

# Estructuras de Datos

---

## Hash

Supongamos que necesitamos almacenar  $n$  números enteros, sabiendo de antemano que dichos números se encuentran en un rango conocido por ejemplo:

$$0, \dots, k-1$$

Para resolver este problema, basta con crear un arreglo de tamaño  $k$  y marcar con valor `true` o `1` o lo que sea, los casilleros del arreglo cuyo índice sea igual al valor del elemento a almacenar. De esta forma determinamos que el elemento está presente.

Un ejemplo de esto sería:

Elementos = {1, 4, 7}

0	1	2	3	4	5	6	7
false	true	false	false	true	false	false	true

Podemos ver que con esta estructura de datos el costo de búsqueda, inserción y eliminación es  $O(1)$ .

Este enfoque tiene dos grandes problemas, si pensamos en el hecho que puedo replicar esta idea para cualquier tipo de dato que se quiera almacenar:

1. El valor de  $k$  puede ser muy grande, y por lo tanto no habría lugar en memoria para almacenar el arreglo completo.

Pensemos, por ejemplo, en todos los posibles nombres de personas.

2. Los datos a almacenar pueden ser pocos, con lo cual se estaría desperdiciando espacio de memoria ya que estaríamos pidiendo lugar para todos los datos posibles.



## Definiciones - Función de Hash

---

Nuestro objetivo es poder construir un intervalo de números menor. Para esto vamos a usar una función  $h$ , denominada *función de hash*.

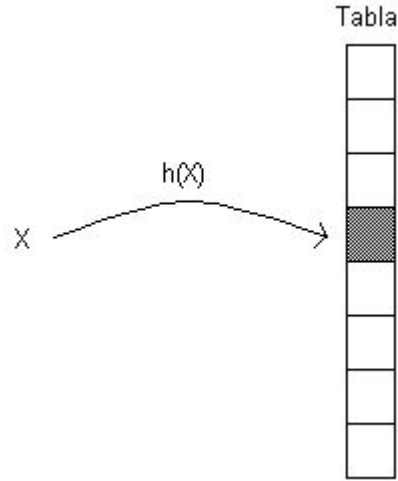
Esta función es tal que: dado un elemento  $X$  perteneciente a nuestro universo de datos esperados, en nuestro ejemplo sería un número en el rango  $[0, \dots, k-1]$ , su valor de retorno, es decir  $h(X)$ , es un número en el rango  $[0, \dots, m-1]$ . Siendo  $m \ll k$  ( $m$  es significativamente más chico que  $k$ ).

En este caso, se marca el casillero, cuyo índice es  $h(X)$ , para indicar que el elemento  $X$  pertenece al conjunto de elementos dados.



## Definiciones - Función de Hash

Fíjense que estamos transformando nuestro espacio de valores posibles de  $[0, \dots, k-1]$  a  $[0, \dots, m-1]$ , siendo  $m \ll k$  ( $m$  es significativamente más chico que  $k$ ).  
Esta estructura de datos es conocida como *tabla hash*.



Al  $h(X)$  se lo llama clave o key.



## Definiciones - Función de Hash

---

La función  $h$  debe distribuir los valores lo más uniformemente posible dentro de la tabla, es decir, tendría que ser igualmente probable que salga  $0, 1, 2, \dots$  o  $m-1$ .

Dado que tenemos  $m$  posibles valores, la probabilidad de que la clave obtenida sea  $z$  (donde  $z$  está en  $[0, \dots, m-1]$ ) a partir de un  $X$  es  $1/m$ . Esto lo notamos como:

$$\Pr(h(X)=z) = 1/m \text{ para todo } z \text{ en } [0, \dots, m-1].$$

## Definiciones - Función de Hash

---

En general, el valor  $X$  se puede interpretar como un número entero, y las funciones  $h(X)$  genéricamente son de la forma:

$$h(X) = (c \cdot X \bmod p) \bmod m$$

donde  $c$  es una constante,  $p$  es un número primo y  $m$  es el tamaño de la tabla de hashing.

Distintos valores para estos parámetros producen distintas funciones de hash.

## Definiciones - Colisiones

---

El problema que tiene este enfoque es que dos elementos pueden tener la misma clave, es decir, siendo  $X1$  distinto de  $X2$ ,  $h(X1) = h(X2)$ .

A este problema se lo denomina Colisiones.

Para ilustrar esto veamos el siguiente ejemplo.

¿Cuál es el número  $n$  mínimo de personas que es necesario reunir en una sala para que la probabilidad que dos de ella tengan su cumpleaños en el mismo día sea mayor que  $1/2$ ?

