CONCEPTOS DE ALGORITMOS DATOS Y PROGRAMAS

31/03/2024 - 13/06/2024

Tipos de datos

Simple

Aquellos que toman un único valor, en un momento determinado, de todos los permitidos para ese tipo.

Compuesto

Pueden tomar varios valores a la vez que guardan alguna relación lógica entre ellos, bajo un único nombre.

Existen varios tipos de datos en Pascal
Entre ellos tenemos

- Entero
- Real
- Boolean
- Char
- String
- Record
- Array
- Puntero
- Lista

Definido por el Lenguaje

Son provistos por el lenguaje y tanto la representación como sus operaciones y valores son reservadas al mismo.

Entero

Tipo de dato simple, ordinal
Puede tomar valores tanto negativos como positivos
Además, poseen un número mínimo y uno máximo
Se declara como “integer”

Real

Tipo de dato simple
Permiten representar números con decimales
Posee un número mínimo y uno máximo
Se declara como “real”

Lógico

Tipo de dato simple, ordinal
Los valores que puede tomar son
TRUE = Verdadero
FALSE = Falso
Se declara como “boolean”

Char

Tipo de dato simple, ordinal
Los valores que pueden tomar son caracteres
Se declara como “char”

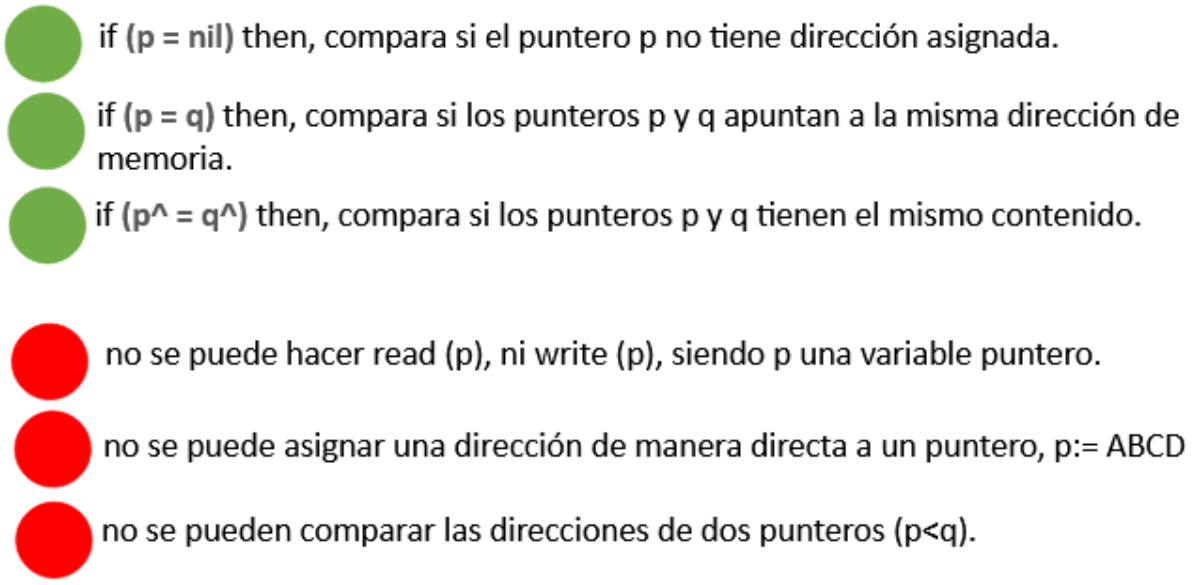
String

Tipo de dato compuesto
Los valores que pueden tomar son cadenas de caracteres
Se declara como “string”

Puntero

Tipo de dato simple
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).
Se declara como “^”
Ej: Puntero = ^Nodo;

Para reservar memoria dinámica utilizamos la instrucción “new (VARIABLE);”
Para liberar la memoria dinámica utilizamos la instrucción “dispose (VARIABLE);”



Definido por el Programador

Permiten definir nuevos tipos de datos a partir de los tipos simples.
Un tipo de dato definido por el usuario es aquel que no existe en la definición del lenguaje, y el programador es el encargado de su especificación.

Declaración

```
program uno;
const
…
módulos
var
  x: identificador;
  …
begin
  …
end.
```

Caso

```
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 [X]
end.
```

Subrango

Es un tipo ordinal que consiste de una sucesión de valores de un tipo ordinal (predefinido o definido por el usuario) tomado como base.
Es simple y ordinal

Caso

```
program uno;
const
...
typel
  notas = 0..10;
  letras = 'a'..'l';
var
```

```
  not1,not2: notas;
  let:letras;
begin
  …
end.
```

Operaciones permitidas
- Asignación
- Comparación
- Todas las operaciones permitidas para el tipo base

