

CADP Final

Tipos de datos definidos por el usuario:

```
type
  numeritos = integer;
var
  num1:numeritos;
  num2:integer;
begin
  read(num1);          // Ambos casos serian validos
  num1:= num1 div 10; //
  num2 := num1 + 3; //> No puede hacerse ninguna operacion que los relacione
```

Cuando defino un tipo de dato nuevo, este no puede mezclarse con otros tipos de datos.

ej:

No se puede asignar una variable `cadena20 := una variable string[20]` porque son de distinto tipo (pascal chequea por tipos)

^Ventajas TDDU:

- Flexibilidad
- Documentacion
- Seguridad

Subrango: Es un tipo ordinal (integer o char) (predefinido o definido por el usuario)

- Es simple
- Es ordinal

```
type
  Mayusculas = "A" .. "Z"; //> Puede realizar todas las operaciones de char
                        //> y solo puede tomar alguno de los valores declarados
```

El subrango se puede utilizar como indice en un for, case, etc.

- Es valido el rango '0' .. '10' ?
 - No es valido porque no es ordinal, '10' es una cadena, no un caracter.
 - Es valido el rango '0' .. '9' ?
 - Si es valido porque es un tipo ordinal
-

Funciones

- Devuelven un unico valor de tipo simple (punteros, subrangos,integer,real,boolean,char, tddu?)
- Debe coincidir el tipo de la variable con el tipo que retorna la funcion

```
function ejemplo (num:integer):integer;
var
  resultado:integer;
begin
  ejemplo:=resultado;
end;
```

Cuando invoco a una funcion y lo que devuelve se lo asigno a una variable, esa variable debe ser del mismo tipo. —> retorna a la misma linea que la invoco

Una funcion puede ser invocada desde:

- Estructuras de control
- Asignacion a variables
- Write

```
if (esPar(num)) then ...
ok:= esPar(num);
write(getTexto(variable));
```

- **Una funcion puede devolver un tipo de dato registro (V O F)**

- F. Un registro es un tipo de dato compuesto y las funciones solo devuelven tipos de datos simples.
 - **Una funcion puede devolver un TDDU simple? (V o F)**
 - Si, porque es un tipo de dato simple.
-

Estructuras de datos

- Clasificacion:
 - Tipo de elemento: homogenea / heterogenea
 - Tamaño: estatica / dinamica
 - Tipo de acceso: secuencial / directo
 - Linealidad: lineal / no lineal

Registro

- Heterogeneo
- Estatico
- Acceso directo
- Lineal

Vector

- Homogenea
- Estatica
- Acceso directo
- Lineal

Lista

- Homogenea
- Dinamica
- Acceso secuencial
- Lineal

- Si se la cantidad maxima de elementos a almacenar en una estructura, siempre elegimos una estructura estatica?
 - No, saber la cantidad maxima de elementos me da la **posibilidad** de elegir un vector, y no saberlo me obliga a utilizar una lista. Por lo tanto hay que evaluar que estructura es mas eficiente segun el problema.
-

Alcance de variables

- Variables globales: pueden ser utilizadas en todo el programa (incluyendo modulos).
- Variable local: Pueden ser utilizadas solo en el proceso que estan declaradas.
- Variables locales al programa: Solo pueden ser usadas en el cuerpo del programa.

Orden de busqueda de variables **en modulos**:

- 1- Es variable local?
- 2- Es un parametro?
- 3- Es una variable global?
- 4- Si no es ninguna anterior, da error.

Orden de busqueda de variables **en programa principal**:

- 1- Es variable local?
- 2- Es variable global?
- 3- Si no es ninguna anterior, da error.

Si debemos imprimir el contenido de una variable **sin inicializar**, decimos que imprime basura

Si debemos imprimir el contenido de una variable **no declarada**, decimos que el programada error.

Problemas con variables globales:

- Demasiados identificadores
- Conflictos con los nombres de variables
- Posibilidad de perder integridad de datos

La solución a estos problemas causados por variables globales es una combinación de **ocultamiento de datos** (data hiding) y uso de **parametros**

- Ocultamiento de datos: Metodo para ocultar datos y proteger su integridad.
Los datos exclusivos de un modulo no deben ser accesibles fuera del mismo.
- El uso de parametros significa que los datos compartidos se deben especificar como parametros que se transmiten entre modulos

Parametro por valor: El modulo recibe (sobre una variable local) un valor proveniente de otro modulo o programa principal.

