Programación 3

TUDAI 2024

Facultad de Ciencias Exactas. UNICEN

Estructuras de Dispersión (Hashing)

Docente: Federico Casanova

Motivación

- Se requiere un mecanismo de acceso asociativo (por contenido) a registros con la menor cantidad posible de accesos (una lectura solamente si fuese posible).

- Hasta el momento
  - Array ordenado: Log_2 (N) (búsqueda binaria)
  - Árbol binario ordenado: Log_2 (N) (balanceado)
  - Árboles n-arios de orden *p* = Log_p (N) (balanceados)

donde en todos los casos N es la cantidad de claves

¿Alguna organización del almacenamiento que mejore estos valores?

Estructuras de dispersión (tablas de Hashing)

Definimos una función **h** (llamada función de hashing) que, dada una clave de búsqueda x, determine la posición de almacenamiento de sus datos asociados.

Se aporta x, se aplica h(x) → posición de los datos asociados a x

La idea de las estructuras de dispersión es llevar este tiempo de búsqueda a O(1) o cercano, utilizando las técnicas de dispersión o hashing.

Es una estructura particularmente valiosa en aplicaciones con una frecuencia de búsquedas muy alta, donde el tiempo de búsqueda sea crítico.

Conceptos

- x: clave de búsqueda perteneciente al dominio estructurante (por el cual se guarda la información en la estructura por ej: DNI, Apellido, Nro Libreta Universitaria, etc.).
- h(x) función que convierte el elemento x del dominio estructurante en un elemento del espacio de almacenamiento → **direccionamiento por contenido**
- Espacio de almacenamiento dividido en M baldes.
- Cada balde puede guardar más de una clave → cada lugar es llamado ranura.
- La función de hashing más utilizada es h(x)= x mod M. *(el resto de la división x/M)*
- Generalmente se elige M como un número primo ya que la función mod dispersa los datos de manera más uniforme.

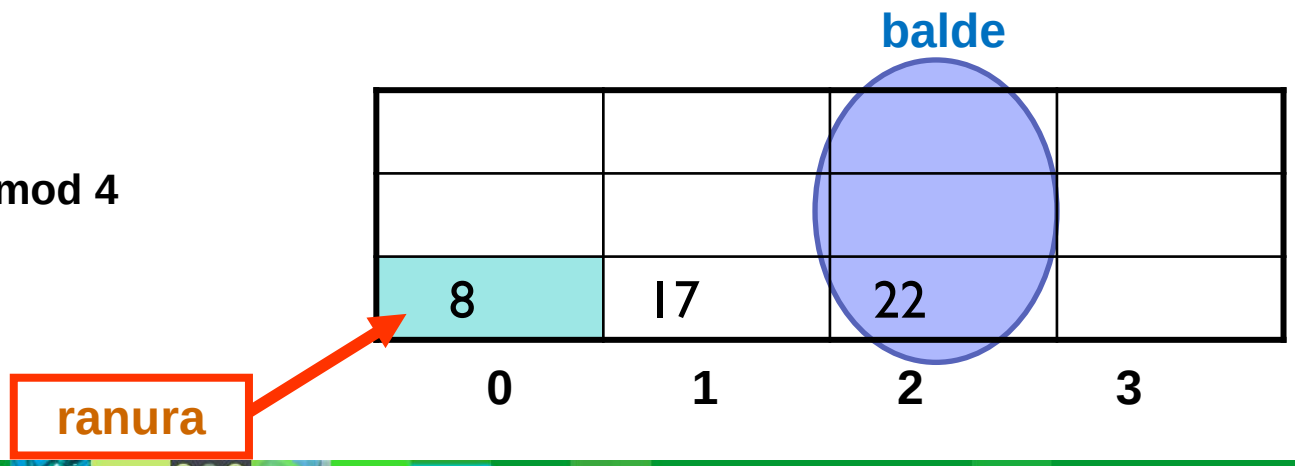
M= 4 baldes
r = 3 ranuras por balde

h(x)= x mod M → **h(x)= x mod 4**

h(8)= 8 mod 4 = 0

h(17)= 17 mod 4 = 1

h(22) = 22 mod 4= 2



Conceptos

Factor de Carga ρ (o densidad de almacenamiento): Es la proporción de espacio utilizado respecto del espacio total de almacenamiento asignado.

ρ *(rho)* = cantidad de datos / capacidad total de los baldes

nro.baldes * nro.ranuras = M*r

16			
0	17	22	7
0	**1**	**2**	**3**

$$\rho = \frac{5}{4 * 3}$$

2. Conceptos

- Característica de las estructuras de hashing:
  - No existe un ordenamiento físico de los datos (se encuentran dispersos por toda la estructura).
  - Facilita la inserción y eliminación rápida de registros por la clave de búsqueda x.
  - La búsqueda asociativa, es decir aportar la clave x y recuperar el resto de los datos que la acompañan, en promedio tiene un costo muy bajo (cercano a 1 acceso)
  - Se dan dos situaciones dependiendo de cómo distribuye los datos la función de hashing:

Hashing Perfecto

También conocido como distribución perfecta o direccionamiento directo.

