Funciones de hash y ordenación lineal

Clase 12

IIC 2133 - Sección 2

Prof. Mario Droguett

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

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

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

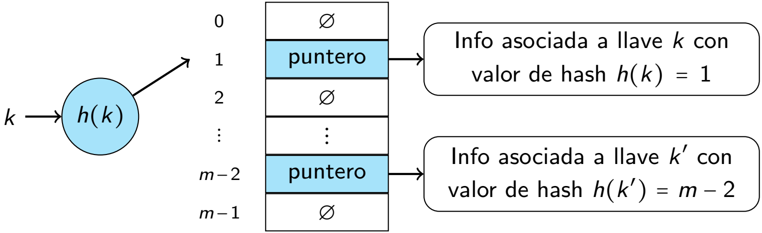
Notemos que

- 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**

Objetivos de la clase

- Identificar buenas propiedades de una función de hash
- Comprender algunas familias de funciones de hash comunes
- Conocer algoritmos de ordenación en tiempo lineal
- Comprender la limitación de dominio para tener tales algoritmos

Sumario

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 $h(k)$.

¿Qué caracteriza al espacio de llaves K?

- En principio podemos aceptar cualquier tipo de conjunto K
- Haremos la distinción entre llaves **numéricas** y **no numéricas**
- En cualquier caso, nos interesa llevar esas llaves a $\{0, \ldots, m-1\}$
- En general, las no numéricas primero se convierten a entero

Nos centraremos en las llaves numéricas

Funciones de hash

Nos interesa tener *buenas* funciones de hash

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

¿Cómo lograr esto? ¿o al menos acercarnos?

Sabemos que las llaves de K, en última instancia, se representan en binario

- Hay operaciones en binario muy eficientes
- Idea: que el valor $h(k)$ dependa de todos/muchos bits de k

Ejemplo

Consideremos $m = 16$ (tamaño de la tabla) y $k = 1234567$. El cuadrado de la llave es $k^2 = 1524155677489$, cuya representación binaria tiene n bits y es

$$(k^2)_2 = (01011000101101111011000001111101100110001)_2$$

Nos centraremos en los $m = 16$ bits intermedios

$$0101100010110\underline{\mathbf{1111011000001111}}1101100110001$$

Consideremos el valor de hash $h(k)$ dado por

$$(\underline{\mathbf{1111011000001111}})_2 = (62991)_{10}$$

de manera que $h(1234567) = 62991$

Funciones de hash

Ejemplo

Generalicemos este resultado para definir una función de hash que considere los m bits centrales de k^2

Recordemos que en representación binaria

- Dividir a por 2^b **borra** los b bits de la derecha de $(a)_2$
- $(2^b)_2$ es un 1 seguido de b ceros
- Como $[a \textbf{ mod } 2^b]$ es el resto al dividir entre 2^b y $[a \textbf{ mod } 2^b] < 2^b$, entonces $(a \textbf{ mod } 2^b)_2$ son los b bits de la derecha en $(a/2^b)_2$

Ejemplo

La función de hash que definimos es

$$h_m(k) = \left(\frac{k^2}{2^r}\right) \textbf{ mod } 2^m$$

donde $r = n - m/2$ es la cant. de bits ignorados en el lado derecho de $(k^2)_2$

- Modificando m podemos alterar la cantidad de bits que *importan*
- Si $m = n$, entonces $r = 0$ y

$$h_n(k) = \left(\frac{k^2}{2^0}\right) \textbf{ mod } 2^n = k^2 \textbf{ mod } 2^n = k^2$$

  que corresponde a la función h_m que más bits de k^2 incorpora

Hashing modular

Una familia simple de funciones razonable, que ya estudiamos, es de la forma

$$h(k) = k \ \mathbf{mod} \ m$$

y se conoce como **hashing modular** o **método de la división**

- Es fácil de calcular
- Distribuye uniformemente las llaves si estas son *aleatorias*

Hashing modular

Como se vio en el ejemplo, no siempre calculamos el módulo de un *k puro*... puede ser resultado de operaciones entre otros números