Operaciones NO permitidas
- Depende del tipo base

Programa 1: Se pide realizar que lea caracteres (suponga sólo minúsculas) hasta leer el carácter ‘z’. Al finalizar informar cuántos caracteres leídos estaban entre ‘a’ y ‘h’; cuántos entre ‘i’ y ‘n’ y cuántos entre ‘ñ’ y ‘y’.

```
program uno;
type
  letras = ‘a’..’z’;
var
  primer,segundo,tercer:integer;
  letra:letras;
begin
  primer:= 0;  segunda:=0; tercer:=0;
  read (letra);
while (letra <> ‘z’) do begin
  case letra of
    ‘a’..’h’: primer:= primer + 1;
    ‘i’..’n’: segunda:= segunda + 1;
    ‘ñ’..’y’: tercer:= tercer + 1;
  end;
read (letra);
end;
write (primer,segunda,tercer);
end.
```

Registros

Es un tipo de datos estructurado, que permite agrupar diferentes clases de datos en una estructura única bajo un sólo nombre
Una manera natural y lógica de agrupar los datos de cada perro en una sola estructura es declarar un tipo **REGISTRO** asociando el conjunto de datos de cada uno.
La característica principal es que un registro permite representar la información en una única estructura.

```
Program uno;
Const
  ….
Type
```

```
  perro = record
   raza: string;
   nombre: string;
   edad: integer;
end;
Var
   ani1, ani2: perro;
```

La única operación permitida es la asignación entre dos variables del mismo tipo
Puedo realizar las operaciones permitidas según el tipo de campo del registro
La única forma de acceder a los campos es variable.nombrecampo

```
Begin
 ….
 ani2:= ani1;
 …
End.
```

Lectura de campos de un registros

```
Begin
  leer (ani1);
  write (an1.raza);
  write(ani1.nombre);
  write(ani1.edad);
End.
```

Cuando tenemos un registro de registros, tenemos que leer los datos del segundo registro PRIMERO y luego, asignarle al campo del registro principal el registro directamente.
Podemos leerlo usando un proceso, como lo hacemos normalmente, devolverlo y asignarle lo leído en el campo del registro declarado.

Vectores

Los arreglos son, como vagones que cargan cosas (datos), cada vagón tiene una posición, la cual es ORDINAL.
Para acceder a la posición de un arreglo utilizaremos los corchetes “[]”
entonces…

Declaración:

```
Type
        vnumeros = array [1..10] of integer;

For i:=1 to 10 do begin
        vnumeros[i]:=0;
end;
```

Ponemos en cada posición del arreglo “vnumeros”, un 0

```
vnumeros[1] = 0
…
vnumeros[10] = 0
```

Si tenemos que acceder al campo, de un registro dentro de un arreglo y queremos imprimir en las 10 posiciones del vector, lo que haya en el campo del registro declarado, haríamos lo siguiente:

```
For i:=1 to 10 do begin
        writeln(vdatos[i].campo);
end;
```

Así de sencillo, accedimos a la posición de “i” del arreglo, y al campo del registro previamente declarado (En este caso es un ejemplo)

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.

En memoria estática se declara una variable tipo PUNTERO (ya que son las únicas que pueden almacenar direcciones). La dirección almacenada en esa variable representa la dirección donde comienza la lista. Inicialmente ese puntero no contiene ninguna dirección.
Luego a medida que se quiere agregar elementos a la lista (nodo), se reserva una dirección de memoria dinámica y se carga el valor que se quiere guardar.
El último nodo de la lista indica que la dirección que le sigue es nil.

Estructura de datos

Permite al programador definir un tipo al que se asocian diferentes datos que tienen valores lógicamente relacionados y asociados bajo un nombre único.

Homogénea: Los elementos que la componen son del mismo tipo

Heterogénea: Los elementos que la componen pueden ser de distinto tipo

Estática: El tamaño de la estructura no varía durante la ejecución del programa

Dinámica: El tamaño de la estructura puede variar durante la ejecución del programa

Secuencial: Para acceder a un elemento particular se debe respetar un orden predeterminado, por ejemplo, pasando por todos los elementos que le preceden, por ese orden

Directo: Se pueden acceder a un elemento particular, directamente, sin necesidad de pasar por los anteriores a él, por ejemplo, referenciando una posición

Lineal: Está formada por ninguno, uno o varios elementos que guarden una relación de adyacencia ordenada donde a cada elemento y le sigue uno y le precede uno, solamente

No lineal: Para un elemento dado pueden existir 0, 1 ó más elementos que le suceden y 0, 1 ó más elementos que le preceden

Operaciones

Operadores matemáticos

Tenemos:

- Suma (+)
- Resta (-)
- Multiplicación (*)
- División (/)

Operadores lógicos

- Menor (<)
- Mayor (>)
- Distinto (<>)
- Igual (=)
- Menor Igual (<=)
- Mayor Igual (=>)

Operadores enteros

- DIV (Devuelve el cociente entero de la división)
- MOD (Devuelve el resto de la división)

Variables

Es una zona de memoria cuyo contenido va a ser alguno de los tipos mencionados anteriormente. La dirección inicial de esta zona se asocia con el nombre de la variable.
Puede cambiar su valor durante el programa.