La función de hash define una dirección del espacio de almacenamiento diferente para cada elemento del dominio estructurante, o sea

$$\forall\, X_1 \neq X_2 \Rightarrow h(X_1) \neq h(X_2)$$

Hay una función que transforma una clave de búsqueda en una posición única, que no será asignada a ningún otro elemento.

Son difíciles de definir → Se aplica a dominios con regularidades matemáticas que puedan ser explotadas (ej. arreglos multidimensionales).

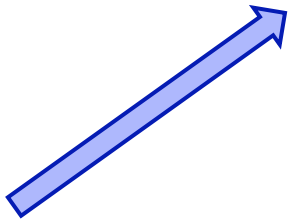
Si se logra definir, encontrar un elemento requiere un solo acceso ya que o está en el lugar dado por la función o no está almacenado.

Hashing Perfecto
Ejemplo

Almacenamiento en memoria contigua de arreglos multidimensionales (matriz).

A(0,0)	A(0,1)	A(0,2)	A(0,3)
A(1,0)	A(1,1)	A(1,2)	A(1,3)
A(2,0)	A(2,1)	A(2,2)	A(2,3)

Dominio estructurante (las claves que quiero guardar)



$h(A_{[i,j]}) = 4*i + j$

(almacena por fila)

Posición	Elemento
0	A(0,0)
1	A(0,1)
2	A(0,2)
3	A(0,3)
4	A(1,0)
5	A(1,1)
6	A(1,2)
7	A(1,3)
8	A(2,0)
9	A(2,1)
10	A(2,2)
11	A(2,3)

Tabla de hashing -> Espacio de Almacenamiento (memoria)

otra opción: $h(A_{[i,j]}) = 3*j + i$ **(almacena por columna)**

Hashing Puro

Dos elementos diferentes del dominio estructurante pueden ser asignados a una misma dirección del espacio de almacenamiento.

$$\exists\, X_1 \neq X_2 \;/\; h(X_1) = h(X_2)$$

M =13 , r =1

h(x)= x mod 13

Elementos = {22, 39, 46, 54, 79, 198, 14}

h(22) = 22 mod 13= 9

h(39) = 0

h(46) = 7

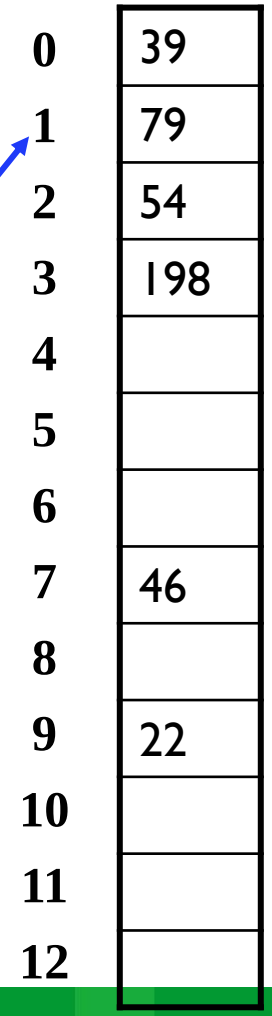
h(54) = 2

h(79) = 1

h(198) = 3

Insertar 14 : h(14)=1

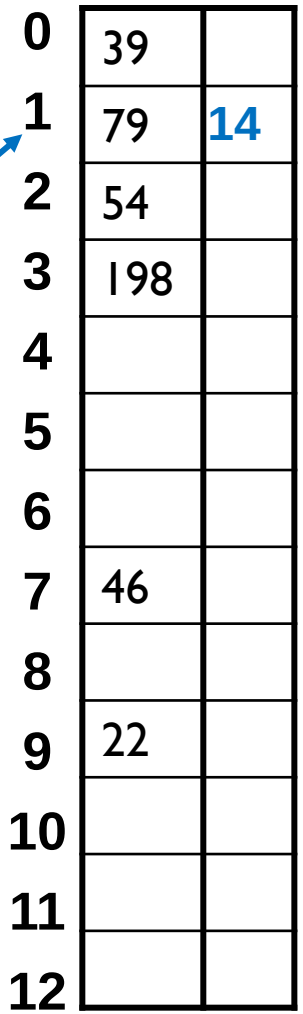
Colisión + Overflow



Colisión y Overflow (desborde)

h(x)= x mod 13 con M=13 y r=2

Insertar 14 : h(14)=1



Colisión

- En las funciones de **hashing puro no está garantizado que la función envíe cada elemento a una dirección diferente**: dos claves diferentes pueden tener el mismo resultado en su función de hash (SINÓNIMOS) produciendo una colisión en ese balde.

- Overflow o Desborde: Situación en la que un registro es asignado a una dirección (balde) que no tiene suficiente espacio para ser almacenado.

