Fundamentos de Programación II

Tema 4. Estructuras no lineales de datos: árboles

Luís Rodríguez Baena (luis.rodriguez@upsam.net)

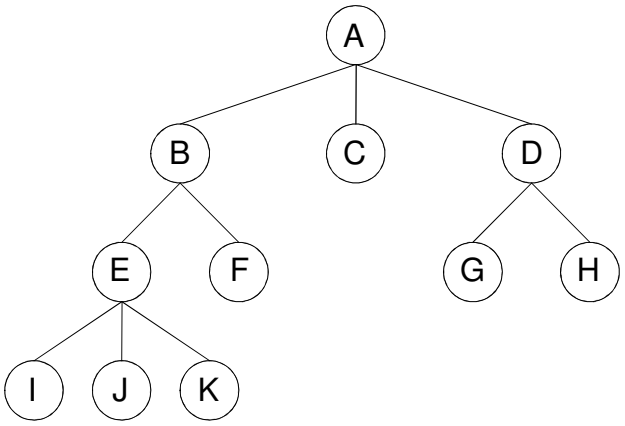
Universidad Pontificia de Salamanca (campus Madrid)
Escuela Superior de Ingeniería y Arquitectura

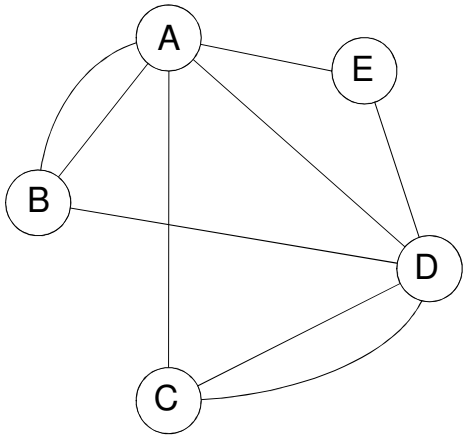
Estructuras de datos no lineales

- En una estructura lineal, cada elemento sólo puede ir enlazado al siguiente o al anterior.
- A las estructuras de datos no lineales se les llama también **estructuras de datos multienlazadas**.
  - Cada elemento puede estar enlazado a cualquier otro componentes.
- Se trata de estructuras de datos en las que cada elemento puede tener varios sucesores y/o varios predecesores.
  - Árboles.
  - Grafos.

Estructuras de datos no lineales (II)

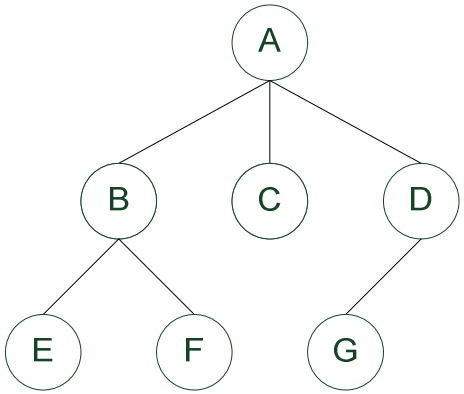
- Árboles.
  - Cada elemento sólo puede estar enlazado con su predecesor y sus sucesores.
    - ✓ Puede tener varios sucesores.
- Grafos.
  - Cada elemento puede estar enlazado a cualquier otro.

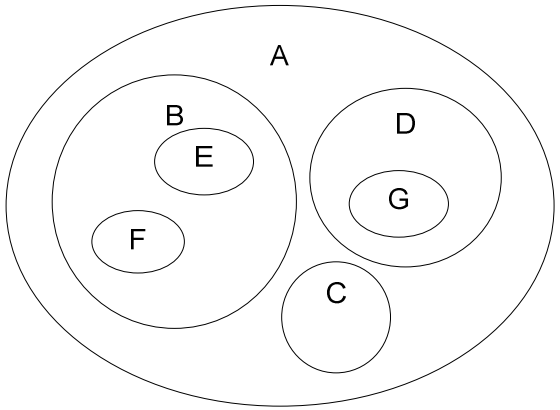




Árboles

- Estructura no lineal jerárquica en la que cada elemento tiene un único antecesor y puede tener varios sucesores.
  - Existe un único camino entre el primer nodo de la estructura y cualquier otro nodo.
- Se utilizan para representar todo tipo de jerarquías: árbol genealógico, taxonomías, diagramas de organización, etc.
- En informática se utilizan para aplicaciones algorítmicas (ordenación, búsqueda), compilación (árboles sintácticos, árboles de expresiones), etc.
- Formalmente, un árbol A es un conjunto finito de elementos con 0 o más nodos de forma que:
  - Se trata de una estructura vacía.
  - Si tiene componentes, los nodos restantes se dividen en uno o más conjuntos disjuntos cada uno de los cuales es a su vez un árbol. A estos nodos se les llama subárboles del raíz.
  - Se trata de una estructura recursiva.



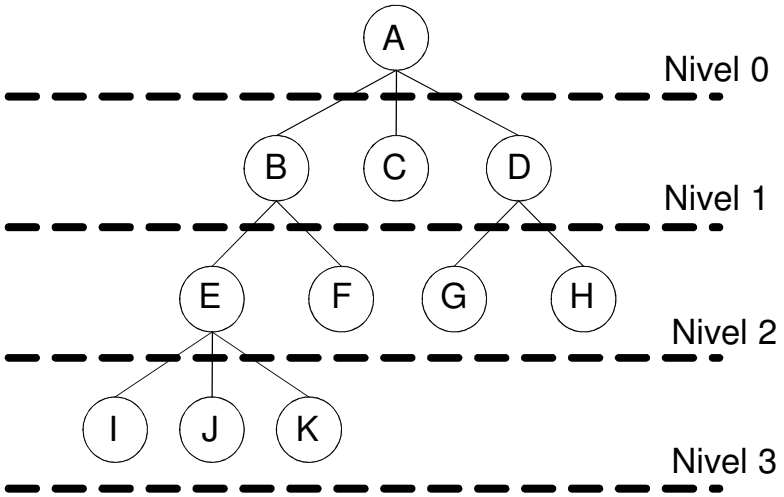


Terminología

- **Nodo**: los vértices o elementos de un árbol.
- **Enlace/arco/arista**: Conexión entre dos nodos consecutivos.
- Los nodos pueden ser:
  - **Nodo raíz**: nodo superior de la jerarquía.
  - **Nodo terminal u hoja**: nodo que no contienen ningún subárbol.
  - **Nodos interiores**: nodos con uno o más subárboles; nodos que no son hojas.
  - **Descendientes o hijos**: cada uno de los subárboles de un nodo.
  - **Ascendiente, antecesor o padre**: nodo de jerarquía superior a uno dado.
  - **Nodos hermanos**: nodos del mismo padre.
- **Bosque**: colección de árboles.

Terminología (II)

- **Camino**: enlace entre dos nodos.
  - No existe un camino entre todos los nodos.
- **Rama**: camino que termina en una hoja.
- **Grado de un nodo**: número de subárboles que tiene.
- **Nivel de un nodo o longitud del camino**: número de arcos o enlaces que hay desde el nodo raíz hasta un nodo dado.
- **Altura o profundidad de un árbol**: número máximo de nodos de una rama; el nivel más alto de un árbol más uno.
- **Peso de un árbol**: número de nodos terminales.

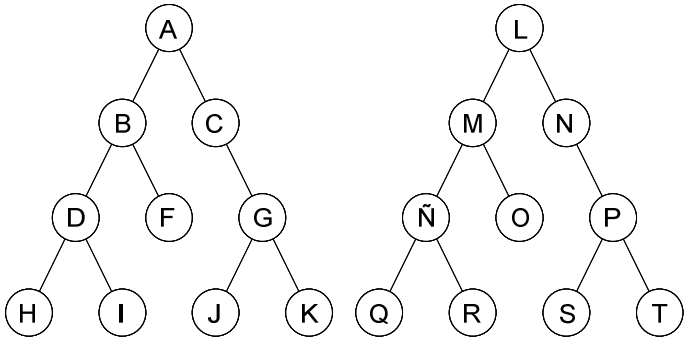


Altura del árbol = 4
Peso del árbol = 7

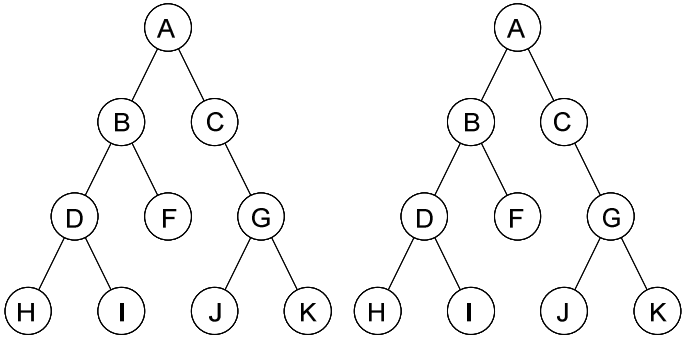
Árboles binarios

- Un árbol general sería un árbol en el que cada nodo puede tener un número ilimitado de subárboles.
- Un árbol binario sería un conjunto de 0 o más nodos en el cual existe un nodo raíz y cada uno de los nodos, incluido el raíz podrán tener 0, 1 o dos subárboles:
  - Subárbol izquierdo y subárbol derecho.
  - Cada nodo es como máximo de grado 2.
- Terminología:
  - **Árboles similares**: árboles con la misma estructura.
  - **Árboles equivalentes**: árboles con la misma estructura y contienen la misma información.
  - **Árboles completos o árboles perfectos**: todos los nodos, excepto las hojas, tienen grado 2.
    - ✓ Un árbol binario de nivel n tiene 2^n-1 nodos.
  - **Árbol equilibrado**: un árbol en el que las alturas de los dos subárboles de cada uno de los nodos tiene como máximo una diferencia de una unidad.
  - **Árbol degenerado**: todos sus nodos sólo tienen un subárbol.

