



Búsqueda mediante Hashing

- Existe un método que puede aumentar la velocidad de búsqueda donde los datos no necesitan estar ordenados y esencialmente es independiente del número n . Este método se conoce como **transformación de claves** o **hashing**. El hashing consiste en convertir el elemento almacenado (numérico o alfanumérico) en una dirección (índice) dentro del array.
- La idea general de usar la clave para determinar la dirección del registro es una excelente idea, pero se debe modificar de forma que no se desperdicie tanto espacio. Esta modificación se lleva a cabo mediante una función que transforma una clave en un índice de una tabla y que se denomina **función de Randomización o Hash**. Si **H** es una función hash y **X** es un elemento a almacenar, entonces **H(X)** es la función hash del elemento y se corresponde con el índice donde se debe colocar X.



Búsqueda mediante Hashing

- El método anterior tiene una deficiencia: suponer que dos elementos X e Y son tales que $H(X) = H(Y)$. Entonces, cuando un el elemento X entra en la tabla, éste se inserta en la posición dada por su función Hash, **$H(X)$** . Pero cuando al elemento Y le es asignado su posición donde va a ser insertado mediante la función hash, resulta que la posición que se obtiene es la misma que la del elemento X . Esta situación se denomina ***Randomización o Hashing con colisión o choque***.
- Una buena función Hash será aquella que minimice los choques o coincidencias, y que distribuya los elementos uniformemente a través del array. Esta es la razón por la que el tamaño del array debe ser un poco mayor que el número real de elementos a insertar, pues cuanto más grande sea el rango de la función de randomización, es menos probable que dos claves generen el mismo valor de asignación o hash, es decir, que se asigne una misma posición a más de un elemento.



Métodos de Transformación de claves

TRUNCAMIENTO: Ignora parte de la clave y se utiliza la parte restante directamente como índice. Si las claves, por ejemplo, son enteros de 8 dígitos y la tabla de transformación tiene 100 posiciones, entonces el 1º, 2º y 5º dígitos desde la derecha pueden formar la función hash. Por ejemplo, 72588495 se convierte en 895.

El truncamiento es un método muy rápido, pero falla para distribuir las claves de modo uniforme.

MÉTODO DE DIVISIÓN: Se escoge un número m mayor que el número n de elementos a insertar, es decir, el array posee más posiciones que elementos a insertar. La función hash H se define por:

$$H(X) = X \% m \quad \text{o} \quad H(X) = (X \% m) + 1$$

donde $X \% m$ indica el resto de la división de X por m . La segunda fórmula se usa cuando queremos que las direcciones hash vayan de 1 a m en vez de desde 0 hasta $m-1$.

El mejor resultado del método de división se obtiene cuando m es primo (es decir, m no es divisible por ningún entero positivo distinto de 1 y m).



Métodos de Transformación de claves

MÉTODO DEL MEDIO DEL CUADRADO: La clave es multiplicada por sí misma y los dígitos del medio (el número exacto depende del rango del índice) del cuadrado son utilizados como índice.

MÉTODO DE SUPERPOSICIÓN: Consiste en la división de la clave en diferentes partes y su combinación en un modo conveniente (a menudo utilizando suma o multiplicación) para obtener el índice. La clave X se divide en varias partes X_1, X_2, \dots, X_n , donde cada una, con la única posible excepción de la última, tiene el mismo número de dígitos que la dirección especificada. A continuación, se suman todas las partes. En esta operación se desprecian los dígitos más significativos que se obtengan de arrastre o acarreo.

Hay dos formas de conseguir la función hash mediante este método:

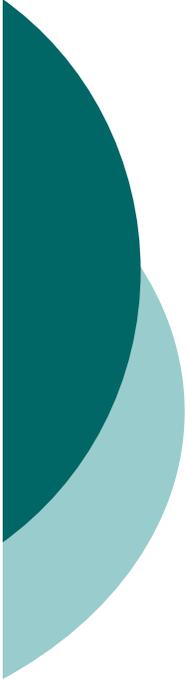
1. **superposición por desplazamiento** donde todas las partes se suman entre sí.
2. **superposición por plegado** se hace la inversa a las partes pares, X_2, X_4, \dots , antes de sumarlas, con el fin de afinar más.



