

Árboles binarios

Definición de funciones en árboles

En general, la forma más simple de definir una función sobre un árbol (n -ario o binario) es **trasladar la definición recurrente (o recursiva) del dominio a la definición de ésta.**

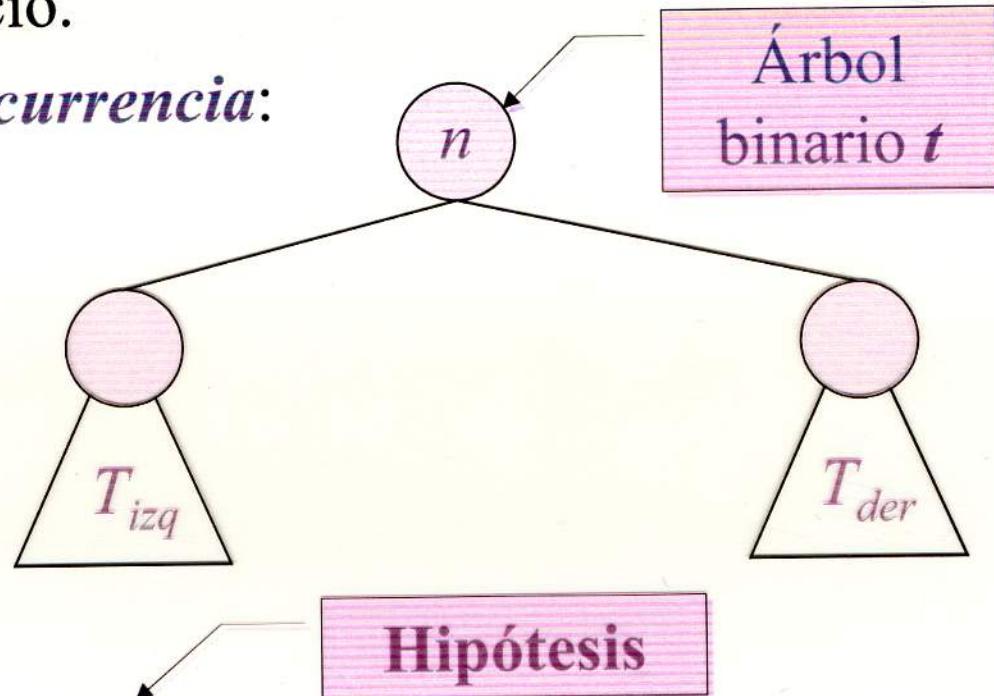
Esto no quiere decir que toda función que se defina sobre un árbol deba ser recursiva. Así, podemos encontrarnos con problemas cuya solución exija escribir funciones iterativas, dado que no es posible encontrar una función recursiva (por extensión de la definición recurrente del dominio) que lo solucione, como ocurre, por ejemplo, en el recorrido por niveles de un árbol.

Función $f(t)$ sobre un árbol binario t

Definición por extensión de la definición recurrente del conjunto de árboles binarios.

Base: Valor de la función si t es el árbol vacío.

Recurrencia:



Se **supone conocida** la función para cada uno de los subárboles binarios, T_{izq} y T_{der} , de t .

Se **calcula** el valor final de la función supuestos conocidos los valores anteriores.

Ejemplo: árboles binarios iguales

Definición de la función (igual)

Base: Si t_1 y t_2 son árboles binarios vacíos entonces son iguales.

Recurrencia: Hipótesis

- $\text{igual}(tizq}_1, tizq_2)$
- $\text{igual}(tder}_1, tder_2)$

donde $tizq_i$ es el subárbol izquierdo de t_i y $tder_i$ es el subárbol derecho de t_i .

Tesis: Los árboles binarios t_1 y t_2 serán iguales si se cumplen las condiciones

- $t_1.\text{label}()=t_2.\text{label}()$
- $\text{igual}(tizq}_1, tizq_2)$ e
 $\text{igual}(tder}_1, tder_2)$

Ejemplo: altura de árboles binarios

Definición de la función (altura)

Base: Si t es un árbol binario vacío entonces su altura es 0.

Recurrencia:

Hipótesis:

Conocidos

- $\text{altura } (t_{izq}) = a_{izq}$
- $\text{altura } (t_{der}) = a_{der}$

donde t_{izq} es el subárbol izquierdo de t y t_{der} es el subárbol derecho de t .

Tesis: La altura de t se calculará:

- $1 + \max(a_{izq}, a_{der})$

- Contar el número de nodos
- Calcular el grado de un árbol binario

Ejemplo: árboles binarios isomorfos

Definición de la función (iso)

Base: Si t_1 y t_2 son árboles binarios vacíos entonces son isomorfos.

Recurrencia: Hipótesis

- $\text{iso}(\text{tizq}_1, \text{tizq}_2)$ • $\text{iso}(\text{tizq}_1, \text{tder}_2)$
- $\text{iso}(\text{tder}_1, \text{tder}_2)$ • $\text{iso}(\text{tder}_1, \text{tizq}_2)$

donde tizq_i es el subárbol izquierdo de t_i y tder_i es el subárbol derecho de t_i .

Tesis: Los árboles binarios t_1 y t_2 serán isomorfos si se cumplen las condiciones

