Descripción de la asignatura

- Tema 1: Introducción
- Tema 2: Árboles generales
- **Tema 3: Mapas y diccionarios**
- Tema 4: Mapas y diccionarios ordenados
- Tema 5: Grafos
- Tema 6: EEDD en Memoria Secundaria

Bloque 1

Bloque 2

Bloque 3

Tema II.3
Mapas y Diccionarios

jose.velez@urjc.es
angel.sanchez@urjc.es
mariateresa.gonzalezdelena@urjc.es

abraham.duarte@urjc.es
raul.cabido@urjc.es

Resumen

- Mapas
- Tablas *Hash*
  - Posibles implementaciones
  - Funciones *Hash*
  - Códigos *Hash*
  - Colisiones
- Diccionarios

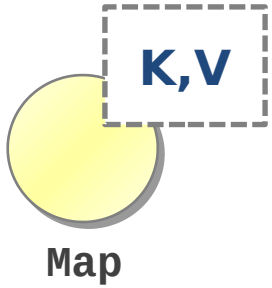
Mapas

Mapas

- Un **mapa** almacena información que puede ser localizada eficientemente usando claves
- Almacena colección de pares clave-valor, denominadas **entradas**
  - Claves representan una forma de acceso (índice) a un valor asociado
- Operaciones principales
  - Insertar, buscar, borrar
- No están permitidas entradas con la **misma clave**
- Aplicaciones prácticas
  - Cuando se necesita indexar valores por un índice que no sea un número entero (cadenas, coordenadas, matrículas...)
  - Cuando se necesita indexar valores con enteros no consecutivos.

Mapas

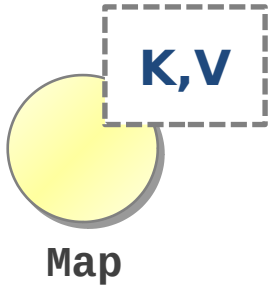
- **Métodos** del TAD Mapa (equivalente a `java.util.map`)



  - `size()`, `isEmpty()`
  - `get(k)`: si el mapa M tiene una entrada con clave k, devuelve el valor asociado; en caso contrario, null
  - `put(k, v)`: inserta la entrada (k,v) en el mapa. Si la clave no estaba en el mapa, devuelve `null`; en caso contrario devuelve el antiguo valor asociado a k
  - `remove(k)`: si el mapa tiene una entrada con clave `k`, la elimina y devuelve el valor correspondiente; en caso contrario devuelve `null`
  - `keySet()`, `entrySet()`, `values()`: `iterable` de claves, entradas o valores

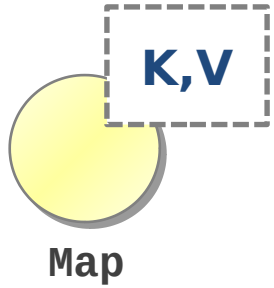
Mapas

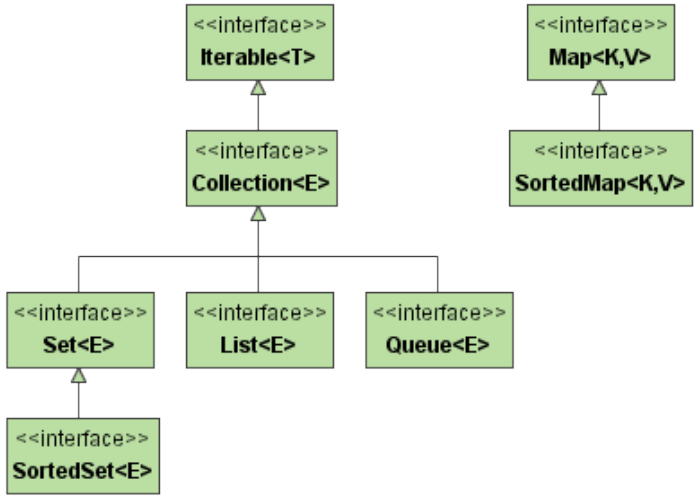
```
public interface Map<K, V> {

    public int size();

    public boolean isEmpty();

    public V put(K key, V value);

    public V get(K key);

    public V remove(K key);

    public Set<K> keySet();

    public Collection<V> values();

    public Set<Map.Entry<K, V>> entrySet();
}
```



Mapas

```
public interface Map<K, V> {

    public int size();

    public boolean isEmpty();

    public V put(K key, V value);

    public V get(K key);

    public V remove(K key);

    public Set<K> keySet();

    public Collection<V> values();

    public Set<Map.Entry<K, V>> entrySet();
}
```



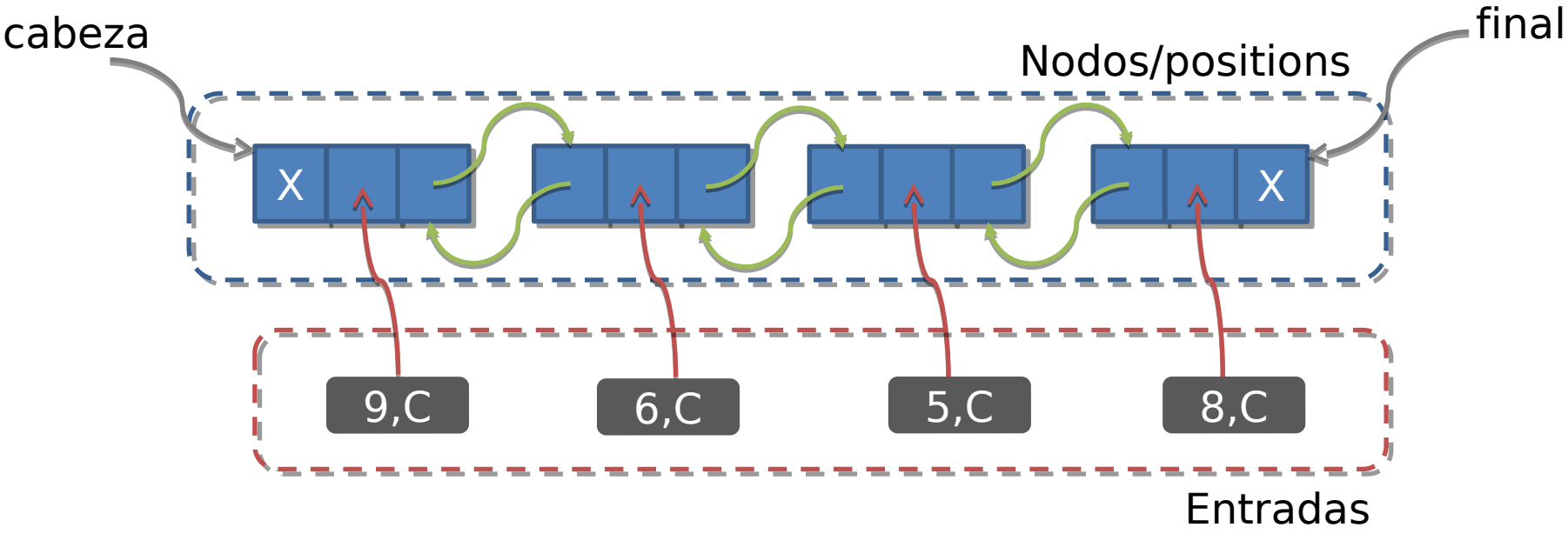


Ejemplo de uso

Operación	Salida	Mapa
isEmpty()	true	Ø
put(5,A)	null	(5,A)
put(7,B)	null	(5,A), (7,B)
put(2,C)	null	(5,A), (7,B), (2,C)
put(8,D)	null	(5,A), (7,B), (2,C), (8,D)
put(2,E)	C	(5,A), (7,B), (2,E), (8,D)
get(7)	B	(5,A), (7,B), (2,E), (8,D)
get(4)	null	(5,A), (7,B), (2,E), (8,D)
get(2)	E	(5,A), (7,B), (2,E), (8,D)
size()	4	(5,A), (7,B), (2,E), (8,D)
remove(5)	A	(7,B), (2,E), (8,D)
remove(2)	E	(7,B), (8,D)
get(2)	null	(7,B), (8,D)
isEmpty()	false	(7,B), (8,D)

Mapas

- **Implementaciones** de mapas
  - Estructuras lineales (lista, vectores, etc.)



  - Tablas *Hash*

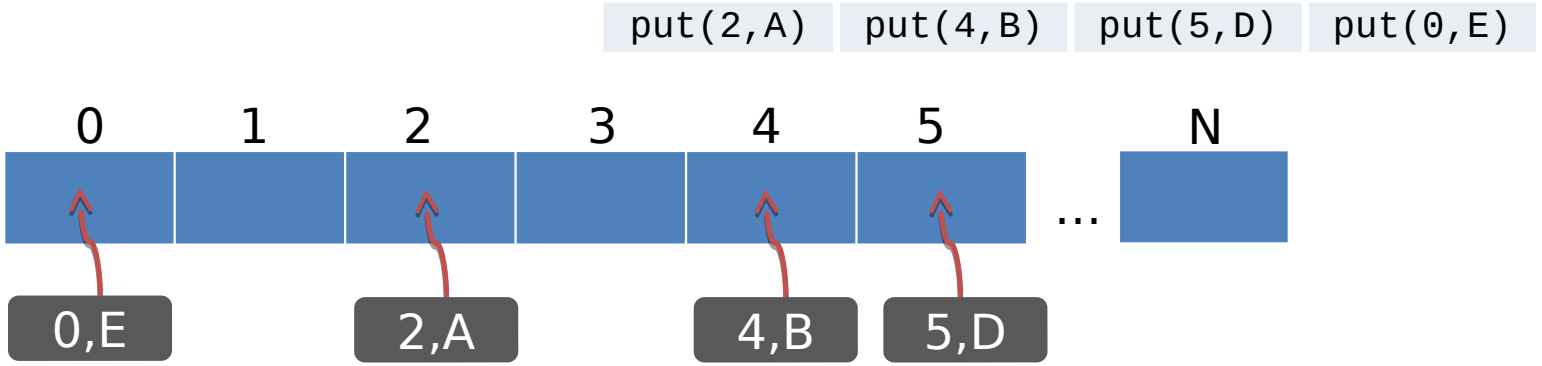
Tablas Hash

Tablas *Hash*

- Las **tablas *hash*** constituyen una forma eficiente de implementar mapas:
  - Utilizan la clave *K* como dirección para acceder al valor *V*
  - Las operaciones de inserción, búsqueda y borrado son *O(1)* en promedio
- Una **tabla *hash*** se compone fundamentalmente de:
  - un ***Array*** (llamado tabla o buket) de tamaño *N*
  - una **función *hash*** → *h(k)*
- Implementar un mapa con una tabla *hash* consiste en almacenar la entrada *(k, v)* en la posición *i = h(k)* del *array*

función Hash…¿por qué?

- Si las claves son enteros en el rango la tabla es todo lo que necesitamos



- Problemas:
  - ¿Qué ocurre si N es mucho más grande que el número de entradas que tenemos en el mapa?
  - ¿Qué ocurre si las claves no son enteros en el rango ?
    - Ejemplos: número de expediente, nombre de una variable de entorno, ….

