Práctica 2. TAD pila

Objetivos.

Se trata de construir el TAD PILA con dos implementaciones (una acotada y otra no acotada sin cabecera) que *compartirán el mismo módulo de definición*. Haciendo importación de este TAD se escribirá una aplicación simple.

Se introducen los conceptos de *iterador* y *procedimiento genérico,* utilizando parámetros de tipo procedimiento.

Enunciado.

Construir el TAD PILA según el siguiente módulo de definición:

```
DEFINITION MODULE Pila;
        TYPE    PILA;
                ITEM = <<a definir>>;
                TIPO_ERROR= <<a definir>>;
                TIPO_OPERACION= PROCEDURE (ITEM);

        PROCEDURE Error(): TIPO_ERROR;

        PROCEDURE Crear(): PILA;

        PROCEDURE Apilar(VAR p: PILA;x: ITEM);

        (* Pre: NOT EsVacia(p) *)
        PROCEDURE Desapilar(VAR p: PILA);

        (* Pre: NOT EsVacia(p) *)
        PROCEDURE Cima(p: PILA): ITEM;

        PROCEDURE EsVacia(p: PILA): BOOLEAN;

        PROCEDURE Destruir(VAR p: PILA);

        PROCEDURE Aplicar(p:PILA; op:TIPO_OPERACION);
END Pila.
```

donde ITEM representa el tipo base de la pila, TIPO_ERROR es el tipo definido para tratar los errores tal y como se explicó en la práctica anterior, y TIPO_OPERACION es un tipo cuyo papel se explicará posteriormente.

El TAD PILA se implementará de dos formas diferentes:

Implementación acotada

```
CONST
        MAXPILA= <<a definir>>;
TYPE
        PILA =  POINTER TO ESTRUCTURA;
        ESTRUCTURA =RECORD
                                cima : [0..MAXPILA];
                                cont: ARRAY[1..MAXPILA] OF ITEM;
                        END;
```

Implementación no acotada sin cabecera

```
TYPE
        PILA =  POINTER TO ESTRUCTURA;
        ESTRUCTURA =RECORD
                                cont: ITEM;
                                anterior : PILA
                        END;
```

Es importante entender que ambas implementaciones deben compartir el mismo módulo de definición mostrado en la página anterior. Esto significa que **no debe definirse** un error específico PilaLlena para indicar que la pila acotada está llena y que no pueden apilarse más elementos, pues este error carece de sentido en la implementación no acotada sin cabecera, que en la situación análoga (no hay memoria dinámica disponible) debería indicar el error SinMemoria. Por ello, tomaremos como norma durante el resto del curso que las implementaciones acotadas devuelvan el error SinMemoria cuando el usuario intente insertar un nuevo elemento y no quede espacio libre en la estructura acotada. Por lo tanto, en la implementación acotada del TAD PILA, el procedimiento Apilar debe devolver el error SinMemoria cuando se intente apilar sobre una pila que esté llena; es decir, cuando se hayan utilizado todas las posiciones del array cont.

Una situación diferente es la que se presenta al tratar el error ObjetoNoCreado. En el caso de las estructuras acotadas, un puntero a NIL indica que el objeto no ha sido creado previamente. Por otro lado, en las estructuras no acotadas *sin nodo de cabecera*, un puntero NIL representa un contenedor vacío, mientras que en las estructuras *con nodo de cabecera* un puntero a NIL indica que el objeto no ha sido creado (en particular, no se ha creado la cabecera). Puesto que el tratamiento del error ObjetoNoCreado es inevitable en las estructuras acotadas y en las no acotadas con cabecera, tomaremos por norma durante el resto del curso que este error se definirá en todos los módulos de definición, aunque tal situación de error no se presentará al emplear implementaciones no acotadas sin cabecera. Además, las aplicaciones que se escriban tratarán siempre esta situación de error explícitamente, pues el usuario del TAD no sabe qué tipo de implementación está usando (acotada, no acotada con cabecera, o no acotada sin cabecera).

Como aplicación y haciendo uso de dos pilas ordenar en orden creciente una serie de números (el menor debe estar en la base de la pila). Es importante destacar que el código fuente de esta aplicación debe ser el mismo independientemente de la implementación que se use (acotada o no acotada sin cabecera), dado que ambas comparten el mismo módulo de definición.

