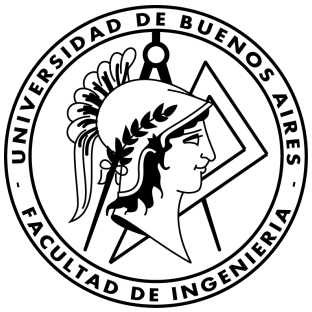


TDA Hash

Algoritmos y Programación II
Segundo cuatrimestre de 2022

Alumno:	Fernández, Facundo Nahuel
Número de padrón:	109097
Email:	ffernandez@fi.uba.ar

Índice

1. Introducción **2**

2. Teoría **2**
2.1. Diccionario 2
2.2. Función y tablas de hash 2

1. Introducción

El objetivo de este trabajo es implementar un hash, creando un iterador interno para dicho hash.

Para compilar el programa (el archivo pruebas.c), se debe acceder desde la terminal a la carpeta en donde se encuentra dicho archivo y escribir el comando ´make´.

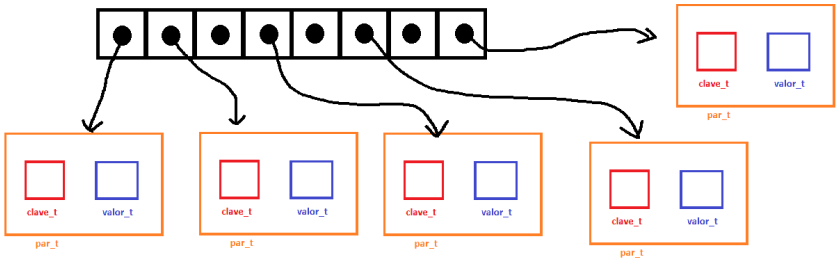
En este informe se desarrolla los conceptos de diccionario, de función de hash y de tabla de hash. Además, se habla sobre los distintos métodos de resolución de colisiones para la tabla de hash.

2. Teoría

2.1. Diccionario

Un diccionario es un tipo de dato abstracto que es una colección de pares clave/valor. La utilidad del diccionario es poder encontrar un valor a partir del conocimiento de la clave asociada a ese valor. En un diccionario se debería poder insertar o quitar una clave (y su valor), además de buscarla.

Hay diferentes formas de implementar un diccionario. Un ejemplo puede ser el de la implementación de un diccionario con un vector, en donde en cada posición de este se apunta a un par que contiene una clave y un valor:



Esta implementación no es incorrecta pero sí puede traer problemas: para buscar una clave en el vector debemos recorrerlo o hacer una búsqueda binaria. Lo ideal sería poder encontrarla directamente para reducir la complejidad de esta acción.

Se podría también pensar en una implementación con listas o árboles binarios de búsqueda. Esto tampoco es incorrecto, pero el caso de una lista tendremos que recorrerla toda para hallar una clave (complejidad O(n) si n es la cantidad de pares) y para un abb sucederá algo similar ya que tendremos que buscar dentro del árbol (Comlejidad O(log n) o complejidad O(n) si el abb degeneró a lista).

Lo mejor para implementar un diccionario serán las tablas de hash, de lo que se hablará a contunuación.

2.2. Función y tablas de hash

Para hablar sobre tablas de hash, en primer lugar se debe definir qué es una función de hash: es una función H(k) que recibe una clave k de cualquier tipo y devuelve un número. Esta función nos ayuda a facilitar la búsqueda de una clave dentro de un vector: si conocemos el valor de H(k) y este valor es utilizado para insertar un par clave/valor dentro del vector, será mucho más fácil encontrarlo ya que podremos ir directamente a la posición que corresponda a la clave sin tener que recorrerlo para buscarla.

Una tabla de hash es una estructura que contiene pares clave/valor, en donde se puede hallar un valor a partir de la clave asociada a él, como era en el caso del diccionario. La diferencia reside en que la posición que tendrá una clave en una tabla de hash estará determinada por una función de hash H(k). Si se implementa una tabla de hash con un vector dinámico y este tiene tamaño N, una clave k se ubicará en la posición H(k) %N. Ahora, para buscar dentro de una tabla, podemos ir directamente a esta posición del vector y hallaremos la clave sin recorrerla.

