Operaciones principales para TADs

		Conjunto	Colección	Lista indexada	Lista secuencial	Lista ordenada	Pila	Cola	Bicola	Cola prioridad	Tabla	Diccionario
Acceso	pertenencia	☑	·	·	·	·					☑	·
	recuento	⊘	☑	·	·	·					·	☑
	primero	⊘	⊘	·	·	⊘		☑	☑	⊘	⊘	⊘
	último	⊘	⊘	·	·	⊘	☑		☑	⊘	⊘	⊘
	posición	⊘	⊘	☑	·	⊘				⊘	⊘	⊘
	cursor	⊘	⊘	·	☑	·	⊘	⊘	⊘	⊘	☑	☑
	mínimo	·	·	·	·	·				☑	·	·
	i-ésimo menor	·	·	·	·	☑					☑	☑
Inserción	según valor/clave	☑	☑	⊘	⊘	☑	⊘	⊘	⊘	☑	☑	☑
	al principio	⊘	⊘	·	·	⊘			☑	⊘	⊘	⊘
	al final	⊘	⊘	·	·	⊘	☑	☑	☑	⊘	⊘	⊘
	en posición	⊘	⊘	☑	·	⊘				⊘	⊘	⊘
	delante/detrás cursor	⊘	⊘	·	☑	⊘	⊘	⊘	⊘	⊘	⊘	⊘
Borrado	por valor/clave	☑	☑	·	·	·					☑	☑
	primero	⊘	⊘	·	·	⊘		☑	☑	⊘	⊘	⊘
	último	⊘	⊘	·	·	⊘	☑		☑	⊘	⊘	⊘
	posición	⊘	⊘	☑	·	⊘				⊘	⊘	⊘
	cursor	⊘	⊘	·	☑	·	⊘	⊘	⊘		☑	☑
	mínimo	·	·	·	·	·				☑	·	·
	i-ésimo menor	·	·	·	·	☑					☑	☑
Otras	cursor al siguiente	⊘	⊘	·	☑	·	⊘	⊘	⊘		☑	☑
	cursor al anterior	⊘	⊘	·	☑	·	⊘	⊘	⊘		☑	☑
	concatenar/fusionar	☑	☑	☑	☑	☑				☑	☑	☑

☑ - Operación fundamental. • - Operación realizable mediante op. fundamentales
⊘ - Operación no aplicable. En blanco: Realizable pero mejor usar otro TAD.

Nota: En TADs ordenados (lista ordenada, cola de prioridad, tabla, diccionario) aparecen prohibidas las operaciones basadas en posición debido a que la posición de un elemento no se puede fijar externamente ya que depende únicamente de su valor.

Eficiencia según representación

		Contigua Lineal	Contigua Circular	Enlazada Lineal	Enlazada Circular	Contigua Ordenada	Montículo	Arbol bin. Búsqueda	Arbol AVL	Tabla Disp.
Acceso	pertenencia	O	O	O	O	■	O	O	■	·[4]
	recuento	●	●	●	●	■[2]	O	O	■[2]	⊘
	primero	·	·	·	·	·	⊘	O	■	⊘
	último	·	·	●	·	·	⊘	O	■	⊘
	posición *i*-ésima	·	·	O	O	·	⊘	O	■	⊘
	cursor	·	·	·	·	·	⊘	·	·	⊘
	mínimo	●	●	●	●	·	·	O	■	●
	i-ésimo menor	(†)	(†)	(†)	(†)	·	(§)	O	■[3]	⊘
Inserción	según valor/clave	⊘	⊘	⊘	⊘	O	■	O	■	·[5]
	al principio	●	·	·	·	⊘	⊘	⊘	⊘	⊘
	al final	·	·	●	·	⊘	⊘	⊘	⊘	⊘
	en posición *i*-ésima	O	O	O	O	⊘	⊘	⊘	⊘	⊘
	delante/detrás cursor	O	O	·[1]	·[1]	⊘	⊘	⊘	⊘	⊘
Borrado	por valor/clave	O	O	O	O	O	O	O	■	·[6]
	primero	●	·	·	·	●	⊘	O	■	⊘
	último	·	·	●	·[1]	·	⊘	O	■	⊘
	posición	O	O	O	O	O	⊘	O	■[3]	⊘
	cursor	O	O	·[1]	·[1]	O	⊘	O	■	⊘
	mínimo	●	●	●	●	●	■	O	■	●
	i-ésimo menor	(†)	(†)	(†)	(†)	O	(§)	O	■[3]	⊘
Otras	cursor al siguiente	·	·	·	·	·	⊘	O	■	⊘
	cursor al anterior	·	·	·[1]	·[1]	·	⊘	O	■	⊘
	concatenar/fusionar	●	●	●	·	●	●	OO	●■	●
	ordenar	(‡)	(‡)	(‡)	(‡)	⊘	⊘	⊘	⊘	⊘

[1] Se requiere lista doblemente enlazada o almacenar enlace anterior o posterior al cursor.
[2] Depende del número de elementos repetidos.
[3] Se requiere almacenar información de posición en los nodos del arbol.
[4] Siempre que la función de dispersión sea uniforme y el factor de carga sea adecuado.
[5] Si se cumple lo anterior y no se produce reestructuración (dispersión cerrada).
[6] Se supone estrategia perezosa.

· = $O(1)$	■ = $\Theta(\lg n)$	O = $O(n)$	● = $\Theta(n)$	●■ = $\Theta(n \lg n)$	OO = $\Theta(n^2)$

Algoritmos Especiales

- (†) Algoritmo de selección del *i*-ésimo menor:
  - Algoritmo de búsqueda de mínimos sucesivos: Orden $\Theta(i \cdot n)$.
  - Algoritmo basado en el método de partición: Caso promedio $\Theta(n)$ y peor caso $\Theta(n^2)$. Este algoritmo modifica la posición de elementos en el vector.
  - Existe una versión del algoritmo anterior (más compleja) que garantiza $\Theta(n)$ en el peor caso.
  - Estrategia divide y vencerás consistente en elegir un valor al azar y contar el número de elementos mayores y menores que él: Caso promedio $\Theta(n)$ y peor caso $\Theta(n^2)$. El vector permanece inalterado.
- (‡) Algoritmo de ordenación: En el caso de representación enlazada el algoritmo de fusión sigue teniendo un orden $\Theta(n \lg n)$ y el espacio adicional pasa a ser $\Theta(\lg n)$.
- (§) Para acceder, eliminar o modificar el elemento *i*-ésimo menor en montículos el algoritmo más sencillo de implementar es efectuar *i* borrados del mínimo y despues efectuar *i* inserciones de los elementos borrados (no se requieren estructuras adicionales). El orden sería de $\Theta(i \lg n)$.

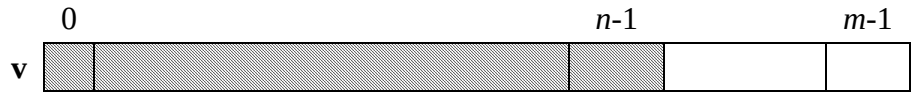
Otras Consideraciones

- En las operaciones de inserción, los ordenes de las representaciones contiguas son tiempos amortizados (puede ser necesario ampliar el vector).
- Los órdenes del arbol binario de búsqueda se refieren al peor caso. En el caso promedio sus órdenes son iguales a los del arbol AVL.

Representaciones Contiguas (I)

• Contigua lineal:

(*m* = capacidad del vector, *n* = número de elementos almacenados)



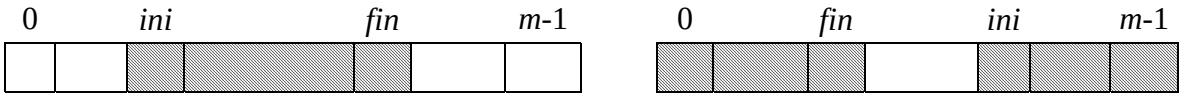
- Acceso a elemento posición *i* : v[i]
- Inserción de elemento *x* en posición *i* : ($0 \le i \le n$)

```
if n = m then ampliar(v);
for j := n-1 downto i do v[j+1] := v[j];
v[i] := x;
n := n+1
```

- Borrado de elemento en posición *i* : ($0 \le i < n$)

```
for j := i+1 to n-1 do v[j-1] := v[j];
n := n-1
```

• Contigua circular:



- Acceso a elemento posición *i*: v[(i+ini-1) **mod** m]
- Inserción de elemento *x* en posición *i* : ($0 \le i \le n$)

```
if n = m-1 then ampliar(v);
p = (ini+i) mod m; { posición i-ésimo }
j := fin; { se desplazan n-i elementos }
for k := 1 to n-i do { j va de fin a p }
begin
   v[(j+1) mod m] := v[j]; { v_{j+1} := v_j }
   j := (j+m-1) mod m { j := j-1 }
end;
v[p] := x;
fin := (fin+1) mod m; { fin := fin+1 }
n := n+1
```

Desplazamiento de la mitad derecha

```
if n = m-1 then ampliar(v);
p = (ini+i) mod m; { posición i-ésimo }
j := ini; { se desplazan i elementos }
for k := 1 to i do { j va de ini a p }
begin
   v[(j+m-1) mod m] := v[j]; { v_{j-1} := v_j }
   j := (j+1) mod m; { j := j+1 }
end;
v[p] := x;
ini := (ini+m-1) mod m; { ini := ini-1 }
n := n+1
```

Desplazamiento de la mitad izquierda

Representaciones Contiguas (II)

- Borrado de elemento en posición i : $(0 \le i < n)$

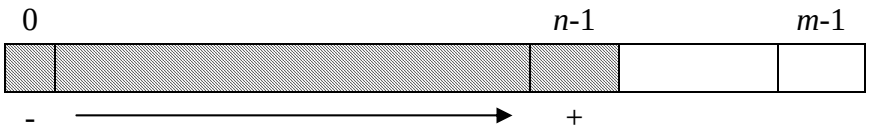
```
p = (i+ini) mod m; { posición i-ésimo }
j := (p+1) mod m ; {desplazar n-i+1 elem}
for k := 1 to n-i-1 do { j va de p+1 a fin }
begin
   v[(j+m-1) mod m] := v[j]; { v_{j-1} := v_j }
   j := (j+1) mod m; { j := j+1 }
end;
fin := (fin+m-1) mod m; { fin := fin-1 }
n := n-1
```

Desplazamiento de la mitad derecha

```
p = (i+ini) mod m; { posición i-ésimo }
j := (p+m-1) mod m; {desplazar i-1 elem}
for k := 1 to i-1 do { j va de p-1 a ini }
begin
   v[(j+1) mod m] := v[j]; { v_{j+1} := v_j }
   j := (j+m-1) mod m; { j := j-1 }
end;
ini := (ini+1) mod m; { ini := ini+1 }
n := n-1
```