Iteradores y Procedimientos genéricos.

En esta práctica se presenta una nueva utilidad del lenguaje, que permite el paso de procedimientos como parámetros. Esta posibilidad nos permitirá definir *iteradores* sobre los tipos abstractos de datos. Un iterador es un procedimiento que visita todos los elementos de la estructura de datos y aplica sobre ellos la misma operación, definida por el parámetro de tipo procedimiento. Hay dos detalles que merece la pena destacar:

1. el iterador no puede modificar los nodos del TAD
2. no es un error iterar sobre un TAD vacío

El procedimiento iterador se puede considerar un procedimiento genérico en el sentido de que puede aplicar una operación u otra dependiendo del procedimiento que reciba como parámetro.

Para definir un procedimiento iterador en Modula-2 son necesarios los siguientes pasos:

1. Definir el tipo procedimiento en el módulo de definición del tipo abstracto. En esta definición simplemente se indica el tipo de los parámetros del procedimiento, si son por valor o por referencia y el tipo del valor que se devuelve, si lo hay.

```
TIPO_OPERACION = PROCEDURE (ITEM);
```

2. Definir el procedimiento iterador, que recibirá como parámetros la variable sobre la que se iterará y el procedimiento que se aplicará a los elementos de la estructura.

```
PROCEDURE Aplicar(p:PILA; op: TIPO_OPERACION);
```

3. Escribir el código del procedimiento iterador en el módulo de implementación. El iterador recorrerá la estructura de datos que implementa el tipo abstracto pudiendo acceder directamente a la estructura. En este recorrido le aplica el procedimiento parámetro a cada elemento.

```
PROCEDURE Aplicar (p: PILA; op:  TIPO_OPERACION);
BEGIN
...
        op(p^.cont[i]);
...
END Aplicar;
```

Para usar un procedimiento genérico en un módulo cliente es necesario:

1. Importar del tipo abstracto el tipo del procedimiento que se pasará como parámetro y el procedimiento genérico.

```
FROM Pila IMPORT . . ., TIPO_OPERACION, Aplicar;
```

2. Definir un procedimiento que corresponda con el tipo importado (o bien usar uno ya definido en otro módulo).

```
MODULE UsuarioPila;

PROCEDURE MiEscribir(x:ITEM);
BEGIN
        WrCard(x,1)
END MiEscribir;
```

3. Invocar el procedimiento iterador importado pasándole como parámetro el procedimiento definido en el paso anterior.

```
BEGIN
. . .
        Aplicar(p, MiEscribir)
. . .
```

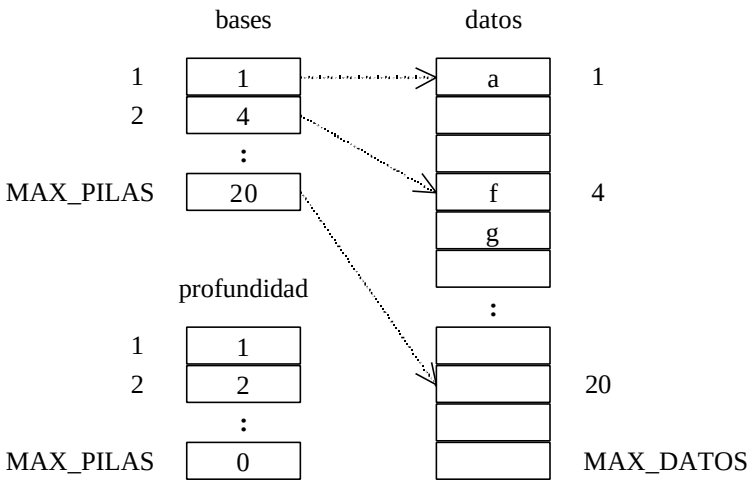
Práctica Suplementaria.

Se trata de implementar el tipo VECTOR_PILAS, que ofrece una estructura en la que un número limitado de pilas comparten el espacio de memoria.