Con el puede realizar operaciones pero no producira ningun cambio ni tendra incidencias fuera del modulo.

Parametro por referencia: El modulo recibe el nombre de una variable (referencia a una direccion) conocida en otros modulos del sistema.

Puede operar con ella y su valor original dentro del modulo, y las modificaciones que se produzcan se reflejaran en los demas modulos que conocen la variable.

- Un parametro por referencia y un parametro por valor pueden coincidir en tamaño?
 - Si, porque un parametro por referencia es un puntero que siempre pesa 4 bytes, y un parametro por valor puede ser un integer que pesa 4 bytes tambien

Modularizacion

- Descomposicion funcional
- Favorece reusabilidad

Permite aislar los errores producidos con mayor facilidad y se corrigen en menor tiempo.

- Beneficios modularizacion:
 - Legibilidad
 - Productividad
 - Reusabilidad
 - Mantenimiento
 - Facilidad crecimiento
 - Puede un modulo tener su propio type interno?
 - Si, ya que un modulo es una unidad auto-contenida y puede encapsular tipos propios.
 - Es posible utilizar variables globales en un programa modularizado?
 - V, es posible porque son dos formas distintas de comunicacion que pueden coexistir.
 - Se pueden utilizar variables locales para comunicacion entre modulos?
 - No, porque las variables locales solo tienen alcance local.
 - Como se pueden comunicar los modulos?
 - Los modulos pueden comunicarse mediante el uso de parametros y variables globales, aunque estas ultimas no se recomiendan.
 - Se puede declarar un tipo nuevo dentro de un modulo? de ser asi, donde puedo declarar variables de ese tipo nuevo?
 - V, porque un modulo es una unidad auto-contenida y puede encapsular tipos propios. Solo se pueden declarar dentro de ese modulo.
 - Un programa que utiliza solo variables globales no requiere modularizacion (V O F)
 - F, la modularizacion ayuda al mantenimiento, legibilidad, depuracion del programa, aunque el mismo utilice variables globales, seria una buena practica que este modularizado.
-

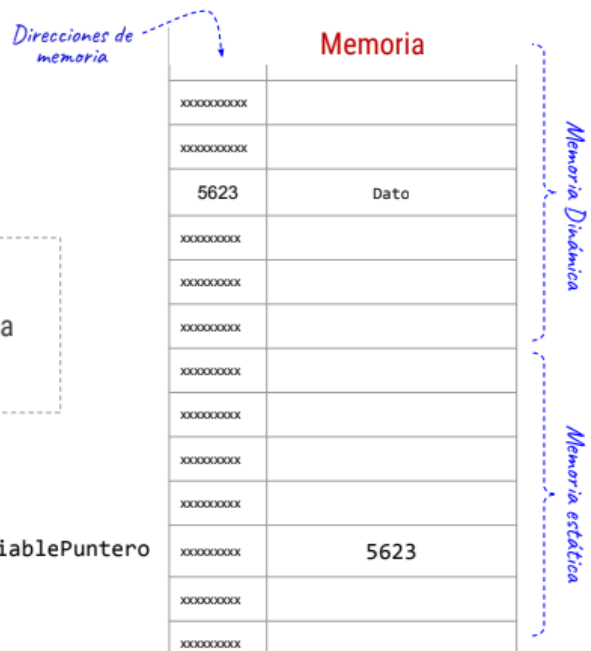
Punteros

Almacena una dirección de memoria dinámica.

VARIABLES DINAMICAS

Variable Puntero: variable estática que almacena la dirección en memoria de otra variable (*llamada variable dinámica*).

miVariablePuntero



USO DE PUNTEROS

Declaración (ejemplo)	Valores posibles de un puntero
TYPE PunteroEntero = ^ integer; VAR pun, otroPun: PunteroEntero;	<ul style="list-style-type: none">• NIL• Dirección de memoria dinámica
¿Qué se puede hacer con punteros? <ul style="list-style-type: none">• Reservar memoria dinámica (new)• Liberar memoria dinámica (dispose)• Asignar punteros• Comparar punteros• Acceder al dato apuntado por el puntero (^)	

- Cada variable puede apuntar a un único tipo de dato.
- Siempre ocupa la misma cantidad de memoria estática
- Puede reservar y liberar memoria dinámica durante ejecución

Diferencias entre dispose y nil:

CADP – TIPOS DE PUNTERO

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.



DISPOSE -NIL



`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.

Cuando un puntero se pasa **por referencia**, lo que se modifica es **la dirección** a la que apunta.

