

FINAL TEÓRICO

Informática es la ciencia que estudia el análisis y resolución de problemas utilizando computadoras.

Ciencia — Metodología y razonamiento.

Resolución — con aplicación en varias áreas

Desarrollo

Computadora — Máquina digital (señales eléctricas e información discreta, es decir, 0 y 1) y sincrónica (operaciones coordinadas por un reloj central). Con cierta capacidad de cálculo numérico y lógico controlado por un programa almacenado y con probabilidad de comunicación con el mundo exterior.

Paradigmas de programación

Funcional - imperativo - lógico - orientado a objetos

PARADIGMA:

- Modelo básico de diseño y desarrollo de programas.
- Patrones de pensamiento para resolución de problemas.

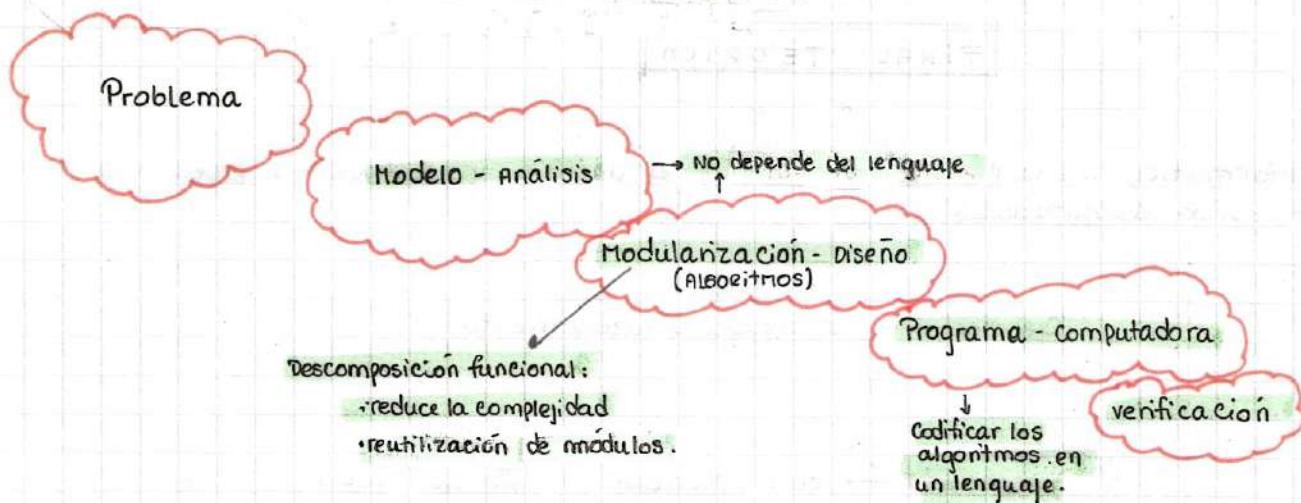
Los lenguajes de programación pueden ser clasificados a partir del modelo que siguen para definir y operar información. Permite jerarquizarlas según el paradigma que siguen.

En este curso: Imperativo - procedural

El código fuente de lenguajes imperativos encadena instrucciones. Los valores utilizados en las variables se modifican durante la ejecución del programa. Para gestionar las instrucciones, se integran estructuras de control como bucles o estructuras anidadas en el código.

Ej: Fortran, java, pascal, C, C++, C#, ensambladores, python.. etc.

ETAPA DE RESOLUCIÓN



- **Primer etapa**: - requerimientos del usuario.
 - modelo preciso del ambiente del problema y del objeto a resolver.
- **Etapa de diseño**: - a partir del modelo se diseña la solución. En el paradigma procedural esto involucra la modularización del problema y los datos necesarios. (ALGORITMOS)
- **implementación de solución**: - Escritura de los programas. (ALGORITMO + DATOS)
- **Verificación**: - una vez escritos y depurados de errores de sintaxis, se debe verificar la ejecución, si hace lo que se le pide utilizando datos representativos.

ALGORITMO

- especificación rigurosa
- tiempo fínito
- lenguaje ejecutable por la máquina.

Es una secuencia de pasos (instrucciones) a realizar sobre un automata para alcanzar un resultado deseado en tiempo fínito. (comienza y termina).

PROGRAMA

ALGORITMO + DATO

Conjunto de instrucciones ejecutables sobre una computadora.

valores de información de los que se necesita disponer y en ocasiones transformar para ejecutar la función de un programa.
Cada lenguaje de programación tiene los propios.

DATO

Constante & variable

Representación de un objeto del mundo real. Con él podemos modelizar aspectos del problema que se quiere resolver con un programa sobre una computadora.

Constante

no cambia durante la ejecución del programa

Variable

si cambia durante la ejecución del programa.

Requerimientos:

Desarrollador

- operatividad
Debe funcionar
- legibilidad
Fácil de entender (comentarios)
- organización
descomposición en módulos
- documentación
Proceso de análisis, diseño
y solución debe estar documentado.

Computadora

- instrucciones válidas
- Deben ser fímitas.
- NO debe utilizar recursos inexistentes.

Lípo de dato

- ES UNA CLASE DE OBJETO DE DATOS LIGADOS A UN CONJUNTO DE OPERACIONES PARA CREAMOS Y MANIPULARES
(rango-tipo)
Tienen un conjunto de operaciones y valores permitidos junto con una representación interna.

SIMPLE

Toma un único valor en un momento determinado.

DEFINIDO POR EL LENGUAJE
(provisto por el)

Tanto la representación como sus operaciones y valores son reservadas al mismo.

- NÚMÉRICO
- CARÁCTER
- BOOLEAN

COMPUESTO

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

DEFINIDO POR EL

PROGRAMADOR
Permite definir nuevos tipos de datos a partir de los tipos simples.

• SUBTIPO

• PUNTERO

- Permite abstracción de datos
- más seguridad en las operaciones sobre el dato.
- Conjunto nuevo de valores.

