

Estructuras de Datos y Algoritmos

El TAD tabla y las tablas dispersas

LECCIÓN 19

© All wrongs reversed – bajo licencia [CC-BY-NC-SA 4.0](#)



Universidad
Zaragoza

Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza, España

Grado en Ingeniería Informática
UNIVERSIDAD DE ZARAGOZA



Adaptadas de diapositivas de Javier Campos

Índice

- 1 Conceptos
- 2 Especificación
- 3 Tablas dispersas (o *tablas hash*)
- 4 Recomendaciones finales

Índice

1 Conceptos

2 Especificación

3 Tablas dispersas (o *tablas hash*)

4 Recomendaciones finales

TAD tabla y tablas dispersas

Conceptos

- Una tabla es un conjunto o colección de pares $\langle c, v \rangle$
 - c se denomina *clave*, y v se denomina *valor asociado* a la clave c
 - En la colección no podrá haber dos pares con la misma clave
- Las tablas se denominan también *diccionarios*, *tipos de datos asociativos* o *tipos funcionales* por **representar una función cuyo dominio es el género de las claves y cuyo rango es el género de los valores**
 - La función suele ser parcial (es decir, existen claves sin valor asociado)
 - Podría considerarse total si se admitiese un valor especial “indefinido”
- Operaciones básicas para las tablas:
 - Inserción/modificación (del valor) de un nuevo par $\langle c, v \rangle$
 - Obtención del valor v asociado a una clave c (operación parcial)
 - Borrado, dada una clave c , eliminar el par $\langle c, v \rangle$

Índice

1 Conceptos

2 Especificación

3 Tablas dispersas (o *tablas hash*)

4 Recomendaciones finales

TAD tablas

Vista en lección 10, con el nombre diccionario

```
espec tablasGenéricas
  usa booleanos, naturales
  parámetros formales
    géneros clave, valor
    operación
    {suponemos que en el género de las claves hay definidas funciones de comparación "=", "<"}
    _= : clave c1, clave c2 -> booleano
    _< : clave c1, clave c2 -> booleano

fpf
género tabla
{Los valores del TAD representan conjuntos de pares (clave,valor) en los que
 no se permiten claves repetidas}
operaciones
  crear: -> tabla
  {Devuelve una tabla vacía, sin elementos}
  añadir: tabla t, clave c, valor v -> tabla
  {Si en t no hay ningún par con clave c, devuelve la tabla resultante de añadir el par (c, v) a t;
   si en t hay un par (c, v'), entonces devuelve el resultado de sustituirlo por el par (c, v)}
  pertenece?: clave c, tabla t -> booleano
  {Devuelve verdad si y sólo si en t hay algún par (c, v)}
  parcial obtenerValor: clave c, tabla t -> valor
  {Devuelve el valor asociado a la clave c en t.
   Parcial: la operación no está definida si c no está en t}
  quitar: clave c, tabla t -> tabla
  {Si c está en t, devuelve la tabla resultante de borrar c y su valor de t;
   si c no está en t, devuelve una tabla igual a t}
  cardinal: tabla t -> natural
  {Devuelve el número de elementos en la tabla t}
  esVacio?: tabla t -> booleano
  {Devuelve verdad si y sólo si d no tiene elementos}

fespec
```

TAD tablas

Algunas implementaciones ya vistas

■ Representación de acceso directo:

- Vector v $[1..max]$ de valores
- Debe existir una función inyectiva que codifica la clave en un entero comprendido en el rubrango de $1..max$ (max , razonablemente pequeño)

TAD tablas

Algunas implementaciones ya vistas

■ Representación de acceso directo:

- Vector v $[1..max]$ de valores
- Debe existir una función inyectiva que codifica la clave en un entero comprendido en el rubrango de $1..max$ (max , razonablemente pequeño)

■ Representación alfabética u ordenada

- Vector v $[1..max]$ de pares $\langle c, v \rangle$ ordenados según c
- Exige existencia relación de orden total definida sobre c
- Número de pares en la tabla acotado por max
- Útil para tablas poco dinámicas (pocas modificaciones), pero ineficiente si tablas muy dinámicas (inserciones y borrados tienen coste lineal)

TAD tablas

Algunas implementaciones ya vistas

■ Representación de acceso directo:

- Vector v $[1..max]$ de valores
- Debe existir una función inyectiva que codifica la clave en un entero comprendido en el rubrango de $1..max$ (max , razonablemente pequeño)

■ Representación alfabética u ordenada

- Vector v $[1..max]$ de pares $\langle c, v \rangle$ ordenados según c
- Exige existencia relación de orden total definida sobre c
- Número de pares en la tabla acotado por max
- Útil para tablas poco dinámicas (pocas modificaciones), pero ineficiente si tablas muy dinámicas (inserciones y borrados tienen coste lineal)

■ Representación mediante árboles binarios de búsqueda:

- Pares almacenados en un árbol binario de búsqueda (normalmente equilibrado para acotar el coste peor en $\mathcal{O}(\log N)$) ordenado

Índice

- 1 Conceptos
- 2 Especificación
- 3 Tablas dispersas (o *tablas hash*)
- 4 Recomendaciones finales

Algunas implementaciones ya vistas

Costes caso peor

Representación	insertar	cambiar	elValor	borrar
Acceso directo	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Alfabética	$\mathcal{O}(N)$	$\mathcal{O}(\log N)$	$\mathcal{O}(\log N)$	$\mathcal{O}(N)$
ABB equilibrado	$\mathcal{O}(\log N)$	$\mathcal{O}(\log N)$	$\mathcal{O}(\log N)$	$\mathcal{O}(\log N)$

Tablas dispersas (o *tablas hash*)

Acceso directo

■ La mejor representación en cuanto a costes en tiempo, pero...

- Debe existir una función inyectiva que codifica la clave en un entero comprendido en el subrango de $1..max$ (con max pequeño y conocido)
vector $[1..max]$ de valores
- No siempre es posible encontrar tal función inyectiva de codificación de las claves en un subrango pequeño de enteros
- Normalmente, sí es posible encontrar una función h no inyectiva

$$h : \mathcal{D}_{claves} \mapsto 1..max \quad \text{o bien} \quad h : \mathcal{D}_{claves} \mapsto 0..max - 1$$

Tablas dispersas (o *tablas hash*)

Acceso directo

■ La mejor representación en cuanto a costes en tiempo, pero...

- Debe existir una función inyectiva que codifica la clave en un entero comprendido en el subrango de $1..max$ (con max pequeño y conocido)
vector $[1..max]$ de valores
- No siempre es posible encontrar tal función inyectiva de codificación de las claves en un subrango pequeño de enteros
- Normalmente, sí es posible encontrar una función h no inyectiva

$$h : \mathcal{D}_{claves} \mapsto 1..max \quad \text{o bien} \quad h : \mathcal{D}_{claves} \mapsto 0..max - 1$$

- Habitualmente, se busca una función no inyectiva que distribuya las claves de la forma más uniforme posible en el rango $1..max$ (o $0..max - 1$; índices del vector)
 - Haciendo que la probabilidad de que $h(c_1) = h(c_2)$ para dos claves $c_1 \neq c_2$ sea lo más baja posible
 - Si $h(c_1) = h(c_2)$ y $c_1 \neq c_2$, se dice que se ha producido una *colisión*

Tablas dispersas (o *tablas hash*)

Definición

- Representación del TAD tabla (o diccionario) basada en la utilización de una función de codificación no inyectiva para acceder a la posición de un vector en la que almacenar cada par $\langle c, v \rangle$
- Para definir una tabla dispersa, se han de tomar dos decisiones, independientes:
 - 1 Elegir una función h no inyectiva. Se le denomina **función de localización** (desmenuzamiento, transformación, hash, dispersión, etc.) – en inglés, *hashing function*
 - Dada una clave, la función h devolverá la entrada (componente) del vector que corresponde a dicha clave: **entrada primaria**
 - Cuando a una clave c_2 le corresponde una entrada primaria que ya está ocupada por otro par $\langle c_1, v \rangle$, con $c_1 \neq c_2$, se dice que se produce una **colisión**
 - 2 Seleccionar un método para la resolución de colisiones

Tablas dispersas (o *tablas hash*)

Elección de función de *hashing*

Características que debe tener

- Debe distribuir las claves esperadas de la forma más uniforme posible sobre el rango $0..max - 1$, o el usado como índice del vector
 - Provocará pocas colisiones
- Debe poder ser evaluada de forma eficiente (por ejemplo, con operaciones aritméticas sencillas)

