EDA

Apuntes

UNED

Curso 2000 – 2001

Nota:

Los apuntes que tienes ahora en tus manos son los que he confeccionado para el estudio de la asignatura durante el curso 2000-2001. Son resúmenes de los temas de las Unidades Didácticas. He omitido aquellas partes del temario que, bien por su complejidad, bien porque no he visto que en los exámenes anteriores que he tenido en mis manos se preguntara nada sobre los mismos, he considerado que no valía la pena estudiar. Por supuesto esta es una valoración completamente subjetiva y en algunas cuestiones probablemente muy equivocada. Por eso quiero advertirte de tal extremo. Por otra parte he procurado en todo momento ser fiel en la transcripción, no obstante en algunos temas puntuales hay razonamientos propios y ligeras modificaciones respecto al texto base. En todos estos casos lo he mencionado convenientemente.

Espero que te sirvan de ayuda.

Un saludo, José Manuel.

1.- INTRODUCCIÓN Y CONCEPTOS FUNDAMENTALES

1.1.- Estructuras de datos y tipos de datos abstractos (TDA)

Un *tipo de datos* es una colección de valores

Un *tipo de datos abstracto* (TDA) es un tipo de datos definido de forma única mediante un tipo y un conjunto dado de operaciones definidas sobre el tipo.

Una *estructura de datos* es la implementación física de un tipo de datos abstracto.

Debido al proceso de abstracción, el diseño deberá realizarse siguiendo tres pasos fundamentales:

1.- Análisis de datos y operaciones
2.- Elección del Tipo de Datos Abstracto
3.- Elección de la implementación

1.2.- Tipos de datos básicos

TDA Entero: tiene como tipo el conjunto de números enteros, y como operaciones la suma, resta, multiplicación y división entera

TDA Real: tiene como tipo el conjunto de números reales y como operaciones la suma, resta multiplicación y división

TDA Booleano: tiene como tipo el conjunto de valores booleanos True, False y como operaciones las definidas en el álgebra de Boole (AND, OR, NOT)

TDA Carácter: tiene como tipo el conjunto de caracteres definido por un alfabeto dado y como operaciones todos los operadores relacionales (<, >, =, >=, <=, <>)

Se denominan tipos de datos ***escalares*** a aquellos en los que el conjunto de valores está ordenado y cada valor es atómico: *Carácter, Entero, Real y Booleano.*

Se denominan tipos de datos ***ordinales*** a aquellos en los que cada valor tiene un predecesor (excepto el primero), y un único sucesor (excepto el último): *Carácter, Entero y Booleano.*
Los lenguajes de alto nivel suelen presentar tres funciones adicionales asociadas a los tipos de datos ordinales: posición (*Ord*), predecesor (*Pred*) y sucesor (*Suc*) de un elemento.

Otro tipo de datos son los denominados ***compuestos*** dado que son divisibles en *componentes* que pueden ser accedidas individualmente. Las operaciones asociadas a los tipos compuestos o estructurados son las de almacenar y recuperar sus componentes individuales.

Los TDA compuestos básicos son:

TDA Conjunto: colección de elementos tratados con las operaciones unión, intersección y diferencia de conjuntos.

TDA Arreglo: colección homogénea de longitud fija tal que cada una de sus componentes pueden ser accedidas individualmente mediante uno o varios índices, que serán de tipo ordinal, y que indican la posición de la componente dentro de la colección.

TDA Registro: tipo de datos heterogéneo compuesto por un número fijo de componentes denominadas campos a las que se accede mediante un selector de campo.

El TDA *Conjunto* no es estructurado ya que no está organizado mediante el modo de acceso a sus componentes individuales, mientras que el Arreglo y el Registro si lo son. Los registros pueden tener como campos tipos de datos simples o arreglos o incluso otros registros.

Los TDA básicos son los utilizados con mayor frecuencia, pero además cada problema puede requerir la definición de varios TDA propios. Por ejemplo, un TDA muy utilizado en aplicaciones informáticas es el TDA lista definido de la siguiente forma:

TDA lista (o *Secuencia*): colección homogénea de datos, ordenados según su posición en ella, tal que cada elemento tiene un predecesor (excepto el primero) y un sucesor (excepto el último) y los operadores asociados son:

Insertar: inserta un elemento x, en una posición p de la lista
Localizar: localiza la posición p en la que se encuentra el dato x
Recuperar: encuentra el elemento x que esta en la posición p
Suprimir: elimina de la lista el elemento de la posición p
Suprimir_dato: elimina de la lista cualquier aparición del elemento x
Anula: ocasiona que la lista se vacíe
Primero (*Fin*): proporciona el primer (último) elemento de la lista
Imprimir: imprime todos los elementos de la lista en su orden.

En general el término "lista" se utiliza cuando el TDA se implementa sobre memoria principal mientras que "secuencia" suele reservarse para implementaciones sobre memoria secundaria.

La lista es una estructura dinámica ya que su longitud dependerá del número de elementos que tenga, aumentará al insertar y se reducirá al suprimir. Ahora bien, la lista puede implementarse de formas muy diferentes. Una de ellas es mediante arreglos. Así, la lista, que es dinámica independientemente de su implementación, se realiza mediante una implementación estática.

1.3.- Punteros y Listas enlazadas:

PUNTEROS

Un puntero es un tipo de datos simple *cuyo valor es la* ***dirección*** de una variable de otro tipo, denominada variable referenciada.

Los punteros permiten realizar una reserva dinámica de memoria. Las variables referenciadas son variables dinámicas, es decir, se crean en tiempo de ejecución.

El valor de un puntero es un número entero, sin embargo, este valor no es de tipo entero sino una dirección.

El contenido de la variable referenciada puede ser cualquier tipo de dato: registro, otro puntero, etc.

Declaración de los tipo puntero: TYPE

Tipo_puntero = POINTER TO Tipo_var_referenc;

Declaración de la variable puntero: VAR

variable_puntero : Tipo_puntero;

La memoria reservada para la variable puntero es reservada y ocupada de la misma manera que las variables estáticas.

La variable referenciada no existe hasta que no ha sido creada (en Modula2 Allocate):

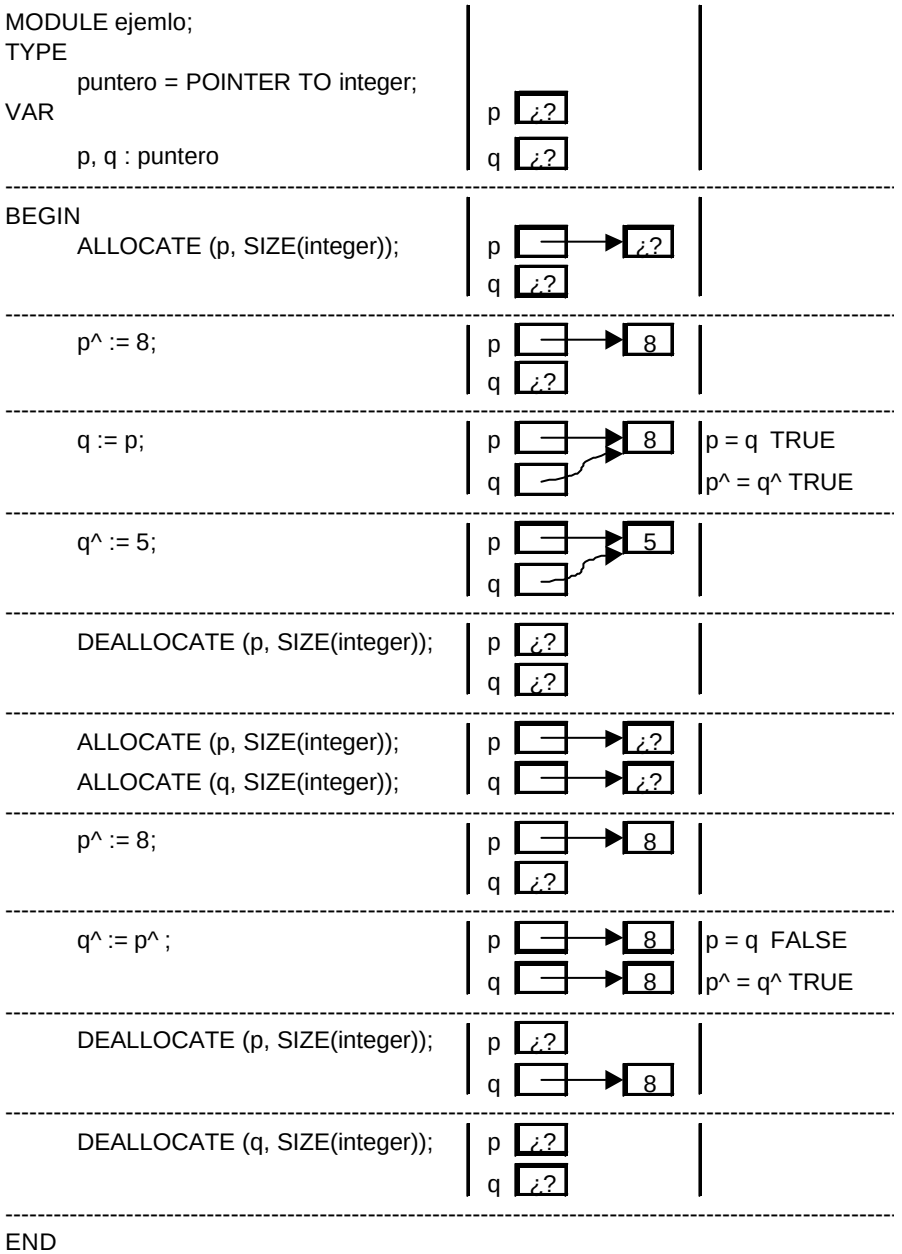
Allocate (variable_puntero, SIZE (Tipo_variable_referenciada));

Las variables referenciadas recién creadas no tienen ningún valor (están sin inicializar). Para asignarlo se accede a la variable referenciada mediante el nombre del puntero que la apunta y el operador de acceso "^", realizándose la asignación mediante su operador "**:=**"

Una vez que la variable referenciada ya no va a ser utilizada por el programa, será destruida o liberada (en Modula2 Deallocate):

Deallocate (variable_puntero, SIZE(Tipo_referenciada));

Las operaciones asociadas al TDA puntero son la asignación y la comparación:
La *asignación de punteros* consiste en la asignación de "direcciones" entre variables puntero.
La *comparación de punteros* consiste en comprobar si sus direcciones son iguales o no



Cuando un puntero no señala a ninguna variable referenciada se le asigna un valor que representa dicha circunstancia (en Modula2 NIL)

En definitiva, el TDA puntero es un tipo de dato simple cuyos valores son direcciones de memoria, y sus operadores asociados son la asignación de punteros y la comparación de punteros.

Los punteros son la base de construcción de todas las denominadas estructuras dinámicas, como los árboles, las listas enlazadas, etc.

LISTAS ENLAZADAS

La finalidad de las listas enlazadas, como en el caso de los arreglos, es almacenar organizadamente datos del mismo tipo.

Las listas enlazadas son tipos de datos dinámicos que se construyen con ***nodos***.

Un *nodo es un registro con dos campos:* ***datos*** *y* ***enlace***. El campo datos debe entenderse conceptualmente, es decir, en una implementación concreta pueden ser varios los campos que se consideran como datos.

El TDA lista enlazada es una colección de nodos ordenada según su posición, tal que cada uno de ellos es accedido mediante un campo enlace del nodo anterior.

Llegados a este punto es importante reflexionar sobre las características de los TDA Arreglo, Lista enlazada y Lista. Los arreglos y las listas tiene como tipo componente cualquiera, mientras que las listas enlazadas tienen como componentes nodos. Los arreglos son estructuras estáticas, mientras que las otras dos son dinámicas. El arreglo accede a sus elementos de forma aleatoria mediante un índice, mientras que la lista y la lista enlazada lo realizan secuencialmente. Además, la lista enlazada accede a los elementos necesariamente mediante el campo enlace.

Las listas enlazadas se pueden implementar estáticamente, sin mas que definir el tipo del arreglo como nodos y utilizar los índices en el campo enlace para determinar el siguiente nodo.

En la implementación dinámica el campo de enlace será un puntero y los nodos son variables referenciadas:

```
TYPE   Ptr_Nodo = POINTER TO Nodo;
       Nodo = RECORD
                  datos :   Tipo_datos
                  enlace : Ptr_Nodo;
              END;
```

La declaración de la lista se hace mediante un puntero externo

```
VAR   lista : Ptr_Nodo
```

Se suele definirse un nuevo tipo Tlista y declarar las variables de este nuevo tipo:

```
TYPE    Tlista = Ptr_Nodo;

VAR     lista : Tlista;
```

Funciones elementales:

Inserción por la cabeza:

1.- Crear el nuevo nodo (reservar memoria)
2.- Almacenar el dato en el campo correspondiente (datos)
3.- Como el nuevo nodo será el primero, su campo enlace apuntará al hasta ahora primer nodo.
4.- El enlace externo (lista), deberá apuntar ahora al nuevo nodo

```
*************************************************************************
      PROCEDURE Insertar_cabeza(VAR lista: Ptr_Nodo; nuevo_dato : Tipo_datos);
      VAR
            Nuevo_nodo: Ptr_Nodo;
      BEGIN
            ALLOCATE(Nuevo_nodo,SIZE(Nodo));
1           Nuevo_nodo^.datos:=nuevo_dato;
2           Nuevo_nodo^.enlace :=lista;
3           lista:=Nuevo_nodo;
      END Insertar_cabeza;
*************************************************************************
```

Inserción por el final

Deberemos llegar al final y hacer la inserción. Para recorrer la lista hasta llegar al final utilizamos un puntero auxiliar (Actual).
Tenemos que distinguir dos situaciones según la lista esté o no vacía. En caso de que la lista este vacía estaremos en el caso de realizar una inserción por la cabeza. En cualquier otro caso el procedimiento a seguir será:

1.- crear el nuevo nodo
2.- introducir los nuevos datos en el campo de datos
3.- se asigna NIL a su campo enlace (ya que será el último)
4.- inicializamos el puntero auxiliar Actual (Actual:= lista) para que apunte al primer elemento de la lista.
5.- hacemos avanzar este puntero hasta que alcance el final (Actual^.enlace sea NIL)
6.- realizamos el enlace entre el último elemento de la lista y el nuevo nodo

```
*************************************************************************
      PROCEDURE Insertar_final (VAR lista: Ptr_Nodo;nuevo_dato : Tipo_datos);
      VAR
         Nuevo_nodo, Actual : Ptr_Nodo;
      BEGIN
         IF lista=NIL THEN Insertar_cabeza(lista,nuevo_dato) ELSE
      1      ALLOCATE(Nuevo_nodo,SIZE(Nodo));
      2      Nuevo_nodo^.datos := nuevo_dato;
      3      Nuevo_nodo^.enlace := NIL;
      4      Actual := lista;
      5      WHILE Actual^.enlace <> NIL DO
               Actual := Actual^.enlace;
             END;
      6    Actual^.enlace := Nuevo_nodo
         END;
      END Insertar_final;
*************************************************************************
```

Suprimir por la cabeza:

1.- Se almacena la dirección del primer elemento, apuntado por la variable lista, en un puntero auxiliar (Actual), Actual := lista;
2.- Asignamos al parámetro formal dato, el valor almacenado en el elemento auxiliar
3.- Se realiza la asignación de la lista al nuevo primer elemento (lista:=lista^.enlace)
4.- Se libera la memoria del nodo que ya no pertenece a la lista

```
*************************************************************************
      PROCEDURE Suprimir_cabeza (VAR lista: Ptr_Nodo; VAR dato : Tipo_datos);
      VAR
         Actual: Ptr_Nodo;
      BEGIN
      1   Actual:=lista;
      2   dato:=lista^.datos;
      3   lista:=lista^.enlace;
      4   DEALLOCATE(Actual,SIZE(Nodo));
      END Suprimir_cabeza;
*************************************************************************
```

Suprimir por el final

El procedimiento requiere localizar la última posición de la lista y liberar el último nodo. Además debemos asignar el valor NIL al penúltimo elemento que tras la eliminación será el último. Para ello utilizamos un puntero auxiliar (Actual) que al final del recorrido apuntará al último elemento y otro puntero (Ant) que quedará apuntando al anterior. Si solo hay un nodo en la lista (Ant = Actual), se deberá hacer lista := NIL

1.- Inicializar los punteros
2.- Recorrido de la lista hasta alcanzar el final
3.- Recuperar el dato almacenado en el último nodo
4.- Eliminación del nodo y marcado del nuevo último nodo
5.- Liberamos la memoria del nodo

```
*************************************************************************
      PROCEDURE Suprimir_final (VAR lista: Ptr_Nodo; VAR dato : Tipo_datos);
      VAR
         Actual, Ant : Ptr_Nodo;
      BEGIN
      1   Ant:=lista;
          Actual:=lista;
      2   WHILE Actual^.enlace <> NIL DO
            Ant:=Actual;
            Actual:=Actual^.enlace
          END;
      3   dato:=Actual^.datos;
      4   Ant^.enlace := NIL;
          IF Ant=Actual THEN   (*si solo hay un nodo*)
             lista:=NIL;
          END;
      5   DEALLOCATE(Actual,SIZE(Nodo))
      END Suprimir_final;
*************************************************************************
```

Inserción según un criterio de orden

En general deben considerarse tres situaciones posibles: que la posición a insertar sea la primera (insertar al principio), la última (insertar al final) o una entre éstas (caso general).

Hay que tener en cuenta que al realizar la inserción, en el caso general, debemos acceder simultáneamente a dos nodos consecutivos, por lo que necesitamos dos punteros: Actual y Anterior.

```
*****************************************************************************************************
       PROCEDURE Insertar_orden_delante (VAR lista: Ptr_Nodo; dato : Tipo_datos);
       VAR
          Encontrado: BOOLEAN;
          Actual,Nuevo_nodo,Anterior: Ptr_Nodo;
       BEGIN
          IF (lista=NIL)OR(dato<lista^.datos) THEN
         (* Insercion al principio *)
            Insertar_cabeza(lista,dato)
          ELSE
            ALLOCATE(Nuevo_nodo,SIZE(Nodo));
            Nuevo_nodo^.datos:=dato;
            (*Busca el punto de insercion*)
            Anterior:=lista;
            Actual := lista^.enlace;
            Encontrado := FALSE;
            WHILE (Actual<>NIL)AND(NOT Encontrado) DO
               IF dato>Actual^.datos THEN
                  Anterior:=Actual;
                  Actual:=Actual^.enlace
               ELSE
                  Encontrado:=TRUE;
               END;
            END;
            (*Insertar el nuevo nodo*)
             Nuevo_nodo^.enlace := Actual;
             Anterior^.enlace := Nuevo_nodo
         END
       END Insertar_orden_delante;
*****************************************************************************************************

       PROCEDURE Insertar_orden_detras (VAR lista: Ptr_Nodo; dato : Tipo_datos);
       VAR
          Encontrado: BOOLEAN;
          Actual, Nuevo_nodo,Anterior: Ptr_Nodo;
       BEGIN
          IF (lista=NIL)OR(dato<lista^.datos) THEN
         (* Insercion al principio *)
            Insertar_cabeza(lista,dato)
          ELSE
            ALLOCATE(Nuevo_nodo,SIZE(Nodo));
            Anterior:=lista;
            Actual := lista^.enlace;
            Encontrado := FALSE;
            WHILE (Actual<>NIL)AND(NOT Encontrado) DO  (*Ins. en medio*)
               IF dato>Actual^.datos THEN
                  Anterior:=Actual;
                  Actual:=Actual^.enlace
               ELSE
                  Nuevo_nodo^:=Actual^;
                  Actual^.enlace:=Nuevo_nodo;
                  Actual^.datos:=dato;
                  Encontrado:=TRUE;
               END;
            END;
            IF NOT Encontrado THEN    (*Insercion al final*)
               Nuevo_nodo^.datos:=dato;
               Nuevo_nodo^.enlace:=NIL;
               Anterior^.enlace:=Nuevo_nodo
            END
         END
       END Insertar_orden_detras;
*****************************************************************************************************
```

Imprimir una lista enlazada:

Podremos realizar la impresión en orden o en orden inverso (mediante una implementación recursiva)

```
PROCEDURE Imprimir_lista(lista:Ptr_Nodo);
VAR
   Aux : Ptr_Nodo;
BEGIN
   Aux := lista;
   WHILE Aux <> NIL DO
      WriteInt(Aux^.datos,7);
      Aux := Aux^.enlace;
   END;
END Imprimir_lista;
```

```
PROCEDURE Imprimir_contrario
(lista: Ptr_Nodo);
BEGIN
   IF lista <> NIL THEN
Imprimir_contrario(lista^.enlace);
      WriteInt(lista^.datos,6);
   END;
END Imprimir_contrario;
```

---------- (desde aquí hasta final tema 1solo para entender – no estudiar) ----------