Recordemos las propiedades de la aritmética modular

- $(a + b) \textbf{ mod } c = ((a \textbf{ mod } c) + (b \textbf{ mod } c)) \textbf{ mod } c$
- $(a \cdot b) \textbf{ mod } c = ((a \textbf{ mod } c) \cdot (b \textbf{ mod } c)) \textbf{ mod } c$

Ejemplo

Calculemos $29 \cdot 72 \textbf{ mod } 13$ sin efectuar el producto $29 \cdot 72$

$$\begin{aligned}
29 \cdot 72 \textbf{ mod } 13 &= ((29 \textbf{ mod } 13) \cdot (72 \textbf{ mod } 13)) \textbf{ mod } 13 \\
&= (3 \cdot 7) \textbf{ mod } 13 \\
&= 21 \textbf{ mod } 13 \\
&= 8
\end{aligned}$$

Hashing modular

Algunas propiedades del hashing modular

- Si m es muy pequeño, habrán muchas colisiones
- Si $m = 2^b$, solo serán relevantes los b bits menos significativos... pueden repetirse mucho ciertos patrones
- Además, no queremos que m tenga muchos divisores
  - Si m es par y las llaves k son pares, k **mod** m es par
  - ¡La mitad de la tabla va a estar vacía!

Es conveniente tomar m como primo, cercano al tamaño ideal de la tabla

Método de la multiplicación

Otra familia razonable considera un real $0 < A < 1$ y es de la forma

$$h(k) = \lfloor m \cdot (A \cdot k \ \mathbf{mod}\ 1) \rfloor$$

y se conoce como **método de la multiplicación**

- Se extrae la parte decimal de $A \cdot k$
- A diferencia del hashing modular, el valor de m no es crítico
- Tomar m como potencia de 2 simplifica los cálculos

Es conveniente tomar m como primo, cercano al tamaño ideal de la tabla

Llaves strings

Si las llaves son strings debemos convertirlas a enteros

Una forma de realizar la conversión a entero es usar códigos ASCII y concatenarlos

Ejemplo

Si sabemos que

$$\texttt{ASCII}(\text{“A”}) = 64 = (01000001)_2 \qquad \texttt{ASCII}(\text{“B”}) = 66 = (01000010)_2$$

entonces una posible interpretación numérica de $s =$ “AB” es

$$(0100000101000010)_2 = 16706$$

Ahora bien, si

$$s' = \text{“A LONG EXAMPLE STRING”}$$

¿Qué tan grande sería esta representación para s'?

Llaves strings

Una forma común es interpretar s como entero en una **base apropiada**

Si $s = s_0 s_1 \ldots s_p$, una forma de convertirlo a un entero k es

$$k = \#(s_p) + \#(s_{p-1}) \cdot R + \cdots + \#(s_0) \cdot R^p = \sum_{i=0}^{p} \#(s_{p-i}) \cdot R^i$$

donde

- $\#(s_i) = \texttt{ASCII}(s_i)$
- $R \in \{31, 37\}$, i.e. un número primo que permite que los bits de todos los s_i jueguen un rol

Con esta estrategia estamos interpretando a s como un número de $p + 1$ dígitos en base R

Finalmente, convertimos k al rango $\{0, \ldots, m - 1\}$ con alguna función de hash

Restricciones de dominio

El recorrido de una función de hash es de la forma $\{0, \ldots, m - 1\}$

- Elementos del recorrido son naturales
- Esto permite usarlos para indexar posiciones de un arreglo
- Arreglo es de tamaño m

¿Qué sucede si queremos ordenar un conjunto de naturales $\{0, \ldots, k\}$?