Insertar 27 : h(27)=1

Colisión +
Overflow (no hay espacio)

=>

Se debe reasignar al elemento a otra posición disponible en el almacenamiento

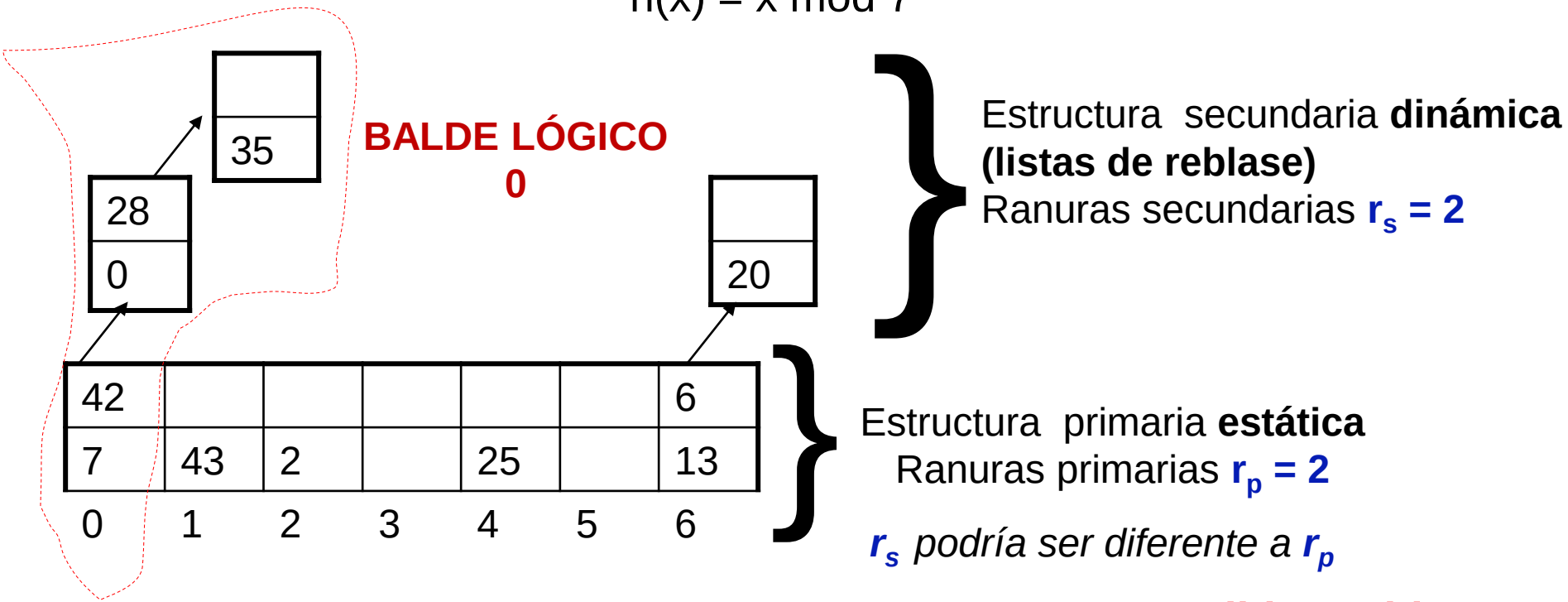
Estrategias para el overflow: Técnicas abiertas o dinámicas

- Se buscan estructuras dinámicas.
- El espacio de almacenamiento se expande o reduce a partir de las bajas y altas de los datos.
- Veremos dos técnicas:
  - El espacio físico primario (array) se mantiene fijo
    - Hashing Separado
  - El espacio físico primario se modifica
    - Hashing Separado con Crecimiento

Técnicas abiertas o dinámicas

- **Hashing separado (o con encadenamiento):** Se crea una lista de baldes asociada a cada balde de la estructura primaria para manejar el overflow (balde lógico).

