TEMARIO

I. **Diseño y análisis de algoritmos.**

- I.1 Conceptos básicos.
- I.2 Diseño de algoritmos.
- I.3 Computabilidad y complejidad de algoritmos.

II. **Tipos de datos y TAD'S.**

- II.1 Abstracción.
- II.2 TAD.
- II.3 Especificación e implementación.

III. **Recursividad**. Definición y conceptos básicos.

IV. **Tablas**.

- IV.1 Estructura, representación y operaciones básicas.
- IV.2 Transformación de clases (HASH).

V. **Listas**.

- V.1 Pilas y colas.
- V.2 Listas lineales.
- V.3 Listas secuenciales.
- V.4 Listas encadenadas.

VI. **Arboles**.

- VI.1 Estructura y representación.
- VI.2 Operaciones básicos.
- VI.3 Arboles equilibrios.
- VI.4 Arboles multicaminos y binarios.

VII. **Grafos**.

- VII.1 Fundamento matemático.
- VII.2 Representación.
- VII.3 Algoritmo de manipulación.

VIII. **Clasificación y búsqueda**.

- VIII.1 Conceptos básicos.
- VIII.2 Clasificación interna, métodos.
- VIII.3 Clasificación externa, métodos.
- VIII.4 Búsqueda.

IX. **Resolución general de problemas**. BACKTRAKING.

PROFESOR: **Miguel Angel Díaz**, Despacho 2117 bloque II.

1. CONCEPTOS BASICOS.

Una posible definición de algoritmo es un conjunto de reglas que permiten obtener un resultado determinado apartir de ciertas reglas definidas.

Otra definición sería, algoritmo es una secuencia finita de instrucciones, cada una de las cuales tiene un significado preciso y puede ejecutarse con una cantidad finita de esfuerzo en un tiempo finito. Ha de tener las siguientes características: Legible, correcto, modular, eficiente, estructurado, no ambiguo y a ser posible se ha de desarrollar en el menor tiempo posible.

Características de un algoritmo de computador:

Ser algoritmo: Tiene que consistir en una secuencia de instrucciones claras y finitas.

Ser correcto: El algoritmo ha de resolver el problema planteado en todas sus facetas.

Ser legible.

Ser eficiente: Es relativa porque depende de la maquinas en la que lo ejecutemos. Existen ejemplos de algoritmos eficientes que ocupan demasiado espacio para ser aplicados sin almacenamiento secundario lento, lo cual puede anular la eficiencia.

Un algoritmo eficiente pero complicado puede ser inapropiado porque posteriormente puede tener que darle mantenimiento otra persona distinta del escritor.

2. DISEÑO DE ALGORITMOS.

Fases de diseño de algoritmos.

1. Diseño: se dan las especificaciones en lenguaje natural y se crea un primer modelo matemático apropiado. La solución en esta etapa es un algoritmo expresado de manera muy informal.

2. Implementación: El programador convierte el algoritmo en código, siguiendo alguna de estas 3 metodologías.

A. TOP-DOWN se alcanza el programa sustituyendo las palabras del palabras del pseudocódigo por secuencias de proposiciones cada vez mas detalladas, en un llamado refinamiento progresivo.

B. BOTTON-UP parte de las herramientas mas primitivas hasta que se llega al programa.

C. TAD'S modularización dependiendo de los recursos.Tenemos unas estructuras abstractas implementadas, y una serie de conocimientos asociados a esos recursos.

3. Pruebas: Es un material que se pasa al programa para detectar posibles errores.Esto no quiere decir que el diseño no tenga errores, puede tenerlos para otros datos.

3. COMPLEJIDAD DE ALGORITMOS.

La eficiencia de un determinado algoritmo depende de la maquina, y de otros factores externos al propio diseño. Para comparar dos algoritmos sin tener en cuenta estos factores externos se usa la complejidad. Esta es una media informativa del tiempo de ejecución de un algoritmo, y depende de varios factores:

¨ Los datos de entrada del programa. Dentro de ellos, lo más importante es la cantidad, su disposición, etc.

¨ La calidad del código generado por el compilador utilizado para crear el programa.

¨ La naturaleza y rapidez de las instrucciones empleados por la máquina y la propia máquina.

¨ La propia complejidad del algoritmo base del programa.

El hecho de que el tiempo de ejecución dependa de la entrada, indica que el tiempo de ejecución del programa debe definirse como una función de la entrada. En general la longitud de la entrada es una medida apropiada de tamaño, y se supondrá que tal es la medida utilizada a menos que se especifique lo contrario.

Se acostumbra, pues, a denominar **T(n)** al tiempo de ejecución de un algoritmo en función de **n** datos de entrada.Por ejemplo algunos programas pueden tener un tiempo de ejecución. $T(n)=Cn^2$, donde C es una constante que engloba las características de la máquina y otros factores.

Las unidades de T(n) se dejan sin especificar, pero se puede considerar a T(n) como el numero de instrucciones ejecutadas en un computador idealizado, y es lo que entendemos por complejidad.

Para muchos programas, el tiempo de ejecución es en realidad una función de la entrada especifica, y no sólo del tamaño de ella.En cualquier caso se define T(n) como el tiempo de ejecución del peor caso, es decir, el máximo valor del tiempo de ejecución para las entradas de tamaño n.

¨ Un algoritmo es de orden polinomial si T(n) crece mas despacio, a medida que aumenta n, que un polinomio de grado n. Se pueden ejecutar en un computador.

¨ En el caso contrario se llama exponencial, y estos no son ejecutables en un computador.

3.1 Notación O grande.

Para hacer referencia a la velocidad de crecimiento de los valores de una función se usara la notación conocida como "O grande".Decimos que un algoritmo tiene un orden O(n) si existen y un n_0 y un c, siendo c>0, tal que para todo $n>=n_0$, $T(n)<=c \times f(n)$.
C estara determinado por: -Calidad del código obtenido por el compilador.ç
-Características de la propia máquina.

Ejemplo:

$$T(n) = (n+1)^2$$
$$n^2+2n+1 <= c-n^2$$
$$n >= n_0$$
$$n_0 = 1$$

Orden n^2 es $O(n^2)$

3.2 Propiedades de la notación O grande.

Si multiplicamos el orden de una función por una constante, el orden del algoritmo sigue siendo el mismo.

$$O(c-f(n)) = O(f(n))$$

La suma del orden de dos funciones es igual al orden de la mayor.

$$O(f(n)+g(n)) = O(máx(f(n)*g(n)).$$

Si multiplicamos el orden de dos funciones el resultado esta multiplicación de los ordenes.

3.4 Orden de crecimiento de funciones conocidas.

O(1) < O(log(n)) < O(n) < O(log(n)n) < O(n^k) < O(k^n)

3.5 Reglas de calculo de complejidad de T(n).

El tiempo de ejecución de cada sentencia simple, por lo común puede tomarse como O(1).
El tiempo de ejecución de una secuencia de proposiciones se determina por la regla de la suma. Es el máximo tiempo de ejecución de una proposición de la sentencia.
Para las sentencias de bifurcación (IF,CASE) el orden resultante será el de la bifurcación con mayor orden.
Para los bucles es el orden del cuerpo del bucle sumado tantas veces como se ejecute el bucle.
El orden de una llamada a un subprograma no recursivo es el orden del subprograma.

Ejercicio 1.

Para realizar una tarea disponemos de dos algoritmos a1 y a2, ambos son correctos.Para N datos de entrada, cada uno de los algoritmos consume los siguientes recursos:

	Tiempo	*Almacenamiento*
a1	1500n *Ciclos*.	1500n *bytes*.
a2	30000n *Ciclos*.	50n *bytes*.

	Memoria	*Duración ciclo*
m1	32 *Kbytes.*	20 *microseg.*
m2	2048 *Kbytes*.	200 *microseg.*

La maquina m1 tiene limitación de memoria, por tanto el algoritmo mas apropiado para esta maquina es el que usa menos memoria, el a2.En cambio, la maquina m2 tiene limitación de velocidad, luego usaremos el algoritmo mas rápido el a1.

Ejercicio 2.

El algoritmo a1, tarda $5n^2$ segundos en resolver un determinado problema con n datos de entrada, mientras que el a2 tarda n^2+400n ¿cual de los 2 es mas eficiente en función de los datos de entrada.?

$$5n^2 >= n^2+400n$$
$$4n^2 >= 400n$$
$$(n^2/n) >= (400/4) \quad \text{luego} \quad n >= 100$$

para n<=100 lo será el a1
para n>100 lo será el a2

Ejercicio 3.

Calcular la complejidad de los siguientes programas.

```
a/ BEGIN
     Error:=a+n-1>B;                  O(1)
     IF NOT Error
          THEN BEGIN
                  for i:=1 TO n do c:=c+1;          O(n)
                  for i:=1 TO n do d:=d+1;          O(n)
               END
   END
```

La complejidad es O(n)

```
b/ BEGIN
      RESET(fich);                  O(1)
      WHILE NOT EOF(fich) DO
      BEGIN
         READ (fich,ele);             O(1)
         for i:=1 TO m do elem[i]:=6 ;          O(m)
      END
   END
```

La complejidad es O(m*n) ya que el bucle WHILE se ejecuta 1 vez por cada uno de los n elementos del fichero.

```
c/ BEGIN
      i:=i+1;
      WHILE i<=n do
      BEGIN
         m.indice=1;
         j:=i+1;
         WHILE j<=n do
         BEGIN
            IF datos[s]<datos[m.indice] THEN m.indice:=1;
            j:=j+1
         END;
         intercambiar(i,j);                O(1)
         i:=i+1
      END
   END
```

$$\sum_{j=1}^{n}\left(O\left(\sum_{j=i+1}^{n} O(1) \right)\right)=\sum_{j=1}^{n}(n-i)=\sum_{j=1}^{n} n-\sum_{i=1}^{n} i = n^2-\frac{n+n^2}{2}$$

La complejidad es $O((n^2-n)/2)$

Ejercicio 4.

Dado un vector unidimensional con la siguiente declaración:

```
TYPE
  tipovector=ARRAY [1..n] OF 0..1
```

se pide diseñar un procedimiento de complejidad O(n) que ordene dicho vector.

```
BEGIN
   cont:=0;
   for i:=1 TO n do IF tabla[i]:=0 THEN cont:=cont+1;
   for i:=1 TO cont do tabla[i]:=0;
   for i:=cont+1 TO n do tabla[i]:=1
END.
```

Ejercicio 5.

En un vector declarado de la siguiente forma:

```
CONST max=...;
TYPE tserie=ARRAY [0..max] OF INTEGER;
```

se tiene almacenado una serie de longitud l, siendo 1 <= l <= max la longitud almacenada en la posición 0.Dicha serie esta ordenada ascendentemente y puede contener numeros repetidos (en cuyo caso aparecerán, como es lógico, en posiciones consecutivas.)

Diseñar un algoritmo que, sin usar ninguna estructura de datos auxiliar,"comprima" la serie de tal manera que aparezcan los mismos numeros de antes, y con el mismo orden, pero sin repeticiones. No se puede mover el contenido de una casilla mas de una vez.

Vamos a utilizar un indice para el número de elementos no repetidos y otro para recorrer la tabla.

```
BEGIN
   nrepet:=1;
   for i:=1 TO tabla[0] do
          IF tabla[nrepet] <> tabla[c]
                THEN BEGIN
                        nrepet:=nrepet+1;
                        tabla[nrepet]:=tabla[i]
                     END;
   tabla[0]:=nrepet
END.
```

Ejercicio 6.

Ordenar las siguientes funciones por orden de crecimiento:

a. n	f. n/log n
b. SQRT(n)	g. SQRT(n)×$\log^2 n$
c. log n	h. $(1/3)^n$
d. log (log n)	i. $(3/2)^n$
e. $\log^2 n$	j. 17

h < j < d < c < e < b < g < f < a < i

1. ABSTRACCION.

Para realizar un programa complicado, debemos dividirlo en partes que resuelvan subproblemas del problema original.En este caso el objetivo debe ser la descomposición del programa en módulos que sean ellos mismos programas de tamaño pequeño y que interactuen unos con otros de forma sencilla y bien definida.

En programación estructurada se recomienda usar la abstracción como forma de descomponer un problema.Al llevarla a cabo se ignoran ciertos detalles de manera que el problema original se vea en forma mas simple al atender solo a que operaciones hay que realizar, pero no a como hay que realizarlas.

El método de descomposición en base a abstracciones se continua aplicando hasta llegar a operaciones lo bastante sencillas como para ser programadas de forma directa.

2. TAD.

La denominación TAD engloba dos clases de abstracciones:

Abstracciones de datos.
Abstracciones funcionales.

Las primeras aparecen al abstraer el significado de los diferentes tipos de datos significativos que intervienen en el problema.Permiten definir nuevos tipos de datos especificando sus posibles valores y las operaciones que los manipulan.

Las segundas surgen al plantearse de una manera abstracta las operaciones significativas del problema.Son una herramienta muy poderosa que permite dotar a una aplicación de operaciones que no están definidas directamente en el lenguaje en el que estamos trabajando.

En resumen, un TAD es un conjunto de valores y unas operaciones definidas sobre esos valores.Cada vez que deseemos emplear el TAD solo lo podemos hacer con las operaciones definidas, incluso no sabiendo como están implementadas.

2.1 Características de los TAD.

× Cada modulo es una abstracción del problema, no conocemos el detalle.

× Se puede realizar una especificación breve y precisa del TAD.

× Con esta especificación se puede usar correctamente el TAD.

× Si realizamos, no se va a alterar considerablemente la especificación.

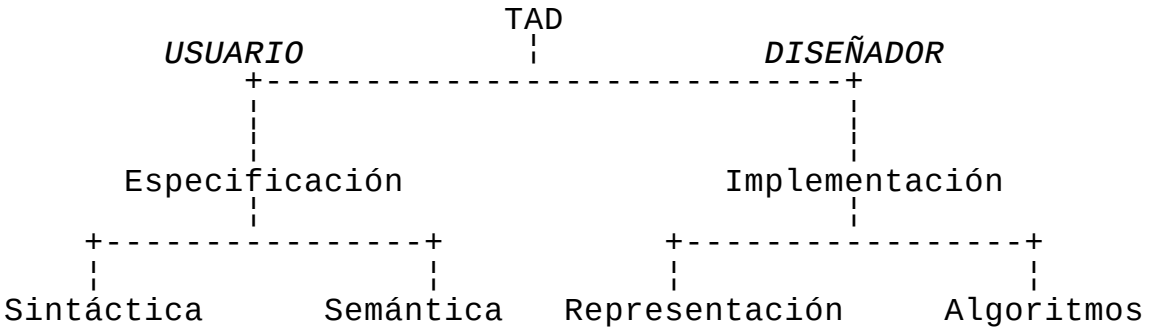
× Cada modulo TAD va a poder ser compilado por separado.

× Las comunicaciones entre los módulos van a ser pocas y claras.

× Cada modulo debe tener un tamaño razonable.

3. ESPECIFICACION E IMPLEMENTACION.

Esquema del diseño de un TAD:



3.1 Especificación.

× Sintáctica: Se definen los símbolos de los operadores, nombre de los procedimientos y funciones, tipos y numero de los operandos utilizados, y el tipo y numero de las salidas de estas operaciones.

× Semánticas: Para cada una de la posibles entradas a nuestras operaciones diremos la salida o salida de errores.

3.2 Implementación.

Los algoritmos unido a la representación da como resultado una posible implementación del TAD.

1ª Especificaciones en lenguaje natural.

Se describen en lenguaje natural el nombre de la operaciones que vamos a usar, así como las excepciones que tienen esas operaciones.Al especificar la abstracción solo se deben describir aquellos aspectos que sean relevantes para la solución del problema.

Es esencial utilizar un lenguaje que sea preciso y de fácil comprensión, de forma que no quepan ambigüedades entorno al comportamiento deseado.La especificación se puede realizar mediante un esquema similar a lo que será la cabecera del programa al codificarlo.Dicho esquema deberá contener la forma de la llamada, con el nombre de la función, sus argumentos y resultados, así como una información de si modifica alguno de esos argumentos o bien accede a variables globales.

Se debe incluir una descripción general de lo que hace la función, pero sin entrar en el detalle de como lo hace.

```
OPERACION nombre (argumentos) DEVUELVE (resultado)
    REQUISITOS:
    MODIFICA:
    EFECTOS:
```

2º Especificaciones operacionales.

Describe el TAD mediante un lenguaje de alto nivel, sin tener en cuenta la eficiencia.

3º Especificaciones funcionales.

Se describe o implementa una función matemática que define una relación entre la entrada y la salida.

4º Especificaciones lógicas.

Se realizan mediante asertos de entrada y salida o invariantes. Estos van a describir las condiciones iniciales y finales de ejecución de nuestro TAD.

5º Especificaciones Algebraicas.

Describen al TAD como un algebra, y para implementarlo vamos a tener que desarrollar los axiomas que definen este algebra.Con estos axiomas va a ser posible un desarrollo matemático de las excepciones.Vamos a tener dos partes: sintáctica (o definición de los operadores) y semántica (axiomas).

El conjunto de operaciones definidas sobre el tipo abstracto de datos debe ser cerrado, es decir, solo se debe acceder a los valores de los datos empleando las operaciones abstractas definidas sobre ellos.La abstracción de datos es como una caja que encierra los datos y solo permiten acceder a ellos de manera controlada.

La abstracción de datos debe realizar una importante labor de ocultamiento, haciendo que desde el exterior solo sea visible el efecto global de las operaciones, pero no el detalle de como se realizan ni como se representan internamente los valores de los datos.

3.3 Propiedades de los TAD.

× Instanciabilidad: Vamos a poder declarar variables del tipo TAD, y estas van a tener las mismas propiedades y características que el TAD.

× Privacidad: No vamos a conocer detalladamente las operaciones ni la estructura que utiliza ese TAD.

× Componibilidad jerárquica: Un TAD puede estar formado por otros TAD mas básicos.

× Enriquecimiento: Podemos añadir operaciones al TAD.

Ejemplo 1

Especificaciones algebraicas semánticas de un TAD pila con los siguientes operaciones:

- CreaPila () ---> Pila
- Introducir (Pila,Elem) ---> Pila
- Decapitar (Pila) ---> Pila
- Cima (Pila) ---> Elem
- Vacia (Pila) ---> BOOLEAN

Para toda P de tipo Pila y E de tipo Elemento tenemos:

× Vacia (CreaPila(P)) ---> TRUE
× Vacia (Introducir(P,E)) ---> FALSE
× Decapitar (CrearPila(P)) ---> ERROR
× Decapitar (Introducir(P,E)) --->P
× Cima (CreaPila(P)) ---> ERROR
× Cima (Introducir(E)) --->E

Ejemplo 2

Editor de ficheros secuenciales con cinco operaciones: Crear un nuevo fichero, insertar, reemplazar, eliminar, avanzar y retroceder, actuando siempre sobre el registro actual.

Para las especificaciones en lenguaje natural hay que describir el nombre de las operaciones que vamos a realizar, junto con las excepciones de dichas operaciones.

Operaciones:

× **Fichero_Nuevo (Fichero)**: Crea un fichero nuevo sin introducir ningún registro en el (número de registros es 0).

× **Insertar (Fichero,Registro)**: Inserta un registro después del registro actual y el nuevo pasa a ser el actual.

× **Reemplazar (Fichero,Registro)**: Cambia el registro actual por el nuevo.

× **Eliminar (Fichero)**: Borra el registro actual y una vez eliminado el actual pasa a ser el siguiente.

× **Avanzar (Fichero)**: El registro siguiente pasa a ser el actual.

× **Retroceder (Fichero)**: El registro anterior pasa a ser el actual.

Excepciones:

	CONDICION	ACCION
Insertar	Fichero vacio	Inserta en la primera posición
Reemplazar	Fichero vacio	Nada
Eliminar	Fichero vacio	Nada
	Actual es el último	Borra y actual será el anterior
Avanzar	Fichero vacio	Nada
	Actual es el último	Nada
Retroceder	Fichero vacio	Nada
	Actual es el primero	No hay registro actual

La descripción del TAD mediante un lenguaje de alto nivel no tiene porque tener en cuenta la eficiencia.

Definición de tipos:

```
Fichero=ARRAY [1..n] OF INTEGER;
 Longitud:INTEGER;
 Reg_Actual:INTEGER;
```

Implementación de las operaciones:

```
Fichero_Nuevo (Fichero);
BEGIN
   Longitud:=0;
   Reg_Actual:=0;
END;


Insertar (Fichero,RegNuevo);
BEGIN
   for j:=Longitud doWNTO RegActual do
   Fichero [j+1]:=Fichero[j];
   Fichero[RegActual+1]:=RegNuevo;
   Longitud:=Longitud+1;
   RegActual:=RegActual+1
END;


Retroceder (Fichero);
BEGIN
   IF RegActual<>0 THEN RegActual:=RegActual-1
END;


Eliminar (Fichero);
BEGIN
   IF RegActual<>0
         THEN for j:=Actual TO Longitud-1 do
                     Fichero[j]:=Fichero[j+1];
   IF RegActual>Longitud THEN RegActual:=RegActual-1
END;
```

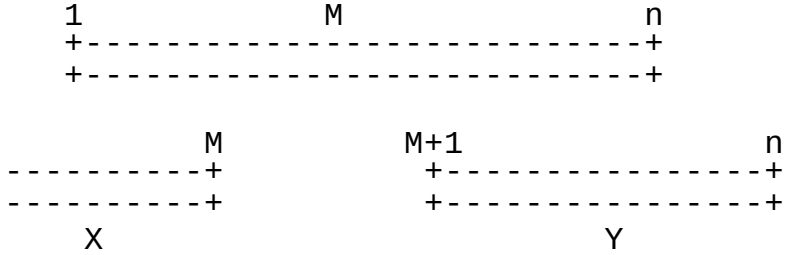
```
Avanzar (Fichero);
BEGIN
   IF RegActual<>Longitud
         THEN RegActual:=RegActual+1
END;


Reemplazar (Fichero,RegNuevo);
BEGIN
   IF RegActual<>0
        THEN Fichero[RegActual]:=RegNuevo
END;
```

Especificaciones algebraicas sintácticas:

```
Fichero_Nuevo () ---> Fichero
Insertar (Fichero) ---> Fichero
Reemplazar (Fichero,Registro) ---> Fichero
Eliminar (Fichero) ---> Fichero
Avanzar (Fichero) ---> Fichero
Retroceder (fichero) ---> Fichero
```

Especificaciones algebraicas semánticas:

Vamos a definir la operación M(x,y) que une las dos partes, x e y y deja el fichero como estaba:



- Fichero_Nuevo() = M(Fichero_Nuevo,Fichero_Nuevo)

- Insertar(r,M(x,y)) = M(Insertar(r,x),y)
  Se inserta en la parte X porque es la que contendría el registro actual.

- Reemplazar(s,M(Insertar(r,x),y)) = M(Insertar(s,x),y)

- Reemplazar(r,M(Fichero_Nuevo,y)) = M(Fichero_Nuevo,y)
  Tener la parte X sin elemento es como tener el fichero vacio, porque el registro actual está en la parte X.

- Eliminar(M(Insertar(r,x),y)) = M(x,y)
  Si insertas y eliminas no haces nada.

- Eliminar(M(Fichero_Nuevo,y)) = M(Fichero_Nuevo,y)

- Avanzar(M,(x,Insertar(r,y))) = M(Insertar(r,x),y)
- Avanzar(M,(x,Fichero_Nuevo)) = M(x,Fichero_Nuevo)

- Retroceder(M(Insertar(r,x),y)) = M(x,Insertar(r,y))

- Retroceder(M(Fichero_Nuevo,y)) = M(Fichero_Nuevo,y)

Ejercicio 1.

Se dispone de un compilador que únicamente tiene implementada la estructura de datos pila, y las siguientes operaciones asociadas.

- PROCEDURE Inicializar (VAR Pila:Tpila);
- PROCEDURE Apilar (VAR Pila:Tpila,elem:Telem);
- PROCEDURE Decapitar (VAR Pila:Tpila);
- PROCEDURE Cima (Pila:Tpila, VAR Elemento:Telem);
- FUNCTION Vacia (Pila:Tpila):BOOLEAN;

Se pide implementar un procedimiento iterativo (no recursivo) que invierta una pila utilizando las operaciones descritas anteriormente.Se podrán utilizar aquellas estructuras auxiliares que tengan implementadas el compilador.

Invertiremos el contenido de la pila utilizando una pila auxiliar.

```
PROCEDURE intercambiar (VAR p1,p2:Tpila);
VAR Elem:Telem;
BEGIN
  WHILE NOT Vacia(p1) do
  BEGIN
     Cima (p1,Elem);
     Apilar (p2,Elem);
     Decapitar (p1)
  END
END

BEGIN (* Programa principal *)
   Inicializar (p1);
   Intercambiar (p0,p1);
   Inicializar (p2);
   Intercambiar (p1,p2);
   Intercambiar (p2,p0)
END.
```

Ejercicio 2.

Se pretende construir un TAD pila en el que se van a introducir numeros enteros positivos de un dígito.Para ello se dispone del TAD conjunto, de enteros de dos dígitos en el que se han definido las siguientes operaciones:

```
Tdigitos=0..99;
Tconjunto=(!Desconocido!);
```

- P.Inicializar (VAR Conjunto:Tconjunto);
- P.Introducir (VAR Conjunto:Tconjunto; Digito:Tdigito);
- P.Eliminar (VAR Conjunto:Tconjunto; Digito:Tdigito);
- F.Pertenece (Conjunto:Tconjunto; Digito:Tdigito):BOOLEAN;

Se pide implementar la operación introducir un elemento en la pila usando el TAD conjunto.

Vamos a hacer que la unidad sea el elemento y la decena el orden del elemento, por tanto: 0..9 sería el primer elemento,10..19 el segundo, ... hasta llegar a 90..99 que sería el décimo.

```
TYPE
   Tpila=RECORD
   Conjunto:Tconjunto;
   Nele:INTEGER
END;

PROCEDURE Apilar (VAR Pila:Tpila;Ele:INTEGER);
BEGIN
   IF Pila.Nele>=10
       THEN < Pila llena >
       ELSE BEGIN
               Introducir (Pila.Conjunto, Pila.Nele*10+Ele);
               Pila.Nele:=Pila.Nele+1
            END
END;
```

Ejercicio 3.

Dado el TAD pila de enteros con las operaciones del ejercicio 1, se pide implementar un procedimiento que, usando las operaciones del TAD descrito, copie el contenido de una pila en otra.Tendrá la cabecera:

```
PROCEDURE Copiar (Origen:Tpila ; VAR Destino:Tpila);
```

1. DEFINICION.

Hablamos de recursividad, tanto en el ámbito informático como en el ámbito matemático, cuando definimos algo (un tipo de objetos, una propiedad o una operación) en función de si mismo.La recursividad en programación es una herramienta sencilla, muy útil y potente.
Ejemplo:

× La potenciación con exponentes enteros se puede definir:

$$a^{/,0} = 1$$
$$a^{n} = a*a^{(n-1)} \qquad \text{si } n>/,0$$

× El factorial de un entero positivo suele definirse:

$$/,0! = 1$$
$$n! = n*(n-1)! \qquad \text{si } n>/,0$$

En programación la recursividad supone la posibilidad de permitir a un subprograma llamadas a si mismo, aunque también supone la posibilidad de definir estructuras de datos recursivas.

2. TIPOS.

Podemos distinguir dos tipos de recursividad:

× **Directa**: Cuando un subprograma se llama a si mismo una o mas veces directamente.

× **Indirecta**: Cuando se definen una serie de subprogramas usándose unos a otros.La recursividad indirecta se consigue en Pascal, a pesar de la condición general de que todo subprograma ha de definirse antes de ser usado, con una predeclaración de uno de los subprogramas (la cabecera completa) seguida de la directiva FORWARD. Al declarar después ese subprograma solo es necesario escribir el nombre, no los parámetros.

El lenguaje PASCAL permite el uso de la recursividad.En cambio,hay una serie de lenguajes que no la permiten, como el FORTRAN, Basic, etc, y otros que la consideran una estructura de control principal, como el LISP y todos los lenguajes funcionales.

3. CARACTERISTICAS.

Un algoritmo recursivo consta de una parte recursiva, otra iterativa o no recursiva y una condición de terminación.La parte recursiva y la condición de terminación siempre existen.En cambio la parte no recursiva puede coincidir con la condición de terminación.

Algo muy importante a tener en cuenta cuando usemos la recursividad es que es necesario asegurarnos de que llega un momento en que no hacemos más llamadas recursivas.Si no se cumple esta condición el programa no parará nunca.

Para asegurarnos de su terminación podemos tener, por ejemplo, un parámetro o una función de los parámetros que disminuya en cada llamada y, además, la llamada recursiva debe estar condicionada para que no se ejecute siempre (con una instrucción IF, WHILE, CASE, etc).A esta pregunta es a lo que llamamos condición de terminación.

```
FUNCTION Fact (n:INTEGER):INTEGER;
BEGIN
   if n<=1
        THEN Fact:=1
        ELSE Fact:=n*Fact(n-1)
END;
```

Para saber si un programa recursivo funciona bien, debemos comprobar lo siguiente:

× Que existe al menos una condición de terminación para la cual la evaluación es no recursiva, y que funciona correctamente en el caso o casos no recursivos, si los hay.

× Cada llamada se realiza con un dato mas pequeño, en el sentido de conseguir que se acabe cumpliendo la condición de terminación.

