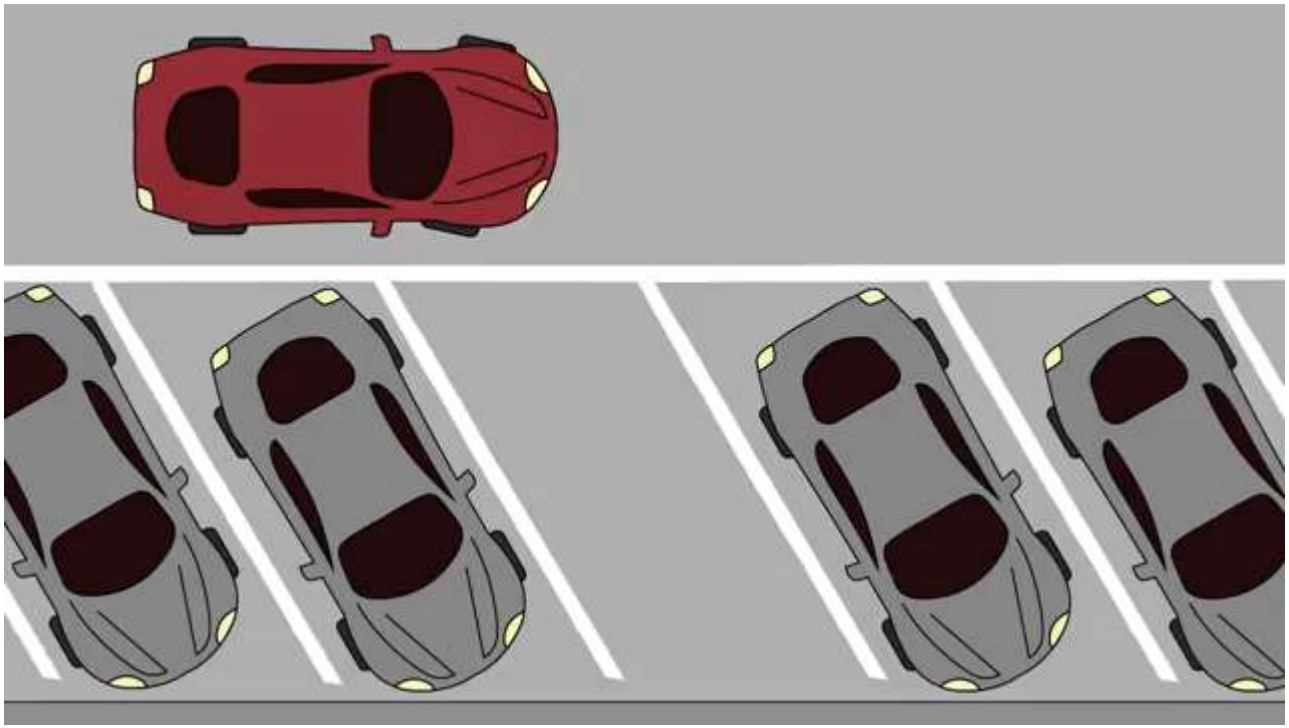
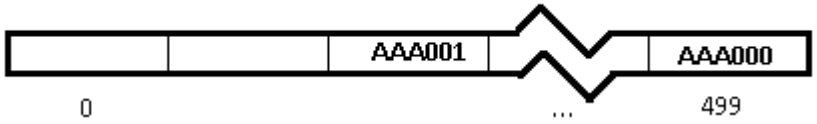
C++ HASHING

Hashing: Introducción

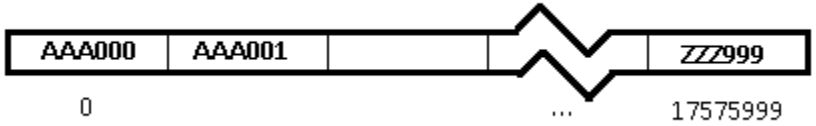


Hashing: Introducción

- Opción 1:



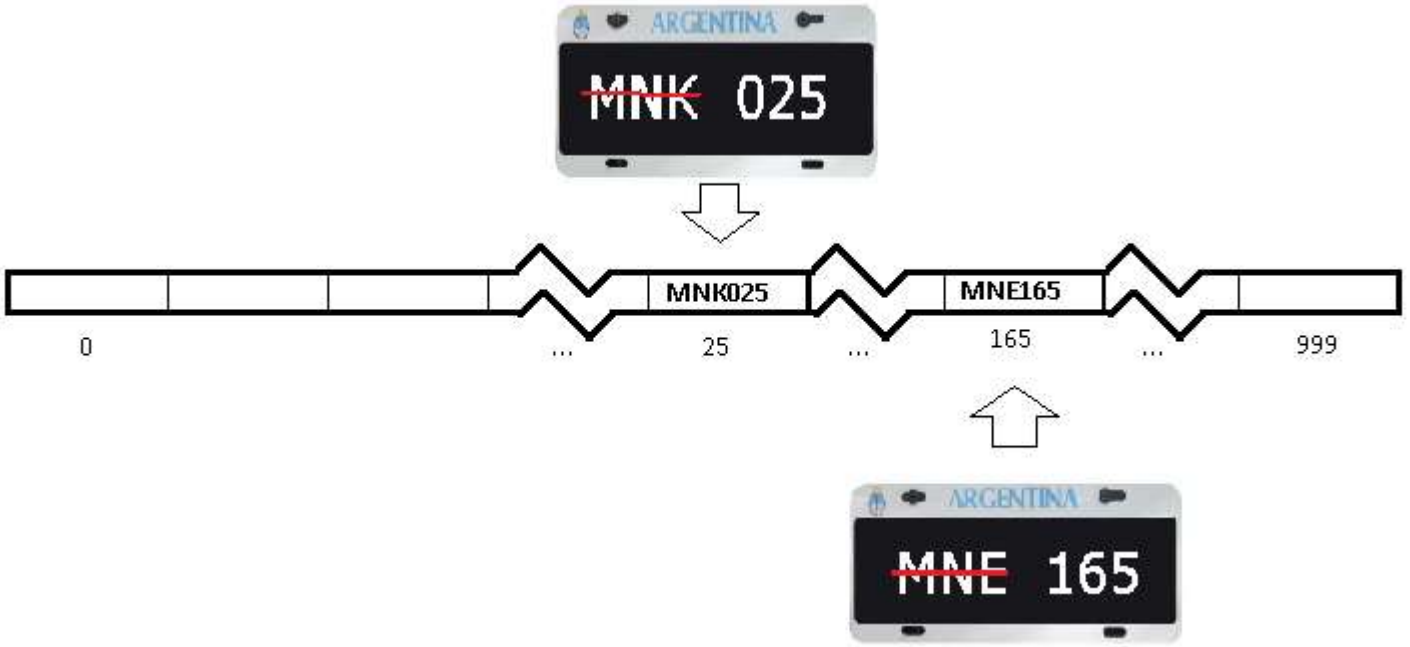
- Opción 2:



26x26x26x10x10x10 = 17576000 posiciones!

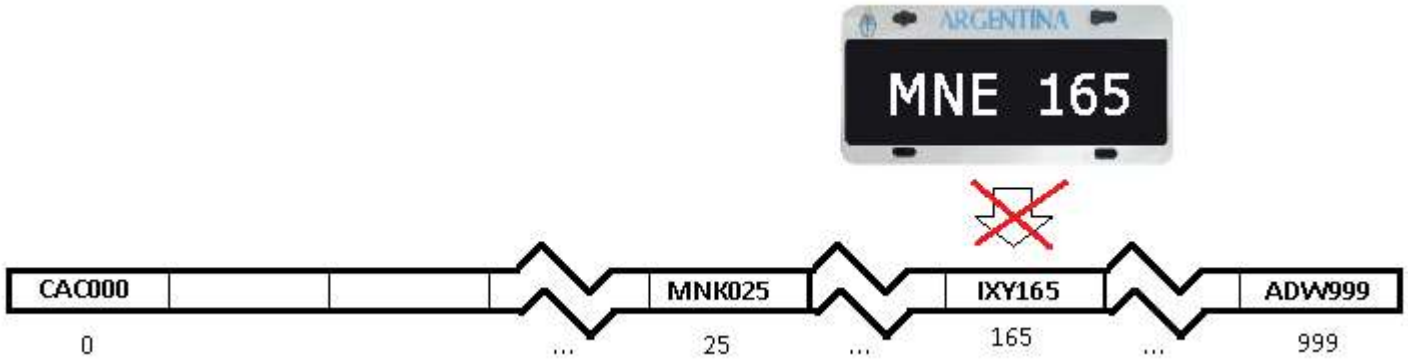
Hashing: Introducción

- Función de hashing:



Hashing: Introducción

- Colisión:

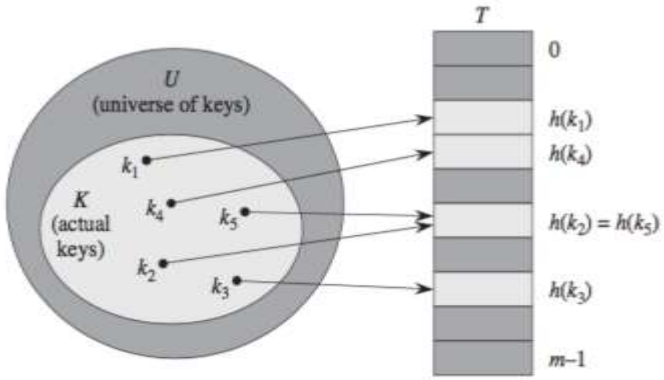


Funciones de hashing: Definición

- Una *tabla de hash* es una estructura de datos que asocia llaves o claves con valores que funcionan a modo de direcciones en la misma.
- Una *función de hash* o dispersión toma la clave del dato y devuelve un valor que será el índice de entrada de la tabla.
- Si *k* es una clave y *p* es una posición o entrada de la tabla decimos que:

$h: K \rightarrow P$

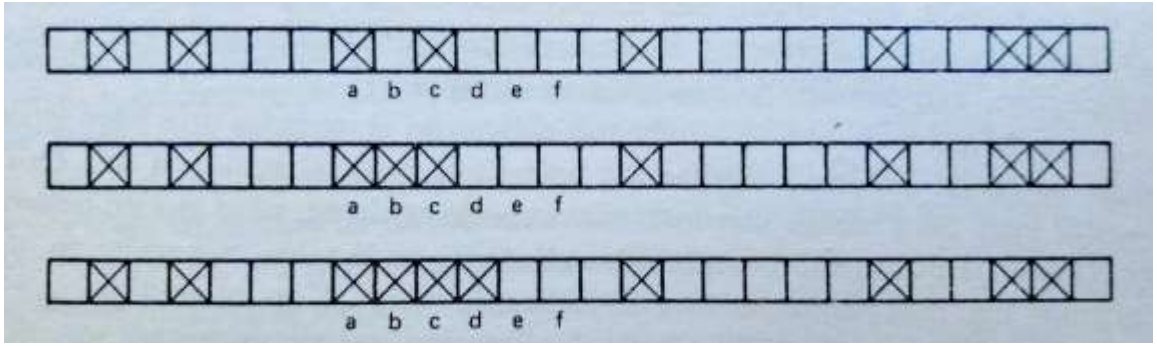
$p = h(k)$



Funciones de hashing: Definición

Una buena función de hashing:

- Producirá poca agrupación de elementos (buena dispersión).
- Evitará lo más posible las colisiones entre claves.



Funciones de hashing: Tamaño tabla

- Si la cantidad de claves a guardar es *n*, el tamaño de la tabla *m* debería ser tal que la proporción entre *n* y *m* sea aproximadamente 0.8.
- Esta proporción se llama *factor de carga* λ.
- Entonces:

$$\lambda = n/m$$

y pedimos:

$$\lambda <= 0{,}8$$

Funciones de hashing: Índices

- Si nuestras claves no son de tipo numéricas enteras debemos convertirlas a un formato de ese tipo.
- Si son letras, podríamos pasarlas a un *valor entero* con alguna convención, como tomar sus valores ASCII y sumarlos:

$$\text{“ab”} = 97 + 98 = 195$$

- Como los códigos ASCII se representan con 128 posiciones se aconseja aplicar esta base para dichos códigos:

$$ab = 97 \times 128 + 98 \times 128 = 12416 + 12544$$

Funciones de hashing:

- División
- Doblamiento (Folding)
- Mid-Square
- Extraction
- Radix Transformation

Funciones de hashing: División

- Se divide la clave *k* por la cantidad de posiciones de la tabla *t* y se toma el *resto* (operadores mod , %).

- Entonces:

$$p = h(k) = k\%t$$

- El operador resto nos genera valores que van en el rango 0..*t* − 1.
- Se recomienda que el valor de *t*, que es el tamaño de la tabla, sea un *número primo*.

Funciones de hashing: División

Ejemplo.

