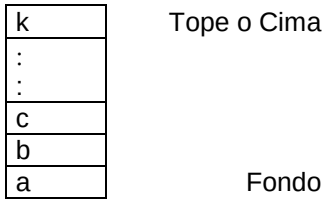
UNIVERSIDAD DE EL SALVADOR
FACULTAD DE INGENIERIA Y ARQUITECTURA
ESCUELA DE INGENIERIA DE SISTEMAS INFORMATICOS
PROGRAMACION II

ESTRUCTURAS DE DATOS PILAS

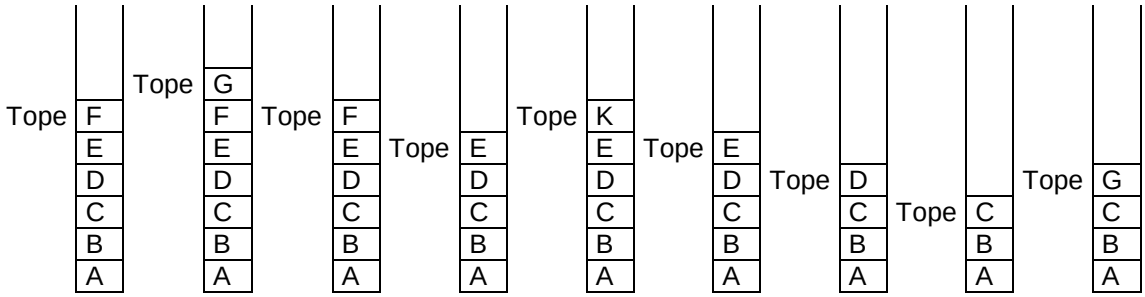
La pila es una lista ordenada de elementos caracterizada porque las operaciones de inserción y eliminación se realizan solamente en un extremo de la estructura. El extremo donde se realizan estas operaciones se denomina habitualmente **'cima'** o '**tope'** (top en ingles).

A diferencia de un arreglo, la definición de la pila considera la inserción y eliminación de elementos, por lo cual se dice que una pila es un objeto dinámico en constante cambio.

Dada una pila P=(a,b,c,...k), se dice que "a", es el elemento más inaccesible de la pila, está en el **fondo** de la pila (bottom) y que "k", el más accesible, está en la **cima**.



Las restricciones definidas para la pila implican que si una serie de elementos A, B, C, D, E se añaden, en este orden a una pila, entonces el primer elemento que se borre de la estructura deberá ser el E. Por tanto, resulta que el último elemento que se inserta en una pila es el primero que se borra. Por esta razón, se dice que una pila es una lista LIFO (Last In First Out, es decir, el último que entra es el primero que sale), la siguiente figura muestra el crecimiento de una pila en un determinado momento.



Un ejemplo típico de pila lo constituye un montón de platos, cuando se quiere introducir un nuevo plato, éste se pone en la posición más accesible, encima del último plato. Cuando se coge un nuevo plato, éste se extrae, igualmente, del punto más accesible, el último que se ha introducido.

Otro ejemplo natural de la aplicación de la estructura pila aparece durante la ejecución de un programa de computadora, en la forma en que la máquina procesa las llamadas a los procedimientos. Cada llamada a un procedimiento (o función) hace que el sistema almacene toda la información asociada con ese procedimiento (parámetros, variables, constantes, dirección de retorno, etc..) de forma independiente a otros procedimientos y permitiendo que unos procedimientos puedan invocar a otros distintos (o a si mismos) y que toda esa información almacenada pueda ser recuperada convenientemente cuando corresponda. Como en un procesador sólo se puede estar ejecutando un procedimiento, esto quiere decir que sólo es necesario que sean accesibles los datos de un procedimiento (el último activado que está en la cima). De ahí que la estructura pila sea muy apropiada para este fin.

OPERACIONES BÁSICAS EN UNA PILA

Los dos cambios que pueden hacerse a una pila son **agregar** y **remover** elementos de la pila. Dada una pila s y un elemento *i*, al ejecutar la operación **push(s, *i*)** agrega el elemento i al tope de la pila s. La operación **pop(s)** remueve el elemento superior y lo retorna como valor de función, por lo tanto, la operación

***i* = pop(s) ;**

remueve el elemento de tope de la pila s y le asigna su valor a la variable *i*.