Esta estructura está compuesta por tres arrays. Uno de ellos, de longitud MAX_DATOS, es donde se guardan los elementos. Los otros dos, de longitud MAX_PILAS, guardan, respectivamente, las posiciones del primer elemento de cada pila (base) y el

número de elementos que contiene (profundidad). Los elementos de una misma pila están consecutivos en memoria, pero las pilas están separadas entre sí. En el array de datos, tras la pila i-ésima se encuentra la pila (i+1)-ésima, es decir, el orden de los índices de las pilas es el que determina el orden de las pilas en el array de datos.

La estructura es de la forma:



En el ejemplo de la figura la primera pila empieza en la posición 1 y contiene un elemento, la letra 'a'. La segunda pila empieza en la posición 4 y contiene dos elementos: las letras 'f' y 'g'. La última pila empieza en la posición 20 y está vacía.

El módulo de definición sería el siguiente:

```
DEFINITION MODULE V_P;
        CONST
                MAX_PILAS = << a definir >>;
        TYPE  VECTOR_PILAS;
        ITEM = <<a definir >>;
        TIPO_ERROR= <<a definir>>;
        TIPO_OPERACION = PROCEDURE (ITEM);

        PROCEDURE Error(): TIPO_ERROR;

        PROCEDURE Crear(): VECTOR_PILAS;

        (*Pre: Hay sitio en la estructura y (1 ≤ ind ≤ MAX_PILAS) *)
        PROCEDURE Apilar(VAR p:  VECTOR_PILAS;ind: CARDINAL; x: ITEM);

        (* Pre: NOT EsVacia(p) y (1 ≤ ind ≤ MAX_PILAS) *)
        PROCEDURE Desapilar(VAR p:  VECTOR_PILAS ind: CARDINAL;);

        (* Pre: NOT EsVacia(p) y (1 ≤ ind ≤ MAX_PILAS) *)
        PROCEDURE Cima(p:  VECTOR_PILAS;ind: CARDINAL): ITEM;

        (* Pre: 1 ≤ ind ≤ MAX_PILAS *)
        PROCEDURE EsVacia(p:  VECTOR_PILAS; ind: CARDINAL): BOOLEAN;

        PROCEDURE Destruir(VAR p:  VECTOR_PILAS);

        (* Pre: 1 ≤ ind ≤ MAX_PILAS *)
        PROCEDURE Aplicar(   p: VECTOR_PILAS; ind: CARDINAL;
                             op: TIPO_OPERACION);
```

```
END V_P.
```

Funcionamiento.

Cuando se crea la estructura, ha de haber MAX_PILAS pilas vacías y sus posiciones base deben estar distribuidas uniformemente a lo largo del array de datos.

Cuando en esta estructura se quiera apilar un dato en una pila, puede ocurrir que hayamos llegado a la base de la siguiente pila. En ese caso hay que reorganizar la estructura desplazando las pilas para distribuir el espacio libre. La reorganización implica mover la posición de, posiblemente, todas las pilas, bien hacia delante o hacia atrás para que todas ellas dispongan de sitio libre para poder seguir apilando elementos.

La reorganización se puede dividir en dos pasos, la política de distribución del espacio libre entre las pilas y el algoritmo de desplazamiento de las pilas.

Política de distribución de espacio libre

Hay diferentes políticas de distribución del espacio libre a las pilas. Por ejemplo, dividir equitativamente el espacio libre entre todas las pilas, o dividirlo proporcionalmente al tamaño de cada pila, o de forma inversamente proporcional, etc. Con la política elegida, se implementará un procedimiento que calcule las nuevas posiciones para las bases de las pilas.

Algoritmo de desplazamiento de pilas

También aquí existen diferentes soluciones. Una solución simple, pero con mayor gasto de memoria consiste en hacer uso de un array temporal donde copiar el array datos. Otra solución, más compleja, pero que no necesita de espacio extra de almacenamiento, consiste en mover las pilas *in situ*. Hay que tener en cuenta que las pilas se pueden mover hacia adelante o hacia atrás y hay que mover la pila 'i' antes que la 'j' si la nueva posición de la pila 'j' se solapa con la antigua de la 'i'. Para ello, se han de considerar las pilas en orden, de la primera a la última y, a la vez, mantener una *pila de objetivos*. Cada objetivo consiste en mover una pila a su nueva posición. Si al considerar una nueva pila se puede mover de forma segura, se hace y se reconsidera la pila que hay en la cima de la pila de objetivos. Si no se puede mover de forma segura, se apila en la pila de objetivos.