5. Estructuras avanzadas: colas de prioridad y tablas hash

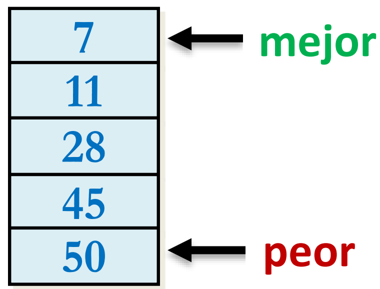
Tema V

Colas de prioridad

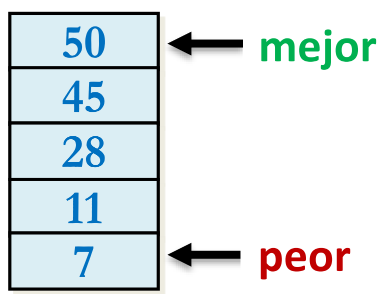
- Una **cola de prioridad** es una estructura de datos que se usa en problemas en los que interesa acceder frecuentemente al elemento "*mejor*" de una colección de valores.
  - Por ejemplo, problemas en los que se mantiene una lista de tareas organizada por prioridades (sistemas operativos, procesos de simulación, colas de un hospital, etc.).
- Los elementos de la colección se deben poder ordenar por algún criterio: **la prioridad**
- Para determinar la prioridad de los elementos se puede usar una función, la **función de prioridad**.

Tipos de colas de prioridad

- Con ordenamiento ascendente
  - El elemento ***mejor*** (más importante) es el de prioridad menor



- Con ordenamiento descendente
  - El elemento ***mejor*** es el de prioridad mayor

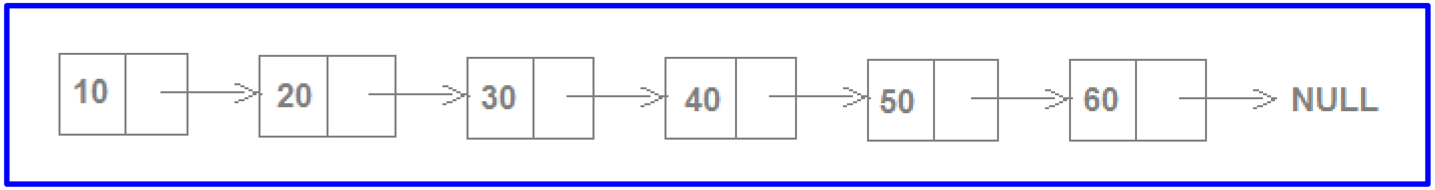


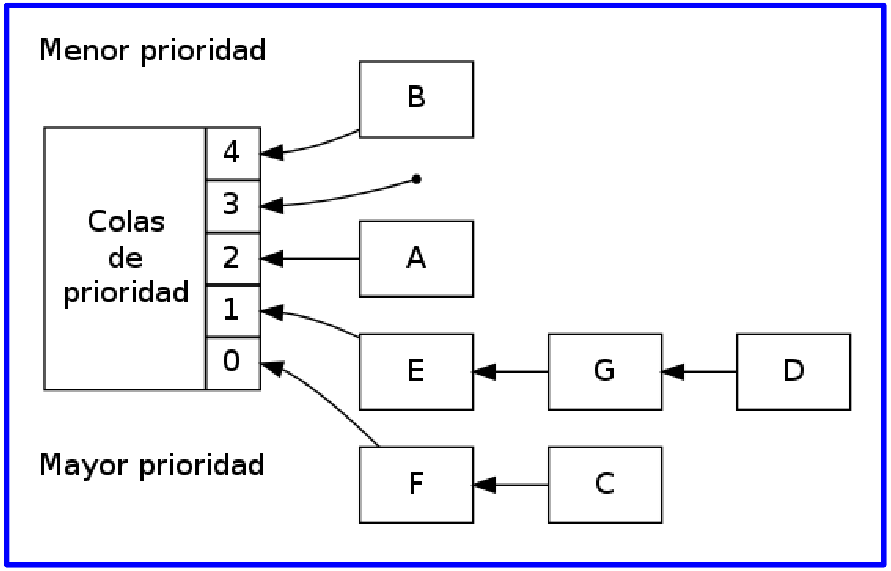
Colas de prioridad - Operaciones

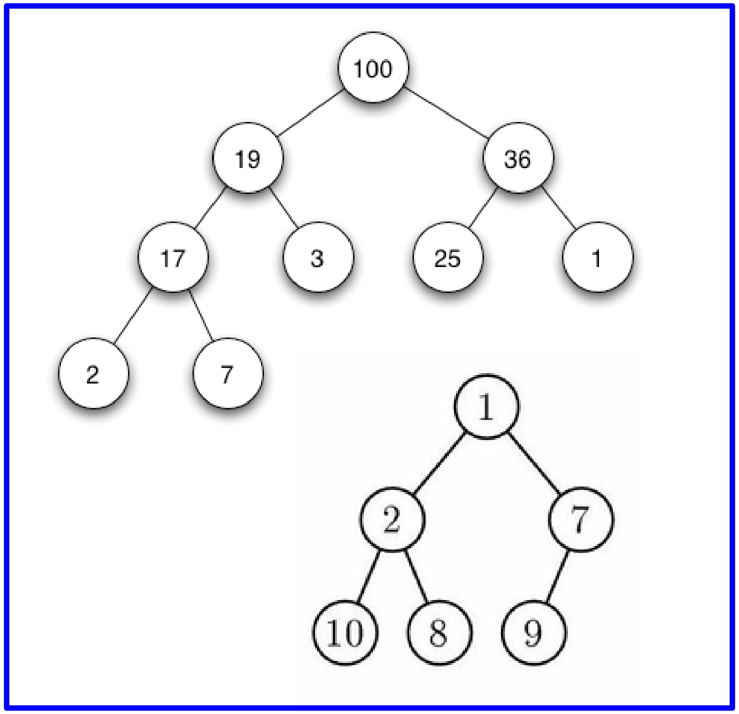
- Las operaciones básicas de una cola de prioridad son:
  - Acceso al elemento de *mejor prioridad**
  - Borrado del elemento de *mejor prioridad**
  - Inserción de un nuevo elemento **ordenado por su prioridad** (el orden puede ser total o parcial)

(*) El elemento mejor depende de si el ordenamiento es ascendente o descendente

Implementaciones





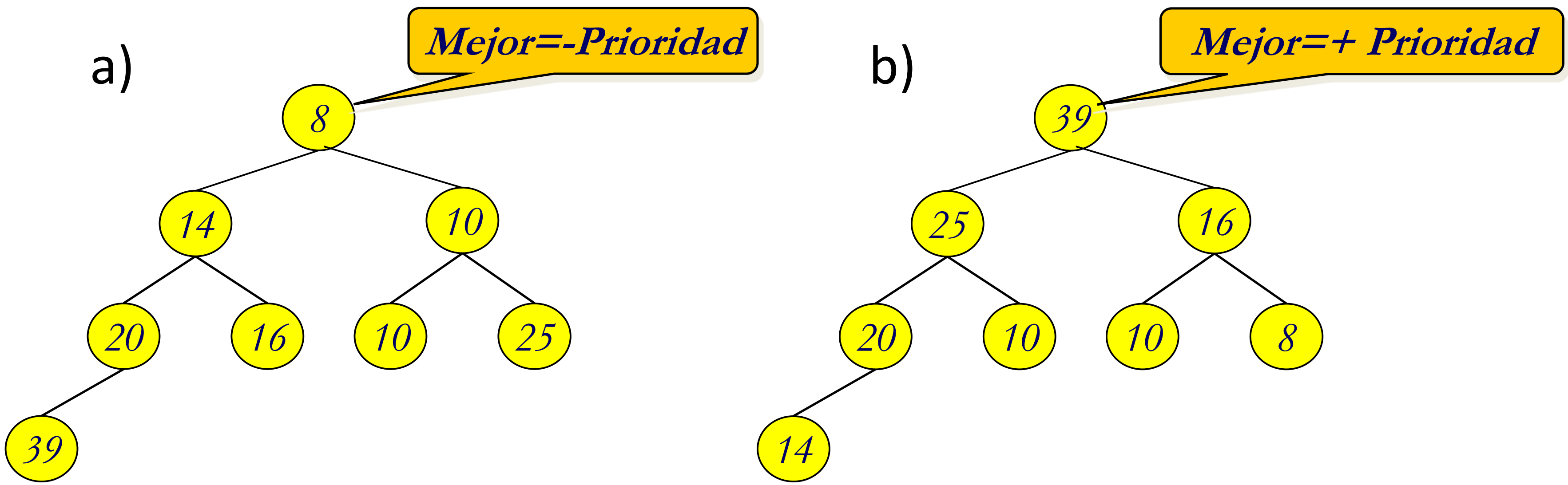


Posibles Representaciones

- **Mediante listas ordenadas**
  Está representación tiene el inconveniente de que la inserción de un elemento es una operación de O(n), aunque la obtención y supresión del elemento de más prioridad se puede realizar en tiempo constante.
- **Mediante árboles de búsqueda equilibrados**
  En este caso todas las operaciones serían de O(log n).
- **Mediante árboles parcialmente ordenados**
  Presentan un coste igual o menor que los anteriores:
  Los accesos se hacen en O(1), los borrados en O(log n) y las inserciones en O(1) de media y O(log n) en el peor caso.
- …

