CAPÍTULO I: CONCEPTOS BÁSICOS

- *Programa*: Es un conjunto de instrucciones, ejecutables sobre una computadora, que permite cumplir una función específica.
- *Dato*: Es una representación de un objeto del mundo real mediante la cual se pueden modelizar aspectos de un problema.
- *Abstracción*: Es la interpretación de los aspectos esenciales de un problema del mundo real, y la expresión de los mismos en términos precisos.
- *Precondición*: Es una información que se conoce como verdadera antes de iniciar el programa.
- *Poscondición*: Es una información que debiera ser verdadera al concluir el programa, si se cumple con lo pedido.
- *Especificación*: Es el objetivo que se desea que el programa realice.
- *Lenguaje de Programación*: Es el conjunto de instrucciones permitidas y definidas por sus reglas sintácticas y su valor semántico, para la expresión de soluciones de problemas.
- *Algoritmo*: Es la especificación rigurosa de la secuencia de instrucciones a realizar sobre una computadora, para alcanzar el resultado deseado en un tiempo finito.

CAPÍTULO II: ALGORITMOS

1) ESTRUCTURAS DE CONTROL

Son aquellas estructuras que controlan el *flujo de control* de un programa.

Se encuentran divididas en 5 categorías:

- *Secuencia*: Estructura compuesta por una sucesión de operaciones, en la que el orden de ejecución coincide con el orden físico de aparición de las instrucciones.
- *Decisión*: Estructura que permite elegir entre dos alternativas (bifurcación) dependiendo de condición (expresión booleana: *True* o *False*) definida en ella.

  Pueden ser bifurcaciones simples, en cascada o anidadas:

Si (Condición) entonces
  <<Acciones por condición verdadera>>
Sino
  <<Acciones por condición falsa>>
Fin Si
	If (A < B) then
	  Min := A
	Else
	  Min := B;

Si (Condición1) y (Condición2) entonces
  <<Acciones por condiciones verdaderas>>
Sino
  Si (Condición1) y (Condición3) entonces
  <<Acciones por condiciones verdaderas>>
  Sino
  <<Acciones por al menos una condición falsa>>
  Fin Si
Fin Si
	If (A < B) and (A < C)
	  Min := A
	Else
	  If (B < A) and (B < C) then
	  Min := B
	  Else
	  Min := C;

Si (Condición1) entonces
  Si (Condición2) entonces
  <<Acciones por condición verdadera>>
  Sino
  <<Acciones por condición falsa>>
  Fin Si
Fin Si
	If (A < B) then
	  Begin
	  If (A < 5) then
	  Min5 := A
	  Else
	  Min5 := -1;
	  End;

- *Selección*: Estructura que permite elegir entre dos o más alternativas dependiendo de la variable ordinal que interviene.

Caso (Variable) de
  Caso1: <<Acciones por Caso1 verdadero>>
  Caso2: <<Acciones por Caso2 verdadero>>
  Caso3: <<Acciones por Caso3 verdadero>>
  Sino
  <<Acciones no contempladas por los casos>>
Fin Caso
	Case Nro of
	  1: Writeln ('Uno');
	  2: Writeln ('Dos');
	  3: Writeln ('Tres')
	  Else
	  Writeln ('Otro');
	End;

Iteración: Estructura que se ejecuta un número desconocido de veces dependiendo de la condición definida en ella.

La iteración puede ser pre-condicional (se evalúa la condición primero) o pos-condicional (se evalúa la condición al final). En el primer caso, si la condición es verdadera, se ejecutan todas las instrucciones inmediatas, haciendo que el bloque itere 0, 1 ó más veces. En el segundo caso, el bloque deja de ejecutarse cuando la condición es verdadera, por lo tanto, se itera al menos una vez todas las instrucciones inmediatas.

Mientras (Condición) hacer <<Acciones por Condición verdadera>>	Readln (A); While (A <= 0) do

	Readln (A);
Repetir <<Acciones por Condición falsa>> Hasta (Condición)	Repeat Readln (A); Until (A > 0);

- *Repetición*: Estructura que se ejecuta un número fijo de veces, que puede ser conocido de antemano (`ValorMáximo - ValorMínimo + 1`), hasta que `Índice` supere `ValorMáximo` (o `ValorMínimo`). `Índice` se incrementa (o decrementa) al siguiente valor de acuerdo al tipo de variable ordinal asociada; en algunos lenguajes el incremento (o decremento) es automático y en otros, se puede establecer.

  El valor de `Índice` solo se utiliza dentro de está estructura, por lo que, en Pascal, no es posible modificarla y el valor de la misma se pierde al terminar la repetición.

Para Índice := ValorMínimo a ValorMáximo hacer <<Acciones>>	For X := 1 to 10 do Writeln (X);
Para Índice := ValorMáximo a ValorMínimo hacer <<Acciones>>	For X := 10 downto 1 do Writeln (X);

2) ESTRUCTURA ESQUEMÁTICA DE UN PROGRAMA EN PASCAL

```
Program IdentificadorPrograma;
Uses NombreUnidades;

Const
   {Declaraciones de constantes globales}

Type
   {Declaraciones de tipos de datos globales definidos por el programador}


Procedure IdentificadorProcedimiento (Parámetros);
Begin
   {Instrucciones ejecutables}
End;

Function IdentificadorFuncion(Parámetros): ValorRetornado;
Begin
      {Instrucciones ejecutables}
End;

Var
   {Declaraciones de variables globales}


Begin
   {Instrucciones ejecutables}
End.
```

- *Program*: Indica el inicio del programa.
- *IdentificadorPrograma*, *IdentificadorProcedimiento*, *IdentificadorFuncion* y *NombreUnidades*: Identifican a los procedimientos y funciones, así como a la aplicación y a las unidades. No puede repetirse ningún identificador cualquiera sea su pertenencia.

- `Uses`: Permite utilizar los procedimientos, funciones, variables, constantes y/o tipos de datos definidos que pertenezcan a otras unidades.
- `Const`: Permite definir datos que no variaran a lo largo de la aplicación.
- `Type`: Permite definir tipos de datos propios utilizando los definidos por el lenguaje.
- `Var`: Permite definir las variables que se utilizarán, indicando el tipo de dato a la que pertenecen.
- `Procedure` y `Function`: Módulos que contienen algoritmos. Su utilización facilita el desarrollo de los programas y agrega legibilidad al mismo.
- `Begin` y `End`: Establecen el inicio y fin de los bloques de instrucciones. Cuando aparece un ";" después del `End` se indica el fin de un bloque común, pero cuando se encuentra un "." Se indica el fin de la aplicación.

3) IMPORTANCIA DE LA DOCUMENTACIÓN DE UN ALGORITMO

Un algoritmo bien documentado, permite que su mantenimiento sea más rápido y sencillo, gracias a que su legibilidad está beneficiada por los comentarios lógicos y por las nomenclaturas de los identificadores:

- *Comentarios Lógicos*: Aportan una gran legibilidad al código, dado que su único propósito es ese, especificando, de la manera más completa posible, cual es el problema a resolver. Siendo su lectura lo suficientemente clara como para entender cuales son las acciones que llevan a cabo los herederos de estos.

  Los comentarios son una parte importante del programa por lo tanto, su uso es altamente recomendado.

- *Nomenclaturas*: Aportan legibilidad al código cuando estas son autoexplicativas (deben estar relacionadas con el propósito del módulo correspondiente).

4) IMPORTANCIA DE LA VERIFICACIÓN

Un algoritmo es ***eficaz*** (correcto) cuando se cumplen con las metas pautadas. Por lo tanto, se debe tener una especificación completa, precisa y libre de ambigüedades del problema a resolver.

A través de pruebas con datos reales, se puede comprobar si el algoritmo escrito, cumple con los requisitos pedidos.

5) EFICIENCIA DE UN ALGORITMO

Un algoritmo es ***eficiente*** cuando utiliza de la mejor manera posible, los recursos a su disposición (tiempo y memorias).
Dado que existen varias soluciones posibles para un problema, la solución más eficiente es aquella que se realiza en el menor tiempo posible, con la menor cantidad de pasos y uso de memorias.

6) DESCOMPOSICIÓN DE PROBLEMAS

MODULARIZACIÓN

También conocida como ***Top-Down***, consiste en descomponer el problema principal en sub-problemas o módulos, y su vez, estos sub-problemas en sub-problemas más pequeños. A tal punto de que las resoluciones de los mismos sean independientes y más simples; y la suma de estos, la solución del problema principal.
Por esto, la ***importancia*** de la modularización, sobretodo una buena, recae en 4 conceptos básicos:

- *Conocimiento Humano*: Dada la limitada capacidad de retención de información que el ser humano posee, la resolución de un problema resulta más fácil si esta se realiza por partes.
- *Trabajo en equipo*: Permite que un programa complejo se desarrolle más rápido, dado que las diversas tareas se dividen entre los programadores, haciendo que las soluciones de estas se haga en paralelo.

  El programador solo se preocupa por solucionar su problema, y si necesita utilizar los módulos creados por sus pares, lo hace con las instrucciones y limitaciones correspondientes.

- *Mantenimiento del programa*: Tanto para corregir los errores como para modificar el código, es más costoso mantener un programa que no ha sido correctamente modularizado de aquel que sí lo fue.
- *Reusabilidad del código*: Aquellos códigos que fueron correctamente modularizados, pueden ser reutilizados, siempre y cuando sirvan en la resolución de otros problemas, tanto en el mismo programa como en otros.

Cuando un módulo se ejecuta, este toma el control del programa hasta que su liberación (cuando se ejecuta la última sentencia del mismo) retorna el control a la instrucción siguiente a la invocación (en el caso de los procedimientos) o la misma que la invoco (en el caso de las funciones). Tanto las funciones como los procedimientos deben ser declarados antes de ser utilizados.

