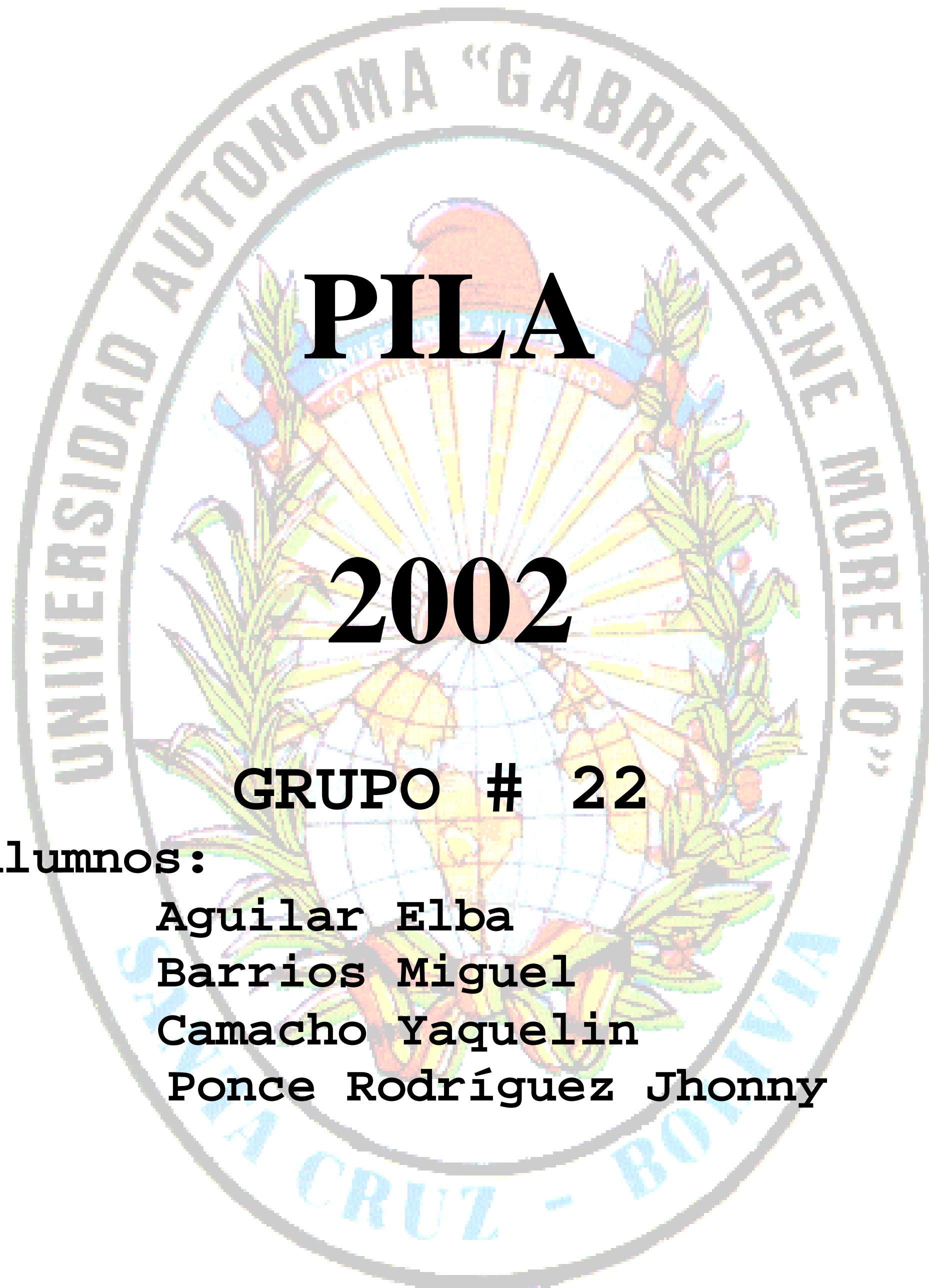


PILA

2002

GRUPO # 22

Alumnos:

Aguilar Elba
Barrios Miguel
Camacho Yaquelin
Ponce Rodríguez Jhonny

ESTRUCTURAS DE DATOS

TEMA 3

Estructura de datos Pila. Punteros

ÍNDICE

3.1. El tipo Pila....3
  3.1.1. Definición y ejemplos 3
  3.1.2. El TAD Pila....5
  3.1.3. Implementación estática mediante vectores....6
3.2. Gestión dinámica de la memoria....10
  3.2.1. Punteros....10
    Operaciones con punteros en Pascal....12
  3.2.2. Implementación dinámica del tipo Pila 14
  3.2.3. Comparación de las implementaciones....18
3.3. Ejercicios....19

BIBLIOGRAFÍA

- (Dale y Lilly, 1989), Cap. 3.
- (Joyanes y Zahonero, 1998), Cap. 7.
- (Joyanes y Zahonero, 1999), Cap. 6.
- (Horowitz y Sahni, 1994), Cap. 3.

OBJETIVOS

- Conocer el concepto, funcionamiento y utilidad del tipo Pila.
- Conocer el tipo abstracto de datos Pila y sus operaciones asociadas.
- Saber implementar el TAD Pila mediante el uso de una variable estática de tipo vector. Conocer las ventajas e inconvenientes de esta implementación.
- Recordar el funcionamiento de las variables dinámicas en Pascal mediante el uso de punteros.
- Saber implementar el TAD Pila mediante el uso de variables dinámicas. Conocer las ventajas e inconvenientes de esta implementación con respecto a la estática.

3.1. El tipo Pila

3.1.1. Definición y ejemplos

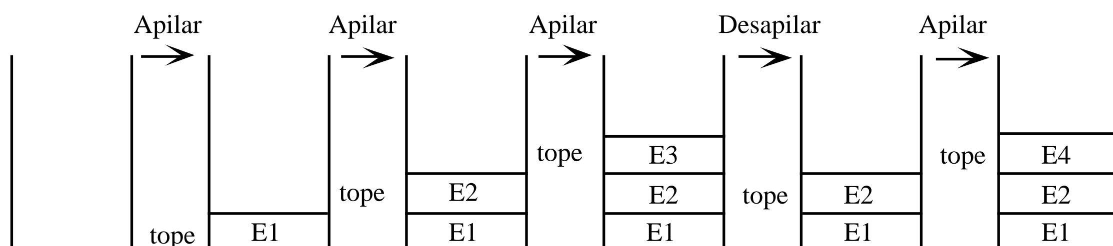
La Pila se caracteriza por ser una estructura de datos en la que el último elemento que se añade a la estructura es el primero en salir. Este modo de funcionamiento se conoce como política LIFO (Last In, First Out). En todo momento, el elemento que ocupa el extremo variable de la pila y su posición se denominan **tope**.

