Archivos Secuenciales

1 INTRODUCCIÓN

1.1 CONCEPTOS FUNDAMENTALES

- **Dato**
  Representación de una cosa real o ideal o de un evento ocurrido o programado para que ocurra, en una unidad lógica[1] de manipulación llamada ***registro***, en términos de características descriptivas (también llamadas atributos) que se representan en unidades componentes del registro llamadas ***campos***.
- **Archivo de Datos**
  Unidad lógica de almacenamiento persistente de registros de datos, administrada por un Sistema Operativo (sólo un Sistema Operativo, mediante su *Sistema de Archivo,* permite que usuarios y programas creen o eliminen archivos, y que programas accedan y actualicen la información contenida en ellos). Dentro de un archivo los registros pueden organizarse en otras unidades lógicas llamadas ***bloques* o *páginas***, ya sea para agruparlos con algún criterio de afinidad o para almacenarlos con longitud variable.
  En archivos con registros de longitud fija no organizados en bloque, la unidad de transferencia que se lee del archivo o se escribe en él es el registro, mientras que en archivos con registros organizados en bloques o páginas, la unidad de transferencia es el bloque o página.

1.2 CLASIFICACIÓN DE ARCHIVOS

Los archivos se pueden clasificar según la clase de información que contengan. En general pueden ser

- **Maestros**
  Registros de datos de un sistema de información que representan entidades de existencia real o ideal, por ejemplo productos o servicios, o valores de referencia para determinar características o atributos de otros datos (dominios de atributos definidos por extensión).
- **Transaccionales**
  Registros de hechos o eventos de un sistema de información relacionados con datos maestros, por ejemplo de ventas de productos o de prestaciones de servicios.
- **De Reporte**
  Información editada para su presentación al usuario (en general en formatos pdf, html o de texto).
- **De Trabajo**
  Resultados parciales o intermedios de procesamiento, o datos de intercambio entre programas.
- **De Control de Datos**
  Para almacenar metadatos (definiciones de datos), administrar espacios libres o acceder al contenido de otro archivo (índices y tablas de acceso).
- **De Intercambio de Datos**
  Pueden ser para representar datos en formatos estándar de manera que puedan ser procesados libremente conociendo el estándar. Generalmente son archivos de texto, con alguna convención para rotular o delimitar datos, que pueden incluir o no definiciones sobre la estructura de la información contenida (un estándar actual es el XML: eXtensible Markup Language).

[1] Propia de un programa o aplicación.

También pueden ser archivos de comunicación entre procesos llamados *sockets*.

- **De Recursos de Programa o Unidades Grandes de Información**
  Imágenes (jpg, bmp, …), audio (mp3, …), vídeo (mpg, …).
- **Dependientes de Programas**
  Archivos con tipo asociado a un programa o aplicación (doc, odt, xls, ppt, ...).
- **De Empaquetado de Archivos**
  Para agrupar, normalmente en forma comprimida, archivos y directorios, con propósitos de trasmisión o respaldo (zip, rar, tar, …).

En esta asignatura los archivos fundamentales son los maestros y los transaccionales, que se implementan en Pascal como archivos binarios, con registros de longitud fija de tipo definido, o con bloques de agrupamiento de registros de longitud fija o variable. En los programas que manipulan archivos maestros o transaccionales también se usan los archivos de texto, ya sea como fuente de datos de carga o como archivos de reporte para visualizar el contenido de archivos binarios. También se considera a los archivos de control, como soporte de estructuras de acceso a archivos de datos (índices y tablas de acceso).

1.3 Archivos en Pascal

Las variables de tipos básicos en todo lenguaje de programación representan a la dirección de una sección de tamaño fijo de la memoria principal donde se almacena un valor del tipo correspondiente. Pero los archivos tienen extensiones variables en el tiempo, y sus extensiones pueden exceder la capacidad de almacenamiento de la RAM, por lo que no se puede reservar espacio para cargar archivos completos. Por eso, las variables de archivos no representan al contenido de los mismos sino a una posición relativa del programa dentro del archivo donde éste se encuentre almacenado; además, toda operación sobre archivos debe realizarse a través de instrucciones del lenguaje que implican llamadas al Sistema Operativo *(system calls)*, sea para asociar una variable archivo con un archivo real (asignación), sea para crear un archivo, para abrir uno existente o para cerrarlo, sea para establecer una posición relativa del programa dentro del archivo (posicionamiento), o sea para transferir datos del archivo a la RAM o viceversa (lectura o escritura de registros o bloques).

Entonces, para manipular archivos no sólo se requieren variables para los archivos, sino también para manipular sus registros o bloques en RAM, ya que lo que se transfiere de un archivo a la RAM o en sentido inverso, son registros o bloques (unidades que componen el archivo). Los tipos para variables de registros y de archivos y las operaciones sobre archivos (búsqueda de registros, altas, bajas y modificaciones) se suelen definir en unidades (*units*), como tipos de datos abstractos. Los parámetros para archivos sólo pueden ser por referencia y con tipo definido por el programador, y es conveniente que sean registros que agrupen todas las variables necesarias para la manipulación de un archivo (registro de control o *handle* del archivo).

1.3.1 Tipos de archivo

- **De registros de longitud fija (File of <tipo de registro>):** la variable de lectura/escritura debe ser del mismo tipo de los registros del archivo → lectura/escritura de registros, con acceso secuencial o relativo.

```
Type
    tPersona = Record {tipo de registro para personas}
    DNI: Longword; {0..4294967295}
    Apellido: String[20];
    Nombres: String[20];
    FechaNac: Longword {AAAAMMDD}
    end;
```

```
    aPersonas = File of tPersona; {tipo de archivo para registros de personas}
Var
    personas: aPersonas; {variable para archivo de personas}
    p: tPersona; {variable para registros de personas}
```

Por defecto, el tamaño de los registros es la suma de tamaños de sus campos, pero cada campo siempre ocupando un número par de bytes; así, Apellido y Nombres ocupan 22 bytes cada uno (un byte para el prefijo de longitud, 20 para representación de valores en ASCII, y uno extra de relleno para completar un número par de bytes). El tamaño real de un registro tPersona es entonces de 4+22+22+4=52 bytes. Para evitar los bytes de relleno (con los campos alineados a byte), se puede definir al registro

```
    tPersona = Packed Record {tipo de registro "empaquetado" para personas}
```

De esta manera cada registro ocuparía 50 bytes tanto en memoria como en el archivo.

- **De bloques de bytes (File):** cuando se abre el archivo se indica el tamaño en bytes de los bloques de transferencia, y se puede transferir uno o varios bloques por operación, desde o hacia variables con cualquier estructura → lectura/escritura en bloques, con acceso secuencial o relativo. Estos archivos se utilizan para almacenar registros de longitud variable.

```
Const
    LongBloque = 1024;
Type
    tBloque = Array[1..LongBloque] of Byte;
    abPersonas = File; {tipo de archivo para empaquetar registros de longitud variable de personas en bloques}
Var
    bpersonas: abPersonas; {variable para archivo de bloques de personas}
    b: tBloque; {variable para bloques de personas}
```

- **De Texto (Text):** de caracteres estructurados en líneas → lectura/escritura con conversión automática de tipos (se puede escribir tipos numéricos a archivos de texto, o leer con variables de tipo numérico desde archivos de texto), con acceso exclusivamente secuencial. Estos archivos son muy útiles para cargar archivos de otros tipos con datos editados y validados, y para exportar archivos de otros tipos a un formato accesible por cualquier editor de textos.
  A los caracteres del contenido del archivo se agregan automáticamente otros caracteres de control para separar líneas (<LF> y <CR>: *Line Feed* –salto de línea- y *Carriage Return* –retono de carro), para representar varios blancos (<Tab>) y para representar el final del archivo (<EoF>).

```
Var
    cargaPersonas: Text; {archivo de texto para cargar datos de personas al archivo personas o bpersonas}
    listaPersonas: Text; {archivo de texto para reportar datos de personas}
```

Para escribir en un archivo de texto registros de tipo tPersona, se debe escribir campo por campo.