h(x) = x mod 7

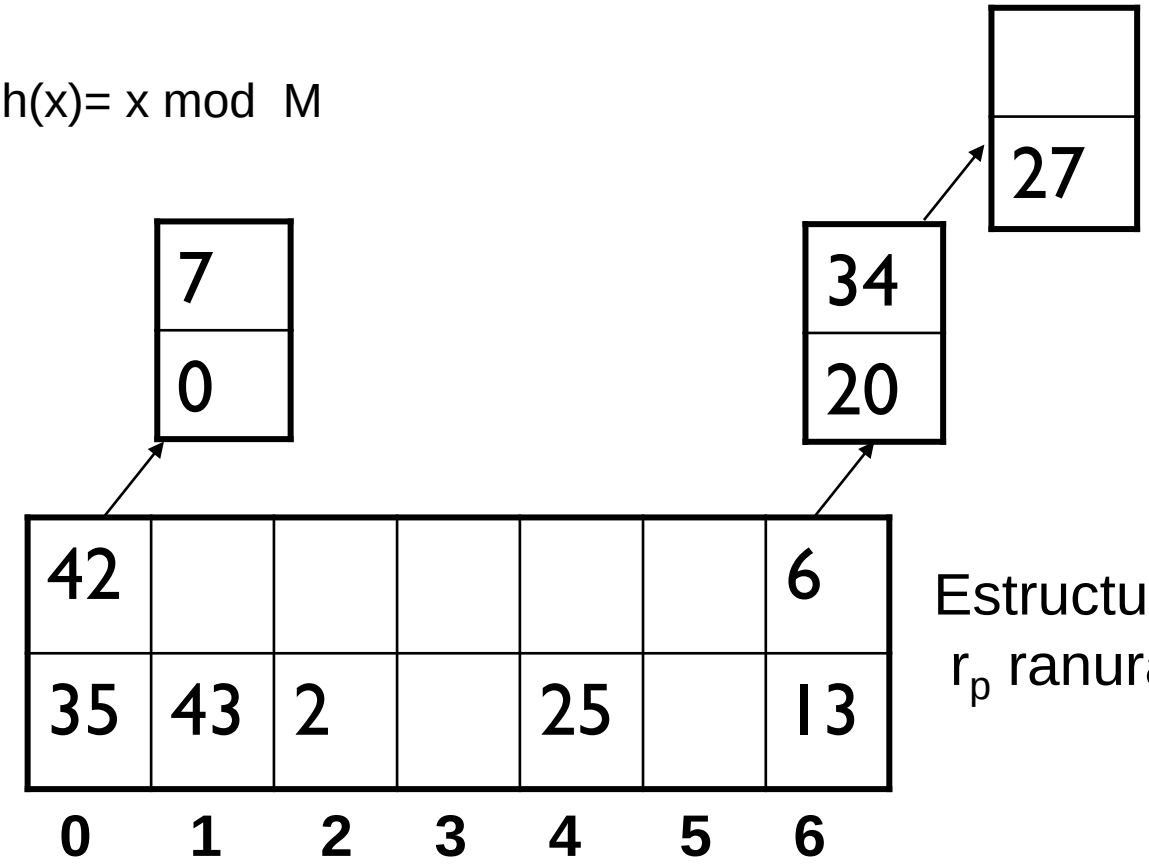


¿Posible problema?

Técnicas abiertas o dinámicas

- **Hashing separado con crecimiento:**

h(x)= x mod M



Estructura secundaria dinámica, r_s ranuras secundarias

Estructura primaria dinámica, r_p ranuras por balde primario

Se pretende que las listas vinculadas a cada balde físico (primario) no crezcan demasiado una respecto a otras …

Técnicas abiertas o dinámicas: Hashing separado con crecimiento

- Si hay colisiones se agregan elementos con un criterio de hashing separado.

- La estructura primaria se podrá expandir y contraer dinámicamente bajo algún criterio o algoritmo. Cuando la estructura primaria crece, se reorganizan todos los elementos en la tabla de hashing (operación costosa), el efecto buscado es que decrezca la longitud de las listas de rebalse (bajando así el costo de acceso).

- El crecimiento de la estructura primaria no se da continuamente, sino cuando se cumple cierta condición. Esa condición evalúa cómo es la cantidad de elementos actual respecto al tamaño de la estructura primaria.

Técnicas abiertas o dinámicas: Hashing separado con crecimiento

• Se define el un factor de carga de diseño preestablecido (ρ_d).

• Como ya vimos, un factor de carga es una relación entre la cantidad elementos que tiene al estructura y la capacidad de la misma (teniendo en cuenta la tabla primaria solamente).

• La estructura primaria que ahora será dinámica tendrá M baldes inicialmente (array de M baldes)

• Se calcula el límite de crecimiento L, a partir del cual la estructura primaria debe crecer. Cuando la cantidad de elementos en la estructura sea mayor o igual a L, la estructura debe crecer (cambia M). Se calcula L como:

$$L = \lfloor M * rp * \rho_d \rfloor$$

• ¿Cómo crece la estructura ? dependerá de la implementación (de a un balde, de a muchos, duplicando tamaño, etc). El valor de M cambia entonces L también se recalcula. Hay que reubicar todos los elementos !!

Técnicas abiertas o dinámicas: Hashing separado con crecimiento

Supongamos una tabla de M=12 y rp=1, y definimos un $\rho_d = 0,5$
Entonces L= $\lfloor 12 * 1 * 0,5 \rfloor = 6$

Entonces estamos diciendo que cuando la cantidad de elementos >= 6, vamos a hacer crecer la estructura primaria.

$\rho_d = 0,5$ significa un 50% de ocupación.

El valor que se elige para ρ_d mantiene un equilibrio entre longitud de listas de rebalse vs espacio que se desperdicia.

Un valor grande provocará que la estructura se llene demasiado y probablemente más colisiones y listas de rebalse más largas.

Un valor muy bajo mantendrá baja la relación entre elementos en la estructura respecto al tamaño de la misma, o sea un mayor desperdicio de espacio.

Técnicas abiertas o dinámicas: JAVA HASHTABLE

Veamos el caso de la HASHTABLE de la librería de Java

Valores por defecto de HASHTABLE : M=11 y $\rho_d = 0,75$
Siempre tiene rp=1

Estos valores (M y ρ_d) pueden setearse vía el constructor de Hashtable.
Cuando se crea calcula L (threshold).