Tablas *Hash*

- Una **función hash** *h* "mapea" las claves de un determinado tipo a enteros en un determinado intervalo *[0, .., N – 1]*

$h(k) = k \bmod N$ → ejemplo función *hash* para variables enteras

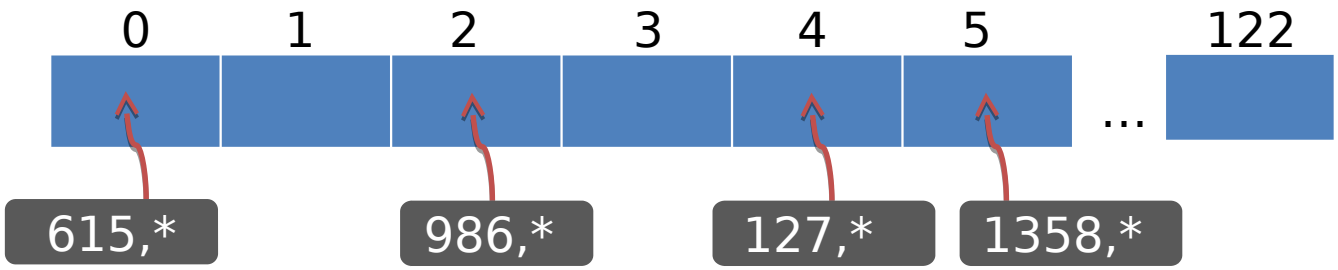
- El valor se conoce como **clave dispersada** (*hash value*)

```
put(127,new Alumno(…))
```

```
put(615, new Alumno(…))
```

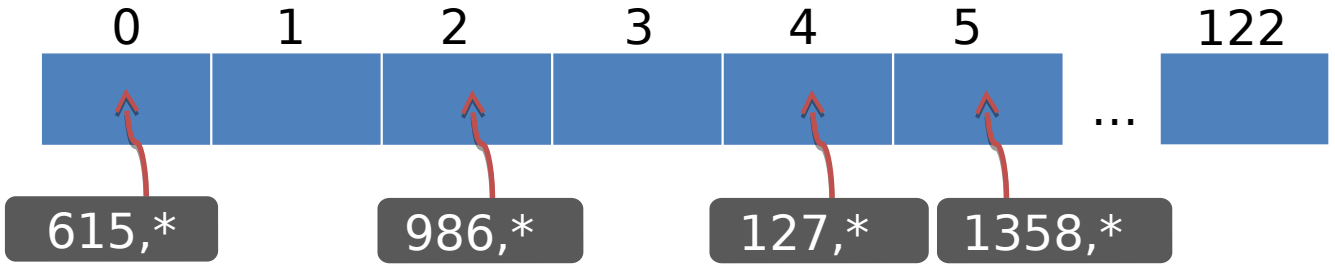
```
put(1358, new Alumno(…))
```

```
put(986, new Alumno(…))
```



Tablas *Hash*

¿Y si dos claves presentan el mismo *hash value?*



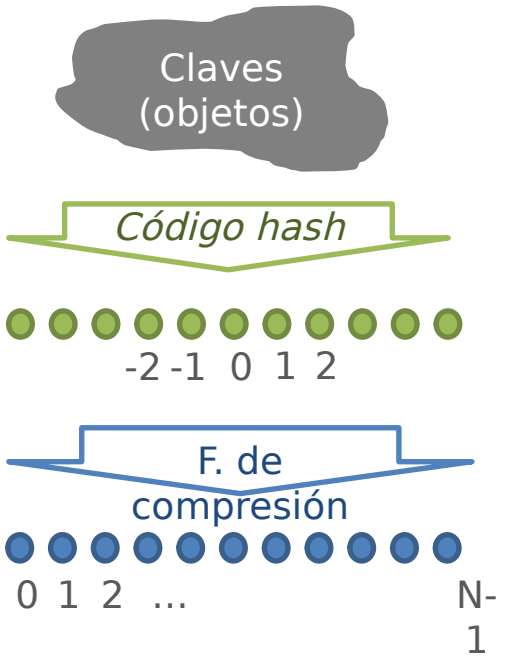

```
put(1845,new Alumno(…))
```

$h(1845) = 0$

- Colisión
  - Seleccionar una buena función *hash*
    - Minimice las colisiones
    - Fácil y rápida de computar
  - Otras estrategias (se detallan más adelante)

Tablas *Hash*

- Una **función *hash*** se suele especificar como la composición de dos funciones:
  - Código *Hash*: h_H: $Key \rightarrow int$
  - Función de compresión h_C: $int \rightarrow [0,N-1]$



- El **código *hash*** se aplica primero a la clave y después se aplica la compresión

$$h(k) = h_C(h_H(k))$$

Tablas Hash:
Códigos Hash y funciones de compresión

Tablas *Hash* (códigos *Hash*)

- Consistencia
  - Dos claves que se consideren iguales deben presentar el mismo código *hash*

- En Java la función $h_H(k)$ tiene el siguiente interface:

```
int hashCode()
```

La función mapea un objeto a un valor entero (`int`)

- Interpretación entera de la dirección de memoria
- Mal funcionamiento para *strings*
  - `String` redefine el comportamiento heredado de `Object`
- Objetos que puedan actuar como clave
  - Redefinir comportamiento `hashcode()`
  - Generar valores enteros dispersos y **consistentes**

Tablas *Hash* (códigos *Hash*)

- **Conversión (*cast*) a entero**
  - Interpretación de los bits de la clave como un entero
  - Válido para tipos `int` → `byte`, `short`, `int`, `char`, `float`

```
// si k es float
int hashCode () {return Float.floatToIntBits (k);}
```

  - Y para tipos `int`? → `double, long`

- **Suma de componentes:**
  - Interpretar bits de orden superior e inferior como dos enteros
  - Sumarlos e ignorar *overflows*

```
// si k es long
int hashCode () {return (int) (k>>32) + (int) k;}
```

  - Extensible a cualquier objeto
    - n-tupla (k_0, k_1, k_2,..., k_{n-1}) de enteros

$$\sum_{i=0}^{n-1} k_i$$

- **Acumulación polinómica**
  - Dividir los bits de la clave en componentes de la misma longitud

$$(k_0, k_1, k_2, \ldots, k_n)$$

  - Seleccionar una constante x *distinta de cero y de 1*

$$k_0 + k_1 x + k_2 x^2 + \ldots + k_n x^n$$

$$\Updownarrow$$

$$k_0 + x\left(k_1 + x^2\left(k_2 + \ldots + x^{n-1}\left(k_{n-1} + k_n x\right)\right)\right)$$

  - Adecuado para claves `string`

> *x = 33* produce como mucho 6 colisiones para 50.000 palabras!!!!

  - El polinomio p se puede evaluar en *O(n)*
    - Regla de Horner
  - Cálculo recursivo de la clave dispersada

$$b_n = k_n$$

$$b_i = k_i + x\, b_{i-1}$$

  - Cada polinomio se calcula en *O(1)*
  - La clave dispersada es finalmente

$$p(x) = b_0$$

División

$$h_C(y) = y \bmod N$$

- El tamaño de la tabla N suele ser un número primo.
- La teoría de números apunta que el uso de números primos evita colisiones. Por ejemplo, cada clave de {200,205,210...600} colisiona con otras 3 si N es 100, pero ninguna si N es 101.

Multiplicar, sumar y dividir (MAD en inglés)

$$h_C(y) = ((ay+b) \bmod p) \bmod N$$

- p es un primo mayor que n
- a y b son enteros no negativos aleatorios en rango $[0,p-1]$

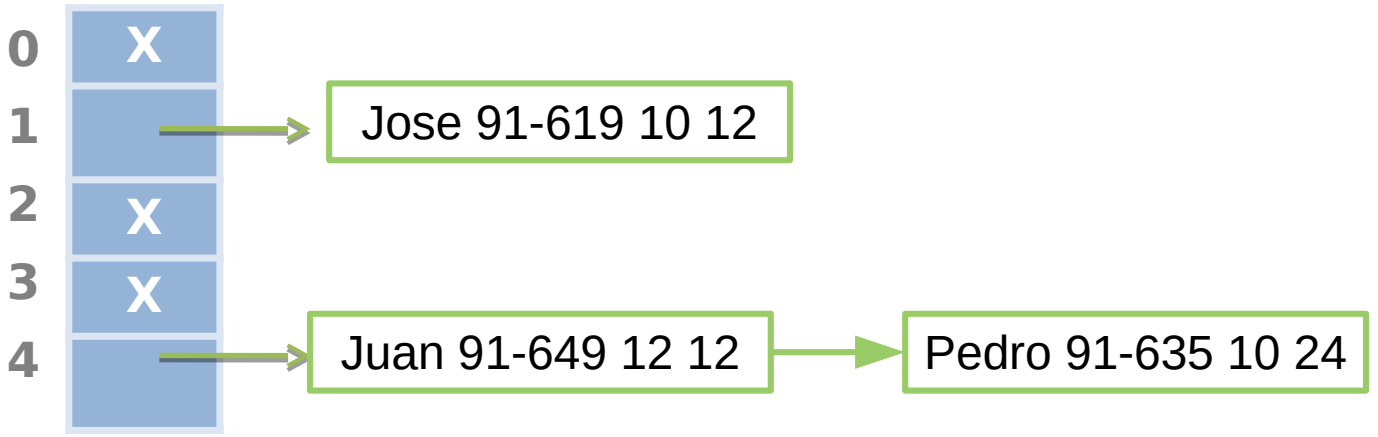
Tablas *Hash* (colisiones)

- Dos claves pueden coincidir en la misma posición. A este fenómeno se le denomina **colisión**
- Existen técnicas efectivas para resolver los conflictos que generan las colisiones
- Las más populares son:
  - **Encadenamiento separado**
  - **Direccionamiento abierto**

Resolución de colisiones: Encadenamiento separado

Encadenamiento separado

- Cada celda de la tabla tiene una referencia a todas las entradas con la misma clave dispersada.
- La secuencia de elementos con la misma clave dispersada suele ser una lista enlazada. Así, la secuencia es un mapa en miniatura con elemento (k,v) tales que $h(k)=p$, siendo p la prueba de inserción.



