

Tipos de datos

Numérico

- **Tipo de dato entero:** Es un tipo de dato simple, ordinal. Toma los valores positivos y negativos, tienen una representación interna y esta representación tiene un número máximo y un número mínimo.

Las operaciones permitidas son: +, *, -, /, >, <, >=, <=, MOD y DIV.

- **Tipo de dato real:** Es un tipo de dato simple, pero **NO** ordinal. Toma los valores positivos o negativos, ya sean con decimales o no, tiene una representación interna, que permite un número máximo y un número mínimo.

Las operaciones permitidas son: +, -, *, /, <, >, =, >=, <=.

Lógico

Permite representar datos que pueden tomar dos valores verdadero o falso. Es un tipo de dato simple, ordinal. Los valores son de la forma:

True = verdadero

False = falso

Las operaciones permitidas son: AND (conjunción), OR (disyunción), NOT (negación).

Carácter

Representa un conjunto finito y ordenado de caracteres que la computadora reconoce. Un dato de tipo carácter contiene un solo carácter.

Es un tipo de dato simple, ordinal. Los valores son de la forma a, B, !, \$, L, 4

Las operaciones permitidas son: <, <=, >, >=, =, <>.

Strings

Es una colección de caracteres. Tiene como máximo 256 caracteres. En general se utilizan para representar nombres.

Es un tipo de dato compuesto. Los valores son de la forma casa / erml, todos los valores que sean un conjunto de caracteres.

Las operaciones permitidas son: < , > , = , => , = , <>.

Estructura de control

Secuencia


Consta en especificar un conjunto de instrucciones que se ejecutan una debajo de la otra. Cuando una termina se ejecuta la siguiente instrucción y así sucesivamente hasta terminar el programa.

```
Program uno;  
...  
var  
    num: integer;  
begin  
    read (num);  
    write (num);  
end.
```

Decisión

En un algoritmo representativo de un problema real es necesario tomar decisiones en función de los datos del problema.

```
if (condición) then  
    acción;
```


más de
una acción

```
if (condición) then  
    begin  
        acción 1;  
        acción 2;  
    end;
```

```
if (condición) then  
    acción 1  
else  
    acción 2;
```

```
if (condición) then  
    begin  
        acción 1;  
        acción 2;  
    end  
else  
    acción 3;
```

```
if (condición) then  
    begin  
        acción 1;  
        acción 2;  
    end  
else  
    begin  
        acción 3;  
        acción 4;  
    end;
```

Selección

Permite dado un código evaluar un conjunto de condiciones al mismo tiempo, esas condiciones deben ser disjuntas y la condición que me dé verdadera (solo una me va a dar verdadera) es la que se va a ejecutar esas acciones.

```
case (variable) of
  condición1: accion1;
  condición 2: acción2;
  ....
  condición n: acción n;
end;
```



más de una acción

```
case (variable) of
  condición1: accion1;
  condición 2: begin
    acción2;
    accion3;
  end;
  ....
  condición n: accion;
end;
```

Iteración

Se viene ejecutando un bloque de acciones en un momento se evalúa una condición, dependiendo del resultado de esa evaluación es que se ejecutan o no las acciones.

Estas estructuras se clasifican en **pre-condicionales** y **post-condicionales**

- **Pre-Condicionales:** Se venía ejecutando un bloque de acciones se evalúa la condición, si esa condición es verdadera se ejecutan todas las acciones que están dentro de la condición. Dicho bloque se puede ejecutar 0,1 o más veces.

```
while (condición) do
  accion;
```



más de una acción

```
while(condición) do
  begin
    acción 1;
    acción 2;
  end;
```

- **Post-Condicionales:** Primero ejecutan las acciones y después evalúan la condición, si la condición es falsa se vuelven a ejecutar las acciones. Dicho bloque se pueda ejecutar 1 o más veces.

```
repeat
  accion;
until (condición);
```



más de una acción

```
repeat
  acción 1;
  acción 2;
until (condicion);
```

Repetición

Es una extensión natural de la secuencia. Consiste en repetir N veces un bloque de acciones.

Este número de veces que se deben ejecutar las acciones es fijo y conocido de antemano

```
for indice := valor_inicial to valor_final do
    accion 1;
```



```
más de una      for indice := valor_inicial to valor_final do
acción          begin
                  accion 1;
                  accion 2;
                  end;
```

Máximos y Mínimos

Máximo

Las variables **máximo** deben inicializarse siempre en el valor más chico que se pueda tener dependiendo de la aplicación.

Utilizar una variable que representará al máximo.

Inicializar la variable antes de comenzar la lectura de los datos. El máximo en un valor bajo.

Actualizar la variable máximo cuando corresponda.

```
Program uno;
```

```
var
```

```
    prom: real; alu: integer; max:real; maxalu:integer;
```

```
begin
```

```
    read(prom);
```

```
    read(alu);
```

```
    max:= -1;
```

```
    while (prom <> 0) do begin
```

```
        if(prom >= max)then begin
```

```
            max:= prom;
```

```
            maxalu:= alu;
```

```
        end;
```

```

        read(prom);
        read(alu);

    end;

    write('El mejor alumno es:', maxalu);

end.