1.4.- TDA lista implementado con listas enlazadas

```
PROCEDURE Insertar (VAR L: Ptr_Nodo; x : Tipo_datos;p: INTEGER);
(*inserta un elemento, x, en una posicion p de L, pasando los elementos
 de la posicion p y siguientes a la posicion inmediatamente posterior*)
VAR
  Anterior, Actual,Nuevo_nodo: Ptr_Nodo;
  cont: INTEGER;

BEGIN
   IF p=1 THEN   (*Inserta en la primera posicion*)
      ALLOCATE(Nuevo_nodo,SIZE(Nodo));
      Nuevo_nodo^.datos:=x;
      Nuevo_nodo^.enlace:=L;
      L:=Nuevo_nodo
   ELSE
      (*localiza la posicion p de insercion*)
      Anterior:=NIL;
      Actual:=L;
      cont:=1;
      WHILE (Actual#NIL)&(cont#p) DO
         Anterior:=Actual;
         Actual:=Actual^.enlace;
         cont:=cont+1
      END;
      IF p<cont+1  THEN(*p menor que el numero de elementos de L*)
            ALLOCATE(Nuevo_nodo,SIZE(Nodo));
            Nuevo_nodo^.datos:=x;
            Anterior^.enlace:=Nuevo_nodo;
         IF(Actual=NIL)OR((Actual=L)&(cont=p-1))THEN(*insertar en la ultima posicion*)
            Nuevo_nodo^.enlace:=NIL
         ELSIF cont=p THEN                  (*insertar en posicion intermedia*)
            Nuevo_nodo^.enlace:=Actual;
         END
      END
   END
END Insertar;

(*****************************************************************************)
```

```
(*****************************************************************************)

      PROCEDURE Localizar (L: Ptr_Nodo; x : Tipo_datos):INTEGER;
      (*localiza la posición en la que se encuentra un elemento dado x,
      si no lo encuentra devuelve 0*)
      VAR
         Actual,Anterior: Ptr_Nodo;
         Encontrado: BOOLEAN;
         p: INTEGER;

      BEGIN
         Anterior:=NIL;
         Actual:=L;
         p:=0;

         Encontrado:=FALSE;
         WHILE NOT Encontrado & (Actual#NIL) DO
            IF Actual^.datos=x THEN
               Encontrado:=TRUE;
               p:=p+1
            ELSE
               Anterior:=Actual;
               Actual:=Actual^.enlace;
               p:=p+1;
            END;
         END;
         IF NOT Encontrado THEN
         (*Elemento no encontrado*)
               p:=0;
         ELSE
            Actual:=Actual^.enlace;
         END;
         RETURN p
      END Localizar;

(*****************************************************************************)



PROCEDURE Recuperar (L: Ptr_Nodo; VAR x : Tipo_datos;p: INTEGER;
VAR encontrado:BOOLEAN);
(*encuentra el elemento x que esta en la posición p, si la posición p es
mayor que el numero de elementos de L, devuelve encontrado a FALSE*)
VAR
   Actual: Ptr_Nodo;
   cont:INTEGER;

BEGIN
   encontrado:=FALSE;
   Actual:=L;
   cont:=1;
   WHILE (Actual#NIL)&(cont<p) DO
      Actual:=Actual^.enlace;
      cont:=cont+1
   END;
   IF (cont=p)&(Actual#NIL) THEN
      x:=Actual^.datos;
      encontrado:=TRUE;
   END
END Recuperar;

(*****************************************************************************)
```

```
(*****************************************************************************)

      PROCEDURE Suprimir (VAR L: Ptr_Nodo; p: INTEGER);
      (*elimina de L el elemento de la posición p*)
      VAR
         Anterior,Actual: Ptr_Nodo;
         cont:INTEGER;

      BEGIN
         Anterior:=NIL;
         Actual:=L;
         cont:=1;
         (*Localizar la posición p*)
         WHILE (Actual^.enlace#NIL)&(cont<p) DO
            Anterior:=Actual;
            Actual:=Actual^.enlace;
            cont:=cont+1
         END;
         IF cont=p THEN
            (*Eliminar*)
            IF cont=1 THEN    (*eliminar el primer elemento*)
               L:=Actual^.enlace
            ELSIF Actual^.enlace=NIL THEN  (*eliminar el ultimo*)
               Anterior^.enlace:=NIL
            ELSE (*intermedio*)
               Anterior^.enlace:= Actual^.enlace;
            END;
               DEALLOCATE(Actual,SIZE(Nodo))
         END
      END Suprimir;

(*****************************************************************************)

PROCEDURE Suprimir_dato (VAR L: Ptr_Nodo; x: Tipo_datos);
(*elimina de L todas las apariciones del elemento x*)
VAR
   Anterior,Actual,Aux: Ptr_Nodo;
   Encontrado: BOOLEAN;

BEGIN
   Anterior:=NIL;
   Actual:=L;
   WHILE Actual#NIL DO  (*búsqueda de todas las apariciones*)
      Encontrado:=FALSE;
      WHILE NOT Encontrado & (Actual#NIL) DO
         IF Actual^.datos=x THEN
            Encontrado:=TRUE
         ELSE
            Anterior:=Actual;
            Actual:=Actual^.enlace;
         END;
      END;
      IF Encontrado THEN  (*Eliminar*)
         IF Actual=L THEN    (*eliminar el primer elemento*)
            L:=Actual^.enlace;
            Aux:=L;
         ELSIF (Actual^.enlace=NIL)&(Anterior#NIL) THEN  (*eliminar el ultimo*)
            Anterior^.enlace:=NIL;
            Aux:=NIL;
         ELSE (*intermedio*)
            Anterior^.enlace:= Actual^.enlace;
            Aux:=Actual^.enlace;
         END;
         DEALLOCATE(Actual,SIZE(Nodo));
         Actual:=Aux;
      END;
   END;
END Suprimir_dato;

(*****************************************************************************)
```

```
(*****************************************************************************)

       PROCEDURE Anula (VAR L: Ptr_Nodo);
       (*ocasiona que L se vacie*)
       VAR
          Actual: Ptr_Nodo;

       BEGIN
          IF L#NIL THEN
             Actual:=L;
             WHILE Actual^.enlace#NIL DO
                L:=Actual^.enlace;
                DEALLOCATE(Actual,SIZE(Nodo));
                Actual:=L;
             END;
             DEALLOCATE(L,SIZE(Nodo));
             L:=NIL;
          END;
       END Anula;


(************************************************************)

PROCEDURE Primero (L: Ptr_Nodo): Tipo_datos;
(*proporciona el primer elemento de L*)

BEGIN
   IF L#NIL THEN
      RETURN L^.datos
   END;
END Primero;


(************************************************************)

       PROCEDURE Fin (L: Ptr_Nodo): Tipo_datos;
       (*proporciona el ultimo elemento de L*)
       VAR
          Anterior, Actual: Ptr_Nodo;

       BEGIN
             Anterior:=NIL;
             Actual:=L;
             WHILE Actual#NIL DO
                Anterior:=Actual;
                Actual:=Anterior^.enlace
             END;
             IF Anterior#NIL THEN
                RETURN Anterior^.datos
             END;
       END Fin;


(************************************************************)

PROCEDURE Imprimir (L: Ptr_Nodo);
(*imprime todos los elementos de la lista en su orden*)
VAR
   Actual : Ptr_Nodo;

BEGIN
   Actual := L;
   WHILE Actual # NIL DO
      WriteInt(Actual^.datos,6);
      Actual := Actual^.enlace;
   END;
END Imprimir;
```

4.- TDA DINÁMICOS LINEALES

1.- Introducción

Un TDA es estático cuando el número de elementos que lo componen es fijo y es dinámico si es variable. Para los estáticos la memoria se reserva en tiempo de compilación, para los dinámicos, en tiempo de ejecución.

Los registros son un ejemplo de TDA estático. Las listas enlazadas son el TDA dinámico más sencillo.

2.- Listas doblemente enlazadas (solo entender – no estudiar)

Cuando se necesita recorrer la estructura en ambos sentidos se puede definir el tipo nodo de manera que el campo enlace tenga dos punteros, uno que apunte al sucesor y otro al predecesor

```
(* Declaración lista doblemente enlazada *)
      TYPE
         tipo_datos= INTEGER;
         tipo_doble= POINTER TO nododoble;
         nododoble= RECORD
                       datos: tipo_datos;
                       sig,ant: tipo_doble
                    END;
         tlista=tipo_doble;
      VAR
         lista: tlista;
```

```
(* Inserción por el principio en una lista doblemente enlazada *)
      PROCEDURE ins_ini(VAR lista:tlista;nuevo:tipo_datos);
      VAR
         aux: tipo_doble;
      BEGIN
         ALLOCATE(aux, SIZE(nododoble));
         aux^.datos:=nuevo;
         aux^.sig:=lista;
         aux^.ant:=NIL;
         IF lista#NIL THEN lista^.ant:=aux; END;
         lista:=aux;
      END ins_ini;
```

```
(* Inserción en orden en una lista doblemente enlazada *)
      PROCEDURE ins_orden(VAR lista:tlista;nuevo: tipo_datos);
      VAR
        aux,ant,tempo: tipo_doble;
      BEGIN
         IF lista=NIL THEN
            ins_ini(lista,nuevo)
         ELSE
            ALLOCATE(tempo,SIZE(nododoble));
            tempo^.datos:=nuevo;
            aux:=lista; ant:=NIL;
            WHILE (aux#NIL) AND (nuevo>aux^.datos)DO
               ant:=aux;
               aux:=aux^.sig;
            END;
            tempo^.sig:=aux;
            tempo^.ant:=ant;
            IF ant=NIL THEN lista:=tempo ELSE ant^.sig:=tempo END;
            IF aux#NIL THEN aux^.ant:=tempo END;
         END;
      END ins_orden;
```

(* Eliminación de la primera aparición de un elemento dado en una lista doblemente enlazada*)

```
PROCEDURE sup_nodo(VAR lista:tlista;dato:tipo_datos);
VAR
  aux: tipo_doble;
BEGIN
   aux:=lista;
   WHILE (aux#NIL) & (aux^.datos#dato) DO
     aux:=aux^.sig;
   END;
   IF aux#NIL THEN   (*Si existe el nodo*)
     IF aux^.sig#NIL THEN   (*Si tiene siguiente*)
        aux^.sig^.ant:=aux^.ant;
     END;
     IF aux^.ant#NIL THEN    (*Si tiene anterior*)
        aux^.ant^.sig:=aux^.sig
     ELSE
        lista:=aux^.sig
     END;
     DEALLOCATE(aux,SIZE(nododoble));
   END;
END sup_nodo;
```

(* Imprimir en sentido inverso una lista doblemente enlazada *)

```
PROCEDURE Imprimir(lista: tlista);
(*Imprime lista en orden inverso*)
VAR
  aux: tipo_doble;
BEGIN
   aux:=lista;
   IF lista#NIL THEN
   WHILE aux^.sig#NIL DO
      aux:=aux^.sig
   END;
   WriteInt(aux^.datos,4); (*ultimo*)
   WHILE aux^.ant#NIL DO
      WriteInt(aux^.ant^.datos,4);
      aux:=aux^.ant
   END;
   END;
   WriteLn;
END Imprimir;
```

Las listas doblemente enlazadas requieren más memoria (un puntero más por nodo). Sólo se justifica su utilidad cuando la necesidad de recorrer la estructura hacia los predecesores sea frecuente; por ejemplo, si en función de un valor en un nodo se necesita volver a examinar sus predecesores.

En otros casos, puede definirse una lista enlazada con algún puntero auxiliar que solucione el problema. Por ejemplo si se desea tener un acceso rápido al último elemento sin necesidad de tener que recorrer toda la estructura bastará añadir un puntero externo que mantenga la dirección del último nodo. Evidentemente las funciones realizadas sobre el TDA deberán actualizar el puntero final de la lista.

También se utilizan las listas enlazadas “con memoria”. En este tipo de listas se añade un puntero externo adicional, denominado índice, que está apuntando siempre al último elemento accedido, lo que permite detener el recorrido sobre la lista para efectuar cualquier otra tarea y continuar después en el nodo donde se paró, sin necesidad de comenzar el recorrido desde el principio. Además, este tipo de listas con memoria permiten el recorrido externo de la lista, de modo que el recorrido lo va realizando el programa.:

```
1.- poner el índice al principio (final) de la lista
2.- Mientras el índice sea distinto de NIL
       2.1.- Consultar la lista para obtener el elemento apuntado por el índice
       2.2.- Procesar el dato
       2.3.- Avanzar (retroceder) a la posición siguiente (anterior).
```

Una variación habitual de las listas doblemente enlazadas son las listas circulares doblemente enlazadas en las cuales, el último nodo tiene como sucesor al primero, y el primero tiene como predecesor al último.

Un tipo de estructura especialmente interesante son las listas multireferenciadas en las cuales el nodo que vertebra la estructura no contiene la información directamente sino un puntero a la misma. La información se almacena en una variable referenciada a la que se añade un contador que lleva la cuenta de cuantos punteros (listas) están apuntando en cada momento la mencionada información.

3.- PILAS (LIFO: último en entrar, primero en salir)

Los elementos se añaden y suprimen por un único extremo: tope o cabeza de la pila.
Una pila es un TDA dinámico homogéneo al que sólo se tiene acceso por la cabeza o cima de la pila, los operadores básicos son `Meter` y `Sacar` elementos de la pila, y los operadores auxiliares asociados son:

- `Iniciar_pila`: crea una pila o limpia una existente inicializándola a una pila vacía
- `Pila_Vacia`: consultar si la pila está vacía
- `Pila_Llena`: consultar si la pila está llena
- `Consultar_Pila`: consultar el contenido de la cima de la pila sin sacar el elemento de la misma

El carácter dinámico de la pila se debe a que el tamaño de la pila es variable, independientemente de la implementación.

3.1.- Implementación de pilas mediante arreglos

La pila vendrá representada por un registro con dos campos, uno el arreglo en que se almacenarán los elementos de la pila y otro, denominado cima, que almacenará la posición del tope de la pila, es decir, la posición del último elemento introducido en la pila.

```
TYPE  TipoIndice= [1..MAXPILA];
      Tipo_datos= INTEGER;
      Tipo_pila = RECORD
                     datos : ARRAY TipoIndice OF Tipo_datos;
                     cima : INTEGER;
                  END;
```

Se toma como criterio que cima indica la posición del último elemento introducido. Cuando cima es cero indica que la pila está vacía:

```
PROCEDURE Inicia_pila (VAR pila: Tipo_pila);
BEGIN
   pila.cima:= 0;
END Inicia_pila;
```

Para introducir un nuevo elemento habrá que incrementar primero la cima y seguidamente introducir el elemento:

```
PROCEDURE Meter_pila (VAR pila: Tipo_pila; nuevo_dato: Tipo_datos);
BEGIN
   pila.cima := pila.cima+1;
   pila.datos[pila.cima] := nuevo_dato;
END Meter_pila;
```

En la operación `Sacar` primero se toma el elemento al que apunta la cima y posteriormente se decrementa:

```
PROCEDURE Sacar_pila ( VAR pila: Tipo_pila; VAR dato : Tipo_datos);
BEGIN
   dato := pila.datos[pila.cima];
   pila.cima := pila.cima-1;
END Sacar_pila;
```

No debe confundirse la pila con el arreglo utilizado para implementarla. El arreglo puede contener elementos y sin embargo la pila estar vacía, es decir, puede haber elementos en el arreglo que no formen parte de la pila.

Operaciones auxiliares de consulta de la pila implementada mediante arreglos:

```
PROCEDURE Pila_vacia (pila:Tipo_pila):BOOLEAN;
BEGIN
   RETURN pila.cima = 0;
END Pila_vacia;

(*********************************************************)

PROCEDURE Pila_llena (pila:Tipo_pila) : BOOLEAN;
BEGIN
   RETURN pila.cima = MAXPILA
END Pila_llena;

(*********************************************************)

PROCEDURE Consultar_pila(pila:Tipo_pila; VAR dato: Tipo_datos);
BEGIN
   dato:=pila.datos[pila.cima]
END Consultar_pila;

(*********************************************************)

PROCEDURE Imprimir_pila(pila:Tipo_pila);
VAR
   i:INTEGER;
BEGIN
   FOR i:=1 TO pila.cima DO
      WriteInt(pila.datos[i],6);
      WriteLn;
   END
END Imprimir_pila;
```

3.2.- Implementación de pilas mediante listas enlazadas

La implementación es inmediata utilizando listas enlazadas

```
TYPE
   Tipo_pila = POINTER TO Nodopila;
   Tipo_datos = INTEGER;
   Nodopila = RECORD
                  datos : Tipo_datos;
                  enlace : Tipo_pila;
              END;
```

Se accede a la pila mediante el puntero externo que apunta a la lista enlazada. La cima de la pila es obviamente la cabeza de la lista enlazada. Así, la operación `Meter` se implementa de la misma forma que se hacía en la inserción por la cabeza de la lista enlazada, y `Sacar` es equivalente a sacar por la cabeza.

```
PROCEDURE Meter_pila (VAR pila: Tipo_pila; nuevo_dato : Tipo_datos);
VAR
   Nuevo_nodo : Tipo_pila;
BEGIN
   ALLOCATE(Nuevo_nodo,SIZE(Nodopila));
   Nuevo_nodo^.datos := nuevo_dato;
   Nuevo_nodo^.enlace := pila;
   pila := Nuevo_nodo;
END Meter_pila;

(*******************************************************)

PROCEDURE Sacar_pila (VAR pila: Tipo_pila; VAR dato : Tipo_datos);
VAR
  Aux : Tipo_pila;
BEGIN
   Aux := pila;
   dato := pila^.datos;
   pila := pila^.enlace;
   DEALLOCATE(Aux,SIZE(Nodopila));
END Sacar_pila;
```

Para inicializar la pila bastará que el puntero externo sea NIL y para comprobar si está vacía habrá que consultar si tiene este valor. Respecto a la operación `Pila_llena` no debe obviarse aunque la implementación sea dinámica. En este caso significaría que no existe memoria libre suficiente para crear un nuevo nodo.

```
PROCEDURE Inicia_pila (VAR pila: Tipo_pila);
BEGIN
   pila := NIL;
END Inicia_pila;


(*******************************************************)

PROCEDURE Pila_vacia (pila:Tipo_pila) : BOOLEAN;
BEGIN
   RETURN pila = NIL;
END Pila_vacia;


(*******************************************************)

PROCEDURE Pila_llena (pila:Tipo_pila) : BOOLEAN;
BEGIN
   RETURN ~Available(SIZE(Nodopila));
END Pila_llena;

(*******************************************************)


PROCEDURE Consultar_pila(pila: Tipo_pila; VAR dato: Tipo_datos);
BEGIN
   IF pila<>NIL THEN
      dato:=pila^.datos;
   END
END Consultar_pila;

(*******************************************************)


PROCEDURE Imprimir (VAR pila: Tipo_pila);
VAR
   Actual : Tipo_pila;

BEGIN
   Actual := pila;
   WHILE Actual <> NIL DO
      WriteInt(Actual^.datos,6);
      Actual := Actual^.enlace;
   END;
   WriteLn;
END Imprimir;
```

4.- COLAS (FIFO: primero en entrar – primero en salir)

Los elementos se añaden por un extremo, denominado final de la cola, pero en este caso se suprimen por el otro, conocido como principio de la cola.

Aplicaciones típicas de colas son todas aquellas en las que debamos sincronizar la velocidad de llegada de datos con la velocidad de procesamiento.

Una cola es un TDA dinámico homogéneo con acceso de tipo FIFO al que sólo se tiene acceso al principio de la cola para sacar elementos, y al final de la misma para meterlos. Los operadores básicos asociados son `Meter` y `Sacar` elementos de la cola, y los operadores auxiliares asociados son:

- `Inicia_cola`: crea una cola o limpia una existente inicializándola a una cola vacía
- `Cola_Vacia`: consultar si la cola está vacía
- `Cola_Llena`: consultar si la cola está llena
- `Consultar_Cola`: consultar el contenido del principio de la cola sin sacar el elemento de la misma.

La operación `Cola_Vacia` será necesaria cuando se vaya a sacar un elemento de la cola ya que no puede sacarse elementos de una cola vacía, y `Cola_Llena` cuando se vaya a meter un elemento en la cola.

4.1.- Implementación mediante arreglos

En un principio se podría pensar en utilizar un único índice que almacene el final, presuponiendo que el primer elemento está siempre en la primera posición del arreglo. Esto resulta inadecuado por ineficaz ya que al sacar un elemento por el principio de la cola deberemos seguidamente mover todos los elementos del arreglo una posición hacia el principio.

Es conveniente utilizar dos índices, uno que indique el final (*fi*) y otro el frente (*fr*) de la cola. Con esta implementación, las posiciones del arreglo en las que se han introducido elementos y posteriormente se han sacado quedan inutilizadas. Este comportamiento hace que se alcance el final del arreglo y no se puedan añadir elementos aunque haya numerosas posiciones libres al comienzo del mismo.

Para aprovechar las posiciones del arreglo que han sido utilizadas y están disponibles se vuelve a comenzar por el principio una vez que se ha llegado al final. Estamos considerando el arreglo como si fuera circular, de manera que el elemento que sigue al último es el primero.

Deberemos tener en cuenta que la cola puede quedar vacía o llena en posiciones intermedias del arreglo. Tenemos que establecer criterios para los apuntadores que nos permitan identificar de forma inequívoca estas dos situaciones.

Si el criterio fuera:

- Final (*fi*) apunta al último elemento introducido, y aumenta cada vez que se introduce un nuevo elemento en la cola
- Frente (*fr*) apunta al primer elemento a sacar, y aumenta al eliminar un elemento de la cola

Supongamos que hay un solo elemento (*fr=fi*) y que vamos a eliminarlo. Entonces *fr=sig(fi)* , es decir *fr=1* si *fi=MAXCOLA* o *fr=fi+1* en otro caso. Esta será la condición de cola vacía.

Supongamos ahora que a la cola le falta un solo elemento para estar llena y tras una nueva inserción va a quedar llena. En esta situación entre ambos índices hay una posición libre (*fr=fi+2* ya que *fi* apunta al último elemento, *fi+1* la posición vacía, y por tanto *fi+2* será el principio de la cola). Cuando el nuevo elemento es introducido se incrementa el índice final (*fi*). La condición de cola llena será también *fr=sig(fi)*. Por tanto utilizando este criterio no es posible distinguir cuando la cola está vacía o llena.

Una posible solución sería mantener un contador que nos indique los elementos que hay en la cola, pero sería mejor una solución que evite realizar operaciones adicionales. Para ello tendremos que cambiar los criterios.

Establecemos:	*Condición cola vacía:*	*fr = fi*
	Condición cola llena:	*Si fi = MAXCOLA fr=1* *Si fi<>MAXCOLA fr=fi+1*
Criterio1:	El frente, *fr*, apunta al elemento anterior al primer elemento de la cola El final, *fi*, apunta al último elemento introducido en la cola	
(como *fr* apunta a un elemento que nunca se rellena, la cola estará llena dejando una posición libre en el arreglo).		

Con este criterio, la estructura consistirá en un registro con el arreglo en el campo de datos, y los dos apuntadores *fr* y *fi*. Por ejemplo:

```
CONST   MAXCOLAN=10;
        MAXCOLA=MAXCOLAN+1;

TYPE
       TipoIndice = [1..MAXCOLA];
       Tipo_datos= INTEGER;
       Tipo_cola = RECORD
                      datos : ARRAY TipoIndice OF Tipo_datos;
                      fr,fi : TipoIndice;
                   END;
VAR cola:Tipo_cola
```

Los operadores básicos se implementan de forma inmediata atendiendo al criterio utilizado y al carácter circular de la cola. Habrá que considerar dos situaciones, una cuando las operaciones se realizan en los extremos del arreglo sobre el que se implementa la cola y otra cuando se realizan en posiciones intermedias.

Para implementar el operador `Meter_Cola` se manipulará el índice final (*fi*): para introducir el nuevo elemento habrá que incrementar primero *fi* y seguidamente introducir el nuevo dato. Al incrementar el índice *fi* debemos tener en cuenta dos situaciones: si éste se encuentra apuntando al fina de la cola (*cola.fi=MAXCOLA*), el nuevo elemento habrá que introducirlo en la primera posición (*cola.fi := 1*), y en cualquier otro caso bastará incrementar el índice *fi* (*cola.fi := cola.fi + 1*).
Parecidos argumentos deberemos tener en cuenta al implementar el operador `Sacar` respecto al índice *fr* distinguiendo si dicho índice se encuentra al final o en posiciones intermedias.

```
PROCEDURE Meter_cola(VAR cola:Tipo_cola; nuevo_dato:Tipo_datos);
BEGIN
   IF cola.fi=MAXCOLA THEN cola.fi:=1
   ELSE cola.fi:=cola.fi+1
   END;
   cola.datos[cola.fi]:=nuevo_dato
END Meter_cola;

PROCEDURE Sacar_cola (VAR cola: Tipo_cola; VAR dato: Tipo_datos);
BEGIN
   IF cola.fr=MAXCOLA THEN cola.fr:=1
   ELSE cola.fr:=cola.fr+1
   END;
   dato:=cola.datos[cola.fr]
END Sacar_cola;
```

Respecto a los operadores auxiliares `Iniciar` debe satisfacer `cola.fr = cola.fi` y dado que el arreglo es circular, cualquier elemento es igualmente válido para imponer esta condición.
Si adoptamos el criterio de que partiendo de una cola vacía, el primer elemento se introducirá en la primera posición del arreglo la cola se inicializará con *cola.fr = cola.fi = MAXCOLA*

El operador `Cola_Vacia` comprobará si *fr = fi*, y `Cola_Llena` comprobará si *cola.fr = cola.fi +1* en las posiciones intermedias y cuando *cola.fi = MAXCOLA*, la cola estará llena si *cola.fr = 1*

```
PROCEDURE Inicia_cola (VAR cola:Tipo_cola);
BEGIN
   cola.fr:=MAXCOLA;
   cola.fi:=MAXCOLA
END Inicia_cola;

(*********************************************************)

PROCEDURE Cola_vacia (cola:Tipo_cola): BOOLEAN;
BEGIN
   RETURN cola.fi=cola.fr;
END Cola_vacia;

(*********************************************************)

PROCEDURE Cola_llena (cola:Tipo_cola): BOOLEAN;
BEGIN
   IF cola.fi=MAXCOLA THEN
       RETURN cola.fr=1
   ELSE
       RETURN cola.fr=cola.fi+1;
   END
END Cola_llena;

(*********************************************************)

PROCEDURE Consultar_final_cola(cola: Tipo_cola; VAR dato: Tipo_datos);
BEGIN
   dato:=cola.datos[cola.fi];
END Consultar_final_cola;

(*********************************************************)

PROCEDURE Consultar_frente_cola(cola: Tipo_cola;VAR dato: Tipo_datos);
BEGIN
   IF cola.fr=MAXCOLA THEN dato:=cola.datos[1]
   ELSE dato:=cola.datos[cola.fr+1]
   END;
END Consultar_frente_cola;

(*********************************************************)

PROCEDURE Imprimir_cola(cola:Tipo_cola);      (IMPLEMENTACIÓN PROPIA)
BEGIN
   WHILE cola.fr<>cola.fi DO
     IF cola.fr=MAXCOLA THEN
        cola.fr:=1
     ELSE cola.fr:=cola.fr+1;
     END;
     WriteInt(cola.datos[cola.fr],6);
     WriteLn
   END;
END Imprimir_cola;
```

Las condiciones de distinguibilidad de la cola vacía y cola llena también se satisfacen con el siguiente criterio:

Criterio2: El frente, *fr*, apunta al primer elemento de la cola
El final, *fi* apunta al elemento siguiente al último introducido en la cola

Lógicamente la implementación de operadores básicos y auxiliares será distinta, pero una vez implementados se utilizarán de la misma manera.

