14. ÁRBOLES

14.1 Fundamentos y terminología básica

Hasta ahora hemos visto estructuras de datos lineales, es decir, los datos estaban estructurados en forma de secuencia. Sin embargo, las relaciones entre los objetos no siempre son tan simples como para ser representadas mediante secuencias (incluso, en ocasiones, es conveniente que no sea así), sino que la complejidad de las relaciones entre los elementos puede requerir otro tipo de estructura. En esas situaciones se pasaría a tener estructuras de datos no lineales. Éste es el caso de la estructura de datos conocida como árbol.

Los árboles establecen una estructura jerárquica entre los objetos. Los árboles genealógicos y los organigramas son ejemplos comunes de árboles.

Un árbol es una colección de elementos llamados nodos, uno de los cuales se distingue como raíz, junto con una relación que impone una estructura jerárquica entre los nodos. Formalmente, un árbol se puede definir de manera recursiva como sigue:

Definición: Una estructura de árbol con tipo base *Valor* es:

(i) Bien la estructura vacía.

(ii) Un conjunto finito de uno o más nodos, tal que existe un nodo especial, llamado nodo raíz, y donde los restantes nodos están separados en $n \geq 0$ conjuntos disjuntos, cada uno de los cuales es a su vez un árbol (llamados subárboles del nodo raíz).

La definición implica que cada nodo del árbol es raíz de algún subárbol contenido en el árbol principal.

Ejemplos de estructuras arborescentes:

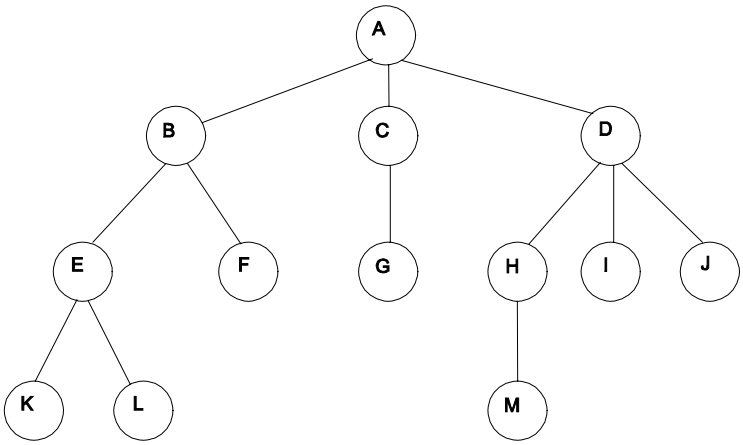


Figura 1: Ejemplo de árbol.

El índice de un libro también se puede representar en forma de árbol, en esta representación se reflejan con claridad las relaciones de dependencia entre cada una de las partes del libro:

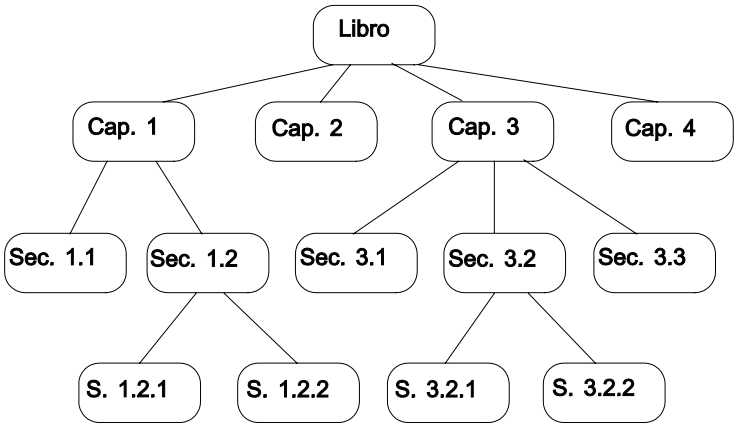


Figura 2: Estructura arborescente representando la estructura de un libro.

Antes de continuar avanzando en las características y propiedades de los árboles, veamos algunos términos importantes asociados con el concepto de árbol:

- **Grado de un nodo:** Es el número de subárboles que tienen como raíz ese nodo (el número de subárboles que "cuelgan" del nodo).
- **Nodo terminal:** Nodo con grado 0, no tiene subárboles.
- **Grado de un árbol:** Grado máximo de los nodos de un árbol.
- **Hijos de un nodo:** Nodos que dependen directamente de ese nodo, es decir, las raíces de sus subárboles.
- **Padre de un nodo:** Antecesor directo de un nodo, nodo del que depende directamente.
- **Nodos hermanos:** Nodos hijos del mismo nodo padre.
- **Camino:** Sucesión de nodos del árbol $n_1, n_2, \ldots, n_k$, tal que n_i es el padre de n_i+1.
- **Antecesores de un nodo:** Todos los nodos en el camino desde la raíz del árbol hasta ese nodo.
- **Nivel de un nodo:** Longitud del camino desde la raíz hasta el nodo. El nodo raíz tiene nivel 1.
- **Altura (profundidad) de un árbol:** Nivel máximo de un nodo en un árbol.

- **Longitud de camino de un árbol:** Suma de las longitudes de los caminos a todos sus componentes.
- **Bosque:** Conjunto de $n>0$ árboles disjuntos.

La representación de un árbol general dependerá de su grado, es decir, del número de relaciones máximo que puede tener un nodo del árbol. Resulta más simple la representación y manipulación de una estructura árbol cuando el grado de éste es fijo y no variable. Por esa razón, para introducir los aspectos más concretos de la manipulación de árboles, vamos a tratar con un tipo particular de árboles de grado fijo, los llamados árboles binarios.

14.2. Árboles binarios

Los árboles binarios constituyen un tipo particular de árboles de gran aplicación. Estos árboles se caracterizan porque no existen nodos con grado mayor que dos, es decir, un nodo tendrá como máximo dos subárboles.

Definición: Un árbol binario es un conjunto finito de nodos que puede estar vacío o consistir en un nodo raíz y dos árboles binarios disjuntos, llamados subárbol izquierdo y subárbol derecho.

En general, en un árbol no se distingue entre los subárboles de un nodo, mientras que en un árbol binario se suele utilizar la nomenclatura subárbol izquierdo y derecho para identificar los dos posibles subárboles de un nodo determinado. De forma que, por ejemplo, los dos siguientes árboles, a pesar de contener la misma información son distintos por la disposición de los subárboles:

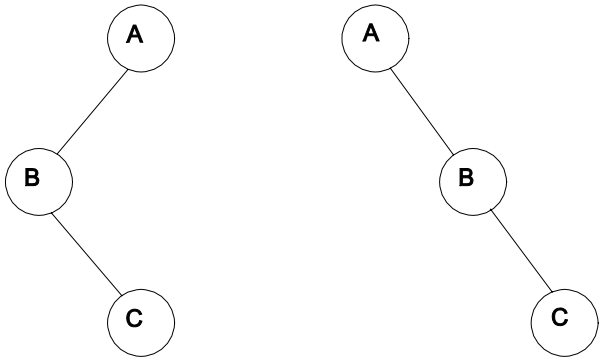


Figura 3: Ejemplos de árboles.

Antes de pasar a la representación de los árboles binarios, vamos a hacer algunas observaciones relevantes acerca del número y características de los nodos en este tipo de árboles.

Lema 1: El número máximo de nodos en el nivel i de un árbol binario es 2^{i-1}, $i\geq 1$, y el número máximo de nodos en un árbol binario de altura k es 2^k-1, $k\geq 1$.

Lema 2: Para cualquier árbol binario no vacío, si n_0 es el número de nodos terminales y n_2 es el número de nodos de grado 2, entonces se cumple que $n_0=n_2+1$. [1]

Igualmente, para poder entender alguna de las formas de representación de los árboles binarios, vamos a introducir dos nuevos conceptos, lo que se entiende por árbol binario lleno y por árbol binario completo.

Definición: Se dice que un árbol binario está lleno si es un árbol binario de profundidad k que tiene 2^k-1 nodos.

Un árbol binario lleno es aquel que contiene el número máximo de posibles nodos. Como en estos casos no existen subárboles vacíos excepto para los nodos terminales, es posible realizar una

[1] Demostraciones de estos lemas se pueden encontrar en Fundamentals of Data Structures in Pascal, Horowitz&Sahni, págs. 260,261

representación secuencial eficiente de este tipo de árboles. Esta representación suele implicar la numeración de los nodos. La numeración se realiza por niveles y de izquierda a derecha. Este proceso de numeración (etiquetado) de los nodos permite una identificación elegante de las relaciones de parentesco entre los nodos del árbol y se verá más adelante (en C/C++ suele empezar la numeración en '`0`' y terminar en '`n-1`'. En otros lenguajes, como Pascal, la numeración suele empezar en '`1`' y terminar en '`n`'. Nosotros nos ceñiremos a la numeración en C/C++.)

Definición: Un árbol binario con `n` nodos y profundidad `k` se dice que es completo si y sólo si sus nodos se corresponden con los nodos numerados de `0` a `n-1` en el árbol binario lleno de profundidad `k`.

Cuando un árbol no es lleno pero es completo, también es posible la representación secuencial eficiente de la estructura de nodos que lo componen.

14.3. Fundamentos

Al igual que hemos visto en el resto de temas, empezaremos por introducir el tipo abstracto de datos 'Arbol Binario'.

Como en las otras estructuras de datos vistas, empezaremos viendo las operaciones que consideraremos básicas para la manipulación de los árboles. Y al igual que hicimos en los otros TAD tendremos diferentes tipos de operaciones:

- *De consulta sobre la información del TAD:* En este caso, se deberá poder consultar tanto la información guardada en el nodo raíz como los hijos del nodo raíz (Informacion, HijoIzdo, HijoDcho.)
- *De consulta sobre la propia estructura:* Si la estructura contiene información, o por el contrario, está vacía (ArbolVacio.)

El hecho de añadir y eliminar una cierta información del árbol, es un problema ciertamente complejo, que depende de diferentes criterios y que es difícil de abordar desde el punto de vista general. El problema de la eliminación de información lo veremos de forma más concreta más adelante.

La adición de información, la veremos desde un punto de vista concreto en este momento, aunque se puede ver de diferentes maneras (veremos otros puntos de vista más adelante, en árboles binarios de búsqueda y en montículos.)

- *Para añadir información:* En este caso, la manera en que consideraremos que se puede añadir información será creando un nuevo árbol a partir de dos árboles ya existentes y con la información que desamos añadir en el nodo raíz, cada subárbol será un hijo del nuevo árbol (HacerArbol.)

Resumiendo, estas operaciones quedarían de la siguiente forma:

IniciarArbol ( ) $\rightarrow$ ArbolBinario
ArbolVacio (ArbolBinario) $\rightarrow$ Boolean
HacerArbol (ArbolBinario, Valor, ArbolBinario) $\rightarrow$ ArbolBinario
HijoIzquierdo (ArbolBinario) $\rightarrow$ ArbolBinario
HijoDerecho (ArbolBinario) $\rightarrow$ ArbolBinario
Informacion (ArbolBinario) $\rightarrow$ Valor

Y cumplen los siguientes axiomas:

$\forall p, r \in ArbolBinario \wedge d \in Valor \Rightarrow$

ArbolVacio (IniciarArbol ( )) → TRUE
ArbolVacio (HacerArbol (p, d, r)) → FALSE
HijoIzquierdo (HacerArbol (p, d, r)) → p
HijoIzquierdo (IniciarArbol ( )) → ERROR
HijoDerecho (HacerArbol (p, d, r)) → r
HijoDerecho (IniciarArbol ( )) → ERROR
Informacion (HacerArbol (p, d, r)) → d
Informacion (IniciarArbol ( )) → ERROR

A partir de esta información, ya somos capaces de codificar en C++ la interfaz de la clase árbol:

```
class Arbol
{
    public:
        typedef .??. Valor;

        Arbol (void);                /* IniciarArbol -> Constructor */
        Arbol (const Arbol &);       /* Constructor de copia        */
        ~Arbol (void);               /* Destructor                  */

        void HacerArbol (Arbol &, Valor, Arbol &);

        bool ArbolVacio (void);
        bool Informacion (Valor &);
        Arbol &HijoIzdo (void);
        Arbol &HijoDcho (void);

   private:
      ...
};
```

Vista la parte pública de la clase, ya podemos hacer uso de la misma.

14.4. Operaciones con árboles binarios: Recorrido de árboles binarios

Si se desea manipular la información contenida en un árbol, lo primero que hay que saber es cómo se puede recorrer ese árbol, de manera que se acceda a todos los nodos del árbol solamente una vez. El recorrido completo de un árbol produce un orden lineal en la información del árbol. Este orden puede ser útil en determinadas ocasiones.

