

**Guillermo Román**

`guillermo.roman@upm.es`

**Lars-Åke Fredlund**

`larsake.fredlund@upm.es`

**Manuel Carro**

`manuel.carro@upm.es`

**Marina Álvarez**

`marina.alvarez@upm.es`

**Julio García**

`juliomanuel.garcia@upm.es`

**Tonghong Li**

`tonghong.li@upm.es`

- Fechas de entrega y penalización asociada:

Hasta el miércoles 26 de octubre, 23:59 horas	0 %
Hasta el jueves 27 de octubre, 23:59 horas	20 %
Hasta el viernes 28 de octubre, 23:59 horas	40 %

Después la puntuación máxima será 0
- Se comprobará plagio y se actuará sobre los detectados.
- Usad las horas de tutoría para preguntar sobre programación – son oportunidades excelentes para aprender.

# Entrega

- Todos los ejercicios de laboratorio se deben entregar a través de

`http://deliverit.fi.upm.es`

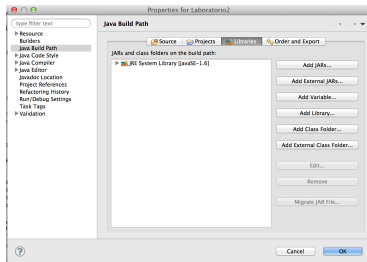
- Los ficheros que hay que subir es `HashTable.java`.

# Configuración previa

- Arrancad Eclipse
- Podéis utilizar cualquier versión reciente de Eclipse. Es suficiente con que instaléis la *Eclipse IDE for Java Developers*.
- Cambiad a “Java Perspective”.
- Debéis tener instalado al menos Java JDK 8.
- Cread un proyecto Java llamado aed:
  - ▶ Seleccionad separación de directorios de fuentes y binarios.
  - ▶ **No debéis elegir la opción de crear el fichero** module-info.java
- Cread un *package* aed.hashtable en el proyecto aed, dentro de src
- Aula Virtual → AED → Laboratorios → Laboratorio 3 → Laboratorio3.zip; descomprimidlo
- Contenido de Laboratorio3.zip:
  - ▶ HashTable.java, TesterLab3.java

# Configuración previa

- Importad al paquete `aed.hashtable` los fuentes que habéis descargado (`HashTable.java`, `TesterLab3.java`)
- Si no lo habéis hecho, añadid al proyecto `aed` la librería `aedlib.jar` que tenéis en Moodle (en Laboratorios).

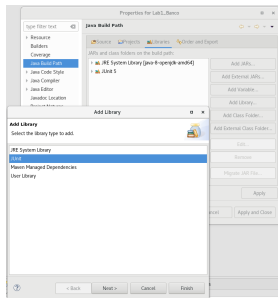


Para ello:

- Project → Properties → Java Build Path. Se abrirá una ventana como la de la izquierda
- Usad la opción “Add External JARs...”.
- Si vuestra instalación distingue `ModulePath` y `ClassPath`, instalad en `ClassPath`

# Configuración previa

- Si no lo habéis hecho, añadid al proyecto aed la librería JUnit 5



Para ello:

- Project → Properties → Java Build Path. Se abrirá una ventana como la de la izquierda;
- Usad la opción “Add Library...” → Seleccionad “JUnit” → Seleccionad “JUnit 5”
- Si vuestra instalación distingue ModulePath y ClassPath, instalad en ClassPath

- En la clase TesterLab3 tenéis las pruebas, para ejecutarlas, abrid el fichero TesterLab3, pulsando el botón derecho sobre el editor, seleccionar “Run as...” → “JUnit Test”
- NOTA: Si al ejecutar no aparece la vista “JUnit”, podéis incluirla en “Window” → “Show View” → “Java” → “JUnit”

# Documentación de la librería aedlib.jar

- La documentación de la API de aedlib.jar esta disponible en <http://costa.ls.fi.upm.es/teaching/aed/docs/aedlib/>
- También se puede añadir la documentación de la librería a Eclipse (*no es obligatorio*): en el “Package Explorer”: “Referenced Libraries” → aedlib.jar y elige la opción “Properties”. Se abre una ventana donde se puede elegir “Javadoc Location” y ahí se pone como “javadoc location path:”

<http://costa.ls.fi.upm.es/teaching/aed/docs/aedlib/>  
y presionar el boton “Apply and Close”

# Tarea de hoy: implementar un *map* con una tabla de dispersión

- Un *map* es una estructura de datos que guarda valores (de tipo V) asociados a claves (de tipo K)
- Interfaz Map:

```
public interface Map<K,V> extends Iterable<Entry<K,V>> {  
    public boolean isEmpty();  
    public int size();  
    public boolean containsKey(Object key);  
    public V get(K key);  
    public V put(K key, V value);  
    public V remove(K key);  
    public Iterable<K> keys();  
    public Iterable<Entry<K,V>> entries();  
}
```



