

# Estructuras de Datos y Algoritmos

## Tema 5: Tablas de Dispersión

Departamento de Informática  
Universidad de Valladolid

**Curso 2022-23**

Grado en Ingeniería Informática  
Grado en Estadística





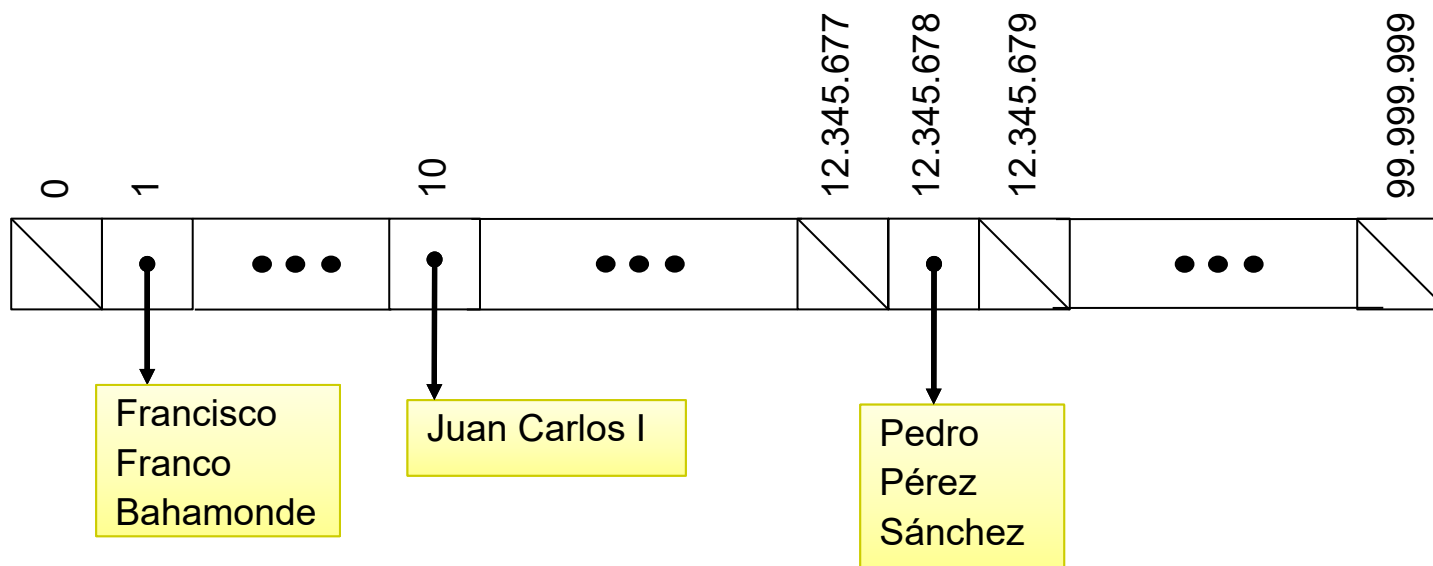
# 1. DEFINICIONES Y OBJETIVOS

# Motivación

- Los **árboles equilibrados** consiguen un orden  $O(\log n)$  para las operaciones básicas de los TADs **Conjunto**, **Mapa**, **Diccionario** y **Lista ordenada**.
- En el tema 3, sin embargo, vimos que los **arrays de bits** pueden conseguir un orden  $O(1)$  para acceso, inserción y borrado por valor.
- Problemas:
  - Los valores/claves deben ser números naturales.
  - El espacio de almacenamiento era proporcional al número de claves posibles,  $u$ . En general este valor es enorme.
  - Las operaciones de acceso al  $i$ -ésimo menor y recorrido en orden son  $O(u)$ .
- Esto los convierte en inútiles para la gran mayoría de casos.

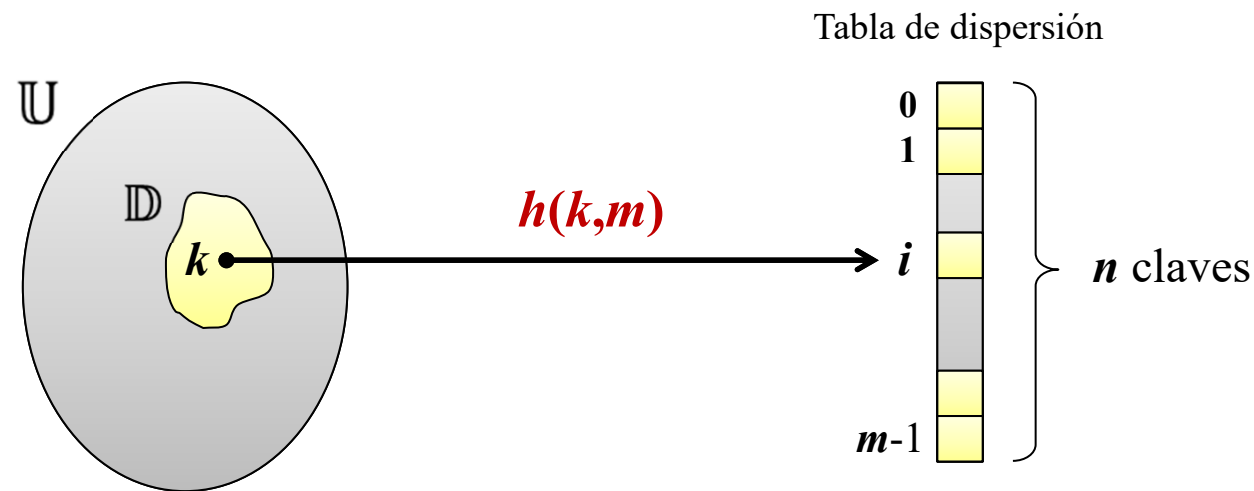
# Caso práctico

- Supongamos un **mapa** en el que se almacenan datos sobre personas cuya clave es el DNI (número de 8 dígitos).
- Se necesitaría un array indexado por DNI, que almacena **100.000.000** de enlaces a registros/objetos.
- Ese espacio es **independiente** del número de registros que realmente almacenemos.



# Solución

- La solución consiste en definir una **función** que permita **traducir** y **comprimir** la clave a un **índice** de una **tabla**, cuyo tamaño,  **$m$** , sea proporcional al número de elementos almacenados,  **$n$** , en vez de al número de claves posibles.
- El problema de éste enfoque son las **colisiones**. Según la manera de resolverlo hablaremos de dispersión **perfecta**, **ábierta** o **cerrada**.