Cuando se recorre un árbol se desea tratar cada nodo y cada subárbol de la misma manera. Existen entonces seis posibles formas de recorrer un árbol binario:

(1) nodo - subárbol izquierdo - subárbol derecho
(2) subárbol izquierdo - nodo - subárbol derecho
(3) subárbol izquierdo - subárbol derecho - nodo
(4) nodo - subárbol derecho - subárbol izquierdo
(5) subárbol derecho - nodo - subárbol izquierdo
(6) subárbol derecho - subárbol izquierdo - nodo

Si se adopta el convenio de que, por razones de simetría, siempre se recorrerá antes el subárbol izquierdo que el derecho, entonces tenemos solamente tres tipos de recorrido de un árbol (los tres

primeros en la lista anterior). Estos recorridos, atendiendo a la posición en que se procesa la información del nodo, reciben, respectivamente, el nombre de recorrido prefijo, infijo y posfijo.

Los tres tipos de recorrido tienen una definición muy simple si se hace uso de una expresión recursiva de los mismos.

```
Algoritmo Prefijo
Entrada
        Arbol arb
Inicio
        si ( arb no es Vacio ) entonces
                Procesar ( Informacion de arb )
                Prefijo ( Hijo izquierdo de arb )
                Prefijo ( Hijo derecho de arb )
        fin_si
fin
```

■

```
Algoritmo Infijo
Entrada
        Arbol arb
Inicio
        Si ( arb no es Vacio ) entonces
                Infijo ( Hijo izquierdo de arb )
                Procesar ( Informacion de arb )
                Infijo ( Hijo derecho de arb )
        Fin_si
Fin
```

■

```
Algoritmo Posfijo
Entrada
        Arbol arb
Inicio
        Si ( arb no es Vacio ) entonces
                Posfijo ( Hijo izquierdo de arb )
                Posfijo ( Hijo derecho de arb )
                Procesar ( Informacion de arb )
        Fin_si
Fin
```

■

Ejemplo de recorrido de un árbol binario:

Recorrido Prefijo: A B C D E F G

Recorrido Infijo: C B D E A G F

Recorrido Posfijo: C E D B G F A

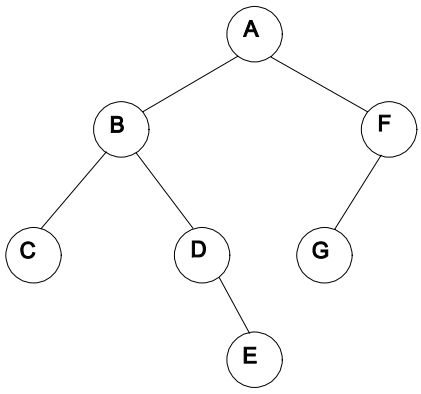


Figura 5

■

En C++, la implementación de los recorridos es muy simple. Por ejemplo el recorrido Prefijo quedaría como sigue:

```
Prefijo (const Arbol & arb)
{
   Valor inf;

   if (arb.ArbolVacio () == false)
   {
      arb.Informacion (inf);
      Procesar (inf);

      Prefijo (arb.HijoIzdo() );
      Prefijo (arb.HijoDcho() );
   }
}
```

El paso del parámetro lo hacemos por referencia, pero constante, para no tener que realizar la copia de la información y hacer un método más eficiente.

14.5. Otras operaciones con árboles binarios

La primera operación compleja a considerar sobre árboles binarios será su generación. En este caso, no vamos a entrar a considerar todos los casos que pueden aparecer al insertar un nuevo nodo en un árbol, la problemática puede ser amplia y el proceso de generación de un árbol dependerá de las reglas impuestas por una aplicación particular. Vamos a ver, sin embargo, algún ejemplo concreto que permita ilustrar esta operación.

Al manipular la estructura árbol, la situación más habitual es que el problema imponga una serie de reglas para la construcción del árbol en función de los datos de entrada. Estos datos, en principio, serán desconocidos antes de la ejecución del programa. El procedimiento de generación del árbol deberá, por tanto, reproducir de la manera más eficiente posible esas reglas de generación.

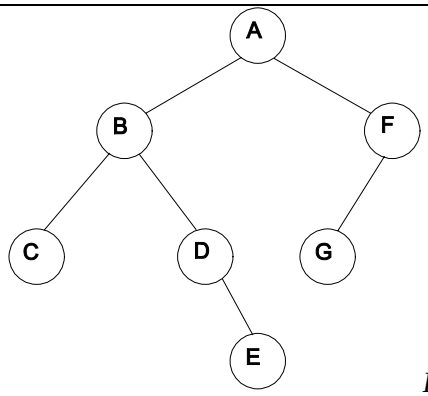
Ejemplo 1: Generación de un árbol a partir de su recorrido prefijo.

Supongamos que se desea generar en memoria una estructura de árbol binario con unos datos cuyas relaciones se conocen previamente. Es decir, el usuario va a trasladar al ordenador una estructura de árbol conocida. Para hacer esto, se establece una notación secuencial que permita la interpretación simple de las relaciones. La notación consiste en una secuencia de caracteres que indica la información de cada nodo en el árbol completo asociado, de manera que se indica también si algún subárbol está vacío y se supone que la información del subárbol izquierdo va siempre antes que la del subárbol derecho. En realidad se trata de una representación secuencial del recorrido prefijo del árbol. Se considerará, para simplificar, que la información asociada con cada nodo es simplemente un carácter y que los subárboles vacíos se representan con un '.'. Entonces la entrada al algoritmo de generación puede ser simplemente una cadena de caracteres, donde cada uno de ellos se interpreta como un nodo del árbol.

En este caso, las reglas de generación del árbol binario a partir de su representación secuencial serían:

(1) Leer carácter a carácter la secuencia de entrada.
   (a) Si el carácter que se lee es '.' no hay que crear ningún nodo, el subárbol está vacío.

(b) Si se lee un carácter distinto de '**.**' entonces se crea un nuevo nodo con la información leída y se pasa a generar los subárboles izquierdo y derecho de ese nodo, según los mismos criterios y en ese mismo orden.



Representación secuencial del árbol:
A B C . . D . E . . F G . . .

Figura 6

■

Un algoritmo que implemente estas reglas de generación podría ser:

```
Algoritmo Generar_Arbol
Salida
        ArbolBinario arb
Inicio
        Leer ( carácter )
        Si carácter ≠ '.' entonces
                Informacion de arb ← caracter
                Generar_Arbol ( Hijo izquierdo de arb )
                Generar_Arbol ( Hijo derecho de arb )
        Sino
                arb es vacío
        Fin_si
Fin
```

■

Utilizando exclusivamente las operaciones básicas definidas previamente, el algoritmo de generación quedaría como sigue:

```
Algoritmo Generar_Arbol
Salida
        ArbolBinario arb
Auxiliar
        ArbolBinario izq, der
Inicio
        Leer ( carácter )
        Si(1) carácter ≠ '.' entonces
                izq ← Generar_Arbol
                der ← Generar_Arbol
                arb ← HacerArbol (izq, caracter, der)
        Sino(1)
                arb ← IniciarArbol ( )
        Fin_si(1)
Fin
```

■

Se puede observar como la recursividad permite definir de una manera simple y elegante el proceso de generación.

Esta función escrita en C++ quedaría como sigue:

```
void GenerarArbol (Arbol & arb)
{
   Arbol izq, der;

   cin >> x

   if (x != '.')
   {
      GenerarArbol (izq);
      GenerarArbol (der);
      arb.HacerArbol (izq, x, der);
   }
}
```

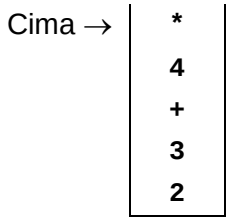
Resaltar, que en ningún momento se llama a ninguna función para iniciar el árbol, ya que por el hecho de declararlo el árbol es creado vacío (se llama al constructor por defecto de la clase.)

Ejemplo 2: Árbol binario para la representación de una expresión algebraica.

Generar el árbol binario de representación de una expresión algebraica. Para ello se realizan las siguientes consideraciones:

(1) La entrada del programa deberá ser una expresión escrita en notación postfija guardada en una pila.

Por ejemplo, la expresión: 2 3 + 4 * (que equivaldría a *(2+3)*4*) estaría guardada en una pila de la siguiente manera:



(2) En las expresiones sólo se tendrán en cuenta los cuatro operadores binarios básicos: **+, -, *, /**.

(3) Los operandos serán exclusivamente caracteres (1 carácter, 1 operando).

A partir de estas consideraciones, la generación del árbol de la expresión quedaría:

Algoritmo Crear_Arbol

Entradas
pila_expr: Pila
arb: Arbol
Salidas
arb: Arbol
Inicio
Si$_{(1)}$ (no *Vacia_Pila* (pila_expr)) **entonces**
arb ← crear_espacio
arb^.info (*Desapilar* (pila_expr)

```
                Si(2) ( arb^.info es operador ) entonces
                {* como se lee la expresión al revés, el primer operando que se lee  *}
                {* es en realidad el último, por eso se genera antes el árbol derecho *}
                        Crear_Arbol ( arb^.der )
                        Crear_Arbol ( arb^.izq )
                Sino(2)
                        arb^.der ← NULO
                        arb^.izq ← NULO
                Fin_si(2)
        Fin_si(1)
Fin
```

■

Al igual que en el ejemplo anterior, si nos ceñimos a las operaciones básicas el algoritmo quedaría de la siguiente manera:

```
Algoritmo Crear_Arbol

Entradas
        pila_expr: Pila
Salidas
        arb: ArbolBinario
Variables
        der, izq: ArbolBinario
Inicio
        Si(1) ( no Vacia_Pila ( pila_expr ) ) entonces
                y ← Desapilar ( pila_expr )
                Si(2) ( y es operador ) entonces
                {* como se lee la expresión al revés, el primer operando que se lee  *}
                {* es en realidad el último, por eso se genera antes el árbol derecho *}
                        Crear_Arbol ( der )
                        Crear_Arbol ( izq )
                Sino(2)
                        der ← IniciarArbol ( )
                        izq ← IniciarArbol ( )
                Fin_si(2)
                arb ← HacerArbol ( izq, y, der )
        Fin_si(1)
Fin
```

■

Ejemplo 3: Copia de un árbol binario en otro.

Como un ejemplo más de manipulación de un árbol, veamos un algoritmo que nos permita realizar un duplicado de un árbol ya existente (sin utilizar el constructor de copia). Al copiar el árbol se debe copiar toda la información contenida en el mismo, con la misma estructura de relaciones que existe en el árbol original.

El proceso de copia implica el recorrido sistemático del árbol original, al mismo tiempo que se genera un nuevo árbol con la misma información.

```
Algoritmo Copiar

Entrada
        org: ArbolBinario
```

```
Salida
        dest: ArbolBinario
Variable
        aux: ArbolBinario
Inicio
        Si ( org = NULO ) entonces
                dest ← NULO
        Sino
                aux ← Crear_Espacio
                aux^.info ← org^.info
                aux^.izq ← Copiar(org^.izq)
                aux^.der ← Copiar(org^.der)
                dest ← aux
        Fin_si
Fin
```

■

Este algoritmo quedaría, utilizando las operaciones básicas:

```
Algoritmo Copiar

Entrada
        org: ArbolBinario
Salida
        dest: ArbolBinario
Inicio
        Si ( VacioArbol ( org ) ) entonces
                dest ← IniciarArbol ( )
        Sino
                dest ← HacerArbol (  Copiar( HijoIzquierdo ( org ) ),
                                     Informacion ( org ),
                                     Copiar( HijoIzquierdo ( org ) ) )
        Fin_si
Fin
```

■

Esta función escrita en C++ quedaría como sigue:

```
void CopiarArbol (Arbol & des, Arbol ori)
{
   Arbol izq, der;
   Valor x;
   bool b_aux;

   if (!ori.ArbolVacio () )
   {
      CopiarArbol (izq, ori.HijoIzdo () );
      CopiarArbol (der, ori.HijoDcho () );
      b_aux = ori.Informacion (x);
      des.HacerArbol (izq, x, der);
   }
}
```

Ejemplo 4: Equivalencia entre árboles binarios.

