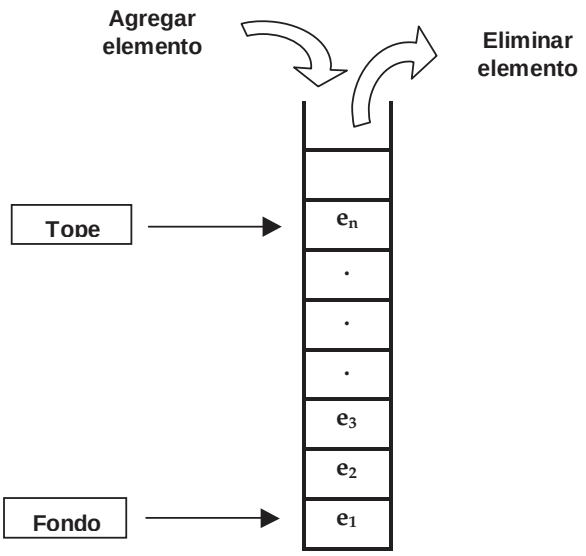
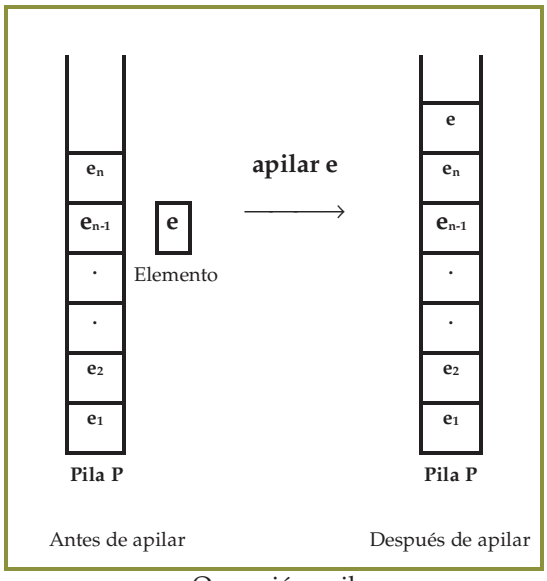
TALLER: TAD Pila

Una **pila** (*stack*) es un conjunto de elementos del mismo tipo que solamente puede crecer o decrecer por uno de sus extremos. Una pila también se la conoce con el nombre de estructura de tipo **LIFO** (*last in first out*), porque el último elemento en llegar es el primero en salir.

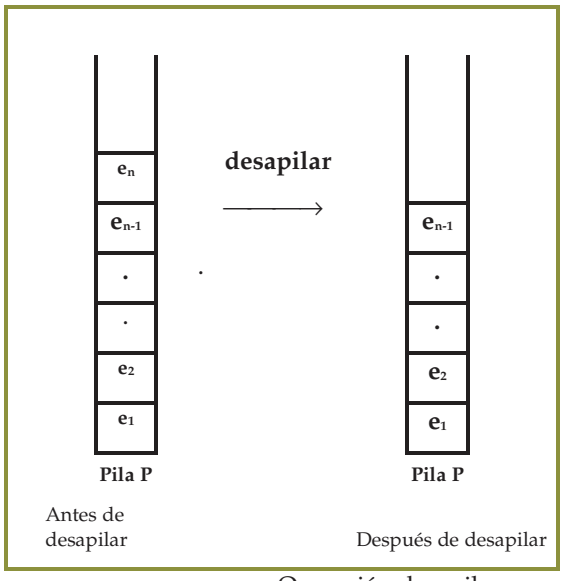
La representación gráfica de una pila es:



El último elemento agregado se denomina **tope** (*top*) y el primero **fondo** (*back*). El único elemento visible y por consiguiente al único que se puede acceder es al tope. A esta operación la denominaremos **recuperarTope** (*getTop*). Una pila sin elementos es una pila vacía que se representa con el símbolo ϕ, y una operación analizadora importante es saber si la pila está o no vacía. Las operaciones que modifican el estado de una pila son agregar un nuevo elemento a la pila y sacar el elemento agregado más recientemente. A la primera operación la denominaremos **apilar** (*push*) y a la segunda, **desapilar** (*pop*).



Operación apilar



Operación desapilar

Las pilas tienen muchas aplicaciones en informática. Por ejemplo se usan para:

- ✓ evaluar expresiones aritméticas

- ✓ analizar sintaxis
- ✓ simular procesos recursivos

Antes de estudiar estas aplicaciones veremos una especificación del TAD Pila. Las operaciones primitivas se aplican sobre un objeto **p** de tipo Pila.