Encadenamiento separado

Algoritmo sobre mapas

- ***B*** es la **secuencia de entradas** con la misma clave dispersada
- Los métodos le pasan la responsabilidad de buscar a B

```
method put(k,v)
   if A[h[k]] = null
      List B = {}
      A[h[k]] = B
   else
      List B = A[h[k]]
      B.remove(k)
   B.insert(k,v)
```

```
method get(k)
  List B = A[h(k)]
  if B = null
    return null
  else
    return B.find(k)
```

```
method remove(k)
   List B = A[h(k)]
   if B = null
      return null
   else
      return B.remove(k)
```

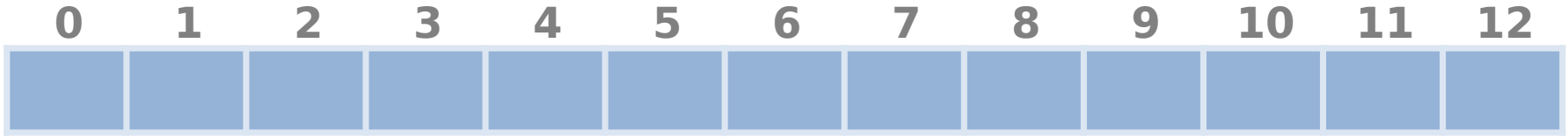
Encadenamiento separado

Utilizando la función de hash: $h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A> <41,Y> <22,C> <44,D> <18,E>

```
method put(k,v)
   if A[h(k)] = null
      List B = {}
      A[h(k)] = B
   else
      List B = A[h(k)]
      B.remove(k)
   B.insert(k,v)
```



Encadenamiento separado

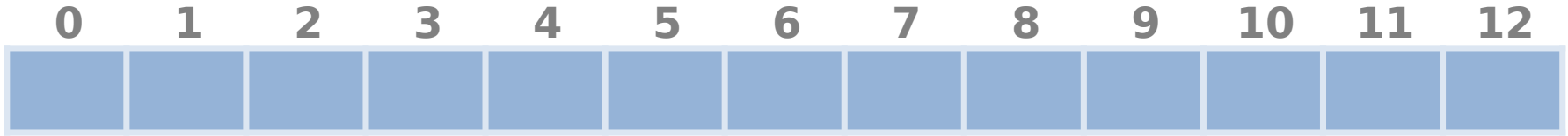
Utilizando la función de hash: $h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A> <41,Y> <22,C> <44,D> <18,E>

```
method put(k,v)
    if A[h(k)] = null
        List B = {}
        A[h(k)] = B
    else
        List B = A[h(k)]
        B.remove(k)
    B.insert(k,v)
```

k=18
v=A



Encadenamiento separado

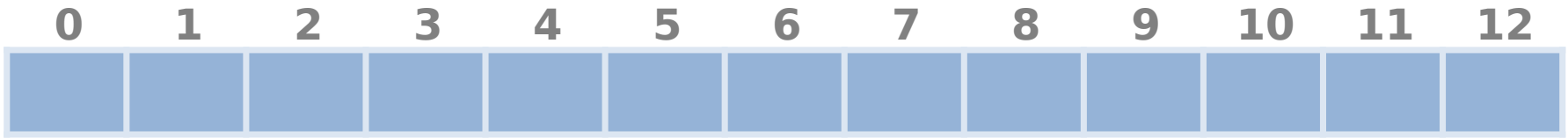
Utilizando la función de hash: $h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A> <41,Y> <22,C> <44,D> <18,E>

```
method put(k,v)
    if A[h(k)] = null
        List B = {}
        A[h(k)] = B
    else
        List B = A[h(k)]
        B.remove(k)
    B.insert(k,v)
```

k=18
v=A
h(k)=5



Encadenamiento separado

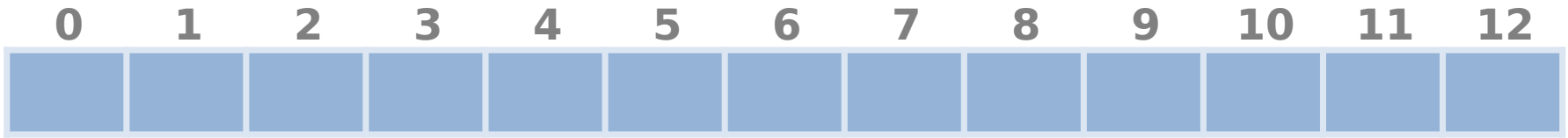
Utilizando la función de hash: $h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A> <41,Y> <22,C> <44,D> <18,E>

```
method put(k,v)
    if A[h(k)] = null
        List B = {}
        A[h(k)] = B
    else
        List B = A[h(k)]
        B.remove(k)
    B.insert(k,v)
```

k=18
v=A
h(k)=5



B={}

Encadenamiento separado

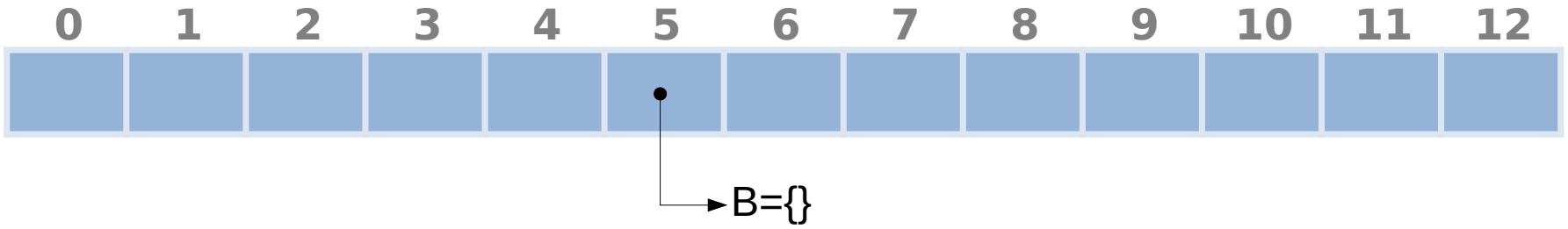
Utilizando la función de hash: $h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A> <41,Y> <22,C> <44,D> <18,E>

```
method put(k,v)
    if A[h(k)] = null
        List B = {}
        A[h(k)] = B
    else
        List B = A[h(k)]
        B.remove(k)
    B.insert(k,v)
```

k=18
v=A
h(k)=5



Encadenamiento separado

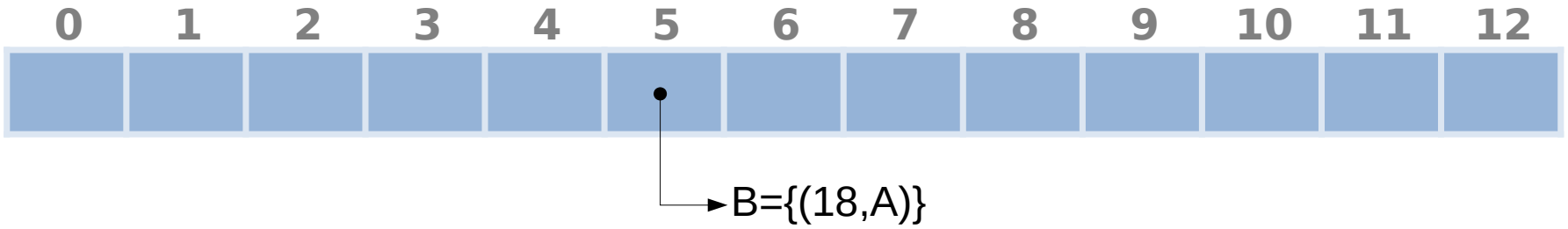
Utilizando la función de hash: $h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A> <41,Y> <22,C> <44,D> <18,E>

```
method put(k,v)
    if A[h(k)] = null
        List B = {}
        A[h(k)] = B
    else
        List B = A[h(k)]
        B.remove(k)
    B.insert(k,v)
```

k=18
v=A
h(k)=5



Encadenamiento separado

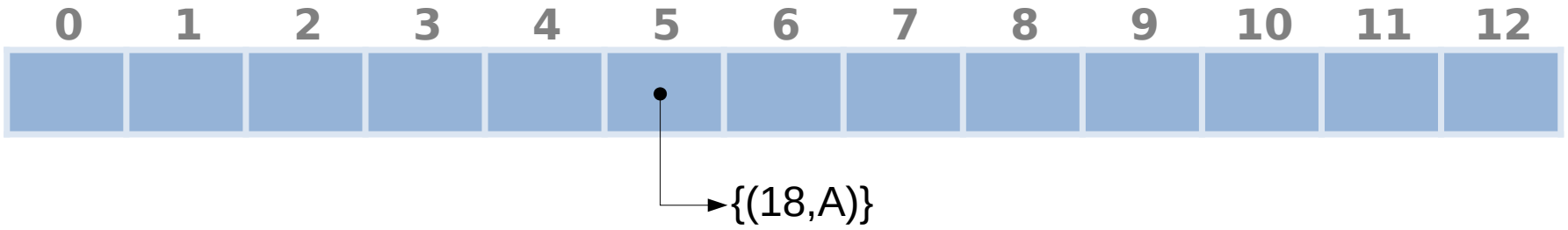
Utilizando la función de hash: $h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A> <41,Y> <22,C> <44,D> <18,E>

```
method put(k,v)
    if A[h(k)] = null
        List B = {}
        A[h(k)] = B
    else
        List B = A[h(k)]
        B.remove(k)
    B.insert(k,v)
```

k=41
v=Y



Encadenamiento separado

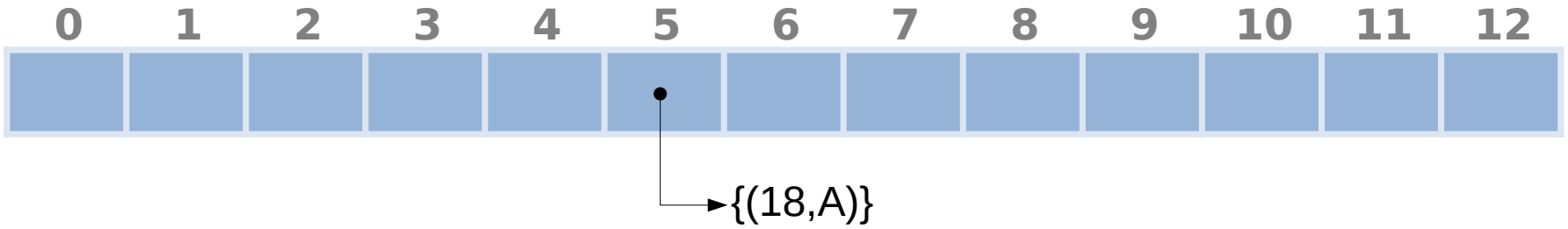
Utilizando la función de hash: $h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A> <41,Y> <22,C> <44,D> <18,E>

```
method put(k,v)
    if A[h(k)] = null
        List B = {}
        A[h(k)] = B
    else
        List B = A[h(k)]
        B.remove(k)
    B.insert(k,v)
```

k=41
v=Y
h(k)=2



Encadenamiento separado

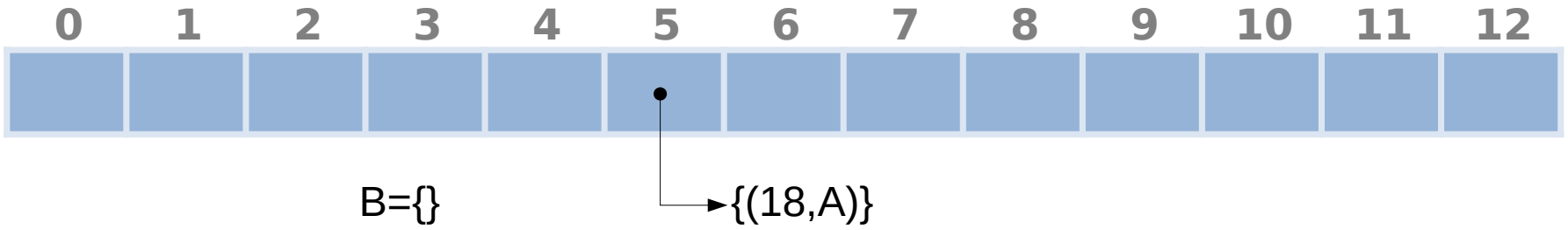
Utilizando la función de hash: $h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A> <41,Y> <22,C> <44,D> <18,E>

```
method put(k,v)
    if A[h(k)] = null
        List B = {}
        A[h(k)] = B
    else
        List B = A[h(k)]
        B.remove(k)
    B.insert(k,v)
```