# Ejemplo de funcionamiento

```
HashTable<Integer,String> h =  
    new HashTable<Integer,String>(5); // size 5, key Integer,  
                                       // value String  
  
h.size();           => 0           // returns number of entries  
h.put(2,"Hola");    => null        // no previous value for key 2  
h.put(2,"Hi");      => "Hola"     // returns the previous value  
                                       // for key 2  
  
h.size();           => 1           // only one value per key  
h.get(3);           => null        // no value with key 3 exists  
h.get(2);           => "Hi"  
h.remove(2);        => "Hi"       // returns the value associated  
                                       // with the removed key  
  
h.size();           => 0           // The entry with key 2 was removed  
  
h.put(1,"dulce"); h.put(5,"navidad");  
  
Iterable<Integer> it = h.keys(); // returns iterable over keys  
for(Integer i : h.keys()) {print(i);} // prints 1 5
```

## Tarea de hoy: implementar una “Hash Table”

- Completar `HashTable.java`: implementar *Map* con una “Hash Table” (tabla de dispersión) con “open addressing” para resolver colisiones.
- Atributo `buckets` dentro `HashTable.java`: almacena los datos de la “hash table” dentro un array:

```
Entry<K,V>[] buckets;
```

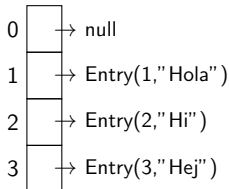
- Las celdas del array –los “buckets”– guardan “entradas” `Entry<K,V>` que asocian un valor (tipo `V`) con una clave (tipo `K`).
- Interfaz de `Entry`:

```
public interface Entry<K,V> {  
    public K getKey();  
    public V getValue();  
}
```

- `EntryImpl` implementa el interfaz `Entry`. Tiene un constructor `EntryImpl(K key, V value)`.

## Ejemplo Hash Table

- Supongamos que buckets tiene tamaño 4, las claves son de tipo Integer y los valores son de tipo String:



- Determinar dónde guardar Entry: calcular “hash” de clave y aplicar función de compresión para “reducirlo” al tamaño del vector.
- Por ejemplo:

$$index(key) \equiv abs(key.hashCode()) \bmod size(table)$$

- NOTA: Hay mejores funciones de compresión/dispersión

# Ejemplo Calculo de Índices

Supongamos:

- Tamaño tabla = 4.
- Claves ( $n$ ) son Integer.
- El *hash* de un entero es el mismo entero.

Integer( $n$ )	$n$ .hashCode()	index( $n$ )
1	1	1
2	2	2
3	3	3
4	4	0
10	10	2

# Inserción

Para insertar una clave y valor: *buscar el bucket libre mas cercano al lugar “preferido” por la clave.*

- Calcular indice del bucket donde debería ir la clave.
  - ▶ Si `buckets[indice]` esta vacío (`null`), insertamos allí.
  - ▶ Si no, recorremos el vector circularmente buscando el siguiente bucket vacío.
  - ▶ Si no hay ningun bucket vacio:
    - ★ Crear un array más grande.
    - ★ Insertar todas las entradas ahí.
    - ★ Repetir la búsqueda.
    - ★ Insertar (ahora sí debe haber espacio).

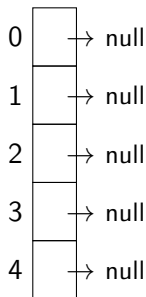
## Simulacro insertar – “open addressing”

- Mostramos la inserción de claves y valores en una tabla de tamaño 5:

0	→ null
1	→ null
2	→ null
3	→ null
4	→ null

## Simulacro insertar – “open addressing”

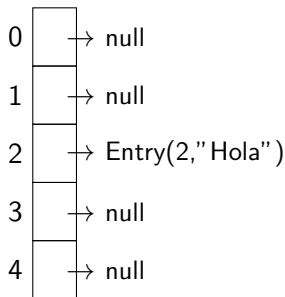
- Mostramos la inserción de claves y valores en una tabla de tamaño 5:



- Llamada a `put(2, "Hola")`
- Calculamos el “bucket” donde debería estar:  
 $\text{index}(2) \equiv 2$ , porque  $2.\text{hashCode}() \equiv 2$  y  $2 \bmod 5 \equiv 2$

## Simulacro insertar – “open addressing”

- Mostramos la inserción de claves y valores en una tabla de tamaño 5:

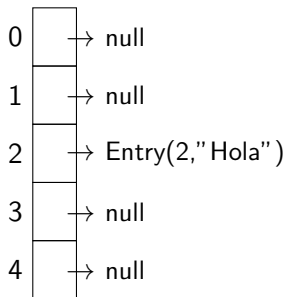


- Llamada a `put(2, "Hola")`
- Calculamos el “bucket” donde debería estar:  
 $\text{index}(2) \equiv 2$ , porque  $2.\text{hashCode}() \equiv 2$  y  $2 \bmod 5 \equiv 2$
- Como el “bucket” 2 está vacío, insertamos `Entry(2, "Hola")` allí.



