ALGORITMOS Y PROGRAMAS:
TEMA 1:

1. **Sistemas de procesamiento de la información.**
2. **Concepto de algoritmo.**
3. **Lenguaje de programación.**
4. **Datos, tipos de datos y operaciones primitivas.**
5. **Constantes y variables.**
6. **Expresiones: tipos y operadores.**
7. **Funciones internas.**
8. **La operación de asignación.**
9. **Entrada y salida de la información.**

1. SISTEMAS DE PROCESAMIENTO DE LA INFORMACIÓN:

Un ordenador es una máquina de procesamiento de información. Es una máquina porque tiene cables, chips,... , procesa porque es capaz de procesar cosas, e información porque maneja conjuntos ordenados de datos).

Para procesar la información está el hardware (microprocesador, RAM,...), y el software (que sirve para manejar el hardware).

2. CONCEPTO DE ALGORITMO:

El algoritmo trata de resolver problemas mediante programas.

Fases:

- **Análisis preliminar o evaluación del problema: Estudiar el problema en general y ver que parte nos interesa.**
- **Definición o análisis del problema: Ver que es lo que entra y que es lo que sale, las posibles condiciones o restricciones, ...**
- **Diseño del algoritmo: Diseñar la solución.**
- **El programa: Codificación del algoritmo en un lenguaje de programación.**
- **Ejecución del programa y las pruebas: Ver si el programa hace lo que queríamos.**

¿Qué es un algoritmo?:

Es una formula para resolver un problema. Es un conjunto de acciones o secuencia de operaciones que ejecutadas en un determinado orden resuelven el problema. Existen n algoritmos, hay que coger el más efectivo.

Características:

- **Tiene que ser preciso.**
- **Tiene que estar bien definido.**
- **Tiene que ser finito.**

La programación es adaptar el algoritmo al ordenador.

El algoritmo es independiente según donde lo implemente.

3. EL LENGUAJE DE PROGRAMACIÓN:

Existen diferentes tipos, de bajo nivel y de alto nivel.

Instrucciones en una computadora y sus tipos:

Una instrucción es cada paso de un algoritmo, pero que lo ejecuta el ordenador. Un programa es un conjunto de instrucciones que ejecutadas ordenadamente resuelven un problema.

Tipos de instrucciones:

- **E/S: Pasar información del exterior al interior del ordenador y al revés.**
- **Aritmético-lógicas: Aritméticas: +,-,*,... ; Lógicas: or, and, <, >, ...**
- **Selectivas: Permiten la selección de una alternativa en función de una condición.**
- **Repetitivas: Repetición de un número de instrucciones un número finito de veces.**

Tipos de lenguajes:

- **Lenguaje máquina: Todo se programa con 1 y 0, que es lo único que entiende el ordenador.**
  Ventaja: No necesita ser traducido.
  Inconveniente: La dificultad, la confusión, para corregir errores, es propia de cada máquina.
- **De bajo nivel o ensamblador: Se utilizan mnemotécnicos (abreviaturas).**
  Ventaja: No es tan difícil como el lenguaje máquina.
  Inconvenientes: Cada máquina tiene su propio lenguaje, necesitamos un proceso de traducción.

- **El programa escrito en ensamblador se llama programa fuente y el programa que se obtiene al ensamblarlo se llama programa objeto.**
  - **Lenguajes de alto nivel: Los más cercanos al lenguaje humano.**

Ventaja: Son independientes de cada maquina (los compiladores aceptan las instrucciones estándar, pero también tienen instrucciones propias).

Inconveniente: El proceso de traducción es muy largo y ocupa más recursos. Aprovecha menos los recursos internos.

Proceso de traducción y ejecución de un programa escrito en un lenguaje a alto nivel:

Usamos un editor y obtenemos el programa fuente, y el compilador es el que traduce el programa al lenguaje máquina. El compilador internamente ha sido diseñado para traducir.

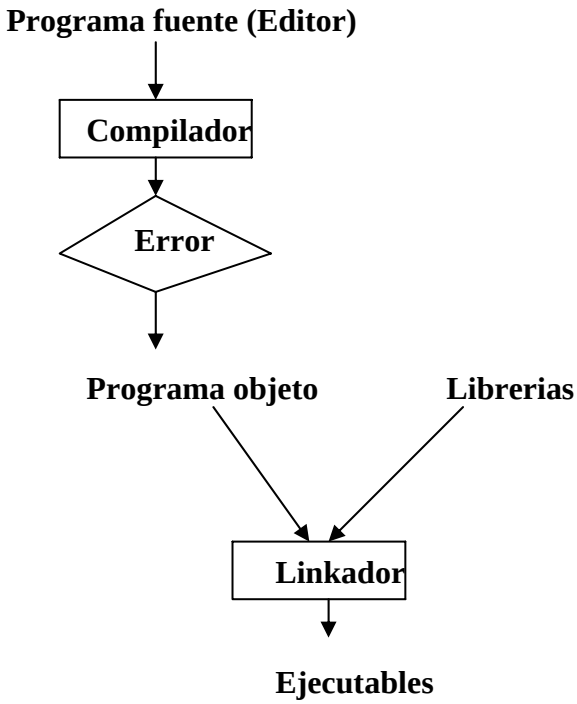
El compilador obtiene el programa o el fichero objeto. El compilador tiene que buscar los errores.

Normalmente no sale un ejecutable, sino que necesita elementos, librerías, ...

Mediante un linkador juntamos el programa objeto y las librerías, y se forma un programa ejecutable.

Cuando se ejecuta el programa, el cargador lleva al programa a memoria para que éste pueda ser ejecutable.

Debbuger: Depura el programa ejecutándolo paso a paso, viendo la memoria paso a paso para encontrar el error.



Para traducir puedo utilizar el compilador o un interprete, con el compilador cojo todo el programa al completo y el interprete lee cada instrucción y lo va ejecutando.

El interprete es más rápido, pero menos eficiente.

Todos los lenguajes tienen compiladores, pero no todos tienen interpretes.

LISP (Lenguaje de inteligencia artificial) : Sólo tiene interpretes.

4. DATOS, TIPOS DE DATOS Y OPERACIONES PRIMITIVAS:

- Dato: Es un objeto o elemento que tratamos a lo largo de diversas operaciones.

Tienen 3 caracteristicas:

- **Un nombre que los diferencia del resto.**
- **Un tipo que nos determina las operaciones que podemos hacer con ese dato.**
- **Un valor que puede variar o no a lo largo de la operación.**

Existen diferentes tipos de datos.

- Caracteristicas de los tipos:

- **Cada tipo se representa o almacena de forma diferente en la computadora.**
  Bit:1/0; Byte=8 bits.
- **Un tipo agrupa a los valores que hacen las mismas operaciones.**
- **Si tiene definida una relación de orden es un tipo escalar.**
- **Cardinalidad de un tipo: Número de valores distintos que puede tomar un tipo.**
  Pueden ser finitos (caracteres), y si son infinitos el ordenador los toma como finitos porque esta limitado por el tamaño de los bytes en el que la cifra es almacenada.

- Los datos pueden ser:

- **Simples: Un elemento.**

- **Compuestos: Varios elementos.**

- Los tipos pueden ser:

- **Estandar: Que vienen en el sistema por defecto.**
- **No estandar: Son los que crea el usuario.**

- Los tipos simples más importantes son:

- **Numéricos.**
- **Lógicos.**
- **Caracteres.**

- **Numéricos:**
  - **Entero: Subconjunto finito del conjunto matemático de los numéros enteros. No tiene parte decimal. El rango de los valores depende del tamaño que se les da en memoria.**
  - **Real: Subconjunto finito del conjunto matemático de los números reales. Llevan signo y parte decimal. Se almacenan en 4 Bytes (dependiendo de los modificadores). Si se utilizan números reales muy grandes, se puede usar notación científica que se divide en mantisa, base y exponente; tal que el valor se obtiene multiplicando la mantisa por la base elevada al exponente.**
- **Lógicos o booleanos:**
  - **Aquel que sólo puede tomar uno de los dos valores, verdadero o falso (1/0).**
- **Carácter:**
  - **Abarca al conjunto finito y ordenado de caracteres que reconoce la computadora (letras, digitos, caracteres especiales, ASCII).**

Tipo de cadena o String: Conjunto de caracteres, que van a estar entre "". El propio lenguaje puede añadir más tipos, o se pueden añadir modificadores.

Entero : Int → Long int

En algunos lenguajes se definen tipos especiales de fecha y hora, sobre todo en los más modernos.

5. CONSTANTES Y VARIABLES:

- **Constantes: Tienen un valor fijo que se le da cuando se define la constante y que ya no puede ser modificado durante la ejecución.**
- **Variables: El valor puede cambiar durante la ejecución del algoritmo, pero nunca varia su nombre y su tipo.**

Antes de usar una variable hay que definirla o declararla, al hacerlo hay que dar su nombre y su tipo. El nombre que le damos tiene que ser un nombre significativo, va a ser un conjunto de caracteres que dependiendo del lenguaje hay restricciones. Tiene que empezar por una letra, y el tamaño depende del lenguaje.

Identificador: Palabra que no es propia del lenguaje.

El valor de la variable si al declararla no se la inicializa, en algunos lenguajes toma una por defecto. En cualquier caso el valor de la variable podemos darle uno incial o podemos ir variandolo a lo largo de la ejecución.

Las constantes pueden llevar asociados un nombre o no, si no lo llevan, se llaman literales. Su valor hay que darlo al definir la constante y ya no puede cambiar a lo largo de la ejecución, y en cuanto al tipo, dependiendo de los

lenguajes en algunos hay que ponerlo, y en otros no hace falta ponerlo porque toma el tipo del dato que se le asigna. → Const PI=3,1416.

Hay que inicializar todas las variables.

< > → Hay que poner algo obligatoriamente.

[ ] → Puede llevar algo o no llevarlo.

La ventaja de usar constantes con nombre es que en cualquier lugar donde quiera que vaya la constante, basta con poner su nombre y luego el compilador lo sustituira por su valor.

Las constantes sin nombres son de valor constante: 5, 6, 'a', "hola".

Cuando una cadena es de tipo carácter, se encierra entre '' → 'a'.

Relación entre variables y constantes en memoria:

Al detectar una variable o una constante con nombre, automaticamente se reserva en memoria espacio para guardar esa variable o constante. El espacio reservado depende del tipo de la variable.

En esa zona de memoria es en la que se guarda el valor asociado a la variable o constante y cuando el programa use esa variable, ira a esa zona de memoria a buscar su valor.

6. EXPRESIONES: TIPOS Y OPERADORES:

Una expresión es una combinación de constantes, variables, signos de operación, paréntesis y nombres especiales (nombres de funciones estandar), con un sentido unívoco y definido y de cuya evaluación resulta un único valor.

Toda expresion tiene asociada un tipo que se corresponde con el tipo del valor que devuelve la expresion cuando se evalua, por lo que habrá tantos tipos de expresiones como tipos de datos. Habra expresiones numéricas y lógicas.

Numéricas: Operadores aritméticos.

Son los que se utilizan en las expresiones numéricas (una combinación de variables y/o constantes numéricas con operadores aritméticos y que al evaluarla devuelve un valor numérico.

+, -, *, /.

Operación resto: Lo que devuelve es el resto de una división entera.

Mod: Pascal. → 5 mod 3 = 2

%: C.

División entera: Nos devuelve el cociente de una división entera (en la que no se sacan decimales).

Div: Pascal. → 5 div 3 = 1

\: C.

Potencia: ^ → 5^2.

Todos estos operadores son binarios (el operador se situa en medio), el menos tambien puede ser unario (lleva un único operando) y significa signo negativo.

Reglas de precedencia:

El problema es cuando una expresión entera según como la evalue pueda dar diferentes valores.

7*3+4 → 25

→ 49

La solución es aplicar prioridad entre los operadores, de modo que ante la posibilidad de usar varios siempre aplicaremos primero el de mayor prioridad.

Cada lenguaje puede establecer sus propias reglas de prioridad o precedencia de operadores. Si no nos acordamos, siempre podemos poner ( ).

1ª) ^
2ª) * / div mod
3ª) + -

Entre dos operaciones que tienen la misma precedencia para resolver la ambigüedad, hay que usar la regla de la asociatividad. La más normal es la de la asociatividad a izquierdas (primero lo de la izquieda).

<u>Expresiones lógicas: Operadores relacionales y lógicos.</u>

Una expresión lógica es aquella que sólo puede devolver dos valores (Verdadero o Falso). Los valores que pueden aparecer en una expresión lógica son de 2 tipos: lógicos y relacionales.

La particularidad de las expresiones lógicas es que mientras en una expresión numérica por devolver un valor numérico los operandos solo pueden ser números, en una expresión lógica los operandos no tienen porque ser booleanos aunque se devuelva un valor booleano. Esto es lo que ocurre cuando en la expresión lógica utilizamos operadores relacionales con lo cual se obtienen valores lógicos o booleanos a partir de otros que no lo son.

En cambio cuando los operadores son lógicos los operandos obligatoriamente tambien tienen que ser lógicos.

Operadores relacionales:

<
>
=
<> → en C: !=
≤
≥

< Operando1> operador < Operando2>
5 > 3 → Verdadero

¿Cómo se evalua una expresión relacional?:

- **Primero se evalua el primer operando y se sustituye por su valor.**
- **Luego se evalua el seguno operando y se sustituye por su valor.**
- **Finalmente se aplica el operador relacional y se devuelve el valor booleano correspondiente.**

((3*2)+1-5^2) < (2-1)
-18 < 1 → Verdadero.

El problema del tipo real:

Desde la informática, los numeros reales son finitos, ya que tenemos un máximo de cifras decimales. Cuando trabajamos con un =, matematicamente da un valor exacto, pero informaticamente no es exacto.

1/5 * 5 =1
1.0/5.0 *5.0 <>1.0

Soluciones:

- **La que nos aporte el propio lenguaje de programación. Se considera un valor de error.**
- **Trabajar con números enteros siempre que sea posible.**
- **Utilizar las comparaciones <> en vez de ≤ ≥ si es posible.**
- **Si hay que preguntar por igual, cambiar la condición utilizando valores absolutos y un valor mínimo de error.**

  Si la diferencia < 0.00000001 y ABS (A-B)< min ; son iguales.

Operadores lógicos:

El problema es que a veces queremos preguntar o evaluar por más de una condición al mismo tiempo y para esto estan los operadores lógicos.

Y → and → &&

O → or → !!

No → not → ~!

Y, O, son operadores binarios (necesitan 2 operandos de tipo logico).
Operando 1 Operador Operando 2

El No es unario y se coloca primero el No, y despues el operando.

El resultado es lógico y depende de:

Operando 1	Operando 2	AND	OR
V	V	V	V
V	F	F	V
F	V	F	V
F	F	F	F

El No niega :

NOT	1/0
V	F
F	V

Prioridades de los operadores:

- **Lo más prioritario es el NOT**
- **Luego el Y y el O.**
- **<, > =,...**

Tabla de prioridades:

^ NO
/ div mod y
+ - O
<, >, =, <>,...

7. FUNCIONES INTERNAS:

Son funciones matemáticas diferentes de las operaciones básicas pero que se incorporan al lenguaje y que se consideran estandar. Dependen del lenguaje. Normalmente se encuentran en la librería de matemáticas del lenguaje de programación.

Fórmulas:

Abs (x)
Arctan (x)
Cos (x)
Sen (x)
Exp (x)
Ln (x)
Log 10 (x)
Redondeo (x)
Trunc (x)
Cuadrado (x)
Raiz (x)

8. OPERACIÓN DE ASIGNACIÓN:

Consiste en atribuir un valor a una variable. El valor sera una expresión (constante, variable,...).

← Otros lenguajes = , :=

Variable a la que se le asigna el valor ← el valor que le vamos a asignar.

A ← 5

En C A = = B Comparación
A = B Asignación
En Pascal A:=B Asignación
A=B Comparación

El proceso de asignacion se realiza en 2 fases:

- **Se evalúa la expresión de la parte derecha de la asignación obteniendose un único valor.**
- **Se asigna ese valor a la variable de la parte izquierda.**

¿Qué es lo que hay que tener en cuenta?:

- **En la parte izquierda sólo puede haber una variable.**
- **La variable a la que se le asigna el valor pierde su valor anterior.**
- **La variable que aparece en la derecha ya que como se evalua primero la de la derecha cuando se tenga que evaluar el valor de esa variable se tomara su valor antiguo.**
- **EL TIPO DEL VALOR QUE SE OBTIENE AL EVALUAR LA PARTE DERECHA TIENE QUE SER EL MISMO QUE EL TIPO DE LA VARIABLE DE LA PARTE IZQUIERDA, ES DECIR A UNA VARIABLE SOLO SE LE PUEDEN DAR VALORES DE SU MISMO TIPO.**
  - **En Pascal da un error.**
  - **En C, no da error porque el compilador trunca el numero.**

A: entero
A ← 2
A ← 3*A + A = 8

9. ENTRADA Y SALIDA DE LA INFORMACIÓN:

Las dos operaciones básicas de cada salida son las de lectura y de escritura. La lectura es equivalente a la asignación en cuanto que va a haber una variable que recibe un valor, pero este valor no resulta de evaluar ninguna expresión, sino que el valor lo vamos a leer de un dispositivo externo de entrada.

Leer (nombre de la variable)

El valor introducido por el dispositivo externo, tiene que ser del mismo tipo del que la variable que se le asigne.

La operación de escritura lo que hace es mostrar el valor de una variable en un dispositivo externo de salida.

Escribir (variable)

La operación de escritura no es una operación destructivo en memoria.

Al pedir un valor al usuario hay que decirle que es lo que se le quiere pedir escribiendo un mensaje.

EJERCICIOS: TEMA 1

1. A ← (3*2^5 mod 1 + 8*(3-5) < (2+8-1 mod 1)
A ← (3*32 mod 1 + (-16)) < 10
A ← -16 < 10
A ← Verdadero

1.1. A ← A o (3+5*8) < 3 y ((-6/3 div 4)*2 < 2)
A ← Verdadero o 43 < 3 y (0*2 < 2)
A ← Verdadero o Falso y Verdadero
A ← Verdadero o Falso
A ← Verdadero

2. B ← 3 mod 2 div 3
B ← 1 div 3
B ← 0

2.1. C ← (-B*2 <> 8*3 mod 4) y ('A'>'B')
C ← (0 <> 24 mod 4) y Falso
C ← Falso y Falso
C ← Falso

2.2. A ← C o no (3=5) y (8 <> 3+B)
A ← Falso o Verdadero y Verdadero
A ← Falso o Verdadero
A ← Verdadero

RESOLUCIÓN DE PROBLEMAS CON COMPUTADORA Y HERRAMIENTAS DE PROGRAMACIÓN:
TEMA 2

1. **Resolución de problemas.**
2. **Análisis del problema.**
3. **Diseño del algoritmo.**
4. **Resolución en la computadora.**
5. **Flujogramas.**
6. **Diagramas NS o de NASSI-SCHEDERMAN**
7. **Pseudocódigo.**

1. RESOLUCIÓN DE PROBLEMAS:

La resolución de un problema desde el punto de vista algorítmico tiene 3 fases:

- **Análisis del problema: Comprensión.**
- **Diseño del algoritmo: Resolución algorítmica.**
- **Resolución en computadora: Implantación del algoritmo en un lenguaje de programación.**

2. ANÁLISIS DEL PROBLEMA:

El objetivo de ésta fase es comprender el problema para lo cual como resultado tenemos que obtener la especificación de las entradas y salidas del problema. Tiene que quedar claro que entra y que sale.

3. DISEÑO DEL ALGORITMO:

Una vez comprendido el problema se trata de determinar que pasos o acciones tenemos que realizar para resolverlo.

Como criterios a seguir a la hora de dar la solución algorítmica hay que tener en cuenta:

1. **Si el problema es bastante complicado lo mejor es dividirlo en partes más pequeñas e intentar dividirlo en partes más pequeñas e intentar resolverlas por separado. Esta metodología de "divide y vencerás" también se conoce con el nombre de diseño descendente.**
2. **Las ventajas de aplicar esto son:**

- **Al dividir el problema en módulos o partes se comprende más fácilmente.**
- **Al hacer modificaciones es más fácil sobre un módulo en particular que en todo el algoritmo.**
- **En cuanto a los resultados, se probarán mucho mejor comprobando si cada módulo da el resultado correcto que si intentamos probar de un golpe todo el programa porque si se produce un error sabemos en que módulo ha sido.**

Una segunda filosofía a la hora de diseñar algoritmos es el refinamiento por pasos, y es partir de una idea general e ir concretando cada vez más esa descripción hasta que tengamos algo tan concreto para resolver. Pasamos de lo más complejo a lo más simple.

La representación de los algoritmos:

Una vez que tenemos la solución hay que implementarla con alguna representación. Las representaciones más usadas son los flujogramas, los diagramas NS y el pseudocódigo.

También la solución se puede escribir en algunos casos en lenguaje natural pero no se hace porque es muy ambiguo, e incluso otras formas de expresión como fórmulas matemáticas.

Escritura del algoritmo:

Al escribir el algoritmo hay que tener en cuenta:

- **Las acciones o pasos a realizar tienen que tener un determinado orden.**
- **En cada momento solo se puede ejecutar una acción.**
- **Dentro de las sentencias del algoritmo pueden existir palabras reservadas (palabras propias del lenguaje de programación que tienen para el compilador un determinado significado).**
- **Si estamos utilizando pseudocódigo tenemos también que usar la identación (aumenta la legibilidad del problema para que se pueda leer mejor).**

4. RESOLUCIÓN EN LA COMPUTADORA:

Es hacer entender nuestro algoritmo a la computadora para que lo pueda hacer.

1. **Codificamos el algoritmo en un leguaje de programación.**
2. **Ejecutar el programa antes compilado.**
3. **Comprobar los resultados y si no funciona, corregirlo.**

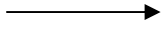
5. FLUJOGRAMAS:

Es una notación gráfica para implementar algoritmos. Se basa en la utilización de unos símbolos gráficos que denominamos cajas, en las que escribimos las acciones que tiene que realizar el algoritmo.

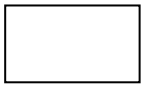
Las cajas están conectadas entre sí por líneas y eso nos indica el orden en el que tenemos que ejecutar las acciones.

En todo algoritmo siempre habrá una caja de inicio y otra de fin, para el principio y final del algoritmo.

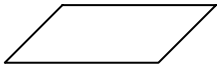
Los símbolos:



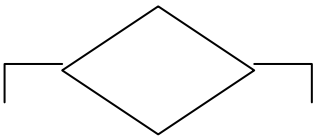
Líneas de flujo: Una línea con una flecha que sirve para conectar los símbolos del diagrama y la flecha indica la secuencia en la que se van a ejecutar las acciones.



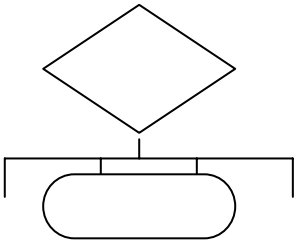
Símbolo de proceso: Indica la acción que tiene que realizar la computadora. Dentro escribimos la acción.



Representa las acciones de entrada y salida. Dentro colocaremos las acciones de lectura y escritura.



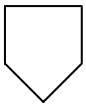
Condición: Dentro se va a colocar una condición. Sirve para representar estructuras selectivas y repetitivas y lo que se hace al encontrar ese signo es evaluar la condición que hay dentro tal que según la condición sea verdadera o falsa iremos por caminos distintos.



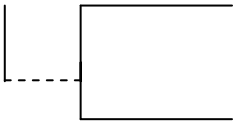
Principio y fin: Dentro del símbolo ira la palabra inicio o fin del algoritmo.

Subprograma: Dentro se coloca el nombre del subprograma al que se llama.

Conectores: Nos sirven cuando un flujograma no me cabe en una columna de la hoja y hay que seguir en otra columna:

- **Si es en la misma hoja:** 
- **Si es en hoja distinta:** 

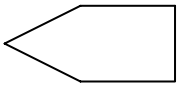
Los conectores se ponen uno donde termina la columna y otra donde empieza.



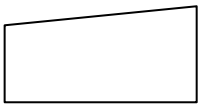
Es una aclaración para entender mejor el código, pero no es parte del código, no se ejecuta.

<u>Otros símbolos:</u>

- Pantalla: Cuando una salida es por pantalla.

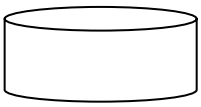


- Teclado: Para representar una entrada por teclado.



- Impresora:

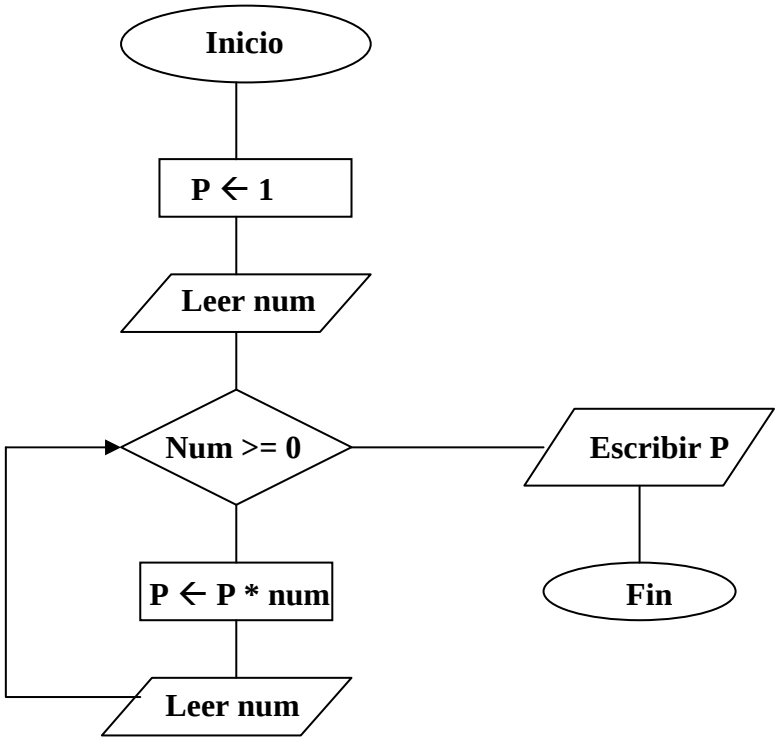
- Entrada/Salida por disco:



***Problema:**

Queremos hallar el producto de varios números positivos introducidos por teclado y el proceso termina cuando se meta un número negativo.

1. **Iniciar la variable del producto.**
2. **Leer el primer número.**
3. **Preguntar si es negativo o positivo.**
4. **Si es negativo nos salimos y escribimos el producto.**
5. **Si es positivo, multiplicamos el número leído y luego leemos un nuevo número, y se vuelve al paso 3.**

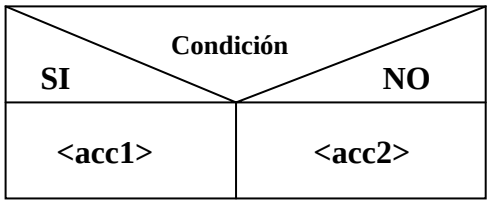


5. DIAGRAMAS N-S O DE NASSI-SCHEDERMAN:

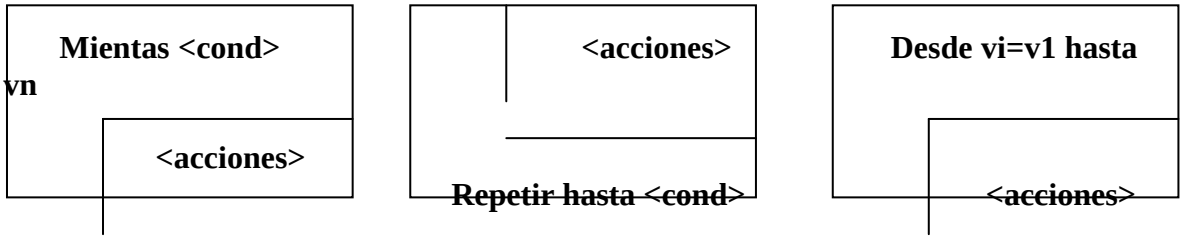
Es semejante al flujograma, peor sin flechas y cmabiando algo los símbolos de condición y repetición. Las cajas van unidas.

<acción>

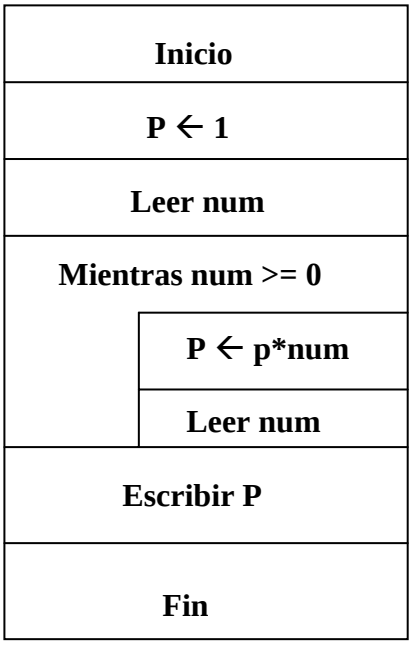
Condiciones:



Repetitivas:



Problema anterior:



6. PSEUDOCÓDIGO:

Es un lenguaje de especificación de algoritmos, pero muy parecido a cualquier lenguaje de programación, por lo que luego su traducción al lenguaje es muy sencillo, pero con la ventaja de que no se rige por las normas de un lenguaje en particular. Nos centramos más en la lógica del problema.

El pseudocódigo también va a utilizar una serie de palabras clave o palabras especiales que va indicando lo que significa el algoritmo.

1. **Inicio y Fin: Por donde empieza y acaba el algoritmo.**
   Begin /end : Pacal.
   { } : En C.
2. **Sí <cond>**
   Entonces <acc1> → If then else
   Sino <acc2>

3.Mientras <cond> /hacer → while do

4. Repetir / hasta → repeat until

5. Desde /hasta → for .. to

6. **Según sea → Case**
   Swith

- **Los comentarios van encerrados entre llaves.**
- **Hay que utilizar la identación.**

```
Algoritmo <nombre alg>
Var
  <nombre>: <tipo>
Inicio
  <Instrucciones>
Fin
```

```
Algoritmo Producto
Var
  P, num: entero
Inicio
  P ← 1
  Leer num
  Mientras num >=0 hacer
    P ← p*num
    Leer num
  Fin mientras
```

```
  Escribir p
Fin
```

EJERCICIOS: TEMA 2

1. Realizar un programa que convierta los grados a radianes.

```
Algoritmo convertir
Var
  Grados, rad: real
Inicio
  Escribir "Introduce los grados"
  Leer grados
  Si grados >= 360
    Entonces grados ← grados mod 360
  Fin si
  Rad ← grados / 180
  Escribir rad " Π radiantes"
Fin
```

2. Realizar un algoritmo que pida un valor entero que equivale a un número de duros y me calcule a cuantos billetes de 5000, 1000, monedas de 200, 25, 1.

```
Algoritmo cambio
Var
  Duros: real
Inicio
  Escribir "Introduce los duros"
  Leer duros
  Duros ← duros * 5
  Escribir duros div 5000 "billetes de 5000"
  Duros ← duros mod 5000
  Escribir duros div 1000 "billetes de 1000"
  Duros ← duros mod 1000
  Escribir duros div 200 "monedas de 200"
  Duros ← duros mod 200
  Escribir duros div 25 "monedas de 25"
  Duros ← duros mod 25
  Escribir duros "monedas de 1"
Fin
```

3. Realizar un programa que pida al usuario la velocidad en m/s y el radio de la circunferencia de la pista, y resultada el programa devuelve el tiempo que tarda el atleta en dar 2 vueltas a la pista, sabiendo que el atleta descansa 1 minuto cada 1000 metros.

```
Algoritmo recorrido
Var
  Velocidad,radio,tiempo,longitud: entero
Inicio
  Escribir “Introduce la velocidad”
  Leer velocidad
  Escribir “Introduce el radio”
  Leer radio
  Longitud ← 4 * 3.1416 * radio
  Descanso ← longitud div 1000
  Tiempo ← longitud div velocidad + descanso * 60
  Escribir tiempo
Fin
```

ESTRUCTURA GENERAL DE UN PROGRAMA
TEMA 3

1. **Concepto de programa.**
2. **Instrucciones y tipos.**
3. **Elementos básicos de un programa.**
4. **Estructura de algoritmos y programas.**

1. CONCEPTO DE PROGRAMA:

Un programa es un conjunto de instrucciones que al ser ejecutadas resuelven un problema.

Un programa tiene 3 partes:

1. **Entrada de datos: Normalmente se va a ejecutar a través de instrucciones de lectura, y en lo que se le pide al usuario la información que el programa va a necesitar para ejecutarse y se hace a través de lecturas.**
2. **Acciones de un algoritmo: Parte en la que se resuelve el problema usando los datos de entrada.**
3. **Salida: Mostrar en un dispositivo de salida los resultados de las acciones anteriormente realizadas. Son acciones de escritura.**

En la parte de las acciones a ejecutar se distinguirán dos partes:

- **Declaración de variables.**
- **Instrucciones del programa.**

2. INSTRUCCIONES Y TIPOS:

Para que una instrucción se ejecute tiene que ser llevada a memoria. En cuanto al orden de ejecución de las instrucciones, el programa puede ser de dos tipos:

- **Programas lineales: Se va ejecutando una instrucción más otra y el orden de ejecución es igual al orden de escritura.**
- **Programas no lineales: Las instrucciones no se ejecutan en el mismo orden en el que aparecen escritas, sino que se realizan saltos que nos mandan de unas instrucciones a otras.**
  - **Nunca se deben hacer saltos no lineales.**

Tipos de instrucciones:

1. **Inicio y fin.**
2. **Asignación: Dar un valor a una variable.**
3. **Lectura / escritura: Introducir o sacar información por dispositivos E/S.**
4. **Instrucciones de bifurcación: Alternan el orden de ejecución del programa. Salto a otra instrucción que no es la siguiente.**

4.1. Bifurcación incondicional: El salto se produce siempre que el programa vaya a esa instrucción: Goto → Ir a.

4.2. Bifurcación condicional: Se ejecutan un conjunto de instrucciones u otras dependiendo del valor devuelto al evaluar una condición.
Es la que vamos a usar.

3. ELEMENTOS BÁSICOS DE UN PROGRAMA:

¿Qué es la sintaxis de un lenguaje?:

Conjunto de reglas que tenemos que seguir a la hora de escribir un programa en ese lenguaje tal que si no seguimos esas reglas de sintaxis el compilador da errores.

Elementos del lenguaje de programación:

1. **Palabras reservadas: Son un conjunto de palabras especiales que nos sirven para definir la estructura del programa, y solo se pueden usar para el fin para el que están reservadas.**
2. **Identificadores: Son los nombres que aparecen en el programa dados por el usuario. Son por tanto los nombres de variables, de constantes, de subprogramas y nombres de tipos creados por el usuario.**
3. **Caracteres especiales: Sirven como separadores entre sentencias, por ejemplo el ;.**
4. **Instrucciones: De 3 tipos, secuenciales, repetitivas y selectivas, y pueden aparecer elementos especiales (bucles, contadores, interruptores y acumuladores).**
    - **Bucle: Un conjunto de instrucciones que se repiten un número finito de veces. Lleva asociado aparte de las instrucciones una condición que es la que determina cuando se termina un bucle. Ejecución de un bucle (iteración). Los bucles se pueden anidar unos dentro de otros, y puede haber varios bucles al mismo nivel, pero nunca se entrelazan.**
    - **Contador: Un elemento cuyo valor se incrementa o decrementa en un valor constante en cada iteración de un bucle, y se utiliza para controlar la condición del bucle.**
    - **Acumulador: Es una variable que también se suele usar en los bucles y que se incrementa o decrementa en cada iteración del bucle, pero no en una cantidad constante.**

```
Algoritmo ejemplo
Var cont, num, sum: entero
Inicio
  Cont ← 0
  Sum ← 0
  Mientras cont <> 3
    Leer num
    Sum ← sum + num
    Cont ← cont +1
  Fin mientras
  Escribir suma
End
```

- **Interruptor (marca, bandera o flag): Es una variable que sirve como indicador de una determinada información y que solo puede tomar uno de dos valores. El valor de la variable tiene asociado un signo y puede variar a lo largo de la ejecución.**

```
Algoritmo ejemplo
Var cont, num, suma: entero
    Neg: boolean
Inicio
 Cont ← 0
 Sum ← 0
 Neg ← falso
 Mientras cont <>3
  Leer num
  Si num < 0
   Entonces neg ← verdadero
  Fin si
  Sum ← sum + num
  Cont ← cont + 1
 Fin mientras
 Si neg=verdadero
  Entonces escribir ("Se ha leído negativos")
  Sino escribir ("No negativos")
 Fin si
Fin
```

Si es leer un número negativo o hasta 3 números:
Mientras (cont <> 3) y (neg = verdadero)

4. ESCRITURA DE ALGORITMOS Y PROGRAMAS:

En pseudocódigo el programa tiene dos partes, la cabecera y el cuerpo. La cabecera contiene el nombre del algoritmo, y el cuerpo contiene 2 partes.

La primera es la zona de declaraciones de var y const, y la segunda es la zona de las instrucciones del programa.