Una función H(k) debe tener las siguientes características:

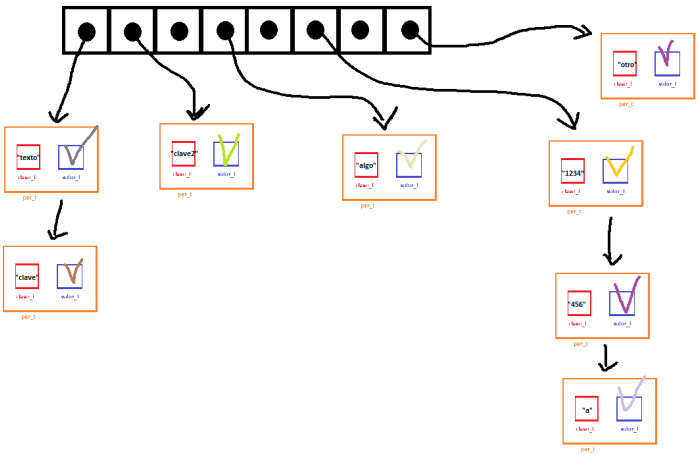
- Ser consistente, es decir, ante una clave k debe devolver siempre el mismo número.
- Ser rápida, no queremos que una función de hash tarde mucho en darnos un valor.
- Espaciar los resultados lo más posible.

  Con el uso de una función de hash resolvimos el problema de la complejidad en la búsqueda, pero se nos presenta el siguiente: puede haber dos claves k1 y k2 con k1 != k2 tal que H(k1) %N == H(k2) %N, lo que causaría que haya dos claves en una misma posición de la tabla. A esto se lo denomina colisión. Se hablará más adelante sobre qué hacer ante colisiones, pero se espera que la cantidad de colisiones sea lo mínima posible (que haya una distribución de colisiones uniforme) y la función de hash debe asegurarnos eso.

  Existe el hashing perfecto, en donde la función de hash es inyectiva y asigna a cada clave (dentro de un espacio de claves limitado) un número distinto, que permiter que no existan colisiones.

Existen dos tipos de tablas de hash:

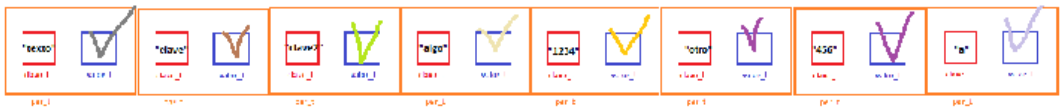
En primer lugar, tenemos el **hash abierto** o de **direccionamiento cerrado**. Se le conoce como hash abierto puesto que los pares clave/valor se almacenan por fuera de la tabla y de direccionamiento cerrado ya que la posición en la que se encuentra una clave h será H(k) %N sí o sí. La figura utilizada para ilustrar un diccionario también vale para un hash abierto (sin colisiones). En este tipo de hash, se resuelven las colisiones encadenándolas. Si tenemos una clave que debe ubicarse en la misma posición que otra, se inserta el nuevo para encadenado al par de la clave que teníamos en la posición, como si se tuviese una lista de pares en cada posición del hash. Para quitar un par clave/valor, simplemente se busca su ubicación y se elimina. Si este par estaba en una lista y apuntaba a otro, se hace que el anterior apunte a este otro (como cuando se eliminaba un nodo de una lista enlazada).



En la figura se ve un hash en donde hubo una colisión en la primera y dos en la sexta. Por último, si se quiere insertar un par clave/valor que ya se encuentra en el hash, se busca la clave dentro del hash y se reemplaza el valor que tenía por el nuevo, sin una colisión (podría insertarse una clave repetida y estaríamos hablando de un multidiccionario, pero eso no fue implementado).