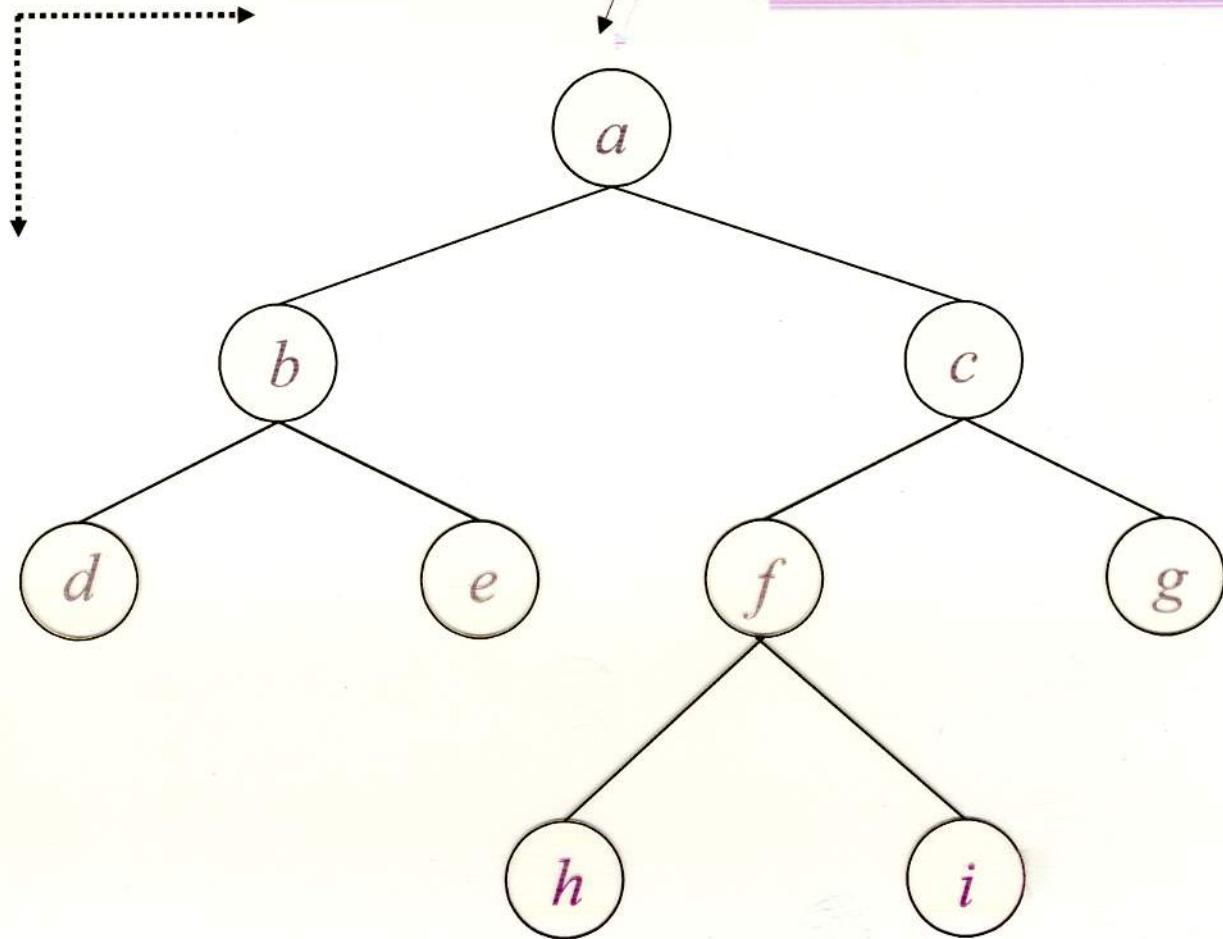
- $t_1.\text{label}() = t_2.\text{label}()$
- $\text{iso}(\text{tizq}_1, \text{tizq}_2)$ y $\text{iso}(\text{tder}_1, \text{tder}_2)$
o bien
 $\text{iso}(\text{tizq}_1, \text{tder}_2)$ y $\text{iso}(\text{tder}_1, \text{tizq}_2)$

Representación mediante vectores

Las etiquetas de los nodos se almacenan en un vector. Los nodos se enumeran de la forma siguiente:

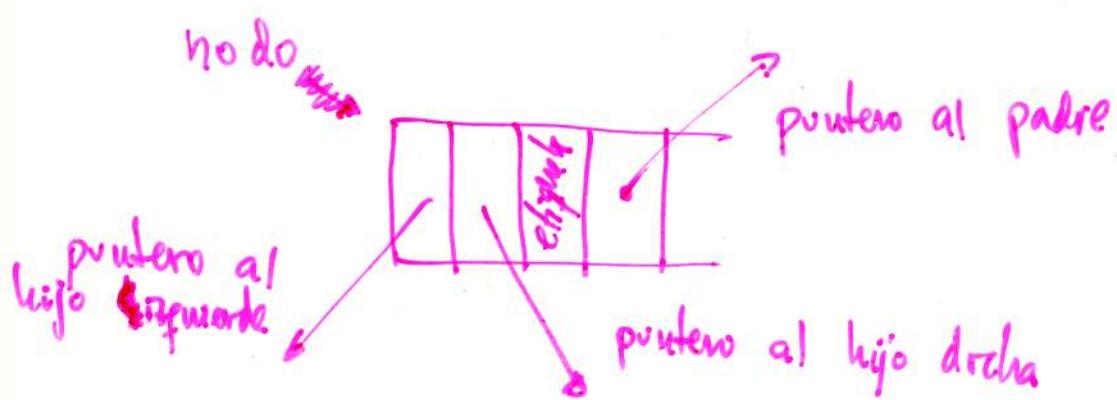
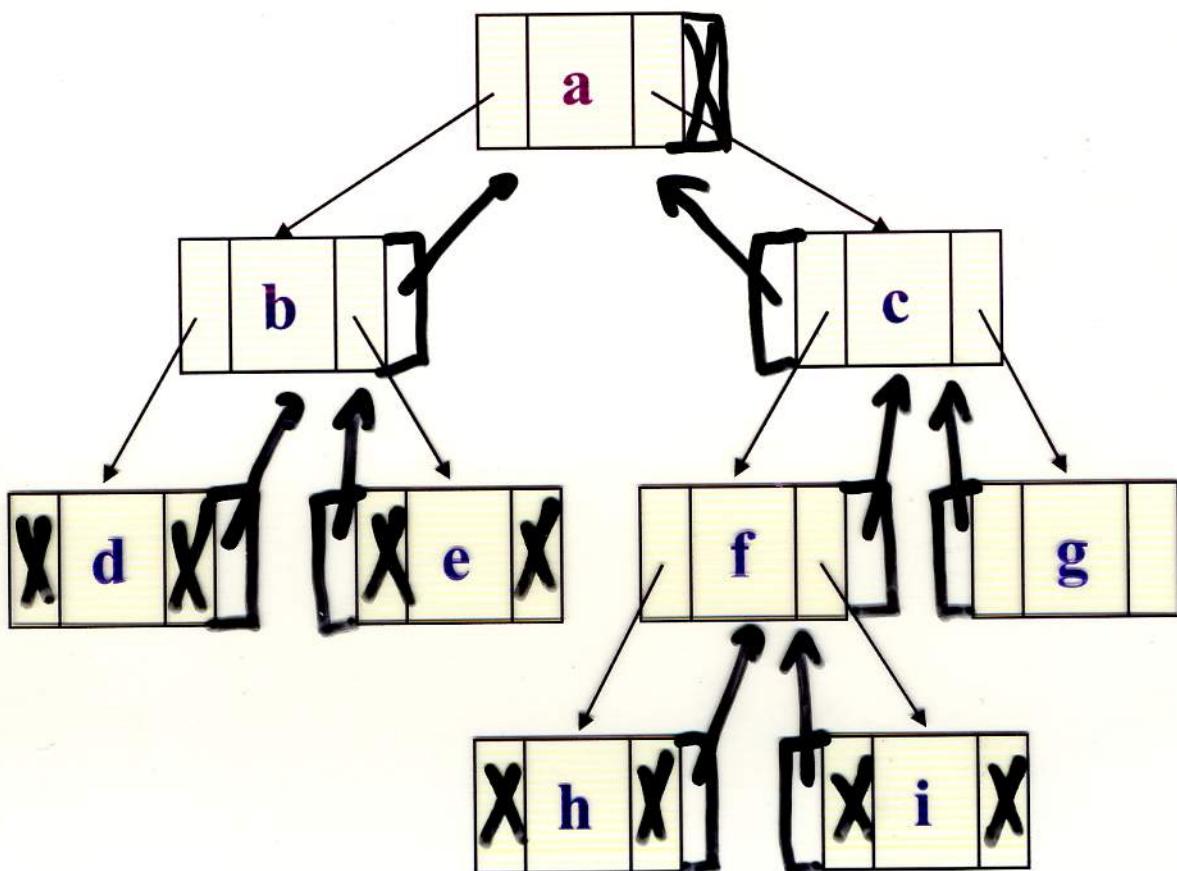
- A la raíz le corresponde el índice **0**.
- Si a un nodo le corresponde el índice **k**:
 - Su hijo izquierdo, si tiene, está en la posición **$2*(k+1)-1 \rightarrow 2*k+1$**
 - Su hijo derecho, si tiene, está en la posición **$2*(k+1) \rightarrow 2*k+2$**
 - Su padre, si tiene, está en la posición **$(k-1) / 2$**

