ÁRBOLES AVL.

Los árboles binarios de búsqueda tal y como los hemos visto, adolecen del problema de que en el peor de los casos pueden tender parcialmente hacia el árbol degenerado, de manera que la búsqueda de un elemento cualquiera puede ser de un orden superior a O(lg n), y tender a O(n).

Este problema queda solucionado con los árboles AVL, o balanceados en altura. Denominados así en honor a sus autores (Adelson-Velskii y Landis, en una publicación soviética del año 1.962), estos árboles aseguran una serie de propiedades que permiten que la búsqueda de cualquier elemento quede acotada por una complejidad de orden O(lg n), con un coeficiente de aproximadamente 1'45. El orden de las operaciones de inserción y eliminación sigue siendo O(lg n). Por tanto, las aplicaciones de estos árboles son las mismas que las de un ABB, y además puede emplearse en sistemas en tiempo real en los que es necesario establecer una cota superior aceptable del tiempo que tardarán en ejecutarse ciertas operaciones.

Denominamos árbol AVL a aquél árbol binario de búsqueda que *o es vacío, o ambos hijos son también AVL y la diferencia entre sus alturas es menor o igual que 1*.

| Altura(Hijo_izq(a)) - Altura(Hijo_dch(a)) | # 1

Al valor | Altura(Hijo_izq(a)) - Altura(Hijo_dch(a)) | lo podemos llamar *factor de balance*.

La propiedad que debe cumplir un ArbolBB para ser AVL es la siguiente:

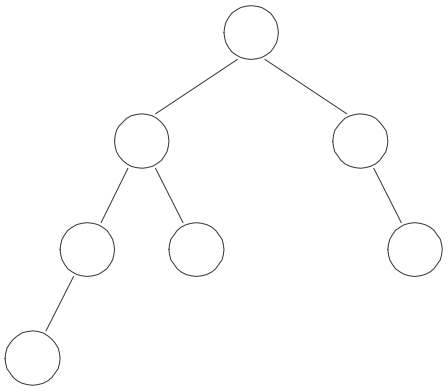
Es_AVL : ArbolBB ⟶ Lógico

ecuaciones r : elemento i, d : ArbolBB

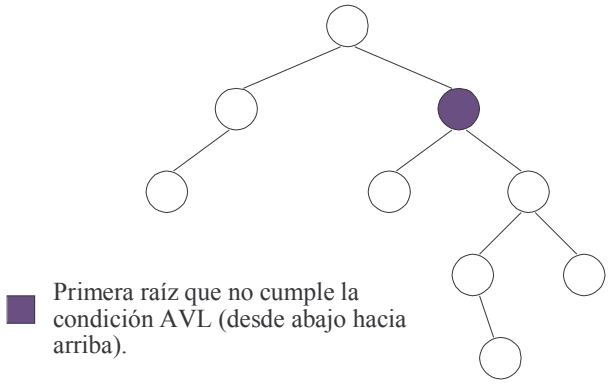
Es_AVL(Crear) == V

Es_AVL(Arbol_binario(r, i, d)) == Es_AVL(i) and
Es_AVL(d) and
(-1 # Altura(d) - Altura(i) # 1) and
(not Es_Vacio(i) 6 (r > Máximo(i))) and
(not Es_Vacio(d) 6 (r < Mínimo(d)))

El secreto de conseguir un factor de balance que se mantenga entre los límites establecidos, se encuentra tanto en el proceso de inserción como de eliminación. De fundamental importancia serán las rotaciones a derecha y a izquierda (que no tienen absolutamente nada que ver con sus homónimas en los anillos).



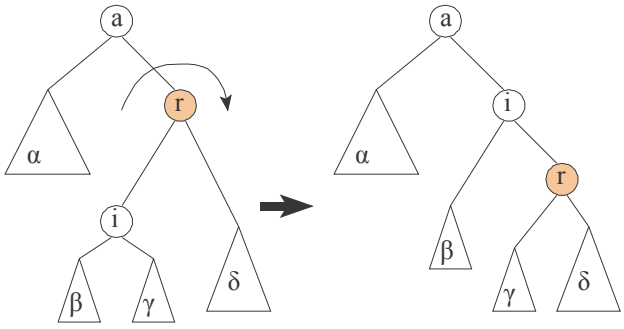
Árbol AVL.



Árbol no AVL.

Antes de pasar a especificar estas operaciones, supondremos que el tipo ArbolAVL posee las mismas operaciones básicas que el ArbolBB, o sea, Crear, Arbol_binario, Raiz, Hijo_izq e Hijo_dch, aunque, al igual que en el ArbolBB, la operación Arbol_binario estará oculta al usuario, que sólo podrá aumentar el tamaño de un árbol mediante la operación Insertar, cuya especificación veremos más adelante.

Una vez establecido el marco de trabajo, vamos a ver el significado gráfico de las operaciones Rotar_Izq y Rotar_Dch, de cuya existencia el usuario no tendrá conocimiento: serán auxiliares. Una de tales rotaciones se produce siempre sobre un nodo determinado; la figura ilustra el resultado de efectuar una rotación a la derecha sobre el nodo que se halla sombreado. Una rotación a la derecha sobre un nodo **r,** con hijos **i** y **d,** consiste en sustituir **r** por Raiz(**i**), como hijo izquierdo colocamos a Hijo_izq(**i**), y como hijo derecho construímos un nuevo árbol que tendrá a **r** como raíz, y a Hijo_dch(**i**) y a **d** como hijos izquierdo y derecho respectivamente.



Rotación derecha sobre el nodo r.

Un hecho muy importante de estas rotaciones, es que mantiene la ordenación del árbol, o sea, si el árbol original era ArbolBB, el resultado también lo seguirá siendo.

La especificación de estas operaciones es muy sencilla. Por supuesto, no se permite rotar al árbol vacío, ni tampoco rotaciones que requieran poner como raíz final a la raíz de un árbol vacío; en otras palabras:

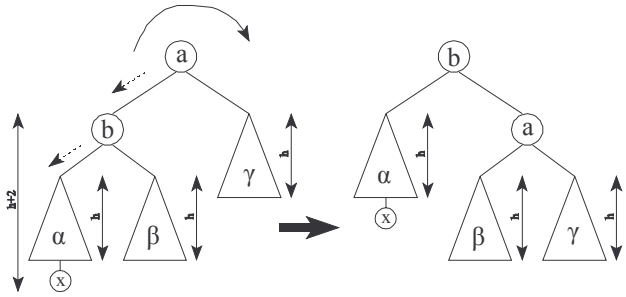
Rotar_Izq : ArbolAVL $\not\longrightarrow$ ArbolAVL
Rotar_Dch : ArbolAVL $\not\longrightarrow$ ArbolAVL
precondiciones a : ArbolAVL
Rotar_Izq(a) : (not Es_Vacío(a)) and (not Es_Vacío(Hijo_Dch(a))
Rotar_Dch(a) : (not Es_Vacío(a)) and (not Es_Vacío(Hijo_Izq(a))
ecuaciones r, r' : Elemento i, d, i', d' : ArbolAVL
Rotar_Izq(Arbol_binario(r, i, Arbol_binario(r', i', d'))) ==
Arbol_binario(r', Arbol_binario(r, i, i'), d')
Rotar_Dch(Arbol_binario(r, Arbol_binario(r', i', d'), d)) ==
Arbol_binario(r', i', Arbol_binario(r, d', d))

Supongamos ahora que se hace una inserción o una eliminación según el método tradicional del ArbolBB. Nuestra tarea posterior será analizar si se ha producido un desbalanceo, y si es así, entonces poner remedio al asunto para volver a alcanzarlo, de manera que el árbol resultante siga siendo binario de búsqueda, o sea, mediante rotaciones.