Podemos hacernos una imagen más gráfica, pensando en una pila de bandejas en una cafetería, una pila de platos en un fregadero, una pila de latas en un expositor de un supermercado: en cualquiera de estos ejemplos, los elementos se retiran y se añaden por un mismo extremo. En una pila de platos podríamos intentar retirar uno de los intermedios con el consiguiente peligro de derrumbe. Sin embargo, en una estructura de datos de tipo pila, esto no es posible. El único elemento de la pila que podemos retirar es el situado en el tope de la misma, y si queremos retirar otro, será necesario previamente haber borrado todos los situados "por encima" de él.

Sobre una pila podemos realizar dos operaciones básicas de manipulación:

- **Apilar**: añade un elemento en la posición siguiente a la del tope actual.
- **Desapilar**: elimina el elemento situado en el tope.

Veamos mediante un ejemplo, como evoluciona el contenido de la estructura conforme aplicamos sucesivamente estas operaciones



En la figura anterior podemos observar que el valor del tope (posición del último elemento) varía si operamos sobre la pila (si añadimos o eliminamos elementos).

Si la pila está vacía, es decir, si no tiene ningún elemento, el tope no tiene un valor definido. Del ejemplo, también se desprende una propiedad interesante: la pila es una estructura de datos dinámica, ya que su tamaño varía cuando hacemos una operación (apilar o desapilar) sobre ella.

De hecho este comportamiento dinámico nos lleva a considerar, en teoría, que una pila no tiene restricciones de tamaño. En su definición no ocurre como en la definición de un vector, por ej., en la que debemos decir cuántos elementos vamos a almacenar como máximo. En la pila, simplemente se añaden valores según sea necesario, sin ninguna restricción en su cantidad. No obstante, cuando en apartados posteriores describamos las posibles implementaciones de esta estructura de datos, veremos que siempre existirá algún tipo de

restricción de tamaño en la práctica. En último termino, el tamaño de la pila estará limitado por el disponible para almacenar datos en memoria.

Las aplicaciones de esta estructura de datos serán aquellas en las que los datos se manejen de modo dinámico siguiendo una política LIFO. Por ejemplo, uno de los usos más comunes de la pila es el de almacenar la información de las sucesivas llamadas a subrutinas en un programa, lo que nos permitirá volver de las mismas tras su finalización.

```
     PROGRAM P            PROC A1();        PROC A2();        PROC A3();
       . . . .              . . . .           . . . .           . . . .
       . . . .              . . . .           . . . .           . . . .
r-1:  A1();         s-1:  A2();       t-1:  A3();               . . . .
  r:   . . . .        s:   . . . .      t:   . . . .            . . . .
       . . . .              . . . .           . . . .           . . . .
     END.                 END;              END;              END;
```

En el programa P, la instrucción r-1 es una llamada al procedimiento A1; sabemos que esta instrucción implica pasar a la ejecución de A1 y cuando éste finalice, pasar a ejecutar la instrucción siguiente, esto es, la r.

Por su parte, la instrucción s-1 de A1 supone la ejecución de una llamada a A2. Se pasa el control a A2 y cuando finalice, la instrucción a ejecutar en A1 será la que ocupa la posición s. Este proceso puede repetirse un número cualquiera de veces, y es necesario disponer de un mecanismo que nos permita ejecutar las instrucciones en el orden adecuado.

En la memoria del ordenador, el Sistema Operativo reserva una zona que se denomina pila que se maneja como una estructura de datos de este tipo, es decir, siguiendo una política LIFO. El uso de esta pila nos ayuda a almacenar la información necesaria para poder devolver el control de forma correcta después de cada llamada: cuando se realiza una llamada a un subprograma, además de otra información relacionada con los argumentos del mismo, se almacena en la pila la dirección de retorno. Cada vez que finalice la ejecución de un subprograma, en el tope de la pila tendremos la información necesaria para saber en que instrucción nos habíamos quedado cuando realizamos la llamada. Al volver del subprograma, eliminaremos del tope de la pila la dirección de retorno, lo que dejará como nuevo tope la información necesaria para volver al finalizar el subprograma actual. El siguiente gráfico representa en forma simplificada, la información que tendremos en la pila del sistema durante el desarrollo del programa del ejemplo anterior:

Tras llamar a A1	Tras llamar a A2	Tras llamar a A3	Al volver de A3	Al volver de A2
		t		
	s	s	s	
r	r	r	r	r

Al ser una estructura LIFO, la pila devuelve la información en orden contrario al de almacenamiento. En un proceso como el anterior, este es justo el orden que necesitamos.

Definición

Una PILA es una estructura ordenada y homogénea, en la que podemos apilar o desapilar elementos en una única posición llamada TOPE, siguiendo una política LIFO (Last In, First Out).

Cuando decimos que una pila es una estructura ordenada, queremos decir que sus elementos se sitúan siguiendo un cierto orden, no que estén ordenados en función de su valor. En otras palabras, para cada elemento, salvo el primero y el último, podemos hablar de un anterior y un siguiente.

Por otro lado, cuando decimos que se trata de una estructura homogénea, queremos decir que todos sus elementos son del mismo tipo. En principio no existe ninguna restricción sobre el tipo de los distintos componentes de una pila, y éstos pueden ser tanto simples (Enteros, Reales, ...) como compuestos (Registros, Vectores, ...).

3.1.2. El TAD Pila

Veamos cual es la especificación formal del tipo de datos abstracto pila:

TAD: pila

Operaciones:

`CrearPila: → Pila`

Crea una pila vacía.

`Apilar: Pila x tipo_base → Pila`

Dada una pila p y un valor e, del tipo base, devuelve una nueva pila formada al apilar en p el nuevo elemento sobre la posición indicada por el valor del tope.

`Desapilar: Pila → Pila`

Dada una pila, elimina el elemento indicado por el valor del tope y devuelve la nueva pila.

`Tope: Pila → tipo_base`

Devuelve el valor del elemento que está apuntado por el tope.

`Pilavacia: Pila → logico`

Devuelve el valor verdadero si la pila está vacía y falso en caso contrario.

Axiomas: ∀ S ∈ Pila, ∀ e ∈ tipo_base,

```
1)  Pilavacia (CrearPila) = Verdadero
2)  Pilavacia (Apilar (S, e)) = Falso
3)  Desapilar (CrearPila) = error
4)  Desapilar (Apilar (S, e)) = S
5)  Tope (CrearPila) = error
6)  Tope (Apilar (S, e)) = e
```

Como en otros TAD's, los axiomas describen el comportamiento de las operaciones en los distintos casos y la relación entre las mismas, es decir, confieren una semántica al tipo definido. En particular los axiomas 4 y 6 definen el comportamiento LIFO de la pila.