Constantes

Es una zona de memoria cuyo contenido va a ser alguno de los tipos mencionados anteriormente. La dirección inicial de esta zona se asocia con el nombre de la variable.
NO puede cambiar su valor durante el programa.

Estructura del programa

```
program nombre;
const
…
módulos {luego veremos cómo se declaran}
var
begin
…
end.
```

```
program nombre;
const
   N = 25;
   pi = 3.14;
módulos {luego veremos cómo se declaran}
var
edad: integer;
peso: real;
letra: char;
resultado: boolean;
begin
  edad:= 5;
  peso:= -63.5;
  edad:= edad + N;
  letra:= ‘A’;
  resultado:= letra = ‘a’;
end.
```

PRE y POST condición

Pre-Condición: Es la información que se conoce como verdadera antes de iniciar el programa o módulo

Post-Condición: Es la información que debería ser verdadera al concluir el programa o módulo, si se cumplen adecuadamente los pasos especificados

READ y WRITE

READ: Es una operación que contienen la mayoría de los lenguajes de programación. Se usa para tomar datos desde un dispositivo de entrada (por defecto desde teclado) y asignarlos a las variables correspondientes

```
program uno;
var
   cant: integer;
begin
  read (cant); ← Se lee el valor para la variable “cant”
end.
```

WRITE: Es una operación que contienen la mayoría de los lenguajes de programación. Se usa para mostrar el contenido de una variable, por defecto en pantalla

```
program uno;
var
   cant: integer;
begin
  read (cant); ← Se lee el valor para la variable “cant”
  cant:= cant + 1; ← Se le suma 1 a lo leído en la línea anterior
  write (cant);  ← Se escribe el valor de la variable “cant”
end.
```

Estructuras de control

- Secuencia
- Decisión
- Iteración
- Repetición
- Selección

Secuencia

La estructura de control más simple, está representada por una sucesión de operaciones (por ej. asignaciones), en la que el orden de ejecución coincide con el orden físico de aparición de las instrucciones.

Decisión

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

Realice un programa que lea dos números enteros e informe si la suma de los mismos es mayor a 20.

```
program uno;
var
  num1,num2,suma:integer;
begin
  read (num1);
  read (num2);
  suma:= num1 + num2;
if (suma > 20) then
  write (“La suma supera 20”)
else
  write (“La suma NO supera 20”);
end.
```

Otra forma:

```
program uno;
```

```
var
  num1,num2:integer;
begin
  read (num1);
  read (num2);
if ((num1+num2) > 20) then
  write (“La suma supera 20”)
else
  write (“La suma NO supera 20”);
end.
```

Realizar un programa que lea un carácter y al finalizar informe si se leyó un carácter mayúscula, minúscula, dígito, y ó especiales ha leído.

```
program uno;
var
  car:char;
begin
  read (car);
case car of
  ‘a’.. ‘z’: write (“minúscula”);
  ‘A’.. ‘Z’: (“mayúscula”);
  ‘0’.. ‘9’: (“dígito”);
else (“especial”);
end;
end.
```

Iteración

Puede ocurrir que se desee ejecutar un bloque de instrucciones desconociendo el número exacto de veces que se ejecutan.
Para estos casos existen en la mayoría de los lenguajes de programación estructurada las estructuras de control iterativas condicionales.
Como su nombre lo indica las acciones se ejecutan dependiendo de la evaluación de la condición.
Estas estructuras se clasifican en pre-condicionales y post-condicionales

Pre-condicional: Evalúan la condición y si es verdadera se ejecuta el bloque de acciones. Dicho bloque se pueda ejecutar 0, 1 o más veces.
Importante: el valor inicial de la condición debe ser conocido o evaluable antes de la evaluación de la condición

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

Realizar un programa que lea códigos de productos hasta leer un código igual a 30. Al finalizar informe la cantidad de productos con código par.

```
program dos;
var
  prod:integer;
  total:integer;
begin
  total:=0;
  read (prod);
while (prod <> 30) do begin
  if (prod MOD 2 = 0) then
  total:= total + 1;
  read (prod);
end;
write (total);
end.
```

Post-condicional: Ejecutan las acciones luego evalúan la condición y ejecutan las acciones mientras la condición es falsa. Dicho bloque se pueda ejecutar 1 ó más veces

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

Realizar un programa que lea códigos de productos hasta leer un código igual a 30. Al finalizar informe la cantidad de productos con código par. El último producto debe procesarse.

```
program uno;
var
  prod:integer;
  total:integer;
begin
  total:=0;
repeat
  read (prod);
  if (prod MOD 2 = 0) then
    total:= total + 1;
until (prod = 30); ← Se ejecuta cuando la condición es falsa
write (total);
end.
```

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
La variable índice debe ser de tipo ordinal
La variable índice no puede modificarse dentro del lazo
La variable índice se incrementa y decrementa automáticamente
Cuando el for termina la variable índice no tiene valor definido