Desplazamiento de la mitad izquierda

- **Contigua ordenada**:



- Acceso a elemento posición i (i-ésimo menor): v[i]

- Inserción de elemento x :

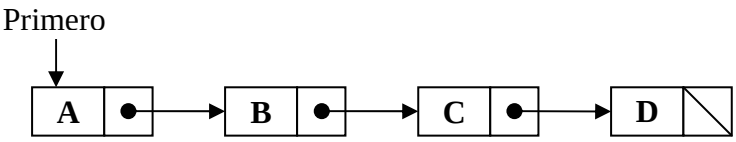
```
if n = m then ampliar(v);
{ Búsqueda binaria de posición donde insertar }
izda := 0; dcha := n-1;
while izda <= dcha do
begin
   { Invariante: v[0..izda-1] <= x, v[dcha+1..n-1] > x }
   med := (izda+dcha) div 2;
   if v[med] <= x then izda := med+1 else dcha := med-1
end;
{ Post: 0 <= izda <= n, v[0..izda-1] <= x < v[izda..n-1] }
for j := n-1 downto izda do v[j+1] := v[j];
v[izda] := x;
n := n+1
```

- Borrado de elemento en posición i : $(0 \le i < n)$

```
for j := i+1 to n-1 do v[i-1] := v[i];
n := n-1
```

Representaciones Enlazadas (I)

- Ejemplo:



- Enlaces mediante índices:

```
const N = …
type
   TNodo = record
      dato : char; sig : integer;
   end;
   TVector = array[1..N] of TNodo;
var
   V : TVector; primero : integer;
begin
   primero := 3;
   V[3].dato := 'A'; V[3].sig := 5;
   V[5].dato := 'B'; V[5].sig := 1;
   V[1].dato := 'C'; V[1].sig := 4;
   V[4].dato := 'D'; V[4].sig := -1;
end;
```

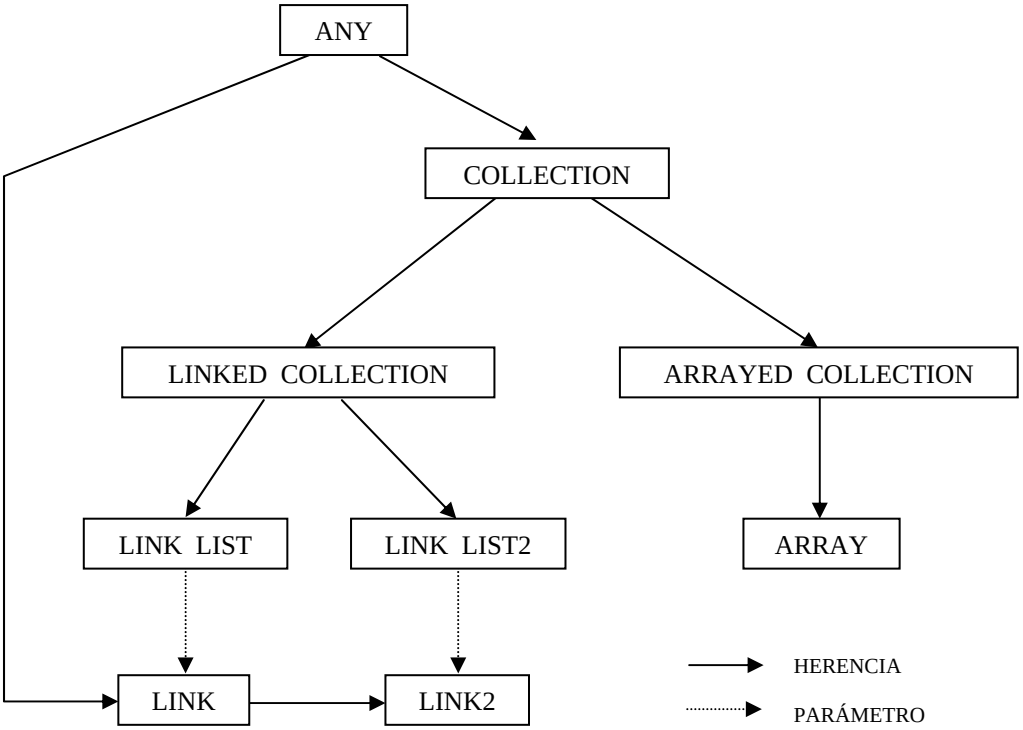
1	2	3	4	5	6
C		**A**	**D**	**B**	
4		**5**	**-1**	**1**	

- Enlaces mediante referencias a objetos:

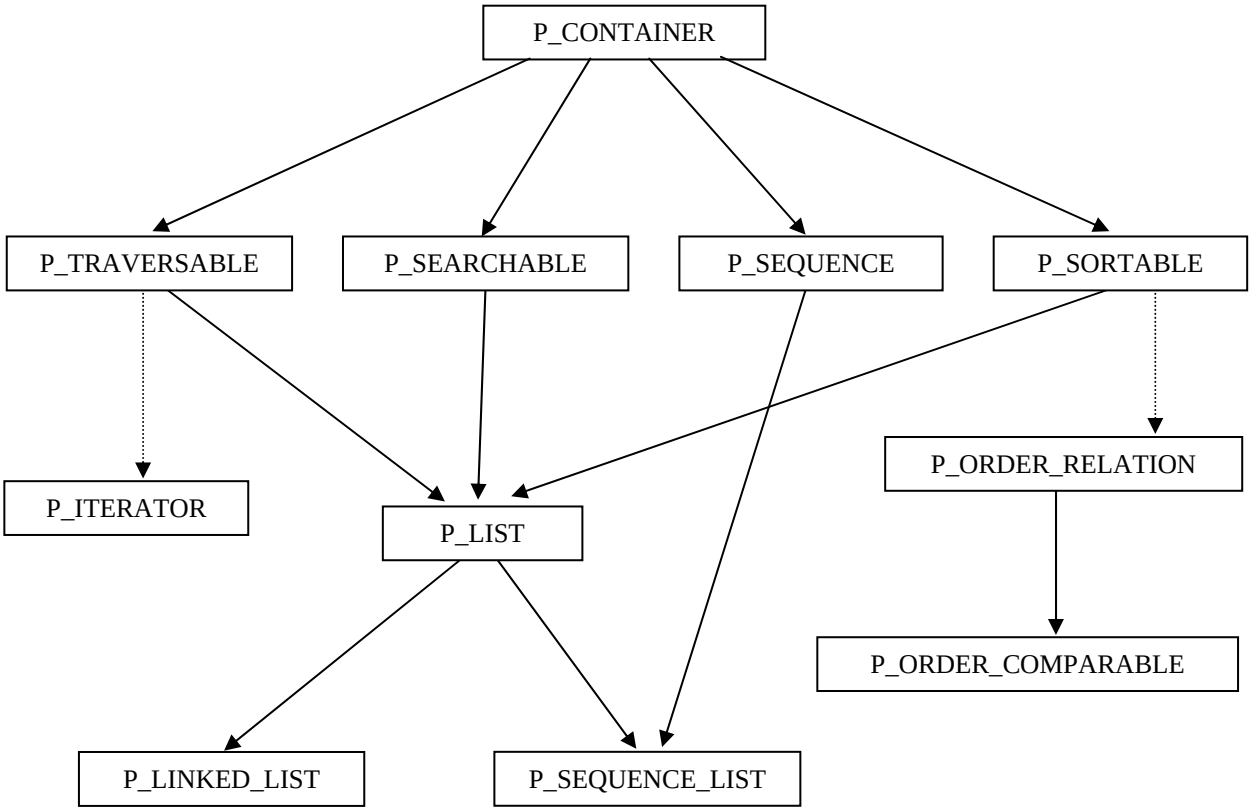
```
class NODO[E]
creation make
feature
   dato : E;
   sig : like Current;

   make(e: like elem; s: like sig) is
   do
      dato := e; sig := s;
   end;

end -- NODO
```

```
class EJEMPLO
creation make
feature
   primero, temp : NODO[CHARACTER];

   make is
   do
      create temp.make('D',Void);
      create temp.make('C',temp);
      create temp.make('B',temp);
      create temp.make('A',temp);
      primero = temp
   end
end -- EJEMPLO
```

Listas en SmallEiffel



Listas en Pylon



Ejemplo de implementación de lista basada en cursor

```
indexing
  "Nodo simplemente enlazado que"
  "almacena un valor de tipo ELEMENTO"
class NODO[ELEMENTO]
creation make
feature { LISTA_CUR }
    elem : ELEMENTO;
    sig : like Current;

    make(e: like elem; s: like sig) is
    do
        elem := e; sig := s
    end

    cambia_sig(s: like sig) is
    do
        sig := s
    end
end -- class NODO
```

```
indexing
    descripcion : "Definición del TAD LISTA_CUR."
                  "Clase abstracta salvo la operación concatenar"
deferred class LISTA_CUR[ELEMENTO]
feature { ANY }

    crear_lc is deferred
    ensure -- Vacía la lista (cursor fuera de lista)
    end -- crear_lc

    fin_de_lista : BOOLEAN is deferred
    ensure  -- True si el cursor está situado despues del último elemento
            -- (fuera de lista) o la lista está vacía. False en caso contrario.
    end -- fin_de_lista

    actual : ELEMENTO is
    require not fin_de_lista
    deferred
    ensure  -- Devuelve el elemento sobre el que está situado el cursor
    end -- actual

    ir_a_inicial is deferred
    ensure -- El cursor se situa sobre el primer elemento. Si la lista está
           -- vacía el cursor sigue fuera de lista.
    end -- ir_a_inicial

    ir_a_siguiente is deferred
    ensure -- El cursor pasa al siguiente elemento. Si el cursor estaba
           -- sobre el último elemento o fuera de lista pasa a estar
           -- fuera de lista.
end -- ir_a_siguiente

    insertar(elem: ELEMENTO) is deferred
    ensure -- inserta el elemento antes del actual. El cursor no cambia. Si
           -- el cursor estaba fuera de lista se inserta el elemento al final
           -- y el cursor sigue fuera de lista.
    end -- insertar

    cambiar(elem: ELEMENTO) is
    require not fin_de_lista
    deferred
    ensure -- cambia el elemento actual y el cursor pasa al siguiente
    end -- cambiar

    quitar is
    require not fin_de_lista
    deferred
    ensure -- borra elemento actual, cursor pasa a siguiente
    end -- quitar

    -- Concatena la otra lista al final de ésta. El cursor de ambas
    -- listas pasa a estar fuera de lista. No se modifica el contenido
    -- de la otra lista (se comparten referencias a elementos).
    concatenar(otra : like Current) is
    do
      -- situar cursor fuera de lista
      from until fin_de_lista loop
        ir_a_siguiente
      end
      -- recorrer la otra lista insertando sus elementos
      from otra.ir_a_inicio until otra.fin_de_lista loop
          insertar(otra.actual)
          otra.ir_a_siguiente
      end
    end -- concatenar
end -- class LISTA_CUR
```