1.3.2 Operaciones con Archivos y Directorios en la unidad System (disponibles en todo programa)

- **Asignación:** una variable de tipo archivo representa a cualquier archivo del tipo definido → se le debe asociar un archivo real (existente en el disco) o el nombre de uno a crear, para poder usarla.

  procedure Assign(var f: file; const Name: String);

```
Assign(bpersonas, 'personas.b');
```

  procedure Assign(var f: file; p: PChar);
  procedure Assign(var f: file; c: Char);
  procedure Assign(var f: TypedFile; const Name: String);

```
Assign(personas, 'personas.r');
```

  procedure Assign(var f: TypedFile; p: PChar);
  procedure Assign(var f: TypedFile; c: Char);
  procedure Assign(var t: Text; const s: String);

```
Assign(cargaPersonas, 'personas.txt');
Assign(listaPersonas, 'listado.txt');
```

  procedure Assign(var t: Text; p: PChar);
  procedure Assign(var t: Text; c: Char);

- **Apertura**: establece la dirección de transferencia entre el programa y el archivo y la posición inicial del programa en el archivo. Cuando se abre un archivo, Pascal le asocia un buffer interno para alojar una sección del archivo durante las operaciones de lectura y escritura; de este modo, no todas las operaciones de lectura o escritura implican una llamada al Sistema Operativo, ya que un registro a leer puede encontrarse en el buffer interno, y cuando se escribe un registro se lo guarda en el mismo, difiriendo la escritura real en el disco por intermedio del SO hasta que se requiera reutilizar el buffer para otra sección del archivo. *Si no se cierra un archivo luego de haber escrito en él, los registros escritos en el buffer interno se pierden.*
  - **Rewrite**: crea un archivo para escribir, con el nombre asignado a la variable, y si ya existe borra su contenido, posicionando al programa al comienzo del archivo.

    procedure Rewrite(var f: file; l: LongInt); *{l longitud de bloques de bytes}*

```
Rewrite(bpersonas, LongBloque);
```

    procedure Rewrite(var f: file); *{por defecto, la longitud de bloques de bytes es 128}*
    procedure Rewrite(var f: TypedFile);
    procedure Rewrite(var t: Text);

```
Rewrite(listaPersonas);
```

  - **Reset**: abre el archivo tanto para leer como para escribir, con el nombre asignado a la variable, posicionando al programa al comienzo del archivo (el archivo debe existir).

procedure Reset(var f: file; l: LongInt); *{l longitud de bloques de bytes}*

```
Reset(bpersonas, LongBloque);
```

procedure Reset(var f: file); *{por defecto, la longitud de bloques de bytes es 128}*
procedure Reset(var f: TypedFile);

```
Reset(personas);
```

procedure Reset(var t: Text);

```
Reset(cargaPersonas);
```

- **Append**: abre un archivo de texto para escribir con el nombre asignado a la variable, posicionando al programa al final del archivo.

  procedure Append(var t: Text);

- **Posicionamiento (Seek):** posiciona al programa en el número relativo de unidad que se especifica (no se permite en archivos de texto).

  procedure Seek(var f: file; Pos: Int64);
  Los registros (archivos con tipo definido) o bloques de bytes con tamaño definido en la apertura (archivos sin tipo), se numeran a partir de 0 (cero).

- **Lectura**: copia bytes del archivo a la variable que se proporciona, desde la posición donde está el programa, y posiciona al programa a continuación → Read (archivos de registros y de texto), ReadLn (archivos de texto), BlockRead (archivo de bloques de bytes).

  procedure Read(var F: Text; Args: Arguments); *{los argumentos pueden ser una o más variables de tipo Char, Integer, Real, String }*
  procedure Read(var F: TypedFile; Args: Arguments); *{los argumentos pueden ser una o más variables del tipo especificado en la declaración de F}*

```
Read(personas, p);
```

procedure ReadLn(var F: Text; Args: Arguments); *{los argumentos pueden ser una o más variables de tipo Char, Integer, Real, String.*
Luego de la lectura, el programa se posiciona al comienzo de la siguiente línea del archivo. El fin de línea se marca con una secuencia de caracteres de control (dependiente de la plataforma). La marca de fin de línea no se considera parte de la línea y se ignora.}

Para poder leer datos de personas de un archivo de texto, los atributos numéricos pueden disponerse en una línea, separados por uno o más espacios, pero los de tipo string, deben estar aislados en una línea. Por ejemplo

```
20349584 19850209
```

```
Fernández
Manuel
22349584 19910811
Lagos
María Eugenia
…
```

```
ReadLn(cargaPersonas, p.DNI, p.FechaNac);
ReadLn(cargaPersonas, p.Apellido);
ReadLn(cargaPersonas, p.Nombres);
```

procedure BlockRead(var f: file; var Buf; count: Int64; var Result: Int64);
procedure BlockRead(var f: file; var Buf; count: LongInt; var Result: LongInt);
procedure BlockRead(var f: file; var Buf; count: Cardinal; var Result: Cardinal);
procedure BlockRead(var f: file; var Buf; count: Word; var Result: Word);
procedure BlockRead(var f: file; var Buf; count: Word; var Result: Integer);
procedure BlockRead(var f: file; var Buf; count: Int64);
BlockRead lee count o menos registros del archivo f. Un registro es un bloque de bytes con tamaño especificado en los procedimientos Rewrite o Reset. El resultado se ubica en Buffer, que debe contener suficiente espacio para Count registros. No lee registros parciales. Si se especifica Result, devuelve el número de registros efectivamente leídos. Si no se especifica y se leyeron menos de Count registros, se produce un error en tiempo de ejecución. Esto puede controlarse con la directiva al compilador {$I-} y la función IOResult.

```
BlockRead(bPersonas, b, 1);
```

- **Escritura**: copia bytes de la variable que se proporciona al archivo, a partir de la posición donde está el programa, y posiciona al programa a continuación → Write (archivos de registros y de texto), WriteLn (archivos de texto), BlockWrite (archivo de bloques de bytes).

  procedure Write(var F: Text; Args: Arguments); *{los argumentos pueden ser una o más variables de tipo Char, Integer, Real, String, y pueden incluir como parámetro la cantidad de caracteres a imprimir con el valor justificado según el tipo de la variable: Write(F, n:10) escribe al entero n justificado a derecha en 10 caracteres, con blancos al comienzo; Write(F, r:10:2) escribe al real r justificado a derecha en 10 caracteres, con dos dígitos decimales –los parámetros de edición sirven para encolumnar valores}*
  procedure Write(var F: TypedFile; Args: Arguments); *{los argumentos pueden ser una o más variables del tipo especificado en la declaración de F}*

```
Write (personas, p);
```

procedure WriteLn(var F: Text; Args: Arguments); *{ídem Write, pero agrega al final la marca de fin de línea}*

```
WriteLn(listadoPersonas, p.DNI:8, ' ', p.Apellido:21, p.Nombres:20, p.FechaNac:9);
```

```
20349584 Fernández          Manuel              19850209
22349584 Lagos              María Eugenia       19910811
```

```
...
```

```
procedure BlockWrite( var f: file; const Buf; Count: Int64; var Result: Int64 );
procedure BlockWrite( var f: file; const Buf; Count: LongInt; var Result: LongInt );
procedure BlockWrite( var f: file; const Buf; Count: Cardinal; var Result: Cardinal );
procedure BlockWrite( var f: file; const Buf; Count: Word; var Result: Word );
procedure BlockWrite( var f: file; const Buf; Count: Word; var Result: Integer );
procedure BlockWrite( var f: file; const Buf; Count: LongInt );
```

BlockWrite escribe count registros de Buf al archivo f. Un registro es un bloque de bytes con tamaño especificado en los procedimientos Rewrite o Reset. Si los registros no pueden escribirse en el disco, se produce un error en tiempo de ejecución. Esto puede ser controlado con la directiva al compilador {$I-} y la función IOResult.

```
BlockWrite(bPersonas, b, 1);
```

- **Cierre (Close):** desactiva la dirección de transferencia entre el programa y el archivo y el posicionamiento del programa en el archivo, y fuerza la escritura en disco del buffer interno, si éste estuviera actualizado para la escritura en el archivo (si no se cierra un archivo que se estuvo actualizando, los últimos registros actualizados pueden perderse).

  ```
  procedure Close( var f: file );
  procedure Close( var t: Text );
  ```

  Close libera el buffer interno del archivo f forzando su escritura en disco si fue modificado (flush) y lo cierra (esto implica que si el buffer del archivo contiene registros modificados o nuevos, el buffer se escribe en el disco). Luego del cierre de un archivo, no pueden realizarse operaciones sobre él. Para reabrir un archivo cerrado, no es necesario asignarlo nuevamente. Con ejecutar Reset o Rewrite es suficiente.

- **Consultas**: permiten obtener características del archivo o de la posición del programa en el archivo.
  - **FileSize** (tamaño en unidades del archivo –registros o bloques de bytes, para archivos con tipo de registro definido o sin tipo, respectivamente, o bytes para archivos de texto).

    ```
    function FileSize( var f: file ):Int64;
    ```

  - **FilePos** (número relativo de unidad sobre la que está posicionado el programa en el archivo –no se permite en archivos de texto).

    ```
    function FilePos( var f: file ):Int64;
    ```

  - **EoF** (*End of File*: si el programa está posicionado a continuación de la última unidad del archivo).

    ```
    function EOF( var f: file ):Boolean;
    function EOF( var t: Text ):Boolean;
    ```

- **EoLn** (*End of Line*: si el programa está posicionado sobre un carácter de control de fin de línea del archivo -sólo archivos de texto).

  function EoLn(var t: Text):Boolean;

- **IOResult** (resultado de lectura o escritura)

  function IOResult: Word;

DOS errors	I/O errors	Fatal errors
0: Ok 2: No se encontró el archivo 3: No se encontró el camino de acceso (*path*) 4: Demasiados archivos abiertos 5: Acceso denegado 6: Handle inválido 12: Modo de acceso inválido 15: Número de disco inválido 16: No se puede eliminar al directorio actual 17: No se puede renombrar en otro volumen	100: Error de lectura en el disco 101: Error de escritura en el disco 102: Archivo no asignado 103: Archivo no abierto 104: Archivo no abierto para lectura 105: Archivo no abierto para escritura 106: Número inválido	150: Disco protegido de escrituras 151: Dispositivo desconocido 152: Dispositivo no preparado 153: Comando desconocido 154: Chequeo CRC (Contro de Redundancia Cíclica) fallido 155: Se especificó *drive* inválido 156: Error de *seek* en el disco 157: Tipo de medio inválido 158: Sector no encontrado 159: Impresora sin papel 160: Error de escritura en el dispositivo 161: Error de lectura en el dispositivo 162: Falla de hardware

Por defecto, cuando ocurre un error de lectura o escritura el programa termina su ejecución en forma anómala; para controlar estos errores e informar adecuadamente al usuario lo que sucedió (consultando IOResult), se debe usar la directiva al compilador {$I-}.