Como función de hashing usa h(x) = x.hashCode( ) mod M

x.hashCode( ) es un método definido en object y el valor que retorna es un número que tiene que ver con la dirección de memoria donde se aloja x. Esto permite calcular la función de hash sobre claves x que no son necesariamente numéricas.

Técnicas abiertas o dinámicas: JAVA HASHTABLE

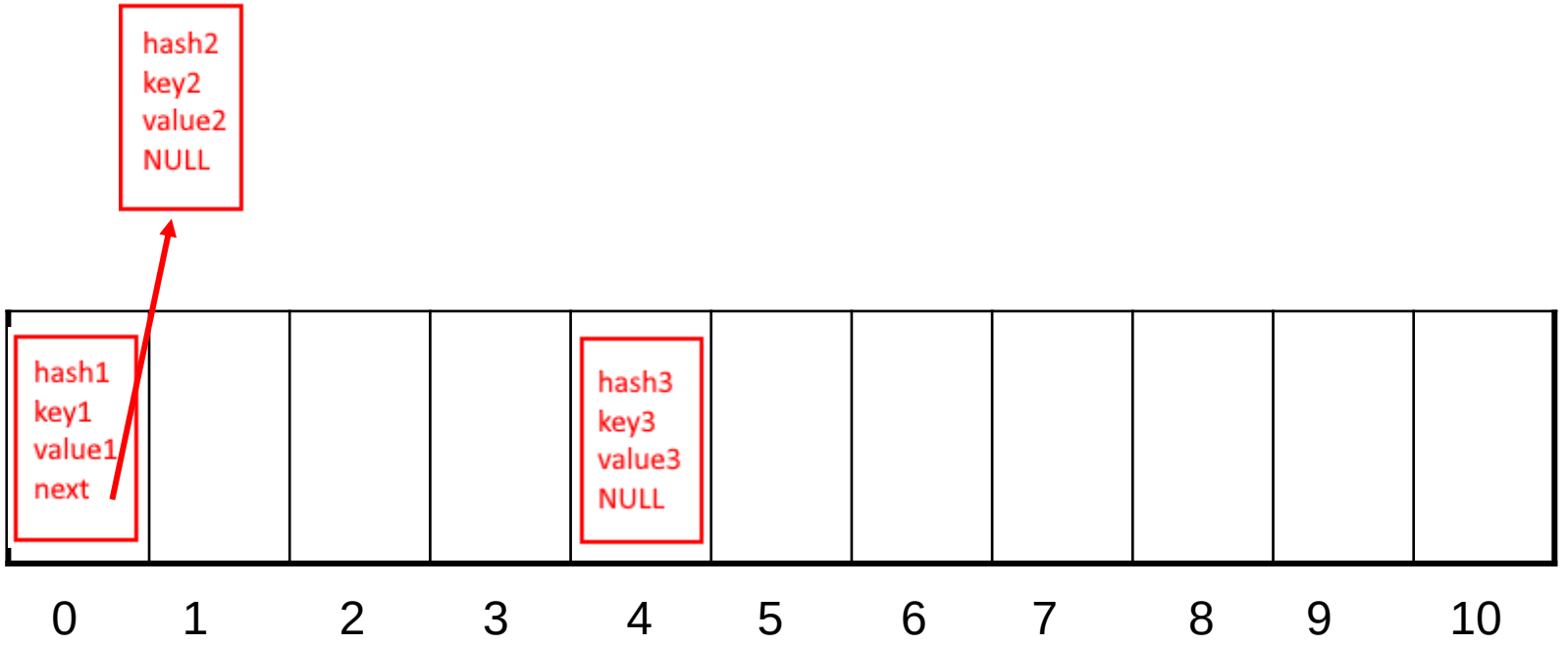
- public class Hashtable<K,V> extends Dictionary<K,V> implements Map<K,V>

- Declara la tabla de hashing como un array de Entry:
  - private transient Entry<?,?>[ ] table;

- La definición de Entry es:
  - private static class Entry<K,V> implements Map.Entry<K,V> {

```
        int hash;
        K key;
        V value;
        Entry<K,V> next;          //es una lista vinculada !! Esta es la lista de
                                  // rebalse del balde.
    …. }
```

Técnicas abiertas o dinámicas : JAVA HASHTABLE



Técnicas abiertas o dinámicas: JAVA HASHTABLE

- Agrega elementos con:
  - public synchronized V put(K key, V value) { … }
- Lo de synchronized se entenderá cuando veamos Threads….
- Calcula el índice mediante key.hashCode( ) mod table.length.
- Si la clave ya había sido agregada, reemplaza el value con el nuevo y devuelve el anterior en el return.
- Sino genera un nuevo Entry, y lo agrega al inicio de la lista del balde, y devuelve null.
- Si la cantidad de elementos en table es >= L, entonces llama a Rehash( ), este método crea una nueva tabla con el doble de tamaño de la anterior más uno, y luego va Entry por Entry de la tabla anterior reubicándolos en la tabla nueva (recalcula la función de hash a cada uno para saber su nueva ubicación). Y luego ubica el elemento nuevo en la nueva tabla.