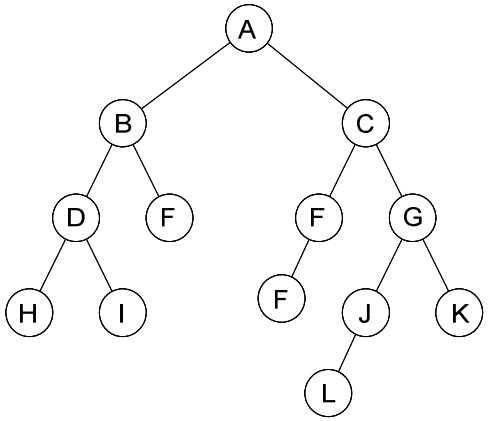
Árboles binarios (II)



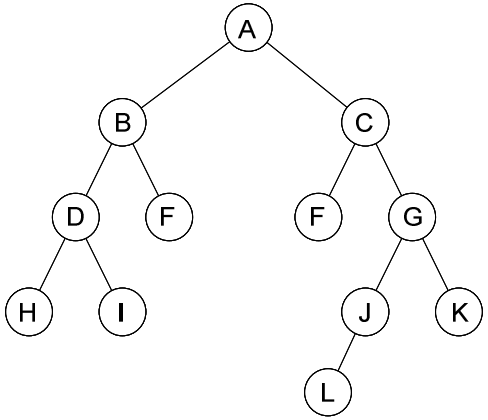
Árboles similares



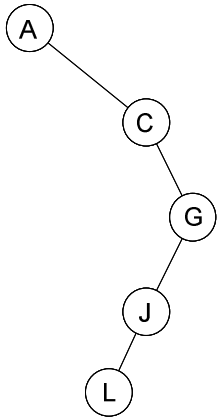
Árboles equivalentes



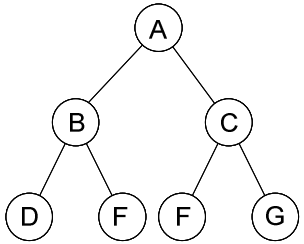
Árbol equilibrado



Árbol no equilibrado



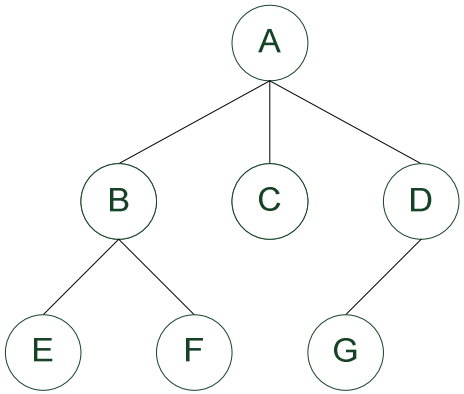
Árbol degenerado



Árbol completo

Implementaciones de árboles generales

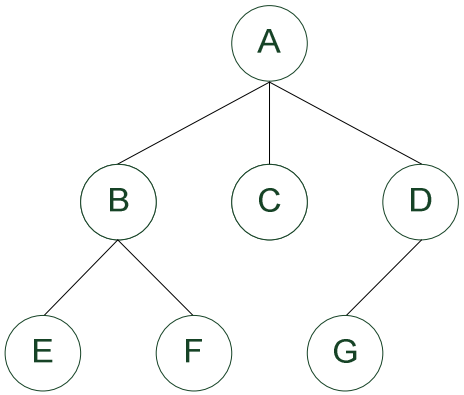
- Cada nodo debería tener un número indeterminado de enlaces que apunten a cada uno de sus hijos.
- Problema ¿cuántos enlaces reservamos?
- Se pueden implementar como un array con indicaciones al padre de cada nodo.

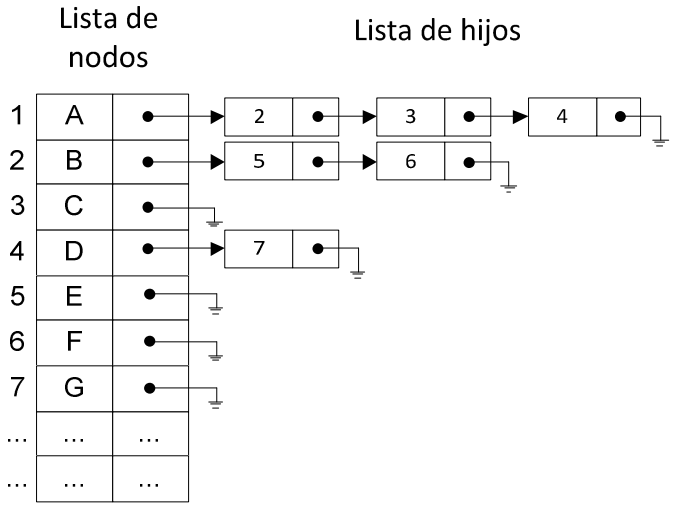


	1	2	3	4	5	6	7	
Nodos	A	B	C	D	E	F	G	...
Padres	0	1	1	1	2	2	4	...

Implementaciones de árboles generales (II)

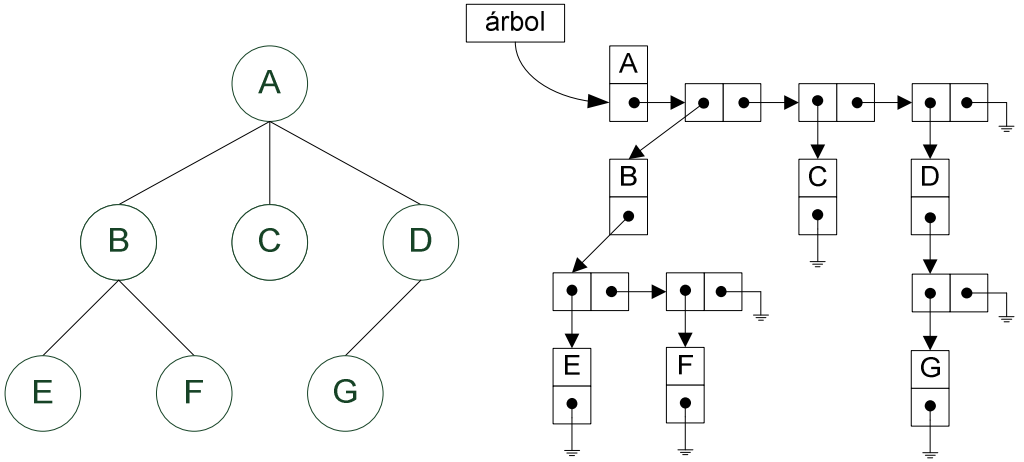
- Implementación mediante una lista de hijos.
  - En un índice aparecen todos los nodos.
  - Por cada nodo existe una lista de hijos.





Implementaciones de árboles generales (III)

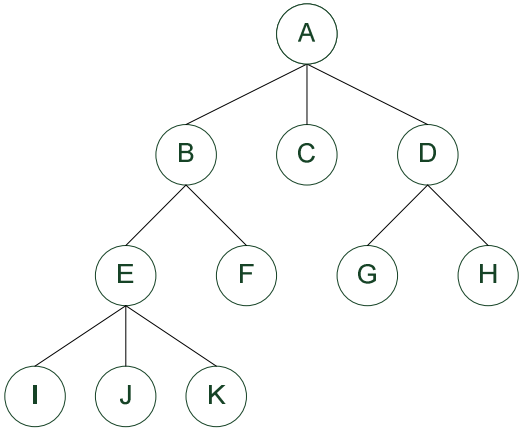
- Implementación mediante una lista listas.
  - Cada nodo apunta a una lista enlazada.
  - En esa lista, cada nodo apunta a un nodo hijo.

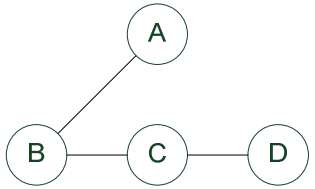


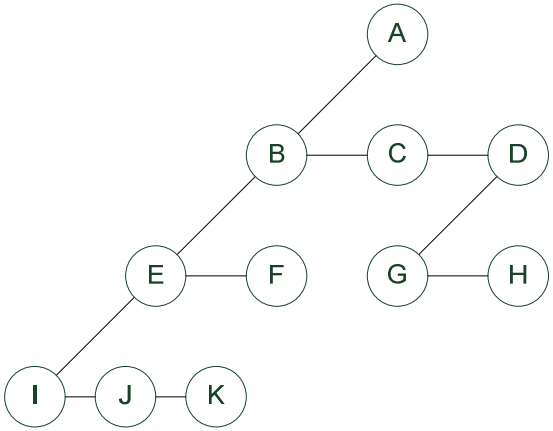
Conversión de árboles generales en binarios

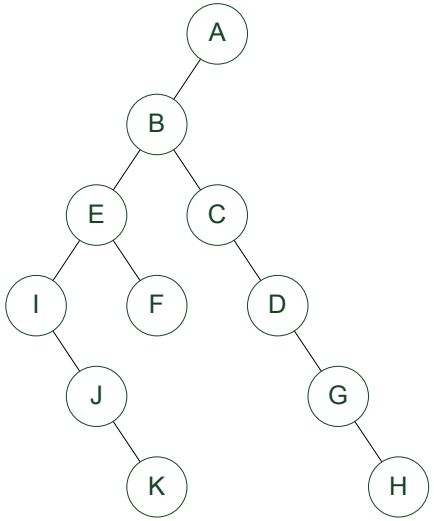
- Los árboles generales son más difíciles de implementar que los árboles binarios.
- En este curso se implementarán árboles binarios.
- De cualquier forma, un árbol general se puede convertir a binario.
  - La raíz del árbol binario será la raíz del árbol general.
  - Se enlaza el hijo situado más a la izquierda del nodo raíz del árbol general como hijo izquierdo del nodo raíz en el árbol binario.
  - Se enlazan todos los hermanos como hijos derechos en el árbol binario.
  - El proceso se repite con cada nodo.

Conversión de árboles generales en binarios (II)



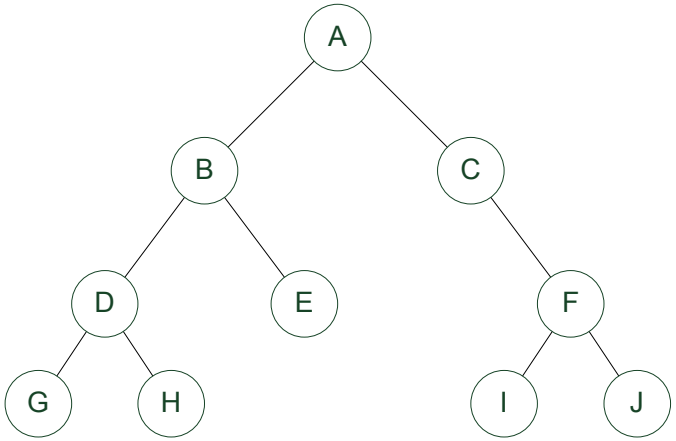






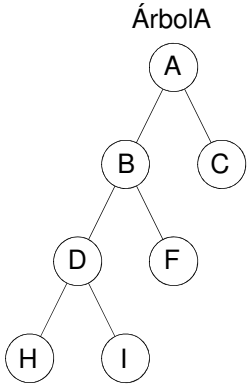
Implementación de árboles binarios

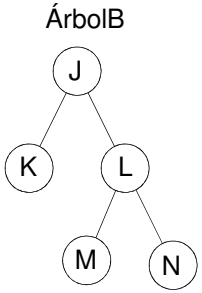
- Se pueden implementar como un array de elementos del tipo base del árbol.
  - El primer elemento se corresponde al nivel 0 del árbol.
  - Los dos siguientes elemento se corresponden al nivel 1 del árbol.
  - Los cuatro siguientes elementos se corresponden al nivel 2 del árbol.
  - Los ocho siguientes elementos se corresponden al nivel 3 del árbol.
  - Los dieciséis siguientes elementos se corresponden al nivel 4 del árbol.
  - …



Implementación de árboles binarios (II)

- Se puede implementar como un array de nodos.





	hizq	raíz	hder
1	15	D	11
2			-1
3	7	A	9
4			-1
5	0	F	0
6	0	M	0
7	1	B	5
8	14	J	12
9	0	C	0
10	0	N	0
11	0	I	0
12	6	L	10
13			-1
14	0	K	0
15	0	H	0

ÁrbolA
3

ÁrbolB
8

Implementación de árboles binarios (III)

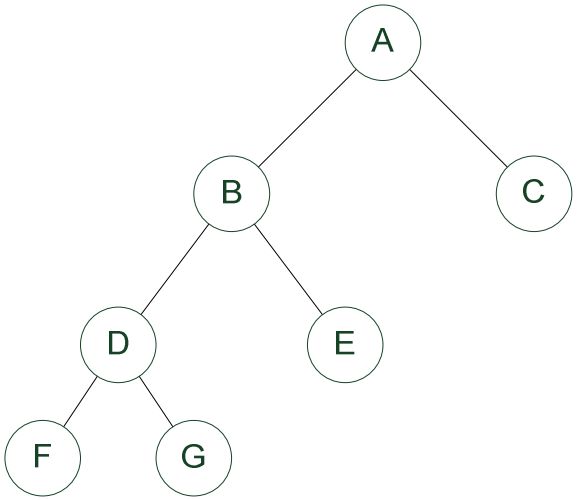
- El almacenamiento será un array de nodos.
- El tipo `árbol` será un índice de array.
- Cada nodo tendrá los campos `raíz` de tipo `TipoElemento`, `hizq` de tipo `árbol` y `hder` de tipo `árbol`.
  - Los punteros `hizq` e `hder` serían índices del array.
  - Un valor -1 en el hijo derecho podría corresponder a un elemento que no contiene un nodo válido.