Por ejemplo las operaciones de agregar y remover elementos de la pila de la grafica anterior serian las siguientes:

```
push(s, A) ;      // letra A
push(s, B) ;      // letra B
push(s, C) ;      // letra C
push(s, D) ;      // letra D
push(s, E) ;      // letra E
push(s, F) ;      // letra F

push(s, G) ;      // letra G
pop(s) ;          // letra G

pop(s) ;          // letra F
push(s, K) ;      // letra K
pop(s) ;          // letra K

pop(s) ;          // letra E
pop(s) ;          // letra D
push(s, G) ;      // letra G
```

No hay limite superior de la cantidad de elementos que puede contener una pila. Si una pila contiene un solo elemento y éste es removido, entonces la pila no contiene ningún elemento y se llama **pila vacía**, por lo tanto la operación de extracción de elementos de una pila (pop) no se puede aplicar a una pila vacía porque esta no tiene elementos para remover.

Antes de aplicar el operador **pop** a una pila debe de verificarse que la pila no este vacía. La operación **empty(s)** determinará si una pila **s** está o no vacía. Si la pila está vacía, **empty(s)**, retornará **TRUE**; sino retornará **FALSE**.

Otra operación es la de determinar cual es el elemento superior de la pila. Esta operación se escribe **stacktop(s)** y retorna el elemento al que apunta el tope de la pila s.

i = stacktop(s) ;

Es equivalente a :

***i* = pop(s) ;**
push(s, *i*) ;

igual que la operación **pop**, no se define la operación **stacktop** para una pila vacía. El resultado de un intento no válido por remover o acceder un elemento de una pila vacía se llama **subdesbordamiento**.

Esto se consigue verificando que la operación **empty(s)** sea **FALSE** antes de intentar la operación **pop(s)** o **stacktop(s)**.

REPRESENTACIÓN DE LAS ESTRUCTURAS DE DATOS PILAS

Los lenguajes de programación de alto nivel no suelen disponer de un tipo de datos pila. Aunque por el contrario, en lenguajes de bajo nivel (ensamblador) es posible manipular directamente alguna estructura pila propia del sistema. Por lo tanto, en general, es necesario representar la estructura pila a partir de otros tipos de datos existentes en el lenguaje.

1. IMPLEMENTACIÓN DE PILAS EMPLEANDO MEMORIA ESTÁTICA:

La forma más simple, y habitual, de representar una pila es mediante un vector unidimensional. Este tipo de datos permite definir una secuencia de elementos (de cualquier tipo) y posee un eficiente mecanismo de acceso a la información contenida en él.

Al definir un arreglo hay que determinar el número de índices válidos y, por lo tanto, el número de componentes definidos. Entonces, la estructura pila representada por un arreglo tendrá limitado el número de posibles elementos.

Una pila en "C" se declara como una estructura que contiene dos objetos: un arreglo para contener los elementos de la pila y un entero para indicar la posición del tope de la pila actual dentro del arreglo.

```
#define STACKSIZE 100
struct stack{
      int top ;
      int items[STACKSIZE] ;
      }
```

Se declara una pila *s* mediante la siguiente instrucción:

```
      struct stack s ;
```

los elementos de la pila s contenidos en el arreglo s.items son enteros y que la pila en ningún momento contendrá más que enteros STACKSIZE (tamaño de la pila), en este ejemplo STACKSIZE se establece en 100, para indicar que la pila solo puede contener 100 elementos (items[0] hasta items[99]).

Los elementos de la pila podrían haberse declarado como

```
float items[STACKSIZE] ; // ó

char items[STACKSIZE] ; // o cualquier otro tipo de dato
```

el identificador *top* (tope o cima de la pila) siempre debe declararse como entero, porque su valor representa la posición dentro del arreglo *items* del elemento en el tope de la pila. Por lo tanto si el valor de s.top=4, existen 5 elementos de la pila: s.items[0], s.items[1], s.items[2], s.items[3], s.items[4] . cuando se remueve de la pila, el valor de s.top cambia a 3 , para indicar que en este momento solo existen 4 elementos en la pila y que s.item[3] es el elemento superior.