3.1.3. Implementación estática mediante vectores

Puesto que la pila es una estructura lineal y ordenada, una opción directa y sencilla es recurrir a los vectores para su implementación. Sin embargo, hay que atenerse a las diferencias existentes entre el tipo vector y el tipo pila:

- En un vector puede accederse a cualquiera de sus elementos. En una pila sólo puede accederse a uno de sus elementos, el situado en el tope.
- Un vector es un conjunto finito, mientras en una pila no hay condiciones de límite de elementos.

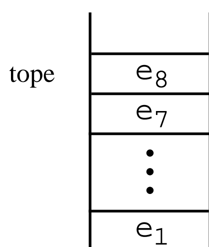
Para obviar estas diferencias, la implementación mediante vectores debe hacerse teniendo en cuenta que,

- Además del vector se necesita otro dato que indique cuál es en cada momento el elemento tope de la pila en el vector.
- Si se implementa la pila mediante vectores, se está restringiendo el concepto de pila. Esto es así, porque al tener los vectores un tamaño constante y predeterminado, en la propia definición de la pila se impone un máximo al número de elementos que va a tener.

De acuerdo a la primera consideración anterior, es posible utilizar las siguientes implementaciones del tipo pila bajo el soporte del tipo vector:

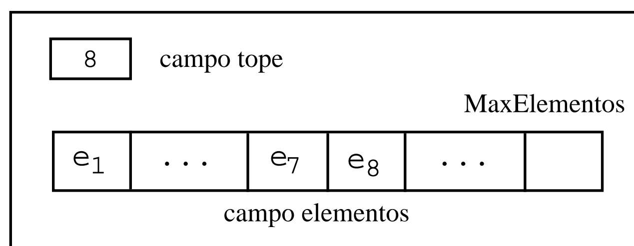
1. Pila, como vector de n elementos y una variable global tope.
2. Pila, como un registro con dos campos, uno de ellos el vector y otro el tope.

Descartaremos la primera de las soluciones porque hace uso de dos variables distintas para definir una única estructura de datos pila. Es mejor asociar al tipo definido una única entidad que contenga tanto los elementos de la pila como la referencia a su tope. De este modo además, nos evitamos posibles problemas cuando queramos utilizar varias pilas y, por lo tanto, varios topes.



Tipo abstracto

Registro TipoPila



Implementación mediante un vector

La estructura de datos en Pascal necesaria para implementar el TAD Pila siguiendo el esquema anterior será la siguiente

```
CONST
  MaxElemen = ...{nº máximo de elementos en la pila}
TYPE
  TipoPila = RECORD
                elementos : ARRAY [1..MaxElemen] OF TipoBase;
                tope : 0..MaxElemen
             END;
VAR
  s: TipoPila;
```

Con esta definición del tipo Pila las operaciones asociadas que hemos especificado en el apartado anterior quedarían del siguiente modo:

```
PROCEDURE CrearPila (VAR s: TipoPila);
BEGIN
  s.tope := 0
END;
```

Este procedimiento crea una pila vacía y hace que el tope de la misma valga 0. Dado que el elemento 0 del vector no existe, el tope no está definido.

```
FUNCTION PilaVacia (s: TipoPila): BOOLEAN;
BEGIN
  PilaVacia := s.tope=0
END;
```

La función `PilaVacia` nos dice si la pila está vacía. Con la implementación utilizada, esta condición equivale a decir que el tope vale 0.

```
PROCEDURE Desapilar (VAR s: TipoPila; VAR error: BOOLEAN);
BEGIN
  IF PilaVacia(s) THEN
    error := true
  ELSE
  BEGIN
    error := false;
    s.tope := (s.tope) - 1
  END
END;
```

El procedimiento `Desapilar` elimina el elemento situado en el tope de la pila. Para ello simplemente hace que el campo tope del registro apunte al elemento inmediatamente anterior. Para poder implementar el caso dado por el axioma 3 del TAD utilizamos un argumento adicional en la cabecera del procedimiento. Este argumento de tipo lógico, al que denominamos `error`, será verdadero cuando intentemos desapilar un elemento de una pila vacía.

Démonos cuenta que el procedimiento no elimina el elemento del vector, ni lo sobreescribe con otro valor, simplemente hace que el tope se encuentre por debajo de él. Dado que el acceso a la pila tan sólo debe realizarse mediante las operaciones definidas en el TAD, esta operación equivale a desapilar el elemento, ya que lo hace inaccesible.

```
PROCEDURE Tope (s: TipoPila; VAR e: TipoBase; VAR error: BOOLEAN);
BEGIN
  IF PilaVacia(s) THEN
    error := true
  ELSE
  BEGIN
    error := false;
    e := s.elementos[s.tope]
  END
END;
```

El procedimiento `Tope` devuelve el valor del elemento situado en el tope de la pila sin desapilarlo de la misma. Para acceder al mismo simplemente devuelve el elemento apuntado por el campo tope en el campo elementos. Al igual que ocurre con la operación `Desapilar`, es necesario utilizar un argumento de tipo lógico que devuelva el valor verdadero cuando intentemos conocer el tope de una pila vacía.

¿Podría implementarse la operación anterior como una función?

```
PROCEDURE Apilar ( VAR s: TipoPila; e: TipoBase);
BEGIN
  s.tope := (s.tope) + 1;
  s.elementos[s.tope] :=  e
END;
```

Si seguimos estrictamente la especificación de la operación `Apilar` en el TAD, su implementación correcta es la anterior. Sin embargo, si bajamos del nivel de definición abstracto al de implementación mediante un vector, se nos plantea el problema del tamaño del mismo. Tal y como hemos dicho antes, el tamaño del vector utilizado para contener los elementos de la pila debe definirse en tiempo de compilación, y durante su utilización éste constituirá una limitación al número de elementos que podemos apilar. Aunque por definición, una pila no tiene un límite en el número de sus elementos, al utilizar un vector para implementarla este límite existe en la práctica.