Realizar un programa que lea precios de 10 productos que vende un almacén. Al finalizar informe la suma de todos los precios leídos.

```
program uno;
var
  precio,total:real;
  i:integer;
begin
  total := 0;
for i:= 1 to 10 do begin
  read (precio);
  total:= total + precio;
end;
  write (“La suma de los precios de los productos del almacén son: ”,total);
end.
```

Máximos y Mínimos

Una explicación simple para los máximos y mínimos, son variables que toman valores dependiendo de una estructura de condición, en caso de que se lea un número que cumpla con la condición del máximo o del mínimo, debemos actualizar la variable correspondiente con el valor correspondiente

Realizar un programa que lea número de alumno y promedio hasta leer un promedio igual a 0. Al finalizar informar el promedio más alto.

```
program uno;
var
  prom:real;alu:integer;
begin
  read(prom);
  read(alu);
  max:=-1; ← Muy importante inicializar los máximos en un valor MUY bajo
while (prom <> 0) do begin
  read(prom);
  read(alu);
end;
write (“El mejor promedio es:”,max);
end.
```

Realizar un programa que lea número de alumno y promedio hasta leer un promedio igual a 0. Al finalizar informar el promedio más bajo.

```
program uno;
var
  prom:real;alu:integer;
```

```
  min:real;
begin
  read(prom);
  read(alu);
  min:=11 ← Recordar inicializar los mínimos en el valor más alto posible
while (prom <> 0) do begin
  If (prom <= min) then
    min:= prom;
    read(prom);
    read(alu);
end;
write (“El mejor promedio es:”,min);
end.
```

Modularización

Significa dividir un problema en partes funcionalmente independientes, que encapsulan operaciones y datos.

No se trata simplemente de subdividir el código de un sistema de software en bloques con un número de instrucciones dado.

Separar en funciones lógicas con datos propios y datos de comunicación perfectamente especificados.

Módulo

Tarea específica bien definida se comunican entre sí adecuadamente y cooperan para conseguir un objetivo común.
Encapsula acciones tareas o funciones.
En ellos se pueden representar los objetivos relevantes del problema a resolver.
Existen diferentes metodologías para usarlos en los programas en particular nosotros usaremos la METODOLOGÍA TOP-DOWN

Ejemplo estructura de un proceso:

```
Programa nombre
areas
  …
Procesos
  proceso nombre (parámetros)
  variables ← Variables locales
  comenzar
```

```
  … ← Código del procedimiento
  fin

Variables

…

comenzar

… ← Código del programa principal
fin
```

Un efecto de la modularización es una mayor claridad para leer y comprender el código fuente. El ser humano maneja y comprende con mayor facilidad un número limitado de instrucciones directamente relacionadas.

Procedure

Conjunto de instrucciones que realizan una tarea específica y retorna 0, 1 ó más valores.
No podremos invocarlas dentro de estructuras de control, evitar hacer:
if (procesoleer) X
Se pueden declarar procesos dentro de otros procesos, si, se pueden

```
Program uno;
Const
  ....
Type
  ....
procedure auxiliar;
Var
  x:integer;
begin
  x:=8;
end;
Var
  ...
Begin
  auxiliar; ← Se llama por su nombre
End.
```

Function

Conjunto de instrucciones que realizan una tarea específica y retorna 1 valor de tipo simple.
tipo debería ser un tipo de dato simple
Para retornar el valor la última instrucción de la función debe ser asignarle a su nombre el valor que se quiere retornar
¡Mala práctica, usar parámetros de referencia en la función! X

```
Program uno;
```

```
Const
  ....
Type
  ....
function auxiliar (parámetros): tipo;
Var
 x:integer;
begin
 x:=8;
 …
 auxiliar:= valor que se quiere retornar;
end;
Var
 ....
Begin
  …
End.
```

Invocar una función utilizando una variable:

```
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.
```

Invocación de una función usando estructuras de control:

```
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.
```

Invocación de una función utilizando el 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.
```

Alcance de las variables

