



Instituto Politécnico Nacional  
La Técnica al Servicio de la Patria

**Unidad Profesional Interdisciplinaria de  
Ingeniería Campus Tlaxcala UPIIT**

# Algoritmos y Estructuras de Datos

Esaú Eliezer Escobar Juárez

Ingeniería en Inteligencia Artificial (IIA)

---



# Algoritmos de búsqueda

# Concepto de Búsqueda

- La búsqueda es una operación que tiene por objeto la localización de un elemento dentro de la estructura de datos.
- Cuando se realizan operaciones de búsqueda es preciso considerar la eficiencia(complejidad) de los algoritmos empleados en la búsqueda. En el grado de eficiencia en una búsqueda será vital cuando se trata de localizar una información en una lista o tabla memoria, o bien en un archivo de datos.

# Problema: Búsqueda

- Se dan una lista de registros.
- Cada registro tiene una llave asociada.
- Proporcione un algoritmo eficiente para buscar un registro que contenga una clave en particular.
- La eficiencia se cuantifica en términos de análisis de tiempo promedio (número de comparaciones) para recuperar un artículo.

# Búsqueda

[ 0 ]

[ 1 ]

[ 2 ]

[ 3 ]

[ 4 ]

[ 700 ]

Number 701466868



Number 281942902



Number 233667136



Number 580625685



Number 506643548



...

Number 155778322



Cada registro de la lista tiene una clave asociada. En este ejemplo, las claves son números de identificación.

Dada una clave en particular, ¿cómo podemos recuperar de manera eficiente el registro de la lista?

**Number 580625685**



# Tipos de algoritmos de búsqueda

- Búsqueda secuencial
- Búsqueda binaria
- Búsqueda por índice

# Búsqueda secuencial o serial

- Recorra la variedad de registros, uno a la vez.
- Busque un registro con clave coincidente.
- La búsqueda se detiene cuando
  - se encuentra el registro con clave coincidente
  - o cuando la búsqueda ha examinado todos los registros sin éxito.

# Pseudocódigo para la búsqueda serial

```
// Search for a desired item in the n array elements
// starting at a[first].
// Returns pointer to desired record if found.
// Otherwise, return NULL
...
for(i = first; i < n; ++i )
    if(a[first+i] is desired item)
        return &a[first+i];

// if we drop through loop, then desired item was
// not found
return NULL;
```



# Análisis de la búsqueda serial

- ¿Cuáles son los tiempos de ejecución promedio y peores para la búsqueda en serie?
- Debemos determinar la notación  $O$  para el número de operaciones requeridas en la búsqueda.
- El número de operaciones depende de  $n$ , el número de entradas en la lista.

# Tiempo del peor caso para la búsqueda serial

- Para una matriz de  $n$  elementos, el peor de los casos para la búsqueda en serie requiere  $n$  accesos a la matriz:  $O(n)$ .
- Considere los casos en los que debemos recorrer todos los  $n$  registros:
  - el registro deseado aparece en la última posición de la matriz
  - el registro deseado no aparece en la matriz en absoluto

# Caso promedio para la búsqueda serial

- Supuestos:

1. Todas las claves son igualmente probables en una búsqueda
2. Siempre buscamos una clave que esté en el arreglo.

- Ejemplo:

- Tenemos un arreglo de 10 registros.
- Si busca el primer registro, entonces requiere 1 acceso al arreglo; si es el segundo, entonces 2 accesos al arreglo. etc.

- El promedio de todas estas búsquedas es:

$$(1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10) / 10 = 5,5$$

# Tiempo del caso promedio para la búsqueda serial

Generalizar para un tamaño de arreglo  $n$ .

Expresión para el tiempo de ejecución del caso promedio:

$$(1 + 2 + \dots + n) / n = n(n + 1) / 2n = (n + 1) / 2$$

Por lo tanto, la complejidad del tiempo de caso promedio para la búsqueda en serie es  $O(n)$ .

# Búsqueda binaria

- ¿Quizás podamos hacerlo mejor que  $O(n)$  en el caso promedio?
- Supongamos que se nos da un arreglo de registros ordenados. Por ejemplo:
  - Un arreglo de registros con claves enteras ordenadas de menor a mayor (por ejemplo, números de identificación), o
  - Un arreglo de registros con claves de cadena ordenadas en orden alfabético (por ejemplo, nombres)

# Pseudocódigo de la búsqueda binaria

```
...
if(size == 0)
    found = false;
else {
    middle = index of approximate midpoint of array;
    if(target == a[middle])
        target has been found!
    else if(target < a[middle])
        search for target in area before midpoint;
    else
        search for target in area after midpoint;
}
...
```

# Búsqueda Binaria

Ejemplo: arreglo ordenado de llaves enteras. Target=7.

[ 0 ]	[ 1 ]	[ 2 ]	[ 3 ]	[ 4 ]	[ 5 ]	[ 6 ]
3	6	7	11	32	33	53

# Búsqueda Binaria

Ejemplo: arreglo ordenado de llaves enteras. Target=7.

[ 0 ]	[ 1 ]	[ 2 ]	[ 3 ]	[ 4 ]	[ 5 ]	[ 6 ]
3	6	7	11	32	33	53



Encontrar el punto medio aproximado



# Búsqueda Binaria

Ejemplo: arreglo ordenado de llaves enteras. Target=7.

[ 0 ]	[ 1 ]	[ 2 ]	[ 3 ]	[ 4 ]	[ 5 ]	[ 6 ]
3	6	7	11	32	33	53



¿7 = la llave del punto medio? NO.

# Búsqueda Binaria

Ejemplo: arreglo ordenado de llaves enteras. Target=7.

[ 0 ]	[ 1 ]	[ 2 ]	[ 3 ]	[ 4 ]	[ 5 ]	[ 6 ]
3	6	7	11	32	33	53




¿7 < la llave del punto medio? YES.

# Búsqueda Binaria

Ejemplo: arreglo ordenado de llaves enteras. Target=7.

[ 0 ]	[ 1 ]	[ 2 ]	[ 3 ]	[ 4 ]	[ 5 ]	[ 6 ]
3	6	7	11	32	33	53



Buscar el target en el área antes del punto medio.

# Búsqueda Binaria

Ejemplo: arreglo ordenado de llaves enteras. Target=7.

[ 0 ]	[ 1 ]	[ 2 ]	[ 3 ]	[ 4 ]	[ 5 ]	[ 6 ]
3	6	7	11	32	33	53



Encontrar el punto medio aproximado

# Búsqueda Binaria

Ejemplo: arreglo ordenado de llaves enteras. Target=7.

[ 0 ]	[ 1 ]	[ 2 ]	[ 3 ]	[ 4 ]	[ 5 ]	[ 6 ]
3	6	7	11	32	33	53



¿Target = la llave del punto medio? NO.

# Búsqueda Binaria

Ejemplo: arreglo ordenado de llaves enteras. Target=7.

[ 0 ]	[ 1 ]	[ 2 ]	[ 3 ]	[ 4 ]	[ 5 ]	[ 6 ]
3	6	7	11	32	33	53



¿Target < la llave del punto medio? NO.

# Búsqueda Binaria

Ejemplo: arreglo ordenado de llaves enteras. Target=7.

[ 0 ]	[ 1 ]	[ 2 ]	[ 3 ]	[ 4 ]	[ 5 ]	[ 6 ]
3	6	7	11	32	33	53

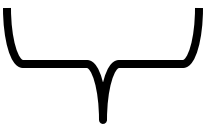


¿Target > la llave del punto medio? SI.

