

# Tecnología de la Programación

## Árboles

---

2020

Juan Antonio Sánchez Laguna  
Grado en Ingeniería Informática  
Facultad de Informática  
Universidad de Murcia

## **TABLA DE CONTENIDOS**

<b>ESTRUCTURAS DE DATOS ENLAZADAS ARBORESCENTES: ÁRBOLES</b>	<b>3</b>
TERMINOLOGÍA	3
TIPOS DE ÁRBOL	4
RECORRIDOS	5
REPRESENTACIÓN DE ÁRBOLES BINARIOS EN C	6
OPERACIONES CON ÁRBOLES BINARIOS EN C	7
ARBOLES BINARIOS Y EXPRESIONES	11
REPRESENTACIÓN DE ÁRBOLES N-ARIOS EN C	12
REPRESENTACIÓN DE ÁRBOLES GENERALES EN C	13
OPERACIONES CON ÁRBOLES GENERALES EN C	14
ARBOLES BINARIOS DE BÚSQUEDA	17

## Estructuras de datos enlazadas arborescentes: Árboles

Un árbol representa una colección de elementos organizados jerárquicamente o mediante alguna relación de parentesco. Son un caso particular de grafo, por lo que están compuestos de nodos y aristas. Cada nodo representa un elemento mientras que las aristas representan las relaciones existentes entre los elementos.

Los árboles se caracterizan por tener un nodo designado como la raíz del árbol. La elección del nodo raíz determina la noción de padre e hijo. Cada nodo, excepto la raíz, tendrá un único padre que será el primer nodo en la secuencia que une a éste con la raíz. Si un nodo  $p$  es el padre de otro nodo  $c$ , diremos que  $c$  es hijo de  $p$ .

Habitualmente, se dibujan invertidos. Es decir, con la raíz en la parte superior y las hojas en la parte inferior. Cada nodo se dibuja como un círculo con el elemento que guarda. La raíz se puede destacar con un trazo de mayor grosor. Las relaciones entre nodos se representan con líneas que unen los círculos.

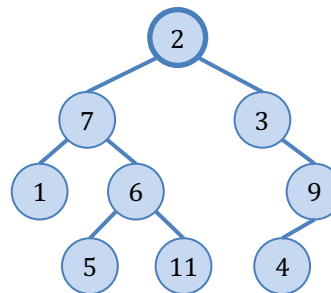


Fig. 1 Representación gráfica de un árbol

Los hijos de un nodo en un árbol también son nodos, pero cualquier nodo puede considerarse como la raíz de un árbol. Por lo tanto, los hijos de un nodo también pueden considerarse árboles o subárboles. De ahí que sea posible definir un árbol de forma recursiva del siguiente modo: un árbol es vacío o bien consiste en un nodo, denominado raíz, con uno o más árboles como hijos.

### Terminología

Para poder referirse a las distintas propiedades de los árboles y describir algoritmos que utilizan árboles conviene definir algunos términos.

Una hoja es un nodo sin hijos, mientras que un nodo interno es el que tiene uno o más hijos. Usando el árbol de la Fig. 1 como ejemplo se puede decir que el nodo 2 es la raíz, los nodos 1, 5, 11 y 4 son hojas y que 2, 7, 6, 3 y 9 son nodos internos.

Se dice que dos nodos son hermanos si tienen el mismo padre, como 7 y 3 o 5 y 11.

Un camino es una secuencia de nodos en la que cada uno es padre del siguiente. La longitud de un camino es el número de aristas que tenga, es decir, el número de nodos menos uno. El camino entre los nodos 7 y 11 tiene longitud 2.

Los ancestros de un nodo son todos los nodos en el camino que une la raíz con él mismo. Y si  $c$  es el ancestro de un nodo  $d$ ,  $d$  es descendiente de  $c$ .

La altura de un nodo es la longitud del camino más largo entre él y cualquiera de sus descendientes. La altura de cualquier hoja es cero. La altura de un árbol es la altura de su raíz. El árbol de la Fig. 1 tiene altura 3 y la de los nodos 7 y 3 es 2.

La profundidad de un nodo es la longitud del camino que va de la raíz a él. La profundidad de la raíz es cero. En el ejemplo de la Fig. 1 la profundidad de 9 es 2.

Un nivel es el conjunto de nodos que está a una misma profundidad. Un árbol de altura  $h$  tiene  $h + 1$  niveles. El nivel 2 del árbol de la Fig. 1 tiene tres nodos: 1, 6 y 9.

## Tipos de árbol

Los **árboles generales** son los que no tienen ninguna limitación en cuanto al número de hijos que puede tener cada nodo.

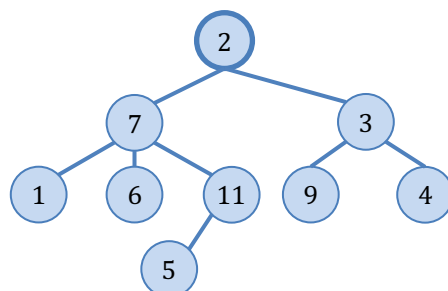


Fig. 2. Ejemplos de árbol general

Los **árboles n-arios** son árboles en los que cada nodo puede tener un máximo de  $n$  hijos. Para referirse a los hijos de un mismo nodo en un árbol general o n-ario se suele establecer un orden, por ejemplo, de izquierda a derecha. Así pues, en el ejemplo de la Fig. 2, el nodo 1 es el primer hijo o hijo más a la izquierda del nodo 7 y el nodo 11 es su último hijo.

Los **árboles binarios** son un tipo especial de árbol 2-ario caracterizados por diferenciar los dos hijos que tiene cada nodo. En un árbol binario cada nodo tiene un hijo izquierdo y un hijo derecho. Evidentemente, para que esta definición tenga sentido se debe admitir la existencia del árbol vacío o nulo.

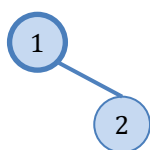


Fig. 3. Los dos únicos árboles binarios existentes compuestos por dos nodos

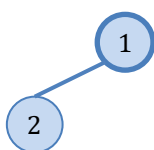


Fig. 4. El único árbol 2-ario existente compuesto por dos nodos

No es lo mismo un árbol binario que un árbol 2-ario. La diferencia se ve al dibujar los dos únicos árboles binarios existentes compuestos por dos nodos: Fig. 3 y compararlos con el único árbol 2-ario de dos nodos que existe: Fig. 4. Ser hijo izquierdo o derecho en un árbol binario aporta una información extra a la propia de ser hijo. En un árbol 2-ario, el único hijo de un nodo es simplemente su hijo.

Los **árboles binarios de búsqueda** son un tipo especial de árbol binario caracterizados por imponer un criterio a todos los nodos que los componen. En concreto, que el elemento asociado a cualquier nodo debe ser mayor que todos los de su subárbol izquierdo y menor que todos los de su subárbol derecho.

Los árboles binarios de búsqueda tienen muchas aplicaciones prácticas.

Combinan la flexibilidad de las estructuras enlazadas lineales y la eficiencia de la búsqueda binaria en arrays ordenados. Por lo tanto, son muy útiles para ordenar y buscar elementos de forma eficiente.

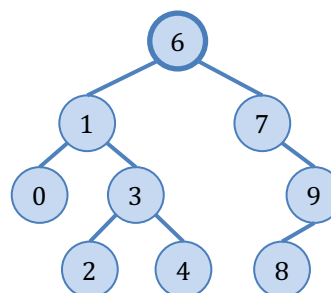


Fig. 5. Ejemplo de árbol binario de búsqueda

## Recorridos

Recorrer o atravesar un árbol significa visitar todos los nodos del mismo una única vez siguiendo un método concreto, entendiendo por visitar el hecho de tener acceso al nodo y hacer algo con él o los datos que almacene.