- **Truncamiento (Truncate):** trunca el archivo a partir de la posición actual.

  procedure Truncate(var F: file);

- **Renombrado de archivo físico (Rename):** cambia el nombre del archivo asignado a la variable.

  procedure Rename(var f: file; const s: String);
  procedure Rename(var f: file; p: PChar);
  procedure Rename(var f: file; c: Char);
  procedure Rename(var t: Text; const s: String);
  procedure Rename(var t: Text; p: PChar);
  procedure Rename(var t: Text; c: Char);
  El archivo debe estar asignado y no abierto.

- **Eliminación de archivo físico (Erase):** elimina el archivo asignado a la variable.

  procedure Erase(var f: file);
  procedure Erase(var t: Text);
  El archivo debe estar asignado y no abierto.

- **Cambio de Directorio:** cambia el directorio actual del programa.

procedure chdir(const s: String);

- **Creación de Directorio:** crea nuevo directorio en el directorio actual.

  procedure mkdir(const s: String);

- **Eliminación de Directorio:** elimina directorio del directorio actual.

  procedure rmdir(const s: String);

1.3.3 Operaciones con Nombres de Archivos y Directorios de la Unidad sysutils (se debe declarar Uses sysutils)

function FileExists(const FileName: String):Boolean; *{determina si el archivoFileName existe en el disco}*
function FileSearch(const Name: String; const DirList: String):String; *{busca al archivo Name en la lista de directorios DirList (separados por comas), y devuelve el nombre del archivo completo (con el path relativo) de la primera ocurrencia que encuentre}*
function GetCurrentDir: String; *{devuelve el directorio actual}*

1.4 Organización de Registros de Datos

1.4.1 Diseño de Registros de datos

- **Identificadores**
  Cada cosa o evento que se represente en un registro como dato, debe poder distinguirse de las demás del mismo archivo: debe haber uno o más atributos que identifiquen unívocamente a cada dato, conformando lo que se llama un ***identificador***. Un conjunto de datos puede tener más de un identificador.
  Siempre que se agregue un registro a un archivo de datos debe verificarse que no exista otro registro con el mismo valor de su o sus identificadores, salvo que se esté efectuando una carga masiva de registros cuya unicidad ya está asegurada.
  También puede haber atributos que relacionan un dato con otro dato, ya sea del mismo archivo o de otro: se denominan ***identificadores externos***. *Un identificador externo debe ser identificador en el archivo al que pertenezca el dato relacionado.*
- **Atributos Compuestos**
  Hay atributos que sólo se pueden describir en términos de otros atributos (por ejemplo un domicilio suele constar de una calle, una ubicación y una localidad que son características propias de una dirección).
  En Pascal pueden definirse como registros anidados.
- **Atributos Opcionales**
  Hay atributos que pueden desconocerse al momento de registrar un dato, o representar una característica que un dato particular puede no poseer.
  La determinación de que un atributo de un registro sea opcional hace que el programa o aplicación que manipula al archivo permita que se agreguen registros sin valor establecido para el atributo. También es necesario determinar una convención para representar valores nulos de modo que cuando se recuperen registros se distinga a los valores nulos de los valores del dominio de esos atributos, cualesquiera sean sus tipos.

- **Atributos Polivalentes**
  Hay atributos que pueden precisarse en términos de una lista de valores del mismo tipo o estructura (por ejemplo una institución puede tener varios números telefónicos, o para una persona se pueden registrar varios domicilios –real, laborales, legal).
  Los registros con algún atributo polivalente suelen almacenarse con longitud variable organizados en bloques.

1.4.2 Problemas de Organización de Registros

- **Representación de valores nulos**
  Como para algunos dominios (tipos) de valores de atributos resulta problemático decidir valores nulos (por ejemplo para un atributo opcional con valores numéricos habría que considerar un valor que no sea válido para ese atributo, lo que implica decidir cuál sería ese valor y definir una constante para representarlo), una solución genérica es tener un ***mapa de bits en cada registro*** para indicar si sus atributos opcionales tienen valor nulo o no.

```
Type
        tReg = Packed Record
                a: LongWord;
                b: String[20];
                c, d: Real; {opcionales}
                d: Char;
                e: Integer; {opcional}
                mapaNulos: Byte {mapa de valores nulos}
                end;
Const
        {Máscaras para consultar o establecer la nulidad de atributos (en valores binarios)}
        cNulo: Byte = %10000000;
        dNulo: Byte = %01000000;
        eNulo: Byte = %00100000;

Function EsNulo(var r:tReg; mascara: Byte): Boolean;
Begin
if (r.mapaNulos and mascara) = mascara
then EsNulo:=True
else EsNulo:=False
end;

Procedure Nulificar(var r:tReg; mascara: Byte);
Begin
r.mapaNulos :=r.mapaNulos or mascara
end;
```

- **Determinación de tamaños de bloque**
  Cuando se quiera organizar los registros de un archivo en bloques o páginas, debe considerarse que el Sistema Operativo implementa una memoria caché para acceder a los archivos, que almacena a los últimos registros físicos de dispositivos de almacenamiento leídos o escritos. Como los registros físicos de los discos tienen 512 bytes, conviene que los bloques tengan tamaños del orden de *512*2^n bytes*, para trabajar en coordinación con el Sistema Operativo.

- **Administración de espacio libre**
  Cuando se elimina un registro de un archivo, el espacio que ocupa dicho registro se puede reutilizar para almacenar un registro nuevo, si los registros son de longitud fija, o liberarlo para aumentar el espacio disponible para registros en el bloque del cual se elimina, si los registros son de longitud variable organizados en bloques.
  En cualquier caso, se requiere alguna estructura de control que permita obtener espacio libre para agregar un nuevo registro *sin necesidad de leer secuencialmente el archivo*, ya que éste sería un costo inadmisible.
  Para archivos con **registros de longitud fija**, las opciones típicas son
  - Mapa de bits desde el primer registro del archivo (registro/s inicial/es de control)

```
Type
    tPersona = Packed Record {tipo de registro para personas}
        Case valido: Boolean of {discriminante de registros para indicar si son válidos (de personas) o no (de mapa o borrados); ocupa un byte}
        True: (     DNI: Longword; {0..4294967295}
                    Apellido: String[20];
                    Nombres: String[20];
                    FechaNac: Longword {AAAAMMDD} ); {el registro es de una persona}
        False: (    mapa: Array[0..49] of Byte {capacidad para 50*8=400 registros} ); {el registro es de mapa de espacio libre; se define la variante con la misma extensión en bytes que la variante para personas}
        end;
    aPersonas = File of tPersona; {tipo de archivo para registros de personas con mapa de espacio libre en la posición 0}
    ctlPersonas = Record {tipo de registro de control para archivo de personas (handle); se pasa como parámetro por referencia a cualquier subprograma que acceda al archivo}
        arch: aPersonas;
        p: tPersona; {registro buffer para lectura o escritura de personas}
        longMapa: Byte; {longitud en registros del mapa, ubicado en los primeros registros del archivo}
        mapa: Array[0..9] of ^tPersona {arreglo de punteros a registros de mapa de espacio libre en memoria, para el caso de que se extienda en varios registros}
        end;
Const
    regsPorMapa = SizeOf(tPersona.mapa);
    mascEL: Byte = %10000000; {máscara para buscar espacio libre}

Procedure Crear(var a: ctlPersonas); {recibe a.arch asignado}
begin
Rewrite(a.arch);
a.p.valido:=False; a.p.mapa[0]:=0; {el primer registro del archivo está ocupado por el mapa}
Write(a.arch, a.p);
Close(a.arch)
end;

Procedure Abrir(var a: ctlPersonas); {recibe a.arch asignado}
var n: Byte; {número relativo de registro de mapa}
begin
Reset(a.arch);
{Inicialización de la longitud del mapa}
a.longMapa:=FileSize(a.arch) div regsPorMapa;
```

```
if FileSize(a.arch) mod regsPorMapa > 0 then Inc(a.longMapa);
{Carga del mapa en a.mapa}
for n:=0 to a.longMapa-1 do
        begin
        New(a.mapa[n]); Read(a.arch, a.mapa[n]^) {los registros de mapa deben mantenerse
        consecutivos desde la posición 0}
        end
end;

Procedure Cerrar(var a: ctlPersonas); {recibe a.arch abierto}
var n: Byte; {número relativo de registro de mapa}
begin
{Liberación de registros de a.mapa en RAM}
for n:=0 to a.longMapa-1 do Dispose(a.mapa[n]); {siempre que se actualiza el mapa se escribe en el
archivo inmediatamente, por lo que no es necesario reescribir sus registros al cerrar el archivo}
Close(a.arch)
end;

Function Libre(var a: ctlPersonas): Longword; {devuelve un número de registro libre (primer bit en 1 en
el mapa) o 0 si no hay}
var
        nrr: Longword; {número relativo de registro}
        masc: Byte; {máscara para la búsqueda de espacio libre}
        pos1: Byte; {posición del 1 en la máscara para la búsqueda de espacio libre (0..7)}
        encontrado: Boolean = False;
begin
nrr:=a.longMapa; {se saltean los registros del archivo que ocupa el mapa}
while (not encontrado) and (nrr<FileSize(a.arch)) do
        begin
        pos1:=nrr mod 8; if pos1>0 then masc:=mascEL shr pos1 else masc:=mascEL;
        encontrado:=(a.mapa[nrr div regsPorMapa]^.mapa[(nrr mod regsPorMapa) div 8] and masc =
        masc);
        Inc(nrr)
        end;
if encontrado then Libre:=nrr-1 then Libre:=0
end;
```

Cuando se agreguen registros al final del archivo (Libre(a.arch)=0), se debe verificar si se requiere extender el mapa; en este caso, el primer registro de datos a continuación del último registro del mapa se debe relocalizar al final del archivo para agregar en su posición original un nuevo registro de extensión del mapa.
Esta variante requiere una primitiva especial para escribir que contemple la actualización del mapa cuando se escriba en una posición libre o al final del archivo. También requiere una primitiva especial para eliminar registros, que actualice el mapa.