4.2.- Implementación de colas mediante listas enlazadas

Modificaremos la lista enlazada dotándola de un puntero externo más que apunte al final de la misma.

```
TYPE
   Ptr_nodo = POINTER TO Nodo;
   Tipo_datos=INTEGER;
   Nodo = RECORD
             datos : Tipo_datos;
             enlace : Ptr_nodo;
          END;
   Tipo_cola = RECORD
                  frente,final: Ptr_nodo;
               END;
```

Respecto a las operaciones `Meter` y `Sacar` hemos de tener en cuenta que añadimos por el final de la lista y eliminamos por el principio de la misma.
Estableciendo como criterio que la cola está vacía cuando el puntero final sea *NIL*, la inicialización de la cola se realiza simplemente asignando *NIL* a `cola.final`

`Meter` se realizará aumentando el puntero final: en primer lugar se crea un nuevo nodo, se introduce el dato en su campo de datos y se establece *NIL* en su enlace. Seguidamente se distinguen dos situaciones: si la cola ya contiene elementos se enlaza el último nodo de la lista con el nuevo nodo (`cola.final^.enlace:=Nuevo_nodo`). Si la cola está vacía es el puntero externo frente el que debe apuntar a este nuevo nodo. Por último, se asigna la dirección del nuevo nodo al puntero externo final independientemente del estado inicial de la cola.

`Sacar` toma los datos de la variable referenciada a la que apunta `cola.frente` y asigna el puntero externo frente al siguiente nodo (`cola.frente := cola.frente^.enlace`). Nuevamente hay que distinguir la situación de elemento único (al sacarlo la cola quedará vacía) de la situación en que la lista tiene más de un elemento. En el primer caso debe asignarse *NIL* al puntero externo final. Finalmente, se libera el nodo con la ayuda de un puntero auxiliar (*aux*)

```
PROCEDURE Meter_cola (VAR cola : Tipo_cola; nuevo_dato : Tipo_datos);
VAR
   Nuevo_nodo : Ptr_nodo;
BEGIN
   ALLOCATE(Nuevo_nodo,SIZE(Nodo));
   Nuevo_nodo^.datos := nuevo_dato;
   Nuevo_nodo^.enlace := NIL;
   IF cola.final=NIL THEN
      cola.frente := Nuevo_nodo
   ELSE
      cola.final^.enlace := Nuevo_nodo
   END;
   cola.final := Nuevo_nodo
END Meter_cola;

(*******************************************************)
PROCEDURE Sacar_cola (VAR cola: Tipo_cola; VAR dato : Tipo_datos);
VAR
   aux : Ptr_nodo;
BEGIN
   aux := cola.frente;
   dato := cola.frente^.datos;
   cola.frente := cola.frente^.enlace;
   IF cola.frente = NIL THEN
      cola.final := NIL
   END;
   DEALLOCATE(aux,SIZE(Nodo));
END Sacar_cola;
(*******************************************************)

PROCEDURE Inicia_cola (VAR cola: Tipo_cola);
BEGIN
   cola.final := NIL;
END Inicia_cola;
```

```
(*********************************************************)

PROCEDURE Cola_vacia (cola:Tipo_cola) : BOOLEAN;
BEGIN
   RETURN cola.final = NIL;
END Cola_vacia;

(*********************************************************)

PROCEDURE Cola_llena (cola:Tipo_cola) : BOOLEAN;
BEGIN
   RETURN ~Available(SIZE(Nodo));
END Cola_llena;

(*********************************************************)

PROCEDURE Consultar_frente_cola(cola:Tipo_cola; VAR dato: Tipo_datos);
BEGIN
   IF cola.frente#NIL THEN
      dato:=cola.frente^.datos;
   END;
END Consultar_frente_cola;

(*********************************************************)

PROCEDURE Consultar_final_cola(cola:Tipo_cola; VAR dato: Tipo_datos);
BEGIN
   IF cola.final#NIL THEN
      dato:= cola.final^.datos;
   END;
END Consultar_final_cola;

(*********************************************************)

PROCEDURE Imprimir_cola (VAR cola: Tipo_cola);
VAR
   Actual : Ptr_nodo;
BEGIN
   Actual := cola.frente;
   WHILE Actual <> NIL DO
      WriteInt(Actual^.datos,6);
      Actual := Actual^.enlace;
   END;
   WriteLn;
END Imprimir_cola;
```

Consideremos la posibilidad de implementar una cola haciendo uso de una lista doblemente enlazada con dos punteros externos (frente y final). Esto no representa ninguna ventaja ni desventaja en relación al coste computacional respecto a la implementación anterior. Sin embargo resulta inadecuado siguiendo el criterio de economía de almacenamiento. Además, al tener dos enlaces por nodo, la actualización de punteros requerirá el doble de operaciones. Por tanto, sería una solución incorrecta aunque funcione.

5.- Consideraciones generales:

En ocasiones, se han encontrado soluciones que no son adecuadas aunque funcionen. No todo lo que funciona es la solución correcta.

Los TDA son estáticos o dinámicos dependiendo exclusivamente de su definición, nunca de su implementación. Así, en la implementación estática no debe confundirse la pila con el arreglo

Las implementaciones estáticas tienen la ventaja de que son más rápidas.

La clave fundamental para decidir entre una implementación estática o dinámica es si se conoce o no a priori un número máximo de datos que deberán ser almacenados.

Debe tenerse en cuenta que la utilización de un TDA dinámico para resolver un problema en el que el número de elementos a almacenar es variable no es siempre adecuada.

Hemos de tener mucho cuidado en `Sacar` todos los elementos hasta que la pila o la cola esté vacía en la operación `Vaciar`. En las implementaciones con arreglos `Vaciar` puede implementarse mediante una simple llamada a `Iniciar`. Sin embargo, hacer esto con una implementación dinámica tiene resultados catastróficos puesto que la lista enlazada sobre la que se implementó la pila o la cola no ha sido liberada, ocupa memoria y no estará ya accesible para liberarla. Este es un error muy grave.

Una solución elegante es definir un nuevo TDA en el que se incorpore la operación `Vaciar` entre las básicas, llamándola por ejemplo "`Pila con vaciado`", o "`Cola con vaciado`".

2.- CLASIFICACIÓN EN MEMORIA PRINCIPAL (ordenación interna)

En lo que vamos a ver a continuación las colecciones de elementos a clasificar se almacenarán en arreglos:

```
TYPE Tipo_datos = RECORD
                    llave : Tipo_llave;
                    (*otros componentes*)
                  END

VAR  a: ARRAY [1..n]  OF  Tipo_datos;
```

El caso general será el de un arreglo de registros con varios campos donde uno de ellos se utiliza como campo llave.

2.1.- Metodos directos de clasificación

A.- Clasificación por inserción: consiste en insertar un elemento dado en el lugar que le corresponda

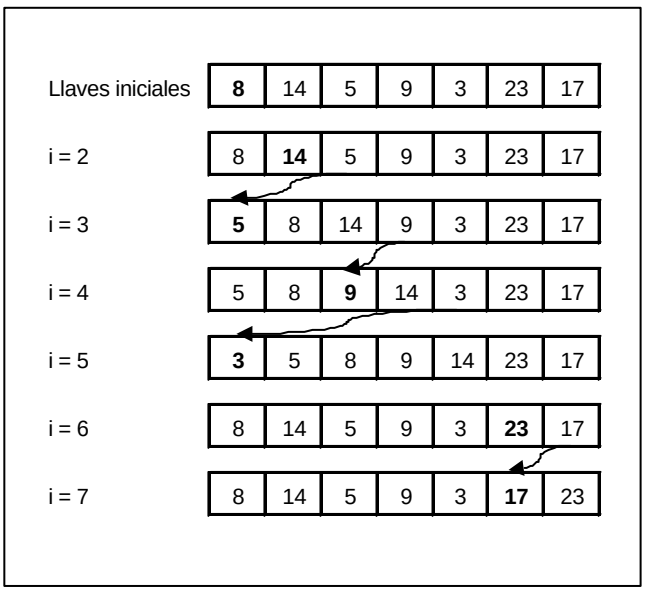
A1.- Clasificación por inserción directa:

En un primer paso se considera que la parte ordenada esta formada por el primer elemento del arreglo. El procedimiento será

- Tomar un elemento en la posición i ($i= 2,3,\ldots n$)
- Buscar su lugar en las posiciones anteriores (parte ordenada)
- Mover hacia la derecha los restantes
- Insertarlo.

Se requerirá una variable temporal que almacene el elemento a insertar. Si utilizamos a[0] como variable temporal podemos además utilizar este elemento como centinela en el bucle.

```
PROCEDURE inserdir(VAR a: Tipoarray);
   VAR
      i,j: Tipo_indice;
   BEGIN
     FOR i:=2 TO n DO
       a[0]:=a[i];(* Garantiza la salida del while *)
       j:=i;
         WHILE a[0]<a[j-1] DO
            a[j]:=a[j-1]; (* movimientos *)
            j:=j-1
         END;
       a[j]:=a[0]; (* inserción *)
     END
   END;
```



- **Análisis:**

El análisis del algoritmo requiere el cálculo del número de comparaciones y de movimientos en los casos mejor, peor y promedio.

El mejor caso se tendrá cuando el arreglo está ordenado inicialmente, el peor cuando esté ordenado en sentido contrario, y el caso promedio cuando esté ordenado aleatoriamente.

- ***Comparaciones:***

a) *Mejor caso*: La comparación (`a[0]<a[j-1]`) sólo se realiza una vez en cada paso. Como el bucle FOR se realiza n-1 veces:

$$C_{min} = n\text{-}1$$

b) *Peor caso*: En el primer pase (*i=2*) se realiza dos comparaciones: a[1] con a[0] y a[0] con a[0] (acción centinela). En el segundo pase se realizan tres comparaciones, y en el paso *n-1* se realizan *n* comparaciones:

$$C_{max} = 2+3+4+\ldots+n = 1+2+3+\ldots+n - 1 = (n^2 + n - 2) / 2$$

c) *Caso promedio*: La probabilidad de que dos números enteros cualesquiera estén ordenados es *1 / 2***.** Por tanto, el número de comparaciones promedio será:

$$Cprom = C_{max} / 2 = (n^2 + n - 2) / 4$$

- ***Movimientos***

a) *Mejor caso*: nunca se entra en bucle *WHILE*, sin embargo el bucle *FOR* se realiza *n-1* veces, y dentro de él se realizan *dos movimientos* (a[0] := a[i] y a[j] := a[0]). Por tanto:

$$M_{min} = 2(n\text{-}1)$$

b) *Peor caso*: además de los *2(n-1)* movimientos internos al bucle *FOR* se realizará *una* asignación (a[j] := a[j-1]) cada vez que se entre en el *WHILE*. Así, para *i=2* se realiza *un* movimiento, para *i=3* habrá *dos* movimientos, … y para *i=n* se realizan *n-1* movimientos:
$M_{max(while)} = 1+2+3+\ldots+ n\text{-}1 = (n^2 - n) / 2$ $M_{max} = 2(n\text{-}1)+(n^2 - n) / 2 = (n^2 + 3n - 4) / 2$

c) *Caso promedio*: el número de movimientos promedio en el *WHILE* será $M_{prom\,(while)} = C_{prom}$ y sumando los *2(n-1)* movimientos que se realizan fuera del *WHILE* $M_{prom} = (n^2+9n-10)/4$

A2.- Clasificación por inserción binaria:

Mejora el método de inserción. Para ello se toma el elemento que ocupa la posición central de la parte ordenada y se compara con el elemento a insertar.. Si es mayor se descarta la parte izquierda y si es menor la derecha. Aplicando sucesivamente el mismo proceso se obtiene la posición donde se debe insertar:

- Tomar un elemento de la posición *i*
- Buscar dicotómicamente su lugar en las posiciones anteriores
- Mover hacia la derecha los restantes
- Insertarlo

```
PROCEDURE inserbin(VAR a: Tipoarray);
   VAR
      i,j,m,L,R: Tipo_indice;x: Tipo_datos;
BEGIN
   FOR i:=2 TO n DO
        x:=a[i]; (*L:índice al elemento de la parte izquierda del subarray en el que
                   se realiza la búsqueda en cada paso*)
        L:=1;    (*R:índice al elemento de la parte derecha del subarray en el que
                   se realiza la búsqueda en cada paso*)
        R:=i;
        WHILE L<R DO
           m:=(L+R)DIV 2;
           IF a[m]<=x THEN
              L:=m+1
```

```
            ELSE
               R:=m
            END
         END;
         FOR j:=i TO R+1 BY -1 DO
            a[j]:=a[j-1](* desplazamiento *)
         END;
         a[R]:=x; (* inserción *)
   END
END
```

Análisis:
El número de movimientos no mejora significativamente y seguirá siendo de orden n^2.
El número de comparaciones no depende significativamente del orden inicial de los datos a clasificar.
Al realizar la búsqueda binaria se divide una longitud n por la mitad un número N de veces, siendo $N = \log n$ (logaritmo en base 2).
Así, para una longitud i se tiene $N_i = \log i$. Por tanto, el número de comparaciones será $C = \sum_{i=1}^{n} \log i$, que aproximando por la integral se tiene $C = \int_1^n \log x \, dx$. Realizando los cálculos matemáticos oportunos así como las transformaciones necesaria se llega a $C = n(\log n - \log e) + \log e$, siendo C del orden $n \log n$

B.- Clasificación por selección directa:

En un primer paso se recorre el arreglo hasta encontrar el elemento menor que se intercambia con el de la primera posición. Seguidamente se considera únicamente la parte del arreglo no ordenada y se repite el proceso:

- Seleccionar el elemento menor de la parte del arreglo no ordenada
- Colocarlo en la primera posición de la parte no ordenada del arreglo.

```
PROCEDURE selecdir(VAR a:Tipoarray);
   VAR
      i,j,k: Tipo_indice; x: Tipo_datos;
BEGIN
   FOR i:=1 TO n-1 DO
        k:=i;
        x:=a[i];        (* selección inicial *)
        FOR j:=i+1 TO n DO      (* búsqueda en secuencia fuente *)
           IF a[j]<x THEN
              k:=j;
              x:=a[k] (* selección *)
           END
        END;
        a[k]:=a[i];     (*intercambio*)
        a[i]:=x;
   END
END selecdir;
```

Análisis:
El número de comparaciones es independiente del orden inicial de las llaves del arreglo, sin embargo para los movimientos sí podemos distinguir diferentes casos: cuando el arreglo está completamente ordenado no habrá ninguna sustitución, mientras que si el arreglo está en orden inverso, todos los elementos serán elegidos como menor y serán asignados a la variable temporal.

Llaves iniciales	**8**	14	5	9	**3**	23	17
i = 2	3	**14**	**5**	9	8	23	17
i = 3	3	5	**14**	9	**8**	23	17
i = 4	3	5	8	**9**	14	23	17
i = 5	3	5	8	9	**14**	23	17
i = 6	3	5	8	9	14	**23**	**17**
Llaves finales	3	5	8	9	14	17	23

- *Comparaciones*: La comparación (a[j]<*x*) se realizará n-1 veces para la primera elección, *n-2* para la segunda y así sucesivamente. Tendremos: $C = 1+2+3+\ldots+n-1 = (n^2-n)/2$

- *Movimientos*

a) <u>*Mejor caso*</u>: nunca se entra en el *IF*, por tanto, el movimiento (*x*:=a[k]) no se raliza nunca, sin embargo el bucle *FOR* se realiza *n-1* veces, y dentro de él se realizan *tres movimientos* (*x* := a[i], a[k] := a[i] y a[i] := *x*). Por tanto:

$$M_{min} = 3(n-1)$$

b) <u>*Peor caso*</u>: además de los *3(n-1)* movimientos internos al bucle *FOR* se realizará *una* asignación (*x* := a[k]) cada vez que se entre en el *IF*. Obsérvese que al intercambiar la primera posición con el menor (última posición) ambos quedan ya colocados. De esta manera, se van ordenando de dos en dos con lo que el máximo de ordenaciones de parejas es *n/2*.Así, el máximo de movimientos será:

$M_{max(if)} = 1+3+5+\ldots+ n/2$ $\qquad\qquad$ $M_{max} = 3(n-1)+n^2/4$

***Comentario personal**: El mismo razonamiento lo hago de la forma siguiente. Para* i=1, *la asignación* x:=a[k] *se ejecutará* (n-1) *veces. Después se realiza el intercambio entre la primera y última posición del arreglo, con lo que* a[1] *y* a[n] *ya "estarán en su sitio". Por ejemplo, si tuvieramos 6,5,4,3,2,1 obtendríamos 1,5,4,3,2,6. Para* i=2 *se realizarán* (n-3) *asignaciones (ya que para j=n no se cumple la condición a[j]<x). Se intercambarán el segundo y penúltimo elemento que quedarán ordenados. Para* i=3 *tendremos* (n-5) *asignaciones y así sucesivamente. Si el número de elementos del vector es par cuando se llegue a las posiciones centrales se ejecutará una asignación, y por consiguiente el último intercambio. Se habrán realizado* $M_{max(if)} = (n-1)+(n-3)+(n-5)+\ldots+1 = n^2/4$ *ya que se trata de la suma de* n/2 *términos de una progresión aritmética de diferencia* 2. *Ahora podemos concluir* $M_{max} = 3(n-1)+n^2/4$.

Hemos supuesto un número par de elementos. En caso de ser impar el resultado sería el mismo ya que el elemento central está en su sitio y por tanto no influiría. Ejemplo: en 7,6,5,4,3,2,1 se hacen los mismos movimientos que en 6,5,4,3,2,1

c) <u>*Caso promedio*</u>: El algoritmo sólo realiza el intercambio de elementos cuando encontramos, en la parte aún sin ordenar, un elemento menor que *x*. La probabilidad de que un entero sea menor que otro es *1/2*, la probabilidad de que un entero sea el menor de tres dados es *1/3*, y la de que un entero sea el menor de *i* dados *1/i* Por tanto $M_{prom,i}=1/2+1/3+\ldots+1/i=Hi-1$ donde *Hi* es el *i-ésimo* número armónico. $M_{prom} = \sum_{i=1}^{n} M_{prom,i}$. Aproximando por la integral y realizando las oportunas transformaciones matemáticas resulta $M_{prom} = n(\ln n+\gamma)$ donde γ es la constante de Euler.

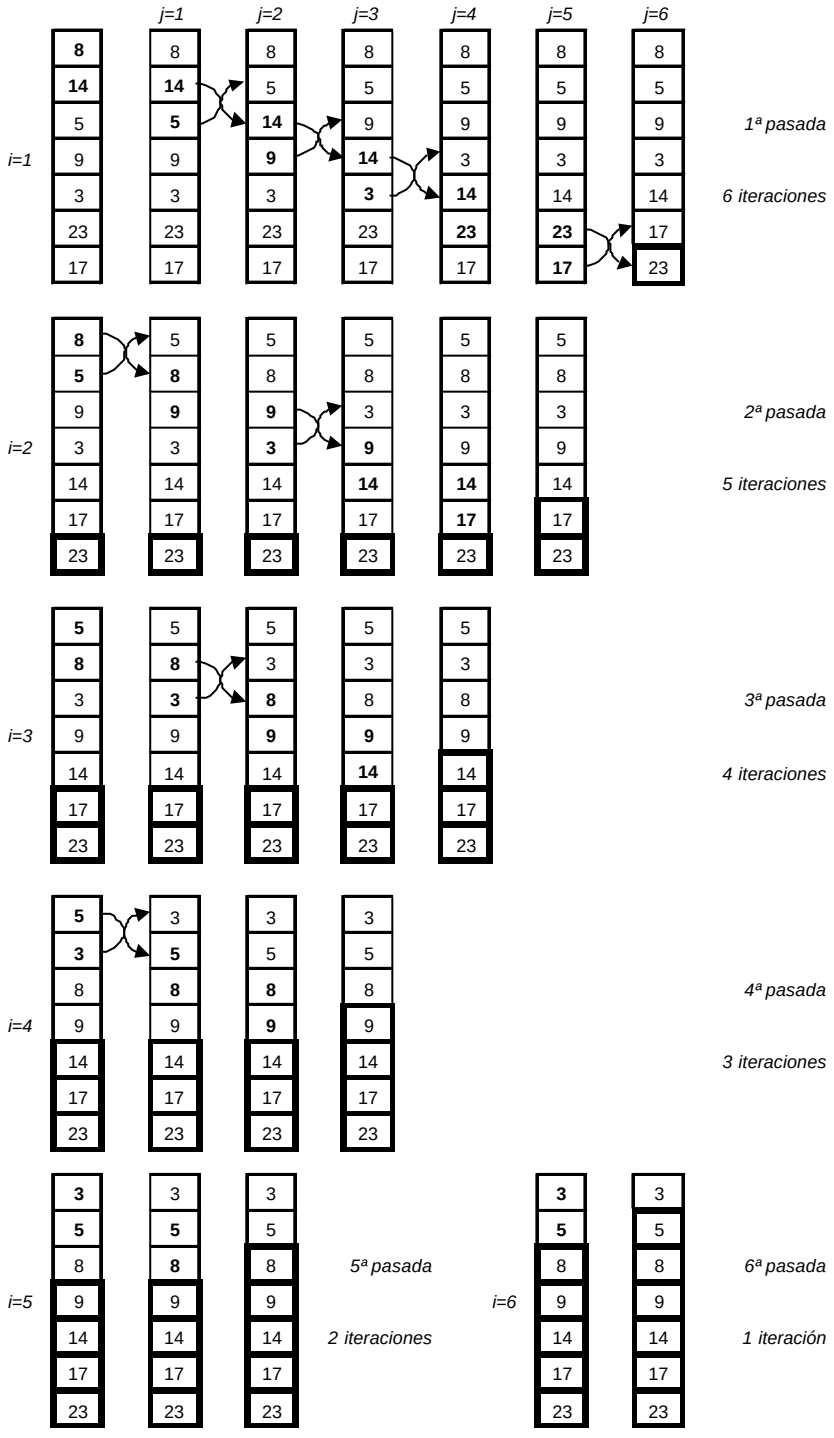
Conclusión:

El método de selección directa es preferible al de inserción directa ya que su número de movimientos promedio es de orden $n \ln n$.
En general, la implementación de una comparación tiene un coste inferior al de un movimiento. Por tanto, la selección directa será la opción a elegir en general, aunque la inserción es algo más rápida cuando las llaves se clasifican predominantemente al principio.

C.- Clasificación por intercambio

C.1.- Clasificación por intercambio directo (burbuja):

El algoritmo puede describirse de la siguiente forma: Se comparan pares de elementos contiguos y se intercambian si están desordenados.



```
PROCEDURE burbuja(VAR a: Tipoarray);
   VAR
      i,j: Tipo_indice; aux: Tipo_datos;
BEGIN
   FOR i:=1 TO n-1 DO
        FOR j:=1 TO n-i DO
           IF a[j]>a[j+1] THEN  (*comparar elementos contiguos*)
              aux:=a[j+1];        (*intercambiar*)
              a[j+1]:=a[j];
              a[j]:=aux;
           END;
        END;
   END;
END burbuja;
```

- **Análisis:**

El número de comparaciones es independiente del orden inicial de las llaves del arreglo, mientras que para los movimientos sí se distinguen diferentes casos. Así si el arreglo está completamente ordenado no habrá ningún intercambio mientras que si está en orden inverso se realizará un intercambio con cada comparación, es decir, se realizaran todos los intercambios posibles.