Los recorridos son una de las operaciones más habituales cuando se trabaja con árboles. Sirven para mostrar el contenido de un árbol, para buscar un elemento en el mismo, para hacer una copia de un árbol, para comparar dos árboles, etc.

Sin embargo, en la estructura ramificada de un árbol no se puede aplicar una estrategia de recorrido secuencial como con las listas. En cada nodo se plantean varias alternativas a tomar y se debe elegir una por la que comenzar, pero no se deben dejar sin explorar el resto de alternativas.

Afortunadamente, un árbol se puede definir recursivamente como un nodo cuyos hijos son otros árboles. Por lo tanto, la solución recursiva al problema del recorrido consiste en visitar el nodo raíz y recorrer sus subárboles. Según el orden en que se hagan estas operaciones se obtiene un algoritmo u otro. Los tres algoritmos clásicos para recorrer recursiva y sistemáticamente un árbol son los siguientes:

- **Preorden:** Consiste en visitar el nodo raíz del árbol antes de hacer lo mismo recursivamente con los hijos de izquierda a derecha.
- **Postorden:** Consiste en visitar recursivamente cada hijo de izquierda a derecha antes de hacer lo mismo con el nodo raíz del árbol.
- **Inorden:** Consiste en visitar primero el hijo izquierdo, después el nodo raíz del árbol y después el hijo derecho.

Cuando se recorren árboles binarios queda claro qué hijo es el izquierdo y cuál el derecho. Al recorrer árboles generales o n-arios se hacen las visitas respetando el orden de izquierda a derecha establecido entre los hijos. Pero en este caso sólo preorden y postorden tienen una utilidad práctica.

Aplicando cada algoritmo al árbol de la Fig. 6 y mostrando el valor de cada nodo al visitarlo, se obtienen los siguientes resultados:

- **Preorden:** 2, 7, 1, 6, 5, 11, 3, 9, 4.
- **Postorden:** 1, 5, 11, 6, 7, 4, 9, 3, 2.
- **Inorden:** 1, 7, 5, 6, 11, 2, 3, 4, 9

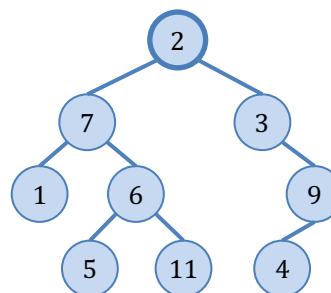


Fig. 6. Ejemplo de árbol

Los tres algoritmos de recorrido anteriores se clasifican como algoritmos de **recorrido en profundidad** pues examinan cada rama completa antes de empezar con las siguientes.

Existe otro tipo de recorrido, denominado **recorrido en anchura**, que visita los nodos de un árbol por niveles. Empieza por la raíz, después visita todos sus hijos, después todos sus nietos y continúa visitando todos los nodos de cada nivel antes de empezar con el siguiente. El recorrido en anchura del árbol de la Fig. 6 sería el siguiente: 2, 7, 3, 1, 6, 9, 5, 11, 4

## Representación de árboles binarios en C

Para representar un árbol en C se pueden usar diversas estructuras de datos. En este apartado se estudia cómo hacerlo usando estructuras enlazadas no lineales y memoria dinámica, pero también es posible hacerlo usando arrays.

Dado que un árbol es un conjunto de nodos interconectados, éste queda definido si se define la estructura de sus nodos. Obviamente, cada nodo debe incluir el elemento que vaya a guardar, pero también se necesita saber quiénes son sus hijos y cómo acceder a ellos.

El caso más sencillo es el del árbol binario porque el número de hijos de cada nodo está limitado a dos. Por lo tanto, el problema se resuelve dotando a cada nodo de dos punteros: uno para el hijo izquierdo y otro para el hijo derecho.

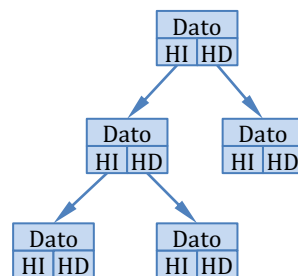


Fig. 7. Representación gráfica de una estructura enlazada con forma de árbol binario

Así pues, para representar los nodos de un árbol binario en C se usa una estructura con tres campos. El primero, llamado **dato**, almacenará un elemento del tipo de datos de los elementos que vaya a guardar cada nodo del árbol. Los otros dos, llamados **hijoIzquierdo** e **hijoDerecho** serán punteros a estructuras como la que se está definiendo, y se encargarán de guardar la dirección en memoria de los nodos que contengan el hijo izquierdo y el derecho respectivamente.

```
struct Nodo {
    int dato;
    struct Nodo * hijoIzquierdo;
    struct Nodo * hijoDerecho;
};
typedef struct Nodo * ArbolBinario;
```

Fig. 8. Definición de una estructura enlazada para representar árboles binarios en C

La Fig. 8 muestra la definición de la estructura `struct Nodo` que sirve para representar un nodo de un árbol binario. También se define la palabra `ArbolBinario` como sinónimo de puntero a esa estructura. La naturaleza recursiva de la definición de árbol se refleja aquí en el hecho de que `hijoIzquierdo` e `hijoDerecho` son punteros a `struct Nodo`, es decir, son árboles en sí mismos.

Para representar el árbol vacío se puede usar el valor `NULL`. Por lo tanto, para indicar que un nodo no tiene hijo izquierdo o derecho, o lo que es lo mismo, que su hijo izquierdo o derecho es el árbol vacío, se guarda el valor `NULL` en el campo adecuado.

Finalmente, también se podría añadir a cada nodo un tercer puntero que mantenga la dirección del nodo padre. Esto aumenta el uso de memoria, pero reduce el tiempo necesario para recorrer el árbol en sentido ascendente.

## Operaciones con árboles binarios en C

Antes de poder realizar cualquier operación sobre un árbol binario es necesario contar con uno. Para representar un árbol binario en el programa se usa una variable de tipo `ArbolBinario` que se encargará de mantener la dirección del nodo designado como raíz. El árbol se construye creando nodos en memoria dinámica y enlazando unos con otros. Por ejemplo, la Fig. 9 muestra cómo crear un árbol binario con tres nodos.

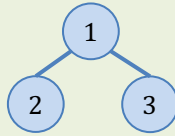
```
#include <stdio.h>
#include <stdlib.h>

struct Nodo {
    int dato;
    struct Nodo * hijoIzquierdo;
    struct Nodo * hijoDerecho;
};

typedef struct Nodo * ArbolBinario;

ArbolBinario crea_arbol( int dato ) {
    ArbolBinario nuevo = malloc( sizeof( struct Nodo ) );
    nuevo->dato = dato;
    nuevo->hijoIzquierdo = NULL;
    nuevo->hijoDerecho = NULL;
    return nuevo;
}

int main( int argc, char * argv[] ) {
    ArbolBinario raiz = crea_arbol( 1 );
    raiz->hijoIzquierdo = crea_arbol( 2 );
    raiz->hijoDerecho = crea_arbol( 3 );
}
```



```
graph TD
    1((1)) --- 2((2))
    1 --- 3((3))
```

Fig. 9. Programa de ejemplo que crea un árbol binario en C

El programa mostrado en la Fig. 9 incluye la definición de las estructuras de datos necesarias para crear y utilizar un árbol binario y la función `crea_arbol`. Esta función devuelve la dirección en memoria dinámica de un árbol compuesto por un único nodo. Dicho nodo contendrá el dato pasado como parámetro a la función y sus dos hijos tendrán el valor `NULL`.