Ejemplos de Transformación de claves

Supongamos un array con 100 posiciones para guardar las claves de los empleados. Aplica las funciones hash anteriores para cada uno de los empleados: 3205 7148 2345.

- **Método de la división:** Escojo un n° primo próximo a 100 ($m = 97$). Aplico la función Hash $H(X) = X \% m$.
 - $H(3205) = 4$, $H(7148) = 67$, $H(2345) = 17$.
- **Método del medio del cuadrado:** Se escogen el 4^o y el 5^o dígitos por la derecha para obtener la dir. Hash
 - K: 3205 7148 2345
 - K^2 : 10272025 51093904 5499025



Ejemplos de Transformación de claves

○ Método de Superposición:

- Por desplazamiento:

- $H(3205) = 32 + 05 = 37$

- $H(7148) = 71 + 48 = 19$

- $H(2345) = 23 + 45 = 68$

- Por plegado:

- $H(3205) = 32 + 50 = 82$

- $H(7148) = 71 + 84 = 55$

- $H(2345) = 23 + 54 = 77$



Soluciones al problema de las colisiones

- o Supongamos que queremos añadir un nuevo elemento k , pero la posición de memoria $H(k)$ ya está ocupada. Esta situación se llama **colisión**.
- o Para resolver las colisiones se utilizan los siguientes métodos:
 - Rehashing o Reasignación.
 - Encadenamiento o Tablas Hash Abiertas.
 - Zona de Desbordamiento.



Soluciones al problema de las colisiones

- Un factor importante en la elección del procedimiento a utilizar es la relación entre el número n de elementos y el número m de tamaño del array. Conocido como **factor de carga**, $\lambda = n/m$.
- La **eficiencia** de una función hash con un procedimiento de resolución de colisiones se mide por el número medio de *pruebas* (comparaciones entre elementos) necesarias para encontrar la posición de un elemento X dado. La eficiencia depende principalmente del factor de carga λ . En concreto, interesa conocer:
 - $S(\lambda)$ = nº medio de celdas examinadas para una búsqueda CON éxito
 - $U(\lambda)$ = nº medio de celdas examinadas para una búsqueda SIN éxito.



Rehasing o Reasignación

- o Se trata de insertar o buscar un elemento cuya posición ya está ocupada, en otra posición disponible en la tabla.
- o Esto se hace mediante la **función de reasignación $R(H(X))$** , la cual acepta un índice de la tabla y produce otro.
- o Métodos de reasignación:
 - Prueba lineal
 - Prueba cuadrática
 - Doble dirección hash



Reasignación. Prueba Lineal

- Consiste en que una vez detectada la colisión se debe recorrer el array secuencialmente a partir del punto de colisión, buscando al elemento. El proceso de búsqueda concluye cuando el elemento es hallado, o bien cuando se encuentra una posición vacía.
- Se trata al array como a una estructura circular: el siguiente elemento después del último es el primero.
- La función de rehashing es, por tanto, de la forma: $R(H(X)) = (H(X) + 1) \% m$



Código Prueba Lineal

```
public static int pruebaLineal(int X, int[] A){
    int m = A.length;
    int dirHash = X%m;

    if (A[dirHash] == X) return dirHash;
    else {
        int dirReh = (dirHash + 1)%m;
        while ((A[dirReh] != X) && (A[dirReh] != 0) && (dirReh !=
dirHash))
            dirReh = (dirReh + 1)%m;

        /* Se ha encontrado el elemento buscado*/
        if (A[dirReh] == X) return dirReh;
        else return -1;
    }
}
```



