ÁRBOLES BINARIOS 2002

GRUPO # 22

Alumnos:

Aguilar Elba
Barrios Miguel
Camacho Yaquelin
Ponce Rodríguez Jhonny

ESTRUCTURAS DE DATOS

TEMA 6

Estructuras de datos no lineales. Árboles binarios

ÍNDICE

6.1. Introducción. Terminología básica y definiciones 3
Terminología básica: 4
6.2. Árboles binarios. Recorrido 5
6.2.1. El TAD Arbol binario 6
6.2.2. Implementaciones del Árbol binario 7
6.2.2.1. Implementación estática mediante un vector 7
6.2.2.2. Implementación dinámica mediante punteros 9
6.2.3. Recorrido de un Árbol binario 11
6.3. Árboles binarios de búsqueda 13
6.3.1. Búsqueda de un elemento 15
6.3.2. Inserción de un elemento 16
6.3.3. Eliminación de un elemento 19
6.4. Ejercicios 24

BIBLIOGRAFÍA

- (Joyanes y Zahonero, 1998), Cap. 9.
- (Joyanes y Zahonero, 1999), Cap. 8.
- (Dale y Lilly; 1989), Caps. 9 y 10.
- (Horowitz y Sahni, 1994), Cap. 5.

OBJETIVOS

- Conocer el concepto, funcionamiento y utilidad del tipo Árbol, así como toda la terminología asociada.
- Conocer el tipo abstracto de datos y sus operaciones asociadas.
- Conocer distintas implementaciones del TAD Árbol, tanto mediante vectores como mediante punteros.
- Conocer diversos algoritmos para el recorrido de árboles binarios: preorden, inorden y postorden.
- Conocer el concepto, utilidad y operaciones de manejo de árboles binarios de búsqueda.

6.1. Introducción. Terminología básica y definiciones

Al igual que ocurría con las estructuras de datos vistas en los temas anteriores, todo el mundo tiene claro el concepto de árbol, al menos en su aspecto botánico. Sin embargo, los árboles no son sólo eso de lo que estamos rodeados cuando nos perdemos en un bosque, sino que también se usan en otros muchos ámbitos. Así por ejemplo, todos hemos manejado alguna vez el concepto de árbol genealógico, o hemos visto clasificaciones jerárquicas como las del reino animal. En todos esos casos manejamos el concepto de árbol.

Centrándonos en el mundo de la informática, los árboles se utilizan en distintos ámbitos. Por ejemplo, al organizar la información para facilitar la búsqueda en un disco duro, utilizamos una estructura de directorios y subdirectorios en forma de árbol. También se usan los árboles asociados a distintos esquemas para el desarrollo de los algoritmos, tales como la programación dinámica, la ramificación y poda, el esquema divide y vencerás, etc.

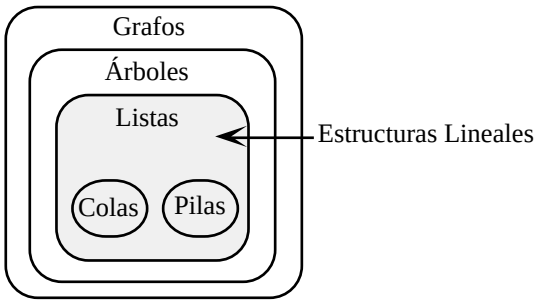
Si nos referimos a estructuras de datos, ya dijimos en el tema anterior que las pilas, colas y listas son estructuras lineales, puesto que en todas ellas cada elemento tiene un único elemento anterior y un único elemento posterior. Pero, además, existe estructuras de datos no lineales, en las que esta restricción desaparece. Esto es, en estas estructuras cada elemento puede tener varios anteriores y/o varios posteriores.

Definición

Un árbol es una estructura de datos no lineal y homogénea en el que cada elemento puede tener varios elementos posteriores, pero tan sólo puede tener un elemento anterior.

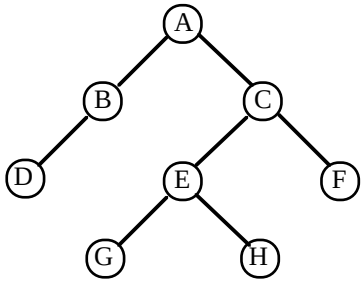
De hecho, podemos establecer una clasificación jerárquica de todos los tipos de datos que hemos visto, de modo que unos sean casos particulares de otros. Así, el tipo de estructura más general son los **grafos**. En un grafo cada elemento puede tener varios elementos anteriores y varios elementos posteriores. Los **árboles** no son más que un tipo especial de grafo en el que cada elemento puede tener varios posteriores, pero tan sólo puede tener un elemento anterior. Tanto grafos como árboles son estructuras no lineales.

Si añadimos a los árboles la restricción de que cada elemento puede tener un solo posterior, llegamos a las estructuras lineales, y más concretamente a las **listas**. Así pues, las listas no son más que un caso particular de los árboles. En este punto, si añadimos ciertas restricciones de acceso a las listas llegamos a las **colas** o a las **pilas**. Por lo tanto, tanto colas como pilas, son tipos particulares de listas. En definitiva, gráficamente podemos ver la relación entre las distintas estructuras comentadas del siguiente modo:

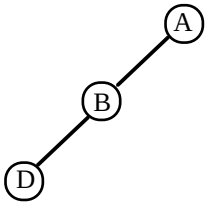


Terminología básica:

Asociados al concepto de árbol, existen toda una serie de términos que es necesario conocer para manejar esta clase de estructura de datos. Supongamos los siguientes ejemplos de árboles:



Ejemplo 1



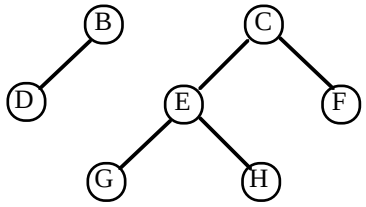
Ejemplo 2

Ejemplo 3

Veamos algunas definiciones básicas:

- *Nodo Padre* de un nodo N es aquel que apunta al mismo. En un árbol cada nodo sólo puede tener un padre. En el ejemplo 1, A es el padre de B y C, y a su vez, B es el padre de D.
- *Nodo Hijo* de otro nodo A es cualquier nodo apuntado por el nodo A. Un nodo puede tener varios hijos. En el ejemplo 1, B y C son los nodos hijos de A y todos los nodos tienen uno o dos hijos.
- *Nodo Raíz* es el único del árbol que no tiene padre. En la representación que hemos utilizado, el nodo raíz es el que se encuentra en la parte superior del árbol: A.
- *Hojas* son todos los nodos que no tienen hijos. En la representación del ejemplo 1 son hojas los nodos situados en la parte inferior: D, G, H y F.
- *Nodos Interiores* son los nodos que no son ni el nodo raíz, ni nodos hoja. En el ejemplo 1, son nodos interiores B, C y E.
- *Camino* es una secuencia de nodos, en el que dos nodos consecutivos cualesquiera son padre e hijo. En el ejemplo 1 A-B-D es un camino, al igual que E-G y C-E-H.
- *Rama* es un camino desde el nodo raíz a una hoja. En el ejemplo 1, A-C-E-G y A-C-F son ramas.

- *Altura* es el máximo número de nodos de las ramas del árbol. Dicho en otros términos, el máximo número de nodos que hay que recorrer para llegar de la raíz a una de las hojas. La altura del árbol del ejemplo 1 es 4, ya que esa es la longitud de la rama A-C-E-H, que junto a A-C-E-G son las dos más largas.
- *Grado* es el número máximo de hijos que tienen los nodos del árbol. Así, en el ejemplo anterior el árbol es de grado dos. Démonos cuenta de que una lista no es más que un árbol de grado uno, tal y como podemos ver en los ejemplos 2 y 3.
- *Nivel* de un nodo, es el número de nodos del camino desde la raíz hasta dicho nodo. En el árbol del ejemplo 1, A tiene nivel 1; B y C tienen nivel 2; D, E y F tienen nivel 3 y G y H tienen nivel 4.
- *Bosque* colección de dos o más árboles. Un ejemplo de bosque sería el siguiente:



6.2. Árboles binarios. Recorrido

Un tipo especial de árbol que se usa muy a menudo son los árboles binarios

Definición 1

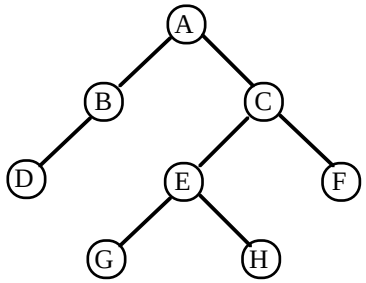
Un Árbol binario es un árbol de grado 2.