Numeración de los nodos por niveles



| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|-----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | ... |
| a | b | c | d | e | f | g | | | | | | | h | i | |

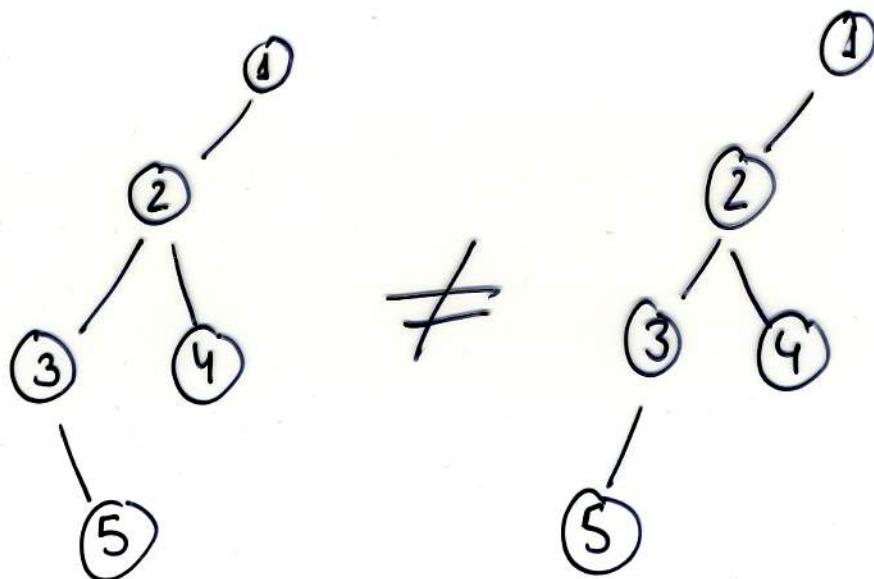
Representación mediante nodos enlazados



TDA ARBOL BINARIO

Definición

Un **árbol binario** es un **árbol** que o es vacío o en cada nodo puede tener a lo más 2 hijos, denominados hijo a la **izquierda** e hijo a la **derecha**, permitiendo un nodo tener un hijo a la derecha y no tener un hijo a la izquierda o viceversa.



```
/* Fichero: ArbolBinario.h */
```

```
/**
```

```
@memo T.D.A. ArbolBinario
```

@doc Definición: Una instancia a del tipo de dato abstracto *ArbolBinario* sobre un dominio T_{base} se puede construir como

- Un objeto vacío (árbol vacío) si no contiene ningún elemento. Lo denotamos $\{\}$
- Un árbol que contiene un elemento destacado, el nodo raíz, con un valor e en el dominio T_{base} (denominado *etiqueta*), y dos subárboles (T_i :izquierdo y T_d derecho) del T.D.A. *ArbolBinario* sobre T_{base} . Se establece una relación *padre-hijo* entre cada nodo y los nodos raíz de los subárboles (si

los hubiera) que cuelgan de él. Lo denotamos $\{e, \{T_i\}, \{T_d\}\}$.

Para poder usar el tipo de dato *ArbolBinario* se debe incluir el fichero

ArbolBinario.h

El espacio requerido para el almacenamiento es $O(n)$. Donde n es el número de nodos del árbol.

