| INFORMATICA INACAP | Versión en desarrollo Beta 1.0 |
|---|---|

# MANUAL DE ESTRUCTURA DE DATOS

**Preparado por: Wilfredo Soler Jaldin.**
**Sede Antofagasta.**

# UNIDAD I: Introducción

Las computadoras fueron diseñadas o ideadas como una herramienta mediante la cual podemos realizar operaciones de alta complejidad sobre ellas en un lapso de mínimo tiempo. Pero la mayoría de las aplicaciones de este fantástico invento del hombre, son las de almacenamiento y acceso de grandes cantidades de información.

La información que se procesa en la computadora central es un conjunto de datos, que pueden ser simples o estructurados. Los datos simples son aquellos que ocupan sólo una localidad de memoria, mientras que los estructurados son un conjunto de casillas de memoria a las cuales hacemos referencia mediante un identificador único.

Debido a que por lo general tenemos que tratar con conjuntos de datos y no con datos simples(enteros, reales, booleanos, etc.) que por sí solos no nos dicen nada, ni nos sirven de mucho, es necesario tratar con estructuras de datos adecuadas a cada necesidad, permitiendo que existan operaciones que puedan acceder a la manipulación de los datos que puedan existir en las estructuras.

**CONCEPTOS**

**Datos:** Representa una abstracción de la realidad en el sentido de que ciertas propiedades y características de un objeto real se seleccionan. Por ejemplo Edad = 20.

Por otro lado se puede ver como una característica de una entidad, tratable por software, la cual puede ser procesada, a fin de obtener resultados o información útil.

**Tipo de datos:** Tipificación que caracteriza esencialmente el conjunto de valores al que pertenece una constante o sobre el que puede tomar valores una variable o expresión o cuyos elementos pueden ser generados por una función.

**Valor:** Es el conjunto de posibles expresiones(operadores y/o operando) relacionado a un dato.

Por lo anterior se tiene una primera aproximación a la definición de **Estructura de Datos**, esto es un elemento de representación de datos que esta compuesta de identificador, atributo y valor.

A lo anterior podemos agregar que es un conjunto de datos homogéneos que se tratan como una sola entidad.

Por lo tanto se puede obtener una definición de **Estructura de Datos**, en donde, se tiene que es una colección de variables (datos) posiblemente de distintos tipos de datos interralacionadas entre sí, que en su conjunto y con operadores disponibles permiten organizar e interrelacionar.

Las estructuras de datos pueden ser divididas en dos grandes tipos, que son:

**Estructuras de Datos Estáticas.**

Son aquellas en las que se asigna una cantidad fija de memoria cuando se declara la variable.

En grandes ocasiones se necesitan colecciones de datos que crezcan y reduzcan su tamaño en memoria a medida que el programa progrese. Esto se logra implementando las estructuras dinámicas.

| Simple | Compuestas |
|---|---|
| BOOLEAN | Arreglos |
| CHAR | Conjuntos |
| INTEGER | Strings |
| REAL | Archivos |

**Estructuras de datos Dinámicas.**

Son aquellas cuya ocupación en memoria puede aumentar o disminuir en tiempo de ejecución.

| Lineales | No lineales |
|---|---|
| Pila | Árboles |
| Cola | Grafos |
| Lista | |

Ver figura 1.1, para mas detalles del espectro global de las estructuras de datos.

**Importancia y utilidad de las estructuras de datos.**

La programación estructurada significa escribir un programa de acuerdo a las siguientes reglas.

1. - El programa tiene un diseño modular
2. - Los módulos son diseñados de un modo descendente
3. - Cada módulo se codifica utilizando las tres estructuras de control básicas:

a)Secuencia
b)Selección
c) Repetición

La programación de estructurada se refiere a un conjunto de técnicas que aumentan considerablemente la productividad del programa reduciendo en elevado grado el tiempo requerido para escribir, verificar, depurar y mantener los programas. Utiliza un número limitado de estructuras de control que minimizan la complejidad de los programas y por consiguiente reducen los errores y hacen los programas en general más eficientes.

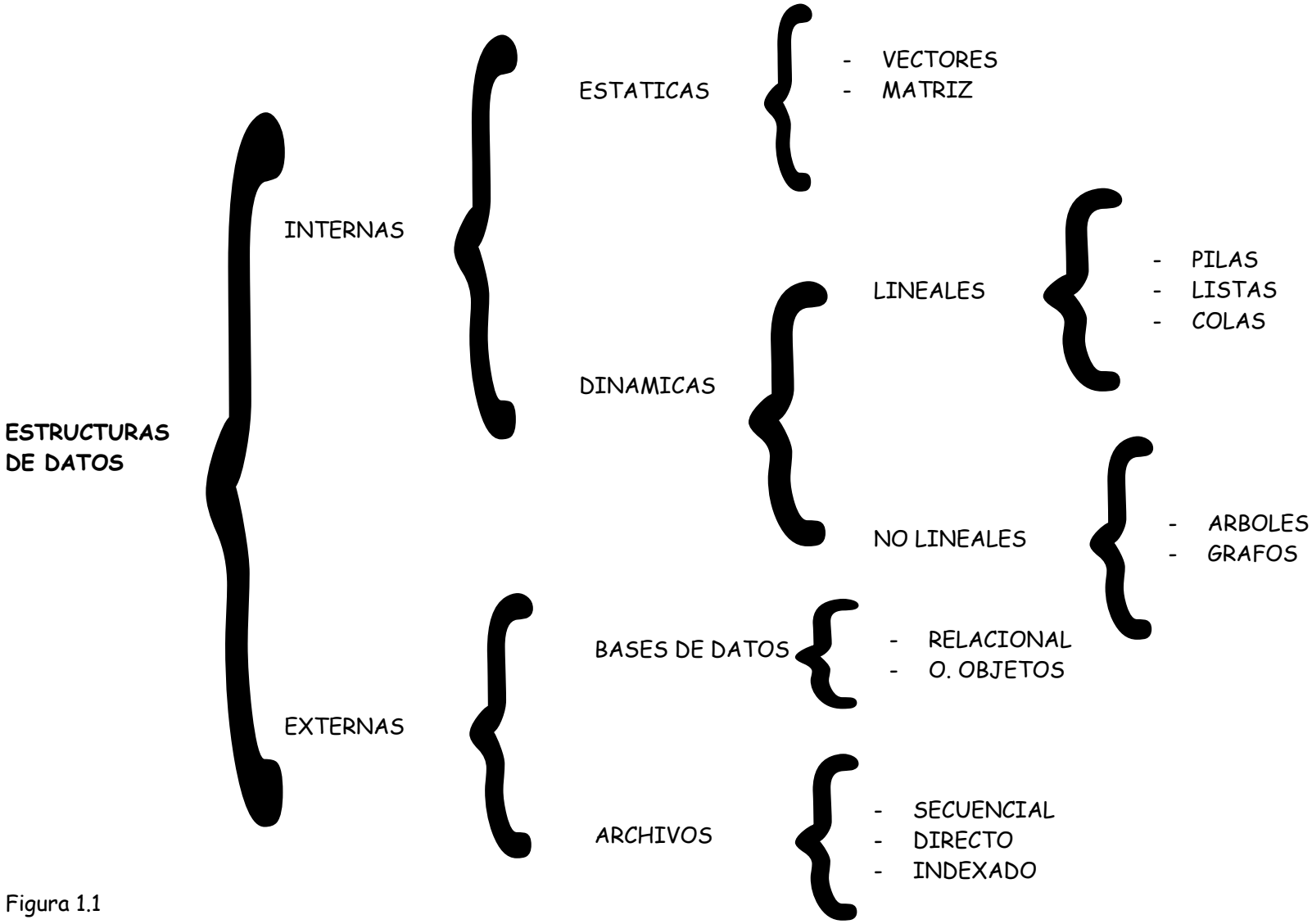

ESTRUCTURAS DE DATOS
INTERNAS
EXTERNAS
ESTATICAS
DINAMICAS
- VECTORES
- MATRIZ
LINEALES
NO LINEALES
- PILAS
- LISTAS
- COLAS
- ARBOLES
- GRAFOS
BASES DE DATOS
ARCHIVOS
- RELACIONAL
- O. OBJETOS
- SECUENCIAL
- DIRECTO
- INDEXADO

Figura 1.1

**Conceptos complementarios.**

**Abstracción**: Es un mecanismo para combatir la complejidad es decir, simplificar la realidad seleccionando de ellas solo las propiedades esenciales de los objetos relevantes para el problema en estudio. La abstracción ha sido un enfoque que reduce la complejidad. Modela en forma cercana a la realidad y facilita la construcción, validación y mantención de programas y hace factible la comprensión de la solución.

**Objeto**: Es la entidad relevante para modelar una realidad. Agrupar un conjunto de datos que son tomados como una unidad por ejemplo: Universidad: Estudiantes, cursos, salas, etc.

**IDEA BASICA DE LA ABSTRACCIÓN DE DATOS ES:**

*Tener un modelo* ⟷ *Después me preocupo de construcción.*

*Abstracción de datos* → *Objeto* → *Conducta.*

La abstracción de datos también tiene como beneficio la flexibilidad y ocultamiento de la información, la protección y por supuesto la reducción de la complejidad al descomponer en objetos o modelos.

De lo anterior se obtiene un nuevo concepto que es el de **Tipo de Datos Abstractos(TDA)**, que es un modelo matemático finito con varias operaciones definidas sobre él, el modelo matemático representa un nuevo tipo de datos conceptualmente más complejo de las cuales solo se conocen sus propiedades. Las operaciones que se pueden definir sobre tipo de datos abstractos son los procedimientos que afectan los cambios de estado del modelo el cual muestra un comportamiento particular. Esto es lo que se le es permitido al **TDA**.

| **TDA** |
|---|
| Propiedades |
| Operaciones(conducta) |

La abstracción de datos consiste en un conjunto de operaciones relacionados que actúan sobre una clase particular de objetos con la restricción de que la conducta de los objetos puede ser solo observado por las aplicaciones de esas operaciones.

Ejemplo 1: Definición del TDA conjunto.

Conjunto(modelo): Es una colección de elementos. Cada elemento puede ser un conjunto o un elemento inicial. Todos los miembros son distintos y no representan orden.

Operaciones: Unión, Intersección y Diferencia.

Ejemplo 2: Definición del TDA STACK.

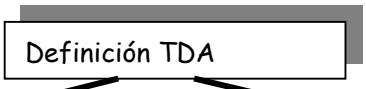
Modelo STACK: Lista lineal en que la inserción y eliminación de un elemento son realizadas por un extremo(LIFO: **L**AST **I**N **F**IRST **O**UT).

Operaciones: PUSH: Ingresa elementos al STACK.
POP: Elimina elementos del STACK.
EMPTY: Determina sí el STACK esta vacío.

Los TDA presentan dos aspectos que permiten conformar un objeto de esta naturaleza:

Generalización: Representan un desarrollo más avanzado con respecto a los tipos de datos primitivos(enteros, reales, etc.) los TDA son generalizaciones de los tipos de datos primitivos.

Encapsulación: Es posible localizar la definición del tipo y todas las operaciones asociadas en un solo lugar. Sólo una formulación y acceso están disponibles a través de procedimientos definidos.




Implementación del TDA Desarrollo en función del TDA.

La aplicación(Implementación) de un TDA en un lenguaje de programación se hace a través de las estructuras de datos definidas en función de los tipos de datos básicos y los operadores manejados por el lenguaje más las funciones y procedimientos necesarios. Una aplicación elige una estructura de datos para representar el objeto.

EJEMPLO: STACK.

```
TYPE  STACK: RECORD
        FIN: INTEGER
        VALOR: ARRAY[1. . 1000] OF INTEGER
```

Adicionalmente se puede mencionar que los TDA son la pieza central de la POO. Un TDA es un modelo que abarca un tipo y un conjunto de operaciones asociadas. Estas operaciones están definidas para el tipo y caracterizan su conducta.

### Elementos bases en el manejo de estructuras de datos

En los elementos que se cuentan para manejar las estructuras de datos se tiene los recursos básicos que son medios de alcenamiento de datos y la formación de estructuras de datos.

### 1) Recursos Básicos(Estructuras de datos fundamentales)

**A) Arreglo:** Un mecanismo de agregación más sencillo en la mayor parte de los lenguajes de programación es aveces el único medio ara crear estructuras de dato. Es una sucesión de celdas de un tipo de dato. Se define por:

NOMBRE_ARRAY: **ARRAY**[TIPO INDICE] **OF** TIPO_CELDA.

**B) Registros:** Mecanismo más general para mecanismos mas estructurados. Un registro es un conjunto de celdas llamados campos que pueden ser de distintos tipos. Los registros son a menudos agrupados en arreglos. Se define por:

```
TYPE NOMBRE: TIPO_RECORD= RECORD OF
                              NOMB_CAMPO1 TIPO CELDA;
                              NOMB_CAMPO2 TIPO CELDA
                          END
```

**C) Punteros:** Mecanismo para representar relaciones entre celdas de una estructura de datos. Un puntero es una celda cuyo valor indica o señala a otra cuando dibujamos estructuras de datos se indica el hecho que la celda "X", es un puntero a la celda "Y" dibujando una flecha desde "X" a "Y".

Ejemplo en PASCAL:

```
VAR
Puntero: ^TIPOCELDA
```

Gráfico:


NOTA: Estructuras de datos fundamentales son aquellas que son provistas por los lenguajes de programación alto nivel.

## 2) Formación de Estructuras de Datos

Los lenguajes de programación modernos poseen distintos y poderosas herramientas de estructuras de datos, se define una estructura de dato como un conjunto de celdas de memoria que presentan una organización particular, la cual posibilita el almacenamiento de información y una efectiva recuperación.

El elemento celda es la unidad básica para construir estructuras, se caracteriza como una caja capaz de almacenar un valor de algún tipo de dato ya sea elemental o compuesto. Las estructuras de datos se crean dando nombre a ciertas agrupaciones de celdas.

Una estructura de datos es una unidad compuesta de celdas y procedimientos específicos que mantiene durante la ejecución de un programa de datos relevantes para la aplicación, esta unidad no solo tiene por misión guardar datos, sino también entregar servicios de actualización, recuperación de información y mantenimiento de la organización.

Dentro de una estructura de datos encontramos tres componentes básicos para su operación estas son: Las celdas de información, un método de organización y algoritmos de servicio.

El método de una organización de estructura de datos, es el medio para relacionar las celdas que la componen. Entre estas tenemos las ya vistas: Arreglo, registro y punteros. Una estructura de datos en particular podría poseer una mezcla de estos medios por ejemplo: usar arreglos.

La construcción de una estructura de datos nace de la necesidad de implementar los modelos de datos específicamente los tipos de datos TDA y los de tipo de objeto. Los distintos objetos que las aplicaciones requieren son derivados de estas plantillas las que deben ser representadas con el uso de estructuras y algoritmos para resolver las problemáticas computacionales con eficacia y alto rendimiento. La idea básica de formar una estructura de datos no solo apunta a entregar un conjunto de celdas necesaria para almacenar la información relevante sino que además de suministrar los mecanismos necesaria de consulta y actualización de los datos mantenidos en la organización.

La organización de una estructura de datos: no es un proceso sencillo ni al azar la disposición de los datos de una aplicación y los algoritmos asociados para su acceso están internamente ligadas a la abstracción que se requiere representar y a la complejidad de la Implementación. Un tipo de datos o un objeto puede ser aplicado de distintas formas y con variados recursos computacionales sin que se pierda por este hecho la conducta de la abstracción. Este es un hito significativo e importante dentro de la metodología propuesta, puesto que conforma la independencia de las definiciones de alto nivel de la Implementación.

EJEMPLO: Matriz poco poblada.

Se define una matriz como poco poblada cuando tiene muchas entradas ceros.

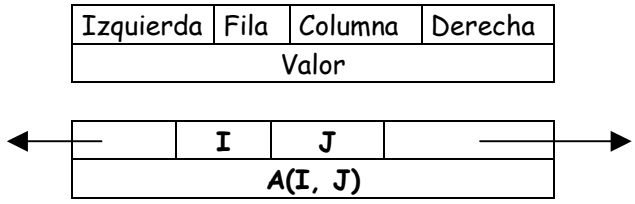
$$A = \begin{bmatrix} 4 & 0 & 0 & 5 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 1 & 0 & 2 \end{bmatrix}$$

Esta representación de la matriz no es eficiente para el manejo del recurso espacio(memoria). Se necesita usar una manera alternativa en que se almacenen solo los elementos no nulos par ellos cada elemento estará únicamente caracterizado por su posición en la columna y la fila. Luego se almacena la matriz como una secuencia de tríos, de la forma(I, J, DATO). Se puede agregar una tupla adicional en primer lugar con la cantidad de filas, columnas y datos de la matriz.

Ejemplo: (4, 4, 6)

Esta representación es útil cuando la información a almacenar no representa muchas actualizaciones y mantiene un numero estable elemento.

Otra manera de representar una matriz poco poblada sería usando listas enlazadas, En este caso cada elemento no nulo de la matriz se representaría por un nodo con la siguiente estructura.