Complejidad Prueba Lineal

- o Utilizando este método de resolución de colisiones, el número medio de pruebas para una búsqueda CON éxito es:
$$S(\lambda) = \frac{1}{2} (1 + (1 / (1-\lambda)))$$
- o y el número medio de pruebas para una búsqueda SIN éxito es:
$$U(\lambda) = \frac{1}{2} (1 + (1 / (1-\lambda)^2))$$

La principal desventaja de este método es que puede haber un fuerte agrupamiento alrededor de ciertas claves, mientras que otras zonas del array permanezcan vacías. Si las concentraciones de claves son muy frecuentes, la búsqueda será principalmente secuencial perdiendo así las ventajas del método hash. Una solución es que la función hash acceda a todas las posiciones de la tabla, pero no de una en una posición sino avanzando varias posiciones.



Rehasing, mejora

- Para evitar que la Prueba Lineal se convierta en una búsqueda Lineal: Se cumple que, para cualquier función $R(H(i)) = (i+c) \% m$, donde m es el número de elementos de la tabla y c es una constante tal que c y m son *primos relativos* (es decir, no tienen factores en común), **genera valores sucesivos que cubren toda la tabla.**
- Sin embargo, si m y c tienen factores en común, el número de posiciones diferentes de la tabla que se obtienen será el cociente de dividir m entre c .
- Basándonos en esta última afirmación, el algoritmo de búsqueda (y el de carga) tienen un problema: utilizando una función de rehashing de este tipo podrían salir sin inserción aun cuando exista alguna posición vacía en la tabla

Ejemplo:

Dada la función de reasignación:

$RH(i) = (i + 200) \% 1000$ Con esta función, cada clave solo puede ser colocada en cinco posiciones posibles: $(1000/200 = 5)$

si $i = 215$, y esta posición está ocupada, entonces se reasigna de la siguiente forma:

$RH(215) = (215+200) \% 1000$	=	415
$RH(415) = (415+200) \% 1000$	=	615
$RH(615) = (615+200) \% 1000$	=	815
$RH(815) = (815+200) \% 1000$	=	15
$RH(15) = (15+200) \% 1000$	=	215
$RH(215) = (215+200) \% 1000$	=	415

.....

Si estas cinco posiciones posibles están ocupadas, saldremos sin inserción aun cuando haya posiciones libres en la tabla.



Rehashing: Agrupamiento o clustering

- A pesar de la utilización de una función de rehashing donde c y m sean primos, con el método de prueba lineal los elementos siguen tendiendo a *agruparse*, o sea, a aparecer unos junto a otros, cuando el factor de carga es mayor del 50%.
- Cuando la tabla está vacía es igualmente probable que cualquier elemento al azar sea colocado en cualquier posición libre dentro de la tabla. Pero una vez que se han tenido algunas entradas y se han presentado varias colisiones en la asignación, esto no es cierto.

Ejemplo de Agrupamiento

Clave	Índice
4618396	396
4957397	397
	398
1286399	399
	400
	401
.....	
0000990	990
0000991	991
1200992	992
0047993	993
	994
9846995	995

- En este caso, es cinco veces más probable que un registro sea insertado en la posición 994 que en la posición 401. Esto se debe a que cualquier registro cuya clave genera la asignación 990, 991, 992, 993 o 994 será colocado en la 994, mientras que cualquier registro cuya clave genera la asignación 401 será colocado en su posición.
- Este fenómeno en el que **dos claves que inicialmente generan una asignación en dos sitios diferentes, luego compiten entre sí en reasignaciones sucesivas**, se denomina **agrupamiento o clustering**.



Agrupamiento o clustering

- **Agrupamiento o clustering:** $H(X) \leftrightarrow H(Y)$ y $RH(X) = RH(Y)$ en reasignaciones sucesivas, se cumple que:
 - cualquier función de reasignación que dependa únicamente del índice dará lugar a agrupamiento.
 - daría agrupamiento la función de reasignación $Rh(i) = (i + c) \% m$ aunque c y m fuesen primos.