```
Program alcance;
Var
   a,b: integer; ← Variables globales al programa, se pueden usar por todo lo que este
debajo del código
procedure prueba;
Var
  c: integer; ← Variables locales al PROCESO, se pueden usar solo en el proceso
Begin
End.
Var
  d:integer; ← Variables del programa principal, solo las utiliza el cuerpo del programa
Begin
End.
```

Si es una variable utilizada en un proceso

- Se busca si es una variable local
- Se busca si es un parámetro
- Se busca si es una variable global al programa

Si es una variable usada en un programa

- Se busca si es una variable local al programa
- Se busca si es una variable global al programa

Comunicación entre módulos

Parámetros: La solución a estos problemas ocasionados por el uso de variables globales es una combinación de ocultamiento de datos (Data Hiding) y uso de parámetros.

El ocultamiento de datos significa que los datos exclusivos de un módulo NO deben ser "visibles" o utilizables por los demás módulos.

El uso de parámetros significa que los datos compartidos se deben especificar como parámetros que se transmiten entre módulos.

Parámetros por valor: El módulo recibe un valor, puede realizar operaciones y/o cálculos, pero **no producirá ningún cambio** ni tampoco tendrá incidencia fuera del módulo

Parámetros por referencia: El módulo reciba una dirección, puede realizar operaciones y/o cálculos, que **producirán cambios** y tendrán incidencia fuera del módulo

Corte de control

Cuando nos piden, que se lea, se procese o se informe dicha cantidad, como si fuese una condición de corte
debemos considerar lo siguiente:
¿Cuál es la condición de corte?
¿Debo heredar la condición?
¿Puedo usar booleans?

Los puntos más importantes son el 1 y 2 para la mayoría de ejercicios:

While (DATO-A-PROCESAR <> CONDICIÓN) do begin

While (DATO-A-PROCESAR <> CONDICIÓN) AND (OTRO-DATO = CONDICIÓN-2) do begin

…

entonces, ahí entendemos lo que es una “condición heredada” o “while anidado”, teniendo en cuenta cual es la condición de corte.

Dimensiones

Física

Se especifica en el momento de la declaración y determina su ocupación máxima de memoria.
La cantidad de memoria total reservada no variará durante la ejecución del programa.

Lógica

Se determina cuando se cargan contenidos a los elementos del arreglo.
Indica la cantidad de posiciones de memoria ocupadas con contenido real. Nunca puede superar la dimensión física
La dimensión lógica es una variable, a comparación de la física que es una constante, entonces, si nosotros queremos almacenar a lo sumo, N elementos, debemos contar la cantidad REAL de elementos en nuestra estructura (vector), podremos tener un vector de

100 espacios, pero de esos 100, ocupar solo 80, entonces nuestra Dimensión Lógica (real), es 80, no 100.

Vectores (Agregar)

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
Pasos a tener en cuenta:

1. Verificar que hay espacio, la dimensión lógica debe ser menor a la dimensión física
2. Agregar al final de los elementos ya existentes el elemento nuevo
3. Incrementar la cantidad de elementos actuales

Ejemplo en código:

```
PUDE:=FALSE;
if (DIML+1 <=  DIMF) then
        begin
        DIML:=DIML+1;
        V[DIML]:=ELEMENTO;
        PUDE:=TRUE;
        end
end;
```

Cuando devuelve verdadero, es porque pude agregar al vector el elemento, siempre recordando que la dimensión lógica se incrementa.

Vectores (Insertar)

Significa agregar en el vector, en una posición determinada, siempre y cuando esta sea válida ya que no puedo insertar en una posición inexistente (fuera de rango)

1. Verificar que hay espacio, la dimensión lógica debe ser menor o igual a la dimensión física
2. Verificar que la posición sea válida (Entre los valores de dimensión definida (física) y la dimensión lógica)
3. Hacer lugar para insertar el elemento, con un corrimiento hacia algún lado
4. Incrementar la cantidad de elementos actuales

Ejemplo en código:

```
VAR
i:integer;
PUDE:=FALSE;
if (DIML < DIMF) AND (POS >= 1) AND (POS <= DIML) then
begin
        for i:=DIML downto POS do begin
        V[i+1]:=V[i];
        end;
PUDE:=TRUE;
V[POS]:=ELEMENTO; ← En la posición que quería insertar el elemento, lo inserto
DIML:=DIML+1;
```

```
end
end;
```

Vectores (Eliminar)

Significa eliminar (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 que el elemento no exista

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

Ejemplo en código:

```
VAR
i:integer;
begin
	PUDE:=FALSE;
	if(POS >= 1) AND (POS <= DIML)  then begin
		for ii:=POS to DIML-1 do begin
			V[i]:=V[i+1];
		end;
		PUDE:=TRUE;
		DIML:=DIML-1;
	end
end;
```

Vectores (Búsqueda)

Significa recorrer el vector para buscar un elemento, este puede o no estar y, se debe tener en cuenta que no es lo mismo buscar en un vector ordenado que desordenado

Ordenado

Se debe recorrer el vector teniendo en cuenta el orden:

- Búsqueda mejorada
- Búsqueda binaria

Búsqueda mejorada

1. Inicializar la búsqueda en la posición “1”
2. Mientras el elemento buscado sea menor al valor en el vector, y no termine de recorrer, sigo avanzando en el vector, +1 a la posición
3. Determinar la condición por la cual terminó el while y devolver el resultado

Ejemplo en código

```
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 <> Vector[medio]) y (inf <= sup))

   Si (El elemento buscado sea < Vector[medio]) entonces

   Actualizo primero

   Sino

   Actualizo último

   Calculo nuevamente el medio

3. Determinar por cuál condición se ha terminado el while y devolver el resultado.

Ejemplo en código:

```
Function dicotómica (vec: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;
  dicotómica:= ok;
end.
```