```

Mínimos

Las variables mínimo se inicializan en el valor más grande que podía tener la aplicación.

Utilizar una variable que representará al mínimo.

Inicializar la variable antes de comenzar la lectura de los datos. El mínimo en un valor alto.

Actualizar la variable mínimo cuando corresponda.

```

Program uno;

var

    prom: real; alu: integer; min:real;

begin

    read(prom);

    read(alu);

    min:= 11;

    while (prom <> 0) do begin

        if(prom <= min)then

            min:= prom;

            read(prom);

            read(alu);

        end;

        write('El mejor promedio es:', min);

    end.

```

TDU (tipos de datos definidos por el usuario)

Es aquel que no existe en el lenguaje, pero lo puede definir el programador y es el encargado de hacerlo.

Type

identificador = tipo;

Un tipo estándar

Un tipo definido por el lenguaje

program uno;

Const

...

Type

nuevotipo1 = tipo;

Módulos

Var

x: identificador;

...

Begin

...

End.

Ejemplo y Operación no permitida

Program nombre;

Const

N = 25;

pi = 3.14;

Type

nuevotipo1 = integer;

módulos {luego veremos como se declaran}

var

edad: integer;

peso: real;

valor: nuevotipo1;

begin

end.

La variable valor:

- puede tener asignado cualquier valor permitido para los enteros.
- Puede utilizar cualquier operación permitida para los enteros.
- NO puede relacionarse con ninguna otra variable que no sea de su mismo tipo

```

Program nombre;
Const
    N = 25;
    pi = 3.14;


Type
    nuevotipo1 = integer;

var
    edad: integer;
    peso: real;
    valor: nuevotipo1;

begin
    valor:= 8;  read (valor); if (valor MOD 3 = ...) then ...

    edad:= valor;  edad:= edad + valor
end.

```



Subrango

Como su nombre lo indica es un tipo de datos simple ordinal, que consiste en especificar una sucesión de valores contiguos de un tipo ordinal que ya está predefinido en el lenguaje tomado como base.

Es simple y ordinal. Existe en la mayoría de los lenguajes

Las operaciones permitidas son: Asignación, comparación y Todas las operaciones permitidas para el tipo de base.

Las operaciones que no están permitidas dependen del tipo base.

```

program uno;
Const
    ...
Type
    tipo1 = valor1..valor2;
var
    x,y: tipo1;

Begin
    ...
End.

```

Modularización

Procedimiento

Es un conjunto de instrucciones que realizan una tarea específica y que pueden devolver 0, 1 o más valores.

```
procedure nombre (parámetros);  
var  
    ....  
begin  
    ....  
end;
```

} Variables locales

} Código del procedimiento

Funciones

Es un conjunto de instrucciones que realizan una tarea específica, retorna un único valor y ese valor además debe de ser de tipo simple.

```
function nombre (parámetros): tipo;  
var  
    ....  
begin  
    ....  
end;
```

} Variables locales

} Código de la función

Invocación

INVOCACION POR SU NOMBRE

Invocación usando variable

El resultado se asigna
a una variable del
mismo tipo que
devuelve la función.

```
program uno;  
Function auxiliar: real;  
Var  
    x, y, cociente:real;  
  
begin  
    x:= 10;  
    y:= 4;  
    cociente:= x/y;  
    auxiliar:= cociente;  
end;  
Var  
    aux:real;  
begin  
    aux:= auxiliar;  
    write (aux);  
end.
```

El retorno de la
función es a la
misma línea de
invocación

INVOCACION POR SU NOMBRE

Invocación en un while o en un if

El resultado se utiliza
para evaluar la
condición.

```
program uno;  
Function auxiliar: real;  
Var  
    x, y, cociente:real;  
  
begin  
    x:= 10;  
    y:= 4;  
    cociente:= x/y;  
    auxiliar:= cociente;  
end;  
  
begin  
    while (auxiliar = 5.5) do  
  
        if (auxiliar = 5.5) then  
end.
```

El retorno de la
función es a la misma
línea de invocación

INVOCACION POR SU NOMBRE

Invocación en un write

El resultado se utiliza
para informar en la
sentencia write.

```
program uno;  
  
Function auxiliar: real;  
Var  
    x, y, cociente:real;  
  
begin  
    x:= 10;  
    y:= 4;  
    cociente:= x/y;  
    auxiliar:= cociente;  
end;  
  
begin  
    write ('El resultados es,auxiliar);  
end.
```

Parámetros

- **Parámetros por valor:** Un dato de entrada por valor es llamado parámetro IN y significa que el módulo recibe (sobre la variable local) un valor proveniente de otro módulo (o del programa principal). Con él puede realizar operaciones y/o cálculos, pero no producirá ningún cambio ni tampoco tendrá incidencia fuera de módulo.



```
Program porValor;
```

```
procedure uno (num: integer);  
Begin  
    if (num = 7) then  
        num:= num + 1;  
        write (num);  
end;  
var  
    x: integer;  
begin  
    x:= 7;  
    uno (x);  
end.
```

