

Estructuras de Datos y Algoritmos

Árboles binarios

LECCIÓN 12

© All wrongs reversed – bajo licencia CC-BY-NC-SA 4.0



Universidad
Zaragoza

Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza, España

Curso 2024/2025

Grado en Ingeniería Informática
UNIVERSIDAD DE ZARAGOZA

Aula 0.04, Edificio Agustín de Betancourt



Adaptadas de diapositivas de Javier Campos

Índice

- 1 Concepto
- 2 Especificación
- 3 Recorridos
- 4 Implementación estática
- 5 Implementación dinámica

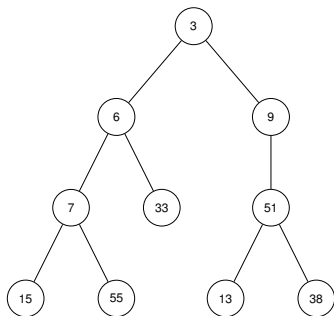
Índice

- 1 Concepto
- 2 Especificación
- 3 Recorridos
- 4 Implementación estática
- 5 Implementación dinámica

Árboles binarios

Definición

- Conjunto de elementos (o nodos) del mismo tipo, tal que:
 - O bien es el conjunto vacío (y entonces se llama **árbol vacío**);
 - O bien es no vacío, en cuyo caso existe un elemento destacado llamado **raíz** y el resto de los elementos se distribuyen en dos subconjuntos disjuntos (llamados **subárbol izquierdo** y **subárbol derecho**), cada uno de los cuales es un árbol binario



Índice

1 Concepto

2 Especificación

3 Recorridos

4 Implementación estática

5 Implementación dinámica

Árboles binarios

Especificación

```
espec árbolesBinarios
  usa booleanos,naturales
  parámetro formal
    género elemento
  fpf
  género arbin
  operaciones
    vacío: -> arbin
    {Devuelve el árbol vacío}
    plantar: elemento e, arbin ai, arbin ad -> arbin
    {Devuelve un árbol cuyo elemento raíz es e, subárbol izquierdo es ai y derecho es ad}
    esVacio?: arbin a -> booleano
    {Devuelve verdad si y sólo si a es el árbol vacío}
    parcial raíz: arbin a -> elemento
    {Devuelve el elemento raíz de a. Parcial: la operación no está definida si a es vacío}
    parcial subIzq: arbin a -> arbin
    {Devuelve el subárbol izquierdo de a. Parcial: la op. no está definida si a es vacío}
    parcial subDer: arbin a -> arbin
    {Devuelve el subárbol derecho de a. Parcial: la op. no está definida si a es vacío}
    parcial altura: arbin a -> natural
    {Devuelve la altura de a. Parcial: la operación no está definida si a es vacío}
  fspespec
```

Índice

- 1 Concepto
- 2 Especificación
- 3 Recorridos**
- 4 Implementación estática
- 5 Implementación dinámica

Árboles binarios

Recorridos en profundidad

- Consiste en visitar todos los elementos del árbol una sola vez

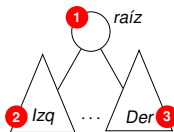
Árboles binarios

Recorridos en profundidad

- Consiste en visitar todos los elementos del árbol una sola vez

- Recorrido en pre-orden

- 1 Se visita la raíz
- 2 Se recorre (en pre-orden) el hijo izquierdo
- 3 Se recorre (en pre-orden) el hijo derecho



Árboles binarios

Recorridos en profundidad

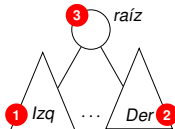
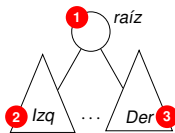
- Consiste en visitar todos los elementos del árbol una sola vez

■ Recorrido en pre-orden

- 1 Se visita la raíz
- 2 Se recorre (en pre-orden) el hijo izquierdo
- 3 Se recorre (en pre-orden) el hijo derecho

■ Recorrido en post-orden

- 1 Se recorre (en post-orden) el hijo izquierdo
- 2 Se recorre (en post-orden) el hijo derecho
- 3 Se visita la raíz