× Suponiendo que las llamadas recursivas funcionan bien, que el subprograma completo funciona bien.

4. VENTAJAS E INCONVENIENTES.

La principal ventaja es la simplicidad de comprensión y su gran potencia, favoreciendo la resolución de problemas de manera natural, sencilla y elegante; y facilidad para comprobar y convencerse de que la solución del problema es correcta.

El principal inconveniente es la ineficiencia tanto en tiempo como en memoria, dado que para permitir su uso es necesario transformar el programa recursivo en otro iterativo, que utiliza bucles y pilas para almacenar las variables.

5. USO DE LA RECURSIVIDAD.

× En estructuras de datos definidas recursivamente.
× Cuando no haya una solución iterativa clara y simple.
× Cuando el algoritmo sea claramente recursivo.

EJEMPLO 1

Permutaciones, variaciones sin repetición, se dan todos los números y todos ellos repetidos.El siguiente es el algoritmo iterativo para cuatro numeros:

```
a:ARRAY[1..4] OF 1..4;

FOR a[1]:=1 TO 4 DO
   FOR a[2]:=1 TO 4 DO
      FOR a[3]:=1 TO 4 DO
         FOR a[4]:=1 TO 4 DO
            IF (a[1]<>a[2]) AND (a[2]<>a[4]) AND (a[1]<>a[3])
                  AND (a[1]<>a[4]) AND (a[2]<>a[3])
                  AND (a[2]<>a[3]) AND (a[3]<>a[4])
                       THEN escribir numero
```

El orden de este algoritmo es 4^4, luego si lo tuviésemos que implementar para n elementos el orden seria n^n.Hay que realizar una reducción de bucles FOR anidados.

```
FOR a[1]:=1 TO 4 DO
   FOR a[2]:=1 TO 4 DO
      IF a[1]<>a[2]
          THEN FOR a[3]:=1 TO 4 DO
                  IF (a[1]<>a[3]) AND (a[2]<>a[3])
                       THEN FOR a[4]:=1 TO 4 DO
                               IF (a[1]<>a[4]) AND (a[2]<>a[4])
                                    AND (a[3]<>a[4]) THEN valida
```

Si lo queremos para n numeros necesitamos n FOR, por tanto cada número necesitará una implementación diferente, cada una con un número diferente de FOR.La mejor solución es un algoritmo recursivo:

- × Generamos el primer número y permutamos (n-1).
- × Generamos el segundo número y permutamos (n-2) si es válida.
- × Generamos el tercer número y permutamos (n-3) si es válida.

```
PROCEDURE Permutar (n,Tam:INTEGER);
BEGIN
   if n<=Tam
      THEN FOR a[n]:=1 TO Tam DO
              IF Valida(n)
                  THEN Permutar(n+1,Tam)
                  ELSE escribir
END;
```

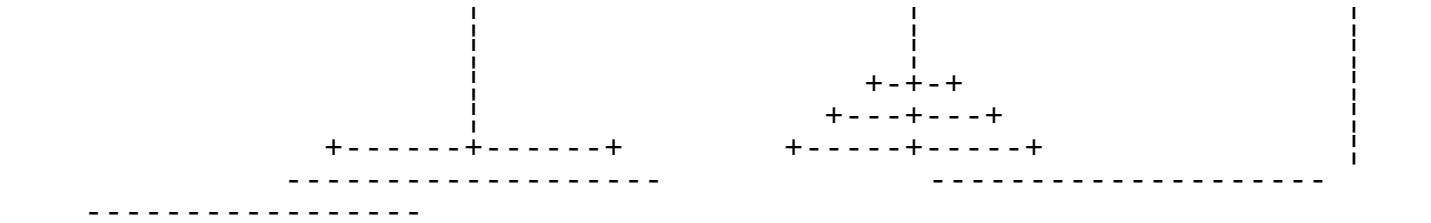
```
FUNCTION Valida (n:INTEGER):INTEGER;
VAR i:INTEGER;
BEGIN
   Valida:=TRUE;
   FOR i:=1 TO n-1 DO
          IF a[i]=a[n] THEN Valida:=FALSE
END;
```

6. LAS TORRES DE HANOI

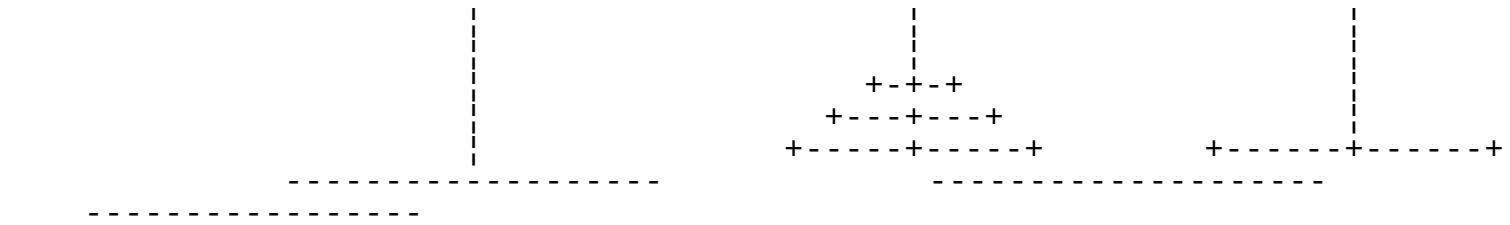
Tenemos tres torres y **n** discos en la primera de ellas, que vamos a llamar origen, y los tenemos que llevar a la torre destino teniendo en cuenta que:

- × Solo podemos mover un disco a la vez.
- × No podemos poner un disco encima de otro menor.

Para mover el primer disco a destino hay que mover los **n-1** primeros de origen a trabajo, y movemos el que nos queda a destino.



De la situación anterior se pasa a la siguiente:



Y ahora tenemos el mismo problema solo que hay que mover n-1 discos usando origen como auxiliar.

```
PROCEDURE Hanoi (n,Origen,Trabajo,Destino:INTEGER);
BEGIN
   if n=1
      THEN Mover (Origen,Destino)
      ELSE BEGIN
              Hanoi(n-1,Origen,Destino,Trabajo);
              Mover(Origen,Destino);
              Hanoi(n-1,Trabajo,Origen,Destino)
           end
end;
```

Seguimiento para n=3

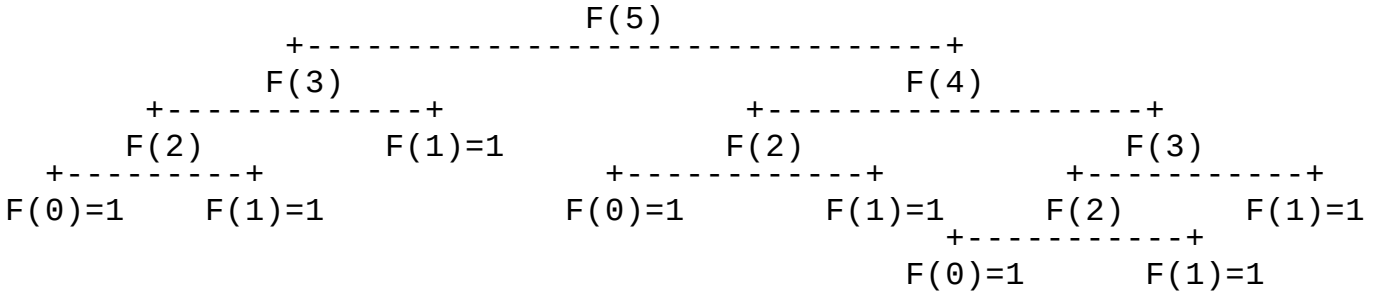
```
    * Primera llamada n=3 , o=1 , t=2 , d=3, llama H(2,1,3,2)
    * Segunda llamada n=2 , o=1 , t=3 , d=2, llama H(1,1,2,3)
    * Tercera llamada n=1 , o=1 , t=2 , d=3,

         Mueve disco de origen a destino M(1,3)
       ** Vuelve a n=2 (segunda llamada)
          Mueve disco M(1,2)
          Hace otra llamada H(1,3,1,2) y como n=1, Mueve disco M(3,2)

       ** Vuelve a N=3 (primera llamada)
          Mueve disco M(1,3)
          Hace otra llamada H(2,2,1,3) o=2 , t=1 , d=3

           *** Llama H(1,2,3,1) o=2 , t=3 , d=1 y como n=1 , M(2,1)
           *** Vuelve y ejecuta M(2,3)
           *** Ultima llamada H(1,1,2,3) y como n=1 ejecuta M(1,3)
```

7. FUNCION DE FIBONACCHI

$$Fibo(n) = \begin{cases} 1 & \text{si } n=1 \text{ o } n=0 \\ Fibo(n-1) + Fibo(n-2) & \text{para el resto} \end{cases}$$

```
FUNCTION Fibo(n:INTEGER):INTEGER;
BEGIN
   if n<=1
       THEN Fibo:=1
       ELSE Fibo:=Fibo(n-1)+Fibo(n-2)
END;
```

Si hacemos su seguimiento para n=5 vemos que esta implementación resulta muy ineficiente al realizar dos llamadas.



Si vamos guardando el valor de F(n-2) y pasándolo por variable se evita este rápido crecimiento.

```
FUNCTION Fibo (n:INTEGER;VAR fn_2:INTEGER):INTEGER;
VAR fn_1:INTEGER;
BEGIN
   IF n<=1
      THEN BEGIN
              Fibo:=1;
              fn_2:=1;
           END
      ELSE BEGIN
              fn_1:=Fibo(n-1,fn_2);
              Fibo:=fn_1+fn_2;
              fn_2:=fn_1;
           END
END;
```

Vamos a ver ahora un seguimiento de esta nueva implementación para n=5.

```
n=5  Fibo(5,variable)
     fn_1:= Fibo(4,...) = 5
     Fibo:= 5+3 = 8
     fn_2:= 5

n=4  fn_1:= Fibo(3,...) = 3
     Fibo:= 3+2 = 5
     fn_2:= 3

n=3  fn_1:= Fibo(2,..) = 2
     Fibo:= 2+1 = 3
     fn_2:= 2
n=2  fn_1:= Fibo(1,..) = 1
     Fibo:= 1+1 = 2
     fn_2:= 1

n=1  fn_1:= Fibo(1) = 1
     Fibo:= 1
     jn_2:= 1
```

8. FUNCION DE ACKERMANN

```
          +
          |      N+1          si M=0
          |
A(M,N)=   |    A(M-1,1)       si N=0
          |
          | A(M-1,A(M,N-1))   para el resto
          +
```

Una posible implementación es la siguiente:

```
FUNCTION Ack (m,n:INTEGER):INTEGER;
BEGIN
   IF m=0
      THEN Ack:=n+1
      ELSE IF n=0
             THEN Ack:=Ack(m-1,1)
             ELSE Ack:=Ack(m-1,Ack(m,n-1))
END;
```

Si realizamos un seguimiento para A(1,4) podemos observar que la complejidad crece exponencialmente según crecen los parámetros.

```
A(1,4)  m=1 n=4  ---->  A = A(0 , A(1,3)) = 6
A(1,3)  m=1 n=3  ---->  A = A(0 , A(1,2)) = A(0,4) = 5
A(1,2)  m=1 n=2  ---->  A = A(0 , A(1,1)) = A(0,3) = 4
A(1,1)  m=1 n=1  ---->  A = A(0 , A(1,0)) = A(0,2) = 3
A(1,0)  m=1 n=0  ---->  A = A(0,1) = 2
A(0,1) = 2
```

EJERCICIOS

Ejercicio 1.

Realizar un seguimiento de estos subprogramas recursivos.

a)

```
PROCEDURE p1 (a:INTEGER);
BEGIN
   IF a>0
       THEN BEGIN
               WRITELN;
               p1(a-1)
            END
       ELSE WRITELN('FIN')
END;
```

Salida: a, a-1, a-2, ..., 1, FIN

Ejercicio 2.

Escribir una función recursiva que calcule la suma de todos los elementos contenidos en un ARRAY de enteros.No se puede utilizar ningún bucle.

```
FUNCTION Suma (VAR a:Ttabla, n:INTEGER):INTEGER;
BEGIN
   IF n>0
      THEN Suma:=a[n]+Suma(a,n-1)
      ELSE Suma:=0
END;
```

Ejercicio 3.

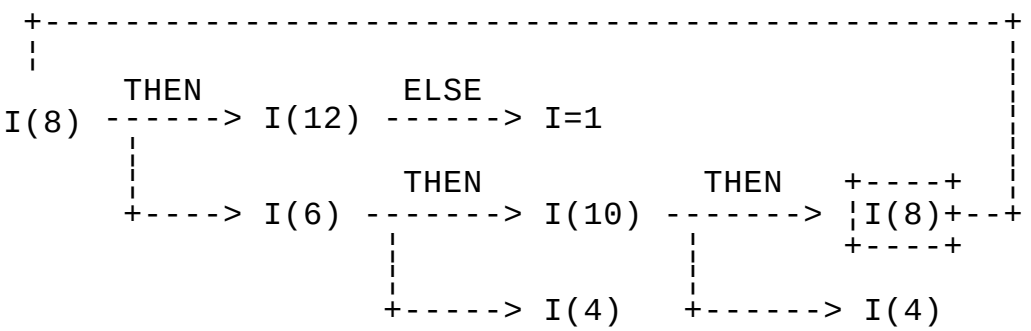
Escribir una función que diga si una palábra es o no capicúa (la palabra esta en una tabla)

```
FUNCTION Capicua (Palabra:Tpalabra;Ini,Fin:INTEGER):BOOLEAN;
BEGIN
  IF Ini>=Fin THEN
    Capicua:=TRUE
  ELSE
    Capicua:=(Palabra[Ini]=Palabra[Fin]) AND (Capicua(Ini+1,Fin-1)
END;
```

Ejercicio 4.

Hacer un seguimiento y obtener el resultado para **Incognita(8)**.

```
FUNCTION Incognita (Num:INTEGER):INTEGER;
BEGIN
   IF (Num>0) AND (Num<11)
         THEN Incognita:=Incognita(Num-1)+Incognita(Num+4)
         ELSE IF Num<=0
                 THEN Incognita:=0
                 ELSE Incognita:=1
END;
```



Se produce un bucle infinito, el programa no acaba nunca.

Ejercicio 5

Realizar un seguimiento de la siguiente función, para los parámetros:

a) x:=Concurso (0,3)
b) y:=Concurso(10,-7)
c) z:=Concurso(5,5)

```
FUNCTION Concurso (Base,Lim:INTEGER):INTEGER;
BEGIN
   IF Base=Lim
         THEN Concurso:=1
         ELSE IF Base>Lim
                 THEN Concurso:=0
                 ELSE Concurso:=Base+Concurso(Base+1,Lim)
END;
```

La salida es: **x=4**, **y=0** y **z=1**

B=0,L=3 ----> Con=0+Con(1,3)=4 ; B=1,L=3 ----> Con(2,3)=3

B=2,L=3 ----> Con=2+Con(3,3)=1 ; B=3,L=3 ----> Con=1

Ejercicio 6

Simular el bucle FOR mediante procesos recursivos.

```
PROCEDURE Bucle (Inf,Sup:INTEGER);
BEGIN
   IF Inf<=Sup
        THEN BEGIN
                Proceso(...);
                Bucle(Inf+1,Sup)
             END
END;
```

Ejercicio 7.

Implementar una función recursiva que dados dos números, dé como resultado la multiplicación del primero por el segundo.Solo se puede utilizar la función SUCC de pascal (entero) y no se pueden usar bucles.

```
a*b = a+a+a+...+a          b veces.
a+b = a+1,+1,+1,... ----> Hacer b veces SUCC de a.
```

```
FUNCTION Suma (a,b:INTEGER):INTEGER;
BEGIN
   IF b>m
       THEN Suma:=Suma(SUCC(a),b,SUCC(m))
       ELSE Suma:=a
END;

FUNCTION Producto (c,d,m:INTEGER):INTEGER;
BEGIN
   IF c=0 OR d=0
         THEN Producto:=0
         ELSE IF d>m
                  THEN Producto:=Suma (Producto(c,d,SUCC(m)),c,0)
                  ELSE Producto:=0
END;
```

Ejercicio 8.

Dado el siguiente procedimiento recursivo:

a) Calcular el numero de llamadas que se harán al procedimiento recursivo en función de n.

b) Si en el punto * p * la pila interna del sistema tiene capacidad para almacenar un máximo de 32405 bytes y suponiendo que:

- La dirección de retorno de un procedimiento ocupa 4 bytes.
- Una variable entera ocupa 2 bytes.

Calcular el valor máximo de n para el que se pueda ejecutar la llamada P(n) sin que se desborde la pila.Razonar las respuestas.

```
PROCEDURE p(n:INTEGER);

   PROCEDURE Recursivo (m,n:INTEGER);
   VAR i:INTEGER;
   BEGIN
      IF m>1
          THEN IF n>1
                   THEN Recursivo(m,n-1)
                   ELSE Recursivo (m-1,m-1);
      i:=m*n;
      WRITELN(i);
   END;

BEGIN
   IF n>=1
       THEN BEGIN (** p **)
               Recursivo(n,n);
               WRITELN('fin')
            END
END;
```

Cada llamada ocupa 10 bytes de la pila, la llamada 8 ,2 de cada variable y 2 también de cada variable local

a)
- m=n=1 se acaba.
- m>1 n>1 realiza (m,n-1)
- m>1 n=1 realiza (m-1,m-1)

si se hace la llamada con n tenemos:

```
   (n,n)      +--->  (n-1,n-1)    +--->   (n-2,n-2)
   (n,n-1)    |      (n-1,n-2)    |       (n-2,n-3)
   (n,n-2)    |      (n-1,n-2)    |          ...
     ...      |        ...        |       Hasta que los
   (n,1)   ---+      (n-1,1)   ---+        dos sean 1
```

lo que nos dá la siguiente sucesión de llamadas:

$$n+(n-1)+(n-2)+(n-3)+\ldots+1=\frac{(1+n)n}{2}$$

es decir,

```
                                                          +---------
----------------------------------------------------------------+
¦ EL PRIMERO MAS EL ULTIMO, POR EL NUMERO DE TERMINOS, PARTIDO POR DOS
¦
+-----------------------------------------------------------------
+
```

b)

$$\frac{n+n^2}{2}\_10\geq 32405 \qquad n\geq 80$$

Ejercicio 9.

Invertir una pila sin ninguna estructura auxiliar, y con las operaciones: cima, decapitar, vacía, introducir.

```
PROCEDURE Invertir (VAR p:Tpila);
VAR a:Telemento;
```

```
BEGIN
   IF NOT Vacia(p)
        THEN BEGIN
                Cina(p,a);
                Decapitar(p);
                Invertir(p);
                InsertaFinal(p,a)
             END
END;

PROCEDURE InsertaFinal (VAR p:Tpila; a:Tele);
VAR Ele:Tele;
BEGIN
   IF NOT Vacia(p)
         THEN BEGIN
                 Cima(p,Ele);
                 Decapitar(p);
                 InsertaFinal(p,a);
                 Introducir(p,Ele)
              END
         ELSE Introducir (p,a)
END;
```

Ejercicio 10.Practica 3.

Dada una secuencia de numeros positivos cualquiera, dispuestos sin ningún tipo de orden definido.Una subserie no decreciente de la misma es una secuencia de numeros extraídos de la secuencia original, manteniendo una posición relativa entre los mismos numeros idéntica a la que tenían originalmente, y de manera que cada numero es menor o igual al siguiente:

Ejemplo:

8,3,2,3,5,6,10,5,5,6,10,1,3

la secuencia:

3,3,5,6,10,10

es una subserie no decreciente de la misma con longitud 6, sin embargo:

1,8,10

no lo es.

Se pide desarrollar un algoritmo recursivo que calcule la longitud de la subserie no decreciente mas larga de una secuencia dada.

```
FUNCTION LongSerie (Maximo:INTEGER;Posicion:Indice):INTEGER;
BEGIN
   IF Posicion>=LongitudTabla
       THEN IF Tabla[Posicion]>=Maximo
                  THEN LongSerie:=1
                  ELSE LongSerie:=0
       ELSE IF Tabla[Posicion]>=Maximo
                  THEN LongSerie:=Mayor(LongSerie(Maximo,Posicion+1),
                                 LongSerie(Tabla[Posición],Posición+1)+1)
                  ELSE LongSerie:=LongSerie(Maximo,Posición+1)
END;
```

EJERCICIOS DE RECURSIVIDAD

Descríbase el efecto de los siguientes procedimientos y funciones:

1)
```
PROCEDURE p2 (a,b:INTEGER);
BEGIN
   IF a MOD b <>0
         THEN BEGIN
                 WRITELN(a);
                 p2(a+1,b)
              END
END;
```

Suponiendo que la llamada al procedimiento p2 se hace siempre de la siguiente forma:

```
                    if a>b
                       THEN p2 (abs(a),abs(b));
```

2)
```
PROCEDURE p3(a,b:INTEGER);
BEGIN
   IF a>0
      THEN p3 (a-1,b+a)
      ELSE WRITELN(b);
END;
```

Suponiendo que la llamada al procedimiento p3 se hace de la forma:

p3(a,0) siendo a un valor entero cualquiera

3)
```
PROCEDURE p8 (VAR a:INTEGER;b:INTEGER);
VAR c:REAL;
BEGIN
   c:=SQRT(b);
   IF c=TRUNC(c)
         THEN a:=b
         ELSE p8(a,b-1);
END;
```

4)
```
PROCEDURE p4 (a:INTEGER);
BEGIN
   if a>0
       THEN BEGIN
               p4(a-1);
               WRITELN(a);
            end
       ELSE WRITELN('fin');
END;
```

5)
```
PROCEDURE p5(a,b:INTEGER;c:REAL);
BEGIN
   IF ABS(a/b-c)<0.001
         THEN WRITELN(a,'/',b)
         ELSE IF a/b<c
                  THEN p5(a+1,b,c)
                  ELSE p5(a,b+1,c)
END;
```

Suponiendo que la llamada a p5 se hace siempre de la forma:

```
                        p5(1,1,c);
```

siendo c un valor real mayor o igual que cero

6)
```
PROCEDURE p6(VAR a:INTEGER);
BEGIN
   WRITELN(a);
   IF a>0
       THEN a:=a-1
       ELSE IF A<0
                THEN A:=A+1;
                IF A<>0
                    THEN P6(A);
END;
```

En este ejercicio, ¿Qué ocurria si el parametro **a** no estuviera pasado por VAR.

7)
```
PROCEDURE P7 (VAR a:INTEGER);
BEGIN
   WRITELN(a);
   IF a>0
       THEN BEGIN
               a:=a-1;
               p7(a)
            END;
   WRITELN(a)
END;
```

¿Qué ocurria en este ejercicio si el parametro **a** no estuviera pasado por VAR?

8)
```
PROCEDURE p9 (a:INTEGER);
VAR i:INTEGER;
BEGIN
   WRITELN(a);
   FOR i:=a DOWNTO 1 DO
               p9(i-1)
END;
```

9)
```
PROCEDURE p10(a:INTEGER;c:CHAR);
VAR i:INTEGER;
    b:BOOLEAN;
BEGIN
   b:=FALSE;
   FOR i:=1 TO a-1 DO
   BEGIN
      WRITE(' ');
      b:=TRUE
   END;
   WRITELN(c);
   IF b
      THEN p10(a-1,c)
END;
```

10)
```
PROCEDURE p11 (VAR a:INTEGER;b:INTEGER);
BEGIN
   IF b>0
       THEN BEGIN
               a:=a+1;
               p11(a,b-1);
            END
END;
```

11)
```
FUNCTION f1(a:INTEGER):INTEGER;
BEGIN
   IF a>0
      THEN f1:=f1(a-1)+1
      ELSE f1:=0
END;
```

12)
```
FUNCTION f2(a:INTEGER):INTEGER;
BEGIN
   IF a>0
       THEN f2:=f2(a-1)+a
       ELSE f2:=0
END;
```

13)
```
FUNCTION f3(a,b:INTEGER):INTEGER;
BEGIN
   IF a>b
       THEN f3:=a+f3(a-1,b)
       ELSE IF b>a
                THEN f3:=b+f3(a,b-1)
                ELSE f3:=0
END;
```

14)
```
FUNCTION f4(a:INTEGER):INTEGER;
BEGIN
   IF a>2
       THEN f4:=(f4(a-1)+f4(a-2))*a
       ELSE f4:=a
END;
```

15)
```
FUNCTION f5 (a:INTEGER):INTEGER;
VAR r,i:INTEGER;
BEGIN
   IF a>0
       THEN BEGIN
               r:=a;
               FOR i:=a-1 DOWNTO 1 DO
                        r:=r+f5(i);
               f5:=r
            END
       ELSE f5:=a
END;
```

16)
```
FUNCTION f6(a,b:INTEGER):INTEGER;
VAR r,i:INTEGER;
BEGIN
   IF b>0
       THEN BEGIN
               r:=b;
               FOR i:=a DOWNTO 1 DO
                       r:=r+f6(i,b-1);
               f6:=r
            END
       ELSE f6:=b
END;
```

17) Calcular la expresión de la complejidad de los algoritmos de los ejercicios anteriores.Así mismo, identificar la condición de parada en todos estos ejercicios y, en caso de que así sea, explicar de que manera esta cada llamada recursiva "más cerca" de la condición de parada.

18) Examen Junio 1988.

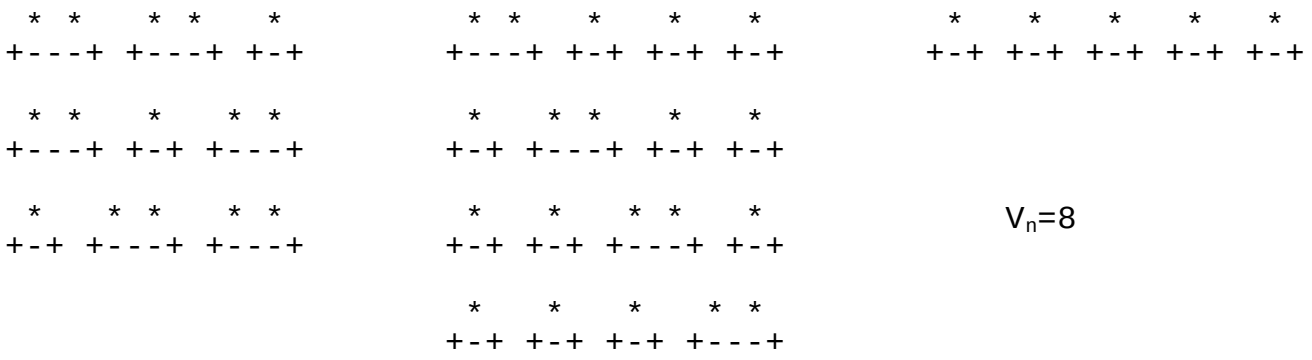
Diseñar un procedimiento que, dado un numero entero n que recibe como parametro, tenga una complejidad O(n!).No se podrá utilizar ningún operador aritmético, salvo + o -.

19) Examen Junio 1988

Sea V_n el numero de maneras posibles de dividir una fila de n canicas en grupos de 1 o 2 canicas, sin cambiar su orden.

Ejemplos:

1.- n=5 (5 canicas)



2.- Para n=6,Vn=13

Se pide escribir una función en PASCAL que calcule el valor de V_n para cualquier valor de n.Téngase en cuenta que se pide calcular cuantas maneras hay que dividir la fila, y no cuales son estas maneras.(Supóngase siempre n mayor que 0).La función debe ser recursiva.

20) Examen Junio 1989

Dado el procedimiento:

```
PROCEDURE p(n,x:INTEGER;VAR y:INTEGER);
BEGIN
   IF n>=x
        THEN BEGIN
                y:=y+1;
                p(n,2*x,y)
             END
END;
```

Deseamos ejecutarlo con un compilador que no dispone de estructuras de control iterativas (WHILE, FOR ni REPEAT), tampoco dispone de la sentencia GOTO ni admite recursividad.

Codificar, razonándolo, otra versión del procedimiento anterior que produzca exactamente el mismo efecto ateniendose a las limitaciones anteriormente descritas, sabiendo que la llamada inicial va a ser siempre:

```
y:=0;
p(n,2,y)        siendo n>0
```

1. ESTRUCTURA REPRESENTACION

Una tabla es una estructura homogénea en la que todos los elementos que la componen son del mismo tipo.Son estáticas, no crecen ni decrecen en tiempo de ejecución y tienen un límite preestablecido antes de la compilación.

Para acceder a los elementos de una tabla se utilizan los *"índices"* y estos pueden ser de cualquier tipo escalar de PASCAL (enumerados, INTEGER, CHAR, subrango, BOOLEAN).Por ello las tablas son estructuras de acceso directo o acceso por índice.

Si los elementos de la tabla ocupan un número no entero de palabras de memoria, se reserva para cada elemento el entero inmediatamente superior.Cuando ocupa menos de una palabra se puede empaquetar declarándolo PACKED ARRAY.Hay dos procedimientos pascal relacionados con este tema:

- PACK: Empaqueta un ARRAY normal.
- UNPACK: Desempaqueta un ARRAY empaquetado.

