Árboles N-arios

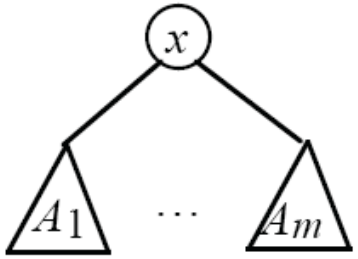
Lección 15

Esquema

- Una especificación de TADs arborescentes para árboles n-arios genéricos
- Implementación dinámica
- Implementación de recorridos
- Implementación estática

Conceptos, definiciones y terminología básica

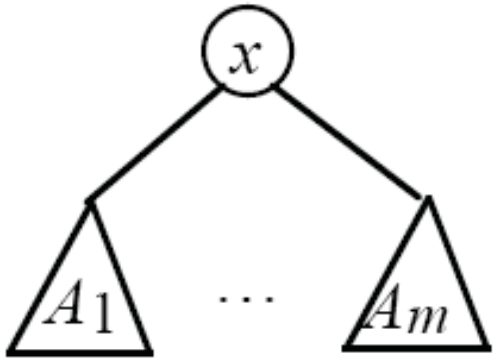
- **Árbol**:
  - Definición no recursiva:
    - Conjunto de elementos de un mismo tipo, denominados *nodos*, que pueden representarse en un grafo no orientado, conexo y acíclico, en el que existe un vértice destacado denominado *raíz*
  - Definición recursiva:
    - Un árbol es un conjunto *no vacío* de elementos del mismo tipo tal que:
      - Existe un elemento destacado llamado **raíz** del árbol
      - el resto de los elementos se distribuyen en m subconjuntos disjuntos (0 ≤m), llamados **subárboles** del árbol original, cada uno de los cuales es a su vez un árbol



Conceptos, definiciones y terminología básica

El grado, n, es el número máximo de subárboles hijos que puede tener un árbol n-ario

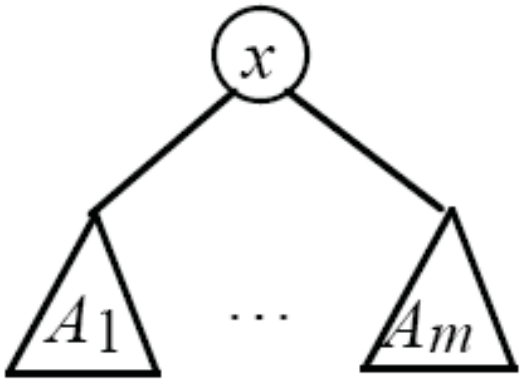
- **Árbol N-ario**:
  - Un árbol n-ario (con grado n≥1) es un conjunto no vacío de elementos del mismo tipo tal que:
    - Existe un elemento destacado llamado raíz del árbol
    - el resto de los elementos se distribuyen en m subconjuntos disjuntos (0 ≤m ≤n), llamados subárboles del árbol original, cada uno de los cuales es a su vez un árbol n-ario



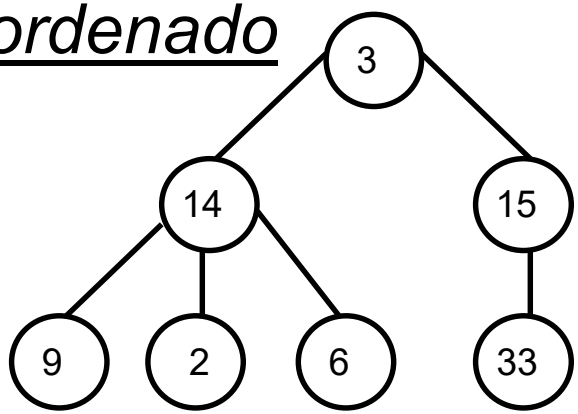
Por definición, un árbol n-ario no puede ser vacío.