En la función `main` se declara la variable `raiz` de tipo `ArbolBinario` que será la usada para representar el árbol completo. Tras asignarle el resultado de ejecutar `crea_arbol(1)`, se almacena en sus hijos la dirección de otros dos nodos creados de la misma forma. Así pues, la variable `raiz` representará un árbol con tres nodos.

Una vez que se tiene un árbol construido, para mostrar todos sus elementos se puede usar cualquiera de los algoritmos de recorrido que ya se han comentado. En la Fig. 10 se puede ver cómo hacerlo con el algoritmo denominado preorden.

```
void muestra( ArbolBinario a ) {
    if ( a == NULL ) return;
    printf( "%d ", a->dato );
    muestra( a->hijoIzquierdo );
    muestra( a->hijoDerecho );
}
```

Fig. 10. Procedimiento que recorre un árbol binario aplicando el algoritmo de preorden en C

El procedimiento muestra comienza comprobando si el árbol pasado como parámetro es vacío pues, coincidiendo con la definición recursiva de árbol, éste será el caso base de la recursión. El caso general se resuelve mostrando primero el valor de la raíz, es decir, el dato contenido en el nodo cuya dirección se ha recibido como parámetro, y después, aplicando recursivamente el procedimiento al hijo izquierdo y al derecho en ese mismo orden.

Si lo único que se necesita es mostrar los elementos del árbol, y no importa el orden en el que aparezcan, cualquiera de los tres algoritmos de recorrido serviría. Evidentemente, el resultado, es decir, el orden en el que se mostrarían los datos, sería distinto en cada caso. La implementación de los otros dos algoritmos de recorrido se obtiene a partir de esta sin más que cambiar el orden de las tres últimas instrucciones siguiendo la descripción del algoritmo en cuestión. Para seguir un postorden la impresión sería la última instrucción y para seguir un inorden, la impresión estaría entre las dos llamadas recursivas.

Pero algunas veces el orden en el que se realiza el recorrido sí es importante. Por ejemplo, para liberar la memoria dinámica usada por un árbol no se puede liberar primero la de la raíz y después la de sus hijos. Es necesario hacer la liberación en el orden inverso. En este caso, es necesario utilizar un postorden para recorrer el árbol, liberando en primer lugar las hojas, después sus padres y así sucesivamente hasta eliminar el árbol completo. La Fig. 11 muestra cómo hacerlo.

```
void libera( ArbolBinario a ) {  
    if ( a == NULL ) return;  
    libera( a->hijoIzquierdo );  
    libera( a->hijoDerecho );  
    free( a );  
}
```

**Fig. 11. Función que libera la memoria asociada a un árbol binario**

Los recorridos también sirven para buscar un elemento entre los incluidos en un árbol, contar los que cumplan cierto criterio, y muchos otros ejemplos.

```
int existe( ArbolBinario a, int buscado ) {  
    if ( a == NULL ) return 0;  
    if ( a->dato == buscado ) return 1;  
    if ( existe( a->hijoIzquierdo, buscado ) ) return 1;  
    if ( existe( a->hijoDerecho, buscado ) ) return 1;  
    return 0;  
}
```

**Fig. 12. Función que comprueba la existencia de un elemento en un árbol binario**

La Fig. 12 muestra cómo implementar una función que comprueba la existencia de un elemento en un árbol. En esencia, lo que se hace es un recorrido en preorden porque antes de las dos llamadas recursivas se accede al nodo raíz para comprobar si ahí está el dato buscado. La razón de elegir este algoritmo es simple. No tendría sentido buscar primero en el subárbol izquierdo o en el derecho si el elemento buscado ya se ha encontrado en la raíz. Pero, la diferencia con los procedimientos muestra y libera es que ahora sí se trata de una función. Su objetivo es devolver un valor: 1 si el elemento buscado está en el árbol y 0 si no lo está. Por tanto, si el elemento está en la raíz se acaba la recursión, si el elemento se ha encontrado en el subárbol izquierdo también se acaba con resultado positivo y, si no se encuentra en el subárbol derecho, la función acaba devolviendo cero.



Un caso ligeramente diferente es el que se da al resolver el problema de contar los elementos que cumplan cierto criterio, por ejemplo, ser mayores que cero.

```
int positivos( ArbolBinario a ) {  
    if ( a == NULL ) return 0;  
    int n = 0;  
    if ( a->dato > 0 ) n = 1;  
    return n + positivos( a->hijoIzquierdo ) + positivos( a->hijoDerecho );  
}
```

**Fig. 13. Función que cuenta el número de elementos positivos en un árbol binario**

El código mostrado en la Fig. 13 resuelve el problema aplicando la siguiente solución recursiva: el número de elementos positivos de un árbol es cero si el árbol está vacío. Si no, será la suma del número de elementos positivos de cada subárbol más uno cuando el de la raíz también lo sea.

Al contar apariciones, lo más importante es revisar todos los nodos, pero no contar ninguno más de una vez. Como cualquiera de los tres algoritmos de recorrido garantiza precisamente esto, no importa cuál se aplique. En este ejemplo, la función hace un recorrido en preorden, pero es sencillo cambiarlo para que se siga cualquiera de los otros dos. Basta con guardar en una variable el resultado de cada llamada recursiva para poder ordenar las tres acciones (dos llamadas recursivas y una comprobación del nodo raíz) como se desee y, finalmente, devolver la suma.

La copia de un árbol es una operación que también se puede expresar de forma recursiva muy fácilmente: la copia de un árbol vacío es otro árbol vacío, y la de un árbol no vacío será un nuevo nodo con el mismo dato que la raíz, y cuyos hijos serán las copias de los hijos del primero. La función copia mostrada en la Fig. 14 implementa directamente esta solución haciendo uso de la función crea\_arbol que se definió anteriormente.

```
ArbolBinario copia( ArbolBinario a ) {  
    if ( a == NULL ) return NULL;  
    ArbolBinario nuevo = crea_arbol( a->dato );  
    nuevo->hijoIzquierdo = copia( a->hijoIzquierdo );  
    nuevo->hijoDerecho = copia( a->hijoDerecho );  
    return nuevo;  
}
```

**Fig. 14. Función que devuelve una copia un árbol binario**

Otro ejercicio interesante es el de comparar dos árboles. Si los dos árboles son vacíos entonces son iguales. Si no, si sólo uno es vacío, serán distintos. En caso de que ninguno sea vacío la comparación depende, primero, del elemento raíz. Si las dos raíces son distintas no hace falta continuar. Pero si son iguales, la igualdad de los árboles depende de que sean iguales todos sus hijos dos a dos. Es decir, el izquierdo de uno igual que el izquierdo del otro y lo mismo con los derechos.

```
int compara( ArbolBinario a, ArbolBinario b ) {  
    if ( a == NULL && b == NULL ) return 1;  
    if ( a == NULL || b == NULL ) return 0;  
    if ( a->dato != b->dato ) return 0;  
    return compara( a->hijoIzquierdo, b->hijoIzquierdo ) &&  
           compara( a->hijoDerecho, b->hijoDerecho );  
}
```

**Fig. 15. Función que compara dos árboles binarios**

La altura de un árbol se puede calcular sumando uno a la altura del mayor de sus subárboles, teniendo en cuenta que las hojas tienen altura cero. Además, la altura del árbol vacío no está definida, por lo que se debe prestar especial atención al caso base cuando se escriba la función recursiva que resuelve el problema.

```
int altura( ArbolBinario a ) {  
    if ( a == NULL ) return 0;  
    if ( a->hijoIzquierdo == NULL && a->hijoDerecho == NULL ) return 0;  
    int ahi = altura( a->hijoIzquierdo );  
    int ahd = altura( a->hijoDerecho );  
    if ( ahi > ahd ) return 1 + ahi;  
    return 1 + ahd;  
}
```

**Fig. 16. Función que calcula la altura de un árbol binario**

El caso base en la función mostrada en la Fig. 16 es doble. Por un lado, se comprueba si el árbol recibido como parámetro es vacío y en caso afirmativo se devuelve cero. Este caso no está contemplado en el problema, pero protege la función ante posibles usos incorrectos de la misma. A continuación, aparece el caso base real, que se da cuando el árbol recibido es una hoja. En ese caso la altura es cero. Si la función no ha terminado en este punto se trata el caso general. Primero se calcula la altura de los dos hijos, y después, se devuelve la mayor incrementada en uno, pues la raíz actual está un nivel por encima del de todos sus hijos.

