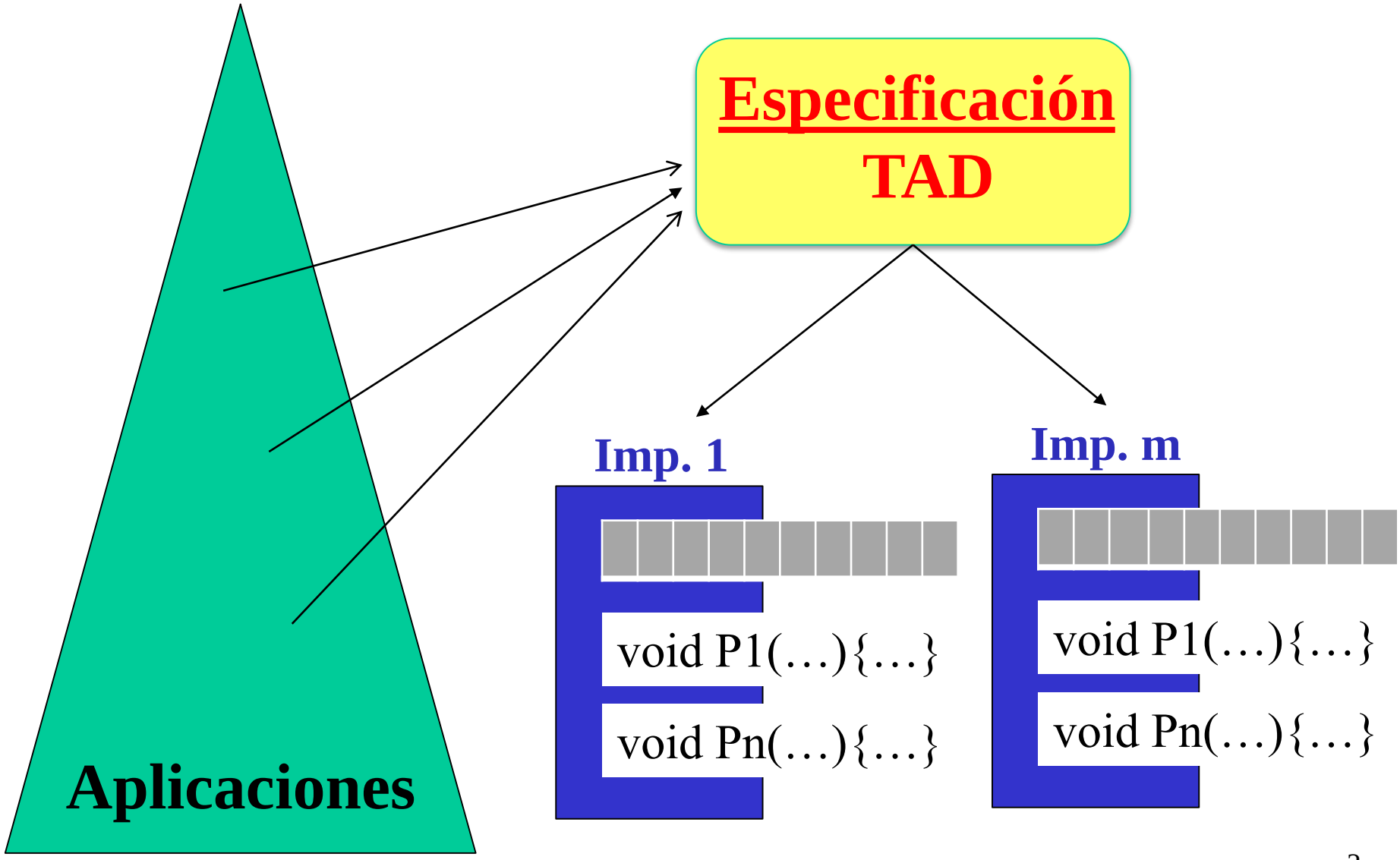
Estructuras de Datos y Algoritmos 1

TADs Colecciones: Conjuntos, Diccionarios, Multiconjuntos y Tablas (*Mappings*) Implementaciones simples y avanzadas

Sobre TADs


Especificación
TAD
Aplicaciones
Imp. 1
void P1(…){…}
void Pn(…){…}
Imp. m
void P1(…){…}
void Pn(…){…}

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**: construye el conjunto 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.**

Especificación del TAD SET

Otras operaciones adicionales que suelen considerarse para conjuntos son, por ejemplo, las siguientes:

- **Borrar**: dado un elemento lo elimina del conjunto, si es éste pertenece al mismo. También **Borrar_min** y **Borrar_max** (borran el mínimo y el máximo elem).
- **Min (Max)**: devuelve el elemento menor (mayor) del conjunto. Pre-condición: el conjunto es no vacío. Esta operación requiere un orden lineal sobre los elementos (o sobre sus claves).
- **Igual**: dado un conjunto, devuelve true si y sólo si el conjunto parámetro es igual al cual se le aplica el método. La igualdad es una operación útil y necesaria para la mayoría de los TADs.
- **Inclusiones, Cardinalidad, …**

Implementación del TAD SET

Algunos lenguajes de programación permiten una implementación de este tipo abstracto, con ciertas restricciones, de forma directa.

Por ejemplo, Modula-2 y Pascal. En estos lenguajes, el constructor de tipos SET OF < tipo-base> permite definir el tipo de todos los posibles conjuntos de elementos de tipo <tipo-base>.

- <tipo-base> debe ser un ordinal.

C y C++ no proveen implementaciones primitivas de este TAD.

Implementaciones del TAD SET

A continuación veremos el TAD Diccionario y analizaremos implementaciones de éstos que son válidas para el TAD Set:

- usando **listas encadenadas** (ordenadas o no);
- por medio de un **arreglo de booleanos (o de bits)**;
- mediante **arreglos con tope**;
- a través de árboles ordenados (**ABBs**, **AVLs**);
- con **tablas de hash**.

El TAD Diccionario

Hay dos TADs especialmente utilizados, basados en el modelo de conjuntos:

Los **diccionarios** y las **colas de prioridad**.

Este último será estudiado en el próximo módulo del curso. Ahora veremos diccionarios.

Cuando se usa un conjunto en el diseño de un algoritmo podría no ser necesario contar con operaciones de unión o intersección o diferencia.

El TAD Diccionario

A menudo lo que se necesita es simplemente manipular un conjunto de objetos al que periódicamente se le agregan o quitan elementos. También es usual que uno desee verificar si un determinado elemento forma parte o no del conjunto.

Un Tad Set con las operaciones Vacio, Insertar, EsVacio, Borrar y Pertenece recibe el nombre de ***diccionario***.

Una ***cola de prioridad*** es esencialente un Tad Set con las operaciones Vacio, Insertar, EsVacio, Borrar_Min y Min (Borrar_Max y Max).

El TAD Diccionario

- **Vacio**: construye el diccionario (conjunto) vacío
- **Insertar x d: agrega x a d, si no estaba el elemento en el diccionario**
- **EsVacio** d: retorna true si y sólo si el diccionario d está vacío
- **Pertenece** x d: retorna true si y sólo si x está en d
- **Borrar** x d: elimina a x del diccionario d, si estaba
- **Destruir** d: destruye el diccionario d, liberando su memoria.

Ejercicio: Especificar el Tad Diccionario en C++.

Implementación del TAD Diccionario

Existen distintas posibles implementaciones de este tipo abstracto.

Una primera posibilidad es usando **listas encadenadas** **(ordenadas o no)**

-memoria dinámica-

Otra posible implementación de un diccionario es por medio de un **arreglo de booleanos**. Este enfoque requiere que:

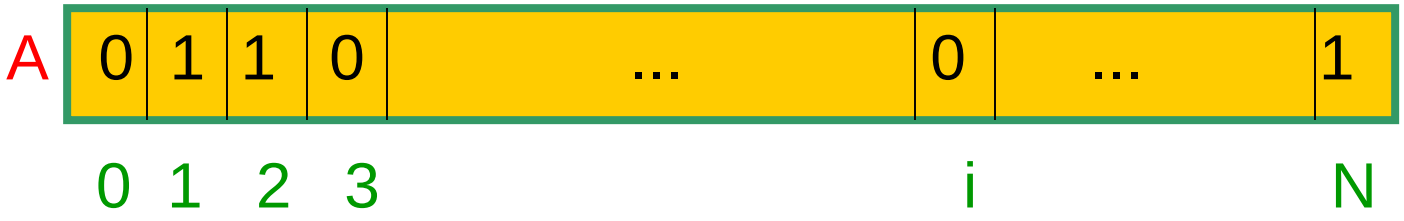
- los elementos del tipo base pertenezcan al subrango 0..N, para algún natural N, o
- que pueda definirse una correspondencia uno a uno entre el tipo base y el anterior subrango de naturales.

Implementación del TAD Diccionario

Una tercer posible implementación es usar un **arreglo con tope**. Este enfoque garantiza una representación correcta siempre y cuando el tamaño del diccionario no supere el largo del arreglo.

adicionalmente, una frecuentemente usada implementación es mediante **ABB** (o **AVL**). Este enfoque permite una eficiente definición de las operaciones y utilización del espacio de memoria.

Implementación de Diccionarios y Conjuntos con arreglos de booleanos

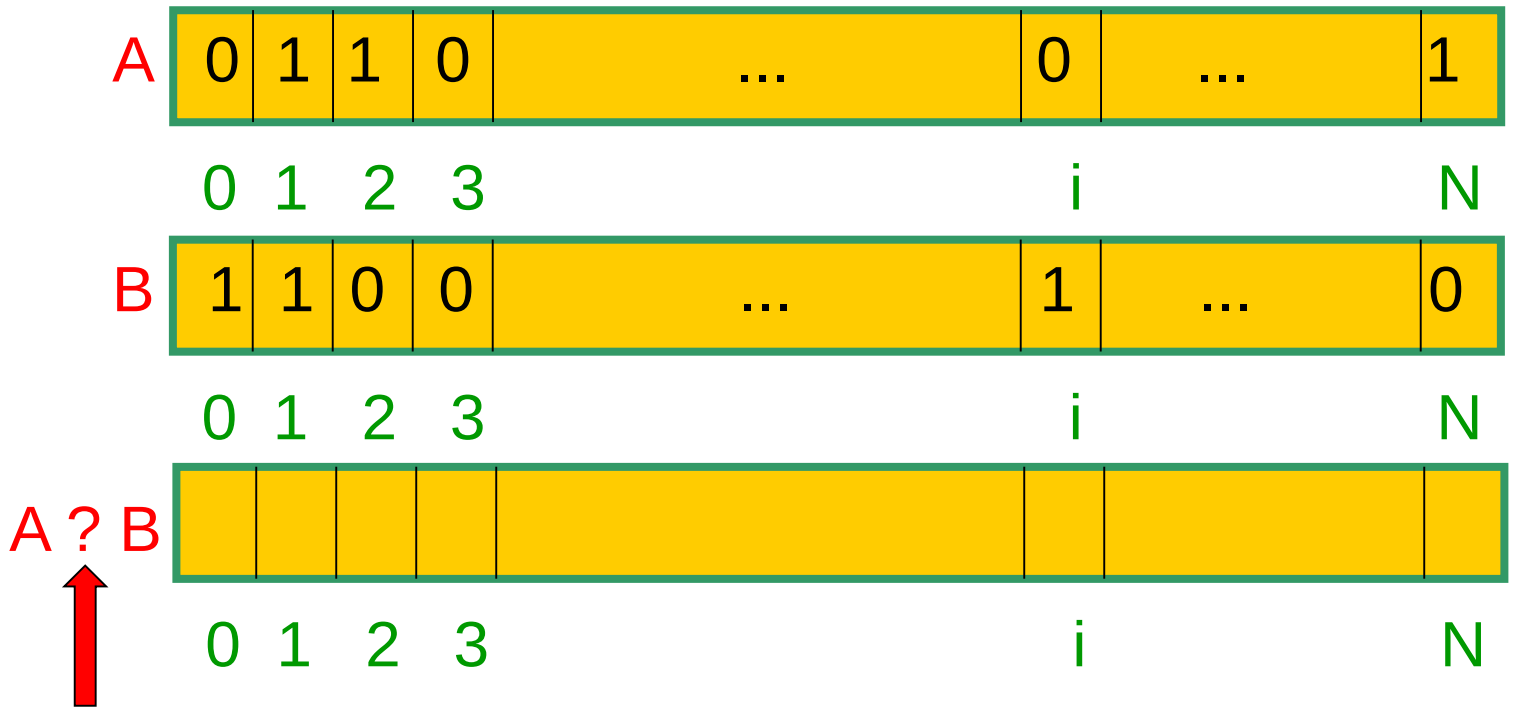


x pertenece al conjunto (diccionario) $\Leftrightarrow$ A[x]=1

- Las operaciones Insertar, Borrar y Pertenece se pueden implementar mediante una referencia directa al elemento (tiempo constante, O(1)).
- ¿ Vacio y EsVacio ?
- ¿ Qué incluye la representación del TAD para arreglos de boleanos ? Esto es, ¿ Cuáles son los campos del struct para esta representación ?

Implementación de Diccionarios y 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 Diccionarios y Conjuntos con arreglos de booleanos

- Esta implementación requiere considerar un conjunto universal finito.

  Los conjuntos (o diccionarios) son subconjuntos de este conjunto universal.

  Naturalmente esta implementación se adecua a una versión acotada de Diccionarios y Conjuntos.

Implementación de Diccionarios y 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 Diccionario. Analizar el tiempo de ejecución de las operaciones.

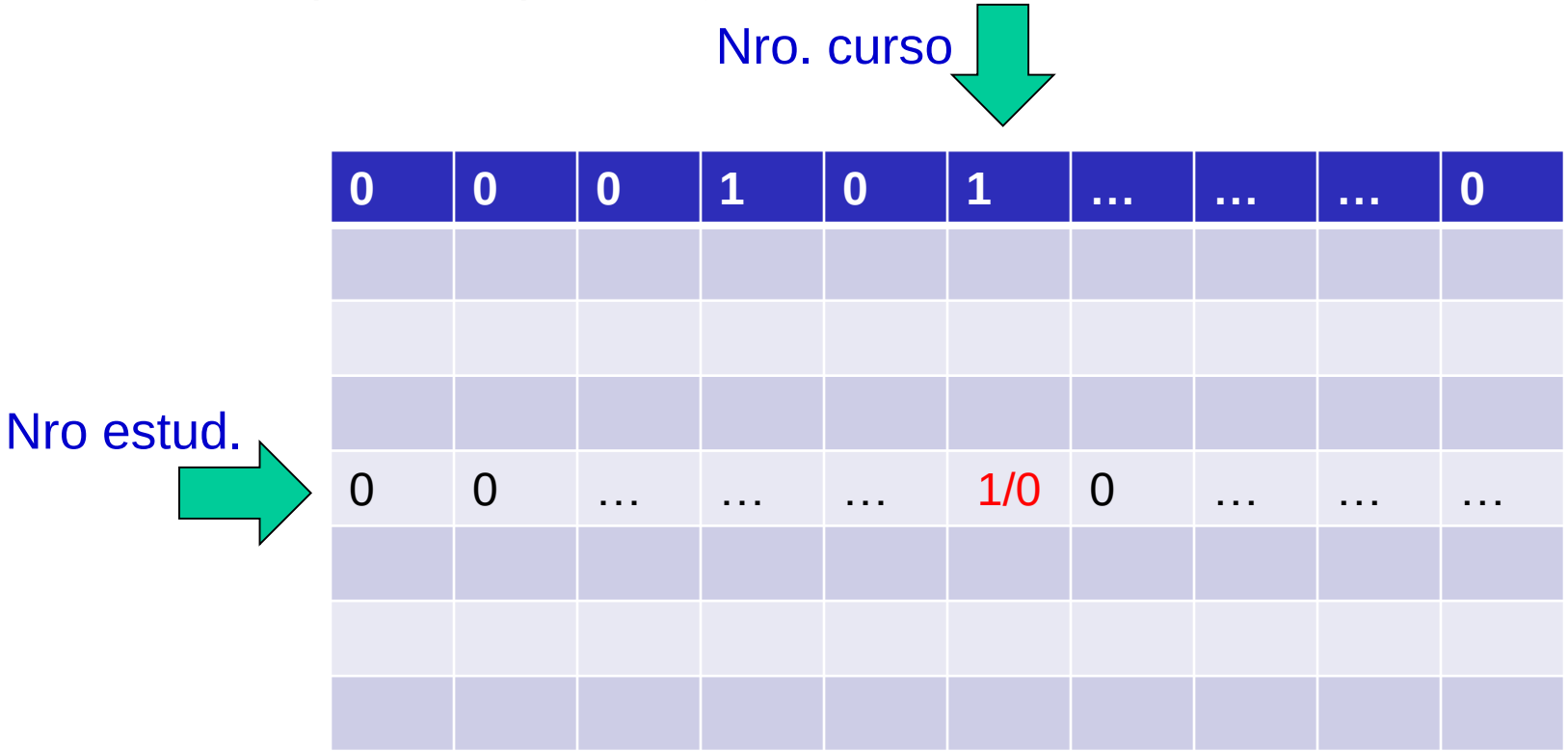
  ¿Qué ocurre con el espacio de almacenamiento?

Implementación de Diccionarios y Conjuntos con arreglos de booleanos

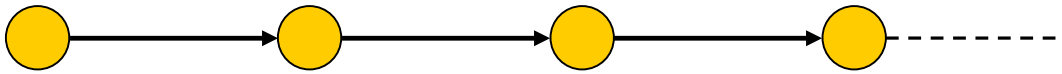
¿Qué ocurre con el espacio de almacenamiento?

¿Se reserva la memoria mínima necesaria?

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



Implementación de Diccionarios y Conjuntos con listas encadenadas



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

Implementación de Diccionarios y Conjuntos con listas encadenadas

- Más aún, ésta representación es más general, puesto que puede manejar conjuntos (diccionarios) que no necesitan ser subconjuntos de algún conjunto universal finito.

  Esta implementación se adecua bien a una versión NO acotada de Diccionarios y Conjuntos.

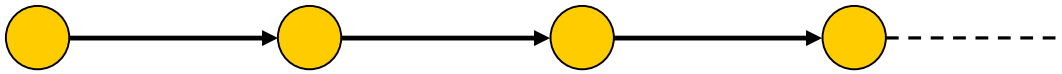
- Las listas pueden ser ordenadas o no.

  El tiempo de ejecución de las operaciones cambia según esta elección.

  Por ejemplo, para Sets, la intersección, la unión y la diferencia sobre listas no ordenadas requieren...

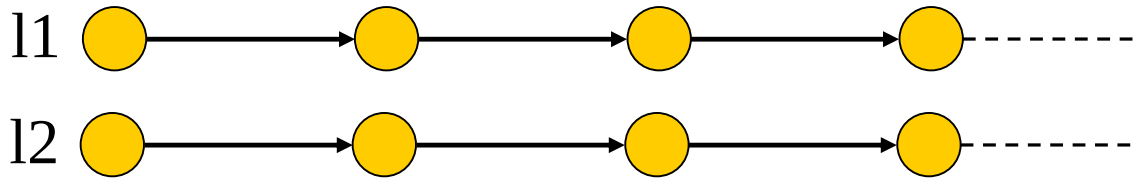
Implementación de Diccionarios y Conjuntos con listas encadenadas

- Las operaciones Insertar, Suprimir y Pertenece para listas ordenadas y no ordenadas requieren en el peor caso O(n), con n el tamaño del conjunto (diccionario). Y ¿en el caso promedio?



Implementación de Diccionarios y 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
  - l1 = [2,3,7,9,18]; l2 = [1,3,4,7,18,99]
  - l1 U l2 = [1,2,3,4, …]

Implementación de Diccionarios y Conjuntos con listas encadenadas

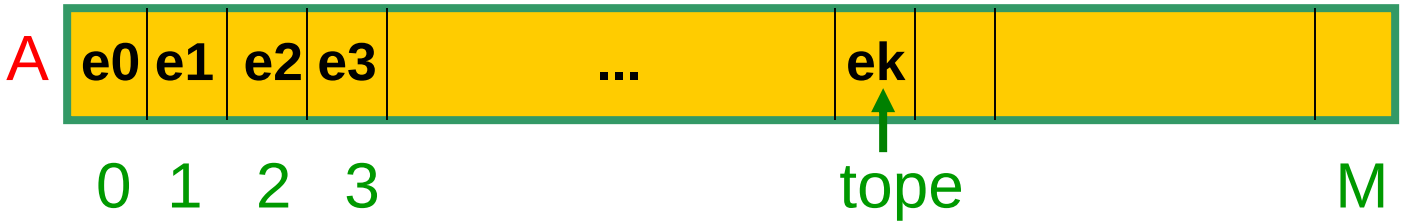
¿Es posible hacer búsqueda binaria (por bipartición) sobre listas encadenadas ordenadas, de manera de garantizar O(log (n))?

Implementación de Diccionarios y Conjuntos con listas encadenadas

- Para las operaciones Min, Max, Borrar_Min y Borrar_Max, ¿es igualmente eficiente usar listas ordenadas o no ordenadas?
- Para implementar diccionarios, conjuntos o colas de prioridad con listas, ¿qué debe incluir la representación del TAD?

Ejercicio: Desarrollar algoritmos para los TADs Diccionario, Conjunto y Cola de prioridad, luego de escribir sus especificaciones. Analizar luego sus órdenes de tiempo de ejecución en el peor caso para las implementaciones analizadas previamente.

Implementación de Diccionarios mediante arreglos con tope



- Esta realización sólo 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 Diccionarios y Conjuntos.
- Tiene la ventaja de la sencillez sobre la representación con listas encadenadas.

Implementación de Diccionarios mediante arreglos con tope

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

  (1) el diccionario no puede crecer arbitrariamente,

  (2) la operación de borrado es dificultosa y

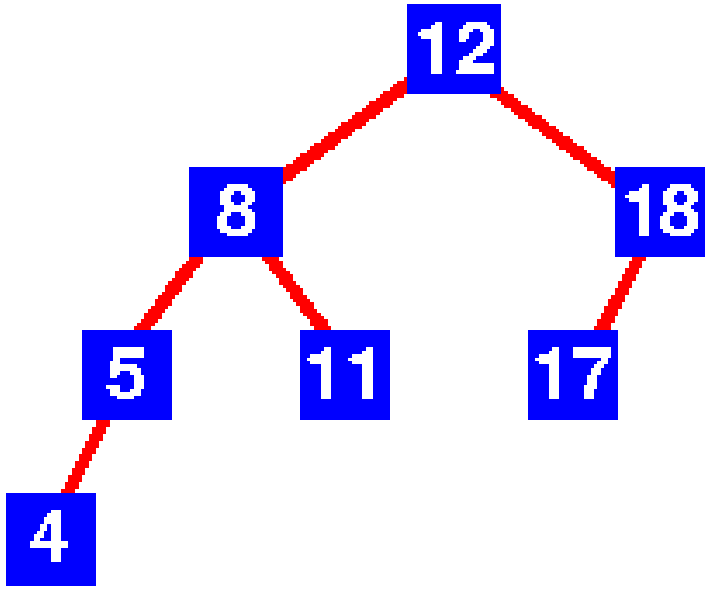
  (3) el espacio de almacenamiento es ineficientemente utilizado.

Implementación de Diccionarios con ABB

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

  Esta implementación se adecua a una versión NO acotada de Diccionarios y Conjuntos.