- Encadenamiento LIFO de registros (estructura de pila) con cabecera en el primer registro del archivo (registro inicial de control)

```
Type
        tPersona = Packed Record {tipo de registro para personas}
                valido: Boolean of {discriminante de registros para comprobar si son válidos (de
                personas) o no (de encadenamiento de posiciones libres)}
```

```
            True: (     DNI: Longword; {0..4294967295}
                        Apellido: String[20];
                        Nombres: String[20];
                        FechaNac: Longword {AAAAMMDD} ); {el registro es de una persona}
            False: (    proxLibre: Longword ); {el registro es de encadenamiento de espacio
            libre; si proxLibre=0 no hay posiciones libres en el archivo}
            end;
        aPersonas = File of tPersona; {tipo de archivo para registros de personas con registro de
        control de espacio libre en la posición 0}
        ctlPersonas = Record {tipo de registro de control para archivo de personas (handle); se pasa
        como parámetro por referencia a cualquier subprograma que acceda al archivo}
            arch: aPersonas;
            p: tPersona; {registro buffer para lectura o escritura de personas}
            libre: Longword {posición del siguiente registro libre (para ahorrar la lectura del
            primer registro del archivo cuando se requiera agregar un nuevo registro de
            persona)}
            end;

Procedure Crear(var a: ctlPersonas); {recibe a.arch asignado}
begin
Rewrite(a.arch);
a.p.valido:=False; a.p.proxLibre:=0; {no hay posiciones libres en el archivo}
Write(a.arch, a.p);
Close(a.arch)
end;

Procedure Abrir(var a: ctlPersonas); {recibe a.arch asignado}
begin
Reset(a.arch); Read(a.arch, a.p); a.libre:=a.p.proxLibre
end;
```

Las primitivas para insertar y eliminar registros deben contemplar la actualización del encadenamiento de posiciones libres.

- Archivo auxiliar con registro de posiciones relativas de registros eliminados del archivo de datos (archivo de control)

```
Type
        tPersona = Packed Record {tipo de registro para personas}
            valido: Boolean of {discriminante de registros para comprobar si son válidos (de
            personas) o no (borrados)}
            DNI: Longword; {0..4294967295}
            Apellido: String[20];
            Nombres: String[20];
            FechaNac: Longword {AAAAMMDD}
            end;
        aPersonas = File of tPersona; {tipo de archivo para registros de personas con mapa de espacio
        libre en la posición 0}
        aLibres = File of Longword; {tipo de archivo para posiciones libres de registros de personas}
        ctlPersonas = Record {tipo de registro de control para archivo de personas (handle); se pasa
        como parámetro por referencia a cualquier subprograma que acceda al archivo}
            arch: aPersonas;
```

```
        p: tPersona; {registro buffer para lectura o escritura de personas}
        libres: aLibres {archivo de posiciones libres en arch}
        end;

Procedure Crear(var a: ctlPersonas); {recibe a.arch y a.libres asignados}
begin
Rewrite(a.arch); Rewrite(a.libres); {crea a ambos archivos vacíos}
Close(a.arch); Close(a.libres)
end;

Procedure Abrir(var a: ctlPersonas); {recibe a.arch y a.libres asignados}
begin
Reset(a.arch); Reset(a.libres)
end;

Procedure Cerrar(var a: ctlPersonas); {recibe a.arch y a.libres abiertos}
begin
Close(a.arch); Close(a.libres)
end;
```

Las primitivas para insertar y eliminar registros deben contemplar la actualización del archivo de posiciones libres (en la inserción de registros, si hay espacio libre –FileSize(a.libres)>0– se lee el último registro del archivo de posiciones libres posicionándose antes con Seek(a.libres, FileSize(a.libres)-1), se vuelve atrás una posición con Seek(a.libres, FilePos(a.libres)-1) y se trunca al archivo con Truncate(a.libres); en la eliminación de registros, la posición del registro eliminado se agrega al final del archivo de posiciones libres).
No es recomendable para programas con muchos archivos, ya que el Sistema Operativo maneja una tabla general de archivos abiertos por programas en ejecución (procesos), y al abrir un archivo puede ocurrir el error 4 (demasiados archivos abiertos).

Para archivos con **registros de longitud variable organizados en bloque**, las opciones típicas son

- Listas de bytes libres por bloque del archivo en el primer bloque del archivo (bloque de control)

```
Const
        LongBloque = 1024;
Type
        tNroBloque = Word; {tipo para números de bloque, que se puede cambiar a LongWord para archivos muy grandes}
        tBloque = Array[1..LongBloque] of Byte;
        abPersonas = File; {tipo de archivo para empaquetar registros de longitud variable de personas en bloques}
        tEstado = (C, E, LE); {estado del archivo: (C(errado), abierto para E(scrituras) secuenciales sin búsqueda de espacio libre (appends), o para L(ectura)E(scritura) con posicionamiento previo en bloques}
Const
        LongListaEL = LongBloque div SizeOf(Word);
Type
        tBloqueCtl = Array[1..LongListaEL] of Word; {tipo de bloque para la lista de bytes libres por bloque del archivo (no incluye al bloque 0, que es el de control)}
```

```
        tPersona = Record {tipo de registro desempaquetado para personas}
                DNI: Longword; {0..4294967295}
                Apellido: String[20];
                Nombres: String[20];
                FechaNac: Longword {AAAAMMDD}
                end;
        ctlPersonas = Record {tipo de registro de control para archivo de personas (handle); se pasa
        como parámetro por referencia a cualquier subprograma que acceda al archivo}
                estado: tEstado;
                arch: abPersonas; {archivo de bloques de personas}
                b: tBloque {bloque buffer para leer o escribir en el archivo de personas}
                ib: Word; {índice de posicionamiento en b}
                libres: tBloqueCtl {bloque de control de espacio libre (primero del archivo), para
                conservarlo en memoria cuando el archivo está abierto}
                pe: Array[1..60] of Byte; {registro buffer para registros empaquetados de personas}
                lpe: Byte; {longitud de registro empaquetado de persona}
                p: tPersona; {registro buffer para registros desempaquetados de personas}
        end;

Procedure Crear(var a: ctlPersonas); {recibe a.arch asignado y lo devuelve abierto para escrituras
secuenciales}
begin
Rewrite(a.arch, LongBloque);
BlockWrite(a.arch, a.libres, 1); {no hace falta inicializarlo porque sólo se consultará espacio libre para
los bloques de datos del archivo, y aún no hay ninguno}
a.estado:=E {abierto para escrituras secuenciales}
a.ib:=1 {inicialización del puntero de posicionamiento en el bloque buffer para escrituras}
end;

Procedure Abrir(var a: ctlPersonas; modo: tEstado); {recibe a.arch asignado}
begin
Reset(a.arch, LongBloque);
a.estado:=modo;
BlockRead(a.arch, a.libres) {lectura del bloque de control con la lista de bytes libres en los bloques de
datos del archivo}
if modo=E
then begin
        Seek(a.arch, FileSize(a.arch)-1); {posicionamiento en el último bloque del archivo}
        BlockRead(a.arch, a.b, 1); {lectura del último bloque del archivo}
        ib:=LongBloque - a.libres[a.pb]+1 {inicialización del puntero de posicionamiento en el bloque
        buffer para escrituras}
        end
end;

Procedure Cerrar(var a: ctlPersonas); {recibe a.arch abierto}
begin
if modo=E
then begin
        BlockWrite(a.arch, a.b, 1) {escritura del último bloque del archivo}
        Seek(a.arch, 0); {posicionamiento en el primer bloque del archivo}
        BlockWrite(a.arch, a.libres, 1); {escritura del bloque de control de espacio libre del archivo}
        end
```

```
Close(a.arch);
a.Estado:=C
end;

Function Libre(var a: ctlPersonas): tNroBloque; {recibe en a.LE la longitud del registro empaquetado a
insertar en el archivo y devuelve la posición de un bloque con espacio libre para guardarlo, o 0 si no
hay ninguno}
var
        pb: Word;
        encontrado: Boolean = False;
begin
pb:=1;
while (not encontrado) and (pb<=FileSize(a.arch)-1) do
        begin
        encontrado:=(a.libres[pb]<a.LE);
        Inc(pb)
        end;
if not encontrado then Libre:=0
end;
```