Técnicas abiertas o dinámicas: JAVA HASHTABLE

- Rehash (parte más salientes del código):

```
protected void rehash() {
        int oldCapacity = table.length;
        Entry<?,?>[ ] oldMap = table;

        int newCapacity = (oldCapacity << 1) + 1;       // M = M*2 + 1
        Entry<?,?>[ ] newMap = new Entry<?,?>[newCapacity];

        threshold = (int)(newCapacity * loadFactor);
        table = newMap;

        for (int i = oldCapacity ; i -- > 0 ;) {
            for (Entry<K,V> old = (Entry<K,V>)oldMap[i] ; old != null ; ) {
                Entry<K,V> e = old;
                old = old.next;
                int index = (e.hash & 0x7FFFFFFF) % newCapacity;     // index = h(e)
                e.next = (Entry<K,V>) newMap[index];    // lo inserta al inicio
                newMap[index] = e;                      // de la lista del balde
            }
        }
    }
```

"& 0x7FFFFFFF convierte en positivo"
newCapacity sería el nuevo M.

Técnicas abiertas o dinámicas: JAVA HASHTABLE

- Obtiene el valor asociado a una clave con:

```
public synchronized V get(Object key) {
        Entry<?,?> tab[ ] = table;

        int hash = key.hashCode();

        int index = (hash & 0x7FFFFFFF) % tab.length;

        for (Entry<?,?> e = tab[index] ; e != null ; e = e.next) {    /// busca en la lista vinculada

              if ((e.hash == hash) && e.key.equals(key)) {
                  return (V) e.value;
              }
        }

        return null;
    }
```

- Busca de forma secuencial el elemento en la lista de Entry correspondiente al balde.

Técnicas abiertas o dinámicas: JAVA HASHTABLE

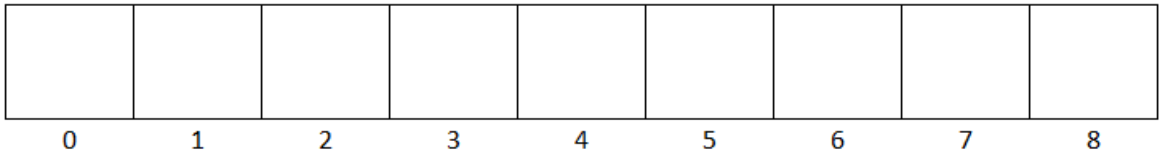
Consideraciones finales:

- El crecimiento busca reducir la longitud promedio de las listas de rebalse.
- **Agregar** un elemento se considera **O(1)**
- Y **buscar** un elemento puede suponerse en **promedio** de tiempo constante Θ(1), al igual que el **eliminar**.
- Puede ser útil y a veces necesario redefinir el método HashCode para los objetos que usaremos como clave. Por ejemplo si usamos como calve un valor int, se puede redefinir (override) hashCode = int que contiene, e Equals dirá si los int son iguales.

Técnicas abiertas o dinámicas: JAVA HASHTABLE

- Supongamos que definimos una HashTable con tamaño inicial M=4 y $\rho_d = 0,9$
- Entonces resultará L = $\lfloor 4 * 0,9 \rfloor = 3$ (threshold)
- Y queremos agregar las siguientes claves x del tipo Integer:

X	68	40	47	6	76	95	88	23	90	85	31	71	60	10	46	61	50	92	74	7	97	66	1	56
x mod 4	0	0	3																					
x mod 9	5	5	2	6	4	5	7																	
Nº	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24