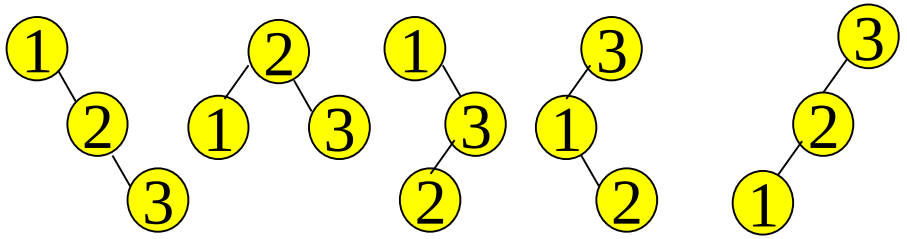
ABB



Diccionario

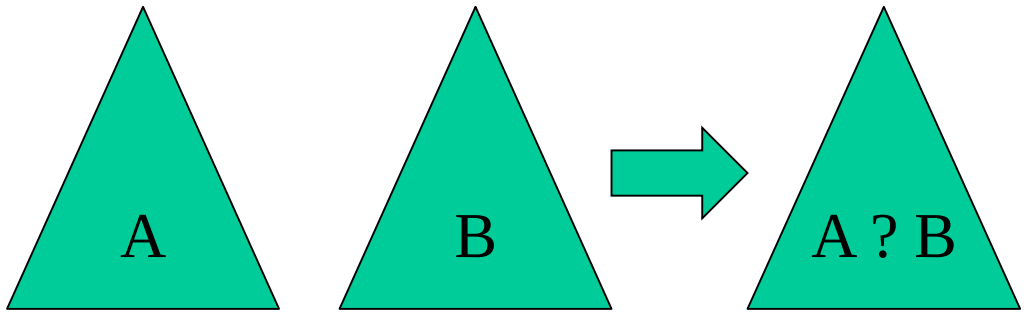
{ 4, 5, 11, 17, 8, 18, 12 }

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



Implementación de Diccionarios con ABB

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



Implementación de Diccionarios con ABB

- Si se piensa en Colas de Prioridad, ¿cómo serían las operaciones Min / BorrarMin con ABB?; ¿cuál sería la eficiencia de estas operaciones?
- La longitud de un camino promedio de la raíz de un ABB a una hoja es O(log (n)). Luego, las operaciones de diccionarios sobre ABB requieren O(log (n)) en el caso promedio.
- No obstante, estas operaciones requieren O(n) en el peor caso: las secuencias de supresiones e inserciones pueden transformar un árbol en el árbol degenerado correspondiente a una lista.

Implementación de Diccionarios con ABB

- Para ello se utilizan los AVL (o árboles balanecados) que tienen en el peor caso (y en el promedio) igual complejidad que el caso promedio de los ABB, para las operaciones de diccionarios O(log (n)).

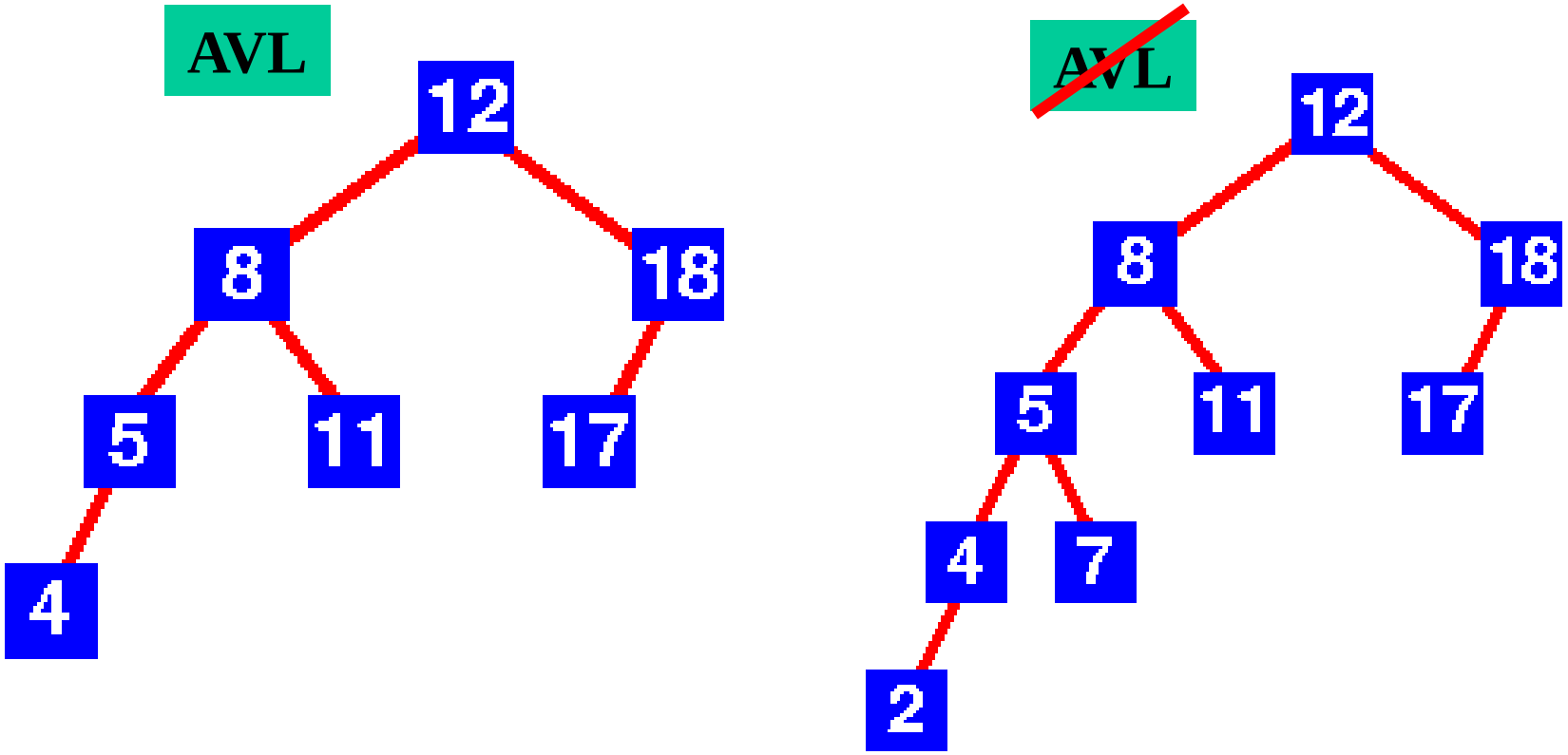
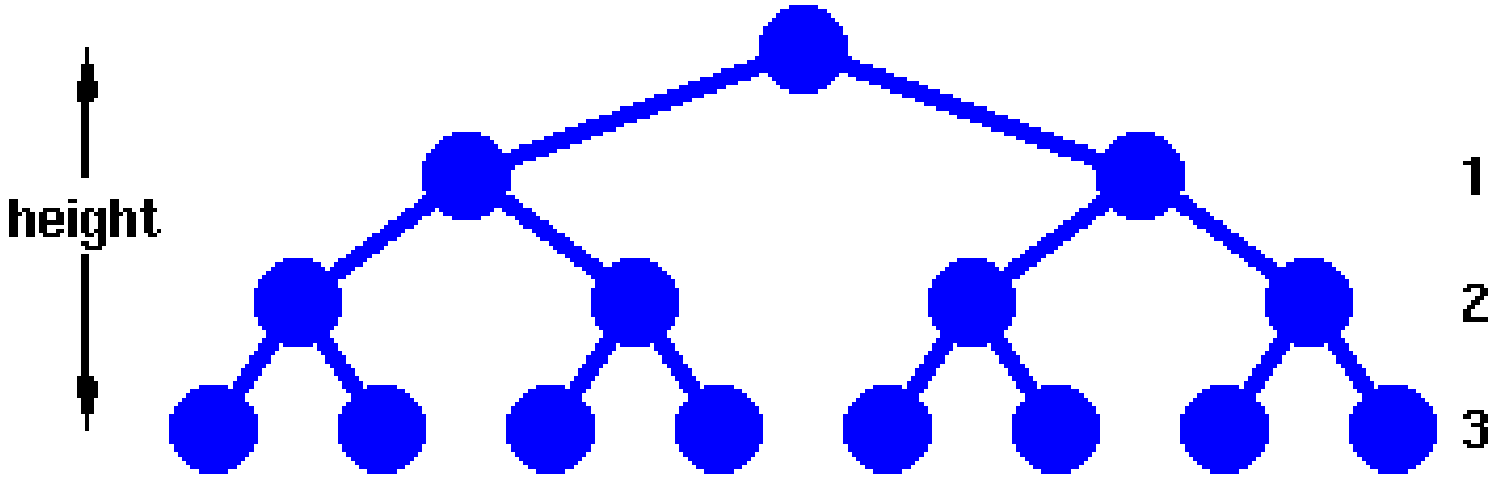


ABB - Análisis

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).

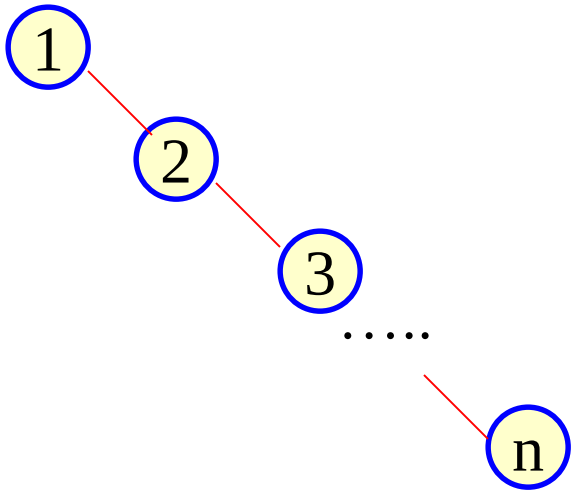


ABB - Análisis

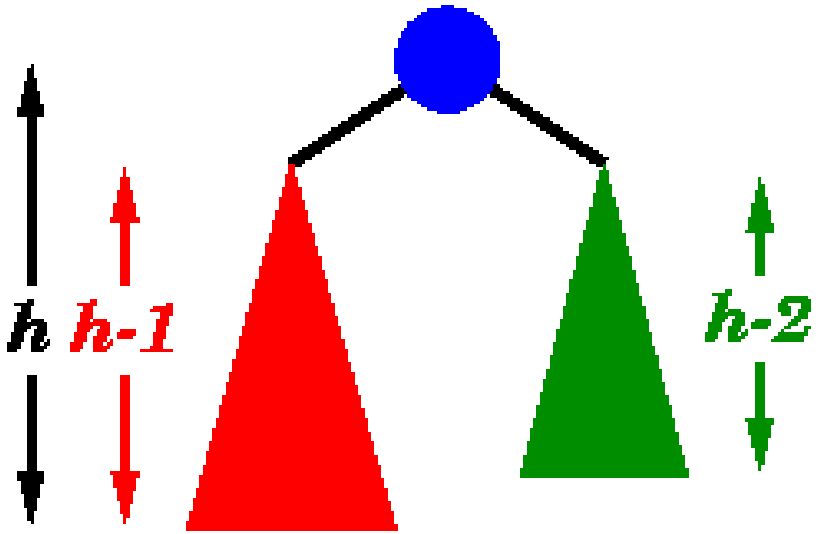
Si bien el orden de tiempo de ejecución promedio de las operaciones citadas para ABB's es $O(\log_2 n)$, el peor caso es $O(n)$.

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

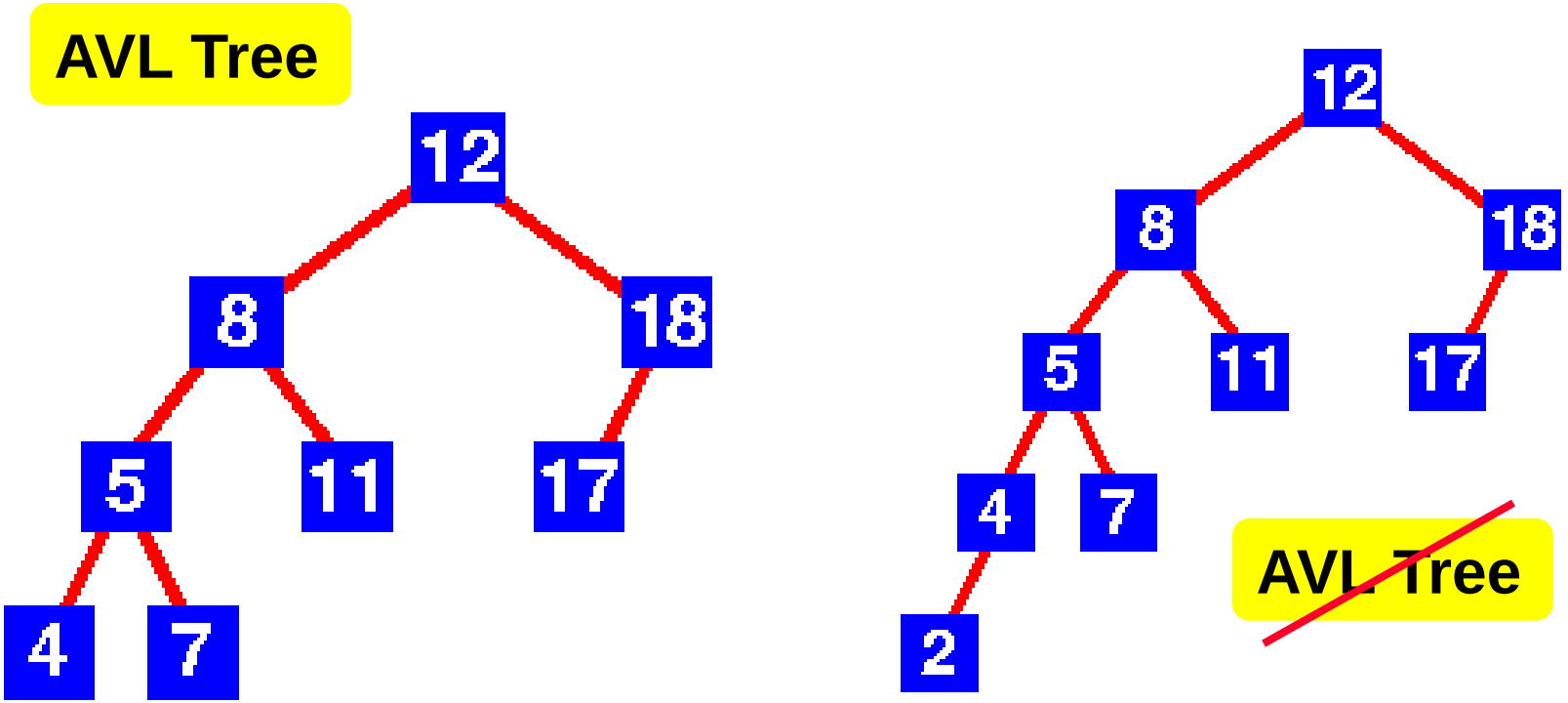
Arboles AVL

- 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).



Arboles 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).



Arboles 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.

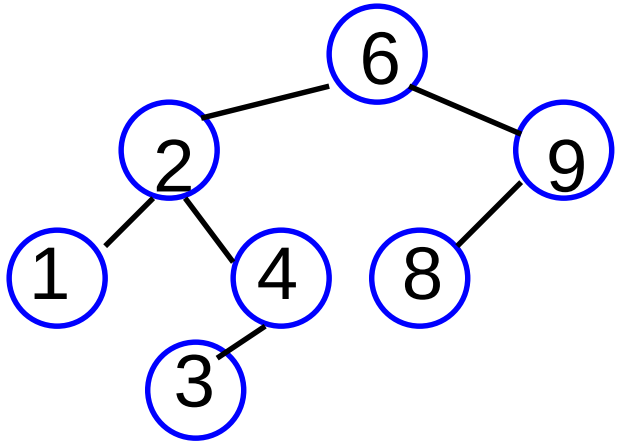


ABB y **AVL**

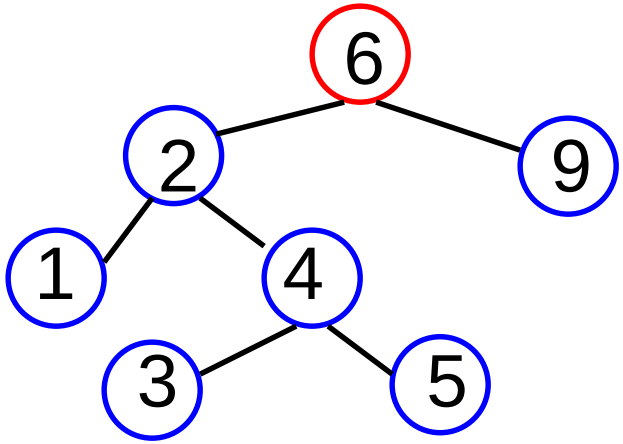


ABB pero **no AVL**

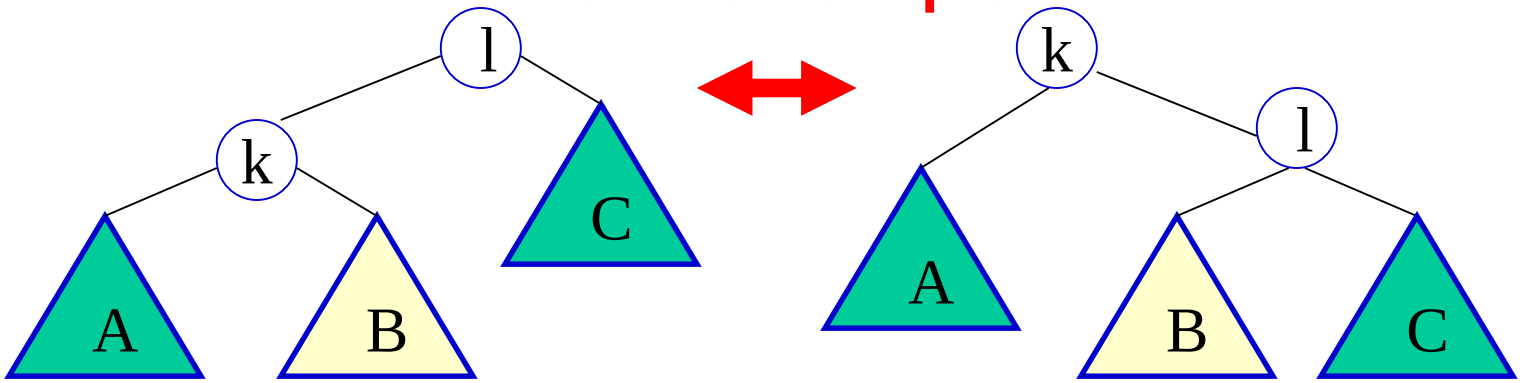
Arboles AVL: rotaciones

- 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).

Arboles AVL (cont)

Rotación simple

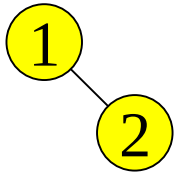


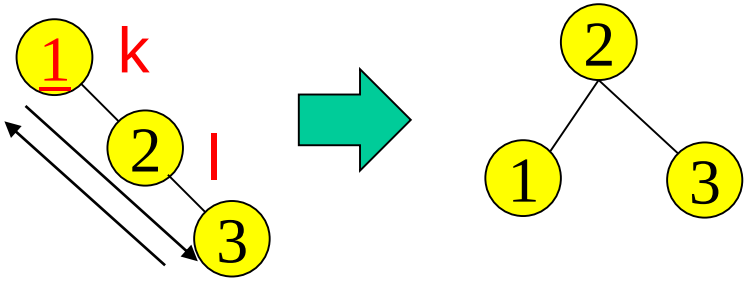
- **Ambos son ABB's ?**
- **Qué implica una rotación con una implementación dinámica del árbol ?**
- **Qué punteros cambian ?**
- **Qué pasa con las alturas de los subárboles A, B y C ?**
- **Coinciden los recorridos In-order de ambos árboles ?**

Arboles AVL (cont)

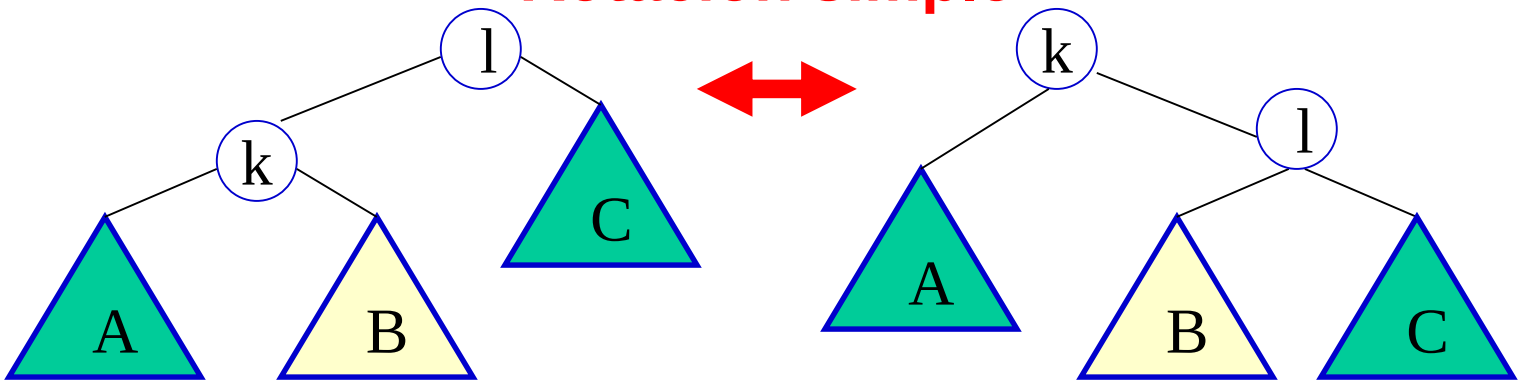
- Insertar los elementos 1 al 7 a partir de un AVL vacio.

1) 

2) 

3) 

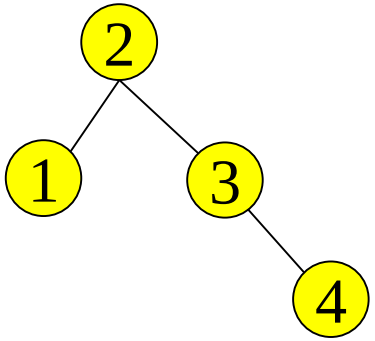
Rotación simple



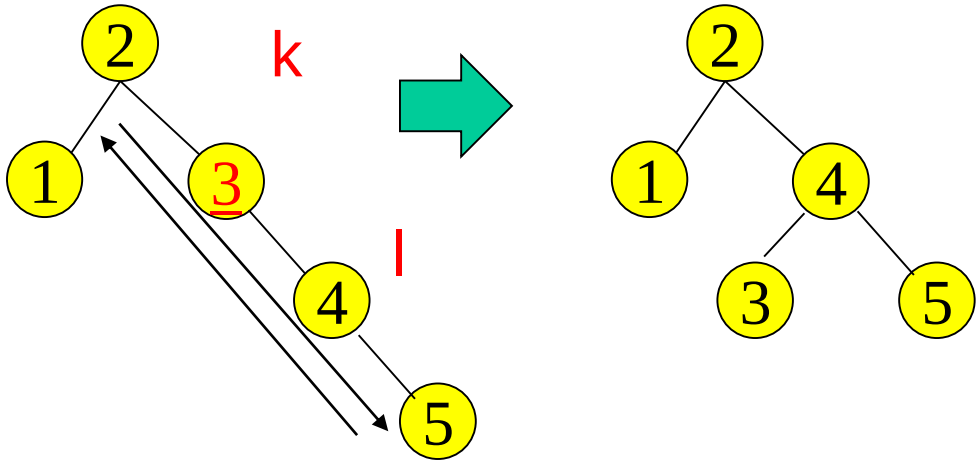
Arboles AVL (cont)

- Inserción del 4 y luego del 5.