ALCANCE DE LOS DATOS

- *Dato Global*: Es aquel que está declarado en la sección `Var` del programa. Su uso está recomendado cuando se necesitan datos que son utilizados por varios módulos.
- *Dato Local*: Es aquel que está declarado dentro de un módulo, en la correspondiente sección `Var`, y su uso esta restringido a dicho módulo.

Un dato perteneciente a un módulo está oculto, porque no puede ser utilizado por otros y por lo tanto está protegido.

PARÁMETROS

Son variables que se utilizan para transferir información entre los módulos, por lo tanto, la alternativa al uso de variables globales.

Existen dos tipos de parámetros:

- *Parámetro por Valor*: Son aquellos que se utilizan cuando se quiere mandar una copia del dato.

  De esta manera, el dato está protegido de las modificaciones que realice el módulo receptor, ya que no afecta a la variable pasada (parámetro actual), sino a la copia de la misma.

  También son conocidos como parámetros de Entrada o de "Solo lectura", porque envían información al módulo.

- *Parámetro por Referencia*: Son aquellos se utilizan cuando se quiere modificar el valor de la variable pasada o cuando se quiere obtener información de un módulo.

  En estos casos, no se envía una copia sino la dirección de memoria de la misma. Por lo tanto, los cambios efectuados en ella se ven reflejados al finalizar el módulo.

  El parámetro por referencia, también conocido como parámetro de Salida, de Entrada/Salida, de "Solo Escritura" o de "Lectura y Escritura", se declara anteponiendo, al identificador del parámetro formal, la palabra `Var`.

El uso de parámetros versus variables globales está fundado en 4 conceptos:

- *Integridad de los Datos*: Se puede identificar cuales son las variables globales que usan y/o comparten entre sí los módulos, así como también el uso que hacen de ellas, tanto para lectura como para escritura, o lectura y escritura.

- *Protección de los Datos*: Se evita que se modifique por accidente el valor de una variable, si esta es pasada como un parámetro en lugar de actuar directamente sobre ella.

- *Utilidad del uso de Parámetros*: Permite que los programadores desarrollen los módulos sin tener que preocuparse por el nombre de los parámetros actuales, o identificadores, que sus pares declararon. Por lo tanto facilita el desarrollo de programas modularizados y la corrección de los posibles errores en ellos.

- *Reutilización de códigos*: Permite que un módulo pueda ser reutilizado varias veces (usando distintas listas de parámetros actuales) dado que los parámetros declarados en él, separan el nombre del dato del dato en sí.

Los parámetros se declaran en la cabecera de los módulos, indicando nombre y tipo de dato.

Existen dos formas de correspondencia entre parámetros:

- *Por posición*: En Pascal, los parámetros actuales deben coincidir con los tipos de los parámetros formales. Correspondiendo el primer parámetro actual con el primero formal, el segundo actual con el segundo formal, y así sucesivamente.
  También deben coincidir en número; la cantidad de parámetros actuales debe ser igual a la cantidad de parámetros formales, de lo contrario, en Pascal, se abortará el programa.

- *Por nombre*: En otros lenguajes, se puede indicar en la invocación del módulo, que parámetro formal corresponderá a que parámetro actual.

CAPITULO III: TIPOS DE DATOS SIMPLES

1) TIPOS DE DATOS

Es una clase de objetos ligados a un conjunto de operaciones para crearlos y manipularlos.

Poseen un rango de valores posibles, un conjunto de operaciones realizables y representación interna, todos ellos, predefinidos y acotados por el lenguaje de programación. Cuando se declara uno, se está indicando cual es la clase de valores que poseerá y cuales son las operaciones que se pueden hacer sobre él.

Los tipos de *datos simples* se clasifican en:

TIPO DE DATO NUMÉRICO

Es un conjunto de valores numéricos que se pueden representar de dos formas:

- *Enteros*: Es el tipo de dato numérico más simples y posee un rango de representación limitado por el número de bytes disponibles: en los enteros con signo, el rango es de `-maxint <= n <= maxint`, siendo `n` el número que se quiere representar. En los enteros sin signo, el rango es de `0 <= n <= maxint`.

- *Reales*: Es el tipo de dato numérico que permiten representar números fraccionarios bajo la norma IEEE 754. El rango de los reales depende de la disponibilidad de bytes que se tengan, por lo tanto, su representación es limitada y aproximada.

Las *operaciones* válidas que se realizan sobre este tipo de dato son:

- *Asignación*: Mediante `:=` se puede asignar un valor o el contenido de una variable numérica a otra.

- *Aritméticas*: **Suma (+), Resta (-), División entera (`Div`), Módulo (`Mod`) y División real (`/`).**

  Tanto la suma, la resta y la multiplicación aceptan operandos enteros como reales y, por lo tanto, su resultado puede ser entero o real. Sin embargo, la división entera restringe a los operandos, debiendo ser estos enteros. Se utiliza `Div` (que devuelve el resultado entero) y `Mod` (que retorna el resto), y el resultado es un entero. En el caso de la división real, no existe ninguna limitación con los operandos. Se utiliza `/` (que devuelve el resultado con parte fraccionaria) y el resultado puede ser entero o real.

  Cuando se realizan estas operaciones, el resultado de las mismas puede estar fuera del rango de representación. Si se supera el máximo valor binario permitido, se produce un *Overflow*. Si el número está por debajo del mínimo valor binario permitido, se produce un *Underflow*. Ambos, son desbordamientos que algunos sistemas los detectan como errores y otros los ignoran convirtiéndolos en un resultado incorrecto dentro del rango del tipo.

  El ***orden de precedencia***, cuando existen varios operandos en la expresión, está regido por las mismas reglas que en las Matemáticas:

  - Los paréntesis `( )`, que alteran el orden en las operaciones.
  - Operadores `*, /, Div y Mod`
  - Operadores `+, -`.

- *De Comparación*: **Igualdad (=), Desigualdad (<>), Menor que (<), Menor o Igual que (<=), Mayor que (>) y Mayor o Igual que (>=).**

  El resultado de estas operaciones es un booleano.

TIPO DE DATO LÓGICO

Es un tipo de dato que puede tomar uno de los dos valores posibles, Verdadero (*True*) o Falso (False), pero no ambos al mismo tiempo.

Se utiliza en casos en donde se representan dos alternativas a una condición.

Las *operaciones* válidas que se realizan sobre este tipo de dato son:

- *Asignación*: Mediante := se puede asignar un valor o el contenido de una variable booleana a otra.
- *Lógicas o Booleanas*: **Negación (`Not`), Conjunción (`And`), Disyunción Incluyente (`Or`) y Disyunción Excluyente (`XOr`).**

  El resultado de estas operaciones es un booleano:

P	Q	P And Q
True	True	True
True	False	False
False	True	False
False	False	False

P	Q	P Or Q
True	True	True
True	False	True
False	True	True
False	False	False

P	Q	P XOr Q
True	True	False
True	False	True
False	True	True
False	False	False

P	Not P
True	False
False	True

El *orden de precedencia* para los operadores lógicos es:

- Los paréntesis ( ), que alteran el orden en las operaciones.
- Operador `Not`
- Operador `And`
- Operadores `Or` y `XOr`.

- *De Comparación*: **Igualdad (=), Desigualdad (<>), Menor que (<), Menor o Igual que (<=), Mayor que (>) y Mayor o Igual que (>=).**

  El resultado de estas operaciones es un booleano.

TIPO DE DATO CARACTER

Es un tipo de dato que contiene solo un elemento como su valor, establecido y normatizado por los estándares ASCCI, EBCDIC, ANSI, etc.; siendo ASCII el más común a escala internacional.
Tanto los elementos como el *orden de precedencia* de los mismos, está establecido por el estándar que se utilice. Sin embargo, los caracteres se dividen en 4 grupos comunes:

- *Letras Minúsculas*: 'a', 'b', 'c',... 'x', 'y', 'z'.
- *Letras Mayúsculas*: 'A', 'B', 'C',... 'X', 'Y', 'Z'.
- *Dígitos*: '0', '1', '2',... '8', '9'.

- *Caracteres Especiales*: '%', '@', '#', '¬', '¿', '?',...

Las *operaciones* válidas sobre este tipo de dato son:

- *Asignación*: Mediante := se puede asignar un valor o el contenido de una variable caracter a otra.

- *De Comparación*: **Igualdad (=), Desigualdad (<>), Menor que (<), Menor o Igual que (<=), Mayor que (>) y Mayor o Igual que (>=).**

  Las comparaciones entre caracteres se realizan por el valor ordinal de los mismos, siendo el resultado de estas un booleano.

2) CONSTANTES Y VARIABLES

- *Variable*: Es una posición de memoria que almacena información que puede variar a lo largo de la ejecución del programa.

  La ***declaración***, en Pascal, se logra anteponiendo al identificador de esta, la palabra reservada `Var`. Y colocando detrás de este, los : más el tipo de dato (*Integer*, *Real*, *Boolean*, *Char*...):

```
Var NombreVariable: TipoDato;
```

  La ***asignación*** de valores, en Pascal, se hace de la manera:

```
NombreVariable := Expresión;
```

  Donde `NombreVariable` se encuentra a la izquierda e indicando cual es la variable que recibirá a `Expresión`, que se encuentra del lado derecho. Ambos, deben pertenecer al mismo tipo de dato.

- *Constante*: Es una posición de memoria que almacena información que no varia a lo largo de la ejecución del programa.

  La ***declaración***, en Pascal, se logra anteponiendo al identificador de esta, la palabra reservada `Const`. Y colocando detrás de este, el signo = más `Expresión`:

```
Const NombreConstante = Expresión.
```

El tipo de dato (*Integer*, *Real*, *Boolean*, *Char*...) no necesita ser declarado, aunque queda implícito.

