Programa de teoría

Parte I. Estructuras de Datos.

1. Abstracciones y especificaciones.
2. Conjuntos y diccionarios.
3. **Representación de conjuntos mediante árboles.**
4. Grafos.

Parte II. Algorítmica.

1. Análisis de algoritmos.
2. Divide y vencerás.
3. Algoritmos voraces.
4. Programación dinámica.
5. Backtracking.
6. Ramificación y poda.

PARTE I: ESTRUCTURAS DE DATOS

Tema 3. Representación de conjuntos mediante árboles.

3.1. Árboles Trie.

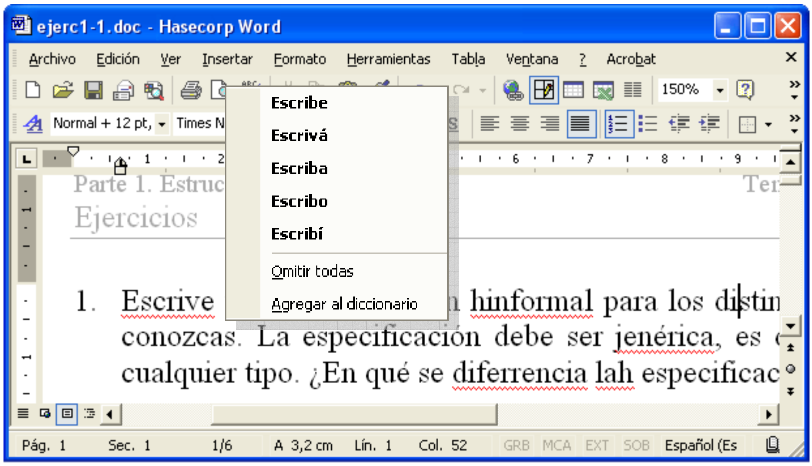
3.2. Relaciones de equivalencia.

3.3. Árboles de búsqueda balanceados.

3.4. Árboles B.

3.1. Árboles Trie.

- **Aplicación**: representación de diccionarios (o en general conjuntos) grandes de palabras.
- **Ejemplo**. Corrector ortográfico interactivo.



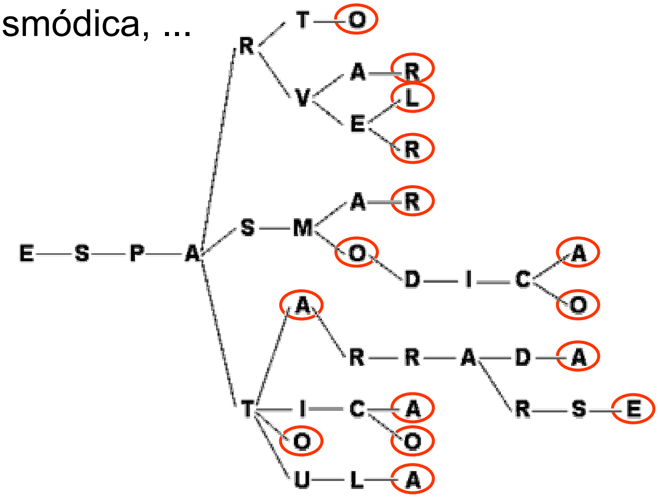
3.1. Árboles Trie.

- **Diccionario español:** ~ 3 millones de palabras.
- Muchas palabras → Mucha memoria y operaciones lentas.
- Pero la búsqueda de una palabra no puede tardar más de 1 milisegundo...

... esparto esparvar esparvel esparver espasmar espasmo espasmódica espasmódico espata espatarrada espatarrarse espática espático espato espátula espatulomancia espaviento espavorecida espavorecido espavorida espavorido espay especería especia especial ...

3.1. Árboles Trie.

- **Idea**: muchas palabras tienen prefijos comunes. P. ej.: espasmar, espasmo, espasmódico, espasmódica, ...



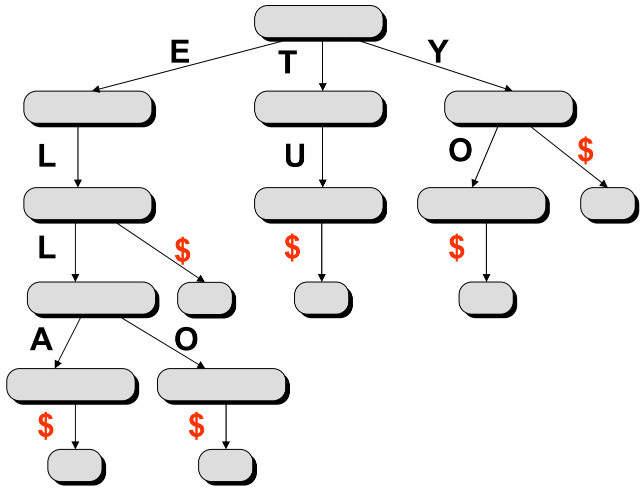
3.1. Árboles Trie.

- Un Trie es, básicamente, un árbol de prefijos.
- Sea **A** un alfabeto. Por ejemplo A= {a, b, c, ..., z}
- Añadimos a **A** una marca de fin de palabra: **$**.

- **Definición:** un **Trie** es una estructura de árbol en la que:
  1. La **raíz** del árbol representa la cadena vacía.
  2. Un nodo puede tener tantos hijos como caracteres del alfabeto **A** más uno. Cada hijo está etiquetado con un carácter o una marca de fin $.
  3. La sucesión de etiquetas desde la raíz hasta un nodo hoja, etiquetado con la marca de fin $, representa una **palabra**.
  4. A todos los nodos, excepto a la raíz y a las hojas etiquetadas con $, se les denomina **prefijos** del árbol.

3.1. Árboles Trie.

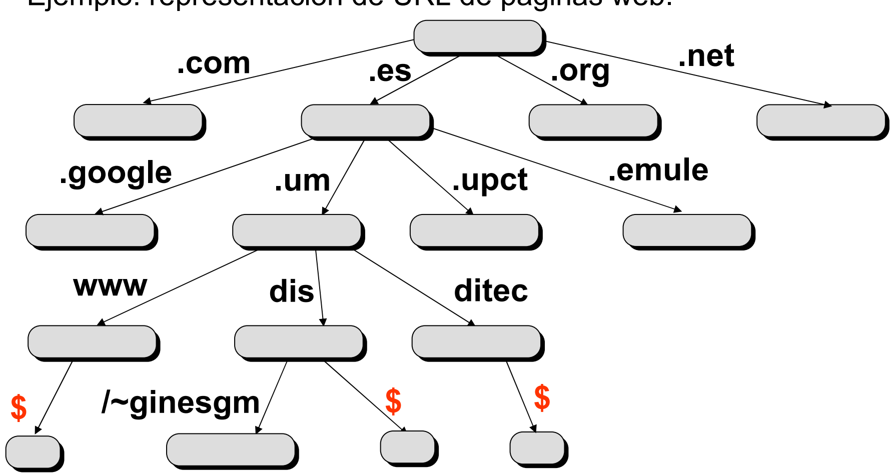
- Ejemplo, C= {ELLA, ELLO, EL, TU, Y, YO}



- ¿Cómo usarlo en el corrector interactivo?

3.1. Árboles Trie.

- Se pueden representar otros tipos de información, cambiando el alfabeto **A**.
- Ejemplo: representación de URL de páginas web.



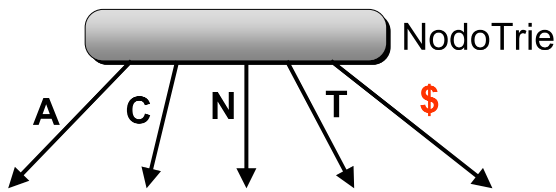
3.1.1. Representación de tries.

- **Cuestión:** ¿Cómo representar árboles trie?

  tipo

  ArbolTrie[A]= Puntero[NodoTrie[A]]

- **Reformulamos** la pregunta: ¿Cómo representar los **nodos** del árbol trie?



3.1.1. Representación de tries.

- Un **NodoTrie[A]** es un **Diccionario**[*tclave*, *tvalor*], donde *tclave*= A y *tvalor*= Puntero[NodoTrie[A]]
- **Operaciones**:

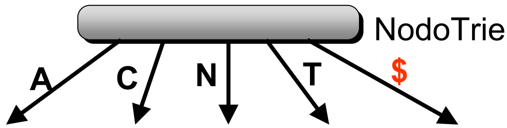
  Inserta (**var n**: NodoTrie[A]; **caract**: A;
  ptr: Puntero[NodoTrie[A]])

  Consulta (**n**: NodoTrie[A]; **caract**: A):
  Puntero[NodoTrie[A]]

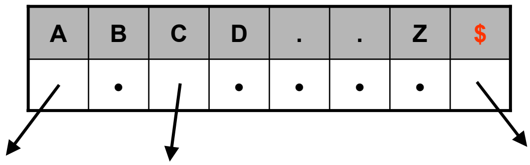
  Anula (**var n**: NodoTrie[A])

  TomaNuevo (**var n**: NodoTrie[A]; **caract**: A)

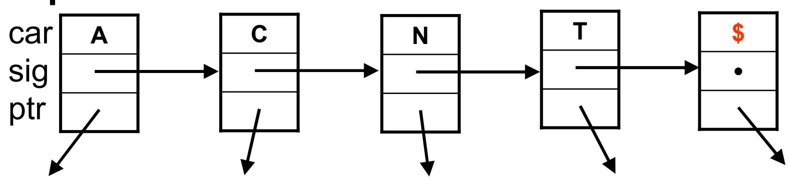
3.1.1. Representación de tries.



