Unidad Didáctica I. Backtracking y Hashing

# ***Tema 2. Hashing (almacenamiento y búsqueda mediante cálculo de dirección basado en clave)***

Algoritmia

Profesor: Andrés Muñoz

Escuela Politécnica





# *Índice*

# *Índice*







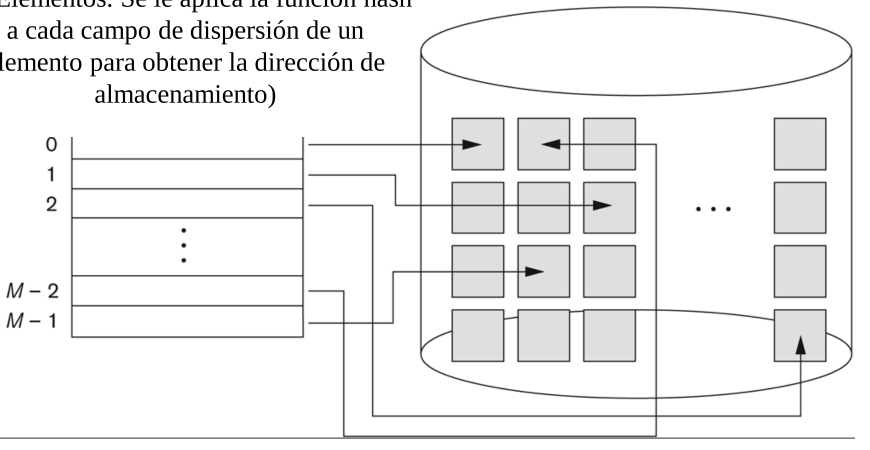
# Introducción

- ✓ La técnica de Hashing consiste en:
  - Almacenar y buscar elementos a través de una estructura de posiciones direccionables dado un campo del elemento (campo de dispersión).
  - Este campo de dispersión es utilizado como índice para almacenar y recuperar el elemento completo
- ✓ Los problemas asociados a la técnica Hashing son:
  - Disponemos de un número elevado de campos clave (K) y un conjunto pequeño de lugares (A) donde guardar el elemento.
  - Necesitamos una función que las relacione

# Introducción

**A** (espacios de almacenamiento)

**K** (Elementos. Se le aplica la función hash a cada campo de dispersión de un elemento para obtener la dirección de almacenamiento)








# Introducción

- ✓ La técnica de Hashing utiliza una función hash (H) para obtener la dirección de almacenamiento de un elemento:
  - La función toma como entrada el valor del campo de dispersión del elemento
  - La función devuelve como salida el espacio de almacenamiento asignado

$$H: K \rightarrow A$$

# Introducción

- ✓ Ejemplo: Suponer el siguiente registro

```
struct empleado {
    int id;
    char nombre[20];
    char apellidos[40];
    char nif[9];
    int sueldo;
    int id_trabajo;
    char departamento[20];
};
```

- ✓ Suponer también que el atributo "*nif*" es elegido como campo de dispersión
- ✓ Suponer que A = [0..99] (100 lugares de almacenamiento)
- ✓ Entonces podemos definir una función hash:

  H: nif → [0..99]

  - Por ejemplo, podemos tomar los dos últimos números del NIF para decidir el lugar de almacenamiento

    H(48482921) = 21
    H(48412907) = 07





# *Índice*

# Funciones Hash

- ✓ Supondremos, sin perdida de generalidad, que:
  - La estructura direccionable es una tabla (array) de tamaño estático y conocido
  - Los elementos que contendrá la tabla hash son registros

- ✓ Hay que tener cuidado con el problema de las colisiones a la hora de seleccionar la función H.





# Funciones Hash

- ✓ **Problema de colisión**: una función de cálculo H provoca *colisión* entre dos claves $k_1$ y $k_2$ si se cumple que $H(k_1) = H(k_2)$
  - En la tabla sólo tenemos 1 hueco para el mismo valor de dispersión!!