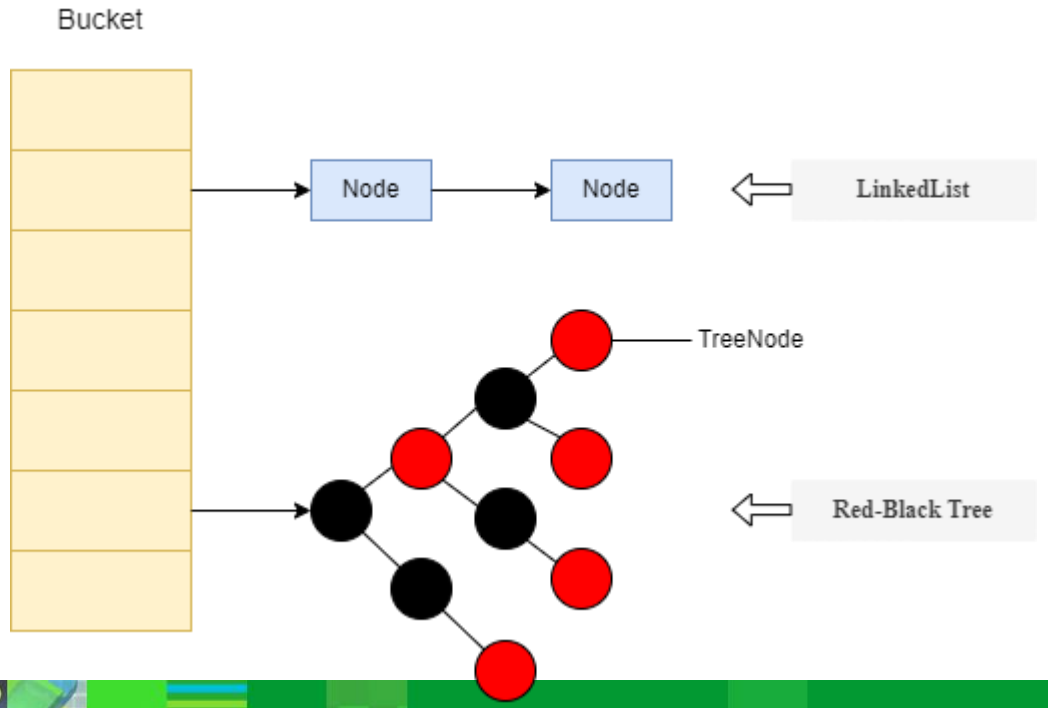


M = M*2 + 1 = 9

L = $\lfloor 9 * 0,9 \rfloor = 8$

Técnicas abiertas o dinámicas: JAVA HASHMAP

- Más avanzada que HASTABLE. Tiene algunas diferencias (investigar). Es más rápida.
- Agrega mejora sobre la lista de rebalse, cuando la cantidad de elementos en la lista supera una cierta cantidad establecida, convierte la lista de rebalse del balde en un tipo de árbol binario de búsqueda balanceado llamado árbol Red-Black (usando el hashCode como clave), llevando asi los tiempos de búsqueda en el balde de O(n) a O(log2 n), con n = cantidad elementos del balde.



En general, en cualquier estructura de datos ¿qué hacer si hay claves repetidas?

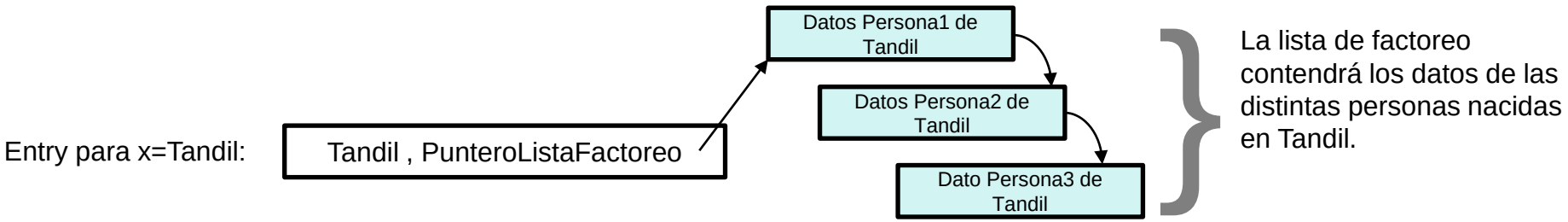
• En cualquier estructura de datos podemos distinguir las siguientes situaciones:

• Caso 1: La clave de busqueda **coincide con algun identificador (clave) de la información** almacenada. Quiere decir que para cada valor de clave de búsqueda se corresponde con **un único registro** de datos asociados.

Por ejemplo una tabla de hashing para datos de personas, con clave de búsqueda = DNI. Dado un DNI éste identifica a una única persona.
¿Qué otra clave se le ocurre si hablamos de alumnos de la Facultad?

• Caso 2: La clave de busqueda **no coincide con algun identificador de la información** almacenada. Quiere decir que para cada valor posible de clave de búsqueda puedo tener **uno o más registros** de datos asociados. Esto es un problema …

Por ejemplo archivo disperso de personas, con clave de búsqueda = Ciudad de Nacimiento. Para cada ciudad de nacimiento vamos a tener una o más personas que nacieron en esa ciudad. Por ejemplo cave de búsqueda="Tandil".
Dado que no es eficiente almacenar claves repetidas, la estrategia que se utiliza es la llamada ***lista de factoreo****.*



Usos de las Estructuras de Datos

Podemos identificar dos usos principales para las estructuras de datos (ej array, lista, árbol, tabla hash, etc)

1- **Para almacenar información**.
Podríamos guardar todos los datos en una estructura de datos principal (array, lista, árbol, archivo en disco, etc).
¿Y si queremos hacer búsquedas eficientes por algún campo?. En este caso usaremos una estructura de datos eficiente y se tendrá una clave de búsqueda, y un conjunto de información asociada. Al encontrar el elemento con esa clave de búsqueda en la estructura, habremos encontrado también su información asociada. Entonces por ej si usamos una HASHTABLE la ranura que almacena datos de alumnos, y cuya clave de búsqueda es DNI se "vería" así.

HASHTABLE

Qué pasa si además de buscar una persona por DNI lo quiero hacer por fecha de nacimiento o por rango de fecha de nacimiento (nacidos entre tal y tal fecha).