- *Comparaciones*: La comparación (a[j]>a[j+1]) se realizará *n-1* veces para el primer pase (*i=2*), *n-2* en el segundo (*i=3*) y así sucesivamente. Tendremos: $C = 1+2+3+\ldots+n-1 = (n^2-n) / 2$

- Movimientos

a) <u>*Mejor caso*</u>: nunca se entra en el *IF*, por tanto, el intercambio formado por los tres movimientos (*aux* := a[j+1], a[j+1] := a[j] y a[j] := *aux*)no se raliza nunca. Por tanto: $M_{min} = 0$

b) <u>*Peor caso*</u>: Se realiza el intercambio siempre que se realiza la comparación. Así, el máximo de movimientos será: $M_{max} = 3C = 3(n^2-n) / 2$

c) <u>*Caso promedio*</u>: El intercambio se realiza dependiendo de que un número entero sea mayor o menor que otro. La probabilidad de que un entero sea menor que otro es 1 / 2 . Por tanto:

$$M_{prom} = 1 / 2\ M_{max} = 3(n^2-n) / 4$$

En el método de la burbuja los elementos bajan bien (varias posiciones en cada pase) pero suben mal (una posición en cada pase). Este hecho lo corrige la clasificación por sacudida.

<u>C.2.- Clasificación por sacudida o vibración:</u>

Mejora el método de la clasificación por burbuja alternando la dirección de pases consecutivos.

El segundo pase comienza en las posiciones finales y se recorre en sentido inverso al anterior.

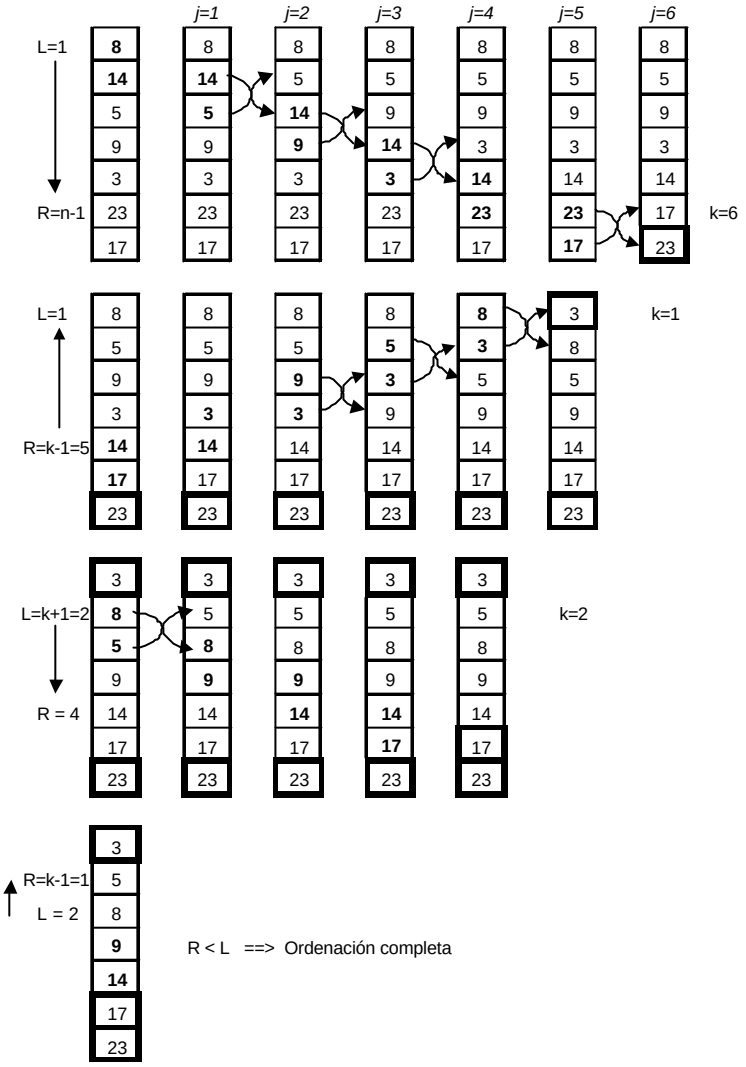
Así los pases impares se hacen en sentido descendente y los pares en sentido contrario.

La parte del arreglo a ordenar estará en todo momento marcada por los índices *L* y *R.*

En cada pasada hacia el final se decrementa el índice final (*R*) y en cada pasada hacia el inicio se incrementa el índice de inicio (*L*).

El algoritmo se para, obviamente, cuando los índices se cruzan.

```
PROCEDURE vibracion (VAR a:Tipoarray);
   VAR
      j,k,L,R: Tipo_indice;  aux: Tipo_datos;
BEGIN
   L:=1; R:=n-1; k:=1;
      REPEAT
        FOR j:=L TO R DO
           IF a[j]>a[j+1] THEN
              aux:=a[j+1];
              a[j+1]:=a[j];
              a[j]:=aux;
              k:=j
           END
        END;
        R:=k-1;
        FOR j:=R TO L BY -1 DO
           IF a[j]>a[j+1] THEN
              aux:= a[j+1];
              a[j+1]:=a[j];
              a[j]:=aux;
              k:=j;
           END;
        END;
        L:=k+1;
      UNTIL L>R;
END vibracion;
```



- **Análisis:**

No se mejora el número de movimientos a realizar. La mejora se realiza únicamente en el número de comparaciones. El número mínimo de comparaciones es $C_{min} = n-1$, el máximo $C_{max}=n-k_1\ddot{O}n$ y en el caso promedio es proporcional a $C_{prom} = 1/2(n^2-n(k_2+\ln n))$

Conclusión: Comparando los métodos directos de clasificación sobre arreglos, la selección directa será la opción a elegir en general, aunque la inserción es algo más rápida cuando las llaves predominantemente se clasifican al principio y la vibración puede ser mejor cuando el arreglo está inicialmente casi ordenado.

2.2.- Metodos avanzados de clasificación

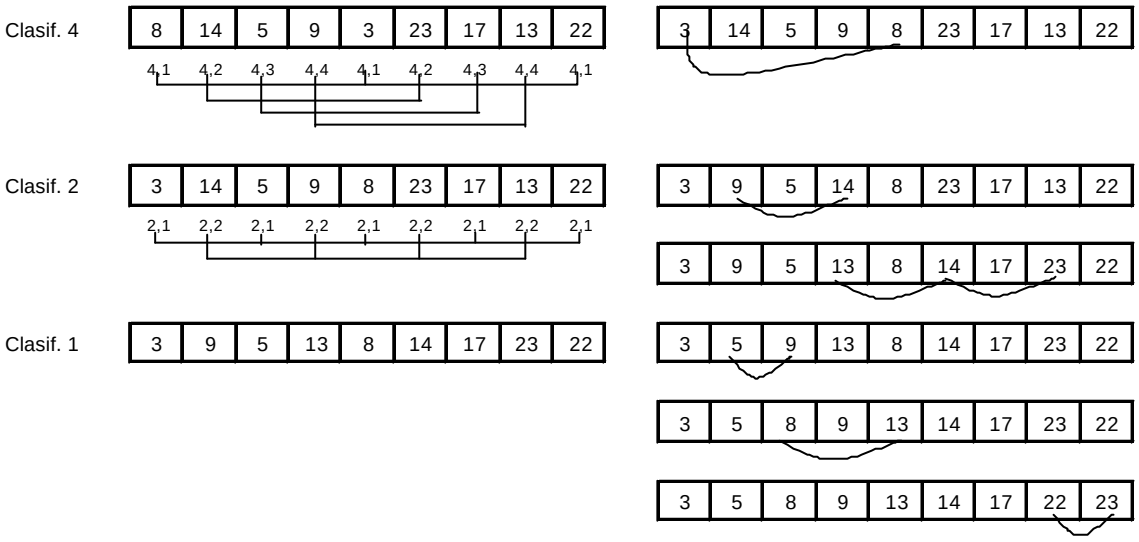
- ***Inserción por incremento decreciente (Shell)***

Es una mejora del de inserción directa. Se basa en la ordenación por inserción de los elementos que difieren h_1 posiciones, después de los que difieren h_2, y así sucesivamente según un conjunto de incrementos decrecientes h_i hasta ordenar los que difieren $h_t=1$.

Se comienza en cada caso por el *incremento* mayor que permita el tamaño del arreglo y se va reduciendo en cada actuación.

A la hora de hacer la ordenación por inserción, hay que tener en cuenta que los desplazamientos necesarios para insertar un elemento no se harán ahora de una posición a la siguiente adyacente como se hacía en el método de inserción convencional, sino que ahora los *incrementos* vienen dados por el nivel de clasificación en que nos encontremos y sólo el último paso (*Clasificación1*) se comporta de la misma forma que el algoritmo convencional por inserción. De hecho, el último paso es una clasificación por inserción directa.

Ejemplo:




```
PROCEDURE shell(VAR a: Tipoarray);
CONST
   t=3;      (*número de ordenaciones por inserción directa*)
VAR
   i,j,k,
   s: Tipo_indice;   (*posición del centinela de cada clasificación*)
   m:[1..t];          (*índice que indica el número de ordenación*)
   h:ARRAY[1..t] OF INTEGER;  (*vector de incrementos*)

BEGIN
     h[1]:=4; h[2]:=2; h[3]:=1;
     FOR m:=1 TO t DO
         k:=h[m]; s:=-k;
         FOR i:= k+1 TO n DO
            j:=i-k;
            IF s=0 THEN s:=-k END;
            s:=s+1; a[s]:=a[i];        (*Inserción directa*)
            WHILE a[s]<a[j] DO
               a[j+k]:= a[j]; j:=j-k
            END;
            a[j+k]:=a[s]
         END;
     END
END shell;
```

Análisis:

El análisis detallado requiere una elaboración matemática compleja y rigurosa en función de los posibles valores de incrementos decrecientes a elegir.

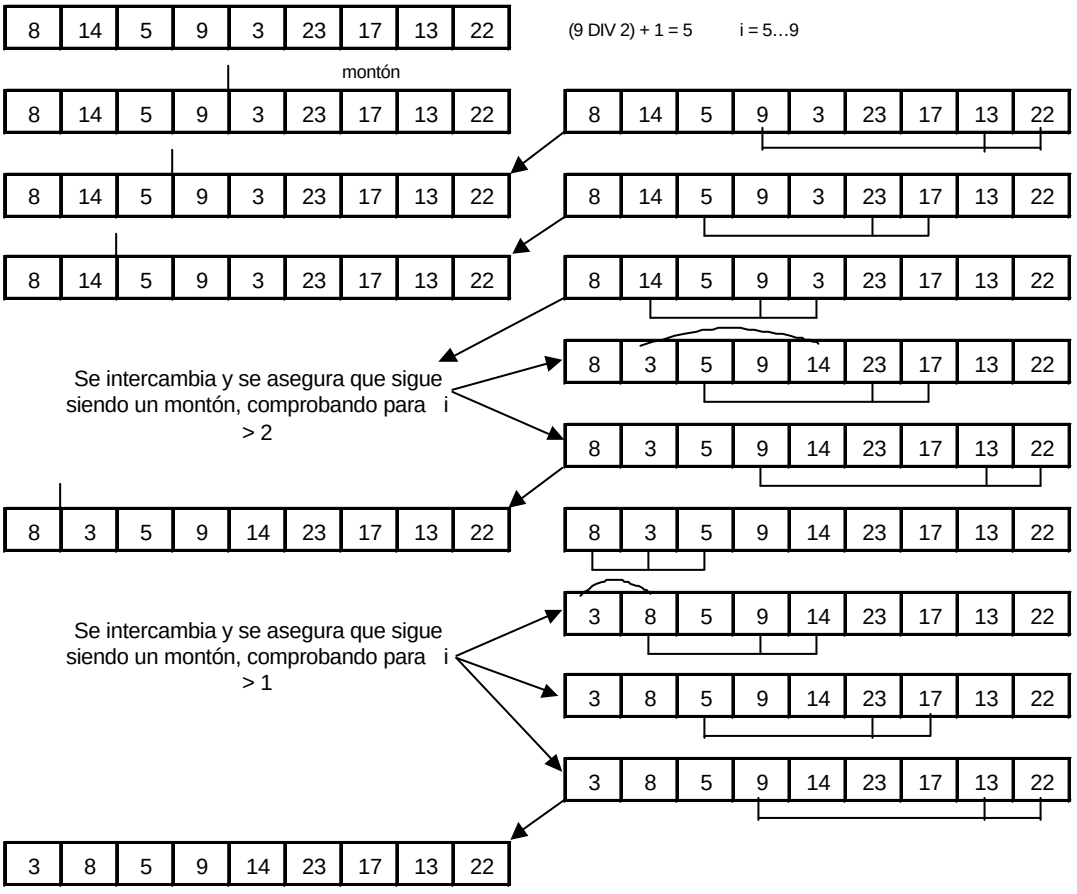
Tenemos la seguridad de que en último extremo la *Clasificación-1* realizará toda la tarea de ordenación. Este sería el peor caso. Vamos a suponer que *n* es potencia de 2 y que se encuentran *n/2* llaves con valores grandes en las posiciones pares y *n/2* con valores pequeños en las impares. Los incrementos se toman pares, excepto el último (*h=1*). Cuando se llega al último pase los *n/2* números de valores grandes están todavía en las posiciones pares, y los *n/2* menores en las impares. Para colocar en el último pase los *n/2* elementos menores en sus posiciones se necesitarán $\sum_{i=1}^{n/2} i - 1$, que es una suma aritmética y por tanto de orden n^2.

Deben evitarse conjuntos de incrementos en los que unos sean múltiplos de otros ya que no aprovechan la clasificación realizada anteriormente. La secuencia 1, 3, 7, 2^k-1 (incrementos de Hibbard), evita los múltiplos de sí mismos. El coste máximo para esta secuencia es de orden $n^{1.3}$, con lo que mejora los métodos directos.

- ***<u>Clasificación por montón</u>***

Un *montículo* o *montón* se define como una secuencia de llaves h_i , $i = L, L+1,...R$ tal que mantienen la siguiente relación de orden: $h_i \leq h_{2i}$ y $h_i \leq h_{2i+1}$ para $i = L,...,R/2$

Ejemplo de construcción de un montón:



Dado un arreglo de *n* elementos, $h_1...h_n$, los elementos a partir de la mitad del arreglo, *k=(nDIV2)+1*, hasta el final, forman un montón ($h_k...h_n$) puesto que no hay dos índices *i*,*j* en esta parte que no satisfagan la definición de montón. Seguidamente se amplía ($h_k...h_n$) tomando un elemento por la izquierda y se reconstruye el montón con este nuevo elemento y así sucesivamente.

Cada vez que se añade un elemento al montón hay que comprobar que se cumple la definición de Williams y si no, hay que *intercambiarlo*. Cada vez que se hace un intercambio hay que comprobar el resto del montón para garantizar que no se ha alterado el resto del mismo.

Una vez organizado el arreglo como un montón, el elemento h_1, es el menor de todos los elementos del montón. Para clasificar un arreglo de *n* elementos se retira la cima y se construye un montón con los *n-1* elementos restantes. Su cima será de nuevo el menor de los que quedan, se retira y se vuelve a construir un montón con los *n-2* restantes, y así sucesivamente.

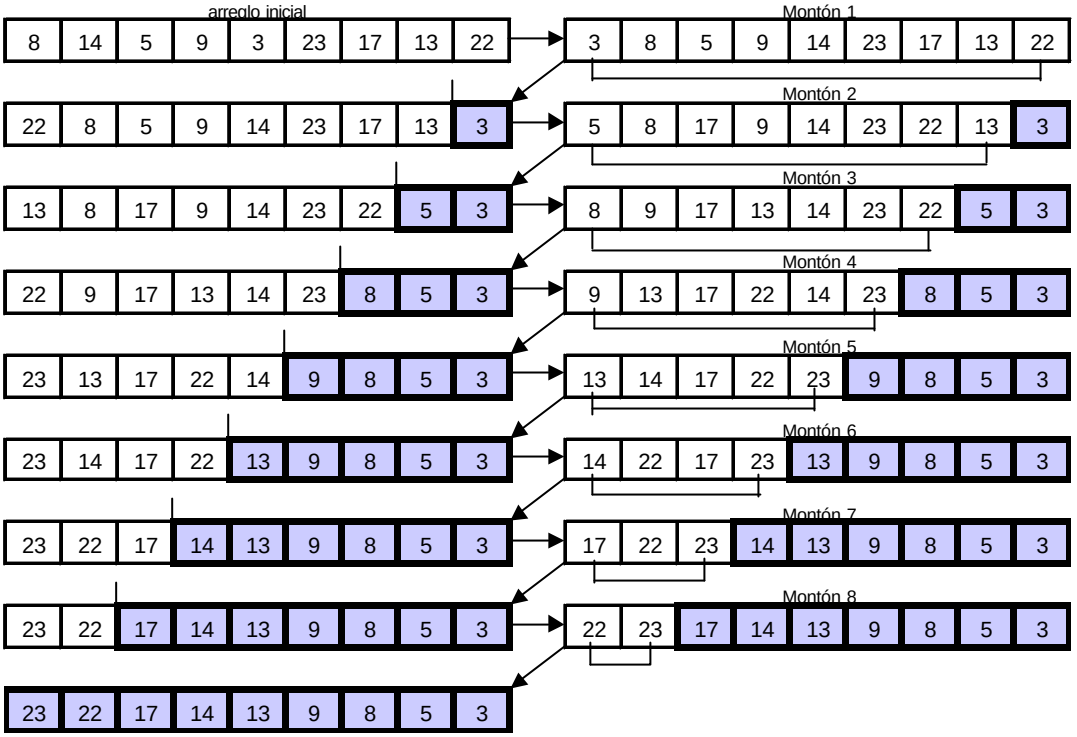
Si las sucesivas cimas se almacenaran en otro arreglo distinto se estaría utilizando una memoria de tamaño *2n*. Una solución consiste en intercambiar la cima, primer elemento del arreglo, con el último, y construir un montón con los *n-1* primeros. Seguidamente se intercambia la cima con el último del nuevo montón, posición *n-1*, y se reconstruye un montón con los *n-2* primeros, y así sucesivamente.

```
PROCEDURE monton(VAR a: Tipoarray);
      VAR
         L,R: Tipo_indice;  x: Tipo_datos;

      PROCEDURE amontonar(L,R:Tipo_indice);
      (*R:posición del extremo derecho del montón*)
      (*L:posición del elemento que se esta incorporando al montón*)
        VAR i,j:Tipo_indice; x:Tipo_datos;
      BEGIN
         i:=L;
         j:=2*L;
         x:=a[L];   (*elemento que se incorpora al montón*)
         IF (j<R) & (a[j+1]<a[j]) THEN
            j:=j+1
         END;
         (*j: índice del elemento que intercambiar  de entre el 2i y el 2i+1*)
         WHILE (j<=R) & (a[j]<x) DO
           a[i]:= a[j];
           a[j]:=x;             (*intercambio*)
           i:=j;
           j:=2*j
           IF(j<R) & (a[j+1]<a[j]) THEN
              j:=j+1
           END                  (*verificar montón hasta el final*)
         END;
      END amontonar;

      BEGIN  (*monton*)
       L:=(n DIV 2)+1;
       R:=n;
       WHILE L>1 DO
          L:=L-1;    (*posición del nuevo elemento que se incorpora al montón*)
          amontonar(L,R);
       END;
       WHILE R>1 DO     (*clasificación 'in situ' por montón*)
          x:= a[1];
          a[1]:=a[R];
          a[R]:=x;
          R:=R-1;
          amontonar(L,R);
       END;
  END monton;
```

Ejemplo:



Como hemos podido observar, la clasificación por montón ordena el arreglo en orden inverso. Si se desease que estuviese clasificado en orden ascendente no hay más que definir un montón de forma alternativa tal que: $h_i \geq h_{2i}$ y $h_i \geq h_{2i+1}$ y realizar las modificaciones asociadas a este hecho en las implementaciones.

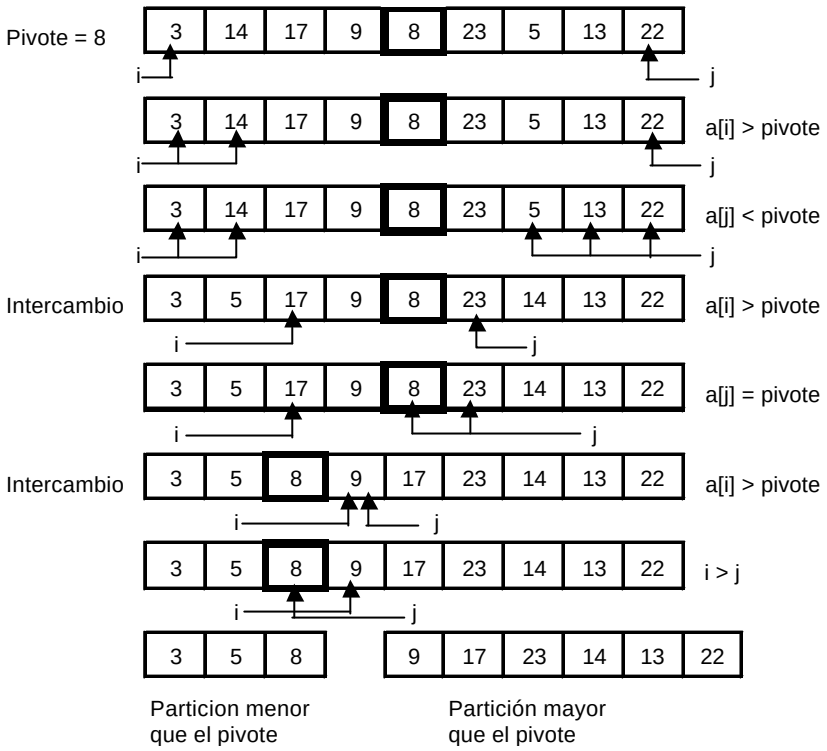
Análisis: El peor caso se produce cuando se realiza el máximo de intercambios en la construcción de los sucesivos montones. En cada pase i la reconstrucción del montón requerirá $2\lceil \log i \rceil$ comparaciones como máximo. Sumando todos los pases el número de comparaciones en el peor caso en $2n\log n - O(n)$. En el caso promedio resulta del orden $2n\log n - O(n\log\log n)$.

- ***<u>Clasificación por partición (Quicksort)</u>***

Se fundamenta en el método de intercambio directo (que era el peor de los directos). Se basa en el hecho de que cuanto mayor sea la distancia entre los elementos que se intercambian más eficaz será la clasificación.

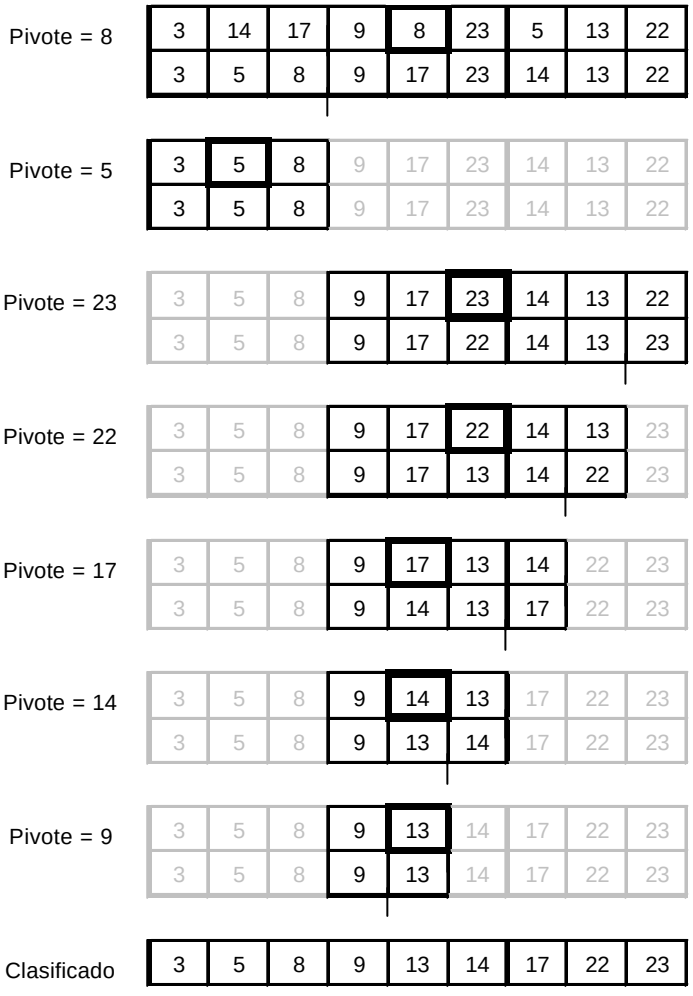
Se elige un valor de llave al azar, x, denominado *pivote.* Se recorre el arreglo desde la izquierda hasta encontrar una llave mayor que el pivote, y desde la derecha hasta encontrar una menor. Se *intercambian* y se repite el proceso hasta que los índices de incremento y decremento se crucen. De esta forma tendremos a la izquierda todos los elementos menores que el pivote y a la derecha los mayores. A cada una de estas partes se denominan *particiones*.