- ✓ Por ejemplo, se van a almacenar los socios de un club deportivo mediante técnica Hash:
  - Se parte con una tabla hash de 1000 socios
  - Se utiliza el DNI como clave de dispersión
  - DNI = $10^8$ >> 1000 socios → Varios socios para una misma posición!!

# Funciones Hash

- ✓ Tres estrategias para la definición de funciones Hash:
  - Truncamiento
  - Doblamiento
  - Aritmética modular
  - Otras funciones:
    - Funciones criptográfica: SHA-1, MD5
    - Función de Jenkins
    - Cuckoo hash
    - Robin Hood hash
    - Rayuela (Hopscotch hashing)





# Truncamiento

- ✓ Ignorar una parte de la clave y considerar únicamente el resto.
- ✓ Ejemplo:

H(2**3**.945.**6**6**7**) = 367

# Doblamiento

- ✓ Dividir directamente la clave en varias partes y combinar éstas (sumas, restas, etc.)
- ✓ Si el resultado es mayor que el número de posiciones, se trunca

H(23.945.667) = 23+ 94+56+67=240





# Aritmética Modular

- ✓ Convertir la clave en un valor numérico entero (si no lo es) y calcular el módulo

H(23.945.667)= 23945667 mod1000 = 667

# Implementación de la técnica Hash

**TIPOS Y CONSTANTES**

```
#define LIBRE (un valor adecuado que depende de t_clave, p.e. "-1")
#define N_ELEMS (tamaño de la tabla hash)

typedef struct {
   t_clave clave;
   …
}t_base;

t_base tabla_hash[N_ELEMS];
```





# Implementación de la técnica Hash

**Funciones**

```
void iniciar (t_base tabla_hash[N_ELEMS]){
       for(int i=0; i < N_ELEMS; i++){
              tabla_hash[i].clave = LIBRE;
       }
}

void insertar (t_base tabla_hash[N_ELEMS], t_base reg){
       int p;
       p = H(reg.clave);
       if (tabla_hash[p].clave != LIBRE)
              printf("Colisión");
       else
              tabla_hash[p] = reg;
}
```

# Implementación de la técnica Hash

**Funciones**

```
t_base buscar (t_base tabla_hash[N_ELEMS], t_clave cl){
	int p;
	p = H(cl);
	if (tabla_hash[p].clave == LIBRE) {
		printf("No existe");
		return NULL;
	}
	else if (tabla_hash[p].clave != cl){
		printf("Colisión");
		return NULL;
	}
	else
		return tabla_hash[p];
}
```





# *Índice*

# Alternativas a una colisión

- ✓ Si la posición para un elemento ya está ocupada entonces hay que insertarlo en otra.
- ✓ En la búsqueda de un elemento habrá que tener en cuenta esto.
- ✓ ¿Cómo calcular la nueva posición?
  - Técnicas Hashing Cerrado:
    - Prueba lineal
    - Prueba cuadrática
    - Pruebas dependientes de clave
    - ¿Prueba aleatoria?
  - Técnicas Hashing Abierto: Encadenamiento





# Alternativas a una colisión

- ✓ Todas las alternativas se pueden expresar mediante una función
  - En primer lugar, tenemos $h_0 = H(k)$, dirección candidata para la clave k
  - Si hay colisión, se entra en un bucle en busca de una posición libre en la tabla hash, utilizando una variable *i* para iterar.
  - La variable *i* se inicia a 0, y a continuación se ejecuta la siguiente iteración:

$$i = i + 1;$$

$$h = G(k,i)$$

siendo G(x,y) una función de cálculo del incremento.

  - La función G(x,y) dependerá del tipo de alternativa escogido

# Alternativa I: Prueba lineal

- ✓ Se trata la tabla hash como a una estructura circular: el siguiente elemento después del ultimo es el primero
- ✓ **Inserción:** Cuando ocurre una colisión se debe de recorrer la tabla hash secuencialmente y de forma circular a partir del punto de colisión, buscando el siguiente hueco libre.
- ✓ **Búsqueda:** Si la búsqueda produce colisión se recorre la tabla hash secuencialmente y de forma circular. El proceso concluye cuando el elemento es hallado, o bien cuando se encuentra una posición vacía.