k=41
v=Y
h(k)=2



Encadenamiento separado

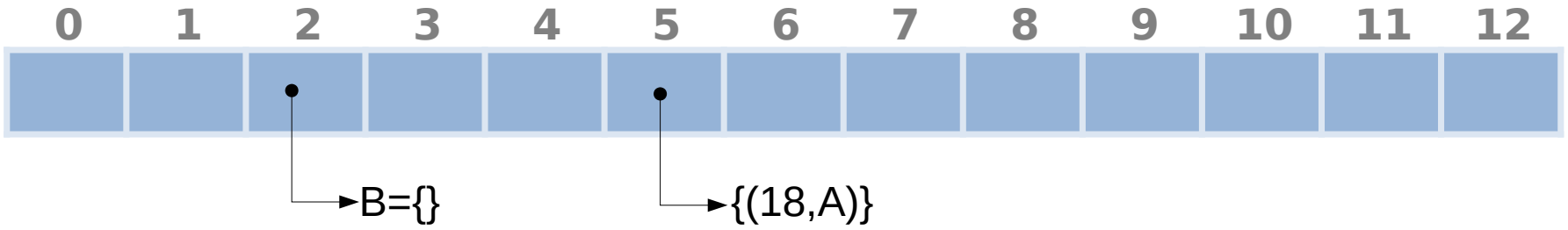
Utilizando la función de hash: $h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A> <41,Y> <22,C> <44,D> <18,E>

```
method put(k,v)
    if A[h(k)] = null
        List B = {}
        A[h(k)] = B
    else
        List B = A[h(k)]
        B.remove(k)
    B.insert(k,v)
```

k=41
v=Y
h(k)=2



Encadenamiento separado

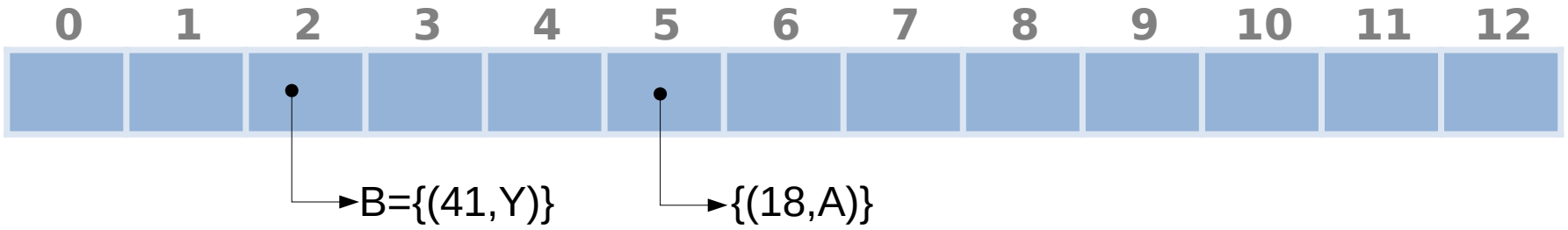
Utilizando la función de hash: $h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A> <41,Y> <22,C> <44,D> <18,E>

```
method put(k,v)
    if A[h(k)] = null
        List B = {}
        A[h(k)] = B
    else
        List B = A[h(k)]
        B.remove(k)
    B.insert(k,v)
```

k=41
v=Y
h(k)=2



Encadenamiento separado

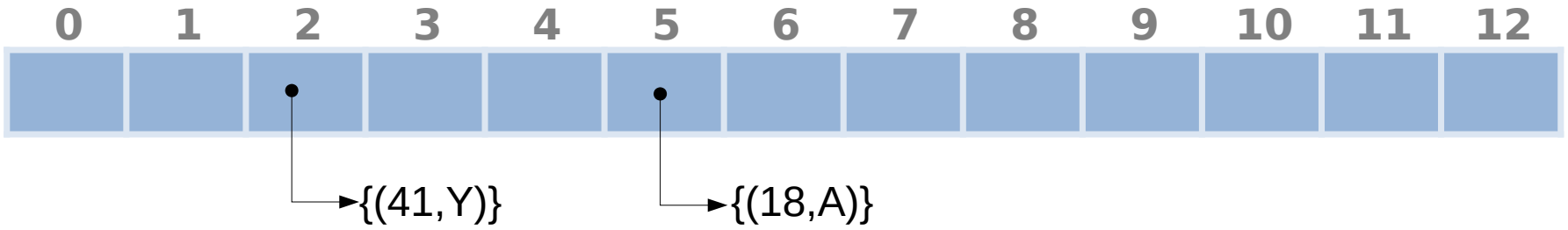
Utilizando la función de hash: $h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A> <41,Y> <22,C> <44,D> <18,E>

```
method put(k,v)
    if A[h(k)] = null
        List B = {}
        A[h(k)] = B
    else
        List B = A[h(k)]
        B.remove(k)
    B.insert(k,v)
```

k=22
v=C



Encadenamiento separado

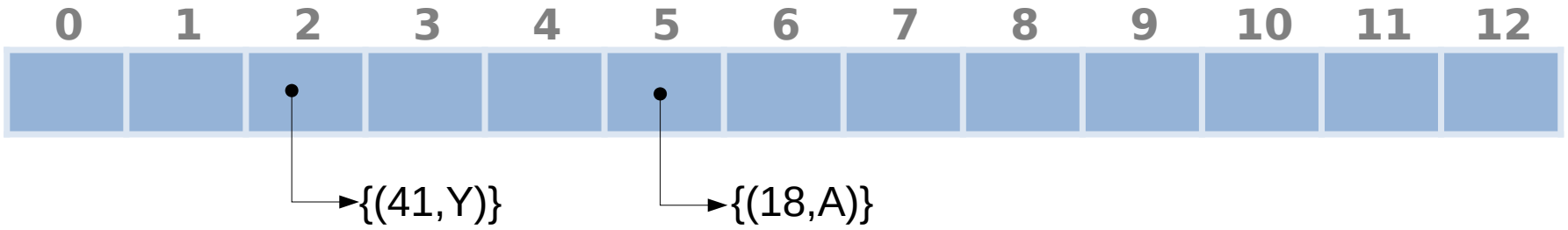
Utilizando la función de hash: $h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A> <41,Y> <22,C> <44,D> <18,E>

```
method put(k,v)
    if A[h(k)] = null
        List B = {}
        A[h(k)] = B
    else
        List B = A[h(k)]
        B.remove(k)
    B.insert(k,v)
```

k=22
v=C
h(k)=9



Encadenamiento separado

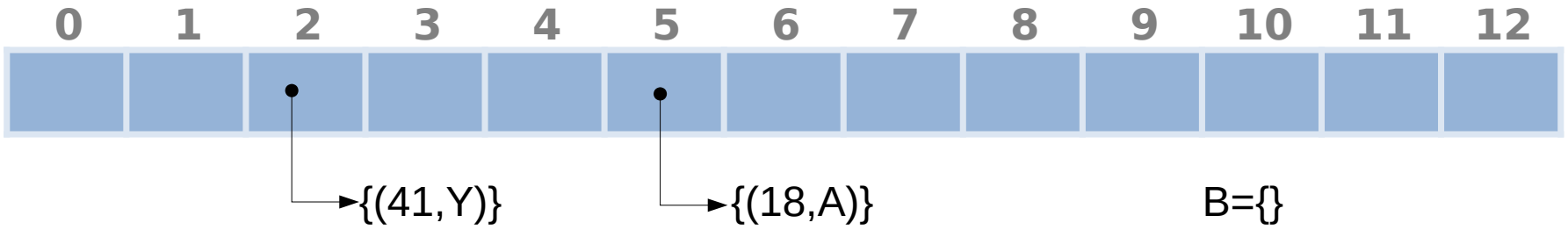
Utilizando la función de hash: $h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A> <41,Y> <22,C> <44,D> <18,E>

```
method put(k,v)
    if A[h(k)] = null
        List B = {}
        A[h(k)] = B
    else
        List B = A[h(k)]
        B.remove(k)
    B.insert(k,v)
```

k=22
v=C
h(k)=9



Encadenamiento separado

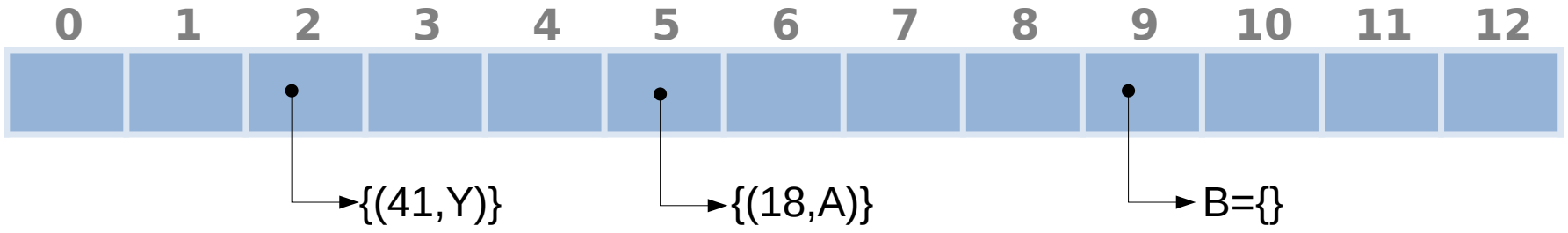
Utilizando la función de hash: $h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A> <41,Y> <22,C> <44,D> <18,E>

```
method put(k,v)
    if A[h(k)] = null
        List B = {}
        A[h(k)] = B
    else
        List B = A[h(k)]
        B.remove(k)
    B.insert(k,v)
```

k=22
v=C
h(k)=9



Encadenamiento separado

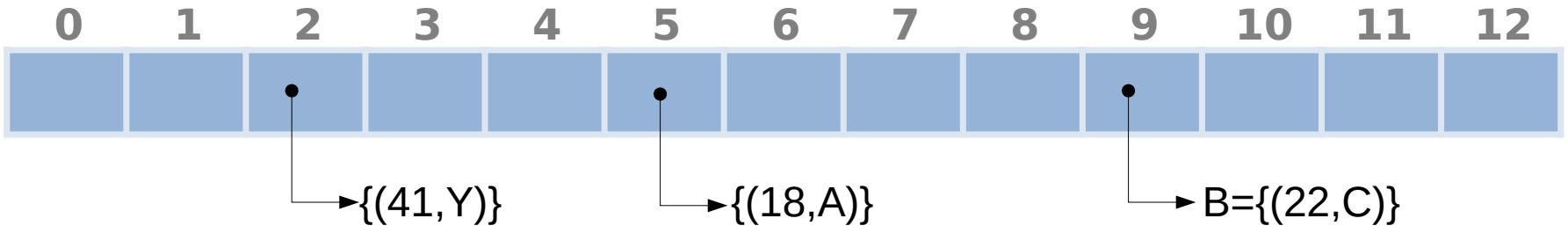
Utilizando la función de hash: $h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A> <41,Y> <22,C> <44,D> <18,E>

```
method put(k,v)
    if A[h(k)] = null
        List B = {}
        A[h(k)] = B
    else
        List B = A[h(k)]
        B.remove(k)
    B.insert(k,v)
```