Hagamos el siguiente razonamiento:

- La primera persona que consideremos puede cumplir años cualquier día, es decir, tiene 365 posibilidades.
- La segunda tiene 365 días menos el del cumpleaños de la persona anterior, es decir:  $365-1$  posibilidades.
- La tercer persona tiene 365 días menos los días de las 2 personas anteriores, es decir,  $365-2$ .
- La  $k$ -ésima persona tiene 365 días menos los días de las  $k-1$  personas anteriores, es decir,  $365-(k-1)$  o, lo que es lo mismo,  $365-k+1$ .

## Definiciones - Colisiones

---

Es decir entonces que la cantidad de posibilidades para  $n$  personas sería:

$$365(365-1)(365-2)\dots(365-n+1)$$

Estas serían las posibles fechas de cumpleaños de  $n$  personas sin que coincidan en un día. La probabilidad de esto sería dividiendo este valor por los casos posibles, es decir, que cada persona cumpla en cualquiera de los 365 días. Si llamamos  $d_n$  a esta probabilidad entonces tenemos que:

$$d_n = \left(\frac{365}{365}\right)\left(\frac{364}{365}\right)\left(\frac{363}{365}\right)\dots\left(\frac{365-n+1}{365}\right)$$

## Definiciones - Colisiones

---

Responder nuestra pregunta: ¿Cuál es el número  $n$  mínimo de personas que es necesario reunir en una sala para que la probabilidad que dos de ella tengan su cumpleaños en el mismo día sea mayor que  $1/2$ ?

Es lo mismo que: ¿Cuál es el mínimo  $n$  tal que  $d_n < 1/2$ ?

Respuesta:  $n = 23 \Rightarrow d_n = 0.4927$ . Notemos que 23 es bastante más chico que 365.

Esto quiere decir que con una pequeña fracción del conjunto de elementos es posible tener colisiones con alta probabilidad.

En la próxima presentación veremos cómo tratar este problema.

# Estructuras de Datos

---

## Hash

## Definiciones - Colisiones

---

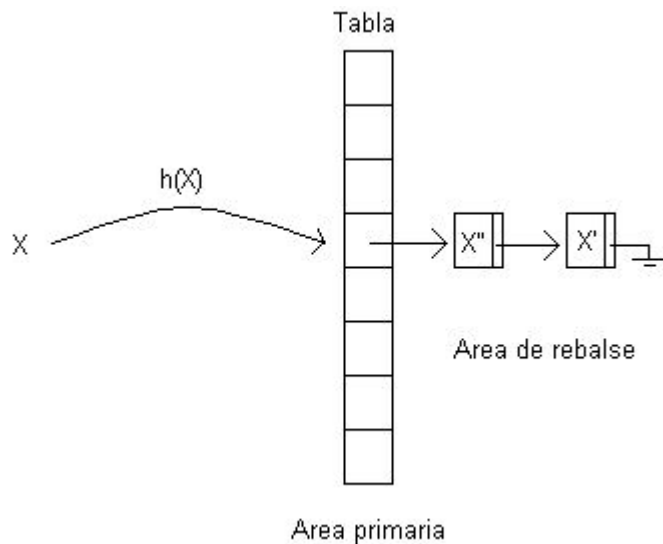
Para resolver el problema de las colisiones, existen dos grandes familias de métodos:

- Encadenamiento (usar estructuras dinámicas).
- Direccionamiento abierto (intentar con otra función de hash).

En esta presentación vamos a analizar el Encadenamiento.

# Colisiones - Encadenamiento

La idea de este método es que todos los elementos que caen en el mismo casillero se almacenen en una estructura auxiliar. En este ejemplo podemos ver que se usa una LSE en la cual se realiza una búsqueda secuencial.





## Colisiones - Encadenamiento - Hashing con listas mezcladas

---

Esta es una variante de Encadenamiento donde en lugar de usar un área de rebalse los elementos se almacenan en cualquier lugar libre del área primaria. Para poder saber dónde buscar se utiliza un puntero al siguiente elemento, es decir, construimos listas dentro de la tabla hash.

Veamos un ejemplo a continuación.



# Colisiones - Encadenamiento - Hashing con listas mezcladas

---

Supongamos que tenemos una tabla hash con  $m=10$ . Nuestra función de hash es simplemente:

$$h(X) = X \bmod 10$$

Usamos Hashing con listas mezcladas para resolver colisiones.



# Colisiones - Encadenamiento - Hashing con listas mezcladas

---

Inicialmente tenemos la tabla hash vacía.

Podemos ver que tenemos un espacio para guardar el dato y un espacio para el puntero al siguiente elemento.

