Tipos de datos definidos por el usuario :

- de tipo simple :
  - Subrango
  - Enumerados
- de tipo estructurado :
  - Conjunto

## SUBRANGO.

Se define a partir de un ordinal estableciendo unos límites inferior y superior para el tipo subrango.
Declaración :

```
Type
    Subrango =liminf...limsup ;
Var
    s :Subrango ;
```

Las variables de un tipo subrango admiten las mismas operaciones que el ordinal del cual proceden. También resulta posible asignar a una variable de tipo subrango otra que ha sido declarada como perteneciente al tipo ordinal del cual se deriva el subrango. Los subrangos se utilizan para dar una mayor legibilidad a los programas.

```
Type
    Meses = 1..12 ;
Var
    m :meses ;
    i :byte ;
Begin
    Repeat
        Readln(i) ;
    Until (i >= 1) AND (i <=12) ;
    m :=i ;
```

Estas variables se emplean también en un array con salario de 12 meses del año :

```
Type
    Meses = 1..12 ;
    Arr =Array [meses] of real ;
```

Los subrangos pueden ser también :

```
Type
    Mayusculas = A..Z ;{de tipo char (ordinal)}
```

## ENUMERADOS.

Han de ser de tipo ordinal. Son aquellos en los que el programador establece los valores que van a poder tomar las variables al declarar el tipo mediante la especificación de una lista de identificadores válidos.
Los identificadores han de comenzar en carácter, y no se pueden repetir en la enumeración de tipos.
Declaración :

```
Type
    Enumerado = (id1, id2, id3, ... ,idn) ;
                    {0   1   2  ...  n}
Var
    e :Enumerado ;
```

Esta variable sólo podrá recibir la lista de identificadores :

```
Begin
        e :=id4 ; {por ejemplo}
```

Los tipos enumerados se utilizan para una mayor legibilidad y para limitar el número de valores que va a poder tomar una determinada variable.
No es posible leer directamente valores de tipo enumerado ni desde teclado, ni desde un archivo de texto. Tampoco se pueden escribir en pantalla ni en archivos de texto (secuenciales) variables de este tipo. Solamente se pueden utilizar para manipulación interna de datos.

Operaciones que admite :

- de relacción (>,<,<>,>=,<=,=...)
- de asignación
- Ord (e) ;
- Pred (e) ;
- Succ (e) ;

El valor ordinal de un dato de este tipo se corresponde con la posición del identificador en la declaración del tipo, y el 1º tiene valor ordinal 0 :

```
Writeln Ord(e)) ; {e :=id4} → {Solución = 3}
```

Las funciones Pred y Succ quedan indefinidas cuando se trata del 1º o último identificador de la lista.
Las variables de tipo enumerado pueden ser variables de control en bucles FOR o usarse de selectores en un CASE.
Ejemplo : Definir tipo enumerado.

```
Type
        Marcas = (seat, fiat, renault, citroen) ;
Var
        m : Marcas ;
Begin
        For m =seat to citroen do
                <acciones >;
```

Para transformar For en While :

```
m :=seat ;
While m <citroen do
Begin
        <acciones> ;
        If m<>citroen then
                m :=Succ(m) ;
        <acciones> ;
End ;
```

En estructuras selectivas :

```
Case m of
        seat : acc1 ;
        fiat :acc2 ;
        renault :acc3 ;
        citroen :acc4 ;
End ;
```

Las variables de tipo enumerado se emplean en Registros Variantes, que se utilizan para ahorrar espacio tanto en la memoria del ordenador (arrays) como en disco (ficheros).

## REGISTROS VARIANTES.

Los Registros Variantes constan de una parte fija que en ocasiones es posible que no la tengan. Si la tienen, ha de ser lo primero que aparezca en la declaración del tipo. A continuación tienen siempre un campo selector y por úlltimo aparece la parte variante.
Ejemplo : Array de empleados

| tipo por horas | B | A |
|---|---|---|
| nombre | nombre | nombre |
| categoría | categoría | categoría |
| nº horas | salario base | salario base |
| ptas/hora | nº horas extra | tlfo casa |
| complementos | tlfo móvil | tlfo móvil |

Necesitaríamos lo siguiente :

| | | |
|---|---|---|
| nombre | salario base | nº horas |
| categoría | ptas/hora | nº horas extra |
| tlfo casa | complementos | tlfo móvil |

Sin embargo, perdemos espacio porque no se rellenan todos los campos.
Registros Variantes (se toma el que más ocupa) (sería el 3º, pero tomamos el 2º)

Parte fija → nombre
campo selector → categoría
(tipo enumerado)

Se utilizan para transformar datos de un tipo a otro.

```
        Type
                c.selector = (porhoras,B,A) ;
        Type
                categoria=(porhoras,B,A) ;{tipo enumerado}
                Reg = RECORD
                        nombre : String [40] ;
                        Case categoria :cargo of {no son anidados}
                            por horas : (numh :byte ;ph :integer) ;
                            B :(salariobase :real ;hextr :byte ;complementos :real ;tlfnomovil :string) ;
                            A : (salariobaseA :real ;tlfno : string[12] ;tlfnomov :string[12]) ;
                        End ;
                End ;
        {para dejar un sólo valor vacío }
        {       c : ( ) ;               }
        {     arr=array[1..100]of reg ;   }
Var
        a :arr ;
        i :integer ;
        resp :string ;
Begin   {leer info. de empleados}
        For i :=1 to 100 do
        Begin
```

```
                Readln(a[i].nombre) ;
                Writeln('Deme categoría (porhoras, A,B)') ;
                Readln(resp) ;
                If resp = 'porhoras' then
                Begin
                        a[i].categoria :=porhoras ;
                        Readln(a[i].numh) ;
                        Readn(a[i].ph) ;
                End ;
                Else
                        If resp='categoria A' then
                        Begin
                                a[i].categoria :=A ; {tipo enumerado}
                                Readln(a[i].salariobaseA) ;
                                Readln(a[i].....etc.....
                                        . . .
                        End ;
                        Else
                                If resp='Clase B' then
                                Begin
                                        a[i].categoria := B ;
                                        Readln(a.[i].salariobase) ;
                                        . . .
                                End ;
                        End ;
                End ;
        End ;
        For i :=1 to 100 do {presenta datos}
        Begin
                Writeln(a[i].nombre) ;
                Case a[i].categoria of {es un enumerado}
                        porhoras :Begin
                                        Writeln(a[i].numh) ;
                                        Writeln(a[i].ph) ;
                                   End ;
                        B : Begin
                                Writeln(a[i].salariobase) ;
                                Writeln(a[i]. .....etc.....
                                . . .
                            End ;
                        A : Begin
                                Writeln(a[i].salariobaseA) ;
                                Writeln(a[i]. .....etc.....
                                . . .
                            End ;
                End ;
        End ;
End.
```

Para ordenarlo se hace como siempre, se intercambian registros completos.

**CONJUNTO.**

Es un tipo de dato estructurado en el que las variables pueden almacenar varios valores de un tipo simple, al cual se le denomina tipo base. La declaración de un tipo conjunto se hace :

```
Type
        Conjunto = SET of tipobase ;
Var
        c :conjunto ;
```

El tipo base ha de ser un ordinal. Los valores ordinales de los elementos han de estar entre 0..255.

Las siguientes declaraciones son ilegales :

```
Conjunto = SET of integer ;
Conjunto = SET of 1997..2000 ;
```

Las siguientes declaraciones son legales :

```
Conjunto = SET of char ;
Conjunto =SET of 'A'..'Z' ;
```

Aunque para facilitar la legibilidad se suele escribir :

```
Type
        Letras ='A'.. 'Z' ;
        Conjunto = SET of Letras ;
```

Ejemplo : Conjunto de letras de una frase :

```
For i :=1 to Lenght(frase) do    {para coger carácter a carácter no puedo Copy(frase,i,1) }
        frase[i] ;                        {porque da una subcadena, y no puede entrar en un conjunto}
```

Podemos simular conjuntos mediante arrays de booleanos :

```
ConjuntoArr = ARRAY [1..6] of boolean ; { aprox. como Conjunto = SET of 1..6 ;}
```

Para introducir un elemento :

```
ConjuntoArr[5] :=True ; {se mete el elemento 5 en el conjunto}
```

En los conjuntos, no puede haber repeticiones en los elementos (solo habrá un elemento de cada).

```
Type
        Letras='A'..'Z' ;
        Conjunto=SET of Letras ;
        Arr= ARRAY['A'..'Z'] of boolean ;
Var
        c :conjunto ;
        a :arr ;
        i :integer ;
```

```
        car :char ;
    Begin

    {usando conjuntos}
        c :=[ ] ; { se asigna conjunto vacío, es obligatorio}
        Readln (car) ;
        c :=c+[car] ; {elemento c UNION conjunto car}{[car] = conjunto del contenido de car}

    {usando arrays}
        For car :='A'.. 'Z' do ;
            a[car] :=False ; {inicialización del conjunto}
        Readln[car] ;
        a[car] :=True ; {introducido elemento en conjunto}
```

Para ver si un elemento está en un conjunto :

```
    {usando conjuntos}
    For car :='A'.. 'Z' do
        If car IN c then        {comprueba que esté en el conjunto}
            Writeln(car) ;

    {usando arrays}
    For car :='A'.. 'Z' do
        If a[car] then    {comprueba que esté en el conjunto}
            Writeln(car) ;
```

Operaciones de conjuntos :

- Definición de tipos :

```
    Type
        Conj=SET of tipobase ; {tipobase=simple y ordinal con valores ordinales entre 0..255}
```

- Declaración de variables :

```
    Var
        c :Conj ;
```

también es posible la forma : c : SET of 'A'..'Z' ; pero tiene poca utilidad, y en programación modular, para el paso por variable, se necesita haber declarado el tipo, ya que :

```
    Procedure Ejemplo(variable p1 :SET of 'A'..'Z') ;
```

no es válido, produciéndose un error de concordancia
Hay que declarar también :

```
    Var
        car :char ;{tipobase}
        c :Conj ;
    Begin
    {hay que INICIALIZAR siempre}
    c :=[ ] ;{inicializamos a conjunto vacío} o c :=['A'..'Z'] ;{inicializa a conjunto universal } ;
```

Para dar valores a variables de tipo conjunto (que no pueden ser leidos desde teclado o escritos por pantalla), debemos hacerlo elemento a elemento, para lo que usamos la variable tipobase.

- ASIGNACIÓN :

```
c :=['A'..'D'] ;
{también se puede asi → c :=['A'..'D','5',car], pero se utiliza poco}
```

las variables han de ser de tipo base.

- INTRODUCCIÓN DE ELEMENTOS :

```
{leemos variable de tipobase}
Repeat
        Readln(car) ;
        car :=Ucase(car) ;
{se comprueba que es del rango al que tiene que pertenecr}
Until (car>='A') AND (car<='Z')
        c :=c + [car] ;          {car → Elemento}
                                 {[car] → Conjunto con el elemento}
        Unión entre conjuntos
```

Una vez introducidos los datos podemos visualizarlos.