Árboles parcialmente ordenados

- Un **árbol parcialmente ordenado** es un árbol binario **completo** en el que cada nodo tiene una prioridad más alta que la de cada uno de sus hijos.
- La raíz del árbol contiene el elemento de *mejor prioridad*

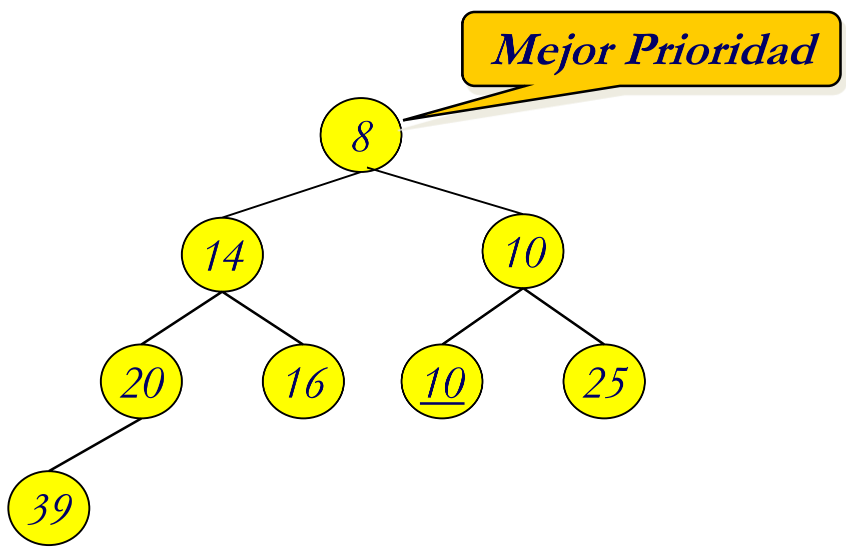


Árboles parcialmente ordenados (II)

- La relación de orden es parcial ya que no todos los nodos están ordenados entre sí (sólo padres e hijos, no los hermanos)
- Dado que el árbol parcialmente ordenado es un árbol binario completo (se va llenando por niveles) , éste se puede representar de forma eficiente mediante un *array*.
  - Aunque en este caso las operaciones de inserción y borrado de la cola de prioridad también son de O(log n), la sencillez de la representación de los árboles parcialmente ordenados y el ahorro de la operación de equilibrado, hacen que en la práctica ésta sea la forma más eficiente de representar colas de prioridad.
  - Una cola de prioridad representada de esta forma se llama **HEAP**.

Representar árboles p. o. en un *array*

- Los nodos del árbol se numeran de la forma siguiente:
  - A la raíz le corresponde el índice 0.
  - Si a un nodo le corresponde el índice k, a sus hijos izquierdo y derecho, si tiene, les corresponden los índices 2*k+1 y 2*k+2
  - Si a un nodo le corresponde el índice k, al padre le corresponde el índice (k-1) div 2 (división entera)

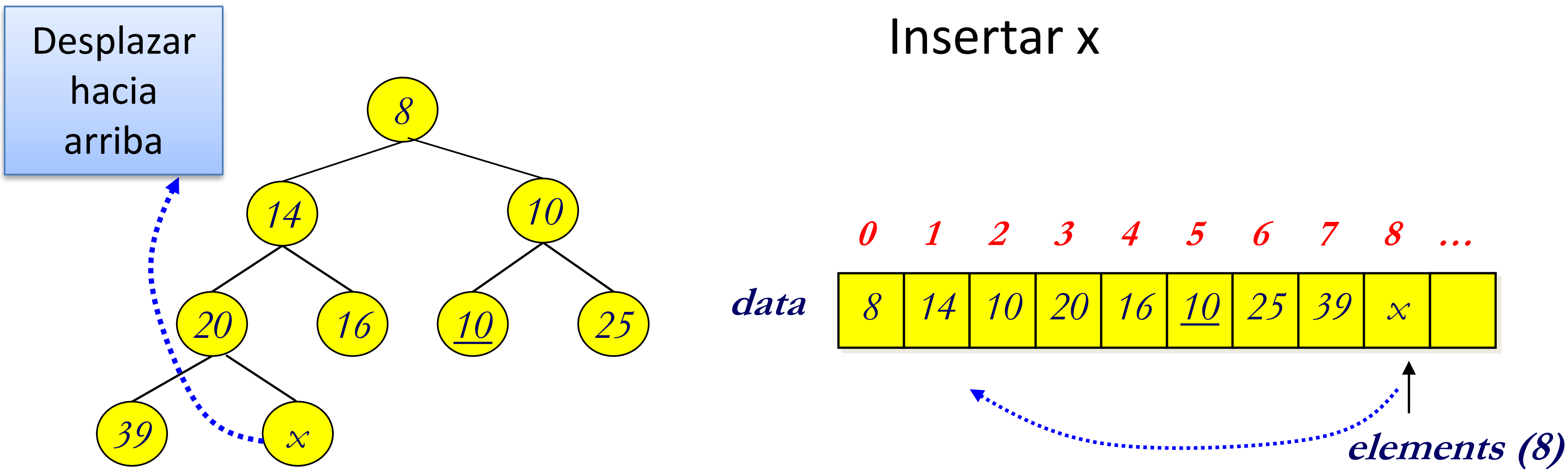


0	1	2	3	4	5	6	7	8	...
8	14	10	20	16	10	25	39		

HEAP: Inserción

- El nuevo elemento se inserta en el último nivel, lo más a la izquierda posible. Posteriormente se desplaza hacia arriba en el árbol hasta que se sitúa en la posición que le corresponde de acuerdo a su prioridad.

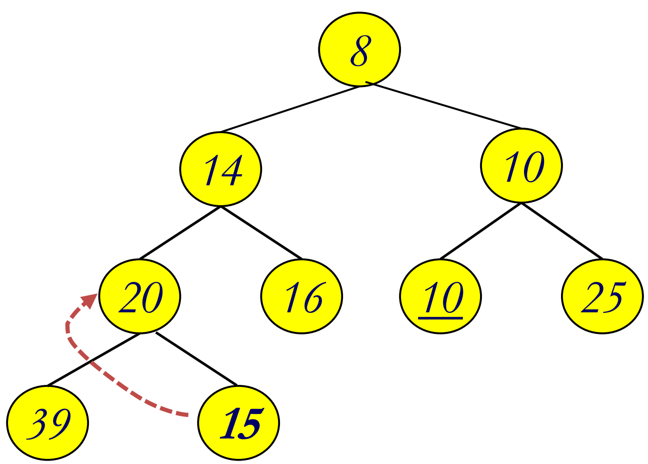
Insertar x



HEAP: Inserción (II)

- El elemento insertado se mueve hacia arriba intercambiándolo con su padre hasta que prioridad(padre_de_x) <= prioridad(x) o hasta que se alcanza la raíz del árbol.