@author

@version 1.0

*/

```
template < class Tbase >
class ArbolBinario {
private:
/** @name Implementación de T.D.A. Arbol-
Binario
@memo Parte privada. */

/** 
@memo Nodo
@doc En cada estructura nodo se almacena
una etiqueta del árbol, que se implementa como
un conjunto de nodos enlazados según la relación
padre-hijo */

struct nodo {
Tbase etiqueta;
/** 
@memo Elemento almacenado
@doc En este campo se almacena la etiqueta
que corresponde a este nodo */
```

```
struct nodo *izqda;  
/**  
@memo Puntero al hijo izquierdo  
@doc En este campo se almacena un puntero  
al nodo raíz del subárbol izquierdo, o el valor  
0 si no tiene. */  
struct nodo *drcha;  
/**  
@memo Puntero al hijo derecho  
@doc En este campo se almacena un puntero  
al nodo raíz del subárbol derecho, o el valor 0  
si no tiene */  
struct nodo *padre;  
/**  
@memo Puntero al padre  
@doc En este campo se almacena un puntero  
al nodo padre, o el valor 0 si es la raiz */  
struct nodo *laraiz;  
/**  
@memo Puntero a la raiz.
```

```
template <class Tbase>
```

```
class Arbolbinario {
```

private:

```
struct nodo {
```

```
    Tbase etiqueta;
```

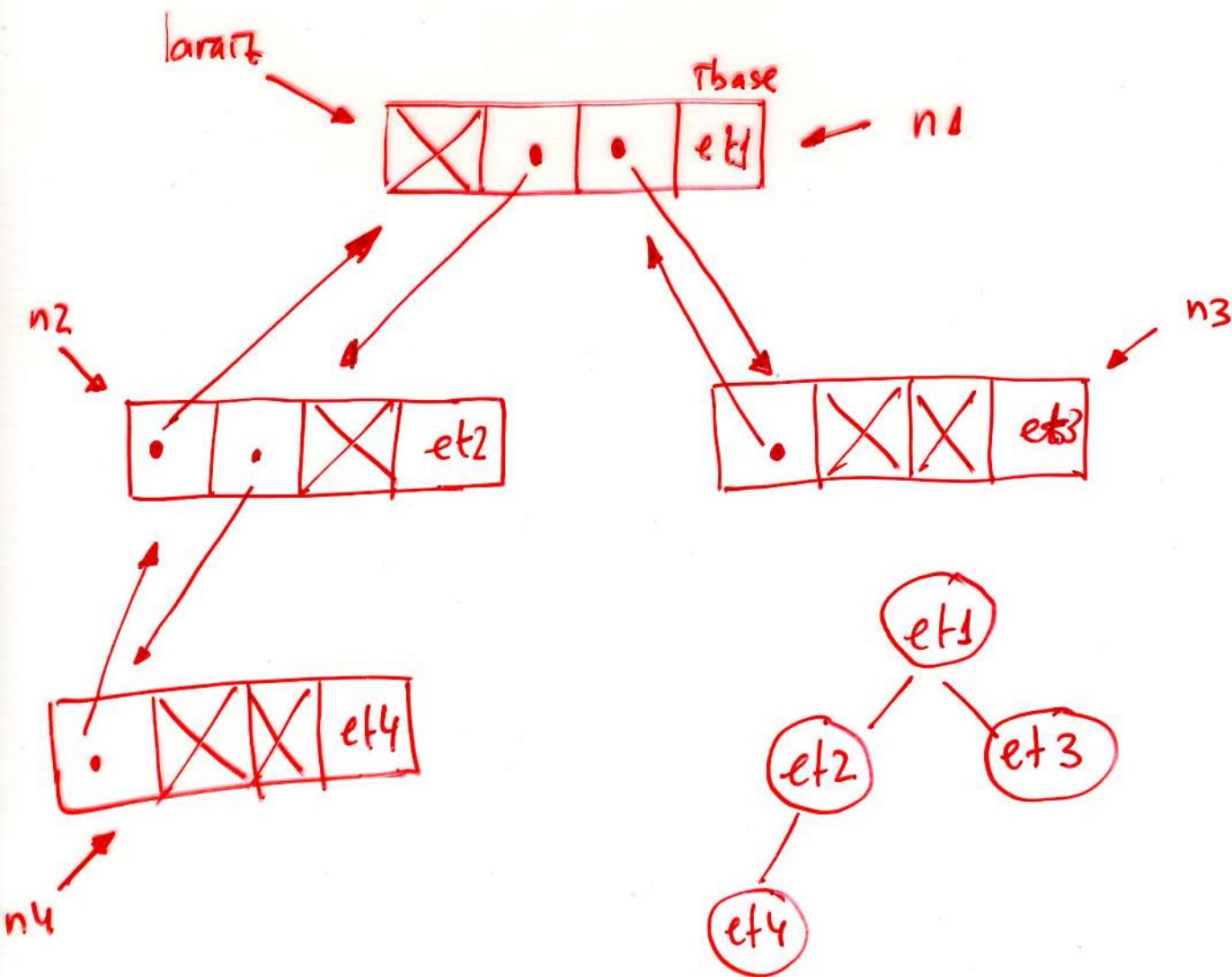
```
    struct nodo * izqda;
```

```
    struct nodo * drcha;
```

```
    struct nodo * padre;
```

```
};
```

```
struct nodo * lariz;
```



@doc Este miembro es un puntero al primer nodo, que corresponde a la raíz del árbol. Vale cero si el árbol es vacío */

/** **Invariante de la representación**
@memo Inv. de ArbolBinario

@doc

Sea T , un árbol binario sobre el tipo $Tbase$, entonces el invariante de la representación es

Si T es vacío, entonces $T.laraiz=0$, y si no:

$T.laraiz \rightarrow padre = 0$ Y

$\forall n$ nodo de T , $n \rightarrow izqda \neq n \rightarrow drcha$ Y

$\forall n, m$ nodos de T , si $n \rightarrow izqda = m$ o $n \rightarrow drcha = m$ entonces $m \rightarrow padre = n$

*/

/ @name Función de abstracción**

@memo F.A. de ArbolBinario.

@doc

Sea T un árbol binario sobre el tipo $Tbase$, entonces si lo denotamos también $rbol(T.laraiz)$, es decir, como el árbol que cuelga de su raíz, entonces éste árbol del conjunto de valores en la representación se aplica al árbol

{ $T.laraiz \rightarrow$ etiqueta, { $Arbol(T.laraiz \rightarrow izqda)$ },
{ $Arbol(T.laraiz \rightarrow drcha)$ } }.

donde {0} es el árbol vacío.