Dentro del procedimiento uno, el
parámetro num copia el valor
enviado por x (variable del
programa)

- **Parámetros por referencia:** La comunicación por referencia (OUT, INOUT) significa que el módulo recibe el nombre de una variable (referencia a una dirección) conocida en otros módulos del sistema. Puede operar con ella y su valor original dentro del módulo, y las modificaciones que se produzcan se reflejan en los demás módulos que conocen la variable.



```
procedure uno (var num: integer);  
Begin  
    if (num = ...) then  
        num:= num + 1;  
        write (num);  
    end;  
var  
    x: integer;  
begin  
    x:= 7;  
    uno (x);  
end.
```

Dentro del procedimiento uno,
el parámetro num comparte la
dirección de memoria con x
(variable del programa)

Registros

Es un tipo de datos estructurados compuesto, que permite agrupar diferentes clases de datos en una única estructura bajo un único nombre.

Es homogénea, estática y posee campos.

```
Program uno;  
Const  
    ...  
Type  
  
nombre = record  
    campo1: tipo;  
    campo2: tipo;  
    ...  
end;  
  
Var  
    variable: nombre;
```



Se nombra cada campo.
Se asigna un tipo a cada campo.
Los tipos de los campos deben ser estáticos.

Operación permitida

```
Program uno;  
Const  
...  
Type  
  
perro = record  
  raza: string;  
  nombre: string;  
  edad: integer;  
end;  
  
Var  
  ani1, ani2: perro;
```

Begin

...
ani2:= ani1;
...
End.



La única operación permitida es la asignación entre dos variables del mismo tipo

Campos

```
Program uno;  
Const  
...  
Type  
  
perro = record  
  raza: string;  
  nombre: string;  
  edad: integer;  
end;  
  
Var  
  ani1, ani2: perro;
```

Begin

...
Puedo realizar las operaciones permitidas según el tipo de campo del registro
...
End.



La única forma de acceder a los campos es `variable.nombrecampo`

ani1.nombre

Operaciones No permitidas

```
Program uno;  
Const  
...  
Type
```

```
perro = record  
  raza: string;  
  nombre: string;  
  edad: integer;  
end;
```

```
Var  
  ani1, ani2: perro;
```

```
Begin  
  ani1.raza:= 'Callejero';  
  ani1.nombre:= 'Bob';  
  ani1.edad:= 1;  
End.
```

```
Begin  
  read (ani1.raza);  
  read(ani1.nombre);  
  read(ani1.edad);  
End.
```



No se puede hacer
read (ani1)

```
Program uno;  
Const  
...  
Type
```

```
perro = record  
  raza: string;  
  nombre: string;  
  edad: integer;  
end;
```

```
Var  
  ani1, ani2: perro;
```

```
Begin  
  leer (ani1);  
  write (ani1.raza);  
  write(ani1.nombre);  
  write(ani1.edad);  
End.
```



No se puede hacer
write (ani1)

```
Program uno;  
Const
```

```
...  
Type
```

```
perro = record  
  raza: string;  
  nombre: string;  
  edad: integer;  
end;
```

```
Var  
  ani1, ani2: perro;
```

```
Begin  
  leer (ani1);  
  leer (ani2),
```

```
  if ((ani1.raza = ani2.raza)and  
      (ani1.nombre = ani2.nombre) and  
      (ani1.edad = ani2.edad))  
  then  
    write (`Los registro son iguales`);  
  End.
```



No se puede hacer
ani1 = ani2

Modularización con registros

```
Procedure leer (var p:perro);
```

```
Begin  
  read (p.raza);  
  read(p.nombre);  
  read(p.edad);  
End.
```

```
Program uno;  
Const
```

```
...  
Type  
perro = record  
  raza: string;  
  nombre: string;  
  edad: integer;  
end;
```

```
Procedure leer (var p:perro);  
begin  
  ...  
end;
```

```
Var  
  ani1, ani2: perro;
```

```
Begin  
  leer (ani1);  
  ani2:= ani1;  
End.
```

```
Procedure imprimir (p:perro);
```

```
Begin
```

```
  write (p.raza);
```

```
  write(p.nombre);
```

```
  write(p.edad);
```

```
End.
```

```
Program uno;  
Const  
  ...  
Type  
perro = record  
  raza: string;  
  nombre: string;  
  edad: integer;  
end;  
Procedure leer (var p:perro);  
begin  
  ...  
end;  
Procedure imprimir (p:perro);  
begin  
  ...  
end;  
Var  
  ani1, ani2: perro;  
Begin  
  leer (ani1);  
  imprimir(ani1);  
End.
```