# Notación utilizada

- $\mathbb{U}$ : **Espacio de claves**. Representa todos los valores posibles del tipo de datos que representa a las claves. Usaremos el símbolo  $k$  para representar una clave.  $u$  es el tamaño de  $\mathbb{U}$ , el número de posibles valores de clave.
  - $\mathbb{D}$ : **Espacio de datos**. Es el subconjunto de  $\mathbb{U}$  que representa las claves que se van a almacenar.  $n$  es el tamaño de  $\mathbb{D}$ , el número de claves almacenadas.
  - $m$  : Capacidad de la **tabla de dispersión**. El objetivo es que sea proporcional a  $n$ , no a  $u$ . Es decir  $m \in O(n)$ .
  - $h'(k)$  : **Función de dispersión primaria**. Traduce (y posiblemente comprime) un valor de clave a una secuencia de bits que se interpreta como un número natural.
  - $h(k, m)$  : **Función de dispersión secundaria**. Traduce y comprime un valor de clave a un entero en el rango  $[0..m-1]$ .
- $i = h(k, m)$  es el **índice** de la tabla correspondiente a la clave  $k$ .

# Funciones de Dispersión (Hash)

- Una **función de dispersión** (hash function) **traduce** un valor de un determinado tipo de datos (posiblemente complejo) a una secuencia de bits que suele interpretarse como un **número natural**.
- Si dos claves son **iguales** (en el contexto en que se trabaja), la función de dispersión debe devolver **el mismo valor** para ambas (**determinismo**):

$$k_1 = k_2 \Rightarrow h'(k_1) = h'(k_2)$$

- Pero el inverso no es cierto: Es perfectamente posible (y habitual) que dos claves **distintas** obtengan **el mismo valor** al aplicarseles la función de dispersión (**colisión**):

$$k_1 \neq k_2 \nRightarrow h'(k_1) \neq h'(k_2)$$

# Funciones de Dispersión Criptográficas

- En criptografía, las funciones de dispersión (o resumen) se aplican a un **documento** o una serie de datos para obtener un **valor resumen** (digest, MAC, checksums) que se pueda usar para validarlos (los datos y el resumen se adquieren por “canales” distintos)
- Se obtiene una secuencia de bits (típicamente 256 ó 512 bits)
- El objetivo primordial es diseñar funciones “**de un sólo sentido**”, en el que el cálculo  $i = h(k)$  sea sencillo, pero el inverso (conocido  $i$  obtener un  $k$  cualquiera que al aplicar la función devuelva  $i$ ) sea básicamente imposible (por muestreo)
- No suelen usarse para tablas de dispersión (existen alternativas más eficientes)
- Ejemplos: MD5, ~~SHA-1~~, SHA-2, SHA-3, BLAKE3.



## Funciones de Dispersión Normales

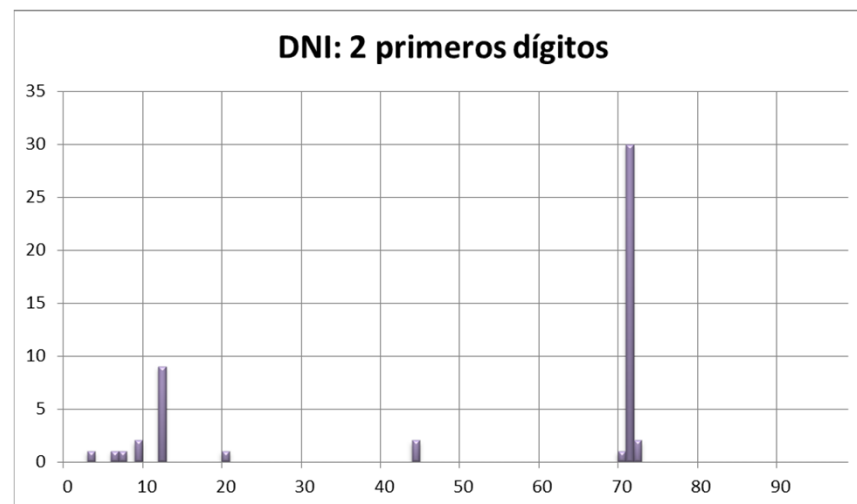
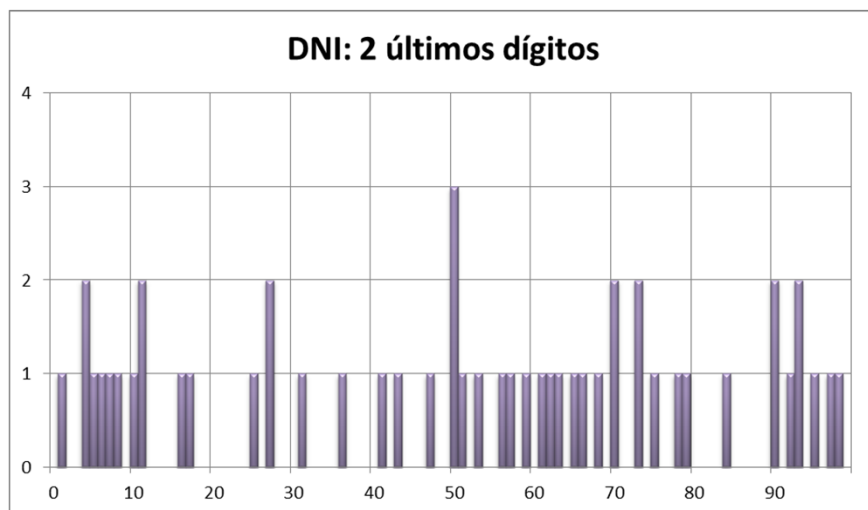
- Lo habitual es que el resultado sea un entero positivo en el rango típico de la máquina (32 o 64 bits)
- Existe una función **distinta** por cada **tipo de datos** que se vaya a utilizar para representar las claves. Ejemplos:
  - Enteros:  $h'(n) = |n|$
  - Datos con más bits que un entero: Se dividen en trozos con el tamaño en bits de un entero y se hace un **xor** entre ellos.
  - Cadenas de caracteres: 
$$h'(c_0..c_{n-1}) = \sum_{i=0}^{n-1} 31^i \cdot c_i$$
- En Python existe la función **hash** para tipos de datos simples.
- En Java la clase **Object** incluye la función **hashCode**, por lo que todo objeto tiene definida una función de dispersión. **Cuidado**: Para nuevas clases devuelve la dirección de memoria del objeto.

# Colisiones y Uniformidad

- Se produce una **colisión** cuando a dos claves **distintas** la función de dispersión les asigna la **misma** posición en la tabla.
  - Si se desconocen los datos concretos que se van a introducir..
  - .. y el tamaño de la tabla es menor que el tamaño del espacio de claves ( $m < u$ )..
  - .. entonces la existencia de colisiones es **inevitable**.
- Respecto a la eficiencia, es importante que las funciones de dispersión tengan un comportamiento **uniforme** respecto al **conjunto de datos** que se almacenan.
  - Uniforme significa que la probabilidad de que a cualquier clave  $k$  del conjunto de datos se le asigne el índice  $i$  de la tabla sea  $1/m$ .
  - La misma función de dispersión puede ser uniforme para un conjunto de datos y no serlo para otro (no se puede **garantizar** que una función de dispersión sea uniforme para cualquier conjunto de datos)

## Ejemplo de uniformidad

- Tomamos los DNI's de 50 alumnos de EDA y escogemos dos funciones de dispersión, la izquierda toma los dos últimos dígitos del DNI y la derecha los dos primeros:



- Si el conjunto de datos fuera “personas con DNI terminado en 00”, la primera función pasaría a no ser uniforme.