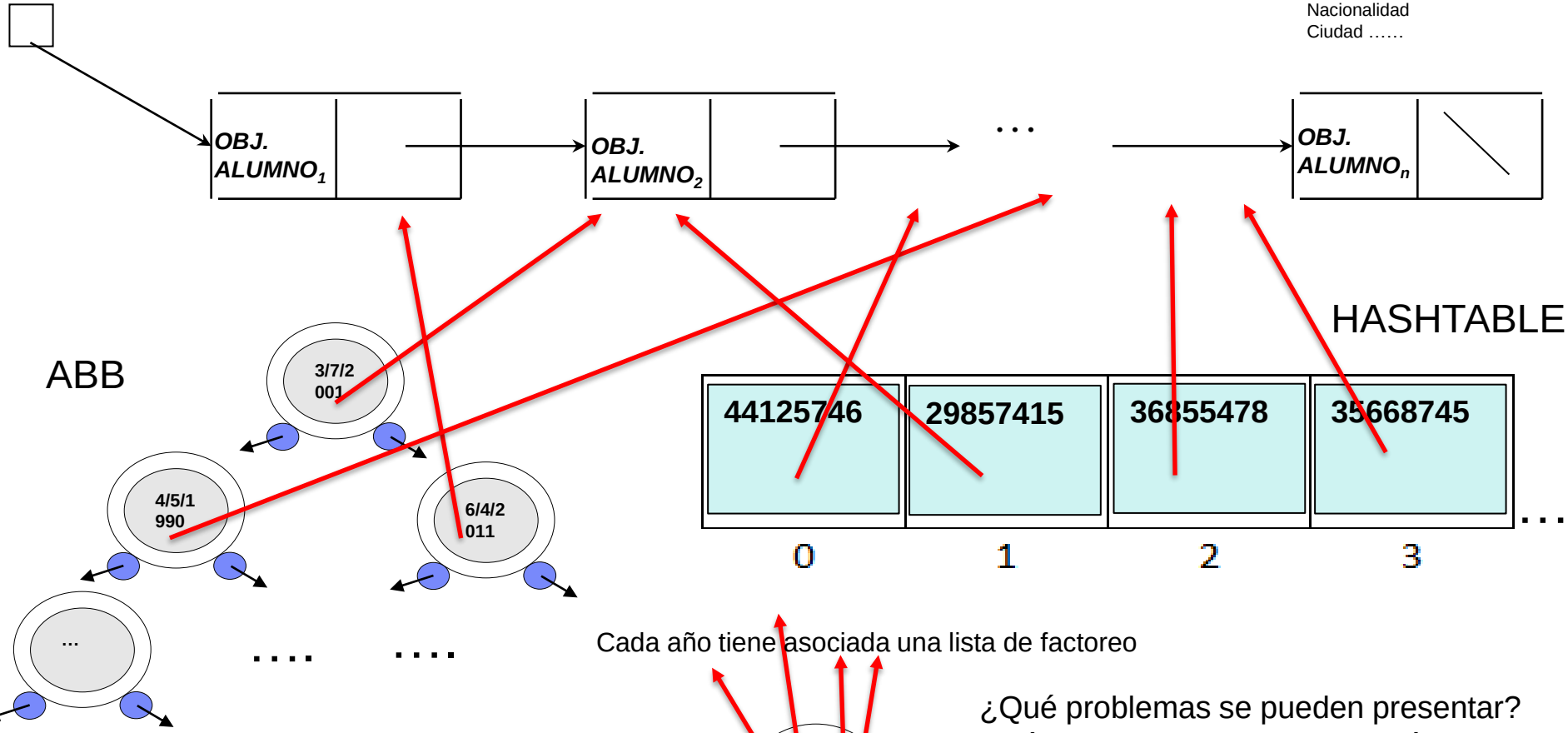
Usos de las Estructuras de Datos

Podemos pensar también que la información principal podría estar guardada de forma ordenada o desordenada en una estructura de datos (por ej. una lista vinculada) y si queremos buscar eficientemente por alguna clave podemos “referenciar esta información”.

2- **Como índices de acceso eficiente a la información**.
Podríamos por ejemplo guardar todos los alumnos en una lista vinculada (podría ser un tipo de archivo en disco por ej, u otra estructura cualquiera).
Y definir una estructura de datos (indice) para buscar por una clave y otra (indice) por la otra clave, y ambas referenciando a la información en la estructura de datos principal de almacenamiento.
Y podría tener varios criterios de búsqueda cada uno con un índice asociado que haga eficiente esa búsqueda.

Usos de las Estructuras de Datos

OBJ. ALUMNO
Nombre
Apellido
DNI
LU
Direccion
Fecha Nacimiento
Nacionalidad
Ciudad ……



¿Por qué use un ABB para la búsqueda por año?

¿Cada año tendría una única referencia asociada?

¿Qué problemas se pueden presentar? Qué pasa si agrego información, o borro información…. Acomodar los índices. Incremento los tiempos, pero gano velocidad de búsqueda.

Tarea:

- Investigar cómo funcionan la clase Java.util.Hashtable y Java.util.HashMap. Por ejemplo en:

https://hg.openjdk.org/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/util/