**Aspectos adicionales al ejemplo 2(TDA PILA visto anteriormente)**

Especificaciones algebraicas semánticas de un TAD pila con las siguientes operaciones:

CREAPILA () ---> PILA
INTRODUCIR (PILA, ELEM) ---> PILA
DECAPITAR (PILA) ---> PILA
VACIA (PILA) ---> BOOLEAN

Para toda P de tipo Pila y E de tipo Elemento tenemos:

VACIA (CREAPILA(P)) ---> TRUE
VACIA (INTRODUCIR(P,E)) ---> FALSE
DECAPITAR (CREARPILA(P)) ---> ERROR
DECAPITAR (INTRODUCIR(P, E)) --->P

## Ejemplo completo para su estudio

Editor de ficheros secuenciales con cinco operaciones: Crear un nuevo fichero, insertar, reemplazar, eliminar, avanzar y retroceder, actuando siempre sobre el registro actual.

Para las especificaciones en lenguaje natural hay que describir el nombre de las operaciones que vamos a realizar, junto con las excepciones de dichas operaciones.

Operaciones:

- **Fichero_Nuevo (Fichero)**: Crea un fichero nuevo sin introducir ningún registro en el (número de registros es 0).
- **Insertar (Fichero, Registro)**: Inserta un registro después del registro actual y el nuevo pasa a ser el actual.
- **Reemplazar (Fichero, Registro)**: Cambia el registro actual por el nuevo.

**Eliminar (Fichero)**: Borra el registro actual y una vez eliminado el actual pasa a ser el siguiente.
**Avanzar (Fichero)**: El registro siguiente pasa a ser el actual.
**Retroceder (Fichero)**: El registro anterior pasa a ser el actual.

**CONDICION ACCION**
- Insertar Fichero vació Inserta en la primera posición
- Reemplazar Fichero vacío Nada
- Eliminar Fichero vacío Nada

Actual es el último Borra y actual será el anterior
- Avanzar Fichero vacío Nada
Actual es el último Nada
- Retroceder Fichero vacío Nada
Actual es el primero No hay registro actual

La descripción del TDA mediante un lenguaje de alto nivel no tiene porque tener en cuenta la eficiencia.

Definición de tipos:

```
FICHERO=ARRAY [1..N] OF INTEGER;
LONGITUD:INTEGER;
REG_ACTUAL:INTEGER;
```

Implementación de las operaciones:

```
Fichero_Nuevo (Fichero);
BEGIN
        LONGITUD:=0;
        REG_ACTUAL:=0;
END;

Insertar (Fichero, RegNuevo);
BEGIN
        FOR J:=LONGITUD DOWNTO REGACTUAL DO
        FICHERO [J+1]:=FICHERO[J];
        FICHERO[REGACTUAL+1]:=REGNUEVO;
        LONGITUD:=LONGITUD+1;
        REGACTUAL:=REGACTUAL+1
END;

Retroceder (Fichero);
BEGIN
        IF REGACTUAL<>0 THEN REGACTUAL:=REGACTUAL-1
END;

Eliminar (Fichero);
BEGIN
        IF REGACTUAL<>0
        THEN FOR J:=ACTUAL TO LONGITUD-1 DO
        FICHERO[J]:=FICHERO[J+1];
        IF REGACTUAL>LONGITUD THEN REGACTUAL:=REGACTUAL-1
END;

Avanzar (Fichero);
BEGIN
        IF REGACTUAL<>LONGITUD
        THEN REGACTUAL:=REGACTUAL+1
END;

Reemplazar (Fichero,RegNuevo);
BEGIN
        IF REGACTUAL<>0
        THEN FICHERO[REGACTUAL]:=REGNUEVO
END;
```

Especificaciones sintácticas:

FICHERO_NUEVO () ---> FICHERO
INSERTAR (FICHERO) ---> FICHERO
REEMPLAZAR (FICHERO,REGISTRO) ---> FICHERO
ELIMINAR (FICHERO) ---> FICHERO
AVANZAR (FICHERO) ---> FICHERO
RETROCEDER (FICHERO) ---> FICHERO

**Resumen**

En resumen, un TDA es un conjunto de valores y unas operaciones definidas sobre esos valores. Cada vez que deseemos emplear el TDA solo lo podemos hacer con las operaciones definidas, incluso no sabiendo como están implementadas

**Características de los TDA.**

- Cada modulo es una abstracción del problema, no conocemos el detalle.
- Se puede realizar una especificación breve y precisa del TDA.
- Con esta especificación se puede usar correctamente el TDA.
- Si realizamos, no se va a alterar considerablemente la especificación.
- Cada modulo TDA va a poder ser compilado por separado.
- Las comunicaciones entre los módulos van a ser pocas y claras.
- Cada modulo debe tener un tamaño razonable.

**Propiedades de los TDA.**

- Instanciabilidad: Vamos a poder declarar variables del tipo TDA, y estas van a tener las mismas propiedades y características que el TDA.
- Privacidad: No vamos a conocer detalladamente las operaciones ni la estructura que utiliza ese TDA.
- Componibilidad jerárquica: Un TDA puede estar formado por otros TDA mas básicos.
- Enriquecimiento: Podemos añadir operaciones al TDA.

# Unidad II: Estructuras de Datos Internas.

## 2.1 Identificación de características generales de las estructuras de datos internas.

### DEFINICIONES BÁSICAS

**Nodo:** Está formado por dos campos. Un campo de información que contiene el valor del elemento (INFO) y el campo NEXT que es el que apunta a la dirección del elemento Siguiente.

**Campo de información (INFO):** Contiene el elemento actual o valor del nodo en la lista.

**Campo de dirección (NEXT):** Contiene la dirección del nodo siguiente de la lista, si nodo fuera él ultimo de la lista, el campo NEXT apuntaría a **NIL**.


TROZO DE MEMORIA DE LA ESTRUCTURA QUE ESTAS VIENDOS.

**Dirección**: Numeración entregada pros el sistema operativo encargada de administrar el almacenamiento existente. A través de este podemos acceder a nuestro trozo de memoria.

## 2.2 CLASIFICACIÓN DE LAS ESTRUCTURAS DE DATOS

### ESTRUCTURAS DE DATOS ESTÁTICAS

| Simples o primitivas | Compuestas |
|---|---|
| a) Boolean<br>b) Char<br>c) Integer<br>d) Real | a) Arreglos<br>b) Conjuntos<br>c) Strings<br>d) Registros<br>e) Archivos |

### ESTRUCTURA DE DATOS DINAMICAS

| Lineales | No lineales |
|---|---|
| a) Pila<br>b) Cola<br>c) Lista | a) Árboles<br>b) Grafos |

## 2.3 Diferencias entre estructuras de datos estáticas y dinámicas.

**Estructuras de datos estáticas**

Son aquellas en las que se asigna una cantidad fija de memoria cuando se declara la variable.

En grandes ocasiones se necesitan colecciones de datos que crezcan y reduzcan su tamaño en memoria a medida que el programa progrese. Esto se logra implementando las estructuras dinámicas.

**Estructura de datos dinámicas**

Son aquellas cuya ocupación en memoria puede aumentar o disminuir en tiempo de ejecución

## 2.4 Variables de tipo puntero (Su Definición en PASCAL).

**Asignación estática y dinámica de memoria:**

Una variable tiene **asignación estática** de memoria cuando su tamaño se define en el momento de la compilación. (Ejemplo: A:INTEGER;)

Una variable tiene **asignación dinámica** de memoria cuando se define en la compilación, pero no ocupa memoria (no existe realmente) hasta la ejecución. (Ejemplo: X:INTEGER;).

**2.5 Operaciones con punteros.**

**PUNTERO**: Es una variable que almacena una dirección de memoria. Las **variables dinámicas** se definen y se accede a ellas a través de las variables de tipo puntero.

```
IMPLEMENTACION
TYPE
Puntero = ^Integer;
VAR
P, Q: PUNTERO;
BEGIN
NEW (P);
```

**NEW:** Este procedimiento asigna al puntero P, a través del parámetro, una dirección de memoria libre. En esta dirección es donde se almacena la **variable dinámica**.

```
DISPOSE(P);
```

**DISPOSE**: Este procedimiento libera al puntero P, a través del parámetro, una dirección de memoria libre. En esta dirección es donde se almacenaba la **variable dinámica**.

P^: Guarda el contenido de la dirección de memoria.

**NIL:** Indica que la dirección de memoria es nula.

**2.6 Modelos procesos de los punteros.**

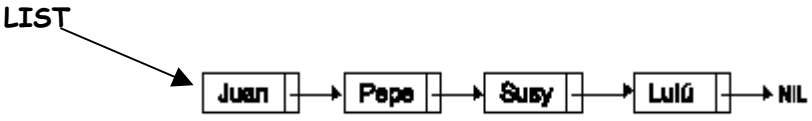
A continuación veremos como los elementos antes mencionados son aplicados, a través de un programa PASCAL.

**Ejemplo**:

```
TYPE Puntero = ^Integer;
VAR P, Q: puntero;
BEGIN
  NEW (P);
  NEW (Q);
  Q^:=7; Dirección de memoria de q guarda un 7.
  P^:=5; Dirección de memoria de p guarda un 5.
  P:=Q; Asignación de P, lo que tenga Q.
  WRITE (P^); Visualizo un 7.
  WRITE (Q^); Visualizo un 7.
END;
```

**2.7 Estructuras de datos: LISTAS**

**Lista lineal enlazada(encadenada):** Es cuando los elementos fueron ordenados explícitamente en nodos, es decir, cada elemento contiene dentro de sí mismo la dirección del elemento siguiente. Esta es una estructura dinámica, donde el número de nodos en una lista puede variar a medida que los elementos son insertados y removidos, por lo tanto la naturaleza dinámica de una lista contrasta con la naturaleza estática de un arreglo que permanece en forma constante.



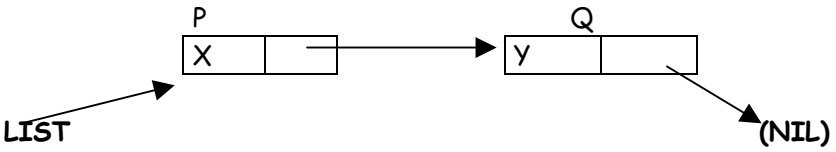

**LIST:** Es un puntero externo que apunta hacia el primer nodo de la lista.

**NIL(NULO):** Es el campo de dirección siguiente al último nodo en la lista, el cual es una dirección no válida. Este puntero nulo es utilizado para señalar el final de la lista.

**LISTA VACIA:** Es una lista sin nodos, es decir, el valor del puntero externo de la lista es el puntero nulo. Se puede especificar así: **LIST = NIL**

**2.8 Anatomía de un nodo.**

Sea




Se tiene

**INFO**(P)=X
**NEXT** (P)=Q
**INFO** (NEXT (P))=Y
**NEXT** (Q)=NIL

## 2.9 Operaciones más comunes en una lista lineal encadenada

**ALGORITMO PARA INSERCIÓN DE NODOS EN UNA LISTA LINEAL ENCADENADA**

```
P=GETNODE()
INFO(P)=X
NEXT (P)=P
LIST=P
```

En donde:

**GETNODE():** es un mecanismo en PSEUDOLENGUAJE para traer nodos vacíos.

**ALGORITMO PARA REMOVER DE NODOS EN UNA LISTA LINEAL ENCADENADA**

```
P=LIST
LIST=NEXT (P)
X=INFO (P)
FREENODE (P)
```

En donde:

**FREENODE():** es un mecanismo en PSEUDOLENGUAJE para eliminar nodos.

**LISTA DE DISPONIBILIDADES**

Si observamos claramente notamos que el algoritmo de eliminación no está concluido si no se agrega el mecanismo **FREENODE(P),** pues el nodo P eliminado de la lista encadenada todavía aparecería ahí, aunque ya no se encuentre dentro de la lista.

Anteriormente en el algoritmo para inserción existía un mecanismo denominado **GETNODE** del cual obtenemos nodos vacíos, y el mecanismo de **FREENODE** nos sirve para que guarde nodos vacíos pero eliminados de una lista encadenada, para cuando deseen ser utilizados.

Podrá pensarse: si **GETNODE** ofrece nodos vacíos, ¿Por qué son necesario nodos eliminados que ya fueron usados?. La realidad es que las computadoras tienen un límite en cuanto a memoria por lo tanto se deben utilizar los dos mecanismos de la siguiente forma:

Se utilizará una lista encadenada la cual se denominará lista de disponibilidades, en ésta lista se guardarán los nodos disponibles con el mecanismo **FREENODE** y con el mecanismo **GETNODE** se removerán de la lista de disponibilidades.

Por lo tanto, cuando ésta lista apunte a **NIL** pasará un error de desbordamiento superior (**OVERFLOW**), pues ya se llegó al límite de la capacidad de nodos (memoria disponible).

La lista de disponibilidades se representará de la siguiente manera con su campo INFO vacío.

**AVAIL = NIL**

**ALGORITMO PARA INSERCIÓN DE NODOS CONSIDERANDO LISTA DE DISPONIBILIDADES**

```
OPERACIÓN GETNODE
SI AVAIL = NIL entonces
   "error de OVERFLOW "
de lo contrario
   P = AVAIL
   AVAIL  = NEXT (AVAIL)
```

**ALGORITMO PARA INSERCIÓN DE NODOS ENTRE DOS NODOS DETERMINADOS**

**INSAFTER (P, X)**

```
INICIO
  Q= GETNODE ()
  INFO (Q)=X
  NEXT (Q)=NEXT (P)
  NEXT(P)=Q
FIN
```

Donde:

**INSAFTER(P, X):** Representa la operación de inserción de un elemento x en la lista después de que un nodo es apuntado por p.

AREA INFORMATICA - INACAP

**ALGORITMO PARA REMOVER NODOS CONSIDERANDO LISTA DE DISPONIBILIDADES**

```
DELAFTER (P, X)
        INICIO
          Q=NEXT (P)
          X=INFO(Q)
          NEXT (P)=NEXT (Q)
          FREENODE(Q)
        FIN
```

Donde:

**DELAFTER(P, X):** Significa un elemento "x" después de que un nodo es apuntado por P

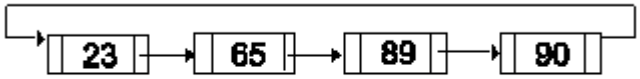
**BUSQUEDA SECUENCIAL ENCADENADA**

```
INICIO
   F = FALSE
   Q =NIL
   P = LIST
   MIENTRAS(p <> NIL) y (NOT f)HACER
    Sí (INFO (p)=llave) ENTONCES
      posición = p
      f = TRUE
    EN CASO CONTRARIO
       Q = P
       P = NEXT (P)
       Sí (NOT f)entonces
      posición = 0
   FIN SI
   FIN MIENTRAS
FIN
```

## 2.11 Lista circular encadenada

Aunque las listas lineales son muy útiles tiene sin Embargo algunas desventajas, la principal es que dado un nodo (P), no se puede alcanzar otro nodo que sea poseído por P.

Supongamos que hacemos un pequeño cambio a la estructura de la lista lineal de al manera que el campo **NEXT** en el último nodo contiene un puntero que va al primer nodo en lugar de apuntar a **NIL**.

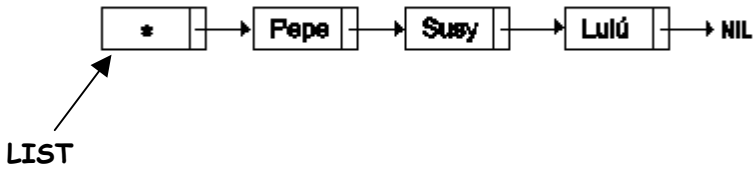



Esta lista tiene la característica que desde cualquier punto se puede ir hacia otro.

**NODOS DE ENCABEZAMIENTO**

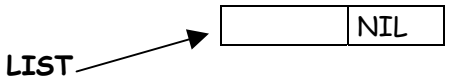
En algunas ocasiones es deseable mantener un nodo extra al frente de la lista, este nodo no representa un elemento en la lista y es llamado nodo de encabezamiento. Tendría sus dos campos, la porción **INFO** no es muy utilizado o solo para hacer referencia a algo especial.

Un nodo de encabezamiento sin información se representaría:




Si deseara que el campo INFO guardará alguna referencia.

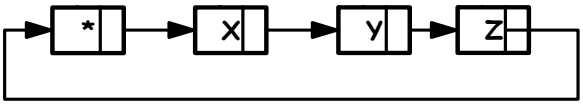
Hace referencia que la lista está formada por 3 elementos. Cuando una lista contiene un nodo de encabezamiento y está vacía se representa así:




Así se puede utilizar una lista como cola pues se maneja un puntero externo que apunta al primer nodo y en el campo **INFO** del nodo de encabezamiento tienen la dirección del último nodo evitando así los punteros **REAR(puntero al ultimo nodo insertado)** y **FRONT(puntero al primer nodo insertado)**.