# Factor de Carga

- Se define **factor de carga** como el ratio entre el número de elementos y la capacidad de una tabla de dispersión.

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

$$\alpha = \frac{n}{m + 1}$$

← Versión que utilizaremos para los análisis.  $L \cong \alpha$

- En dispersión cerrada (0 o 1 claves por celda) es un valor entre 0 (tabla vacía) y 1 (tabla llena)
- En dispersión abierta (una celda puede almacenar varias claves) puede tomar valores mayores que 1.
- La **eficiencia** de las operaciones sobre las tablas de dispersión dependen directamente del factor de carga.
- Para que se puedan considerar de tiempo constante, se establece un límite,  $L_{max}$ . Cuando al insertar elementos el factor de carga supera ese límite, se **reestructura** la tabla: Se amplía su tamaño,  $m$  (típicamente se duplica).

# Funciones de dispersión secundaria

En general existen dos niveles en la definición de funciones de dispersión:

- En el primero se definen funciones que **traducen** el tipo de datos a un número natural.
  - Las denominamos  $h'(k)$
  - Están asociadas a cada tipo de dato concreto (definidas en la clase base (Java) o mediante una interfaz).
- En el segundo nivel se **comprime** el valor obtenido del primer nivel a un índice en el rango  $[0..m-1]$ .
  - Las denominamos  $h(k, m)$ .
  - Se debe tener en cuenta que el tamaño de la tabla de dispersión,  $m$ , puede cambiar debido a las reestructuraciones.

## Funciones de dispersión secundaria

- El método más utilizado se denomina **método de división**:

$$h(k, m) = h'(k) \bmod m$$

- Para evitar problemas con la uniformidad, se debería escoger un valor de  $m$  que fuera primo (al reestructurar se escogería el siguiente primo mayor que  $2 \cdot m$ )
- Otro enfoque (Java) consiste en “barajar” los bits de  $h'(k)$  antes de calcular el módulo. Con ello se pueden usar valores de  $m$  que sean potencias de dos (simplifica la reestructuración):

$$(h'(k) \xrightarrow{\text{Desplaz. bits derecho}} \gg 20) \otimes (h'(k) \gg 12) \xrightarrow{\text{xor}} (h'(k) \gg 7) \otimes (h'(k) \gg 4) \otimes h'(k)$$

- Existen otros métodos (multiplicación, producto de vectores) que no estudiaremos.

# Recapitulación

- Una **tabla de dispersión** consistirá en una **tabla** (array) de capacidad  $m$  que contiene **elementos o listas de elementos**,  $n$  elementos en un momento dado. Los elementos pueden ser pares clave-valor.
- Se necesita una **función de dispersión primaria** que traduzca las claves a números positivos.
- Se define un **función de dispersión secundaria** que comprima esos números a índices (típicamente el método de división).
- Se especifica un **factor de carga máximo**. Cuando al insertar se supere ese valor, se **reestructura** la tabla.
- Veremos 3 **variantes** de tablas, según el método de resolver el problema de las **colisiones**: Dispersión perfecta, abierta y cerrada.



## 2. DISPERSIÓN PERFECTA



# Dispersión Perfecta

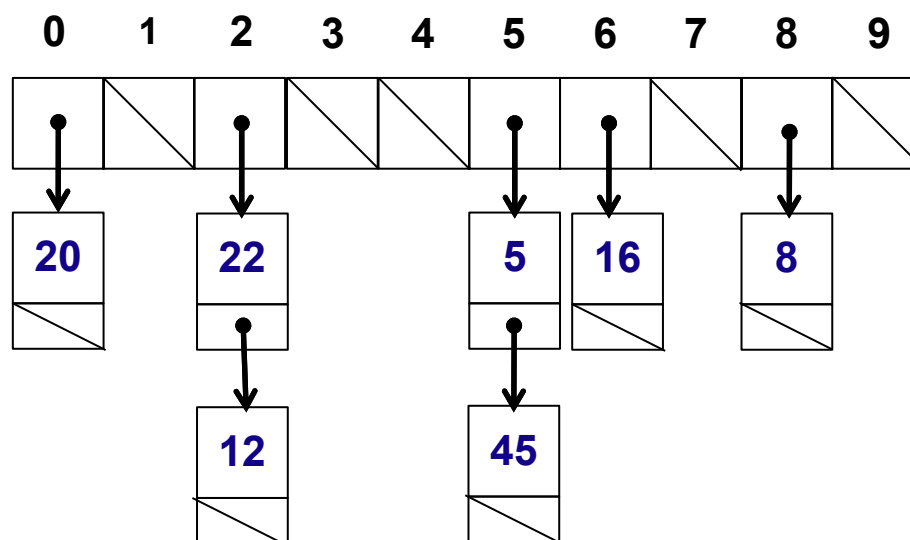
- La estrategia de **dispersión perfecta** (perfect hashing) consiste en encontrar un **conjunto de funciones de dispersión** que al ser aplicadas a un conjunto de datos conocido no produzca **ninguna colisión** entre ellos.
- El conjunto de datos debe conocerse **antes** del diseño de la tabla, y **no se puede modificar**.
- Sólo se permiten operaciones de **acceso**, no se puede modificar la tabla (ni insertar ni borrar).
- Se basa en un conjunto de funciones de dispersión, parametrizadas por un valor aleatorio (producto vectorial).
- La búsqueda de las funciones adecuadas se realiza al **azar**, pero está garantizado que en un tiempo promedio  $O(n)$  se pueden encontrar las funciones adecuadas.



### 3. DISPERSIÓN ABIERTA (CHAINING)

## Dispersión Abierta (*chaining*)

- La estrategia de **dispersión abierta** resuelve el problema de las colisiones permitiendo que se almacene **más de un elemento** en cada celda de la tabla.
- Lo habitual es que cada celda almacene un enlace a una **lista simplemente enlazada** de elementos.
- Ejemplo: (Secuencia de inserciones: 16, 8, 12, 20, 22, 45, 5)



$$h(k, 10) = k \bmod 10$$

# Operaciones

- Las operaciones basadas en **valor** (búsqueda, inserción y borrado por valor) consisten en:
  - Usar la función de dispersión secundaria para obtener el **índice** de la tabla correspondiente al valor (o clave).
  - Realizar la operación sobre la **lista enlazada** referenciada en esa posición.
  - La búsqueda y el borrado **recorren** la lista hasta encontrar el elemento (búsqueda secuencial).
  - La inserción es **inmediata**: Se crea un nuevo nodo con el elemento (o par clave/valor) y se inserta al principio de la lista.
  - Si en la inserción se supera el factor de carga máximo, se **reestructura** la tabla.
- Para cualquier otro tipo de operación (acceso  $i$ -ésimo menor, recorrido ordenado, fusión, etc.) se comporta como un **vector de listas desordenadas**.