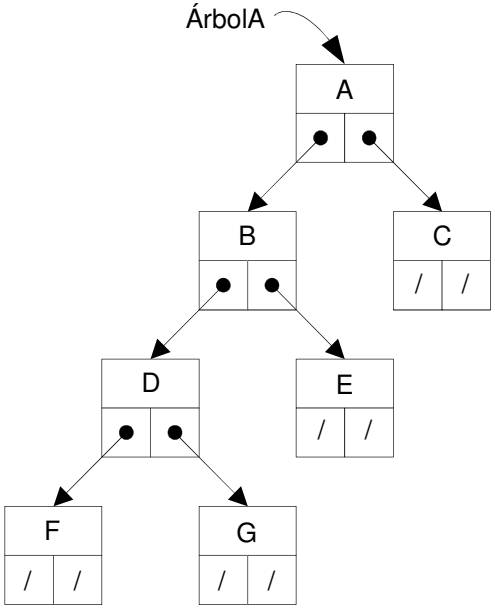
```
const
   MaxArbol = …
tipos
   … = TipoElemento
   entero = árbol
   registro = nodo
      TipoElemento = raíz
      árbol = hizq, hder
   fin_registro
   array[1..MaxArbol] de nodo = almacenamiento
var
   almacenamiento : alm
   árbol : a
```

Implementación de árboles binarios (IV)

- Mediante estructuras dinámicas.
  - El árbol estaría compuesto por una serie de nodos.
  - Cada nodo contendría:
    - ✓ El campo raíz con la información del tipo base del árbol.
    - ✓ Los punteros `hizq` e `hder` que serían punteros a nodo, es decir, serían también árboles.
    - ✓ Si alguno se esos hijos es un árbol vacío contendrían un valor nulo.
  - El tipo de dato árbol sería un puntero a nodo.
    - ✓ Un puntero a nodo indicaría el nodo de inicio de la estructura, es decir, la raíz del árbol.

Implementación de árboles binarios (V)






```
tipos
   … = TipoElemento
   puntero_a nodo = árbol
   registro = nodo
      TipoElemento = raíz
      árbol = hizq, hder
   fin_registro
var
   árbol : a
```

Recorridos en árboles binarios

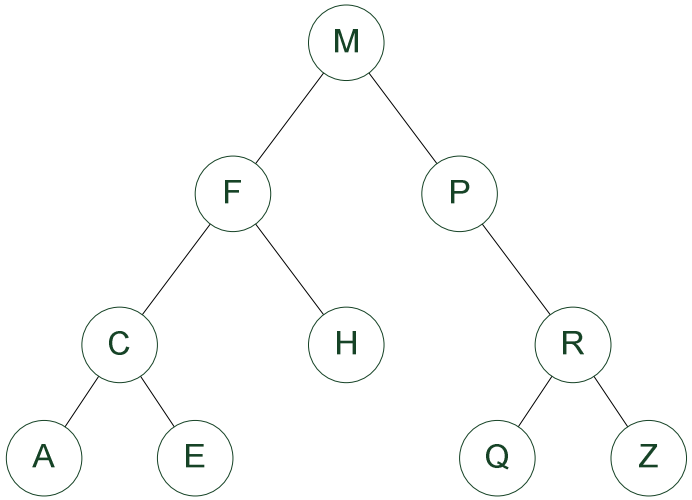
- En una estructura de datos lineal sólo es posible un tipo de recorrido en dos sentidos distintos:
  - Del primero al último o del último al primero.
- Una estructura de datos no lineal se puede recorrer de distintas maneras:
  - ¿Se realiza un recorrido por niveles, accediendo a los nodos hermanos de cada nivel?
  - Una vez situados en un nodo, ¿a qué hijo se accede primero?
  - En que momento se accede a la información del nodo, ¿antes o después de los hijos?
- En general existen dos grandes grupos de recorridos:
  - Recorridos en profundidad.
    - ✓ Se recorren las ramas de un nodo dado.
  - Recorridos en anchura.
    - ✓ Se accede a los nodos hermanos en cada uno de los niveles del árbol.

Recorridos en árboles binarios
Recorridos en profundidad

- Tres tipos de recorridos dependiendo del orden en que se acceda al subárbol izquierdo, al subárbol derecho o al nodo raíz.
  - En los tres recorridos se accede antes al hijo izquierdo que al hijo derecho.
  - La variación reside en el momento en que se recorrerá la información del nodo.
    - ✓ Recorrido preorden u orden previo (RID, raíz-hijo izquierdo-hijo derecho).
    - ✓ Recorrido inorden u orden simétrico (IRD, hijo izquierdo-raíz-hijo derecho).
    - ✓ Recorrido postorden u orden posterior (IDR, hijo izquierdo- hijo derecho-raíz).

Recorridos en árboles binarios
Recorridos en profundidad (II)

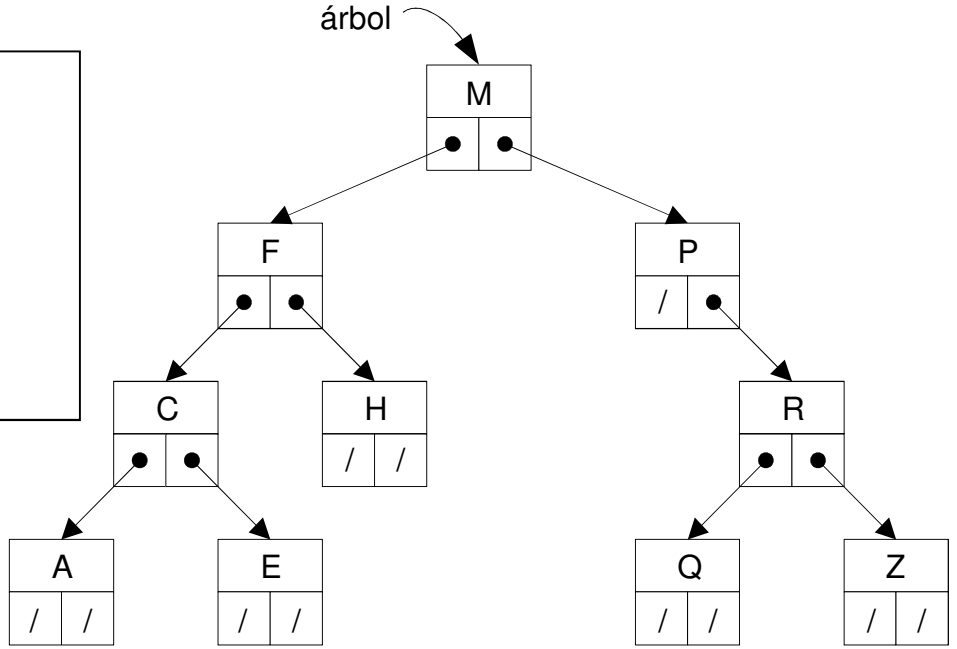
- Recorrido preorden.
  - Por cada elemento no vacío:
    - ✓ Se accede primero al nodo raíz.
    - ✓ Se accede al hijo izquierdo en preorden.
    - ✓ Se accede al hijo derecho en preorden.
    - ✓ Para el siguiente árbol:
      - M,F,C,A,E,H,P,R,Q,Z
- Recorrido inorden.
  - Por cada elemento no vacío:
    - ✓ Se accede al hijo izquierdo en inorden.
    - ✓ Se accede primero al nodo raíz.
    - ✓ Se accede al hijo derecho en inorden.
    - ✓ Para el siguiente árbol:
      - A,C,E,F,H,M,P,Q,R,Z
- Recorrido postorden.
  - Por cada elemento no vacío:
    - ✓ Se accede al hijo izquierdo en postorden.
    - ✓ Se accede al hijo derecho en postorden.
    - ✓ Se accede primero al nodo raíz.
    - ✓ Para el siguiente árbol:
      - A,E,C,H,F,Q,Z,R,P,M



Recorridos en árboles binarios
Recorrido preorden

- Los tres recorridos tienen una definición recursiva.
  - En la definición del recorrido, interviene el objeto definido.
- Se procesa el nodo raíz y se recorre el subárbol izquierdo en preorden, y después el subárbol derecho también en preorden.

```
procedimiento PreOrden(valor arbol: a)
inicio
   si a <> nulo entonces
      //Procesar elemento raíz
      escribir(a↑.raíz)
      PreOrden(a↑.hizq)
      PreOrden(a↑.hder)
   fin_si
fin_procedimiento
```

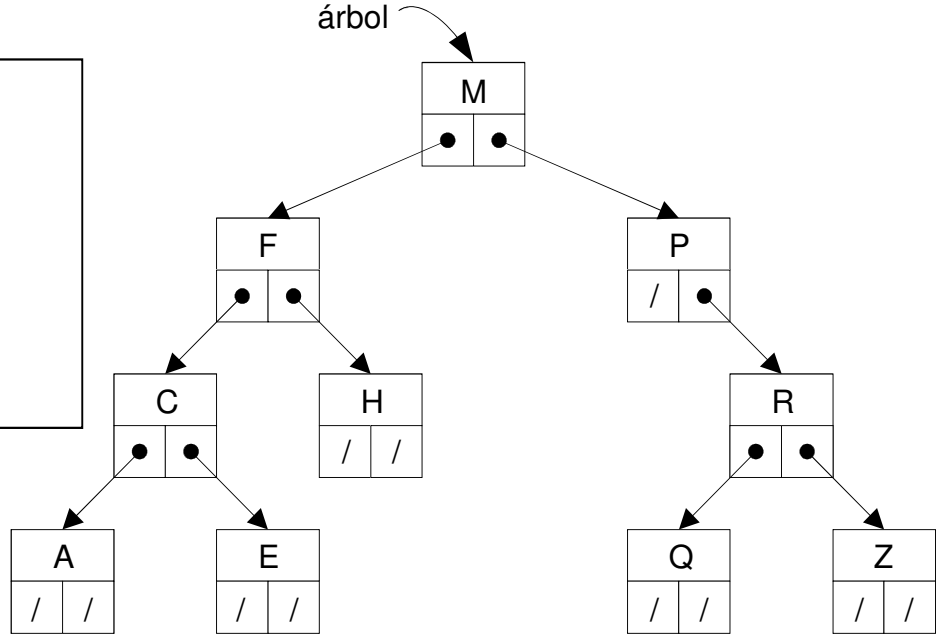


Recorrido preorden: M F C A E H P R Q Z

Recorridos en árboles binarios
Recorrido inorden

- Se recorre el subárbol izquierdo en inorden, se procesa el raíz y después el subárbol derecho también en inorden.

```
procedimiento InOrden(valor arbol: a)
inicio
   si a <> nulo entonces
      InOrden(a↑.hizq)
      //Procesar elemento raíz
      escribir(a↑.raíz)
      InOrden(a↑.hder)
   fin_si
fin_procedimiento
```