0		
1		
2		
3		
4		
5		
6		
7		
8		
9		



# Colisiones - Encadenamiento - Hashing con listas mezcladas

Si almacenamos el 25, podemos ver que, usando la función hash, se ocuparía la clave 5 en la tabla por lo que tendríamos:

0		
1		
2		
3		
4		
5	25	→
6		
7		
8		
9		



## Colisiones - Encadenamiento - Hashing con listas mezcladas

Ahora se almacena el 39 usando la función hash, se ocupa la clave 9 en la tabla por lo que tendríamos:

0		
1		
2		
3		
4		
5	25	└─┐ └─┘
6		
7		
8		
9	39	└─┐ └─┘



# Colisiones - Encadenamiento - Hashing con listas mezcladas

Se guarda el 36, usando la función hash la clave es 6. La tabla queda:

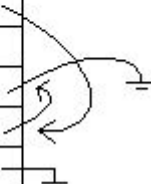
0		
1		
2		
3		
4		
5	25	└─┴─┘
6	36	└─┴─┘
7		
8		
9	39	└─┴─┘



# Colisiones - Encadenamiento - Hashing con listas mezcladas

Un tiempo después miramos la tabla y lo que vemos es:

0		
1		
2		
3		
4		
5	25	
6		
7	48	
8	15	
9	39	





# Colisiones - Encadenamiento - Hashing con listas mezcladas

Analizándola podemos concluir que:

- el 15 está almacenado en una clave que no le corresponde porque su clave estaba ocupada (25), seguramente en el momento de guardarla las claves sucesivas también lo estaban y, la primera que encontró libre fue la clave 8;
- el 48, al querer guardarse encontró su clave ocupada y, la primer clave que encontró libre fue la 7.

0		
1		
2		
3		
4		
5	25	
6		
7	48	
8	15	
9	39	

## Colisiones - Encadenamiento

---

Ahora bien, ¿cuándo conviene usar un área de rebalse y, cuándo usar listas mezcladas?

No hay una única respuesta sino que en realidad depende del problema pero, en general, esta variante es útil cuando la tabla no está densamente poblada.

Para tener una idea de este concepto vamos a definir factor de carga de una Tabla hash.

## Colisiones - Encadenamiento

---

El grado de ocupación de una tabla hash se determina mediante el factor de carga, que es la fracción ocupada de la tabla y es un número que está entre 0 y 1 si está vacía o llena respectivamente.

Dicho factor se obtiene dividiendo la cantidad de elementos almacenados sobre el total de espacio disponible. Es decir, si llamamos  $\alpha$  a dicho factor, hay  $n$  elementos almacenados en una tabla hash de tamaño  $m$  entonces:

$$\alpha = \frac{n}{m}$$

Se puede verificar que el costo de búsqueda sólo depende del factor de carga  $\alpha$ , y no del tamaño de la tabla.

## Colisiones - Encadenamiento - Eliminación

---

El caso de eliminar con un área de rebalse es sencillo: sólo hay que eliminar el elemento allí, por lo que la complejidad viene dada por el costo de eliminarla de la misma.

Pero, en el caso del hashing con las lista mezcladas, el algoritmo es más complejo, ya que debe reenlazar las listas. Pensemos en el ejemplo anterior: ¿qué pasaría si elimino el 15? En ese caso deberíamos reenlazar el 48 ya que de otra forma lo perderíamos.

En la siguiente presentación vamos a ver el otro enfoque: Direcccionamiento Abierto.

# Estructuras de Datos

---

## Hash

## Colisiones - Direcccionamiento Abierto

---

La idea de Direcccionamiento Abierto consiste en una sucesión de funciones hash, es decir,  $\{h_0, h_1, \dots, h_n\}$ .

Supongamos que tenemos al elemento  $X$ . Lo primero que se intenta es almacenar el dato aplicando la función de hash  $h_0$ , si  $\text{tabla}[h_0(X)]$  está ocupado se prueba con  $h_1$ , es decir me fijo si  $\text{tabla}[h_1(X)]$  está ocupado, y así sucesivamente.

Es decir, se resuelven las colisiones aplicando una función hash. Hay diferentes formas de generar esta sucesión de funciones, comenzamos con Linear probing.



## Colisiones - Direccionamiento Abierto - Linear probing

---

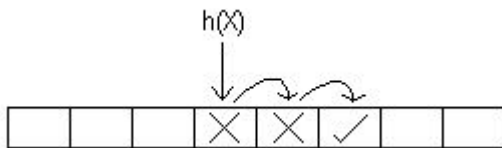
Es el método más simple de direccionamiento abierto, en donde las funciones de hash se definen como:

$$\begin{aligned}h_0(X) &= h(X) \\ h_{k+1}(X) &= (h_k(X) + 1) \bmod m\end{aligned}$$

siendo  $m$  el tamaño de la tabla.