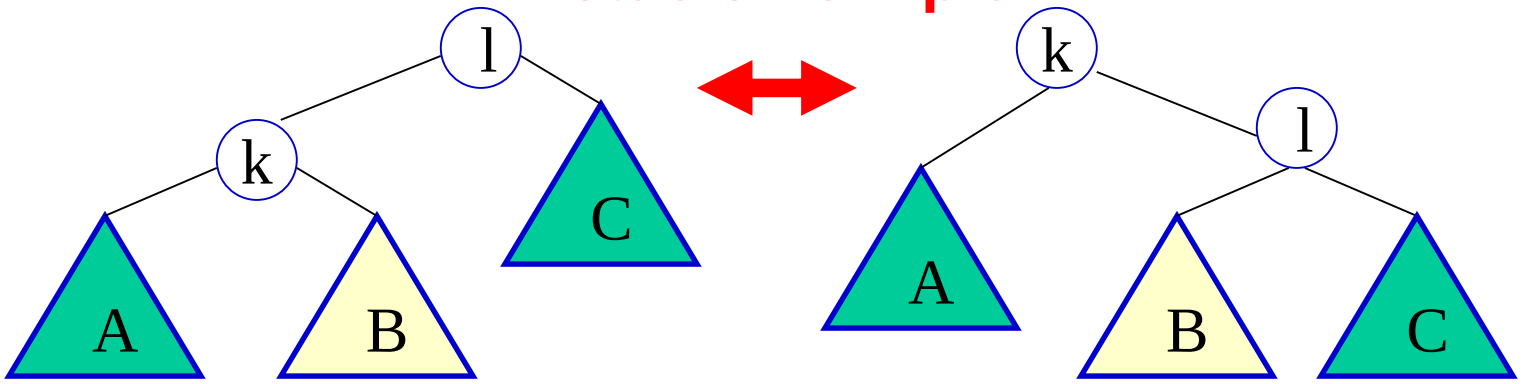
4)



5)



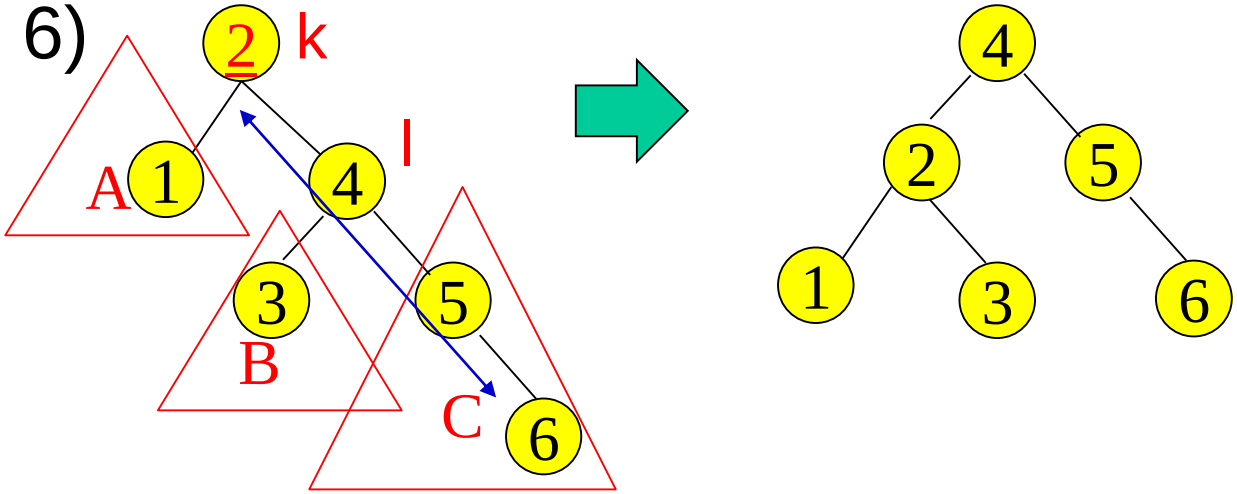
Rotación simple



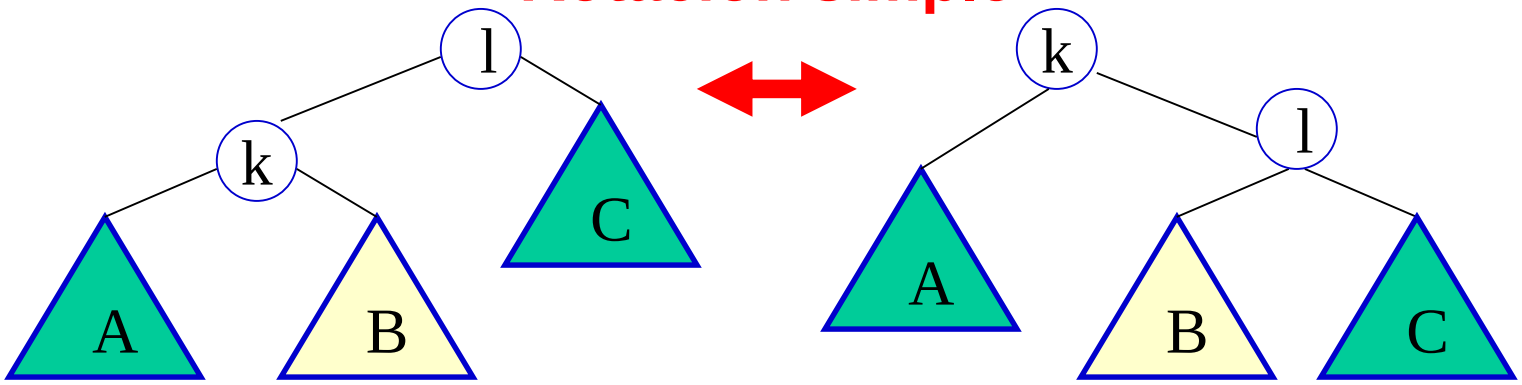
Arboles AVL (cont)

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

6)



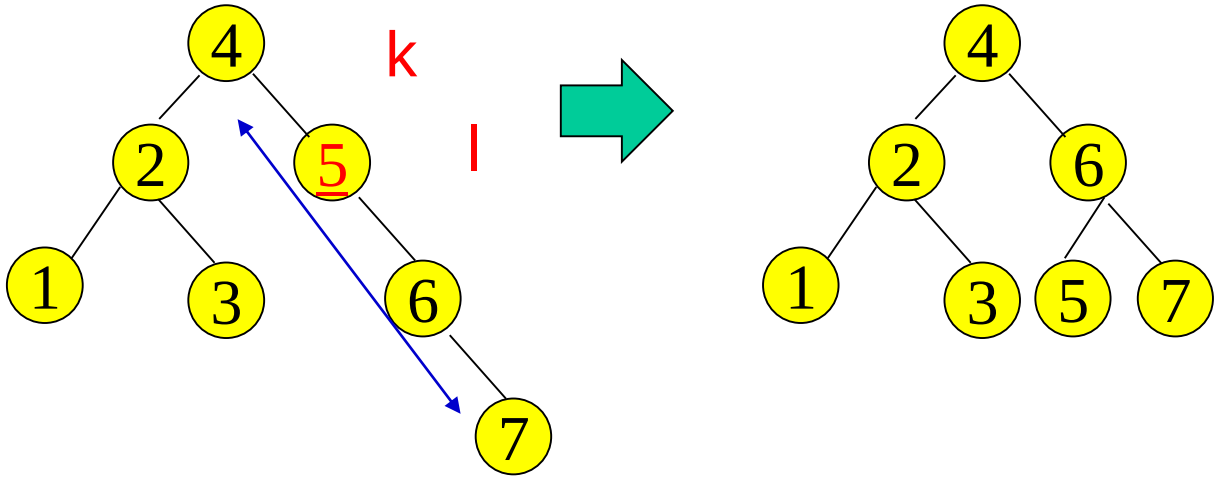
Rotación simple



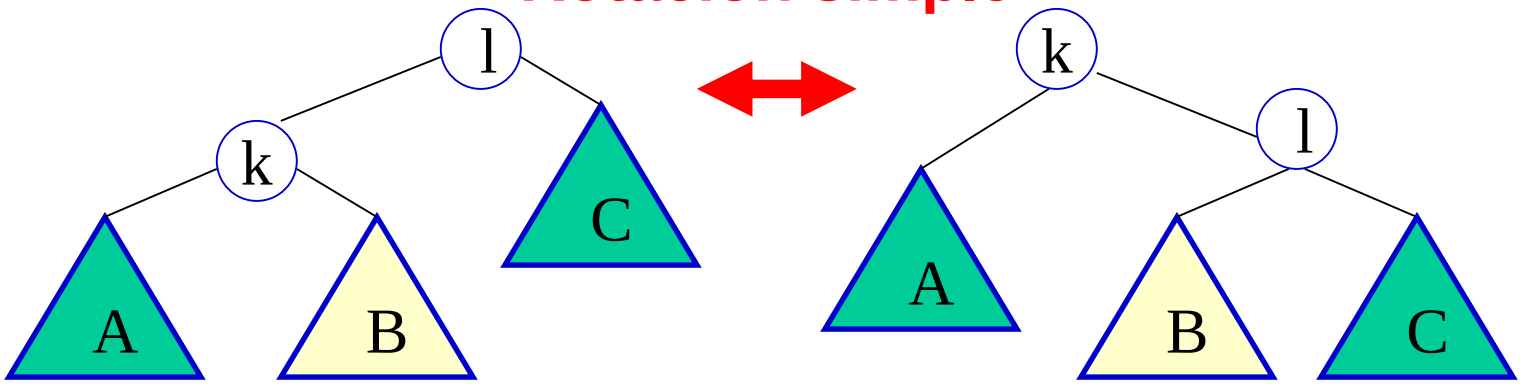
Arboles AVL (cont)

- Inserción del 7.

7)



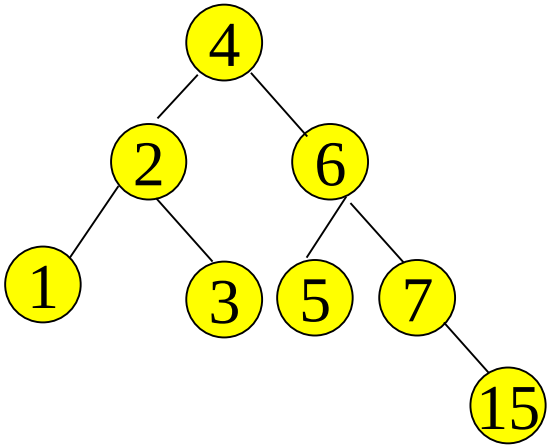
Rotación simple



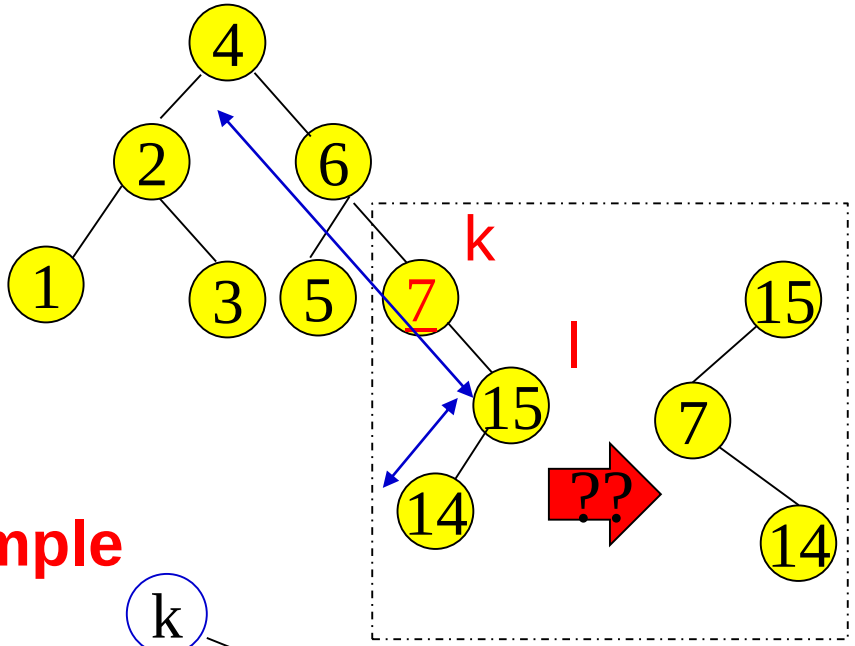
Arboles AVL (cont)

- Ahora insertar los elementos del 15 al 12.

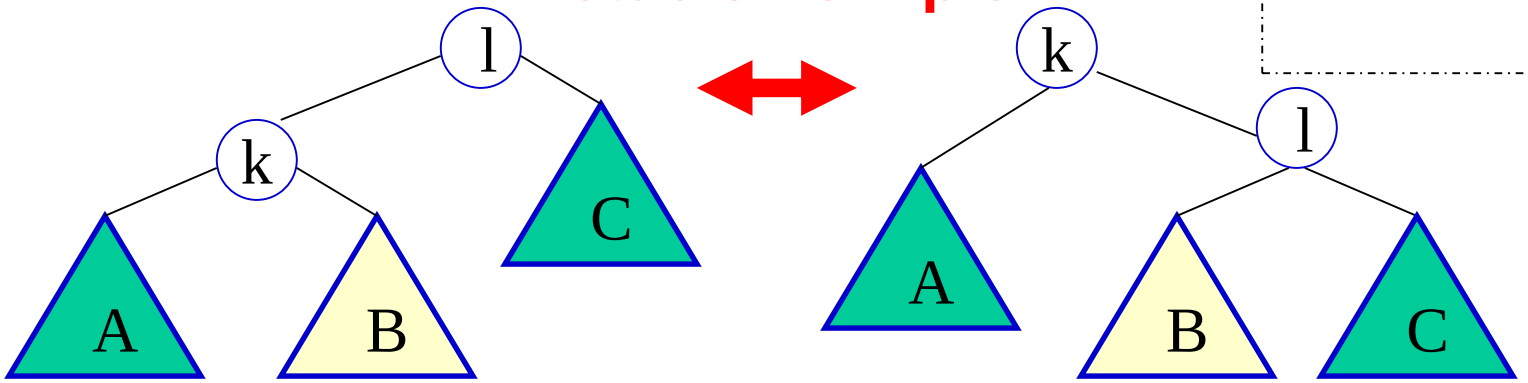
15)



14)



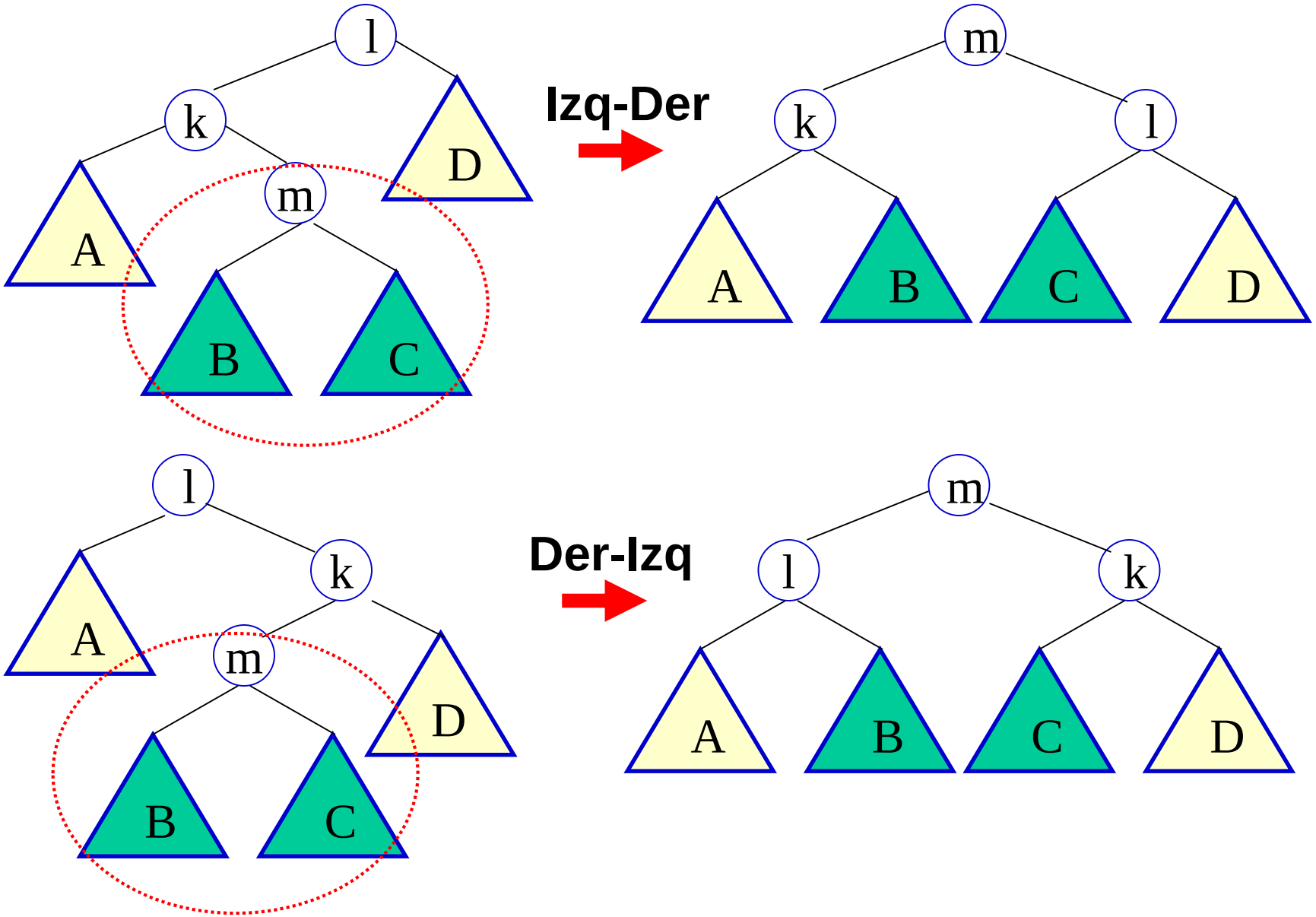
Rotación simple



Arboles 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.

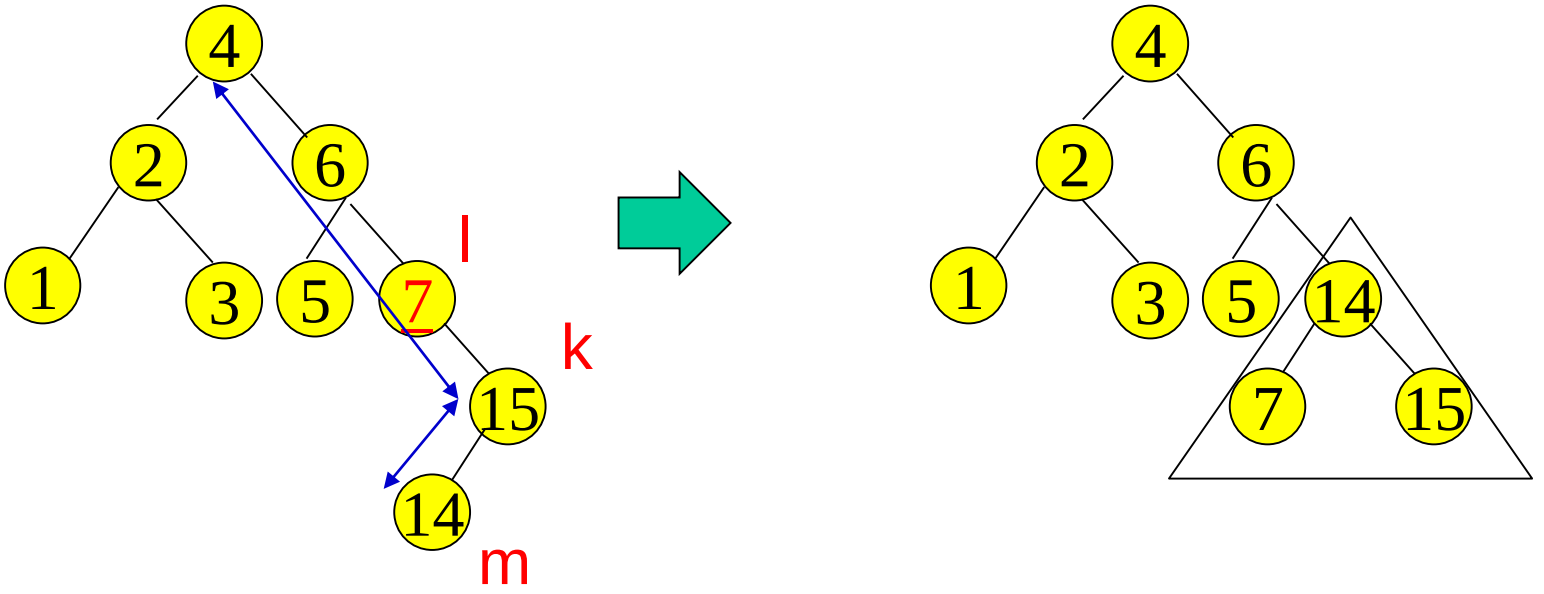
Arboles AVL: **rotación doble**

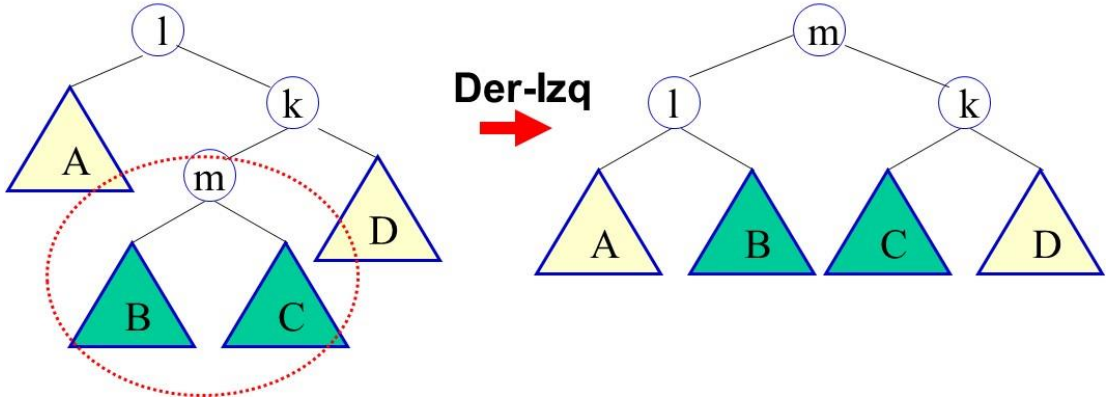


Arboles AVL (cont)

- Inserción del 14.

