# Apuntes de Teórico

# PROGRAMACIÓN 3

# Hash

Versión 1.1

**Índice**

# Introducción

Entre las implementaciones conocidas del TAD Diccionario se puede encontrar:

- Listas
- Arreglos
- ABB

Otra posibilidad para representar diccionarios es la **tabla de dispersión** o **hash**, que permite realizar operaciones de inserción, eliminación y búsqueda en un **tiempo medio constante**.

# Idea general

La estructura de una tabla de dispersión es simplemente un arreglo que contiene el producto cartesiano <clave, información> de los elementos del diccionario (en general la clave será una cadena de caracteres o un número, por ejemplo: nombre, cédula, etc.).
El tamaño de la tabla es un valor que se denomina **TAMAÑO_T**, y se utiliza la convención de que los índices del arreglo se declaren en el rango 0..TAMAÑO_T-1, haciéndose corresponder cada clave de elemento con un valor del rango.

$$h : Claves \rightarrow 0..(TAMAÑO\_T - 1)$$

Esta correspondencia es denominada **función de dispersión** o **función de hash**, la cual debe ser una función **simple de calcular** e idealmente debe asegurar que dos **claves distintas** caigan en **celdas distintas** del arreglo.

$$c_1 \neq c_2 \Rightarrow h(c_1) \neq h(c_2)$$

Sin embargo, como existe un número finito de celdas en el arreglo y potencialmente un número infinito de claves, es prácticamente imposible asegurar que dos claves distintas caigan en celdas distintas, produciéndose **colisiones** de elementos en una misma celda. Por esto es necesario buscar una función de dispersión que **distribuya de manera uniforme (homogéneamente)** las claves de los elementos entre las celdas del arreglo.

$$P\left(h\left(c_j\right) = k\right) = \frac{1}{TAMAÑO_T} \forall j \in 1..\forall k \in 1..TAMAÑO_T$$

Esto es, el resultado de aplicarle la función de dispersión a una clave $(c_j)$ tiene igual probabilidad de ser cualquiera de los TAMAÑO_T valores disponibles.

Dicho de otra forma, cada elemento de la tabla tiene equiprobabilidad de ser ocupado por un elemento de clave $c_j$ .

Vista la idea básica de la estructura tabla de dispersión, resta analizar los problemas de elección de la función de dispersión, resolución de colisiones y valor del TAMAÑO_T.

## Función de dispersión

El hecho de buscar buenas funciones de dispersión es toda un área de investigación. Se verán algunas funciones sencillas, que resultan de utilidad para varios casos:

Si las claves de los elementos son **valores enteros**, una función de dispersión aceptable en casos de claves aleatorias es hallar el **módulo** entre el valor de la clave y el tamaño de la tabla.

**`clave % TAMAÑO_T`**

Esta función es útil salvo que los valores de las claves tengan algunas **particularidades** que impliquen un mayor cuidado. Por ejemplo, si el tamaño de la tabla es 10 y todas las claves (o la mayoría) terminan en 0 (ej.: 0, 10, 20, 30, 40,…) el módulo es una muy mala opción, dado que todos los elementos van al mismo lugar del arreglo.
Una posibilidad para evitar estas situaciones, y que por otras razones generalmente es una buena opción, es asegurarse que el tamaño de la tabla de dispersión sea un **número primo**.

Si por ejemplo las claves de los elementos son **cadenas de caracteres**, una posibilidad para la función de dispersión es sumar los valores ASCII de cada carácter de la cadena.

```
int h (const char * clave, int largo){
     int disp = 0;

     for(int i=0;i<largo;i++)
          disp += clave[i];

     return disp % TAMAÑO_T;
}

a = 97, b=98, c=99, …
→ h(“a”,1) = 97
→ h(“ab”,2) = 195
→ h(“abc”,3) = 294
```

Obteniendo así rápidamente un número y pasando entonces a la situación anterior.

Existen problemas cuando se tienen tablas muy grandes. Por ejemplo:

```
TAMAÑO_T = 10.007
Largo máximo de c/clave = 8
```

Considerando que el máximo valor ASCII es 127

→ `0 <= h(x) <= 1.016`

Lo cual no es homogéneo, dado que las celdas del 1.017 al 10.006 no se usarían.