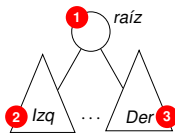
Árboles binarios

Recorridos en profundidad

- Consiste en visitar todos los elementos del árbol una sola vez

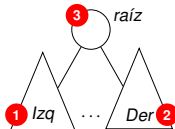
■ Recorrido en pre-orden

- 1 Se visita la raíz
- 2 Se recorre (en pre-orden) el hijo izquierdo
- 3 Se recorre (en pre-orden) el hijo derecho



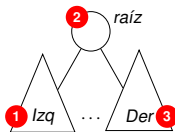
■ Recorrido en post-orden

- 1 Se recorre (en post-orden) el hijo izquierdo
- 2 Se recorre (en post-orden) el hijo derecho
- 3 Se visita la raíz



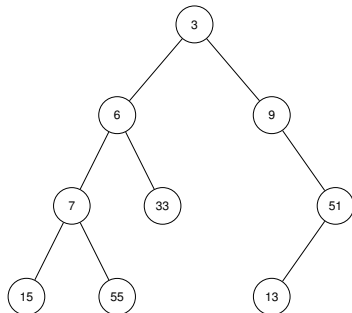
■ Recorrido en in-orden

- 1 Se recorre (en in-orden) el hijo izquierdo
- 2 Se visita la raíz
- 3 Se recorre (en in-orden) el hijo derecho



Árboles binarios

Recorridos en profundidad – ejemplo

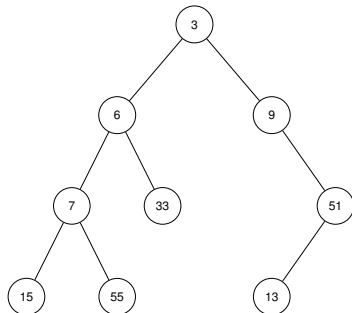


¿Cuál será la secuencia de números que se obtendrá al recorrer el árbol utilizando ...

- 1 recorrido en pre-orden?
- 2 recorrido en post-orden?
- 3 recorrido en *in*-orden?

Árboles binarios

Recorridos en profundidad – ejemplo

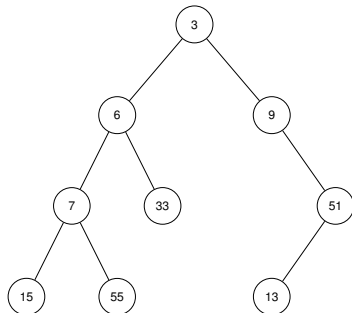


¿Cuál será la secuencia de números que se obtendrá al recorrer el árbol utilizando ...

- 1 recorrido en pre-orden? Respuesta: 3 → 6 → 7 → 15 → 55 → 33 → 9 → 51 → 13
- 2 recorrido en post-orden?
- 3 recorrido en in-orden?

Árboles binarios

Recorridos en profundidad – ejemplo

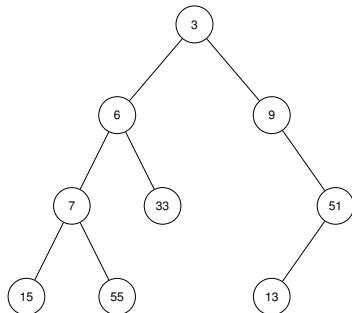


¿Cuál será la secuencia de números que se obtendrá al recorrer el árbol utilizando ...

- 1 recorrido en pre-orden? Respuesta: 3 → 6 → 7 → 15 → 55 → 33 → 9 → 51 → 13
- 2 recorrido en post-orden? Respuesta: 15 → 55 → 7 → 33 → 6 → 13 → 51 → 9 → 3
- 3 recorrido en *in*-orden?

Árboles binarios

Recorridos en profundidad – ejemplo

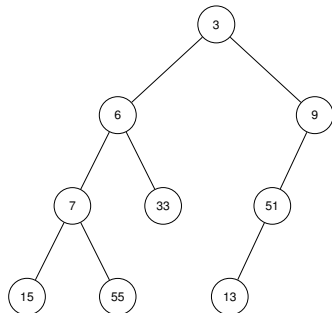


¿Cuál será la secuencia de números que se obtendrá al recorrer el árbol utilizando ...