ESTRUCTURADOS

- REGISTROS
- ARREGLOS
- LISTAS

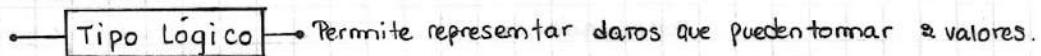
• STRINGS



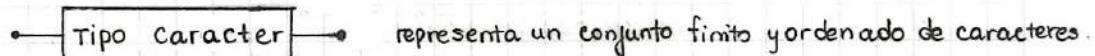
Entero: **simples** y **ordinal**, tienen un máximo y un mínimo.

Es decir, es un conjunto **finito**, que guarda una relación. Escalar, con mayor y menor (**cual procede** y **cual sucede**). -finito y ordenado-

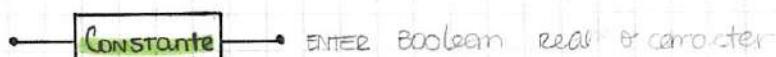
REAL: **simples**, NO ordinal, maximo y minimo.



Booleano: **simples** y **ordinal** (true, false).



Carácter: **simples** y **ordinal** (Tabla ASCII) - tipo de dato carácter contiene solo un carácter.



Zona de la memoria cuya contenido va a ser **alguno** de los tipos anteriores.

La dirección inicial de esta zona se asocia con el nombre de la variable.

Pero a diferencia de la variable, **(NO)** cambia su valor durante la ejecución del programa.

Especificación = declaración.

- Algunos lenguajes exigen especificación de cada variable (el tipo) → se llaman "fuertemente tipados".
- Otros lenguajes que verifican el tipo de las variables según su nombre se llaman autotipados.
- Existe una tercera clase que permite que una variable tome valores de distinto tipo durante la ejecución. se denominan dinámicamente tipados.

Tipo de dato definido por el usuario.

(Aquel que **no existe** en la definición del lenguaje.)



- Mayor abstracción de datos.
- Mayor seguridad sobre c/ operación sobre c/dato.
- Límites pre establecidos sobre valores posibles.

Ventajas:

Flexibilidad: se modifica una declaración en lugar de un conjunto de variables.

Documentación: nombres autoexplicativos

Se reducen errores por el uso de operaciones inadecuadas del dato a manejar. -seguridad

Algunas definiciones

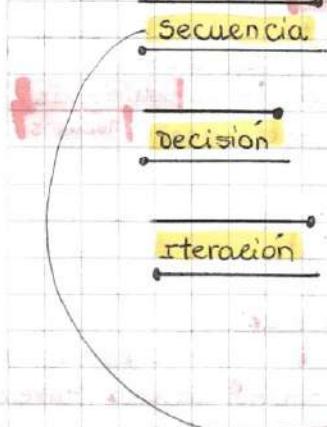
- Pre-condición: información de conocimiento previo a la ejecución.
- mientras sea verdadero -
- Post-condición: información
- mientras sea falso -

Operaciones

Read: Toma datos desde un dispositivo de entrada (teclado por defecto) y asignarlo a una variable asociada. *Leyendo en la memoria*

Write: Para mostrar el contenido de una variable en pantalla.

ESTRUCTURAS DE CONTROL



estructura más simple. sucesión de operaciones, el orden de ejecución coincide con el orden físico (*case*).

Algoritmo representativo de un problema donde se deben tomar decisiones en función de los datos (*if*)

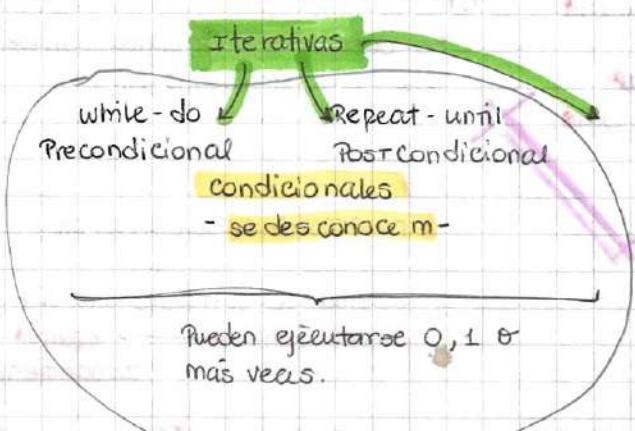
Iteración

Ejecución de bloques repetidamente.

• Iterativa condicional: *Repeat until* y *while*.

• Repetición: *For*.

la repetición es una extensión de la secuencia, repetir un bloque *m* cantidad de veces. (*entero, boolean, char*) la variable índice debe ser de tipo ordinal y no puede reemplazarse durante el lazo (*i*).



MODULARIZACIÓN

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

Ante la:

- complejidad
- extensión
- modificaciones

de problemas del mundo real

se necesita:

- abstracción
- descomposición
- independencia funcional

NO se trata de subdivide código en bloques con tanta cantidad de instrucciones.
Es separar en funciones lógicas con datos propios y datos de comunicación especificados.

- ventajas

- Reusabilidad
- Fácil mantenimiento
- legibilidad
- Productividad

→ Desarrollo, trabajo simultáneo en varios módulos

subproblemas

- Resolución independiente
- Combinación de las soluciones para resolver el original.
- Módulos

módulos

- tarea específica.
- comunicación entre ellos (objetivos en común).
- Encapsulan tareas, acciones, funciones.
- Representan los objetivos relevantes.

IMPORTANTE

Parámetro real:
invocación del módulo

Parámetro formal:
declaración del módulo.

METODOLOGÍA TOP DOWN

se basa en el paradigma "divide y vencerás".

- Dividir en partes más simples.

mejoras de modularización

sistema de software dividido en módulos funcionalmente independientes → equipo → Trabajo simultáneo, menor tiempo de desarrollo.

La modularización permite disminuir los riesgos y costos de incorporar nuevas cosas a un sistema en funcionamiento (crecimiento).

Function - retorna un único valor
TIPO SIMPLE

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

- El retorno es a la misma línea de invocación.



- Última instrucción: la asignación del nombre a la variable que retorna.

Procedure - retorna 1 o más valores

Ej: `PUT TERO`

la diferencia con la función es la cantidad de valores que devuelven, forma de manejar y el parámetro que puede utilizarse (función - por valor).

Globales

Pueden ser usadas en todo el programa (incluyendo módulos).

Locales del Proceso

Pueden ser usadas sólo en el proceso que están declaradas.