14)

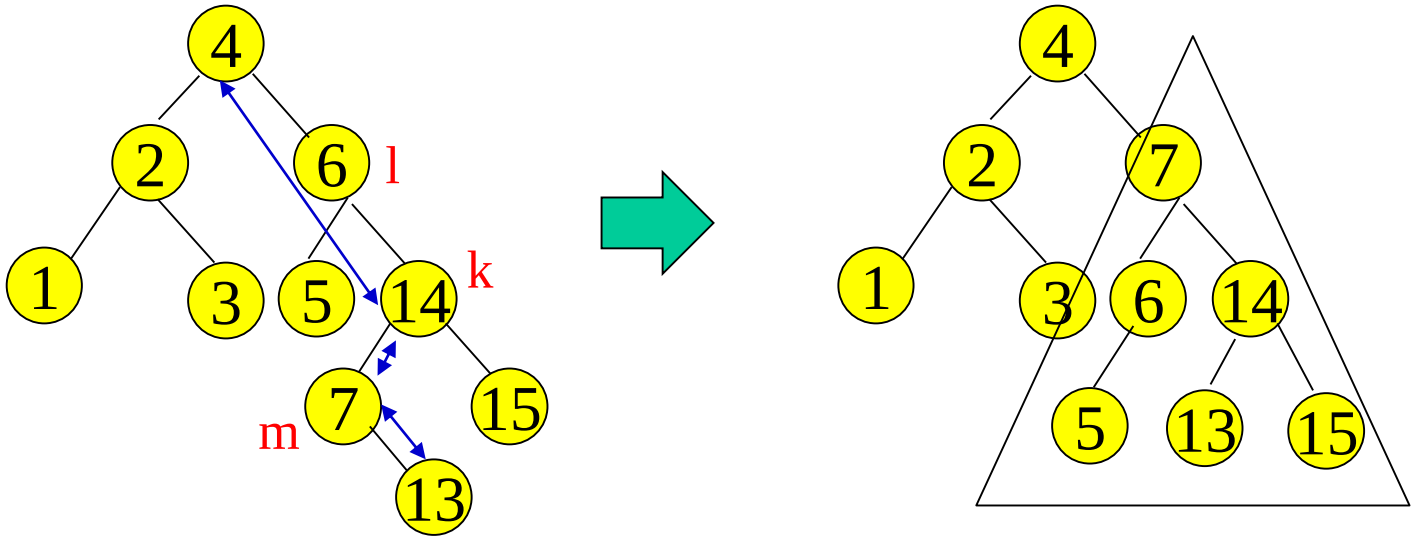


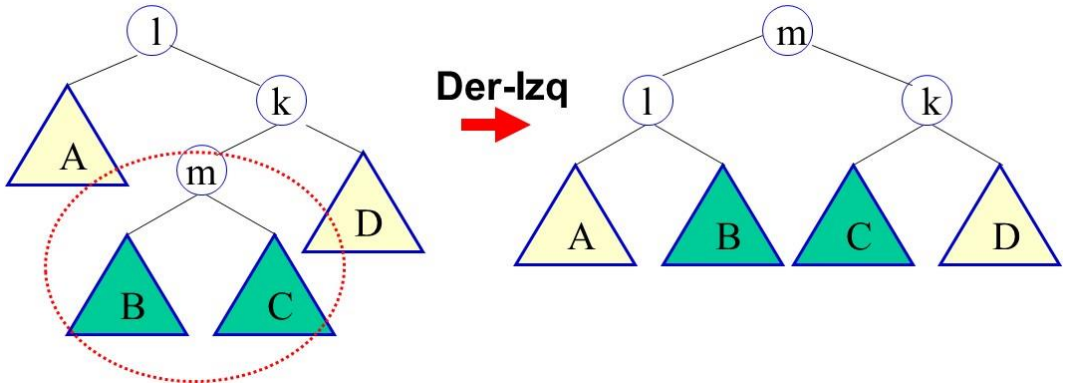


Arboles AVL (cont)

- Inserción del 13.

13)

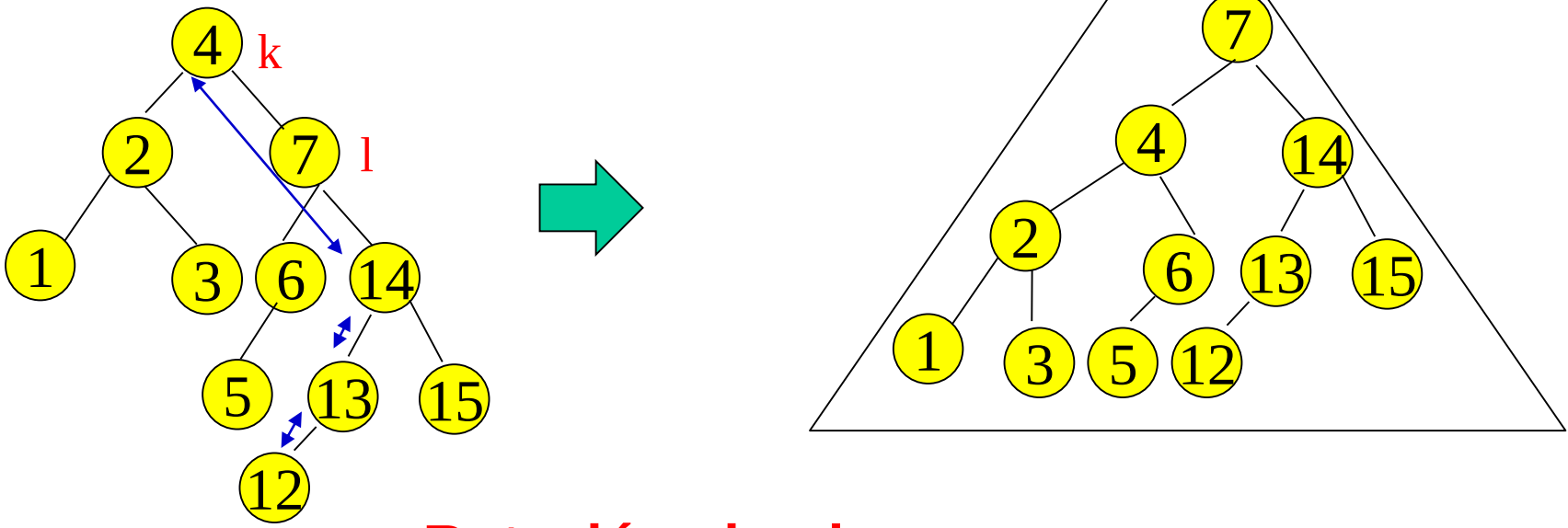




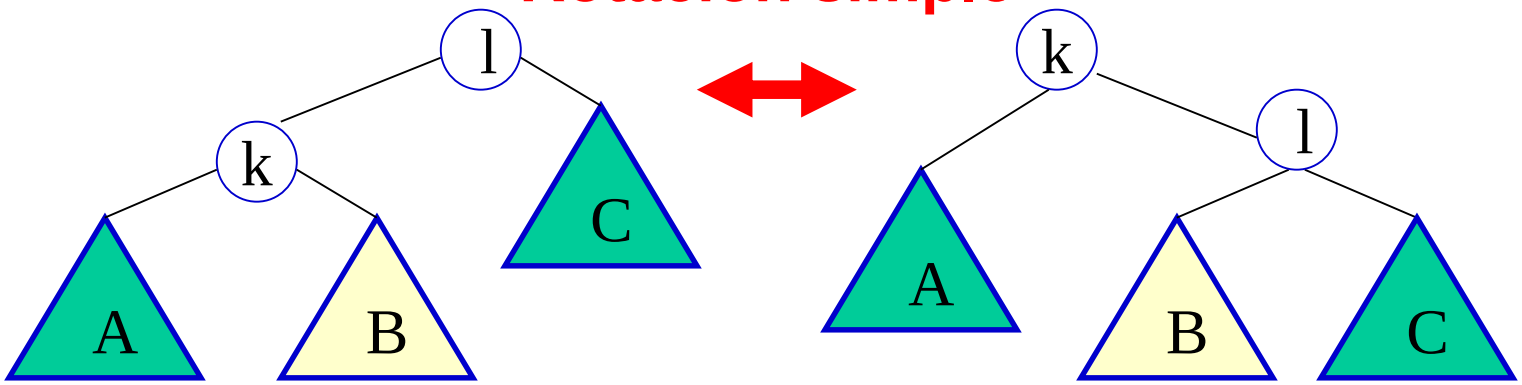
Arboles AVL (cont)

- Inserción del 12.

12)



Rotación simple



Arboles AVL: inserción

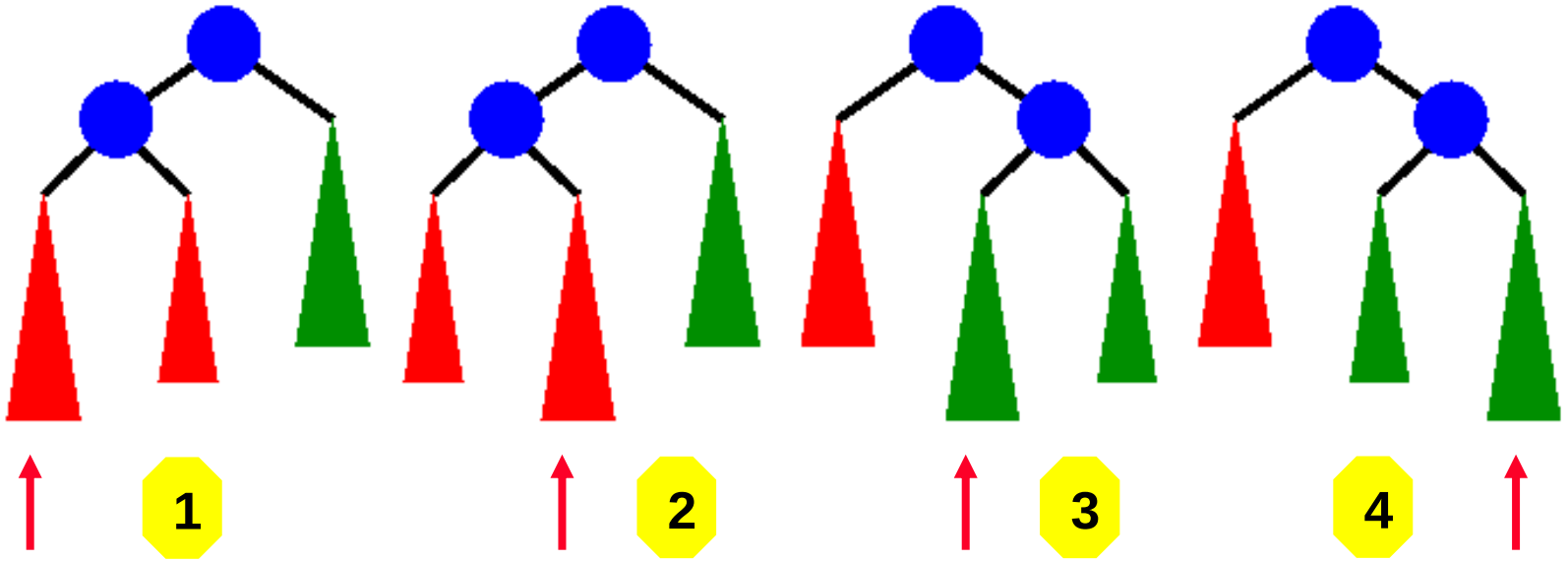
Algoritmo de inserción

insertar como en los ABB y luego iniciar en el nodo insertado y subir en el árbol, actualizando la información de equilibrio en cada nodo del camino. Acabamos si se llega a la raíz sin haber encontrado ningún nodo desequilibrado. Si no, 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

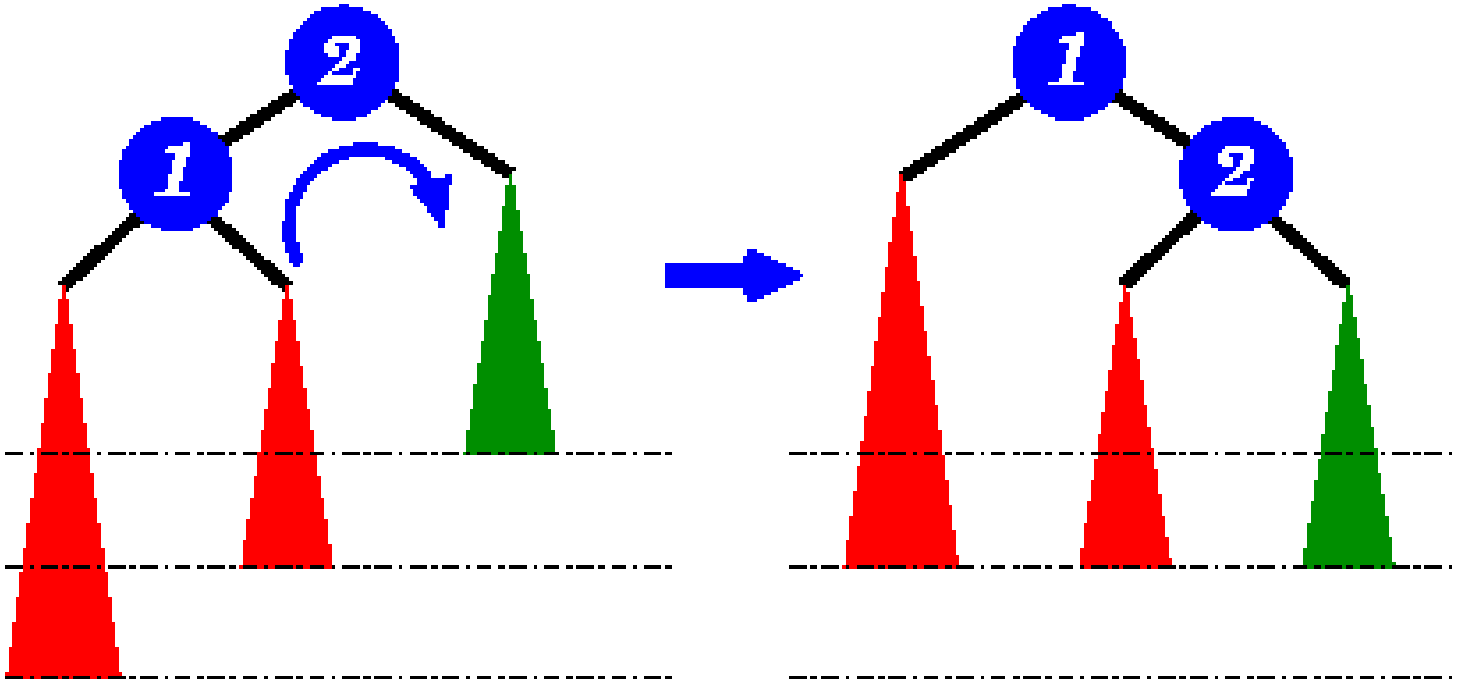
- 4 casos



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

Rebalanceando árboles AVL

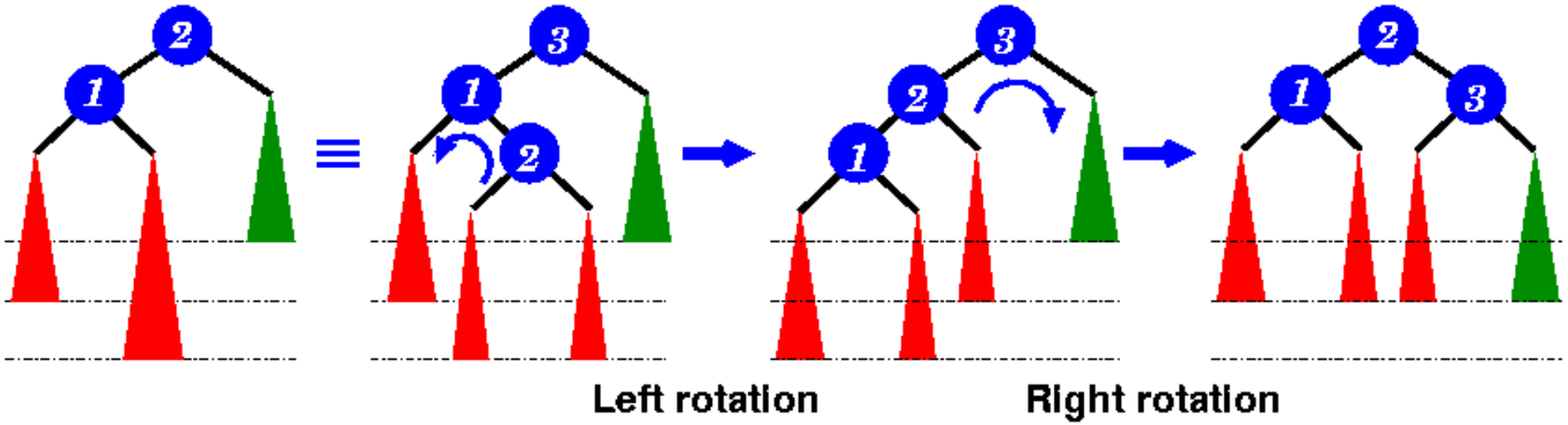
Caso 1 solucionado por rotación simple



– Caso 4 es similar

Rebalanceando árboles AVL

Caso 2 necesita una rotación *doble*



– Caso 3 es similar

Arboles 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.

Ejercicios sobre ABB's y AVL's

Sugeridos:

Implementar el TAD Diccionario y el TAD Conjunto usando ABB's y AVL's.

Analizar la eficiencia en el peor caso y en el caso promedio para cada operación desarrollada.

Hashing

Operaciones relevantes en una colección (Conjunto/Set y Diccionario)

Consideremos las operaciones en un Set:

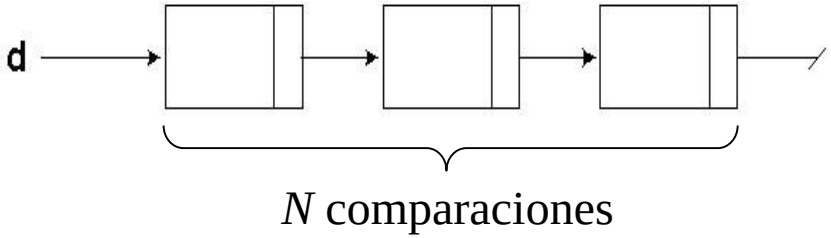
- Insertar (T x, Set & s)
- Borrar (T x, Set & s)
- Pertenece (T x, Set s)

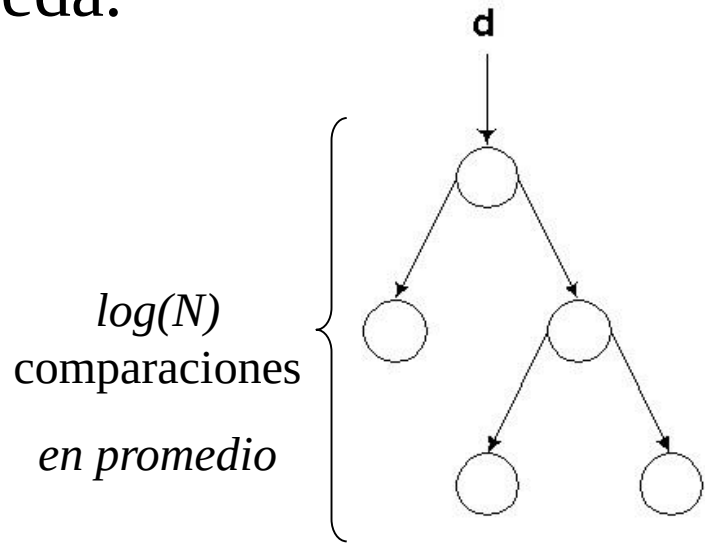
Interesa particularmente la verificación de pertenencia de un elemento.

Hashing
Representaciones

Algunas alternativas conocidas mencionadas:

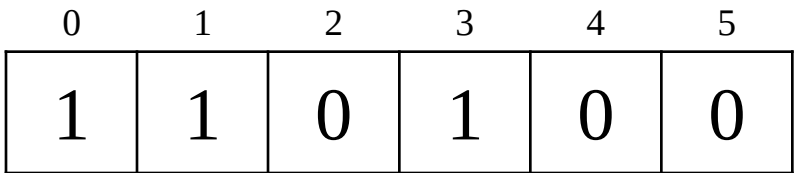
- Listas encadenadas.
- Árboles binarios de búsqueda.





Arreglo de booleanos / bits

Sea *S* un set implementado con un arreglo de booleanos, 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

Tipos y rangos de datos

El arreglo de booleanos permite determinar la pertenencia con una sola operación. Insertar, Borrar y Pertenece tienen O(1) peor caso.

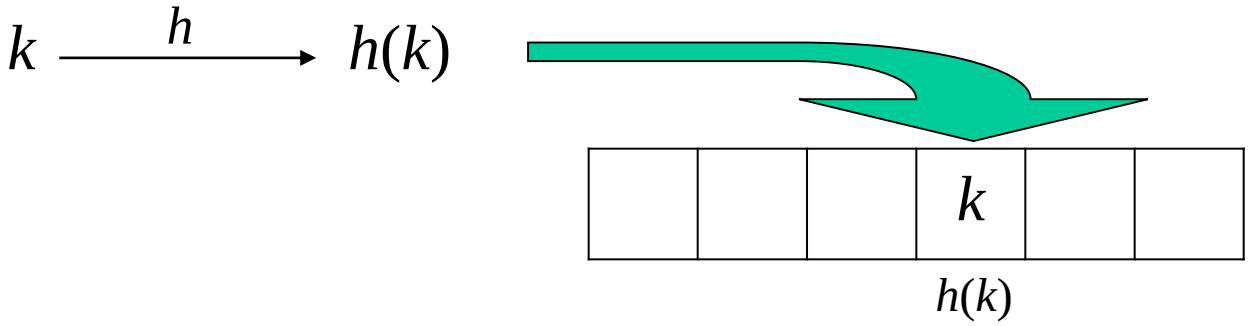
Soporta únicamente datos numéricos (en un subrango de los naturales).

El rango de los elementos puede ser un impedimento para esta técnica, por el alto requerimiento de memoria.