Registros con registros

```
Program uno;  
Type  
fecha = record  
  dia: integer;  
  mes:integer;  
  año:integer;  
end;  
perro = record  
  raza: string;  
  edad: integer;  
  nombre: string;  
  fechaVis:fecha;  
end;
```

Módulo de lectura

```
procedure leer (var p:perro);
```

```
Begin
```

```
    read (p.raza);  
    read(p.nombre);  
    read(p.edad);  
    read(p.fechaVis.dia);  
    read(p.fechaVis.mes);  
    read(p.fechaVis.año);
```

```
end;
```

```
procedure leerFecha (var f:fecha);
```

```
Begin
```

```
    read(f.dia);  
    read(f.mes);  
    read(f.año);
```

```
end;
```

```
procedure leer (var p:perro);
```

```
var
```

```
    fec:fecha;
```

```
Begin
```

```
    read(p.raza);  
    read(p.nombre);  
    read(p.edad);  
    leerFecha (fec);  
    p.fechaVis:= fec;
```

```
end;
```

Otra opción

```
procedure leer (var p:perro);
```

```
begin
```

```
    read(p.raza);  
    read(p.nombre);  
    read(p.edad);  
    leerFecha (p.fechaVis);
```

```
end;
```

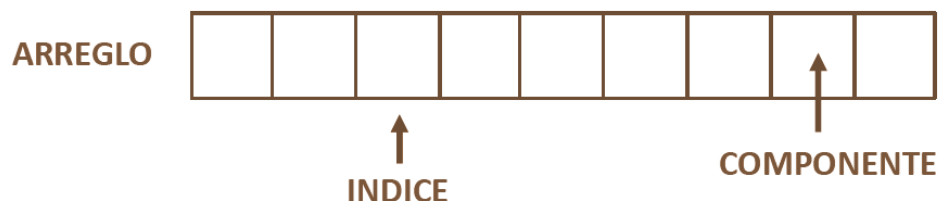
Arreglos

Es una estructura de datos compuesta que permite acceder a cada componente por medio de una variable índice.

Los elementos están almacenados en la memoria uno al lado de otro empezando desde una posición de memoria hasta el final del arreglo

Tenemos tres componentes:

- El arreglo entero.
- Tenemos que trabajar con un índice.
- Tener en cuenta los elementos de los arreglos.



Vectores

Es una colección de elementos que se guardan de forma consecutiva en la memoria y se pueden acceder a través de un índice.

Es homogénea, estática e indexada.

```
Program uno;  
Const
```

```
...  
Type
```

```
vector = array [rango] of tipo;
```

```
Var  
variable: vector;
```



El **rango** debe ser un tipo ordinal
char, entero, booleano, subrango

**Unos
ejemplos ...**



El **tipo** debe ser un tipo estático
char, entero, booleano, subrango, real
registro, vector

Operaciones permitidas y no permitidas

```
Program uno;  
Const  
...  
Type
```

```
vector = array [1..10] of integer;
```

```
Var  
v1,v2: vector;
```

```
Begin
```

```
...  
v2:= v1;  
...  
End.
```



**La única operación permitida
es la asignación entre dos
variables del mismo tipo**

```

Program uno;
Const
  ...
Type

  vector = array [1..10] of integer;

```

```

Var
  v1,v2: vector;

```

```

Begin

```

```

  ...

```

```

End.

```



La única forma de acceder a los elementos es utilizando un índice

variable [pos]

variable [4]

Las operaciones con el elemento son las permitidas para el tipo de datos del elemento

Operaciones

- Carga de valores
- Lectura / Escritura
- Recorridos
- Agregar elementos al final
- Insertar elementos
- Borrar elementos
- Búsqueda de un elemento
- Ordenación de los elementos

Carga de valores

```

Procedure carga (var v: vector);

```

```

  var
    i,valor:integer;

```

```

  begin
    for i:= 1 to tam do
      begin
        read (valor);
        v[i]:= valor;
      end;
    end;

```

?	?	?	?	?	?	?	?	?	?
1	2	3	4	5	6	7	8	9	10

-1	18	4	0	5	57	-2	3	8	4
1	2	3	4	5	6	7	8	9	10

ALTERNATIVA

```

Procedure carga (var v: vector);
  var
    i:integer;
  begin
    for i:= 1 to tam do
      read (v[i]);
    end;

```

Lectura / Escritura

```
Procedure imprimir (v: vector);
```

```
var
  i,valor:integer;
```

	-1	18	4	0	5	57	-2	3	8	4
v	1	2	3	4	5	6	7	8	9	10

```
begin
  for i:= 1 to tam do
    begin
      valor:= v[i];
      write(valor);
    end;
  end;
```

ALTERNATIVA

```
Procedure imprimir (v: vector);
var
  i:integer;
begin
  for i:= 1 to tam do
    write (v[i]);
  end;
```

Agregar elementos al final

Significa agregar en el vector un elemento detrás del último elemento cargado en el vector. Puede pasar que esta operación no se pueda realizar si el vector está lleno.

