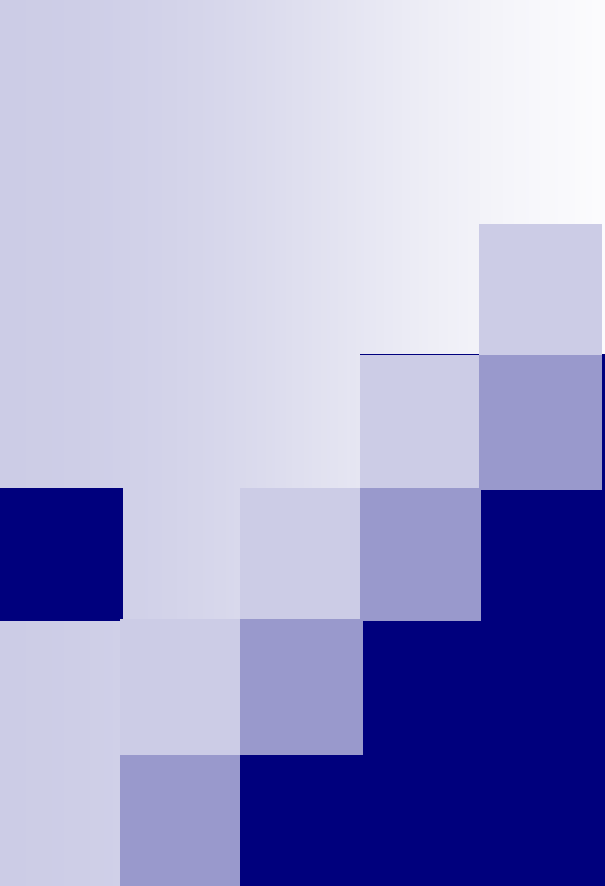
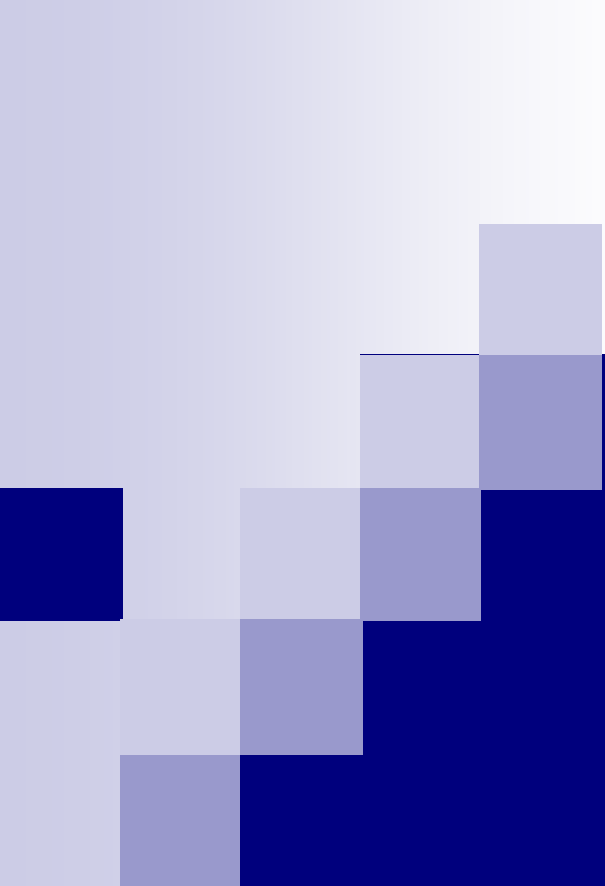




Tema 3: Algoritmos de Búsqueda



3.1 Algoritmos básicos de búsqueda

Resultados conocidos

- Búsqueda lineal

- $W_{BLin}(N) = N$ con OBcdc

- $A_{BLin}^e(N) = \sum_{i=1}^N n_{BLin}(k = T[i]) p(k == T[i]) \sim \frac{S_N}{C_N}$

- Búsqueda binaria

- $W_{BBin}(N) = \lceil \lg(N) \rceil = \lg(N) + O(1) = A_{BBin}^f(N)$

- Tenemos que calcular $A_{BBin}^e(N)$

Coste medio de BBin con éxito.

- Veamos un ejemplo:

$$T=[1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7] \quad N=7=2^3-1$$

$$A_{BBin}^e(N) = \frac{1}{7} \sum_{i=1}^7 n_{BBin}(k = T[i]) = \frac{1}{7} (1 + \underset{4}{2} + \underset{2}{2} + \underset{6}{3} + \underset{1}{3} + \underset{3}{3} + \underset{5}{3})$$

$$\Rightarrow A_{BBin}^e(N) = \frac{1}{7} (1 + 2 \cdot 2 + 3 \cdot 4) = \frac{1}{7} (1 \cdot 2^0 + 2 \cdot 2^1 + 3 \cdot 2^2)$$

- Para $N=2^k-1$ se tiene

$$A_{BBin}^e(N) = \frac{1}{N} \sum_{i=1}^k i 2^i = \frac{1}{N} [k 2^k - 2^k + 1] \Rightarrow A_{BBin}^e(N) = \frac{1}{N} [N \lg(N) - N + 1] \Rightarrow$$

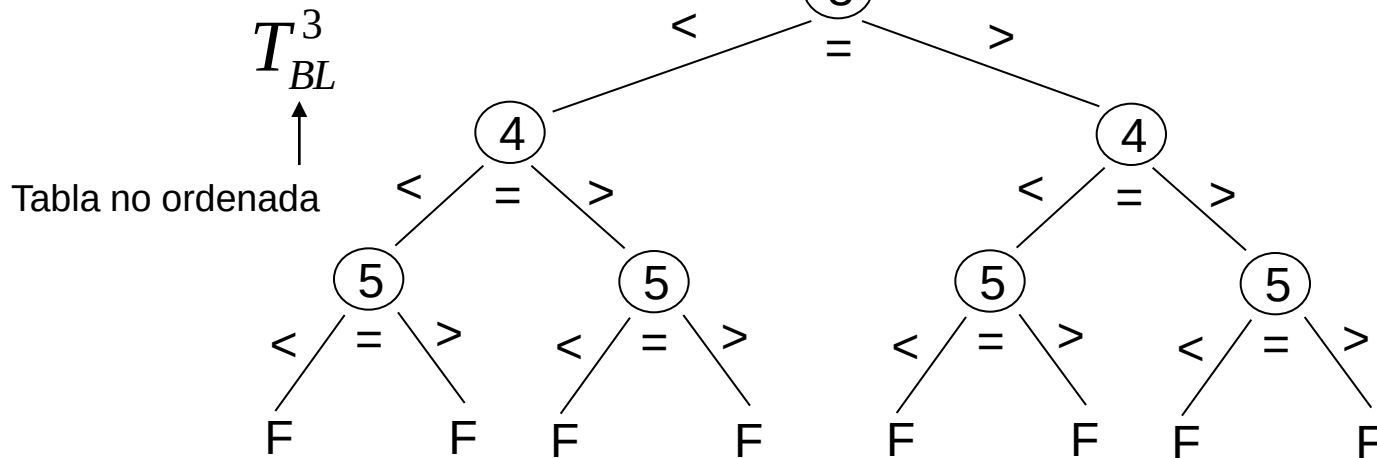
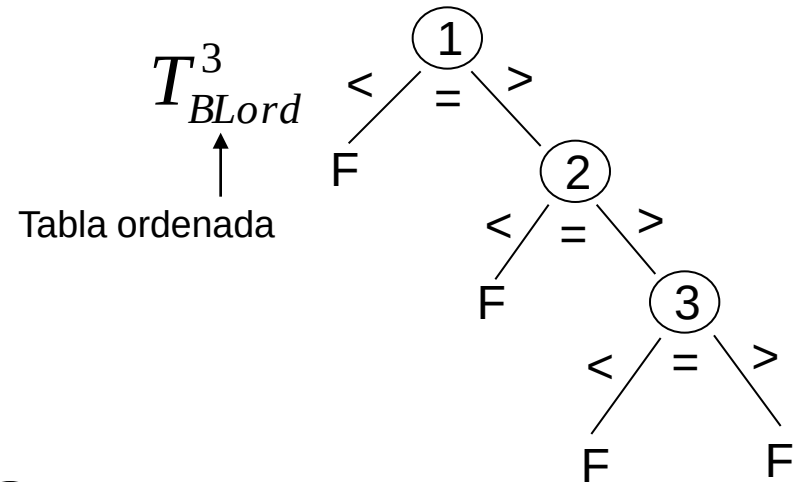
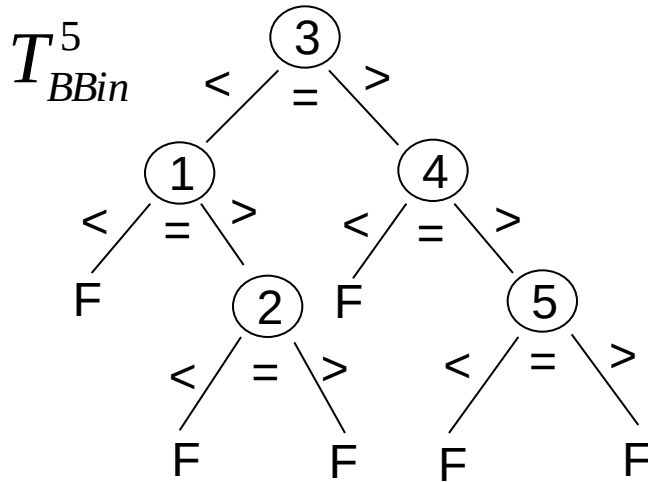
Obs: $N=2^k-1 \Rightarrow k \approx \lg(N)$

$$A_{BBin}^e(N) = \lg(N) - 1 + \frac{1}{N} \Rightarrow \boxed{A_{BBin}^e(N) = \lg(N) + O(1)}$$

Árbol de decisión para algoritmos de búsqueda por CDC: Definición

- Si **A** es un algoritmo de búsqueda por comparación de clave y **N** es un tamaño de tabla, se puede construir su **árbol de decisión** T_A^N para $\sigma \in \Sigma_N$ tal que cumple las siguientes 5 condiciones:
 1. Contiene nodos de la forma **i** que indica la cdc entre el elemento i-ésimo de la tabla y la clave **k**.
 2. Si **k** coincide con el elemento i-ésimo ($T[i] == k$) entonces la búsqueda de la clave **k** termina en el modo **i**
 3. El subárbol izquierdo del nodo **i** en T_A^N contiene el trabajo (cdcs) que realiza el algoritmo **A** si $k < i$.
 4. El subárbol derecho del nodo **i** en T_A^N contiene el trabajo (cdcs) que realiza el algoritmo **A** si $k > i$.
 5. Las hojas H_σ en T_A^N recogen la evolución de las búsquedas fallidas.

Árbol de decisión: Ejemplos



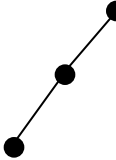
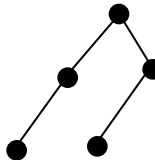
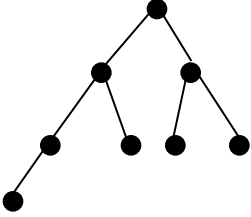
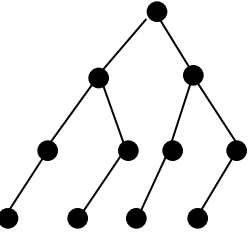
Obs: T_A^N es un árbol binario con al menos N nodos internos. Esto da la cota inferior:

$$W_A(N) \geq \text{prof}_{\min}(N)$$

Profundidad mínima de un árbol con al menos N nodos internos

Árbol de decisión: Cotas inf. Caso Peor

- Estimamos $\text{prof}_{\min}(N)$

N	T	$\text{Prof}_{\min}(N)$
1		1
2		2
3		2
4		3
7		3

Cotas inferiores en búsqueda por cdcs

- Tenemos que

$$W_A(N) \geq \text{prof}_{\min}(N) = \lfloor \lg(N) \rfloor + 1 \Rightarrow$$

$$W_A(N) = \Omega(\lg(N)) \quad \forall A \in \mathcal{B} \text{ con}$$

$$\mathcal{B} = \{A: \text{algoritmo de búsqueda por cdc}\}$$

- BBin es **óptimo** para el caso **peor**.

- Se puede demostrar también

$$A_A(N) = \Omega(\lg(N)) \quad \forall A \in \mathcal{B}$$

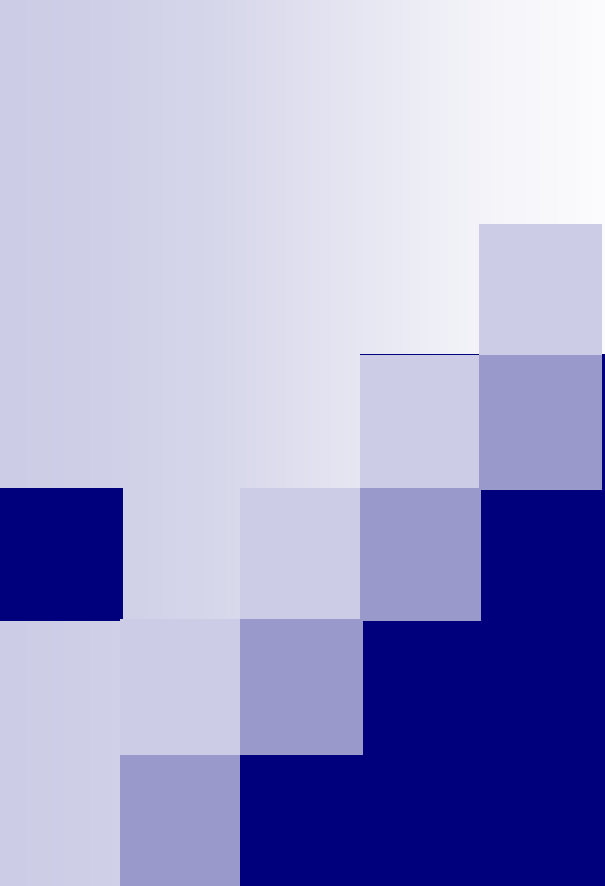
- BBin es **óptimo** para el caso **medio**.

¿Ya hemos acabado?

- **Observación:** la búsqueda no es una operación aislada
- Los elementos no sólo se buscan sino que también se **insertan** o se **borran**
- No sólo importa cómo se busca sino también dónde se busca
- Contexto: TAD **Diccionario**

En esta sección hemos ...