# Eficiencia promedio

- Si la función de dispersión es **uniforme** para el conjunto de datos utilizado:
  - Cada clave tiene la misma probabilidad de que se le asigne cualquiera de los  $m$  índices.
  - La longitud promedio de las listas es  $n/m$ , igual al factor de carga
- **Búsquedas exitosas:**  $1 + L/2$  accesos en promedio.
  - En una lista de longitud  $L$  el promedio es  $L/2$  accesos.
  - El **borrado** de una clave equivale a una búsqueda exitosa.
- **Búsquedas fallidas:**  $1 + L$  accesos en promedio.
  - $m$  búsquedas fallidas, una por cada índice de la tabla, recorren la suma de longitudes de las listas:  $(m + n)/m = 1 + L$
- **Inserción:**  $1$  acceso mejor caso,  $O(1)$  en tiempo amortizado.
  - La **reestructuración** es  $O(n)$ , pero garantiza  $n$  inserciones en  $O(1)$

## Eficiencia peor caso

- El peor caso se da cuando la función de dispersión es **extremadamente no uniforme**: Asigna la misma posición a todas o la mayoría de las claves del conjunto de datos.
- La tabla contiene **una única lista** con los  **$n$**  elementos.
- En circunstancias normales la probabilidad de caer en el peor caso es insignificante ( $1/m!$ )
- Pero definida una función de dispersión, siempre es posible diseñar un conjunto de datos que provoque el peor caso (ataque por degradación de eficiencia).
  - **Búsqueda, borrado**:  **$O(n)$**  accesos en promedio.
  - **Inserción**: Sigue siendo  **$O(1)$**  en tiempo amortizado.

# Resumen eficiencia

$L = n/m < L_{\max}$   
 $m \in O(n)$   
 $L \in O(1)$   
 \* Tiempo amortizado

	Uniforme promedio (exacto)	Uniforme promedio	No uniforme promedio
Búsqueda exitosa	$1 + L/2$	$O(1)$	$O(n)$
Búsqueda fallida	$1 + L$	$O(1)$	$O(n)$
Borrado por valor	$1 + L/2$	$O(1)$	$O(n)$
Inserción por valor	$1 \mid n$	$O(1)^*$	$O(1)^*$
Espacio	$O(n + m)$	$O(n)$	$O(n)$

# Ejemplo en Java (I)

```
// Tabla de dispersión abierta que almacena pares clave-valor
public class TablaDispAbi<K,V> {
    // Clase interna que representa un
    // nodo de una lista simplemente enlazada
    private class Nodo<K,V> {
        K clave;
        V valor;
        Nodo<K,V> sig;
        Nodo(K clave, V valor, Nodo<K,V> sig) {
            this.clave = clave; this.valor = valor;
            this.sig = sig;
        }
    }

    int m;           // Capacidad de la tabla
    int n;           // Número de elementos
    double maxL;     // Máximo factor de carga
}
```



## Ejemplo en Java (II)

```
// Tabla de dispersión (array de listas de pares)
Nodo<K,V>[] tabla;

// Constructor con valores por defecto
public TablaDispAbi() { this(16,2.5); }

// Constructor: m0 - capacidad inicial
// maxL - factor de carga máximo
public TablaDispAbi(int m0, double maxL) {
    this.maxL = maxL;
    this.m = m0;
    tabla = new Nodo[m];
    for(int i = 0; i < m; i++) tabla[i] = null;
    this.n = 0;
}

// Devuelve el indice correspondiente a esa clave
protected int indice(K c) { return Math.abs(c.hashCode()) % m; }
```

## Ejemplo en Java (IV)

```
protected void reestructurar() {  
    // Salvamos la tabla anterior  
    Nodo<K,V>[] tmp = tabla;  
    // Creamos una nueva tabla  
    n = 0; m = 2*m; // Duplicamos el tamaño  
    tabla = new Nodo[m];  
    for(int i = 0; i < m; i++) tabla[i] = null;  
    // Recorremos la tabla anterior insertando elementos  
    for(int i = 0; i < tmp.length; i++) {  
        Nodo<K,V> nodo = tmp[i];  
        while(nodo != null) {  
            ins(nodo.clave, nodo.valor);  
            nodo = nodo.sig;  
        }  
    }  
}
```

## Ejemplo en Java (V)

```
public V get(K clave) {  
    // Aplicar función de dispersión a la clave  
    int i = indice(clave);  
    // Buscar en la lista i-ésima  
    Nodo<K,V> p = tabla[i];  
    while(p != null && !p.clave.equals(clave)) p = p.sig;  
    return (p == null) ? null : p.valor;  
}  
  
public void ins(K clave, V valor) {  
    // Incrementar n y comprobar factor de carga  
    n++;  
    if((1.0*n)/m > maxL) reestructurar();  
    // Aplicar función de dispersión a la clave  
    int i = indice(clave);  
    // Insertar al principio de la lista i-ésima  
    tabla[i] = new Nodo(clave, valor, tabla[i]);  
}
```

## Ejemplo en Java (VI)

```
public boolean del(K clave) {  
    // Aplicar función de dispersión a la clave  
    int i = indice(clave);  
    // Buscar nodo controlando elemento anterior  
    Nodo<K,V> ant = null;  
    Nodo<K,V> act = tabla[i];  
    while(act != null && !act.clave.equals(clave)) {  
        ant = act;  
        act = act.sig;  
    }  
    if(act == null) { return false; }  
    // Comprobar caso especial borrado del primero  
    if(ant == null) tabla[i] = act.sig; else ant.sig = act.sig;  
    n--;  
    return true;  
}
```



## 4. DISPERSIÓN CERRADA (OPEN ADDRESSING)

## Dispersión Cerrada (*open addressing*)

- En la estrategia de **dispersión cerrada** cada celda de la tabla almacena **un único elemento** (o ninguno).
  - Una consecuencia directa es que, a diferencia de la dispersión abierta, el número de elementos almacenados **no puede superar** la capacidad de la tabla:  **$n < m$ ,  $L < 1.0$**
- El problema de las colisiones se resuelve estableciendo una **secuencia de posiciones** (ruta de exploración) en las que puede encontrarse un elemento en la tabla.
  - Dispersión abierta: Varios elementos en cada celda.
  - Dispersión cerrada: Varias celdas disponibles para cada elemento
- Cada celda de la tabla puede estar en 3 estados distintos:
  - **Ocupada**: Contiene un elemento, no se puede insertar en ella.
  - **Vacía**: Se puede insertar en ella, detiene la exploración.
  - **Borrada**: Se puede insertar en ella, no detiene la exploración.