Tanto las variables como las constantes, utilizan ***Identificadores***, que las referencian en lugar de la posición real de memoria y su valor, y deben ser declarados antes de poder ser utilizados. Los identificadores están formados por letras, dígitos y algunos caracteres especiales, con la restricción que el primer carácter debe ser una letra o un "_".

3) TIPOS ORDINALES

Un tipo de dato es ordinal si para cada elemento, existe uno anterior y otro posterior (siempre y cuando no se trate de los extremos del rango), que pertenezcan al mismo tipo de dato.
Por lo tanto, los enteros, los caracteres y los booleanos son tipos de datos ordinales, y pueden ser utilizados como variables de control en las estructuras de control. En cambio, los reales quedan excluidos porque no es posible saber el antecesor o el sucesor de ellos, ya que estos son infinitos.

4) TIPOS DE DATOS DEFINIDOS POR EL USUARIO

Son aquellos tipos de datos que son definidos por el usuario, a través de la palabra reservada `Type`, y que no existen en la definición del lenguaje.

Por lo tanto, es el usuario el que se encargará de establecer el conjunto de valores permitidos y las operaciones válidas sobre él.

Cuando un lenguaje permite definir tipos de datos propios, y permite usarlos y validarlos del mismo modo que los tipos de datos estándares, se tiene un lenguaje con:

- *Mejor posibilidad de abstracción de datos*: Permitiendo una mayor riqueza expresiva del lenguaje, ya que agrega claridad a la lectura del programa.
- *Mayor seguridad en las operaciones*: Permitiendo un mayor número de validaciones sobre el tipo del dato.
- *Limitaciones de los valores*: Prestableciendo los valores posibles que pueden tomar las variables pertenecientes al tipo de dato propio.
- *Mayor Flexibilidad*: Si se desea modificar la representación de la información contenida en un tipo de dato propio, basta con cambiar la declaración del tipo, en lugar de alterar una serie de declaraciones de variables.
- *Mayor Seguridad*: Reduciendo los errores de correspondencia entre el valor que se pretende asignar y el previamente declarado.

La ***declaración*** de tipos es de la forma:

```
Type NombreTipo = TipoBase.
```

Donde `NombreTipo` es el identificador con el se conocerá al tipo en el programa, y `TipoBase` puede ser un tipo de dato predefinido o estándar. Este identificador puede ser usado en declaraciones de tipos, variables y módulos posteriores.

TIPO DE DATO ENUMERATIVO

Es un tipo de dato escalar formado por valores constantes *strings* simbólicos.

Este tipo de dato es utilizado cuando se quiere brindar claridad y legibilidad a los programas. Dado que a veces, existen objetos que son más fáciles de comprender si poseen sus propios nombres.

La ***declaración*** de los enumerativos, en Pascal, se logra encerrando entre paréntesis los valores a enumerar:

```
Type NombreTipo = (Valor1, Valor2,… ValorN);
```

Donde `Valor1, Valor2,… ValorN`, son constantes simbólicas que, en Pascal, no pueden repetirse en las definiciones de otros enumerativos.

Las ***operaciones*** válidas sobre este tipo son:

- *Asignación*: Mediante := se puede asignar uno de los valores constantes correspondientes a los definidos en el tipo perteneciente.
- *Comparación*: **Igualdad (=) y Desigualdad (<>).**

  El resultado de estas operaciones es un booleano.

No se pueden realizar, en Pascal, operaciones de E/S.

TIPO DE DATO SUB-RANGO

Es un tipo de dato ordinal que está formado por una secuencia contigua de valores ascendentes de algún tipo ordinal (tipo base del sub-rango).

Este tipo de dato es utilizado cuando no se requiere trabajar con todos los valores que un estándar ordinal provee.

La *declaración* se determina mediante el límite inferior y el superior de la secuencia, separados por dos puntos:

```
Type NombreTipo = LimiteInferior..LimiteSuperior

Var NombreVariable: LimiteInferior..LimiteSuperior.
```

Siendo `LimiteInferior` menor que `LimiteSuperior`, de lo contrario la declaración es ilegal.

Las *operaciones* válidas para este tipo de dato son las mismas que las de su tipo base.

TIPO DE DATO CONJUNTO

Es una colección de datos simples, sin repeticiones (en caso de haberlas se ignoran) ni relación de orden, pertenecientes al mismo tipo ordinal (enteros, caracteres o enumerativos). Siendo su limite, en Pascal, de 256 elementos.

La *declaración* se logra mediante la forma:

```
Type NombreTipo = Set of TipoOrdinal

Var NombreVariable: Set of TipoOrdinal
```

Donde `TipoOrdinal` es el tipo al que pertenecen los elementos del conjunto.

La *construcción* se establece definiendo los elementos individuales consecutivamente, encerrados entre corchetes `[]` y separados por comas:

```
Conjunto := [Elemento1, Elemento2,... ElementoN].
```

Los elementos del conjunto, pueden ser también variables que pertenezcan al mismo tipo base de este. Además, se puede definir varios elementos consecutivos mediante `..`:

```
Conjunto := [ElementoK..ElementoK+I].
```

Un conjunto está vacío si fue declarado de las siguientes maneras:

- `Conjunto := []; {Sin contenido}`
- `Conjunto := ['Z'..'A']; {Con orden inverso}`

Las *operaciones* válidas sobre este tipo de dato son:

- *Asignación*: Mediante `:=` se puede asignar un valor o el contenido de un conjunto a otro.

- *Aritméticas*: **Unión (+), Intersección (*), Diferencia (-) y Pertenencia (In).**

  Tanto la unión, como la intersección y la diferencia retornan otro conjunto. Mientras que pertenencia devuelve un booleano.

- *De Comparación*: **Igualdad (=) y Desigualdad (<>).**

  El resultado de estas operaciones es un booleano.

Sobre los conjuntos no es posible realizar operaciones de lectura y de escritura.

TIPO DE DATO *STRING*

Es una secesión de caracteres que se almacenan en un área contigua de la memoria principal, que puede ser leída y/o escrita.

Posee un tamaño dado por la cantidad de caracteres que la variable contiene, siendo está su longitud. La ***máxima capacidad***, en Pascal, es de 255 caracteres. En memoria, ocupa `Longitud+1` caracteres, dado que la posición cero de la misma contiene la `Longitud`.

En la ***declaración*** de un *string*, se puede especificar el tamaño del mismo, pudiendo ser menor, el contenido real de este al establecido:

```
Type NombreTipo = String[Longitud]

Var NombreVariable = String[Longitud]
```

Posición	0	1	2	...	255
Contenido	Longitud	A	n		5

Donde `Longitud` es el número máximo de caracteres que puede contener. Si no se establece una longitud, por defecto, el tamaño del *string* es la máxima capacidad.

Las ***operaciones*** válidas sobre este tipo de dato son:

- *Asignación*: Mediante `:=` se puede asignar un valor o el contenido de un *string* a otro, truncando los caracteres sobrantes si se excede la capacidad del *string* receptor.

- *Concatenación*: Permite adosar un *string* a continuación de otro:

  - *Encolar*: `Str1 := Str1 + Str2;`
  - *Apilar*: `Str1 := Str2 + Str1;`

- *De Comparación*: **Igualdad (=), Desigualdad (<>), Menor que (<), Menor o Igual que (<=), Mayor que (>) y Mayor o Igual que (>=).**

  La Igualdad retorna verdadero, si ambos *strings* poseen la misma Longitud y contienen los mismos caracteres en el mismo orden, caso contrario devuelve falso. La Desigualdad funciona de la misma manera que la Igualdad. Las demás operaciones, comparan carácter por carácter, según su valor ordinal ASCII, hasta determinar un resultado. El resultado de estas operaciones es un booleano.

CAPITULO IV: PROCEDIMIENTOS, FUNCIONES Y PARÁMETROS

1) MÓDULOS

Conjunto de instrucciones y de definiciones de datos que realiza una tarea lógica.

Pueden realizar las mismas acciones que un programa. La diferencia está en que el módulo se ejecuta cada vez que se lo invoca desde el programa principal o desde otro; dándole el control durante su ejecución y retornándolo al finalizar.

Cada módulo debe tener un identificador único, con el cual será llamado, y puede poseer una serie de parámetros asociados a él, con los que se enviará y/o recibirá información.

El *alcance* de un módulo está regido por el contexto (visibilidad en el mismo nivel) de este y, a su vez, depende de la ubicación dentro de la definición de las rutinas.

Un módulo conoce todo lo que anteriormente fue definido a él y todo lo que está definido en su interior. El programa principal no puede invocar a los módulos que estén dentro de otros módulos.

2) PROCEDIMIENTOS

Es un conjunto de instrucciones que realizan una tarea específica y, como resultado, pueden retornar o no, uno o más valores como resultado.

Su invocación en el programa, u otro módulo, es una simple línea: el nombre del módulo.

```
Procedure Indentificador (Var Parametro1: TipoBase; Parametro2: TipoBase);
Const
   {Constantes Locales}

Type
   {Tipos Locales}

Var
   {Variables Locales}

Begin
   {Instrucciones}
End;
```

Los Procedimientos pueden recibir tanto parámetro por referencia (`Var`) como por valor.

3) FUNCIONES

Es un conjunto de instrucciones que realizan una tarea específica y, como resultado, retorna siempre una única respuesta.

Su invocación en el programa, u otro módulo, es a través de una asignación y/o comparación. También, el resultado de una función puede ser impreso.

```
Function Indentificador(Parametro1, Parametro2: TipoBase): ValorRetornado;
Const
   {Constantes Locales}
```

```
Type
   {Tipos Locales}

Var
   {Variables Locales}

Begin
   {Instrucciones}
End;
```

Las funciones solamente deben recibir parámetro por valor, aunque pueden recibir por referencia en el caso de los arreglos, para no violar el concepto de función: *retornar un único valor*. Y su valor debe ser siempre consumido.