- La cantidad de claves a almacenar es 5000.
- Con un factor de carga de 0.8 deberíamos tener un tamaño de la tabla aproximado a:

$$t = 5000/0{,}8 = \mathbf{6250}$$

- El primo superior más cercano es **6257**.
- La función de hash a utilizar será:

$$p = k\%6257$$

Funciones de hashing: División

Ejemplo.

- Si necesitamos ingresar a la tabla la clave 113521 deberá ir en la posición:

$$p = 113521\%6257 = \mathbf{895}$$

- Y la clave 28413 debe ir a la posición:

$$p = 28413\%6257 = \mathbf{3385}$$

- Si luego debemos ingresar el dato con la clave 44694:

$$p = 44694\%6257 = \mathbf{895}$$

Funciones de hashing: Multiplicación

- Se multiplica a la clave por un valor *A* tal que *0 < A < 1*.
- Se toma la parte fraccionaria del resultado y se la multiplica por el tamaño de la tabla.
- El resultado final se redondea y da la posición final.

$$h(k) = t * ((k * A) \bmod 1)$$

Funciones de hashing: Folding

Ejemplo. El número de CUIT 23-31562313-7 podemos dividirlo en 4 partes tomando de a 3 dígitos: 233 - 156 - 231 – 37.

- Estos valores los sumamos: 233 + 156 + 231 + 37 = **657.**
- Si el tamaño de la tabla es menor al número obtenido se aplica la *función módulo.*
- En el caso de claves de tipo *string* se puede hacer algún corte obteniendo strings más pequeños para realizar un *xor* entre ellos.
- Logra mejor dispersión.

Funciones de hashing: Folding

Ejemplo.

- Se tiene la cadena *s* = "*abcd*".
- La cortamos en dos cadenas: *s*1 = "*ab*" y *s*2 = "*cd*", tomamos los valores ASCII de cada una.
- El valor de a es 97, el de b, 98, etc.
- Tomamos los bits correspondientes y hacemos un *xor*.

posición/cadena	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
ab	0	1	1	0	0	0	0	1	0	1	1	0	0	0	1	0
cd	0	1	1	0	0	0	1	1	0	1	1	0	0	1	0	0
xor	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1	0

- El resultado anterior nos da: 512 + 4 + 2 = **518.**

Funciones de hashing: Mid-square

- Toma la clave, la eleva al cuadrado, y luego se queda con los dígitos centrales.

Ejemplo.

- La clave es 1536, entonces se hace $1536^2 = 23\mathbf{592}96$
- Nos quedamos con los dígitos centrales: 592.
- Siempre se puede aplicar si fuera necesario la función módulo.

Funciones de hashing: Extraction

- Se ignora una parte de la clave y se utiliza la parte restante.

Ejemplo.

- Si utilizamos el mismo ejemplo del número de CUIT **23-31**562**3 13-7**
- Tomamos los 4 primeros dígitos: 2331, con los últimos 4: 3137, o una combinación tomando los dos primeros con los dos últimos: 2337, etc.
- No logra una buena distribución pero es muy veloz.

Función hashing: Radix Transformation

- Toma la clave y la cambia de base.

Ejemplo.

- Si la clave es 425, lo asume como que está expresado en base 16.
- La nueva clave será:

$$K' = 4 \cdot 16^2 + 2 \cdot 16^1 + 5 \cdot 16^0 =$$

$$= 4 \cdot 256 + 2 \cdot 16 + 5 = \mathbf{1061}$$

Función hashing: Universal hashing

- Elige para cada clave una función de hashing *aleatoria* de un conjunto de funciones a disposición.

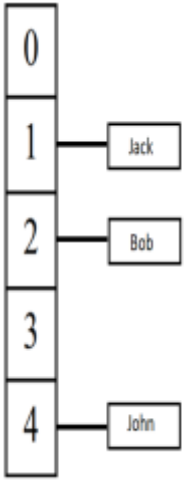
- Reduce muchísimo las chances de que suceda el peor caso O(n), ya que es muy poco probable que todas las claves vayan a parar al mismo lugar.

