Tablas de hash

Clase 12

IIC 2133 - Sección 2

Prof. Mario Droguett

Sumario

Recordatorio: 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

Diccionarios

Los ABB efectivamente soportan las operaciones de diccionario

- La complejidad de las operaciones es $\mathcal{O}(h)$
- Cuando están balanceados, $h \in \mathcal{O}(\log(n))$ para n llaves almacenadas

Ejemplo

Podemos mantener pares (llave, valor) de la forma (`rut`, `archivo`)

- Podemos saber si un `rut` está en el dic. haciendo búsqueda por `rut`
- Inserción usa `rut` para ubicar el nodo, balanceando si es necesario

Hay algo más que los ABB poseen y no es un requisito de los diccionarios

Diccionarios

Los ABB no solo soportan las operaciones de diccionario

- La propiedad de **árbol de búsqueda** garantiza que los datos están **ordenados**
- Si el orden es importante, esto es necesario

¿Qué es lo más importante en un diccionario?

Diccionarios

El principal objetivo de los diccionarios es **búsqueda eficiente** de llaves

- El objetivo secundario es **inserción/modificación eficiente** de pares llave-valor
- Por esto, el orden de las llaves deja de ser relevante

¿Podemos buscar e insertar más rápido
si nos olvidamos de mantener el orden?

Diccionarios

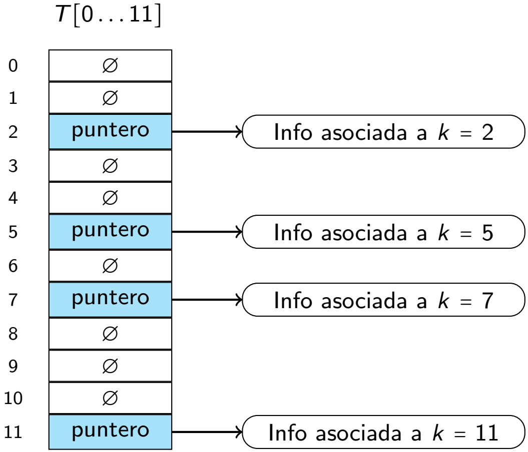
Para motivar nuestra siguiente estructura, consideremos un escenario ideal

- Conjunto de llaves posibles $K = \{0, \dots, 11\}$ fijo y conocido
- Dada una llave $k \in K$, interesa saber si esta se encuentra asociada a un valor en la EDD

Este escenario se puede manejar con la siguiente EDD básica

- Arreglo $T[0 \dots 11]$ iniciado con $\varnothing$ en cada celda
- No almacenamos las llaves en las celdas, sino un puntero al valor asociado a la llave k en $T[k]$
- Es decir, $T[k] = \varnothing \Leftrightarrow$ no hay valor asociado a k en T

Diccionarios



¿Cuál es la complejidad de la búsqueda y la inserción en T?

Diccionarios

En la estructura T los accesos a $T[k]$ son **accesos por índice**

- Verificar si $T[k] = \varnothing$ es $\mathcal{O}(1)$
- Insertar/modificar valor en $T[k]$ es $\mathcal{O}(1)$

No solo las operaciones deseadas son súper eficientes

- A diferencia de un ABB, se almacena solo un puntero al valor guardado
- No se usan punteros a padres-hijos para mantener la estructura

¿Qué tan ideal es este escenario de llaves naturales K?

Diccionarios

El escenario de llaves K puede ocurrir en aplicaciones prácticas

- Rango de valores razonable
- Llaves siempre naturales (para ser usadas como índices de arreglos)

Ejemplo

En la universidad hay aproximadamente 25.000 estudiantes este año.

- Asignamos un $k \in \{0, \dots, 24.999\}$ a cada estudiante
- Usamos cada natural como índice del arreglo T

Diccionarios

Ejemplo

Cada estudiante ya posee un rut único

- Rango de rut's abarca hasta el 25.000.000
- Cantidad de estudiantes mucho menor (25.000)
- **Problema:** solo 1/1000 celdas del arreglo T indexado por ruts estarán ocupadas

No solo los ruts son llaves posibles

- Números de teléfono
- Patentes de vehículos
- ...

¿Cómo acercarnos a un conjunto de llaves K razonable?

Objetivos de la clase

- Comprender el concepto de función de hash
- Identificar limitaciones en el almacenamiento a través de arreglos indexados
- Comprender concepto de tabla de hash
- Comprender concepto de colisión y sus posibles manejos
- Distinguir diferencias entre encadenamiento y direccionamiento abierto

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 $h(k)$.

