Programación 2

TAD Conjunto

Incluye estructuras de datos: Arreglos de Bits, Hashing y Árboles Balanceados (AVL)

El TAD Set (Conjunto)

En el diseño de algoritmos, la noción de conjunto es usada como base para la formulación de tipos de datos abstractos muy importantes.

Un conjunto es una colección de elementos (o miembros), los que a su vez pueden ellos mismos ser conjuntos o si no elementos primitivos, llamados también átomos.

Todos los elementos de un conjunto son distintos, lo que implica que un conjunto no puede contener dos copias del mismo elemento.

El TAD Set

Una notación usual para exhibir un conjunto es listar sus elementos de la siguiente forma: {1, 4}, que denota al conjunto cuyos elementos son los naturales 1 y 4. Es importante destacar que los conjuntos no son listas, el orden en que los elementos de un conjunto son listados no es relevante ({4, 1} , y también {1,4,1} denotan al mismo conjunto).

Lists (orden y repetición posible de elementos)

MultiSets (no hay orden pero si se permite repet.)

Sets (no hay orden ni repetición de elementos)

El TAD Set

La relación fundamental en teoría de conjuntos es la de pertenencia, la que usualmente se denota con el símbolo $\in$. Es decir, a $\in$ A significa que a es un elemento del conjunto A. El elemento a puede ser un elemento atómico u otro conjunto, pero A tiene que ser un conjunto.

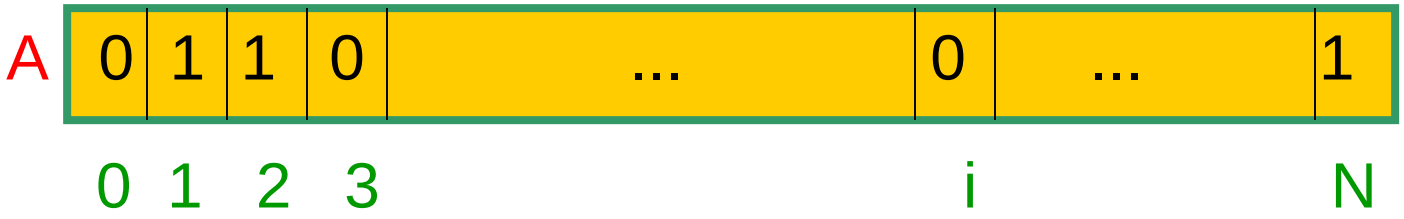
Un conjunto particular es el conjunto vacío, usualmente denotado $\varnothing$, que no contiene elementos.

Las operaciones básicas sobre conjuntos son **unión, intersección y diferencia**.

El TAD Set

- **Vacio** c: construye el conjunto c vacío;
- **Insertar x c: agrega x a c, si no estaba el elemento en el conjunto;**
- **EsVacio** c: retorna true si y sólo si el conjunto c está vacío;
- **Pertenece** x c: retorna true si y sólo si x está en c;
- **Borrar** x c: elimina a x del conjunto c, si estaba;
- **Destruir** c: destruye el conjunto c, liberando su memoria;
- **Operaciones para la unión, intersección y diferencia de conjuntos, entre otras.**

Implementación de Conjuntos con arreglos de booleanos (1/0 o true/false)

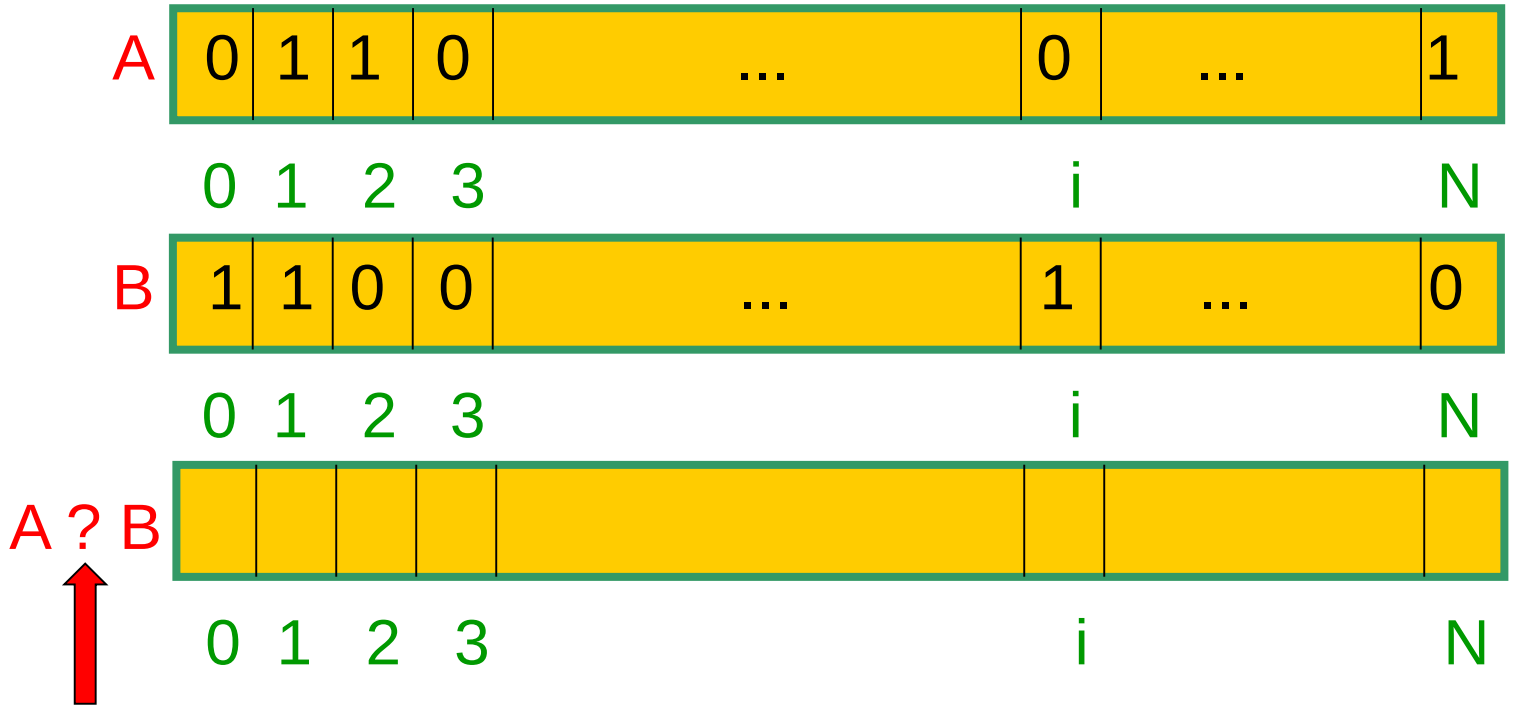


x pertenece al conjunto $\Leftrightarrow$ A[x]=1

- Restricciones ?
- Eficiencia en tiempo de ejecución ?
- Uso de espacio de almacenamiento ?

Implementación de Conjuntos con arreglos de booleanos

- Para Sets, las operaciones Union, InterSec y Diferencia se pueden implementar en un tiempo proporcional al tamaño del conjunto universal.



Unión / Intersección / Diferencia

Implementación de Conjuntos con arreglos de booleanos

- Es necesario establecer una correspondencia de los elementos del dominio con un rango 0..N, para un determinado número N. Ejemplos:
  - número de estudiante (en la facultad)
  - número de curso (en el plan de estudios)
  - número de empleado (en una empresa)
- Discutir los algoritmos para las operaciones de Conjuntos (en general) y Conjunto. Analizar el tiempo de ejecución de las operaciones.

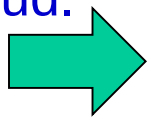
  ¿Qué ocurre con el espacio de almacenamiento?

Implementación de Conjuntos con arreglos de booleanos

¿Qué ocurre con el espacio de almacenamiento?

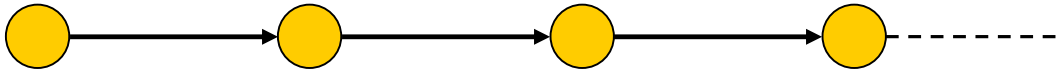
¿Se reserva la memoria mínima necesaria?

Pensar por ejemplo en inscripciones de estudiantes a cursos (matriz) ¿Hay más 0's que 1's?



0	0	0	1	0	1	…	…	…	0
0	0	…	…	…	1/0	0	…	…	…

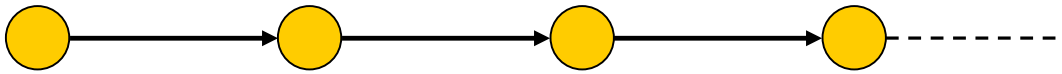
Implementación de Conjuntos con listas encadenadas



- Restricciones ?
- Eficiencia en tiempo de ejecución ?
- Uso de espacio de almacenamiento ?

Conjunto (Set): Vacio, Insertar, EsVacio, Pertenece, Borrar, Unión, Intersección y Diferencia.

Implementación de Conjuntos con listas encadenadas



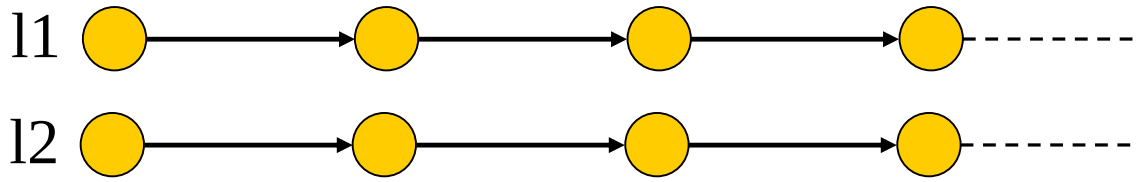
- Esta implementación consiste en representar a los conjuntos por medio de listas encadenadas, cuyos elementos son los miembros del conjunto.
- A diferencia de la implementación con arreglos de booleanos, la representación mediante listas utiliza un espacio proporcional al tamaño del conjunto representado, y no al del conjunto universal.

Implementación de Conjuntos con listas encadenadas

- Más aún, ésta representación es más general, puesto que puede manejar conjuntos que no necesitan ser subconjuntos de algún conjunto universal finito. Esta implementación se adecua bien a una versión no acotada de Conjuntos, pero puede usarse en una versión acotada, llevando la cantidad de elementos en la representación (además de la cota).
- Las listas pueden ser ordenadas o no. El tiempo de ejecución de las operaciones cambia según esta elección.

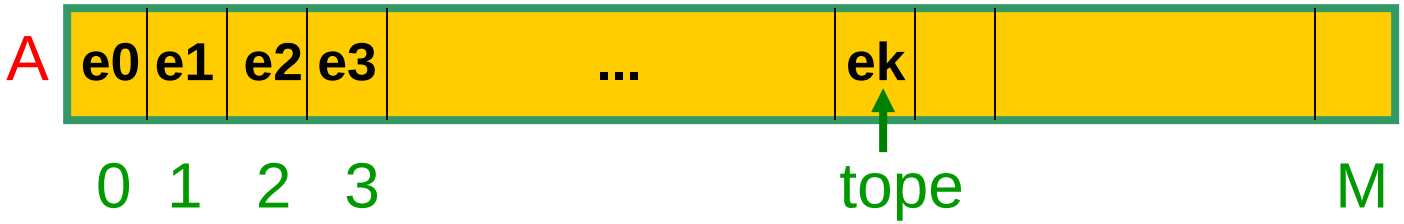
Implementación de Conjuntos con listas encadenadas

- Para Sets, la intersección, la unión y la diferencia sobre listas no ordenadas de longitudes n y m es O(n*m) y, sobre listas ordenadas puede implementarse en O(n+m).



- No ordenadas
  - l1 = [2,7,3,1,9,18,4]; l2 = [8,1,10,3,7,4,99,6,13]
  - l1 ? l2 = […]
- Ordenadas (variantes del merge en listas ordenadas)
  - l1 = [2,3,7,9,18]; l2 = [1,3,4,7,18,99]
  - l1 ? l2 = […]

Implementación de Conjuntos mediante arreglos con tope



- Esta realización es factible si se puede suponer que los conjuntos nunca serán más grandes que la longitud del arreglo en uso. Naturalmente esta implementación se adecua a una versión acotada de Conjuntos.
- Tiene la ventaja de la sencillez sobre la representación con listas encadenadas.