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 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:

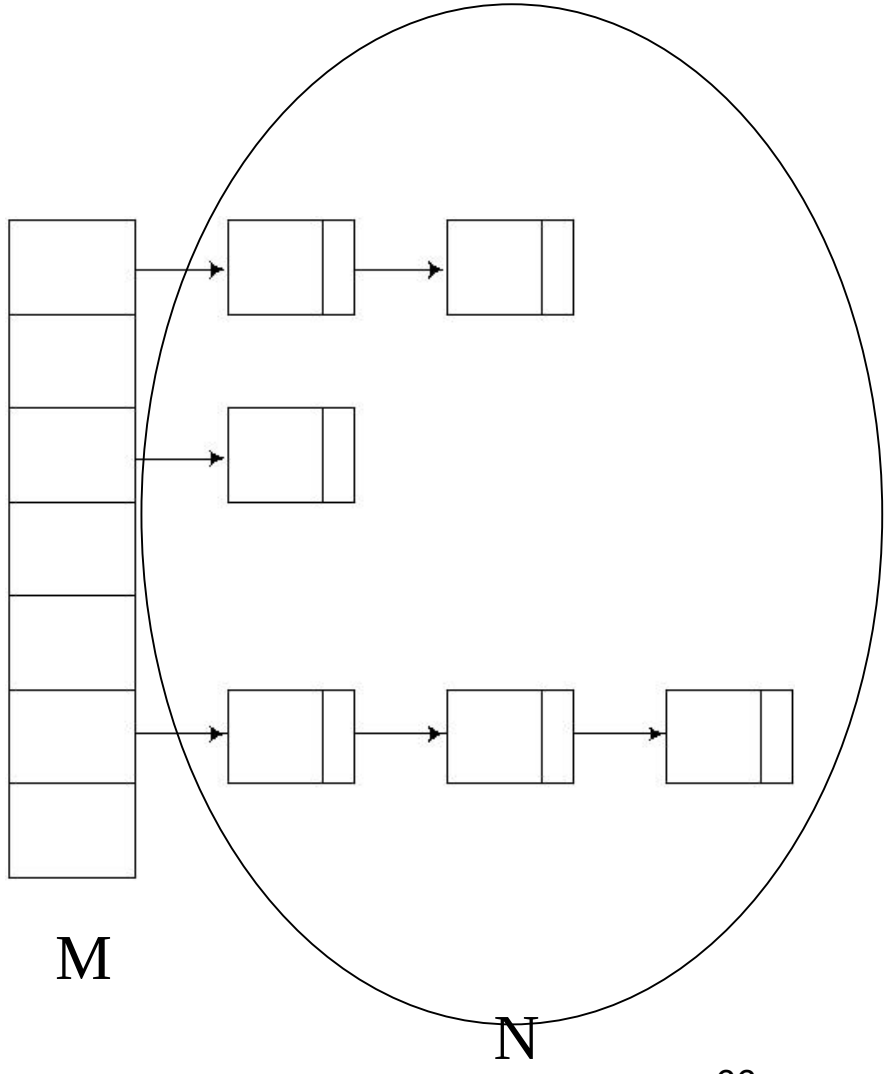
- Separate chaining (*hashing* abierto)
- Linear probing (*hashing* cerrado)
- Double hashing

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

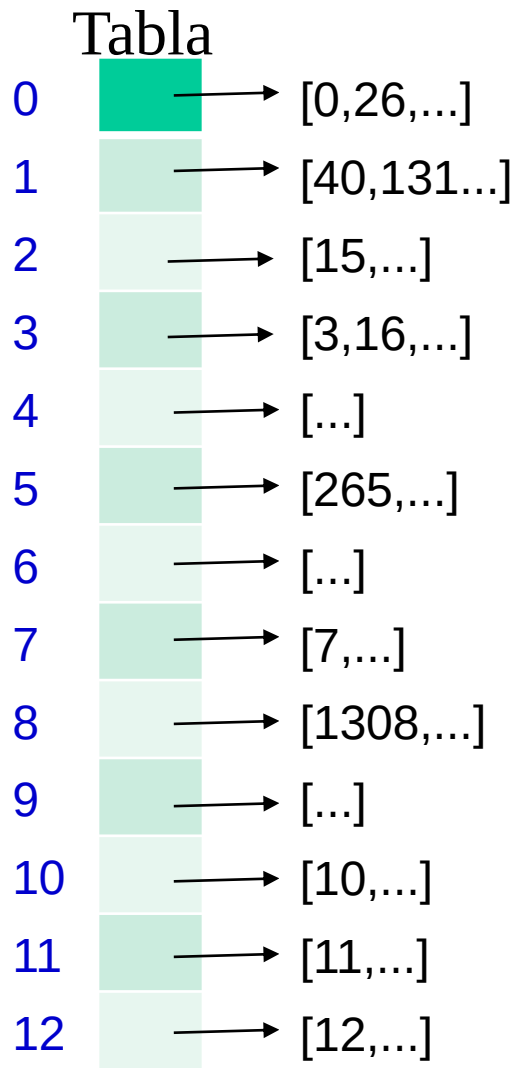
Ejemplo:

- M=11
- hash: int -> int, hash(x) = x%M

5, 10, 15, 20 …

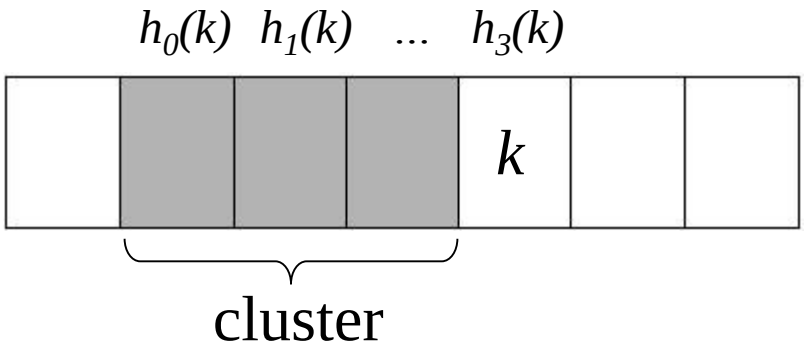
Con M=13, insertar:

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.

Hashing
Ejercicios

Implementar el TAD Diccionario usando:

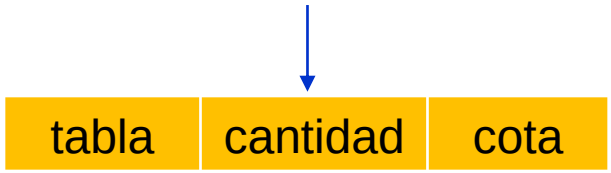
- *Listas no ordenadas; listas ordenadas; arreglos de booleanos/bits; arreglos con tope.*
- *open hashing*, considerando que existe un número esperado *M* de elementos de tipo *int* y que todos son igualmente probables. Si el tipo de los elementos es de tipo *string*, pero se mantienen las mismas condiciones, ¿qué cambia en la implementación realizada?
- *Analizar ventajas y desventajas de las implementaciones anteriores, incluyendo eficiencia en tiempo y espacio.*

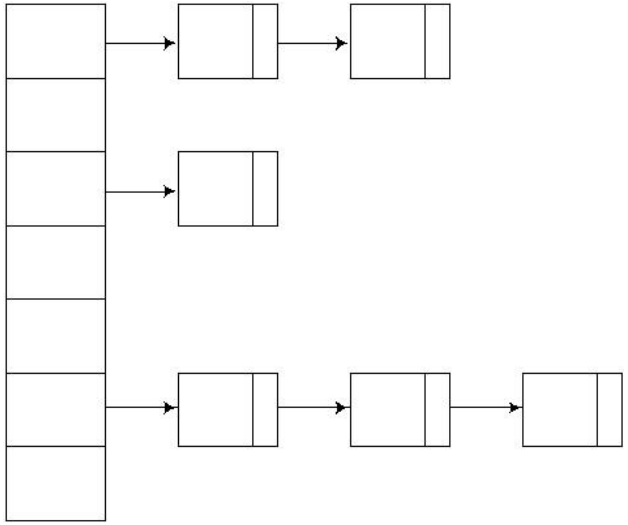
Especificación del TAD Diccionario de enteros

```
#ifndef _DICCIONARIO_H
#define _DICCIONARIO_H

struct RepresentacionDiccionario;
typedef RepresentacionDiccionario * Diccionario;


Diccionario crearDiccionario (int cota);
// Devuelve el diccionario vacío para el que se estiman cota elementos.

void insertarDiccionario (int i, Diccionario &d);
// Agrega i en d, si no estaba. En caso contrario, no tiene efecto.

void eliminarDiccionario (int i, Diccionario &d);
// Elimina i de d, si estaba. En caso contrario, no tiene efecto.

bool perteneceDiccionario (int i, Diccionario d);
// Devuelve true si y sólo si i está en d.

bool esVacioDiccionario (Diccionario d);
// Devuelve true si y sólo si d es vacío.

void destruirDiccionario (Diccionario &d);
// Libera toda la memoria ocupada por d.

#endif /* _DICCIONARIO_H */
```

Implementación de un Diccionario de enteros con hashing abierto

```
#include ...
#include "diccionario.h"

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

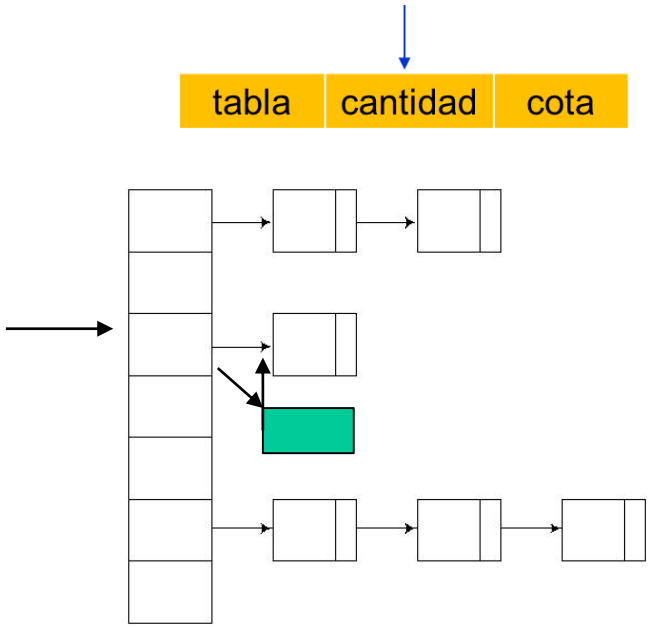




```
Diccionario crearDiccionario (int cota) {
   Diccionario d = new RepresentacionDiccionario();
   d->tabla = new (nodoHash*) [cota];
   for (int i=0; i<cota; i++) d->tabla[i]=NULL;
   d->cantidad = 0;
   d->cota = cota;
   return d;
}
```

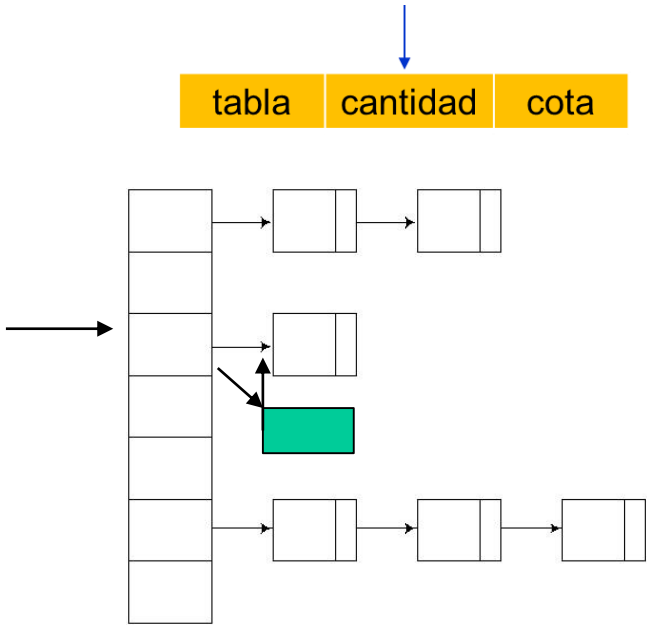
Implementación de Diccionario de enteros con hashing abierto

```
void insertarDiccionario (int i, Diccionario &d){
// no se chequea el factor de carga
if !peteneceDiccionario(i, d){
    nodoHash* nuevo = new nodoHash;
    nuevo->dato = i;
    nuevo->sig = d->tabla[hash(i)%(d->cota)];
    d->tabla[hash(i)%(d->cota)] = nuevo;
    d->cantidad++;
    }
}

bool perteneceDiccionario (int i, Diccionario d) {
    nodoHash* lista = d->tabla[hash(i)%(d->cota)];
    while (lista!=NULL && lista->dato!=i)
          lista = lista->sig;
    return lista!=NULL;
}

bool esVacioDiccionario (Diccionario d) { return (d->cantidad==0); }

void destruirDiccionario (Diccionario &d) {
    for (int i=0; i<cota; i++)
          destruirLista(d->tabla[i]); // destruirLista queda pendiente.
    delete [] d->table;
    delete d;
}
```



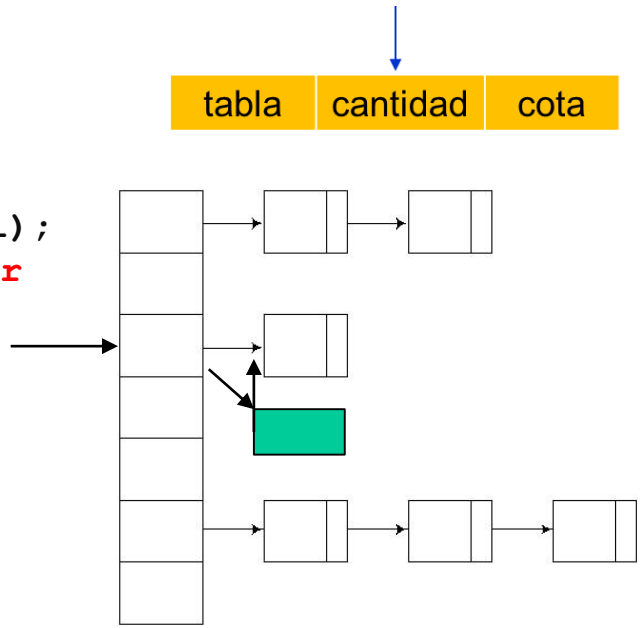
Implementación de Diccionario de enteros con hashing abierto

```
void destruirLista (nodoHash* & l){
    if (l!=NULL){
          destruirLista(l->sig);
          delete l;
          // l = NULL;
    }
}

// versión iterativa
void destruirLista (nodoHash* & l){
    nodoHash* aBorrar;
    while (l!=NULL){
          aBorrar = l;
          l = l->sig;
          delete aBorrar;
    }
}
```



Implementación de Diccionario de enteros con hashing abierto

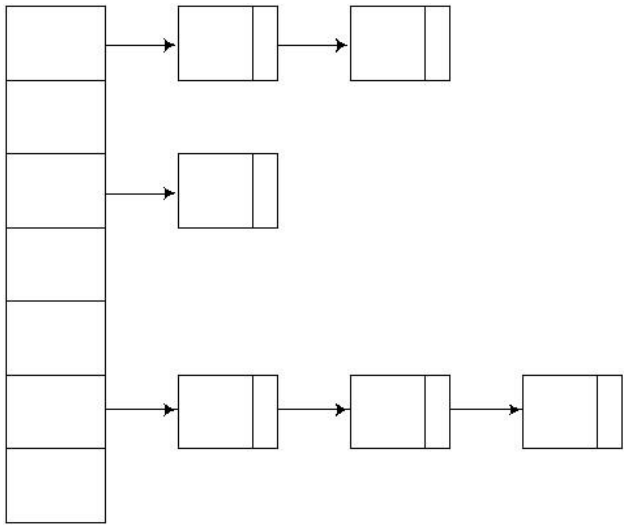
```
void eliminarDiccionario (int i, Diccionario &d) {
    if peteneceDiccionario(i, d){
        borrarLista(d->tabla[hash(i)%(d->cota)], i);
                // asumiendo la operación auxiliar
        d->cantidad--;
    }
}


void borrarLista (nodoHash* & l, int i){
    if (l!=NULL){
        if (l->dato==i){
                nodoHash* aBorrar = l;
                l = l->sig;
                delete aBorrar;
        }
        else    borrarLista(l->sig, i);
    }
}
```



Definición de un CLON para un Diccionario de enteros implementado con hashing abierto.

Asumimos que la especificación incluye la operación clonDiccionario

```
Diccionario clonDiccionario (Diccionario d){
   Diccionario clon = cearDiccionario(d->cota);
   for (int i=0; i<d->cota; i++){
        nodoHash* lista = d->tabla[i];
        while (lista!=NULL){
                insertarDiccionario(lista->dato, clon);
                lista = lista->sig;
        }
   }
   return clon;
}
```

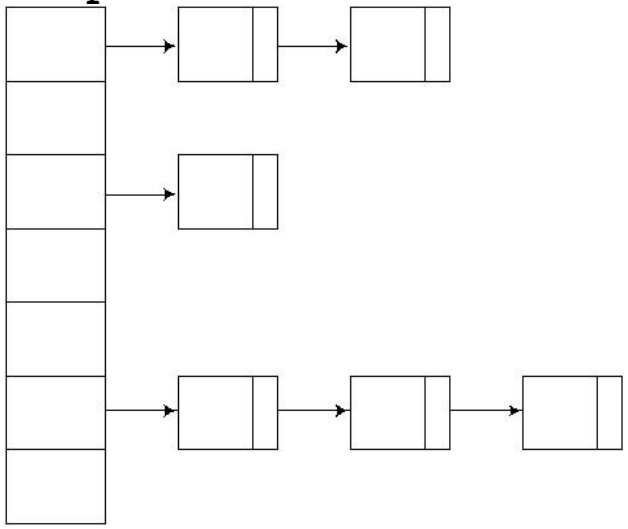


Definición de un CLON para un Diccionario de enteros implementado con hashing abierto.

Asumimos que la especificación incluye la operación clonDiccionario

Otra posibilidad:

```
Diccionario clonDiccionario (Diccionario d){
   Diccionario clon = cearDiccionario(d->cota);
   for (int i=0; i<d->cota; i++){
        clon->tabla[i] = copia(d->tabla[i]);
        // copia es auxiliar; copia la lista sin compartir memoria
   }
   clon->cantidad = d->cantidad;
   return clon;
}
```



Multisets

Un TAD Multiset

- **Vacio** m: construye el multiset m vacío;
- **Insertar x m: agrega x a m;**
- **EsVacio** m: retorna true si y sólo si el multiset m está vacío;
- **Ocurrencias** x m: retorna la cantidad de veces que está x en m;
- **Borrar** x m: elimina una ocurrencia de x en m, si x está en m;
- **Destruir** m: destruye el multiset m, liberando su memoria.

M = { e1, e4, e1, e3, e1, e1, e9, e4} =

{ (e1,4), (e4,2), (e3,1), (e9,1) }

Ejercicios

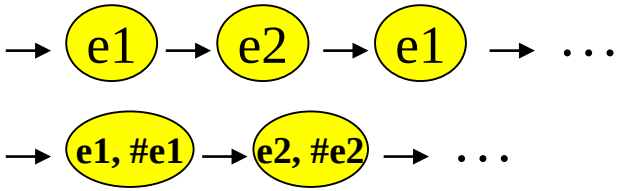
¿Cómo se pueden adaptar las implementaciones de Sets (Conjuntos) a *multisets*?

- Especificar el Tad Multiset en C++.
- Adaptar y analizar para *multisets* las siguientes implementaciones vistas para conjuntos:
  - Variantes de Listas
  - Arreglos de Booleanos (ahora…)
  - ABBs

  Notar que un multiset se puede ver como un set de pares de elementos: (ei, #ei).

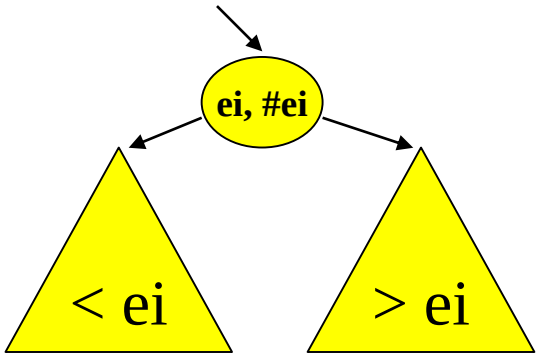
Ejercicios (cont)

- Adaptar y analizar para *multisets* las siguientes implementaciones vistas para conjuntos:
  - Variantes de Listas



  - Arreglos de Booleanos (ahora…)

  - ABBs



Tablas o Funciones Parciales (Mappings)

El TAD Tabla (Mapping, Table)

Una **tabla** es una función parcial de elementos de un tipo, llamado el tipo *dominio*, a elementos de otro (posiblemente el mismo) tipo, llamado el tipo *recorrido* o *rango*.

Que una tabla T asocie el elemento r del recorrido al elemento d del dominio lo denotaremos T(d) = r.

Existen funciones, como cuadrado(i) = i^2, que pueden ser fácilmente implementadas como una función de C++ que implementa la expresión que computa la función. Sin embargo, para muchas funciones no existe otra forma de describir T(d) más que almacenar para cada d el valor T(d).

El TAD Tabla (Mapping, Table)

Ejemplo: implementar una función que asocie a cada estudiante el conjunto de asignaturas de la carrera que tiene aprobadas.

Otro ejemplo: para aplicar una función de nómina que asocie a cada empleado un salario semanal, es necesario almacenar el salario actual de cada empleado (si no hay una regla general).

Veamos cuáles son las operaciones que definen a este tipo abstracto.

Dado un elemento d del dominio nos interesa recuperar el elemento T(d) o saber si T(d) está definido (si d pertenece al dominio de T).

El TAD Tabla (Mapping, Table)

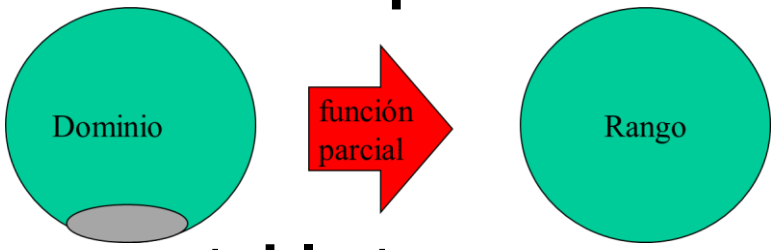
También nos interesa dar de alta elementos en el dominio de T y almacenar sus correspondientes valores.

Alternativamente, también nos interesa modificar el valor T(d) para un determinado d.

Finalmente, necesitamos una operación que nos permita construir una tabla vacía, es decir, la tabla cuyo dominio es vacío.

TAD Tabla/Mapping. Operaciones para:

- construir una tabla vacía;
- insertar una correspondencia (d,r) en una tabla t. Si d está definida en t (tiene imagen), actualiza su correspondencia con r;
- saber si una tabla está vacía;



- saber si un valor d tiene imagen en una tabla t;
- obtener la imagen de un valor d en una tabla t;
- eliminar una correspondencia de una tabla.
- destruir una tabla.

Tabla o función parcial entre Dominio y Rango es un subconjunto del producto cartesiano Dominio x Rango: {(d1,r1), … (di,ri)…}, donde no hay un d con más de un r distinto.

Implementación del TAD Tabla (Mapping, Table)

Existen muchas posibles implementaciones de tablas con dominios finitos. Una de las implementaciones más utilizadas es por medio de tablas de Hashing.

Es posible que el tipo dominio de una tabla sea un tipo elemental de C/C++ y por lo tanto pueda ser usado como el tipo índice de un arreglo. En este caso, la tabla puede ser implementada directamente como un arreglo.

Implementacióm del TAD Tabla (Mapping, Table)

En general, cualquier tabla con dominio finito puede ser representada como una lista de pares (d_1, r_1), (d_2, r_2), ..., (d_k, r_k) donde los d_i son los elementos corrientes del dominio y r_i es el valor que la tabla asocia con d_i, para i = 1..k.

Nuevamente, si el dominio de la tabla admite un orden total, también es una alternativa de implementación razonable usar ABB (o AVL).

Implementacióm del TAD Tabla (Mapping, Table)

En una implementación con arreglos, deberíamos considerar un valor mínimo minD y un valor máximo maxD para el dominio.

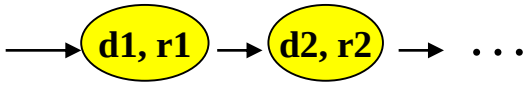
Ejercicios:

- Especificar el Tad Tabla (*mapping*).
- Discutir, desarrollar y analizar todas las implementaciones vistas previamente, ahora para *tablas* (*mappings*): Listas, Arreglos de Booleanos?, ABBs, Arreglos con tope, Hashing.

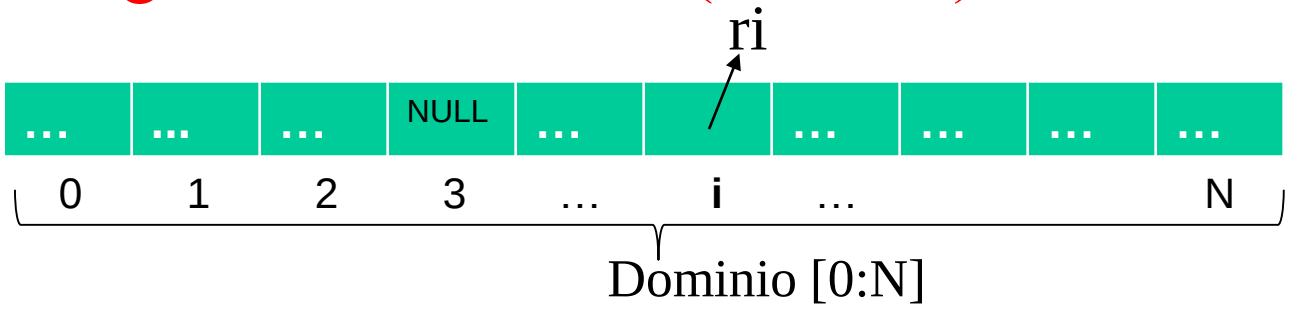
Notar que un *mapping* se puede ver como un conjunto de pares: Dominio - Rango: (d, r).

Ejercicios (cont)

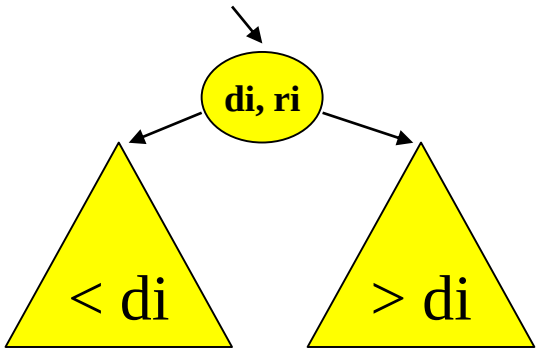
- Adaptar y analizar para *mappings* las siguientes implementaciones vistas para conjuntos:
  - Listas



  - Arreglos de Booleanos (ahora…)



  - ABBs



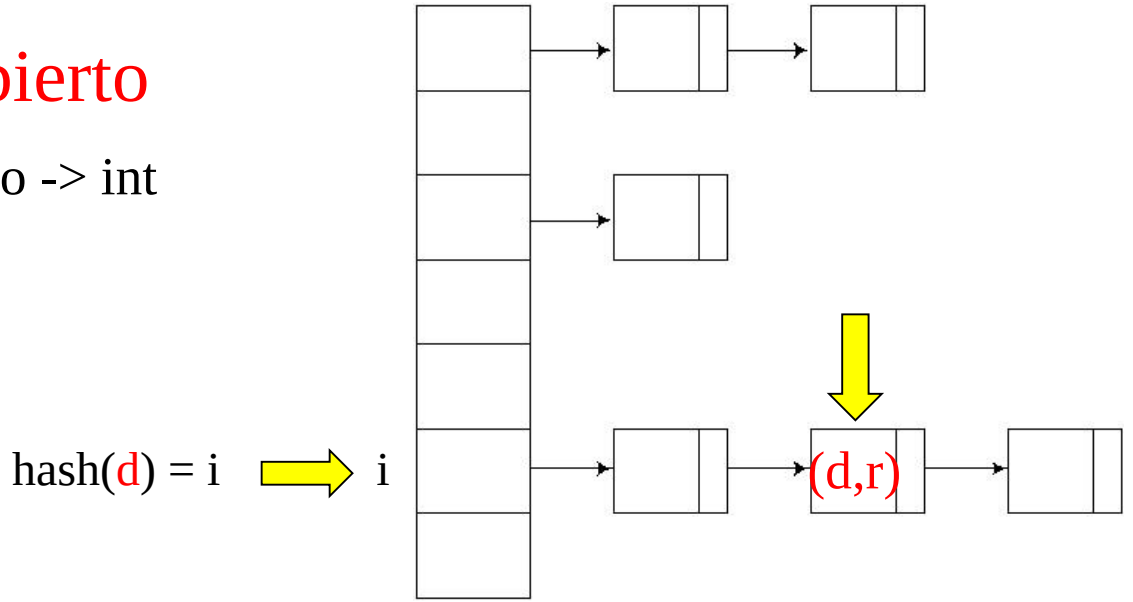
Ejercicios (cont)

- Adaptar y analizar para *mappings* las siguientes implementaciones vistas para conjuntos:
  - Arreglos con tope de pares (d,r) ordenado por dominio

< Dominio

  - Hashing abierto

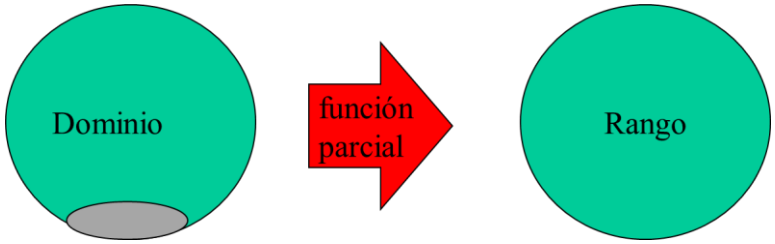
hash: Dominio -> int



Soluciones de algunos ejercicios

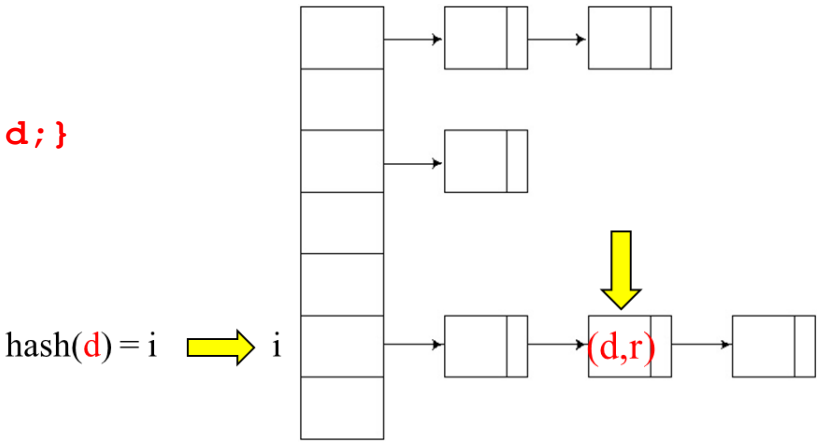
Especificación del TAD Tabla no acotada de D en R (D→R)

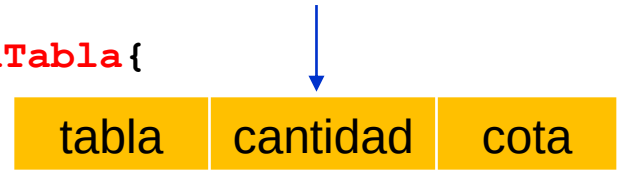
```
#ifndef _TABLA_H
#define _TABLA_H
struct RepresentacionTabla;
typedef RepresentacionTabla * Tabla;