- VISUALIZACIÓN DE ELEMENTOS :

Se puede realizar mediante un bucle for :

```
For car : ='A'to 'Z' do
        If car IN c then            {el operador IN admite elemento de distinto tipo. Lo usamos}
                Writeln(car) ;  {para saber si car pertenece a c}
```

Aunque es más conveniente un while (por si sólo hay 2 elementos en el conjunto) :

```
car :='A' :
caux :=c ;      {para no vaciar el conjunto inicial lo metemos en una variable auxiliar}
While (car<='Z') AND (caux < > [ ] then
Begin
        If car IN caux then
        Begin
                Writeln(car) ;
                caux :=caux - [car] ;    {diferencia de conjuntos}
        End ;
        If caux < > [ ] then
                car := Succ(car) ;              {siguiente elemento}
End ;
```

{como car es de tipo char, habrá sucesor. Si fuese de tipo enumerado, podría no haberlo}

Operadores con conjuntos :

| | |
|---|---|
| + | Unión de conjuntos. |
| - | Diferencia de conjuntos. |
| * | Intersección de conjuntos. |
| IN | Pertenencia de elemento en un conjunto. |
| >= | Un conjunto incluye a otro conjunto. |
| <= | Un conjunto es incluido en otro conjunto. |

Prioridad de operadores :

- NOT
- /, *, AND
- -, +, OR
- >,<=, <,>=, =, <>, IN

Simulación de conjuntos :

- Mediante conjunto de enumerados (ya que un conjunto no puede tener cadenas o más de 255 elementos).
- Mediante arrays :

Ejemplo : Crear las primitivas (operaciones básicas) para trabajar con conjuntos.

```
Program Conjuntos ;
Type
        Conj=ARRAY [1..5] of boolean ; {genérico}
Var
        c :Conj ; {este es el conjunto}
        e : 1.. 5 ;
{-------------------------------------------------------------------}
Procedure Inicializar(variable c :conj) ;
Var
        e : 1 .. 5 ;
Begin
        For e :=1 to 5 do
                c[e] :=False ;
End ;
{-------------------------------------------------------------------}
Procedure Introducir (variable c :conj ; e : integer) ;
Begin
        c[e] :=True ;
End ;
{-----------------PROGRAMA PRINCIPAL------------------}
Begin
        Inicializar (c) ;
        Readln(e) ;
        Introducir (c,e) ;
End.
{-------------------------------------------------------------------}
Function En (c :conj ; e :integer) :boolean ;       {Implementación del IN}
Begin
        If c[e] then
                En :=True
        Else
                En :=False ;
        End ;
{lo mismo se puede hacer asi : Begin                    }
{                                       c :=c[e] ;      }
{                               End ;                   }
```

```
{-------------------------------------------------------------------}
Procedure Union (c1,c2 :conj ;variable c3 :conj) ;
Var
        auxi :integer ;
Begin
        For auxi :=1 to 5 do
                c3[auxi] :=c1[auxi] OR c2[auxi] ;
End ;
{-------------------------------------------------------------------}
Procedure Interseccion (c1,c2 :conj ;variable c3 :conj) ;
Var
        auxi :integer ;
Begin
        For auxi :=1 to 5 do
                c3[auxi] :=c1[auxi] AND c2[auxi] ;
End ;
```

## ARCHIVOS.

Los archivos son una colección de datos, tratados como estructura de datos, almacenada en un dispositivo de almacenamiento externo, en la que los datos se encuentran almacenados de forma lógica.
Pascal tiene tres tipos diferentes de archivos :

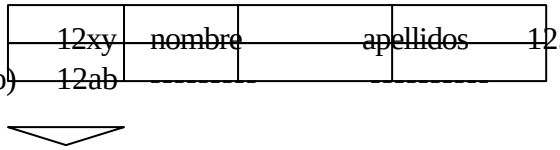
1. Archivos de tipo Texto. (Acceso secuencial)
2. Archivos de tipo 'file of tipobase'. (Acceso Directo)
3. Archivos sin tipo (se declaran como tipo 'file'). (acceso directo)

## ARCHIVOS TIPO TEXTO (TEXT).

Son archivos de texto.La información sólo se puede leer y escribir de forma secuencial (acceso secuencial). Se consideran formados por una serie de líneas las cuales a su vez están formadas por una serie de caracteres. La longitud de éstas líneas no puede exceder 127 caracteres.
En Pascal, los archivos de texto vienen definidos como de tipo 'text'. Para Pasacal, la pantalla y el teclado son archivos de tipo texto asociados con la E/S estándar de DOS.

- Teclado : Input.
- Pantalla : Output.

Pascal estándar obligga a especificar el tipo de archivo, pero Turbo Pascal no.
En un archivo, los registros hay que escribirlos campo a campo.Un registro se leerá de teclado campo a campo. (las variables de tipo registro se leen y escriben campo a campo).
En un archivo de texto se pueden escribir tanto caractéres como numeros. Los datos no sufren una conversión automática a carácter al escribir en pantalla, pero se diferencian en memoria.
Los ficheros de tipo texto son lentos trabajando con números, pero son útiles para pasar datos de un programa a otro.
Son editables. Los archivos de tipo texto, no sirven para escribir cartas, porque todos son registros.
Para Pascal, todo el registro lo considera como una cadena, si está declarado así.



Espacio para 8 caracteres

Se soluciona usando cadenas de longitud fija : si cad[8], sólo lee los 8 caracteres, y después, el nuveno, puede ser otra cadena o una variable numérica.
Gotoxy NO FUNCIONA en archivos de tipo texto.
Éste tipo de archivos es el único ejemplo secuencial de Pascal (los archivos de texto), por lo que NO se pueden abrir a la vez para lectura y escritura.
Para laectura un archivo tipo texto hay que recorrerlo todo desde el principio.
Para escritura, se puede posicionar el puntero de escritura al principio (se borraría todolo anterior al introducir nuevos datos) o al final (se añaden datos).

```
Var
        f : text ; {fichero de texto}
        cad : string ;
Begin
        Assign (f, <nombre_fichero_en_disco>) ;        {Asignamos nombre en el disco}
        {Abre para escritura al principio del fichero y coloca marca de fda}
        Rewrite(f) ;      {hace proceso de creación de archivo secuencial}
        Readln(cad) ;
        While cad < > '*' do
        Begin
                Writeln(f, cad) ; {f,nombre variable tipo fichero}
                Readln(cad)
        End ;
        Close(f) ; {Cerramos el fichero}
End.
```

Cuando queremos leer información, ha de haber sido primero cerrado, si ya está asignado.

```
Reset (f) ;{Abrir lectura }
While NOT EOF(f) do
Begin
        Readln(f,cad) ;
        Writeln(cad) ;
End ;
Close(f) ;
```

La apertura de lectura pone el puntero al principio del archivo en modo lectura. Con la lectura del último registro, se detecta la marca de fin de archivo (EOF).

- Mezcla de 2 ficheros ordenados :

Comenzamos leyendo 2 registros (uno de cada fichero). Los comparamos, escribimos el más pequeño y se lee del que se ha escrito. Repetimos esto hasta que se detecta EOF de uno de los archivos. Si en el otro aún quedan registros, se escriben todos los que faltan (hasta detectar EOF de este fichero) y por último cerramos todos los archivos.

- Longitud de un fichero de texto :

```
Var
        f :text ;
        n1,n2 :integer ;
```

```
Begin
        Assign(f,'num.dat) ;
        Rewrite(f) ;
        Readln(n1) ;
        While n1<>0 do
        Begin
                Readln(n2) ;
                Writeln(f,n1,n2) ;
                Readln(n2)
        End ;
        Close(f) ;
        Reset(f) ;
        While NOT EOF(f) do
        Begin
                Readln(n1,n2) ; →error →84→n1, porque sobra n2, al no haber espacio separador lo
                                                almacena como una sóla variable.
                Writeln(n1,n2) ;                Solución :dar formato.
        End ;                           Writeln(f,n1 :5,n2 :4) o bien Writeln(f,n1,' ',n2) {escribiendo
                                                                                        separador}
        Close(f)
End.

Var
        f :text ;
        n1,n2 :real ;
```

Idem.
El contenido del fichero está en notación exponencial.

Funciones :
En archivos secuenciales se puede escribir con : Write y Writeln.

```
        Write (id, lista expresiones) ;
        Writeln (id, lista expresiones) ;
```

Y se puede escribir con : Read y Readln.

```
        Read (id, lista variables) ;
        Readln (id, lista variables) ;
```

{En archivos de acceso directo NO se pueden usar WriteLn ni tampoco ReadLn}
La impresora (asi como el teclado y la pantalla), también se con sidera como un archivo de texto :
Cada vez que se ejecuta un Write o un Read, el puntero avanza al siguiente registro (tanto en secuencial como en acceso directo).

```
While NOT EOF(f) do          {secuencial : el puntero de registro avanza automáticamente}
Begin
        ReadLn(f,v) ;
        WriteLn(v) ;
End ;

        Var
                Impresora :text ;
        Begin
```

```
        Assign(Impresora,'LPT1') ;
        Rewrite(Impresora) ;
        WriteLn(Impresora,'archivo','hola','adios','5') ;
        Close(Impresora) ;
End.
```

Esto es similar a lo que hay en la unidad PRINTER, por lo que se indica el Uses Pinter, sólo hay que hacer :

```
WriteLn(LST, 'archivo','hola','adios','5') ;
```

Apertura de archivo :

- 1ª vez → Rewrite (f) ; {abierto para escritura}
- Si ya existe → Reset (f) ; {abre para lectura}
  Append(f) ; {abre para escritura, al final (añadir)}

## ARCHIVOS CON TIPO.

Además de archivos de texto, Pascal cuenta con archivos con tipo, que se caracterizan porque los datos son almacenados en disco de la misma manera que se almacenan en memoria. Son archivos de acceso directo, por lo que para acceder a un registro noes necesario recorrer previamente todos los anteriores.
Los archivos de acceso directo no se escriben campo a campo, sino registro a registro.
En esta tipo de archivos no se pueden emplear WriteLn ni ReadLn, porque los dato no están en líneas en el fichero.
Declaración de tipos :

```
Type
    Reg=RECORD
        c1 :tipo1 ;
        c2 :tipo2 ;
            .
            .
            .
    End ;
    Arch = file of Reg ;
```

Declaración de variables :

```
Var
    f : Arch ;
    r :Reg ;
```

Apertura de archivo :

- 1ª vez → Rewrite (f) ; {crea archivo} L/E
- Si ya existe → Reset (f) ; L/E

Para escribir en un archivo directo :

```
Write (f, r) ; { No se escribe campo a campo, sino registro a registro}
        {NO se puede emplear WriteLn porque los datos no están en líneas en el fichero.}
```

Para leer desde un archivo de texto :

```
Read (f, r) ;  { se lee un registro completo, no campo a campo}
        {NO se puede emplear ReadLn porque no hay señal de fda.}
```

Después de ejecutarse un Write o un Read, el puntero avanza al siguiente registro (tanto en secuencial como en acceso directo). En Pascal los registros dentro de un fichero directo se numeran desde el registro 0 (primer registro) hasta el registro n (último registro).

Para posicionarnos en un registro determinado :

- Seek (f, posición) ; {Coloca el puntero en la posición deseada}

Para ver a que registro apunta el puntero actualmente :

- Filepos (f) ; {Registro en el qu está el putero actualmente}

Para conocer el número de registros que tiene el fichero :

- Filesize (f) ;

Para cerrar el fichero :

- Close (f) ;

Para eliminar un fichero : (el fichero deberá estar asignado y cerrado)

- Erase (f) ;

Para renombrar un fichero : (el fichero deberá estar asignado y cerrado)

- Rename(f, 'nuevo_nombre') ;

- <u>Creación de un archivo de acceso directo :</u>

```
Type
        Reg =RECORD
                c1 :
                c2 :
        End ;
        Arch : file of Reg ;
Var
        f :Arch;
        r :Reg ;
        nombre :string ;
Begin
        Write ('Deme nombre de archivo : ') ;
        Readln (nombre) ;
        Assign(f, nombre) ;
        {$I-} {Desactiva las interrupciones por error de E/S. Ioresult ←→ función}
        Reset (f) ; {Abre archivo ya existente para L/E}
        If Ioresult<>0 then {se ha producido error de apertura}
```

```
        Begin
                Rewrite(f) ;{lo crea por si no existe ya}
                Seek (f, Filesize(f)) ;
        End ;
        {$I+}{Reactivamos interrupciones}
        Write('Deme primer campo') ;
        ReadLn (r.c1) ; {valor anómalo}
        While r.c1<>  0       do
                        '*'
        Begin
                Write('Deme resto : ') ;
                ReadLn (r.c2,r.c3,.....) ;
                Write (f, r) ;
                Write ('Deme primer campo') ;
                ReadLn(r.c1)
        End ;
        Seek(f,0) ; {No es necesario cerrar y abrir para leer porque ya está abierto para L/E}
        While NOT EOF(f) do
        Begin
                Read (f,r ) ;
                WriteLn (r.c1,r.c2,r.c3,.....)
        End :
        Close(f)
End. {Reset, cierra y abre al principio, aunque no es lo correcto}
```

En este tipo de archivos, la información se puede escribir tanto secuencialmente (acceso secuencial) como aleatoriamente (acceso directo).