Colisiones: Definición

- Se producen cuando más de una clave nos devuelve *el mismo valor* luego de aplicarle la o las funciones de dispersión.
- La cantidad de colisiones depende de la función de hash aplicada y también del tamaño de la tabla.
- Puede afectar también la eliminación o la búsqueda.
- Métodos de resolución: Hash cerrado, hash abierto, bucket addressing

Colisiones: Hash cerrado

- Todos los elementos se guardan en la misma tabla (vector).
- Hash(clave) = posición en el vector.
- Si dos claves tienen la misma posición se tiene que resolver la colisión con los siguientes métodos: Open Addressing, Linear Probing, Quadratic Probing, Double Hashing



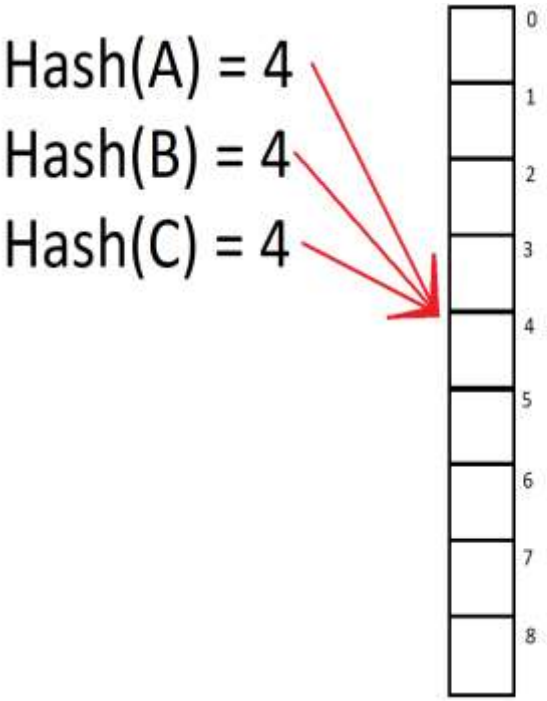
Colisiones: Open addresing

- Si se produce una colisión se busca una nueva posición tomando alguna nueva función que prueba en primer lugar con el valor 1, luego con el 2, etc.
- Si $h(k)$ está ocupada prueba con $h(k) + p(1)$, y si también lo está intenta con $h(k) + p(2)$, etc, donde p es una nueva función.
- Al finalizar siempre se aplica la función módulo.

Colisiones: Linear probing

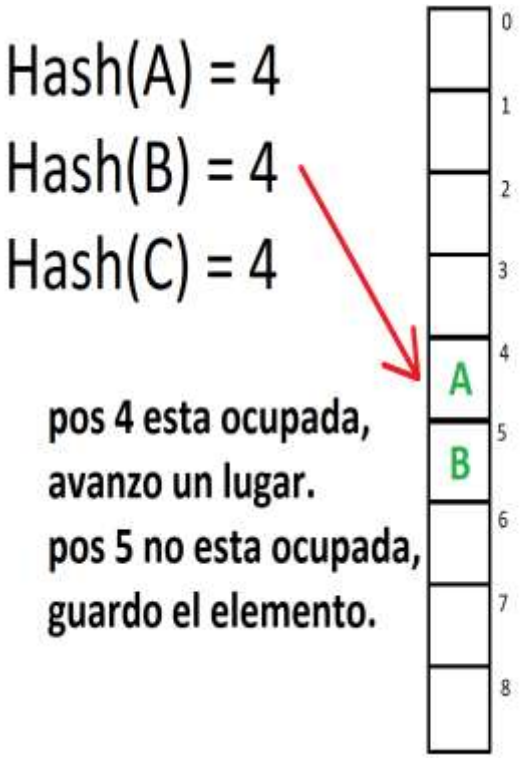
- *Sondeo lineal* es la decisión más simple del direccionamiento abierto.
- Se define la función *p* como $p(i) = i$ por lo tanto, lo que hace es ir verificando las posiciones siguientes a la que la función de hash indica.
- Por ejemplo, si $h(k) = 132$ y esta posición no está libre, se intenta en la posición 133, luego en la 134, etc.
- Este procedimiento se repite hasta alcanzar el final de la tabla, en cuyo caso se comienza desde el principio hasta encontrar la primera posición libre.