- Recordado los costes peor y medio de búsqueda lineal y binaria
- Aprendido el concepto de árbol de decisión en búsqueda por cdcs
- Aprendido a construir árboles de decisión en búsqueda por cdcs
- Visto que la búsqueda binaria es óptima en los casos peor y medio dentro de los algoritmos de búsqueda por cdcs



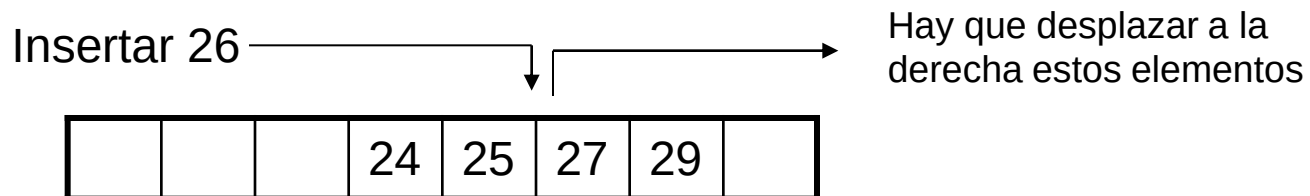
3.2 Búsqueda sobre diccionarios

TAD Diccionario

- **Diccionario:** conjunto ordenado de datos con las primitivas.
 - pos Buscar(clave **k**, dicc **D**)
 - Devuelve la posición de la clave **k** en el diccionario **D** o un código de error **ERR** si **k** no está en **D**.
 - status Insertar (clave **k**, dicc **D**)
 - Inserta la clave **k** en el diccionario **D**, devuelve **OK** o **ERR** si **k** no se pudo incorporar a **D**.
 - void Borrar (clave **k**, dicc **D**)
 - Elimina la clave **k** en el diccionario **D**

EdEs para Diccionarios I

- ¿Qué EdE es la más adecuada para un diccionario?
- Opción 1: Tabla ordenada ($|D|=N$)
 - **Buscar:** Usamos BBin $\Rightarrow n_{\text{Buscar}}(k,D)=O(\log(N)) \Rightarrow$ óptimo.
 - **Insertar:** Hay que mantener la tabla ordenada $\Rightarrow$ la inserción es costosa
 - Ejemplo:



- Si se inserta en la posición 1, hay que desplazar N elementos
- En el caso medio se desplazan $N/2$ elementos
- Por tanto $n_{\text{Insertar}}(k,D)=\Theta(N)$: **malo!!**

EdEs para Diccionarios II

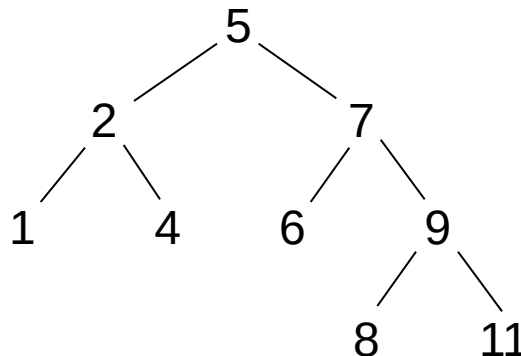
- Opción 2: **Árbol binario de búsqueda** (ABdB)
- **Definición:** Un **ABdB** es un árbol binario **T** que para todo nodo $T' \in T$ se cumple:

$$\text{Info}(T'') < \text{Info}(T') < \text{Info}(T''')$$

para cualquier nodo T'' a la izq de T' y T''' a la derecha de T'

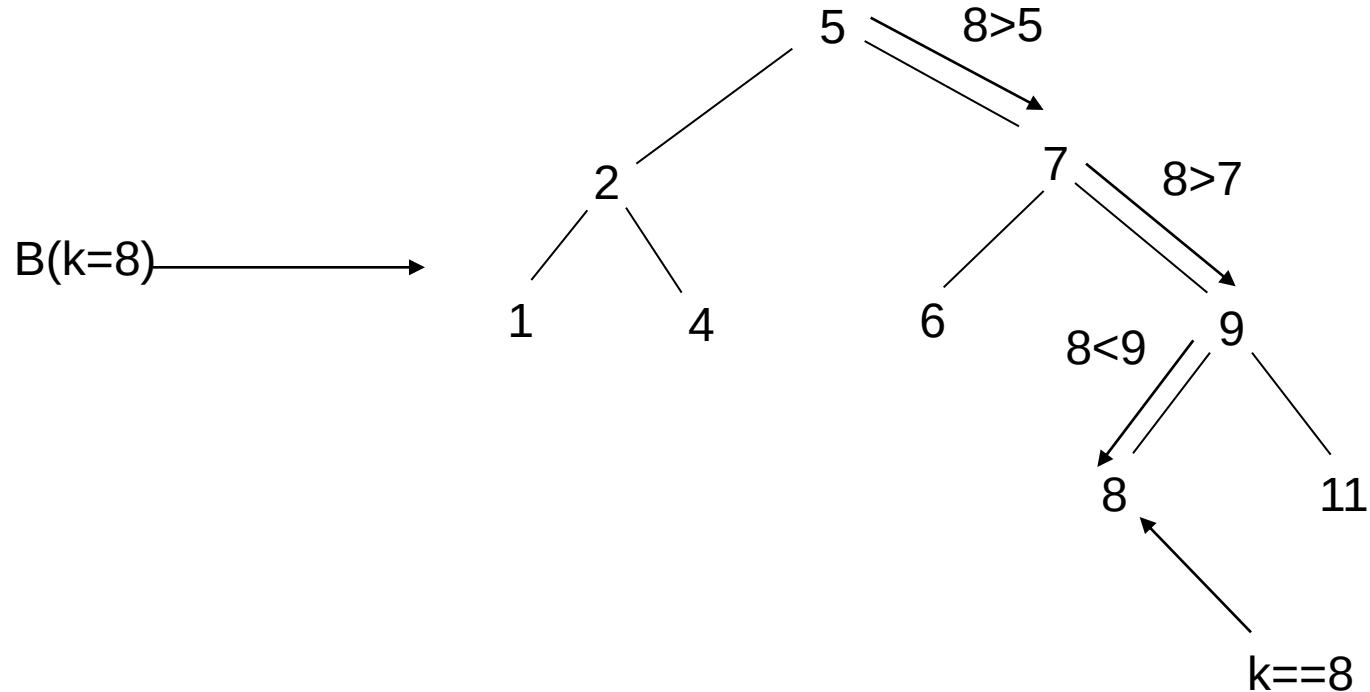
- Es decir, todos los nodos a la izquierda de T' tienen un valor menor que $\text{info}(T')$ y todos los nodos a la derecha de T' tienen un valor mayor que $\text{info}(T')$

Ejemplo:



Buscar sobre ABdBs I

■ Ejemplo



Buscar sobre ABdBs II

- Pseudocódigo:

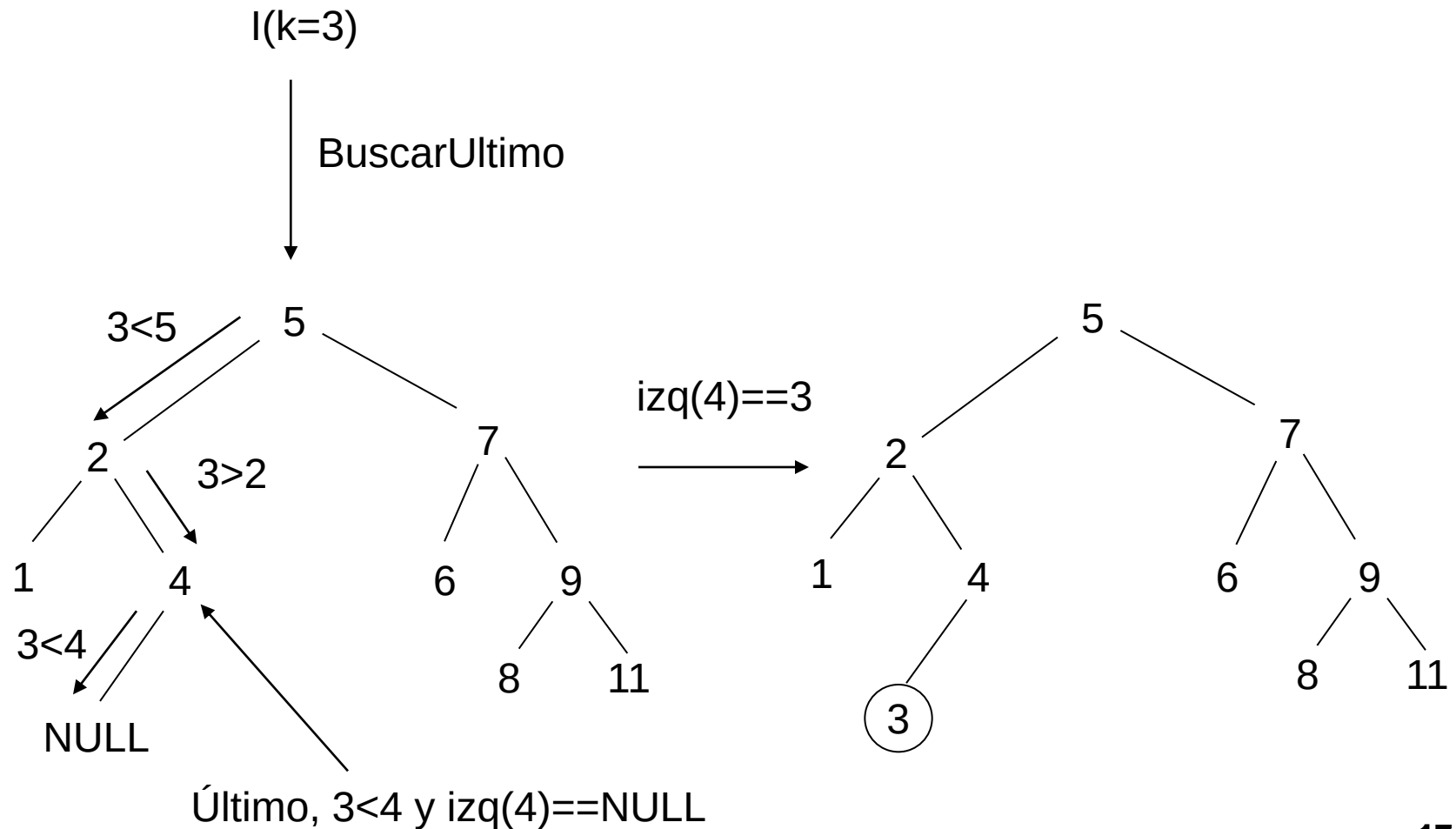
```
AB Buscar (clave k, AB T)  
  si T==NULL : return NULL;  
  si info(T)==k : return T;  
  si k<info(T) :  
    return Buscar(k,izq(T));  
  si k>info(T) :  
    return Buscar(k,der(T));
```

- Observación:

$$n_{\text{Buscar}}(k,T)=\text{prof}(T)$$

Insertar en ABDBs I

■ Ejemplo



Insertar en ABDBs II

■ Pseudocódigo

status Insertar (clave k, AB T)

```
T'=BuscarUltimo(k,T);  
T''=GetNodo();  
si T''==NULL : return ERR;  
info(T'')=k;  
si k<info(T') :  
    izq(T')=T''  
else :  
    der(T')=T'';  
return OK;
```

AB BuscarUltimo(clave k, AB T)

```
si k == info(T): return NULL;  
si (k<info(T) y izq(T) ==NULL) o  
   (k>info(T) y der(T) ==NULL):  
    return T;  
si k<info(T) y izq(T) !=NULL :  
    return BuscarUltimo(k,izq(T));  
si k>info(T) y der(T) !=NULL :  
    return BuscarUltimo(k,der(T));  
return T ;
```

■ Observación

$$n_{\text{Insertar}}(k, T) = n_{\text{BuscarUltimo}}(k, T) + 1 \Rightarrow n_{\text{Insertar}}(k, T) = O(\text{prof}(T))$$

Borrar en ABdBs

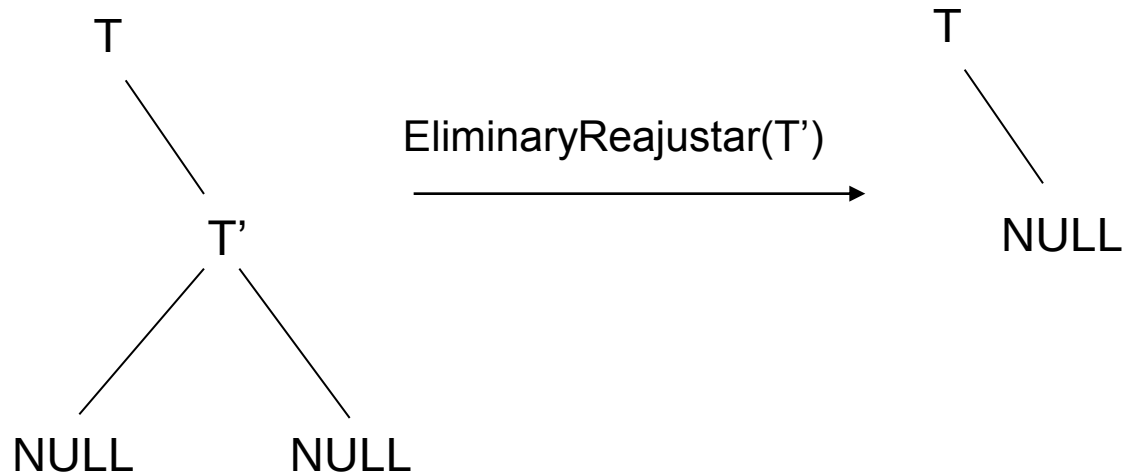
- Pseudocódigo:

```
void Borrar (clave k, AB T)  
  T'=Buscar(k,T);  
  si T'!=NULL :  
    EliminaryReajustar(T',T);
```

- En **EliminaryReajustar** hay tres posibles casos, dependiendo del número de hijos que tenga el nodo T' a eliminar

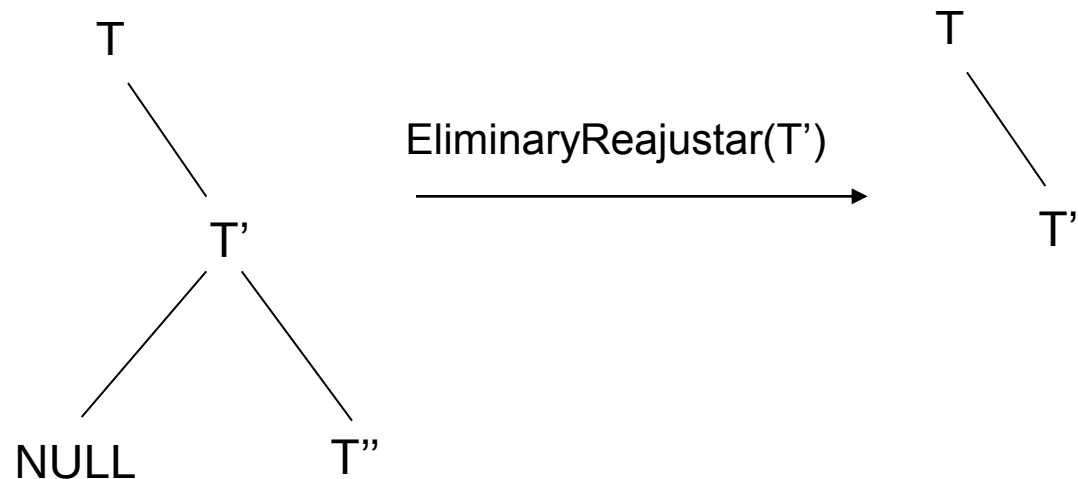
Eliminar y Reajustar I

- **Caso 1: El nodo a eliminar **no** tiene hijos**
 - se libera el nodo T' ($\text{free}(T')$), y
 - el puntero del padre de T' que apuntaba a T' se reasigna a NULL.
- **Coste EliminaryReajustar = $O(1)$**