Tablas dispersas (o *tablas hash*)

Elección de función de *hashing* – funciones habituales

Para claves enteras

■ Método de la división (o del módulo)

- $h(c) = c \bmod \max$ (los índices del vector son $0.. \max - 1$)
- Puede funcionar mal. Por ejemplo, si $\max = 10$ y casi todas las claves acaban en el mismo dígito
- Es conveniente elegir un número primo para valor de $\max$

■ Método del centro del cuadrado

- $h(c) = f(c)$, siendo $f(c)$ los $\log_2 \max$ dígitos centrales de c^2 (es decir, se eleva al cuadrado la clave, y se extraen unos dígitos de la parte central del número resultante)

Véanse ejemplos en: <http://webdiis.unizar.es/%7Eelvira/eda/addeda/modulos/tablas/c43.html>

Tablas dispersas (o *tablas hash*)

Elección de función de *hashing* – funciones habituales

Para claves del tipo cadena de caracteres

■ Método de la suma de ordinales y división

- 1 Dada una cadena s , sumar los valores de $\text{ord}(s[i])$, $\forall i \in [1, \text{long}(s)]$
- 2 Calcular el resto de la división del resultado anterior por max

- Puede aplicarse por fragmentos de la cadena
- No funciona bien si max es demasiado grande, o si no distribuye las claves esperadas de forma uniforme. Ejemplos:
 - Si $\text{max} = 10007$ y las cadenas tienen todas 8 o menos caracteres
→ $256 \cdot 8 = 2048$ posiciones utilizables
 - Si $\text{max} = 100$ y las claves son "A0" . . "A99": se concentrarían en 29 componentes (habría muchas colisiones; es decir, a muchas claves se les asignaría el mismo valor. Por ejemplo, "A18", "A27", "A36", ..., "A90")

■ Otro método: **suma ponderada y división**

- Calcular la suma ponderada de los valores $\text{ord}(s[i])$, $\forall i \in [1, \text{long}(s)]$ (por ejemplo, interpretando la cadena como si fuera un entero en base 256)

Tablas dispersas (o *tablas hash*)

Implementación

constantes

```
cardinalCódigo = 256 {Suponemos el código ASCII, con 256 valores}
max = ... {El tamaño del vector con el que implementar}
```

función `h(c: cadena)` devuelve `0..max - 1`

variables

```
i, suma, factor: entero
```

principio

```
suma := 0; {Para calcular la suma ponderada explicada antes}
```

```
factor := 1; {Valor inicial de factor =  $\text{cardinalCódigo}^0 = 1$ }
```

```
para i:=1 hasta long(c) hacer
```

```
    suma := suma + ord(c[i])*factor;
```

```
    factor := factor*cardinalCódigo {factor =  $\text{cardinalCódigo}^i$ }
```

```
fpara;
```

```
devuelve(suma mod max) {Método de la división}
```

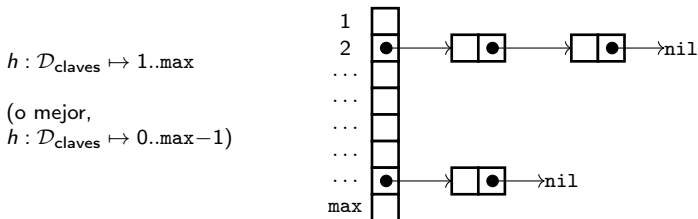
fin

Tablas dispersas (o *tablas hash*)

Resolución de colisiones

Por encadenamiento (o *dispersión abierta*)

- Situar los pares que colisionan sobre una entrada primaria en una zona de desbordamiento
- La zona de desbordamiento puede consistir en una estructura de datos dinámica (por ejemplo, una lista)
 - La componente del vector guarda el puntero a la zona de desbordamiento (lista), y ésta contiene todos los pares que colisionan en esa componente



Tablas dispersas (o *tablas hash*)

Resolución de colisiones por encadenamiento – implementación

módulo tablasDispersas

importa defClave, defValor {donde estén definidos los tipos clave y valor}

exporta

tipo tabla

procedimiento crear(sal t: tabla)

{Post: devuelve una tabla vacía}

procedimiento añadir(e/s t: tabla; ent c: clave; ent v: valor)