Locales del Programa

Pueden ser usadas sólo en el cuerpo del programa.

Comunicación entre módulos y módulos y el programa principal.

- Parámetros
- Variables globales

IMPORTANTE

Si una variable utilizada en un proceso.

- se busca si es variable local
- si es un parámetro
- si es variable global al programa

Si es una variable usada en un programa.

- se busca si es variable local al programa
- si es variable global al programa

• El problema de utilizar variables globales es la posibilidad de perder integridad de los datos, al modificar en un módulo datos de alguna variable que luego se utiliza. (Es decir que los programadores pueden modificar la misma variable global).

También problemas con los nombres identificadores que utiliza cada programador.

Están vigentes en la memoria durante todo el programa más allá de ser utilizadas una vez, ocupan memoria.

Como la utilizan varios módulos, no se sabe cuál la modifica.

Solución a problemas asociados por el uso de variables globales

Combinación

DATA HIDING

PARÁMETROS

Data hiding: datos exclusivos de un módulo, no deben ser visibles o utilizables por los demás.

Parámetros: datos compartidos, especificados como parámetros que se transmiten entre módulos.

Parámetro por entrada: se refiere a un dato de entrada por valor que significa que un módulo recibe (sobre una variable local) un valor proveniente de otro módulo (o del programa principal). - Copia del valor - no producen cambio fuera del módulo.

La comunicación por referencia significa que el módulo referencia a una dirección conocida en otros módulos del sistema.

Puede operar con su valor original dentro del módulo y las modificaciones que se produzcan se reflejan en los demás módulos que conocen la variable.

Al invocar el procedimiento el parámetro se relaciona por posición.

Parámetro: comunicación externa con el resto del sistema.

CARACTERÍSTICAS

El nº y tipo de los argumentos = el número y tipo de parámetros del encabezamiento del módulo.

Parámetro por valor → variable de la cual el procedimiento hace una copia y la utiliza localmente. Modificación realizada que en el módulo.

Parámetro por referencia → operan sobre la dirección de memoria donde se encuentra la variable original. No requiere memoria local → IMPORANTE

Análisis de Gonzalo Villa real:

Variable en el programa principal (memoria estática) → invoca a un módulo

variable por valor,
copia en memoria
dinámica

Ej: Variable interna a un módulo → la memoria se reserva (se aloca) al momento de invocar al módulo, y esa variable debe estar en memoria dinámica.

Repasar

• **String:** sucesión de caracteres de determinada longitud (max 255 caracteres) y cada carácter ocupa 1 byte.

Cada string ocupa 265 bytes, string[m] donde la anchura es m. Estos caracteres ocuparán $m+1$ bytes en memoria.

• **Subtringo:** simple y ordinal → importante

Clasificación de estructuras de datos

ELEMENTOS

- qué tipo son

ACESO

TAMAÑO

-memoria.

LINEALIDAD

-almacenamiento

• Homogéneo

• secuencial

• dinámica

• No lineal

• heterogéneo

• directo

• estática

• lineal.

Homogénea: los elementos que la componen son del mismo tipo.

Heterogénea: Pueden ser de distinto tipo.

En cuanto a la memoria, si la estructura puede variar su tamaño durante la ejecución del programa.

Estática: el tamaño no varía durante la ejecución del programa. (tamaño fijo).

- Pero puede cambiar el valor

Dinámica: Puede cambiar la cantidad de elementos que almacena durante la ejecución.

Otra clasificación es en cuanto a cómo se accede a los elementos que la componen.

secuencial: Para acceder, se debe recorrer un orden predefinido.

Directo: se accede directamente sin necesidad de pasar por el anterior/antiguos. (referenciando una posición por ejemplo). **Importante**

y al hablar de la linealidad, es cómo se almacenan o se encuentran almacenados los elementos que la componen.

Lineal: formada por mínimo, uno o varios elementos que guardan una relación de adyacencia, ordenada. A cada elemento le sigue y precede uno solo.

No lineal: Para cada elemento hay 0, 1 o más elementos "antes" y "después".

Registros

ESTRUCTURA HETEROGENEA Y ESTÁTICA

Compuesta por campos - cada tipo del campo debe ser estático

Acceso: Variable Registro. NOMBRE CAMPO. NOMBRE SUB CAMPO.

IMPORTANTE

Patrón de corte de control

while (condición)

un circuito variable de condición

while (condición 2) AND (condición 1)

3

Nueva lectura e siguiente

emdi;

end.

America

ESTRUCTURA HOMOGENEA, ESTÁTICA E INDEXADA.

INDEXADA: PARA ACCEDER A CADA ELEMENTO, SE DEBE UTILIZAR UNA VARIABLE INDICE QUE ES DE TIPO ORDINAL → IMPORTANTE

El rango debe ser de tipo ordinal

- Subramgo
 - Boolean
 - Integer
 - CHAR

los elementos del arreglo son de tipo ESTÁTICO.

IMPORTANTE

- PUNTERO
 - INTEGER
 - REAL
 - CHAR
 - BOOLEAN
 - STRING
 - register
 - Areglo

Impresión de um vector

```
Procedure Mostrar (a:mumeros); | Procedure mmostrar (a:mumeros);  
var i:integer; valor:integer; | Var i:integer;  
begin | begin  
for i:=1 to constante do |  
begin | write (a[i]);  
valor := a[i]; | end;  
write (valor); |  
end; |  
end;
```

¿Cuántas veces aparece um maximo?

Buscar maximo .(a: mumeros):integer;

fumetion maximo (a: mumeros): integer;

var i:integer; max:integer;

begin

```
max:=-1;  
for i:=1 to constante do.  
if (a[i]>=max) then  
max:=a[i];
```

maximo:=max;

end;

fumetion cantidad (a: número, num: integer): integer;

Var i:integer; cant:integer;

begin

```
cant:=0;  
for i:=1 to constante do.  
if (a[i] = num) then  
cant:= cant+1;
```

Cant:= cant;

end;

Var
a: mumeros
max: integer

begin

```
cargar (a);  
max:=máximo (a);  
write(' la cantidad de  
veces que apareció  
el nro max em el  
vector es: ',  
cantidad(a, max));  
end;
```

Recomendado de un vector IMPORTANTE.