Especificación del TAD Pila

Nombre del TAD: Pila

Representación:

Si la Pila *p* tiene por lo menos un elemento la representaremos como:

$$p = <e_1 ,e_2 ,e_3 ,. . . ,e_n>$$

siendo e_1 el fondo y e_n el tope.

Si la Pila *p* no tiene elementos, entonces:

$$p = \phi$$

Constructoras:

- **crearPila: -** → **Pila**
  - Construye una pila vacía, esto es solicita memoria para poder almacenar los elementos en la pila.
  - precondiciones: -
  - postcondciones: Pila $p = \phi$

Modificadoras:

- **apilar : Pila X Elemento** → **Pila**
  - Agrega a la Pila **p** un nuevo elemento *e*
  - precondiciones: Pila $p = <e_1 ,e_2 ,e_3 . . . ,e_n>$ y elemento *e* ó
    Pila $p = \phi$ y elemento *e*
  - postcondciones: Pila $p = <e_1 ,e_2 ,e_3 . . . ,e_n , e >$ ó
    Pila $p = <e>$

- **desapilar : Pila** → **Pila**
  - Saca de la pila **p** el elemento insertado más recientemente.
  - precondiciones: Pila $p \neq \phi$ o sea $p = <e_1 ,e_2 ,e_3 . . . ,e_n >$
  - postcondciones: Pila $p = <e_1 ,e_2 ,e_3 . . . ,e_{n-1} >$

Analizadoras:

- **recuperarTope: Pila** → **Elemento**
  - Recupera el valor del elemento que está en el tope.
  - precondiciones**:** Pila $p \neq \phi$ o sea $p = <e_1 ,e_2 ,e_3 . . . ,e_n >$
  - postcondciones: Elemento e_n

- **esVacia: Pila → booleano**
  - Informa si la Pila **p** está o no vacía.
  - precondiciones: Pila p
  - postcondiciones: VERDADERO si $p = \phi$; FALSO si $p \neq \phi$.

Destructora:

- **destruirPila : Pila → -**
  - Destruye la pila **p** liberando la memoria que ésta ocupaba.
  - precondiciones: Pila *p*
  - postcondciones: -

Prototipos de las funciones primitivas de Pila en lenguaje C

```
Pila crearPila();
void apilar(Pila p, Elemento e);
void desapilar(Pila p);
Elemento recuperarTope(Pila p);
int esVacia(Pila p);
void destruirPila(Pila p);
```

Ejemplo

El siguiente ejemplo muestra el uso de las operaciones primitivas del TAD Pila.

Enunciado: Invertir los elementos de una Pila.

Precondición: Pila p = $<e_1, e_2, e_3, \ldots, e_n>$

Postcondición: Pila q = $<e_n, e_{n-1}, \ldots, e_1>$

Código en C

```
Pila Invertir (Pila p) {
   Pila q = crearPila();
   while (! esVacia (p) ) {
       apilar (q, verTope(p);
       desapilar (p);
   }
   return q;
}
```

Ejercicios

📖 **Ejercicio 1**

Realizar el seguimiento en forma gráfica de las instrucciones que se dan a continuación. Suponer que ***Pila1*** es una pila de números enteros y las variables ***x***, ***y*** y ***z*** son tipo entero.

a. Crear Pila1; apilar el número 5; apilar el número 8; apilar el número3; mostrar el tope; desapilar; mostrar el tope; desapilar; mostrar el tope;

b. Crear Pila1; apilar el número 5; apilar el número 8; apilar el número3; retornar el tope en x; desapilar; retornar el tope en y; desapilar; apilar el valor de x; apilar el valor de y; retornar tope en z; desapilar;
c. Crear Pila1; apilar el número 5; apilar el número 8; apilar el número3; mientras no está la pila vacía hacer mostrar el tope y desapilar;

🕮 **Ejercicio 2**
Desarrollar los siguientes algoritmos, en el lenguaje que desee, utilizando sólo las operaciones primitivas de pila:

a. Imprimir el contenido de una pila de enteros sin cambiar su contenido.
b. Colocar en el fondo de una pila un nuevo elemento.
c. Calcular el número de elementos de una pila sin modificar su contenido.
d. Eliminar de una pila todas las ocurrencias de un elemento dado.
e. Intercambiar los valores del tope y el fondo de una pila.
f. Duplicar el contenido de una pila.
g. Verificar si el contenido de una pila de caracteres es un palíndrome.
h. Calcular la suma de una pila de enteros sin modificar su contenido.
i. Calcular el máximo de una pila de números reales.