Ej: Insertar 15



elements (9)

	0	1	2	3	4	5	6	7	8	...
data	8	14	10	20	16	10	25	39	**15**	

padre(8)=(8-1) div 2 = 3 --> intercambio

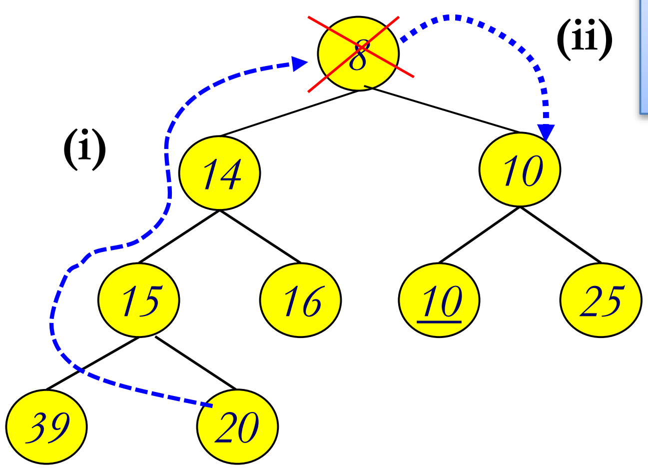
	0	1	2	3	4	5	6	7	8	...
data	8	14	10	**15**	16	10	25	39	20	

padre(3)=(3-1) div 2 = 1 --> no intercambio

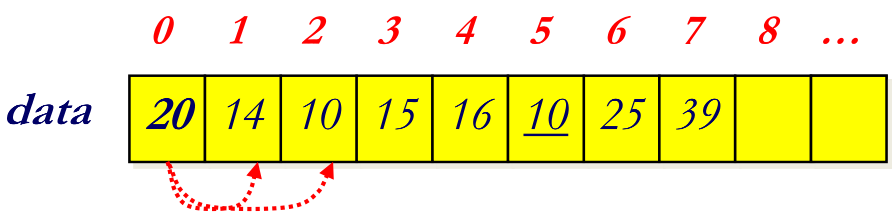
HEAP: Borrado

i. Se borra la raíz del árbol y se pasa el último elemento de la cola a la raíz.

ii. Posteriormente, se desplaza la raíz hacia abajo hasta que se sitúa en el lugar que le corresponde de acuerdo a su prioridad.



Mover hacia abajo



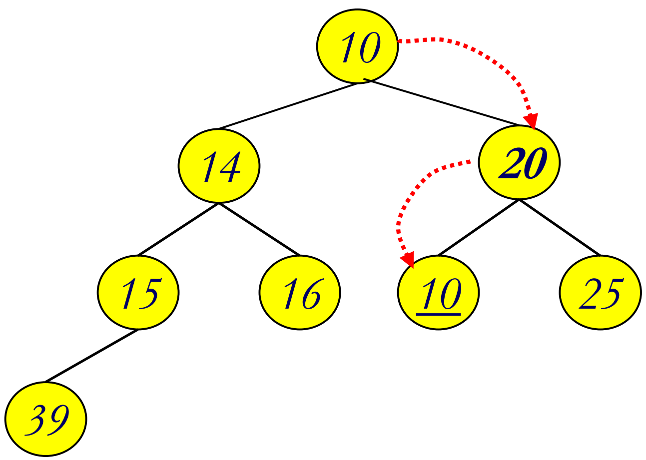
izdo(0)=(2*0)+1 = 1
dcho(0)=(2*0)+2 = 2

--> Intercambio con el 10

elements (8)

HEAP: Borrado (II)

- El elemento de la raíz se desplaza hacia abajo, intercambiándolo con el hijo de menor prioridad, hasta que ambos hijos tengan mayor o igual prioridad o se llegue a una hoja.



	0	1	2	3	4	5	6	7	8	...
data	*10*	*14*	***20***	*15*	*16*	*10*	*25*	*39*		

izdo(2)=(2*2)+1 = 5
dcho(2)=(2*2)+2 = 6

--> Intercambio con el 10

elements (8)

Interfaz para Colas de prioridad

```
public interface PriorityQueue<E> {
        /* Retorna cierto si la cola está vacía */
        boolean isEmpty ();
        /* Retorna el número de elementos de la cola */
        int size ();
        /* Borra todos los elementos de la cola */
        void clear ();
        /* Retorna el elemento más prioritario de la cola */
        E element ();
        /* Extrae y retorna el elemento del frente de la cola */
        E remove ();
        /* Añade el elemento especificado en la cola */
        boolean  add (E e);
  }
```

Clase simple para Colas de prioridad

```
public heap<E> implements PriorityQueue<E> {
        private E[] data;
        private int elements;
        private int compare(E e1, E e2);

        public heap();
        public heap(Comparator<? Super E> c);

        public boolean isEmpty ();
        public int size ();
        public void clear ();
        public E element ();
        public E remove ();
        public boolean  add (E e);
  ... }
```

Compara la **prioridad** de dos elementos

Se puede pasar en el constructor un **comparador** para ordenar los elementos de la cola de manera diferente al *orden natural*

Add (prioridad menor es mejor)

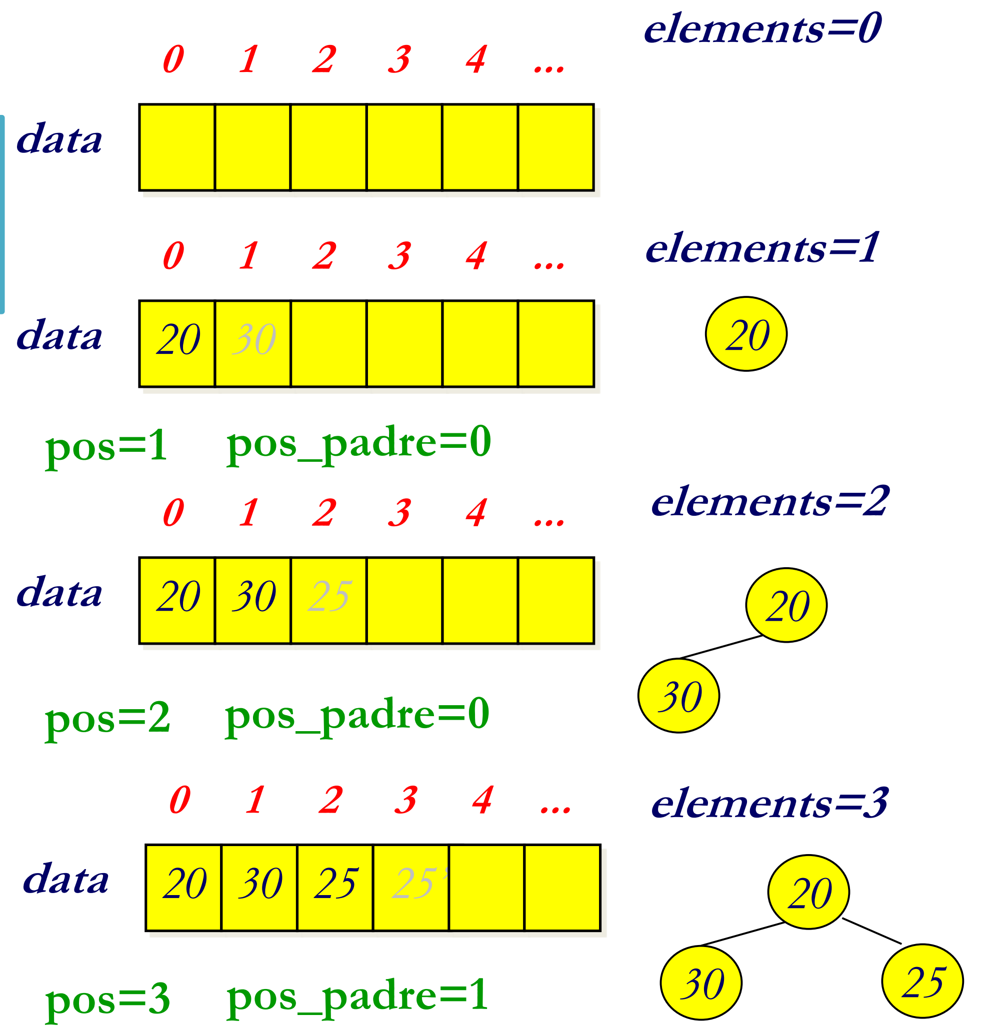
```
public boolean  add (E e)  {
        if (elements+1 == data.length)
                resize(2*data.length+1);        //redimensionamos la cola
        if (elements==0) {    //la cola está vacía
                data[0]=e;
                elements++; }
        else {
                int pos=elements++;             //posición de inserción
                int pos_padre= (pos-1)/2;       //posición del padre
                while (pos>0  && compare(e , data[pos_padre])<0) {
                    data[pos]=data[pos_padre];          //intercambio
                    pos=pos_padre;                      //ir al padre
                    pos_padre= = (pos-1)/2;
                }
                data[pos]=e;
        }
        return true;
}
```