Implementación de Conjuntos mediante arreglos con tope

- Las desventajas de este tipo de representación, al igual que cuando discutimos el TAD Lista, son

  (1) el Conjunto no puede crecer arbitrariamente,

  (2) la operación de borrado es dificultosa y

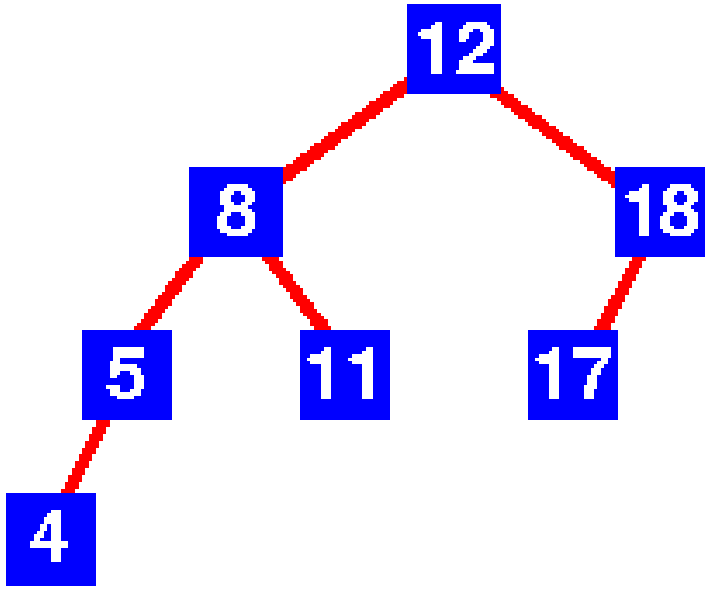
  (3) el espacio de almacenamiento es ineficientemente utilizado.

Implementación de Conjuntos con ABB

- La realización de Conjuntos mediante ABB requiere que los elementos (o su campos claves) permitan un orden lineal.

  Esta implementación se adecua más a una versión no acotada de Conjuntos.

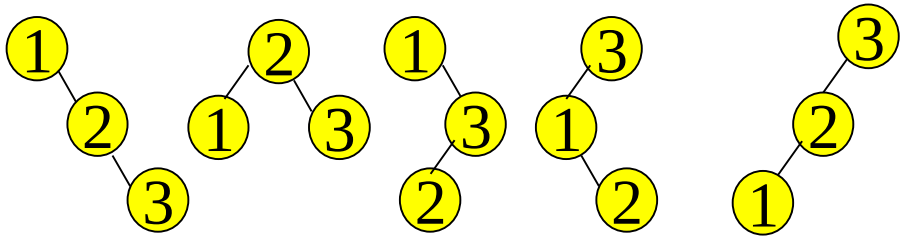
ABB



Conjunto

{ 4, 5, 11, 17, 8, 18, 12 }

Notar que para un conjunto dado existen múltiples ABBs posibles. Por ejemplo **{1, 2, 3}**:



Implementación de Conjuntos con ABB

- La implementación de las operaciones de Conjuntos: Vacio, EsVacio, Insertar, Pertenece y Borrar, corresponden a las mismas operaciones de ABB.
- Si se piensa en Conjuntos, ¿cómo serían las operaciones de unión, intersección y diferencia con ABB?; ¿cuál sería la eficiencia de estas operaciones?

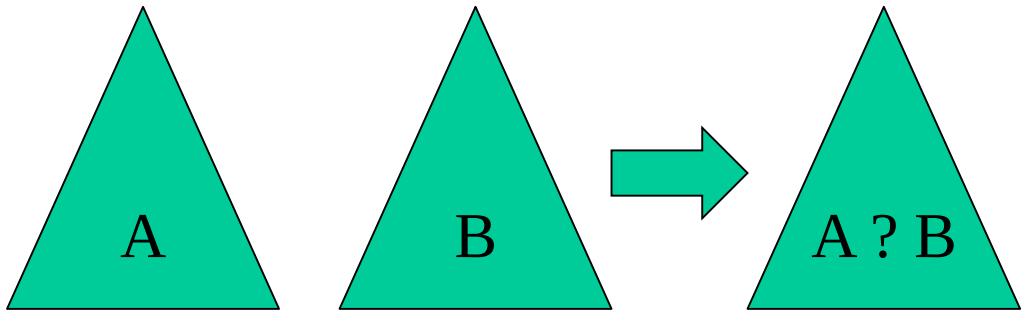
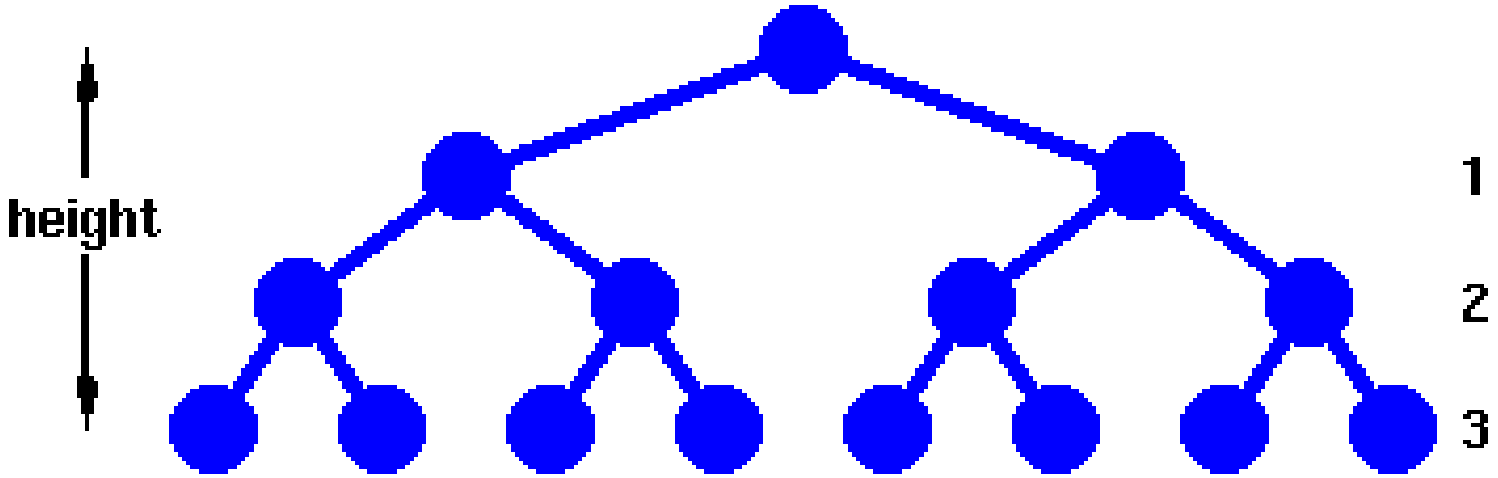


ABB - Análisis

Consideremos un AB (o ABB) completo:



$n = 1 + 2^1 + 2^2 + \ldots + 2^h = 2^{h+1} - 1$

Luego, la altura es $O(\log_2 n)$ en el caso de un árbol completo, pero O(n) de un árbol arbitrario. ¿Por qué?

¿Qué podemos concluir de este análisis sobre la eficiencia en tiempo de ejecución de las operaciones sobre un ABB?

ABB - Análisis

Insertar, Buscar y Borrar tienen tiempo de ejecución proporcional a $\log_2 n$ (siempre se recorre un sólo camino del árbol) si el árbol es completo pero, la eficiencia puede caer a orden n si el árbol se degenera (el caso extremo es una lista).

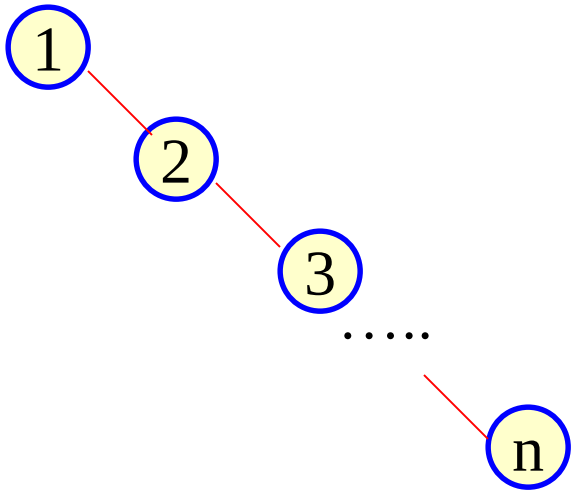


ABB - Análisis

Si bien el orden de tiempo de ejecución promedio de las operaciones citadas para ABB's es $O(\log_2 n)$, el peor caso es $O(n)$.

La idea es entonces tratar de trabajar con ABB's que sean "completos", o al menos que estén "balanceados"…

Tratando de refinar esta idea es que surgen los AVL, que veremos más adelante.

Hashing
Operaciones relevantes en un Conjunto (Set)

Consideremos las operaciones en un Set:

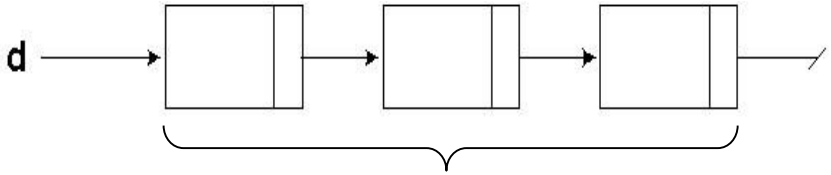
- Insertar (T x, Set & s)
- Borrar (T x, Set & s)
- Pertenece (T x, Set s)

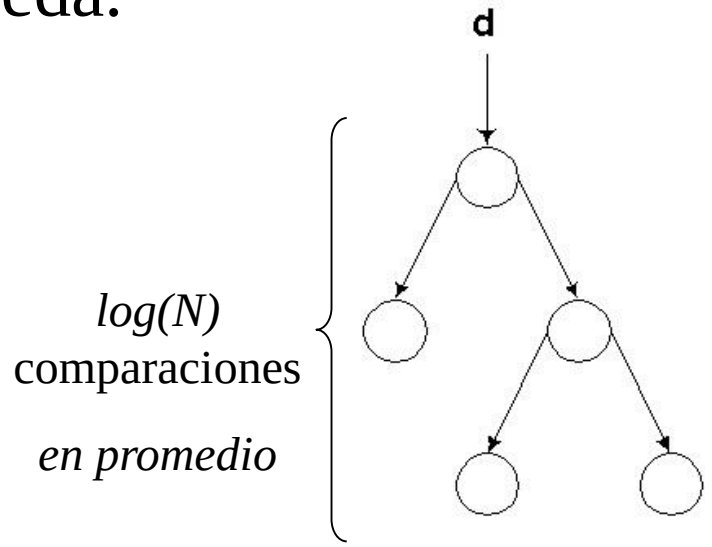
Interesa particularmente la verificación de pertenencia de un elemento.

Hashing
Representaciones

Algunas alternativas conocidas mencionadas:

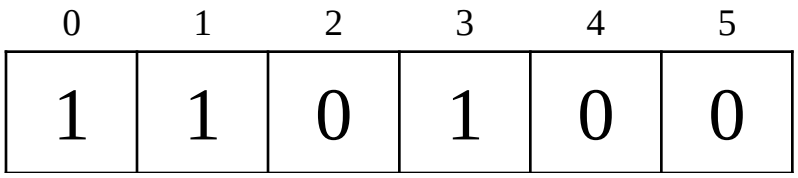
- Listas encadenadas.
- Árboles binarios de búsqueda.