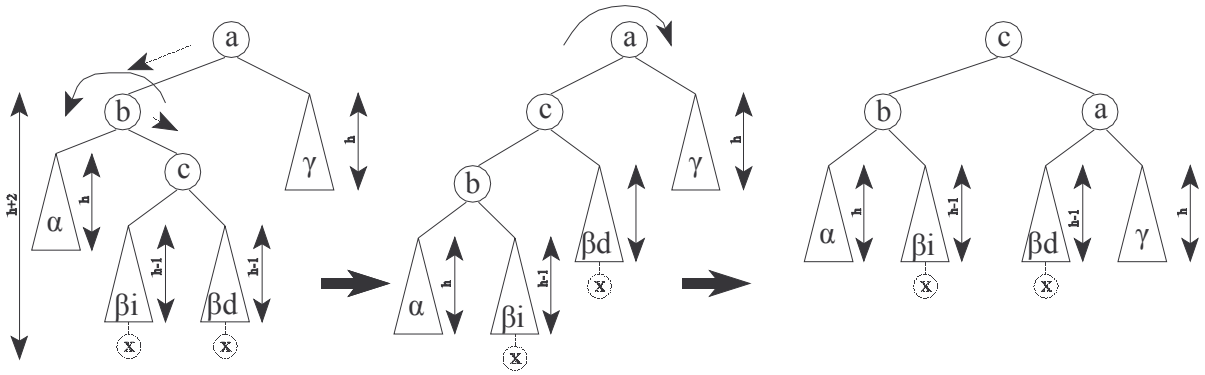
Tras efectuar una de estas operaciones pueden ocurrir diversos casos:

1) No se ha producido desbalanceo, es decir, a partir de todo nodo, la diferencia de alturas entre sus hijos es menor o igual que 1. En este caso no haremos nada. El árbol sigue siendo AVL.

2) Existe algún nodo que se ha desbalanceado. Una regla importante es actuar sobre el primer nodo que incumpla el factor de balance 1, empezando desde abajo hacia arriba, o sea, desde las hojas hacia la raíz. P.ej., en la figura de antes, actuaremos sobre el nodo sombreado, en lugar de sobre la raíz directamente.



Inserción Izq-Izq.



Inserción Izq-Dch.

Sin pérdida de generalidad, vamos a centrarnos en el caso en que sea el subárbol izquierdo el de mayor altura **h+2**, mientras que el derecho poseerá altura **h**. El caso contrario se resuelve de forma especular.

2.1) Como puede verse en la figura, el árbol de raíz b se halla compensado (hijos de alturas h+1 y h respectivamente), mientras que el a no. Observamos pues, que el nodo que produce la descompensación, **x**, se insertó por la rama izquierda de a. A su vez, x se ha insertado también a la izquierda de b. Estamos ante un caso Izquierda-Izquierda (II). Este caso se resuelve fácilmente efectuando una rotación a derecha sobre el nodo descompensado a, dando como resultado un árbol AVL.

2.2) Sin embargo, el método anterior no resulta suficiente cuando la inserción es Izquierda-Derecha, como puede comprobar por sí mismo el lector. En este caso, habremos de considerar también a la raíz del hijo derecho de b, o sea, a **c**. Nos dará igual si el nodo **x** se ha insertado a la

derecha o a la izquierda de c. En el dibujo se ha representado esto colocando x a ambos lados; el lector no debe confundirse, la inserción se ha hecho en cualquiera de los dos **pero no en ambos**; por tanto el factor de balanceo entre Hijo_izq(c) e Hijo_Dch(c) es exactamente 1. La solución consiste en dos pasos:

a) Efectuar una rotación a la izquierda sobre el nodo b.

b) Efectuar una rotación a la derecha sobre el nodo a.

Los casos Derecha-Derecha y Derecha-Izquierda se tratan de manera especular o simétrica.

Manualmente, para hacer la inserción, iremos marcando con flechitas el camino seguido por el nodo a insertar. Una vez incluído el nodo, seguiremos el camino inverso al recorrido (de abajo arriba), y en el momento en que nos encontremos un nodo desbalanceado le aplicaremos las rotaciones correspondientes según el caso. Así sucesivamente hasta llegar a la raíz.

El caso de la eliminación debe considerarse como inserción según el camino opuesto. Es lo mismo descompensar una balanza poniendo más peso en un lado que quitando peso del otro.

De esta forma, las operaciones Insertar y Eliminar, quedarían:

Insertar : Elemento × ArbolAVL ⟶ ArbolAVL

Eliminar : Elemento × ArbolAVL ⟶ ArbolAVL

ecuaciones e, r : Elemento i, d, i', d' : ArbolAVL

```
Insertar(e, Crear) == Arbol_binario(e, Crear, Crear)
Insertar(e, Arbol_binario(r, i, d)) ==
        SI e = r ENTONCES
                Arbol_binario(e, i, d)
        SI NO SI e > r ENTONCES
                SEA d' = Insertar(e, d) EN
                        SI Altura(d') - Altura(i) <= 1 ENTONCES
                                Arbol_binario(r, i, d')
                        SI NO SI e > Raiz(d) ENTONCES
                                Rotar_Izq(Arbol_binario(r, i, d'))
                        SI NO
                                Rotar_Izq(Arbol_binario(r, i, Rotar_Dch(d')))
        SI NO
                SEA i' = Insertar(e, i) EN
                        SI Altura(i') - Altura(d) <= 1 ENTONCES
                                Arbol_binario(r, i', d)
                        SI NO SI e < Raiz(i) ENTONCES
                                Rotar_Dch(Arbol_binario(r, i', d))
                        SI NO
                                Rotar_Dch(Arbol_binario(r, Rotar_Izq(i'), d))
Eliminar(e, Crear) == Crear
Eliminar(e, Arbol_binario(r, i, d)) ==
        SI e = r ENTONCES
                SI Es_Vacio(i) ENTONCES
```

```
                d
        SI NO SI Es_Vacio(d) ENTONCES
                i
        SI NO
                SEA d' = Eliminar(Mínimo(d), d) EN
                        SI Altura(i) - Altura(d') <= 1 ENTONCES
                                Arbol_binario(Mínimo(d), i, d')
                        SI NO SI Altura(Hijo_izq(i))$Altura(Hijo_Dch(i)) ENT__
                                Rotar_Dch(Arbol_binario(Mínimo(d), i, d'))
                        SI NO
                                Rotar_Dch(
                                        Arbol_binario(Mínimo(d),Rotar_Izq(i), d')
                                )
SI NO SI e < r ENTONCES
        SEA i' = Eliminar(e, i) EN
                SI Altura(d) - Altura(i') <= 1 ENTONCES
                        Arbol_binario(r, i', d)
                SI NO SI Altura(Hijo_dch(d)) >= Altura(Hijo_izq(d)) ENTONCES
                        Rotar_Izq(Arbol_binario(r, i', d))
                SI NO
                        Rotar_Izq(Arbol_binario(r, i', Rotar_Dch(d)))
SI NO
        SEA d' = Eliminar(e, d) END
                SI Altura(i) - Altura(d) <= 1 ENTONCES
                        Arbol_binario(r, i, d')
                SI NO SI Altura(Hijo_izq(i)) >= Altura(Hijo_dch(i)) ENTONCES
                        Rotar_Dch(Arbol_binario(r, i, d'))
                SI NO
                        Rotar_Dch(Arbol_binario(r, Rotar_Izq(i), d'))
```