Por lo tanto, el uso de un vector en la implementación nos obliga a contemplar en las operaciones la posibilidad de que se llene la pila. Este hecho se producirá en la operación `Apilar` cuando se llene el vector, es decir, cuando el tope apunte al último de sus elementos. En este caso, una opción posible es devolver un error. Para ello necesitamos añadir un argumento adicional de tipo lógico al procedimiento, lo que nos llevará a la siguiente versión del mismo:

```
PROCEDURE Apilar ( VAR s:TipoPila; e:TipoBase; VAR error:BOOLEAN);
BEGIN
  IF (s.tope = MaxElementos) THEN
    error := true
  ELSE
  BEGIN
    error := false;
    s.tope := (s.tope) + 1;
    s.elementos[s.tope] := e
  END
END;
```

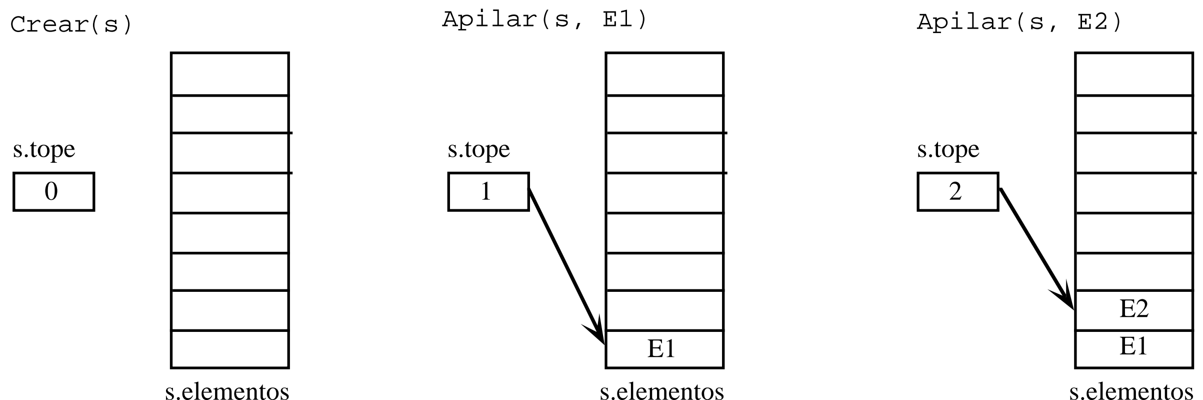
La condición de pila llena se producirá cuando el tope apunte al último de los elementos del vector, es decir, cuando valga MaxElementos. Otra posibilidad, que nos permite un mayor grado de modularidad, es la de introducir la operación `PilaLlena`:

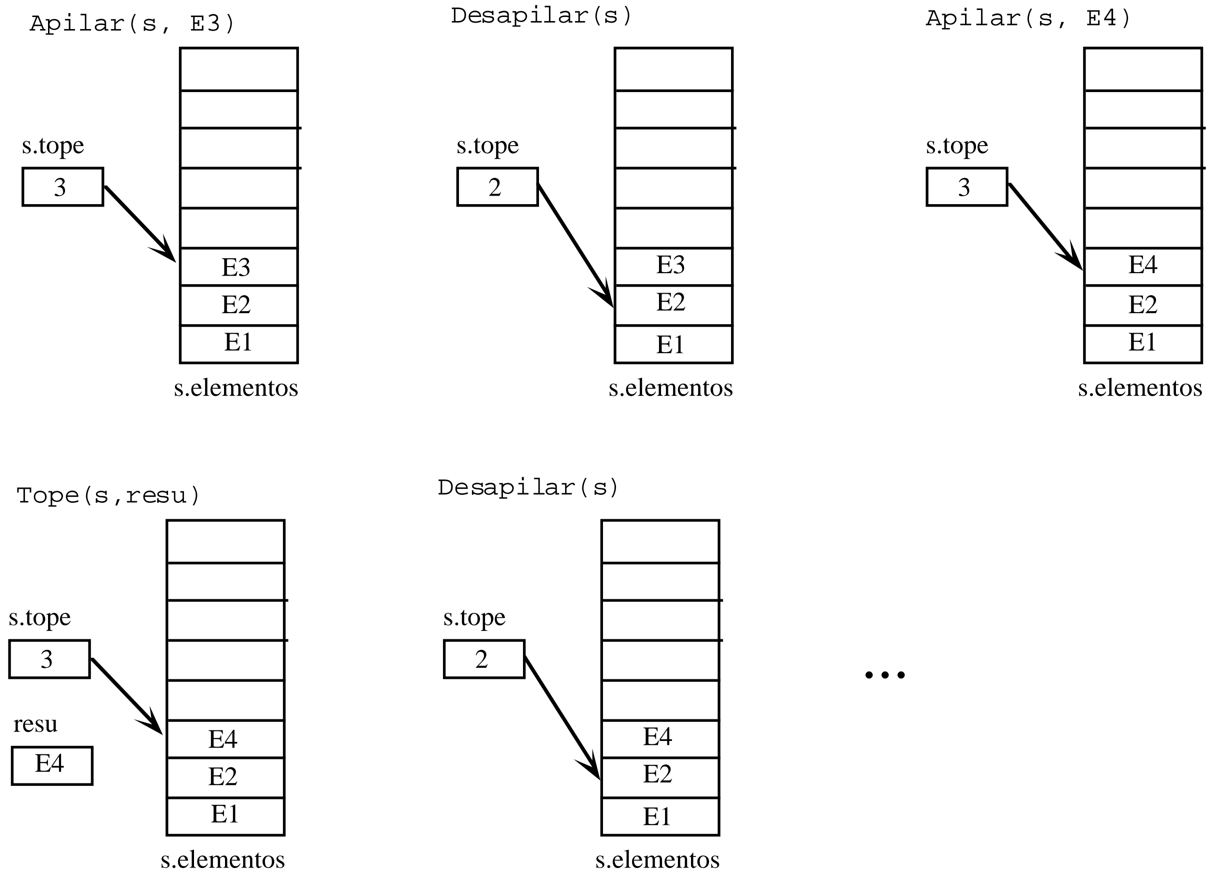
```
FUNCTION PilaLlena (s: TipoPila): BOOLEAN;
BEGIN
  PilaLlena := s.tope=MaxElementos
END;
```

y modificar la operación `Apilar` del siguiente modo:

```
PROCEDURE Apilar ( VAR s:TipoPila; e:TipoBase; VAR error:BOOLEAN);
BEGIN
  IF PilaLlena(s) THEN
    error := true
  ELSE
  BEGIN
    error := false;
    s.tope := (s.tope) + 1;
    s.elementos[s.tope] := e
  END
END;
```

Veamos un ejemplo de manipulación de una pila utilizando esta implementación.





El coste temporal de todas las operaciones descritas es O(1). Esto significa que su coste es independiente del tamaño de la pila.

3.2. Gestión dinámica de la memoria

3.2.1. Punteros

Como hemos comentado anteriormente, al implementar la pila mediante un vector, nos encontramos con el problema de tener que forzar un tamaño máximo de la estructura. Este problema surge del hecho de estar utilizando una estructura estática, como es el vector, para implementar una estructura dinámica, como es la pila. El vector debe tener un tamaño definido en tiempo de compilación, y este tamaño no cambia durante toda la ejecución del programa. Este comportamiento es contrario al de la estructura pila, cuyo tamaño varía mientras es utilizada conforme se le añaden y borran elementos.