Ejemplo: (eligiendo como pivote el elemento de la posición central)



Se repite el proceso para cada una de las particiones y así sucesivamente.

Ejemplo de clasificación por partición (quicksort)



La implementación es recursiva.

```
PROCEDURE particion(a: Tipoarray);

    PROCEDURE ordena (L, R : Tipo_indice);
    VAR
        i,j: Tipo_indice;
        piv,aux: Tipo_datos;
    BEGIN
       i:=L; j:=R;
       piv:=a[(L+R)DIV 2];       (* pivote *)
       REPEAT
          WHILE a[i] < piv DO i := i+1 END;
          WHILE piv < a[j] DO j := j-1 END;
          IF i <= j THEN
             aux:=a[i]; a[i]:=a[j]; a[j]:=aux;   (*intercambio*)
             i:=i+1;
             j:=j-1
          END
       UNTIL i>j;
       IF L<j THEN ordena(L,j) END;
       IF i<R THEN ordena(i,R) END;
    END ordena;

BEGIN                    (*particion*)
      ordena (1,n);
END particion;
```

El mejor pivote sería la *mediana* de los elementos del arreglo. Sin embargo, su cálculo previo a la clasificación tiene un coste excesivo para utilizarlo. Por eso se utilizan otros métodos, por ejemplo, se puede utilizar como pivote la media del primero, el último y el central.

Análisis:

Dado que el método es de naturaleza recursiva el coste vendrá dado por dos llamadas más el tiempo que transcurre en la partición: $T(n) = T(i) + T(n-i-1) + cn \quad T(1) = T(0) = 1;$ donde i es el número de elementos en la partición de los menores.

En el peor caso el pivote es un extremo. Entonces $i=0$ y $T(n) = T(n-1) + cn$, $n> 1$ despreciando T(0)=1 por no ser significativo. Resolviendo la recurrencia tenemos $T(n) = O(n^2)$.

En el mejor de los casos el pivote está en el centro. En este caso $T(n) = 2T(n/2) + cn$ y resolviendo tenemos $T(n) = O(n \log n)$.

En el caso promedio se supone que todos los tamaños de la partición con elementos menores al pivote son igualmente probables. Por tanto su probabilidad es $1/n$. En este caso el coste de la clasificación por partición es también de orden $n \log n$.

Cálculo del k-ésimo mayor elemento

La clasificación por partición permite, además de clasificar un arreglo, calcular de forma eficaz el *k-ésimo* elemento mayor del mismo. La mediana consiste en calcular el *k-ésimo* mayor, con $k=n/2$.

```
PROCEDURE Encuentra_kesimo(a: Tipoarray;k: INTEGER;VAR x:Tipo_datos);
VAR L,R,i,j: INTEGER;w: Tipo_datos;
BEGIN
   L:=1;R:=n;
   WHILE L<R DO
      x:=a[k];i:=L;j:=R;
      REPEAT
         WHILE a[i]<x DO i:=i+1 END;
         WHILE x<a[j] DO j:=j-1 END;
         IF i<=j THEN
            w:=a[i];a[i]:=a[j];a[j]:=w;i:=i+1;j:=j-1
         END;
      UNTIL i>j;
      IF j<k THEN L:=i END;
      IF k<i THEN R:=j END;
   END;
   x:=a[k];
END Encuentra_kesimo;
```

Este método es excelente para calcular el *k-ésimo* elemento mayor que es de orden lineal, si bien en el peor caso es de orden n^2.

Comparación de los métodos avanzados de clasificación

Comparando los *métodos directos* de clasificación sobre arreglos, la selección directa será la opción a elegir en general, aunque la inserción es algo más rápida cuando las llaves predominantemente se clasifican al principio y la vibración puede ser mejor cuando el arreglo está inicialmente casi ordenado.

Respecto a los *métodos avanzados*, la clasificación rápida tiene un magnífico comportamiento en los casos mejor y promedio, sin embargo en el caso peor es deficiente (cuando las llaves son muy parecidas en valor).

En este caso es aconsejable otro método, por ejemplo, la clasificación por montón.

Los métodos avanzados presentan unos costes computacionales claramente inferiores a los directos pero no son los más eficaces para todo n ya que requieren más operaciones auxiliares que los directos. Por eso, para n pequeño son preferibles los métodos directos ya que en estos casos las diferencias entre los órdenes n^2 y $n\log n$ no son muy significativas y las operaciones auxiliares de los avanzados hacen que estos métodos sean menos eficaces.

En el caso de clasificación por partición, en principio se realizará el proceso de partición, pero sólo hasta que el tamaño de las particiones es lo suficientemente pequeño para aplicar un método directo eficaz, como la selección directa.

A modo de resumen, la comparativa entre los métodos directos de clasificación es:

		Mínimo	Promedio	Máximo
Inserción	Comparaciones	$n - 1$	$\frac{n^2 + n - 2}{4}$	$\frac{n^2 + n - 2}{2}$
	Movimientos	$2(n - 1)$	$\frac{n^2 + 9n - 10}{4}$	$\frac{n^2 + 3n - 4}{2}$
Selección	Comparaciones	$\frac{n^2 - n}{2}$	$\frac{n^2 - n}{2}$	$\frac{n^2 - n}{2}$
	Movimientos	$3(n - 1)$	$n(\ln n + g)$	$3(n - 1) + n^2/4$
Intercambio	Comparaciones	$\frac{n^2 - n}{2}$	$\frac{n^2 - n}{2}$	$\frac{n^2 - n}{2}$
	Movimientos	0	$\frac{3(n^2 - n)}{4}$	$\frac{3(n^2 - n)}{2}$

3.- CLASIFICACIÓN EN MEMORIA SECUNDARIA

El TDA a considerar será la secuencia, implementada mediante una cinta, por lo que debemos tener en cuenta que el acceso al *k-ésimo* elemento no será posible sin haber leído los *k-1* anteriores.

Se entiende por *mezcla* la tarea de combinar varias secuencias en una sola mediante selección de los componentes accesibles en cada momento.

Se entiende por *fase* cada operación que trata un conjunto completo de datos.

Se entiende por *pase (etapa)* al proceso más corto que por repetición constituye un proceso de clasificación.

3.1.- Clasificación basada en Mezclas

3.1.1.- Mezcla directa:

- Tomar como *fuente* la secuencia original c_1
- Dividir la fuente en dos mitades, en las cintas *destino* c_2 y c_3
- Mezclar c_2 y c_3 combinando cada elemento accesible en pares ordenados, en c_1
- Repetir el proceso: se obtiene una cinta con cuádruplos ordenados.
- Repetir el proceso hasta que toda la cinta esté ordenada

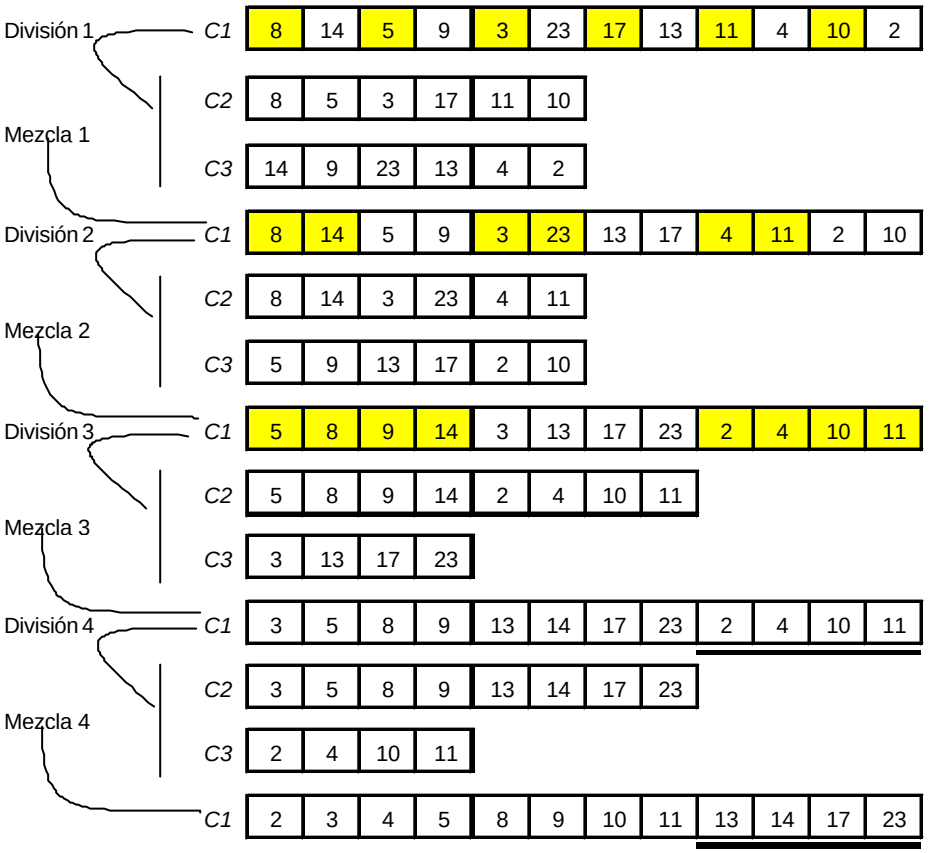
Cada *pase o etapa* consta de *dos fases*, una de *división* y otra de *mezcla*, por ello se denomina *mezcla de dos fases o mezcla de tres cintas.*

Ejemplo:

> Para crear las dos cintas *destino* (auxiliares), se recorre la cinta *fuente* y sus elementos se van enviando alternativamente a las cintas c_2 y c_3. De esta manera, la cinta c_2 contendrá los elementos de posiciones impares en la cinta fuente, y c_3 los de las posiciones pares. Si la cinta fuente tiene un número impar de elementos, c_2 tendrá un elemento más.

Tras la primera división sólo se tiene acceso al primer elemento de cada una de las cintas c_2 y c_3. Se compara el primer elemento de c_2 y el primero de c_3, se ordenan y se escriben en la cinta temporal *c*. Seguidamente ya se tiene acceso a los segundos elementos de las cintas c_2 y c_3. Se repite el proceso y el resultado será que *c* tiene pares de elementos ordenados.

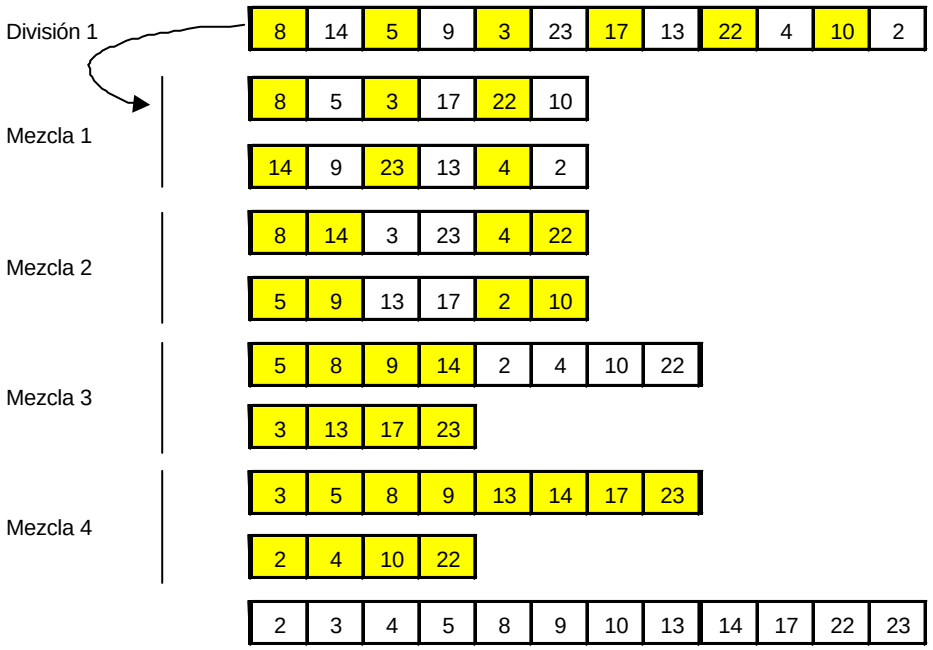
Terminada la primera fase de *mezcla*, *c* pasa a ser c_1 y se divide por segunda vez. En la división 2 se crean dos nuevas cintas, enviando a cada una de ellas "*pares*" de elementos ya ordenados. En la mezcla 2 se combinan pares de elementos originando *cuádruplos* ordenados. Este proceso continua hasta agotar alguna de las dos cintas auxiliares. Cuando esto sucede, se copian de forma directa el resto de elementos de la otra secuencia, que ya están ordenados, lo que se conoce como *copiado de cabos.*



Debemos observar que la fase de división no aporta nada a la clasificación y sin embargo tiene un coste computacional significativo ya que realizan aproximadamente la mitad de las operaciones de copia. Sería interesante reducir esta fase.

La fase de división puede eliminarse completamente combinando la fase de división con la de mezcla. En vez de mezclar en una sola cinta c, que posteriormente será dividida, puede irse separando en dos, de manera que en el siguiente pase ya estarán divididas. Este método se conoce como *mezcla de una fase o mezcla directa balanceada.*

Ejemplo:



Evidentemente el coste en tiempo será menor ya que sólo realiza la mitad de las operaciones de copia de elementos. Será necesario utilizar una *cuarta cinta* ya que el trabajo se realiza leyendo de dos cintas y escribiendo en otras dos, pero esto no presenta ningún inconveniente (salvo que no se disponga de esta cuarta cinta).

Análisis:

Las operaciones más frecuentes serán los movimientos que se realizan en el copiado de elementos entre cintas y las comparaciones.

En los métodos de clasificación sobre memoria secundaria el análisis detallado de las comparaciones no es significativo debido a que éstas se realizan en memoria principal.

En la mezcla directa se producen $\lceil \log n \rceil$ pases, por tanto el número de movimientos es $n \log n$ y el de comparaciones es inferior ya que en las operaciones de copiado de cabos no se produce ninguna.

3.1.2.- Mezcla natural

Aprovecha el hecho de que entre los elementos de la secuencia original, algunos elementos consecutivos ya se encontrarán ordenados entre sí.

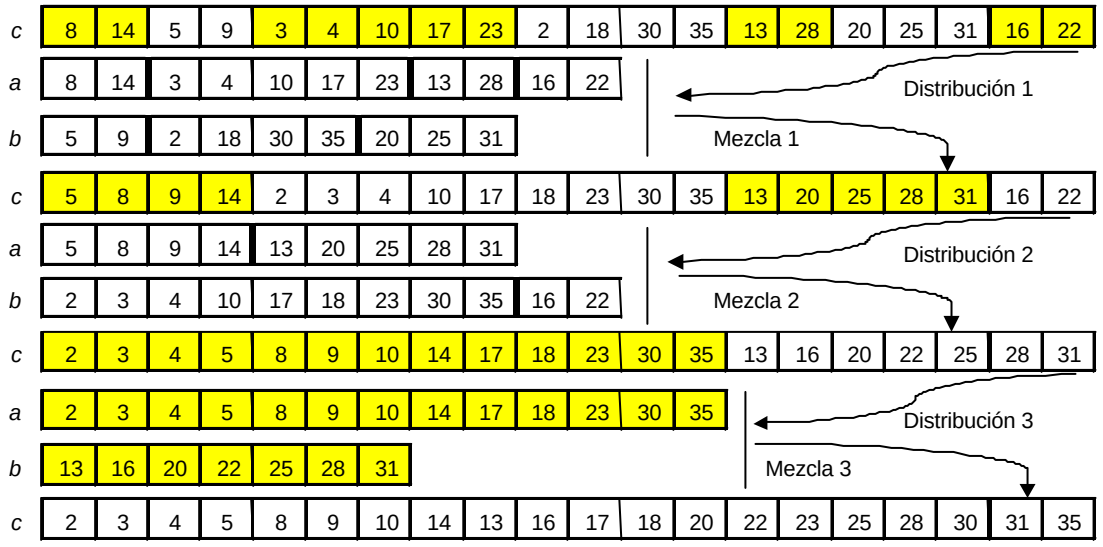
Se denomina *subsecuencia ordenada* al máximo conjunto de elementos consecutivos $a_i...a_j$ de la secuencia a ordenar que satisfacen: $(a_{i-1} > a_i)$ AND $(a_k £ a_{k+1},\ "k\ i£k£j)$ AND $(a_j > a_{j+1})$

La mezcla natural se basa en la combinación de subsecuencias ordenadas. Las subsecuencias ordenadas de la cinta fuente, se distribuyen en dos cintas destino auxiliares *a* y *b*. Seguidamente se mezcla una subsecuencia ordenada de cada cinta auxiliar.

- Fuente *c*
- Distribuir las subsecuencias ordenadas de la fuente en las cintas destino *a* y *b*
- Mezclar *a* y *b* en *c*, combinando subsecuencias ordenadas de cada cinta auxiliar.

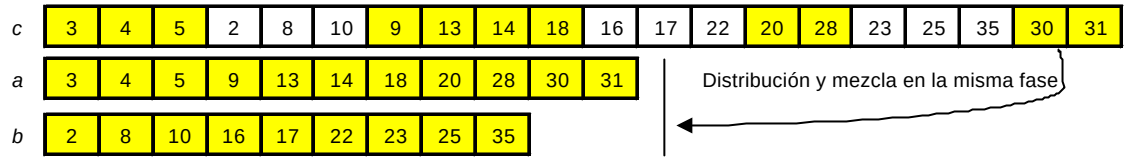
Cada *pase* en la mezcla natural consta de *dos fases*, una de *distribución* y otra de *mezcla*.

Ejemplo:



Es posible que se produzca "*mezcla*" en la fase de distribución cuando el primer elemento de la subsecuencia ordenada *(k+2)-ésima* es mayor que el último elemento de la *k-ésima*.

Ejemplo:



Análisis:

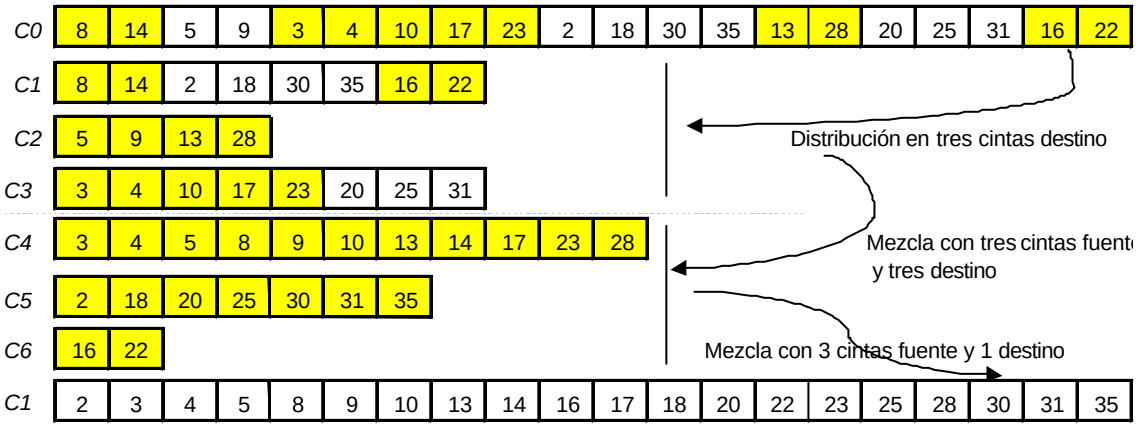
En el peor caso el número de movimientos es de orden $n \log n$, e inferior en el caso promedio. El número de comparaciones es mucho mayor, pero al ser el coste de una comparación muy inferior al de un movimiento, este incremento no resulta significativo.

3.1.3.- Mezcla balanceada múltiple

El algoritmo de mezcla natural copia tanto en la fase de distribución como en la de mezcla. Las copias pueden disminuirse si se consigue reducir el número de pases. Para ello la fase de distribución puede realizarse entre más de dos cintas, de manera que la mezcla combine simultáneamente varias (más de dos) subsecuencias ordenadas, con lo que el número de pases será menor.

Además podemos eliminar la fase de distribución mediante la copia de las subsecuencias ordenadas, durante el proceso de mezcla, en cintas destino auxiliares que harán de fuente en el siguiente pase, y así las cintas fuente y destino se alternan consecutivamente.

Ejemplo: mezcla balanceada con seis cintas



Análisis:

Supóngase que se utilizan N cintas destino en la fase de distribución. Mezclar m subsecuencias ordenadas que están distribuidas uniformemente en N cintas origina, en una primera fase de mezcla, una subsecuencia ordenada de m/N subsecuencias ordenadas, y en la *k-ésima* m/N^k.
El número total de pases necesarios para clasificar n subsecuencias ordenadas con N cintas será, en el peor de los casos $k = \lceil \log_N n \rceil$. Como cada pase necesita n copias, el número total de copias vendrá dado por $M = n\lceil \log_N n \rceil$

3.1.4.- Clasificación polifásica.

Mejora el rendimiento de la mezcla balanceada. Las cintas fuente y destino no son establecidas a priori, sino como consecuencia de la mezcla realizada.

Ejemplo: clasificación polifásica con tres cintas.

	C_1	C_2	C_3
Distribución inicial	13	8	0
Mezcla 1	5	0	8
Mezcla 2	0	5	3
Mezcla 3	3	2	0
Mezcla 4	1	0	2
Mezcla 5	0	1	1
Mezcla 6	1	0	0

En el proceso de mezcla dos hacen de fuente y la tercera de destino, al finalizar las combinaciones hará de cinta destino aquella que se haya agotado, la cual sólo podrá ser identificada tras la mezcla.

La clasificación polifásica aprovecha al máximo las cintas ya que con *N* cintas realiza mezclas de *N-1* subsecuencias ordenadas.

Lo que se pretende es que al final haya una sola subsecuencia ordenada en una cinta y las demás estén agotadas. Esto no siempre es posible.

	C_1	C_2	C_3
Distribución inicial	14	8	0
Mezcla 1	6	0	8
Mezcla 2	0	6	2
Mezcla 3	2	4	0
Mezcla 4	0	2	2
Mezcla 5	2	0	0

Para que esto no ocurra, la suma en cada nivel debe ser un número de *Fibonacci* (aunque esta condición no es suficiente). Veamos cómo podemos obtener el número de subsecuencias ordenadas en cada cinta para que la clasificación polifásica con tres cintas sea satisfactoria:

Nivel	$C_1(n)$	$C_2(n)$	C_1+C_2
0	1	0	1
1	1	1	2
2	2	1	3
3	3	2	5
4	5	3	8
5	8	5	13
6	13	8	21
7	21	13	34

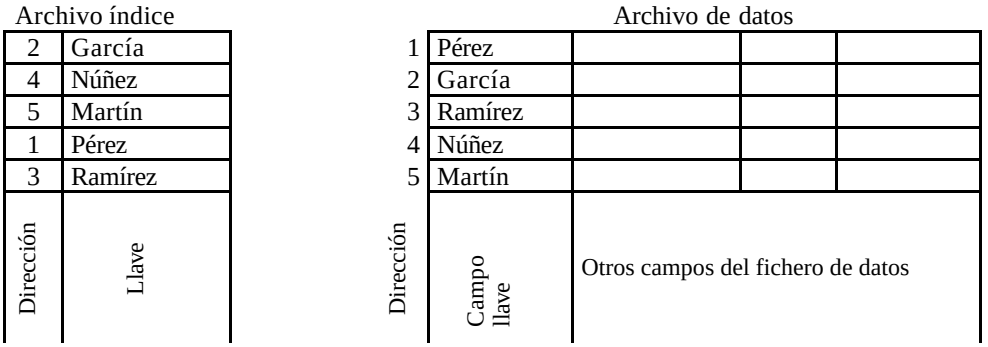
Con $C_1(n)$ y $C_2(n)$ se indican las cintas que hacen la función de c_1 y c_2 en cada nivel *n*.

