TEMA III

ESTRUCTURAS DE DATOS DINÁMICAS

UNIVERSIDAD NACIONAL DE EDUCACION A DISTANCIA

Dpto. de Informática y Automática
Facultad de Ciencias

Asignatura: ESTRUCTURAS DE DATOS Y ALGORITMOS
Profesor: Roberto Hernández
Coordinador Jose Luis Fernández Marrón

TEMA III
ESTRUCTURAS DE DATOS DINÁMICAS

3.1 INTRODUCCIÓN

En el primer tema se recordaron los tipos de datos estructurados. Recuérdese que una estructura de datos es una colección de componentes cuya organización se caracteriza por las funciones de acceso que se usan para almacenar los elementos individuales. Así por ejemplo, los arreglos unidimensionales se definen como una colección estructurada de elementos del mismo tipo, identificada con un mismo nombre, y tal que se accede a cada componente mediante un índice que indica su posición dentro de la colección. Del mismo modo, el registro es un tipo de datos estructurado compuesto por un número fijo de componentes no necesariamente del mismo tipo (denominadas campos del registro), y tal que a cada componente del registro se accede mediante un selector de campo. En Modula, este selector está formado por el nombre del registro seguido de un punto y del nombre del identificador de campo.

Estas estructuras de datos tienen dos características reseñables. En primer lugar son estructuras incorporadas en los lenguajes de alto nivel, es decir, estos lenguajes disponen de palabras clave para su declaración y expresiones sintácticas determinadas para el acceso a sus elementos.

En segundo lugar, son estructuras de datos estáticas. Una estructura de datos es estática cuando el número de elementos que la componen es fijo y es dinámica si es variable. Por ejemplo, los arreglos unidimensionales son estructuras de datos estáticas por su propia definición ya que en ellos el número de elementos es fijo y viene determinado por el número de valores diferentes que puede tomar su índice. Análogamente, los registros son estructuras de datos estáticas puesto que el número de campos es fijo. Debe observarse que la naturaleza estática o dinámica de una estructura se debe a su propia definición y es independiente del uso que se haga de ella.

- En este capítulo se presentan otros tipos de datos que no satisfacen ninguna de estas dos características, es decir, no están incorporados por los lenguajes de alto nivel y son dinámicos.

3.2 PILAS Y COLAS

En esta sección se presentan dos tipos de datos muy conocidos y utilizados: las pilas y las colas.

- Una pila se define como una estructura de datos en la que el último elemento en entrar es el primero en salir. A estas estructuras se les denomina LIFO (last in, first out).

Por tanto, es una estructura en la que los elementos están ordenados y se añaden y suprimen únicamente por un único extremo, conocido como tope o cabeza de la pila. Un ejemplo típico es una pila de cajas. Sólo se puede coger la caja de arriba de la pila, y sólo ahí se puede dejar una caja. Obsérvese que esta disposición de las cajas es una pila porque se actúa así sobre ella. Efectivamente, usted podría pensar en suprimir una caja que no es la primera o podría ser lo suficientemente habilidoso como para introducir una entre dos ya colocadas. En este caso la disposición de las cajas no es en una estructura pila, será otra cosa (una lista enlazada, por ejemplo). Como puede observarse, la estructura pila es dinámica ya que el número de elementos que la componen es variable.

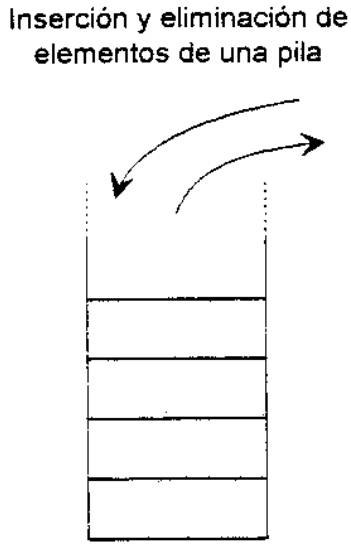


Fig 1. Inserción y eliminación de elementos de una pila

- Una cola se define como una estructura de datos en la que los elementos están ordenados y el primero en entrar es el primero en salir. A estas estructuras se les denomina FIFO (first in, first out).

Por tanto, es una estructura en la que los elementos se añaden por un extremo, denominado final de cola, y se suprimen por otro, conocido como principio de cola. Un ejemplo típico es una cola de un cine. El primero que llega a la cola es el primero que compra la entrada y sale. Podría pensarse en introducirse por el medio, pero colarse no está permitido. Pero además, debe observarse que tampoco está permitido por la definición de cola el abandonarla si no se está al principio. Por tanto, si se desiste de comprar la entrada y se sale en una posición intermedia la estructura ya no es una cola, puesto que no satisface su definición. De nuevo puede observarse que la estructura cola es dinámica ya que el número de elementos que la componen es variable.

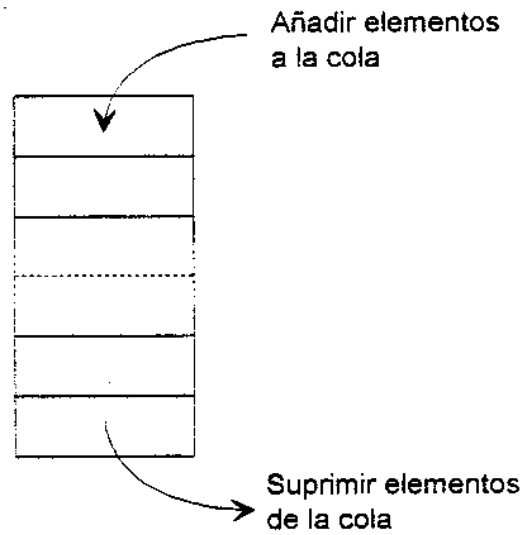


Fig 2. Inserción y eliminación de elementos de una pila

Implementación de Pilas y Colas mediante arreglos

En este apartado se discute la implementación de una pila y una cola utilizando un arreglo. En ambos casos los elementos a almacenar son del mismo tipo, con lo que la utilización de un arreglo es adecuada. En primer lugar se analizan las operaciones necesarias para el tratamiento de la estructura, implementandose seguidamente en Modula.

Pilas.

Las dos operaciones básicas en una pila son evidentemente meter y sacar elementos de la pila. Estas operaciones las realizarán los procedimientos Meter_pila() y Sacar_pila() respectivamente. Pero además son convenientes las siguientes operaciones auxiliares:

Inicia_pila(): crea una pila o limpia una existente inicializándola a una pila vacía, que es aquella que no contiene elementos. Permite partir de un estado previamente establecido.

Pila_Vacia(): operación utilizada para consultar si la pila está vacía. Esta operación será necesaria cuando se vaya a sacar un elemento de la pila ya que no se pueden sacar elementos de una pila vacía.

Pila_Llena(): operación utilizada para consultar si la pila está llena. Esta operación será necesaria cuando se vaya a meter un elemento en la pila ya que no se pueden introducir elementos en una estructura llena.

Para su implementación disponemos de la siguiente estructura de datos

```
CONST  MAXPILA=100;
TYPE   Tipo_Indice = 1..MAXPILA;
       Tipo_pila = RECORD
                      datos : ARRAY Tipo_indice OF Tipo_datos;
                      cima : INTEGER;
                   END;
VAR    pila : Tipo_pila;
```

Como puede observarse, la pila se representa mediante un registro con dos campos. En el campo datos se dispone del arreglo en el que se almacenan los elementos de la pila, y el campo cima es un entero que indica la posición del tope de la pila, es decir, la posición del último elemento introducido en la pila.

La operación Inicia_pila() vendrá dada por el siguiente procedimiento

```
PROCEDURE Inicia_pila (VAR pila: Tipo_pila);
  BEGIN
     pila.cima := 0;
  END Inicia_pila;
```

Veamos a nivel lógico cuál es el efecto de los procedimientos Meter_pila() y Sacar_pila(). En la Fig 3.a se muestra una pila vacía. Obsérvese que es pila.cima quien indica que la pila está vacía, y que para "vaciar" la pila no se realiza ninguna acción sobre los elementos del arreglo. Los posibles contenidos de pila.datos deben considerarse "basura".