k=22
v=C
h(k)=9



Encadenamiento separado

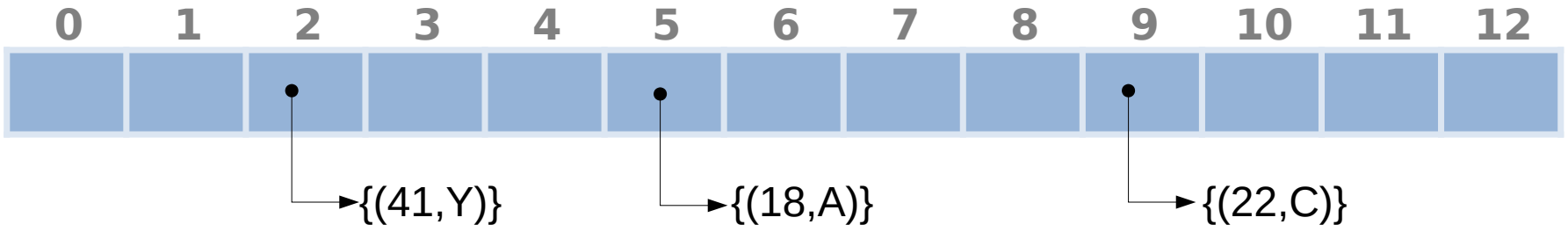
Utilizando la función de hash: $h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A> <41,Y> <22,C> <44,D> <18,E>

```
method put(k,v)
    if A[h(k)] = null
        List B = {}
        A[h(k)] = B
    else
        List B = A[h(k)]
        B.remove(k)
    B.insert(k,v)
```

k=44
v=D



Encadenamiento separado

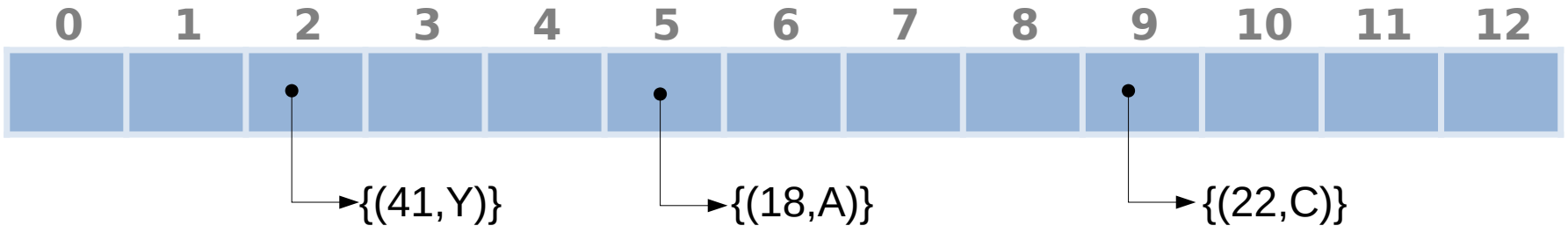
Utilizando la función de hash: $h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A> <41,Y> <22,C> <44,D> <18,E>

```
method put(k,v)
    if A[h(k)] = null
        List B = {}
        A[h(k)] = B
    else
        List B = A[h(k)]
        B.remove(k)
    B.insert(k,v)
```

k=44
v=D
h(k)=5



Encadenamiento separado

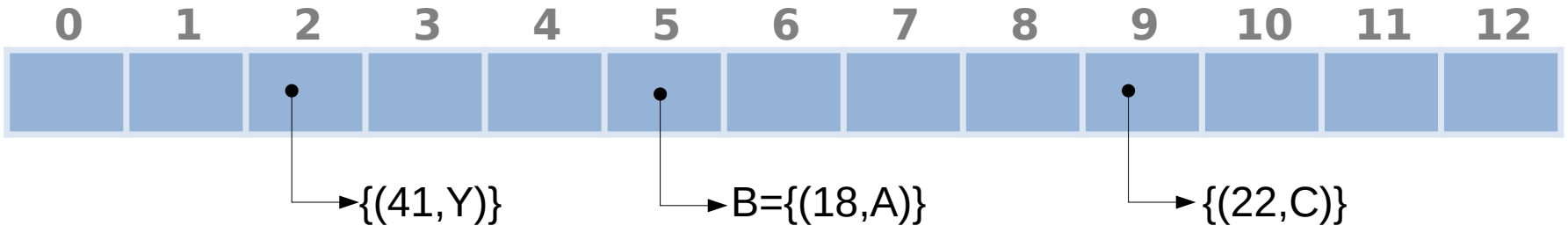
Utilizando la función de hash: $h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A> <41,Y> <22,C> <44,D> <18,E>

```
method put(k,v)
    if A[h(k)] = null
        List B = {}
        A[h(k)] = B
    else
        List B = A[h(k)]
        B.remove(k)
    B.insert(k,v)
```

k=44
v=D
h(k)=5



Encadenamiento separado

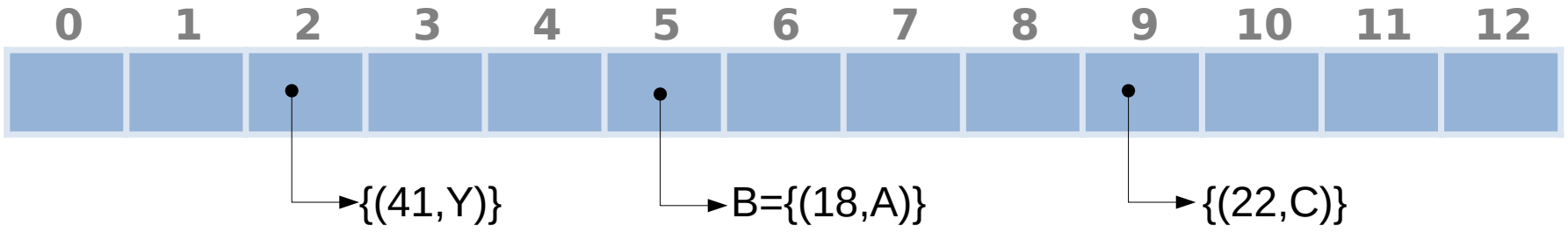
Utilizando la función de hash: $h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A> <41,Y> <22,C> <44,D> <18,E>

```
method put(k,v)
    if A[h(k)] = null
        List B = {}
        A[h(k)] = B
    else
        List B = A[h(k)]
        B.remove(k)
    B.insert(k,v)
```

k=44
v=D
h(k)=5



Encadenamiento separado

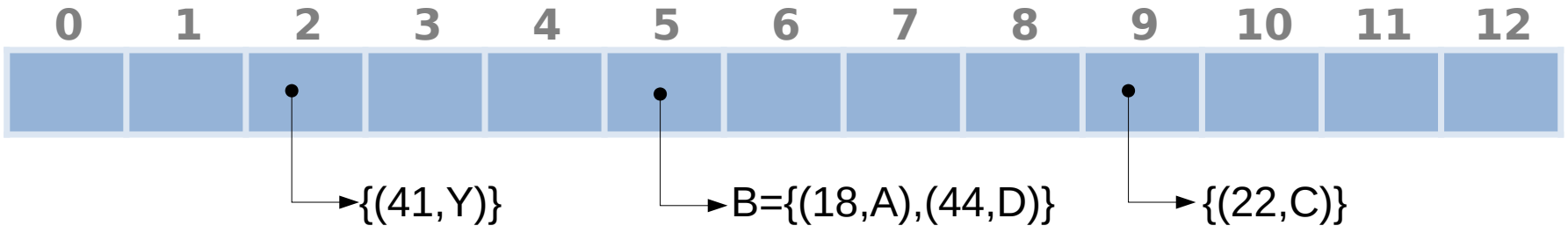
Utilizando la función de hash: $h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A> <41,Y> <22,C> <44,D> <18,E>

```
method put(k,v)
    if A[h(k)] = null
        List B = {}
        A[h(k)] = B
    else
        List B = A[h(k)]
        B.remove(k)
    B.insert(k,v)
```

k=44
v=D
h(k)=5



Encadenamiento separado

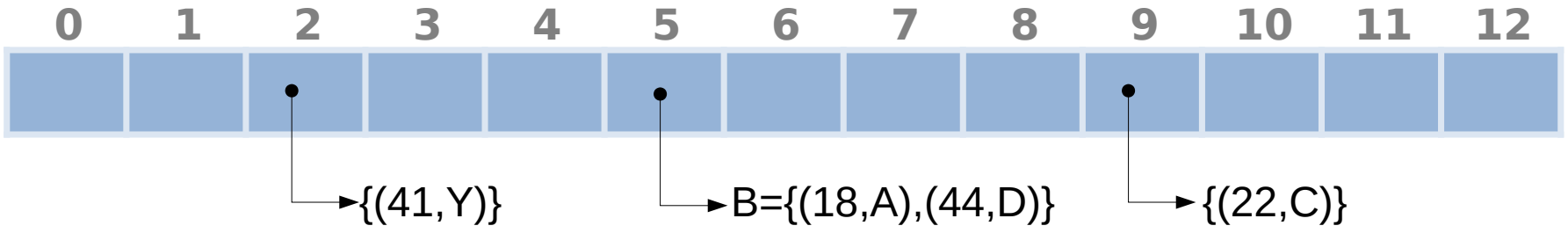
Utilizando la función de hash: $h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A> <41,Y> <22,C> <44,D> <18,E>

```
method put(k,v)
    if A[h(k)] = null
        List B = {}
        A[h(k)] = B
    else
        List B = A[h(k)]
        B.remove(k)
    B.insert(k,v)
```

k=18
v=E



Encadenamiento separado

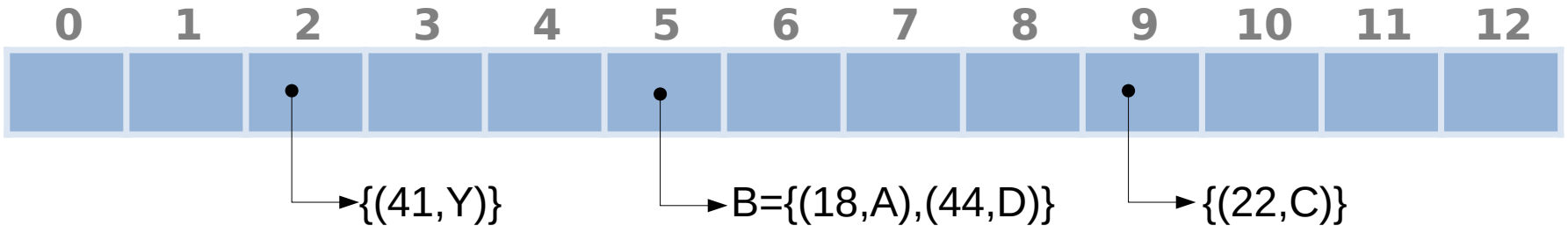
Utilizando la función de hash: $h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A> <41,Y> <22,C> <44,D> <18,E>

```
method put(k,v)
    if A[h(k)] = null
        List B = {}
        A[h(k)] = B
    else
        List B = A[h(k)]
        B.remove(k)
    B.insert(k,v)
```