{Post: devuelve la tabla resultante de añadir el par (c, v) a t;

si en t ya había un par (c, v'), entonces lo sustituye por el par (c, v)}

procedimiento buscar(ent t: tabla; ent c: clave; sal éxito: booleano; sal v: va

{pertenece? y obtenerValor se implementan en una única operación}

procedimiento quitar(ent c: clave; e/s t: tabla)

{Post: dada una clave c, la borra de la tabla junto con su valor;

si c no estaba en t, t se queda igual}

implementación

constante

max = 1000 {por ejemplo}

tipos

ptNodo = ↑nodo;

nodo = registro

laClave: clave;

elValor: valor;

sig: ptNodo

freg;

tabla = vector[0..max - 1] de ptNodo

...

```

...
función h(c: clave) devuelve 0..max - 1
{Función de localización (hashing) que distribuye uniformemente el dominio de las
 claves en el subrango 0..max - 1}
...

procedimiento crear(sal t:tabla)
variable
    i: 0..max - 1
principio
para i := 0 hasta max - 1 hacer
    t[i] := nil
fpara
fin

procedimiento añadir(e/s t: tabla; ent c: clave; ent v: valor)
variables
    aux: ptNodo; entrada: 0..max - 1
principio
    entrada := h(c);
    si t[entrada] = nil entonces
        nuevoDato(t[entrada]);
        aux := t[entrada];
        aux↑.laClave := c;
        aux↑.elValor := v;
        aux↑.sig := nil
    sino { t[entrada] ≠ nil }
        aux := t[entrada]; {buscar en la zona de desbordamiento (no ordenada):}
        mientrasQue(aux↑.laClave ≠ c) and (aux↑.sig ≠ nil) hacer
            aux := aux↑.sig
        fmq;
        si aux↑.laClave = c entonces
            aux↑.elValor := v
        sino
            nuevoDato(aux↑.sig);
            aux := aux↑.sig;
            aux↑.laClave := c;
            aux↑.elValor := v;
            aux↑.sig := nil
        fsi
    fsi
fin
...

```

```

...
procedimiento buscar(ent t: tabla; ent c: clave; sal éxito: booleano; sal v: valor)
variable
    aux: ptNodo
principio
    aux := t[h(c)];
    si aux = nil entonces
        éxito := f also
    sino {buscar en la zona de desbordamiento (no está ordenada):}
        mientrasQue(aux↑.laClave ≠ c) and (aux↑.sig ≠ nil) hacer
            aux := aux↑.sig
        fmq;
        si aux↑.laClave = c entonces
            éxito := verdad;
            v := aux↑.elValor
        sino
            éxito := falso
        fsi
    fsi
fin

procedimiento quitar(ent c: clave; e/s t: tabla)
variables
    entrada: 0..max - 1; aux, sigAux: ptNodo; borrado: booleano
principio
    entrada := h(c);
    aux := t[entrada];
    si aux ≠ nil entonces {lista no vacía}
        si aux↑.laClave = c entonces {borrar el primero}
            t[entrada] := aux↑.sig;
            disponer(aux);
            borrado := verdad
        sino {borrar otro o ninguno}
            borrado := falso; {buscar en el resto de la zona (no está ordenada):}
            mientrasQue (aux↑.sig ≠ nil) and not borrado hacer
                si aux↑.sig↑.laClave = c entonces
                    sigAux := aux↑.sig;
                    aux↑.sig := sigAux↑.sig;
                    disponer(sigAux);
                    borrado:=verdad
                sino
                    aux := aux↑.sig
            fsi
        fmq
    fsi
fin

fin {del módulo tablasDispersas (resolución de colisiones por encadenamiento))}

```

Tablas dispersas (o *tablas hash*)

Resolución de colisiones – Factor de carga

- En general, si en una tabla de tamaño max , se almacenan N pares, y la función de localización es uniforme:
 - El número medio de pares para cada entrada primaria es $\frac{N}{\text{max}}$ (*tamaño medio de las zonas de desbordamiento*)
- El **factor de carga** de la tabla se define como $\alpha = \frac{N}{\text{max}}$
 - Si el factor de carga es pequeño, las listas de desbordamiento son cortas, y la resolución de colisiones por encadenamiento es bastante rápida (y, por supuesto, α puede ser > 1 , si $N > \text{max}$)
- Se puede calcular¹ el número medio de comparaciones en una operación de búsqueda:
 - Si la búsqueda no tiene éxito: $C = \left(1 - \frac{1}{\text{max}}\right)^N + \frac{N}{\text{max}} \approx e^{-\alpha} + \alpha$
 - Si la búsqueda tiene éxito: $C = 1 + \frac{N-1}{2\text{max}} \approx 1 + \frac{1}{2}\alpha$
- Es decir, **el coste medio es del orden del factor de carga de la tabla**

¹Véase el libro de Donal Knuth en la bibliografía de la asignatura para más detalles.