## Simulacro insertar – “open addressing”

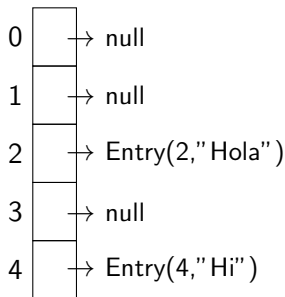
- Mostramos la inserción de claves y valores en una tabla de tamaño 5:



- Llamada a `put(4, "Hi")`
- Calculamos donde debería estar el “bucket”:  
 $\text{index}(4) \equiv 4$ , porque  $4.\text{hashCode}() \equiv 4$  y  $4 \bmod 5 \equiv 4$

## Simulacro insertar – “open addressing”

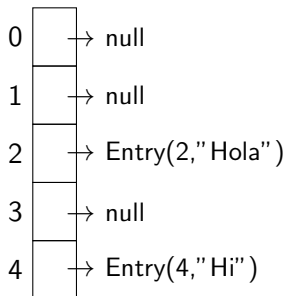
- Mostramos la inserción de claves y valores en una tabla de tamaño 5:



- Llamada a `put(4, "Hi")`
- Calculamos donde debería estar el “bucket”:  
 $\text{index}(4) \equiv 4$ , porque  $4.\text{hashCode}() \equiv 4$  y  $4 \bmod 5 \equiv 4$
- Como el “bucket” 4 está vacío, insertamos `Entry(4, "Hi")` allí.

## Simulacro insertar – “open addressing”

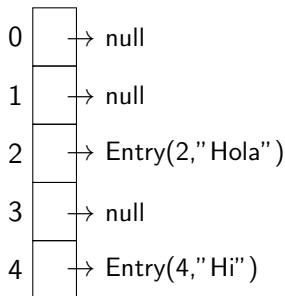
- Mostramos la inserción de claves y valores en una tabla de tamaño 5:



- Llamada a `put(7, "Privet")`
- Calculamos en qué “bucket” debería estar:  
 $\text{index}(7) \equiv 2$ , porque  $7.\text{hashCode}() \equiv 7$  y  $7 \bmod 5 \equiv 2$

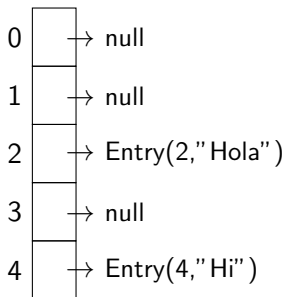
## Simulacro insertar – “open addressing”

- Mostramos la inserción de claves y valores en una tabla de tamaño 5:



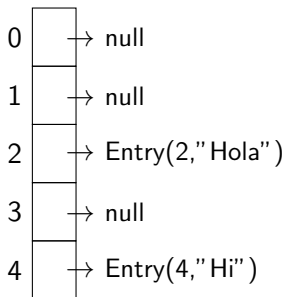
- Llamada a `put(7, "Privet")`
- Calculamos en qué “bucket” debería estar:  
 $\text{index}(7) \equiv 2$ , porque  $7.\text{hashCode}() \equiv 7$  y  $7 \bmod 5 \equiv 2$
- Intentamos insertar `Entry(7, "Hi")` en el “bucket” 2
- Pero “bucket” 2 no está vacío: *colisión*. ¿Qué hacer?

## Insertar: manejando colisiones – “open addressing”



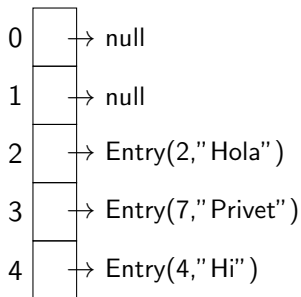
- **Idea:** localizar el primer “bucket” libre tras el bucket deseado e insertar la entrada ahí con una *búsqueda circular*.

## Insertar: manejando colisiones – “open addressing”



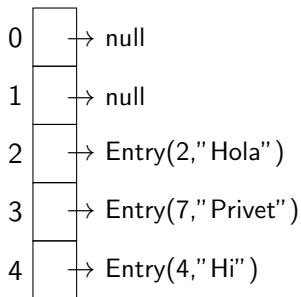
- **Idea:** localizar el primer “bucket” libre tras el bucket deseado e insertar la entrada ahí con una *búsqueda circular*.
- No podemos insertar Entry(7, "Privet") en el bucket 2, así que la insertamos en el siguiente bucket libre, el 3.