Colisiones: Linear probing



Colisiones: Linear probing

Colisiones: Linear probing



Colisiones: Linear probing

Colisiones: Quadratic probing

- Una mejora es tomar la función p como cuadrática.
- En este caso $p(i) = i^2$. Por lo tanto, en el mismo ejemplo anterior, si $h(k) = 132$ se encuentra ocupada, intenta

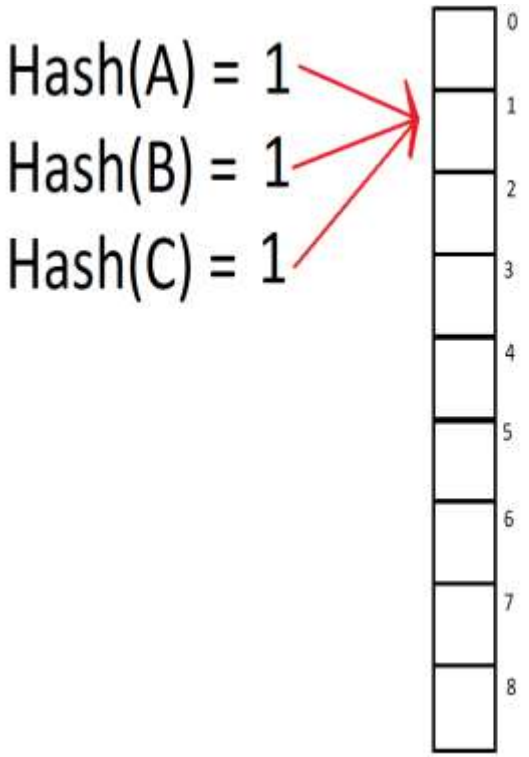
$$h(k) + p(1) = 132 + 1^2 = \mathbf{133}$$

- Si esta posición está también ocupada, intenta con

$$h(k) + p(2) = 132 + 2^2 = \mathbf{136}$$

- Los datos quedan de forma más dispersa y no se agrupan en ciertas zonas.

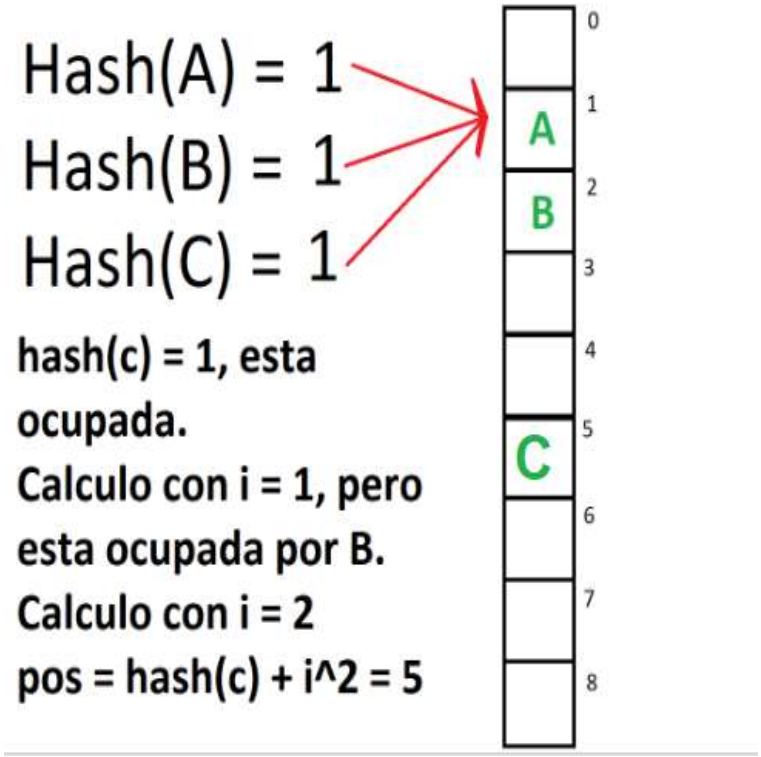
Colisiones: Quadratic probing



Colisiones: Quadratic probing

Colisiones: Quadratic probing

Colisiones: Quadratic probing



Colisiones: Double hashing

- Se usan 2 funciones de hash para obtener la posición.
- La *secuencia* hasta encontrar una posición vacía es la siguiente:

$$p = h(k) + i \cdot h_2(k)$$

con $i = 0, 1, 2...$

- Si se produce una colisión se suma una nueva función de dispersión, en el caso de seguir colisionando, se sumará el doble de dicha función, etc.
- Se debe finalizar tomándole módulo con el tamaño de la tabla.

Colisiones: Double hashing

Ejemplo.