- 1 recorrido en pre-orden? Respuesta: 3 → 6 → 7 → 15 → 55 → 33 → 9 → 51 → 13
- 2 recorrido en post-orden? Respuesta: 15 → 55 → 7 → 33 → 6 → 13 → 51 → 9 → 3
- 3 recorrido en in-orden? Respuesta: 15 → 7 → 55 → 6 → 33 → 3 → 9 → 13 → 51

Árboles binarios

Recorridos en profundidad – ejemplo

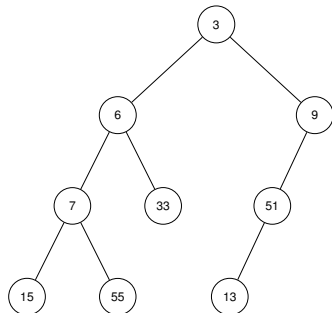


¿Cuál será la secuencia de números que se obtendrá al recorrer el árbol utilizando ...

- 1 recorrido en pre-orden?
- 2 recorrido en post-orden?
- 3 recorrido en in-orden?

Árboles binarios

Recorridos en profundidad – ejemplo

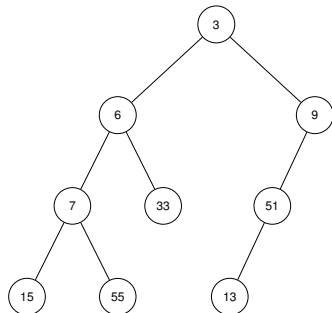


¿Cuál será la secuencia de números que se obtendrá al recorrer el árbol utilizando ...

- 1 recorrido en pre-orden? Respuesta: 3 → 6 → 7 → 15 → 55 → 33 → 9 → 51 → 13
- 2 recorrido en post-orden?
- 3 recorrido en in-orden?

Árboles binarios

Recorridos en profundidad – ejemplo

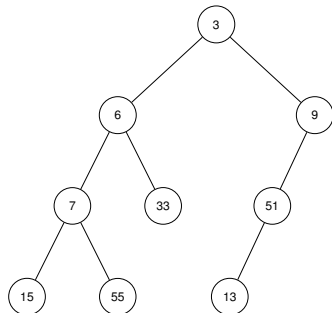


¿Cuál será la secuencia de números que se obtendrá al recorrer el árbol utilizando ...

- 1 recorrido en pre-orden? Respuesta: 3 → 6 → 7 → 15 → 55 → 33 → 9 → 51 → 13
- 2 recorrido en post-orden? Respuesta: 15 → 55 → 7 → 33 → 6 → 13 → 51 → 9 → 3
- 3 recorrido en *in*-orden?

Árboles binarios

Recorridos en profundidad – ejemplo



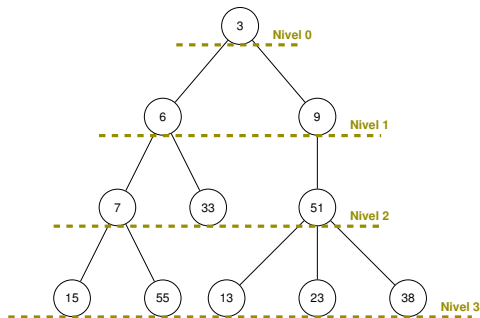
¿Cuál será la secuencia de números que se obtendrá al recorrer el árbol utilizando ...