1.1 Búsqueda secuencial.

```
TYPE
   Ttabla=ARRAY[1..n] OF Ele;      (* Ele será un registro con clave *)

PROCEDURE Busqueda (Tabla:Ttabla;VAR Componente:Tele;Clave:Tclave;
                                 VAR Encontrado:BOOLEAN);
VAR i:INTEGER;
BEGIN
   i:=0;
   REPEAT
      i:=i+1
   UNTIL (Tabla[i].Clave=Clave) OR (i=n);
   IF Tabla[i].Clave=Clave
      THEN BEGIN
              Encontrado:=TRUE;
              Componente:=Tabla[i]
           END
      ELSE Encontrado:=FALSE
END;
```

1.2 Búsqueda secuencial con centinela.

Simplifica el algoritmo anterior para realizar solo una comparación.Se usa una tabla de N+1 elementos para trabajar con una de N y se guarda el elemento que se quiere buscar en la última posición de la tabla.

```
UNTIL Tabla[i].Clave=Clave
```

Así nos podemos ahorrar la segunda comparación, ya que nunca se superarán los límites de la tabla.

Para mejorar aun más el procedimiento anterior vamos a devolver el indice de la posición en la que se encuentra el elemento buscado en lugar del contenido para poder luego cambiarlo o borrarlo.

```
PROCEDURE Buscar(Tabla:Ttabla; VAR Posicion:INTEGER;
                 VAR Encontrado:BOOLEAN; Clave:Tclave);
VAR i:INTEGER;
BEGIN
   i:=0;
   REPEAT
      i:=i+1;
   UNTIL Tabla[i].Clave=Clave;
   IF i=n+1
        THEN Encontrado:=FALSE
        ELSE BEGIN
                Posicion:=i;
                Encontrado:=TRUE
             END
END;
```

1.3 Búsqueda dicotómica.

El hecho de que la tabla esté ordenada facilita principalmente la operación de búsqueda de una clave en la misma.Esta mayor eficiencia, se debe a la posibilidad de realizar una búsqueda dicotómica dentro de la tabla, en lugar de tener que examinar todos los elementos de la misma, hasta encontrar el valor buscado o se agote la tabla.

Sin embargo, las operaciones de inserción y borrado se complican más, ya que deben colocar cada elemento en la posición que le corresponda según el orden que se haya establecido.En la tabla desordenada es posible insertar y borrar de cualquier lugar sin restaurar ningún orden.

Es común realizar las inserciones de forma desordenada y luego ordenar el vector mediante algún método de ordenación para posteriormente realizar las operaciones que precisen una búsqueda.

```
PROCEDURE Binaria (Tabla:Ttabla;VAR Posicion:INTEGER;Clave:Tclave;
                                    inferior,superior:INTEGER);
VAR Centro:INTEGER;
BEGIN
   REPEAT
      Centro:=(Inferior+Superior) DIV 2
      IF Tabla[Centro].Clave>Clave
           THEN Superior:=Centro-1
           ELSE Inferior:=Centro+1
   UNTIL (Tabla[Centro].Clave=Clave) OR (Inferior>Superior);
   IF Tabla[Centro].Clave=Clave
        THEN Posicion:=0
        ELSE Posicion:=Centro
END;
```

1.4 Búsqueda dicotómica recursiva.

```
FUNCTION Binaria (Tabla:Ttabla;Clave:Tclave;Inf,Sup:INTEGER):INTEGER;

FUNCTION B_Binaria (Clave:Tclave;Inf,Sup:INTEGER):INTEGER;
VAR Centro:INTEGER;
BEGIN
  IF Inf>Sup
     THEN B_Binaria:=0      (* Indica que no existen elementos *)
     ELSE BEGIN
             Centro:=(Inf+Sup) DIV 2;
             IF Tabla[Centro].Clave=Clave
                 THEN B_Binaria:=Centro
                 ELSE IF Tabla[Centro].Clave>Clave
                         THEN B_Binaria(Clave,Inf,Centro-1)
                         ELSE B_Binaria(Clave,Centro+1,Sup)
          END
END;
BEGIN      (* Binaria *)
  binaria:=b_binaria(clave,inf,sup)
END;
```

2. TRANSFORMACION DE CLAVES (HASH)

Para buscar un elemento en la tabla se puede hacer de varias formas: secuencial, dicotómica, si tiene una clave numérica podríamos hacer coincidir la clave con el índice y la otra forma es utilizar HASH.

En la estructura de tablas es interesante el poder determinar de forma unívoca la posición de un elemento dentro de la tabla en función de su clave.En esta realización se utilizará una función de transformación, que asigne a la clave correspondiente a un elemento una posición determinada dentro del vector.

A partir de ahora vamos a tener un conjunto de elementos del mismo tipo ordenados por un campo clave.La transformación de claves va a tratar de conseguir la posición del elemento en una tabla a partir de su clave.

Vamos a definir una función H aplicación del conjunto de claves sobre el conjunto de direcciones de la tabla.Dada una clave C_1, al aplicarla la función H nos va a dar una dirección de tabla, aunque, realmente lo que va a generar es un índice.A esta función se la conoce con el nombre de transformación de claves.

2.1 MANEJO DE COLISIONES

Cuando se utiliza el método HASH, el problema principal es la elección de la función de transformación.Dado que el número de claves posibles puede ser mayor que el número de índices del vector que va a contener la tabla, habrá varias claves que se reflejen en un mismo índice del vector.Cuando surge este problema, se dice que hay **colisiones**.

Si tenemos dos claves C_1 y C_2 distintas, y ambas pertenecientes al conjunto de claves C, si aplicamos $H(C_1)$ y nos da la dirección que $H(C_2)$ entonces decimos que C_1 y C_2 son sinónimos.

Cuando se produce una colisión hay que tratar de una manera especial algún elemento, hay que recurrir a las funciones de manejo de colisiones.Estas dependen del tipo de almacenamiento y existen dos:Almacenamiento externo e interno.

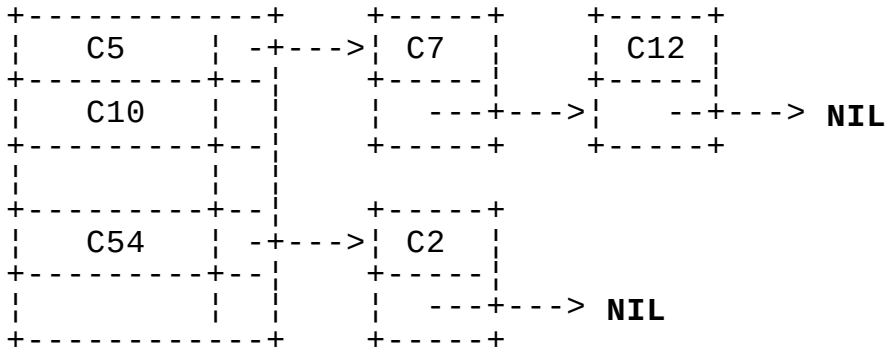
Almacenamiento externo

Usamos espacios fuera de las de la tabla para colocar las colisiones.Dentro del almacenamiento externo hay varios tipos:Encadenamiento directo y zona de overflow.

Encadenamiento directo.

Esta realización considera la tabla como un vector en el que cada posición contiene un elemento y un campo adicional con el comienzo de la lista de elementos con los que existe colisión.Es decir, las posibles colisiones se resuelven construyendo una lista de elementos cuya imagen hash coincida.

Al aplicar la función de transformación, si la posición no está ocupada, se incluye el elemento en la posición calculada.En otro caso se incluye el elemento en la lista de elementos que entran en colisión con el que ocupa la posición calculada en la tabla.

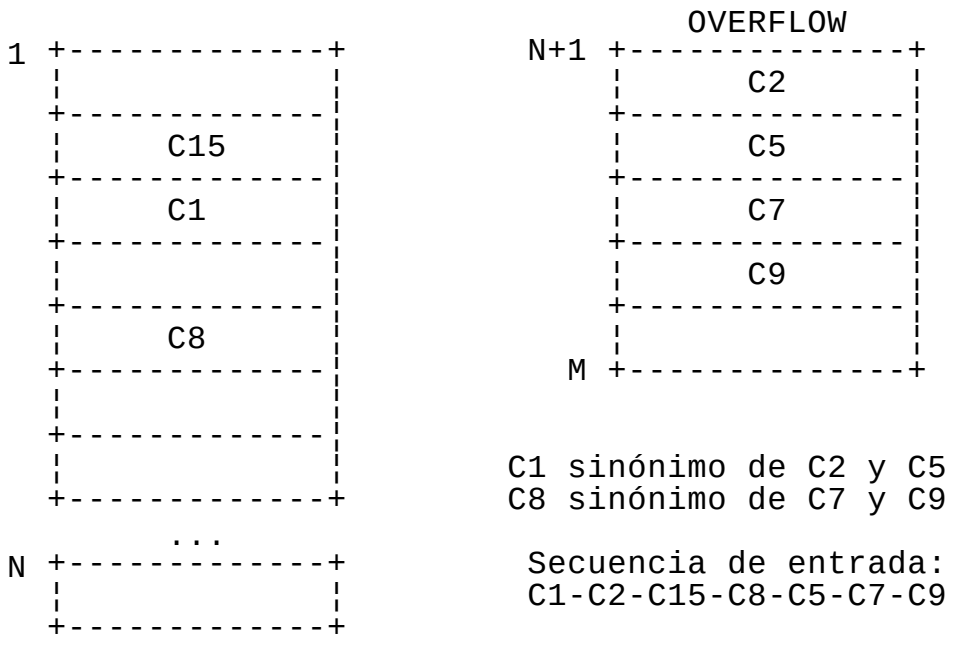


× Ventajas: eficientes y rápidos.

× Inconvenientes: Para cada elemento de la lista se debe reserVAR un espacio para punteros lo que significa un desaprovechamiento de memoria en el "manejo de lista".

Zona de Overflow.

Se reserva espacio en cierta zona de externa a la propia tabla, de aproximadamente el 10% de su tamaño, para introducir las colisiones.Cada sinónimo se almacena en la primera celda disponible de la zona de overflow.



× Inconveniente: Desaprovechamiento de memoria (poco).Es poco eficiente cuando se han producido colisiones, ya que la búsqueda en la zona de overflow es secuencial.

× Ventajas: Ocupa menos memoria que el anterior.El algoritmo de búsqueda y de inserción es mas sencillo.

Almacenamiento interno

Cuando el espacio usado para almacenar las colisiones esta dentro de los límites de la tabla.Dentro del almacenamiento interno están:Encadenamiento directo y encadenamiento vacío.

Encadenamiento directo.

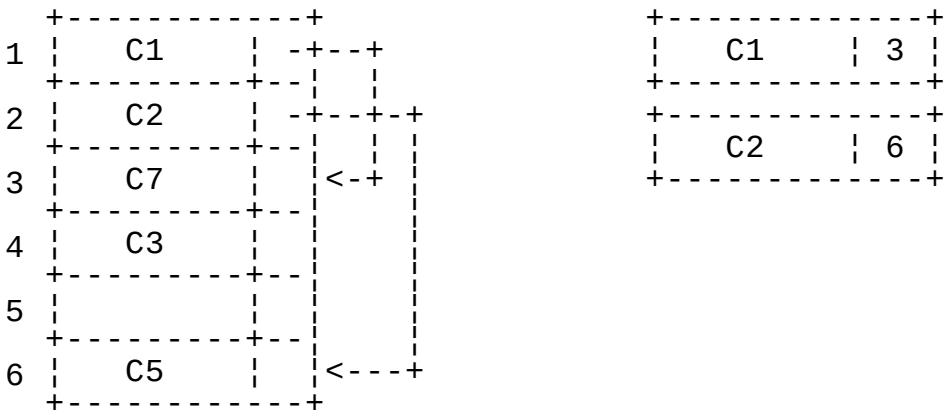
Se usa dentro de la tabla un campo de tipo puntero para que apunte al siguiente colisionado, que estará dentro de la tabla.En ese campo se guarda la dirección del siguiente colisionado.

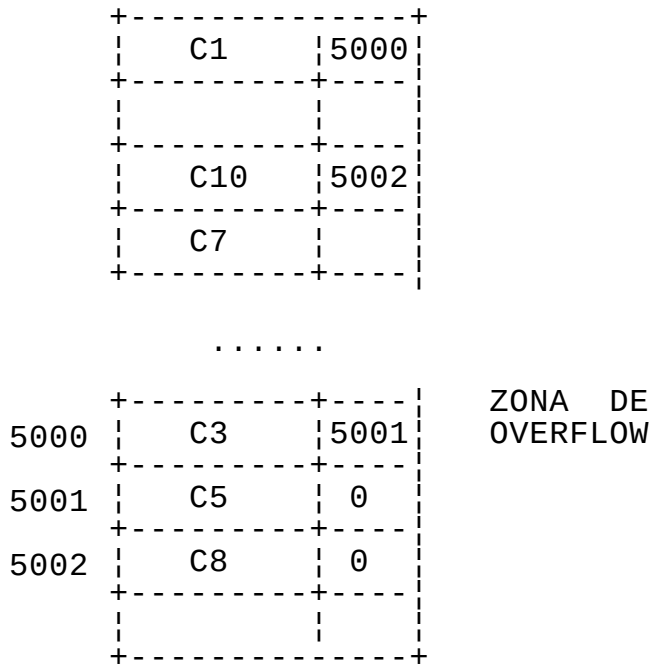
En el encadenamiento directo con zona de overflow podemos sobredimensionar la tabla para almacenar las colisiones, en esta zona las casillas estarán encadenadas con una variable que apunte al primer espacio libre de la zona de overflow.Consiste en enlazar todos los elementos cuyas claves generan igual indice primario por medio de enlaces dentro de la tabla a las nuevas posiciones ocupadas por estos elementos.

× Inconvenientes: Espacio reservado en cada elemento para el enlace.

× Ventajas: Más rápido que el externo con zona de overflow ya que evita la búsqueda secuencial.

× Ocupación de memoria: Depende del método usado.El primer caso ocupa menos memoria, y el segundo es más rápido.





Insertando un cero en el campo correspondiente a las colisiones en este último ejemplo, queremos indicar que ya no hay más colisiones a partir de él.En este son sinónimos C1,C2 y C3 por una parte, y por otra C8 y C10.

Encadenamiento Vacío.

Al utilizar este método, cuando se produce una colisión, se realiza una busqueda en el resto de la tabla, hasta que se encuentre la clave o bien hasta que se encuentre una posición no ocupada.

Si deseamos realizar una inserción,cuando encontremos la clave, detectamos que ya existía el elemento, y al encontrar un lugar vacío, detectamos la posición en la que lo podemos insertar.

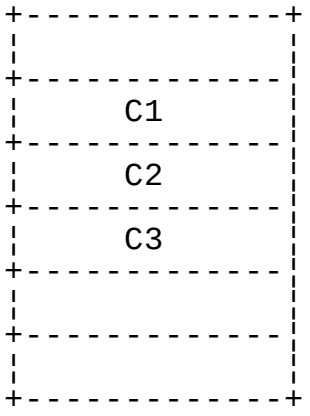
Si la operación a realizar es de búsqueda, el hecho de encontrar la clave determina que el elemento está en la tabla.Por el contrario, encontrar un lugar vacío significa que el elemento no está en la tabla.

Evidentemente, en los dos tipos de operaciones es posible recorrerse toda la tabla sin haber encontrado ningún espacio vacío ni haber encontrado la clave correspondiente al elemento.Por tanto, otra terminación de la operación se producirá también cuando se haya examinado la tabla en su totalidad, sin que se hayan cumplido las condiciones anteriores.

La forma más usual de realizar la búsqueda, es tratar las posiciones de la tabla de forma secuencial.Esto es, después de examinar una posición se examina la siguiente, considerando la primera posición como siguiente a la último, y así sucesivamente.

× Ventajas: Elimina enlaces.Poca ocupación de memoria.

× Inconvenientes: Si queremos buscar C_2 en el ejemplo, encontraría C_1 con colisión y entonces secuencialmente en el resto de la tabla.En cuanto hay una colisión la búsqueda e inserción es secuencial y, por tanto, más lento.



Serían sinónimos los tres, e insertados en el orden siguiente C1-C2-C3

Inspección en la zona de overflow

Existen dos métodos principales de inspección dentro de una tabla HASH: La inspección lineal y la cuadrática.

Inspección lineal: la generación de índice se hace de la siguiente forma:

× Se aplica la función HASH a la clave que llega y se obtiene así la primera dirección.

$$H(C_1)=H_0$$

× Si se produce una colisión se utiliza la siguiente formula para obtener la siguiente dirección a inspeccionar:

$$Hj=(H_0+i)MOD\ N$$

donde i es el número de colisiones que llevamos hasta el momento, y N es el número primo más cercano al tamaño de la tabla por defecto.

× El algoritmo va a parar cuando:

T(hi)=vacío, la dirección de la tabla esté vacía.
T(hi)=clave, la clave ya está almacenada.
$hi=H_0$ da toda la vuelta a la tabla luego la tabla esta llena.

Este método tiene la desventaja de que los elementos tienen a agruparse alrededor de las claves primarias (Claves que han sido insertadas sin colisionar).Idealmente, desde luego, debería escogerse una función que, a su vez, distribuyera uniformemente las claves en los sitios restantes.En la practica, sin embargo, este tiende a ser demasiado costoso, y son preferibles métodos que sean sencillo de cálculo y superiores a la función lineal.

Inspección cuadrática: la generación de índice se hace de la siguiente forma:

× Se aplica la función HASH a la clave que llega y se obtiene así la primera dirección.

$$H(C_1)=H_0$$

× Si se produce una colisión se utiliza la siguiente formula para obtener la siguiente dirección a inspeccionar:

$$Hj=(H_0+i^2)MOD\ N$$

donde N será un número primo e i tomará los valores 1, 4, 9, ..., 16, ...

× Este método no inspecciona toda la tabla, por lo tanto, puede avisar de que la tabla llena sin que lo esté.Si N es primo al menos mira la mitad de las posiciones de la tabla.

× Como no podemos asegurar que $hi=H_0$ en algún momento, se tiene que parar de otras forma, por ejemplo cuando i=n/2 ya que no va a mirar todas las posiciones, va a mirar la mitad.

Aunque prácticamente no necesita cálculo adicional, evita bien el agrupamiento primario.Una ligera desventaja es que el método no inspecciona todos los lugares de la tabla, es decir, puede no encontrarse donde insertar un nuevo elemento, a pesar de haber lugares vacíos.Si N es un número primo, la inspección cuadrática recorre, de todas formas, al menos la mitad de la tabla.

En la práctica, la desventaja de este método no tiene importancia, ya que es muy raro que se presenten N/2 colisiones consecutivas, y sólo sucede cuando la tabla está casi llena.

2.2 FUNCION DE TRANSFORMACION

Elección de la función de transformación.

Es muy deseable que una buena función de transformación distribuya las claves en el campo de los valores indice de forma tan uniforme como sea posible.Aparte de esto, la función puede distribuir las claves de cualquier manera y, de hecho, es mejor si se comporta de forma completamente aleatoria ya que se produce un número mínimo de colisiones y, por tanto, mínimo tiempo de tratamiento de la clave en caso de colisión.

Tiene que ser rápida y fácil de calcular, es decir, debe estar formada por unas pocas operaciones aritméticas simples.Esta función debe eliminar la información no distintiva de la clave.

Sobredimensionar la tabla un 10% no afecta a la elección de la función de transformación, pero lo hacemos siempre.

Funciones de Transformación.

Básicamente todos los métodos siguen dos pasos:

- Si la clave no es numérica pasarla a numérica.
- Ajustar la clave al espacio de direcciones de la tabla.

Método del centro de los cuadrados.

Se multiplica la clave por si misma y se toman los dígitos centrales del cuadrado, ajustándolos al rango.

Ejemplo

$172148^2 = 246\ 3493\ 3904$

+--------------> 3493 * 0.7 para el ajuste

en una tabla de 7000 posiciones.

Extracción de bits.

Esta función consiste en considerar la representación binaria de la clave y el calculo de la posición de la clave en función de aquellos bits que tengan menor probabilidad de coincidir, con lo que se minimizarían las colisiones.

Esta función es principalmente útil cuando el tamaño de la tabla es potencia de 2.Si tenemos una tabla con rango $1..2^n$ deberíamos tomar los n bits con menor probabilidad de colisión.

Método de la división.

Es el más empleado.Consiste en dividir la clave por un número no primo ligeramente inferior al tamaño de la tabla.Este método produce para claves consecutivas direcciones consecutivas.La desventaja es que los dígitos finales influyen mas que los demás en la transformación.

La elección del número por el que se divide es importante.En general pueden darse razones para elegir un número primo.

Ejemplo

172148 en tabla de 7000 posiciones.

172148 MOD 6997 = 4220

No hay que ajustar porque siempre va a estar en el rango de posiciones de la tabla.

Método del desplazamiento.

Los dígitos exteriores de la clave se desplazan hacia dentro y se suman.

Ejemplo

54 2422 241 en una tabla de 7000 posiciones.

```
54 2422 241

  2422
   241
+   54
  ----
  2717 * 0.7   para el ajuste
```

Método de superposición

En este método la clave se divide en varias partes, todas ellas de la misma longitud excepto quizás la última.Estas partes se combinan para calcular la dirección correspondiente a la clave.

Hay dos formas de conseguir la función mediante este método:

× **Superposición por desplazamiento** en la que todas las partes en las que se ha partido la clave se desplazan hacia el bit menos significativo y se suman entre si.

× **Superposición por plegado**, y desplaza las partes que ocupan posición impar igual que en la forma anterior y las que ocupan posición par las desplaza después de haberlas invertido.Posteriormente realiza la suma de las diferentes partes para calcular el valor de la función.

Ejemplo

345147098100

× Por desplazamiento 345+147+098+100 = 690
× Por plegado 345+741+098+001 = 1185

2.3 FACTOR DE CARGA

$$a = \frac{Nœmero\ de\ posiciones\ ocupadas}{Nœmero\ de\ posiciones\ totales}$$

Este dato nos va a dar una idea del porcentaje de tabla que estamos ocupando y, por tanto, del desperdicio de memoria.

× Si tiende a 0, el desperdicio es grande porque hay muchos huecos.
× Si tiende a 1, hay aprovechamiento de la memoria, pocos huecos.

2.4 HASH PERFECTO Y MINIMO

Hash perfecto.

Seria un método Hash que no produzca ninguna colisión (hipotético porque aunque estén muy bien diseñado se van a producir).Todos los métodos Hash que se diseñen tienen que aproximar a este Hash perfecto.

Hash mínimo.

Sería un método Hash hipotético que no produciría ningún desperdicio de memoria.También todos los diseñados se tienen que acercar al Hash mínimo en el sentido de que produzcan una ocupación de memoria lo mas pequeña posible.

Hash perfecto y mínimo.

Conjuntando estos 2 conceptos anteriores sería un método que, por una parte, no produzca colisiones (perfecto) y tenga un tiempo de acceso mínimo; y por otro, mínimo desprovechamiento (mínimo).
Cuando nos acercamos al Hash perfecto nos alejamos del mínimo y viceversa.Si la tabla está poco ocupada, menor es el riesgo de colisión y si esta mas ocupada hay más riesgo.
Normalmente se obtiene un buen rendimiento cuando sobredimensionamos la tabla entorno a un 10%, es decir, con un factor de carga 0'9.Un método para calcular el factor de carga es el siguiente:

$$E = \frac{-1}{a} \_ \ln(1-a)$$

siendo E el número medio de accesos que hay que realizar a la tabla para obtener o grabar un elemento.

Dando algunos valores a esta función tenemos:

alfa	0.1	0.25	0.5	0.75	0.9	0.95	0.99
E	1.05	1.15	1.39	1.85	2.56	3.15	4.66

Esta formula es únicamente valida para aquellos métodos HASH que distribuyan las claves uniformemente.Para el método de inspección lineal se usará la siguiente:

$$E = \frac{1-\frac{a}{2}}{1-a}$$

Y tendríamos los siguientes valores:

alfa	0.1	0.25	0.5	0.75	0.9	0.95
E	1.06	1.17	1.50	2.50	5.50	10.5

Se puede llegar a acercarse a un Hash perfecto complicando la función de transformación, pero esta complicación puede llegar a ser tal que no compense porque redunda en un mayor tiempo de acceso.

En concreto existe un método, método de Cichelli, que es una solución para obtener una función Hash perfecta y mínima.Pero tiene una importantes restricción:Las claves con las que va a trabajar la función tiene que ser estático y conocido (el número de claves no puede aumentar ni disminuir).

Ejercicio 1.

Después de realizar un número conveniente de pruebas, se ha comprobado que la cantidad de sinónimos de la siguiente función de transformación es elevada, dado el carácter alfabético de la clave.
Se pide modificar oportunamente la función de forma que se reduzca el número de sinónimos.

```
FUNCTION Hash (a:Tipoclave):INTEGER;
VAR
   i,Suma:INTEGER;
BEGIN
   Suma:=0;
   FOR i:=n DOWNTO 1 DO
                  Suma:=Suma+(ORD(a[i])-ORD('A'));
   Hash:=Suma MOD 32565
END;
```

Tenemos que ponderarlo ya que si no palabras como ROSA y AROS serían sinónimos al dar la misma clave.Por ejemplo, podríamos multiplicar la variable **Suma** por la posición **i** de cada letra.

Ejercicio 2.

Se desea codificar un programa para la DGT de Madrid que gestione la ITV durante un cierto año.Dado que el nº de vehículos con matricula de Madrid que tienen que pasar la inspección a lo largo de un año nunca supera los 50000, se ha elegido como estructura de datos una tabla Hash.
Los formatos de las matrículas serían M-XXXXXX (packed ARRAY 8 CHAR).Intentar codificar una función Hash dadas las características del problema que sea lo mas efectiva posible.

× Debemos conseguir eliminar información redundante como M- y el primer dígito también es poco significativo porque casi todos serán 8 ó 9.

× Método de la división.
nº MOD primo más cercano a 499999.

× Para pasar la matricula números.

uni * 10^0 + dec * 10^1 + cen * 10^2 + mil * 10^3 + decmil * 10^4 ...

× Carácter a nº
ORD('9') - ORD('0');

Posición en tabla	Exp
8	0
7	1
6	2
5	3
4	4

8-Pos=Exp

---

a^b = EXP(b*LN(a))

```
FUNCTION Hash1 (Matricula:PACKED ARRAY ...):LONGINTEGER;
VAR i,Suma:INTEGER;
BEGIN
   Suma:=0;
   FOR i:=8 DOWNTO 4 DO
       Suma:=Suma+TRUNC((ORD(Matricula[i])-ORD('0'))*EXP((8-
i)*LN(10)));
   Hash1:=Suma MOD Tamanio      (* nº primo más cercano *)
END;
```

Para implementar el mismo HASH con manejo de colisiones mediante inspección lineal.

```
FUNCTION Hash (Matri...):INTEGER;
VAR Di:INTEGER;

   FUNCTION Hash1 (Matri...):INTEGER;

   FUNCTION PosicionVacia (D1:INTEGER):BOOLEAN;
   BEGIN
      PosicionVacia:=Tabla[D1].CampoClave=Vacia;
   END;

   FUNCTION BuscaHuecos (D0:INTEGER;Matricula...):INTEGER;
   VAR Di,i:INTEGER;
   BEGIN
      i:=1;     (* Al entrar es porque hay colisión *)
      REPEAT
         Di:=(D0+i) MOD Tamanio;
         i:=i+1
       UNTIL PosicionVacia (Di) OR (Di=D0)   (* Ha dado vuelta a toda
tabla *)
               OR (Matricula=Tabla[Di].CampoClave); (* Dato ya en tabla
*)
      IF Di=D0
          THEN "Tabla llena"
          ELSE IF PosicionVacia(Di)
                    THEN BuscaHueco:=Di
                    ELSE "La clave está"
   END;

BEGIN        (* Función Hash *)
   Di:=Hash1(Matricula);
   IF PosicionVacia(Di)
       THEN HASH:=Di
       ELSE  Hash:=BuscaHuecos(Di,Matricula); (* si esa pos no es la
clave *)
END;
```

Ejercicio 3. (7 JUN 90)

Se desea implementar un TAD "matriz enorme de enteros", de 5000 filas y 5000 columnas.Tras un estudio del problema, se ha observado que las matrices de dicho TAD nunca van a contener mas de 1500 enteros.Por tanto, se ha llegado a la conclusion de que si se implemente dicha tabla con una estructura del tipo ARRAY

```
[1..5000,1..5000] of INTEGER;
```

se requeriria un espacio de memoria de 25 millones de posiciones de enteros seria sumamente inefectivo.

a/ Diseñar una estructura de datos que permita implementar el TAD anteriormente descrito con un consumo racional de memoria, teniendo en cuenta que, el acceso a una posicion de la estructura se realiza proporcionando un número de fila y otro de columna, ambos entre 1 y 5000, y que dicho acceso deberá ser muy rápido.

b/ Diseñar una funcion de acceso a dicha estructura, con la siguiente cabecera:

```
FUNCTION contenido (fila,columna:rango):INTEGER;
```

que devuelva el entero contenido en la posicion (fila,columna) de la matriz simulada.(Rango es un tipo de 1 a 5000)

a/ Tenemos 25 millones de claves reales y mediante la función HASH tenemos que convertirlas en 1500 ya que se usará una tabla de 1500 posiciones, sobredimensionada un 10% para la inspeccion lineal, luego tendrá 1650 posiciones.El número primo superior mas cercano es 1657.

```
CONST Tamanio=1657
TYPE Elemento:=RECORD
                  Fila,Col:[0..5000];
                  Num:INTEGER
               END;
Ttabla=ARRAY[0..1656] OF Elemento;
```

Utilizaremos una estructura de tipo Hash porque el acceso es muy rapido, el número de posibles claves es muy grande en realcion a número de elementos reales.Sobredimensionamos porque vamos a usar inspeccion lineal.

b/

```
FUNCTION Contenido (Fila,Columna:Rango):INTEGER;
VAR D0,Col:INTEGER;

   FUNCTION Hash (fila,columna:rango):INTEGER;
   VAR Aux:REAL;
       Cociente:INTEGER;
   BEGIN
      Aux:=((Fila-1)*5000)+(Col-1);
      Cociente:=TRUNC(Aux/Tamanio);
      Hash:=TRUNC(Aux-(Cociente*Tamanio))
   END;
```

```
BEGIN      (* Función contenido *)
   D0:=Hash(Fila,Col);
   Col:=0;
   INTEGER (Tabla[D0].Fila<>Fila) OR (Tabla[D0].Col<>Col)
              and (Tabla[D0].Fila<>0 and Col<Tamanio-1);
   BEGIN
      D0:=(D0 MOD Tamanio)+1;
      Col:=Col+1
   END;
   IF Tabla[D0].Fila=0
       THEN "no esta"
       ELSE Contenido:=Tabla[D0].Num
END;
```

17 Septiembre 91.Ejercicio 2.

En un HASH perfecto hay un espacio en la tabla para cada elemento luego no es necesario relizar operaciones de borrado.También es necesario conocer el conjunto de claves, luego no se puede usar en ningún apartado.