- **Representación** mediante arrays.

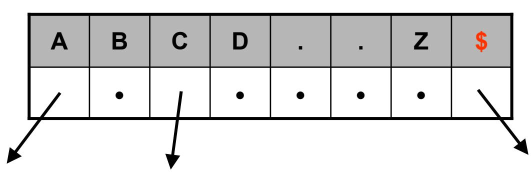


- **Representación** mediante listas.



3.1.1. Representación de tries.

- **Representación** mediante arrays.



tipo

NodoTrie[A]= **array** [A] **de** Puntero[NodoTrie[A]]

- **Ventaja**: acceso muy rápido a los valores.
- **Inconveniente**: desperdicia muy mucha memoria.

3.1.1. Representación de tries.

Inserta (**var n**: NodoTrie[A]; **car**: A; **ptr**: Puntero[NodoTrie[A]])
    n[car]:= ptr

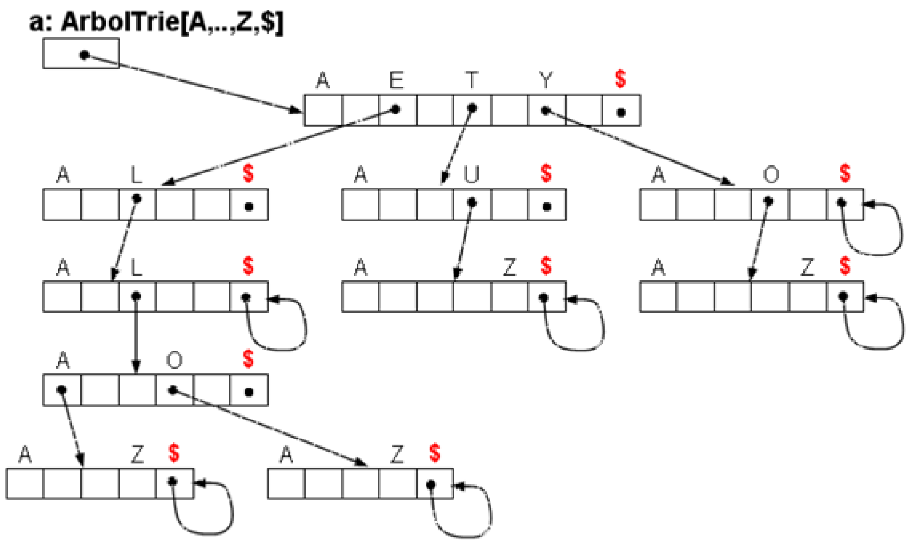
Consulta (**n**: NodoTrie[A]; **car**: A): Puntero[NodoTrie[A]]
    devolver n[car]

Anula (**var n**: NodoTrie[A])
    para i en Rango(A) **hacer**
        n[i]:= NULO

TomaNuevo (**var n**: NodoTrie[A]; **car**: A)
    n[car]:= NUEVO NodoTrie[A]
    Anula (n[car])

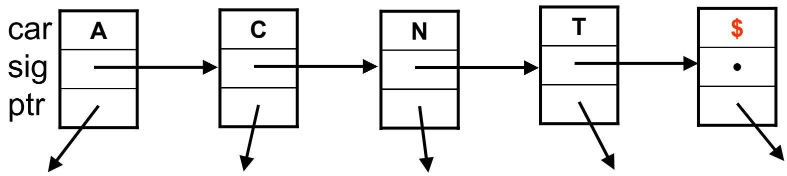
3.1.1. Representación de tries.

- Ejemplo, C= {ELLA, ELLO, EL, TU, Y, YO}



3.1.1. Representación de tries.

- **Representación** mediante listas.



tipo NodoTrie[A]= **registro**
car: A
sig, ptr: Puntero[NodoTrie[A]]
finregistro

- **Ventaja**: uso razonable de memoria.
- **Inconveniente**: operaciones más lentas.

3.1.1. Representación de tries.

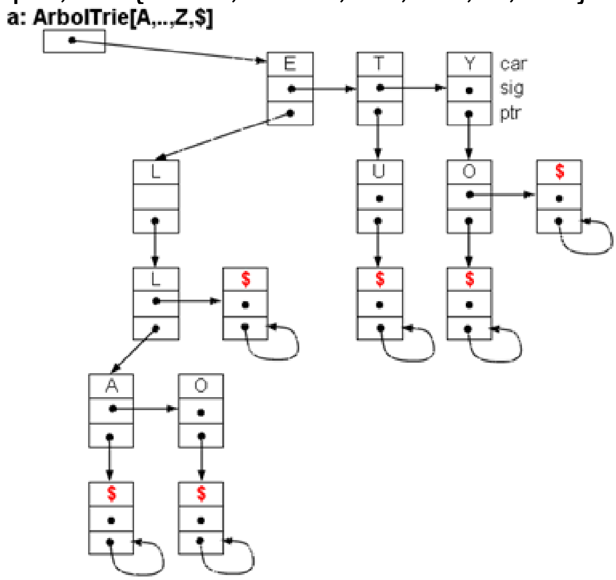
Consulta (**n**: NodoTrie[A]; **car**: A): Puntero[NodoTrie[A]]
tmp:= PunteroA(n)
mientras tmp ≠ NULO AND tmp→car < c **hacer**
tmp:= tmp→sig
si tmp→car ≠ c **entonces devolver** NULO
sino devolver tmp→ptr

Inserta (**var n**: NodoTrie[A]; **car**: A; **ptr**: Puntero[NodoTrie[A]])

1. Recorrer la lista buscando el carácter **car**
2. Si se encuentra, modificar el puntero **ptr**
3. En otro caso, añadir un nuevo nodo en la posición adecuada, con el carácter **car** y el puntero **ptr**

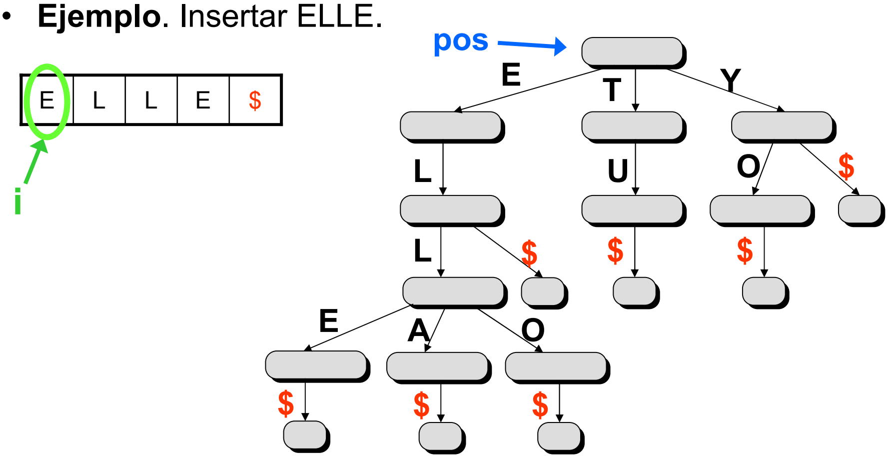
3.1.1. Representación de tries.

- Ejemplo, C= {ELLA, ELLO, EL, TU, Y, YO}



3.1.2. Operaciones con tries.

- Utilizando la representación de nodos trie (con listas o con arrays) implementar las operaciones de inserción, eliminación y consulta sobre el trie.
- **Ejemplo**. Insertar ELLE.



3.1.2. Operaciones con tries.

operación Inserta (**var** a: ArbolTrie[A]; s: cadena)
var pos: Puntero[NodoTrie[A]]

    i:= 1
    pos:= a
    mientras s[i] ≠ $ **hacer**
        si Consulta (pos↑, s[i]) == NULO **entonces**
            TomaNuevo (pos↑, s[i])
        s:= Consulta (pos↑, s[i])
        i:= i + 1
    finmientras
    Inserta (pos↑, $, pos)

- Modificar el procedimiento para que haga una consulta.
- Si queremos añadir información asociada a cada palabra, ¿dónde debería colocarse?

3.1.2. Operaciones con tries.

- ¿Cómo sería el uso del trie en el corrector interactivo?
- **Empezar una palabra**
  Colocar **pos** en la raíz del árbol
- **Pulsar una tecla c en una palabra**
  Si Consulta (pos↑, c) == NULO entonces la palabra es incorrecta, en otro caso moverse en el árbol
- **Acabar una palabra**
  Si Consulta (pos↑, $) == NULO entonces la palabra es incorrecta, en otro caso es correcta
- **Borrar una letra de una palabra**
  Moverse hacia atrás en el árbol...

3.1.3. Evaluación de los tries.

Tiempo de ejecución

- El principal factor en el tiempo de ejecución es la longitud de las palabras: **m**.
- Nodos con **arrays**: **O(m)**
- Nodos con **listas**: **O(m*s)**, donde **s** es la longitud promedio de las listas. En la práctica, ~ **O(m)**.

- ¿Cómo es el tiempo en comparación con las tablas de dispersión?

- En el caso del corrector interactivo, la eficiencia es aún más interesante.