CAPITULO V: ESTRUCTURAS DE DATOS COMPUESTOS

1) CONCEPTOS BÁSICOS

Una estructura de datos es un conjunto de variables, que pueden no ser de mismo tipo, relacionadas entre sí de diversas formas.

Construidos a partir de datos simples o estructurados, la rentabilidad de las estructuras compuesta, recae en poder representar elementos del mundo real más complejos que los datos simples.
Estas estructuras pueden clasificarse por dos criterios:

- *Datos constructores*:
  - *Homogéneas*: Los datos que componen la estructura son del mismo tipo.
  - *Heterogéneas*: Los datos que componen la estructura son de distintos tipo.
- *Espacio en memoria*:
  - *Estáticas*: La cantidad de elementos es fija, por lo tanto, la cantidad de memoria que se utiliza no varía a lo largo de la ejecución de un programa.

    La ***desventaja*** de estás estructuras esta en la cantidad fija de memoria, ya que se puede tanto malgastar el espacio reservado, si sobra, así como también, quedarse sin él para más datos. Sin embargo, la ***ventaja*** recae en el acceso directo a los datos, ya que las posiciones de los mismos se pueden calcular como desplazamientos de la posición inicial de la estructura, dado que se encuentran físicamente consecutivos.
  - *Dinámicas*: La cantidad de elementos puede variar a lo largo de la ejecución del programa, por lo tanto, la cantidad de memoria también.

    La ***ventaja*** de estás estructuras esta en la cantidad variable de memoria, ya que se reserva la necesaria para cada elemento a medida que el algoritmo lo requiere. Sin embargo, las ***desventajas*** recaen en el acceso secuencial a los datos (debido a que las posiciones de memoria de estos no se encuentran físicamente consecutivas, sino lógicamente) y en el tiempo extra de procesamiento que el lenguaje y el sistema operativo realizan (debido a la reserva y la liberación de memoria).

2) REGISTROS

Son estructuras de datos, estáticas y heterogéneas, que permiten agrupar diferentes tipos de datos con una conexión lógica y en una única estructura. Los datos contenidos, son conocidos como *campos* y cada uno de ellos posee un identificador único dentro de la misma.

La ***declaración*** de los registros es de la forma:

```
Type
   NombreTipo = Record
                   Campo1: TipoCampo1;
                   Campo2: TipoCampo2;
                   ...
                   CampoN: TipoCampoN;
                End;

Var
```

```
NombreVariable = Record
                    Campo1: TipoCampo1;
                    Campo2: TipoCampo2;
                    ...
                    CampoN: TipoCampoN;
                 End;
```

Donde `Campo1`, `Campo2`... `CampoN` son los identificadores de los campos y `TipoCampo1`, `TipoCampo2`... `TipoCampoN` los tipos correspondientes.

El ***acceso a los campos*** de un registro es a través del nombre del registro, seguido por el punto y el nombre del campo:

```
NombreRegistro.NombreCampo
```

Esto es lo que se denomina ***calificar*** el campo. También se pude acceder a los campos a través de la sentencia `With`:

```
With NombreRegistro do
   begin
      Campo1 := Valor1;
      Readln (Campo2);
      Writeln (Campo1 + Campo2);
   end;
```

El uso de `With` recae en evitar la tediosa tarea de la calificación de campos, ya que nombrando el registro una sola vez, se puede acceder a ellos como variables ordinarias.

Las ***operaciones*** válidas sobre este tipo de dato son:

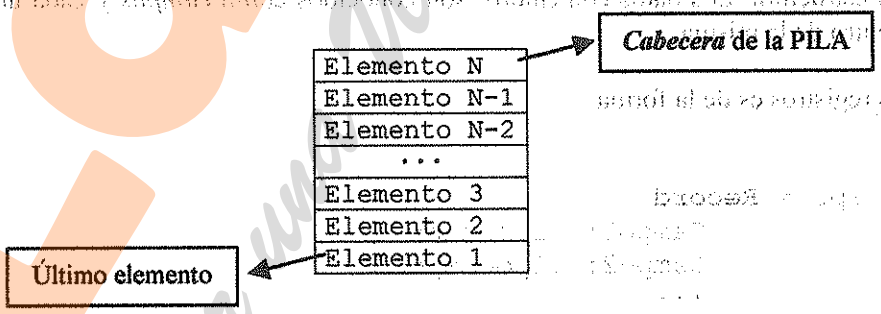
- *Asignación*: Mediante := se puede asignar un valor a un campo o el contenido entero de un registro a otro, siempre y cuando pertenezcan al mismo tipo.

Las demás operaciones están restringidas a los campos del registro y dependen del tipo de los mismos. Por lo tanto, las comparaciones como la aritmética entre registros completos están prohibidas.

3) PILAS

Es una colección ordenada, dinámica y homogénea de elementos, que retorna los datos almacenados en el orden inverso en el que fueron ingresados (*LIFO*: *Last In, First Out* – Último en Ingresar, Primero en Salir).

A partir de una posición en memoria, los datos se van depositando sucesivamente adelante del primer elemento, convirtiéndose estos en la ***cabecera*** o ***tope*** de la pila.



La ***declaración***, en Seudo-Pascal, de una pila es de la forma:

```
Type NombreTipo = TipoPila;      Var NombreVariable: TipoPila;
```

Donde `TipoPila` es alguno de los siguientes valores posibles:

- `Stack_Integer`
- `Stack_Real`
- `Stack_Char`
- `Stack_String` (De 255 caracteres)
- `Stack_Rec_TipoRegistro` (`TipoRegistro` es el identificador del tipo de registro deseado).

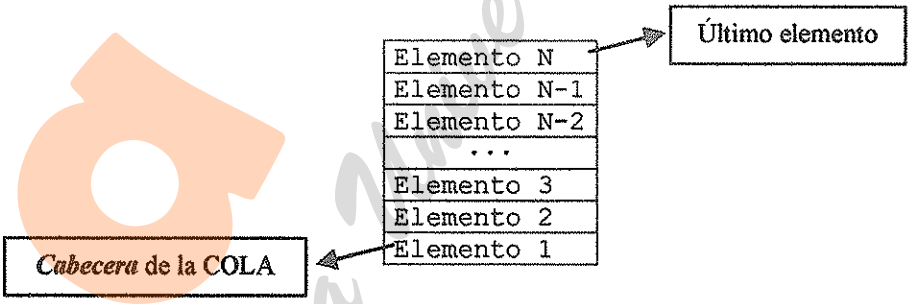
Las ***operaciones*** válidas sobre este tipo de datos son:

- *`St_Create (NombrePila)`*: Procedimiento que activa o crea a `NombrePila` asignándole una dirección de inicio en la memoria.
- *`St_Push (NombrePila, Elemento)`*: Procedimiento que agrega (apila) a `Elemento` en `NombrePila`, poniéndolo en el tope de la misma e incrementando en uno su tamaño actual.
- *`St_Pop (NombrePila, Elemento)`*: Procedimiento que quita (desapila) a `Elemento` de `NombrePila`, sacándolo del tope de la misma y decrementando en uno su tamaño actual. `NombrePila` debe contener un elemento como mínimo para que esta operación sea válida.
- *`St_Empty(NombrePila)`*: Función que retorna un valor booleano: *True* si `NombrePila` está vacía o *False* en caso contrario.
- *`St_Length(NombrePila)`*: Función que retorna el número (un entero) de elementos almacenados en `NombrePila`.
- *`St_Top(NombrePila)`*: Función que retorna el contenido del elemento que está en el tope de `NombrePila`, sin quitarlo de la misma.

4) COLAS

Es una colección ordenada, dinámica y homogénea de elementos, que retorna los datos almacenados en el mismo orden en el que fueron ingresados (*FIFO*: *First In, First Out* – Primero en Entrar, Primero en Salir).

A partir de una posición en memoria, los datos se van depositando sucesivamente detrás del último elemento.



La ***declaración***, en Seudo-Pascal, de una cola es de la forma:

```
Type NombreTipo = TipoCola;      Var NombreVariable: TipoCola;
```

Donde `TipoCola` es alguno de los siguientes valores posibles:

- `Queue_Integer`

- `Queue_Real`
- `Queue_Char`
- `Queue_String` (De 255 caracteres)
- `Queue_Rec_TipoRegistro` (`TipoRegistro` es el identificador del tipo de registro deseado).

Las ***operaciones*** válidas sobre este tipo de datos son:

- *`Q_Create (NombreCola)`*: Procedimiento que activa o crea a `NombreCola` asignándole una dirección de inicio en la memoria.
- *`Q_Push (NombreCola, Elemento)`*: Procedimiento que agrega (encola) a `Elemento` en `NombreCola`, poniéndolo al final de la misma e incrementando en uno su tamaño actual.
- *`Q_Pop (NombreCola, Elemento)`*: Procedimiento que quita (desencola) a `Elemento` de `NombreCola`, sacándolo del tope de la misma y decrementando en uno su tamaño actual. `NombreCola` debe contener un elemento como mínimo para que esta operación sea válida.
- *`Q_Empty (NombreCola)`*: Función que retorna un valor booleano: *True* si `NombreCola` está vacía o *False* en caso contrario.
- *`Q_Length (NombreCola)`*: Función que retorna el número (un entero) de elementos almacenados en `NombreCola`.
- *`Q_Top (NombreCola)`*: Función que retorna el contenido del elemento que está en el tope de `NombreCola`, sin quitarlo de la misma.
- *`Q_Bottom (NombreCola)`*: Función que retorna el contenido del elemento que está al final de `NombreCola`, sin quitarlo de la misma.