Arreglo de booleanos / bits

Sea *S* un set implementado con un arreglo de booleanos (1 / 0), y s un array de *N* elementos booleanos. Se define a s[*n*] como verdadero si y solo si Pertenece(*n*,*S*).



s[0] = true ➔ 0 ∈ S

s[1] = true ➔ 1 ∈ S

s[2] = false ➔ 2 ∉ S

s[3] = true ➔ 3 ∈ S

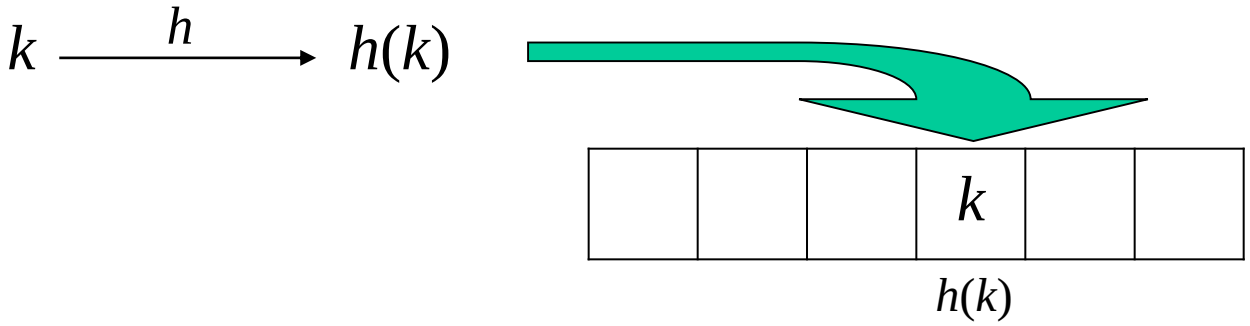
s[4] = false ➔ 4 ∉ S

s[5] = false ➔ 5 ∉ S

Noción de hash

Consiste en:

- Array de *M* elementos conocidos como “buckets” (tabla de hash).
- Función *h* de dispersión sobre las claves.
- Estrategia de resolución de colisiones.



Función de hash

Propiedades imprescindibles:

- Dependiente únicamente de la clave
- Fácil de computar (rápida)
- Debe minimizar las colisiones

Su diseño depende fuertemente de la naturaleza del espacio de claves. Se busca que la función distribuya uniformemente.

Algunas funciones utilizadas comúnmente:

división, *middle square*, polinomial

Método de división

Se define la función h según $h(k) = k \bmod M$, donde M es el tamaño de la tabla (podría eventualmente ser un parámetro de h) y mod es el resto de la división entera (%).

Se recomienda que M sea un número primo.

Valores consecutivos de claves k producen buckets adyacentes en el hash, lo cual no siempre es deseable.

Para cadenas (strings)

Por lo general las claves (o los elementos) son números o cadenas de caracteres.

En este último caso, una opción es sumar los valores ASCII de los caracteres de la cadena y retornar la suma módulo M (tamaño de la tabla). No obstante, la función no distribuye bien las claves si el tamaño de la tabla es grande. Ejemplo: M=10007 y longitud(x)<=8. $h(x) \in [0..1016]$ (1016=127*8).

Para cadenas (strings)

En el libro de Weiss (cap.5) hay más ejemplos de funciones de hash, como por ejemplo: la función que calcula $h(k) = k1 + c*k2 + c^2*k3 + \ldots \bmod M$ (con c una constante (ej. 32)) y k_i el codigo ASCII del i-ésimo caracter de la cadena k.

Importante: Se sugiere usar tablas de un tamaño proporcional a la cantidad esperada de elementos y que este tamaño sea un número primo.

Colisiones

Ninguna función de hash puede garantizar que la estructura está libre de colisiones.

Toda implementación de hash debe proveer una estrategia para su resolución.

Estrategias más comunes:

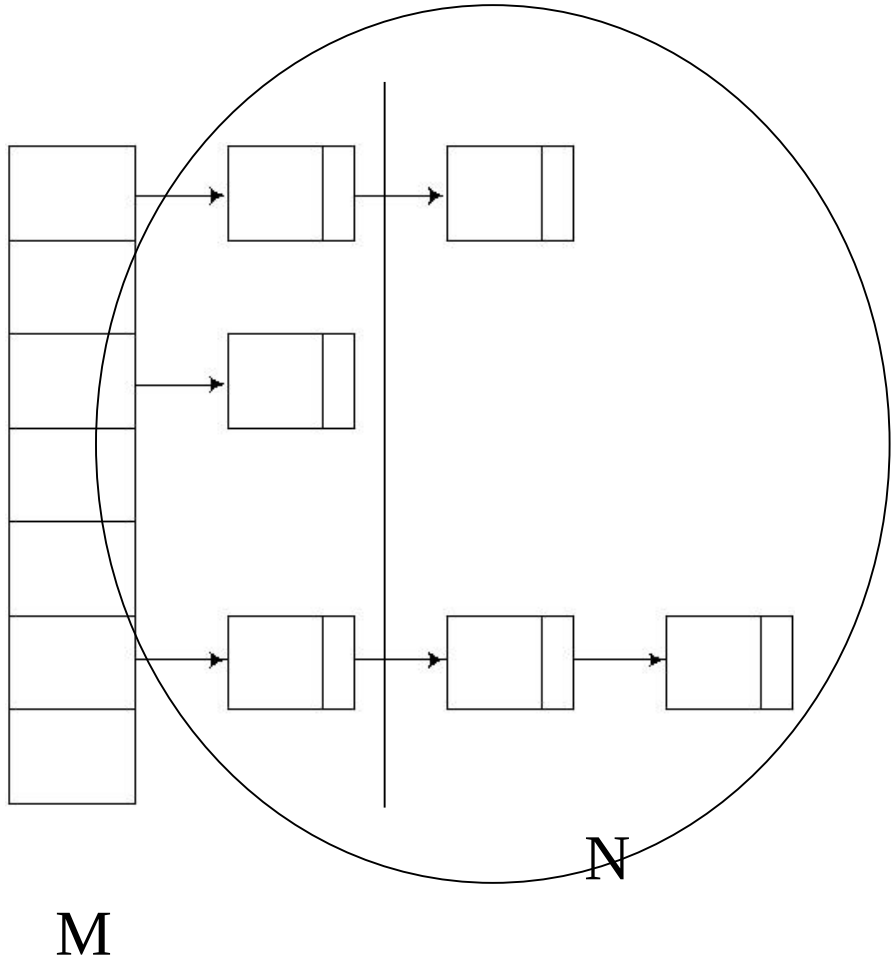
- Separate chaining (*hashing* abierto)
- Linear probing (*hashing* cerrado)
- Double hashing

Separate chaining
Hasing Abierto

Las colisiones resultan en un nuevo nodo agregado en el bucket.

Las listas en promedio tienen largo acotado.

La verificación de pertenencia de *k* implica buscar en la lista del bucket *h*(*k*).



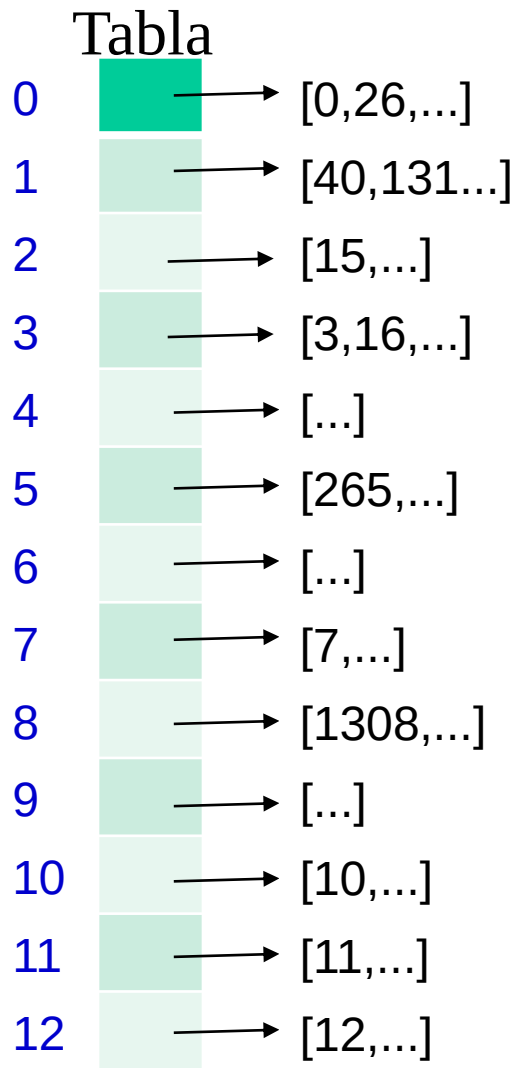
Separate chaining
Hasing Abierto

Ejemplo:

- M=10
- hash: int -> int, hash(x) = x%M
- Insertar: 0, 5, 10, 15, 20, 25 …

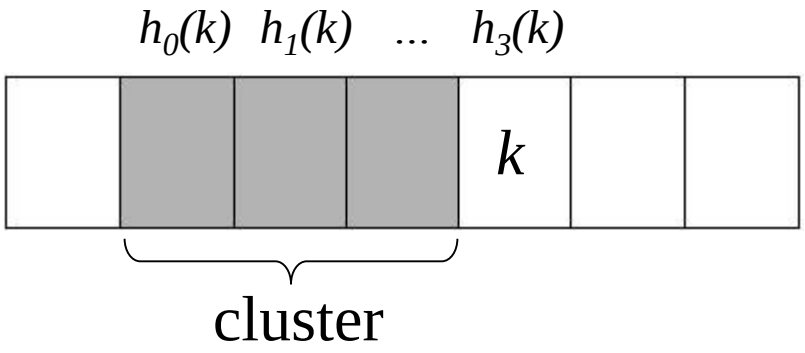
Ahora Insertar (con M=13):

3, 0, 7, 16, 11, 26, 10, 40, 12, 131, 1308, 265, 15 …



Linear probing
Ejemplo de Hashing Cerrado

Dada una clave *k*, la secuencia de prueba hasta encontrar un lugar libre en una tabla de tamaño *M* es $h_i(k) = (h(k) + i) \bmod M$ con $i \in [0, M\text{-}1]$.



Factor de Carga

El factor de carga (FC) es N/M (N es el número de elementos en la tabla y M el tamaño de ésta). La longitud media de una lista en hashing abierto es FC.

El esfuerzo requerido para una búsqueda es el tiempo que hace falta para evaluar la función de hash más el tiempo necesario para evaluar la lista.

Factor de Carga

En una búsqueda infructuosa el nro promedio de enlaces por recorrer es FC.

En una búsqueda exitosa, el nro es FC / 2.

Este análisis demuestra que el tamaño de la tabla no es realmente importante, pero el factor de carga sí lo es.

La regla general en open hashing es hacer el tamaño de la tabla casi tan grande como el nro de elementos esperados (FC $\cong$ 1) y elegir a M un nro primo.

Tiempo promedio y peor caso

O(1) promedio: Hashing utiliza tiempo constante por operación, en promedio, y no existe la exigencia de que los conjuntos sean subconjuntos de algún conjunto universal finito (como en los arreglos de booleanos). *El FC y la función de hash son factores claves.*

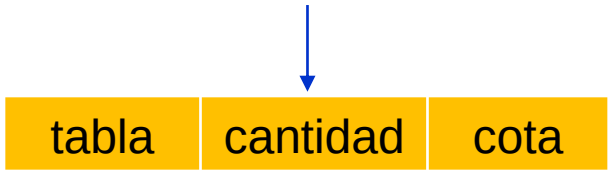
O(n) peor caso: En el peor caso este método requiere, para cada operación, un tiempo proporcional al tamaño del conjunto, como sucede con realizaciones con listas encadenadas y arreglos.