Por el contrario si se agrega un elemento nuevo a la pila, el valor de s.top debe incrementarse en 1 para llegar a 6 y el objeto nuevo insertado en s.item[5]. La pila vacía no contiene elementos y, por lo tanto, se indica mediante top = −1, para inicializar una pila *s* para un estado vacío se ejecuta al principio la siguiente instrucción:

```
s.top = −1 ;
```

Para determinar en tiempo de ejecución el estado de una estructura de tipo pila (Es decir, si una pila esta vacía o no) se utiliza la siguiente instrucción:

```
if (s.top == –1) //entones
     // la pila esta vacía
else
     // la pila no esta vacía
```

Nota: Recuerde que el acceso al componente top de la pila s, es posible mediante la utilización del operador "**.**" Para variables "normales" de tal forma que estas se modifican de forma directa. El operador "**->**" para variables declaradas tipo puntero de tal forma que el componente se afecta de forma indirecta.(Revisar estructuras y uniones).

Escribiendo la función empty(s) del modo siguiente:

```
#define STACKSIZE 100
#define TRUE 1
#define FALSE 0
int empty(struct stack *ps) {
          if (ps->top == –1)
               return (TRUE) ;
          else
               return (FALSE) ;
          } // fin de empty
```

una vez definida la función empty, se implementa una prueba de la pila vacía mediante el siguiente código:

```
if (empty(&s))          // el parámetro se pasa por referencia
     // la pila esta vacía
else
     // la pila no esta vacía
```

Implementación de la operación pop.

La posibilidad de un **subdesbordamiento** debe considerarse al implementarse la operación **pop**, porque inadvertidamente se puede querer remover un elemento de una pila vacía, a continuación se describe la función pop que ejecuta las siguientes acciones:

1.- Si la pila esta vacía, imprime un mensaje de advertencia y detiene la ejecución.
2.- Remueve el elemento superior de la pila (al que apunta Tope).
3.- Retorna ese elemento al programa que lo invocó.

Si la pila consta de tipos de datos simples, se implementa como se muestra el siguiente código, pero si consta de tipos de datos complejos como una estructura, entonces la operación pop se implementaría para que retorne un apuntador a un elemento de datos de un tipo adecuado.

```
int pop(struct stack  *ps) {
          if (empty(ps))  {
               printf("%s", "Error..... Subdesbordamiento de la Pila");
               exit(1) ;
               } // fin del if
          return(ps->items[ps->top––]) ;
          }  // fin de la función pop
```

Si la pila no está vacía, el elemento superior de la pila se conserva como el valor retornado. Después, este elemento es quitado de la pila mediante la expresión **ps->top**––.

Como utilizar la función pop:

```
int x;
x = pop (&s) ;
```

Donde ***x*** contiene el valor removido de la pila.

Antes de remover un elemento de la pila, deberá de asegurarse que la pila no este vacía cuando se invoque la función pop. Para verificar si la pila no está vacía se podrá utilizar el siguiente código:

```
If (! empty(&s))
     x = pop(&s) ;
else
   // código para corregir el Error
```

Implementación de la operación push.

A continuación se muestra una primera versión del procedimiento de insertar elementos a una pila (push).

```
void push(struct stack  *ps, int x) {
        // versión: 1
        ps->items[++(ps->top)] = x ;
         return ;
         }  // fin de la función push
```

Este módulo prepara un espacio para agregar el elemento definido con el argumento ***x***, en la pila incrementando el tope en 1 (++ps->top) y después agrega *x* en el arreglo *s.items*.

Es posible que una pila se haga más grande que el arreglo que se reservó para contenerla. Esto ocurre cuando el arreglo está lleno, cuando la pila tiene tantos elementos como el arreglo y se quiere agregar otro elemento a la pila, el resultado de esto se le llama **desbordamiento**. Esto se da cuando se quiere sobrepasar el limite de elementos establecidos para la pila, por medio de la constante STACKSIZE que en este caso tiene el valor de 100 (para un índice desde 0 hasta 99).

