Árboles – Definiciones

- **Arbol**: Un arbol consiste en un nodo (**r**, llamado nodo **raiz**) y una lista o conjunto de subárboles ($\mathbf{A}_1$, $\mathbf{A}_2$, ... , $\mathbf{A}_k$). Si el orden de los subárboles importa, entonces se representan como una lista, y se denomina **árbol ordenado**. En caso contrario se representan como una colección y se denomina **arbol no ordenado**.

- Se definen como **nodos hijos** de **r** a los los nodos raices de los subárboles $\mathbf{A}_1$, $\mathbf{A}_2$, ... , $\mathbf{A}_k$

- Si **b** es un **nodo hijo** de **a** entonces **a** es el **nodo padre** de **b**.

- Un nodo puede tener cero o más hijos, y uno o ningún padre. Si no tiene nodo padre entonces es el **nodo raiz** del árbol.

- Un nodo sin hijos se denomina **nodo hoja**.

- Se define un **camino** en un arbol como cualquier secuencia de nodos del arbol, $\mathbf{n_1}$ ... $\mathbf{n_p}$, que cumpla que cada nodo es padre del siguiente en la secuencia (es decir, que $\mathbf{n_i}$ es el padre de $\mathbf{n_{i+1}}$). La **longitud** del camino se define como el número de nodos de la secuencia menos uno (**p-1**).

- Los **descendientes** de un nodo son aquellos nodos accesibles por un camino que comienze en el nodo. Los **ascendientes** de un nodo son los nodos del camino que va desde la raiz a él.

- La **altura de un nodo** en un arbol se define como la longitud del camino más largo que comienza en el nodo y termina en una hoja. La altura de un nodo hoja será de cero, y la altura de un nodo se puede calcular sumando uno a la mayor altura de sus hijos.

- La **altura de un árbol** se define como la altura de su raiz.

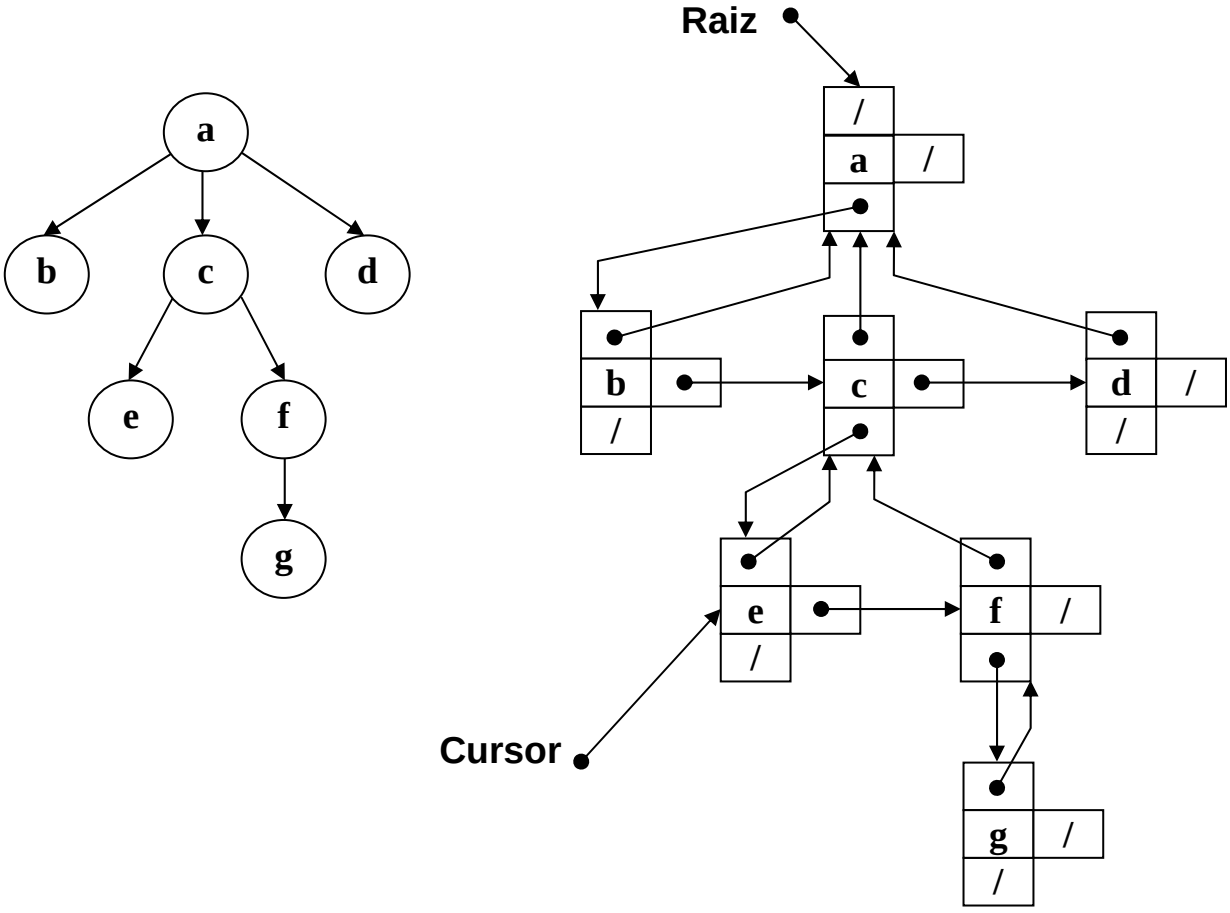
- La **profundidad de un nodo** se define como la longitud del camino (único) que comienza en la raiz y termina en el nodo. La profundidad de la raiz es cero, y la profundidad de un nodo se puede calcular como la profundidad de su padre mas uno. A la profundidad de un nodo también se la denomina **nivel** del nodo en el árbol.

Representación de Árboles

- Salvo casos particulares, un árbol se almacena mediante nodos con referencias al nodo padre y a la lista de nodos hijos, o bien con referencias al nodo padre, al primer hijo y al nodo hermano. Se suele utilizar un acceso basado en cursor.

- Las operaciones principales son el acceso al nodo raiz, la inserción y borrado de subárboles hijos del nodo actual y el cambio del cursor (del nodo actual a su padre y del nodo actual a uno de sus hijos).

- Ejemplo (representación padre-primer hijo-hermano):



Ejemplo de Implementación de un Arbol

```
indexing
    descripcion : "Nodo de un arbol con representación"
        "padre - primer hijo - hermano"
class NODOARB[ELEM]

creation make
feature { ANY }
    elem : ELEM
    padre, hijo, hermano : like Current

    make(e: like elem) is
    do
      elem := e;
      padre := Void; hijo := Void; hermano := Void
    end -- make

    cambia_padre(nodo: like padre) is
    do
      padre := nodo
    end -- cambia_padre

    cambia_hijo(nodo: like hijo) is
    do
      hijo := nodo
    end -- cambia_hijo

    cambia_hermano(nodo: like hermano) is
    do
      hermano := nodo
    end -- cambia_hermano
end -- class NODOARB
```

```
indexing
    descripcion : "Implementación de un ARBOL"
    representacion : "Enlazada padre-hijo-hermano"
class ARBOL[ELEM]

creation crear_arbol

feature { NONE }
    raiz, actual : NODOARB[ELEM]

feature { ANY }

    crear_arbol(elem_raiz: ELEM) is
    do
      create raiz.make(elem_raiz)
      actual := raiz
    end -- crear_arbol

    elem_actual : ELEM is
    do
      Result := actual.elem
    end -- elem_actual

    ir_a_raiz is
    do
      actual := raiz
    end -- ir_a_raiz

    ir_a_padre is
    do
      if actual /= raiz then actual := actual.padre end
    end – ir_a_padre

    -- Numero de hijos del elemento actual
    num_hijos: INTEGER is
    local nodo: NODOARB[ELEM]
    do
      Result := 0;
      from nodo := actual.hijo until nodo = Void loop
        nodo := nodo.hermano
        Result := Result+1
      end
    end -- num_hijos

    -- Ir al hijo n-esimo del actual
    ir_a_hijo(n: INTEGER) is
    require
      hijo_existe: (n > 0) and (n <= num_hijos)
    local i : INTEGER
    do
      actual := actual.hijo
      from i := 1 until i = n loop
        actual := actual.hermano
        i := i+1
      end
    end -- ir_a_hijo

    -- Inserta un elemento como ultimo hijo del actual
    insertar(elem: ELEM) is
    local nuevo, ult_hijo : NODOARB[ELEM]
    do
      create nuevo.make(elem)
      nuevo.cambia_padre(actual)
      if actual.hijo = Void then -- Primer hijo
        actual.cambia_hijo(nuevo)
      else
        -- Buscar ultimo hijo
        ult_hijo := actual.hijo
        from until ult_hijo.hermano = Void loop
          ult_hijo := ult_hijo.hermano
        end
        ult_hijo.cambia_hermano(nuevo)
      end
    end -- insertar
```

Recorridos sobre árboles (ordenados)