- La función $h(k)$ toma el módulo de la clave con cierto valor t (tamaño de la tabla), digamos que $t = 1019$.
- La segunda función de hash también sea módulo pero con un valor distinto, por ejemplo, 1018.
- Si tenemos la clave 15924, $h(15924) = 639$.
- En caso de tener ocupada esa posición hacemos:

$$p = h(15924) + h_2(15924)$$

$$= 639 + 654 = \mathbf{1293}$$

Colisiones: Double hashing

Ejemplo.

- A este valor volvemos a aplicarle módulo con 1019 y queda **274**.
- Si tuviéramos una nueva colisión, los cálculos serían:

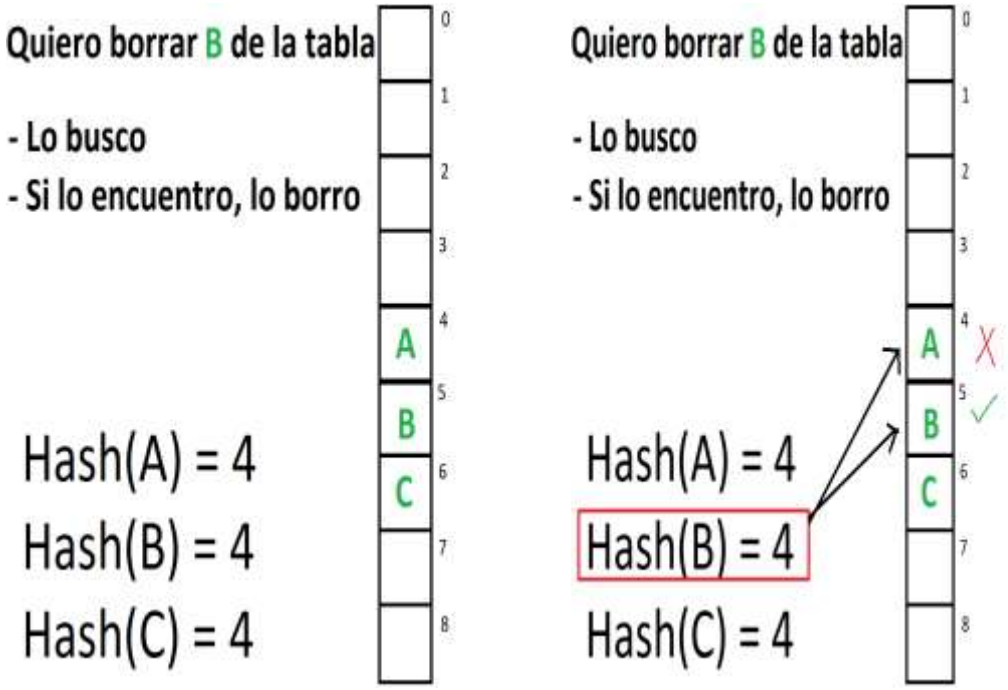
$$p = h(15924) + 2 \cdot h_2(15924)$$

$$= 639 + 2 \cdot 654$$

$$= 639 + 1308 = \mathbf{1947}$$

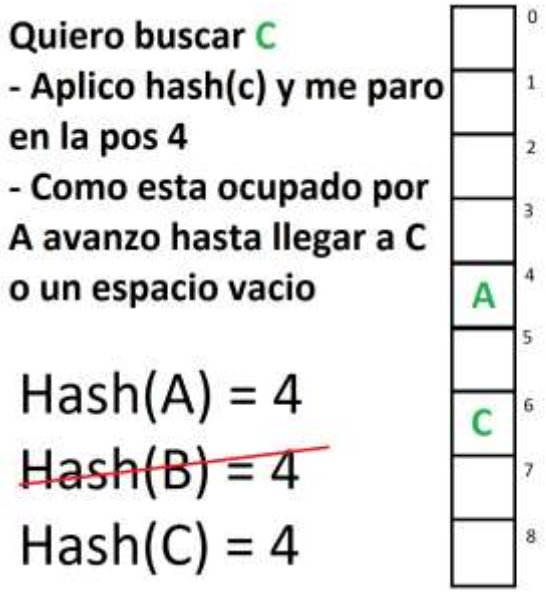
Colisiones: Borrado

- Para eliminar un elemento tengo que buscarlo en la tabla y si existe borrarlo.