Otro problema que se puede solucionar fácilmente utilizando la recursión es el de determinar la equivalencia entre dos árboles binarios. Se dice que dos árboles binarios son equivalentes

si presentan la misma topología (la misma estructura de relaciones entre nodos) y la información en los nodos correspondientes es la misma.

```
Algoritmo Equivalente

Entradas
        ArbolBinario arbol1, arbol2
Salida
        Boolean (CIERTO, FALSO)
Variable
        Boolean equiv: (CIERTO, FALSO)
Inicio
        equiv ← FALSO
        Si(1) ( (arb1 está vacío) y (arb2 está vacío) ) entonces
                equiv ← CIERTO
        Sino(1)
            Si(2) ( (arb1 no está vacío) y (arb2 no está vacio) ) entonces
                Si(3) (Información de arb1 = Información de arb2) entonces
                    Si(4) (Equivalente (Hijo izquierdo arb1, Hijo izquierdo arb2) ) entonces
                        equiv ← Equivalente (Hijo derecho arb1, Hijo derecho arb2)
                    Fin_si(4)
                Fin_si(3)
            Fin_si(2)
        Fin_si(1)

        Devolver equiv
Fin
```

■

Restringiéndonos nuevamente a las operaciones básicas, el algoritmo de comprobación sería el siguiente:

```
Algoritmo Equivalente

Entradas
        ArbolBinario arb1, arb2
Salida
        Boolean (CIERTO, FALSO)
Variable
        Boolean equiv : (CIERTO, FALSO)
Inicio
    equiv ← FALSO
    Si(1) ( ArbolVacio ( arb1 ) y ArbolVacio ( arb2 ) ) entonces
        equiv ← CIERTO
    Sino(1)
        Si(2) ( ( no ArbolVacio ( arb1 ) y no ArbolVacio ( arb2 ) ) entonces
            Si(3) ( Informacion ( arb1 ) = Informacion ( arb2 ) ) entonces
                Si(4) (Equivalente (HijoIzquierdo (arb1), HijoIzquierdo (arb2) ) ) entonces
                    equiv ← Equivalente (HijoDerecho (arb1), HijoDerecho (arb2) )
                Fin_si(4)
            Fin_si(3)
        Fin_si(2)
    Fin_si(1)
    Devolver equiv
Fin
```

■

La función, escrita en C++, siguiendo la interfaz de la clase propuesta quedaría como sigue:

```
bool Equivalente (Arbol arb1, Arbol arb2)
{
   bool b_aux, equiv = false;

   if (arb1.ArbolVacio () && arb2.ArbolVacio () )
      equiv = true;
   else
   {
      if (!arb1.ArbolVacio () && !arb2.ArbolVacio () )
      {
         b_aux = arb1.Informacion (x);
         b_aux = arb2.Informacion (y);
         if (x == y)
            if (Equivalente (arb1.HijoIzdo (), arb2.HijoIzdo() )
               equiv = Equivalente (arb1.HijoDcho (), arb2.HijoDcho() );
      }
   }
   return equiv;
}
```

14.6. Representación de los árboles binarios

Representación mediante *arrays*

Como hemos visto, si el árbol binario que se desea representar cumple las condiciones de árbol lleno o árbol completo, es posible encontrar una buena representación secuencial del mismo. En esos casos, los nodos pueden ser almacenados en un array unidimensional, **A**, de manera que el nodo numerado como **i** se almacena en **A[i]**. Esto permite localizar fácilmente las posiciones de los nodos padre, hijo izquierdo e hijo derecho de cualquier nodo **i** en un árbol binario arbitrario que cumpla tales condiciones.

Lema 3: Si un árbol binario completo con **n** nodos se representa secuencialmente, se cumple que para cualquier nodo con índice **i**, **0≤i≤(n-1)**, se tiene que:

(1) El padre del nodo **i** estará localizado en la posición **(i-1)/2** si **i>0**. Si **i=0**, se trata del nodo raíz y no tiene padre.

(2) El hijo izquierdo del nodo **i** estará localizado en la posición **(2*(i+1))-1** si **2*(i+1)-1<n**. Si **2*(i+1)-1≥n**, el nodo no tiene hijo izquierdo.

(3) El hijo derecho del nodo **i** estará localizado en la posición **(2*(i+1))** si **2*(i+1)<n**. Si **2*(i+1)≥n**, el nodo no tiene hijo derecho.

Evidentemente, la representación puramente secuencial de un árbol se podría extender inmediatamente a cualquier árbol binario, pero esto implicaría, en la mayoría de los casos, desaprovechar gran cantidad del espacio reservado en memoria para el *array*. Así, aunque para árboles binarios completos la representación es ideal y no se desaprovecha espacio, en el peor de los casos, para un árbol lineal (una lista) de profundidad **k**, se necesitaría espacio para representar 2^k-1 nodos, y sólo **k** de esas posiciones estarían realmente ocupadas.

Además de los problemas de eficiencia desde el punto de vista del almacenamiento en memoria, hay que tener en cuenta los problemas generales de manipulación de estructuras secuenciales. Las operaciones de inserción y borrado de elementos en cualquier posición del array implican necesariamente el movimiento potencial de muchos nodos. Estos problemas se puede solventar adecuadamente mediante la utilización de una representación enlazada de los nodos.

Representación mediante cursores

Se podría simular una estructura de enlaces, como la utilizada para las listas, mediante un array. En ese caso, los campos de enlace de unos nodos con otros no serían más que índices dentro del rango válido definido para el array. La estructura de esta representación enlazada pero ubicada secuencialmente en la memoria correspondería al siguiente esquema para cada nodo:

Información	
Hijo Izquierdo	Hijo Derecho

donde el campo información guarda toda la información asociada con el nodo y los campos hijo izquierdo e hijo derecho, guardan la posición dentro del array donde se almacenan los respectivos hijos del nodo.

Ejemplo:

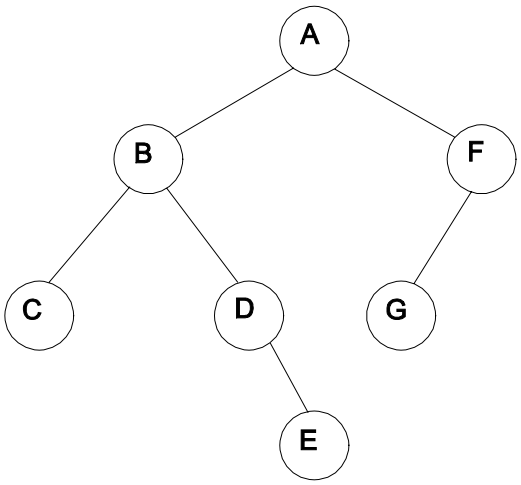


Figura 4

	Información	*Hijo Izquierdo*	*Hijo Derecho*
1	A	2	6
2	B	3	4
3	C	0	0
4	D	0	5
5	E	0	0
6	F	7	0
7	G	0	0
...			

■

Los problemas de esta representación están asociados con la manipulación de la misma. En el caso de la inserción de nuevos nodos, el único problema que puede aparecer es que se supere el espacio reservado para el array, en cuyo caso no se podrían insertar más nodos. Sin embargo, la operación de borrado implica mayores problemas de manipulación. Al poder borrar cualquier nodo del árbol, se van a dejar "huecos" en la estructura secuencial del array, esto se podría solucionar, como siempre, mediante el desplazamiento de elementos dentro del vector. Sin embargo, si de por sí esa operación no resulta recomendable, en este caso mucho menos, ya que implica a su vez modificar los enlaces que hayan podido variar con el desplazamiento. Otra solución, podría consistir en almacenar las posiciones del array que están libres y que se pueden ocupar en la inserción de nodos. Esto podría resultar más eficiente, pues no implica el desplazamiento de información, pero requiere la utilización de otra estructura de datos auxiliar que maneje esa información.

Representación mediante punteros

La mejor solución, de nuevo, para evitar los problemas asociados con la manipulación de *arrays*, es la representación de los árboles binarios mediante estructuras dinámicas "puras", en el sentido de la creación en tiempo de ejecución de los nodos que sean necesarios (no más) y la utilización de punteros para enlazar estos nodos.

La estructura de cada nodo en esta representación coincide con la estructura de tres campos vista anteriormente. La única diferencia reside en la naturaleza de los campos de enlace, en este caso se trata de punteros y no de índices de un array.

```
Tipos:
   Arbol = Puntero a NodoArbol;

   NodoArbol = registro
                 Info    : Valor;
                 Izq, Der: Arbol;
               fin;
```

■

Se puede comprobar como la representación en memoria de la estructura, en cuanto a definición de tipos de datos, coincide exactamente con la de, por ejemplo, una lista doblemente ligada, sin que ello implique que se esté hablando de la misma estructura de datos. De nuevo hay que dejar muy claro que es la interpretación que hacemos de la representación en memoria la que define una estructura de datos (definición de operaciones), no la representación en sí misma.

La representación enlazada tiene como principal desventaja que no resulta inmediato la determinación del padre de un nodo. Es necesario buscar el camino de acceso al nodo dentro del árbol para poder obtener esa información. Sin embargo, en la mayoría de las aplicaciones es la representación más adecuada. En el caso en que la determinación del padre de un nodo sea una operación importante y frecuente en una aplicación concreta, se puede modificar ligeramente la estructura de cada nodo añadiendo un cuarto campo, también de tipo **`Puntero a NodoArbol`**, que guarde la posición del padre de cada nodo.

Una posible representación, con clases de C++, basándonos en estas ideas podría ser la siguiente:

```
class Arbol
{
    public:
        ...
    private:
        typedef Arbol * PunteroArbol;

        bool esvacio;

        Valor info;
        PunteroArbol izdo;
        PunteroArbol dcho;
};
```

Con esta representación, la codificación de las operaciones de la interfaz de la clase en C++ sería la siguiente:

Constructores de la clase

El constructor por defecto se limitará a poner el árbol como vacío:

```
Arbol::Arbol (void)
{
   esvacio = true;
}
```

El constructor de copia se puede hacer básicamente utilizando la misma idea que la función de copia entre árboles, pero implementando la función de copia de árboles como un método privado de la clase, y utilizándola en el constructor de copia. La función CopiarArbol, la podremos reutilizar en otras operaciones de copia de árboles.

```
void Arbol::Copiar (const Arbol & ori)
{
   esvacio = ori->esvacio;

   if (!ori->esvacio)
   {
      info = ori->info;

      izdo = new Arbol;
      izdo->CopiarArbol (ori->izdo);

      dcho = new Arbol;
      dcho->CopiarArbol (ori->dcho);
   }
}
```

Utilizando este método el constructor de copia quedaría como sigue:

```
void Arbol:: Arbol (const Arbol & ori)
{
    this->Copiar (ori);
}
```

Otra manera de implementar el constructor de copia sería utilizando la idea de que la llamada al **new** realiza la llamada implícita al constructor de la clase, si el **new** lo utilizamos con clases.

Cuando invocamos al **new**, se reserva espacio para el nuevo elemento del tipo del operando con que se invoca al **new**. Si éste es una clase, se llama de forma automática al constructor adecuado, en principio si hacemos **new Arbol** se llamará al constructor por defecto (el que no tiene parámetros). Sin embargo si hacemos llamada con la clase con parámetros, se buscará entre los constructores aquél que tenga el parámetro adecuado. Si hacemos la llamada con **new Arbol (arb)**, el constructor que acepta como parámetro un árbol es el constructor de copia, de manera que se reservará espacio para el nuevo elemento y se llamará al constructor de copia con el parámetro '**arb**'. Con esto ya conseguimos hacer la reserva recursiva de todos los nodos del árbol

```
Arbol::Arbol (const Arbol & arb)
{
    if (arb.esvacio == true)
        esvacio = true;
    else
    {
        esvacio = false;
        info = arb.info;
        izdo = new Arbol(*arb.izdo);
        dcho = new Arbol(*arb.dcho);
    }
}
```

Método `HacerArbol`

El método **`HacerArbol`**, al igual que el constructor de copia, puede realizarse utilizando el método de copia de árboles, o basarse en la utilización del **`new`**, que llamará al constructor de copia y que básicamente copia árboles:

```
void Arbol::HacerArbol (Arbol & izq, Valor inf, Arbol & der)
{
    esvacio = false;

    info = inf;

    izdo = new Arbol;
    izdo->Copiar (izq);

    dcho = new Arbol;
    dcho->Copiar (der);
}
```

O en la otra versión:

```
void Arbol::HacerArbol (Arbol & izq, Valor inf, Arbol & der)
{
   esvacio = false;

   info = inf;

   izdo = new Arbol (izq);
   dcho = new Arbol (der);
}
```

Métodos de consulta

Los métodos de consulta, tanto sobre el propio árbol, como sobre el contenido del árbol, serán relativamente sencillos:

```
bool Arbol::ArbolVacio (void)
{
   return esvacio;
}
```

```
bool Arbol::Informacion (Valor & inf)
{
   bool exito;

   if (esvacio == true)
      exito = false;
   else
   {
      exito = true;

      inf = info;
   }
   return exito;
}
```

```
Arbol & Arbol::HijoIzdo (void)
{
   return *izdo;
}
```

```
Arbol & Arbol::HijoDcho (void)
{
   return *dcho;
}
```

Destructor de la clase

Para el destructor, tendremos que implementar una función recurrente que borre todos los nodos del árbol. Además podemos utilizar que el **delete** tiene un comportamiento similar en cierta manera al **new**. Cuando llamamos a **delete** si el operando es un objeto de una clase, se llama de forma automática al destructor de la clase, y una vez finalizada la ejecución del método, se libera la memoria asociada al objeto.