**Lista circular con nodo de encabezamiento**

Supongamos que deseamos recorrer una lista circular, esto se puede hacer mediante la acción repetitiva de **p** = **NEXT**(p), donde p es inicialmente un puntero al comienzo de la lista, sin embargo como es lista circular no sabe cuando se había recorrido a menos que exista el puntero **LIST** y se efectúe la condición **p** = **LIST**, otro método de solución es el de agregar un nodo de encabezamiento, así la porción de **INFO** contendrá información de referencia, es decir no será similar a la información de nodos que preceden al nodo de encabezamiento, de ésta forma el puntero p se irá recorriendo hasta llegar al nodo de encabezamiento.




**2.14 Lista doblemente encadenada.**

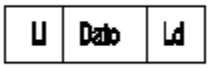
A pesar de que las listas lineales y circulares son bastante eficientes tiene dos desventajas:

1. **No se puede atravesar la lista circular o lineal en dirección contraria.**
2. **Es necesario buscar el nodo "p" dado para realizar alguna operación.**

Para eliminar estas desventajas existen las listas doblemente encadenadas, las cuales tienen las siguientes características:

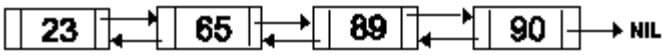
a) Como cualquier lista encadenada no vacía está formada por nodos, pero la configuración de estos es diferente.

Contiene tres campos, uno para la información y los dos restantes funcionan como apuntadores o bien, contienen la dirección de los dos nodos de ambos lados, esos dos campos se denominan **LI(Lado Izquierdo)** y **LD(Lado Derecho)**.

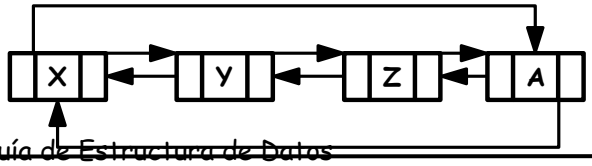



b) Las listas doblemente encadenadas se pueden presentar en tres formas:

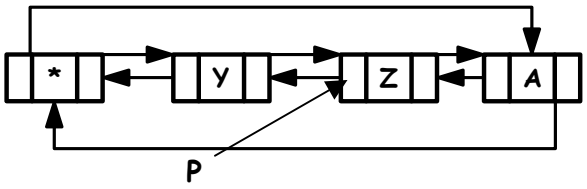
**1- LISTA LINEAL DOBLEMENTE ENCADENADA**




**2- LISTA CIRCULAR DOBLEMENTE ENCADENADA**



**3- LISTA CIRCULAR DOBLEMENTE ENCADENADA CON NODO DE ENCABEZAMIENTO**




**PROPIEDAD**

**LI (LD (P)) = P = LD (LI (P))**

**ALGORITMO PARA ELIMINAR NODOS EN UNA LISTA DOBLEMENTE ENCADENADA (CIRCULAR)**

```
INICIO
        Sí P=0 entonces
           error de UNDERFLOW
        EN CASO CONTRRIO
          x = INFO(P)
          Q = LI (P)
          R = LD (P)
          LD (Q)= R
          LI(R) = Q
          FREENODE(P)
        FIN SI
FIN
```

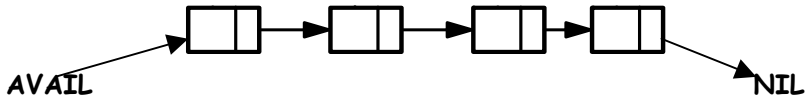
**ALGORITMO PARA INSERTAR NODOS EN UNA LISTA CIRCULAR DOBLEMENTE ENCADENADA**

```
INCIO
        Q = GETNODE
        INFO (Q)=X
        R = LD (P)
        LI(R) = Q
        LD (Q)=R
        LI (Q)=P
        LD (P)=Q
        NEXT (P)=LIST
        LIST=P
FIN
```

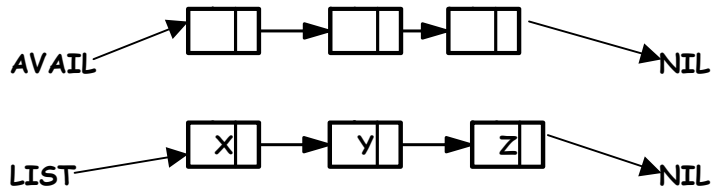
**ALGORITMO DE INSERCIÓN Y ELIMINACIÓN DE NODOS**

En la siguiente lista encadenada se agregará un nodo con información x, esto, utilizando la lista de disponibilidades.

Al iniciar las listas tendrían la siguiente estructura.




Insertando un nodo x quedarían como sigue:




## ANALISIS DE ALGORITMOS COMPLETOS EN SEUDO LENGUAJE ASOCIADOS A LOS DISTINTOS TIPOS DE LISTAS

| A. LISTAS SIMPLES (UN ENLACE) | B. LISTAS DOBLES |
|---|---|
| **1. ALGORITMO DE CREACIÓN**<br>INICIO<br>LIST = NIL<br>REPITE<br>MENSAJE ('INGRESE INFORMACION DE NODO?')<br>P = GETNODE()<br>LEER (INFO(P))<br>SÍ LIST == NIL ENTONCES<br>LIST = P<br>Q = P<br>NEXT(P) = NIL<br>EN CASO CONTRARIO<br>NEXT(Q) = P<br>NEXT(P) = NIL<br>Q = P<br>FIN SÍ<br>MENSAJE ('CONTINUA INGRESANDO NODOS?')<br>LEE (RESPUESTA)<br>HASTA RESPUESTA == NO<br>FIN | **1. ALGORITMO DE CREACIÓN**<br>INICIO<br>LIST = NIL<br>REPITE<br>MENSAJE ('INGRESE INFORMACION DE NODO?')<br>P = GETNODE()<br>LEER (INFO(P))<br>SÍ LIST == NIL ENTONCES<br>LD(P) = P<br>LI(P) = P<br>LIST = P<br>EN CASO CONTRARIO<br>R = LD(LIST)<br>LI (R) = P<br>LD(P) = R<br>LI(P) = LIST<br>LD(LIST) = P<br>FIN SÍ<br>MENSAJE('CONTINUA INGRESANDO NODOS?')<br>LEE (RESPUESTA)<br>HASTA RESPUESTA == NO |

**2. ALGORITMO PARA RECORRIDO DE UNA LISTA LINEAL**

```
INICIO
       P = LIST
       MIENTRAS P<>NIL HAZ
            ESCRIBE(INFO(P))
            P = NEXT(P)
       FIN MIENTRAS
FIN
```

**3. ALGORITMO PARA INSERTAR AL FINAL**

```
INICIO
       P = LIST
       MIENTRAS NEXT(P)<>NIL HAZ
            P = NEXT(P)
       Q = GETNODE()
       NEXT(P) = Q
       NEXT(Q) = NIL
FIN
```

**4. ALGORITMO PARA INSERTAR ANTES DE 'X' INFORMACIÓN**

```
INICIO
   P = LIST
   MIENTRAS P<>NIL HAZ
         SÍ INFO(P) == 'X' ENTONCES
              Q = GETNODE()
              LEER (INFO(Q))
              NEXT(Q) = P
              SÍ P==LIST ENTONCES
                   LIST = Q
              EN CASO CONTRARIO
                   NEXT(R) = Q
              P = NIL
         EN CASO CONTRARIO
              R = P
              P = NEXT(P)
         FIN SÍ
   FIN MIENTRAS
FIN
```

**5. ALGORITMO PARA BORRAR UN NODO**

```
INICIO
 P = LIST
FIN
```

**2. ALGORITMO PARA RECORRER LA LISTA DOBLE**

RECORRIDO A LA DERECHA.

```
INICIO
 P = LIST
 REPITE
   ESCRIBE(INFO(P))
   P = LD(P)
 HASTA P==LIST
FIN
```

**3. ALGORITMO PARA INSERTAR ANTES DE 'X' INFORMACIÓN**

```
INICIO
 P = LIST
 MENSAJE (ANTES DE ?)
 LEE(X)
 REPITE
   SÍ INFO(P) ==X  ENTONCES
         Q = GETNODE()
         LEER(INFO(Q))
         SÍ P=LIST ENTONCES
             LIST = Q
         LD(Q) = P
         LI(Q) = LI(P)
         LD(LI(P)) = Q
         LI(P) = Q
         P = LIST
   EN CASO CONTRARIO
         P = LD(P)
   FIN SÍ
 HASTA P == LIST
FIN
```

**4. ALGORITMO PARA BORRAR UN NODO**

```
INICIO
 P = LIST
 MENSAJE(INGRESE VALOR A BORRAR)
 LEE(VBORRAR)
 REPITE
    SÍ INFO(P) == VBORRAR  ENTONCES
         LD(LI(P)) = LD(P)
         LI(LD(P)) = LI(P)
```

```
 LEER (VBORRAR)
 MIENTRAS P<>NIL HAZ
     SÍ INFO(P) == VBORRAR ENTONCES
          SÍ P ==LIST ENTONCES
             SÍ NEXT(P) == NIL ENTONCES
                LIST = NIL
             EN CASO CONTRARIO
                LIST = LIST(NEXT)
          EN CASO CONTRARIO
             NEXT(Q)<--NEXT(P)
          FIN SÍ
          FREENODE(P)
          P = NIL
     EN CASO CONTRARIO
          Q = P
          P = NEXT(P)
      FIN SÍ
 FIN MIENTRAS
FIN
```

```
          SÍ P == LIST ENTONCES
             SÍ LD(P) == LI(P) ENTONCES
                LIST = NIL
             EN CASO CONTRARIO
                LIST = LD(LIST)
             FIN SÍ
             FREENODE(P)
             P = LIST
     EN CASO CONTRARIO
          P = LD(P)
     FIN SÍ
 HASTA P == LIST
FIN
```

## B. LISTAS CIRCULARES SIMPLES.

### 1. ALGORITMO DE CREACIÓN

```
INICIO
 REPITE
    P = GETNODE()
    MENSAJE (INGRESAR INFORMACIÓN DE NODO)
    LEE(INFO(P))
     SÍ LIST == NIL ENTONCES
      LIST = P
      Q = P
    EN CASO CONTRARIO
      NEXT(Q) = P
      Q = P
    FIN SÍ
    NEXT(P) = LIST
   MENSAJE (INGRESAR OTRO NODO)
   LEE(RESPUESTA)
 HASTA RESPUESTA == NO
FIN
```

### 2. ALGORITMO PARA RECORRER LA LISTA

```
INICIO
 P = LIST
 REPITE
    ESCRIBE(INFO(P))
```

```
    P = NEXT(P)
 HASTA P == LIST
FIN
```

**3. ALGORITMO PARA INSERTAR ANTES DE 'X' INFORMACIÓN**

```
INICIO
 GETNODE(P)
 LEE(INFO(P))
 SÍ LIST == NIL ENTONCES
    LIST = P
    NEXT(P) = LIST
 EN CASO CONTRARIO
    LEE(X)
    Q = LIST
     R = LIST(NEXT)
     REPITE
       SÍ INFO(Q) == X ENTONCES
             NEXT(P) = Q
             NEXT(R) = P
             SÍ NEXT(P) == LIST ENTONCES
                   LIST= P
       FIN SÍ
       Q = NEXT(Q)
       R = NEXT(R)
     HASTA Q == LIST
  FIN SÍ
FIN
```

**4. ALGORITMO PARA BORRAR**

```
INICIO
 MENSAJE(VALOR A BORRAR)
 LEE(VBORRAR)
 Q = LIST
 R = LIST
 P = LIST
 MIENTRAS NEXT(Q)<>LIST HAZ
      Q = NEXT(Q)
  REPITE
    SÍ INFO(P) == VBORRAR ENTONCES
     SÍ P == LIST ENTONCES
          SI LIST(NEXT) == LIST ENTONCES
                LIST = NIL
          EN CASO CONTRARIO
                LIST = NEXT(LIST)
                NEXT(Q) = LIST
```

```
            FIN SÍ
      EN CASO CONTRARIO
            NEXT(R) = NEXT(P)
      FIN SÍ
      FREENODE(P)
      P = LIST
    EN CASO CONTRARIO
      R = P
      P = NEXT(P)
     FIN SÍ
  HASTA P == LIST
FIN
```

**IMPLEMENTACION EN LENGUAJE PASCAL**

A continuación presentaremos algoritmos desarrollados en lenguaje PASCAL, de algunos programas analizados anteriormente.

Definición de nodo en PASCAL.

```
TYPE
        PUNTERO=^NODO;
        NODO=RECORD
        INFO:...;
        SIG:PUNTERO;
        END;
        VAR
        LISTA,AUX:PUNTERO;
```

**LISTA**: Puntero comienzo, me garantiza la lista completa.
**AUX**: Puntero auxiliar

**A. OPERACIONES CON LISTAS ENLAZADAS LINEALES**

**CREACION:** Estructura dinámica.

```
NEW (LISTA);            Lista es el primer NODO.
READ (LISTA^.INFO);     Aceptar el campo INFO del NODO.
LISTA^.SIG:=NIL;        Meto NIL en SIG, para saber que es el último NODO.
```

Todos los NODOS son iguales, menos el 1º (**jamás debe tocarse el 1º** ).
Hay 2 pasos, creación del primer NODO y el Resto.

**CREACION CON N NODOS:**

```
BEGIN
        NEW (LISTA);
        READ (LISTA^.INFO);
        AUX:=LISTA;
```

```
        FOR I:=1 TO N DO
        BEGIN
                NEW (AUX^.SIG);
                AUX:=AUX^.SIG;
                READ (AUX^.INFO);
        END;
        AUX^.SIG:=NIL;
END.
```

**RECORRIDO EN UNA LISTA ENLAZADA:**

Me sitúo en el primer NODO, con el puntero auxiliar y hasta el último no termina.

```
BEGIN
 AUX:=LISTA;
 WHILE AUX <> NIL DO
 BEGIN
        WRITE (AUX^.INFO);
        AUX:=AUX^.SIG;
 END;
END.
```

**BUSQUEDA DE UN ELEMENTO EN UNA LISTA ENLAZADA:**

Realizar un PROCEDURE que reciba como parámetro el puntero comienzo de una lista enlazada, un elemento del mismo tipo y un puntero sobre el que se devuelve la dirección de memoria del NODO que contiene dicho elemento si es que esta y NIL si no está.

```
PROCEDURE BUSQUEDA(COMIENZO:PUNTERO,ELEM:INTEGER,VAR POS:PUNTERO)
VAR
AUX:PUNTERO; ENC:BOOLEAN;
BEGIN
        AUX:=COMIENZO; ENC:=FALSE;
        WHILE (NOT ENC) AND (AUX<>NIL) DO
        BEGIN
                IF AUX^.INFO=ELEM THEN ENC:=TRUE
                ELSE AUX:=AUX^.SIG;
        END;
        POS:=AUX; Me da la dirección del NODO que busco.
END.
```

**INSERCION DE UN NODO EN UNA LISTA ENLAZADA:**

**Por el PRINCIPIO de la Lista:**

```
Inserta el elemento ELEM, en el principio de la lista.
NEW (AUX); Pedir memoria.
AUX^.INFO:=ELEM; Guardar el elemento.
AUX^.SIG:=LISTA; Engancharlo.
LISTA:=AUX; Cambiar lista al primero.
```

**Por el FINAL de la Lista:**

Inserta el Elemento Elem en el final de la lista.(otra forma en pag. sig.)

```
NEW (AUX);
AUX^.INFO:=ELEM;
AUX^.SIG:=NIL;
P:=LISTA;
WHILE P^.SIG <> NIL DO
BEGIN
  P:=P^.SIG;
END;
P^.SIG:=AUX;
```

**BORRADO DE UN ELEMENTO EN UNA LISTA ENLAZADA:**

**DISPOSE:** Libera memoria, al revés del NEW. La dirección que tiene el puntero que se le pasa como parámetro pasa a ser una dirección libre.

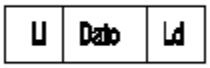
**Si la lista no tiene elementos repetidos:**

Eliminar de la lista el NODO que contiene el elemento 3.

```
IF LISTA^.INFO=ELEM THEN
BEGIN
        AUX:=LISTA;
        LISTA:=LISTA^.SIG;.
        DISPOSE (AUX);
END
ELSE
BEGIN
        ANT:=LISTA;
        AUX:=LISTA^.SIG;
        ENC:=FALSE;
END;
WHILE (NOT ENC) AND (AUX< >NIL) DO
BEGIN
        IF AUX^.INFO=ELEM THEN
        ENC:=TRUE
        ELSE
        BEGIN
        ANT:=AUX;
        AUX:=AUX^.SIG;
        END;
END;
IF ENC THEN
BEGIN
        ANT^.SIG:=AUX^.SIG;
        DISPOSE (AUX);
```

AREA INFORMATICA - INACAP
END;

**B. LISTAS DOBLES O LISTAS DOBLEMENTE ENLAZADAS:**
Es una lista que está enlazada en dos sentidos en la que cada nodo contiene la dirección del anterior y del siguiente.