- 1 recorrido en pre-orden? Respuesta: 3 → 6 → 7 → 15 → 55 → 33 → 9 → 51 → 13
- 2 recorrido en post-orden? Respuesta: 15 → 55 → 7 → 33 → 6 → 13 → 51 → 9 → 3
- 3 recorrido en in-orden? Respuesta: 15 → 7 → 55 → 6 → 33 → 3 → 13 → 51 → 9

Árboles binarios

Recorrido en anchura

- Visitar todos los elementos del árbol una sola vez, de la forma que:
 - 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



Índice

- 1 Concepto
- 2 Especificación
- 3 Recorridos
- 4 Implementación estática**
- 5 Implementación dinámica

Árboles binarios

Implementación estática

Representación basada en cursores a los hijos

- En base a vectores [1..max]
 - Cada componente del vector guarda un nodo con:
 - El *elemento* que contiene
 - Los índices del vector (*cursores*) donde se encuentran sus hijos *izq* y *der*
 - Un booleano, para indicar si la componente del vector está en uso o no
 - El árbol será el índice (*cursor*) donde se encuentra la raíz del árbol y el vector que almacena los datos
 - **Árbol vacío** se representará con valor 0 como índice de la raíz (*decisión de implementación*)

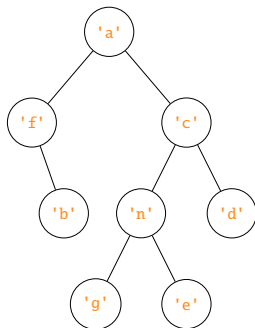
Árboles binarios

Implementación estática

```
constante max = ... {máximo número de elementos almacenables}
tipos
  arbin = 0..max; {el 0 significa árbol vacío}
  nodo = registro
    dato: elemento;
    izq, der: arbin;
    ocupado: booleano
  freg;
  tpVectorDeNodos = vector[1..max] de nodo;
variables
  a: arbin;
  vector_nodos: tpVectorDeNodos
  { un mismo vector puede incluso almacenar varios árboles }
```

Árboles binarios

Implementación estática – ejemplo



a 1

vector_nodos

1	'a'	2	4	v
2	'f'	0	3	v
3	'b'	0	0	v
4	'c'	5	8	v
5	'n'	6	7	v
6	'g'	0	0	v
7	'e'	0	0	v
8	'd'	0	0	v
9	?	?	?	f
10	?	?	?	f

Índice

- 1 Concepto
- 2 Especificación
- 3 Recorridos
- 4 Implementación estática
- 5 Implementación dinámica**

Árboles binarios

Implementación dinámica

Encadenamiento mediante punteros (a los hijos)

tipos

```
arbin = ↑nodo;  
nodo = registro  
        dato: elemento;  
        izq, der: arbin;  
freg;
```

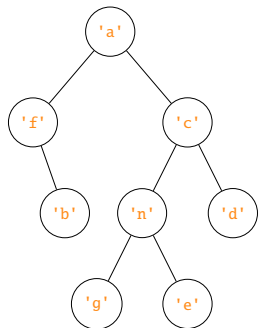
Árboles binarios

Implementación dinámica

Encadenamiento mediante punteros (a los hijos)

tipos

```
arbin = ↑nodo;  
nodo = registro  
    dato: elemento;  
    izq, der: arbin;  
freg;
```



¿Cómo quedará representado el árbol de la izquierda?

Árboles binarios

Implementación dinámica

módulo árbolesBinarios

parámetros

tipo elemento

exporta

tipo arbin

procedimiento vacío(sal a: arbin)

procedimiento plantar(sal a: arbin; ent e: elemento; ent ai, ad: arbin)

función esVacio?(a: arbin) devuelve booleano

procedimiento raíz(ent a: arbin; sal error: booleano; sal e: elemento)

{Si esVacio?(a) devuelve error=verdad. Si no, devuelve error=falso y en e el elemento raíz de a}

procedimiento subIzq(ent a: arbin; sal error: booleano; sal ai: arbin)

{Si esVacio?(a) devuelve error=verdad. Si no, devuelve error=falso y en ai el subárbol izquierdo de a}

procedimiento subDer(ent a: arbin; sal error: booleano; sal ad: arbin)

{Si esVacio?(a) devuelve error=verdad. Si no, devuelve error=falso y en ad el subárbol derecho de a}

procedimiento altura(ent a: arbin; sal error: booleano; sal h: natural)

{Si esVacio?(a) devuelve error=verdad. Si no, devuelve error=falso y en h la altura de a}

procedimiento duplicar(sal nuevo: arbin; ent viejo: arbin)