A tener en cuenta:

1. Verificar si hay espacio (cantidad de elementos actuales es menor a la cantidad de elementos posibles)
2. Agregar al final de los elementos ya existentes el elemento nuevo.
3. Incrementar la cantidad de elementos actuales.

```
Program uno;
const
  fisica = 10;
type
  numeros= array [1..fisica] of integer;
```

```
var
  VN: numeros;
  dimL, valor:integer;
  ok:boolean;

Begin
  cargar (VN,dimL);
  read(valor);
  agregar(VN,dimL,ok,valor);
End.
```

```
Procedure agregar (var a :números; var dL:integer; var pude:boolean; num:integer);
```

```
Begin
```

```
  pude:= false;
```

Verifico si hay espacio

```
  if ((dL + 1) <= física) then
```

```
    begin
```

```
      pude:= true;
```

```
      dL:= dL + 1;
```

```
      a[dL]:= num;
```

```
    end;
```

```
end.
```

**Registro que se pudo realizar
Incremento la dimensión lógica
Agrego elelemento**

Insertar elementos

Significa agregar en el vector un elemento en una posición determinada. Puede pasar que esta operación no se pueda realizar si el vector está lleno o si la posición no es válida.

A tener en cuenta:

1. Verificar si hay espacio (cantidad de elementos actuales es menor a la cantidad de elementos posibles)
2. Verificar que la posición sea válida (esté entre los valores de dimensión definida del vector y la dimensión lógica).
3. Hacer lugar para poder insertar el elemento.
4. Incrementar la cantidad de elementos actuales.

```
Program uno;
```

```
  const
```

```
    física = 10;
```

```
  type
```

```
    numeros= array [1..física] of integer;
```

```
var
```

```
  VN: numeros;
```

```
  dimL, valor,pos:integer;
```

```
  ok:boolean;
```

```
Begin
```

```
  cargar (VN,dimL);
```

```
  read(valor); read(pos);
```

```
  insertar(VN,dimL,valor,ok,pos);
```

```
End.
```

```

Procedure insertar (var a :números; var dL:integer; var pude:boolean;
Var
    num:integer; pos: integer);
    i:integer;

Begin
    pude:= false;
    Verifico si hay espacio y si la
    posición es válida
    if ((dL + 1) <= física) and (pos>= 1) and (pos <= dL) )then begin

        for i:= dL downto pos do
            a[i+1]:= a[i];
        Corro los elementos empezando desde atrás hasta
        la posición a insertar para hacer el hueco donde
        se va a insertar el elemento

        pude:= true;
        a[pos]:= num;
        dL:= dL + 1;
        Registro que se pudo realizar
        Inserto el elemento
        Incremento la dimensión lógica
    end;
end;

```

Borrar elementos

Significa borrar (lógicamente) en el vector un elemento en una posición determinada, o un valor determinado. Puede pasar que esta operación no se pueda realizar si la posición no es válida, o en el caso de eliminar un elemento si el mismo no está.

A tener en cuenta:

1. Verificar que la posición sea válida (esté entre los valores de dimensión definida del vector y la dimensión lógica).
2. Hacer el corrimiento a partir de la posición y hasta el final.
3. Decrementar la cantidad de elementos actuales

```

Program uno;
  const
    fisica = 10;
  type
    numeros= array [1..fisica] of integer;

  var
    VN: numeros;
    dimL,pos:integer;
    ok:boolean;
  Begin
    cargar (VN,dimL);
    read(pos);
    eliminar(VN,dimL,ok,pos);
  End.

```

Procedure eliminar (var a :números; var dL:integer; var pude:boolean;pos: integer);

Var

i:integer;

Begin

pude:= false; **Verifico si la posición es válida**

if ((pos>= 1) and (pos <= dL))then begin

for i:= pos to (dL-1) do
a[i]:= a[i+1];

Corro los elementos empezando desde la posición hasta la dimensión lógica-1 para “tapar” el elemento a eliminar

pude:= true;
dL:= dL - 1;
end;

**Registro que se pudo realizar
Decremento la dimensión lógica**

end;

Búsqueda de un elemento

Significa recorrer el vector buscando un valor que puede o no estar en el vector. Se debe tener en cuenta que no es lo mismo buscar en un vector ordenado que en uno que no lo esté.

Hay dos formas:

- **Vector Desordenado:** Se debe recorrer todo el vector (en el peor de los casos), y detener la búsqueda en el momento que se encuentra el dato buscado o en el que se terminó el vector.
- **Vector Ordenado:** Se debe recorrer el vector teniendo en cuenta el orden:
 - BÚSQUEDA MEJORADA
 - BÚSQUEDA BINARIA

Buscar Desordenado

Se debe recorrer todo el vector (en el peor de los casos), y detener la búsqueda en el momento que se encuentra el dato buscado o en el que se terminó el vector.