Cuando un puntero se pasa **por valor y se modifica el contenido**, los cambios persistiran.

$q = p \rightarrow$ Apuntan a lo mismo

$q^{\wedge} = p^{\wedge} \rightarrow$ Contienen lo mismo

Si p y q apuntan a lo mismo y se hace dispose, ambos liberan la memoria.

Si se asigna nil a un puntero, la memoria sigue ocupada.

- Siempre es necesario hacer nil para reservar memoria dinamica?
 - No, por ejemplo en el modulo agregarAdelante, se reserva memoria dinamica para el nuevo elemento sin la necesidad de hacer nil antes.

Calculo de memoria

- Dinamica: tener en cuenta **new** y **dispose**.

- Estática: tener en cuenta **constantes, variables globales y locales al programa**.
- En el caso de que solo haya módulos como en el siguiente final donde calcular la memoria, tener en cuenta los parámetros:
 - Si el parámetro es por valor, vale lo que vale el tipo de dato
 - Si es por referencia, 4 bytes porque es una dirección de memoria (es un puntero).

PREGUNTA 5

■ Dado la siguiente estructura y los siguientes módulos pertenecientes a un mismo programa. Indique cual de los dos módulos requiere de más memoria para su ejecución. Justifique

```
const
  dimf = 100;
type
  rango = 1..dimf;
  vector = array [rango] of integer;

  producto = record
    cod: string[5];
    categoria: char;
    precio: real;
    disponible: boolean;
  end;

  lista = ^nodo;
  nodo = record
    dato: producto;
    sig: lista;
  end;
end;
```

A Procedure modificar(var v: vector; pos: rango; dial: integer; elem: integer; var ok: boolean); begin ok := false; if (pos > 0) and (pos <= dial) then begin v[pos] := elem; ok := true; end; end;	Referencia: char: 1 byte integer: 4 byte real: 6 byte boolean: 1 byte string: longitud + 1 byte; puntero: 4 bytes
B Procedure modificar(l: lista; elem: producto); Begin ok := true; while (l <> nil) and (l^.dato.cod <> elem.cod) do l := l^.sig; if (l <> nil) then l^.dato := elem; else ok := false; end; end;	Referencia: char: 1 byte integer: 4 byte real: 6 byte boolean: 1 byte string: longitud + 1 byte; puntero: 4 bytes

Punteros siempre ocupan 4 bytes de memoria (siempre revisar tabla porque en el final los suelen cambiar)

- No confundir **new(v)** con **new(v[i])**.
- Ojo con hacer new o dispose en una posición de vector inválida (verificar los rangos (si el for empieza el 0 y el vector empieza en 1))

```
type
  vector = array [1..10] of integer;
var
  v:vector;
for:= 0 to 9 do
  new(v[i]);  // (no existe v[0] porque el arreglo empieza en v[1])
```

Tiempo de ejecucion

Se tienen en cuenta **asignaciones y operaciones aritmeticas.**

If → 1 ut + cuerpo

If / else → Considerar el peor caso

for: $(3.N + 2) + N.(CUERPO)$

Si el for tiene indices invertidos, **no se ejecuta** (-10 downto 10). $n = 0 \rightarrow$ **igualmente calcular con la formula**

Si n es negativo (como el caso anterior), no se ejecuta, se considera 0, y al no ejecutarse nunca el cuerpo = 0

IGUALMENTE SE CALCULA

$$3.0 + 2 + 0.0 = 2ut$$

A veces no se conoce el valor de n y quedara expresado en funcion de n

while: $C.(N + 1) + N.(CUERPO)$

C: cantidad de condiciones + operadores logicos

- while **(n > 0) and (ok)** do = 2 condiciones + 1 operador logico → C = 3
- while **(n <> 0) and (ok) and (cant < total)** do = 3 condiciones + 2 operadores logicos → C = 5

Si no conocemos el valor de n, dejamos la formula expresada en valores de n

Si n vale 0, igualmente se calcula la formula

Como obtener valor de n?

$$n = (\text{limite superior} - \text{limite inferior}) + 1$$

```
for := 2 to 10 do  
for := 1 to 10 do
```

$$N = (10 - 2) + 1 = 9$$

$$N = (10 - 1) + 1 = 10$$

Registro

La unica operacion permitida es la asignacion entre dos variables del mismo tipo.

La unica forma de acceder a los campos es `variable.nombreCampo`

Como saber si dos registros son iguales? → Verificar campo por campo

- Un tipo de dato registro puede ser homogeneo si todos sus campos son del mismo tipo de dato"
 - F. Un caso particular de que todos los elementos sean del mismo tipo no modifica la caracteristica del tipo de dato registro, el cual es heterogeneo.