Dimensión física y lógica

Dimensión física

Se determina en el momento de la declaración, es la ocupación máxima.

La cantidad de memoria no varía durante la ejecución del programa.

Dimensión lógica

Se determina al cargar datos en los elementos del arreglo.

Indica la cantidad de memoria ocupadas como contenido real.

No supera a la dimensión física.

Ej: Programa que carga un arreglo con números enteros hasta leer el número 50, a lo sumo se cargan 300 números.

Luego de terminar la carga, informe cuál es el número más grande de los leídos.

Programa uno;

```
const  física = 300;  
type  numeros = array [1..física] of integer;  
var   v: numeros; dimL: integer;  
begin  
    llenarNumeros (v, dimL);  
    writeln ('El numero mayor es:', mayor (v, dimL));  
end,
```

Procedure llenarNumeros (var v: numeros; var comt: integer);

```
var  
    num: integer;
```

```
begin
```

comt := 0;

read (num);

while ((comt < física) and (num <> 50)) do begin

comt := comt + 1
 v [comt] := num;

read (num);

end;

end;

¿Comt es dimL?

→ No supera la dimfísica

↳ y numero no es 50.

```

fumetion máximo (v: numeros; dimL:integer): Integer;
var i, max: integer;
begin
  max := -999;

  for i:= 1 TO dimL do
    begin
      if (a[i] >= max) then
        max := a[i];
    end;
  maxim0 := max;
end;

```

"Agrega" datos en los elementos del arreglo.

"Dado un arreglo":
ESTA CARGADO Y SE CONOCE LA DIMENSIÓN LÓGICA.

- ¿Hay espacio? (1)
- INCREMENTAR LA DIMENSIÓN LÓGICA (2)
- Agregar Al final el elemento (3)

El vector y su dimensión se modifican.

```

Procedure agregar ( var v: numeros; var dimL:integer; var pude:Boolean;
                    valor:integer);
begin
  pude := false.

  if ((dimL + 1) <= fisica) then begin
    pude := true;
    dimL := dimL + 1; {aumento dimL}
    v[dimL] := valor; {agrego en el último lugar, dimL muda}
  end;
end;

```

Pregunto si dimL+1 es posible ($\leq \text{dimF}$)
 luego dimL es dimL+1
 Agrego dato

• Insertar elementos en un arreglo.

- Verificar si hay espacio (1)
- Verificar que la posición sea válida (2)
- Hacer lugar para poder insertar el elemento (3)
- Incrementar la cantidad de elementos (4)

EL vector, la dimensión lógica y la variable puede se modifican.

Procedure **Insertar** (Var v: números ; Var dimL:integer ; Valor: integer ; pos: integer ; Var puede: Boolean);

Var

i: integer;

{Pos y valor se leen fuera}

Begin

puede := false

 → Si hay espacio.

 → Si la posición está dentro de la dimL

if (((dimL+1) <= física) and (pos <= dimL) and (pos >= 1)) then

 → Mayor a Cero.

begin

 → Desde la en adelante corro los elementos

 for i:= dimL down to pos do. (dejo lugar).

 v[i+1] := v[i]; {lo que estaba en i pasa a estar en i+1}

 puede := true; desde el último(dimL) hasta pos.}

 v[pos] := valor; → {luego coloco el elemento}

 dimL := dimL + 1; → {aumenta la dimensión lógica}

end;

end;

En el programa principal

• Leo el vector

• Leo el valor y la posición

• Inserto el elemento

Para generar un arreglo solo con los números pares; (a partir de otro arreglo).

Procedure **procesar** (v1: números ; dimL1: integer ; Var v2: números ; Var dimL2: integer);

Var i: integer;

Begin

dimL2 := 0; dimL2 := 1

for i := 1 to dimL1 do. begin {Para cada elemento del vector ya armado} {Si el elemento es par}

 if (esPar(v1[i])) then begin

 dimL2 := dimL2 + 1; {El nuevo vector aumenta su dimL}

 v2[dimL2] := v1[i]; {Copia el elemento par}

 dimL2 := dimL2 + 1; {Voy a la siguiente posición}

 end;

end;

end;

ELIMINAR ELEMENTOS EN UN ARREGLO

- VERIFICAR QUE LA POSICIÓN SEA VÁLIDA (1)
- HACER EL CORRIMIENTO (2)
- DECREMENTAR LA CANTIDAD DE ELEMENTOS (3)

Procedure Bonar (var v:mumeros ; var dimL:integer ; pos:integer; var pude: Boolean);
{ Pos lo leo }

```
Var -  
Begin  
    Pude := false  
    { si la posición es válida (mín cero mán mayor a dimL) }  
    if ( (pos >= 1) and (pos <= dimL) ) then begin  
        for i:= pos to (dimL-1) do { desde la posición hasta el anteúltimo }  
            v[i] := v[i+1]; { El elemento en i+1 pasa a estar en i  
                               (el anterior) }  
        Pude := true  
        dimL := dimL - 1; { La dimensión lógica disminuye }  
    end;  
end;
```

BUSCAR ELEMENTO EN UN ARREGLO

DESORDENADO - se recorre todo el vector. hasta que se encuentre o se termine el vector.

ORDENADO - Busqueda mejorada o dicotómica.

DESORDENADO:

function buscar (a:mumeros; dimL:integer; numm:integer): Boolean;
var pos:integer; esta:Boolean;

Begin

```
    esta := false;  
    pos := 1;  
    { MIENTRAS NO ME PASE }  
    while ((pos <= dimL) and (not esta)) do begin
```

```
        If (a[pos] = numm) then esta := true  
        Else pos := pos + 1;
```

end;

Buscar := esta;

end;

ORDENADO

```
function Buscar (a:numero;  
dimL:integer; numM:integer): Boolean;  
  
var pos:integer;  
  
begin  
  
    pos := 1; {NO ME PASO}  
    {y no llegué}  
    while ((pos <= dimL) and (a[pos] < numM)) do  
        pos := pos + 1; {sigo}  
        {NO ME PASO}  
        {llegué}  
    if ((pos <= dimL) and (a[pos] = numM)) then  
        buscar := true  
    else  
        buscar := false;  
  
end;
```

Busqueda dicotómica

Calculo el elemento que está en la posición del medio, y sigo (si no encontré) eligiendo la mitad del arreglo que convenga.