Definición 2

Un Árbol binario es aquel que

a) es vacío, ó

b) está formado por un nodo cuyos subárboles izquierdo y derecho son a su vez árboles binarios.



El árbol del ejemplo anterior es un árbol binario, ya que cada nodo tiene como máximo dos hijos. Démonos cuenta que en cualquier árbol, no sólo en los binarios, si eliminamos el nodo raíz, obtenemos dos árboles. Aquel que colgaba del enlace izquierdo del nodo raíz se denomina subárbol izquierdo y aquel que colgaba del enlace derecho se

denomina subárbol derecho. Además, en un árbol binario, todos los subárboles son también árboles binarios.

De hecho, a partir de cualquier nodo de un árbol podemos definir un nuevo árbol sin más que considerarlo como su nodo raíz. Por tanto, cada nodo tiene asociados un subárbol derecho y uno izquierdo.

Existen algunos tipos especiales de árboles binarios en función de ciertas propiedades. Así por ejemplo:

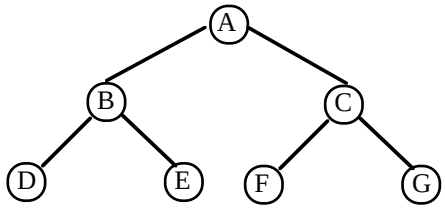
- *Árbol binario equilibrado* es aquel en el que en todos sus nodos se cumple la siguiente propiedad,

  $$| \text{altura(subárbol\_izquierdo)} - \text{altura(subárbol\_derecho)} | \leq 1.$$

  Así, el árbol del ejemplo 1 sería un árbol binario equilibrado, mientras el del ejemplo 2 no lo sería. En el segundo caso el subárbol izquierdo de A tiene una altura 2, mientras su subárbol derecho tiene una altura 0.

- *Árbol binario completo* es aquel en el que todos los nodos tienen dos hijos y todas las hojas están en el mismo nivel. Se denomina completo porque cada nodo, excepto las hojas, tiene el máximo de hijos que puede tener.

  En estos árboles se cumple que en el nivel k hay 2^{k-1} nodos y que, en total, si la altura es h, entonces hay $2^h - 1$ nodos.



  La figura anterior representa un árbol binario completo. En el nivel 1 tenemos $2^0 = 1$ nodos, en el nivel 2 tenemos $2^1 = 2$ nodos y en el nivel 3 tenemos $2^2=4$ nodos. En total el árbol es de altura 3 y por tanto contiene $2^3-1 = 7$ nodos.

6.2.1. El TAD Arbol binario

En este apartado vamos a definir el Tipo Abstracto de Datos Árbol binario.

TAD: ArbolB

Operaciones:

`CrearArbol: → ArbolB`

Su resultado es la creación de un árbol binario vacío.

`ArbolVacio: ArbolB → Logico`

Devuelve cierto si el argumento de entrada es un árbol binario vacío, falso en caso contrario.

`ConstArbol: ArbolB x <tipobase> x ArbolB → ArbolB`

A partir de dos árboles binarios y de un valor de tipo base, el resultado es un nuevo árbol binario cuyo nodo raíz contiene el valor de tipo base y en el que los árboles originales pasan a ser sus subárboles izquierdo y derecho .

`SubIzq: ArbolB → ArbolB`
Dado un árbol binario devuelve su subárbol izquierdo.
`SubDer: ArbolB → ArbolB`
Dado un árbol binario devuelve su subárbol derecho.
`DatoRaiz: ArbolB → <tipobase>`
Dado un árbol binario devuelve la información almacenada en su nodo raíz.

Axiomas: ∀ A, B ∈ ArbolB, ∀ d ∈ TipoBase,

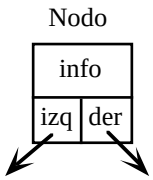
1) `ArbolVacio(CrearArbol) = verdadero`
2) `ArbolVacio(ConstArbol(A, d, B)) = falso`
3) `SubIzq(CrearArbol) = CrearArbol`
4) `SubIzq(ConstArbol(A, d, B) = A`
5) `SubDer(CrearArbol) = CrearArbol`
6) `SubDer(ConstArbol(A, d, B) = B`
7) `DatoRaiz(CrearArbol) = error`
8) `DatoRaiz(ConstArbol(A, d, B) = d`

6.2.2. Implementaciones del Árbol binario

Al igual que ocurre en el caso de las listas, podemos implementar un árbol binario mediante estructuras estáticas o mediante estructuras dinámicas. En ambos casos, cada nodo del árbol contendrá tres valores:

- La información de un tipobase dado contenida en el nodo.
- Un enlace al hijo derecho (raíz del subárbol derecho)
- Un enlace al hijo izquierdo (raíz del subárbol izquierdo)

Gráficamente:



6.2.2.1. Implementación estática mediante un vector

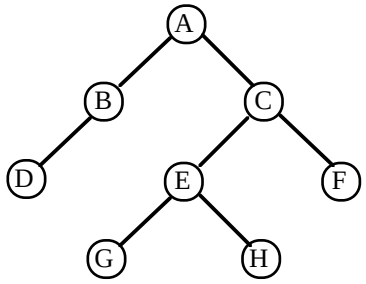
Para realizar la implementación estática del árbol binario utilizamos una estrategia similar a la usada en las listas, en las que simulábamos la memoria dinámica mediante el uso de vectores. Cada nodo es un registro con los tres campos especificados anteriormente y todos los nodos se almacenan en un vector de registros. Para implementar los enlaces utilizaremos números enteros que serán los índices en el vector donde se encuentran los hijos izquierdo y derecho.

Al igual que ocurre en el caso de las listas, las componentes vacías del vector se enlazan formando una lista. Para ello podemos usar como campo de enlace, tanto el correspondiente a los hijos izquierdos como a los derechos.

No vamos a detallar la implementación de todas las operaciones mediante vectores, sino que tan sólo mostraremos la estructura de datos en Pascal que usaremos para llevar a cabo esta implementación. En [Collado, Fernández y Moreno, 87] puede encontrarse un desarrollo más completo de esta implementación:

```
CONST
  MAX = ... {Tamaño del vector y máximo número de nodos del árbol}
TYPE
  Posicion = 0 .. MAX;
  Elemento = RECORD
               info: <TipoBase>;
               der, izq: Posicion
             END;
  TipoArbolB= RECORD
               raiz, vacios: Posicion;
               mem : ARRAY [1..MAX] OF Elemento
             END;
```

Por ejemplo, la representación mediante esta estructura estática del siguiente árbol



vendría dada por el siguiente vector de registros:

raiz=1 vacios=5

	1	2	3	4	5	6	7	8	9	10	11	12	
	A		B	C		E		F	D	H	G		info
	3	0	9	6	12	11	2	0	0	0	0	7	izq
	4		0	8		10		0	0	0	0		der

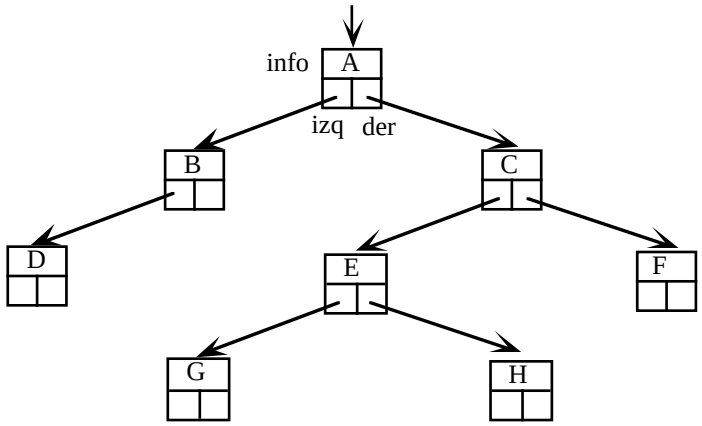
En el ejemplo anterior hemos utilizado el 0 para indicar un enlace que no apunta a ningún nodo. Las componentes del vector cuyos campos `izq` y `der` contienen un 0 son las hojas del árbol. Mientras la componente 2 es en este caso la última de la lista de componentes vacías.

