En Pascal, existen **dos** formas principales de pasar argumentos a procedimientos y funciones: **por valor y por referencia.**

Parámetros por valor:

Se crea una copia del valor del argumento original.

Cualquier modificación realizada dentro del procedimiento o función se realiza en la copia, no en el valor original.

Se utiliza cuando no se desea que el procedimiento o función modifique el valor original de la variable.

Parámetros por referencia:

Se pasa la dirección de memoria de la variable original al procedimiento o función.

Cualquier modificación realizada dentro del procedimiento o función se realiza directamente en el valor original de la variable.

Se utiliza cuando se desea que el procedimiento o función modifique el valor original de la variable. En Pascal, los módulos, conocidos como subprogramas, se dividen en dos categorías: **procedimientos y funciones.** Ambos tipos de módulos permiten agrupar y reutilizar código para mejorar la organización y legibilidad de los programas, pero se diferencian:

Procedimientos:

Proceso: devuelven desde 0 a más valores

Objetivo: ejecutan instrucciones específicas

Estructura: se define usando la palabra “procedure”, seguida del nombre del procedimiento y una lista de parámetros entre paréntesis (opcional). Dentro del bloque se declaran las variables local (si se requiere)y se escriben las instrucción que se ejecutan al llamar al procedimiento.

Ejemplo: un procedimiento para calcular el área de un rectángulo: e imprimir el resultado:

procedure calcularAreaRectangulo(base: real; altura: real)**;**

```
var
  area: real;
begin
  area := base * altura;
  writeln('El area del rectángulo es: ', area);
end;
```

Invocación/Llamada: se invocan desde otro lugar utilizando su nombre seguido de la lista de argumentos entre paréntesis en el mismo orden que se creó. Los argumentos pueden ser pasados por valor o por referencia para proporcionar valores a las variables del procedimiento según se desee.

Seguido el **ejemplo** anterior:

calcularAreaRectangulo(base,altura);

Funciones:

Función: devuelve un valor único (tiene que ser tipo de dato Ordinal/Simple).

Objetivo: ejecutan instrucciones específicas y devuelven el resultado a la unidad que la llamó.

Estructura: se define usando la palabra “function”, seguida del nombre de la función, una lista de parámetros entre paréntesis y un tipo de dato que retorna. Dentro del bloque se declaran las variables locales y se escriben las instrucciones que llevan a calcular un valor de retorno.

Ejemplo: Una función para calcular el cuadrado de un número (num²).

```
function cuadrado(numero: real): real;
begin
  cuadrado := numero * numero;
end;
```

Invocación/ Llamada: se invocan desde otro lugar utilizando su nombre seguido de la lista de argumentos entre paréntesis. Los argumentos pueden ser pasados por valor o por referencia para proporcionar valores a las variables de la función. El valor de retorno se almacena en una variable o se utiliza en una expresión.
Seguido el **ejemplo** anterior, hay 2 opciones a elegir para este caso:

1) **Imprimir el valor:**
   writeln('El cuadrado del número ', num, ' es: ', cuadrado(num));

2) **Utilizar una variable auxiliar** para almacenar el valor (debe ser del mismo tipo que devuelve la función):
   aux:= cuadrado(num);

Este módulo es para leer un tipo de **dato entero**, puede usarse con cualquier tipo de **dato simple:**

01	**procedure leerNum***(var unNum: integer);*		//	*unNum es el parámetro donde se almacena el dato*
02	**begin**			
03	write('Número: ');		//	Escribe en pantalla el mensaje entre comillas
04	readln(unNum);		//	Almacena en el parámetro unNum el valor ingresado
05	**end**;			

También puede aplicarse a tipos de **datos compuestos** como **registros:**

01	**procedure leerAuto***(var au: autos);*		//	*au es el parámetro donde se almacena el dato*
02	**begin**			
03	write('Marca de auto: ');		//	Escribe en pantalla el mensaje entre comillas
04	readln(au.marca);		//	Almacena en el parámetro au, campo marca, el dato ingresado

05	write('Patente de auto: ');		//	Escribe en pantalla el mensaje
06	readln(au.patente);		//	Almacena en el parámetro au, campo patente, el dato ingresado
07	write('Titular: ');		//	Escribe en pantalla el mensaje
08	readln(au.titular);		//	Almacena en el parámetro au, campo titular, el dato ingresado
09	**end**;			

Al usar registros, suele haber **cortes de control**. Por ejemplo: si se tiene que leer autos hasta que se ingresa la marca "ZZZ" que no debe procesarse, debemos preguntar primero por el campo de registro marca (au.marca) para no procesar los demás datos del auto en el módulo . En caso de que sea distinto del corte ("ZZZ") ingresamos los siguientes campos que le corresponden al auto.
Recordar que esto se hace para hacer más eficiente el programa y posteriormente en la carga de cualquier estructura(lista o vector de registros), a cargar, debe considerarse nuevamente el corte de control.