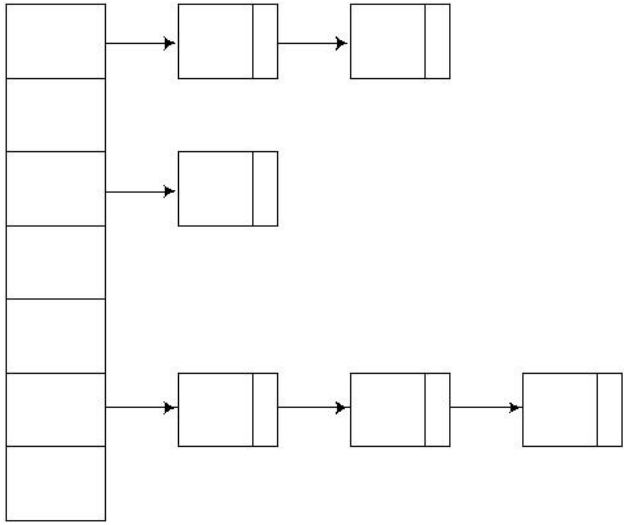
Especificación del TAD Conjunto no acotado? de enteros

```
#ifndef _Conjunto_H
#define _Conjunto_H

struct RepresentacionConjunto;
typedef RepresentacionConjunto * Conjunto;

Conjunto crearConjunto (int cota);
// Devuelve el Conjunto vacío para una cantidad estimada cota.

void insertarConjunto (int i, Conjunto &c);
// Agrega i en c, si no estaba. En caso contrario, no tiene efecto.

void eliminarConjunto (int i, Conjunto &c);
// Elimina i de c, si estaba. En caso contrario, no tiene efecto.

bool perteneceConjunto (int i, Conjunto c);
// Devuelve true si y sólo si i está en c.

bool esVacioConjunto (Conjunto c);
// Devuelve true si y sólo si c es vacío.

void destruirConjunto (Conjunto &c);
// Libera toda la memoria ocupada por c.

#endif /* _Conjunto_H */
```

Implementación de un Conjunto acotado de enteros con hashing abierto

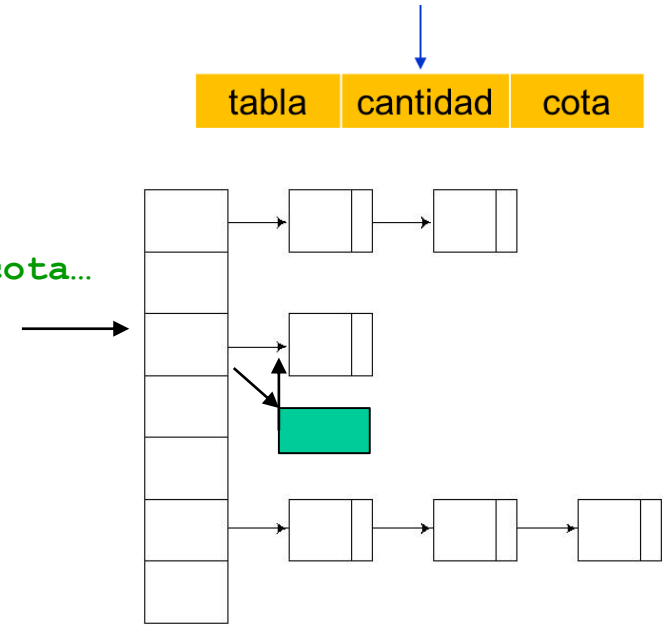
```
#include ...
#include "Conjunto.h"

int hash (int i){return i;} // si todos los enteros son igualmente probables
struct nodoHash{
   int dato;
   nodoHash* sig;
};
struct RepresentacionConjunto{
   nodoHash** tabla;
   int cantidad;
   int cota;
};

Conjunto crearConjunto (int cota) {
   Conjunto c = new RepresentacionConjunto();
   c->tabla = new (nodoHash*) [cota];
   for (int i=0; i<cota; i++) c->tabla[i]=NULL;
   c->cantidad = 0;
   c->cota = cota;
   return c;
}
```





Implementación de Conjunto acotado de enteros con hashing abierto

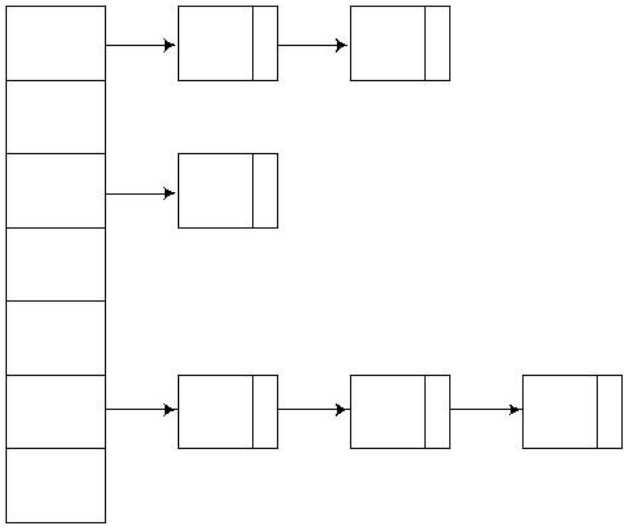
```
void insertarConjunto (int i, Conjunto &c){
\\ No se analiza el factor de carga
if !peteneceConjunto(i, c){
    nodoHash* nuevo = new nodoHash;
    nuevo->dato = i;
    nuevo->sig = c->tabla[hash(i)%(c->cota)];
    \\ la función hash podría aplicarse a i y c->cota…
    c->tabla[hash(i)%(c->cota)] = nuevo;
    c->cantidad++;
    }
}
void eliminarConjunto (int i, Conjunto &c) {
...
}
bool perteneceConjunto (int i, Conjunto c) {
    nodoHash* lista = c->tabla[hash(i)%(c->cota)];
    \\ la función hash podría aplicarse a i y c->cota…
    while (lista!=NULL && lista->cato!=i)
          lista = lista->sig;
    return lista!=NULL;
}

bool esVacioConjunto (Conjunto c) {...}
bool esLlenoConjunto (Conjunto c){...}

void destruirConjunto (Conjunto &c) {...}
```



Definición de un CLON para un Conjunto acotado de enteros implementado con hashing abierto.

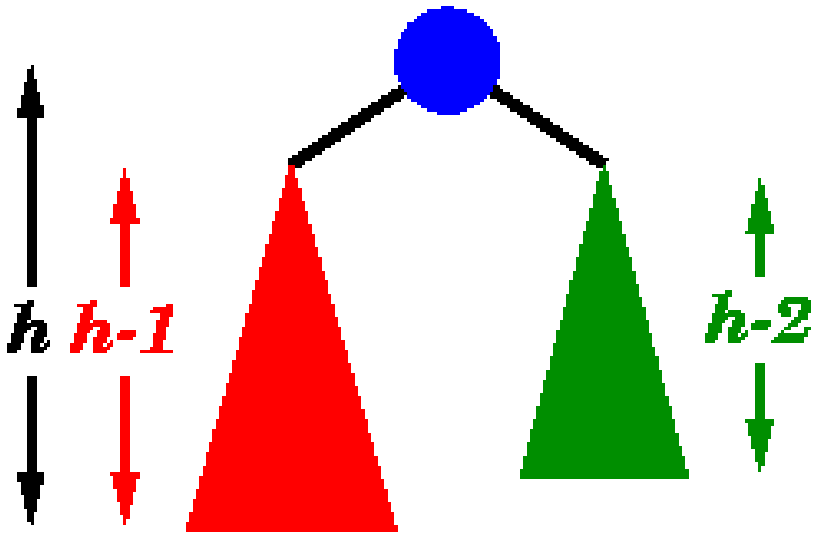
Asumimos que la especificación incluye la operación clonConjunto

```
Conjunto clonConjunto (Conjunto c){
   Conjunto clon = cearConjunto(c->cota);
   for (int i=0; i<c->cota; i++){
        nodoHash* lista = c->tabla[i];
        while (lista!=NULL){
                insertarConjunto(lista->dato, clon);
                lista = lista->sig;
        }
   }
   return clon;
}
```



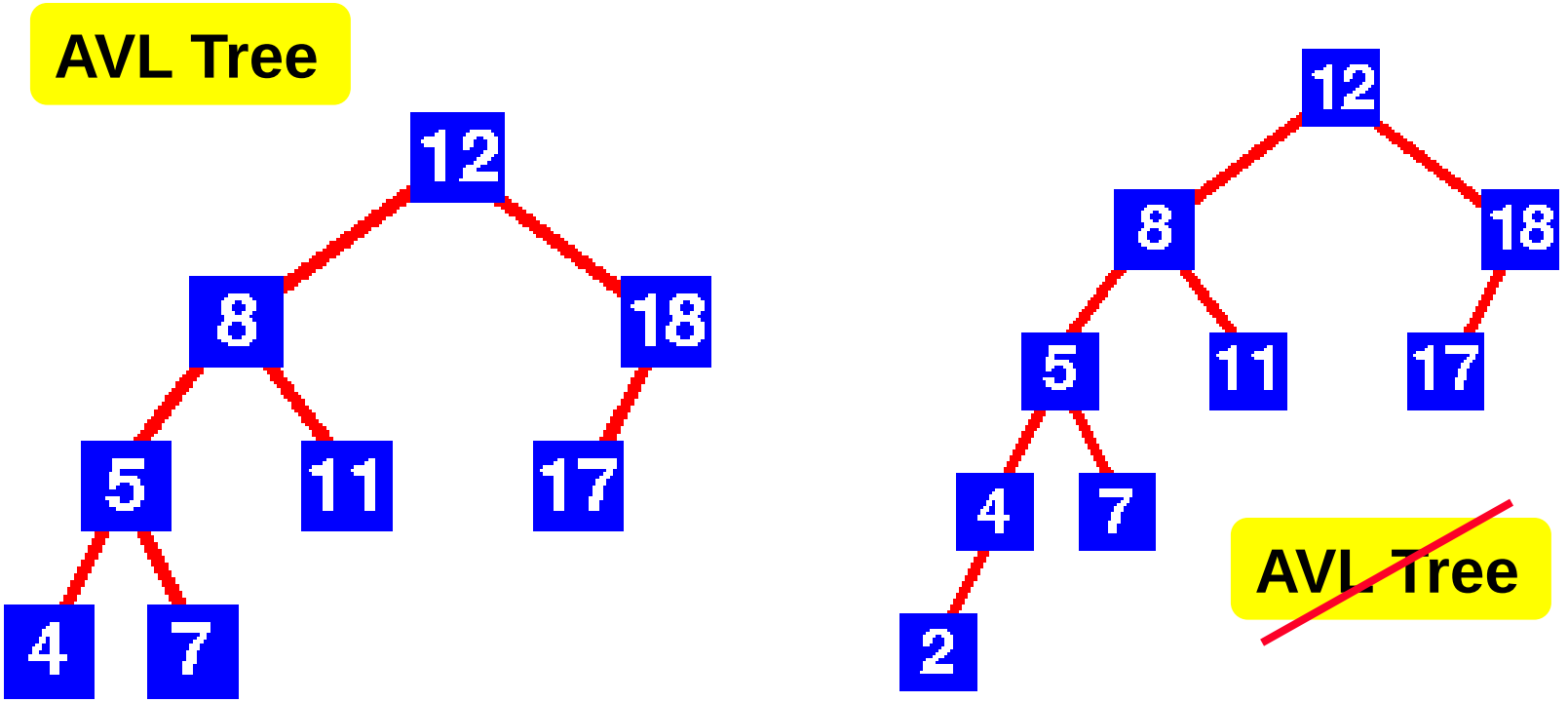
Árboles AVL

- Un árbol AVL (Adelson-Velskii y Landis) es una ABB con una condición de equilibrio. Debe ser fácil mantener la condición de equilibrio, y ésta asegura que la profundidad del árbol es O(log (n)).
- La condición AVL: para cada nodo del árbol, la altura de los subárboles izquierdo y derecho puede diferir a lo más en 1 (hay que llevar y mantener la información de la altura de cada nodo, en el registro del nodo).



Árboles AVL

- Así, todas las operaciones sobre árboles se pueden resolver en O (log(n)), con la posible excepción de la inserción (si se usa eliminación perezosa - recomposición AVL tardía).



Árboles AVL (cont)

- Cuando se hace una inserción es necesario actualizar toda la información de equilibrio para los nodos en el camino a la raíz. La razón de que la inserción sea potencialmente dificíl es que insertar un nodo puede violar la propiedad de árbol AVL.