6.2.2.2. Implementación dinámica mediante punteros

La representación de cada nodo en esta implementación será también un registro de tres campos, pero en este caso los enlaces serán punteros a los subárboles izquierdo y derecho de cada nodo. Por lo tanto, la estructura de datos en Pascal para definir un árbol binario será la siguiente:

```
TYPE
  TArbol = ^Nodo;
  Nodo = RECORD
              info: <tipobase>;
              izq, der: TArbol
         END;
```

De este modo, un árbol se identifica con un puntero a su nodo raíz, a través del cual podemos acceder a sus distintos nodos.



Veamos ahora como se implementarían las distintas operaciones incluidas en el TAD Árbol Binario usando esta representación dinámica.

```
PROCEDURE CrearArbol (VAR A:TArbol);
BEGIN
  A := NIL
END;
```

Para crear un árbol simplemente hacemos que el puntero a su nodo raíz apunte a NIL.

```
FUNCTION ArbolVacio (A:TArbol): BOOLEAN;
BEGIN
  ArbolVacio := (A = NIL)
END;
```

Consideraremos que un árbol está vacío cuando el puntero a su nodo raíz apunte a NIL.

```
PROCEDURE ConstArbol (subi, subd: TArbol; d:<tipobase>;
                      VAR nuevo: TArbol);
BEGIN
  new(nuevo);
  nuevo^.izq := subi;
  nuevo^.der := subd;
  nuevo^.info := d
END;
```

Tal y como hemos definido la operación de construcción de un nuevo árbol, esta se realiza a partir de valor de tipo base y dos subárboles. Se crea un nuevo nodo al que se le asigna el valor pasado como argumento. Los dos subárboles pasan a ser el subárbol derecho e izquierdo del nuevo nodo. El nuevo nodo se convierte en la raíz del árbol recién creado.

```
PROCEDURE SubIzq (A: TArbol; VAR subi:TArbol);
BEGIN
  subi := A^.izq
END;
```

Para acceder al subárbol izquierdo de un árbol, basta con acceder al puntero al hijo izquierdo de su nodo raíz.

```
PROCEDURE SubDer (A: TArbol; VAR subd: TArbol);
BEGIN
  subd := A^.der
END;
```

Para acceder al subárbol derecho de un árbol, basta con acceder al puntero al hijo derecho de su nodo raíz.

```
PROCEDURE DatoRaiz (A: TArbol; VAR d: <tipobase>);
BEGIN
  d := A^.info
END;
```

Para acceder al dato contenido en el nodo raíz de un árbol, basta con acceder al campo `info` del registro que lo representa.

Con la implementación elegida, y tal y como se definió la operación `ConstArbol`, los nodos hoja se caracterizarán por tener los punteros al subárbol izquierdo y al subárbol derecho con valor NIL.

Dado que en esta implementación hemos definido el tipo `TArbol` como un puntero, Pascal permite que este sea devuelto por una función. De este modo, los procedimientos `ConstArbol`, `SubDer` e `SubIzq` podrían haberse implementado como funciones. Asimismo, si el tipo base de la información contenida en cada nodo es escalar, también podemos convertir el procedimiento `DatoRaiz` en una función.

6.2.3. Recorrido de un Árbol binario

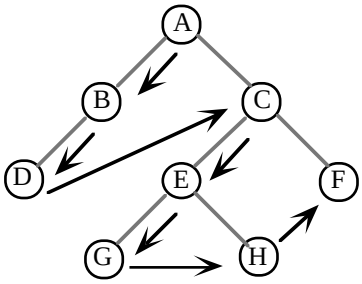
Recorrer un árbol consiste en acceder una sola vez a todos sus nodos. Esta operación es básica en el tratamiento de árboles y nos permite, por ejemplo, imprimir toda la información almacenada en el árbol, o bien eliminar toda esta información o, si tenemos un árbol con tipo base numérico, sumar todos los valores...

En el caso de los árboles binarios, el recorrido de sus distintos nodos se debe realizar en tres pasos:

- acceder a la información de un nodo dado,
- acceder a la información del subárbol izquierdo de dicho nodo,
- acceder a la información del subárbol derecho de dicho nodo.

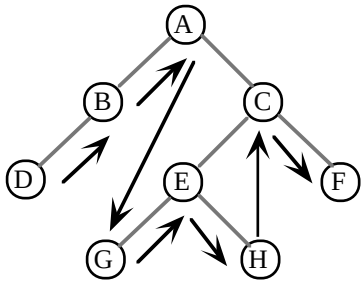
Imponiendo la restricción de que el subárbol izquierdo se recorre siempre antes que el derecho, esta forma de proceder da lugar a tres tipos de recorrido, que se diferencian por el orden en el que se realizan estos tres pasos. Así distinguimos:

- **Preorden**: primero se accede a la información del nodo, después al subárbol izquierdo y después al derecho.



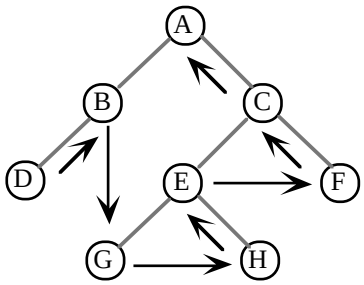
A - B - D - C - E - G - H - F

- **Inorden**: primero se accede a la información del subárbol izquierdo, después se accede a la información del nodo y, por último, se accede a la información del subárbol derecho.



D - B - A - G - E - H - C - F

- **Postorden**: primero se accede a la información del subárbol izquierdo, después a la del subárbol derecho y, por último, se accede a la información del nodo.



D - B - G - H - E - F - C - A

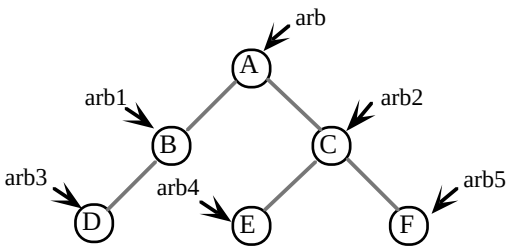
Si el nodo del que hablamos es la raíz del árbol, estaremos recorriendo todos sus nodos. Debemos darnos cuenta de que esta definición del recorrido es claramente recursiva, ya que el recorrido de un árbol se basa en el recorrido de sus subárboles izquierdo y derecho usando el mismo método. Aunque podríamos plantear una implementación iterativa de los algoritmos de recorrido, el uso de la recursión simplifica enormemente esta operación.