Con esto el destructor quedarían como sigue:

```
Arbol::~Arbol (void)
{
   if (!esvacio)
   {
      delete (izdo);
      delete (dcho);
   }
}
```

Eliminación de nodos en un árbol binario

La eliminación de nodos es otra operación habitual de manipulación de árboles binarios. Hay que tener en cuenta que esta operación debe mantener la estructura del árbol y, por tanto, dependerá del orden que se haya establecido dentro del árbol (si lo hay). Esto hace que la eliminación de nodos se pueda convertir en una operación más compleja y que no se pueda hablar de un algoritmo para borrar en general, más bien se podrá hablar de algoritmos para borrar nodos para un determinado orden dentro del árbol.

Si por ejemplo, un árbol ha sido generado teniendo en cuenta que sus nodos está correctamente ordenados si se recorre el árbol en orden infijo, se debe considerar un algoritmo de borrado que no altere este orden. La secuencia de orden a que da lugar el recorrido infijo se debe mantener al borrar cualquier nodo del árbol.

Existen dos casos a considerar cuando se borra un nodo del árbol. El primer caso, el más simple, implica borrar un nodo que tenga al menos un subárbol vacío. El segundo caso, más complejo, aparece cuando se desea borrar un nodo cuyos dos subárboles son no vacíos. En este último caso, hay que tener que el nodo borrado debe ser sustituido por otro nodo, de manera que se mantenga la estructura inicial del árbol. Cuando el nodo a borrar posee dos subárboles no vacíos, el proceso de reestructuración de las relaciones dentro del árbol resulta más complejo, ya que cuando el nodo es terminal, eliminar ese nodo es tan simple como eliminar el enlace que le mantiene unido al árbol desde su nodo padre, y si el nodo posee un único subárbol no vacío, el nodo debe ser sustituido por su nodo hijo. Estas dos últimas situaciones no implican una reestructuración importante del árbol.

Suponiendo que el árbol debe mantener el orden establecido entre los nodos por un recorrido particular del mismo, el problema es determinar qué nodo debe sustituir al nodo que se va a borrar. Si volvemos al ejemplo anterior, el orden establecido por el recorrido infijo del árbol sería:

Recorrido infijo: C B D E A G F

Si se desea borrar el nodo B, que tiene dos nodos hijos, el nodo que debería sustituirle sería el nodo D, ya que es su sucesor en el orden establecido por el recorrido infijo (criterio escogido en este ejemplo particular). En general, sea cuál sea el orden establecido entre los nodos, el sustituto de un nodo que se va a borrar deberá ser su sucesor en dicho orden. De esta manera se mantiene la estructura global del árbol.

Los diferentes criterios de ordenación de los nodos van a dar lugar a distintos algoritmos para determinar el sucesor de un nodo, según el orden establecido, y para modificar los enlaces entre nodos.

Ejemplo

Veamos el ejemplo concreto de un algoritmo que permita borrar un nodo en el caso en que se supone como válido el orden establecido entre los nodos por un recorrido infijo del árbol:

```
Algoritmo BorrarNodo

Entrada
        ArbolBinario arb        { Raiz del árbol de donde se pretende borrar }
        ArbolBinario donde      { Árbol a borrar }
Variables
        ArbolBinario pad, suc

Inicio
        Si(1) (donde no es vacío) entonces
                { Subárbol izquierdo vacío (o nodo terminal) }
                Si(2) (Hijo izquierdo de donde es vacío) entonces
                        Si(3) (donde = arb) entonces
                                arb ← Hijo derecho de donde
                        Sino(3)
                                pad ← Padre (donde, arb)
                                Si(4) ( Hijo izquierdo de pad = donde ) entonces
                                        Hijo izquierdo de pad  ← Hijo derecho de donde
                                Sino(4)
                                        Hijo derecho de pad ← Hijo derecho de donde
                                Fin_si(4)
                        Fin_si(3)
                        Liberar_Espacio (donde)
                Sino(2)
                        {Subárbol derecho vacío}
                        Si(5) (Hijo derecho de donde es vacío) entonces
                                Si(6) ( donde = arb ) entonces
                                        arb ← Hijo izquierdo de donde
                                Sino(6)
                                        pad ← padre (donde, arb)
                                        Si(7) (Hijo izquierdo de pad = donde ) entonces
                                                Hijo izquierdo de pad ← Hijo izquierdo de donde
                                        Sino(7)
                                                Hijo derecho de pad ← hijo izquierdo de donde
                                        Fin_si(7)
                                Fin_si(6)
                                Liberar_Espacio ( donde )
```

```
                    Sino(5)
                        { Subárboles NO vacíos }
                        { El nodo raiz no es un caso especial }
                        suc ← Sucesor_Infijo (donde)
                        Informacion de donde ← Informacion de suc
                        { Se puede utilizar la recursión :}
                        {       BorrarNodo ( arb, suc )       }
                        { O borrarlo directamente:            }
                        pad ← padre (suc, arb)
                        Si(8) (donde = pad) entonces
                            Hijo derecho de pad ← Hijo derecho de suc
                        Sino(8)
                            Hijo izquierdo de pad ← Hijo derecho de suc
                        Fin_si(8)
                        Liberar_Espacio (suc)
                    Fin_si(5)
                Fin_si(2)
            Fin_si(1)
        Fin
```

■

Hace falta especificar los algoritmos que permitirán encontrar el padre (*Padre)* y el sucesor infijo de un nodo del árbol (*Sucesor_Infijo*).

```
Algoritmo Padre

Entradas
    ArbolBinario donde, arb     { Se pretende buscar el subárbol 'donde' en el subárbol 'arb'}
Salida
    ArbolBinario
Variable
    ArbolBinario aux

Inicio
    Si(1) (arb no es vacío) entonces
        Si(2) (Hijo izquierdo de arb = donde ) o (Hijo derecho de arb = donde) entonces
            Devolver (arb)
        Sino(2)
            aux ← Padre (donde, Hijo izquierdo de arb)
            Si(3) (aux es vacío) entonces
                aux ← Padre (donde, Hijo derecho de arb)
            Fin_si(3)
            Devolver ( aux )
        Fin_si(2)
    Sino(1)
        Devolver (Arbol vacío)
    Fin_si(1)
Fin
```

■

El algoritmo que busca el padre de un nodo es en realidad un algoritmo general de búsqueda, que permite recorrer todo el árbol hasta encontrar un nodo que cumpla la condición establecida, independientemente del orden que pudiera existir en el árbol. Por lo tanto, puede ser fácilmente modificado para poder localizar un nodo a partir de la información que

contiene. En este caso, el algoritmo general de búsqueda de un nodo por su información sería:

```
Algoritmo Buscar

Entradas
        Valor x
        ArbolBinario arb
Salida
        ArbolBinario
Variable
        aux: ArbolBinario

Inicio
        Si(1) (arb no es vacío) entonces
                Si(2) (Información de arb = x ) entonces
                        Devolver (arb)
                Sino(2)
                        aux ← Buscar (x, Hijo izquierdo de arb)
                        Si(3) (aux es vacío) entonces
                                aux ← Buscar (x, Hijo derecho de arb)
                        Fin_si(3)
                        Devolver (aux)
                Fin_si(2)
        Sino(1)
                Devolver (Arbol vacío)
        Fin_si(1)
Fin
```

■

Volviendo al algoritmo de borrado, hemos visto que es necesario localizar el sucesor infijo de un nodo. En este caso el algoritmo es muy simple. Se trataría de recorrer el subárbol derecho del nodo, siempre a través de los enlaces izquierdos, hasta encontrar un nodo que tenga el subárbol izquierdo vacío. Este algoritmo se puede representar fácilmente tanto utilizando un esquema recursivo como iterativo. A continuación se muestran los dos algoritmos.

```
Algoritmo Sucesor_Infijo_Recursivo

Entrada
        ArbolBinario arb
{  En este caso el primer valor de entrada del algoritmo debe ser el subárbol derecho    }
{  del nodo del cual se desea encontrar el sucesor. La llamada deberia hacerse como:    }
{       Sucesor_Infijo_Recursivo (Hijo derecho de arb )                                 }
Salida
        ArbolBinario

Inicio
        Si (Hijo izquierdo de arb no es un árbol vacío) entonces
                Devolver (Sucesor (hijo izquierdo de arb) )
        Sino
                Devolver (arb)
        Fin_si
Fin
```

■

```
Algoritmo Sucesor_Infijo_Iterativo

Entrada
        ArbolBinario arb
Salida
        PunteroArbol

Inicio
        arb ← Hijo derecho de arb
        Mientras (Hijo izquierdo de arb no sea vacío) hacer
                arb ← arb^.izq
        Fin_mientras
        Devolver ( arb )
Fin
```

■

Métodos adicionales

El hecho de tener una representación enlazada para implementar internamente los árboles, hace que ciertas operaciones con ellos sean, cuanto menos 'delicadas'.

Así una operación tan simple como la asignación o la reutilización de variables en nuestro programa pueden producir efectos no deseados como la pérdida de información o la inutilización de areas de memoria que quedan reservadas pero inaccesibles.

Por ello, es conveniente, en la representación enlazada, definir dos métodos adicionales que eviten estos efectos.

Los métodos que vamos a añadir a nuestra clase van a ser 'Borrar' y 'Asignar'.

Método `Borrar`

Este método tiene como tarea fundamental liberar el espacio ocupado por los hijos del árbol, y dejar el objeto que ha realizado listo para poder ser reutilizado, o simplemente vacío.

La idea es muy parecida a la del destructor, de hecho se puede realizar llamando al destructor para cada uno de los hijos, si estos existen y dejando el valor de **`esvacio`** a **`true`**.

```
void Arbol::Borrar (void)
{
   if (!esvacio)
   {
        delete (izdo);
        delete (dcho);

        esvacio = true;
   }
}
```

Método `Asignar`

Este método tiene que realizar básicamente dos tareas. Una primera tarea de vaciar el árbol en el que vamos a dejar la información y una segunda tarea de copia de un árbol en otro. Para la primera tarea tenemos un metodo que vacía árboles (**`Borrar`**) y para la segunda tenemos un método privado que copia árboles (**`Copiar`**).

Utilizando estos métodos nos quedaría:

```
void Arbol::Asignar (const Arbol & arb)
{
    Borrar ();
    Copiar (arb);
}
```

14.7. Árboles binarios enlazados ("hilvanados")

Al estudiar la representación enlazada de un árbol binario es fácil observar que existen muchos enlaces nulos. De hecho, existen más enlaces nulos que punteros con valores reales. En concreto, para un árbol con `n` nodos, existen `n+1` enlaces nulos de los `2*n` enlaces existentes en la representación (más de la mitad). Como el espacio de memoria ocupado por los enlaces nulos es el mismo que el ocupado por los no nulos, podría resultar conveniente utilizar estos enlaces nulos para almacenar alguna información de interés para la manipulación del árbol binario. Una forma de utilizar estos enlaces es sustituirlos por punteros a otros nodos del árbol (este tipo especial de punteros se conocen en la literatura anglosajona con el nombre de threads, que se podría traducir por hilos). En particular, los enlaces nulos situados en el subárbol derecho de un nodo se suelen reutilizar para apuntar al sucesor de ese nodo en un determinado recorrido del árbol, por ejemplo infijo, mientras que los enlaces nulos en subárboles izquierdos se utilizan para apuntar al predecesor del nodo en el mismo tipo de recorrido. Si para algún nodo no existe predecesor (porque es el primero en el recorrido establecido) o sucesor (porque es el último), se mantiene con valor nulo el enlace correspondiente.

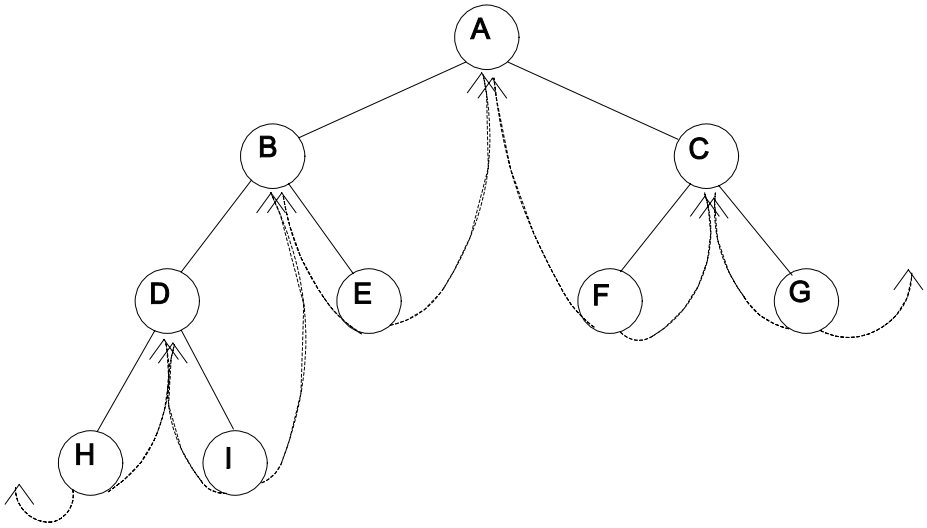


Figura 7. Árbol binario hilvanado

La ventaja de este tipo de representación no es sólo el mejor aprovechamiento de la memoria disponible, sino por la posibilidad de un acceso rápido al sucesor (o al predecesor) de un nodo, que como hemos visto anteriormente, puede ser una operación frecuentemente necesaria. En general, los algoritmos que impliquen recorrer el árbol se podrán diseñar de una manera más eficiente.