Al igual que con los árboles binarios de búsqueda, las operaciones Insertar y Eliminar conservan la propiedad de ser AVL; en otras palabras, se cumple:

teoremas e : Elemento a : ArbolAVL

Es_AVL(a) 6 Es_AVL(Insertar(e, a))

Es_AVL(a) 6 Es_AVL(Eliminar(e, a))

Se demuestra que, por norma general, cuando el árbol tiene un tamaño decente, suele ser necesaria una sola rotación por cada 2 inserciones, y una sola rotación por cada 5 eliminaciones.

Como puede observarse, todas las operaciones están basadas en la altura de cada árbol. Si cada vez que hacemos una inserción tenemos que calcular y recalcular las alturas de los árboles, estaremos ante operaciones de complejidad bastante superior al orden O(lg n). Esto en la especificación no es ningún problema, ya que en ellas nuestro objetivo es tan sólo expresar el comportamiento de las operaciones. Sin embargo en las implementaciones supone un recargo de tiempo considerable. Para solucionar esto, podemos guardar junto con cada nodo, la altura a que

se encuentra, aunque para árboles muy grandes, esto puede ser un exceso innecesario de memoria. Otra solución más factible es utilizar un campo que nos diga únicamente si sus dos hijos tienen alturas iguales, o quien de los dos tiene mayor altura.

Implementación.

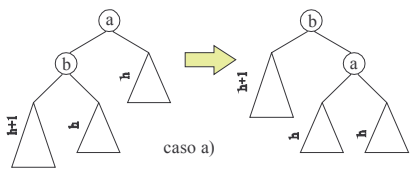
A continuación vamos a ver la implementación de los AVL, con punteros y sin cabecera. La definición es:

```
DEFINITION MODULE ArbolAVL;
TYPE
   AVL;
   ITEM = CARDINAL;
   COMPARAR = PROCEDURE(ITEM, ITEM) : INTEGER;
            (* -1  < *)
            (*  0  = *)
            (* +1  > *)
   ERROR = (SinError, SinMemoria, ArbolVacio);
VAR
   error : ERROR;
PROCEDURE Crear() : AVL;
PROCEDURE Es_Vacio(a : AVL) : BOOLEAN;
PROCEDURE Hijo_izq(a : AVL) : AVL;
PROCEDURE Hijo_dch(a : AVL) : AVL;
PROCEDURE Raiz(a : AVL) : ITEM;
PROCEDURE Insertar(e : ITEM; VAR a : AVL; comp : COMPARAR);
PROCEDURE Eliminar(e : ITEM; VAR a : AVL; comp : COMPARAR);
PROCEDURE Esta(e : ITEM; a : AVL; comp : COMPARAR) : BOOLEAN;
PROCEDURE Destruir(VAR a : AVL);
END ArbolAVL.
```

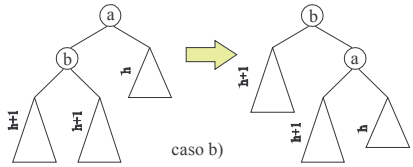
El módulo de implementación constituye un nodo de la forma ya conocida, pero además incluye un campo **balance** que indica cuál de sus dos hijos tiene mayor peso. El orden de la secuencia del tipo enumerado ESTADO, es muy importante, ya que la vamos a usar para hacer comparaciones, y facilitar el trabajo de examinar los cambios sufridos por un subárbol. Este tipo permite, como ya se dijo anteriormente, hacer una implementación con menores requerimientos de memoria.

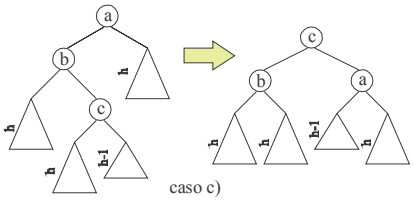
```
IMPLEMENTATION MODULE ArbolAVL;
FROM Storage IMPORT ALLOCATE, DEALLOCATE, Available;
FROM IO IMPORT WrStr;

TYPE
   PTR_NODO = POINTER TO NODO;
   AVL = PTR_NODO;
   (* El orden de los valores del ESTADO es muy importante. *)
   ESTADO = (Nulo, AmbosIgual, IzqMayor, DchMayor);
   NODO =  RECORD
            cont: ITEM;
            izq, dch : AVL;
            balance : [AmbosIgual..DchMayor];
```

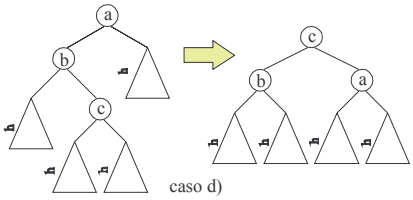
```
        END;

PROCEDURE Crear() : AVL;
BEGIN
   error := SinError;
   RETURN NIL;
END Crear;

PROCEDURE Es_Vacio(a : AVL) : BOOLEAN;
BEGIN
   error := SinError;
   RETURN (a = NIL);
END Es_Vacio;

PROCEDURE Hijo_izq(a : AVL) : AVL;
BEGIN
   error := SinError;
   IF Es_Vacio(a) THEN error := ArbolVacio; RETURN NIL; END;
   RETURN a^.izq;
END Hijo_izq;

PROCEDURE Hijo_dch(a : AVL) : AVL;
BEGIN
   error := SinError;
   IF Es_Vacio(a) THEN error := ArbolVacio; RETURN NIL; END;
   RETURN a^.dch;
END Hijo_dch;

PROCEDURE Raiz(a : AVL) : ITEM;
VAR
   basura : ITEM;
BEGIN
   error := SinError;
   IF Es_Vacio(a) THEN error := ArbolVacio; RETURN basura; END;
   RETURN a^.cont;
END Raiz;

PROCEDURE Estado(a : AVL) : ESTADO;
BEGIN
   IF Es_Vacio(a) THEN
      RETURN Nulo;
   ELSE
      RETURN a^.balance;
   END;
END Estado;

PROCEDURE Izq_Excesivo(VAR a : AVL);
VAR
   aux : PTR_NODO;
BEGIN
   (* Primero se actualizan los balances, y por último *)
   (* se efectúan las rotaciones.                      *)
   IF a^.izq^.balance = IzqMayor THEN  (* Inserción Izq-Izq. Caso
a) *)
      a^.balance := AmbosIgual;
```