## Insertar: manejando colisiones – “open addressing”



- **Idea:** localizar el primer “bucket” libre tras el bucket deseado e insertar la entrada ahí con una *búsqueda circular*.
- No podemos insertar Entry(7, "Privet") en el bucket 2, así que la insertamos en el siguiente bucket libre, el 3.

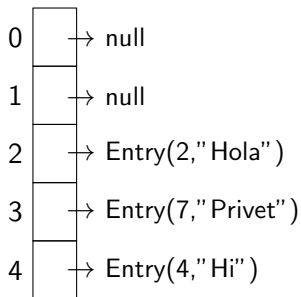
## Insertar: manejando colisiones – “open addressing”



- Llamada a put(9, "Hej").

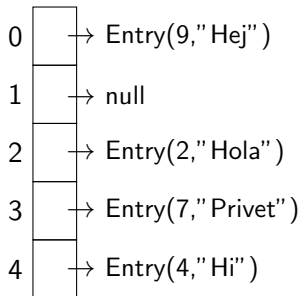


## Insertar: manejando colisiones – “open addressing”



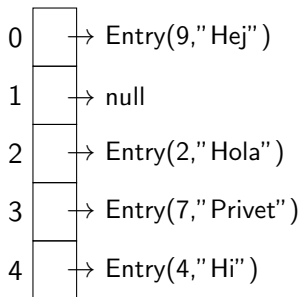
- Llamada a put(9, "Hej").
- Su bucket preferido es 4, pero está ocupado. Buscamos el siguiente bucket libre, que es el 0 (recordad: búsqueda “circular”).

## Insertar: manejando colisiones – “open addressing”



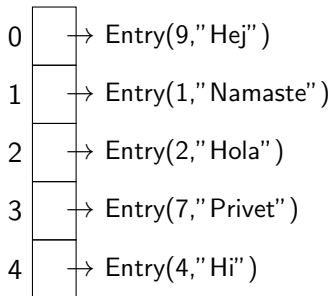
- Llamada a `put(9, "Hej")`.
- Su bucket preferido es 4, pero está ocupado. Buscamos el siguiente bucket libre, que es el 0 (recordad: búsqueda “circular”).
- Insertamos `Entry(9," Hej")` en bucket 0

## Insertar: manejando colisiones – “open addressing”



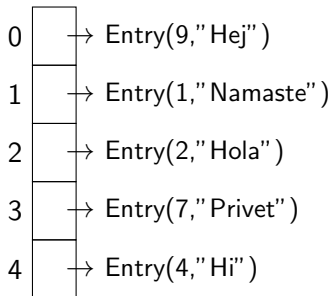
- Llamada a `put(1, "Namaste")` – su bucket preferido es 1, y no está ocupado. Lo insertamos.

## Insertar: manejando colisiones – “open addressing”



- Llamada a put(1, "Namaste") – su bucket preferido es 1, y no está ocupado. Lo insertamos.

## Insertar: manejando colisiones – “open addressing”



- Llamada a `put(12, "Salut")` – su bucket preferido es 2, y está ocupado — **todos los buckets están ocupados**. ¿Qué hacemos?

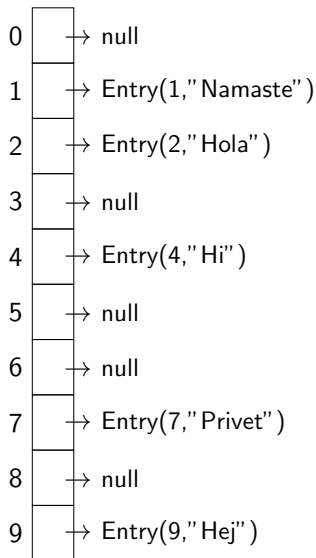
# Insertar: todos buckets ocupados

- **Idea:** hacemos un “rehashing”:

- ▶ Crear otro array duplicando el tamaño del anterior.
- ▶ Insertar en él los componentes del anterior usando el mismo procedimiento que antes.
- ▶ Los índices de cada `Entry` pueden cambiar: el array tiene un tamaño distinto, y la función `index` debe tener en cuenta el tamaño de la tabla:

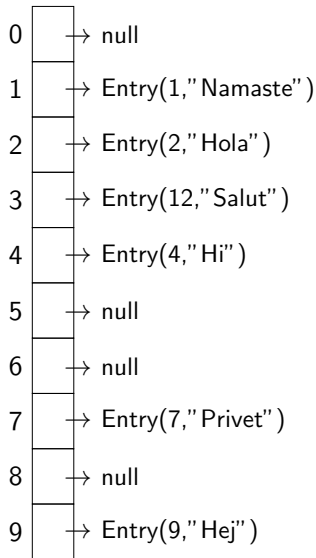
$$\text{index}(\text{key}) \equiv \text{abs}(\text{key.hashCode()}) \bmod \text{size}(\text{table})$$

## Rehashing: resultado



Tras del "rehashing" insertamos (12,"Salut"):

## Rehashing: resultado





# Búsqueda de una clave en la tabla

- Debe tener en cuenta que podría estar en lugar diferente de su bucket “preferido”: podría haber habido colisiones en la inserción.
- Algoritmo:
  - ▶ Calcular *index*, el bucket preferido de la clave.
  - ▶ Asignar un variable  $i = index$ .
  - ▶ Repetir:
    - ★ Si `buckets[i]` está vacío, la clave no esté en la tabla.
    - ★ Si la clave de `buckets[i]` es la que buscamos, hemos encontrado la Entry que necesitamos. Devolver su valor.
    - ★ Si no, ir al siguiente  $i$  (circularmente).  
Si  $i = index$  termina la búsqueda sin éxito.

# Borrando en la tabla

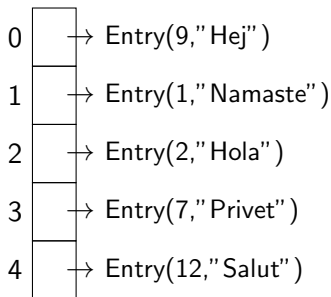
- Borrar una clave necesita encontrar primero la clave en la tabla.
  - ▶ Si la encontramos, podemos vaciar el bucket donde estaba.
- **Problema:** se pueden crear huecos indebidos, porque la búsqueda de una clave solo continúa si no hay huecos.
- Tras vaciar un bucket necesitamos “compactar” la tabla resultante moviendo entradas para evitar huecos indebidos.

## Borrando en la tabla: simulacro

0	→	Entry(9," Hej" )
1	→	Entry(1," Namaste" )
2	→	Entry(2," Hola" )
3	→	Entry(7," Privet" )
4	→	Entry(12," Salut" )

- Queremos borrar la clave 7

## Borrando en la tabla: simulacro



- Queremos borrar la clave 7
- Empezamos a buscar en el bucket 2, porque como  $7.\text{hashCode()} \bmod 5 \equiv 2$ .

## Borrando en la tabla: simulacro

0	→	Entry(9,"Hej")
1	→	Entry(1,"Namaste")
2	→	Entry(2,"Hola")
3	→	Entry(7,"Privet")
4	→	Entry(12,"Salut")

- Queremos borrar la clave 7
- Empezamos a buscar en el bucket 2, porque como  $7.hashCode() \bmod 5 \equiv 2$ .
- No esta ahí, pero como *no está vacío* miramos en el siguiente (el bucket 3).

## Borrando en la tabla: simulacro

0	→	Entry(9,"Hej")
1	→	Entry(1,"Namaste")
2	→	Entry(2,"Hola")
3	→	Entry(7,"Privet")
4	→	Entry(12,"Salut")

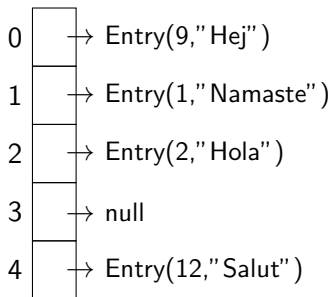
- Queremos borrar la clave 7
- Empezamos a buscar en el bucket 2, porque como  $7.hashCode() \bmod 5 \equiv 2$ .
- No esta ahí, pero como *no está vacío* miramos en el siguiente (el bucket 3).
- Ahi lo encontramos (la entrada `Entry(7,"Privet")`).

## Borrando en la tabla: simulacro

0	→	Entry(9,"Hej")
1	→	Entry(1,"Namaste")
2	→	Entry(2,"Hola")
3	→	Entry(7,"Privet")
4	→	Entry(12,"Salut")

- Queremos borrar la clave 7
- Empezamos a buscar en el bucket 2, porque como  $7.hashCode() \bmod 5 \equiv 2$ .
- No esta ahí, pero como *no está vacío* miramos en el siguiente (el bucket 3).
- Ahi lo encontramos (la entrada `Entry(7,"Privet")`).
- Vaciamos el bucket

## Borrando en la tabla: simulacro



- Queremos borrar la clave 7
- Empezamos a buscar en el bucket 2, porque como  $7.hashCode() \bmod 5 \equiv 2$ .
- No está ahí, pero como *no está vacío* miramos en el siguiente (el bucket 3).
- Ahí lo encontramos (la entrada `Entry(7, "Privet")`).
- Vaciamos el bucket



## Borrando en la tabla: simulacro

0	→ Entry(9,"Hej")
1	→ Entry(1,"Namaste")
2	→ Entry(2,"Hola")
3	→ null
4	→ Entry(12,"Salut")

- ¿Si ahora buscamos la clave 12 que pasa?