Para poder manejar correctamente toda la información de la que se dispone en la representación hilvanada (con hilos) del árbol binario, es necesario poder distinguir entre lo que son punteros normales, que representan las relaciones reales entre los nodos, y lo que son hilos. Esto se puede hacer añadiendo dos campos booleanos a la representación de los nodos del árbol. Estos nuevos campos indicarán si los enlaces izquierdo y derecho son hilos o no.

La estructura de un nodo, siguiendo la representación utilizada en lenguaje Pascal, vendría dada por la siguiente declaración:

```
Type
      ArbolHilvanado = ^Nodo

      Nodo = record
                  Info: Valor;
                  HiloIzq, HiloDer: Boolean;
                  Izq, Der: ArbolHilvanado;
            end;
```

Con el objeto de no mantener absolutamente ningún enlace nulo y para facilitar el recorrido del árbol, se suele añadir a la estructura un nodo raíz que no contiene información real (como se ha visto para otras estructuras de datos). El nodo raíz que representa el árbol vacío tendrá la estructura que se muestra en la siguiente figura:

HiloIzq	*Izq*	*Info*	*Der*	*HiloDer*
CIERTO/FALSO	**NULO**	**--**	**NULO**	**CIERTO/FALSO**

Supongamos que el recorrido que se desea hacer del árbol es recorrido infijo. En este caso, cada puntero hilo enlazará con el sucesor infijo, si es un enlace derecho, y con el predecesor infijo, si es un enlace izquierdo. Se puede observar como, con esta nueva información, el proceso de recorrido del árbol se simplifica. Se comprueba que para cualquier nodo **`donde`** de un árbol binario enlazado de esta manera, se cumple que si **`donde^.HiloDer=CIERTO`**, el sucesor infijo de **`donde`** es **`donde^.Der`** (por definición de los hilos). En otro caso (**`donde^.HiloDer=FALSO`**), el sucesor de **`donde`** se obtiene siguiendo un camino de enlaces izquierdos desde **`donde^.Der`** hasta que se encuentre un nodo para el que **`HiloIzq=CIERTO`** (similar a la búsqueda del sucesor infijo en árboles binarios no hilvanados). El siguiente algoritmo encuentra el sucesor de cualquier nodo **`donde`** en un árbol binario hilvanado.

Algoritmo Sucesor_Infijo

Entrada
    arb: ArbolBinarioHilvanado
Salida
    ArbolBinarioHilvanado
Variable
    aux: ArbolBinarioHilvanado

Inicio
    aux ← arb^.Der
    Si no arb^.HiloDer **entonces**
        Mientras no aux^.HiloIzq **hacer**
            aux ← aux^.izq
        Fin_mientras
    Fin_si
    Devolver (aux)
fin

Es interesante destacar que ahora es posible encontrar el sucesor en orden infijo de cualquier nodo sin necesidad de utilizar una pila auxiliar (o recursividad), con lo que se reducen los requerimientos de almacenamiento en memoria durante el proceso.

De manera análoga, se puede encontrar el predecesor en orden infijo de cualquier nodo. Entonces se cumple que para un nodo **donde**, si **donde^.HiloIzq=CIERTO**, el predecesor infijo es **donde^.Izq**, sino el predecesor se obtiene siguiendo un camino de enlaces derechos desde **donde^.Izq** hasta que se encuentre un nodo para el que **HiloDer=CIERTO**.

Si se desea recorrer el árbol binario siguiendo el orden infijo, es posible diseñar un algoritmo iterativo que lo haga eficientemente mediante llamadas sucesivas al algoritmo que localiza el sucesor de un nodo.

```
Algoritmo Recorrido_Infijo

Entrada
        arb: ArbolBinarioHilvanado
Variable
        aux: ArbolBinarioHilvanado

Inicio
        aux ← arb
        Mientras aux ≠ arb hacer
                aux ← Sucesor_Infijo ( aux )
                Si ( aux = arb ) entonces
                        Procesar ( aux^.info )
                Fin_si
        Fin_mientras
Fin
```

Se puede comprobar que el coste computacional asociado a este algoritmo es O(n) para un árbol binario hilvanado con n nodos.

En general, la utilización de enlaces hilos simplifica los algoritmos de recorrido del árbol, por lo que resultan recomendables cuando el recorrido del árbol (o los movimientos parciales dentro del mismo) es una operación frecuente. Sin embargo, desde el punto de vista de la manipulación general del árbol, hay que tener en cuenta que la inserción de nuevos nodos debe mantener en todo momento esta estructura de enlaces con los nodos sucesor y predecesor, y que cada vez que se inserte un nodo se deberá comprobar si existen enlaces de este tipo que deban ser modificados o creados, además de los punteros normales que relacionan los nodos del árbol.

14.8. Representación de árboles generales como árboles binarios

Vamos a ver en este apartado que cualquier árbol se puede representar como un árbol binario. Esto es importante porque resulta más complejo manipular nodos de grado variable (número variable de relaciones) que nodos de grado fijo. Una posibilidad evidente de fijar un límite en el número de relaciones sería seleccionar un número **k** de hijos para cada nodo, donde **k** es el grado máximo para un nodo del árbol. Sin embargo, esta solución desaprovecha mucho espacio de memoria.

Lema 4: Para un árbol *k*-ario (es decir, grado **k**) con **n** nodos, cada uno de tamaño fijo, el número de enlaces nulos en el árbol es **n*(k-1)+1** de los **n*k** campos de tipo enlace existentes, **n≥1**.

Esto implica que para un árbol de grado 3, más de 2/3 de los enlaces serán nulos. Y la proporción de enlaces nulos se aproxima a 1 a medida que el grado del árbol aumenta. La importancia de poder utilizar árboles binarios para representar árboles generales reside en el hecho de que sólo la mitad de los enlaces son nulos, además de facilitar su manipulación.

Para representar cualquier árbol por un árbol binario, vamos a hacer uso implícito de que el orden de los hijos de un nodo en un árbol general no es importante.

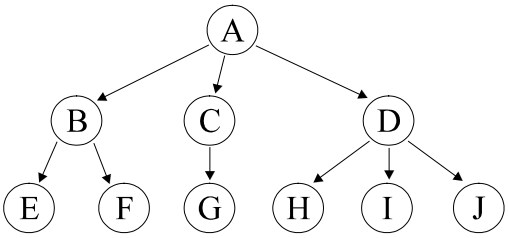


Figura 8

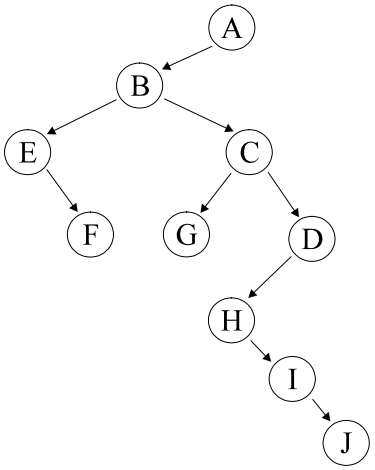


Figura 9

La razón por la cual, para representar un árbol general, se necesitarían nodos con muchos enlaces es que, hasta ahora, hemos pensado en una representación basada en la relación padre-hijo dentro del árbol, y un nodo pueden tener un gran número de hijos. Sin embargo, se puede encontrar una forma de representación donde cada nodo sólo necesite tener dos relaciones, una que lo una con el hijo que tenga más a la izquierda y otra que lo una con el siguiente nodo hermano por la derecha. Estrictamente hablando, como el orden de los hijos en un árbol no es importante, cualquiera de los hijos de un nodo podría ser el hijo que está más a la izquierda el nodo padre y cualquiera de los nodos hermanos podría ser el siguiente hermano por la derecha. Por lo tanto, la elección de estos nodos no dejará de ser, hasta cierto punto, arbitraria. Podemos basarnos para esta elección en la representación gráfica del árbol que se desea almacenar. Veamos el siguiente ejemplo, el árbol binario correspondiente al árbol de la figura se obtiene conectando juntos todos los hermanos de un nodo y eliminando los enlaces de un nodo con sus hijos, excepto el enlace con el hijo que tiene más a la izquierda.

Este tipo de representación se puede identificar con los árboles binarios que ya hemos visto, asociando el enlace izquierdo del árbol con el enlace al hijo de la izquierda y el enlace derecho con el enlace al nodo hermano. Se observa que de esta manera el nodo raíz nunca tendrá un subárbol derecho, ya que no tiene ningún nodo hermano. Pero esto nos permite representar un bosque de árboles generales como un único árbol binario, obteniendo primero la transformación binaria de cada árbol y después uniendo todos los árboles binarios considerando que todos los nodos raíz son hermanos. Ver el ejemplo de la figura.

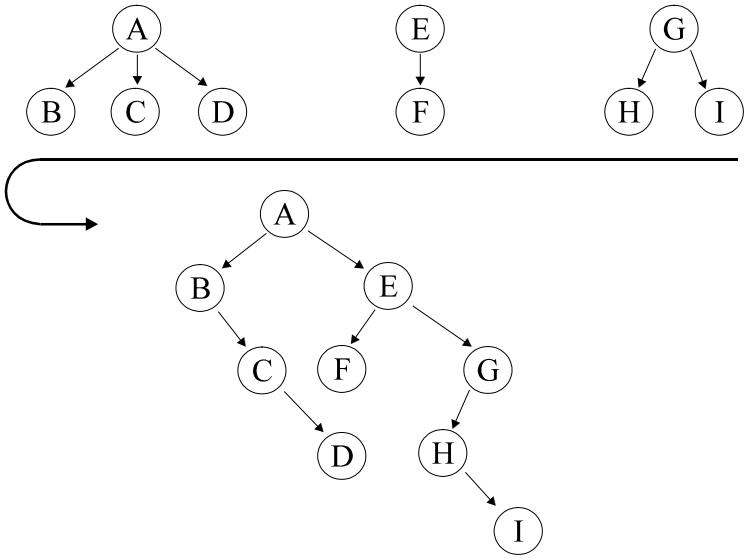


Figura 10

En todas estas transformaciones de árbol general a árbol binario hay que tener en cuenta la interpretación que se deben hacer de las relaciones, no es lo mismo que un nodo esté a la izquierda o a la derecha de otro, ya que los enlaces izquierdos unen un nodo con su hijo, mientras que los enlaces derechos unen dos nodos hermanos. De cualquier forma, teniendo en cuenta este aspecto, es posible aplicar los algoritmos de manipulación de árboles binarios que se han visto hasta ahora. De hecho, en todos estos algoritmos se diferenciaba claramente entre el tratamiento del subábol izquierdo y el del subárbol derecho.

14.9. Árboles binarios de búsqueda

Los árboles binarios de búsqueda son estructuras de datos que presentan un gran rendimiento cuando las funciones a realizar implican búsquedas, inserciones y eliminación de nodos. De hecho, con un árbol de búsqueda, dichas operaciones se pueden realizar tanto a partir de un valor clave, como por un valor ordinal (es decir, encontrar un elemento con clave **x**, encontrar el sexto elemento más pequeño, borrar el elemento con clave **x**, borrar el sexto elemento más pequeño, insertar un elemento y determinar su ordinal dentro del árbol).

Definición: Un árbol binario de búsqueda es un árbol binario, que puede estar vacío, y que si es no vacío cumple las siguientes propiedades:

(1) Todos los nodos están identificados por una clave y no existen dos elementos con la misma clave.

(2) Las claves de los nodos del subárbol izquierdo son menores que la clave del nodo raíz.

(3) Las claves de los nodos del subárbol derecho son mayores que la clave del nodo raíz.

(4) Los subárboles izquierdo y derecho son también árboles binarios de búsqueda.

La primera propiedad resultaría redundante, ya que de las propiedades (2), (3) y (4) se puede deducir que la clave de un nodo es única. Sin embargo, la aparición explícita de esta propiedad hace que la definición sea más clara.

Operaciones en árboles binarios de búsqueda

Veamos ahora cómo manipular este tipo especial de árboles binarios.

En este tipo de árboles tendremos como operaciones fundamentales las operaciones de búsqueda, inserción y borrado de elementos.

Con estas operaciones y las básicas de árboles generales la interfaz de la clase nos quedaría:

```
class ABB //Arbol Binario de Busqueda
{
    public:
        typedef .??. Valor;

        ABB();
        ABB (const ABB &);
        ~ABB();

        bool ABBVacio() ;
        bool Informacion ( Valor& ) ;
        ABB& HijoIzq() ;
        ABB& HijoDer();

        void Asignar ( ABB& );
        void Borrar ();
        ABB& Buscar ( Valor );
        bool Insertar ( Valor );
```

```
        bool Eliminar ( Valor );

    private:
        typedef ABB * Puntero;

        bool esvacio;
        Valor info;
        Puntero izdo, dcho;

        void Copiar (ABB &);
};
```

Nos centraremos en las operaciones 'especiales' de árboles binarios de búsqueda, ya que el resto de las operaciones son las mismas que en los árboles binarios generales.