Como último ejemplo se plantea el consistente en contar el número de hojas. Este problema se soluciona aplicando una estrategia similar a la usada para contar los nodos cuyos elementos cumplan algún criterio, como en el ejemplo de la función positivos, pero usando como caso base el mismo que en la función altura.

Pero, aunque el caso base sea el mismo, se debe tratar de forma diferente. En esta función cuando se alcanza una hoja hay que devolver un uno, pues el objetivo es contar nodos que sean hojas. El caso general se resuelve obteniendo recursivamente el número de hojas de los subárboles y devolviendo su suma.

```
int numero_hojas( ArbolBinario a ) {  
    if ( a == NULL ) return 0;  
    if ( a->hijoIzquierdo == NULL && a->hijoDerecho == NULL ) return 1;  
    return numero_hojas( a->hijoIzquierdo ) + numero_hojas( a->hijoDerecho );  
}
```

**Fig. 17. Función que calcula el número de hojas de un árbol binario**

En las funciones `altura` y `numero_hojas` y se ha supuesto que el problema no está definido para el caso vacío. Se trata para proteger la función de usos incorrectos, pero el verdadero caso base se da cuando se alcanza un nodo hoja.

Otras funciones, como por ejemplo la función `positivos`, también se podrían resolver usando esta misma estrategia. Pero normalmente, el código es más simple cuando el único caso base es el árbol vacío.

## Árboles binarios y expresiones

Una de las aplicaciones típicas de los árboles binarios es la de representar expresiones aritméticas o lógicas. Estos árboles contienen operadores en los nodos internos y operandos en las hojas. La Fig. 18 muestra dos expresiones aritméticas representadas mediante sendos árboles binarios.



Fig. 18. Ejemplos de árboles binarios representando expresiones aritméticas

Cuando un árbol de este tipo se recorre en preorden se obtiene lo que se denomina la notación prefija. En el ejemplo anterior, la expresión representada por el árbol de la izquierda se describe en notación prefija como  $+ * 2 4 3$ . Un recorrido en postorden genera la denominada notación postfija. En el ejemplo anterior, la notación postfija asociada al árbol de la izquierda sería:  $2 4 * 3 +$ .

El recorrido en inorden de un árbol binario que represente una expresión aritmética o lógica es muy útil porque genera la habitual notación infija. En el ejemplo anterior, la notación infija de la expresión de la izquierda sería:  $2 * 4 + 3$ .

Los árboles binarios pueden representar las expresiones aritméticas incluyendo la información relativa a la prioridad de aplicación de los operadores. En los dos árboles del ejemplo anterior se ve claramente qué operación se hace antes. Sin embargo, el recorrido infijo del árbol de la derecha da un resultado que se puede interpretar erróneamente:  $3 * 4 + 2$ . Al no aparecer los paréntesis se puede leer la expresión de forma que primero se haga la multiplicación y después la suma.

Por suerte, es muy fácil escribir un procedimiento que, haciendo un recorrido en inorden, muestre la expresión sin ambigüedades usando paréntesis.

```
void infijo( ArbolBinario a ) {
    if ( a == NULL ) return 0;
    if ( !isdigit(a->dato) ) printf( "(" );
    infijo( a->hijoIzquierdo );
    printf( "%c", a->dato );
    infijo( a->hijoDerecho );
    if ( !isdigit(a->dato) ) printf( ")" );
}
```

Fig. 19. Función que calcula la altura de un árbol binario

La función mostrada en la Fig. 19 asume que los elementos del árbol binario son caracteres y que representa una expresión aritmética cuyos operandos son los caracteres de los dígitos del 0 al 9, y cuyos operadores son los caracteres '+', '-', '\*', y '/'.

Usando la función `isdigit` declarada en el fichero `ctype.h` de la biblioteca estándar se determina si el nodo es un operador o no. En caso positivo se encierra todo entre paréntesis. De este modo, en el caso de la expresión de la derecha de la Fig. 18, la función mostraría:  $(3*(4 + 2))$ , que no presenta ambigüedad al leerse, aunque tenga algún paréntesis de más.

## Representación de árboles n-arios en C

Los árboles n-arios suelen representarse utilizando un array de  $n$  punteros. Y siempre que la mayoría de nodos tenga un número de hijos cercano al máximo, esta es la mejor alternativa porque permite un acceso directo a cada hijo.

Por ejemplo, un quadtree es un árbol 4-ario que se utiliza para describir eficientemente el área ocupada por una figura en una superficie plana. La idea consiste en dividir la superficie en cuatro partes iguales, y repetir el proceso recursivamente en aquellas partes usadas parcialmente por la figura. La división recursiva se detiene cuando el área es suficientemente pequeña.

En un quadtree cada nodo del árbol guarda el estado del área que representa. El estado puede ser “Gris” si el área está parcialmente cubierta, “Blanco” si está vacía y “Negro” si está completamente cubierta. Las áreas grises se subdividen en cuatro cuadrantes numerados de izquierda a derecha y de arriba abajo. Y cada nodo tiene un array de cuatro punteros, uno para el árbol que representa cada subdivisión.

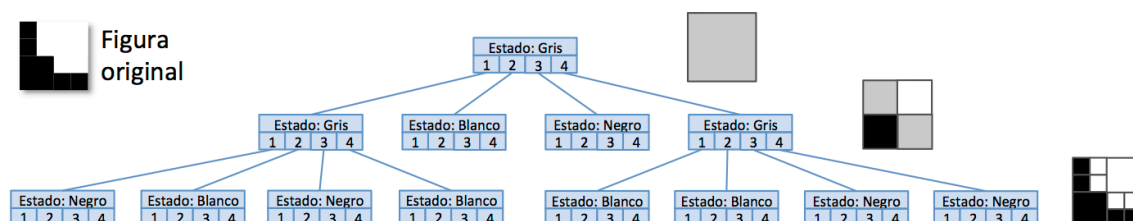


Fig. 20. Ejemplo de árbol n-ario quadtree

En el ejemplo mostrado en la Fig. 20, la raíz del árbol, que representa la visión global de la figura, está en estado “Gris” porque hay partes cubiertas y partes que no. Pero en el segundo nivel del árbol ya hay dos cuadrantes que no necesitan ser subdivididos (2 y 3) pues el área que representan está completamente cubierta o vacía. En el tercer nivel todos los cuadrantes están llenos o vacíos, por lo que no es necesario seguir aumentando la profundidad del árbol para representar la figura.

Esta misma idea se aplica también para modelar objetos tridimensionales dividiendo el volumen en ocho partes iguales. Estos árboles, denominados octree, se usan en videojuegos como Minecraft para acelerar el proceso de detección de colisiones, o determinar qué partes son visibles de forma eficiente.