Lista

- **Agregar**

```
procedure agregarAdelante (var l:lista ; elem:dato);
var
    nue:lista;
begin
    new(nue);
    nue^.dato := elem;
    nue^.sig := L;
    L := nue;
end;

procedure agregarAtras (var l,ult: lista ; elem:dato);
var
    nue:lista;
begin
    new(nue);
    nue^.dato := elem;
    nue^.sig := nil;
    if (L = nil) then
        L := nue
    else
        ult^.sig := nue;
        ult := nue;
    end;
end;

procedure insertarOrdenado (var l:lista; elem:dato);
var
    nue,act,ant:lista;
```

```

begin
  new(nue);
  nue^.dato := elem;
  act := l;
  ant := l;
  while (act <> nil) and (act^.dato < elem) do begin
    ant := act;
    act := act^.sig;
  end;
  if (act = ant) then
    L := nue
  else
    ant^.sig := nue;
    nue^.sig := act;
  end;
end;

```

- **Buscar un elemento:** Implica recorrer la lista desde el comienzo pasando nodo a nodo hasta encontrar el elemento buscado o que se termine la lista.

Busqueda en lista desordenada

```

function buscar (L:lista ; valor:integer):boolean;
var
  ok:boolean;
begin
  ok := false;
  while (l <> nil) and (not ok) do begin
    if (L^.dato = valor) then
      ok := true
    else
      L := L^.sig;
    end;
  buscar := ok;
end;

```

Busqueda en lista ordenada:

```

function buscar (L:lista ; valor:integer):boolean;
var
  ok:boolean;
begin
  ok:=false;
  while (l <> nil) and (valor > l^.dato) do
    L := L^.sig;
  if (l <> nil) and (valor = l^.dato) then
    ok := true;

```

```
    buscar := ok;
end;
```

- Importa el orden de las condiciones? **si**, primero siempre se pregunta si es \neq nil y despues si es igual al valor buscado

Insertar un elemento: Se necesita que la lista tenga un orden e implica agregar el elemento de manera que la lista siga ordenada.

```
procedure insertarOrdenado (var L:lista; elem:integer);
var
    nue, ant, act:lista;
begin
    new(nue);
    nue^.dato := elem;
    ant:=L;
    act:=L;
    while (act <> nil) and (elem > act^.dato) do begin
        ant := act;
        act := act^.sig;
    end;
    if (act = ant) then
        L := nue
    else
        ant^.sig := nue;
        nue^.sig := act;
    end;
end;
```

- **Eliminar:** Implica recorrer la lista hasta encontrar el elemento y eliminarlo.

Puede que el elemento no este en la lista.

2 casos:

- El elemento a eliminar sea el primer nodo de la lista
- El elemento a eliminar **no** sea el primer nodo de la lista
- Eliminar primer ocurrencia en lista desordenada

```
procedure eliminarListaDesordenada (var L:lista ; dato:integer);
var
    act, ant:lista;
begin
```

```

    ant := l;
    act := l;
    while (act <> nil) and (dato <> act^.dato) do begin
        ant := act;
        act := act^.sig;
    end;
    if (act <> nil) then begin
        if (act = ant) then
            L := L^.sig
        else
            ant^.sig := act^.sig;
            dispose[act];
            act := ant;
        end;
    end;
end;

```

- Eliminar todas las ocurrencias en lista desordenada

```

procedure eliminar (var l:lista ; valor:integer);
var
    ant,act:lista;
begin
    act:= l;
    ant := l;
    while (act <> nil) do begin
        if (act^.dato = valor) then begin
            if (act = ant) then
                L := L^.sig
            else
                ant^.sig := act^.sig;
                dispose[act];
                act := ant;
            end
        else begin
            ant := act;
            act := act^.sig;
        end;
    end;
end;

```

- Eliminar en lista ordenada primera ocurrencia

```

procedure eliminarListaOrdenada (var L:lista ; dato:integer);
var
    ant,act:lista;
begin
    ant := l;
    act := l;
    while (act <> nil) and (act^.dato < dato) do begin
        ant := act;
        act := act^.sig;
    end;
end;

```

```

end;
if (act <> nil) and (act^.dato = dato) then begin
  if (act = ant) then
    L := l^.sig
  else
    ant^.sig := act^.sig
  dispose[act];
  act := ant;
end;
end;