# Búsqueda Binaria

Ejemplo: arreglo ordenado de llaves enteras. Target=7.

[ 0 ]	[ 1 ]	[ 2 ]	[ 3 ]	[ 4 ]	[ 5 ]	[ 6 ]
3	6	7	11	32	33	53



Buscar el target en el área después del punto medio.



# Búsqueda Binaria

Ejemplo: arreglo ordenado de llaves enteras. Target=7.

[ 0 ]	[ 1 ]	[ 2 ]	[ 3 ]	[ 4 ]	[ 5 ]	[ 6 ]
3	6	7	11	32	33	53



Encontrar el punto medio aproximado.  
¿Target = llave del punto medio? SI.

# Implementación de la búsqueda binaria

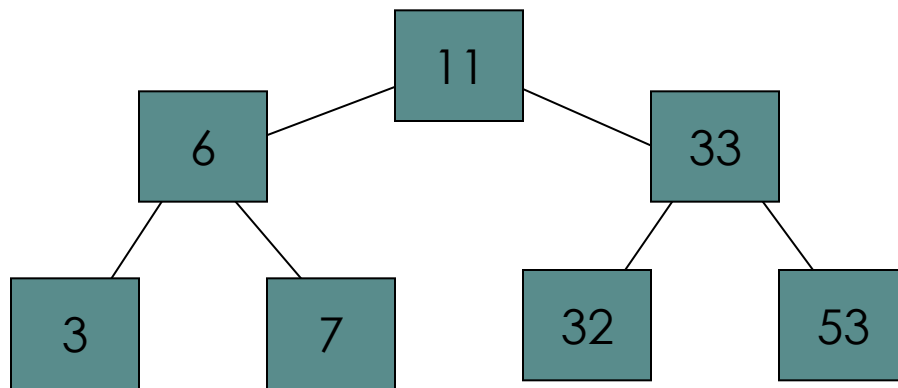
```
void search(const int a[ ], size_t first, size_t size, int target,
            int& found, size_t& location)
{
    size_t middle;
    if(size == 0) found = false;
    else {
        middle = first + size/2;
        if(target == a[middle]){
            location = middle;
            found = true;
        }
        else if (target < a[middle])
            // target is less than middle, search subarray before middle
            search(a, first, size/2, target, found, location);
        else
            // target is greater than middle, search subarray after middle
            search(a, middle+1, (size-1)/2, target, found, location);
    }
}
```

# Relación con el árbol de búsqueda binaria

Arreglo del ejemplo previo:

3	6	7	11	32	33	53
---	---	---	----	----	----	----

Árbol de búsqueda binaria completo correspondiente

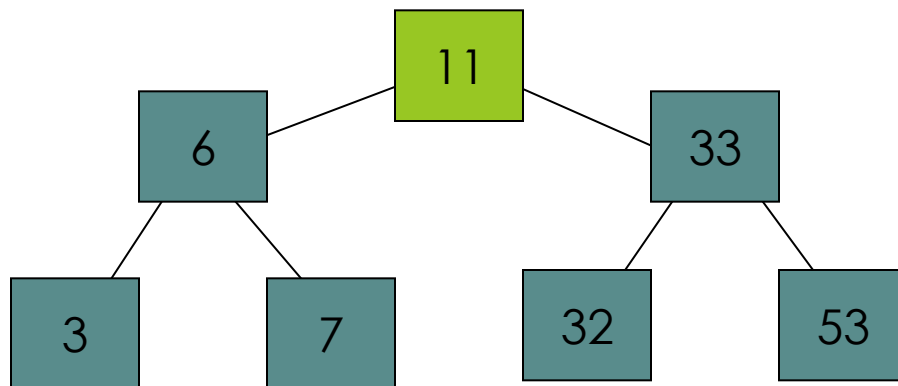


# Buscar target = 7

Encontrar punto medio:

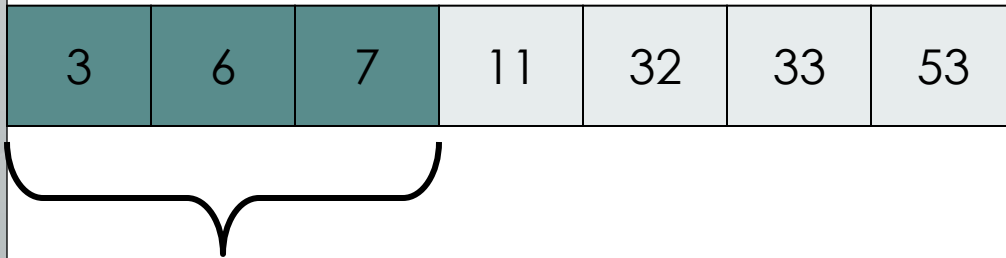
3	6	7	11	32	33	53
---	---	---	----	----	----	----

Comenzar en la raíz:

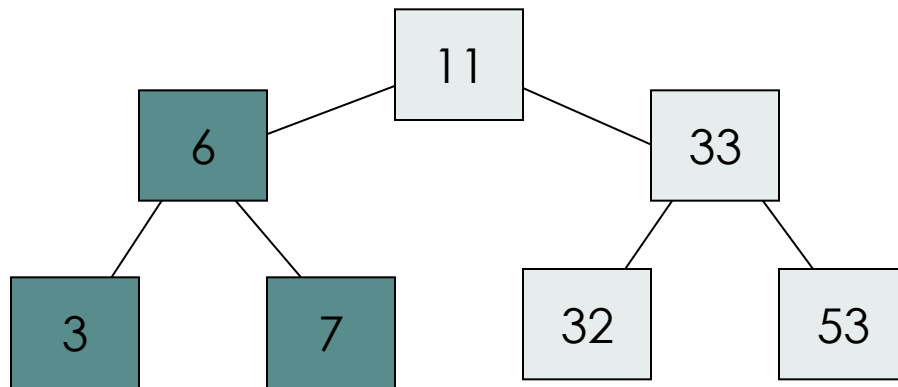


# Buscar target = 7

Buscar en el sub-arreglo izquierdo:



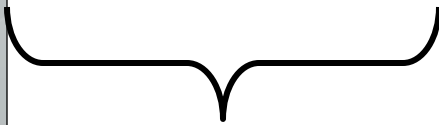
Buscar en el sub-arbol izquierdo:



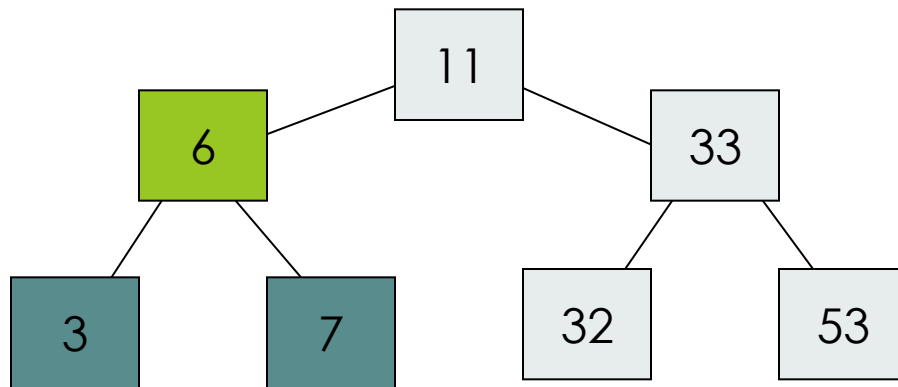
# Buscar target = 7

Encontrar el punto medio del sub-arreglo aproximadamente

3	6	7	11	32	33	53
---	---	---	----	----	----	----



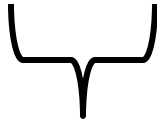
Visitar la raíz del sub-arbol



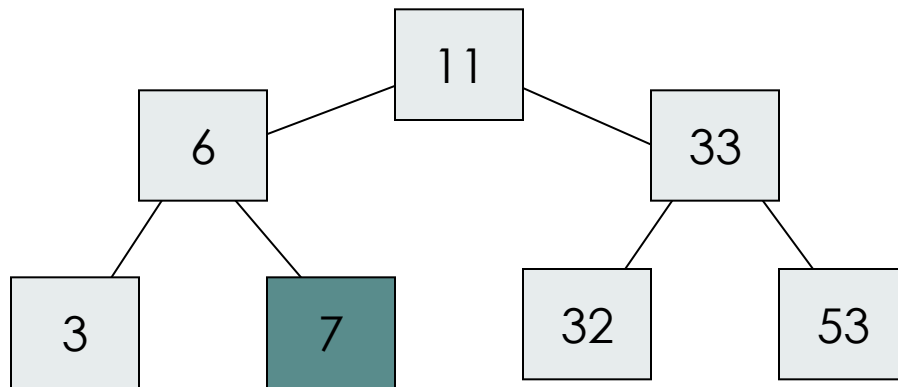
# Buscar target = 7

Buscar en el sub-arreglo derecho

3	6	7	11	32	33	53
---	---	---	----	----	----	----



Buscar en el sub-arbol derecho



# Búsqueda binaria: Análisis

- ¿La peor complejidad del caso?
- ¿Cuál es la profundidad máxima de las llamadas recursivas en la búsqueda binaria en función de  $n$ ?
- En cada nivel de la recursividad, dividimos la matriz por la mitad (dividimos por dos).
  - Por lo tanto, la profundidad máxima de recursividad es  $\text{floor}(\log_2 n)$ , el peor de los casos =  $O(\log_2 n)$ .
  - El caso promedio también es =  $O(\log_2 n)$ .





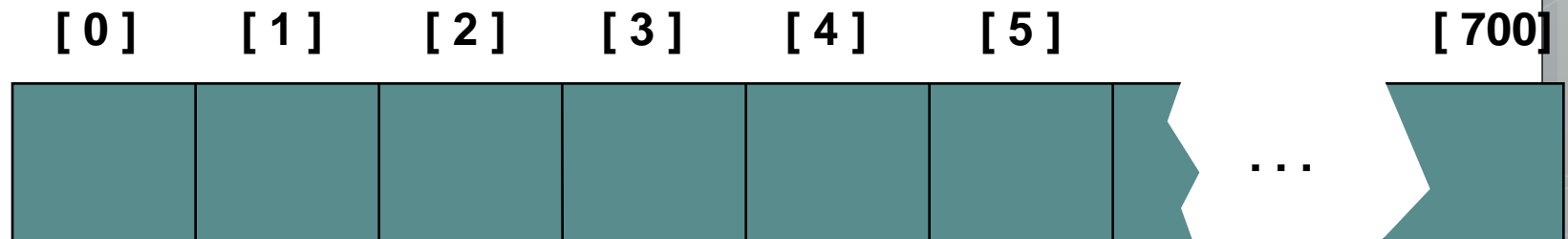
¿Podemos hacerlo mejor que  $O(\log_2 n)$ ?

- Promedio y peor caso de búsqueda en serie =  $O(n)$
- Promedio y peor caso de búsqueda binaria =  $O(\log_2 n)$
- ¿Podemos hacerlo mejor que esto?

SI. ¡Usa una tabla hash!

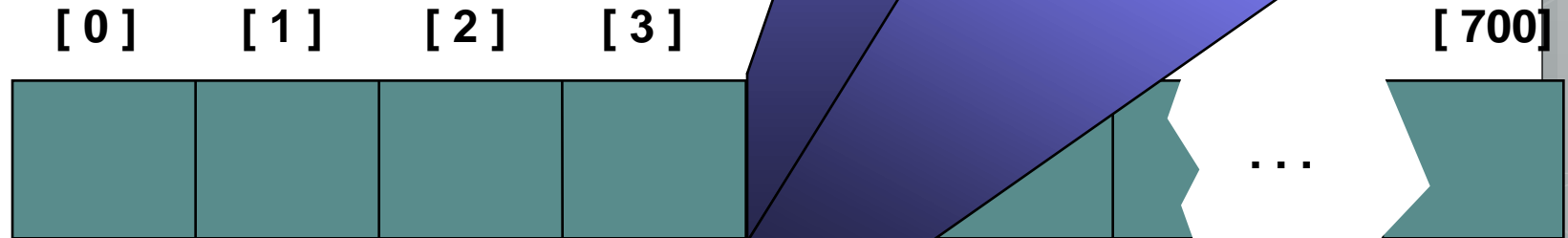
# ¿Qué es una tabla hash?

- El tipo más simple de tabla hash es una matriz de registros.
- Este ejemplo tiene 701 registros.



## ¿Qué es una tabla hash?

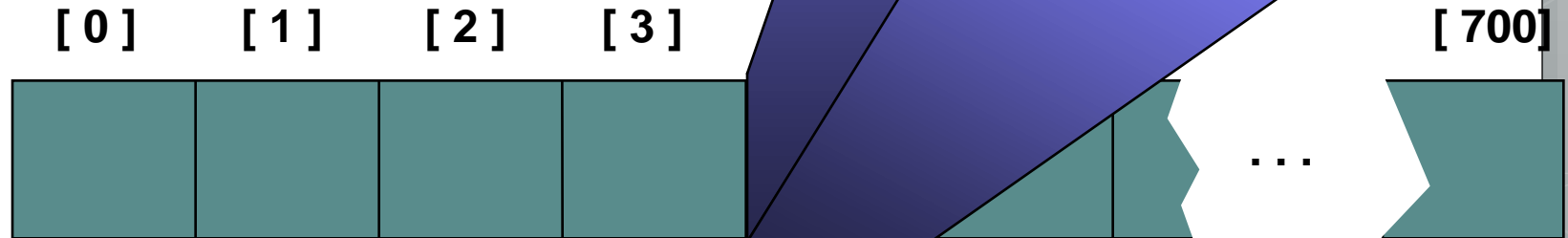
- Cada registro tiene un campo especial, llamado **clave**.
- En este ejemplo, la clave es un campo entero largo llamado Número.



[ 4 ]

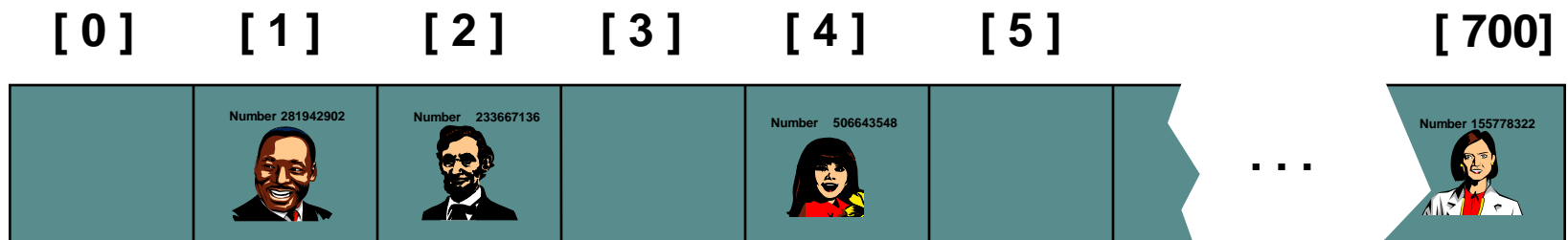
## ¿Qué es una tabla hash?

