Ordenación lineal

Clase 11

IIC 2133 - Sección 3

Prof. Eduardo Bustos

Sumario

Diccionarios

Definición

Un **diccionario** es una estructura de datos con las siguientes operaciones

- **Asociar** un valor a una llave
- **Actualizar** el valor asociado a una llave
- **Obtener** el valor asociado a una llave
- En ciertos casos, **eliminar** de la estructura una asociación llave-valor

Los ABB fueron nuestra primera EDD para implementar diccionarios

Funciones de hash

Definición

Dado un espacio de llaves K y un natural $m > 0$, una **función de hash** se define como

$$h : K \to \{0, \ldots, m-1\}$$

Dado $k \in K$, llamaremos **valor de hash de k** a la evaluación $h(k)$.

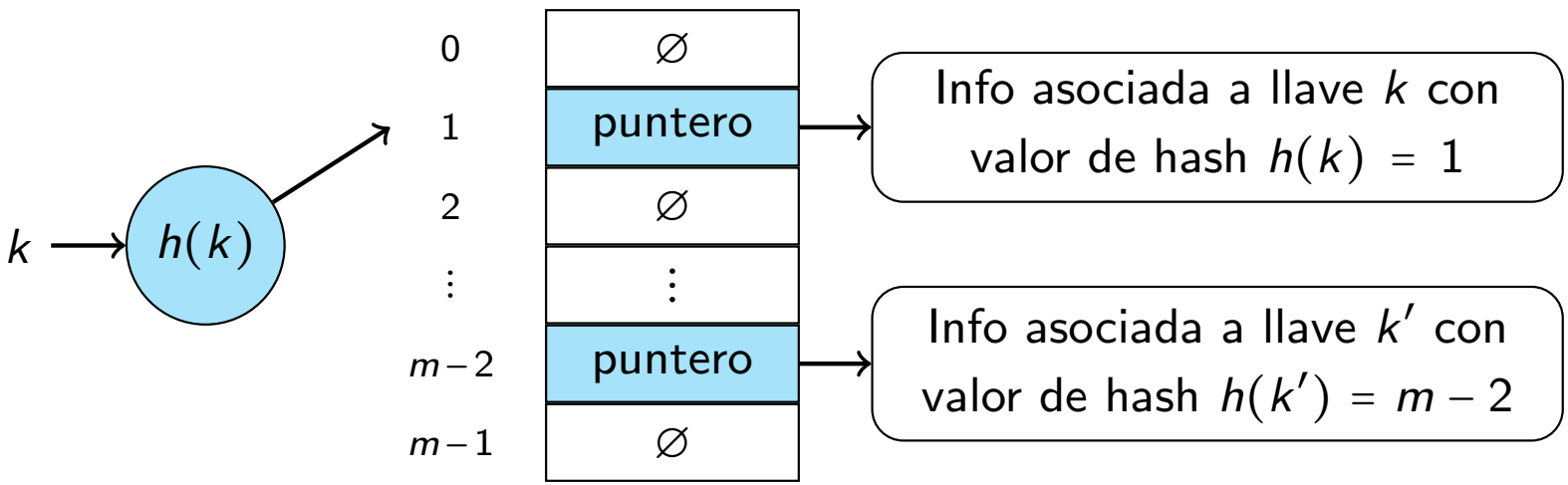
Notemos que

- Una función de hash nos permite mapear un espacio de llaves a otro más pequeño (con m razonable)
- Una función de hash no necesariamente es **inyectiva**
- Si $m < |K|$, entonces no puede ser inyectiva
- En la práctica, $m \ll |K|$

Tablas de hash

Definición

Dado $m > 0$ y un conjunto de llaves K, una **tabla de hash** T es una EDD que asocia valores a llaves indexadas usando una función de hash $h : K \to \{0, \ldots, m-1\}$. Diremos que tal T es de tamaño m.



Una **colisión** ocurre cuando $k_1 \neq k_2$ y $h(k_1) = h(k_2)$

¿Cómo manejamos colisiones?