3.1.3. Evaluación de los tries.

Uso de memoria

- Longitud promedio de las palabras: **m**. Longitud total: **l**
- Número de palabras: **n.** Número de prefijos: **p**
- k_1 bytes/puntero, k_2 bytes/carácter
- **d** caracteres en el alfabeto (incluido $)
- **n << p << l**

- **Nodos con arrays: $d*k_2$ (p + 1) bytes**
  - **p+1** Nodos en el árbol
  - $d*k_2$ bytes por nodo
- **Nodos con listas: $(2k_1 + k_2)(n + p)$ bytes**
  - **n + p** Nodos en el árbol
  - $2k_1 + k_2$ bytes por nodo

3.1.3. Evaluación de los tries.

Uso de memoria

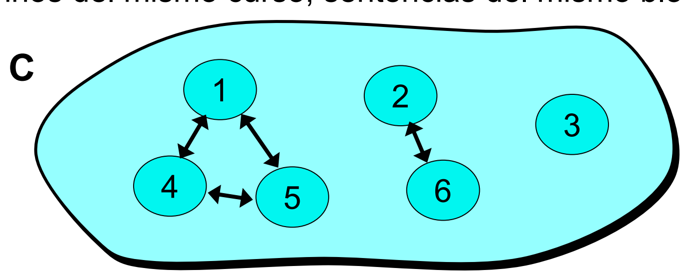
- **Con listas simples: $2k_1$*n + k_2*l bytes**
- La eficiencia de memoria depende de la relación **l/p**
  - Si **l/p** es grande: las palabras comparten muchos prefijos.
  - Si **l/p** es pequeña: hay pocos prefijos compartidos y se gasta mucha memoria.
- En la práctica, mejora ≈ l/p > 6

Conclusiones

- La estructura es adecuada en aplicaciones donde aparezcan muchos prefijos comunes.
- El tiempo de ejecución sólo depende (casi) de la longitud de las palabras, ¡independientemente de cuántas hayan!

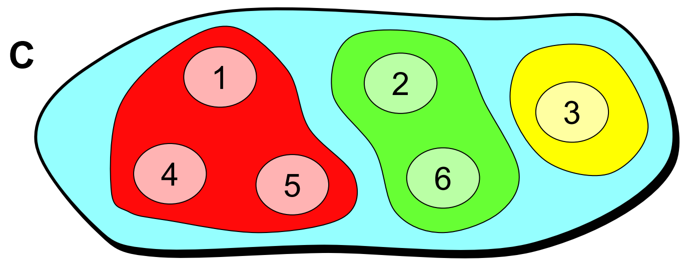
3.2. Relaciones de equivalencia.

- **Definición:** Una **relación de equivalencia** en un conjunto **C** es una relación **R** que satisface:
  - **Reflexiva:** a R a, $\forall$ a $\in$ C.
  - **Simétrica:** a R b $\Leftrightarrow$ b R a.
  - **Transitiva**: Si (a R b) y (b R c) entonces a R c.
- **Ejemplos:** relación de ciudades en el mismo país, alumnos del mismo curso, sentencias del mismo bloque.



3.2. Relaciones de equivalencia.

- **Definición:** La **clase de equivalencia** de un elemento **a** ∈ **C**, es el subconjunto de **C** que contiene todos los elementos relacionados con **a**.
- Las clases de equivalencia forman una **partición** de **C** (subconjuntos disjuntos y completos).

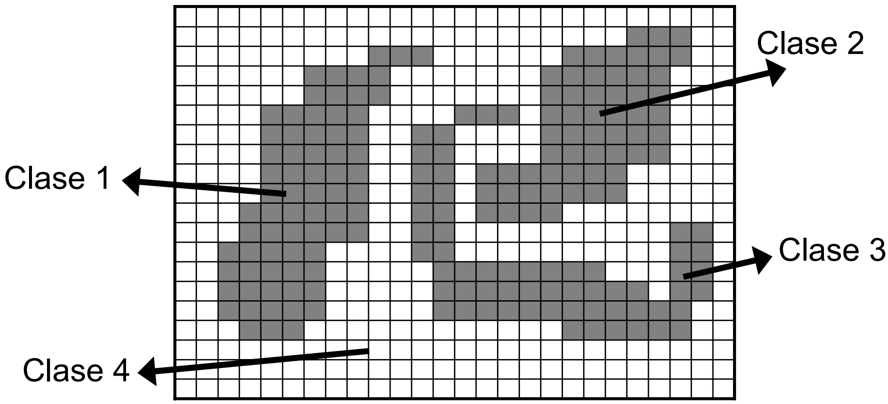


3.2. Relaciones de equivalencia.

- Definimos un **TAD** para las relaciones de equivalencia, sobre un conjunto C.
- **Operaciones:**
  - **Crear (C: Conjunto[T]) : RelEquiv[T]**
    Crea una relación vacía, en la que cada elemento es una clase de equivalencia en sí mismo.
  - **Unión (var R: RelEquiv[T]; a, b: T)**
    Combina dos clases de equivalencia (las de **a** y **b**) en una nueva. Es una unión de conjuntos disjuntos.
  - **Encuentra (R: RelEquiv[T]; a: T) : T**
    Devuelve la clase a la que pertenece **a**.
- **Ojo:** el “nombre” de la clase es también de tipo T. Puede ser un elemento cualquiera de esa clase.

3.2. Relaciones de equivalencia.

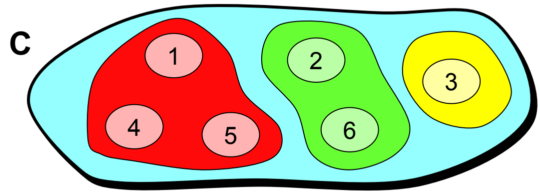
- **Ejemplo de aplicación:** procesamiento de imágenes.
- **Relación:** Dos píxeles están relacionados si son adyacentes y tienen el mismo color.



3.2. Relaciones de equivalencia.

- Imagen de 800 x 600 = 480.000 píxeles
- El conjunto contiene medio millón de elementos. Las operaciones Unión y Encuentra son muy frecuentes.
- **Observaciones:**
  - Sólo es necesario conocer en qué clase de equivalencia está cada elemento.
  - El nombre de la clase es arbitrario, lo que importa es que **Encuentra(x)** = **Encuentra(y)** si y sólo si **x** e **y** están en la misma clase de equivalencia.
- ¿Cómo implementar el tipo Relación de Equivalencia de forma eficiente?

3.2.1. Representaciones sencillas.



- **Representación mediante un array.** Para cada elemento **i** indicar la clase a la que pertenece.

R : array [1..6]

1	2	3	4	5	6
1	2	3	1	1	2

3.2.1. Representaciones sencillas.

Representaciones mediante un array

- **Encuentra (R: RelEquiv[T]; a: T) : T**
  devolver R[a]

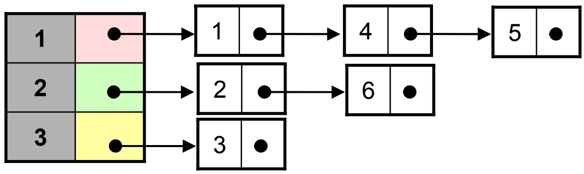
- **Unión (var R: RelEquiv[T]; a, b: T)**
  Recorrer todo el array, cambiando donde ponga **b** por **a**...

- **Resultado:**
  – La búsqueda de la clase de equivalencia es muy rápida.
  – La unión de clases de equivalencia es muy lenta.

3.2.1. Representaciones sencillas.

Representaciones mediante listas de clases

- Para cada clase una lista de sus miembros.



- **Unión (var R: RelEquiv[T]; a, b: T)**
  Concatenar dos listas. Se puede conseguir en un **O(1)**, con una representación adecuada de las listas.

3.2.1. Representaciones sencillas.

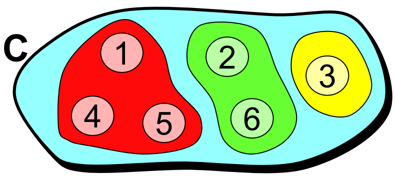
Representaciones mediante listas de clases

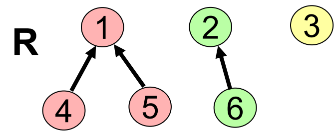
- **Encuentra (R: RelEquiv[T]; a: T) : T**
  Recorrer todas las listas hasta encontrar **a**. El tiempo es **O(N)**, siendo **N** el número de elementos.

- **Resultado:**
  - La unión de clases de equivalencia es muy rápida.
  - La búsqueda de la clase de equivalencia es muy lenta.

- **Solución:** usar una estructura de árboles.
  - Un árbol para cada clase de equivalencia.
  - El nombre de la clase viene dado por la raíz del árbol.

3.2.2. Representación mediante árboles.





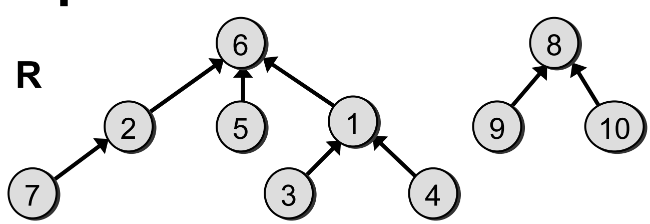
- Usamos una representación de árboles mediante **punteros al padre**.

  tipo

  RelEquiv[N] = **array** [1..N] **de** entero

- **R[x]** == 0, si **x** es una raíz del árbol.
- En otro caso, **R[x]** contiene el padre de **x**.