- El número puede ser el número de identificación de una persona y el resto del registro contiene información sobre la persona.



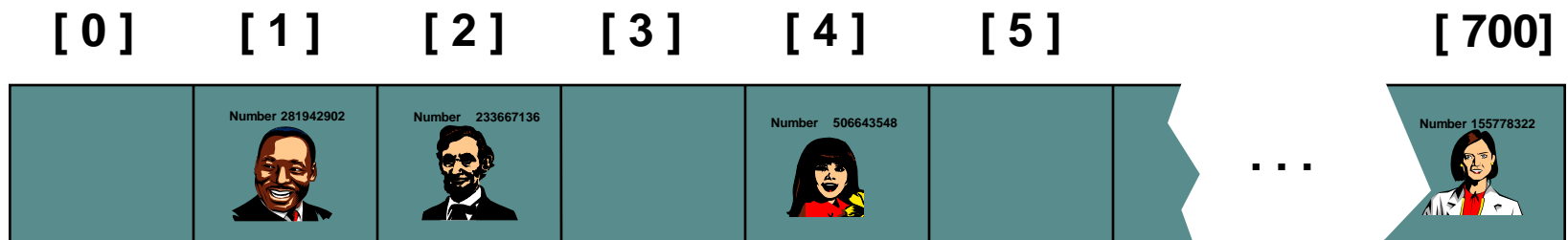
# ¿Qué es una tabla hash?

- Cuando se utiliza una tabla hash, algunos espacios contienen registros válidos y otros espacios están "vacíos".



## Hash de direcciones abiertas

- Para insertar un nuevo registro, la clave debe convertirse de alguna manera en un índice de arreglo.
- El índice se denomina valor hash de la clave.

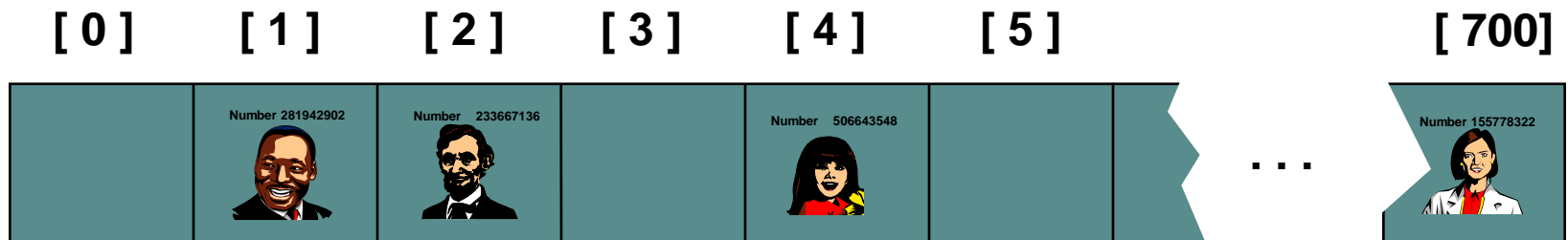


## Insertando un nuevo registro

- Forma típica de crear un valor hash:

**(Number mod 701)**

*¿Qué es  $(580625685 \% 701)$ ?*

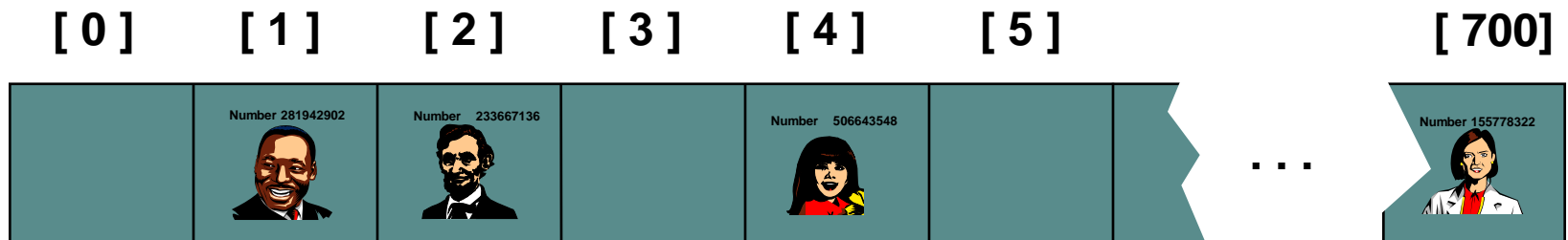


## Insertando un nuevo registro

- Forma típica de crear un valor hash:

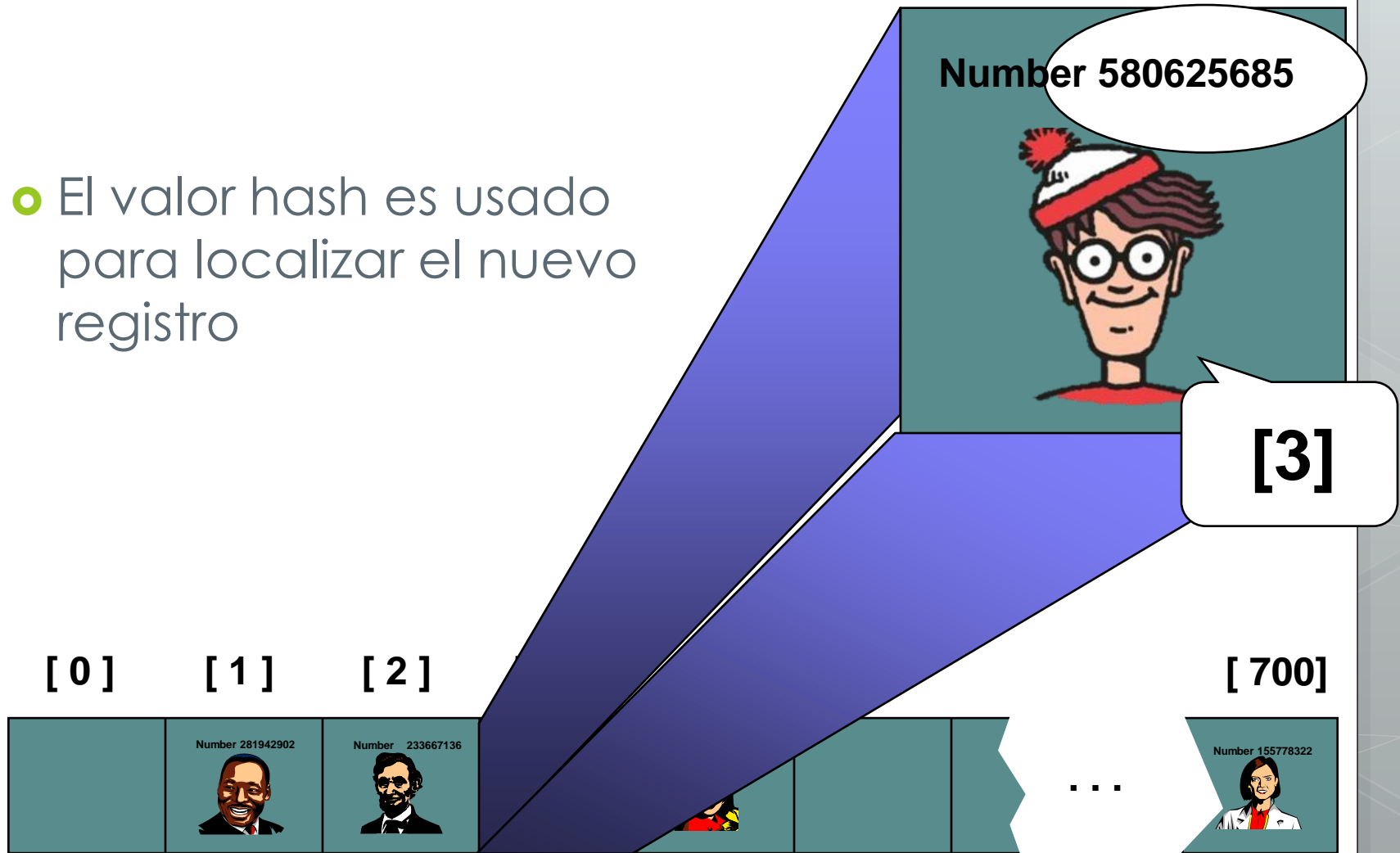
**(Number mod 701)**

*¿Qué es  $(580625685 \% 701)$ ?*



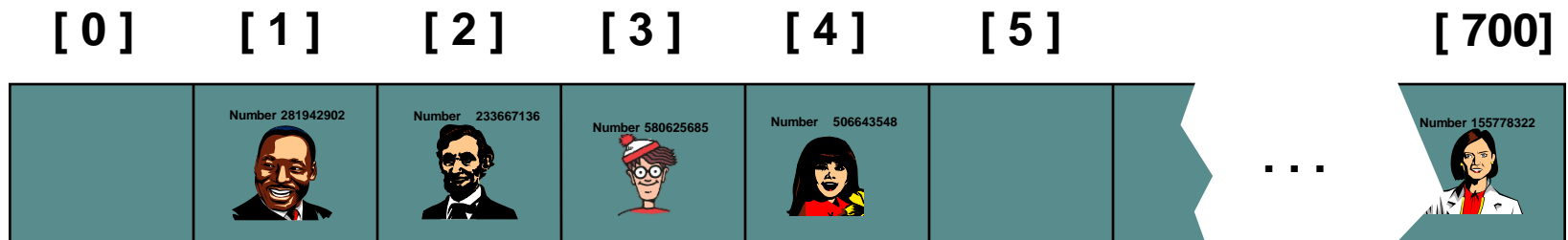


- El valor hash es usado para localizar el nuevo registro



# Insertar un nuevo registro

- El valor hash es usado para localizar el nuevo registro



# Colisiones

- Hay otro registro nuevo a insertar, con un valor hash de 2.

Number 701466868



Mi valor hash es [2].

[0]

[1]

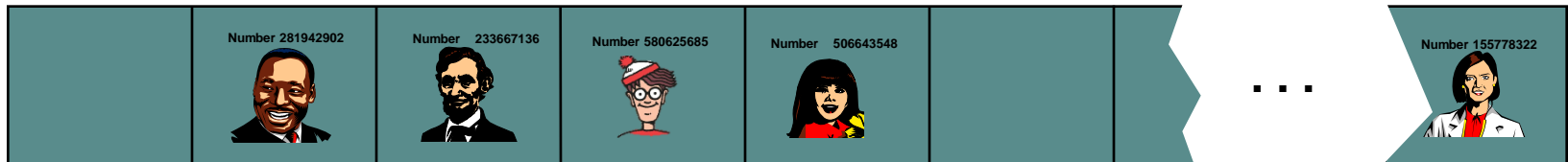
[2]

[3]

[4]

[5]

[700]



# Colisiones

- Esto se conoce como colisión, porque ya hay otro registro valido en [2].

Cuando ocurre una colisión, avanzar hasta que se encuentre un lugar vacío.

Number 701466868



[0]

[1]

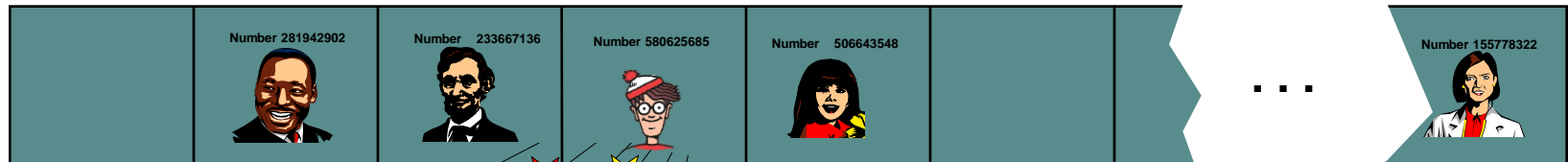
[2]

[3]

[4]

[5]

[700]



# Colisiones

- Esto se conoce como colisión, porque ya hay otro registro valido en [2].

Cuando ocurre una colisión, avanzar hasta que se encuentre un lugar vacío.

Number 701466868



[0]

[1]

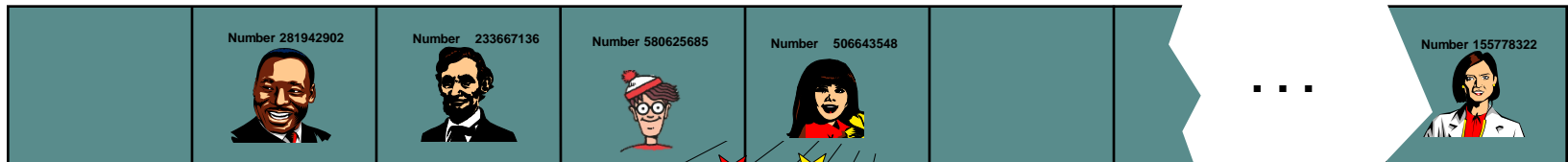
[2]

[3]

[4]

[5]

[700]



# Colisiones

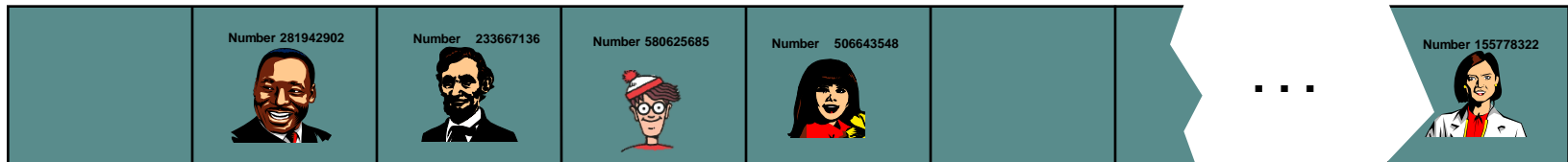
- Esto se conoce como colisión, porque ya hay otro registro valido en [2].

Cuando ocurre una colisión, avanzar hasta que se encuentre un lugar vacío.