{Duplica la representación del árbol viejo guardándolo en nuevo.}

procedimiento liberar(e/s a: arbin)

{Libera la memoria dinámica accesible desde a, quedando a vacío.}

función iguales(a1, a2: arbin) devuelve booleano

{Devuelve verdad si y sólo si a1 y a2 tienen los mismos elementos y en las mismas posiciones.}

implementación

```
tipos
  arbin = ↑nodo;
  nodo  = registro
          dato: elemento;
          izq, der: arbin;
          freg;

procedimiento vacío(sal a: arbin)
principio
  a := nil
fin

procedimiento plantar(sal a: arbin; ent e: elemento; ent ai, ad: arbin)
principio
  nuevoDato(a);
  a↑.dato := e;
  a↑.izq := ai;
  a↑.der := ad
fin

función esVacio(a: arbin) devuelve booleano
principio
  devuelve a = nil
fin

procedimiento raíz(ent a: arbin; sal error: booleano; sal e: elemento)
principio
  si esVacio(a) entonces
    error := verdad
  sino
    error := falso;
    e := a↑.dato
  fsi
fin
```

...

```

...
procedimiento subIzq(ent a: arbin; sal error: booleano; sal ai: arbin)
principio
  si esVacio(a) entonces
    error := verdad
  sino
    error := falso;
    ai := a↑.izq { ¡Ojo! no crea copia del subárbol, simplemente devuelve puntero al subárbol}
  fsi
fin
procedimiento subDer(ent a: arbin; sal error: booleano; sal ad: arbin)
{... análogo al anterior ...}

procedimiento altura(ent a: arbin; sal error: booleano; sal h: natural)
principio
  si esVacio(a) entonces
    error := verdad
  sino
    error := falso;
    h := alturaRec(a)
  fsi
fin
función max(a, b: entero) devuelve entero {función máximo de dos enteros}
principio
  si a ≥ b entonces devuelve a sino devuelve b fsi
fin
función alturaRec(a: arbin) devuelve natural
{PRECONDICIÓN: a es no vacío}
principio
  selección
    (a↑.izq = nil) and (a↑.der = nil): devuelve 0;
    (a↑.izq = nil) and (a↑.der ≠ nil): devuelve 1 + alturaRec(a↑.der);
    (a↑.izq ≠ nil) and (a↑.der = nil): devuelve 1 + alturaRec(a↑.izq);
    (a↑.izq ≠ nil) and (a↑.der ≠ nil): devuelve 1 + max(alturaRec(a↑.izq), alturaRec(a↑.der))
  fselección
fin
...

```

```

...
procedimiento duplicar(sal nuevo: arbin; ent viejo: arbin)
variables ai,ad: arbin
principio
  si viejo = nil entonces
    nuevo := nil
  sino
    nuevoDato(nuevo);
    nuevo↑.dato := viejo↑.dato;
    duplicar(ai, viejo↑.izq);
    duplicar(ad, viejo↑.der);
    nuevo↑.izq := ai;
    nuevo↑.der := ad
  fsi
fin
procedimiento liberar(e/s a: arbin)
principio
  si a ≠ nil entonces
    liberar(a↑.izq);
    liberar(a↑.der);
    disponer(a);
    a:=nil
  fsi
fin

función iguales(a1, a2: arbin) devuelve booleano
principio
  si a1 = nil entonces
    devuelve a2 = nil
  sino_si a2 = nil entonces
    devuelve falso
  sino {a1 y a2 son no nulos}
    devuelve a1↑.dato = a2↑.dato and
      iguales(a1↑.izq, a2↑.izq) and
      iguales(a1↑.der, a2↑.der)
  fsi
fin
fin {fin del modulo árbolesBinarios}