Considerando lo anterior, el módulo quedaría:

01	**procedure leerAuto***(var au: autos);*		//	*au es el parámetro donde se almacenacarga el dato*
02	**begin**			
03	write('Marca de auto: ');		//	Escribe en pantalla el mensaje entre comillas
04	readln(au.marca);		//	Almacena en el parámetro au, campo marca, el dato ingresado
05	if (au.marca <> 'ZZZ')then begin		//	Consulta si el dato ingresado es diferente al corte de control
06	write('Patente de auto: ');		//	Escribe en pantalla el mensaje
07	readln(au.patente);		//	Almacena en el parámetro au, campo patente, el dato ingresado
08	write('Titular: ');		//	Escribe en pantalla el mensaje
09	readln(au.titular);		//	Almacena en el parámetro au, campo titular, el dato ingresado
10	end;		//	Finaliza el IF
11	**end**;		//	Finaliza el proceso

Teniendo datos cargados podemos implementar **comparaciones** cómo: calcular máximos, mínimos o modificar contadores. Estas variables auxiliares (max, min, cant) deben inicializarse antes de empezar a llamar los procesos:

01	**procedure encontrarMaximo***(unNum: integer; unaMarca: string; var max: integer; var marcaMax: string);*		//	*unNum es el número actual que se utilizará para comparar con el máximo, unaMarca es la marca actual. max y marcaMax son los valores más grande almacenados.*
02	**begin**			
03	if(unNum > max)then begin		//	Comprueba si el valor actual es mayor al maximo actual
04	max:= unNum;		//	Si lo es, actualiza valor del máximo
05	marcaMax:= unaMarca;		//	Actualiza marca del valor máximo
06	**end**;			

También aplicable a **mínimos:**

01	**procedure encontrarMinimo***(unNum: integer; unaMarca: string; var min: integer; var marcaMin: string);*		//	*unNum es el número actual que se utilizará para comparar con el minimo, unaMarca es la marca actual. min y marcaMin son los valores más chicos almacenados.*
02	**begin**			
03	if(unNum < min)then begin		//	Comprueba si el valor actual es menor al mínimo actual
04	min:= unNum;		//	Si lo es, actualiza valor del mínimo
05	marcaMin:= unaMarca;		//	Actualiza marca del valor mínimo
06	**end**;			

Varios ejercicios piden encontrar dos de los anteriores:

01	**procedure encontrarMaximos***(unNum: integer; unaMarca: string; var max, max2: integer; var marcaMax, marcaMax2: string);*		*//*	*unNum es el número actual que se utilizará para comparar con el máximo, unaMarca es la marca actual. max y marcaMax son los valores más grande almacenados. max2 y marcaMax2 son los segundos valores mas grandes almacenados.*
02	**begin**			
03	if(unNum > max)then begin		//	Comprueba si el valor actual es mayor al maximo mas grande
04	max2:= max;		//	Si lo es, actualiza, maximo2 toma valor del mas grande anterior
05	marcaMax2:= marcaMax;		//	Actualiza, marcaMax2 toma marca del mas grande anterior
06	max:= unNum;		//	Actualiza valor, max toma el valor actual
07	marcaMax:= unaMarca;		//	Actualiza marca, marcaMax toma la marca mas grande actual
08	**end else**		//	Si el actual no es mas grande que el max mas grande
09	if(unNum > max2)then begin		//	Verifico si es mas grande que el segundo mas grande
10	max2:= unNum;		//	Si lo es, max2 toma valor actual
11	marcaMax2:= unaMarca;		//	marcaMax2 toma marca del segundo valor mas grande
12	end;			
13	end;			

01	**procedure encontrarMinimos***(unNum: integer; unaMarca: string; var min, min2: integer; var marcaMin, marcaMin2: string);*		*//*	*unNum es el número actual que se utilizará para comparar con el minimo, unaMarca es la marca actual. min y marcaMin son los valores más chicos almacenados. min2 y marcaMin2 son los segundo valores más chicos almacenados.*
02	**begin**			
03	if(unNum < min)then begin		//	Comprueba si el valor actual es menor al minimo mas chico
04	min2:= min;		//	Si lo es, actualiza, min2 toma valor del mas chico anterior
05	marcaMin2:= marcaMin;		//	Actualiza, marcaMin2 toma marca del mas chico anterior
06	min:= unNum;		//	Actualiza valor, min toma el valor actual
07	marcaMin:= unaMarca;		//	Actualiza marca, marcaMin toma la marca mas chica actual

08	**end else**		//	Si el actual no es mas chico que el minimo mas chico
09	if(unNum < min2)then begin		//	Verifico si es mas chico que el segundo mas chico
10	min2:= unNum;		//	Si lo es, min2 toma valor actual
11	marcaMin2:= unaMarca;		//	marcaMin2 toma marca del segundo valor mas chico
12	end;			
13	end;			

Modificar contadores:

01	**procedure verificarSiEs4***(unNum: integer;var cont4s: integer);*		//	*unNum es el número actual que se verifica, cont4s será el parametro que contara si el número cumple la condicion*
02	**begin**			
03	if(unNum = 4)then		//	Comprueba si el valor actual es igual a 4
04	cont4s:= cont4s +1;		//	Si lo es, suma al valor actual de cont4s, 1.
05	**end**;			

Además de estas validaciones hay funciones que suelen utilizarse para **cálculos, de sectores geométricos, promedios, porcentaje, incluso también verificaciones:**