Implementaciónes contigua y enlazada

```
indexing
    descripcion : "Implementación de la clase"
        "abstracta (TAD) LISTA_CUR"
    representacion :  "Contigua lineal con un índice al"
        "elemento sobre  el que está situado el cursor"
    eficiencia : "Inserción peor caso O(n), amortizado"
        "O(n-i), borrado orden O(n-i), concatenación"
        "O(n+m), resto operaciones O(1). n es el"
        "número de elementos, i la posición del cursor"
        "m nº de elementos de la lista que se concatena"
class LISTA_CUR_CL[ELEMENTO]
inherit LISTA_CUR[ELEMENTO]
creation crear_lc
feature { LISTA_CUR_CL }
    vec : ARRAY[ELEMENTO]
    num : INTEGER -- Numero de elementos
    act : INTEGER -- Indice del elemento actual (cursor)
feature { ANY }
    crear_lc is
    do
      create vec.make(1,100)
      num := 0 ; act := 1
    end -- crear_lc
    fin_de_lista : BOOLEAN is
    do
      Result := act > num
    end -- fin_de_lista
    actual : ELEMENTO is
    do
      Result := vec @ act
    end -- actual
    ir_a_inicial is
    do
      act := 1
    end -- ir_a_inicial
    ir_a_siguiente is
    do
      act := act+1
    end -- ir_a_siguiente

    -- Continua en la siguiente página
```

```
indexing
    descripcion : "Implementación de la clase"
        "abstracta (TAD) LISTA_CUR"
    representacion : "Enlazada simple circular con referencia"
        "adicional al elemento ANTERIOR al cursor y una"
        "indicación de cursor fuera de lista. Los elementos se"
        "almacenan en objetos cuya clase es NODO"
    eficiencia : "Todas las operaciones son de orden O(1)."
class LISTA_CUR_ESC[ELEMENTO]
inherit
    LISTA_CUR[ELEMENTO]
        redefine concatenar
    end
creation crear_lc
feature { LISTA_CUR_ESC }
    ult : NODO[ELEMENTO] -- Ultimo nodo de la lista
    ant : NODO[ELEMENTO] -- Nodo anterior al nodo actual
    -- Cuando la referencia al anterior apunta al último existe
    -- la doble posibilidad de que el cursor sea el primero
    -- (en una lista circular el primero es el siguiente al
    -- último) o bien que el cursor esté fuera de la lista
    -- (despues del último, esto es necesario para poder
    -- insertar un elemento al final).
    -- Esta variable sirve para indicar cual de esas dos
    -- interpretaciones es la adecuada si ant = ult.
    fin : BOOLEAN
feature { ANY }
    crear_lc is
    do
      ult := Void ; ant := Void
      -- Si la lista está vacía el cursor está fuera de lista
      fin := True
    end -- crear_lc
    fin_de_lista : BOOLEAN is
    do
      Result := fin
    end -- fin_de_lista
    actual : ELEMENTO is
    do
      Result := ant.sig.elem
    end -- actual
    ir_a_inicial is
    do
      ant := ult
      -- Si la lista está vacía el cursor está fuera de lista
      fin := (ult = Void)
    end -- ir_a_inicial
    ir_a_siguiente is
    do
      -- Si la lista vacía o cursor fuera de lista no hace nada
      if ult /= Void and not fin_de_lista then
        ant := ant.sig
        -- Si el cursor estaba en el último elemento pasa a
        -- estar fuera de lista
        fin := (ant = ult)
      end
    end -- ir_a_siguiente

    -- Continua en la siguiente página
```

Implementaciones (continuación)

```
insertar(elem: ELEMENTO) is
    local i : INTEGER
    do
      -- comprobar capacidad del vector
      if num >= vec.upper then
        vec.resize(1,2*vec.upper)
      end
      -- desplazar actual y posteriores a la derecha
      from i := num until i < act loop
        vec.put(vec @ i, i+1)
        i := i-1
      end
      -- insertar elemento, actualizar actual y número
      vec.put(elem, act)
      act := act+1 -- actual debe ser el mismo
      num := num+1
    end -- insertar

    cambiar(elem: ELEMENTO) is
    do
      vec.put(elem, act)
      act := act+1 -– se pasa al siguiente
    end -- cambiar

    quitar is
    local i : INTEGER
    do
      -- desplazar posteriores al actual a la izquierda
      from i := act+1 until i > num loop
        vec.put(vec @ i, i-1)
        i := i+1
      end
      num := num-1
    end -- quitar

    -- concatenar tal como se define en LISTA_CUR
end -- class LISTA_CUR_CL
```

```
    insertar(elem: ELEMENTO) is
    local nuevo : NODO[ELEMENTO] -- nuevo nodo
    do
      create nuevo.make(elem, Void);
      if ult = Void then -- caso lista vacía
        nuevo.cambia_sig(nuevo) -- nuevo es primero y último
        ult := nuevo ; ant := nuevo
        fin := True -- cursor fuera de lista en este caso
      else
        nuevo.cambia_sig(ant.sig)
        ant.cambia_sig(nuevo)
        -- si se inserta al final se debe adaptar ult y ant
        if fin_de_lista then ult := nuevo; ant := nuevo end
      end
    end -- insertar

    cambiar(elem: ELEMENTO) is
    do
      ant.sig.cambia_elem(elem)
      ir_a_siguiente -– se pasa al siguiente
    end -- cambiar

    quitar is
    do
      if ant.sig = ult then -- borrado del último
        ult := ant
      end
      ant.cambia_sig(ant.sig.sig)
    end -- quitar

    -- Versión especial de concatenar adaptada a esta
    -- implementación. La otra lista se vacía (para evitar el
    -- riesgo de tener estructuras con nodos comunes)
    concatenar(otra: like Current) is
    require
      otra /= Current  -- No se puede concatenar una lista
                       -- consigo misma
    local primero : NODO[ELEMENTO]
    do
      if otra.ult /= Void then -- la otra lista no está vacía
        if ult = Void then -- esta lista vacía
          ult := otra.ult
        else -- ninguna lista vacía
          primero := ult.sig
          ult.cambia_sig(otra.ult.sig)
          otra.ult.cambia_sig(primero)
          ult := otra.ult
        end
      end
      -- hacer que el cursor este fuera de la lista
      ant := ult
      fin := True
      -- vaciar la otra lista
      otra.crear_lc
    end -- concatenar

  invariant
    -- invariante de clase
    fin_de_lista_consistente: (fin = True) implies (ant = ult)
end -- class LISTA_CUR_ESC
```

Ejemplo de implementación de Pila y Cola

```
indexing
    descripcion : "Implementación del TAD PILA"
    representacion : "Contigua circular"
    eficiencia : "Todas las operaciones O(1) (tpo amort)"
class PILA[ELEM]
creation crear_pila
feature { NONE }
    vec : ARRAY[ELEMENTO];
    tam,num : INTEGER; -- capacidad y numero de elem.

feature { ANY }
    crear_pila is
    do
      tam := 100; num := 0;
      create vec.make(1,tam);
    end -- crear_pila

    vacia : BOOLEAN is
    do
      Result := (num = 0)
    end -- vacia

    cabeza : ELEM is
    require not vacia
    do
      Result := vec @ num
    end -- cabeza

    -- insertar por la cabeza
    insertar(e: ELEM) is
    local i, j : INTEGER
    do
      num := num+1
      if num >= tam then -- Ampliar el vector
        vec.resize(1,2*tam)
      end
      -- insertar al final
      vec.put(e , num)
    end -- insertar

    -- quitar elemento cabeza
    quitar is
    require not vacia
    do
      num := num-1
    end – quitar
end -- class PILA
```

```
indexing
    descripcion : "Implementación del TAD COLA"
    representacion : "Contigua circular"
    eficiencia : "Todas las operaciones O(1) (tpo amortizado)"
class COLA[ELEM]
creation crear_cola
feature { NONE }
    vec : ARRAY[ELEMENTO];
    tam,num : INTEGER; -- capacidad y numero de elementos
    ini,fin : INTEGER; -- límites de zona ocupada

feature { ANY }

    crear_cola is
    do
      tam := 100; num := 0; ini := 0; fin := -1;
      create vec.make(0,tam-1);
    end -- crear_cola

    vacia : BOOLEAN is
    do
      Result := (num = 0)
    end -- vacia

    cabeza : ELEM is
    require not vacia
    do
      Result := vec @ ini
    end -- cabeza

    -- insertar por la cola
    insertar(e: ELEM) is
    local i, j : INTEGER
    do
      num := num+1
      if num >= tam then -- Ampliar el vector
        vec.resize(0,2*tam-1)
        if ini > fin then -- Reorganizar vector
          from i := tam-1; j := 2*tam-1 until i < ini loop
            vec.put(vec @ i , j)
            i := i-1; j := j-1
          end
          ini := j+1
        end
        tam := 2*tam-1
      end
      fin := (fin+1) \\ tam;
      vec.put(e , fin)
    end -- insertar

    -- quitar elemento cabeza
    quitar is
    require not vacia
    do
      num := num-1
      ini := (ini+1) \\ tam
    end -- quitar
end -- class COLA
```

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.