## Borrando en la tabla: simulacro

0	→ Entry(9,"Hej")
1	→ Entry(1,"Namaste")
2	→ Entry(2,"Hola")
3	→ null
4	→ Entry(12,"Salut")

- ¿Si ahora buscamos la clave 12 que pasa?
- Buscamos, empezando con el bucket 2, porque  $12 \bmod 5 \equiv 2$ .

## Borrando en la tabla: simulacro

0	→ Entry(9,"Hej")
1	→ Entry(1,"Namaste")
2	→ Entry(2,"Hola")
3	→ null
4	→ Entry(12,"Salut")

- ¿Si ahora buscamos la clave 12 que pasa?
- Buscamos, empezando con el bucket 2, porque  $12 \bmod 5 \equiv 2$ .
- La clave 12 no esta en el bucket 2; continuamos en el bucket 3...

## Borrando en la tabla: simulacro

0	→ Entry(9,"Hej")
1	→ Entry(1,"Namaste")
2	→ Entry(2,"Hola")
3	→ null
4	→ Entry(12,"Salut")

- ¿Si ahora buscamos la clave 12 que pasa?
- Buscamos, empezando con el bucket 2, porque  $12 \bmod 5 \equiv 2$ .
- La clave 12 no esta en el bucket 2; continuamos en el bucket 3...
- El bucket 3 está vacío  $\implies$  terminamos la búsqueda.

## Borrando en la tabla: simulacro

0	→ Entry(9,"Hej")
1	→ Entry(1,"Namaste")
2	→ Entry(2,"Hola")
3	→ null
4	→ Entry(12,"Salut")

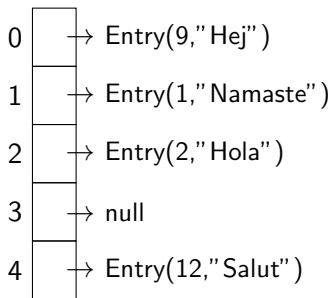
- ¿Si ahora buscamos la clave 12 que pasa?
- Buscamos, empezando con el bucket 2, porque  $12 \bmod 5 \equiv 2$ .
- La clave 12 no esta en el bucket 2; continuamos en el bucket 3...
- El bucket 3 está vacío  $\implies$  terminamos la búsqueda.
- ¡Pero esta en el bucket 4! Algo está mal...

## Borrando en la tabla: simulacro

0	→ Entry(9,"Hej")
1	→ Entry(1,"Namaste")
2	→ Entry(2,"Hola")
3	→ null
4	→ Entry(12,"Salut")

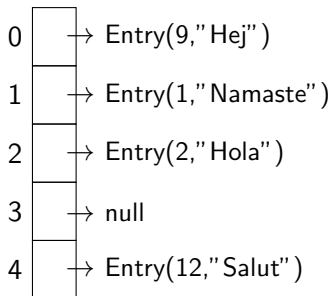
- ¿Si ahora buscamos la clave 12 que pasa?
- Buscamos, empezando con el bucket 2, porque  $12 \bmod 5 \equiv 2$ .
- La clave 12 no esta en el bucket 2; continuamos en el bucket 3...
- El bucket 3 está vacío  $\implies$  terminamos la búsqueda.
- ¡Pero esta en el bucket 4! Algo está mal...
- Solución: tras vaciar un bucket (y crear un hueco), deberíamos comprobar si hay que mover otra entrada a ese hueco.

## Colapsando huecos despues de haber borrado: simulacro



- Examinamos los siguientes buckets para ver si podemos mover sus entradas al hueco en el bucket 3.

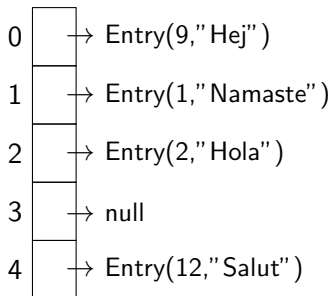
## Colapsando huecos despues de haber borrado: simulacro



- Examinamos los siguientes buckets para ver si podemos mover sus entradas al hueco en el bucket 3.
- En el bucket 4 está `Entry(12,"Salut")`. ¿Deberíamos moverla al bucket 3?