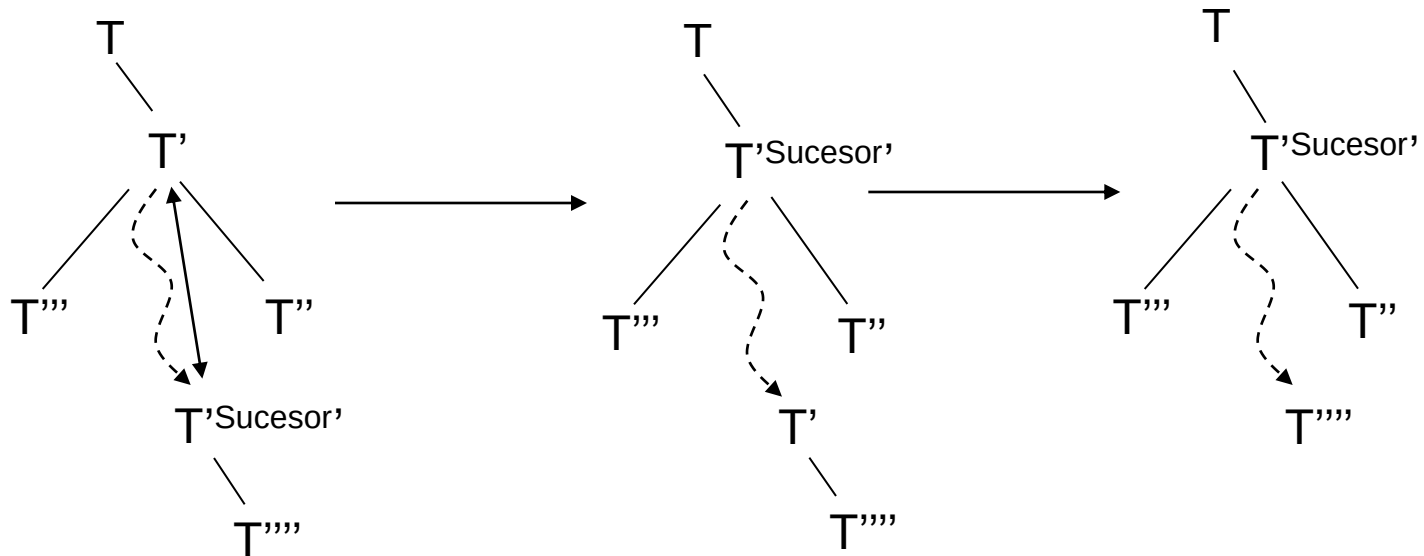
Eliminar y Reajustar II

- Caso 2: El nodo a eliminar tiene **sólo un hijo**
 - el puntero del padre de T' que apuntaba a T' se hace apuntar al único hijo de T' y
 - se libera T'
- Coste `EliminaryReajustar` = $O(1)$



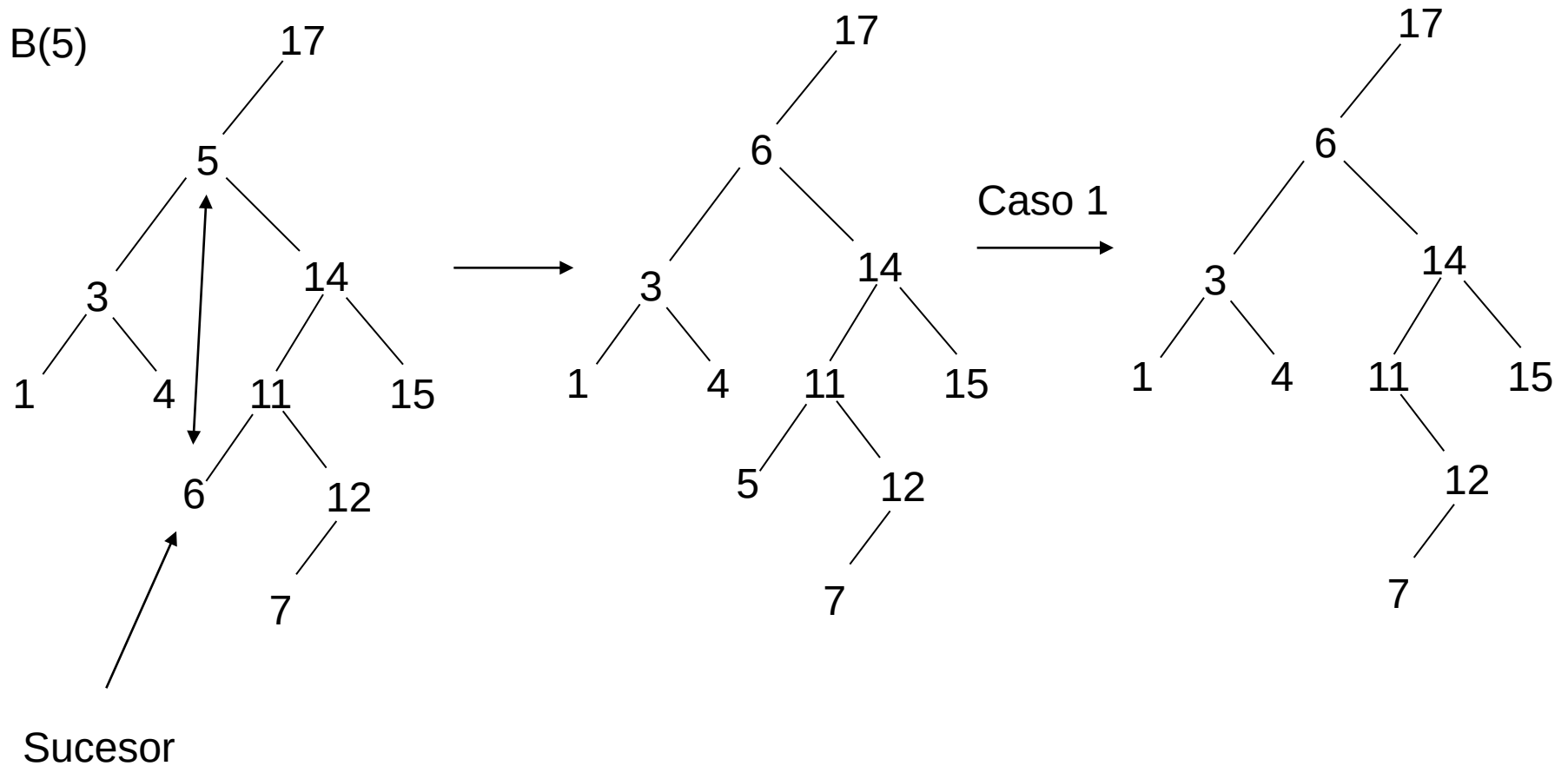
Eliminar y Reajustar III

- Caso 3: El nodo a eliminar tiene **dos hijos**
 - T' se sustituye por el nodo que contiene al **sucesor** (el elemento siguiente en la tabla ordenada), y
 - se elimina el nodo T' según el caso 1 o 2.
- Coste EliminaryReajustar $\leq \text{prof}(T)$



Eliminar y Reajustar III

■ Ejemplo



Búsqueda del Sucesor

■ Pseudocódigo

AB BuscarSucesor(AB T')

T''=der(T');

mientras izq(T'')!=NULL :

T''=izq(T'');

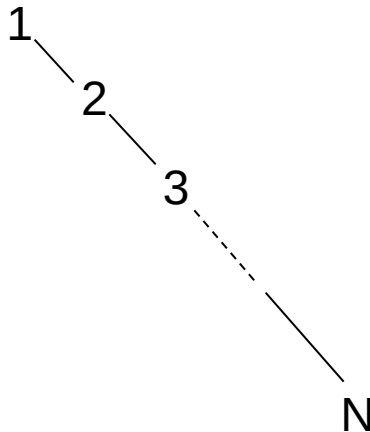
return T'' ;

■ **Obs:** Si k' es el sucesor en un ABdB de k , entonces **izq(k')==NULL**:

- Si $\text{izq}(k')=k''$ se tendría que $k'' < k'$
- Pero $k'' > k$, pues k'' está a la derecha de k
- Luego se tiene que $k < k'' < k'$ y
- Por tanto, k' **no puede ser el sucesor** de k .

Primitivas de diccionarios con TAD ABdB

- $n_{\text{EyR}}(T', T) = n_{\text{BuscarSucesor}} + n_{\text{ReajustarPunteros}} = O(\text{prof}(T)) + O(1)$
- Por tanto
$$n_{\text{Borrar}}(k, T) = O(\text{prof}(T)) + \underbrace{O(\text{prof}(T))}_{\text{Buscar}} = \underbrace{O(\text{prof}(T))}_{\text{EyR}}$$
- ABdB es eficaz siempre que $\text{prof}(T) = \Theta(\lg(N))$
- Pero $W_{\text{Buscar}}(N) = N$: ¡¡**malo!!**



Coste medio de búsqueda en ABdBs I

- $A_{\text{Buscar}}^e(N)$ = coste medio de **(1)** la búsqueda de todos los elementos **(2)** para todos los T_σ

$$\begin{aligned} A_{\text{Buscar}}^e(N) &= \frac{1}{N!} \sum_{\sigma \in \Sigma_N} A_{\text{Buscar}}^e(T_\sigma) = \frac{1}{N!} \sum_{\sigma \in \Sigma_N} \frac{1}{N} \sum_{i=1}^N n_{\text{Buscar}}(\sigma(i), T_\sigma) = \\ &= \frac{1}{N!} \sum_{\sigma \in \Sigma_N} \frac{1}{N} \sum_{i=1}^N [\text{prof}(\sigma(i)) + 1] = \frac{1}{N!} \sum_{\sigma \in \Sigma_N} \left[1 + \frac{1}{N} \sum_{i=1}^N \text{prof}(\sigma(i)) \right] = \\ &= 1 + \frac{1}{N} \left(\frac{1}{N!} \sum_{\sigma \in \Sigma_N} n_{\text{Crear}}(T_\sigma) \right) \end{aligned}$$

Por tanto

$$A_{\text{Buscar}}^e(N) = 1 + \frac{1}{N} A_{\text{Crear}}^e(N)$$

Coste medio de búsqueda en ABdBs II

■ Vemos a un pseudocódigo **alternativo** para Crear

AB Crear (tabla σ)

$T = \text{IniAB}(\sigma);$

$\text{InsAB}(\sigma(1), T);$

$\text{Repartir}(\sigma, \sigma_i, \sigma_d);$

$T_i = \text{Crear}(\sigma_i);$

$T_d = \text{Crear}(\sigma_d);$

$\text{izq}(T) = T_i;$

$\text{der}(T) = T_d;$

return $T;$

Caso similar a QS:

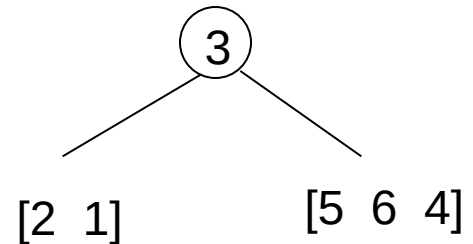
$$n_C(\sigma) = N - 1 + n_C(\sigma_i) + n_C(\sigma_d)$$



$$A_{\text{Crear}}(N) = 2N \log(N) + O(N)$$

$\sigma = [3 \ 5 \ 2 \ 6 \ 4 \ 1]$

Crear



Por tanto:

$$A_{\text{Buscar}}^e(N) = 1 + \frac{1}{N} A_{\text{Crear}}^e(N) = 1 + \frac{1}{N} [2N \log(N) + O(N)] = \Theta(\log(N))$$

Resumen de primitivas sobre ABdBs

- Si B es algoritmo de Búsqueda por cdc

$$W_B(N) = \Omega(N)$$

- Si la EdD es un ABdB las primitivas son eficaces **en promedio**
- Si aseguramos que para todo $\sigma \in \Sigma_N$ se tiene un ABdB tal que
$$\text{prof}(T_\sigma) = \Theta(\lg(N)),$$
tendríamos que

$$W_{\text{Buscar}}(N) = \Theta(\lg(N))$$



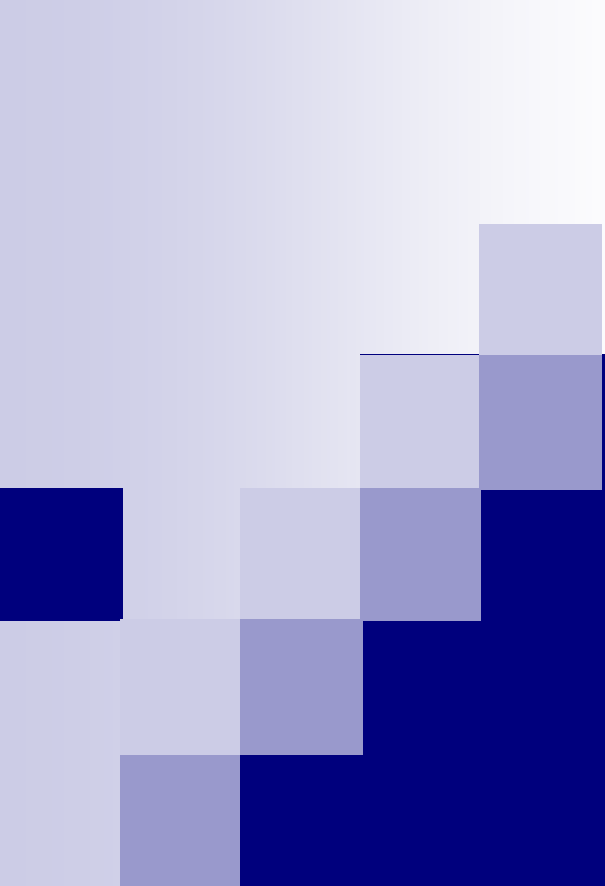
En esta sección hemos ...

