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

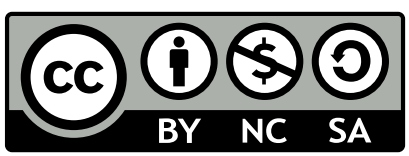
Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza, España

Curso 2023/2024

Grado en Ingeniería Informática

Universidad de Zaragoza

Aula 0.04, Edificio Agustín de Betancourt



Adaptadas de diapositivas de Javier Campos

Índice

Universidad Zaragoza

Índice

Universidad Zaragoza

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 $\langle c, v \rangle$

Universidad Zaragoza

Índice

Universidad Zaragoza

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}
        esVacío?: 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)

Universidad Zaragoza

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 $O(\log N)$) ordenado

Universidad Zaragoza

Índice

Universidad Zaragoza

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

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

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

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

- 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*

Universidad Zaragoza

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

Universidad Zaragoza

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)

Universidad Zaragoza

Tablas dispersas (o *tablas hash*)

Elección de función de *hashing* – funciones habituales

Para claves enteras

- **Método de la división** (o del módulo)
  - $h(c) = c \mod \texttt{max}$ (los índices del vector son `0..max - 1`)
  - Puede funcionar mal. Por ejemplo, si $\texttt{max} = 10$ 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 $\log \texttt{max}$ 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

Universidad Zaragoza

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 `ord(s[i])`, $\forall i \in [1, \texttt{long(s)}]$
  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 `max` $= 10007$ y las cadenas tienen todos 8 o menos caracteres $\rightarrow 256 \cdot 8 = 2048$ posiciones utilizables
    - Si `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 `ord(s[i])`, $\forall i \in [1, \texttt{long(s)}]$ (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 posib
                max = ... {El tamaño del vector con el que implementar la t

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 = cardinalCódigo^0 = 1}
    para i:=1 hasta long(c) hacer
        suma := suma + ord(c[i])*factor;
        factor := factor*cardinalCódigo {factor = cardinalCódigo^i}
    fpara;
    devuelve(suma mod max) {Método de la división}
fin
```

Tablas dispersas (o *tablas hash*)

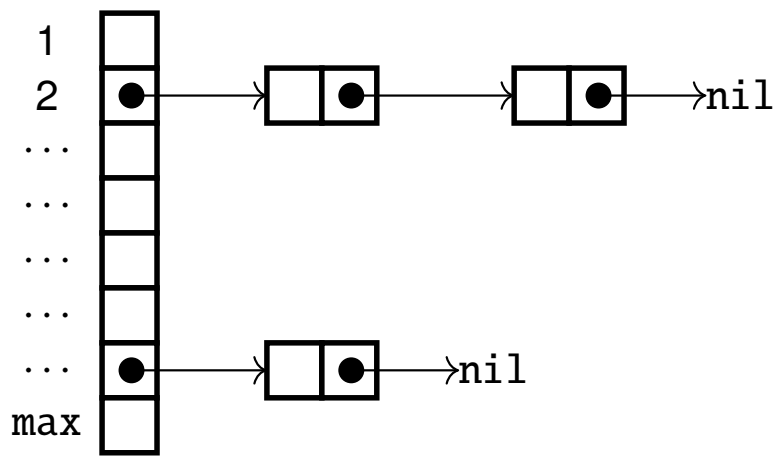
Resolución de colisiones

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

$h : \mathcal{D}_{\text{claves}} \mapsto 1..\texttt{max}$

(o mejor,
$h : \mathcal{D}_{\text{claves}} \mapsto 0..\texttt{max} - 1$)



Universidad Zaragoza

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: valor)
    {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
        fsi
    fin
fin {del módulo tablasDispersas (resolución de colisiones por encadenamiento)}
```

Universidad Zaragoza 1542

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 $\frac{N}{\texttt{max}}$ (*tamaño medio de las zonas de desbordamiento*)
- El ***factor de carga*** de la tabla se define como $\alpha = \frac{N}{\texttt{max}}$
  - *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 $N >$ `max`)*
- Se puede calcular[1] el número medio de comparaciones en una operación de búsqueda:
  - Si la búsqueda no tiene éxito: $C = \left(1 - \frac{1}{\texttt{max}}\right)^N + \frac{N}{\texttt{max}} \approx e^{-\alpha} + \alpha$
  - Si la búsqueda tiene éxito: $C = 1 + \frac{N-1}{2\texttt{max}} \approx 1 + \frac{1}{2}\alpha$
- Es decir, **el coste medio es del orden del factor de carga de la tabla**

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

 [CC BY-NC-SA 4.0 © R.J. Rodríguez]

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
  - $\alpha = \frac{N}{\texttt{max}}$ y siempre $\alpha \leq 1$ (no puede haber datos fuera del vector)
- Se utiliza un algoritmo de recolocación (`h2` 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:

    $$P_i(c) = ((h(c) + hh(i)) \quad \text{mód } \texttt{max}$$

    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

Universidad Zaragoza

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 α = dfracNmax ≤ 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
        fsi
    fin
```

¡Ojo!

Evitar ciclos infinitos)

Tablas dispersas (o *tablas hash*)

Resolución de colisiones por recolocación en el mismo vector

$$P_i(c) = ((h(c) + hh(i)) \quad \text{mód } \texttt{max}$$

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 si 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 $h(c)$

Universidad Zaragoza

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 `max` *es primo Agrupamientos secundarios*: idénticas secuencias de búsquedas para claves con mismo $h(c)$
- **Recolocación por doble hashing** (o doble dispersión): $hh(i) = i \cdot \texttt{hash2}(c)$
  - 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 $hash2(c_1) \neq hash2(c_2)$

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:

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

  - 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

Universidad Zaragoza 1542

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

Universidad Zaragoza

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)*

Universidad Zaragoza

Índice

Universidad Zaragoza

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 $O(\log N)$)

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 $O(\log N)$)
- *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 $O(\log N)$)
- *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 $\alpha = \frac{N}{\texttt{max}}$ es pequeño. Si esto no se cumple, se puede optar por:
    - Recolocar todo sobre una nueva tabla de doble capacidad y "nueva" función `h` (a este procedimiento se le llama *rehashing*)

Universidad Zaragoza 1542

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*

Universidad Zaragoza

Trabajo encargado

Leer lección 19 de los apuntes.

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

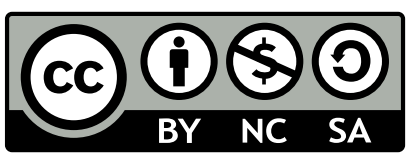
Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza, España

Curso 2023/2024

Grado en Ingeniería Informática

Universidad de Zaragoza

Aula 0.04, Edificio Agustín de Betancourt



Adaptadas de diapositivas de Javier Campos