Búsqueda

La definición de árbol binario de búsqueda especifica un criterio en la estructuración del árbol en función de las claves de los nodos. En este caso, existe un criterio de ordenación de los nodos. Por lo tanto, será bastante simple describir un método eficiente de búsqueda que explote esa ordenación.

Suponer que se busca un elemento que posea una clave x. La búsqueda comenzará por el nodo raíz del árbol. La clave de ese nodo informará por dónde debe continuar la búsqueda, ya no es necesario recorrer exhaustivamente todos los nodos del árbol. Si la clave del nodo es igual a x, la búsqueda finaliza con éxito. Si la clave es menor que x, se sabe que si existe un nodo en el árbol que posea como clave el valor x deberá estar en el subárbol derecho, por lo tanto la búsqueda deberá continuar por esa parte del árbol. Si, por el contrario, la clave del nodo es mayor que x, entonces la búqueda deberá continuar por el subárbol izquierdo. El proceso continuará hasta que se encuentre un nodo con clave igual a x o un subárbol vacío, en cuyo caso se puede asegurar que no existe ningún nodo con clave x en el árbol. Este método de búsqueda sugiere seguir un esquema recursivo.

Suponiendo que la clave que identifica cada nodo es un campo contenido en la información general del nodo, el algoritmo de búsqueda quedaría como sigue:

Algoritmo Buscar_Recurrente_ABB
{ Buscar en el árbol* arb *la clave* x **}*
{ Devolver la posición del nodo donde se encuentre,* ***NULO*** *si no está*}*

Entradas
Valor(clave) x
ArbolBinario arb

Salida
ArbolBinario

Inicio
Si$_{(1)}$ (arb es un árbol vacío) **entonces**
Devolver (Árbol vacío)
Sino$_{(1)}$
Si$_{(2)}$ (x = Informacion(clave) de arb) **entonces**
Devolver (arb)
Sino$_{(2)}$
Si$_{(3)}$ (x < Informacion(claver) de arb) **entonces**
Devolver (Buscar_Recurrente_ABB (x, Hijo izquierdo de arb))
Sino$_{(3)}$

```
                    Devolver (Buscar_Recurrente_ABB (x, Hijo derecho de arb) )
                Fin_si(3)
            Fin_si(2)
        Fin_si(1)
Fin
```

En este ejemplo, la recursividad puede ser fácilmente susituida por un esquema iterativo mediante la utilización de un bucle de repetición mientras. En ese caso, el algoritmo de búsqueda iterativo sería:

```
Algoritmo Buscar_Iterativo_ABB
{ Buscar en el árbol arb la clave x                                    }
{ Devolver la posición del nodo donde se encuentre, NULO si no está }

Entradas
    Valor(clave) x
    ArbolBinario arb

Salida
    ArbolBinario

Variables
    ArbolBinario aux
    Boolean encontrado: (CIERTO, FALSO)

Inicio
    aux ← arb
    encontrado ← falso
    Mientras (aux no sea árbol vacío) y (NO encontrado) hacer
        Si(1) (x = informacion (clave) de aux) entonces
            encontrado ← CIERTO
        Sino(1)
            Si(2) (x < Informacion (clave) de donde) entonces
                aux ← Hijo izquierdo de aux
            Sino(2)
                aux ← Hijo derecho de aux
            Fin_si(2)
        Fin_si(1)
    Fin_mientras

    Si(3) (encontrado) entonces
        Devolver (aux)
    Sino(3)
        Devolver (Árbol vacío)
    Fin_si(3)
Fin
```

El proceso de búsqueda en un árbol binario de búsqueda resulta muy eficaz. El coste asociado sería O(h), donde h es la altura del árbol. Hay que hacer notar que la dependencia es lineal con la altura del árbol y no con el número de elementos del mismo. En función del número de nodos en el árbol, n, el coste sería O(log n) (Esto es cierto si el árbol está suficientemente equilibrado. En el peor de los casos el árbol binario de búsqueda degeneraría en una lista y el coste llegaría a ser lineal.)

El método de búsqueda se asemeja mucho al método de búsqueda binaria sobre arrays ordenados, tras la comparación con un elemento se puede decidir en qué región de la estructura se puede encontrar la información buscada, descartándose el resto de los elementos de la estructura. De esta forma, se reduce considerablemente el número de comparaciones necesarias para localizar el elemento.

Los algoritmos descritos, escritos en C++ podrían quedar de la siguiente manera:

```
Arbol & Arbol::BuscarRecurrenteABB (Valor x)
{
    PunteroArbol p_aux;

    if (esvacio == true)
        p_aux = this;
    else
        if (info == x)
            p_aux = this;
        else
            if (x < info)
                p_aux = &(izdo.BuscarRecurrenteABB (x) );
            else
                p_aux = &(dcho.BuscarRecurrenteABB (x) );

    return (*p_aux);
}
```

```
Arbol & Arbol::BuscarIterativoABB (Valor x)
{
    PunteroArbol p_aux;

    p_aux = this;

    while ( (p_aux->esvacio != true) && (p_aux->info != x) )
    {
        if (x < p_aux->info)
            p_aux = p_aux->izdo;
        else
            p_aux = p_aux->dcho;
    }

    return (*p_aux);
}
```

En cuanto a buscar nodos en función de la posición ocupada en el orden de las claves, es decir, por el ordinal de las claves, es necesario añadir a cada nodo un campo adicional que guarde el número de elementos que posee su subárbol izquierdo más uno (sea este campo **`TamIzq`**). De esta forma, el siguiente algoritmo permitiría realizar la búsqueda:

Algoritmo Buscar_Orden

Entrada
k: entero
arb: ArbolBinario

```
Salida
        ArbolBinario

Variables
        aux: ArbolBinario
        encontrado: (CIERTO, FALSO)
        i: entero

Inicio
        aux ← arb
        encontrado ← FALSO
        i ← k
        Mientras ( aux ≠ nulo ) y ( NO encontrado ) hacer
                Si(1) ( i = aux^.TamIzq ) entonces
                        encontrado ← CIERTO
                Sino(1)
                        Si(2) ( i < aux^.TamIzq ) entonces
                                aux ← aux^.Izq
                        Sino(2)
                                i ← i - aux^.TamIzq
                                aux ← aux^.Der
                        Fin_si(2)
                Fin_si(1)
        Fin_mientras
        Si(3) ( encontrado ) entonces
                Devolver ( aux )
        Sino(3)
                Devolver ( NULO )
        Fin_si(3)
Fin
```

Como se puede observar, en este caso también es posible realizar el proceso de búsqueda con un coste que depende linealmente de la altura del árbol.

Inserción

La inserción de un nuevo nodo en un árbol binario de búsqueda debe realizarse de tal forma que se mantengan las propiedades del árbol. De modo que lo primero que hay que hacer es comprobar que en el árbol no existe ningún nodo con clave igual a la del elemento que se desea insertar. Si la búsqueda de dicha clave falla, entonces se insertará el nuevo nodo en el punto donde se ha parado la búsqueda, que será el punto del árbol donde, de existir, debería estar ubicada dicha clave. Hay que considerar que todo proceso de búsqueda fallido termina en un enlace nulo, por tanto, el nodo a insertar siempre será un nodo terminal, lo que facilitará la operación.

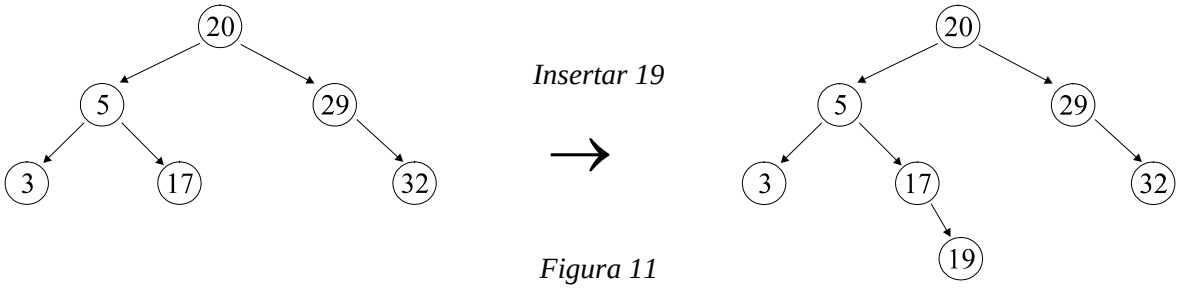


Figura 11

El algoritmo que implementa la estrategia de inserción es el siguiente:

```
Algoritmo Insertar

Entrada
    Valor x
    ArbolBinario (por referencia) arb

Salida
    (CIERTO, FALSO)        {* el algoritmo informa si ha sido posible la inserción *}

Variables
    PunteroArbol donde, padre
    encontrado: (CIERTO, FALSO)

Inicio
    {*    buscar x.clave en el árbol, donde indica la localización        *}
    {*    si donde=NULO entonces no está y padre es el padre de donde     *}
    padre ← Arbol Vacío
    donde ← arb
    encontrado ← FALSO
    Mientras ( (donde no sea un árbol vacío) y (no encontrado) ) hacer
        padre ← donde
        Si(1) (x.clave = Informacion (clave) de donde) entonces
            encontrado ← CIERTO
        Sino(1)
            Si(2) (x.clave < Informacion (clave) de donde) entonces
                donde ← Hijo izquierdo de donde
            Sino(2)
                donde ← Hijo derecho de donde
            Fin_si(2)
        Fin_si(1)
    Fin_mientras

    {insertar}
    Si(3) (no encontrado) entonces
        donde ← Crear_espacio
        Hijo izquierdo de donde ← Árbol vacío
        Hijo derecho de donde ← Árbol vacío
        Información de donde ← x
        Si(4) (padre es árbol vacío) entonces
            arbol ← donde              {* insertar el primer nodo, la raíz *}
        Sino(4)
            Si(5) (x.clave < informacion (clave) de padre) entonces
                Hijo izquiedo de padre ← donde
            Sino(5)
                Hijo derecho de padre ← donde
            Fin_si(5)
        Fin_si(4)
        Devolver (CIERTO)
    Sino(3)
        Devolver (FALSO)
    Fin_si(3)
Fin
```

Esta función se podría implementar fácilmente de forma recursiva de la siguiente manera:

Algoritmo Insertar

Entrada
valor x
PunteroArbol arb (por referencia)

Salida
(**CIERTO**, **FALSO**) *{* el algoritmo informa si ha sido posible la inserción *}*

Inicio
Si$_{(1)}$ (arb es árbol vacío) **entonces**
arb ← **Crear_espacio**
Hijo izquierdo de arb ← Árbol vacío
Hijo derecho de arb ← Árbol vacío
informacion de arb ← x
Devolver (**CIERTO**)
Sino$_{(1)}$
Si$_{(2)}$ (x.clave = Informacion (clave) de arb) **entonces**
Devolver (**FALSE**)
Sino$_{(2)}$
Si$_{(3)}$ (x.clave < Informacion (clave) de arb) **entonces**
Devolver (Insertar (Hijo izquierdo de arb, x))
Sino$_{(3)}$
Devolver (Insertar (Hijo derecho de arb, x)
Fin_si$_{(3)}$
Fin_si$_{(2)}$
Fin_si$_{(1)}$
Fin

Estos algoritmo en C++, quedarían como sigue:

```
bool Arbol::InsertarIterativo (Valor x)
{
    PunteroArbol donde;
    bool enc = false, exito = false;

    while ( (donde->esvacio != true) && (!enc) )
    {
        if (x == donde->info)
            enc = true;
        else
            if (x < donde->info)
                donde = donde->izdo;
            else
                donde = donde->dcho;
    }

    if (!enc)
    {
        exito = true;
        donde->esvacio = false;
        donde->izdo = new Arbol;
        donde->dcho = new Arbol;
        donde->info = x;
    }
}
```

```
bool Arbol::InsertarRecurrente (Valor x)
{
    bool exito = false;

    if (esvacio == true)
    {
        éxito = true;

        esvacio = false;
        izdo = new Arbol;
        dcho = new Arbol;
        info = x;
    }
    else
        if (x != info)
            if (x < info)
                exito = izdo->InsertarRecurrente (x);
            else
                exito = dcho->InsertarRecurrente (x);

    return exito;
}
```

Si nos fijamos, la primera parte de la inserción es la búsqueda del lugar en dónde queremos insertar. Podemos utilizar el método de búsqueda para evitar esa parte. Utilizando el método de búsqueda (nos daría igual el método iterativo o el recursivo para hacerlo) nos quedaría la siguiente inserción:

```
bool Arbol::Insertar (Valor x)
{
    bool exito = false;
    Arbol & donde = Buscar (x);

    if (donde->esvacio)
    {
        exito = true;
        donde->esvacio = false;
        donde->izdo = new Arbol;
        donde->dcho = new Arbol;
        donde->info = x;
    }

    return exito;
}
```

Si los nodos del árbol poseen el campo **`TamIzq`**, comentado anteriormente, en el algoritmo de inserción habría que considerar la actualización de este campo en aquellos nodos donde sea necesario.