Si s.top=99 y se quisiera ingresar un elemento mas a la pila s.top aumenta a 100 y se hace un intento de agregar *x* en s.items[100], por lo tanto ese intento de asignación produciría un error. Por esta razón, deberá modificarse el procedimiento *push* de la siguiente forma:

```
void push(struct stack  *ps, int x) {
        // versión 2
        if (ps->top == STACKSIZE – 1)  {
             printf("%s", "Desbordamiento de la Pila");
             exit(1) ;
             }
        else
             ps->items[++(ps->top)] = x ;
         return ;
        }  // fin de la función push
```

En esta nueva versión del procedimiento primero se verifica si el arreglo está lleno antes de intentar agregar otro elemento a la pila. El arreglo está lleno si ps->top == STACKSIZE – 1, si la condición es cierta, la ejecución se detiene de inmediato después de desplegar el mensaje de error.

Implementación de la operación stacktop.

Esta operación retorna el elemento superior de una pila sin removerlo de ella

```
int stacktop(struct stack  *ps) {
        if (empty(ps))  {
              printf("%s", "Subdesbordamiento de la Pila");
              exit(1) ;
              }
        else
               return(ps->items[ps->top]) ;
        }  // fin de stacktop
```

APLICACIONES DE LAS ESTRUCTURAS DE DATOS PILAS

Considere, la suma de A y B. Aplicando el operador "+" a los operandos A y B. Existen tres formas de representar esta operación:

A + B	Notación interfija
+ A B	Notación prefija
A B +	Notación postfija

Los prefijos que anteceden a la palabra "fija" hacen referencia a la posición relativa del operador en relación con los dos operandos. En la notación prefija el operador precede a los dos operandos, en la notación postfija el operador está después de los dos operandos y en la notación interfija el operador está entre los dos operandos.

Ejemplos:

Evaluación de la expresión A + B * C, se sabe que la multiplicación antecede a la adición, por lo tanto la expresión anterior se puede representar así A + (B * C), se dice que la multiplicación tiene preferencia sobre la adición, se desea escribir la expresión en notación postfija.

A + (B * C)	Los paréntesis solo son para enfatizar
A + (BC *)	Convierte la multiplicación
A (BC *) +	Convierte la adición
ABC *+	Notación postfija

Convertir la expresión (A + B) * C en notación postfija

(A + B) * C	Forma interfija
(AB +) * C	Convierte la adición
(AB +) C *	Convierte la multiplicación
AB + C *	Notación postfija

Otros ejemplos:

Interfija	Postfija	Prefija
A + B	AB +	+ AB
A + B – C	AB + C –	– + ABC
(A + B) * (C – D)	AB + CD –*	*+ AB – CD
(A + B) * C – (D – E)	AB + C * DE––	–*+ ABC –DE

Observe que las únicas reglas que deben recordarse durante el proceso de conversión son: que las operaciones con procedencia mayor se convierten primero (Para una mejor comprensión de las conversiones estudie la jerarquía de los operadores.

Algoritmo para la evaluación de una expresión postfija.

PilaExpresion = PilaVacia.

/* Se explora la cadena correspondiente a la expresión leyendo un elemento a la vez*/

/* hasta alcanzar el fin de la cadena */

Mientras (No se alcance el fin de la cadena){

Símbolo = Siguiente carácter de la cadena

Si (símbolo es operando)

Push(PilaExpresion, símbolo)

Sino {

Operando2 = Pop(PilaExpresion)

Operanso1 = Pop(PilaExpresion)

Valor = Resultado de aplicar símbolo a operando2 y operando1

Push(PilaExpresion,Valor)

}

}

Retornar (Pop(PilaExpresion))

Ejemplo de la aplicación del algoritmo para una expresión dada(ejecución del algoritmo).

ABC+ - CDB / + * B ^ C + . Donde A=6; B=2; C=3; D=8.