🕮 **Ejercicio 3**
Escribir una función que utilizando las primitivas del TAD Pila reciba una pila de enteros y retorne dos pilas una con los números pares y otra con los impares.

🕮 **Ejercicio 4**
Construir un algoritmo en el que se manipulen dos pilas de enteros. La entrada de datos son pares de números (i, j) con $1 \leq |i| \leq 2$.

Si i es positivo indica que se debe insertar el elemento j en la Pila i.
Si i es negativo indica que se debe insertar el elemento j en la Pila –i
Si i es cero indica que se ha terminado el proceso.

Modificar el algoritmo anterior para admitir ternas de números enteros (i, j, k) donde i, j tienen el mismo significado y k es un número entero que puede tomar los valores –1 y 0 con el siguiente significado:

–1 indica vaciar la pila i.
0 proceso normal.

🕮 **Ejercicio 5 (Opcional)**
Se tiene un garage con una sola entrada y cuyo ancho es tal que sólo puede estacionar un auto detrás de otro.

Cuando llega un auto se coloca al final. Cuando se debe retirar un auto se estacionan provisoriamente en la vereda uno detrás de otro los que están delante de él. Los autos se identifican por la matrícula. Utilizando los TDA **Auto** y **Pila** escribir un programa que simule el funcionamiento del garage.

Aplicaciones de Pilas

Se peresentan a continuación tres aplicaciones del TAD Pila.

Balanceo de símbolos

Una de las funciones de un compilador es controlar el balanceo de símbolos, esto es QUE los corchetes, llaves y paréntesis derechos correspondan a su contraparte izquierda. La secuencia {[]} es correcta, en cambio {[}] es errónea. Por simplicidad sólo revisaremos el equilibrio de paréntesis, llaves y corchetes, ignorando cualquier otro carácter que aparezca.

Supongamos que queremos analizar el siguiente fragmento de programa fuente escrito en C.

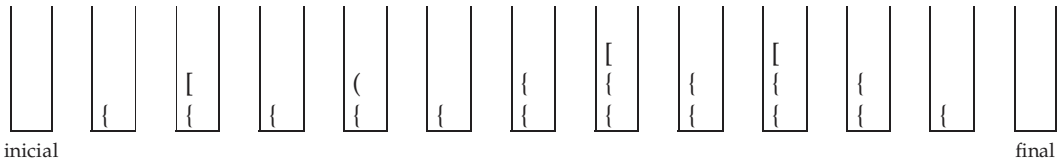
```
class estudiante {
   int  notas[2];
   char *nombre;

   void asignarNotas(int n1, int n2) {
        notas[1] = n1;
        notas[2] = n2;
   }
}
```

El algoritmo es el siguiente:

1. Crear una pila.
2. Leer el primer carácter.
3. Si el carácter es de apertura entonces apilarlo.
4. Si es de cierre y la pila está vacía se informa un error.
5. Si es de cierre y la pila no está vacía, recuperar el elemento que está en el tope de la pila. Si este elemento no es el correspondiente símbolo de apertura se informa un error, sino despilar.
6. Leer el próximo carácter y repetir los pasos 3, 4, 5 y 6 hasta el fin del archivo.

Estados de la pila a medida que avanza el algoritmo.

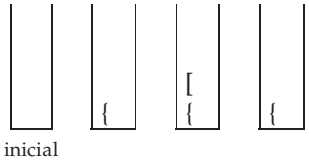


Si fuera el siguiente texto,

```
class estudiante {
   int  notas[2};
   char *nombre;

   void asignarNotas(int n1, int n2) {
        notas[1] = n1;
        notas[2] = n2;
   }
}
```

Estados de la pila a medida que avanza el algoritmo.



Se produce un error porque el [no es símbolo que corresponde a la }.