En la zona de instrucciones para que quede más legible hay que usar la identación y si es necesario hay que usar comentarios entre llaves.

EJERCICIOS: TEMA 3

1. ¿Cuáles y cuántos son los números primos comprendidos entre 1 y 1000?

```
Algoritmo n_primos
Const
  Primero=1
  Limite=1000
Var
  cont, i, j: entero
  primo: boolean
Inicio
  Cont ← 0
  Desde i= primero hasta limite
    primo ← verdadero
    j ← 2
    mientras (i>j) y (primo =verdadero)
      Si i mod j = 0
        Entonces  primo ← falso
        Sino j ← j + 1
      Fin si
    Fin mientras
    Si primo = verdadero
      Entonces  escribir i" "
                Cont ← cont + 1
    Fin si
  Fin desde
  Escribir "Entre "primero" y "limite" hay "cont" nº primos"
Fin
```

2. Calcular el máximo de números positivos introducidos por teclado, sabiendo que metemos números hasta que introduzcamos uno negativo. El negativo no cuenta.

```
Algoritmo maximo
Var
```

```
  Num, max: entero
Inicio
  Max ← 0
  Escribir "Introduzca nº positivos y para acabar introduzca uno negativo"
  Leer num
  Mientras num >=0
    Si num >max
      Entonces  max ← num
    Fin si
    Leer num
  Fin mientras
  Escribir "El mayor número es" max
Fin
```

3. Determinar cuales son los múltiplos de 5 comprendidos entre 1 y N.

```
Algoritmo multiplos
Var
  i: entero
Inicio
  Desde i=1 hasta n
    Si i mod 5 =0
      Entonces escribir i
    Fin si
  Fin desde
Fin
```

INTRODUCCIÓN A LA PROGRAMACIÓN ESTRUCTURADA: TEMA 4:

1. **Técnicas de programación.**
2. **Programación modular.**
3. **Programación constructora.**
4. **Estructura secuencial.**
5. **Estructuras selectivas.**
6. **Estructuras repetitivas.**
7. **Anidación de bucles y condicionales.**
8. **Control de datos de entrada.**

1. TÉCNICAS DE PROGRAMACIÓN:

El programar con flujogramas o diagramas NS resulta muy lioso en el momento en que el programa se complica, por eso vamos a utilizar siempre el pseudocódigo, en el que vamos a utilizar dos técnicas de programación que no se usan por separado, sino que son complementarios.

Estas técnicas son:

- **Programación modular: Consiste en dividir el programa en partes llamadas módulos, e implementar cada uno de esos módulos por separado.**
- **Programación estructurada: Cuyo objetivo es hacer más legible y lógico la estructura del programa utilizando para ello solamente tres tipos de estructuras: selectivas, secuenciales (condicionales) y repetitivas.**

2. PROGRAMACIÓN MODULAR:

Se basa en dividir el programa en partes llamadas módulos, que se analizan y codifican de forma independiente y que realizan una determinada tarea que será en realidad una parte del problema total a resolver.

En todo algoritmo o programa existirá un módulo o programa principal que es al que transfiere el control cuando comienza la ejecución del programa, y luego desde él, se va llamando al resto de los subprogramas.

Llamar a un subprograma quiere decir transferirle el control, es decir que se comienza a ejecutar y lo hará desde el comienzo del subprograma hasta que se

termine y al terminar devuelve el control al subprograma que lo llamó. Dentro del subprograma a su vez también puedo llamar a otros pero en cualquier caso al final se devuelve el control al módulo que hace la llamada. El control al programa llamado se devuelve a la instrucción siguiente a la que se hizo la llamada.

Un módulo solo tiene acceso a los módulos a los que llama y a los submódulos a los que a su vez llaman éstos. Los resultados producidos por un módulo pueden ser utilizados por otros.

No existe un criterio fijo para determinar el tamaño, ni muy grandes ni muy pequeños, la idea fundamental es que realicen una única cosa y muy concreta.

Los módulos o subprogramas reciben diferentes nombres según el lenguaje de programación y según su tipo. Se llaman procedimientos y funciones (Pascal, C), subrutinas (basic, fortran), secciones (cobol),...

3. PROGRAMACIÓN ESTRUCTURADA:

La característica fundamental es que se va a basar en el uso únicamente de tres estructuras de control. Para ello se apoya en las siguientes filosofías:

1. **Recursos abstractos: Son los recursos con los que no contamos a la hora de programar, pero en los que nos apoyamos a la hora de solucionarlos. Estos recursos se tienen que ir transformando en recursos concretos.**
2. **Diseño descendente (top down): Se trata de ir descomponiendo el problema en niveles o pasos cada vez más sencillos, tal que la salida de una etapa va a servir como entrada de la siguiente. En las primeras etapas tomamos el punto de vista externo, es decir, que entradas hay y que salidas hay, y a medida que vamos bajando de nivel, lo vamos viendo de modo interno (como lo hace por dentro).**
3. **Estructuras básicas de control: Para hacer cualquier programa siguiendo los anteriores pasos de razonamiento, al final codificamos el programa usando tres tipos de secuencias (repetitivas, alternativas y secuenciales).**

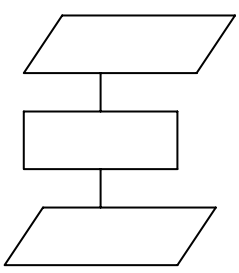
Al final todo programa va a tener una única entrada y una única salida.

Desde la entrada tienen que existir caminos que nos permiten pasar por todas las partes del programa y llevarnos a la salida, y finalmente no se van a permitir los bucles infinitos.

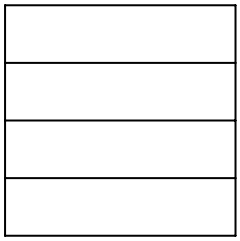
4. ESTRUCTURA SECUENCIAL:

Es cuando una instrucción sigue a otra en secuencia, es decir, la salida de una instrucción es la entrada de la siguiente.

FLUJOGRAMA:



DIAGRAMAS NS:



PSEUDOCÓDIGO:

Leer num
Num ← num*2
Escribir num

5. ESTRUCTURAS SELECTIVAS:

Se evalúa la condición y en función del resultado se ejecuta un conjunto de instrucciones u otro. Hay tres tipos de selectivas (simple, doble o múltiple):

*** Simple: Es la estructura :**

```
Sí <cond>
   entonces <acciones>
fin sí
```

Evaluamos la condición y si es verdadera ejecutamos el conjunto de condiciones asociadas al entonces, y si es falso, no hacemos nada.

FLUJOGRAMA:

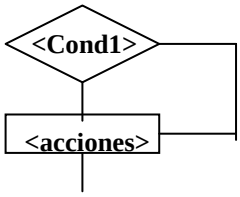
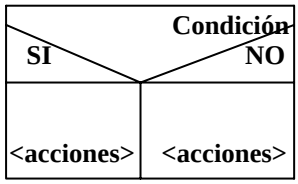
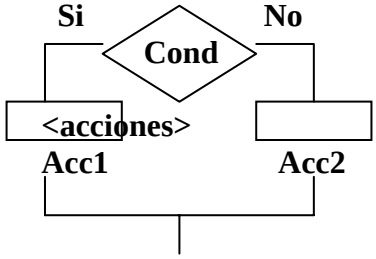


DIAGRAMA NS:

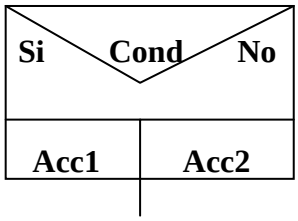


*** Doble: Se evalúa la condición y si es verdad se ejecutan el conjunto de acciones asociadas a la parte entonces, y si es falso se ejecutan el conjunto de acciones asociadas a la parte sino.**

FLUJOGRAMA:



DIAGRAMAS NS:



PSEUDOCÓDIGO:

```
Sí <cond>
   Entonces
   Sino <acciones>
Fin si
```

Una condición se ejecuta una única vez.

*** Alternativa múltiple: Se evalúa una condición o expresión que puede tomar n valores. Según el valor que la expresión tenga en cada momento se ejecutan las acciones correspondientes al valor. En realidad equivale a un conjunto de condiciones anidadas. En cualquier lenguaje, es Case o Swith.**

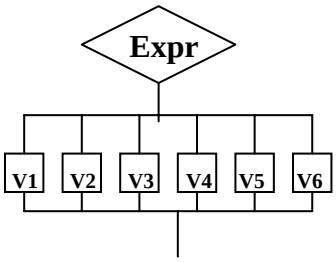
PSEUDOCÓDIGO:

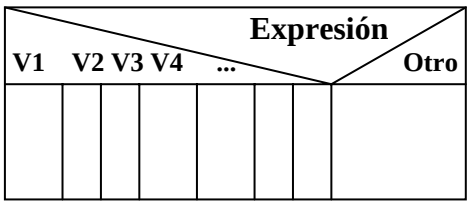
```
Según sea <expresión>
   <Valor1>: <acción1>
   <valor2>: <acción2>
   .....
   [<otro>: <acciones>]
fin según
```

- **Otro: Las acciones asociadas al valor otro se ejecutan cuando la expresión no toma ninguno de los valores que aparecen antes. Otherwise, Else.**

El valor con el que se compara la expresión, va a depender de los lenguajes, de lo que sea ese valor. En general ese valor puede ser un valor constante, un rango de valores o incluso otra condición

FLUJOGRAMA: **DIGRAMAS NS:**





Hacer un programa que pueda dibujar una recta, un punto o un rectángulo.
Algoritmo dibujo
Var op: carácter
Escribir ("Introduce una opción"

1. **Punto**
2. **Recta**
3. **Rectángulo")**

Leer op
Según sea op
"1": leer x
.....
"2": leer x
.....
"3": leer x
.....
"otro": escribir "opción errónea"
fin según

Para un rango de valores:
Leer una nota y escribir en pantalla la calificación:
Var nota: entero
Leer nota
Según sea nota
1..4: escribir "suspenso"
5..6: escribir "aprobado"
7..8: escribir "Notable"
9: escribir "Sobresaliente"
10: escribir "Matricula de honor"
fin según

En algunos lenguajes se permite poner una condición:
Según sea nota

Nota >=1 y nota <=4: escribir "suspenso"

En pseudocódigo no se pueden poner condiciones.

6. ESTRUCTURAS REPETITIVAS:

En aquella que contiene un bucle (conjunto de instrucciones que se repiten un número finito de veces). Cada repetición del bucle se llama iteración. Todo bucle tiene que llevar asociada una condición, que es la que va a determinar cuando se repite el bucle.

Hay cuatro tipos de bucles, aunque solo se pueden usar tres:

1. **Mientras hacer → While do**
2. **Repetir hasta → repeat until**
3. **Desde → for**
4. **Iterar → loop : No se usa.**

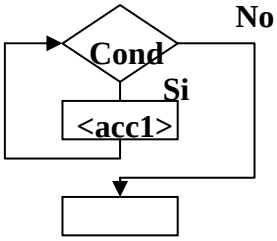
Mientras hacer:

Sintaxis:

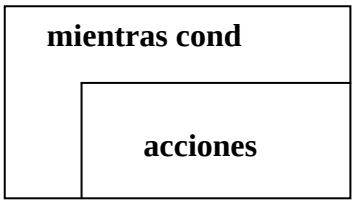
PSEUDOCÓDIGO:

```
Mientras <cond> hacer
        <acciones>
fin mientras
```

FLUJOGRAMA:



DIAGRAMAS NS:



Funcionamiento:

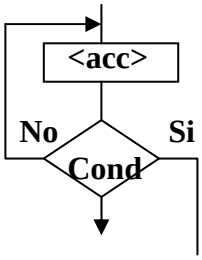
La condición del bucle se evalúa al principio, antes de entrar en él. Si la condición es verdadera, comenzamos a ejecutar las acciones del bucle y después de la ultima volvemos a preguntar pro la condición. En el momento en el que la condición sea falsa nos salimos del bucle y ejecutamos la siguiente condición al bucle.

Al evaluarse la condición antes de entrar en el bucle al principio, si la condición al ser evaluada la primera vez es falsa, no entraremos nunca en el bucle, el bucle puede que se ejecute 0 veces, por tanto usaremos obligatoriamente este tipo de bucle en el caso de que exista la posibilidad de que el bucle pueda ejecutarse 0 veces.

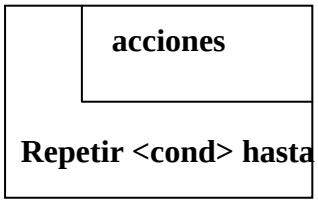
Repetir hasta:

Sintaxis:

FLUJOGRAMA:

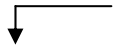


DIAGRAMAS NS:



FLUJOGRAMA:

```
Repetir
        <acciones>
hasta <condición>
```



Función:

Se repite el bucle hasta que la condición sea verdadera. Se repite mientras la condición sea falsa. La condición se evalúa siempre al final del bucle, si es falsa volvemos a ejecutar las acciones, si es verdad se sale del bucle.

Como la condición se evalúa al final, incluso aunque la primera vez ya sea verdadera, habremos pasado al menos una vez por el bucle.

Es decir que cuando un bucle se tenga que ejecutar como mínimo una vez, podremos usar una estructura repetir o mientras, la única diferencia que habrá entre las dos, es que para hacer lo mismo, las condiciones tienen que ser contrarias.

Leer 3 números y dar su suma:

```
Cont ← 0
Suma ← 0
Mientras cont <> 3
        Suma ← suma + num
        Leer num
        Cont ← cont + 1
Fin mientras
```

```
Cont ← 0
suma ← 0
repetir
                leer num
        suma ← suma + num
        cont ← cont + 1
hasta cont = 3
```

<u>Desde:</u>

Este tipo de bucles se utiliza cuando se sabe ya antes de ejecutar el bucle el número exacto de veces que hay que ejecutarlo. Para ello el bucle llevara asociado una variable que denominamos variable índice, a la que le asignamos un valor inicial y determinamos cual va a ser su valor final y además se va a incrementar o decrementar en cada iteración de bucle en un valor constante, pero esto se va a hacer de manera automática, el programador no se tiene que ocupar de incrementar o decrementar esta variable en cada iteración, sino que va a ser una operación implícita (lo hace por defecto).

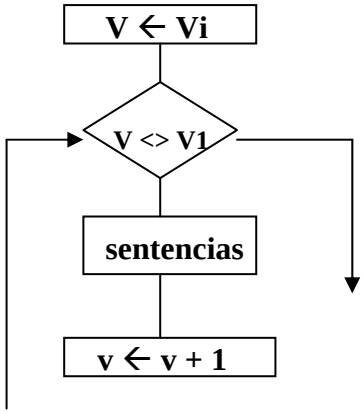
Por tanto en cada iteración del bucle, la variable índice se actualiza automáticamente y cuando alcanza el valor que hemos puesto como final se termina la ejecución del bucle.

Sintaxis:

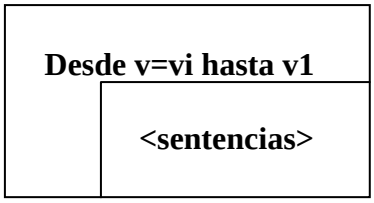
PSEUDOCÓDIGO:

```
Desde <var índice>=<valor inicial> hasta <valor final>
        <acciones>
fin desde
```

FLUJOGRAMAS:



DIAGRAMAS NS:



Bucle con salida interna: loop → iterar.

Permite la salida del bucle desde un punto intermedio del mismo siempre que se cumpla la condicion que aparece, entonces nos salimos a la siguiente instrucción del bucle.

```
Iterar
  <sent1>
  salir si <cond>   →   NUNCA SE USA
  <sent2>
fin iterar
```

7. ESTRUCTURAS ANIDADAS:

Tanto las estructuras selectivas como los bucles se pueden anidar unos dentro de otros.

Anidación de condicionales:

La ventaja de anidar sentencias condicionales, es que cuando una se cumple no hay por que mirar a las que estan debajo. Tenemos que tratar anidar la condicion en la parte sino (else) en vez que en la parte entonces.

```
Si <condicion1>
  Entonces <sentencia1>
  Sino si <condicion2>
      Entonces <sentencia2>
      Sino si <condicion2>
          Entonces <sentencia3>
          Fin si
      Fin si
Fin si
```

El case siempre equivale a una anidación de condicionales, pero no al reves.

Bucles anidados:

Al igual que podemos colocar unas expresiones dentro de otras, unos bucles pueden estar dentro de otros, pero nunca pueden cruzarse. Al anidar bucles hay que tener en cuenta que el bucle mas interno funciona como una sentencia mas del bloque mas externo y por tanto en cada iteración del bucle mas externo se van a ejecutar todas las iteraciones del bucle mas interno por cada iteración del mas externo.

Si el bucle mas externo se repite n veces y el mas interno se repite m veces, si por cada iteración del mas externo se repite el mas interno, el número total de iteraciones será m*n.

Los bucles que se anidan pueden se de igual o distinto tipo.

```
Desde i=1 hasta 8
```

```
    Desde j=1 hasta 5
        Escribir "Profesor"i"introduzca su asignatura nº"j
        Leer asignatura
    Fin desde
Fin desde
```

8. CONTROL DE DATOES DE ENTRADA:

1. Cuando existe un valor centinela que determina el fin del bucle:

El bucle se va a repetir mientras no se lea un determinado valor. La primera lectura se va a realizar fuera del bucle, y si la primera vez que lo leo ya encuentro ese valor centinela no podemos entrar en el bucle y seguirá a continuación, sino entramos en el bucle.

Se va utilizar una estructura Mientras.

2. Lectura secuencial de un fichero:

Tendremos que leer del primero al último registro del fichero. Habrá que usar un Mientras, aunque hay dos posibilidades: Leer fuera del bucle y al final de bucle, o leer dentro al principio del bucle. Se usa una u otra dependiendo del lenguaje.

3. Cuando en un bucle sabemos el número exacto de veces que se repite el bucle:

Usamos el Desde...Hasta.

4. Control de datos para no permitir datos erroneos:

No dejar seguir al usuario hasta que no introduzca datos correctos. Hay que usar un Repetir...Hasta.

```
Repetir
  Leer datos
Hasta (datos correctos)
```

```
    Repetir
  leer op
hasta (op>=1) y (op<=3)
```

EJERCICIOS: TEMA 4

1. Al final de curso deseamos saber cual ha sido el alumno de primero con mejor nota media. Se sabe que este año entraron 150 alumnos y que en primero todos tienen 5 asignaturas. Dar el nombre y la nota media.

```
Algoritmo nota_media
Const
  Alumnos=150
  Asignaturas=5
Var
  Nombre, mejor_alum: cadena
  Nota, suma, media, acum: real
  I, j: entero
Inicio
  Acum ← 0
  Desde i=1 hasta alumnos hacer
    Suma ← 0
    Escribir "Introduzca el nombre del alumno"
    Leer nombre
    Desde j=1 hasta asignaturas hacer
      Escribir "Introduzca la nota de la asignatura"
      Leer nota
      Suma ← suma + nota
    Fin desde
    Media ← suma / asignaturas
    Si media > acum
      Entonces  acum ← media
```

```
                Mejor_alum ← nombre
    Fin si
  Fin desde
  Escribir "El mejor alumno es "mejor_alum
  Escribir "Su nota media es "acum
Fin
```

2. Calcular la suma de los divisores de cada número introducido por teclado. Terminaremos cuando el número sea negativo o 0.

```
Algoritmo divisores
Var
  Numero, i, suma :entero
Inicio
  Escribir "Introduce un número, y para acabar uno negativo"
  Leer número
  Mientras numero > 0
    Suma ← 0
    Desde i=1 hasta numero /2
      Si numero mod i =0
        Entonces suma ← suma + i
      Fin si
    Fin desde
    Suma ← suma + numero
    Escribir "La suma de los divisores del número es "suma
    Leer numero
  Fin mientras
 Fin
```

3. Se coloca un cápital C, a un interes I, durante M años y se desea saber en cuanto se habrá convertido ese capital en m años, sabiendo que es acumulativo.

```
Algoritmo interes
Var
  I, j, m: entero
  C, temporal: real
Inicio
```

```
  repetir
    Escribir "Introduce el capital, el interes y el tiempo"
    Leer c, i, m
  Hasta (c>0) y ((i<=0)y(i<=100)) y (m >=0)
  Desde j=1 hasta m
    C ← c * (1+I/100)
  Fin desde
  Escribir "Tienes "c" pts"
Fin
```

4. Dada una fecha en formato dia/mes/año determinar el número de días y el nombre del mes de dicha fecha, y sacar por pantalla la fecha convertida a formato de dia "de" mes "de" año.

```
Algoritmo fecha
Var
  Dia, mes, año, n_dias: entero
  N_mes: cadena
Inicio
  Repetir
    Escribir "Introduce la fecha en formato dia mes año"
    Leer dia, mes, año
    Según sea mes
      1,01: n_mes ← "enero"
            n_dias ← 31
      2,02: n_mes ← "febrero"
            si año mod 4=0
              entonces n_dias ← 29
              entonces n_dias ← 28
      3,03: n_mes ← "marzo"
            n_dias ← 31
      4,04: n_mes ← "abril"
            n_dias ← 30
      5,05: n_mes ← "mayo"
            n_dias ← 31
      6,06: n_mes ← "junio"
            n_dias ← 30
      7,07: n_mes ← "julio"
```

```
            n_dias ← 31
      8,08: n_mes ← "agosto"
            n_dias ← 31
      9,09: n_mes ← "septiembre"
            n_dias ← 30
        10: n_mes ← "octubre"
            n_dias ← 31
        11: n_mes ← "noviembre"
            n_dias ← 30
        12: n_mes ← "diciembre"
            n_dias ← 31
    fin según sea
  hasta (dia <=n_dias) y ((mes >=1)y(mes <=12)) y (año >=0)
  escribir "El mes de "n_mes"tiene "n_dias" dias"
  escribir "La fecha es: "n_dias" de "n_mes" de "año
fin
```

$$X = \frac{\sum_{i=1}^{n}((a-b)^{i}-3)+n}{\prod_{i=2}^{n-1}(2+a*(i-1))}$$

5. Dada la siguiente fórmula:

Averiguar el valor de x pidiendo al usuario los valores de n, a, b.

```
Algoritmo ecuacion
Var
  N, a, b, primer, según, i, j: entero
  X: real
Inicio
  Primer ← 0
  Según ← 1
  Repetir
    Escribir "Introduce el valor de n, a, b"
    Leer n, a, b
  Hasta n>=0
  Desde i=1 hasta n
    Primer ← (((a-b)^i –3)+n)+primer
  Fin desde
  Desde j=2 hasta n-1
    Según ← ((2*a*(i-1))*según)
  Fin desde
```

```
   X ← primer / según
   Escribir x
Fin
```

6. Dada la siguiente serie matemática:

a1=0
a2=0
an=an-1 + (2*an-2) para n>=3

Determinar cual es el valor y el rango del primer término cuyo valor sea mayor o igual a 2000.

```
Algoritmo serie
Var
   A1, a2, an, cont: entero
Inicio
   A1 ← 1
   A2 ← 0
   An ← a1 + (2*a2)
   N ← 3
   Mientras an < 2000
      A2 ← a1
      A1 ← an
      An ← a1 + (2*a2)
      Cont ← cont + 1
   Fin mientras
   Escribir “El rango es “cont” y el resultado es”an
fin
```

SUBPROGRAMAS: PROCEDIMIENTOS Y FUNCIONES.
TEMA 5

1. **Introducción a los subprogramas o subalgoritmos.**
2. **Funciones.**
3. **Procedimientos.**
4. **Ámbitos: variables locales y globales.**
5. **Comunicación entre subprogramas: Paso de parámetros.**
   - **Tipos y métodos de paso de parámetros.**
   - **Paso de parámetros por copia.**
   - **Paso de parámetros por referencia.**
   - **Paso de parámetros por nombre.**
6. **Funciones y procedimientos como parámetros.**
7. **Efectos laterales.**
8. **Recursividad.**

1. INTRODUCCIÓN A LOS SUBPROGRAMAS O SUBALGORITMOS:

La programación modular es una de las técnicas fundamentales de la programación. Se apoya en el diseño descendente y en la filosofía de "divide y vencerás", es decir se trata de dividir el problema dado, en problemas más simples en que cada uno de los cuales lo implementaremos en un módulo independiente. A cada uno de estos módulos es a lo que llamamos subalgoritmos o subprogramas.

Siempre existirá un módulo o programa principal que es con el que comienza la ejecución de todo el programa, y a partir de él iremos llamando al resto.

Cada vez que se llama a un subprograma se le pasa la información que necesita en la llamada, a continuación comienza a ejecutarse el subprograma

llamado, y cuando termine su ejecución, devuelve el control a la siguiente instrucción a la de llamada en el programa que lo llamó.

En cuanto a la estructura de un subprograma es igual a la estructura de un programa, va a tener una información de entrada que es la que le pasamos al hacer la llamada y que se coloca junto al nombre del subprograma. Después va a tener un conjunto de acciones, declarar otras variables propias del subprograma, y al terminar la ejecución puede que devuelva o no resultados al programa que lo llamó.

Hay dos tipos fundamentales de subprogramas: Funciones y procedimientos.

2. FUNCIONES:

Desde el punto de vista matemático, una función es una operación que toma uno o varios operandos, y devuelve un resultado. Y desde el punto de vista algorítmico, es un subprograma que toma uno o varios parámetros como entrada y devuelve a la salida un único resultado.

Pascal: En las funciones se puede devolver más de un único resultado mediante parámetros.

C: Se devuelve todo por parámetros.

Este único resultado irá asociado al nombre de la función. Hay dos tipos de funciones:

- **Internas: Son las que vienen definidas por defecto en el lenguaje.**
- **Externas: Las define el usuario y les da un nombre o identificador.**

Para llamar a una función se da su nombre, y entre paréntesis van los argumentos o parámetros que se quieren pasar.

Declaración de una función:

La estructura de una función es semejante a la de cualquier subprograma. Tendrá una cabecera (con el nombre y los parámetros) y un cuerpo(con la declaración de los parámetros de la función y las instrucciones).

Sintaxis:

```
Funcion <nombre_funcion> (n_parametro: tipo, n_parametro: tipo): tipo funcion
Var <variables locales funcion>
Inicio
  <acciones>
   retorno <valor>
fin <nombre_funcion>
```

La lista de parámetros es la información que se le tiene que pasar a la función. Los parámetros luego dentro de la función los podemos utilizar igual que si fueran variables locales definidas en la función y para cada parámetro hay que poner su nombre y tipo.

El nombre de la función lo da al usuario y tiene que ser significativo.

En las variables locales se declaran las variables que se pueden usar dentro de la función.

Entre las acciones tendrá que existir entre ellas una del tipo *retorno <valor>*. Esta sentencia pondrá fin a la ejecución de la función y devolverá el valor de la función, es decir, como valor asociado al nombre de mismo tipo que el tipo de datos que devuelve a la función, este valor por tanto tiene que ser del mismo tipo

que el tipo de datos que devuelve la función, que es el que habremos indicado al declarar la función en la parte final de la cabecera.

No se permiten funciones que no devuelvan nada.

Los parámetros que aparecen en la declaración de la función se denominan parámetros formales, y los parámetros que yo utilizo cuando llamo a la función se denominan parámetros actuales o reales.

Invocación de una función:

Para llamar a una función se pone el nombre de la función, y entre paréntesis los parámetros reales, que podrán ser variables, expresiones, constantes,... pero siempre del mismo tipo que los parámetros normales asociados

<nombre_funcion> (parámetros reales)

La función puede ser llamada desde el programa principal o desde cualquier otro subprograma.

Para llamar a la función desde cualquier parte, implica el conocimiento previo de que ésta función existe.

A través de los parámetros reales de la llamada se proporciona a la función la información que necesita, para ello, al hacer la llamada lo que se produce es una asociación automática entre parámetros reales y parámetros formales. Esta asociación se realiza según el orden de la aparición y de izquierda y derecha.

Si el parámetro formal y real no son del mismo tipo, en Pascal se produce un error, y en C se transforman los tipos si es posible.

La llamada a una función, siempre va a formar parte de una expresión, de cualquier expresión en la que en el punto en la que se llama a la función, pudiera ir colocado cualquier valor del tipo de datos que devuelve la función, esto se debe a que el valor que devuelve una función esta asociado a su nombre.

Pasos para hacer la llamada a una función:

1. **Al hacer la llamada y ceder el control a la función, se asocia (asigna el valor) de cada parámetro real a cada parámetro formal asociado, siempre por orden de aparición y de izquierda a derecha, por lo que siempre que no coincidan los tipos y el número de parámetros formales y reales, se produce un error.**
2. **Si todo ha ido bien, se ejecutan las acciones de la función hasta que lleguemos a una de tipo *retorno <valor>* que pondrá fin a la ejecución. Pueden existir varias sentencias de retorno en la misma función, pero en cada llamada solo se podrá ejecutar uno.**
3. **Se le asocia al nombre de la función el valor retornado y se devuelve el control al subprograma que hizo la llamada pero sustituyendo el nombre de la función por el valor devuelto.**

Otra forma de especificar el retorno de una función:

Se le asigna el valor devuelto al nombre de la función.

N_funcion ← valor

Ejemplo de función:

Una función que calcule la mitad del valor que le paso parámetro. Suponemos que es un valor entero.

```
Funcion mitad (n: entero): real
Var m: real
Inicio
  M ← n/2
  Retorno m
```

Fin mitad

```
Algoritmo calc_mitad
Var num: entero
Inicio
  Escribir "Introduce un número para hallar su mitad"
  Leer num
  Escribir "La mitad de "num" es "mitad(num)
Fin
```

**** La función solo puede ser llamada desde una expresión.***

3. PROCEDIMIENTOS:

El inconveniente de una función es que solo puede devolver un único valor, por lo que sí nos interesa devolver 0 o N valores, aunque puedo usarlo para devolver un solo valor, debo usar un procedimiento.

Un procedimiento es un subprograma o un subalgoritmo que ejecuta una determinada tarea, pero que tras ejecutar esa tarea no tienen ningún valor asociado a su nombre como en las funciones, sino que si devuelve información, lo hace a través de parámetros.

Al llamar a un procedimiento, se le cede el control, comienza a ejecutarse y cuando termina devuelve el control a la siguiente instrucción a la de llamada.

Diferencias entre funciones y procedimientos:

1. **Una función devuelve un único valor y un procedimiento puede devolver 0,1 o N.**
2. **Ninguno de los resultados devueltos por el procedimiento se asocian a su nombre como ocurría con la función.**
3. **Mientras que la llamada a una función forma siempre parte de una expresión, la llamada a un procedimiento es una instrucción que por sí sola no necesita instrucciones.**
   Esta llamada consiste en el nombre del procedimiento y va entre paréntesis van los parámetros que se le pasan. En algunos lenguajes (como el C), se pone delante la palabra Llamar a (Call) procedimiento (parámetro).

Sintaxis:

```
Procedimiento <nombre_proc> (<tipo_paso_par> <nombre_par>: tipo_par,...)
Var <variables locales>: tipo
Inicio
  <sentencias>
fin <nombre_proc>
```

La cabecera va a estar formada por el nombre del procedimiento que será un identificador y que debe de ser significativo, y luego entre paréntesis los parámetros o la información que se le pasa al procedimiento. Para cada parámetro hay que indicar el tipo de paso de parámetro. Hay dos tipos fundamentales de paso de parámetros, por valor y por referencia, si no ponemos tipo de paso de parámetros, se toma el tipo de paso de parámetros por valor.

En el cuerpo del procedimiento donde van las sentencias ya no habrá ninguna de tipo <retorno valor>, ahora bien, si el procedimiento devuelve resultados a través de sus parámetros, cosa que solo podrá hacer a través de los parámetros que se pasan por referencia, tendrán que existir sentencias de

asignación de valores a estos parámetros pasados por referencia, a través de los cuales se van a devolver los resultados.

Como se llama a un procedimiento:

[llamar a (Call)] nombre_proc (par_reales)

Pasos para hacer la llamada a un procedimiento:

1. **Se cede el control al procedimiento al que se llama y lo primero que se hace al cederle el control es sustituir cada parámetro formal de la definición por el parámetro actual o real de la llamada asociado a él. Esta asociación entre parámetros formales y reales se hace de izquierda a derecha por orden de colocación y para que se pueda producir esta asociación tienen que existir el mismo número de parámetros formales que reales, y además el tipo tiene que coincidir con el del parámetro formal asociado, sino se cumple alguna de estas condiciones hay un error.**
2. **Si la asociación ha sido correcta comienzan a ejecutarse las instrucciones del procedimiento hasta llegar a la última instrucción. Al llegar a la instrucción se vuelven a asociar los parámetros formales que devuelven los resultados a los parámetros formales asociados en la llamada, es decir, de esta manera algunos de los parámetros reales de la llamada ya contendrán los resultados del procedimiento.**
3. **Finalmente se cede el control a la siguiente instrucción a la que se hace la llamada, pero teniendo en cuenta que en esta instrucción y en las siguientes puedo usar ya los parámetros reales en los que se devolvieron los resultados del procedimiento para trabajar con ellos.**

```
Procedimiento mitad (num:entero,ent-sal M:real)
Inicio
  M ← num/2
Fin mitad

Algoritmo calc_mitad
Var
  N: entero
  Mit: real
Inicio
  Escribir “Introduce un número”
  Leer n
  Mitad (n,mit)
  Escribir “La mitad es”mit
fin
```

4. ÁMBITOS: VARIABLES LOCALES Y GLOBALES:

¿Qué es el ámbito de un identificador?:

El ámbito de un identificador (variables, constantes, funciones,...) es la parte del programa en la que se conoce y por tanto se puede usar un identificador.

Según el ámbito hay 2 tipos de variables, locales y globales:

1. **<u>Local</u>: Aquella que está declarada y definida dentro de un subprograma luego su ámbito coincidirá con el ámbito del subprograma en la que este definida.**
   Esto quiere decir que la variable no tiene ningún significado, no se conoce y no se puede acceder a ella desde fuera del subprograma y que tiene una posición de memoria distinta a la de cualquier otra, incluso si es de una variable que tiene el mismo nombre pero que está definida fuera del subprograma.
   Las variables locales a un subprograma se definen en la parte de la definición de variables del mismo. Los parámetros formales que se le ponen a un subprograma se comportan dentro de él como si fueran también variables locales a él.
2. **<u>Globales</u>: Son las que están definidas a nivel del programa, es decir, su ámbito es el programa o algoritmo principal y todos los subprogramas que van junto con él.**
   A esta variable podemos acceder desde cualquiera de los subprogramas y el programa principal, salvo que alguno de esos subprogramas tenga definida una variable local con el mismo nombre que la variable global, en este caso si utilizo el nombre de esa variable me referiré a la local, nunca a la global(ya que tienen 2 zonas de memoria distintas).

<u>Lugar en el que se definen las variables globales:</u>

En algunos lenguajes se define en el programa principal, y esa variable será global, en otros lenguajes se definen fuera del programa principal y fuera de cualquier otro subprograma(antes de empezar el programa principal). → Método que vamos a usar.

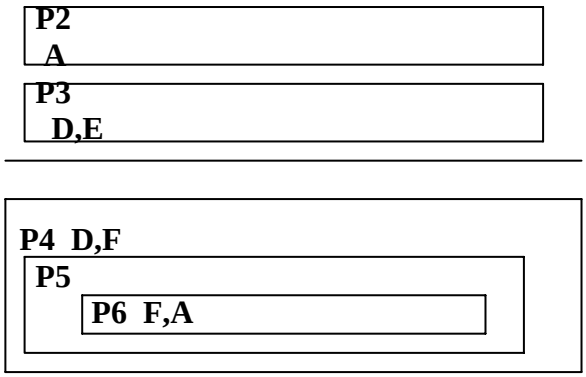
El problema de usar variables globales es que como todos los subprogramas las pueden modificar, puede ser posible que haya usos indebidos cuando un subprograma utiliza una variable global sin saber que otro la ha modificado, por esa razón nunca usaremos para pasar información entre los subprogramas variables globales, sino que usaremos variables de entrada-salida, salvo que no nos quede más remedio.

<u>Procedimientos anidados:</u>

La anidación de procedimientos no se permite en todos los lenguajes y consiste en que dentro de un procedimiento podamos definir o meter el código de otros.

Si la anidación de procedimientos está permitida, se plantean más problemas en cuanto al ámbito, desde este punto de vista, se dice que una variable local se conoce en el procedimiento en el que está definida y en todos los procedimientos anidados a él que son los que componen el ámbito de dicho procedimiento, salvo que en alguno de esos procedimientos anidados este definida una variable local con el mismo nombre en cuyo caso dentro de ese procedimiento siempre nos referiremos a la variable local definida en él porque siempre se considera el ámbito más restringido.

```
PP
  Var A,B
  P1
    A,C,D
```



VARIABLES	ÁMBITO SUBPROGRAMA
A del PP	P4,P5
B del PP	P1,P2,P3,P4,P5,P6
A del P1	P1,P3
C del P1	P1,P2,P3
D del P1	P1,P2,P3
A del P2	P2
D del P3	P3
E del P3	P3
D del P4	P4,P5,P6
F del P4	P4,P5
F del P6	P6
A del P6	P6

5. COMUNICACIÓN ENTRE SUBPROGRAMAS: PASO DE PARÁMETROS.

La mejor forma para llevar a cabo la comunicación ente subprogramas, es el paso de parámetros. Trataremos de evitar siempre que sea posible el uso de variables globales.