3.2.2. Representación mediante árboles.



R : Rel-Equiv[10]

1	2	3	4	5	6	7	8	9	10
6	6	1	1	6	0	2	0	8	8

- **Unir dos clases (raíces):** apuntar una a la otra.
- **Buscar la clase de un elemento:** subir por el árbol hasta llegar a la raíz.
- ±

3.2.2. Representación mediante árboles.

operación Crear (N: entero) : RelEquiv[N]
    para cada i:= 1, ..., N **hacer**
        R[i]:= 0
    devolver R

operación Unión (**var** R: RelEquiv[N]; a, b: entero)
    R[a]:= b

operación Encuentra (R: RelEquiv[N]; a: entero) : entero
    si R[a]==0 **entonces**
        devolver a
    sino devolver Encuentra (R, R[a])

- El procedimiento **Unión** supone que **a** y **b** son raíces de los árboles. ¿Cómo sería la operación si no lo son?

3.2.2. Representación mediante árboles.

- **Ejemplo.** Iniciar una relación R[6] vacía y aplicar: Unión(3, 4), Unión (6, 5), Unión (4, 5), Unión (5, 2), Unión (1, 2).

R : Rel-Equiv[10]

1	2	3	4	5	6
0	0	0	0	0	0

3.2.2. Representación mediante árboles.

Eficiencia de las operaciones

- La operación **Unión** tiene un **O(1)**.
- En el caso promedio la operación **Encuentra** es de orden menor que **O(log N)**.
- Sin embargo, en el peor caso los árboles son cadenas y el coste es **O(N)**.

- Debemos garantizar que los árboles sean lo más anchos posible.
- **Idea:** Al unir **a** y **b** se puede poner **a** como hijo de **b**, o al revés. **Solución:** Colocar el menos alto como hijo del más alto.

3.2.3. Balanceo del árbol y compresión.

- **Modificación:** Si un nodo **x** es raíz, **R[x]** indica (con números negativos) la profundidad de su árbol.
- Al unir dos raíces, apuntar la de menor profundidad a la de mayor (**balanceo del árbol**).

operación Unión (**var** R: RelEquiv[N]; a, b: entero)
    si R[a] < R[b] **entonces** R[b]:= a
    sino
        si R[a]==R[b] **entonces**
            R[b]:= R[b] – 1
        finsi
        R[a]:= b
    finsi

3.2.3. Balanceo del árbol y compresión.

- **Ejemplo.** Iniciar una relación R[6] vacía y aplicar: Unión(3, 4), Unión (6, 5), Unión (4, 5), Unión (5, 2), Unión (1, 5).

R : Rel-Equiv[10]

1	2	3	4	5	6
0	0	0	0	0	0

3.2.3. Balanceo del árbol y compresión.

- **Segunda idea:** Si aplicamos Encuentra(R, a) y encontramos que la clase de a es x, podemos hacer R[a]:= x **(compresión de caminos)**.

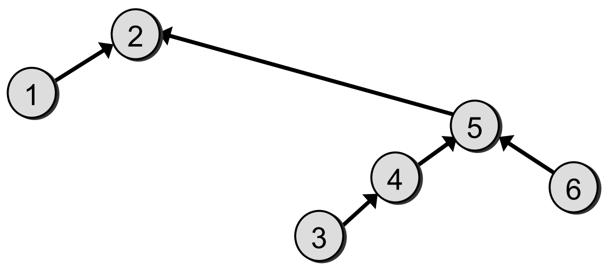
```
operación Encuentra (R: RelEquiv[N]; a: entero) : entero
    si R[a] ≤ 0 entonces
        devolver a
    sino
        R[a]:= Encuentra (R, R[a])
        devolver R[a]
    finsi
```

3.2.3. Balanceo del árbol y compresión.

- **Ejemplo.** Aplicar Encuentra(R,3), Encuentra(R,6).

R : Rel-Equiv[10]

1	**2**	**3**	**4**	**5**	**6**
2	-3	4	5	2	5



- **Ojo.** No se recalcula la altura en la raíz.

3.2.3. Balanceo del árbol y compresión.

Tiempo de ejecución

- El tiempo de la operación Unión es **O(1)**.
- El tiempo de Encuentra está entre **O(1)** y **O(log N)**.

Conclusiones

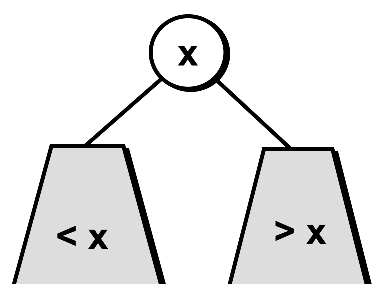
- La estructura de datos usada es un array (exactamente igual que la solución sencilla).
- Pero ahora el array es manejado como un **árbol (árbol de punteros al padre)**.
- Para conseguir eficiencia es necesario garantizar que el árbol está **equilibrado**.

3.3. Árboles de búsqueda balanceados.

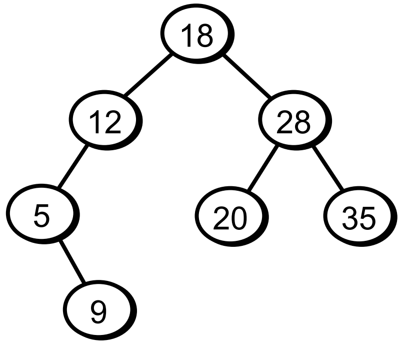
- **Problema general de representación de conjuntos y diccionarios**:
  - **Tablas de dispersión:** Acceso rápido a un elemento concreto, pero recorrido secuencial u ordenado lento.
  - **Listas:** Recorrido secuencial eficiente, pero acceso directo muy lento.
  - **Arrays:** Problemas con el uso de memoria.
  - **Tries:** Específicos de aplicaciones donde aparecen muchos prefijos comunes.
- **Solución:** Utilizar árboles. En concreto, árboles de búsqueda.

3.3. Árboles de búsqueda balanceados.

- **Árboles binarios de búsqueda (ABB).**
  - Cada nodo tiene cero, uno o dos hijos, denominados **hijo izquierdo** e **hijo derecho**.
  - Los hijos de un nodo **x** con valores menores que **x** se encuentran en el subárbol izquierdo y los mayores en el derecho.

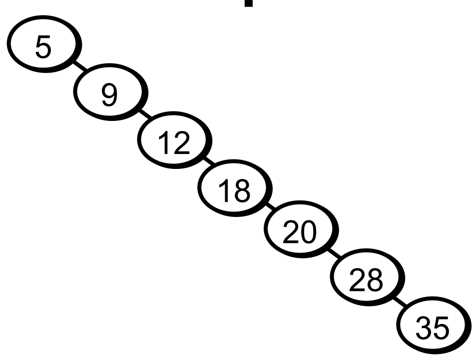


3.3. Árboles de búsqueda balanceados.



- Son útiles para realizar búsqueda e inserción en **O(log n)** y recorrido ordenado en **O(n)**.
- **Inconveniente:** En el peor caso los árboles son cadenas y la búsqueda necesita **O(n)**.

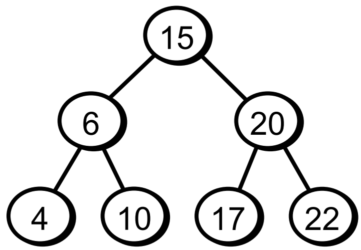
3.3. Árboles de búsqueda balanceados.



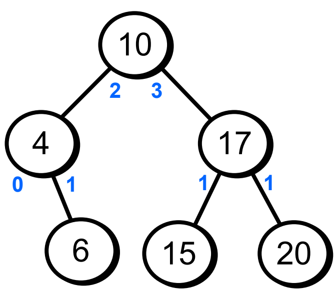
- **Conclusión:** Es necesario garantizar que el árbol está balanceado o equilibrado.
- **Condición de balanceo:** Basada en número de nodos o en altura de subárboles.

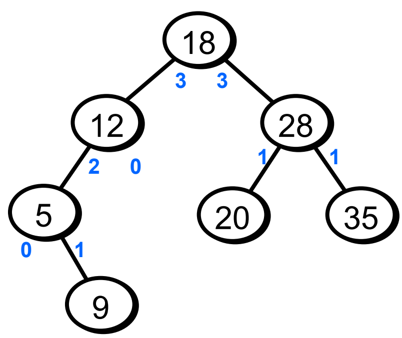
3.3. Árboles de búsqueda balanceados. Árbol de búsqueda perfectamente balanceado

- **Definición:** Un **ABB perfectamente balanceado** es un ABB donde, para todo nodo, la cantidad de nodos de su subárbol izquierdo difiere como máximo en 1 de la cantidad de nodos del subárbol derecho.



3.3. Árboles de búsqueda balanceados.

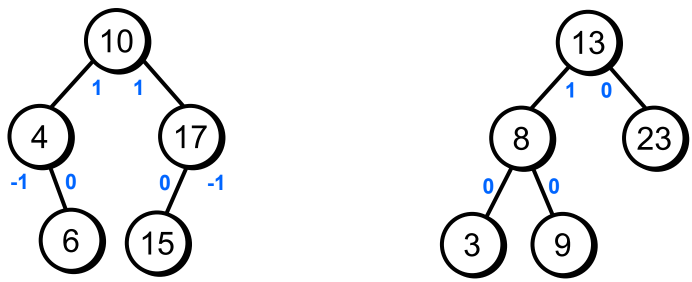




- **Resultado:**
  - La búsqueda es **O(log n)** en el peor caso.
  - Pero mantener la condición de balanceo es muy costoso. La inserción puede ser **O(n)**.