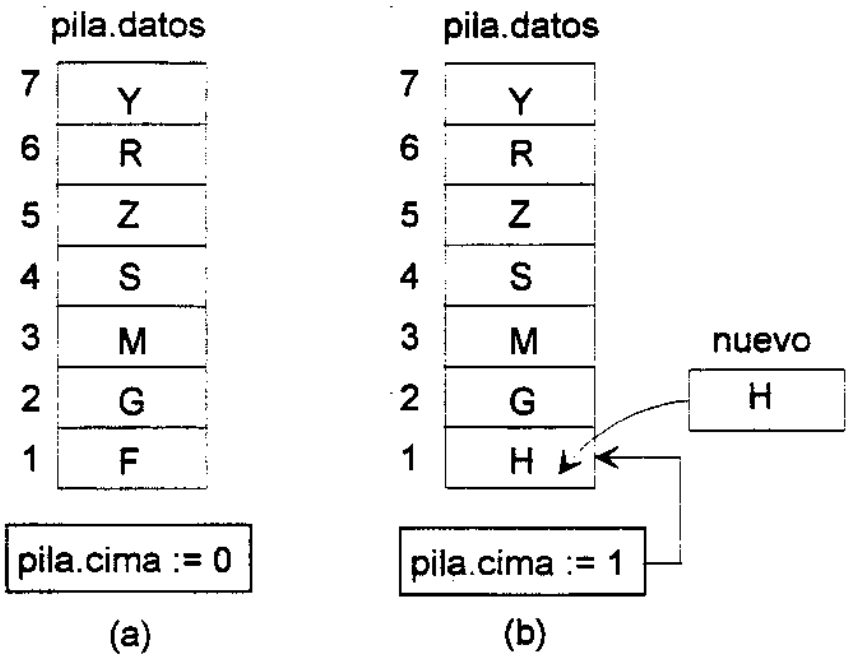


Fig 3. (a) Pila vacía. (b) Introducción de un nuevo elemento en la pila.

El procedimiento Meter_pila() debe por tanto incrementar el valor de la cima e introducir el nuevo elemento.

```
PROCEDURE Meter_pila (VAR pila: Tipo_pila; nuevo_dato : Tipo_datos);
  BEGIN
    pila.cima := pila.cima+1;
    pila.datos[pila.cima] :=nuevo_dato;
  END Meter_pila;
```

El procedimiento Sacar_pila() deberá actuar de modo inverso sacando el elemento indicado por la cima y decrementando seguidamente su valor.

```
PROCEDURE Sacar_pila (VAR pila: Tipo_pila; VAR dato   Tipo_datos)
   BEGIN
     dato := pila.datos[pila.cima];
     pila.cima:=pila.cima-1;
   END Sacar_pila;
```

Debe observarse el carácter dinámico de la pila. La cantidad de elementos que pertenecen a la pila es variable. Así, en la situación de la Fig 3.a la pila no tiene elementos mientras que en la de la Fig 3.b tiene un elemento. En esta situación los elementos del arreglo de índice 2, 3, etc., no forman parte de la pila. Ésta está formada únicamente por el elemento de índice 1. No debe confundirse la pila con el arreglo utilizado para implementarla.

Finalmente, las operaciones de consulta vendrán dadas por las siguientes funciones

```
PROCEDURE Pila_vacia (pila:Tipo_pila) : BOOLEAN;
  BEGIN
     RETURN pila.cima = 0;
  END Pila_vacia;
```

```
PROCEDURE Pila_llena (pila:Tipo_pila) : BOOLEAN;
  BEGIN
     RETURN pila.cima = MAXPILA
  END Pila_llena;
```

En algunas ocasiones es conveniente conocer el contenido del último elemento introducido en la pila pero sin eliminarlo de la misma. Para ello puede utilizarse el siguiente procedimiento

```
PROCEDURE Consultar_pila (pila:Tipo_pila; VAR dato : Tipo_datos);
  BEGIN
     dato := pila.datos[pila.cima]
  END Consultar_pila;
```

Colas.

Para la implementación de una cola mediante arreglos se necesita almacenar el principio y el final de la cola. En la implementación con arreglos se podría considerar en principio la utilización de un único índice para almacenar el final, presuponiendo que el primer elemento de la cola está siempre en la primera posición del arreglo. Sin embargo, esta estructura es inadecuada por ineficaz ya que al sacar un elemento por el principio de la cola (posición 1 del arreglo) deberemos seguidamente mover todos los elementos de la cola una posición hacia el principio tal y como se ilustra en la Fig 4. Este coste computacional hace que la estructura no sea admisible, (aunque "funcione").

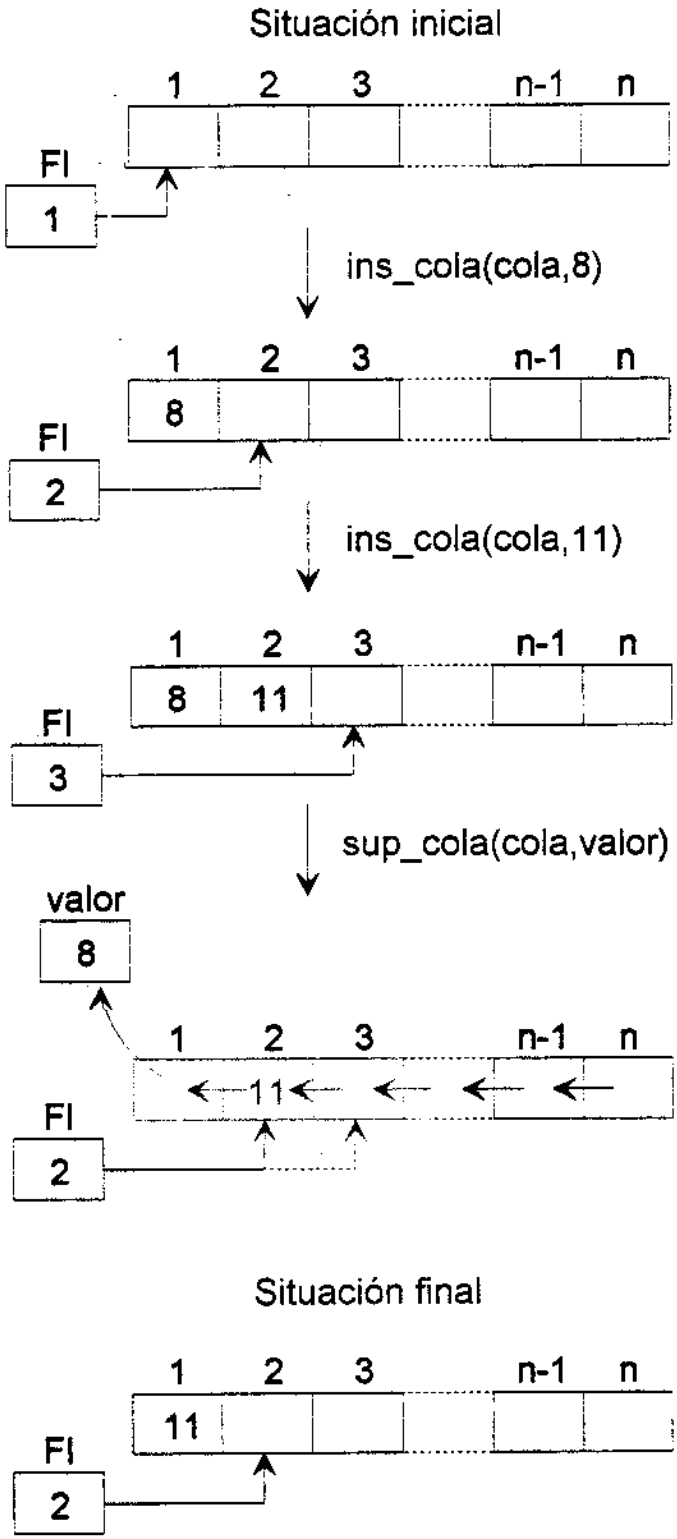


Fig 4. Cola con un único índice (FI = índice de FInal de cola)

Por tanto, es conveniente utilizar dos índices, uno que indique el final (FI) y otro el frente (FR) de la cola, de forma que FI indica el último elemento introducido y aumenta cada vez que se introduce un nuevo elemento en la cola y otro que indique el principio o

frente (FR), que señala el primer elemento a sacar y aumenta al eliminar un elemento. Con esta estructura las posiciones del arreglo en las que se han introducido elementos y posteriormente se han sacado quedan inutilizadas, debido al hecho de que los índices no se decrementan nunca. Este comportamiento se ilustra en la Fig 5. Pensar en decrementar los índices para reutilizar esas posiciones implica mover todos los elementos de la cola, como sucedía en el caso de la estructura con único puntero, y por tanto esta opción debe desestimarse.