Tabla crearTabla (int cantidadEsperada);
// Devuelve la Tabla vacía no acotada, donde se estiman cantidadElementos.
void insertarTabla (D d, R r, Tabla &t);
/* Agrega la correspondencia (d,r) en t, si d no tenia imagen en t. En
   caso contrario actualiza la imagen de d con r. */
bool estaDefinidaTabla (D d, Tabla t);
// Devuelve true si y sólo si d tiene imagen en t.
bool esVaciaTabla (Tabla t);
// Devuelve true si y sólo si t es vacía.
R recuperarTabla (D d, Tabla t);
/* Retorna la imagen de d en t.
   Precondición: estaDefinidaTabla(d,t). */
void eliminarTabla (D d, Tabla &t);
/* Elimina de t la correspondencia que involucra a d, si d está definida
   en t. En caso contrario la operación no tiene efecto. */
int cantidadEnTabla (Tabla &t);
// retorna la cantidad de correspondencias (d,r) en t.
Tabla copiarTabla (Tabla t);
// retorna una copia de t sin compartir memoria.
void destruirTabla (Tabla &t);
// Libera toda la memoria ocupada por t.

#endif /* _Tabla_H */
```



Implementación de un Tabla no acotada de D en R (D→R) con hashing abierto

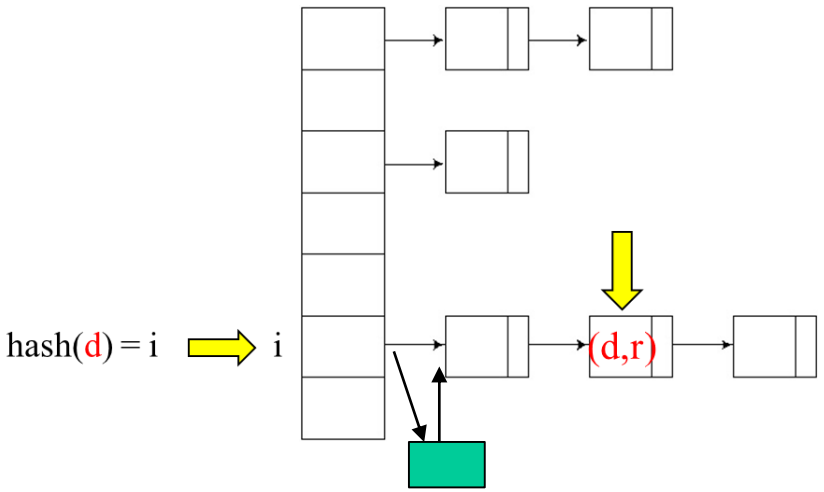
```
#include ...
#include "Tabla.h"
int hash (D d){ return ... }
\\unsigned int hash (unsigned int d){return d;}
struct nodoHash{
    D dom;
    R ran;
    nodoHash* sig;
}
struct RepresentacionTabla{
    nodoHash** tabla;
    int cantidad;
    int cota;
}
Tabla crearTabla (int cantidadEsperada) {
    Tabla t = new RepresentacionTabla();
    t->tabla = new (nodoHash*) [cantidadEsperada];
    for (int i=0; i<cantidadEsperada; i++) t->tabla[i]=NULL;
    t->cantidad = 0;
    t->cota = cantidadEsperada;
    return t;
}
```





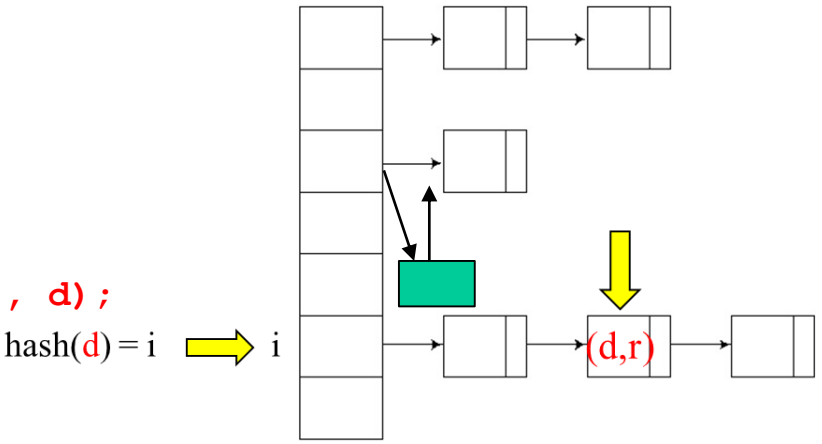
Implementación de Tabla acotado de D en R (D→R) con hashing abierto

```
void insertarTabla (D d, R r, Tabla &t) {
    int posicion = hash(d)%(t->cota);
    nodoHash* lista = t->tabla[posicion];
    while (lista!=NULL && lista->dom!=d)
          lista = lista->sig;
    if (lista==NULL){
          nodoHash* nuevo = new nodoHash;
          nuevo->dom = d;
          nuevo->ran = r;
          nuevo->sig = t->tabla[posicion];
          t->tabla[posicion] = nuevo;
          t->cantidad++;
    }
    else lista->ran = r;
}

bool estaDefinidaTabla (D d, Tabla t) {
    int posicion = hash(d)%(t->cota);
    nodoHash* lista = t->tabla[posicion];
    while (lista!=NULL && lista->dom!=d)
          lista = lista->sig;
    return lista!=NULL;
}

bool esVaciaTabla (Tabla t) { return t->cantidad==0; }
```



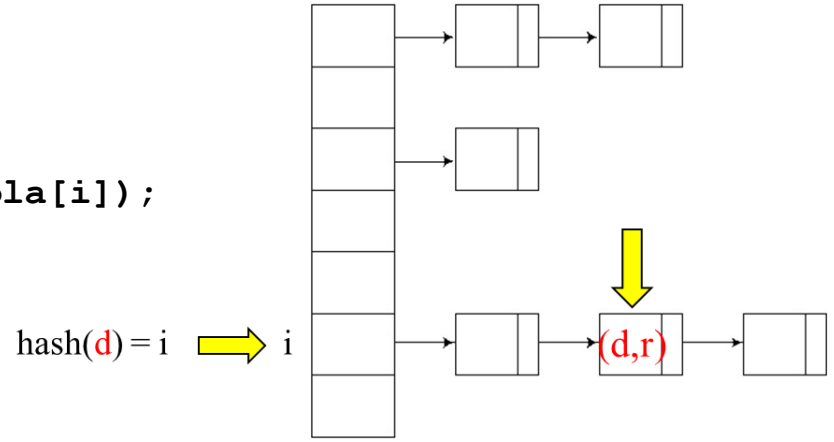
Implementación de Tabla acotado de D en R (D→R) con hashing abierto

```
R recuperarTabla (D d, Tabla t) {
    assert(estaDefinidaTabla(d,t));
    int posicion = hash(d)%(t->cota);
    nodoHash* lista = t->tabla[posicion];
    while (lista->dom!=d)
          lista = lista->sig;
    return  lista->ran;
}

void eliminarTabla (D d, Tabla &t){
    if (estaDefinidaTabla(d,t)){
          int posicion = hash(d)%(t->cota);
          eliminarNodoLista(t->tabla[posicion], d);
          t->cantidad--;
    }
}

int cantidadEnTabla (Tabla t) {return t->cantidad;}

void destruirTabla (Tabla &t) {...}
```



Implementación de Tabla acotado de D en R (D→R) con hashing abierto

```
Tabla copiarTabla (Tabla t) {

    Tabla clon = crearTabla(t->cota);
    for (int i=0; i<t->cota; i++){
          clon->tabla[i] = copiarLista(t->tabla[i]);
    }
    clon->cantidad = t->cantidad;
    return clon;

}
```



EJERCICIO DEL PARCIAL MATUTINO DE JULIO 2020

```
Tabla crearTabla (int cantidadEsperada);
void insertarTabla (D d, R r, Tabla &t);
bool estaDefinidaTabla (D d, Tabla t);
R recuperarTabla (D d, Tabla t);
void eliminarTabla (D d, Tabla &t);
int cantidadEnTabla (Tabla &t);
void destruirTabla (Tabla &t);

D = unisigned int
R = float
```

Una empresa almacena los sueldos de sus empleados en tablas (de tipo *Tabla*), donde el dominio de tipo *unsigned int* corresponde al número de empleado y el rango de tipo *float* corresponde al salario del empleado. Dado que la empresa quiere consolidar salarios dispersos en varias tablas, se propone implementar una función *consolidarTablas* que, dadas dos tablas *t1* y *t2* (de tipo *Tabla*) genere una nueva tabla (de tipo *Tabla*) que contenga los sueldos de los empleados que están en *t1* o en *t2*. Si un empleado está en las dos tablas, se tomará la suma de los salarios que tenga en ambas tablas, y si está solamente en una tabla, se tomará su sueldo en dicha tabla. Para la operación *consolidarTablas* se considerarán solamente empleados cuyos número de empleado sea mayor o igual que *inf* y menor o igual que *sup*; es decir, en el rango [*inf* : *sup*] (asumimos *inf* < *sup*). Implemente *consolidarTablas* sin acceder a la representación del TAD *Tabla*.

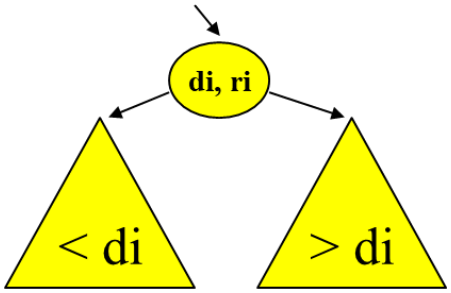
EJERCICIO DEL PARCIAL MATUTINO DE JULIO 2020

```
Tabla crearTabla (int cantidadEsperada);
void insertarTabla (D d, R r, Tabla &t);
bool estaDefinidaTabla (D d, Tabla t);
R recuperarTabla (D d, Tabla t);
void eliminarTabla (D d, Tabla &t);
int cantidadEnTabla (Tabla &t);
void destruirTabla (Tabla &t);

D = unisigned int
R = float
```

```
Tabla consolidarTablas (Tabla t1, Tabla t2, int inf, int sup){
    Tabla t3 = crearTabla(cantidadEnTabla(t1)+cantidadEnTabla(t2)); // sup-inf
    for (int i = inf; i <= sup; i++){
        bool estaT1 = estaDefinidaTabla(i, t1);
        bool estaT2 = estaDefinidaTabla(i, t2);
        if (estaT1 && estaT2)
            insertarTabla(i, recuperarTabla(i, t1)+recuperarTabla(i, t2), t3);
        else if (estaT1)
            insertarTabla(i, recuperarTabla(i, t1), t3);
        else if (estaT2)
            insertarTabla(i, recuperarTabla(i, t2), t3);
    }
    return t3;
}
```

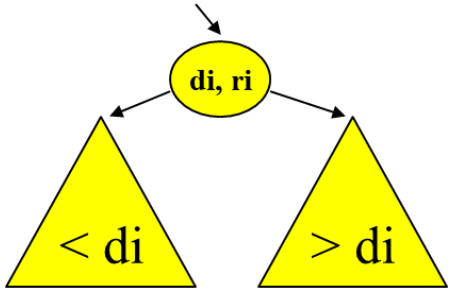
Implementación de un Tabla no acotada de D en R (D→R) con un ABB

```
#include ...
#include "Tabla.h"

struct nodoABB{
   D dom;
   R ran;
   nodoABB* izq;
   nodoABB* der;
};
struct RepresentacionTabla{
   nodoABB* abb;
   int cantidad;
};

Tabla crearTabla (int cantidadEsperada) {
   Tabla t = new RepresentacionTabla();
   t->abb = NULL;
   t->cantidad = 0;
   return t;
}
```



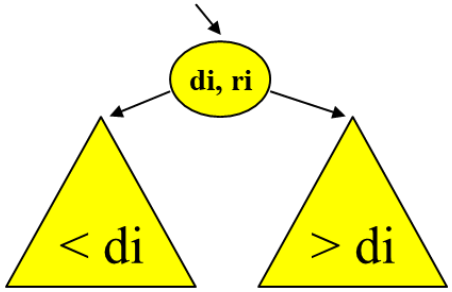
Implementación de Tabla acotado de D en R (D→R) con un ABB

```
/* Inserta en el ABB la pareja (d,r); si d está en a actualiza el r y devuelve
   true. En caso contrario retorna false. */
bool insertarABB (D d, R r, nodoABB* & a){
   if (a==NULL){
        a = new nodoABB;
        a->dom = d; a->ran = r;
        a->izq = a->der = NULL;
        return false;
   }
   else if (d == a->dom){
                a->ran = r;
                return true;
   }
   else if (d < a->dom) return insertarABB(d, r, a->izq);
   else return insertarABB(d, r, a->der);
}

void insertarTabla (D d, R r, Tabla &t) {
   if (!insertarABB(d, r, t->abb))
        t->cantidad++;
}
```



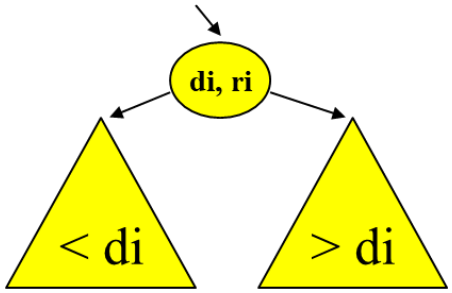
Implementación de Tabla acotado de D en R (D→R) con un ABB

```
bool estaDefinidaTabla (D d, Tabla t) {
    nodoABB * iter = t->abb;
    while (iter!=NULL && iter->dom!=d){
          if (d < iter->dom)
                    iter = iter->izq;
          else      iter = iter->der;
    }
    return iter!=NULL;
}

bool esVaciaTabla (Tabla t) {...}

R recuperarTabla (D d, Tabla t) {
    assert(estaDefinidaTabla(d,t));
    nodoABB * iter = t->abb;
    while (iter->dom!=d){
          if (d < iter->dom)
                    iter = iter->izq;
          else      iter = iter->der;
    }
    return iter->ran;
}

void eliminarTabla (D d, Tabla &t) {...}
int cantidadEnTabla (Tabla &t) {...}
Tabla copiarTabla (Tabla t) // ver diapositiva siguiente
void destruirTabla (Tabla &t) {...}
```



Implementación de Tabla acotado de D en R (D→R) con un ABB

```
// Auxiliar que copia un ABB sin compartir memoria
nodoABB* copiarABB(nodoABB * t){
    if (t!=NULL){
          nodoABB * nuevo = new nodoABB;
          nuevo->dom = t->dom;
          nuevo->ran = t->ran;
          nuevo->izq = copiarABB(t->izq);
          nuevo->der = copiarABB(t->der);
          return nuevo;
    }
    else return NULL;
}

Tabla copiarTabla (Tabla t) {
    Tabla nueva = crearTabla(cantidadEnTabla(t));
    nueva->cantidad = t->cantidad;
    nueva->abb = copiarABB(t->abb);
    return nueva;
}
```



Considere la siguiente especificación para el TAD Tabla no acodata de elementos de tipo int para el dominio y de tipo int para el rango, en donde los valores validos *d* del dominio quedan comprendidos entre dos constantes MIN_DOMINIO y MAX_DOMINIO (MIN_DOMINIO <= *d* <= MAX_DOMINIO) que se definen para cada instancia de la tabla:

```
struct RepresentacionTabla;
typedef RepresentacionTabla* Tabla;
typedef int Dominio;
typedef int Rango;

/*
PRE: -
POST: Retorna una nueva tabla no acotada que puede contener elementos d del Dominio tales que
      minDominio <= d <= maxDominio.
*/
Tabla crearTabla(Dominio minDominio, Dominio maxDominio);

/*
PRE: dominioValidoTabla(t, d)
POST: Si el dominio d no estaba definido en la tabla, entonces inserta en t la correspondencia (d,r). En otro
      caso, actualiza la imagen de d con r.
*/
void insertarTabla(Tabla& t, Dominio d, Rango r);
/*

PRE: -
POST: Retorna true si y sólo si d está dentro de los valores válidos definidos para el dominio de la tabla:
      MIN_DOMINIO <= d <= MAX_DOMINIO
*/
bool dominioValidoTabla(Tabla t, Dominio d);
```

```
/*
PRE: -
POST: Retorna true si y sólo si la tabla t está vacía.
*/
bool esVaciaTabla(Tabla t);

/*
PRE: -
POST: Retorna true si y sólo si existe una asociación con dominio d en la estructura.
*/
bool estaDefinidoTabla(Tabla t, Dominio d);

/*
PRE: estaDefinidoTabla(t,d)
POST: Retorna la imagen asociado al elemento d.
*/
Rango recuperarTabla(Tabla t, Dominio d);

/*
PRE: -
POST: Si estaDefinidoTabla(t, d), entonces elimina la correspondencia con dominio d de la tabla. En otro
      caso, la operación no tiene efecto.
*/
void borrarTabla(Tabla& t, Dominio d);
```

/*

PRE: -

POST: Elimina todas las correspondencias de la tabla y libera la memoria ocupada.

*/

void destruirTabla(Tabla& t);

Se pide:

Implemente parcialmente el TAD basado en la especificación anterior y usando como representación una estructura de ABB. Muestre el código completo de la representación del TAD (RepresentacionTabla) y de las operaciones crearTabla, dominioValidoTabla, insertarTabla *y* estaDefinidoTabla. Puede asumir que el resto de las operaciones están implementadas.