- Actualización de archivos directos, con registros introducidos secuencialmente .

Tendremos dos ficheros : empresa.dat y cambios.dat

```
Type
        Reg = RECORD
                c1 : ----- ;
                c2 : ----- ;
                c3 : ----- ;
        End ;
        Arch = file of Reg ;
Var
        f1,f2 : Archivo ;
        r1,r2 : Reg ;
        Encontrado : boolean ;
Begin
        Assign(f1,'empresa.dat') ;
        Assign(f2,'cambios.dat') ;
        Reset(f1) ;
        Reset(f2) ;
        While NOT EOF(f2) do
```

```
        Begin
                Read(f2,r2) ;
                Encontrado :=False ;
                While NOT EOF(f1) AND NOT Encontrado do
                Begin
                        Read (f1,r1) ;
                        Encontrado := (r1.c1 = r2.c2) ;
                        If Encontrado then
                        Begin
                                Seek(f1,filepos(f1)-1) ;
                                Write(f1,r2) ;
                        End ;
                Write(f1,r2) ;
                Seek (f1,0) ;
                End ;
        Close (f1) ;
        Close (f2) ;
End.
```

**ARCHIVOS INDEXADOS.**

Se buscan por una clave y se lee esa posición. El archivo de datos ha de ser de tipo directo (file of ...).
Si el índice es de tipo texto, se carga en un array (es lo más rápido en lectura secuencial e interesa trabajar con array porque es mucho más rápida la memoria que el disco).
Si el índice es de acceso directo no sabemos el tamaño, por lo que si es muy grande no cabe en memoria, y ordenarlo en disco en complicado y no se puede hacer búsqueda binaria.
Ejemplo de archivo indexado :

```
Uses Crt ;
Type
        Reg = RECORD
                Clave : string[10] ;
                c2 : ----- ;
                c3 : ----- ;
                End ;
        Regi = RECORD
                Clave : string[10] ;
                Posic : integer ;
                End ;
        Arr =ARRAY [1..50]of Regi ;
        Arch = file of Reg ;
        Archi = File of Regi ;
Var
        f : Arch;
        t : Archi ;
        n : integer ;{nº registro fichero}
        a : Arr ;
        i : integer ;
        opcion : char ;
```

```
Begin
        Assign (f,'Datos.dat') ;
        Assign(t,'Indice.dat') ;
        {$I-}
        Reset (f) ;{Suponemos que Datos.dat ya existe, pero puede que no sea así}
        If IOResult<>0 then
                Rewrite(f) ;
        Reset (t) ;{Suponemos que Indice.dat ya existe, pero puede que no sea así}
        If IOResult<>0 then
                Rewrite(t) ;
        {$I+}
        i :=1 ; {cargar en memoria}
        While NOT EFO(t) do
        Begin
                Read (t, a[i]) ;
                i :=i+1 ;
        End ;
        n :=i-1 ;
        Close(t) ; {Ahora que t está en memoria ya no lo necesitamos, por lo que lo cerramos}
        opcion := '  ' ;
        While opcion <>'3' do
        Begin
                WriteLn('-------MENU---------') ;
                WriteLn('1. Altas') ;
                WriteLn('2. Bajas') ;
                WriteLn('3. Fin') ;
                WriteLn('Elija opción :') ;
                opcion :=Readkey ; {necesita Uses Crt ;}

                Case opcion of
                        '1' : Altas(f, a, n) ;
                        '2' ; Bajas(f, a, n) ; { Baja lógica → Quitando el elemento del índice}
                End ;
        End ;
        Rewrite(t) ;
        For i := 1 to n do
                Write(t,a[i]) ;
        Close(t) ;
        Close(f) ;
End.
{---ALTAS-----------------------------------------------------------------------------------------------------}

Procedure Altas(variable f :archivo ; variable a :arr ; variable n :integer) ;
Variable
        r :reg ; {registro del fichero datos (directo)}
        clave :string ;
        encontrado :boolean ;
Begin
```

```
        If n=50 then
                Writeln('Archivo lleno') ;{tabla llena}
        Else
        begin
                WriteLn('Deme clave a dar de alta') ;
                ReadLn(Clave) ;
                While Clave <> '*' do
                begin
                        encontrado :=false ;
                        InsertarEnTabla(a, n, f, clave, encontrado) ;
                        If NOT encontrado then
                        begin
                                WriteLn('Deme resto campos') ;
                                Seek(f, Filesize(f)) ;
                                ReadLn(r.c2, r.c3, ...) ; {campo a campo}
                                r.c1 :=clave ;
                                Write(f, r) ;
                        end
                Else
                        WriteLn('Ya existe') ;
                WriteLn('Deme otra clave o * para FIN') ;
                ReadLn(clave) ;
        end ;
End ;

{--------INSERTARENTABLA----------------------------------------------------------------------------------}

Procedure InsertarEnTabla (var a :arr ;var n :integer ; var f :archivo ; clave :string ;
                        var encontrado :boolean) ;
Var
        i :integer ;
        primero :integer ;
Begin
        BusBin(a, n, clave, primero, encontrado) ; {primero es donde debe estar}
        If NOT encontrado then
        begin
                For i :=n downto primero do
                        a[i+1] :=a[i] ;
                a[primero.clave] :=clave ;
                a[primero.posicion] :=Filesize(f) ; {todavía no está colocado en f}
                n :=n+1 ;
        end ;
End ;

{No haces ordenación de tabla, y sirve al crearlo y al dar altas}

{--------------BAJAS-------------------------------------------------------------------------------------
--}
```

```
Procedure Bajas(var a :arr ; var n :integer) ;
Variable
        clave :string ;
        encontrado :boolean ;
Begin
        WriteLn('Deme clave a dar de baja') ;
        ReadLn(Clave) ;
        While Clave <> '*' do
        begin
        BorrarDeTabla(a, n, clave, encontrado) ;
        If NOT encontrado then
        begin
                WriteLn('No está') ;
                WriteLn('Deme otra clave o * para FIN') ;
                ReadLn(clave) ;
        end ;
End ;

{---------------BORRARDETABLA--------------------------------------------------------------------------}

Procedure BorrarDeTabla(var a :arr ; var n :integer ; clave :string ; var encontrado :boolean) ;
Var
        primero :integer ;
        i :integer ;
Begin
        encontrado :=false ;
        BusBin(a, n, clave, primero, encontrado) ;
        If encontrado then
        begin
                For  i :=primero to n do
                        a[i] :=a[i+1] ;
                n :=n-1 ;
        end ;
End ;

{--------------BUSQUEDA BINARIA-----------------------------------------------------------------------}
Procedure BusBin(a :arr ; n :integer ;clave :string ;var p :integer ; var encontrado :boolean) ;
Var
        primero,ultimo,central : integer ;
Begin
        encontrado :=false ;
        primero :=1 ;
        ultimo :=n ;
        While (primero<=ultimo) AND (NOT encontrado) do
        begin
                central :=(primero+ultimo) DIV 2 ;
                If a[central.clave] :=clave then
```

```
                begin
                        encontrado :=true ;
                        p :=central ;
                end
        Else
                If a[central.calve] >clave then
                        ultimo :=central-1
                Else
                        ultimo :=central+1 ;
        end ;
        If NOT encontrado then
                p :=primero ;
End ;

{----------CONSULTA----------------------------------------------}

BusBin(.....pos.....) ;
If encontrado then
begin
        Seek (f, a[primero].posicion) ;
        Read(f, r) ;
        WriteLn(r.c1, r.c2, r.c3,.....) ;
        ---------------------------------- ;
end;
```

## ARCHIVOS DE ACCESO DIRECTO POR TRANSFORMACIÓN DE CLAVES.

Se asigna mediante una función HASH una posición a unas claves determinadas.
{por ejemplo, una función HASH sería dividir el número de clave entrew el número de elementos totales, y tomar el resto como resultado}
El problema con que nos vamos a encontrar son las Colisiones (elementos con claves distintas producen el mismo resultado en la función HASH).
Los elementos que colisionan se pueden llevar a :

- Otro fichero (Fichero de Colisiones).
- Al mismo fichero a partir de la íltima posición.

Si llegamos a una posición y la clave no es igual, se va a buscar a la zona de colisiones (secuencialmente).
{por ejemplo, si tengo 100 elementos, el número primo más próximo a 100 dará menos colisiones, y ese será el número de zona (de colisiones)) }

```
Type
        Reg=RECORD
                clave :integer ; {suele ser string → transformarlo a número}
                c1 : ------- ;
                c2 : ------- ;
                ocupado : boolean ; { también podemos usar un Char}
                end ;

        Archivo=file of Reg ;
Var
        r :Reg ;
        f :Arch;
        i :integer ;



{--------------HASH-----------------------------------------------------------------------------------------}
Function Hash (clave :string) :integer ; {por ejemplo si la clave fuese '1234R'}
Var
        suma :integer ;
        i :byte ;
Begin
        suma :=0 ;
        For i :=1 to Length(clave) do
                suma :=suma+Ord(clave[i]) ;
        Hash :=suma MOD 100 ;
End ;

BEGIN
        Assign(f,'nombre.dat') ;
        Rewrite(f) ; {creación}
        r.ocupado :='' ;
        For i :=0 to 99 do
        begin
                Seek(f,i) ;
                Write(f,r) ;
        end ;
END.

{-------------------------------------------------------------------------------------------------------------
---------}
Program Altas ;
Type...

Variable
        r :Reg ;
        f :Archivo ;
        Clave :integer ;
```

```
        encontrado :boolean ;
        p :integer ;
Begin
        Reset(f) ;
        WriteLn('Deme clave') ;
        ReadLn(clave) ;
        While Clave<>0 do
        begin
                p :=Hash(clave) ;
                Seek(f,p) ;
                Read(f,r) ;
                If r.ocupado='' then
                        encontrado :=true ; {el sitio}
                Else
                begin
                        p :=99 ;
                        encontrado :=false ;
                        While (p<124) AND NOT encontrado do
                        begin
                                p :=p+1 ;
                                Seek(f,p) ;
                                Read(f,r) ;
                                If r.ocupado='' then
                                        encontrado :=true ;
                        end ;
                end ;
                If encontrado then
                begin
                        Seek(f,p) ;
                        Write('Deme campos) ;
                        ReadLn(c1,c2,.... ) ;
                        r.ocupado :='*' ;
                        r.Clave :=Clave ;
                        Write(f,r) ;
                end ;
                Write('Deme tora clave o 0 para fin ') ;
        end ;
        Close(f) ;
End.
{-------------------------------------------------------------------------------------------------------------
---------}

f :text ; {obligatorio}

ReadLn(f,r,...) ;
While NOT EOF(f) do
begin
        Write(...) ;
```

```
        Read(...) ;
end ; {último no se escribe}
WriteLn(....) ;
<<<Arreglado>>>{aquí no hay que escribir cuando sale}
fin :=false ;
If NOT EOF(f) then
        ReadLn(f,....) ;
Else
        fin :=true ;
While NOT fin do
begin
        Write(f,r,...) ;
        If NOT EOF(f) then
                ReadLn(f,r,...) ;
        Else
                fin :=true ;
end ;

{-------------------------------------------------------------------------------------------------------------
---------}
Program Ordenacion_Mezcla_Natural ;
Type
        Reg=RECORD
                cod :byte ;
                cantidad :integer ;
                descuento :real ;
                end ;
        Archivo =text ;
Variable
        f1,f2,f :Archivo ;
        r1,r2,r :Reg ;
        ant1,ant2,ant :byte ; {cod}
        ordenado,crece :boolean ;
        fin1,fin2,fin :boolean ;
        numsec :integer ;
Begin
        Assign(f,'inicial.dat') ;
        Assign(f1,'auxi1.dat') ;
        Assign(f2,'auxi2.dat') ;
        ordenado :=false ;
        While NOT ordenado do
        begin
                Reset(f) ;
                Rewrite(f1) ;
                Rewrite(f2) ;
                fin :=false ;
                If NOT EOF(f) then
                        ReadLn(f,r.cod,r.cantidad,r.descuento) ;
```

```
Else
        fin :=true ;
While NOT fin do
begin
        ant :=r.cod ;
        crece :=true ;
        While (NOT fin) AND crece do
                If ant<=r.cod then
                begin
                        WriteLn(f1, r.cod, r.cantidad, r.descuento) ;
                        ant :=r.cod ;
                        If NOT EOF(f) then
                                ReadLn(f, r.cod, r.cantidad, r.descuento) ;
                        Else
                                fin :=true ;
                        End
                Else
                        crece :=false ;
        ant :=r.cod ;
        crece :=true ;
        While (NOT fin) AND crece do
                If ant<=r.cod then
                begin
                        WriteLn(f2, r.cod, r.cantidad, r.descuento) ;
                        ant :=r.cod ;
                        If NOT EOF(f) then
                                ReadLn(f, r.cod, r.cantidad, r.descuento) ;
                        Else
                                fin :=true ;
                        End
                Else
                        crece :=false ;
End ;
Close(f) ;
Close(f1) ;
Close(f2) ;
{ fin partición }
Reset (f1) ;
Reset (f2) ;
Rewrite(f) ;
fin1 :=false ;
fin2 :=false ;
If NOT EOF(f1) then
        ReadLn(f1, r1.cod, r1.cantidad, r1.descuento)
Else
        fin1 :=true ;
If NOT EOF(f2) then
        ReadLn(f2, r2.cod, r2.cantidad, r2.descuento) ;
```

```
        Else
                fin2 :=true ;
        numsec :=0 ;
        While (NOT fin1) AND (NOT fin2) do
        begin
                ant1 :=r1.cod ;
                ant2 :=r2.cod ;
                crece :=true ;
```