*/

private:

void destruir (nodo * n);

void copiar (nodo * & dest, nodo * orig);

int contar (nodo * n);

bool soniguales (nodo * n1, nodo * n2);

void escribe_arbol (ostream & out, nodo * nod) const;

void lee_arbol (istream & in, nodo * & nod);

public:

typedef struct nodo * Nodo;

//static const Nodo nodonulo = 0;

ArbolBinario();

ArbolBinario (const Tbase & e);

ArbolBinario (const ArbolBinario <Tbase> & v);

~ArbolBinario();

ArbolBinario <Tbase> & operator = (const ArbolBinario <Tbase> & v);

void AsignarRaiz (const Tbase & e);

Nodo raiz () const;

Nodo izquierda (const Nodo n) const;

Nodo derecha (const Nodo n) const;

Nodo padre (const Nodo n) const;

Tbase & etiqueta (const Nodo n);

const Tbase & etiqueta (const Nodo n) const;

```
void asignar_subarbol (const ArbolBinario<Tbase> &orig,
                      const Nodo nod);
```

```
void podar_izquierda (Nodo n, ArbolBinario<Tbase> &dest);
```

```
void podar_derecha (Nodo n, ArbolBinario<Tbase> &dest);
```

```
void insertar_izquierda (Nodo n, ArbolBinario<Tbase> &rama);
```

```
void insertar_derecha (Nodo n, ArbolBinario<Tbase> &rama);
```

```
void clear();
```

```
int size(); const;
```

```
bool empty(); const;
```

```
bool operator == (const ArbolBinario<Tbase> & v) const;
```

```
bool operator != (const ArbolBinario<Tbase> & v) const;
```

```
template <class T>
```

```
friend istream & operator >> (istream & in, ArbolBinario<T> & v);
```

```
template <class T>
```

```
friend ostream & operator << (ostream & out, const ArbolBinario<T>
                                  & v);
```

```
}
```

```
void destruir(nodo * n);
/** @memo Destruye el subárbol
@param n: nodo que destruir junto con sus
descendientes
@doc libera los recursos que ocupa n y sus
descendientes */
```

```
void copiar(nodo *& dest, nodo * orig);
/** @memo Copia un subárbol
@param dest: referencia al puntero del que
cuelga la copia
@param orig: puntero a la raiz del subárbol a
copiar
@doc Hace una copia de todo el subárbol que
cuelga de orig en el puntero dest. Es impor-
tante ver que en dest→padre (si existe) no se
asigna ningún valor, pues no se conoce */
```

~~int~~ contar(nodo * n);
/** @memo Cuenta el número de nodos
@param n: nodo del que cuelga el subárbol de

nodos a contabilizar.

*@doc Copia cuántos nodos cuelgan de n , incluído éste */*

bool soniguales(nodo * n1, nodo * n2);

/**

@memo Comprueba igualdad de subárbol

@param n1: Primer subárbol a comparar

@param n2: Segundo subárbol a comparar

*@doc Comprueba si son iguales los subárboles que cuelgan de $n1$ y $n2$. Para ello deberán tener los mismos nodos en las mismas posiciones y con las mismas etiquetas */*

~~std::~~

void escribe_arbol(ostream& out, nodo * nod)

const;

/**

@memo Escribe un subárbol

@param out: stream de salida donde escribir

@param nod: nodo del que cuelga el subárbol a escribir

@doc Escribe en la salida todos los nodos del subárbol que cuelga del nodo *nod* siguiendo un recorrido en preorden. La forma de impresión de cada nodo es:

Si el nodo es *nodo_nulo*, imprime el carácter 'x'

Si el nodo no es *nodo_nulo*, imprime el carácter 'n' seguido de un espacio, al que sigue la impresión de la etiqueta. */

```
void lee_arbol(istream& in, nodo *& nod);
/**  

 * @memo Lee un subárbol  

 * @param in: stream de entrada desde el que leer  

 * @param nod: referencia al nodo que contendrá el subárbol leído  

 * @doc Lee de la entrada in los elementos de un árbol según el formato que se presenta en la función de escritura. @see escribe_arbol */
```

public:

typedef struct nodo * Nodo;

/**

@memo Tipo Nodo

@doc Este tipo nos permite manejar cada uno de los nodos del árbol. Los valores que tomará serán tantos como nodos en el árbol (para poder referirse a cada uno de ellos) y además un valor destacado *nodo_nulo*, que indica que no se refiere a ninguno de ellos.

Una variable *n* de este tipo se declara

ArbolBinario::Nodo n;

Las operaciones válidas sobre el tipo nodo son:

- Operador de Asignación (=)
- Operador de comprobación de igualdad (==).
- Operador de comprobación de no igualdad (!=). */

ArbolBinario::nodo_nulo

@memo Nodo nulo

@doc El valor de nodo nulo se podrá indicar como */

static const Nodo **nodo_nulo** = 0; */

/**

@name Operaciones de T.D.A. Arbol Binario

@memo Operaciones sobre ArbolBinario

/

ArbolBinario();