Advertencia: *en este ejemplo se asume que el archivo nunca tendrá más de LongListaEL bloques de datos; si se previera que un único bloque de control pudiere no ser suficiente, debería preverse una política de extensión del bloque de control (por ejemplo, análoga a la de mapas de registros libres, incluyendo la posibilidad de poder cargar más de un bloque de control en el registro de tipo ctlPersonas para manejar al archivo).*

- Archivo auxiliar con registro de bytes libres por cada bloque del archivo (archivo de control)

```
Const
        LongBloque = 1024;
Type
        tNroBloque = Word; {tipo para números de bloque, que se puede cambiar a LongWord para
        archivos muy grandes}
        tBloque = Array[1..LongBloque] of Byte;
        abPersonas = File; {tipo de archivo para empaquetar registros de longitud variable de
        personas en bloques}
        tEstado = (C, E, LE); {estado del archivo: (C(errado), abierto para E(scrituras) secuenciales sin
        búsqueda de espacio libre (appends), o para L(ectura)E(scritura) con posicionamiento previo
        en bloques}
        tPersona = Record {tipo de registro desempaquetado para personas}
                DNI: Longword; {0..4294967295}
                Apellido: String[20];
                Nombres: String[20];
                FechaNac: Longword {AAAAMMDD}
                end;
        ctlPersonas = Record {tipo de registro de control para archivo de personas (handle); se pasa
        como parámetro por referencia a cualquier subprograma que acceda al archivo}
                estado: tEstado;
                arch: abPersonas; {archivo de bloques de personas}
                b: tBloque {bloque buffer para leer o escribir en el archivo de personas}
                ib: Word; {índice de posicionamiento en b}
```

```
            libres: File of Word; {archivo de bytes libres en cada bloque del archivo de datos}
            libre: Word; {variable buffer para leer libres de pb}
            pe: Array[1..60] of Byte; {registro buffer para registros empaquetados de personas}
            lpe: Byte; {longitud de registro empaquetado de persona}
            p: tPersona; {registro buffer para registros desempaquetados de personas}
      end;

Procedure Crear(var a: ctlPersonas); {recibe a.arch y a.libres asignados y los devuelve abierto para
escrituras secuenciales}
begin
Rewrite(a.arch, LongBloque); Rewrite(a.libres);
a.estado:=E {abierto para escrituras secuenciales}
a.ib:=1 {inicialización del puntero de posicionamiento en el bloque buffer para escrituras}
end;

Procedure Abrir(var a: ctlPersonas; modo: tEstado); {recibe a.arch y a.libres asignados}
begin
Reset(a.arch, LongBloque); Reset(a.libres);
a.estado:=modo;
if modo=E
then begin
      Seek(a.arch, FileSize(a.arch)-1); {posicionamiento en el último bloque del archivo}
      BlockRead(a.arch, a.b, 1); {lectura del último bloque del archivo}
      Seek(a.libres, FileSize(a.libres)-1); {posicionamiento en el registro de bytes libres del último
      bloque del archivo}
      Read(a.libres, a.libre); {lectura de bytes libres en el último bloque del archivo}
      ib:=LongBloque – a.libre+1 {inicialización del puntero de posicionamiento en el bloque buffer
      para escrituras}
      end
end;

Procedure Cerrar(var a: ctlPersonas); {recibe a.arch abierto}
begin
if modo=E
then begin
      BlockWrite(a.arch, a.b, 1); {escritura del último bloque del archivo}
      Write(a.libres, alibre)
      end
Close(a.arch); Close(a.libres);
a.Estado:=C
end;

Const
      NoHay: tNroBloque = High(tNroBloque); {indica que no se encontró un bloque del archivo con
      espacio libre para insertar un registro de determinada longitud}

Function Libre(var a: ctlPersonas): tNroBloque; {recibe en a.LE la longitud del registro empaquetado a
insertar en el archivo y devuelve en a.pb la posición de un bloque con espacio libre para guardarlo, o
NoHay si no hay ninguno}
var
      encontrado: Boolean = False;
begin
```

```
Seek(a.libres, 0);
while (not encontrado) and (FilePos(a.libres)<FileSize(a.libres)) do
        begin
        Read(a.libres, a.libre);
        encontrado:=(a.libre<a.LE)
        end;
if encontrado then Libre:=FilePos(a.libres)-1) else Libre:= NoHay
end;
```

Esta variante es similar a la del archivo de posiciones de registros de longitud fija borrados, sólo que lo que se almacena en el archivo de espacio libre son los bytes libres de cada bloque del archivo, incluyendo al primero (el registro 0 también se usa para datos). Tampoco es recomendable para programas con muchos archivos.

- **Organización de registros de longitud variable en bloques**
  Para manipular los registros en un programa en Pascal, se emplean tipos ***Record*** que pueden incluir campos de tipo ***String***, que tienen una representación en memoria de tamaño fijo (la máxima longitud definida más un byte inicial para la longitud real), y, para atributos polivalentes, pueden incluir bien un ***Array*** junto con un campo que indique cuántos valores hay efectivamente cargados o bien un ***puntero*** a una lista en memoria dinámica.
  Para guardar estos registros en un bloque en el archivo, se debe copiar el valor de cada campo del registro a un ***arreglo de bytes*** (empaquetamiento del registro) respetando las longitudes reales, que luego debe agregarse al bloque (otro arreglo de bytes de mayor tamaño) para luego escribirlo en el archivo. Para leer registros del archivo, debe leerse un bloque como arreglo de bytes, y luego copiar del bloque campo por campo del registro empaquetado a los campos del registro de Pascal. Para copiar valores Pascal dispone del procedimiento *Move(dirOrigen, dirDestino, cantBytes)*.
  Para el empaquetamiento de registros hay distintas alternativas:
  - Almacenar los valores numéricos con la representación de Pascal y los strings con prefijos de longitud.

    Así, para copiar un campo r.n de cualquier tipo numérico y otro r.s de tipo string[30] a un registro empaquetado re de tipo Array[1..100] of Byte, se debe disponer de un índice i de llenado de re que apunte al primer byte libre del mismo, y hacer:

    ```
    Move(r.n, re[i], SizeOf(r.n)); Inc(i, SizeOf(r.n));
    Move(r.s, re[i], Length(r.s)+1); Inc(i, Length(r.s)+1);
    ```

    Para copiar el registro empaquetado a un bloque b de tipo Array[512] of Byte, se debe disponer de otro índice ib para apuntar al primer byte libre del bloque y hacer:

    ```
    Move(re[1], b[ib], i-1); Inc(ib, i-1);
    ```

    A estos registros se los puede prefijar con un campo de control para representar la longitud del registro empaquetado, que resulta útil cuando se debe buscar registros secuencialmente dentro de un bloque; en tal caso, para copiar el registro al bloque se haría:

    ```
    Dec(i); {se resta 1 a i para representar la longitud del registro empaquetado}
    Move(i, b[ib], SizeOf(i)); Inc(ib, SizeOf(i));
    Move(re[1], b[ib], i); Inc(ib, i); {i puede ser de tipo Byte o Word, según la longitud máxima que se prevea para los registros empaquetados}
    ```

  - Almacenar todos los campos convertidos a string con separadores de campos y de registros.

    Así, para copiar un campo r.n de cualquier tipo numérico y otro r.s de tipo string[30] a un registro empaquetado re de tipo Array[100] of Byte, se debe disponer de un índice i de llenado de re que apunte al primer byte libre del mismo, de una variable auxiliar sx para convertir campos numéricos a

string y en cualquier caso concatenarle el separador al final, y de dos constantes de tipo char SC para separar campos y SR para separar registros y hacer:

```
Str(r.n, sx); sx:=sx+SC; Move(sx[1], re[i], Length(sx)); Inc(i, Length(sx));
sx:=s+SC; {suponiendo que s no es el último campo del registro, si no, se concatenaría SR}
Move(sx[1], re[i], Length(sx)); Inc(i, Length(sx));
```

Para representar campos con valores nulos se almacena únicamente SC (o SR si fuera el último del registro).

Para empaquetar atributos polivalentes, para ambas alternativas se puede almacenar antes de los valores un campo de control que indique la cantidad de valores que se almacenan, que puede ser de tipo Byte (no es necesario convertir el valor a string, para que ocupe un solo byte). Para esto se define al bloque como arreglo de bytes, en lugar de un arreglo de caracteres.