- **Recorrido Preorden**: Se actua sobre la raiz y luego se recorre en preorden cada uno de los subárboles.
- **Recorrido Postorden**: Se recorre en postorden cada uno de los subárboles y luego se actua sobre la raiz.
- **Recorrido Inorden**: Se recorre en inorden el primer subárbol (si existe). A continuación se actua sobre la raiz y por último se recorre en inorden cada uno de los subárboles restantes.
- **Recorrido por Niveles**: Se etiquetan los nodos según su profundidad (nivel). Se recorren ordenados de menor a mayor nivel, a igualdad de nivel se recorren de izquierda a derecha.

```
feature { ANY }

    -- Los recorridos devuelven una cola con los elem.
    -- del arbol en el orden adecuado
    preorden : COLA[ELEM] is
    do
      create Result.crear_cola
      preorden_rec(Result,raiz)
    end -- preorden

    postorden : COLA[ELEM] is
    do
      create Result.crear_cola
      postorden_rec(Result,raiz)
    end -- postorden

    inorden : COLA[ELEM] is
    do
      create Result.crear_cola
      inorden_rec(Result,raiz)
    end -- inorden

    -- recorrido por niveles
    niveles : COLA[ELEM] is
    local
      aux : COLA[NODOARB[ELEM]] -- cola auxiliar
      nodo, s : NODOARB[ELEM]
    do
      create Result.crear_cola
      create aux.crear_cola
      aux.insertar(raiz)
      from until aux.vacia loop
        -- Se extrae primer nodo de cola auxiliar
        nodo := aux.cabeza; aux.quitar
        Result.insertar(nodo.elem)
        -- Se insertan nodos hijos en cola auxiliar
        from s := nodo.hijo until s = Void loop
          aux.insertar(s)
          s := s.hermano
        end
      end
    end -- niveles
```

```
feature { NONE }

    -- Recorridos recursivos
    preorden_rec(cola: COLA[ELEM]; nodo: NODOARB[ELEM]) is
    local s : NODOARB[ELEM]
    do
      cola.insertar(nodo.elem) -- raiz
      from s := nodo.hijo until s = Void loop
        preorden_rec(cola,s); s := s.hermano
      end
    end -- preorden_rec

    postorden_rec(cola: COLA[ELEM]; nodo: NODOARB[ELEM]) is
    local s : NODOARB[ELEM]
    do
      from s := nodo.hijo until s = Void loop
        postorden_rec(cola,s); s := s.hermano
      end
      cola.insertar(nodo.elem) -- raiz
    end -- postorden_rec

    inorden_rec(cola: COLA[ELEM]; nodo: NODOARB[ELEM]) is
    local s : NODOARB[ELEM]
    do
      s := nodo.hijo
      if s /= Void then inorden_rec(cola,s); s := s.hermano end
      cola.insertar(nodo.elem) -- raiz
      from until s = Void loop
        inorden_rec(cola,s); s := s.hermano
      end
    end -- inorden_rec
end – class ARBOL[ELEM]
```

Variantes de Árboles

- **Árbol binario**: Árbol que consta de un nodo raiz y de dos subárboles, llamados **subárbol izquierdo** y **subárbol derecho**. Se permite que existan árboles vacíos (sin ningún nodo, ni siquiera el raiz). Los árboles vacíos tienen altura –1.

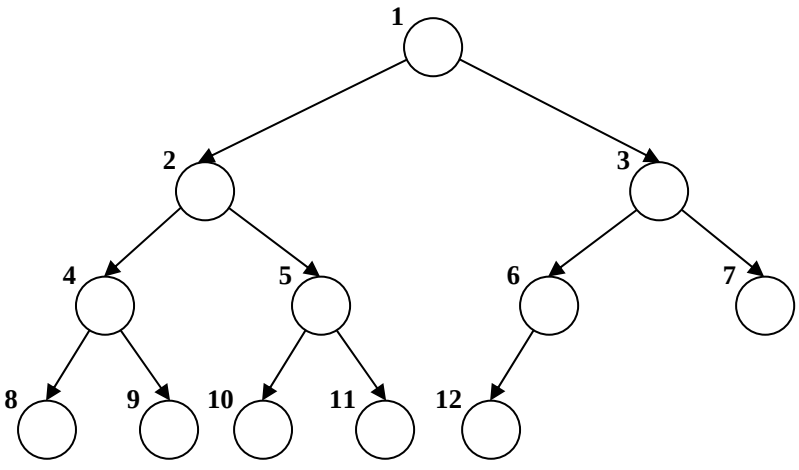
- Cada nodo de un árbol binario puede tener ningún hijo (subárbol izquierdo y derecho vacíos), un hijo (subarbol izquierdo **o** derecho vacío) o dos hijos. Dependiendo de si son la raiz del subarbol izquierdo o derecho se denominan **hijo izquierdo** e **hijo derecho**.

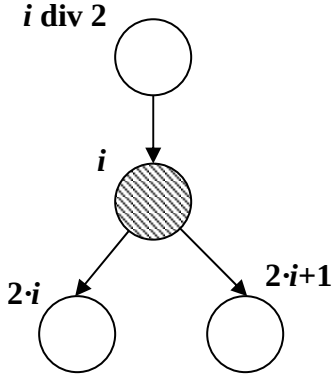
- **Árbol binario estricto**: No se permite que un subárbol esté vacío y el otro no lo esté. Por lo tanto cada nodo puede tener cero o dos hijos.

- **Árbol binario perfectamente equilibrado (árbol lleno)**: La altura del subarbol izquierdo es igual a la altura del subarbol derecho y además ambos subárboles también están perfectamente equilibrados. Un arbol perfectamente equilibrado tiene 2^{h+1}-1 nodos (*h* es la altura del árbol).

- **Árbol binario completo**: Un árbol perfectamente equilibrado hasta el penúltimo nivel, y en el último nivel los nodos se encuentran agrupados a la izquierda.

  En un árbol completo se pueden indexar los nodos mediante un recorrido por niveles, y a partir de ese índice es posible conocer el índice del nodo padre y los índices de los nodos hijos. Esta propiedad permite almacenar un árbol completo en un vector sin necesidad de información adicional (referencia al nodo padre y a los nodos hijos), simplemente almacenando cada nodo en la posición del vector que indica el recorrido por niveles.





- **Montículo**: Un árbol binario completo que almacena elementos con campo clave y donde los nodos cumplen la **propiedad de montículo**: Todo nodo del arbol almacena un elemento cuya clave es **menor** que las claves de sus **descendientes** en el arbol.

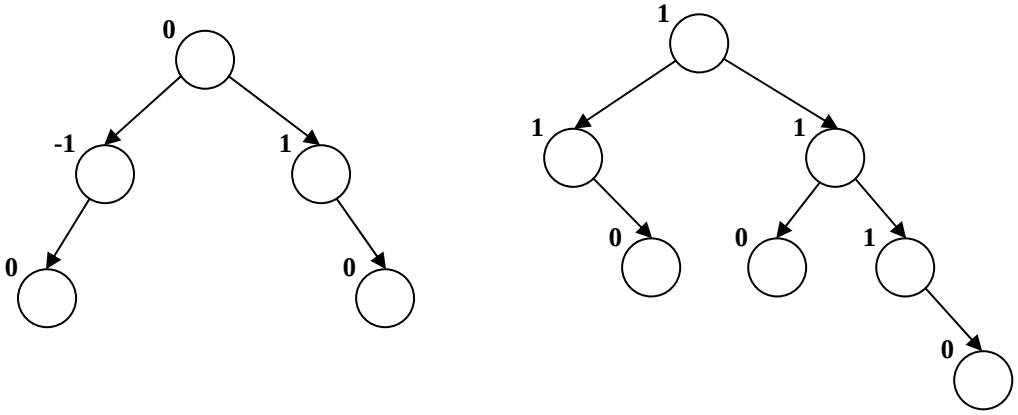
  La definición anterior es de un montículo cuya raiz es el elemento mínimo. Alternativamente, podemos definir un montículo cuya raiz sea el elemento máximo con sólo cambiar la palabra *menor* por *mayor*.

  Si n es el número de elementos del montículo y h su altura se cumple:

  $$n \in \{2^h ... 2^{h+1}-1\}, \quad h = \lfloor \lg n \rfloor, \quad \text{Nivel del nodo } i\text{-ésimo} = \lfloor \lg i \rfloor$$

- **Árbol binario de búsqueda**: Un árbol binario que almacena elementos con campo clave y donde los nodos cumplen la **propiedad de ordenación**: Todo nodo del arbol almacena un elemento cuya clave es **mayor** (o igual) que las claves de los nodos de su **subárbol izquierdo**, y **menor** (o igual) que las claves de los nodos de su **subárbol derecho**.

- **Arbol binario equilibrado**: Un arbol binario en el que la altura del subárbol izquierdo y la del subárbol derecho o son iguales o se diferencian en una unidad, y además ambos subárboles son equilibrados. Se define **factor de equilibrio** de cada nodo como el resultado de restar la altura del subarbol izquierdo a la altura del subarbol derecho. Sólo puede tomar los valores –1, 0 y +1 para un árbol binario equilibrado. Ejemplos:



- **Arbol AVL**: Un arbol binario de búsqueda equilibrado. Comparten las características de los árboles binarios de búsqueda pero el orden de las operaciones de acceso (búsqueda), inserción y borrado es estricto (no es un caso promedio).