Conceptos, definiciones y terminología básica

- **Árbol ordenado**:
  - Si en el conjunto de subárboles de un árbol se supone definida una relación de orden total, el árbol se denomina *ordenado*

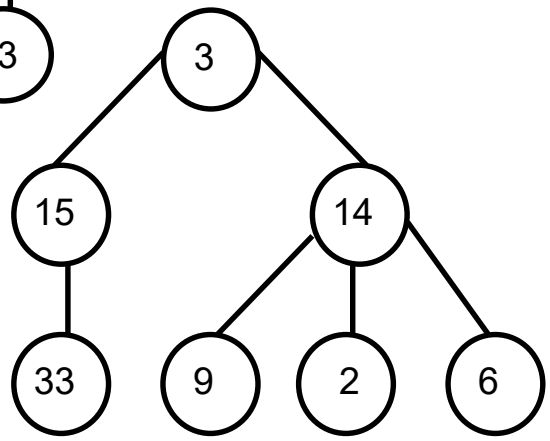


Árbol ordenado con raíz X y subárboles $A_1 \ldots A_m$



← Árbol de números enteros de grado 3

Otro árbol distinto, → de números enteros y grado 3



Leyenda: **Nodo** ○ **Árbol** △

Conceptos, definiciones y terminología básica

- En la definición recursiva, definimos ***bosque***:
  - Un **bosque ordenado** de grado n (con n≥1) es una secuencia A_1, …A_m, con 0 ≤m ≤n , de árboles n-arios ordenados.
    - Si m=0, el bosque se llama vacío

Un bosque, de árboles n-arios, puede ser un bosque vacío.

  - Un **árbol n-ario** se genera a partir de un elemento y un *bosque ordenado de grado n*: basta considerar el elemento como raíz del árbol, y el bosque como sus subárboles

Por definición, un árbol n-ario no puede ser vacío.

  - Los **bosques** se generarán como secuencias de árboles, a partir de un bosque vacío, y una operación de añadir por la derecha un árbol a un bosque

{y de acuerdo a estas definiciones se construye la siguiente especificación: árbolesOrdenados… sin limitación en el grado}

Especificación

espec árbolesOrdenados *{contiene la especificación de 2 TADs: árbol, y bosque (de cualquier grado, sin limitar por el grado)}*

usa booleanos, naturales

parámetro formal

género elmto

fpf

géneros bosque, árbol *{Los valores del género bosque representan secuencias de elementos del género árbol. Los valores del género árbol se forman con una raíz de género elmto y un bosque, de árboles hijos, del mismo género. Un bosque puede ser vacío, pero un árbol no puede ser vacío (al menos tiene un elmto raíz).}*

operaciones

crearVacío: → bosque

{Devuelve el bosque vacío, es decir, la secuencia vacía de árboles}

añadirÚltimo: bosque b , árbol a → bosque

{Devuelve un bosque igual al resultante de añadir el árbol a como último árbol en b}

long: bosque b → natural

{Devuelve el nº de árboles del bosque b, es decir, la longitud de la secuencia árboles}

parcial observar: bosque b , natural n → árbol

{Devuelve el n-ésimo árbol de la secuencia de árboles b.
Parcial: la operación sólo está definida si 0<n<=long(b) }

parcial resto: bosque b → bosque

{Devuelve un bosque igual al resultante de descartar el primer árbol de b.
Parcial: la operación sólo está definida si long(b)>0 }

…

Especificación

...

plantar: elmto e , bosque b → árbol

{Devuelve un árbol cuya raíz es e y sus subárboles son un bosque igual que b}

raíz: árbol a → elmto

{Devuelve la raíz del árbol a}

elBosque: árbol a → bosque

{Devuelve un bosque igual al bosque de los subárboles (hijos) del árbol a}

numHijos: árbol a → natural

{Devuelve el nº de subárboles de a, es decir long(elBosque(a)) }

parcial subÁrbol: árbol a , natural n → árbol

{Devuelve un árbol igual al n-ésimo subárbol de a.

Parcial: la operación sólo está definida si 0<n<=numHijos(a) }

esHoja?: árbol a → booleano

{Devuelve verdad si y sólo si a no tiene subárboles, es decir, numHijos(a)=0 }

alturaBosque: bosque b → natural

{Si long(b)>0, devuelve la altura del árbol más alto de b. Si long(b)=0 entonces devuelve 0 (caso de bosque vacío)}

alturaÁrbol: árbol a → natural

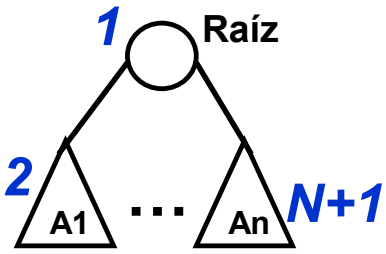
{Devuelve la altura del árbol a (longitud del camino más largo desde la raíz a una hoja)}

...