< falta la página 15 y la mitad de la cara dedetrás porque no la entiendo. Preguntar a Marta>

## RECURSIVIDAD.

Es una alternativa a la iteración, que además resulta menos eficiente que ella en términos de ordenador.
La recursividad pretende ayudar al programador cuando trata de resolver problemas que por su naturaleza están definidos en términos recursivos.
Ejemplo : Factorial de un número

n ! = n (n-1) !

Se define un problema recursivamente cuando se resuelve en términos más sencillos del mismo problema. Puede ser :

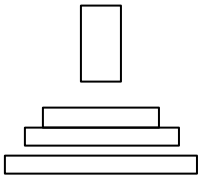
a. Directa : La recursividad es directa cuando un procedimiento se llama a sí mismo pasando cada vez a un paso más pequeño.
b. Indirecta : La recursividad indirecta consiste en que un procedimiento llame a otro que a su vez vuelve a llamar al primero.

y necesita siempre del uso de procedimientos y funciones.
En Pascal tendríamos que empezar :

```
Program Recursividad ;

Procedure b ; Forward
Procedure a ; Forward
begin
        ... ;
        b ;
        ... ;

Procedure b ;forward
begin
        If <condición> then
                a ;
end ;
```

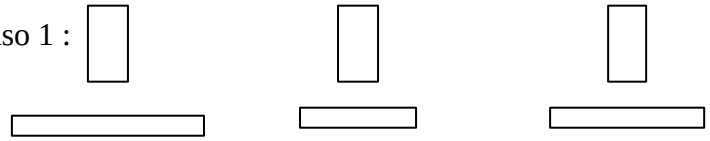
Forward permite recursividad indirecta.
Cuando un procedimiento se llama de forma recursiva los parámetros por valor y las variables locales se almacenan en la pila. El procedimiento trabaja con esos valores y cuando sale, retorna al nivel anterior y recupera los valores de la pila del nivel anterior.

Ejemplo : Las torres de Hanoi
Estado inicial :

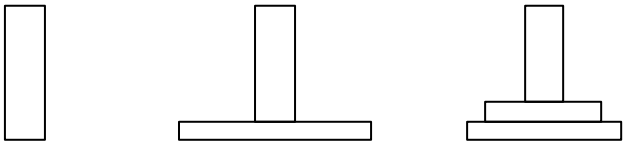


Origen Destino Auxiliar

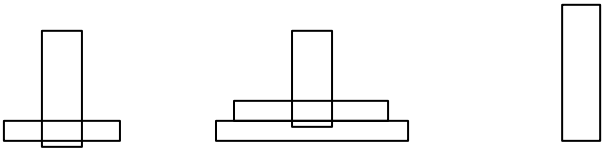
Paso 1 :



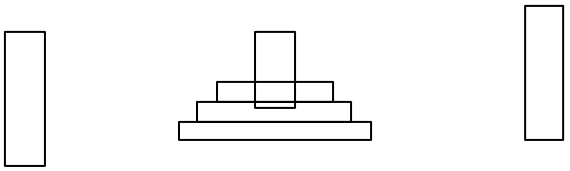
Paso 2 :



Paso 3 :



Paso 4 :



Origen Destino Auxiliar

La secuencia sería :

1 a 2
1 a 3
2 a 3
1 a 2
3 a 1
3 a 2
1 a 2

Program Torres ;
Var

```
        n :integer ;
Procedure Mover(n, origen, destino, aux :integer) ;
Begin
        If n=1 then {condición de salida }
                WriteLn('De',origen,'a',destino)
        Else
        begin
                Mover(n-1, origen,aux,destino) ;
                WriteLn('De',origen,'a',destino) ;
                Mover(n-1, aux,destino,origen) ;
        end ;
End ;

Begin
        ReadLn(n) ;
        Mover(n,1,2,3) ;
End.
```

```
                                        1 1 2 3
                        2 1 3 2
                                        1 2 3 1
        3 1 2 3
                                        1 3 1 2
                        2 3 2 1
                                        1 1 2 3
```

Ordenación Recursiva : QUICKSORT.

Divide una lista en otras 2 para lo cual toma un elemento y coloca en una sublista todos los elementos menores y en otra sublista todos los mayores que el elegido. Repite todo este proceso con cada una de las sublistas que van surgiendo hasta que éstas consten de un único elemento, en cuyo momento termina la ordenación.

```
Procedure QuickSort (variable a :arr ; izq,der :integer) ;
Var
        i, j :integer ;
        pivote :integer ; {tipo elemento del array}
Begin
        i :=izq ;
        j :=der ;
        pivote :=a[(izq+der) DIV 2] ;
        Repeat
                While a[i]< pivote do
                        Inc(i) ;
                While a[j]> pivote do
                        Dec(j) ;
                If i<=1 then
                begin
```

```
                Intercambiar(a[i], a[j]) ;
                Inc(i) ;
                Dec(j) ;
            end ;
        Until i > j ; {está hecho para números repetidos}
        If izq < j then
            QuickSort(a, izq, j) ;
        If i< der then
            QuickSort(a, i, der) ;
End ;
```

En la pila sólo entran valores por valor, y las variables locales.

1 6 4 4 5

1 6 3 5 5

1 6 1 6 5

## ESTRUCTURAS DINÁMICAS DE DATOS.

Son aquellas en las que el espacio que ocupan en memoria se determinan en tiempo de ejecución a diferencia de las estáticas, en las que el espacio que ocupaban se establecían en tiempo de compilación. Una variable dinámica se crea en tiempo de ejecución, y necesita para su creación del uso de punteros.

Las variables de tipo puntero se representan  y almacenan la dirección de memoria donde se encuentra la variable dinámica apuntada.

Declaración :

```
    Type
        puntero=ìnteger ; {se reserva en memoria espacio para enteros}
    Var
        p, q : puntero ;
```

Operaciones con punteros :

a. Asignación :

```
p :=NIL ; {valor nulo, para inicialización}
p :=q ; {p apuntará a donde apunte q }
```

b. Crear la variable dinámica :

```
New(p) ; {Reserva espacio para la variable dinámica. Con New(p) se crea la variable dinámica apuntada. }
        { Ésta no tiene nombre porque no está declarada, por lo que nos referiremos a ella como :
        {                   p^   {p apuntada}                                   }
        {almacena en p una dirección de memoria donde se pueden guardar datos de tipo entero }

Dispose(p) ; {Libera ese espacio de memoria ocupado por la variable dinámica apuntada }
```

c. Comparación :

Sólo acepta < > y = (nunca > o < ).Ej : If p < > q then

Los punteros no pueden ser leidos de teclado ni verlos en pantalla.
Las variables dinámicas admiten cualquiera de las operaciones que admitiría una variable de ese tipo.
Si se puede :

```
ReadLn(p^) ; { porque p^ es un entero }
WriteLn(p^) ;
```

Nunca :

```
ReadLn(p) ; { porque p es un puntero }
```

Las variables dinámicas (apuntadas) pueden ser de cualquier tipo :

```
Type
    puntero=^integer ;
        ^real ;
        ^char ;
        ^string ;
        ^array ;
        ^Registro ; {Registro se debe declarar ántes que el puntero :
                        Registro=RECORD
                                    c1 :char ;
                                    c2 :integer ;
                                end ; }
```

Si es puntero a variable de tipo registro :

```
Var p :puntero
Begin
    New(p) ;    { p → [ char | integer ]  ←creada en}
                {en tiempo de ejecución}
    ReadLn(p^.c1) ;
    ReadLn(p^.c2) ;
```

Otro caso : {estructura de datos enlazada}

```
Registro=RECORD
        c1 :char ;
        c2 :puntero ;{que no está declarado, se declarará después}
    end ;
puntero=^Registro ;

Var p :puntero ;
Begin
    New(p) ;
    New(p^.c2) ;
    ReadLn(p^.c1) ;{ no se puede leer p^.c2 porque es puntero}
    ReadLn(p^.c2^.c1) ;
```

Como no es práctico se maneja de otra manera :

```
Type
    puntero=^nodo ;{→Registro especial y uno de sus campos es puntero}
```

```
        nodo=RECORD
                c1 :char ;
                p :puntero ;
        end ;
    Variable p :puntero ;
```

Las estructuras dinámicas de datos pueden ser :

a. Lineales (Listas, Pilas y Colas) {Las pilas y colas son tipos especiales de listas}
   En una estructura dinámica lineal, cada elemento tiene un único elemento sucesor o siguiente y cada elemento y un único elemento predecesor o anterior.

b. No Lineales (Árboles y Grafos)
   Las estructuras no lineales se caracterizan porque sus elementos pueden tener más de un elemento sucesor o siguiente (Árboles : un sólo predecesor y varios posibles sucesores). Los Grafos pueden tener más de un elemento predecesor (varios predecesores y varios sucesores).

Hay muchas variantes porque cada nodo puede tener más punteros.

