Tablas de *hash*

Estructuras de Datos y Algoritmos – IIC2133

Los **diccionarios** —como estructuras de datos— son conjuntos dinámicos que permiten tres operaciones:

insertar un elemento

eliminar un elemento

buscar un elemento

P.ej., la tabla de símbolos de un compilador

P.ej., si (las claves de) los objetos fueran números enteros entre 0 y 65,535 (16 bits),

… podríamos usar un arreglo **a** y *direccionamiento directo*:

- inicializamos **a**[0], …, **a**[65,535] con 0s
- *insert*[*k*] ejecuta **a**[*k*] = **a**[*k*]+1 (de modo que **a**[*k*] representa el número de veces que *k* ha sido insertado)
- *find*[*k*] verifica que **a**[*k*] no sea 0
- *delete*[*k*] exige (como precondición) que **a**[*k*] sea positivo y ejecuta **a**[*k*] = **a**[*k*]–1

¿Y si los enteros son de 32 bits en vez de 16?

El arreglo debería tener capacidad para almacenar unos 4,295 millones de elementos —no es práctico

¿Y si las claves de los objetos son *strings*?

Podríamos interpretar un *string* como un entero

P.ej.,

- cada carácter ASCII es representado en 7 bits como un entero entre 0 y 127 (= 2^7 - 1)
- un *string* representa un número en base 128
- p.ej., el valor de "clave" es 'c'·128^4 + 'l'·128^3 + 'a'·128^2 + 'v'·128^1 + 'e'·128^0 = 26,803,207,141
- esta representación produce enteros gigantescos y desperdiciaría la mayor parte del arreglo

Una solución eficaz es usar **hashing**

En vez de usar arreglos gigantescos,

… usamos una función *h* que relaciona números muy grandes (o *strings* interpretados como números)

… con números más pequeños y manejables —una **función de hash *h***

P.ej., si x es un número entero ≥ 0, entonces

$$h(x) = x \,\%\, m$$

produce un número entre 0 y $m - 1$

… apropiado para usar como índice en un arreglo de tamaño m

- p.ej., si m = 10,000, entonces el *string* “clave” sería indexado a 7,141

Con direccionamiento directo (el uso habitual de los arreglos), un objeto con clave *k* es almacenado en la casilla *k*

Con hashing, este objeto es almacenado en la casilla *h*(*k*)

… es decir, usamos la *función de hash h* para calcular la casilla a partir de la clave *k*

La **función de hash** relaciona el universo U de claves con las casillas de una **tabla de hash** $L[0 \ldots m-1]$:

$$h : U \rightarrow \{ 0, 1, \ldots, m - 1 \}$$

El objeto con clave k **hace hash** a la casilla $h(k)$

$h(k)$ es el **valor de hash** de la clave k

Hashing presenta algunos desafíos

1. Nos gustaría que las inserciones, eliminaciones y búsquedas tomaran tiempo O(1), como en los arreglos

Como vamos a ver, buscar un objeto en la tabla de hash efectivamente toma tiempo O(1)

… pero *sólo en promedio*

Desafíos

2. Además, dos (o más) claves distintas pueden hacer hash a la misma casilla —una **colisión**

   El ideal sería evitar las colisiones,

   … pero como $|U| > m$, debe haber al menos dos claves que tengan el mismo valor de hash: el ideal es imposible

   Hay técnicas para resolver colisiones

Podemos resolver las colisiones usando **encadenamiento**

Mantenemos un **arreglo *L* de listas ligadas** —una segunda estructura de datos:

$L_0, L_1, \ldots, L_{m-1}$

La función de hash nos dice dos cosas:

1) en cuál lista hay que insertar un objeto *X*
2) durante una búsqueda, cuál lista contiene (o debiera contener) a *X*

… usando **encadenamiento …**

Las búsquedas ya no son O(1) en el peor caso:

- buscar en una lista ligada toma tiempo lineal —O(*n*)— con respecto al número *n* de elementos que hay en la lista (no en toda la tabla)
- … pero si las listas son cortas, las búsquedas serán rápidas

h(*K k*)

calcula el valor de hash de la clave *k*

`insert`(*T x*)

inserta *x* al comienzo de la lista *L*[*h*(*x*.**`key()`**)]

`find`(*K k*)

busca un objeto con clave *k* en la lista *L*[*h*(*k*)]

`delete`(*T x*)

elimina *x* de la lista *L*[*h*(*x*.**`key()`**)]