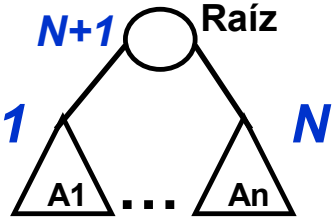
fespec

Recorridos en árboles n-arios

Un recorrido de un árbol consiste en visitar todos los elementos del árbol una sola vez



- **Recorridos en profundidad**:
  - Recorrido en **pre-orden**:
    1. se visita la raíz
    2. se recorren en pre-orden todos los subárboles, de izquierda a derecha



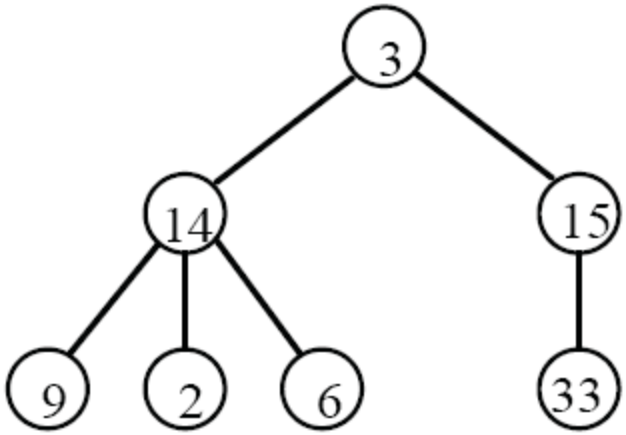
  - Recorrido en **post-orden**:
    1. se recorren en post-orden todos los subárboles, de izquierda a derecha
    2. se visita la raíz

- El **recorrido en anchura**:
  - primero se visitan los elementos del nivel 0, luego los del nivel 1, y así sucesivamente,
  - En cada nivel, se visitan los elementos de izquierda a derecha

Implementación Dinámica

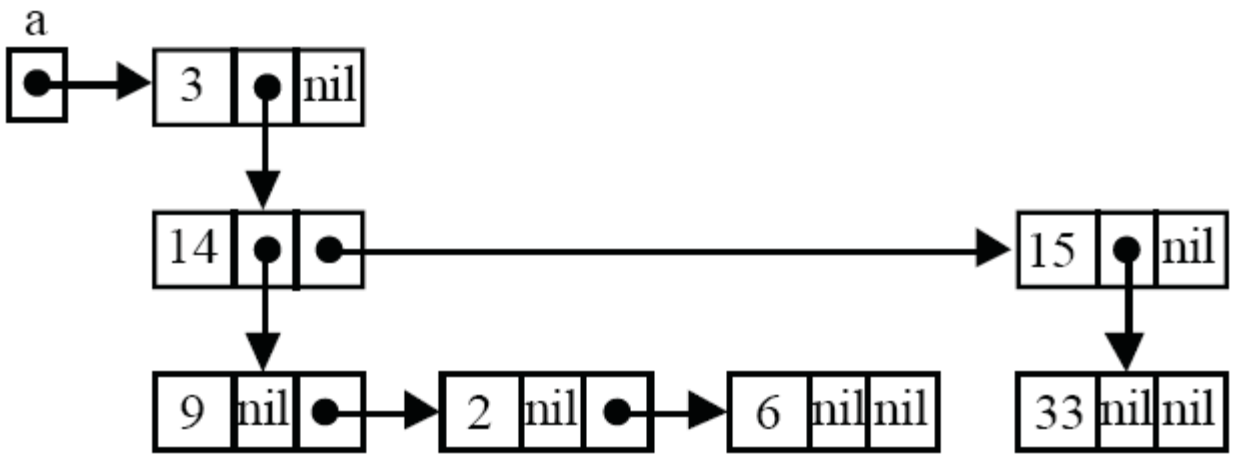
{También llamada: Child-Brother}

- Representación “Primogénito - sig. Hermano”




```
tipos árbol = ↑nodo;
      nodo = registro
               dato:elemento;
               primogénito,sigHermano:árbol
             freg;
bosque = árbol
```

{O bien de tipo bosque}



Implementación "primogénito - sig. hermano"