Colisiones

Dos estrategias de manejo de colisiones

- Encadenamiento: listas ligadas en cada $T[h(k)]$
- Direccionamiento abierto: mediante un tipo de sondeo, se insertan los datos en posiciones vacías en T

Además, definimos una medida de qué tan llena está la tabla

Definición

Dada una tabla de hash T de tamaño m con n valores almacenados, se define su **factor de carga** como

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

Si λ no es aceptable, hacemos **rehashing**

Rehashing

Si λ es grande y ya no es aceptable, las operaciones se vuelven costosas

Una solución es hacer **rehashing**

- Se crea una nueva tabla más grande
- Aproximadamente del doble del tamaño original
- Como el espacio de índices ya no es de tamaño m, se define una nueva función de hash
- Mover los datos a la nueva tabla

Esta es una operación costosa para tablas de hash

- Es $\mathcal{O}(n)$ para n datos insertados
- No obstante, es **infrecuente**

Funciones de hash

Para promover un buen uso de la tabla, necesitamos "*buenas*" funciones de hash

¿Qué propiedades son deseables en una función de hash?

Funciones de hash

Dos ingredientes deseables en funciones de hash

- Esperamos que el mapeo de llaves sea uniforme en el rango $\{0, \ldots, m-1\}$ cuando suponemos llaves extraidas de manera uniforme desde K
- Debe ser fácil/barata de calcular

Lamentablemente, no hay reglas de diseño infalibles...

El diseño de funciones de hash depende fuertemente del contexto y es recomendable experimentar de forma práctica

Objetivos de la clase

- ☐ Conocer algoritmos de ordenación en tiempo lineal
- ☐ Comprender la limitación de dominio para tener tales algoritmos

Sumario

Ordenación en tiempo lineal

Consideremos una A secuencia de n naturales entre 0 y k

- Para todo $a_i \in A$, se tiene que $0 \leq a_i \leq k$
- Notemos que no necesariamente $n = k - 1$
- La secuencia A puede tener elementos repetidos

Propondremos un algoritmo de ordenación que **no compara**

- Para cada dato, contaremos cuántos datos son menores que él
- Esto nos indica la posición final de cada elemento

Si $k \in \mathcal{O}(n)$, entonces este algoritmo será $\Theta(n)$

El algoritmo `CountingSort()`

input : Arreglo $A[0 \ldots n-1]$, natural k
output: Arreglo $B[0 \ldots n-1]$

`CountingSort` (A, k)**:**

1 $B[0 \ldots n-1] \leftarrow$ arreglo vacío de n celdas
2 $C[0 \ldots k] \leftarrow$ arreglo vacío de $k+1$ celdas
3 **for** $i = 0 \ldots k$ **:**
4 $\quad C[i] \leftarrow 0$
5 **for** $j = 0 \ldots n-1$ **:**
6 $\quad C[A[j]] \leftarrow C[A[j]] + 1$
7 **for** $p = 1 \ldots k$ **:**
8 $\quad C[p] \leftarrow C[p] + C[p-1]$
9 **for** $r = n-1 \ldots 0$ **:**
10 $\quad B[C[A[r]] - 1] \leftarrow A[r]$
11 $\quad C[A[r]] \leftarrow C[A[r]] - 1$
12 **return** B

El algoritmo `CountingSort()`

```
CountingSort (A, k):
1    B[0...n-1] ← arreglo vacío
2    C[0...k] ← arreglo vacío
3    for i = 0...k :
4        C[i] ← 0
5    for j = 0...n-1 :
6        C[A[j]] ← C[A[j]] + 1
7    for p = 1...k :
8        C[p] ← C[p] + C[p-1]
9    for r = n-1...0 :
10       B[C[A[r]] - 1] ← A[r]
11       C[A[r]] ← C[A[r]] - 1
12   return B
```

- La complejidad del algoritmo es $\Theta(n+k)$
- Si $k \in \mathcal{O}(n)$, entonces `CountingSort()` es $\Theta(n)$

¡Este es un mejor tiempo que $\mathcal{O}(n \log(n))$!