## Dispersión Cerrada (II)

- Los **estados** de las celdas se pueden representar:
  - Mediante **valores especiales** de las claves (uno para indicar celda vacía y otro para indicar celda borrada)
  - O mediante una **tabla extra** que almacene valores que indiquen en cuál de los 3 estados se encuentra la celda correspondiente de la tabla principal.
- Además de la función de dispersión, en este tipo de tablas es necesario definir una **estrategia de exploración**
  - La estrategia indica cuál es la **siguiente celda** que se debe explorar cuando la celda actual no esté vacía (inserción) o no contenga el elemento buscado (acceso, borrado)
  - Se suele expresar mediante una **función** que depende de la **posición inicial** (la proporcionada por la función de dispersión) y el **número de intento** (número de colisiones hasta el momento).
  - Cada intento debe proporcionar una posición de la tabla por la que no se halla pasado en intentos anteriores (**recorrido completo**)

# Exploración Lineal (consecutive probing)

- Es la estrategia de exploración más sencilla:
  - Cada nuevo intento explora la **siguiente** celda de la tabla.
  - Si estamos en la última celda, pasamos a la primera.

Posición inicial,  $i_0 = h(k, m)$

$$f(i_0, j, m) = (i_0 + j) \bmod m$$

Número de intento

- Ejemplo: Inserción del valor 28 en una tabla con  $m = 10$  y los siguientes elementos ya incluidos:

0	1	2	3	4	5	6	7	8	9
30		22				56		48	18

$$i_0 = h(28, 10) = 8$$





# Exploración Lineal (consecutive probing)

- Es la estrategia de exploración más sencilla:
  - Cada nuevo intento explora la **siguiente** celda de la tabla.
  - Si estamos en la última celda, pasamos a la primera.

Posición inicial,  $i_0 = h(k, m)$

$$f(i_0, j, m) = (i_0 + j) \bmod m$$

Número de intento

- Ejemplo: Inserción del valor 28 en una tabla con  $m = 10$  y los siguientes elementos ya incluidos:

0	1	2	3	4	5	6	7	8	9
30		22				56		48	18

$j = 1$

# Exploración Lineal (consecutive probing)

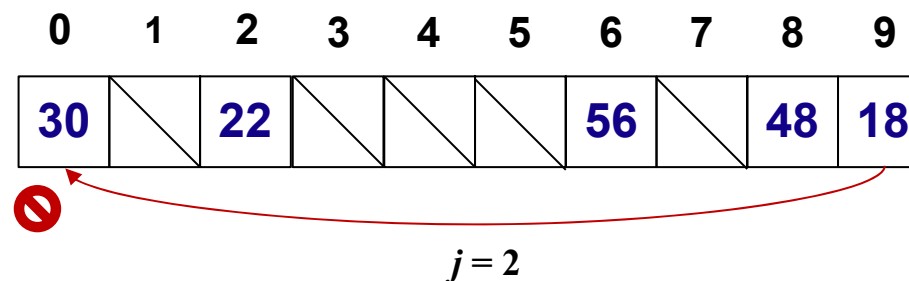
- Es la estrategia de exploración más sencilla:
  - Cada nuevo intento explora la **siguiente** celda de la tabla.
  - Si estamos en la última celda, pasamos a la primera.

Posición inicial,  $i_0 = h(k, m)$

$$f(i_0, j, m) = (i_0 + j) \bmod m$$

Número de intento

- Ejemplo: Inserción del valor 28 en una tabla con  $m = 10$  y los siguientes elementos ya incluidos:



# Exploración Lineal (consecutive probing)

- Es la estrategia de exploración más sencilla:
  - Cada nuevo intento explora la **siguiente** celda de la tabla.
  - Si estamos en la última celda, pasamos a la primera.

Posición inicial,  $i_0 = h(k, m)$

$$f(i_0, j, m) = (i_0 + j) \bmod m$$

Número de intento

- Ejemplo: Inserción del valor 28 en una tabla con  $m = 10$  y los siguientes elementos ya incluidos:

0	1	2	3	4	5	6	7	8	9
30	28	22				56		48	18

$j = 3$

# Agrupamiento (clustering)

- La estrategia de exploración lineal sufre del problema del **agrupamiento**: La formación de grandes bloques de celdas ocupadas.
  - Si se forma (al azar) un bloque de celdas ocupadas..
  - Si una clave es enviada a una posición dentro del bloque..
  - ..deberá explorarlo hasta el final, ocupando la siguiente posición vacía contigua al bloque..
  - ..y haciendo que el bloque incremente su tamaño!
- **Consecuencia**: Las claves se **agrupan en bloques** donde el **número de intentos es alto**, degradando la eficiencia.
  - Este problema es **independiente** de la uniformidad de la función de dispersión.
  - Deriva directamente de la estrategia de exploración.
- **Solución**: Usar un **factor de carga límite** más pequeño.

## Exploración Doble (double hashing)

- Otra solución reside en hacer que la exploración no dependa tan sólo de la posición inicial, sino también del **propio valor de la clave**.
  - De esa forma claves distintas que han sido enviadas a la **misma posición inicial** seguirán **rutas distintas**.
  - Y se consigue un mejor aprovechamiento de las posiciones vacías que existan en la tabla.
  - Para obtener otro parámetro dependiente de la clave se necesita definir una **segunda función de dispersión**, distinta de la usada habitualmente (por eso el nombre de exploración **doble**).
- Lo habitual es que ese segundo parámetro defina el **salto** en la exploración:

Cada intento salta **d** celdas

$$f(i_0, j, d, m) = (i_0 + d \cdot j) \bmod m$$

## Exploración Doble (II)

- Cada nuevo intento explora la celda situada a una **distancia** de  $d$  celdas a la derecha (la tabla se interpreta como si fuera circular). Si  $d = 1$  tendríamos una exploración lineal.
- El valor del salto,  $d$ , **depende** del valor de la clave. Un método habitual de definirlo, si se utiliza como función de dispersión secundaria el método de división, es:

$$i_0(k, m) = h'(k) \bmod m$$

$$d(k, m) = \max\{1, h'(k) \operatorname{div} m\}$$

- De esta forma se consigue un valor de salto **dependiente** de la clave pero **independiente** de la posición inicial.
- Existen algunos detalles a tener en cuenta para garantizar una exploración completa.