## LISTAS.

Una Lista es una colección o serie de datos del mismo tipo que se almacenan en la memoria del ordenador. En una lista cada elemento podrá tener un único sucesor o siguiente y un único antecesor. Además la inserción y elimminación de elementos se podrá efectuar en cualquier lugar de la lista. Como casos especiales de listas son las pilas y las colas, en las que la inserción o eliminación de elementos está restringida a efectuarse por un determinado lugar.
Las Listas pueden ser de varios tipos :

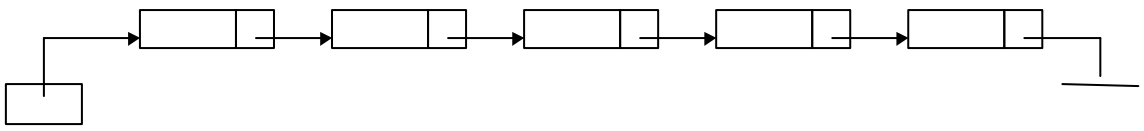
a. Listas Contiguas :

   En ellas, los elementos que las constituyen ocupan posiciones consecutivas en memoria. Se suelen implementar por medio de arrays.

b. Listas Enlazadas :

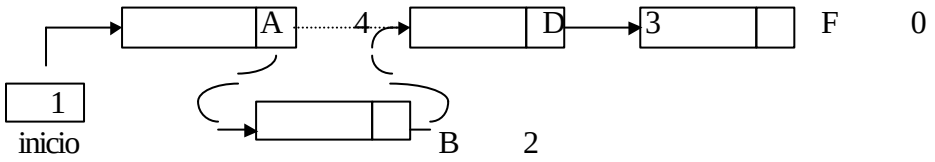
Son las 'verdaderas listas'. Tienen la ventaja de que la inserción y eliminación de un elemento en cualquier lugar nunca obliga al desplazamiento de los restantes elementos de la lista. En las listas enlazadas, los elementos no ocupan posiciones consecutivas de memoria, por lo que cada uno de ellos se ve obligado a almacenar la posición o dirección de memoria donde se encuentra el siguiente elemento de la lista. Para recorrer una lista enlazada hay que hacerlo siempre desde el principio de la lista (para seguir los punteros). Estas listas se representan con punteros en general, lo que consigue que la compilación en memoria pueda variar en la ejecución del programa.

Las listas, sin punteros, se simulan mediante arrays, lo que provoca el inconveniente de que el espacio reservado en memoria es constante (deja de ser una estructura dinámica para convertirse en estática).



Mediante arrays sería : (por ejemplo la inserción del elemento B)
(memoria)

| | | |
|---|---|---|
| 1 | A | 4 |
| 2 | D | 3 |
| 3 | F | 0 |
| 4 | B | 2 |
| 5 | | |
| 6 | | |
| ... | | |




(con punteros enteros)

Se inserta en su sitio (el hueco) sin despazar elementos.

Para borrar hacemos que inicio apunte al siguiente.

Para simular New y Dispose :

En principio toda la memoria del ordenador es una lista con posiciones vacías. Con New(p) lo quitamos de la lista de vacío.

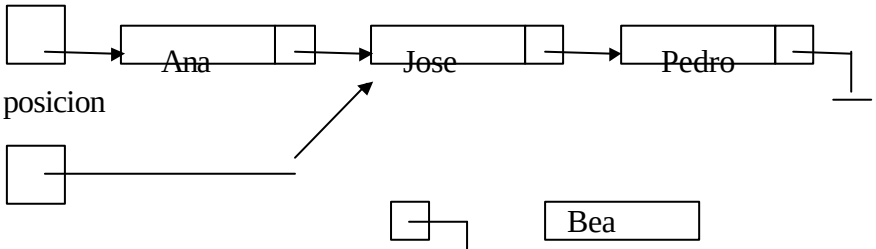
```
Type
        puntero=^nodo ;
        nodo=RECORD
                elem :TipoElemento ;
                SIG :puntero ;
        end ;
Var
        inicio, anterior, posición :puntero ;
        elemento :Tipoelemento ;
        encontrado :boolean ;

Procedure Inicializar(variable inicio :puntero) ;
begin
        inicio :=NIL ;
end ;

Function Vacía(inicio :puntero ;encontrado :boolean) ;
begin
        vacía :=inicio=NIL ;
end ;

Procedure Insertar(elemento :Tipoelemento ;anterior :puntero ;variable inicio :boolean) ;
{distinguir entre :insertar1º lista o cualquier otro}
Var
         auxi :puntero ;
begin
        New(auxi) ;
        auxi^.elem :=elemento ;
        If anterior=NIL then
        begin
                auxi^.sig :=inicio ;
                inicio :=auxi
        end
        else
        begin
```

```
                auxi^.sig :=anterior^.sig ;
                anterior^.sig :=auxi
        end
end ;
```

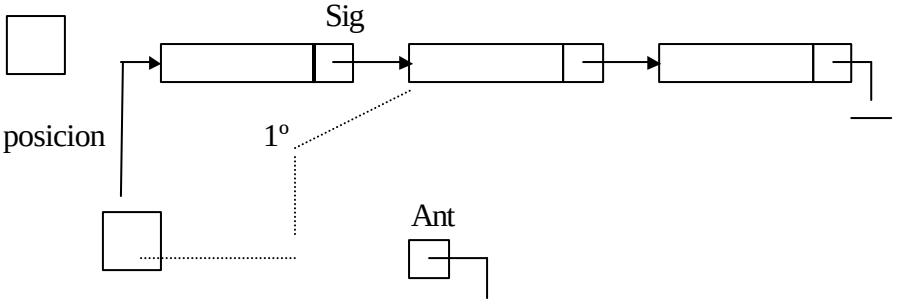
En un array :

Reservar(auxi) en lugar de New(auxi).
a[auxi].sig en lugar de auxi^.sig.

Elementos ordenados en una lista :




```
Procedure Consultar (variable anterior, posicion,inicio :puntero ; elemento :
                        Tipoelemento ;variableencontrado :boolean) ;
Var
        Salir :boolean ;
Begin
        anterior :=NIL ;
        salir :=false ;
        posicion :=inicio ;
        While (posicion<>NIL) AND (NOT salir) do
                If posicion^.elemento<elemento then
                Begin
                        anterior :=posición ;
                        posicion :=posicion^.sig ;
                end
                Else
                        salir :=true ;
        If (posicion<>NIL) AND (posicion^.elemento=elemento) then
                encontrado :=true
        Else
                encontrado :=false ;
End ;
```



Inicio

- Suprimir :
  - 1º de la lista.
  - Cualquier otro.

```
If ant = NIL then
        inicio :=posic^.Sig ;
{ liberar posición }

If ant <> NIL then   { consulta da anterior y posicion del que vaya a eliminar }
        ant^.sig := posic^.sig
        { liberar posicion }
```

Si es el último da igual (no hay que considerarlos).

```
{-------------------------------------------------------------------------------------------}
Procedure Suprimir (ant, posicion: puntero ;var inicio : puntero) ;
Begin
        If ant = NIL then
                inicio :=posicion^.Sig
        else { consulta da anterior y posicion del que vaya a eliminar }
                ant^.sig := posicion^.Sig ;
        Dispose (posicion) ; { libera posicion}
end ;
```

- Recorrer una Lista :

```
Procedure Recorrer(inicio :puntero) ;
Var
        posicion :puntero ;
Begin
        posicion :=inicio ;
        While posicion<>NIL do
                Begin
                        WriteLn(posicion^.elemento) ;
                        posicion :=posicion^.Sig ;
                end ;
End ;
{-------------------------------------------------------------------------------------------}
```

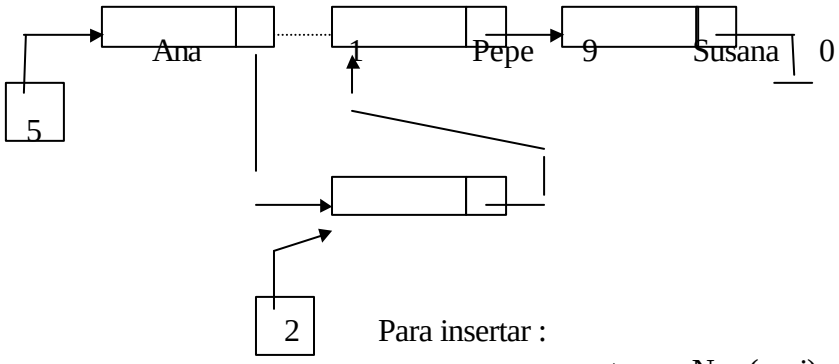
- Simulación de una Lista Enlazada con Arrays :

| | | elemento | sig(dirección del siguiente elemento de la lista) |
|---|---|---|---|
| | 1 | Pepe | 9 |
| Inicio1 | 2 | Carolina | 7 |

| | | | | |
|---|---|---|---|---|
| 5 | 3 | | | |
| | 4 | | | a[1].elemento |
| | 5 | Ana | 2 | a[1].Sig |
| Inicio2 | 6 | | | |
| (libres) | 7 | Federico | | 1 |
| | 8 | | | |
| | 9 | Susana | 0 | |

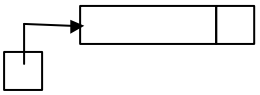
Inicialmente no es así, porque de las 2 listas inicio hay un alista de vacío :

| | | elemento | sig(dirección del siguiente elemento de la lista) | |
|---|---|---|---|---|
| | 1 | Pepe | 9 | |
| Inicio1 | 2 | xxxxxxxxx | 1 | |
| 5 | 3 | | | |
| | 4 | | | a[1].elemento |
| | 5 | Ana | 2 | a[1].Sig |
| Inicio2 | 6 | | | |
| (libres) | 7 | | | |
| | 8 | | | |
| | 9 | Susana | 0 | |




Para insertar :

- con punteros : New(auxi) ;



- con arrays : Reservar (auxi) ;

El procedimiento Insertar de una estructura dinámica simulada por un array es igual que el de la estructura dinámica sin simular.

```
Procedure Insertar (var a :arr ; elemento :Tipoelemento ;ant :integer ; var inicio, vacio :integer) ;
Var
        auxi :integer ;
Begin
        Reservar(auxi^) ; { equivale a New(auxi) en punteros}
        a[auxi].elemento :=elemento ;
        If ant=0 then
        begin
```

```
                a[auxi].sig :=inicio ; {a[auxi].sig equivale a auxi^.sig en punteros}
                inicio :=auxi ;
        End
        Else
        begin
                a[auxi].sig :=a[ant].sig ;
                a[ant].sig :=auxi ;
        end ;
End ;
```

■ Gestión de libres :

Al introducir el primer libre :
(Reservar (auxi))
auxi ← vacío {1}
vacío ← a[vacío].sig {2}
Para insertar otro :
(Reservar (auxi))
auxi← vacío {2}
vacío← a[vacío].sig {3}...

```
Procedure Reservar (variable auxi :integer ;a :arr ;variable vacío :integer) ;
Begin
        auxi :=vacío ;
        vacío :=a[vacío].sig ;
End ;
```

■ Borrado de un elemento :

Primer caso (ant = 0) :
inicio← a[posicion].sig
Liberar posición.
Otros casos (ant<>0) :
a[anterior].sig :=a[posicion].sig
Liberar posición.

```
{-------------------------------------------------------------------------------------------------}
Procedure Suprimir(var ant, inicio :integer {puntero} ;elemento :Tipoelemento ; var a :arr ;
Begin                                   variable vacio :integer ;variable posicion :integer) ;
        If ant=0 then {ant =NIL}
                inicio :=a[posicion].sig
        Else
                a[ant].sig :=a[posicion].sig ;      {ant^.sig :=posicion^.sig}
        Liberar (posicion,a,vacio) ;       {Dispose (posicion) }
        (ant :=0 ;posicion :=inicio) ; {para poder hacer estas llamadas sin consultar}
End ;
{-------------------------------------------------------------------------------------------------}
Procedure Liberar (posicion :integer ;var a :arr ;var vacio :integer) ;
Begin
```

```
        a[posicion].sig :=vacio ;
        vacio :=posicion ;
End ;
Procedure InicializarListas (variable inicio :integer {puntero}) ;
Begin
        inicio :=0 ; {inicio :=NIL en punteros}
End ;
{-------------------------------------------------------------------------------------}
Procedure Iniciar (variable vacio :integer ;variable a :arr) ;
Var
        i :integer ;
Begin
        vacio :=1 ;
        For i :=1 to max-1 do
                a[i].sig :=i+1 ;
        a[max].sig :=0 ;
End ;
{-------------------------------------------------------------------------------------}
```