```

- Eliminar en lista ordenada todas las ocurrencias

```

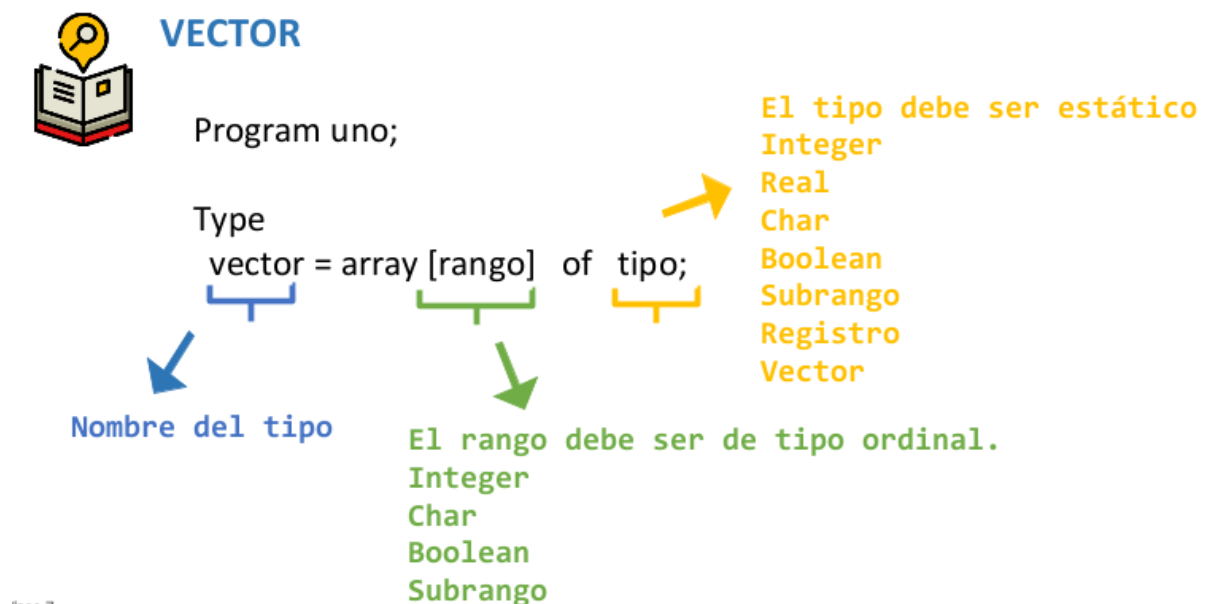
procedure eliminarListaOrdenadaTodo (var L:lista ; dato:integer);
var
  ant,act,aux:lista;
begin
  ant :=l;
  act :=l;
  while (act <> nil) and (dato > act^.dato) do begin
    ant := act;
    act := act^.sig;
  end;
  while (act <> nil) and (dato = act^.dato) do begin
    if (act = ant) then begin
      while (act <> nil) and (act^.dato = dato) do begin
        act := act^.sig;
        dispose[ant];
        ant := act;
      end;
      dispose[ant];
      L := act;
    end
    else begin
      aux = ant;
      while (act <> nil) and (act^.dato = dato) do begin
        ant := act;
        act := act^.sig;
        dispose[ant];
      end;
      dispose[ant];
      aux^.sig = act;
    end;
  end;
end;

```

- "Nunca es posible acceder al nodo de la posición N en una estructura de tipo lista".
 - F, si es posible, si la lista no está vacía y el nodo N existe

- Puedo utilizar un for para recorrer una lista?
 - Si, pero debo conocer algunas precondiciones como la cantidad de elementos.
- Lista **siempre** preguntar si no es nil (si pri=nil entonces ult=nil) (nunca se garantiza que la estructura este cargada)
- Si borro varios nodos contiguos en una lista ordenada, el enlace entre ant y act^.sig se realiza una sola vez al final.

Vector



RECORRIDO TOTAL → Implica analizar **todos** los elementos del vector, lo que lleva a recorrer completamente la estructura.

RECORRIDO PARCIAL → Implica analizar los elementos del vector, hasta encontrar aquel que cumple con lo pedido. Puede ocurrir que se recorra todo el vector.