- Introducido el concepto de diccionario y sus primitivas
- Estudiado su implementación sobre ABdBs
- Comprobado que su coste viene determinado por la profundidad del ABdB
- Comprobado que dicha implementación es óptima en el caso medio
- Comprobado que en el caso peor dicha implementación tiene un coste $\Theta(\lg(N))$



Herramientas y técnicas a trabajar

- La construcción y uso de Árboles Binarios de Búsqueda
- Eliminación de nodos en ABdBs
- Problemas a resolver (al menos)
110, 112, 114, 115



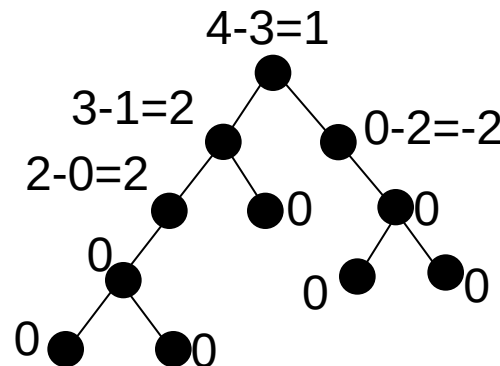
3.3 Árboles AVL

Árboles AVL (Adelson-Velskii-Landis)

- **Definición:** El factor de equilibrio de un nodo T en un ABdB se define

$$FE(T) = \text{prof}(T_i) - \text{prof}(T_d)$$

- **Ejemplo:**



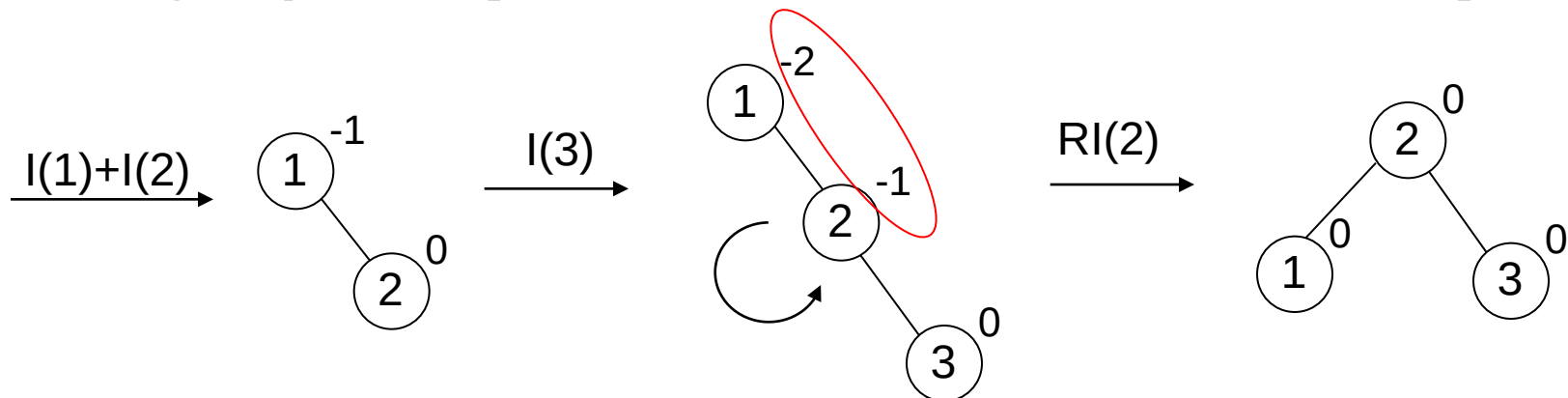
- **Definición:** Un AVL T es un ABdB tal que $\forall$ subárbol T' de T se verifica

$$FE(T') = \{-1, 0, 1\}$$

Construcción de AVLs

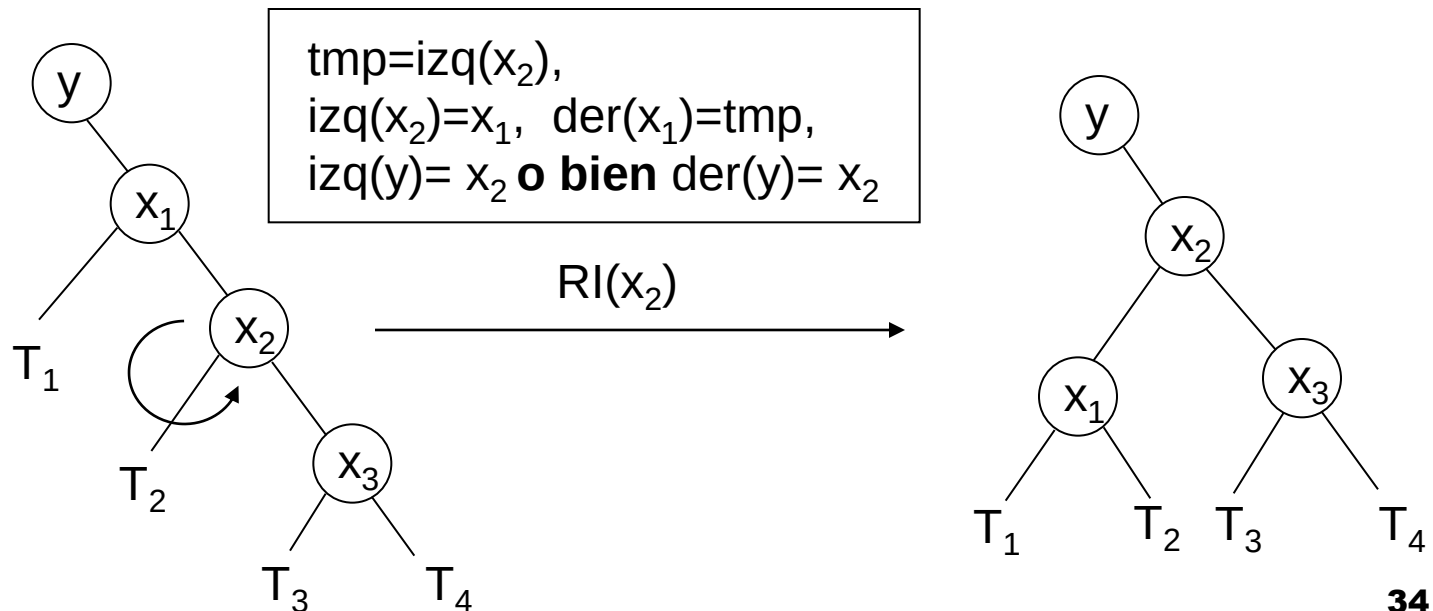
- Para construir un AVL se procede en 2 pasos.
 - Paso 1: Se realiza la inserción **normal** de un nodo en un ABdB
 - Paso 2: Si es necesario se corrigen desequilibrios y una vez corregidos, se vuelve al paso 1

Ejemplo: $T=[1\ 2\ 3\ 4\ 5\ 6\ 7\ 15\ 14\ 13\ 12\ 11\ 10\ 9\ 8]$



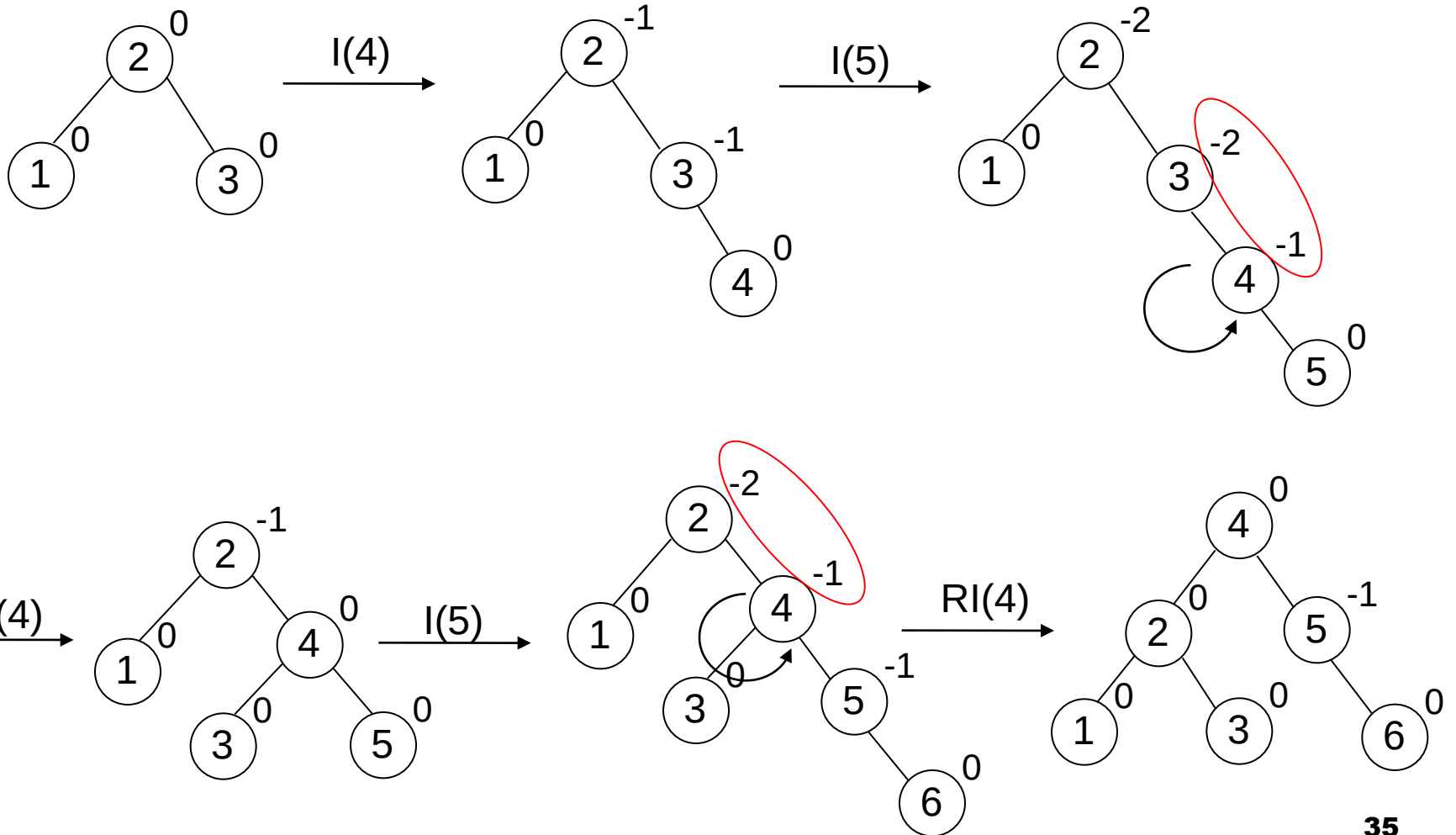
Construcción de AVLs

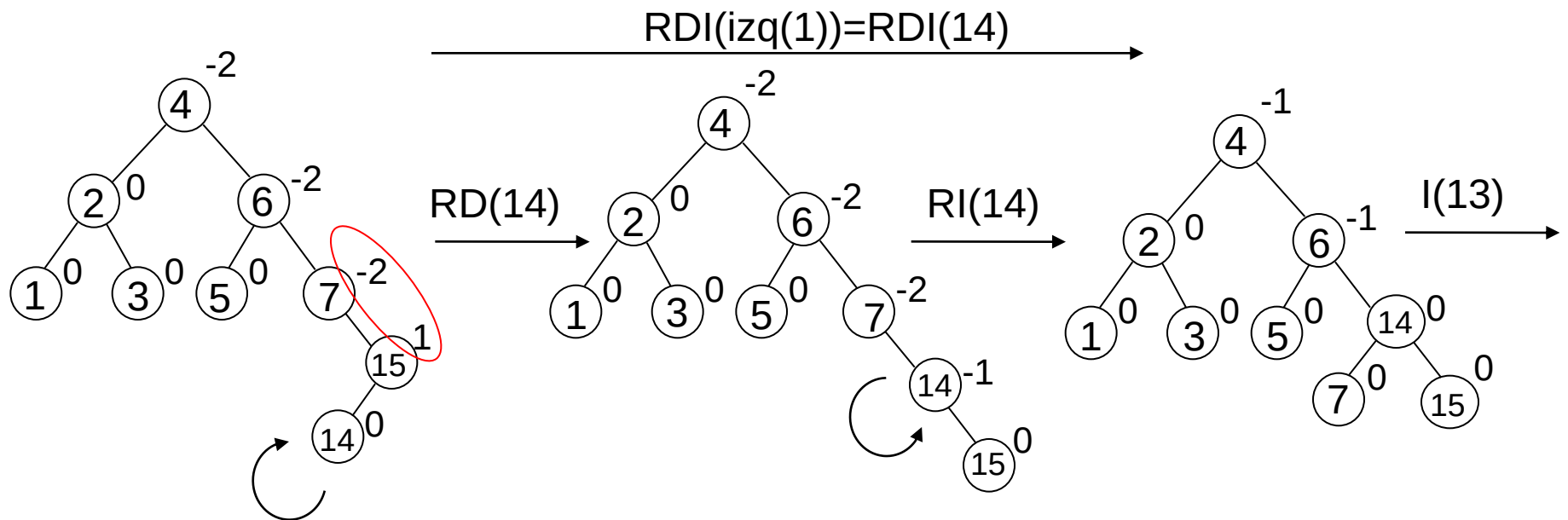
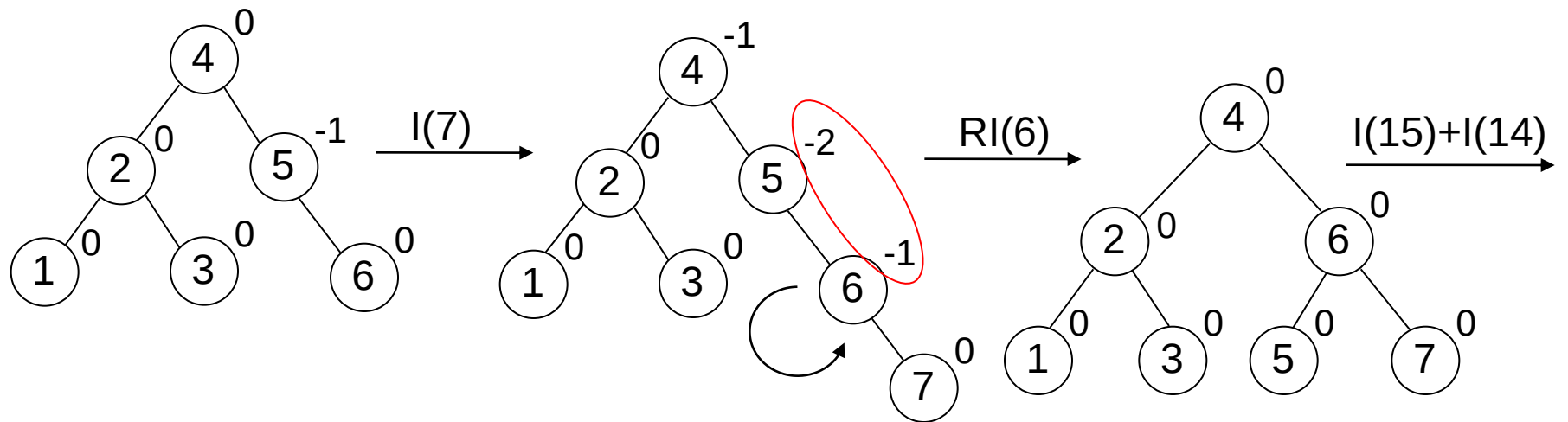
- La operación que acabamos de hacer se denomina **Rotación a la Izquierda en el -1**, en este caso en el elemento 2.
- En realidad la rotación a la izquierda en el -1 corresponde a la siguiente reasignación de punteros