## Exploración Doble (III)

- La exploración lineal garantiza de forma trivial la exploración de toda la tabla.
- Pero una exploración a base de saltos de  $d$  celdas no siempre va a recorrer todas las celdas.
  - Por ejemplo, si la tabla tiene tamaño 12 y el salto es 4, sólo vamos a recorrer 3 celdas distintas antes de entrar en un ciclo.
- **Teorema:** Si  $m$  y  $d$  son **primos entre sí** (no tienen factores en común) esta garantizado un recorrido completo.
  - **Solución #1:** Imponer que  $m$  sea un **número primo**. Al reestructurar, escoger el siguiente primo mayor que  $2m$
  - **Solución #2:** Imponer que  $m$  se una **potencia de dos**, y que  $d$  sea un **número impar** (si es par, se le suma 1)

# Exploración Doble - Ejemplo

- Tabla con  $m = 2^3 = 8$  (solución #2), inserción de 17:

0	1	2	3	4	5	6	7
40	9	2		36		14	


 $i_0 = 17 \bmod 8 = 1$

0	1	2	3	4	5	6	7
40	9	2		36		14	



$d = 17 \text{ div } 8 = \cancel{2} \rightarrow 3$


0	1	2	3	4	5	6	7
40	9	2		36		14	17



# Exploración Doble - Ejemplo

- Misma tabla, inserción de 73:

0	1	2	3	4	5	6	7
40	9	2		36		14	17

  $i_0 = 73 \bmod 8 = 1$

0	1	2	3	4	5	6	7
40	9	2		36		14	17



0	1	2	3	4	5	6	7
40	9	2	73	36		14	17

$d = 73 \operatorname{div} 8 = 9$

## El problema del borrado (I)

- Supongamos que tenemos la siguiente tabla (método de exploración lineal) donde la secuencia de inserción ha sido 14, 25, 4, 5:

0	1	2	3	4	5	6	7	8	9
				14	25	4	5		

- Si buscamos el valor 15, comenzaremos en la posición 5 e iremos explorando las celdas siguientes hasta llegar a la 8, que está vacía. Detenemos la búsqueda porque si se hubiera insertado el valor 15 la exploración lo hubiera colocado en la celda 8 (o anteriores). Si está vacía, es que no se ha insertado.
- Pero si ahora borramos el valor 25, los valores 4 y 5 son inalcanzables!

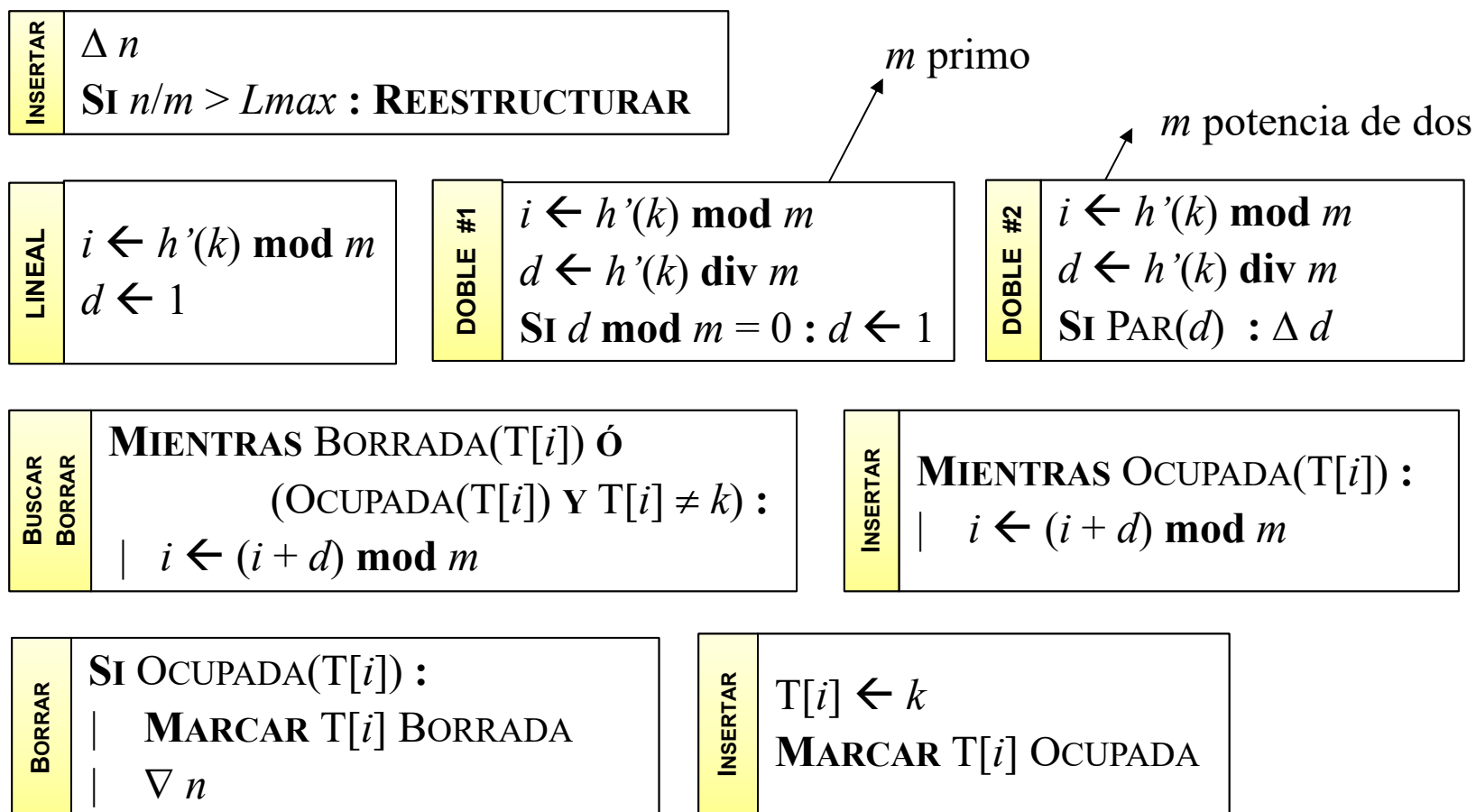
0	1	2	3	4	5	6	7	8	9
				14		4	5		

## El problema del borrado (II)

- Una posición vacía indica el **final** de una ruta de exploración
- Si al borrar un elemento marcamos la casilla como vacía, entonces se **rompen** las rutas en las que éste elemento es un punto intermedio.
- Si no se elimina el elemento, el **borrado** sería **imposible** (una búsqueda sobre él volvería a encontrarlo)
- Si la exploración no se detiene en las casillas vacías, las **búsquedas fallidas** recorrerían toda la tabla:  $O(m)$
- El recolocar los elementos de las rutas rotas causaría que el **borrado** fuera  $O(n)$ : El recolocar un elemento supone de hecho borrarle y puede producir nuevas recolocaciones en cascada.
- **Solución:** No eliminar el elemento, pero **marcar la casilla como borrada** para que la búsqueda no lo encuentre.