A tener en cuenta:

1. Inicializar la búsqueda desde la posición 1 (pos).
2. Mientras ((el elemento buscado no se igual al valor en el arreglo[pos]) y (no se termine el arreglo))
 - 2.1 Avanzó una posición
3. Determino por qué condición se ha terminado el while y devuelvo el resultado.

```
Program uno;  
  const  
    fisica = 10;  
  type  
    numeros= array [1..fisica] of integer;  
  
  var  
    VN: numeros;  
    dimL, valor:integer;  
    ok:boolean;  
  
  Begin  
    cargar (VN,dimL);  
    read(valor);  
    res:= buscar(VN,dimL,valor);  
  End.
```

```
function buscar (a :números; dL:integer; valor:integer): boolean;  
Var  
  pos:integer;  
  esta:boolean;  
  
Begin  
  esta:= false;  
  pos:=1;  
  while ( (pos <= dL) and (not esta) ) do  
    begin  
      if (a[pos]= valor) then esta:= true  
      else  
        pos:= pos + 1;  
      end;  
    buscar:= esta;  
  end.
```

Buscar Ordenado

Búsqueda Mejorada

1. Inicializar la búsqueda desde la posición 1 (pos).
2. Mientras ((el elemento buscado sea menor al valor en el arreglo[pos]) y (no se termine el arreglo))
 - 2.1 Avanzó una posición
3. Determino por qué condición se ha terminado el while y devuelvo el resultado.

```
Program uno;  
  const  
    fisica = 10;  
  type  
    numeros= array [1..fisica] of integer;
```

```
var  
  VN: numeros;  
  dimL,pos:integer;  
  ok:boolean;  
Begin  
  cargar (VN,dimL);  
  read(valor);  
  ok:= existe(VN,dimL,valor);  
End.
```

```
Function existe (a:números; dL:integer; valor:integer):boolean;
```

```
Var  
  pos:integer;
```

```
Begin  
  pos:=1;  
  
  while ( (pos <= dL) and (a[pos]< valor)) do  
    begin  
      pos:= pos + 1;  
    end;  
  
    if ( (pos <= dL) and (a[pos]= valor)) then buscar:=true  
    else buscar:= false;  
  end.
```

Búsqueda Dicotómica

1. Se calcula la posición media del vector (teniendo en cuenta la cantidad de elementos)
2. Mientras ((el elemento buscado sea \neq arreglo[medio]) y (inf \leq sup))
 - Si ((el elemento buscado sea $<$ arreglo[medio]) entonces
Actualizo sup
 - Sino

Actualizo inf

Calculo nuevamente el medio

3. Determino por qué condición se ha terminado el while y devuelvo el resultado.

```
Program uno;
  const
    fisica = 10;
  type
    numeros= array [1..fisica] of integer;

  var
    VN: numeros;
    dimL,pos:integer;
    ok:boolean;
  Begin
    cargar (VN,dimL);
    read(valor);
    ok:= dicotomica(VN,dimL,valor);
  End.
```

```
Function dicotomica (a:números; dL:integer; valor:integer):boolean;
  Var
    pri, ult, medio : integer;
    ok:boolean
  Begin
    ok:= false;
    pri:= 1 ; ult:= dL; medio := (pri + ult ) div 2 ;

    While ( pri <= ult ) and ( valor <> vec[medio]) do
      begin
        if ( valor < vec[medio] ) then
          ult:= medio -1 ;
        else pri:= medio+1 ;
          medio := ( pri + ult ) div 2 ;
        end;
        if (pri <=ult) and (valor = vec[medio]) then ok:=true;
      end;
    dicotomica:= ok;
  end.
```

Puntero

Es un tipo de variable usada para almacenar una dirección en memoria dinámica. En esa dirección de memoria se encuentra el valor real que almacena. El valor puede ser de cualquiera de los tipos vistos (char, boolean, integer, real, string, registro, arreglo u otro puntero).

Un puntero es un tipo de datos simple.

Type

puntero = ^ tipo de datos;

Var

p:puntero;



Puede ser cualquiera de los tipos vistos previamente: integer, boolean, char, real, subrango, registro, vector.

Operaciones

- Creación de una variable puntero.
- Destrucción de una variable puntero.
- Asignación entre variables puntero.
- Asignación de un valor al contenido de una variable puntero.
- Comparación de una variable puntero

CREACIÓN

Implica reservar una dirección memoria dinámica libre para poder asignar contenidos a la dirección que contiene la variable de tipo puntero. ***new(variable tipo puntero)***.

```
Program uno;  
Type  
  puntero = ^integer;  
Var  
  num:integer;  
  p:puntero;  
  
Begin  
  new (p);  
  ...  
End.
```

ELIMINACIÓN

Implica liberar la memoria dinámica que contenía la variable de tipo puntero. ***dispose(variable tipo puntero)***.

```

Program uno;
Type
    puntero = ^integer;

Var
    num:integer;
    p:puntero;

Begin
    new (p);
    dispose (p);
End.