01	**function calcularPerimetroCuadrado**(unaBase, unaAltura: integer): integer;		//	*unaBase seria valor de la base actual, unaAltura valor de la altura actual. Devuelve un valor integer.*
02	**begin**			
03	calcularPerimetroCuadrado:= (unaBase * unaAltura) * 2;		//	Asigna valor de retorno el cálculo de la base x altura x 2.
04	**end**;			

```
01  function calcularPromedio(cantTotal, cantCumple: integer): real;   //  cantTotal seria valor de elementos en total, cantCumple valor de elementos que cumplian un criterio. Devuelve un valor real.
02  begin
03  calcularPromedio:= cantCumple / cantTotal;                          //  Asigna valor de retorno el cálculo de los que cumplen dividido el total.
04  end;
```

```
01  function esMultiploDe5(unNum: integer): boolean;   //  unNum es un valor que se verificará. Devuelve un valor bool
02  begin
03  if(unNum MOD 5 = 0)then                            //  Verifica si el valor actual dividido 5 deja resto 0
04  esMultiploDe5:= true                                   Si lo hace, retorna true
05  else
06  esMultiploDe5:= false;                                 Sino, retorna false
07  end;
```

La misma función puede escribirse más directa:

```
01  function esMultiploDe5(unNum: integer): boolean;   //  unNum es un valor que se verificará. Devuelve un valor bool
02  begin
03  esMultiploDe5:= unNum MOD 5 = 0;                   //  Retorna true/false si esa comparación se cumple o no.
07  end;
```

Recordar que las funciones solo devuelven tipos de datos ordinales, si quisiera devolver un registro/ vector/ lista o cualquier tipo de estructura compuesta no se podrá, en ese caso debe utilizarse un procedimiento.

El siguiente módulo carga un **vector completo** hasta que el índice llegue al valor de su límite (dimensión física):

```
01  procedure cargarVector(var v: vector);                    //  v es un vector declarado
02  var
03  dato, i: integer;                                         //  Declaro i como índice y dato para ingresar valores
04  begin
05  for i:= 1 to dimF do begin                                //  Repito dimF veces (dimF es cantidad max del vector)
06  leerNum(dato);                                            //  Utilizo leerNum para pedir e ingresar un valor en dato
07  v[i]:= dato;                                              //  Almaceno el dato en el vector en la posición i
08  end;
```

El siguiente módulo carga un vector parcialmente (hasta que corte al ingresar un dato específico):

```
01  procedure cargarVector(var v: vector; var dimL:           //  v es un vector declarado. dimL es la cantidad de datos cargados
    integer);                                                     en el vector
02  var
03  dato: integer;                                            //  Declaro dato para ingresar valores
04  begin
05  dimL:= 0;                                                 //  Inicializo dim lógica en 0, puede hacerse fuera
06  leerNum(dato);                                            //  Utilizo leerNum para pedir e ingresar un valor en dato
07  while(dato <> -1)AND(dimL < dimF)do begin                  //  Mientras el valor ingresado no sea -1 y la dimL no llegue a dimF
08  dimL:= dimL +1;                                           //  Incrementa dim logica
09  v[dimL]:= dato;                                           //  Almacena en vector, en posición valor de dim lógica el dato
10  if(dimL < dimF)then                                       //  Verifica si dim lógica sigue siendo menor a la física
11  leerNum(dato);                                            //  Si lo es, ingresó un nuevo valor.
12  end;
13  end;
```

El siguiente proceso imprime un **vector de registros**, el cual, podría adaptarse a cualquier tipo de dato que almacena el vector.En este caso, imprime el vector completo:

01	**procedure imprimirVector**(v: vector);		*//*	*v es un vector declarado y cargado*
02	**var**			
03	i: integer;		//	Declaro i como índice
04	**begin**			
05	for i:= 1 to dimF do begin		//	Repito dimF veces (dimF es la cantidad física del vector)
06	write('Marca: ',v[i].marca,' patente: ',v[i].patente,' Nombre de titular: ',v[i].titular);		//	Imprime el mensaje concatenado con valores almacenados en el registro dentro del vector en la posición i actual
07	end;			
08	**end**;			

En caso de que el vector **no está cargado completamente,** necesitaremos utilizar el valor de su **dimensión lógica**:

01	**procedure imprimirVector**(v: vector; dimL: integer);		*//*	*v es un vector declarado y cargado. dimL la cantidad de elementos cargados*
02	**var**			
03	i: integer;		//	Declaro i como índice
04	**begin**			
05	for i:= 1 to dimL do begin		//	Repito dimL veces (dimL es cantidad de elementos del vector)
06	write('Marca: ',v[i].marca,' patente: ',v[i].patente,' Nombre de titular: ',v[i].titular);		//	Imprime el mensaje concatenado con valores almacenados en el registro dentro del vector en la posición i actual
07	end;			
08	**end**;			

Además de los procesos anteriores existen métodos para **ordenar** el vector por cierto criterio. El siguiente es el **método por Selección**, que recorre el vector completo (y en cada ejecución verifica desde su siguiente posición hasta el final hasta encontrar donde debería quedar según el ordenamiento).
En este caso es ordenamiento de forma **ascendente** (de menor a mayor):