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 $m < |K|$, 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.

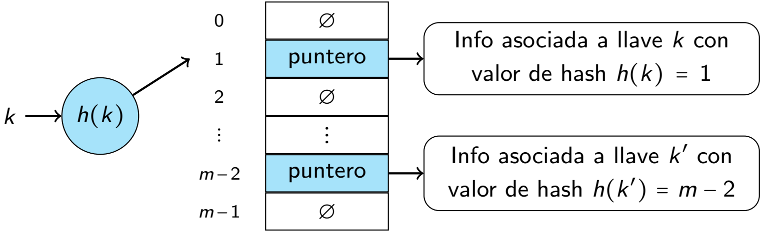


Tabla de hash

El ejemplo *ideal* que estudiamos es una tabla de hash

- La función de hash es $h : K \to K$ dada por

$$h(k) = k$$

- Las operaciones de diccionario son sencillas

```
IdentityHashSearch (T, k):
    return T[k]
IdentityHashInsert (T, k, v):
    T[k] = v
IdentityHashDelete (T, k):
    T[k] = ∅
```

0	$\varnothing$
1	$\varnothing$
2	puntero
3	$\varnothing$
4	$\varnothing$
5	puntero
6	$\varnothing$
7	puntero
8	$\varnothing$
9	$\varnothing$
10	$\varnothing$
11	puntero

Tabla de hash

- Usar la misma estrategia para hashing general sería

  `HashSearch` (T, k):
      return $T[h(k)]$
  `HashInsert` (T, k, v):
      $T[h(k)] = v$
  `HashDelete` (T, k):
      $T[h(k)] = \varnothing$

- Pero sabemos que h no necesariamente es inyectiva

- Es decir, puede ocurrir una **colisión** $h(k_1) = h(k_2)$ para $k_1 \neq k_2$

Veremos formas de manejar las colisiones

Sumario

Una función de hash típica

Para ejemplificar el problema de las colisiones, consideremos la siguiente función de hash

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

Se le conoce como **hashing modular** y corresponde al resto al dividir k entre m

- Notemos que $h(k) \in \{0, \ldots, m-1\}$ para todo k
- Todas las llaves con el mismo resto al dividir entre m generan una colisión, i.e.

$$h(k_1) = h(k_2) \Leftrightarrow k_1 \equiv_m k_2$$

Ejemplo

Tomando $m = 100$ y $K = \{0, \ldots, 999\}$, el hashing modular cumple

- $h(12) = h(112) = \cdots = h(912) = 12$
- $h(18) = \cdots = h(918) = 18$

Inserción

Usaremos la función de hashing modular para experimentar con **inserciones**

Consideremos $m = 7$. Insertemos la llave 15 en la siguiente tabla de hash

- Su valor de hash es $h(15) = 15 \textbf{ mod } 7 = 1$

0	$\varnothing$
1	$\varnothing$
2	$\varnothing$
3	$\varnothing$
4	$\varnothing$
5	$\varnothing$
6	$\varnothing$

Inserción

La posición $h(15) = 1$ está libre y guardamos la llave

0	$\varnothing$	0	$\varnothing$	0	$\varnothing$
1	$\varnothing$	1	$\varnothing$	1	15
2	$\varnothing$	2	$\varnothing$	2	$\varnothing$
3	$\varnothing$	3	$\varnothing$	3	$\varnothing$
4	$\varnothing$	4	$\varnothing$	4	$\varnothing$
5	$\varnothing$	5	$\varnothing$	5	$\varnothing$
6	$\varnothing$	6	$\varnothing$	6	$\varnothing$

Inserción

Ahora insertamos la llave 37

- Su valor de hash es $h(37) = 15 \ \mathbf{mod}\ 7 = 2$

0	$\varnothing$	0	$\varnothing$	0	$\varnothing$
1	15	1	15	1	15
2	$\varnothing$	2	$\varnothing$	2	37
3	$\varnothing$	3	$\varnothing$	3	$\varnothing$
4	$\varnothing$	4	$\varnothing$	4	$\varnothing$
5	$\varnothing$	5	$\varnothing$	5	$\varnothing$
6	$\varnothing$	6	$\varnothing$	6	$\varnothing$

Inserción

Ahora insertamos la llave 51

- Su valor de hash es $h(51) = 15 \textbf{ mod } 7 = 2$

0	$\varnothing$
1	15
2	37
3	$\varnothing$
4	$\varnothing$
5	$\varnothing$
6	$\varnothing$