Hashing con encadenamiento tiene buen desempeño

… bajo la suposición de que cualquier elemento tiene la misma probabilidad de hacer hash a cualquiera de las *m* casillas

Hashing con encadenamiento tiene buen desempeño

Sea n el número de datos almacenados:

- definimos el **factor de carga**, $\lambda = n/m$
- λ puede ser < 1, $= 1$, > 1
- una lista promedio tiene longitud λ
- la inserción toma tiempo O(1) —se inserta al comienzo de la lista
- la eliminación toma tiempo O(1) —listas doblemente ligadas
- el número esperado de intentos para una búsqueda sin éxito es $1 + \lambda$
- … y para una búsqueda exitosa es $2 + \lambda/2 - \lambda/2n = 1+(n-1)/2m$ — esperamos tener que recorrer la mitad de la lista

Una buena función de hash, *h*, satisface la suposición de **hashing uniforme simple** (HUS):

> cada clave tiene la misma probabilidad de hacer hash (es decir, de ir a parar) a cualquiera de las *m* casillas, independientemente de las otras claves

Pero, típicamente, no es posible verificar HUS:

- no conocemos la distribución de probabilidades de las claves
- las claves pueden no ser independientes

P.ej., si las claves son números reales aleatorios, k, distribuidos independiente y uniformemente en el rango $0 \leq k < 1$,

… la función $h(k) = \lfloor km \rfloor$ satisface HUS

Bajo la suposición de HUS, se puede demostrar las afirmaciones de la diap. #16

En la práctica, empleamos técnicas heurísticas para diseñar buenas funciones de hash

Método de la división*: *h*(*k*) = *k* % *m

- también, *hashing modular*
- *m* no debería ser una potencia de 2 (si $m = 2^p$, entonces *h*(*k*) es simplemente los *p* bits menos significativos de *k*)
- un buen valor para *m* es un número primo no cercano a una potencia de 2
- **es el método preferido cuando no sabemos nada acerca de las claves**

k	m=100	m=97
212	12	18
618	18	36
302	2	11
940	40	67
702	2	23
704	4	25
612	12	30
606	6	24
772	72	93
304	4	13
423	23	35
650	50	68
317	17	26
907	7	34
507	7	22

También puede usarse cuando las claves son strings:

```
int hash = 0
for i = 0 … s.length()-1:
      hash = (R*hash+s.charAt(i)) %m
```

`charAt()` devuelve un entero no negativo, de 16 bits

Si *R* es mayor que el valor de cualquier carácter,

… el resultado equivale a tratar a *s* como un entero de *n* (= `s.length()`) dígitos en base *R*,

… calculando el resto que se produce cuando dividimos el número por *m*

Si *R* es un primo pequeño —p.ej., 31— no ocurre *overflow* y el resultado está entre 0 y *m*–1

… empleamos técnicas heurísticas …

***Método de la multiplicación*: $h(k) = \lfloor m\ (k\ A\ \%\ 1) \rfloor$**

- A es una constante en el rango $0 < A < 1$ ($kA\ \%\ 1$ es la parte fraccional de kA)
- es más lento que el método de la división
- el valor de m no es crítico, y conviene elegirlo como una potencia de 2 (para facilitar el cálculo de $h(k)$)
- Knuth sugiere que $A \approx (\sqrt{5} - 1)/2 = 0.618\ldots$ produce buenos resultados

¿Cómo elegimos *m*?

Pequeño, para no desperdiciar un área grande de memoria contigua con listas vacías

… pero grande, para no perder tiempo buscando en listas muy largas

La decisión no es crítica

Si aparecen más claves que las que esperábamos, las búsquedas se van a demorar un poquito más

… y si hay pocas claves almacenadas, entonces las búsquedas serán y rápidas a costa de algún espacio desperdiciado

Si el espacio no es un recurso crítico, podemos usar un *m* suficientemente grande para que el tiempo de búsqueda sea constante

… y si es crítico, elegimos *m* tan grande como podamos y aún tendremos un muy buen comportamiento

También podemos resolver las colisiones mediante **direccionamiento abierto**

Todos los elementos son almacenados en la tabla:

- cada casilla contiene ya sea un objeto o *null*
- la tabla se puede llenar, es decir, $\lambda \leq 1$
- generalmente, $\lambda \leq 0.5$

… **direccionamiento abierto …**

Evitamos el uso de punteros:

- en vez de seguir punteros (como en el caso de las listas ligadas al usar encadenamiento)
- … *calculamos* la secuencia de casillas que vamos a revisar