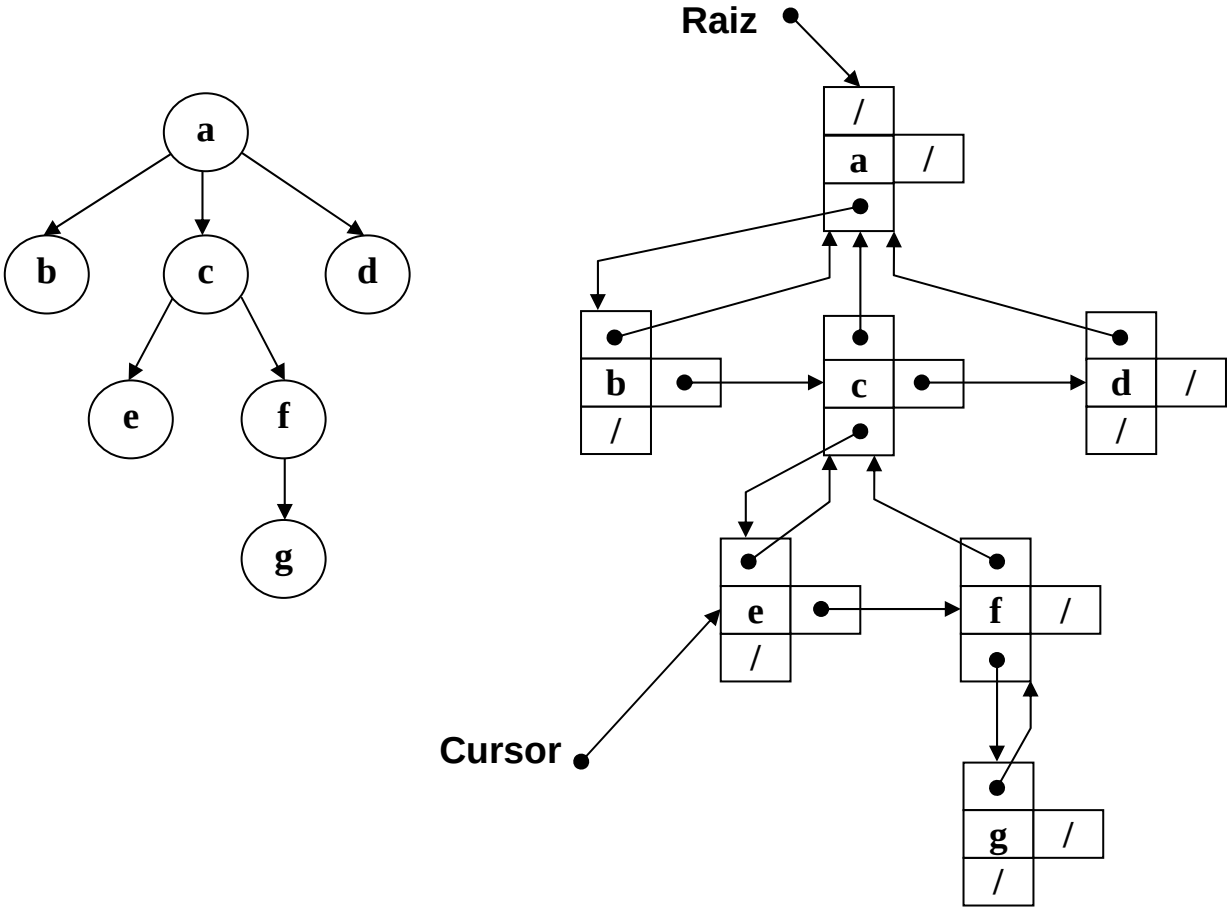
- La **profundidad de un nodo** se define como la longitud del camino (único) que comienza en la raiz y termina en el nodo. La profundidad de la raiz es cero, y la profundidad de un nodo se puede calcular como la profundidad de su padre mas uno. A la profundidad de un nodo también se la denomina **nivel**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

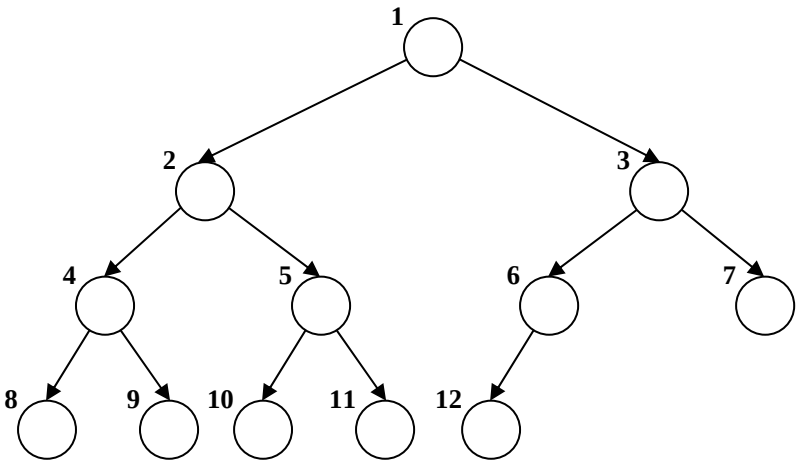
- Cada nodo de un árbol binario puede tener ningún hijo (subárbol izquierdo y derecho vacíos), un hijo (subarbol izquierdo **o** derecho vacío) o dos hijos. Dependiendo de si son la raiz del subarbol izquierdo o derecho se denominan **hijo izquierdo** e **hijo derecho**.

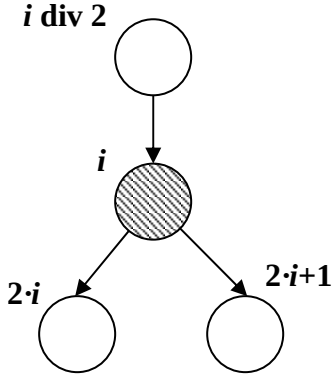
- **Árbol binario estricto**: No se permite que un subárbol esté vacío y el otro no lo esté. Por lo tanto cada nodo puede tener cero o dos hijos.

- **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.

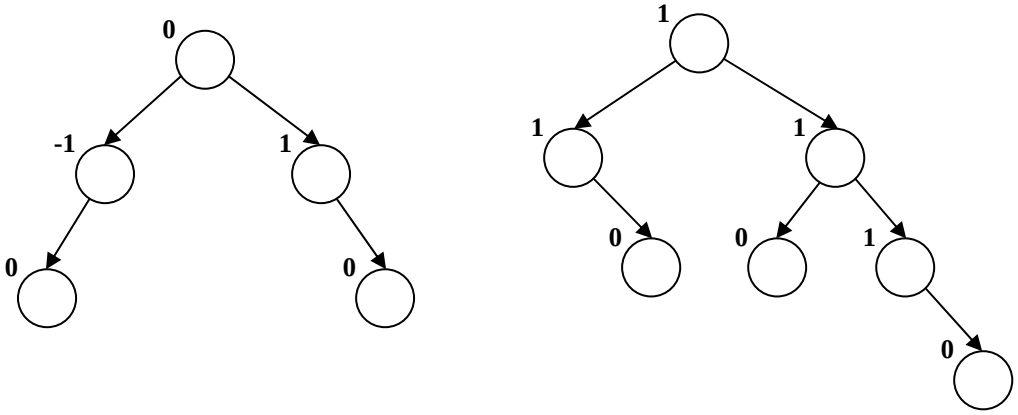
  La definición anterior es de un montículo cuya raiz es el elemento mínimo. Alternativamente, podemos definir un montículo cuya raiz sea el elemento máximo con sólo cambiar la palabra *menor* por *mayor*.

  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 }
```

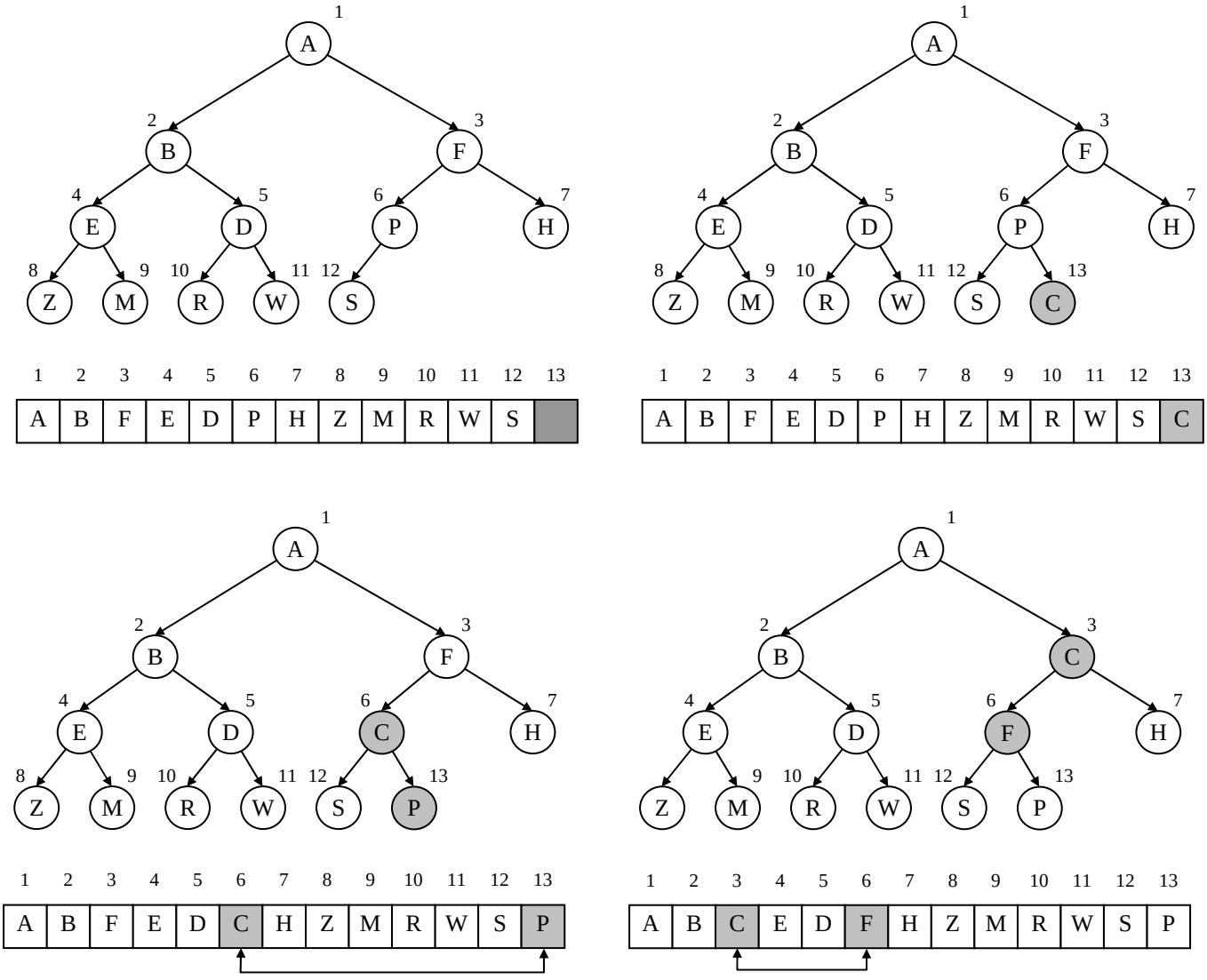
Reestructuración de un (sub)montículo:

```
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

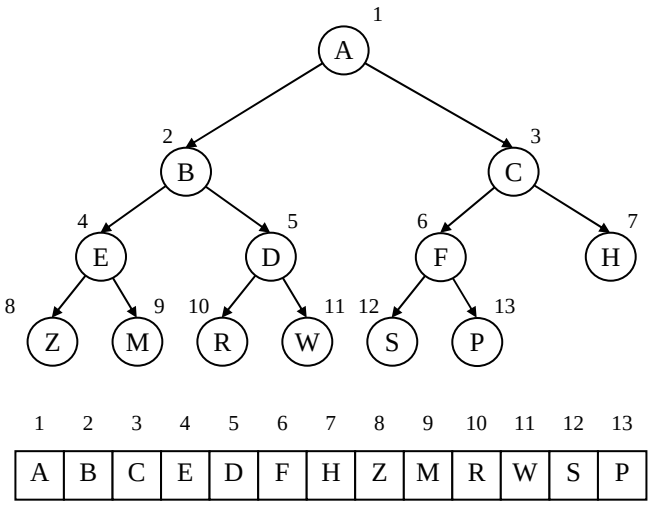
Inserción de un elemento:

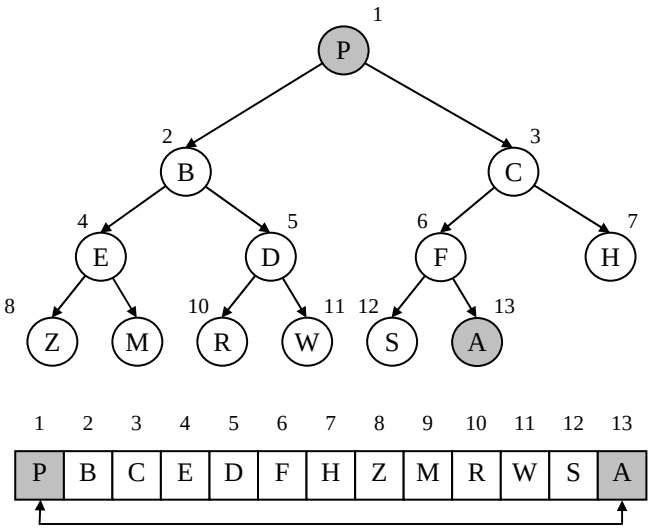
```
procedure Insertar(var M: TMonticulo ; E: TElemento) ;
{ Inserta el elemento E en el montículo M de manera que se siga manteniendo la estructura de
  montículo. Eficiencia: O(lg n) }
begin
   M.Num := M.Num+1 ;
   if M.Num > MAX then { ampliar capacidad de M.Vec } ;
   M.Vec[M.Num] := E;  { Inserción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

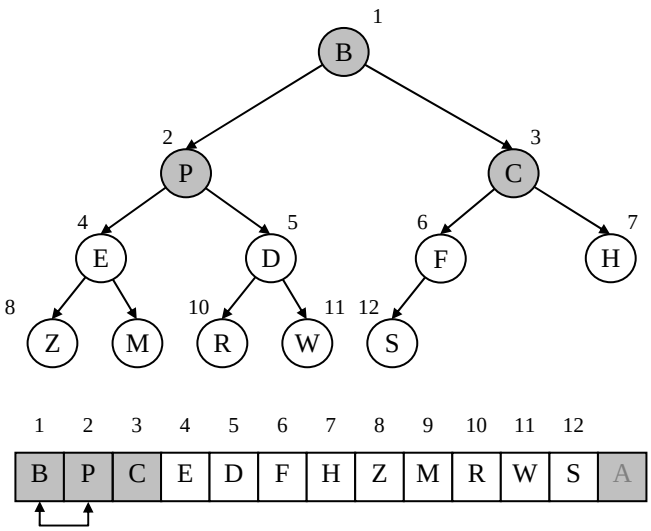


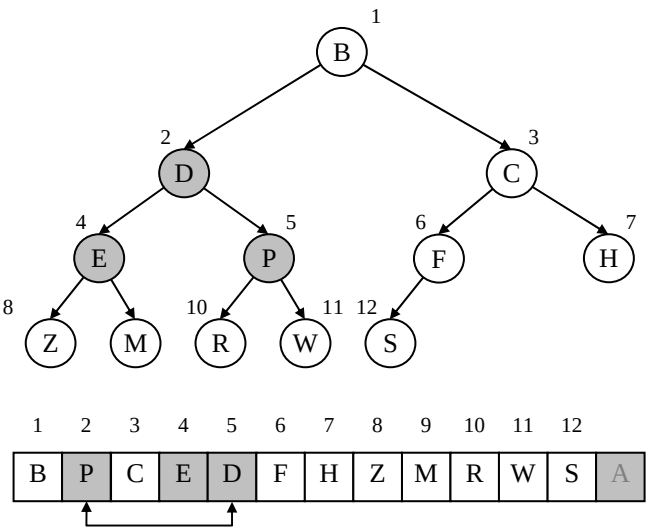
Borrado del nodo raiz (elemento con clave mínima):

```
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)

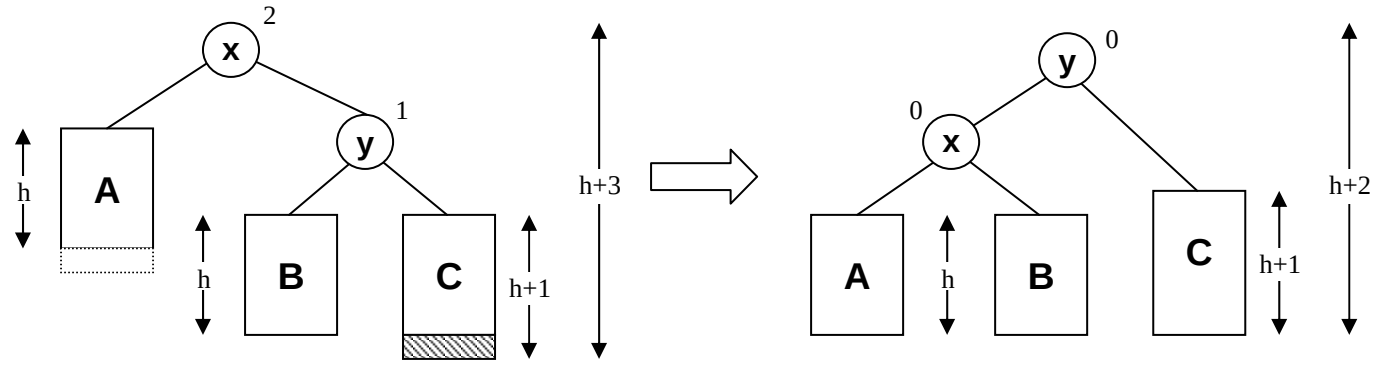
- Operaciones de recorrido basado en cursor:

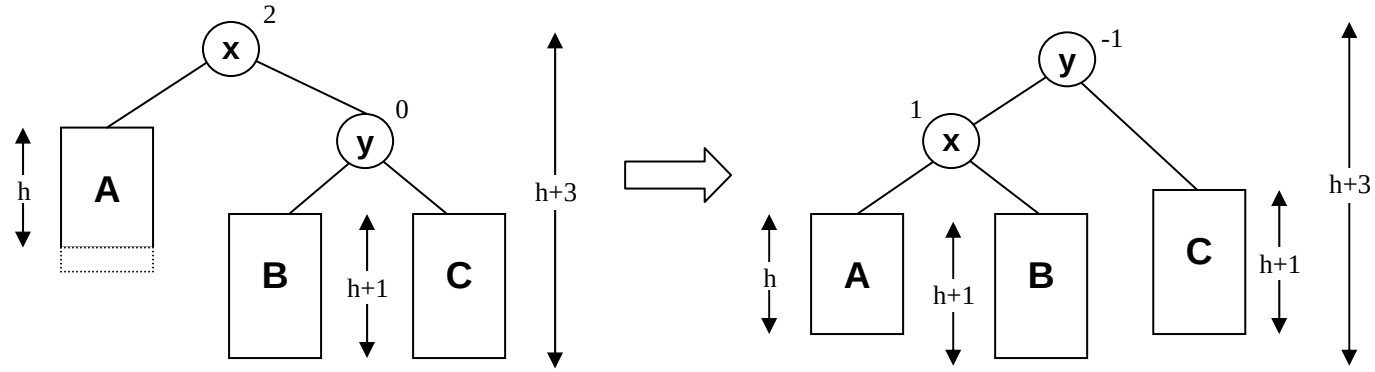
```
-- Nota: Al insertar, borrar, o ir al siguiente del último el cursor pasa a no ser válido
actual_valido : BOOLEAN is
do
  Result := act /= Void
end -- actual_valido

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
```

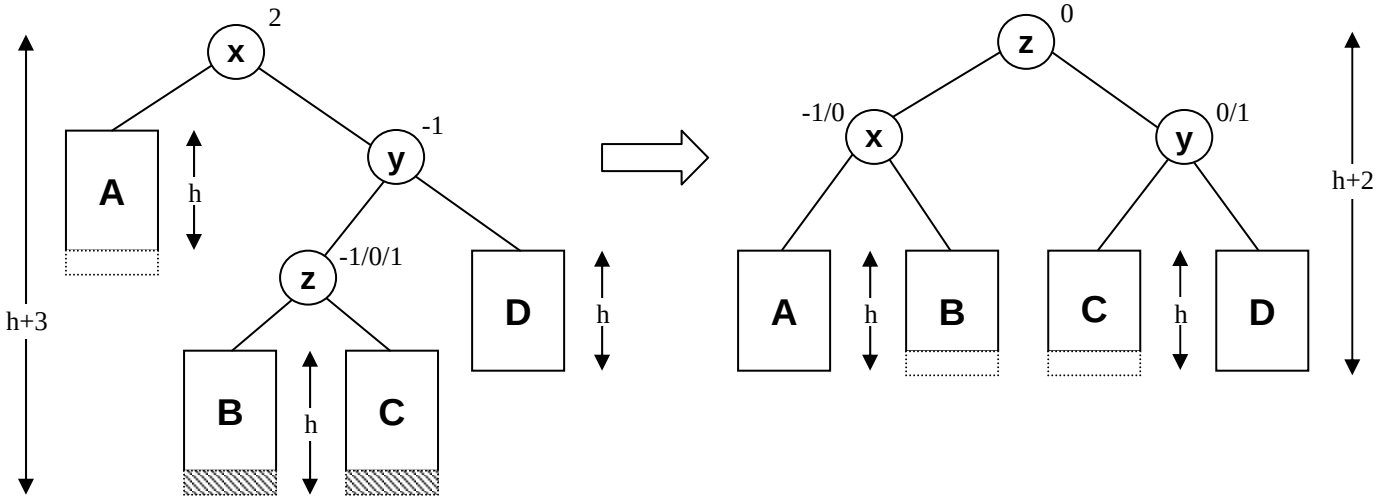
- Invariantes de clase (arbol correcto):

```
-- 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
```