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

4. Inicializar la búsqueda en la posición “1”
5. Mientras el elemento buscado no sea igual al valor en el vector, y no termine de recorrer, sigo avanzando en el vector, +1 a la posición
6. Determinar la condición por la cual terminó el while y devolver el resultado

Ejemplo en código:

```
function buscar (a :números; dL:integer; valor:integer): boolean;
Var
  pos:integer;
  esta:boolean;
Begin
  esta:= false;
  pos:=1;
  while ( (pos <= dL) and (esta = false) ) do
     begin
       if (a[pos]= valor) then
              esta:= true ← Si lo encontre, devuelve verdadero
       else
          pos:= pos + 1; ← Paso a la siguiente posición si no lo encontre
     end;
   buscar:= esta;
end.
```

Listas (Crear)

```
program crear;
type
  Lista=^Nodo;
  Nodo=record
    dato:integer; ← Dato siempre es el campo que va a contener la información
    sig:Lista; ← Sig siempre es el campo que va a contener el siguiente nodo
  end;
var
  L:Lista;
begin
  L:=NIL;
end.
```

Listas (Recorrer)

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

Program uno;

```
Type
Lista= ^Nodo;

Nodo= record
  dato:integer;
  sig:Lista;
end;

Var
  L:Lista;
begin
  L:=NIL;
  while (L <> NIL) do begin ← Mientras la lista NO este vacia
    Aca podriamos hacer lo que tengamos que hacer con los elementos de la lista
    L:=L^.sig; ← Pasamos al siguiente nodo
end.
```

¿Y qué pasaría con un corte de control, como me doy cuenta?
“La lista se encuentra ordenada por país”

```
while (L <> NIL) do begin
  PaisAct:=L^.dato.pais;
  while (L <> NIL) AND (L^.dato.pais = PaisAct) do begin
    Aca podriamos hacer lo que tengamos que hacer con los elementos de la lista
    L:=L^.sig;
  end;
end;
```

El primer while me indica que, mientras la lista no esté vacía siga procesando, pero también, me dice que la lista se encuentra **ordenada** por país, entonces utilizó un corte de control que, mientras la lista no esté vacía **y** el país de la lista siga siendo el mismo que tengo en “PaisAct” tengo que procesar los datos de ese nodo y avanzar al siguiente nodo hasta que cambie el dato del corte de control.

Listas (Agregar Adelante)

Implica reservar espacio en memoria para generar un nuevo nodo, asignarle los datos y hacer que este nuevo nodo sea el primero

```
procedure AgregarAdelante (var L:Lista; D:Dato);
var
  NuevoNodo:Lista; ← Siempre del mismo tipo!
begin
  new(NuevoNodo); ← Reservo espacio para el nuevo nodo
  NuevoNodo^.dato:=D; ← D puede ser cualquier cosa, mientras sea del mismo tipo que el campo “dato” del “NuevoNodo”
  NuevoNodo^.sig:=L; ← El nuevo nodo apunta a la Lista, para generar un enlace
  L:=NuevoNodo; ← La lista pasa a ser el “NuevoNodo” y está va a apuntar a NIL (El primer nodo)
end;
```

Listas (Agregar Atrás)

Implica reservar espacio en memoria para generar un nuevo nodo, asignarle los datos y hacer que el último nodo sea el nuevo nodo, manteniendo un orden en el que los datos fueron ingresados.

```
procedure AgregarAtras (var Pri,Ult:Lista; D:Dato);
var
  NuevoNodo:Lista; ← Siempre del mismo tipo!
begin
  new(NuevoNodo); ← Reservo espacio para el nuevo nodo
  NuevoNodo^.dato:=D; ← D puede ser cualquier cosa, mientras sea del mismo tipo que el campo “dato” del “NuevoNodo”
  NuevoNodo^.sig:=NIL; ← El nuevo nodo apunta NIL
  if (Pri <> NIL) then
    Ult^.sig:=NuevoNodo; ← Si la lista tiene nodos, el siguiente a Ult va a ser el NuevoNodo
    Ult:=NuevoNodo; ← El último nodo va a pasar a ser el NuevoNodo
  else begin
      Pri:=NuevoNodo; ← Si la lista está vacía, Pri y Ult pasan a ser el NuevoNodo
      Ult:=NuevoNodo;
  end;
end;
```

Listas (Insertar Ordenado)

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

```
procedure InsertarOrdenado(var L:Lista; valor:integer);
var
  Ant,Act:Nue:Lista;
begin
  New(Nue);
  Nue^.Dato:=valor;
  Nue^.Sig:=NIL;
  Act:=L;
   while (Act <> NIL) AND (Act^.Dato ** Nue^.Dato) do begin ← “**” simboliza una operación que depende del criterio de ordenamiento, si es ascendente, descendente o que sean distintos (como el caso del parcial de 2024 tema 2, en el que tenías una lista ORDENADA por país)
    Ant:=Act; ← Voy posicionando Anterior para que quede atras del nodo Act
    Act:=Act^.Sig; ← Me posiciono en el siguiente nodo
  end;
  if (Act = L) then begin ← Pregunto si Act es igual a L (la lista está vacía)
    Nue^.Sig:=L; ← Agregar Adelante
    L.=Nue;
  end;
  else begin
    Ant^.Sig:=Nue; ← Hago el enlace entre lo que apunta el Anterior con el Nuevo Nodo
    Nue^.Sig:=Act; ← Hago el enlace entre lo que apunte el Nuevo Nodo con Actual
  end;
end;
```