Modulo genérico árbolesOrdenados
importa listasGenéricas *{las de la lección 8, para las operaciones de recorridos}*
parámetro
  tipo elemento
exporta
  tipos árbol, bosque *{Los valores del tipo bosque representan secuencias de elementos del tipo árbol. Los valores del tipo árbol se forman con una raíz de tipo elemento y un bosque de árboles hijos. Un bosque puede ser vacío, pero por definición, un árbol no puede ser vacío, al menos tiene un elemento raíz. }*
  procedimiento crearVacío(**sal** b:bosque)
  {devuelve el bosque vacío, es decir, la secuencia vacía de árboles}
  procedimiento añadirÚltimo(**e/s** b:bosque; **ent** a:árbol) *{Añade el árbol a al bosque b, como último árbol del bosque (Cuidado: no crea copia profunda de dicho árbol)}*
  función long(b:bosque) **devuelve** natural
  {devuelve el nº de árboles del bosque b, es decir, la longitud de la secuencia}
  procedimiento observar(**ent** b:bosque; **ent** n:natural; **sal** error:booleano; **sal** a:árbol)
  {si 1≤n≤long(b): devuelve el n-ésimo árbol de la secuencia de árboles b dada y error=falso; en caso contrario devuelve error=verdad. (Cuidado: no crea una copia profunda de dicho árbol)}
  procedimiento resto(**ent** b:bosque; **sal** error:booleano; **sal** rb:bosque)
  {si b no es vacío devuelve en rb el bosque resultante de descartar el primer árbol de b y error=falso; en caso contrario devuelve error=verdad. (Cuidado: no crea una copia profunda de dicho bosque)}
  ...

Implementación "primogénito - sig. hermano"

... *{Recordatorio: un árbol a no puede ser árbol vacío, por definición, y las operaciones no permiten obtener un árbol vacío}*

procedimiento plantar(**sal** a:árbol; **ent** e:elemento; **ent** b:bosque)

{devuelve un árbol a cuya raíz es e y sus subárboles son el bosque b dado. (Cuidado: no crea una copia profunda de dicho bosque)}

función raíz(a:árbol) **devuelve** elemento

{devuelve la raíz del árbol a}

procedimiento elBosque(**ent** a:árbol; **sal** b:bosque)

{devuelve el bosque de los subárboles del árbol a. (Cuidado: no crea una copia profunda de dicho bosque)}

función numHijos(a:árbol) **devuelve** natural

{devuelve el nº de subárboles de a, es decir long(elBosque(a) }

procedimiento subÁrbol(**ent** a:árbol; **ent** n:natural;
sal error:booleano; **sal** sa:árbol)

{si 1≤n≤numHijos(a): devuelve el n-ésimo subárbol de a y error=falso; en caso contrario error=verdad. (Cuidado: no crea una copia profunda de dicho árbol)}

función esHoja(a:árbol) **devuelve** booleano *{devuelve verdad si y sólo si a no tiene subárboles, es decir, numHijos(a)=0 }*

función alturaBosque(b:bosque) **devuelve** natural

{devuelve la altura del árbol más alto de b, o devuelve 0 si long(b)=0}

función alturaÁrbol(a:árbol) **devuelve** natural

{devuelve la altura del árbol a dado}

...

Implementación "primogénito - sig. hermano"

...

procedimiento duplicarBosque(**sal** nuevo:bosque; **ent** viejo:bosque)

{Duplica la representación del bosque viejo guardándolo en nuevo (copia profunda)}

procedimiento liberarBosque(**e/s** b:bosque)

{Libera la memoria dinámica accesible desde b, y dejar b como bosque vacío}

procedimiento duplicarÁrbol(**sal** nuevo:árbol; **ent** viejo:árbol)

{Duplica la representación del árbol viejo guardándolo en nuevo (copia profunda)}

procedimiento liberarÁrbol(**e/s** a:árbol)

{Libera la memoria dinámica accesible desde a, su resultado no debe ser usado como entrada, para ninguna otra operación}

procedimiento preOrden(**ent** a:árbol; **e/s** l:lista)

{Añade a la lista l los elementos de a recorridos en pre-orden}

procedimiento postOrden(**ent** a:árbol; **e/s** l:lista)

{Añade a la lista l los elementos de a recorridos en post-orden}

procedimiento preBosque(**ent** b:bosque; **e/s** l:lista)

{añade a la lista l los elementos de todos los árboles del bosque b recorridos en pre-orden}

procedimiento postBosque(**ent** b:bosque; **e/s** l:lista)

{añade a la lista l los elementos de todos los árboles del bosque b recorridos en post-orden}

...

Implementación "primogénito - sig. hermano"

...