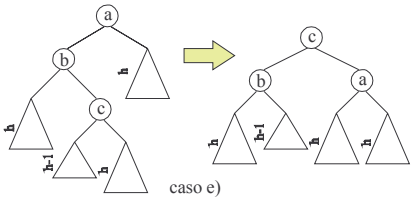


```
      a^.izq^.balance := AmbosIgual;
   ELSIF a^.izq^.balance = AmbosIgual THEN (* Caso de eliminación. Caso b) *)
      a^.balance := IzqMayor;
      a^.izq^.balance := DchMayor;
   ELSE (* Inserción Izq-Dch. *)
      IF a^.izq^.dch^.balance = IzqMayor THEN (* Caso c) *)
         a^.balance := DchMayor;
         a^.izq^.balance := AmbosIgual;
         a^.izq^.dch^.balance := AmbosIgual;
       ELSIF a^.izq^.dch^.balance = AmbosIgual THEN (* Caso d)
*)
         a^.balance := AmbosIgual;
         a^.izq^.balance := AmbosIgual;
         a^.izq^.dch^.balance := AmbosIgual;
      ELSE (* Caso e) *)
         a^.balance := AmbosIgual;
         a^.izq^.balance := IzqMayor;
         a^.izq^.dch^.balance := AmbosIgual;
      END;
      (* Rotación izqd. sobre a^.izq. *)
      aux := a^.izq;
      a^.izq := a^.izq^.dch;
      aux^.dch := a^.izq^.izq;
      a^.izq^.izq := aux;
   END;
   (* Rotación dcha. sobre a. *)
   aux := a;
   a := a^.izq;
   aux^.izq := a^.dch;
   a^.dch := aux;
END Izq_Excesivo;

PROCEDURE Dch_Excesivo(VAR a : AVL);
VAR
   aux : PTR_NODO;
BEGIN
   (* Primero se actualizan los balances, y por último *)
   (* se efectúan las rotaciones.                      *)
   IF a^.dch^.balance = DchMayor THEN  (* Inserción Dch-Dch. *)
      a^.balance := AmbosIgual;
      a^.dch^.balance := AmbosIgual;
   ELSIF a^.dch^.balance = AmbosIgual THEN (* Caso de eliminación. *)
      a^.balance := DchMayor;
      a^.dch^.balance := IzqMayor;
   ELSE (* Inserción Dch-Izq. *)
      IF a^.dch^.izq^.balance = DchMayor THEN
         a^.balance := IzqMayor;
         a^.dch^.balance := AmbosIgual;
         a^.dch^.izq^.balance := AmbosIgual;
      ELSIF a^.dch^.izq^.balance = AmbosIgual THEN
         a^.balance := AmbosIgual;
         a^.dch^.balance := AmbosIgual;
         a^.dch^.izq^.balance := AmbosIgual;
      ELSE
         a^.balance := AmbosIgual;
```









```
            a^.dch^.balance := DchMayor;
            a^.dch^.izq^.balance := AmbosIgual;
        END;
        (* Rotación dcha. sobre a^.dch. *)
        aux := a^.dch;
        a^.dch := a^.dch^.izq;
        aux^.izq := a^.dch^.dch;
        a^.dch^.dch := aux;
    END;
    (* Rotación izqd. sobre a. *)
    aux := a;
    a := a^.dch;
    aux^.dch := a^.izq;
    a^.izq := aux;
END Dch_Excesivo;

PROCEDURE Minimo(a : AVL) : ITEM;
BEGIN
    WHILE a^.izq <> NIL DO
        (* a no se destruye porque se pasa por valor. *)
        a := a^.izq;
    END;
    RETURN a^.cont;
END Minimo;

PROCEDURE Insertar(e : ITEM; VAR a : AVL; comp : COMPARAR);
VAR
    estado : ESTADO;
BEGIN
    error := SinError;
    IF Es_Vacio(a) THEN
        IF NOT Available(SIZE(NODO)) THEN error := SinMemoria; RETURN; END;
        NEW(a);
        a^.cont := e;
        a^.izq := NIL;
        a^.dch := NIL;
        a^.balance := AmbosIgual;
    ELSIF comp(e, a^.cont) = 0 THEN (* e = a^.cont *)
        a^.cont := e;
    ELSIF comp(e, a^.cont) = -1 THEN (* e < a^.cont *)
        estado := Estado(a^.izq);
        Insertar(e, a^.izq, comp);
        IF Estado(a^.izq) > estado THEN
            IF a^.balance = AmbosIgual THEN
                a^.balance := IzqMayor;
            ELSIF a^.balance = DchMayor THEN
                a^.balance := AmbosIgual;
            ELSE
                Izq_Excesivo(a);
            END;
        END;
    ELSE                            (* e > a^.cont *)
        estado := Estado(a^.dch);
        Insertar(e, a^.dch, comp);
        IF Estado(a^.dch) > estado THEN
```

```
            IF a^.balance = AmbosIgual THEN
               a^.balance := DchMayor;
            ELSIF a^.balance = IzqMayor THEN
               a^.balance := AmbosIgual;
            ELSE
               Dch_Excesivo(a);
            END
         END;
      END;
   END;
END Insertar;

PROCEDURE Eliminar(e : ITEM; VAR a : AVL; comp : COMPARAR);
VAR
   estado : ESTADO;
   minimo : ITEM;
   aux : PTR_NODO;
   hacer_derecho : BOOLEAN;
BEGIN
   error := SinError;
   IF Es_Vacio(a) THEN RETURN; END;
   hacer_derecho := FALSE;
   IF comp(e, a^.cont) = 0 THEN (* e = a^.cont *)
      IF a^.izq = NIL THEN
         aux := a;
         a := a^.dch;
         DISPOSE(aux);
      ELSIF a^.dch = NIL THEN
         aux := a;
         a := a^.izq;
         DISPOSE(aux);
      ELSE
         minimo := Minimo(a^.dch);
         a^.cont := minimo;
         estado := Estado(a^.dch);
         Eliminar(minimo, a^.dch, comp);
         hacer_derecho := TRUE;
      END;
   ELSIF comp(e, a^.cont) = -1 THEN (* e < a^.cont *)
      estado := Estado(a^.izq);
      Eliminar(e, a^.izq, comp);
      IF (Estado(a^.izq) < estado) AND (Estado(a^.izq) < IzqMayor) THEN
      (* Si se ha efectuado una transición a AmbosIgual o a Nulo. *)
         IF a^.balance = AmbosIgual THEN
            a^.balance := DchMayor;
         ELSIF a^.balance = IzqMayor THEN
            a^.balance := AmbosIgual;
         ELSE
            Dch_Excesivo(a);
         END;
      END;
   ELSE                          (* e > a^.cont *)
      estado := Estado(a^.dch);
      Eliminar(e, a^.dch, comp);
      hacer_derecho := TRUE;
   END;
```

```
   IF hacer_derecho THEN
      IF (Estado(a^.dch) < estado) AND (Estado(a^.dch) < IzqMayor) THEN
      (* Si se ha efectuado una transición a AmbosIgual o a Nulo. *)
         IF a^.balance = AmbosIgual THEN
            a^.balance := IzqMayor;
         ELSIF a^.balance = DchMayor THEN
            a^.balance := AmbosIgual;
         ELSE
            Izq_Excesivo(a);
         END
      END;
   END;
END Eliminar;

PROCEDURE Esta(e : ITEM; a : AVL; comp : COMPARAR) : BOOLEAN;
BEGIN
   error := SinError;
   WHILE (a <> NIL) AND (comp(e, a^.cont) <> 0) DO
      IF comp(e, a^.cont) = -1 THEN  (* e < a^.cont *)
         a := a^.izq;
      ELSE
         a := a^.dch;
      END;
   END;
   RETURN (a <> NIL);
END Esta;

PROCEDURE Destruir(VAR a : AVL);
BEGIN
   error := SinError;
   IF NOT Es_Vacio(a) THEN
      Destruir(a^.izq);
      Destruir(a^.dch);
      DISPOSE(a);
   END;
END Destruir;

BEGIN
   error := SinError;
END ArbolAVL.
```