k=18
v=E
h(k)=5



Encadenamiento separado

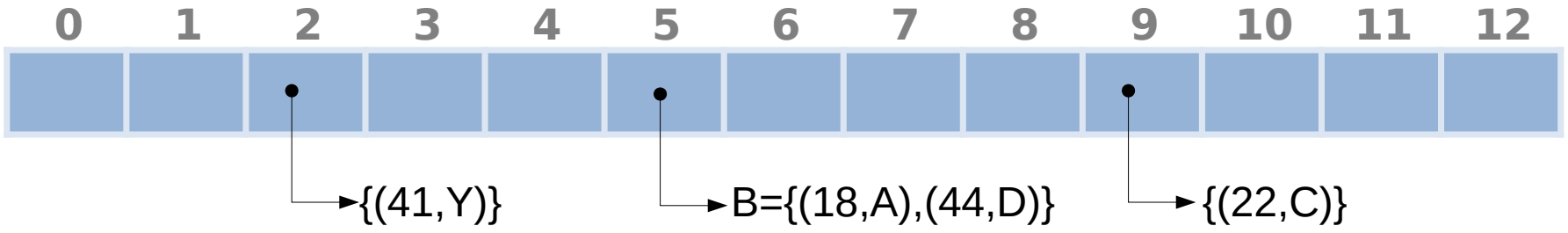
Utilizando la función de hash: $h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A> <41,Y> <22,C> <44,D> <18,E>

```
method put(k,v)
    if A[h(k)] = null
        List B = {}
        A[h(k)] = B
    else
        List B = A[h(k)]
        B.remove(k)
    B.insert(k,v)
```

k=18
v=E
h(k)=5



Encadenamiento separado

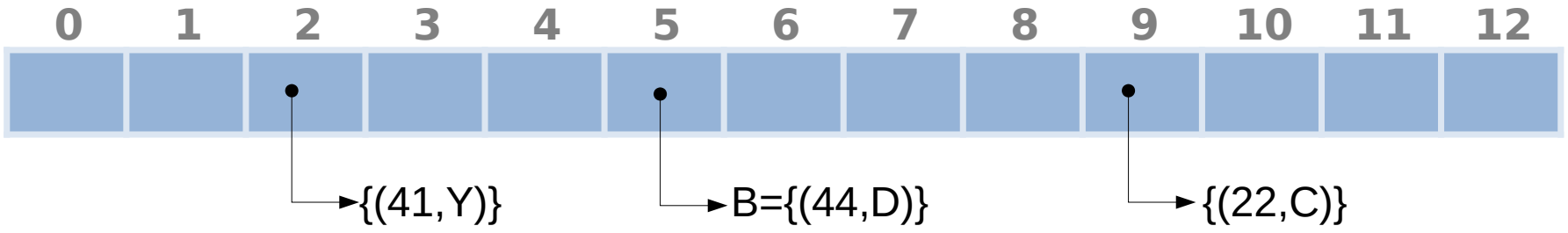
Utilizando la función de hash: $h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A> <41,Y> <22,C> <44,D> <18,E>

```
method put(k,v)
    if A[h(k)] = null
        List B = {}
        A[h(k)] = B
    else
        List B = A[h(k)]
        B.remove(k)
    B.insert(k,v)
```

k=18
v=E
h(k)=5



Encadenamiento separado

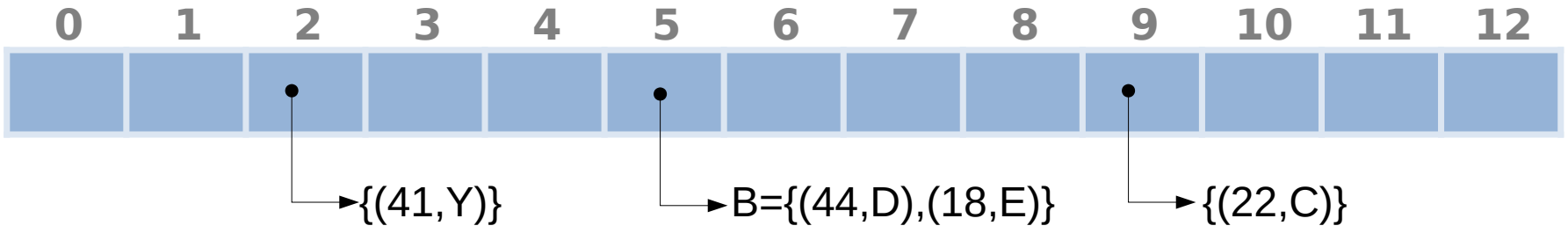
Utilizando la función de hash: $h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A> <41,Y> <22,C> <44,D> <18,E>

```
method put(k,v)
    if A[h(k)] = null
        List B = {}
        A[h(k)] = B
    else
        List B = A[h(k)]
        B.remove(k)
    B.insert(k,v)
```

k=18
v=E
h(k)=5



Resolución de colisiones: Direccionamiento abierto

Direccionamiento abierto

- DA: la entrada que **colisiona** se introduce en otra posición de la tabla
- **Prueba lineal**: resuelve las colisiones insertando la entrada en la siguiente posición libre (de manera circular).
  - Cada celda consultada se conoce como **prueba** "*probe*"
  - Los elementos que colisionan se agrupan en ***clusters***

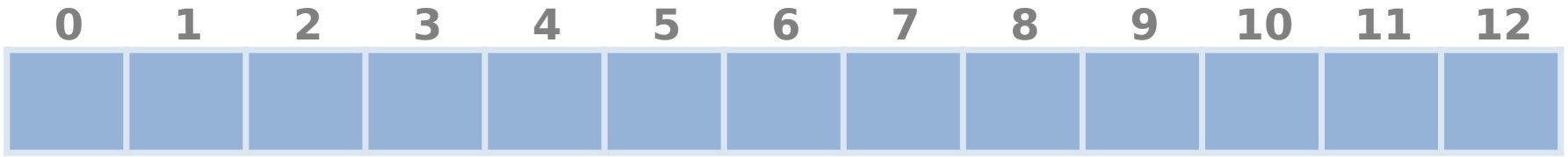
```
method put(k,v)
   p = 0
   do
      i = h(k) + p
      p = p + 1
   while !avaiable(A[i]) and p<N
   if p < N
      A[i] = B
   else
      throw no_space_avaiable_error
```

Direccionamiento abierto

Utilizando la función de hash:
$h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A>, <41,Y>, <22,C>, <44,D>, <59,E>, <32,F>, <31,Z>,<73,W>



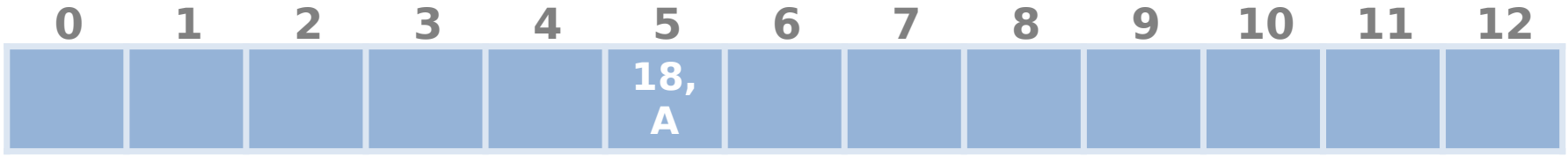
Direccionamiento abierto

Utilizando la función de hash:
$h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A>, <41,Y>, <22,C>, <44,D>, <59,E>, <32,F>, <31,Z>,<73,W>

k	*h(k)*	*Pruebas*
18	5	5



Direccionamiento abierto

Utilizando la función de hash:
$h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A>, <41,Y>, <22,C>, <44,D>, <59,E>, <32,F>, <31,Z>,<73,W>

k	*h(k)*	*Pruebas*
18	5	5
41	2	2

0	1	2	3	4	5	6	7	8	9	10	11	12
		41, Y			18, A							

Direccionamiento abierto

Utilizando la función de hash:
$h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A>, <41,Y>, <22,C>,
<44,D>, <59,E>, <32,F>,
<31,Z>,<73,W>

k	*h(k)*	*Pruebas*
18	5	5
41	2	2
22	9	9

0	1	2	3	4	5	6	7	8	9	10	11	12
		41, Y			18, A				22, C			

Direccionamiento abierto

Utilizando la función de hash:
$h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A>, <41,Y>, <22,C>, <44,D>, <59,E>, <32,F>, <31,Z>,<73,W>

k	*h(k)*	*Pruebas*
18	5	5
41	2	2
22	9	9
44	5	5

0	1	2	3	4	5	6	7	8	9	10	11	12
		41, Y			18, A				22, C			

Direccionamiento abierto

Utilizando la función de hash:
$h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A>, <41,Y>, <22,C>, <44,D>, <59,E>, <32,F>, <31,Z>,<73,W>

k	*h(k)*	*Pruebas*
18	5	5
41	2	2
22	9	9
44	5	5 6

0	1	2	3	4	5	6	7	8	9	10	11	12
		41, Y			18, A	44, D			22, C			

Direccionamiento abierto

Utilizando la función de hash:
$h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A>, <41,Y>, <22,C>, <44,D>, <59,E>, <32,F>, <31,Z>,<73,W>

k	*h(k)*	*Pruebas*
18	5	5
41	2	2
22	9	9
44	5	5 6
59	7	7

0	1	2	3	4	5	6	7	8	9	10	11	12
		41, Y			18, A	44, D	59, E		22, C			

Direccionamiento abierto

Utilizando la función de hash:
$h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A>, <41,Y>, <22,C>, <44,D>, <59,E>, <32,F>, <31,Z>,<73,W>

k	*h(k)*	*Pruebas*
18	5	5
41	2	2
22	9	9
44	5	5 6
59	7	7
32	6	6

0	1	2	3	4	5	6	7	8	9	10	11	12
		41, Y			18, A	44, D	59, E		22, C			

Direccionamiento abierto

Utilizando la función de hash:
$h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A>, <41,Y>, <22,C>, <44,D>, <59,E>, <32,F>, <31,Z>,<73,W>

k	*h(k)*	*Pruebas*
18	5	5
41	2	2
22	9	9
44	5	5 6
59	7	7
32	6	6 7

0	1	2	3	4	5	6	7	8	9	10	11	12
		41, Y			18, A	44, D	59, E		22, C			

Direccionamiento abierto

Utilizando la función de hash:
$h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A>, <41,Y>, <22,C>, <44,D>, <59,E>, <32,F>, <31,Z>,<73,W>

k	*h(k)*	*Pruebas*
18	5	5
41	2	2
22	9	9
44	5	5 6
59	7	7
32	6	6 7 8

0	1	2	3	4	5	6	7	8	9	10	11	12
		41, Y			18, A	44, D	59, E	32, F	22, C			

Direccionamiento abierto

Utilizando la función de hash:
$h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A>, <41,Y>, <22,C>, <44,D>, <59,E>, <32,F>, <31,Z>,<73,W>

k	*h(k)*	*Pruebas*
18	5	5
41	2	2
22	9	9
44	5	5 6
59	7	7
32	6	6 7 8
31	5	5

0	1	2	3	4	5	6	7	8	9	10	11	12
		41, Y			18, A	44, D	59, E	32, F	22, C			

Direccionamiento abierto

Utilizando la función de hash:
$h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A>, <41,Y>, <22,C>, <44,D>, <59,E>, <32,F>, <31,Z>,<73,W>

k	*h(k)*	*Pruebas*
18	5	5
41	2	2
22	9	9
44	5	5 6
59	7	7
32	6	6 7 8
31	5	5 6 7 8 9 10

0	1	2	3	4	5	6	7	8	9	10	11	12
		41, Y			18, A	44, D	59, E	32, F	22, C	31, Z		

Direccionamiento abierto

Utilizando la función de hash:
$h(k) = k \bmod 13$

Insertar las siguientes entradas:

<18,A>, <41,Y>, <22,C>, <44,D>, <59,E>, <32,F>, <31,Z>,<73,W>

k	*h(k)*	*Pruebas*
18	5	5
41	2	2
22	9	9
44	5	5 6
59	7	7
32	6	6 7 8
31	5	5 6 7 8 9 10
73	8	8

