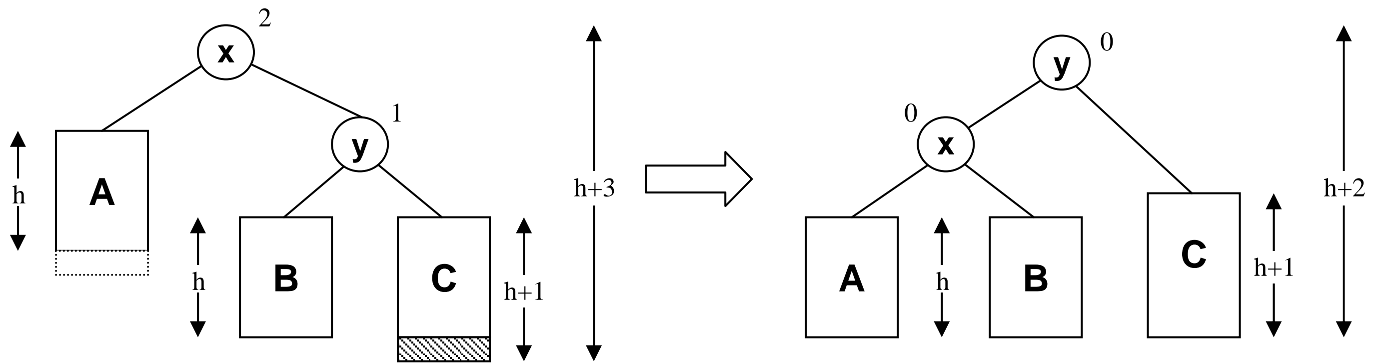
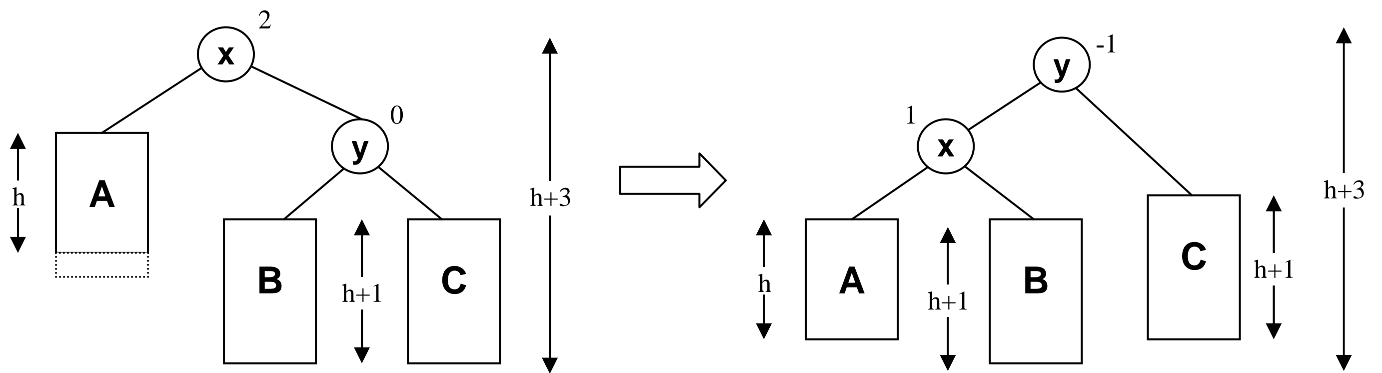
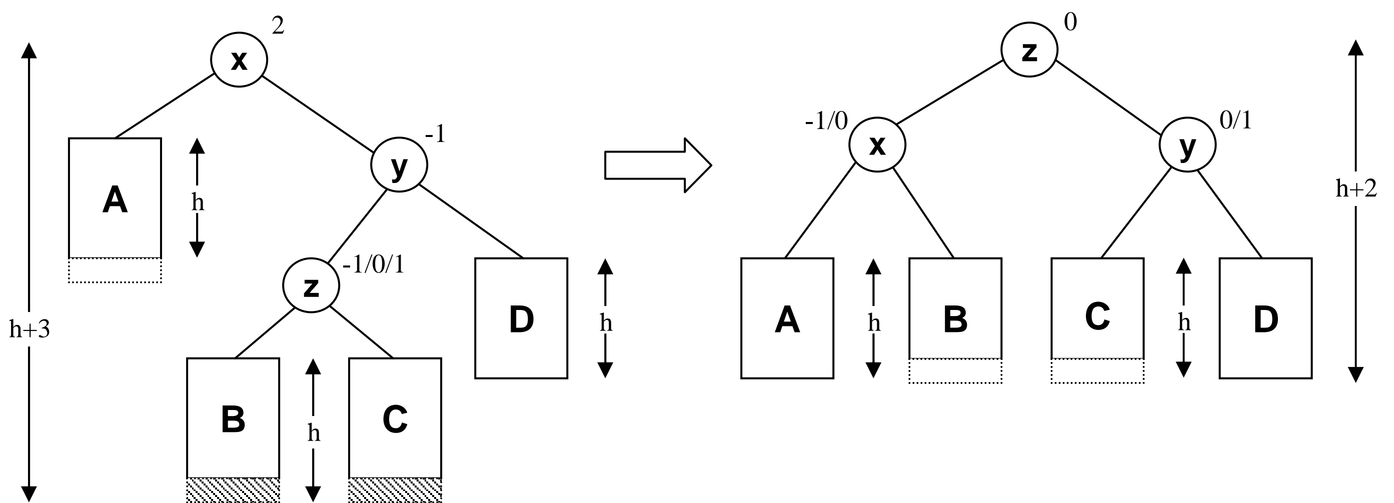
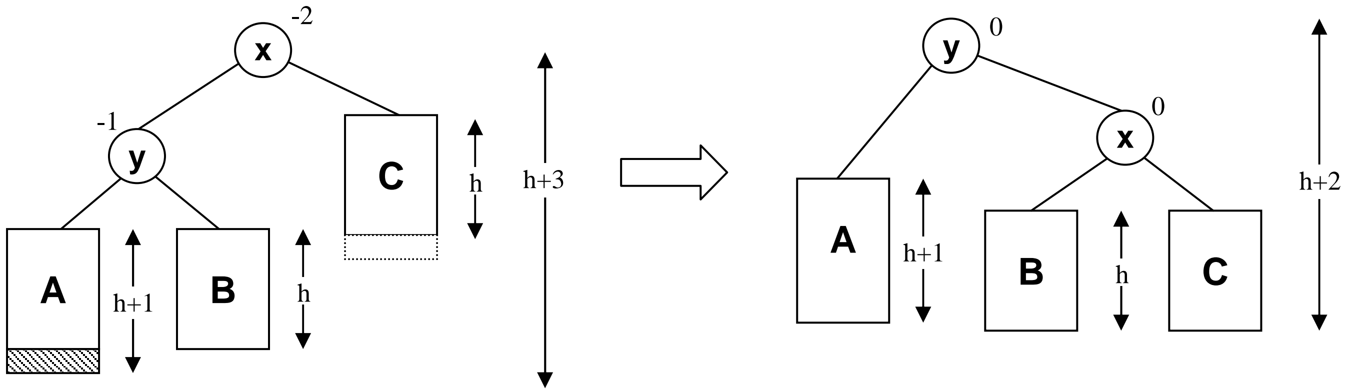
Arboles AVL - Rotaciones

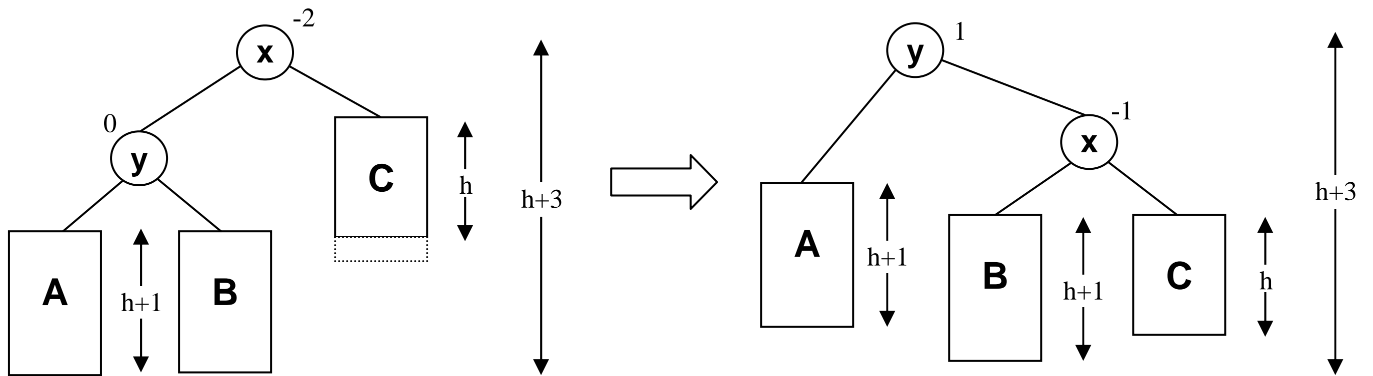


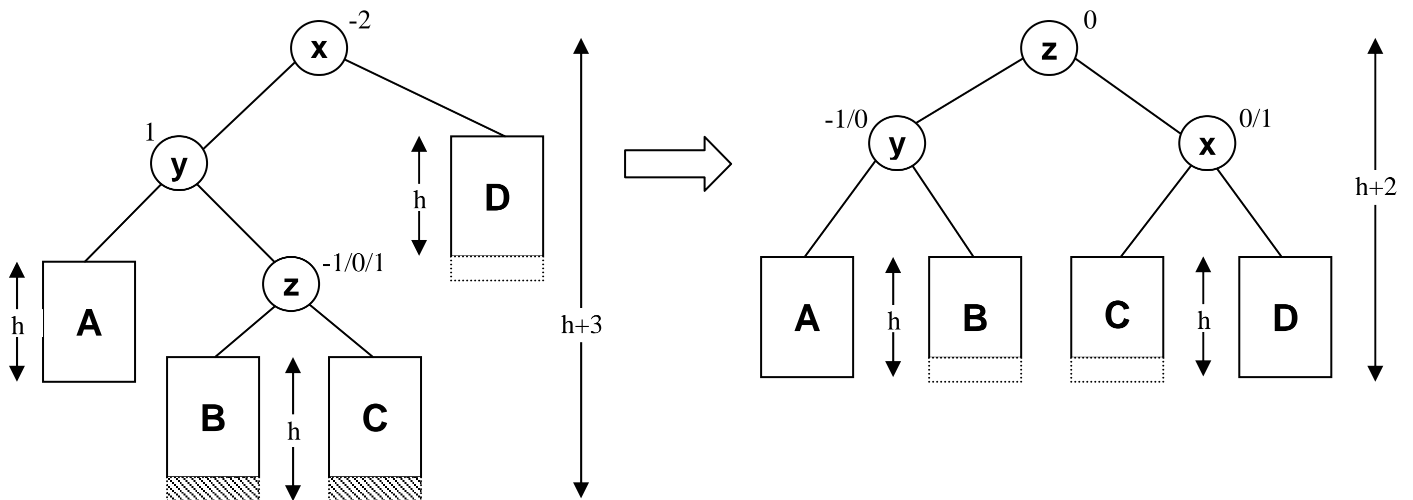




Arboles AVL – Rotaciones simétricas







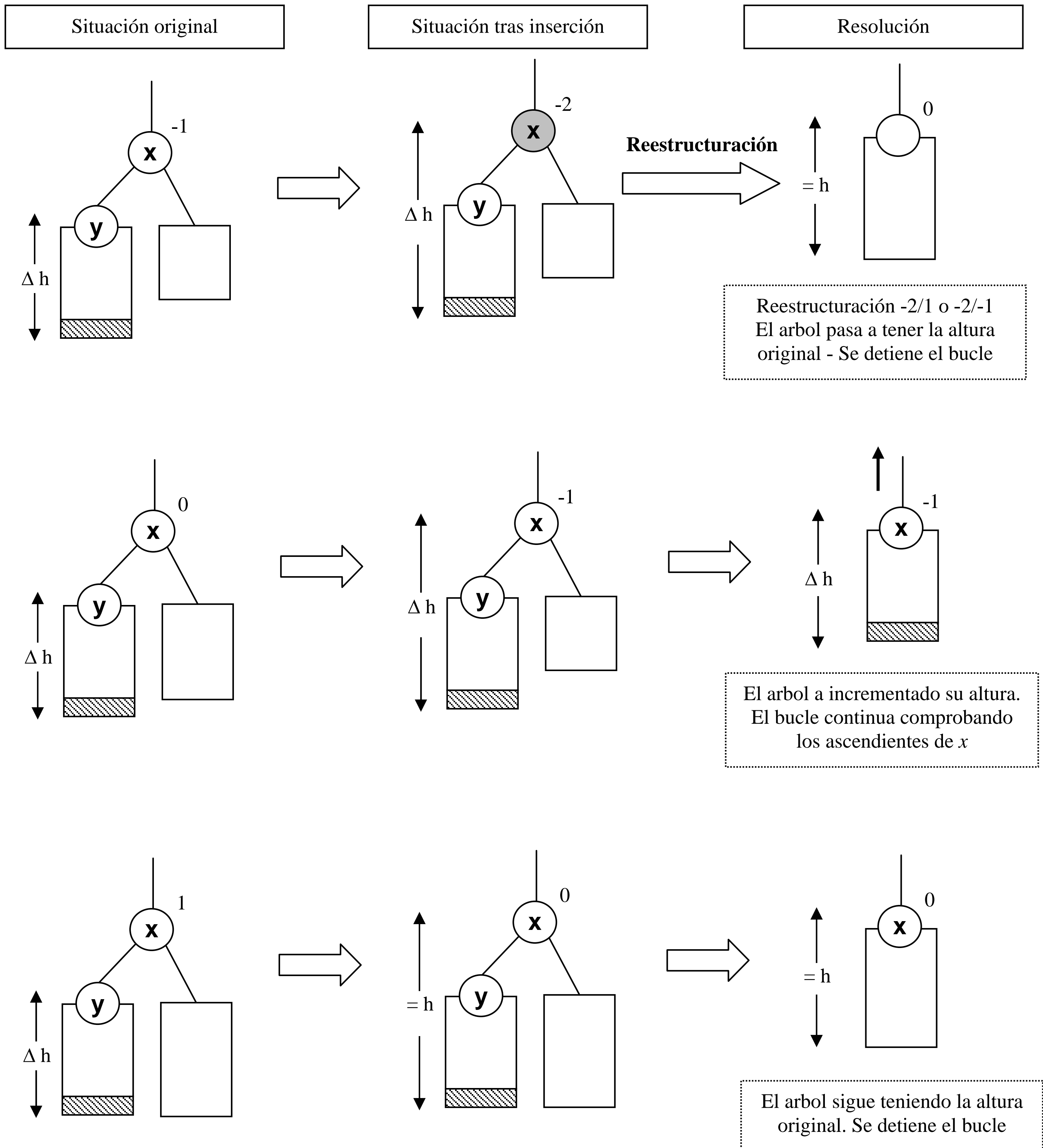
Implementación de un Arbol AVL (I)

- Definición de la clase y rotaciones:

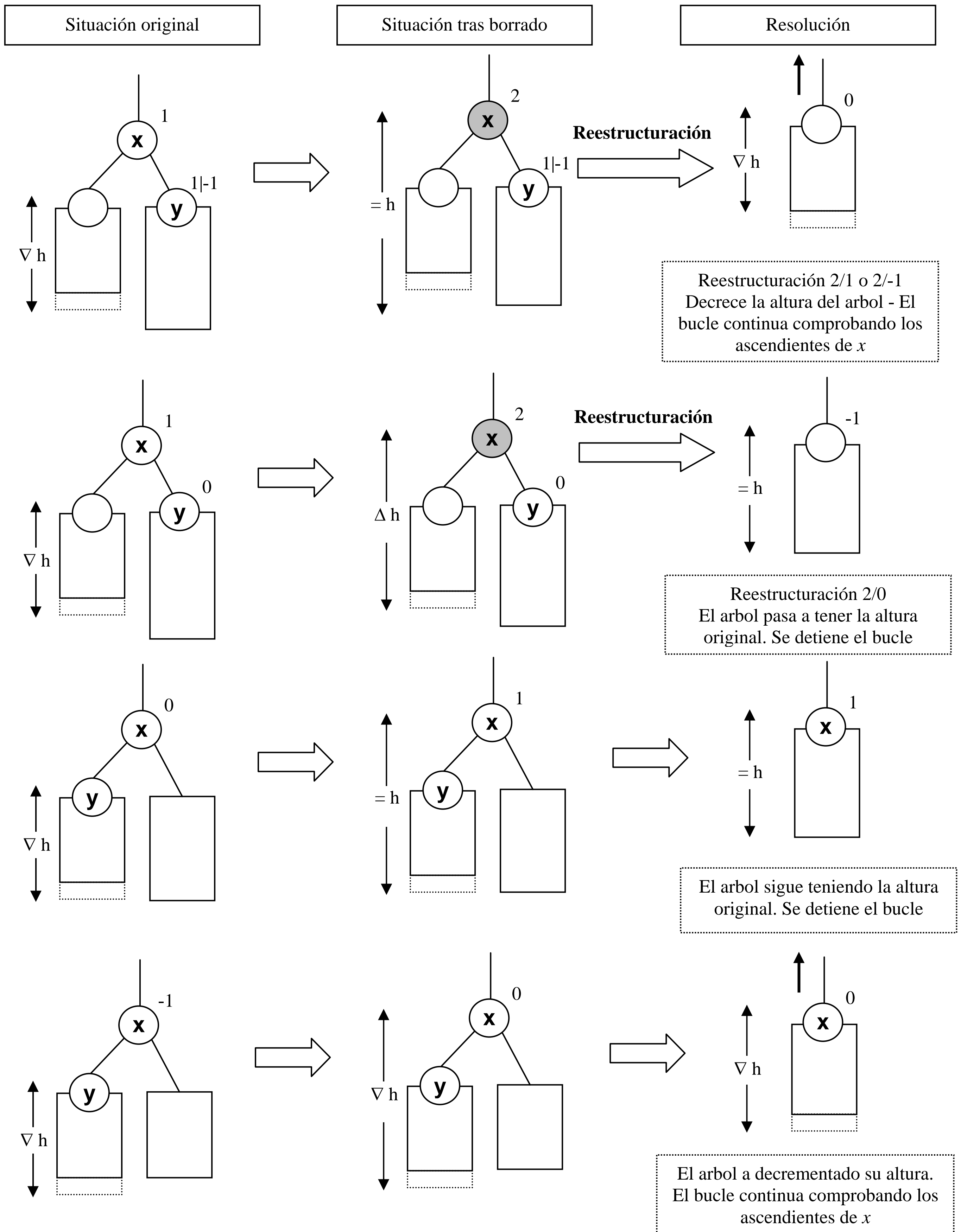
```
class ARBOL_AVL[ELEMENTO -> COMPARABLE]
inherit ARBOL_ABB[ELEMENTO]
  redefine insertar, quitar
end
creation crea_arbol
feature { NONE }

   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 deseado. El método más común para reducir el valor es hallar el resto de su división por el tamaño de la tabla (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: ($i \in [0..m\text{-}1]$ es el índice del elemento)

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

  - 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 $k \equiv k_0 \ldots k_{b\text{-}1}$ (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.

$$h(k_1) = h(k_2),\ \ k_1 \neq k_2$$

- 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. Si la función de dispersión se comporta de manera **uniforme** para el conjunto de datos utilizado, entonces L representa el tamaño promedio de las listas.

- 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. 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. 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, el tamaño promedio de las listas será de $L = n/m$ (**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:

- $h(dni) = dni$:

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: (Usa los 3 primeros dígitos del DNI)

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**: 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** (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)	$1+L/2$	$\ln(1/(1-L))/L$
Acceso (fallido)	$1+L$	$1/(1-L)$
Inserción	0 (salvo reestructuración)	$1/(1-L)$
Borrado	$1+L/2$	$\ln(1/(1-L))/L$

- En la dispersión abierta la dependencia es lineal con el factor de carga. En la dispersión cerrada, sin embargo, 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 $L > 0.8$.