Number 701466868



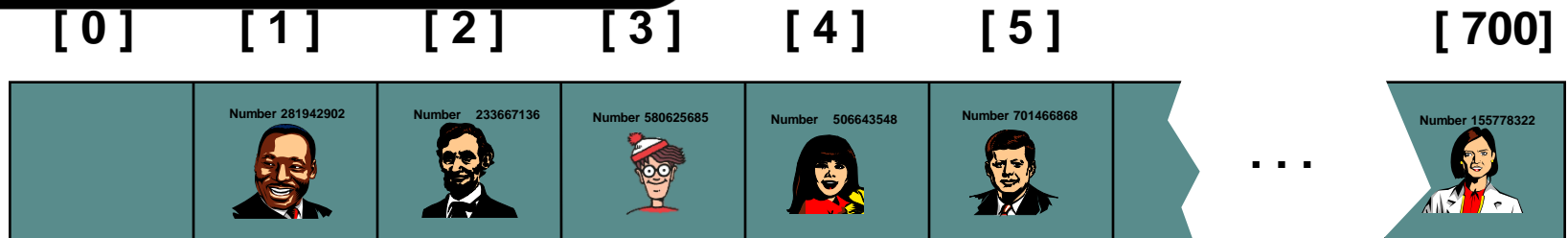
[ 0 ]      [ 1 ]      [ 2 ]      [ 3 ]      [ 4 ]      [ 5 ]      ...      [ 700 ]



# Colisiones

- Esto se conoce como colisión, porque ya hay otro registro valido en [2].







El nuevo registro se aloja en el espacio vacio.



# Búsqueda de una clave

- Los datos que se relacionan a una clave se pueden encontrar con bastante rapidez.

Number 701466868

[ 0 ]	[ 1 ]	[ 2 ]	[ 3 ]	[ 4 ]	[ 5 ]	...	[ 700 ]
	Number 281942902 	Number 233667136 	Number 580625685 	Number 506643548 	Number 701466868 		Number 155778322 

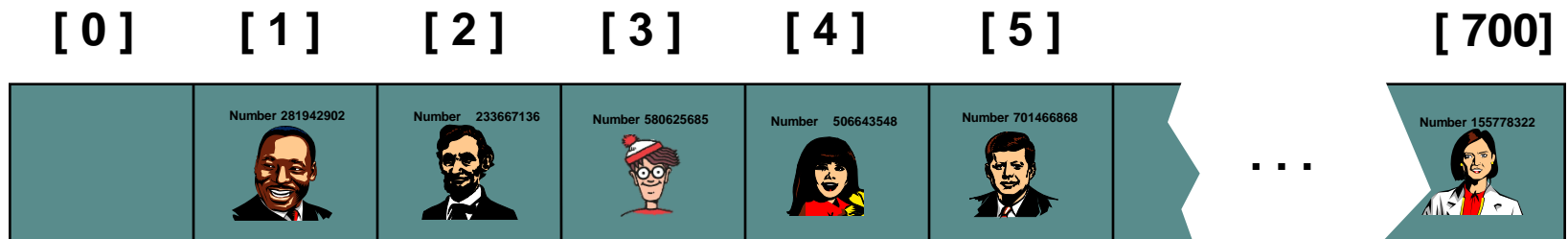


- Calcula el valor hash.
- Verifique la ubicación de la matriz para la clave.

Number 701466868

Mi valor hash is [2].

No soy yo.

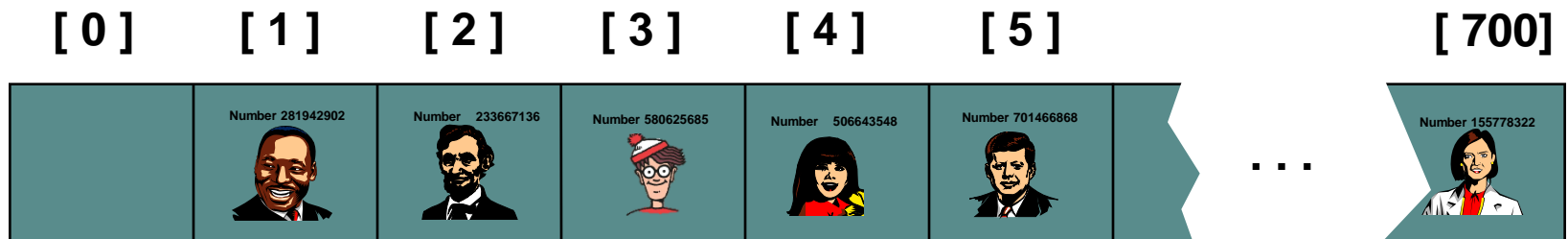


- Sigue avanzando hasta que encuentres la llave o llegues a un lugar vacío.

Number 701466868

Mi valor hash es [2].

No soy yo.



- Sigue avanzando hasta que encuentres la llave o llegues a un lugar vacío.

Number 701466868

Mi valor hash es [2].

No soy yo.

[ 0 ]

[ 1 ]

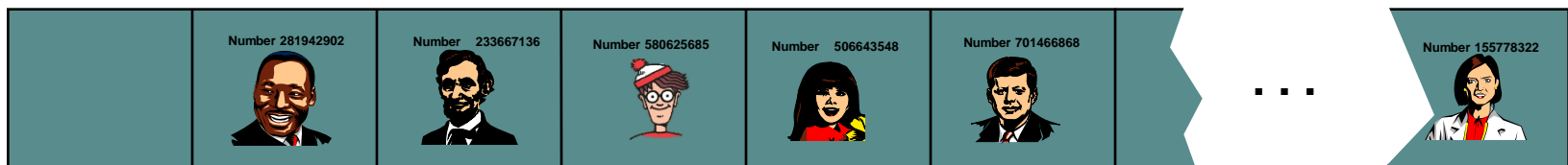
[ 2 ]

[ 3 ]

[ 4 ]

[ 5 ]

[ 700 ]



- Sigue avanzando hasta que encuentres la llave o llegues a un lugar vacío.

Number 701466868

Mi valor hash es [2].

Si!

[ 0 ]

[ 1 ]

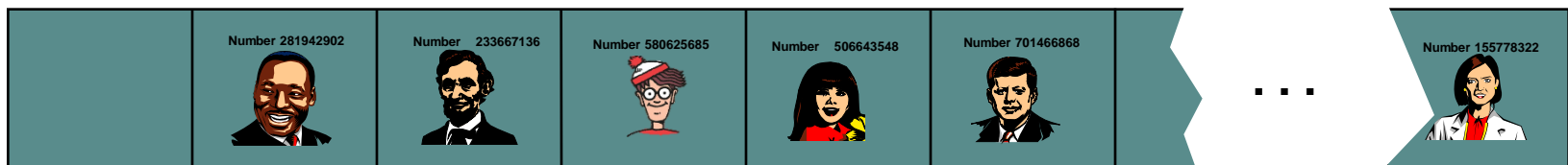
[ 2 ]

[ 3 ]

[ 4 ]

[ 5 ]

[ 700 ]



- Cuando se encuentra el elemento, la información se puede copiar en la ubicación necesaria.

Number 701466868



Mi valor hash es [2].

Si!