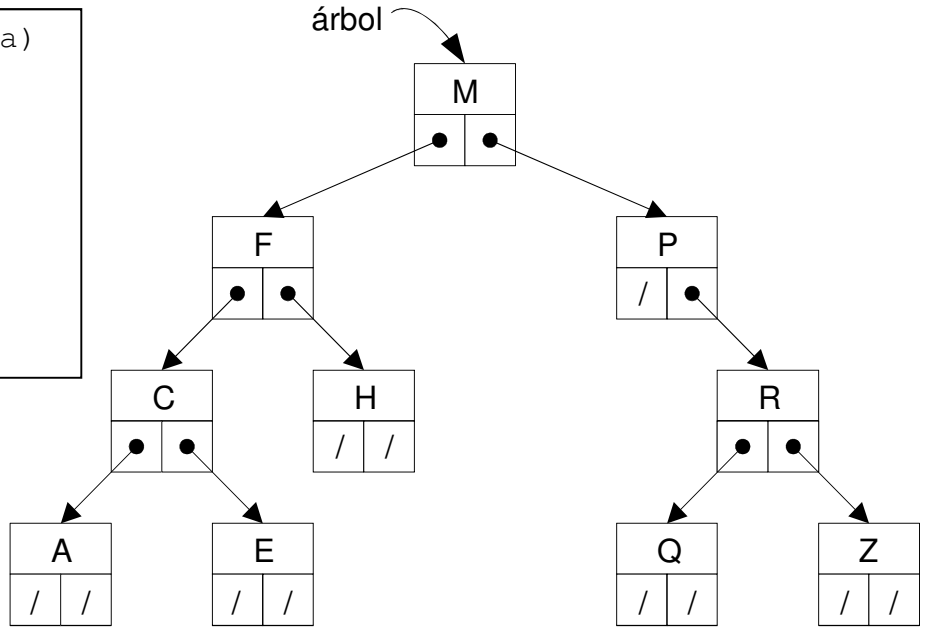


Recorrido inorden: A C E F H M P Q R Z

Recorridos en árboles binarios
Recorrido postorden

- Se procesa se recorre el subárbol izquierdo en postorden, después el subárbol derecho también en postorden y por último se procesa el nodo ráiz.

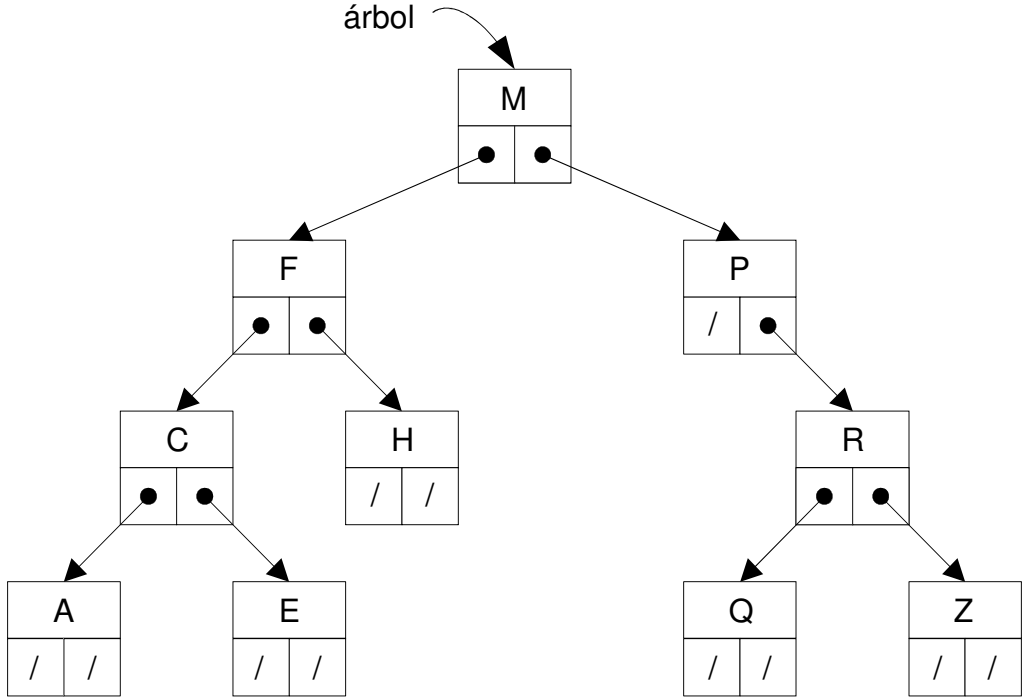
```
procedimiento PostOrden(valor arbol: a)
inicio
   si a <> nulo entonces
      PostOrden(a↑.hizq)
      PostOrden(a↑.hder)
      //Procesar elemento raíz
      escribir(a↑.raíz)
  fin_si
fin_procedimiento
```



Recorrido postorden: A E C H F Q Z R P M

Recorrido en anchura

- El recorrido se hace por niveles a partir del nivel 0.
  - Por cada nivel se recorren los nodos hermanos de izquierda a derecha.



Recorrido en anchura: M F P C H R A E Q Z

Recorrido en anchura (II)

- Para su implementación hay que procesar el nodo raíz y recordar las direcciones de sus hijos izquierdo y derecho para acceder a ellos en el mismo orden en que se encuentran.
- Hay que repetir el proceso por cada nodo, recordando las direcciones de sus hijos izquierdo y derecho y recuperarlas en el mismo orden en que han entrado.
- Existe una estructura de datos que extrae los elementos en el mismo orden en que entran: la cola.
  - Se utiliza una cola que almacene árboles (punteros a nodos).
  - Se introduce la dirección del nodo raíz en la cola.
  - Se extraen elementos de la cola y por cada elemento que se extrae que no sea un árbol vacío (un puntero nulo), se introducen las direcciones de sus hijos izquierdo y derecho.
  - El proceso termina cuando la cola esté vacía.

Recorrido en anchura (II)

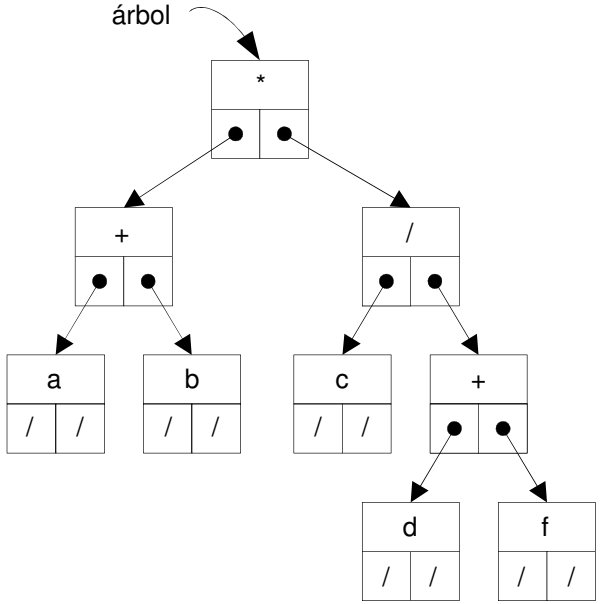
```
procedimiento Anchura(valor árbol : a)
tipos
   TipoElementoCola : árbol //La cola contiene punteros a nodo
   …
var
   cola : niveles
   árbol : aux
inicio
   ColaNueva(niveles)
   CInsertar(niveles,a)
   repetir
      Primero(niveles, aux)
      CBorrar(niveles)
      si aux <> nulo entonces
          //Procesar ráiz
          escribir(a↑.raíz)
          CInsertar(niveles, a↑.hizq)
          CInsertar(niveles, a↑.hder)
      fin_si
   hasta_que EsColaVacía(niveles)
fin_procedimiento
```

Aplicaciones de árboles

- En una lista enlazada sólo hay una forma de insertar o de eliminar un elemento:
  - Para insertar un elemento hay que colocarlo entre al anterior y el siguiente.
  - Para eliminar un elemento, hay que hacer que el anterior apunte al siguiente.
- En un árbol, dependiendo de la aplicación que se haga del árbol esas operaciones pueden cambiar:
  - Si se inserta un elemento ¿se inserta como hijo izquierdo o como hijo derecho? ¿qué se hace con los hijos del nodo anterior?
  - Si se elimina un elemento, ¿qué se hace con los hijos de ese nodo?
- Algunas aplicaciones de árboles binarios:
  - Árbol de expresiones.
  - Árbol binario de búsqueda.

Árboles de expresiones

- Representación de una expresión.
- Los nodos internos son los operadores y las hojas los operandos.
- Si hay varias operaciones, las de menos prioridad ocupan los niveles más altos.
- Sea la expresión: (a+b) * (c/(d+f))…



Expresión infija (recorrido inorden):
- a+b*c/d+f

Expresión prefija (recorrido preorden):
- *+ab/c+df

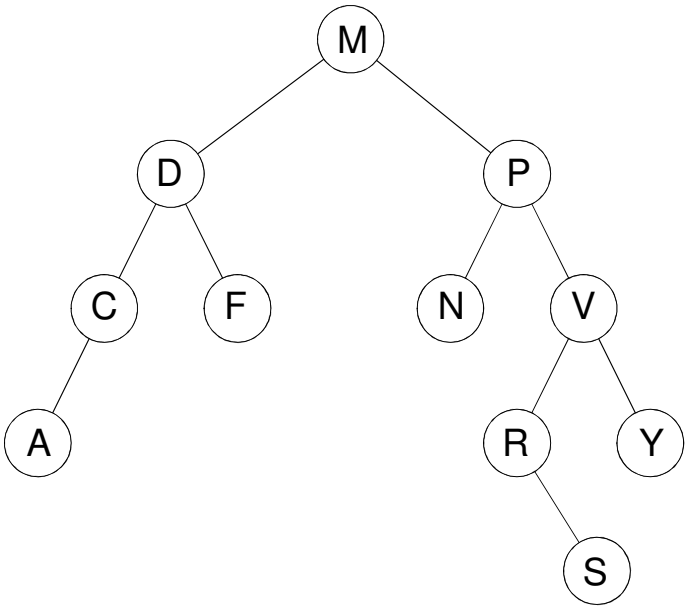
Expresión postfija (recorrido postorden):
- ab+cdf+/*

Árbol binario de búsqueda

- Una de las aplicaciones más importantes de árboles se basa en su capacidad para ordenar y buscar elementos.
- Los árboles binarios de búsqueda se utilizan para esas aplicaciones.
- Se trata de un árbol en el que se colocan los elementos de una lista según un determinado criterio:
  - El primer elemento de la lista es el raíz.
  - Los siguientes elementos se colocan de forma que los elementos menores queden en el subárbol izquierdo y los mayores en el derecho.