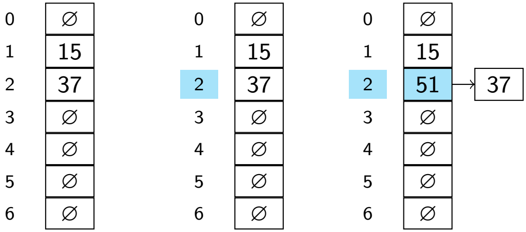
0	$\varnothing$
1	15
2	37
3	$\varnothing$
4	$\varnothing$
5	$\varnothing$
6	$\varnothing$

¿Qué hacemos con la colisión?

Inserción con encadenamiento

Primera propuesta: **encadenamiento**

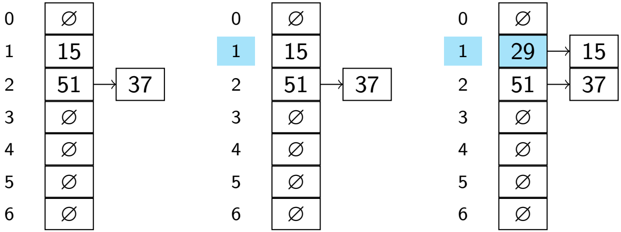
- Cada valor guardado es un nodo de una lista ligada
- Cada colisión agrega un nodo al principio/final de la lista



Inserción con encadenamiento

Al insertar la llave 29 seguimos la misma idea

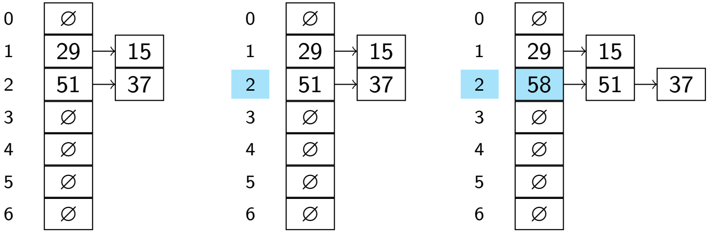
- Su valor de hash es $h(29) = 29 \ \mathbf{mod}\ 7 = 1$



Inserción con encadenamiento

Al insertar la llave 58 seguimos la misma idea

- Su valor de hash es $h(58) = 58 \textbf{ mod } 7 = 2$

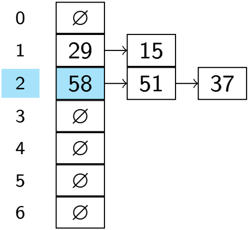


Encadenamiento

- Las operaciones de diccionario involucran la lista ligada $T[h(k)]$

  `ChainedHashSearch` (T, k):
  Buscar llave k en $T[h(k)]$
  `ChainedHashInsert` (T, k, v):
  Insertar (k, v) como cabeza de $T[h(k)]$
  `ChainedHashDelete` (T, k):
  Eliminar llave k de $T[h(k)]$

- La complejidad de estas operaciones depende de qué tan largas sean las listas
- Una *buena* función de hash repartiría las llaves de manera más o menos homogénea



Otra estrategia para colisiones

Volvamos a la inserción con colisión del 51

- Su valor de hash es $h(51) = 15 \ \mathbf{mod}\ 7 = 2$

0	$\varnothing$
1	15
2	37
3	$\varnothing$
4	$\varnothing$
5	$\varnothing$
6	$\varnothing$

0	$\varnothing$
1	15
2	37
3	$\varnothing$
4	$\varnothing$
5	$\varnothing$
6	$\varnothing$

¿Alguna alternativa al encadenamiento?

Inserción con sondeo lineal

Segunda propuesta: **direccionamiento abierto**

- Buscamos sistemáticamente una celda vacía
- Puede producir nuevas colisiones no previstas por h

Una forma de buscar: el **sondeo lineal** inserta en la primera celda vacía a la derecha de la colisión

0	$\varnothing$	0	$\varnothing$	0	$\varnothing$
1	15	1	15	1	15
2	37	2	37	2	37
3	$\varnothing$	3	$\varnothing$	3	51
4	$\varnothing$	4	$\varnothing$	4	$\varnothing$
5	$\varnothing$	5	$\varnothing$	5	$\varnothing$
6	$\varnothing$	6	$\varnothing$	6	$\varnothing$