CAPITULO VI: DATOS COMPUESTO INDEXADOS - ARREGLOS

1) ARREGLOS

Es una colección ordenada, indexada, estática y homogénea de elementos, que retorna cualquier dato, indicándole la posición que ocupa el mismo dentro de la estructura.

Los arreglos son de la forma:

```
Type NombreTipo = Array [LimiteInferior..LimiteSuperior] of TipoDato;

Var NombreVariable: Array [LimiteInferior..LimiteSuperior] of TipoDato;
```

Donde `LimiteInferior` debe ser menor o igual que `LimiteSuperior`, y juntos conforman el Índice, que debe ser ordinal.

Pueden existir varios índices o dimensiones en un arreglo dependiendo del lenguaje y de la memoria que ocupe la estructura.

VECTORES

Es un tipo de dato arreglo de una dimensión o índice.

Elem. 1	Elem. 2	Elem. 3	Elem. 4	···	Elem. N-1	Elem. N

Cada *celda* del vector, ocupa un lugar en forma consecutiva a partir de la dirección inicial de memoria asignada. La forma de *acceder* a las celdas, es utilizando el nombre de la variable junto con los [ ] y la posición dentro del vector:

```
Vector[Posición]
```

Donde `Posición` debe ser un valor literal o una variable correspondiente con el tipo de Índice, así como también, el valor literal o variable que se le pudiera *asignar* a la celda.

La *definición* de un vector es de la forma:

```
Type NombreTipo = Array [Índice] of TipoDato;

Var NombreVariable: Array [Índice] of TipoDato;
```

Donde `Índice` es el rango de valores que se toman para direccionar a las celdas y, además, establece la cantidad de estas que contendrá (`LimiteSuperior - LimiteInferior + 1`).

Las *operaciones* válidas sobre este tipo de dato, son las correspondientes al tipo de dato al que pertenece, pero solo se realizan sobre las celdas y no sobre el vector en sí. La única excepción es la asignación entre vectores, y ambos deben tener la misma dimensión, el mismo `Índice` y `TipoDato`.

MATRICES

Es un tipo de dato arreglo de dos dimensiones o índices.

	Elem. 1, 1	Elem. 1, 2	...	Elem. 1, N-1	Elem. 1, N
	Elem. 2, 1	Elem. 2, 2	...	Elem. 2, N-1	Elem. 2, N
	Elem. 3, 1	Elem. 3, 2	...	Elem. 3, N-1	Elem. 3, N
	Elem. 4, 1	Elem. 4, 2	...	Elem. 4, N-1	Elem. 4, N
	...	...	...	...	...
	Elem. M-1, 1	Elem. M-1, 2	...	Elem. M-1, N-1	Elem. M-1, N
	Elem. M, 1	Elem. M, 2	...	Elem. M, N-1	Elem. M, N

Filas

Columnas

Cada ***celda*** de la matriz, ocupa un lugar en forma consecutiva a partir de la dirección inicial de memoria asignada. La forma de ***acceder*** a las celdas, es utilizando el nombre de la variable junto con los `[ ]` y las posiciones dentro de la matriz:

```
Matriz[Posición1, Posición2]
```

Donde `Posición1` y `Posición2` deben ser valores literales o variables correspondientes con el tipo de Índice, así como también, el valor literal o variable que se le pudiera ***asignar*** a la celda.

La ***definición*** de una matriz es de la forma:

```
Type NombreTipo = Array [Índice1, Índice2] of TipoDato;

Var NombreVariable: Array [Índice1, Índice2] of TipoDato;
```

Donde `Índice1` e `Índice2` son el rango de valores que se toman para direccionar a las celdas y, además, establecen la cantidad de estas que contendrá:

```
(LimiteSuperior1 - LimiteInferior1 + 1) * (LimiteSuperior2 - LimiteInferior2 + 1)
```

`Índice1` referencia a las filas de la matriz, mientras que `Índice2` a las columnas. Ambos, ordinales, pueden no pertenecer al mismo `TipoDato` o incluso poseer longitudes diversas. Por lo tanto, no se deben alterar el orden de los índices: no es lo mismo [1, 2] que [2, 1]. El primero direcciona a la celda de la fila 1, columna 1; mientras que el segundo, a la celda de fila 2, columna 1.

Las ***operaciones*** válidas sobre este tipo de dato, son las correspondientes al tipo de dato al que pertenece, pero solo se realizan sobre las celdas y no sobre la matriz en sí. La única excepción es la asignación entre matrices, y ambas deben tener las mismas dimensiones, los mismos índices (`Índice1` e `Índice2`) y `TipoDato`.

2) ARREGLOS COMO PARÁMETROS

Si bien se recomienda el uso de parámetros por valor, en los casos en los que no se modifica la estructura, con los arreglos es preferible el uso de parámetros por referencia. De este modo, no se duplican y se evita malgastar recursos, ya que la eficiencia en cuanto al uso de memoria y tiempo disminuye. La única precaución que se debe tomar es de no alterarlos si no se lo desea.

CAPITULO VII: RECURSIVIDAD

1) RECURSIVIDAD

Es una técnica de resolución de problemas en donde el resultado de estos, se obtiene a través de módulos que se invocan a sí mismos.

Cuando un módulo se invoca a sí mismo, reduce al problema P a una instancia de sí, P_1. Al volverse a invocar, esta instancia se reduce a otra, P_2, y después de otra invocación se reduce a P_3, y así sucesivamente. A medida que cada instancia se establece, se va alcanzando el *caso degenerado*, el cual corta la recursividad por ser una instancia distinta a las demás (ya que no vuelve a invocar al módulo), y además, es la solución de P. El caso degenerado debe figurar siempre, de manera implícita o explícita, de lo contrario no tendrían fin las invocaciones.

El ejemplo más claro de entender, es el de la búsqueda de una persona en una guía:

```
Proceso Buscar (Guía, Persona)
Inicio
    Si (Guía contiene una sola página) entonces
       BuscarPersonaEnPagina {Caso Degenerado}
    Sino
       Inicio
          Dividir Guía a la mitad
          Si (Persona está en la 1ª mitad de Guía) entonces
             Buscar (1ª mitad de Guía, Persona)
          Sino
             Buscar (2ª mitad de Guía, Persona)
          Fin
Fin
```

Aunque esta metodología es similar a la del diseño *Top-Down*, su diferencia recae en que el problema P_1 es de la misma naturaleza que el problema original P, pero más simple.

La ***desventaja*** de la recursividad recae en ineficiencia que conlleva las sucesivas invocaciones: mayor tiempo de ejecución y mayor gasto de memoria.

2) EJECUCIÓN DE UN PROGRAMA Y LA PILA DE EJECUCIÓN

La pila de activación es una estructura de datos que se comporta de la siguiente manera:

- Posee un tope y un fondo.
- Cada elemento es código más datos.
- Los elementos solo pueden ser sacados del tope uno por uno.

La pila se crea cuando comienza la ejecución de un programa, en donde cada elemento que se agrega a la misma, ocupará un segmento de la memoria que recibe el nombre de *frame*. Cuando el programa inicia, el primer *frame* se crea conteniendo dos clases de datos: los declarados en sección de declaración del programa principal y una variable que contiene al ***Contador de Programa***.

Cuando se invoca a un procedimiento o una función, se asigna un nuevo *frame* para ese módulo que permanecerá en la pila hasta que finalice se ejecución. Logrado esto, se libera la memoria reservada para ese *frame*, y el programa o módulo que había realizado la invocación retoma el control.

CAPITULO VIII: CONCEPTO DE EFICIENCIA

1) Análisis de eficiencia de un algoritmo

Los algoritmos pueden ser analizados bajo dos aspectos de eficiencia:

- *Tiempo de Ejecución*: Se consideran más eficientes aquellos algoritmos que cumplan con las especificaciones del problema en el menor tiempo posible.

  Para poder medir este tiempo de ejecución, se tienen en cuenta dos operaciones elementales: las ***comparaciones*** y las ***asignaciones***. Y puede suceder, por lo tanto, que un algoritmo varié su tiempo de ejecución dependiendo de los datos ingresados, ya que pueden realizar más o menos operaciones.

  Esto establece la existencia del ***mejor caso*** (menor tiempo), del ***caso promedio*** (tiempo promedio), y del ***peor caso*** (mayor tiempo). En general, se tiene mayor interés en el comportamiento del algoritmo en el peor caso o en el caso promedio, que en el mejor caso, dado que importa que el programa sea más rápido en los casos adversos que en los favorables. Por lo tanto, *no importa cuan largo sea un algoritmo, sino que cantidad de operaciones realice.*

  Existen dos formas de ***medir el tiempo de ejecución***:

  - *Análisis Teórico*: Consiste en calcular el número de comparaciones y asignaciones que requiere un algoritmo, sin importar cuál es la máquina en el que se ejecuta.

    Al someter a las diversas alternativas de solución de un problema al mismo análisis teórico, se obtiene cual es el algoritmo más eficiente y, teóricamente, más rápido. Dado que permite estimar el orden del tiempo de respuesta, que servirá como comparación relativa, sin depender de los datos de experimentación.

  - *Análisis Empírico*: Consiste en someter a un algoritmo a diversos juegos de datos a fin de calcular su tiempo de respuesta.

    Este análisis empírico depende de la máquina en el que se realiza y en mayor parte de los datos que se emplean, ya que pueden favorecer a un algoritmo, pudiendo no ser este el caso general, con lo cual las conclusiones del análisis puede ser erróneas.

- *Uso de la memoria*: Se consideran eficientes aquellos algoritmos que utilicen las estructuras de datos adecuadas de manera de minimizar la memoria ocupada. El énfasis está puesto en el volumen de información a manejar en memoria.

2) Algoritmos de Búsqueda

Son algoritmos que permiten ubicar información en una colección de datos.