Ejemplo de ejecución

```
CountingSort (A, k):
1     B[0...n-1] ← arreglo vacío
2     C[0...k] ← arreglo vacío
3     for i = 0...k :
4         C[i] ← 0
5     for j = 0...n-1 :
6         C[A[j]] ← C[A[j]] + 1
7     for p = 1...k :
8         C[p] ← C[p] + C[p-1]
9     for r = n-1...0 :
10        B[C[A[r]] - 1] ← A[r]
11        C[A[r]] ← C[A[r]] - 1
12    return B
```

A

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

Hacemos el llamado `CountingSort`$(A, 7)$

Ejemplo de ejecución

```
CountingSort (A, k):
1     B[0 ... n − 1] ← arreglo vacío
2     C[0 ... k] ← arreglo vacío
3     for i = 0 ... k :
4         C[i] ← 0
```

A

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

C

0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7

Ejemplo de ejecución

```
CountingSort (A, k):
1    B[0...n-1] ← arreglo vacío
2    C[0...k] ← arreglo vacío
3    for i = 0...k :
4        C[i] ← 0
5    for j = 0...n-1 :
6        C[A[j]] ← C[A[j]] + 1
```

A

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

C

0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7

C

1	2	0	2	0	2	0	3
0	1	2	3	4	5	6	7

Hasta aquí, $C[x]$ contiene el número de copias de x en A

Ejemplo de ejecución

```
CountingSort (A, k):
1       B[0...n-1] ← arreglo vacío
2       C[0...k] ← arreglo vacío
3       for i = 0...k :
4           C[i] ← 0
5       for j = 0...n-1 :
6           C[A[j]] ← C[A[j]] + 1
7       for p = 1...k :
8           C[p] ← C[p] + C[p-1]
```