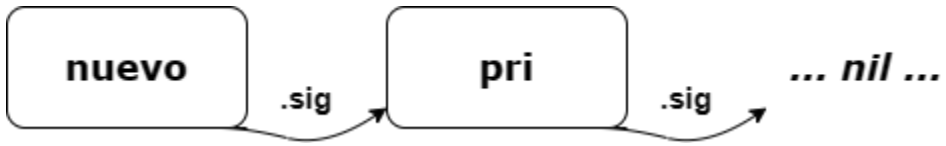
01	**procedure ordenarPorSeleccion**(var v: vector);		//	*v es un vector declarado y cargado*
02	**var**			
03	i, j, pos: integer;		//	Declaro i, j como índices pos para auxiliar
04	aux: elementoVector;		//	Declaro aux para guardar valores momentaneamente
05	**begin**			
06	for i:= 1 to dimF-1 do begin		//	Repito dimF-1 veces (dimF es cantidad max del vector)
07	pos:= i;		//	Pos guarda posición actual del vector
08	for j:= i+1 to dimF do		//	Recorre desde la siguiente posición (de la pos actual)hasta el final del vector
09	if(v[j] < v[pos]) then		//	Verificando si la actual(j) es menor a la actual original(i)
10	pos:= j;		//	Si lo es, guarda en pos el índice del elemento más chico
117	aux:= v[pos];		//	Terminado el for, aux guarda el valor del valor mas chico del vector
12	v[pos]:= v[i];		//	En el lugar de pos, intercambia valor con i (elemento original)
13	v[i]:= aux;		//	En el lugar de i, intercambio valor con pos (valor mas chico)
14	end;			
15	**end**;			

Listas:

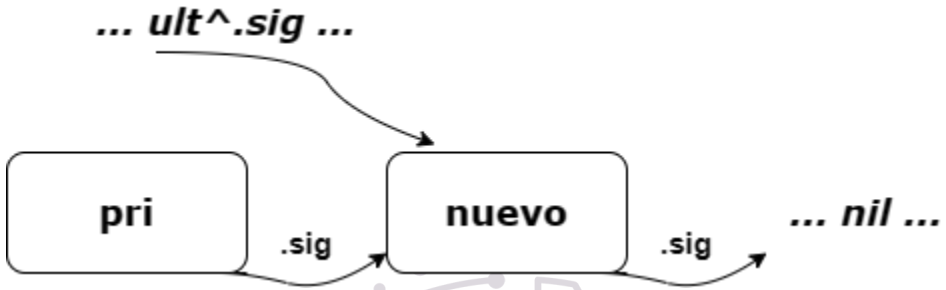
Formas de **agregar dato** a una lista:

01	**procedure insertarAdelante***(var pri: lista; num: integer);*		//	*pri es la lista principal y num el dato a cargar en el nodo*
02	**var** nuevo: lista		//	Creo un nodo auxiliar para agregar a la lista principal
03	**begin**			
04	new(nuevo);		//	Creo el nodo nuevo
05	nuevo^.dato:= num;		//	El dato / elemento del nodo tomará el valor del dato que ingresa
06	nuevo^.sig:= pri;		//	El nodo apunta a la lista principal (se posiciona a la izquierda)
07	pri:= nuevo;		//	La lista principal a hora se pisa con el nuevo nodo para agregar este mismo a la lista
08	**end**;			

- **Si la lista principal ingresa vacía, el primer nodo se posiciona y queda apuntando a vacío (se convierte en el primer nodo de la lista)**



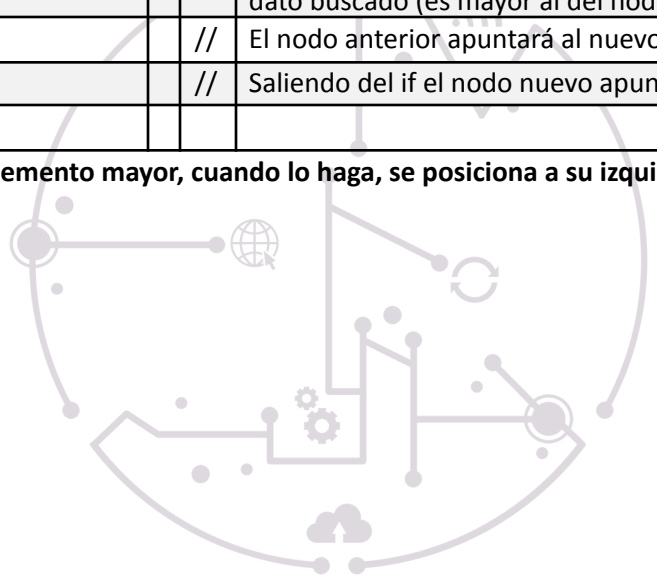
01	***procedure insertarAtras****(var pri, ult: lista; num: integer);*		*//*	*pri es la lista principal, ult es el punter a nuestro último nodo de la lista y num el dato a cargar en el nodo*
02	**var**			
03	nuevo: lista ;		//	Creo un nodo auxiliar para agregar a la lista principal
04	**begin**			
05	new(nuevo);		//	Creo el nodo nuevo (actualmente nil)
06	nuevo^.dato:= num;		//	El dato / elemento del nodo tomará el valor del dato que ingresa
07	nuevo^.sig:= nil;		//	El nodo apunta a vacío (se convierte en el último al final de la lista)
08	If (pri <> nil) **then**		//	Evaluó si la lista principal tiene algo
09	ult^.sig:= nuevo;		//	Si SI tiene algo, mi puntero de la última posición apuntará al nuevo nodo
10	**else**			
11	pri:= nuevo;		//	Si NO tiene nada, se carga en la lista principal el nuevo nodo
12	ult:= nuevo;		//	Mi puntero ult toma los valores del último nodo que se agrega a la lista.
13	**end**;			