Existen dos estilos de búsqueda:

- *Búsqueda Lineal o Secuencial*: Consiste en recorrer el vector en forma secuencial desde el inicio, e ir analizando uno a uno los elementos contenidos hasta encontrar el dato deseado o hasta llegar al final de la misma.

  Esta búsqueda es utilizada cuando una estructura no posee o se le conoce un orden.

- *Búsqueda Binaria o Dicotómica*: Consiste en comparar el elemento buscado con el que se encuentra en la mitad de el vector: si coinciden, la búsqueda termina, si no, se analiza si el elemento solicitado se encuentra en la "primera mitad" (si es menor que el del medio) o en la "segunda mitad" (si es mayor que el del medio).

Después de desechar una de las dos mitades, se vuelve a repetir el mismo procedimiento sobre la mitad elegida, y de no hallarse el elemento buscado, se continua dividiendo hasta que la porción de estructura sea tan pequeña que no puede almacenar datos en ella o hasta haber localizar el elemento buscado.

Esta búsqueda se utiliza cuando el vector posee un orden.

3) Algoritmos de Ordenación

Son algoritmos por los cuales, un grupo de elementos pueden ser ordenados.

Se dividen en dos categorías:

- *Ordenación Ineficiente*: Son aquellos que son no recursivos.
  - *Selección*: Consiste en buscar, de a uno, los elementos menores (o mayores dependiendo del criterio de ordenación) y colocarlos al comienzo del vector, en posiciones consecutivas.

    Se busca el primer menor y se lo intercambia con el con la primera posición, luego se busca el segundo menor y se lo intercambia con la segunda posición; así sucesivamente hasta que el elemento `n-1` ocupe la penúltima posición. El último elemento, el mayor (o menor), queda ubicado en la última posición.

    Este método realiza `N-1` pasadas para ordenar la estructura.

```
Procedure Seleccion (Var Vector: Array of Integer; NroEle: Integer);
Var
   X, Y, Z, Pos: Integer;
   Aux: Integer;

Begin
   Z := NroEle - 1;
   For X := 1 to Z do
      begin
         Pos := X;
         For Y := X+1 to NroEle do
            If Vector[Y] < Vector[Pos] then
               Pos := Y;

         Aux := Vector[Pos];
         Vector[Pos] := Vector[X];
         Vector[X] := Aux;
      end;
End;
```

- *Burbujeo o Intercambio*: Consiste en comparar los elementos adyacentes del vector y moverlos si estos están desordenados.

  Al igual que la *selección*, el burbujeo realiza `N-1` pasadas para ordenar la estructura. Pero solo muevo los elementos a una distancia de una posición. Por lo tanto, después de una pasada, el elemento mayor (o menor dependiendo del criterio de ordenación) es arrastrado al final del vector. En la segunda pasada, el segundo mayor es arrastrado a la penúltima posición; así sucesivamente hasta que el elemento menor (o mayor) se encuentre en la primera posición.

```
Procedure Burbujeo (Var Vector: Array of Integer; NroEle: Integer);
Var
   X, Y, Z, Pos: Integer;
   Aux: Integer;

Begin
   For X := NroEle downto 2 do
```

```
            begin
                Z := X - 1;
                For Y := 1 to Z do
                    begin
                        Pos := Y + 1;
                        If Vector[Y] > Vector[Pos] then
                            begin
                                Aux := Vector[Y];
                                Vector[Y] := Vector[Pos];
                                Vector[Pos] := Aux;
                            end;
                    end;
            end;
    End;
```

- *Inserción*: Consiste en insertar los elementos del vector en la posición correcta, entre los elementos que ya están ordenados.

  Se comienza del segundo elemento, suponiendo que el primero ya está ordenado. Compara ambos elementos, si no están ordenados, los ordena. Luego toma el tercero y lo inserta en la posición correcta con respecto a los dos primeros. Así sucesivamente hasta terminar con el último elemento.

  Si un elemento está ubicado correctamente no lo mueve de lugar.

```
Procedure Insercion (Var Vector: Array of Integer; NroEle: Integer);
Var
   X, Y: Integer;
   Aux: Integer;

Begin
   For X := 2 to NroEle do
      begin
         Aux := Vector[X];
         Y := X - 1;
         While (Y > 0) and (Vector[Y] > Aux) do
            begin
               Vector[Y+1] := Vector[Y];
               Y := Y - 1;
            end;

         Vector[Y+1] := Aux;
      end;
End;
```

- *Ordenación Eficiente*: Son algoritmos recursivos.
  - *Ordenación por Mezcla (Sorting by Merging)*: Consiste en dividir varias veces al vector en partes iguales y ordenar cada una de ellas, para luego mezclarlas entre sí y obtener de esta manera, la estructura ordenada.

    Es un algoritmo recursivo que trabaja dividiendo al vector por la mitad, en forma iterativa, hasta que cada subvector contenga un único elemento: en la primer invocación del procedimiento se pasa al vector entero, junto con los limites del mismo; en la segunda invocación, se pasan la primera mitad (n/2 elementos) y los limites de esta; en la tercera invocación, se pasan la primera mitad de la primera mitad del vector original (n/4 elementos) y los limites... así sucesivamente. Finalizadas las divisiones, es la mezcla de los subvectores (*merge*) la que ordena a la estructura.

```
Type
   TVector = Array [1..N] of Integer;
```

```
Procedure Mezclar (Var Vector: TVector; Inicio, Medio, Fin: Integer);
Var
   VectorAux: TVector;
   Inicio2, X: Integer;

Begin
   X := 0;
   Inicio2 := Medio + 1;
   While (Inicio <= Medio) and (Inicio2 <= Fin) do
      begin
         X := X + 1;
         {Se copia el elemento más chico de los dos subvectores a}
         {VectorAux[X]}
         If Vector[Inicio] < Vector[Incio2] then
            begin
               VectorAux[X] := Vector[Inicio];
               Inicio := Inicio + 1;
            end
         else
            begin
               VectorAux[X] := Vector[Inicio2];
               Inicio2 := Inicio2 + 1;
            end;
      end;

   While Inicio <= Medio do {Mientras haya elementos en el 1° subvector}
      begin
         X := X + 1;
         VectorAux[X] := Vector[Inicio];
         Inicio := Inicio + 1;
      end;


   While Inicio2 <= Fin do {Mientras haya elementos en el 2° subvector}
      begin
         X := X + 1;
         VectorAux[X] := Vector[Inicio2];
         Inicio2 := Inicio2 + 1;
      end;


   Vector := VectorAux;
End;


Procedure OrdenarPorMezcla (Var Vector: TVector; Inicio, Fin:
                            Integer);
Var
   Media: Integer;

Begin
   If Inicio < Fin then
      begin
         Medio := (Inicio + Fin) div 2;
         OrdenarPorMezcla (Vector, Inicio, Medio);
         OrdenarPorMezcla (Vector, Medio + 1, Fin);
         Mezclar (Vector, Inicio, Medio, Fin);
      end;
End;
```

Este método de ordenación es más eficiente que aquellos que no son recursivos, pero posee un *costo adicional* relacionado la estructura auxiliar: si `VectorAux` es muy grande, las sucesivas invocaciones de `Ordenar` consumirían mucha memoria. Por lo tanto, el uso de este método está recomenda-

do para vectores de poco tamaño.

- *Ordenación Rápida (QuickSort)*: Consiste en dividir varias veces al vector en partes iguales (en el caso promedio), en donde los elementos de la primera mitad son menores que los de la segunda, y que al ordenar cada una de ellas, se obtiene la estructura ordenada.

  Es un algoritmo recursivo que trabaja de manera similar que la *Ordenación por Mezcla*, pero al no utilizar una estructura auxiliar, resulta ser más eficiente ya que realiza menos intercambios. Consta de dos etapas:

  División del vector por la "mitad": Se toma al último elemento (el que está más a la derecha) como ***punto de división***, el cual se "saca" de la estructura dejando un ***hueco***. Luego se busca, comenzando desde la izquierda (`Izq`), un elemento mayor que el punto de división. Si existe, se lo transfiere a dicho hueco. Después, comenzando desde la derecha (`Der`), se busca un elemento menor que el punto de división, que, si existe, será colocado en el hueco dejado por el elemento transferido de la izquierda.

  De esta manera, se repite sucesivamente la búsqueda y reubicación de elementos mayores y menores que el punto de división, hasta que las marcas `Izq` y `Der` se encuentren en el mismo lugar (en el cual se ubicará el punto de división).

  Ordenación de los subvectores: Se invoca tanto para la primera mitad como para la segunda el procedimiento de división, anteriormente descripto.

```
Procedure Dividir (Var Vector: Array of Integer; Var Medio:
                   Integer; Izq, Der: Integer);

Var
   Aux: Integer;

Begin
   Aux := Vector[Ultimo];

   While Izq < Der do
      begin
         {Se mueve la marca Izq hacia la derecha}
         While (Izq < Der) and (Vector[Izq] <= Aux) do
            Izq := Izq + 1;

         If Izq < Der then
            begin
               Vector[Der] := Vector[Izq];
               Der := Der - 1;
            end;

         {Se mueve la marca Der hacia la izquierda}
         While (Izq < Der) and (Vector[Der] >= Aux) do
            Der := Der - 1;
         If Izq < Der then
            begin
               Vector[Izq] := Vector[Der];
               Izq := Izq + 1;
            end;
      end;
   Medio := Der; {Medio := Izq;}
```

```
    Vector[Medio] := Aux;
End;


Procedure OrdenacionRapida (Var Vector: Array of Integer;
                            Inicio, Fin: Integer);
Var
   Medio: Integer;

Begin
   If Inicio < Fin then
      begin
         Dividir (Vector, Medio, Inicio, Fin);
         OrdenacionRapida (Vector, Inicio, Medio-1);
         OrdenacionRapida (Vector, Medio, Fin);
      end;
End;
```

CAPÍTULO IX: DATOS COMPUESTOS ENLAZADOS: LISTAS Y ÁRBOLES

1) Punteros

Es una variable numérica que almacena la dirección en memoria de un dato.