A

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

C

1	2	0	2	0	2	0	3
0	1	2	3	4	5	6	7

C

1	3	3	5	5	7	7	10
0	1	2	3	4	5	6	7

Hasta aquí, $C[x]$ contiene cuántos elementos menores o iguales a x hay en A

Ejemplo de ejecución

9 **for** $r = n - 1 \ldots 0$ **:**

10 $\quad B[C[A[r]] - 1] \leftarrow A[r]$

11 $\quad C[A[r]] \leftarrow C[A[r]] - 1$

Para $r = 9$

A

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

B

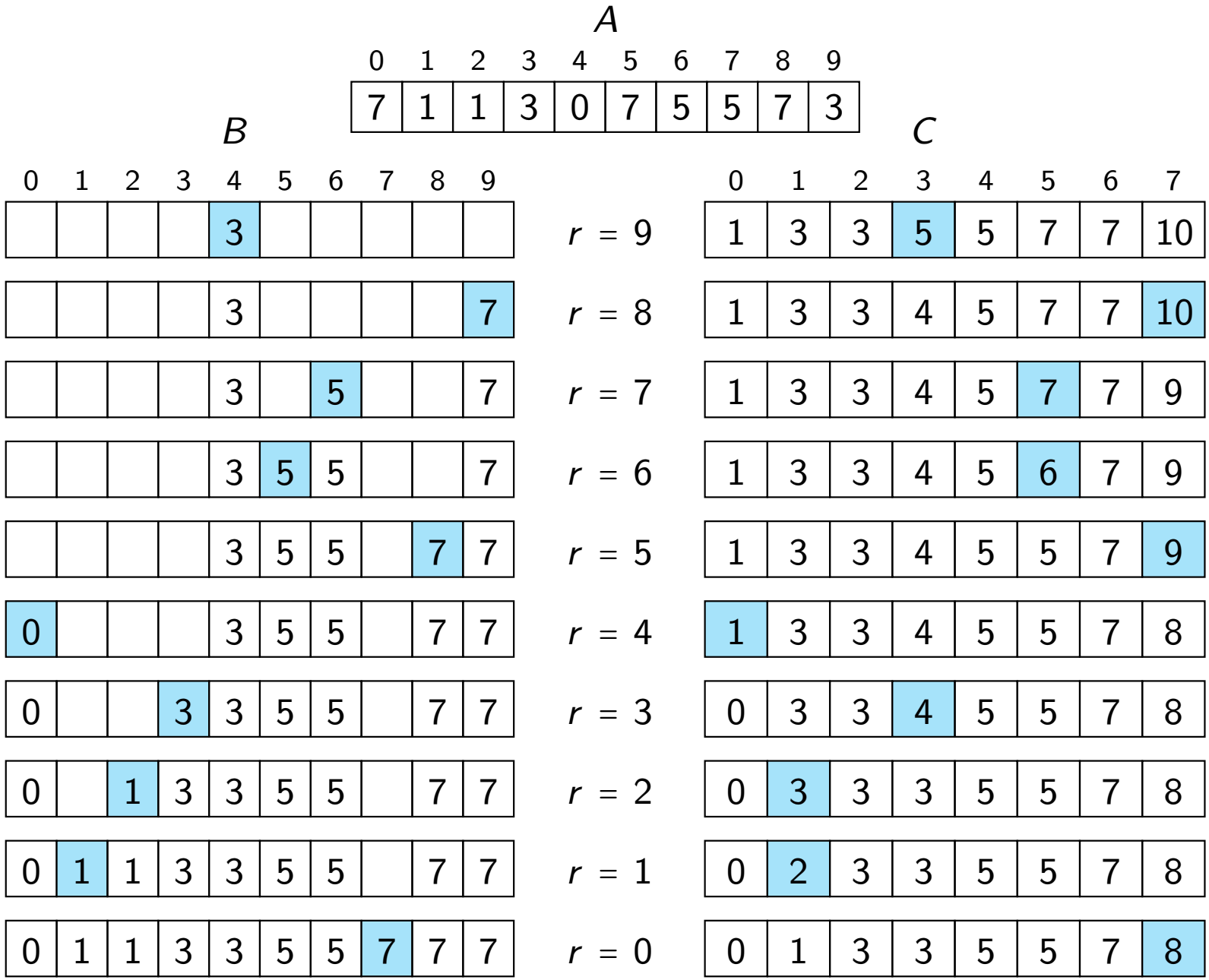
				3					
0	1	2	3	4	5	6	7	8	9

C

1	3	3	5	5	7	7	10
0	1	2	3	4	5	6	7

1	3	3	4	5	7	7	10
0	1	2	3	4	5	6	7

Ejemplo de ejecución



RadixSort

Otro algoritmo de ordenación en tiempo lineal es `RadixSort`

- Usado por las máquinas que ordenaban tarjetas perforadas
- Cada tarjeta tiene 80 columnas y 12 líneas
- Cada columna puede tener un agujero en una línea

RadixSort

El algoritmo ordena las tarjetas revisando una columna determinada

- Si hay un agujero en la columna, se pone en uno de los 12 compartimientos
- Las tarjetas con la perforación en la primera columna quedan encima
- La misma idea funciona para d columnas

Podemos generalizarla para un número natural de d dígitos

Ordenando por dígito

Consideremos un número natural de d dígitos $n_0 n_1 \cdots n_{d-1}$

- Podemos ordenar según el dígito más significativo n_0
- Según este dígito, separamos los números en *compartimientos*
- Luego, ordenados recursivamente cada compartimiento por su segundo dígito más significativo n_1
- Finalmente, combinamos los contenidos de cada compartimiento

Problema: posiblemente muchos llamados recursivos

Ordenación estable

Un ingrediente fundamental para el algoritmo que plantearemos es el siguiente

Definición

Dada una secuencia $A[0 \ldots n-1]$, sea $B[0 \ldots n-1]$ la secuencia resultante de ordenar A usando un algoritmo de ordenación $\mathcal{S}$. Sean a, a' elementos en A tales que para el algoritmo $\mathcal{A}$ son equivalentes y a aparece antes que a' en A. Diremos que $\mathcal{S}$ es **estable** si los elementos correspondientes b y b' aparecen en el mismo orden relativo en B.