[ 0 ]

[ 1 ]

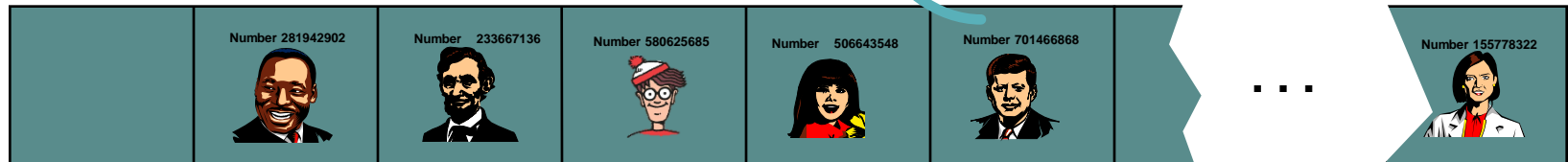
[ 2 ]

[ 3 ]

[ 4 ]

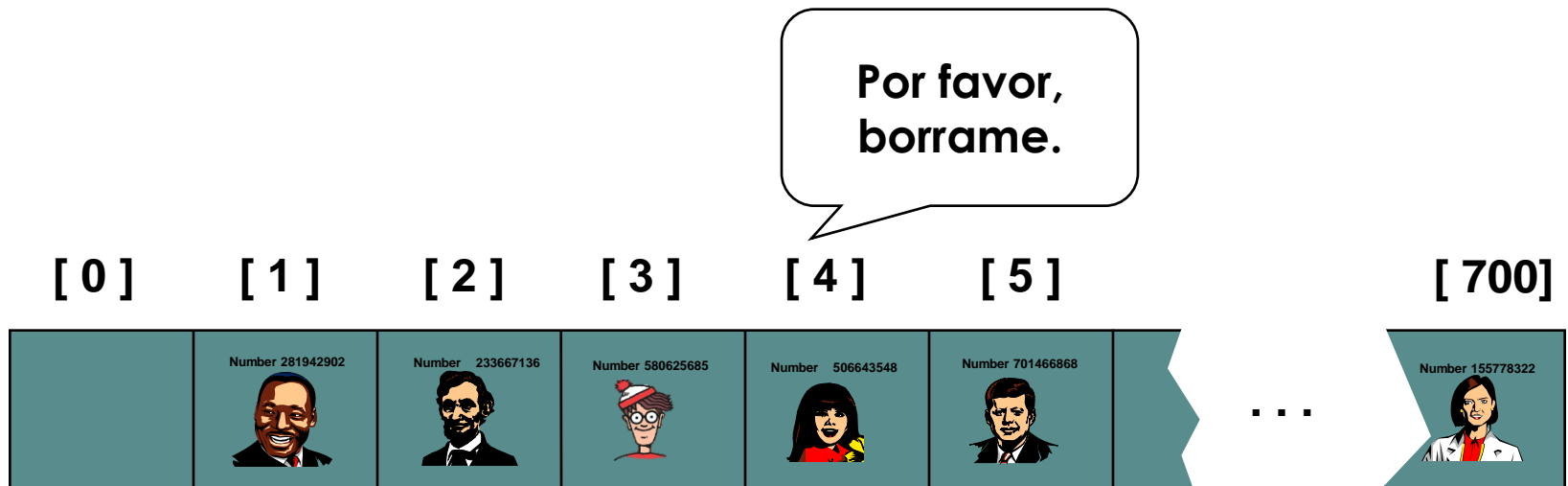
[ 5 ]

[ 700 ]



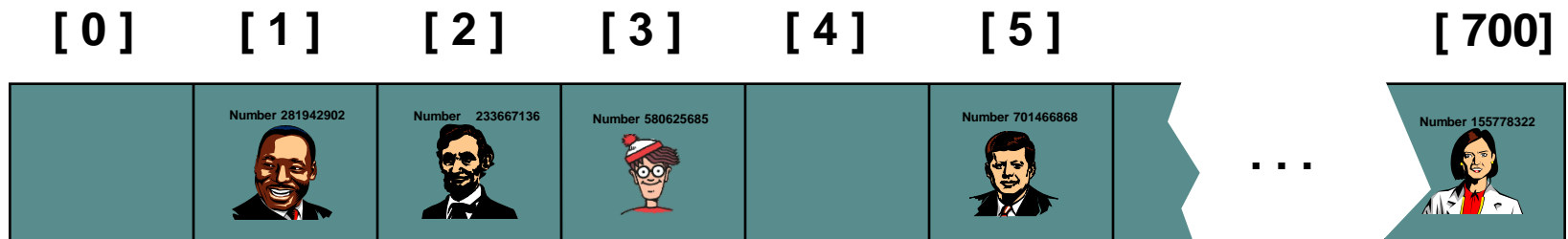
# Borrando un registro

- Los registros también se pueden eliminar de una tabla hash.



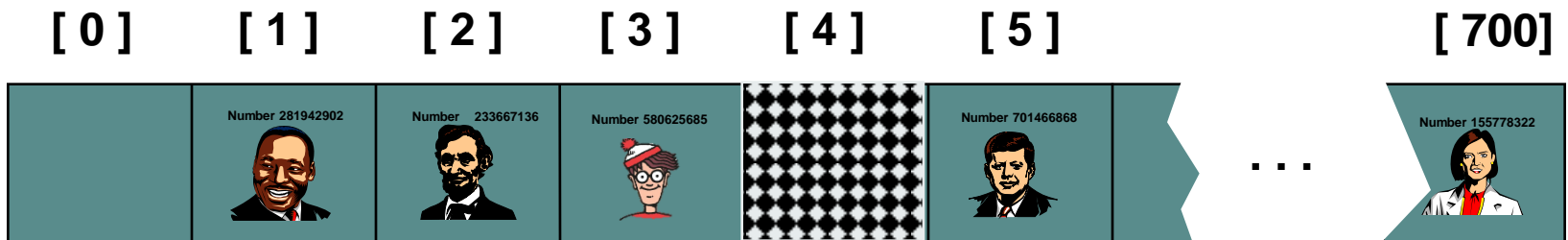
# Borrando un registro

- Los registros también se pueden eliminar de una tabla hash.
- Pero la ubicación no debe dejarse como un "lugar vacío" ordinario, ya que podría interferir con las búsquedas.



# Borrando un registro

- Los registros también se pueden eliminar de una tabla hash.
- Pero la ubicación no debe dejarse como un "lugar vacío" ordinario, ya que podría interferir con las búsquedas.
- La ubicación debe estar marcada de alguna manera especial para que una búsqueda pueda indicar que el lugar solía tener algo dentro.





# Hashing

- Las tablas hash almacenan una colección de registros con claves.
- La ubicación de un registro depende del valor hash de la clave del registro.
- Hash de dirección abierta:
  - Cuando ocurre una colisión, se usa la siguiente ubicación disponible.
  - La búsqueda de una clave en particular suele ser rápida.
  - Cuando se elimina un elemento, la ubicación debe marcarse de una manera especial, para que las búsquedas sepan que el lugar solía ser utilizado.
- Consulte el texto para la implementación.

# Hashing de dirección abierta

- Para reducir las colisiones ...
  - CAPACIDAD de la tabla = número primo de la forma  $4k + 3$
  - Funciones hash:
    - Función hash de división:
$$clave \% CAPACIDAD$$
    - Función de cuadrado medio:
$$(clave * clave) \% CAPACIDAD$$
    - Función hash multiplicativa: la clave se multiplica por una constante positiva menor que uno. La función hash devuelve los primeros dígitos del resultado fraccionario.

# Clustering

- En el método hash descrito, cuando la inserción encuentra una colisión, avanzamos en la tabla hasta que se encuentra un lugar vacante. A esto se le llama sondeo lineal.
- Problema: cuando se aplican hash a varias claves diferentes en la misma ubicación, se rellenan los puntos adyacentes de la tabla. Esto conduce al problema de la agrupación.
- A medida que la tabla se acerca a su capacidad, estos grupos tienden a fusionarse. Esto hace que la inserción tarde mucho tiempo (debido al sondeo lineal para encontrar un lugar vacante).