Al buscar un objeto, revisamos sistemáticamente las casillas de la secuencia,

… hasta que lo encontramos o queda claro que no está

Al insertar, la secuencia de casillas revisadas depende de la clave que estamos insertando

… pero también del número de intentos fallidos que llevamos hasta el momento:

$$h : U \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}$$

Para cada k, la secuencia de revisión

$$h(k, 0), h(k, 1), \ldots, h(k, m-1)$$

… debe ser una permutación de $\{0, 1, \ldots, m-1\}$:

todas las casillas de la tabla son finalmente consideradas, a medida que la tabla se llena

Un problema del direccionamiento abierto

Al eliminar un objeto de la casilla *r*, no podemos marcar la casilla como vacía (*null*):

- sería imposible encontrar un objeto cuya secuencia de revisión pasó por la casilla *r* y la encontró ocupada

Una solución sería marcar la casilla *r* con un valor especial

… pero entonces habría que modificar la función *insert*, para que considere esas casillas como vacías

… y los tiempos de búsqueda ya no dependerían de λ

En conclusión,

cuando necesitamos eliminar objetos, es mejor usar encadenamiento

Hay tres técnicas para calcular secuencias de revisión de casillas —estrategias de resolución de colisiones:

a) Revisión lineal

b) Revisión cuadrática

c) Hashing doble

El ideal sería que cada clave tuviera la misma probabilidad de tener cualquiera de las $m!$ permutaciones de $\langle 0, 1, \ldots, m-1 \rangle$ como su secuencia de revisión:

- todas garantizan que la secuencia es una permutación de $\langle 0, 1, \ldots, m-1 \rangle$ para cada clave
- ninguna satisface el ideal —a lo más pueden generar m^2 secuencias de revisión distintas (hashing doble), pero no $m!$

a) $h(k, i) = (h'(k) + i) \% m$, $i = 0, 1, \ldots, m - 1$

- h' es una función de hash ordinaria
- la revisión inicial determina toda la secuencia de revisión — sólo hay m secuencias de revisión distintas

S	6						**S**									
E	10						S				**E**					
A	4				**A**		S				E					
R	14				A		S				E			**R**		
C	5				A	**C**	S				E			R		
H	4				**A**	**C**	**S**	**H**			E			R		
E	10				A	C	S	H			**E**			R		
X	15				A	C	S	H			E			R	**X**	
A	4				**A**	C	S	H			E			R	X	
M	1		**M**		A	C	S	H			E			R	X	
P	14	**P**	M		A	C	S	H			E			**R**	**X**	
L	6	P	M		A	C	**S**	**H**	**L**		E			R	X	

b) $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \% m$, $i = 0, 1, \ldots, m - 1$

- h' es una función de hash ordinaria
- c_1 , $c_2 \neq 0$ son constantes
- si dos claves revisan inicialmente la misma casilla, entonces sus secuencias de revisión son las mismas —sólo se usan m secuencias de revisión distintas
- sólo si m es primo y al menos la mitad de la tabla está vacía tenemos garantía de encontrar un espacio para una nueva clave

c) $h(k, i) = (h'(k) + ih''(k)) \% m$, $i = 0, 1, \ldots, m - 1$

- h' y h'' son funciones de hash
- para hacer uso de toda la tabla, el valor de h'' debe ser un primo relativo de m —p.ej., si m es 2^p, elegir h'' tal que produzca sólo números impares
- $h''(k) = R - (k \bmod R)$, con R un primo menor que m, funciona bien
- produce m^2 secuencias de revisión distintas, mucho mejor que a) o b)

P.ej., si insertamos las claves 89, 18, 49, 58 y 69, en este orden, en una tabla con 10 casillas*, usando