Si la prioridad de ***e*** es menor que la prioridad del ***padre***

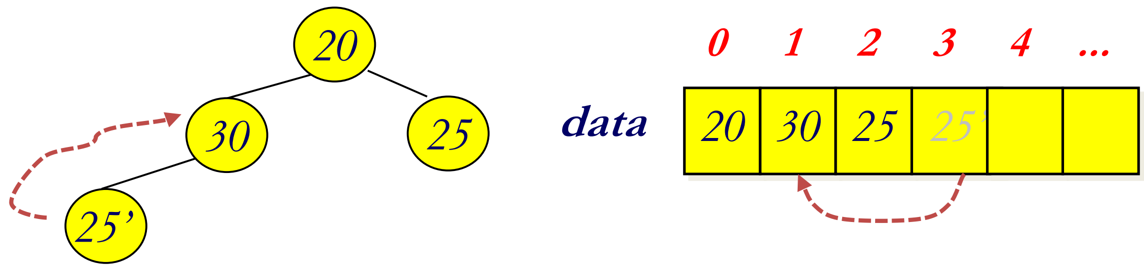
Ej: Insertar estos valores de “e”: 20,30,25,25,10

```
if (elements==0) {    //la cola está vacía
     data[0]=e;
     elements++; }
```

```
int pos=elements++;
int pos_padre= (pos-1)/2;
while (pos>0  &&
          compare(e , data[pos_padre])<0
      //intercambio
      data[pos]=data[pos_padre];
      pos=pos_padre;
      pos_padre= = (pos-1)/2;
 }
```



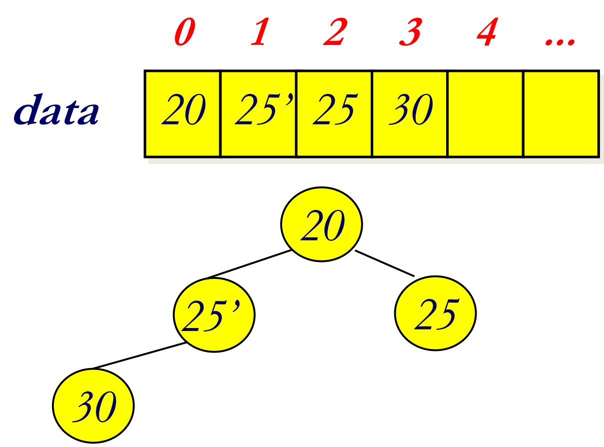
Ej: Insertar estos valores de "e": 20,30,25,25,10



pos=3 pos_padre=1

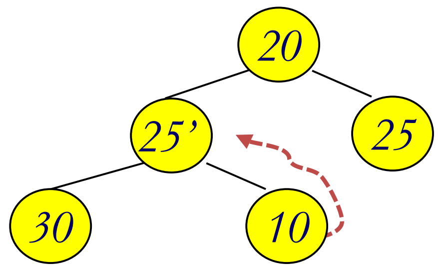
```
int pos=elements++;
int pos_padre= (pos-1)/2;
while (pos>0  &&
        compare(e , data[pos_padre])<0
    //intercambio
    data[pos]=data[pos_padre];
    pos=pos_padre;
    pos_padre= = (pos-1)/2;
}
```

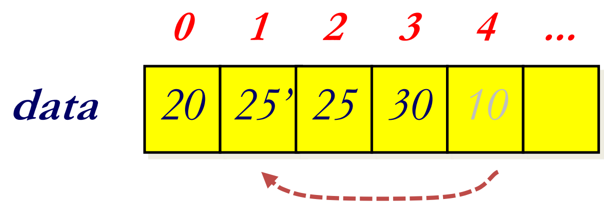
elements=4



Ej: Insertar estos valores de "e": 20,30,25,25,10

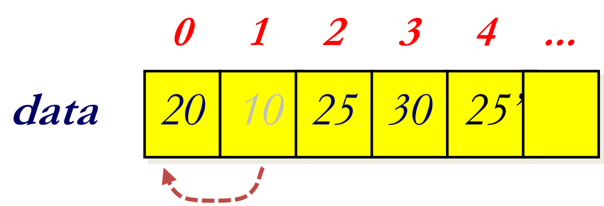
elements=4

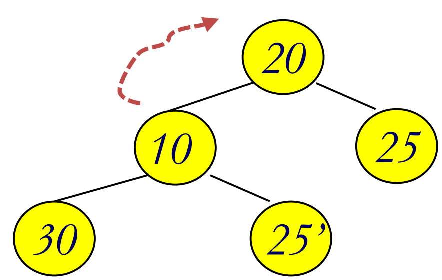




pos=4 pos_padre=1

```
int pos=elements++;
int pos_padre= (pos-1)/2;
while (pos>0  &&
        compare(e , data[pos_padre])<0
    //intercambio
    data[pos]=data[pos_padre];
    pos=pos_padre;
    pos_padre= = (pos-1)/2;
}
```



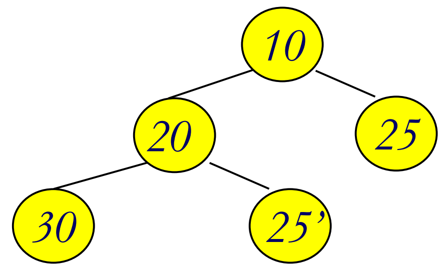


pos=1
pos_padre=0

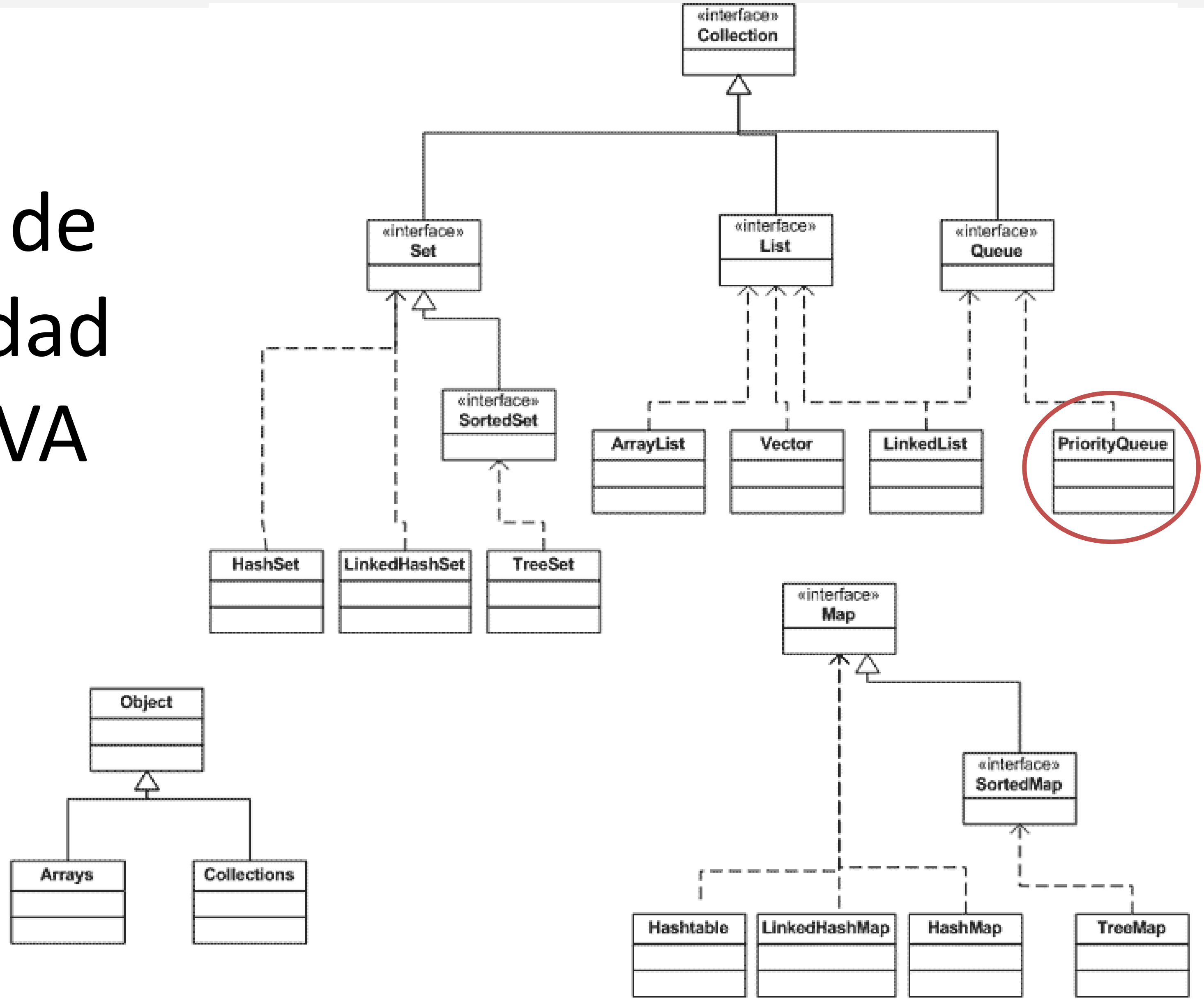
Ej: Insertar estos valores de “e”: 20,30,25,25,10