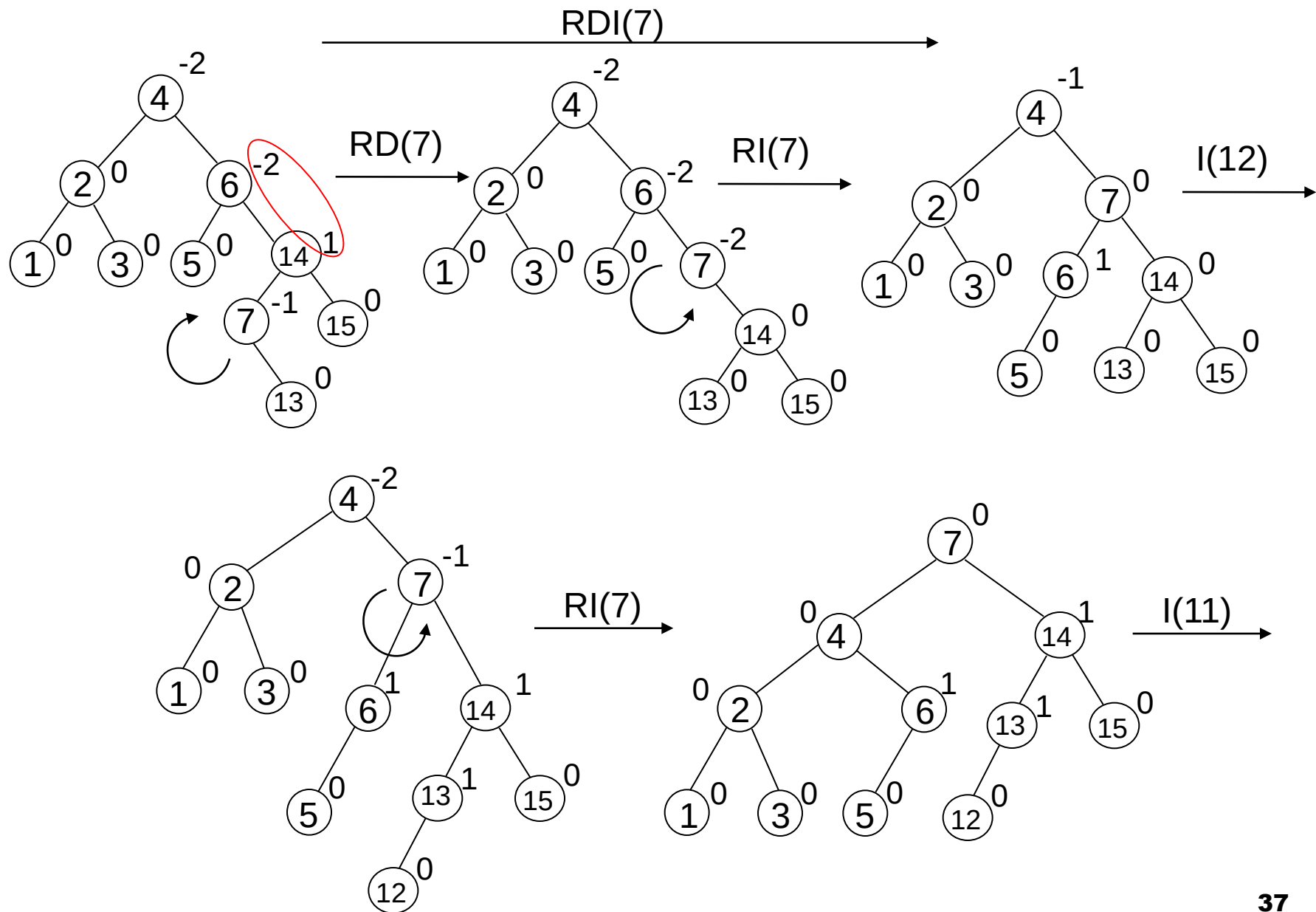


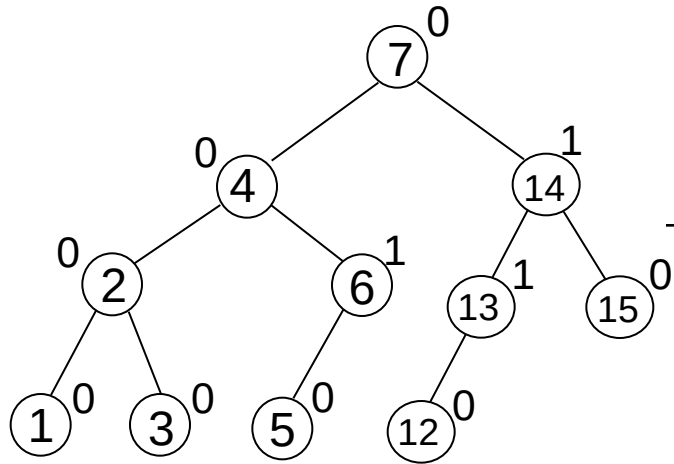
Construcción de AVLs

■ Seguimos el proceso

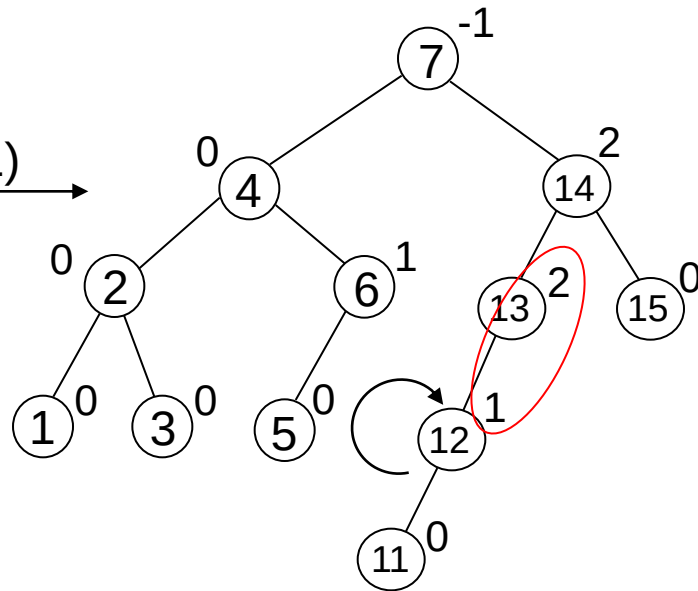




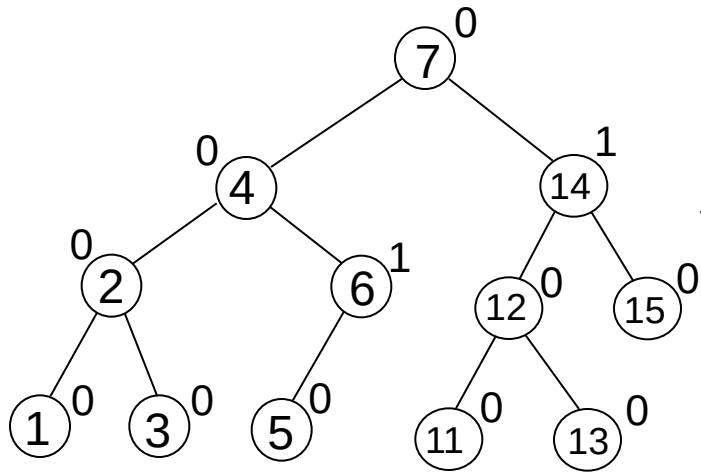




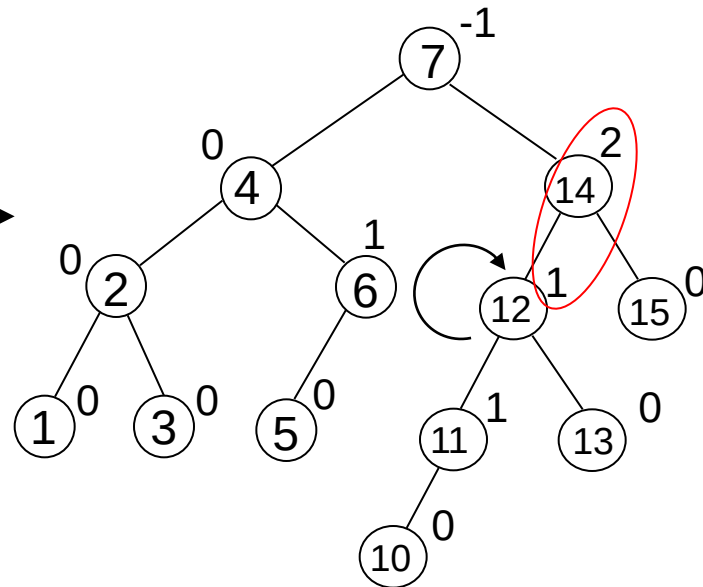
$I(11)$



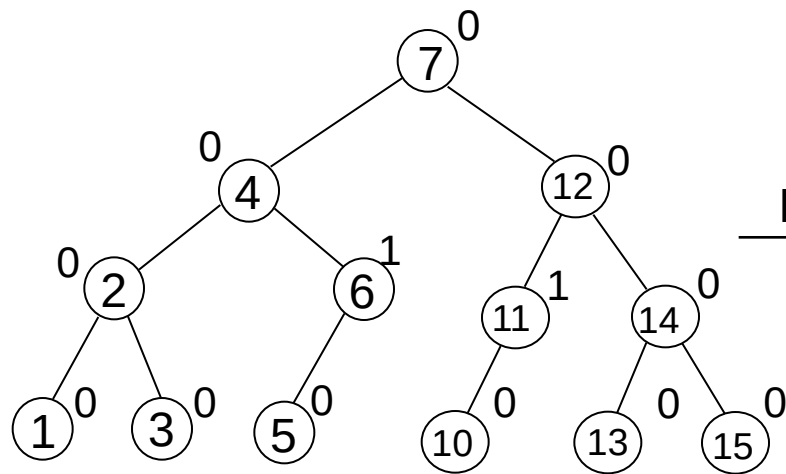
$RD(12)$



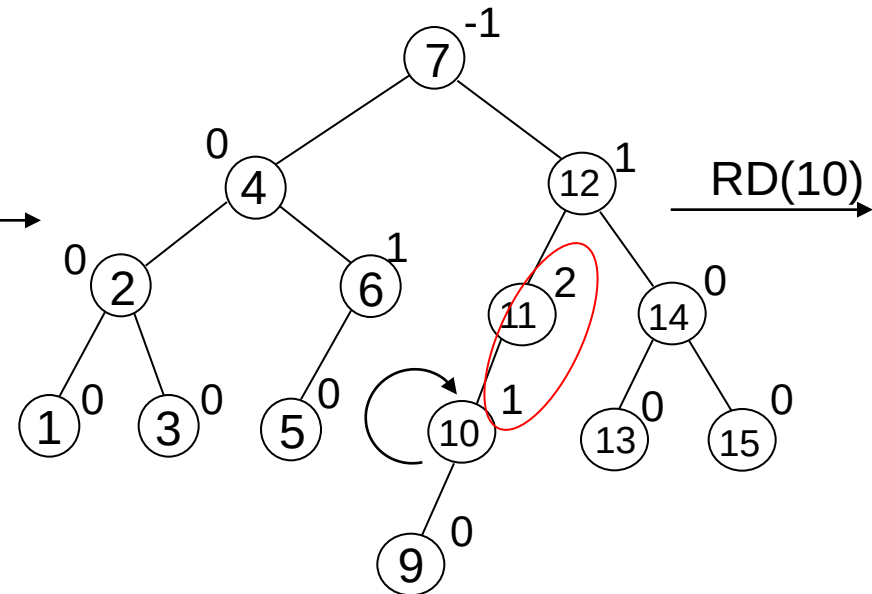
$I(10)$



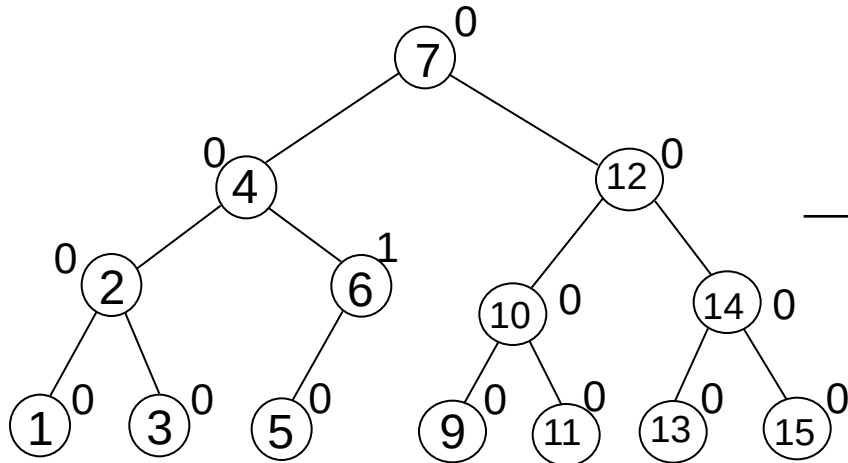
$RD(12)$



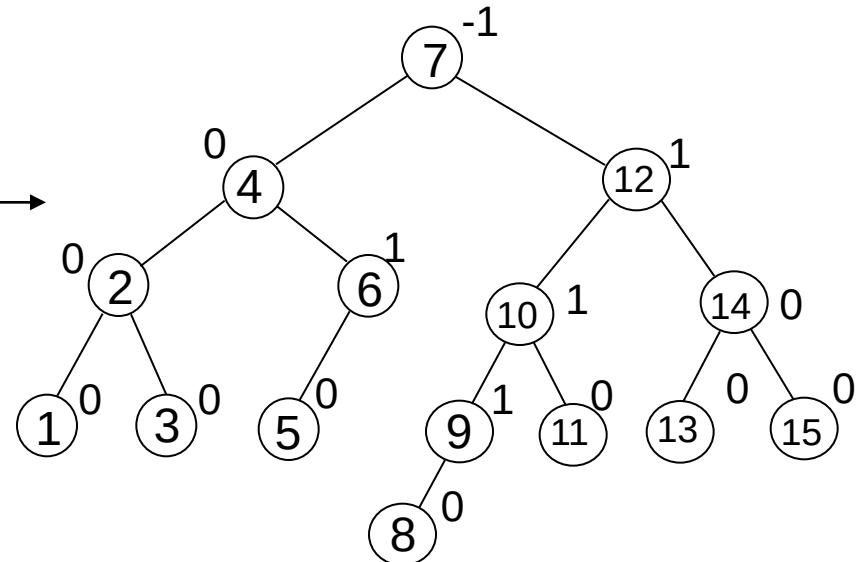
$I(9)$



$RD(10)$



$I(8)$



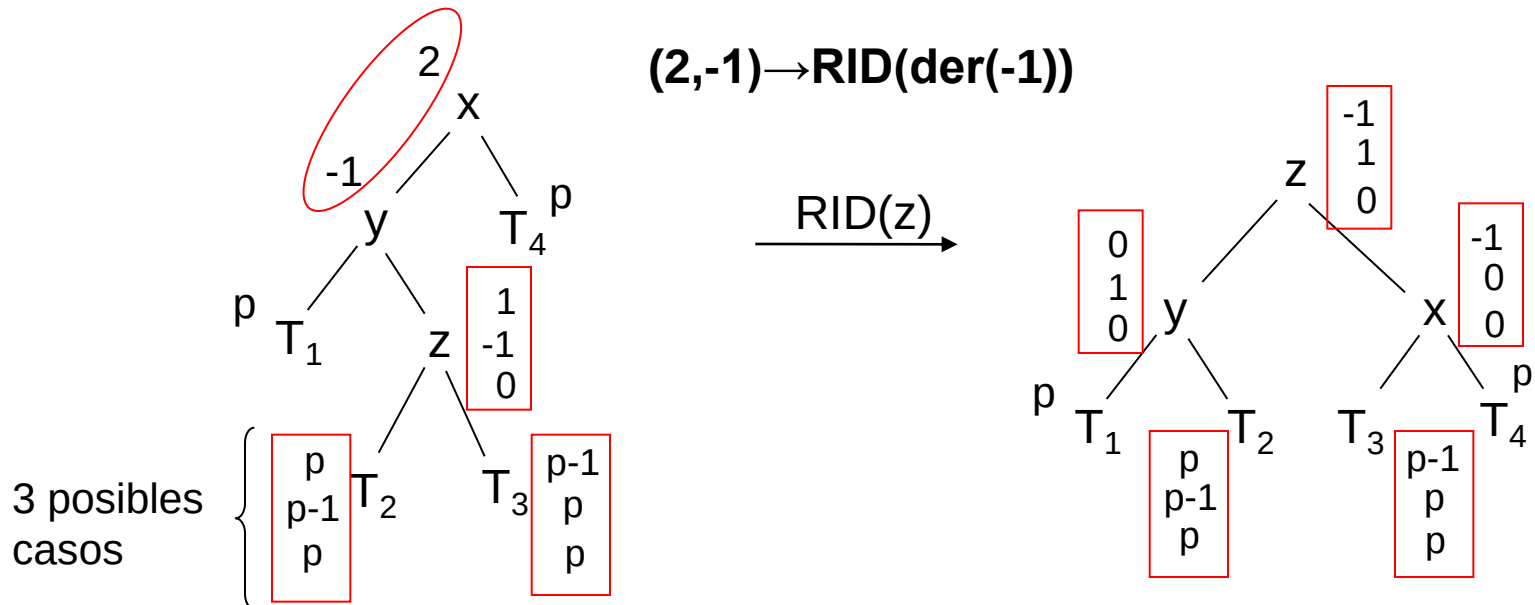
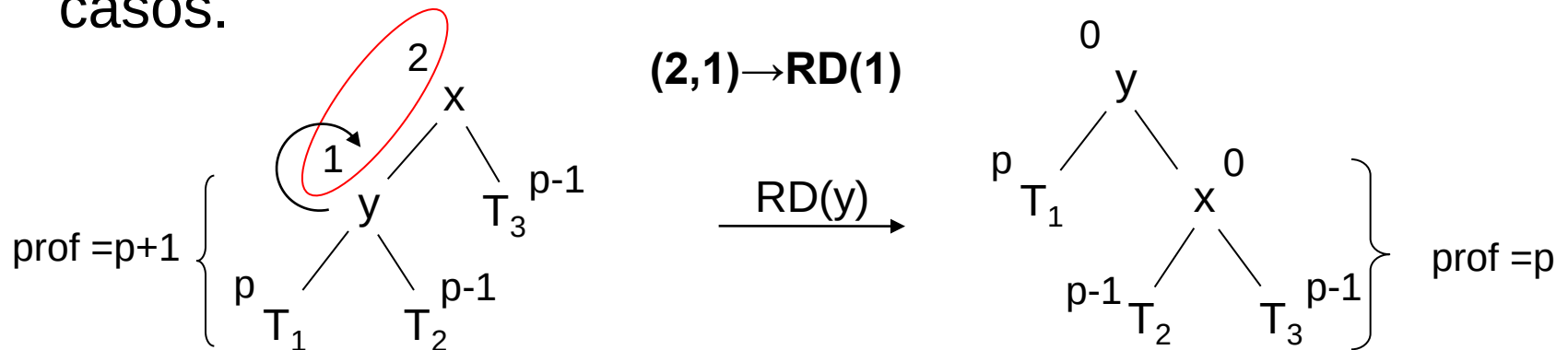
Hemos construido el AVL