# Doble Hashing

- Una técnica común para evitar la agrupación se llama *doble hash*.
- Llamemos a la función hash original *hash1*
- Definir una segunda función *hash2*

## *Algoritmo de doble hash:*

1. *Cuando se inserta un elemento, use hash1 (clave) para determinar la ubicación de inserción  $i$  en el arreglo como antes.*
2. *Si ocurre una colisión, use hash2 (clave) para determinar cuánto avanzar en el arreglo en busca de un lugar vacante:*  
*$$\text{siguiente ubicación} = (i + \text{hash2}(\text{clave})) \% \text{CAPACIDAD}$$*

# Doble Hashing

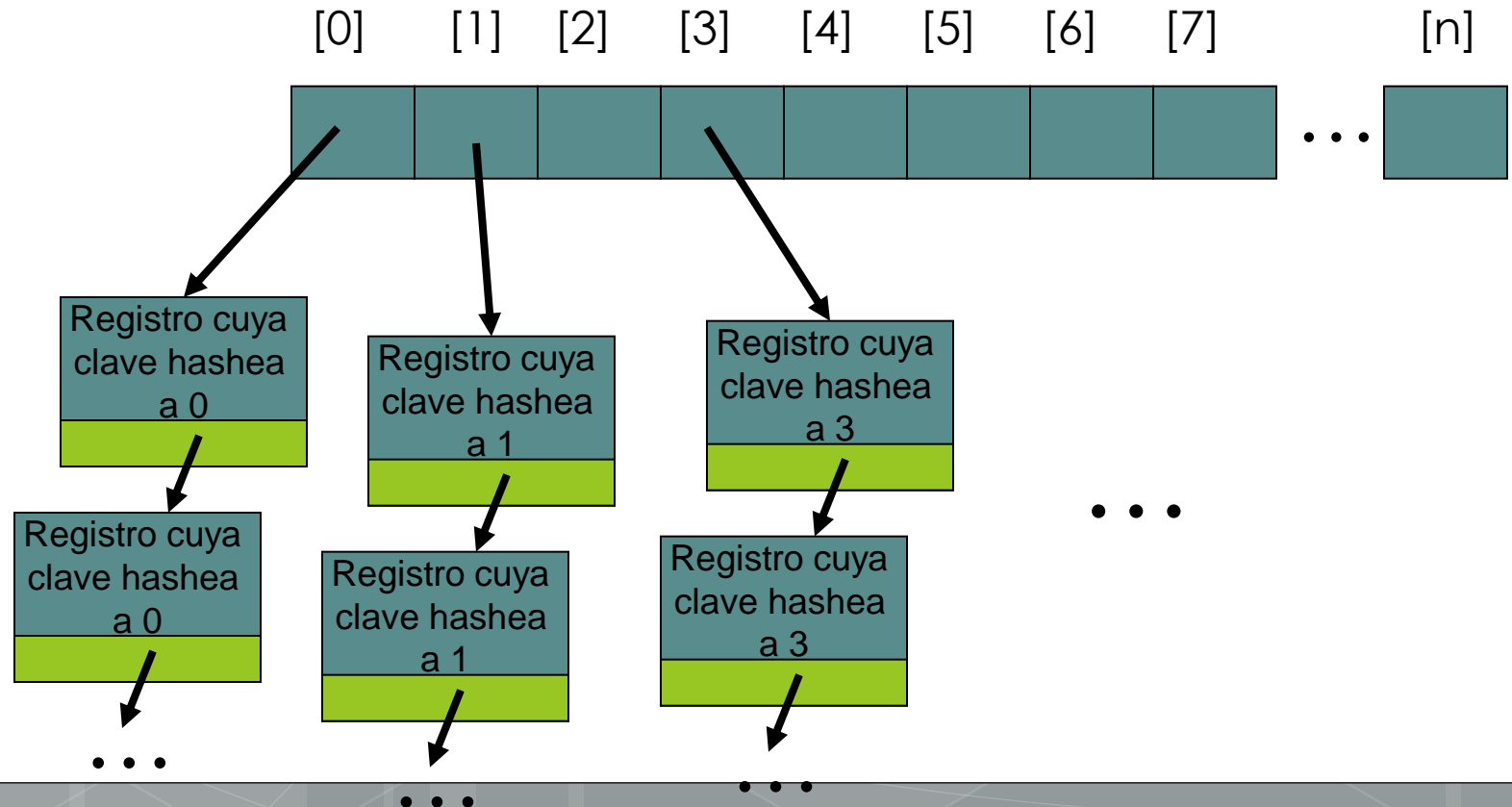
- La agrupación en clústeres tiende a reducirse, porque  $hash2( )$  tiene diferentes valores para las claves que inicialmente se asignan a la misma ubicación inicial a través de  $hash1( )$ .
- Esto contrasta con el hash de sondeo lineal.
- Ambos métodos son *hash de direcciones abiertas*, porque los métodos toman el siguiente lugar abierto en la matriz.
  - En sondeo lineal
$$hash2(clave) = (i + 1) \% CAPACIDAD$$
  - En doble hash,  $hash2( )$  puede ser una función general de la forma
$$hash2(clave) = (i + f(clave)) \% CAPACIDAD$$

# Hashing Encadenado

- En el hash de direcciones abiertas, una colisión se maneja sondeando el arreglo por el siguiente lugar vacante.
- Cuando el arreglo está lleno, no se pueden agregar elementos nuevos.
- Podemos resolver esto cambiando el tamaño del arreglo.
- Alternativa: hash encadenado.

# Hashing Encadenado

- En el hash encadenado, cada ubicación en la tabla hash contiene una lista de registros cuyas claves se asignan a esa ubicación:



# Análisis de tiempo del Hashing

- En el peor de los casos: ¡todas las claves se codifican con el mismo índice del arreglo! ¡Búsqueda  $O(n)$ !
- Afortunadamente, el caso promedio es más prometedor.
- Primero definimos una fracción llamada *factor de carga*  $\alpha$  de la tabla hash:

$$\alpha = \frac{\text{número de posiciones ocupadas}}{\text{tamaño del arreglo}}$$



## Tiempo de búsqueda promedio

Para el direccionamiento abierto con sondeo lineal, la cantidad promedio de elementos de la tabla examinados en una búsqueda exitosa es aproximadamente:

$$\frac{1}{2} \left( 1 + \frac{1}{(1 - \alpha)} \right)$$

Double hashing:  $\frac{-\ln(1-\alpha)}{\alpha}$

Chained hashing:  $1 + \frac{\alpha}{2}$

Número promedio de elementos examinados  
durante una búsqueda exitosa

Factor de carga ( $\alpha$ )	Dirección abierta, sondeo lineal $\frac{1}{2} (1+1/(1-\alpha))$	Dirección abierta doble hashing $-\ln(1-\alpha)/\alpha$	Hashing Encadenado $1+\alpha/2$
0.5	1.50	1.39	1.25
0.6	1.75	1.53	1.30
0.7	2.17	1.72	1.35
0.8	3.00	2.01	1.40
0.9	5.50	2.56	1.45
1.0	No aplica	No aplica	1.50
2.0	No aplica	No aplica	2.00
3.0	No aplica	No aplica	2.50

# Resumen

- Búsqueda Secuencial: caso promedio  $O(n)$
- Búsqueda binaria: caso promedio  $O(\log_2 n)$
- Hashing
  - Hashing de dirección abierta
    - Sondeo lineal
    - Doble hashing
  - Hashing encadenado
  - El número promedio de elementos examinados es una función del factor de carga  $\alpha$ .