	0	1	2	3	4	...	
data	10	20	25	30	25’		elements=5

```
int pos=elements++;
int pos_padre= (pos-1)/2;
while (pos>0  &&
        compare(e , data[pos_padre])<0
     //intercambio
     data[pos]=data[pos_padre];
     pos=pos_padre;
     pos_padre= = (pos-1)/2;
 }
```



Colas de prioridad en JAVA



JAVA PriorityQueue

Method Summary	
boolean	`add(E o)` Adds the specified element to this queue.
void	`clear()` Removes all elements from the priority queue.
`Comparator<? super E>`	`comparator()` Returns the comparator used to order this collection, or `null` if this collection is sorted according to its elements natural ordering (using `Comparable`).
`Iterator<E>`	`iterator()` Returns an iterator over the elements in this queue.
boolean	`offer(E o)` Inserts the specified element into this priority queue.
E	`peek()` Retrieves, but does not remove, the head of this queue, returning `null` if this queue is empty.
E	`poll()` Retrieves and removes the head of this queue, or `null` if this queue is empty.
boolean	`remove(Object o)` Removes a single instance of the specified element from this queue, if it is present.
int	`size()` Returns the number of elements in this collection.

Methods inherited from class java.util.AbstractQueue
`addAll`, `element`, `remove`

Tablas Hash

- Las colecciones basadas en **listas** o **árboles** presentan un coste asintótico , para las operaciones de inserción, borrado y test de pertenencia, de O(n) y O(log n), respectivamente.
- El objetivos de las **tablas de dispersión** (**tablas hash**), tanto abiertas como cerradas, consiste en proporcionar un coste menor que el conseguido con listas o árboles.
  - La técnica llamada ***hashing*** pretende conseguir que las operaciones se realicen en **tiempo constante** (en media).
  - En el peor caso, el coste de las operaciones es lineal (proporcional al número de elementos).

Tablas Hash (II)

- Hay dos estrategias de dispersión (*hashing*) diferentes:
  - **Dispersión abierta o externa** (*open addressing*)
    Permite almacenar un número infinito de elementos en la tabla.
  - **Dispersión cerrada o interna.**
    El número de elementos que se puede almacenar está limitado.
- Las operaciones básicas de cualquier tabla de dispersión son: **Inserción**, **Borrado** y **Test de pertenecia**.
- Las tablas hash se idearon como una representación avanzada de conjuntos por lo que **no permiten** elementos repetidos.
- Los elementos almacenados en una tabla hash pueden ser:
  - Simples
  - Pares de la forma (clave, valor)

Principios de funcionamiento

- Para conseguir reducir el coste de las operaciones, **los miembros potenciales de la colección se dividen en un número finitos de clases.**
  - Así, si se desea tener b clases, numeradas de 0 a b-1, los miembros de la colección se distribuyen en las mismas mediante lo que se conoce como la **función de dispersión (h).**
- La función de dispersión **h** aplicada a un elemento x (o a la clave de x si se trata de un par), devuelve un valor entre 0 y b-1 que indica la clase a la que pertenece el elemento:
  - h(x)= 0..b-1 (valor de dispersión)
  - h(clave_de_x)= 0..b-1 (valor de dispersión)

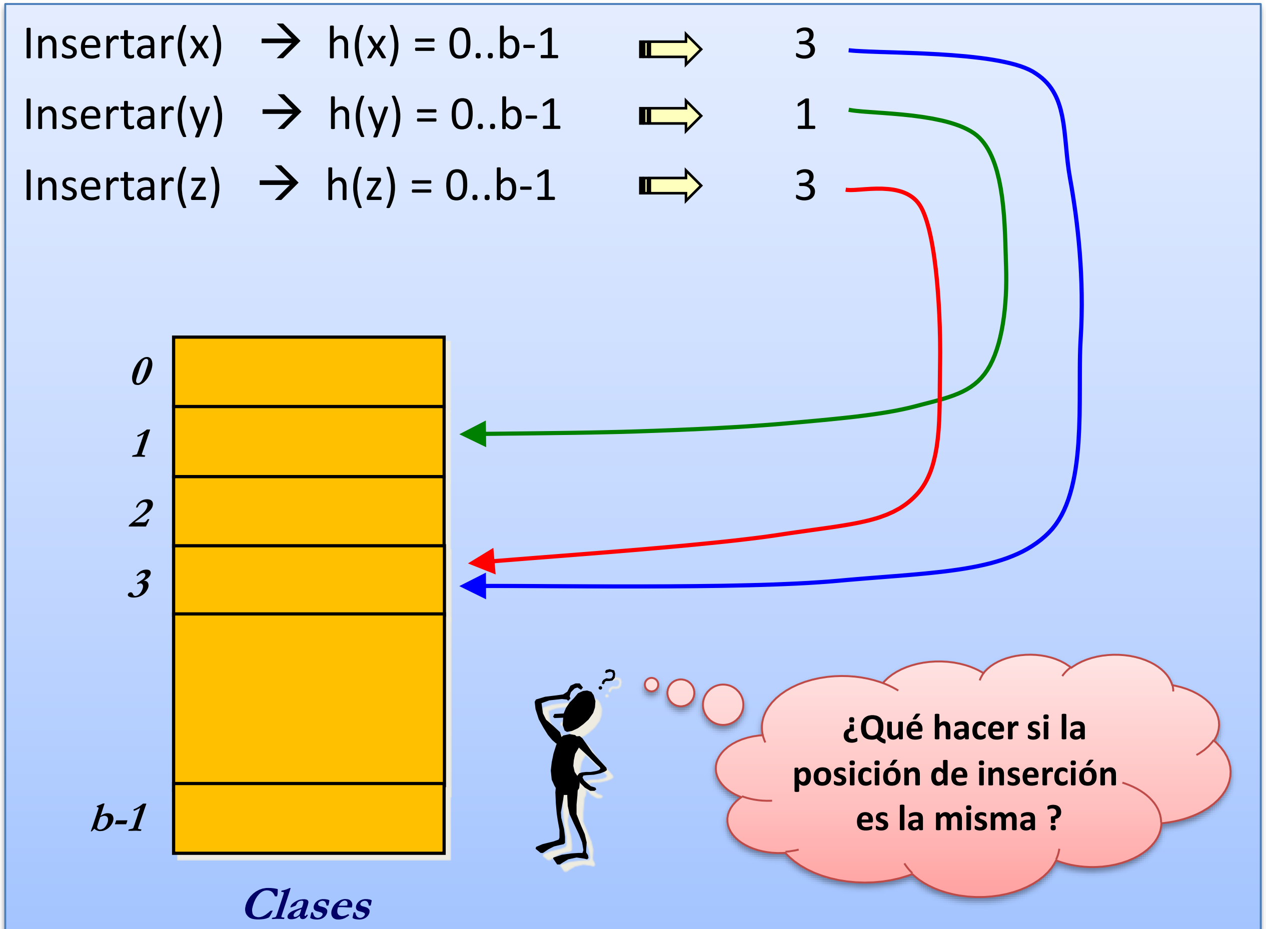
Principios de funcionamiento (II)

- La función de dispersión debe elegirse de forma que reparta lo más uniformemente posible los **n** elementos de la colección entre las **b** clases.
- Hay que tener en cuenta que dos elementos distintos pueden tener el mismo valor de dispersión (**colisión**)→ **¿qué se hace?**

La estrategia seguida en caso de **colisión** es lo que distingue el hashing abierto del hashing cerrado

El tamaño de las tablas hash (**b**) es con frecuencia un **número primo**, ya que de esta manera muchas funciones hash producen **menos colisiones**.