0	1	2	3	4	5	6	7	8	9	10	11	12
		41, Y			18, A	44, D	59, E	32, F	22, C	31, Z		

Direccionamiento abierto

Utilizando la función de hash:
$h(k) = k \bmod 13$

Insertar las siguientes entradas:

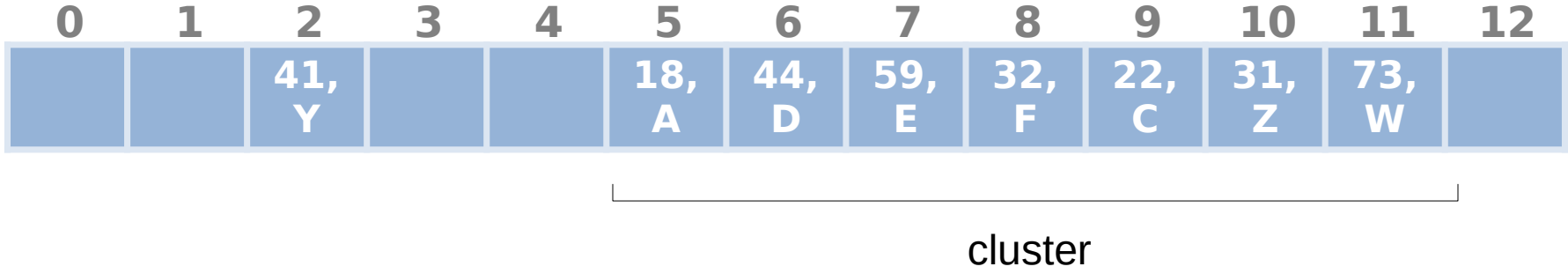
<18,A>, <41,Y>, <22,C>, <44,D>, <59,E>, <32,F>, <31,Z>,<73,W>

k	*h(k)*	*Pruebas*
18	5	5
41	2	2
22	9	9
44	5	5 6
59	7	7
32	6	6 7 8
31	5	5 6 7 8 9 10
73	8	8 9 10 11

0	1	2	3	4	5	6	7	8	9	10	11	12
		41, Y			18, A	44, D	59, E	32, F	22, C	31, Z	73, W	

Direccionamiento abierto

- El direccionamiento abierto con prueba lineal tiene el problema de que genera agrupaciones de datos que llamamos clusters.
- Cuando se producen colisiones, los datos se van agrupando en clusters.
- Estos clusrters reducen la eficiencia de las tablas de hash, haciendo que la complejidad en la inserción deje de ser O(1) para el caso medio.



cualquier inserción en el rango de un cluster genera una sucesión de pruebas

Direccionamiento abierto (búsqueda)

- **Búsqueda** en una tabla con prueba lineal `get(k)`:
  - Se empieza buscando en *h(k)*
  - Se prueban celdas consecutivas hasta que ocurre:
    - Se encuentra el elemento con clave *k*
    - Se encuentra una celda vacía (ojo con los borrados)
    - Se recorren *N* celdas sin encontrar el elemento

```
method get(k)
  i ← h(k)
  p ← 0
  repeat
    c ← A[i]
    if avaiable(c)
      return null
    else if c.key() = k
      return c.element()
    i ← (i + 1) mod N
    p ← p + 1
  until p = N
  return null
```

Direccionamiento abierto (actualización)

- **Búsquedas y borrados** necesitan ***available***
  - posición que fue borrada

- `remove(k)`
  - Se busca el la entrada con clave `k`
  - Si se encuentra, se reemplaza por *available*
  - En caso contrario se devuelve `null`

- `put(k,e)`
  - Se lanza una excepción si la tabla está llena
  - Se empieza en la celda *h(k)*
  - Se prueban celdas consecutivas hasta que:
    - Se encuentra una celda vacía o *available*
    - Se recorren *N* celdas sin encontrar el elemento
  - Se almacena entrada `(k,e)` en la celda *i*

Prueba cuadrática

Son aquellas que utilizan funciones hash con forma:

$$h(k,p) = (h_c(k) + c_1 p + c_2 p^2) \bmod N$$

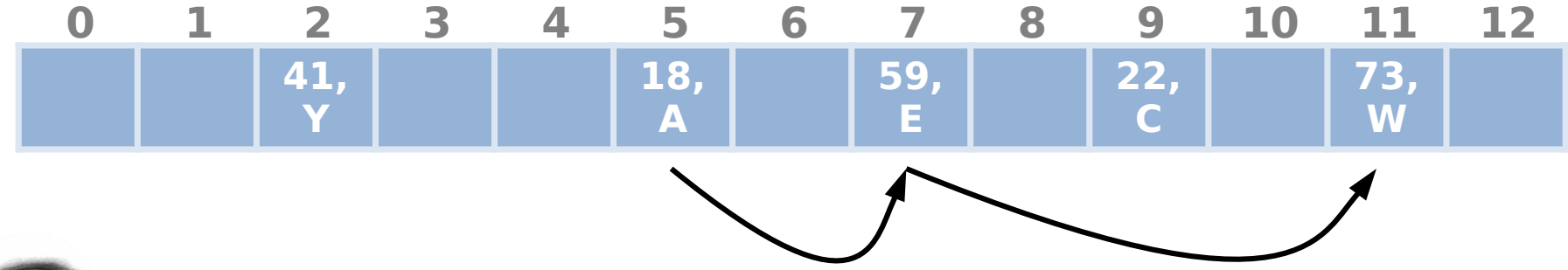
donde c_1 y c_2 son constantes auxiliares y $c_2 \neq 0$

```
method put(k,v)
  p = 0
  do
    i = (hc(k)+c1*p+c2*p*p) mod N
    p = p + 1
    if (avaiable(i))
      i_aux = i
    else if A[i].key = k
      i_aux = i
  while A[i]!=null and p<N
  if p < N
    A[i_aux] = B
  else
    throw no_space_avaiable_error
```

Prueba cuadrática

La posición inicial es $A[h(k)]$ y las siguientes dependen de manera cuadrática del número de prueba p.

- Evita el clustering primario, de manera que no se crean agrupaciones de elementos en el array. Esto mejora el comportamiento en el tiempo medio de inserción.
- Sin embargo, como dos claves con la misma posición inicial generan la misma secuencia de prueba, se produce clustering secundario.
- El clustering secundario consiste en que en caso de colisión se generan series largas de pruebas.



Hashing doble

Son aquellas que utilizan funciones hash con forma:

$$h(k,p) = (h_c(k) + d(k)p) \bmod N$$

donde d es es una función de hash auxiliar de forma:

$d(k) = q - (h_c(k) \bmod q)$, q entero primo y menor que N.

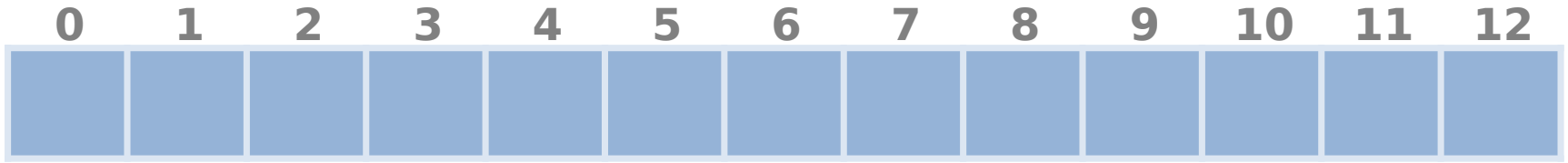
- Evita que se formen clusters primarios.
- Evita que se formen clusters secundarios gracias a que en cada prueba se tiene en cuenta el valor de la clave.
- Es uno de los mejores métodos de hash.

```
method put(k,v)
  p = 0
  do
    i = (h(k)+d(k)*p) mod N
    p = p + 1
    if (avaiable(i))
      i_aux = i
    else if A[i].key = k
      i_aux = i
  while A[i]!=null and p<N
  if p < N
    A[i_aux] = B
  else
    throw no_space_avaiable_error
```

Ejemplo de Hashing doble

- Tabla hash con doble hashing y clave entera
  - N = 13
  - h(k) = k mod 13
  - d(k) = 7 – k mod 7
- Insertar las claves:
  - <18,A>, <41,Y>, <22,C>, <44,D>, <59,E>, <32,F>, <31,Z>, <73,W>

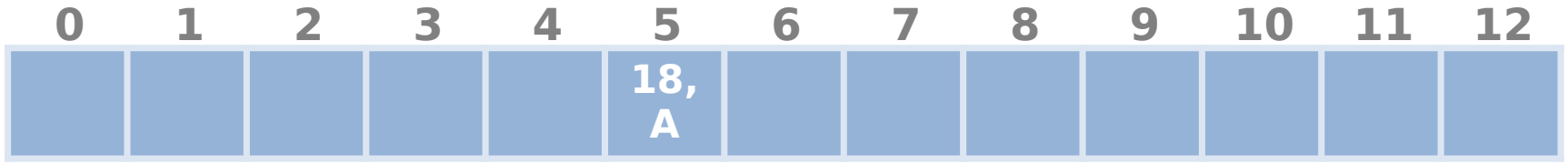
k	*h(k)*	d(k)	*Prueba*



Ejemplo de Hashing doble

- Tabla hash con doble hashing y clave entera
  - N = 13
  - h(k) = k mod 13
  - d(k) = 7 – k mod 7
- Insertar las claves:
  - <18,A>, <41,Y>, <22,C>, <44,D>, <59,E>, <32,F>, <31,Z>, <73,W>

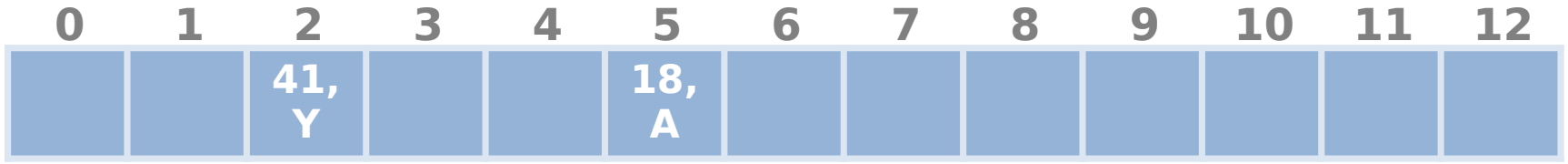
k	*h(k)*	d(k)	*Prueba*
18	5		5



Ejemplo de Hashing doble

- Tabla hash con doble hashing y clave entera
  - N = 13
  - h(k) = k mod 13
  - d(k) = 7 – k mod 7
- Insertar las claves:
  - <18,A>, <41,Y>, <22,C>, <44,D>, <59,E>, <32,F>, <31,Z>, <73,W>

k	*h(k)*	d(k)	*Prueba*
18	5		5
41	2		2



Ejemplo de Hashing doble

- Tabla hash con doble hashing y clave entera
  - N = 13
  - h(k) = k mod 13
  - d(k) = 7 – k mod 7
- Insertar las claves:
  - <18,A>, <41,Y>, <22,C>, <44,D>, <59,E>, <32,F>, <31,Z>, <73,W>

k	*h(k)*	d(k)	*Prueba*
18	5		5
41	2		2
22	9		9

0	1	2	3	4	5	6	7	8	9	10	11	12
		41, Y			18, A				22, C			

Ejemplo de Hashing doble

- Tabla hash con doble hashing y clave entera
  - N = 13
  - h(k) = k mod 13
  - d(k) = 7 – k mod 7
- Insertar las claves:
  - <18,A>, <41,Y>, <22,C>, <44,D>, <59,E>, <32,F>, <31,Z>, <73,W>