Procedure Dicotómica (var v: numeros;
dimL:integer ; bus :integer ; var ok: Boolean);

```
var pri , ult , medio :integer;  
begin
```

```
ok := false;  
pri := 1; ult := dimL; medio := (pri+ult) div 2;
```

```
while (pri <= ult) and (bus >> v[medio]) do  
begin
```

```
if (bus < v[medio]) then
```

```
    ult := medio - 1;  
else pri := medio + 1;
```

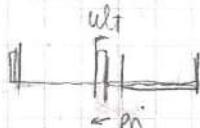
```
medio := (pri+ult) div 2;
```

```
end;
```

```
If (pri = ult) and (bus = v[medio]) then
```

```
ok := true;
```

```
end;
```



CARGAR UN ARREGLO.

HASTA SU DINF.

Procedure carga (var a: numeros);

Var i: integer

begin

for i:= 1 TO CONSTANTE DO.

read (v[i])

end

& read (x);
a[i]:= x;

ORDENAR ELEMENTOS DE UN ARREGLO.

PROCESO ALGORITMO DE ORDENACIÓN

PROCEDURE ORDENAR (var v: numeros; dml: integer).

Var i,j: integer

Begin

for i:= 2 TO dml DO begin.

actual := v[i];

j := i-1;

while (j > 0) and (v[j] > actual) do begin

v[j+1] := v[j];

j := j-1;

end;

v[j+1] := actual;

end;

end;

los Algoritmos de:

Selección

Intercambio

Inserción

difieren en

dificultad

memoria

tiempo

PUNTEROS

Variable estática que almacena la dirección en memoria de otra variable (dinámica).

Valores posibles : Nil & una dirección en memoria. IMPORTANTE

EN MEMORIA (STACK)

Puntero: 4 bytes

Integer: 4 bytes

Char: 1 byte

Real: 8 bytes

Boolean: 1 byte

Registro: La suma de los campos

Arreglo: dimf * tipo del elemento

String: Comt de elementos.

IMPORTANTE

UNA FUNCIÓN

PUEDE DEVOLVER UN PUNTERO.

POr ser de tipo simple

MEMORIA ESTÁTICA (capacidad limitada)

Variables estáticas: no varian en dimensión (durante la ejecución). Estas variables reservan memoria cuando se declaran (y lo tipos).

MEMORIA DINÁMICA (Heap)

Permiten modificar su tamaño durante la ejecución del programa.

MEMORIA

Dir	-
variable	5623
Dir	-
Dir	-
Dir	5623
Dir	-

Puntero

con el se accede (una variable de memoria) a un valor almacenado.

MEMORIA
ESTÁTICA



MEMORIA
DINÁMICA

ocupa una cantidad de memoria fija (4 bytes), independientemente del tipo de dato al que apunta.

solo apunta a direc. memoria almacenadas en la memoria heap.

Apunta a un único tipo de dato.

Reserva y libera memoria durante la ejecución del programa.

IMPORTANTE

Un dato apuntado no tiene inicialmente espacio reservado en memoria.

Operaciones

creación new (variable puntero)

Reserva espacio según el tipo de dato ('pide' al sistema operativo). Aquí es donde el lenguaje le asigna al puntero una dirección libre en memoria heap.

Eliminación dispose (variable puntero)

libera la 'comunicación'. El puntero no tiene dirección definida y la memoria es liberada.

```
procedure f(p:puntero)
begin
    new(p);
end;
El dato queda en memoria pero es inaccesible
```

dispose vs nil

Ambos liberan la "conexión" entre la variable puntero y la posición de memoria.

dispose libera la posición de memoria, con nil la memoria sigue ocupada.

Con dispose la memoria que se liberó, puede utilizarse en otro momento del programa, pero con nil, la memoria no se puede referenciar "se pierde". Al finalizar el programa, la memoria se limpia automáticamente.

Asignación: un puntero a otro.

El parámetro por referencia se relaciona con los punteros

porque el parámetro comparte una dirección con el parámetro utilizado en la declaración y la variable en la invocación.

P: dirección de memoria
P^: contenido (dato).

¡IMPORTANTÍSIMO!

Si aún no reserva memoria (solo declaré variables), P sigue ocupando 4 bytes. Luego ocupa 4 bytes + el contenido que tiene la dirección a la que apunta.

4 bytes en memoria estática mientras que el contenido es dinámica.

* modos *

Cont que ocupa en memoria el dato+puntero

NO se puede hacer

write(p);
read(p);

IMPORTANTE

p:= una dirección

o

P.</> q (otro puntero).

Si un proceso trabaja sobre el dato apuntado, aun cuando el puntero NO es pasado por referencia, lo modifica.

LA MEMORIA DINÁMICA ES CONOCIDA POR TODO EL PROGRAMA

Listas

Tipo de dato **compuesto** - Estructura **recursiva**

La lista es una colección de elementos homogéneos, relación lineal entre ellos. Cada elemento tiene un único predecesor (excepto el primero) y un único sucesor (excepto el último).

Los elementos pueden aparecer dispersos en la memoria, pero mantienen un orden lógico.

• Organizadas
• Óptimas

NODOS = (dato + puntero al elemento siguiente).

- conectados por los punteros
- A medida que se requiere espacio, se alocan y agregan a la estructura (new).
- Si el nodo ya no se necesita, dispose.

IMPORTANTE

CREAR UNA LISTA VACIA

Iniciarla en mil siempre. NADA MAS.

{Programa principal}

Begin

P := mil;
READ (dato);
CARGA LISTA (L);

end.

IMPORANTE

: Agregar adelante. NO ordenadamente

Procedure Agregar_Adelante (Var p: lista; valor:integer);

Var aux : lista;

Begin

New (aux); { Alocacióñ (reservo espacio en memoria)}

aux^.dato := valor;

aux^.sig := mil;

If (p = mil) then p:=aux; { Si es el primer elemento de la lista a agregar, entonces le asigno el puntero inicial }

Else begin

- ① aux^.sig := P; { El modo creado guarda la dirección del anterior y el puntero inicial (p) se actualiza, guardando la dirección del ultimo modo creado}
- ② p := aux;

end;

Si la lista no esta vacia, antes de avanzar al siguiente modo con el puntero principal, SIEMPRE tengo que guardar la dirección del anterior en el nuevo modo.