Montículos: Operaciones auxiliares

Elevación de un nodo:

```
type
   { El montículo se representa por un vector que almacena sus elementos (registros con campo
     clave) en el orden de un recorrido por niveles. El vector tiene una capacidad máxima de Max
     elementos, y en un momento dado almacena únicamente Num elementos en los índices 1..N }
   TMonticulo = record
      Vec : array[1..MAX] of TElemento;
      Num : integer
   end
```

```
procedure Elevar(var M: TMonticulo; I: Integer);
{ Reorganiza un montículo en el que, al cambiar de valor el nodo I-ésimo, es posible que ya no
  cumpla la propiedad de monticulo para los ascendientes de ese nodo. El algoritmo consiste en
  intercambiar el  nodo con sus ascendientes hasta restablecer la propiedad. Eficiencia: O(lg n) }
var
   Padre, Hijo : integer;
   Seguir : boolean;
begin
   Hijo := I ; Seguir := true ;
   while (Hijo > 1) and Seguir do
   begin
      Padre := Hijo div 2 ;
      if M.Vec[Padre].Clave > M.Vec[Hijo].Clave then
      begin
         { No se cumple la propiedad de montículo: Se intercambia el nodo padre con el nodo hijo y se
           sigue comprobando los ascendientes }
         M.Vec[Padre] ⇔ M.Vec[Hijo] ;
         Hijo := Padre
      end else begin
         Seguir := false
      end { if }
   end { while }
end; { Elevar }
```

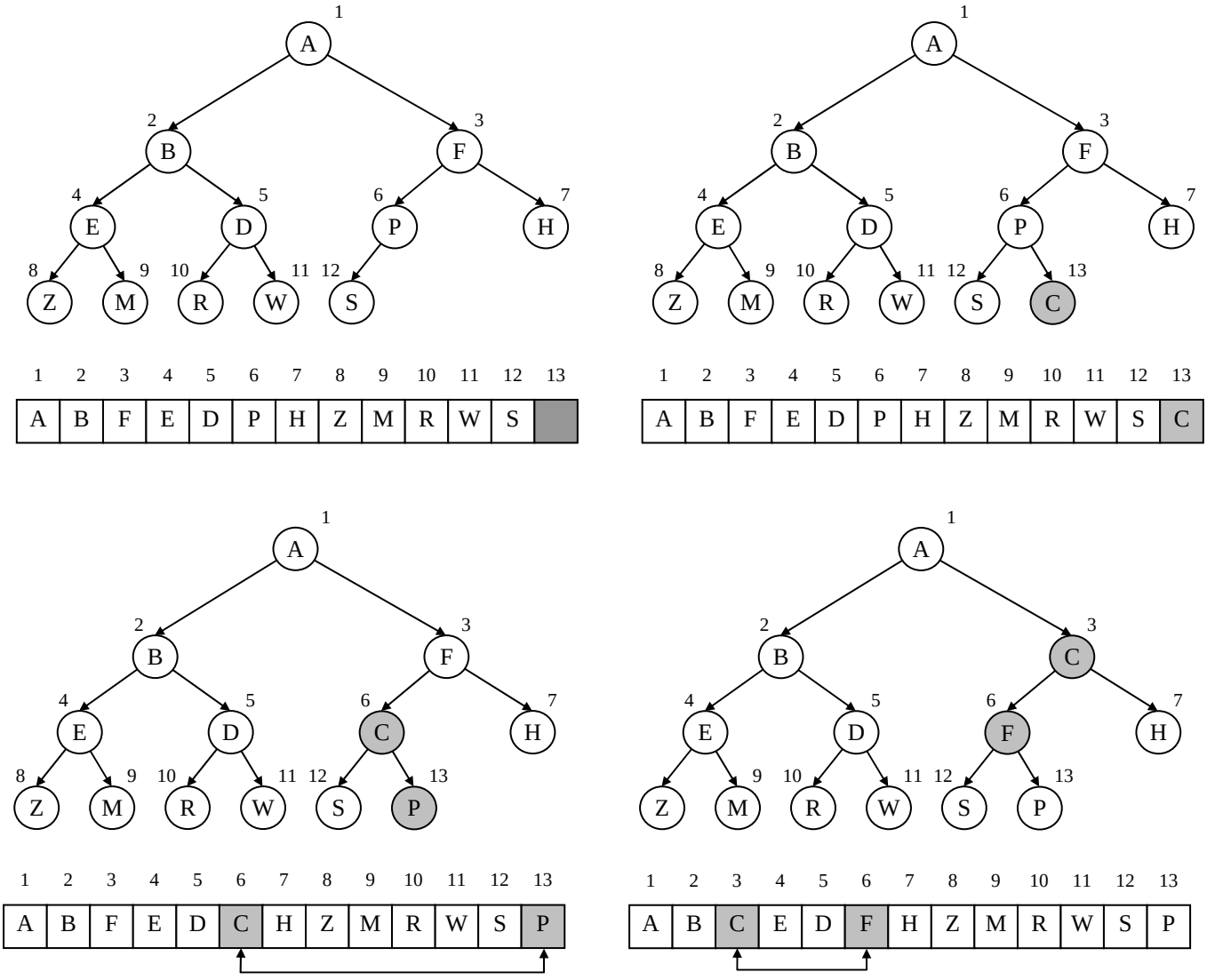
Reestructuración de un (sub)montículo:

```
procedure Reestructurar(var M: TMonticulo; I: integer) ;
{ Reorganiza un montículo en el que, al cambiar de valor el nodo I-ésimo, es posible que ya no se
  cumpla la propiedad de monticulo para los descendientes de ese nodo. Alternativamente, esta
  operación se puede contemplar como reorganizar un (sub)montículo cuya raiz es el nodo I, donde
  todos los nodos excepto la raiz cumplen la propiedad de montículo. El algoritmo consiste en
  intercambiar el nodo con sus descendientes hasta restablecer la propiedad. Eficiencia: O(lg n) }
var
   Padre, Hijo : integer;
   Seguir : boolean;
begin
   Padre := I;
   Hijo := 2*Padre ;   { hijo izquierdo }
   Seguir := Cierto
   while (Hijo ≤ M.Num) and Seguir do
   begin
      { Comprobar cual es el hijo con clave menor }
      if Hijo < M.Num then   { existe hijo derecho }
         if M.Vec[Hijo+1].Clave < M.Vec[Hijo].Clave then  { hijo derecho es el menor }
            Hijo := Hijo+1;
      { Comprobar si el padre tiene una clave mayor que la del hijo menor }
      if M.Vec[Padre].Clave > M.Vec[Hijo].Clave then
      begin
         { No se cumple la propiedad de montículo: Se Intercambia el nodo padre con el nodo hijo y
           se sigue comprobando los descendientes }
         M.Vec[Padre] ⇔ M.Vec[Hijo] ;
         Padre := Hijo ;
         Hijo := 2*Padre
      end else begin
         Seguir := false
      end  { if }
   end  { while  }
end;  { reestructurar }
```

Operaciones sobre montículos

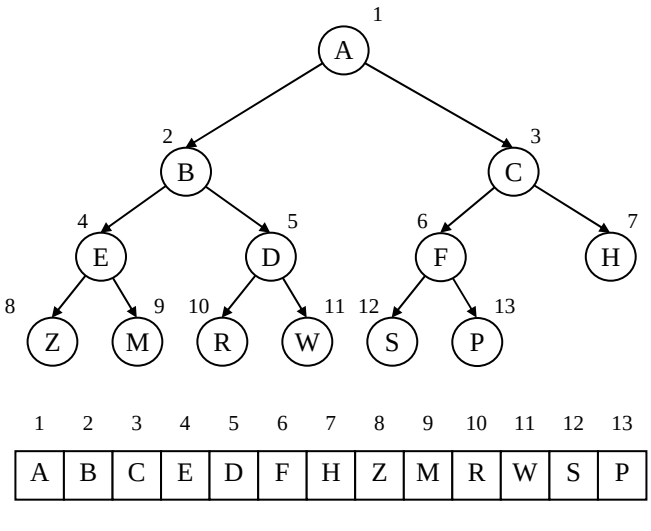
Inserción de un elemento:

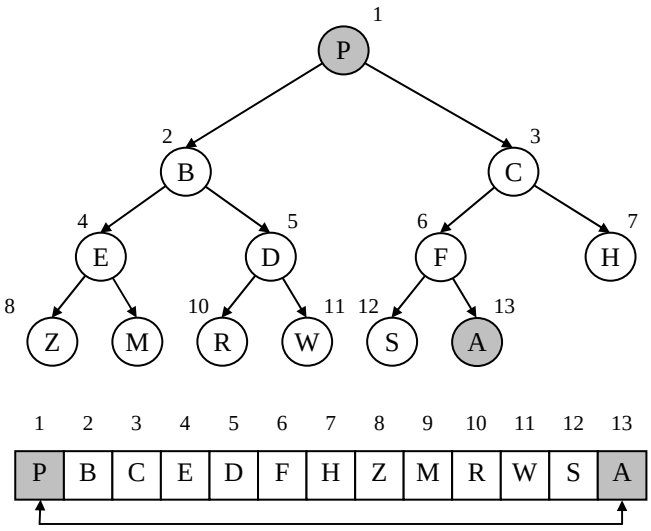
```
procedure Insertar(var M: TMonticulo ; E: TElemento) ;
{ Inserta el elemento E en el montículo M de manera que se siga manteniendo la estructura de
  montículo. Eficiencia: O(lg n) }
begin
   M.Num := M.Num+1 ;
   if M.Num > MAX then { ampliar capacidad de M.Vec } ;
   M.Vec[M.Num] := E;  { Inserción al final del vector, como el último nodo hoja }
   Elevar(M, M.Num)      { Reorganizar el montículo elevando lo necesario el nodo insertado }
end; { Insertar }
```