Tablas dispersas (o *tablas hash*)

Resolución de colisiones

Por recolocación en el mismo vector (o *dispersión cerrada*)

- Los pares que colisionan se colocan en otras posiciones del vector, diferentes a la entrada primaria
 - $\alpha = \frac{N}{\max}$ y siempre $\alpha \leq 1$ (no puede haber datos fuera del vector)
- Se utiliza un algoritmo de recolocación (h_2 o hh) para calcular una secuencia de entradas secundarias para una clave, hasta encontrar una posición libre en el vector
 - Se calcula una entrada secundaria con cada llamada a hh ; si está libre se utiliza, si no está libre, se calcula otra entrada secundaria, etcétera
Seguir buscando posiciones hasta cierto número de intentos, o estar seguro de que si hubiese alguna posición libre accesible se habría encontrado
 - Se genera una secuencia de saltos o posiciones a comprobar para c :

$$P_i(c) = ((h(c) + hh(i)) \bmod \max)$$

donde i representa el i -ésimo intento o posición a probar y $\max$ el tamaño del vector

- **Inconveniente:** los pares recolocados ocupan las posiciones de otras futuras claves
 - Aumentan las colisiones (ahora sobre entradas primarias y secundarias)
Y por tanto, aumentará el coste promedio de las operaciones

Tablas dispersas (o *tablas hash*)

Resolución de colisiones por recolocación en el mismo vector

... {mismo inicio que implementación de resolución de colisiones por encadenamiento}
constante

max = 1000 {por ejemplo} {Siempre $\alpha = \frac{N}{\text{max}} \leq 1$ }

tipos

```
componente = registro
    libre : booleano;
    laClave: clave;
    elValor: valor;

freg;
tabla = vector[0..max - 1] de componente
```

procedimiento buscar(ent t: tabla; ent c: clave; sal éxito: booleano; sal v: valor)

variables

```
primaria, secundaria: 0..max - 1;
i: 1..max
```

principio

```
primaria := h(c);
si t[primaria].libre entonces
    éxito := falso
sino
    si t[primaria].laClave = c entonces
        éxito := verdad;
        v := t[primaria].elValor
    sino { t[primaria].laClave ≠ c }
        i := 1; {contador de intentos de recolocación}
        secundaria := (primaria + hh(i)) mod max;
        mientrasQue (not t[secundaria].libre and (t[secundaria].laClave ≠ c) hacer
            i := i + 1;
            secundaria := (primaria + hh(i)) mod max
        fmq;
        si t[secundaria].libre entonces
            éxito := falso
        sino
            éxito := verdad;
            v := t[secundaria].elValor
    fsi
fsi
```

¡Ojo!
Evitar ciclos infinitos

Tablas dispersas (o *tablas hash*)

Resolución de colisiones por recolocación en el mismo vector

$$P_i(c) = ((h(c) + hh(i)) \bmod \max)$$

Funciones $_{hh}$ habituales

■ **Recolocación simple:** $hh(i) = i$

- *Problema:* los elementos tienden a “amontonarse” alrededor de las entradas primarias
- Efecto denominado agrupamiento o amontonamiento primario: *las secuencias de posiciones ocupadas consecutivas tienden a hacerse cada vez más grandes* (posiciones cercanas a posiciones ya ocupadas tienen mayor probabilidad de estar ocupadas que otras posiciones más aisladas)

■ **Recolocación lineal:** $hh(i) = k \cdot i$ (si $k = 1$, recolocación simple)

- Es necesario que k y $\max$ sean primos entre sí para garantizar que se puede acceder a todas las posiciones de la tabla
- Pueden producirse amontonamientos (primarios y secundarios)
- *Agrupamientos secundarios:* idénticas secuencias de búsquedas para claves con mismo $h(c)$