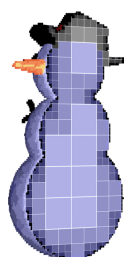


Fig. 21. Ejemplo de octree para modelar un objeto tridimensional (Fuente: wikimedia)

Sin embargo, cuando el número de hijos que puede tener cada nodo es grande y el número de nodos con muy pocos hijos también es alto, entonces, el espacio no usado de los arrays puede representar un gasto de memoria excesivo. En estos casos, es mejor utilizar la representación de árboles generales que se estudia a continuación, aunque, como se verá, se pierda la posibilidad de acceder directamente a cada hijo.

## Representación de árboles generales en C

Los árboles generales permiten representar datos organizados jerárquicamente en los que el número máximo de hijos por nodo es desconocido a priori. La forma más sencilla de representar un árbol general es con listas de hijos. Con esta representación cada nodo tiene el elemento que guarda y una lista de hijos representada mediante una estructura enlazada lineal. De este modo, el número de hijos puede crecer dinámicamente según las necesidades de la aplicación. La Fig. 22 muestra un ejemplo de árbol general y su representación usando esta idea.

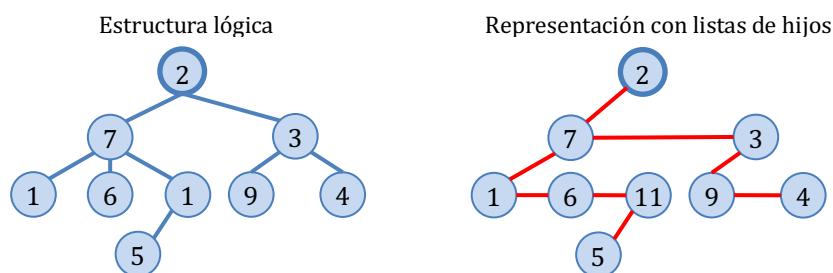


Fig. 22. Función que calcula el número de hojas de un árbol binario

Así pues, cada nodo del árbol es, al mismo tiempo, un posible padre y un posible hermano. Por lo tanto, cada nodo necesita tener dos punteros: uno que apunte al primero de sus hijos y otro que apunte al siguiente de sus hermanos. La raíz no tiene hermanos, pero el hecho de ser raíz es circunstancial. Un nodo raíz puede convertirse en hijo de otro nodo en cualquier momento, y a partir de ahí, podría tenerlos. Así que todos los nodos deben tener la misma estructura.

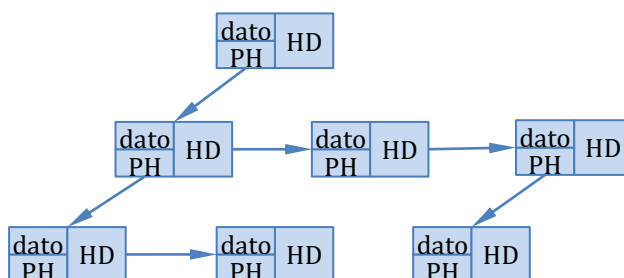


Fig. 23. Estructura de datos para representar un árbol general en C

Para definir los nodos de un árbol general en C se usa una estructura como la que aparece en la Fig. 24. El campo dato será del tipo de datos de los elementos que vaya a contener el árbol. Y los campos primerHijo y hermanoDerecho serán punteros a esta misma estructura. Y como ya se ha dicho, el primero apuntará al nodo que represente el primero de la lista de hijos del nodo actual. El segundo apuntará al nodo que represente el siguiente hermano del nodo actual.

Esta forma de representar un árbol general se denomina representación "hijo más a la izquierda - hermano derecho", o "primer hijo - hermano derecho". A diferencia de la usada para el árbol n-ario, aquí no se desperdicia memoria, pero a cambio, se necesita hacer una búsqueda lineal para localizar al hijo i-ésimo.

```
struct Nodo {
    int dato;
    struct Nodo * primerHijo;
    struct Nodo * hermanoDerecho;
};
typedef struct Nodo * ArbolGeneral;
```

Fig. 24. Definición de una estructura enlazada para representar árboles generales en C

## Operaciones con árboles generales en C

Como ya se ha visto para el caso de los árboles binarios, los algoritmos de recorrido en preorden y postorden son la base sobre la que se construyen las soluciones a todos los problemas. De hecho, el recorrido en inorden sólo tiene aplicación en árboles binarios.

La forma natural de implementar los recorridos en preorden y postorden en el caso de los árboles generales implica escribir un bucle secuencial que procese la lista de hijos de cada nodo. La Fig. 25 muestra cómo imprimir todos los elementos de un árbol general usando ambos algoritmos.

```
void preorden( ArbolGeneral a ) {  
    if ( a == NULL ) return;  
    printf( "%d ", a->dato );  
    ArbolGeneral aux = a->primerHijo;  
    while ( aux != NULL ) {  
        preorden( aux );  
        aux = aux->hermanoDerecho;  
    }  
}
```

```
void postorden( ArbolGeneral a ) {  
    if ( a == NULL ) return;  
    ArbolGeneral aux = a->primerHijo;  
    while ( aux != NULL ) {  
        postorden( aux );  
        aux = aux->hermanoDerecho;  
    }  
    printf( "%d ", a->dato );  
}
```

Fig. 25. Procedimientos en C que recorren un árbol general en preorden y postorden

Sin embargo, hay una forma más sencilla de conseguir un recorrido en preorden que sirve en muchas situaciones. En un árbol general representado usando el esquema “primer hijo – hermano derecho” cada nodo tiene dos punteros, y el recorrido en preorden se puede hacer como si fuera un árbol binario.

```
void preorden( ArbolGeneral a ) {  
    if ( a == NULL ) return;  
    printf( "%d ", a->dato );  
    preorden( a->primerHijo );  
    preorden( a->hermanoDerecho );  
}
```

Fig. 26. Recorrido en preorden de un árbol general como si fueran un árbol binario

La función preorden de la Fig. 26 es equivalente a la de la Fig. 25. Por lo tanto, muchos de los problemas tienen una solución prácticamente idéntica a la usada con los árboles binarios.

El recorrido en preorden se aplica a problemas en los que la solución no dependa de la información almacenada en el árbol completo y, por tanto, merece la pena detener la recursión tan pronto se encuentre la solución. Por tanto, en estos casos conviene examinar la raíz antes de hacer lo propio con sus hijos para terminar lo antes posible. Un buen ejemplo es una búsqueda como la de la función existe.

```
int existe( ArbolGeneral a, int buscado ) {  
    if ( a == NULL ) return 0;  
    if ( a->dato == buscado ) return 1;  
    if ( existe( a->primerHijo, buscado ) ) return 1;  
    if ( existe( a->hermanoDerecho, buscado ) ) return 1;  
    return 0;  
}
```

Fig. 27. Función en C que comprueba la existencia de un elemento en un árbol general

Para liberar la memoria de un árbol general representado mediante el esquema “primer hijo – hermano derecho” hay que ser muy cuidadoso. No sólo hay que empezar por las hojas e ir ascendiendo, sino que cada lista de hijos debe liberarse en orden inverso, empezando por el último y terminando por el primero. El recorrido en postorden descrito en la Fig. 25 no actúa así. Empieza por las hojas y va ascendiendo, pero cada lista de hijos se procesa de izquierda a derecha. Por lo tanto, no sirve de modelo para escribir la función de liberación de memoria.