Árbol binario de búsqueda (II)

- Dada la siguiente lista: M D P V C A F R Y N S, el árbol de búsqueda resultante sería:



- Obsérvese que:
  - El recorrido inorden del árbol devolvería los elementos en orden ascendente.
  - Para buscar un elemento, habría que comparar el elemento con el raíz y, dependiendo de que el elemento sea mayor o menor, ir al subárbol derecho o izquierdo respectivamente.

Búsqueda en árboles binarios de búsqueda

- Para buscar un elemento hay que compararlo con el que ocupa la raíz del árbol.
  - Si es igual, ya se ha encontrado.
  - Si es menor, se busca en el subárbol izquierdo.
  - Si es mayor, se busca en el subárbol derecho.
- Admite una definición recursiva:
  - Casos base:
    - ✓ Si el árbol es nulo, el elemento no está.
    - ✓ Si se trata del elemento raíz, el elemento ya se ha encontrado.
  - Casos generales.
    - ✓ Si el elemento es menor que el raíz hay que hacer la búsqueda en el subárbol izquierdo (llamada recursiva).
    - ✓ Si el elemento es mayor, hay que hacer la búsqueda en el subárbol derecho (llamada recursiva).
- La función de búsqueda devolverá un árbol, es decir un puntero a nodo.
  - Si el elemento está en el árbol, será la dirección del nodo dónde se encuentra el elemento buscado.
  - Si el elemento no está en el árbol, devolverá un puntero nulo.

Búsqueda iterativa

```
árbol : función Buscar(valor árbol : a; valor TipoElemento : e)
var
   lógico : encontrado
inicio
   encontrado ← falso
   mientras no encontrado y (a <> nulo) hacer
       si a↑.raíz = e entonces
           encontrado ← verdad
       si_no
           si a↑.raíz > e entonces
              a ← a↑.hizq
           si_no
              a ← a↑.hder
           fin_si
       fin_si
   fin_mientras
   si encontrado entonces
      devolver(a)
   si_no
      devolver(nulo)
   fin_si
fin_función
```

Búsqueda recursiva

- Funciona de forma similar a la búsqueda binaria recursiva.
  - Si el árbol el nulo, no se encuentra y devuelve nulo.
  - Si el elemento raíz coincide, devuelve el valor de a.
  - Si el elemento raíz es mayor, se busca en el subárbol izquierdo.
  - Si el elemento raíz es menor, se busca en el subárbol derecho.

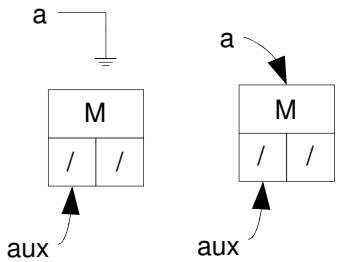
```
árbol : función Buscar(valor árbol : a; valor TipoElemento : e)
inicio
   si a = nulo entonces
      devolver(nulo)
   si_no
      si a↑.raíz = e entonces
         devolver(a)
      si_no
         si a↑.raíz > e entonces
            Buscar(a↑.hizq,e)
         si_no
            Buscar(a↑.hder,e)
         fin_si
     fin_si
   fin_si
fin_función
```

Inserción en un árbol de búsqueda

- En un árbol binario de búsqueda, todos los elementos se insertan como hojas.
- Para insertar un elemento, hay que desplazarse por los súbárboles izquierdo o derecho, dependiendo del valor del nodo, hasta encontrar un árbol nulo.
- Según el valor del último elemento raíz, insertar como raíz del árbol, como hijo derecho del anterior o como hijo izquierdo del anterior.

Inserción en un árbol de búsqueda (II)

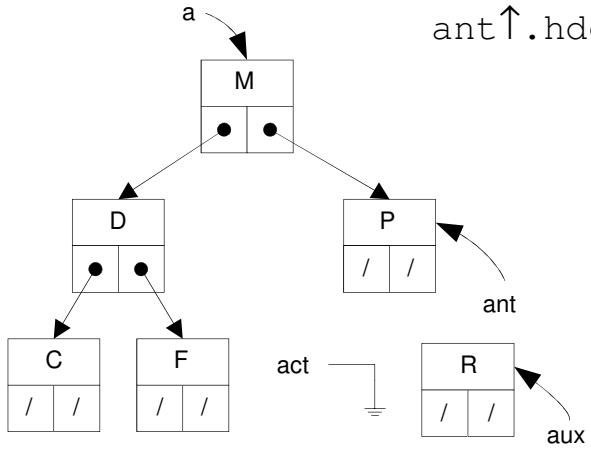
Caso 1: Insertar en un árbol vacío: Insertar(a,'M')



`a ← aux`

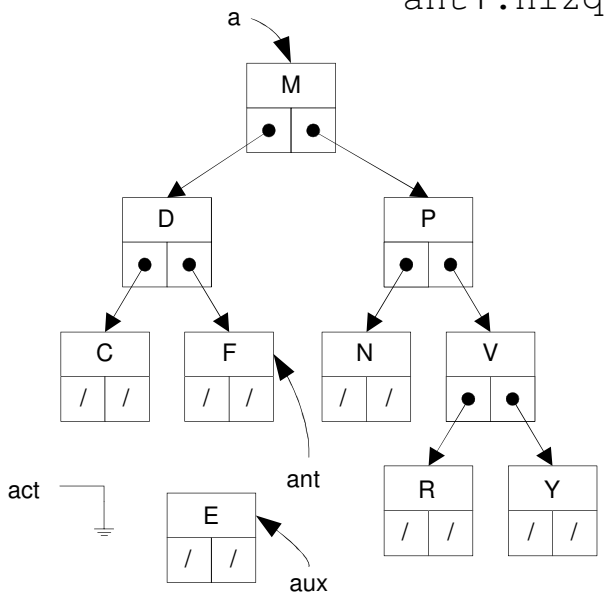
Caso 3: Insertar como hijo derecho del anterior: Insertar(a,'R')

`ant↑.hder ← aux`



Caso 2: Insertar como hijo izquierdo: del anterior Insertar(a,'E')

`ant↑.hizq ← aux`



Inserción en un árbol de búsqueda (III)

```
procedimiento Insertar(ref árbol :a ; valor TipoElemento : e)
var
   árbol : act, ant, aux
inicio
   //Inicializar punteros
   act ← a
   ant ← nulo
   //Reservar espacio para un nuevo nodo
   //Siempre se inserta el elemento
   reservar(aux)
   aux↑.raíz ← e
   aux↑.hizq ← nulo
   aux↑.hder ← nulo
   //Buscar la posición del nuevo elemento
   mientras act <> nulo hacer
      ant ← act
      si act↑.raíz > e then
         act ← act↑.hizq
      si_no
         act ← act↑.hder
      fin_si
   fin_mientras
   //Si es el primer nodo
   si ant = nulo entonces
      a ← aux
   si_no
      //Es hijo izq del anterior
      si ant↑.raíz > e entonces
         ant↑.hizq ← aux
      si_no
         ant↑.hder ← aux
      fin_si
   fin_si
fin_procedimiento
```

Inserción en un árbol de búsqueda
Versión recursiva

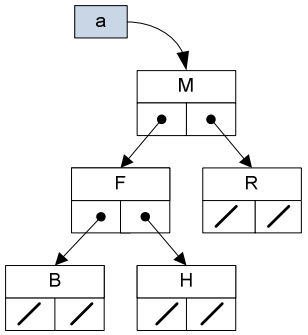
- Es posible hacer una definición recursiva de la inserción en un árbol binario de búsqueda.
  - Caso base.
    - ✓ Si el árbol es nulo.
      - Se inserta un nuevo nodo colgando del árbol.
  - Caso general.
    - ✓ Si el árbol no está vacío.
      - Si el elemento a insertar es menor que el elemento raíz, se inserta en el subárbol izquierdo.
      - Si el elemento a insertar es mayor que el elemento raíz, se inserta en el subárbol derecho.

Inserción en un árbol de búsqueda
Versión recursiva (II)

- Versión recursiva.

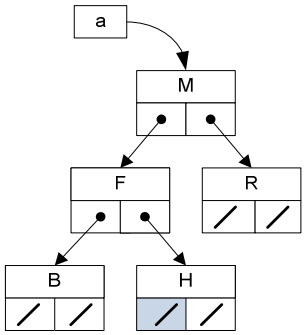
```
procedimiento Insertar(ref árbol :a ; valor TipoElemento : e)
inicio
   si a = nulo
      //Hemos llegado a una hoja. Insertamos
      reservar(a)
      a↑.raíz ← e
      a↑.hizq ← nulo
      a↑.hder ← nulo
   si_no
      si a↑.raíz > e then
         Insertar(a↑.hizq,e)
      si_no
         Insertar(a↑.hder,e)
      fin_si
   fin_si
fin_mientras
```

Inserción en un árbol de búsqueda
Versión recursiva (III)

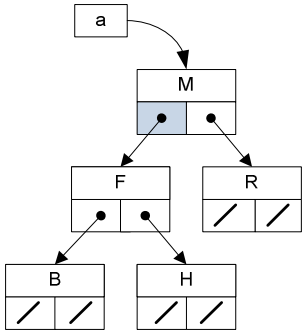



```
Insertar(a,'G')
```

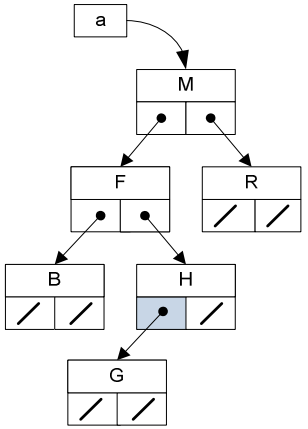
```
si a↑.raíz > e entonces
   Insertar(a↑.hizq,e)
```



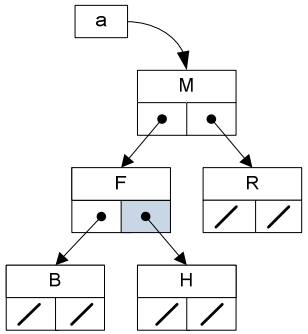

```
si a = nulo  entonces
```




```
si a↑.raíz < e entonces
   Insertar(a↑.hder,e)
```




```
reservar(a)
a↑.raíz ← e
a↑.hizq ← nulo
a↑.hder ← nulo
```

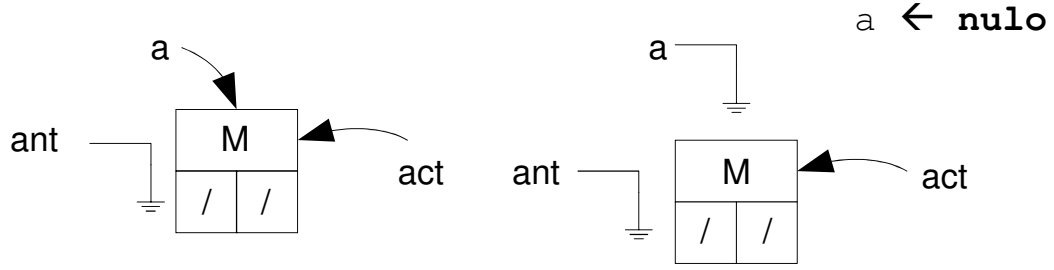



```
si a↑.raíz > e entonces
   Insertar(a↑.hizq,e)
```