```
struct RepresentacionTabla{
...
}
```

```
struct nodoABB{
    Dominio dom;
    Rango ran;
    nodoABB* izq;
    nodoABB* der;
}
struct RepresentacionTabla{
    nodoABB* abb;
    int cantidad;
    Dominio minD;
    Dominio maxD;
}
```

Parcial 2 – nocturno - Julio 2020

```
struct RepresentacionTabla{
    nodoABB* abb;
    int cantidad;
    int minD;
    int maxD;
};

Tabla crearTabla(Dominio minDominio, Dominio maxDominio){
    Tabla nuevo = new representacionTabla;
    nuevo->abb = NULL;
    nuevo->cantidad = 0;
    nuevo->minD = minDominio;
    nuevo->maxD = maxDominio;
    return nuevo;
}

bool dominioValidoTabla(Tabla t, Dominio d){
    return ((t->minD <= d) && (d <= t->maxD));
}
```

```
struct RepresentacionTabla{
    nodoABB* abb;
    int cantidad;
    int minD;
    int maxD;
};
bool estaDefinidoTabla(Tabla t, Dominio d){
    nodoABB * arbol = t->abb;
    while (arbol != NULL && arbol->dom != d){
        if (d < arbol->dom)
                arbol = arbol->izq;
        else    arbol = arbol->der;
    }
    return (arbol != NULL);
}
Rango recuperarTabla(Tabla t, Dominio d){
    assert(estaDefinidoTabla(t,d));
    nodoABB * arbol = t->abb;
    while (arbol->dom != d){
        if (d < arbol->dom)
                arbol = arbol->izq;
        else    arbol = arbol->der;
    }
    return (arbol->ran);
}
```

```
struct RepresentacionTabla{
    nodoABB* abb;
    int cantidad;
    int minD;
    int maxD;
};

/* Inserta en el ABB la pareja (d,r); si d está en a actualiza el r y devuelve
   true. En caso contrario retorna false. */
bool insertarABB (Dominio d, Rango r, nodoABB* & a){
    if (a==NULL){
          a = new nodoABB;
          a->dom = d; a->ran = r;
          a->izq = a->der = NULL;
          return false;
    }
    else if (d == a->dom){
                  a->ran = r;
                  return true;
    }
    else if (d < a->dom) return insertarABB(d, r, a->izq);
    else return insertarABB(d, r, a->der);
}

void insertarTabla (Tabla &t, Dominio d, Rango r) {
    assert(dominioValidoTabla(t, d));
    if (!insertarABB(d, r, t->abb))
          t->cantidad++;
}
```

Considere la especificación del TAD Tabla del ejercicio anterior. Implemente, sin acceder a la representación del TAD Tabla, la operación componer

```
Tabla componer(Tabla t1, Tabla t2)
```

que dadas dos tablas t_1 y t_2, retorne una nueva tabla t_3 que es el resultado de componer las funciones parciales representadas por t_1 y t_2. Esto es, para cada dominio d en t_1, $t_3(d) = t_2(t_1(d))$. En caso de que $t_1(d)$ no esté definido en t_2, el dominio d no debe aparecer en t_3.

Ejemplo:

t_1

d_1	$t_1(d_1)$
1	10
2	20
3	30
4	40
5	50
6	60
7	70

t_2

d_2	$t_2(d_2)$
30	300
22	220
50	500
75	750
50	500

t_3

d_1	$t_2(t_1(d_1))$
3	300
5	500

Notar que no todos los valores en el dominio de t_1 terminan representados en el dominio de t_3.

Tabla crearTabla(Dominio minDominio, Dominio maxDominio);
bool estaDefinidoTabla(Tabla t, Dominio d);
Rango recuperarTabla(Tabla t, Dominio d);
void insertarTabla(Tabla& t, Dominio d, Rango r);

Tabla componer (Tabla t1, Tabla t2, Dominio minComponer, Dominio maxComponer) {

```
   Tabla t3 = crearTabla(minComponer, maxComponer);
   for (int d = minComponer; d <= maxComponer; d++){
        if (estaDefinidoTabla(t1,d))
               int imagenT1 = recuperarTabla(t1,d);
               if (estaDefinidoTabla(t2,imagenT1))
                   insertarTabla(t3,d,recuperarTabla(t2,imagenT1));
   }
   return t3;
}
```

Parcial 2 – diciembre 2020 - matutino

Considere la siguiente especificación del TAD *Multiset* no acotado de números enteros:

struct RepresentacionMultiset;

typedef RepresentacionMultiset Multiset;*

// POS: Devuelve el multiset vacío.

Multiset crear();

// POS: Agrega n ocurrencias del elemento x al multiset m. PRE: n>0.

void insertar (Multiset & m, int x, unsigned int n);

// POS: Devuelve la cantidad total de elementos del multiset m (0 si m está vacío).

int cantidad (Multiset m);

// POS: Devuelve la cantidad de ocurrencia del elemento x del multiset m (0 si x no está en m).

unsigned int ocurrencias (Multiset m, int x);

/ POS: Elimina a lo sumo n ocurrencia del elemento x del multiset m. Si ocurrencias(m, x)<=n entonces en m no quedarán ocurrencias del elemento x, y se liberará la memoria correspondiente. */*

void eliminar (Multiset & m, int x, unsigned int n);

// POS: Retorna el mínimo elemento del multiset m (independientemente del número de ocurrencias). PRE: m no vacío.

int min (Multiset m);

// POS: Retorna el máximo elemento del multiset m (independientemente del número de ocurrencias). PRE: m no vacío.

int max (Multiset m);

Se pide:

- Implemente el TAD ***Multiset*** anterior de tal manera que las operaciones *insertar*, *ocurrencias* y *eliminar* tengan O(log(*n*)) de tiempo de ejecución en el caso promedio, siendo *n* la cantidad de elementos diferentes del multiset, y las operaciones *crear*, *cantidad*, *min* y *max* tengan O(1) peor caso. Concretamente, defina en C++ la **representación del TAD** (***RepresentacionMultiset***) y escriba los códigos de las operaciones ***insertar***, ***ocurrencias y min***. Omita el código del resto de las operaciones, que puede asumir están implementadas.

- Implemente el procedimiento ***vaciar*** que dado un multiset *m*, según la especificación dada previamente, elimine todos sus elementos, dejando a éste vacío. El procedimiento *vaciar* no debería acceder a la representación del TAD ni dejar memoria colgada (sin liberar).

  void vaciar (Multiset & m)

- Dos multisets son iguales si poseen los mismos elementos la misma cantidad de veces. Defina la función ***iguales*** que calcule la igualdad entre dos multisets. La función no debe modificar los multisets ni acceder a la representación del TAD.

  bool iguales (Multiset m1, Multiset m2)

SOLUCIÓN

```
struct NodoABB {
    int dato;
    unsigned int ocurrencias;
    NodoABB* izq;
    NodoABB* der;
}

struct RepresentacionMultiset{
    NodoABB* arbol; // Necesario para cumplir los órdenes log(n) promedio pedidos
    int cantidad; // se pide conocer la cantidad total en O(1)
    int min; // Necesario para cumplir O(1) en la búsqueda del mínimo
    int max; // Necesario para cumplir O(1) en la búsqueda del máximo
}
```

```
/* Auxiliar */
//PRE:
    //POS: Agrega n ocurrencias del elemento e en el arbol t
void insABB(int e, unsigned int n, NodoABB*& t){
    if(t == NULL){
        t = new NodoABB;
        t->dato = e;
        t->ocurrencias = n;
        t->izq = t->der = NULL;
    }
    else if (e < t->dato)
        insABB(e, n, t->izq);
    else if (e > t->dato)
        insABB(e, n, t->der);
    else    t->ocurrencias += n;
}

void insertar(Multiset & m, int e, unsigned int m){
    insABB(e, n, m->arbol);
    if (m->cantidad == 0 || e < m->min)
        m->min = e;
    if (m->cantidad == 0 || e > m->max)
        m->max = e;
    m->cantidad += n;
}
```

```
unsigned int ocurrencias(Multiset m, int e){
    NodoABB * t = m->arbol;
    while (t != NULL && t->dato != e){
        if(e < t->dato) t = t->izq;
        else t = t->der;
    }
    if (t == NULL) return 0 else return t->ocurrencias;
}
```

```
int min(Multiset m) {
    assert(m->cantidad > 0);
    return m->min;
}
```

```
void vaciar(Multiset & m){
    while(cantidad(m) > 0)
        eliminar(m, min(m), ocurrencias(m, min(m)));
}
```

```
bool iguales(Multiset m1, Multiset m2){
    if(cantidad(m1) == 0 && cantidad(m2) == 0)
        return true;
    else if (cantidad(m1) == 0 || cantidad(m2) == 0)
        return false;
    else if (min(m1) != min(m2) || max(m1) != max(m2)) // min y max existen si no son vacíos
        return false
    else{  // Ambos son no vacíos y sus mínimos y máximos coinciden
        for (int i = min(m1); i < = max(m1); i++){
            if (ocurrencias(m1, i) != ocurrencias(m2, i))
                return false;
        }
        return true;
    }
}
```

Estructuras Múltiples (Multiestructuras)

Estructuras Múltiples

Con frecuencia, un problema aparentemente simple de representación de un conjunto o correspondencia conlleva un difícil problema de elección de estructuras de datos.

La elección de una estructura de datos para el conjunto simplifica ciertas operaciones, pero hace que otras lleven demasiado tiempo y al parecer no existe una estructura de datos que posibilite lograr cierta eficiencia en un conjunto de operaciones.

En tales casos, la solución suele ser el uso simultáneo de dos o más estructuras diferentes para el mismo conjunto o correspondencia, buscando acceso rápido pero sin redundancia de información.

Ejemplos (1)

PROBLEMA: Ranking de la FIFA

Se desea mantener una escala de equipos de futbol en la que cada equipo esté situado en un único puesto. Los equipos nuevos se agregan en la base de la escala, es decir, en el puesto con numeración más alta. Un equipo puede retar a otro que esté en el puesto inmediato superior (el i al i-1, i>1), y si le gana, cambia de puesto con él.

Pensar en una representación para esta situación !!

Ejemplos (1)

Se puede representar la situación anterior mediante un TAD cuyo modelo fundamental sea una correspondencia de nombres de equipos (cadenas de char) con puestos (enteros 1, 2, …).

Las 3 operaciones a realizar son:

- AGREGA(nombre): agrega el equipo nombrado al puesto de numeración más alta.
- RETA(nombre): es una función que devuelve el nombre del equipo del puesto i-1 si el equipo nombrado está en el puesto i, i>1.
- CAMBIA(i): intercambia los nombres de los equipos que estén en los puesto i e i-1, i>1.

Ejemplos (1)

ALTERNATIVA 1:

Un arreglo de ESCALA, donde ESCALA[i] sea el nombre del equipo en el puesto i.

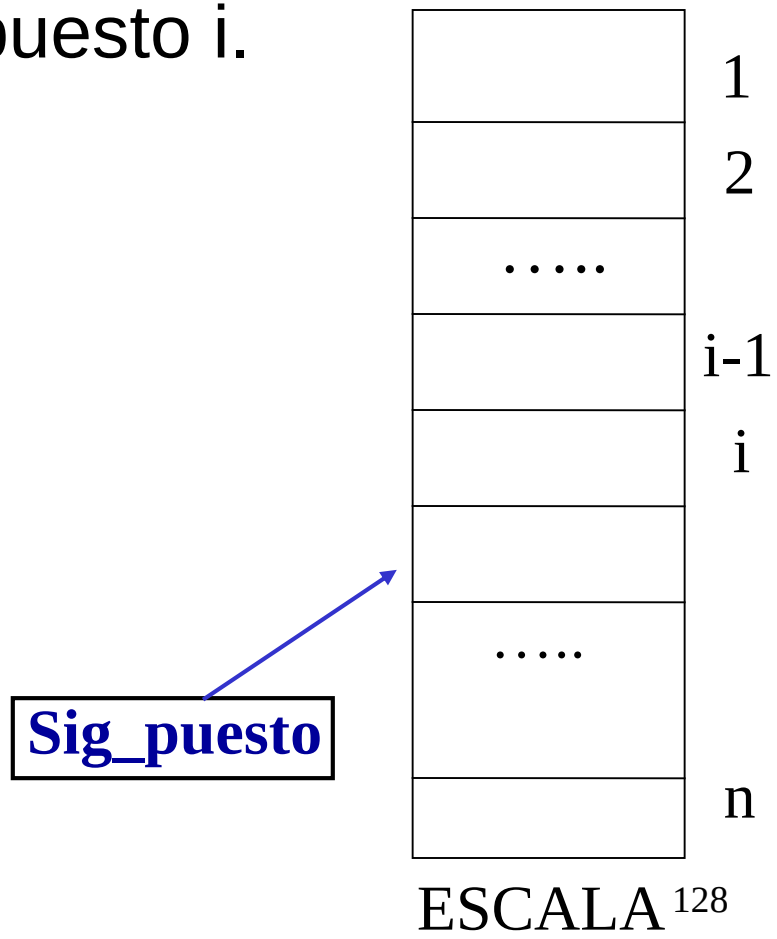
- AGREGA: Cómo sería?, Qué tiempo llevaría?
- CAMBIA: Cómo sería?, Qué tiempo llevaría?
- RETA(nom): Cómo sería?, Qué tiempo llevaría?

Ejemplos (1)

ALTERNATIVA 1:

Un arreglo de ESCALA, donde ESCALA[i] sea el nombre del equipo en el puesto i.

- AGREGA(nombre)
- RETA(nombre)
- CAMBIA(i)



Ejemplos (1)

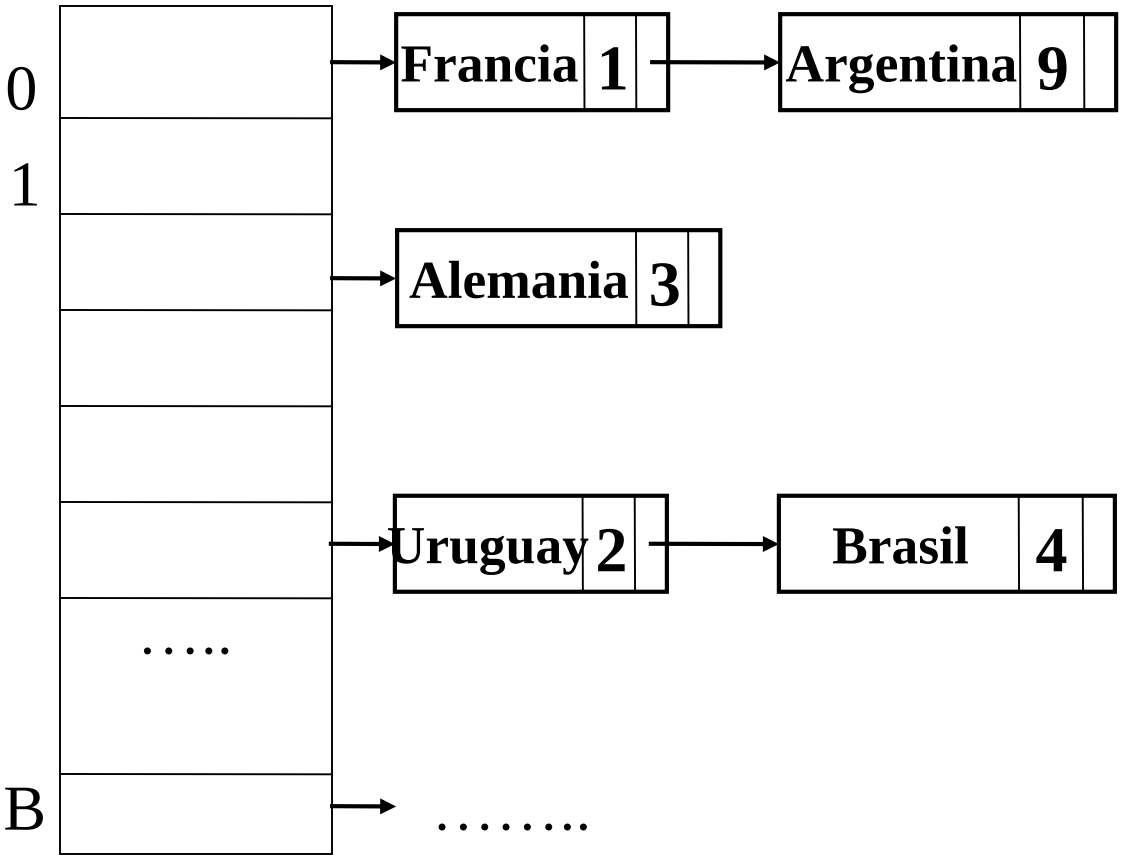
ALTERNATIVA 2: Qué otra representación podría considerarse?

=> OPEN HASING (en el supuesto que es posible mantener en número de *buckets* proporcional al número de equipos)

- AGREGA: Qué tiempo llevaría?
- CAMBIA: Qué tiempo llevaría?
- RETA(nom): Qué tiempo llevaría?

Ejemplos (1)

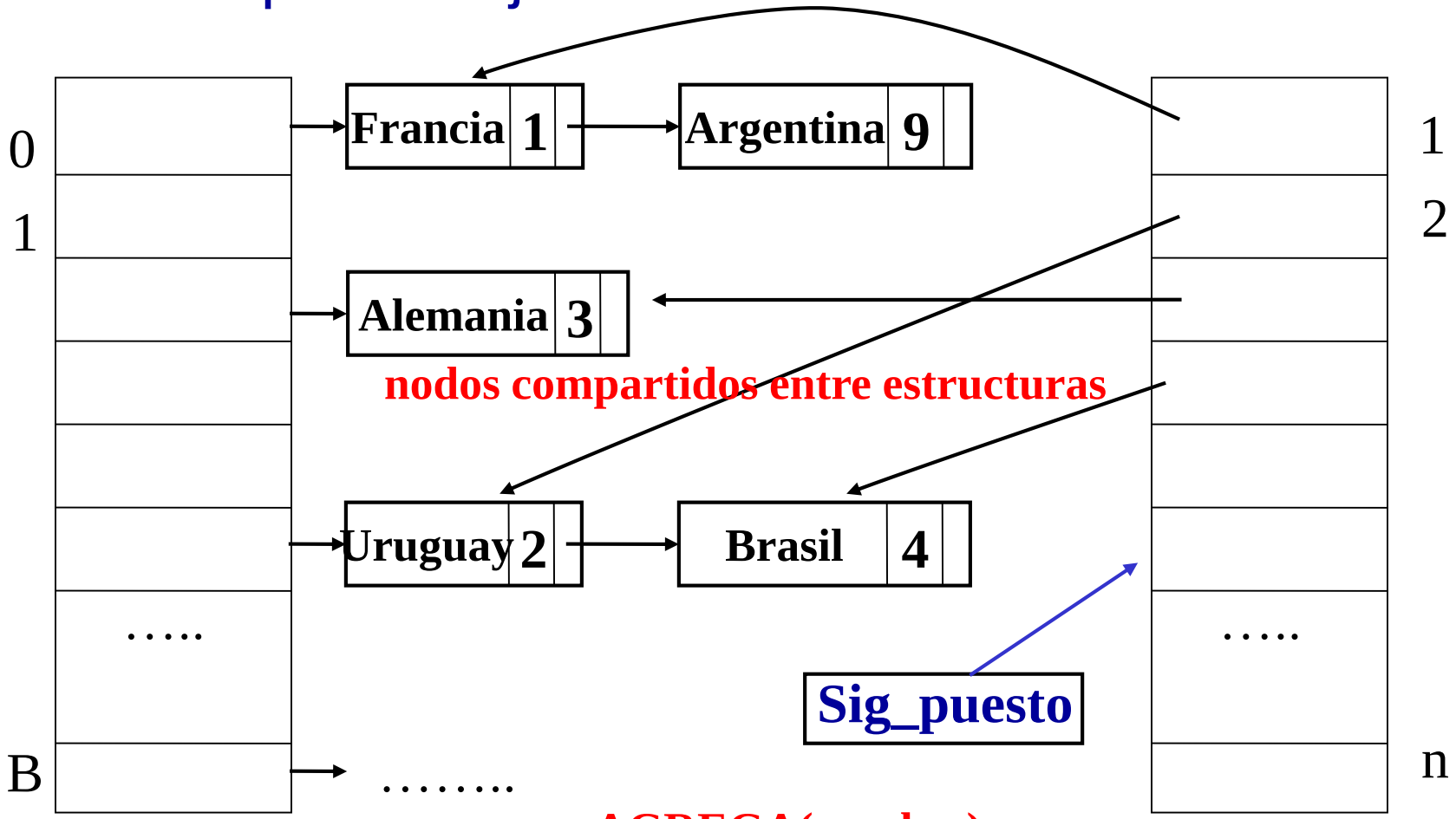
ALTERNATIVA 2: OPEN HASING para asociaciones:
nombre – puesto; con clave: nombre de equipo.



- AGREGA(nombre)
- RETA(nombre)
- CAMBIA(i)

Ejemplos (1)

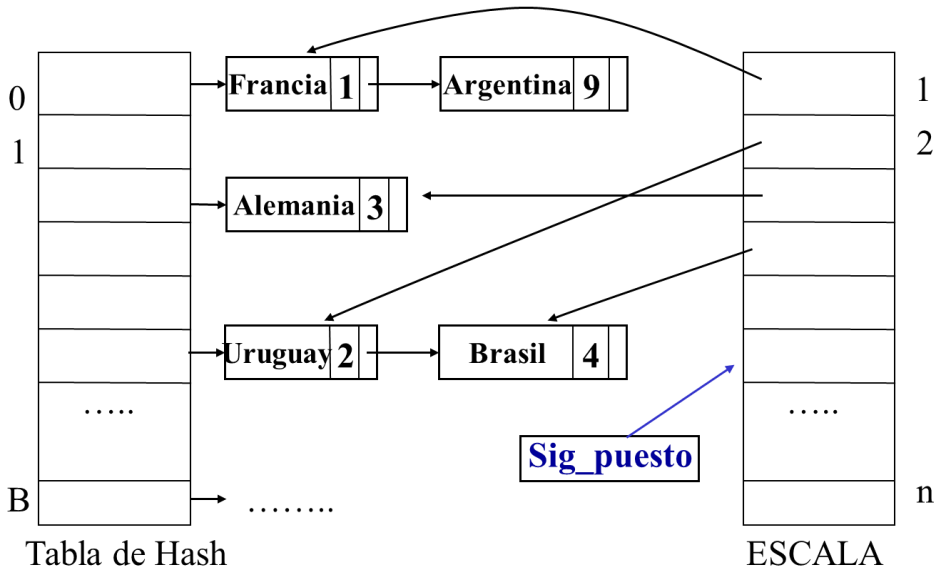
Y si combinamos las dos estructuras, ¿ cuáles son los tiempos de ejecución ?



AGREGA(nombre)
RETA(nombre)
CAMBIA(i)

Ejemplos (1)