La función libera mostrada en la Fig. 28 aplica un esquema que recuerda al de postorden usado con los árboles binarios. Pero si se estudia con atención, se descubre que, en este caso, la llamada a libera con el hermano derecho produce una liberación recursiva de la lista de hermanos empezando por el último. Esto es justo lo que se necesitaba para no usar campos de nodos previamente liberados.

```
void libera( ArbolGeneral a ) {  
    if ( a == NULL ) return;  
    libera( a->primerHijo );  
    libera( a->hermanoDerecho );  
    free( a );  
}
```

**Fig. 28. Procedimiento en C que libera la memoria asociada a un árbol general**

En aquellos casos en los que la solución requiera examinar todo el árbol sin un orden concreto se puede usar preorden porque es más sencillo de implementar. Por ejemplo, al contar o hacer algo con los elementos que cumplan cierta propiedad, como en el caso de la función positivos.

```
int positivos( ArbolGeneral a ) {  
    if ( a == NULL ) return 0;  
    int n = 0;  
    if ( a->dato > 0 ) n = 1;  
    return n + positivos(a->primerHijo ) + positivos( a->hermanoDerecho );  
}
```

**Fig. 29. Procedimiento en C que cuenta el número de elementos positivos en un árbol general**

La copia y comparación de árboles generales también puede hacerse con la versión simplificada del recorrido en preorden. En la función copia, mostrada en la Fig. 30, se usa la función crea\_arbol. Pero en este caso, se supone que la función devuelve la dirección en memoria dinámica de un árbol general compuesto por un único nodo, que contendrá el dato pasado como parámetro a la función y tendrá el valor NULL en los campos primerHijo y hermanoDerecho.

```
ArbolGeneral copia( ArbolGeneral a ) {  
    if ( a == NULL ) return NULL;  
    ArbolGeneral nuevo = crea_arbol( a->dato );  
    nuevo->primerHijo = copia( a->primerHijo );  
    nuevo->hermanoDerecho = copia( a->hermanoDerecho );  
    return nuevo;  
}
```

**Fig. 30. Función que devuelve una copia un árbol general**

Para comparar dos árboles generales se usa una estrategia similar a la usada con los árboles binarios. Se comparan primero las raíces y, si son iguales se sigue comparando por sus descendientes. En este caso (Fig. 31) también se puede usar el algoritmo simplificado de recorrido en preorden visto anteriormente.



```

int compara( ArbolGeneral a, ArbolGeneral b ) {
    if ( a == NULL && b == NULL ) return 1;
    if ( a == NULL || b == NULL ) return 0;
    if ( a->dato != b->dato ) return 0;
    return compara( a->primerHijo, b->primerHijo ) &&
           compara( a->hermanoDerecho, b->hermanoDerecho );
}

```

**Fig. 31. Función que compara dos árboles generales**

Para calcular el número de hojas de un árbol general hay que explorar todo el árbol y el orden en que se haga no es relevante siempre que la exploración sea total. Por tanto, se puede usar un esquema similar al de la función positivos. En este caso, para determinar si un nodo es hoja, como muestra la Fig. 32, basta con comprobar que su campo primerHijo es nulo.

```

int numero_hojas( ArbolGeneral a ) {
    if ( a == NULL ) return 0;
    int n = 0;
    if ( a->primerHijo == NULL ) n = 1;
    return n + numero_hojas( a->primerHijo ) + numero_hojas( a->hermanoDerecho );
}

```

**Fig. 32. Función que calcula el número de hojas de un árbol general**

Sin embargo, cuando las operaciones a realizar en cada nodo, o el valor a obtener a partir de los calculados en los subárboles, dependen del nivel de profundidad al que esté cada nodo, es necesario procesar la lista de hijos de forma secuencial.

Por ejemplo, para calcular la altura de un árbol, ya sea general o binario, hay que sumar uno a la altura del mayor de sus subárboles, teniendo en cuenta que las hojas tienen altura cero.

Así pues, es necesario obtener la altura de cada subárbol recorriendo la lista de hijos de forma secuencial y aplicando a cada uno de forma recursiva la misma función. Finalmente, sumar uno a la mayor de las alturas registradas durante el recorrido.

```

int altura( ArbolGeneral a ) {
    if ( a == NULL ) return 0;
    if ( a->primerHijo == NULL ) return 0;
    int max = 0;
    ArbolGeneral aux = a->primerHijo;
    while ( aux != NULL ) {
        int h = altura( aux );
        if ( h > max ) max = h;
        aux = aux->hermanoDerecho;
    }
    return 1 + max;
}

```

**Fig. 33. Función que calcula la altura de un árbol general**



## Arboles Binarios de Búsqueda

Un árbol binario de búsqueda (ABB) es un caso particular de árbol binario en el que todos los nodos deben cumplir la denominada “Propiedad de los árboles binarios de búsqueda”. Esta propiedad establece que el elemento asociado a cualquier nodo debe ser mayor que todos los de su subárbol izquierdo y menor que todos los de su subárbol derecho.

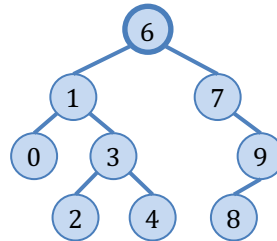


Fig. 34. Ejemplo de árbol binario de búsqueda

Esta organización de la información permite que las búsquedas e inserciones se puedan llevar a cabo muy eficientemente. Además, cuando se recorren en inorden se obtiene la lista de elementos ordenada de menor a mayor. En el árbol de la Fig. 34 el recorrido en inorden genera la secuencia: 0, 1, 2, 3, 4, 6, 7, 8, 9.

Para definir en C los nodos que representen un árbol binario de búsqueda se puede usar la misma estructura que para un árbol binario cualquiera. Pero para marcar la diferencia semántica existente se les puede asociar un nombre distinto: ABB.

```
struct Nodo {  
    int dato;  
    struct Nodo * hijoIzquierdo;  
    struct Nodo * hijoDerecho;  
};  
typedef struct Nodo * ABB;
```

Fig. 35. Definición de una estructura enlazada para representar árboles binarios de búsqueda en C

Gracias a la forma en que organizan la información, para **buscar un elemento en un árbol binario de búsqueda** se puede usar el algoritmo de la búsqueda binaria.

```
int existe( ABB a, int buscado ) {  
    if ( a == NULL ) return 0;  
    if ( buscado == a->dato ) return 1;  
    if ( buscado < a->dato ) return existe( a->hijoIzquierdo, buscado );  
    return existe( a->hijoDerecho, buscado );  
}
```

Fig. 36. Función que comprueba la existencia de un elemento en un árbol binario de búsqueda

Como puede verse en la Fig. 36, la traducción a código del algoritmo de la búsqueda binaria es directa. Si el árbol está vacío no puede contener el elemento buscado. Si el elemento buscado coincide con el de la raíz del árbol se termina con éxito. Y en caso negativo, se repite la búsqueda recursivamente en uno de los dos subárboles. La elección del subárbol depende del resultado de la comparación entre el elemento buscado y el de la raíz. Si la raíz es mayor que el elemento buscado, éste sólo puede estar en el subárbol izquierdo. En caso contrario, sólo tiene sentido buscarlo en el subárbol derecho.

La **inserción en un árbol binario de búsqueda** es una operación que tiene como objetivo añadir un nuevo elemento, pero al mismo tiempo, debe garantizar que todos los nodos del árbol resultante sigan cumpliendo la “Propiedad de los árboles binarios de búsqueda”.

Si se usa el algoritmo de búsqueda binaria para encontrar el elemento que se desea insertar, y este no se encuentra en el árbol, se llegará a un enlace con valor NULL. Reemplazando dicho enlace con un nuevo nodo que contenga el elemento a insertar se resuelve el problema. Esta es la idea del siguiente algoritmo recursivo:

- **Caso base:** La inserción en un árbol vacío se resuelve sustituyéndolo por un nuevo árbol cuya raíz contenga el elemento a insertar.
- **Caso general:** Si el elemento a insertar es menor que el de la raíz se inserta recursivamente en su hijo izquierdo. Si el elemento a insertar es mayor que el de la raíz se inserta recursivamente en su hijo derecho.