Resumen de rotaciones

Tipo de desequilibrio	Rotación
$(-2,-1)$	Rot Izquierda en -1 (hijo izq de -1 pasa a hijo der de -2)
$(2,1)$	Rot Derecha en 1 (hijo der de 1 pasa a hijo izq de 2)
$(-2,1)$	Rot Derecha Izquierda en izquierda de 1 $\text{RotIzq}(\text{izq}(1)) + \text{RotDer}(\text{izq}(1))$
$(2,-1)$	Rot Izquierda Derecha en derecha en -1 $\text{RotDer}(\text{der}(-1)) + \text{RotIzq}(\text{der}(-1))$

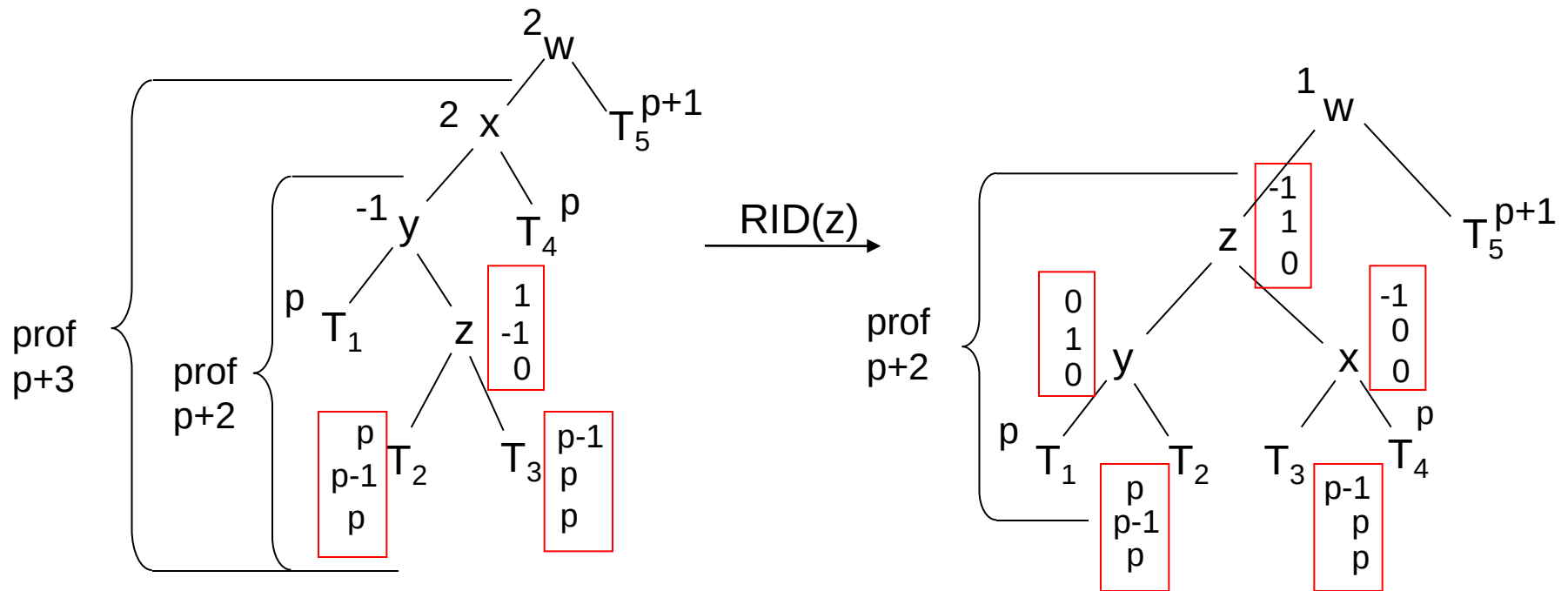
Funcionamiento de las rotaciones

- Para ver que efectivamente las rotaciones solucionan los desequilibrios es necesario ver cada uno de los casos.



Funcionamiento de las rotaciones

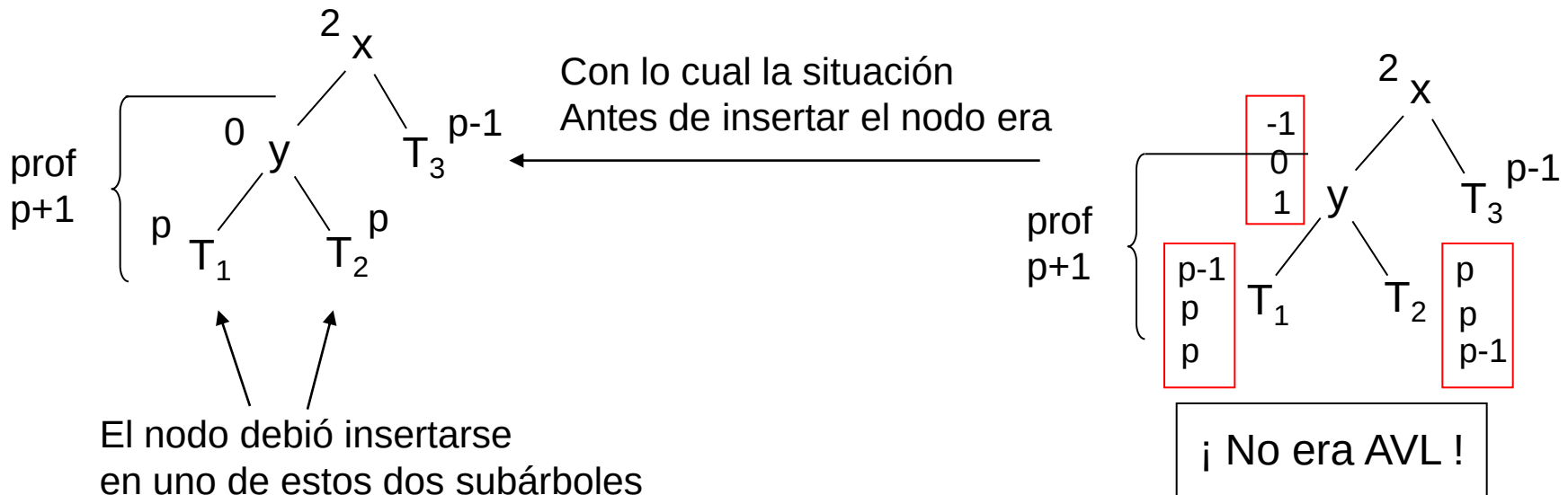
- **Observación:** Las rotaciones resuelven desequilibrios de tipo ± 2 , situados mas arriba del desequilibrio ($\pm 2, \pm 1$)



Funcionamiento de las rotaciones

- **Observación:** Si se tiene un AVL, tras la inserción de un elemento no pueden darse desequilibrios de la forma $(\pm 2, 0)$.

Supongamos que tras una inserción tenemos



Profundidad de Árboles AVL

- **Proposición:** Si T es un AVL con N nodos entonces

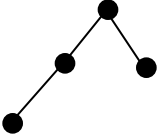
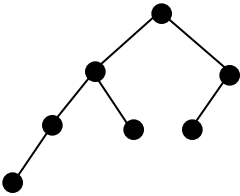
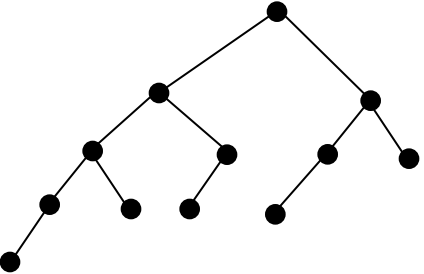
$$\text{prof}(T) = O(\log(N))$$

- Dado que para cualquier árbol binario con n nodos se tiene que $\text{prof}(T) = \Omega(\log(N))$, tenemos que si T es AVL entonces

$$\text{prof}(T) = \Theta(\log(N))$$

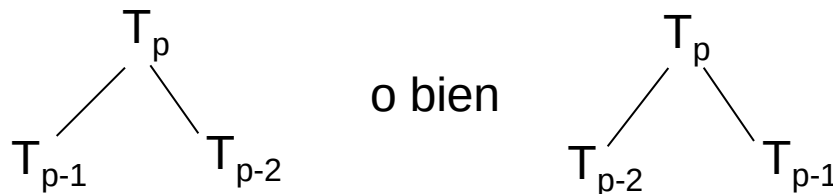
- Para ver lo anterior vamos a estimar el número mínimo de nodos n_p de un AVL T_p de profundidad p .

AVLs mínimos

p	AVL	n_p	n_{p+1}	F_{p+2}
0		1	2	F_2
1		2	3	F_3
2		4	5	F_4
3		7	8	F_5
4		12	13	F_6
...				

AVL de Fibonacci I

- F_p es el p-ésimo número de Fibonacci.
- Los números de Fibonacci verifican:
 - $F_n = F_{n-1} + F_{n-2}$, con $F_0 = F_1 = 1$
- Los AVL T_p se construyen



- $n_p = 1 + n_{p-1} + n_{p-2}$, por tanto se tiene

$$\underbrace{1 + n_p}_{H_p} = \underbrace{1 + n_{p-1}}_{H_{p-1}} + \underbrace{1 + n_{p-2}}_{H_{p-2}}$$

Obs:

$$\begin{aligned}
 H_0 &= 1 + n_0 = 2 = F_2, \\
 H_1 &= 1 + n_1 = 3 = F_3
 \end{aligned}$$

- Por tanto $n_p + 1 = H_p = F_{p+2}$

AVL de Fibonacci II

- Se puede demostrar que en N-simo número de Fibonacci es

$$F_N = \frac{1}{\sqrt{5}} \left(\underbrace{\Phi^{N+1}}_{\downarrow N \rightarrow \infty \atop \infty} - \underbrace{\Psi^{N+1}}_{\downarrow N \rightarrow \infty \atop 0} \right) \text{ donde } \Phi = \frac{1+\sqrt{5}}{2} \text{ y } \Psi = \frac{1-\sqrt{5}}{2}$$

ya que $\Phi > 1$ y $|\Psi| < 1$

- Con lo cual se tiene $F_N \approx (1/\sqrt{5})\Phi^N$ y como $n_p = F_{p+2} - 1$ obtenemos

$$n_p \approx \frac{\Phi^3}{\sqrt{5}} \Phi^p = C\Phi^p,$$

donde p es la profundidad y C una constante

Profundidad de un AVL II

- Entonces si T es un AVL con N nodos y profundidad p , se sigue que

$$N \geq n_p \approx C\Phi^p$$

- Esto es, se tiene que

$$\lg(N) \geq p \cdot \lg(\Phi) + \lg(C),$$

es decir $\log(N) = \Omega(\text{prof}(T))$, y por lo tanto

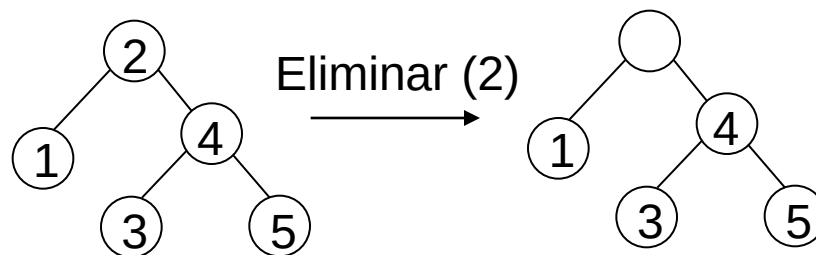
$$\text{prof}(T) = O(\lg(N))$$

- Es decir, el coste de Buscar sobre un AVL es **$O(\lg(N))$ en el caso peor**

Conclusión

- Si usamos un AVL como EdD para un diccionario, tanto **Buscar** como **Insertar** tienen un coste $O(\log(N))$ en el caso peor.
- ¿Qué ocurre con **Borrar**?
 - No es fácil reajustar los nodos de un AVL después de haber eliminado un nodo.
 - La solución habitual es realizar un **Borrado Perezoso**: en lugar de eliminar el nodo, se marca como libre, además si el elemento se reinserta, la inserción es muy fácil y rápida.

Ejemplo:



- El inconveniente de este método es que se pierden posiciones de almacenamiento

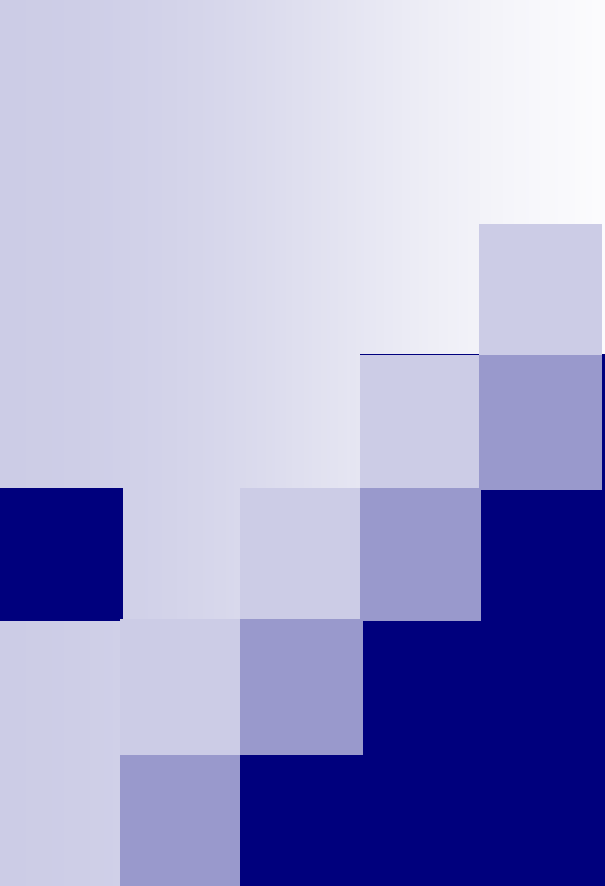
En esta sección hemos aprendido...