# Alternativa I: Prueba lineal

- ✓ **Función G(k,i)**

  Sin colisión: $h_0$ = H(k) = k mod NELEMS

  Colisión: h= G(k,i) = (H(k) + i) mod NELEMS *i= [0..NELEMS]*

# Alternativa I: Prueba lineal

- ✓ Ventajas:
  - Se exploran todas las direcciones.
  - Sólo se produce fallo si se agotan.
- ✓ Inconvenientes:
  - Se produce agrupamiento alrededor de ciertas claves mientras que otras zonas del arreglo permanecerían vacías.
  - Si las concentraciones de claves son muy frecuentes, la búsqueda será principalmente secuencial perdiendo así las ventajas del método hash **(agrupamiento primario)**
    - 90% ocupación de la tabla → 50 intentos de media para insertar
    - 75% ocupación de la tabla → 8,5 intentos de media para insertar
    - 50% ocupación de la tabla → 2,5 intentos de media para insertar
  - Lo ideal es tener una tabla hash cuyo tamaño sea el doble del número de elementos que queremos insertar…
    - Pero entonces se desperdicia mucha memoria!!!
    - Imagínese la memoria necesaria para almacenar 1.000.000 de registros de personas, donde cada registro son 100 bytes. ¿Cuántos bytes son necesarios si queremos mantener la ocupación al 50%? ¿Y cuántos se utilizan en realidad?





# Alternativa I: Prueba lineal

- ✓ Ejemplo

Insertamos una ‘a’, H(‘a’) = 3

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | a | | | | |

Insertamos una ‘b’, H(‘b’) = 4

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | a | b | | | |

Insertamos una ‘t’, H(‘t’) = 3 → Colisión

i=0;
i = i +1; G(t,1) = H(t) +1 mod 8 = 3+1 mod 8= 4; Colisión, se busca siguiente hueco
i = i +1; G(t,2) = 3+2 mod 8 = 5; No colisión, se inserta ahí

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | a | b | t | | |

Insertamos una ‘c’, H(‘c’) = 4 → Colisión

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | a | b | t | c | |

# Alternativa II: Prueba cuadrática

- ✓ Similar a la prueba lineal, la diferencia es que para el desplazamiento de la inserción y la búsqueda se utiliza **el cuadrado del valor de la iteración** actual

- ✓ **Función G(k,i)**

  Sin colisión: $h_0$ = H(k) = k mod NELEMS

  Colisión: h= G(k,i) = (H(k) + $i^2$) mod NELEMS *i= [0..NELEMS]*

  Ejemplos: $h_0$ +1, $h_0$ +4, $h_0$ +9, $h_0$ +16, …



UCAM | UNIVERSIDAD CATÓLICA SAN ANTONIO

Alternativa II: Prueba cuadrática

- ✓ Ventaja:
  - Evita el agrupamiento primario de la prueba lineal
- ✓ Inconvenientes:
  - No se realiza la prueba en todas las direcciones potencialmente libres.
    - Pueden quedar libres y decir que no las hay!!
  - ¿Cuándo terminar el ciclo? ¿Cómo se sabe que ya se ha recorrido circularmente la tabla una vez para parar?
    - Como máximo para un elemento se iterará N_ELEMS-1 veces. ¿Por qué?
  - Aparece **agrupamiento secundario →** Las claves que colisionen en la misma posición seguirán el mismo camino de saltos, haciéndolo cada vez más largo (más posiciones a recorrer)

# Alternativa III: Dependiente de la clave

- ✓ Los anteriores métodos no han tenido en cuenta la clave.

$$G\ (k,i) = G(i)$$

- ✓ Tiene como ventajas las rapidez y la sencillez.
- ✓ Tiene como inconveniente que si $H(k_1) = H(k_2) = h_0$, siempre se producirá colisión para esas dos claves





# Alternativa III: Dependiente de la clave

- ✓ Una solución reside en hacer que la exploración tras la colisión 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 tras la colisión.
  - 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**
- ✓ Lo habitual es que esa segunda función defina el salto en la exploración