{Recordatorio:

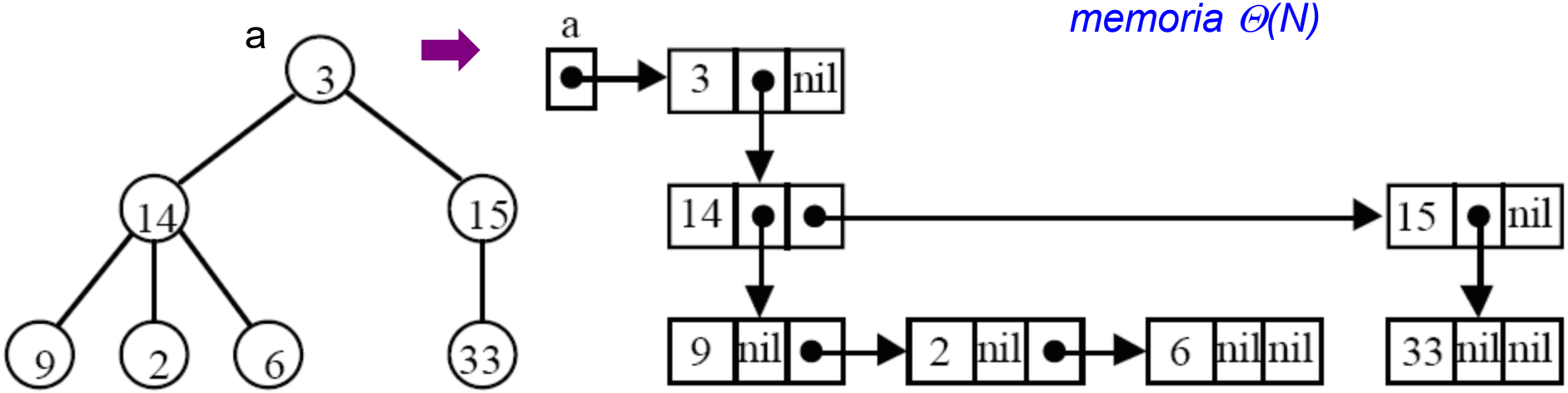
un árbol a no puede ser árbol vacío, por definición, y las operaciones del TAD no permiten obtener un árbol vacío.

Pero, internamente árbol es un puntero…}

```
implementación
    tipos
        árbol = ↑nodo;
        nodo = registro
                  dato:elemento;
                  primogénito,sigHermano:árbol
               freg;
        bosque = árbol
```

...

Coste en memoria Θ(N)



Implementación "primogénito - sig. hermano"

```
...
procedimiento crearVacío(sal b:bosque)        Coste Θ(1)
principio
    b:=nil
fin
                                  (cuidado: no se crea una copia profunda de dicho árbol)
procedimiento añadirÚltimo(e/s b:bosque; ent a:árbol)
  variable aux:bosque
principio
    si b=nil entonces
        b:=a {puede requerir casting o conversión de tipos, ejemplo:  b:=(bosque)a  }
    sino    {Versión iterativa de recorrido por la lista de hermanos (es decir, el bosque)}
        aux:=b;
        mientrasQue aux↑.sigHermano≠nil hacer
            aux:=aux↑.sigHermano
        fmq;
        aux↑.sigHermano:=a
    fsi
fin
...
```

¿Coste?
Depende de la long. de la lista → lineal número de hermanos (=grado)

Coste Θ(lineal en el grado del árbol)

```
…
función long (b:bosque) devuelve natural
principio
    si b=nil entonces
        devuelve 0
    sino
        devuelve 1+long(b↑.sigHermano)
    fsi
fin
```

{Versión recursiva de recorrido por la lista de hermanos (árboles del bosque), en este caso contando el número de elementos}

long. de la lista → número de árboles en el bosque (=grado)

Coste Θ(lineal en el grado del árbol)

(cuidado: no se crea una copia profunda de dicho árbol)

```
procedimiento observar(ent b:bosque; ent n:natural;
                      sal error:booleano; sal a:árbol)
principio
    si n<1 or b=nil entonces error:=verdad
    sino
        si n=1 entonces
            a:=b; error:=falso
        sino
            observar(b↑.sigHermano,n-1,error,a)
        fsi
    fsi
fin
…
```

Coste Θ(n), siendo n el grado

{llamada recursiva por cada árbol del bosque hasta llegar al buscado…}

```
…                                          (cuidado: no crea una copia profunda de dicho bosque)
procedimiento resto(ent b:bosque; sal error:booleano; sal rb:bosque)
principio
    si b=nil entonces
        error:=verdad
    sino
        error:=falso; rb:=b↑.sigHermano
    fsi
fin
```