Ej: Informe la primer posición donde aparece un múltiplos de 3. Suponga que los nros leídos son positivos y que **existe al menos un múltiplo de 3**.


```

function posicion (v:vector):integer;
var
  pos:integer;
  ok:boolean;
begin
  pos := 1;
  ok := true
  while true do begin
    if (v[pos] MOD 3 = 0) then
      ok := false
    else
      pos := pos + 1;
    end;
    posicion := pos;
  end;
end;

```

Que cambio si el enunciado **no asegura** que haya al menos un multiplo de 3?

```

function posicion (v:vector; dimL:integer):integer;
var
  pos:integer;
  ok:boolean;
begin
  pos := 1;
  ok := false;
  while (pos <= dimL) and (not ok) do begin
    if (v[pos] MOD 3 = 0) then
      ok := true;
    else
      pos := pos + 1;
    end;
    if (ok = true) then
      posicion := pos
    else
      posicion := -1;
    end;
  end;
end;

```

Forma correcta de cargar un arreglo utilizando dimesion logica

```

procedure cargar (var v:vector ; var dimL:integer);
var
  num:integer;
begin
  dimL:= 0;
  read(num);
  while ((dimL < dimF) and (num <> 50)) do begin
    dimL := dimL + 1;
    v[dimL] := num;
    read(num);
  end;
end;

```

```
end;  
end;
```

Operaciones en vectores

- **Agregar:** Significa agregar al final de los elementos que tiene el vector.

Puede que no se pueda realizar si el vector esta lleno.

- 1- Verificar si hay espacio ($\text{dimL} < \text{dimF}$)
- 2- Agregar al final de los elementos ya existentes el nuevo elemento.
- 3- Incrementar la cantidad de elementos actuales.

```
procedure agregar (var v:vector; var dimL:integer; elem:integer; var ok:boolean);  
begin  
  ok := false;  
  if (dimL < dimF) do begin  
    v[dimL + 1] := elem;  
    dimL := dimL + 1;  
    ok := true;  
  end;  
end;
```

- **Insertar:** Significa agregar en el vector un elemento en una posicion determinada.

Puede que no se pueda realizar si el vector esta lleno o la posicion no es valida.

- 1- Verificar si hay espacio ($\text{dimL} < \text{dimF}$)
- 2- Verificar que la posicion sea valida ($\text{pos} \geq 1$ and $\text{pos} \leq \text{dimL}$)
- 3- Hacer lugar para poder insertar el elemento
- 4- Incrementar la cantidad de elementos actuales

```
procedure insertar (var v:vector ; var dimL:integer ; elem:integer; pos:integer; var ok);  
var  
  i:integer;  
begin  
  ok := false;  
  if ((dimL + 1) <= dimF) and ((pos >= 1) and (pos <= dimL)) then begin  
    for i:= dimL downto pos do  
      v[i+1] := v[i];  
    v[pos] := elem;
```

```

    dimL := dimL + 1;
    ok := true
end;
end;

```

- **Eliminar:** Significa borrar logicamente en el vector un elemento en una posición determinada, o un valor determinado.

Puede que no se pueda realizar si la posición no es válida, o en el caso de eliminar un elemento si el mismo no está.

Eliminar de una posición:

- 1- Verificar que la posición sea válida ($\text{pos} \geq 1$ and $\text{pos} \leq \text{dimL}$)
- 2- Hacer el corrimiento a partir de la posición y hasta el final
- 3- Decrementar la cantidad de elementos actuales

```

procedure eliminar (var v:vector ; var dimL:integer; pos:integer; var ok:boolean);
var
    i:integer;
begin
    ok := false;
    if (pos >= 1) and (pos <= dimL) then begin
        for i:= pos to dimL-1 do
            v[i] := v[i+1];
        ok := true;
        dimL := dimL - 1;
    end;
end;

```

- **Buscar:** significa recorrer el vector buscando un valor que puede o no estar en el vector.

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 que se terminó el vector.

Vector ordenado: Se debe aprovechar el orden → búsqueda mejorada y búsqueda dicotómica.

```

function buscar (v:vector ; dimL:integer; valor:integer):boolean;
var
  ok:boolean;
  i:integer;
begin
  i:=1;
  ok := false;
  while (i <= dimL) and (not ok) do begin
    if (v[i] = valor) then
      ok := true
    else
      i := i + 1;
    end;
  buscar := ok;
end;

```

- Siempre en ejercicios verificar que antes de cualquier operacion, se verifique que $\text{dimL} \geq 1$ (nunca se garantiza que las estructuras esten cargadas)
- Un vector siempre se utiliza teniendo en cuenta la dimension logica (V O F)
 - No, por ejemplo cuando utilizamos un vector contador, utilizamos el vector teniendo en cuenta la dimension fisica.