Las operaciones Izq_Excesivo y Dcho_Excesivo, simplifican la labor de efectuar las rotaciones y actualizar los balances de los nodos. Ambas operaciones se han construído de forma que sirvan tanto después de efectuar una inserción como una eliminación, ya que el objetivo es uniformizar el tratamiento. En estas operaciones, se estudia primero el caso en que nos encontramos, y una vez localizado, se cambian los balances de los nodos involucrados (lo que hace que temporalmente el árbol quede inconsistente), y por último se efectúan las rotaciones necesarias. Como el lector puede observar, hay ciertas partes que se pueden compactar reduciendo el código, aunque a costa de disminuir la legibilidad.

Como operaciones ocultas se tiene también a Minimo, que devuelve el nodo de menor valor de un árbol, y a Estado, que devuelve el estado en que se encuentra el árbol en cuestión (su raíz), o Nulo si el árbol es vacío.

Por otro lado, se ha sido fiel a la especificación en el sentido de que no se permiten elementos repetidos. Aunque pueda parecer absurdo el trozo de código:

```
   ELSIF comp(e, a^.cont) = 0 THEN (* e = a^.cont *)
      a^.cont := e;
```

en la operación Insertar, debe tenerse en cuenta que un ITEM puede ser un registro formado por una clave y una información asociada; así, dos ítemes con la misma clave y diferente información asociada darían positivo al ser sometidos a la operación de igualdad (comp de tipo COMPARAR), a pesar de no coincidir en la información adicional. Por tanto, este trozo de código sustituye la información asociada antigua por la nueva.

Como puede verse, la operación Está hace uso del hecho de que un AVL es también un ABB, permitiendo que la búsqueda sea de O(lg n).

Por último decir que el tratamiento de errores es muy simple.

A continuación se tiene el siguiente driver de pruebas, sumamente simplificado:

```
MODULE DRIVER;
FROM ArbolAVL IMPORT AVL, COMPARAR, ERROR, error,
             Crear, Insertar, Eliminar, Destruir, Es_Vacio, Raiz,
             Hijo_izq, Hijo_dch;
FROM IO IMPORT WrStr, WrCard, WrChar, WrLn, RdKey, RdCard;
VAR
  a : AVL;
  n : CARDINAL;
  k : CHAR;

PROCEDURE VerNiveles(a : AVL);
   PROCEDURE VerNivel(n : CARDINAL; a : AVL) : CARDINAL;
   BEGIN
      IF Es_Vacio(a) THEN
         RETURN 0
      ELSIF n = 1 THEN
         WrCard(Raiz(a), 6); WrStr(", ");
         RETURN 1;
      ELSE
         RETURN VerNivel(n-1, Hijo_izq(a)) + VerNivel(n-1, Hijo_dch(a));
      END;
   END VerNivel;
VAR
   i : CARDINAL;
   k : CHAR;
BEGIN
   i := 0;
   REPEAT
      INC(i);
      WrLn;
   UNTIL VerNivel(i, a) = 0;
   k := RdKey();
END VerNiveles;

PROCEDURE MiComparar(a, b : CARDINAL) : INTEGER;
BEGIN
   IF a = b THEN RETURN 0;
   ELSIF a < b THEN RETURN -1;
```

```
   ELSE RETURN +1;
   END;
END MiComparar;

BEGIN
   a := Crear();
   LOOP
      WrStr("0.- Acabar.");
      WrStr("1.- Insertar.");
      WrStr("2.- Eliminar.");
      WrLn;
      WrStr("Opción. "); k := RdKey();
      IF k = '0' THEN EXIT; END;
      WrStr("Número: "); n := RdCard();
      IF k = '1' THEN
         Insertar(n, a, MiComparar);
      ELSE
         Eliminar(n, a, MiComparar);
      END;
      IF error <> SinError THEN
         WrStr("La cosa no va bien."); WrLn;
         WrStr("¿(C)ancelar? "); k := RdKey(); WrLn;
         IF (k = 'c') OR (k = 'C') THEN EXIT END;
      END;
      VerNiveles(a);
   END;
   Destruir(a);
END DRIVER.
```

El tratamiento de errores de este driver es sumamente escueto. Sólo se han incluído dos operaciones a testear en el menú, ya que las demás operaciones siguen el mismo tratamiento que en el ABB, ya comprobado en capítulos anteriores.

En este driver se ha incluído un procedimiento iterativo-recursivo para recorrer el árbol por niveles. Para comprobaciones y depuración es muy interesante hacer público el tipo ESTADO, y el procedimiento interno Estado, de forma que en VerNiveles se puede incluir justo tras:

```
WrCard(Raiz(a), 6);
```

el código:

```
CASE Estado(a) OF
  | Nulo          : WrStr(" -, ");
  | AmbosIgual    : WrStr(" =, ");
  | IzqMayor      : WrStr(" >, ");
  | DchMayor      : WrStr(" <, ");
END;
```

de manera que se puede hacer un seguimiento más exhaustivo en tiempo de ejecución. El lector que lo desee puede incluso modificar más esta operación de visualización para tener aún más claro de quien es hijo cada nodo visualizado (esto puede ser útil si la implementación no se comporta siquiera como un ABB).

A partir de esta implementación, el lector también puede modificarla almacenando en cada nodo la altura en lugar de un valor de tipo ESTADO.