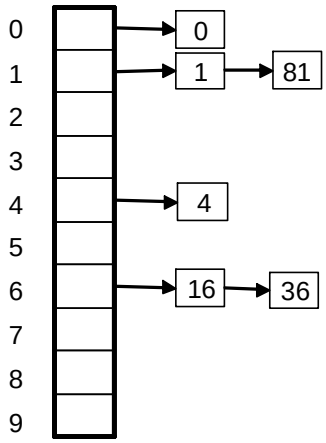
Por lo tanto las funciones de dispersión se deben estudiar para cada caso, dependiendo de la naturaleza de las entradas y del tamaño de la tabla.

# Resolución de colisiones

Otro problema es el relativo a como resolver las colisiones (cuando dos elementos caen en una misma celda del arreglo). Para resolver este problema se verán dos posibilidades: **dispersión abierta** y **dispersión cerrada**.

## *Dispersión Abierta*

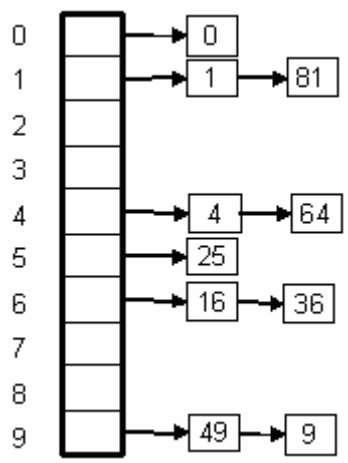
En esta estrategia se resuelven las colisiones utilizando **listas** en las que se mantienen todos los elementos que tienen la misma dispersión.
Por ejemplo, en un hash de tamaño 10 y función h(x) = x % 10.



Si se agrega:

25 → h(25) = 5
64 → h(64) = 4
9 → h(9) = 9
49 → h(49) = 9

Se tiene:




La operación de búsqueda implica evaluar primero la función de dispersión con la clave del elemento, de manera de conocer en cual lista se debe buscar. Luego se recorre la lista hasta llegar al elemento o al final de la misma. La operación de eliminación es similar, dado que implica una búsqueda, en cuanto a la inserción depende que tipo de inserción sobre la lista se realice y de las precondiciones.

```
bool pertenece(Hash tabla, Tipo_Clave clave){
     Lista lista = tabla[h(clave)];
     bool encontre = false;

     while (!encontre && !vaciaLista(lista)){
          Tipo_Elem e = primeroLista(lista);
          encontre = comparar(darClaveElem(e),clave);
          lista = restoLista(lista);
     }

     return encontre;
}
```

Se denomina **factor de carga** (λ) de una tabla de dispersión, a la razón entre la cantidad de elementos de la tabla (**N**) y el tamaño de la misma.

$$\lambda = \frac{N}{TAMAÑO\_T}$$

En el ejemplo anterior, se tiene N=10 y TAMAÑO_T=10, por lo que $\lambda = 1$.

Si la distribución de elementos es homogénea, entonces la longitud media de cada lista es $\lambda$.

Por lo cual una búsqueda lleva tiempo:

1) El tiempo constante (**k**) requerido para evaluar la función de dispersión.
2) El tiempo necesario para recorrer la lista ($\lambda$).

**➔ O(k+$\lambda$) ➔ O(1)**

La regla general en el caso de dispersión abierta es hacer que el **tamaño de la tabla** sea **casi tan grande como el número de elementos esperados**, de manera que $\lambda \cong 1$, y que sea un **número primo**, de manera de obtener una buena distribución.

De esta manera se logra que las operaciones de inserción, eliminación y búsqueda tengan un **tiempo promedio** que es **constante** y "no depende" del tamaño de la entrada (cantidad de elementos).

## *Dispersión Cerrada*

Es una alternativa al uso de una segunda estructura (listas) para la resolución de colisiones. En este caso, cuando ocurren colisiones se resuelven tratando de **buscar una celda libre alternativa** en el arreglo.

La inserción funciona de la siguiente manera: se busca colocar el elemento *x* en la celda *h(x)*, si esta celda ya tiene un elemento (colisión) se debe disponer de una estrategia de **redispersión**, buscando colocar el elemento en una sucesión de celdas $h_0(x)$, $h_1(x)$, $h_2(x)$, … Se prueba en cada una de éstas celdas, en orden, hasta encontrar una vacía, si no se encuentra la tabla de dispersión está llena y no es posible insertar el elemento x.

Es importante recalcar que, como todos los elementos se colocan en la tabla de dispersión, es necesario que el **tamaño** de la misma sea **mayor que en el caso de dispersión abierta**. En general el factor de carga debería ser: $\lambda \leq 0.5$.