Además de este problema, el uso de vectores hace que la pila pueda llenarse, hecho que va en contra de la propia definición de la estructura. No sólo eso, sino que si queremos evitar que este problema se produzca, tendremos que reservar un vector de gran tamaño para guardar todos los elementos posibles que vaya a contener la pila. Esto hace que en la mayor parte de los casos tan sólo utilicemos realmente una pequeña parte del vector, con el consiguiente desaprovechamiento de la memoria reservada.

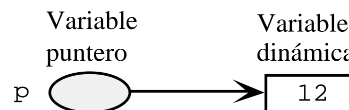
En definitiva, la mejor solución para implementar una estructura dinámica como es la pila es utilizar memoria dinámica. Al usar memoria dinámica, no reservamos una zona de memoria para la pila en tiempo de compilación, sino que vamos reservando espacio adicional

conforme lo vamos necesitando durante la ejecución del programa. Para lograr esto la mayor parte de los lenguajes, y entre ellos el Pascal, nos proporcionan un tipo de datos un tanto especial: el **puntero**.

Definición

El tipo puntero es un tipo simple (como entero, real o lógico), tal que una variable de tipo puntero contiene la dirección de memoria de otra variable. Se dice entonces que la variable puntero apunta a otra variable.

◊ **Ejemplo**



En la figura anterior representamos una variable puntero p que contiene la dirección de una variable dinámica cuyo valor es 12. Diremos que el puntero apunta a la variable dinámica.

Con un puntero se puede crear de forma dinámica una variable (se reserva espacio en memoria para la variable a la que apunta), usarla (utilizar la variable a la que hace referencia), y eliminar la variable (liberar el espacio de memoria que ocupa la variable referenciada).

◊ **Ejemplo**

```
PROGRAM tomaya;
TYPE
  TipoPuntero = ^INTEGER;
VAR
  p, q: TipoPuntero;   (* p, q: ^INTEGER *)
BEGIN
  new(p);
  q := p;
  p^ := 8;
  q^ := p^;
  dispose(p)  {p=NIL}
END.
```

En el ejemplo anterior, la declaración de una variable de tipo puntero (`p, q: TipoPuntero`) indica al compilador que reserve la memoria necesaria para almacenar una dirección de memoria. El compilador reserva el espacio necesario para contener ambos punteros, pero en este momento éstos no apuntarán a ninguna otra variable. Esto es así, porque al declararlas su contenido está indefinido como el de cualquier otra variable. Cuando hablamos de variables dinámicas, no nos estamos refiriendo a los punteros, que se declaran del mismo modo que cualquier otra variable estática, sino que nos referimos a las variables apuntadas por estos.

Démonos cuenta además, de que una variable puntero no puede apuntar a cualquier tipo de variable, sino sólo a las de un determinado tipo base. En el ejemplo anterior, los punteros `p` y `q` tienen como tipo base el entero (INTEGER). Se trata de punteros a enteros, y no podrán apuntar a variables de otro tipo (reales, lógicas, vectores, etc.)

Por otro lado, la declaración de un puntero no supone la creación de ninguna de las variables del tipo base (entero en este caso). Es necesario ser muy cuidadosos a la hora de operar con punteros y hacerlo a través de operaciones bien definidas por el lenguaje.

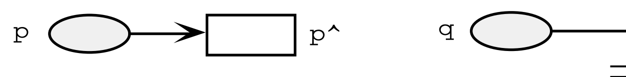
Operaciones con punteros en Pascal

Creación de una variable dinámica

Para crear una nueva variable dinámica utilizaremos la operación **`new(p)`**. Esta operación reserva el espacio necesario para una variable del tipo apuntado por `p` (en este caso un entero). Asimismo, asigna al puntero `p` la dirección de la nueva variable, con lo que a partir de ese momento p apuntará a la variable dinámica.

En el ejemplo anterior, cuando declaramos las variables de tipo puntero a entero `p` y `q`, estas no apuntan a ninguna dirección en particular.

Para hacer que apunten a una variable dinámica es necesario reservar el espacio para la misma, y para ello utilizaremos la operación `new(p)`. En este ejemplo concreto reservamos espacio para una variable de tipo entero y hacemos que `p` contenga su dirección, es decir, que apunte a la nueva variable dinámica.

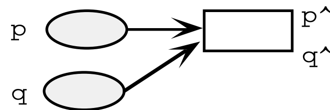


A partir de este momento disponemos de una nueva variable dinámica de tipo entero que podremos tratar como a cualquier otra variable de este tipo. Si embargo, esta variable no tendrá un nombre, sino que para referirnos a ella utilizaremos la expresión `p^`, donde `p` será el nombre de la variable puntero que la apunta.

Asignación de valor a punteros y a variables dinámicas

En Pascal estándar existen dos formas de asignar un valor a un puntero. O bien asignamos a un puntero el valor de otro del mismo tipo, o bien, cuando queremos indicar que un puntero no apunta a ninguna variable, le asignaremos el valor constante predefinido NIL.

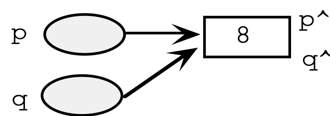
En el ejemplo anterior, cuando realizamos la asignación `p := q`, estamos asignando a `p` la dirección contenida en `q`. A partir de este momento ambas variables apuntarán a la misma dirección, y por tanto a la misma variable dinámica.



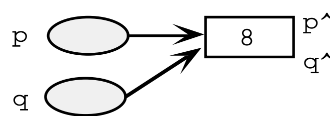
Un efecto adicional de esta asignación, es que a partir de este momento nos podremos referir a la variable dinámica, como `p^` y como `q^`. En ambos casos nos estaremos refiriendo al mismo espacio en memoria ocupado por un entero.

Es muy importante no confundir la asignación de valor a un puntero con una asignación referida a la variable apuntada. Como hemos dicho anteriormente, la variable apuntada se comporta como cualquier otra variable de tipo entero. En el ejemplo anterior realizamos dos asignaciones sobre la variable apuntada:

Después de `p^ := 8`



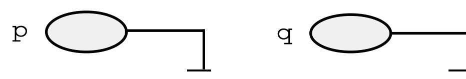
Después de `q^ := p^`



Después de realizar la primera de las asignaciones, la variable apuntada tomará el valor entero 8. Después de la segunda de las asignaciones, la situación no cambia. Esto es así, porque tanto `p^` como `q^` apuntaban a la misma variable dinámica.