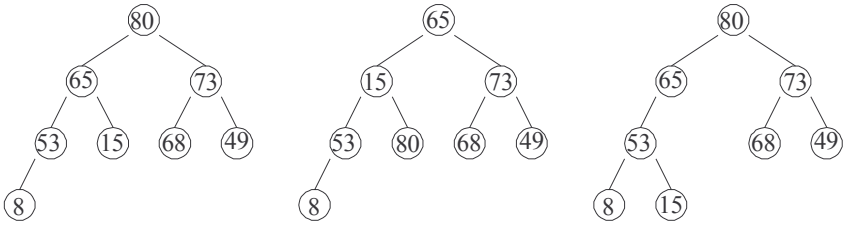
HEAPS.

Un heap o montículo es un árbol binario completo, y además parcialmente ordenado.

Ya hemos visto el concepto de árbol completo. Un árbol parcialmente ordenado es aquél que tiene todas y cada una de sus ramas, consideradas como listas, totalmente ordenadas, ya sea de forma creciente o decreciente. En relación al heap, se exige una ordenación decreciente, lo que puede dar lugar a una definición más natural de heap: un heap es un árbol binario completo, tal que el valor de su raíz es mayor o igual que las raices de sus hijos, siendo también heaps ambos hijos.

A este concepto le añadiremos ciertas operaciones, lo cual desemboca en el TAD Heap.

La siguiente figura ilustra un ejemplo de heap, así como casos de árboles que no lo son: por no ser parcialmente ordenados, o por no ser completos.



Árbol heap.

No es heap, dado que no es parcialmente ordenado

No es heap, dado que no es completo.

Al igual que para el árbol binario de búsqueda, las operaciones básicas son las mismas que para el árbol binario simple, y al igual que en aquél, el generador Arbol_binario está oculto al usuario.

Necesitamos además un predicado que nos diga cuando un árbol es un heap:

Es_Heap: ArbolB ⟶ Lógico

ecuaciones a : ArbolB

Es_Heap(a) == Es_Completo(a) and Es_Parc_Orden(a)

Ya hemos especificado formalmente cuando un árbol es completo. Nuestra definición de parcialmente ordenado se completa con la siguiente especificación:

Es_Parc_Orden : ArbolB ⟶ Lógico

ecuaciones r : Elemento i, d : ArbolB

Es_Parc_Orden(Crear) == V

Es_Parc_Orden(Arbol_binario(r, i, d)) == (Es_Vacio(i) or r >= Raiz(i)) and
(Es_Vacio(d) or r >= Raiz(d)) and
Es_Parc_Orden(i) and
Es_Parc_Orden(d)

Vamos a ver ahora unas cuantas operaciones interesantes. La primera de ellas inserta un elemento en un árbol completo, y hace que el resultado siga siendo un árbol completo:

Ins_Completo: Elemento × ArbolB ⟶ ArbolB
ecuaciones e, r : Elemento i, d : ArbolB
Ins_Completo(e, Crear) == Arbol_binario(e, Crear, Crear)
Ins_Completo(e, Arbol_binario(r, i, d)) ==
SI ((Altura(i) > Altura(d)) and (not Es_Lleno(i))) or
((Altura(i) = Altura(d)) and Es_Lleno(d)) ENTONCES
Arbol_binario(r, Ins_Completo(e, i), d)
SI NO
Arbol_binario(r, i, Ins_Completo(e, d))

de manera que se verifica:
teorema e : Elemento a : ArbolB
Es_Completo(a) 6 Es_Completo(Insertar(e, a))

A partir de esta operación, podemos convertir cualquier árbol en completo, sin más que recorrer todos sus nodos, e ir insertándolos en otro árbol que se inicializará a vacío; para ello usaremos una operación auxiliar:

A_Completo: ArbolB ⟶ ArbolB
A_Completo_aux : ArbolB × ArbolB ⟶ ArbolB
ecuaciones r : Elemento a, b, i, d : ArbolB
A_Completo(a) == A_Completo_aux(a, Crear)
A_Completo_aux(Crear, b) == b
A_Completo_aux(Arbol_binario(r, i, d), b) ==
Ins_Completo(r, A_Completo_aux(i, A_Completo_aux(d, b)))

de manera que se verifica:
teorema a : ArbolB
Es_Completo(A_Completo(a)) == V

Una vez que hemos convertido cualquier árbol en completo, ya sólo nos queda convertirlo en heap, que es lo que hace la operación Completo_A_Heap, que parte del árbol completo, y aplica recursivamente Arbol_heap a cada raíz, y a sus dos hijos ya convertidos es heap:

Completo_A_Heap: ArbolB ⟶/ ArbolB
Arbol_heap: Elemento × ArbolB × ArbolB ⟶ ArbolB
precondiciones a:ArbolB
Completo_A_Heap(a) : Es_Completo(a)
ecuaciones r : Elemento i, d : ArbolB
Completo_A_Heap(Crear) == Crear
Completo_A_Heap(Arbol_binario(r, i, d)) ==
Arbol_heap(r, Completo_A_Heap(i), Completo_A_Heap(d))

```
        Arbol_heap(r, i, d) ==
                SI Es_Vacio(i)
                        Arbol_binario(r, Crear, Crear)
                SI NO SI Es_Vacio(d) ENTONCES
                        SI r > Raiz(i) ENTONCES
                                Arbol_binario(r, i, Crear)
                        SI NO
                                Arbol_binario(Raiz(i), Arbol_binario(r, Crear, Crear), Crear)
                SI NO SI (r $ Raiz(i)) and (r $ Raiz(d)) ENTONCES
                        Arbol_binario(r, i, d)
                SI NO SI (Raiz(i) $ r) and (Raiz(i) $ Raiz(d)) ENTONCES
                        Arbol_binario(Raiz(i), Arbol_heap(r, Hijo_izq(i), Hijo_dch(i)), d)
                SI NO
                        Arbol_binario(Raiz(d), i, Arbol_heap(r, Hijo_izq(d), Hijo_dch(d)))
teoremas        a, i, d : ArbolB                r : Elemento
        Completo(a) 6 Es_Heap(Completo_A_Heap(a))
        Es_Completo(Arbol_binario(r, i, d)) and
        Es_Heap(i) and Es_Heap(d) 6 Es_Heap(Arbol_heap(r, i, d))
```

No obstante, podemos simplificar estas tareas hasta cierto punto, con una sola operación que inserte un elemento en un heap, de manera similar a como lo hace el Insertar de los árboles binarios de búsqueda. Ello da lugar a la operación:

```
        Insertar: Elemento × Heap ⟶ Heap
ecuaciones      e, r : Elemento         i, d : Heap
        Insertar(e, Crear) == Arbol_binario(e, Crear, Crear)
        Insertar(e, Arbol_binario(r, i, d)) ==
                SI e > r ENTONCES
                        SI Es_Vacio(i) ENTONCES
                                Arbol_binario(e, Arbol_binario(r, Crear, Crear), Crear)
                        SI NO SI Es_Vacio(d) ENTONCES
                                Arbol_binario(e, i, Arbol_binario(r, Crear, Crear))
                        SI NO SI        ((Altura(i) > Altura(d)) and (not Es_Lleno(i))) or
                                        ((Altura(i) = Altura(d)) and Es_Lleno(d)) ENTONCES
                                Arbol_binario(e, Insertar(r, i), d)
                        SI NO
                                Arbol_binario(e, i, Insertar(r, d))
                SI NO
                        SI Es_Vacio(i) ENTONCES
                                Arbol_binario(r, Arbol_binario(e, Crear, Crear), Crear)
                        SI NO SI Es_Vacio(d) ENTONCES
                                Arbol_binario(r, i, Arbol_binario(e, Crear, Crear))
                        SI NO SI        ((Altura(i) > Altura(d)) and (not Es_Lleno(i))) or
                                        ((Altura(i) = Altura(d)) and Es_Lleno(d)) ENTONCES
                                Arbol_binario(r, Insertar(e, i), d)
                        SI NO
```