Podemos aprovechar que sirven como índices de un arreglo

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

`CountingSort` (A, k):

$B[0 \dots n-1] \leftarrow$ arreglo vacío de n celdas
$C[0 \dots k] \leftarrow$ arreglo vacío de $k+1$ celdas
for $i = 0 \dots k$ **:**
  $C[i] \leftarrow 0$
for $j = 0 \dots n-1$ **:**
  $C[A[j]] \leftarrow C[A[j]] + 1$
for $p = 1 \dots k$ **:**
  $C[p] \leftarrow C[p] + C[p-1]$
for $r = n-1 \dots 0$ **:**
  $B[C[A[r]] - 1] \leftarrow A[r]$
  $C[A[r]] \leftarrow C[A[r]] - 1$
return B

El algoritmo `CountingSort()`

```
CountingSort (A, k):
    B[0 ... n − 1] ← arreglo vacío
    C[0 ... k] ← arreglo vacío
    for i = 0 ... k :
        C[i] ← 0
    for j = 0 ... n − 1 :
        C[A[j]] ← C[A[j]] + 1
    for p = 1 ... k :
        C[p] ← C[p] + C[p − 1]
    for r = n − 1 ... 0 :
        B[C[A[r]] − 1] ← A[r]
        C[A[r]] ← C[A[r]] − 1
    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):

$B[0 \ldots n-1] \leftarrow$ arreglo vacío
$C[0 \ldots k] \leftarrow$ arreglo vacío
for $i = 0 \ldots k$ **:**
  $C[i] \leftarrow 0$
for $j = 0 \ldots n-1$ **:**
  $C[A[j]] \leftarrow C[A[j]] + 1$
for $p = 1 \ldots k$ **:**
  $C[p] \leftarrow C[p] + C[p-1]$
for $r = n-1 \ldots 0$ **:**
  $B[C[A[r]] - 1] \leftarrow A[r]$
  $C[A[r]] \leftarrow C[A[r]] - 1$
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):
    B[0 ... n − 1] ← arreglo vacío
    C[0 ... k] ← arreglo vacío
    for i = 0 ... k :
        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):
    B[0 ... n − 1] ← arreglo vacío
    C[0 ... k] ← arreglo vacío
    for i = 0 ... k :
        C[i] ← 0
    for j = 0 ... n − 1 :
        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):

$B[0 \ldots n-1] \leftarrow$ arreglo vacío
$C[0 \ldots k] \leftarrow$ arreglo vacío
for $i = 0 \ldots k$ **:**
  $C[i] \leftarrow 0$
for $j = 0 \ldots n-1$ **:**
  $C[A[j]] \leftarrow C[A[j]] + 1$
for $p = 1 \ldots k$ **:**
  $C[p] \leftarrow C[p] + C[p-1]$

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

for $r = n-1 \ldots 0$ **:**
  $B[C[A[r]]-1] \leftarrow A[r]$
  $C[A[r]] \leftarrow C[A[r]]-1$

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

A

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

B / *C*

B: 0	1	2	3	4	5	6	7	8	9		C: 0	1	2	3	4	5	6	7
				3						$r = 9$	1	3	3	5	5	7	7	10
				3					7	$r = 8$	1	3	3	4	5	7	7	10
				3		5			7	$r = 7$	1	3	3	4	5	7	7	9
				3	5	5			7	$r = 6$	1	3	3	4	5	6	7	9
				3	5	5		7	7	$r = 5$	1	3	3	4	5	5	7	9
0				3	5	5		7	7	$r = 4$	1	3	3	4	5	5	7	8
0			3	3	5	5		7	7	$r = 3$	0	3	3	4	5	5	7	8
0		1	3	3	5	5		7	7	$r = 2$	0	3	3	3	5	5	7	8
0	1	1	3	3	5	5		7	7	$r = 1$	0	2	3	3	5	5	7	8
0	1	1	3	3	5	5	7	7	7	$r = 0$	0	1	3	3	5	5	7	8

Otro algoritmo de ordenación lineal

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

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 d_0
- Según este dígito, separamos los números en *compartimientos*
- Luego, ordenados recursivamente cada compartimiento por su segundo dígito más significativo d_1
- Finalmente, combinamos los contenidos de cada compartimiento

Problema: recursión busca que no mezclemos compartimientos antes de terminar

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 $j = 0 \ldots d-1$ **:**
        `StableSort`(A, j) $\triangleright$ 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

- 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

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