Si ordenamos por el segundo dígito, un orden estable dejaría elementos que comparten segundo dígito en el mismo orden en que nos llegaron

RadixSort

El algoritmo `RadixSort` ordena por dígito **menos significativo**

- Ordena por dígito n_{d-1}
- Luego, usando el mismo arreglo, ordena por dígito n_{d-2}, **con un algoritmo estable**
- Luego de ordenar k dígitos, los datos están ordenados si solo miramos el fragmento $n_{d-k} \cdots n_{d-1}$
- Se requieren solo d pasadas para ordenar la secuencia completa

```
RadixSort(A, d):
    for j = 0 ... d − 1 :
        StableSort(A, j)    ▷ algoritmo de ordenación estable por
                              j-ésimo dígito menos significativo
```

Ejemplo de ejecución

	Arreglo inicial	Ordenado por unidad	Ordenado por decena	Ordenado por centena
0	0 6 4	0 0 **0**	0 **0** 0	**0** 0 0
1	0 0 8	0 0 **1**	0 **0** 1	**0** 0 1
2	2 1 6	5 1 **2**	0 **0** 8	**0** 0 8
3	5 1 2	3 4 **3**	5 **1** 2	**0** 2 7
4	0 2 7	0 6 **4**	2 **1** 6	**0** 6 4
5	7 2 9	1 2 **5**	1 **2** 5	**1** 2 5
6	0 0 0	2 1 **6**	0 **2** 7	**2** 1 6
7	0 0 1	0 2 **7**	7 **2** 9	**3** 4 3
8	3 4 3	0 0 **8**	3 **4** 3	**5** 1 2
9	1 2 5	7 2 **9**	0 **6** 4	**7** 2 9

RadixSort

```
RadixSort(A, d):
    for j = 0 ... d − 1 :
        StableSort(A, j)    ▷ algoritmo de ordenación estable por
                              j-ésimo dígito menos significativo
```

Supongamos que A tiene n datos naturales con d dígitos y se usa algoritmo estable lineal

- Si cada dígito puede tomar k valores distintos...
- Entonces `RadixSort` toma tiempo $\Theta(d \cdot (n + k))$
- Si d es constante y $k \in \mathcal{O}(n)$, entonces `RadixSort` es $\Theta(n)$

Dos implementaciones

Estas ideas tiene dos implementaciones

LSD string sort (*Least Significant Digit*)

- Si todos los strings son del mismo largo (patentes, IP's, teléfonos)
- Funciona bien si el largo es pequeño
- No recursivo

MSD string sort (*Most Significant Digit*)

- Si los strings tienen largo diferente
- Ordenamos con `CountingSort()` por primer caracter
- Recursivamente ordenamos subarreglos correspondientes a cada caracter (excluyendo el primero, que es común en cada subarreglo)
- Como `Quicksort`, puede ordenar de forma independiente
- **Pero** particiona en tantos grupos como valores del primer caracter

MSD en acción

she	are	are	are	...	are
sells	by	by	by		by
seashells	she	sells	seashells		sea
by	sells	seashells	sea		seashells
the	seashells	sea	seashells		seashells
sea	sea	sells	sells		sells
shore	shore	seashells	sells		sells
the	shells	she	she		she
shells	she	shore	shore		she
she	sells	shells	shells		shells
sells	surely	she	she		shore
are	seashells	surely	surely		surely
surely	the	the	the		the
seashells	the	the	the		the

Cuidados de MSD

En la ejecución de MSD string sort se debe considerar

- Si un string s_1 es prefijo de otro s_2, s_1 es menor que s_2

$$\text{she} \preceq \text{shells}$$

- Pueden usarse diferentes alfabetos
  - binario (2)
  - minúsculas (26)
  - minúsculas + mayúsculas + dígitos (64)
  - ASCII (128)
  - Unicode (65.536)
- Para subarreglos pequeños (e.g. $|A| \leq 10$)
  - cambiar a `InsertionSort` que *sepa* que los primeros k caracteres son iguales

Sumario

Objetivos de la clase

- ☐ Conocer algoritmos de ordenación en tiempo lineal
- ☐ Comprender la limitación de dominio para tener tales algoritmos