**IMPLEMENTACION:**
**TYPE**
PUNTERO=^NODO;
NODO=RECORD
INFO: ...;
ANT,SIG: PUNTERO;
END;
**VAR**
Comienzo, fin: puntero;

**CREACIÓN DE UNA LISTA DOBLE CON N NODOS:**
```
BEGIN
 NEW (COMIENZO);
 READ (COMIENZO^.INFO);
 COMIENZO^.ANT:=NIL;
 FIN:=COMIENZO;
 FOR I:=1 TO N DO
 BEGIN
        NEW(FIN^.SIG);
        FIN^. SIG^.ANT:=FIN;
        FIN:=FIN^.SIG
        READ (FIN^.INFO);
 END;
 FIN^.SIG:=NIL;
END;
```

**INSERTAR UN NODO DESPUES DEL NODO QUE APUNTA P:**
```
BEGIN
        NEW (AUX);
        AUX^.INFO:=ELEM;
        AUX^.SIG:=P^.SIG;
        AUX^.ANT:=P;
        P^.SIG:=AUX;
        AUX^. SIG^.ANT:=AUX;
END;
```

## 2.19 Estructura de datos: PILAS

Es una colección ordenada de elementos en la cual en un extremo se pueden insertar nuevos elementos y de la cual se pueden retirar otros, llamada parte superior de la pila.

Estructura de datos lineal de elementos homogéneos, en la cual los elementos entran y salen por un mismo extremo, llamado **tope, cabeza** o **cima de la pila (O STACKPOINTER)**.

Las pilas son otro tipo de estructura de datos lineales, las cuales presentan restricciones en cuanto a la posición en la cual pueden realizarse las inserciones y las extracciones de elementos.

Una pila es una lista de elementos en la que se pueden insertar y eliminar elementos sólo por uno de los extremos. Como consecuencia, los elementos de una pila serán eliminados en orden inverso al que se insertaron. Es decir, el último elemento que se metió a la pila será el primero en salir de ella.

En la vida cotidiana existen muchos ejemplos de pilas, una pila de platos en una alacena, una pila de latas en un supermercado, una pila de papeles sobre un escritorio, etc.

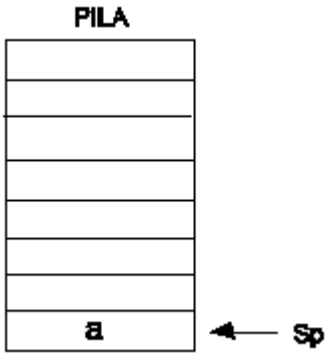
Debido al orden en que se insertan y eliminan los elementos en una pila, también se le conoce como estructura **LIFO** (**L**ast **I**n, **F**irst **O**ut: último en entrar, primero en salir).

### REPRESENTACIÓN EN MEMORIA

Las pilas no son estructuras de datos fundamentales, es decir, no están definidas como tales en los lenguajes de programación. Las pilas pueden representarse mediante el uso de:

- Arreglos.
- Listas enlazadas.

Nosotros ahora usaremos los arreglos. Por lo tanto debemos definir el tamaño máximo de la pila, además de un apuntador al último elemento insertado en la pila el cual denominaremos SP. La representación gráfica de una pila es la siguiente:

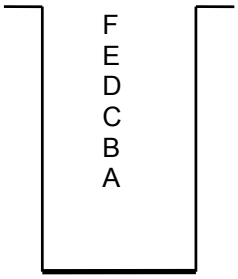



Como utilizamos arreglos para implementar pilas, tenemos la limitante de espacio de memoria reservada. Una vez establecido un máximo de capacidad para la pila, ya no es posible insertar más elementos.

Una posible solución a este problema es el uso de espacios compartidos de memoria. Supóngase que se necesitan dos pilas, cada una con un tamaño máximo de n elementos. En este caso se definirá un solo arreglo de 2*n elementos, en lugar que dos arreglos de n elementos.

En este caso utilizaremos dos apuntadores: **SP1** para apuntar al último elemento insertado en la pila 1 y **SP2** para apuntar al último elemento insertado en la pila 2. Cada una de las pilas insertará sus elementos por los extremos opuestos, es decir, la pila 1 iniciará a partir de la localidad 1 del arreglo y la pila 2 iniciará en la localidad 2n. De este modo si la pila 1 necesita más de n espacios (hay que recordar que a cada pila se le asignaron n localidades) y la pila 2 no tiene ocupados sus n lugares, entonces se podrán seguir insertando elementos en la pila 1 sin caer en un error de desbordamiento.

**Diagrama: PILA DE SEIS ELEMENTOS**




El elemento más alto es f, d es más alto que c, b y a, pero menor que e y f. Una pila se utiliza para cálculos o para ejecución de instrucciones, puede implantarse por hardware de diseño especial o en memoria controladas por programas.

Según la definición de una pila un solo extremo de la pila puede designarse como parte superior, los nuevos elementos que se insertan deben colocarse en la parte superior de la pila, en éste caso la pila se mueve hacia arriba o bien los elementos pueden ser removidos o

eliminados en cuyo caso la pila se mueve hacia abajo; para designar la parte de arribo o de abajo se debe indicar cual es la parte superior; **PUSH**(agregar), **POP**(eliminar).

**Operaciones básicas de las pilas**

Las principales operaciones que podemos realizar en una pila son:

- Insertar un elemento (**PUSH**).
- Eliminar un elemento (**POP**).

Planteado como un TDA seria:

**PUSH(S, X)** = Insertar un elemento x en la pila S(**STACK**).
**POP(S)** = Remover un elemento de la pila.

Los algoritmos para realizar cada una de estas operaciones se muestran a continuación apoyado en el manejo de arreglos para manipular los elementos de la pila. La variable **MÁXIMO** para hacer referencia al máximo número de elementos en la pila.

**Operación de inserción (PUSH)**

```
INICIO
        SÍ SP == MÁXIMO ENTONCES
            MENSAJE (OVERFLOW)
        EN CASO CONTRARIO
            SP =  SP + 1
            PILA[SP] =  VALOR
        FIN SÍ
FIN
```

**Operación de eliminación (POP)**

```
INICIO
        SÍ SP == 0 ENTONCES
            MENSAJE (UNDERFLOW)
        EN CASO CONTRARIO
            X = PILA[SP]
            SP =  SP - 1
        FIN SÍ
FIN
```

**Operación pila vacía**

Una pila se denomina vacía cuando ésta contiene un solo elemento, y el elemento es retirado de la pila, por lo tanto antes de retirar un elemento es necesario asegurarse que la pila no esté vacía, esto será con la operación "**EMPTY**(s) ", si ésta regresa el valor de verdadero la pila está vacía de lo contrario retorna un valor de falso.

Otra operación será determinar el elemento superior de la pila sin retirarlo, esto será con la operación "**STACKTOP**(s)".

Es necesario utilizar siempre **EMPTY** ya que debemos asegurarnos que la pila está vacía de lo contrario si pedimos que nos diga cual es el elemento superior (**STACKTOP**) nos marcaría un error de **UNDERFLOW**.

**Operación CIMA**

Esta es una operación la cual retorna el valor que se encuentra en el tope de la una pila.

**ALGORITMOS ASOCIADOS A LAS PILAS**

Hasta ahora para el movimiento de las pilas solamente se han visto las operaciones **PUSH** y **POP** pero se ha tomado en cuenta la forma de representar una pila correctamente, como son contadores para el valor de **SP** o bien se han visto pilas de tamaño indefinido.

La pila se ha visto como arreglo pero desgraciadamente son diferentes, pues el número de elementos en un arreglo es fijo y en una pila es dinámico cuyo tamaño va cambiando a medida que los elementos son insertados y removidos, sin embargo aunque un arreglo no puede ser una pila éste si puede ser un recipiente de ella, en donde tendrá las siguientes características:

1. Un extremo del arreglo será el fondo fijo de la pila
2. Una parte superior estará cambiando constantemente
3. Un campo adicional, el cual durante la ejecución en cualquier momento llevará registro de la parte superior de la pila

Se tomará en cuenta que la pila tendrá un límite en cuanto a tamaño; si por ejemplo la pila puede tener de 0 a 100 y el valor de **SP** igual a 100 la pila estará llena, en cambio si **SP**=0 la pila estará vacía.

**ALGORTIMO PARA RETIRO O SUPRESION DE ELEMENTOS**

```
INICIO
        SI EMPTY(S) == VERDADERO ENTONCES
            ERROR DE UNDERFLOW
        DE LO CONTRARIO
            POP (S)
            DATO = PILA (SP)
            SP = SP - 1
        FIN SÍ
FIN
```

**ALGORTIMO PARA INSERCION O EMPUJE DE ELEMENTOS**

```
INICIO
        SI SP == LÍMITE ENTONCES
            ERROR DE OVERFLOW
```

```
AREA INFORMATICA - INACAP
          DE LO CONTRARIO
              SP = SP+1
              PUSH (S)
              PILA (SP) = DATO
          FIN SÍ
FIN
```

**UTILIZACIÓN DE PILAS**

Considérese la suma A y B, pensando que la aplicación del operador "+" a los operando A y B, se escribirá A+B, esta representación se denomina entrefijo.

Existen otras dos formas de representar la suma de A + B utilizando el signo "+", estas son el **PREFIJO** y el **POSTFIJO**. Los prefijos **PRE** y **POST** se refieren a la posición relativa del operador con respecto a los operando.

En la notación de **PREFIJO** el operador precede a los dos operando y en la notación **POSTFIJO** el operador va después de los dos operando.

Consideremos la expresión a + b * c escrita en entrefijo, observe que existen dos operadores, se quiere saber cual de estos se realizará primero, es lógico que por jerarquía se realizaría primero la multiplicación y posteriormente la suma.

La única regla durante el proceso de conversión es que las operaciones que tienen relación o jerarquía más alta se convertirá primero y después de que esta parte de la expresión ha sido convertida se maneja como operando simple.

El orden de precedencia o jerarquía de los operadores son: primero el exponente, posteriormente multiplicación y división y por último la suma y la resta, Cuando dos operadores tienen la misma precedencia, se sigue el orden de izquierda a derecha, a excepción de la exponenciación donde se asume un orden de derecha a izquierda.

**REGLAS**

1. Dar jerarquías o precedencias sin afectar a la expresión
2. De acuerdo a las jerarquías se inicia las conversiones
3. Las conversiones se harán de la siguiente manera:

a) El orden de los operando no cambia, solo el de los operadores.
b) Los paréntesis que aparezcan la expresión original, no aparecerán en las conversiones.
c) El operador irá precedido de los operando y operadores que afecten, esto en el caso del **PREFIJO**. En forma contraria funcionará el **POSTFIJO**.

En expresiones de **POSTFIJO** no existen los paréntesis, pero el orden en que se presentan los operadores determinan el orden actual de las operaciones al evaluar la expresión, haciendo en este caso que los paréntesis no sean necesarios.

Al hacer las expresiones de entrefijo a **POSTFIJO** a simple vista perdemos la habilidad de observar a los operando asociados con un operador, por lo tanto la notación **POSTFIJO** de la expresión original puede parecer simple si no fuera por el hecho que parece difícil de evaluar; la solución es la aplicación de un algoritmo para evaluar expresiones en **POSTFIJO**.

**ALGORITMO PARA EVALUAR EXPRESIONES EN POSTFIJO**

```
INICIO
  MIENTRAS HAY MÁS CARACTERES EN LA HILERA HAZ
      LEE SYMB
     SÍ SYMB = OPERANDOS ENTONCES
         PUSH (OPNDSTK, SYMB)
     DE LO CONTRARIO
        OPND2 = POP (OPNDSTK)
        OPND1 = POP (OPNDSTK)
        VALOR=RESULTADO DE APLICAR
       SYMB A OPND1 SOBRE OPND2
      PUSH(OPNDSTK,VALOR)
     FIN SÍ
  FIN MIENTRAS
  RESULTADO = POP(OPNDSTK)
FIN
```

Donde:

OPNDSTK=PILA DE OPERANDOS
OPND2= PRIMER OPERANDO RETIRADO
OPND1=SEGUNDO OPERANDO RETIRADO
SYMB=SIGUIENTE SIMBOLO A LEER

**UTILIZACIÓN DE PILA CON LISTAS ENLAZADAS**

Se observan las listas encadenadas que de la parte del puntero externo (**LIST**) se accede a la lista, es decir, de ahí mismo sé accesa y remueve nodos, tienen las mismas características, puesto que se insertan y se remueven elementos de un solo lugar.

Una pila puede ser accesada a través de su elemento superior y una lista puede ser accesada a través del puntero al primer elemento. De esta forma se ha descubierto la forma de implementar una pila. Esta puede estar representada por una lista lineal encadenada donde el primer nodo de la lista es el elemento superior de la pila.

**ALGORITMO PARA EMPUJAR NODOS EN UNA PILA ENCADENADA (PUSH)**

```
INICIO
 P = GETNODE ()
 INFO (P) = X
 NEXT (P) = S
 S=P
FIN
```

**ALGORITMO PARA ELIMINAR NODOS EN UNA PILA ENCADENADA (POP).**

```
INICIO
 SÍ EMPTY (S) == TRUE ENTONCES
    "ERROR DE UNDERFLOW"
 DE LO CONTRARIO
    P=S
    S = NEXT (P)
    X = INFO (P)
    FREENODE(P)
  FIN SÍ
FIN
```

**2.21 Implementacion de operaciones de pila en lenguaje PASCAL.**

A continuación presentaremos algoritmos desarrollados en lenguaje PASCAL, de algunos programas analizados anteriormente.

**A. Definición de pila en PASCAL.**

| ESTATICA O SECUENCIAL | DINAMICA O ENLAZADA |
|---|---|
| TYPE<br>TIPOPILA = RECORD<br>DATOS : ARRAY [ 1.. Max] OF ..... ;<br>CAB: 0 .. Max;<br>END;<br>VAR<br>PILA1, PILA2 : TIPOPILA; | TYPE<br>TIPOPILA = ^ NODO<br>NODO = RECORD<br>INFO : ..... ;<br>SIG : TIPOPILA;<br>END;<br>VAR<br>PILA1,PILA2 : TIPOPILA ; |

**B. OPERACIONES CON PILAS.**

A continuación vamos a ver los algoritmos de las operaciones más comunes que se realizan con pilas:

**1. LIMPIAR PILA.**

En ambos casos (tanto en la estática como en la dinámica) será un procedimiento en el que se pasa como parámetro una pila.

| ESTÁTICA | DINÁMICA |
|---|---|
| PROCEDURE LIMPIA_PILA (VAR PILA1: TIPOPILA);<br>BEGIN<br>PILA1. CAB := 0;<br>END; | PROCEDURE LIMPIA_PILA (VAR PILA1: TIPOPILA);<br>BEGIN<br>PILA1:= NIL;<br>END; |

**2. FUNCION PILA VACIA.**

Vamos a hacer una función BOOLEAN que recibe una pila como parámetro y devuelve T si está vacía o F si no lo está.

| ESTÁTICA | DINÁMICA |
|---|---|
| FUNCTION PILA_VACIA (VAR PILA1: TIPOPILA): BOOLEAN;<br>BEGIN<br>PILA_VACIA := PILA1.CAB = 0;<br>END; | FUNCTION PILA_VACIA (VAR PILA1: TIPOPILA): BOOLEAN;<br>BEGIN<br>PILA_VACIA := PILA1 = NIL;<br>END; |

**3. FUNCION PILA LLENA.**

Vamos a hacer una función BOOLEAN que recibe una pila como parámetro y devuelve T si está vacía o F si no lo está.

| ESTÁTICA | DINÁMICA |
|---|---|
| FUNCTION PILA_LLENA (VAR PILA1: TIPOPILA): BOOLEAN;<br>BEGIN<br>PILA_LLENA := PILA1.CAB = MAX ;<br>END; | *** En la estructura dinámica nunca estará llena.** |

**4. FUNCION CIMA**

| ESTÁTICA | DINÁMICA |
|---|---|
| FUNCTION CIMA (VAR PILA1: TIPOPILA): INTEGER;<br>BEGIN<br>CIMA := PILA1.DATOS[PILA1. CAB]<br>END; | FUNCTION CIMA (VAR PILA1: TIPOPILA): INTEGER;<br>BEGIN<br>CIMA := AUX ^. INFO<br>END; |

**5. INSERTAR UN ELEMENTO EN UNA PILA.**

| ESTÁTICA | DINÁMICA |
|---|---|
| PROCEDURE INSERTAR(VAR PILA1: TIPOPILA ; ELEM : ....) ;<br>BEGIN | PROCEDURE INSERTAR(VAR PILA1: TIPOPILA ; ELEM : ....) ; VAR AUX : TIPOPILA ; |

```
 IF PILA_LLENA ( PILA1 ) = FALSE THEN
  BEGIN
    PILA1. CAB := PILA1. CAB+1;
    PILA1.DATOS[PILA1. CAB] := ELEM;
  END;
END;
```

```
BEGIN
  NEW ( AUX ) ;
  AUX ^. INFO := ELEM;
  AUX ^. SIG := PILA1;
  PILA1:= AUX ;
END;
```

**6. EXTRAER UN ELEMENTO DE UNA PILA.**

**ESTÁTICA**

```
PROCEDURE SACAR (VAR PILA1: TIPOPILA
; ELEM : ....) ;
BEGIN
  IF NOT ( PILA_VACIA ( PILA1 )) THEN
   BEGIN
     ELEM := PILA1. DATOS [PILA1. CAB];
     PILA1 . CAB:= PILA1. CAB-1;
  END;
END;
```

**DINÁMICA**

```
PROCEDURE SACAR (VAR PILA1: TIPOPILA
; ELEM : ....) ;
VAR AUX : TIPOPILA ;
BEGIN
  IF NOT ( PILA_VACIA ( PILA1 )) THEN
  BEGIN
    ELEM := PILA1.INFO;
    AUX := PILA 1 ;
    PILA1 := PILA1^. SIG;
    DISPOSE (AUX);
  END;
END;
```

## 2.22 Estructura de datos: Colas

**Introducción**

Es una colección ordenada de elementos, a partir de la cual se pueden eliminar elementos de un extremo llamado FRENTE DE LA COLA y en la cual se pueden agregar nuevos elementos en el otro extremo llamado PARTE POSTERIOR O ATRÁS.