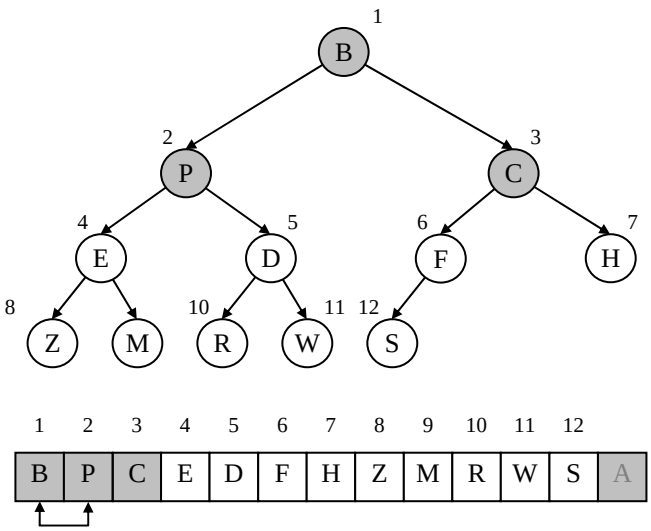


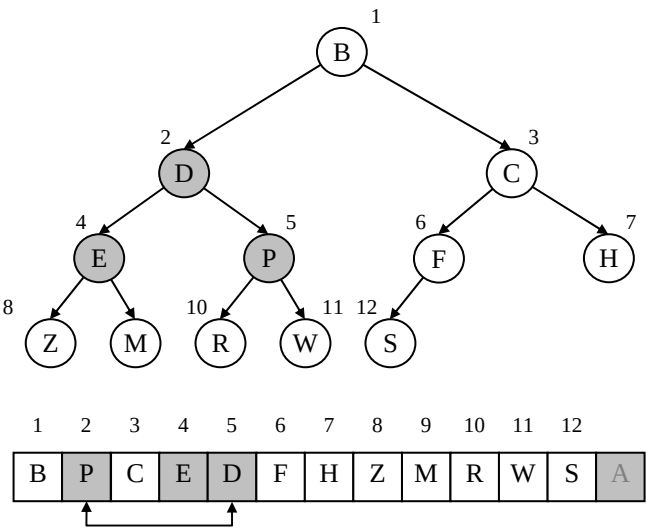
Borrado del nodo raiz (elemento con clave mínima):

```
procedure Borrar(var M: TMonticulo) ;
{ Precondición: M.Num > 1 }
begin
   M.Vec[1] ⇔ M.Vec[M.Num] ;   { Se intercambia el raiz con el último }
   M.Num := M.Num-1 ;          { Se borra el último elemento (el que era antes el raiz) }
   Reestructurar(M, 1)         { Se reestructura todo el montículo (se hace descender la raiz) }
end; { Borrar }
```









Modificación de un nodo:

```
procedure Modificar(var M: TMonticulo; I: integer; E: TElemento) ;
{ Cambia el valor del elemento I-ésimo del montículo por E }
begin
   if E.Clave < M.Vec[I].Clave then { Se cambia un elemento por otro menor }
      { Sólo puede afectar a los ascendientes del nodo que cambia }
      M.Vec[I] := E ;
      Elevar(M, I)                { Elevar el nodo lo necesario }
   end else begin                 { Se cambia un elemento por otro mayor }
      { Sólo puede afectar a los descendientes del nodo que cambia }
      M.Vec[I] := E ;
      Reestructurar(M, I)         { Descender el nodo lo necesario }
   end
end; { Modificar }
```

Creación de un montículo a partir de un vector desordenado:

```
procedure Crear(var M: TMonticulo) ;
{ En el array M.Vec[1..M.Num] se almacenan elementos desordenados. Este procedimiento
  reorganiza el array para que pase a tener estructura de montículo. Eficiencia: O(n) }
var I : integer;
begin
   { Realiza una secuencia de reestructuraciones desde los niveles inferiores (comenzando por el
     padre del último nodo) hasta la raiz. }
   for I := M.Num div 2 downto 1 do
      Reestructurar(M, I)
end; { Crear }
```

Implementación de Colas de prioridad

	Lista no ordenada*		Lista ordenada**	Montículo
Acceder al mínimo	$O(n)$	$O(1)$	$O(1)$	$O(1)$
Insertar	$O(1)$	$O(1)$	$O(n)$	$O(\lg n)$
Borrar el mínimo	$O(n)$	$O(n)$	$O(1)$	$O(\lg n)$
Modificar	$O(1)$	$O(n)$	$O(n)$	$O(\lg n)$

*La columna derecha representa la variante de almacenar una referencia al elemento minimo y actualizarla adecuadamente en el resto de operaciones. Esto supone que el acceso es $O(1)$ y modificar pasa a ser $O(n)$.

**Se supone que está ordenada de mayor a menor.

Ordenación por montículos:

```
procedure OrdMonticulos(var M: TMonticulo);
{ Ordena el vector M.Vec[1..M.Num] de mayor a menor. Primero reorganiza el vector para dotarle
  de estructura de montículo, y despues extrae sus mínimos sucesivos (que se van depositando al
  final del vector). Cuando el montículo queda vacío, la zona del vector M.Vec[1..M.Num] contiene los
  elementos originales ordenados de mayor a menor. Si se desea el orden habitual de menor a mayor
  se debe trabajar con un montículo de máximos en lugar de mínimos (basta con cambiar la
  comparación entre padres e hijos de menor a mayor en los subprogramas Elevar y Reestructurar }
var I, N : integer;
begin
   N := M.Num;   { Tamaño original }
   Crear(M);     { Reorganiza el vector como montículo }
   { Extracción de los elementos máximos, que se depositan en orden al final del vector, fuera de la
     parte que representa el montículo. El último elemento no es necesario extraerlo. }
   for I := 1 to N-1 do  Borrar(M);
   M.Num := N    { Se restablece el tamaño original }
end; { OrdMonticulos }
```

Comparación de algoritmos avanzados de ordenación:

	Eficiencia		**Ctes. de proporcionalidad**	
	Tiempo	**Espacio**	**Movimientos**	**Comparaciones**
Fusión	$O(n \log n)$	$O(n)$	2.00	0.92
Rápida	$\Omega(n \log n)^*$	$\Omega(\log n)^*$	0.75	1.35
Montículos	$O(n \log n)$	$O(1)$	1.33	1.80

* Se dan las cotas y constantes del caso promedio, el peor caso sería tiempo $O(n^2)$ y espacio $O(n)$

Implementación de un Arbol Bin. de Busqueda (I)

- Definición del nodo (adaptado para árboles AVL y para la operación de acceso por posición):

```
class NODO_AVL[ELEMENTO]
creation make
feature { ANY }
    clave : ELEMENTO;       -- datos
    num_nod : INTEGER;      -- numero de nodos que almacena el subarbol cuya raiz es este nodo
    fe : INTEGER;           -- factor de equilibrio del nodo
    izdo,dcho,padre : like Current; -- enlaces

    make(k: ELEMENTO) is
    do
      clave := k ; num_nod := 1; fe := 0;
      izdo := Void ; dcho := Void ; padre := Void
    end -- make

    calc_num_nod is -- recalcula el numero de nodos
    do
      num_nod := 1
      if izdo /= Void then num_nod := num_nod + izdo.num_nod end
      if dcho /= Void then num_nod := num_nod + dcho.num_nod end
    end -- calc_num_nod

    cambia_clave(k: ELEMENTO) is do clave := k end -- cambia_clave

    cambia_num_nod(n: INTEGER) is do num_nod := n end -- cambia_num_nod

    cambia_fe(n: INTEGER) is do fe := n end -- cambia_fe

    cambia_izdo(i: like izdo) is do izdo := i end -- cambia_izdo

    cambia_dcho(d: like dcho) is do dcho := d end -- cambia_dcho

    cambia_padre(p: like padre) is do padre := p end -- cambia_padre
end -- NODO_AVL
```

- Definición del nodo y operaciones auxiliares:

```
class ARBOL_ABB[ELEMENTO -> COMPARABLE]
creation crea_arbol
feature { NONE }
    raiz, act : NODO_AVL[ELEMENTO];

    -- Búsqueda en el arbol del elemento k. Devuelve el nodo  donde debería insertarse (como hijo izquierdo o
    -- derecho). Eficiencia : O(lg n) promedio, O(n) peor
    busq_abb(k: ELEMENTO) : like raiz is
    require raiz /= Void
    local
      nodo : like raiz; fin : BOOLEAN;
    do
      from nodo := raiz; fin := False until fin loop
        if k < nodo.clave then
          if nodo.izdo /= Void then nodo := nodo.izdo else fin := true end
        else -- k >= nodo.clave
          if nodo.dcho /= Void then nodo := nodo.dcho else fin := true end
        end
      end
      Result := nodo
    ensure
      insercion_correcta:
        k <   Result.clave implies Result.izdo = Void
        k >= Result.clave implies Result.dcho = Void
    end -- busq_abb

    -- Continúa  en la siguiente página
```

Implementación de un Arbol Bin. de Busqueda (II)

- Definición del nodo y operaciones auxiliares (continuación):

```
    -- Búsqueda del n-ésimo nodo
    busq_enesimo(nodo: like raiz; n: INTEGER) : like raiz is
    require nodo /= Void
    local niz : integer;
    do
     if nodo.izdo /= Void then niz := nodo.izdo.num_nod else niz := 0 end
     if n = niz+1 then
       Result := nodo
     elseif n < ni+1 then
       Result := busq_nesimo(nodo.izdo, p)
     else -- n > niz+1
       Result := busq_nesimo(nodo.dcho, n-niz-1)
     end
    end -- busca_enesimo

    -- Recorre ascendientes incrementando o decrementando el número de nodos
    adapta_num_nod(nodo: like raiz; incr: INTEGER) is
    require n /= Void
    do
     nodo.cambia_num_nod(nodo.num_nod+incr)
     if n.padre /= Void then adapta_num_nod(nodo.padre,incr) end
    end -- adapta_num_nod

    -- Encuentra el nodo con valor mínimo del subarbol cuya raiz es nodo
    minimo(n: like raiz) : like raiz is
    require n /= Void
    do
     if nodo.izdo = Void then Result := nodo else Result := minimo(nodo.izdo) end
    end -- minimo
feature { ANY }
    -- Operaciones principales del arbol bin. búsqueda
feature { NONE }
    -- Operaciones del invariante del arbol bin. búsqueda
invariant –- Invariante de la clase
    es_arbol_binario(raiz) and then es_arbol_bin_busq(raiz)
end – ARBOL_ABB
```