Cuando llamamos a una función o procedimiento, le pasamos a través de los parámetros la información que necesita, y en el caso de un procedimiento también devolvemos a través de sus parámetros los resultados. Para ello definiremos el tipo del parámetro a principio del subprograma, que es lo que conocemos como parámetros formales, y al hacer la llamada pasamos la información a través de los parámetros reales.

¿Cómo se efectúa la correspondencia entre parámetros formales y reales?:

Existen 2 métodos:

1. **Correspondencia posicional: En este caso se emparejan los parámetros formales y reales por la posición que ocupan (orden de declaración) y de izquierda a derecha. Para que se pueda realizar esta asociación, tiene**

que haber el mismo número de parámetros formales y reales, y con el mismo tipo.

F (x:entero,y:real)
Var a:real
F (3,A)

2. **<u>Correspondencia por nombre implícito</u>: Ahora en la llamada al subprograma se pone explícitamente a que parámetro formal corresponde cada real.**
   Ahora en la llamada ponemos el nombre del parámetro formal, separado por dos puntos (:) y el nombre del parámetro real que le pasamos, con lo cual ya no importa la posición en la que coloquemos la información.

F (x:3,y:A)
F (y:A,x:3)

Un lenguaje que permite esto es ADA.
Usaremos siempre el primer método (posicional).

<u>Paso de parámetros:</u>

Del modo de paso de parámetros va a depender el resultado que se obtenga.

<u>1. Tipos de parámetros:</u> Según se usen para meter datos o para obtener resultados.
Existen 3 tipos:

1. **<u>De entrada</u>: Son parámetros que solo aportan información de entrada al subprogama en el que pertenecen como parámetros. Aporta el valor que tienen como entrada al subprograma. En el caso de una función, todos sus parámetros son de este tipo.**
   Como solo sirven como entrada, solo pueden ser leídos, pero no modificados, y aunque se modificasen dentro de un subprograma, fuera no va a tener efecto esa modificación.
2. **<u>De salida</u>: Se usan solo y exclusivamente para devolver resultados a través de ellos. Su valor al hacer la llamada al subprograma no nos importa, sino que ese valor solo va a tener importancia cuando termina la ejecución del programa. Un parámetro de este tipo teóricamente nunca se puede leer, solo se va actualizar.**

3. **<u>De entrada y salida</u>: El valor del parámetro tiene importancia tanto a la entrada como a la salida del subprograma, nos aporta información cuando llamamos al subprograma y por otra parte devolvemos a través de él resultados cuando terminamos el subprograma, en este caso, tiene sentido tanto leer como actualizar el parámetro.**

- **Solo ADA es un lenguaje que va a soportar los 3 tipos de paso de parámetro. Se ponen como In, Out, In-Out.**
- **El resto de los lenguajes solo permiten dos tipos de parámetros, de entrada (solo para leer datos) y de entrada-salida (para devolver resultados, aunque también se puede usar para leer datos).**

<u>2. Métodos de paso de parámetros:</u>

- **Paso de parámetros por copia:**

- **Por valor.**
- **Por resultado.**
- **Por valor-resultado.**

- **Paso de parámetros por referencia.**
- **Paso de parámetros por nombre o nominal.**

<u>Paso de parámetros por copia:</u>

La característica fundamental de este método de paso de parámetros es que el parámetro formal siempre se considera que tiene asociada una dirección de memoria en la que está almacenado y que por tanto se comporta igual que una variable local del subprograma en que aparece.

En este método lo que importa es el valor del parámetro actual.

1. **<u>Por valor:</u> Nos interesa el valor del parámetro actual a la entrada, para ello este valor se copia en la dirección de memoria del parámetro formal asociado. En este caso el parámetro real puede se una constante, expresión o variable, y nunca se va a usar para devolver resultado a través de él, por esa razón precisamente puede ser una constante o una expresión, porque al no devolver resultados a través de él no necesita tomar ninguna zona de memoria en la que este almacenado, es más, incluso si el parámetro actual fuera una variable y la modificásemos dentro del subprograma (algo que no deberíamos hacer), fuera del subprograma no tendría ninguna repercusión esta modificación, es decir, esa variable serviría valiendo lo mismo en el programa desde el que se hace la llamada después y antes de hacerla.**

   El funcionamiento sería el siguiente:

   - **Al hacer la llamada al subprograma se evalúa el valor del parámetro real, y ese es el que se asocia, es decir, el que se guarda o asigna al parámetro formal asociado.**
   - **Comenzamos a ejecutar el subprograma y si por casualidad se produce algún cambio en el parámetro formal, el parámetro actual no se verá afectado, es decir, su valor seguirá siendo el mismo en el subprograma desde donde se llama que antes de hacer la llamada al subprograma.**

```
Algoritmo Ej
  Var A:entero
Inicio
  A ← 3
  P (A)
  Escribir A
Fin
Procedimiento P(x:entero)
Inicio
  X ← x*2
  Escribir x
Fin p
```

3

A del PP

⁄3 6

X de P

El valor de A sería 3 y el de X sería 6.

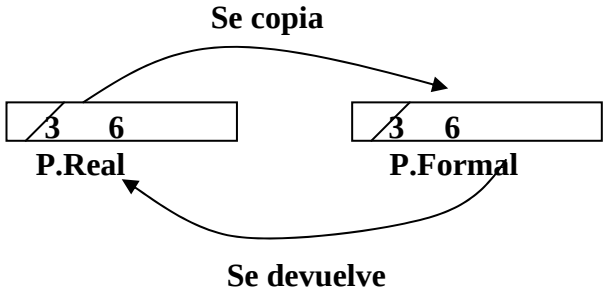
2. Por valor-resultado: En el valor-resultado nos interesa el valor del parámetro actual tanto a la entrada como a la salida de la ejecución del subprograma.

Esto quiere decir que se cambia el valor del parámetro formal cambiará también el valor de su parámetro real asociado, cosa que no ocurría antes, y esto supone por tanto que ahora el parámetro real tiene que tener asociada obligatoriamente una dirección de memoria, por lo que siempre tendrá que ser una variable (no una constante ni una expresión).

El proceso sería el siguiente:

- Al hacer la llamada al subprograma se evalúa el parámetro real y su valor se copia en el parámetro formal asociado y al terminar la ejecución el valor del parámetro formal se vuelve a copiar en el parámetro real asociado.

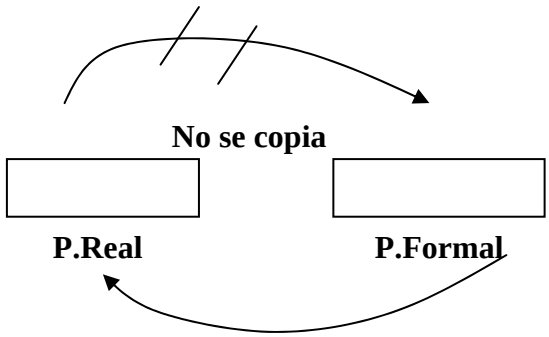
```
Algoritmo Ej
  Var A:entero
Inicio
  A ← 3
  P (A)
  Escribir A
Fin
Procedimiento P(x:entero)
Inicio
  X ← x*2
  Escribir x
Fin p
```



El valor de la A y la X sería 6.

3. Por resultado: Nos interesa el valor del parámetro real solamente a la salida o fin de la ejecución del subprograma en que aparece. Esto significa que al hacer la llamada no se copia el valor del parámetro real en el parámetro formal asociado, sin embargo a la salida se copia el valor del parámetro formal en la dirección del parámetro real asociado, esto significa por tanto, que el parámetro real tiene que tener asociada una expresión que tiene que ser una variable (no puede ser una constante o una expresión).

```
Algoritmo ej
  Var A:entero
Inicio
  A ← 3
  P (A)
  Escribir A
Fin
Procedimiento P(x:entero)
Inicio
  X ← 1
  X ← x*2
```



Escribir x
Fin p
El valor de A y de X es 2.

En la práctica la mayor parte de los lenguajes dentro del tipo de paso de parámetro por copia solo van a soportar el tipo de paso de parámetro por valor, que se usará cuando un parámetro solo lo quiero utilizar como entrada de información al subprograma, pero no para devolver resultados a través de él.

Los otros dos tipos de paso de parámetros por copia (por valor y por valor-resultado), no se implementan normalmente porque los efectos son prácticamente iguales que el paso de parámetros por referencia, es decir, cuando quiero usar un parámetro no solo para pasar información a través de él sino también para devolver resultados o si lo voy a usar sólo para devolver resultados, utilizaré el paso de parámetros por referencia que vamos a ver a continuación.

Para simbolizar que el tipo de paso de parámetros es por valor, en la definición del subprograma no pongo ningún tipo de paso de parámetros para ese parámetro, es decir, no poner ningún tipo significa que por defecto el tipo de paso de parámetros es por valor.

En el caso de las funciones como solamente pueden recibir información de entrada pero nunca pueden devolver información por sus parámetros ya que lo devuelven a través de la sentencia retorno y asociado a su nombre.

El tipo de paso de sus parámetros va a ser siempre por valor.

<u>Paso de parámetros por referencia:</u>

Ahora la característica principal de este tipo de paso de parámetros es que el parámetro formal va a tener también asignada una dirección de memoria en la que se almacena, pero en esa dirección NO SE GUARDA SU VALOR, sino que se almacena la dirección de su parámetro real asociado, es decir, el parámetro formal apunta al parámetro real que tiene asociado y cualquier modificación que se efectúe sobre el parámetro formal tendrá una repercusión directa en el parámetro real asociado ya que lo que modificará será el valor almacenado en la dirección que indica el parámetro formal que es la de su parámetro formal asociado.
El proceso será por tanto el siguiente:

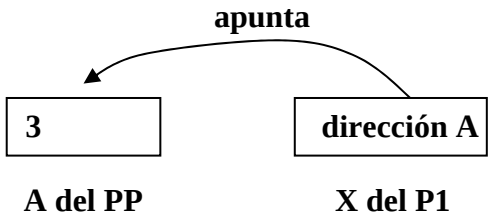
- **Al hacer la llamada al procedimiento en el parámetro formal que se pasa por referencia, se va a guardar la dirección del parámetro real asociado para que apunte a él.**
- **Durante la ejecución cualquier referencia al parámetro formal se hará accediendo a la dirección apuntada por dicho parámetro, es decir, accediendo directamente al parámetro real asociado, por lo que cualquier cambio en el parámetro formal afectará directamente al parámetro real asociado. De esta manera habremos pasado el resultado.**

Para indicar que el tipo de paso de parámetro es por referencia, vamos a utilizar la palabra clave ent-sal precediendo al parámetro que se pasa por referencia.

A estos parámetros también se les llama parámetros variables (porque su valor varía), por eso en Pascal se usa la palabra clave Var para indicarlo.

En otros lenguajes como C, se usan como parámetros punteros para indicar que son direcciones.

```
Algoritmo EJ
Var A
Inicio
  A ← 3
  P1(A)
  Escribir (A)
Fin
Procedimiento P1(ent-sal x:entero)
Inicio
  X ← x*2
Fin P1
```



Por valor el parámetro actual no cambia de valor.
Por referencia el parámetro actual puede cambiar.

<u>Paso de parámetros por nombre:</u>

En este caso, el parámetro formal se sustituye literalmente por el parámetro actual asociado. Esta sustitución literal del parámetro formal por el parámetro actual no se produce hasta que no se usa el parámetro formal.

La ventaja es que si no usamos en ningún momento el parámetro formal dentro del subprograma llamado (cosa poco probable), no se tendrá que hacer ningún tipo de substitución.

Pero el gran inconveniente es que si se usa el parámetro formal, hay que ir a buscar en ese momento la expresión del parámetro real asociado.

El paso de parámetro por nombre es lo que se parece más a la substitución de parámetros en una función matemática.

F (x)= x+2
F (a+1) = a+1*2
F (x)= x+2
F (a+1)= a +1 2 <> (a+1)*2

```
Algoritmo EJ
Var A: entero
Inicio
  A ← 3
```

```
  P1(A+1)
  Escribir (A)
Fin
Procedimiento P1(ent-sal x: entero)
Inicio
  X ← x*2
Fin P1
```

Al final solo vamos a usar 2 tipos de paso de parámetros, que son los que usan casi todos los lenguajes: Por valor y por referencia.

<u>Por valor</u>: Solo lo usamos cuando un parámetro solo sirve para información de entrada, no devolvemos nada con él. Por eso este paso puede ser una constante, variable o expresión. Para simbolizarlo en la declaración de variables no ponemos nada.

<u>Por referencia</u>: Lo usamos cuando un parámetro lo usamos como entrada de información y como salida o solo como salida. Por esta razón ahora sí que se va a variar el parámetro formal y por lo tanto no podemos usar constantes ni expresiones, solo variables. Lo simbolizamos con la palabra ent-sal.

6. FUNCIONES Y PROCEDIMIENTOS COMO PARÁMETROS:

En la mayor parte de los lenguajes se permite también que una función o procedimiento pueda ser pasado como parámetro de otro subprograma. En este caso, es decir, cuando el parámetro formal es de tipo función o procedimiento, pasaremos como parámetro real funciones o procedimientos que tengan la misma definición que el que hemos puesto como parámetro formal, y en nuestro caso para indicar que se pasa como parámetro real una función o procedimiento, basta con poner su nombre.

Desde el subprograma al que se pasa como parámetro esa función o procedimiento podemos llamar en cualquier momento a esa función pasada como parámetro que en cada momento podrá ser una distinta dependiendo del parámetro formal asociado.

```
Funcion <nombre_f>          (par:tipo;funcion
<n_f>(x:entero,y:carácter):entero
Prodedimiento <nombre_f>          procedimiento (x:tipo,...);...
```

```
Algoritmo EJ
Var v1,v2: entero
Inicio
  V1 ← 1
  V2 ← 2
  P(v1,f1,v2)
```

```
  P(v1,f2,v2)
Fin
Procedimiento P(x:entero;funcion F(A:entero;B:carácter):entero;ent-sal y:entero)
Inicio
  X ← 2
  Y ← F(x,'a')
Fin P
Funcion F1(x:entero;y:carácter):entero
Inicio
  Escribir y
  Retorno (x+1)
Fin
Funcion F2(A:entero;B:carácter):entero
Inicio
  Escribir "Hola" B
  Retorno A
Fin F2
```

El paso de funciones y procedimientos como parámetros, nos va a permitir que desde un subprograma podamos llamar a otros, pero teniendo en cuenta que el subprograma llamado no va a ser uno determinado, sino que va a depender en cada momento del subprograma pasado como parámetro real, de esta manera el subprograma puede llamar a un conjunto de n subprogramas que cumplan unas determinadas características, pero solo uno en cada momento.

No hay que confundir el paso de una función como parámetro con la llamada a una función cuando aparece como parámetro real en la llamada a un subprograma. Para diferenciarlo, cuando paso una función como parámetro solo pondré su nombre, en cambio cuando llamo a una función para pasar el valor que devuelve como parámetro pondré el nombre de la función y con sus argumentos para indicar que ahí se puede hacer la llamada. Esto se puede hacer porque la llamada a una función puede aparecer en cualquier expresión en la que apareciese cualquier valor del tipo de datos que devuelve.

```
Procedimiento P(A:entero)
Inicio
  Escribir A
Fin P
Funcion F(x:entero):entero
Inicio
  Retorno (x*2)
Fin F
```

```
Algoritmo EJ
Var I
Inicio
  I ← 2
  P(F(I)) → Esto no es pasar una función como parámetro
Fin
```

7. EFECTOS LATERALES:

Un efecto lateral es cualquier modificación que un subprograma (sea función o procedimiento), realiza en elementos situados fuera de él pero sin hacer esas modificaciones a través del paso de parámetros.

Los efectos laterales siempre hay que evitarlos porque no somos conscientes o no controlamos que se han producido.

Para realizar comunicación entre subprogramas solo se hará a través del paso de parámetro para evitar los efectos laterales.

Los efectos laterales normalmente se producen por el uso de variables globales o variables locales que abarcan varios procedimientos (esto solo es posible si hay anidación de subprogramas). Por lo tanto evitaremos su uso excepto que sea imprescindible.

```
Var A:entero
Algoritmo EJ
Var B:entero
Inicio
  B ← 1
  A ← 2
  P(B)
  Escribir A
Fin
Procedimiento P(x:entero)
Inicio
  A ← x+2
Fin P
```

8. RECURSIVIDAD:

Se dice que un subprograma es recursivo cuando se llama a sí mismo. La recursividad se va a poder usar en subprogramas que pueden definirse en términos recursivos, es decir, en termino de sí mismo, como procesos de alguna manera repetitivos.

La recursividad trataremos de evitarla siempre que sea posible, es decir, siempre que lo mismo se pueda solucionar con un bucle, ya que cada vez que un subprograma se llama a sí mismo hay que almacenar en la pila del sistema la dirección de retorno de ese subprograma, con lo cual si hacemos muchas llamadas recursivamente iremos llenando la pila del sistema, y se desbordara acabando con la memoria.

Todo programa recursivo tiene que tener alguna condición que ponga fin a la recursividad, es decir, que el programa deje de llamarse a sí mismo cuando se cumpla la condición, sino se formaría un bucle infinito.

```
Funcion potencia (base:entero;exp:entero):real
Var P:real
    I:entero
Inicio
  P ←1
```

```
  Desde i=1 hasta exp
    P ← P*base
  Fin desde
  Retorno P
Fin

Funcion potencia (base:entero;exp:entero):real
Inicio
  Si exp=0 entonces retorno 1
  Sino
    Retorno (base*potencia(base,exp-1)
Fin potencia
```

TEMA 5: EJERCICIOS

1. Implementar un subprograma que realice la serie de Fibonacci, que es:

Fibonacci (1)= Fibonacci (2)=1

N > 2 Fibonacci (n)= Fibonacci (n-1) + Fibonacci (n-2)

```
Algoritmo serie_fibonacci
Var
  I, n: entero
Inicio
  Escribir "Introduce un número"
  Leer n
  Desde i=1 hasta n
    Escribir "La serie de fibonacci de "i" es "fibonacci (i)
  Fin desde
Fin
Funcion fibonacci (num: entero): entero
Inicio
  Si (num=1) o (num=2)
    Entonces retorno 1
  Sino
    Retorno (fibonacci (num-1) + fibonacci (num-2)
  Fin si
Fin fibonacci
```

2. Implementar un subprograma al que pasándole como parámetros 2 valores enteros M y N, me calcule el combinatorio

```
Algoritmo combinatorio
Var
  M,n: entero
Inicio
  Repetir
    Escribir "Introduce el valor de M y N"
    Leer m,n
  Hasta m >n
  Escribir "El combinatorio es "factorial (m) div (factorial(n)*factorial(m-n))
Fin
Funcion factorial (num: entero): real
Inicio
  Si num=0
    Entonces retorno 1
  Sino
    Retorno (num * factorial (num-1))
  Fin si
Fin factorial
```

3. Implementar un subprograma que me halle cual es la primera potencia en base 2 mayor que un número que pasamos como parámetro, devolviendo el valor de dicha potencia y el exponente al que está elevado.

```
Algoritmo elevar
```

```
Var
  Numero, resp1, resp2: entero
Inicio
  Escribir "Introduce un número"
  Leer numero
  Comprueba (numero, resp1, resp2)
  Escribir "2^"resp1"="resp2" >"numero
Fin
Procedimiento comprueba (num: entero; ent-sal n: entero; ent-sal pot: entero)
Inicio
  N ←1
  Mientras pot < n
    Pot ← pot *2
    N ← n+1
  Fin mientras
Fin comprueba
```

4. Implementar un subprograma que calcule recursivamente en cuanto se convierte un capital C al final de N años y a un interés I.

```
Funcion calculo (c: entero;i: entero; n: entero): real
Inicio
  Si m=0
    Entonces retorno C
  Sino
    Retorno (c*(1+i/100)+calculo (c,i,n-1)
  Fin si
Fin calculo
```

5. Calcular el primer término de la siguiente serie que sea mayor o igual a un valor V que se le pasa como parámetro y me devuelva el lugar que ocupa en la serie y el valor.
Ai=0
An=n+(An-1)!

```
Funcion factorial (num: entero): entero
Var
  I, acum: entero
Inicio
  Acum ←1
  Desde i=1 hasta num
    Acum ← acum * i
  Fin desde
  Retorno acum
Fin factorial
Procedimiento serie (v: entero; ent-sal an: entero; ent-sal n: entero)
Var
  A1: entero
Inicio
  A1 ← 0
  An ← 0
  N ← 1
  Mientras an <= V
    N ← n+1
    An ← n + factorial (a1)
    A1 ← an
  Fin mientras
Fin serie
```

6. Calcular el valor de la serie donde N es un valor que se pasa como parámetro al subprograma que hace el cálculo.

$$\sum_{i=0}^{n-1}(1+i*\sum_{i=0}^{n}\frac{1}{n})$$

```
Funcion suma (n: entero): real
Var
  I, j: entero
  X, y, acum1, acum2, suma1, suma2: real
Inicio
  Acum1 ← 0
  Acum 2 ← 0
  Desde i=0 hasta n
    X ← 1/n
    Acum1 ← acum1 + x
  Fin desde
  Desde j=0 hasta n-1
    Y ← 1 +i*acum1 + y
  Fin desde
  Retorno acum2
Fin suma
```

7. ¿Qué se escribe en pantalla tras la siguiente ejecución?

```
Algoritmo EJ
Var
  A,B,C: entero
```

```
Inicio
  A ← 1
  B ← 2
  C ← A+3
  P1 (A, B-C, C)
  C ← C – F(A)
  P2 (A,C)
  P1 (C, B, A)
  Escribir A, B, C
Fin
Procedimiento P1 (ent-sal x: entero; y: entero; ent-sal z: entero)
Inicio
  X ← y +z
  Y ← x +1
  Z ← y * 2
Fin P1
Procedimiento P2 (ent-sal x: entero; y: entero)
Inicio
  X ← x +1 –y
  Y ← 3
Fin P2
Funcion F (x: entero): entero
Inicio
  X ← x + 3
  Retorno (x – 1)
Fin F
```

- **La solución es A=8; B=2; C=3**

ESTRUCTURAS DE DATOS: ARRAYS. TEMA 6

1. **Introducción a las estructuras de datos.**

2. **Arrays unidimensionales o vectores.**
3. **Operaciones con arrays unidimensionales o vectores.**
4. **Arrays bidimensionales o matrices.**
5. **Arrays multidimensionales.**
6. **Almacenamiento de arrays en memoria.**
7. **Arrays como parámetros de subprogramas.**
8. **Arrays de "punteros".**

1. INTRODUCCIÓN A LAS ESTRUCTURAS DE DATOS:

Clasificación de los tipos de datos según su estructura:

- Simples:
  - **Estándar (entero, real, carácter, booleano)**
  - **No estándar (enumeración, subrango)**

- Estructurados:
  - **Estáticos (arrays, cadena, registros, ficheros, conjuntos)** — **Pilas**
  - **Dinámicos (punteros, listas enlazadas, árboles, grafos)** — **Colas**

Los tipos simples son cuando cada dato representa un único elemento:
- **Estándar: Están definidos por defecto por el lenguaje.**
- **No estándar: Tipos simples definidos por el usuario.**

Los tipos de datos estructurados, son cuándo un dato es una estructura que se construyen a partir de otros complementos.
- **Estáticos: Ocupan un tamaño de memoria fijo, que tengo que definir antes de declararlo.**
- **Dinámicos: La estructura no ocupa un tamaño fijo de memoria, sino que ocupa la memoria que ocupa en cada momento. Se van a manejar a través del tipo de dato puntero.**
- **Puntero: Es una variable cuyo contenido es una dirección de memoria y esa dirección de memoria corresponde a la dirección de memoria de otra variable, que es la variable apuntada.**
  Según el tipo de datos de la variable apuntada variará el tipo de puntero. A través de una variable de tipo puntero podemos establecer la conexión o enlace entre los elementos que van a formar la estructura, y según se realizan estos enlaces vamos a tener diferentes tipos de estructuras (listas enlazadas, árboles, grafos).

Las pilas y las colas son 2 estructuras de datos con un funcionamiento especial, que pueden implementarse con memoria estática o dinámica.

2. ARRAYS UNIDIMENSIONALES: VECTORES.

Un array unidimensional, o lineal, o vector, es un conjunto finito y ordenado de elementos homogéneos.

Es finito porque tiene un número determinado de elementos. Homogéneo porque todos los elementos almacenados van a ser del mismo tipo. Ordenado porque vamos a poder acceder a cada elemento del array de manera independiente porque va a haber una forma de referenciar cada elemento. Para referenciar cada elemento de un array vamos a usar índices (valor que directa o indirectamente referencia la posición del array).

Los índices tienen que ser de cualquier tipo de datos escalar (entre los que se puede definir un orden, y que entre 2 elementos consecutivos no puede haber

infinitos elementos), aunque normalmente como índices se van a utilizar números enteros.

Para referenciar un elemento de un array usaremos el nombre del array y entre corchetes [ ] el índice que determina la posición de ese elemento en el array.

El rango o longitud de un vector o array lineal es la diferencia entre el índice de valor máximo y el índice de valor mínimo de ese array + 1. Normalmente los índices comienzan a enumerarse, es decir, el valor mínimo del índice es 0 ó 1, dependiendo del lenguaje (en Pascal con 1 y en C con 0). Sin embargo nadie impide que comiencen en cualquier otro valor.

Los arrays se almacenan siempre en posiciones consecutivas de memoria y podemos acceder a cada elemento del array de manera independiente a través de los índices. Un índice no tiene porque ser un valor constante, sino que puede ser también una variable o una expresión que al ser evaluada devuelva ese índice.

A la hora de definir un array siempre habrá que dar el nombre del array, el rango de sus índices y el tipo de los datos que contiene, y para hacer esa declaración, se utiliza la siguiente nomenclatura.

<nom_array>: array [LI .. LS] de <tipo>
sueldo: array [1 .. 8] de real
sueldo: array [1990 .. 1997] de real
sueldo [1992] ← 100000
I: entero
I ← 1992
Sueldo [I]
Sueldo [I+2]

3. OPERACIONES CON ARRAYS UNIDIMENSIONALES O VECTORES:

1. Asignación de un dato a una posición concreta del array:

<nom_array>[indice] ← valor

1	2	3	4	5	6	7	8	9	10	11	12

Ventas [3] ← 800000

2. Lectura y escritura de datos:

```
leer <nom_array>[indice]
escribir <nom_array>[indice]
desde i=1 hasta 12
  escribir "Introduce las ventas del mes"i
  leer ventas [i]
fin desde
desde i=1 hasta 12
    escribir "Ventas del mes"i"="ventas [i]
fin desde
```

3. Recorrido o acceso secuencial de un array:

- Consiste en pasar por todas las posiciones del array para procesar su información.

Desde i=1 hasta 12
Ventas [i] ← ventas [i] + 1000000
Fin desde

4. Actualización de un array:

1º) Añadir datos:

Es un caso especial de la operación de insercción de un elemento en un array, pero el elemento lo metemos después de la última posición que contiene información válida en el array.

Para que esto se pueda hacer es necesario que si actualmente el array tiene K posiciones de información válida, tenga un tamaño de al menos K+1 para que pueda añadir otro elemento a continuación de K.

<nom_array>[K+1] ← valor

2º) Inserción de datos:

Consiste en introducir un elemento en el interior de un array para lo cual será necesario desplazar todos los elementos situados a la derecha del que vamos a insertar una posición a la derecha con el fin de conservar el orden relativo entre ellos.

Para que se pueda insertar un nuevo elemento en el array si ya existen N elementos con información en el array, el array tendrá que tener un tamaño de cómo mínimo N+1 para poder insertar el elemento.

C	E	F	J	M	O		

"G"

Siendo K la posición en la que tengo que insertar el nuevo elemento y N el número de elementos válidos en el array en el momento de la inserción y siempre suponiendo de N+1, el algoritmo de inserción será:

Desde i=N hasta K
A[I+1] ← A[I]
Fin desde
A[K] ← valor

3º) Borrar datos:

Para eliminar un elemento de un array si ese elemento está posicionado al final del array, no hay ningún problema, simplemente si el tamaño del array era N, ahora hay que considerar que el tamaño del array es N-1.

Si el elemento a borrar ocupa cualquier otra posición entonces tendré que desplazar todos los elementos situados a la derecha del que quiero borrar una posición hacia la izquierda para que el array quede organizado.

C	E	F	J	M	O		

Borrar J.

Suponiendo que el número de elementos validos actualmente es N y que quiero borrar el elemento de la posición K.

Desde i=K hasta N-1
A[I] ← A[I+1]
Fin desde

Para indicar que el número de elementos validos es N, podríamos indicarlo como N ← N-1.

4. ARRAYS BIDIMENSIONALES O MATRICES:

En un array unidimensional o vector cada elemento se referencia por un índice, en un array bidimensional cada elemento se va a referenciar por 2 índices, y ahora la representación lógica ya no va a ser un vector, sino una matriz.

Un array bidimensional de M*N elementos es un conjunto de M*N elementos, todos del mismo tipo, cada uno de los cuales se referencia a través de 2 subíndices. El primer subíndice podrá variar entre 1 y M si hemos empezado a numerar los índices por 1, y el segundo índice variará entre 1 y N, si hemos empezado a numerar los índices por el 1.

Y más en general podemos definir un array de 2 dimensiones de la siguiente manera.

<nom_array>: array [LI1..LS2,LI2..LS2] de <tipo>
<var_array>[I , J]
LI1 <= I <= LS1
LI2 <= J <= LS2
El tamaño del array será (LS1-LI1 +1)*(LS2-LI2 +1)

Ventas de cada mes de 1990 a 1995:
Ventas: array [1990..1995,1..12] de real
6*12=72

La representación lógica de un array bidimensional es una matriz de dimensiones M*N donde M es el número de filas de la matriz y N es el número de columnas, es decir, la 1ª dimensión indicaría las filas y la 2ª dimensión indicaría las columnas, es decir, al intentar acceder a un elemento I,J estaríamos accediendo al elemento que ocupa la fila I y la columna J.

En memoria, sin embargo todos los elementos del array se almacenan en posiciones contiguas pero nosotros lo vemos como una matriz.

	1	2	3	4	5	6	7	8	9	10	11	12
1990												
1991												
1992												
1993			X									
1994												
1995												

Ventas [1993,3]

En memoria estaría almacenado todo seguido:

Manejo de matrices:

- Para poner a 0 todos los elementos de la matriz.

```
M: array [1..N,1..M] de entero
Var i,j entero
Desde i=1 hasta n
  Desde j=1 hasta M
    M [i,j] ← 0
  Fin desde
Fin desde
```

- Para poner a 0 solo los elementos de la fila I:

```
Desde i=1 hasta N
  M [i,j] ← 0
Fin desde
```

5. ARRAYS MULTIDIMENSIONALES:

Un array multidimensional es un array de 3 ó más dimensiones. Si tenemos un array de N dimensiones, cada dimensión de tamaño d1,d2,..,dN, el número de elementos del array será d1*d2*..*dN, y para acceder a un elemento concreto del array utilizaremos N índices, cada uno de los cuales referenciará a una posición dentro de una dimensión, siempre según el orden de declaración.

En memoria, el array se sigue almacenando en posiciones consecutivas.

La declaración de un array multidimensional sería:

Nom_array: array [LI1..LS1,LI2..LS2,LI3..LS3,LIN..LSN] de tipo
Nom_array [I1,I2,I3,IN]

LI1 <= I1 <= LS2

LIN <= I2 <= LSN

6. ALMACENAMIENTO DE ARRAYS EN MEMORIA:

Un array en memoria siempre se almacena en posiciones contiguas a partir de la dirección base de comienzo del array que me la dan cuando yo declaro una variable del tipo array.

El tamaño que ocupa el array en memoria es el producto del número de sus elementos por el tamaño de cada uno de ellos.

Array [1..100] de carácter

1 byte * 100 elementos = 100 bytes

En el caso de un array bidimensional, también se seguirá almacenando en posiciones contiguas de memoria, y su tamaño será el producto de sus elementos pro el tamaño.

Ahora, la forma en la que están almacenados esos elementos va a depender de que el array se almacene fila a fila o columna a columna, que también se conoce como almacenamiento por orden de fila mayor o por orden de columna mayor.

C1	C2	.....	Cm	C1	C2	...	Cm					
		F1			F2				F3			

```
A: array [1..N,1..M] de <tipo>
Fila
```

1,1	1,2	1,3	.....	1,M	2,1	2,2	2,3	....	2,M	N,1	N,2	...	N,m	

Como hallar la dirección base en la que comienza a almacenarse un determinado elemento de un array:

- Unidimensionales:

DIR(A[I])= Dir_Base + (I-1) * Tamaño

Número elemento = 3

Tamaño = 5

Dir_Base = 100

A[3] comienza en la dirección 110

- Bidimensionales:

- **Orden de fila mayor**
  Dir(A[i,j])= Dir_Base + [((I-1)*M)+(J-1)]*Tamaño
- **Orden de columna mayor**
  Dir(A[i,j])= Dir_Base + [((J-1)*N)+(I-1)]*Tamaño

7. ARRAYS COMO PARÁMETROS DE SUBPROGRAMAS:

Un array es también un tipo de datos estructurado, y se puede pasar como parámetro a un subprograma, pero el problema es que no sabemos como indicar si su paso es por valor o por referencia.

En principio, el paso de un array siempre tiene sentido que sea por referencia (que se modifique y tenga repercusión en el exterior), y si es por valor solo serviría para leer información.

La mayor parte de los lenguajes de programación, pasan los arrays por referencia (si se modifica el array se nota en el exterior). Nosotros también lo haremos así.

Si paso el array y no quiero modificarlo, con no modificarlo basta.

Es distinto pasar un array como parámetro a pasar una posición del array, si paso el array entero como parámetro, el parámetro formal asociado será un array con la misma estructura que el parámetro real, pero si lo que quiero pasar es un elemento de una posición concreta del array, el parámetro formal asociado no será del tipo array, sino que será del tipo de datos que contenga el array, porque cada elemento de un array es un elemento independiente que se puede usar por separado.

En el caso de que pasemos un array entero como parámetro, el parámetro formal asociado, tendrá que ser un array de la misma estructura que tiene el mismo tipo de datos y el mismo tamaño.

En C, el parámetro formal asociado tiene que ser del mismo tipo de datos, pero no hace falta que sea del mismo tamaño.

Para pasar un array como parámetro real, utilizare el nombre del array, en cambio si pasamos una posición concreta, usaré el nombre y el índice.

En C, el nombre de un array indica su dirección de comienzo en memoria, y por eso siempre se pasa por referencia.

Siempre vamos a pasar los arrays por referencia, sin la palabra clave ent-sal.

Como en las funciones no se pasan los parámetros por referencia, lo mejor es no pasar los arrays como parámetros de las funciones.

Procedimiento <nombre_p> (<nom_par>:array [LI1..LS1] de <tipo>; ...)

Var A: array [LI1..LS1] de <tipo>

Nom_p (array) → Sólo se pasa el nombre.

8. ARRAYS DE “PUNTEROS”:

En realidad lo que vamos a explicar con este concepto es la posibilidad de que desde un array se apunte a la posición de otro array. Que unos arrays sirvan para referenciar a otros.

¿Qué es un puntero?:

Un puntero es una variable que contiene una dirección en la que está almacenado un elemento que llamamos apuntado.

Desde este punto de vista, un índice indirectamente también es un puntero porque indirectamente indica una posición del array, pero no lo es directamente porque no almacena una dirección.

Desde este punto de vista, si un array contiene valores de índices, es decir, que cada posición de ese array lo que hace es apuntar a la posición de otro array, entonces decimos que es un array de punteros, aunque en realidad la definición mas correcta es que es un array de índices a otro array.

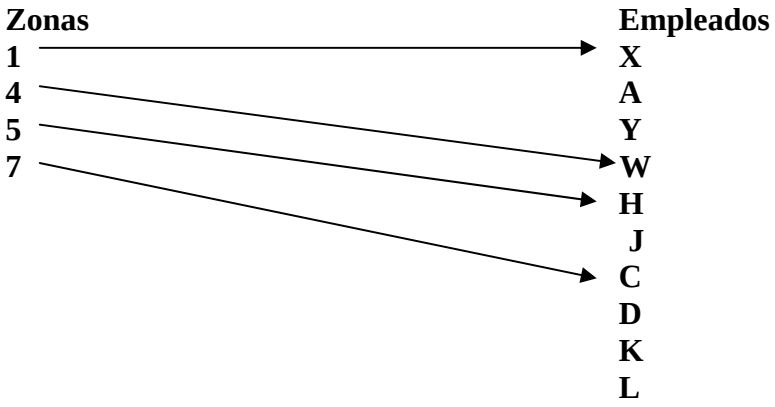
¿Cuándo es útil usar un array de índice?:

Imaginemos que tenemos una compañía que quiere tener almacenados los datos de sus empleados organizados por zonas. Imaginemos que hay 4 zonas.