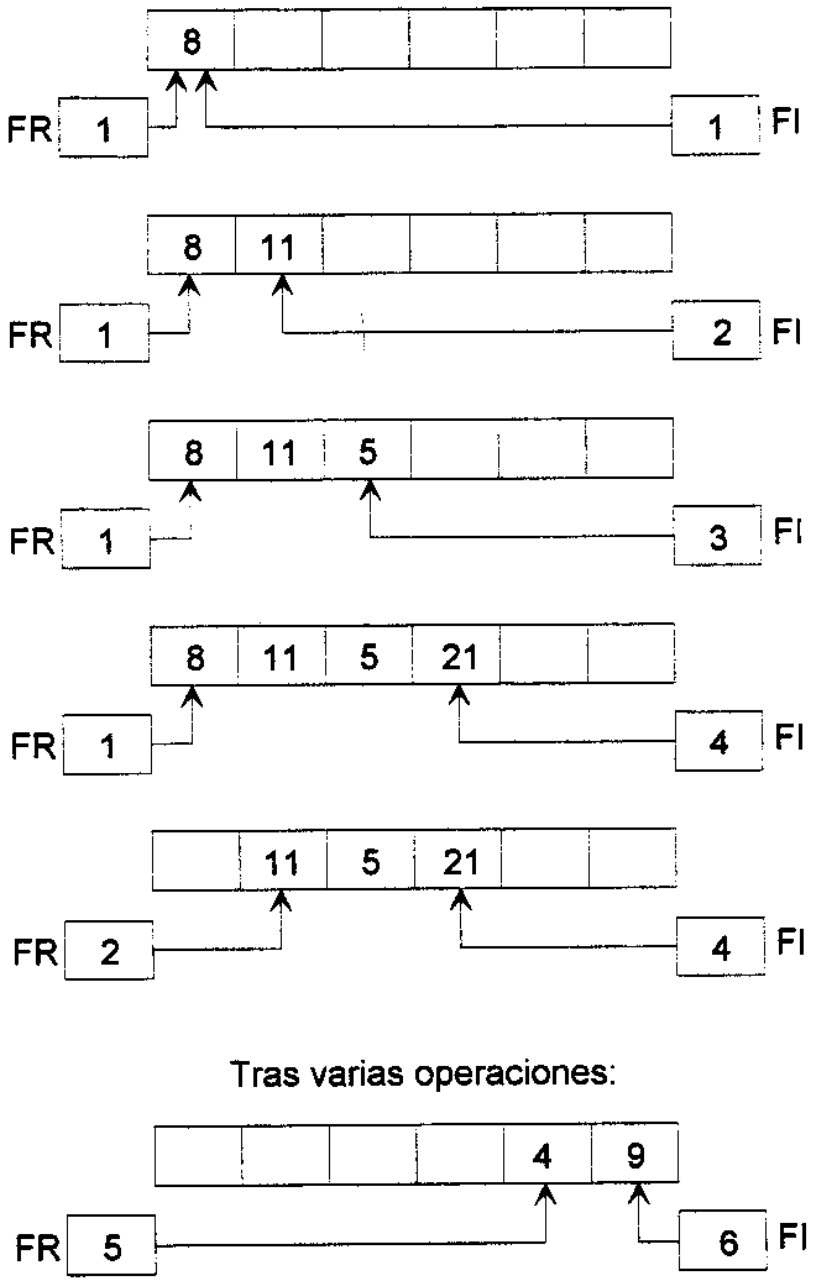


Fig 5. Cola con dos índices y arreglo lineal. Obsérvese la situación final: hay muchos elementos libres pero según el criterio de índices dado no se puede seguir insertando

Para aprovechar las posiciones del arreglo que han sido utilizadas y están disponibles se sustituye la estructura lineal del arreglo por una de tipo circular, tal como se muestra en la Fig 6.

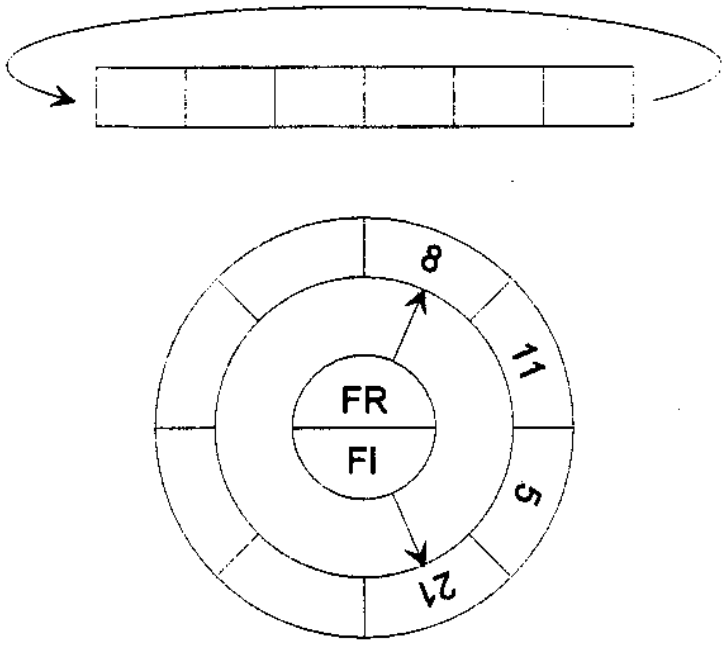
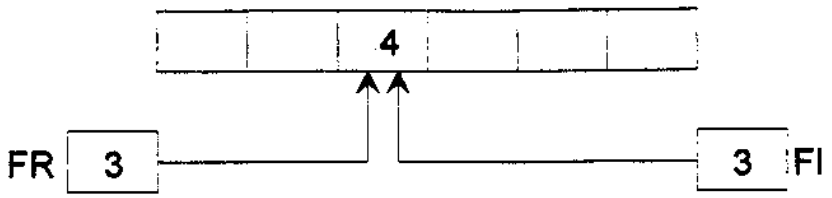


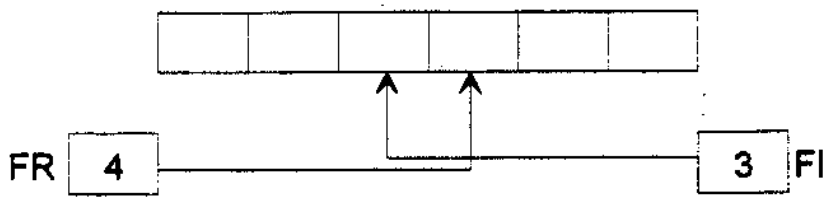
Fig 6. Estructura circular de la cola

Antes de aceptar esta estructura debe analizarse el comportamiento de la misma. Las operaciones de meter y sacar elementos no presentan ningún inconveniente. Sin embargo, no es posible distinguir si la cola está llena o vacía. Debe tenerse en cuenta que con la estructura circular la cola puede quedar vacía o llena en posiciones intermedias del arreglo. La Fig 7.a muestra una situación en la que queda un único elemento en la cola. Entonces ambos índices tienen el mismo valor. Cuando es eliminado el índice frente (FR) se incrementa. En la Fig 7.b se muestra una situación con la cola casi llena. Cuando se inserta un elemento nuevo, el índice final se incrementa y se llena la cola. Como puede observarse los valores de los índices FR y FI son iguales en ambas situaciones, con lo que no puede distinguirse cuando la cola está vacía y cuando está llena.

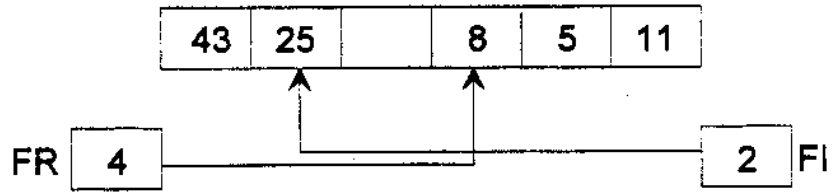
La primera solución que puede pensarse es evidente: mantener un contador que indique los elementos que hay en la cola. Esta solución, además de poco elegante, lleva a la necesidad de incrementar este contador con cada nueva inserción y decrementarlo con cada eliminación.