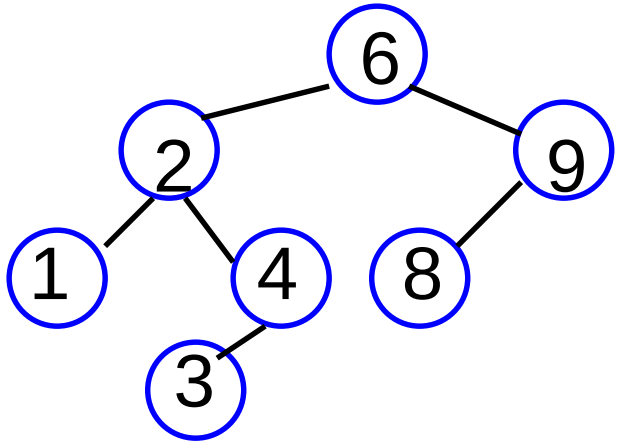


ABB y **AVL**

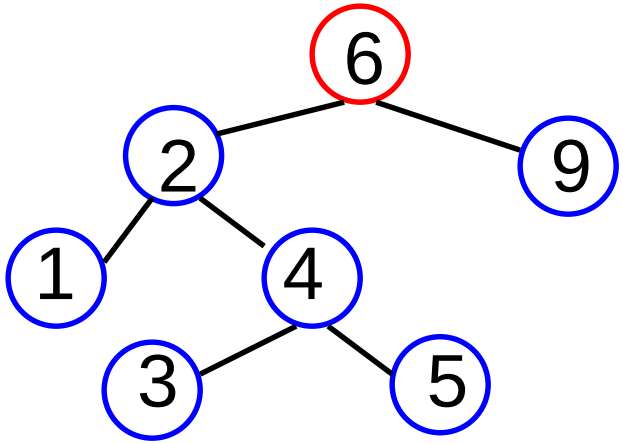


ABB pero **no AVL**

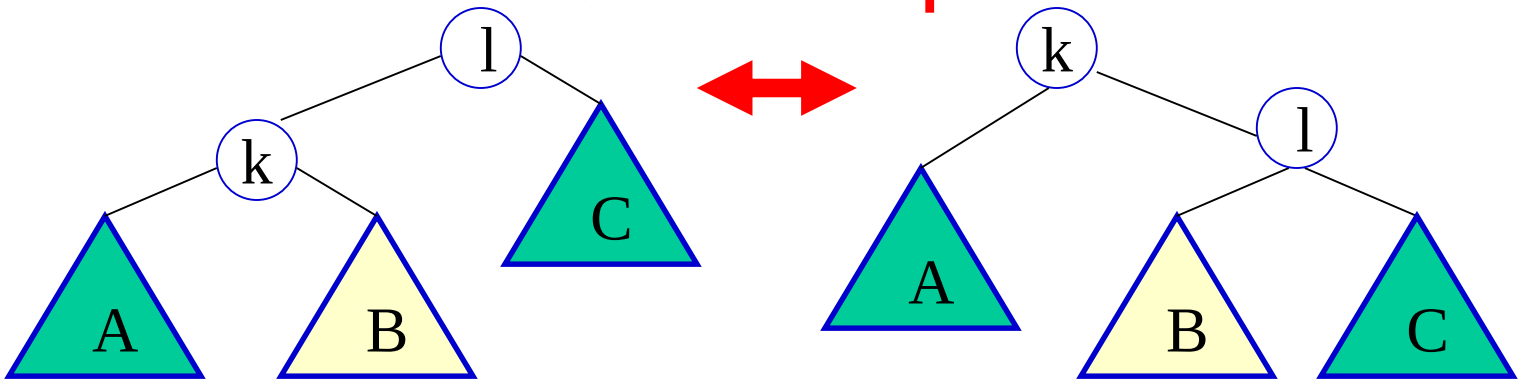
Árboles AVL: rotaciones

- Al insertar (según la inserción ABB) el 7 en el AVL anterior se viola la propiedad AVL para el nodo 9.

- Si éste es el caso, se tiene que restaurar la propiedad AVL antes de dar por culminado el proceso de inserción. Resulta que esto se puede hacer siempre con una modificación al árbol, conocida como **rotación** (simple y doble).

Árboles AVL (cont)

Rotación simple

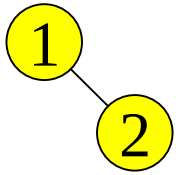


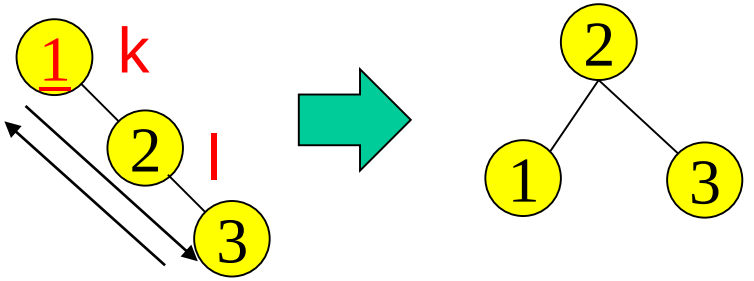
- **Ambos son ABB's ?**
- **Qué implica una rotación con una implementación dinámica del árbol ?**
- **Qué punteros cambian ?**
- **Qué pasa con las alturas de los subárboles A, B y C ?**
- **Coinciden los recorridos In-order de ambos árboles ?**

Árboles AVL (cont)

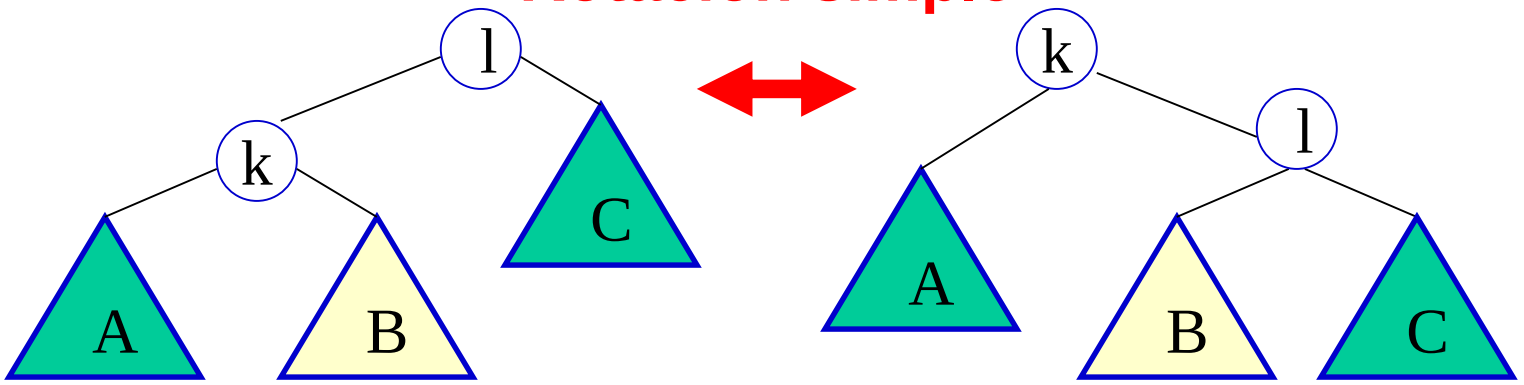
- Insertar los elementos 1 al 7 a partir de un AVL vacio.

1) 

2) 

3) 

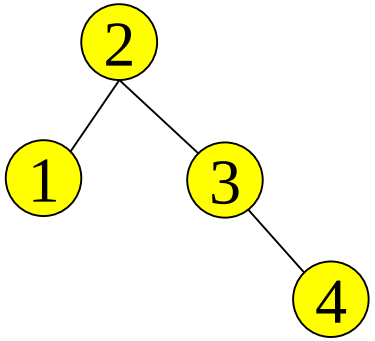
Rotación simple



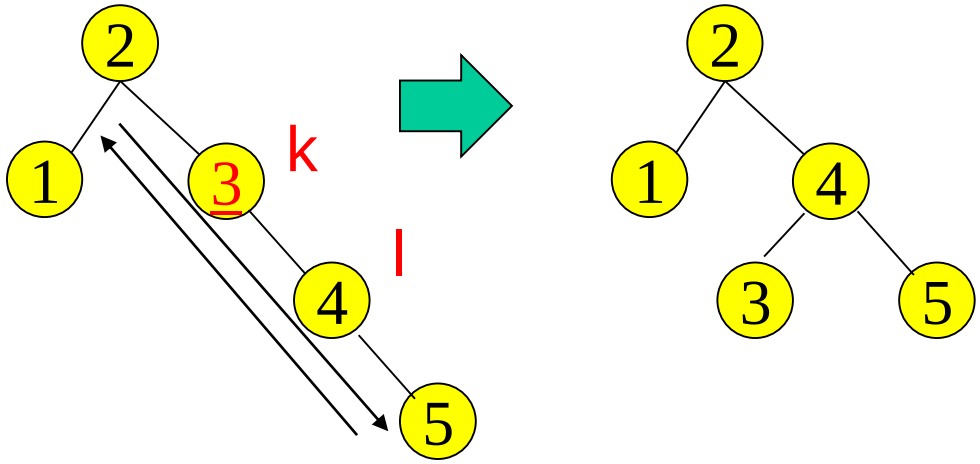
Árboles AVL (cont)

- Inserción del 4 y luego del 5.

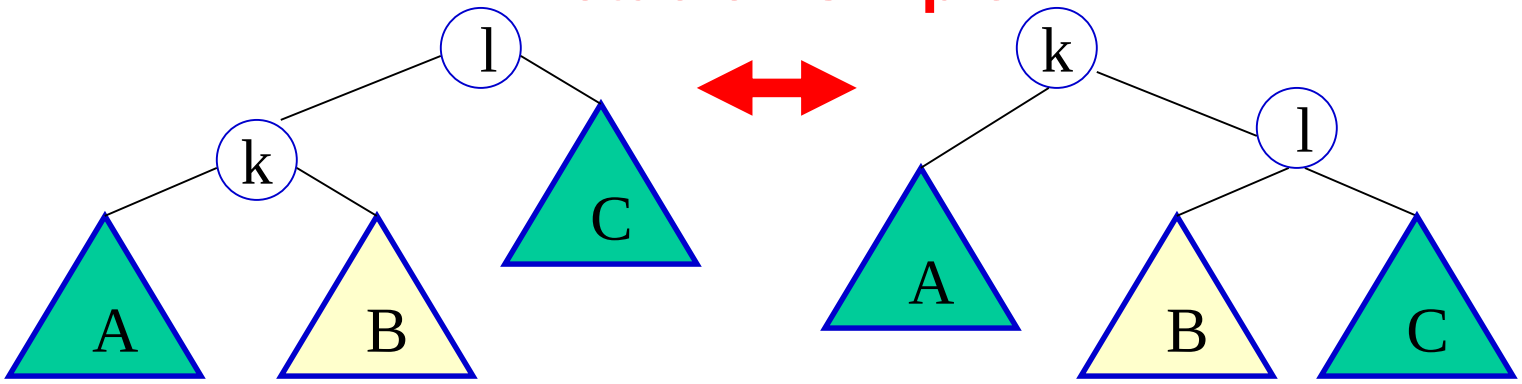
4)



5)

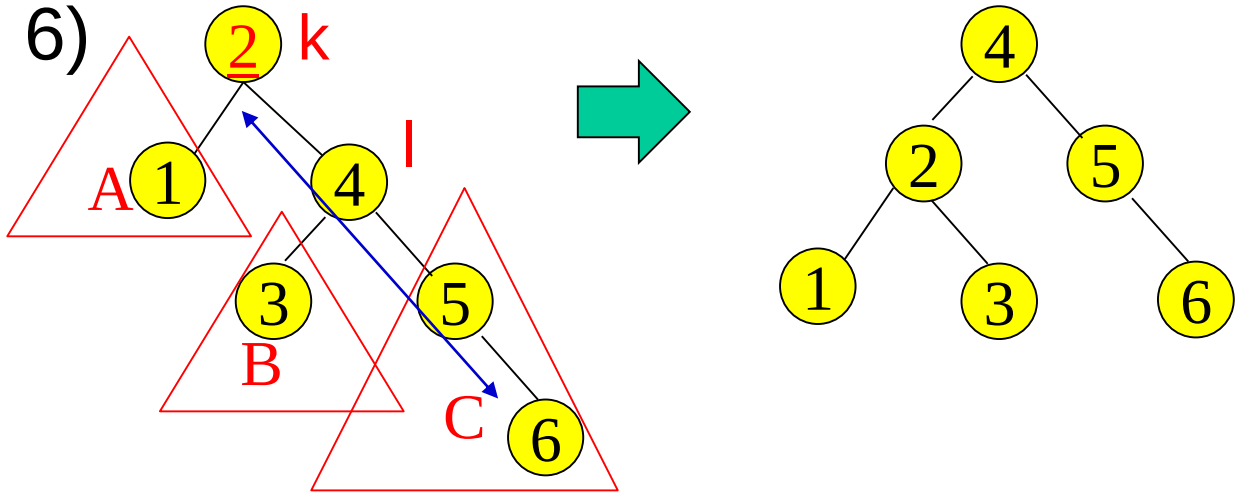


Rotación simple

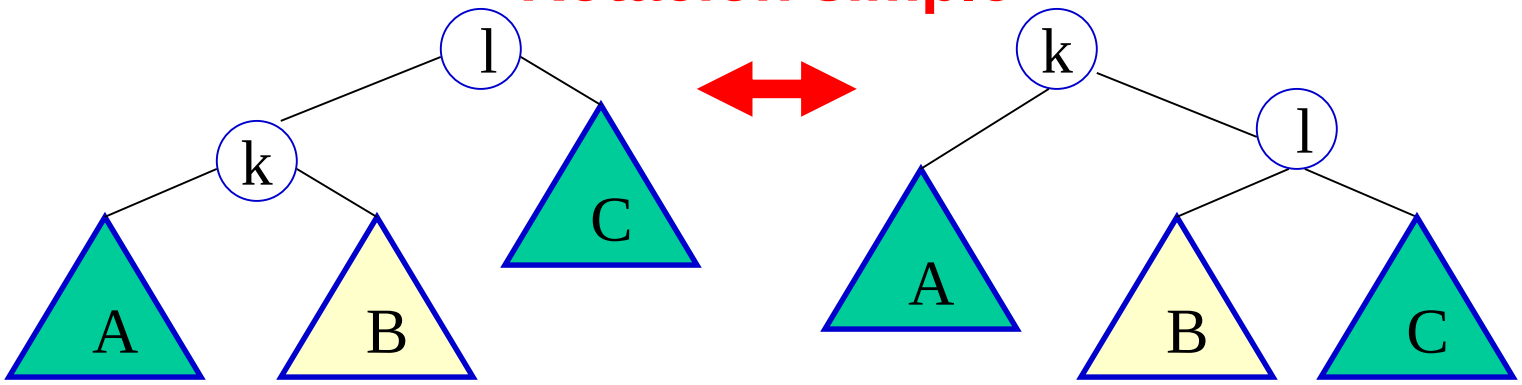


Árboles AVL (cont)

- Insertar los elementos 1 al 7 a partir de un AVL vacio.