Arbol_binario(r, i, Insertar(e, d))

La especificación anterior queda más compacta de la forma:

```
        Insertar: Elemento × Heap ⟶ Heap
auxiliares
        min, max : Elemento × Elemento ⟶ Elemento
ecuaciones   e, r : Elemento        i, d : Heap
        min(a, b) ==  SI a # b ENTONCES a
                      SI NO                    b
        max(a, b) ==  SI a > b ENTONCES        a
                      SI NO                    b
        Insertar(e, Crear) == Arbol_binario(e, Crear, Crear)
        Insertar(e, Arbol_binario(r, i, d) ==
                SI    ((Altura(i) > Altura(d)) and (not Es_Lleno(i))) or
                      ((Altura(i) = Altura(d)) and Es_Lleno(d)) ENTONCES
                      Arbol_binario(max(r, e), Insertar(min(r, e), i), d)
                SI NO
                      Arbol_binario(max(r, e), i, Insertar(min(r, e), d))
```

lo cual puede dar una idea de lo importantes que pueden ser las operaciones auxiliares a la hora de esclarecer una especificación, y que por ello el diseñador no debe escatimar su uso.

Aplicaciones e implementaciones.

Colas de prioridad.

El árbol heap puede emplearse para facilitar la extracción de elementos en las colas de prioridad, ya que el elemento de mayor prioridad está siempre en la raíz. Aunque para ello también podemos emplear un árbol binario de búsqueda, en éste no se nos asegura que el árbol será de altura mínima, característica que poseen los heap, lo cual asegura que los algoritmos de extracción e inserción son realmente de $O(lg_{2)}n)$, y no tan sólo una aproximación como ocurre en los binarios de búsqueda simples.

Otra ventaja del heap a este respecto, es que puede almacenarse en un array mediante la estructura dispersa en amplitud, dando lugar a una tabla compacta ya que es un árbol completo, cosa que no ocurre normalmente en un binario de búsqueda. Como contrapartida, la eliminación en el heap, a pesar de ser también de orden $O(lg_{2)}n)$, tiene una constante de proporcionalidad superior, ya que es necesario profundizar en el árbol dos veces: 1) para tomar el elemento más a la derecha del último nivel, y 2) para insertar adecuadamente dicho nodo en el árbol.

Ordenación *heapsort*.

En su momento vimos un método para almacenar árboles mediante estructuras estáticas compactas, que se adecuaba especialmente al almacenamiento de árboles completos, almacenando

los hijos de una raíz situada en la posición **i** a partir de las posiciones **2i** y **2i+1** respectivamente. Resulta evidente que dicho método es el que mejor viene para almacenar árboles heap.

Una de las utilidades fundamentales de los heap, es que permiten ordenar una secuencia de elementos en un tiempo de orden $O(n \lg_{2)} n)$ en el peor de los casos, y con una complejidad espacial de O(n) con coeficiente 1, al igual que ocurre con el método Quicksort. Lo primero que hay que hacer es construir la estructura de heap en la tabla dada. Para ello basta efectuar la operación Arbol_heap con los n/2 elementos del árbol que tienen hijos (los otros n/2 son hojas, y por tanto ya son heaps), lo cual se lleva un tiempo menor que $O(n \lg_{2)} n)$.

Una vez conseguida la estructura de heap almacenada en una tabla, se pretende convertirla en una lista ordenada de menor a mayor. Para ello operaremos como sigue, suponiendo que la tabla tiene un tamaño inicial n=N.

1) Si n = 1 ya hemos acabado.

2) Por ser heap, el elemento de la posición 1 será el mayor de todos. Intercambiamos el elemento de la posición 1 con el de la posición n. A continuación hacemos n := n - 1, ya que el elemento recién colocado ya está ordenado.

3) Suponiendo que las posiciones 1..n de la tabla restante es un árbol, lo sometemos a Arbol_heap, para convertirlo en heap (nótese que se verifican las precondiciones necesarias dadas en el teorema para conseguir esto con una sola aplicación de Arbol_heap). Esta operación puede requerir un máximo de $O(\lg_{2)} n)$ iteraciones.

4) Volvemos al paso 1.

Dado que esta secuencia de pasos se efectúa una vez para cada elemento (n), y que cada paso consume un máximo de $O(\lg_{2)} n)$, la complejidad total del algoritmo es $O(n \lg_{2)} n)$.

Este algoritmo tiene la ventaja sobre el quicksort de que es completamente iterativo, y además tiene el mismo orden de complejidad.

Podemos especificar este algoritmo de ordenación mediante las siguientes operaciones (para simplificar, la lista sale ordenada de mayor a menor):

Ordenar : Heap ⟶ Lista

auxiliares

Eliminar_Ultimo : Heap ⟶/ Heap

Ultimo : Heap ⟶/ Elemento

precondiciones a : Heap

Eliminar_Ultimo(a) : not Es_Vacio(a)

Ultimo(a) : not Es_Vacio(a)

ecuaciones r : Elemento a, i, d : Heap

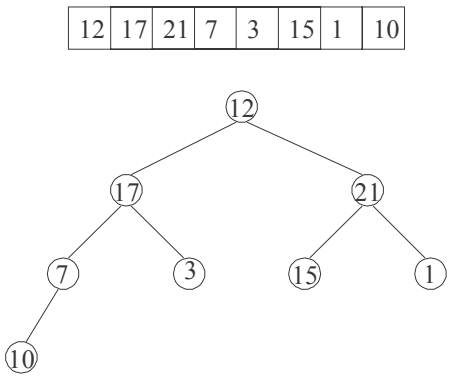
```
Eliminar_Ultimo(Arbol_binario(r, i, d)) ==
        SI Es_Vacio(i) ENTONCES
                Crear
        SI NO SI Es_Vacio(d) ENTONCES
                Arbol_binario(r, Crear, Crear)
        SI NO SI Altura(i) > Altura(d) ENTONCES
                Arbol_binario(r, Eliminar_Ultimo(i), d)
        SI NO
```