Z1	Z2	Z3	Z4
X	W	H	C
A		J	D
Y			K
			L

Podemos almacenar esta información de distintas formas:

- **Un array bidimensional 4*N, en donde 4 son las zonas y N es el número máximo de empleados que hay por zona.**
  El inconveniente es la perdida de espacio.
- **Usar un array unidimensional o vector donde estén todos los datos seguidos.**
  El inconveniente es que no tengo los empleados ordenados por zonas.
- **Usar un array unidimensional, y entre cada 2 zonas meter en la posición que va entre ellas una marca especial para indicar que paso de zona.**
  El inconveniente es que para pasar a una zona tengo que pasar por todas las anteriores.
- **Usar un array de índices:**
  Un array con la información de los empleados.
  Otro array de índices que contendrá tantas posiciones como zonas haya.
  En cada posición de cada zona almacenaré el índice en donde comienza el primer empleado de esa zona en el array de empleados.



*** Listar todos los empleados de la zona 3:**

```
Desde i=zona[3] hasta zona[4]-1
   Escribir Empleados [i]
Fin desde
```

*** Número de empleados:**

Zona [4] – Zona [3]

EJERCICIOS: TEMA 6

1. **Hay unos multicines con 5 salas, y cada sala con 100 personas distribuidas en 20 asientos y 5 filas.**
   Si yo pido entrada para una sala, implementar un programa que me diga si hay sitio en la sala.

```
Algoritmo cines
Const
  Salas=5
  Asientos=20
  Filas=5
Var
  N_salas,j,k: entero
  Marca: boolean
  A: array [1..salas,1..asientos,1..filas] de entero
Inicio
  Pedir_datos (n_salas)
  Mientras n_salas <> 0
    Marca ← falso
    J ← 0
    K ← 1
    Repetir
      Si j > asientos
        Entonces j ←1
                 K ← k+1
      Fin si
      Si (j=asientos) y (k>=filas)
        Entonces escribir "Sala llena"
                 Marca ← verdadero
        Sino si a [n_salas,j,k]=0
                 Entonces a[n_salas,j,k] ← 1
                           Escribir "Asiento"j"fila"k
                           Marca ← verdadero
              Fin si
      Fin si
    Hasta (a[n_salas,j,k]=1) y (marca=verdadero)
    Pedir_datos (n_salas)
  Fin mientras
Fin

Procedimiento pedir_datos (ent-sal s: entero)
Inicio
  Repetir
    Escribir "¿En qué sala quieres entrar?"
    Leer s
  Hasta (s>=0) y (s<=salas)
Fin pedir_datos
```

2. ¿Qué escribe este programa?

```
Algoritmo Prin
Var
  Datos: array [1..10] de entero
  i: entero
inicio
  desde i=1 hasta 10
    datos [i] ← i
  fin desde
  P1 (datos,datos[3])
  Desde i=1 hasta 10
    Escribir datos [i]
  Fin desde
Fin

Procedimiento P1 (a: array [1..10] de entero; ent-sal x: entero)
Inicio
  X ← 0
  Desde i=1 hasta 10
    X ← x + a[i]
  Fin desde
Fin P1
```

*** Solución: 1, 2, 55, 4, 5, 6, 7, 8, 9, 10.**

3. Dada una matriz A de M*N elementos, actualizarla tal que la matriz resultante tenga divididos a los elementos de la diagonal principal por la suma de los elementos que no forman parte de ella.

```
Algoritmo div_matriz
Var
  A: array [1..M,1..N] de real
  Suma: real
Inicio
  Pedir_datos (a)
  Sumar (a,suma)
  Escribir (a)
Fin

Procedimiento pedir_datos (matriz: array [1..M,1..N] de real)
Var
  I,j: entero
Inicio
  Desde i=1 hasta M
    Desde j=1 hasta N
      Escribir “Introduce el elemento”i”,”j
      Leer a[i,j]
    Fin desde
  Fin desde
```

```
Fin pedir_datos
Procedimiento sumar (matriz: array [1..M,1..N] de real; ent-sal s: real)
Var
  I,j: entero
Inicio
  S ← 0
  Desde i=1 hasta M
    Desde j=1 hasta N
      Si i< > j
        Entonces s ← s + matriz [i,j]
      Fin si
    Fin desde
  Fin desde
Fin sumar

Procedimiento escribir (matriz: array [1..M,1..N] de real; s: real)
Var
  I,j: entero
Inicio
  Desde i=1 hasta M
    Desde j=1 hasta N
      Si i=j
        Entonces escribir a[i,j]/s
        Sino escribir a[i,j]
      Fin si
    Fin desde
  Fin desde
Fin escribir
```

4. Tengo guardado en una estructura los alumnos de nuestra escuela, sabiendo que hay 3 cursos, M alumnos por curso y N asignaturas por alumno, determinar mediante subprogramas:

1. **Cual es la nota media de un determinado curso.**
2. **Cuantos aprobados y suspensos hay en una determinada asignatura.**
3. **Cual es el alumno de la escuela con mejor nota media.**

```
Algoritmo escuela
Const
  Cursos=3
  Alumnos=M
  Asignaturas=N
Tipo
  Dato=array [1..cursos,1..alumnos,1..asignaturas] de real
Var
  Nota: dato
Inicio
  Pedir_datos (nota)
  Media_curso (nota)
  Ap_susp (nota)
  Media_alum (curso)
```

```
Fin
Procedimiento pedir_datos (n: datos)
Var
  I,j,k: entero
Inicio
  Desde i=1 hasta cursos
    Desde j=1 hasta alumnos
      Desde k=1 hasta asignaturas
        Repetir
          Escribir “Nota del alumno”j”asignatura”k”curso”i
          Leer n[i,j,k]
        Hasta (n[i,j,k>=0) y (n[i,j,k] <=10)
      Fin desde
    Fin desde
  Fin desde
Fin pedir_datos

Procedimiento media_curso (n: dato)
Var
  J,k,resp: entero
  Media,suma: entero
Inicio
  Suma ← 0.0
  Repetir
    Escribir “¿De qué curso quieres hacer la media?”
    Leer resp
  Hasta (resp<=1) y (resp<=cursos)
  Desde j=1 hasta alumnos
    Desde k=1 hasta asignaturas
      Suma:=suma + n[resp,j,k]
    Fin desde
  Fin desde
  Media ← suma /alumnos*asignatura
  Escribir “La nota media del curso”resp”es”media
Fin media_curso
```

```
Procedimiento ap_susp (n:dato)
Var
  Susp,ap,i,j: entero
Inicio
  Susp ←0
  Ap ← 0
  Repetir
    Escribir "¿Qué asignatura quieres ver?"
    Leer asig
  Hasta (asig>=1) y (asig<=asignaturas)
  Desde i=1 hasta cursos
    Desde j=1 hasta alumnos
      Si n[i,j,asig] >=5
        Entonces ap ← ap + 1
        Sino susp ← susp + 1
      Fin si
    Fin desde
  Fin desde
  Escribir "En la asignatura"asig"hay"ap"aprobados"
  Escribir "En la asignatura"asig"hay"susp"suspensos"
Fin ap_susp

Procedimiento media_alum (n: dato)
Var
  I,j,alum,curs: entero
  Suma,media,mayor: real
Inicio
  Mayor ← 0.0
  Desde i=1 hasta cursos
    Desde j=1 hasta alumnos
      Suma ← 0.0
      Desde k=1 hasta asignaturas
        Suma ← suma + n[i,j,k]
      Fin desde
      Media ← suma / asignaturas
      Si media > mayor
        Entonces mayor ← media
                 Curs ← i
                 Alum ← j
      Fin si
    Fin desde
  Fin desde
  Escribir "El alumno con mayor media es el"alum"del curso"curs
  Escribir "y su nota es de"mayor
Fin media_alum
```

5. Multiplicar 2 matrices de dimensiones M*N y P*Q.

```
Algoritmo multiplicar_matrices
Tipo
  Matriz1=array [1..M,1..N] de entero
  Matriz2=array [1..P,11..Q] de entero
  Resultado=array [1..M,1..Q] de entero
Var
  Mat1: matriz1
  Mat2: matriz2
  R: resultado

Inicio
  Pedir_datos (mat1,mat2)
  Multiplicar (mat1,mat2,r)
  Visualizar (r)
Fin

Procedimiento pedir_datos (m1: matriz1; m2: matriz2)
Var
  I,j: entero
Inicio
  Desde i=1 hasta M
    Desde j=1 hasta N
      Escribir "Introduce el elemento"i","j"de la matriz 1"
      Leer m1[i,j]
    Fin desde
  Fin desde
  Desde i=1 hasta P
    Desde j=1 hasta Q
      Escribir "Introduce el elemento"i","de la matriz 2"
      Leer m2[i,j]
    Fin desde
  Fin desde
Fin pedir_datos

Procedimiento multiplicar (m1: matriz1; m2: matriz2; ent-sal resul: resultado)
Var
  Fila,i,j,acum: entero
Inicio
  Desde fila=1 hasta N
    Desde i=1 hasta P
      Resul[fila,i] ← 0
      Desde j=1 hasta Q
        Resul[fila,i] ← resul[fila,i]+m1[fila,j]*m2[j,i]
      Fin desde
    Fin desde
  Fin desde
Fin multiplicar
```

```
Procedimiento visualizar (resul: resultado)
Var
  I,j: entero
Inicio
  Desde i=1 hasta M
    Desde j=1 hasta Q
      Escribir resul[i,j]
    Fin desde
  Fin desde
Fin visualizar
```

6. Resolver la siguiente serie:

$$X = \frac{\sum_{i=1}^{m} i * \sum_{j=1}^{n} a[i, j] * j}{\sum_{j=2}^{n-1} b[n, j]}$$

```
Algoritmo serie
Tipo
  Matriz1= array [1..M,1..N] de entero
  Matriz2= array [1..N,1..N] de entero
Var
  Suma1,suma2,suma3: entero
  X: real

Inicio
  Sumas (suma1,suma2,suma3)
  Resultado (suma1,suma2,suma3,x)
  Escribir "El resultado es"x)
Fin

Procedimiento sumas (s1: entero;s2: entero;s3: entero)
Var
  I,j: entero
Inicio
  Desde i=1 hasta M
    S2 ← 0
    Desde j=1 hasta N
      S2 ← s2 + a[i,j]*j
    Fin desde
    S1 ← s1 + I*s2
  Fin desde
  S3 ← 0
  Desde j=2 hasta N-1
    S3 ← s3 + b[n,j]
  Fin desde
Fin sumas
```

```
Procedimiento resultado (s1: entero; s2: entero; s3: entero; ent-sal y: real)
Inicio
  Y ← s1*s2/s3
Fin resultado
```

7. Una empresa consta de 5 departamentos con 20 empleados cada departamento, si tengo todas las ventas en una estructura, determinar:

- **Ventas de un determinado departamento en un determinado mes.**
- **Ventas de un determinado empleado en un determinado departamento.**
- **Cual es el departamento con más ventas.**

```
Algoritmo empresa
Const
  D=5
  E=20
  M=12
Tipo
  Matriz= array [1..d,1..e,1..m] de real
Var
  Ventas: matriz
  Cantidad: real
  Departamentos: entero
Inicio
  Pedir_datos (ventas)
  Dep_mes (ventas)
  Dep_empleado (ventas)
  Mejor_dep (ventas)
Fin

Procedimiento pedir_datos (a: matriz)
Var
  I,j,k: entero
Inicio
  Desde i=1 hasta D
    Desde j=1 hasta E
      Desde k=1 hasta M
        Escribir “Ventas del departamento”i”empleado”j”mes”k”
        Leer a[i,j,k]
      Fin desde
    Fin desde
  Fin desde
Fin pedir_datos
```

```
Procedimiento dep_mes (a: matriz)
Var
  Dep,j,mes: entero
  V: real
Inicio
  V ← 0.0
  Repetir
    Escribir “Departamento”
    Leer dep
  Hasta (dep >=1) y (dep <=d)
  Repetir
    Escribir “Mes”
    Leer mes
  Hasta (mes >=1) y (mes <=12)
  Desde j=1 hasta E
    V ← V + a[dep,j,mes]
  Fin desde
  Escribir “Las ventas del departamento”dep”en el mes”mes”son”v
Fin dep_mes

Procedimiento dep_empleado (a: matriz)
Var
  Dep,empleado,k: entero
  V: real
Inicio
  V ← 0.0
  Repetir
    Escribir “Empleado”
    Leer empleado
  Hasta (empleado >=1) y (empleado <=E)
  Repetir
    Escribir “Departamento”
    Leer dep
  Hasta (dep >=1) y (dep <=d)
  Desde k=1 hasta M
    V ← V + a[dep,empleado,k]
  Fin desde
  Escribir “Las ventas del empleado”empleado”del departamento”dep”son”v
Fin dep_empleado
```

```
Procedimiento mejor_dep (a: matriz)
Var
  I,j,k,dep: entero
  Mayor,v: real
Inicio
  Mayor ← 0.0
  Desde i=1 hasta D
    V ← 0
    Desde j=1 hasta E
      Desde k=1 hasta M
        V ← V + a[i,j,k]
      Fin desde
    Fin desde
    Si v > mayor
      Entonces mayor ← v
                Dep ← i
    Fin si
  Fin desde
  Escribir "El mejor departamento es el"dep"con"mayor
Fin mejor_dep
```

8. Dado un array A de M*N elementos donde los elementos son números enteros, hallar la dirección de comienzo del elemento 4º del array sabiendo que se almacena a partir de la dirección 1200 y que en nuestra máquina los enteros ocupan 2 bytes.

Dir A(1,4)= 1200 + 3*2 =1206

9. Dado un array de 4 * 5 elementos que contiene caracteres, sabiendo que se almacena a partir de la posición 500, en que posición comienza a almacenarse el elemento A[3,5].

Dir A[3,5] = 500 + (2*5+4)*1 = 514 → En orden de fila mayor
Dir A[3,5] = 500 + (4*4+2)*1 = 518 → En orden de columna mayor

LAS CADENAS DE CARACTERES
TEMA 7

1. **Juego de caracteres.**
2. **Cadena de caracteres.**
3. **Datos de tipo carácter.**
4. **Operaciones con cadenas.**

1. JUEGO DE CARACTERES:

En principio se programaba todo con 0 y 1, pero como esto costaba mucho, apareció la necesidad de crear un lenguaje semejante al humano para entendernos más fácilmente con la computadora, y para ello aparecen los juegos de caracteres.

El juego de caracteres es una especie de alfabeto que usa la máquina.

Hay 2 juegos de caracteres principales:

- **ASCII: El que más se usa.**
- **EBCDIC: Creado por IBM.**

Hay 2 tipos de ASCII, el básico y el extendido.

En el ASCII básico, cada carácter se codifica con 7 bits, por lo que existen 2^7=128 caracteres.

En el ASCII extendido, cada carácter ocupa 8 bits (1 byte) por lo que existirán 2^8= 256 caracteres, numerados del 0 al 255.

En el EBCDIC, cada carácter ocupa también 8 bits.

En cualquiera de los 2 juegos, existen 4 tipos de caracteres:

- **Alfabéticos: Letras mayúsculas y minúsculas.**
- **Numéricos: Números.**
- **Especiales: Todos los que no son letras y números, que vienen en el teclado.**
- **Control: No son imprimibles y tienen asignados caracteres especiales. Sirven para de terminar el fin de línea, fin de texto. Van del 128 al 255.**

Un juego de caracteres es una tabla que a cada número tiene asociado un número.

2. CADENA DE CARACTERES:

Es un conjunto de 0 ó más caracteres. Entre estos caracteres puede estar incluido el blanco.

En pseudocódigo, el blanco es el b. Las cadenas de caracteres se delimitan con dobles comillas “ “, pero en algunos lenguajes se delimitan con ‘ ‘.

Las cadenas de caracteres se almacenan en posiciones contiguas de memoria.

La longitud de una cadena es el número de caracteres de la misma. Si hubiese algún carácter que se utilizara como especial para señalar el fin de cadena, no se consideraría en la longitud.

Si una cadena tiene longitud 0, la llamamos cadena nula por lo que no tiene ningún carácter, pero esto no quiere decir que no tenga ningún carácter válido, por que puede haber algún carácter especial no imprimible que forme parte de la cadena.

Una subcadena es una cadena extraída de otra.

3. DATOS DE TIPO CARÁCTER:

1. **Constantes: Una constante de tipo cadena es un conjunto de 0 o más caracteres encerrados entre " ".**
   Si dentro de la cadena quiero poner como parte de la cadena las ", las pongo 2 veces. Esto depende del lenguaje.
   "Hola""Adios" → Hola"Adios
   En algunos lenguajes hay un carácter de escape. En C, el carácter de escape es la \.
   Una constante de tipo carácter es un solo carácter encerrado entre comillas simples.
2. **Variables: Hay que distinguir entre una variable de tipo carácter y una variable de tipo cadena, el contenido de una variable de tipo cadena es un conjunto de 0 ó más caracteres encerrados entre " ", mientras que una variable de tipo carácter es un solo carácter encerrado entre ' '.**

Formas de almacenamiento de cadenas en memoria:

1. **Almacenamiento estático:**
   La longitud de la cadena se tiene que definir antes de ser usada y siempre va a tener esa longitud, almacenándose en posiciones contiguas de memoria.
   Si la cadena no ocupa todo el espacio, el resto se rellena con blancos, y esos blancos se consideran parte de la cadena.
   Esto es muy deficiente y no se usa casi en ningún lenguaje.
2. **Almacenamiento semiestático:**
   Antes de usar la cadena, hay que declarar la longitud máxima que puede tener y ese es el espacio que se reserva en memoria para almacenar la cadena, siempre en posiciones contiguas.
   La longitud real de la cadena durante la ejecución puede variar aunque siempre tiene que ser menor que el máximo de la cadena.
   Puesto que la cadena no tiene porque ocupar la longitud máxima, para determinar que parte ocupa realmente esa cadena, se pueden utilizar diferentes métodos.
   Pascal lo que hace es reservar los 2 primeros bytes de la zona de memoria en que guardamos la cadena para indicar el primero la longitud máxima que puede tener la cadena y el segundo la longitud actual.

10	4	H	O	L	A						
		1	2	3	4	5	6	7	8	9	10

Otros lenguajes como C, utilizan un carácter especial que indica el fin de cadena tal que los caracteres que utilizan parte de la cadena son todos los almacenados hasta encontrar ese carácter especial.

H	O	L	A	\O					
0	1	2	3	4	5	6	7	8	9

La diferencia entre almacenar un solo carácter en un tipo carácter o en un tipo cadena, sería la siguiente:

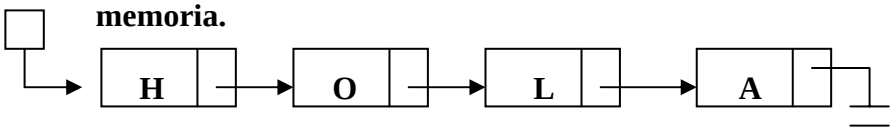
C ← 'a'

C ← "a"

3. Almacenamiento dinámico:

No hay que definir la longitud de la cadena antes de usarla, ni siquiera la máxima. Para esto, se utiliza la memoria dinámica, y para establecer el número de elementos de la cadena usaremos listas enlazadas en las que cada nodo de la lista contara un carácter de la cadena y se enlazaría mediante punteros.

La información no tiene que estar almacenada en posiciones contiguas de memoria.



4. OPERACIONES CON CADENAS:

Al igual que con cualquier tipo de datos, podemos hacer operaciones de entrada y salida (leer y escribir).

Var cad: cadena

Leer (cad)

Escribir (cad)

Aparte de estas instrucciones, la mayor parte de los lenguajes permiten realizar operaciones especiales con las variables de tipo cadena.

La mayor parte de los lenguajes tienen operaciones de tratamiento de cadenas, y esas operaciones vienen en librerías externas.

Las operaciones más usadas son:

Longitud de una cadena:

Es una función a la que se le pasa una cadena como parámetro y como resultado devuelve su longitud.

Funcion longitud (c:cadena): entero

Comparación de cadenas:

Las cadenas se pueden comparar entre si usando los símbolos de comparación. Esto se puede realizar porque lo que voy a comparar son los valores ASCII asociados a cada carácter.

En el caso de que se comparen 2 cadenas de diferente longitud tal que la cadena de menor longitud tiene N caracteres y estos N caracteres coinciden con los N primeros caracteres de la cadena más larga, se considera mayor la cadena más larga.

PEPE > PAPA

PEPES > PEPE

En la mayor parte de los lenguajes, hay una función que hace la comparación.

En C es la función strcmp (C1,C2).

Funcion comparacion (C1:cadena;C2:cadena): entero
Esta función devuelve:
- **0 si C1=C2**
- **Un positivo si C1 > C2**
- **Un negativo si C1 < C2**

Concatenación:

Lo que permite es unir varias cadenas en una sola, manteniendo el orden de los caracteres que se unen.

En pseudocódigo se usa el símbolo &: C1&C2
C1=”Hola”
C2=”Adios”
C3=C1&C2=”HolaAdios”
Procedimiento concatenacion (ent-sal C1:cadena;C2:cadena)
- **Se devuelve en C1 la concatenación C1&C2.**

Subcadenas:

Extrae parte de una cadena.

Se pueden usar 3 procedimientos:

Procedimiento subcadena (c:cadena;inicio:entero;longitud:entero;ent-sal s:cadena)
- **Si inicio es negativo, o es mayor que longitud, se devuelve la cadena nula.**
- **Si inicio+longitud es mayor que el tamaño de la cadena, devuelvo desde inicio hasta de fin de la cadena.**

Procedimiento subcadena (c:cadena;inicio:entero;fin:entero;ent-sal s:cadena)
- **Si fin es menor que inicio, devuelve la cadena nula.**
- **Si fin es mayor que la longitud de la cadena, se devuelve desde inicio hasta el fin de la cadena.**

Procedimiento subcadena (c:cadena;inicio:entero;ent-sal s:cadena)
- **S va desde el inicio hasta el final de la cadena.**

Inserción:

Consiste en meter una cadena dentro de otra a partir de una determinada posición.

Procedimiento insertar (ent-sal C1:cadena;C2:cadena;inicio:entero)
C1=”Pepe”
C2=”Nuria”
Insertar (C1,C2,3)= PeNuriape
- **Si inicio es mayor que la longitud de C1 o inicio es menor que 0, no se inserta nada.**

Borrado:

Consiste en borrar de una cadena una subcadena que forma parte de ella. Hay que dar la posición de inicio y final de la subcadena que quiero borrar.

Procedimiento borrar (ent-sal C1:cadena;inicio:entero;fin:entero)

- **Si fin es menor que inicio, o inicio es mayor que la longitud de la cadena, no se borra nada.**

Procedimiento borrar (ent-sal C1:cadena;inicio:entero;fin:entero)

C1=”Casa azul”
Borrar (C1,3,5)
C1=”Caazul”

Intercambio:

Consiste en substituir la aparición de una subcadena dentro de una cadena, por otra subcadena. Para eso la primera subcadena tiene que aparecer en la otra.

Procedimiento intercambio (ent-sal C1:cadena;S1:cadena;S2:cadena)

C1=”Casa azul”
S1=”asa”
S2=”asita”
C1=”Casita azul”
Si S1 no está en C1, no se cambia nada.

Conversión de cadenas a números:

Es una función que se le pasa una cadena caracteres numéricos y devuelve el número asociado.

Funcion valor (c:cadena): entero
I=”234”
Var i: entero
I= valor (C1) devuelve 234.

Conversión de números a cadenas:

Es un procedimiento al que se le pasa un número y lo convierte a una cadena.

Procedimiento conv_cad (n:entero;ent-sal c:cadena)
I=234
Conv_cad (i,C1) devuelve “234”

Función que devuelve el carácter ASCII de un número:

Funcion conv_car (n:entero): carácter

Función que devuelve el número asociado de un carácter ASCII:

Funcion ASCII (c:carácter): entero

Las cadenas van a estar almacenadas en arrays de caracteres donde el carácter de fin de cadena es el $.

EJERCICIOS: TEMA 7

- **Los ejercicios de este tema van a consistir en implementar funciones y procedimientos relacionados con el manejo de las subcadenas.**

```
Funcion numero (c:carácter):entero
Inicio
  Según sea C
    '0': retorno 0
    '1': retorno 1
    '2': retorno 2
    '3': retorno 3
    '4': retorno 4
    '5': retorno 5
    '6': retorno 6
    '7': retorno 7
    '8': retorno 8
    '9': retorno 9
  fin según sea
fin numero
```

```
Funcion caract (n: entero):carácter
Inicio
  Según sea C
    0: retorno '0'
    1: retorno '1'
    2: retorno '2'
    3: retorno '3'
    4: retorno '4'
    5: retorno '5'
    6: retorno '6'
    7: retorno '7'
    8: retorno '8'
    9: retorno '9'
  fin según sea
fin caract
```

```
Funcion longitud (c:cadena): entero
var
  l: entero
Inicio
  L ← 0
  Mientras c[L+1] < > '$'
    L ← L+1
  Fin mientras
  Retorno L
Fin longitud


Funcion Valor (c:cadena): entero
Var
  Cif,num,i: entero
Inicio
  Cif ← longitud (c)
  Si cif > 0
    Entonces num ← 0
              Desde i=1 hasta Cif
                Num ← num+numero (c[i]*10^(cif-i)
              Fin desde
              Retorno (num)
    Sino retorno (-1)
  Fin si
Fin valor

Procedimiento conv_cad (n: entero; ent-sal c: cadena)
Var
  Cif,aux: entero
Inicio
  Cif ← 1
  Aux ← N
  Mientras (aux >10)
    Aux ← aux div 10
    Cif ← cif + 1
  Fin mientras
  Desde i=cif hasta 1
    C[i] ← caract(n mod 10)
    N ← n div 10
  Fin desde
  C[cif+1] ← '$'
Fin conv_cad

Procedimiento borrar (ent-sal c:cadena;inicio:entero;fin:entero)
Var
  I,f: entero
Inicio
```

```
  Si (inicio >0) y (inicio <= fin) y (inicio <= longitud(c))
    Entonces i ← inicio
                F ← fin +1
                Mientras (i <=fin) y (c[f] < > '$')
                  C[i] ← c[f]
                   I ← i+1
                   F ← f+1
                 Fin mientras
                 C[i] ← '$'
  Fin si
Fin borrado
```

```
Procedimiento subcadena (c1:cadena;inicio:entero;longitud:entero;ent-sal c2: cadena)
Var
  I,k: entero
Inicio
  Si (inicio <=0) o (inicio > longitud (c1))
    Entonces c2[1] ← '$'
    Sino i ← inicio
          K ← 1
          Mientras (i <=inicio + longitud) y (c1[i] < > '$')
            C2[k] ← c1[i]
            K ← k+1
            I ← i+1
          Fin mientras
          C2[k] ← '$'
  Fin si
Fin subcadena

Funcion comparacion (c1:cadena;c2:cadena):entero
Var
  I: entero
Inicio
  I ← 1
  Mientras (c1[i]=c2[i]) y (c1[i]< > '$') y (c2[i] < > '$')
    I ← I+1
  Fin mientras
  Si c1[i]=c2[i]
    Entonces retorno (0)
    Sino retorno(ascii (c1[i])- ascii (c2[i]))
  Fin si
Fin comparacion
```

```
Funcion busqueda (c: cadena;s1: cadena): entero
Var
  I,pos,k: entero
  Encontrado: booleano
Inicio
  I ← 1
  Encontrado ← falso
  Mientras (c[i] < >'$') y (encontrado=falso)
    Si c[i] = s[i]
      Entonces pos ← i
                K ← 1
                Mientras (c[i]=s[k]) y (c[i] <>'$') y (s[k]< >'$'
                  I ← I+1
                  K ← k+1
                Fin mientras
                Si s[k]='$'
                  Entonces encontrado ← verdadero
                  Sino I ← pos +1
                        Pos ← 0
                Fin si
      Sino I ← I+1
    Fin si
  Fin mientras
  Retorno (pos)
Fin busqueda

Procedimiento borrar (ent-sal c:cadena;ini:entero;fin:entero)
Var
  I,j: entero
Inicio
  I ← ini
  F ← fin + 1
  Mientras c[f] <>'$'
    C[i] ← c[f]
    I ← i + 1
```

```
   Fin mientras
   C [i] ← '$'
Fin borrar
```

```
Procedure insertar (ent-sal c:cadena;s:cadena;pos:entero)
Var
   P,j,i: entero
Inicio
   P ← pos
   J ← 1
   Mientras s[j]<>'$'
      Desde i ← longitud (c) +1 hasta P
        C[i+1] ← c[i]
      Fin desde
      C[p] ← s[j]
      J ←  j+1
      P ← p+1
   Fin mientras
Fin insertar
```

```
Procedimiento intercambio (ent-sal c:cadena;c2:cadena;c3:cadena)
Var
   I,pos: entero
Inicio
   I ← 1
   Mientras c1[i] <>'$'
      Si c1[i]=c2[i]
        Entonces pos ← buscar(c1,c2)
                  Si pos <> 0
                    Entonces borrar (c1,pos,pos+longitud(c2)-1)
                              Insertar (c1,c3,pos)
                               I ← pos + longitud(c3)
                     Sino I ← i +1
                  Fin si
        Sino I ← I+1
      Fin si
   Fin mientras
Fin intercambio
```

FICHEROS O ARCHIVOS:
TEMA 8

1. **El tipo registro de datos.**
2. **Noción de archivo.**
3. **Terminología de archivos.**
4. **Tipos de soporte.**
5. **Tipos de organizaciones de ficheros.**
6. **Operaciones sobre ficheros.**
7. **Tratamiento de ficheros secuenciales.**
8. **Ficheros de texto.**

1. EL TIPO REGISTRO DE DATOS:

El tipo registro de datos es un tipo estructurado de datos. Un tipo registro va a estar formado por datos que pueden ser de diferentes tipos.

A cada uno de esos datos lo denominamos campos, y el tipo de estos campos pueden ser uno estandar o uno definido por el usuario, que puede ser cualquier cosa.

La sintaxis que usaremos para definir un tipo registro es la siguiente:

```
Tipo <nom_tipo_registro> = registro
        <campo1>: <tipo>
        <campo2>: <tipo>
        <campo3>: <tipo>
     fin registro
```

Una vez definida una estructura de tipo registro, ya puede pasar a declarar variables de ese tipo.

```
Var
  <nom_var>: <nom_tipo_registro>
```

```
tipo alumno = registro
       DNI: array [1..8] de caracteres
       Nombre: array [1..100] de caracteres
       Nota: real
     Fin registro
Var
```

A1,A2: alumno

Para declarar una variable de un tipo registro, basta con poner:
Var
<nom_var> : <nom_tipo_reg>

Para acceder a un campo concreto de una variable de tipo registro, utilizamos el operador punto.
<nom_var>.<nom_campo>
c.nombre ← Pepe

Y con ese valor puedo trabajar igual que trabajaría con cualquier valor del mismo tipo que ese campo.

```
Tipo  cliente = registro
          DNI: array [1..8] de caracteres
          Nombre: Cadena
          Saldo: Real
      Fin registro
Var
  C: cliente
Borrar (c.nombre,3,5)
C.saldo ← 5+3*8000
```

Almacenamiento de registros en memoria:

El tamaño que ocupa una variable de tipo registro en memoria es el que resulta de la suma del tamaño de cada uno de sus campos, y esa información también estará almacenada de manera contigua y según el orden en que hayamos declarado los campos.

2. NOCIÓN DE ARCHIVO:

Las estructuras anteriores se almacenaban en memoria principal, y el problema es que la información se pierde al apagar el ordenador. La ventaja es que los accesos son más rápidos. Para resolver esto están los dispositivos de almacenamiento secundario.

Fichero: Es un conjunto de datos estructurados en una colección de unidades elementales denominadas registros, que son de igual tipo y que a su vez están formados por otras unidades de nivel más bajo denominados campos. Todos son del mismo tipo.

3. TERMINOLOGÍA CON FICHEROS:

- **Campo: Es una unidad elemental de información que representa un atributo de una entidad. Se define con un nombre, un tamaño y un tipo de datos.**
- **Registro lógico: Es un conjunto de campos relacionados lógicamente que pueden ser tratados como unidad en el programa.**
  Desde el punto de vista de programación simplemente es una estructura de tipo registro.

- **Archivo o fichero: Es un conjunto de registros del mismo tipo y organizados de tal forma que esos datos pueden ser accedidos para añadir, borrar o actualizar.**
- **Clave de un fichero: Es un campo o conjunto de campos que sirve para identificar un registro y distinguirla del resto del fichero.**
- **Registro físico o bloque: Es la cantidad de datos que se transfieren en una operación de E/S entre el dispositivo externo y la memoria.**
  Desde este punto de vista, un registro físico puede estar compuesto por 0,1,2,... registros lógicos. El número de registros lógicos que hay por cada registro físico, es decir, que se transfiere en una operación de E/S, es lo que se denomina factor de bloqueo.
  Ese factor de bloqueo puede ser <1,=1,>1:
  - **Si es <1, quiere decir que el registro lógico ocupa más que el físico, se transfiere menos de un registro lógico en cada operación de E/S.**
  - **Si es =1, significa que el tamaño del registro lógico y el físico es el mismo, y se transfiere un registro lógico en cada operación de E/S.**
  - **Si es >1, lo más normal, en cada operación de E/S se transfiere más de un registro lógico.**

¿Cómo nos interesa que sea el factor de bloqueo?:

Desde un punto de vista, cuanto mayor sea el factor de bloqueo más registros lógicos se transferirán, menos operaciones habrá que hacer y se tardará menos en procesar el fichero. Según esto cuanto mayor sea mejor.

Por otra parte, cada vez que se transfiere información de E/S, se deja en una zona de memoria especial llamada *buffer*. Cuanto mayor sea el tamaño de bloqueo, mayor será el buffer y menos memoria me quedará para el resto de la información.

Al final hay que lograr un equilibrio para determinar el tamaño idóneo.

- **Bases de datos: Es un conjunto de datos relacionados almacenados internamente en un conjunto de ficheros.**

4. TIPOS DE SOPORTE:

Los soportes de almacenamiento secundario son en los que almaceno la información, y pueden ser de 2 tipos:

- **Secuenciales: Para acceder a un registro o dato concreto dentro de él, tengo que pasar previamente por todos los registros anteriores a él. El ejemplo es una cinta magnética.**
- **Direccionables: Es posible acceder directamente a una dirección concreta de soporte. El ejemplo es un disco.**

5. TIPOS DE ORGANIZACIONES DE FICHEROS:

Organización de archivos:

Viene determinada por 2 características:

- **Método de organización: Técnica que utilizo para colocar la información de los registros dentro del dispositivo.**
- **Método de acceso: Conjunto de programas que me permiten acceder a la información que previamente he almacenado y van a depender mucho del método de organización:**

- **Acceso directo: Para acceder a un acceso concreto no hay que pasar por los anteriores. El soporte tiene que ser direccionable.**
- **Acceso secuencial: Para acceder a un registro hay que pasar por todos los anteriores, y esto es posible si el soporte es secuencial, aunque también puedo hacerlo en uno direccionable.**

Métodos de organización:

Hay 3 tipos de organización:

- **Secuencial**
- **Directa**
- **Secuencial indexada**

- Secuencial: Los registros se van grabando en un dispositivo unos detrás de otros consecutivamente, sin dejar huecos y según el orden en que son grabados o guardados. Al final para determinar el fin de fichero se usa la marca EOF (End Of File).

- Directa: Se puede acceder a un registro directamente. Para ello son necesarias 2 cosas:

1º) Que cada registro tenga asociada una clave que lo identifique.

2º) Tiene que existir una función especial llamada función de direccionamiento que sea capaz de convetir la clave a una dirección física real en el dispositivo, que es a la que accederemos para localizar la información buscada.

En la práctica en realidad hay un paso intermedio entre la clave y la dirección física que es la dirección relativa a registro.

Suponiendo un fichero de n registros lógicos numerados de 0 a n-1, este valor asociado al registro es lo que sería su dirección relativa a registro, es decir, que lo que tiene que existir es una función que convierta la clave a dirección relativa a registro, y luego una función que convierta la dirección relativa a registro a dirección física, donde la dirección física tendrá Volumen, Cilindro, Pista y Registro, y esta función puede ser por ejemplo el procedimiento de divisiones sucesivas.

Para realizar este procedimiento, primero hay que decir el número de volúmenes, el número de cilindros, el número de pistas y el número de registros que hay.

Después hay que realizar sucesivas divisiones:

DRR | nº registros

R1 C1 | nº pistas
Registros

R2 C2 | nº cilindros
Pistas

R3 C3
Cilindros Volumen

Volumen : C3
Cilindro: R3
Pista: R2
Registros: R1

Empezando a contar desde 0.

Tenemos un dispositivo con 5 volúmenes, 3 cilindros por volumen, 4 pistas por cilindro y 20 registros por pista. ¿Dónde está el registro 1001?

```
1001 | 20
   1   50 | 4
         2   12 | 3
               0   4
```

Solución → 4021 ó 4022, según empecemos a numerar por 0 o por 1.

El problema está en como convertir la clave de un registro a dirección relativa a registro, si en un soporte como máximo podemos almacenar n registros la dirección relativa a registro irá de 0 a n-1 luego habrá que convertir la clave a uno de estos valores. En cualquier caso suele ocurrir que el rango de claves (conjunto de todas las claves posibles que se pueden dar) es menor que n, ya que en la práctica el espacio reservado es menor que el rango de n porque no usamos todo el espacio.