/**

@memo Constructor por defecto

@doc Reserva los recursos e inicializa el árbol a vacío {}. La operación se realiza en tiempo O(1). */

```
ArbolBinario(const Tbase& e);  
/**  
 @memo Constructor de raiz  
 @doc Reserva los recursos e inicializa el árbol  
 con un único nodo raiz que tiene la etiqueta  
 e, es decir, el árbol {e,{},{}}. La operación se  
 realiza en tiempo O(1). */  
  
ArbolBinario (const ArbolBinario<Tbase>&  
 v);  
/**  
 @memo Constructor de copia  
 @param v: ArbolBinario a copiar  
 @doc construye el árbol duplicando el contenido  
 de v en el árbol receptor. La operación se re-  
 aliza en tiempo O(n), donde n es el número de  
 elementos de v. */
```

```
~ArbolBinario();
 $\text{/**}$ 
@memo Destructor
@doc Libera los recursos ocupados por el árbol receptor. La operación se realiza en tiempo  $O(n)$  donde  $n$  es el número de elementos del árbol receptor.  $*/$ 

ArbolBinario<Tbase>& operator=(const
ArbolBinario<Tbase> &v);
 $\text{/**}$ 
@memo Asignación
@param v: ArbolBinario a copiar
@return Referencia al árbol receptor.
@doc Asigna el valor del árbol duplicando el contenido de v en el árbol receptor. La operación se realiza en tiempo  $O(n)$ , donde  $n$  es el número de elementos de v.  $*/$ 
```

```
void AsignaRaiz(const Tbase& e);
 $\ast\ast$ 
@memo Asignar nodo raíz
@param e: etiqueta a asignar al nodo raíz
@doc Vacía el árbol receptor y le asigna como
valor el árbol de un único nodo cuya etiqueta
es e.  $\ast\ast$ 
```

```
Nodo raiz() const;
 $\ast\ast$ 
@memo Raíz del árbol
@return Nodo raíz del árbol receptor
@doc Devuelve el nodo raíz, que coincide con
nodo_nulo si el árbol está vacío. La operación
se realiza en tiempo O(1).  $\ast\ast$ 
```

```
Nodo izquierda(const Nodo n) const;
 $\ast\ast$ 
@memo Hijo izquierda
@param n: nodo del que se quiere obtener el
hijo a la izquierda. n no es nodo_nulo
```

@return Nodo hijo a la izquierda
@doc Devuelve el nodo hijo a la izquierda de *n*, que valdrá *nodo_nulo* si no tiene hijo a la izquierda. La operación se realiza en tiempo $O(1)$. */

Nodo **derecha**(const Nodo *n*) const;
/**
@memo Hijo derecha
@param *n*: nodo del que se quiere obtener el hijo a la derecha. *n* no es *nodo_nulo*
@return Nodo hijo a la derecha
@doc Devuelve el nodo hijo a la derecha de *n*, que valdrá *nodo_nulo* si no tiene hijo a la derecha. La operación se realiza en tiempo $O(1)$. */

Nodo **padre**(const Nodo *n*) const;
/**
@memo Nodo padre
@param *n*: nodo del que se quiere obtener el

padre. *n* no es *nodo_nulo*
@return Nodo padre
@doc Devuelve el nodo padre de *n*, que valdrá *nodo_nulo* si es la raíz. La operación se realiza en tiempo O(1). */

Tbase& **etiqueta**(const Nodo *n*);
/**
@memo Etiqueta en un nodo
@param *n*: nodo en el que se encuentra el elemento. *n* no es *nodo_nulo*
@return Referencia al elemento del nodo *n*
@doc Devuelve una referencia al elemento del nodo *n* y por tanto se puede modificar o usar el valor. La operación se realiza en tiempo O(1).*/

const Tbase& **etiqueta**(const Nodo *n*) **const**;
/**
@memo Etiqueta en un nodo

@param n: nodo en el que se encuentra el elemento. *n* no es *nodo_nulo*
@return Referencia constante al elemento del nodo *n*.
@doc Devuelve una referencia al elemento del nodo *n*. Es constante y por tanto no se puede modificar el valor. La operación se realiza en tiempo O(1). */

void asignar_subarbol(const ArbolBinario<Tbase>& orig, const Nodo nod);
/**
@memo Copia Subárbol
@param orig: árbol desde el que se va a copiar una rama
@param nod: nodo raíz del subárbol que se copia. Es un nodo del árbol *orig* y no es *nodo_nulo*
@doc El árbol receptor acaba con un valor copia del subárbol que cuelga del nodo *nod* en el árbol *orig*. La operación se realiza en tiempo O(*n*) donde *n* es el número de nodos del subárbol */

```
void podar_izquierda(Nodo n, ArbolBinario<Tbase>& dest);  
/**  
 @memo Podar subárbol izquierda  
 @param n: Nodo al que se le podará la rama  
 hijo izquierda. No es nodo_nulo y es un nodo  
 válido del árbol receptor.  
 @param dest: árbol que recibe la rama cortada  
 @doc Asigna un nuevo valor al árbol dest, con  
 todos los elementos del subárbol izquierdo del  
 nodo n en el árbol receptor. Este se queda sin di-  
 chos nodos. La operación se realiza en tiempo  
 O(1). */
```

```
void podar_derecha(Nodo n, ArbolBinario<Tbase>& dest);  
/**  
 @memo Podar subárbol derecha  
 @param n: Nodo al que se le podará la rama  
 hijo derecha. No es nodo_nulo y es un nodo  
 válido del árbol receptor.
```

*@param dest: árbol que recibe la rama cortada
@doc Asigna un nuevo valor al árbol *dest*, con todos los elementos del subárbol derecho del nodo *n* en el árbol receptor. *ste* se queda sin dichos nodos. La operación se realiza en tiempo O(1). */*

void insertar_izquierda(Nodo n, ArbolBinario <Tbase>& rama);
*/**
@memo Insertar subárbol izquierda
@param n: Nodo al que se insertará el árbol *rama* como hijo izquierdo. No es *nodo_nulo* y es un nodo válido del árbol receptor
@param rama: árbol que se insertará como hijo izquierdo.
@doc El árbol *rama* se inserta como hijo izquierda del nodo *n* del árbol receptor. El árbol *rama* queda vacío y los nodos que estaban en el subárbol izquierdo de *n* se eliminan. */*

```
void insertar_derecha(Nodo n, ArbolBinario
<Tbase> & rama);
 $\frac{3}{2} \times 10^8$ 
/** @memo Insertar subárbol derecha
@param n: Nodo al que se insertará el árbol
rama como hijo derecho. No es nodo_nulo y
es un nodo válido del árbol receptor
@param rama: árbol que se insertará como hijo
derecho.
@doc El árbol rama se inserta como hijo dere-
cho del nodo n del árbol receptor. El árbol
rama queda vacío y los nodos que estaban en
el subárbol izquierdo de n se eliminan. */
```

```
void clear();
/** @memo Borra todos los elementos
@doc Borra todos los elementos del árbol re-
ceptor. Cuando termina, el árbol está vacío.
La operación se realiza en tiempo  $O(n)$ , donde
n es el número de elementos del árbol recep-
tor. */
```

```
int size() const;
 $\ast\ast$ 
@memo Número de elementos
@return El número de elementos del árbol receptor.
@doc La operación se realiza en tiempo O(n).
@see contar */
```



```
bool empty() const;
 $\ast\ast$ 
@memo Vacío
@return Devuelve true si el número de elementos del árbol receptor es cero, false en otro caso.
@doc La operación se realiza en tiempo O(1). */
```

```
bool operator==(const ArbolBinario <Tbase>&
v) const;
/**
@memo Igualdad
@param v: ArbolBinario con la que se desea
comparar.
@return Devuelve true si el árbol receptor tiene
los mismos elementos y en el mismo orden.
false en caso contrario.
@doc La operación se realiza en tiempo O(n).
@see soniguales */
```

```
bool operator!=(const ArbolBinario <Tbase>&
v) const;
/**
@memo Distintos
@param v: ArbolBinario con la que se desea
comparar.
@return Devuelve true si el árbol receptor no
tiene los mismos elementos y en el mismo or-
den. false en caso contrario.
@doc La operación se realiza en tiempo O(n). */
```