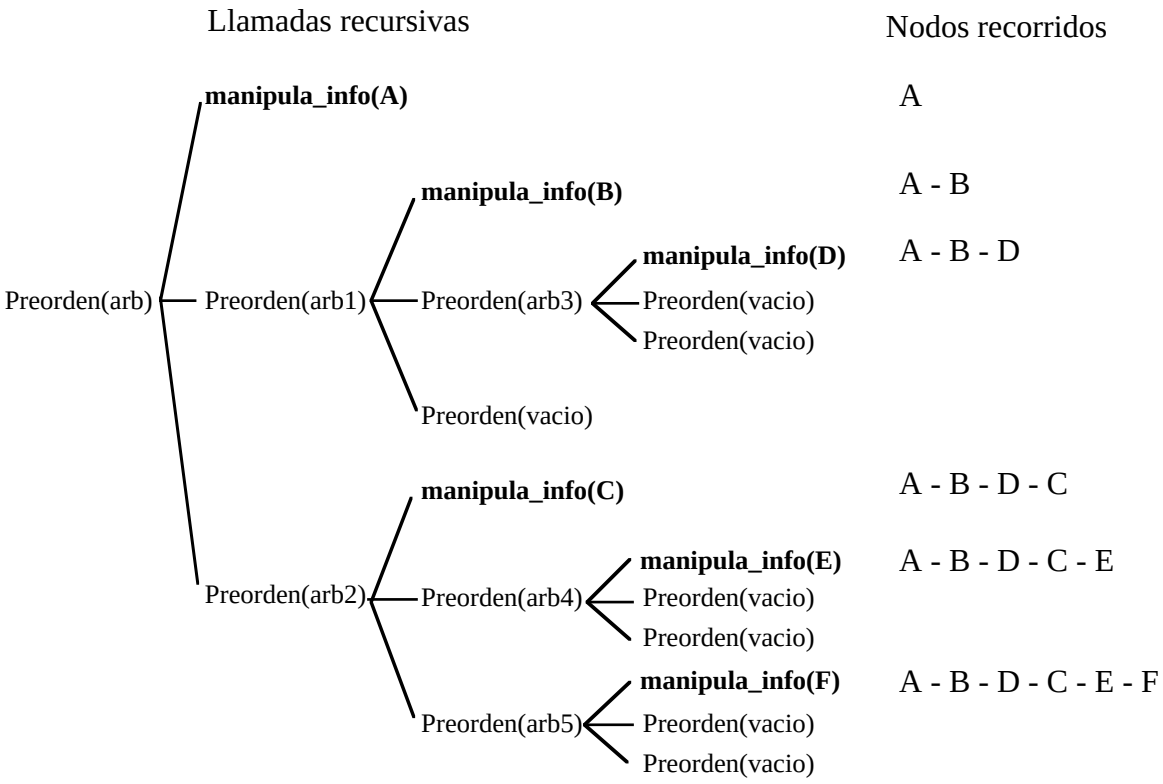
Así pues, utilizando la recursividad, podemos plantear la siguiente implementación de los tres tipos de recorrido descritos:

```
PROCEDURE Preorden (A: TArbol);
BEGIN
  IF (NOT ArbolVacio(A)) THEN
  BEGIN
    manipula_info(DatoRaiz(A));
    Preorden(SubIzq(A));
    Preorden(SubDer(A))
  END
END;

PROCEDURE Inorden (A: TArbol);
BEGIN
  IF (NOT ArbolVacio(A)) THEN
  BEGIN
    Inorden(SubIzq(A));
    manipula_info(DatoRaiz(A));
    Inorden(SubDer(A))
  END
END;

PROCEDURE Postorden (A: TArbol);
BEGIN
  IF (NOT ArbolVacio(A)) THEN
  BEGIN
    Postorden(SubIzq(A));
    Postorden(SubDer(A))
    manipula_info(DatoRaiz(A));
  END
END;
```

En esta implementación el recorrido de cada nodo se realiza mediante la operación `manipula_info`. Cambiando esta operación podemos por ejemplo escribir toda la información almacenada, sumar los posibles valores numéricos contenidos en los distintos nodos, etc.

Veamos como funciona por ejemplo el procedimiento recursivo Preorden mediante una traza. Para ello recorreremos el siguiente árbol, en el que hemos dado nombre a todos los subárboles que lo componen.





6.3. Árboles binarios de búsqueda

Imaginémonos que queremos encontrar un elemento en una lista ordenada. Para hacerlo deberemos recorrer sus elementos desde el primero hasta encontrar el elemento buscado o uno mayor que este. El coste medio de esta operación involucrará en un caso medio el recorrido y comparación de n/2 nodos, y un coste en el caso peor O(n). Si en lugar de utilizar una lista, estructuramos la información de modo adecuado en un árbol, podremos reducir el coste de la búsqueda a $O(\log_2 n)$.

Para hacernos una idea de lo que supone esta reducción del coste, supongamos que queremos encontrar un elemento entre 1000. Si almacenamos toda la información en una lista ordenada, esta búsqueda puede suponernos recorrer y comparar hasta 1000 nodos. Si esta misma información la almacenamos en un árbol binario de búsqueda, el coste máximo será de $\log_2(1000)<10$. Hemos reducido el coste de 1000 a 10 al cambiar la estructura de datos utilizada para almacenar la información.

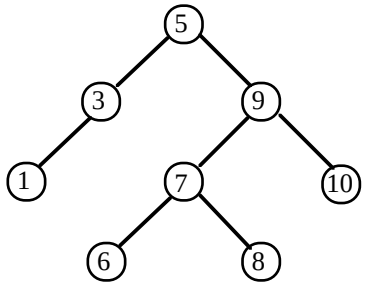
Tal y como hemos dicho, no basta con almacenar la información en un árbol para facilitar la búsqueda, debemos utilizar un tipo especial de árbol: un árbol binario de búsqueda. Si además queremos que esta búsqueda sea lo más eficiente posible debemos utilizar árboles de búsqueda binarios equilibrados.

Definición

Un árbol binario de búsqueda es una estructura de datos de tipo árbol binario en el que para todos sus nodos, el hijo izquierdo, si existe, contiene un valor menor que el nodo padre y el hijo derecho, si existe, contiene un valor mayor que el del nodo padre.

Obviamente, para establecer un orden entre los elementos del árbol, el tipo base debe ser escalar o debe tratarse de un tipo compuesto con una componente que actúe como clave de ordenación.

La siguiente figura es un ejemplo de árbol binario de búsqueda conteniendo enteros.



A continuación definiremos el Tipo Abstracto de Datos asociado a los árboles binarios de búsqueda. Para no alargar la descripción del mismo supondremos incluidas las operaciones del TAD `ArbolB`, tal y como aparecen en el apartado 6.2.1.

TAD: ArbolBB

Operaciones:

`CrearArbol:` → `ArbolBB`

Su resultado es la creación de un árbol binario vacío.

`ArbolVacio: ArbolBB` → `Logico`

Devuelve cierto si el argumento de entrada es un árbol binario vacío, falso en caso contrario.

`BuscarNodo: ArbolBB x <tipobase>` → `ArbolBB`

A partir de un árbol binario de búsqueda y de un valor de tipo base, el resultado es el árbol binario de búsqueda cuyo nodo raíz contiene dicho valor.

`InsertarNodo: ArbolBB x <tipobase>` → `ArbolBB`

A partir de un árbol binario de búsqueda y de un valor de tipo base, el resultado es un nuevo árbol binario de búsqueda en el que se ha añadido el nuevo valor (es decir, la inserción es ordenada).

`EliminarNodo: ArbolBB x <tipobase>` → `ArbolBB`

A partir de un árbol binario de búsqueda y de un valor de tipo base, el resultado es un nuevo árbol binario de búsqueda en el que se ha eliminado el nodo que contenía la información de tipo base.

En este TAD la representación axiomática de la semántica de algunas de las operaciones es muy complicada, por lo que nos basaremos en su descripción informal y pasaremos a su implementación.

Existen diversas implementaciones de las operaciones descritas. Por un lado podemos distinguir las implementaciones recursivas de las iterativas. Por otro lado, podemos intentar que las operaciones nos lleven siempre a árboles equilibrados o no. El mantener el equilibrado en los árboles complica bastante la implementación, por lo que no vamos a exigir esta condición en el árbol resultado de las operaciones.

Así, `InsertarNodo` siempre es insertar un nodo hoja en el sitio que toque para que siga ordenado y `EliminarNodo` siempre sustituye el nodo eliminado por otro que mantiene el orden.

6.3.1. Búsqueda de un elemento

La operación de búsqueda en un árbol binario de búsqueda es bastante sencilla de entender. Supongamos que buscamos un elemento x en el árbol. Lo primero que haremos será comprobar si se encuentra en el nodo raíz. Si no es así, si el elemento buscado es menor que el contenido en el nodo raíz sabremos que, de estar en el árbol, se encuentra en el subárbol izquierdo. Si el elemento buscado es mayor que el contenido en el nodo raíz sabremos que, de estar en el árbol, se encuentra en el subárbol derecho. Para continuar la búsqueda en el subárbol adecuado aplicaremos recursivamente el mismo razonamiento. Por lo tanto, el esquema del algoritmo `BuscarNodo` será el siguiente:

1. Si el valor del nodo actual es igual al valor buscado, lo hemos encontrado.
2. Si el valor buscado es menor que el del nodo actual, deberemos inspeccionar el subárbol izquierdo.
3. Si el valor buscado es mayor que el del nodo actual, deberemos inspeccionar el subárbol derecho.