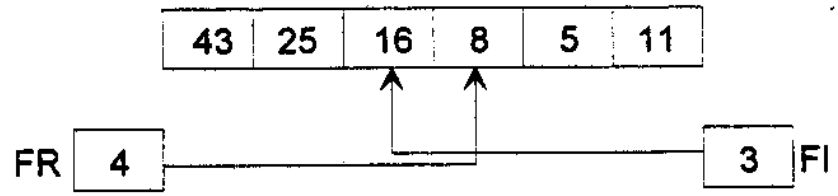
Al suprimir FR aumenta:



(a) La cola está vacía con FR=4 y FI=3



Al insertar FI aumenta:



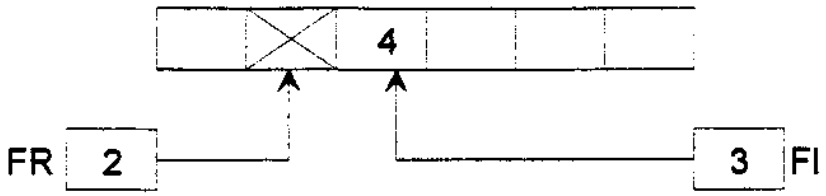
(b) La cola está llena con FR=4 y FI=3

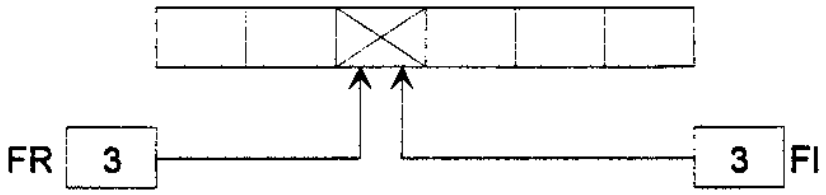
Fig 7. Problema en la distinción entre cola llena y vacía

Una solución que evita estas operaciones adicionales consiste en dejar una posición del arreglo libre. De este modo se define FR tal que apunta a un elemento que nunca se rellena y que es el anterior al primero de la cola, y FI tal que apunta al último elemento introducido. Con estas definiciones las condiciones de cola llena y vacía son las siguientes

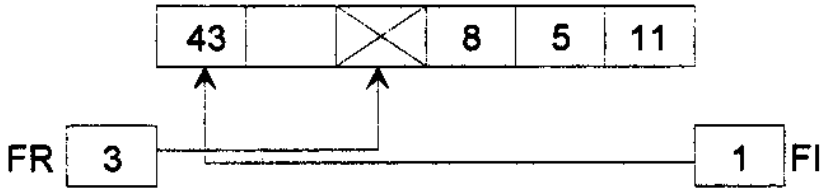
Condición de vacía:	FR=FI
Condición de llena:	FR=FI+1 o (FI=MAX y FR=1)

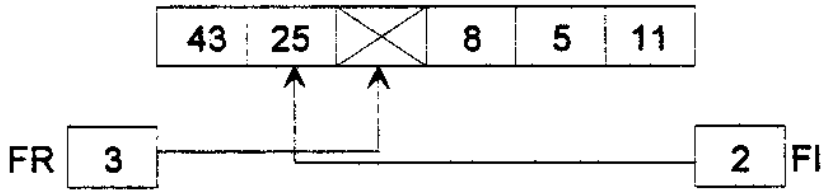
La Fig 8. ilustra estas consideraciones.





(a) La cola está vacía con FR=3 y FI=3





(b) La cola está llena con FR=3 y FI=2

Fig 8. Distinción entre cola llena y vacía

En conclusión, la implementación de la cola mediante arreglos se realiza mediante la siguiente estructura

```
CONST MAXCOLAN=100;
      MAXCOLA=MAXCOLAN+1;

TYPE
      TipoIndice = [1..MAXCOLA];
      Tipo_cola = RECORD
                    datos : ARRAY TipoIndice OF Tipo_datos;
                    fr,fi : TipoIndice;
                  END;
VAR   cola: Tipo_cola;
```

siendo MAXCOLAN el máximo número de elementos que pueden almacenarse en la cola, MAXCOLA = MAXCOLAN + 1 el tamaño del arreglo necesario para implementar la cola, FI el índice que indica el final de la cola, es decir, la posición del arreglo en la que se introdujo el último elemento, y FR el que indica el frente, es decir, la posición anterior al elemento a sacar de la cola.

Las operaciones para manejarla vienen dadas por los procedimientos Inicia_Cola(), Meter_Cola() y Sacar_Cola() y las funciones Cola_Vacia() y Cola_LLena(). Para su implementación deben tenerse en cuenta las siguientes consideraciones. Al inicializar la cola estará vacía y por tanto FR = FI. Dado que la estructura del arreglo es circular cualquier elemento es igualmente válido para hacerlo. Si adoptamos el criterio de que el primer elemento introducido en la cola cuando está vacía se coloca en la primera posición del arreglo, entonces con la cola vacía FI deberá apuntar a MAXCOLA ya que el último elemento es anterior al primero. Por tanto FR = FI = MAXCOLA. Por otro lado, los procedimientos Meter_Cola() y Sacar_Cola() deberán atender por un lado el caso general en el que las operaciones se efectúan en posiciones intermedias del arreglo, y por otro aquel en el que se producen en el extremo indicado por MAXCOLAy que habrá que considerar independientemente.

```
PROCEDURE Inicia_cola (VAR cola: Tipo_cola);
  BEGIN
    cola.fr := MAXCOLA;
    cola.fi := MAXCOLA
  END Inicia_cola;
```

```
PROCEDURE Meter_cola (VAR cola : Tipo_cola; nuevo_dato : Tipo_datos);
  BEGIN
    IF cola.fi = MAXCOLA THEN cola.fi := 1
    ELSE cola.fi := cola.fi+1
    END;
    cola.datos[cola.fi] := nuevo_dato
  END Meter_cola;
```

```
PROCEDURE Sacar_cola (VAR cola: Tipo_cola; VAR dato : Tipo_datos);
  BEGIN
    IF cola.fr = MAXCOLA THEN cola.fr := 1
    ELSE cola.fr := cola.fr + 1
    END;
    dato := cola.datos[cola.fr]
  END Sacar_cola;
```

Finalmente, las funciones de consulta se implementan teniendo en cuenta los criterios adoptados inicialmente.

```
PROCEDURE Cola_vacia (cola:Tipo_cola) : BOOLEAN;
  BEGIN
    RETURN cola.fi = cola.fr;
  END Cola_vacia;
```

```
PROCEDURE Cola_llena (cola:Tipo_cola) : BOOLEAN;
  BEGIN
    IF cola.fi = MAXCOLA THEN
      RETURN cola.fr =1
    ELSE
      RETURN cola.fr = cola.fi + 1;
    END
  END Cola_llena;
```

Para almacenar de una forma organizada un conjunto de datos homogéneos se dispone de los arreglos. Esta estructura, como se dijo, es estática y está incorporada en los lenguajes de alto nivel. En esta sección se presentan las listas enlazadas. Son estructuras dinámicas a nivel lógico (como las pilas y las colas). Su finalidad, como en el caso de los arreglos, es almacenar organizadamente datos del mismo tipo. Se estudia con detalle una implementación basada en variables referenciadas. Por ello se presenta previamente un breve resumen sobre los punteros (o apuntadores) con el fin de recordar su manejo y los conceptos básicos relacionados con ellos.

Punteros

Un puntero es un tipo de datos simple cuyo valor es la dirección de una variable de otro tipo, denominada variable referenciada.

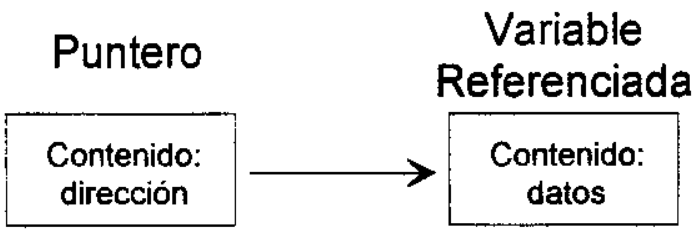


Fig 9. Concepto de punteros

Las variables referenciadas son variables dinámicas, es decir, se crean en tiempo de ejecución. Recuérdese que crear una variable significa reservar en memoria dicha variable, es decir, tomar la memoria libre del sistema necesaria e indicar que está ocupada por dicha variable. Para las variables estáticas, esta indicación se realiza en la sección de declaraciones de forma que en tiempo de ejecución la memoria está reservada siempre, se utilice o no. Sin embargo, la memoria requerida por las variables dinámicas se reserva (como memoria ocupada) durante la ejecución del programa y sólo cuando es necesaria. Cuando no hace falta se libera, es decir, se indica que es memoria libre del sistema y podrá ser reutilizada para otras necesidades.

En la mayoría de las computadoras el valor de un puntero es un entero, debido a que las posiciones de memoria van desde cero hasta el tamaño de la memoria menos uno. Sin embargo, este valor no es de tipo entero sino una dirección. El contenido de la variable referenciada puede ser cualquier tipo de datos, registros, otros punteros, etc.

La declaración de los tipos puntero se realiza en la sección TYPE con la siguiente sintaxis

```
TYPE
    Tipo_puntero = POINTER TO Tipo_referenciada
```

donde Tipo_referenciada es el tipo de datos al que pertenecerán las variables referenciadas. La declaración de la variable puntero se realiza en la sección VAR con la sintaxis

```
VAR
    variable_puntero : Tipo_puntero
```

La variable referenciada es accedida con el puntero mediante el operador ^.

