# Práctica 4. TAD lista posicional

**Objetivos.**

Se trata de construir el TAD lista posicional con una implementación acotada. Haciendo uso de este TAD, se implementará una biblioteca de cálculos estadísticos elementales.

Se introducen los *cursores* como técnica alternativa a los punteros para implementar estructuras de datos dinámicas. También se justifica la necesidad de extender la definición de TadItem mediante la inclusión de la función ValorPorDefecto.

**Enunciado.**

Construir el TAD LISTAPOS según el siguiente módulo de definición:

```
DEFINITION MODULE ListaPos;

FROM TadItem IMPORT ITEM;

TYPE   LISTAPOS;
       TIPO_ERROR = <<a definir>>;

       PROCEDURE Error():  TIPO_ERROR;

       PROCEDURE Crear(): LISTAPOS;

       PROCEDURE EsVacia(l: LISTAPOS): BOOLEAN;

       PROCEDURE Longitud(l: LISTAPOS): CARDINAL;

       (* 1 <= i <= Longitud(l) + 1 *)
       PROCEDURE Insertar(VAR l: LISTAPOS;i: CARDINAL;x: ITEM);

       (* 1 <= i <= Longitud(l) *)
       PROCEDURE Cambiar(VAR l: LISTAPOS;x: ITEM;i: CARDINAL);

       (* 1 <= i <= Longitud(l) *)
       PROCEDURE Eliminar(VAR l: LISTAPOS;i: CARDINAL);

       (* (1 <= i <= Longitud(l)) *)
       PROCEDURE Elemento(l: LISTAPOS;i: CARDINAL): ITEM;

       PROCEDURE Copiar(VAR l1:LISTAPOS; l2:LISTAPOS);

       PROCEDURE Destruir(VAR l: LISTAPOS);

END LISTAPOSPos.
```

Una vez construido el TAD LISTAPOS, implementar una biblioteca estadística con las siguientes operaciones:

```
DEFINITION MODULE Estadis;
       FROM TadItem IMPORT ITEM;
       FROM ListaPos IMPORT LISTAPOS;

       (* NOT EsVacia(l) *)
       PROCEDURE Media(l: LISTAPOS):REAL;
```

```
        (* NOT EsVacia(l) *)
        PROCEDURE Moda (l: LISTAPOS):ITEM;
END Estadis;
```

La media se define como la suma de todos los elementos dividida por el número de elementos y la moda como el elemento de la lista que se repite más veces (si existe más de una moda, se toma la menor de ellas).

Finalmente, se debe construir un programa de prueba para este TAD y su biblioteca.

**Detalles de implementación.**

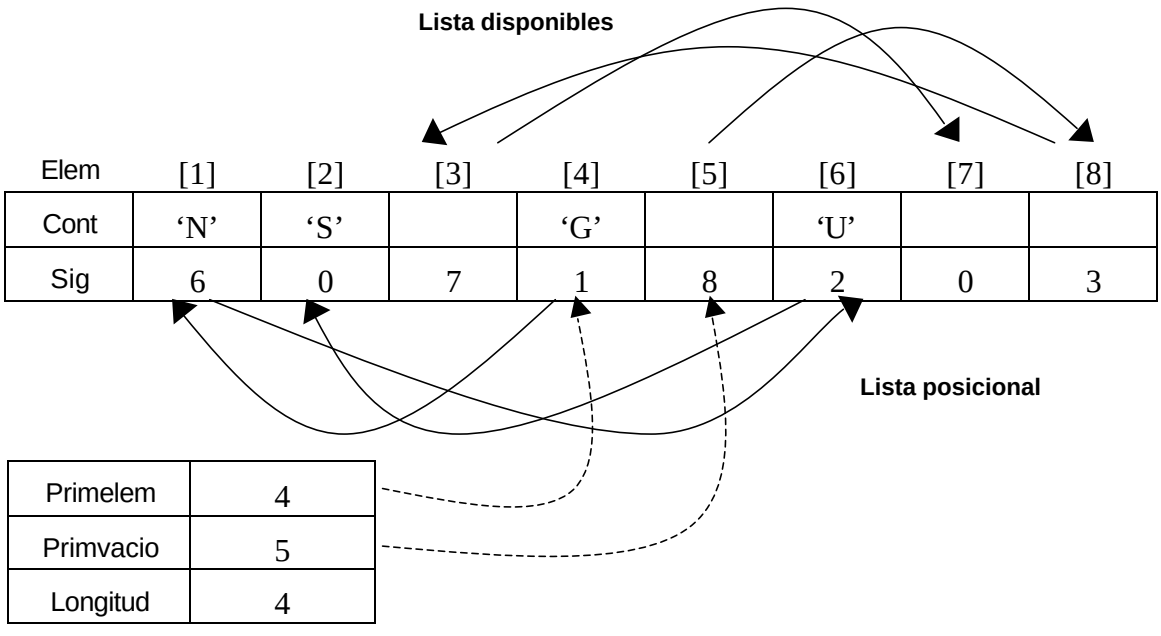
El tipo LISTAPOS se implementará mediante una tabla de *cursores*:

```
CONST MAXLISTAPOS = <<a definir>>;

TYPE
        CURSOR = [0..MAXLISTAPOS];
        CELDA =         RECORD
                                cont: ITEM;
                                sig : CURSOR
                        END;
        ESTRUCTURA =RECORD
                                primelem : CURSOR;
                                primvacio: CURSOR;
                                longitud : CURSOR;
                                elem: ARRAY[1..MAXLISTAPOS] OF CELDA
                        END;
        LISTAPOS = POINTER TO ESTRUCTURA;
```