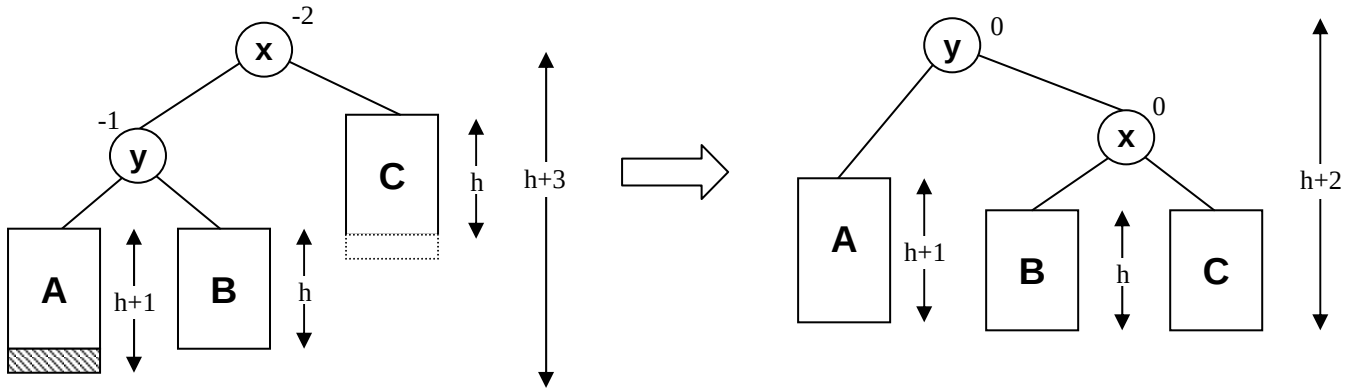
Arboles AVL - Rotaciones

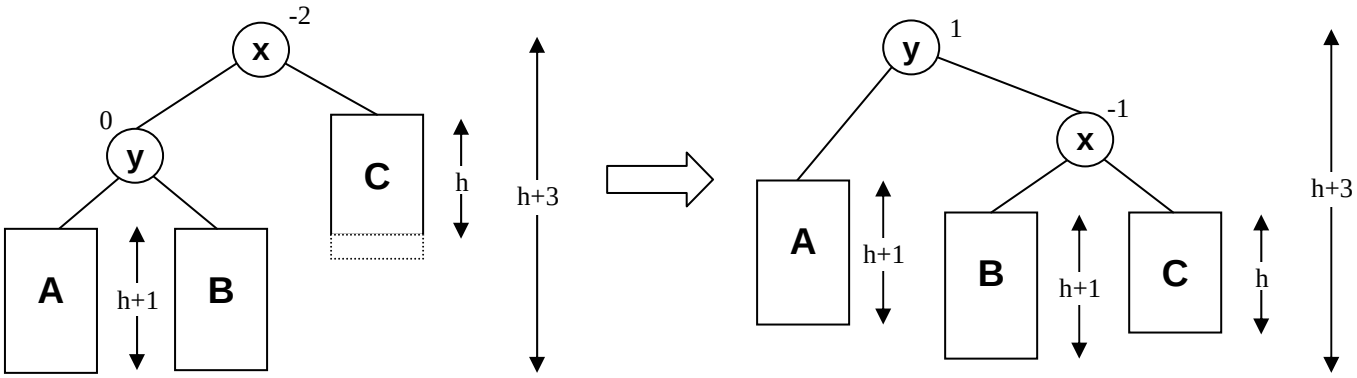


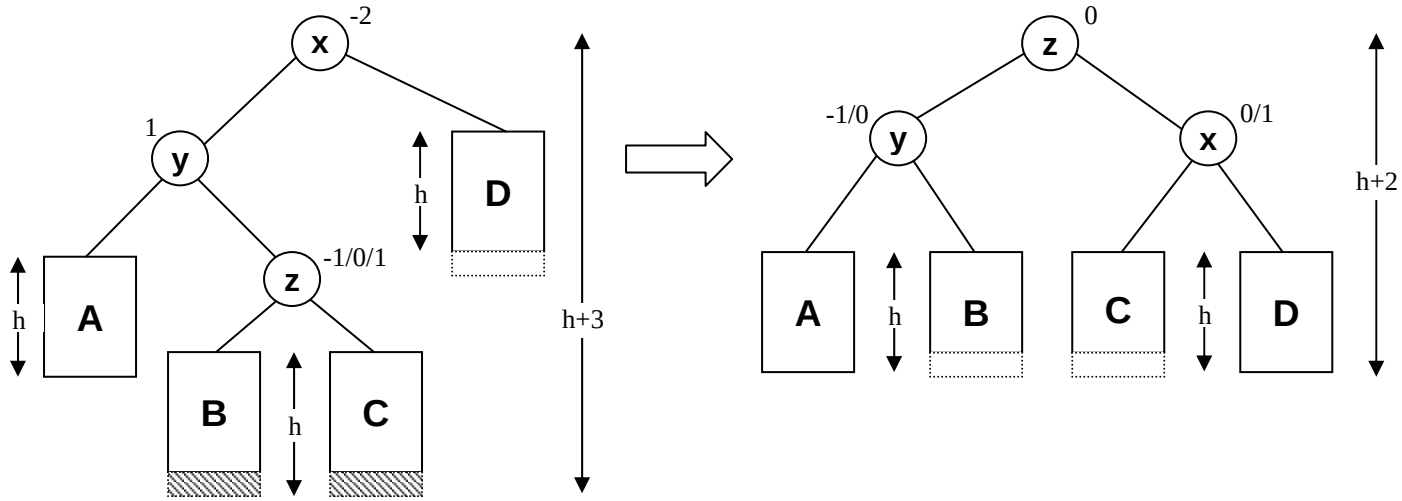




Arboles AVL – Rotaciones simétricas







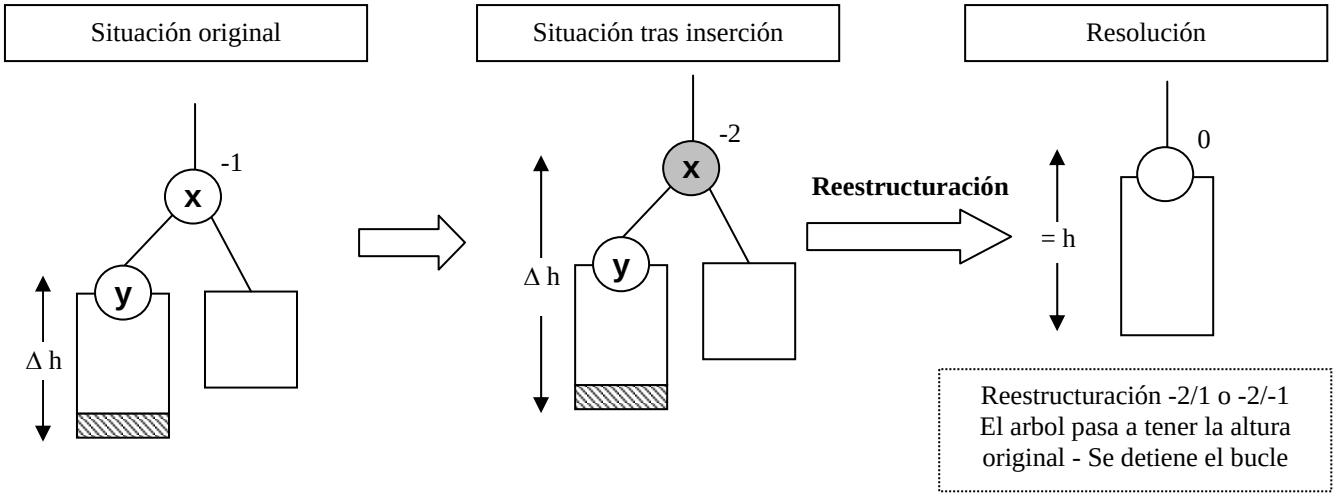
Implementación de un Arbol AVL (I)

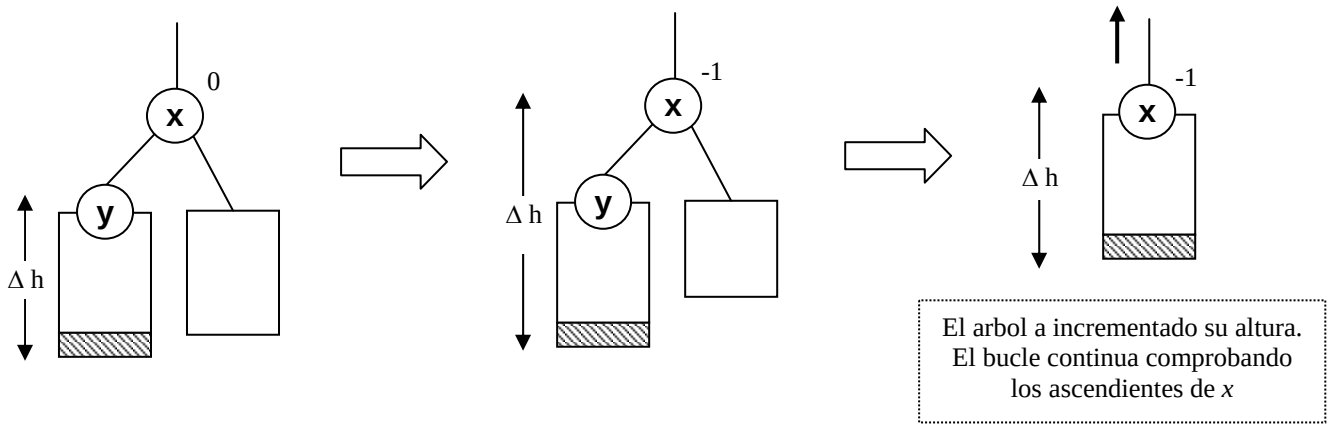
- Definición de la clase y rotaciones:

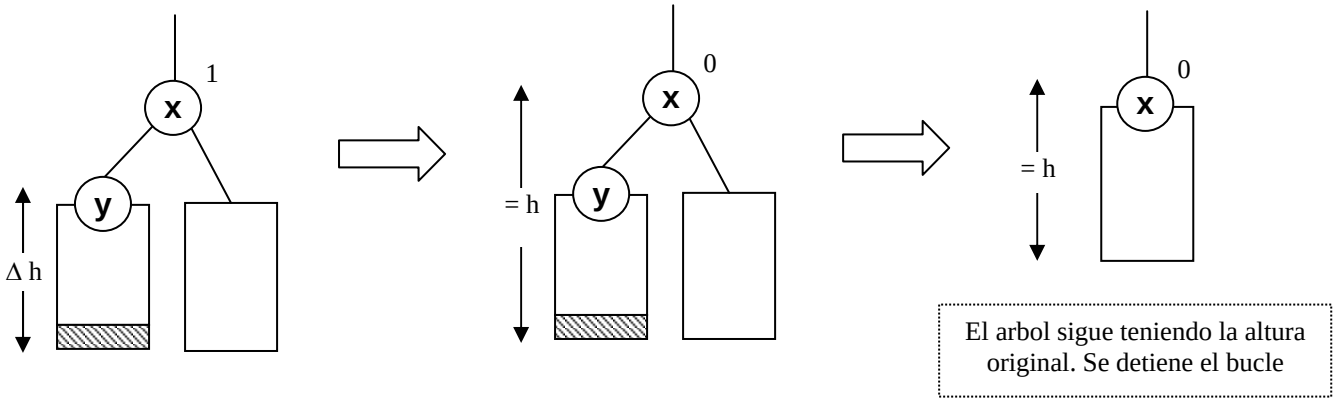
```
class ARBOL_AVL[ELEMENTO -> COMPARABLE]
inherit ARBOL_ABB[ELEMENTO]
  redefine insertar, quitar
end
creation crea_arbol
feature { NONE }

    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_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_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_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

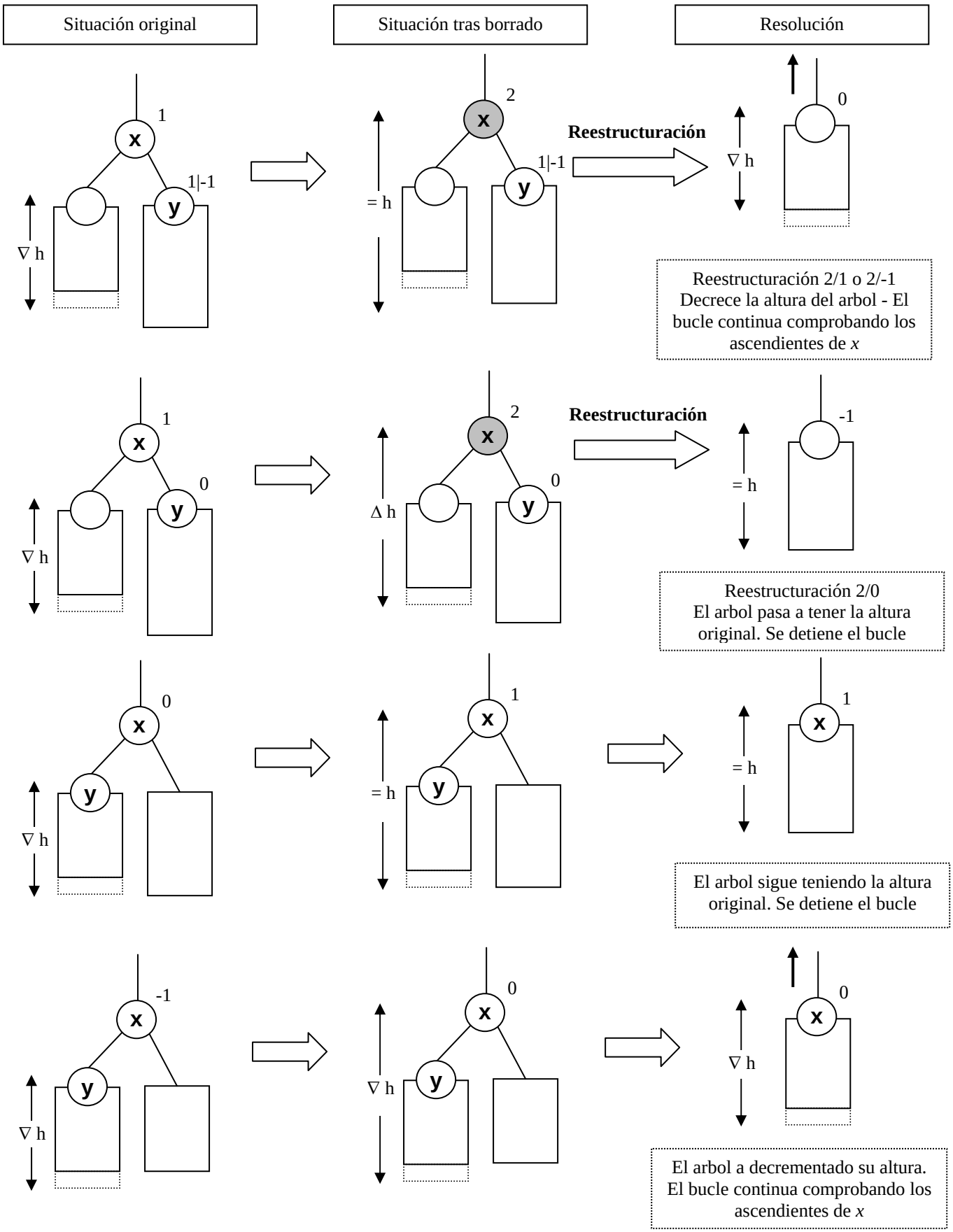
La comprobación consiste en un bucle donde se analiza el cambio sufrido por un nodo, *x*, en el que uno de sus subarboles a incrementado su altura en 1 debido a una inserción. Existen 6 posibilidades (3 casos cuando el subarbol que cambia es el izquierdo y otros 3 cuando es el derecho), dependiendo del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

Tablas de Dispersión

- Representación de datos especialmente diseñada para que las operaciones de acceso, inserción y borrado por valor o campo clave sean eficientes (tiempo promedio constante, independiente del número de elementos).

- Una primera aproximación es utilizar la clave (k) como índice de un vector que contiene referencias a elementos. Este enfoque se denomina **vector asociativo (lookup array)**. Problemas:

  - No todos los tipos de clave sirven de índice de vectores (por ejemplo, cadenas de caracteres, números reales).

  - El tamaño del vector (m) es igual al del rango de la clave (número de posibles valores distintos que pueda tener). Este tamaño es independiente del número de elementos almacenados (n), y puede ser muy grande.

- Las **tablas de dispersión** resuelven el problema definiendo una **función de dispersión** que traduzca la clave a un valor numérico que luego se *reduce* al rango 0..m-1. Los elementos se almacenan en una tabla cuyo tamaño (m) se elige con unos determinados criterios. m no tiene porqué ser igual a n (pueden existir posiciones vacías o varios elementos en cada posición). El método más común para reducir el valor numérico obtenido de la clave al rango 0..m-1 (índice a la tabla) es hallar el resto de su división por m.

  - Si m es el tamaño elegido para la tabla y $h(k)$ la función de dispersión, un elemento cuya clave sea k se almacenará en la posición i de la tabla, calculada de esta forma:

$$i = h(k) \textbf{ mod } m$$

  - La función de dispersión se diseña teniendo en cuenta el tipo de datos de la clave y otras características (uniformidad sobre el conjunto de datos). Si la clave es de tipo entero, la función de dispersión más sencilla es devolver el propio valor de la clave:

$$h(k) = k$$

  - Si la clave es una cadena de b caracteres, podemos tratarla como una secuencia de enteros $k \equiv k_0 \ldots k_{b-1}$ (tomando como el valor asociado a cada carácter su posición en la tabla de códigos). Una función de dispersión muy utilizada es la siguiente:

$$h(k) = \sum k_i \cdot 31^i \quad (i = 0 \ldots b\text{-}1)$$

Tablas de Dispersión Abierta

- El enfoque anterior sufre del problema de las **colisiones**: La función de dispersión puede asignar el mismo índice a claves distintas.

$$h(k_1) \textbf{ mod } m = h(k_2) \textbf{ mod } m \ , \ \ k_1 \neq k_2$$

- Las tablas de dispersión **abierta** utilizan como estrategia de resolución del problema de las colisiones el permitir que varios elementos se encuentren almacenados en la misma posición de la tabla: Es decir, el contenido de la tabla no son elementos, sino listas de elementos.

- Las listas suelen implementarse mediante representación enlazada, con enlaces simples y sin mantener un orden entre los elementos.

- Se define el **factor de carga** (L) de la tabla como el valor $L = n/m$, donde n es el número de elementos almacenados. L representa el tamaño promedio de las listas. Si la función de dispersión se comporta de manera **uniforme** para el conjunto de datos utilizado, la mayoría de las listas tendrán un tamaño cercano al valor de L.

- Algoritmos de las operaciones de acceso, inserción y borrado:

Acceso	**Inserción**	**Borrado**
$i \leftarrow h(k)$ **mod** m *lista* $\leftarrow$ *tabla*[i] *Búscar secuencialmente en la lista un elemento cuya clave sea k* *Devolver resultado de la búsqueda*	$i \leftarrow h(k)$ **mod** m *lista* $\leftarrow$ *tabla*[i] *Insertar el elemento al principio de la lista.*	$i \leftarrow h(k)$ **mod** m *lista* $\leftarrow$ *tabla*[i] *Búscar secuencialmente en la lista un elemento cuya clave sea k* *Borrar elemento de la lista*
$O(L)$	$O(1)$	$O(L)$

Nota: La tabla se define como un vector de listas indexado de 0 a m-1.

Propiedades de la dispersión abierta

- El número de elementos almacenados puede ser mayor que el tamaño de la tabla (puede darse que $n > m$). No es estrictamente necesario realizar operaciones de reestructuración (cambio de tamaño) de la tabla, salvo para garantizar que el coste de las operaciones de acceso y borrado (que es lineal con L) sea constante.

- Las operaciones de acceso y borrado se convierten en operaciones de búsqueda y borrado en listas: La eficiencia depende del tamaño de las listas, para que se considere que el tiempo es constante se debe cumplir:

  - El tamaño de las listas debe ser más o menos uniforme: No debe darse el problema de que unas pocas listas contengan la mayoría de los elementos (problema del **agrupamiento primario**). Para conseguirlo se debe cumplir que la función de dispersión sea **uniforme** para los conjuntos de datos que se van a utilizar.

  - Si se cumple la condición anterior, la mayoría de las listas tendra un tamaño que no se desviará mucho del valor promedio $L = n/m$ (**factor de carga**). No se debe permitir que L sea muy grande, por lo que el tamaño de la tabla debe ser del mismo orden que el máximo número de elementos que se van a almacenar.

Ejemplo de agrupamiento primario

- Se almacenan datos de 177 personas en un tabla de tamaño m = 150 y usando como clave el DNI. El factor de carga es de L = 1.18. Se utilizan dos funciones de dispersión, la primera es uniforme y la segunda no:

- $h(dni) = dni$: (Nº promedio accesos = 1.6)

Tamaño de la lista	**0**	**1**	**2**	**3**	**4**	**5**
Nº de listas de ese tamaño	52	43	38	11	5	1
Porcentaje de elementos que están en listas de ese tamaño		24 %	43 %	19 %	11 %	3 %

- $h(dni) = dni$ **div** 100000: (Nº promedio accesos = 15.9)

Tamaño de la lista	**0**	**1**	**2**	**5-9**	**14-18**	**65**
Nº de listas de ese tamaño	125	11	5	4	4	1
Porcentaje de elementos que están en listas de ese tamaño		6 %	6 %	14 %	37 %	37 %

Tablas de Dispersión Cerrada

- Las tablas de dispersión **cerrada** utilizan como estrategia de resolución del problema de colisiones el asignar otra posición en la tabla al elemento cuya posición está ocupada.

- Se define una función adicional, la **función de exploración**, que calcula una nueva posición para un elemento a partir de su posición inicial y del número de intentos de realojamientos (nº de colisiones sufridas) en el proceso de hallar una posición vacía.

- El contenido de las tablas de dispersión cerrada son referencias a elementos: A diferencia de la dispersión abierta, sólo se puede almacenar un elemento (o ninguno) en cada celda.

- Cuando se busca un elemento en la tabla se sigue el mismo camino de exploración que se ha seguido en la inserción. La aparición de una posición vacía indica que no existe el elemento en la tabla, ya que en caso contrario se hubiera insertado en esa posición.

- La estrategia anterior implica que no se debe permitir que la tabla esté completamente llena, ya que impediría detectar que un elemento no existe. Por lo tanto se debe exigir que $n < m$. Si al insertar un elemento se llena la tabla se debe **reestructurar** (crear una nueva tabla de tamaño mayor e insertar todos los elementos en la nueva tabla).

- Además se plantea el problema de que borrar un elemento cambiando su posición en la tabla a vacía puede impedir el hallar otros elementos que sufrieron una colisión en esa posición, ya que aparece una posición vacía en su ruta de exploración.

- La solución más utilizada es la estrategia **perezosa** de borrado: Los elementos no se borran marcando su posición como **vacía**, sino que se marca es posición como **borrada**. Una casilla borrada se puede usar para insertar un elemento (al igual que una posición vacía), pero no indica el final de una exploración (a diferencia de una posición vacía).

Implementación de una tabla de dispersión cerrada