k	*h(k)*	d(k)	*Prueba*
18	5		5
41	2		2
22	9		9
44	5		5

0	1	2	3	4	5	6	7	8	9	10	11	12
		41, Y			18, A				22, C			

Ejemplo de Hashing doble

- Tabla hash con doble hashing y clave entera
  - N = 13
  - h(k) = k mod 13
  - d(k) = 7 – k mod 7
- Insertar las claves:
  - <18,A>, <41,Y>, <22,C>, <44,D>, <59,E>, <32,F>, <31,Z>, <73,W>

k	*h(k)*	d(k)	*Prueba*
18	5		5
41	2		2
22	9		9
44	5	5	5 10

0	1	2	3	4	5	6	7	8	9	10	11	12
		41, Y			18, A				22, C	44, D		

Ejemplo de Hashing doble

- Tabla hash con doble hashing y clave entera
  - N = 13
  - h(k) = k mod 13
  - d(k) = 7 – k mod 7
- Insertar las claves:
  - <18,A>, <41,Y>, <22,C>, <44,D>, <59,E>, <32,F>, <31,Z>, <73,W>

k	*h(k)*	d(k)	*Prueba*
18	5		5
41	2		2
22	9		9
44	5	5	5 10
59	7		7

0	1	2	3	4	5	6	7	8	9	10	11	12
		41, Y			18, A		59, E		22, C	44, D		

Ejemplo de Hashing doble

- Tabla hash con doble hashing y clave entera
  - N = 13
  - h(k) = k mod 13
  - d(k) = 7 – k mod 7
- Insertar las claves:
  - <18,A>, <41,Y>, <22,C>, <44,D>, <59,E>, <32,F>, <31,Z>, <73,W>

k	*h(k)*	d(k)	*Prueba*
18	5		5
41	2		2
22	9		9
44	5	5	5 10
59	7		7
32	6		6

0	1	2	3	4	5	6	7	8	9	10	11	12
		41, Y			18, A	32, F	59, E		22, C	44, D		

Ejemplo de Hashing doble

- Tabla hash con doble hashing y clave entera
  - N = 13
  - h(k) = k mod 13
  - d(k) = 7 – k mod 7
- Insertar las claves:
  - <18,A>, <41,Y>, <22,C>, <44,D>, <59,E>, <32,F>, <31,Z>, <73,W>

k	*h(k)*	d(k)	*Prueba*
18	5		5
41	2		2
22	9		9
44	5	5	5 10
59	7		7
32	6		6
31	5	4	5

0	1	2	3	4	5	6	7	8	9	10	11	12
		41, Y			18, A	32, F	59, E		22, C	44, D		

Ejemplo de Hashing doble

- Tabla hash con doble hashing y clave entera
  - N = 13
  - h(k) = k mod 13
  - d(k) = 7 – k mod 7
- Insertar las claves:
  - <18,A>, <41,Y>, <22,C>, <44,D>, <59,E>, <32,F>, <31,Z>, <73,W>

k	*h(k)*	d(k)	*Prueba*
18	5		5
41	2		2
22	9		9
44	5	5	5 10
59	7		7
32	6		6
31	5	4	5 9

0	1	2	3	4	5	6	7	8	9	10	11	12
		41, Y			18, A	32, F	59, E		22, C	44, D		

Ejemplo de Hashing doble

- Tabla hash con doble hashing y clave entera
  - N = 13
  - h(k) = k mod 13
  - d(k) = 7 – k mod 7
- Insertar las claves:
  - <18,A>, <41,Y>, <22,C>, <44,D>, <59,E>, <32,F>, <31,Z>, <73,W>

k	*h(k)*	d(k)	*Prueba*
18	5		5
41	2		2
22	9		9
44	5	5	5 10
59	7		7
32	6		6
31	5	4	5 9 0

0	1	2	3	4	5	6	7	8	9	10	11	12
31, Z		41, Y			18, A	32, F	59, E		22, C	44, D		

Ejemplo de Hashing doble

- Tabla hash con doble hashing y clave entera
  - N = 13
  - h(k) = k mod 13
  - d(k) = 7 – k mod 7
- Insertar las claves:
  - <18,A>, <41,Y>, <22,C>, <44,D>, <59,E>, <32,F>, <31,Z>, <73,W>

k	*h(k)*	d(k)	*Prueba*
18	5		5
41	2		2
22	9		9
44	5	5	5 10
59	7		7
32	6		6
31	5	4	5 9 0
73	8		8

0	1	2	3	4	5	6	7	8	9	10	11	12
31, Z		41, Y			18, A	32, F	59, E	73, W	22, C	44, D		

Tablas Hash: Conclusiones

Factor de carga y *rehashing*

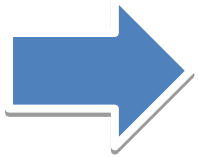
- Factor de carga (λ)

$$\lambda = \lceil n/N \rceil$$

  - Interesa que se mantenga menor que 1
    - Direccionamiento abierto < 0.5
    - Encadenamiento separado <0.75

¿Estrategia a adoptar si el factor de carga supera el umbral establecido?

  - Redimensionar la tabla
    - Tamaño?. Doblar el actual.
    - Definir la nueva función de hash
      - Atendiendo al nuevo tamaño
      - Calculando nuevos parámetros: a y b en MAD
    - Re-insertar las entradas en la nueva tabla



Rehashing

Conclusiones de tablas *Hash*

- En el **caso peor** las tablas *hash* tienen una complejidad ***O(n)*** para todas las operaciones
  - Sólo ocurriría cuando todas las claves colisionan
  - La **complejidad esperada** para las operaciones es ***O(1)***
- El **factor de carga** $a = n/N$ afecta al rendimiento de la tabla
  - No debe estar próximo al 100%
- **Aplicaciones** prácticas
  - Pequeñas BBDD
  - Compiladores
  - Caches de navegadores

Conclusiones de tablas *Hash*

Código de Hash

Permite obtener para cualquier objeto un número.

Idealmente se busca una relación biyectiva.

Se suele implementar en los propios objetos que se insertan en la estructura de datos.

No debe cambiar a lo largo de la vida del objeto, debe ser inmutable.

Función de resumen

Restringen el código hash al tamaño de la tabla.

Funciones de resumen:

- Truncamiento
- MAD

Se suele implementar dentro de la estructura de datos.

Encadenamiento separado

Cuando se produce colisión se utiliza una lista para almacenar los diferentes valores.

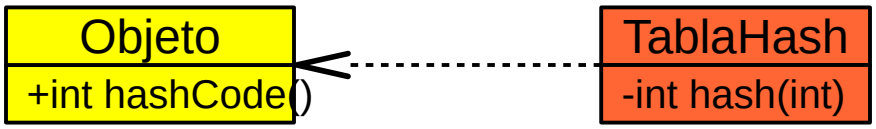
El tamaño no es fijo.

Direccionamiento abierto

Cuando se produce colisión los valores se almacenan en la misma tabla.

Necesita resolución de colisiones:

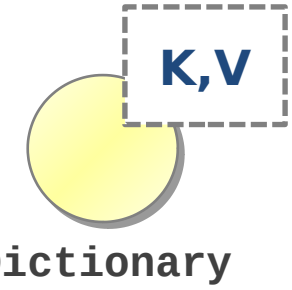
- Prueba lineal
- Prueba cuadrática
- Hashing doble



Diccionarios

Diccionarios

- El **TAD Diccionario** modela un colección de pares (clave, valor) en la que se puede buscar de forma eficiente
- La **diferencia** fundamental con los **mapas** es que está permitido que varios elementos tengan la misma clave
- **Aplicaciones**
  - Pares <término – definición del término>
  - Autorización de tarjetas de crédito
  - “Mapeo” DNS de hosts (un nombre con varias IPs)

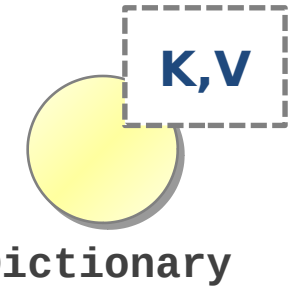


TAD Diccionario

- `size()`, `isEmpty()`
- `find(k)`: si el diccionario tiene la entrada con clave `k`, retorna la primera ocurrencia. En caso contrario, `null`
- `findAll(k)`: retorna una colección iterable que almacena todas las entradas con clave `k`
- `insert(k, v)`: inserta y retorna la entrada `(k, v)`
- `remove(k,v)`: si el diccionario tiene una entrada `<k,v>` la elimina y devuelve dicha entrada; en caso contrario devuelve `null`
- `entries()`: devuelve un colección iterable con todas las entradas del diccionario

Diccionarios

```
public interface Dictionary<K, V> {

   public int size();

   public boolean isEmpty();

   public Entry<K, V> find(K key) throws InvalidKeyException;

   public Iterable<Entry<K, V>> findAll(K key)
               throws InvalidKeyException;

   public Entry<K, V> insert(K key, V value)
               throws InvalidKeyException;

   public Entry<K, V> remove(Entry<K, V> e)
               throws InvalidEntryException;

   public Iterable<Entry<K, V>> entries();
}
```



Diccionarios (Ejemplo de uso)

Operation	Salida	Diccionario
`insert(5,A)`	`(5,A)`	(5,A)
`insert(7,B)`	`(7,B)`	(5,A),(7,B)
`insert(2,C)`	`(2,C)`	(5,A),(7,B),(2,C)
`insert(8,D)`	`(8,D)`	(5,A),(7,B),(2,C),(8,D)
`insert(2,E)`	`(2,E)`	(5,A),(7,B),(2,C),(8,D),(2,E)
`find(7)`	`(7,B)`	(5,A),(7,B),(2,C),(8,D),(2,E)
`find(4)`	`null`	(5,A),(7,B),(2,C),(8,D),(2,E)
`find(2)`	`(2,C)`	(5,A),(7,B),(2,C),(8,D),(2,E)
`findAll(2)`	`(2,C),(2,E)`	(5,A),(7,B),(2,C),(8,D),(2,E)
`size()`	`5`	(5,A),(7,B),(2,C),(8,D),(2,E)
`remove(find(5))`	`(5,A)`	(7,B),(2,C),(8,D),(2,E)
`find(5)`	`null`	(7,B),(2,C),(8,D),(2,E

Diccionarios (Implementaciones)

- **Diccionarios desordenados**
  - **Estructuras de datos lineales** (listas, *array*, etc.)
    - Inserción, borrado y búsqueda es *O(n)*
  - **Tablas Hash**.
    - Encadenamiento separado
      - Inserción, borrado y búsqueda = inserción, borrado y búsqueda en A[h(k)]
        - » En promedio O(1)
        - » Pocas entradas con misma clave en comparación con el tamaño del diccionario

Casos de uso

Utilidades de HashSet, HashMap y HashDictionary:

- Saber si un elemento pertenece a un conjunto en O(1)
  - Comprobar la no repetición de elementos que se recorren en un algoritmo.
  - Almacenar elementos (eliminando duplicados).
- Indexar elementos por un elemento complejo en O(1)
  - Crear un diccionario de objetos indexado por una palabra.
  - Usarla como estructura de datos secundaria de otra principal para localizar rápidamente elementos de la primera.