A continuación mostramos una versión iterativa de esta operación. Se deja como ejercicio la implementación de una versión recursiva equivalente.

```
FUNCTION BuscarNodo (A: TArbol; x:<tipobase>):TArbol;
VAR
  p: TArbol;
  enc: BOOLEAN;
BEGIN
  p := A;
  enc := false;
  WHILE (NOT enc) AND (NOT ArbolVacio(p)) DO
  BEGIN
    enc := (DatoRaiz(p) = x);
    IF NOT enc THEN
      IF (x < DatoRaiz(p)) THEN
        p := SubIzq(p)
      ELSE
        p := SubDer(p)
  END;
  BuscarNodo := p
END;
```

Si al acabar la operación el resultado devuelto es NIL, interpretaremos que no hemos encontrado el elemento buscado.

6.3.2. Inserción de un elemento

La operación de inserción de un nuevo nodo en un árbol binario de búsqueda consta de tres fases básicas:

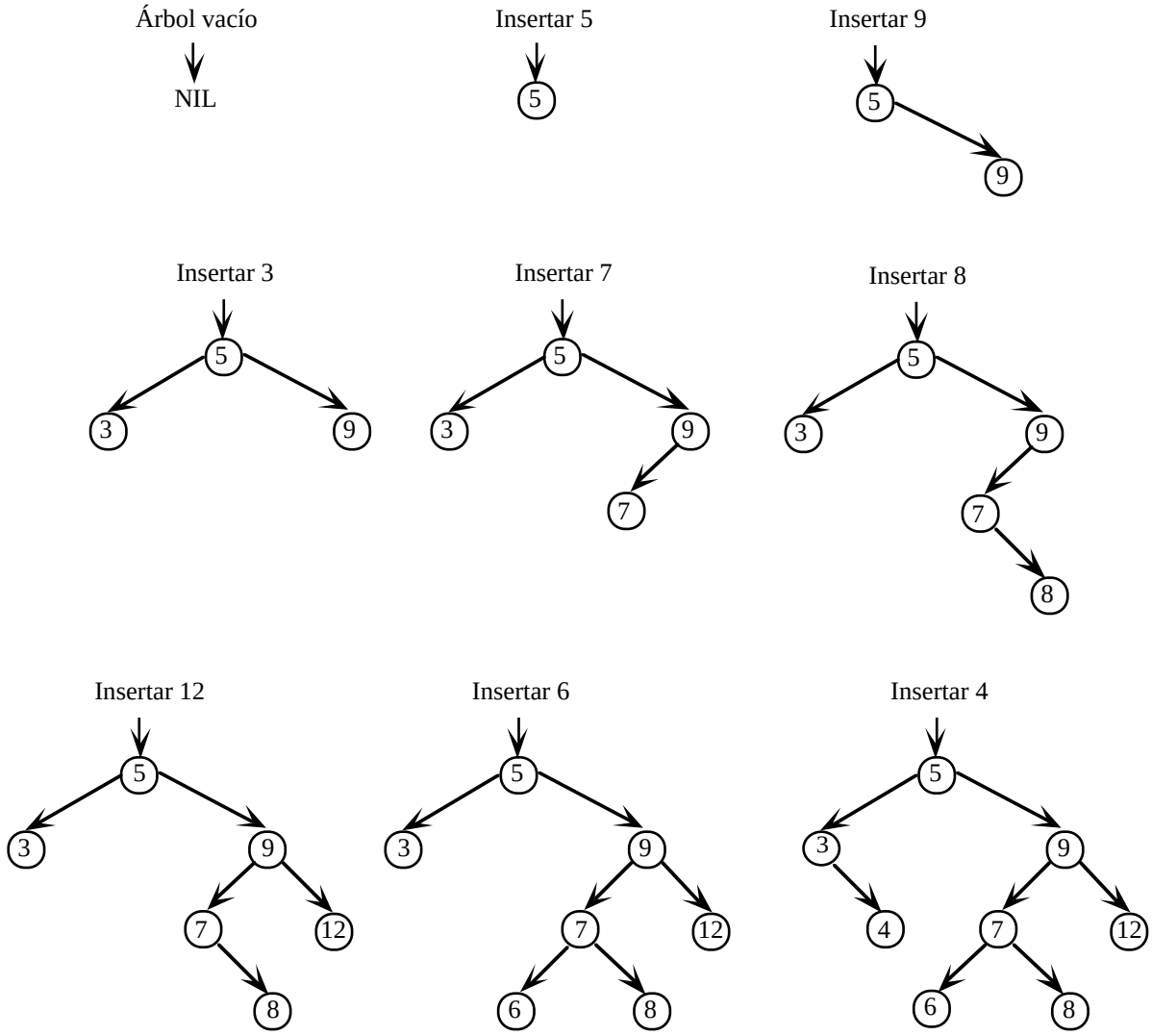
1. Creación del nuevo nodo
2. Búsqueda de su posición correspondiente en el árbol. Se trata de encontrar la posición que le corresponde para que el árbol resultante siga siendo de búsqueda.
3. Inserción en la posición encontrado. Se modifican de modo adecuado los enlaces de la estructura.

La creación de un nuevo nodo supone simplemente reservar espacio para el registro asociado y rellenar sus tres campos.

Dado que no nos hemos impuesto la restricción de que el árbol resultante sea equilibrado, consideraremos que la posición adecuada para insertar el nuevo nodo es la hoja en la cual se mantiene el orden del árbol. Insertar el nodo en una hoja supone una operación mucho menos complicada que tener que insertarlo como un nodo interior y modificar la posición de uno o varios subárboles completos.

La inserción del nuevo nodo como una hoja supone simplemente modificar uno de los enlaces del nodo que será su padre.

Veamos con un ejemplo la evolución de un árbol conforme vamos insertando nodos siguiendo el criterio anterior respecto a la posición adecuada.



El siguiente código implementa de modo iterativo la operación de inserción de un nodo siguiendo la descripción anterior.

```
PROCEDURE InsertarNodo (VAR A: TArbol; x:<tipobase>);
VAR
  p, aux, padre_aux: TArbol;
BEGIN
  { Crear el nuevo nodo }
  new(p);
  p^.info := x;
  p^.izq := NIL;
  p^.der := NIL;
  IF ArbolVacio(A) THEN
    A := p
  ELSE
```

```
    BEGIN
      { Buscar el lugar que le corresponde }
      aux := A;
      WHILE NOT ArbolVacio(aux) DO
      BEGIN
        padre_aux := aux;
        IF x <= DatoRaiz(aux) THEN
          aux := SubIzq(aux)
        ELSE
          aux := SubDer(aux)
      END;
      { Insertar el nuevo nodo }
      IF x <= DatoRaiz(padre_aux) THEN
        padre_aux^.izq := p
      ELSE
        padre_aux^.der := p
    END
  END;
```

En el algoritmo podemos diferenciar claramente las tres fases. Para acabar de entenderlo son necesarios algunos comentarios sobre su código.

En primer lugar, si el árbol pasado como argumento está vacío, el árbol resultante tan sólo contiene el nuevo nodo creado, es decir, es un puntero al mismo.

Para buscar la posición adecuada donde insertar el nuevo nodo recorremos el árbol desde su raíz hasta encontrar su posición como hoja que mantenga el orden. Para este recorrido manejaremos un puntero `aux` que ira recorriendo una rama del árbol. Durante este recorrido, cuando nos encontremos en un nodo cualquiera, pasaremos a su hijo izquierdo si el valor a insertar es menor que el del nodo actual, y pasaremos a un hijo derecho si el valor a insertar es mayor que el del nodo actual.