El campo elem es una tabla que contiene dos secuencias de celdas enlazadas mediante cursores: una secuencia que comienza en primelem almacena los elementos de la lista posicional propiamente dicha, y otra secuencia que comienza en primvacio enlaza las celdas libres de la tabla. El campo longitud almacena el número de elementos contenidos en la lista posicional.

La siguiente figura muestra cómo se almacena una lista posicional (asumiendo ITEM=CHAR) que contenga los elementos ‘G’, ‘N’,’U’, ‘S’:



**Estructuras dinámicas mediante cursores.**

Tradicionalmente, las estructuras de datos dinámicas (es decir, aquéllas que alteran en tiempo de ejecución el número y/o disposición de los nodos que las constituyen) se representan mediante memoria dinámica enlazada a través de punteros. Los cursores surgen como una alternativa a los punteros para representar las estructuras de datos dinámicas. La idea consiste básicamente en simular la memoria dinámica y los punteros mediante arrays e índices de array, respectivamente.

En nuestro ejemplo, reservaremos un array (el campo elem de ESTRUCTURA) que asumirá el papel de la memoria dinámica; mientras que los cursores (datos de tipo CURSOR) asumirán el papel de los punteros. Un cursor será válido (apunta a memoria dinámica válida) si se encuentra dentro de los límites del array (1..MAXLISTAPOS). El cursor nulo (puntero NIL) se representará por un valor fuera de tales límites (en nuestro ejemplo, el valor 0). Una ventaja de los cursores frente a los punteros de Modula-2 es que los primeros son seguros; es decir, se puede asegurar que apuntan a memoria dinámica válida, mientras que no hay forma en Modula-2 de asegurar que un puntero distinto de NIL apunta realmente a memoria dinámica asignada a través del procedimiento NEW.

Por otro lado, el operador ^ que permite acceder a la memoria dinámica apuntada por un puntero es sustituido en nuestra simulación por [], de manera que p^ es equivalente a elem[p], donde se ve la necesidad de referirse a la memoria dinámica (elem) a la que se está accediendo. Esto, lejos de ser un inconveniente es una ventaja, pues abre la posibilidad de emplear varios espacios de memoria dinámica en un mismo programa.

Finalmente, para completar la simulación es necesario sustituir los procedimientos NEW y DISPOSE por versiones análogas que manipulen cursores. El procedimiento

```
PROCEDURE NewCursor(VAR l: LISTAPOS): CURSOR;
```

devuelve un cursor que apunta a una celda libre del array. En nuestra implementación las celdas libres se encuentran enlazadas en una lista de celdas disponibles, de manera que si ésta no está vacía es suficiente devolver el cursor de la cabeza de tal lista (primvacio) y actualizar el valor de la cabeza. Si la lista de disponibles estuviera vacía, se devolverá un cursor nulo. El procedimiento

```
PROCEDURE DisposeCursor(VAR l: LISTAPOS; i: CURSOR);
```

libera la celda apuntada por el cursor i insertándola en la lista de celdas disponibles. Lo más simple es insertar la celda liberada por la cabeza de la lista de celdas disponibles, de manera que ésta se comporta como una pila.

La siguiente tabla muestra un resumen de las equivalencias entre punteros y cursores:

| | Punteros | Cursores |
|---|---|---|
| Memoria | dinámica (heap) | estática (array) |
| Espacio de direcciones | direcciones físicas | índices de array |
| Dirección nula | NIL | índice fuera de rango |
| Tipos | POINTER TO | CURSOR |
| Operador de acceso | ^ ( P^ ) | [] (Mem[P]) |
| Asignación de memoria | NEW | NewCursor |
| Liberación de memoria | DISPOSE | DisposeCursor |

Como se desprende de la tabla anterior, siempre que el programador emplee las equivalencias de modo adecuado y no trate de sacar ventaja de la representación física de los cursores, la programación con cursores es idéntica a la programación con punteros.