3.3. Árboles de búsqueda balanceados.

- **Moraleja:** definir una condición de balanceo, pero menos exigente.
- **Definición de árbol balanceado ó AVL (Adelson-Velskii y Landis)**: Un **AVL** es un ABB donde, para todo nodo, la **altura** de sus subárboles difiere como máximo en 1.



3.3. Árboles de búsqueda balanceados.

Operaciones sobre un AVL

- La **búsqueda** en un AVL es exactamente igual que sobre un ABB.
- La **inserción** y **eliminación** son también como en un ABB, pero después de insertar o eliminar hay que comprobar la condición de balanceo.
  - Almacenar la altura de cada subárbol.
  - Inserción o eliminación normal (procedimiento recursivo).
  - Al volver de la recursividad, en los nodos por los que pasa, comprobar la condición de balanceo.
  - Si no se cumple, **rebalancear** el árbol.

3.3. Árboles de búsqueda balanceados.

- **Definición del tipo de datos:**

tipo
ArbolAVL[T] = Puntero[NodoAVL[T]]
NodoAVL[T] = **registro**
clave: T
altura: entero
izq, der: ArbolAVL[T]
finregistro

altura	izq	clave	der
h		x	

operación Altura (A: ArbolAVL[T]) : entero
si A == NULO **entonces devolver** -1
sino devolver A→altura

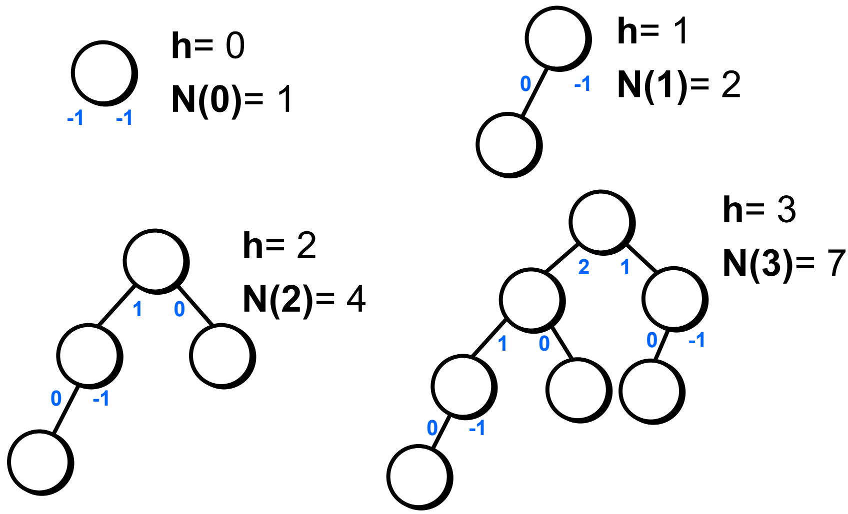
- **Uso de memoria:** un puntero más que con una lista... y un entero más, por nodo, que un ABB normal...

3.3.1. Peor caso de AVL.

- ¿Cuánto será el tiempo de ejecución de la búsqueda en un AVL en el peor caso, para **n** nodos?
- El tiempo será proporcional a la altura del árbol.
- **Cuestión:** ¿Cuál es la máxima altura del árbol para **n** nodos?
- Le **damos la vuelta** a la pregunta: ¿Cuál es el mínimo número de nodos para una altura **h**?

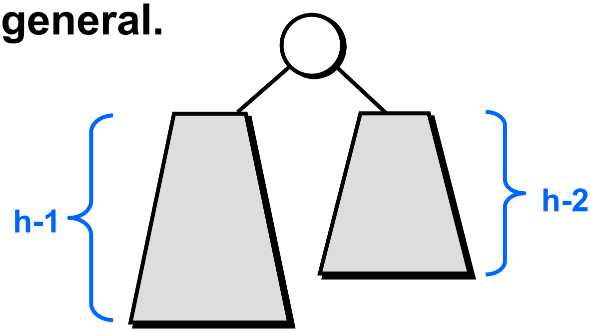
3.3.1. Peor caso de AVL.

- **N(h)**: Menor número de nodos para altura **h**.



3.3.1. Peor caso de AVL.

- **Caso general.**



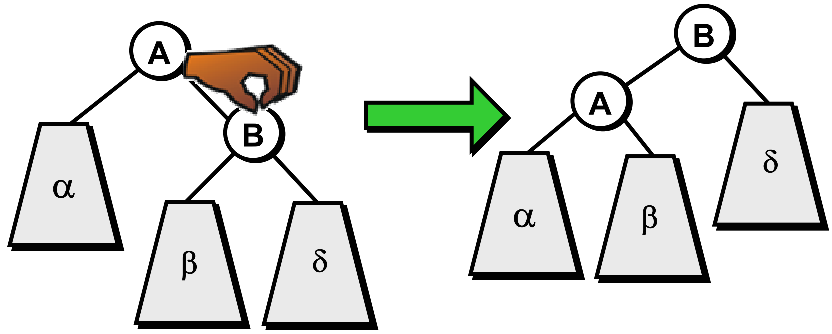
- N(h) = N(h-1) + N(h-2) + 1
- Sucesión parecida a la de **Fibonacci**.
- **Solución:** N(h) = C·$1{,}62^h$ + ...

3.3.1. Peor caso de AVL.

- **Mínimo número de nodos para altura h:** $N(h) = C \cdot 1{,}62^h + \ldots$
- **Máxima altura para n nodos:** $h(N) = D \cdot \log_{1,62} n + \ldots$
- **Conclusión:**
  - En el peor caso, la altura del árbol es **O(log n)**.
  - Por lo tanto, la búsqueda es **O(log n)**.
  - Inserción y eliminación serán de **O(log n)** si el **rebalanceo** se puede hacer en **O(1)**.

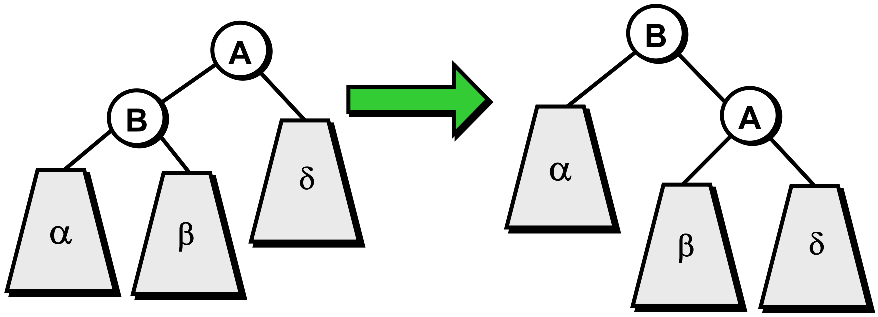
3.3.2. Rotaciones en un AVL.

- Los rebalanceos en un AVL hacen uso de operaciones conocidas como **rotaciones en ABB**.
- **Rotación:** cambiando algunos punteros, obtener otro árbol que siga siendo un ABB.
- **RSD(A). Rotación simple a la derecha de un ABB**



3.3.2. Rotaciones en un AVL.

- **RSI(A). Rotación simple a la izquierda de un ABB**



- Programar las operaciones de rotación simple.

3.3.2. Rotaciones en un AVL.

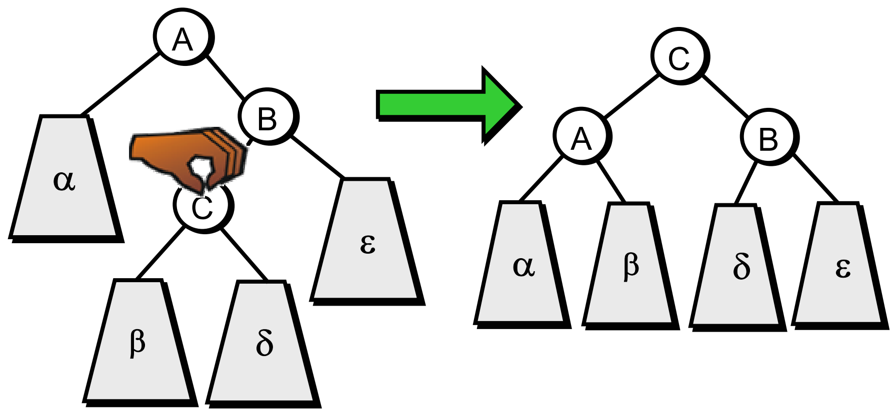
operación RSI (var A: ArbolAVL[T])

B:= A→izq
A→izq:= B→der
B→der:= A
A→altura:= 1+max(Altura(A→izq), Altura(A→der))
B→altura:= 1+max(Altura(B→izq), A→altura)
A:= B

- ¿Cuánto es el tiempo de ejecución de una rotación simple?