La operación de inserción de un nodo se puede hacer en un tiempo O(h), donde 'h' es la altura del árbol de búsqueda.

Borrado

La operación de borrado en un árbol de búsqueda es muy simple. Hay que tener en cuenta que en todo árbol de búsqueda los nodos se pueden ordenar por su clave si se recorre el árbol siguiendo el

criterio infijo. El orden establecido por este recorrido se debe mantener aunque se elimine un nodo del árbol.

Podemos establecer como caso básico la eliminación de un nodo que no tenga hijos. Si ese es el caso la eliminación de la información es inmediata (se marca el nodo como vacío y se libera el espacio de sus hijos vacios).

Si no es este el caso se busca su sucesor en el árbol (que estará a la izquierda del árbol que es hijo derecho del nodo a borrar y se sustituye la información de ese nodo por la que pretendemos borrar y se llama de nuevo a borrar con ese nodo. Si no existiese un sucesor, existiría un antecesor y realizariamos la misma operación.

La implementación en C++ de este código, sería la siguiente:

```
bool ABB::Eliminar (Valor x)
{
    Puntero p_aux, p_tmp;
    bool exito;

    p_aux = & (this->Buscar (x) );

    if (p_aux->esvacio)
        exito = false;
    else
    {
        //Intento sustituir por el menor en p_aux->dcho
        p_tmp = p_aux->dcho;
        if ( !p_tmp->esvacio)
        {
            while ( !p_tmp->izdo->esvacio)
                p_tmp = p_tmp->izdo;
            p_aux->info = p_tmp->info;
            exito = p_tmp->Eliminar(p_tmp->info);
        }
        else
        {
            //Intento sustituir por el mayor en aux->izq
            p_tmp = p_aux->izdo;
            if ( !p_tmp->esvacio)
            {
                while ( !p_tmp->dcho->esvacio)
                    p_tmp = p_tmp->dcho;
                p_aux->info = p_tmp->info;
                exito = p_tmp->Eliminar(p_tmp->info);
            }
            else //Nodo terminal
            {
                p_aux->esvacio = true;
                delete p_aux->izdo;
                delete p_aux->dcho;
                exito = true;
            }
         }
    }

    return exito;
}
```

Ejemplo de aplicación de árboles binarios de búsqueda

Contar el número de veces que aparecen las palabras que forman un texto.

Algoritmo:

- Leer el texto palabra a palabra.
- Almacenar las palabras leidas en forma de árbol binario de búsqueda con un campo asociado que indique el núnero de veces qe aparece cada una.
- Cuando se lee una nueva palabra, comprobar si está o no en el árbol, lo que indica si ya había aparecido anteriormente o no. Si la palabra está en el árbol aumentar su contador de apariciones en 1, sino insertarla en el árbol e iniciar el contador a 1.
- Cuando se alcance el final del texto se tendrá toda la información de interés en forma de árbol de búsqueda, si se recorre este árbol en orden infijo se podrá obtener un listado en orden alfabético de las palabras con la frecuencia de aparición de cada palabra.

Escribiendo este algoritmo en forma de programa en Pascal, éste podría quedar como sigue:

```
Program Frecuencia_Palabras (input,output);

{* Contar la frecuencia de aparición de las palabras de un texto *}
{* almacenado en un fichero                                      *}
{* Ejemplo de utilización de árboles binarios de búsqueda        *}

Type
    Cadena = String[25];
    Numeros = 1..MaxInt;

    Valor = Record
                Pal : Cadena;
                Cont: Numeros;
            End;

    PunteroArbol = ^NodoArbol;

    NodoArbol = record
                    Info    : Valor;
                    Izq, Der: PunteroArbol;
                end;

Var
    nombre : Cadena;
    fichero: Text;
    letras : Set Of Char;

{* Leer el texto palabra a palabra *}
{* La función indica si se ha alcanzado el final del fichero *}

Function Leer ( Var texto: Text; Var palabra: Cadena): Boolean;

Var
    fin: Boolean;
    ch : Char;

Begin
    palabra := '';
    fin := FALSE;
```

```
    While ( ( Not Eof ( texto ) ) and ( Not fin ) ) Do
    Begin
        Read ( texto, ch );
        If ( ch In letras ) Then
            palabra := palabra + ch
        Else
            If ( palabra <> '' ) Then
                fin := TRUE;
     End;
     leer := Eof ( texto );
End;

{* Recorrido infijo del árbol escribiendo la informacion de interes *}

Procedure Infijo ( arbol: PunteroArbol );

Begin
    If ( arbol <> NIL ) Then
    Begin
        Infijo ( arbol^.Izq );
        WriteLn ( arbol^.Info.Pal:15, ':':3, arbol^.Info.Cont:3 );
        Infijo ( arbol^.Der );
    End
End;

{* Operacion de insercion en el arbol de busqueda *}

Procedure Insertar ( x: Cadena; Var arbol: PunteroArbol );

Var
    nodo, padre: PunteroArbol;
    encontrado : Boolean;

Begin
    padre := NIL;
    nodo := arbol;
    encontrado := FALSE;
    While ( ( nodo <> NIL ) And ( Not encontrado ) ) Do
    Begin
        padre := nodo;
        If ( x = nodo^.Info.Pal ) Then
            encontrado := TRUE
        Else
            If ( x < nodo^.Info.Pal ) Then
                nodo := nodo^.Izq
            Else
                nodo := nodo^.Der;
    End;
    If ( Not encontrado ) Then
    Begin
        New ( nodo );
        nodo^.Izq := NIL;
        nodo^.Der := NIL;
        nodo^.Info.Pal := x;
        nodo^.Info.Cont := 1;
        If ( padre = NIL ) Then
            arbol := nodo
        Else
            If ( x < padre^.Info.Pal ) Then
                padre^.Izq:=nodo
            Else
                padre^.Der:=nodo
    End
```

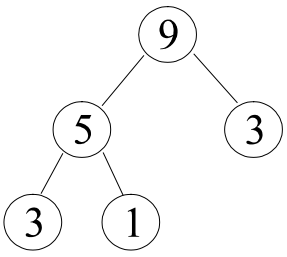
```
    Else
        nodo^.cont := nodo^.cont + 1;
End;

{* Algoritmo para contar la frecuencia de aparición de las palabras *}
Procedure Frecuencia ( Var texto: Text );

Var
    arbol    : PunteroArbol;
    palabra  : Cadena;
    fin_texto: Boolean;

Begin
    arbol := NIL;
    fin_texto := FALSE;
    While ( Not fin_texto ) Do
    Begin
        fin_texto := Leer ( texto, palabra );
        If ( palabra <> '' ) Then
            insertar ( palabra, arbol );
    End;
    infijo ( arbol );
End;

{* Bloque principal del programa *}
BEGIN
    letras := ['a'..'z', 'A'..'Z'];
    Write ( 'Nombre del fichero a analizar: ' );
    ReadLn ( nombre );
    Assign ( fichero, nombre);
    Reset ( fichero );
    frecuencia ( fichero );
    Close ( fichero )
END.
```

■

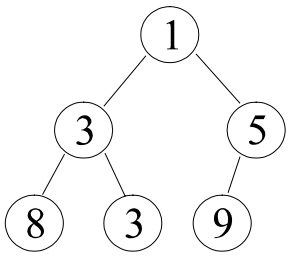
14.10. Heaps (montículos)

Veamos otro tipo especial de árbol binario, los llamados *heaps* (motículos), que se pueden representar eficazmente con un vector.

Definición: Un heap de máximos (mínimos), *max heap* (*min heap*), es un árbol binario completo tal que, el valor de la clave de cada nodo es mayor (menor) o igual que las claves de sus nodos hijos (si los tiene).



Montículo de máximos
max heap



Montículo de mínimos
mineap

Figura 12

De la definición se deduce que la clave contenida en el nodo raíz de un *max heap* (*min heap*) es la mayor (menor) clave del árbol. Pero esto no quiere decir que sea la única con ese valor.

La estructura heap tiene interesantes aplicaciones, por ejemplo, la ordenación de arrays (algoritmo heapsort) o el mantenimiento de las llamadas colas de prioridad. Las colas de prioridad son un tipo especial de colas donde todo elemento que se inserta en la cola lleva asociado una prioridad, de forma que el elemento que se borra de la cola es el primero entre los que tengan la máxima (o mínima) prioridad.

Los heaps, como árboles binarios completos que son, se pueden representar de forma eficaz mediante una estructura secuencial (un array). A cada nodo del árbol se le asigna una posición dentro de la secuencia en función del nivel del árbol en el que se encuentra y dentro de ese nivel de la posición ocupada de izquierda a derecha. Por ejemplo:

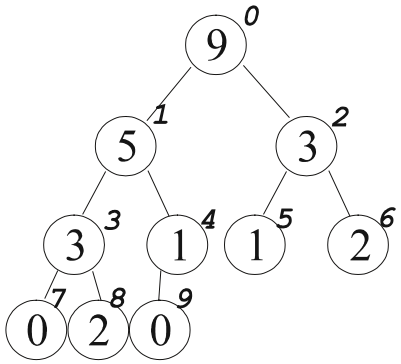


Figura 13

De forma que para un nodo representado por el elemento A[i] se cumple que:

(i) El nodo padre estará localizado en la posición **`[(i-1) / 2]`**, si **`i > 0`**.
El nodo raíz, **`i=0`**, no tiene padre).

(ii) Si tiene hijos estarán localizados en las posiciones **`[2(i+1)-1] y [2(i+1)]`**.
Si **`i > [(n+1) / 2 – 1]`**, el nodo no tiene hijos.

Una característica fundamental de esta estructura de datos es la propiedad que caracteriza a un motículo puede ser restaurada eficazmente tras cualquier modificación de un nodo (cambiar el valor de la clave, insertar un nodo, borrar un nodo, etc.).

Representación de montículos

Ya se ha comentado que por ser el montículo un árbol lleno, es eficiente su representación mediante *arrays*. De manera que la representación del montículo contendrá necesariamente un *array* donde guardar la información. Esto supondrá tener que estimar un máximo para la cantidad de elementos que podemos guardar en el montículo.

Además del *array* será necesario llevar la cuenta de cuantos de los elementos del *array* son válidos (forman parte del *heap*).

```
Tipos:
        Heap = registro
                        Info: Vector [0..MAX-1] de Valor;
                        Num : Entero entre 0 y MAX-1;
               fin
```

A partir de esta información si hacemos la implementación en una clase de C++, la parte privada de la clase *heap* quedaría como sigue:

```
const int MAX = ??

class Heap
{
   public:
      ...
   private:
      Valor Info[MAX];
      int Num;
}
```

Operaciones básicas sobre montículos

Supongamos que se trata de un montículo de máximos (cualquier comentario sobre este tipo de montículos se puede extender fácilmente a los montículos de mínimos). En este caso, se debe cumplir que la clave de cualquier nodo sea mayor o igual que las claves de sus hijos. Si se modifica el valor de un nodo incrementando su valor, entonces es posible que la propiedad del montículo no se cumpla con respecto a sus nodos antecesores, no respecto a sus hijos. Esto implicaría ir ascendiendo por el árbol comprobando si el nodo modificado es mayor que el padre, si es así intercambiarlos y repetir el proceso en un nivel superior hasta que se encuentre que no es necesario realizar ningún intercambio o que se llegue al nodo raíz. En cualquier caso, tras ese proceso se habrá restaurado la propiedad del montículo. Este algoritmo quedaría como sigue:

```
Algoritmo Subir
{* Se utiliza cuando se incrementa el valor del nodo situado en la posicion pos *}
Entradas
      Heap arb
      Posicion pos (índice)

Inicio
      Si ( (pos > 0) y (arb.Info[pos] > arb.Info[(pos – 1) / 2] ) entonces
            Intercambiar (arb.Info[pos] , arb.info[(pos – 1) / 2] )
            Subir (arb, (pos – 1) / 2 )
      Fin_si
Fin
```

Si por el contrario, modificamos un nodo haciendo disminuir su valor, el problema consistirá en comprobar la propiedad respecto a sus descendientes. Si la clave del nodo es mayor que las claves de sus hijos, entonces la propiedad se sigue cumpliendo, si no es así, hay que intercambiar la clave de este nodo con la del hijo que tenga la clave máxima y repetir todo el proceso desde el principio con este nodo hijo hasta que no se realice el intercambio.

```
Algoritmo Bajar
{* Se utiliza cuando disminuye el valor del nodo situado en la posicion pos *}
Entradas
      Heap arb
      Posicion pos (índice)

Variables
      Posición max (índice)

Inicio
```

```
        max ← pos
        Si ( ( (2*(pos+1) – 1) < n) y (arb.Info[2*(pos+1)-1] > arb.Info[max] ) ) entonces
                max ← 2*(pos+1)-1
        Fin_si

        Si ( (2*(pos+1) < n ) y ( arb.Info[2*(pos+1)] > arb.Info[max] ) entonces
                max ← 2*(pos+1)
        Fin_si

        Si ( max ≠ pos ) entonces
                Intercambiar ( arb[pos], arb[max] )
                Bajar (arb, max)
        Fin_si
Fin
```

A partir de estos dos algoritmos básicos que permiten restaurar la estructura del montículo, se pueden definir las operaciones de inserción y borrado de elementos en esta estructura de datos.