- Operaciónes de creación, búsqueda y acceso al n-ésimo menor:

```
    crea_arbol is
    do
     raiz := Void ; act := Void
    end -- crea_arbol

    num_elems : INTEGER is
    do
     if raiz = Void then Result := 0 else Result := raiz.num_nod end
    end -- num_claves

    -- Eficiencia: O(lg n) promedio, O(n) peor
    existe(k: ELEMENTO) : BOOLEAN is
    local nodo : like raiz;
    do
     if raiz = Void then Result := false else nodo := busq_abb(k) ; Result := (nodo.clave = k) end
    end – existe

    -- Eficiencia: O(lg n) promedio, O(n) peor
    elem_nesimo(n: INTEGER) : ELEMENTO is
    do
     Result := busq_nesimo(raiz, n)
    end -- elem_nesimo
```

Implementación de un Arbol Bin. de Busqueda (III)

- Inserción de un elemento:

```
-- Eficiencia: O(lg n) promedio, O(n) peor
insertar(k: ELEMENTO) is
local nodo, nuevo : like raiz;
do
 if raiz = Void then
   create raiz.make(k)
    act := Void -- actual ya no es valido
 else
   nodo := busq_abb(k);
    -- Crear nuevo nodo
   create nuevo.make(k)
   -- Insertarle como hijo izdo o dcho
   nuevo.cambia_padre(nodo)
   if k < nodo.clave then nodo.cambia_izdo(nuevo) else nodo.cambia_dcho(nuevo) end
    -- Actualizar numero de nodos
    adapta_num_nod(nodo,+1)
    -- actual ya no es valido
    act := Void
 end
end -- insertar
```

- Borrado de un elemento:

```
-- Eficiencia: O(lg n) promedio, O(n) peor
quitar(k: ELEMENTO) is
local nodo, hijo: like raiz; fin : BOOLEAN;
do
 if raiz /= Void then -- no es arbol vacio
   nodo := busq_abb(k)
   if nodo.clave = k then -- clave existe
     -- Este bucle se repite solo 1 o 2 veces
     from fin := False until fin loop
       if nodo.izdo = Void or nodo.dcho = Void then -- caso 0 o 1 hijo
         -- hijo del nodo borrado que le sustituye (puede estar vacío)
         if nodo.izdo = Void then hijo := nodo.dcho else hijo := nodo.izdo end
         if nodo = raiz then  -- El nodo borrado es el raiz y solo tiene un hijo
           raiz := hijo
         else
           hijo.cambia_padre(nodo.padre)
           -- Adaptar enlaces del padre
           if padre.izdo = nodo then nodo.padre.cambia_izdo(hijo) else nodo.padre.cambia_dcho(hijo) end
           -- Decrementar numero nodos en ascendientes
           adapta_num_nod(nodo.padre,-1)
         end
         act := Void -- actual ya no es valido
         fin := True -- fin del bucle
       else -- caso 2 hijos
         -- Se busca el minimo del subarbol dcho
         hijo := minimo(nodo.dcho)
         -- El minimo toma el lugar del nodo a borrar
         nodo.cambia_clave(hijo.clave);
         -- En la siguiente iteracion se borra el nodo mínimo (sus datos ya estan salvados)
         nodo := hijo
       end -- deteccion de casos
     end -- bucle
   end -- clave existe
 end -- arbol no vacio
end -- quitar
```

Implementación de un Arbol Bin. de Busqueda (IV)

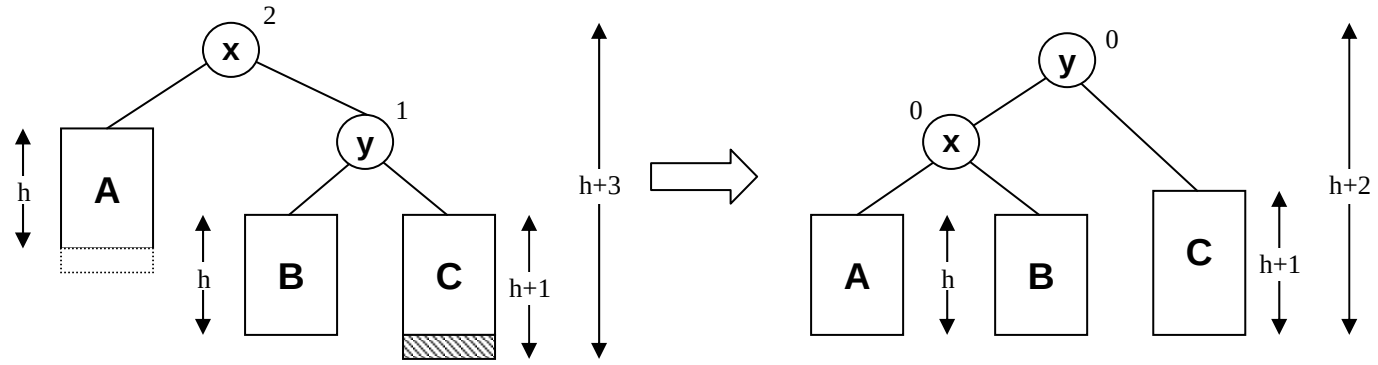
- Operaciones de recorrido basado en cursor:

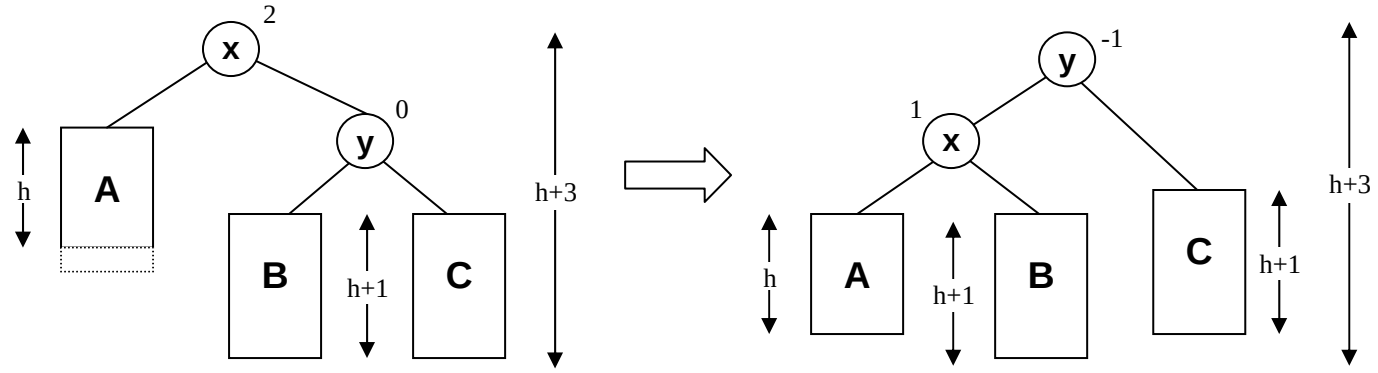
```
-- Nota: Al insertar, borrar, o ir al siguiente del último el cursor pasa a no ser válido
actual_valido : BOOLEAN is
do
  Result := act /= Void
end -- actual_valido

actual : ELEMENTO is
require actual_valido
do
  Result := act.clave
end -- actual

-- Eficiencia: O(lg n) promedio, O(n) peor
ir_a_inicio is
do
  if raiz = Void then act := Void else act := minimo(raiz) end
end -- ir_a_inicio

-- Eficiencia: O(lg n) promedio, O(n) peor
ir_a_siguiente is
local nodo : like raiz;
do
  if act /= Void then
    if act.dcho /= Void then
      act := minimo(act.dcho) -- El siguiente es el mínimo del subarbol dcho
    else
      -- El siguiente es el padre del primer ascendiente que sea hijo izdo
      from nodo := act until (nodo.padre = Void) or else (nodo.padre.izdo = nodo) loop
        nodo := nodo.padre
      end
      act := nodo.padre
    end
  end
end -- ir_a_siguiente
```