Listas (Búsqueda)

Desordenado

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 (L:Lista; Valor:Dato):boolean;
var
  encontre:boolean;
begin
  encontre:=false;
while (L <> NIL) AND (encontre = false) do begin
  if (L^.dato.elemento = Valor) then
    encontre:=true
  else
    L:=L^.sig;
end;
buscar:=encontre;
end;
```

Ordenado

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 (L:Lista; Valor:Dato):boolean;
var
  encontre:boolean;
begin
  encontre:=false;
while (L <> NIL) AND (L^.dato.elemento ** valor) do begin ← “**” representa una operación
que la determina el enunciado, si es que busca de forma ascendente o descendente
  L:=L^.sig;
end;
if (L <> NIL) AND (L^.dato.elemento = valor) then
  encontre:=true; ← Si encontre el elemento, entonces pongo en true “encontre”
buscar:=encontre;
end;
```

Listas (Eliminar)

Sin repeticiones

```
procedure eliminar (var L:lista; valor:dato);
var
 act,ant:lista;
begin
  act:=L;
  while (actual <> NIL) and (actual^.elem <> valor) do begin ← Si donde estoy parado, no
```

```
encontré el elemento, sigo buscándolo
   ant:=act; ← Anterior toma el lugar de Actual
   actual:= actual^.sig; ← Actual avanza al siguiente nodo
  end;
  if (act <> NIL) then ← Me fijo si estoy parado al comienzo, en el medio o al final
   if (act = L) then
     L:= L^.sig ← Si era el primer nodo, L pasa a ser el siguiente (NIL)
   else
    ant^.sig:= actual^.sig; ← Si era un nodo en alguna posición, hago un enlace entre lo que
apunta Anterior con el siguiente a Actual, así no pierdo el enlace
   dispose(act); ← Eliminar el nodo que tenía que eliminar
end;
```

Con repeticiones

```
procedure eliminar (Var L: lista; valor:integer);
Var
 Act, Ant:lista;
Begin
  Act:=L;
  while (Act <> nil) do begin ← Recorrer los nodos
   if (Act^.elem ** valor) then begin ← Si la lista está ordenada uso la operación
dependiendo de si es ascendente o descendente, si está desordenada, uso el distinto “<>”.
    Ant:=Act; ← El Anterior es Actual
    Act:= Act^.sig; El Actual pasa al siguiente nodo
   end;
   else begin
    if (Act = L) then ← Si Actual es igual a L es porque es NIL
      L:= L^.sig; ← Paso al siguiente nodo
    else
      Ant^.sig:= Act^.sig; ← Hago que lo que apunte Anterior sea el siguiente de Actual para
no perder el enlace de todos los nodos
    dispose (Act); ← Eliminó finalmente el nodo
    Act:=Ant ← Tengo que reubicar el actual con el anterior, para seguir recorriendo la lista;
   end;
End;
```

Corrección de Programas

Cuando se desarrollan los algoritmos hay dos conceptos importantes que deben tenerse en cuenta: Corrección y Eficiencia del programa

Testing

El propósito del Testing es proveer evidencias convincentes de que el programa hace el trabajo esperado.

- Decidir cuáles aspectos del programa deben ser testeados y encontrar datos de prueba para cada uno de esos aspectos

- Determinar el resultado que se espera que el programa produzca para cada caso de prueba
- Poner atención en los casos límite
- Diseñar casos de prueba sobre la base de lo que hace el programa y no de lo que se escribió del programa. Lo mejor es hacerlo antes de escribir el programa.

Una vez que el programa ha sido implementado y se tiene el plan de pruebas:
- Se analiza el programa con los casos de prueba
- Si hay errores se corrigen
  (Estos dos pasos se repiten hasta que no haya errores)

Debugging

Es el proceso de descubrir y reparar la causa del error
Para esto pueden agregarse sentencias adicionales en el programa que permiten monitorear el comportamiento más cercanamente
Los errores encontrados pueden ser de tres tipos
- Sintácticos: Se detectan en la compilación
- Lógicos: Generalmente se detectan en la ejecución
- Sistema: Son muy raros los casos en los que ocurren

Walkthroughs

Es el proceso de recorrer un programa frente a una audiencia
La lectura de un programa a alguna otra persona provee un buen medio para detectar errores
- Esta persona no comparte preconceptos y está predispuesta a descubrir errores u omisiones
- A menudo, cuando no se puede detectar un error, el programador trata de probar que no existe, pero mientras lo hace, puede detectar el error, o bien puede que el otro lo encuentre

