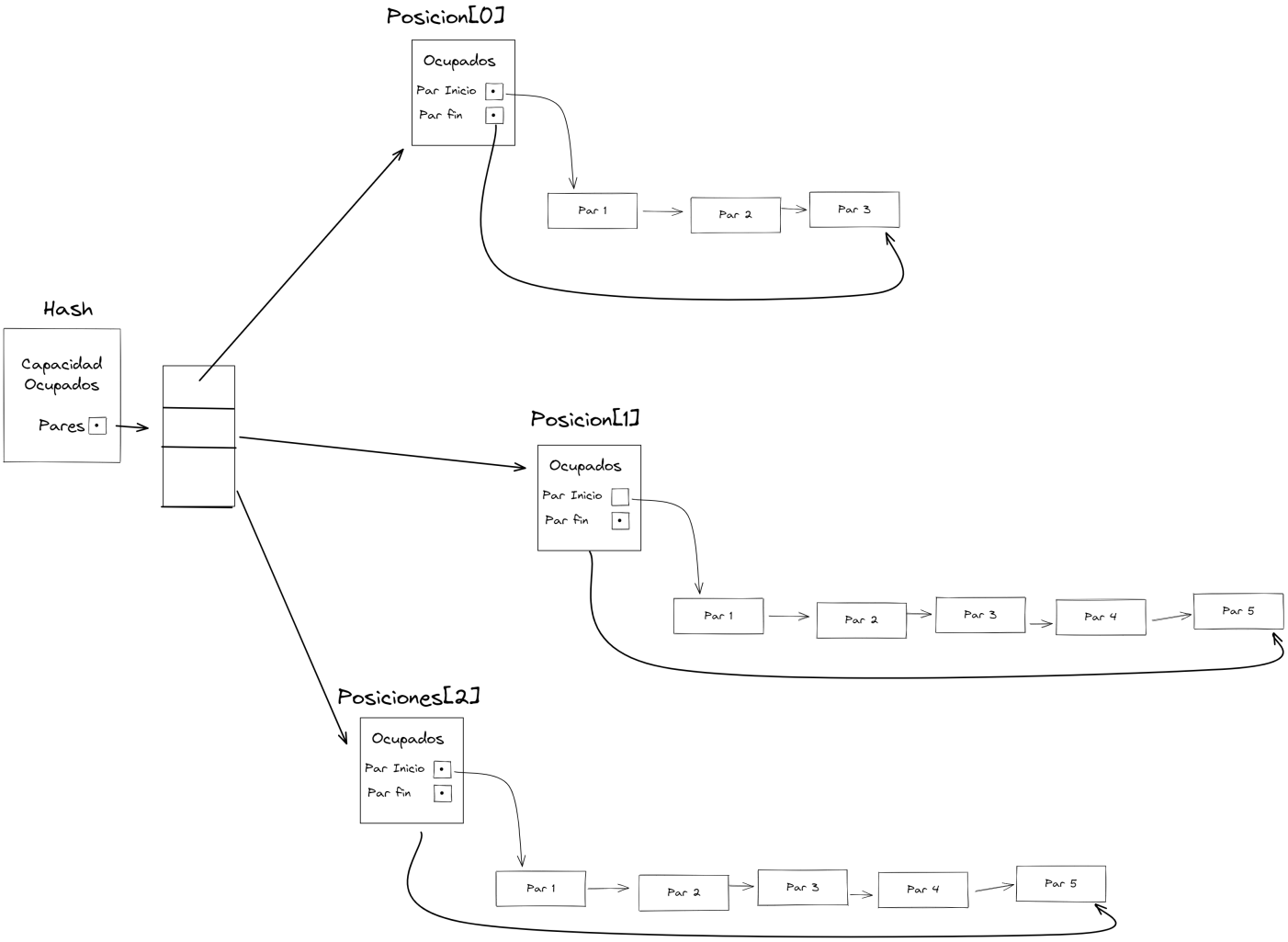

Diagrama de un hash abierto claramente degenerado (según mi implementación)

Hash cerrado

Un hash cerrado siempre guarda sus elementos dentro de su estructura original. A la hora de resolver colisiones, se buscan nuevas posiciones libres a las que asignar los elementos colisionados. De esta forma, siempre vamos a tener un tamaño de tabla mayor o igual al número de claves. Es por este modo de reasignar colisiones que este es

un hash de direccionamiento abierto, la posición del elemento puede variar si se encuentra colisionado.
Para buscar nuevas posiciones libres a la hora de redireccionar las colisiones, se pueden utilizar diferentes métodos:

- Probing lineal: Se trata de buscar el próximo espacio libre inmediato.
- Probing cuadrático: Busca el espacio libre inmediato, de no encontrarlo, busca la próxima posición libre tomando en cuenta estadísticamente la posición actual y la posición a la que se quiere llegar.
- Hash doble: aplicar por segunda vez la misma funcion hash buscando que nos devuelva una posición no ocupada. Esto puede llevar, en hashes ya muy cargados, a volver a colisionar y tener que repetir el proceso hasta encontrar una posición libre.

Sus complejidades son:

crear	Destruir	Agregar un elemento	Quitar un elemento	Buscar un elemento
O(1)	O(1)	O(n)	O(n)	O(n)

En un hash cerrado en el peor escenario, para agregar deberíamos recorrer todo el hash. Suponiendo que la posicion dada por la función hash sea la primera y la única posición libre, la última.
Luego, a la hora de quitar y buscar, al igual que en el hash abierto, vamos a tener que recorrer todo el hash hasta encontrarlo (tomando siempre el peor caso).

A la hora de quitar, es importante reordenar los elementos de modo que no haya espacios vacíos entre dos elementos contiguos (es decir, que hayan colisionado anteriormente y hayan sido desplazados al próximo espacio libre), porque la búsqueda depende de esas condiciones iniciales. Adicionalmente, se pueden definir flags que le indiquen al programa que en una posición actualmente libre hubo antes un elemento (ahora eliminado). De lo contrario, al buscar un elemento, nos dirigiremos a la posición que el hasheo nos devuelva, avanzaremos hasta un espacio vacío haciendo probing lineal y pensaremos que el elemento no se encuentra en el hash, cuando en realidad se encuentra cierta cantidad de espacios después pero no fue debidamente reordenado.

Factor de carga

Para saber la relación entre la cantidad de espacio disponible en el hash y el número de elementos insertados se utiliza un factor de carga. Este se calcula como número de claves almacenadas sobre capacidad del hash. Al superar un factor de carga establecido, se deberá rehashear para no afectar negativamente la complejidad de la estructura. Este siempre se va a encontrar entre 0 (hash vacío) y 1 (hash lleno).

3. Detalles de implementación

A la hora de correr el ejemplo se puede utilizar el comando `make valgrind-pruebas` y, para el ejemplo, similarmente `make valgrind-ejemplo`.
Para mi implementación, decidí emplear listas simplemente enlazadas para concatenar los elementos colisionados. De esta manera, no fue complicada la búsqueda ni la eliminación de elementos, ya que era muy similar a la del TDA Lista entregado anteriormente.
Como estructuras declaré un hash con un vector de listas, una cantidad máxima de elementos posibles a almacenar y una cantidad de ocupados (haciendo referencia a los elementos actuales existentes en el hash) esto me permitió tener un control mayor de la cantidad de elementos que se almacenaban en mi estructura.
Para cada lista, utilicé un puntero al primer y al último elemento de estas, así como un contador de posiciones ocupadas. Estas listas contenían como elementos los pares <clave;valor>, es decir, estructuras con un puntero a la clave y un puntero al valor, además de un puntero al siguiente par.
Como factor de carga máximo decidí elegir 0.65, de esta forma el hash nunca superaría (idealmente) los 2 tercios de su capacidad.

En cuanto a la modularización, me decidí por "extraer" de las primitivas originales aquellas funciones o procedimientos que podían ser considerados casos bordes o especiales. En `hash_insertar`, por ejemplo, se llama a `sobreescribir_elemento` para, de ser necesario, actualizar en una clave ya existente el elemento por uno nuevo ingresado. No me pareció propicio, entonces, modularizar el resto de la inserción (llenado del par y cargado del mismo al final de la lista de la posición correspondiente), puesto a que me parecía que esas breves líneas de código eran el tronco de la primitiva. Decidí también no utilizar hash_contiene para verificar si la sobreescripcion era necesaria, ya que esto conllevaría una doble iteración a lo largo de la lista y, por lo tanto, mayor complejidad algorítmica.

Otro ejemplo de este lineamiento en la modularización puede verse en `hash_quitar`, donde modularizo el caso de quitar un elemento en una lista de cantidad 1 y mantengo dentro de la primitiva la eliminación en una de cantidad mayor, iterando a lo largo de los pares de la lista hasta encontrarlo y eliminarlo. Considerando ese procedimiento como el caso base de la primitiva.