La clasificación polifásica de tres cintas resultará satisfactoria si la distribución inicial de subsecuencias entre las cintas fuente es tal que son números consecutivos de Fibonacci. Esta condición es suficiente pero no necesaria; es decir, existen otras distribuciones iniciales para las que se clasifica satisfactoriamente.

Para que una clasificación polifásica con *N* cintas sea satisfactoria (o perfecta) el número de subsecuencias ordenadas iniciales tiene que ser suma de cualquier *N-1, N-2,...1* sumas de números de Fibonacci de orden *N-2*.

3.2.- Archivos indexados

El método de clasificación mediante archivos indexados se basa en considerar, asociada a cada llave, la dirección física del registro que caracteriza. Cuando el archivo índice contiene la dirección del registro de la llave se conoce como indexado de *índice denso*.



Las operaciones de clasificación y búsqueda suelen realizarse en memoria principal sobre el archivo de índices y no sobre el archivo de datos.

La idea básica de los archivos indexados suele refinarse agrupando las llaves en bloques. En este caso, al archivo de índices se le denomina *de índice disperso* y está formado por grupos de pares (x, b) donde b es la dirección física del bloque en el cual el primer registro tiene la llave de valor x. La búsqueda se realiza por bloques lo que evita comparar con todos y cada uno de los elementos.

En la distribución de registros en bloques suele preferirse dejar espacios para registros adicionales que puedan insertarse posteriormente.

3.3.- Tablas de dispersión (Hashing)

La idea básica consiste en transformar las llaves en direcciones de memoria mediante una *función de transformación*.

Las tablas de dispersión se aplican cuando el conjunto de llaves posibles es mucho mayor que el de llaves reales a almacenar.

***Ejemplo**: Se quiere desarrollar una aplicación para registrar los participantes en un congreso. Se esperan alrededor de 3000 participantes y se requiere almacenar los datos de cada uno de ellos, como nombre, apellidos, DNI, etc. Se utilizará como campo llave el primer apellido. Suponiendo que los apellidos pueden tener hasta 16 caracteres habrá 26^{16} llaves posibles. Sería absurdo pretender reservar memoria para todas ellas. Bastará reservar memoria suficiente para 3000 que son los asistentes previstos. El problema radica en cómo asignar una llave de todas las posibles a una dirección de memoria.*

Para una función de transformación existen varias llaves distintas que se aplican a la misma dirección de memoria. A los registros con estas llaves se les denomina sinónimos. Cuando aparecen dos sinónimos se dice que se ha producido una colisión.

Definiciones:

Dado un conjunto de llaves posibles *X*, y un conjunto de direcciones de memoria *D*, una función de transformación *H(x),* es una aplicación suprayectiva del conjunto de llaves posibles en el conjunto de direcciones de memoria: $H: X \rightarrow D$

El TDA *tabla de dispersión* es un tipo de datos homogéneo, denominadas celdas, de tamaño fijo, *Ttabla*, compuesto por un número fijo de componentes a las que se accede mediante una dirección de memoria resultante de una función de transformación. Sobre este TDA se definen los operadores *Insertar, Buscar y Eliminar*.

Dos llaves distintas x_1 y x_2 son sinónimas si $H(x_1) = H(x_2)$

Se dice que se ha producido *desbordamiento* cuando una nueva llave se aplica a una dirección de memoria completamente ocupada, y se dice que se ha producido una *colisión* cuando dos llaves distintas se aplican sobre la misma celda.

Se denomina *densidad de llaves* al cociente entre el número de llaves en uso, *m*, y el número total de llaves posibles, *nx*

Se denomina *factor de carga (o densidad de carga),* α, al cociente entre el número de llaves en uso y el número total de registros almacenables en la tabla de dispersión: $\alpha = m / (s*b)$; donde *s* es el número de registros por bloque y *b* el número de bloques que hay en la tabla de dispersión.

Ejemplo: Tabla de dispersión que consta de mil bloques con dos registros por bloque. El conjunto de llaves posibles es el conjunto de números del DNI (00000000 al 99999999) y la función de dispersión se define tal que los tres primeros dígitos del DNI asignan la dirección de memoria (000 a 999).

000	00077641	00034567
001	00131117	
549	54913110	54939189
999	99911011	99931419

Las llaves 54913110 y 54939189 son sinónimos y por tanto ocupan el mismo bloque. Supongamos que se requiere almacenar la llave 54900000. Al intentar la inserción en su bloque se encuentra que la primera celda está ocupada produciéndose una colisión. Como cada bloque tiene dos celdas se buscaría en la siguiente y al estar ocupada, además de una colisión se produce desbordamiento.

Suponiendo que el número de llaves en uso es de 500, entonces $\alpha = (500 / 2000) = 0.25$

3.3.1.- Funciones Hash

La dirección de memoria debe ser calculada con sencillez pero además es importante que se produzcan el menor número posible de colisiones.

Lo normal será dividir el conjunto de llaves posibles en grupos de llaves aproximadamente iguales y asignar a cada grupo una dirección de bloque. La función de transformación más conocida es el resto de la división entera: $f(x) = X \bmod M$. Así, el espacio de direcciones de los bloques será [*0, M-1*] y la tabla tendrá como mínimo *M* bloques. La elección de *M* es crucial. Evidentemente, si las posibles llaves son números acabados en cero, la elección de $M = 10$ es desastrosa ya que el resto es siempre 0.

En todo lenguaje existen sufijos que aparecen frecuentemente mientras que otros no lo hacen nunca. Supóngase que los caracteres se almacena en un byte y que las cadenas de caracteres son de 8 bits. Si por ejemplo $M = 2^k$, $k<=24$, todas las cadenas que tengan los tres últimos caracteres iguales se asignarán a la misma dirección, ya que dividir entre dos es desplazarse dos bits hacia la izquierda, produciéndose un gran número de colisiones. Por ello suele recomendarse de M sea un número primo, de manera que sus únicos divisores sean *1* y *M*. Knuth demostró que si M divide a r^{ka} donde *a* y *k* son números pequeños y *r* es el número de bits utilizados para la representación del carácter, entonces $f(x) = X \bmod M$ tiende a ser la superposición de los caracteres de *X*, teniendo un efecto semejante a promediar sobre todos sus caracteres.

Otra función hasingh clásica es la conocida como *plegado*: la dirección se obtiene dividiendo la llave en partes iguales y sumando todas ellas. La suma de las partes puede realizarse directamente (*plegado por desplazamiento*) o plegar el identificador por las fronteras de las partes y sumar los dígitos coincidentes (*plegado por las fronteras*).

Ejemplo: a) cadena de ocho caracteres representados por los números de orden dentro de la secuencia de cotejo correspondiente.
b) Plegado por desplazamiento
c) Plegado en las fronteras en base decimal
d) Plegado en las fronteras en base binaria
e) Detalle del plegado en base binaria

a)

68	75	82	71	63	93	102	75

	b)	***c)***	***d)***
	75	75	75
	102	201	102
	93	93	93
	63	36	252
	71	71	71
	82	28	74
	75	75	75
	68	86	34
Dirección =	629	665	776

e)

```
102 => 01100110                01100110 => 102
 63 => 00111111   ------->     11111100 => 252
 82 => 01010010                01001010 =>  74
 68 => 01000100                00100010 =>  34
```

3.3.2.- Manejo de desbordamiento o sobrecarga

Cuando se ha de insertar una nueva llave, si la celda que le corresponde está ocupada se produce una *colisión* y si todo el bloque está lleno, se ha producido un *desbordamiento o sobrecarga*.

El método de manejo de sobrecargas más evidente es la *exploración lineal*. Consiste en buscar en el bloque siguiente. Se distinguen dos acciones: *inserción* y *búsqueda*. Para la inserción se busca en el bloque siguiente una celda libre, si vuelve a producirse sobrecarga se busca en el siguiente y así sucesivamente. Para la búsqueda de una llave se busca en la dirección siguiente y así sucesivamente hasta encontrar la llave, o encontrar que la celda del bloque está vacía o la tabla está llena.

En general la exploración puede expresarse como: *dirección = f(x) + g(x)*; conde *f(x)* es la función de dispersión y *g(x)* es la función de tratamiento de sobrecargas. Considerando la tabla circular, la dirección vendrá dada por: *dirección = (f(x) + g(x))* mod *TamañoTabla*

En el caso de exploración lineal: $g(x) = i$, $i = 1 \ldots TamañoTabla$

La exploración lineal tiende a colocar las llaves poco uniformemente de manera que las sobrecargas producidas irán llenando los bloques cercanos a los ocupados.

Una técnica que mejora este comportamiento es la *exploración cuadrática*, definida por:

$$g(x) = i^2 , \ i = 1 \ldots TamañoTabla$$

Con esta función de exploración la *búsqueda* se realiza examinando los bloques:

$$f(x) , \qquad (f(x) + i^2) \text{ mod } TamañoTabla , \qquad (f(x) - i^2) \text{ mod } TamañoTabla,$$

con $1 \leq i \leq (TamañoTabla-1) / 2$

Cuando *TamañoTabla* es un número primo de la forma *4j+3*, *j* entero, la exploración cuadrática puede recorrer todos los bloques de la tabla.

La técnica de *rehasingh* generaliza los conceptos anteriores, de forma que la función de exploración será una familia de funciones de dispersión que se examinan sucesivamente en un orden dado. Así $g(x) = f_i(x) , \ i = 1 .. m$ donde cada $f_i(x)$ es una función de dispersión.

Existen otras estrategias de tratamiento de colisiones. La más elemental consiste en mantener una *lista asociada* a cada bloque de la tabla hash, de manera que en ella se almacenan todos los sinónimos que no se pueden insertar en el bloque correspondiente. La implementación más adecuada de esta lista es mediante una lista enlazada. Esta estrategia se denomina *encadenamiento* y requiere disponer en cada bloque del espacio necesario para almacenar un enlace (por ejemplo, un puntero).

Análisis:

En ausencia de sobrecargas el coste es independiente del número de llaves en uso, *n*.

Sin embargo, la inserción y la búsqueda por transformación de llaves tiene en el caso peor un rendimiento pésimo ya que siempre hay colisiones y sobrecargas que obligarán a realizar todas las comparaciones.

Para el caso promedio, el número promedio de exploraciones necesarias para recuperar una llave aleatoria de la tabla *E* será aproximadamente igual a $-(\ln(1-\alpha)) / \alpha$, (siempre que la función de dispersión sea uniforme: exploración cuadrática o rehashing).

Para la estrategia de encadenamiento tendremos $E = 1 + \alpha^2$

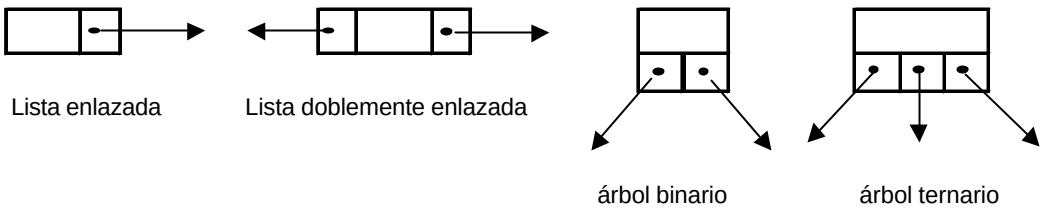
***Nota**: En los capítulos 5 y 6 se han omitido todos los procedimientos en Modula 2 lo cual, evidentemente, no quiere decir que no sean importantes o que no puedan caer en el examen. Es recomendable echarles un vistazo, incluso en algún caso, como en la construcción de un árbol perfectamente balanceado vale la pena estudiarlo ya que suele caer en exámenes. No obstante he considerado que lo importante es entender los conceptos y procedimientos, y en ese sentido he centrado los apuntes.*

5.- TDA DINÁMICOS NO LINEALES: ÁRBOLES

5.1.- Introducción y Definiciones

- Se denomina *nodo* a cualquier tipo cuyos elementos son registros formados por un campo *Datos* y un número dado de *apuntadores o enlaces.*

Ejemplo de diferentes tipos de nodos



- El TDA *árbol* de grado *n* está formado por nodos con *n* apuntadores. Las listas enlazadas pueden considerarse un caso particular del TDA árbol (árboles degenerados).

- Se denomina *grado* del nodo al número de descendientes directos del nodo. El grado del árbol es el mayor de los grados de los nodos.

- Definición de árbol binario:

```
TYPE Ptr_Nodo = POINTER TO Nodo
TYPE Nodo = RECORD
        Dato  :  Tipo_datos;
        izq, dch  :  Ptr_Nodo;
        END
```

- Se denomina *nivel* de un nodo al número de descendientes que deben recorrerse desde la raíz hasta el nodo. El nivel de la raíz es cero. El nivel máximo se denomina *altura* o *profundidad*.

- La *longitud de trayectoria interna de un nodo* es el número de aristas o ramas que se recorren desde la raíz hasta el nodo. La longitud de trayectoria de un árbol o *longitud de trayectoria interna, L_I,* es la suma de las longitudes de trayectoria de sus nodos. La *longitud de trayectoria media* es:

$$L_I^m = \frac{\sum_{i=1}^{n} n_i \cdot i}{n}$$ donde n_i es el número de nodos en el nivel *i.*

- Dado un árbol, su *árbol extendido* es el árbol ampliado con nodos especiales tal que todos los subárboles son completos, de manera que todos los apuntadores sin nodos descendientes apuntan a un nodo especial. Los nodos especiales no tienen descendientes.

- Se define la *longitud de trayectoria externa* de un árbol L_E, como la suma de las longitudes de trayectoria de todos sus nodos especiales. La *longitud de trayectoria externa media* es:

$$L_E^m = \frac{\sum_{i=1}^{m} m_i \cdot i}{m}$$, donde m_i es el número de nodos especiales en el nivel *i.*

En los árboles binarios $L_E = 2n + L_I$

- Sean *g* el grado, *h* la altura y *n* el número de nodos del árbol, entonces:

  El número de nodos especiales es $m = (g - 1)\, n + 1$

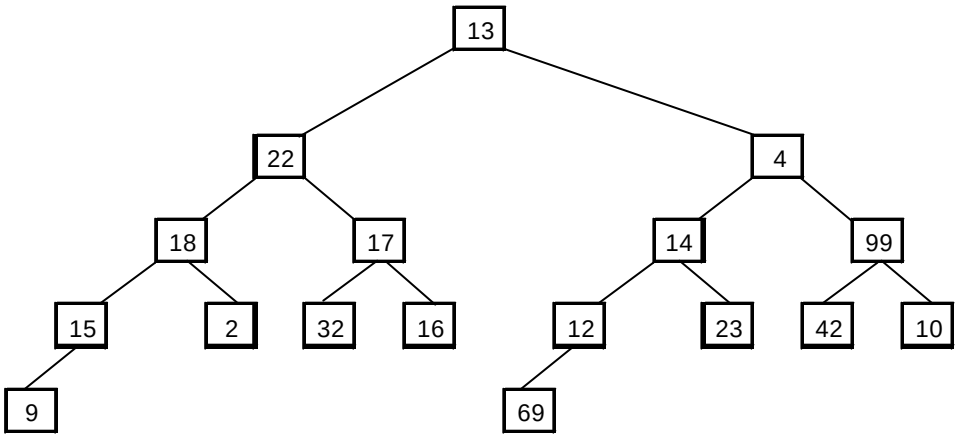
  El número máximo de nodos es $N_g = \sum_{i=0}^{h-1} g^i$

5.2.- Árboles perfectamente balanceados

- Un árbol *perfectamente balanceado* es aquel en el que para cada nodo el número de nodos en sus subárboles derecho e izquierdo difieren como máximo en uno.

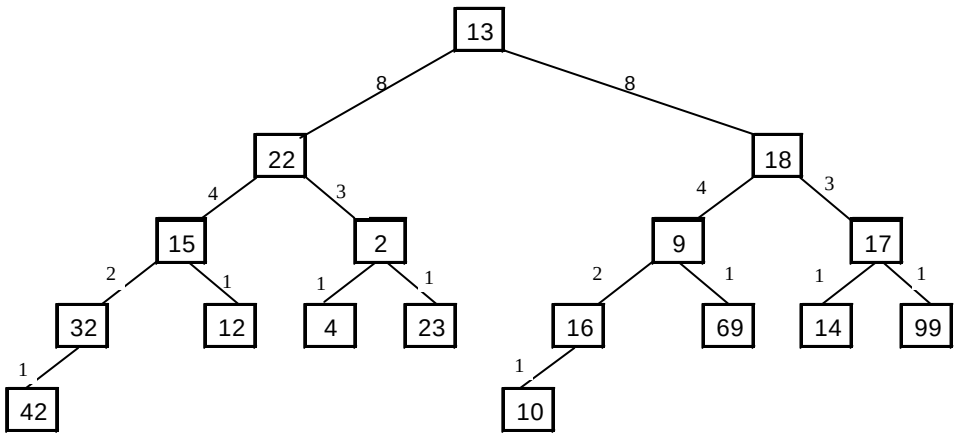
Si se conoce el número *n* de nodos del árbol binario perfectamente balanceado, para cada nodo se construye un subárbol izquierdo perfectamente balanceado de n_izq = n DIV 2 y otro derecho de n_dch = n – n_izq – 1 nodos.

Ejemplo: *Nodos:* *13, 22, 18, 15, 9, 2, 17, 32, 16, 4, 14, 12, 69, 23, 99, 42, 10*



Lo importante es entender que se trata de un procedimiento recursivo en el que se van creando los nodos e introduciendo en ellos el valor correspondiente, seguidamente se llama al procedimiento recursivamente para crear el subárbol izquierdo, no produciéndose los enlaces hasta que finaliza la recursión.

Lo normal será que no conozcamos el número de nodos sino que se vaya construyendo el árbol a medida que se van creando los nuevos nodos. En la construcción tendremos que tener en cuenta a la hora de añadir el nuevo nodo la condición de que sea perfectamente balanceado. Así, el árbol resultaría:



5.3.- Árboles binarios ordenados según el recorrido

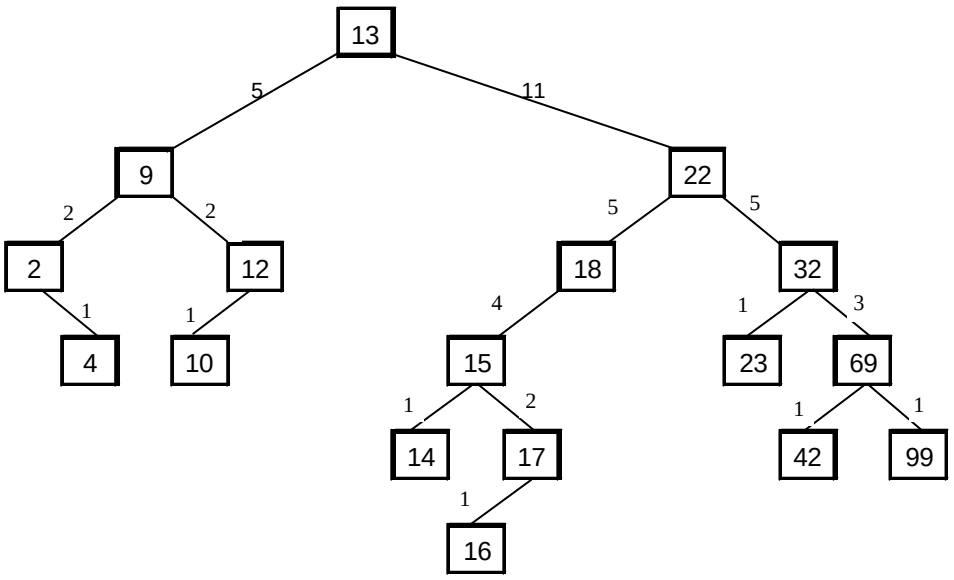
- Un *árbol binario ordenado según el recorrido* es aquél que para cada nodo se visita el nodo, su subárbol izquierdo y su subárbol derecho en un orden establecido:

  Preorden: visitar el nodo antes que los subárboles (NDI, NID)
  En orden: visitar el nodo después de un subárbol y antes que el otro (DNI, IND)
  Postorden: visitar el nodo después de los subárboles (DIN, IDN)

5.4.- Árboles de búsqueda binarios

Un árbol binario de búsqueda, es un árbol binario en el que dadas dos condiciones mutuamente excluyentes (por ejemplo > y <), para cada nodo, todas las llaves de su subárbol izquierdo satisfacen una condición y todas las de su subárbol derecho la otra.

Con los nodos del ejemplo anterior, el árbol binario de búsqueda resultante sería:



La declaración del TDA árbol binario de búsqueda, en Modula2, es

```
TYPE Ptr_Nodo = POINTER TO Nodo;
TYPE Nodo = RECORD
        llave : Tipo_llave;
        izq, dch : Ptr_Nodo;
        END
```

y el árbol de búsqueda vacío se representa por NIL.

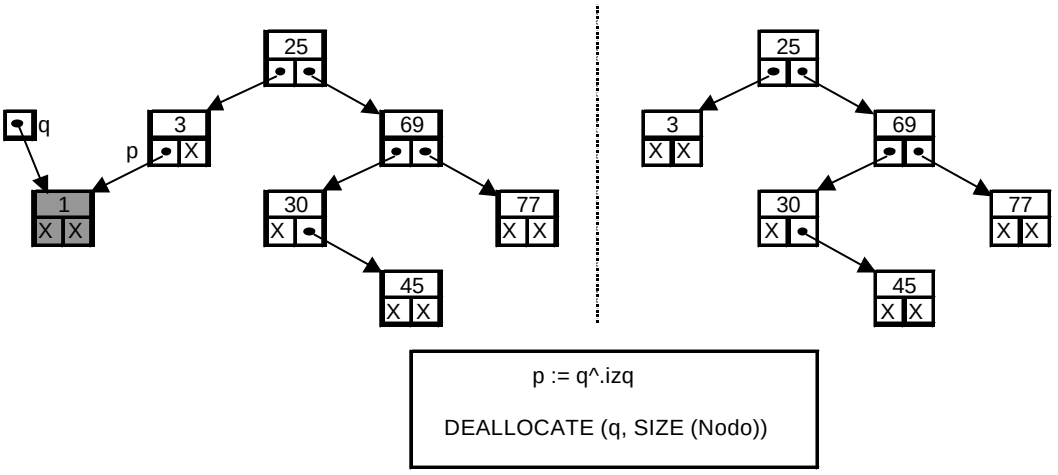
En la operación de ***inserción*** de un elemento en el árbol de búsqueda, para cada nodo se consulta si el dato es menor o mayor que la llave, decidiendo así si se prosigue la búsqueda por la izquierda o por la derecha respectivamente. El procedimiento termina cuando se alcanza el puntero *NIL*, ya que esto querrá decir que el elemento no se ha encontrado y hay que insertarlo.

En la eliminación hemos de tener en cuenta que se nos pueden presentar *tres situaciones*:

1.- No hay ningún nodo llave igual al que se desea eliminar. (no hay nada que hacer)

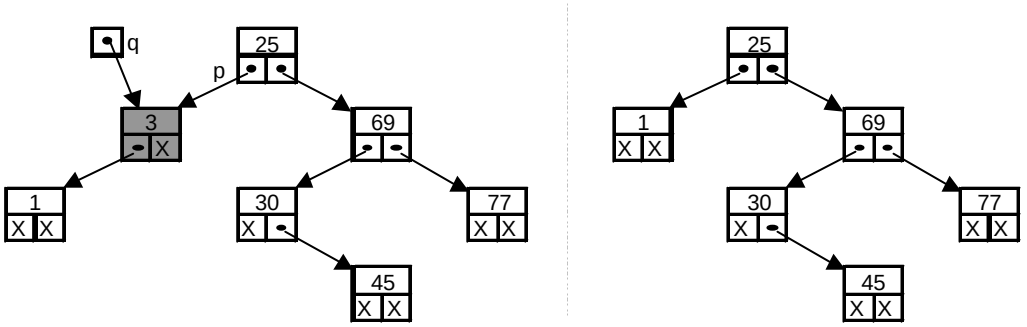
2.- El nodo a eliminar tiene como máximo un subárbol descendiente.