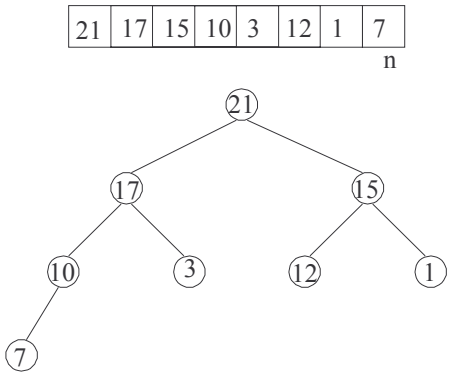
```
                Arbol_binario(r, i, Eliminar_Ultimo(d))
Ultimo(Arbol_binario(r, i, d)) ==
        SI Es_Vacio(i) ENTONCES
                r
        SI NO SI Es_Vacio(d) ENTONCES
                Raiz(i)
        SI NO SI Altura(i) > Altura(d) ENTONCES
                Ultimo(i)
        SI NO
                Ultimo(d)
Ordenar(a) ==
        SI Es_Vacio(a) ENTONCES
                Crear
        SI NO SI Es_Vacio(i) and Es_Vacio(d) ENTONCES
                Construir(r, Crear)
        SI NO
                SEA a' = Eliminar_Ultimo(a)
                EN
                Construir(Raiz(a),
                                Ordenar(
                                        Arbol_heap(
                                                Ultimo(a),
                                                Hijo_izq(a'),
                                                Hijo_dch(a')
                                        )
                                )
                )
```

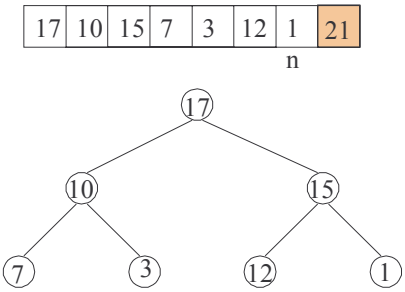
Las siguientes figuras ilustran el comienzo del proceso de ordenación de una lista de 7 elementos. En cada figura aparece el array en el que se encuentran los elementos, y el árbol a que corresponde (siguiendo una estructura de almacenamiento dispersa en amplitud, que se convierte en compacta para árboles completos, como es nuestro caso).

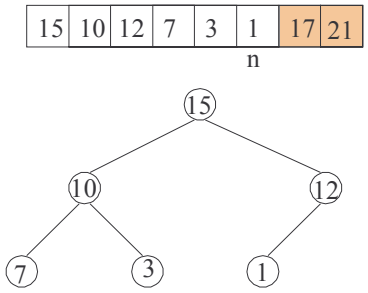
Inicialmente el árbol binario hay que convertirlo en heap. Ello se puede hacer fácilmente partiendo de la base de que la mitad de los elementos ya son heap (10, 3, 15, y 1, forman árboles de un solo nodo), y sólo tenemos que preocuparnos de los demás, haciendo con cada uno de ellos la operación Arbol_heap vista anteriormente y optimizada para el caso en que los dos subárboles ya son heaps también.

Una vez hecho esto, basta con aplicar el algoritmo descrito más arriba, e iremos obteniendo la secuencia indicada.



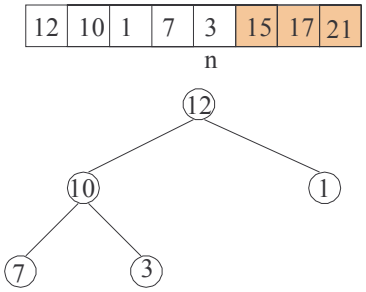






A continuación veremos la implementación del algoritmo de ordenación por el método del heapsort. Se ordena ascendentemente una lista de MAXELEM elementos.

Para ver un caso de ejemplo, se ha incluído el algoritmo y los procesos de carga y visualización del resultado en un sólo módulo.




```
MODULE HEAPSORT;
FROM IO IMPORT WrCard, WrLn;
FROM Lib IMPORT RAND;
CONST
   MAXLISTA = 20;
TYPE
   INDICE = [1..MAXLISTA];
   TABLA = ARRAY INDICE OF CARDINAL;
VAR
   N : INDICE;
   l : TABLA;
   i : INDICE;
```

```
   (* Proc auxiliar de intercambio de elementos. *)
   PROCEDURE SWAP(VAR a : CARDINAL; VAR b : CARDINAL);
   VAR
         c : CARDINAL;
   BEGIN
         c := a;
         a := b;
         b := c;
   END SWAP;

PROCEDURE Arbol_heap(VAR l : TABLA; i, n : INDICE);
BEGIN
      IF (i*2 > n) THEN
            RETURN
      ELSIF (i*2 = n) THEN
            IF l[i] < l[i*2] THEN
               SWAP(l[i], l[i*2]);
            END;
      ELSIF (l[i*2] >= l[i]) AND (l[i*2] >= l[i*2+1]) THEN
            SWAP(l[i], l[i*2]);
            Arbol_heap(l, 2*i, n);
      ELSIF (l[i*2+1] >= l[i]) AND (l[i*2+1] >= l[i*2]) THEN
            SWAP(l[i], l[i*2+1]);
            Arbol_heap(l, 2*i+1, n);
      END;
END Arbol_heap;


BEGIN
  N := MAX(INDICE);
  (* Carga y visualización de la tabla. *)
  FOR i := MIN(INDICE) TO N DO
     l[i] := TRUNC(RAND()*1000.0);
     WrCard(l[i], 6);
  END;
  WrLn;
  (* Conversión a heap. *)
  FOR i := N DIV 2 TO 1 BY -1 DO
     Arbol_heap(l, i, N);
  END;
  (* Ahora comienza el bucle de extracción de elementos ordenados. *)
  FOR i := N TO 2 BY -1 DO
     SWAP(l[1], l[i]);
     Arbol_heap(l, 1, i-1);
  END;
  (* Visualización del resultado. *)
  FOR i := MIN(INDICE) TO N DO
     WrCard(l[i], 6);
  END;
END HEAPSORT.
```

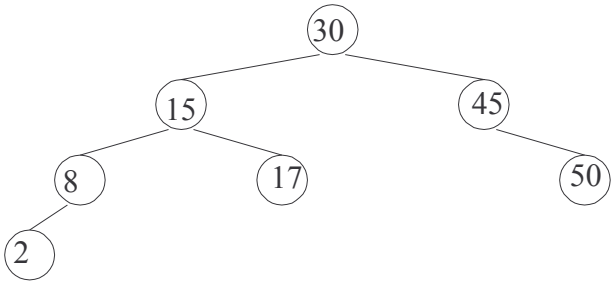
Ejercicios.

1.- Especificar una operación que convierta un ABB en un AVL, de la forma más simple posible.

2.- Especificar una operación que convierta un ABB en un AVL que cumpla la condición de ser completo.

3.- Sea el árbol AVL siguiente. Efectuar en él la siguiente secuencia de elementos: 60, 10 y 11.

Eliminar asimismo el valor 30, considerando que si se elimina un nodo con sus dos hijos, su valor deberá ser sustituído por el menor de su hijo derecho.



4.- Especificar una operación tal que dados dos heaps cualesquiera, los fusione en uno solo.

5.- Especificar una operación de Eliminación sobre heaps, que permita eliminar cualquier elemento, y no sólo el que se halla en la raíz.