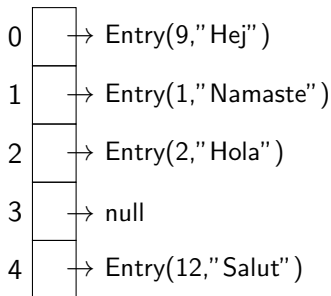


## Colapsando huecos despues de haber borrado: simulacro



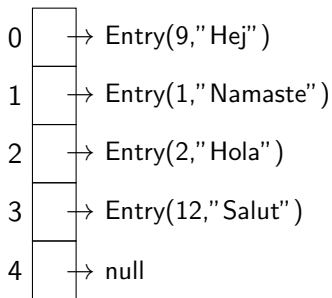
- Examinamos los siguientes buckets para ver si podemos mover sus entradas al hueco en el bucket 3.
- En el bucket 4 está Entry(12,"Salut"). ¿Deberíamos moverla al bucket 3?
- **¡Sí!** El hueco en el bucket 3 está más *cerca* de su posición preferida (bucket 2) que su lugar actual (el bucket 4).

## Colapsando huecos despues de haber borrado: simulacro



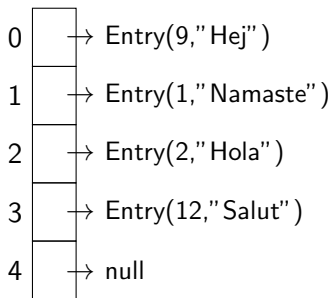
- Examinamos los siguientes buckets para ver si podemos mover sus entradas al hueco en el bucket 3.
- En el bucket 4 está `Entry(12, "Salut")`. ¿Deberíamos moverla al bucket 3?
- **¡Sí!** El hueco en el bucket 3 está más *cerca* de su posición preferida (bucket 2) que su lugar actual (el bucket 4).
- Movemos la entrada en bucket 4 a bucket 3.

## Colapsando huecos despues de haber borrado: simulacro



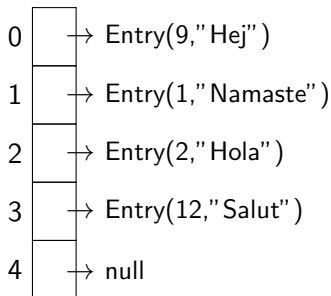
- Se crea un hueco en el bucket 4: deberíamos intentar eliminarlo con el mismo procedimiento, empezando con el bucket 0.

## Colapsando huecos despues de haber borrado: simulacro



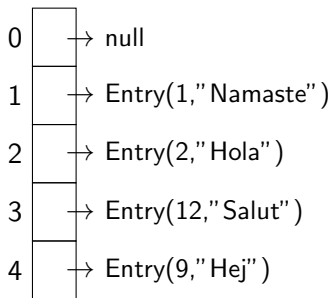
- Se crea un hueco en el bucket 4: deberíamos intentar eliminarlo con el mismo procedimiento, empezando con el bucket 0.
- El bucket 0 contiene Entry(9,"Hej"). ¿Podemos moverla al 4?

## Colapsando huecos despues de haber borrado: simulacro



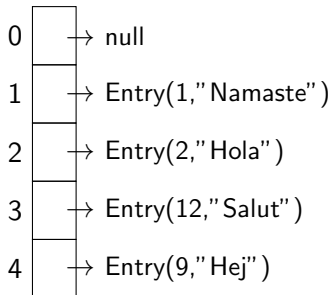
- Se crea un hueco en el bucket 4: deberíamos intentar eliminarlo con el mismo procedimiento, empezando con el bucket 0.
- El bucket 0 contiene Entry(9, "Hej"). ¿Podemos moverla al 4?
- **¡Si!** Su posición "preferida" es el bucket 4. La movemos.

## Colapsando huecos despues de haber borrado: simulacro



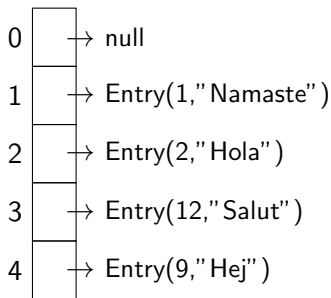
- Tenemos un hueco en bucket 0. ¿Podemos mover Entry(1, "Namaste") ahí?

## Colapsando huecos despues de haber borrado: simulacro



- Tenemos un hueco en bucket 0. ¿Podemos mover Entry(1, "Namaste") ahí?
- No, el índice preferido de la clave 1 es el bucket 1, y ya está ahí. No deberíamos moverlo.

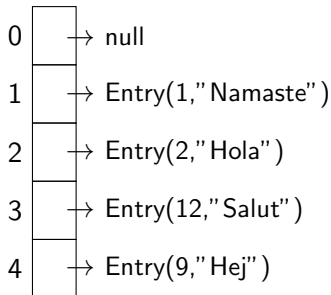
## Colapsando huecos despues de haber borrado: simulacro



- Tenemos un hueco en bucket 0. ¿Podemos mover Entry(2, "Hola") ahí?