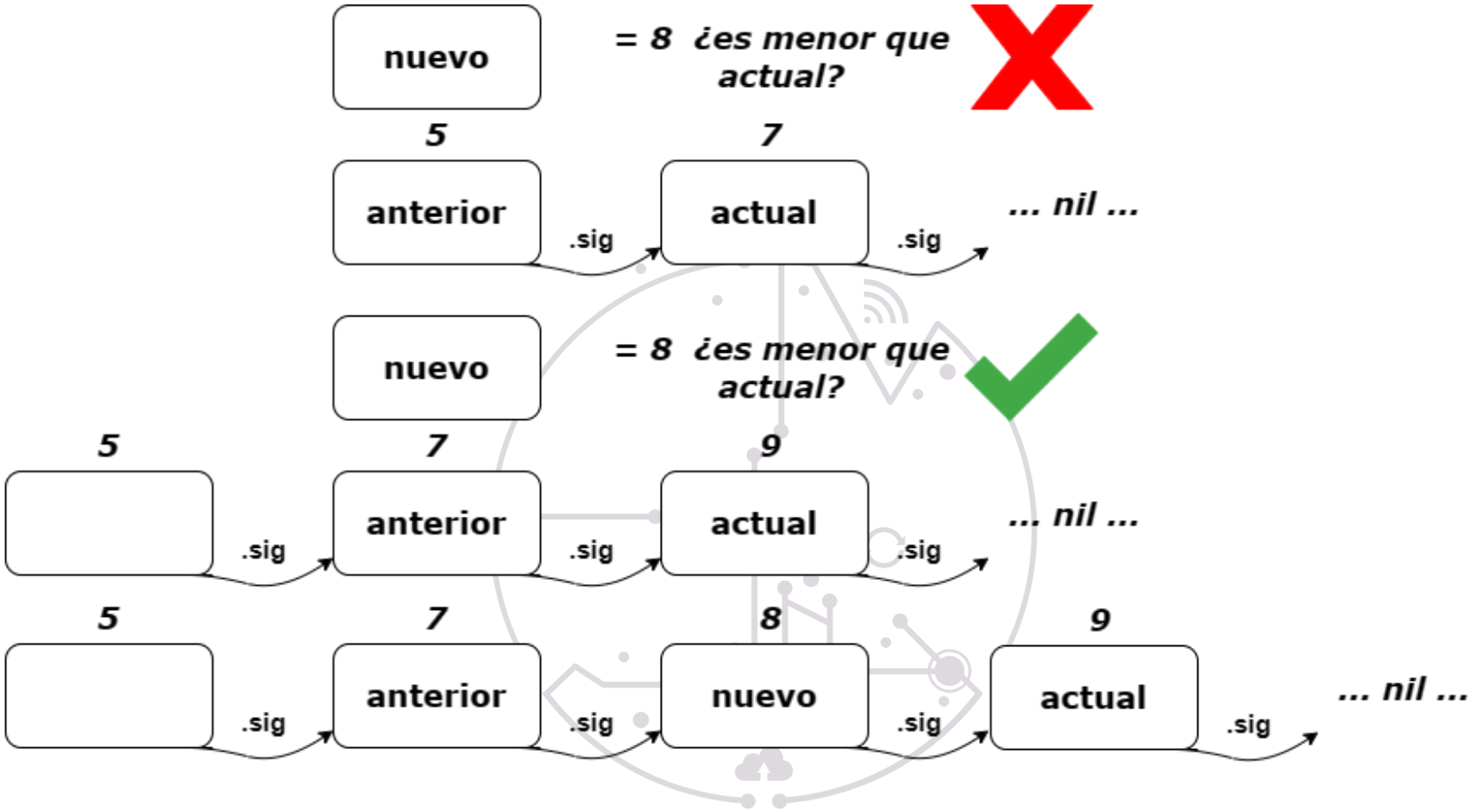


01	***procedure insertarOrdenado****(var pri: lista; num: integer);*		*//*	*pri es la lista principal y num el dato a cargar en el nodo*
02	**var**			
03	nuevo, anterior, actual: lista;		//	Creo tres nodos auxiliares para ordenar la lista principal
04	**begin**			
05	new(nuevo);		//	Creo el nodo nuevo
06	nuevo^.dato:= num;		//	El dato / elemento del nodo tomará el valor del dato que ingresa
07	actual:= pri;		//	El nodo actual adquiere valor del nodo entrante
08	anterior:= pri;		//	El nodo anterior adquiere valor del nodo entrante
09	while (actual <> nil) and (act^.dato < num) **do begin**		//	Utilizando actual como nodo para recorrer la lista principal y verificando si el dato ingresado es mayor al del nodo actual
10	anterior:= actual;		//	El nodo anterior adquiere valores de actual para no perder el nodo y seguir avanzando