Principios de funcionamiento (III)



Tablas hash abiertas

- Los elementos con el mismo valor de dispersión se almacenan juntos en una **colección** asociada a la clase correspondiente. Estas colecciones reciben el nombre de **cubetas** o **buckets**.
- No existe límite en el número de elementos que se pueden almacenar
- Si la función de dispersión distribuye uniformemente los n elementos de la colección entre las b clases, todas las cubetas contendrán aproximadamente el mismo número de elementos (n/b).
- La relación n/b se denomina **factor de carga.**
- Las tablas hash abiertas admiten factores de carga > 1

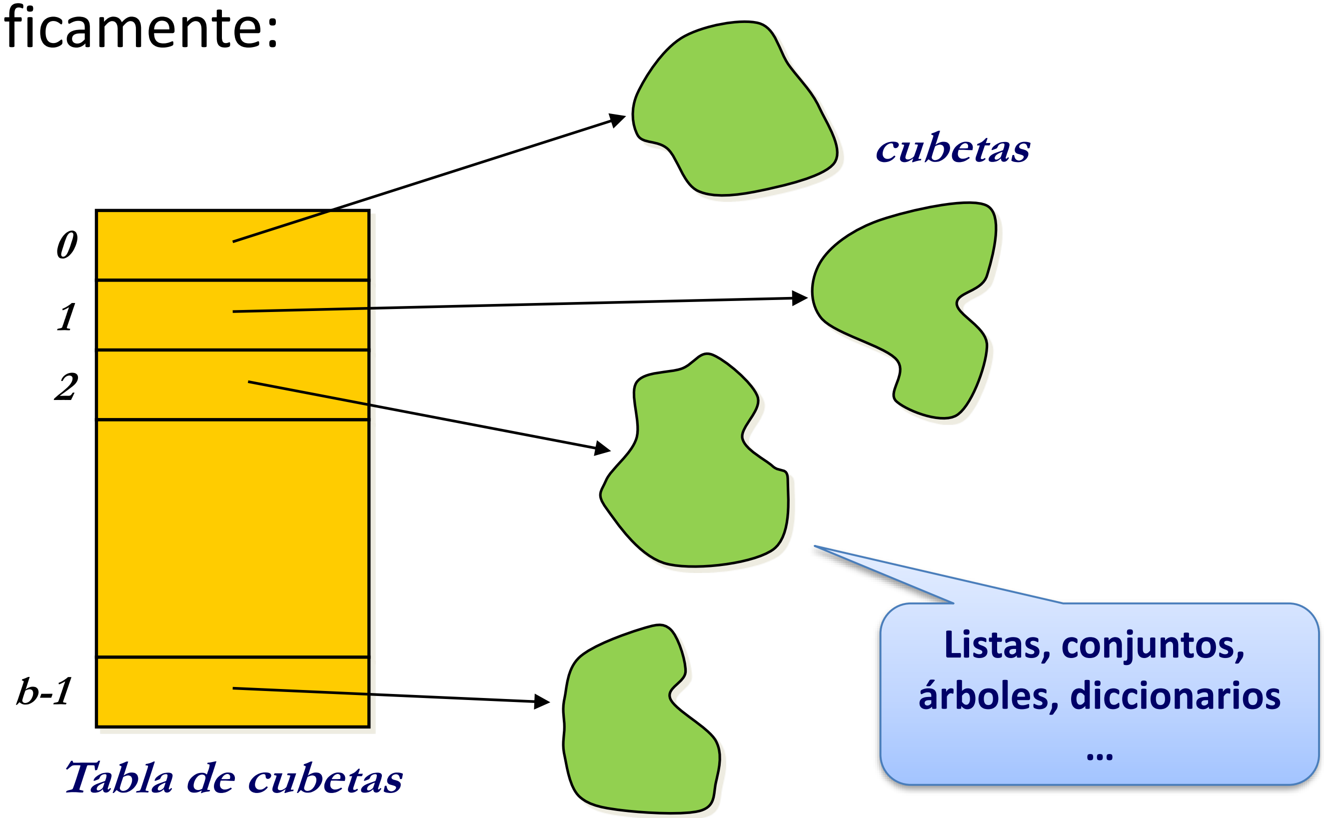
Tablas hash abiertas (II)

- El coste de las operaciones se calcula de la forma siguiente:
  O (h(x)) + O (acceso_colección) + O (tamaño_colección)

  - Si los valores de dispersión se obtienen realizando cálculos simples, el primero de los costes resulta ser de O(1).
  - El acceso a las cubetas (colecciones) será también de O(1) si se almacenan en un array o similar (tabla de cubetas).
  - Por tanto, el coste depende del tamaño de las colecciones. Se suele coger un tamaño de b aproximadamente igual a n para que cada colección tenga entre uno y dos elementos.
  - Con todos estos requisitos, el coste de las operaciones en una tabla hash abierta sería **O(1) en media**

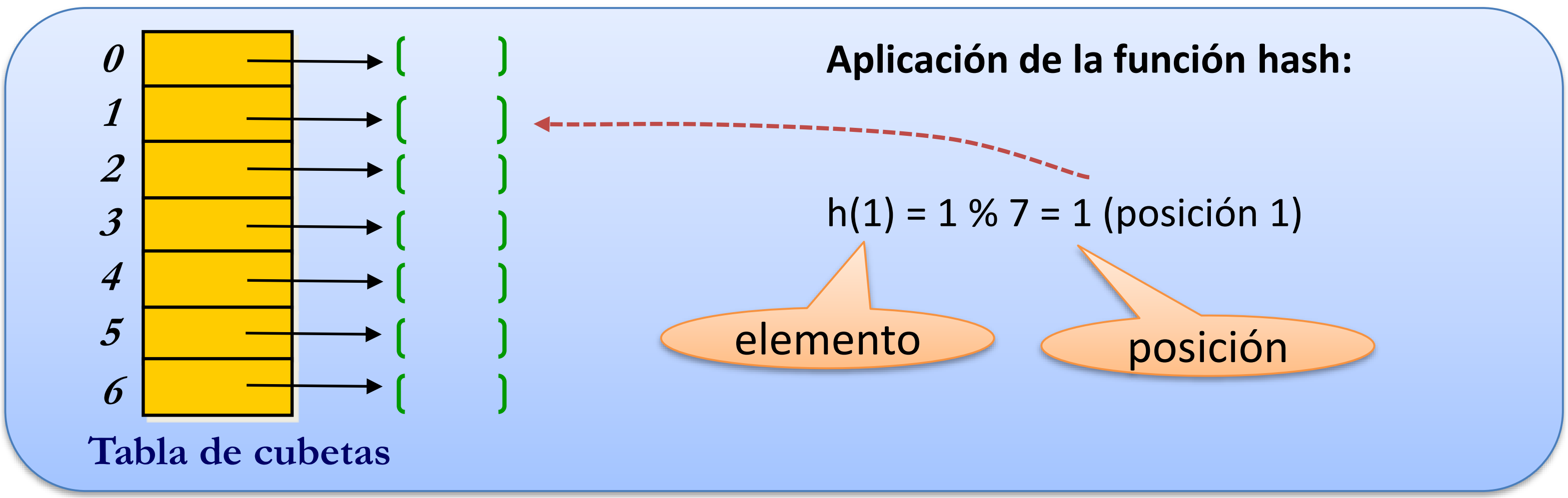
Tablas hash abiertas (III)

- Gráficamente:



Ejemplo de tabla hash abierta

- **Características de la tabla:**
  - Tamaño: b=7
  - Los elementos son números enteros
  - Función hash para enteros: h(x) = x % b
- **Operaciones:**
  - Insertar: 1,7,8,15,20,8



Ejemplo de tabla hash abierta

- **Características de la tabla:**
  - Tamaño: b=7
  - Los elementos son números enteros
  - Función hash para enteros: h(x) = x % b
- **Operaciones:**
  - Insertar: 1,7,8,15,20,8

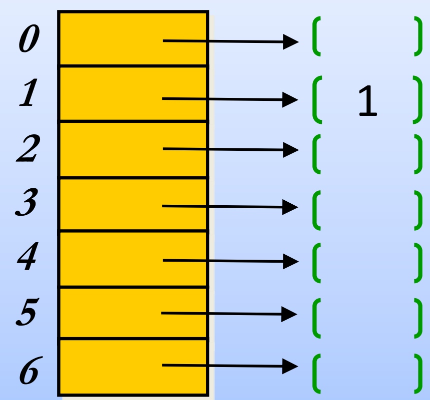


Tabla de cubetas

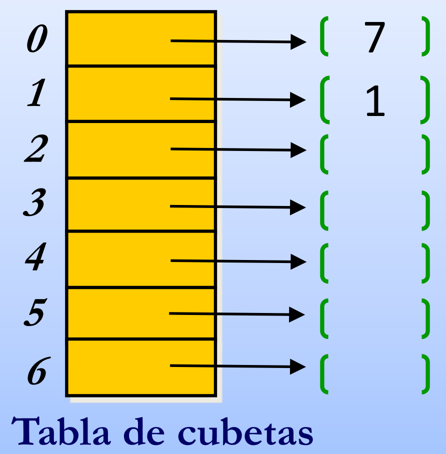
Aplicación de la función hash:

h(1) = 1 % 7 = 1 (posición 1)

h(7) = 7 % 7 = 0 (posición 0)

Ejemplo de tabla hash abierta

- **Características de la tabla:**
  - Tamaño: b=7
  - Los elementos son números enteros
  - Función hash para enteros: h(x) = x % b
- **Operaciones:**
  - Insertar: 1,7,8,15,20,8