Rehashing. Prueba cuadrática

- o El clustering se debe a que asignaciones sucesivas siguen la misma secuencia. Si se consigue que esa secuencia varíe en cada reasignación se evitaría la formación de cluster.
- o Las técnicas de **prueba cuadrática** y **doble dirección hash** minimizan el agrupamiento.



Rehashing. Prueba cuadrática.

- Este método es similar al de la prueba lineal. La diferencia consiste en que, en lugar de buscar en las posiciones con direcciones: dir_Hash , $\text{dir_Hash} + 1$, $\text{dir_Hash} + 2$, $\text{dir_Hash} + 3$,
- buscamos linealmente en las posiciones con direcciones: dir_Hash , $\text{dir_Hash} + 1$, $\text{dir_Hash} + 4$, $\text{dir_Hash} + 9$, ..., **$\text{dir_Hash} + i^2$**
- Si el número m de posiciones en la tabla T es un número primo y el factor de carga no excede del 50%, sabemos que siempre insertaremos el nuevo elemento X y que ninguna celda será consultada dos veces durante un acceso.



Código. Prueba Cuadrática.

```
public static int pruebaCuadratica(int X, int[] A){
    int m = A.length;
    int dirHash = X%m;
    if (A[dirHash] == X) return dirHash;
    else {
        int i = 1;
        int cont = 0;
        int dirReh = (dirHash + 1)%m;

        while ((A[dirReh] != X) && (A[dirReh] != 0) && (cont < m*10))
        {
            i++;
            dirReh = (dirHash + (i*i))%m;
            cont++;
        }
        if (A[dirReh] == X) return dirReh;
        else return -1;
    }
}
```



Rehashing. Doble dirección hash.

- Consiste en que una vez detectada la colisión se debe generar otra dirección aplicando una función hash H_2 a la dirección previamente obtenida. Entonces buscamos linealmente en las posiciones que se encuentran a una distancia $H_2(X)$, $2 H_2(X)$, $3 H_2(X)$, ...
- La función hash H_2 que se aplique a las sucesivas direcciones puede ser o no ser la misma que originalmente se aplicó a la clave. No existe una regla que permita decidir cuál será la mejor función a emplear en el cálculo de las sucesivas direcciones. Pero una buena elección sería $H_2(X) = R - (X \% R)$, siendo R un número primo más pequeño que el tamaño del array. Y se obtienen unos resultados mejores cuando el tamaño del array es un número primo.



Código. Doble dirección Hash.

```
public static int dobleDireccionamiento(int X, int R, int[] A){
    int m = A.length;
    int dirHash = X%m;
    if (A[dirHash] == X) return dirHash;
    else {
        int dirHash2 = R - (X%R);
        int i = 1;
        int dirReh = (dirHash + dirHash2)%m;
        while ((A[dirReh] != X) && (A[dirReh] != 0) )
        {
            i++;
            dirReh = (dirHash + i*dirHash2)%m;
        }
        if (A[dirReh] == X) return dirReh;
        else return -1;
    }
}
```



Problemas de Rehasing

- **INSERCIÓN:** Como se asume una tabla de tamaño fijo, si el número de elementos aumenta más allá de ese tamaño es imposible insertarlo sin que sea necesario asignar una tabla más grande y recalcular los valores de asignación de las claves de todos los elementos que ya se encuentran en la tabla utilizando una nueva función de asignación.
- **BORRADO:** Es difícil eliminar un elemento. Por ejemplo, si el elemento $r1$ está en la posición p , para añadir un elemento $r2$ cuya clave $k2$ queda asignada en p , éste debe ser insertado en la primera posición libre de las siguientes: $Rh(p)$, $Rh(Rh(p))$.. Si luego $r1$ es eliminado y la posición p queda vacía, una búsqueda posterior del elemento $r2$ comenzará en la posición $H(k2) = p$. Como esta posición está ahora vacía, el proceso de búsqueda puede erróneamente llevarnos a la conclusión de que el elemento $r2$ no se encuentra en la tabla.
Solución: marcar el elemento eliminado como '*eliminado*' en vez de '*vacío*', y continuar la búsqueda cuando se encuentra una posición como '*eliminada*'.