<u>Liberación del espacio de una variable dinámica</u>

En el caso de las variables estáticas con las que hemos tratado hasta este momento, el espacio que utilizan es reservado por el compilador y permanece así durante toda la ejecución del programa. Sin embargo, en el caso de las variables dinámicas, al igual que podemos reservar espacio para las mismas durante la ejecución del programa, también podemos liberar este espacio. Para ello utilizaremos la operación **`dispose(p)`**. Al aplicar esta operación sobre el puntero `p`, se libera el espacio asignado a la variable apuntada. Ahora `p` no apunta a nada y el espacio que ocupaba la variable dinámica queda disponible para que el gestor de memoria lo vuelva a utilizar. Según la implementación de la operación `dispose`, el puntero `p` puede quedar con un valor NIL o con un valor indefinido.



Curiosamente, al liberar el espacio apuntado por la variable `p`, y dado que `q` apunta al mismo espacio, el valor de esta última variable también cambiará del mismo modo. Así, como efecto lateral de aplicar el `dispose` sobre una variable, otra también se ve modificada. Si a partir de este momento intentamos acceder a la variable dinámica, tanto mediante `p^` como mediante `q^`, se producirá un error de ejecución. Este hecho es suficientemente significativo como para que debamos ser especialmente cuidadosos cuando trabajemos con punteros.

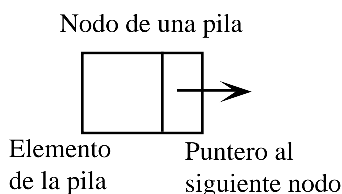
3.2.2. Implementación dinámica del tipo Pila

Para realizar la implementación de una pila utilizando memoria dinámica debemos tener en cuenta las siguientes consideraciones:

- Sólo podemos acceder a un elemento de la pila, el tope, es decir, la estructura sólo es visible a través del puntero a la posición de memoria que ocupa el tope. De acuerdo con esto, la definición de la pila se identifica con un puntero:

```
TipoPila = ^ElemPila;
```

- Es una estructura ordenada, por lo que se necesita un mecanismo que una entre sí los elementos que la forman sin perder el orden. Según esto, cada elemento será realmente un registro cuyos campos son:
  - El valor del elemento de la pila (de TipoBase).
  - Un puntero al siguiente nodo de la pila (contiene la dirección de memoria del siguiente nodo de la pila).

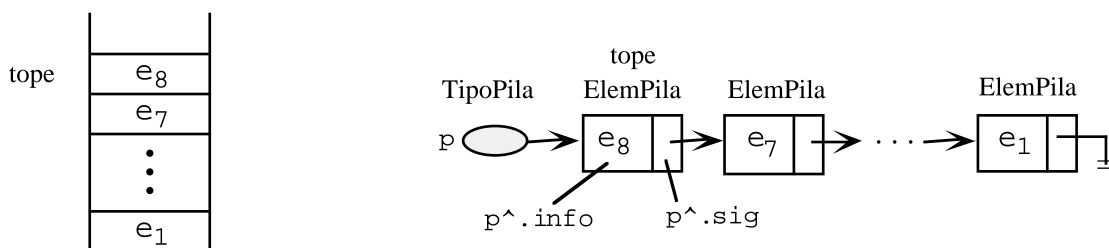


La declaración del tipo pila en Pascal queda como sigue:

```
TYPE
  TipoPila = ^ElemPila;
  ElemPila = RECORD
               info: TipoBase;
               sig: TipoPila
             END;
VAR
  p: TipoPila;
```

La definición anterior es recursiva, ya que la definición de `TipoPila` depende de la de `ElemPila`, y la definición de este último depende de la definición de `TipoPila`, pues lo utiliza en uno de sus campos. Debido a este hecho, en ocasiones este tipo de estructura, y otras que veremos en temas posteriores (colas, árboles, ...), se denominan estructuras de datos recursivas.

Utilizando esta implementación, la estructura de datos pila será como la representada en la siguiente figura:



Tipo abstracto Implementación mediante variables dinámicas

Utilizando la definición anterior del tipo pila, veamos cual sería la implementación de las operaciones que hemos asociado a este tipo abstracto de datos en el apartado 3.1.2.

```
PROCEDURE CrearPila (VAR p: TipoPila);
BEGIN
  p := NIL
END;
```

La operación `CrearPila` devuelve una variable de tipo `TipoPila`. En el momento de crear una pila el valor de su tope está indefinido. Dado que en este caso, la pila se identifica con un puntero que apunta a su tope, asignamos el valor NIL a este puntero.

```
FUNCTION PilaVacia (p: TipoPila):BOOLEAN;
BEGIN
  PilaVacia := (p=NIL)
END;
```

La pila estará vacía cuando no contenga ningún elemento, es decir, cuando su tope no apunte a ningún elemento. Con esta implementación está condición se dará cuando su tope valga NIL.

```
PROCEDURE Tope (p: TipoPila; VAR e: TipoBase; VAR error: BOOLEAN);
BEGIN
  IF PilaVacia(p) THEN
    error := true
  ELSE
  BEGIN
    error := false;
    e := p^.info
  END
END;
```

Sabemos que la variable de tipo `TipoPila`, se identifica con un puntero a su tope, es decir, apunta al nodo de la pila situado en su tope. Así pues, el elemento situado en el tope de la pila será el contenido en el campo `info` de este nodo: `p^.info`.

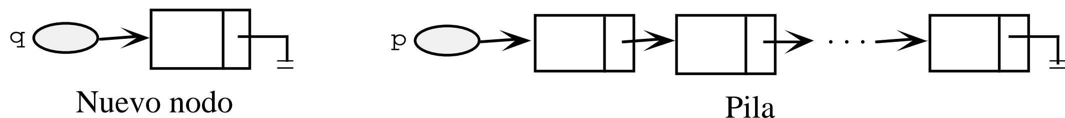
Al tratar de acceder al tope de una pila hemos de tener en cuenta la posibilidad de que este vacía. Si se produce esta condición, la operación `Tope` devuelve el valor verdadero en un

argumento de tipo lógico denominado `error`. Si la pila no está vacía este argumento tomará el valor falso.