Inserción

Al intentar añadir un nuevo elemento al montículo, no se sabe cuál será la posición que ocupará el nuevo elemento de la estructura, pero lo que si se sabe es que, por tratarse de un árbol completo, dónde se debe enlazar el nuevo nodo, siempre será en la última posición de la estructura.

Por ejemplo, en el siguiente montículo formado por cinco elementos, si se intenta añadir un nuevo elemento, sea el que sea, se conoce cuál será la estructura final del árbol:

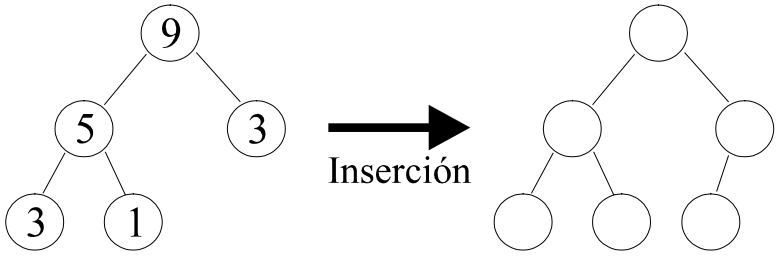


Figura 14

Si se almacena en el nuevo nodo la información que se desea insertar, lo más probable será que la propiedad del montículo no se conserve. Como el nuevo nodo siempre es terminal, para mantener la propiedad del montículo bastará con aplicar el algoritmo subir a partir del nodo insertado, con lo que el algoritmo de inserción resulta verdaderamente simple.

```
Algoritmo Insertar

Entrada
        Heap arb
        Valor x

Inicio
        arb.Num ← arb.Num + 1
        arb.Info[arb.Num - 1] ← x
        Subir (arb, arb.Num - 1)
Fin
```

En C++ el algoritmo sería:

```
bool Monticulo::Insertar (Monticulo::Valor x)
{
    bool exito;

    if (n == MAX)
        exito = false;
    else
    {
        exito = true;
        info[n] = x;
        n++;
        Subir (n - 1);
    }
    return exito;
}
```

Borrar (una posición '*pos*' cualquiera)

Al igual que en la operación de inserción, hay que tener en cuenta que el montículo es un árbol completo y que, para que se mantenga esa propiedad, el nodo que debe desaparecer realmente es el que está situado en la última posición. Por lo tanto, la estrategia a seguir para borrar cualquier nodo del montículo podría ser: sustituir la información del nodo a borrar por la del útlimo nodo del montículo y considerar que el árbol tiene un nodo menos (n-1); como con esta modificación seguramente se habrá alterado la propiedad del montículo, entonces aplicar el algoritmo subir o bajar, lo que sea preciso, para restaurar esa propiedad. La aplicación del algoritmo subir o del algoritmo bajar dependerá de que la modificación de la información del nodo haya implicado un incremento (aplicar subir) o una disminución (aplicar bajar) de la clave.

```
Algoritmo Borrar

Entrada
    Heap arb
    Posición pos (índice)

Variable
    x: Valor

Inicio
    x ← arb.info[pos]
    arb.Info[pos] ← arb.Info[arb.Num - 1]
    arb.Num ← arb.Num - 1
    Si(1) (arb.Info[pos] ≠ x) entonces
        Si(2) (arb.Info[pos] > x ) entonces
            Bajar (arb, pos)
        Sino(2)
            Subir (arb, pos)
        Fin_si(2)
    Fin_si(1)
Fin
```

Aplicación de los montículos

Los montículos tienen como aplicación fundamental el mantenimiento de las llamadas colas de prioridad. Esta estructura de datos es un tipo especial de cola, donde cada elemento lleva asociado un valor de prioridad, de forma que cuando se borra un elemento de la estructura, éste será el primero de los elementos con la mayor (menor) prioridad. En cualquier instante se puede insertar un elemento con prioridad arbitraria. Si la utilización de la cola de prioridad requiere borrar el elemento con la mayor (menor) prioridad, entonces se utiliza para su representación un montículo de máximos (mínimos), donde resulta inmediata la localización de ese elemento.

En un montículo de máximos resulta inmediato obtener el valor máximo de las claves del árbol. Como se ha visto anteriormente, este valor está siempre situado en la raíz del árbol, por lo tanto bastaría con acceder al primer elemento del array A[0].

En C++:

```
bool Monticulo::Maximo (Monticulo::Prioridad& res)
{
    bool exito;

    if (MonticuloVacio () )
        exito = false;
    else
    {
        exito = true;
        res = arbol[0];
    }
    return exito;
}
```

Y borrar el máximo sería pues eliminar la posición cero del montículo.

```
bool Monticulo::EliminarMax ()
{
    bool exito;

    if (MonticuloVacio () )
        exito = false;
    else
    {
        exito = true;

        info[0] = info[n];
        n--;
        Bajar(0);
    }
    return exito;
}
```

Con estas operaciones definidas como básicas quedaría la interfaz de la clase como sigue:

```
class Monticulo //Monticulos binarios de maximos
{
    public:
        typedef int Valor;

        Monticulo (void);
```

```
        bool MonticuloVacio (void);

        bool Maximo (Valor &);
        bool EliminarMax (void);
        bool Insertar (Valor);

    private:
        typedef Valor Vector[MAX];

        Vector info;
        int n;

        void Subir (int);
        void Bajar (int);
};
```

Quedaría por implementar el constructor por defecto y el método '**MonticuloVacio**'.

Veamos algunos ejemplos prácticos de colas de prioridad:

Ejemplo 1:

Supongamos que se desea informatizar la lista de espera para la realización de operaciones quirúrgicas en un quirófano de un hospital. Se desea utilizar el quirófano de la manera más eficiente posible, de forma que en todo instante se opere al enfermo que más lo necesite siguiendo el orden de la lista de espera. En ese caso, es posible asignar algún tipo de prioridad a cada uno de los enfermos, en función, por ejemplo, de la gravedad de la operación, del tiempo de ocupación del quirófano o incluso, del precio de la operación. De esta forma, cada vez que el quirófano quede libre se recurirá a la lista de espera y se seleccionará a aquel enfermo con la prioridad más alta.

Desde el punto de vista de programación del proceso, se podría utilizar una representación de la lista de espera en función de un montículo de máximos, siendo la información asociada con los nodos del árbol los datos de cada enfermo y la clave de identificación la prioridad de los mismos. Cuando un enfermo ingresa para ser operado en el hospital, se le asigna una prioridad y se insertan sus datos como un nuevo nodo del montículo (operación insertar), cuando el quirófano queda libre, se ocupará con el enfermo cuyos datos estén situados en la raíz del montículo (máxima prioridad) y se reestructurará el mismo.

Ejemplo 2:

La gestión de los recursos de un ordenador en un entorno multiusuario (CPU, memoria, disco, periféricos, etc.) se suele realizar también utilizando colas de prioridad. El problema se asemeja mucho al del ejemplo anterior. Existe un dispositivo que se desea utilizar y existen varios posibles usuarios, por lo tanto hay que seleccionar aquel usuario que permita utilizar de la manera más eficiente posible dicho dispositivo. Para ello, el sistema operativo asigna una prioridad a cada petición de utilización, de manera que se atiende a las peticiones en función de la prioridad y el orden temporal en que se han realizado. La prioridad asignada a cada prioridad puede depender de varias cosas. el tipo de usuario que la realiza (en un sistema multiusuario, no todos los usuarios tienen el mismo tratamiento, existen usuarios con más prioridad que otros), el tiempo de utilización del dispositivo solicitado (normalmente un trabajo que requiera poco tiempo de utilización del dispositivo se deberá atender antes que otro que requiera mucho más tiempo, de esta manera se agiliza el tiempo de respuesta del sistema), el tiempo que hace que el usuario no accede a ese recurso, etc... En base a todas estas consideraciones se puede establecer una prioridad a cada petición de

utilización de un recurso del ordenador y mantener una cola de prioridad (generalmente un montículo de máximos) para cada uno de estos recursos que gestione eficientemente su uso.

■

14.11. Ordenación con árboles

En los dos últimos tipos de árboles binarios comentados, se ha visto como se puede organizar eficientemente la información en estas estructuras en base a una clave de información, de manera que sea fácil extraer información relacionada con esa clave (buscar por clave, acceder a la clave máxima, acceder a la clave mínima, etc.). Esto hace que parezca evidente la utilización de estas estructuras de datos para resolver un problema tan habitual como pueda ser el problema de la ordenación de información.

Si nos centramos en el problema de la ordenación interna sobre *arrays*, que fue el problema estudiado en su momento, se puede pensar en los montículos como el tipo de árboles que podrían resolver el problema. Hay que tener en cuenta que, normalmente, la representación de los montículos se hace precisamente con *arrays* y que en estas estructuras es muy fácil acceder al elemento con clave máxima (o mínima, según sea la organización del montículo), con lo que se podría establecer algún tipo de algoritmo que aprovechando esta propiedad ordene la información del *array*.

Si consideramos que inicialmente el *array* puede estar arbitrariamente ordenado, esto implicará que, en general, no se cumplirá la propiedad de montículo de máximos, por lo que habría que empezar por formar un montículo a partir de ese *array*. Una vez formado el montículo, es fácil saber qué elemento debe ocupar la última posición del array ordenado, ese elemento siempre será el que ocupa la raíz del árbol (el primero del array). Si se intercambia el primer elemento con el último, se habrá logrado ubicar correctamente el elemento máximo. Se puede ahora considerar que el array tiene un elemento menos (el que está bien colocado) y reorganizar el montículo con un elemento menos. Si se repite este proceso hasta que se tenga un montículo con un único elemento, se puede asegurar que el *array* estará finalmente ordenado. Este método de ordenación se conoce con el nombre de *Heapsort* y el algoritmo sería el siguiente:

```
Algoritmo Heapsort

Entrada
    Array[0..n - 1] vect

Variable
    Indice i

Inicio
    Hacer_Heap (vect)
    Desde i ← n - 1 hasta 1 hacer
        Intercambiar (vect[0], vect[i])
        Bajar (vect[0..i-1], 0)
    Fin_desde
Fin
```

Para terminar, haría falta especificar cómo se puede construir un montículo de máximos a partir de un array arbitrariamente ordenado. La solución es bastante simple, ir formando montículos de tamaño creciente hasta formar uno que contenga los n elementos del array. Comenzando desde atrás hacia adelante, se puede considerar que los **`n div 2`** últimos elementos son nodos terminales y que, por lo tanto, forman montículos de tamaño **`1`** que no requieren ningún tipo de reorganización. Si consideramos ahora secuencialmente los elementos a partir de **`n div 2`** hasta **`1`**, es como si se fuese

añadiendo cada vez un nodo más a cada uno de los montículos ya formados, como esta inserción modifica la estructura del montículo corespondiente, es necesario reorganizarlo mediante el procedimiento bajar. Al final del proceso, se puede asegurar que el array estará organizado en forma de montículo de máximos. El algoritmo sería el siguiente:

Algoritmo Hacer_Heap

Entrada
Array[0..n-1] vec

Variable
i: Indice

Inicio
Desde i ← (n – 1) / 2 **hasta** 0 **hacer**
Bajar (vect, i)
Fin

Hay que tener en cuenta que tanto en este caso (algoritmo *HacerHeap*), como en el algoritmo anterior (*Heapsort*) se está pasando al algoritmo *Bajar* un vector y no un *heap*, de manera que habría que modificar convenientemente el algoritmo *Bajar* para que funcionase de forma correcta.

En cuanto al coste del algoritmo de ordenación hay que tener en cuenta el coste de cada una de las dos partes del algoritmo, *Hacer_Heap* y la reorganización de 'n-1' montículos mediante el algoritmo *Bajar*. La formación del montículo tiene un coste lineal con el tamaño del array, *O(n)*, mientras que el coste del bucle que reorganiza los montículos de tamaño decreciente es *O(n*log n)*. Asintóticamente, la función logarítmica domina a la lineal y el coste final del algoritmo de ordenación será *O(n*log n)*. Por lo tanto, resulta que el método de ordenación heapsort es más eficiente que los métodos directos vistos en el tema dedicado a vectores.