Según esto puede ocurrir que a diferentes claves les correspondan la misma dirección relativa a registro y por tanto la misma dirección física para almacenarse. A estos registros se les denomina sinónimos, y cuando esto ocurre habrá que almacenar a los registros que tienen la misma posición en un lugar aparte reservado para guardar los sinónimos.

El área reservada se puede colocar en varios sitios dentro de un dispositivo.

Por ejemplo si el rango de claves va de 0 a 999, y la clave es 11919943 una forma de almacenarla, sería cogiendo los 3 últimos números 943.

En cualquier caso la función que convierta una clave a dirección relativa a registro será tal que produzca el menor número de colisiones de sinónimos posibles.

- Secuencial indexada: En esta organización también se puede acceder a un registro directamente, pero ahora lo haré mediante investigación jerárquica de índices.

Para que se pueda aplicar esta organización, obligatoriamente los registros tienen que tener asociados una clave.

En un archivo con esta organización se distinguen 3 áreas:

- **Area de datos**
- **Area de índices**
- **Area de excedentes**

Cuando se crea el fichero la información se va guardando en el área de datos y al mismo tiempo se van creando índices en el área de índices para poder luego localizar esos datos. Después de la creación del fichero la información se almacena en el área de excedentes y se va actualizando también el área de índices.

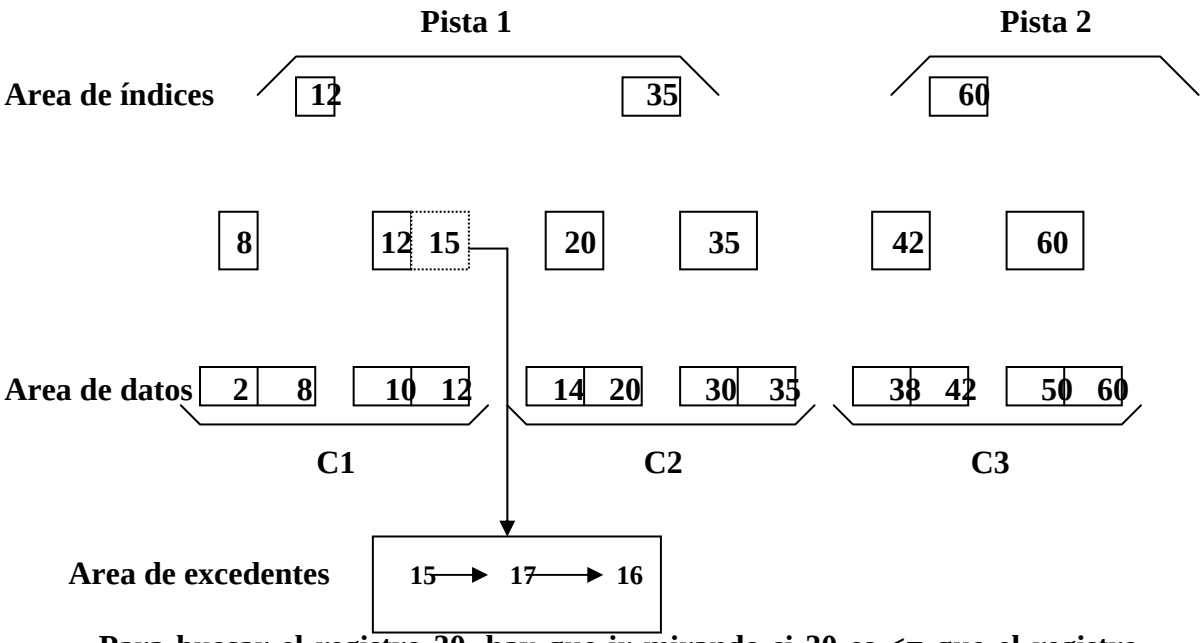
<u>Tipos de índices:</u>

- **Indice de pista: Es el de más bajo nivel y es obligatorio. Hay un índice de pista por cada pista del cilindro, y este índice contiene 2 entradas o valores. El primero determina cual es el valor de la clave más alta**

almacenada en esa pista, y el segundo determina cual es la clave del primer registro de excedente de esa pista.

- **Indice de cilindro: Es el de siguiente nivel y también es obligatorio. Tiene una entrada por cada cilindro y en esa entrada contendrá la clave más alta de los registros almacenados en ese cilindro.**
- **Indice maestro: No es obligatorio. Solo se usa cuando el índice de cilindro ocupa más de 4 pistas. Tiene una entrada por cada pista del índice de cilindros y contendrá la clave más alta de los registros a los que referencian todos los cilindros de esa pista.**

Ej: 2 pistas / cilindro y 2 registros / pista



Para buscar el registro 30, hay que ir mirando si 30 es <= que el registro con el que lo comparamos.

Si llega el 15, lo metemos en el área de excedentes porque no cabe.

6. OPERACIONES CON FICHEROS:

1. Creación o carga de un fichero:

Consiste en meter los datos por primera vez en un fichero que no existe. Para ello hay que determinar el espacio que hay que reservar para el fichero y el método de acceso.

2. Reorganización de un fichero:

Consiste en crear un fichero nuevo a partir de uno que ya existe. Conviene reorganizarlo cuando ya resulta ineficiente su uso porque hay demasiadas colisiones de sinónimos o muchos registros que ya no existen.

3. Clasificación de un fichero:

Consiste en ordenar los registros por uno o varios campos. En los sistemas grandes, el propio sistema operativo ya soporta operaciones que realizan la organización (SORT).

4. Destrucción de un fichero:

Destrucción de un fichero: Significa eliminar toda la información que contiene el fichero.

5. Reunión o fusión de un fichero:

Crear un fichero a partir de varios.

6. Rotura o estallido de un fichero:

A partir de un fichero crear varios.

7. Gestión de un fichero:

Para trabajar con un fichero lo primero que tengo que hacer es crearlo. Para crear un fichero tengo que dar un nombre que es con el que le va a guardar en el dispositivo. Pero para manejar ese fichero dentro de un programa tendré que asignarle una variable de tipo de fichero que luego tendré que asociar con el nombre físico del fichero para poder trabajar con él.

A parte de esto, también puedo dar el tamaño del fichero, su organización y el tamaño de su bloque o registro físico, aunque estos 3 últimos parámetros, solo será necesario darlos para sistemas grandes.

Antes de crear el fichero tengo que definir la estructura de su registro para luego al definir la variable que va a referenciar el fichero determinar que fichero va a contener ese tipo de registro.

Un fichero se crea cuando realizamos la operación de apertura del mismo, aunque hay lenguajes en los que hay que especificar 2 cosas (creación y apertura), y en otros si al abrirlo existe se crea.

Al crear un fichero si ya existe, va a depender del lenguaje, en unos da error y en otros se machaca lo que ya existe.

En la operación de apertura si ya existe de fichero, depende de la forma de apertura que ese fichero que ya existía sea machacado o permanezca. De cualquier forma en la operación de abrir tenemos que indicar la variable de tipo fichero que nos va a servir para trabajar con ese fichero en el programa, a continuación y entre " " el nombre real de ese fichero en el dispositivo, y finalmente el modo de apertura del fichero.

Abrir (<nom_fich>,"nom_real",<modo_apert>)

Los modos de apertura, dependen del lenguaje de programación y del modo de organización. Se distinguen 3 modos de apertura:

- **Modo de entrada o lectura.**
- **Modo de salida o escritura.**
- **Modo de entrada/salida o lectura/escritura.**

APPEND: Existe en algunos lenguajes.

Para declarar una variable de tipo fichero, hay que usar una variable de tipo fichero que es: Fichero de <tipo_reg>.

```
Tipo alumno: registro
                Cod_alum: entero
                Direccion: cadena[30]
                Nota: real
        Fin registro
Var f: fichero de alumno

Abrir (F,"c:\estudiantes.dat",<modo>)
```

- **Modo de entrada: Ese fichero lo voy a usar para leer información de él pero no lo voy a modificar. Para que un fichero se abra en este modo previamente tiene que existir y si no es así, nos dará un error.**
- **Modo de escritura: Utilizo un fichero para escribir información en él. Al abrirlo no hace falta que exista, si ya existiese, depende del lenguaje, pero normalmente se machaca lo que hay (lo que vamos a hacer).**

- **Modo de entrada/salida: Voy a hacer operaciones de consulta y actualización de la información.**
- **Modo APPEND: En pseudocódigo no lo vamos a usar. Append significa añadir, y se utiliza para escribir en un fichero pero añadiendo la información al final del mismo.**

En algunos lenguajes también existe la operación RESET, que lo que me hace es posicionarme al principio del fichero.

Reset (<var_fich>)

En todo momento, existirá un puntero que me indica en que posición del fichero estoy situado. Al abrir un fichero normalmente sea cual sea el modo de apertura me voy a quedar situado al comienzo del fichero, salvo que use el modo de apertura Append, qu me posicionaría al final del fichero. A medida que voy haciendo operaciones en el fichero, este valor se va actualizando.

Cerrar un fichero: Quiere decir deshacer la conexión entre la variable del programa que estoy usando para manejar el fichero y el fichero real que está en el dispositivo.

Cerrar (<var_fich>)

Siempre que se acabe de trabajar con un fichero o que queramos cambiar el modo de trabajar con ese fichero, primero lo cerraremos y si queremos volver a trabajar con él, lo volvemos a abrir.

NUNCA HAY QUE DEJAR UN FICHERO ABIERTO.

En algunos lenguajes existe la operación borrar un fichero y lo que hace es eliminar físicamente el fichero.

Aparte de todo esto también existen otras relacionadas con su actualización, que son las altas, las bajas y las modificaciones.

Las consultas son una operación de mantenimiento del fichero.

<u>Alta:</u>

Consiste en añadir un registro al fichero. Si en el fichero los registros tienen algún campo clave, lo primero que hay que comprobar al dar una alta es que no exista ya otro registro con la misma clave.

<u>Baja:</u>

Consiste en eliminar un registro del fichero. La baja puede ser física o lógica.

- **Física: Cuando elimino físicamente el registro del fichero.**
- **Lógica: Cuando al registro lo marcó de una manera especial para indicar que ya no es válido pero no lo elimino físicamente. A rasgos del tratamiento de la información es como si ya no existiera.**

En ficheros secuenciales no se permite la modificación de la información tal que para dar de baja a un registro de un fichero, lo que hay que hacer es crear otro fichero con todos los registros menos el que queremos eliminar.

En el resto de las organizaciones normalmente se da la baja lógica (con una marca) y cuando tengo muchos registros marcados como baja lógica reorganizo el fichero (baja física).

Si los registros contienen clave para dar de baja a un registro tendré que dar su clave mirar lo primero si existe y solo en ese caso lo podré borrar.

<u>Reglas importantisimas:</u>

- **Para dar la alta a un registro con clave, primero hay que ver si no hay ningún registro con esa clave.**
- **Para dar la baja a un registro:**
  1. **Hay que buscar si el registro existe.**
  2. **Mostrar la información por pantalla y pedir la confirmación del usuario.**

Modificación:

Modificar un registro consiste en cambiar la información que contiene, es decir, cambiar el valor de uno o más de sus campos.

En el caso de la organización secuencial como no se pueden modificar directamente, lo que hay que hacer es a partir de la información que ya existe crear otro fichero con todos los registros menos el que quiero modificar, que lo grabaré con la información que quiera.

Algunos lenguajes que soportan la organización secuencial sí permiten hacer la modificación del registro en el mismo fichero (como el C).

Cuando se modifican registros de un fichero con campos clave en el registro, hay que tener en cuenta que lo que jamas se puede modificar es el valor del campo clave, porque hacer eso es como hacer un alta.

Consultas:

Las consultas lo único que hace es leer información del fichero y mostrarme la información.

Para hacer una consulta hay que dar 2 cosas:

- **Criterio con el que consulto: Contendrá el campo o campos por los que quiero consultar y el valor que tienen que tener esos campos.**
- **Información que quiero mostrar: A la hora de mostrar la información, decir que campos quiero mostrar.**

7. TRATAMIENTO DE FICHEROS SECUENCIALES:

Operaciones básicas:

1. **Definir la estructura del fichero.**
2. **Apertura del fichero.**
3. **Tratamiento de los registros.**
   - **Escribir (<var_fich>,<var_reg>)**
   - **Leer (<var_fich>,<var_reg>)**

Los ficheros secuenciales no se pueden modificar en una posición concreta, por esa razón cada vez que se quiere dar un alta, baja o modificación de un registro, como no se puede hacer directamente, tendré que usar un fichero auxiliar tal que si es un alta, en el nuevo fichero tendré toda la información del fichero antiguo más el nuevo registro, si es dar de baja, copio en el nuevo fichero todo lo que había en el anterior fichero menos el registro que quiero borrar, y si quiero modificar un registro, copio toda la información y al llegar al registro a modificar, lo copio ya modificado.

Consultas:

Todo tratamiento de ficheros, lo voy a hacer en un bucle, y el final de fichero lo señala la marca EOF.

La primera lectura la voy a hacer fuera del bucle, y las siguientes al final del bucle, y el bucle terminará cuando se lee la marca EOF.

Normalmente, en cualquier lenguaje existe ya implementada una función que dice sí hemos llegado al final del fichero pero que lo que hace es buscar la marca EOF.

```
Abrir (<var_fich>,"nom",entrada/Salida)
Leer (<var_fich>,<var_reg>)
Mientras no eof (<var_fich>)
        <proceso con registro>
        leer (<var_fich>,<var_reg>)
Fin mientras
```

En Pascal la marca de fin de fichero está físicamente en el fichero, mientras que en C, se compara el tamaño del fichero con la del puntero del fichero, y si coinciden quiere decir que es el fin del fichero.

Problemas que podemos encontrar:

1. **Si el fichero está clasificado por 2 campos y voy a hacer un tratamiento de él, normalmente tendré que hacer 2 bucles anidados y aprovechar que estén clasificados por campos.**
2. **Si el fichero está clasificado por un campo, para buscar los ficheros por ese campo, hacemos un solo bucle aprovechando que están clasificados por ese campo.**
3. **Si enfrento 2 ficheros, si uno está clasificado y otro no, leeré en el bucle más externo el que no está clasificado, y en el más interno el que está clasificado, aprovechando que estén ordenados.**
4. **Si ninguno está clasificado, da igual el orden de los bucles, porque en los 2 tendré que buscar hasta el final.**
5. **Si lo que me piden es sacar información a través de un campo que no es el campo de clasificación pero que se sabe el número de valores concretos que puede tomar ese campo, utilizaremos una estructura del tipo array.**

Tenemos un fichero con datos de alumnos clasificados por curso y número de alumno, teniendo en cuenta que los alumnos se empiezan a numerar por 1.

La información por alumno es curso, número, nombre y nota. Determinar cuantos alumnos aprobados hay en cada curso.

```
Tipo alumno : registro
        Curso: entero
        Num_al: entero
        Nom: cadena
        Nota: real
    Fin registro
Algoritmo clase
Var
  F_al: fichero de alumno
  R_al: alumno
  Aprob: entero
  Cur: entero
Inicio
```

```
  Abrir (f_al,"alumnos.dat",entrada)
  Leer (f_al,r_al)
  Mientras no eof (f_al)
    Aprob ← 0
    Cur ← r_al.curso
    Mientras cur = r_al.curso
      Si (r_al.nota >= 5) y (no eof(f_al))
        Entonces aprob ← aprob + 1
      Fin si
      Leer (f_al,r_al)
    Fin mientras
    Escribir "En el curso"cur"han aprobado"aprob
  Fin mientras
  Cerrar (f_al)
Fin
```

Tenemos un fichero de empleados clasificado por el número de empleado y un fichero de bajas no clasificado que contiene los números de los empleados que se han dado de baja. Obtener un tercer fichero que contenga a los empleados que realmente hay en la empresa.

```
Tipo empleado: registro
        Num_emp: entero
        Nombre: cadena
        Sueldo: real
      Fin registro
Algoritmo baja
Var
  F_e_ini,f_e_fin: fichero de empleado
  F_baja: fichero de entero
  R_e_ini,r_e_fin: empleado
  R_b: entero
Inicio
```

```
  Abrir (f_e_ini,"Empleado1.dat",entrada)
  Abrir (f_e_fin,"Empleado2.dat",salida)
  Leer (f_e_ini,r_e_ini)
  Mientras no eof (f_e_ini)
    Abrir (f_b,"Bajas.dat",entrada)
     Leer (f_b,r_b)
     Mientras r_b <> r_e_ini.num_emp) y no eof (f_b)
       Leer (f_b,r_b)
     Fin mientras
     Si eof (f_b)
       Entonces escribir (f_e_fin,r_e_fin)
     Fin si
     Cerrar (f_b)
     Leer (f_e_ini,r_e_ini)
  Fin mientras
  Cerrar (f_e_ini)
  Cerrar (f_e_fin)
Fin
```

Tenemos un fichero de clientes no clasificados y la clave del fichero de clientes es el D.N.I. y un fichero de morosos que contiene el D.N.I. y que está clasificado. Queremos obtener un listado de los clientes no morosos.

```
Leer (f_c,r_c)
Mientras no eof (fc)
  Abrir (fm,"Morosos.dat",entrada)
  Leer (f_m,r_m)
  Mientras r_c.DNI > r_m y no eof (f_m)
    Leer (f_m,r_m)
  Fin mientras
  Si r_c.DNI <> r_m
    Entonces escribir r_c.nombre
  Fin si
  Cerrar (f_m)
  Leer (f_c,r_m)
Fin mientras
```

Tenemos un fichero de ventas que tiene el número de venta, total de venta y número de vendedor, sabemos que en nuestra empresa trabajan 5 vendedores numerados del 1 al 5.

Queremos saber el total vendido por cada vendedor.

```
Tipo venta: registro
        Cod_venta: entero
        Total: real
        Vendedor: entero
      Fin registro
Algoritmo ventas
Var
  F_v: fichero de venta
  R_v: venta
  Vend: array [1..5] de real
  I: entero
Inicio
  Desde i=1 hasta 5
    Vend[i] ← 0
  Fin desde
  Abrir (f_v,"ventas.dat",entrada)
  Leer (f_v,r_v)
  Mientras no eof (f_v)
    Vend[r_v.vendedor] ← vend[r_v.vendedor] + r_v.total
    Leer (f_v,r_v)
  Fin mientras
  Cerrar (f_v)
  Desde i=1 hasta 5
    Escribir "El vendedor"i"ha vendido"vend[i]
  Fin desde
Fin
```

<u>"Acceso directo" en archivos secuenciales:</u>

Existen lenguajes que tienen funciones que simulan el acceso directo a un registro dentro de un fichero secuencial (en Pascal → Seek; en C → fseek), a estas funciones le indicamos el número de registro (en Pascal) o el número de bytes (en C), y nos acceden directamente a esa posición.

Sin embargo es simulado, porque internamente esa función va a hacer una búsqueda secuencial hasta que haya llegado al número de registro o al número de bytes.

<u>Mantenimiento de ficheros con organización directa:</u>

- **Alta: Primero hay que ver que no existe el registro, y si no existe se da de alta. Para ver si existe o no existe, a partir de la clave del registro que quiero dar de alta, calculo con la función de direccionamiento la posición en la que tendría que meter ese registro. Si esa posición está ocupada, no lo doy de alta y si no lo meto allí.**

- **Baja: Para ver si existe, hago igual que antes. Si existe lo marco como baja lógica, y cuando se reorganice el fichero, lo doy como baja física real.**
- **Modificación: Para ver si existe, hago igual que antes. Si existe lo modifico y lo vuelvo a grabar en la misma posición.**
- **Consultas: Tienen sentido las consultas de registros en particular con una determinada clave (a través de la función de direccionamiento), pero no las búsquedas secuenciales.**

Mantenimiento de ficheros con organización secuencial indexada:

Es igual que en organización directa, pero a la hora de localizar un registro, en vez de aplicar una función de direccionamiento a su clave, utilizamos su clave para investigar a través de los índices en donde va a estar situado.

Las altas si las doy cuando el fichero se crea, se graban en el área de datos, y si el fichero ya existía se graban en el área de excedentes.

En las consultas tienen sentido tanto las consultas de un registro en particular como las consultas secuenciales a través de una clave porque van a estar todas seguidas.

No hay que confundir organización secuencial indexada, con ficheros indexados o de índices, que son un tipo especial de ficheros asociados a un fichero secuencial pero independiente de él, que me agilizan las búsquedas en el fichero secuencial en el que están indexados. Suelen utilizarse en bases de datos domésticas.

8. FICHEROS DE TEXTO:

Son un tipo de ficheros especiales, en cuanto a lo que podemos leer de ellos y escribir son cadenas de caracteres.

Sin embargo tienen significados distintos según el lenguaje, y su objetivo es que podemos trabajar con cadenas de caracteres.

F : Fichero de texto

C : Cadena
escribir (F, C)
leer (F, C)

EJERCICIOS: TEMA 8

1. **Tenemos un array con la información de nuestros productos, por cada producto almacenamos su código, descripción, stock actual y stock mínimo.**
   Se trata de obtener otro array que contenga los productos de los que halla que hacer pedidos porque su stock sea inferior al mínimo, tal que al proveedor le tenemos que dar como datos la identificación del producto y la cantidad que pedimos, que coincidirá con el stock mínimo.
   Normalmente trabajamos con 100 productos.

   Tipo producto = registro

```
                    Codigo: entero
                    Descripción: cadena
                    Stock: entero
                    Stock_min: entero
    Fin registro
    Pedido = registro
                Codigo: entero
                Cantidad: entero
    Fin registro
  Lista_producto = array [1..100] de producto
  Lista_pedido = array [1..100] de pedido

Algoritmo almacen
Var
  Prod: lista_producto
  Ped: lista_pedido
  I,j: entero
Inicio
  J ← 1
  Desde i=1 hasta 100
    Si prod[i].stock < prod[i].stock_min
      Entonces ped[j].codigo ← prod[i].codigo
               Ped[j].cantidad ← prod[i].stock_min
               J ← j+1
    Fin si
  Fin desde
Fin
```

2. **Dado un array que contiene la información de los alumnos de una clase de 100 alumnos, y teniendo en cuenta que de cada uno de ellos almacenamos su número de expediente, nombre y nota media. Hallar la media de todos los alumnos de esa clase y dar otra opción que pida el nombre de un alumno y me de su nota si este alumno existe.**

```
Tipo alumno = registro
              Expediente: entero
              Nombre: cadena
              Media: real
  Fin registro
  Lista = array[1..100] de alumno
```

```
Algoritmo notas
Var
  Alum: lista
  Op,i: entero
  Marca: booleano
Inicio
  Presentar (op)
  Mientras (op <>0)
    Según sea op
      1: escribir “La media de la clase es”nota_media (alum)
      2: escribir “Introduce un nombre”
         leer nombre
         marca ← falso
         i ← 1
         repetir
           si comparar (alum[i].nombre,nombre) = verdadero
              entonces marca = verdadero
              sino i ← i+1
           fin si
         hasta (i > 100) o (marca=verdadero)
         si marca = verdadero
           entonces escribir “La nota de”nombre”es”alum[i].media
           sino escribir “El alumno no existe”
         fin si
    fin según sea
    presentar (op)
  fin mientras
fin
```

3. **Tenemos un array con la indicación de cada producto, stock, descripción y fecha. Hallar una opción que nos sirva iterativamente para hacer pedidos de productos y que termina cuando pidamos el producto 0.**
   Por cada pedido se da el identificador de producto que se pide, y la cantidad de producto, y lo que nos dice es si hay cantidad suficiente, respondiendo “Pedido suministrado” y actualizando el stock, y si el producto solicitado no existe o no hay suficiente cantidad, mostrará un mensaje de error explicando la causa.

```
Tipo t_fecha = registro
                 Dia: entero
                 Mes: entero
                 Anno: entero
               Fin registro
```

```
    Producto = registro
                Codigo: entero
                Descripicion: cadena
                Stock: entero
                Fecha: t_fecha
    Fin producto
  Lista = array[1..10] de producto

Algoritmo pedidos
Var
  Prod: lista
  Codigo,cantidad,i: entero
  Marca: booleano
Inicio
  Escribir "Introduce el codigo"
  Leer codigo
  Escribir "Introduce la cantidad"
  Leer cantidad
  Mientras codigo <> 0
    I ← 1
    Marca ← falso
    Repetir
      Si codigo = prod[i].codigo
        Entonces marca ← verdadero
        Sino i ← i+1
      Fin si
    Hasta (marca = verdadero) o (i > 100)
    Si marca = falso
      Entonces escribir "No existe el producto"
      Sino si prod[i].stock < cantidad
            Entonces escribir "No hay cantidad suficiente"
            Sino prod[i].stock ← prod[i].stock – cantidad
                 Escribir "pedido suministrado"
          Fin si
    Fin si
  Fin mientras
Fin
funcion nota_media (a: lista): real
var
  i: entero
  acum: real
inicio
  acum ← 0
  desde i=1 hasta 100
      acum ← acum + a[i].nota
  fin desde
  retorno (acum/100)
fin nota_media

funcion comparacion (c1:cadena,c2:cadena): booleano
```

```
var
  i: entero
inicio
  i ← 1
  mientras (c1[i]=c2[i]) y (c1[i]<>'$') y (c2[i]<>'$')
    i ← i+1
  fin mientras
  si c1[i]=c2[i]
    entonces retorno verdadero
    sino retorno falso
  fin si
fin comparacion

Procedimiento presentar (ent-sal opcion: entero)
Inicio
  Repetir
    Escribir "0. Salir"
    Escribir "1. Hallar nota media"
    Escribir "2. Hallar la nota de un alumno"
    Escribir "Introduce una opción"
    Leer opcion
  Hasta (opcion >=0) y (opcion <=2)
Fin presentar
```

4. **Tenemos un fichero de facturas. En la factura viene el número de cliente al que pertenece la factura, el número de factura, el importe de la factura y un campo pagado que será booleano.**
   El fichero está clasificado pro número de cliente y de factura.
   Diseñar un programa que obtenga por pantalla para cada cliente el total que nos debe, y al final me dé el total de lo que la empresa no ha cobrado.

```
Tipo factura: registro
        Num_cl: entero
        Num_fact: entero
        Importe: real
        Pagado: boolean
    Fin registro
Algoritmo facturas
```

```
Var
  Fich: fichero de factura
  Fact: factura
  Total,deuda: real
  Cliente: entero
Inicio
  Abrir (fich,"facturas.dat",entrada)
  Leer (fich,fact)
  Total ← 0
  Mientas no eof (fich)
    Deuda ← 0
    Cliente ← fact.num_cl
    Mientras (cliente = fact.num_cl) y no eof (fich)
      Si fact.pagado = falso
        Entonces deuda ← deuda + fact.importe
                  Total ← total + fact.importe
      Fin si
      Leer (fich,fact)
    Fin mientras
    Escribir "El cliente"cliente"debe"deuda
  Fin mientras
  Escribir "La empresa no ha cobrado"total
Fin
```

5. **Tenemos un fichero de empleados que contiene el departamento, código de empleado, nombre, categoría y sueldo. Hay 12 categorías numeradas del 0 al 11.**
   Por cada departamento sacar el número total de empleados que tiene y el sueldo medio por departamento. Lo mismo hacerlo por categorías.
   El fichero está clasificado por código de departamento y empleado.

```
Tipo emp: registro
            Cod_dep: entero
            Cod_emp: entero
            Nomb_emp: cadena
            Categoria: entero
            Sueldo: real
      Fin registro
```

```
    Cat: registro
        Sueldo: real
        Emp: entero
    Fin registro
Algoritmo empleados
Var
  F_emp: fichero de emp
  R_emp: emp
  T_dep: real
  N_dep,i,d: entero
  C: array [0..11] de cat
Inicio
  Desde i=0 hasta 11
    C[i].sueldo ← 0
    C[i].empleado ← 0
  Fin desde
  Abrir (f_emp,"empleados.dat",entrada)
  Leer (f_emp,r_emp)
  Mientras no eof (f_emp)
      D ← r_emp.cod_dep
      T_dep ← 0
      N_dep ← 0
      Mientras (d=r_emp.cod_dep) y no eof (f_emp)
        T_dep ← t_dep + r_emp.sueldo
        N_dep ← n_dep + 1
        C[r_emp.categoria].sueldo ← c[r_emp.categoria].sueldo + r_emp.sueldo
        C[r_emp.categoria].emp ← c[r_emp.categoria].emp + 1
        Leer (f_emp,r_emp)
      Fin mientras
      Escribir "El sueldo medio del departamento" d " es" t_dep / n_dep
  Fin mientas
  Desde i = 0 hasta 11
    Escribir "El sueldo de la categoría"i"es"c[i].sueldo/c[i].emp
  Fin desde
  Cerrar (f_emp)
Fin
```

6. **Para controlar los stocks de un almacén tenemos:**
   - **Un fichero "almacén", no clasificado que contiene las existencias de cada producto en almacén, y sabemos que hay un máximo de 750 productos. Cada registro tiene el código de producto(1..750), la cantidad en stock de ese producto y el nombre del producto**
   - **Un fichero "pedidos" que contiene todos los pedidos que se solicitan al almacén. Cada pedido contiene el número de pedido, el código de producto que se pide y la cantidad que se pide. Este fichero está clasificado por número de pedido.**

   Obtener un fichero de stock actualizado a partir de los pedidos, pudiéndose dar:

1. **El producto que se pide existe en el almacén y que la cantidad que se pide sea menor o igual a la que hay. Sólo se actualiza el fichero.**
2. **El producto existe, pero la cantidad que nos piden es mayor que la que hay. Guardaremos la información del pedido en el fichero "denegados" que tiene el código del producto, la cantidad pedida y la cantidad que había en stock.**
3. **El código de producto no existe. Mostramos por pantalla un mensaje que ponga "Código de producto erroneo".**

Cada producto aparece como máximo una vez.

```
Tipo alm: registro
            Prod: entero
            Stock: entero
      Fin registro
      Ped: registro
            N_ped: entero
            Prod: entero
            Cant: entero
      Fin registro
      Den: registro
            Prod: entero
            Stock: entero
            Cant: entero
       Fin registro
Algoritmo almacén
Var
  F_a,f_al: fichero de alum
  F_d: fichero de den
  F_p: fichero de pedidos
  R_a: alm
  R_p: ped
  R_d: den
Inicio
  Abrir (f_p,"pedidos.dat",entrada)
  Abrir (f_n,"almacen_n.dat",salida)
  Abrir (f_d,"denegados.dat",salida)
  Leer (f_p,r_p)
  Mientras no eof (f_p)
    Abrir (f_a,"almacen.dat",entrada)
    Leer (f_a,r_a)
    Mientras (r_p.prod <> r_a.prod) y no eof (f_a)
      Leer (f_a,r_a)
    Fin mientras
    Si r_p.prod = r_a.prod
      Entonces si r_p.cantidad < r_a.stock
                  Entonces r_an.prod ← r_a.prod
                           R_an.stock ← r_a.stock – r_p.cantidad
                           Escribir (f_an,r_an)
                  Sino r_d.prod ← r_a.prod
                       R_d.cant ← r_p.cant
                       R_d.stock ← r_a.stock
                       Escribir (f_d,r_d)
```

```
                Fin si
            Sino escribir "Error. No existe el producto"r_p.producto
        Fin si
        Cerrar (f_a)
        Leer (f_p,r_p)
    Fin mientras
    Cerrar (f_p)
    Cerrar (f_d)
    Cerrar (f_an)
    Actualizar_almacen (f_a,f_an)
Fin

Procedimiento actualizar_almacen (f1: fichero de almacen; f2: fichero de almacen)
Var
    R1,r2: alm
    F3: fichero de alm
Inicio
    Abrir (f1,"almacen.dat",entrada)
    Abrir (f3,"alm.act",salida)
    Leer (f1,r1)
    Mientras no eof (f1)
        Abrir (f2,"almacen_n.dat",entrada)
        Leer (f2,r2)
        Mientras (r1.prod <> r2.prod) y no eof (f2)
            Leer (f2,r2)
        Fin mientras
        Si r1.prod = r2.prod
            Entonces escribir (f3,r2)
            Sino escribir (f3,r1)
        Fin si
        Cerrar (f2)
    Fin mientras
    Cerrar (f1)
    Cerrar (f3)
Fin
```

ORDENACIÓN, BÚSQUEDA E INTERCALACIÓN INTERNA: TEMA 9

1. **Introducción.**
2. **Ordenación:**
    - **Método de la burbuja.**
    - **Método de inserción.**
    - **Método de selección.**

- **Método de Quick Short.**

3. **Búsqueda:**
   - **Búsqueda secuencial.**
   - **Búsqueda binaria.**
   - **Búsqueda por conversión de claves o Hashing.**
4. **Intercalación.**

1. INTRODUCCIÓN:

A la hora de tratar datos, muchas veces nos conviene que estén ordenados. Estos métodos nos pueden resultar eficientes cuando hacemos las búsquedas.

2. ORDENACIÓN:

Consiste en organizar un conjunto de datos en un orden determinado según un criterio.

La ordenación puede ser interna o externa:

- **Interna: La hacemos en memoria con arryas. Es muy rápida.**
- **Externa: La hacemos en dispositivos de almacenamiento externo con ficheros.**

Para determinar lo bueno que es un algoritmo de ordenación hay que ver la complejidad del algoritmo (cantidad de trabajo de ese algoritmo), se mide en el número de operaciones básicas que realiza un algoritmo. La operación básica de un algoritmo es la operación fundamental, que es la comparación.

Método de la burbuja:

La filosofía de este método es ir comparando los elementos del array de 2 en 2 y si no están colocados correctamente intercambiarlos, así hasta que tengamos el array ordenado.

Hay que comparar la posición 1 y la 2 y si no están ordenadas las intercambio. Luego la 2 y la 3 y así sucesivamente hasta que comparo las últimas posiciones.

Con esta primera pasada lograremos que quede ordenado el último elemento del array.

Teóricamente, en cada pasada iremos colocando un elemento, y tendríamos que hacer n –1 pasadas. Si en una pasada no se hacen cambios, el array ya está ordenado.

```
Procedimiento burbuja (datos: array [1..N] de <tipo>)
Var
  Ordenado: booleano
  I, J: entero
  Aux: <tipo>
Inicio
```

```
  Ordenado ← falso
  I ← 1
  Mientras (ordenado = falso) y (i <> n – 1)
    Ordenado ← verdadero
    J ← I
    Desde j = 1 hasta n – 1
      Si datos [j] > datos [j + 1]
        Entonces aux ← datos [j]
                  Datos [j] ← datos [j + 1]
                  Datos [j] ← aux
                  Ordenado ← falso
      Fin si
    Fin desde
    I ← I + 1
  Fin mientras
Fin
```

Método de inserción:

Se supone que se tiene un segmento inicial del array ordenado, y hay que ir aumentando la longitud de segmento hasta que coincide con la longitud del array.

Para ello insertaremos el siguiente elemento en el lugar adecuado dentro del segmento ordenado.

Esto se hace moviendo cada elemento del segmento ordenado a la derecha hasta que se encuentre uno menor o igual al elemento que queremos colocar en el segmento o hasta que no tenemos elementos, y lo coloco en esa posición.

Para arrancar este método se parte de que el segmento ordenado inicial este es la primera posición.

```
Procedimiento insercion (datos: array [1..N] de <tipo>)
Var
  I, J: entero
  Aux: <tipo>
Inicio
  Desde i = 2 hasta N
    Aux ← datos [i]
    J ← i – 1
    Mientras (j > 0) y (aux < datos[j])
      Datos[j + 1] ← datos[j]
      J ← j – 1
    Fin mientras
    Datos [j + 1] ← aux
  Fin desde
Fin
```

Método de la selección:

Se trata de buscar el elemento más pequeño y colocarlo en la primera posición, después el segundo más pequeño y colocarlo en la segunda posición, y así sucesivamente hasta que el array este ordenado.

Para ello vamos a recorrer el array, y por cada elemento buscamos a la derecha de esa posición cual es el más pequeño, y lo intercambio con el elemento que estoy examinando.

```
Procedimiento selecccion (datos: array[1..N] de <tipo>)
Var
  I,j,pos: entero
  Aux: <tipo>
Inicio
  Desde i = 1 hasta N-1
    Aux ← datos[i]
    Pos ← i
    Desde j = i+1 hasta N
      Si datos[j] < aux
        Entonces pos ← j
                 Aux ← datos[j]
      Fin si
    Fin desde
    Datos[pos] ← datos[i]
    Datos[i] ← aux
  Fin desde
Fin
```

<u>Método de ordenación rápida o QuickShort:</u>

Consiste en dividir la lista inicial en otras dos que ordenamos por separado recursivamente.

Para ello, se elige un elemento de la lista al que llamamos pivote, tal que a la derecha del pivote va a quedar lo más grande, y a la izquierda lo más pequeño, es decir, que el pivote quedará colocado en su posición.

```
Procedimiento QuickShort (ini: entero; fin: entero; datos: array[1..N] de <tipo>)
Inicio
  Si ini < fin
    Entonces sublistas (ini,fin,pivote,datos)
             Quickshort (ini,pivote-1,datos)
             Quickshort (pivote+1,fin,datos)
  Fin si
Fin
```

```
Procedimiento sublistas (ini:entero;fin:entero;ent-sal pivote:entero;
                         datos:array[1..N]de <tipo>)
Inicio
  Pivote ← ini
```

```
  Aux ← datos[ini]
  Desde i = pivote+1 hasta fin
    Si datos[i] < aux
      Entonces pivote ← pivote + 1
                Aux2 ← datos[i]
                Datos[i] ← datos[pivote]
                Datos[pivote] ← aux2
    Fin si
  Fin desde
  Datos[ini] ← datos[pivote]
  Datos[pivote] ← aux
Fin
```

3. BÚSQUEDAS:

Hay 2 tipos de búsquedas, internas que se hacen en memoria y externas que se hacen en ficheros. Cuando buscamos en un fichero, normalmente lo hacemos a través de una clave.