Si no tiene *ningún descendiente* basta con asignar NIL al puntero que apunta al nodo a eliminar y liberar el nodo (utilizaremos un puntero auxiliar)




```
p := q^.izq
DEALLOCATE (q, SIZE (Nodo))
```

En el caso de que tenga *un descendiente* hay que tener en cuenta si es por la izquierda o por la derecha. En cualquiera de estos casos, el puntero que lo apunta hay que modificarlo para que apunte a su descendiente y liberar el nodo a eliminar utilizando un puntero auxiliar (lógicamente sin infringir la condición de árbol de búsqueda



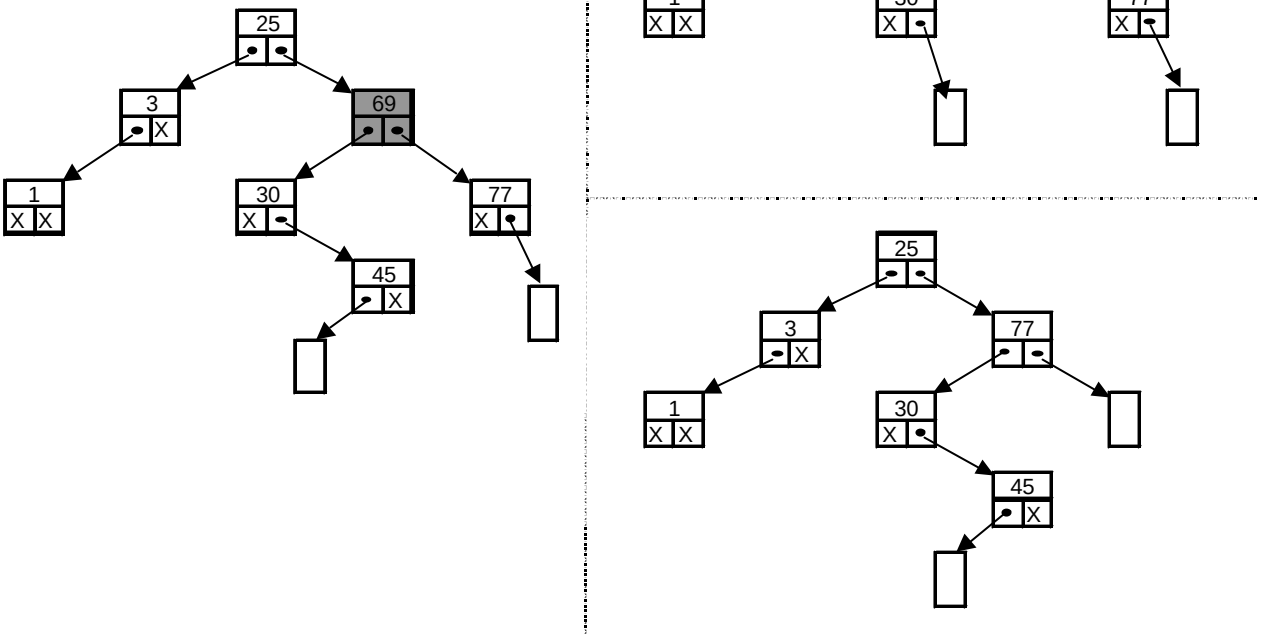

```
p := q^.izq
DEALLOCATE (q, SIZE (Nodo))
```

3.-El nodo a eliminar tiene dos subárboles descendientes

Presenta la dificultad de que la eliminación del nodo no es directa, ya que el puntero que apunta al nodo a eliminar no puede ser reasignado a los dos punteros izq y dch del nodo eliminado.

Hay dos posibles soluciones:

- Sustituir el nodo eliminado por el nodo más a la derecha de su subárbol izquierdo, es decir, toma su lugar el nodo de llave mayor, de entre todos los menores que él.
- Sustituirlo por el nodo más a la izquierda de su subárbol derecho, es decir, toma su lugar el nodo con llave menor de entre todos los mayores que él.



Análisis:

Para buscar en el árbol, el número de comparaciones a realizar dependerá del número de nodos que debe consultarse, o lo que es lo mismo, del recorrido a realizar.

En el peor caso será *n/2* y se da cuando se genera una lista enlazada. En el mejor caso es log *n* cuando el árbol está perfectamente balanceado.

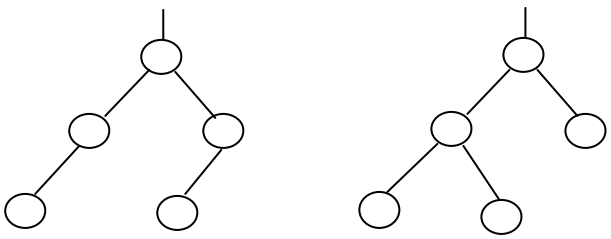
El número de comparaciones promedio para encontrar una llave en un árbol de búsqueda con *n* nodos es del orden del 39% mayor que las correspondientes a un árbol perfectamente balanceado (aprox. log *n*). Por tanto no se justifica el costo necesario para convertir el árbol de búsqueda en un árbol perfectamente balanceado en cada inserción.

5.5.- Árboles de búsqueda balanceados (AVL)

Resulta apropiado incorporar a los árboles de búsqueda alguna restricción en cuanto a su forma de crecimiento que evite las posibles construcciones poco eficaces.

- Se dice que un *árbol binario* está *balanceado* si y solo si en cada nodo las *alturas* de sus dos subárboles difieren como máximo en 1

Evidente todos los árboles perfectamente balanceados son AVL pero no al revés:



Árbol perfectamente balanceado AVL (no perfectamente balanceado

Los árboles AVL son de búsqueda. Por tanto la ***inserción*** se hará en principio igual que en los árboles de búsqueda. Pero además, incorporan un criterio de equilibrio.

Supongamos que se va a insertar un elemento en un subárbol con *N* como nodo padre y con subárboles *I* y *D* terminales de alturas h_I y h_D. Antes de insertar un elemento, *N* puede encontrarse de tres formas diferentes: $h_I = h_D$, $h_I < h_D$ o $h_I > h_D$

Consideremos que el nuevo nodo se inserta en *I*, entonces:

1.- Si N tenía $h_I = h_D$, entonces el árbol seguirá siendo AVL con
2.- Si N tenía $h_I < h_D$, entonces el árbol seguirá siendo AVL con
3.- Si N tenía $h_I > h_D$ entonces el árbol no será AVL

En el tercer caso será necesario manipular el árbol para que siga siendo AVL (*rebalanceo*). Para realizar el rebalanceo se deberá guardar información sobre el equilibrio en cada nodo. La siguiente definición de los nodos introduce un campo balance que se calcula como $bal(N)=h_D - h_I$

```
TYPE  Ptr_Nodo  =  POINTER  TO  Nodo;
TYPE  Nodo  =  RECORD
        llave  :  Tipo_llave;
        izq,  dch  :  Ptr_Nodo;
        bal  :  Balance
        END
```

5.5.1.- Inserción de un nuevo nodo por la izquierda de un subárbol

- La situación inicial en la que deberá realizarse rebalanceo es:

$$bal(N) = -1 \quad y \quad bal(N_I) = 0$$

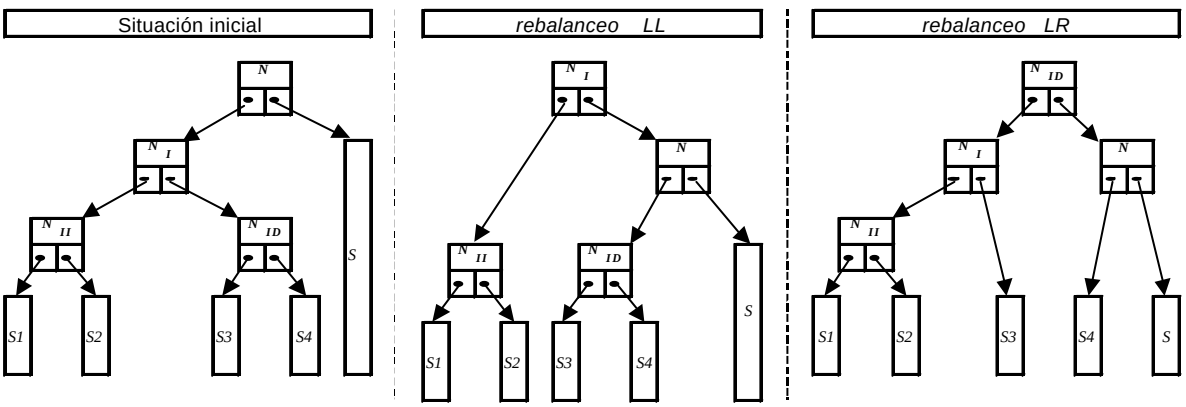
a) El nuevo nodo se inserta *en el subárbol izquierdo de* N_I: **rebalanceo LL - rotación simple**

- El nodo N_I toma el lugar de *N*

y se reasigna el subárbol derecho de N_I al subárbol izquierdo de *N*

b) El nuevo nodo se inserta *en el subárbol derecho de* N_I: **rebalanceo LR - rotación doble**

- N_{ID} se coloca entre *N* y N_I, colocando el nodo N_I como su hijo izquierdo y *N* como su hijo derecho.
- El subárbol izquierdo de N_{ID} pasa a ser el subárbol derecho de N_I y el subárbol derecho de N_{ID} pasa a ser subárbol izquierdo de *N*

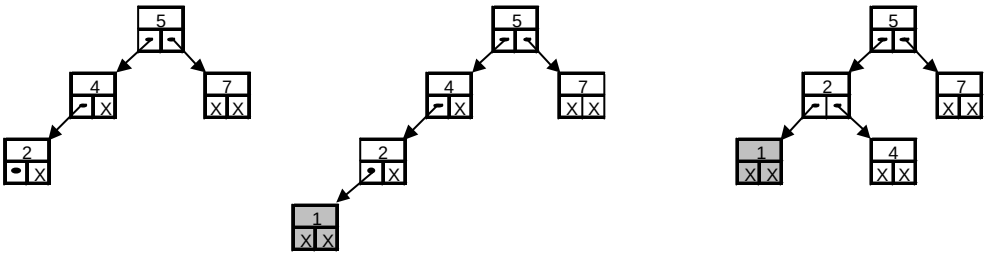


Los rebalanceos, cuando el nuevo nodo se inserta por la derecha de N se desarrollan igual que los anteriores teniendo en cuenta la simetría de ambas situaciones.

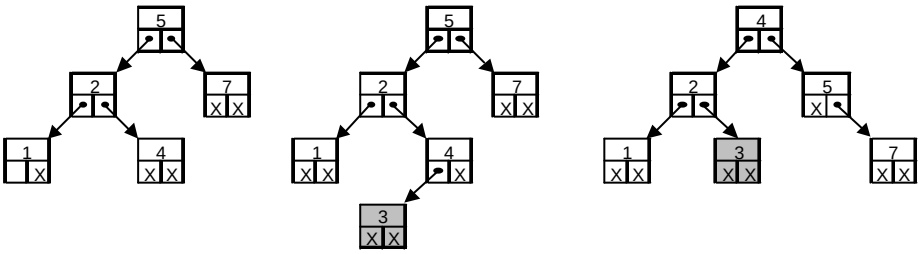
El proceso de inserción está formado por tres partes:

1. Buscar siguiendo la trayectoria de búsqueda, con lo que se distinguirá la inserción por la izquierda o por la derecha
2. Insertar el nodo y determinar su balance
3. Retroceder y verificar el factor de balance en cada nodo, realizando el rebalanceo en caso necesario.

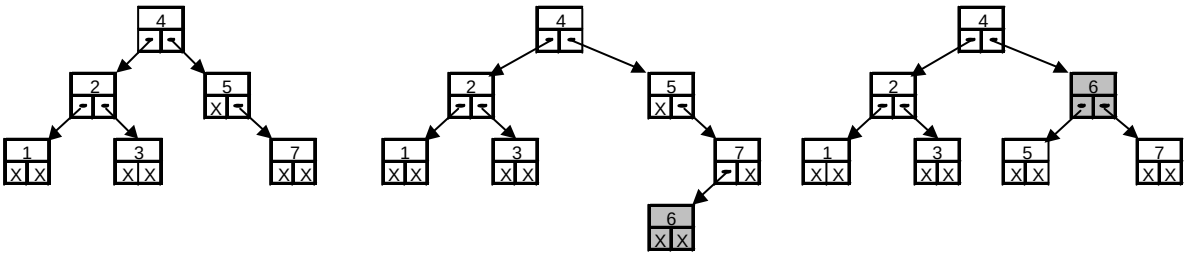
Ejemplos de inserción: Partiendo del árbol de la figura insertar los datos 1, 3 y 6



Al insertar el dato 1 se necesita rebalanceo LL para N=4



Al insertar el dato 3 se necesita rebalanceo LR para N=5



Al insertar el dato 6 se necesita rebalanceo RL para N=5

5.5.2.- Eliminación en árboles AVL

Debe tenerse en cuenta las mismas consideraciones que para la eliminación en los árboles de búsqueda más los rebalanceos necesarios.

La supresión de los nodos terminales y la de los nodos con un único descendiente es directa.

Si el nodo que debe suprimirse tiene dos subárboles deberá mantenerse el árbol de búsqueda, sustituyéndolo tras su eliminación por el nodo más a la izquierda de su subárbol derecho o el más a la derecha de su subárbol izquierdo.

Tras la sustitución del nodo a eliminar la altura habrá cambiado y tendremos que inspeccionar el árbol y hacer los rebalanceos necesarios.

Los rebalanceos son parecidos a los presentados en la inserción, pero no iguales. Cuando se elimina un nodo por la *izquierda* será necesario rebalanceo cuando el balance del nodo *N* sea 1, puesto que al eliminar el nodo por la izquierda el subárbol quedaría *cargado* en 2 por la derecha. El rebalanceo necesario es *RR.*

Cuando el balance de *N* es 0, al eliminar por la izquierda aumenta a 1 pero su altura no ha disminuido.

Cuando el balance de *N* es –1, pasa a 0 y ha variado la altura del árbol. Será una rotación *simple* o *doble* dependiendo del balance del descendiente derecho de *N*, N_D. Si es 1 la rotación es *RR*, si es 0 también es *RR* pero los balances son diferentes puesto que N_D tiene subárboles derecho e izquierdo; y si es –1, entonces la rotación es *RL*.

Cuando se elimina un nodo por la *derecha* será necesario rebalanceo cuando el balance del nodo *N* sea –1, puesto que al eliminar el nodo por la derecha el subárbol quedaría *cargado* en 2 por la derecha. El rebalanceo necesario es *LL*.

Si el balance de *N* es 0, al eliminar por la derecha pasa a –1 pero su altura no ha disminuido.

Cuando el balance de *N* es 1, pasa a 0 y ha variado la altura del árbol. Será una rotación *simple* o *doble* dependiendo del balance del descendiente izquierdo de *N*, N_I. Si es 1 la rotación es *LL*, si es 0 también es *LL* pero los balances son diferentes puesto que N_I tiene subárboles derecho e izquierdo; y si es 1, entonces la rotación es *LR.*

Análisis:

El objetivo fundamental en el análisis de árboles AVL es conocer cuánto puede crecer como máximo, es decir, la altura en el peor de los casos.

Teorema: La altura de un árbol balanceado con *n* nodos siempre es mayor o igual a $\log(n + 1)$ y menor que $1.4404 \log(n + 2) – 0.328$.

El teorema indica que la altura de árbol AVL nunca es mayor que el 45 por ciento respecto a su árbol perfectamente balanceado. Por tanto, se ha mejorado sensiblemente el comportamiento de los árboles de búsqueda.

En cuanto al procedimiento de inserción, las pruebas experimentales indican que el rebalanceo es necesario cada dos inserciones, siendo igualmente probables las rotaciones simples y dobles, y que la altura esperada $h_{esp} = \log(n + 0.25)$. Por tanto, el árbol AVL es tan satisfactorio como el perfectamente balanceado, siendo además mucho más fácil de mantener.

La eliminación puede necesitar una rotación en cada nodo de la trayectoria de búsqueda. Sin embargo, los resultados empíricos muestran que sólo es necesario rebalanceo en una de cada cinco eliminaciones, con lo que en definitiva su comportamiento es comparable al de la inserción.

Los árboles AVL deben utilizarse sólo en el caso de que las operaciones de búsqueda sean más frecuentes que las de inserción.

En conclusión, las operaciones de búsqueda, inserción y eliminación tienen un costo del orden de $\log n$. Por tanto, los árboles AVL pueden considerarse como la mejor solución de compromiso entre el equilibrio y la organización ordenada del árbol.

6.- ÁRBOLES AVANZADOS

Se denominan árboles multicamino a aquellos árboles de grado mayor que dos. Son muy utilizados en la construcción y mantenimiento de árboles de búsqueda con gran cantidad de nodos, y por tanto, usualmente almacenados en memoria secundaria.

Se organizan de manera que un árbol se subdivide en subárboles y éstos se representan como unidades a las que se accede simultáneamente y reciben el nombre de *páginas.*

Cada acceso a página requiere un único acceso a memoria secundaria.

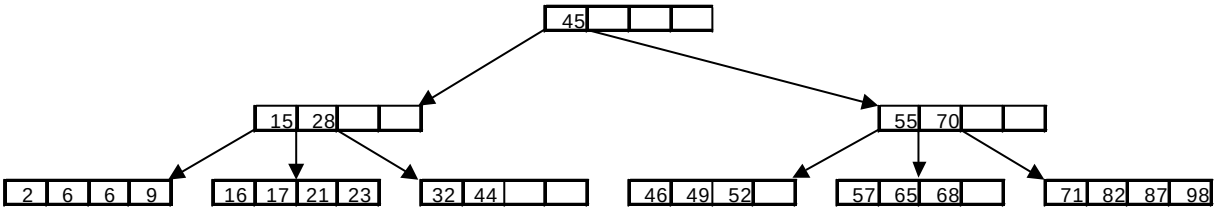
6.1.- Árboles B

Un *árbol B de orden n* es aquel árbol de búsqueda que satisface las siguientes propiedades:

1. Cada página contiene como máximo *2n* llaves.
2. Cada página contiene como mínimo *n* llaves, excepto la raíz que puede contener sólo una
3. Cada página o es una página de hoja o tiene *m + 1* descendientes, siendo *m* el número de llaves en esta página
4. Todas las páginas de hoja aparecen al mismo nivel.

Presentan dos características: un árbol B con *N* elementos requiere en el peor caso $\log_n N$ accesos de página y las páginas están como mínimo llenas por la mitad (factor uso memoria > 50%).

Ejemplo de árbol B de orden 2:



a) Satisface las condiciones de árbol de búsqueda. b) cada página contiene como máximo 2x2 = 4 llaves, y como mínimo 2 llaves, excepto la raíz que almacena una sola. c) todas las páginas de hoja están en el mismo nivel, el nivel 3, y las que no son de hoja tienen m+1 descendientes. Así la raíz con una sola llave (m=1) apunta a dos páginas, y en el segundo nivel, las páginas contienen dos llaves (m=2) y por tanto tienen tres descendientes.

En general las páginas de un árbol B se representan con *m* llaves y *m+1* apuntadores y las búsquedas se realizan en base a los siguientes criterios:

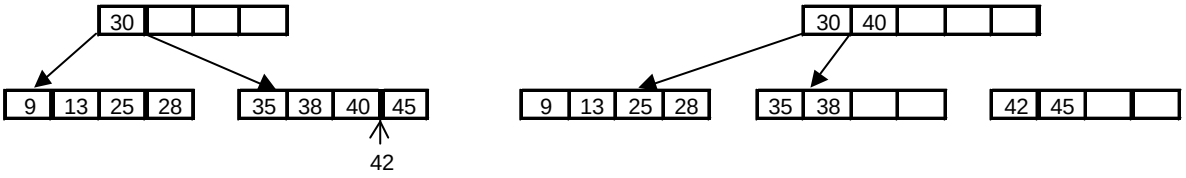
1. Si el argumento de búsqueda se encuentra entre dos llaves de la página, $ll_i < x < ll_{i+1}$ entonces la búsqueda sigue en la página apuntada por el puntero entre ellas, p_i
2. Si el argumento de búsqueda es mayor que la última llave de la página, $ll_m < x$, entonces se sigue la búsqueda en la página apuntada por el último apuntador, p_m
3. Si el argumento de búsqueda es menor que la primera llave de la página, $x < ll_1$, entonces se sigue la búsqueda en la página apuntada por el primer apuntador, p_0
4. Si en cualquier caso el apuntador a la página en la que hay que seguir la búsqueda es *NIL* entonces el argumento de búsqueda *x* no está en el árbol y finaliza la búsqueda.

La inserción se realiza buscando el lugar donde debe insertarse de la misma manera en que se realizó la operación de búsqueda. Localizada dicha posición si la página correspondiente tiene $m<2n$ llaves se inserta en ella en la posición dada por el criterio de búsqueda, pero si la página está llena ($m=2n$) entonces el árbol debe reorganizarse:

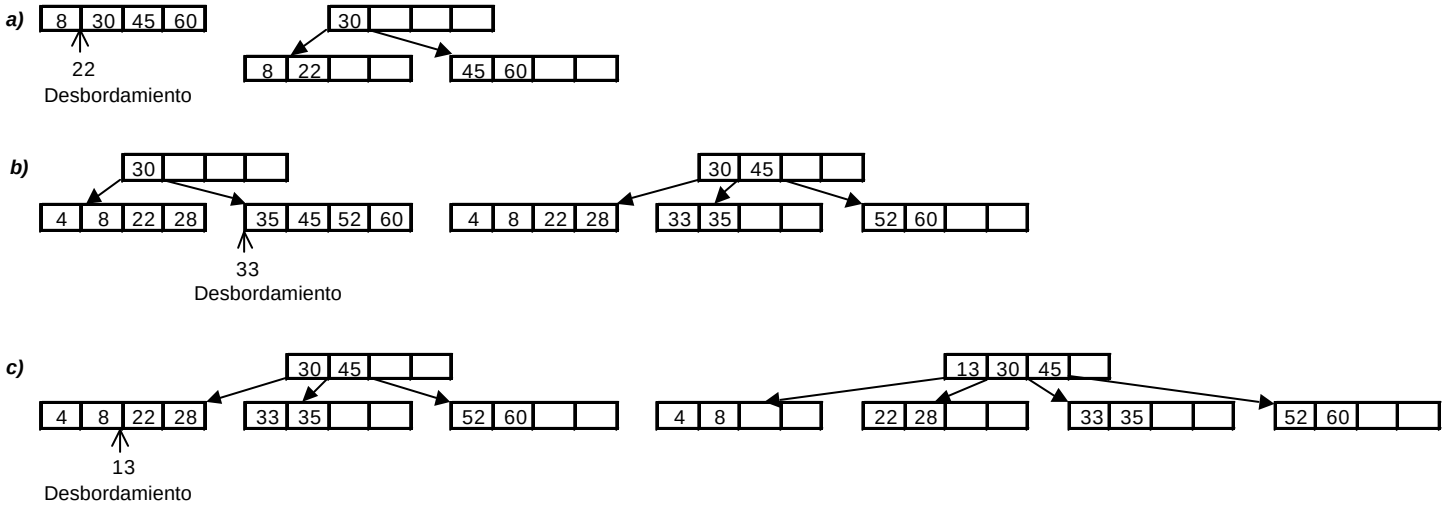
1. La página se divide en dos nuevas páginas.
2. La llave que se encontraba en la mitad, ll_{mitad}, se sube un nivel colocándola en la página madre. A la hora de determinar la llave central se tiene en cuenta también la nueva llave que se pretende insertar.
3. Las páginas que se encontraban a la izquierda se distribuyen en la página nueva apuntada por la izquierda de ll_{mitad} y las que se encontraban a la derecha se distribuyen en la página nueva apuntada por la derecha de ll_{mitad}.