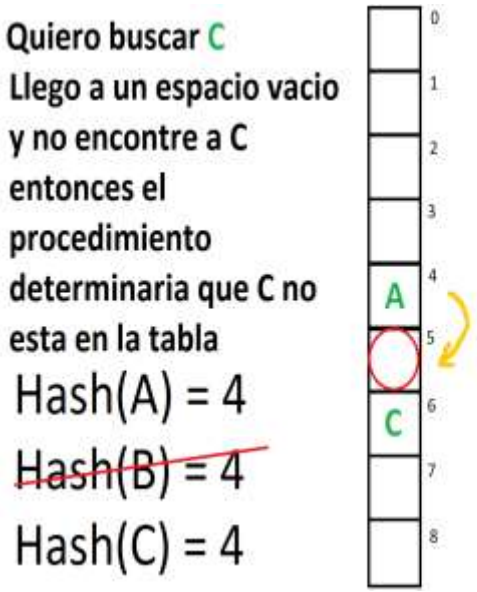
Colisiones: Borrado

- Esto trae otro problema, porque si yo ahora intento de buscar o borrar C.



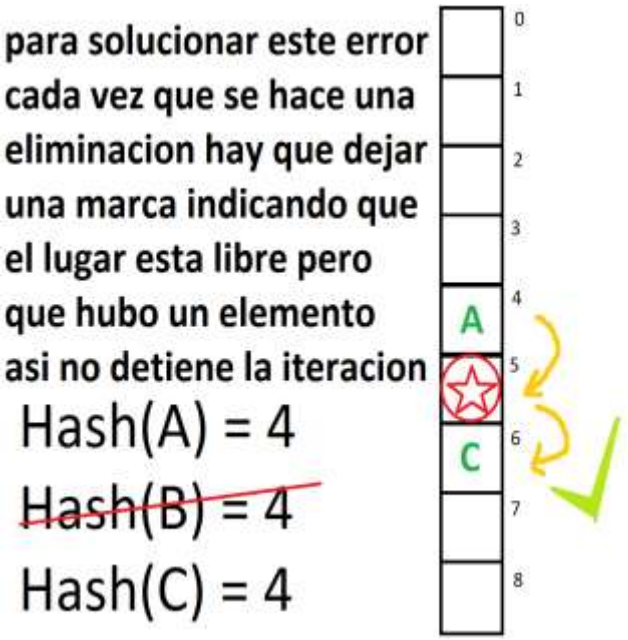
Colisiones: Borrado

- Se interrumpe la iteración por llegar a un espacio vacío sin haber encontrado el elemento.



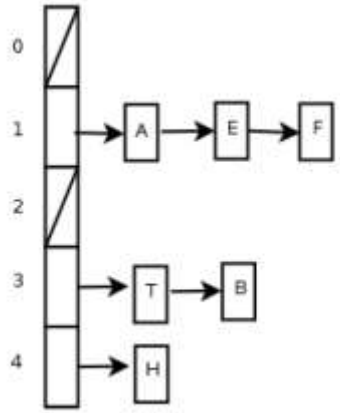
Colisiones: Borrado

- Dejar una marca indicando que el lugar esta libre, pero hubo un elemento.



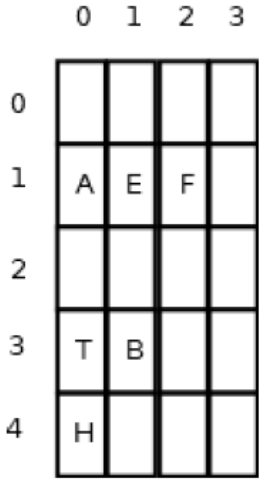
Colisiones: Hash abierto (chaining)

- Cada posición de la tabla en realidad será un puntero a una lista enlazada con las claves.
- Por ejemplo, supongamos que las claves A, E y F devuelven el valor 1 luego de aplicarles la función de hash, las claves B y T, el valor 3 y H el valor 4.
- Entonces, si las claves ingresan en el siguiente orden: A, H, E, T, F y B, la tabla queda:



Colisiones: Bucket addresing

- Varios elementos se almacenan en una misma posición de la tabla.
- Se utiliza una estructura de matriz.
- Cada posición es un bloque que contiene varios elementos por lo que se las llama “buckets”.



Funciones de hashing perfectas

- Es una función de hash que *no produce colisiones.*
- Si las claves no son conocidas no podemos garantizar que esto suceda porque desconocemos si alguna futura clave pueda llegar a producir una colisión.
- En los problemas en que las claves son conocidas de antemano podemos conseguir plantear una función de dispersión sin colisiones.

Fin