Para desempaquetar registros de un bloque y copiar sus campos a un registro de Pascal, puede prescindirse del arreglo re y copiar los valores directamente del el bloque a los campos del registro.
Para la primera opción, asumiendo que ib apunta al primer byte del campo numérico, se hace:

```
Move(b[ib], r.n, SizeOf(r.n)); Inc(ib, SizeOf(r.n));
Move(b[ib], r.s, b[ib]+1); Inc(ib, b[ib]+1); {antes del incremento, ib apunta a la longitud del campo s empaquetado}
```

Para la segunda opción, para copiar los campos primero hay que recorrer b desde la posición i hasta encontrar a SC o SR para determinar la longitud de los campos en su representación en caracteres lc (por ejemplo, se puede codificar una función longCampo que reciba como parámetros a b y a ib, ya posicionado al comienzo del campo, y que devuelva la longitud sin considerar al separador); entonces se haría:

```
lc:=longCampo(b, ib); Move(b[ib], sx, lc); SetLength(sx, lc); Val(sx, r.n, cod); Inc(ib, lc+1);
lc:=longCampo(b, ib); Move(b[ib], r.s, lc); SetLength(r.s, lc); Inc(ib, lc+1);
{En ambos casos ib se incrementa en lc+1 para saltear al separador; cod es el código de retorno del procedimiento Val, que en caso de no ser 0 indica que no se pudo hacer la conversión (se puede agregar código para contemplar casos de error)}
```

1.5 Conceptos de Organización de Archivos

La organización de un archivo implica cuestiones como determinar la organización de los registros y dónde almacenar registros nuevos y cómo encontrar registros dentro del archivo para eliminarlos, modificarlos o recuperarlos para consulta.

Los esquemas de organización se basan en los ***modos de acceso*** a archivos que proveen los sistemas operativos: ***secuencial*** (los registros se acceden en orden de posición) y ***relativo*** (se accede a registros con posicionamiento previo -seek).

Para determinar cómo organizar un archivo se debe considerar los patrones de acceso que requiere.

Las organizaciones indicadas cuando hay predominio de acceso secuencial son la secuencial, balanceadas secuenciales (B+ o B#) o la secuencial indexada; y cuando hay predominio de acceso relativo son las balanceadas no secuenciales (B o B*), las directas o la indexada.

1.6 Primitivas de Organización de Archivos

Se implementan como tipos de datos abstractos y deben especializarse para cada organización particular.

- **De Creación**
  Creación y carga inicial sin validación de unicidad ni búsqueda de espacio libre (alta de carga).

- **De Actualización de Registros**
  Inserción con validación de unicidad y búsqueda de espacio libre, modificación y supresión.
- **De Recuperación de Registros**
  Consulta o recuperación unitaria de registros, y reporte (exportación a un archivo de texto).
- **De Mantenimiento**
  Depuración (copia de registros recientes a un archivo nuevo, sólo para archivos transaccionales), y respaldo secuencial sin espacio libre.

2 Organización Secuencial

Es la organización más simple y la primera en aparecer, ya que era la única posible cuando los únicos dispositivos de almacenamiento permanente eran las cintas magnéticas.

Se basa en el acceso secuencial pero también puede optimizarse utilizando acceso relativo.

La organización de los registros varía según sean de longitud fija o variable, y en caso de que sean de longitud variable, según se requiera actualizarlos o no: los registros de longitud variable actualizables se organizan en bloques; los no actualizables (transacciones efectuadas) se pueden organizar secuencialmente.

Las primitivas varían según los registros se dispongan ordenados por un identificador o desordenados, y según se requiera actualizarlos o no (en muchos archivos transaccionales sólo se hacen altas al final del archivo y nunca se actualizan registros).

2.1 Registros Ordenados por Identificador

Para determinado tipo de operaciones, conviene que los registros estén ordenados por identificador (por ejemplo cuando hay que hacer una fusión de archivos sin registros duplicados o realizar actualizaciones con el esquema maestro/detalles); para esto puede ser que el archivo de datos se mantenga ordenado, o que sea necesario ordenarlo mediante un proceso de ordenamiento externo (como no se puede cargar el archivo completo en memoria RAM, se ordena por fragmentos que luego se fusionan).

El mantenimiento de archivos secuenciales ordenados tiene las siguientes ventajas y desventajas:

Ventajas	Desventajas
→ Permite optimizar búsquedas: elimina necesidad de leer todo el archivo (búsqueda binaria) → Permite procesamiento coordinado con otros archivos → Permite cortes de control con un único recorrido	→ Problemas de inserción: altas implican la reconstrucción del archivo o deben diferirse → Bajas costosas: bajas lógicas con necesidad de reestructuración, o bajas físicas con reconstrucción del archivo o diferidas

2.2 Casos de Archivos Secuenciales

- Archivos maestros con pocos registros y pocas actualizaciones
- Archivos de trabajo
  - Reordenación de transacciones para totalizaciones parciales (cortes de control –p.e. ordenación de líneas de facturas por identificación de producto, para totalizar ventas por producto)
  - Resultados parciales (persistencia de operaciones no terminadas - p.e. consumos en mesas abiertas en un restaurante)

2.3 PRIMITIVAS

2.3.1 Archivos Secuenciales con Registros Desordenados

- **De Creación**
  - Creación para escribir con Rewrite(a) e inicialización de estructuras para manejo de espacio libre.
  - Carga de registros por importación desde archivo de texto usando la primitiva de alta de carga (no comprueba unicidad de registros ni busca espacio libre –los registros siempre se agregan al final del archivo), con el archivo recién abierto por su creación con Rewrite(a), o con Reset(a) y posicionamiento previo al final del archivo con Seek(a, FileSize(a)).
    Para registros de longitud variable organizados en bloques, se bufferizan los bloques conservando como estructura de control del archivo abierto al último bloque con un índice al primer byte libre.

Por ejemplo, para registros de longitud fija con mapa de bits para control de posiciones libres (ver definición de tipos y primitivas para crear, abrir, cerrar y buscar espacio libre en 1.4.2):

```
Procedure Ocupar(var a: ctlPersonas; pos: Longword); {marca en el mapa de posiciones libres al bit correspondiente a pos indicando posición ocupada (con 0).
Precondición: se invoca luego de escribir en el archivo mediante Cargar o Insertar, de modo que pos=FilePos(a.arch)-1}
var
        n: Byte; {número relativo de registro de mapa}
        masc: Byte; {máscara para poner en 0 el bit correspondiente a la posición ocupada}
        pos1: Byte; {posición del 1 en la máscara para poner en 0 con xor el bit correspondiente a la posición ocupada (0..7)}
begin
n:=pos div regsPorMapa; pos1:=pos mod 8;
if n=a.longMapa {corresponde extender el mapa un registro}
then begin
        {Relocalización del primer registro de persona del archivo para extender el mapa}
        Seek(a.arch, n); Read(a.arch, a.p); Seek(a.arch, FileSize(a.arch)); Write(a.arch, a.p);
        {Extensión del mapa un registro}
        New(a.mapa[n]); a.mapa[n]^.valido:=False; a.mapa[n]^.mapa[0]:=0; Inc(a.LongMapa)
        end
else if (pos=FileSize(a.arch)-1) and (pos1=0) {corresponde extender el mapa un byte}
        then a.mapa[n]^.mapa[(pos mod regsPorMapa) div 8] :=0
        else begin {corresponde invertir un bit en 1 del mapa}
                if pos1>0 then masc:=mascEL shr pos1 else masc:=mascEL;
                a.mapa[n]^.mapa[(pos mod regsPorMapa) div 8]:= a.mapa[n]^.mapa[(pos mod
                regsPorMapa) div 8] xor masc
                end;
Seek(a.arch, n); Write(a.arch, a.mapa[n]^) {se actualiza el mapa en el archivo}
end;

Procedure Cargar(var a: ctlPersonas);
{Precondiciones: a.arch abierto para leer y escribir y en condición FilePos(a.arch)=FileSize(a.arch), ya que esta primitiva se usa para cargas consecutivas de muchos registros; registro a cargar en a.p}
var
        nrr: Longword; {número relativo de registro}
begin
nrr:=FilePos(a.arch); Write(a.arch, a.p); Ocupar(a.arch, nrr)
```

```
end;
```

- **De Recuperación de Registros**
  Deben contemplarse primitivas para recuperar el primer registro del archivo, para recuperar el siguiente registro cualquiera sea la posición actual, para recuperar un registro por identificador (usando las dos primitivas anteriores) y para reportar datos en un archivo de texto, de acuerdo con la política de control de espacio libre.

Por ejemplo, para registros de longitud fija con encadenamiento LIFO de registros (estructura de pila) con cabecera en el primer registro del archivo para control de posiciones libres (ver definición de tipos y primitivas para crear, abrir y cerrar en 1.4.2):

```
Type tResult = (Ok, duplicado, inexistente); {códigos de resultado para primitivas de carga, recuperación y actualización de registros}

Procedure Primero(var a: ctlPersonas; var cod: tResult); {recibe a.arch abierto para leer y escribir y devuelve en a.p al primer registro de persona del archivo}
begin
Seek(a.arch, 1); cod:=inexistente;
while (not eof(a.arch)) and (cod=inexistente) do begin Read(a.arch, a.p); if a.p.valido then cod:=Ok end
end;

Procedure Siguiente(var a: ctlPersonas; var cod: tResult); {recibe a.arch abierto para leer y escribir y devuelve en a.p al siguiente registro de persona del archivo}
begin
cod:=inexistente;
while (not eof(a.arch)) and (cod=inexistente) do begin Read(a.arch, a.p); if a.p.valido then cod:=Ok end
end;

Procedure Recuperar(var a: ctlPersonas; dni: Longword; var cod: tResult); {recibe a.arch abierto para leer y escribir y devuelve en a.p al registro de persona del archivo con a.p.DNI=dni}
begin
Primero(a, cod);
if cod=Ok then while (a.p.DNI<>dni) and (cod=Ok) do Siguiente(a, cod);
end;

Procedure Reportar(var a: ctlPersonas; reporte: Text); {recibe a.arch abierto para leer y escribir y reporte asignado y sin abrir}
begin
Rewrite(reporte);
Primero(a, cod);
if cod=Ok
then repeat
        WriteLn(reporte, a.p.DNI:8, ' ', a.p.Apellido:21, a.p.Nombres:21, a.p.FechaNac:8);
        Siguiente(a, cod)
        until cod=inexistente;
Close(reporte)
end;
```

Para registros de longitud variable organizados en bloques, se bufferizan los bloques conservando como estructura de control del archivo abierto al último bloque accedido con un índice al comienzo del siguiente registro.