En el caso 2 ocupa más memoria porque se necesita un campo adicional en cada elemento de la tabla para la lista.Para el B.

El más rápido es el de listas ya que el otro es el peor de los casos tendremos que mirar en toda la tabla para encontrar el elemento.En el caso de la lista solo habrá que mirar en la lista.

7 Junio 90.Ejercicio 3.

Cada vez que borra una clave pone el campo clave a borrado, pero a la hora de buscar un hueco ignora a los borrados.Hay que insertar en el REPEAT-UNTIL del insertar un

```
                OR Tabla[Pos].Clave=Borrada
```

Detrás del REPEAT-UNTIL hay un IF

```
 IF Tabla[Pos].Clave=Elem.Clave OR Tabla[Pos].Clave=Borrada
```

donde pone Pos:=Pos+Colisiones, tendría que poner Pos:=Pos+1 para que fuese una inspección lineal.

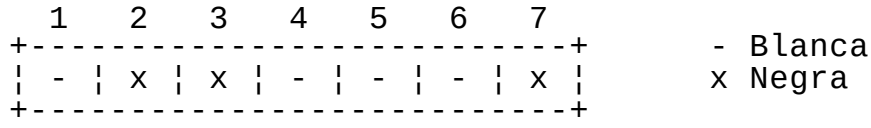
Ejercicio 6 (Febrero 92)

Dada la siguiente declaración de tipo:

```
Tcanica=(Blanca,Negra);
Tbingo=ARRAY[1..N] OF Tcanica;
```

Codificar un procedimiento que imprima todas las posibles cominaciones de dos canicas tomadas del ARRAY en las que una sea blanca y la otra negra.Por cada combinación se imprimirá la posición del ARRAY que ocupa cada canica.

Ejemplo:



Resultado:
(1,2)(1,3)(1,7)(2,4)(2,5)(2,6)(3,4)(3,5)(3,6)(4,7)(5,7)(6,7).La combinación (2,1) se considera equivalente a la (1,2) y no se imprimirá.

```
PROCEDURE Bingo (Bingo:Tbingo);
VAR
   i,j:INTEGER;
BEGIN
   FOR i:=1 TO n-1 DO
      FOR j:=i+1 TO n DO
         IF Bingo[i]<>Bingo[j] THEN WRITELN(i,j)
END;
```

1. CONCEPTO.

Una lista es una estructura de datos homogénea y dinámica, que va a estar formada por una secuencia de elementos, donde cada uno de ellos va seguido de otro o de ninguno.

- Homogénea: Todos los elementos que la forman tienen el mismo tipo base.

- Dinámica: Puede crecer o decrecer en tiempo de ejecución según nuestras necesidades.

dos listas pueden ser diferentes si:

- No tienen el mismo número de elementos:

  L1: gato, perro.
  L2: gato, canario, cerdo.

- Cuando, aun teniendo el mismo número de elementos, estos son distintos:

  L1: gato, perro.
  L2: gato, cerdo.

- Cuando, aun teniendo el mismo número de elementos y siendo estos los mismos, no están dispuestos en el mismo orden.

  L1: gato, perro.
  L2: perro, gato.

2. CRITERIOS DE CLASIFICACION.

Hay varios criterios para clasificar las listas: según su modo de acceso o según su información de acceso.

2.1 Modo DE ACCESO.

Atendiendo a este, se dividen en densas y enlazadas.El modo de acceso es independiente de la implementación realizada.

Listas densas

Se caracterizan porque los elementos siguen una secuencia física.Sabemos cuales es el siguiente elemento porque para acceder a él hemos tenido que pasar por todos los anteriores.
La localización de un elemento cualquiera será:

- El primero si es el primer elemento de la lista.
- N-esimo si para llegar a el hemos pasado por N-1 elementos.

Siguen una estructura física secuencial luego se pueden implementar utilizando ficheros, ARRAYs y punteros.

Listas enlazadas

Son aquellas en las que cada elemento que los compone contiene la información necesaria para acceder al elemento siguiente.La localización de un elemento cualquiera será:

- Un elemento de la lista tendrá la dirección K si K es el primero y K es conocido (dirección de inicio).
- Estará en la dir J si J está contenida en el elemento anterior.

Se pueden implementar con punteros enlazados y con ARRAYs.

2.2 INFORMACION DE ACCESO.

Listas ordinales

Los elementos se van colocando en la lista a medida que llegan y se identifican por el orden de llegada.El acceso a un elemento es por su orden o posición relativa dentro de la lista.

Listas calificadas

Los elementos se clasifican por una clave y pueden estar ordenados o no estarlo. A un elemento se accede por la información contenida en un campo clave.

Diferencias: En la primera clase importa en orden de llegada, mientras que en la segunda depende de la clave.

3. PILAS.

Una pila es una lista ordinal en la que el modo de acceso a sus elementos es del tipo LIFO.Los añadidos y extracciones de elementos de una estructura se realizan solo por un extremo, luego el único elemento accesible de la pila es el que se encuentre en la cima.Esto exigirá que la manipulación sobre un elemento, necesite que el mismo ocupe la posición de cima.

Sobre una estructura de tipo pila, surgen de forma natural las operaciones que permiten añadir elementos y quitar elementos.

3.1 Especificaciones del TAD pila

Operaciones: Crear, Introducir un elemento, Extraer o sacar (Cima y Borrardo), Vacía.Vamos a definir estas operaciones con el siguiente formato:

Nombre, parámetros de entrada, salida, efecto y excepciones.

- **Crear**: Pila, Pila, Crea una pila, Ninguna.

- **Introducir**: Pila y elemento, Pila, Incrementar en un elemento la pila, Llena si está limitada (ARRAY) y ninguna si no hay límites.

- **Cima**: Pila, El primer elemento, Ninguno, Pila vacía.

- **Borrado**: Pila, Pila, Decrementa en uno el número de elementos de la pila, Pila vacía.

- **Extraer**: Pila, Elemento y pila, Decrementa en uno el número de elementos de pila, Pila vacía.

- **Vacía**: Pila, TRUE o FALSE, Ninguno, Ninguno.

En el caso de implementación con ARRAYs se necesitará:

- **Llena**: Pila, BOOLEAN, Ninguno, Ninguno.

En esta definición tenemos dos constructores de la estructura, que son las operaciones **Crear** y **Introducir**.Con ellas conseguimos una pila que no tenga ningún elemento y con inserciones sucesivas, podemos conseguir cualquier pila.

El resto de las operaciones nos permitirán extraer elementos de la pila, consultar el elemento de cima y decidir si la pila tiene, o no, elementos.

3.2 Implementación utilizando tablas

Esta realización consiste en ir guardando consecutivamente los elementos de la pila en un vector de tamaño fijo.Un índice marcará la posición del último elemento que se ha añadido a la pila.Por tanto, las inserciones en la estructura se realizarán en la posición inmediatamente siguiente a la posición marcada como cima, pasando a ser esta nueva posición ocupada la nueva cima de la pila.

El hecho de utilizar un vector para almacenar los elementos, puede conducir a la situación en que la pila esté llena, es decir, que no quepa ningún elemento más.Esto se producirá cuando el índice que señala la cima de la pila sea igual al tamaño del vector.

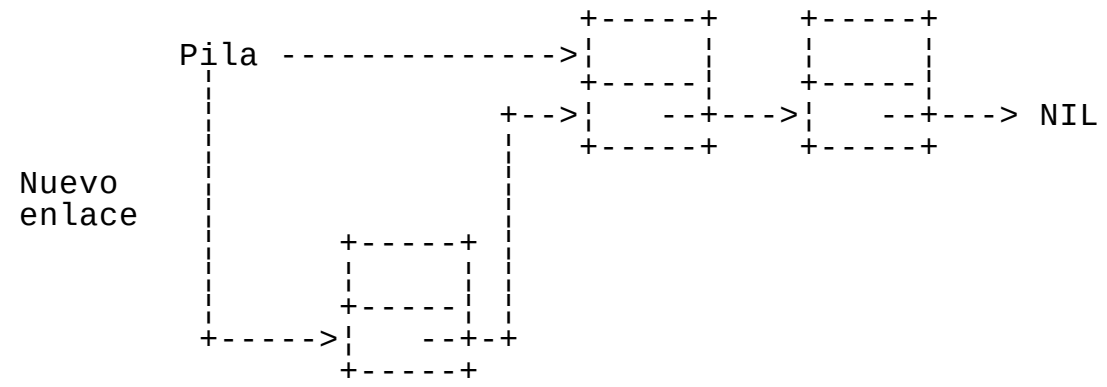
```
TYPE
   Tpila=RECORD
            Elemento:ARRAY [1..n] OF Elem;     (* Lista de elementos *)
            Puntero:INTEGER;          (* Apunta a la cima de la pila *)
         END;


PROCEDURE Crear (VAR Pila:Tpila);
BEGIN
     Pila.Puntero:=0;       (* Pila vacía *)
END;
```

```
PROCEDURE Introducir (VAR Pila:Tpila;El:Ele);
BEGIN
     IF Llena (Pila)
          THEN " PILA LLENA "
          ELSE BEGIN
                  Pila.Puntero:=Pila.Puntero+1;
                  Pila.Elemento[Pila.Puntero]:=El
               END
END;


FUNCTION Cima(Pila:Tpila):Ele;
BEGIN
   IF Vacia (pila)
        THEN " ERROR "
        ELSE Cima:=Pila.Elemento[Pila.Puntero]
END;


PROCEDURE Borrar (VAR Pila:Tpila);
BEGIN
   IF Vacia (Pila)
        THEN " ERROR "
        ELSE Pila.Puntero:=Pila.Puntero-1
END;


FUNCTION Vacia (Pila:Tpila):BOOLEAN;
BEGIN
       Vacia:=Pila.Puntero=0;
END;


PROCEDURE Extraer (VAR Pila:Tpila,VAR Ele:Tele);
BEGIN
      IF Vacia(Pila)
           THEN " ERROR "
           ELSE BEGIN
                   Ele:=Pila.Elemento[Pila.Puntero];
                   Pila.Puntero:=Pila.Puntero-1
                END
END;


FUNCTION Llena (Pila:Tpila):BOOLEAN;
BEGIN
       Llena:=Pila.Puntero=n      (* El ARRAY se ha declarado de 1 a n
*)
END;
```

3.3 Implementación con punteros

En esta realización se construirá la pila como una lista encadenada mediante puenteros.Por ello cada elemento de la pila tendrá, ademas del campo de información correspondiente, un campo de enlace con el siguiente elemento de la lista.

```
TYPE
   Tpila=^Elemento;
   Elemento=RECORD
               Dato:Tinfo;
               Sig:Tpila;
            END;


PROCEDURE Crear (VAR Pila:Tpila);
BEGIN
       Pila:=NIL;
END;


PROCEDURE Introducir (VAR Pila:Tpila,Elem:Tinfo);
VAR Aux:Tpila;
BEGIN
   NEW(Aux);
   Aux^.Dato:=Elem;
   Aux^.Sig:=Pila;
   Pila:=Aux
END;
```




```
FUNCTION Cima (Pila:Tpila):Tinfo;
BEGIN
   IF Vacia(Pila)
        THEN " ERROR "
        ELSE Cima:=Pila^.Dato
END;


PROCEDURE Borrar (VAR Pila:Tpila);
VAR Aux:Tpila;
BEGIN
   IF NOT Vacia(Pila)
        THEN BEGIN
                Aux:=Pila;
                Pila:=Pila^.Sig;
                DISPOSE(Aux)
             END
END;
```

```
FUNCTION Vacia (Pila:Tpila):BOOLEAN;
BEGIN
       Vacia:=Pila=NIL
END;


PROCEDURE Sacar (VAR Pila:Tpila,VAR Ele:Tinfo);
VAR Aux:Tpila;
BEGIN
   IF NOT Vacia(Pila)
        THEN BEGIN
                Ele:=Pila^.Dato;
                Aux:=Pila;
                Pila:=Pila^.Sig;
                DISPOSE (Aux)
             END
END;
```

4. COLAS.

4.1 Especificaciones del TAD cola

Podemos definir la estructura cola, como una secuencia de elementos, en la que la adición de nuevos elementos a la estructura se realiza por un extremo, que llemeremos **final**, y las extracciones de elementos se realiza por el otro, que llamaremos **principio**.

La misma definición de la estructura de cola, implica que el primer elemento que se inserte en la estructura, será el primero que pueda extraerse de la misma (FIFO).Asi pues, la manipulación de un elemento de la cola, exige que dicho elemento ocupe la posición de principio.

Vamos a ver esta estructura implementada con ARRAYS y con listas enlazadas, definiendo las operaciones con el siguiente formato:

Nombre, parámetros de entrada, salida, efecto y excepciones.

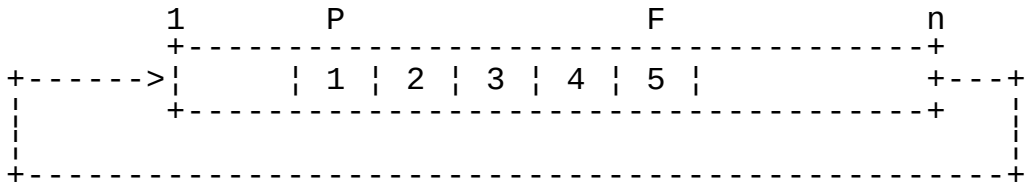
- **Crear**: Cola, Cola, Crea una cola, Ninguna.
- **Introducir**: Cola y elemento, Cola, Añade un elemento a la Cola, Llena si está limitada (ARRAY) y ninguna si no hay límites.
- **Primero**: Cola, Elemento, Ninguno, Cola vacía.
- **Borrado**: Cola, Cola, Quita elemento de de la cola, Cola vacía.
- **Sacar**: Cola, Elemento y cola, Decrementa en uno el número de elementos de Cola, Cola vacía.
- **Vacía**: Cola, BOOLEAN, Ninguno, Ninguno.

En el caso de implementación con ARRAYs se necesitará:

- **Llena**: Cola, BOOLEAN, Ninguno, Ninguno.

4.2 Implementación utilizando tablas

Esta realización consiste en representar la cola como una estructura de tipo registro con cuatro campos.Un campo **principio**, con el índice del primer elemento de la cola; un campo **final**, con el indice del último elemento de la cola; un campo **numero de elementos**, con la cantidad de elementos introducidos; y un campo **elementos**, que es una tabla con los elementos que están en la cola.



En este ejemplo, el primer elemento que puede salir de la cola es el apuntado por **principio**, es decir, el 1.Al insertar cualquier elemento, se colocaría detrás del apuntado por **final**.

Si almacenamos la cola en un vector circular se evita que, cuando el indice **final** llege al final de la tabla, sea imposible la inserción aunque queden libres las primeras posiciones de la tabla.

```
TYPE
   Tcola=RECORD
            Ele:ARRAY [1..n] OF Tinfo;
            Princ,Final,NumElem:INTEGER
         END;


FUNCTION Siguiente (Pos:INTEGER):INTEGER;
BEGIN
   Siguiente:=(Pos MOD Tamanio)+1
END;


FUNCTION Vacia (Cola:Tcola):BOOLEAN;
BEGIN
   Vacia:=Cola.NumEle=0;
END;


FUNCTION Llena (Cola:Tcola):BOOLEAN;
BEGIN
   Llena:=Cola.NumEle=n;
END;
```

Primero desplaza y luego introduce, por ello coloca en puntero F al final de la tabla y el puntero P al principio al crear, y número de elementos a cero.

```
PROCEDURE Crear (VAR Cola:Tcola);
BEGIN
   WITH Cola do
   BEGIN
      Final:=n;
      Principio:=1;
      NumEle:=0
   END
END;


PROCEDURE Introducir (VAR Cola:Tcola,Ele1:Tinfo);
BEGIN
   IF NOT Llena(Cola)
        THEN WITH Cola do
             BEGIN
                Final:=Siguiente(Final);
                Ele[Final]:=Ele1;
                NumEle:=NumEle+1
             END
END;


PROCEDURE Sacar (VAR Cola:Tcola;VAR Elem1:Tinfo);
BEGIN
   IF NOT Vacia(Cola)
        THEN WITH Cola do
             BEGIN
                Ele1:=Ele[Principio];
                Princ:=Siguiente[Principio];
                NumEle:=NumEle-1
             END
END;
```

Si no usamos la variable NumEle:

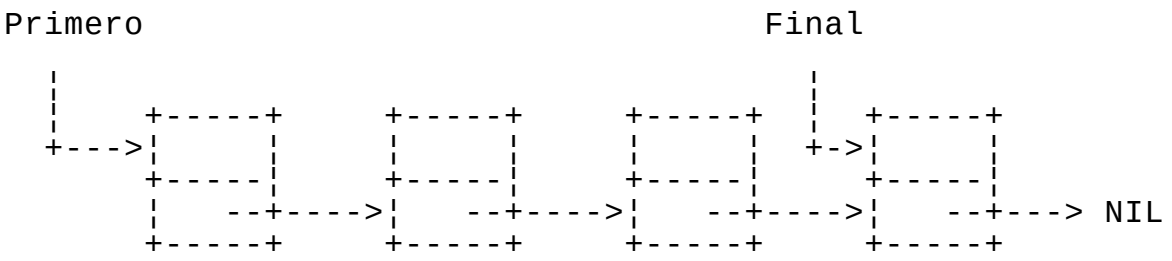
```
Vacia:=Siguiente[Final]=Principio;
Llena:=Siguiente[Siguiente[Final]]=Principio;
```

4.3 Implementación de colas con punteros

Hay varias formas de implementar una cola con punteros:

- Utilizando dos punteros, uno principio y otro final.
- Utilizando un solo puntero que apunte a la lista.

De cualquier forma siempre se inserta al final de la cola y se saca del principio.



Usando dos punteros

```
TYPE Puntero=^Elemento;
     Elemento=RECORD
                 Dato:Tinfo;
                 Sig:Puntero
               END;
     Tcola=RECORD
               Princ,Fin:Puntero
           END;


PROCEDURE Crear (VAR Cola:Tcola);    (* Pondremos ambos punteros a NIL
*)
BEGIN
   Cola^.Princ:=NIL;
   Cola^.Fin:=NIL
END;


PROCEDURE Introducir (VAR Cola:Tcola;Ele:Tinfo);
VAR Aux:Puntero;
BEGIN
   NEW(Aux);
   Aux^.Dato:=Ele;
   Aux^.Sig:=NIL;
   IF Vacia(Cola)
        THEN Cola^.Princ:=Aux
        ELSE Cola.Fin^.Sig:=Aux;
   Cola^.Fin:=Aux
END;


PROCEDURE Sacar (VAR Cola:Tcola, VAR Ele:Tinfo);
VAR Aux:Puntero;
BEGIN
   IF NOT Vacia(Cola)
       THEN WITH Cola do
            BEGIN
               Ele:=Princ^.Dato;
               Aux:=Princ;
               Princ:=Aux^.Sig;
               IF Princ=NIL THEN Fin:=NIL;
               DISPOSE (Aux)
            END
END;
```

Más de un elemento	Solo un elemento
× Ele:=Cola.Princ^.Dato	× Ele:=Cola.Princ^.Dato
× Aux:=Cola.Princ	× Aux:=Cola.Princ
× Cola.Princ:=Aux^.Sig	× Cola.Princ:=Aux^.Sig
× DISPOSE(Aux)	× Cola.Fin:=NIL DISPOSE(Aux)

```
FUNCTION Vacia (Cola:Tcola):BOOLEAN;
BEGIN
   Vacia:=Cola.Princ=NIL
END;
```

Usando un solo puntero

```
TYPE Tcola=^Elemento;
     Elemento=RECORD
                 Info:Tinfo;
                 Sig:Tcola
              END;


PROCEDURE Introducir (VAR Cola:Tcola, Ele:Tinfo);
VAR Aux,Nuevo:Tcola;
BEGIN
   Aux:=Cola;
   NEW(Nuevo);
   Nuevo^.Info:=Ele;
   Nuevo^.Sig:=NIL;
   IF Vacia(Cola)
        THEN Cola:=Nuevo
        ELSE BEGIN
                INTEGER Aux^.Sig<>NIL do
                    Aux:=Aux^.Sig;
                Aux^.Sig:=Nuevo
             END
END;
```

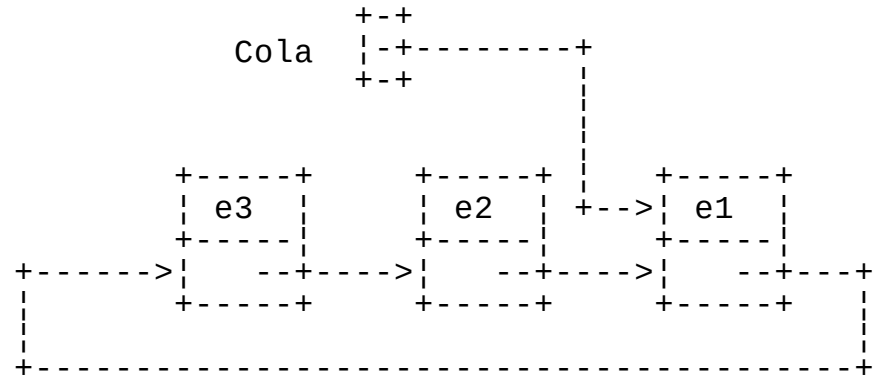
Cola vacía	Cola con varios elementos
× NEW(Nuevo)	× INTEGER Aux^.Sig<>NIL do Aux:=Aux^.Sig
× Nuevo^.Info:=Ele	× NEW(Nuevo)
× Nuevo^.Sig:=NIL	× Nuevo^.Info:=Ele
× Cola:=Nuevo	× Nuevo^.Sig:=NIL
	× Aux^.Sig:=Aux

```
PROCEDURE Sacar (VAR Cola:Tcola; VAR Ele:Tinfo);
VAR Aux:Tcola;
BEGIN
   IF NOT Vacia(Cola)
        THEN BEGIN
                Aux:=Cola;
                Ele:=Cola^.Info;
                Cola:=Cola^.Sig;
                DISPOSE(Aux)
             END
END;
```

4.4 Implementacion mediante lista circular

Otra realización es considerar la lista de almacenamiento de la cola como una lista circular, es decir, una lista en la que la última componente de la misma tiene como siguiente la primera.

Para utilizar una lista de este tipo, tendremos que guardar la posición de comienzo de la misma.Por esta razón, si realizamos una cola según esta idea, podemos considerarla como un puntero a la lista de almacenamiento.Dicho puntero tendrá siempre la dirección del último elemento insertado, ya que las inserciones se reallizarán siempre detrás del último elemento de la cola.Y el siguiente a este último elemento, será siempre el primero.



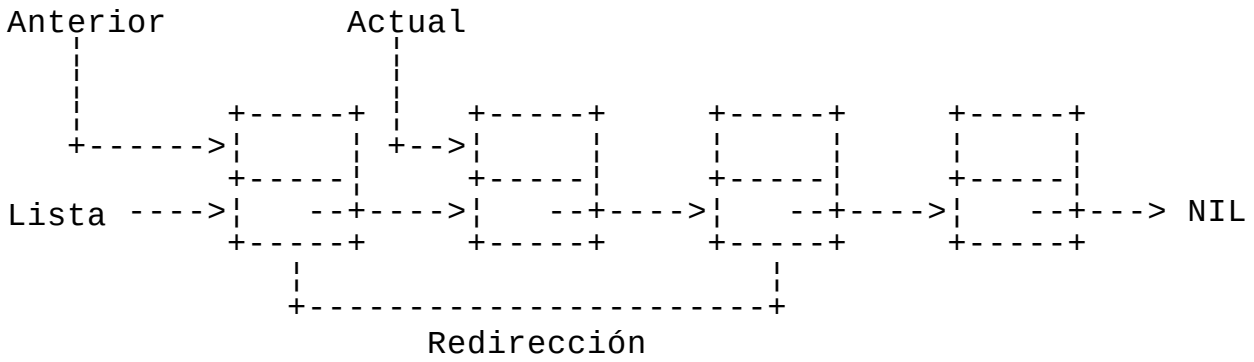
En este caso el primer elemento de la cola es el e3, observese que en esta realización ocupa la componente siguiente a la apuntada por **Cola**. Precisamente la componente apuntada por **Cola** es la correspondiente a e1, que es el último elemento de la cola.

5.LISTA CALIFICADA NO ORDENADA

La inserción en una lista calificada no ordenada se puede realizar antes o después.Si insertamos después reconocemos la lista preguntando por la clave. Si insertamos antes hay que utilizar dos punteros porque si preguntamos por la clave perdemos el puntero que apunta a esa clave.

5.1 Borrado

Para borrar hay que recorrer la lista con dos punteros porque sino, al encontrar al elemento a borrar, perdemos la información del anterior, que es el que hay que redirigir.



5.2 Inserta después

Si no se encuentra la clave el el procedimiento de búsqueda no insertamos el nuevo elemento.

```
PROCEDURE Inserta_Despues (VAR Lista:Tlista;ClaveBus,ClaveEle:Tclave;
                                    InfoEle:Tinfo);
VAR  Aux,Nuevo:Tpuntero;
     Encontrado:BOOLEAN;
BEGIN
   Encon:=FALSE;
   Aux:=Lista;
   INTEGER (Aux<>NIL) and (NOT Encontrado) do
       IF Aux^.Clave=ClaveBus
             THEN Encontrado:=TRUE
             ELSE Aux:=Aux^.Sig;
   IF Encontrado
         THEN BEGIN
                 NEW (nuevo);
                 Nuevo^.Clave:=ClaveEle;
                 Nuevo^.Info:=InfoEle;
                 Nuevo^.Sig:=Aux^.Sig;
                 Aux^.Sig:=Nuevo
              END
END;
```

5.3 Inserta antes

```
PROCEDURE Inserta_Antes (VAR Lista:Tlista;ClaveBus,ClaveEle:Tclave;
                                  InfoEle:Tinfo);
VAR
     Anterior,Actual,Nuevo:Tlista;
     Encontrado:BOOLEAN;
BEGIN
   Encontrado:=FALSE;
   Anterior:=Lista;
   Actual:=Lista;
   INTEGER Actual<>NIL and NOT Encontrado do
   BEGIN
      IF Actual^.Clave=ClaveBus
                THEN Encontrado:=TRUE
                ELSE BEGIN
                        Anterior:=Actual;
                        Actual:=Actual^.Sig
                     END
   END;
   IF Encontrado
        THEN BEGIN
                NEW(Nuevo);
                Nuevo^.Info:=InfoEle;
                Nuevo^.Clave:=ClaveEle;
                IF Lista=Actual
                        THEN BEGIN
                                Nuevo^.Sig:=Lista;
                                Lista:=Nuevo
                             END
                        ELSE BEGIN
                                Nuevo^.Sig:=Actual;
                                Anterior^.Sig:=Nuevo;
                             END
             END
END;
```

Las instrucciones **Nuevo^.Sig:=Lista;** y **Nuevo^.Sig:=Actual;** se pueden poner fuera de la sentencia IF-ELSE porque en el caso de insertar al principio se va a dar que **Actual** = **Siguiente** = **Lista**.