Hablando sobre comlpejidades, si no existieran las colisiones la complejidad de la búsqueda sería O(1), ya que iríamos directamente a la posición en la que se encuentra una clave y obtendríamos su valor. Debido a las colisiones, en cada posición puede haber una lista y, para hallar el valor de una clave, se debe recorrerla, lo que hará que la complejidad sea algo parecido a O(n) si pensamos en el peor caso, en el que todas las claves fueron a parar a una misma posición (la función de hash es muy mala en este caso).

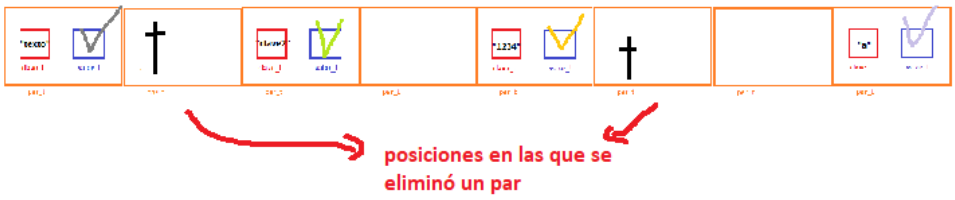
En segundo lugar, tenemos el **hash cerrado** o de **direccionamiento abierto**. Es cerrado porque los pares se almacenan dentro de la tabla, y de direccionamiento abierto porque la posición en la que se guarda una clave k no es siempre H(k) %N, sino que puede variar al tener colisiones.



En la figura se observa una posible forma de hash cerrado, en donde se fueron insertando pares clave/valor hasta llenarse. La posición en la que se encuentra cada clave k_i no es necesariamente H(k_i) %N, sino que fue colisionando. A simple vista se observa un problema: buscar una clave implicará recorrer la tabla ya que no conocemos exactamente donde se encuentra y la tabla está llena (algo que no debe suceder nunca, ni en el hash cerrado ni en el abierto, y se hablará después sobre cómo solucionarlo), por lo que la complejidad no será O(1) sino que se aproximará a O(n) en este caso.

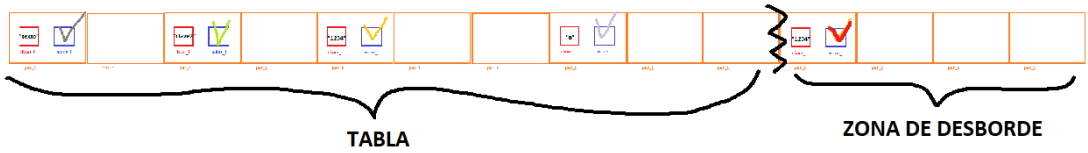
Para resolver colisiones, estas ya no se encadenan, sino que se puede resolver de distintas formas.

Una de ellas es el probing lineal: Si estamos ante una colisión, en vez de insertar en la posición que corresponde, se inserta en la siguiente que no esté ocupada, recorriendo hacia adelante (se considera la tabla como circular; si se llega al final se sigue recorriendo desde el primer lugar como si fuera el siguiente al último). De este modo, cuando buscamos una clave, vamos a la posición que le correspondería desde un inicio y, si no se encuentra allí, recorremos hacia adelante hasta encontrarla. Se puede presentar un problema al eliminar pares: si se elimina uno y busco una clave a la que le correspondía una posición anterior a esta y se encuentra en una posterior (a causa de diversas colisiones), voy a llegar a una posición vacía y se podría concluir que la clave buscada no se encuentra en la tabla (si se insertó colisionando, debería hallarse antes de una vacía). Para solucionar esto, en las posiciones en las que se elimina se ubica una 'marca' para saber que allí hubo un par.