11	actual:= actual^.sig;		//	El nodo actual adquiere valor del siguiente nodo de la lista para seguir verificando datos de la lista
12	**end**;			
13	If (anterior = actual)**then begin**		//	Caso que la lista está vacía (la posición buscada es la primera o no hay nada cargado)
14	nuevo^.sig:= nil;		//	El nuevo nodo apunta a vacío
15	pri:= nuevo; **end**		//	Inserto el nodo nuevo en la lista
16	else		//	Sino es el primer nodo, significa que salió porque el nodo siguiente contiene el dato buscado (es mayor al del nodo actual)
17	anterior^.sig:= nuevo;		//	El nodo anterior apuntará al nuevo nodo
18	nuevo^.sig:= actual;		//	Saliendo del if el nodo nuevo apuntará al actual (nodo con dato mayor)
19	**end**;			

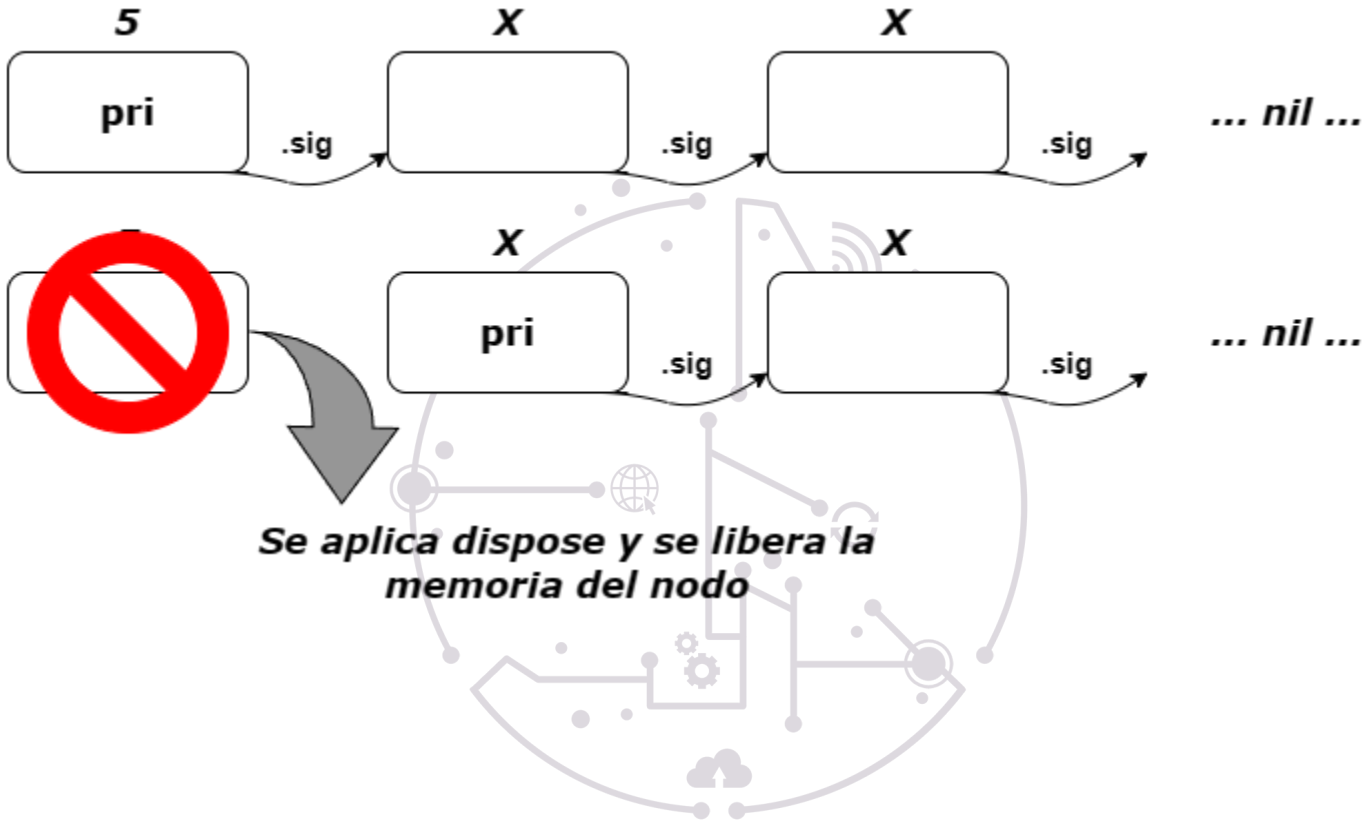
- **Recorre la lista hasta encontrar un elemento mayor, cuando lo haga, se posiciona a su izquierda**