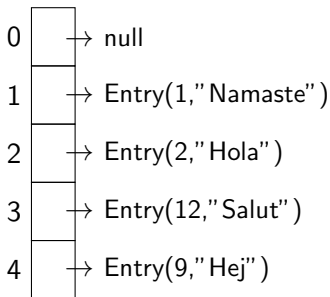


## Colapsando huecos despues de haber borrado: simulacro



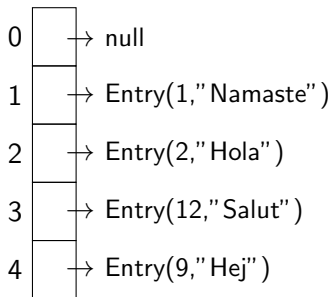
- Tenemos un hueco en bucket 0. ¿Podemos mover Entry(2,"Hola") ahí?
- No, el índice preferido de la clave 2 es el bucket 2 y ya esta ahí. No deberíamos moverlo.

## Colapsando huecos despues de haber borrado: simulacro



- Empezabamos el procedimiento de colapsar en el índice 3 y hemos vuelto. Terminado.

## Colapsando huecos despues de haber borrado: simulacro



- Empezabamos el procedimiento de colapsar en el índice 3 y hemos vuelto. Terminado.
- El procedimiento de colapsar huecos termina cuando encontramos un bucket vacío *no creado* por el procedimiento mismo o cuando llegamos al principio (al bucket originalmente borrado).

# Colapsando huecos: Algoritmo

- Asumimos que el nuevo hueco está en el índice  $index_{hueco}$
- Asignamos  $start = index_{hueco}$  y  $i = (index_{hueco} + 1) \bmod buckets.length$
- Repetir mientras  $i \neq start$  and  $buckets[i] \neq null$ 
  - ▶ Calcular el índice preferido –  $index_{preferido}$  – de la clave que está en  $buckets[i]$
  - ▶ Si  $index_{hueco}$  está más “cerca” de  $index_{preferido}$  que  $i$  (su posición actual):
    - ★  $buckets[index_{hueco}] = buckets[i]$
    - ★  $buckets[i] = null$  y  $index_{hueco} = i$
  - ▶ Incrementar  $i = (i + 1) \bmod buckets.length$

La condición de que  $index_{hueco}$  esta más cerca  $index_{preferido}$  que  $i$  se cumple cuando:

- si  $i \geq index_{preferido}$  entonces  $index_{preferido} \leq index_{hueco} < i$
- si  $index_{preferido} > i$  entonces  $index_{hueco} \geq index_{preferido}$  o  $index_{hueco} < i$

Os recomendamos dibujar escenarios para entenderlos mejor

# Métodos auxiliares útiles **no obligatorios**

Podría resultar útil definir algunos métodos auxiliares:

- **int** `index(Object key)` – calcula el índice del array donde se busca la clave `key` primero. Debería calcular el `hashCode` de `key`, y después “comprimir” el resultado en el rango `0..buckets.length - 1`.
- **int** `search(K key)` – busca la clave en el array, según el procedimiento de búsqueda explicado anteriormente. Devuelve:
  - ▶ El índice donde reside una entrada con la clave `key`, si está en la tabla.
  - ▶ Si no hay entrada con la clave `key` en `buckets`, devuelve, *si hay buckets libres*, el índice del siguiente “bucket” libre (con elemento **null**) dentro `buckets`
  - ▶ Si no hay entrada con clave `key`, ni existe un bucket libre, devuelve -1.
- **void** `rehash()` – implementa la operación de “rehashing” según la explicación anterior.
- `Entry<K,V>[] createBuckets(int size)` – crea un array nuevo de entradas de tamaño `size`. **Este método ya está programado en HashTable.java.**

# Puntuación

- Bien implementado todo excepto `remove`: 9 puntos máximo
- Bien implementado todo incluyendo `remove`: 12 puntos máximo

- Notad que `HashTable.java` no compila; falta implementar los métodos necesarios para que la clase implementa la interfaz `Map`.
- Es **obligatorio** usar el atributo `buckets` para guardar los elementos de la tabla Hash. No está permitido cambiar su tipo.
- El proyecto debe compilar sin errores, debe cumplir la especificación de los métodos a completar, y debe ejecutar `TesterLab3` correctamente sin mensajes de error
- Nota: una ejecución sin mensajes de error no significa que el método sea correcto (es decir, que funcione bien para cada posible entrada)
- Todos los ejercicios se comprueban manualmente