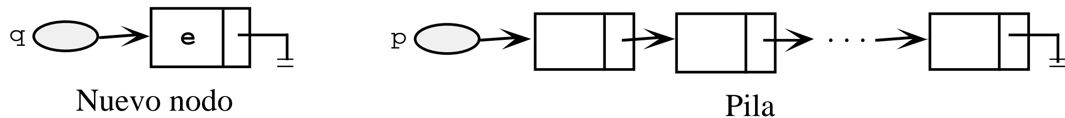
```
PROCEDURE Apilar ( VAR p: TipoPila; e: TipoBase);
VAR
   q: TipoPila;
BEGIN
   new(q);
   q^.info := e;
   q^.sig := p;
   p := q
END;
```

Para apilar un nuevo elemento en la pila, lo hacemos "sobre" su tope, y este nuevo elemento pasa a ser un nuevo tope. Para implementar esta operación recorremos los siguientes pasos:

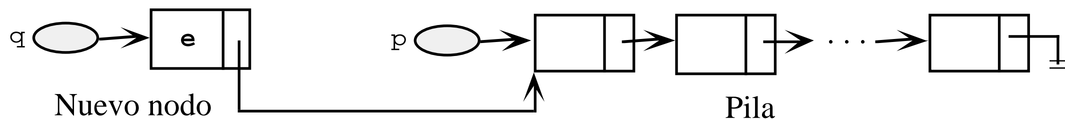
- En primer lugar reservamos espacio para un nuevo nodo en la pila mediante la operación `new(q)`. Para poder realizar esta operación hemos definido una variable local auxiliar `q` de tipo `TipoPila` que apuntará al nuevo nodo durante su proceso de creación.



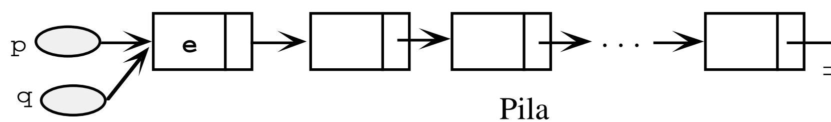
- En segundo lugar asignamos el valor al nuevo elemento. Para ello asignamos el valor pasado como parámetro de entrada al campo que contendrá la información en el nuevo nodo: `q^.info := e`.



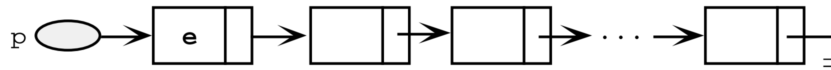
- En tercer lugar colocamos el nuevo nodo "sobre" el tope anterior. Para lograr esto, hacemos que el campo que apunta al siguiente en el nuevo nodo, apunte al tope anterior de la pila: `q^.sig := p`.



- Finalmente hacemos que el nuevo nodo sea el tope de la pila, y para ello hacemos que la variable que apunta al tope de la misma, `p`, apunte al nuevo nodo: `p := q`.



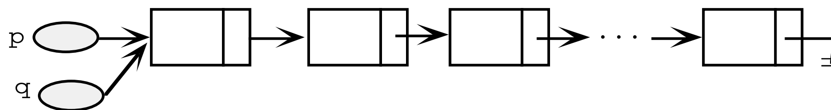
Así pues, hemos añadido un nuevo elemento a la pila. Este nuevo elemento tiene el valor `e` y actúa como nuevo tope de la misma. Dado que `q` es una variable local al procedimiento `Apilar`, al finalizar el mismo se liberará el espacio que ocupa y la pila quedará como:



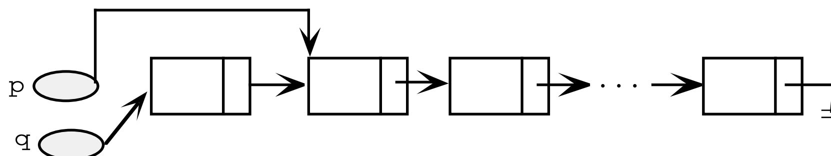

```
PROCEDURE Desapilar (VAR p: TipoPila; VAR error: BOOLEAN);
VAR
   q: TipoPila;
BEGIN
   IF PilaVacia(p) THEN
      error := true
   ELSE
   BEGIN
      error := false;
      q := p;
      p := p^.sig;
      dispose(q)
   END
END;
```

Para desapilar el elemento que se encuentra en el tope de la pila podríamos pensar que basta con liberar el espacio que ocupa mediante una operación `dispose` sobre la variable `p`. Sin embargo, si solamente hiciésemos esto, el puntero al tope quedaría con un valor indefinido y perderíamos la única referencia que tenemos a los elementos de la pila. Con el fin de evitar este problema, desarrollamos los siguientes pasos:

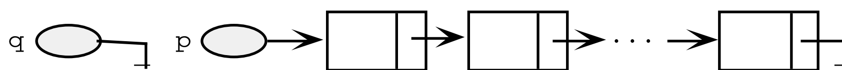
- En primer lugar utilizamos una variable local auxiliar `q` de tipo `TipoPila` y hacemos que esta apunte también al tope de la pila: `q: = p`.



- A continuación hacemos que el tope de la pila pase a apuntar al siguiente elemento de la misma: `p : = p^.sig`.



- Finalmente, liberamos el espacio ocupado por el tope actual utilizando para ello la variable auxiliar `q`: `dispose(q)`.



La variable auxiliar `q` quedará con un valor indefinido, pero al finalizar el procedimiento `Desapilar`, dado que es una variable local al mismo, se liberará el espacio que ocupa en memoria.

Al igual que ocurre con el procedimiento `Tope`, al intentar desapilar un elemento de la pila hemos de considerar la posibilidad de que se encuentre vacía. En este caso devolveremos un valor verdadero en un argumento adicional `error` de tipo lógico, mientras que si la pila no está vacía devolveremos falso sobre este argumento.

3.2.3. Comparación de las implementaciones

En este apartado compararemos la implementación de la pila mediante memoria dinámica con la que hemos descrito en el apartado 3.1.3. que utilizaba vectores.

En primer lugar cabe recordar que, dado que estamos tratando con un tipo abstracto de datos dinámico, es más adecuado utilizar para su implementación una estructura de datos dinámica que una estática. En este sentido se encuentra más acorde con la definición lógica del tipo pila el uso de una implementación mediante variables dinámicas que la utilización de un vector estático.

El razonamiento anterior se concreta en los siguientes aspectos. Por un lado, cabe decir que la complejidad temporal de las operaciones de gestión de la pila es de O(1) con ambas implementaciones. Sin embargo, esto no es así si hablamos de la complejidad espacial de las mismas, esto es, del espacio de memoria requerido en cada caso.

En el caso de la implementación mediante vectores o estática, hay que declarar un tamaño máximo para el tipo Pila. Siempre tenemos que definir un tamaño mayor que el que necesitamos para prevenir desbordamientos, en consecuencia la memoria queda infrautilizada prácticamente siempre. Aún en este caso, es necesario contemplar la posibilidad de que se llene la pila. Este hecho hace que en la operación `Apilar` debamos estudiar la condición de `PilaLlena` y dar lugar a un error en el caso de que sea verdadera. Démonos cuenta de que con la implementación mediante variables dinámicas la operación `Apilar` no contempla este caso.