Paso	Símbolo	Operando1	Operando2	Valor	PilaExpresion
1	6				6
2	2				6,2
3	3				6,2,3
4	+	2	3	5	6,5
5	-	6	5	1	1
6	3	6	5	1	1,3
7	8	6	5	1	1,3,8
8	2	6	5	1	1,3,8,2
9	/	8	2	4	1,3,4
10	+	3	4	7	1,7
11	*	1	7	7	7
12	2	1	7	7	7,2
13	^	7	2	49	49
14	3	7	2	49	49,3
15	+	49	3	52	52

Observe: que el ultimo valor en la pila de expresión 52 (ultima columna de la tabla anterior) Se considera el valor de retorno que surge como resultado de evaluar la expresión con los valores dados. Puede verse además, como la pila crece y decrece dinámicamente en la medida que se van evaluando los operandos.

2. IMPLEMENTACIÓN PILAS EMPLEANDO MEMORIA DINAMICA:

Una de las aplicaciones más interesantes y potentes de la memoria dinámica y los punteros son las estructuras dinámicas de datos. Las estructuras básicas disponibles en C y C++ tienen una importante limitación: no pueden cambiar de tamaño durante la ejecución. Los arreglos están compuestos por un determinado número de elementos, número que se decide en la fase de diseño, antes de que el programa ejecutable sea creado.

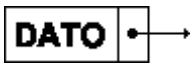
En muchas ocasiones se necesitan estructuras que puedan cambiar de tamaño durante la ejecución del programa. Las estructuras dinámicas nos permiten crear estructuras de datos que se adapten a las necesidades reales a las que suelen enfrentarse los programas. Pero no sólo eso, también permiten crear estructuras de datos muy flexibles, ya sea en cuanto al orden, la estructura interna o las relaciones entre los elementos que las componen.

Las estructuras de datos están compuestas de otras pequeñas estructuras a las que llamaremos nodos o elementos, que agrupan los datos con los que trabajará nuestro programa y además uno o más punteros autoreferenciales, es decir, punteros a objetos del mismo tipo nodo.

Una estructura básica de un nodo para crear listas de datos seria:

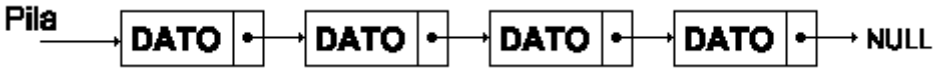
```
typedef struct _nodo {
   int dato;
   struct _nodo *siguiente;
} tipoNodo;

typedef tipoNodo *pNodo;
typedef tipoNodo *Pila
```

Durante el presente curso usaremos gráficos para mostrar la estructura de las estructuras de datos dinámicas. El nodo anterior se representará así: (considerando un campo para el almacenamiento de datos y otro para las direcciones de memoria)



Las estructuras dinámicas son una implementación de TDAs o TADs (Tipos Abstractos de Datos). En estos tipos el interés se centra más en la estructura de los datos que en el tipo concreto de información que almacenan.

tipoNodo es el tipo para declarar nodos, evidentemente; **pNodo** es el tipo para declarar punteros a un nodo; **Pila** es el tipo para declarar pilas.

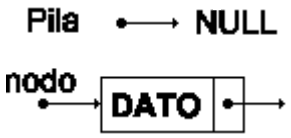


Es evidente, a la vista del gráfico, que una pila es una lista abierta. Así que sigue siendo muy importante que el programa nunca pierda el valor del puntero al primer elemento. Para no perder el acceso a los datos. Teniendo en cuenta que las inserciones y borrados en una pila se hacen siempre en un extremo.

Las operaciones con pilas son muy simples, no hay casos especiales, salvo que la pila esté vacía.

Push en una pila vacía:

Partiremos de que ya tenemos el nodo a insertar y, por supuesto un puntero que apunte a él, además el puntero a la pila valdrá NULL:



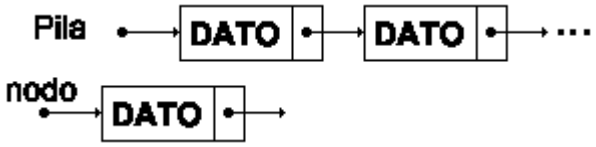
El proceso es muy simple, bastará con que:

1. nodo->siguiente apunte a NULL.
2. Pila apunte a nodo.