Inserción con sondeo lineal

Al insertar la llave 29 seguimos la misma idea del sondeo lineal

- Su valor de hash es $h(29) = 29 \ \mathbf{mod} \ 7 = 1$

0	$\varnothing$	0	$\varnothing$	0	$\varnothing$	0	$\varnothing$
1	15	1	15	1	15	1	15
2	37	2	37	2	37	2	37
3	51	3	51	3	51	3	51
4	$\varnothing$	4	$\varnothing$	4	$\varnothing$	4	29
5	$\varnothing$	5	$\varnothing$	5	$\varnothing$	5	$\varnothing$
6	$\varnothing$	6	$\varnothing$	6	$\varnothing$	6	$\varnothing$

Búsqueda con sondeo lineal

Si las inserciones son con sondeo lineal, la búsqueda debe tenerlo en cuenta

- No necesariamente k está guardado en $T[h(k)]$
- Debemos revisar esa celda, y si no corresponde, buscar **hacia adelante**

Por ejemplo, al buscar la llave 29 comenzamos la búsqueda en la pos. 1

0	$\varnothing$	0	$\varnothing$	0	$\varnothing$	0	$\varnothing$
1	15	1	15	1	15	1	15
2	37	2	37	2	37	2	37
3	51	3	51	3	51	3	51
4	29	4	29	4	29	4	29
5	$\varnothing$	5	$\varnothing$	5	$\varnothing$	5	$\varnothing$
6	$\varnothing$	6	$\varnothing$	6	$\varnothing$	6	$\varnothing$

La búsqueda sigue la misma secuencia que la inserción

Búsqueda con sondeo lineal

¿Cómo detectamos si la llave no está?

- Comenzamos la búsqueda en $T[h(k)]$
- Si al buscar a la derecha llegamos a un $\varnothing$ significa que no está

Por ejemplo, al buscar la llave 10, tal que $h(10) = 10 \mod 7 = 3$

0	$\varnothing$	0	$\varnothing$	0	$\varnothing$
1	15	1	15	1	15
2	37	2	37	2	37
3	51	3	51	3	51
4	29	4	29	4	29
5	$\varnothing$	5	$\varnothing$	5	$\varnothing$
6	$\varnothing$	6	$\varnothing$	6	$\varnothing$

Concluimos que 10 no está almacenada

Eliminación con sondeo lineal

Para eliminar llaves guardadas tenemos un problema

- Si borramos una llave, la reemplazamos por $\varnothing$

Por ejemplo, si borramos el 51 y buscamos el 29 con $h(29) = 29 \mod 7 = 1$

0	$\varnothing$	0	$\varnothing$	0	$\varnothing$	0	$\varnothing$
1	15	1	15	1	15	1	15
2	37	2	37	2	37	2	37
3	51	3	$\varnothing$	3	$\varnothing$	3	$\varnothing$
4	29	4	29	4	29	4	29
5	$\varnothing$	5	$\varnothing$	5	$\varnothing$	5	$\varnothing$
6	$\varnothing$	6	$\varnothing$	6	$\varnothing$	6	$\varnothing$

Concluimos que 29 no está almacenado...

Si necesitamos eliminación, es mejor usar encadenamiento

Otros sondeos

Sondeo lineal

- Si $h(k) = H$, para alguna constante d buscamos en

$$H,\ H + d,\ H + 2d, \ldots$$

- Se debe cumplir $d = 1$ o que d y m son primos relativos

Sondeo cuadrático

- Si $h(k) = H$, buscamos en

$$H,\ H + 1,\ H + 4,\ H + 9, \ldots$$

Doble hashing

- Usamos dos funciones de hash h_1 y h_2 y buscamos en

$$h_1(k),\ h_1(k) + h_2(k),\ h_1(k) + 2h_2(k), \ldots$$

Todos ellos presentan el problema de la eliminación

Factor de carga

Dado que las colisiones impactan la tabla, nos interesa medir cuántos datos tenemos almacenados

Definición

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

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

El factor de carga es una medida de *qué tan llena está la tabla*

Según la estrategia de resolución de colisiones

- Encadenamiento: es aceptable $\lambda \approx 1$
- Direccionamiento abierto: $\lambda > 0.5$ resulta en inserciones y búsquedas muy lentas

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**

Sumario

Objetivos de la clase

- ☐ Comprender el concepto de función de hash
- ☐ Identificar limitaciones en el almacenamiento a través de arreglos indexados
- ☐ Comprender concepto de tabla de hash
- ☐ Comprender concepto de colisión y sus posibles manejos
- ☐ Distinguir diferencias entre encadenamiento y direccionamiento abierto