# Alternativa III: Dependiente de la clave

✓ **Función G(k,i)**

Sin colisión: $h_0$ = H(k) = k mod NELEMS

Colisión: h= G(k,i) = (H(k) + **d**·i) mod NELEMS *i= [0..NELEMS]*

✓ Cada nuevo intento explora el hueco situado a una distancia ***d*** a la derecha. 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:

d = max(1, **k** div NELEMS)

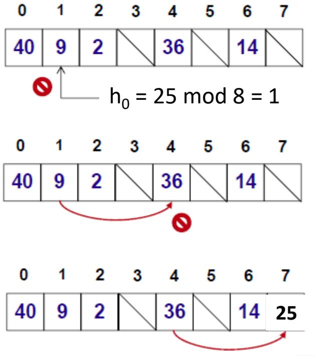




# Alternativa III: Dependiente de la clave

✓ Tabla hash con NELEMS = 8

- Inserción del valor **25**



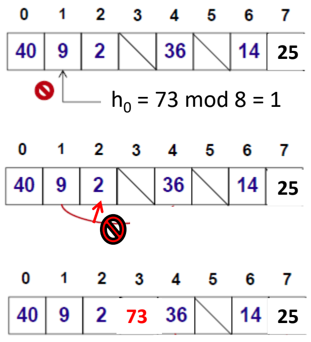

**d** = 25 div 8 = 3

h = G(25,1)= (H(25) + 3*1) mod 8 = 4

h = G(25,2)= (H(25) + 3*2) mod 8 = 7

# Alternativa III: Dependiente de la clave

- ✓ Tabla hash con NELEMS = 8
  - Inserción del valor **73** (mismo $h_0$ que 25)




**d** = 73 div 8 = 9

h = G(73,1)= (H(73) + 9*1) mod 8 = 2

h = G(73,2)= (H(73) + 9*2) mod 8 = 3



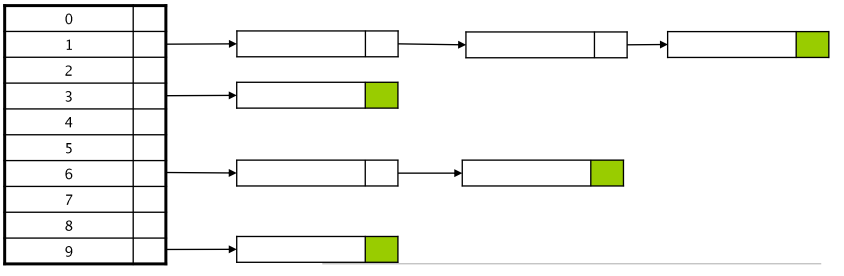


# Alternativa III: Dependiente de la clave

- ✓ Una exploración a base de saltos de *d* celdas no siempre recorrerá 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 NELEMS *y d* son primos entre sí se garantiza un recorrido completo.
  - **Propuesta 1:** Imponer que NELEMS sea un número primo y **d no sea múltiplo de NELEMS.** Si hay que reestructura la tabla para hacerla más grande, escoger el siguiente primo mayor que 2xNELEMS
  - **Propuesta 2:** Imponer que NELEMS sea una potencia de dos, *y* que *d* sea un número impar (si es par, se le suma 1)

# Alternativa IV: Encadenamiento

- ✓ Los registros que colisionen se van añadiendo a una lista asociada a la posición que colisiona
- ✓ Lo habitual es que cada celda almacene un enlace a una lista simplemente enlazada de elementos

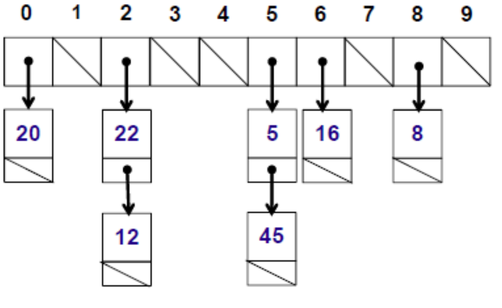





UCAM | Universidad Católica San Antonio

# Alternativa IV: Encadenamiento

- ✓ Ejemplo
  - Secuencia de inserciones: 16, 8, 12, 20, 22, 45, 5




$$h(k, 10) = k \textbf{ mod } 10$$

# *Índice*

- ✓ *Introducción*
- ✓ *Funciones hash y problema de colisión*
- ✓ *Soluciones al problema de colisión*
- ✓ ***Problema de la eliminación***
- ✓ *Redispersión*