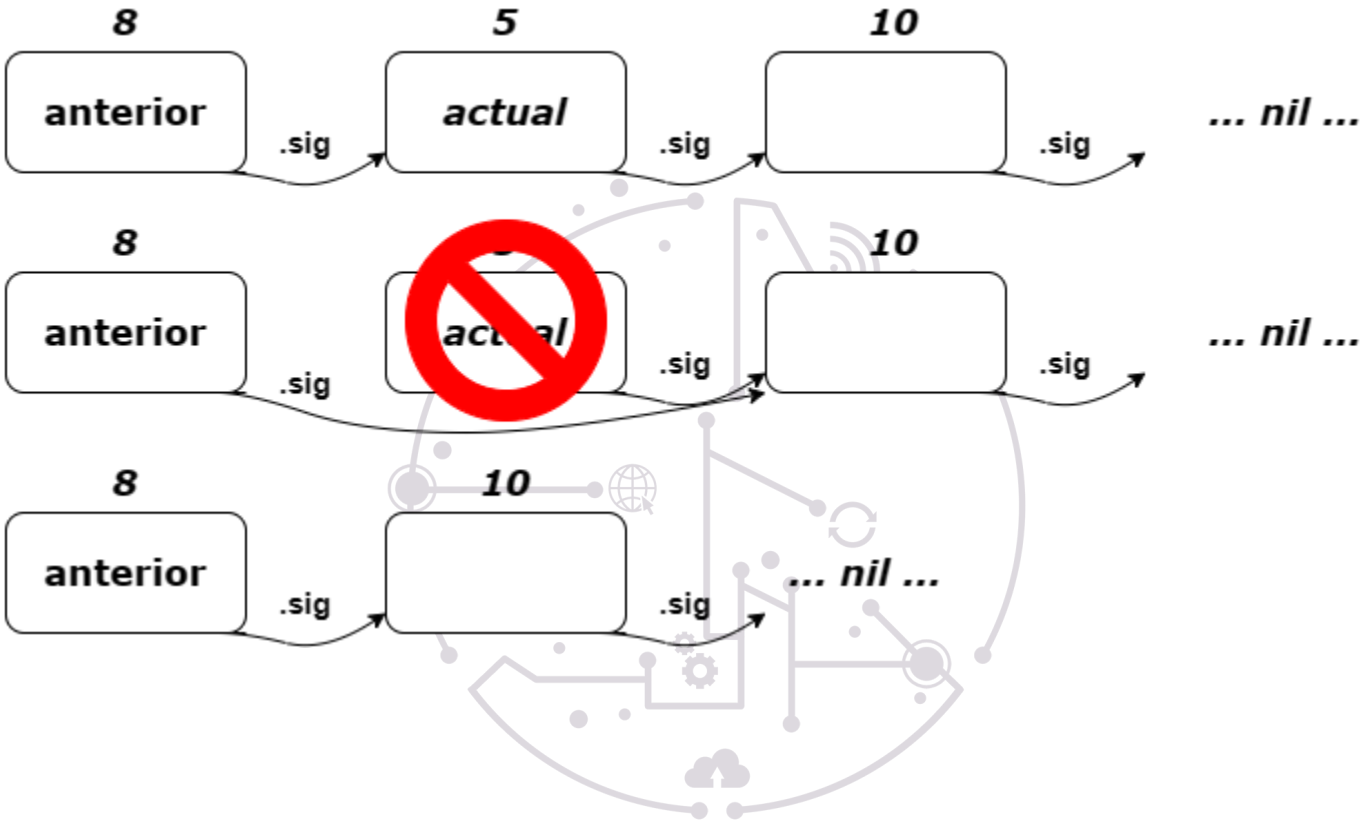

nuevo
= 8 ¿es menor que actual?
5
7
anterior
.sig
actual
.sig
... nil ...
nuevo
= 8 ¿es menor que actual?
5
7
9
.sig
anterior
.sig
actual
.sig
... nil ...
5
7
8
9
.sig
anterior
.sig
nuevo
.sig
actual
.sig
... nil ...

01	***procedure eliminarElemento****(var pri: lista; num: integer);*		*//*	*pri es la lista principal y num es el dato a buscar y eliminar*
02	**var**			
03	anterior, actual: lista;		//	Creo dos nodos auxiliares para ordenar la lista principal
04	**begin**			
05	actual:= pri;		//	Le doy a actual los valores de la lista principal
06	while (actual <> nil) and (actual^.dato <> num) **do begin**		//	Mientras no termine de recorrer la lista o haya encontrado el dato a eliminar
07	anterior:= actual;		//	El nodo anterior toma el nodo de actual para no perder el enlace
08	actual:= actual^.sig;		//	El nodo anterior adquiere valor del nodo entrante
09	**end;**			
10	if (actual <> nil) **then begin**		//	Si actual tiene un dato no vacío, significa que no termine de recorrer la lista, por lo tanto encontré el dato deseado
11	if (actual = pri) **then**		//	Si actual es igual a pri significa que el dato del nodo es el que buscaba
12	pri:= pri^.sig		//	El primer nodo de pri tomará valores del siguiente nodo (sus datos y direccionamiento)
13	**else**		//	Si no es el primer nodo significa que lo encontre en otra posición
14	anterior^.sig:= actual^.sig;		//	Hago que el nodo anterior apunte al nodo siguiente del actual
15	**end**;		//	
16	dispose(actual);		//	Elimino la posición del nodo ‘actual’
17	**end;**			