2) Listas

Es una colección dinámica de elementos homogéneos, donde los mismos pueden aparecer físicamente dispersos en memoria, pero manteniendo un orden lógico interno. Son estructuras lineales, por lo tanto, cada elemento de la misma, menos el primero y el último, posee un antecesor y un sucesor.

Existen 4 tipos básicos de listas:

- *Simples*: Son aquellas que poseen un único enlace, debido a que sus nodos solo tienen un puntero que los vincula. Poseen un principio y fin, pero siempre se accede a los diversos elementos contenidos a través del puntero principal y en un solo sentido (Inicio-Fin).

- *Circulares*: Son parecidas a las listas simples, pero difieren en que al ser circulares no poseen un fin, porque el último elemento apunta siempre al primero o principio de la lista. Poseen un nodo especial que indica cual es el inicio de las mismas, pero se pueden acceder a cualquier elemento desde cualquier nodo.

- *Doblemente Enlazadas*: Son aquellas que poseen dos enlaces (debido a que sus nodos contienen dos punteros que los vincula) y pueden ser recorridas en dos sentidos (Inicio-Fin, Fin-Inicio) ya que hay dos punteros principales. Poseen un principio y un fin (en donde los datos pueden ser accedidos desde cualquiera de los dos) y ocupan más memoria que las listas simples.

- *Multienlaces*: Son aquellas que poseen dos o más enlaces, debido a que sus nodos contienen varios punteros que los vinculan. Estos enlaces son índices por los cuales se puede acceder a los nodos en distintos órdenes. Poseen varios principios y/o fines.

La ***creación*** de una lista simple es a través de la unión de ***nodos*** (los elementos) que se dividen en dos partes:

- *Dato*: Es la información que se maneja y puede estar constituida por un conjunto de elementos de algún tipo previamente definido.

- *Identificador*: Es una variable puntero cuyo contenido es la dirección del próximo elemento de la estructura.

El nodo de una lista simple, en Pascal, esta formado de la siguiente manera:

```
Type
   Lista = ^Nodo;
   Nodo = Record
             Numero: Integer;
             Siguiente: Lista;
          End;
```

Estos nodos son punteros de registros, por lo tanto, deben ser creados antes de ser utilizados. En sí, una lista es en realidad un puntero que recibe el nombre de ***puntero inicial o principal***, y contiene la dirección de inicio de la misma.

Las ***operaciones*** más comunes que se realizan sobre listas simples son:

- *Inicialización*: Se asigna el valor `Nil` al puntero principal.

- *Inserción*: Se busca la posición correcta en la que irá el nuevo elemento, que será creado antes de insertarlo. Luego se enlaza los punteros de los elementos adyacentes a este.

  Si se inserta al inicio de la lista, el puntero principal apuntará al nuevo y este al que ahora es el segundo. Si se inserta al final, el puntero del último elemento apuntará al nuevo, convirtiéndose este en el último, con un identificador al siguiente en `Nil`. Y si se inserta entre los extremos de la lista, el nuevo puntero apuntará al elemento `Actual` y el identificador al siguiente de `Anterior`, apuntará al nuevo.

```
Procedure Insertar (Var ListaNumeros: Lista; K: Integer);
Var
   Nuevo, Actual, Anterior: Lista;

Begin
   Actual := ListaNumeros;
   While (Actual <> Nil) and (K > Actual^.Numero) do
      begin
         Anterior := Actual;
         Actual := Actual^.Siguiente;
      end;


   New (Nuevo);
   If Actual = ListaNumeros then
      ListaNumeros := Nuevo
   else
      Anterior^.Siguiente := Nuevo;
   Nuevo^.Numero := K;
   Nuevo^.Siguiente := Actual;
End;
```

- *Borrado*: Se busca la posición correcta del elemento que se borrará. Se enlazan los punteros adyacentes a este y el elemento se borra haciendo un *dispose* del mismo.

  Si se borra el elemento del inicio de la lista, el puntero principal apuntará al siguiente elemento del primero. Si se borra el del final, el identificador al siguiente del penúltimo elemento será nulo (`Nil`), convirtiéndose este en el último. Y si se borra un elemento entre los extremos de la lista, el identificador al siguiente de `Anterior` a puntará al siguiente del elemento `Actual`.

```
Procedure Borrar (Var ListaNumeros: Lista; K: Integer);
Var
   Actual, Anterior: Lista;

Begin
   Actual := ListaNumeros;
   While (Actual <> Nil) and (K <> Actual^.Numero) do
      begin
         Anterior := Actual;
         Actual := Actual^.Siguiente;
      end;

   If Actual^.Numero = K then
      begin
         If Actual = ListaNumeros then
            ListaNumeros := Actual^.Siguiente;
```

```
            else
               Anterior^.Siguiente := Actual^.Siguiente;
         Dipose (Actual);
      end;
   End;
```

- *Recorrido*: Se comienza leyendo el puntero inicial, siempre y cuando no este vacía la estructura, el cual contiene el identificador del segundo dato. Se lee este que contiene el identificador del tercero y este, a su vez, el del cuarto. Así sucesivamente hasta leer un nodo que posea un identificador al siguiente elemento ***nulo*** (`Nil`), el cual es el último.

```
While (Actual <> Nil) do
   Actual := Actual^.Siguiente;
```

- *Búsqueda*: Se recorre la lista hasta que se encuentre el elemento deseado o hasta que se llegue al final de la misma.

```
While (Actual <> Nil) and (Actual^.Numero <> N) do
   begin
      Anterior := Actual;
      Actual := Actual^.Siguiente;
   end;
```

Las ***ventajas*** del uso de listas recaen en que son más fáciles de manipular que otras estructuras estáticas, ya que las operaciones de borrado y de inserción son más simples de implementar, porque no realizan un corrimiento masivo de elementos, como hacen los vectores. Además, al ser una estructura dinámica solo ocupan la memoria necesaria.

La ***desventaja*** que conllevan las listas es el acceso secuencial a los elementos de las mismas, ya que no se encuentran físicamente consecutivos.

3) Árboles

Es una estructura, dinámica, homogénea y recursivas, que satisface tres propiedades:

- Cada elemento del árbol se puede relacionar con cero o más elementos, los cuales se llaman ***hijos***.
- Si el árbol no está vacío, existe un único elemento que no posee un padre y recibe el nombre de ***raíz***.
- Todo elemento posee un padre, excluyendo a la raíz, y es descendiente de esta.

Además, los árboles poseen 6 características más:

- Cada nodo que no posea descendientes, recibe el nombre de ***hoja*** y se encuentra siempre al final del árbol.
- Cada secuencia lineal de elementos, donde cada uno es el padre del próximo ítem de la secuencia, recibe el nombre de ***camino***.
- Cada camino que se extienda desde la raíz hasta una hoja, recibe el nombre de ***rama***.
- Cada nodo junto con sus descendientes, recibe el nombre de ***subárbol***.
- Cada nodo posee un ***nivel*** que es el número de ítems que hay en el único camino que lo une a la raíz.

- La *altura* de un árbol es el máximo entre los niveles de los nodos de este.

ÁBOLES BINARIOS

Son aquellos árboles donde cada nodo no tiene más de dos hijos. A lo sumo, poseen un hijo izquierdo y un hijo derecho.

Son también conocidos como *Árboles Binarios de Orden*, dado que cada nodo de este es mayor que los elementos de su subárbol izquierdo (el que tiene como raíz a su hijo izquierdo) y menor o igual que los de su subárbol derecho (el que tiene como raíz a su hijo derecho).

La *creación* de un árbol binario es a través de la unión de *nodos* (los elementos) que se dividen en tres partes:

- *Dato*: Es la información que se maneja y puede estar constituida por un conjunto de elementos de algún tipo previamente definido.
- *Identificador Izquierdo*: Es una variable puntero cuyo contenido es la dirección del próximo elemento izquierdo de la estructura.
- *Identificador Derecho*: Es una variable puntero cuyo contenido es la dirección del próximo elemento derecho de la estructura.

El nodo de un árbol, en Pascal, esta formado de la siguiente manera:

```
Type
   Arbol = ^Hoja;
   Hoja = Record
             Numero: Integer;
             HijoIzq, HijoDer: Arbol;
          End;
```

Estos nodos son punteros de registros, por lo tanto, deben ser creados antes de ser utilizados. En sí, un árbol es en realidad un puntero (la raíz) que contiene la dirección de inicio del mismo.

Las *operaciones* más comunes que se realizan sobre árboles binarios son:

- *Inicialización*: Se asigna el valor `Nil` a la raíz.
- *Inserción*: Cada elemento nuevo se incorpora al árbol como una nueva hoja, preservando solo el orden del mismo sin importar su altura. Se busca primero el lugar en donde se agregará la nueva hoja y, luego se acomodan los punteros.

```
Procedure Insertar (Var Ar: Arbol; N: Integer);
Begin
   If (Ar = Nil) then
      begin
         New (Ar);
         Ar^.Nro := N;
         Ar^.HijoIzq := Nil;
         Ar^.HijoDer := Nil;
      end
   else
      IF (N < Ar^.Numero) then
         Insertar (Ar^.HijoIzq, N)
      else
```

```
            Insertar (Ar^.HijoDer, N);
    End;
```

- *Recorrido*: Existen tres formas de recorrer un árbol binario:
  - *En Orden*: Se recorre el árbol de la forma Hijo-Padre-Hijo.

```
Procedure RecorrerArbolInOrden (Ar: Arbol);
Begin
    If (Ar <> Nil) then
       begin
          ImprimirArbolInOrden (Ar^.HijoIzq);
          Write (Ar^.Numero); {Padre}
          ImprimirArbolInOrden (Ar^.HijoDer);
       end;
End;
```

  - *Pre-Orden*: Se recorre el árbol de la forma Padre-Hijo-Hijo.

```
Procedure RecorrerArbolPreOrden (Ar: Arbol);
Begin
    If (Ar <> Nil) then
       begin
          Write (Ar^.Numero); {Padre}
          ImprimirArbolInOrden (Ar^.HijoIzq);
          ImprimirArbolInOrden (Ar^.HijoDer);
       end;
End;
```

  - *Pos-Orden*: Se recorre el árbol de la forma Hijo-Hijo-Padre.

```
Procedure RecorrerArbolPosOrden (Ar: Arbol);
Begin
    If (Ar <> Nil) then
       begin
          ImprimirArbolInOrden (Ar^.HijoIzq);
          ImprimirArbolInOrden (Ar^.HijoDer);
          Write (Ar^.Numero); {Padre}
       end;
End;
```

- *Búsqueda*: Se recorre el árbol de manera pre-orden hasta encontrar el elemento deseado o hasta que se llegue a final de la rama.