Tablas dispersas (o *tablas hash*)

Resolución de colisiones por recolocación en el mismo vector

Funciones $_{hh}$ habituales

- **Recolocación cuadrática:** $hh(i) = i^2$
 - Minimiza amontonamientos, **pero** puede no encontrar huecos libres (existentes pero inaccesibles)
El problema se reduce si m es primo
 - *Agrupamientos secundarios:* idénticas secuencias de búsquedas para claves con mismo $h(c)$
- **Recolocación por doble hashing** (o doble dispersión): $hh(i) = i \cdot \text{hash2}(c)$
 - Función hash secundaria para la clave, que vuelve a dispersar las claves a las que h les asigna la misma entrada primaria, de forma que hash2 les asigna diferentes entradas secundarias
 - Función hash secundaria tal que, si $h(c_1) = h(c_2)$, entonces $\text{hash2}(c_1) \neq \text{hash2}(c_2)$

Tablas dispersas (o *tablas hash*)

Resolución de colisiones por recolocación en el mismo vector

Inconveniente: *pares recolocados ocupan posiciones de otras futuras claves*

- En general, el funcionamiento es bueno si el factor de carga α se mantiene por debajo cierto valor

- En recolocación simple o lineal con $hh(i) = i$

Número medio de comparaciones hasta encontrar una clave es C :

$$C = \frac{1 - \frac{\alpha}{2}}{1 - \alpha} \quad \left(\alpha = \frac{N}{\max} \text{ es el factor de carga} \right)$$

- Resultados muy buenos, independientemente de N , siempre que α sea inferior a 0.75:

α	0.10	0.25	0.50	0.75	0.90	0.95
C	1.06	1.17	1.50	2.50	5.50	10.50

Tablas dispersas (o *tablas hash*)

Resolución de colisiones por recolocación en el mismo vector

- Idealmente, la función de recolocación debería distribuir las claves uniformemente entre las posiciones vacías

- Con una función de recolocación “ideal”: $C \approx \frac{-1}{\alpha} \ln(1 - \alpha)$

Número medio de comparaciones hasta encontrar una clave es C

- Resultados muy buenos, independientemente de N , siempre que α sea inferior a 0.90

α	0.10	0.25	0.50	0.75	0.90	0.95	0.99
C	1.05	1.15	1.39	1.85	2.56	3.15	4.66

- Funciones cercanas al ideal son costosas de encontrar, mejor utilizar métodos más sencillos

Tablas dispersas (o *tablas hash*)

Resolución de colisiones por recolocación en el mismo vector

Inconveniente: *pares recolocados ocupan posiciones de otras futuras claves*

- Resolver el problema creado por el ***borrado de pares***
 - *Al buscar se recalculan posiciones (primero con h y luego con hh) hasta encontrar la clave c buscada o una posición libre (siguiendo una secuencia de posiciones o saltos reproducible)*
 - *Si se borra una clave x que está en una de las posiciones de la secuencia de saltos, es una posición libre que romperá la secuencia de búsqueda para encontrar la clave c*
- **Solución:** **marcar la posición como libre (estado inicial), ocupada, o borrada**
 - *Al borrar una clave: marcar su posición como "borrada" (no libre)*
 - *Al buscar una clave: utilizar las posiciones marcadas como "borradas" igual que si estuviesen marcadas como "ocupadas" por claves distintas a la buscada*
 - *Al insertar una clave: utilizar las posiciones marcadas como "borradas", igual que si estuviesen marcadas como "libres" y reutilizarlas, pero teniendo cuidado de **no dejar duplicados** (apariciones previas de la clave localizadas en posiciones posteriores en la cadena de saltos)*

Índice

- 1 Conceptos
- 2 Especificación
- 3 Tablas dispersas (o *tablas hash*)
- 4 Recomendaciones finales**

Recomendaciones finales

Elección de implementación para el TAD tabla

- Si valor máximo N (número de claves) no es estimable, o es muy grande:
 - **Representación basada en árboles de búsqueda equilibrados** (da unos costes en el caso peor en $\mathcal{O}(\log N)$)