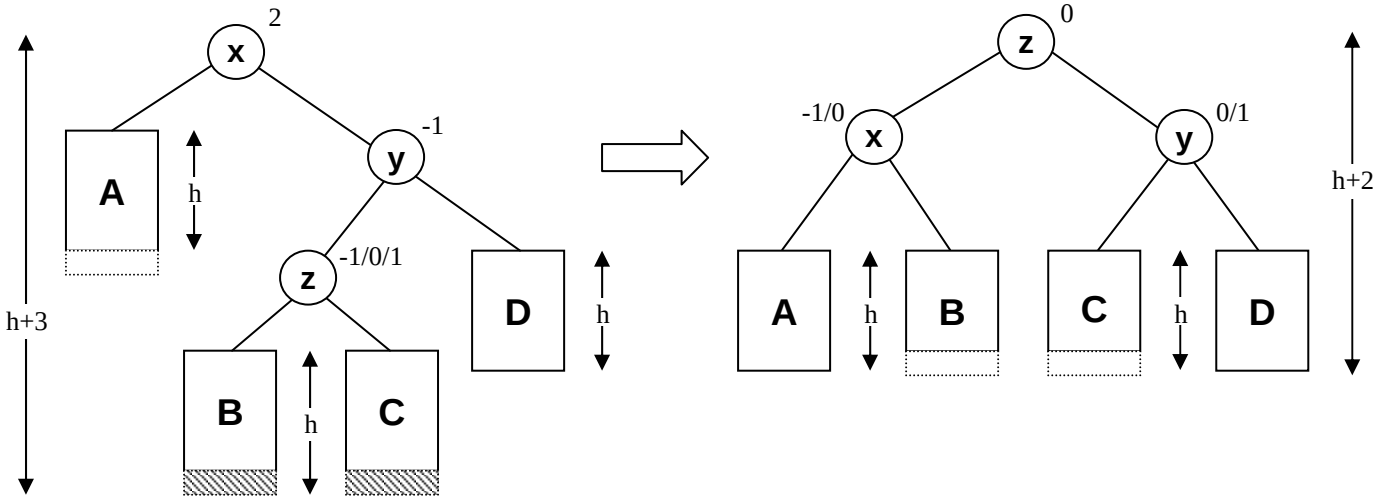
- Invariantes de clase (arbol correcto):

```
-- Comprueba que es un subarbol binario correcto
es_arbol_binario(nodo: like raiz) : BOOLEAN is
do
  Result := True
  if nodo /= Void then
    if nodo.izdo /= Void then
      Result := (nodo.izdo.padre = nodo) and then es_arbol_binario(nodo.izdo)
    end
    if Result and nodo.dcho /= Void then
      Result := (nodo.dcho.padre = nodo) and then es_arbol_binario(nodo.dcho)
    end
  end
end -- es_arbol_binario

-- Comprueba que cumple la propiedad de orden
es_arbol_bin_busq(nodo: like raiz) : BOOLEAN is
do
  Result := True
  if nodo /= Void then
    if nodo.izdo /= Void then
      Result := (nodo.izdo.clave <= nodo.clave) and then es_arbol_bin_busq(nodo.izdo)
    end
    if Result and nodo.dcho /= Void then
      Result := (nodo.dcho.clave >= nodo.clave) and then es_arbol_bin_busq(nodo.dcho)
    end
  end
end -- es_arbol_bin_busq
```

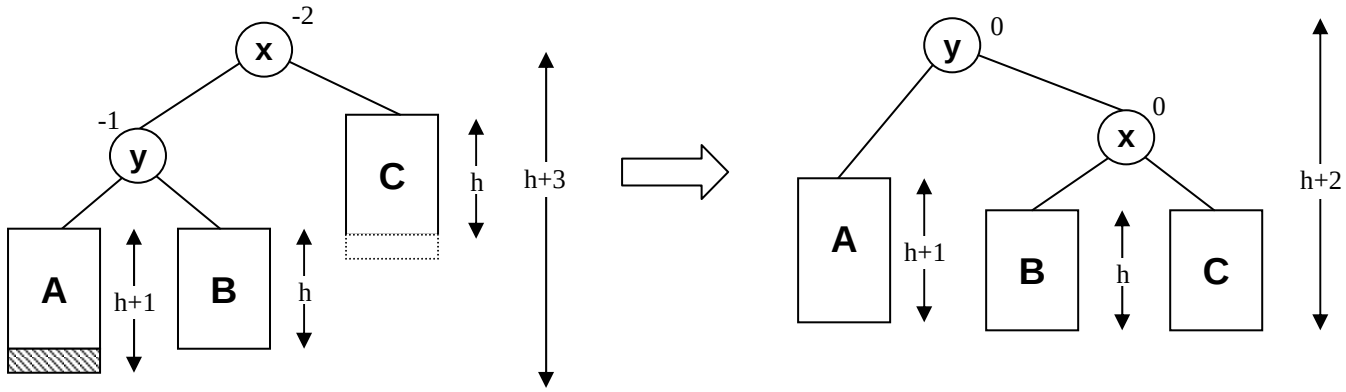
Arboles AVL - Rotaciones

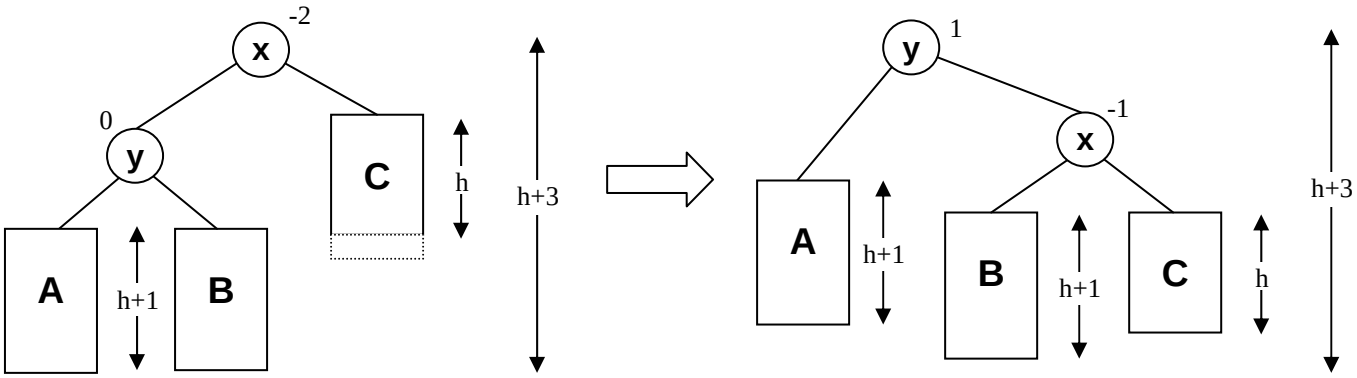


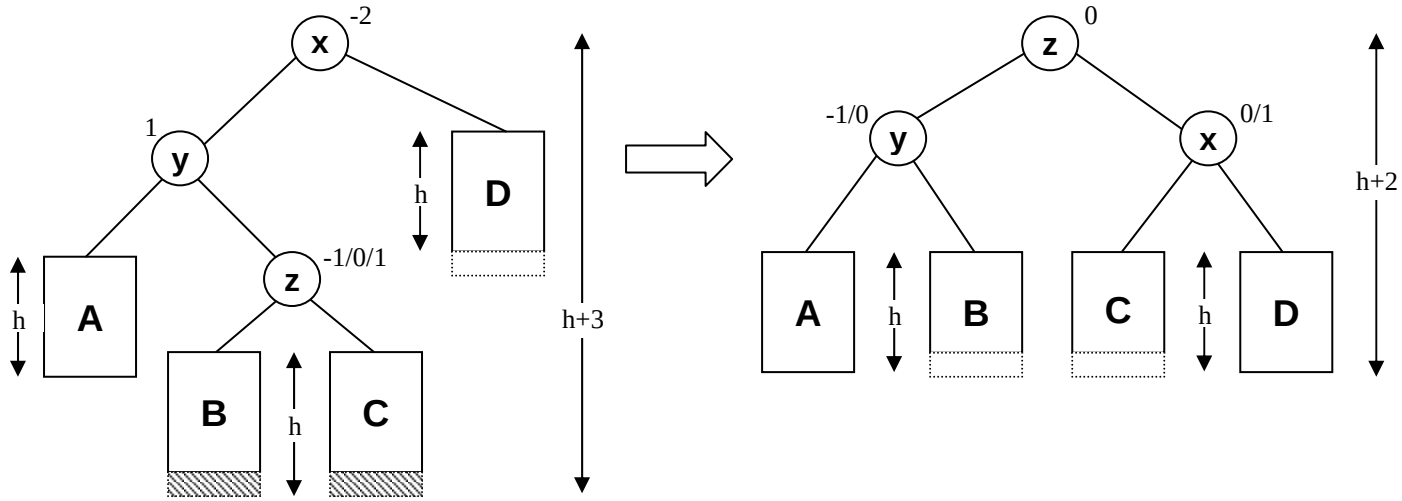




Arboles AVL – Rotaciones simétricas







Implementación de un Arbol AVL (I)

- Definición de la clase y rotaciones:

```
class ARBOL_AVL[ELEMENTO -> COMPARABLE]
inherit ARBOL_ABB[ELEMENTO]
  redefine insertar, quitar
end
creation crea_arbol
feature { NONE }
```

```
rot_simple_pos(x: like raiz) is
local y,b : like raiz;
do
  y := x.dcho ; b := y.izdo
  -- Modificar enlaces del padre de x
  if x.padre /= Void then
    if x.padre.izdo = x then
      x.padre.cambia_izdo(y)
    else
      x.padre.cambia_dcho(y)
    end
  else
    raiz := y
  end
  y.cambia_padre(x.padre) ; y.cambia_izdo(x)
  x.cambia_padre(y) ; x.cambia_dcho(b)
  if b /= Void then b.cambia_padre(x) end
  -- Factores de equilibrio
  if y.fe = +1 then
    y.cambia_fe(0) ; x.cambia_fe(0)
  else
    y.cambia_fe(-1) ; x.cambia_fe(+1)
  end
  -- Recalcular numero de nodos
  x.calcula_num_nod ;  y.calcula_num_nod
end -- rot_simple_pos
```