Push en una pila no vacía:

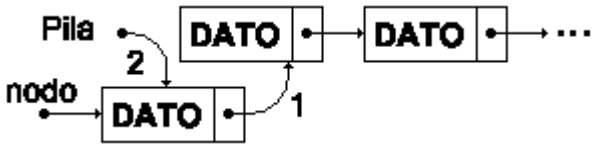
Podemos considerar el caso anterior como un caso particular de éste, la única diferencia es que podemos y debemos trabajar con una pila vacía como con una pila normal.

De nuevo partiremos de un nodo a insertar, con un puntero que apunte a él, y de una pila, en este caso no vacía:



El proceso sigue siendo muy sencillo:

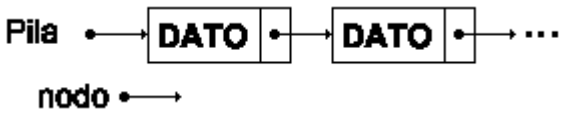
1. Hacemos que nodo->siguiente apunte a Pila.
2. Hacemos que Pila apunte a nodo.



Pop, leer y eliminar un elemento

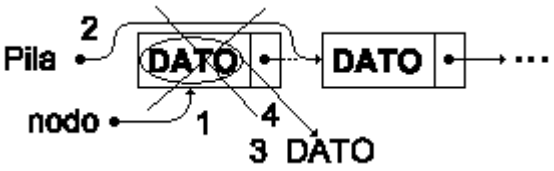
Ahora sólo existe un caso posible, ya que sólo podemos leer desde un extremo de la pila.

Partiremos de una pila con uno o más nodos, y usaremos un puntero auxiliar, nodo:



1. Hacemos que nodo apunte al primer elemento de la pila, es decir a Pila.
2. Asignamos a Pila la dirección del segundo nodo de la pila: Pila->siguiente.
3. Guardamos el contenido del nodo para devolverlo como retorno, recuerda que la operación pop equivale a leer y borrar.

4. Liberamos la memoria asignada al primer nodo, el que queremos eliminar.



Si la pila sólo tiene un nodo, el proceso sigue siendo válido, ya que el valor de Pila->siguiente es NULL, y después de eliminar el último nodo la pila quedará vacía, y el valor de Pila será NULL.

Algoritmo de la función "push"

1. Creamos un nodo para el valor que colocaremos en la pila.
2. Hacemos que nodo->siguiente apunte a Pila.
3. Hacemos que Pila apunte a nodo.

```
void Push(Pila *pila, int v) {
   pNodo nuevo;

   /* Crear un nodo nuevo */
   nuevo = (pNodo)malloc(sizeof(tipoNodo));
   nuevo->valor = v;

   /* Añadimos la pila a continuación del nuevo nodo */
   nuevo->siguiente = *pila;
   /* Ahora, el comienzo de nuestra pila es en nuevo nodo */
   *pila = nuevo;
}
```

Algoritmo de la función "pop"

1. Hacemos que nodo apunte al primer elemento de la pila, es decir a Pila.
2. Asignamos a Pila la dirección del segundo nodo de la pila: Pila->siguiente.
3. Guardamos el contenido del nodo para devolverlo como retorno, recuerda que la operación pop equivale a leer y borrar.
4. Liberamos la memoria asignada al primer nodo, el que queremos eliminar.

```
int Pop(Pila *pila) {
   pNodo nodo; /* variable auxiliar para manipular nodo */
   int v;      /* variable auxiliar para retorno */

   /* Nodo apunta al primer elemento de la pila */
   nodo = *pila;
   if(!nodo) return 0; /* Si no hay nodos en la pila retornamos 0 */
   /* Asignamos a pila toda la pila menos el primer elemento */
   *pila = nodo->siguiente;
   /* Guardamos el valor de retorno */
   v = nodo->valor;
   /* Borrar el nodo */
```

```
    free(nodo);
    return v;
}
```

Para una revisión completa de la implementación de estructuras de tipo pilas consulte las guia de laboratorio No. 4.