Expresiones aritméticas

Cualquier calculadora calcula correctamente la siguiente expresión:

2 * 5 + 9 =

Pero el resultado de

2 + 5 * 9 =

dependerá de la calculadora utilizada. Una calculadora simple dará el resultado erróneo de 63, mientras que otras mejores darán el resultado correcto de 47. Estas expresiones están escritas en notación ***infija***. En esta notación un operador se encuentra entre dos operandos:

operando ***operador*** *operando*

Para evaluar correctamente una expresión en notación infija es necesario conocer las reglas de prioridades de los operadores. Por ejemplo el * tiene mayor prioridad que el +. Otra forma de escribir las expresiones aritméticas es utilizando la notación ***postfija***, en la cual el operador va detrás de los dos operandos. En este caso no hay necesidad de saber ninguna regla de prioridad; ésta es una ventaja obvia.

operando operando ***operador***

En notación postfija las dos expresiones anteriores se escriben como:

2 5 * 9 + =

2 5 9 * + =

Todos sabemos evaluar una expresión aritmética que está en notación infija pero ¿cómo se evalúa una expresión que está en postfija? La forma más fácil es utilizar una pila.

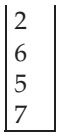
Para simplicidad supongamos que la expresión aritmética está formada por números de un dígito y los operadores +, -, * y /. Analizamos la expresión carácter por carácter; si el carácter que estamos analizando es un número, lo apilamos; en cambio, si es un operador desapilamos los dos últimos elementos de la pila y realizamos la operación entre ellos cuyo resultado se apila. Cuando la expresión finaliza la pila contiene un único elemento que es el resultado de la expresión.

Por ejemplo, la expresión postfija

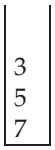
7 5 6 2 / 8 * + 3 + *

se evalúa como sigue:

los primeros cuatro números se apilan.

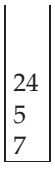


Cuando se lee el carácter / se desapilan el 2 y el 6. Se realiza la operación entre estos dos elementos, es decir **6 / 2** , y el resultado se apila.



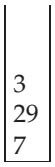
Luego se apila el 8.

Ahora aparece un *, así que se sacan el 8 y 3 y se apila 3 * 8 = 24.



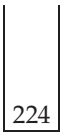
Después aparece un +, así que se sacan de la pila el 24 y 5 para apilar 5 + 24 = 29.

Ahora se apila el 3.



A continuación el + saca el 3 y 29 y apila 29 +3 = 32.

Por último aparece un * y se sacan 32 y 7 y se apila 7 * 32 = 224



No sólo se puede usar una pila para evaluar expresiones en notación infija, sino que una pila también puede servir para convertir una expresión en forma estándar (es decir en notación infija) en posfija.

Consideremos que queremos convertir la expresión en notación infija

a + b * c - d / e =

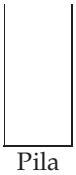
a notación postfija. Para esta transformación es necesario tener en cuenta la prioridad de los operadores. Los operadores * y / tienen mayor prioridad que los operadores + y -.

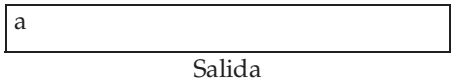
Cuando se lee un operando, se coloca en la salida. Cuando leemos un operador y la pila está vacía, éste se apila. Si la pila no está vacía, se desapilan y se van colocando en la salida todos los operadores que tienen prioridad mayor o igual al que estamos analizando, y luego éste se coloca en la pila.

Cuando leemos el símbolo = se desapilan todos los elementos que hayan quedado en la pila enviándolos a la salida.

A continuación se muestra como va quedando el estado de la pila y de la salida a medida que se analizan los caracteres de la expresión anterior.