5.4 Recorridos con un solo puntero

Siempre pueden usar mas punteros para hacer NEW o DISPOSE, pero solo uno para recorrer la lista. La única forma de hacerlo es recursivamente, y recorrer la lista con el propio puntero que la ha creado, con el puntero interno.

```
PROCEDURE Buscar (Lista:Tlista;ClaveBus:Tclave;VAR Info:Tinfo);
BEGIN
  IF Lista<>NIL THEN (* Parar si llega a NIL *)
    IF Lista^.Clave>=ClaveBus THEN (* Por si se ha pasado a un
                                      elemento de mayor clave *)
      IF Lista^.Clave=ClaveBus THEN
        Info:=Lista^.Info
      ELSE
        Busca (Lista^.Sig,ClaveBus,Info)
END;
```

Vamos a terminar cuando lleguemos a NIL ya que insertamos al final.Encontramos el hueco si la clave del siguiente elemento es mayor que la clave que tenemos que insertar.

```
PROCEDURE Insertar(VAR Lista:Tlista;ClaveEle:Tclave;InfoEle:Tinfo);
VAR
  Encontrado:BOOLEAN;
  Nuevo:Tlista;
BEGIN
  Encontrado:=FALSE;
  IF Lista=NIL THEN
    Encontrado:=TRUE
  ELSE
    IF Lista^.Clave>ClaveEle THEN
      Encontrado:=TRUE
    ELSE
      IF Lista^.Clave=ClaveEle THEN
        (* Ya está insertado *)
      ELSE
        Insertar(Lista^.sig,ClaveEle,InfoEle);
  IF Encontrado THEN
    BEGIN
      NEW(Nuevo);
      Nuevo^.Clave:=ClaveEle;
      Nuevo^.Info:=InfoEle;
      Nuevo^.Sig:=Lista;
      Lista:=Nuevo
  END
END;
```

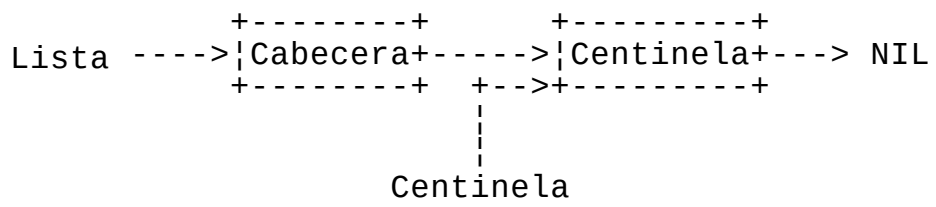
Nuevo^.Sig apuntará al valor de Lista porque ese valor es el puntero al siguiente elemento después de donde tenemos que insertar el nuevo.

```
PROCEDURE Borrar (VAR Lista:Tlista;ClaveBus:Tclave);
VAR Aux:Tlista;
BEGIN
  IF Lista=NIL THEN
    "No esta"
  ELSE
    IF Lista^.Clave<ClaveBus THEN
      Borrar(Lista^.Sig,ClaveBus)
    ELSE
      IF Lista^.Clave=ClaveBus THEN
        BEGIN
          Aux:=Lista;
          Lista:=Lista^.Sig;
          DISPOSE(Aux);
        END
END;
```

No hace falta hacer NEW(Aux) para hacer Aux:=NIL.

5.5 Listas con cabecera ficticia y centinela

El siguiente esquema representa una lista con cabecera fictícia y centinela vacía.En este tipo de listas se realiza más fácilmente la inserción antes y, en el caso del centinela, también es útil para las búsquedas.



Recorridos normales con un solo puntero

```
INTEGER Aux<>NIL and NOT Encontrado
     IF Aux^.Clave=Clave
          THEN Encontrado:=TRUE
          ELSE Aux:=Aux^.Sig;
```

Recorridos con centinela

Lo que hacemos primero es meter la clave que buscamos en el centinela para que el puntero nunca pueda llegar a NIL y seguimos el siguiente método:

```
     INTEGER Act^.Clave<>Clave do
     BEGIN
     ...
     ...
     END;
```

Búsqueda, inserción y borrado

```
PROCEDURE      Buscar      (Centinela,Lista:Tlista;ClaveBus:Tclave;VAR
Info:Tinfo);
VAR Aux:Tlista;
BEGIN
   Centinela^.Clave:=ClaveBus;
   Aux:=Lista^.Sig;              (* Para saltar la cabecera ficticia *)
   INTEGER Aux^.Clave<ClaveBus do
        Aux:=Aux^.Sig;
   IF Aux<>Centinela and Aux^.Clave=ClaveBus
         THEN Info:=Aux^.Info;
    (* En cualquier otro caso no hemos encontrado la clave *)
END;


PROCEDURE Crear (VAR Lista,Centinela:Tlista);     (* Las 2 por VAR *)
BEGIN
   NEW(Lista);
   NEW(Centinela);
   Lista^.Sig:=Centinela;
   Centinela^.Sig:=NIL
END;
```

```
PROCEDURE Insertar (VAR Lista:Tlista;Centinela:Tlista;ClaveEle:Tclave;
                                        Info:Tinfo);
VAR Nuevo,Actual,Anterior:Tlista;
BEGIN
   Centinela^.Clave:=ClavEle;

(* Aunque la lista este vacía tendría dos elementos por lo tanto
podríamos hacer la asignación anterior *)

   Ant:=Lista;
   Actual:=Lista^.Sig;
   INTEGER Actual^.Clave<ClaveEle do
   BEGIN
      Anterior:=Actual;
      Actual:=Actual^.Sig
   END;
   IF Actual^.Clave=ClaveEle and Actual<>Centinela
        THEN "Elemento ya esta"
        ELSE BEGIN

     (* Si es la misma clave pero apuntamos al centinela, tendremos que
             insertar porque el nuevo elemento no estará en la lista *)

                Nuevo^.Info:=Info;
                Nuevo^.Clave:=ClaveEle;
                Nuevo^.Sig:=Actual;
                Anterior^.Sig:=Nuevo
             END
END;


PROCEDURE Borrado (VAR Lista:Tlista;Centinela:Tlista;ClaveEle:Tclave);
VAR Actual,Auxiliar:Tlista;
BEGIN
   Centinela^.Clave:=ClaveEle;
   Actual:=Lista;
   INTEGER Actual^.Sig^.Clave<ClaveEle do
          Actual:=Actual^.Sig;
   IF Actual^.Sig^.Clave=ClaveEle and Actual^.Sig<>Centinela

(* Se da la primera condición si la clave que buscamos que esta en la
lista o llegamos al centinela.Tenemos que insertar si cuando nos
paramos no apuntamos al centinela *)

        THEN BEGIN
                Aux:=Actual^.Sig;
                Actual^.Sig:=Aux^.Sig;
                DISPOSE(Aux)
             END
END;
```

6 OTROS TIPOS DE LISTAS

6.1 Listas reorganizables

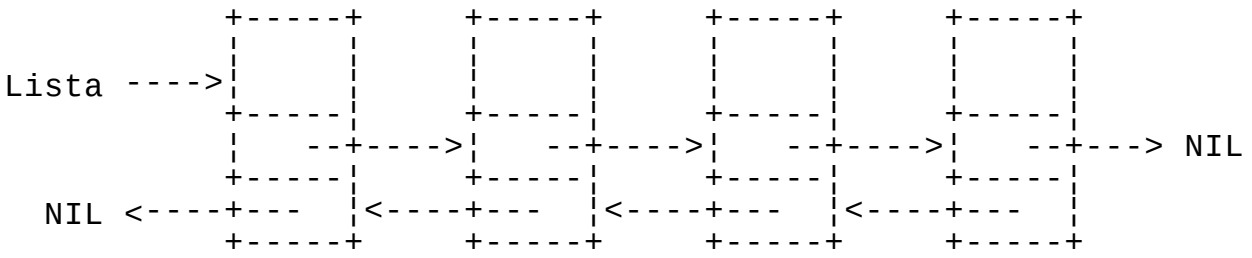
Son aquellas listas en las que el último elemento consultado se sitúa al principio.

6.2 Listas circulares

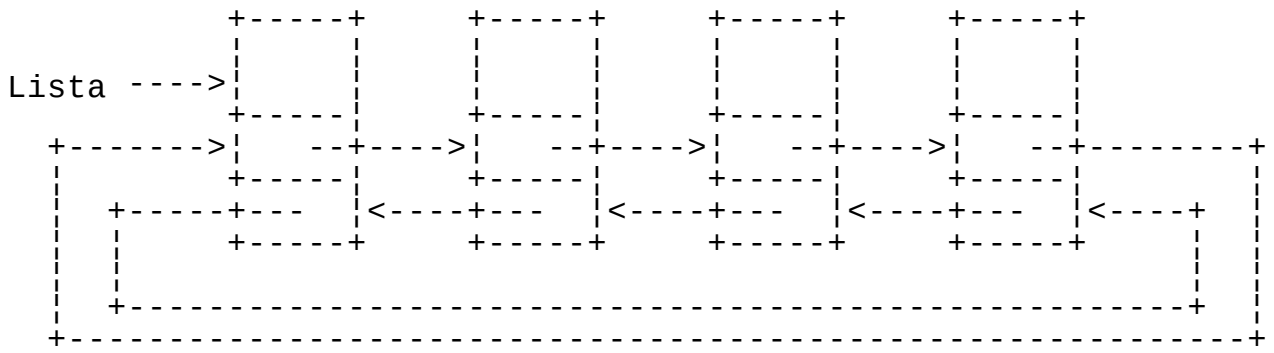
En ellas el último elemento apunta al primero.

6.3 Listas doblemente enlazadas

Cada elemento tiene dos punteros, uno de los cuales apunta al elemento siguiente y otro al anterior.



6.4 Listas circulares doblemente enlazadas



Ejercicio 1

Dada la siguiente declaración de tipo lista circular ordenada.

```
TYPE
   Tlista:^Elemento;
    Elemento=RECORD
              Clave:INTEGER;
              Info:Tinfo;
              Sig:Tlista
             END;
```

Codificar un algoritmo que realice la inserción de un elemento, representando por una clave y una información, en una lista circular como la anteriormente descrita.

Para eliminar el caso de insertar antes del primer elemento vamos a utilizar una lista con cabecera ficticia.Todo queda reducido a insertar en cualquier lugar o si la lista está vacía.

```
PROCEDURE Introducir (VAR Lista:Tlista;Eclave:INTEGER;Einfo:Tinfo);
VAR Nuevo,Ant,Act:Tlista;
                (* Para insertar antes necesitamos dos punteros *)
BEGIN
  IF lista=NIL THEN
    BEGIN
      NEW(Lista);          (* Crea la cabecera ficticia *)
      NEW(Lista^.Sig);
      Lista^.Sig^.Clave:=Eclave;
      Lista^.Sig^.Info:=Einfo;
      Lista^.Sig^.Sig:=Lista
    END
  ELSE
    BEGIN
      Ant:=Lista;
      Act:=Lista^.Sig;   (* Al primer elemento de la lista *)
      INTEGER act<>lista   (* Mientras no apunte al último elemento *)
          and Act^.Clave<Eclave do
        BEGIN
          Ant:=Act;
          Act:=Act^.Sig
        END;
      NEW(Nuevo);
      Nuevo^.Info:=Einfo;
      Nuevo^.Clave:=Eclave;
      Nuevo^.Sig:=Act;
      Anterior^.Sig:=Act;
      Anterior^.Sig:=Nuevo
    END
END;
```

Ejercicio 2

Dada la siguiente declaración de lista:

```
TYPE
   Tlista:^Artículo;
   Articulo=RECORD
               Codigo,Cont_Pedida:INTEGER;
               Sig:Tlista
            END;
```

Se tiene una lista en la que se han introducido pedidos según el orden de llegada, por lo que puede haber pedidos del mismo artículo.Se pide, codificar un procedimiento que, dada una lista como la anteriormente descrita, devuelva un único elemento por cada artículo, en el cual el campo **Cant_Pedida** tenga la suma de todas las cantidades pedidas de ese artículo.

No se pueden utilizar estructuras auxiliares, así como tampoco estructuradas de control como INTEGER, for o REPEAT.

```
PROCEDURE Eli_Codigo(VAR Lista:Tlista);

PROCEDURE Elimina(VAR L:Tlista;Cod:INTEGER;VAR Suma:INTEGER);
VAR Aux:Tlista;
BEGIN
   IF L<>NIL
        THEN IF L^.Codigo=Cod
               THEN BEGIN
                       Aux:=L;
                       Suma:=Suma+L^.Cant_Pedida;
                       L:=L^.Sig;
                       DISPOSE(Aux);
                       Elimina(L,Cod,Suma)
                    END
               ELSE Elimina(L^.Sig,Cod,Suma)
END;


BEGIN
  IF Lista<>NIL
       THEN BEGIN
               Elimina(Lista^.Sig,Lista^.Codigo,Lista^.Cant_Pedida);
               Eli_Codigo(Lista^.Sig)
            END
END;
```

14 Septiembre 90.Ejercicio 1.

× En el bucle INTEGER sobra el igual, porque sino se salta el centinela.

× En el IF es Lista^.Sig porque cuando se sale del INTEGER Lista nunca va a apuntar al centinela.

× En el THEN debe poner:

```
           THEN BEGIN
                  ...
                          Lista^.Sig:=Aux
                          END
```

Ejercicio 3

Utilizando el TAD pila pasar de la notación normal a la notación polaca inversa.Ejemplos: (AB+), (AB+C*).El símbolo '#' va a ser el principio y el final de la formula.

1. Símbolo que llega introducimos en la pila.

2. El primer elemento en la pila se saca y se intenta introducir el signo que entra.

3. Si se dan dos paréntesis complementarios se eliminan los dos.

4. Llega el final de la formula (#).

Contenido	Entradas: #	+	-	*	/	(	)
#	4	1	1	1	1	1	E
+	2	2	2	1	1	1	2
-	2	2	2	1	1	1	2
*	2	2	2	2	2	1	2
/	2	2	2	2	2	1	2
(	E	1	1	1	1	1	3

28 Junio 91.Ejercicio 1.

Una solución podría ser:

```
IF Lista^.Sig<>NIL
     THEN BEGIN
             Aux:=Lista^.Ant;
             Lista^.Ant:=Lista^.Sig;
             Lista^.Sig:=Aux;
             Invertir(Lista^.Ant)
          END
```

Aunque es mejor llamar recursivamente hasta que se encuentre NIL, entonces pone **Lista** apuntando al último elemento y a la vuelta de la recursividad va haciendo el intercambio de **Sig** y **Ant**.

14 Septiembre 90.Ejercicio 2.

A la ida de la recursividad borramos el elemento siguiente a la clave dada, y en la vuelta, en la llamada **n-1** borramos el anterior.Para saber que estamos situados sobre el anterior utilizamos la variable de tipo BOOLEAN llamada **EsClave**, que se pondrá a TRUE cuando encuentre el elemento, y por tanto se tiene que pasar por VAR.

```
PROCEDURE Eliminar (VAR Lista:Tlista;Clave:Tclave;VAR EsClave:BOOLEAN);
VAR
    Aux:Tlista;
BEGIN
  IF Lista=NIL
       THEN EsClave:=FLASE
       ELSE IF Lista^.Clave=Clave
                THEN BEGIN
                        EsClave:=TRUE;
                        IF Lista^.Sig<>NIL
                              THEN BEGIN
                                      Aux:=Lista^.Sig;
                                      Lista^.Sig:=Aux^.Sig;
                                      DISPOSE(Aux)
                     END
                ELSE BEGIN
                        Eliminar(Lista^.Sig,Clave,EsClave);
                          IF EsClave   (* Ya hemos borrado el siguiente
*)
                             THEN BEGIN
                                     Aux:=Lista;
                                     Lista:=Aux^.Sig;
                                     EsClave:=FALSE
                                (* Para no borrar los demás *)
                                  END
                     END
END;
```

Si la clave que buscamos es el primer elemento de la lista, como no hay ninguna llamada recursiva, entonces borra el siguiente, pero no el anterior.

28 Junio 90.Ejercicio 2.

Se recorrerá la lista con un único puntero, y solo se puede pasar una vez.Las variables **Ntotal** y **Actual** indicarán el número total de elementos de la lista y la posición actual en donde estamos colocados (por eso no es por VAR) respectivamente.En la primera llamada **Actual** valdrá 1.

Tendremos que parar en la vuelta atrás de la recursividad cuando **Actual** tenga el valor **Ntotal** DIV 2.Luego, si **Ntotal** es impar tenemor que hacer una copia del emento para L1, y si es par solo hay que dividir la lista.

```
PROCEDURE Dividir (L1:Tlista; VAR L2:Tlista; VAR Ntotal:INTEGER;
Actual:INTEGER);
BEGIN
  IF L1=NIL THEN
    L2:=NIL
  ELSE IF L1^.Sig=NIL THEN
    BEGIN
      Ntotal:=Actual;
      IF Ntotal=1 THEN
        BEGIN  (* Lista impar con un solo elemento *)
          NEW(L2);
          L2^.Clave:=L1^.Clave;
          L2^.Sig:=NIL
        END
    END
  ELSE
    BEGIN
      Dividir(L1^.Sig,L2,Ntotal,Actual+1);
      IF Actual=Ntotal DIV 2 THEN
        IF ODD(Ntotal) THEN
          BEGIN
            L2:=L1^.Sig;
            NEW(L1^.Sig);
            L1^.Sig^.Clave:=L2^.Clave;
            L1^.Sig^.Sig:=NIL
          END
        ELSE
          BEGIN
            L2:=L1^.Sig;
            L1^.Sig:=NIL
          END
    END
END;
```

4 Septiembre 89.Ejercicio 2.

```
PROCEDURE Mezcla (VAR L3:Tlista;L1,L2:Tlista);
VAR Menor:1..2;
BEGIN
   IF L1<>NIL OR L2<>NIL
         THEN BEGIN
                 IF L1=NIL
                      THEN Menor:=2  (* Indica donde insertaremos *)
                      ELSE IF L2=NIL
                               THEN Menor:=1
                               ELSE IF L1^.Clave>L2^.Clave
                                       THEN Menor:=2
                                       ELSE Menor:=1;
              NEW(L3);
              IF Menor=1
                   THEN BEGIN
                           L3^.Clave:=L1^.Clave;
                           Mezcla(L3^.Sig,L1^.Sig,L2)
             (* Porque hemos metido el elemento de la lista 1 *)
                        END
                   ELSE BEGIN
                           L3^.Clave:=L2^.Clave;
                           Mezcla(L3^.Sig,L1,L2^.Sig)
                        END
              END
         ELSE L3:=NIL    (* Los dos a NIL luego hemos terminado *)
END;
```

1. DEFINICIONES.

Varias definiciones:

× Un arbol es una lista en la que cada uno de sus elementos apunta a uno, ninguno o varios elementos del mismo tipo.

× Un arbol es una estructura dinámica y homogénea, por tanto está copuesto de elementos del mismto tipo base T, de forma que la estructura puede estar vacía o compuesta por un elemento del tipo base T del que cuelgan un número finito de estructuras arbol similar, a las que llamaremos subarboles, y entre si son disjuntos.

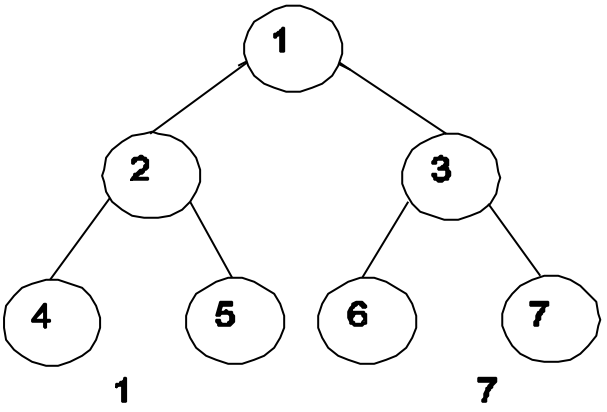
Dos arboles son iguales cuando tengan igual número de nodos, igual contenido en ellos y, además, estén dispuestos de igual manera.

2. REPRESENTACION.

2.1 Matriz de adyacencia.

Es un array [1..n,1..n] OF BOOLEAN donde n es el número de nodos que tiene el árbol y cada posición de la matriz indicará si existe un enlace entre dos nodos.Esto es normal hacerlo en lenguajes en que no pueden crearse componentes dinámicamente, y referenciarles por medio de punteros.

Ejemplo:

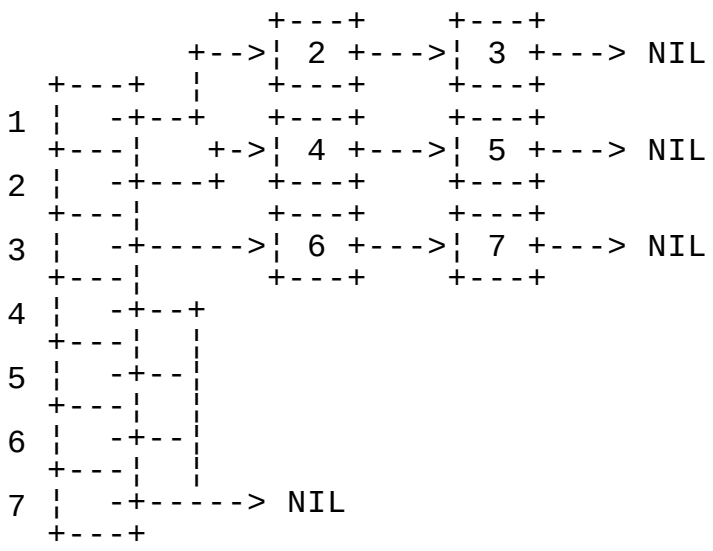


	1	2	3	4	5	6	7
1	F	**T**	**T**	F	F	F	F
2	F	F	F	**T**	**T**	F	F
3	F	F	F	F	F	**T**	**T**
4	F	F	F	F	F	F	F
5	F	F	F	F	F	F	F
6	F	F	F	F	F	F	F
7	F	F	F	F	F	F	F

2.2 Lista de adyacencia.

Es una tabla de 1 a n, siendo n el número de nodos, donde cada elemento de la tabla representa cada uno de los nodos del arbol y de cada uno de los elementos sale un puntero a una lista que está formada por todos los nodos que están enlazados con él.

Ejemplo:



2.3 Estructura dinámica pura.

En ella, todos los componentes o nodos, se representan por punteros explicitos.Vamos a tener un tipo como:

```
Tarbol=^Elemento;
Elemento=RECORD
            Info:Tinfo;
            Clave:Tclave;
            Enlace1,Enlace2, ..., Enlacen:Tarbol
         END;
```

3. CONCEPTOS ASOCIADOS.

× **Nodo**: Se denomina nodo a cada uno de los elementos de un árbol.

× **Rama**: Una rama es cada uno de los enlaces que existe entre los nodos de un árbol.

× **Antecesor o padre**: Es un nodo del que cuelga algún otro, llamado descendiente o hijo.

× **Descendiente directo o antecesor directo**: Un nodo es descendiente directo de otro, llamado antecesor directo si se puede acceder a el recorriendo únicamente una rama.

× **Raiz**: La raiz de un árbol es aquel nodo que no tiene antecesores, es decir, todos son descendientes directos o indirectos de el.

× **Nivel, profundidad, altura, longitud o longitud de camino de un nodo**: Es el número de ramas que hay que recorrer para llegar a él desde la raiz, tendiendo en cuenta que la raiz tiene nivel uno.

× **Subarbol**: Se llama subarbol de raiz m al conjunto de nodos que dependen directa o indirectamente de él, asi como al propio m.

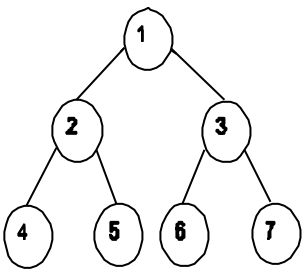
× **Grado de un nodo**: Es el número de descendientes directos que tiene.El grado de un arbol es igual al del nodo con mayor grado.

× **Nodo terminal u hoja**: Es aquel que tiene grado cero, es decir, que no tiene ningún descendiente.

× **Nodo interno**: Es aquel que no es hoja.

× **Longitud de camino interno de un arbol**: Es la suma de las longitudes de camino de todos sus nodos.

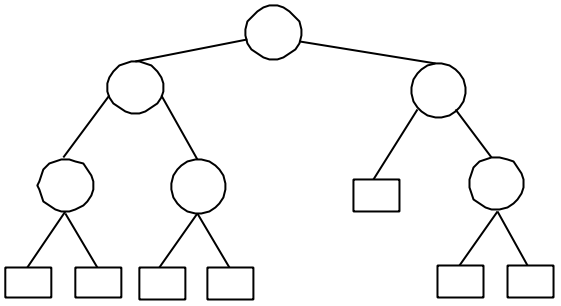
LCI = 1+2+2+3+3+3 = 14



× **Longitud de camino externa**: Es la suma de las longitudes de camino de todos los nodos especiales.

× **Nodo especial**: Es aquel que se hace colgar de aquellos nodos que no tienen completo el cupo de descendientes.

En el ejemplo anterior, los nodos 1,2,3 y 4 no tienen completo el número de descendientes.A la operación de colocar nodos especiales en un arbol se le llama **expandir el arbol**.

LCE = 3+4+4+4+4+4 =  27

× **Longitud de camino interna media**: Es la longitud de camino interna partido del número de nodos.

4. ARBOLES BINARIOS.

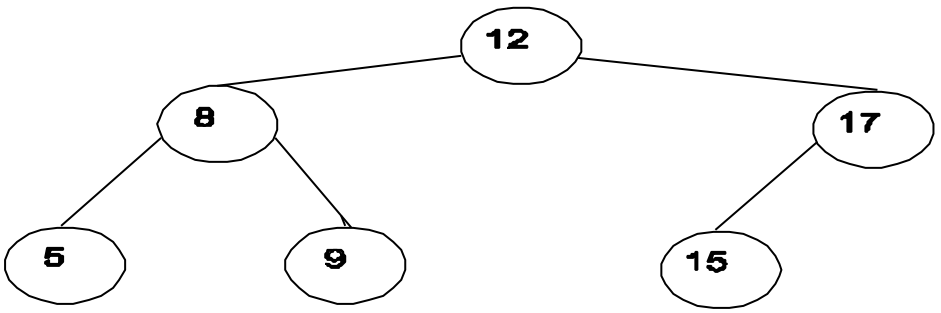
Los árboles de grado 2 tienen una especial importancia.Se les conoce con el nombre de árboles binarios.Se define un árbol binario como un conjunto finito de elementos (nodos) que bien está vacio o está formado por una raiz con dos árboles binarios disjuntos, llamados subárbol izquierdo y derecho de la raiz.

En los apartados que siguen se considerarán únicamente árboles binarios y, por lo tanto, se utilizará la palabra árbol para referirse a árbol binario.Los árboles de grado suparior a 2 reciben el nombre de árboles multicamino.

4.1 Arbol binario de búsqueda.

Los árboles binarios se utilizan frecuentemente para representar conjuntos de datos cuyos elementos se identifican por una clave única.Si el árbol está organizado de tal manera que la clave de cada nodo es mayor que todas las claves su su subarbol izquierdo, y menor que todas las claves del subarbol derecho se dice que este árbol es un árbol binario de búsqueda.

Ejemplo:



4.2 Operaciones básicas

Una tarea muy común a realizar con un árbol es ejecutar una determinada operación con cada uno de los elementos del árbol.Esta operación se considera entonces como un parámetro de una taré más general que es la visita de todos los nodos o, como se denomina usualmente, del recorrido del árbol.

Si se considera la tarea como un proceso secuencial, entonces los nodos individuales se visitan en un orden específico, y pueden considerarse como organizados según una estructura lineal.De hecho, se simplifica considerablemente la descripción de muchos algoritmos si puede hablarse del proceso del siguiente elemento en el árbol, según un cierto orden subyacente.

Hay dos formas básicas de recorrer un árbol: El recorrido en amplitud y el recorrido en profundidad.

- **Recorrido en amplitud**

Es aquel recorrido que recorre el árbol por niveles, en el último ejemplo sería:

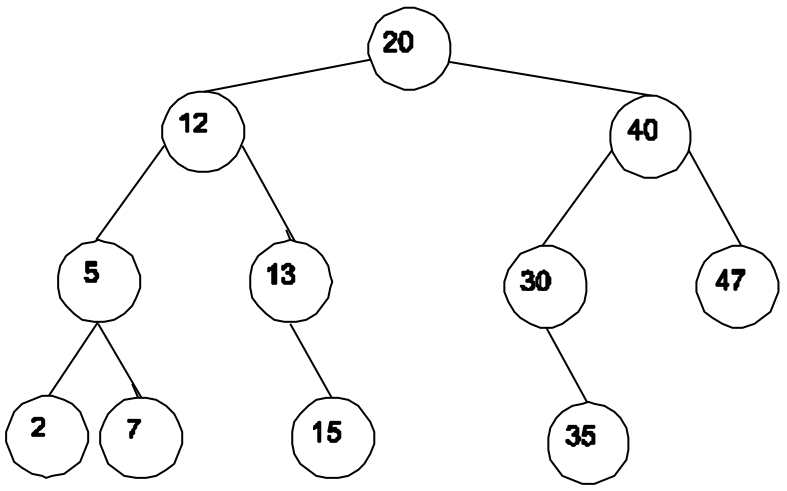
12 - 8,17 - 5,9,15

- **Recorrido en profundidad**

Recorre el árbol por subárboles.Hay tres formas: Preorden, orden central y postorden.

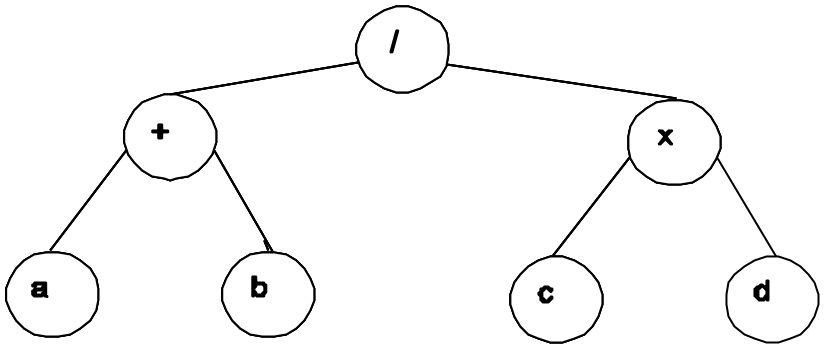
- **Preorden**: Raiz, subarbol izquierdo y subarbol derecho.
- **Orden central**: Subarbol izquierdo, raiz, subarbol derecho.
- **Postorden**: Subarbol izquierdo, subarbol derecho, raiz.