# Problema de la eliminación

- ✓ Supongamos que tenemos la siguiente tabla hash que sigue el método de la prueba 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 | | |

# Problema de la eliminación

- ✓ 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 la exploración no se detiene en las casillas vacías, las búsquedas fallidas recorrerían toda la tabla
- ✓ El recolocar los elementos de las rutas rotas causaría nuevas recolocaciones en cascada
- ✓ **Solución: No eliminar el elemento, pero marcar la casilla como** borrada para que la búsqueda no lo encuentre





# Problema de la eliminación

- ✓ Utilizando esta solución, se puede insertar en la casilla borrada si llega un nuevo elemento que debe ir a esa celda
- ✓ Ejemplo

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | | | | 14 | B | 4 | 5 | | |

- ✓ Si ahora se inserta el valor 15, se puede insertar directamente en la celda 5

# *Índice*







# Redispersión

- Cuando la tabla hash se llena (o se supera el umbral de inserción eficiente en las pruebas lineales, cuadráticas, etc) se degrada el rendimiento y es necesario realizar una **redispersión**
- Redispersión → Obtener una nueva tabla hash cuyo tamaño depende de la alternativa de colisión elegida
  - Ej: Siguiente número primo, siguiente potencia de dos
- Lo importante es recordar que se han de pasar los valores de la antigua tabla a la nueva:
  - Por cada elemento en la vieja tabla, se calcula su nueva posición en la nueva tabla y se inserta en ella
  - Si no se recolocan los valores ya insertados se producirán errores a la hora de recuperalos!!

# Redispersión

- Normalmente, la redispersión se aplica cuando:
  - La ocupación de la tabla es superior a 50%
  - Cuando se supera cierto número de colisiones
  - Cada vez que hay una colisión (consume más tiempo)

- La redispersión, aunque necesaria, supone un alto coste:
  - Mayor uso de memoria dinámica
  - Coste en tiempo de ejecución, debido a la recolocación de los elementos y del cálculo del siguiente tamaño de la tabla

- También se puede hacer **redispersión inversa** → Acortar o reducir el tamaño de la tabla hash cuando su ocupación es < 30%





# *Índice*

# Eficiencia. Inserción

- El factor de carga de una tabla hash se denota como "$\alpha$"
- Se calcula como

$$\alpha = N/M$$

donde ***N*** es el num. de celdas ocupadas y ***M*** es el tamaño de la tabla hash

- La probabilidad de que un nuevo registro colisione en el primer intento es igual al factor de carga $\alpha$ (suponiendo que todos los huecos tienen la misma probabilidad de ser accedidos)
  - La probabilidad de colisión en el segundo intento es *N(N-1))/(M(M-1))*
  - La probabilidad de colisión en el intento i-ésimo es *(N(N-1) ... (N-i+1))/(M(M-1) ... (M-i+1))*
  - **Para N y M grandes, la probabilidad de colisión en el intento i-ésimo se toma aproximadamente como: $(N/M)^i$**

- Siguiendo lo anterior, el número de intentos antes de insertar un registro se calcula como

**Num intentos_inserción** $= 1 + \Sigma_{(i=1 \text{to inf})} (N/M)^i =$ $\mathbf{1/(1-\alpha)}$





# Eficiencia. Inserción

- Demostración (No estudiar!)
  - Calculamos $I$ como el número de intentos para una inserción

$$I = 1 + \sum_{i=0}^{\infty} i * P\{\text{debamos hacer exactamente i pruebas } \textit{antes} \text{ de insertar}\}$$

$$= 1 + \sum_{i=0}^{\infty} P\{\text{debamos hacer almenos i pruebas } \textit{antes} \text{ de insertar}\}$$