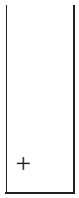
Carácter leido: a


Pila

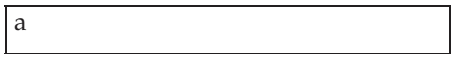


Salida

Carácter leido: +

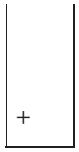


Pila

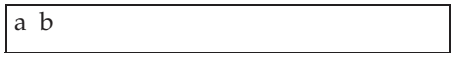


Salida

Carácter leido: b

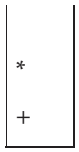


Pila

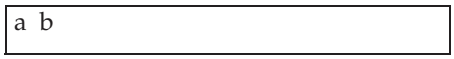


Salida

Carácter leido: *

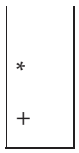


Pila

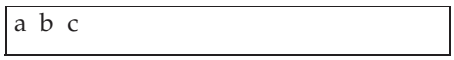


Salida

Carácter leido: c

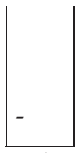


Pila

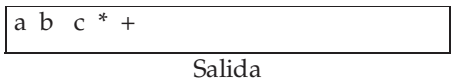


Salida

Carácter leido: -

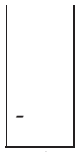


Pila

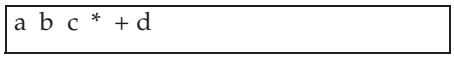


Salida

Carácter leido: d

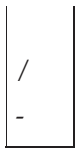


Pila

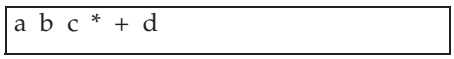


Salida

Carácter leido: /

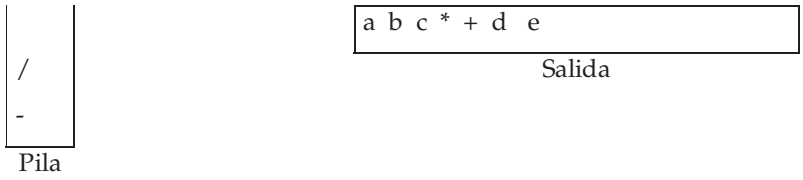


Pila

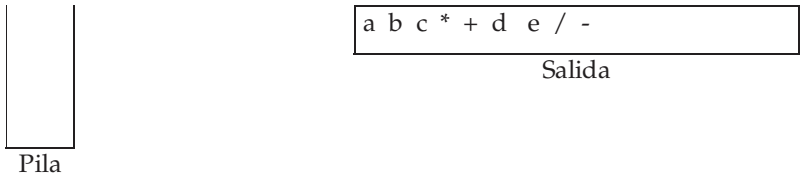


Salida

Carácter leido: e



Carácter leido: =



Si la expresión tiene paréntesis se procede de la misma manera. Los paréntesis izquierdo también se apilan. Si vemos un paréntesis derecho, sacamos los elementos de la pila y los pasamoa a salida hasta encontrar el correspondiente paréntesis izquierdo, el cual se saca de la pila pero no se envía a salida.

Llamadas a funciones

Cuando un programa hace una llamada a un procedimiento o función, el sistema debe guardar el estado de sus variables y con que instrucción debe continuar cuando el procedimiento termine. Si esta función llama otra debe hacer lo mismo. Cuando termina la segunda función se debe continuar con la primera y cuando termina ésta con el programa. Como se ve el compilador debe utilizar una pila con la información necesaria.

Ejercicios

🕮 **Ejercicio 6**
Utilizando las primitivas de Pila, pasar a postfijo las siguientes expresiones:

a. 1 + 2 - 3 * 4
b. 1 * 2 - 3 * 4
c. 1 + 2 * 3 - 4 * 5 + 6
d. (1 + 2) * 3 - (4 * (5 - 6))

🕮 **Ejercicio 7**
Sean ***u***, ***v***, ***w***, ***x***, ***y*** y ***z*** variables de tipo entero cuyos valores son respectivamente 1, 2, 3, 4, 5 y 6. Utilizando las primitivas de Pila calcular las siguientes expresiones en postfijo.

a. u v + w x * -
b. u v * w x * +
c. u v w * + x y * - z +

Almacenamiento en memoria

Los elementos de una pila los podemos guardar en memoria utilizando dos tipos de almacenamiento:

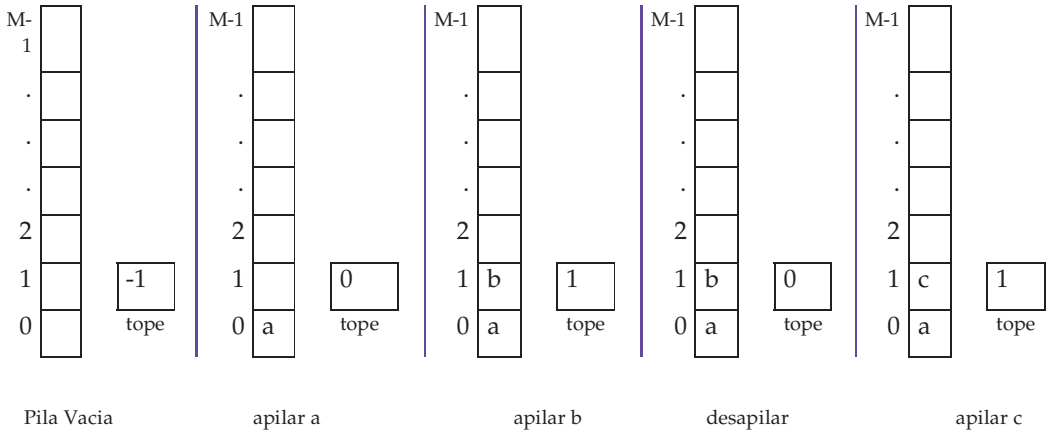
- memoria estática
- memoria dinámica

Memoria estática o contigua

Una pila puede ser almacenada utilizando un arreglo para guardar sus elementos y una variable de tipo entero que llamaremos **tope** cuyo valor es la posición del arreglo donde se encuentra el tope. Si la pila está vacía el valor de **tope** es –1. La operación ***apilar*** consiste en incrementar el valor de **tope** en 1 y en esa posición del arreglo guardar el nuevo elemento. La operación ***desapilar*** consiste simplemente

en restar 1 al valor de **tope** y ***recuperarTope*** retorna el elemento del arreglo que está en la posición indicada en **tope**.

En el siguiente diagrama mostramos el estado de la pila y el valor de tope a medida que se realizan operaciones de apilar y desapilar. M es el tamaño del arreglo.

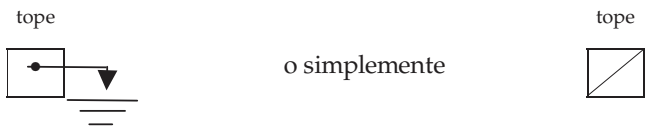


Observar que cuando se desapila, el elemento dasapilado sigue permaneciendo en la pila, pero no es posible acceder a él.

Todas las operaciones son O(1). El mayor inconveniente de este almacenamiento es que debemos definir a priori el tamaño del arreglo y esto puede producir un desperdicio de memoria o falta de ella. En este último caso en lenguajes como Java, C++ o C podemos expandir el arreglo en tiempo de ejecución. Esto puede ocasionar un desperdicio de memoria aún más importante si se realizan muchas inserciones seguidas, y en consecuencia expansiones del arreglo y luego varias eliminaciones. Supongamos al inicio pedimos memoria para 10 elementos y se realizan 11 inserciones seguidas. Cuando queremos apilar el onceavo elemento duplicamos el arreglo con lo cual tenemos 20 posiciones de memoria reservadas. Luego hacemos 9 eliminaciones con lo cual sólo 2 posiciones de las 20 tienen valores representativos.

Memoria dinámica

Los elementos de la pila se encuentran almacenados en celdas de memoria denominadas *nodos* que se almacenan en lugares no necesariamente contiguos de memoria. Cada nodo está formado por dos campos: uno contiene el dato propiamente dicho y el otro contiene la referencia o puntero del próximo nodo. Además una variable de tipo nodo que llamaremos **tope** contiene la referencia o apunta al nodo que contiene al elemento tope. Cuando la pila está vacía el valor de la variable **tope** es NULO y se lo suele graficar de la siguiente manera.



Para implementar la operación apilar se deben realizar los siguientes pasos.

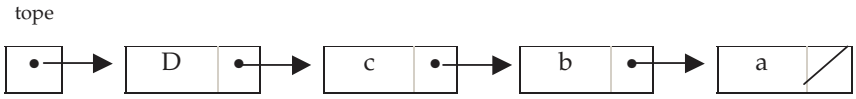
1. Pedir memoria para un nuevo nodo.

2. Asignar al campo de datos del nodo el nuevo elemento a apilar. Supongamos que éste sea "a".

3. Asignar al campo de referencia del nodo el valor de la variable **tope**.

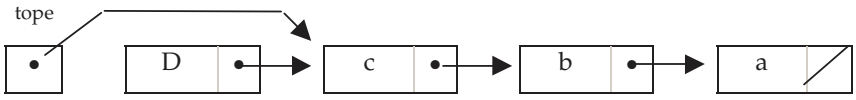
4. Asignar a la variable **tope** la dirección del nuevo nodo.

Luego de apilar b, c y d en ese orden , la pila queda:



La operación desapilar consiste:

1. Asignar a la variable **tope** la dirección del nodo cuya dirección está almacenada en campo de referencia del nodo apuntado por la variable **tope**.



2. Liberar la memoria ocupada por el nodo eliminado.

La operación **recuperarTope** consiste simplemente en retornar el campo dato del nodo que está apuntado por **tope** y la operación **esVacia** debe retornar verdadero si el valor de tope es NULO y falso en caso contrario.

Ejercicios

📖 **Ejercicio 8**
Algunas especificaciones del TaD Pila agregan la operación primitiva vaciar. Le parece conveniente? Por que?