Árbol que se pasa como argumento

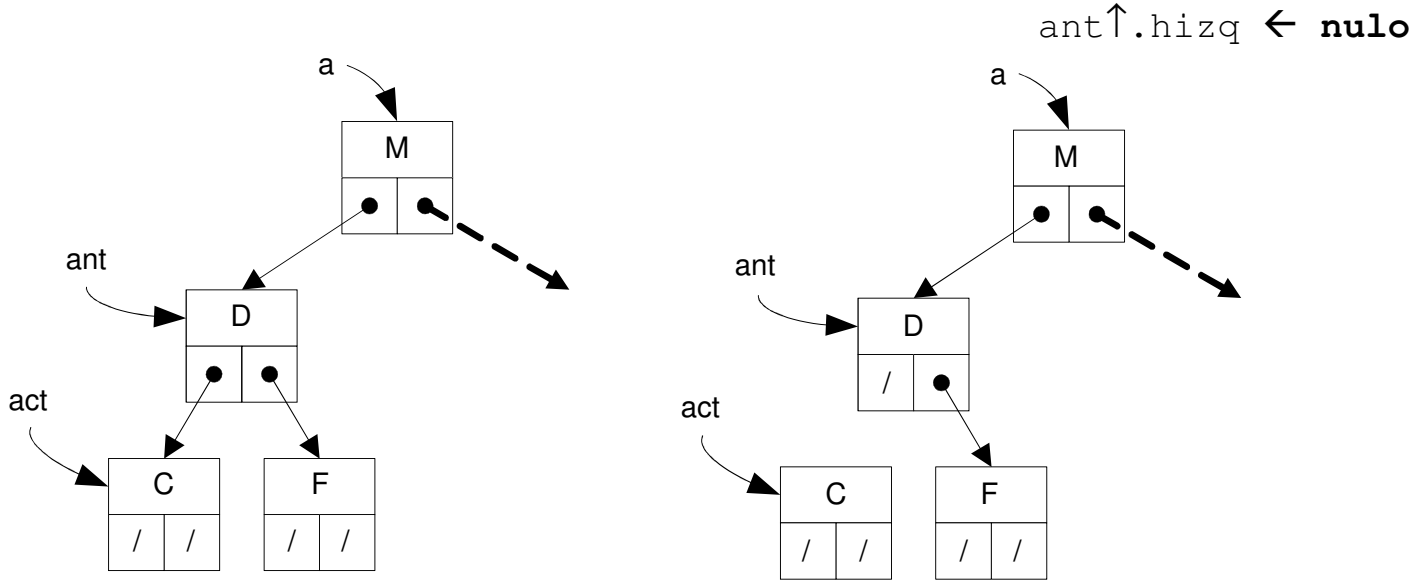
Borrado de un nodo

- Al algoritmo de borrado de un nodo debemos suministrarle la dirección del nodo a borrar (`act`) y la dirección del nodo anterior (`ant`).
  - Si queremos borrar el elemento raíz, `ant` será nulo.
- Dependiendo del valor del nodo anterior habrá que tomar distintas decisiones.
- Caso 1: Borrar una hoja.
  - Caso 1a: borrar el último nodo de un árbol de un único elemento.
    - ✓ El árbol toma un valor nulo.



Borrado de un nodo (II)

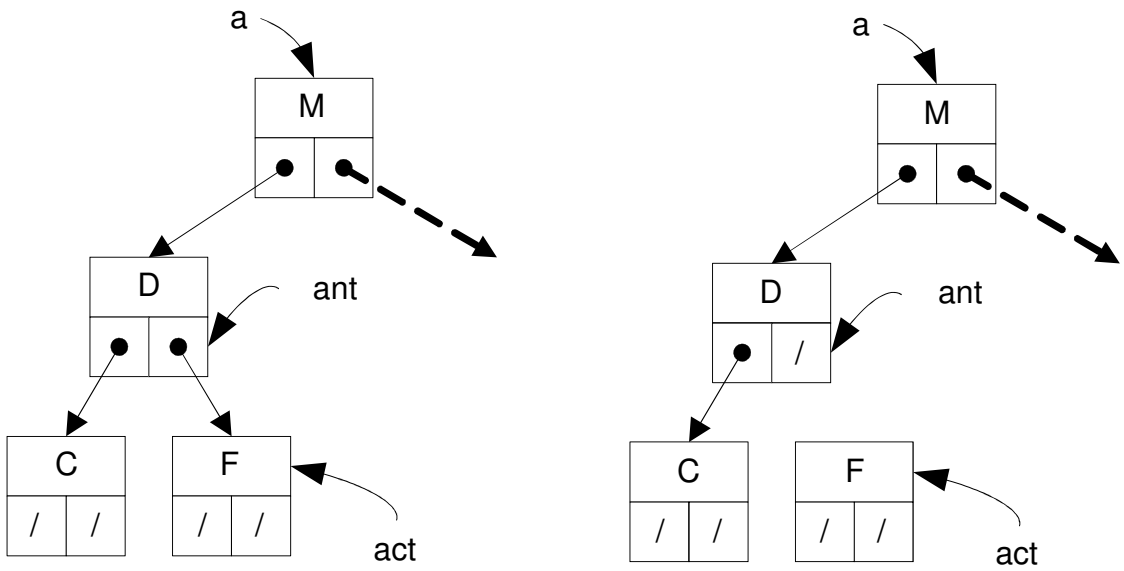
- Caso 1: Borrar una hoja.
  - Caso 1b: El nodo a borrar es hijo izquierdo del anterior.
    - ✓ El hijo izquierdo del anterior será nulo.



Borrado de un nodo (III)

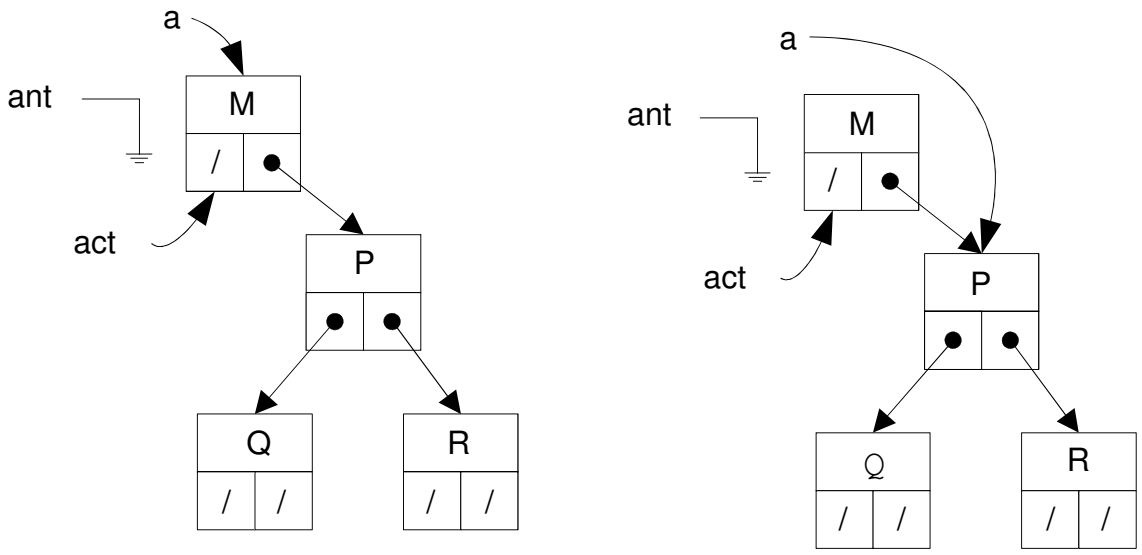
- Caso 1: Borrar una hoja.
  - Caso 1c: El nodo a borrar es hijo derecho del anterior.
    - ✓ El hijo derecho del anterior será nulo.

`ant↑.hder` ← **`nulo`**



Borrado de un nodo (IV)

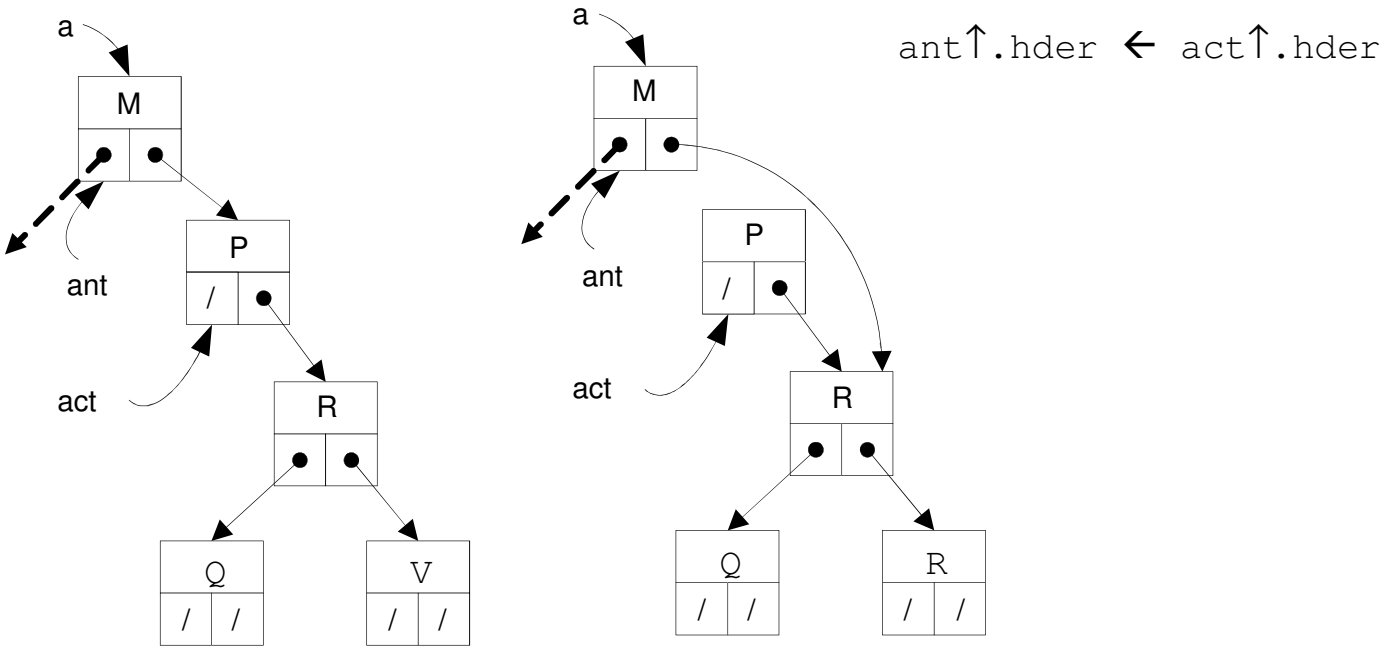
- Caso 2: Borrar un nodo con un solo hijo.
  - Caso 2.1: Sólo tiene hijo derecho
    - ✓ Caso 2.1.a: es el nodo raíz.
      - El nodo raíz apuntará al hijo derecho del anterior.




```
a ← a↑.hder
```