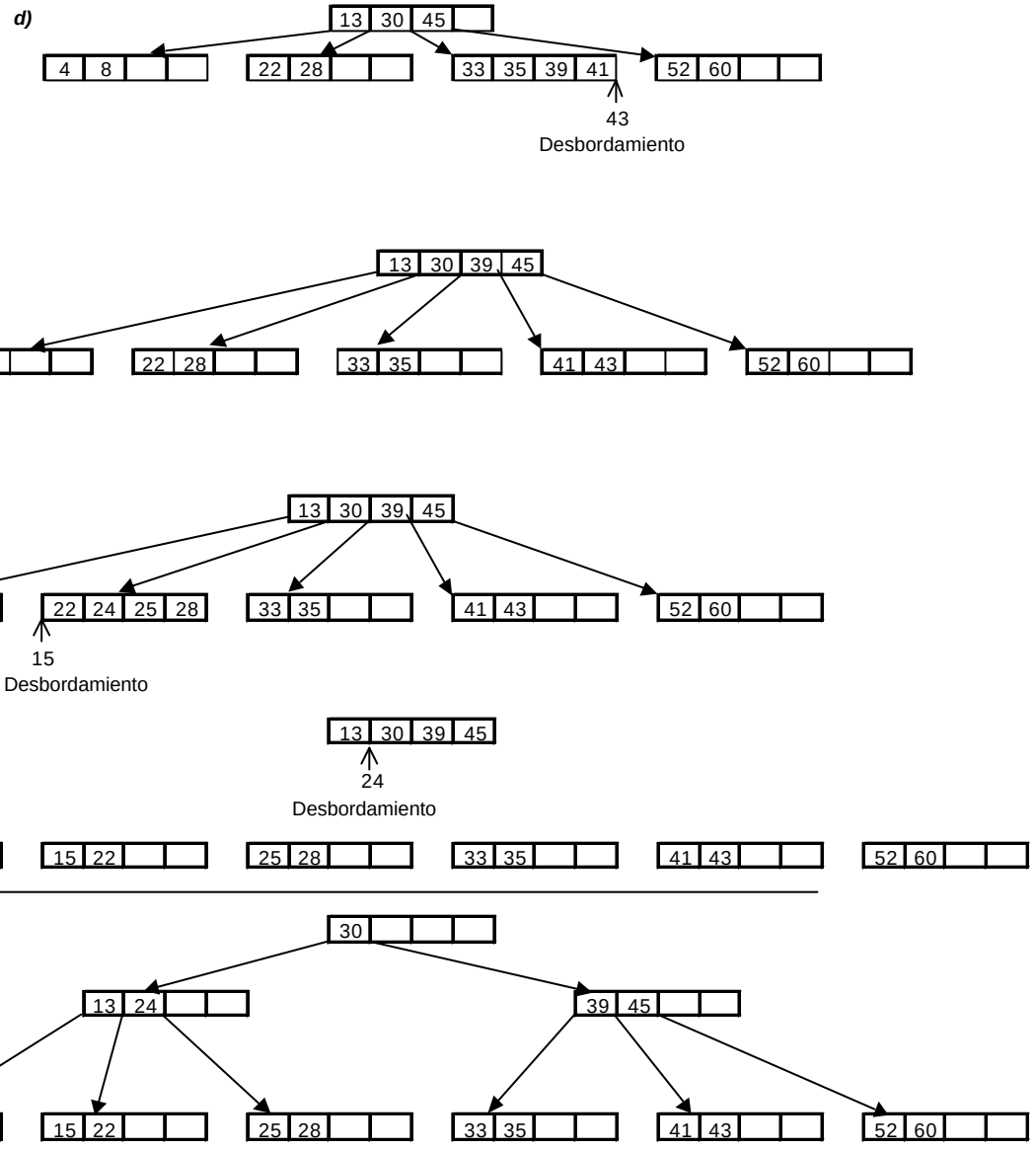
La división de páginas puede propagarse hasta la raíz. Es obvio que el árbol crece de las hojas a la raíz.

Ejemplo: inserción del dato 42 en un árbol B de orden 2



Ejemplo: supóngase que en un árbol B de orden dos, inicialmente vacío, se introducen consecutivamente las siguientes llaves: 30, 60, 45, 8, 22, 35, 4, 28, 52, 33, 13, 39, 41, 43, 24, 25, 15.





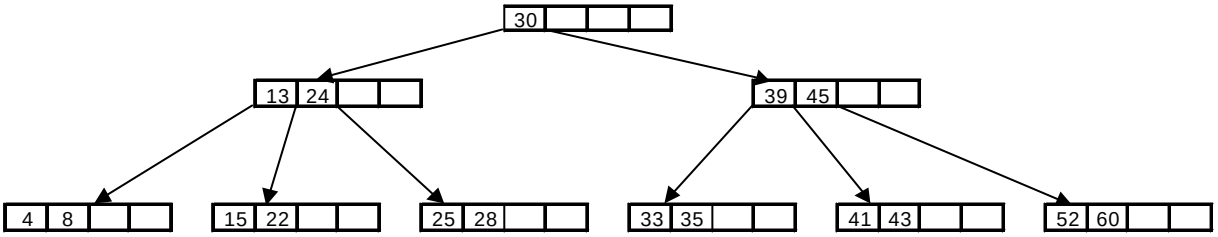
La eliminación en árboles B de orden 2 es similar a la de todo árbol de búsqueda.
Una vez localizado el elemento a eliminar, si se encuentra en una página de hoja la eliminación será directa, mientras que si no es una página de hoja seguiríamos una estrategia similar a la utilizada en la eliminación de árboles binarios de búsqueda.

En cualquier caso, es imprescindible que como mínimo cada página tenga *n* elementos.

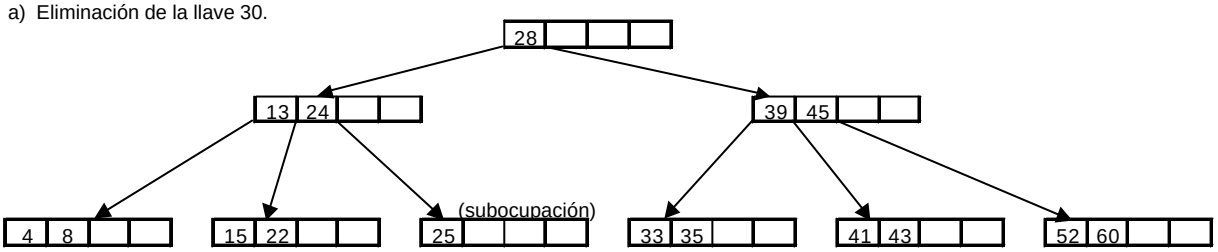
Si tras la eliminación este número es inferior, se produce *subocupación*, que puede ser resuelta mediante la estrategia habitual de rebalanceo de páginas.

En este caso consiste en tomar los elementos de dos páginas contiguas más el que las apunta desde la página madre, combinarlos en una sola página, y si existe *sobreocupación* se sube el elemento central.

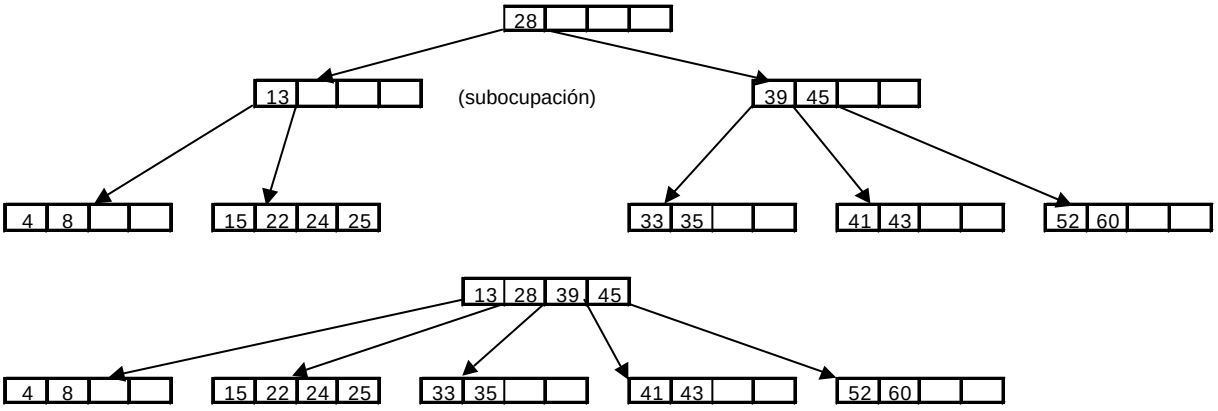
Ejemplo: vamos a eliminar consecutivamente las llaves 30, 35 y 60



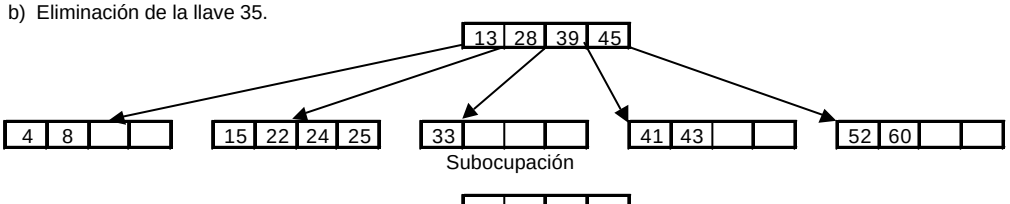
a) Eliminación de la llave 30.

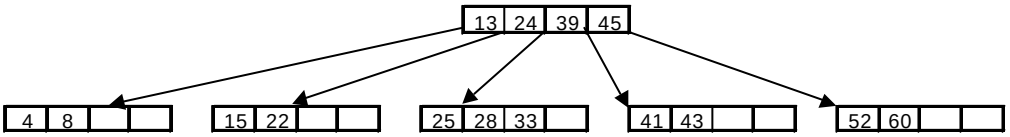


Se produce subocupación en la página que contenía la llave 28

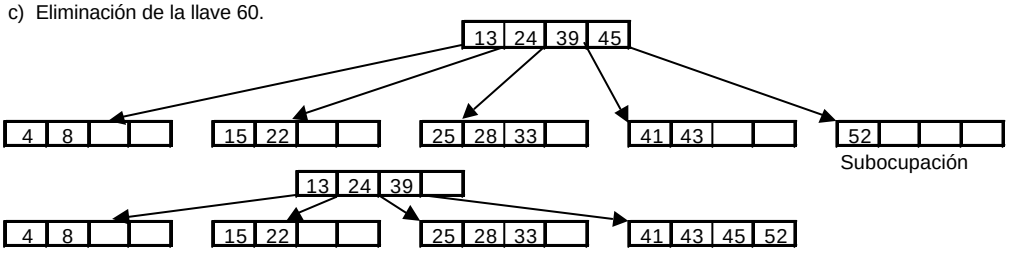


b) Eliminación de la llave 35.





c) Eliminación de la llave 60.



6.2.- Árboles B binarios

Un *árbol B Binario* (BB) es un árbol B de orden 1.

Consta de páginas (nodos) con una o dos llaves, y cada página contiene dos o tres punteros a los descendientes (árboles 2-3).

Todas las páginas de hoja aparecen al mismo nivel, y todas las páginas que no son hoja tienen dos o tres descendientes, incluida la raíz.

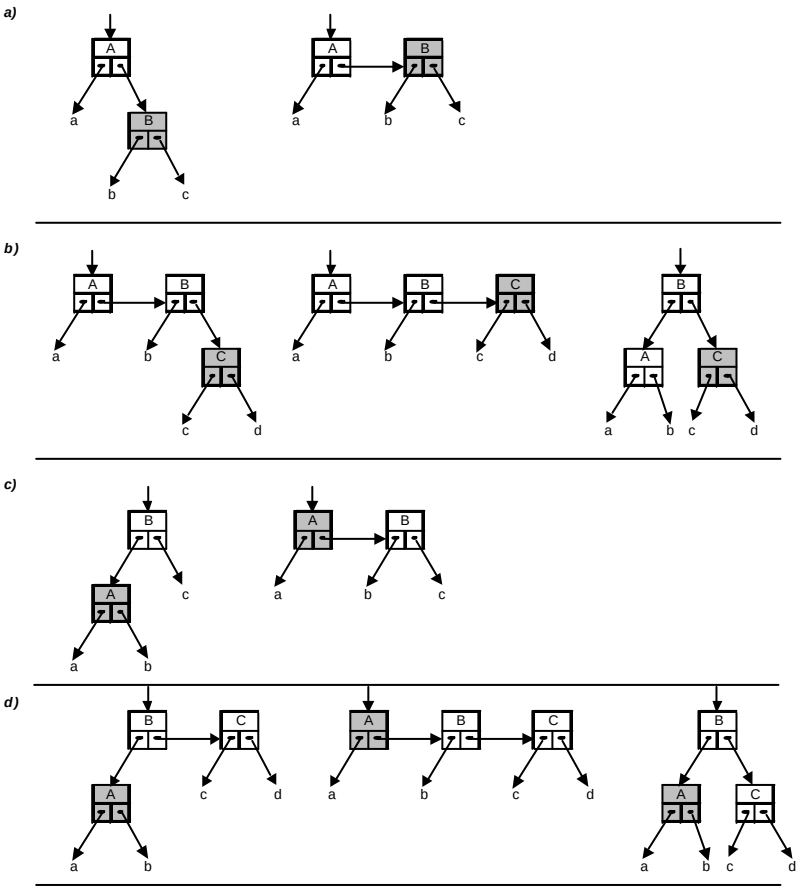
Los apuntadores pueden ser *verticales* u *horizontales.* En los árboles BB el apuntador horizontal tiene un *sentido* único, por convenio, hacia la derecha.

La definición en Modula2 es:

```
TYPE  Ptr_Nodo  =  POINTER  TO  Nodo;
TYPE Nodo  =  RECORD
                 llave  :  INTEGER;
                 . . . . . .
                 izq, dch  :  Ptr_Nodo;
                 h  :  BOLEAN (*rama horizontal a la derecha*)
                 END
```

Con esta estructura se garantiza una longitud de trayectoria máxima $L_{max} = 2\lceil \log N \rceil$

Inserción en árboles BB: a) Inserción de hermano. b) Tres nodos en la misma página que obliga a dividir en dos niveles. c) Inserción por la izquierda en un nodo sin hermano. d) Inserción por la izquierda de un nodo con hermano.



6.3.- Árboles B Binarios Simétricos (BBS)

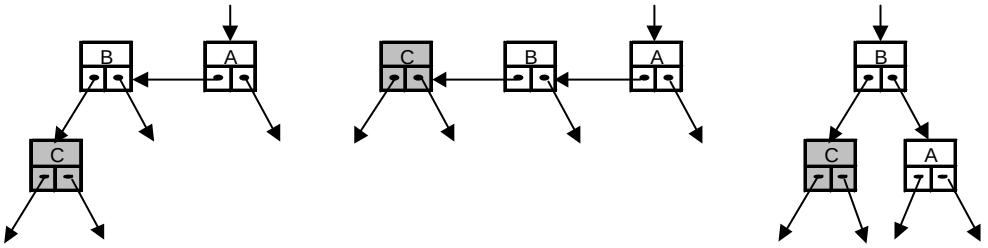
Los *árboles B Binarios Simétricos* se definen con la idea de garantizar un crecimiento simétrico de los subárboles derecho e izquierdo del árbol. Se definen como árboles de búsqueda que satisfacen las siguientes propiedades:

1. Todo nodo contiene una llave y, como máximo, dos apuntadores a subárboles.
2. Todo apuntador es horizontal o vertical. *No existen dos apuntadores consecutivos horizontales en ninguna trayectoria de búsqueda*
3. Todos los nodos terminales aparecen al mismo nivel.

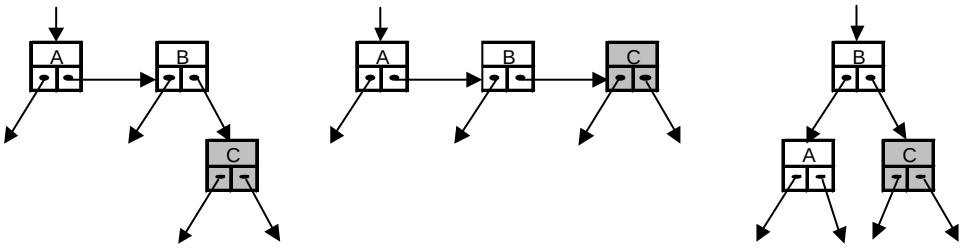
Un árbol BBS con n nodos no puede tener una altura superior a $\log n$, por tanto, la longitud de trayectoria de búsqueda es como máximo $L_{max} = 2\lceil \log n \rceil$

En la inserción en árboles BBS puede darse cuatro casos en los que es necesario rebalanceo:

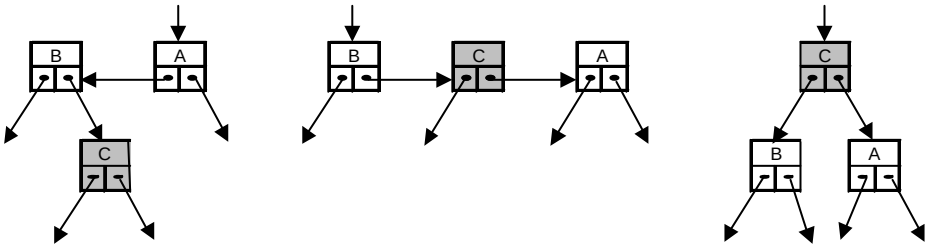
a) Inserción y rebalanceo LL en un árbol BBS



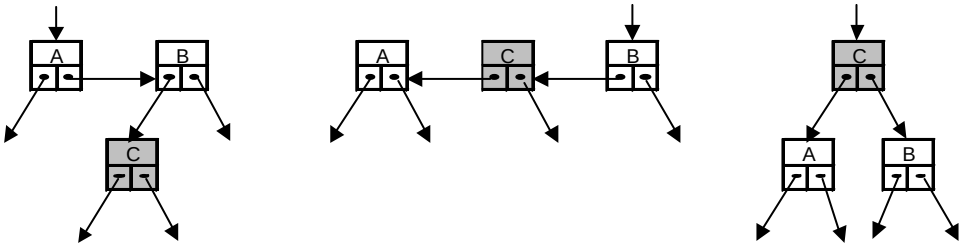
b) Inserción y rebalanceo RR en un árbol BBS



c) Inserción y rebalanceo LR en un árbol BBS



d) Inserción y rebalanceo RL en un árbol BBS



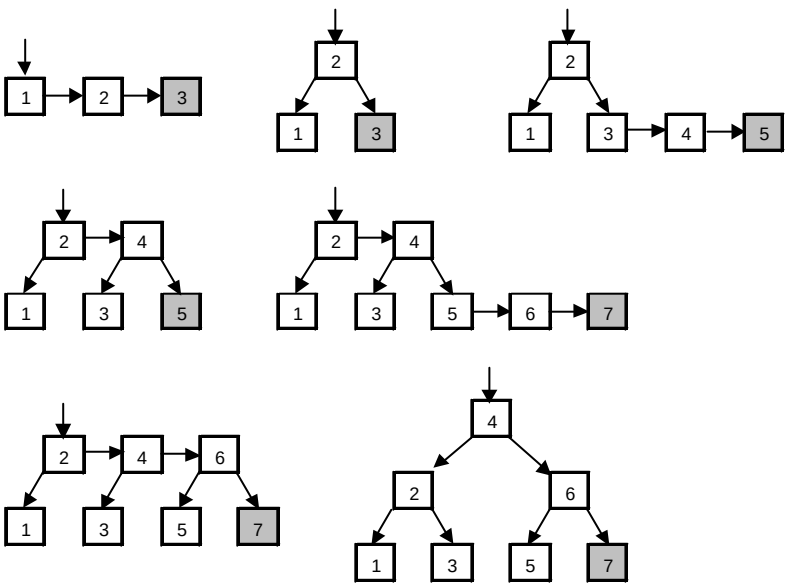
La declaración en Modula2 del nodo de un árbol BBS es:

```
TYPE  Ptr_Nodo  =  POINTER  TO  Nodo;
TYPE Nodo  =  RECORD
                 llave  :  INTEGER;
                 cont  :  CARDINAL
                 izq, dch  :  Ptr_Nodo;
                 h_izq , h_dch  :  BOLEAN (*rama horizontal*)
                 END
```

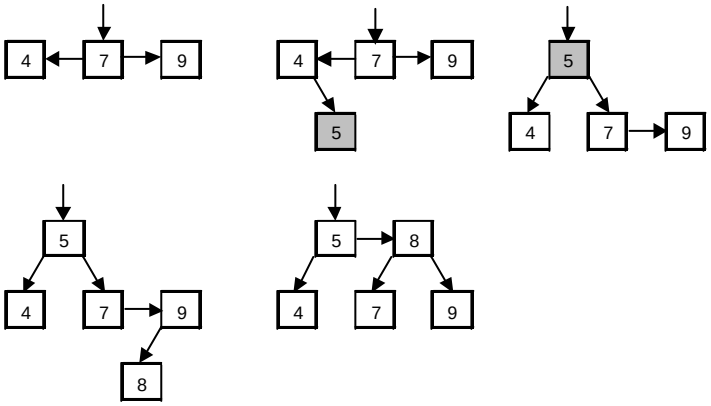
Además se utiliza la variable entera h para indicar el crecimiento del árbol con el siguiente significado:

1. h = 0 : el subárbol *p* no requiere cambios en la estructura
2. h = 1 : el nodo *p* ha obtenido un hermano
3. h = 2 : el subárbol *p* ha aumentado de altura

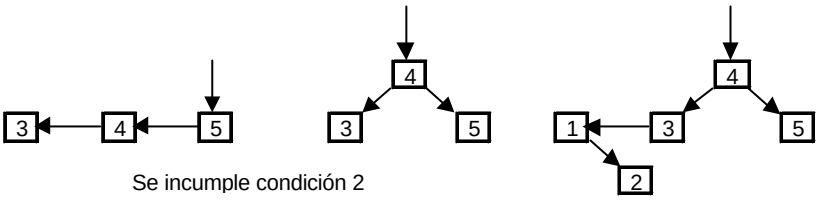
Ejemplo1: Supongamos que en un árbol BBS inicialmente vacío se insertan consecutivamente las llaves: 1, 2, 3, 4, 5, 6, 7.



Ejemplo2: Supongamos ahora que partiendo nuevamente de un árbol BBS inicialmente vacío se insertan consecutivamente las llaves: 7, 4, 9, 5, 8.



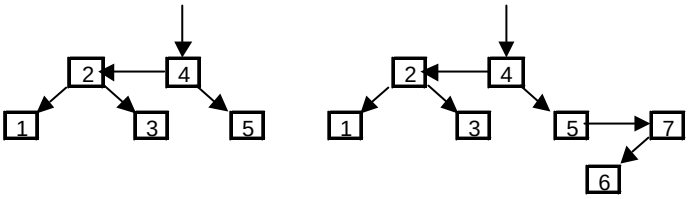
Ejemplo3: Supongamos ahora que partiendo nuevamente de un árbol BBS inicialmente vacío se insertan consecutivamente las llaves: 5, 4, 3, 1, 2, 7, 6.



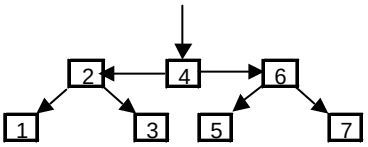
Se incumple condición 2
Subimos nodo central un nivel

Se incumple condición 3
Subimos nodo 2 de nivel

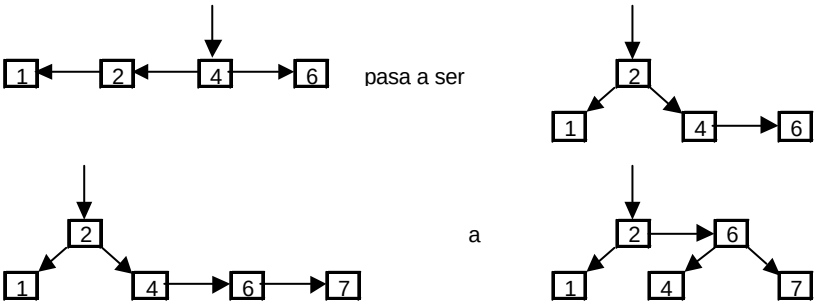
Al subir el nodo 2 se incumple condición 2 (nodos 1-2-3) por lo que se vuelve a subir de nivel el nodo central (2). Se cambia el enlace izquierdo del nodo raíz.



Se procede de manera análoga para el nodo 6 y se tiene



Ejemplo4: Supongamos ahora que partiendo nuevamente de un árbol BBS inicialmente vacío se insertan consecutivamente las llaves: 4, 6, 2, 1, 7, 3, 5.



Finalmente quedará