# Estrategia de borrado perezoso

- Se definen **3 estados** asociados a cada celda: **ocupado**, **vacío**, **borrado**. La lógica de las operaciones es:



# Ejemplo en Java (I)

```
// Tabla de dispersión cerrada que almacena pares clave-valor
public class TablaDispCer<K,V> {
    // Clase interna que representa un par clave-valor
    private class Par<K,V> {
        K clave;
        V valor;
        Par(K clave, V valor, Nodo<K,V> sig) {
            this.clave = clave; this.valor = valor;
        }
    }

    int m;           // Capacidad de la tabla
    int n;           // Número de elementos
    double maxL;     // Máximo factor de carga
    // Tabla de pares clave-valor. Si un par existe pero la
    // clave es nula, se ha realizado un borrado perezoso.
    Par<K,V>[] tabla;
}
```

## Ejemplo en Java (II)

```
// Constructor con valores por defecto
public TablaDispCer() { this(16,0.6); }

// Constructor: m0 - capacidad inicial (potencia de dos)
// maxL - factor de carga máximo (maxL < 1)
public TablaDispCer(int m0, double maxL) {
    this.maxL = maxL; this.m = m0;
    tabla = new Par[m];
    for(int i = 0; i < m; i++) tabla[i] = null;
    this.n = 0;
}

// Devuelve el indice correspondiente a esa clave
protected int indice(K c) { return Math.abs(c.hashCode()) % m; }

// Calcula el salto de exploración
protected int salto(K c) {
    int s = Math.abs(c.hashCode()) / m;
    return (s % 2 == 0) ? s+1 : s; }

```



VER TRANSP. 39

## Ejemplo en Java (IV)

```
protected void reestructurar() {  
    // Salvamos la tabla anterior  
    Par<K,V>[] tmp = tabla;  
    // Creamos una nueva tabla  
    n = 0; m = 2*m; // Duplicamos el tamaño  
    tabla = new Par[m];  
    for(int i = 0; i < m; i++) tabla[i] = null;  
    // Recorremos la tabla anterior insertando elementos  
    for(int i = 0; i < tmp.length; i++) {  
        Par<K,V> par = tmp[i];  
        if(par != null && par.clave != null) {  
            ins(par.clave, par.valor);  
        }  
    }  
}
```

## Ejemplo en Java (V)

```
public V get(K clave) {  
    // Aplicar función de dispersión a la clave  
    int i = indice(clave);  
    // Calcular el salto de exploración  
    int d = salto(clave);  
    // Explorar la tabla hasta posición nula o encontrado  
    // Una par nulo detiene la exploración, pero una  
    // clave nula no (borrado perezoso)  
    while(tabla[i] != null &&  
        (tabla[i].clave == null ||  
         !tabla[i].clave.equals(clave))) {  
        i = (i+d) % m;  
    }  
    return (tabla[i] == null) ? null : tabla[i].valor;  
}
```



## Ejemplo en Java (VI)

```
public void ins(K clave, V valor) {  
    // Incrementar n y comprobar factor de carga  
    n++; if(1.0*n/m > maxL) reestructurar();  
    // Aplicar función de dispersión a la clave  
    int i = indice(clave);  
    int d = salto(clave);  
    // Explorar la tabla hasta encontrar un par nulo o  
    // una clave nula (borrado perezoso)  
    while(tabla[i] != null && tabla[i].clave != null) i = (i+d) % m;  
    // Insertar clave en posición  
    if(tabla[i] == null) {  
        tabla[i] = new Par(clave,valor);  
    } else { // Borrado perezoso, reutilizar el par  
        tabla[i].clave = clave; tabla[i].valor = valor;  
    }  
}
```

## Ejemplo en Java (VII)

```
public boolean del(K clave) {  
    // Aplicar función de dispersión a la clave  
    int i = indice(clave);  
    int d = salto(clave);  
    // Explorar la tabla hasta posición nula o encontrado  
    // Las claves nulas no detienen la exploración (borrado perezoso)  
    while(tabla[i] != null &&  
        (tabla[i].clave == null || !tabla[i].clave.equals(clave))) {  
        i = (i+d) % m;  
    }  
    if(tabla[i] == null) { return false; }  
    tabla[i].clave = null; // Borrado perezoso  
    n--;  
    return true;  
}
```

# Análisis dispersión cerrada (I)

- Llamamos  $T_f(m, n)$  al número **promedio** de accesos hasta encontrar una **posición vacía** en una tabla de capacidad  $m$  que contiene  $n$  elementos.
  - Si la **función de dispersión** es **uniforme**, la probabilidad de caer en una celda ocupada es  $n/m$
  - En ese caso, la **función de exploración** calculará **otra posición** de la tabla. Ya que no va a volver a elegir la celda donde estamos, el problema es idéntico al de buscar una posición vacía en una tabla de capacidad  $m-1$  con  $n-1$  elementos.
- Se tiene la relación de **recurrencia**:

$$T_f(m, n) = 1 + \frac{n}{m} \cdot T_f(m - 1, n - 1)$$

## Análisis dispersión cerrada (II)

- Esta relación se puede resolver por tanteo, explorando la forma que adquiere con valores crecientes de  $n$

$$T_f(m, n) = 1 + \frac{n}{m} \cdot T_f(m-1, n-1)$$

$$T_f(m, 0) = 1$$

$$T_f(m, 1) = 1 + \frac{1}{m} \cdot T_f(m-1, 0) = 1 + \frac{1}{m} = \frac{m+1}{m}$$

$$T_f(m, 2) = 1 + \frac{2}{m} \cdot T_f(m-1, 1) = 1 + \frac{2}{m} \cdot \frac{m}{m-1} = \frac{m+1}{m-1}$$

$$T_f(m, 3) = 1 + \frac{3}{m} \cdot T_f(m-1, 2) = 1 + \frac{3}{m} \cdot \frac{m}{m-2} = \frac{m+1}{m-2}$$

$$T_f(m, n) = \frac{m+1}{m+1-n} = \frac{1}{1-\frac{n}{m+1}} = \boxed{\frac{1}{1-\alpha}}$$

## Análisis dispersión cerrada (III)

- Llamamos  $T_e(m, n)$  al número **promedio** de accesos hasta encontrar un **elemento existente** en una tabla de capacidad  $m$  que contiene  $n$  elementos.
  - Es el tiempo de una **búsqueda exitosa** de un elemento.
  - La búsqueda recorre el **mismo camino** seguido en la **inserción** de ese elemento en la tabla
  - El tiempo de búsqueda del  $i$ -ésimo elemento que se insertó en la tabla es igual al del tiempo promedio para encontrar una **posición vacía** en una tabla de capacidad  $m$  y  $i-1$  elementos.
  - Se realiza el promedio sobre los  $n$  elementos existentes:

$$T_e(m, n) = \frac{1}{n} \sum_{i=1}^n T_f(m, i - 1)$$

# Análisis dispersión cerrada (IV)

- Sustituyendo:

$$T_e(m, n) = \frac{1}{n} \sum_{i=1}^n T_f(m, i-1) = \frac{m+1}{n} \sum_{i=0}^{n-1} \frac{1}{m+1-i}$$