- El concepto de árbol AVL.
- A construir un árbol AVL insertando como en ABdBs y corrigiendo desequilibrios mediante rotaciones
- A estimar el número mínimo de nodos de un AVL de profundidad P
- La relación de lo anterior con los números de Fibonacci y algunas propiedades de estos
- Que la profundidad de un AVL con N nodos es $O(\lg(N))$
- Que el caso peor de búsqueda en un AVL es $O(\lg(N))$



Herramientas y técnicas a trabajar

- Construcción y propiedades de árboles AVL
- Construcción y propiedades de árboles de Fibonacci
- Problemas a resolver (al menos)
119, 120, 125, 126



3.4 Hashing

Ordenación y búsqueda

- A grandes rasgos, los costes de búsqueda son $1/N$ veces los de ordenación

	Ordenación	Búsqueda
Métod. malos	$O(N^2)$	$O(N)$
Métod. Buenos	$O(N \lg N)$	$O(\lg N)$
Límite	$O(N)$	$O(1)$

Ordenación y búsqueda II

- ¿Es posible hacer búsquedas en tiempo inferior a $O(\log(N))$?
 1. Imposible mediante comparaciones de clave
 2. Pero muy fácil cambiando el punto de vista!!
- Escenario:
 1. TAD diccionario con $\mathcal{D}=\{\text{datos } \mathbf{D}\}$.
 2. Cada dato $\mathbf{D}$ tiene una clave única $\mathbf{k}=\mathbf{k}(\mathbf{D})$.
 3. Buscamos **por** claves pero **no mediante** claves (no cdcs).

Idea 1

1. Calculamos $k^* = \max\{k(D) : D \in \mathcal{D}\}$
2. Guardamos cada D en una tabla T de tamaño k^* (suponiendo que no hay claves repetidas).

Pseudocódigo:

```
ind Buscar(dato D, tabla T)
    si  $T[k(D)] == D$ 
        return  $k(D)$ ;
    else
        devolver NULL
)
```

- Consecuencia: **$n\text{Buscar}(k, D) = O(1)$!!!**
- **Problema:** si k^* es muy grande (aunque $|\mathcal{D}|$ sea pequeño), la cantidad de memoria necesaria para la tabla T es exagerada.

Idea 2

1. Fijamos $M \succsim |\mathcal{D}|$ y se define una función inyectiva (si, $k \neq k' \Rightarrow h(k) \neq h(k')$) $h : \{k(D)/D \in \mathcal{D}\} \rightarrow \{1, 2, 3, \dots, M\}$.
2. Situamos el elemento D en la posición $h(k(D))$ de la tabla T.
3. Pseudocódigo de Buscar :

```
ind Buscar2(dato D, tabla T)
  si T[h(k(D))] == D
    return h(k(D));
  else
    devolver NULL
```

Obs: $n_{\text{Buscar2}}(k, D) = O(1)$

- Tiempo de búsqueda constate con un consumo de memoria no exagerado.
- Problema: muy difícil encontrar una función **inyectiva y universal** (independiente del conjunto de claves).

Idea 3

1. Buscamos una función h universal (válida para cualquier conjunto de claves).
2. Somos flexibles con la inyectividad de h :
Permitimos que h no sea inyectiva, luego dos datos distintos pueden optar a ocupar la misma posición en la tabla T , pero
 - a) Imponemos que el número de **colisiones**, esto es, pares $k \neq k'$ pero $h(k) = h(k')$ sean pocos.
 - b) Implementamos algún mecanismo de resolución de colisiones
1. Cuestiones abiertas:
 - a) Cómo encontrar una tal h
 - b) Cómo resolver colisiones

Funciones hash

- Objetivo: probabilidad pequeña de colisiones
- Si T tiene M datos, lo óptimo sería que
$$p(\text{colisión})=1/M$$
- Idea: $h(D)$ = valor al lanzar un dado de M caras pero
 - Cada vez que aparece D el dado lo puede enviar a posiciones distintas!!
 - Luego queremos que $h(k(D))$ siempre valga lo mismo para cada $k(D)$ particular.
- Esto es, queremos que h sea **función y aleatoria**
- Como rand() en C
- Q: ¿cómo construir funciones aleatorias?

Hash por división

- Dado un diccionario $\mathcal{D}$ fijamos un número $m > |\mathcal{D}|$, que sea primo.
- Definimos $h(k) = k \% m$
- Con alguna condición adicional sobre m , se puede conseguir que para valores k_j aleatorios, los valores $h(k_j)$ también lo parezcan
 - Esto es, superan diversos tests de aleatoriedad

Hash por multiplicación

- Fijamos un número $m > |\mathcal{D}|$, no necesariamente primo (por ejemplo 2^k o 10^k) y un número Φ irracional (p. ej. $(1+\sqrt{5})/2$ ó $(\sqrt{5}-1)/2$)
- Definimos
$$h(k) = \lfloor m \cdot (k \cdot \Phi) \rfloor,$$
con (x) la parte fraccionaria de x : $(x) = x - \lfloor x \rfloor$
- De nuevo se puede conseguir que para valores k_j aleatorios, los valores $h(k_j)$ también lo parezcan
- Cuestión pendiente: **cómo resolver colisiones.**

Función hash uniforme

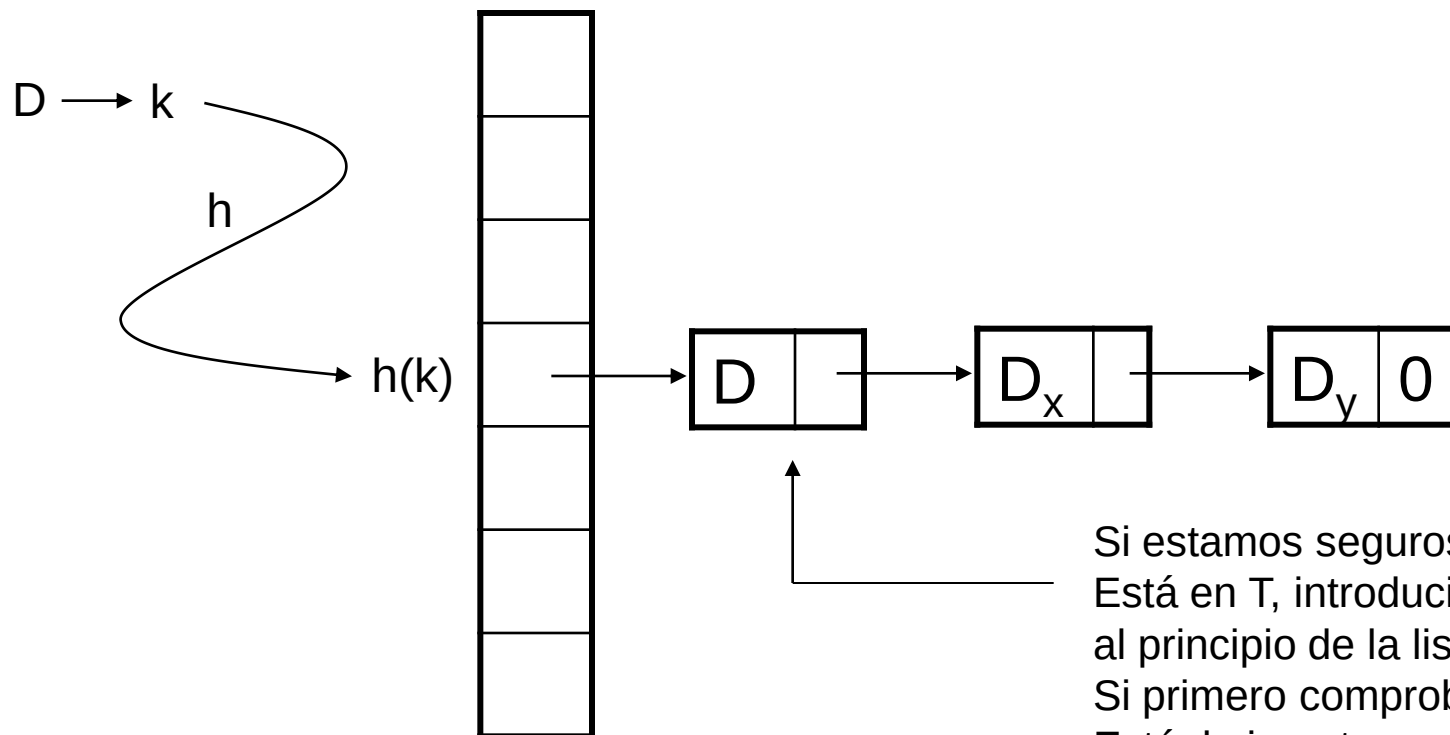
- **Definición:** Decimos que una función hash h es **uniforme** si

Dados k, k' ; $k \neq k'$ entonces $p(h(k)=h(k'))=1/m$

- Las funciones hash uniforme son “ideales”.
 1. No se pueden conseguir por medios algorítmicos.
 2. Pero el rendimiento para ellas es óptimo.
- Las usaremos en nuestros análisis teóricos

Resolución por Encadenamiento

- En el hash por encadenamiento, usamos como tabla hash una **tabla de punteros a listas enlazadas**



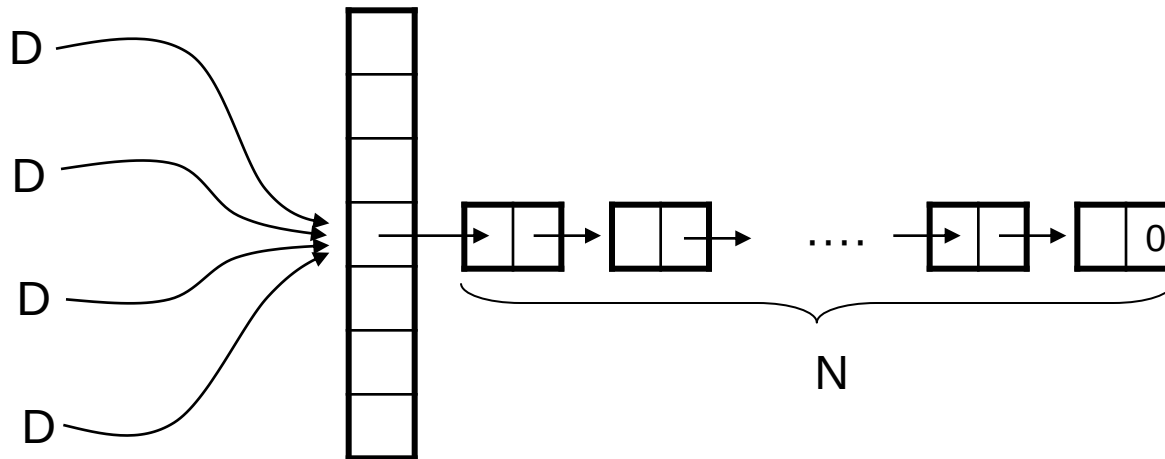
Si estamos seguros de que D no Está en T , introducimos el dato D al principio de la lista enlazada. Si primero comprobamos que no Está, lo insertamos al final

Buscar en Encadenamiento

- Pseudocódigo: Bln en lista enlazada.

```
ind Buscar(dato D, tabla T)
  return BLin(D, T[h(k(D))]);
```

- Coste ya no $O(1)$, pues en **BLin** hay un bucle
- Además $W_{\text{BLin}}(N) = N$ si para todo k , $h(k) = h_0$



Obs: Esta situación puede pasar, pero debería ser muy poco probable si la función hash está bien construida.

Encadenamiento con hash uniforme I

- **Proposición:** Sea h función hash uniforme en tabla hash con encadenamiento y dimensión m y sean N los datos a introducir; entonces:

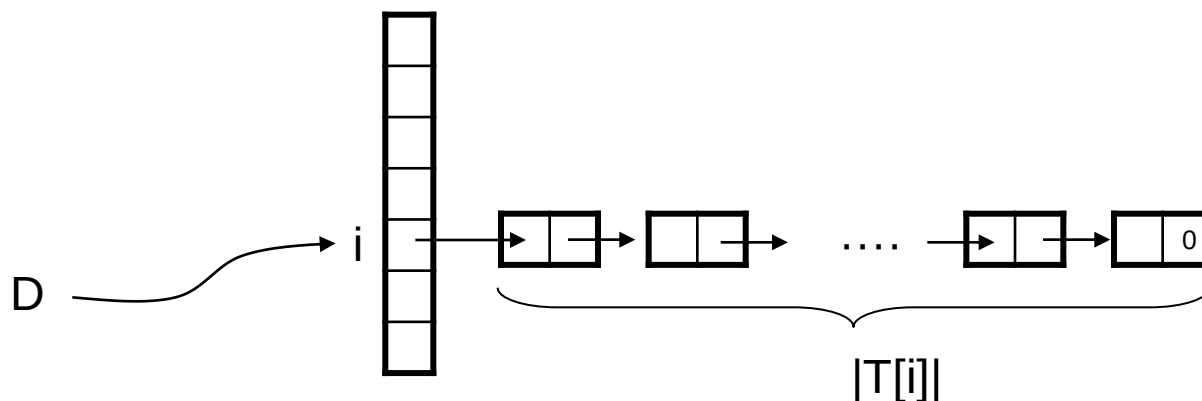
$$(i) \quad A_{BHE}^f(N, m) = \frac{N}{m} = \lambda \quad \longleftarrow \quad \text{Factor de carga}$$

$$(ii) \quad A_{BHE}^e(N, m) = 1 + \frac{\lambda}{2} + O(1)$$

- λ se denomina el **factor de carga**: cuanto mayor es, tanto más costosa es la búsqueda

Coste medio en búsqueda sin éxito

- **Demostración (i):** Sea **D** un dato que no esta en la tabla hash y sea $h(k(D))=h(D)=i$, sea $n_{BHE}(D,T)=|T[i]|$ (número de elementos en la lista enlazada de la posición i de la tabla hash).



$$\Rightarrow A_{BHE}^f(N, m) = \sum_{i=1}^m \underbrace{p(h(D) = i)}_{h \text{ uniforme}} |T[i]| = \frac{1}{m} \sum_{i=1}^m |T[i]| = \frac{N}{m} = \lambda$$

Coste medio en búsqueda con éxito

- **Demostración (ii):** Vamos a reducir la búsqueda con éxito a una búsqueda sin éxito en una tabla más pequeña.
- Para ello numeramos los datos de la tabla T , según el orden en el que los introducimos en la tabla T , $\{D_1, D_2, \dots, D_j, \dots, D_N\}$
- Además denotamos por T_i al estado de la tabla T **antes** de introducir el elemento D_i (la tabla T_i tiene los elementos $D_1, D_2, \dots, D_{j-1}$),
- Por tanto D_i **no está** en T_i , con lo cual se tiene:

$$\underbrace{n_{BHE}^e(D_i, m)}_{\text{búsqueda con éxito}} = 1 + \underbrace{n_{BHE}^f(D_i, m)}_{\text{búsqueda sin éxito}}$$

Búsqueda con éxito =
1 + Búsqueda sin éxito

Nota: aquí asumimos que cada elemento D_i se inserta al final de la lista enlazada.