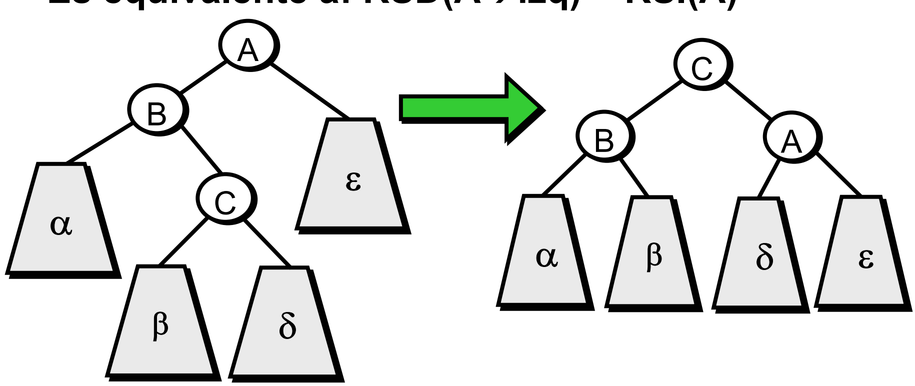
3.3.2. Rotaciones en un AVL.

- **RDD(A). Rotación doble a la derecha de un ABB**
  Es equivalente a: RSI(A→der) + RSD(A)



3.3.2. Rotaciones en un AVL.

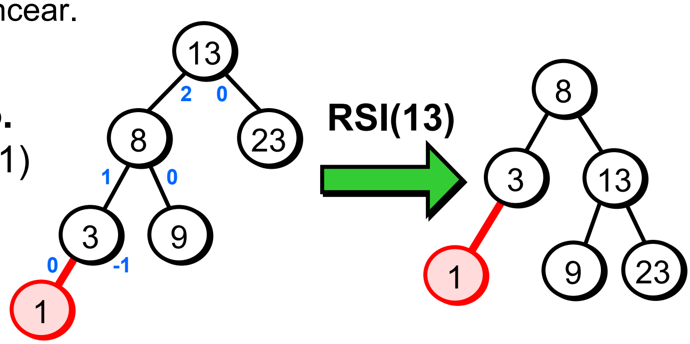
- **RDI(A). Rotación doble a la izquierda de un ABB**
  Es equivalente a: RSD(A→izq) + RSI(A)



- Todas las rotaciones mantienen la estructura de ABB y son **O(1)**.

3.3.3. Operación de inserción en un AVL.

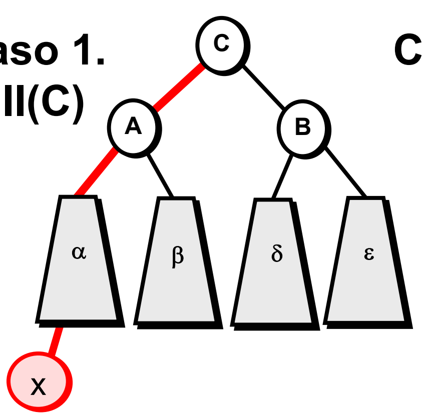
- Inserción normal como en un ABB.
- En cada nodo **A** (a la vuelta de la recursividad), si la altura del árbol no se modifica, acabar.
- Si la altura se incrementa en 1 entonces:
  - Si |Altura(A→izq) – Altura(A→der)|>1 entonces rebalancear.
- **Ejemplo.** Insertar(1)



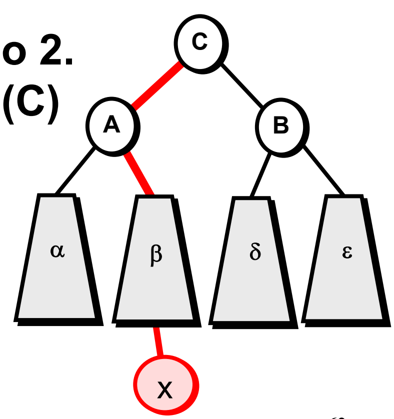
3.3.3. Operación de inserción en un AVL.

- ¿Qué rotación aplicar en cada caso de desbalanceo?
- Se pueden **predefinir 4 situaciones** diferentes, cada una asociada con un tipo de rotación.

Caso 1. II(C)



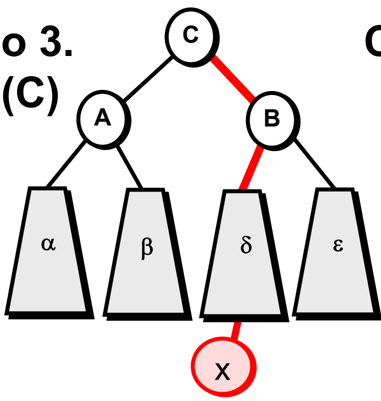
Caso 2. ID(C)



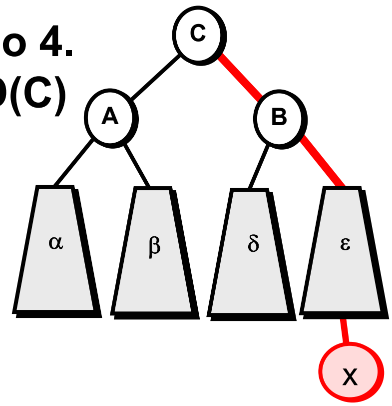
3.3.3. Operación de inserción en un AVL.

- ¿Qué rotación aplicar en cada caso de desbalanceo?
- Se pueden **predefinir 4 situaciones** diferentes, cada una asociada con un tipo de rotación.

Caso 3. DI(C)

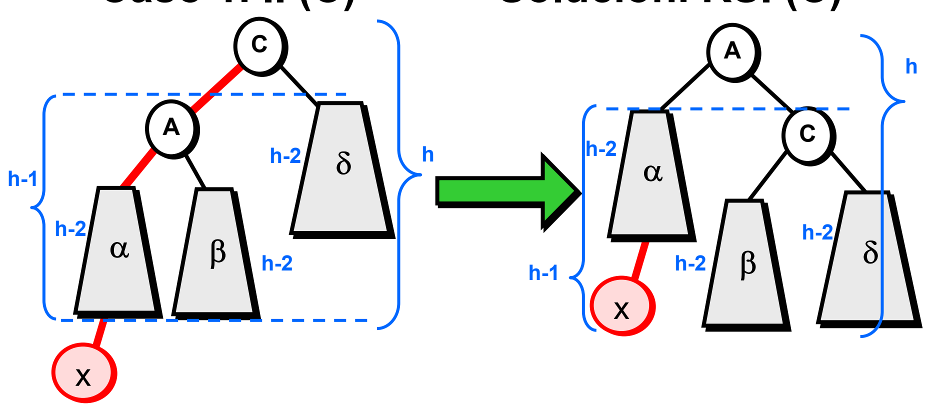


Caso 4. DD(C)



3.3.3. Operación de inserción en un AVL.

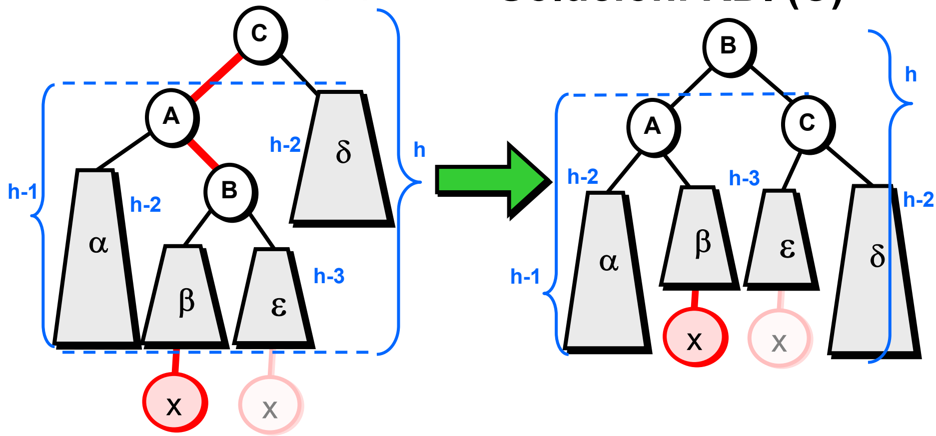
Caso 1. II (C) **Solución. RSI (C)**



- El árbol resultante está balanceado.
- Adicionalmente, la altura del árbol no cambia.

3.3.3. Operación de inserción en un AVL.

Caso 2. ID (C) **Solución. RDI (C)**



- La altura final del árbol tampoco cambia.

3.3.3. Operación de inserción en un AVL.

operación Inserta (var A : ArbolAVL[T]; x : T)

```
si A == NULO entonces
    A:= NUEVO NodoAVL[T]
    A→clave:= x
    A→der:= A→izq:= NULO
    A→altura:= 0
sino  // Subárbol izquierdo
    si x < A→clave entonces
        ...
sino  // Subárbol derecho
    si x > A→clave entonces
        ...
finsi
```

```
Inserta (A→izq, x)
si Altura(A→izq) –
        Altura(A→der)>1 entonces
    si x < A→izq→clave entonces
        RSI (A)    // Caso II(A)
    sino
        RDI (A)    // Caso ID(A)
    finsi
sino
    A→altura:= 1+max(
        Altura(A→izq), Altura(A→der))
finsi
```

3.3.3. Operación de inserción en un AVL.

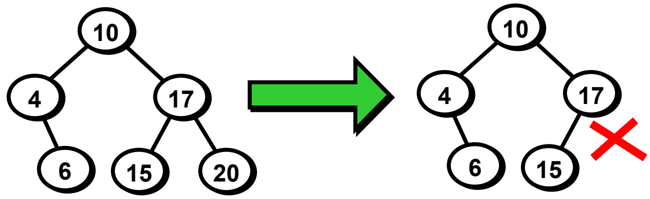
- El procedimiento sigue recursivamente hasta la raíz.
- Pero cuando se haga el primer balanceo no será necesario hacer otros balanceos. ¿Por qué?
- **Ejemplo**: Dado un árbol nuevo insertar 4, 5, 7, 2, 1, 3, 6.
- ¿Cuál es el orden de complejidad del algoritmo de Inserta?