$$= 1 + \sum_{i=0}^{\infty} \mathrm{p_i}$$

$$p_1 = n/m$$

$$p_2 = \left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right)$$

$$p_i = \left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right) \cdot \left(\frac{n-i+1}{m-i+1}\right) \le \left(\frac{n}{m}\right)^i = \alpha^i$$

$$\therefore I = 1 + \sum_{i=0}^{\infty} \mathrm{p_i} \le 1 + \alpha^1 + \alpha^2 + \alpha^3 + \ldots$$

$$= \frac{1}{1-\alpha}$$

# Eficiencia. Búsqueda

- El factor de carga de una tabla hash se denota como "$\alpha$"
- Se calcula como

$$\alpha = N/M$$

donde ***N*** es el num. de celdas ocupadas y ***M*** es el tamaño de la tabla hash

- El número de intentos antes de encontrar un registro se calcula como

**Num intentos_búsqueda** = **1/$\alpha$ * ln(1/(1- $\alpha$))**

- La secuencia seguida para buscar la clave k es la misma seguida cuando k fue insertada. Se calcular el promedio de pruebas para insertar las claves desde la primera hasta la n-ésima.
- Lo mismo se aplica para el **borrado** (para borrar el elemento primero hay que encontrarlo)





# Eficiencia. Búsqueda

- Demostración (No estudiar!)
  - Si k es la (i+1)ésima clave, el número esperado de intentos para su inserción es 1/(1-i/m). Por lo tanto en número promedio de intentos será la suma de todos los intentos dividida por el número de claves insertadas.

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{1}{\left(1-\frac{i}{m}\right)}=\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{(m-i)}=\frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{(m-i)}=$$

$$=\frac{1}{\alpha}\left(\frac{1}{m}+\frac{1}{m-1}+\frac{1}{m-2}+\frac{1}{m-3}+\ldots+\frac{1}{m-(n-1)}\right)$$

$$=\frac{1}{\alpha}\left(\frac{1}{m}+\frac{1}{m-1}+\frac{1}{m-2}+\frac{1}{m-3}+\ldots+\frac{1}{m-(n-1)}+\left(\frac{1}{m-n}+..+1\right)-\left(\frac{1}{m-n}+..+1\right)\right)$$

$$=\frac{1}{\alpha}\left(\sum_{j=1}^{m}1/j-\sum_{j=1}^{m-n}1/j\right) = \frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$$

# Eficiencia. Ejemplo

- Supongamos una tabla hash con valores M = 10.000 y N = 6550
- Factor de carga → $\alpha$ = 6550/10000 = 0,655 (65,5%)
- Prob. de colisión para un nuevo registro en el primer intento → 65,5%
- Prob. de colisión para un nuevo registro en el segundo intento →
  (6550*6549)/(10000*9999) = 0,429 (42,9%)
- Prob. de colisión para un nuevo registro en el 8º intento →
  $(6550/10000)^8$ = 0,034 (3,4%)

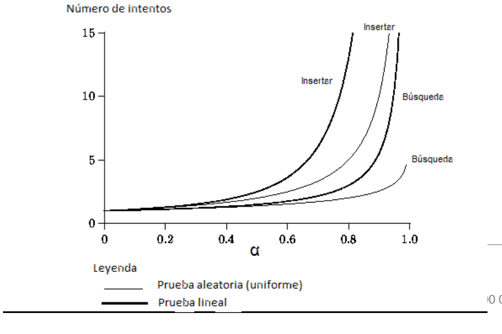
- Num de intentos antes de conseguir insertar un registro en esta tabla –>
  1/(1-0,655) = 2,9, aproximadamente 3
- Num de intentos para encontrar un registro –>
  1/0,655 * ln(2,9) + = 1,62, aproximadamente 2