- Insertar de forma ordenada :

Para borrar necesitamos conocer el anterior y la posicion : ant y a[ant].sig

```
Procedure Consultar (var encontrado :boolean ; variable ant, posic :integer ;elem :Tipoelemento ;a :arr ) ;
Var
        salir :boolean ;
Begin
        ant :=0 ;
        salir :=false ;
        posic :=inicio ;
        While (posic<>0) AND (NOT salir) do
        If a [posic].elem<elem then
        begin
                ant :=posic ;
                posic :=a[posic].sig ;
        end
        Else
                Salir :=true ;
        If (posic<>0) AND (a[posic].elem=elem) then
                encontrado :=true
        else
                encontrado :=false ;
End ;
```

## PILAS.

Son un caso especial de listas, en las que la inserción y borrado de nuevos elementos ha de hacerse por un mismo lugar (cima de la pila). Se dice que son estructuras LIFO (Last In, First Out). Para manipularlas hay que crear los procedimientos : MeterDatos e InicializarPila.

```
Type
        puntero=^nodo ;
        nodo=RECORD
                elemento : Tipoelemento ;
                sig :puntero ;
               end ;
Var
        cima :puntero ; {apunta a la cima de la pila}
        elemento :Tipoelemento ;



Procedure InicializarPila(var cima :puntero) ;
Begin
        cima :=NIL ;    {no NIL porque no hay que crearla aquí}
End ;
{-------------------------------------------------------------------------------------}
Function Vacia (cima :puntero) :boolean ;
Begin
        vacia :=cima=NIL ;
end ;
{-------------------------------------------------------------------------------------}
Procedure MeterDatos(var cima :puntero ; elemento :Tipoelemento) ;
Var
        auxi :puntero ;
Begin
        New(auxi) ;
        auxi^.elemento :=elemento ;
        auxi^.sig :=cima ;
        cima :=auxi ;
end ;
{-------------------------------------------------------------------------------------}
Procedure Sacar (variable cima :puntero ;variable elemento :Tipoelemento) ;
Variable
        auxi :puntero ;
Begin
        auxi :=cima ;
        elemento :=cima^.elemento ; { para mostrar si quiero}
        cima :=cima^.sig ;
        Dispose(auxi) ;
end ;
```

■ Simulación de Pilas con Arrays :

```
Type
        arr=array[1..Max] of Tipoelemento ; {Ya está la pila en array}
Variable
```

```
        cima :integer ;
        a :arr ;
{-------------------------------------------------------------------------------}
Procedure Inicializar (variable cima :integer) ;
Begin
        cima :=0 ;
end ;
{-------------------------------------------------------------------------------}
Function Vacia (cima :integer) :boolean ;
Begin
        vacia :=cima=0;
end ;
{-------------------------------------------------------------------------------}
Procedure Poner (var cima :integer ;elemento :Tipoelemento ; var a :arr) ;
Variable
        auxi :integer ;
Begin
        auxi :=cima+1;
        a[auxi].elem :=elemento ;
        cima :=cima+1 ;
end ;



Procedure Sacar (variable cima :integer ;variable elemento :Tipoelemento ;a :arr) ;
Begin
        elemento :=a[cima].elemento ;
        cima :=cima-1 ;
end ;
{-------------------------------------------------------------------------------}
Function Llena (cima :integer) :boolean ;
Begin
        Llena :=cima=max
end ;
{-------------------------------------------------------------------------------}
```

Se puede crear el regirtro con array y cima, para indicar que todo forma la pila.

## COLAS.

Son un caso especial de listas, donde la adición de nuevos elementos se hace por el final de la estructura, y la eliminación por el frente de la estructura. Se las conod¡ce como estructuras FIFO. Se pueden implementar con arrays o simularlas con arrays.

- Reestructuración : cuando llega al tope, se desplaza todo a la izquierda.
- Retroceso : ir desplazando todo a la izquierda.

Lo mejor es implementarlo con arrays circulares (similares a las listas circulares).

```
Si final=NULO entonces
        frente←auxi
si no
        final^.sig←auxi
fin si
final←auxi
```

```
Procedure Inicialización (variable frente, final :puntero) ;
Begin
        frente :=NIL ;
        final :=NIL ;
end ;
{-----------------------------------------------------------------------------------------}
Procedure Meter (variable final, frente :puntero ;elemento :Tipoelemento) ;
Variable
        auxi :puntero ;
Begin
        New(auxi) ;
        auxi^.elem :=elem ;
        auxi^.sig :=NIL ;
        If final=NIL then
                frente :=auxi
        Else
        Else
                final^.sig :=auxi ;
        final :=auxi ;
end ;
{-----------------------------------------------------------------------------------------}
Procedure Sacar (.....                                          .....) ;{mientras no esté vacía}
Variable auxi :puntero
Begin
        auxi :=frente ;
        elemento :=frente^.elemento ;
        frente :=frente^.sig ;
        If frente=NIL then
                final :=NIL ;
        Dispose(auxi) ;
end ;
{-----------------------------------------------------------------------------------------}
```

■ Simulación con Arrays (circulares) :

Ahora sig no es necesario. Tiene que haber elemento cabecera que marque principio y fin.

```
Procedure Inicializar (variable frente,final :integer) ;
Begin
```

```
        frente :=1 ;      {1 va a ser el elemento cabecera}
        final :=1 ;
end ;
```

Frente apunta siempre a cabecera (vacío), nunca al 1º,y la nueva cabecera aumenta. { se sacan datos y luego aumenta frente}

```
Function Vacia (frente,final :integer) ;
Begin
        Vacia :=frente=final ;
end ;
```

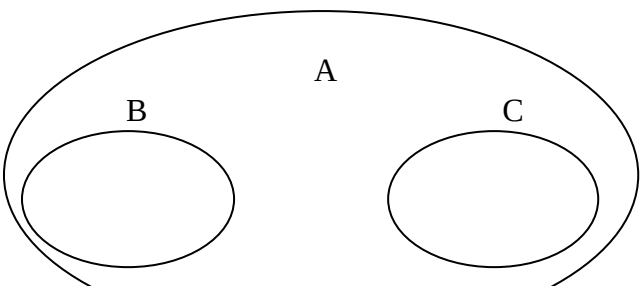
{Cabecera puede ser 1, 3, 5,...}

En Meter :
final←final MOD Max+1 {para que de la vuelta al llegar al final a Max}. Si final^.sig=frente, está llena.

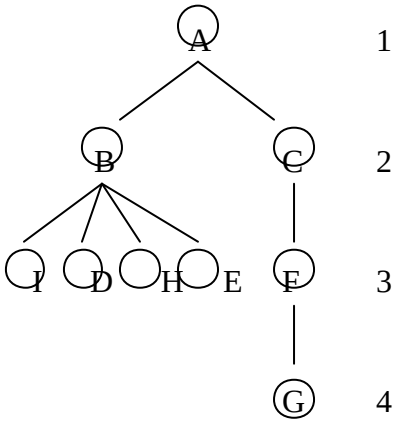
```
Function Llena (final, frente :integer) ;
Begin
        Llena :=final MOD Max+1=frente ;
end ;

Procedure Meter (var final :integer ;elem :Tipoelemento) ;
Begin
        final :=final MOD Max+1 ;
        a[final] :=elem ;
End ;

Procedure Sacar (var frente :integer ;elem :Tipoelemento) ;
Begin
        elemento :=a[frente MOD Max+1] ;
        frente :=frente MOD Max+1 ;
End ;
```

## ÁRBOLES.

Estructura de datos no lineal caracterizada porque en ella cada elemento, excepto la raiz, tiene un único elemento predecesor, y cada elemento puede tener varios elementos sucesores o siguientes.
A los elementos de un árbol se les llama NODOS. La definición de árbol se puede hacer como : un árbol está formado por la raiz y un conjunto disjunto de subárboles. Se pueden reperesentar como conjuntos.
Los recorridos de un árbol se hacen recursivamente.




Y mediante grafos :




- Grado de un Nodo : Es el número de hijos o subárboles que descienden de él. Un Nodo de grado 0 (el de más abajo) se llama HOJA.
- Nivel de un Nodo : Es el número de elementos que hay que recorrer para llegar desde la raíz sabiendo que la raíz está en nivel 1.
- Altura de un árbol : Es el máximo de los niveles de sus Nodos. Los árboles pueden tener un número indeterminado de hijos (árboles generadores).
- Arbol Binario : Es aquel en que cada Nodo tiene como máximo grado 2.
  Un arbol binario está Completo cuando sus Nodos son de grado 2 o son Hojas.
  Un arbol binario está Lleno cuando está completo y además todas sus ramas tienen la misma altura.
  Los árboles binarios se suelen emplear para representar expresiones aritméticas.
- Arbol Equilibrado : Es aquel en el que las alturas de sus ramas se diferencian en 1 como máximo.

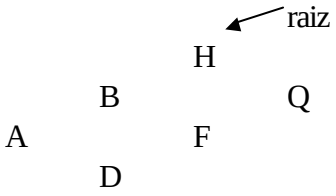
Recorridos de un Árbol :

Los recorridos de un árbol se hacen recursivamente. Estos recorridos pueden ser :

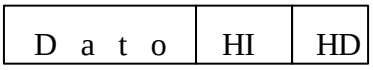
- In Orden :
  Subárbol Izquierdo - Raiz - Subárbol Derecho
- Pre Orden : (Notación Polaca Infija)
  Raiz - Subárbol Izquierdo - Subárbol Derecho
- Post Orden : (Notación Polaca PostFija)
  Subárbol Izquierdo - Subarbol Derecho - Raiz

Árbol Binario de Búsqueda :

Un árbol binario de búsqueda es aquel en el que cada elemento es superior (en valor) a los de su subárbol izquierdo, e inferior a los del subárbol derecho. Por ejemplo : H-B-Q-A-F-D




La ventaja es que se busca por su lista (efectividad como búsqueda binaria).
Los registros que se emplean tienen el siguiente formato :




```
Type
        puntero=^nodo ;
        nodo=RECORD
                elemento :Tipoelemento ;
                izqdo,dcho :puntero ;
                end ;
Var
        elemento :Tipoelemento ;
        raiz :puntero ;

Procedure Inicializar (variable raiz :puntero) ;
Begin
        raiz :=NIL ;
end ;

Function ArbolVacio(raiz :puntero) :boolean ;
Begin
        ArbolVacio :=raiz=NIL ;
end ;

Procedure Buscar (raiz :puntero :elemento :Tipoelemento ;variable ant, act :puntero) ;
Variable
        encontrado :boolean ; { indica si está o no el dato}
Begin
        act :=raiz ;
        ant :=NIL ;
        encontrado :=false ;
        While (act<>NIL) AND (NOT encontrado) do
                If act^.elemento=elemento then
                        encontrado :=true
                Else
                Begin
                        and :=act ;
                        If act^.elemento > elemento then
                                act :=act^.izqdo
                        Else
```

```
                        act :=act^.dcho ;
                end ;
end ;

Procedure Altas (variable raiz :puntero ; elemento :Tipoelemento) ;
Var
        ant,act :puntero ;
        auxi :puntero ;
Begin
        Buscar (raiz,elemento,ant,act) ;
        If act<>NIL then
                WriteLn('Ya existe') ;
        Else {alta}
        Begin
                New (auxi) ;
                auxi^.izqdo :=NIL ;
                auxi^.dcho :=NIL ;
                auxi^.elemento :=elemento ;
                If ant=NIL then
                        raiz :=auxi
                Else
                        If ant^.elemento > elemento then
                                ant^.izqdo :=auxi
                        Else
                                ant^.dcho :=auxi ;
        end ;
end ;

Procedure Bajas (variable raiz :puntero ; elemento :Tipoelemento) ;
Var
        auxi, ant,act :puntero ;
Begin
        Buscar (raiz, elemento, ant, act) ;
        If act=NIL then
                WriteLn('No está')
        Else
        Begin
                If (act^.izq=NIL) AND (act^.dcho=NIL) then
                        If ant=NIL then
                                raiz :=NIL
                        Else
                                If  ant^.izqdo=act then
                                        ant^.izqdo :=NIL
                                Else
                                        ant^.dcho :=NIL
                Else {no es hoja}
                        If (act^.dcho <>NIL) then
                                If ant=NIL then
```

```
                                raiz :=act^.dcho
                        Else
                                If ant^.izqdo= act then
                                        ant^.izqdo :=act^.dcho
                                Else
                                        ant^.dcho :=act^.dcho
                Else
                        If (act^.izqdo<>NIL) AND (act^.dcho=NIL) then
                                If ant=NIL then
                                        raiz :=act^.izqdo
                                Else
                                        If ant^.izqdo=act then
                                                ant^.izqdo :=act^.izqdo
                                        Else
                                                ant^.dcho :=act^.izqdo
                        Else {existe, no es hoja, tiene 2 hijos}
                                {If (act^.izqdo<>NIL) AND (act^.dcho<>NIL) then}
                        Begin
                                ant :=act ;
                                auxi :=act^.izqdo ; {para buscar su inmediato inferior}
                                While auxi^.dcho<>NIL do
                                Begin
                                        ant :=auxi ;
                                        auxi :=auxi^.dcho ;
                                end ;
                                act^.elemento :=auxi^.elemento ;
                                If ant=act then
                                        ant^.izqdo :=auxi^.izqdo ;
                                Else
                                        ant^.dcho :=auxi^.izqdo ;
                                act :=auxi ;
                        end ;
                Dispose(act) ;
        end ;
End ;

Procedure Recorrer (raiz :puntero) ;
Begin {In Orden}
        If raiz<>NIL then {condición de salida de la recursión}
        Begin
                Recorrer (raiz^.izqdo) ;
                WriteLn(raiz^.clave) ;
                Recorrer (raiz^.dcho) ;
        End ;
End ;
```