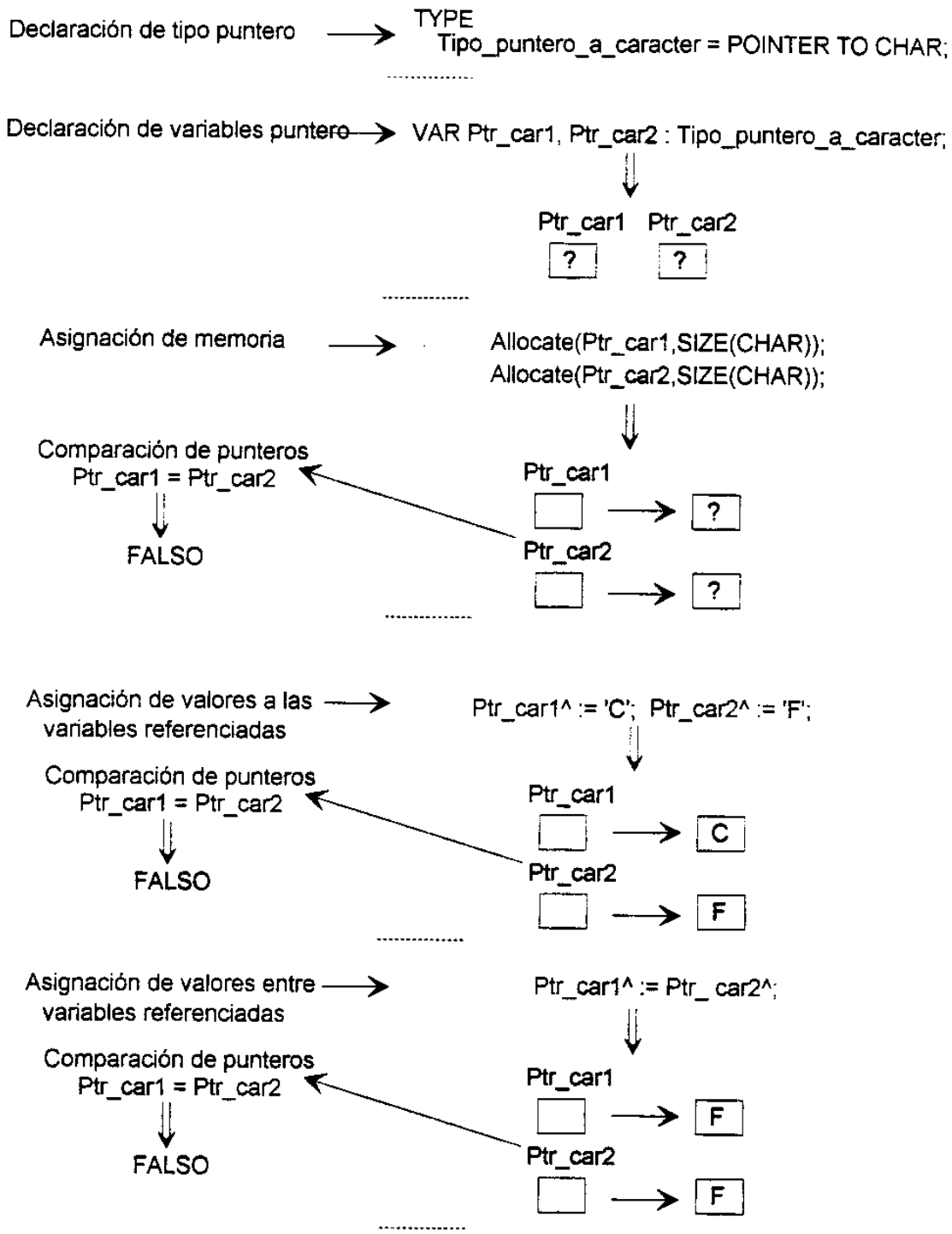
Como puede observarse la memoria necesaria para la variable puntero es reservada y ocupada de la misma forma que las variables estáticas. Sin embargo, la variable referenciada (por ejemplo un registro con varios campos) no existe hasta que no ha sido creada. Para crearla en Modula se dispone del procedimiento

```
Allocate(variable_puntero, SIZE(Tipo_referenciada))
```

Este procedimiento realiza dos acciones. Así, toma la memoria libre necesaria para a variable referenciada y asigna el valor del puntero a la dirección de memoria que ha reservado. Tras este procedimiento la variable referenciada existe en el sentido de que hay memoria asignada para ella y además puede ser accedida por el puntero.

Con los punteros se pueden realizar dos operaciones: la asignación y la comparación. Recuérdese que los valores de los punteros son las direcciones a las que apuntan y por tanto estas operaciones afectan a dichas direcciones. La Fig 10 ilustra estas operaciones y las anteriores consideraciones. Debe tenerse en cuenta que la asignación de punteros puede tener un efecto muy negativo consistente en dejar una variable referenciada sin tener acceso a ello. Este efecto se muestra en la asignación de punteros de la Fig 10. Obsérvese como la variable que estaba apuntada por Ptr_car1 queda sin apuntar por nadie (señalada con un cículo punteado en la Fig 10). Por tanto se está ocupando memoria que no podrá ser reutilizada. Por ello, antes de asignar punteros debe tenerse en cuenta que si la variable referenciada va a ser necesitada posteriormente tiene que estar apuntada por algún puntero, y si no lo va a ser entonces debe ser liberada, es decir, debe ser devuelta a la memoria libre del sistema. Esto se realiza en Modula mediante el procedimiento

```
Deallocate(variable_puntero, SIZE(Tipo_referenciada))
```


Declaración de tipo puntero
TYPE
Tipo_puntero_a_caracter = POINTER TO CHAR;
Declaración de variables puntero
VAR Ptr_car1, Ptr_car2 : Tipo_puntero_a_caracter;
Ptr_car1 Ptr_car2
? ?
Asignación de memoria
Allocate(Ptr_car1,SIZE(CHAR));
Allocate(Ptr_car2,SIZE(CHAR));
Comparación de punteros
Ptr_car1 = Ptr_car2
FALSO
Ptr_car1
?
Ptr_car2
?
Asignación de valores a las variables referenciadas
Ptr_car1^ := 'C'; Ptr_car2^ := 'F';
Comparación de punteros
Ptr_car1 = Ptr_car2
FALSO
Ptr_car1
C
Ptr_car2
F
Asignación de valores entre variables referenciadas
Ptr_car1^ := Ptr_ car2^;
Comparación de punteros
Ptr_car1 = Ptr_car2
FALSO
Ptr_car1
F
Ptr_car2
F

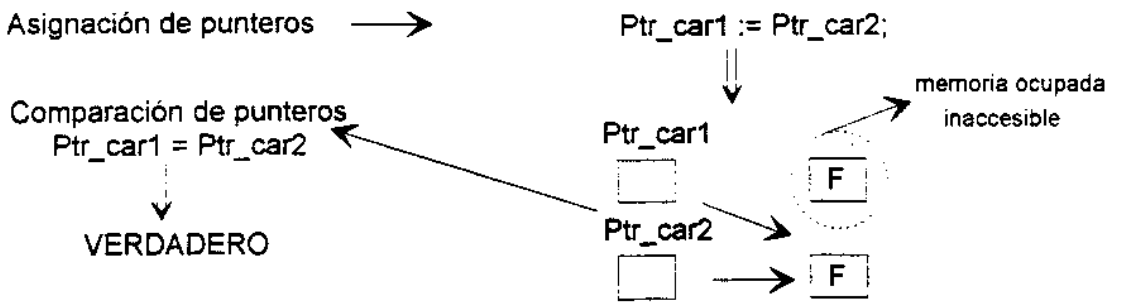


Fig 10. Declaraciones y operaciones con punteros

Por otro lado, es importante determinar cuándo un puntero no señala a ninguna variable referenciada. Para ello en Modula se dispone de la palabra clave NIL. Asi, la asignación de esta variable a un puntero indica que éste no apunta a nada. Su representación gráfica se muestra en la Fig 11.

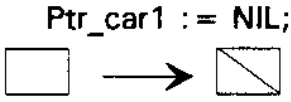


Fig 11. Puntero que no apunta a ninguna variable referenciada

Listas enlazadas

Las listas enlazadas son estructuras de datos dinámicas que se construyen con nodos. Un nodo es un registro con dos campos, uno de ellos contiene las componentes y se le denominará "datos" (o data) y el otro es un valor que señala al siguiente nodo y se le denominará enlace (o next).