```

LIBERACIÓN

Implica cortar el enlace que existe con la memoria dinámica. La misma queda ocupada pero ya no se puede acceder. *nil*

```

Program uno;
Type
    puntero = ^integer;

Var
    num:integer;
    p:puntero;

Begin
    new (p);
    p:= nil;
End.

```

DISPOSE (p)

Libera la conexión que existe entre la variable y la posición de memoria.

Libera la posición de memoria.

La memoria liberada puede utilizarse en otro momento del programa.



p:=nil

Libera la conexión que existe entre la variable y la posición de memoria.

La memoria sigue ocupada.

La memoria no se puede referenciar ni utilizar.

ASIGNACIÓN entre punteros

Implica asignar la dirección de un puntero a otra variable puntero del mismo tipo. :=

```
Program uno;  
Type  
  puntero = ^integer;  
  
Var  
  q:puntero;  
  p:puntero;  
  
Begin  
  new (p);  
  q:=p;  
End.
```

CONTENIDO de un puntero

Implica poder acceder al contenido que contiene la dirección de memoria que tiene una variable de tipo puntero. ^

```
Program uno;  
Type  
  puntero = ^integer;  
  
Var  
  p:puntero;  
  
Begin  
  new (p);  
  p^:=8;  
  
End.
```

- ✓ if (**p = nil**) then, compara si el puntero p no tiene dirección asignada.
- ✓ if (**p = q**) then, compara si los punteros p y q apuntan a la misma dirección de memoria.
- ✓ if (**p^ = q^**) then, compara si los punteros p y q tienen el mismo contenido.

- × no se puede hacer read (p), ni write (p), siendo p una variable puntero.
- × no se puede asignar una dirección de manera directa a un puntero, p:= ABCD
- × no se pueden comparar las direcciones de dos punteros (p<q).

Listas

Es una colección de nodos. Cada nodo contiene un elemento (valor que se quiere almacenar en la lista) y una dirección de memoria dinámica que indica dónde se encuentra el siguiente nodo de la lista.

Toda lista tiene un nodo inicial.



Los **nodos** que la componen pueden no ocupar posiciones contiguas de memoria. Es decir pueden aparecer dispersos en la memoria, pero mantienen un orden lógico interno.

Las listas son homogéneas, dinámicas, lineales y secuenciales.

```
Program uno;
```

```
Type
```

```
    nombreTipo= ^nombreNodo;
```

```
    nombreNodo = record
```

```
        elemento: tipoElemento;
```

```
        punteroSig: nombreTipo;
```

```
    end;
```

```
Var
```

```
    Pri: nombreTipo;
```



tipoElemento es cualquiera de los tipos vistos (entero,char,boolean,registro,arreglo,real,subrangol).
Es una estructura recursiva.
El orden de la declaración debe respetarse

Operaciones

- Creación de una lista.
- Agregar nodos al comienzo de la lista.
- Recorrido de una lista.
- Agregar nodos al final de la lista.
- Insertar nodos en una lista ordenada
- Eliminar nodos de una lista

CREAR UNA LISTA

Implica marcar que la lista no tiene una dirección inicial de comienzo.

```

Program uno;

Type listaE= ^datosEnteros;

      datosEnteros= record
                        elem:integer;
                        sig:listaE;
                        end;

Procedure crear (var p: listaE);
begin
    p:= nil;
end;

Var
    pri: listaE;

Begin
    crear (pri);
End.

```

RECORRER UNA LISTA

Implica posicionarse al comienzo de la lista y a partir de allí ir “pasando” por cada elemento de la misma hasta llegar al final.

```

procedure recorrerLista (pI: listaE);
Var
    aux:listaE;

begin
    aux:= pI;
    while (aux <> nil) do
        begin
            write (aux^.elem);
            aux:= aux^.sig;
        end;
    end;
end;

```

ALTERNATIVA

```

procedure recorrerLista (pI: listaE);

begin
    while (pI <> nil) do
        begin
            write (pI^.elem);
            pI:= pI^.sig;
        end;
    end;
end;

```

AGREGAR ADELANTE

Implica generar un nuevo nodo y agregarlo como primer elemento de la lista.

Reservo espacio en memoria nuevo elemento.

si (es el primer elemento a agregar)
 asigno al puntero inicial la dirección del nuevo elemento.

sino
 indico que el siguiente de nuevo elemento es el puntero inicial.
 actualizo el puntero inicial de la lista con la dirección del nuevo elemento.

```
procedure agregarAdelante (var pI:listaE; num:integer);  
Var
```

```
    nuevo:listaE;   Creo espacio para el  
                    nuevo elemento
```

```
Begin  
    new (nuevo); nuevo^.elem:= num; nuevo^.sig:=nil;  
    if (pI = nil) then pI:= nuevo  
    else begin  
        nuevo^.sig:= pI;  
        pI:=nuevo;  
    end;  
End;
```

Evalúo el caso y
reasigno los
punteros

AGREGAR AL FINAL

Implica generar un nuevo nodo y agregarlo como último elemento de la lista.