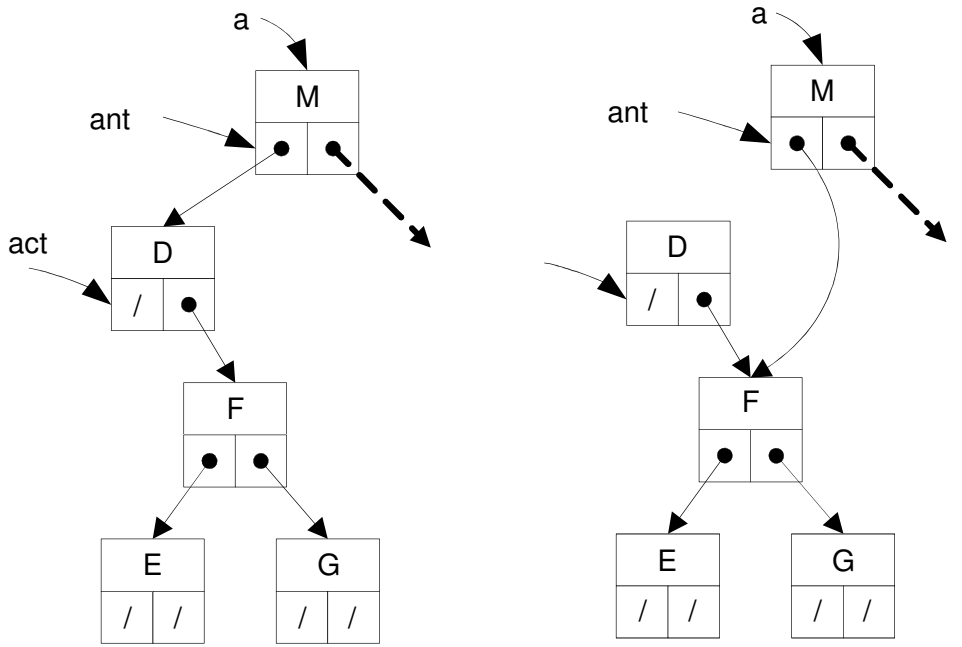
Borrado de un nodo (V)

- Caso 2: Borrar un nodo con un solo hijo.
  - Caso 2.1: Sólo tiene hijo derecho
    - ✓ Caso 2.1.b: es el hijo derecho del anterior.
      - El hijo derecho del anterior apuntará al hijo derecho del nodo a borrar.



Borrado de un nodo (VI)

- Caso 2: Borrar un nodo con un solo hijo.
  - Caso 2.1: Sólo tiene hijo derecho
    - ✓ Caso 2.1.c: es el hijo izquierdo del anterior.
      - El hijo izquierdo del anterior apuntará al hijo derecho del nodo a borrar.




```
ant↑.hizq ← act↑.hder
```

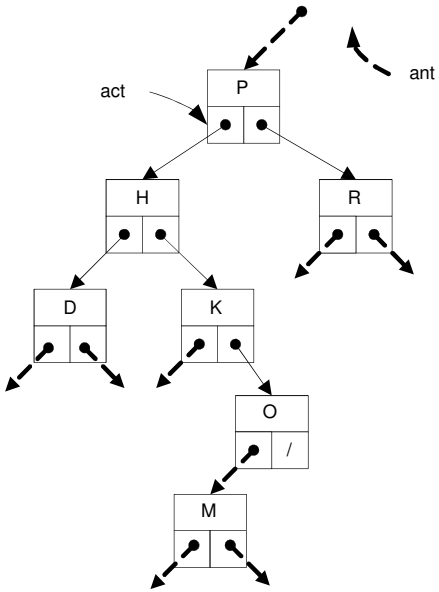
Borrado de un nodo (VI)

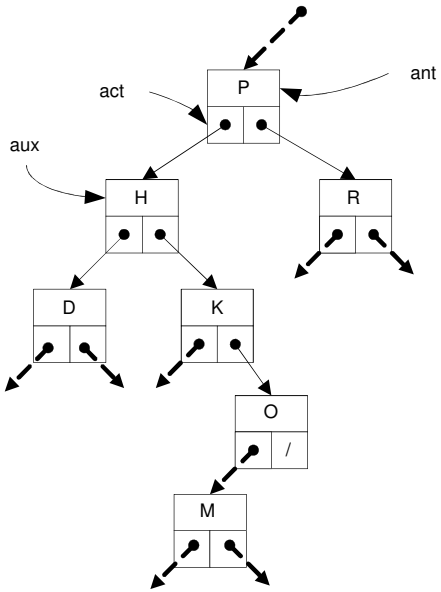
- Caso 2: Borrar un nodo con un solo hijo.
  - Caso 2.2: Sólo tiene hijo izquierdo.
    - ✓ Igual que el caso 2.1, pero se apuntará al hijo izquierdo del actual.
    - ✓ Caso 2.2.a: Es el nodo raíz.
      - `a ← a↑.hizq`
    - ✓ Caso 2.2.b: Es hijo derecho del anterior
      - `ant↑.hder ← act↑.hizq`
    - ✓ Caso 2.2.c: Es hijo izquierdo del anterior
      - `ant↑.hizq ← act↑.hizq`

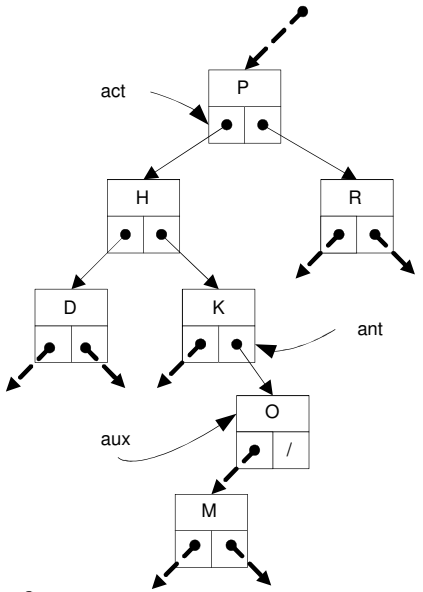
Borrar un nodo (VII)

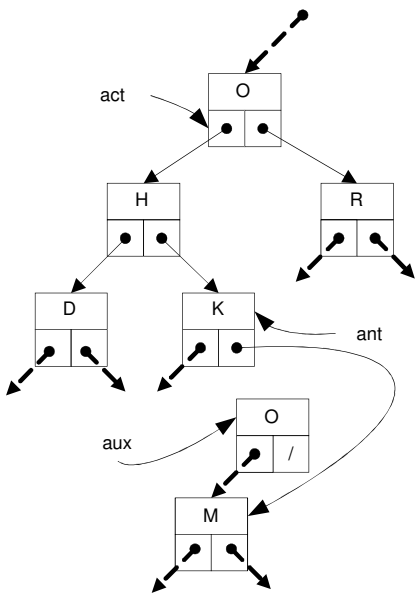
- Caso 3: Borrar un nodo con dos hijos.
  - En este caso no se borra el nodo `act`, sino que se sustituye su valor por el mayor valor entre los situados a la izquierda (o el menor entre los situados a la derecha).
    - ✓ De esta forma se mantienen los elementos y la estructura del árbol binario de búsqueda.
  - Lo que se borra es el nodo que contenía el valor que se ha subido.
    - ✓ Si el nodo situado inmediatamente a la izquierda es el "mayor de los menores", se modifica el hijo izquierdo del actual.

Borrar un nodo (VIII)



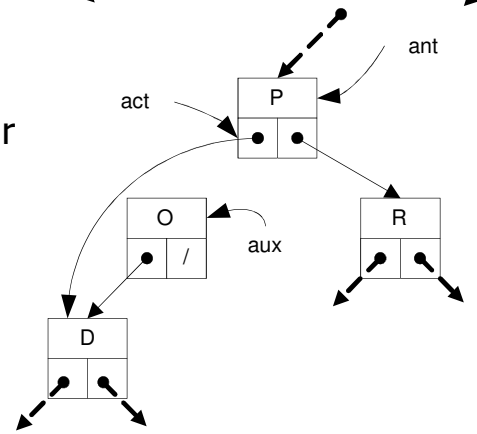






Si el "mayor de los menores" está a la izquierda del elemento a borrar

`act↑.hizq ← aux↑.hizq`



Borrar un nodo (IX)

```
procedimiento Borrar(ref arbol : a; valor arbol : act,ant)
var
   arbol : aux
inicio
   // si borramos una hoja. Caso 1.
   si (act↑.hder = nulo) y (act↑.hizq = nulo) entonces
      si ant = nulo entonces   // es el último nodo de un arbol
         a ← nulo
      si_no
         si ant↑.hder = act entonces // es el subarbol derecho
            ant↑.hder ← nulo
         si_no
            ant↑.hizq ← nulo
         fin_si
      fin_si
   si_no
      // si el nodo a borrar tiene dos hijos. Caso 3.
      si (act↑.hder <> nulo) y (act↑.hizq <> nulo) entonces
         // inicializa los puntero para buscar
         // el valor a reemplazar
         ant ← act
         aux ← act↑.hizq
```

Borrar un nodo (X)

```
        // localiza el nodo que contiene el valor
        // más proximo al que se va a suprimir
        mientras aux↑.hder <> nulo hacer
           ant ← aux
           aux ← aux↑.hder
        fin_mientras
        // pone el valor reemplazado en el nodo
        // cuyo valor se va a suprimir
        act↑.raiz ← aux↑.raiz
        // se suprime el nodo del que se ha tomado el valor *)
        si ant = act entonces
           ant↑.hizq ← aux↑.hizq
        si_no
           ant↑.hder ← aux↑.hizq
        fin_si
        act ← aux  // para poder disponer del nodo
     si_no
        // Solo tiene un hijo. Caso 2.
        // Se actualizan los punteros del nodo a borrar
        // según tenga un hijo izquierdo o un hijo derecho
        si act↑.hder <> nulo entonces   // act tiene un hijo derecho
           si ant = nulo entonces  // se trata del nodo raiz
```

Borrar un nodo (XI)

```
            a ← act↑.hder
         si_no                    // se trata de un nodo no raíz
            si ant↑.hder = act entonces  // se trata del hijo derecho
               ant↑.hder ← act↑.hder
            si_no                 // se trata del hijo izquierdo
               ant↑.hder ← act↑.hder
            fin_si
         fin_si
      si_no                       // act tiene un hijo izquierdo
         si ant = nulo entonces   // se trata de un nodo raíz
            a ← act↑.hizq
         si_no                   // se trata de un nodo no raíz
            si ant↑.hder = act entonces  // se trata del hijo derecho
               ant↑.hder ← act↑.hizq
             si_no                    // se trata del hijo izquierdo
               ant↑.hizq ← act↑.hizq
             fin_si
         fin_si
      fin_si
   fin_si
   fin_si
   liberar(act)
fin_procedimiento
```