Consideremos que hemos llegado al final del recorrido cuando alcancemos una hoja. Si consideramos el nodo actual como raíz de un árbol, esta condición se dará cuando el subárbol al que pretendemos acceder esté vacío.

En todo momento durante el recorrido debemos mantener no sólo un puntero al nodo actual, `aux`, sino también un puntero a su padre, `padre_aux`. Esto es así porque para insertar el nuevo nodo debemos enlazarlo con su padre, y esto sólo es posible desde el nodo padre.

Una vez encontrada la posición adecuada para la inserción, enlazaremos el nuevo nodo con su padre utilizando el puntero adecuado en función de su valor.

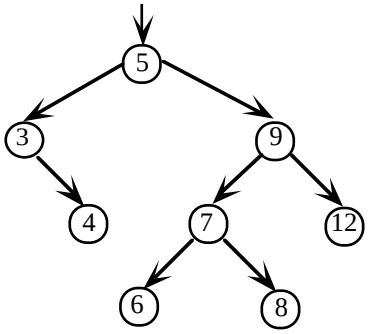
6.3.3. Eliminación de un elemento

La eliminación de un nodo de un árbol binario de búsqueda es más complicada que la inserción, puesto que puede suponer la recolocación de varios de sus nodos. En líneas generales un posible esquema para abordar esta operación es el siguiente:

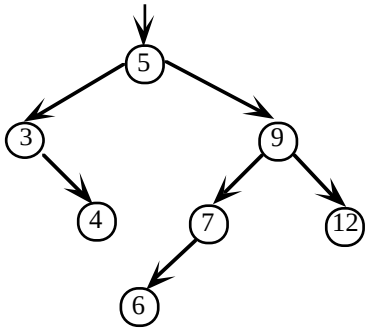
1. Buscar el nodo que se desea borrar manteniendo un puntero a su padre.
2. Si se encuentra el nodo hay que contemplar tres casos posibles:
   a. Si el nodo a borrar no tiene hijos, simplemente se libera el espacio que ocupa
   b. Si el nodo a borrar tiene un solo hijo, se añade como hijo de su padre, sustituyendo la posición ocupada por el nodo borrado.
   c. Si el nodo a borrar tiene los dos hijos se siguen los siguientes pasos:
      i. Se busca el máximo de la rama izquierda o el mínimo de la rama derecha.
      ii. Se sustituye el nodo a borrar por el nodo encontrado.

Veamos gráficamente varios ejemplos de eliminación de un nodo:

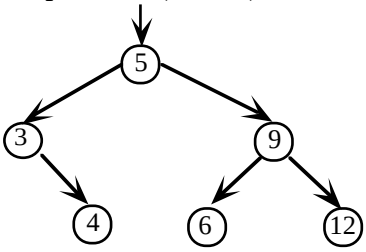
Árbol Inicial



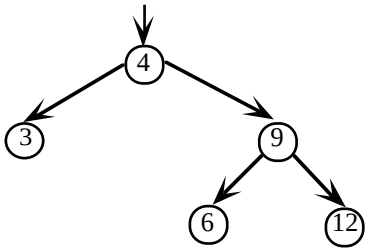
Suprimir 8 (caso a)



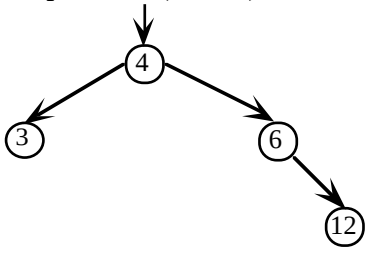
Suprimir 7 (caso b)



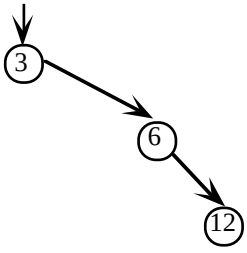
Suprimir 5 (caso c)



Suprimir 9 (caso c)



Suprimir 4 (caso c)



El siguiente código representa una posible implementación de esta operación

```
PROCEDURE EliminarNodo(VAR A:TArbol; x:<tipobase>; VAR enc:BOOLEAN);
VAR
  p, padre_p, sust, p_sust: TArbol; enc:BOOLEAN;
BEGIN
  { Busqueda del elemento a eliminar }
  p := A;
  enc := false;
  WHILE (NOT enc) AND (NOT ArbolVacio(p)) DO
  BEGIN
    enc := (DatoRaiz(p) = x);
    IF NOT enc THEN
    BEGIN
      padre_p := p;
      IF (x <= DatoRaiz(p)) THEN
        p := SubIzq(p)
      ELSE
        p := SubDer(p)
    END
  END;
  { Eliminacion del nodo si se ha encontrado }
  IF enc THEN
  BEGIN
    IF ArbolVacio(SubIzq(p)) THEN {a ó b - sin hijos o el derecho}
      sust := SubDer(p)
    ELSE IF ArbolVacio(SubDer(p)) THEN { caso b - un solo hijo }
      sust := SubIzq(p)
    ELSE
    BEGIN { caso c - nodo con los dos hijos }
      p_sust := p;
      sust := SubIzq(p);
      WHILE NOT ArbolVacio(SubDer(sust)) DO
      BEGIN
        p_sust := sust;
        sust := SubDer(sust)
      END;
      IF p_sust = p THEN
        p_sust^.izq := SubIzq(sust)
      ELSE
        p_sust^.der := SubIzq(sust);
      sust^.izq := SubIzq(p);
      sust^.der := SubDer(p);
    END;
```

```
      IF p=A THEN
        A := sust
      ELSE IF (p = SubIzq(padre_p)) THEN
        padre_p^.izq := sust
      ELSE
        padre_p^.der := sust;
      dispose(p)
    END
END;
```

En el procedimiento anterior podemos diferenciar claramente los dos pasos básicos de que consta la eliminación de un nodo. En el primer paso, para buscar el nodo que queremos eliminar, utilizamos dos punteros: un puntero `p` que apunta al nodo cuyo contenido estamos comprobando y otro `p_padre` que apunta a su nodo padre. Este segundo puntero nos permitirá mantener la conexión dentro del árbol una vez eliminado el nodo. Si salimos del WHILE por la condición de `ArbolVacio`, significa que no hemos encontrado el nodo a eliminar, y en ese caso no pasamos a la segunda fase del algoritmo.

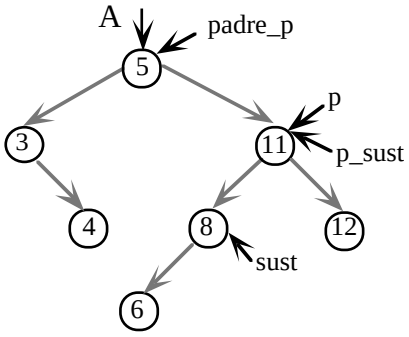
Durante la fase de eliminación del nodo hemos diferenciado los distintos casos. Utilizamos un puntero auxiliar `sust` que apuntará al nodo sustituto del eliminado, es decir, a aquel que ocupará su posición. Si el subárbol izquierdo del nodo a eliminar está vacío, el sustituto será su hijo derecho, mientras que si el subárbol derecho del nodo a eliminar está vacío, el sustituto será su hijo izquierdo. Si no se cumple ninguna de las dos condiciones anteriores, el nodo a sustituir tiene nodos en sus dos subárboles y su eliminación será más compleja. En todo caso, al finalizar el algoritmo, hemos de enlazar el padre del nodo eliminado con el nodo sustituto y liberar la memoria ocupada por el nodo suprimido. Para ello se usa el siguiente código:

```
IF p=A THEN
  A := sust
ELSE IF (p = SubIzq(p_padre)) THEN
  p_padre^.izq := sust
ELSE
  p_padre^.der := sust;
dispose(p)
```

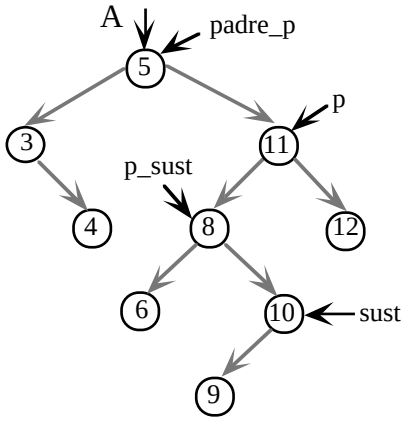
Veamos ahora el caso en el que el nodo a eliminar tiene nodos en sus dos subárboles (caso c). En esta situación, elegiremos como sustituto al nodo con mayor valor de su subárbol izquierdo. Este nodo será el situado más a la derecha de este subárbol. Para buscarlo, comenzaremos por desplazarnos al hijo izquierdo del nodo a eliminar y a partir de este punto nos desplazaremos siempre a sucesivos hijos derechos, mientras estos existan. El código utilizado para llevar a cabo este proceso es el siguiente:

```
p_sust := p;
sust := SubIzq(p);
WHILE NOT ArbolVacio(SubDer(sust)) DO
BEGIN
  sust := SubDer(sust);
  p_sust := sust
END;
```

Como vemos en el código anterior, mantenemos un puntero al nodo sustituto, `sust`, y un puntero a su padre, `p_sust`. Veamos gráficamente un par de ejemplos de como quedarían los distintos punteros auxiliares en este caso.