Verificación

Es el proceso de controlar que se cumplan las pre y post condiciones del mismo
Para determinar la corrección de un programa puedo utilizar una o varias técnicas de corrección la cantidad de veces necesarias hasta que el programa sea correcto

Eficiencia de programas

Una vez que se obtiene un algoritmo y se verifica que es correcto, es importante determinar la eficiencia del mismo
El análisis de la eficiencia de un algoritmo estudia el tiempo de ejecución de un algoritmo y la memoria que requiere para su ejecución
Los factores que afectan la eficiencia de un programa

- Memoria: Se calcula (Como hemos visto previamente) teniendo en cuenta la cantidad de bytes que ocupa la declaración en el programa de:
    - Constante/s
    - Variable/s global/es
    - Variable/s local al programa/es

- Tiempo de ejecución: Puede calcularse haciendo un análisis empírico o un análisis teórico del programa

El tiempo de un algoritmo puede definirse como una función de entrada:

- Existen algoritmos que el tiempo de ejecución no depende de las características de los datos de entrada sino de la cantidad de datos de entrada o su tamaño
- Existen otros algoritmos que el tiempo de ejecución es una función de la entrada "específica", en estos casos se habla del tiempo de ejecución del "peor" caso. En estos casos, se obtiene una cota superior del tiempo de ejecución para cualquier entrada

Para medir el tiempo de ejecución se puede realizar un análisis empírico o un análisis teórico

Análisis empírico
Requiere la implementación del programa, luego ejecutar el programa en la máquina y medir el tiempo consumido para su ejecución

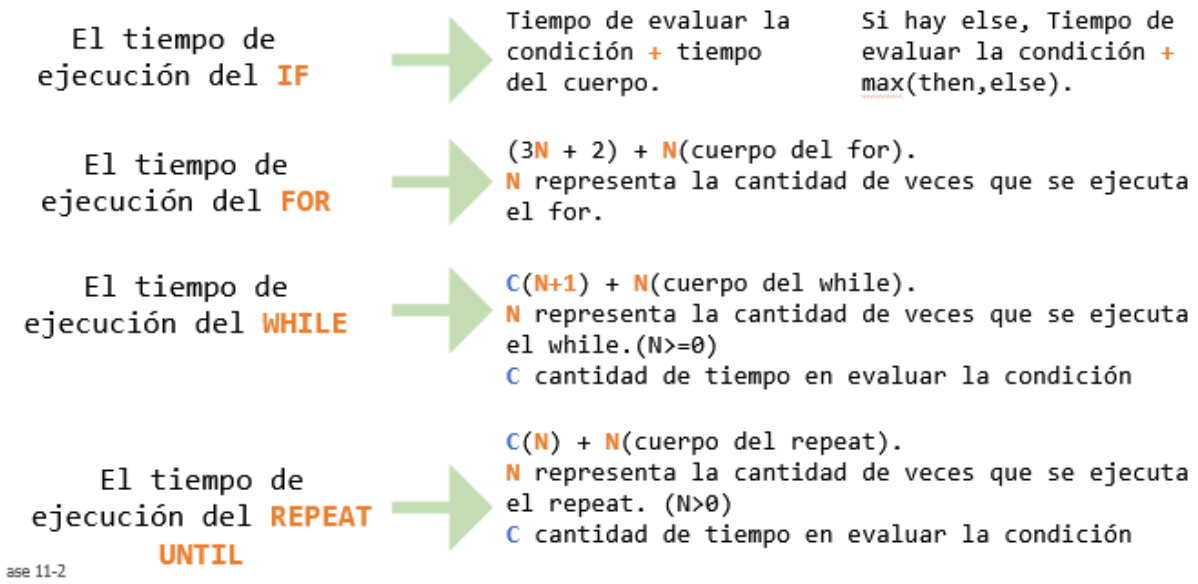
- Es fácil de realizar pero…
- Obtiene valores exactos para una máquina determinada y unos datos determinados
- Completamente dependiente de la máquina donde se ejecuta
- Requiere implementar el algoritmo y ejecutarlo repetidas veces (Para luego calcular un promedio)

Análisis teórico
Implica encontrar cota máxima ("peor caso") para expresar el tiempo de nuestro algoritmo, sin necesidad de ejecutarlo

- A partir de un programa correcto, se obtiene el tiempo teórico del algoritmo y luego el orden de ejecución del mismo. Lo que se compara entre algoritmos es el orden ejecución

Dado un algoritmo que es correcto se calcula el tiempo de ejecución de cada una de sus instrucciones. Para eso se va a considerar:

- Sólo las instrucciones elementales del algoritmo: Asignación, y operaciones aritmético/lógicas
- Una instrucción elemental utiliza un tiempo constante para su ejecución, independientemente del tiempo de dato con el que trabaje 1UT (Unidad de Tiempo).



ase 11-2

Esta tabla podría estar sujeta a cambios, consultar a un docente.