Ejemplo:



Preorden: 20 - 12 - 5 - 2 - 7 - 13 - 15 - 40 - 30 - 35 - 47
Orden central: 2 - 5 - 7 - 12 - 13 - 15 - 20 - 30 - 35 - 40 - 47
Postorden: 2 - 7 - 5 - 15 - 13 - 12 - 35 - 30 - 47 - 40 - 20

Ejemplo:



Preorden: / + a b * c d Notación polaca
Orden central: a + b / c * d Notación infija
Postorden: a b + c d * / Notación polaca inversa

Implementación de los recorridos.

A continuación se formulan los tres métodos de recorrido por medio de tres programas concretos.Nótese que el puntero *Arbol* es un parámetro constante (se pasa por valor).Esto expresa el hecho de que lo que interesa es la referencia al subárbol considerado y no la variable cuyo valor es el puntero, que podría ser alterada en caso de pasarlo como parámetro variable.

```
TYPE
   Tarbol=^Elemento;
   Elemento=RECORD
               Info:Tinfo;
               Clave:Tclave;
               izq,der:Tarbol
            END;


PROCEDURE PreOrden (Raiz:Tarbol);
BEGIN
   IF Raiz<>NIL
         THEN BEGIN
                 ProcesaNodo;
                 PreOrden(Raiz^.izq);
                 PreOrden(Raiz^.der)
              END
END;
```

```
PROCEDURE OrdenCentral (Raiz:Tarbol);
BEGIN
   IF Raiz<>NIL
         THEN BEGIN
                 OrdenCentral(Raiz^.izq);
                 ProcesaNodo;
                 OrdenCentral(Raiz^.der)
              END
END;



PROCEDURE PostOrden (Raiz:Tarbol);
BEGIN
   IF Raiz<>NIL
         THEN BEGIN
                 PostOrden(Raiz^.izq);
                 PostOrden(Raiz^.der);
                 ProcesaNodo
              END
END;
```

4.3 Busqueda de un nodo

En un árbol binario de búsqueda puede localizarse una clave arbitraria empezando por la raiz, y avanzando por un camino de búsqueda de forma que la decisión de continuar por el subarbol izquierdo o derecho de un nodo dado se toma en base únicamente al valor de la clave de dicho nodo.

Tal como ocurrió en el caso de búsqueda en listas (En el caso de recorrido iterativo), la complejidad de la condición de terminación del bucle sugiere buscar una solución mejor.Esta consiste en el uso de un centinela al final de la lista.Puede aplicarse la misma idea al caso de un árbol.El uso de punteros permite que todas las ramas del árbol acaben con el mismo, identico, centinela.

La estructura resultante no es ya un árbol, sino más bien un árbol con todas las hojas atadas a un punto de anclaje común, único.El centinela puede ser considerado como un representante común de todos los nodos externos con los que se extendría el árbol original.

```
FUNCTION Buscar (Arbol:Tarbol;Clave:Tclave):Tinfo;
BEGIN
   IF Arbol=NIL
       THEN BEGIN
               (* No está *)
               Buscar:=0
            END
       ELSE IF Arbol^.Clave=Clave
                  THEN Buscar:=Arbol^.Info
                  ELSE IF Arbol^.Clave>Clave
                             THEN Buscar:=Buscar(Arbol^.izq,Clave)
                             ELSE Buscar:=Buscar(Arbol^.der,Clave)
END;
```

La función *Localizar* es una versión iterativa de la funcíon de búsqueda de un nodo en un árbol binario de búsqueda.

```
FUNCTION Localizar(Arbol:Tarbol;Clave:Tclave):Tinfo;
VAR
   Encontrado:BOOLEAN;
BEGIN
   Encontrado:=FALSE;
   WHILE (Arbol<>NIL) AND NOT Encontrado DO
      IF Arbol^.Clave=Clave
           THEN Encontrado:=TRUE
           ELSE IF Arbol^.Clave>Clave
                     THEN Arbol:=Arbol^.Izq
                     ELSE Arbol:=Arbol^.Der;
   Localizar:=Arbol^.Info
END;
```

4.4 Borrado

La tarea consiste en borrar, es decir, sacar el nodo con clave x de un árbol que tiene las claves ordenadas.Borrar un nodo es fácil si el elemento a borrar es una hoja o solo tiene un descendiente.

- Si hay que borrar una hoja, se borra con DISPOSE y se pone el padre a NIL.

- Para borrar un nodo con un descendiente, la variable arbol, en vez de apuntar al nodo a borrar, se situa el padre apuntando al descendiente y se borra el nodo.

La dificultad está en el borrado de un elemento con dos descendientes, pues no puede señalarse en dos direcciones con un solo puntero.

- Para borrar un nodo con dos descendientes,no vamos a borrar el nodo, sino que subimos un nodo hoja y lo borramos como hoja.Tenemos dos posibilidades: Subir el nodo hoja más a la derecha del subarbol izquierdo, o el más a la izquierda del subarbol derecho.

Los detalles del proceso se muestran en el procedimiento recursivo llamado *Borrar*.Este procedimiento distingue tres casos:

1. No hay ningún nodo con la clave buscada.

2. El componente con la clave buscada tiene un único descendiente como máximo.

3. El componente con la clave buscada tiene dos descentientes.

El procedimiento auxiliar recursivo *SubDer* se activa solo en tercer caso.Entonces desciente a lo largo de la rama más a la derecha del subárbol izquierdo del elemento a borrar, y reemplaza la información de interés.

```
PROCEDURE Borrar (VAR Arbol:Tarbol;Clave:Tclave);
VAR
   Aux:Tarbol;



PROCEDURE SubDer(VAR Puntero:Tarbol);
BEGIN
   IF Puntero^.der<>NIL
         THEN Subder(Puntero^.der)
         ELSE BEGIN
                 Arbol^.Clave:=Puntero^.Clave;
                 Arbol^.Info:=puntero^.Info;
                 Aux:=Puntero;
                 Puntero:=Puntero^.izq
              END
END;
BEGIN
  IF Arbol=NIL
     THEN "NO ESTA"
     ELSE BEGIN
             IF Arbol^.Clave>Clave
                THEN Borrar (Arbol^.izq,Clave)
                ELSE IF Arbol^.Clave<Clave
                        THEN Borrar(Arbol^.der,Clave)
                        ELSE IF Arbol^.izq=NIL AND Arbol^.der=NIL
                                THEN BEGIN     (* Nodo hoja *)
                                        Aux:=Arbol;
                                        Arbol:=NIL
                                     END
                                ELSE IF Arbol^.der=NIL
                                        THEN BEGIN  (* Solo hoja izq *)
                                                Aux:=Arbol;
                                                Arbol:=Arbol^.izq
                                             END
                                        ELSE IF Arbol^.izq=NIL
                                               THEN BEGIN
                                                       Aux:=Arbol;
                                                       Arbol:=Arbol^der
                                                    END
                                               ELSE
Subeder(Arbol^.izq);
             DISPOSE(Aux)
          END
END;
```

4.5 Inserción

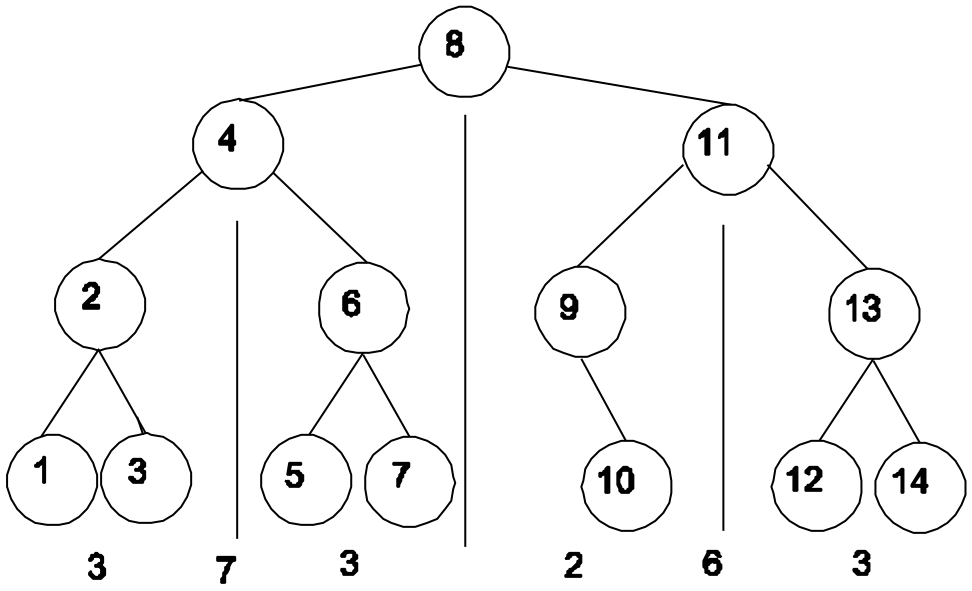
Primero se realiza la búsqueda de la clave a insertar.Si este recorrido lleva a un puntero que sea igual a NIL (subarbol vacio), entonces se realiza la inserción ya que solo se pueden añadir nodos en las hojas, se inserta al final.

```
PROCEDURE Insertar (VAR Arbol:Tarbol;Clave:Tclave;Info:Tinfo);
BEGIN
   IF Arbol=NIL      (* Ha llegado al sitio donde hay que insertar *)
        THEN BEGIN
                NEW (Arbol);
                Arbol^.izq:=NIL;
                Arbol^.der:=NIL;
                Arbol^.Clave:=Clave;
                Arbol^.Info:=Info
             END
        ELSE IF Arbol^.Clave>Clave
                   THEN Insertar (Arbol^.izq,clave,info)
                   ELSE IF Arbol^.Clave<Clave
                             THEN Insertar (Arbol^.der,clave,info)
                             ELSE   (* ERROR *)
END;
```

El parámetro *Arbol* es variable y no uno constante.Esto es esencial porque cuando se presenta el caso de inserción se debe asignar un nuevo valor a la variable que contenia el valor NIL.

5. ARBOLES PREFECTAMENTE EQUILIBRADOS.

Se usan especialmente cuando se insertan los elementos de un fichero ordenado con los métodos de arboles binarios de búsqueda.Para que un arbol no dejenere en una lista entra el concepto de equilibrio.Hay varias formas.Un arbos está perfectamente equilibrado cuando situados en cualquiera de sus nodos, el número de nodos del subarbol izquierdo y el derecho difieren como mucho en uno.Ejemplo:



Ventajas:

La búsqueda es más rápida que en cualquier otro tipo.Cada vez que hacemos una comparación eliminamos la mitad de elementos, porque están equilibrados.Es similar a la búsqueda dicotómica en tablas.El peor de los casos es que el elemento no esté, y en ese caso la complejidad es:

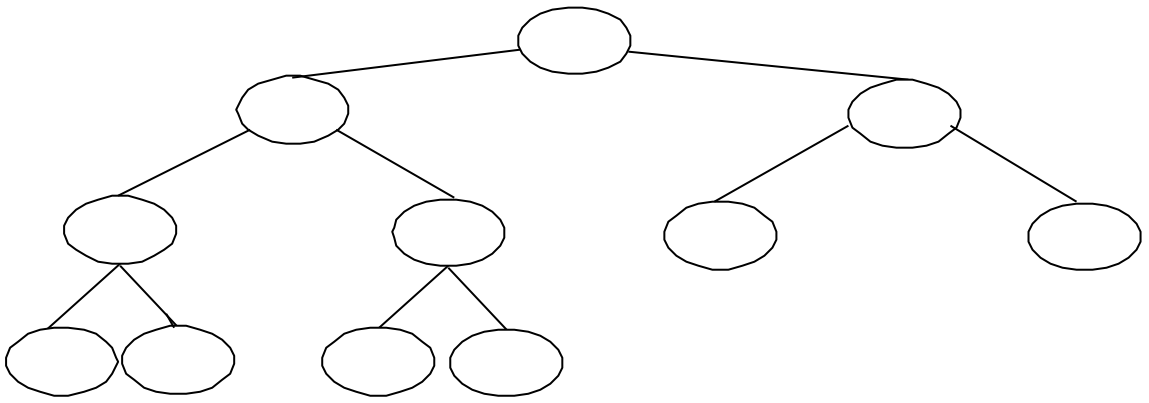
$$Log_2\ n$$

Inconvenientes:

Hay que conocer el número de nodos para saber cuantos irán a la derecha y cuantos a la izquierda.Esto choca con el concepto de arbol (dinamismo), por tanto no se utilizan mucho.

5.1 Arboles equilibrados o AVL

Es una suavización de las restricciones para formar arboles perfectamente equilibrados.Un arbol es AVL cuando situados en cualquiera de sus nodos, la altura del subarbol derecho y del izquierdo difieren como mucho en uno.Ejemplo:



La definición no sólo es simple, sino que además conduce a un procedimiento de reequilibrado relativamente sencillo, y a una longitud de camino media prácticamente idéntica a la del árbol perfectamente equilibrado.

En un árbol AVL, se pueden realizar en O(lon n) unidades de tiempo, incluso en el peor de los casos, las siguientes operaciones:

1. Encontrar un nodo con una clave dada.
2. Insertar un nodo con una clave dada.
3. Borrar un nodo con una clave dada.

Vamos a añadir un campo nuevo a la declaración del tipo Tarbol, que se llamará equilibrio (equ), que va a variar entre -1, 0 y 1.

Condiciones para variar equ:

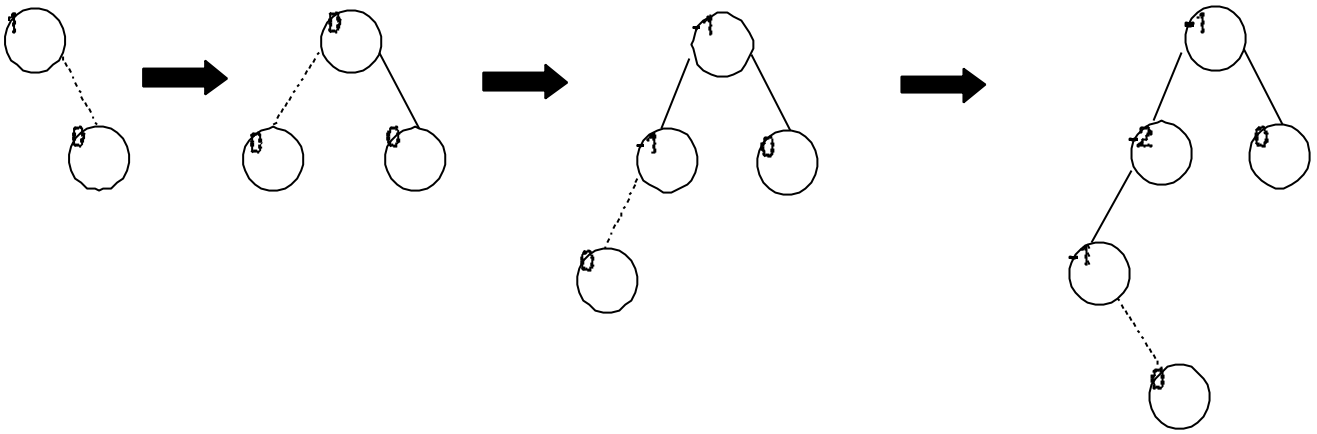
× La inserción se hace en las hojas

× Cuando creamos un nuevo nodo el campo de equilibrio vale 0, ya que la altura del subarbol izquierdo es igual que la del derecho.

× Cuando insertamos por la derecha incrementamos el valor del campo de equilibrio.

× Cuando insertamos por la izquierda decrementamos el valor del campo de equilibrio.

Ejemplos:

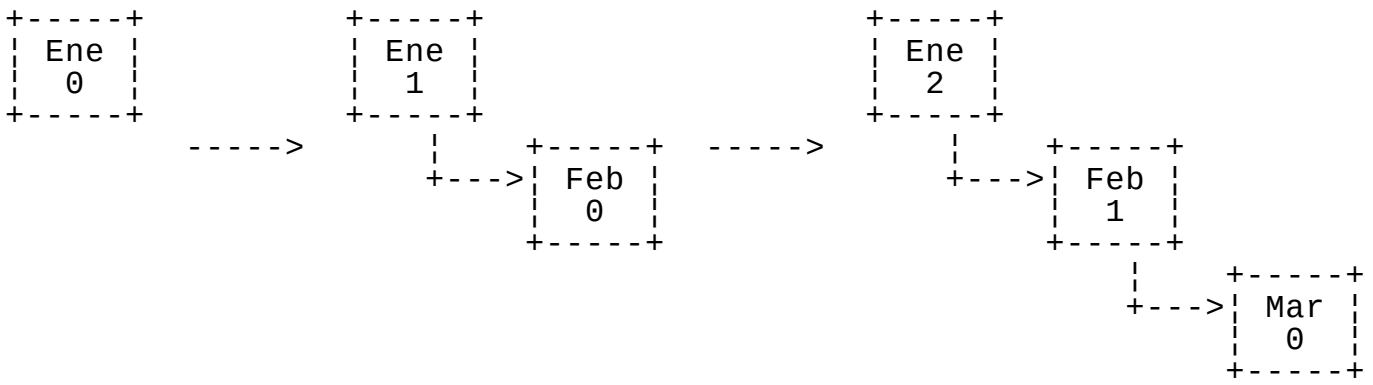


5.1 Inserción en AVL

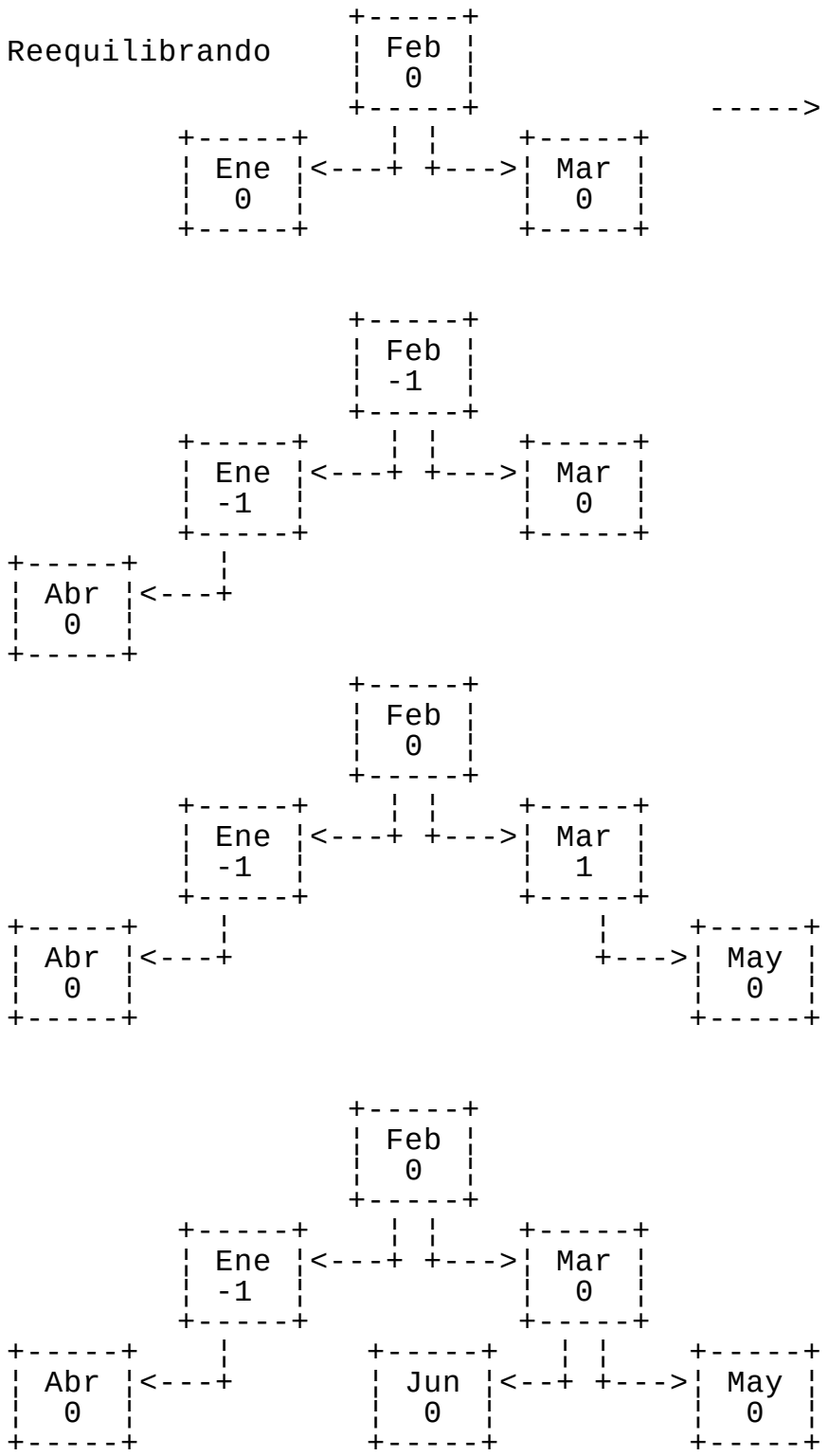
La inserción se hace siempre en las hojas, y vamos a tener un campo de equilibrio.Vamos a tener una variable global llamada crecido de tipo BOOLEAN, que en el momento de insertar lo vamos a poner a TRUE para después controlar si se ha desequilibrado o no.

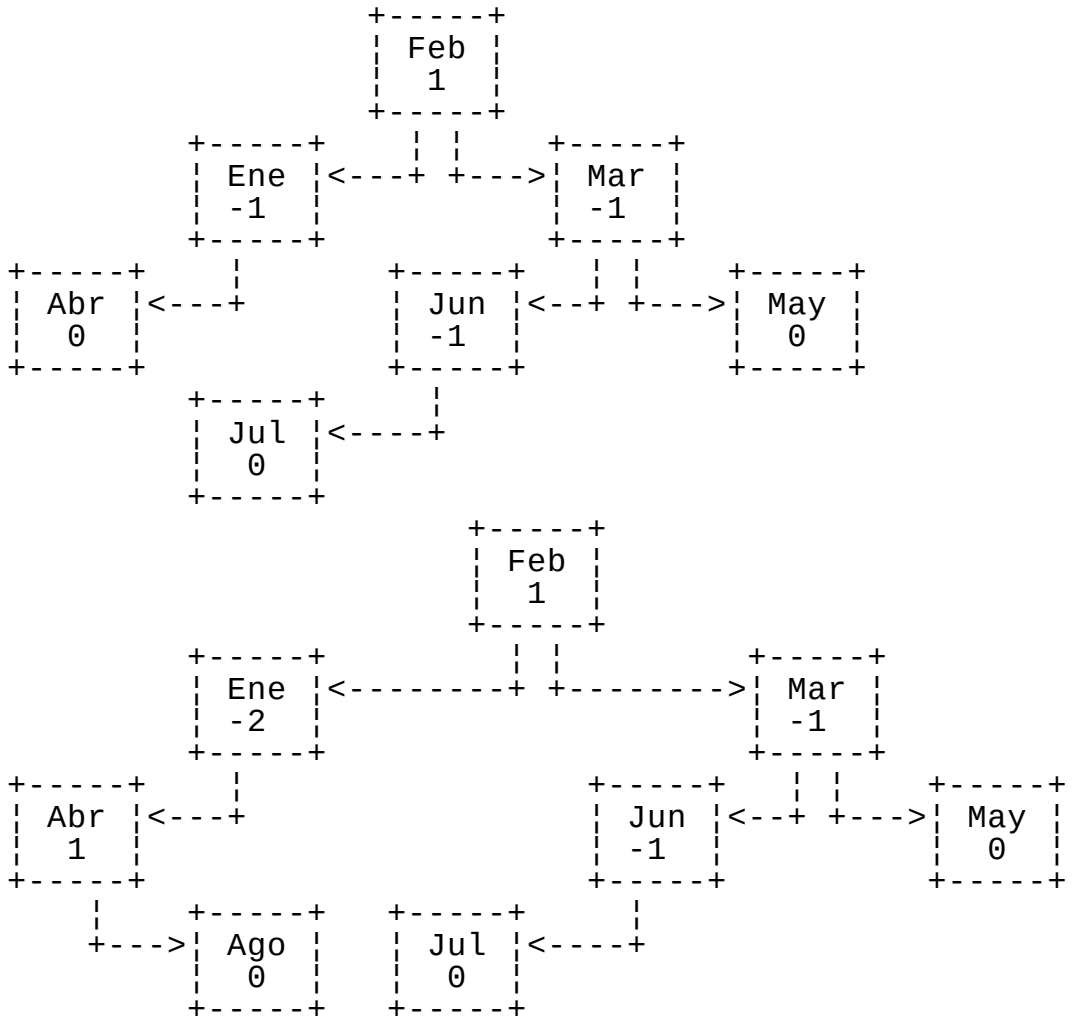
Se sube restando o sumando 1 hasta llegar a un cero.Si a la hora de sumar o restar un uno se sale del rango hay que reequilibrar.