Aplicación de la función hash:

h(1) = 1 % 7 = 1 (posición 1)

h(7) = 7 % 7 = 0 (posición 0)

h(8) = 8 % 7 = 1 (posición 1)

En las tablas abiertas puede ir más de un elemento en la misma posición

Ejemplo de tabla hash abierta

- **Características de la tabla:**
  - Tamaño: b=7
  - Los elementos son números enteros
  - Función hash para enteros: h(x) = x % b
- **Operaciones:**
  - Insertar: 1,7,8,15,20,8

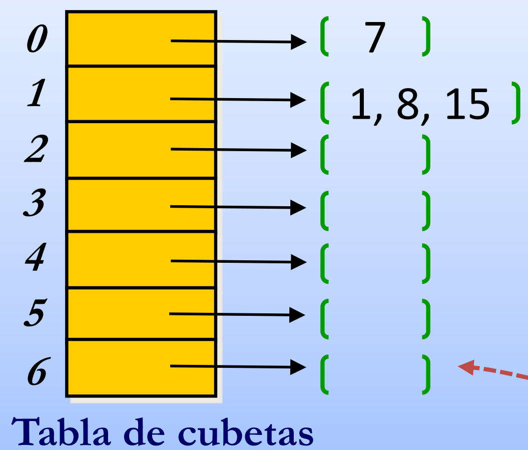
Aplicación de la función hash:

h(1) = 1 % 7 = 1 (posición 1)
h(7) = 7 % 7 = 0 (posición 0)
h(8) = 8 % 7 = 1 (posición 1)
h(15) = 15 % 7 = 1 (posición 1)

En las tablas abiertas puede ir más de un elemento en la misma posición

Ejemplo de tabla hash abierta

- **Características de la tabla:**
  - Tamaño: b=7
  - Los elementos son números enteros
  - Función hash para enteros: h(x) = x % b
- **Operaciones:**
  - Insertar: 1,7,8,15,20,8



Aplicación de la función hash:
h(1) = 1 % 7 = 1 (posición 1)
h(7) = 7 % 7 = 0 (posición 0)
h(8) = 8 % 7 = 1 (posición 1)
h(15) = 15 % 7 = 1 (posición 1)
h(20) = 20 % 7 = 6 (posición 6)

Ejemplo de tabla hash abierta

- **Características de la tabla:**
  - Tamaño: b=7
  - Los elementos son números enteros
  - Función hash para enteros: h(x) = x % b
- **Operaciones:**
  - Insertar: 1,7,8,15,20,8

Aplicación de la función hash:

h(1) = 1 % 7 = 1 (posición 1)
h(7) = 7 % 7 = 0 (posición 0)
h(8) = 8 % 7 = 1 (posición 1)
h(15) = 15 % 7 = 1 (posición 1)
h(20) = 20 % 7 = 6 (posición 6)
h(8) = 8 % 7 = 1 (posición 1) -> REPETIDO

Las tablas hash no admiten elementos repetidos!!!

Ejemplo de tabla hash abierta

- **Características de la tabla:**
  - Tamaño: b=7
  - Los elementos son números enteros
  - Función hash para enteros: h(x) = x % b
- **Operaciones:**
  - Insertar: 1,7,8,15,20,8

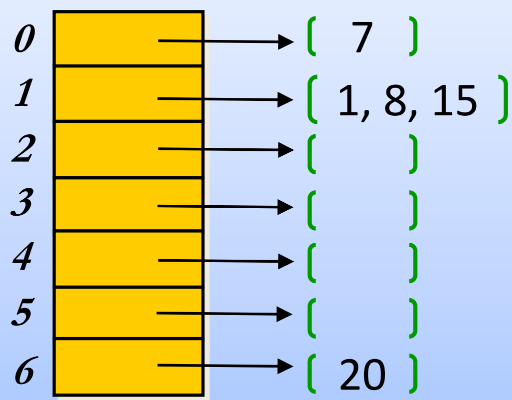


Tabla de cubetas

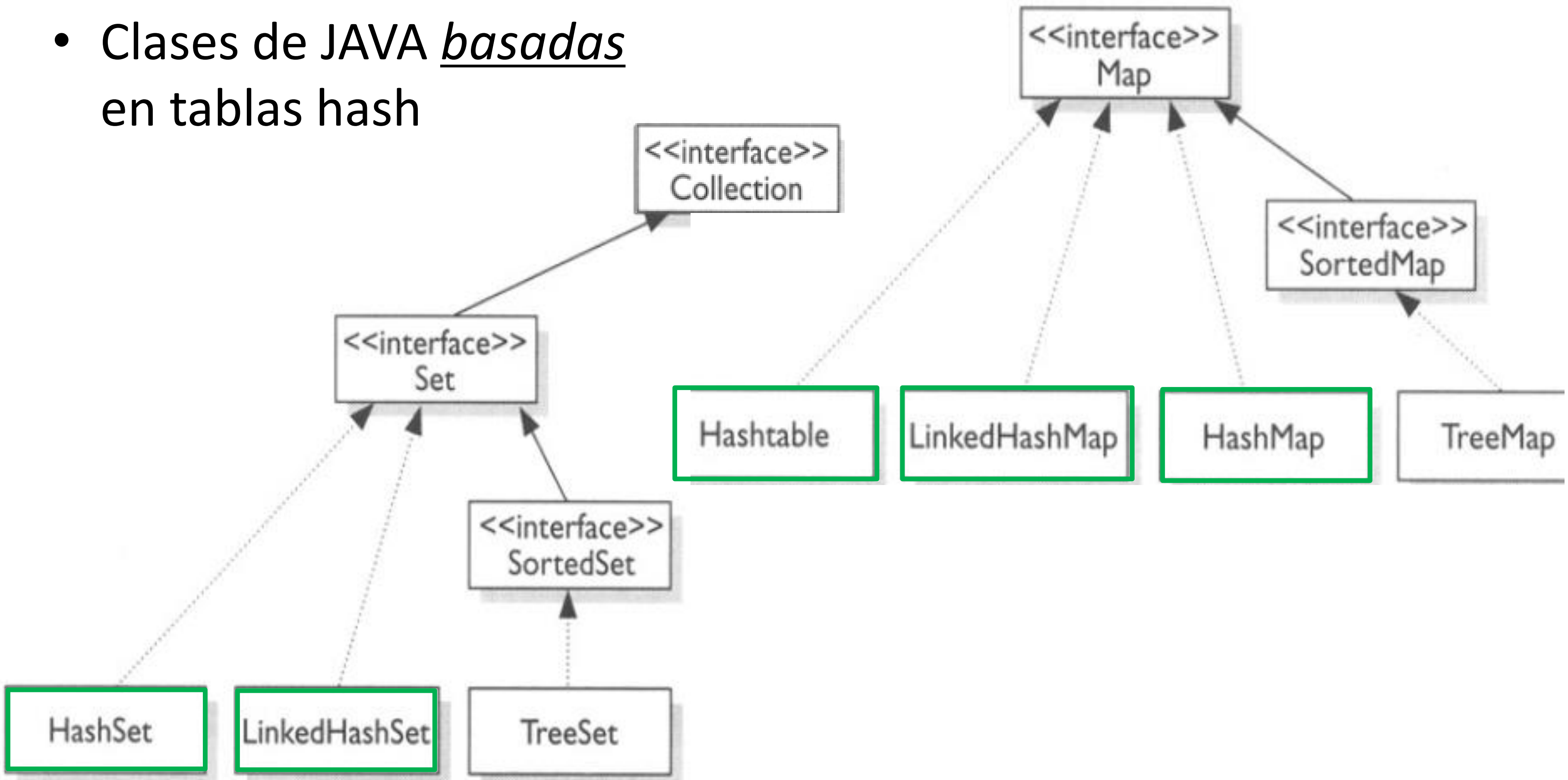
Aplicación de la función hash:

h(1) = 1 % 7 = 1
h(7) = 7 % 7 = 0
h(8) = 8 % 7 = 1
h(15) = 15 % 7 = 1
h(20) = 20 % 7 = 6
h(8) = 8 % 7 = 1 -> repetido!

- Las cubetas pueden ser cualquier colección de enteros
- Los elementos de la tabla hash pueden ser pares (clave, valor).
  Para ello sólo se necesita que las cubetas sean colecciones de pares.