```

Árboles binarios

Implementación dinámica

■ Coste temporal de las operaciones de la especificación: $\Theta(1)$

■ **Todas**, menos altura

```
procedimiento altura(ent a: arbin; sal error: booleano; sal h: natural)
principio
  si esVacio(a) entonces
    error := verdad
  sino
    error := falso;
    h := alturaRec(a)
  fsi
fin
función max(a, b: entero) devuelve entero {función máximo de dos enteros}
principio
  si a ≥ b entonces devuelve a sino devuelve b fsi
fin
función alturaRec(a: arbin) devuelve natural
{PRECONDICIÓN: a es no vacío}
principio
  selección
    (a↑.izq = nil) and (a↑.der = nil): devuelve 0;
    (a↑.izq = nil) and (a↑.der ≠ nil): devuelve 1 + alturaRec(a↑.der);
    (a↑.izq ≠ nil) and (a↑.der = nil): devuelve 1 + alturaRec(a↑.izq);
    (a↑.izq ≠ nil) and (a↑.der ≠ nil): devuelve 1 + max(alturaRec(a↑.izq), alturaRec(a↑.der))
  fselección
fin
```


Árboles binarios

Implementación dinámica

- Coste temporal de las operaciones de la especificación: $\Theta(1)$
 - **Todas**, menos altura

¿Qué podemos hacer para que altura también quede de coste $\Theta(1)$?

Árboles binarios

Implementación dinámica

- Coste temporal de las operaciones de la especificación: $\Theta(1)$
 - **Todas**, menos altura

¿Qué podemos hacer para que altura también quede de coste $\Theta(1)$?

```
tipos
ptNodo = ↑nodo;
nodo = registro
    dato: elemento;
    izq, der: arbin;
freg;
arbin = registro
    altura: entero; {**}
    laRaíz: ptNodo
freg
```

Árboles binarios

Implementación dinámica

- Coste temporal de las operaciones de la especificación: $\Theta(1)$

- **Todas**, menos altura

```
tipos
ptNodo = ↑nodo;
nodo = registro
        dato: elemento;
        izq, der: arbin;
arbin = registro
        freg;
        altura: entero; {**}
        laRaíz: ptNodo
        freg
```

Árboles binarios

Implementación dinámica

■ Coste temporal de las operaciones de la especificación: $\Theta(1)$

■ **Todas**, menos altura

```
tipos
ptNodo = ↑nodo;
nodo = registro
    dato: elemento;
    izq, der: arbin;
freg;
arbin = registro
    altura: entero; {**}
    laRaíz: ptNodo
freg
```

```
tipos
arbin = ↑nodo;
nodo = registro
    altura: entero; {**}
    dato: elemento;
    izq, der: arbin;
freg;
{¿qué habrá que hacer para mantenerla calculada?}
```

Árboles binarios

Implementación dinámica – recorridos

módulo recorridosArbin

importa árbolesBinarios, listasGenéricas

exporta

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

{añade a la lista l la secuencia de elementos resultante de recorrer en pre-orden el árbol a, es decir, lo la raíz, luego el subárbol izquierdo y después el derecho, ambos en pre-orden}

procedimiento inOrden(ent a: arbin; e/s l: lista)

{añade a la lista l la secuencia de elementos resultante de recorrer en in-orden el árbol a, es decir, lo el subárbol izquierdo en in-orden, luego la raíz y después el subárbol derecho en in-orden}

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

{añade a la lista l la secuencia de elementos resultante de recorrer en post-orden el árbol a, es decir, lo el subárbol izquierdo en post-orden, luego el derecho en post-orden y finalmente la raíz}

implementación

Árboles binarios

Implementación dinámica – recorridos

```
procedimiento preOrden(ent a: arbin; e/s l: lista)
variables ai, ad: arbin; r: elemento; error: booleano
principio
    si not esVacio(a) entonces
        raíz(a, error, r);
        añadirÚltimo(l, r);
        subIzq(a, error, ai)
        preOrden(ai, l);
        subDer(a, error, ad);
        preOrden(ad, l)
    fsi
fin
```

Árboles binarios

Implementación dinámica – recorridos

```
procedimiento preOrden(ent a: arbin; e/s l: lista)
variables ai, ad: arbin; r: elemento; error: booleano
principio
  si not esVacio(a) entonces
    raíz(a, error, r);
    añadirÚltimo(l, r);
    subIzq(a, error, ai)
    preOrden(ai, l);
    subDer(a, error, ad);
    preOrden(ad, l)
  fsi
fin
```