La Fig. 37 muestra una de las formas de implementar el algoritmo en C.

```
ABB inserta( ABB a, int dato ) {  
    if ( a == NULL ) return crea_arbol( dato );  
    if ( dato < a->dato ) a->hijoIzquierdo = inserta( a->hijoIzquierdo, dato );  
    if ( a->dato < dato ) a->hijoDerecho = inserta( a->hijoDerecho, dato );  
    return a;  
}
```

Fig. 37. Función que inserta un elemento en un árbol binario de búsqueda

El caso base de esta función recursiva se da cuando el árbol en el que hay que insertar es el vacío. En ese caso se crea un nuevo árbol consistente en un único nodo con el dato recibido como parámetro. La dirección de ese nodo se devuelve como resultado de la inserción. Es decir, la función devuelve el árbol resultante tras la inserción.

El caso general se trata comparando el nuevo elemento con el de la raíz del árbol recibido como parámetro. Esto permite determinar en cuál de los dos subárboles se debe producir la inserción. El subárbol elegido se actualiza pasando a ser el resultante de insertar el elemento en ese mismo subárbol. Finalmente, se devuelve el árbol que originalmente se recibió como parámetro, pues, en este caso, sigue siendo el mismo, ya que lo que habrá cambiado será alguno de sus subárboles.

Si el elemento a insertar ya se encuentra en el árbol no se realiza ninguna acción. Simplemente se devuelve el árbol recibido como parámetro.

Como con el resto de árboles, el manejo de un árbol binario de búsqueda se hace con un puntero externo que apunte a la raíz del árbol. En este caso, será una variable de tipo ABB. Inicialmente, cuando el árbol está vacío, dicha variable valdrá NULL. Tras la primera inserción el árbol tendrá un nodo que será la nueva raíz por lo que, en este caso, es necesario modificar la variable que representa el árbol. El resto de inserciones no modificará la raíz del árbol. Pero, lo habitual es actualizar la variable usada como argumento cada vez que se llama a inserta.

```
int main( int argc, char * argv[] ) {  
    ABB arbol = NULL;  
    for ( int i = 0 ; i < 10 ; i++ ) arbol = inserta( arbol, random() );  
}
```

Fig. 38. Ejemplo de uso de la función inserta para árboles binarios de búsqueda

La **supresión de elementos en árboles binarios de búsqueda** es una operación más compleja. También aquí es necesario garantizar que tras la eliminación de un elemento el árbol sigue cumpliendo la “Propiedad de los árboles binarios de búsqueda”. Sin embargo, mientras que una inserción siempre consiste en añadir una hoja al árbol, la supresión puede implicar la eliminación de nodos internos.

La supresión de un elemento situado en una hoja sí es un caso muy sencillo, pues basta con eliminar dicho nodo. También es sencillo eliminar un nodo que sólo tiene un hijo, pues basta con reemplazarlo por dicho hijo.

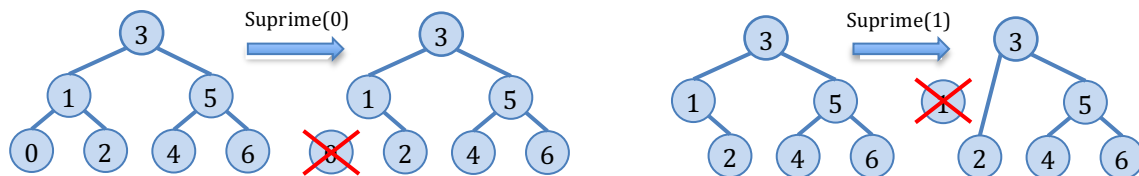


Fig. 39. Supresión de una hoja y de un nodo con un solo hijo en árboles binarios de búsqueda

Sin embargo, la eliminación de un nodo intermedio con dos hijos es una operación más compleja. La solución consiste en reemplazar el nodo eliminado por su sucesor y eliminar al sucesor. Esta sustitución consigue mantener la “Propiedad de los árboles binarios de búsqueda” porque el árbol no puede contener ningún elemento cuyo valor esté entre el eliminado y su sucesor.

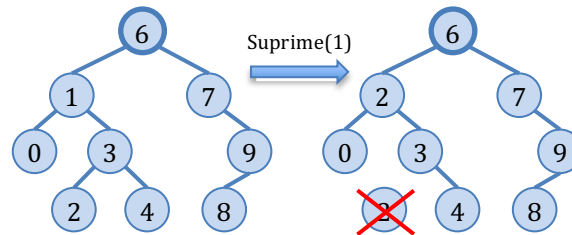


Fig. 40. Supresión de un nodo con dos hijos en árboles binarios de búsqueda

Teniendo en cuenta los tres casos mostrados anteriormente, el proceso de supresión se puede describir mediante el siguiente algoritmo recursivo:

- **Caso base 1:** Si el elemento a suprimir está en un nodo hoja, éste se sustituye por el árbol vacío.
- **Caso base 2:** Si el elemento a suprimir está en un nodo con un único hijo, éste se sustituye por su hijo.
- **Caso base 3:** Si el elemento a suprimir está en un nodo con dos hijos, éste se sustituye por su sucesor y se suprime este último.
- **Caso general:** Si el elemento a suprimir es menor que el de la raíz se suprime recursivamente en su hijo izquierdo. Si el elemento a suprimir es mayor que el de la raíz se suprime recursivamente en su hijo derecho.

El caso base 3 requiere la localización del sucesor de un elemento en el árbol. Afortunadamente, gracias a la organización de los datos en este tipo de árboles esta es una tarea sencilla. El sucesor del elemento situado en un nodo es el mínimo de los valores almacenados en su subárbol derecho. Y el valor mínimo de un árbol binario de búsqueda se puede encontrar recursivamente del siguiente modo:

- **Caso base:** El mínimo de un árbol sin hijo izquierdo está en su raíz.
- **Caso general:** El mínimo de un árbol con hijo izquierdo es el mínimo de su subárbol izquierdo.

La función de la Fig. 41 devuelve el nodo que contiene el valor mínimo de un ABB.

```
ABB minimo( ABB a ) {  
    if ( a == NULL ) return NULL;  
    if ( a->hijoIzquierdo == NULL ) return a;  
    return minimo( a->hijoIzquierdo );  
}
```

**Fig. 41. Función que devuelve el nodo con el valor mínimo de un árbol binario de búsqueda**

Como se puede ver, en lugar de devolver el valor mínimo se devuelve la dirección del nodo que lo contiene. De este modo, el código funciona independientemente del tipo de datos que contenga el árbol. Por otro lado, aunque la definición recursiva que se ha dado de mínimo no admite árboles vacíos, en el código sí se tratan para proteger a la función contra usos incorrectos.

La Fig. 42 muestra una implementación del algoritmo de supresión completo.

```
ABB suprime( ABB a, int dato ) {  
    if ( a == NULL ) return NULL;  
    if ( a->dato == dato ) {  
        if ( a->hijoIzquierdo == NULL ) {  
            ABB hijo = a->hijoDerecho;  
            free( a );  
            return hijo;  
        }  
        if ( a->hijoDerecho == NULL ) {  
            ABB hijo = a->hijoIzquierdo;  
            free( a );  
            return hijo;  
        }  
        ABB sucesor = minimo( a->hijoDerecho );  
        a->dato = sucesor->dato;  
        a->hijoDerecho = suprime( a->hijoDerecho, sucesor->dato );  
        return a;  
    }  
    if ( a->dato < dato ) {  
        a->hijoDerecho = suprime( a->hijoDerecho, dato );  
    }  
    else {  
        a->hijoIzquierdo = suprime( a->hijoIzquierdo, dato );  
    }  
    return a;  
}
```

The diagram on the right side of the code block uses blue curly braces to group the code into four cases:

- Casos base 1 y 2:** Groups the first two `if` blocks inside the `if ( a->dato == dato )` condition.
- Caso base 2:** Points to the second `if` block inside the `if ( a->dato == dato )` condition.
- Caso base 3:** Points to the third `if` block inside the `if ( a->dato == dato )` condition.
- Caso general:** Points to the final `if ( a->dato < dato )` block.