- Implementación de árboles con arrays :

| raiz | elemento | izqdo | dcho | vacio |
|---|---|---|---|---|
| 0 | | | 2 | 1 |
| | | | 3 | |
| | | | 4 | |
| | | | 5 | |
| | | | 6 | |
| | | | 7 | |
| | | | 8 | |
| | | | 0 | |

```
Const Max=100 ;
Type
        Reg=RECORD
                elemento :Tipoelemento ;
                izqdo :integer ;
                dcho :integer ;
                end ;
        arr =ARRAY [1..Max] of Reg ;
Var
        a :arr ;
        raiz :integer ;
        vacio :integer ; {para libres}



Procedure Iniciar (var a :arr ;var vacio :integer) ;
Var     i :integer ;
Begin
        vacio :=1 ;
        For i :=1 to (Max-1) do
                a[i].dcho :=i+1 ;
        a[Max].dcho :=0 ;
end ;

Procedure Inicializar (var raiz :integer) ;
Begin
        raiz :=0 ;
end ;

Procedure Buscar (raiz :integer ;elemento :Tipoelemento ;var ant,act :integer ; vacio :integer ;a :arr) ;
Var     encontrado :boolean ;
Begin
        ant :=0 ;
        act :=raiz ;
        encontrado :=false ;
        While (NOT encontrado) AND (act<>0) do
                If a[act].elemento=elemento then
```

```
                encontrado :=true
        Else
                If a[act].elemento > elemento then
                begin
                        ant :=act ;
                        act :=a[act].izqdo
                end
                Else
                begin
                        ant :=act ;
                        act :=a[act].dcho
                end ;
End ;

Procedure Altas (variable raiz :integer ; elemento :Tipoelemento ;variable a :arr ; variable vacio :integer) ;
Variable
        ant,act :integer ;
        auxi :integer ;
Begin
        Buscar(raiz,elemento,ant,act,vacio,a) ;
        If act<>0 then
                WriteLn('Ya está el dato')
        Else
        begin
                Reservar (auxi, a, vacio) ;
                a[auxi].elemento :=elemento ;
                a[auxi].dcho :=0 ;
                a[auxi].izqdo :=0 ;
                If ant=0 then
                        raiz :=auxi
                Else
                        If a[ant].elemento > elemento then
                                a[ant].izqdo :=auxi
                        Else
                                a[ant].dcho :=auxi ;
                end ;
End ;
Procedure Reservar (var auxi :integer ; var a :arr ; var vacio :integer) ;
Begin
        auxi :=vacio ;
        vacio := a[vacio].dcho ;
End ;

Procedure Bajas (var raiz :integer ;elemento :Tipoelemento ;var vacio :integer ; var a :arr) ;
Variable ant, act,aauxi :integer ;
Begin
        Buscar (raiz, elemento, ant,act,vacio,a) ;
        If act=0 then
```

```
        WriteLn('No existe')
Else
begin
        If  (a[act].dcho=0) AND (a[act].izqdo=0) then
                If ant=0 then
                        raiz :=0
                Else
                        If a[ant].izqdo=act then
                                a[ant].izqdo :=0
                        Else
                                a[ant].dcho :=0
        Else
                If (a[act].izqdo<>0) AND (a[act].dcho=0) then
                        If ant=0 then
                                raiz :=a[act].izqdo
                        Else
                                If a[ant].izqdo =act then
                                        a[ant].izqdo :=a[act].izqdo
                                Else
                                        a[ant].dcho :=a[act].izqdo
                Else
                        If (a[act].izqdo=0) AND (a[act].dcho<>0) then
                                If ant=0 then
                                        raiz :=a[act].dcho
                                Else
                                        If a[ant].izqdo= act then
                                                a[ant].izqdo :=a[act].dcho
                                        Else
                                                a[ant].dcho :=a[act].dcho
                        Else
                                If (a[act].izqdo<>0) AND (a[act].dcho<>0) then
                                begin
                                        ant :=act ;
                                        auxi :=a[act].izqdo ;
                                        While a[auxi].dcho<>0 do
                                        begin
                                                ant :=auxi ;
                                                auxi :=a[auxi].dcho ;
                                        end ;
                                        a[act].elemento :=a[auxi].elemento ;
                                        If ant=act then
                                                a[ant].izqdo :=a[auxi].izqdo
                                        Else
                                                a[ant].dcho :=a[auxi].izqdo ;
                                        act :=auxi ;
                                end ;
End ;
Liberar (act, a, vacio) ;
```

```
        End ;
End ;

Procedure Liberar (act :integer ;variable a :arr ; variable vacio :integer) ;
Begin
        a[act].dcho :=vacio ;
        vacio :=act ;
End ;
```

## UNIDADES.

Una unidad es un conjunto de constantes, variables, tipos de datos, procedimientos, funciones y objetos que se compilan independientemente del programa principal. TurboPascal tiene una serie de unidades estándar entre las cuales destacan :

Crt : Permite ejercer control sobre el teclado y la pantalla.

Dos : Proporciona acceso a las funciones del DOS.

Se usa con GetDate, GetTime, ..., y tiene un procedimiento, Exec, que permite ejecutar otros programas.

Para ello:

1. Hay que establecer $M :tamaño de pila, liberar memoria para libre.
2. Almacenar antes los vectores de interrupción (Swap Vectors) :

```
{$M  9827,0,0}
Begin
        ---------- ;
        ---------- ;
        ---------- ;
        SwapVectors ;
        Exec('c :\dos\command.com','/c  dir') ;
        SwapVectors ;
        ---------- ;
        ---------- ;
end.
```

La instrucción Exec('GetEnv(comspec)', busca el command.com.

Printer : Da acceso a la impresora.

```
Begin
    WriteLn(LST,'Cadena de texto') ;{ Printer asocia la impresora a la variable LST}
```

Para Pascal la impresora es un archivo de texto, y se maneja como tal : (sin usar Printer)

```
Var
        impresora :text ;
Begin
        Assign(impresora,'LPT1') ;
        Rewrite(impresora) ;
```

```
        WriteLn(impresora,'cadena de texto') ;
        Close(impresora) ;
    End.
```

Strings :Permite manejar cadenas terminadas en nulo (arrays con base en 0) :

```
    Type
        cadena=array(0..n) of char ;
```

Estas cadenas terminan con NULL (carácter nulo), lo que tiene la ventaja de no tener un tamaño tope (sólo el de la gestión DOS). Se manipula por un puntero llamado PCHAR, y estas cadenas necesitan de la unidad Strings y de la directiva {$X+}.

Overlays : Proporciona la gestión de recubrimientos, que consisten en que varios procedimientos ocupan simultáneamente las mismas posiciones de memoria, con lo cual los programas pueden ser mayores que la memoria disponible. Necesitamos la unidad Overlays y las directivas {$F+} (Far Calls) y {$O+}.{$O nombre procedimientos afectados}
Para inicializar la unidad Overlays : OVRinit(nombreproced.OVR)

System : No es preciso ponerla ya que se usa por defecto.

Graph : Proporciona procedimientos y funciones para manipular gráficos.

Nosotros podemos crear nuestras propias unidades, que nos servirán para :

- Crear una librería de procedimientos y funciones.
- Dividir un programa.

Las unidades permiten que los programas superen la barrera de los 64KB (al estar divididos) y permiten que diversos programas utilicen los mismos módulos/unidades.

Creación de unidades.

Para crear una unidad se comienza :

```
    Unit Nombre ;
```

Donde Nombre será el nombre que emplearemos para grabar la unidad en memoria.
Después comienza la parte de INTERFACE (parte de declaración de objetos exprtados, es decir, aquellos que son visibles desde los programas que los van a llamar) que terminará cuando comience la parte de implementación :

```
    INTERFACE
        Cabeceras de constantes, variables,tipos, procedimientos y funciones.(objetos)
```

Tras esto empieza la parte de IMPLEMENTATION (parte de declaración de objetos privados, para uso exclusivo de la unidad, no se necesita poner la lista de parámetros formales), que terminará con Begin o con End.

```
    IMPLEMENTATION
        Constantes, variables,tipos, procedimientos y funciones.(objetos)
    BEGIN / END
```

Con Begin : Sección de inicialización (optativo). Esta sección se utiliza para inicializar variables necesitadas por unidades o bien para la apertura de ficheros. Es frecuente que no la haya. Lo que se encuentra en esta sección se ejecuta antes de cualquier otra cosa que haya.
Con End : Indica que sólo son objetos privados.

```
    Unit Ejemplo ;
```

```
INTERFACE
    Procedure Intercambiar(var x, y :integer) ;
    Function Mayor(a, b :integer) ;
IMPLEMENTATION
    Procedure Intercambiar;
    Var aux :integer ;
    Begin
        aux :=x ;
        x :=y ;
        y :=aux ;
    end ;
    Function Mayor;
    Begin
        If a>b then
            Mayor :=a
        Else
            Mayor :=b ;
        end ;
    {Sección de inicialización si la hubiese}
    END.
```

Una unidad puede hacer uso de otras unidades (en interface o implementation). Al terminar una unidad se compila primero en memoria y luego en disco.

1. Primero se guarda como EJEMPLO.PAS
2. Se compila (para detectar errores)
3. Se guarda sin errores.
4. Se compila en disco (si fuera un Programa se originaría un .EXE, pero al ser una unidad se origina un .TPU).