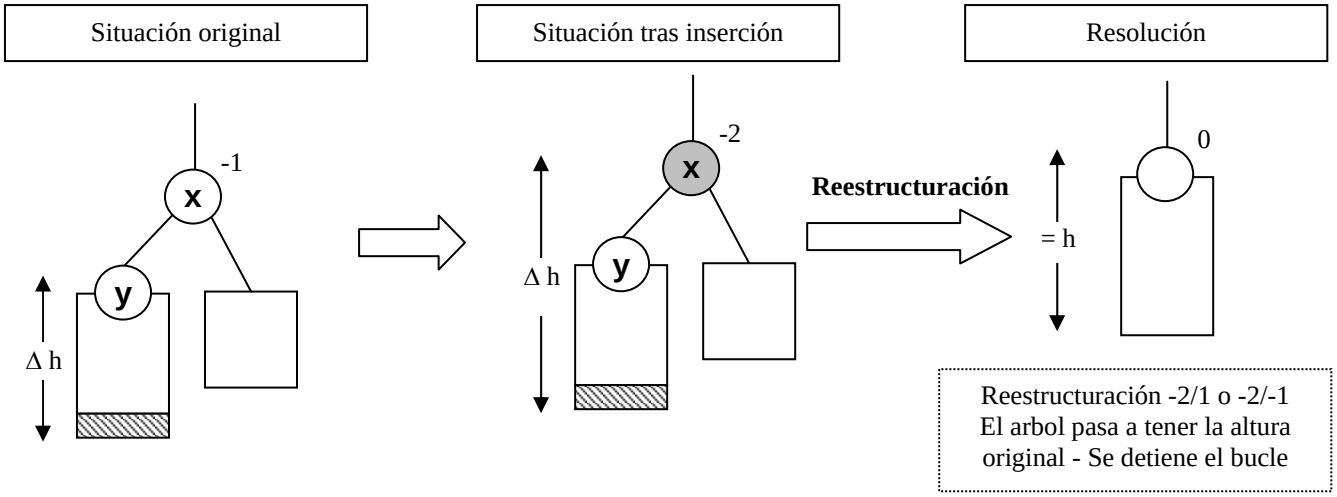
```
rot_simple_neg(x: like raiz) is
local y,b : like raiz;
do
  y := x.izdo ; b := y.dcho
  -- Modificar enlaces del padre de x
  if x.padre /= Void then
    if x.padre.izdo = x then
      x.padre.cambia_izdo(y)
    else
      x.padre.cambia_dcho(y)
    end
  else
    raiz := y
  end
  y.cambia_padre(x.padre) ; y.cambia_dcho(x)
  x.cambia_padre(y) ; x.cambia_izdo(b)
  if b /= Void then b.cambia_padre(x) end
  -- Factores de equilibrio
  if y.fe = -1 then
    y.cambia_fe(0) ; x.cambia_fe(0)
  else
    y.cambia_fe(+1) ; x.cambia_fe(-1)
  end
  -- Recalcular numero de nodos
  x.calcula_num_nod ;  y.calcula_num_nod
end -- rot_simple_neg
```

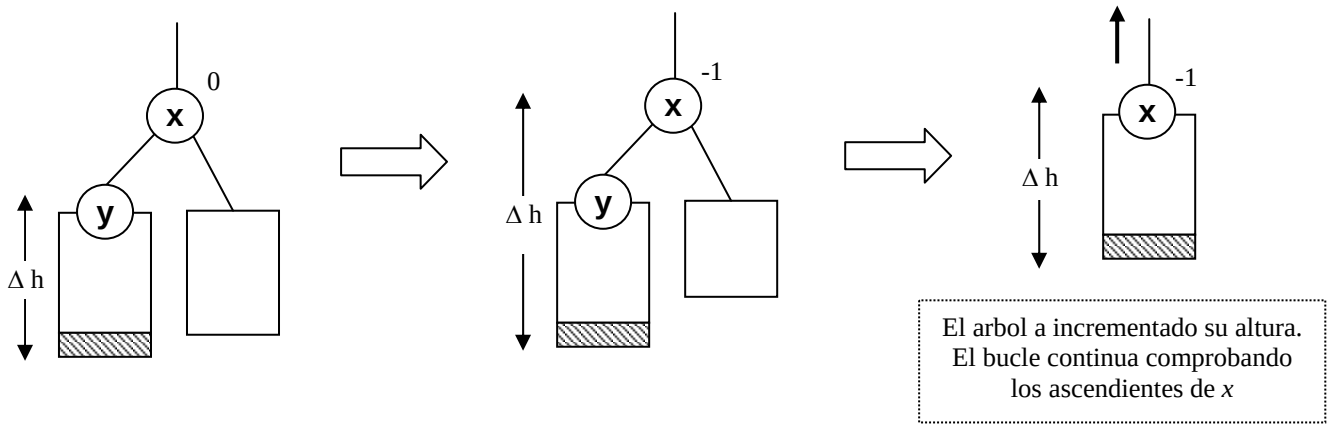
```
rot_doble_pos(x: like raiz) is
local y,z,b,c : like raiz;
do
  y := x.dcho ; z := y.izdo ; b := z.izdo ; c := z.dcho
  -- Modificar enlaces del padre de x
  if x.padre /= Void then
    if x.padre.izdo = x then
      x.padre.cambia_izdo(z)
    else
      x.padre.cambia_dcho(z)
    end
  else
    raiz := z
  end
  z.cambia_padre(x.padre)
  z.cambia_izdo(x) ; z.cambia_dcho(y)
  x.cambia_padre(z) ; x.cambia_dcho(b)
  y.cambia_padre(z) ; y.cambia_izdo(c)
  if b /= Void then b.cambia_padre(x) end
  if c /= Void then c.cambia_padre(y) end
  -- Factores de equilibrio
  if z.fe = -1 then
    x.cambia_fe(0) ; y.cambia_fe(1)
  elseif z.fe = 0 then
    x.cambia_fe(0) ; y.cambia_fe(0)
  else -- z.fe = 1
    x.cambia_fe(-1) ; y.cambia_fe(0)
  end
  z.cambia_fe(0)
  x.calc_num_nod ; y.calc_num_nod ; z.calc_num_nod
end -- rot_doble_pos
```

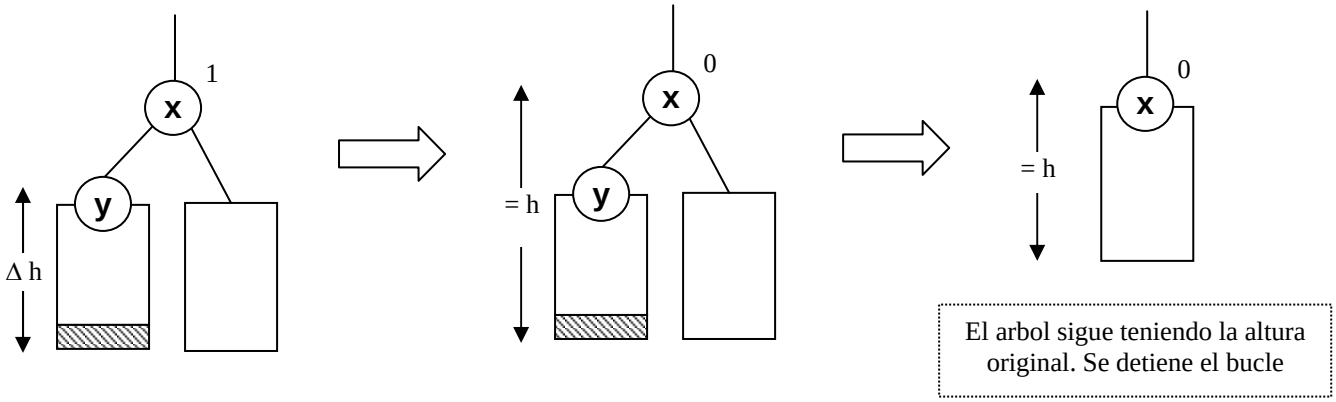
```
rot_doble_neg(x: like raiz) is
local y,z,b,c : like raiz;
do
  y := x.izdo ; z := y.dcho ; b := z.izdo ; c := z.dcho
  -- Modificar enlaces del padre de x
  if x.padre /= Void then
    if x.padre.izdo = x then
      x.padre.cambia_izdo(z)
    else
      x.padre.cambia_dcho(z)
    end
  else
    raiz := z
  end
  z.cambia_padre(x.padre)
  z.cambia_izdo(y) ; z.cambia_dcho(x)
  x.cambia_padre(z) ; x.cambia_izdo(c)
  y.cambia_padre(z) ; y.cambia_dcho(b)
  if b /= Void then b.cambia_padre(y) end
  if c /= Void then c.cambia_padre(x) end
  -- Factores de equilibrio
  if z.fe = -1 then
    x.cambia_fe(1) ; y.cambia_fe(0)
  elseif z.fe = 0 then
    x.cambia_fe(0) ; y.cambia_fe(0)
  else -- z.fe = 1
    x.cambia_fe(0) ; y.cambia_fe(-1)
  end
  z.cambia_fe(0)
  x.calc_num_nod ; y.calc_num_nod ;  z.calc_num_nod
end -- rot_doble_neg
```