Coste $\Theta(1)$

```
                                           (cuidado: no crea una copia profunda de dicho bosque)
procedimiento plantar(sal a:árbol; ent e:elemento; ent b:bosque)
principio
    nuevoDato(a);
    a↑.dato:=e;
    a↑.primogénito:=b;
    a↑.sigHermano:=nil
fin
```

Coste $\Theta(1)$

```
función raíz(a:árbol) devuelve elemento
principio
    devuelve a↑.dato
fin
```

Coste $\Theta(1)$

```
…
```

{Recordatorio: un árbol a no puede ser árbol vacío, por definición, y las operaciones del TAD no permiten obtener un árbol vacío}

```
...
procedimiento elBosque(ent a:árbol; sal b:bosque)
principio
    b:=a↑.primogénito
fin
```

(cuidado: no crea una copia profunda de dicho bosque)

Coste Θ(1)

{Recordatorio: un árbol a no puede ser árbol vacío, por definición, y las operaciones no permiten obtener un árbol vacío}

(cuidado: no crea una copia profunda de dicho árbol)

```
procedimiento subárbol(ent a:árbol; ent n:natural;
                       sal error:booleano; sal sa:árbol)
  variable b:bosque
principio
    elBosque(a,b); observar(b,n,error,sa)
    {O bien:
      observar(a↑.primogénito,n,error,sa)   }
fin
```

Coste Θ(n), siendo n el grado del árbol

```
función numHijos(a:árbol) devuelve natural
  variable b:bosque
principio
    elBosque(a,b); devuelve long(b)
    {O bien:   devuelve long(a↑.primogénito)   }
fin
```

Coste Θ(n), siendo n el grado del árbol

```
función esHoja(a:árbol) devuelve booleano
principio
    devuelve a↑.primogénito=nil
fin
```

Coste Θ(1)

Implementación "primogénito - sig. hermano"

```
...
función alturaBosque(b:bosque) devuelve natural
{Si long(b)>0, devuelve la altura del árbol más alto de b. Si long(b)=0 entonces  devuelve
    0 (caso de bosque vacío)}
principio
    si b=nil entonces
        devuelve 0
    sino
        devuelve max(alturaÁrbol(b),alturaBosque(b↑.sigHermano))
    fsi
fin

función alturaÁrbol(a:árbol) devuelve natural
{Devuelve la altura del árbol a}
  variable b:bosque
principio
    si esHoja(a) entonces
        devuelve 0
    sino
        devuelve 1+alturaBosque(a↑.primogénito)
    fsi
 fin
...
```

¿Costes?
Θ(N), siendo N el número de nodos

{Recordatorio: un árbol a no puede ser árbol vacío, por definición, y las operaciones no permiten obtener un árbol vacío}

Implementación "primogénito - sig. hermano"

```
...
procedimiento duplicarBosque(sal nuevo:bosque; ent viejo:bosque)
  variables primerÁrbol:árbol; error:booleano
principio
    si viejo=nil entonces nuevo:=nil
    sino
        observar(viejo, 1, error, primerÁrbol); {O bien:}
        duplicarÁrbol(nuevo, primerÁrbol);    {duplicarÁrbol(nuevo,viejo); }
        duplicarBosque(nuevo↑.sigHermano, viejo↑.sigHermano)
    fsi
fin

procedimiento duplicarÁrbol (sal nuevo:árbol; ent viejo:árbol)
  variables viejoBosque,nuevoBosque:bosque
principio
    duplicarBosque(nuevoBosque, viejo↑.primogénito);
    plantar(nuevo, raíz(viejo), nuevoBosque)

    { raíz(viejo)     o bien      viejo↑.dato }
fin
...
```

¿Costes?
Θ(N), siendo N el número de nodos

{Recordatorio: un árbol a no puede ser árbol vacío, por definición, y las operaciones no permiten obtener un árbol vacío}

Implementación "primogénito - sig. hermano"

```
...
procedimiento liberarBosque (e/s b:bosque)
principio
   si b≠nil entonces
        liberarBosque(b↑.sigHermano);
        liberarÁrbol(b);
        {b:=nil}
   fsi
fin

procedimiento liberarÁrbol (e/s a:árbol)
principio
   si a≠nil entonces
        liberarBosque(a↑.primogénito);
        disponer(a);
        a:=nil {asignar valor seguro al puntero}
   fsi
fin
```

¿Costes?
Θ(N), siendo N el número de nodos

{Recordatorio: un árbol a no puede ser árbol vacío, por definición, y las operaciones no permiten obtener un árbol vacío…

¡Salvo la operación liberar, que hemos añadido!