**El TAD** TadItem **y la función** ValorPorDefecto**.**

En la práctica anterior se introdujo el TAD TadItem como mecanismo para simular la genericidad en Modula-2. Una de las claves del TAD TadItem es que todas las operaciones sobre del tipo ITEM necesarias para implementar el TAD genérico se exportaban en el módulo de definición de TadItem

```
DEFINITION MODULE TadItem;
        TYPE   ITEM = <<a definir>>;
        (* exportar los procedimientos necesarios para implementar el TAD genérico *)
        ...
END TadItem.
```

de manera que la implementación del TAD genérico jamás accedía de forma directa a la representación física del tipo ITEM, a pesar de tratarse de un tipo transparente.

Considérese ahora la siguiente implementación del procedimiento Elemento del TAD LISTAPOS:

```
(* (1 <= i <= Longitud(l)) *)
PROCEDURE Elemento(l: LISTAPOS;i: CARDINAL): ITEM;
VAR Resultado: ITEM;
BEGIN
        IF l=NIL THEN
                COD_ERROR:= ListaNoCreada;
        ELSIF (i>Longitud(l)) OR (i<1) THEN
                COD_ERROR:= IndiceFueraDeRango;
        ELSE
                ...
                Resultado:= ...;
        END;
        RETURN Resultado;
END Elemento;
```

donde sólo se asigna valor a la variable local Resultado si se satisfacen las precondiciones, de manera que si la lista no ha sido creada o el índice está fuera de rango, se devuelve un valor basura (Modula-2 no inicializa las variables) como resultado. El inconveniente de devolver un valor basura como resultado de una función es que esto puede dar lugar a un error en tiempo de ejecución. Por ejemplo, si el tipo ITEM fuera un enumerado con valores (ROJO,VERDE,AMARILLO,AZUL), representados internamente por valores del 0 al 3, nada nos asegura que el valor inicial de la variable local Resultado se encuentre dentro de este rango. Para evitar que se presenten tales errores, basta inicializar explícitamente la variable local Resultado:

```
PROCEDURE Elemento(l: LISTAPOS;i: CARDINAL): ITEM;
VAR Resultado: ITEM;
BEGIN
        Resultado:= ROJO;
        (* el resto queda como antes*)
        RETURN Resultado;
END Elemento;
```

El inconveniente de tal inicialización es que viola el principio fundamental en que se basa nuestra simulación de la genericidad: si se modifica el tipo ITEM, es necesario revisar todas las inicializaciones explícitas. Para evitar esta dependencia, el TAD TadItem exportará una función

```
PROCEDURE ValorPorDefecto(): ITEM;
```

que devuelve un valor por defecto del tipo ITEM, y las inicializaciones explícitas en la implementación del TAD genérico se implementarán evaluando tal función

```
PROCEDURE Elemento(l: LISTAPOS;i: CARDINAL): ITEM;
VAR Resultado: ITEM;
BEGIN
        Resultado:= ValorPorDefecto();
        (* el resto queda como antes*)
        RETURN Resultado;
END Elemento;
```

En general, no es recomendable que una función devuelva un valor basura aunque su tipo sea un tipo predefinido en Modula-2. Por el contrario, es preferible elegir un valor por defecto para el tipo implicado (por ejemplo, 0 para el CARDINAL y NIL para los punteros) y devolver tal valor en caso de error.

**Práctica Suplementaria.**

Se trata de desarrollar una implementación no acotada con nodo de cabecera de la lista posicional.

Intentaremos optimizar el tiempo de acceso a los elementos haciendo que la lista sea doblemente enlazada y manteniendo tres punteros en la cabecera: uno al primer elemento de la lista, otro al último de la lista y otro al último nodo visitado. Cuando se quiere acceder a una posición, se comprueba de cuál de los tres punteros está más cerca y se llega al nodo deseado a partir de él. El puntero al último visitado se actualizará a esa posición.

Construir dicho tipo con la siguiente estructura:

```
TYPE
PCELDA = POINTER TO CELDA;
CELDA =         RECORD
                        cont: ITEM;
                        sig, ant : PCELDA
                END;
CABECERA = RECORD
                        primero : PCELDA;
                        ultimo: PCELDA;
                        visitado : PCELDA;
                        longitud: CARDINAL;
                        pos_visit: CARDINAL;
                END;
LISTAPOS = POINTER TO CABECERA;
```

El módulo de definición es el mismo que el de la versión acotada. Se debe probar esta implementación empleando la librería estadística y el programa de prueba previamente desarrollados en la práctica.