Recomendaciones finales

Elección de implementación para el TAD tabla

- *Si valor máximo N (número de claves) no es estimable, o es muy grande:*
 - **Representación basada en árboles de búsqueda equilibrados** (da unos costes en el caso peor en $\mathcal{O}(\log N)$)
- *Si valor máximo N es estimable, y no muy grande:*
 - Tabla dispersa (operaciones: coste temporal caso promedio $\Theta(1)$)
 - Para aliviar (reducir) el factor de carga: sobredimensionar el vector soporte, con tamaño aproximadamente un 10 % o 20 % más de componentes que N (y generalmente elegir un número primo)

Recomendaciones finales

Elección de implementación para el TAD tabla

- *Si valor máximo N (número de claves) no es estimable, o es muy grande:*
 - **Representación basada en árboles de búsqueda equilibrados** (da unos costes en el caso peor en $\mathcal{O}(\log N)$)
- *Si valor máximo N es estimable, y no muy grande:*
 - Tabla dispersa (operaciones: coste temporal caso promedio $\Theta(1)$)
 - Para aliviar (reducir) el factor de carga: sobredimensionar el vector soporte, con tamaño aproximadamente un 10 % o 20 % más de componentes que N (y generalmente elegir un número primo)
- *Si se utilizan tablas dispersas:*
 - Si las operaciones de borrado son frecuentes:
 - Mejor utilizar resolución de colisiones por encadenamiento
 - El coste de las operaciones se mantendrá bajo si $\alpha = \frac{N}{\max}$ es pequeño. Si esto no se cumple, se puede optar por:
 - Recolocarlo todo sobre una nueva tabla de doble capacidad y “nueva” función h (a este procedimiento se le llama *rehashing*)

Recomendaciones finales

Las tablas hash están orientadas al acceso individual a los datos

- Si se requiere principalmente acceso consecutivo o por orden a los datos (es decir, un iterador), **no utilizaremos tablas hash**
- Pero si tenemos que utilizar tablas hash, y puntualmente necesitamos acceso consecutivo, exhaustivo o por orden:
 - *¿Cómo podemos hacer un recorrido por todos los datos del diccionario?*
 - Ejemplo: mostrar todos por pantalla, encontrar todos los que cumplen cierta condición, etcétera
 - *¿Cómo podemos hacer un recorrido por todos los datos del diccionario según el orden por clave?*
 - Ejemplo: mostrar todos por pantalla, ordenados de menor a mayor
 - *¿Cómo obtener el elemento con menor clave de todos los almacenados en la tabla?*
 - *Se puede utilizar una estructura auxiliar, pero esto aumenta costes*

Tablas dispersas (o *tablas hash*)

Antipatrón: $\mathcal{O}(1)$ promedio $\neq \mathcal{O}(1)$ peor caso

- En el examen, “coste $\mathcal{O}(1)$ escrito junto a una tabla dispersa sin decir **qué caso** se analiza

× Así no

Afirmar “*buscar* es $\mathcal{O}(1)$ y punto:

- en el **peor caso** todas las claves colisionan: la zona de desbordamiento es una lista de n pares $\Rightarrow \mathcal{O}(n)$
- el $\mathcal{O}(1)$ depende de la función de dispersión y del factor de carga α
- si el enunciado exige peor caso garantizado, **la tabla dispersa no lo cumple**

✓ Así sí

Declarar el caso (paso 2 del método):

- **promedio**: $\mathcal{O}(1 + \alpha)$, que es $\mathcal{O}(1)$ si α se mantiene acotado (con *rehashing*)
- **peor caso**: $\mathcal{O}(n)$
- si se exige peor caso logarítmico $\Rightarrow$ AVL (lección 14)

Regla: con tablas dispersas, **distingue siempre** promedio y peor caso, y di de qué depende el promedio

Tablas dispersas (o *tablas hash*)

Antipatrón: dispersión cerrada sin marcar los borrados

- Con recolocación en el mismo vector, la búsqueda de c sigue la secuencia $P_0(c), P_1(c), \dots$ y **se detiene en la primera posición libre**

× Así no

Borrar = poner la componente a libre:

- las claves recolocadas *más allá* de esa posición se vuelven inaccesibles: la búsqueda para antes de llegar
- la tabla “pierde pares sin avisar”

✓ Así sí

Tres estados por componente: libre, ocupada, borrada:

- buscar **salta** las borradas y solo para en una libre
- insertar **puede reutilizar** una borrada
- si hay muchos borrados, *rehashing*; o usar encadenamiento si el borrado es frecuente