Es una variante de la búsqueda dicotómica: si el elemento apuntado es el deseado, la búsqueda termina. Sino, se comprueba que el elemento buscado sea menor que el apuntado, y de ser así, se vuelve a invocar al proceso pasando como parámetro a `HijoIzq` como árbol. En caso de que sea mayor, se invoca a dicho proceso pasando a `HijoDer`.

```
Function Pertenencia (Ar: Arbol; N: Integer): Boolean;
Begin
    If (Ar = Nil) then
       Pertenencia := False
    else
       If (Ar^.Numero = N) then
          Pertenencia := True
```

```
        else
           If (N > Ar^.Numero) then
              Pertenencia := Pertenencia (Ar^.HijoDer, N)
           else
              Pertenencia := Pertenencia (Ar^.HijoIzq, N);
End;
```

- *Altura del árbol*: Es el valor máximo de altura que posee el árbol (máximo nivel + 1) y se calcula contando al nodo actual más el máximo de hijos entre `HijoIzq` e `HijoDer`.

```
Function Mayor (A, B: Integer): Integer;
Begin
   If (A >= B) then
      Mayor := A
   else
      Mayor := B;
End;

Function Altura (Ar: Arbol): Integer;
Begin
   If (Ar = Nil) then
      Altura := 0
   else
      Altura := 1 + Mayor(Altura (Ar^.HijoIzq), Altura (Ar^.HijoDer))
End;
```

- *Cantidad de Nodos*: Es el número de nodos que posee el árbol y se calcula contando al nodo actual más los descendientes de sus hijos.

```
Function Numero_Nodos (Ar: Arbol): Integer;
Begin
   If (Ar = Nil) then
      Numero_Nodos := 0
   else
      Numero_Nodos := 1 + (Numero_Nodos (Ar^.HijoIzq) +
                           Numero_Nodos (Ar^.HijoDer));
End;
```

- *Cantidad entre valores*: Es el número de nodos que posee el árbol que cumplen con una condición dada de valores:
  - *Iguales a un valor*:

```
Function Cantidad_N (Ar: Arbol; N: Integer): Integer;
Begin
   If (Ar = Nil) then
      Cantidad_N := 0
   else
      IF (Ar^.Numero = N) then
         Cantidad_N := 1 + Cantidad_N (Ar^.HijoDer, N))
      else
         If (N > Ar^.Numero) then
            Cantidad_N := Cantidad_N (Ar^.HijoDer, N)
         else
            Cantidad_N := Cantidad_N (Ar^.HijoIzq, N);
End;
```

- *Mayores que un valor*:

```
Function Cantidad_Mayor_N (Ar: Arbol; Max: Integer): Integer;
Begin
   If (Ar = Nil) then
      Cantidad_Mayor_N := 0
   else
      If (Ar^.Numero > Max) then
         Cantidad_Mayor_N := 1 + (Cantidad_Mayor_N (Ar^.HijoDer, Max) +
                                  Cantidad_Mayor_N (Ar^.HijoIzq, Max))
      else
         Cantidad_Mayor_N := Cantidad_Mayor_N (Ar^.HijoDer, Max);
End;
```

- *Menores que un valor*:

```
Function Cantidad_Menor_N (Ar: Arbol; Min: Integer): Integer;
Begin
   If (Ar = Nil) then
      Cantidad_Menor_N := 0
   else
      If (Ar^.Numero < Min) then
         Cantidad_Menor_N := 1 + (Cantidad_Menor_N (Ar^.HijoIzq, Min) +
                                  Cantidad_Menor_N (Ar^.HijoDer, Min))
      else
         Cantidad_Menor_N := Cantidad_Menor_N (Ar^.HijoIzq, Min);
End;
```

- *Entre valores*:

```
Function Cantidad_Entre (Ar: Arbol; Max, Min: Integer): Integer;
Begin
   If (Ar = Nil) then
      Cantidad_Entre := 0
   else
      If (Ar^.Numero >= Min) and (Ar^.Numero <= Max) then
         Cantidad_Entre := 1 + (Cantidad_Entre (Ar^.HijoIzq, Max, Min) +
                                Cantidad_Entre (Ar^.HijoDer, Max, Min))
      else
         If (Ar^.Numero > Max) then
            Cantidad_Entre := Cantidad_Entre (Ar^.HijoIzq, Max, Min)
         else
            Cantidad_Entre := Cantidad_Entre (Ar^.HijoDer, Max, Min);
End;
```

CAPÍTULO XII: INTRODUCCIÓN AL CONCEPO DE ARCHIVOS

ARCHIVO

Es una estructura, dinámica y homogénea, de datos guardados (o colección de *registros*) en un dispositivo de almacenamiento secundario.

Los archivos se dividen en dos categorías:

- *Físicos*: Son aquellos que están almacenados en la memoria secundaria (discos rígidos, flexibles, etc).

  Son los archivos creados por el sistema operativo. Por lo tanto, es este el encargado de realizar todas aquellas operaciones relacionadas con la creación y ubicación de los datos en el disco, cantidad de espacio a utilizar, tipos de accesos permitidos, etc.

- *Lógicos*: Son aquellos que se utilizan para referenciar a los archivos físicos.

  En realidad son *nombres lógicos* a través de los cuales se interactuarán con los verdaderos archivos almacenados en disco. Los programas solo envían y reciben información desde y hacia los archivos físicos.

La *declaración* de un archivo, en Pascal, es de la forma:

```
Type
    NombreTipo = File of TipoBase;    {Tipados}
    NombreTipo2 = File;               {Sin Tipo}

Var
    NombreVariable: File of TipoBase;
    NombreVariable2: File;
```

Donde `File` es el indicador de archivo.

Los archivos pueden ser:

- *Tipados*: Son aquellos que poseen un tipo (integer, real, *string*, char, registros).

  Colocando `of` más el tipo, se establece a que tipo base pertenecerán los registros del archivo, lo cual hace que sean más fáciles de usar, ya que no se necesita realizar ninguna conversión sobre ellos.

- *Sin Tipo*: Son aquellos que no poseen un tipo.

  No se establece a que tipo pertenecerán los registros del archivo, por lo tanto, es el programador el encargado de hacer la conversiones necesarias para manipularlos.

Las *operaciones* básicas sobre archivos son:

- *Vinculación*: Consiste en relacionar al archivo lógico con el físico.

  ```
  Assign (NombreLogico, NombreFisico);
  ```

- *Apertura*: Realiza la apertura de los archivos indicando el tipo de operación que se va a realizar

```
Reset   (NombreLogico); {Abre el archivo existente para lectura y escritura}
ReWrite (NombreLogico); {Abre el archivo por primera vez, solo para escritura.}
```

Donde ReWrite crea el archivo o lo sobreescribe borrando todo el contenido, si existiera. En el caso de Reset, debe existir el archivo para poder abrirlo.

- *Clausura*: Cierra el archivo, colocando en fin del mismo al final, y vacia el *buffer*, para guardar los últimos registros ingresados.

```
Close (NombreLogico);
```

- *Lectura*: Lee un registro del archivo, siempre y cuando no este vacío, y lo almacena en una variable del mismo tipo, avanzando el puntero del mismo al siguiente registro.

```
Read  (NombreLogico, Variable);
```

- Escritura: Escribe un registro en el archivo avanzando el puntero del mismo al siguiente registro.

```
Write (NombreLogico, Variable);
```

, los archivos, visto desde un programa, son punteros que apuntan siempre a un registro contenido en ellos.

FER

a memoria intermedia, entre un archivo y un programa, donde los datos residen provisoriamente hasta ser al- nados definitivamente en memoria secundaria o donde los datos residen una vez recuperados. Su propósito es que el sistema operativo realice varios accesos al disco.

SO A UN ARCHIVO

cesos a los registros pueden ser:

Secuencial: Permite acceder a los registros uno tras otro y en el mismo orden físico en el que están guarda- dos.

Directo: Permite obtener un registro determinado sin necesidad de haber accedido a los predecesores.

CAS DE ORGANIZACIÓN

rdo a su organización, un archivo se clasifica en:

Secuencial: Consiste en un conjunto de registros almacenados consecutivamente de manera que para acceder l registro n-ésimo se debe acceder a los $n-1$ registros anteriores.

Directo: Consiste en un conjunto de registros donde el ordenamiento físico no corresponde con el orden ló- ico. Los registros se recuperan accediendo a su posición dentro del archivo, sin tener que acceder a los $n-1$ egistros anteriores.

ecuencial Indizado o Indexado: Utiliza estructuras de datos auxiliares para permitir un acceso pseudo di- cto a los registros de un archivo. Esta técnica posee un acceso más rápido que el secuencial, pero necesita ás espacio para mantener las estructuras.