Una cola es una estructura de almacenamiento, donde la podemos considerar como una lista de elementos, en la que éstos van a ser insertados por un extremo y serán extraídos por otro.

Las colas son estructuras de tipo **FIFO** (**FIRST-IN**, **FIRST-OUT**), ya que el primer elemento en entrar a la cola será el primero en salir de ella.

Existen muchos ejemplos de colas en la vida real, como por ejemplo: personas esperando en un teléfono público, niños esperando para subir a un juego mecánico, estudiantes esperando para subir a un camión escolar, etc.

Las colas además se utilizan en la computadora como un espacio de almacenamiento reservado temporalmente para contener información ya sea en la memoria o en sistemas operativos.

Entonces, la función de la cola se define como una línea de espera, utilizando la técnica administración **FIFO**.

**REPRESENTACIÓN EN MEMORIA.**

Podemos representar a las colas de dos formas:

- Como arreglos
- Como listas ordenadas

En esta unidad trataremos a las colas como arreglos de elementos, en donde debemos definir el tamaño de la cola y dos apuntadores, uno para accesar el primer elemento de la lista y otro que guarde el último. En lo sucesivo, al apuntador del primer elemento lo llamaremos F(**QFRONT**), al del último elemento A(**QREAR**) y **MAXIMO** para definir el número máximo de elementos en la cola.

## 2.23 OPERACIONES BÁSICAS QUE PUEDEN SER APLICADAS A LAS COLAS:

1. **INSERT (Q, X)**
   La cual adiciona un elemento "x" en la parte posterior de la cola.

2. **X = REMOVE (Q)**
   La cual retira un elemento del frente de la cola y coloca en "x" su contenido.

3. **EMPTY (q)**
   La cual retorna el valor de falso o verdadero dependiendo si la cola contiene algún elemento o está vacía.

Se utilizará un arreglo para que este contenga los elementos de la cola, por la tanto al estar insertando elementos existe la posibilidad de que se presente un sobre flujo si la cola ya está llena.

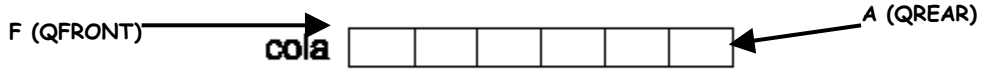
## FORMULA PARA DETERMINAR EL NÚMERO DE ELEMENTOS

**#ELEMENTOS** = (QREAR - QFRONT) + 1

Donde: QFRONT: Se le conocerá como índice al primer elemento.
QREAR: Se le conocerá como índice al último elemento.

## COLA LINEAL

La cola lineal es un tipo de almacenamiento creado por el usuario que trabaja bajo la técnica FIFO (primero en entrar primero en salir). Las colas lineales se representan gráficamente de la siguiente manera:




Las operaciones que podemos realizar en una cola son las de inicialización, inserción y extracción. Los algoritmos para llevar a cabo dichas operaciones se especifican más adelante y se considerara las estructuras de datos arreglos para manipular colas.

Las condiciones a considerar en el tratamiento de colas lineales son las siguientes:

- **OVERFLOW** (cola llena), cuando se realice una inserción.
- **UNDERFLOW**(cola vacía), cuando se requiera de una extracción en la cola.
- **VACÍO**

| **ALGORITMOS PARA MANEJAR COLAS LINEALES** | |
|---|---|
| **INICIALIZACIÓN**<br>INICIO<br>F = 0<br>A = 0<br>FIN | |
| **PARA INSERTAR ELEMENTOS** | **PARA EXTRAER ELEMENTOS** |
| INICIO<br>SÍ A == MÁXIMO ENTONCES<br>MENSAJE (OVERFLOW)<br>EN CASO CONTRARIO<br>A = A + 1<br>COLA[A] = VALOR<br>FIN SÍ<br>FIN | INICIO<br>SI A < F ENTONCES<br>MENSAJE (UNDERFLOW)<br>EN CASO CONTRARIO<br>X = COLA[F]<br>F = F + 1<br>FIN SÍ<br>FIN |

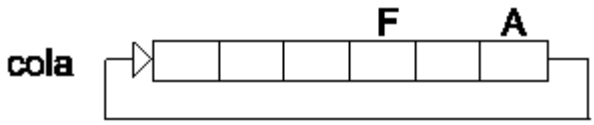
**COLA CIRCULAR**

Las colas lineales tienen un grave problema, como las extracciones sólo pueden realizarse por un extremo, puede llegar un momento en que el apuntador **A** sea igual al máximo número de elementos en la cola, siendo que al frente de la misma existan lugares vacíos, y al insertar un nuevo elemento nos mandará un error de **OVERFLOW** (cola llena).

Para solucionar el problema de desperdicio de memoria se implementaron las colas circulares, en las cuales existe un apuntador desde el último elemento al primero de la cola.

Esto implica que aún si el último elemento del arreglo que forma la cola está ocupado, puede insertarse un nuevo valor detrás de este; como primer elemento del arreglo siempre y cuando ese primer elemento esté vacío.

La representación gráfica de esta estructura es la siguiente:




La condición de vacío en este tipo de cola es que el apuntador F sea igual a cero.

Las condiciones que debemos tener presentes al trabajar con este tipo de estructura son las siguientes:

- **OVERFLOW**, cuando se realice una inserción.
- **UNDERFLOW**, cuando se requiera de una extracción en la cola.
- **VACIO**

**ALGORITMOS PARA MANEJAR COLAS CIRCULARES**

**INICIALIZACIÓN**

```
INICIO
        F = 0
        A = 0
FIN
```

**PARA INSERTAR ELEMENTOS**

```
INICIO
 SÍ (F+1 == A) Ó (F == 1 Y A == MÁXIMO)
ENTONCES
   MENSAJE (OVERFLOW)
 EN CASO CONTRARIO
   SÍ A == MÁXIMO ENTONCES
      A = 1
      COLA[A] = VALOR
  EN CASO CONTRARIO
     A  = A + 1
     COLA[A]<-- VALOR
    SÍ F == 0 ENTONCES
       F = 1
   FIN SÍ
  FIN SÍ
FIN
```

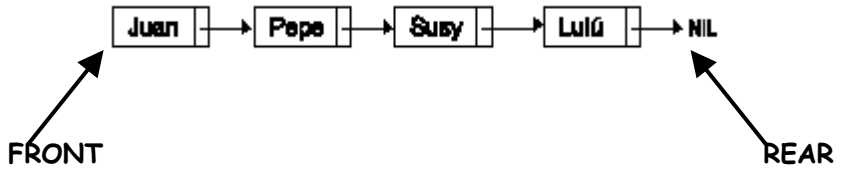
**PARA EXTRAER ELEMENTOS**

```
INICIO
  SÍ F == 0 ENTONCES
    MENSAJE (UNDERFLOW)
  EN CASO CONTRARIO
    X  = COLA[F]
    SÍ F == A ENTONCES
       F = 0
       A =  0
    EN CASO CONTRARIO
       SÍ F == MÁXIMO ENTONCES
          F  = 1
       EN CASO CONTRARIO
          F  = F + 1
     FIN SÍ
  FIN SÍ
FIN
```

**NOTA:** Al iniciar cuando la cola está vacía **QREAR y QFRONT** van a ser igual al máximo valor del valor del arreglo.

**COLAS CON LISTAS ENCADENADAS**

Como se ha visto anteriormente la cola funciona con dos extremos, una parte de enfrente para eliminación de los elementos y una parte de atrás para inserción de los elementos. Una cola encadenada se representará así:




**ALGORITMO PARA REMOVER ELEMENTOS EN UNA COLA ENCADENADA**

```
INICIO
  SÍ EMPTY (Q) == VERDADERO ENTONCES
   "ERROR DE UNDEFLOW"
 DE LO CONTRARIO
   P = FRONT
```

```
AREA INFORMATICA - INACAP
   X = INFO(P)
   FRONT = NEXT (P)
   SÍ FRONT == NIL ENTONCES
      REAR=NIL
   FIN SÍ
   FREENODE(P)
FIN SÍ
FIN
```

**ALGORITMO PARA INSERCIÓN DE ELEMENTOS EN UNA COLA ENCADENADA**

```
INICIO
 P = GETNODE()
 INFO(P) = X
 NEXT(P) = NIL
 SÍ REAR == NIL ENTONCES
    FRONT = P
 DE LO CONTARIO
    NEXT (REAR) = P
 FIN SÍ
 REAR=P
FIN
```

**2.24 Modelo de implementacion de colas en lenguaje PASCAL**

A continuación presentaremos algoritmos desarrollados en lenguaje PASCAL, de algunos programas analizados anteriormente.

**DEFINICIÓN DE NODO EN PASCAL.**

Declaración de tipos y variables asociadas a la manipulación de colas en PASCAL.

**ESTÁTICA.**

Se representa con un vector y dos números. El Nº frente me da la posición del primero en salir y el Nº final el último en entrar. Vamos a hacer un vector circular.

Para obtener una implementación correcta dejaremos una posición libre en el vector, tenemos dos opciones para dejar una posición vacía:

1ª- final va una posición por delante del último elemento que entró.
2ª- frente va una posición por detrás del primer elemento en salir.

Vamos a optar por la 2ª opción.

*COLALLENA = COLAVACIA*

frente siguiente a final frente = final

```
CONST MAX = ?
TYPE TIPOCOLA = RECORD
        DATOS : ARRAY [1..MAX] OF ... ;
```

```
AREA INFORMATICA - INACAP
            FRENTE, FINAL: 1..MAX ;
      END;
      VAR COLA: TIPOCOLA;

DINÁMICA.
      TYPE PUNTERO = ^NODO;
            NODO = RECORD
            DATO: ... ;
            SIG : PUNTERO; FRENTE FINAL
      END;
      TIPOCOLA = RECORD
            FRENTE, FINAL: PUNTERO; COLA
      END;
      VAR COLA: TIPOCOLA;
```

| OPERACIONES EN LENGUAJE PASCAL | |
|---|---|
| **ESTÁTICAS** | **DINÁMICA** |
| **INICIALIZAR COLAS.**<br>ESTÁTICA (SECUENCIAL):<br>PROCEDURE INICOLA (VAR COLA: TIPOCOLA);<br>BEGIN<br>COLA.FRENTE := MAX;<br>COLA.FINAL := MAX;<br>END;<br>**FUNCIÓN COLAVACIA.**<br>FUNCTION COLAVACIA (COLA : TIPOCOLA): BOOLEAN;<br>BEGIN<br>COLAVACIA := COLA.FRENTE:= COLA.FINAL;<br>END;<br>**FUNCIÓN COLALLENA.**<br>FUNCTION COLALLENA (COLA : TIPOCOLA): BOOLEAN;<br>VAR SIGUIENTE : 1..MAX;<br>BEGIN<br>IF COLA.FINAL=MAX THEN | **INICIALIZAR COLAS.**<br>PROCEDURE INICOLA (VAR COLA: TIPOCOLA);<br>BEGIN<br>COLA.FRENTE := NIL;<br>COLA.FINAL := NIL;<br>END;<br>**FUNCIÓN COLAVACIA.**<br>FUNCTION COLAVACIA (COLA: TIPOCOLA): BOOLEAN;<br>BEGIN<br>COLAVACIA := COLA.FRENTE :=COLA.FINAL;<br>END;<br>**FUNCIÓN COLALLENA.**<br>* La representación de COLALLENA en una lista enlazada no existe ya que una lista nunca se llena.<br>**INSERTAR UN ELEMENTO EN LA COLA.**<br>PROCEDURE INSERTAR (VAR COLA : TIPOCOLA; ELEM : ...); |

```
  COLALLENA :=  TRUE
ELSE COLALLENA :=  FALSE;
END;
```

**INSERTAR UN ELEMENTO EN LA COLA.**

```
PROCEDURE INSERTAR (VAR COLA:
TIPOCOLA; ELEM : ...);
BEGIN
 IF NOT COLALLENA(COLA) THEN
   COLA.FINAL :=COLA.FINAL+1
 ELSE IF COLA.FINAL=MAX THEN
       COLA.FINAL:=1;
 COLA.DATOS[COLA.FINAL]:= ELEM;
END;
```

**SACAR UN ELEMENTO DE LA COLA.**

*(SE EXTRAE POR EL FRENTE)*

```
PROCEDURE EXTRAER (VAR COLA :
TIPOCOLA;VAR ELEM: ... );
BEGIN
 IF NOT COLAVACIA(COLA) THEN
 BEGIN
  IF COLA.FRENTE=MAX THEN
    COLA.FRENTE:=1
  ELSE COLA.FRENTE:=COLA.FRENTE+1;
    ELEM:=COLA.DATOS[COLA.FRENTE];
 END;
END;
```

```
BEGIN
 IF NOT COLAVACIA(COLA) THEN
 BEGIN
  NEW(AUX);
  AUX^.DATO:=ELEM;
  COLA.FINAL^.SIG:=AUX;
  COLA.FINAL:=AUX;
 END
ELSE
 BEGIN
  NEW(AUX);
  AUX^.DATO:=ELEM;
  COLA.FRENTE:=AUX;
  COLA.FINAL:=AUX;
 END;
END;
```

**SACAR UN ELEMENTO DE LA COLA.**

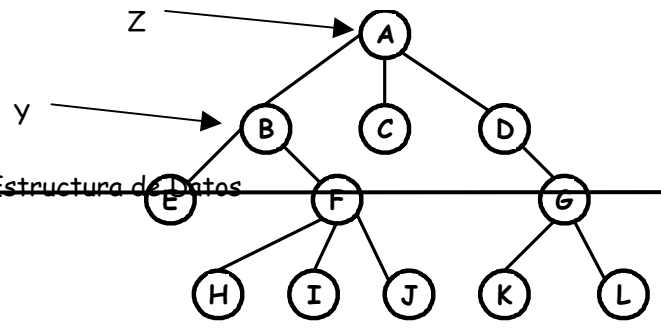
```
PROCEDURE EXTRAER (VAR COLA :
TIPOCOLA;VAR ELEM: ... );
BEGIN
 IF NOT COLAVACIA(COLA) THEN
 BEGIN
  IF COLA.FRENTE=COLA.FINAL THEN
  BEGIN
    AUX:=COLA.FRENTE;
    COLA.FRENTE:=NIL;
    COLA.FINAL:=NIL;
    ELEM:=COLA.FRENTE^.DATO;
    DISPOSE(AUX);
  END
 ELSE
 BEGIN
    ELEM:=COLA.FRENTE^.DATO;
    AUX:=COLA.FRENTE;
    COLA.FRENTE:=AUX^.SIG;
    DISPOSE(AUX);
  END;
 END;
END;
```

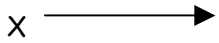
## 2.25 Estructuras de datos no lineales: ARBOL

**Introducción**

Los arboles representan las estructuras no lineales y dinámicas de datos más importantes en computación. Dinámicas porque las estructuras de árbol pueden cambiar durante la ejecución de un programa. No lineales, puesto que a cada elemento del árbol pueden seguirle varios elementos.
Los arboles pueden ser construidos con estructuras estáticas y dinámicas. Las estáticas son arreglos, registros y conjuntos, mientras que las dinámicas están representadas por listas.

La definición de árbol es la siguiente: es una estructura jerárquica aplicada sobre una colección de elementos u objetos llamados nodos; uno de los cuales es conocido como raíz. Además se crea una relación o parentesco entre los nodos dando lugar a términos como padre, hijo, hermano, antecesor, sucesor, ancestro, etc. Formalmente se define un árbol de tipo T como una estructura homogénea que es la concatenación de un elemento de tipo T junto con un número finito de arboles disjuntos, llamados subarboles. Una forma particular de árbol puede ser la estructura vacía.
La figura siguiente representa a un árbol general.






Se utiliza la recursión para definir un árbol porque representa la forma más apropiada y porque además es una característica inherente de los mismos.

Los arboles tienen una gran variedad de aplicaciones. Por ejemplo, se pueden utilizar para representar fórmulas matemáticas, para organizar adecuadamente la información, para construir un árbol genealógico, para el análisis de circuitos eléctricos y para numerar los capítulos y secciones de un libro.