Formalizando:

Al insertar un elemento *x* se intenta colocarlo en una sucesión de celdas $h_0(x)$, $h_1(x)$, $h_2(x)$, …

Donde $h_i(x) = ( h(x) + f(i) ) \% TAMAÑO\_T$
con $f(0) = 0$

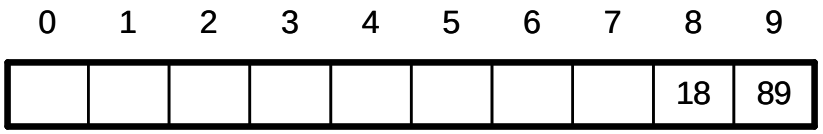
La función *f(x)* es la estrategia de redispersión o estrategia de resolución de colisiones.

## Redispersión Lineal

Por ejemplo, en un hash de tamaño 10 y función h(x) = x % 10

Si se agrega:

```
89 → (9 + 0) % 10 = 9
18 → (8 + 0) % 10 = 8
```

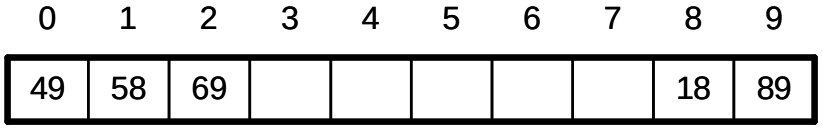



Al insertar el 49 se produce una colisión, ya que la celda 9 está ocupada.

Una de las estrategias de redispersión más simples es la **lineal**, en donde ***f(i)=i***.
Entonces la sucesión de celdas donde se busca insertar sería: $h_i=(h(49)+i)\%10$, que es: 9, 0, 1, 2, etc.

Entonces, si se agrega:

49 → 9 ocupado → 0
58 → 8 ocupado → 9 ocupado → 0 ocupado → 1
69 → 9 ocupado → 0 ocupado → 1 ocupado → 2




Nota: En el caso de usar dispersión cerrada las operaciones de búsqueda y eliminación son un poco más complejas que en el caso de dispersión abierta.

Las pruebas de pertenencia de un elemento *x* requiere examinar la celda *h(x)* y las celdas sucesivas hasta encontrar *x* o una celda vacía.

Simplificando el problema y suponiendo que **no** se permiten **eliminaciones**. Si $h_3(x)$ es la primera celda vacía encontrada, entonces no es posible que el elemento *x* se encuentre en las celdas $h_4(x)$, $h_5(x)$ o más adelante, porque *x* no se podría haber colocado en estas celdas a menos que $h_3(x)$ estuviese ocupado al momento de insertarlo. Por ejemplo, el 58 se busca en 8, 9, 0 y se encuentra en 1; pero el 78 se busca en 8, 9, 0, 1, 2 y, como la celda 3 es vacía, no se encuentra.

Sin embargo, si se permiten las **eliminaciones** nunca puede existir la certeza al encontrar una celda vacía, sin haber encontrado el elemento *x*, que *x* no se encuentre en otra celda y la celda hubiese estado ocupada al momento de su inserción. Por ejemplo, si se borra el 58, al buscar posteriormente el 69 se pasa por 9, 0 y al llegar a la celda 1 se

presume que el elemento no está.

Por esto, cuando se realizan eliminaciones es necesario **marcar** la celda de manera que indique que en la misma se ha eliminado un elemento. De la misma manera es necesario marcar las celdas que no se han utilizado aún.

En la estructura de datos asociada a cada celda de la tabla se tendrá el elemento *x* y un valor de tipo enumerado que indique lo siguiente:

**Vacío** → la celda nunca ha sido usada
**Suprimido** → la celda contenía un elemento que ha sido borrado
**Ocupado** → la celda contiene un elemento

Por ejemplo, si tenemos el siguiente hash, todas las celdas se inician vacías:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| vacio | vacio | vacio | vacio | vacio | vacio | vacio | vacio |
| | | | | | | | |

Al insertar:

*h(a) = 3*
*h(b) = 0*
*h(c) = 4*
*h(d) = 3 colisión → 4 colisión → 5*

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| ocup. | vacio | vacio | ocup. | ocup. | ocup. | vacio | vacio |
| b | | | a | c | d | | |

Buscando *e*, con *h(e) = 4* :
En 4 no se encuentra → busco en 5, no se encuentra → busco en 6, vacio → no está

Al eliminar *c*:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| ocup. | vacio | vacio | ocup. | supr. | ocup. | vacio | vacio |
| b | | | a | | d | | |

Si ahora buscamos *d*, con *h(d)=3*:
En 3 no se encuentra → busco en 4, elemento suprimido → busco en 5, encuentro d.
Notar que de no tener estas marcas se hubiera parado en 4.