Regla: en dispersión cerrada, borrar es **marcar**, no vaciar

Tablas dispersas (o *tablas hash*)

El método en acción: buscar con encadenamiento

- 1 **Operación y tamaño:** buscar(t, c); sean n el número de pares, $\max$ el tamaño del vector y $\alpha = n/\max$ el factor de carga
- 2 **Caso:** los dos: promedio y peor
- 3 **Descomposición:** calcular $h(c)$; recorrer la lista de desbordamiento de esa entrada comparando claves
- 4 **Coste local:** $h(c)$: $\mathcal{O}(1)$ (si la función es de coste constante en la longitud de la clave); cada comparación: $\mathcal{O}(1)$
- 5 **Composición:** secuencia: $\mathcal{O}(1) +$ longitud de la lista recorrida
- 6 **Resultado:** promedio: listas de longitud $\alpha \Rightarrow \mathcal{O}(1 + \alpha)$; peor caso: todas las claves en la misma entrada $\Rightarrow \mathcal{O}(n)$

Método de análisis de coste en 6 pasos (lección 8): **rutina del curso** al cerrar cada implementación

Tabla de decisión: estructura $\leftrightarrow$ requisito

Estructura	Coste de las operaciones clave	Elegirla cuando...	Antipatrón
Pila	apilar, desapilar, cima: $O(1)$	solo importa el último en llegar (LIFO): deshacer, llamadas, delimitadores	buscar o recorrer «dentro» de una pila
Cola	encolar, desencolar, primero: $O(1)$	importa el orden de llegada (FIFO): turnos, esperas	usarla cuando se necesita acceso por clave
Lista	insertar/borrar en el punto de interés: $O(1)$; buscar: $O(n)$	colección recorrible con orden posicional	lista donde el requisito pide búsqueda eficiente
Diccionario	según implementación ($\rightarrow$ ABB, AVL, trie, hash)	acceso por clave	fixar la implementación sin leer el requisito de coste
ABB	buscar, insertar, borrar: $O(\text{altura})$; peor caso $O(n)$	claves con orden; recorrido ordenado	asumir $O(\log n)$ sin garantizar el equilibrio
AVL	buscar, insertar, borrar: $O(\log n)$ garantizado	como ABB, con cota de peor caso exigida	reequilibrar «a ojo»: las rotaciones siguen reglas
Árbol n -ario	según representación (hijo-izq. / herm.-dcho.)	información jerárquica: directorios, genealogías	replicar información compartida en vez de enlazarla
Trie	buscar, insertar: $O(\text{clave})$	claves que son secuencias; búsqueda por prefijos	olvidar que el coste va con la longitud de la clave, no con el n° de claves
Montículo	mínimo: $O(1)$; insertar, borrar raíz: $O(\log n)$	solo interesa el elemento más prioritario	top- K con montículo de máximos (usa mínimos acotado a K)
Tabla hash	buscar, insertar, borrar: $O(1)$ promedio, $O(n)$ peor	acceso por clave sin requisito de orden	prometer $O(1)$ en caso peor; dispersión cerrada sin marcar borrados

Tabla de decisión: estructura $\leftrightarrow$ requisito

Cómo usar el mapa en el examen

- 1 Subraya en el enunciado cada **requisito**: qué operaciones se piden y con qué **coste** (y en qué caso)
- 2 Para cada requisito, busca en la tabla la estructura cuya columna “*elegirla cuando...*” encaja; si dos encajan, decide por el coste exigido (peor vs. promedio, orden sí/no)
- 3 Si una entidad vive en varias estructuras, **enlázala** (punteros recíprocos), no la copies **F2**
- 4 Justifica cada elección con los **6 pasos**: la nota de eficiencia se pierde por no sumar sub-operaciones o por no definir las variables
- 5 Repasa la columna de **antipatrones**: es la lista de lo que más puntos ha costado en exámenes anteriores

Estructuras de Datos y Algoritmos

El TAD tabla y las tablas dispersas

LECCIÓN 19

☺ All wrongs reversed – bajo licencia [CC-BY-NC-SA 4.0](#)



Universidad
Zaragoza

Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza, España

Grado en Ingeniería Informática
UNIVERSIDAD DE ZARAGOZA