*::

template <class T> friend istream& operator >>(istream& in, ArbolBinario <T>& v);

/**

@memo Entrada

@name operator >>

@param in: stream de entrada

@param v: árbol que leer

@return referencia al stream de entrada

@doc Lee de in un árbol y lo almacena en v.
*El formato aceptado para la lectura se puede consultar en la función de salida. */*

*::

template <class T> friend ostream& operator <<(ostream& out, const ArbolBinario <T>& v);

/**

@memo Salida

@name operator <<

@param out: stream de salida

@param v: árbol que escribir

@return referencia al stream de salida

@doc Escribe en la salida todos los nodos del árbol *v* siguiendo un recorrido en preorden. La forma de impresión de cada nodo es:

- Si el nodo es *nodo_nulo*, imprime el carácter 'x'
- Si el nodo no es *nodo_nulo*, imprime el carácter 'n' seguido de un espacio, al que sigue la impresión de la etiqueta.

@see escribe_arbol */

/*@}*/

};

```
/* IMPLEMENTACION DE LAS FUNCIONES */
```

```
#include <cassert>
```

```
using namespace std;
```

```
/*-----*/
```

```
// FUNCIONES PRIVADAS
```

```
/*-----*/
```

```
template <class Tbase>
```

```
void ArbolBinario<Tbase>::destruir(Nodo n)
```

```
{
```

```
    if (n!=0) {
```

```
        destruir(n->izqda);
```

```
        destruir(n->drcha);
```

```
        delete n;
```

```
}
```

```
}
```

```
/*----- */
```

```
template <class Tbase>
int ArbolBinario<Tbase>::contar(Nodo n)
{
    if (n==0)
        return 0;
    else return 1+contar(n->izqda)+  

                contar(n->dcha);
}
```

```
/*----- */
```

```
template <class Tbase>
void ArbolBinario<Tbase>::copiar(Nodo& dest,
                                     Nodo orig)
{
    if (orig==0)
        dest= 0;
    else {
        dest= new nodo;
        dest->etiqueta= orig->etiqueta;
        copiar (dest->izqda,orig->izqda);
        copiar (dest->drcha,orig->drcha);
        if (dest->izqda!=0)
            dest->izqda->padre= dest;
        if (dest->drcha!=0)
            dest->drcha->padre= dest;
    }
}

/*-----*/
```

```
template <class Tbase>
bool ArbolBinario<Tbase>::soniguales(Nodo n1,
                                         Nodo n2)
{
    if (n1==0 && n2==0)
        return true;
    if (n1==0 || n2==0)
        return false;
    if (n1->etiqueta!=n2->etiqueta)
        return false;
    if (!soniguales(n1->izqda,n2->izqda))
        return false;
    if (!soniguales(n1->drcha,n2->drcha))
        return false;
    return true;
}
```

```
/*----- */
```

```
template <class Tbase>
void ArbolBinario<Tbase>::escribe_arbol(
    ostream& out, Nodo nod) const
{
    if (nod==0)
        out << "x ";
    else {
        out << "n "<< nod->etiqueta << " ";
        escribe_arbol(out,nod->izqda);
        escribe_arbol(out,nod->drcha);
    }
}
```

```
/*----- */
```

```
/**
```

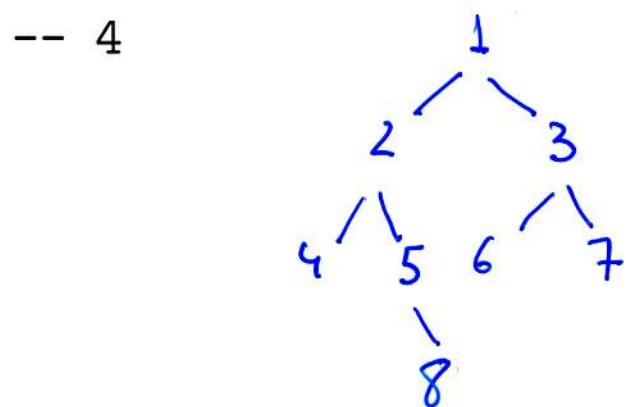
Un ejemplo de formato de entrada/salida es el siguiente:

El \'arbol:

n 1 n 2 n 4 x x n 5 x n 8 x x n 3 n 6 x x n 7 x x
tiene la siguiente estructura:

```
-- 1
  |-- 3
    |   |-- 7
    |
    |-- 6
-- 2
  |-- 5
  |
  |-- 8
  |
  |-- x
-- 4
```

```
*/
```



```
template <class Tbase>
void ArbolBinario<Tbase>::lee_arbol(istream&
                                         in, Nodo& nod)
{
    char c;
    in >> c;
    if (c=='n') {
        nod= new nodo;
        in >> nod->etiqueta;
        lee_arbol(in,nod->izqda);
        lee_arbol(in,nod->drcha);
        if (nod->izqda!=0)
            nod->izqda->padre=nod;
        if (nod->drcha!=0)
            nod->drcha->padre=nod;
    }
    else nod= 0;
}
```

```
/*----- */
```

```
/*-----*/
// FUNCIONES PUBLICAS
/*-----*/
template <class Tbase>
inline ArbolBinario<Tbase>::ArbolBinario()
{
    laraiz=0;
}
/*----- */

template <class Tbase>
ArbolBinario<Tbase>::ArbolBinario(const
                                         Tbase& e)
{
    laraiz = new nodo;
    laraiz->padre= laraiz->izqda=
                    laraiz->drcha= 0;
    laraiz->etiqueta= e;
}
/*----- */
```

```
template <class Tbase>
ArbolBinario<Tbase>::ArbolBinario (const
                                         ArbolBinario<Tbase>& v)
{
    copiar (laraiz,v.laraiz);
    if (laraiz!=0)
        laraiz->padre= 0;
}
/*----- */
```



```
template <class Tbase>
inline ArbolBinario<Tbase>::~ArbolBinario()
{
    destruir(laraiz);
}
/*----- */
```

```
template <class Tbase>
ArbolBinario<Tbase>& ArbolBinario<Tbase>::
    operator=(const ArbolBinario<Tbase>& v)
{
    if (this!=&v) {
        destruir(laraiz);
        copiar (laraiz,v.laraiz);
        if (laraiz!=0)
            laraiz->padre= 0;
    }
    return *this;
}
```

```
/*----- */
```

```
template <class Tbase>
void ArbolBinario<Tbase>::AsignaRaiz(const
                                         Tbase& e)
{
    destruir(laraiz);
    laraiz = new nodo;
    laraiz->padre= laraiz->izqda=
                                laraiz->dcha= 0;
    laraiz->etiqueta= e;
}
/*----- */

template <class Tbase>
hypname inline ArbolBinario<Tbase>::Nodo ArbolBinario
                                         <Tbase>::raiz() const
{
    return laraiz;
}
/*----- */
```

```
template <class Tbase>
typename inline ArbolBinario<Tbase>::Nodo ArbolBinario
    <Tbase>::izquierda(const Nodo p) const
{
    assert(p!=0);
    return (p->izqda);
}
/*----- */
```

```
template <class Tbase>
typename ArbolBinario<Tbase>::Nodo ArbolBinario<Tbase>::
    derecha(const Nodo p) const
{
    assert(p!=0);
    return (p->drcha);
}
/*----- */
```

```
template <class Tbase>
typename ArbolBinario<Tbase>::Nodo ArbolBinario<Tbase>::
    padre(const Nodo p) const
```

```
{
```

```
    assert(p!=0);
    return (p->padre);
```

```
}
```

```
/*----- */
```

```
template <class Tbase>
Tbase& ArbolBinario<Tbase>::etiqueta(const
                                         Nodo p)
```

```
{
```

```
    assert(p!=0);
    return (p->etiqueta);
```

```
}
```

```
/*----- */
```

```
template <class Tbase>
const Tbase& ArbolBinario<Tbase>::etiqueta(
                           const Nodo p) const
{
    assert(p!=0);
    return (p->etiqueta);
}
/*----- */

template <class Tbase>
void ArbolBinario<Tbase>::asignar_subarbol(
const ArbolBinario<Tbase>& orig,const Nodo nod)
{
    destruir(laraiz);
    copiar(laraiz,nod);
    if (laraiz!=0)
        laraiz->padre=0;
}
/*----- */
```

```
template <class Tbase>
void ArbolBinario<Tbase>::podar_izquierda(
    Nodo n, ArbolBinario<Tbase>& dest)
{
    assert(n!=0);
    destruir(dest.laraiz);
    dest.laraiz=n->izqda;
    if (dest.laraiz!=0) {
        dest.laraiz->padre=0;
        n->izqda=0;
    }
}
```

```
/*----- */
```

```
template <class Tbase>
void ArbolBinario<Tbase>::podar_derecha(Nodo n,
                                             ArbolBinario<Tbase>& dest)
{
    assert(n!=0);
    destruir(dest.laraiz);
    dest.laraiz=n->drcha;
    if (dest.laraiz!=0) {
        dest.laraiz->padre=0;
        n->drcha=0;
    }
}
```

```
/*----- */
```

```
template <class Tbase>
void ArbolBinario<Tbase>::insertar_izquierda
(Nodo n, ArbolBinario<Tbase>& rama)
{
    assert(n!=0);
    destruir(n->izqda);
    n->izqda=rama.laraiz;
    if (n->izqda!=0) {
        n->izqda->padre= n;
        rama.laraiz=0;
    }
}

/*----- */
```

```
template <class Tbase>
void ArbolBinario<Tbase>::insertar_derecha(
    Nodo n, ArbolBinario<Tbase>& rama)
{
    assert(n!=0);
    destruir(n->drcha);
    n->drcha=rama.laraiz;
    if (n->drcha!=0) {
        n->drcha->padre= n;
        rama.laraiz=0;
    }
}
```

```
/*----- */
```

```
template <class Tbase>
void ArbolBinario<Tbase>::clear()
{
    destruir(laraiz);
    laraiz= 0;
}
/*----- */

template <class Tbase>
inline int ArbolBinario<Tbase>::size() const
{
    return contar(laraiz);
}
/*----- */

template <class Tbase>
inline bool ArbolBinario<Tbase>::empty() const
{
    return laraiz==0;
}
/*----- */
```

```
template <class Tbase>
inline bool ArbolBinario<Tbase>::operator==
    (const ArbolBinario<Tbase>& v) const
{
    return soniguales(laraiz,v.laraiz);
}
```

```
/*----- */
```

```
template <class Tbase>
inline bool ArbolBinario<Tbase>::operator!=
    (const ArbolBinario<Tbase>& v) const
{
    return !(*this==v);
}
```

```
/*----- */
```

```
template <class Tbase>
inline istream& operator>> (istream& in,
                               ArbolBinario<Tbase>& v)

{
    v.lee_arbol(in,v.laraiz);
    return in;
}

/*----- */
```



```
template <class Tbase>
inline ostream& operator<< (ostream& out,
                               const ArbolBinario<Tbase>& v)

{
    v.escribe_arbol(out,v.laraiz);
    return out;
}

/*----- */
```



```
#endif
```



```
/* Fin Fichero: ArbolBinario.cpp */
```