Dado un determinado valor, se trata de ver si existe un elemento con ese valor en el array de ficheros donde se busca, tal que se devuelve si está o no.

Existen 3 métodos de búsqueda:

- **Secuencial.**
- **Binaria o dicotónica.**
- **Por transformación de claves o Hashing.**

Búsqueda secuencial:

Se puede aplicar para búsquedas internas y externas, y hay que ir pasando secuencialmente por todos los elementos de la estructura hasta encontrar el elemento o acabar la lista.

```
Procedimiento b_secuencial (datos: array[1..N] de <tipo>; elem: <tipo>)
Var
  I: entero
Inicio
  I ← 1
  Mientras (i <= N) y (datos[i] <> elem)
    I ← I + 1
  Fin mientras
  Si datos[i] = elem
    Entonces escribir "Elemento encontrado en la posición"i
    Sino escribir "Elemento no encontrado"
  Fin si
Fin
```

Búsqueda secuencial con centinela:

Se trata de optimizar en cierto modo la búsqueda secuencial normal, lo que hacemos es añadir al final del array el elemento que quiero buscar por lo que siempre lo encontrare.

Si encuentro el elemento en una posición distinta de N+1 significa que no está en la estructura. La ventaja es que en la condición del mientras no tengo que preguntar si se acaba la estructura, me ahorro una condición, el inconveniente es que tiene que sobrar espacio al final del array.

```
Procedimiento b_sec_centineal (datos: array[1..N+1] de <tipo>; elem: <tipo>)
Var
  I: entero
Inicio
  Datos[n+1] ← elem
  I ← 1
  Mientras datos[i] <> elem
    I ← i+1
  Fin mientras
  Si i <> n+1
    Entonces escribir "Elemento encontrado en la posición"i
    Sino escribir "Elemento no encontrado"
  Fin si
Fin
```

<u>Búsqueda binaria o dicotónica:</u>

Para que se pueda aplicar es que la lista en la que queremos buscar el elemento este previamente ordenada.

Se trata de dividir el espacio de búsqueda en sucesivas mitades hasta encontrar el elemento buscado o hasta que ya no pueda hacer más mitades.

Primero hallamos el índice de la mitad del array y miramos si el elemento coincide con él, sino coincide averiguamos donde debería estar el elemento buscado, si en la lista de la derecha o de la izquierda, y dentro de esa mitad hago lo mismo sucesivamente.

```
Procedimiento b_binaria (datos:array [1..N] de <tipo>; elem:<tipo>; ini:entero;
                         fin: entero)
Var
  mit: entero
Inicio
  mit ← (ini+fin) div 2
  mientras (ini < fin) y (elem <> datos[mit])
    si elem < datos[mit]
      entonces fin ← mit – 1
      sino ini ← mit + 1
    fin si
  fin mientras
  si ini < fin
    entonces escribir "Elemento encontrado en la posición" mit
    sino escribir "Elemento no encontrado"
  fin si
fin
```

<u>Búsqueda por transformación de claves o Hashing:</u>

Es necesario que lo que se busque sea por un determinado campo clave. Se trata de convertir ese campo clave en una dirección real, si estamos en un array, en un posición del array y si estamos en un fichero, en un registro del fichero.

Lo que hace que se convierta la clave en una dirección real es la función de direccionamiento. Existen diferentes tipos de funciones de direccionamiento:

- **La más usada es la función módulo, que consiste en dividir la clave entre el número de elementos máximos de la estructura y coger el resto como dirección real de almacenamiento (el índice si es un array, o una dirección relativa si es un fichero).**
- **Entruncamiento: Es la parte de la clave como índice.**
- **Plegamiento: Dividir la clave en partes iguales de tamaño, que va a ser igual al número de cifras del tamaño del array, luego sumarlas y coger las últimas cifras de la suma.**
- **Mitad del cuadrado: Es el cuadrado de la clave y después coger las cifras centrales.**

El problema de estos casos, es que cuando el rango de claves es mayor que el número de posiciones de la estructura, está el problema de que a diferentes claves les corresponde la misma posición, así que cuando se produce el segundo sinónimo hay que ver donde se manda.

Para evitar esto, tratar que la función de direccionamiento produzca el menor número de colisiones posibles.

Para solucionar el problema de los sinónimos:

- **Dejar tanto espacio como rango de claves. Es ilógico.**
- **Si se trata de un array, que por cada posición dejemos una posición más.**
- **La mejor solución es la técnica de encadenamiento, que consiste en que de cada posición del array salga un puntero a una lista enlazada que enlace a todos los elementos que deberían ir posicionados en esa posición o índice del array.**

4. INTERCALACIÓN:

Consiste en juntar varias listas ordenadas en una sola lista que quede ordenada.

```
Procedimiento fusion (a1: T1; a2: T2; a3: T3)
Var
  I,j,k,l: entero
Inicio
  I ← 1
  J ← 1
  K ← 1
  Mientras (i<=n) y (j<=m)
    Si a1[i] < a2[j]
      Entonces a3[k] ← a1[i]
                I ← I + 1
      Sino a3[k] ← a2[j]
           J ← J + 1
    Fin si
    K ← K + 1
  Fin mientras
  Si i < n
    Entonces desde L=i hasta m
                K ←k +1
                A3[k] ← a2[l]
              Fin desde
    Sino desde L=j hasta n
           K ← k+1
           A3[k] ← a1[l]
         Fin desde
  Fin si
Fin
```

1. **Archivos ordenados.**
2. **Fusión o mezcla de archivos ordenados.**
3. **Partición de archivos.**
4. **Clasificación de archivos.**

1. ARCHIVOS ORDENADOS:

Un archivo se puede ordenar por un campo o un conjunto de campos. La mayor parte de los sistemas disponen de una función SORT para ordenar.

2. FUSIÓN O MEZCLA DE ARCHIVOS ORDENADOS:

Dados 2 archivos A y B que tienen la misma estructura y clasificados por el mismo campo o campos, se trata de obtener otro archivo C que tenga la misma estructura de los 2 anteriores y que también queda clasificado por el mismo criterio.

```
Procedimiento fusion (A: t_fich; B: t_fich; C: t_fich)
Var
  R1, r2: <tipo>
Inicio
  Abrir (C,"fich.sal",salida)
  Abrir (A,"fich1",entrada)
  Abrir (B,"fich2",entrada)
  Leer (A,r1)
  Leer (B,r2)
  Mientras no eof (A) y no eof (B)
    Si r1.info < r2.info
      Entonces escribir (C,r1)
                Leer (A,r1)
      Sino escribir (C,r2)
           Leer (B,r2)
    Fin si
  Fin mientras
  Si eof (A)
    Entonces  leer (B,r2)
                mientras no eof (B)
                  Escribir (C,r2)
                  Leer (B,r2)
                Fin mientras
    Sino leer (A,r1)
          mientras no eof (A)
            Escribir (C,r1)
            Leer (A,r1)
          Fin mientras
  Fin si
  Cerrar (A)
  Cerrar (B)
  Cerrar (C)
Fin
```

3. FUSIÓN DE ARCHIVOS:

Se trata de dividir un archivo en varios. Hay diferentes criterios de partición:

Partición por contenido:

Se reparten los registros entre varios registros según el valor de uno o más campos.

< V1 → F1
= V1 → F2
> V1 → F3

```
Procedimiento part_cont (fich: t_fich; ent-sal fich1, fich2, fich3: t_fich)
Var
  R: <tipo>
Inicio
  Abrir (fich,"F",entrada)
  Abrir (fich1,"F1",salida)
  Abrir (fich2,"F2",salida)
  Abrir (fich3,"F3",salida)
  Leer (fich,r)
  Mientras no eof (fich)
    Si r.info < V1
      Entonces escribir (f1,r)
      Sino si r.info > V1
             Entonces escribir (f3,r)
             Sino escribir (f2,r)
           Fin si
    Fin si
    Leer (fich,r)
  Fin mientras
  Cerrar (fich)
  Cerrar (f1)
  Cerrar (f2)
  Cerrar (f3)
Fin
```

Partición por número fijo de registros o por secuencias con clasificación interna:

Se trata de obtener a partir de un fichero varios, tal que todos tengan un número fijo de registros excepto el último.

Si el fichero inicial tiene X registros y quiero obtener como resultado Y ficheros, cada fichero tendrá X/Y registros.

Se trata de ir leyendo secuencias de X/Y registros del fichero F y se vuelcan a un array, clasificar esas secuencias internamente, y cuando estén clasificadas, las vuelco a otro fichero.

El problema es que se usa arrays para la ordenación interna, y a veces el número de ficheros dependen del tamaño máximo del array.

F: 3 9 5 14 2 30 1
F1: 3 5 9
F2: 2 14 30
F3: 1

Partición por secuencias sin clasificación interna:

Se trata de obtener ficheros a partir de uno dado, todos con el mismo tamaño, pero los ficheros no tienen por que estar clasificados.

Nosotros marcamos el tamaño del bloque donde vaya a clasificar la información. Determino también el número máximo de ficheros que quiero tener.

Leeré secuencias de N registros del fichero inicial y los iré grabando en los ficheros resultantes de la partición.

4. CLASIFICACIÓN DE ARCHIVOS:

Es obligatorio que exista una clave. Si el fichero fuese pequeño se llevan todos los registros a un array y los clasifico, pero esto no es lo más usual, por lo que se usa la clasificación externa.

Clasificación por mezcla directa:

Vamos a usar 2 ficheros auxiliares F1 y F2. El fichero F lo organizamos grabando en F secuencias de registros ordenados cada vez más grandes.

Primero leemos las secuencias de un registro de F y las grabo alternativamente en F1 y F2.

Luego leo las secuencias de un registro, una de F1 y otra de F2, y las grabo ordenadas en F.

Vuelvo a leer en F con el doble de secuencia que antes y los grabo en F1 y F2, y repetimos todas las fases duplicando en cada pasada el tamaño de la secuencia hasta que el tamaño obtenido sea igual que el del fichero. A cada tratamiento de F se le llama pasada, y el número de pasadas máximo, será I, tal que el tratamiento se repetirá hasta que 2 elevado a I >= número de registros.

En cada pasada para clasificar la secuencia que leemos de F1 y F2, utilizamos el método de fusión externa.

F: 3 9 5 14 2 30 1 12 10
F1: 3 / 5 / 2 / 1 / 10
F2: 9 / 14 / 30 / 12

F: 3 9 5 14 2 30 1 12 10
F1: 3 9 / 2 30 / 10
F2: 5 14 /1 12

F: 3 5 9 14 1 2 12 30 10
F1: 3 5 9 14 / 10
F2: 1 2 12 30

F: 1 2 3 5 9 12 14 30 / 10
F1: 1 2 3 5 9 12 14 30
F2: 10

F: 1 2 3 5 9 10 12 14 30

Clasificación por mezclas de secuencias equivalentes:

Se parece al método anterior porque también se intenta coger secuencias de registros más grandes, pero ahora usamos 4 ficheros auxiliares y en vez de empezar leyendo secuencias de un registro, hacemos secuencias de N registros, que la primera vez clasificamos internamente, por lo que el valor de N vendrá limitado por el tamaño del array.

Primero leemos del archivo inicial F secuencias de N registros que clasificamos internamente y que grabamos alternativamente en F1 y F2.

Después leo secuencias de N registros de F1 y F2 alternativamente, y por cada par de secuencias leídas las fusiono y las grabo alternativamente ya ordenadas en F3 y F4.

Las secuencias de 2N registros de F3 y F4 las fusiono y la secuencia 4N obtenida ya ordenada la grabo alternativamente en F1 y F2.

Repito esto hasta que todos los ficheros estén vacíos menos 1, y la información de ese fichero la grabo al fichero inicial.

F: 3 9 5 14 2 30 1 12 10
F1: 3 5 9 / 1 10 12
F2: 2 14 30

F1: Vacio
F2: Vacio
F3: 2 3 5 9 14 30
F4: 1 10 12

F1: 1 2 3 5 9 12 14 30 → Vuelco F1 al fichero inicial F.
F2: Vacio
F3: Vacio
F4: Vacio

ESTRUCTURAS DINÁMICAS LINEALES DE DATOS:

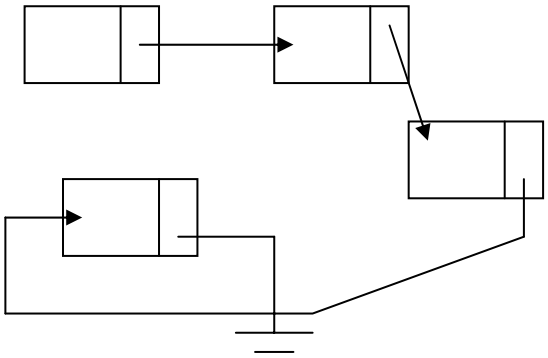
LISTAS ENLAZADAS. PILAS Y COLAS: TEMA 11

1. **Introducción a las estructuras dinámicas de datos.**
2. **Listas.**
3. **Listas enlazadas.**
4. **Procesamiento de listas enlazadas simples.**
5. **Listas circulares con cabecera.**
6. **Listas doblemente enlazadas.**
7. **Pilas.**
8. **Colas.**
9. **Dobles colas o bicolas.**

1. INTRODUCCIÓN A LAS ESTRUCTURAS DINÁMICAS DE DATOS:

Las ventajas de las estructuras dinámicas de datos son:

1. **No hay que definir el tamaño antes de usarla, sino que la voy utilizando según la necesito.**
2. **Los elementos que forman está estructura no están situados en forma contigua en memoria, y esos elementos se relacionan entre sí mediante campos enlace o puntero, y a cada uno de esos elementos lo llamamos nodo de la estructura.**
   Un puntero es un dato cuyo contenido es una dirección de memoria que es en la que está almacenado el dato al que apunta.



Las estructuras dinámicas pueden ser lineales o no lineales según que desde un elemento se pueda acceder solamente a otro o a varios.

Declaración de los punteros en distintos lenguajes:

En C: <tipo> *<var_p>

Int *p

En Pascal: <var_tipo_puntero>: ^<tipo>

P: ^integer

En pseudocódigo: <var_tipo_puntero>: puntero a <tipo>

P: puntero a entero

Acceder a un campo de un registro:

En C: P → nombre

En Pascal: p^.nombre

En pseudocódigo: p → nombre

2. LISTAS:

Una lista es una colección lineal de elementos. Hay 2 formas de almacenarla. Con arrays (usando memoria estática), en donde la relación de los elementos de la lista viene dada porque ocupa posiciones contiguas de memoria.

De forma enlazada con memoria dinámica. Se conoce como listas enlazadas, y la relación entre los elementos de la lista se mantiene mediante campos de enlace o punteros.

3. LISTAS ENLAZADAS:

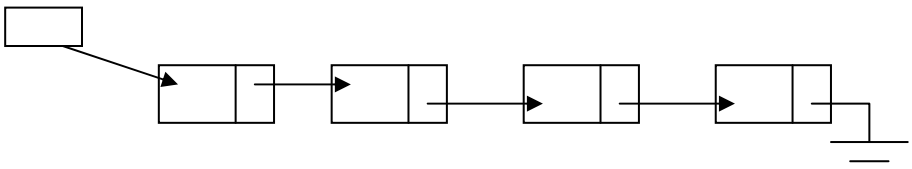
Es una colección lineal de elementos llamados NODOS, donde el orden entre los nodos se establece mediante punteros, y por ser simple, además desde un nodo solo puedo acceder al siguiente directamente.

Una lista enlazada tiene 2 partes:

- **Los nodos que forman la lista. Cada nodo va a tener dos campos. Uno de información que será del tipo de los elementos que contiene la lista y un campo de enlace que es de tipo puntero, y que contendrá la dirección de memoria en la que está almacenado el siguiente nodo.**
  Al primer campo de información lo llamamos INFO, y el nodo lo llamamos SIG.
- **La segunda parte de la lista será una variable de tipo puntero a los nodos de la lista que contiene la dirección del primer nodo de la lista. Por esa razón a esa variable la podemos llamar COMienzo.**

```
Tipo nodo: registro
        Info: <tipo>
        Sig: puntero a nodo
      Fin registro
Var
  Com: puntero a nodo
```

El último nodo de la lista tendrá en su campo de enlace el valor nodo (Null,Nil), que quiere decir que no apunta a ninguna dirección y gráficamente se simboliza con:



```
Tipo nodo: registro
        Info: carácter
        Sig: puntero a nodo
      Fin registro
```

Los nodos de la lista pueden estar en cualquier zona de memoria, no tienen que estar en posiciones consecutivas.

4. PROCESAMIENTO DE LISTAS ENLAZADAS SIMPLES:

Para poder procesar una lista, se necesita la estructura de sus nodos y la variable de comienzo.

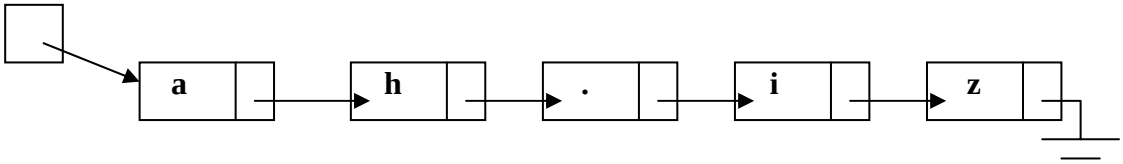
Operaciones:

- **Insertar un elemento.**
- **Borrar un elemento.**
- **Recorrer la lista.**
- **Buscar un elemento.**

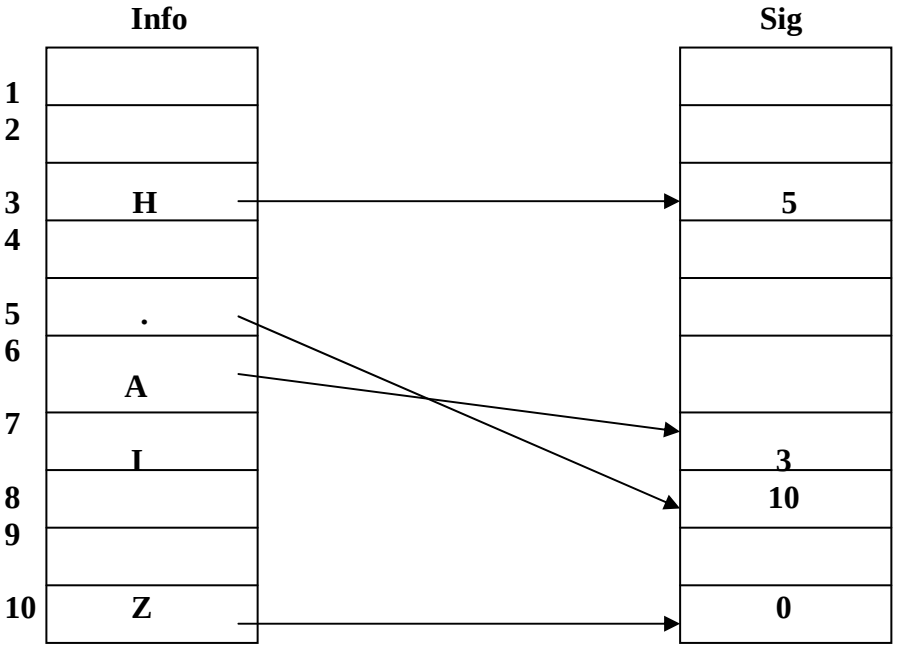
La forma más normal de representar listas enlazadas es usar memoria dinámica. Una lista enlazada también se puede simular usando arrays (memoria estática), pero es poco eficiente.

Para ello utilizaría 2 arrays del mismo tamaño y con la misma numeración de índices, tal que en un array guardaría los campos de información de la lista, y para cada campo de información en la posición correspondiente en el otro array guardaría el índice del siguiente elemento a ese nodo.

Es decir en ese segundo array guardaría los campos enlace mediante valores de índice.



Además de los 2 arrays, tengo que tener una variable que contenga el índice del primer elemento de la lista. El último elemento, tiene en el campo de enlace el valor 0.

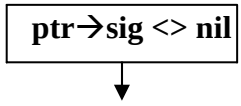


Recorrido e una lista enlazada:

Consiste en paras por todos sus nodos y procesarlos.

```
Tipo nodo: registro
        Info: <tipo>
        Sig: puntero a nodo
      Fin registro
Algoritmo recorrido
Var
  Com: puntero a nodo
Inicio
  Recorrer (com)
Fin

Procedimiento recorrer (prin: puntero a nodo)
Var
  Ptr: puntero a nodo
Inicio
  Si prin = Nil
    Entonces escribir "Lista vacia"
    Sino ptr ← prin
         Mientras ptr <> nil
           {procesar ptr→info}
            ptr ← ptr→sig
         fin mientras
  fin si.
fin
```



posicionarme en el último nodo

Busqueda de un nodo:

Se trata de localizar un nodo que cumpla una determinada condición (por ejemplo que su campo de información valga un valor determinado).

A la hora de hacer las búsquedas tendremos en cuenta si la lista está ordenada o no, y si está ordenada por el criterio de búsqueda tendré que aprovecharlo.

```
Tipo nodo: registro
        Info: <tipo>
        Sig: puntero a nodo
      Fin registro
Algoritmo busqueda
Var
  P,com: puntero a nodo
  Elem: <tipo>
Inicio
  Buscar (com,elem,p)
Fin
Procedimiento buscar (prin: puntero a nodo; e: <tipo>; ent-sal pos: puntero a
nodo)
Inicio
  Pos ← prin
  Mientras (pos→info <> E) y (pos <> NIL)
```

```
    Pos ← pos→sig
  Fin mientras
Fin
```

Búsqueda de un nodo ordenado ascendentemente:

```
Tipo nodo: registro
        Info: <tipo>
        Sig: puntero a nodo
      Fin registro
Algoritmo busqueda
Var
  P,com: puntero a nodo
  Elem: <tipo>
Inicio
  Buscar (com,elem,p)
Fin
Procedimiento buscar (prin: puntero a nodo; e:<tipo>;ent-sal pos: puntero a nodo)
Inicio
  Pos ← prin
  Mientras (e > pos→info) y (pos <> NIL)
    Pos ← pos→sig
  Fin mientras
  Si pos→info <> e
    Entonces pos ← NIL
  Fin si
Fin
```

Tratamiento de la memoria dinámica:

Siempre que se hace una inserción en una estructura dinámica, tenemos que coger de memoria tantos bytes de memoria dinámica como ocupa el nodo de la estructura dinámica, y cuando borramos un elemento, tendremos que liberar la memoria ocupada por ese elemento. Para conseguir este manejo, todos los lenguajes de programación tienen 2 instrucciones que permiten reservar memoria y liberar memoria dinámica, y asignar en el caso de la reserva la memoria a un puntero, y en el caso de la liberación se libera la memoria asignada a un puntero.

Pascal : NEW , DISPOSE
C : MALLOC , FREE
C++ : NEW , DELETE

La instrucción de reserva de memoria llevara un único argumento, que será el tipo de datos para el que se hace la reserva de memoria, y según el tamaño de ese tipo, esa instrucción sabrá el número de bytes que tiene que reservar. Y lo que devuelve es un puntero al comienzo de la zona que se ha reservado. Este valor devuelto será el que tenemos que asignar a un puntero al tipo de datos para el que se hace la reserva.

La función liberar memoria lleva un único argumento que es el tipo puntero, y lo que hace es liberar la memoria asignada a ese puntero, para lo cual es imprescindible que a ese puntero previamente se le haya hecho una asignación de memoria dinámica con al función de reserva.

```
Var p: puntero a <tipo>
P = res_mem (<tipo>)
```

Lib_mem (<var_puntero>)

Nosotros vamos a simular el tratamiento de la memoria dinámica de la siguiente forma: Vamos a suponer que todas las posiciones libres de memoria están enlazadas entre sí a través de una lista enlazada, a la que vamos a llamar DISP (lista de memoria disponible), donde la primera posición de esa lista viene apuntada por la variable DISP.

Cada vez que queremos hacer una inserción en una estructura dinámica, cogeremos la primera posición de DISP, y se la asignaremos al puntero que va a contener a ese elemento.

Y cuando queramos borrar un elemento de una estructura dinámica para indicar que liberamos la memoria utilizada por ese elemento, lo que haremos será insertarlo en el DISP, pero siempre al comienzo.

Según esto, todo borrado en el DISP equivaldrá a una instrucción de reserva de memoria en cualquier lenguaje, y toda inserción en el DISP, equivaldrá a la instrucción de liberar memoria en cualquier lenguaje.

<u>Overflow y Underflow:</u>

La memoria del ordenador tiene un tamaño limitado, por lo que la memoria dinámica también tendrá un tamaño limitado, es decir, que puede ser que se nos agote.

Esta situación, se conoce como OVERFLOW.

Siempre que se haga una instrucción de reserva de memoria, tendremos que comprobar antes si nos queda memoria libre. Eso en cualquier lenguaje ocurre cuando la instrucción de reservar de memoria devuelve el valor NIL.

```
Si DISP = Nil
  Entonces escribir "No hay memoria"
```

Igual se hace para borrar un elemento de una estructura dinámica, para borrarlo, esa estructura tiene que tener al menos un elemento. Si no lo tiene e intentamos borrarlo, se produce un UNDERFLOW.

Por esto, lo primero que tenemos que probar en una estructura dinámica, es preguntar si tiene algo.

```
Si comienzo = Nil
  Entonces escribir "Lista Vacia"
```

Si sólo manejo una estructura dinámica, pregunto por DISP, si trabajo con varias estructuras dinámicas cada una de un tipo, habrá tantas DISP como estructuras dinámicas.

<u>Inserción en una lista enlazada:</u>

Antes de realizar cualquier proceso de inserción, tendré que ver si me queda memoria disponible. Si es posible hacer la inserción, lo primero será reservar la memoria para el elemento que quiero reservar. Para nosotros será coger el primer elemento de DISP.

En segundo lugar, asignaremos los valores correctos al elemento que vamos a insertar, es decir, asignaremos al campo de información del nuevo elemento con la información, y después actualizamos el campo de enlace.

A la hora de asignar un valor al campo de enlace, lo primero será localizar la posición en la que tenemos que localizar el siguiente elemento, y se pueden dar los siguientes casos:

1. **Inserción al comienzo de la lista: Se modifica el valor de la variable comienzo.**
2. **Inserción en cualquier otro lugar de la lista, incluido el final: En este caso, tendremos que conocer también el nodo que va a estar delante de la posición de inserción, ya que a la hora de insertar ese nuevo elemento, va a ocurrir que el nodo de la posición anterior a la de inserción apunte al nuevo elemento, y que el nuevo apunte al nodo de la posición a la que insertamos.**

A la hora de localizar la posición de inserción dependerá de la aplicación en particular.

Como ejemplo de inserción, vamos a dar la inserción en una lista ordenada ascendentemente por el campo de información:

```
Tipo nodo: registro
        Info: <tipo>
        Sig: puntero a nodo
      Fin registro
Procedimiento insercion (ent-sal com: puntero a nodo; elem:<tipo>)
Var
  Lug, Lugp, nuevo: puntero a nodo
Inicio
  Si DISP = Nil
    Entonces escribir "No hay memoria disponible"
    Sino nuevo ← DISP
         DISP ← DISP→SIG
         Nuevo→info ← elem
        Si (com = Nil) o (elem < com→info)
          Entonces nuevo→sig ← Com
                   Com ← nuevo
          Sino lugp ← com
               Lug ← com→sig
               Mientras (elem > lug→info) y (lug <> Nil)
                 Lugp ← lug
                 Lug ← lug→sig
               Fin mientras
               Nuevo→sig ← lug
               Lugp→sig ← nuevo
        Fin si
  Fin si
Fin
```

Borrado en una lista enlazada:

Se trata de borrar un nodo que contenga una determinada información de una lista enlazada simple, aunque el criterio de borrado podría ser otro en lugar del campo de información.

Siempre que se hace un borrado de una estructura dinámica, en general, lo primero es comprobar que la estructura no está vacía, y una vez determinado esto, tendré que localizar el nodo a borrar.

Se pueden dar 2 casos:

- **El nodo a borrar sea el primero.**
- **Que el nodo a borrar ocupe cualquier otra posición.**

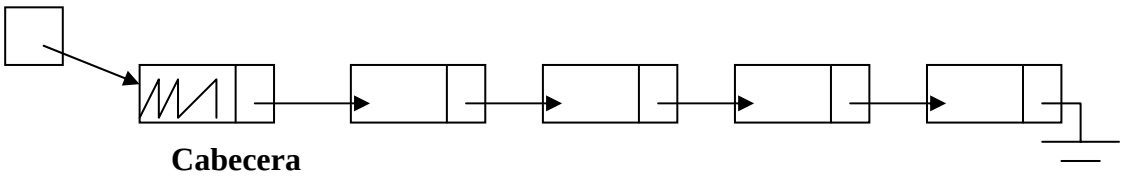
Si es el del comienzo, cambiará la variable de comienzo de la lista porque ahora tendrá que apuntar al siguiente elemento, al de comienzo. Si el elemento ocupa cualquier otra posición, tendré que localizar la posición del elemento y de su predecesor, porque para desenlazarlo de la lista, el predecesor tendrá que apuntar al elemento que le sigue al nodo a borrar.

Finalmente, una vez desenlazado el nodo, habrá que liberar la memoria que ocupaba. Eso en cualquier lenguaje equivale a hacer la operación de liberar memoria, y en pseudocódigo sería insertar el elemento al principio de DISP.

```
Procedimiento borrar (ent-sal com: puntero a nodo; elem: <tipo>)
Var
  Lug, lugp: puntero a nodo
Inicio
  Si com = Nil
    Entonces escribir "Lista vacia"
    Sino lugp ← nil
         Lug ← com
         Mientras (elem > lug→info) y (lug <> Nil)
           Lugp ← lug
           Lug ← lug→sig
         Fin mientras
         Si elem <> lug→info
           Entonces escribir "No existe el nodo a borrar"
           Sino Si lugp = Nil
                  Entonces com ← com→sig
                  Sino lugp→sig ← lug→sig
                Fin si
                Lug→sig ← DISP
                DISP ← lug
         Fin si
  Fin si
Fin
```

5. LISTAS CIRCULARES CON CABECERA:

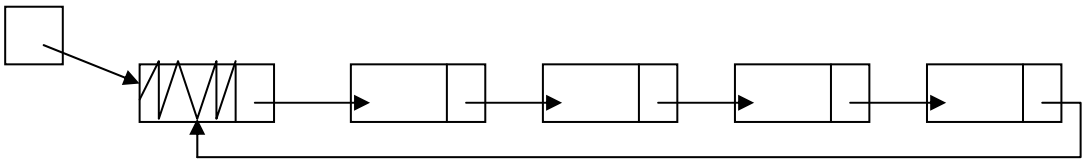
Una lista con cabecera, en general es una lista enlazada simple normal, pero en la que el primer nodo es un nodo especial que no sirve para guardar información válida, sino solamente para marcar que es el primer nodo.



Cabecera

Hay dos tipos de listas enlazadas con cabecera:

- **Listas enlazadas con cabecera y tierra: El primer nodo va a ser la cabecera, y el último apunta a nil. No se utiliza porque perdemos un nodo y no tienen ventajas.**
- **Listas circulares con cabecera: Se caracterizan porque tienen un nodo cabecera, y además el último nodo apunta al nodo cabecera.**



En cualquier lista enlazada con cabecera el primer nodo con información útil es el siguiente a comienzo (Com→sig).

En una lista circular con cabecera, la ventaja que presenta respecto a una normal, es que en las inserciones y borrados no hay que suponer ningún caso especial, pues todos los nodos, incluido el primero útil, tienen un predecesor.

Al recorrer la lista completa, empezaremos buscando la información por el siguiente a comienzo, y sabemos que hemos llegado al final de la lista cuando el siguiente al puntero que recorre la lista sea comienzo.

Búsqueda en una lista circular con cabecera:

```
Procedimiento busqueda (com: puntero a nodo; elem:<tipo>)
Var
  Ptr: puntero a nodo
Inicio
  Ptr ← com→sig
  Mientras (ptr→info <> elem) y (ptr <> com)
    Ptr ← ptr→sig
  Fin mientras
  Si ptr→sig = com
    Entonces escribir "Elemento no encontrado"
```

```
  Fin si
Fin
```

Inserción en una lista circular con cabecera:

```
Procedimiento insercion (com: puntero a nodo; elem: <tipo>)
Var
  Lug,lugp,nuevo: puntero a nodo
Inicio
  Si DISP = Nil
    Entonces escribir "No hay memoria"
    Sino nuevo ← DISP
         DISP ← DISP→sig
         Nuevo→info ←Elem
         Lugp ← com
         Lug ← com→sig
         Mientras (elem > lug→info) y (lug <> com)
           Lugp ← lug
           Lug ← lug→sig
         Fin mientras
         Nuevo→sig ← lug
         Lugp→sig ← nuevo
   Fin si
Fin
```

Borrado en una lista circular con cabecera:

```
Procedimiento borrado (com: puntero a nodo; elem:<tipo>)
Var
  Lug,lugp: puntero a nodo
Inicio
  Si com=com→sig
    Entonces escribir "Lista vacia"
    Sino lugp ← com
         Lug ← com→sig
         Mientras (lug→info <> elem) y (lug <> com)
           Lugp ← lug
           Lug ← lug→sig
         Fin mientras
         Si lug = com
           Entonces escribir "Elemento no encontrado"
           Sino lugp→sig ← lug→sig
                Lug→sig ← DISP
               DISP ← lug
         Fin si
  Fin si
Fin
```

6. LISTAS DOBLEMENTE ENLAZADAS:

Las listas enlazadas simples se caracterizan porque desde un nodo solo puedo acceder al siguiente a ese nodo, por lo que solo puedo acceder al siguiente a ese nodo, por lo que solo puedo recorrer la lista en un sentido, de principio a fin.

Las listas dobles se caracterizan porque desde un nodo podemos acceder directamente tanto al siguiente como al anterior a ese. Es decir, por cada nodo tendremos 2 campos enlace, uno para enlazar un nodo con el siguiente, y otro para enlazar a un nodo con el nodo anterior a él.

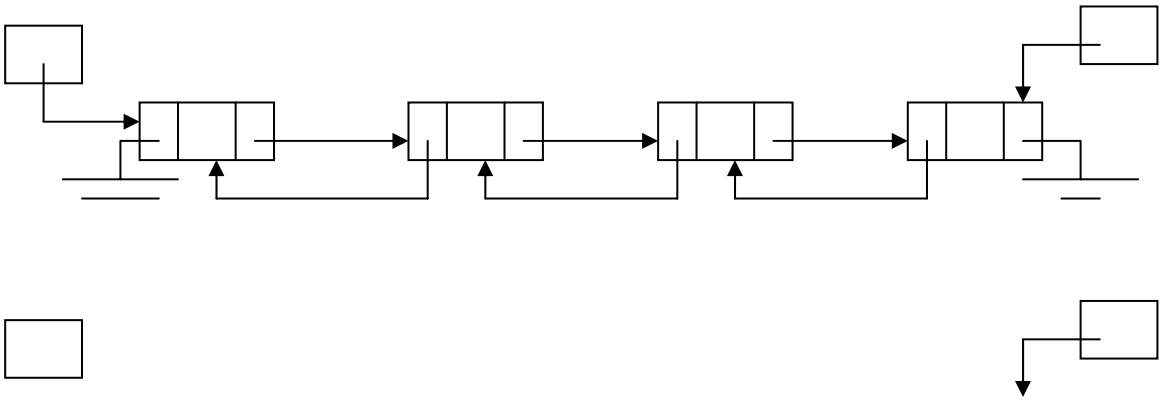
De esta manera, la lista la podemos recorrer en los 2 sentidos, de principio a fin, moviéndonos con el enlace al nodo siguiente o de fin a principio moviéndonos con el enlace al nodo anterior. Para que se pueda hacer este segundo recorrido, además de una variable comienzo para cada lista, que contiene la dirección del primer nodo de la lista, necesitaremos también otra variable especial que denominaremos fin o final, que apunta al último nodo de la lista.

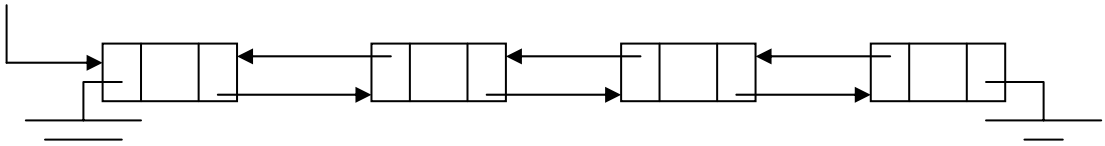
Según esto, la estructura de una lista doblemente enlazada, será la siguiente:

- **Cada lista tendrá 2 variables, comienzo y final, del tipo puntero a nodo doble, que contendrán respectivamente la dirección al primer y último nodo de la lista.**
- **La estructura de cada nodo será:**

  Tipo nodo_doble: registro
  Info: <tipo>
  Sig: puntero a nodo_doble
  Ant: puntero a nodo_doble
  Fin registro

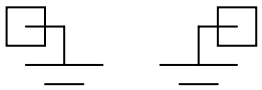
El campo siguiente del último nodo tendrá valor NIL, y el campo anterior del primer nodo también tendrá valor NIL, para indicar que no hay nada antes del primer nodo.