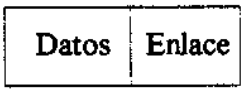


Fig 12. Nodo

Teniendo en cuenta esta definición general de lista enlazada ¿sería posible una implementación estática?

Seguidamente se presenta la implementación dinámica de la lista enlazada. Como se ha dicho, las variables dinámicas pueden crearse y liberarse durante la ejecución del programa. Lógicamente haciendo uso de ellas podrán construirse estructuras de datos con un número de elementos que varía durante la ejecución del programa con lo que evidentemente serán dinámicas. El campo de enlace será un puntero y los nodos son variables referenciadas. Con esto, la declaración de tipos para los nodos es

```
TYPE Ptr_Nodo = POINTER TO Nodo;
     Nodo = RECORD
                datos : Tipo_datos
                enlace : Ptr_Nodo;
            END;
```

Como se ha dicho, una lista enlazada es un conjunto organizado de componentes en las que el orden se establece mediante el campo enlace de cada nodo. Para acceder a la lista se tiene un puntero al primer nodo que se denomina puntero externo. La declaración de la lista se realizará mediante la del puntero externo

```
VAR lista : Ptr_Nodo;
```

La Fig 13. muestra un ejemplo de lista enlazada.

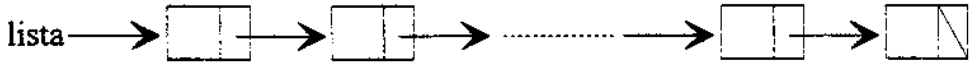


Fig 13. Ejemplo de lista enlazada

Sobre esta estructura se puede realizar cualquier operación necesaria gestionando los punteros. Seguidamente se presentan algunas de las operaciones más elementales.

Inserción por la cabeza

Al ser ésta la primera operación se va a comentar detenidamente para ilustrar el manejo de punteros. En las siguientes operaciones se presentará de forma resumida.

Para insertar un nuevo elemento en la cabeza de una lista ya existente se deberá primero crear un nuevo nodo, seguidamente introducir el nuevo dato a almacenar en el campo de datos y finalmente realizar los enlaces adecuados para recolocar los nodos en la lista. El manejo de punteros se ilustra en la Fig 14. Los números indican el orden en el que debe efectuarse cada enlace, y el nodo rodeado por un circulo punteado es el nuevo nodo. Obsérvese que el orden de las acciones es fundamental. Si primero se realizase la asignación del puntero externo al nuevo nodo, cuando se desease enlazar este nuevo nodo con la lista no se podría, pues su dirección estaba en el puntero lista. La lista se ha perdido. La gestión de punteros debe realizarse de forma cuidadosa. Podría pensarse en almacenar primero todas las direcciones que posiblemente se podrán necesitar, y no prestar mucha atención al orden de los enlaces. Sin embargo esta "solución" no es aceptable (aunque "funcione"), primero porque se utilizan variables innecesarias, segundo porque la lista se manejaría de forma ineficaz y tercero porque se complica la legibilidad del código.

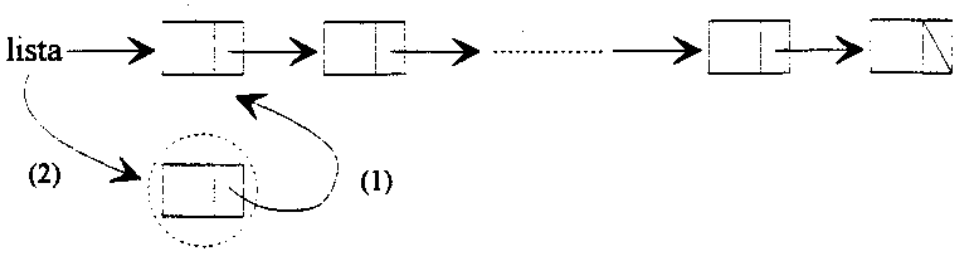


Fig 14. Inserción por la cabeza

Además debe considerarse la situación en la que la lista no existe. En este caso tras crearse el nuevo nodo deberá asignarse el nuevo dato al campo de datos, NIL al de enlace y el puntero externo a este nuevo nodo. En conclusión, el algoritmo en Modula es el siguiente

```
PROCEDURE Insertar_cabeza (VAR lista: Ptr_Nodo; nuevo_dato : Tipo_datos);
VAR
  Nuevo_nodo : Ptr_Nodo;
BEGIN
  ALLOCATE(Nuevo_nodo,SIZE(Nodo));
  Nuevo_nodo^.datos := nuevo_dato;
  IF lista = NIL THEN
     lista := Nuevo_nodo;
     lista^.enlace := NIL;
  ELSE
     Nuevo_nodo^.enlace :=lista;
     lista := Nuevo_nodo;
  END;
END Insertar_cabeza;
```

Insertar por el final

Para insertar por el final deberemos llegar hasta él y hacer la inserción.

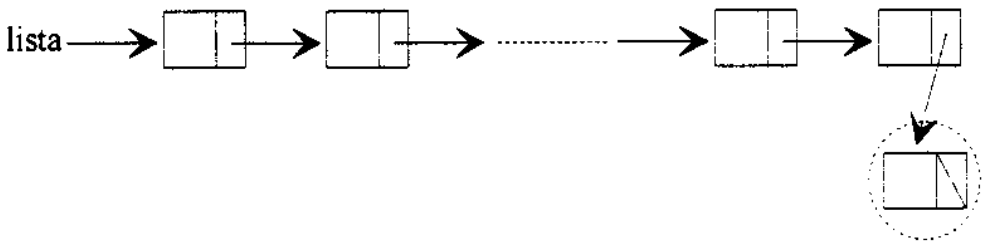


Fig 15. Inserción por el final

```
PROCEDURE Insertar_final (VAR lista: Ptr_Nodo; nuevo_dato : Tipo_datos);
  VAR
    Nuevo_nodo, Actual : Ptr_Nodo;
  BEGIN
    ALLOCATE(Nuevo_nodo,SIZE(Nodo));
    Nuevo_nodo^.datos := nuevo_dato;
    Nuevo_nodo^.enlace := NIL;
    Actual := lista;
    WHILE Actual^.enlace <> NIL DO
      Actual := Actual^.enlace;
    END;
    Actual^.enlace := Nuevo_nodo
  END Insertar_final;
```

Suprimir por la cabeza

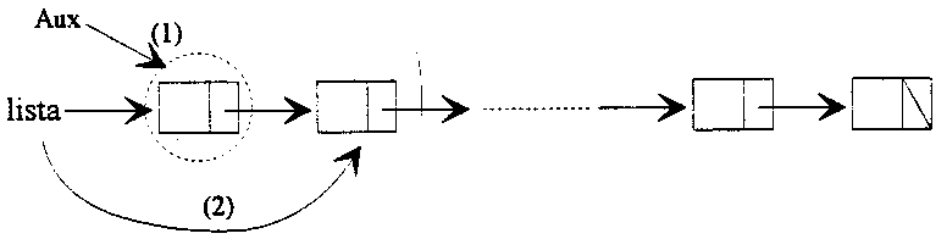


Fig 16. Supresión por la cabeza

```
PROCEDURE Suprimir_cabeza (VAR lista: Ptr_Nodo; VAR dato : Tipo_datos);
  VAR
    Aux : Ptr_Nodo;
  BEGIN
    Aux := lista;
    dato := lista^.datos;
    lista:=lista^.enlace;
    DEALLOCATE(Aux,SIZE(Nodo));
  END Suprimir_cabeza;
```

¿¿Devuelve dato SUPRIMIDO!?

Suprimir por el final

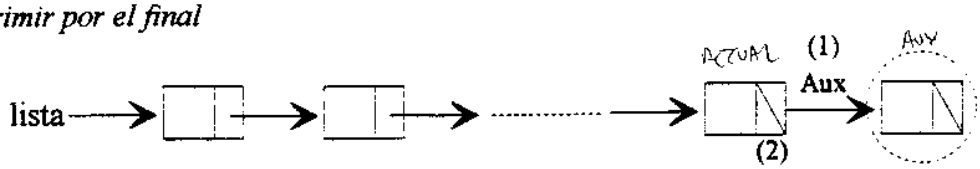


Fig 17. Supresión por el final

```
PROCEDURE Suprimir_final (VAR lista: Ptr_Nodo; VAR dato : Tipo_datos);
  VAR
     Actual, Aux : Ptr_Nodo;
  BEGIN
     Aux := lista;
     WHILE Aux^.enlace <> NIL DO
        Actual := Aux;
        Aux := Aux^.enlace
     END;
     dato := Aux^.datos;
     Actual^.enlace := NIL;
     DEALLOCATE(Aux,SIZE(Nodo))
  END Suprimir_final;
```

Insertar en el medio (según un criterio)

Como último ejemplo de operación se presenta un procedimiento para insertar en el medio. Supóngase que la lista se utiliza para almacenar números enteros (Tipo_datos = INTEGER) y que la lista está ordenada en sentido creciente. El siguiente algoritmo inserta un nuevo número entero en su lugar.