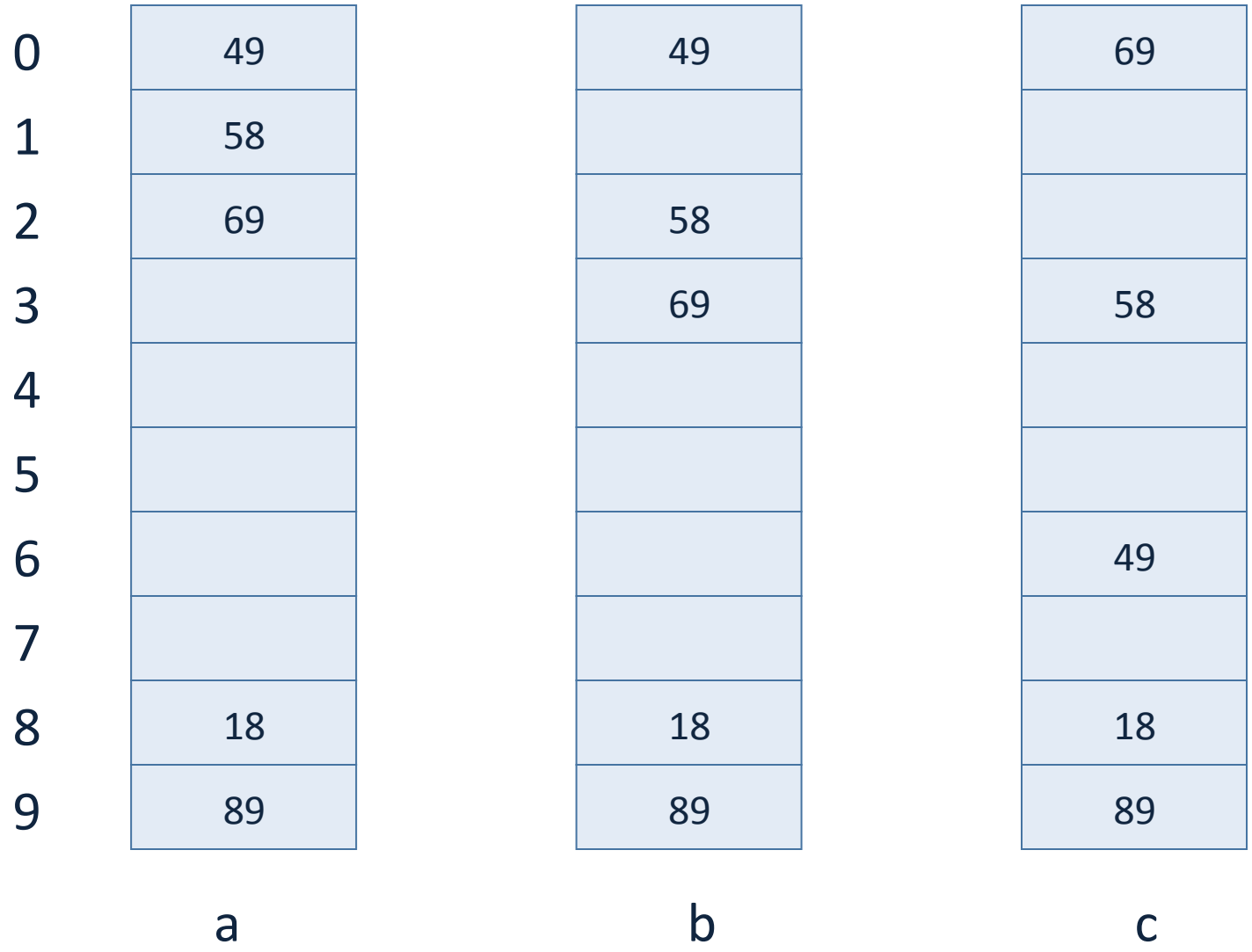
a) $h(k, i) = (k \,\%\, 10 + i\,) \,\%\, 10$

b) $h(k, i) = (k \,\%\, 10 + i^2\,) \,\%\, 10$

c) $h(k, i) = (k \,\%\, 10 + i\,(7 - (k \,\%\, 7)) \,\%\, 10$

… obtenemos los resultados que se muestran en la próxima diapositiva

*Sería mucho mejor, en particular, para b) y c), si el número de casillas fuera un número primo, pero 10 es más simple para los efectos del ejemplo


0
1
2
3
4
5
6
7
8
9
49
58
69
18
89
49
69
58
18
89
69
58
49
18
89
a
b
c

Rehashing

Si la tabla se empieza a llenar, las operaciones van a tomar mucho tiempo (en el caso de revisión cuadrática la inserción podría incluso fallar totalmente)

Construimos otra tabla aproximadamente el doble de grande*, con sus correspondientes nuevas funciones de hash

… y calculamos el nuevo valor de hash para insertar en la tabla nueva cada clave de la tabla original —***rehashing***

Es una operación cara, pero en realidad no es tan terrible porque ocurre muy infrecuentemente:

- tiene que haber habido n/2 inserciones antes del último rehash

  … de modo que esencialmente agrega un costo constante a cada inserción (por eso *)