En el caso de una implementación mediante variables dinámicas, la pila, y con ello el tamaño que ocupa en memoria, crece según las necesidades del programa durante su ejecución. El tamaño máximo de la pila depende de la memoria disponible en la máquina. No obstante, para almacenar cada nodo se necesita más espacio que en la representación estática, ya que se guarda la dirección del nodo siguiente, además de la información pertinente.

3.3. Ejercicios

1. Se desea implementar dos pilas, y se dispone de un solo vector de N componentes. Implementar ambas pilas de manera que se pueda aprovechar al máximo el vector. Las operaciones de pila tendrán que llevar un parámetro adicional que indique sobre qué pila se quiere realizar la operación (1 ó 2).

   (Ayuda: Las dos pilas crecen la una hacia la otra.)

2. Utilizando sólo operaciones de pila, realizar un algoritmo que:
   a) Invierta una pila
   b) Dada una pila y un valor umbral, devuelva otras dos pilas, en una de las cuales se han introducido los valores menores que el umbral, y en la otra los valores mayores o iguales que el umbral. (La pila inicial desaparece).
   c) Cambie todos los elementos de la pila iguales a X por Y, sin modificar el resto de la pila.

3. Implementar una pila de enteros en un vector de 0 a 100 elementos, donde el elemento 0 se emplea como tope de la pila, y el resto de elementos se emplean para albergar los elementos de la pila.

4. Un estacionamiento de coches contiene una sola línea, la cual puede guardar hasta 10 coches. Existe solamente una entrada/salida al estacionamiento en uno de los extremos de la línea. Si un usuario llega a retirar su coche que no está cerca de la salida, todos los coches que están bloqueando su salida son retirados, el coche del usuario sale, y los otros coches vuelven al sitio en que estaban originalmente, dejando el hueco en la parte más cercana a la salida.
   a) Define los tipos de datos necesarios para resolver el problema anterior.
   b) Escribe un algoritmo que procese un grupo de líneas de entrada. Cada línea de entrada contiene una 'e' para llegada o 's' para salida, y el número de matrícula. Se asume que los coches llegan y salen en el orden especificado en la entrada. El programa debe imprimir un mensaje cada vez que llega o sale un coche. Cuando un coche llega el mensaje debe especificar si hay o no espacio para el coche en el estacionamiento. Si no hay espacio, el coche no podrá entrar al estacionamiento. Cuando un coche ha estado dentro del estacionamiento y sale, el mensaje debe incluir el número de veces que fue movido el coche para permitir que otros coches salieran.

5. Observando el siguiente algoritmo, se pide contestar a las preguntas que siguen:

```
PROCEDURE VaciarPila (VAR P: TipoPila);
BEGIN
  IF NOT PilaVacia (P) THEN
  BEGIN
    Desapilar (P, error);
    VaciarPila (P);
  END
END;
```

   a) ¿Finaliza la ejecución de este algoritmo?

b) ¿Cuál es el resultado de su ejecución? (esto es, ¿qué hace?)

c) Escribe un algoritmo iterativo que haga lo mismo.

6. Observando el siguiente algoritmo, se pide contestar a las preguntas que siguen:

```
PROCEDURE CuentaElemsPila (VAR P: TipoPila; VAR n: INTEGER);
VAR
  x: TipoBase;
BEGIN
  IF (NOT PilaVacia(P)) THEN
  BEGIN
    Tope (P, x, error);
    Desapilar (P, error);
    CuentaElemsPila (P, n);
    Apilar (P, x, error);
    n := n+1;
  END;
END;
```

a) ¿Cuál es el error del algoritmo?

b) ¿Cómo lo solucionarías?

c) Una vez solucionado el error, y sabiendo que cuando termina el algoritmo los elementos de la pila no se han modificado, ¿por qué P es de entrada/salida?

d) ¿Se podría resolver de manera iterativa? ¿Cómo?

e) Escribe el algoritmo que resuelva el problema de manera iterativa.

7. Se utiliza, para implementar la estructura de datos Pila, vectores, que a lo sumo pueden contener MAXIMO elementos. (Siendo MAXIMO una constante que se supone definida).

a) Definir la estructura de datos necesaria para poder utilizar pilas con una capacidad máxima de 2*MAXIMO elementos.

b) Desarrollar los algoritmos que implementan las operaciones básicas sobre datos del tipo pila, según la estructura elegida en el apartado a).

8. Redefine la estructura PILA, utilizando memoria dinámica, de forma que el número de elementos que la componen sea una información disponible siempre, de forma actualizada. Redefine, atendiendo a la nueva estructura y aprovechándola al máximo, las operaciones básicas de manejo de pilas.

9. Un funcionario de Juzgados tramita expedientes, de forma que siempre va cogiendo de la bandeja de expedientes a tramitar el que se encuentra más arriba. Cada expediente se puede caracterizar por un código, el asunto del que trata, la fecha de expedición y la fecha de tramitación. Un ordenanza es el encargado de traerle nuevos expedientes, que va dejando encima de los que ya había en la bandeja.

   a) Definir la estructura de datos más adecuada para representar la bandeja de expedientes y la más adecuada para representar el cargamento de expedientes nuevos que trae el ordenanza.

   b) Definir la operación **apilar_exp**, cuyo resultado es añadir a la estructura bandeja de expedientes los datos almacenados en la estructura ordenanza.

   c) Como con este sistema es fácil que algún expediente se retrase mucho, de vez en cuando un juez pide que se tramite un determinado expediente urgentemente. Para ello indica el código de dicho expediente. Definir la operación **urgencia**, que permitirá buscar un determinado expediente en la bandeja y eliminarlo de ella para tramitarlo.

10. Un alumno guarda en un montón los boletines de apuntes de las 7 asignaturas en que se encuentra matriculado (codificadas de 1 a 7). En cada momento tan sólo puede acceder al boletín situado en la parte superior del montón. Por cada boletín conoce la asignatura, el tema, el número de páginas y su precio.

    a) Definir la estructura de datos más adecuada para guardar la información sobre los boletines.

    b) El alumno quiere clasificar los boletines por asignatura en montones separados.

       b.1) Definir la estructura de datos más adecuada para guardar la información sobre los boletines clasificados.

       b.2) Implementar un algoritmo que dado el montón original con los boletines de las 7 asignaturas, nos devuelva los boletines clasificados en montones separados.

    c) Implementar un algoritmo que, dados los boletines clasificados, calcule la asignatura en cuyos apuntes el alumno se ha gastado más dinero.