3.3.4. Operación de eliminación en un AVL.

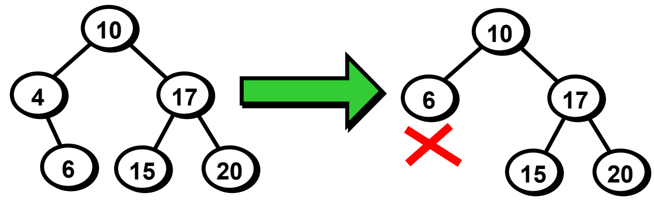
- La **eliminación** de un nodo es algo más compleja. Hay más casos y puede ser necesario balancear en varios niveles distintos.
- **Algoritmo de eliminación:** Eliminación normal en ABB + comprobación de la condición.
- **Eliminación normal en un ABB**. Buscar el elemento a eliminar en el árbol.
  - Si es un nodo hoja se elimina directamente.
  - Si el nodo eliminado tiene un solo hijo, conectar el padre del nodo eliminado con ese hijo.
  - Si el nodo eliminado tiene dos subárboles, escoger el nodo más a la derecha del subárbol izquierdo (o el más a la izquierda del subárbol derecho).

3.3.4. Operación de eliminación en un AVL.

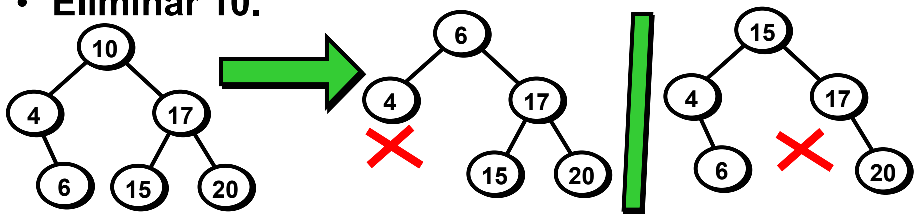
- **Eliminar 20.**



- **Eliminar 4.**

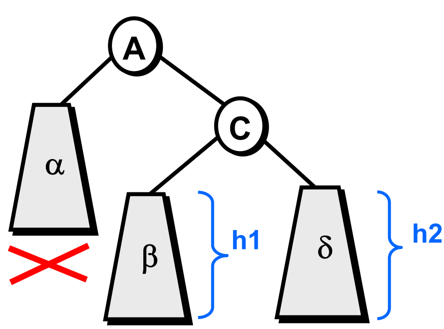


- **Eliminar 10.**



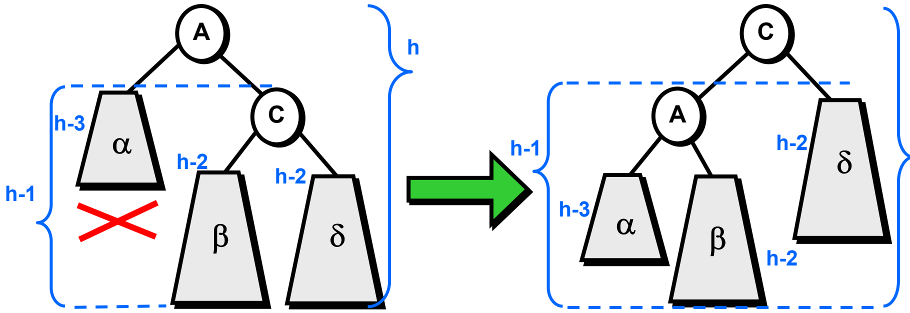
3.3.4. Operación de eliminación en un AVL.

- Después de eliminar un nodo, volver a los nodos antecesores (recursivamente).
- Comprobar si cumple la condición de balanceo.
- En caso negativo **rebalancear**.
- Se pueden predefinir **3 casos** de eliminación en subárbol izquierdo, y los simétricos en subárbol derecho.
- **Ojo:** Los casos de desbalanceo en subárbol izquierdo de **A** dependen de las alturas **h1** y **h2** en el subárbol derecho.



3.3.4. Operación de eliminación en un AVL.

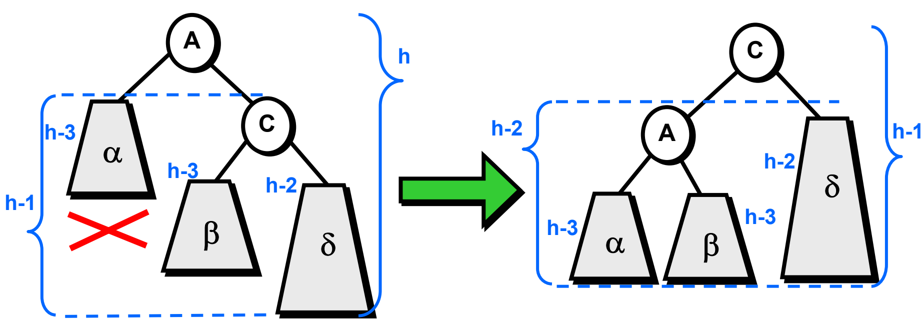
Caso 1. h1=h2 Solución. RSD (A)



- El árbol resultante está balanceado.
- La altura del árbol no cambia.

3.3.4. Operación de eliminación en un AVL.

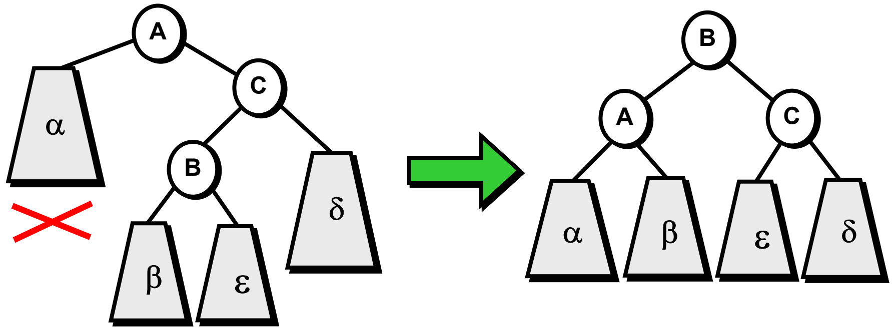
Caso 2. h1<h2 Solución. RSD (A)



- En este caso, la altura del árbol disminuye en 1.

3.3.4. Operación de eliminación en un AVL.

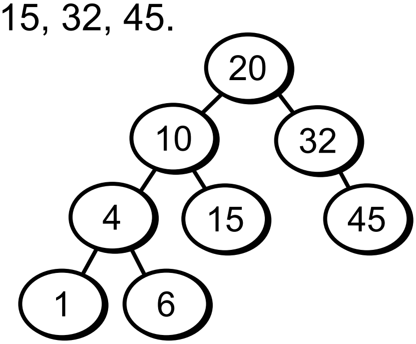
Caso 3. h1>h2 Solución. RDD (A)



- Comprobar (mediante el cálculo de las alturas) que el árbol resultante está balanceado.
- La altura final del árbol disminuye en 1.

3.3.4. Operación de eliminación en un AVL.

- Ejercicio: implementar la operación de eliminación en un AVL.
- ¿Cuál es el orden de complejidad?
- **Ejemplo**: Dado el siguiente AVL, eliminar las claves: 4, 15, 32, 45.



3.3. Árboles de búsqueda balanceados.

Conclusiones:

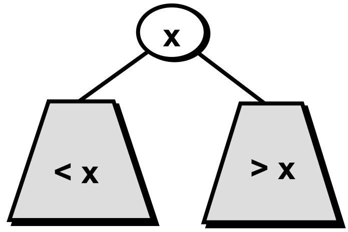
- La idea de los árboles binarios de búsqueda está muy bien.
- Pero para que funcionen en todos los casos es necesario introducir condiciones de balanceo.
- **ABB sin balanceo:** mal eficiencia en peor caso.
- **Balanceo perfecto:** costoso mantenerlo.
- **AVL:** Todos los casos están en **O(log n)** y el balanceo es poco costoso.

3.4. Árboles B.

- Los **árboles B** son muy usados en Bases de Datos.
- **Necesidades propias de las aplicaciones de BD:**
  - Muchos datos, básicamente conjuntos y diccionarios.
  - El acceso secuencial y directo debe ser rápido.
  - Datos almacenados en memoria secundaria (disco) en bloques.
- Existen muchas variantes: árboles B, B+ y B*.
- **Idea:** Generalizar el concepto de árbol binario de búsqueda a **árboles de búsqueda n-arios**.

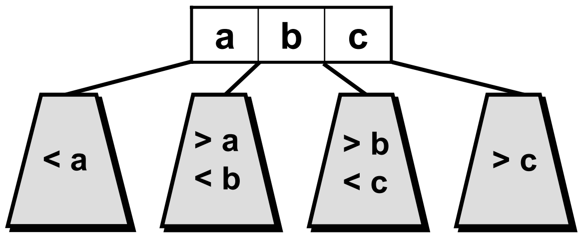
3.4. Árboles B.

Árbol Binario de Búsqueda



Árbol de Búsqueda N-ario

- En cada nodo hay **n** claves y **n+1** punteros a nodos hijos.