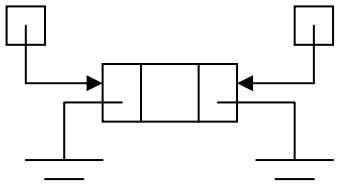


La característica de una lista doblemente enlazada vacía es que el comienzo y el final son iguales y apuntan a NIL.

Si (com = final) y (com = Nil)



Lista doblemente enlazada con un solo elemento:
Si (com = final) y (com <> Nil)



Para implementar la inserción y el borrado de una lista doblemente enlazada, vamos a suponer que la lista de espacio disponible tiene la forma de una lista doblemente enlazada.

<u>Recorrido de una lista doblemente enlazada:</u>

Para pasar por todos los nodos de la lista, podemos hacerlo igual que una lista enlazada simple, solo que ahora la podemos recorrer la lista hacia delante con el campo SIG, y hacia atrás con el campo ANT.

Recorrido hacia delante:

```
Procedimiento recorrido_adelante (com: puntero a nodo doble)
Var
  Ptr: puntero a nodo doble
Inicio
  Ptr ← Comienzo
  Mientras (ptr <> Nil)
    {Procesar PTR.INFO}
    ptr ← ptr→sig
  fin mientras
fin
```

Recorrido hacia atrás:

```
Procedimiento recorrido_adelante (com: puntero a nodo doble)
Var
  Ptr: puntero a nodo doble
Inicio
```

```
  Ptr ← Final
  Mientras (ptr <> Nil)
    {Procesar PTR.INFO}
    ptr ← ptr→ant
  fin mientras
Fin
```

Búsqueda en una lista doblemente enlazada:

Es exactamente igual que en una lista enlazada simple, solo que ahora puedo buscar el elemento buscado desde el comienzo y moviéndome con SIG o desde el final y moviéndome con ANT.

Si tengo una idea de por donde va a estar el elemento, empezare por donde sea mejor.

```
Procedimiento busqueda (com: puntero a doble nodo; final: puntero a doble nodo;
                        elem: <tipo>; ent-sal lug: puntero a doble nodo)
Var
  Ptr, lug: puntero a doble nodo
Inicio
  Lug ← nil
  Ptr ← final
  Mientras (ptr <> nil) y (elemento > ptr→info)
    Ptr ← ptr→ant
  Fin mientras
  Si (ptr <> nil) y (elemento ← ptr→info)
    Entonces lug ← ptr
  Fin si
Fin
```

Inserción en una lista doblemente enlazada:

Lo primero es ver si me queda espacio disponible en memoria dinámica, si me queda, reservo memoria para un nuevo elemento y meto la información que quiero en su campo de información, y lo siguiente es actualizar sus campos de enlace.

Casos particulares:

- **Inserción al comienzo si la lista está vacía.**
- **Inserción al comienzo si la lista no está vacía.**
- **Inserción al final.**
- **Inserción en medio de la lista.**

```
Procedimiento insercion (ent-sal com: puntero a nodo_doble; ent-sal final: puntero
                         a
                             nodo_doble; elem: <tipo>)
var
  lug, nuevo: puntero a nodo_doble
Inicio
  si DISP = Nil
    entonces escribir "No hay memoria"
    sino nuevo ← DISP
         DISP ← DISP→Sig
         Nuevo→info ← elem
         Si com = nil
           Entonces nuevo→sig ← NIL
                    Nuevo→ant ← NIL
                    Com ← nuevo
                    Final ← nuevo
          Sino si elem < com→info
                  Entonces nuevo→sig ← com
                           Nuevo→ant ← nil
                           Com→ant ← nuevo
                           Com ← nuevo
                  Sino lug ← com→sig
                       Mientras (elem > lug→info) y (lug <> Nil)
                         Lug ← lug→sig
                       Fin mientras
                       Si lug <> nil
                         Entonces nuevo→sig ← lug
                                  Nuevo→ant ← lug→ant
                                  Lug→ant→sig ← nuevo
                                  Lug→ant ← nuevo
                         Sino nuevo→sig ← nil
                              Nuevo→ant ← final
                              Final→sig ← nuevo
                              Final ← nuevo
                       Fin si
                Fin si
         Fin si
  Fin si
Fin
```

Borrado en una lista doblemente enlazada:

Primero hay que ver si la lista tiene algún elemento, y si tiene algún elemento, habrá que buscar si el elemento que queremos buscar existe en la lista, y si existe, busco en que posición se encuentra.

Casos particulares:

- **Borrado del primer nodo.**
- **Borrado del último nodo.**
- **Borrado de cualquier nodo.**

Después de todo esto, habrá que liberar la memoria del nodo.

```
Procedimiento borrado (ent-sal com: puntero a nodo_doble; ent-sal final: puntero
                       a
                         nodo_doble; elem: <tipo>)
var
  lug: puntero a nodo_doble
inicio
  si com = Nil
    entonces escribir "Lista vacia"
    sino lug ← com
         mientras (lug→info <> elem) y (lug <> nil)
           lug ← lug→sig
         fin mientras
         si lug = Nil
           entonces escribir "No encontrado"
           sino si lug = com
                  entonces com ← com→sig
                            si com = Nil
                              entonces final ← nil
                              sino com→ant ← nil
                            fin si
                   sino si lug <> final
                          entonces lug→ant→sig ← lug→sig
                                   lug→sig→ant ← lug→ant
                          sino lug→ant→sig ← nil
                               final ← lug→ant
                          fin si
                fin si
                lug→sig ← DISP
                DISP ← lug
         fin si
  fin si
fin
```

7. PILAS:

Una pila es una estructura lineal de datos con la característica especial de que ahora, no podemos hacer las inserciones y las eliminaciones en cualquier lugar, sino que obligatoriamente las tenemos que hacer por un extremo de la lista. Ese extremo lo llamamos cima de la pila.

Esto supone que se procesen los elementos de la pila en orden inverso a su entrada en la estructura, es decir, el primer elemento de la pila que usare será el último que ha entrado (LIFO).

Con una pila podemos hacer dos operaciones básicas, PUSH (meter) y POP (sacar).

Aparte de estas dos funciones se pueden definir otras como la de pila vacía o top, que me dice cual es elemento que está en la cima de la pila pero sin sacarlo.

Se puede implementar con memoria estática (arrays) o con memoria dinámica (listas enlazadas).

Si utilizo un array, tendré que definir cual es el tamaño máximo del array, que será el de la pila, y aparte tendré definida una variable cima que tendrá en cada momento el índice que corresponde al elemento del array que está en la cima de la pila.

Pila = array [1..max_pila] de <info>
Cima: entero

Para señalar que la cima está vacía, cima es igual a 0 porque no tiene que apuntar a ningún elemento.

<u>Implementación de una pila con memoria estática:</u>

```
Algoritmo prin_pila
Const
  Max_pila = <max_pila>
Var
  P: array [1..max_pila] de <info>
  C: entero
  Elem: <info>
Inicio
  C ← 0
  Leer (elem)
  Si pila_vacia  (C) = verdadero
    Entonces escribir “Pila vacia”
    Sino sacar (pila,c,elem)
         Escribir “Elemento sacado”elem
  Fin si
Fin

Funcion pila_vacia (cima: entero): booleano
Inicio
  Si cima = 0
    Entonces retorno verdadero
    Sino retorno falso
  Fin si
Fin

Procedimiento meter (ent-sal pila: array[1..max_pila] de <info>; cima: entero;
                     elem:<info>)
Inicio
 si cima = max_pila
    entonces escribir “Pila llena”
```

```
    sino cima ← cima + 1
         pila[cima] ← elem
  fin si
Fin

Funcion cima_pila (pila: array[1..max_pila] de <info>; cima: entero):<info>
Inicio
  Retorno pila[cima]
Fin
```

```
Procedimiento sacar (ent-sal pila: array[1..max_pila] de <info>; cima: entero;
                     ent-sal e: <info>)
Inicio
  E ← cima_pila (cima,pila)
  Cima ← cima – 1
Fin
```

Implementación de una pila con memoria dinámica:

```
Tipo nodo: registro
       Info: <tipo>
       Sig: puntero a nodo
     Fin registro
Algoritmo manejo_pila
Var
  Cima: puntero a nodo
  Elem: <tipo>
Inicio
  Cima ← nil
  Meter (cima,elem)
  Si pila_vacia (cima)
    Entonces escribir “No hay memoria”
    Sino sacar (cima,elem)
         Escribir “Elemento sacado”elem
  Fin si
Fin

Funcion pila_vacia (cima: puntero a nodo): booleano
Inicio
  Si cima = nil
    Entonces retorno verdadero
    Sino retorno falso
  Fin si
Fin

Procedimiento poner (ent-sal cima: puntero a nodo; elem: <tipo>)
Var
```

```
  Nuevo: puntero a nodo
Inicio
  Si DISP = nil
    Entonces escribir "No hay memoria"
    Sino nuevo ← DISP
         DISP ← DISP→sig
         Nuevo→info ← elem
         Nuevo→sig ← cima
         Cima ← nuevo
  Fin si
Fin
```

```
Funcion cima_pila (cima: puntero a nodo):<tipo>
Inicio
  Retorno cima→info
Fin
```

```
Procedimiento sacar (ent-sal cima: puntero a nodo; ent-sal elem: <tipo>)
Var
  Borrado: puntero a nodo
Inicio
  Elem ← cima_pila (cima)
  Borrado ← cima
  Cima ← cima→sig
  Borrado→sig ← DISP
  DISP ← borrado
Fin
```

Aplicaciones de las pilas:

Un ejemplo típico de uso de pilas, es en las llamadas entre funciones. En este caso, el sistema operativo utiliza una pila en donde va guardando la dirección de retorno de cada una de estas llamadas.

8. COLAS:

Una cola también es una lista lineal de elementos, en la que las inserciones se hacen por un extremo de la lista, y los borrados por otro. Las inserciones se hacen por el final y los borrados por el principio.

Esto significa que es una estructura del tipo FIFO.

Sea cual sea la implementación, siempre tendrán que existir 2 variables, frente y final, que apunten al comienzo y al fin de la cola respectivamente.

Podemos implementar una cola con memoria estática (arrays) y con memoria dinámica (listas enlazadas).

Si implementamos la cola con arrays, tendremos un array llamado cola, que contiene el tipo de información que contiene la cola, y aparte dos valores, frente y final, cuyo valor servirá como el índice que indica que elemento está al frente y al final de la cola.

La cola estará vacía cuando frente y final no apunten a ningún índice del array, y esto será cuando frente y final valgan 0 si el array se empieza a numerar a partir del 1.

Aparte de la función que me dice si la cola está vacía, tendré que implementar funciones de sacar y meter un elemento.

Para insertar un elemento en una cola, habría que incrementar el valor en la variable final, y después introducir en la cola del final el elemento.

Para eliminar un elemento de una cola, bastaría primero ver que elemento está al frente de la cola, y para eliminarlo aumentar el frente en una unidad.

Si solo consideramos esta posibilidad, es decir, manejar el array solo incrementando, no lo estaríamos utilizando eficientemente, porque podríamos llegar al caso en que habiendo posiciones libres al comienzo del array no pudiésemos usarlas cuando el final alcanzase el máximo índice del array.

Para evitar esto, el array que implementa la cola, lo vamos a tratar como un array circular (suponemos que despues de la posición N del array, viene la posición 1). Esto quiere decir que si la cola no está llena, después de final = Max, viene final = 1, y al sacar el elemento de la cola, si no está vacía, después de frente = N, vendrá frente = 1, y si después de sacar el elemento la cola queda vacía, actualizaremos frente y final a 0.

Si frente es igual a final y son distintos de 0 antes de sacar un elemento, quiere decir que ese elemento es el último.

Antes de insertar un elemento, tendré que ver si la cola está llena (frente = 1 y final = Max o frente = final + 1).

<u>Implementación de colas con memoria estática:</u>

```
Tipo datos: array [1..max_cola] de <tipo>
Algoritmo manejo_cola
Var
  Cola: datos
  Frente, final: entero
  Elem, result: <tipo>
Inicio
  Frente ← 0
  Final ← 0
  Meter_cola (frente,final,cola,elem)
  Si cola_vacia (frente,final) = verdadero
    Entonces escribir “Cola vacía”
    Sino sacar_cola (frente,final,cola,result)
         Escribir “Elemento sacado”result
  Fin si
Fin

Funcion cola_vacia (frente: entero; final: entero): boolean
Inicio
  Si (frente = 0) y (final = 0)
    Entonces retorno verdadero
```

```
    Sino retorno falso
  Fin si
Fin
```

```
Procedimiento meter_cola (ent-sal cola:datos; ent-sal frente:entero; ent-sal
final:entero;
                          Elem: <tipo>)
Inicio
  Si ((frente = 1) y (final = Max_cola)) o frente = final + 1
    Entonces escribir "Cola llena"
    Sino si cola_vacia (frente,final) = verdadero
          Entonces frente ← 1
                   Final ← 1
          Sino si final = Max_cola
                 Entonces final ← 1
                 Sino final ← final + 1
               Fin si
        Fin si
        Cola[final] ← elem
  Fin si
Fin
```

```
Procedimiento sacar_cola (ent-sal cola:datos; ent-sal frente:entero; ent-sal
final:entero;
                          Elem: <tipo>)
Inicio
  Elem ← cola[frente]
  Si frente = final
    Entonces frente ← 0
             Final ← 0
    Sino si frente = Max_cola
          Entonces frente ← 1
          Sino frente ← frente + 1
        Fin si
  Fin si
Fin
```

Implementación de colas con memoria dinámica:

```
Funcion cola_vacia (com: puntero a nodo_doble): booleano
Inicio
  Si com = nil
    Entonces retorno verdadero
    Sino retorno falso
  Fin si
Fin
```

```
Procedimiento meter_cola (ent-sal com: puntero a nodo_doble;
                          ent-sal final: puntero a nodo_doble; elem: <tipo>)
var
  nuevo: puntero a nodo_doble
Inicio
  Si DISP = nil
    Entonces escribir "No hay memoria"
    Sino nuevo ← DISP
         DISP ← DISP→sig
         Nuevo→info ← elem
         Nuevo→ant ← final
         Nuevo→sig ← nil
         Final→sig ← nuevo
         Final ← nuevo
         Si cola_vacia (frente) = verdadero
           Entonces frente ← nuevo
         Fin si
  Fin si
Fin
```

```
Procedimiento sacar_cola (ent-sal com: puntero a nodo_doble;
                          ent-sal final: puntero a nodo_doble; result: <tipo>)
var
  ptr: puntero a nodo_doble
Inicio
  Result ← com→info
  Ptr ← com
  Com ← com→sig
  Si com = nil
    Entonces final ← nil
```

Sino com→ant ← nil
Fin si
Ptr→sig ← DISP
DISP ← ptr
Fin

<u>Aplicaciones de las colas:</u>

Las colas se suelen utilizar en los procesos por lotes y en la utilización de recursos del sistema.

Cuando un proceso quiere usar un recurso y otro lo está usando, tendrá que ponerse en la cola, y se irá asignando el recurso según el orden en que se ha pedido.

A veces, la utilización de recursos por parte de los procesos, nos interesa que ciertos procesos tengan mayor prioridad que otros, incluso aunque lleguen más tarde, y para ello utilizaríamos las colas de prioridades.

Cada elemento de la cola, tendrá otro campo que indique su prioridad, tal que a la hora de sacar un elemento de la cola, saco el del frente, y a la hora de insertar un elemento en la cola, tendré en cuenta la prioridad del elemento que quiero insertar, y para ello se inserta en la cola por orden de prioridad, y si hay mas elementos que tienen la misma prioridad que el que queremos insertar, los procesamos según su orden de llegada, es decir, que lo colocamos como el último de los elementos con esa prioridad.

<u>DOBLES COLAS O BICOLAS:</u>

Una bicola es una lista lineal de elementos en la que las inserciones y borrados es pueden hacer por cualquiera de sus extremos.

Va a haber 2 variables, izquierda y derecha, que apuntan a sus extremos.

Hay 2 tipos especiales de bicolas:

- **De entrada restringida: Que permite inserciones solo por un extremo y borrados por los dos.**
- **De salida restringida: Que permite inserciones por cualquier extremo y borrado solo por uno.**

Se pueden implementar con memoria estática o dinámica.

EJERCICIOS: TEMA 11

1. **Diseñar un algoritmo que coloque el primer nodo de una lista enlazada como penúltimo, pero cambiando solo los campos enlace. La lista es una lista enlazada doble.**

```
Procedimiento resultado (ent-sal com: ptr_a nodo; ent-sal final: ptr_a nodo)
Var
  Prin: puntero a nodo
Inicio
  Si (com = final) o (com→sig = final)
    Entonces escribir "No se puede hacer el cambio"
    Sino prin ← com
         Com→sig ← final
         Com→ant ← final→ant
         Final→ant→sig ← com
         Final→ant ← com
         Com ← prin→sig
         Com→ant ← nil
  Fin si
Fin
```

2. **Dada una cadena de caracteres C1, en un array de longitud máxima N, y con el fin de cadena '$', y dada otra cadena C2 almacenada en una lista enlazada simple, que cada nodo de la lista contiene una letra. Se trata de que pasandole a un subprograma 2 cadenas, determinar cuantas veces aparece C1 en C2.**

```
Funcion compara (c1: cadena; c2: ptr_a nodo): entero
Var
  Ini,p: puntero a nodo
  I,res: entero
```

```
Inicio
  Res ← 0
  P ← C2
  Mientras (p<> nil)
    Si (p→info = c1[i])
      Entonces ini ← P
                I ← 1
                Mientras (p→info = c1[i]) y (c1[i] <> '$') y (p <> nil)
                  I ← i + 1
                  P ← p→sig
                Fin mientras
                Si c1[i] = '$'
                  Entonces res ← res + 1
                  Sino p ← ini→sig
                Fin si
      Sino p ← p→sing
    Fin si
  Fin mientras
Fin
```

3. **Dada una lista doblemente enlazada que contiene en cada nodo un dígito decimal (0..9), determinar el valor que se obtiene si consideramos la información de cada nodo de la lista como los términos de un polinomio, donde el término de menor grado es el último nodo y el de máximo grado es el del comienzo, evaluando el polinomio para un valor que se le pasa como parámetro.**

```
Funcion pot (base: entero; exp: entero): real
Var
  I: entero
  Acum: real
Inicio
  Acum ← 1
  Desde i = 1 hasta exp
    Acum ← acum * base
  Fin desde
  Retorno acum
Fin

Funcion polinomio (fin: ptr_a nodo_doble; x: entero): real
Var
  Res: real
  P: puntero a nodo_doble
Inicio
  Exp ← 0
  Res ← 0
  P ← final
  Mientras p <> nil
    Res ← res + (c→info * pot (x,exp)
    P ← p→ant
```

```
    Exp ← exp + 1
  Fin mientras
  Retorno res
Fin
```

4. **Tenemos almacenado en un fichero de productos el stock de cada uno de ellos en el almacén. Cada registro tiene identificador, nombre, stock actual y stock mínimo. Iterativamente, hasta que se introduzca un identificador de producto igual a 0, se irán haciendo pedidos o entregas pidiéndonos el identificador, la cantidad y si es pedido o entrega, y con ellos iremos actualizando la cantidad de productos en almacén. Al final quiero tener el fichero actualizado. Para ello, utilizaremos una lista enlazada como estructura auxiliar.**

```
Tipo producto: registro
        Indentificador: entero
        Nombre: cadena
        Stock_act: entero
        Stock_min: entero
      Fin registro
 Nodo: registro
        Info: producto
        Sig: puntero a nodo
 Fin registro
Var
  Com, ptr: puntero a nodo
  Reg: producto
  Fich: fichero de producto
  Id,cant: entero
  Car: carácter
Inicio
  Com ← nil
  Abrir (fich,"datos.dat",entrada)
  Leer (fich,reg)
  Mientras no eof (fich)
```

```
    Insertar (com,reg)
    Leer (fich,reg)
  Fin mientras
  Cerrar (fich)
  Escribir "Identificador"
  Leer (id)
  Mientras id <> 0
    Escribir "Cantidad"
    Leer cant
    Escribir "Pedido/Entrega"
    Leer car
    Modifica (com,id,cant,car)
    Escribir "Identificador"
    Leer id
  Fin mientras
  Abrir (fich,"datos.dat",salida)
  Ptr ← com

  Mientras com <> nil
    Escribir (fich,com→info)
    Com ← com→sig
    Ptr→sig ← DISP
    DISP ← ptr
    Ptr ← com
  Fin mientras
  Cerrar (fich)
Fin

Procedimiento insertar (ent-sal com: puntero a nodo; elem: producto)
Var
  Lug,lugp,nuevo: puntero a nodo
Inicio
  Si DISP = Nil
    Entonces escribir "No hay memoria"
    Sino nuevo ← DISP
         DISP ← DISP→Sig
         Nuevo→info ← elem
         Si com = nil
           Entonces nuevo→sig ← com
                    Com ← nuevo
           Sino lugp ← com
                Lug ← com→sig
                Mientras (elem.identificador > lug→info) y (lug <> nil)
                  Lugp ← lug
                  Lug ← lug→sig
                Fin mientras
```

```
                Nuevo→sig ← lug
                Lugp→sig ← nuevo
          Fin si
    Fin si
Fin
```

```
Procedimiento modifica (prim: ptr_a nodo; d: entero; c: entero; caract: carácter)
Var
  Ptr: puntero a nodo
Inicio
  Si prin = nil
    Entonces escribir "Lista vacia"
    Sino ptr ← prin
         Mientras (ptr <> nil) y (id <> ptr→info.identificador)
           Ptr ← ptr→sig
         Fin mientras
         Si ptr = nil
           Entonces escribir "El producto no existe"
           Sino si (caract = 'P') o (caract = 'p')
                    Entonces ptr→info.cantidad ← ptr→info.cantidad + c
                    Sino si ptr→info.cantidad >= c
                             Entonces ptr→info.cantidad ← ptr→info.cantidad – c
                             Sino escribir "No hay cantidad suficiente"
                           Fin si
                  Fin si
         Fin si
  Fin si
Fin
```

5. Dadas 2 listas enlazadas simples, L1 y L2, ordenadas en ascendente, actualizar las 2 listas de modo que L2 quede vacía y L1 contenga a todos los elementos de L2, menos a los repetidos. No borrar elementos, hay que enlazarlos.

```
Procedimiento juntar (ent-sal com1: ptr_a nodo; ent-sal com2: ptr_a nodo)
Var
  P1,a1, aux, borrado: puntero a nodo
Inicio
  Si com2 <> nil
    Entonces si com1 = nil
                Entonces com1 ← com2
                Sino si (com2→info < com1→info) y (com2 <> nil)
                        Entonces aux ← com2
                                  Com2 ← com2→sig
                                  Aux→sig ← com1
                                  Com1 ← aux
                        Sino a1 ← com1
                             P1 ← com1→sig
                             Mientras (p1 <> nil) y (com2 <> nil)
                               Si (com2→info < p1→info)
                                 Entonces a1 ← p1
                                           P1 ← p1→sig
                                 Sino si (com2→info = p1→info)
                                         Entonces borrado ← com2
                                                   Com2 ← com2→sig
                                                   Borrado→sig ← DISP
                                                   DISP ← borrado
                                         Sino aux ← com2
                                              A1→sig ← com2
```

```
                                        Com2→sig ← P1
                                        Com2 ← aux→sig
                                        A1 ← a1→sig
                                    Fin si
                                Fin si
                            Fin mientas
                            Si com2 <> nil
                              Entonces a1→sig ← com2
                                       Com2 ← nil
                            Fin si
                        Fin si
                    Fin si
  Fin si
Fin
```

6. Implementar una cola usando listas enlazadas simples:

```
Procedimiento meter (ent-sal com: ptr_a nodo; ent-sal final: ptr_a nodo; e: <tipo>)
Var
  Nuevo: ptr_a nodo
Inicio
  Si DISP = nil
    Entonces escribir "No hay memoria"
    Sino nuevo ← DISP
         DISP ← DISP→sig
         Nuevo→sig ← Nil
         Si com = nil
           Entonces com ← nuevo
                    Final ← nuevo
           Sino final→sig ← nuevo
                 Final ← nuevo
         Fin si
  Fin si
Fin

Procedimiento sacar (ent-sal com:ptr_a nodo; ent-sal final:ptr_a nodo; ent-sal e:<tipo>)
Var
  Borrado: puntero a nodo
Inicio
  E ← com→info
  Borrado ← com
  Com ← com→sig
  Si com = nil
```

```
    Entonces final ← nil
  Fin si
  Borrado→sig ← DISP
  DISP ← borrado
Fin
```

7. Implementar una cola de prioridades con listas enlazadas simples:

```
Procedimiento meter (ent-sal com: ptr_a nodo; e: <tipo>; p: entero)
Var
  Lugp, lug, nuevo: puntero a nodo
Inicio
  Si DISP = nil
    Entonces escribir "No hay memoria"
    Sino nuevo ← DISP
         DISP ← DISP→sig
         Nuevo→info ← e
         Nuevo→prioridad ← p
         Lugp ← nil
         Lug ← com
         Mientras (lug <> nil) y (p >= lug→prior)
           Lugp ← lug
           Lug ← lug→sig
         Fin mientras
         Si lugp = nil
           Entonces nuevo→sig ← com
                    Com ← nuevo
           Sino lugp→sig ← nuevo
                Nuevo→sig ← lug
         Fin si
  Fin si
Fin

Procedimiento sacar (ent-sal com:ptr_a nodo; ent-sal e:<tipo>)
```

```
Var
  Borrado: puntero a nodo
Inicio
  E ← com→info
  Borrado ← com
  Com ← com→sig
  Borrado→sig ← DISP
  DISP ← borrado
Fin
```

8. Implementar una cola de prioridades como lista de listas:

```
Tipo nodo_prior: registro
        Prior: entero
        Prinp: ptr_a nodo_info
        Sig: ptr_a nodo_pr
    Fin registro
    Nodo_info: registro
        Info: <tipo>
        Enl: ptr_a nodo_info
    Fin registro

Procedimiento meter (ent-sal com:ptr_a nodo_prior; elem:<tipo>; p: entero)
Var
  Nuevoi, lugi: puntero a nodo_info
  Nuevop, ptr, ptra: puntero a nodo_prior
Inicio
  Si (DISPP = nil) y (DISPI = nil)
    Entonces escribir "No hay memoria"
    Sino nuevoi ← DISPI
        DISPI ← DISPI→enl
        Nuevoi→info ← elem
        Nuevo→sig ← NIL
        Ptra ← nil
        Ptr ← com
        Mientras (ptr <> nil) y (p > ptr→prior)
          Ptra ← ptr
          Ptr ← ptr→sig
```

```
        Fin mientras
        Si (p = ptr→prior)
          Entonces lugi ← ptr→prinp
                   Mientras (lugi→sig <> nil)
                     Lugi ← lugi→enl
                   Fin mientas
                    Lugi→sig ← nuevo
          Sino nuevop ← DISPP
               DISPP ← DISPP→sig
               Nuevop→prior ← p
               Nuevop→sig ← ptr
               Nuevop→prinp ← nuevoi
               Si (ptra = nil)
                 Entonces com ← nuevop
                 Sino ptra→sig ← nuevop
               Fin si
        Fin si
  Fin si
Fin
```

```
Procedimiento sacar (ent-sal com: ptr_a nodo_prior; ent-sal e:<tipo>)
Var
  Bori: ptr_a nodo_info
  Borp: ptr_a nodo_prior
Inicio
  E ← com→prinp→info
  Bori ← com→prinp
  Com→prinp ← com→prinp→enl
  Bori→enl ← DISPI
  Si com→prinp = nil
    Entonces borp ← com
              Com ← com→sig
              Borp→sig ← DISPP
              DISPP ← borp
  Fin si
Fin
```

ÁRBOLES Y GRAFOS:
ESTRUCTURAS DE DATOS NO LINEALES:
TEMA 12

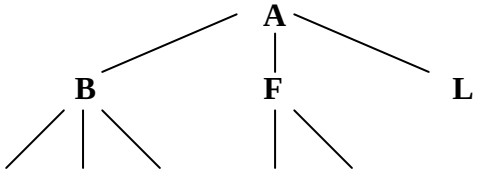
1. **Árboles.**
2. **Árboles binarios.**
3. **Árboles binarios de búsqueda.**
4. **Aplicaciones de los árboles binarios.**
5. **Grafos.**
6. **Operaciones con grafos.**
7. **Aplicaciones de los grafos.**

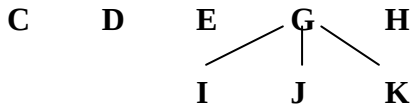
1. ÁRBOLES:

Los árboles están dentro de las estructuras de datos no lineales, que consisten en que desde un nodo tengo acceso a varios elementos.

Un árbol es un conjunto finito de elementos llamados nodos, que guardan entre sí una relación jerárquica tal que siempre va a existir un nodo raíz diferenciado del resto, y los restantes parten de él, formando conjuntos disjuntos cada uno de los cuales es a su vez otro árbol. A cada uno de esos árboles se les llama subárboles del raíz.

Un árbol vacío es aquel que no tiene ningún nodo (Raiz = NIL).





Terminología de los árboles:

- **Raíz: Nodo esencial de que parten todos los demás. Es el único que no tiene antecesor.**
- **Nodo: Cada uno de los elementos del árbol.**
- **Hoja o nodo terminal: Nodo que no tiene ningún otro subárbol debajo de él.**
- **Hijo: Cada nodo que no es hoja tiene debajo de él a otros subárboles.**
- **Padre: Todo nodo excepto el raíz, que tiene asociado un predecesor del que desciende.**
- **Hermano: Relación entre los nodos hijos del mismo padre.**
- **Nodo interno: El que tiene algún hijo.**
- **Nivel: Cada nodo de un árbol tiene asociado un número de nivel. El del raíz es el 0, y a partir de ahí se va aumentando de 1 en 1.**
- **Camino entre A y B: Sucesión de enlaces o nodos por los que hay que pasar para llegar de A a B.**
- **Rama: Camino que termina en una hoja.**
- **Profundidad de un árbol: Es el número máximo de nodos de la rama más larga del árbol. Equivale al número máximo de niveles + 1.**
- **Peso: Número de hojas que tiene un árbol.**
- **Bosque: Colección de 2 o más árboles.**

2. ÁRBOLES BINARIOS:

Son un conjunto finito de elementos llamados nodos que contienen un nodo raíz y donde cada nodo puede tener 0,1, ó 2 hijos. A cada subarbol se le denomina subarbol izquierdo y subarbol derecho.

Terminología:

- **Dos árboles binarios son similares si tienen los nodos colocados igual.**
- **Árboles binarios equivalentes o copias: Tienen la misma estructura y los mismos contenidos.**
- **Árboles binarios equilibrados: Aquellos que en la altura entre su árbol izquierdo y derecho se diferencian como máximo en una unidad.**
- **Árboles binarios completos: Aquellos en que cada nodo del árbol tiene 0 ó 2 hijos, el número máximo de hijos de un determinado nivel i será 2 elevado a i.**
- **Árbol lleno: Cuando todos los niveles están completos.**

Representación de árboles binarios:

Se pueden representar con memoria estática o dinámica. Con memoria dinámica la estructura de un nodo va a tener el campo info y otros 2 nodos que referéncien al hijo izquierdo y derecho de cada nodo.

Representación de árboles binarios con punteros:

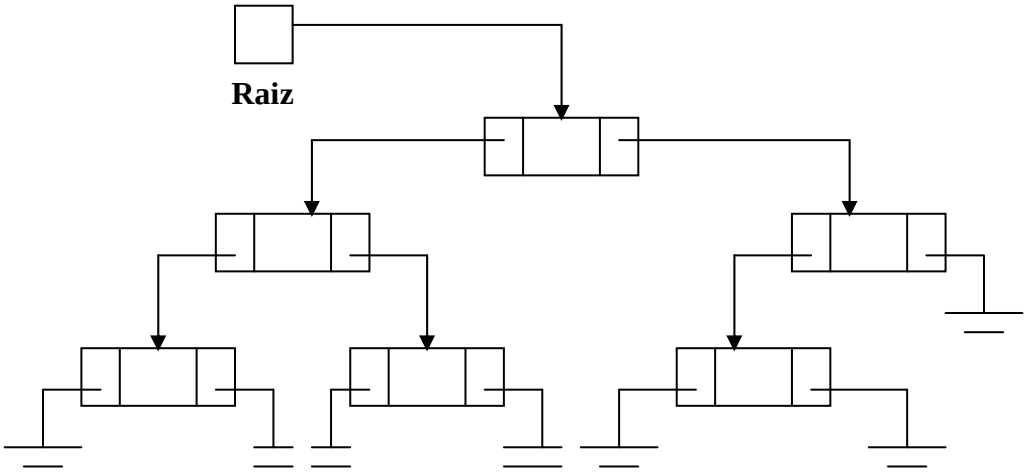
Cada nodo va a tener un registro que contiene el campo de información y 2 campos de tipo puntero.

INFO

IZQ DER

Tipo nodo_arbol: registro
Info: <tipo>
Izq,der: ptr_a nodo_arbol
Fin registro

A parte de los nodos, los árboles tienen una variable especial llamada raíz, que es de tipo puntero a nodo_arbol y apunta al nodo que está al comienzo del árbol.



DISP_AR va a ser una lista enlazada simple en la que las inserciones y borrados los hacemos por el comienzo y como campo de enlace vamos a usar el nodo izquierdo.

<u>Recorrido de los árboles binarios:</u>

Es pasar por todos sus nodos. Para recorrer un árbol lo podemos hacer de 3 formas, según el momento en el que procesemos el nodo raíz. Las 3 formas se llaman : Preorden, Inorden, Posorden.

En cualquiera de las tres, siempre se recorre primero el subárbol izquierdo, y luego el derecho, y se diferencian en el momento en el que recorremos el raíz.

- **Preorden: La raíz se procesa primero, luego el izquierdo y luego el derecho.**
- **Inorden: Primero la raíz, luego el izquierdo y luego el derecho.**
- **Posorden: Primero el izquierdo, luego el derecho y luego la raíz.**

Preorden: A, B, D, E, G, C, F
Inorden: D, B, E, G, A, F, C
Posorden: D, G, E, B, F, C, A

<u>- Recorrido de un árbol binario:</u>

Los arboles son estructuras recursivas, por lo que los algoritmos más eficientes con árboles son los recursivos.

```
Procedimiento preorden (raiz: ptr_a nodo_ar)
Inicio
  Si raiz <> nil
    Entonces {procesar raiz→info)
```

```
                Preorden (raiz→izq)
                Preorden (raiz→der)
  Fin si
Fin

Procedimiento inorden (raiz: ptr_a nodo_ar)
Inicio
  Si raiz <> nil
    Entonces inorden (raiz→izq)
             {procesar raiz→info)
              inorden (raiz→der)
  Fin si
Fin

Procedimiento posorden (raiz: ptr_a nodo_ar)
Inicio
  Si raiz <> nil
    Entonces posorden (raiz→izq)
             Posorden (raiz→der)
             {procesar raiz→info)
  Fin si
Fin
```

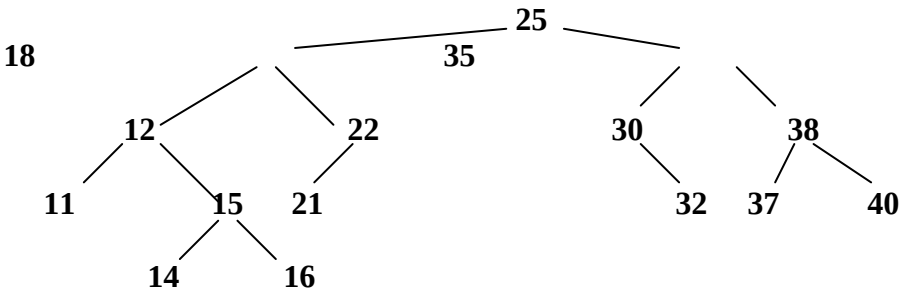
También se puede recorrer un árbol binario con un algoritmo no recursivo:

```
Algoritmo inorden
Var
  Pila: array[1..Max] de puntero a nodo_ar
  Cima: entero
  Raiz, ptr: puntero a nodo_ar
Inicio
  Cima ← 1
  Pila[cima] ← Nil
  Ptr ← raiz
  Mientras (ptr <> nil)
    Cima ← cima + 1
    Pila[cima] ← ptr
    Ptr ← ptr→izq
  Fin mientras
  Ptr ← pila[cima]
  Cima ← cima-1
  Mientras (ptr <> nil)
    {procesar ptr→info}
    si ptr→der <> nil
      entonces ptr ← ptr→der
               mientras ptr <> nil
                 cima ← cima + 1
```

```
                pila[cima] ← ptr
                ptr ← ptr→izq
              fin mientras
    fin si
    ptr ← pila[cima]
    cima ← cima – 1
  fin mientras
Fin
```

3. ÁRBOLES BINARIOS DE BÚSQUEDA:

Son un tipo especial de árboles binarios que se caracteriza porque los elementos del árbol son todos distintos y además están colocados de tal manera que el recorrido en Inorden de ese árbol da lugar a que los elementos se procesen ordenados según un determinado campo de información, para ello es necesario que todos los nodos del árbol cumplan la siguiente regla: Que todo lo que está a la izquierda de un nodo tenga un valor menor que el resto del nodo, y todo lo que esta a la derecha tenga un valor mayor.