## Colisiones - Direccionamiento Abierto - Linear probing

Si analizamos esta definición, podemos ver que simplemente va incrementando en 1 el resultado de la función anterior (módulo  $m$  para no irnos del tamaño de la tabla), es decir, si el casillero está ocupado, prueba con el siguiente hasta encontrar el primer casillero libre.



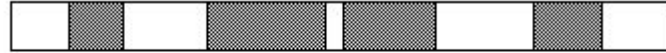
El problema que podemos marcar rápidamente es que cuando el factor de carga es alto este método resulta ser muy lento.



## Colisiones - Direccionamiento Abierto - Linear probing

Un punto que es importante analizar es que a medida que la tabla se va llenando, se observa que empiezan a aparecer *clusters* de casilleros ocupados consecutivos:

Tabla



casilleros ocupados



## Colisiones - Direccionamiento Abierto - Linear probing

---

Si la función de hash distribuye los elementos uniformemente dentro de la tabla, la probabilidad que un cluster crezca es proporcional a su tamaño.

Esto significa que una mala situación se vuelve cada vez peor con mayor probabilidad. Esto se conoce como *clustering primario*.

Sin embargo este no es todo el problema, puesto que lo mismo sucede en hashing con encadenamiento y no es tan malo. El verdadero problema ocurre cuando 2 clusters están separados solo por un casillero libre y ese casillero es ocupado por algún elemento: ambos clusters se unen en uno mucho más grande. Esto genera clusters que dificultan las operaciones.



## Colisiones - Direccionamiento Abierto - Linear probing

---

Otra característica, menos grave que la anterior, que se genera con Linear probing es conocida como *clustering secundario*.

El mismo consiste en lo siguiente: supongamos que al realizar la búsqueda de dos elementos en la tabla se encuentran con el mismo casillero ocupado, entonces toda la búsqueda subsiguiente es la misma para ambos elementos.



## Colisiones - Direccionamiento Abierto - Linear probing

---

Eliminar un elemento es complejo. ¿Por qué? Vamos a analizarlo.

Análogamente a lo que sucedía en Hashing con Listas Mezcladas donde debíamos reenlazar las listas al eliminar un elemento, acá no se puede eliminar un elemento y simplemente dejar su casillero vacío porque las búsquedas terminarían en dicho casillero.

¿Qué podemos hacer para resolver esto? Vamos a ver dos posibles soluciones.



## Colisiones - Direccionamiento Abierto - Linear probing

---

Existen dos maneras para eliminar elementos de la tabla en este caso:

- Marcar el casillero como "eliminado", pero sin liberar el espacio. Esto produce que las búsquedas puedan ser lentas incluso si el factor de carga de la tabla es pequeño.
- Eliminar el elemento, liberar el casillero y mover elementos dentro de la tabla hasta que un casillero "verdaderamente" libre sea encontrado. Implementar esta operación es complejo y costoso.



## Colisiones - Direccionamiento Abierto - Hashing Doble

---

Esta variante de Direccionamiento Abierto usa dos funciones hash:

1. una función conocida como *dirección inicial*  $h$ , tal que  $h(X) \in [0, \dots, m-1]$
2. y una función conocida como *paso*  $s$ , tal que  $s(X) \in [1, \dots, m-1]$

Con ellas construimos la secuencia de funciones de la siguiente forma:

$$\begin{aligned}h_0(X) &= h(X) \\ h_{k+1}(X) &= (h_k(X) + s(X)) \bmod m\end{aligned}$$



## Colisiones - Direccionamiento Abierto - Hashing Doble

---

Podemos ver que cuando  $s(X)=1$  para todo  $X$  tenemos Linear probing.

Elegir  $m$  primo asegura que se va a visitar toda la tabla antes que se empiecen a repetir los casilleros. Obervación: sólo basta que  $m$  y  $s(X)$  sean primos relativos.



## Colisiones - Direccionamiento Abierto - Hashing Doble

---

Existen heurísticas para resolver el problema de las colisiones en hashing con direccionamiento abierto, como por ejemplo *last-come-first-served* hashing (el elemento que se mueve de casillero no es el que se inserta sino el que ya lo ocupaba) y *Robin Hood* hashing (el elemento que se queda en el casillero es aquel que se encuentre más lejos de su posición original), que si bien mantienen el promedio de búsqueda con respecto al método original (*first-come-first-served*) disminuyen notablemente su varianza.