La terminología que por lo regular se utiliza para el manejo de arboles es la siguiente:

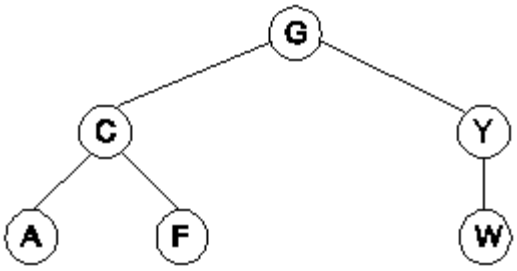
- **HIJO.** X es hijo de Y, sí y solo sí el nodo X es apuntado por Y. También se dice que X es descendiente directo de Y.
- **PADRE.** Z es padre de Y sí y solo sí el nodo Y apunta a Z. También se dice que Y es antecesor de Z.
- **HERMANO.** Dos nodos serán hermanos si son descendientes directos de un mismo nodo.
- **HOJA.** Se le llama hoja o terminal a aquellos nodos que no tienen ramificaciones (hijos).

- **NODO INTERIOR.** Es un nodo que no es raíz ni terminal.
- **GRADO.** Es el número de descendientes directos de un determinado nodo.
- **GRADO DEL ARBOL** Es el máximo grado de todos los nodos del árbol.
- **NIVEL.** Es el número de arcos que deben ser recorridos para llegar a un determinado nodo. Por definición la raíz tiene nivel 1.
- **ALTURA.** Es el máximo número de niveles de todos los nodos del árbol.
- **PESO.** Es el número de nodos del árbol sin contar la raíz.
- **LONGITUD DE CAMINO.** Es el número de arcos que deben ser recorridos para llegar desde la raíz al nodo X. Por definición la raíz tiene longitud de camino 1, y sus descendientes directos longitud de camino 2 y así sucesivamente.

**2.26 Tipo de arboles binarios**

A los arboles ordenados de grado dos se les conoce como arboles binarios ya que cada nodo del árbol no tendrá más de dos descendientes directos. Las aplicaciones de los arboles binarios son muy variadas ya que se les puede utilizar para representar una estructura en la cual es posible tomar decisiones con dos opciones en distintos puntos.

La representación gráfica de un árbol binario es la siguiente:




## 2.27 Definición detallada de la estructura de datos: ARBOL BINARIO.

Es un árbol que o bien esta vacío, o bien esta formado por un nodo o elemento Raíz y dos arboles binarios, llamados subarbol izquierdo y subarbol derecho.

Es un árbol que tiene o 0,1, 2 hijos su nodo. Como máximo dos hijos por elementos.

La terminología que se conoce en la manipulación de la estructura árbol binario es:

- **RAIZ:** Corresponde al primer nodo del árbol.
- **NODO:** cada uno de los elementos de un árbol.
- **SUCESORES DE UN NODO:** Son los elementos de su subarbol izquierdo y de su subarbol derecho.
- **ANTECESORES DE UN NODO:** Son elementos padres de un nodo.
- **NODO HIJO:** Son los sucesores directos de un Nodo.
- **NODO TERMINAL O NODO HOJA:** Es aquel que no tiene hijos.
- **RAMA:** Es cualquier camino que se establece entre la raíz y un nodo terminal.
- **PROFUNDIDAD DE UN ARBOL:** Es el máximo nivel de los nodos de un árbol que coincide con el número de nodos de la rama más larga menos 1(-1)

## 2.28 Representación de un Árbol en memoria.

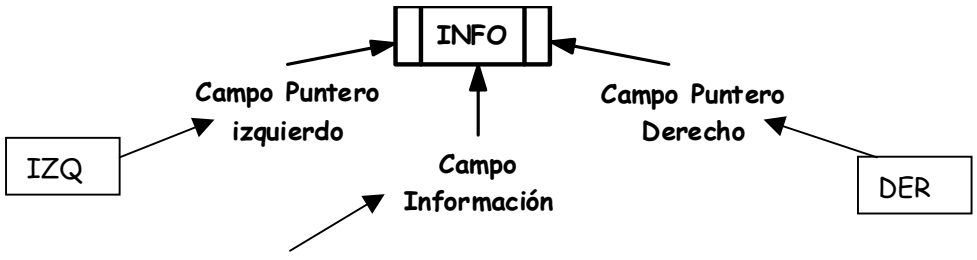
Hay dos formas tradicionales de representar un árbol binario en memoria:

- Por medio de datos tipo punteros también conocidos como variables dinámicas o listas.
- Por medio de arreglos.

Sin embargo la más utilizada es la primera, puesto que es la más natural para tratar estos tipos de estructuras.

Los nodos del árbol binario serán representados como registros que contendrán como mínimos tres campos. En un campo se almacenará la información del nodo. Los dos restantes se utilizarán para apuntar al subarbol izquierdo y derecho del subarbol en cuestión. Esto no quiere decir que en un nodo no pueda existir mas campos de información.

Cada nodo por lo general se representa gráficamente de la siguiente manera:



**RECORRIDO DE UN ARBOL BINARIO**

Hay tres manera de recorrer un árbol: en **INORDEN**, **PREORDEN** y **POSTORDEN**. Cada una de ellas tiene una secuencia distinta para analizar el árbol como se puede ver a continuación:

**ARBOL**

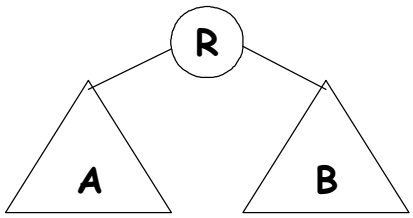



Figura 4.1

1. **INORDEN**
   - Recorrido el subarbol izquierdo en INORDEN.
   - Examinar de la raíz.
   - Recorrido el subarbol derecho en INORDEN.

   Dado la figura 4.1, el recorrido sería en INORDEN:

   **A, R, B**

2. **PREORDEN**
   - Examinar la raíz.
   - Recorrido el subarbol izquierdo en PREORDEN.
   - Recorrido el subarbol derecho en PREORDEN.

   Dado la figura 4.1, el recorrido sería en PREORDEN:

   **R, A, B**

3. **POSTORDEN**
   - Recorrido el subarbol izquierdo en POSTORDEN.
   - Recorrido el subarbol derecho en POSTORDEN.
   - Examinar la raíz.

   Dado la figura 4.1, el recorrido sería en PREORDEN:

   **A, B, R**

## RECORDANDO ALGORITMOS RECURSIVOS

$$\begin{cases} F0 = 1 & Sí \quad n = 0 \end{cases}$$

FACTORIAL =

$$Fn = F(n-1) * n \qquad Sí \quad n > 0$$

**ALGORITMO QUE DA SOLUCIÓN AL PROBLEMA ANTERIOR**

```
FUNCTION FACTORIAL (N: INTEGER): INTEGER;
BEGIN
        IF N = 0 THEN
           FACTORIAL := 1
        ELSE
          FACTORIAL := N*FACTORIAL(N-1)
END;
```

si N=3, entonces

factorial(3) = 3*factorial(2) = 3*2 = **6**
factorial(2) = 2* factorial(1) = 2*1 = 2
factorial(1) = 1* factorial(0) = 1*1 = 1
factorial(0)= 1

**PROGRAMAS EN SEUDO LENGUAJES DE RECORRIDOS DE ARBOLES BINARIOS**

Los algoritmos en **SEUDOLENGUAJE** asociados a cada tipo de recorrido se presentan de acuerdo a una lógica de programa del tipo recursivo(ósea un programa que es llamado así mismo).

Dado, una tipo registro **ARBOL** que posee las siguientes características:

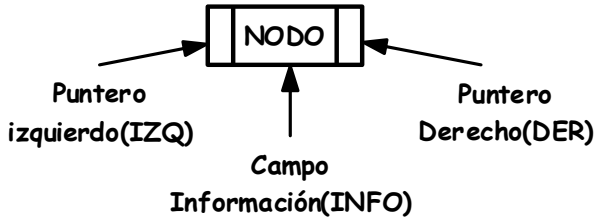
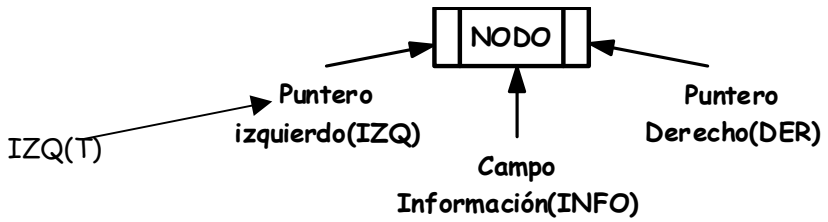
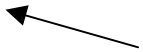



Figura 4.2

Por lo tanto cada nodo que pertenezca a esta estructura de árbol poseerá la organización que se ve en la figura 4.2. Entonces sea **T** una estructura tipo árbol, que esta compuesta de tres campos, que son: **INFO**, **IZQ** y **DER**.





INFO(

DER(T)

Algoritmo de recorrido en **INORDEN**

```
PROCEDURE INORDEN (T: ARBOL)
INICIO
  SÍ T <> NIL ENTONCES
    INORDEN (IZQ(T))
    ESCRIBE (INFO(T))
    INORDEN (DER(T))
  FIN SÍ
FIN
```

Algoritmo de recorrido en **PREORDEN**.

```
PROCEDURE PREORDEN (T: ARBOL)
INICIO
  SÍ T <> NIL ENTONCES
    ESCRIBE (INFO(T))
    PREORDEN (IZQ(T))
    PREORDEN (DER(T))
  FIN SÍ
FIN
```

Algoritmo de recorrido en **POSTORDEN**

```
PROCEDURE POSTORDEN (T: ARBOL)
INICIO
  SÍ T <> NIL ENTONCES
    POSTORDEN (IZQ(T))
    POSTORDEN (DER(T))
    ESCRIBE (INFO(T))
  FIN SÍ
FIN
```

**El algoritmo de creación de un árbol binario es el siguiente:**

```
PROCEDIMIENTO CREAR (Q: NODO)
BEGIN
SÍ Q == NIL ENTONCES
LEER (INFO(P))
IZQ(P) = NIL
```

```
AREA INFORMATICA - INACAP
DER(P) = NIL
Q = P
EN CASO CONTRARIO
  SÍ IZQ(Q) == NIL ENTONCES
        IZQ(P)
        IZQ(Q)= NIL
        CREAR(P)
  CASO CONTRARIO
       IZQ(Q)= NIL
        MENSAJE ("RAMA DERECHA")
       LEE(RESPUESTA)
       SI RESPUESTA="SI" ENTONCES
            NEW(P)
            DER(Q) = P
            CREAR(P)
        EN CASO CONTRARIO
            DER(Q) = NIL
 FIN
INICIO
            NEW(P)
            RAIZ = P
            CREAR(P)
FIN
```

**2.29 Presentación de modelos en lenguaje PASCAL**

A continuación presentaremos algoritmos desarrollados en lenguaje PASCAL, de algunos programas analizados anteriormente.

```
TYPE ARBOL = ^NODO;
        NODO=RECORD
        INFO:.....;
        IZQ,DER:ARBOL;
        END;
VAR RAIZ:ARBOL;
```

Dado el siguiente árbol binario:

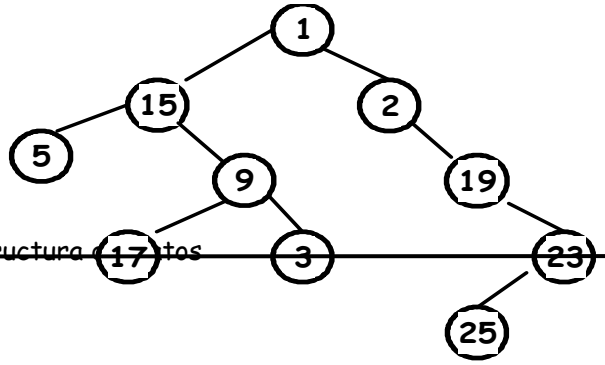


Figura 4.3

**RECORRIDO EN INORDEN**

El árbol anterior sería recorrido así: 5,15,17,9,3,1,2,19,25,23

```
PROCEDURE INORDEN (RAIZ:ARBOL);
BEGIN
        IF RAIZ <> NIL THEN
        BEGIN
                INORDEN (RAIZ^.IZQ);
                PROCESAR (RAIZ^.INFO); (* PUEDE SER IMPRIMIR EL CAMPO INFO*)
                INORDEN (RAIZ^.DER);
        END;
END;
```

**RECORRIDO EN PREORDEN:**

El árbol anterior sería recorrido así: 1,15,5,9,17,3,2,19,23,25

```
PROCEDURE PREORDEN (RAIZ:ARBOL);
BEGIN
 IF RAIZ <> NIL THEN
 BEGIN
   PROCESAR (RAIZ^.INFO); (* PUEDE SER IMPRIMIR EL CAMPO INFO*)
   PREORDEN (RAIZ^.IZQ);
   PREORDEN (RAIZ^.DER);
  END;
END;
```

**RECORRIDO EN POSTORDEN:**

El árbol anterior sería recorrido así: 5,17,3,9,15,25,23,19,2,1

```
PROCEDURE POSTORDEN (RAIZ:ARBOL);
```

```
AREA INFORMATICA - INACAP
BEGIN
 IF RAIZ <>NIL THEN
 BEGIN
  POSTORDEN (RAIZ^.IAZQ);
  POSTORDEN (RAIZ^.DER);
  PROCESAR (RAIZ^.INFO); (* PUEDE SER IMPRIMIR EL CAMPO INFO*)
 END;
END;
```

**ANÁLISIS DE ALGORITMO UTILIZANDO LA ESTRUCTURA DE ÁRBOL BINARIO**

1. Analicemos el siguiente algoritmo que posee una función recursiva que cuenta el nº de nodos que tiene el árbol.

```
FUNCTION CONTAR (RAIZ:ARBOL): INTEGER;
BEGIN
 IF RAIZ=NIL THEN
  CONTAR := 0
 ELSE
  CONTAR := 1 + CONTAR(RAIZ^.IZQ) + CONTAR(RAIZ^.DER)
END;
```

2. Realicemos un análisis del siguiente algoritmo llevando a cabo un seguimiento con el árbol que se entrego anteriormente(utilice la composición del árbol de la figura 4.2).

```
PROCEDURE EJERCICIO(RAIZ:PUNTERO);
BEGIN
        IF RAIZ<>NIL THEN
        BEGIN
         IF RAIZ^.INFO MOD 2=0 THEN WRITE(RAIZ^.INFO);
         EJERCICIO(RAIZ^.IZQ);
         EJERCICIO(RAIZ^.DER);
         IF RAIZ^.INFO>0 THEN WRITE(RAIZ^.INFO);
        END;
END;
```

¿Cuál sería el resultado?

# UNIDAD III: ORDENAMIENTO Y BUSQUEDAS

**Introducción**

La eficiencia de un determinado algoritmo depende de la maquina, y de otros factores externos al propio diseño. Para comparar dos algoritmos sin tener en cuenta estos factores externos se usa la complejidad. Esta es una media informativa del tiempo de ejecución de un algoritmo, y depende de varios factores:

- Los datos de entrada del programa. Dentro de ellos, lo más importante es la cantidad, su disposición, etc.
- La calidad del código generado por el compilador utilizado para crear el programa.
- La naturaleza y rapidez de las instrucciones empleados por la máquina y la propia máquina.

- La propia complejidad del algoritmo base del programa.

El hecho de que el tiempo de ejecución dependa de la entrada, indica que el tiempo de ejecución del programa debe definirse como una función de la entrada. En general la longitud de la entrada es una medida apropiada de tamaño, y se supondrá que tal es la medida utilizada a menos que se especifique lo contrario.

Se acostumbra, pues, a denominar **T(n)** al tiempo de ejecución de un algoritmo en función de **n** datos de entrada.Por ejemplo algunos programas pueden tener un tiempo de ejecución. T(n)=Cn2, donde C es una constante que engloba las características de la máquina y otros factores.

Las unidades de T(n) se dejan sin especificar, pero se puede considerar a T(n) como el numero de instrucciones ejecutadas en una computadora idealizado, y es lo que entendemos por complejidad.

Para muchos programas, el tiempo de ejecución es en realidad una función de la entrada especifica, y no sólo del tamaño de ella.En cualquier caso se define T(n) como el tiempo de ejecución del peor caso, es decir, el máximo valor del tiempo de ejecución para las entradas de tamaño n.

- Un algoritmo es de orden polinomial si T(n) crece mas despacio, a medida que aumenta n, que un polinomio de grado n. Se pueden ejecutar en una computadora.
- En el caso contrario se llama exponencial, y estos no son ejecutables en un computador.

Los algoritmos con el manejo de la información se clasifican en:

**a) De ordenación**
**b) De búsqueda**

Ambos procesos pueden clasificarse como internos o externos dependiendo del lugar en el que se encuentre almacenada la información.

Los internos se llevan a cabo en memoria principal; los externos se realizan en memoria secundaria (Discos flexibles, cintas, discos duros, etcétera).

La operación de **búsqueda** es la que permite recuperar datos previamente almacenados. Podemos definir el proceso de búsqueda como: F(A, X)= Y, donde A es conjunto finito de elementos, y X el elemento a buscar, siendo Y el resultado de la búsqueda.