Recorrido inorden: 11,12,14,15,16,18,21,22,25,30,32,35,37,38,40.

- Búsqueda de un elemento:

Dado un elemento, mirar si se encuentra o no en el árbol, voy a devolver un puntero a ese elemento y a su padre.

```
Procedimiento buscar (raiz: ptr_a nodo_ar; ent-sal act: ptr_a nodo_ar;
                      ent-sal pad: ptr_a nodo_ar; e:<tipo>)
Inicio
  Pad ← Nil
  Act ← raiz
  Mientras (act <> nil) y (act→info <> e)
    Pad ← act
    Si (e < act→info)
      Entonces act ← act→izq
      Sino act ← act→der
    Fin si
  Fin mientras
  Si act <> nil
    Entonces escribir "Elemento encontrado"
    Sino escribir "Elemento no encontrado"
  Fin si
Fin
```

- Inserción en un árbol binario:

Primero hay que ver que no haya otro elemento igual en el árbol, y después habrá que colocar el elemento a la izquierda de su padre si es menor que él o al derecha si es mayor.

Algoritmo no recursivo:

```
Procedimiento insercion (ent-sal raiz: ptr_a nodo_ar; elem:<tipo>)
Var
  Ptr,pad,nuevo: ptr_a nodo_ar
Inicio
  Buscar (raiz,ptr,pad,elem)
  Si ptr <> nil
    Entonces escribir "El elemento ya existe"
    Sino si DISP = Nil
            Entonces escribir "No hay memoria"
            Sino nuevo ← DISP
                  DISP ← DISP→IZQ
                  Nuevo→info ← elem
                  Nuevo→izq ← nil
                  Nuevo→der ← nil
                  Si PAD = nil
                    Entonces raiz ← nuevo
                    Sino si elem < pad→info
                            Entonces pad→izq ← nuevo
                            sino Pad→der ← nuevo
                           fin si
                  fin si
           fin si
  fin si
Fin
```

Algoritmo recursivo:

```
Procedimiento insercion (ent-sal ptr: ptr_a nodo_ar; elem:<tipo>)
Inicio
  Si ptr <> nil
    Entonces si elem < ptr→info
                  Entonces insercion (ptr→izq,elem)
                  Sino si elem > ptr→info
                          Entonces insercion (ptr→der,elem)
                          Sino escribir "El elemento ya existe"
                         Fin si
                 Fin si
    Sino ptr ← DISP
         DISP ← DISP→izq
         Ptr→info ← elem
         Ptr→izq ← nil
         Ptr→der ← nil
  Fin si
Fin
```

- Borrado en un árbol binario:

Si el nodo a borrar sólo tiene un hijo, lo sustituimos por ese y ya se acabo. Pero si el nodo a borrar tiene 2 hijos, hay que buscar cual es el sucesor de ese nodo

(estará a la derecha del más a la izquierda) y el padre de ese sucesor. Sustituir el nodo a borrar por su sucesor. Al final hay que poner como hijos del sucesor (que es el que ha sustituido al nodo borrado) a los hijos del nodo borrado.

- Algoritmo no recursivo:

```
Procedimiento borrar (ent-sal raiz: ptr_a nodo_ar; elem:<tipo>)
Var
  Ptr, pad: ptr_a nodo_ar
Inicio
  Buscar (raiz,ptr,pad,elem)
  Si ptr = nil
    Entonces escribir "El elemento no existe"
    Sino si (ptr→izq <> nil) y (ptr→der <> nil)
           Entonces borrar_2_h (raiz,ptr,pad)
           Sino borrar_01_h (raiz,ptr,pad)
         Fin si
         Ptr→izq ← DISP
         DISP→izq ← ptr
  Fin si
Fin
```

```
Procedimiento borrar_2_h (ent-sal raiz: ptr_a nodo_ar; ent-sal: ptr_a nodo_ar;
                          pad: ptr_a nodo_ar)
Var
  Suc,padsuc,hijosuc: ptr_a nodo_ar
Inicio
  Padsuc ← ptr
  Suc ← ptr→der
  Mientras suc→izq <> nil
    Padsuc ← suc
    Suc ← suc→izq
  Fin mientras
  Hijosuc ← suc→der
  Si padsuc→izq = suc
    Entonces padsuc→izq ← hijosuc
    Sino padsuc→der ← hijosuc
  Fin si
  Si pad = nil
    Entonces raiz ← suc
    Sino si pad→izq = ptr
           Entonces pad→izq ← suc
           Sino pad→der ← suc
         Fin si
```

```
  Fin si
  Suc→izq ← ptr→izq
  Suc→der ← ptr→der
Fin

Procedimiento borrar_01_h (ent-sal raiz: ptr_a nodo_ar; ent-sal ptr: ptr_a nodo_ar;
                                    Pad: ptr_a nodo_ar)
Var
  Hijo: ptr_a nodo_ar
Inicio
  Si ptr→izq <> nil
    Entonces hijo← ptr→izq
    Sino hijo ← ptr→der
  Fin si
  Si pad = nil
    Entonces raiz ← hijo
    Sino si ptr = pad→izq
          Entonces pad→izq ← hijo
          Sino pad→der ← hijo
        Fin si
  Fin si
Fin
```

- Algoritmo recursivo:

```
Procedimiento borrar (ptr: ptr_a nodo_ar; elem:<tipo>)
Inicio
  Si ptr <> nil
    Entonces si elem < ptr→izq
                Entonces borrar (ptr→izq, elem)
                Sino si elem > ptr→der
                        Entonces borrar (ptr→der, elem)
                        Sino si ptr→izq = Nil
                                Entonces ptr ← ptr→izq
                                Sino si ptr→der = Nil
                                        Entonces ptr ← ptr→izq
                                        Sino eliminar (ptr→der, ptr)
                                      Fin si
                              Fin si
                        Fin si
                        Ptr→izq ← DISP
                        DISP ← ptr
                Fin si
    Sino escribir "El nodo no existe"
```

```
   Fin si
Fin

Procedimiento eliminar (ent-sal suc: ptr_a nodo_ar; ent-sal ptr: ptr_a nodo_ar)
Inicio
   Si suc→izq <> nil
      Entonces eliminar (suc→izq, ptr)
      Sino ptr→info ← suc→info
           Ptr ← suc
   Fin si
Fin
```

4. APLICACIONES DE LOS ÁRBOLES BINARIOS:

- **Una de las aplicaciones más importantes es dentro de la inteligencia artificial, y más concretamente en el área de reconocimiento de patrones.**
  Se trata de utilizar los árboles para realizar clasificaciones. La clave está en asignar a cada nodo del árbol un significado y a cada rama, una respuesta que nos ayude a determinar el sentido de la búsqueda.
- **Transformación de una notación algebraica a otra: Prefija, infija, posfija (+ab,a+b,ab+). Para ello la raíz del árbol debe de ser un signo y para las ramas o subárboles, otra expresión o un operando.**
- **Para implementar el algoritmo de Hodman de compresión y codificación.**

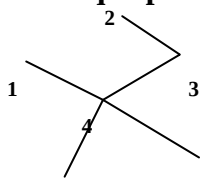
5. GRAFOS:

El grafo es una estructura de datos no lineal, tal que ahora desde un nodo podemos apuntar a varios, pero a su vez un nodo puede ser apuntado desde otros.

Terminología de los grafos:

Un grafo se compone por un conjunto V de vértices y un conjunto A de aristas. Cada arista se identifica con un par de vértices que indican los vértices a los que une la arista. Los vértices de una arista son entre sí nodos adyacentes.

- **Grado de un nodo: Número de aristas que contiene a ese nodo.**
- **Grado de un grafo: Número de vértices de ese grafo. Si el grado de un nodo es 0, se dice que es un nodo aislado.**
- **Camino: Un camino C de longitud N de un nodo V1 a un nodo V2, se define como la secuencia de nodos por los que hay que pasar para llegar del nodo V1 a V2. La longitud de ese camino es el número de aristas que comprende ese camino.**
  El camino es cerrado si empieza y termina en el mismo nodo. El camino es simple si todos los nodos de dicho camino son distintos a excepción de los de los extremos que pueden ser iguales.



C (1,6)
1,4,6 ┐ **Caminos simples**

1,4,5,6–
1,4,3,4,5,6

Aristas especiales:

- **Bucles: Es una arista cuyos extremos son idénticos.**
- **Arista múltiple: Dos o más aristas que conectan los mismos nodos.**

<u>Tipos de grafos:</u>

- **Grafo conectado o conexo: Existe un camino simple entre 2 cualesquiera de sus nodos.**
- **Grafo desconectado: Aquel en que existen nodos que no están unidos por ningún camino.**
- **Grafo dirigido: Cada arista tiene asignada una dirección (identificada por un par ordenado).**
- **Grafo no dirigido: La arista está definida por un par no ordenado.**
- **Grafo sencillo: Aquel que no tiene ni bucles ni aristas múltiples.**
- **Grafo múltiple o multigrafo: Permite la existencia de aristas múltiples o bucles.**
- **Grafo completo: Cada nodo del grafo es adyacente a todos los demás.**
- **Grafo etiquetado con peso ponderado: Aquel en el que a cada arista del grafo va asociado un valor que es lo que llamamos peso de la lista. Se usa para indicar algo, como la longitud de la arista o la importancia de la arista, ...**
- **Peso de un camino: La suma de los pesos de las aristas del camino.**

<u>Representación de los grafos:</u>

Hay 2 formas de representar los grafos, con memoria estática y con memoria dinámica:

Con memoria estática:

- **Matriz de adyacencia: Es una matriz M de N*N elementos donde N es el número de nodos del grafo, donde cada posición M(i,j) índica si hay una conexión o no entre el nodo que aparece asociado a la posición I de la matriz y el nodo que aparece asociado a la matriz J.**

  Si lo que quiero es representar un grafo ponderado en vez de poner 0 y 1, pondremos 0 si no esta conectado y el valor de la arista si existe conexión.

  Si el grafo no es dirigido resulta una matriz simetrica.

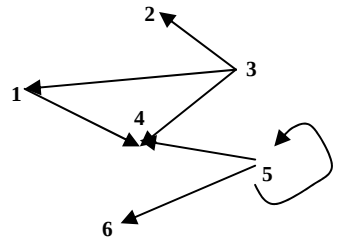
  Las potencias de la matriz de adyacencia M elevado a K nos indican en las posiciones que tiene valor 1 que existe un camino de longitud K entre los nodos asociados a las posiciones I, J de esa matriz.

Destino

	1	2	3	4	5	6
1	0	0	0	1	0	0
2	0	0	0	0	0	0
3	1	1	0	1	0	0
4	0	0	0	0	0	0

5	**0**	**0**	**0**	**1**	**1**	**1**
6	**0**	**0**	**0**	**0**	**0**	**0**

Origen



Con memoria dinámica:

Vamos a utilizar 2 listas enlazadas, una es la lista de nodos (formada por todos los vértices y aristas del grafo.

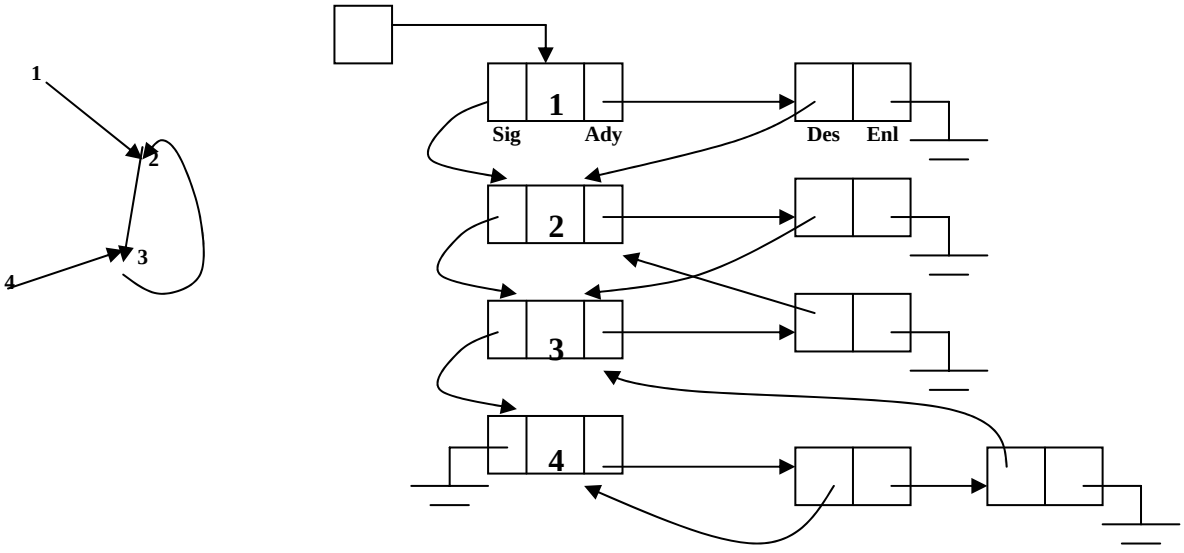
Cada nodo de esa lista tendrá la siguiente información:

- **Info: Información del nodo.**
- **Sig: Puntero que enlaza con el siguiente nodo de la lista de nodos.**
- **Ady: Apunta al primer elemento de la lista de adyacencia.**

La lista de adyacencia contiene o enlaza a todas las aristas que parten de ese nodo. El conjunto de todas las listas de adyacencia, forman una lista de listas.

La lista de adyacencia se implementa como otra lista enlazada en la que cada nodo de esa lista contiene 2 campos:

- **Des (destino): Es un campo puntero al nodo destino de la lista.**
- **Enl (enlace): Es un campo puntero que apunta al siguiente nodo o arista de la lista de adyacencia de ese nodo.**



Tipo vertice: registro

Info: <tipo>

Sig: ptr_a vertice

Ady: ptr_a arista

Fin registro

Tipo arista

Des: ptr_a vertice

Enl: ptr_a arista
(* El peso de la arista iría aquí *)
Fin registro

Para manejar la memoria disponible vamos a tener 2 listas:
- **ADISP: Apunta al primer nodo que enlaza a los nodos de tipo arista disponible.**
- **NDISP: Apunta la primer nodo que enlaza a los nodos de tipo vértices. Nos movemos con el campo sig.**

6. OPERACIONES CON GRAFOS:

- Búsqueda de un nodo:

Lo que se trata es de localizar un nodo que contiene una determinada información, para ello le pasamos la información al procedimiento y nos devuelve la posición si lo encuentra o Nil si no lo encuentra.

```
Procedimiento Buscar_nodo (elem:<tipo>;ent-sal ptr:ptr_a vertice; prin: ptr_a vertice)
Inicio
  Ptr ← prin
  Mientras (ptr→info <> elem) y (ptr <> nil)
    Ptr ← ptr→sig
  Fin mientras
Fin
```

- Búsqueda de una arista:

Se trata de buscar una arista dados sus campos de información origen y destino. Devuelve un puntero a la lista de aristas que apunta a esa arista o Nil si no existe.

```
Procedimiento buscar_arista (prin: ptr_a veritice; orig:<tipo>; des:<tipo>;
                             ent-sal ptr: ptr_a arista; ent-sal p1: ptr_a vertice;
                             ent-sal p2: ptr_a vertice)
Inicio
  Buscar_nodo (prin,orig,p1)
  Si p1 = nil
    Entonces escribir "No existe el nodo origen"
    Sino buscar_nodo (prin,des,p2)
        Si p2 = nil
          Entonces escribir "No existe el nodo destino"
          Sino ptr ← p1→ady
               Mientras (ptr <> nil) y (ptr→des <> p2)
                 Ptr ← ptr→enl
               Fin mientras
        Fin si
```

```
  Fin si
Fin
```

- Inserción de un nodo:

Primero debemos mirar si no existe un nodo que contenga esa información y para insertar el nodo lo insertamos por el principio.

```
Procedimiento insertar_nodo (ent-sal prin: ptr_a vertice; elem: <tipo>
Var
  Nuevo: ptr_a vertice;
Inicio
  Si NDISP = Nil
    Entonces escribir "No hay memoria"
    Sino nuevo ← NDISP
         NDISP ← NDISP→Sig
         Nuevo→info ← elem
         Nuevo→ady ← Nil
         Nuevo→sig ← prin
         Prin ← nuevo
  Fin si
Fin
```

- Inserción de una arista:

Primero hay que comprobar si existen los nodos origen y destino, luego hay que comprobar que no existe una arista igual en esa dirección y para insertarla, como el orden no importa se inserta por el principio.

```
Procedimiento insertar_arista (prin: ptr_a vertice; orig:<tipo>; dest:<tipo>;
                               ent-sal p1: ptr_a vertice; ent-sal p2: ptr_a vertice)
 Var
  Nuevo, pa: ptr_a arista
Inicio
  Si ADISP = Nil
    Entonces escribir "No hay memoria"
    Sino buscar_arista (prin,orig,des,pa,p1,p2)
         Si pa <> nil
           Entonces escribir "La arista ya existe"
           Sino Nuevo ← ADISP
                ADISP ← ADISP→Sig
                Nuevo→dest ← p2
```

```
                Nuevo→enl ← p1→ady
                P1→ady ← nuevo
            Fin si
    Fin si
Fin
```

- Borrado de una arista:

```
Procedimiento borrar_arista (prin: ptr_a vertice; orig:<tipo>; des:<tipo>;
                             p1: ptr_a vertice; p2: ptr_a vertice)
Var
  Act, ant, pa: ptr_a arista
Inicio
  Buscar_arista (prim,or,dest,pa,p1,p2)
  Si pa = nil
    Entonces escribir "La arista ya existe"
    Sino ant ← nil
         Act ← p1→ady
         Mientras (act→dest <> p2) y (act <> nil)
           Ant ← act
           Act ← act→enl
         Fin mientras
         Ant→enl ← act→enl
         Si ant = nil
           Entonces p1→ady ← p1→ady→enl
           Sino ant→enl ← act→enl
         Fin si
         Act→enl ← ADISP
         ADISP ← act
  Fin si
Fin
```

- Borrado de un nodo:

Primero hay que comprobar si existe el nodo y después se borran todas sus aristas.

```
Procedimiento borrar_nodo (ent-sal prin: ptr_a vertice; info:<tipo>)
Var
  Ptr, p1, p2, act, ant: ptr_a vertice
  Pa, borrado: ptr_a arista
Inicio
  Borrar_nodo (prin,elem,ptr)
  Si ptr = nil
    Entonces escribir "No existe el nodo a borrar"
    Sino act ← prin
         Mientras act <> nil
           Borrar_arista (prin,act→info,elem)
           Act ← act→sig
         Fin mientras
         Act→enl ← ADISP
```

```
      ADISP ← ptr→ady    → SOLO EN PSEUDOCÓDIGO
      Ptr→ady ← Nil

      Borrado ← ptr→ady
      Mientras borrado <> nil
        Borrar_arista  (prin,elem,borrado→destino→info)        → EN CUALQUIER
        Borrado ← borrado→enl                                  LENGUAJE
      Fin mientras

      Ant→nil
      Act→prin
      Mientras (act <> nil) y (act→info <> elem)
        Ant ← ant
        Act ← act→sig
      Fin mientras
      Si ant = nil
        Entonces prin ← prin→sig
        Sino ant→sig ← act→sig
      Fin si
      Act→sig ← NDISP
      NDISP ← act
  Fin si
Fin
```

7. APLICACIONES DE LOS GRAFOS:

- **Se pueden representar transformaciones de estado con los grafos.**
- **Sirven en diversos campos de investigación para encontrar el método más corto y que cueste lo menos posible.**
- **Para el álgebra se utiliza la matriz de adyacencia.**

EJERCICIOS: TEMA 12

1. **En un árbol binario de búsqueda, dar todos los elementos inferiores a uno dado con información elem.**

```
Procedimiento buscar (raiz: ptr_a nodo_ar; ent-sal ptr: ptr_a nodo_ar; elem: <tipo>)
Inicio
  Ptr ← raiz
  Mientras (ptr→info <> elem) y (ptr <> nil)
    Si ptr→info > elem
      Entonces ptr ← ptr→der
      Sino ptr ← ptr→izq
    Fin si
  Fin mientras
Fin
```

```
Procedimiento inorden (ptr: ptr_a nodo_ar)
Inicio
  Si ptr <> nil
    Entonces inorden (ptr→izq)
             Escribir (ptr→info)
             Inorden (ptr→der)
  Fin si
Fin

Procedimiento inferiores (raiz: ptr_a nodo_ar; elem: <tipo>)
Var
  Ptr: ptr_a nodo_ar
Inicio
  Buscar (raiz,ptr,elem)
  Si ptr = nil
    Entonces escribir "El elemento no existe"
    Sino inorden (ptr→izq)
  Fin si
Fin
```

2. **Listar en orden descendente el valor de todos los nodos de un árbol binario de búsqueda.**

```
Procedimiento decreciente (p: ptr_a nodo_ar)
Inicio
  Si p <> nil
    Entonces decreciente (p→der)
             Escribir p→info
             Decreciente (p→izq)
  Fin si
Fin
```

3. **Encontrar el máximo valor de un árbol binario de búsqueda.**

```
Procedimiento busca_max (raiz: ptr_a nodo_ar)
Var
  Ptr: ptr_a nodo_ar
Inicio
  Ptr ← raiz
  Mientras ptr→der <> nil
    Ptr ← ptr→der
  Fin mientras
  Escribir ptr→info
Fin
```

4. **Borrar el elemento menor de un árbol binario de búsqueda.**

```
Procedimiento borrar_min (ent-sal raiz: ptr_a nodo_ar)
Var
  Pad, ptr: ptr_a nodo_ar
```

```
Inicio
  Si raiz = nil
    Entonces escribir "Árbol vacio"
    Sino pad ← nil
         Ptr ← raiz
         Mientras ptr→izq <> nil
           Pad ← ptr
           Ptr ← ptr→izq
         Fin mientras
         Si pad = nil
           Entonces raiz ← ptr→der
           Sino pad→izq ← ptr→der
         Fin si
         Ptr→izq ← DISP
         DISP ← ptr
  Fin si
Fin
```

5. Encontrar el segundo elemento mayor de un árbol binario de búsqueda.

```
Procedimiento Dos_mayor (raiz: ptr_a nodo_ar)
Var
  Pad,max,max2: ptr_a nodo_ar
Inicio
  Si raiz = nil
    Entonces escribir "Árbol vacio"
    Sino pad ← nil
         Max ← raiz
         Mientras ptr→der <> nil
           Pad ← max
           Max ← max→der
         Fin mientras
         Si ptr→izq = nil
           Entonces si pad <> nil
                      Entonces escribir "El segundo mayor es" pad→info
```

```
                Fin si
            Sino max2 ← ptr→izq
                Mientras max2→der <> nil
                  Max2 ← max2→der
                Fin mientras
                Escribir "El segundo mayor es" max2→info
        Fin si
    Fin si
Fin
```

6. Añadir un bucle en el nodo con información elem.

```
Procedimiento ins_bucle (prin: ptr_a vertice; elem: <tipo>)
Var
  Ptr: ptr_a vertice
  Nuevo: ptr_a arista
Inicio
  Buscar_nodo (prin,elem,ptr)
  Si ptr = nil
    Entonces escribir "No existe el nodo"
    Sino nuevo ← ADISP
         ADISP ← ADISP→sig
         Nuevo→der ← ptr
         Nuevo→enl ← ptr→ady
         Ptr→ady ← nuevo
  Fin si
Fin
```

7. Dado un grafo, determinar si un nodo con información elem es un nodo fuente (que no le lleguen aristas)

```
Procedimiento nodo_fuente (prin: ptr_a vertice; elem: <tipo>)
Var
  Ptr, pn: ptr_a vertice
  Fuente: boolean
  Pa: ptr_a arista
Inicio
  Buscar_nodo (prin,elem,ptr)
  Si ptr = nil
    Entonces escribir "El nodo no existe"
    Sino si ptr→ady = nil
            Entonces escribir "Nodo sin aristas de origen"
            Sino fuente ← verdadero
                 Pn ← prin
                 Mientras (pn <> nil) y (fuente = verdadero)
                   Pa ← pn→ady
```

```
                    Mientras (pa <> nil) y (fuente = verdadero)
                      Si pa→dest = ptr
                        Entonces fuente ← falso
                        Sino pa ← pa→enl
                      Fin si
                    Fin mientras
                     Pn ← pn→sig
                  Fin mientras
                  Si fuente = verdadero
                    Entonces escribir "Si es un nodo fuente"
                    Sino escribir "No es un nodo fuente"
                  Fin si
            Fin si
      Fin si
Fin
```

8. **Dado un grafo en que cada arista tiene un peso asociado, determinar que aristas del grado pesan más de 100, dando la información del nodo origen y destino.**

```
Tipo arista_peso: registro
        Des: ptr_a vertice
        Enl: ptr_a arista_peso
      Fin registro

Procedimiento ar_peso (prin: ptr_a vertice)
Var
  Pn: ptr_a vertice
  Pa: ptr_a arista_peso
Inicio
  Pn ← prin
  Mientras pn <> nil
    Pa ← pn→ady
    Mientras pa <> nil
```

```
      | Si pa→peso > 100
      |   Entonces escribir "Origen" pn→info
      |   Sino escribir "Destino" pa→des→info
      | Fin si
      Pa ← pa→enl
    Fin mientras
    Pn ← pn→sig
  Fin mientras
Fin
```

TABLAS DE DECISIÓN:
TEMA 13

1. **Qué es una tabla de decisión.**
2. **Tipos de tablas.**
3. **Construcción de tablas.**
4. **Conversión de tablas a programas.**

1. QUÉ ES UNA TABLA DE DECISIÓN:

Una tabla de decisión es una herramienta que me va a servir para representar de manera más fácil la lógica de un problema cuando está es mas o menos complicada.

Para ello se trata de identificar en el problema las acciones que hay que ejecutar y las condiciones que se tienen que cumplir para ejecutar esas acciones. Las acciones normalmente las vamos a identificar a través de los verbos, y las condiciones van a ser las condicionales (si,...).

La tabla va a tener 4 partes:

Conjunto de condiciones o entradas	**Entrada de condiciones combinación de condiciones**
Conjunto de acciones o salidas	**Ejecución de acciones salida de acciones**

Conjunto de condiciones: Son las condiciones que intervienen en el problema.
Entrada de condiciones: Son las combinaciones posibles entre los valores de las condiciones. SI, NO, DA IGUAL.
Conjunto de acciones: Abarca todas las acciones que se tienen que ejecutar cuando se cumplen un conjunto dado de condiciones.
Salida de ejecución: Se determina cuando se ejecuta cada acción.

<u>¿Qué es una regla de decisión?:</u>

Es una combinación de un estado en la entrada de condiciones y de una o más acciones asociadas en la parte de la salida de acciones asociadas en la parte de la salida de acciones siendo N el número de condiciones y considerándolas como binarias (SI/NO) habrá un número máximo de 2 elevado a N reglas.

Cada regla equivale desde el parte de vista de programación a una estructura si.. entonces, y en cada momento solo se puede cumplir una regla.

Las tablas de decisión las podemos usar para controlar la lógica de control de un programa, en los manuales de usuario para controlar un programa, para saber cuando se actualiza un fichero,...

Ejemplo:

Si me hacen fijo me compro un corsa, si me hacen fijo y me suben el sueldo me compro un familiar, y si me toca la lotería me compro un BMW.

C1: Fijo / no fijo
C2: Salario / no salario
C3: Lotería / no lotería

A1: No compro un coche
A2: Compro un corsa
A3: Compro un familiar
A4: Compro un BMW

Fijo	**S**	**S**	**S**	**S**	**N**	**N**	**N**	**N**
Salario	**S**	**S**	**N**	**N**	**S**	**S**	**N**	**N**
Lotería	**S**	**N**	**S**	**N**	**S**	**N**	**S**	**N**
No coche					**X**	**X**	**X**	**X**
Corsa				**X**				
Familiar		**X**						

BMW	**X**		**X**					

2. TIPOS DE TABLAS:

- **Según el planteamiento:**
  - **Propias: Solo tienen reglas simples, solo hay SI / NO.**
  - **Impropias: Son las que tienen reglas compuestas (SI/NO/INDIFERENCIA).**
- **Según entradas:**
  - **Limitadas: Cuando las reglas son S / N / - / X (Si se ejecuta para las acciones).**
  - **Extendidas: Cuando los valores que toman las condiciones son diferentes de SI / NO, y las acciones son distintas de si se ejecuta o no se ejecuta.**

En cualquier caso, siempre se puede traducir una tabla mixta a una tabla limitada. Cada condición dará igual a tantas condiciones simples como valores pueda tomar, y para cada acción tantas acciones como estados pueda tomar.

La ventaja de las tablas mixtas y extendidas es que se ve mejor a primera vista, el inconveniente es que son más difíciles de programar.

- **Según el tratamiento:**
  - **Abiertas: Cuando desde una tabla se hace referencia a otra tabla, pero luego no se vuelve a la inicial.**
  - **Cerradas: Lo mismo que la anterior, pero se vuelve a la original.**
  - **Bucles: Cuando una tabla se llama a sí misma.**

Tipos de reglas:

Normalmente las reglas de una tabla se consideran como reglas AND, se tienen que cumplir todos los valores de las condiciones para que se ejecute la opción asociada a esa regla.

Si se implementa como reglas OR, significa que con que se cumpla una sola regla, se ejecuta la acción asociada.

Las reglas ELSE se hacen cuando un conjunto de reglas de una tabla dan todas lugar a la misma acción, entonces se asocian todas como una regla ELSE.

3. CONSTRUCCIÓN DE TABLAS:

Primero localizamos las condiciones en el enunciado, después las acciones y las reglas entre las acciones y las condiciones.

Después compruebo que la tabla sea correcta. Que no ocurran:

- **Redundancias: Poner la misma regla más de una vez.**
- **Completa: Que no falte alguna regla.**
- **Contradictoria: Que la misma acción de lugar a cosas contradictorias.**

Después se mira a ver si se puede simplificar la tabla, añadiendo si es posible indiferencias (cada una sustituye a 2 reglas).

Una tabla se puede simplificar si hay 2 reglas que dan lugar a la misma acción o conjunto de acciones y solo se diferencia en el estado o valor de una de sus condiciones, entonces sustituyo esas 2 reglas por una sola, que tenga para esa condición el valor de indiferencia.

Fijo	**S**	**S**	**S**	**S**	**N**	**N**	**N**	**N**
Salario	**S**	**S**	**N**	**N**	**S**	**S**	**N**	**N**
Lotería	**S**	**N**	**S**	**N**	**S**	**N**	**S**	**N**
No coche					**X**	**X**	**X**	**X**
Corsa				**X**				
Familiar		**X**						
BMW	**X**		**X**					

4. CONVERSIÓN DE TABLAS A PROGRAMAS:

Primero hay que ver si se puede simplificar la tabla.

Hay 2 formas de hacer la conversión:

- **Programación directa: Que cada regla equivale a una sentencia condicional. Es muy ineficiente.**
- **Pasar la tabla de decisión a flujograma mediante el siguiente el método:**

1. **Para cada regla al final de su columna ponemos el número de reglas simples a las que equivale. Cada indiferencia de una regla equivale a 2 reglas simples.**
2. **Para cada condición al final de su fila hallamos un coeficiente que resulta de la suma de los valores de las reglas para las que esa condición valga indiferente, cuanto menor sea ese valor, más importante es esa regla, porque va a tener menos reglas para las que esa condición es indiferente.**
3. **Elijo la condición con menor valor en su coeficiente, si hay varias condiciones que tienen el mismo valor en su coeficiente y además es el menor, hallo un segundo coeficiente que se obtiene por la diferencia en valor absoluto de la diferencia entre el número de SI y NO en valor absoluto de esas condiciones, y elijo la que tenga la diferencia mayor, y si sigue habiendo empate cojo cualquiera de ellas.**
4. **Para la regla elegida obtengo otras 2 tablas, que se caracterizan porque no contienen ya a la condición por la que he dividido las tablas, y una tabla contiene las reglas para las que esa condición contiene valor SI, y la otra contiene las reglas para las que la condición es NO, y para las reglas con condición indiferencia las pongo en las 2 tablas, y con las 2 tablas vuelvo a hacer otra vez lo mismo hasta que no se pueda dividir más, y cada vez que se hace una división pago la condición en el flujograma.**

Ejemplo anterior:

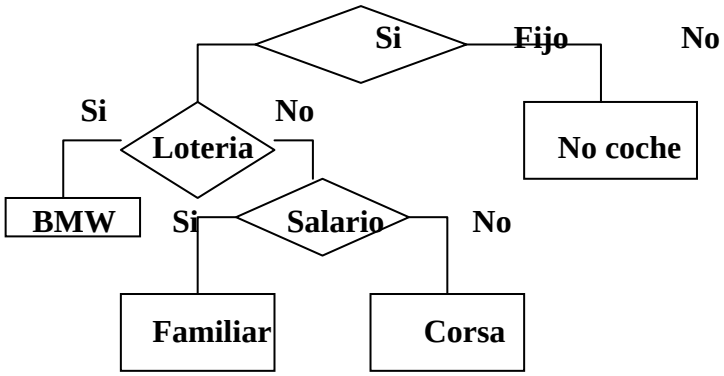
Fijo	**S**	**S**	**S**	**N**	**0**
Salario	**-**	**S**	**N**	**-**	**6**
Loteria	**S**	**N**	**N**	**-**	**4**
No coche				**X**	
Corsa			**X**		
Familiar		**X**			
BMW	**X**				
	2	**1**	**1**	**4**	

Fijo → SI

Salario	**-**	**S**	**N**	**2**
Loteria	**S**	**N**	**N**	**0**
No coche				
Corsa			**X**	
Familiar		**X**		
BMW	**X**			
	2	**1**	**1**	

Fijo → NO

Salario	**-**	
Loteria	**-**	
No coche	**X**	
Corsa		
Familiar		
BMW		




```
Si no fijo
  Entonces escribir "No coche"
  Sino si loteria
        Entonces escribir "BMW"
        Sino si salario
              Entonces escribir "Familiar"
              Sino escribir "Corsa"
            Fin si
      Fin si
Fin si
```

EJERCICIOS: TEMA 13

1. **Dado un fichero de empleados ordenado por número de empleado con el formato, número de empleado, antigüedad, sexo, categoría, se trata de hallar mediante tablas de decisión que prima recibe cada empleado sabiendo que:**
   A: Mujeres que son jefes.
   B: Mujeres empleadas con antigüedad y hombres empleados.
   C: Hombres que son jefes.
   D: Sin prima. El resto.
   Hacer la tabla de decisión y el flujograma.

Antiguedad	**S**	**S**	**S**	**S**	**N**	**N**	**N**	**N**
Mujer	**S**	**N**	**S**	**N**	**S**	**S**	**N**	**N**

Jefe	S	S	N	N	S	N	S	N
A	X				X			
B			X	X				X
C		X					X	
D						X		

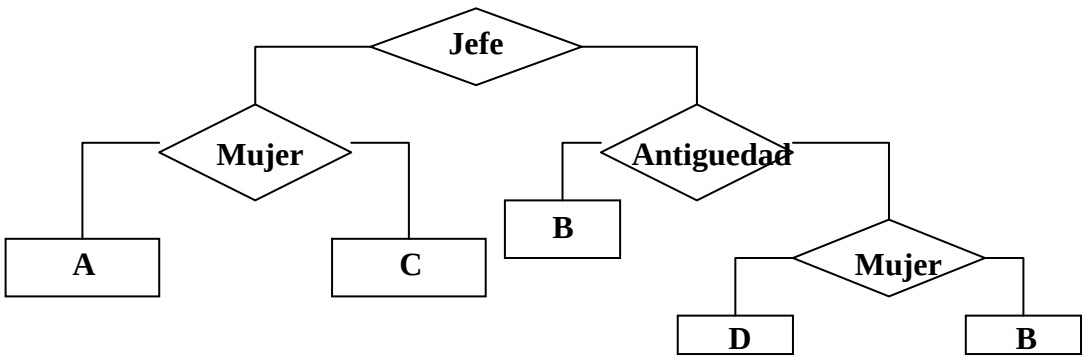
Antiguedad	-	-	S	N	N	4
Mujer	S	N	-	S	N	2
Jefe	S	S	N	N	N	0
A	X					
B			X		X	
C		X				
D				X		
	2	2	2	1	1	

Jefe → SI

Antiguedad	-	-	4
Mujer	S	N	0
A	X		
B			
C		X	
D			
	2	2	

Jefe → NO

Antiguedad	S	N	N	0
Mujer	-	S	N	2
A	X			
B				
C			X	
D		X		
	2	1	1	



Si jefe

```
Entonces si mujer
        Entonces escribir “A”
        Sino escribir “C”
      Fin si
Sino si antigüedad
    Entonces escribir “B”
    Sino si mujer
            Entonces escribir “D”
            Sino escribir “B”
          Fin si
    Fin si
Fin si
```