Otra forma:

Procedure AgregarAdelante (Var p:lista; valor:tipo);

Var aux:lista;

Begin

mew (aux); aux^.dato := valor; aux^.sig := p; p := aux;

end;

IMPRIMIR LA LISTA

```
Procedure IMPRIMIRLISTA (p : Lista);  
Var aux : Lista;  
Begin  
    aux := p; { Inicializo el recorrido }  
    While (aux^.sig <> mil) do begin { mientras no llegue al final }  
        write (aux^.dato);  
        aux := aux^.sig; { avanza a la dirección siguiente }  
    end;  
End;
```

o sin variable auxiliar

```
Procedure IMPRIMIR (p: lista);  
Begin  
    While (p <> mil) do  
        begin  
            write (p^.dato);  
            p := p^.sig;  
        end;  
    end;
```

Agregar al final

```

Procedure AgregarAlFinal (var p:lista; valor: tipo);
var aux, act:lista;
Begin
  New (aux);
  aux^.dato := valor;
  aux^.sig := nil;
  Si es el primero
  If (p=nil) then p:=aux;
  Else begin
    Inicializo en el primer modo
    act:=p;
    LECTURA Hasta
    emcontrar
    el ultimo
    while (act^.sig<>nil) do
      act:=act^.sig;
    act^.sig := aux;
    El ultimo sigue al nuevo modo.
  end;
End;
  
```

{ ult siempre "sigue" al último modo }

```

Procedure AgregarAlFinal (var p, ult:lista; valor: tipo);
var aux;
Begin
  New (aux);
  aux^.dato := valor;
  aux^.sig := nil;
  If (p=nil) then p:=aux;
  Else ult^.sig := aux;
  ult := aux;
End;
  
```

INSERTAR UN ELEMENTO

Casos a tener en cuenta:

- lista vacía
- El elemento va al comienzo
- El elemento a insertarse va al medio
- va al final

Procedure Insertar (var p:lista ; valor : tipo);

Var aux, act, ant: lista;

Begin

 New (aux);

 aux^.dato := valor;

 aux^.sig := mil;

 If (p = mil) then

 p := aux;

 Else begin

 act := p; ant := p;

 While (act <> mil) and (act^.dato < aux^.dato) do

 begin

 ant := act;

 act := act^.sig;

 end;

 If (act = p) then begin

 aux^.sig := p;

 p := aux;

 End;

 Else if (act <> mil) then begin

 ant^.sig := aux;

 aux^.sig := act;

 End;

 Else begin

 ant^.sig := aux;

 aux^.sig := mil;

 End;

 End;

 { Esta vacío?

 } Busco la posición

 { Al principio

 } al medio

 } al final

Otra opción hecho en clase (Prof. Laura)

procedure insertar (var p:lista; valor; tipo);

Var aux, act, ant: lista;

Begin

new (aux);

aux^.dato := valor; | Preparo el modo

aux^.sig := mil;

If (p = mil) then p := aux; } si cargo la lista previamente no lo necesito

Else begin

act := p

| ② act y ant tiemben la dirección del puntero inicial

amt := p;

while (act <> mil) AND (act^.dato < aux^.dato) do begin

amt := act;

| hasta encontrar donde insertar y que no
act := act^.sig; | llegue al final (cuando act^.sig es mil)

end;

if act = mil

If (act = p) then } En caso de que salió del while porque el modo a insertar
va en el primer lugar (la condición podría ser act = amt
begin | por ②).

aux^.sig := p; } guardo mil (podía decir aux^.sig := act)

p := aux; } El puntero inicial apunta a aux

end;

Else begin

ant^.sig := aux;

aux^.sig := act;

| ant guarda la dirección de aux y aux la de act

end;

end;

+ he hecho algunas modificaciones →

Procedure Inseetare (var p: lista, valor: tipo); (insetar ordenado).

var amt, aux, act : lista;

begin

new (aux);

aux^.dato := valor;

act := p;

SIEMPRE PRESUNTO PREVIO SI NO ES NIL
SERIA UN ERROR PREGUNTAR SI NIL ES MAYOR, IGUAL, ETC A DATO.

amt := p;

while (act <> nil) AND (act^.dato < aux^.dato) do begin

amt := act;

act := act^.sig;

end;

If (amt = ^{≡ p}act) then p := aux;

else amt^.sig := aux;

aux^.sig := act; si amt = act = p \rightarrow aux^.sig := act = nil
si no aux^.sig guarda el siguiente modo

End;

Buscar elemento

Si esta desordenada:

Function Buscar (p:lista; valor:integer): Boolean

Var act: lista; encontrado: Boolean;

Begin

act:= p; {Para recorrer la lista inicializada en p}

encontrado:= false;

while (act<>nil) AND (encontrado = false) do

begin

If (act^.dato = valor) then

encontrado := true;

else

{Recorro hasta encontrar}

act:= act^.sig;

end;

Buscar := encontrado;

end;

Si esta ordenada:

Function Buscar (p:lista; valor:integer): Boolean

Var act: lista; encontrado: Boolean;

Begin

act:= p;

encontrado:= false;

while (act<>nil) AND (act^.dato < valor) do begin

act:= act^.sig;

~~If (act<>nil) AND (act^.dato = valor) then encontrado := true;~~

Buscar := encontrado Buscar := (l<>nil) and (l^.dato = valor).

end;

ELIMINAR ELEMENTO

Procedure **eliminmar** (var p;lista; valor:integer);

Var **aet, amt**:lista;

begin

act:=p; amt:=p;

While (aet <> nil) AND (aet^.sig <> valor) do begin

{Si la lista esta ordenada la condic. es <, no >}

amt:=aet;

aet:=aet^.sig;

end;

If (aet <> nil) then begin

if (aet = p) then p:= p^.sig; {Si es el primero, lo eliminamos guardando la dirección del siguiente el elemento}

else amt^.sig:= aet^.sig;

dispose (aet);

{Sino salteo aet, guardando la dirección del siguiente a aet}

end;

end;

Comparacion Vector vs Listas

IMPORANTE

Arreglo

listas

ESTÁTICA, HOMOGENEA, INDEXADA

DINÁMICA, HOMOGENEA, LINEAL

SE ENCUENTRA ALIJADO EN POSICIONES DE MEMORIA CONSECUTIVAS.