Por eso su resultado no debe ser usado, como entrada, para ninguna otra operación,

(o bien habría que modificar todas las operaciones que tengan un árbol de entrada, para que lo tengan en cuenta y se protejan de ese caso) }

Implementación "primogénito - sig. hermano"

Coste en tiempo de las operaciones:

- *crearVacío, resto, plantar, raíz, elBosque* y *esHoja* → $\Theta(1)$
- Modificando la representación(*), puede lograrse que las operaciones *añadirÚltimo, long, numHijos, alturaBosque*, y *alturaArbol* tengan también coste → $\Theta(1)$
- *Observar* y *subárbol* → $\Theta(n)$ siendo n es el grado del árbol (y es de esperar que se mantenga mucho menor que el número de nodos del bosque/árbol)
- *duplicarBosque*, *duplicarÁrbol*, *liberarBosque*, *liberarÁrbol* → Θ(lineal en el Nº de nodos del bosque/árbol)

(*) Cambiando a algo como: Bosque tendría que tener punteros a primera y última celda de una lista enlazada, además de longitud de la lista, y altura del bosque. La Celda contendría un puntero a la siguiente celda y un puntero árbol. Árbol sería un puntero a un nodo (raíz del árbol), y el Nodo tendría que contener el elemento, la altura del árbol (que tiene como raíz ese nodo), y el bosque de subárboles hijos.

Implementación "primogénito - sig. hermano"

```
...
procedimiento preBosque(ent b:bosque; e/s L:lista)
{añade a la lista l los elementos de todos los árboles del bosque b recorridos en pre-orden}
principio
    si b≠nil  entonces
        preOrden(b, L); {recorrer en pre-orden el primer árbol del bosque}
        preBosque(b↑.sigHermano, L) {recorrer en pre-orden el resto del bosque}
    fsi
fin


procedimiento preOrden(ent a:árbol; e/s L:lista)
{añade a la lista l los elementos del árbol a, recorridos en pre-orden}
principio
    añadirÚltimo(L, a↑.dato);          {visitar la raíz, i.e. añadirla a la lista}
    preBosque(a↑.primogénito, L)   {recorrer el bosque de hijos de la raíz}
fin
...
```

Implementación "primogénito - sig. hermano"

```
...
procedimiento postBosque(ent b:bosque; e/s L:lista)
{añade a la lista l los elementos de todos los árboles del bosque b recorridos en post-orden}
principio
    si b≠nil  entonces
        postOrden(b, L); {recorrer en post-orden el primer árbol del bosque}
        postBosque(b↑.sigHermano, L) {recorrer en post-orden el resto del bosque}
    fsi
fin


procedimiento postOrden(ent a:árbol; e/s L:lista)
{añade a la lista l los elementos del árbol a, recorridos en post-orden}
principio
    postBosque(a↑.primogénito, L)   {recorrer el bosque de hijos de la raíz}
    añadirÚltimo(L, a↑.dato);   {visitar la raíz, i.e. añadirla a la lista}
fin
```

{A continuación VERSIONES EQUIVALENTES, más condensadas, de recorridos en pre-orden y post-orden de bosques n-arios (válidos también para recorrer árboles n-arios)}

fin ***{del módulo}***

… VERSIONES EQUIVALENTES CONDENSADAS: (Árbol = bosque de 1 árbol)