Coste medio en búsqueda con éxito II

- Asumimos la aproximación

$$n_{BHE}^e(D_i, m) \cong 1 + A_{BHE}^f(i-1, m) = 1 + \frac{i-1}{m} \quad \leftarrow \text{Factor de carga en } T_i$$

entonces

$$\begin{aligned} A_{BHE}^e(N, m) &= \frac{1}{N} \sum_{i=1}^N n_{BHE}^e(D_i, m) \cong \frac{1}{N} \sum_{i=1}^N \left(1 + \frac{i-1}{m} \right) = 1 + \frac{1}{Nm} \sum_{i=1}^{N-1} i = \\ &= 1 + \frac{1}{Nm} \frac{N(N-1)}{2} = 1 + \frac{1}{2} \frac{N}{m} - \frac{1}{2m} = 1 + \frac{\lambda}{2} + O(1) \end{aligned}$$

$$A_{BHE}^f(N, m) = \frac{N}{m} = \lambda$$

$$A_{BHE}^e(N, m) = 1 + \frac{\lambda}{2} + O(1)$$

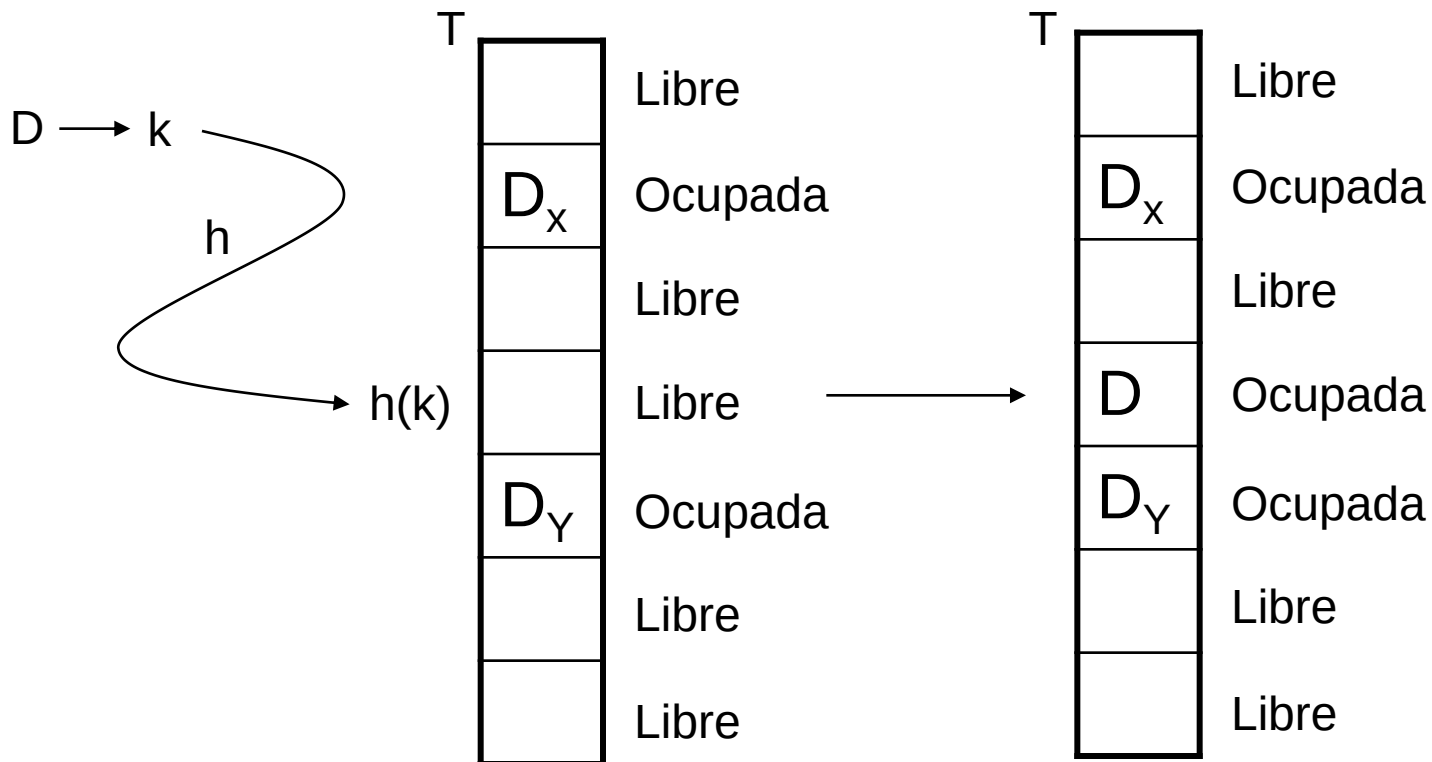
Obs: Si la función hash es uniforme se obtienen búsquedas en tiempo constante si $\lambda = \Theta(1)$, lo cual ocurre si $N \cong m$. Por ejemplo si $N=200$ y $m=100$

$$A^f \cong 200/100 = 2$$

$$A^e \cong 1 + 2/2 = 2$$

Hashing por direccionamiento abierto

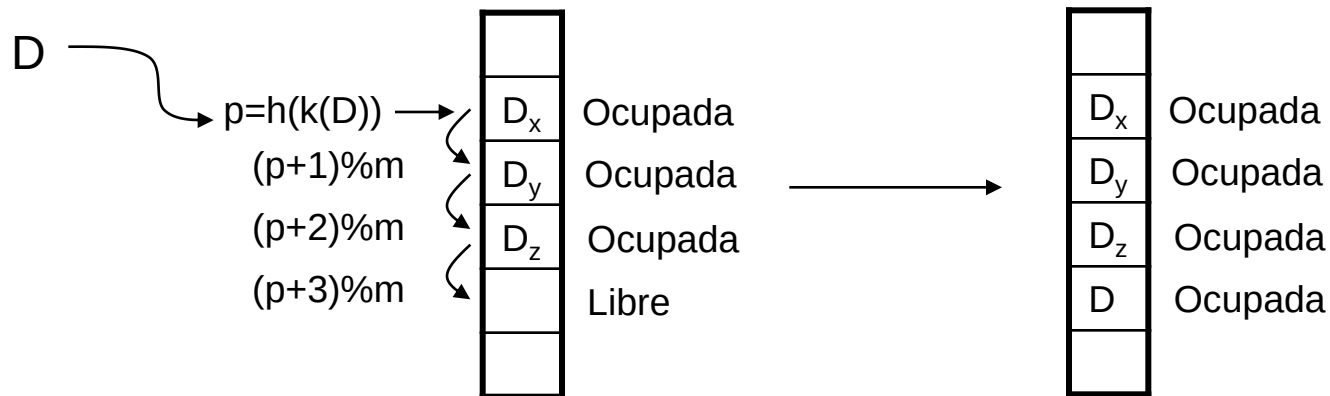
- En el hash por direccionamiento abierto la tabla T contiene los datos



- ¿Qué hacemos cuando la función hash nos asigna una casilla que está ya ocupada (colisión)?

Colisiones en direccionamiento abierto

- Hay varios métodos para resolución de colisiones en direccionamiento abierto mediante repetición de sondeos
- **Sondeos lineales:** Si la posición $p = T[h(D)]$ está ocupada, se intenta colocar D sucesivamente en las posiciones $(p+1)\%m$, $(p+2)\%m$, ..., hasta llegar a un i donde la posición $(p+i)\%m$ está libre.



Colisiones en direccionamiento abierto

- **Sondeos cuadráticos:** Igual que los sondeos lineales pero intentando en las posiciones $p=(p+0^2)\%m, (p+1^2)\%m, (p+2^2)\%m, \dots$, hasta que para algún i , $(p+i^2)\%m$.
- **Sondeos aleatorios:** Intentamos las posiciones, $p_1, p_2, p_3, \dots, p_i$ obtenidas aleatoriamente
 - Este método es inviable en la práctica
 - Es una opción “ideal” en tablas hash
 - Es adecuada para calcular el rendimiento de las búsquedas.

Diferencias en los métodos

- **Obs 1:** En hash con encadenamiento, la posición de un dato D , siempre será una posición fija de la tabla ($h(k(D))$), En hash con direccionamiento abierto, la posición de D dependerá $h(k(D))$ y del estado de la tabla en el momento de la Inserción.

- **Obs 2:** En hash con encadenamiento $\lambda (=N/m)$, puede ser >1 .

En hash con direccionamiento abierto siempre se tiene $N \leq m$, y por tanto $\lambda \leq 1$.

En la práctica se usa $N < m$ y $\lambda < 1$ (por ejemplo $m=2*N$ y $\lambda=.0.5$).

Coste medio con sondeos aleatorios I

- **Proposición:** Sea h función hash uniforme en tabla hash con direccionamiento abierto y sondeos aleatorios. Entonces:

$$(i) \quad A_{BHE}^f(N, m) = \frac{1}{1 - \lambda}$$

$$(ii) \quad A_{BHE}^e(N, m) = \frac{1}{\lambda} \log \frac{1}{1 - \lambda}$$

Obs 1: Si $\lambda \rightarrow 1$ entonces $A_{BHE}^f(N, m) \rightarrow \infty$

Obs 2: Si $\lambda \rightarrow 1$ entonces $A_{BHE}^e(N, m) \rightarrow \infty$

Nota: Estos dos resultados se dejan como ejercicio.

Coste medio en búsqueda sin éxito

■ Demostración (i): Sea T una tabla hash con DA, de dimensión m y N datos. Como h es uniforme se tiene, dado un dato $\mathbf{D}$

$$p(T[h(\mathbf{D})] \text{ ocupada}) = N/m = \lambda$$

N datos en una
Tabla de tamaño m

$$p(T[h(\mathbf{D})] \text{ libre}) = 1 - \lambda$$

$$\Rightarrow A_{BHDA}^e(N, m) = \sum_{k=1}^{\infty} \underset{\substack{\uparrow \\ \text{nº de sondeos hechos}}}{k} \cdot p(\text{hacer } k \text{ sondeos}) = \sum_{k=1}^{\infty} k \cdot \underset{\substack{\uparrow \\ \text{Para hacer } k \text{ sondeos} \\ \text{tiene que ocurrir}}}{\lambda^{k-1}} (1 - \underset{\substack{\uparrow \\ \text{1 posición libre}}}{\lambda}) =$$

$\left\{ \begin{array}{l} k-1 \text{ posiciones ocupadas} \\ \text{y} \\ 1 \text{ posición libre} \end{array} \right\}$
 $\Downarrow$
 $p(\text{hacer } k \text{ sondeos}) = \lambda^{k-1}(1-\lambda)^1$

$$= (1 - \lambda) \sum_{k=1}^{\infty} k \cdot \lambda^{k-1} = (1 - \lambda) \frac{d\left(\sum_{k=0}^{\infty} \lambda^k\right)}{d\lambda} = (1 - \lambda) \frac{d\left(\frac{1}{1 - \lambda}\right)}{d\lambda} = (1 - \lambda) \frac{1}{(1 - \lambda)^2} = \frac{1}{1 - \lambda}$$

Coste medio en búsqueda con éxito I

- **Proposición (ii):** $A_{BHE}^e(N, m) = \frac{1}{\lambda} \log \frac{1}{1-\lambda}$
- **Demostración:** De nuevo vamos a reducir la búsqueda con éxito a una búsqueda sin éxito en una tabla más pequeña.
- Al igual que en BHE enumeramos los datos de la tabla T , según el orden en el que los introducimos en la tabla T , $\{D_1, D_2, \dots, D_j, \dots, D_N\}$, y denotamos por T_i al estado de la tabla T antes de introducir el elemento D_i .

Obs: Si $n_T^e(D_i)$ es el número de sondeos necesarios para **encontrar** (= al numero necesario para **insertar**) el elemento D_i en la tabla T_i tenemos que

$$n_T^e(D_i) = n_{T_i}^f(D_i) \cong A_{BHA}^f(i-1, m)$$

Coste medio en búsqueda con éxito II

- Por tanto se tiene:

$$A_{BHA}^e(N, m) = \frac{1}{N} \sum_{i=1}^N n_T^e(D_i) \cong \frac{1}{N} \sum_{i=1}^N \frac{1}{1 - \frac{i-1}{m}} = \frac{1}{N} \sum_{j=0}^{N-1} \frac{1}{1 - \frac{j}{m}}$$

Esta última expresión se puede aproximar mediante una integral, con lo que se tiene:

$$A_{BHA}^e(N, m) \cong \frac{1}{N} \int_0^N \frac{1}{1 - \frac{x}{m}} dx = \frac{1}{\frac{N}{m}} \int_0^{N/m} \frac{1}{1-u} du = \frac{1}{\lambda} \int_0^{\lambda} \frac{1}{1-u} du$$

Cambio de variable.
 $u=x/m \Rightarrow dx=m \cdot du$

Esta integral es inmediata, con lo que se obtiene:

$$A_{BHE}^e(N, m) = \frac{1}{\lambda} \log \frac{1}{1-\lambda}$$

Costes medios para otros sondeos I

- En la demostración anterior vemos que si tenemos la expresión del rendimiento de la búsqueda sin éxito

$$f(\lambda) = A_{BHA}^f(N, m) = \frac{1}{1-\lambda}$$

podemos calcular el rendimiento de la búsqueda con éxito calculando

$$A_{BHA}^e(N, m) = \frac{1}{\lambda} \int_0^\lambda f(u) du = \frac{1}{\lambda} \int_0^\lambda \frac{1}{1-u} du$$

- Este argumento lo podemos repetir con cualquier tipo de sondeo **S** con direccionamiento abierto, es decir

$$\text{Si } A_S^f(N, m) = f(\lambda) \text{ entonces } A_S^e(N, m) = \frac{1}{\lambda} \int_0^\lambda f(u) du$$

Costes medios para otros sondeos II

- Proposición: Si se usan sondeos lineales:

$$(i) A_{SL}^f(N, m) = \frac{1}{2} \left(1 + \frac{1}{(1-\lambda)^2} \right)$$

$$(ii) A_{SL}^e(N, m) = \frac{1}{\lambda} \int_0^\lambda \frac{1}{2} \left(1 + \frac{1}{(1-\lambda)^2} \right) du = \frac{1}{2} \left(1 + \frac{1}{1-\lambda} \right)$$

En esta sección ...

- Hemos aprendido

- ☐ El Concepto tabla hash.
- ☐ Los mecanismos de construcción y búsqueda en una tabla hash.
- ☐ El concepto de función hash uniforme
- ☐ Algunos tipos universales de funciones hash (división y multiplicación).

En esta sección

■ Y también

- Los principales métodos de resolución de colisiones en una tabla hash:
encadenamiento y direccionamiento abierto
- Los principales métodos de **sondeo** en una tabla hash con **direccionamiento abierto**.
- A estimar el **rendimiento medio** de las búsquedas con o sin éxito en el caso de **sondeos aleatorios**.
- A reducir el rendimiento medio de las búsquedas **con éxito** al rendimiento de las búsquedas **sin éxito**.



Herramientas y técnicas a trabajar

- Funcionamiento y construcción de tablas hash
- Diseño de tablas hash que aseguren un cierto rendimiento
- Estimación de costes medios de búsquedas con éxito a partir de costes medios sin éxito
- Problemas a resolver (al menos)
128, 129, 130, 131, 133, 134, 135, 136