CADA VEZ QUE SE NECESITA AGREGAR UN ELEMENTO SE SOLICITA MEMORIA. NO EXISTE RELACIÓN ENTRE LAS POSICIONES DE MEMORIA.

AGREGAR ADELANTE IMPÍCA CORRER LOS ELEMENTOS Y AUMENTAR LA DIRECCIÓN LÓGICA. NO SIGNIFICA QUE SIEMPRE SE PUEDA.

AGREGAR SIGNIFICA SOLICITAR UN ESPACIO Y REACOMODAR PUNTEROS (SIEMPRE SE PUEDE).

AGREGAR AL FINAL, EN DIML+1, INCREMENTANDO LA DIML Y PUEDE NO HACERSE.

AGREGAR AL FINAL REQUIERE MEVAR UN PUNTERO QUE CONTENGA LA DIRECCIÓN DEL ÚLTIMO ELEMENTO DE LA LISTA, SOLICITAR ESPACIO Y REACOMODAR LOS PUNTEROS.

AL INSERTAR, DEBE BUSCARSE LA POSICIÓN, MEGO CORRER Y CARGAR EL ELEMENTO, SE INCREMENTA DIML. PUEDE NO HACERSE.

AL INSERTAR SE SOLICITA ESPACIO, SE BUSCA LA POSICIÓN Y SE REORGANIZAN LOS PUNTEROS. SIEMPRE PUEDE HACERSE LA OPERACIÓN.

NO SIEMPRE PUEDE HACERSE

PARA ELIMINAR DEBE BUSCARSE EL ELEMENTO Y LUEGO REALIZAR UN CORRIENTO DESDE LA POSICIÓN DONDE ESTÁ EL ELEMENTO HASTA EL FINAL (Y DECREMENTAR DIML).

SE BUSCA EL ELEMENTO Y SE REORGANIZAN LOS PUNTOS.

EN CUANTO AL ACCESO A LA ESTRUCTURA, AL SER UNA ESTRUCTURA INDEXADA, ES DE ACCESO DIRECCIONAL SEGÚN LA POSICIÓN.

AL SER UNA ESTRUCTURA LINEAL, SE DEBE PASAR POR TODOS LOS ELEMENTOS ANTERIORES.

HABLANDO DE MEMORIA OCUPADA, EL ARREGLO ES UNA ESTR. ESTÁTICA POR LO QUE OCUPA LA MISMA CANTIDAD DE MEMORIA ESTÁTICA.

ESTRUCTURA DINÁMICA (SU TAMAÑO CAMBIA AL AGREGAR O ELIMINAR ELEMENTOS).

CUANDO UN ARREGLO ES PASADO POR VALOR, SE COPIA TODO EL ARREGLO EN EL PARÁMETRO CORRESPONDIENTE. CUANDO ES POR REFERENCIA SE PASA SOLO LA DIRECCIÓN.

CUANDO UNA LISTA ES PASADA POR VALOR, SE COPIA LA DIRECCIÓN INICIAL DE LA LISTA. CUANDO ES POR REFERENCIA SE PASA LA DIRECCIÓN INICIAL DE LA LISTA.

IMPORTANTE

```
Cargarlista ( var p: lista );
var R: REGISTRO;
begin
  Leer REGISTRO ( var r );
  while ( Condición ) do begin
    Agregar Nodo ( ... );
    Leer REGISTRO ( R );
  end;
end;
```

Si tengo que ordenar una lista bajo un tipo de condición usar INSERTARE-ORDENADO (var L: lista, R: REGISTRO);
var act, aux

Luego al recomer...

```
while ( p < mil ) do begin
```

```
  while ( Condición bajo la que hay que analizar la
        analizar la lista ordenada ) AND ( p < mil ) do
        begin
```

```
    p := p^.sig;
  end;
  ...
end;
```

Un programa debe ser
fiable, corregible
flexible
eficiente
portable
integridad:
fácil de usar y mantener
fácil de probar
portable
legible (documentación)

calidad del programa.

Corección:

El grado en que una aplicación satisface sus especificaciones y consigue los objetivos recomendados por el cliente.

fiabilidad

El grado que se puede esperar que una aplicación lleve a cabo las operaciones especificadas y con la precisión requerida.

EFICIENCIA

CANT. recursos hardware y software que necesita una aplicación para realizar las operaciones con los tiempos de respuesta adecuados.

Integridad

El grado con que puede controlarse el acceso software o los datos a personal no autorizado.

Facilidad de uso:

El esfuerzo requerido para aprender el manejo de una aplicación, trabajar con ella, introducir datos y conseguir resultados.

Facilidad de mantenimiento.

Esfuerzo requerido para localizar y reparar errores → se vinculará con la modularización y con cuestiones de legibilidad y documentación.

Flexibilidad

Esfuerzo requerido para modificar una aplicación en funcionamiento.

• Facilidad de prueba

Esfuerzo requerido para aprobar una aplicación de forma que cumpla con lo especificado en los requisitos.

• Portabilidad

Esfuerzo para transferir la aplicación a otro hardware o sistema operativo.

• Reusabilidad

Grado en el que las partes de una aplicación pueden utilizarse en otras aplicaciones.

Interoperabilidad

El esfuerzo necesario para comunicar la aplicación con otras aplicaciones o sistemas informáticos.

Lectibilidad

El código fuente debe ser fácil de leer y entender.

Acompañar instrucciones con comentarios adecuados.

El proceso de análisis y diseño del problema y su solución debe estar documentado mediante texto y/o gráficos para favorecer la comprensión, modificación y adaptación a nuevas funciones.

• Se aconseja la inserción de comentarios en el programa.

• Identificadores autoexplicativos.

• Comentario general del objetivo del programa.

• Mantenimiento: actualización de código y comentarios.

Corrección del programa

¿Cumple con el objetivo propuesto?