} si $a \neq \text{nil}$ entonces
añadirÚltimo(l, $a \uparrow \text{dato}$);
preOrden($a \uparrow \text{izq}$, l);
preOrden($a \uparrow \text{der}$, l)

fsi

*A más bajo nivel, si tenemos acceso
a la representación del tipo arbin*

Árboles binarios

Implementación dinámica – recorridos

```
procedimiento inOrden (ent a: arbin; e/s l:lista)
variables ai, ad: arbin; r: elemento; error: booleano
principio
    si not esVacio(a) entonces
        subIzq(a, error, ai)
        inOrden(ai, l);
        raíz(a, error, r);
        añadirÚltimo(l, r);
        subDer(a, error, ad);
        inOrden(ad, l)
    fsi
fin
```


Árboles binarios

Implementación dinámica – recorridos

```
procedimiento inOrden (ent a: arbin; e/s l:lista)
variables ai, ad: arbin; r: elemento; error: booleano
principio
    si not esVacio(a) entonces
        subIzq(a, error, ai)
        inOrden(ai, l);
        raíz(a, error, r);
        añadirÚltimo(l, r);
        subDer(a, error, ad);
        inOrden(ad, l)
    fsi
fin
```

} si a ≠ nil entonces
inOrden(a↑.izq, l);
añadirÚltimo(l, a↑.dato);
inOrden(a↑.der, l)
fsi

A más bajo nivel, si tenemos acceso a la representación del tipo arbin

Árboles binarios

Implementación dinámica – recorridos

```
procedimiento postOrden (ent a: arbin; e/s l:lista)
variables ai, ad: arbin; r: elemento; error: booleano
principio
    si not esVacio(a) entonces
        subIzq(a, error, ai)
        postOrden(ai, l);
        subDer(a, error, ad);
        postOrden(ad, l);
        raíz(a, error, r);
        añadirÚltimo(l, r)
    fsi
fin
```

Árboles binarios

Implementación dinámica – recorridos

```
procedimiento postOrden (ent a: arbin; e/s l:lista)
variables ai, ad: arbin; r: elemento; error: booleano
principio
    si not esVacio(a) entonces
        subIzq(a, error, ai)
        postOrden(ai, l);
        subDer(a, error, ad);
        postOrden(ad, l);
        raíz(a, error, r);
        añadirÚltimo(l, r)
    fsi
fin
```

} si a ≠ nil entonces
postOrden(a↑.izq, l);
postOrden(a↑.der, l);
añadirÚltimo(l, a↑.dato);
fsi

*A más bajo nivel, si tenemos acceso
a la representación del tipo arbin*

Árboles binarios

Implementación dinámica – recorridos

```
{Nota: esta implementación va a mostrar la lista escrita en pantalla en vez de devolverla}
procedimiento nivel(ent a: arbin; ent i: 0..maxEntero)
{Pre: a es no vacío}
{Post: escribe en pantalla los elementos del nivel i de a, de izquierda a derecha}
variables r: elemento; error: booleano; ai, ad: arbin
principio
  si i = 0 entonces
    raíz(a, error, r);
    escribir(r)
  sino {i>0}
    subIzq(a, error, ai);
    subDer(a, error, ad);
    selección
      esVacio(ai) and esVacio(ad): {no hacer nada};
      esVacio(ai) and not esVacio(ad): nivel(ad, i - 1);
      not esVacio(ai) and esVacio(ad): nivel(ai, i - 1);
      not esVacio(ai) and not esVacio(ad): nivel(ai, i - 1); nivel(ad, i - 1)
    fselección
  fsi
fin
```

Árboles binarios

Implementación dinámica – recorridos

```
procedimiento anchura(ent a: arbin)
{Escribe en pantalla los elementos de a recorridos en anchura, es decir
  por niveles desde el 0 hasta el último y, para cada nivel, de
  izquierda a derecha}
variables h, i: entero
principio
  si not esVacio(a) entonces
    altura(a, error, h);
    para i := 0 hasta h hacer
      nivel(a, i)
  fpara
fsi
fin
```

Implementación bastante ineficiente.