Ejemplo:

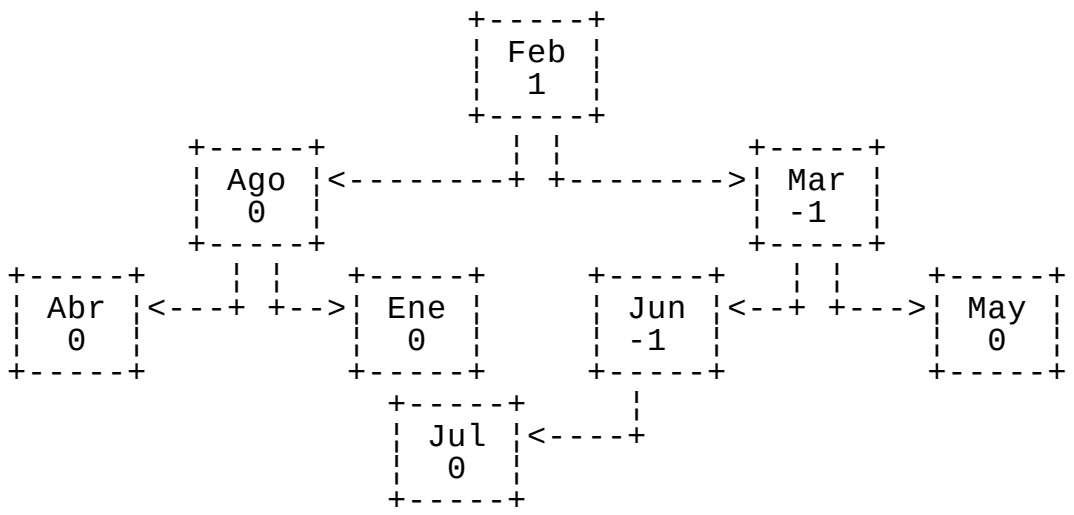


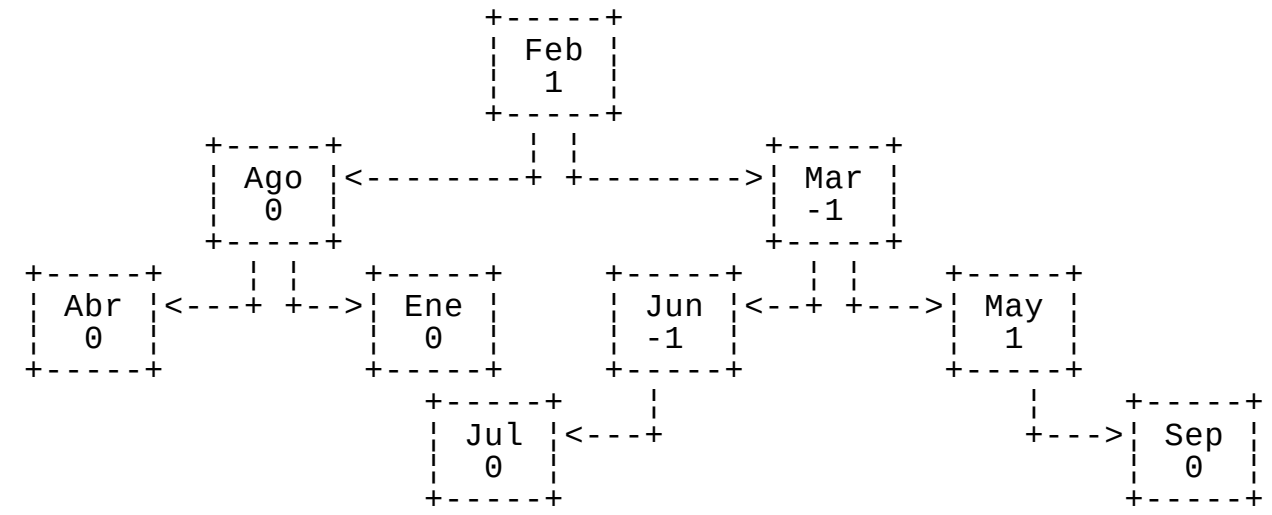
Reequilibrando

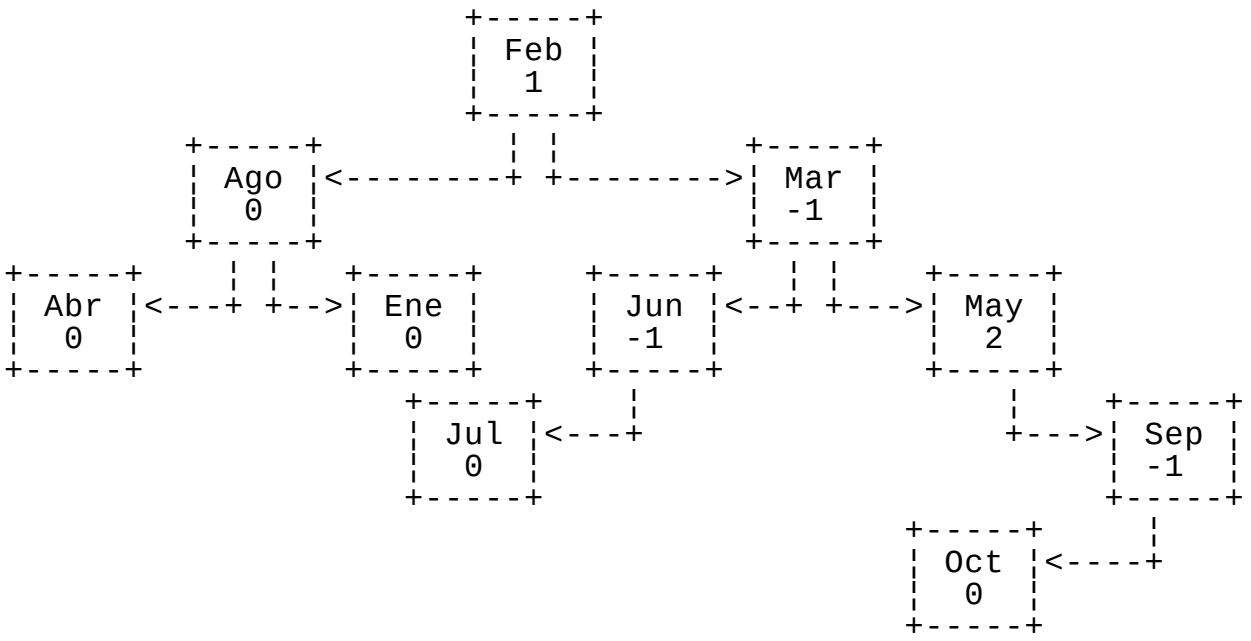




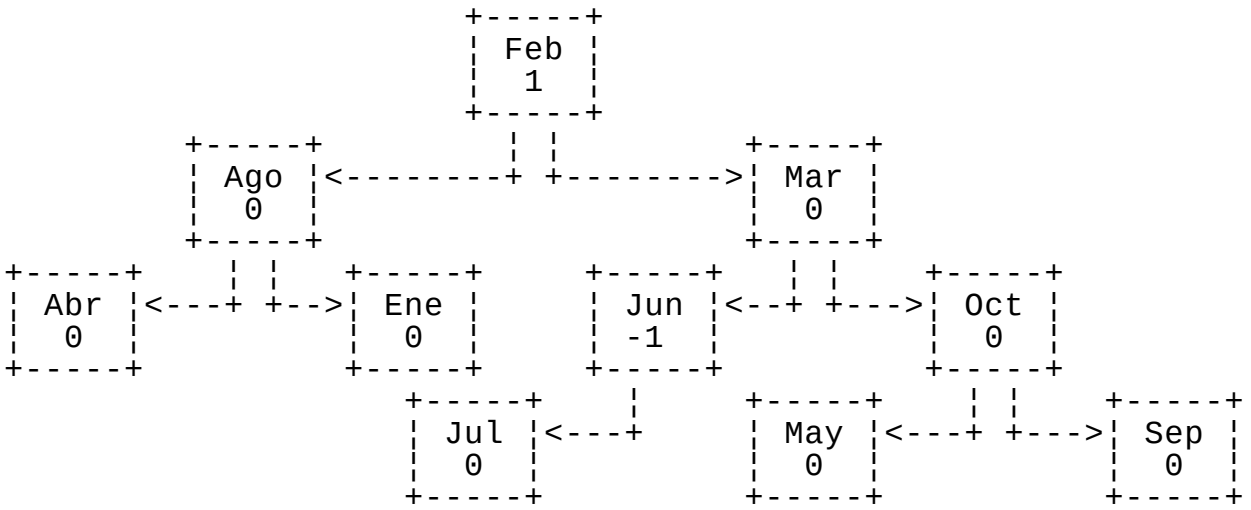
Reequilibrando:



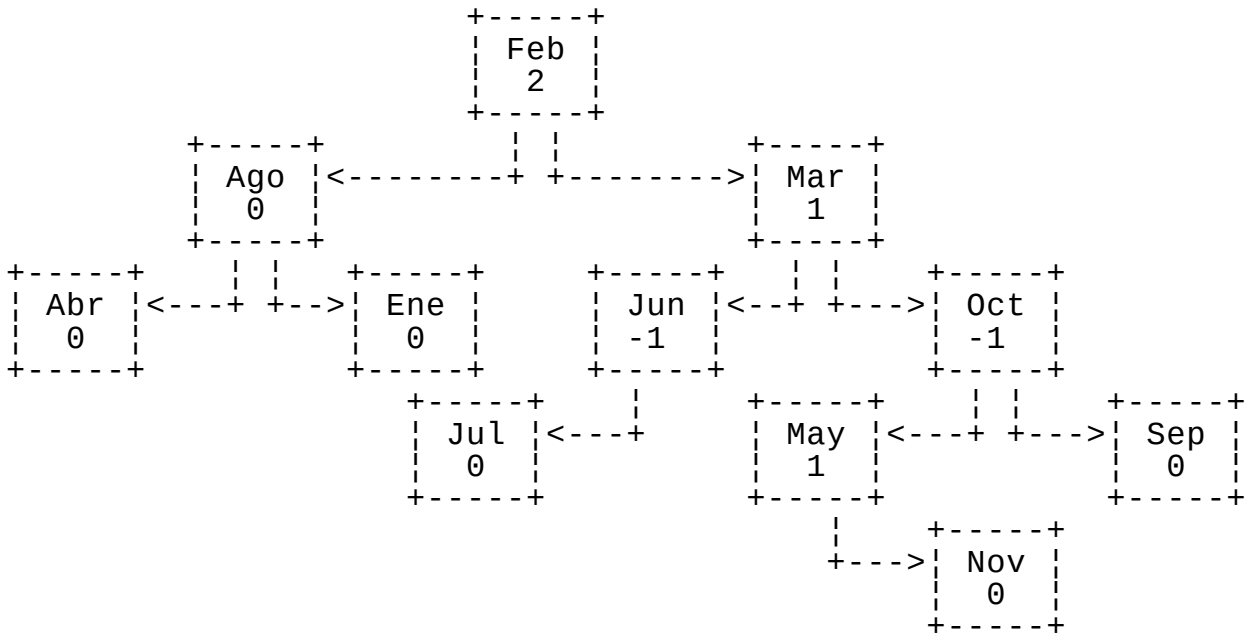




Equilibrando:



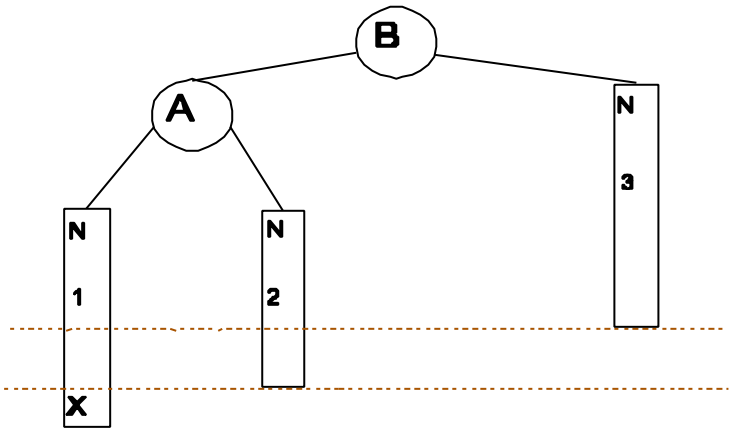
En la siguiente inserción se produce un desequilibrio en la raiz del arbol.



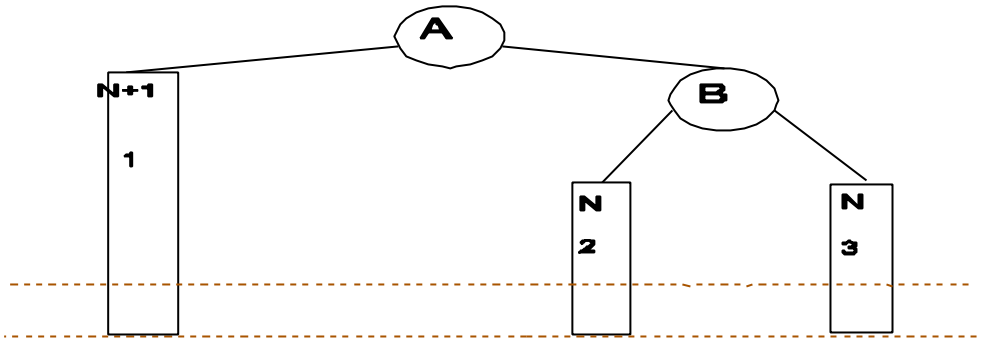
Desequilibrios y reequilibrios

Al insertar un nuevo nodo en un árbol equilibrado se pueden producir desequilibrios, que quedarán representados en los casos mostrados a continuación.En las figuras, las cajas rectangulares representan subárboles, y la altura añadida por la inserción se indica con cruces.Simples transformaciones restauran el equilibrio deseado.

Desequilibrio izquierda-izquierda simple.



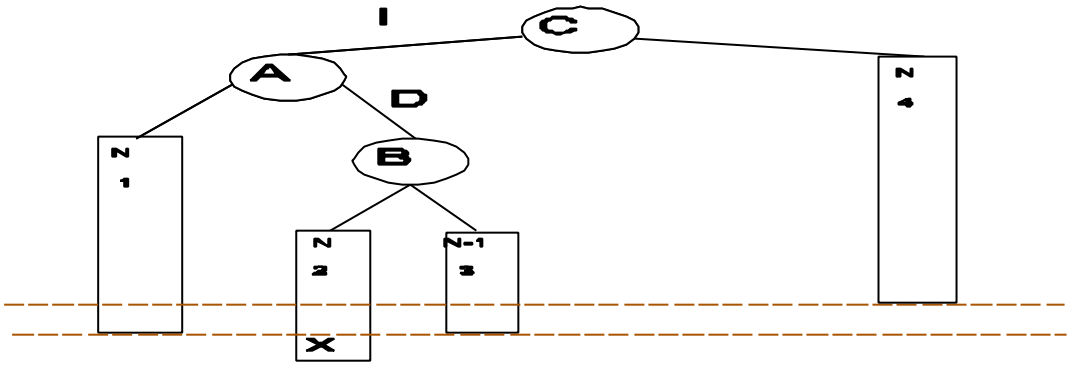
Se corrige con la rotación izquierda-izquierda simple, que consiste en subir el nodo A, que tendrá al final el campo de equilibrio a 0.



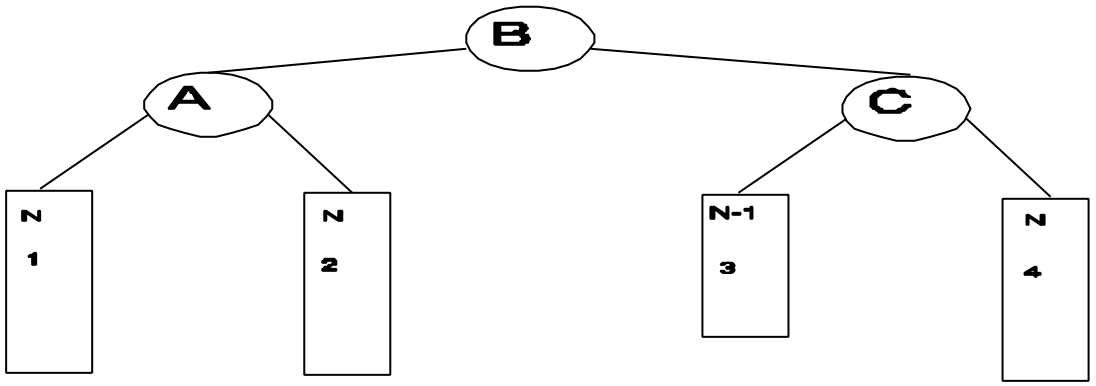
Desequilibrio derecha-derecha simple

Se reequilibra con rotación derecha-derecha simple.Es el reflejado del anterior.

Desequilibrio izquierdo-derecho compuesto



Se corrige con la rotación izquierda-derecha compuesta.Se sube el nodo B que pasa a tener el campo de equilibrio a 0.



Desequilibrio derecho-izquierdo compuesto

Es similar, solo que la imagen reflejada.

Implementación de la inserción

Un algoritmo que inserte y reequilibre dependerá en forma crítica de la forma en que se almacene la información relativa al equilibrio del árbol.Una solución consiste en mantener la información sobre el equilibrio, de forma completamente implicita, en la estructura misma del árbol.En este caso, sin embargo, el factor de equilibrio debe "*averiguarse*" cada vez que se encuentre afectado por una inserción, con el consiguiente trabajo resultante.

Otra forma (la que vamos a usar) consiste en atribuir a, y almacenar con, cada nodo un factor de equilibrio explícito.La definición de nodo se amplía entonces a:

```
TYPE Nodo=RECORD
          Clave:Tclave;
          Info:Tinfo;
          Izq,Der:^Nodo;
          Equi:-1..1
         END
```

El proceso de inserción de un nodo consta fundamentalmente de las tres partes consecutivas siguientes:

- Seguir el camino de búsqueda hasta que se comprueba que la clave aún no está en el árbol.
- Insertar el nuevo nodo y determinar el factor de equilibrio resultante.
- Volver siguiendo el camino de búsqueda y comprobar el factor de equilibrio de cada nodo.

Aunque este método realiza algunas comprobaciones que son redundantes, una vez que se establece el equilibrio, éste no necesita comprobarse en los antecesores del nodo en cuestión.

En cada paso se debe mandar información sobre si la altura del subárbol (en el cual se ha realizado la inserción) ha aumentado o no.Por lo tanto se extiende la lista de parámatros del procedimiento con el BOOLEAN Crecido, que debe ser un parámetro de tipo variable, ya que se utiliza para transmitir un resultado.

```
PROCEDURE Insertar (VAR Arbol:Tarbol;Clave:Tclave;Info:Tinfo;
                                          VAR Crecido:BOOLEAN);

   PROCEDURE Requizq (VAR Arbol:Tarbol;VAR Crecido:BOOLEAN);
   VAR
      Aux1,Aux2:Tarbol;
   BEGIN
      CASE Arbol^.Eq OF
         1: BEGIN
               Arbol^.Eq:=0;     (* Hemos insertado por IZDA 1-1=0 *)
               Crecido:=FALSE    (* Solo va a cambiar cuando Eq=0 *)
            END;
```

```
         0:Arbol^.Eq:=-1;        (* 0-1=-1 *)

        -1: BEGIN                (* Hay que reequilibrar *)
               Aux1:=Arbol^.izq;
               IF Aux1^.Eq=-1
                   THEN BEGIN    (* Izda-Izda simple *)
                           Arbol^.izq:=Aux1^.der;
                           Aux1^.der:=Arbol;
                           Arbol^.Eq:=0;
                           Aux1^.Eq:=0;
                           Arbol:=Aux1
                        END
                   ELSE BEGIN    (* Izda-Dcha compuesto *)
                           Aux2:=Aux1^.der;
                           Aux1^.der:=Aux2^.izq;
                           Aux2^.izq:=Aux1;
                           Arbol^.izq:=Aux2^.der;
                           Aux2^.der:=Arbol;
                           IF Aux2^.Eq=-1
                                THEN BEGIN
                                        Aux^.Eq:=0;
                                        Arbol^.Eq:=1
                                     END
                                ELSE BEGIN
                                        Aux1^.Eq:=-1;
                                        Arbol^.Eq:=0
                                     END
                           Aux2^.Eq:=0;
                           Arbol:=Aux2
                        END;
               Crecido:=FALSE
            END;
   END;
BEGIN
  IF Arbol=NIL
    THEN BEGIN
            CrearNodo(Arbol,Clave,Info);
            Crecido:=TRUE;
         END
  ELSE IF Arbol^.Clave=Clave
    THEN "Error"
  ELSE IF Arbol^.Clave>Clave
    THEN BEGIN
            Insertar(Arbol^.izq,Clave,Info,Crecido);
            IF Crecido
              THEN Requizq(Arbol,Crecido)
         END
    ELSE BEGIN
            Insertar(Arbol^.der,Clave,Info,Crecido);
            IF Crecido
              THEN Requder(Arbol,Crecido)
         END
END;
```

La complejidad de las operaciones de equilibrado sugiere que estos árboles deben utilizarse sólo si las recuperaciones de información son considerablemente más frecuentes que las inserciones.

5.2 Borrado en AVL

Vamos a ver solo las distintas posibilidades que se pueden dar al borrar un nodo en el lado derecho.A la izquierda es simétrico.

Caso 1.Raiz.

Caso 1.1: Si alguno de los subarboles que salen de la raiz está vacio, entonces el otro estará vacio o solo tiene un nodo, por ser un árbol equilibrado.

× Si solo tiene dos nodos se sube el no borrado hacia arriba.
× Si solo está el nodo a borrar, el arbol acaba vacio.

Caso 1.2: Si no hay ningún subarbol vacio se sube el nodo de más a la derecha del subarbol izquierdo, se intercambia los valores de la raiz por los de ese nodo, y despues es borra este último.

Caso 2.Borrado en el subarbol derecho.

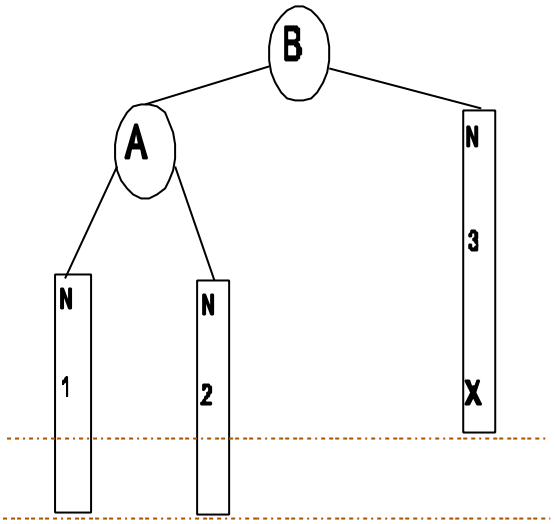
Caso 2.1: Si el campo de equilibrio tiene un cero, los dos subarboles son iguales.Entonces lo borramos y el campo de equilibrio pasa a -1.

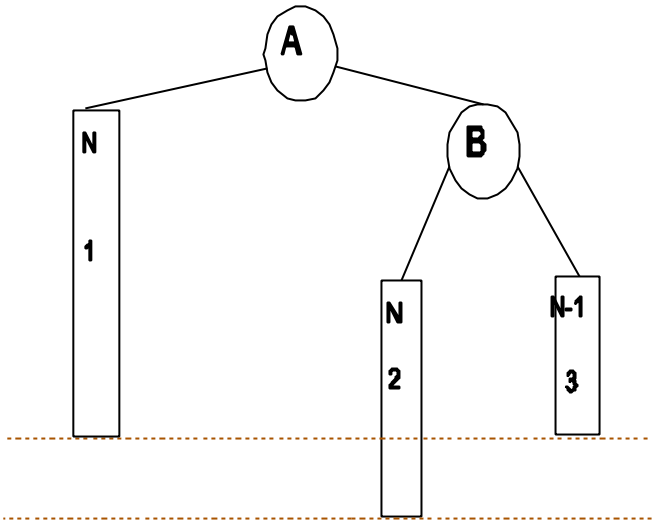
Caso 2.2: Si tiene un 1, entonces el subarbol derecho tiene una altura más que el izquierdo.Al borrar equilibramos y pasa a ser 0 ya que restamos 1.

× Se puede haber desequilibrado por la izquierda porque al borrar se ha disminuido en uno la altura del árbol.

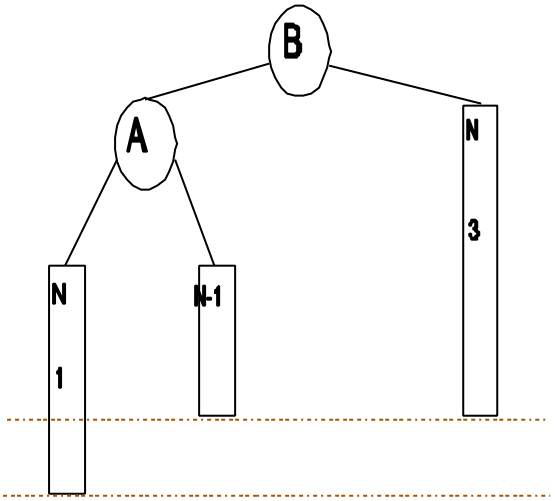
Caso 2.3: Si tiene un -1, la altura del subarbol izquierdo es mayor que la del derecho.Al borrar en el derecho se rompe el equilibrio, que pasa a -2.Hay tres casos.

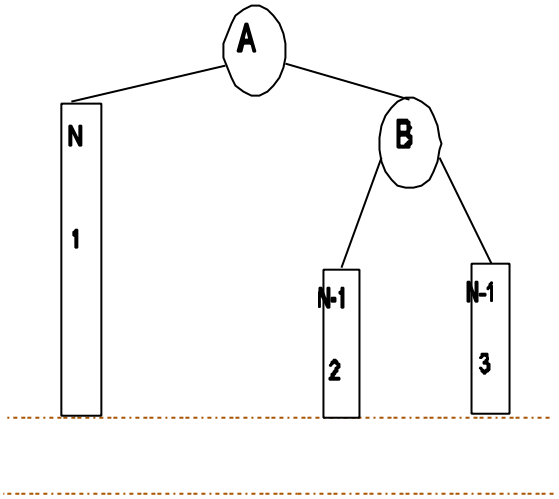
Caso 2.3.1



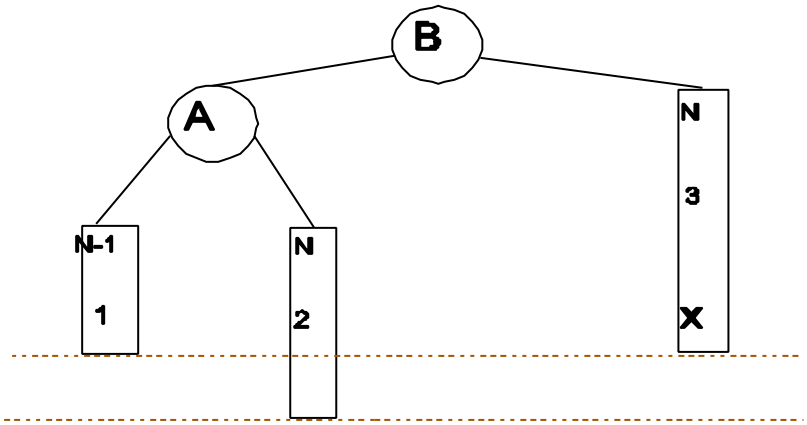


Caso 2.3.2

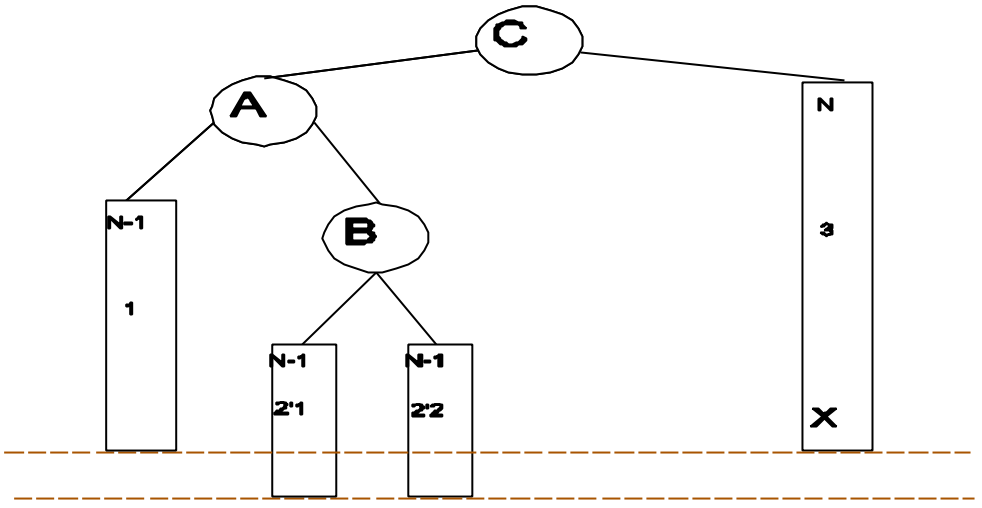




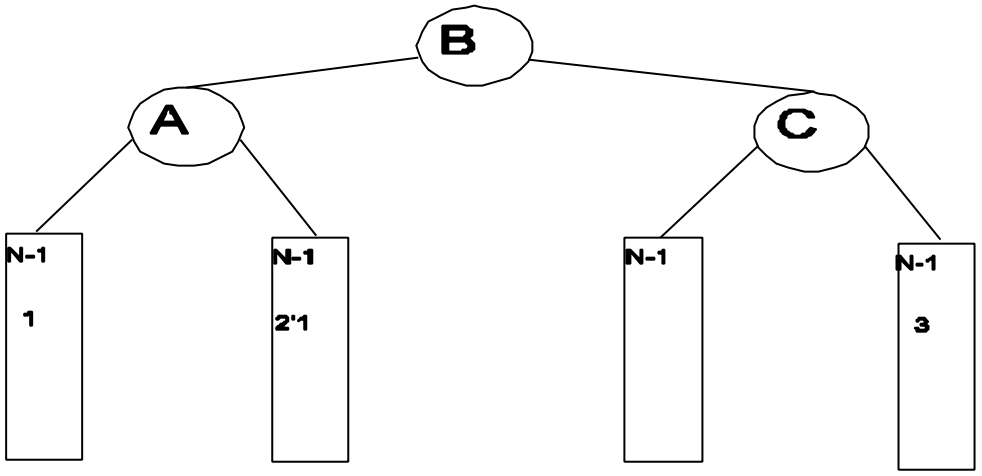
Caso 2.3.3



Que visto de otra forma, puede ser:



× Mediante rotacion izquierda-derecha compuesta queda:



× Hay otros dos casos, que el bloque 2'2 sea el más pequeño, o que lo sea el 2'1.Tienen altura N-2 y por lo demás se tratan igual.

6. ARBOLES DE BUSQUEDA OPTIMOS.

Son arboles binarios o no binarios en los que los nodos con mayor probabilidad de ser consultados se situan más cerca de la raiz.El camino medio, será el camino del nodo por la probabilidad de ser consultado.Son estáticos, una vez creados no se borra, ni se inserta, etc.

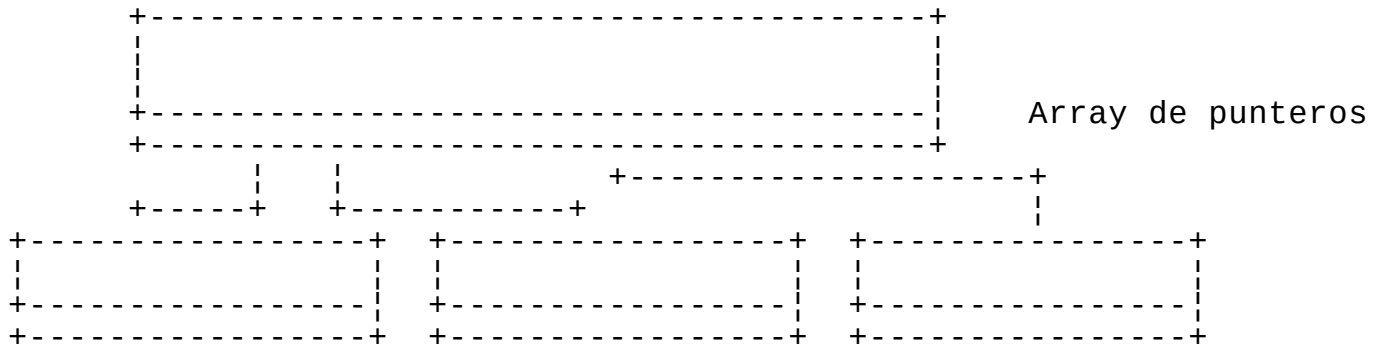
Se utilizan en analizadores de lexico y diccionarios de datos.El camino medio, será el camino del nodo por la probabilidad de ser consultado.

7. ARBOLES MULTICAMINO.

Son aquellos que tiene grado mayor que dos.Hay varias formas de representarlos.