Finalmente, con respecto a mis pruebas, decidí implementar una pequeña parte de ellas de caja blanca para poder comprobar el correcto funcionamiento de la inserción y del rehash. Quizás vistas ahora parezcan un poco redundantes, pero teniendo en cuenta que fueron usadas en el tiempo de desarrollo del TDA, fueron de mucha utilidad para comprobar minuciosamente que las funciones cumplían con su cometido.

5. Diagramas

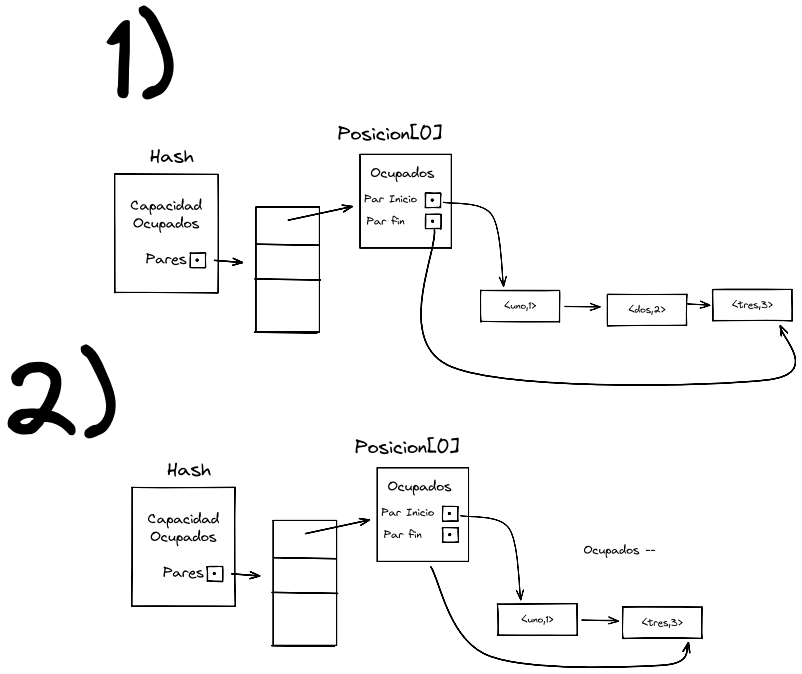

Diagrama de quitado en una hash abierto

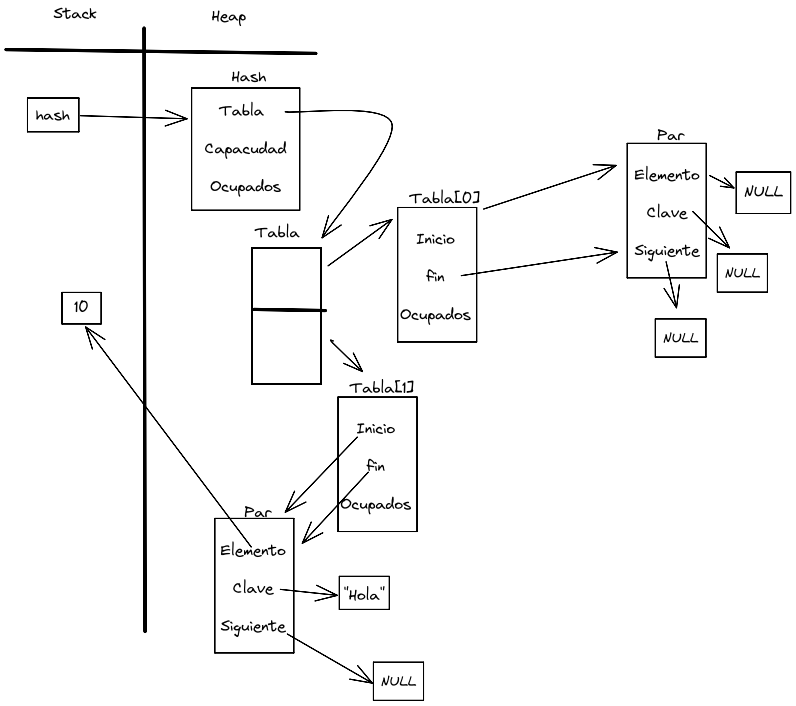

Diagrama de memoria de mi implementacion de hash