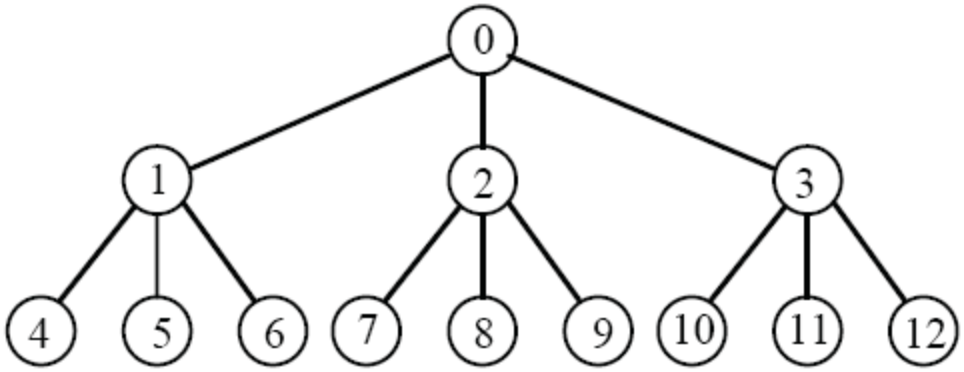
```
procedimiento preOrden(ent b:bosque; e/s L:lista)
{añade a la lista L los elementos de todos los árboles del bosque b recorridos en pre-orden}
principio
    si b≠nil entonces
        {recorrer (en pre-orden) el primer árbol del bosque:}
          añadirÚltimo(L, b↑.dato);
          preOrden(b↑.primogénito, L);
        {recorrer el resto del bosque:}
          preOrden(b↑.sigHermano, L)
    fsi
fin

procedimiento postOrden(ent b:bosque; e/s L:lista)
{añade a la lista L los elementos de todos los árboles del bosque b recorridos en post-orden}
principio
    si b≠nil entonces
        {recorrer (en post-orden) el primer árbol del bosque:}
          postOrden(b↑.primogénito, L);
          añadirÚltimo(L, b↑.dato);
        {recorrer el resto del bosque:}
          postOrden(b↑.sigHermano, L)
    fsi
fin
```

{Se pueden construir versiones iterativas de los recorridos de forma similar a lo visto para árboles binarios: utilizando pila, o cola, auxiliar de punteros a nodos, respectivamente }

Otras implementaciones

- Definiciones:
  - Un **árbol** n-ario se dice **homogéneo** si todos sus subárboles excepto las hojas tienen n hijos.
  - Un **árbol** homogéneo es **completo** cuando todas sus hojas tienen la misma profundidad
  - Un **árbol** se dice **casi-completo** cuando se puede obtener a partir de un árbol completo eliminando hojas consecutivas del último nivel, comenzando por la que está más a la derecha



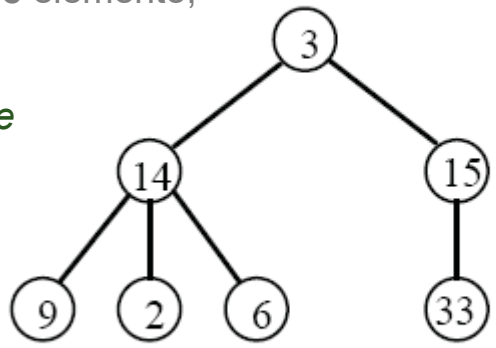
Otras implementaciones

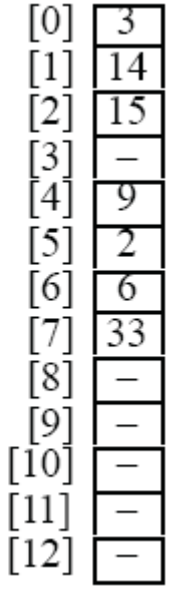
- Implementación estática:

constante max = ... *{máximo número de elementos en el árbol}*

Tipo árbol = **vector**[0..max-1] **de** elemento;

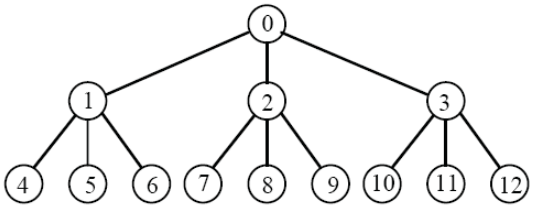
{Implementación eficiente en memoria únicamente si se trata de almacenar árboles completos o casi-completos, de tamaño máximo conocido y ajustado a su uso real}





*La implementación se basa en la forma que tendría el árbol completo de grado n → **se reserva una componente concreta del vector, para cada nodo del árbol completo o casi-completo**.*

La componente reservada corresponde al orden según el recorrido en anchura del árbol completo o casi-completo:



Puede ser necesario marcar en cada componente si se está usando o no.

{ Implementación muy eficiente para acceder desde un nodo (componente) a su: padre, hijos, hermanos, …

ya que, siendo n el grado del árbol, se cumple: }

elemento	*índice*	*condición*
e	i	
hijo k–ésimo de e	$n*i+k$	si $n*i+k<$max
padre de e	$(i-1)$ div n	si $i\neq 0$
hermano siguiente a e	$i+1$	si i mod $n\neq 0$
n.º de orden entre sus hermanos	$((i-1)$ mod $n)+1$	si $i\neq 0$