Reservo espacio en memoria nuevo elemento.

si (es el primer elemento a agregar)
 asigno al puntero inicial la dirección del nuevo elemento.
 asigno al puntero final la dirección del nuevo elemento.

sino
 actualizo como siguiente del puntero final al nuevo elemento
 actualizo el la dirección del puntero final

```

procedure agregarAlFinal2 (var pI,pU:listaE; num:integer);
Var
    nuevo:listaE;

Begin
    new (nuevo); nuevo^.elem:= num; nuevo^.sig:=nil;

    if (pI = nil) then begin
        pI:= nuevo;                                <----- Evalúo si la lista está vacía
        pU:= nuevo;
    end
    else begin
        pU^.sig:=nuevo;
        pU:= nuevo;                                <----- Actualizo el siguiente del
                                                    último nodo y al último nodo
    end;

End;

```

BÚSQUEDA

Significa recorrer la lista desde el primer nodo buscando un valor que puede o no estar. Se debe tener en cuenta si la lista está o no ordenada.

LISTA ORDENADA

Se debe recorrer la lista teniendo en cuenta el orden. La búsqueda se detiene cuando se termina la lista o el elemento buscado es mayor al elemento actual.

```

function buscar (pI: listaE; valor:integer):boolean;
Var
    aux:listaE;
    encontré:boolean;

Begin
    encontré:= false;
    aux:= pI;
    while ((aux <> nil) and (aux^.elem < valor)) do
        begin
            aux:= aux^.sig;
        end;

        if (aux <> nil) and (aux^.elem = valor) then encontré:= true;

    buscar:= encontré;
end;

```

LISTA DESORDENADA

Se debe recorrer toda la lista (en el peor de los casos), y detener la búsqueda en el momento que se encuentra el dato buscado o en el que la lista se terminó.


```

function buscar (pI: listaE; valor:integer):boolean;
Var
  aux:listaE;
  encontré:boolean;

Begin
  encontré:= false;
  aux:= pI;
  while ((aux <> nil) and (encontré = false)) do
    begin
      if (aux^.elem = valor) then
        encontré:=true
      else
        aux:= aux^.sig;
      end;
    buscar:= encontré;
  end;
end;

```

ELIMINAR

Implica recorrer la lista desde el comienzo pasando nodo a nodo hasta encontrar el elemento y en ese momento eliminarlo (dispose). El elemento puede no estar en la lista.

- Si la lista está **desordenada** seguramente la búsqueda se realizará hasta encontrar el elemento o hasta que se termina la lista.
- Si la lista está **ordenada** seguramente la búsqueda se realizará hasta que se termina la lista o no se encuentre un elemento mayor al buscado.

ELIMINAR EN UN LISTA DESORDENADA

Comienzo a recorrer la lista desde el nodo inicial.

mientras ((no sea el final de la lista)y(no encuentre el elemento))

el puntero anterior toma la dirección del puntero actual
 avanzo el puntero actual

si (encontré el elemento) entonces

si (es el primer nodo) entonces

actualizo el puntero inicial de la lista

elimino la dirección del puntero actual

```
sino
    actualizo el siguiente del puntero anterior con el
siguiente de actual
    elimino la dirección del puntero actual
```

```
procedure eliminar (Var pI: listaE; valor:integer);
Var
    actual,ant:listaE;

Begin
    actual:=pI;
    while (actual <> nil) and (actual^.elem <> valor) do begin
        ant:=actual;
        actual:= actual^.sig;
    end;
    if (actual <> nil) then
        if (actual = pI) then
            pI:= pI^.sig;
        else
            ant^.sig:= actual^.sig;

        dispose (actual);
    End;
```

Si el elemento se repite

```
procedure eliminar (Var pI: listaE; valor:integer);
Var
    actual,ant:listaE;

Begin
    actual:=pI;
    while (actual <> nil) do begin
        if (actual^.elem <> valor) then begin
            ant:=actual; actual:= actual^.sig;
        end;
        else begin
            if (actual = pI) then
                pI:= pI^.sig;
            else
                ant^.sig:= actual^.sig;
            dispose (actual);
            actual:= ant;
        end;
    End;
```

INSERTAR

Implica agregar un nuevo nodo a una lista ordenada por algún criterio de manera que la lista siga quedando ordenada.

```
procedure insertar (Var pI: listaE; valor:integer);
Var
  actual,anterior,nuevo:listaE;
Begin
  new (nuevo); nuevo^.elem:= valor; nuevo^.sig:=nil;
  if (pI = nil) then    pI:= nuevo

  else begin
    actual:= pI; ant:=pI;
    while (actual <> nil) and (actual^.elem < nuevo^.elem) do
      begin
        anterior:=actual;
        actual:= actual^.sig;
      end;
    end;
    if (actual = pI) then
      begin
        nuevo^.sig:= pI;  pI:= nuevo;
      end
    else
      begin
        anterior^.sig:= nuevo;  nuevo^.sig:= actual;
      end;
    end;
  End;
```