A continuación se verán algunas operaciones, la tabla se debe inicializar con todas las marcas en vacío.

```
/* Retorna el índice donde se detiene la búsqueda, esto puede ser
por: recorrer todo el arreglo, encontrar el elemento ó encontrar
celda vacía. */
int buscar(Hash tabla, Tipo_Clave clave){
     int ini = h(clave);
     int i   = 0;
     int pos = ini;
     while ((i < TAMAÑO_T) && (darMarca(tabla[pos]) <> VACIO)) {
          if ((darMarca(tabla[pos]) == SUPRIMIDO) ||
                    !cmpClaves(darClave(tabla[pos]), clave)){
               i++;
               pos = (ini+i)%TAMAÑO_T; // redispersión lineal
           }
     }
     return pos;
}
```

```
bool pertenece (Hash tabla, Tipo_Clave clave){
     int pos = buscar(tabla, clave);
     if ((darMarca(tabla[pos]) <> VACIO) &&
                              (darMarca(tabla[pos]) <> SUPRIMIDO))
          return cmpClaves(darClave(tabla[pos]),clave);
     else return false;
}
```

```
void eliminar (Hash tabla, Tipo_Clave clave){
     int pos = buscar(tabla, clave);
     if((darMarca(tabla[pos]) <> VACIO) &&
                 (darMarca(tabla[pos]) <> SUPRIMIDO) &&
                  cmpClaves(darClave(tabla[pos]), clave))
          marcar(tabla[pos], SUPRIMIDO);
}
```

Para la inserción es necesario que la función de localización verifique si el elemento ya se encuentra y devuelva el primer lugar marcado como VACIO o SUPRIMIDO.

```
// Pre-condición: la tabla no está llena
void insertar (Hash tabla, Tipo_Clave clave, Tipo_Elemento e){
     int pos = buscar(tabla, clave);
     if((darMarca(tabla[pos]) == OCUPADO) &&
                    cmpClaves(darClave(tabla[pos]), clave))
          printf("¡Ya existe el elemento en la tabla!");
     else {
          //se detiene también con SUPRIMIDO.
          pos = buscar2(tabla, clave);
          agregarElem(tabla[pos], e);
          marcar(tabla[pos], OCUPADO);
     }
}
```

La **eficiencia** o tiempo de la inserción y de las otras operaciones no solo depende de la función de dispersión y como ésta distribuye los elementos en las celdas, sino también de cómo la estrategia de redispersión evita **colisiones adicionales**.

La estrategia lineal de redispersión presenta un problema debido a lo siguiente: tan pronto como se llenen unas cuantas celdas consecutivas, cualquier clave que se disperse en una de ellas será enviada al final de las mismas. Este efecto se denomina **agrupamiento primario** y el tamaño de estos grupos tiende a crecer. De esta forma, es probable encontrar sucesiones cada vez más largas de celdas ocupadas que si se llenaran **aleatoriamente**, lo cual influye en el tiempo de ejecución de las operaciones.

No es simple saber cuantos intentos son necesarios realizar en promedio para insertar un elemento cuando *N* de las TAMAÑO_T celdas están llenas (asumiendo aleatoriedad en el llenado, dado que es la mejor situación posible).

Sin realizar los cálculos, se puede demostrar que el número de intentos para la resolución lineal es:

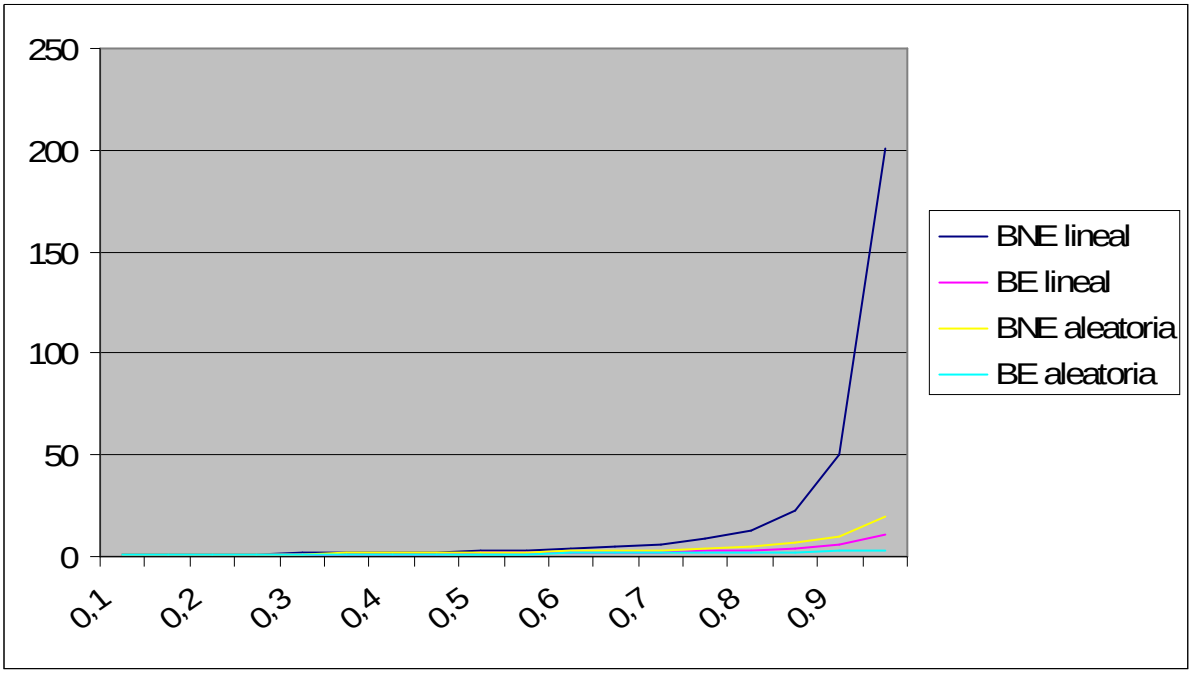
Inserciones y búsquedas no exitosas: $\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$

Búsquedas exitosas: $\frac{1}{2}\left(1+\frac{1}{(1-\lambda)}\right)$

Y si se realiza una estrategia de redispersión aleatoria, en caso de disponer de tal estrategia:

Inserciones y búsquedas no exitosas: $\left(\frac{1}{1-\lambda}\right)$

Búsquedas exitosas: $\frac{1}{\lambda}\ln\left(\frac{1}{1-\lambda}\right)$




## Redispersión Cuadrática

La redispersión cuadrática es un método de resolución de colisiones que elimina el problema de agrupamiento primario que presenta el método anterior. La elección común es: $f(i) = i^2$ .

**E**n la redispersión lineal es una mala idea dejar que la tabla de dispersión esté casi llena, porque se degrada el rendimiento. En este caso la situación es aún más drástica, dado que **no hay garantía de encontrar una celda vacía** si la tabla se llena a **más de la mitad**, o aún antes si el tamaño de la tabla no es primo.

Por ejemplo, si se considera la siguiente tabla:

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| ocup. | ocup. | vacio | vacio |
| 4 | 1 | | |

Si se quiere insertar el 8:

*h(8)=0, ocupado → (0+1)%4=1, ocupado → (0+4)%4=0, ocupado → (0+9)%4=1, ocupado → (0+16)%4=0, ocupado → …*

Hay un teorema, que no se demostrará, que dice:

> Si se utiliza redispersión cuadrática y el tamaño de la tabla de dispersión es primo, entonces siempre se puede insertar un elemento nuevo si la tabla de dispersión está, al menos, medio vacía.

Existe en esta estrategia un **agrupamiento secundario**, dado que los elementos que se dispersan a la misma posición probarán en las mismas celdas alternas.

### Redispersión Doble

En este método se aplica una **segunda función de dispersión**. Entonces usualmente se tiene: ***f(i) = i * h'(x)***. La función de dispersión nunca debe evaluarse a 0, y se debe asegurar que se puede probar con todas las celdas.

Esta estrategia es comparable, en número de intentos, a la estrategia aleatoria, lo que la hace muy interesante. Sin embargo la redispersión cuadrática no requiere de una segunda función de dispersión y es por esto más simple y práctica.

## Reestructuración de las tablas de dispersión

Si se utiliza una tabla de dispersión abierta el tiempo medio de las operaciones crece con (N / TAMAÑO_T), número que crece con rapidez a medida que la cantidad de elementos excede la cantidad de celdas.
En el caso de dispersión cerrada la eficiencia disminuye a medida que N se acerca a TAMAÑO_T y no es posible que lo exceda.

Para mantener el tiempo constante por operación, se sugiere que si N crece demasiado se utilice una nueva tabla de dispersión cuyo tamaño sea el número primo más cercano al doble del tamaño de la tabla anterior. Esta operación se denomina **reestructuración** de la tabla de dispersión, y es de *O(n)*.