**Fig. 42. Función que suprime un elemento en un árbol binario de búsqueda**

La función comienza detectando si el árbol es vacío. Este no es un caso base del algoritmo, pero es muy útil para simplificar el código y protegerlo ante malos usos. Como en el caso de la función inserta, el resultado debe ser el árbol que quede tras hacer la supresión. Por tanto, los dos casos base 2 devuelven el hijo del nodo eliminado. Aunque parece que no hay un tratamiento explícito del caso base 1, realmente se trata junto al primero de los dos casos base 2, pues si el nodo a eliminar no tiene ningún hijo tampoco tiene hijo izquierdo. Y la solución que se da para el caso base 2 coincide con la que habría que dar para el caso base 1: eliminar el nodo y devolver NULL, que es el valor del hijo derecho cuando el nodo a eliminar es una hoja. El caso base 3 devuelve el mismo nodo recibido como raíz del árbol a procesar porque lo que hace es sustituir su valor y eliminar el nodo que tuviera su sucesor. Esta eliminación se hace con una nueva llamada a la propia función. Y el caso general se resuelve invocando a la función con el subárbol adecuado.

## Mejoras

Cuando se trabaja con un árbol binario de búsqueda puede ser necesario conocer el número de elementos que incluye. Este número se puede obtener haciendo un recorrido y contando todos los nodos, pero esta operación tiene un coste de  $O(n)$ . Cuando el número de elementos es muy grande y la frecuencia de uso de esta operación es alta, es mejor recurrir a soluciones como la siguiente, que requieren algo más de memoria por cada nodo, pero reducen el tiempo de ejecución.

Si se añade un campo de tipo entero a cada nodo, y se hacen unas simples modificaciones en el código de las funciones que insertan y suprimen elementos, se consigue que cada nodo mantenga continuamente actualizado el número de nodos incluidos en el árbol del que él es la raíz.

```
struct Nodo {  
    int dato;  
    struct Nodo * hijoIzquierdo;  
    struct Nodo * hijoDerecho;  
    int n;  
};  
typedef struct Nodo * ABB;
```

**Fig. 43. Modificación de la estructura de los nodos que representan árboles binarios de búsqueda en C**

La Fig. 43 muestra cómo queda la estructura que representa cada nodo con el nuevo campo, llamado *n*, usado para llevar la cuenta de nodos del árbol. Obviamente, habrá que modificar el código de la función que crea nodos ya que al crear un nuevo nodo será necesario darle a *n* el valor 1.

```
void actualiza( ABB a ) {  
    a->n = 1;  
    if ( a->hijoIzquierdo != NULL ) a->n += a->hijoIzquierdo->n;  
    if ( a->hijoDerecho != NULL ) a->n += a->hijoDerecho->n;  
}
```

**Fig. 44. Función que actualiza el número de elementos de un árbol binario de búsqueda**

La función *actualiza* mostrada en la Fig. 44 simplemente usa la información de los hijos para calcular la de su padre. Para mantener todo el árbol actualizado es necesario llamar a esta función cada vez que cambia la estructura y propagar los cambios hacia arriba hasta llegar a la raíz del árbol.

Afortunadamente, las funciones *inserta* y *suprime* se han diseñado de forma recursiva, y cuando realizan una modificación en el árbol, las llamadas recursivas terminan recorriendo todo el camino inverso hasta llegar a la raíz. Así pues, justo en ese momento se puede ir actualizando el valor de *n* en todos los nodos involucrados.

En concreto, en la función *inserta* hay que añadir la llamada a *actualiza* justo antes del último *return*. Y en la función *suprime* hay que añadirlo antes del *return* que termina el caso base 3 y antes del *return* del caso general. No es necesario hacer ninguna actualización en los casos base 1 y 2 puesto que se está eliminando el nodo. Y será al terminar la llamada recursiva anterior la que actualice su propio contador reflejando la desaparición de uno de sus hijos.

## Rendimiento de los árboles binarios de búsqueda

Los árboles binarios de búsqueda sirven para representar conjuntos ordenados de elementos como tablas de símbolos y diccionarios. Su característica principal es que permiten realizar búsquedas, inserciones y supresiones de una forma muy eficiente manteniendo los datos ordenados. Además, al recorrerlos en inorden se obtiene la lista completa de elementos ordenada de menor a mayor.

Combinan la capacidad para adaptar su tamaño dinámicamente en función de las necesidades de las estructuras enlazadas lineales, con la eficiencia de la búsqueda binaria sobre arrays previamente ordenados.

Las búsquedas en estructuras enlazadas lineales tienen un coste de  $O(n)$ , pues son estructuras que imponen un acceso secuencial. Buscar en un array ordenado es una operación eficiente porque se puede usar el algoritmo de la búsqueda binaria consiguiendo un tiempo de  $O(\log n)$ . En un árbol binario de búsqueda también se puede conseguir este nivel de eficiencia en las búsquedas. Pero, además, las inserciones y supresiones en un árbol binario de búsqueda son más eficientes que en las otras dos estructuras de datos.

La inserción y supresión en una estructura enlazada lineal en la que se quiera mantener el orden de sus elementos requiere un  $O(n)$ , el mismo coste que conlleva insertar y suprimir en un array ordenado. En el primer caso, es necesario recorrer la lista hasta encontrar la posición donde insertar. En el segundo, además es necesario desplazar los elementos mayores que el nuevo una posición a la derecha para dejar un hueco libre para el nuevo elemento, o para rellenar el hueco que deja al eliminarlo.

Sin embargo, todas estas ventajas sólo se obtienen cuando los árboles binarios de búsqueda están bien equilibrados, y la forma que toma un árbol binario de búsqueda depende del orden en el que se insertan los elementos. La Fig. 45 muestra dos árboles binarios de búsqueda con los mismos elementos, pero distribuidos de forma diferente. El de la izquierda está perfectamente equilibrado y el de la derecha no porque se han insertado en orden creciente.

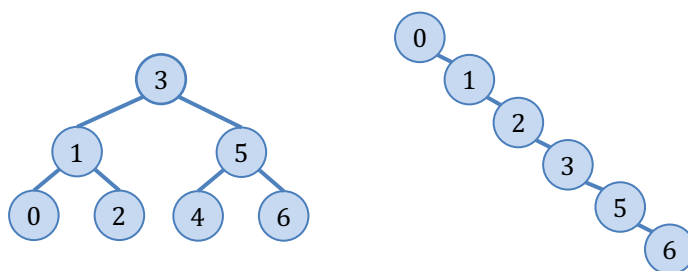


Fig. 45. Dos árboles binarios de búsqueda con los mismos elementos y distinta distribución

Mientras que las búsquedas e inserciones en el primero tienen un coste de  $O(\log n)$  en el segundo la eficiencia baja hasta  $O(n)$  puesto que, básicamente, se trata de una estructura enlazada lineal.

Así pues, resulta fundamental que un árbol binario de búsqueda esté equilibrado para poder aprovechar todas sus ventajas. En cursos superiores se estudian las estructuras de datos y los algoritmos necesarios para conseguirlo.