La operación de **ordenar** significa reorganizar un conjunto de datos u objetos de acuerdo a una secuencia específica. El proceso de ordenación es importante cuando requiere optimizarse un proceso de búsqueda.

Formalmente definimos ordenación de la siguiente manera:
Sea A una lista de n elementos Ao, A1, A2,...,An.

La lista A estará ordenada después de aplicarle un proceso logramos que:
a) Ao <= A1 <= A2 ... <=An (ordenamiento ascendente)
b) Ao>= A1 >= A2 ...>= An (ordenamiento Descendente)

Un método de ordenación es estable si el orden relativo de elementos iguales permanece inalterado durante el proceso de ordenación. La estabilidad es conveniente si los elementos ya se encontraban ordenados conforme a algún otro campo.

Un método de ordenación es inestable si se altera el orden relativo de elementos iguales durante el proceso de ordenación.

## 3.1 DEFINICIÓN DETALLADA DE LA ESTRUCTURA DE DATOS: ALGORITMOS DE ORDENAMIENTOS.

La importancia de mantener nuestros arreglos ordenados radica en que es mucho más rápido tener acceso a un dato en un arreglo ordenado que en uno desordenado.

Existen muchos algoritmos para la ordenación de elementos en arreglos, enseguida veremos algunos de ellos.

### A. SELECCIÓN DIRECTA

Este método consiste en seleccionar el elemento más pequeño de nuestra lista para colocarlo al inicio y así excluirlo de la lista.

Para ahorrar espacio, siempre que vayamos a colocar un elemento en su posición correcta lo intercambiaremos por aquel que la esté ocupando en ese momento.

El método de ordenación por selección directa es más eficiente que los métodos analizados anteriormente.

Pero, aunque su comportamiento es mejor que el de aquéllos y su programación es fácil y comprensible, no es recomendable utilizarlo cuando el número de elementos del arreglo es medio grande.

La idea básica de este algoritmo consiste en buscar el menor elemento en el arreglo y colocarlo en primera posición. Luego se busca el segundo elemento más pequeño del arreglo y se lo coloca en segunda posición. El proceso continua hasta que todos los elementos del arreglo hayan sido ordenados. El método se basa en los siguientes principios:

**1. Seleccionar el menor elemento del arreglo.**
**2. Intercambiar dicho elemento con el primero.**
**3. Repetir los pasos anteriores con los (n-1), (n-2) elementos y así sucesivamente hasta que sólo quede el elemento mayor.**

El algoritmo de selección directa es el siguiente:

```
BEGIN
  I = 1
  MIENTRAS I<= N HAZ
     MIN = I
     J =  I + 1
     MIENTRAS J <= N HAZ
          SI ARREGLO[J] < [MIN] ENTONCES
               MIN = J
          J = J + 1
      FIN MIENTRAS
```

```
AREA INFORMATICA - INACAP
       AUX = ARREGLO[MIN]
      ARREGLO[MIN] = ARREGLO[J]
      ARREGLO[J] = AUX
      I =  I +1
   FIN MIENTRAS
END
```

**B. ORDENACIÓN POR INSERCIÓN DIRECTA**

El método de ordenación por inserción directa es el que generalmente utilizan los jugadores de cartas cuando ordenan éstas, de ahí que también se conozca con el nombre de método de la baraja.

La idea central de este algoritmo consiste en insertar un elemento del arreglo en la parte izquierda del mismo, que ya se encuentra ordenada. Este proceso se repite desde el segundo hasta el n - ésima elemento.

```
BEGIN
 I = 1; J = 1
 MIENTRAS J < N HAZ (* "N" cantidad de elementos a ordenar *)
     I = J
    MIENTRAS (I > 0) AND (ARREGLO[I] < ARREGLO[I-1]) HAZ
       AUX = ARREGLO[I]
       ARREGLO[I] = ARREGLO[I+1]
       ARREGLO[I+1] = AUX
     FIN MIENTRAS
    J = J +1
 FIN MIENTRAS
END
```

**C. ORDENACIÓN POR BURBUJA**

Es el método de ordenación más utilizado por su fácil comprensión y programación, pero es importante señalar que no es el más eficiente de todos los métodos.

Este método consiste en llevar los elementos menores a la izquierda del arreglo ó los mayores a la derecha del mismo. La idea básica del algoritmo es comparar pares de elementos adyacentes e intercambiarlos entre sí hasta que todos se encuentren ordenados.

El método de intercambio directo puede trabajar de dos maneras diferentes. Llevando los elementos más pequeños hacia la parte izquierda del arreglo o bien llevando los elementos más grandes hacia la parte derecha del mismo.

La idea básica de este algoritmo consiste en comparar pares de elementos adyacentes e intercambiarlos entre sí hasta que todos se encuentran ordenados. Se realizan (n-1) pasadas, transportando en cada una de las mismas el menor o mayor elemento (según sea el caso) a su posición ideal. Al final de las (n-1) pasadas los elementos del arreglo estarán ordenados

```
BEGIN
 I = 1
```

```
AREA INFORMATICA - INACAP
 MIENTRAS I < N HAZ
     J =  N
     MIENTRAS J > I HAZ
          SI ARREGLO[J] < ARREGLO[J-1] ENTONCES
             AUX = ARREGLO[J]
             ARREGLO[J] = ARREGLO[J-1]
             ARREGLO[J-1] = AUX
          FIN SÍ
         J =  J - 1
     FIN MIENTRAS
     I = - I +1
 FIN MIENTRAS
END
```

**D. ORDENACIÓN POR MEZCLA**

Este algoritmo consiste en partir el arreglo por la mitad, ordenar la mitad izquierda, ordenar la mitad derecha y mezclar las dos mitades ordenadas en un ARRAY ordenado. Este último paso consiste en ir comparando pares sucesivos de elementos (uno de cada mitad) y poniendo el valor más pequeño en el siguiente hueco.

```
PROCEDIMIENTO MEZCLAR(DAT,IZQP,IZQU,DERP,DERU)
INICIO
     IZQA <- IZQP
     DERA <- DERP
     IND <- IZQP
     MIENTRAS (IZQA <= IZQU) Y (DERA <= DERU) HAZ
          SI ARREGLO[IZQA] < DAT[DERA] ENTONCES
               TEMPORAL[IND] <- ARREGLO[IZQA]
               IZQA <- IZQA + 1
          EN CASO CONTRARIO
               TEMPORAL[IND] <- ARREGLO[DERA]
               DERA <- DERA + 1
          IND <- IND +1
     MIENTRAS IZQA <= IZQU HAZ
          TEMPORAL[IND] <- ARREGLO[IZQA]
          IZQA <- IZQA + 1
          IND <- IND +1
     MIENTRAS DERA <= DERU HAZ
          TEMPORAL[IND] <=DAT[DERA]
          DERA <- DERA + 1
          IND <- IND + 1
     PARA IND <- IZQP HASTA DERU HAZ
          ARREGLO[IND] <- TEMPORAL[IND]
FIN
```

**RESUMEN**

Cabe destacar que dentro de los algoritmos que pertenecen a la familia de los ordenamientos además existen:

- Ordenamiento QUICKSORT
- Ordenamiento HEAPSORT
- Ordenamiento SHAKESORT
- Ordenamiento SHELLSORT

Cada uno de ellos puede ser analizado de acuerdo a su conducta en el manejo de comparaciones y almacenamiento requerido.

## 3.4 DEFINICIÓN DETALLADA DE LA ESTRUCTURA DE DATOS: ALGORITMOS DE BUSQUEDAS.

Una búsqueda es el proceso mediante el cual podemos localizar un elemento con un valor especifico dentro de un conjunto de datos. Terminamos con éxito la búsqueda cuando el elemento es encontrado.

A continuación veremos algunos de los algoritmos de búsqueda que existen.

**A)Búsqueda secuencial**

A este método también se le conoce como búsqueda lineal y consiste en empezar al inicio del conjunto de elementos, e ir a través de ellos hasta encontrar el elemento indicado ó hasta llegar al final de arreglo.

Este es el método de búsqueda más lento, pero si nuestro arreglo se encuentra completamente desordenado es el único que nos podrá ayudar a encontrar el dato que buscamos.

```
IND <- 1
ENCONTRADO <- FALSO
MIENTRAS NO ENCONTRADO Y IND < N HAZ
        SI ARREGLO[IND] = VALOR_BUSCADO ENTONCES
                ENCONTRADO <- VERDADERO
        EN CASO CONTRARIO
                IND <- IND +1
```

**B)Búsqueda binaria**

Las condiciones que debe cumplir el arreglo para poder usar búsqueda binaria son que el arreglo este ordenado y que se conozca el numero de elementos.

Este método consiste en lo siguiente: comparar el elemento buscado con el elemento situado en la mitad del arreglo, si tenemos suerte y los dos valores coinciden, en ese momento la búsqueda termina. Pero como existe un alto porcentaje de que esto no ocurra, repetiremos los pasos anteriores en la mitad inferior del arreglo si el elemento que buscamos resulto menor que el de la mitad del arreglo, o en la mitad superior si el elemento buscado fue mayor.

La búsqueda termina cuando encontramos el elemento o cuando el tamaño del arreglo a examinar sea cero.

```
BEGIN
ENCONTRADO <- FALSO
PRIMERO <- 1
ULTIMO <- N
MIENTRAS PRIMERO <= ULTIMO Y NO ENCONTRADO HAZ
        MITAD <- (PRIMERO + ULTIMO)/2
        SI ARREGLO[MITAD] = VALOR_BUSCADO ENTONCES
                ENCONTRADO <- VERDADERO
        EN CASO CONTRARIO
                SI ARREGLO[MITAD] > VALOR_BUSCADO ENTONCES
                        ULTIMO <- MITAD - 1
                EN CASO CONTRARIO
                        PRIMERO <- MITAD + 1
END
```

**C) Arbol de búsqueda binaria.**

Arbol binario en el cual todos sus nodos cumplen que los nodos de su subarbol izquierdo tienen un valor inferior a él, y los nodos de su subarbol derecho tienen un valor superior a él. No existen valores repetidos.

```
PROCEDURE ABB(RAIZ:AROL;ELEM:INTEGER;POS:ARBOL)
INICIO
  SI RAIZ = NIL ENTONCRES
      POS := NIL
 EN CASO CONTRARIO
    SI INFO(INFO) = ELEM THEN
      POS :=RAIZ
   EN CASO CONTRARIO
     SI ELEM < INFO(RAIZ) ENTONCES
       BUSCAR (IZQ (RAIZ), ELEM, POS)
     EN CASO CONTRARIO
       BUSCAR (DER(RAIZ), ELEM, POS)
     FIN SI
   FIN SI
 FIN SI
FIN

PROCEDURE BUSCAR (RAIZ:ARBOL; ELEM:INTEGER;VAR POS:ARBOL); BEGIN
IF RAIZ=NIL THEN POS:=NIL
ELSE
IF RAIZ^.INFO=ELEM THEN POS :=RAIZ
ELSE
IF ELEM < RAIZ^.INFO THEN
BUSCAR ( RAIZ^.IZQ, ELEM, POS)
```

```
AREA INFORMATICA - INACAP
ELSE
BUSCAR (RAIZ^.DER, ELEM, POS)
END;
```

**C)Búsqueda por HASH**

La idea principal de este método consiste en aplicar una función que traduce el valor del elemento buscado en un rango de direcciones relativas. Una desventaja importante de este método es que puede ocasionar colisiones.

```
FUNCION HASH (VALOR_BUSCADO)
BEGIN
        HASH <- VALOR_BUSCADO MOD NUMERO_PRIMO
END

BEGIN
        INICIO <- HASH (VALOR)
        IL <- INICIO
        ENCONTRADO <- FALSO
        REPITE
                SI ARREGLO[IL] = VALOR ENTONCES
                        ENCONTRADO <- VERDADERO
                EN CASO CONTRARIO
                        IL <- (IL +1) MOD N
        HASTA ENCONTRADO O IL = INICIO
END
```

# UNIDAD IV: ESTRUCTURA DE DATOS EXTERNAS: ARCHIVOS

## 4.1 INTRODUCCIÓN

Los archivos también denominados archivos (**FILE**); es una colección de información (datos relacionados entre sí), localizada o almacenada como una unidad en alguna parte de los sistemas de almacenamientos que posee las computadoras.

Los archivos son el conjunto organizado de informaciones del mismo tipo, que pueden utilizarse en un mismo tratamiento; como soporte material de estas informaciones.

Los archivos como colección de datos sirve para la entrada y salida a la computadora y son manejados con programas(algoritmos).

Los archivos pueden ser contrastados con ARRAYS y registros; Lo que resulta dinámico y por esto en un registro se deben especificar los campos, él número de elementos de un ARRAYS (o arreglo), el número de caracteres en una cadena; por esto se denotan como "Estructuras Estáticas".

En los archivos no se requiere de un tamaño predeterminado; esto significa que se pueden hacer archivos de datos más grandes o pequeños, según se necesiten. Cada archivo es referenciado por su identificador (su nombre.).

Algunos términos asociados son los que a continuación veremos, y podemos ver la figura 4.1 para visualizar dichos conceptos, entonces tenemos.

**Campo(FIELD):** es el elemento de datos básico. Un campo individual contiene un valor único. Esta caracterizado por su longitud y por el tipo de datos. Dependiendo del diseño del archivo, los campos pueden ser de tamaño fijo o variable. Un campo puede contener un subcampo.

**Claves(KEY):** Se denomina a un campo especial del registro que sirve para identificarlo

**Bloque(BLOCK):** Es la cantidad de información que se transfiere en cada operación de lectura o escritura sobre un archivo.

**Registro (RECORD):** Es una colección de campos relacionados que pueden tratarse como una única unidad por un programa de aplicación. Por ejemplo: , un registro de empleados va contener campos como nombre, numero de seguridad social, etc.

También dependiendo del diseño, los registros pueden ser de longitud fija o de longitud variable. Un registro va a tener una longitud variable si algunos de los campos son de tamaños variables o si él numero de campos es variable. Cada campo tiene un nombre de campo.

**Archivo(FILE):** Es una colección de registros similares. El archivo es tratado como una entidad individual por los usuarios y las aplicaciones y puede ser referenciada por el nombre. Los archivos tienen nombres únicos y pueden crearse y borrarse. En un sistema compartido, los usuarios y los programas tienen garantizado o denegado el acceso a archivos completos. En algunos sistemas más complejos, dicho control se aplica a los registros o a los campos.

**Base de datos(DATABASE):** Es una colección de datos relacionados. El aspecto esencial de la base de datos es que la relación que existe entre los elementos de datos es explícita y la base de datos es diseñada para usarse en un numero diferente de aplicaciones. Una base de datos puede contener toda la información relacionado a una organización o proyecto, corno un estudio de mercado o científico. La base de datos consiste en uno o más tipos de archivos.

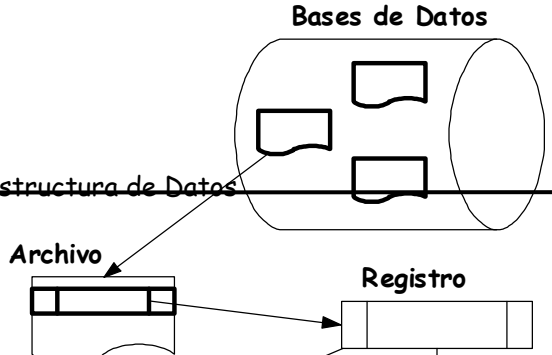


Figura 4.1

**4.2 Definición detallada de la estructura de datos: Archivos.**

Los archivos como colección de datos sirve para la entrada y salida a la computadora y son manejados con programas(algoritmos).

Es una colección de registros similares. El archivo es tratado como una entidad individual por los usuarios y las aplicaciones y puede ser referenciada por el nombre. Los archivos tienen nombres únicos y pueden crearse y borrarse. En un sistema compartido, los usuarios y los programas tienen garantizado o denegado el acceso a archivos completos. En algunos sistemas más complejos, dicho control se aplica a los registros o a los campos

**4.3 Operatorias sobre registros**

Las operaciones que pueden ser llevadas a cabo sobre los registros, pueden ser identificar como:

- **Recuperar Todo (RETRIEVE_ALL):** Recuperar todos los registros de un archivo. Esto va a requerir de una aplicación que deba procesar toda la información de un archivo una vez. Esta opción es usualmente equivalente con el termino de SEQUENTIAL PROCCESSING, ( proceso secuencial), porque todos los registros son accedidos en secuencia.

- **Recuperar Uno (RETRIEVE_ONE):** Esta operación requiere la recuperación de un solo un registro. Las soluciones interactivas orientadas a la transacción necesitan esta operación.

- **Recuperar siguiente (RETRIEVE_NEXT):** Esta operación implica la recuperación del registro que es el siguiente, según una secuencia lógica, el recuperado hace menos tiempo. Un programa que realice búsquedas puede usar también esta operación.

- **Recuperar Previo (RETRIEVE_PREVIOUS):** Es similar a Recuperar Siguiente, pero en este caso el registro que es "previo" al que se esta accediendo en el momento actual.