- **De Actualización de Registros**
  - Inserción: se intenta recuperar un registro con el mismo identificador que el del registro a insertar para validar unicidad; se graba el nuevo registro buscando espacio libre.
  - Modificación: se recupera el registro por el identificador, se actualiza, se vuelve al comienzo de la unidad recién leída (registro o bloque) y se escribe la unidad modificada (en bloques el registro modificado queda al final y puede haber relocalización en caso de que una vez modificado no quepa en el mismo bloque).
  - Eliminación: para registros de longitud fija se recupera el registro a eliminar por el identificador y se libera su posición; para registros de longitud variable se compacta el bloque donde se encontró el registro, se actualiza el espacio libre del bloque y se reescribe el bloque.
- **Mantenimiento**
  Se crea una copia del contenido del archivo sin espacio libre; para registros de longitud variable organizados en bloque, se graban secuencialmente sin organizarlos en bloques.

2.3.2 Archivos Secuenciales con Registros Ordenados

- **De Creación**
  Creación y carga sin validación de unicidad (los registros a cargar ya están validados y ordenados por identificador) u ordenamiento externo de un archivo desordenado.

Se necesita contabilizar los votos de diferentes mesas electorales por provincia y localidad.
Para ello, se posee un archivo con la siguiente información: código de provincia, código de localidad, número de mesa, y cantidad de votos. Los registros están ordenados por códigos de provincia y de localidad.

```
Program CargaVotos;
Uses CRT;
Type
        tRegistroVotos = Record
                codProv: integer;
                codLoc: integer;
                nroMesa: integer;
                cantVotos: integer;
                end;

        tArchVotos = File of tRegistroVotos;

Var
  opc: Byte;
  nomArch, nomArch2: String;
  arch: tArchVotos; carga: Text;
  votos: tRegistroVotos;

begin
WriteLn('VOTOS');
WriteLn;
WriteLn('0. Terminar el Programa');
WriteLn('1. Crear un archivo');
WriteLn('2. Abrir un archivo existente');
Window(1,7,80,22);
Repeat
Write('Ingrese el nro. de opcion: '); ReadLn(opc);
```

```
If (opc=1) or (opc=2) then begin
  WriteLn;
  Write('Nombre del archivo de votos: ');
  ReadLn(nomArch);
  Assign(arch, nomArch)
  end;
Case opc of
   1: begin
     Write('Nombre del archivo de carga: ');
     ReadLn(nomArch2);
     Assign(carga, nomArch2); Reset(carga); Rewrite(arch);
     Repeat
     With votos do ReadLn(carga, codProv, codLoc, nroMesa, cantVotos);
     Write(arch, votos);
     until eof(carga);
     Write('Archivo cargado. Oprima tecla de ingreso para continuar...');
     ReadLn;
     Close(arch); Close(carga)
     end;
   2: begin
     Reset(arch);
     WriteLn;
     Repeat
     Read(arch, votos);
     With votos do WriteLn(codProv:5, codLoc:5, nroMesa:5, cantVotos:5);
     until eof(arch);
     Close(arch);
     WriteLn;
     Write('Oprima cualquier tecla para continuar...');
     ReadLn
     end;
end;
ClrScr
until opc=0;
end.
```

En un archivo de texto se debe disponer de los datos para probar el programa, que incluye las totalizaciones precalculadas para el control de resultados del programa (no leídas por el programa). Por ejemplo:

```
1 1 1 193
1 1 2 148 Localidad 1: 341
1 2 1 195
1 2 2 190 Localidad 2: 385 - Provincia 1: 726
2 1 1 184
2 1 2 191 Localidad 1: 375 - Provincia 2: 375
```

- **De Recuperación**
  - Consulta unitaria: por aproximación lineal o binaria (no se leen todos los registros previos al buscado).
  - Reporte o recuperación comprensiva: cortes de control (cuando los registros están ordenados por más de un campo, se puede realizar cálculos con secuencias de registros que tengan el mismo valor de uno o más campos –p.e. contar registros, promediar o sumar valores de otro campo, etc.).

```
Program ContarVotos;
Const
        valorAlto=99;
Type
        tRegVotos = Record
                codProv: integer;
                CodLoc: integer;
                nroMesa: integer;
                cantVotos: integer;
                end;

        tArchivo = File of tRegVotos;

procedure leer (var a: tArchivo; var dato: tRegVotos);
begin
if (not EoF( a ))
        then  read (a, dato)
        else dato.codProv:=valorAlto
end;

Var
        archivo: tArchivo;
        cantVotosProvincia, cantVotosLocalidad: integer;
        regVotos: tRegVotos;
        codProvAct, codLocAct: integer; {Códigos Activos}

Begin
Assign(archivo, 'votos.dd');
Reset(archivo);
WriteLn('VOTOS'); WriteLn;
Leer(archivo, regVotos);
With regVotos do
   While (codProv <> valorAlto) do
   begin
   codProvAct:=codProv;
   cantVotosProvincia:=0;
   writeLn; writeLn('Provincia ', codProvAct);
   while (codProv=codProvAct) do
            begin
            codLocAct:=codLoc;
            cantVotosLocalidad:=0;
            write('    Localidad ', codLocAct);
            while (codProv=codProvAct) and (codLoc=codLocAct) do
                    begin
                    cantVotosLocalidad:=cantVotosLocalidad+cantVotos;
                    Leer(archivo, regVotos)
                    end;
            cantVotosProvincia:=cantVotosProvincia+cantVotosLocalidad;
            writeLn(': ', cantVotosLocalidad, ' votos')
            end;
   writeLn('Total de votos en la provincia: ', cantVotosProvincia)
   end;
```

```
Close(archivo);
WriteLn; Write('Oprima tecla de ingreso para finalizar...'); ReadLn
end.
```