- Cambiando la variable del sumatorio por  $j = m+1-i$ :

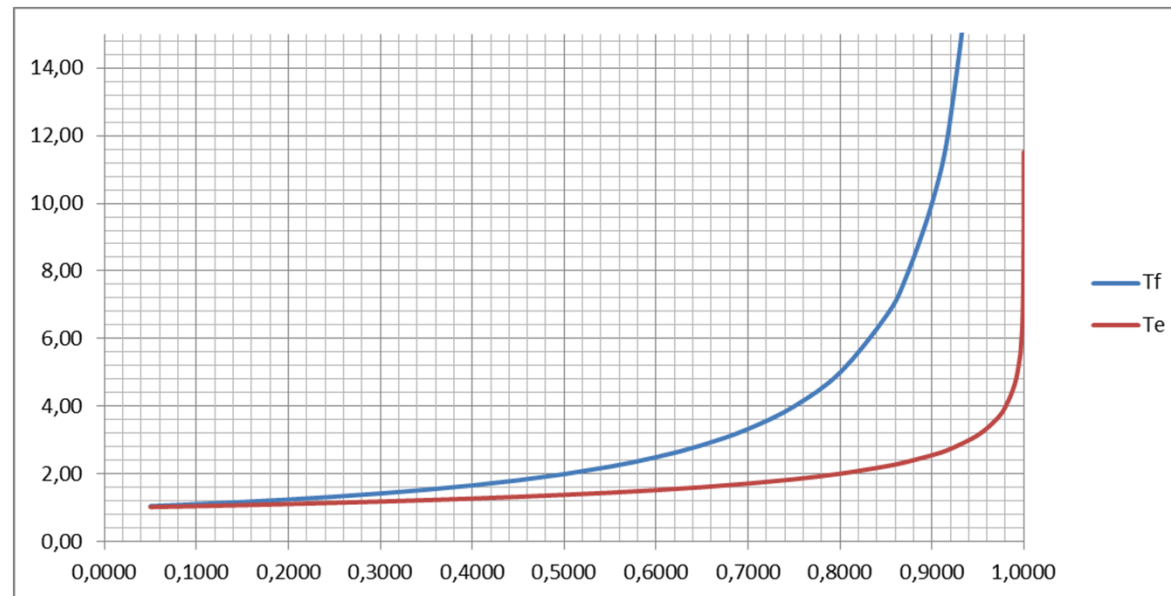
$$T_e(m, n) = \frac{m+1}{n} \sum_{j=m-n+2}^{m+1} \frac{1}{j} = \frac{m+1}{n} \left( \sum_{j=1}^{m+1} \frac{1}{j} - \sum_{j=1}^{m+1-n} \frac{1}{j} \right)$$

- Aplicando la fórmula  $\sum_{i=1}^n 1/i = \ln n + \gamma$

$$T_e(m, n) = \frac{m+1}{n} \ln \frac{m+1}{m+1-n} = \boxed{\frac{1}{\alpha} \ln \frac{1}{1-\alpha}}$$

# Análisis dispersión cerrada (V)

- Del análisis anterior podemos concluir:
  - La **dependencia con respecto al factor de carga** es mucho mayor que en dispersión abierta: Cuando  $L \rightarrow 1$ , el número de accesos tiende a infinito (más acusado en las búsquedas fallidas).
  - La **exploración lineal** es muy sensible a la **uniformidad** de la función de dispersión (por el agrupamiento), la exploración doble es mucho más robusta en ese aspecto.



# Resumen eficiencia

\* Tiempo amortizado

	Dispersión abierta (exacto)	Dispersión cerrada (exacto)	Uniforme	No uniforme
Búsqueda exitosa	$1 + L/2$	$\ln(1/(1+L))/L$	$O(1)$	$O(n)$
Búsqueda fallida	$1 + L$	$1/(1+L)$	$O(1)$	$O(n)$
Borrado por valor	$1 + L/2$	$\ln(1/(1+L))/L$	$O(1)$	$O(n)$
Inserción por valor	$1 \mid n$	$1/(1+L)$	$O(1)^*$	$O(n)$

- Para cualquier otro tipo de operaciones, una tabla de dispersión cerrada se comporta como un vector desordenado con posiciones vacías y borradas entre los elementos.



# Uso de las Tablas de Dispersión

- Las tablas de dispersión, cuando se cumplen todos sus requisitos, son una alternativa mejor que los árboles equilibrados para los **TADs Conjunto y Mapa**.
  - Se debe definir una **función de dispersión**.
  - El factor de carga no debe superar un límite (**reestructuraciones**)
  - La función de dispersión debe ser (+ o -) **uniforme** para el conjunto de datos utilizado.
- No se comportan bien, sin embargo, para TADs con un **orden interno**: **Lista ordenada, Cola de Prioridad, Diccionario**.
- ¿Dispersión abierta o cerrada?
  - La **dispersión cerrada** utiliza mejor el espacio si los datos a almacenar tienen un tamaño pequeño. La dependencia con el factor de carga es mucho más crítica, sin embargo.
  - La **dispersión abierta** tiene una dependencia lineal con el factor de carga (las reestructuraciones pueden ser menos frecuentes).

# Eficiencia TADs Conjunto/Mapa

	Contigua ordenada	Árbol AVL	Tabla de Disp. (promedio)
Pertenencia (conjunto) Acceso por clave (mapa)	$O(\log n)$	$O(\log n)$	$O(1)$
Borrado (por valor/clave)	$O(n)$	$O(\log n)$	$O(1)$
Inserción (por valor)	$O(n)$	$O(\log n)$	$O(1)^*$
Iterar todos los elementos	$O(n)$	$O(n)$	$O(n)$
Unión (ambos tamaño $n$ )	$O(n)$	$O(n \log n)$	$O(n)$

# Eficiencia TAD Diccionario

	Contigua ordenada	Arbol AVL	Tabla Disp (promed.)
Acceso por clave	$O(\log n)$	$O(\log n)$	$O(1)$
Acceso clave $i$ -ésima menor	$O(1)$	$O(\log n)$	$O(n)$
Acceso por iterador	$O(1)$	$O(1)$	$O(1)$
Borrado por clave	$O(n)$	$O(\log n)$	$O(1)$
Borrado clave $i$ -ésima menor	$O(n)$	$O(\log n)$	$O(n)$
Borrado por iterador	$O(n)$	$O(\log n)$	$O(1)$
Inserción por valor	$O(n)$	$O(\log n)$	$O(1)$

# The long and winding road..