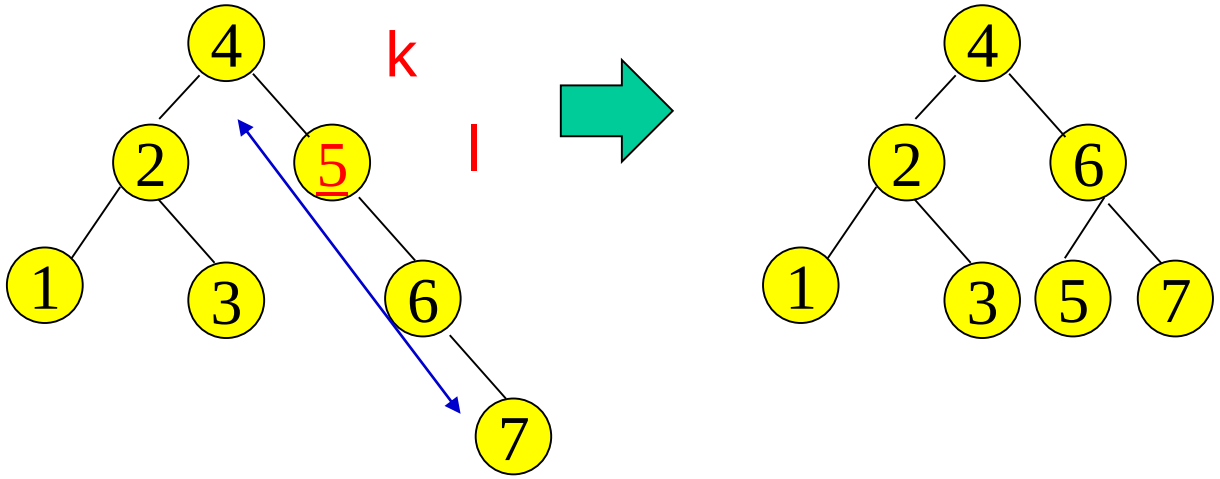
Rotación simple



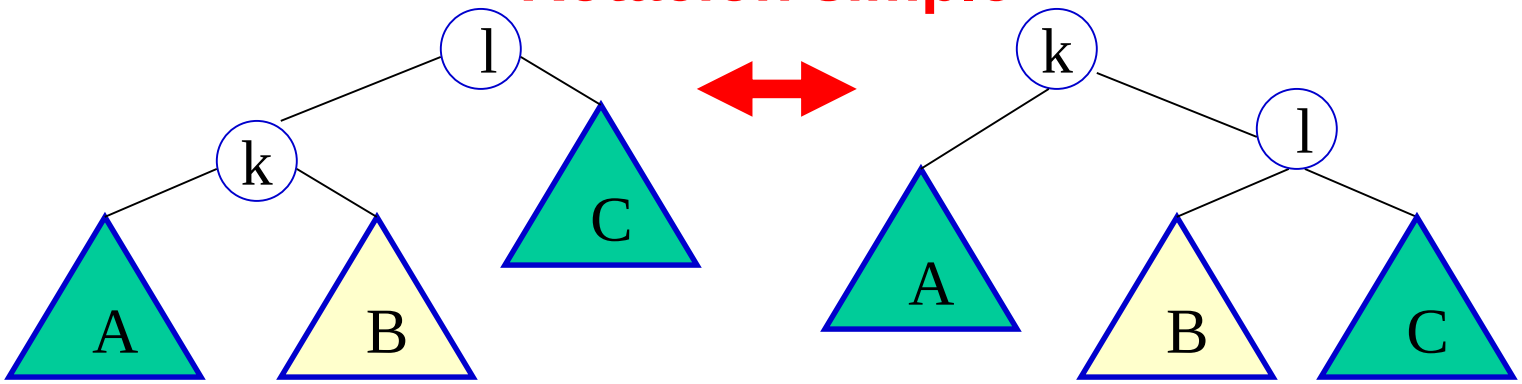
Árboles AVL (cont)

- Inserción del 7.

7)



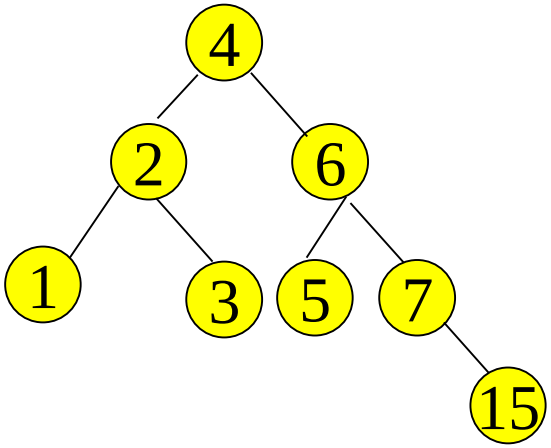
Rotación simple



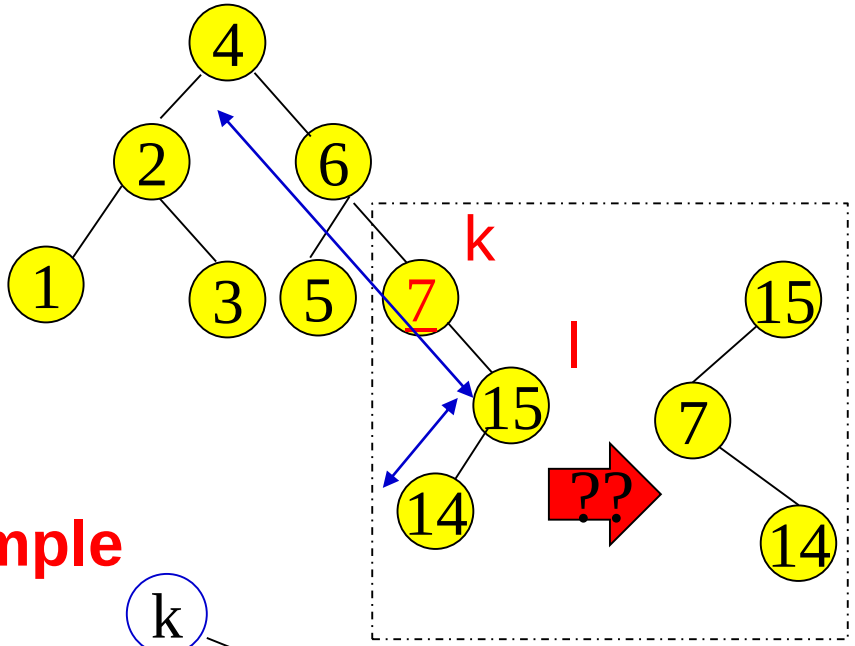
Árboles AVL (cont)

- Ahora insertar los elementos del 15 al 12.

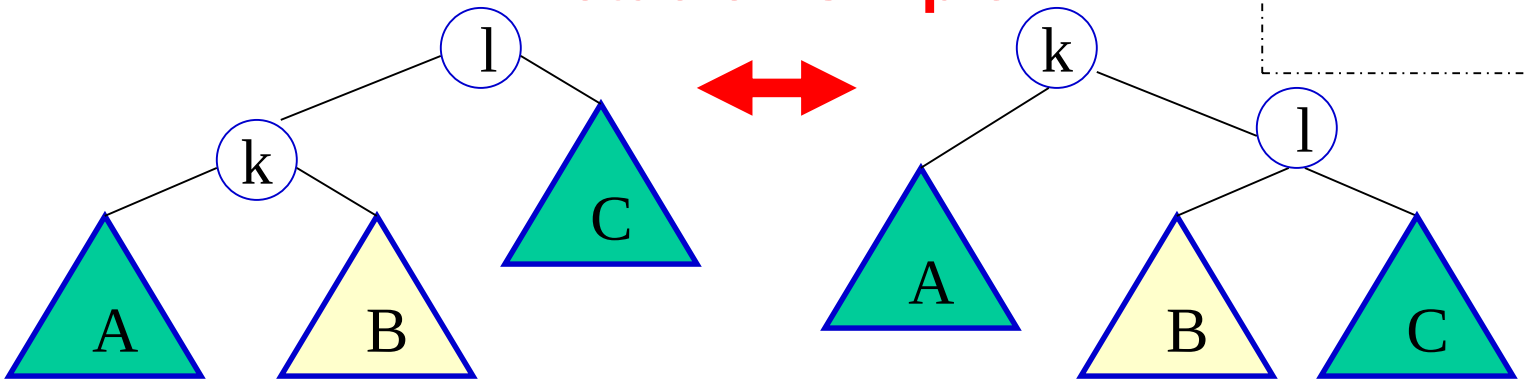
15)



14)



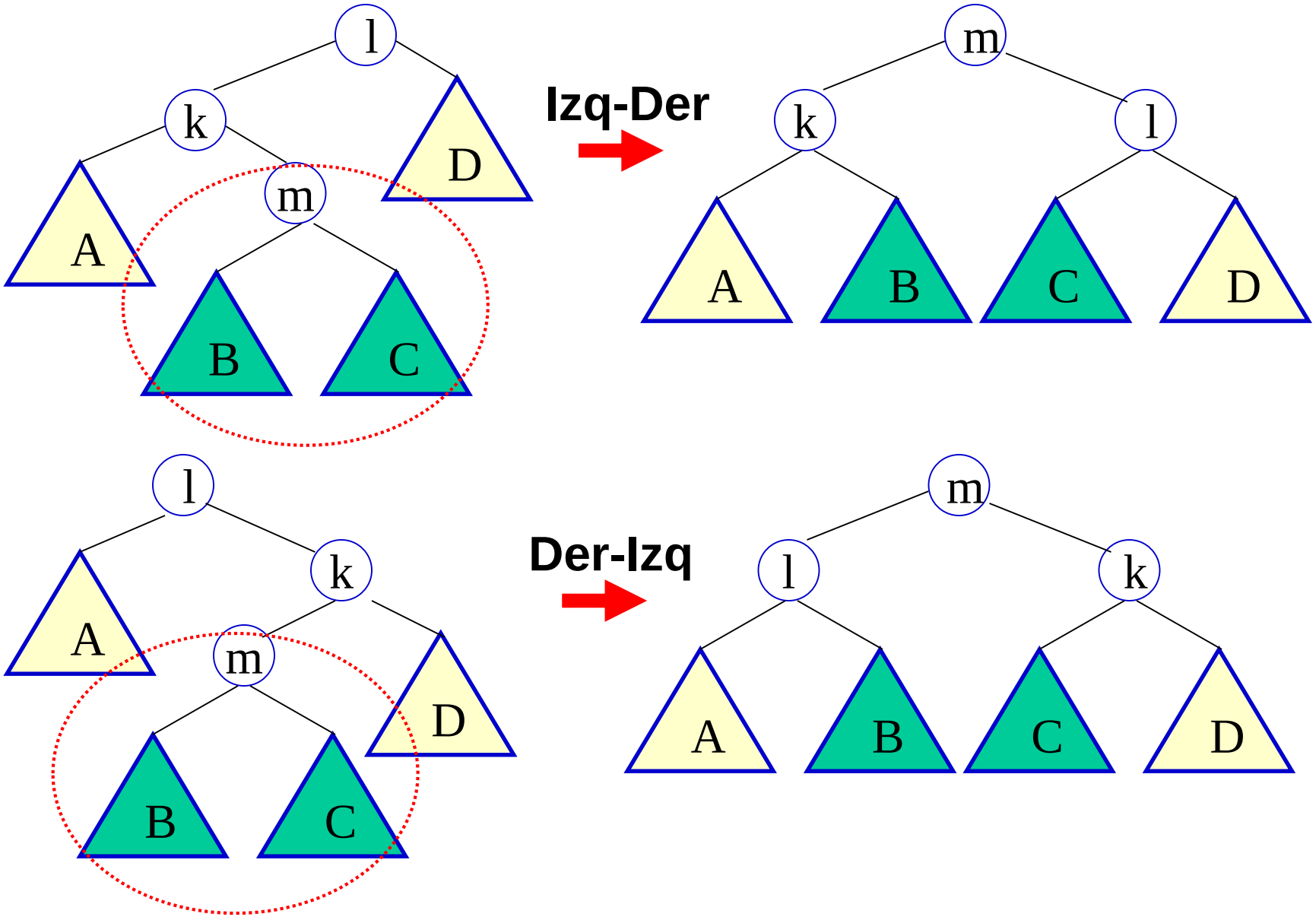
Rotación simple



Árboles AVL (cont)