```
struct nodoHash{
    char* pais;
    int puesto;
    nodoHash* sig;
};
strutct multiFIFA{
    nodoHash** ESCALA;
    int Sig_puesto;
    int cotaPaises;
    nodoHash** TablaHash;
};
typedef multiFIFA* FIFA;
```



Asumimos: *int hash (char* nombre)*, con distribución uniforme sobre [0:-].

Implementar:

FIFA crearFIFA (int cotaPaises){ … }

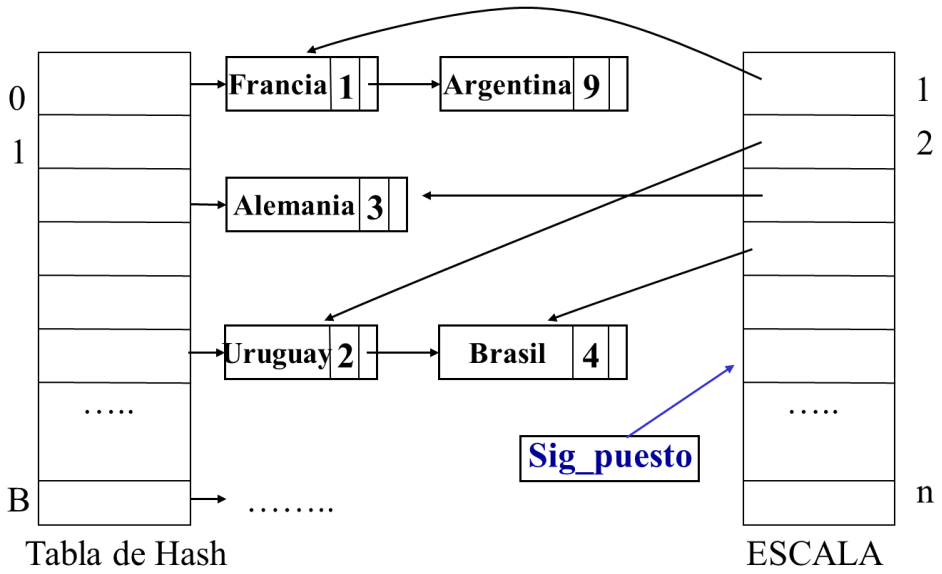
void AGREGA(FIFA & f, char* pais){ … }

char* RETA(FIFA f, char* pais){ … }

void CAMBIA(FIFA & f, int posicion){ … }

Ejemplos (1)

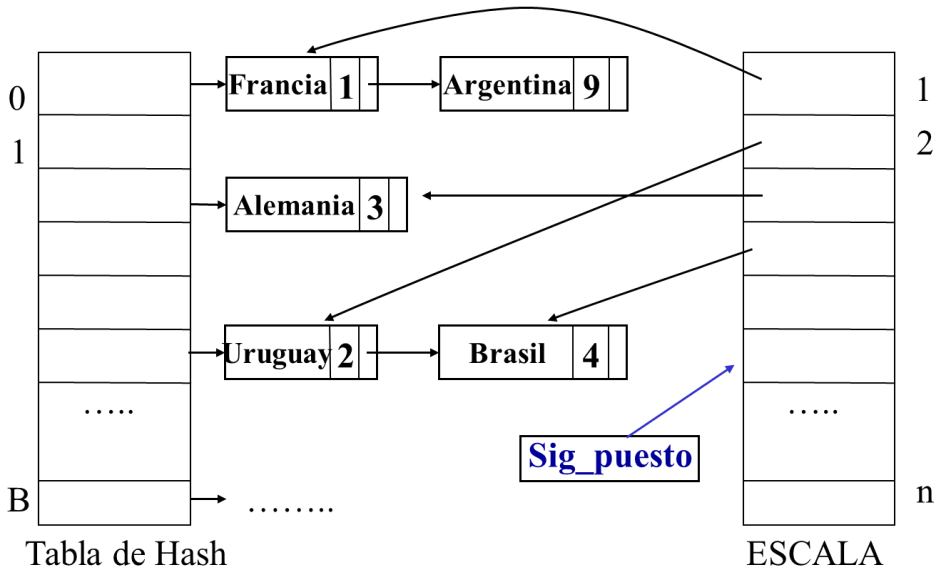
```
struct nodoHash{
        char* pais;
        int puesto;
        nodoHash* sig;
};
strutct multiFIFA{
        nodoHash** ESCALA;
        int Sig_puesto;
        int cotaPaises;
        nodoHash** TablaHash;
};
typedef multiFIFA* FIFA;
```




```
char* RETA(FIFA f, char* pais){
    int posicion = hash(pais)%(f->cotaPaises);
    nodoHash* lista = f->TablaHash[posicion];
    while (lista!=NULL && lista->pais!=pais)
            lista = lista->sig;
    if (lista!=NULL &&  lista->puesto!=1){
            return (f->ESCALA[lista->puesto - 1])->pais;
    }
    else return NULL;
}
```

Ejemplos (1)

```
struct nodoHash{
      char* pais;
      int puesto;
      nodoHash* sig;
};
strutct multiFIFA{
      nodoHash** ESCALA;
      int Sig_puesto;
      int cotaPaises;
      nodoHash** TablaHash;
};
typedef multiFIFA* FIFA;
```




```
void CAMBIA(FIFA & f, int posicion){
      nodoHash* nodoVencedor = f->ESCALA[posicion];
      nodoHash* nodoPerdedor = f->ESCALA[posicion-1];
      nodoVencedor->puesto = posicion-1;
      nodoPerdedor->puesto = posicion;
      f->ESCALA[posicion] = nodoPerdedor;
      f->ESCALA[posicion-1] = nodoVencedor;
}
```

Ejemplos (2)

Analicen el ejemplo de asociaciones muchos a muchos y el uso de una estructura de listas múltiples del libro de Aho, Hopcroftand y Ullman (cap 4, secc 4.12).

estudiantes - cursos

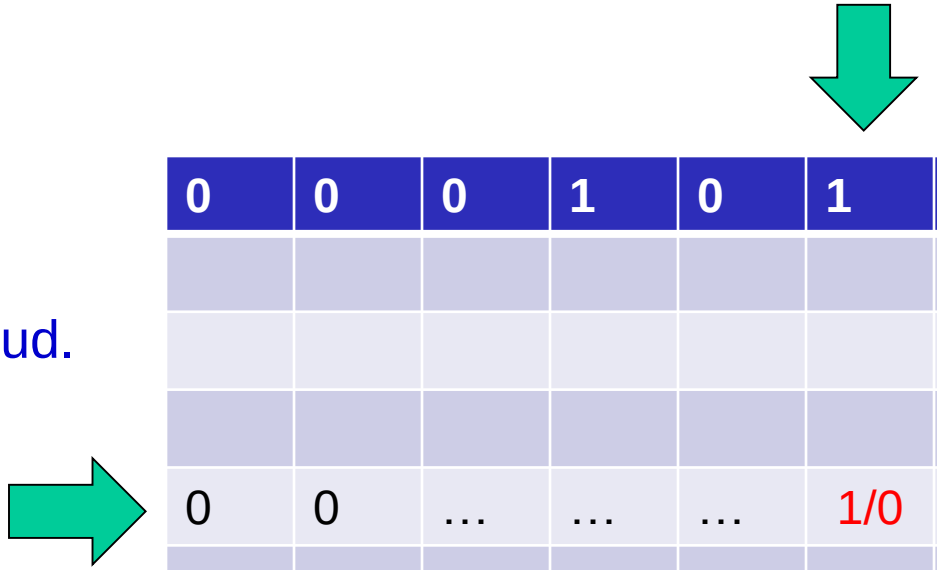
- un estudiante inscripto a más de un curso.
- un curso tiene más de un estudiante.

Pensar en una representación simple y alguna más eficiente (sobre todo en cuanto a espacio de almacenamiento)

Ejemplos (2)

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

Nro. curso



Ejemplos (3)

Se desea una estructura de datos para mantener información de atletas que han participado en la competencia de los 100 metros llanos en los juegos olímpicos en de los últimos 20 años. Los datos que interesan sobre cada competidor son la posición (1 a 8) y el año en que compitió. Cada competidor está identificado por un código (entero). Si un atleta participó más de una vez, aparece con la mejor posición que obtuvo.

Pensar en una representación del problema, para que los siguientes requerimientos se realicen eficientemente:

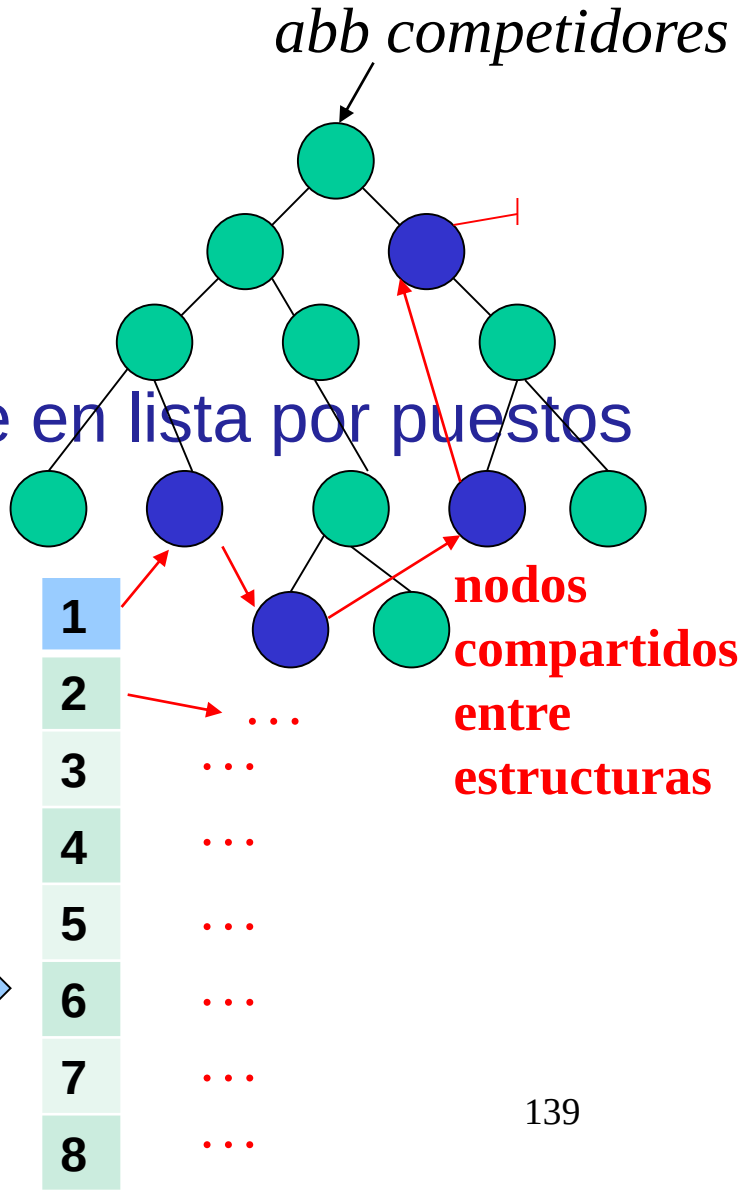
Ejemplos (3)

1. ¿ Cuáles competidores salieron alguna vez en la primera posición ?.
2. ¿ Saber el año en que un competidor obtuvo el máximo puesto ?.
3. Imprimir los competidores ordenados por código.
4. Imprimir todos los competidores que están en el puesto k ($1 \leq k \leq 8$).
5. Saber cuál es el puesto más alto obtenido por un competidor.
6. Las naturales de inserción, supresión y búsqueda de competidores en una posición.

Ejemplos (3)

Considere la siguiente estructura:

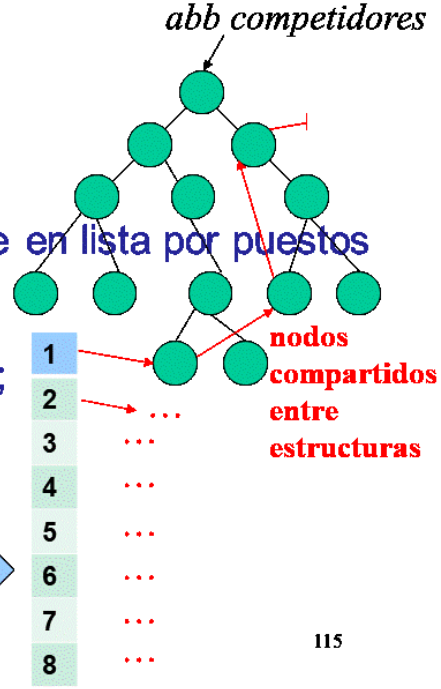
```
typedef struct nodo {
    int codigo, año, puesto;
    struct nodo *izq, *der;
    struct nodo *sig;        // Siguiente en lista por puestos
    } celdaCompe;

typedef celdaCompe *abbCompe;

typedef struct {
    abbCompe competidores;
    abbCompe puestos[9];
} resultados100m;
```



Ejemplos (3)

Considere la siguiente estructura:

```
typedef struct nodo {
   int codigo, año, puesto;
   struct nodo *izq, *der;
   struct nodo *sig;        // Siguiente en lista por puestos
   } celdaCompe;

typedef celdaCompe *abbCompe;

typedef struct {
   abbCompe competidores;
   abbCompe puestos[9];
} resultados100m;
```

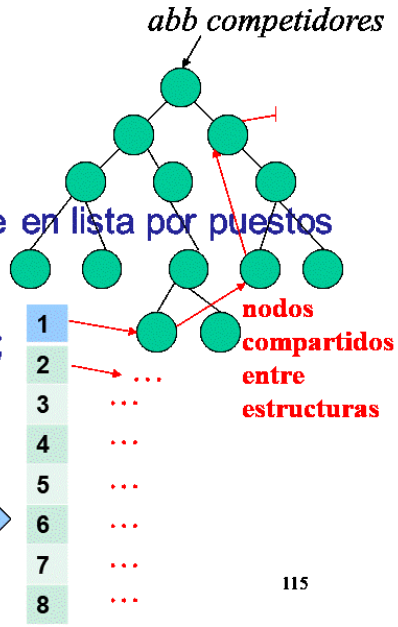



```
void imprimirPorPuesto (resultados100m * e, int k){
    assert(0<k && k<9);
    nodo * lista = e->puestos[k];
    while (lista!=NULL){
            cout << lista->código;
            lista = lista->sig;
    }
}
```

Ejemplos (3)

Considere la siguiente estructura:

```
typedef struct nodo {
   int codigo, año, puesto;
   struct nodo *izq, *der;
   struct nodo *sig;        // Siguiente en lista por puestos
   } celdaCompe;

typedef celdaCompe *abbCompe;

typedef struct {
   abbCompe competidores;
   abbCompe puestos[9];
} resultados100m;
```



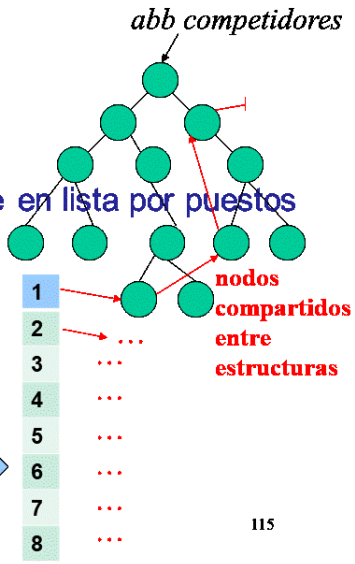

```
void imprimirCompetidores(resultados100m * e){
    imprimirCompetidoresABB(e->competidores);
}
void imprimirCompetidoresABB(nodo * abb){
    if (abb != NULL){
        imprimirCompetidoresABB(abb->izq);
        cout << abb->código;
        imprimirCompetidoresABB(abb->der);
    }
}
```

Ejemplos (3)

```
Nodo * desenganchar NodoPuesto (resultados100m * e, int cod, int k){
    assert(0<k && k<9);
    return eliminarLogico(e->puestos[k], cod);
}
nodo * eliminarLogico (nodo * & l, int cod){
    if (l!=NULL){
        if (l->código == cod){
            nodo * ret = l;
            l =  l->sig;
            return ret;
        }
        else return eliminarLogico (l->sig, cod);
    }
    return NULL;
}
```

Considere la siguiente estructura:

```
typedef struct nodo {
   int codigo, año, puesto;
   struct nodo *izq, *der;
   struct nodo *sig;        // Siguiente en lista por puestos
   } celdaCompe;
typedef celdaCompe *abbCompe;
typedef struct {
   abbCompe competidores;
   abbCompe puestos[9];
} resultados100m;
```



115

Resolver en base a lo de arriba la funcionalidad que permita actualizar el puesto y año de un competidor ya existente.

Ejemplos (3)

La anterior es una estructura dual que permite ingresar por código de jugador o por puesto. El campo competidores es un árbol binario de búsqueda por código del competidor. El campo puestos es un arreglo, que en el lugar i-ésimo contiene un puntero a la lista de competidores que obtuvieron el puesto i. Esta lista está hilvanada sobre el mismo árbol.

- Ejercicio: escribir un procedimiento con el encabezado:

  void BorrarCompetidor (resultados100m *Res, int cod)

  Este procedimiento borra de Res al competidor con código cod.

Ejemplos (3)

- Indique el orden de ejecución para el caso medio del procedimiento **BorrarCompetidor**.

  Modifique la estructura de forma tal que ese tiempo pase a ser O(log n).

- Ejercicio: escriba el procedimiento BorrarCompetidor para la modificación propuesta en el punto anterior.

Ejemplos (4)

Los empleados de cierta compañía se representan en la base de datos de la compañía por su nombre (que se supone único), número de empleado y número de seguridad social. Se sabe que la cantidad de empleados puede estimarse en un valor K. Se pide:

- Sugerir una estructura de datos que permita, dada la representación de un empleado, encontrar las otras dos representaciones del mismo individuo de la forma más rápida, en promedio, y evitando redundancia de información. Qué rápida, en promedio, puede lograrse que sea cada una de estas operaciones?. Justifique.

Ejemplos (4)

- Definir en C++ la estructura del inciso anterior y desarrollar, en C++, la operación que dado el nombre de un empleado imprime su número de empleado y número de seguridad social, si el empleado existe.
- Definir en C++ la operación que dado un empleado lo inserta en la estructura.

Considerando la estructura por usted definida, pueden imprimirse los empleados de la compañía ordenados por su nombre de forma ordenada en O(n)?. Justifique.

Ejemplos (4)

Estructura:

La estructura está compuesta por tres tablas de hash abiertas (por ejemplo) de tamaño K, una por cada representación del empleado. Los nodos de las listas en las tablas son compartidos, entre los tres hash, para evitar redundancia de información. Cada tabla de hash tendría una función de hash (dos de ellas tienen dominio int y la otra string)

Las tres operaciones resultan O(1) promedio ya que el algoritmo de búsqueda en un hash requiere, en promedio, tiempo de ejecución constante.

Ejemplos (4)

```
struct nodoEmpleado
{  char            *nombre;
   int             nro_seg_social;
   int             nro_empleado;
   T               historial;
   nodoEmpleado    *sigHash_nombre;
   nodoEmpleado    *sigHash_nro_empleado;
   nodoEmpleado    *sigHash_nro_seg_social;
};

struct Estructura
{  nodoEmpleado*   HashNombre[K];
   nodoEmpleado*   HashNroEmpleado[K];
   nodoEmpleado*   HashNroSegSocial[K];
};
```

nodos compartidos entre estructuras

Ejemplos (4)

Para definir el procedimiento supondremos definida las siguiente función:

```
int hNomEmp (char *)
```

función de hash para la tabla por nombre de empleado que retorna un entero entre 0 y K-1.

```
void ImprimirDatosEmpleado (Estructura e, char *nom)
{  nodoEmpleado *listNom;
   listNom = e.HashNombre [hNomEmp(nom)];
   while (listNom != NULL && listNom->nombre != nom)
        (* ojo: usar strcmp para comparar strings *)
        listNom = listNom->sigHash_nombre;
   if (listNom != NULL) {
            cout << listNom->nro_seg_social;
            cout << listNom->nro_empleado;
            cout << listNom->historial…
    }
}
```

Ejemplos (5)

Una institución desea mantener información de los alumnos de una carrera universitaria. Para cada alumno interesa su código de alumno de tipo alfanumérico de 10 caracteres (que se supone único), nombre completo (que se supone único), la colección de materias que aprobó, junto con las notas de aprobación obtenidas, y el conjunto de los compañeros con los cuales realizó algún proyecto a lo largo de su carrera. Cada materia está identificada por un código, que se supone único. Estos códigos son números posibles entre 0 y un entero positivo L. Cada alumno puede tener a lo sumo K compañeros de proyectos diferentes a lo largo de toda su carrera. La relación “compañero de proyecto”, entre dos alumnos, es simétrica (necesariamente). Se sabe que el número de alumnos está acotado por un valor M y se pude asumir que L = O(log M).

Se quieren satisfacer los siguientes requerimientos:

Ejemplos (5)

Nombre **nombre** (*Estructura* e, *CodigoAlumno* cod)

en el cual dada la estructura que representa los datos del problema y el código *cod* de un alumno, devuelve el nombre del alumno correspondiente. Si el código de alumno no existe la operación retorna “NN” como nombre. Esta operación se debe realizar en O(1) caso promedio.

CodigoAlumno **codigo** (*Estructura* e, *Nombre* nom)

en el cual dada la estructura que representa los datos del problema y el nombre *nom* de un alumno, devuelve el código de alumno correspondiente. Si el nombre del alumno no existe la operación retorna “-1” como código. Esta operación se debe realizar en O(1) caso promedio.

Ejemplos (5)

void **listado** (*Estructura* e)

en el cual dada la estructura que representa los datos del problema, imprime los nombres de los alumnos ordenados alfabéticamente. Esta operación se debe realizar en O(n) peor caso, siendo *n* la cantidad actual de alumnos.

void **asignar_compañero** (*Estructura &* e, *Nombre* nom, *CodigoAlumno* cod)

en el cual dada la estructura que representa los datos del problema, el nombre *nom* de un alumno y el código *cod* de otro alumno, los asigna como compañeros de proyecto, siempre que esto sea posible. La operación no tiene efecto si *cod* o *nom* no están registrados como alumnos en la estructura, o si se excede el máximo de compañeros permitidos para *nom* o para el alumno con código *cod*. Asumimos que ambos alumnos no son ya compañeros de proyecto. Esta operación se debe realizar en O(1) caso promedio.

Ejemplos (5)

void **cargar_materia** (*Estructura* & e, *CodigoAlumno* cod, *CodigoMateria* cm, *Calificación* cal)

en el cual dada la estructura que representa los datos del problema, el código *cod* de un alumno, el código *cm* de una materia y una calificación *cal*, agrega a la materia de código *cm*, y con calificación *cal*, a la colección de materias aprobadas por el alumno, siempre que esto esa posible. Si el alumno tiene ya la materia aprobada, actualiza la calificación con *cal*. Esta operación se debe realizar en O(1) caso promedio.

void **agregar_alumno** (*Estructura* & e, *Nombre* nom, *CodigoAlumno* cod)

en el cual dada la estructura que representa los datos del problema, el nombre *nom* de un alumno y su código de alumno *cod*, agrega al alumno en la estructura, con ninguna materia cursada y un conjunto vacío de compañeros de proyectos.

Ejemplos (5)

En caso de encontrar algún alumno con el mismo código *cod* o con el mismo nombre *nom* devuelve un mensaje de error sin modificar la estructura. Esta operación se debe realizar en O(log n) caso promedio, siendo *n* la cantidad actual de alumnos.

Se pide:

a) Diseñar estructuras de datos apropiadas para implementar eficientemente dichas operaciones, describiendo como se obtienen las cotas de tiempo pedidas.

b) Dar una declaración en C/C++ de los tipos de datos descriptos en la parte a).

c) Escribir en C/C++ el procedimiento *agregar_alumno*. A los efectos de definir este procedimiento hay que utilizar solamente las estructuras definidas en la parte a).

Ejemplos (5)

Posible multiestructura simplificada (ver compañeros de proyecto, por ej):

```
struct nodoEstudiante {
    char* nombre;
    char* codigo;
    int cantidad_compa_proyecto; // para simplificar solo llevamos la cantidad
    int notas[L+1]; // se guarda para cada materia la nota y -1 si no tiene (por ejemplo)
    nodoEstudiante* sig_nom;
    nodoEstudiante* sig_cod;
    nodoEstudiante* izq;
    nodoEstudiante* der;
}
struct multiEstructura {
    nodoEstudiante* abb_est; // ABB por nombre de estudiante
    nodoEstudiante* tablaHashNom[M]; // Hash por nombre de estudiante
    nodoEstudiante* tablaHashCod[M]; // Hash por código de estudiante
}
typedef multiEstructura* Estructura;
```

Revisar y completar !