Tablas hash en JAVA

- Clases de JAVA *basadas* en tablas hash



Tablas hash en JAVA (II)

- Las clases de JAVA para tablas hash que implementan la interfaz MAP permiten almacenar pares clave/valor
  - HashMap, LinkedHashMap, Hashtable
- Las clases de JAVA para tablas hash que implementan la interfaz SET permiten almacenar elementos (no pares)
  - HashSet, LinkedHashSet

- Todas las clases implementan internamente la estrategia de ***dispersión abierta.***

Tablas hash cerradas

- Los elementos no se almacenan en colecciones asociadas a las **b** clases (posiciones) sino que **se guardan en la propia tabla**.
- **Sólo hay un elemento por clase.**
- El número de elementos que se pueden almacenar está limitado, por tanto, por el tamaño de la tabla.
- Los elementos con el mismo valor de dispersión generan una **colisión**.
- Las colisiones se resuelven aplicando una **estrategia de redispersión (rehashing),** que consiste en escoger posiciones de la tabla alternativas para los elementos que han producido colisiones.

Tablas hash cerradas (II)

- Así, si se intenta colocar x en la clase h(x) y ésta ya tiene un elemento (colisión), la **estrategia de redispersión** elige una sucesión de cubetas alternativas $h_1(x)$, $h_2(x)$, ... hasta encontrar una libre en la que poder ubicar el elemento.
- Las funciones de dispersión y de redispesión serán tanto mejores cuanto menor número de colisiones originen.
- El factor de carga debe mantenerse en torno al 50% para que el número de colisiones sea aceptable.
- Un tamaño de la tabla primo ayuda a reducir el número de colisiones.
- Las posiciones de la tabla que han tenido elementos y que luego han sido borrados, tienen que considerase **ocupadas** cuando se realizan operaciones sobre la tabla.

Tablas hash cerradas (III)

- **Pertenencia**
  Aplicar hash/rehash hasta encontrar el elemento (pertenece) o encontrar una posición vacía (no pertenece)
- **Borrado**
  Aplicar hash/rehash hasta encontrar el elemento (se borra) o encontrar una posición vacía (fin)
- **Inserción**
  Aplico el algoritmo de pertenencia. Si el elemento pertenece no se inserta (repetidos no) y si no pertenece se inserta en la primera posición no ocupada (**vacía o borrada**).

- **Ejemplo de tabla hash cerrada:**
  - Tamaño: b=7
  - Los elementos son enteros
  - Función hash: $h(x) = x \% b$
  - Función rehash lineal: $h_i(x) = (x+i) \% b$ (i es el nº de colisión)

Insertar: 1, 7, y 8

Tabla inicial

	Tabla de elementos	Tabla de estados
0		vacío
1		vacío
2		vacío
3		vacío
4		vacío
5		vacío
6		vacío

elemento

posición

$h(1) = 1 \% 7 = 1$

$h(7) = 7 \% 7 = 0$

$h(8) = 8 \% 7 = 1 \rightarrow$ colisión 1

$h_1(8) = (8+1)\%7 = 2$

Se aplica la función de redispersión con i=número de colision

Un elemento por posición!!

	Tabla de elementos	Tabla de estados
0	7	ocupado
1	1	ocupado
2	8	ocupado
3		vacío
4		vacío
5		vacío
6		vacío

- **Ejemplo de tabla hash cerrada:**
  - Tamaño: b=7
  - Los elementos son enteros
  - Función hash: h(x) = x % b
  - Función rehash lineal: $h_i(x) = (x+i) \% b$ (i es el nº de colisión)

<u>Insertar: 15</u>

Tabla inicial

	Tabla de elementos	Tabla de estados
0		vacío
1		vacío
2		vacío
3		vacío
4		vacío
5		vacío
6		vacío

$h(15) = 15 \% 7 = 1 \rightarrow$ colisión 1
$h_1(15) = (15+1)\%7 = 2 \rightarrow$ colisión 2
$h_2(15) = (15+2)\%7 = 3$

El número de colisión empieza en 1 para cada nueva operación

	Tabla de elementos	Tabla de estados
0	7	ocupado
1	1	ocupado
2	8	ocupado
3	15	ocupado
4		vacío
5		vacío
6		vacío

Borrar: 8

	Tabla de elementos	Tabla de estados
0	7	ocupado
1	1	ocupado
2	8	ocupado
3	15	ocupado
4		vacío
5		vacío
6		vacío

$h(8) = 8 \% 7 = 1 \rightarrow$ colisión (ocupado)
$h_1(8) = 2 \rightarrow$ se borra

	Tabla de elementos	Tabla de estados
0	7	ocupado
1	1	ocupado
2	8	borrado
3	15	ocupado
4		vacío
5		vacío
6		vacío

Si el elemento no está se para al llegar a una posición *vacía.* Las posiciones borradas se consideran como ocupadas

Las búsquedas son exactamente igual que los borrados pero devolviendo *true* o *false*

Insertar: 22 (Insertar con posiciones borradas)

$h(22) = 22 \% 7 = 1$ → colisión 1 (ocupado)

Las posiciones borradas se tratan como ocupadas

$h_1(22) = (22+1)\%7 = 2$ → colisión 2 (borrado)

$h_2(22) = (22+2)\%7 = 3$ → colisión 3 (ocupado)

$h_3(22) = (22+3)\%7 = 4$ → vacío (inserto en el primer “hueco”)

Si llego a una posición vacía es porque el elemento no está y puedo meterlo

Pero OJO: Se inserta en la primera posición borrada/vacía que haya

	Tabla de elementos	Tabla de estados
0	7	ocupado
1	1	ocupado
2	8	borrado
3	15	ocupado
4		vacío
5		vacío
6		vacío

	Tabla de elementos	Tabla de estados
0	7	ocupado
1	1	ocupado
2	22	ocupado
3	15	ocupado
4		vacío
5		vacío
6		vacío

Funciones de dispersión

- La aplicación de funciones de dispersión requiere, en la mayor parte de los casos, realizar dos procesos:
  1. **Transformación de la clave en un valor entero.**
  2. **Convertir el entero resultante en un índice válido de la tabla de dispersión.**
- Para acometer la primera tarea se puede utilizar alguno de los métodos siguientes: **transformación, agrupación, desplazamiento o conversión.**
- Para obtener un índice válido simplemente se calcula el **resto de la división** por el tamaño de la tabla (tamaño que debe ser un número primo).

Funciones de dispersión (II)

- Hemos visto una función hash para enteros, pero en muchas ocasiones los elementos (o claves) de las tablas son cadenas de caracteres.
- Existen muchas funciones hash para cadenas. Por ejemplo, para una cadena **s** con un longitud **n** ,el método *hashCode* para el tipo String de JAVA se calcula como:

hash=s[0]*31^(n-1) + s[1]*31^(n-2) + … + s[n-1]

- En JAVA, para obtener la posición de un elemento en una tabla hash basta con llamar al método *hashCode* de ese elemento y calcular el resto de división entera por el tamaño de la tabla. Para una cadena **s**:

pos = s.hashCode( ) % b

Funciones de redispersión

- La estrategia de resolución de colisiones más simple es la **redispersión lineal**

$$h_i(x) = (x+i) \% b \quad \text{(i es el nº de colisión)}$$

  – Tiene el inconveniente de la agrupación de posiciones llenas en bloques grandes consecutivos
  – A priori, podría pensarse en obtener un comportamiento más aleatorio mediante sondeos a intervalos constantes mayores que uno (**k**)

$$h_i(x) = (x+k*i) \% b$$

  – Un problema de esta estrategia es que si **k** y **b** tienen un factor común mayor que uno, **no permite buscar en toda la tabla**. Además, el problema previo no desaparece, lo único es que los bloques estarían separados por la distancia **k**.

Funciones de redispersión (II)

- Una estrategia de resolución simple que elimina el problema de amontonamiento de la dispersión lineal, es la **dispersión cuadrática**

$$h_i(x) = (x+i^2)\ \%\ b \quad (\text{i es el nº de colisión})$$

  - Ésta es una estrategia bastante popular, pero que tiene el inconveniente de no garantizar el sondeo de todas las cubetas vacías cuando el porcentaje de ocupación de la tabla es superior al 50%.

Funciones de redispersión (III)

- Por último, veremos la estrategia de resolución de colisiones conocida como doble hashing:

$$h_i(x) = (hash_1(x) + i * hash_2(x)) \% b$$

  – Es importante escoger bien la segunda función de dispersión. Es fácil ver que el valor de ésta nunca debe ser 0 ya que se obtendría de nuevo el valor de dispersión correspondiente a la función de dispersión principal.

  – En general las funciones de la forma

$$hash_2(x) = R - (x \% R)$$

  donde **R** es un número primo menor que **b** se comportan bastante bien.