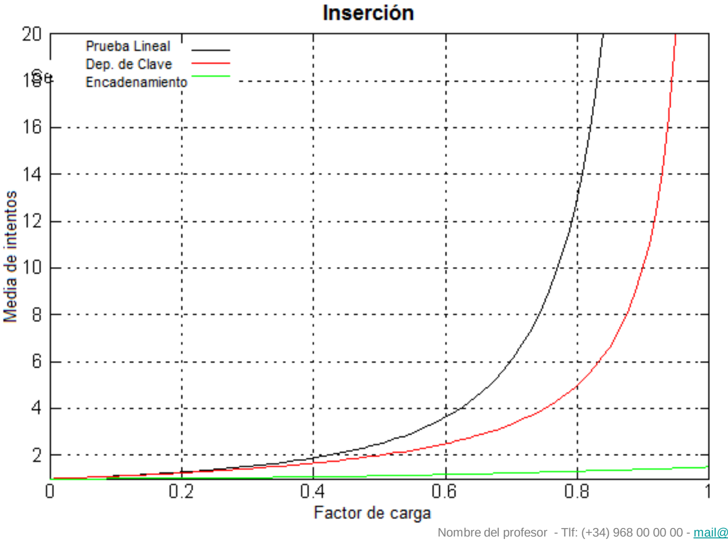
# Eficiencia

- Los cálculos anteriores suponen una distribución uniforme de los accesos a la tabla hash
- Con las técnicas de colisiones lineal y cuadrática no es así (recordad agrupamiento primario y secundario). La eficiencia empeora para esta técnicas
- Para la prueba lineal se estiman los siguiente cálculos
  - Numero de intentos para inserción: $1/2(1 + 1/(1-\alpha)^2)$
  - Numero de intentos para búsqueda: $1/2(1 + 1/(1-\alpha))$
- Comparativa entre distribución uniforme y prueba lineal



# Eficiencia

✓ Gráfica para la operación de insertar con diversos tipos de prueba

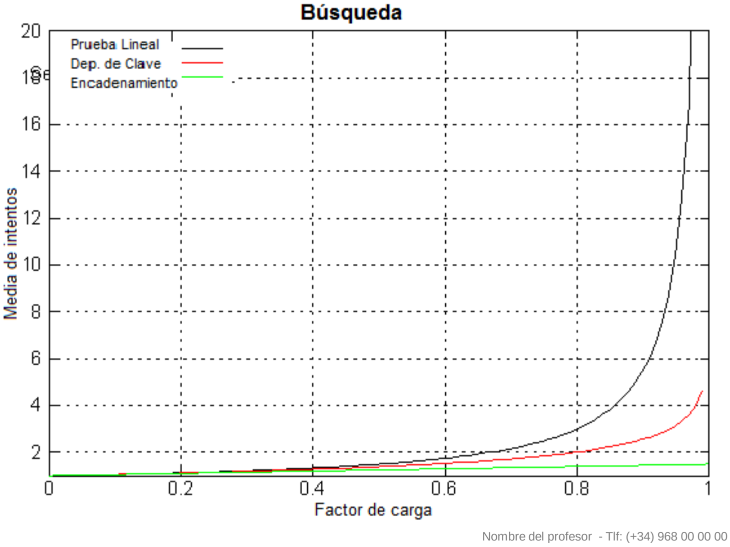







# Eficiencia

✓ Gráfica para la operación de buscar con diversos tipos de prueba



# Eficiencia. Conclusiones

✓ ¿Qué tipo de hashing es mejor, el ***hashing cerrado*** (lineal, cuadrático o doble hashing) o el ***hashing abierto*** (lista enlazada)?

- Ambos tienen un tiempo de inserción y búsqueda O(n) para el peor caso
- Ambos tienen un tiempo de inserción y búsqueda $O(\alpha)$ para el caso medio, cuando $\alpha$ es pequeño ($\alpha < 1$)
- El hashing cerrado usa menos espacio, pero necesita redispersión. Está recomendado para datos almacenados en disco
- El hashing abierto usa más espacio, pero no necesita redispersión (usa memoria dinámica). Está recomendado para datos almacenados en memoria





# Bibliografía

✓ César Vaca Rodríguez**. Tablas de dispersión**. Departamento de Informática. Universidad de Valladolid

✓ Rosa Guerequeta y Antonio Vallecillo. **Técnicas de Diseño de Algoritmos**. Segunda Edición. Servicio de Publicaciones de la Universidad de Málaga, 2000.