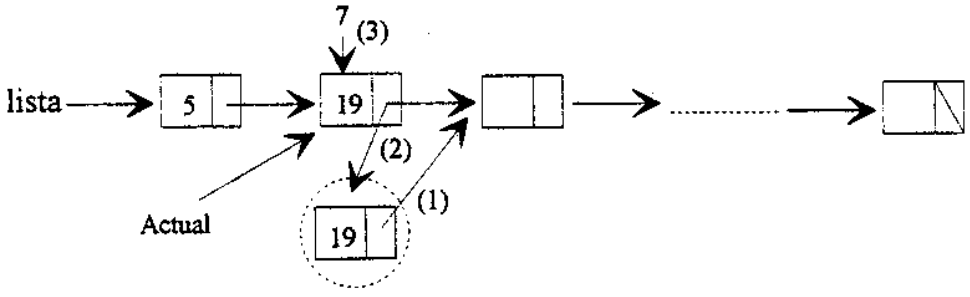
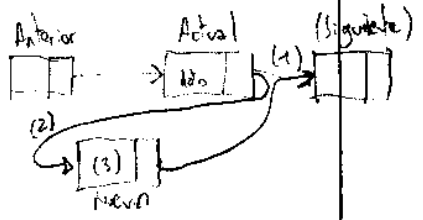


Fig 18. Inserción en el medio, delante de Actual.

Debe observarse que el algoritmo considera las tres situaciones posibles: insertar por el medio una vez localizada su posición, insertar al principio e insertar al final.

Las operaciones sobre listas enlazadas mostradas en este apartado tienen como finalidad fundamental ilustrar el manejo de los punteros, pero por supuesto no son las únicas posibles. Cualquier operación que fuese requerida para la solución de un problema se realizará manejando los punteros de forma adecuada.

```
PROCEDURE Insertar_orden (VAR lista: Ptr_Nodo; dato : Tipo_datos);
  VAR
    Encontrado : BOOLEAN;
    Actual, Nuevo_nodo,Anterior : Ptr_Nodo;
  BEGIN
    IF dato < lista^.datos THEN  (* Insercion al principio *)
      Insertar_cabeza(lista,dato)
    ELSE
      ALLOCATE(Nuevo_nodo,SIZE(Nodo));
      Anterior:=lista;
      Actual := lista^.enlace;
      Encontrado := FALSE;
      WHILE (Actual <> NIL) AND (NOT Encontrado) DO  (* Insercion en el medio *)
        IF dato > Actual^.datos THEN
          Anterior := Actual;
          Actual := Actual^.enlace
        ELSE
          Nuevo_nodo^ := Actual^;          (* (1) *)
          Actual^.enlace := Nuevo_nodo;    (* (2) *)
          Actual^.datos := dato;           (* (3) *)
          Encontrado := TRUE;
        END;
      END;
      IF NOT Encontrado THEN           (* Insercion al final *)
          Nuevo_nodo^.datos := dato;
          Nuevo_nodo^.enlace := NIL;
          Anterior^.enlace := Nuevo_nodo
      END
    END
  END Insertar_orden;
```



3.4. IMPLEMENTACIÓN DINÁMICA DE PILAS Y COLAS

Pilas. La implementación de pilas mediante estructuras dinámicas es inmediata partiendo de una lista enlazada. La operación Meter_pila() es equivalente a insertar por la cabeza de la lista enlazada, y Sacar_pila lo es a sacar por la cabeza. Las funciones y procedimientos son los siguientes.

```
TYPE  Tipo_pila = POINTER TO Nodopila;
      Nodopila = RECORD
                    datos : Tipo_datos;
                    enlace : Tipo_pila;
                 END;
```

```
PROCEDURE Meter_pila (VAR pila: Tipo_pila; nuevo_dato : Tipo_datos);
  VAR
    Nuevo_nodo : Tipo_pila;
BEGIN
  ALLOCATE(Nuevo_nodo,SIZE(Nodopila));
  Nuevo_nodo^.datos := nuevo_dato;
  Nuevo_nodo^.enlace :=pila;
  pila := Nuevo_nodo;
END Meter_pila;
```

```
PROCEDURE Sacar_pila (VAR pila: Tipo_pila; VAR dato : Tipo_datos);
VAR
  Aux : Tipo_pila;
BEGIN
  Aux := pila;
  dato := pila^.datos;
  pila:=pila^.enlace;
  DEALLOCATE(Aux,SIZE(Nodopila));
END Sacar_pila;
```

```
PROCEDURE Inicia_pila (VAR pila: Tipo_pila);
BEGIN
  pila := NIL;
END Inicia_pila;
```

```
PROCEDURE Pila_vacia (pila:Tipo_pila) : BOOLEAN;
BEGIN
  RETURN pila = NIL;
END Pila_vacia;
```

Obsérvese como en esta implementación Meter_pila() no considera el caso en que esté vacía, mientras que el procedimiento Insertar_cabeza() para una lista enlazada considera ambos casos. Esto es debido a que se ha diseñado la función Inicia_pila(), y ésta deberá ser llamada siempre al comienzo del tratamiento de la pila. Compruebe que tras la ejecución de Inicia_pila(), el procedimiento Meter_pila() funciona en todos los casos posibles, y que esto no es cierto si la pila no se inicializa. Por otro lado, la función Pila_llena() no ha sido presentada. Se deja a la consideración del lector la discusión sobre su necesidad y su posible implementación.

Colas. En principio, puede pensarse en utilizar una lista enlazada para implementar una cola, utilizando operaciones análogas a insertar por la cabeza / suprimir por el final o bien insertar por el final / suprimir por la cabeza. Para ello la cola vendría definida por la siguiente estructura.

```
TYPE Tipo_cola = POINTER TO Nodo;
     Nodo = RECORD
                datos : Tipo_datos;
                enlace : Tipo_cola
            END;
```

Se deja al lector la implementación completa de los procedimientos y funciones necesarios en ambos casos. Al hacerlo, observará que con esta estructura de datos la inserción y la supresión por el final requiere un bucle hasta encontrarlo. Esto significa que se tiene un número de comparaciones de punteros igual a la cantidad de elementos que tenga cola. Para mejorar este comportamiento se puede pensar en utilizar otra estructura dinámica, con un puntero externo al principio y otro al final.

```
TYPE  Ptr_nodo = POINTER TO Nodo;
      Nodo = RECORD
                datos : Tipo_datos;
                enlace : Ptr_nodo;
             END;
      Tipo_cola = RECORD
                    frente,final : Ptr_nodo;
                  END;
```

El puntero frente apunta al principio de la lista enlazada y final apunta al último elemento. Debe observarse que con esta estructura de lista enlazada con punteros al primer y último elemento no se puede suprimir al final. Para poder hacerlo, se debería asignar el puntero final al penúltimo elemento, pero su dirección no es conocida a partir

del último nodo. Entonces se debería acceder con un bucle desde el puntero frente, estando en una situación idéntica a la dada con una lista enlazada y un único puntero. Por tanto, se estaría utilizando mal la estructura dinámica propuesta, y la solución dada es incorrecta. En conclusión, la operación Meter_cola() debe realizarse al final de la cola, y la operación Sacar_cola() al principio.

```
PROCEDURE Inicia_cola (VAR cola: Tipo_cola);
BEGIN
   cola.final := NIL;
END Inicia_cola;
```

```
PROCEDURE Meter_cola (VAR cola : Tipo_cola; nuevo_dato : Tipo_datos);
  VAR
     Nuevo_nodo : Ptr_nodo;
  BEGIN
     ALLOCATE(Nuevo_nodo,SIZE(Nodo));
     Nuevo_nodo^.datos := nuevo_dato;
     Nuevo_nodo^.enlace := NIL;
     IF cola.final=NIL THEN
        cola.frente := Nuevo_nodo
     ELSE
        cola.final^.enlace := Nuevo_nodo
     END;
        cola.final := Nuevo_nodo
  END Meter_cola;
```

```
PROCEDURE Sacar_cola (VAR cola: Tipo_cola; VAR dato : Tipo_datos);
  VAR
     Aux : Ptr_nodo;
  BEGIN
     Aux := cola.frente;
     dato := cola.frente^.datos;
     cola.frente := cola.frente^.enlace;
     IF cola.frente =NIL THEN
        cola.final := NIL
     END;
     DEALLOCATE(Aux,SIZE(Nodo));
  END Sacar_cola;
```

```
PROCEDURE Cola_vacia (cola:Tipo_cola) : BOOLEAN;
  BEGIN
    RETURN cola.final = NIL;
  END Cola_vacia;
```

Obsérvese que en esta implementación se ha tomado como criterio de cola vacía que el puntero final apunte a NIL. Se recomienda como ejercicio para el lector el estudio de la implementación de los procedimientos siguiendo el criterio de cola vacía cuando el puntero frente apunta a NIL.