Otra forma de resolución de colisiones es la zona de desborde: consiste en tener una tabla de hash con un tamaño especificado y una zona de desborde que tendrá como tamaño un porcentaje

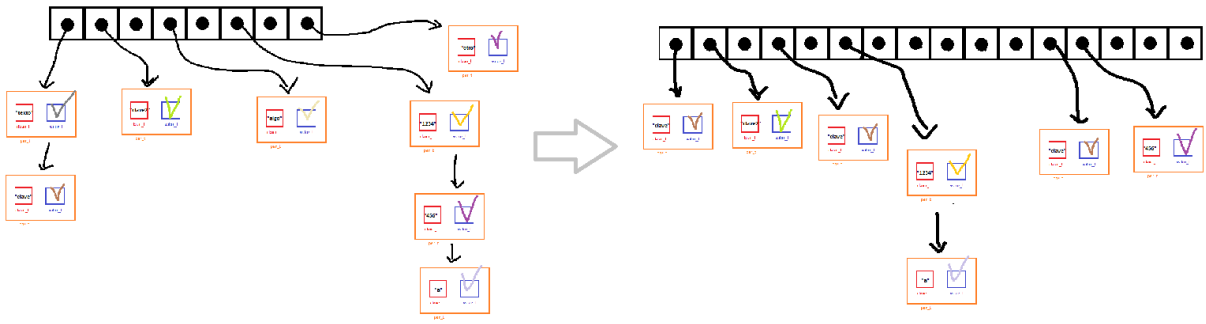
del tamaño de la tabla. Ante una colisión, se ubica al par en la zona de desborde, desde la primera posición de la zona (si la primera está ocupada, se inserta en la segunda, y así).



En la figura se observa la zona de desborde (hubo una clave que colisionó). Para este método no hace falta dejar marcas al eliminar de la tabla, ya que las colisiones se van a la zona por fuera de la tabla (si se quita un elemento de la zona, se deberían reacomodar los que estén por delante). La complejidad de la búsqueda será O(1) para los pares que no hayan colisionado, pero aumentará en la zona de desborde pues debemos recorrerla para encontrar claves allí.

Existen otros métodos para resolver colisiones, como el doble hashing (se usan dos funciones de hash; cuando con una se presenta una colisión, se utiliza la otra para evitarla) o el probing cuadrático (similar al probing lineal pero no se inserta en la posición siguiente sino en otra).

Ahora bien, nos queda hablar sobre la solución al mayor problema que nos encontramos en los dos tipos de tabla: cuando la tabla abierta tiene muchos elemento, se empiezan a formar listas cada vez más grandes en cada posición; y cuando la tabla cerrada alcanza cierta capacidad, los pares quedan en posiciones muy lejanas a las que les correspondía originalmente y esto aumenta la complejidad; o con el método de zona de desborde se nos llena esta zona al haber muchos pares (aumentan las colisiones). Para solucionar estos problemas, se hace un **rehash**. El rehash consiste en aumentar la capacidad de la tabla cuando se llena cierta capacidad de la misma (en general y en la implementación entregada en este trabajo, el 75 %) y reubicar los pares en la tabla, teniendo en cuenta que ahora N (la capacidada de la tabla) es distinto. En los ejemplos de hash hechos en las figuras anteriores (a excepción de la figura con zona de desborde), los pares en la tabla superaban el 75 % de la capacidad de la misma, cosa que no debería suceder, ya que causan que la complejidad de las operaciones de búsqueda, inserción y eliminación aumente y se parezcan a O(n).



En la figura se observa un rehash hecho al hash de la segunda imagen del documento (debería haberse hecho antes ya que nos habíamos excedido del 75 % de la capacidad, pero a fines ilustrativos sirve), en donde se aumentó la capacidad al doble y se reubicó los pares. Puede observarse que hay muchas menos chances de que se produzca una colisión y la mayoría de los pares que estaban encadenados ya no lo están, lo que causa que la complejidad vuelva a ser muy cercana a O(1) cuando querramos hallar el valor que está asociado a una clave.

Para el caso de hash cerrado será similar, se aumentará el tamaño de la tabla (y de la zona de desborde si se utilizó esa implementación), se ubicará de nuevo los elementos y ya no habrá marcas de borrado si se utilizó el probing.