Ejemplo 1

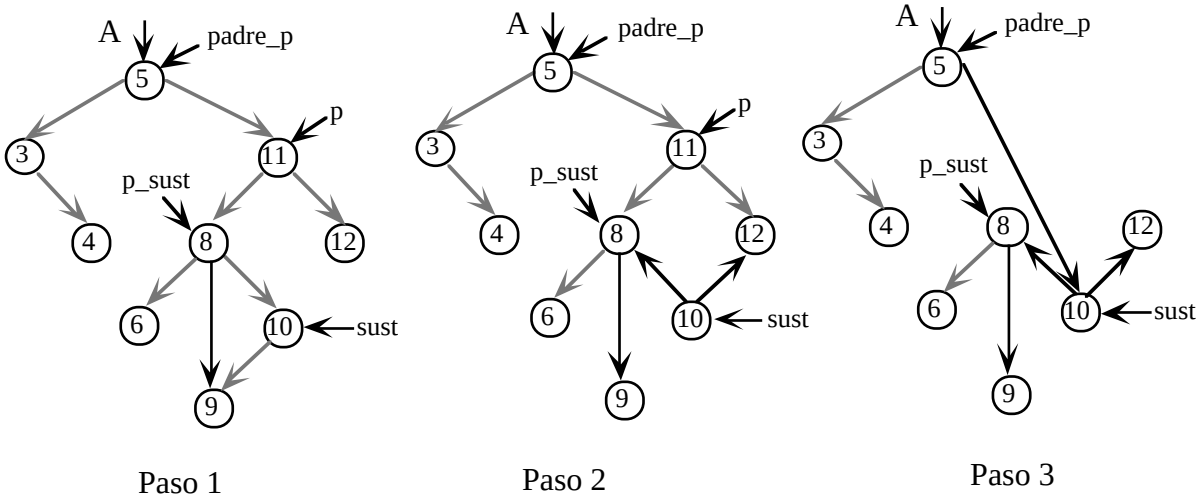


Ejemplo 2

Una vez localizados tanto el nodo a eliminar y su nodo padre, como el nodo sustituto y su nodo padre, podemos ya realizar la sustitución. Para ello comenzaremos por salvaguardar el posible subárbol izquierdo del nodo sustituto. Por la forma en la que lo hemos encontrado, no tendrá subárbol derecho. En los ejemplos anteriores, el subárbol a salvaguardar estará formado por el nodo 6 en el ejemplo 1 y por el nodo 9 en el ejemplo 2.

En el ejemplo 1, cuando el nodo sustituto es hijo del eliminado (`p_sust=p`), el subárbol a salvaguardar deberá colgarse de la rama izquierda de `p_sust`. En el ejemplo 2, cuando no se cumple la condición anterior, el subárbol a salvaguardar deberá colgarse de la rama derecha de `p_sust`, tal y como se ve en el paso 1 de la siguiente figura. El código utilizado para realizar este enlace es el siguiente:

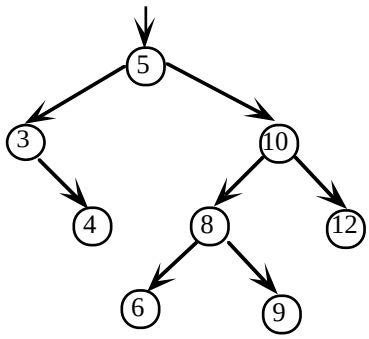
```
IF p_sust = p THEN
  p_sust^.izq := SubIzq(sust)
ELSE
  p_sust^.der := SubIzq(sust);
```



Para finalizar la sustitución, deberemos colgar del nodo sustituto los subárboles del nodo eliminado, tal y como se ve en el paso 2 de la figura anterior. Esto se hace con el siguiente código.

```
sust^.izq := SubIzq(p);
sust^.der := SubDer(p);
```

Como hemos comentado anteriormente, los últimos pasos llevados a cabo por el algoritmo son el enlace del nodo padre del eliminado con el nodo sustituto, y la liberación del espacio ocupado por el nodo eliminando. Estas operaciones constituyen el paso 3 de la figura anterior. Si lo redibujamos de modo más adecuado, el árbol resultante tras eliminar el nodo 11 quedará del siguiente modo:



6.4. Ejercicios

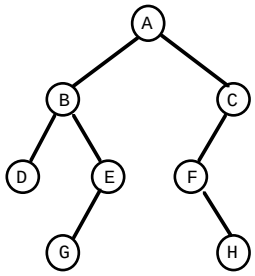
1. Dado el siguiente algoritmo, responde a las siguientes preguntas:

```
PROCEDURE Misterio (A: TArbol);
VAR
  aux: TArbol;
BEGIN
  IF not ArbolVacio(A) THEN
  BEGIN
    Misterio(SubIzq(A));
    Misterio(SubDer(A));
    dispose(A)
  END
END;
```

   a) ¿Qué tipo de recorrido realiza el algoritmo anterior?

   b) ¿Qué hace el algoritmo?

   c) Explica la ejecución del algoritmo, indicando el orden en que se efectúa, en el caso de que se le dé como parámetro el siguiente árbol:



2. Dado el siguiente algoritmo, responde a las siguientes preguntas:

```
FUNCTION Misterio (A: TArbol; m: TIPOBASE): TArbol;
VAR
  aux: TArbol;
BEGIN
  IF ArbolVacio(A) THEN
    CrearArbol(aux)
  ELSE
    IF m = DatoRaiz(A) THEN
      aux:= A
    ELSE
    BEGIN
      aux:= Misterio(SubIzq(A), m);
      IF ArbolVacio(aux) THEN
        aux:= Misterio(SubDer(A), m)
```

```
    END;
  Misterio:= aux
END;
```

a) ¿Qué tipo de recorrido realiza el algoritmo anterior?

b) ¿Qué hace el algoritmo?

c) Se puede hacer lo mismo con otro tipo de recorrido pero, ¿qué ventajas o inconvenientes tienen?

3. Una vez estudiado el algoritmo MISTERIO, responde y razona las siguientes preguntas:

```
PROCEDURE misterio(Datos:TArbol;VAR Resultado:INTEGER);
VAR
  Aux: INTEGER;
BEGIN
  IF (NOT ArbolVacio(Datos)) THEN
  BEGIN
    misterio(SubIzq(Datos), Resultado);
    misterio(SubDer(Datos), Aux);
    IF Resultado < Aux THEN
      Resultado := Aux;
    Resultado:= Resultado+1
  END
END.
```

Este algoritmo intenta contar la longitud de la rama más larga del árbol.

a) ¿Qué tipo de recorrido hace el algoritmo MISTERIO?

b) ¿Qué error tiene el algoritmo? ¿Cómo lo arreglarías?