Para usar esta unidad :

```
Program xxxxxxxx ;
Uses Ejemplo ;
```

Las unidades (archivos con extensión .TPU) primero se buscan en Tpl ( librería que se carga automáticamente al llamar a Turbo.tpl) que contiene todas las unidades estándar. Para añadir o quitar unidades a Tpl empleamos Tpumover.
Después de buscar en Turbo.tpl, se busca en el subdirectorio actual, y si no lo encuentra, lo busca en donde le indique la sección directorio (Opciones / Directorios).
Es posible utilizar más de una unidad en un programa, tanto si son predefinidas como definidas por el usuario :
Uses Crt, Dos, Ejemplo ;
Si hay procedimientos/funciones con el mismo nombre en distintas unidades, para identificarlos emplearemos la notación unidad.nombre_procedimiento/función.

```
Program Amigos ;
Var i, j,m ;
{$I Sumadiv.pas}
Function SonAmigos(i,j :integer) :boolean ;
Begin
```

```
        SonAmigos :=(SumaDivisores(i)=j) AND (SumaDivisores(j)=i)
end ;

Function SumaDivisores(m :integer) :integer ;
Var
        i :integer ;
        suma :integer ;
Begin
        suma :=0 ;
        For i :=(m DIV 2) downto 1 do
                If m MOD i = 0 then
                        Suma :=Suma+i ;
        SumaDivisores :=suma ;
end ;
Begin
        m :=500 ;
        For i :=1 to m-1 do
                For j :=i+1 to m do
                        If SonAmigos(i,j) then WriteLn(i,j)
end.
```

## FASES EN LA CONSTRUCCIÓN DE PROGRAMAS.

La construcción de un programa atraviesa las siguientes fases sucesivas :

1. Análisis del programa.
2. Diseño del algoritmo (siguiendo las reglas de la programación modular).
3. Codificación (escribir el programa en un lenguaje de programación).
4. Depuración (eliminación de errores).
5. Pruebas (funcionamiento del programa) e Integración (por si es muy grande).
6. Verificación (aplicar pruebas matemáticas para controlar el mayor número de casos posibles).
7. Mantenimiento (dotar al programa de documentación interna y externa).

El uso de Unidades y de Programación Orientada a Objetos (POO) permite reutilizar el código.

## PROGRAMACIÓN ORIENTADA A OBJETOS (POO).

En Pascal, los Objetos son perecidos a los registros, con la diferencia de que los Objetos además de incluir datos, incluyen rutinas para la manipulación de dichos datos.
Para trabajar con objetos deberíamos comenzar diseñando las diversas estructuras entresacando de todas ellas las características comunes e implementando un tipo objeto genérico del cual descendieran los demás.

Declaración de objetos :

```
        Type
                Nombre=OBJECT
                                dato1 :tipo1 ;
                                dato2 :tipo2 ;
                                procedure procedimiento1 ;
```

```
                    Procedure procedimiento2[(parámetros)] :
            end ;
```

Dentor del tipo, se incluyen cabeceras de procedimientos y funciones. La implementación de estos procedimientos y funciones dentro del objeto se efectúa de forma independiente, y a estos procedimientos y funciones dentro del objeto se les denomina Métodos.

Declaración de métodos :

```
Procedure Nombre.procedimiento1 ;
Begin
        ------------------ ;
        ------------------ ;
        ------------------ ;
End ;

Procedure Nombre.procedimiento2[(lista parámetros formales)] ;
Begin
        ------------------ ;
        ------------------ ;
        ------------------ ;
End ;
```

Los datos de un objeto son siempre accesibles para los métodos a pesar de que estén declarados en lugares diferentes (de forma independiente) .Ejemplo :

```
Type
        Persona=OBJECT
                        nombre :string ;
                        edad :integer ;
                        domicilio :string ;
                        procedure Inicializar(n :string ;de :integer ;d :string) ;
                end ;

Procedure Persona.Inicializar(n :string ;de :integer ;d :string) ;
Begin
        nombre :=n ;
        edad := ed ;
        domicilio :=d ;
End ;

{Para trabajar con objetos, necesitamos variables de tipo objeto llamadas Instancias del objeto.}
{A los diferentes tipos se les denomina Clases}

Var
        p :Persona ;
        n,d :string :
        ed :integer ;
```

```
BEGIN
        WriteLn('Nombre :') ;
        ReadLn(n) ;
        WriteLn( 'Edad :') ;
        ReadLn(ed) ;
        WriteLn('Domicilio :') ;
        ReadLn(d) ;
        p.inicializar(n,ed,d) ;
END.
```

Los objetos pueden heredar campos y métodos de otros objetos. Cuando esto sucede lo declararemos de la siguiente manera :

```
Type
        Persona=OBJECT
                        nombre :string ;
                        edad :integer ;
                        domicilio :string ;
                        procedure Inicializar(n :string ;de :integer ;d :string) ;
                end ;

        Alumno=OBJECT (Persona) { Hereda Persona}
                        nota :real ;
                        Procedure Inicializar(n :string ; ed :integer ; d :string ; no :real) ;
                end ;

Procedure Persona.Inicializar.....
Begin
        ..............
End ;

Procedure Alumno.Inicialzar(n :string ; ed :integer ; d :string ; no :real) ;
Begin
        nombre :=n ;
        edad :=ed ;
        domicilio :=d ;
        nota :=no ;
End ;

Var a :Alumno ;
```

El objeto descendiente hereda los campos y métodos del objeto padre, y puede tener más campos y métodos nuevos y también puede redefinirlos.

Todos los métodos vistos hasta ahora eran ESTÁTICOS (las referencias se resuelven en tiempo de compilación), pero los métodos también pueden ser VIRTUALES (las referencias se resuelven en tiempo de ejecución). Cuando el compilador detecta la compilación de un método virtual, crea una Tabla de Métodos Virtuales. Las

instancias (variables) declaradas de un tipo objeto con métodos virtuales (tipo método virtual), necesitan inicializarse a través de unCONSTRUCTOR, que enlaza la instancia con la Tabla de Métodos Virtuales.
Para eliminar la conexión se emplean DESTRUCTORES (sólo cuando se trabaja con estructuras dinámicas(punteros)), necesarios siempre que usamos métodos virtuales.
Para que un método sea un constructor, habrá que sustituir la palabra 'procedure' por 'constructor'.
Para que un método sea un destructor, habrá que sustituir la palabra 'procedure' por 'destructor'.
Un método es virtual cuando su cabecera acaba en 'virtual ;'.
Los métodos virtuales son el instrumento adecuado para el POLIMORFISMO. Un método es polimórfico cuando se ejecuta de diferente forma segús el objeto desde el cual se le llame.

## CARACTERÍSTICAS DE LA PROGRAMACIÓN ORIENTADA A OBJETOS.

- Encapsulación.
- Herencia.
- Polimorfismo.

La Encapsulación consiste en que los objetos agrupan tanto los datos como los procedimientos necesarios para su manipulación. No se debe acceder a los datos de un objeto directamente, sino que siempre se debe hacer a través de los métodos correspondientes. (Ejemplo : En lugar de referenciar area, referenciaríamos figura.area, siendo figura el objeto).
Cuando trabajamos con unidades es posible declarar tipos de objeto en dichas unidades. Si se desea que el usuario de una unidad (que use objetos) no pueda acceder a los campos o a determinados métodos del objeto, habrá que declararlos como privados, para lo cual la declaración de objeto se hará en la sección de Interface, pero los métodos tendrán que ser implementados en la sección de Implementation.Ejemplo :

```
Unit <Nombre_Unidad> ; {De 8 caracteres máximo}

INTERFACE
        Type
                Persona=OBJECT
                                nombre :string ;
                                edad :integer ;
                                domicilio :string ;
                                Procedure Inicializar(n :string ; de :integer ; d :string) ;
                        end ;
                Alumno=OBJECT (Persona)
                                nota :real ;
                                procedure Inicializar(n :string ; de :integer ; d :string ;no :real) ;
                        end ;
IMPLEMENTATION
        Procedure persona.Inicializar(--------------------) ;
        Begin
                ------------------ ;
                ------------------ ;
                ------------------ ;
        End ;

        Procedure alumno.Inicializar(--------------------) ;
```

```
Begin
        ------------------ ;
        ------------------ ;
        ------------------ ;
End ;
```

Para declarar las partes privadas :

1. Declaramos las partes PUBLICAS : 1º datos y luego métodos.
2. Declaramos las partes PRIVADAS poniendo ántes 'PRIVATE' :1º datos y luego métodos.

Ejemplo :

```
Type Persona=OBJECT
                domicilio :string ;                                  {Partes}
                procedure Inicializar (-------------------------) ; {Públicas}
                PRIVATE
                        nombre :string ; {Partes}
                        edad ;integer ;   {Privadas}
             end ;
```

En un tipo objeto la declaración de los métodos debe ir después de los datos (excepto cuandoy parte privada).
Es posible utilizar punteros con los objetos de la forma habitual :

```
Type
        puntero=^Persona ;
        Persona=OBJECT
                ------------- ;
                ------------- ;
                end ;
Variable
        p :puntero ;
Begin
        New(p) ;
        p^.Inicializar('Pedro',5,'Goya') ;
        ------------------- ;
        ------------------- ;
        ------------------- ;
        Dispose(p) ;
end ;
```

```
Type
        figura=OBJECT
                nombre :string ;
                area :real ;
                perimetro :real ;
                Constructor Inicializar ;{Para llamar a virtuales}
                Procedure CalcularArea ;VIRTUAL ;
                Procedure CalcularPerimetro ;VIRTUAL ;
                Procedure Visualizar ;
```

```
                end ;
        Rectangulo=OBJECT (figura)
                        base :real ;
                        altura :real ;
                        {Si se redefine virtual se amntiene la cabecera}
                        Constructor Inicializar(d,a :real) ;
                        Procedure CalcularArea ;VIRTUAL ;
                        Procedure CalcularPerimetro ;VIRTUAL ;
                    end ;
        Circulo=OBJECT (figura)
                        radio :real ;
                        Constructor Inicializar(r :real) ;
                        Procedure CalcularArea ;VIRTUAL ;
                        Procedure CalcularPerimetro ;VIRTUAL ;
                    end ;
Constructor figura.Inicializar ;
Begin
        WriteLn('Indique la figura :') ;
        ReadLn(nombre) ;
End ;
Procedure figura.CalcularArea ;
Begin
        area :=0 ;
End ;

Procedure figura.CalcularPerimetro ;
Begin
        perimetro :=0 ;
End ;

Procedure figura.Visualizar ;
Begin
        CalcularArea ;
        CalcularPerimetro ;
        WriteLn('La figura es',nombre) ;
        WriteLn('El area es',area) ;
        WriteLn('El perimetro es',perimetro) ;
end ;

Constructor Rectangulo.Inicializar(b,a :real) ;
Begin
        base :=b ;
        altura :=a ;
        nombre :='Rectangulo' ;
End ;

Procedure Rectangulo.CalcularArea ;
Begin
```

```
        area :=base*altura ;
End ;

Procedure Rectangulo.CalcularPerimetro ;
Begin
        perimetro:=2*(base+altura) ;
End ;

Constructor Circulo.Inicializar(r:real) ;
Begin
        radio :=r ;
        nombre :='Circulo' ;
End ;

Procedure Circulo.CalcularArea ;
Begin
        area :=3.141592*radio*radio;
End ;

Procedure Circulo.CalcularPerimetro ;
Begin
        perimetro:=2*3.141592*radio;
End ;

Var
        f :figura ;
        c :Circulo ;
        rec :Rectangulo ;
        a,b :real ;
        r :real ;

Begin
        WriteLn('Deme Radio') ;
        ReadLn(r) ;
        c.Inicializar(r) ;
        c.Visualizar ; {Es polimórfico:se comporta de una manera o de otra según desde donde se le llame
                        Sin virtual, nos pasa en Visualizar 0,0 (lo que había en compilación).Con virtual pasa lo que
                        hay en ejecución).}
        WriteLn('Deme Base y Altura :') ;
        ReadLn(b,a) ;
        rec.Inicializar(r) ;
        rec.visualizar ;
```

Con punteros :

```
Type
        rt=^Rectangulo ;
        cr=^Circulo ;
Var
        c :cr ;
        rec :rt ;
        a,b,r :real ;
Begin
        WriteLn..... ;
        ReadLn(r) ;
        New(c) ;
        c^.Inicializar(r) ;
        c^.Visualizar ;
        WriteLn..... ;
        New(rec) ;
        rec^.Inicializar(b,a) ;
        rec^.Visualizar ;
```