- Al insertar el 14, una rotación simple no recompone la propiedad AVL.
- Cuando el elemento insertado se encuentra entre los valores correspondientes a los nodos del árbol a rotar (por violación de la propiedad AVL), una rotación simple no recompone la propiedad AVL.
- Solución: usar una rotación doble, que involucra 4 subárboles en vez de 3.

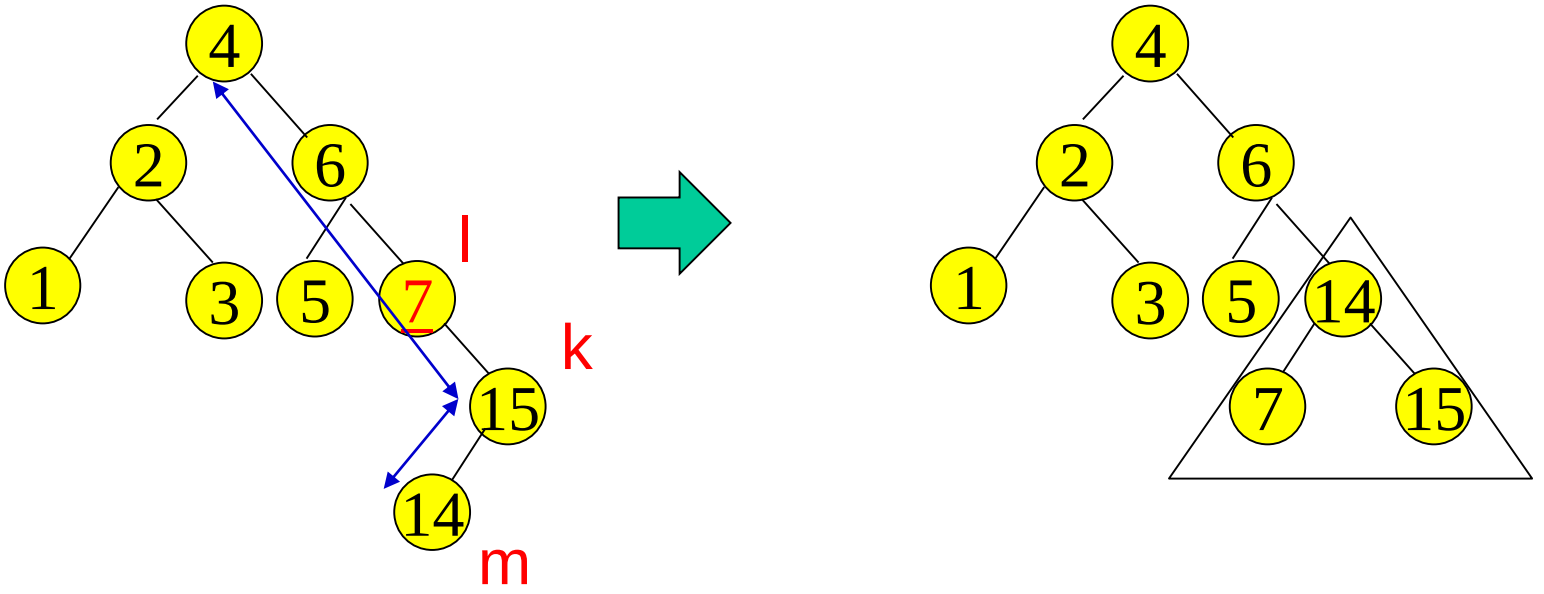
Árboles AVL: **rotación doble**

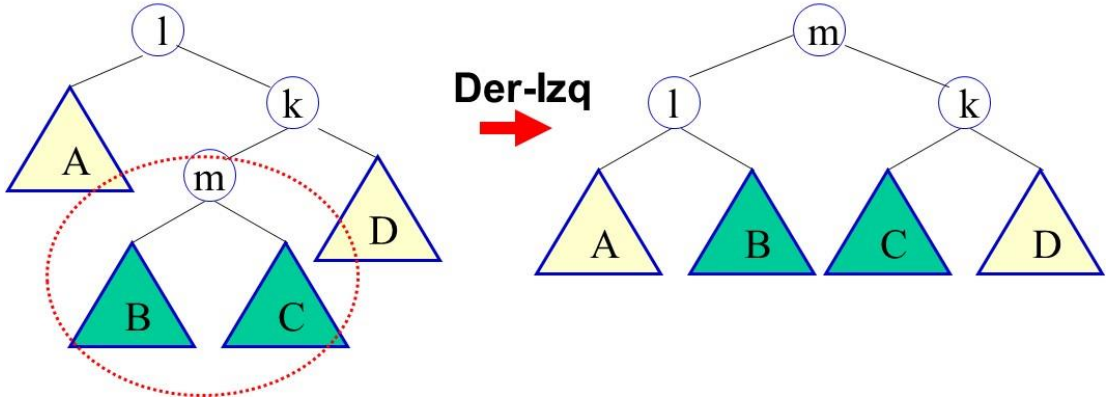


Árboles AVL (cont)

- Inserción del 14.

14)

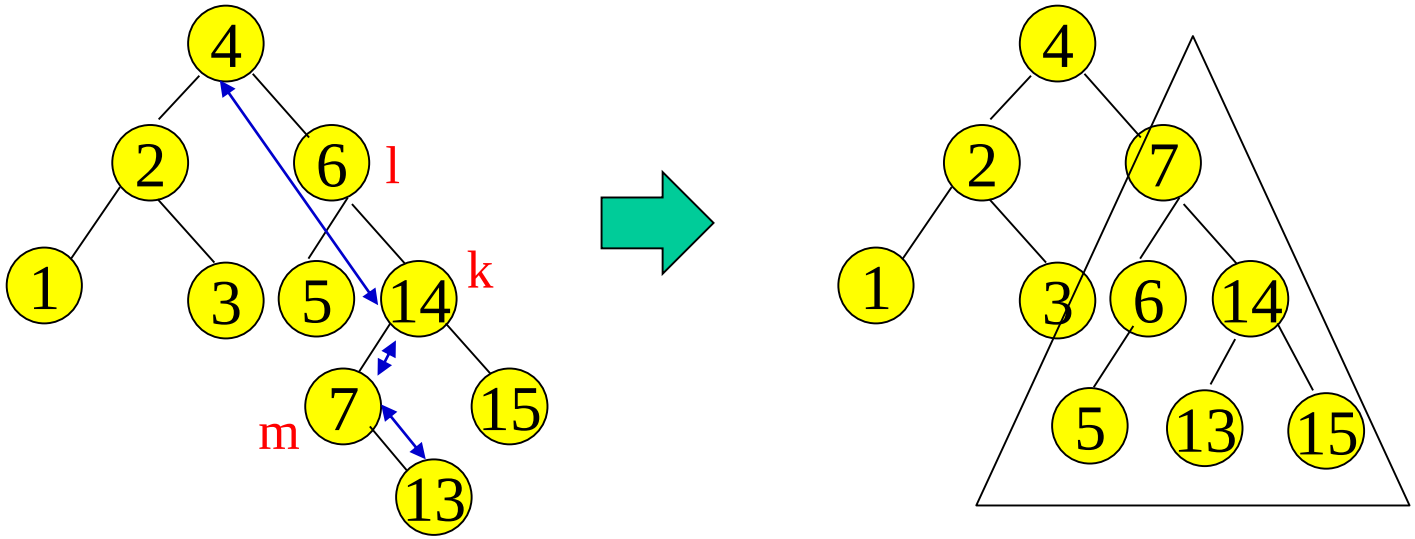


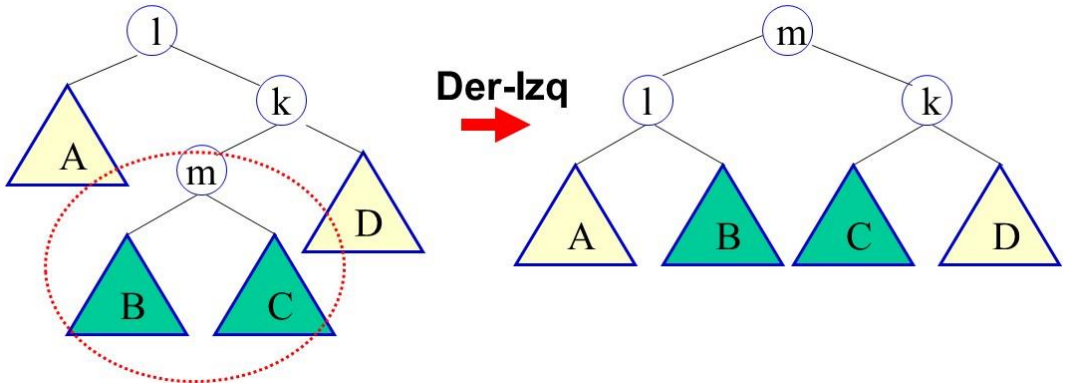


Árboles AVL (cont)

- Inserción del 13.

13)