Tablas Hash Abiertas

- Otro método para resolver las colisiones consiste en mantener una lista encadenada de todos los elementos cuyas claves generan la misma posición. Si la función hash genera valores entre 0 y $(m - 1)$, declaramos un array de nodos de encabezamiento de tamaño m , de manera que $\text{Tabla}[i]$ apunta a la lista con todos los elementos cuyas claves generan posiciones en i .
- ¿INSERCIÓN?: Al buscar un elemento cuya función hash le hace corresponder la posición i , se accede a la cabeza de la lista correspondiente a esa posición, $\text{Tabla}[i]$, y se recorre la lista que dicha posición inicia. Si éste no se encuentra entonces se inserta al final de la lista.
- ¿ELIMINACIÓN?: La eliminación de un nodo de una tabla que ha sido construida mediante randomización y encadenamiento se reduce simplemente a eliminar un nodo de la lista encadenada. Un nodo eliminado no afecta a la eficiencia del algoritmo de búsqueda. El algoritmo continúa como si el nodo nunca se hubiera insertado.



Tablas Hash Abiertas.

- Estas listas pueden reorganizarse para maximizar la eficiencia de búsqueda, utilizando diferentes métodos:
 - **PROBABILIDAD.** Consiste en insertar los elementos dentro de la lista en su punto apropiado. Esto significa que si **pb** es la probabilidad de que un elemento sea el argumento buscado, entonces el elemento debe insertarse entre los elementos $r(i)$ y $r(i+1)$, donde i es tal que $P(i) \geq pb \geq P(i+1)$
 - **MOVIMIENTO AL FRENTE:** Cuando una búsqueda ha tenido éxito, el elemento encontrado es retirado de su posición actual en la lista y colocado en la cabeza de dicha lista.
 - **TRASPOSICIÓN:** Cuando una búsqueda de un elemento ha tenido éxito es intercambiado con el elemento que le precede inmediatamente, de manera que si es accedido muchas veces llegará a ocupar la primera posición de la lista.



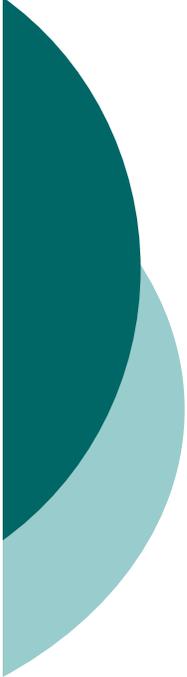
Complejidad Tablas Hash Abiertas

- El número medio de pruebas, con encadenamiento, para una búsqueda con éxito y para una búsqueda sin éxito tiene los siguientes valores aproximados:

$$S(\lambda) = 1 + \frac{1}{2} \lambda$$

$$U(\lambda) = e^{-\lambda} + \lambda$$

- ¿*VENTAJAS?*: Es el método más eficiente debido al dinamismo propio de las listas. Cualquiera que sea el número de colisiones registradas en una posición, siempre será posible tratar una más.
- ¿*DESVENTAJAS?*: La desventaja principal es el espacio extra adicional que se requiere para la referencia a otros elementos. Sin embargo, la tabla inicial es generalmente pequeña en esquemas que utilizan encadenamiento comparado con aquellos que utilizan reasignación. Esto se debe a que el encadenamiento es menos catastrófico si la tabla llega a llenarse completamente, pues siempre es posible asignar más nodos y añadirlos a varias listas.



Zona de desbordamiento

- Se trata de mantener una zona reservada para aquellos elementos que llegan a colisionar, de manera que cuando se produzca una colisión el elemento se va a localizar en esta zona de desbordamiento.
- Al realizar la búsqueda y comprobar que la posición está ocupada por otro elemento con el mismo valor de hashing, se seguirá buscando a partir del inicio de la zona de desbordamiento de manera secuencial, hasta encontrar el elemento o llegar al final de dicha zona de desbordamiento.