¿Cómo podríamos tener una versión iterativa?

Árboles binarios

Implementación dinámica – recorridos

```
{Recorrido en preorden}
procedimiento preorden_pila(ent a: arbin)
variables p: pila { de arbin }; aux: arbin
principio
  crear(p);
  apilar(p, a);
  mientrasQue not esVacía(p) hacer
    aux := cima(p);
    desapilar(p);
    si aux ≠ nil entonces
      escribir(aux↑.dato);
      apilar(p, aux↑.der);
      apilar(p, aux↑.izq)
    fsi
  fmq
fin
```

```
{Recorrido en anchura}
procedimiento anchuraCola(ent a: arbin)
variables c: cola {de arbin}; aux: arbin
principio
  crear(c);
  encolar(c, a);
  mientrasQue not esVacía(c) hacer
    aux := primero(c);
    desencolar(c);
    si aux ≠ nil entonces
      escribir(aux↑.dato);
      encolar(c, aux↑.izq);
      encolar(c, aux↑.der)
    fsi
  fmq
fin
```

Árboles binarios

Implementación dinámica – recorridos

procedimiento inorden(**ent** a: arbin)

{versión iterativa del recorrido in-orden usando una pila auxiliar}

variables p: pila {de arbin}; aux: arbin

principio

{equivale al inicio de un hipotético iterador}

crear(p);

aux := a;

mientrasQue aux ≠ nil **hacer**

 apilar(p, aux);

 aux := aux↑.izq

fmq;

{not(esVacía(p)) es equivalente a 'existeSiguiente' de un iterador}

mientrasQue not esVacía(p) **hacer**

{equivale al código de 'siguiente' de un iterador}

 aux := cima(p);

 desapilar(p);

 escribir(aux↑.dato);

 aux := aux↑.der;

mientrasQue aux ≠ nil **hacer**

 apilar(p, aux);

 aux := aux↑.izq

fmq

fmq

fin

Árboles binarios

Implementación dinámica – recorridos

```
tipo árbol = registro {aumentamos la representación de arbin con pila auxiliar para el iterador}
    raíz: arbin; {puntero a la raíz de un árbol binario}
    iter: pila_de_arbin {pila de punteros a nodos del árbol, para el iterador}
freg
```

```
procedimiento iniciarIterador(e/s a: árbol)
```

```
variable aux: arbin
```

```
principio
```

```
    crear(a.iter); {crea pila vacía de punteros a nodos del árbol}
```

```
    aux := a.raíz; {raíz del árbol}
```

```
    mientrasQue aux ≠ nil hacer
```

```
        apilar(a.iter, aux); {apila el puntero aux (a un nodo del árbol) en la pila del iterador}
```

```
        aux := aux↑.izq
```

```
    fmq
```

```
fin
```

```
función existeSiguiente(a: árbol) devuelve booleano
```

```
principio
```

```
    devuelve not esVacía(a.iter) {hay siguiente si la pila del iterador es no vacía}
```

```
fin
```

```
procedimiento siguiente(e/s a: árbol; sal unDato: elemento; sal error: booleano)
```

```
variable aux: arbin
```

```
principio
```

```
    si existeSiguiente(a) entonces
```

```
        error := falso;
```

```
        aux := cima(a.iter);
```

```
        desapilar(a.iter);
```

```
        unDato := aux↑.dato; {este es el siguiente elemento visitado}
```

```
        aux := aux↑.der;
```

```
        mientrasQue aux ≠ nil hacer
```

```
            apilar(a.iter, aux);
```

```
            aux := aux↑.izq
```

```
        fmq
```

```
    sino
```

```
        error := verdad
```

```
    fsi
```

```
fin
```


Estructuras de Datos y Algoritmos

Árboles binarios

LECCIÓN 12

© All wrongs reversed – bajo licencia CC-BY-NC-SA 4.0



Universidad
Zaragoza

Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza, España

Curso 2024/2025

Grado en Ingeniería Informática

UNIVERSIDAD DE ZARAGOZA

Aula 0.04, Edificio Agustín de Betancourt