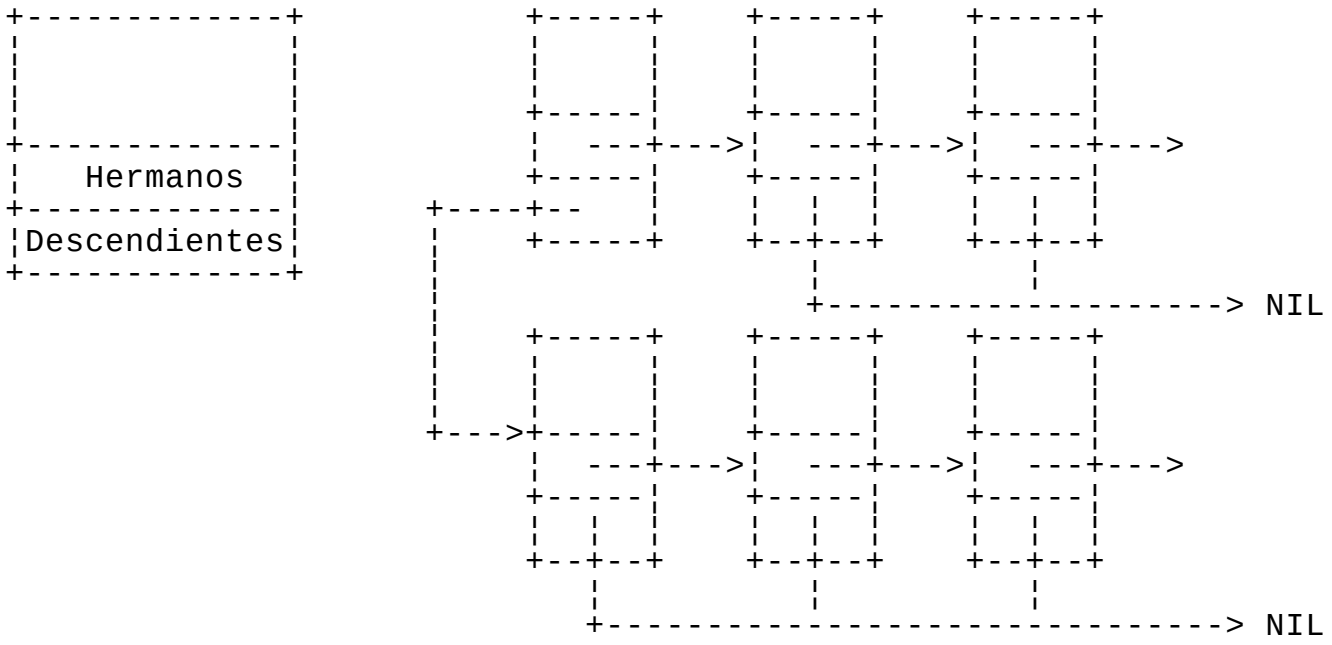
7.1 Formas de representación.

1.Utilizando arrays



× **Inconveniente**: Número de descendientes fijo.

2.Usando una lista

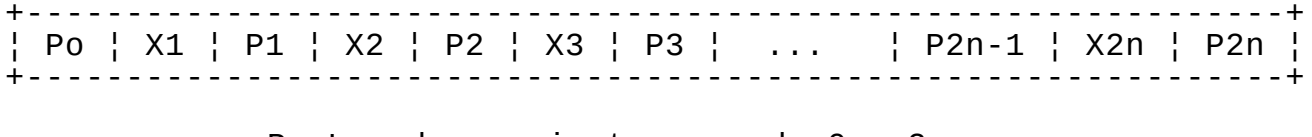


7.2 Arboles B

Los subarboles de un arbol binario normal se agrupan dando lugar a una página.

Características:

- Si la página tiene N claves, tendrá N+1 descendientes.
- Si el árbol tiene orden N, todas las páginas tendrán como mucho 2N claves, y como mínimo N claves, excepto la raiz (factor de ocupación 50%).
- Los árboles B crecen en altura por la raiz.
- Las páginas hoja deben estar al mismo nivel.
- Debe ser un árbol de búsqueda multicamino (debe estar ordenado).



P: Los descencientes van de 0 a 2n
X: Claves

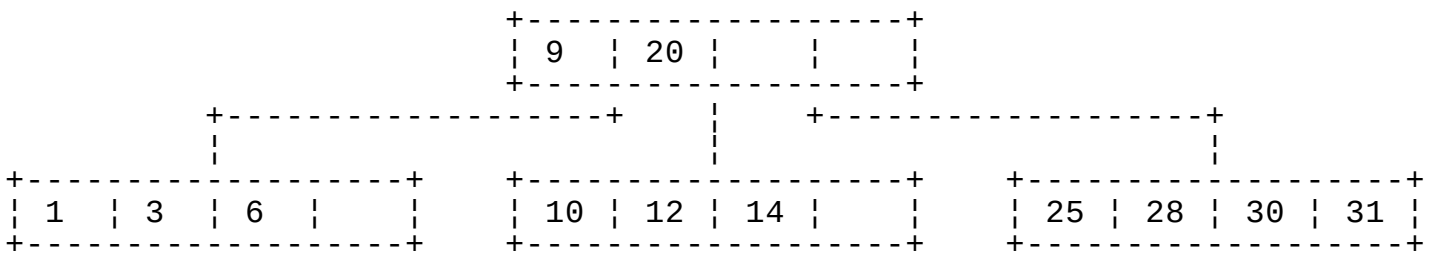
Busqueda en un árbol B.

Se empieza por la raiz y se va descendiendo por los nodos página, hasta que se encuentre la clave o se llegue al final.

Y < X1 ------> Po
Xi < Y< Xi+1 ------> Pi

inserción en un arbol B de orden 2.

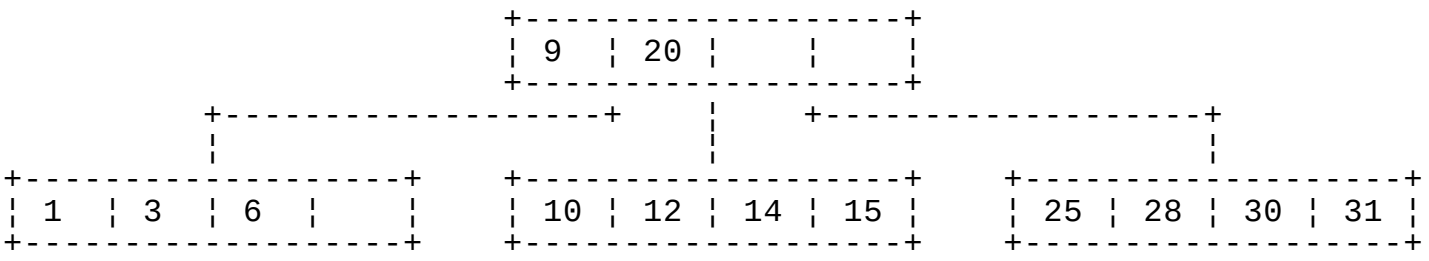
Tendrá en cada página 4 claves y 5 descendientes.



Tenemos los siguientes casos:

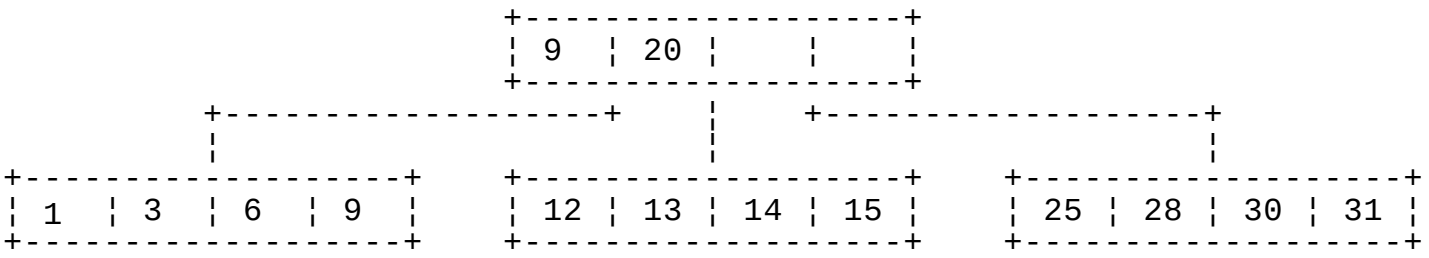
1. Hay sitio en la hoja en la que se tiene que insertar: Se inserta y ya hemos acabado.

Ejemplo el 15.



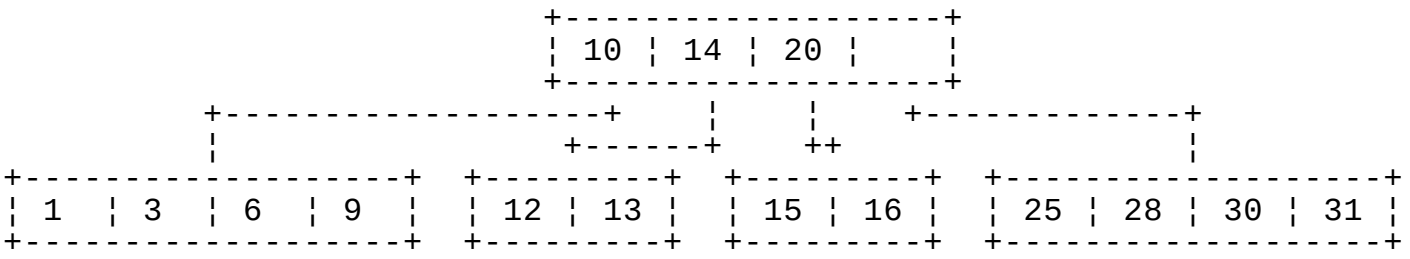
2. No hay sitio en la hoja pero hay huecos en las hojas contiguas: Hacemos como que metemos el número, y vemos si hay sitio libre en la hoja contigua de la derecha o en la de la izquierda.

Ejemplo el 13.Subimos la clave de más a la izquierda en la hoja arriba, y bajamos la que habia a la página de la izquierda.



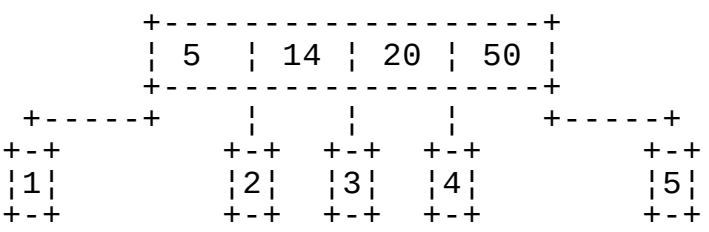
3. No hay sitio, ni en la hoja ni en las contiguas: Hacemos como que metemos el número, y dividimos la hoja por dos, subiendo la clave del medio arriba.

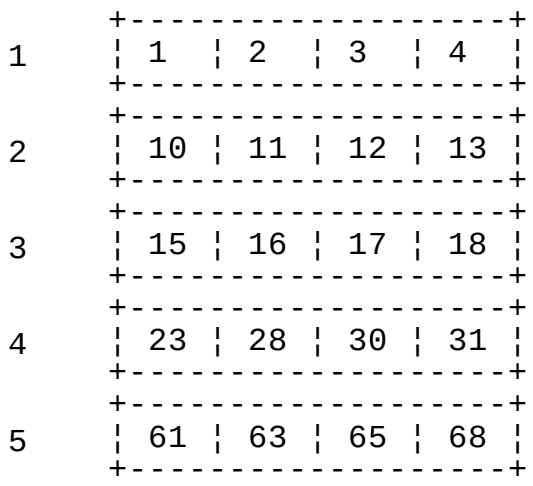
Ejemplo 16.



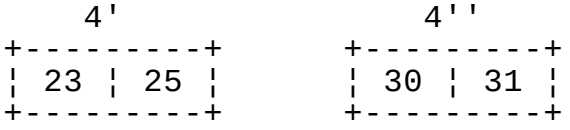
4. Si estan todas las hoj s llenas.Se crece por la raiz.

Ejemplo 25.

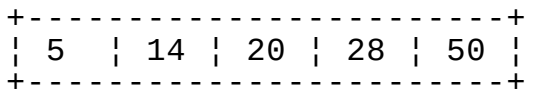




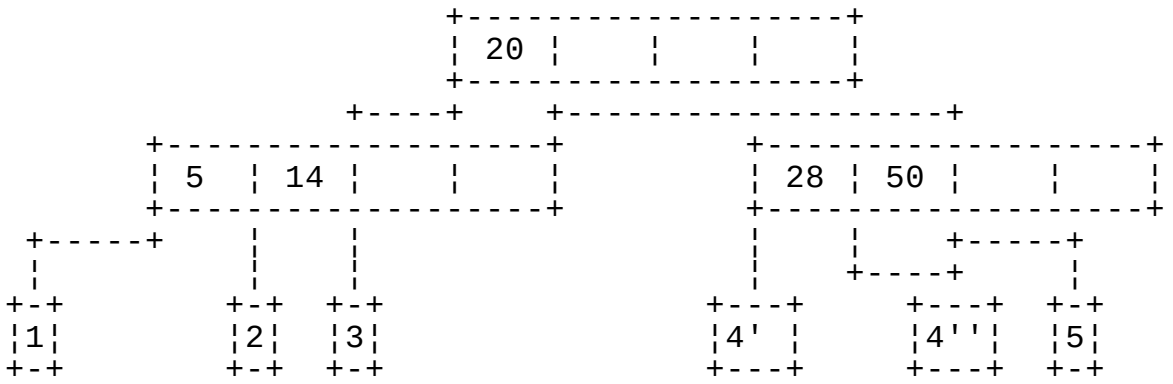
Para insertar el número 25 tendríamos que colocarlo entre el 23 y el 28. Como no hay sitio ni a la izquierda ni a la derecha, tendremos que dividir la hoja y subir el del medio



Subimos el del medio, que es el 28, luego en la página raíz tendremos la siguiente situación:



y hacemos lo mismo que en una hoja, dividimos y subimos hacia arriba la clave del centro:



Declaración de tipo de un árbol B.

```
TYPE
   Tarbol=^Pagina;
   Ele=RECORD
          Clave:Tclave;
          Info:Tinfo
       END;
   Pagina=RECORD
             Elemento:ARRAY [1..2n] OF Ele;
             Descendientes:ARRAY [0..2n] OF Tarbol;
             NumClaves:1..2n
          END;
```

2 de Febrero 1990.

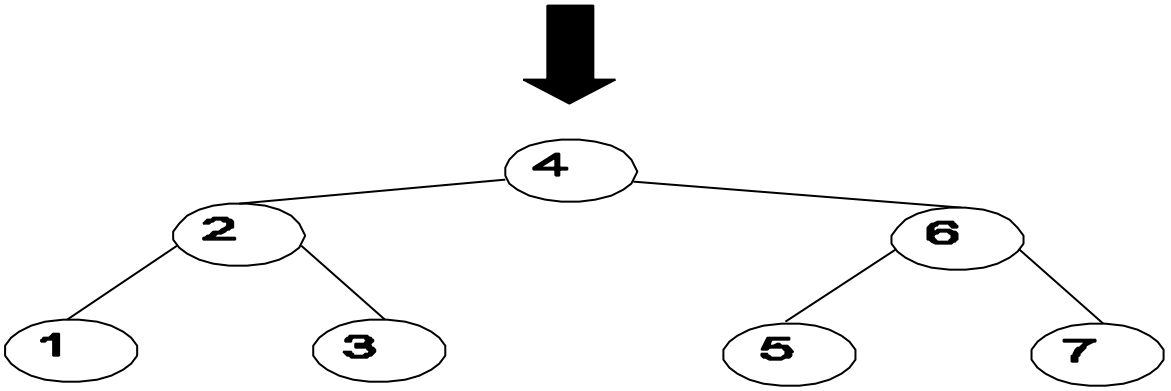
Dada las siguientes declaraciones:

```
Tarbol=^Elem;
Elem=RECORD
        Info:Tinfo;
        Clave:Tclave;
        izq,dcho:Tarbol
     END;
Ttabla=ARRAY[1..n] OF Tclave;
```

Diseñar e implementar un algoritmo recursivo que cree un arbol perfectamente equilibrado con los elementos de la tabla.La tabla está ordenada.El arbol perfectamente equilibrado tiene que ser de búsqueda.

La siguiente tabla:

1	2	3	4	5	6	7




```
PROCEDURE CrearArbol(VAR Arbol:Tarbol;Tabla:Ttabla);

   PROCEDURE Arbol1(VAR Arbol:Tarbol;izdo,dcho:INTEGER);
   VAR
      Medio:INTEGER;
   BEGIN

      IF izdo<=dcho
            THEN BEGIN
                    Medio:=(izdo+dcho) DIV 2;
                    NEW (Arbol);
                    Arbol^.Clave:=Tabla[Medio];
                    Arbol^.izdo:=NIL;
                    Arbol^.dcho:=NIL;
                    Arbol1(Arbol^.izdo,izdo,Medio-1);
                    Arbol1(Arbol^.dcho,Medio+1,dcho)
                 END
   END;

BEGIN
   Arbol1(Arbol,1,n)
END;
```

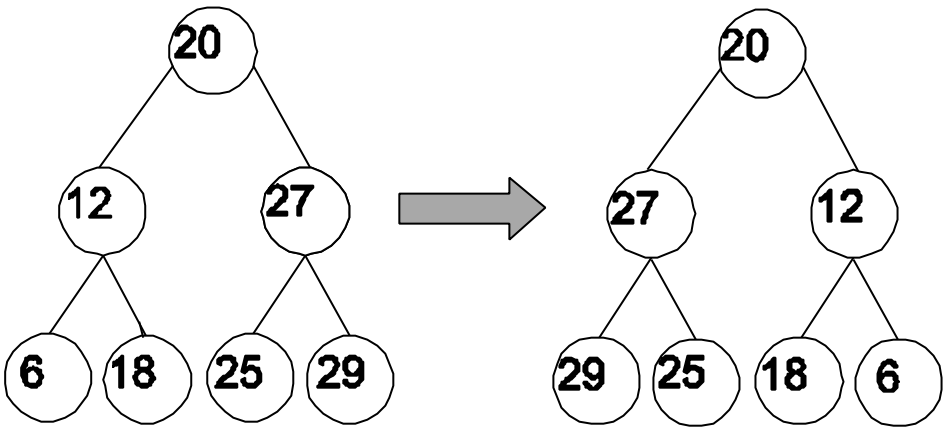
8-Febrero-1991.

Tenemos que hacer la llamada inicializando Nivel a 1.

```
FUNCTION Lc(Arbol:Tarbol;Clave:Tclave;Nivel:INTEGER):INTEGER;
BEGIN
   IF Arbol=NIL
       THEN Lc:=0
       ELSE IF Arbol^.Clave=Clave
                 THEN Lc:=Nivel
                 ELSE Lc:=Lc(Arbol^.izq,Clave,Nivel+1)+
                                    Lc(Arbol^.der,Clave,Nivel+1)
END;
```

Ejercicio 1.

Escribir un procedimiento recursivo que invierta un arbol, que halle su imagen simétrica.
Ejemplo:




```
PROCEDURE Reflejado(VAR Arbol:Tarbol);
VAR
   Aux:Tarbol;
BEGIN
   IF Arbol<>NIL
        THEN BEGIN
                Aux:=Arbol^.izq;
                Arbol^.izq:=Arbol^.der;
                Arbol^.der:=Aux;
                Reflejado(Arbol^.izq);
                Reflejado(Arbol^.der)
             END
END;
```

Ejercicio 2.

Dado un arbol binario de búsqueda, escribir una función que devuelva el nivel del nodo con mayor clave.Hacer lo mismo para un arbol binario que no sea de búsqueda.

```
FUNCTION Nivel_Clave (Arbol:Tarbol):INTEGER;
BEGIN
   IF Arbol=NIL
        THEN Nivel_Clave:=0
        ELSE Nivel_Clave:=1+Nivel_Clave(Arbol^.der)
END;
```

Para arbol binario no de búsqueda.

```
PROCEDURE Nivel_Clave (Arbol:Tarbol;VAR ClaveMA:Tclave;NivelAux:INTEGER
                                  VAR Nivel:INTEGER);
BEGIN
   IF Arbol<>NIL
        THEN BEGIN
                IF Arbol^.Clave>ClaveMa
                      THEN BEGIN
                              Nivel:=NivelAux;
                              ClaveMa:=Arbol^.Clave
                           END;
              Nivel_Clave(Arbol^.der,ClaveMa,NivelAux+1,Nivel);
              Nivel_Clave(Arbol^.izq,ClaveMa,NivelAux+1,Nivel)
           END
END;
```

Ejercicio 3.

Disponemos de un arbol binario de busqueda que puede contener claves repetidas, y la inserción de tales claves se hizo a veces en el izquierdo y otras en el derecho.Se pide codificar una función recursive que dado un arbol como el anterior, determine si existen claves repetidas o no.

Recorriendo en orden central, comparamos con la actual la clave anterior para ver si está repetida.

```
FUNCTION Clave_Repe (Arbol:Tarbol):BOOLEAN;
VAR
   Anterior:Tclave;
BEGIN
   Anterior:=MAXINT;
   Clave_Repe:=Repetidos(Arbol,Anterior)
END;
```

```
FUNCTION Repetidos (ARbol:Tarbol;VAR Ante:Tclave):BOOLEAN;
BEGIN
  IF Arbol=NIL THEN
    Repetidos:=FALSE
  ELSE IF Repetidos(Arbol^.izq,ante) THEN
    Repetidos:=TRUE
  ELSE IF Ante=Arbol^.Clave THEN
    Repetido:=TRUE
  ELSE
    BEGIN
      Ante:=Arbol^.Clave;
      Repetido:=Repetido(Arbol^.der,Ante)
    END
END;
```

Ejercicio 4.

Escribir una función recursive que calcule el número de hojas que hay en un arbol.

```
FUNCTION Numero (Arbol:Tarbol):INTEGER;
BEGIN
   IF Arbol<>NIL
        THEN IF Arbol^.der=Arbol^.izq
                  THEN Numero:=1
                  ELSE Numero:=Numero(Arbol^.izq)+Numero(Arbol^.der)
        ELSE Hojas:=0
END;
```

Ejercicio 5.

Implementar una función que diga si dos arboles binarios son iguales.

```
FUNCTION Igual (Arbol1,Arbol2:Tarbol):BOOLEAN;
BEGIN
  IF Arbol1=NIL AND Arbol2=NIL THEN
    Igual:=FALSE
  ELSE IF ((Arbol1=NIL) AND (Arbol2<>NIL)) OR
      ((Arbol1<>NIL) AND (Arbol2=NIL)) THEN
    Iguales:=FALSE
  ELSE IF Arbol1^.Clave<>Arbol2^.Clave THEN
    Iguales:=FALSE
  ELSE
    Iguales:=Igual(Arbol1^.izq,Arbol2^.izq)                    AND
Igual(Arbol1^.der,Arbol2^.der)
END;
```

4-Septiembre-89.

Dada la misma declaración de árbol, escribir una función BOOLEANA que determine si todas las hojas del árbol están al mismo nivel.

```
FUNCTION Hoja_Nivel(Arbol:Tarbol):BOOLEAN;
VAR
   Nivel:INTEGER;
BEGIN
   Nivel:=0;
   Hoja_Nivel:=Mismo_Nivel(Arbol,Nivel,1)
END;

FUNCTION Mismo_Nivel(Arbol:Tarbol;VAR Primera:INTEGER,
                                  NivelAct:INTEGER):BOOLEAN;
BEGIN
  IF Arbol=NIL THEN
    Mismo_Nivel:=TRUE
  ELSE
    IF (Arbol^.izq=NIL) AND (Arbol^.der=NIL) THEN
      IF Primera=0     (* Es la primera hoja *) THEN
        Primera:=NivelAct
      ELSE
        IF Primera<>NivelAct THEN
          Mismo_Nivel:=FALSE
        ELSE
          Mismo_Nivel:=TRUE
    ELSE
      IF Mismo_Nivel (Arbol^.izq,Primera,NivelAct+1) THEN
        Mismo_Nivel:=Mismo_Nivel (Arbol^.der,Primera, NivelAct+1)
END;
```

15-Septiembre-1988.

Codificar una función recursiva que recibiendo como parámetro un árbol binario de búsqueda de **N** nodos y un número entero **K** menor o igual que N, devuelve el K-esimo número más pequeño del árbol.
No se puede utilizar ninguna estructura auxiliar.No existen números repetidos en el árbol.

```
FUNCTION Kesimo(Arbol:Tarbol;k:INTEGER;VAR contador:INTEGER):INTEGER;
BEGIN
  IF Arbol<>NIL THEN
    BEGIN
      Kesimo:=Kesimo(Arbol^.izq,k,Contador);
      IF Contador<k THEN
        BEGIN
          Contador:=Contador+1;
          IF Contador=k THEN
            Kesimo:=Arbol^.Clave
          ELSE
            Kesimo:=Kesimo(Arbol^.der,k,Contador)
        END
    END
END;

La llamada a la función seria:
              Contador:=0;
              Numero:=Kesimo(Arbol,k,Cont);
```

17-septiembre-1991.

Con la declaración de arbol del anterior ejercicio, implementar un algoritmo que determine si un árbol A está incluido en otro árbol B, es decir, todas las claves del árbol A están en B con la misma topología.

Va a empezar por el árbol entero y va a acabar cuando la raiz del subarbol 2 sea igual a la clave del arbol 1.

```
FUNCTION Incluir(A1,A2:Tarbol):BOOLEAN;
BEGIN
  IF A2=NIL THEN
    Incluir:=TRUE;
  ELSE
    IF A1=NIL THEN
      Incluir:=FALSE
    ELSE
      IF A1^.Clave>A2^.Clave THEN
        Incluir:=Incluir(A1^.izda,A2)
      ELSE
        IF A1^.Clave<A2^.Clave THEN
          Incluir:=Incluir(A1^.dcha,A2)
        ELSE
          Incluir:=Igual(A1,A2)
END;

FUNCION Igual(A1,A2:Tarbol):BOOLEAN;
BEGIN
  IF A2=NIL THEN
    Igual:=TRUE
  ELSE
    IF A1=NIL THEN
      Igual:=FALSE
    ELSE
      IF A1^.Clave=A2^.Clave THEN
        Igual:=Igual(A1^.izda,A2^.izda) AND Igual(A1^.dcha,A2^.dcha)
      ELSE
        Igual:=FALSE
END;
```

12-Junio-1991.

Dada esa declaración de tipo, se pide codificar una fución recursiva que recibiendo un puntero del tipo Tarbol que apunta a un arbol multicamino, que se sabe que es de búsqueda, determine si dicho árbol cumple las caracteristicas de un arbol B.Cada página del árbol únicamente podrá ser visitada una vez.

- N <= Número de claves <= 2N
- Si una página tiene N claves, tiene N+1 descendientes
- Todas las páginas hoja al mismo nivel

Las dos últimas condiciones son en realidad la misma.

```
FUNCTION Esb(Arbol:Tarbol;VAR Nivel:INTEGER;NivelAux:INTEGER):BOOLEAN;
VAR
  i:INTEGER     (* Para hacer un FOR desde el primer puntero
                   hasta el último, e ir bajando por ellos *)
  Aux:BOOLEAN;
BEGIN
  IF Arbol=NIL THEN
    IF Nivel=0 THEN
      Nivel:=NivelAux
    ELSE
      Esb:=Nivel=NivelAux
  ELSE
    IF (NivelAux=1)      (* Es la raiz, no hay que comprobar nada *)
       OR (Arbol^.Numclave>=n) THEN (* Ocupación mayor del 50% *)

               (* Si la pagina es la raiz, la primera comprobación
                  da directamente TRUE, con lo que no imporla la
                  otra.Si es FALSE, mirará la otra. *)

      BEGIN
        Aux:=TRUE;
        FOR i:=0 TO Arbol^.Numclaves DO
          Aux:=Aux AND Esb(Arbol^.Descend[i],Nivel,NivelAux+1)
        Esb:=Aux
      END;
    ELSE
      Esb:=FALSE    (* Porque no cumple las condiciones *)
END;
```

14-septiembre-1990.

Se busca la clave hasta que la encontremos o estemos situados sobre un nodo que sea una hoja.

- Nodo cualquiera ----> Bajar
- Nodo clave ----> Contar
- Hoja clave ----> Contar
- Hoja cualquiera ----> Eliminar

```
FUNCTION Hojas(VAR Arbol:Tarbol;Clave:Tclave):INTEGER;
VAR
  Aux:Tarbol;

  FUNCTION N_Hojas (Arbol:Tarbol):INTEGER;
  BEGIN
    IF Arbol=NIL THEN
      N_Hojas:=0
    ELSE
      IF Arbol^.izq=NIL AND Arbol^.der=NIL THEN
        N_Hojas:=1
      ELSE
        N_Hojas:=N_Hojas(Arbol^.izq)+N_Hojas(Arbol^.der)
  END;

BEGIN
  IF Arbol=NIL THEN
    Hojas:=0
  ELSE
    IF Arbol^.Clave:=Clave THEN
      Hojas:=N_Hojas(Arbol^.izq)+N_Hojas(Arbol^.dcha)
    ELSE
      IF Arbol^.der=NIL AND Arbol^.izq=NIL THEN
        BEGIN
          Aux:=Arbol;
          Arbol:=NIL;
          DISPOSE(Aux);
          Hojas:=0
        END
      ELSE
        Hojas:=Hojas(Arbol^.izq,Clave) + Hojas(Arbol^.der,Clave)
END;
```