Un programa es correcto si cumple con las especificaciones del problema a resolver. Por esa razón, la especificación debe ser completa, precisa.

Técnicas de corrección

- Testing
- Walkthrough
- Debugging

proveen evidencias para corrección

Testing provee evidencias mediante:

- Diseñar un plan de pruebas
- Cuales aspectos del programa deben ser testeados y encontrar datos de prueba para el uno.
(que el programa produzca)
- qué resultado se espera para cada paso de prueba
- poner atención en los casos límite
- Casos prueba sobre la base de lo que hace el programa y no de lo que se escribió del programa.
- Diseñar casos de prueba antes de que comience la escritura del programa.
(así las pruebas no están pensadas en base a lo escrito).

Luego del plan de pruebas y el programa, el plan se aplica sistemáticamente.

- Precondiciones → antes de la ejecución (ej: entradas de datos disponibles)
- Las pre y post condiciones permiten describir la función que realiza un programa, si especificar el algoritmo.
- Postcondiciones → aspectos a cumplirse cuando el programa terminó.

Debugging proceso mediante el cual se pueden identificar y corregir errores.

Se puede involucrar el diseño y aplicación de pruebas adicionales para ubicar y conocer la naturaleza del error.

Sentencias adicionales para monitorear su comportamiento más cercanamente.

Caminos de los errores:

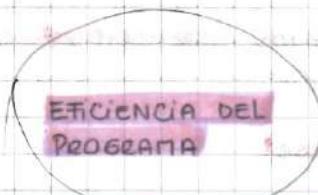
- Diseño defectuoso (del programa)
- Diseño de algoritmo defectuoso.

Se pueden agregar sentencias de salida adicionales que sirven como punto de control o señalar cambios de variables.

walkthroughs recorrer el programa ante una audiencia

que otra persona lo lea provee un medio para detectar errores.
El/ella no comparten preconceptos y tiene mayor predisposición para encontrar errores.

Cuando no se encuentra el error, el programador trata de probar que no existe.



Una vez encontrada la solución correcta, es necesario medir cuántos recursos se utilizan.

Se analiza tiempo de ejecución y memoria utilizada.

EFICIENCIA: métrica de calidad de algoritmos, óptima utilización de los recursos del sistema donde se ejecutará, principalmente memoria utilizada y tiempo de ejecución.

MEDICIÓN DE LA MEMORIA UTILIZADA.

- Se puede calcular ÚNICAMENTE LA MEMORIA ESTÁTICA que utiliza el programa.
- Variables declaradas y tipo correspondiente.

TYPE

Cadena10 = string [10];

ptrString = ^cadena10;

Datos = record

Nombre: Cadena10;

Apellido: Cadena10;

Edad: integer;

Altura: real

end;

personas = Array [1..100] of Datos;

ptrDatos = ^Datos;

Var

frase: ptrString; 4 bytes Como no reservó memoria, solo se declararon

s: cadena10; 11 bytes las variables, los punteros ocupan 4 bytes.

puntero: Pm DATOS; 4 bytes

p: personas; 100 * (11 + 11 + 4 + 8)

∴ ocupa: 3419 bytes

Medición del tiempo de ejecución de un programa

Depende de distintos factores:

- Los datos de entrada del programa
- Tamaño
- Contenido.
- La calidad del código generado por el compilador utilizado.
- Naturaleza y rapidez de las instrucciones de máquina empleadas en la ejecución del programa.
- El tiempo algoritmo base.

Para algunos problemas, el tiempo de ejecución se refiere al tiempo de ejecución del "Peor caso"

Ej: problema de búsqueda secuencial y listas.

Calculo del tiempo de ejecución:

- Análisis Empírico. EJECUTAR EL PROGRAMA Y MEDIR EL TIEMPO EMPLEADO DE EJECUCIÓN

• Teórico

Inconveniente:

- Obtiene Valores experimentados.
- Se implementa sin necesidad de implementar el algoritmo.
- Independiente de la máquina donde se ejecuta.
- Obtiene los valores exactos para UNA máquina.
- Dependiente de la máquina donde se ejecuta.
- Requiere implementar y ejecutar VARIAS veces.

Considera:

- mº operaciones elementales que emplea el algoritmo
- una oper. elemental utiliza tiempo constante - independiente del dato.
- supone que una op. elemental es una asignación, comparación o una operación aritmética simple.

Reglas generales para el cálculo del tiempo de ejecución.

1°

Sentencias consecutivas

For Anidados

While / Repeat until

Última regla

Importante: los comentarios, declaraciones y operaciones

de E/S (read, write) no se consideran al realizar el cálculo.

Program temperaturas;

Var valor, total: real;

begin

total := 0.

Dos Asignaciones ($i := 1, i <= 30$).

For $i := 1$ To 30 do begin

read (valor);
total := total + valor;

end;

write ('...')

prom := total div 30;

end;

Operaciones elementales dentro del for:
 $2 * 30$.

$$2 \times 30 + (3 \times 30 + 2) + 3 = 155$$

$$(3n + 2)(n \text{ con opera elem})$$

Programa con módulos → cálculo de tiempo partiendo de aquellos que no llaman a otros

Después el T. Ejecución de los procesos que llaman a otros y así sucesivamente.

El T. E se puede ignorar si

El programa se va a utilizar algunas veces, el costo de escritura y de rutaación es el dominante.

un algoritmo eficiente pero complicado → costoso mantenimiento.

① read (valor);

② if (valor > 8).

then begin

suma := 0.

for i := 1 to 3000 do

 suma := suma + 1;

end;

③ else begin

 suma := 0.

 for i := 1 to 3000 do

 for j := 1 to 2000 do

 suma := suma + 1

 end;

 end;

FOR

1 de asignación de i

Total de comparaciones $N+1$

$N \cdot 2$ suma $i := i+1$ en C1 repetición

$N \cdot$ operaciones elementales dentro del for

$(3 \cdot N \text{ repes.} + 2) + N \text{ repes.} * \text{cant operaciones.}$

$$1 + (N+1) + 2N = 3N + 2$$

while

$((\text{cant op elem. dentro del while} * n) + (n + \text{cant. op elem. en la condición})) + \text{cant op elem. fuera del while.}$