- **De Actualización de Registros**
  - Inserción: se debe crear un archivo nuevo, copiar los registros con identificadores menores al del registro a insertar, agregar el registro nuevo, copiar el resto de los registros con identificadores mayores, borrar el archivo viejo y renombrar el archivo nuevo con el nombre del viejo.
    También se puede diferir las inserciones manteniendo ordenados los registros a agregar, y luego insertarlos por fusión (encarece recuperaciones por necesidad de consultar el archivo principal y el de inserciones).
  - Modificación: se recupera el registro por el identificador, se actualiza, se vuelve al comienzo de la unidad con el registro y se reescribe (el identificador nunca se puede modificar)
    También puede haber modificaciones por procesamiento coordinado (esquema maestro – detalles).

```
Program ProcesaProductos;

{Un hipermercado tiene 30 cajas que van registrando los productos que facturan en archivos
independientes. Los registros de venta de productos contienen el número de ticket, el código del
producto y la cantidad de unidades del producto.
Al finalizar cada jornada, los archivos correspondientes a las cajas se ordenan por código de producto,
para determinar cuántas unidades de cada producto se vendieron en la jornada y actualizar el archivo
de productos.
Los registros del archivo de productos contienen el código del producto, la descripción, la cantidad en
existencia, y el precio de venta actual.
Se realiza un programa para:
a) Descontar de la existencia de cada producto y registrar la cantidad vendida en la jornada
b) Informar si no se vendió ninguna unidad de algún producto que tenga unidades en existencia
c) Informar los productos cuyas ventas alcancen o superen el 20% de la cantidad en existencia previa.
Usualmente hay cajas que no se habilitan en toda una jornada, por lo que el programa debe funcionar
para cualquier número de cajas hasta 30.}

Uses CRT;

Const
  totCajas = 30;
  codProdFin = 65535;

Type
        tRegistroProducto = Record
                codProd: Word;
                descrip: String;
                existencia: Word;
                precio: Real
                end;

        tArchProductos = File of tRegistroProducto;

        tRegistroVenta = Record
```

```
                nTicket: LongInt;
                codProd: Word;
                cant: Word
                end;

        tArchVentas = File of tRegistroVenta;

   tCajas = Record
     cantCajas: Byte; {cantidad de cajas que se procesan}
     ventasCaja: Array[1..totCajas] of tArchVentas;
     ventaCaja: Array[1..totCajas] of tRegistroVenta
     end;

Procedure LeerProducto(var app: tArchProductos; var p: tRegistroProducto);
Begin
If eof(app) then p.codProd:=codProdFin else Read(app, p)
end;

Procedure LeerVenta(var avv: tArchVentas; var v: tRegistroVenta);
Begin
If eof(avv) then v.codProd:=codProdFin else Read(avv, v)
end;

Function Min(var cc: tCajas): Byte;
Var iMin, i: Byte;
Begin
iMIn:=1;
With cc do For i:=2 to cantCajas do
   If ventaCaja[i].codProd<ventaCaja[iMin].codProd
   then iMin:=i;
Min:=iMin
end;

Var
  nomArch: String;
  productos: tArchProductos;
  producto: tRegistroProducto;
  cajas: tCajas;
  iCaja: Byte; {indice de caja}
  iCajaCodMin: Byte; {indice de caja con codigo de producto minimo}
  cantVend: Word; {cantidad de productos vendidos de un mismo codigo}

begin
{Inicializaciones}
ClrScr;
WriteLn('ACTUALIZACION E INFORME DE PRODUCTOS');
WriteLn;
Write('Ingrese el nombre del archivo de productos: '); ReadLn(nomArch);
Assign(productos, nomArch); Reset(productos); LeerProducto(productos, producto);
WriteLn;
Write('Ingrese a cantidad de cajas a procesar: '); ReadLn(cajas.cantCajas);
WriteLn;
With cajas do
```

```
  For iCaja:=1 to cantCajas do begin
  WriteLn;
  Write('Ingrese el nombre del archivo de ventas de la caja ', iCaja, ': ');
  ReadLn(nomArch);
  Assign(ventasCaja[iCaja], nomArch); Reset(ventasCaja[iCaja]);
  LeerVenta(ventasCaja[iCaja], ventaCaja[iCaja])
  end;
iCajaCodMin:=Min(cajas);  cantVend:=0;

{Proceso}
WriteLn; WriteLn;
With producto, cajas do
  While (codProd<>codProdFin) do
    If codProd<ventaCaja[iCajaCodMin].codProd
    then begin {se procesa el producto}
      If (cantVend=0) and (existencia>0)
      then begin {no se vendio ninguna unidad del producto}
        Write('No se vendio ninguna unidad del prod. ', codProd);
        WriteLn(': '+descrip)
        end
      else {(cantVend<>0) or (existencia=0)}
        If cantVend<>0
        then begin {se vendio alguna unidad del producto}
          If cantVend>=existencia*0.2
          then begin {se vendio el 20% o mas de la existencia}
          Write('se vendio el 20% o mas de la existencia del prod. ');
          WriteLn(codProd, ': '+descrip)
          end;
          {Actualizacion del producto}
          Dec(existencia, cantVend);
          Seek(productos, FilePos(productos)-1);
          Write(productos, producto);
          cantVend:=0;
          end;
        LeerProducto(productos, producto)
        end
    else While codProd=ventaCaja[iCajaCodMin].codProd do
      begin {se procesa la venta}
      Inc(cantVend, ventaCaja[iCajaCodMin].cant);
      LeerVenta(ventasCaja[iCajaCodMin], ventaCaja[iCajaCodMin]);
      iCajaCodMin:=Min(cajas)
      end;

{Finalizacion}
Close(productos);
For iCaja:=1 to cajas.cantCajas do Close(cajas.ventasCaja[iCaja]);
WriteLn;
Write('Pulse la tecla de ingreso para finalizar...'); ReadLn
end.
```

- Eliminación: se puede realizar en forma análoga a la inserción (individual o en forma diferida).

También se puede realizar en forma lógica marcando el registro.

- **Mantenimiento**
  Copias de respaldo con comandos del sistema operativo.
  Si se hacen eliminaciones lógicas, se puede requerir eliminar físicamente los registros marcados en forma periódica (compactación del archivo).

2.4 EJERCICIOS

1. Hacer un programa con un menú principal cíclico con opciones para
   a) Crear y cargar un archivo de números enteros arch: File of Integer con una cantidad de números solicitada al usuario que se generan pseudoaleatoriamente con e:=Random(maxUIntValue) – maxint (antes de la generación se debe inicializar a la función generadora con Randomize). El archivo debe llamarse por defecto 'enteros'.
   b) Abrir un archivo existente de números enteros, con un submenú cíclico para
      i) Exportar los números generados a un archivo de texto por defecto 'numeros.txt', de a doce por línea y en columnas de seis caracteres de ancho: Write(t, e:6), con t: Text y asignado con 'numeros.txt'.
      ii) Informar la cantidad de números positivos y negativos del archivo, con opción de exportarlos a los archivos de texto por defecto 'positivos.txt' y 'negativos.txt', de a doce por línea y también encolumnados.
      iii) Informar la cantidad de números pares (n mod 2 = 0) e impares (n mod 2 = 1) del archivo, con opción de exportarlos a los archivos de texto por defecto 'pares.txt' y 'nones.txt', de a doce por línea y también encolumnados.
      iv) Informar la cantidad de números pares negativos, impares negativos, pares positivos e impares positivos con opción de exportarlos a los archivos por defecto 'pares_neg.txt', 'nones_neg.txt', 'pares_pos.txt' y 'nones_pos.txt', de a doce por línea y también encolumnados.

2. Hacer un programa con un menú principal cíclico con opciones para
   a) Crear y cargar un archivo de números primos arch: **File of Longword** con la cantidad de los primeros números primos ordenados que indique el usuario. El archivo debe llamarse por defecto 'primos'.
   b) Abrir un archivo de números primos existente y agregarle la cantidad especificada por el usuario de los siguientes números primos.
   c) Abrir un archivo de números primos y exportarlos a un archivo de texto 'primos.txt' de a siete por línea y en columnas de once caracteres de ancho.

   Al crear el archivo debe cargarse con los primeros números primos 2 y 3; luego, para obtener cada uno de los primos siguientes, a partir del último obtenido p del archivo se inicia una secuencia de testeos a los siguientes, calculando el módulo del número a testear con cada primo obtenido hasta el momento desde el comienzo del archivo hasta obtener un módulo 0 (no es primo) o agotar todos los primos del archivo (es primo).

3. Una empresa posee un archivo ordenado por código de promotor con información de las ventas realizadas por cada uno de ellos (código de promotor, nombre y monto de venta). Sabiendo que en ese archivo pueden existir uno o más registros por cada promotor, realice un procedimiento que reciba el archivo anteriormente mencionado y lo compacte (esto es, generar un nuevo archivo donde cada promotor aparezca una única vez con sus ventas totales).

   NOTA: No se conoce a priori la cantidad de promotores, y el archivo debe recorrerse una única vez.

4. Una empresa debe procesar mensualmente cinco (5) archivos de detalle con las horas extras del mes de los empleados de cada una de sus sedes actuales, para la liquidación de sueldos de los empleados de toda la fábrica.
   Los registros de todos los archivos de detalle son iguales y se componen del número de empleado, fecha y cantidad de horas extras.
   La administración de la empresa cuenta con un archivo maestro donde para cada empleado registra: número de empleado, número sede, nombre, sueldo básico y el monto que cobra cada hora extra.
   Todos los archivos están ordenados por número de empleado y en los de detalle de horas extras puede haber 0, 1 o más registros de cada empleado en orden ascendente de fecha.
   Declare los tipos de datos necesarios y codifique un procedimiento que reciba el archivo maestro y los archivos de detalle ya asignados y sin abrir, y que simultáneamente:

   a) Genere un nuevo archivo (ordenado por el mismo criterio) con la siguiente estructura: número de empleado y total a cobrar (liquidación total del sueldo).
   b) Imprima en un archivo de texto: número de empleado, número de sede, nombre y total a cobrar

   Nota: Los archivos deben ser recorridos una sola vez.

5. Hacer una versión del programa ProcesaProductos que sólo actualice el stock de productos sin informar ninguna otra cosa (no es necesario procesar todos los registros de productos, por lo que el ciclo principal se ejecuta mientras iCajaCodMin<>codProdFin).

6. Codificar las primitivas
   a) Cargar (agrega un registro de persona al final del archivo sin validar unicidad ni buscar espacio libre)
   b) Primero (devuelve el primer registro de persona del archivo y un código indicando el éxito de la operación o que no existe)
   c) Siguiente (devuelve el siguiente registro de persona del archivo a partir de la posición actual del programa en el archivo y un código indicando el éxito de la operación o que no existe)
   d) Recuperar (recupera un registro de persona dado un número de DNI y un código indicando el éxito de la operación o que no existe)
   e) Exportar (exporta a un archivo de texto los registros del archivo, a razón de uno por línea)
   f) Insertar (agrega un registro de persona si no existe en el archivo otro registro con el mismo DNI y devuelve un código indicando si la operación fue exitosa o si ya existía un registro con el mismo DNI)
   g) Eliminar (elimina un registro de persona con un DNI dado si existe en el archivo y devuelve un código indicando el resultado de la operación)
   h) Modificar (opera sobre un registro de persona, buscando en el archivo un registro con el mismo DNI y lo sobreescribe, y devuelve un código indicando si se encontró a esa persona y se pudo efectuar la operación, o no –no se puede cambiar el dni)
   i) Respaldar (genera una versión nueva del archivo sin espacio libre ni estructuras de control para el mismo)

   Todas las primitivas deben recibir un registro de control con toda la información necesaria para realizarlas, y devolver un código de resultado.
   Codificar las primitivas para todas las versiones de organización de registros y de control de espacio libre no incluidas en los ejemplos.