```
class TABLA_DISP[ELEM <- HASHABLE]
-- Almacena elementos no repetidos
creation crear_tabla
feature { NONE }
  claves  : ARRAY[ELEM];        -- Tabla de elementos
  vacia   : ARRAY[BOOLEAN];     -- Tabla que indica si una posición está vacía
  borrada : ARRAY[BOOLEAN];     -- Tabla que indica si una posición está borrada
  Lmax : DOUBLE;                -- Factor de carga maximo
  m, n : INTEGER;               -- Capacidad y numero de elementos de la(s) tabla(s)

feature { NONE }

explorar(inicial, intento, desp: INTEGER) : INTEGER is
-- Implementa la estrategia de exploración de la
-- tabla. Calcula la posición que corresponde al
-- enesimo intento de encontrar una posicion
-- en la tabla partiendo de una posicion inicial.
local desp : INTEGER;
do
  -- Exploración con  desplazamiento cociente
  Result := (inicial+desp*intento) \\ m
end -- explorar

reestructurar is
-- Incrementa el tamaño de la tabla
local
  -- Copias temporales de la tabla
  copia_claves  : ARRAY[CLAVE];
  copia_vacia   : ARRAY[BOOLEAN];
  copia_borrada : ARRAY[BOOLEAN];
  m_ant, i : INTEGER;
do
  m_ant := m; -- Se salva la capacidad actual
  m := 2*m; -- Se dobla el tamaño
  -- Se crean copias de la tabla anterior
  copia_claves := claves.clone;
  copia_vacia := vacia.clone;
  copia_borrada := borrada.clone;
  -- Se reinicializa la tabla con el nuevo tamaño
  n := 0;
  create claves.make(0,m-1);
  create vacia.make(0,m-1);
  create borrada.make(0,m-1);
  from i := 0 until i >= m loop
    vacia.put(TRUE,i) ; borrada.put(FALSE,i) ; i := i+1
  end
  -- Se insertan todos los elementos en la tabla nueva
  from i := 0 until i >= m_ant loop
    if not (vacia @ i) and not (borrada @ i) then
      insertar(claves @ i)
    end
  end
end – reestructurar

feature { ANY }

crear_tabla(L_Max: DOUBLE) is
require (L_Max > 0.0) and (L_Max < 1.0)
local i : INTEGER;
do
  Lmax := L_Max; m := 100; n := 0;
  create claves.make(0,m-1);
  create vacia.make(0,m-1);
  create borrada.make(0,m-1);
  from i := 0 until i >= m loop
    vacia.put(TRUE,i) ; borrada.put(FALSE,i) ; i := i+1
  end
end -- crear_tabla

pertenece(k: CLAVE) : BOOLEAN is
local i0,i,j,d : INTEGER;
do
  from
    i0 := k.hash_code \\ m;  -- Posición inicial
    d := k.hash_code // m;   -- Desplazamiento
    j := 1; -- Número de intentos
    i := i0; -- Posicion actual
  until (vacia @ i) or else
        (not (borrada @ i) and then (claves @ i = k)) or
        (j > n) loop
    i := explorar(i0, j, d); -- Se explora la siguiente posicion
    j := j+1
  end
  Result := (claves @ i = k)
end -- buscar

insertar(k: CLAVE) is
local i0,i,j : INTEGER;
do
  -- Se comprueba si es necesario reestructurar la tabla
  n := n+1;
  if (n = m) or (n/m > Lmax) then reestructurar end
  -- Se busca el sitio de inserción
  from
    i0 := k.hash_code \\ m
    d := k.hash_code // m
    j := 1; i := i0
  until (vacia @ i) or (borrada @ i) or else (claves @ i = k) or
        (j > n) loop
    i := explorar(i0, j, d) ; j := j+1
  end
  if j > n then -- La exploración no ha encontrado un hueco
    reestructurar ; insertar(k,v) -- Volver a intentar inserción
  else
    claves.put(k,i) ; vacia.put(False,i) ; borrada.put(False,i)
  end
end -- insertar

borrar(k: CLAVE) is
local i0,i,j : INTEGER;
do
  -- Se busca la clave que se debe borrar
  from
    i0 := k.hash_code \\ m
    d := k.hash_code // m
    j := 1; i := i0
  until (vacia @ i) or else
        (not (borrada @ i) and then (claves @ i = k)) or
        (j > n) loop
     i := explorar(i0, j, d) ; j := j+1
  end
  if claves @ i = k then
    -- Se encontro la clave - borrado perezoso
    borrada.put(True,i)
  end
end -- borrar

end -- class TABLA_DISP
```

Funciones de Exploración

- Las funciones de exploración más utilizadas son la exploración lineal y con desplazamiento (tambien llamada doble dispersión):

- **Exploración lineal**: $f_m(i_0 , j) = (i_0 + j) \bmod m$

- **Exploración por desplazamiento**: $f_m(i_0 , j, d) = (i_0 + j \cdot d) \bmod m$

  En este tipo de exploración se debe proporcionar un valor adicional, $d > 0$, el cual se calcula a partir del valor de la clave (existen varios métodos de calcularlo dependiendo del tipo de clave).

Propiedades de las Funciones de Exploración

- La función de **exploración lineal** es la estrategia más sencilla: Cada intento de realojamiento explora la casilla siguiente de la tabla. Se tiene la garantía de que si existe una posición libre esta estrategia la va a encontrar: Se explora toda la tabla.

- Sin embargo es vulnerable al problema del **agrupamiento secundario**: Si se insertan elementos cuyos claves generen índices correlativos, que comprendan una región de la tabla donde ya existen algunos elementos, podemos tener una tabla donde la mayoría de los elementos están desplazados de su posición original aunque gran parte de la tabla esté vacía. Para resolver éste problema es conveniente que las funciones de exploración recorran posiciones alejadas de la original.

- La **exploración con desplazamiento** resuelve el problema explorando posiciones alejadas a una distancia fija (el desplazamiento). Además, como el desplazamiento depende del valor de la clave, este método es menos vulnerable que los anteriores al problema del **agrupamiento primario** (función de dispersión no uniforme).

- Se puede garantizar que la exploración con desplazamiento recorre toda la tabla siempre que el tamaño de la tabla, *m*, sea un número primo.

Eficiencia en las Tablas de Dispersión

- La eficiencia dependerá de la longitud de las listas (dispersión abierta) o de la longitud del proceso de exploración (dispersión cerrada). Existen dos situaciones distintas: Explorar para encontrar un elemento o explorar para encontrar una posición vacía. El análisis en el caso promedio da los resultados siguientes:

	Nº promedio de accesos a tabla	
Operación	**Disp. Abierta**	**Disp. Cerrada**
Acceso (éxitoso)	$1+L/2$	$\mathbf{ln}(1/(1-L))/L$
Acceso (fallido)	$1+L$	$1/(1-L)$
Inserción	0 (salvo reestructuración)	$1/(1-L)$
Borrado	$1+L/2$	$\mathbf{ln}(1/(1-L))/L$

- En la dispersión abierta la dependencia es lineal con el factor de carga, siempre que la función de dispersión sea uniforme (la longitud de las listas no se desvíe demasiado del valor promedio, L). En el peor caso (una única lista contenga todos los elementos y las m-1 restantes no tengan ninguno) el número promedio de accesos a tabla serían proporcionales a n en lugar de ser proporcionales a L.

- En la dispersión cerrada, a medida que el factor de carga se aproxima al valor límite 1 (tabla llena), el número de accesos crece de manera potencial.

- Por lo tanto, si se desea garantizar un orden constante en el caso promedio, se deben cumplir las siguientes condiciones:

  - Diseñar la función de dispersión para que sea uniforme de acuerdo con las características de los datos que se van a utilizar.

  - En tablas de dispersión cerrada, usar exploración con desplazamiento si puede darse el problema de agrupamiento. No permitir que la tabla esté demasiado llena (valores de L cercanos a 1), habitualmente se reestructura si $L > 0.75$.