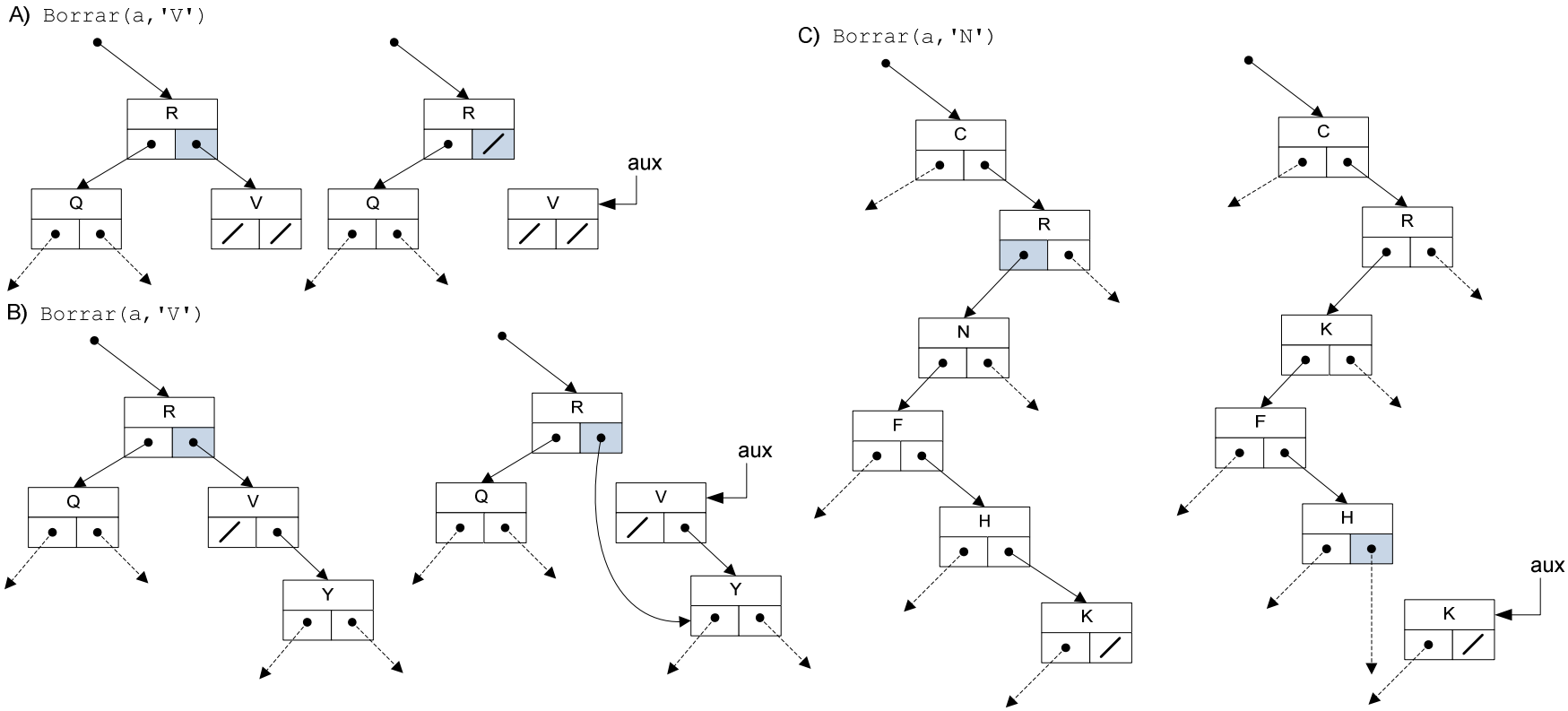
Borrado recursivo

- Se puede simplificar el borrado iterativo utilizando una solución recursiva.
- El procedimiento, en este caso buscaría el nodo y si lo encuentra, lo elimina.
- Caso base.
  - Existen dos casos base:
    - ✓ Si el árbol es nulo, el elemento no existe.
    - ✓ Si el elemento a borrar no es menor ni mayor que el elemento raíz (si se encuentra), se borra el nodo.
- Caso general.
  - Dos llamadas recursivas:
    - ✓ Si el elemento a borrar es menor que el raíz, hay que borrar en el subárbol izquierdo.
    - ✓ Si el elemento a borrar es mayor que el raíz, hay que borrar en el subárbol derecho.

Borrado recursivo

- El borrado del nodo sólo presentaría tres casos:
  - Si es una hoja: el predecesor sería nulo (figura A de la dispositiva siguiente).
  - Si tiene un único subárbol, el predecesor tendrá que apuntar al único hijo de dicho árbol (figura B de la dispositiva siguiente).
  - Si tiene dos hijos, se redistribuyen los nodos para que los nodos restantes mantengan la estructura de árbol binario se búsqueda:
    - ✓ Se sube a la posición del nodo a borrar, el contenido del nodo del elemento mayor del subárbol izquierdo.
    - ✓ Se elimina el nodo situado más a la derecha del subárbol izquierdo (el que se ha subido).
    - ✓ Para este proceso se realizará una llamada a otro procedimiento recursivo que sustituirá el valor del nodo a borrar.

Borrado recursivo (II)



Árbol que se pasa como argumento al procedimiento recursivo

Al final del proceso se liberará el nodo marcado con aux

Borrado recursivo (III)

```
procedimiento BorrarElemento(ref árbol : a; valor TipoElemento : e)
var
   árbol : aux
inicio
   si a = nulo entonces //El elemento no existe
   si_no
      si e < a↑.raíz entonces
         BorrarElemento(a↑.hizq,e)
      si_no
         si e > a↑.raíz entonces
            BorrarElemento(a↑.hder,e)
         si_no //Borra el elemento a
            aux ← a //aux guarda la dirección del nodo a borrar
            si a↑.hder = nulo entonces //Si no tiene hijo derecho
               a ← aux.hizq
            si_no
                si a↑.hizq = nulo entonces //Si no tiene hijo izquierdo
                    a ← aux↑.hder
                si_no //Tiene dos hijos
                   SubirNodo(aux↑.hizq, aux)
                fin_si
            fin_si
```

Borrado recursivo (IV)

```
            liberar(aux)
         fin_si
      fin_si
   fin_si
fin_procedimiento



procedimiento SubirNodo(ref árbol : a, aux)
inicio
   si a↑.hder <> nulo entonces //Moverse al mayor de los menores
      SubirNodo(a↑.hder,aux)
   si_no
      aux↑.raíz ← a↑.raíz
      aux ← a
      a ← aux↑.hizq
   fin_si
fin_procedimiento
```

Ejemplos

- Diseñe una función que devuelva la altura de un árbol.

```
entero : función Altura(valor árbol : a)
inicio
   si a <> nulo entonces
      devolver(1 + MayorAltura(Altura(a↑.hizq),Altura(a↑.hder))
   si_no
      devolver(0)
   fin_si
fin_función

entero función MayorAltura(E entero : a,b)
inicio
   si a > b entonces
      devolver(a)
   si_no
      devolver(b)
   fin_si
fin_función
```

Ejemplos (II)

- Diseñe un procedimiento que copie un árbol

```
procedimiento Copiar(valor árbol : a; ref árbol : c)
inicio
   si a <> nulo entonces
      reservar(c)
      c↑.raíz ← a↑.raíz
      Copiar(a↑.hizq, c↑.hizq)
      Copiar(a↑.hder, c↑.hder)
   si_no
      c ← nulo
   fin_si
fin_procedimiento
```

Ejemplos (III)

- Una función que compruebe si dos árboles son similares.
  - Si los dos no son nulos, son similares si al subárbol derecho y el izquierdo son similares.
  - Si los dos son nulos, son similares.
  - Si uno es nulo y el otro no, no son similares.

```
lógico:función SonSimilares(valor arbol: a1,a2)
inicio
 si (a1 <> nulo) y (a2 <> nulo) entonces
  devolver(SonSimilares(a1↑.hizq,a2↑.hizq) y SonSimilare(a1↑.hder,a2↑.hder))
 si_no
   devolver((a1 = nulo) y (a2 = nulo))
 fin_si
fin_función
```

Ejemplos (IV)

- Una función que comprueba si dos árboles son equivalentes.
  - Si los dos no son nulos y la información es la misma, son equivalentes si al subárbol derecho y el izquierdo son equivalentes.
  - Si la raíz es distinta no son equivalentes
  - Si los dos son nulos, son equivalentes.
  - Si uno es nulo y el otro no, no son equivalentes.

```
lógico:función SonEquivalentes(valor arbol: a1,a2)
inicio
   si (a1 <> nulo) y (a2 <> nulo) entonces
      si a1↑.raíz = a2↑.raíz entonces
         devolver(SonEquivalentes (a1↑.hizq,a2↑.hizq) y
                  SonEquivalentes (a1↑.hder,a2↑.hder))
      si_no
          devolver(falso)
   si_no
      devolver((a1 = nulo) y (a2 = nulo))
   fin_si
fin_función
```

Ejemplos (V)

- Diseñe un algoritmo que calcule el número de nodos de un árbol.
  - Si el árbol está vacío, el número de nodos es 0,
  - Si no está vacío, al menos tiene un nodo y, además puede tener más nodos a la izquierda o a la derecha.

```
entero función NúmeroDeNodos(valor arbol: a)
inicio
   si a = nulo entonces
      devolver(0)
   si_no
      devolver(1 + NúmeroDeNodos(a↑.hizq) + NúmeroDeNodos(a↑.der))
   fin_si
fin_función
```

Ejemplos (VI)

- Diseñe un algoritmo que devuelva el peso de un árbol.
  - Si el árbol está vacío, el peso es 0.
  - Si el hijo izquierdo y el hijo derecho del árbol son nulos, el peso es 1.
  - Si no, el peso será la suma de los pesos de sus dos hijos.

```
entero función Peso(valor arbol: a)
inicio
   si a = nulo entonces
      devolver(0)
   si_no
      si (a↑.hizq = nulo) y (a↑.der = nulo) entonces
          devolver(1)
      si_no
         devolver(Peso(a↑.hizq) + Peso(a↑.der))
      fin_si
   fin_si
fin_función
```

Ejercicios (VII)

1. Se tiene almacenado en un árbol binario de búsqueda una serie de números ordenados de menor a mayor. Diseñe un procedimiento que devuelva los números pero ordenados de mayor a menor.
2. Se tiene almacenada una frase en un archivo de texto. Diseñe un algoritmo que almacene las palabras de la frase en un árbol binario de búsqueda. Las palabras del árbol no se podrán repetir y en cada nodo se almacenará la palabra y el número de veces que aparece en la frase. El algoritmo deberá mostrar por pantalla las palabras ordenadas por la frecuencia de aparición.

Ejercicios (VIII)

3. En un archivo directo se tiene almacenada información sobre una serie de automóviles. Por cada registro del archivo aparece la matrícula (campo clave), la marca y el modelo y el DNI del propietario. Los registros están distribuidos de forma aleatoria por el archivo que tiene una capacidad para 1000 vehículos. Diseñe un algoritmo que cree un índice del archivo en un árbol binario de búsqueda. Cada nodo del árbol deberá almacenar la matrícula y el número de registro relativo del archivo donde está almacenado el vehícuclo.

   A partir de este índice diseñe los procedimientos para:

   - ✓ Realizar un listado con todos los datos de los vehículos del archivo.
   - ✓ Sacar por pantalla los datos de un vehículo cuya matrícula se introduce por teclado.
   - ✓ Realizar la inserción de un nuevo vehículo en el archivo, insertando también la información necesaria en el índice.
   - ✓ Realizar la eliminación de un vehículo en el archivo.