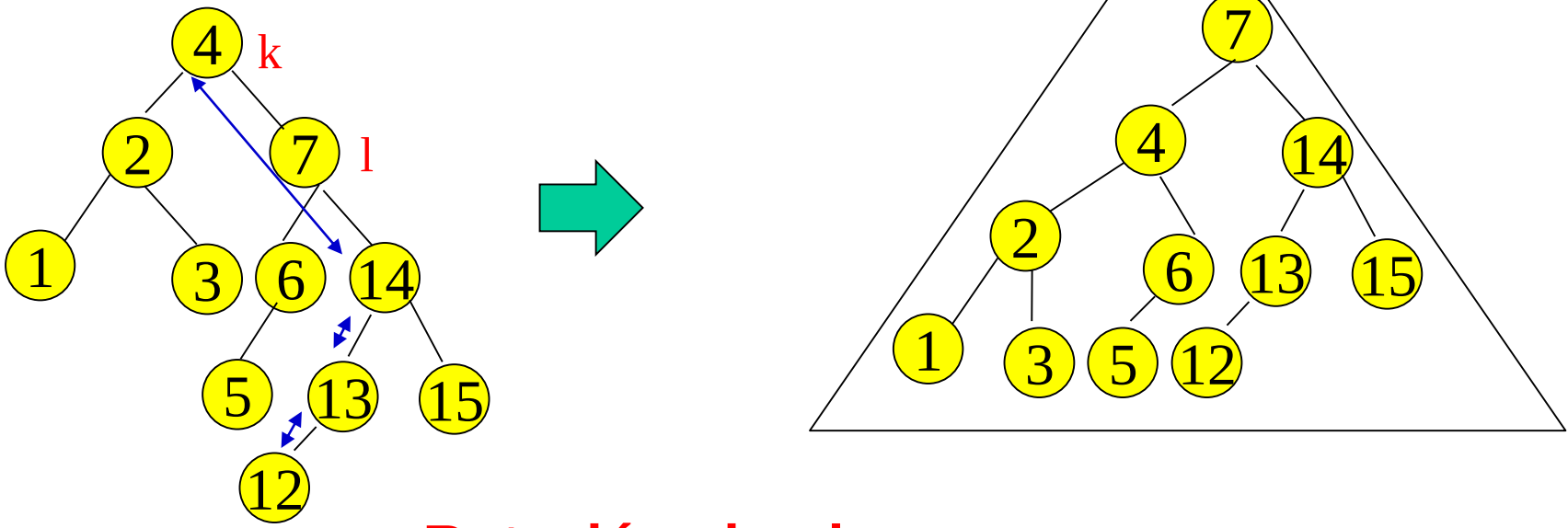




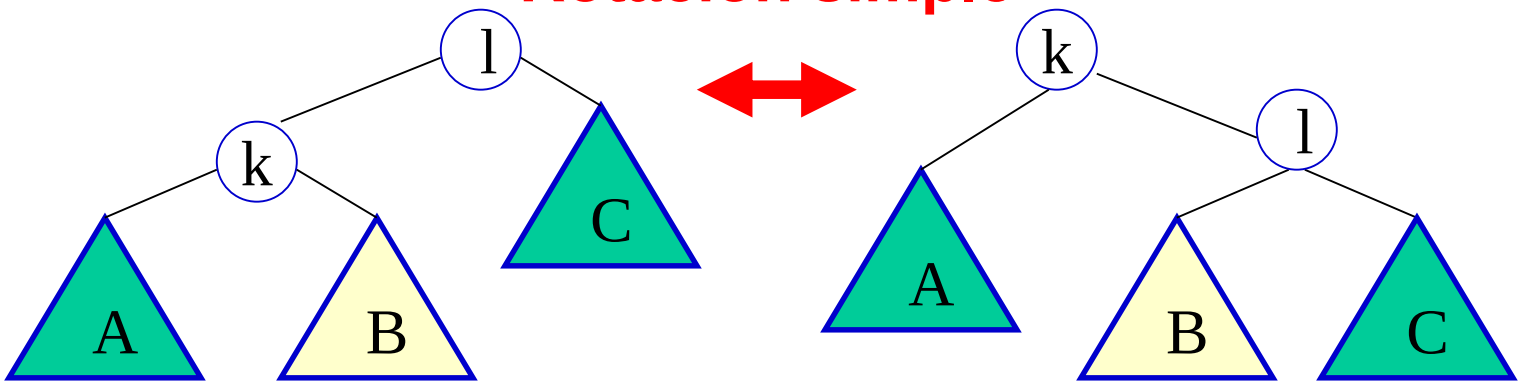
Árboles AVL (cont)

- Inserción del 12.

12)



Rotación simple



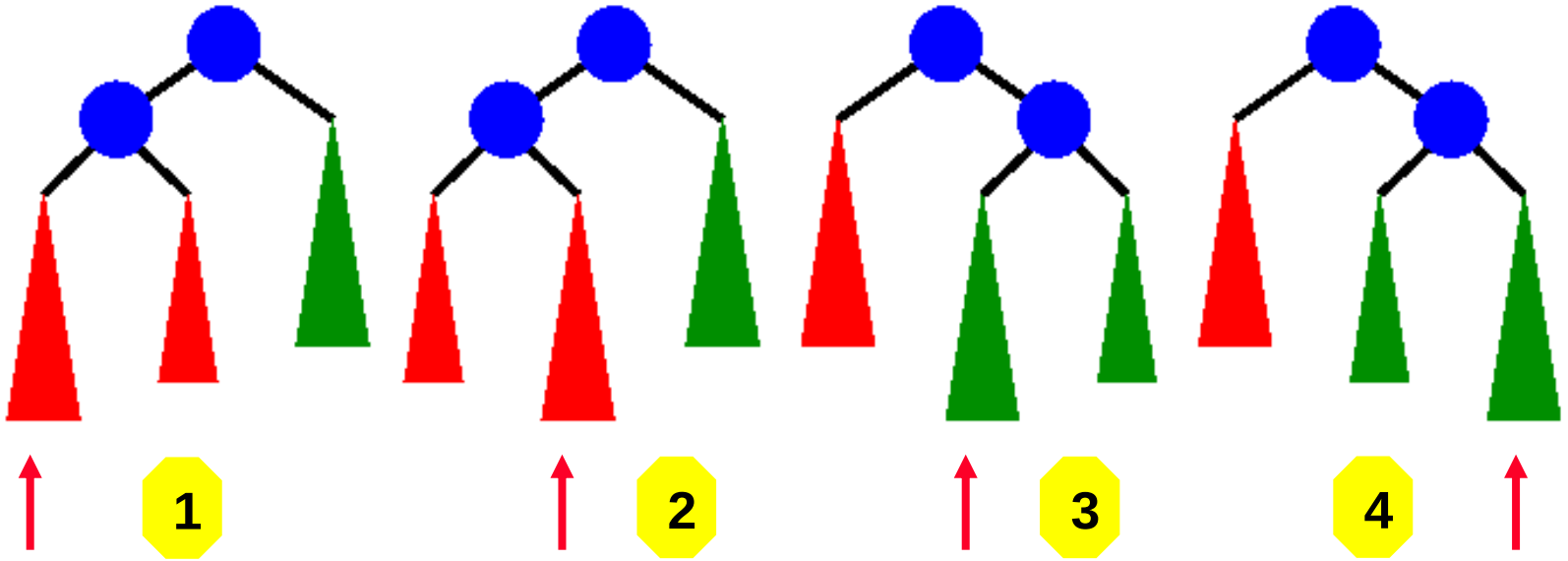
Árboles AVL: inserción

insertar como en los ABB. Luego iniciar en el nodo agregado, subiendo en el árbol y actualizando la información de equilibrio en cada nodo del camino (código luego de las llamadas recursivas). Acabamos si se llega a la raíz sin haber encontrado ningún nodo desequilibrado. Sino, se aplica una rotación (simple o doble, según corresponda) al primer nodo desequilibrado que se encuentre, se ajusta su equilibrio y ya está (no necesitamos llegar a la raíz, salvo que se use eliminación perezosa).

Rebalanceando árboles AVL

Inserciones del tipo ABB que no conducen a un árbol AVL

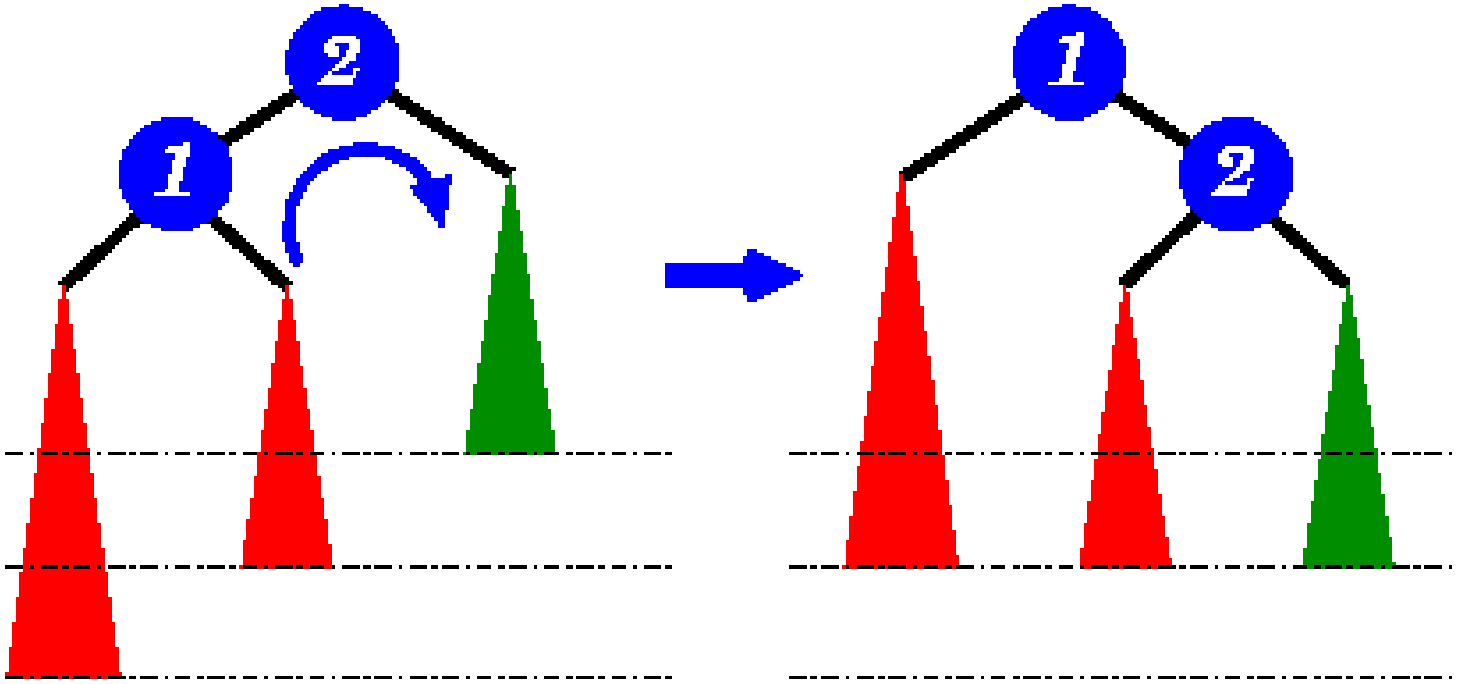
- 4 casos



- 1 y 4 son similares
- 2 y 3 son similares

Rebalanceando árboles AVL

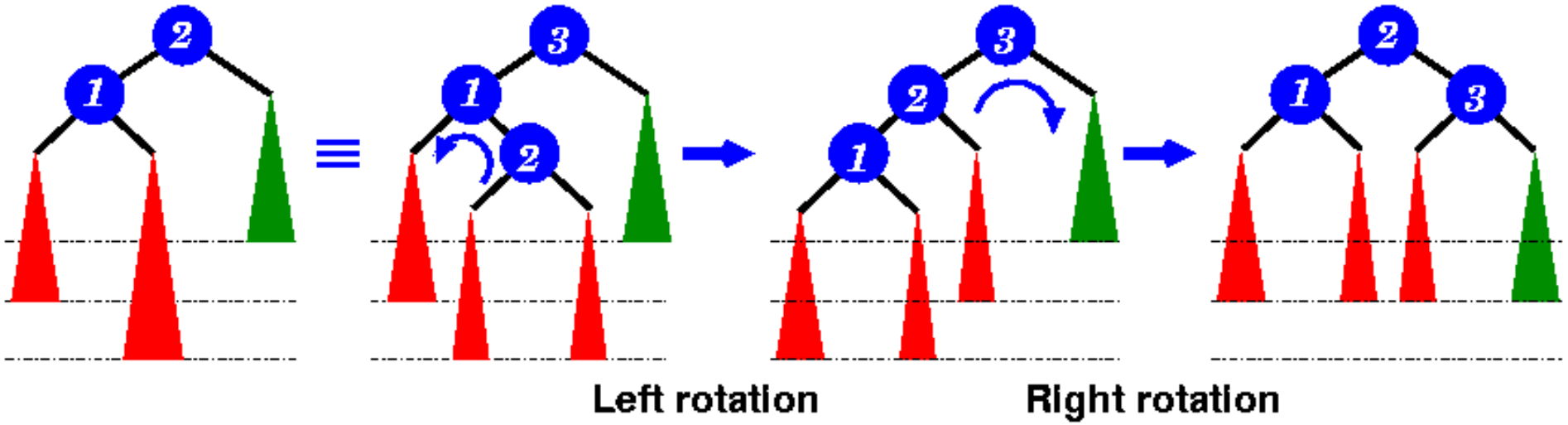
Caso 1 solucionado por rotación simple



– Caso 4 es similar

Rebalanceando árboles AVL

Caso 2 necesita una rotación *doble*



– Caso 3 es similar

Árboles AVL: supresión

- **Algoritmo de supresión**: el algoritmo de eliminación que preserva la propiedad AVL para cada nodo del árbol es más complejo.

  En general, si las eliminaciones son relativamente poco frecuentes, puede utilizarse una estrategia de eliminación perezosa.
- La **pertenencia** es igual que para los ABB.
- Por más detalles y el código de las operaciones para una implementación de AVL con estructuras dinámicas ver el libro de Mark Allen Weiss.