4. Dado el algoritmo,

```
PROCEDURE ¿Que_hace_esto? (v: VECTOR, n: INTEGER; VAR A: TArbol);
VAR
  p, padre, aux: TArbol;
  i : INTEGER;
BEGIN
  CrearArbol(A);
  New(p);
  p^.info:= v[1];
  p^.izq:= NIL;
  p^.der:= NIL;
  A:= p;
  FOR i:= 2 TO n DO
  BEGIN
    New(p);
```

```
        p^.info:= v[i];
        p^.izq:= NIL;
        p^.der:= NIL;
        aux:= A;
        WHILE NOT ArbolVacio(aux) DO
        BEGIN
          padre:= aux;
          IF v[i] < DatoRaiz(padre) THEN
            aux:= SubIzq(aux)
          ELSE
            aux:= SubDer(aux)
        END;
        IF v[i] < DatoRaiz(padre) THEN
          padre^.izq:= p
        ELSE
          padre^.der:= p
      END;
      Imprimir(A)
    END;
```

Se pide:

a) ¿Qué hace este algoritmo?

b) ¿Qué recorrido elegiríamos en el algoritmo **Imprimir**, si se desea que los valores de v se impriman ordenados y por qué?

5. Dado el algoritmo,

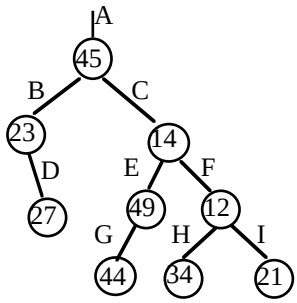
```
PROCEDURE Misterio (VAR A: TArbol);
  VAR
    aux: TArbol;
BEGIN
  IF NOT ArbolVacio(A) THEN
  BEGIN
    Misterio(SubDer(A));
    Misterio(SubIzq(A));
    aux:= SubDer(A);
    A^.der:= SubIzq(A);
    A^.izq:= aux
  END
END;
```

Se pide:

a) ¿Qué recorrido se realiza?

b) ¿Qué hace el algoritmo?

c) Realizar una traza con el siguiente árbol



6. Dado el algoritmo,

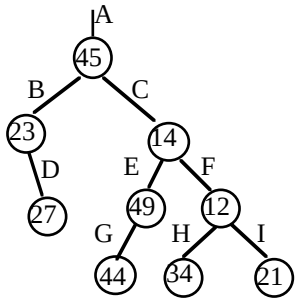
```
PROCEDURE Misterio (A:TArbol;VAR p:INTEGER);
VAR
  n, m:INTEGER;
BEGIN
  IF ArbolVacio(A) THEN
    p:= 0
  ELSE
  BEGIN
    Misterio(SubIzq(A),n);
    Misterio(SubDer(A),m);
    p:= 1 + n + m
  END
END;
```

Se pide:

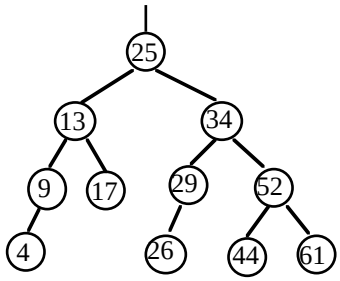
a) ¿Qué recorrido se realiza?

b) ¿Qué hace el algoritmo?

c) Realizar una traza con el siguiente árbol



7. Dado el árbol binario de búsqueda:



y siguiendo los algoritmos vistos en teoría,

a) Realizar una traza del Algoritmo **InsertarNodo**, si se pretende insertar el valor 27.

La traza debe indicar claramente qué valores toman las distintas variables auxiliares, así como cuál sería el árbol resultante.

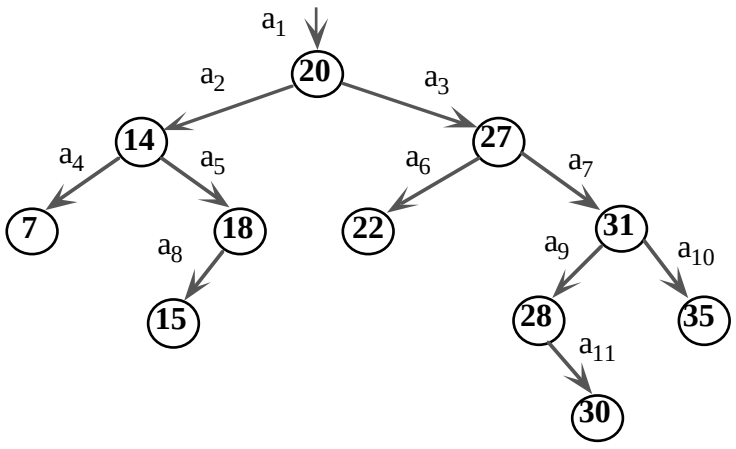
b) A partir del árbol obtenido en el apartado a), realizar una traza del Algoritmo **EliminarNodo**, si se pretende borrar el valor 34.

La traza debe indicar claramente qué valores toman las distintas variables auxiliares, así como cuál sería el árbol resultante.

8. Dada la operación EliminarNodo asociada a los árboles binarios de búsqueda,

a) Modificarla para que, en el caso en que el nodo a eliminar tenga dos hijos, el nodo sustituto sea el menor de su subárbol derecho.

b) Realizar una traza del algoritmo obtenido en el caso de que se quiera eliminar el nodo 27 del siguiente árbol.



Dibujar la posición de los distintos punteros tras localizar el nodo sustituto y redibujar el árbol tras eliminar el nodo.

9. Suponer que TArbol es un árbol binario con una implementación dinámica y TArbolB es un árbol binario con una implementación estática, tal y como se han definido en los apuntes de teoría. Dado el siguiente algoritmo:

```
PROCEDURE misterio(VAR A:TArbol; pos:INTEGER; est:TArbolB);
VAR
  elem:TArbol;
BEGIN
  IF (pos=0) THEN CrearArbol(A)
  ELSE BEGIN
    new(elem);
    elem^.info:=est.mem[pos].info;
    misterio(elem^.izq, est.mem[pos].izq, est);
    misterio(elem^.der, est.mem[pos].der, est);
    A:=elem
  END
END;
```

a) Realizar una traza de la llamada `misterio(A,est.raiz,est)`, en el caso en que el árbol `est` contenga la siguiente información:

`est.raiz=4`

`est.vacios=0`

`est.mem`

	1	2	3	4	5	6
info	4	12	1	3	25	5
izq	6	3	0	1	0	0
der	5	0	0	2	0	0

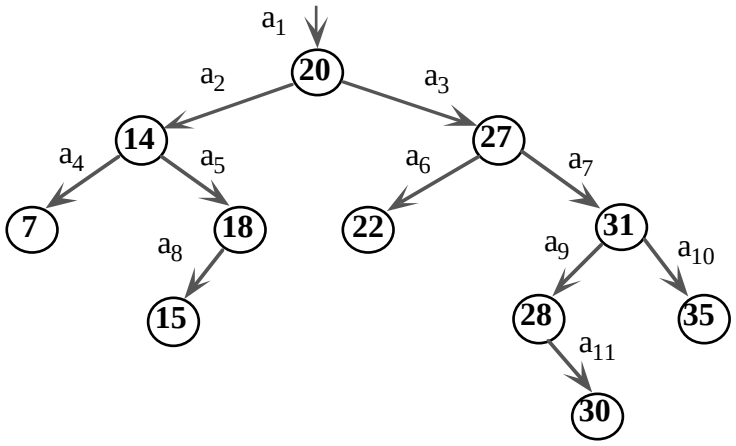
Mostrar el resultado obtenido.

b) Decir qué hace el algoritmo.

10. Dado el siguiente algoritmo, en que TArbol es un árbol binario,

```
FUNCTION misterio(A:TArbol; x:<Tbase>):TArbol;
VAR
  aux:TArbol
BEGIN
  IF ArbolVacio(A) THEN CrearArbol(aux)
  ELSE
    IF (x=DatoRaiz(A)) THEN aux:=A
    ELSE
      IF (x > DatoRaiz(A)) THEN aux:=misterio(HijoDer(A),x)
      ELSE aux:=misterio(HijoIzq(A),x);
  misterio:=aux
END;
```

a) Realizar una traza de la llamada `misterio(a1,28)` con el siguiente árbol:



b) ¿Qué hace el algoritmo?