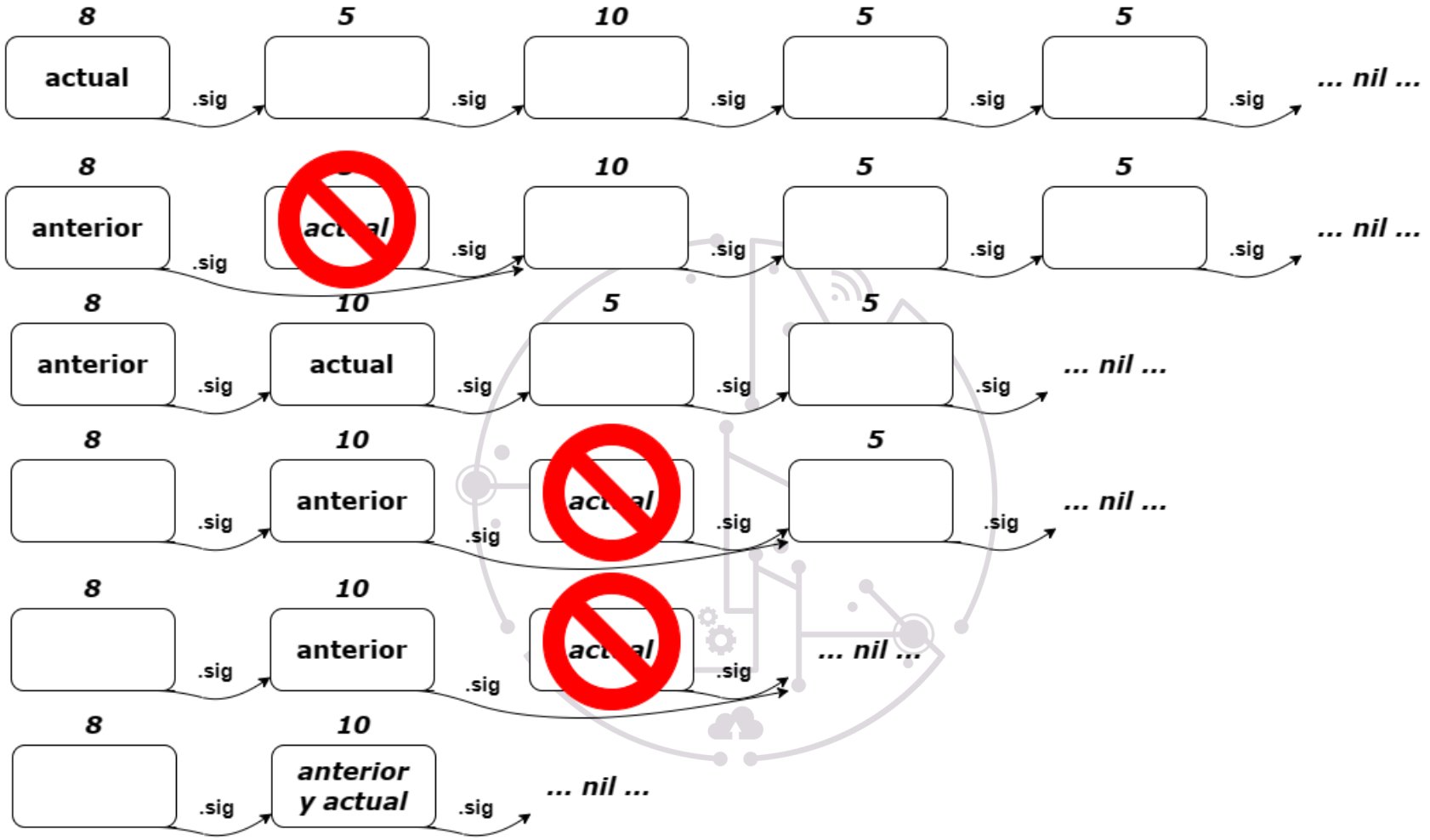
Caso encontre en el primer nodo de la lista - Buscando el nodo que contenga el valor 5



Caso no está en el primer nodo - Buscando el nodo que contenga el valor 5



```
01  procedure eliminarElementoRep(var pri: lista; num: integer);   //  pri es la lista principal y num es el dato a buscar y eliminar
02  var
03  anterior, actual: lista ;                                      //  Creó dos nodos auxiliares para ordenar la lista principal
04  begin
05  actual:= pri;                                                  //  Le doy a actual los valores de la lista principal
06  anterior:= pri;                                                //  Anterior copia valores de la lista principal
07  while (actual <> nil) do begin                                 //  Mientras no termine de recorrer la lista
08    if(actual^.dato <> num)then begin                            //  Verifico si no es un dato que debo eliminar
09      anterior:= actual;                                         //  El nodo anterior toma el nodo de actual para no perder el enlace
10      actual:= actual^.sig;                                      //  El nodo anterior adquiere valor del nodo entrante
11    end
12    else begin                                                   //  Si si debo eliminarlo
13      if (actual = pri) then begin                               //  Confirmo si es el primer nodo de la lista
14        pri:= pri^.sig;                                          //  Asignó a pri el nodo siguiente
15      else                                                       //  Significa que no es el primer nodo
16        anterior^.sig:= actual^.sig;                             //  Hago que el nodo anterior apunte al nodo siguiente del actual
17      dispose(actual)                                            //  Libero la memoria en donde se encuentra actual
18    actual:= anterior;                                           //  Asignó a actual el nodo anterior para seguir recorriendo
19  end;                                                           //
```


8
actual
.sig
5
.sig
10
.sig
5
.sig
5
.sig
... nil ...
8
anterior
.sig
actual
.sig
10
.sig
5
.sig
5
.sig
... nil ...
8
anterior
.sig
10
actual
.sig
5
.sig
5
.sig
... nil ...
8
.sig
10
anterior
.sig
actual
.sig
5
.sig
... nil ...
8
.sig
10
anterior
.sig
actual
.sig
... nil ...
8
.sig
10
anterior
y actual
.sig
... nil ...