Arboles AVL - Comprobación tras Inserción

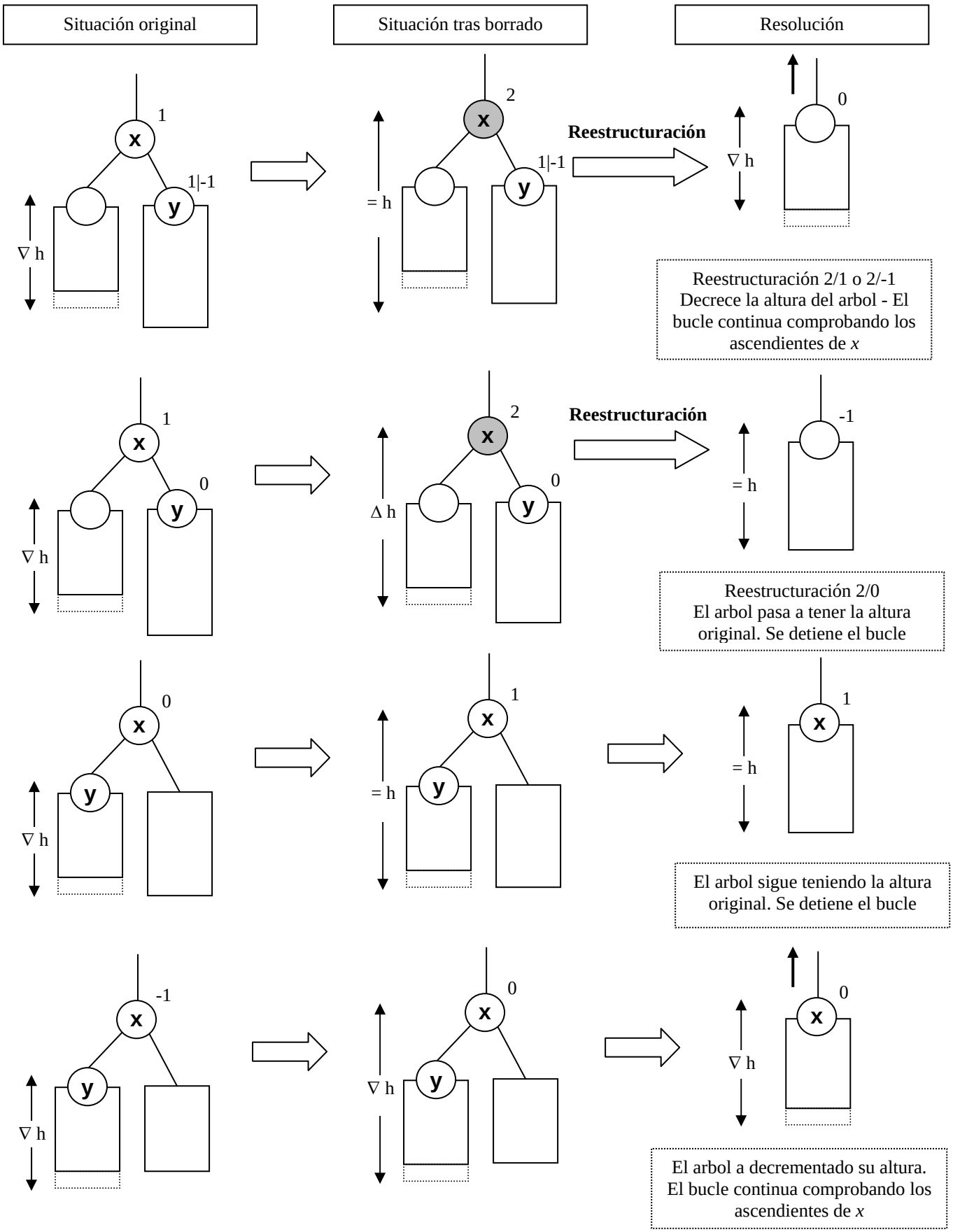
La comprobación consiste en un bucle donde se analiza el cambio sufrido por un nodo, *x*, en el que uno de sus subarboles a incrementado su altura en 1 debido a una inserción. Existen 6 posibilidades (3 casos cuando el subarbol que cambia es el izquierdo y otros 3 cuando es el derecho), dependiendo del factor de equilibrio original de *x*:







Arboles AVL - Comprobación tras Borrado



Implementación de un Arbol AVL (II)

- Comprobaciones tras inserción y borrado:

```
-- Se ha insertado el nodo n en el arbol.
-- Se comprueba si es necesario reestructurar
-- y se adapta el campo número de nodos.
comprobar_insercion(n: like raiz) is
local
  x,y : like raiz; fin : BOOLEAN;
do
  from y := n ; x := n.padre ; fin := False
  until fin or (x = Void) loop
    if x.izdo = y then -- y es hijo izdo
      x.cambia_fe(x.fe-1)
      if x.fe = -2 then -- reestructuracion y final
        if y.fe = +1 then
          rot_doble_neg(x)
        else
          rot_simple_neg(x)
        end
        fin := true
      elseif x.fe = -1 then -- continuar
        x.cambia_num_nod(x.num_nod+1)
        x := x.padre
      else -- x.fe = 0 -- final
        fin := true
      end
    else -- x.dcho = y -- y es hijo dcho
      x.cambia_fe(x.fe+1)
      if x.fe = +2 then -- reestructuracion y final
        if y.fe = -1 then
          rot_doble_pos(x)
        else
          rot_simple_pos(x)
        end
        fin := true
      elseif x.fe = +1 then -- continuar
        x.cambia_num_nod(x.num_nod+1)
        x := x.padre
      else -- x.fe = 0 -- final
        fin := true
      end
    end
  end -- bucle que recorre ascendientes
  -- Incrementar num. nodos ascend. restantes
  if x /= Void then adapta_num_nod(x,+1) end
end -- comprobar_insercion
```

```
-- Se ha borrado un nodo del subarbol izdo
-- (hijo_izdo = True) o del subarbol derecho del nodo n.
-- Se comprueba si es necesario reestructurar el arbol y
-- se adapta el campo número de nodos.
comprobar_borrado(n: like raiz; hijo_izdo: BOOLEAN) is
local
  x : like raiz; es_izdo, fin : BOOLEAN;
do
  from x := n ; es_izdo := hijo_izdo ;      fin := False
  until fin or (x = Void) loop
    if es_izdo then -- El subarbol izdo ha decrecido
      x.cambia_fe(x.fe+1)
      if x.fe = +2 then -- reestructuracion
        if x.dcho.fe = -1 then
          rot_doble_pos(x)
        else
          rot_simple_pos(x)
        end
        if x.dcho.fe = 0 then
          fin := true
        else
          if x.padre /= Void then es_izdo := x.padre.izdo = x end
          x := x.padre
        end
      elseif x.fe = +1 then -- final
        fin := true
      else -- x.fe = 0 -- continuar
        if x.padre /= Void then es_izdo := x.padre.izdo = x end
        x := x.padre
      end
    else -- El subarbol dcho ha decrecido
      x.cambia_fe(x.fe-1)
      if x.fe = -2 then -- reestructuracion
        if x.dcho.fe = +1 then
          rot_doble_neg(x)
        else
          rot_simple_neg(x)
        end
        if x.dcho.fe = 0 then
          fin := true
        else
          if x.padre /= Void then es_izdo := x.padre.izdo = x end
          x := x.padre
        end
      elseif x.fe = -1 then -- final
        fin := true
      else -- x.fe = 0 -- continuar
        if x.padre /= Void then es_izdo := x.padre.izdo = x end
        x := x.padre
      end
    end
  end -- bucle que recorre ascendientes
  -- Decrementar num. nodos ascendientes restantes
  if x /= Void then adapta_num_nod(x,-1) end
end -- comprobar_borrado
```

Implementación de un Arbol AVL (III)

- Operación de inserción:

```
-- Eficiencia: O(lg n)
insertar(k: ELEMENTO) is
local nodo, nuevo : like raiz;
do
  if raiz = Void then
    create raiz.make(k)
    act := Void -- actual ya no es valido
  else
    nodo := busq_abb(k);
    -- Crear nuevo nodo
    create nuevo.make(k)
    -- Insertarle como hijo izdo o dcho
    nuevo.cambia_padre(nodo)
    if k < nodo.clave then nodo.cambia_izdo(nuevo) else nodo.cambia_dcho(nuevo) end
    comprobar_insercion(nuevo)
     -- actual ya no es valido
     act := Void
  end
end -- insertar
```

- Operación de borrado:

```
-- Eficiencia: O(lg n)
quitar(k: ELEMENTO) is
local nodo, hijo: like raiz; fin, es_hijo_izdo : BOOLEAN;
do
  if raiz /= Void then -- no es arbol vacio
    nodo := busq_abb(k)
    if nodo.clave = k then -- clave existe
      -- Este bucle se repite solo 1 o 2 veces
      from fin := False until fin loop
        if nodo.izdo = Void or nodo.dcho = Void then -- caso 0 o 1 hijo
          -- hijo del nodo borrado que le sustituye (puede estar vacío)
          if nodo.izdo = Void then hijo := nodo.dcho else hijo := nodo.izdo end
          if nodo = raiz then  -- El nodo borrado es el raiz y solo tiene un hijo
            raiz := hijo
          else
            hijo.cambia_padre(nodo.padre)
            -- Adaptar enlaces del padre
            if padre.izdo = nodo then
              nodo.padre.cambia_izdo(hijo) ; es_hijo_izdo := True
            else
              nodo.padre.cambia_dcho(hijo) ; es_hijo_izdo := False
            end
            comprobar_borrado(nodo.padre, es_hijo_izdo)
          end
          act := Void -- actual ya no es valido
          fin := True -- fin del bucle
        else -- caso 2 hijos
          -- Se busca el minimo del subarbol dcho
          hijo := minimo(nodo.dcho)
          -- El minimo toma el lugar del nodo a borrar
          nodo.cambia_clave(hijo.clave);
          -- En la siguiente iteracion se borra el nodo mínimo (sus datos ya estan salvados)
          nodo := hijo
        end -- deteccion de casos
      end -- bucle
    end -- clave existe
  end -- arbol no vacio
end -- quitar
```