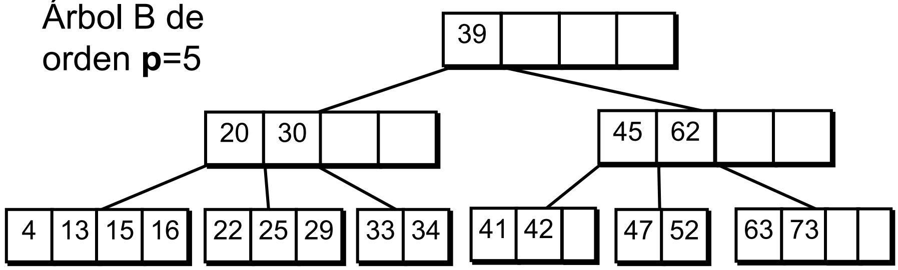
3.4. Árboles B.

- **Definición: Un árbol B de orden p** es un árbol n-ario de búsqueda, que cumple las siguientes propiedades:
  - **Raíz del árbol:** o bien no tiene hijos o tiene como mínimo tiene **2** y como máximo **p**.
  - **Nodos internos:** tienen entre $\lceil \mathbf{p/2} \rceil$ y **p** hijos.
  - **Nodos hoja:** todas las hojas deben aparecer al mismo nivel en el árbol (condición de balanceo).

- **Idea intuitiva:** Cada nodo tiene **p** posiciones (**p** punteros y **p-1** claves) que deben "llenarse" como mínimo hasta la mitad de su capacidad.

3.4. Árboles B.

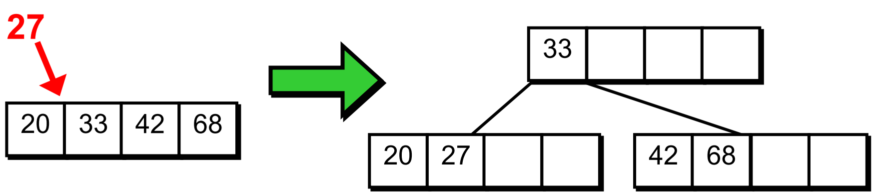
Árbol B de orden **p**=5



- **Búsqueda**: igual que en los árboles binarios, eligiendo la rama por la que seguir.
- La altura del árbol es ~ $\mathbf{log_{p/2}\, n}$, en el peor caso.

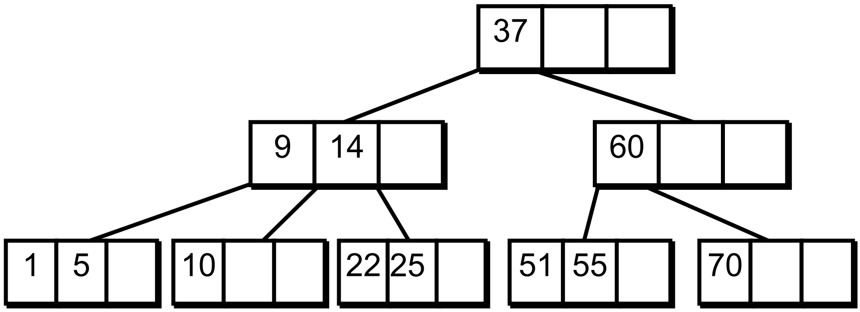
3.4. Árboles B.

- **Inserción de entradas en un árbol B:** Buscar el nodo hoja donde se debería colocar la entrada.
  - **Si quedan sitios libres** en esa hoja, insertarlo (en el orden adecuado).
  - **Si no quedan sitios** (la hoja tiene **p-1** valores) partir la hoja en **2** hojas (con $\lceil \mathbf{(p-1)/2} \rceil$ y $\lfloor \mathbf{(p-1)/2} \rfloor$ nodos cada una) y añadir la mediana al nodo padre.
    - Si en el padre no caben más elementos, repetir recursivamente la partición de las hojas.



3.4. Árboles B.

- **Ejemplo:** En un árbol B de orden **p**=4, insertar las claves: 37, 14, 60, 9, 22, 51, 10, 5, 55, 70, 1, 25.



- ¿Cuál es el resultado en un árbol B de orden **p**=5?

3.4. Árboles B.

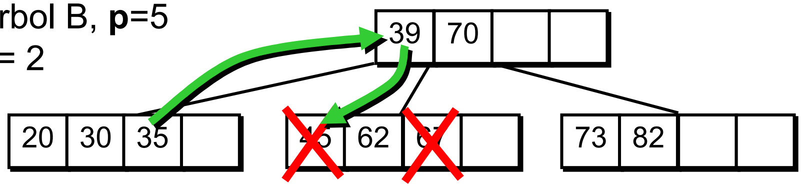
- **Eliminación de entradas en un árbol B:** Buscar la clave en el árbol.
  - **Nodo interno (no hoja):** Sustituirla por la siguiente (o la anterior) en el orden. Es decir, por la mayor de la rama izquierda, o la menor de la rama derecha.
  - **Nodo hoja:** Eliminar la entrada de la hoja.

- **Casos de eliminación en nodo hoja. d = $\lfloor(p-1)/2\rfloor$**
  - Nodo con más de **d** entradas: suprimir la entrada.
  - Nodo con **d** entradas (el mínimo posible): reequilibrar el árbol.

3.4. Árboles B.

- **Eliminación en nodo con d entradas**:
  - **Nodo hermano con más de d entradas:** Se produce un proceso de **préstamo** de entradas: Se suprime la entrada, la entrada del padre pasa a la hoja de supresión y la vecina cede una entrada al nodo padre.

Árbol B, **p**=5
d= 2

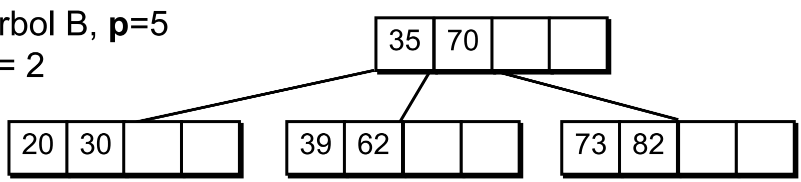


- **Ejemplo.** Eliminar 67, 45.

3.4. Árboles B.

- **Eliminación en nodo con d entradas**:
  - **Nodo hermano con más de d entradas:** Se produce un proceso de **préstamo** de entradas: Se suprime la entrada, la entrada del padre pasa a la hoja de supresión y la vecina cede una entrada al nodo padre.

Árbol B, **p**=5
d= 2

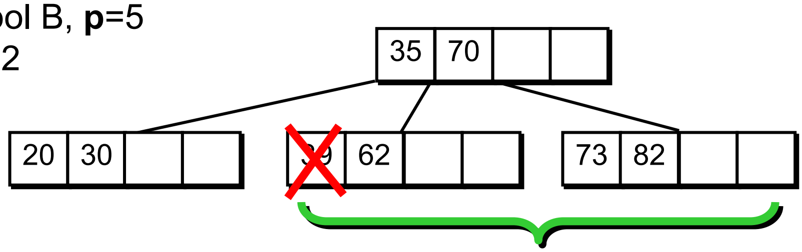


- **Ejemplo.** Eliminar 67, 45.

3.4. Árboles B.

– **Ningún hermano con más de d entradas:** Con la hoja donde se hace la supresión (**d-1** entradas) más una hoja hermana (**d** entradas) más la entrada del padre, se crea una nueva hoja con **2d** entradas.

Árbol B, **p**=5
d= 2

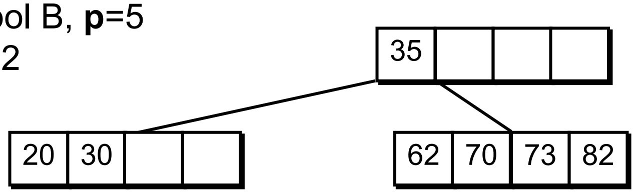


- **Ejemplo.** Eliminar 39.

3.4. Árboles B.

– **Ningún hermano con más de d entradas:** Con la hoja donde se hace la supresión (**d-1** entradas) más una hoja hermana (**d** entradas) más la entrada del padre, se crea una nueva hoja con **2d** entradas.

Árbol B, **p**=5
d= 2



- **Ejemplo.** Eliminar 39.
- **Ojo:** Se suprime una entrada en el padre. Se debe repetir el proceso de eliminación en el nivel superior.

3.4. Árboles B.

Conclusiones

- El orden de complejidad es proporcional a la altura del árbol, ~ $\mathbf{log_{p/2}\, n}$ en el peor caso.
- Normalmente, el orden **p** del árbol se ajusta para hacer que cada nodo esté en un bloque de disco, minimizando el número de operaciones de E/S.
- **Representación en memoria:** mejor usar AVL.
- **Representación en disco:** mejor usar árboles B.

3. Repr. de conjuntos mediante árboles.

Conclusiones generales

- Representaciones **arbóreas frente a** representaciones **lineales** (listas y arrays).
- Necesidad de incluir **condiciones de balanceo** para garantizar eficiencia en todos los casos.
- **Distinción** entre TAD y estructura de datos:
  - TAD árbol, binario, n-ario, etc.
  - Usamos estructuras de árboles para representar el TAD conjunto y diccionario.
  - Para el usuario lo importante es la interface (las operaciones accesibles) independientemente de la representación interna.