Como se ha dicho anteriormente, esta estructura con dos punteros externos permite un menor coste computacional de las operaciones. Además debe observarse que el gasto en memoria es mínimo ya que es sólo un puntero más (y recordar que su valor es simplemente una dirección). Por tanto, ésta es la estructura más adecuada.

3.5 OTRAS ESTRUCTURAS DINÁMICAS

Partiendo de las variables dinámicas se pueden definir otras estructuras dinámicas similares a las listas enlazadas. La necesidad y la definición de éstas vendrán dadas por las especificaciones del problema a resolver. Un ejemplo típico es el de las listas doblemente enlazadas, que son listas enlazadas bidireccionalmente, tal y como ilustra la Fig 19.

Fig 19. Lista doblemente enlazada lineal

Otra posibilidad es la lista doblemente enlazada circular.

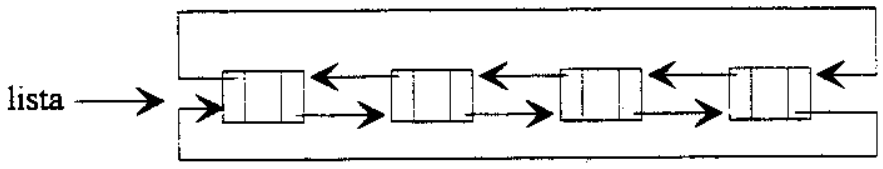


Fig 20. Lista doblemente enlazada circular

Cualquiera de ellas podría ser definida con dos punteros externos (frente y final). A la vista de estas estructuras es conveniente considerar la posibilidad de implementar una cola haciendo uso de una lista doblemente enlazada con dos punteros externos (frente y final). Evidentemente ahora ya no se tiene el problema del costo del bucle para realizar

una de las operaciones (como pasaba en la lista enlazada), ni tampoco la imposibilidad de realizarla insertando al principio y suprimiendo al final (como pasaba en la lista enlazada con dos punteros externos al frente y al final). Pero también es verdad que no presenta ninguna ventaja ni desventaja en relación al coste computacional respecto a la implementación desarrollada en la sección 3.4. Por tanto ¿puede decirse que ambas implementaciones son igualmente buenas?. La respuesta es no. La implementación mediante una lista doblemente enlazada es inadecuada siguiendo el criterio de economía de almacenamiento. Obsérvese que para cada nodo se ha declarado un puntero más y por tanto con cada elemento almacenado se está reservando memoria que no se necesita para implementar la cola. Por tanto sería una solución incorrecta (aunque "funcione").

3.6 CONSIDERACIONES GENERALES Y CONCLUSIONES

Antes de finalizar es conveniente realizar algunas consideraciones generales con el fin de ver con una perspectiva de conjunto los conceptos presentados en este tema y relacionarlos con otros ya conocidos.

- 1) A lo largo del tema se han ido analizando diversas estructuras y diferentes algoritmos y en ocasiones se han encontrado soluciones que no son adecuadas aunque "funcionen". Debemos insistir en este punto. No todo lo que funciona es la solución correcta. Recordar el ejemplo de la implementación de la cola mediante una estructura dinámica. Una lista doblemente enlazada no es adecuada por su innecesario coste de almacenamiento (pero funciona), y una lista lineal con un único puntero externo tampoco es adecuada por su coste computacional (aunque también funciona).

  La conclusión es clara: la lista enlazada con dos punteros externos (frente y final) es la solución adecuada. La idoneidad de una solución vendrá dada, como siempre, por los criterios de economía de almacenamiento y coste computacional.

- 2) En la sección 3.1 se ha definido una estructura de datos dinámica como aquella en la que el número de elementos que la componen es variable. Seguidamente se han definido las pilas y las colas y se ha comentado su naturaleza dinámica. Y posteriormente se ha estudiado su implementación estática y dinámica. Por otro lado, una definición alternativa (y también correcta) de una estructura dinámica es la siguiente: aquella estructura que puede aumentarse o reducirse en tiempo de ejecución. ¿Es esto contradictorio con el hecho de que una pila, por ejemplo, implementada mediante arreglos sea una estructura dinámica?. La respuesta es claramente no. Las estructuras de datos son estáticas o dinámicas dependiendo exclusivamente de su definición, nunca de su implementación. Así, en la implementación estática no debe confundirse la pila con el arreglo. El arreglo no crece ni disminuye en tiempo de ejecución, pero la pila si. Las

componentes del arreglo que no han sido ocupadas introduciendo datos en la pila no forma parte de ésta. Tan dinámica es una pila implementada mediante arreglos como una implementada con una lista enlazada.

3) Las implementaciones estáticas tienen la ventaja de que son más rápidas. El manejo de arreglos utilizando el índice es más rápido que la realización del conjunto de manipulaciones de punteros que son necesarias en las estructuras basadas en variables dinámicas.

4) La diferencia fundamental entre las implementaciones estáticas y dinámicas de estructuras de datos dinámicas radica en el hecho de si se conoce o no a priori un número máximo de datos que deberán ser almacenados. Si este número máximo es conocido la implementación estática no es sólo posible sino que es más adecuada atendiendo a las consideraciones expuestas en el punto 3). Como ejemplo ilustrativo considérese el siguiente problema.

Ejercicio: Realizar un progama en Modula-2 para comprobar que una cadena está bien parentizada. Las cadenas están formadas por paréntesis, corchetes y llaves, abiertos y cerrados ("(", ")", "[", "], "{","}"). No hay prioridades. Ejemplos:

Bien parentizado: {[ ]}( ){[ ]}{}[ ]

Mal parentizado: {[ ])[ ]

a) Supóngase que las cadenas se leen desde consola con buffer.

b) Supóngase que las cadenas se leen desde un fichero.

Al solucionar este ejercicio de programación se llega a la conclusión de que una estructura adecuada es la pila (se recomienda al lector que lo resuelva completamente). La siguiente cuestión a resolver es si debe ser estática o dinámica. En el primer caso a) la especificación dice que las cadenas se leen de consola. Si las consolas tienen un buffer que acepta un número limitado de caracteres, éste puede ser conocido a priori. Por tanto es posible una implementación estática y atendiendo a las consideraciones realizadas en el punto 3) es más adecuada. Sin embargo, en el segundo caso las cadenas se leen desde un fichero. Entonces no podemos conocer a priori un número máximo de caracteres que tienen que almacenarse. Por tanto debe utilizarse una implementación dinámica.

5) Por último debe tenerse en cuenta que la utilización de una estructura dinámica para resolver un problema en el que el número de elementos a almacenar es variable no es siempre adecuada. Supongamos por ejemplo que se necesita almacenar un apellido en una estructura de datos, y que las operaciones a realizar con él descartan la utilización de una pila o una cola. Tendremos que considerar un arreglo o una lista enlazada. Como los apellidos tienen diferente cantidad de caracteres podría pensarse que su almacenamiento

en una lista enlazada es más adecuado que en un arreglo. Sin embargo, esto no es cierto. Es verdad que el número de elementos es variable, pero por otro lado sabemos que el número de caracteres es limitado (cuántos apellidos tienen más de quince caracteres). Si se utilizase una lista enlazada para cada apellido se ahorraría la memoria de las posiciones en las que no hay caracteres. Pero también debe tenerse en cuenta que se gastaría memoria para almacenar los punteros de los nodos (y un carácter ocupa menos que una dirección) y además su manejo es más lento como se indicó en 3). Por tanto, la solución con un arreglo es la adecuada.

En conclusión, no existen reglas ni para decidir de forma absoluta entre las estructuras dinámicas o estáticas ni tampoco para concluir que una implementación dinámica es siempre mejor que una estática para las estructuras dinámicas. Sin embargo, si se dispone de criterios claros para adoptar la solución adecuada al resolver un problema concreto. Estos criterios son la economía de almacenamiento y el costo computacional. Como sucedía en los algoritmos de clasificación se deben conocer todas las posibilidades, en este caso las estructuras y sus posibles implementaciones, y al resolver unas especificaciones dadas elegir razonadamente las más adecuadas.