- **Insertar Uno (INSERT ONE):** Inserta un nuevo registro dentro del archivo. Es necesario que el nuevo registro se ajuste a una posición particular para preservar la secuencia del archivo.

- **Borrar uno (DELETE ONE):** Borra un registro existente. Ciertos enlaces o otras estructuras
  puede que necesiten actualizarse para preservar la secuencia del archivo.

- **Actualizar Uno (UPDATE_ONE):** Recupera un registro o actualiza uno o más de sus campos, y reescribe la actualización en el archivo. Es necesario preservar la secuencia con esta operación. Sí el tamaño del registro esta cambiado, la operación de actualización es más difícil si el tamaño es preservado.

- **Recuperar Varios (RETRIEVE_ FEW):** Recupero un numero de registros.

La naturaleza de las operaciones que comúnmente se ejecutan. sobre un archivo va a influenciar sobre el modo en que se va a organizar el mismo.

**4.4 Tipos de archivos**

Los archivos se clasifican según su uso en dos grupos mayores:

**4.4.1. Según su función.**

Se define por:

**A. Archivos Permanentes:**

Son aquellos cuyo registros sufren pocas o ninguna variación a lo largo del tiempo, se dividen en:

- **Maestro:** Están formados por registros que contienen campos fijos y campos de baja frecuencia de variación en el tiempo.
- **De Situación:** Son los que en cada momento contienen información actualizada.

- **Históricos:** Contienen información acumulada a lo largo del tiempo de archivos que han sufridos procesos de actualización o bien acumulan datos de variación periódica en el tiempo.

**B. Archivos de Movimiento**

Son aquellos que se utilizan conjuntamente con los maestros (constantes), y contienen algún campo común en sus registros con aquellos, para el procesamiento de las modificaciones experimentadas por los mismos.

**C. Archivo de Maniobra o Transitorio**

Son los archivos creados auxiliares creados durante la ejecución del programa y borrados habitualmente al terminar el mismo.

**4.4.2. Según sus elementos.**

Los principales archivos de este tipo son:

**A. Archivo de Entrada**: Una colección de datos localizados en un dispositivo de entrada.
**B. Archivo de Salida**: Una colección de información visualizada por la computadora.
**C. Constantes**: están formados por registros que contienen campos fijos y campos de baja frecuencia de variación en el tiempo.
**D. De Situación**: son los que en cada momento contienen información actualizada.
**C. Históricos**: Contienen información acumulada a lo largo del tiempo de archivos que han sufrido procesos de actualización, o bien acumulan datos de variación periódica en el tiempo.
**E. Archivos de Movimiento o Transacciones**: Son aquellos que se utilizan conjuntamente con los maestros (constantes), y contienen algún campo común en sus registros con aquellos, para el procesamiento de las modificaciones experimentados por los mismos.
**E. Archivos de Maniobra o Transitorios**: Son los archivos auxiliares creados durante la ejecución del programa y borrados habitualmente al terminar el mismo.

**4.5 Operaciones generales que se realizan sobre un archivo.**

Las operaciones generales que se realizan son:

**Creación:** Escritura de todos sus registros.
**Consulta:** Lectura de todos sus registros.
**Actualización:** Inserción supresión o modificación de algunos de sus registros
**Clasificación:** Reubicación de los registros de tal forma que queden ordenados según determinados criterios.
**Borrado:** Eliminando total del archivo, dejando libre el espacio del soporte que ocupaba.
**Compactado:** Esta opción es la más sencilla, siendo útil tanto para registros de tamaño fijo como variable. Puede no ser una solución eficiente para archivo con un alto grado de actualización

**4.6 Factores de utilización de los archivos.**

Teniendo presente las características de cada tipo de organización de archivos y los métodos de acceso, podemos analizar diversos factores que influyen en la decisión de elegir una alternativa durante el proceso de análisis y diseño. Estos factores son:

- Utilización del archivo
- Volatilidad
- Frecuencia de acceso
- Tamaño
- Velocidad y tiempo de respuesta.

**Utilización del archivo:**

La utilización del archivo corresponde al carácter de los datos que se desea almacenar; su grado de estabilidad o permanencia, el método de acceso requerido y los requerimientos propios de la aplicación. Esto nos lleva a definir Archivos maestros y Archivos de trabajo.
Los archivos Maestros contienen información de tipo permanente, que sufre pocas modificaciones a lo largo del tiempo y que es utilizada por varios procesos o programas dentro de la aplicación. Es necesario tener en cuenta diversos factores, tales como:

Organización de los otros archivos que interactuan con este. Por ejemplo, si se debe actualizar un Maestro de Productos, a partir de un archivo de entrada ordenado por código, es perfectamente posible hacerlo en un proceso consecutivo. Ello indica que la organización más adecuada sería una secuencial o indexada.

Si el procesamiento o el archivo de entrada es directo, conviene usar una organización directa para el maestro. De esta forma se realiza un solo acceso de disco en lugar de los dos accesos requeridos para una organización secuencial indexada.

La forma de procesar el archivo en otros programas es importante: si es necesario tener acceso directo y secuencial a la vez, será necesaria una organización indexada o directa y no se podrá usar la secuencial simple.

El tiempo de respuesta requerido para aplicaciones interactivas en línea. Si el usuario desea visualizar un registro en el terminal o pantalla, probablemente lo más rápido sea usar un archivo directo, aunque la mayoría de aplicaciones bastará con un acceso directo por clave a un archivo secuencial indexado.

Los archivos de trabajo contienen información transitoria, un ejemplo el detalle de los productos en una factura. Se usan generalmente para procesos de actualización BATCH de otros archivos (Maestros), impresión de listados, o bien en procesos de entrada de datos interactivo. Normalmente, estos archivos usan una organización directa o secuencial, pero no indexada. Ello se debe al excelente tiempo de respuesta en el caso de aplicaciones interactivas y al menor número de accesos de disco que es necesario efectuar, en el caso de archivos de impresión.

La organización de un archivo debe definirse al momento de crearlo, pero el tipo de acceso puede variar en los diferentes programas que usan el mismo archivo. Por ello es importante tener en cuenta la tabla anterior., que indica todas las posibles combinaciones de organización y método de acceso. Interesa entonces nunca tener que reconstruir un archivo cuando se presente un nuevo requerimiento y la organización usada no autorice el tipo de acceso necesario.

Los requerimientos específicos de la aplicación son muy importante: se requiere procesar todos y cada uno de los registros de un archivo de trabajo para generar un informe impreso, lo más indicado será una organización secuencial. Si se requiere actualizar uno o dos registros de un archivo con más de 10.000 productos en un sistema de inventario, seguramente será mejor una organización directa. Si se desea generar una estadística que cubra el 30% de los productos, tal vez lo más indicado sea una organización indexada. Como norma general, aunque no como axioma, podemos decir, que la organización secuencial es apropiada para procesos Batch, la indexada es para uso general y la directa es para procesos interactivos.

**Volatilidad:**

La volatilidad de un archivo se refiere a ala permanencia de los registros en el mismo. Cuando un archivo es objeto de muchas modificaciones, adiciones y eliminaciones, se dice que este es volátil. En estos casos, generalmente conviene usar una organización directa, debido al menor número de accesos del disco y el tiempo de respuesta bueno.

**Frecuencia de acceso:**

La frecuencia de acceso corresponde al número de veces que realiza una operación de lectura o grabación a diferentes registros de un archivo. Por ejemplo, un archivo de 100 registros que se lee 1.000 veces al día tiene una alta frecuencia de acceso, mientras que un archivo de 1.000 registros que se usa 100 veces al día tiene una baja frecuencia. Generalmente, a mayor frecuencia de acceso, se hace más conveniente tener archivos indexados o directos.

**Tamaño:**

Es necesario tener en cuenta el uso de memoria de almacenamiento al elegir una organización. Tal como vimos anteriormente, una organización secuencial es más eficiente en el uso de memoria que una indexada o directa. Muchas veces, el espacio de disco disponible puede ser un factor critico en el diseño.

**Velocidad y tiempo de respuesta:**

Si la velocidad de proceso o el tiempo de respuesta son factores críticos, generalmente no se pueden usar archivos indexado. Un archivo secuencial es más rápido para procesos consecutivos y una organización directa es más eficiente para proceso de acceso relativo.

**4.7 Acceso a archivos.**

Se refiere al método utilizado para acceder a los registros de un archivo prescindiendo de su organización. Existen distintas formas de acceder a los datos:

- **Secuenciales:** los registros se leen desde el principio hasta el final del archivo, de tal forma que para leer un registro se leen todos los que preceden.

- **Directo:** cada registro puede leerse / escribirse de forma directa solo con expresar su dirección en el archivo por él numero relativo del registro o por transformaciones de la clave de registro en él numero relativo del registro a acceder.

- **Por Índice:** se accede indirectamente a los registros por su clave, mediante consultas secuenciales a una tabla que contiene la clave y la dirección relativa de cada registro, y posterior acceso directo al registro.

- **Dinámico:** es cuando se accede a los archivos en cualquier de los modos anteriormente citados.

La elección del método esta directamente relacionada con la estructura de los registros del archivo y del soporte utilizado.

**4.8 Organización de un Archivo.**

En esta parte vamos a usar él termino organización de archivos para referirnos a la estructura lógica de los registros determinada por la manera en que se accede a ellos. La organización física del archivo en almacenamiento secundario depende de la estrategia de agrupación y de la estrategia de asignación de archivos.

Para seleccionar una organización de archivos hay diversos criterios que son importantes:

- Acceso Rápido para recuperar la información
- Fácil actualización
- Economia de almacenamiento
- Mantenimiento simple.
- Fiabilidad para asegurar la confianza de los datos.

La prioridad relativa de estos criterios va a depender de las aplicaciones que va a usar el archivo.

El numero de alternativas de organización de archivos que se han implementado o propuesto es inmanejable, incluso para un libro dedicado a los sistemas de archivos.

La mayor parte de las estructuras empleadas en los sistemas reales se encuadran en una de estas categorías o puede implementarse como una combinación de estas:

- Pilas **(THE PILE).**
- Archivos secuenciales **(SEQUENTIAL FILE).**
- Archivos Secuenciales indexados**(INDEXED SEQUENTIAL FILE).**
- Archivos indexados**(INDEXED FILE).**
- Archivos directos o de dispersión **(DIRECT, OR HASHED, FILE).**

Por otro lado, los archivos se encuentran organizados lógicamente como una secuencia de registros de varias longitudes diferentes.

**Los archivos de registros de longitud fija**: son los que almacenan la información en los archivos mediante un encabezado y luego se introducen uno a uno los registros ubicados en posiciones consecutivas.
**Los registros de longitud variable**: es el almacenamiento de registros de varios tipos en un archivo y permite uno o más campos de longitudes variables y dichos campos pueden ser repetidos. La longitud de los registros debe estar definida correctamente para poder leer y escribir de forma efectiva.

## ENFOQUES GENERALES PARA LA ORGANIZACIÓN DE ARCHIVOS.

Los enfoques son:

**1. - Enfoque de acceso secuencial:** Se refiere al procesamiento de los archivos de acuerdo con el orden especifico. Ejemplo archivo secuenciales y de texto.

**2. - Enfoque de acceso Directo** Permite recuperar registros individuales sin leer otros registros del archivo, ejemplos archivos indizados.

## ARCHIVOS SECUENCIALES.

Se refiere al procesamiento de los registros, no importa el orden en que se haga, para eso los registros están organizados en forma de una lista y recuperarlos y procesarlos uno por uno de principio a fin.

### Rudimentos de los archivos Secuenciales.

Dependiendo del dispositivo de almacenamiento utilizado el archivo se puede mostrar el usuario como si fuera un sistema secuencial.

Al finalizar un archivo secuencial se denota con una marca de fin de archivo. (End end-of-file). El usuario de un archivo secuencial puede ver los registros en un orden secuencial simple. La única forma de recuperar registros es comenzar al principio y extraerlos en el orden contemplado.

### Cuestiones de programación.

La manipulación de los archivos se hace en el contexto de la programación en un lenguaje por procedimientos de alto nivel. Estos lenguajes tienden a expresar la manipulación de archivos mediante subrutinas que se definen como parte del lenguaje formal o se incluyen como extensiones del lenguaje en una biblioteca estándar.

La mayor parte de los lenguajes por procedimiento de alto nivel cuenta con características que ayudan a detectar la marca de fin de archivo.

## ARCHIVOS INDIZADOS.

Es la aplicación de incluir índices en el almacenamiento de los archivos; de esta forma nos será más fácil buscar algún registro sin necesidad de ver todo el archivo.

Un índice en un archivo consiste en un listado de los valores del campo clave que ocurre en el archivo, junto con la posición de registro correspondiente en el almacenamiento masivo.

### Fundamento de los Índices.

- La colocación de un listado al inicio del archivo: para la identificación del contenido.
- La presentación de un segundo índice: para reflejar la información de cada punto principal del índice anterior.
- La actualización de los índices: Cuando se insertan y eliminan archivos, es preciso actualizar los índices para evitar contratiempos actualizando un archivo.
- La organización de un índice: Nos evita examinar archivo por archivo para recuperar algún registro buscado; por lo tanto ahorraríamos tiempo si tenemos una adecuado organización de los índices.

### Cuestiones de Programación

Algunos lenguajes de alto nivel cuentan con subtítulos para manipular los archivos de un registro indizado.

Valiéndose de las subrutinas es posible escribir programas sin tener que preocuparse por la estructura real del sistema de índices que se aplique.

## ARCHIVOS DISPERSOS.

También llamados (**HASHED FILES**) representan un sistema de almacenamiento de archivos que solo ofrece acceso directo, y permiten calcular la posición de un registro en el almacenamiento masivo.

### Rudimentos de los archivos dispersos.

El usuario debe dividir el área de almacenamiento asignando al archivo en varias secciones llamadas cubetas para poder ingresar los datos.

La distribución de la información en las cubetas es problemática debido a que la estructura de los archivos es dispersa.

Dentro de los archivos se presentan colisiones de información debido al agrupamiento de los registros ingresados.

### Cuestiones de programación.

Casi ninguno de los lenguajes de programación por procedimientos en la actualidad ofrece implantaciones directas de archivos dispersos; esto es debido a las cuestiones dependientes de la aplicación implicadas en el diseño de estos archivos.

**4.10 Tratamiento de archivos secuenciales.**

Estructura de almacenamiento de información, todos del mismo tipo, el almacenamiento se realiza en disco.

**Operaciones existentes.**

- **Definición de un archivo.**
  TYPE registro = ...
  archivo = FILE OF registro;
  VAR f: archivo;

- **Asignación de física de un archivo.**
  ASSIGN (VAR Archivo, 'C:\Archivo físico');

- **Apertura de un archivo.**
  RESET (Var Archivo); Para leer un archivo que ya existe; coloca el puntero en el primer registro del archivo. Si no existe el archivo daría error.

  REWRITE (Var Archivo); Sirve para crear un archivo. Coloca el puntero en el primer archivo.

- **Cierre de un archivo.**
  CLOSE (Var Archivo); Para cerrar el archivo.

- **Escritura de un archivo.**
  WRITE (Var Archivo, Var Registro); Escritura.

- **Lectura de un archivo.**
  READ (Var Archivo, Var Registro); Sirve para leer 1 registro, leemos un registro.

- **Ruptura de control.**
  EOF( ): Devuelve TRUE si es fin de archivo.

**4.11 Operatorias de archivos secuenciales.**

Dada la siguiente declaración de tipos y de variables:
TYPE Fecha =RECORD
DÍA:1..31;
MES: 1..12;
AÑO:1900..1995;

```
AREA INFORMATICA - INACAP
END;
ALUMNO=RECORD
        NOMBRE: STRING;
        F-NAC: FECHA;
        NOTAS: ARRAY [1..6] OF REAL;
END;
ARCHIVO=FILE OF ALUMNO;
VAR
F:ARCHIVO;
```

Hacer un procedimiento que carge de teclado el fichero hasta que el usuario diga que no quiere meter mas datos. Una vez creado hacer un algoritmo que visualice el nombre de los alumnos que tengan más de un 6 de nota media.

```
PROCEDURE CARGAR;
VAR
        REG:ALUMNO;
BEGIN
        ASSIGN(F,'A:\PASCAL\ALUMNOS.DAT);
        RESET(F);
        REWRITE(F);
        SEGUIR:='S';
        REPEAT
        BEGIN
         WRITELN('TECLEE NOMBRE: '); READLN(REG.NOMBRE);
         WRITELN('TECLEE DIA DE NACIMIENTO: '); READLN(REG.FECHA_NAC.DIA);
         WRITELN('TECLEE MES: '); READLN(REG.FECHA_NAC.MES);
         WRITELN('TECLEE AÑO: '); READLN(REG.FECHA_NAC.AÑO);
         WRITELN('TECLEE EL CURSO: '); READLN(REG.CURSO);
         FOR I:=1 TO 6 DO
           WRITE('TECLEE LA NOTA: ');  READ(REG.NOTAS[I]) ;
         WRITELN('¿DESEA SEGUIR?'); SEGUIR:=READKEY;
        END
        UNTIL SEGUIR:='N';
        CLOSE(F);
END;
```