8

PILAS

8.1 DEFINICIÓN

Una pila es una lista ordenada en la cual todas las operaciones (inserción y borrado) se efectúan en un solo extremo llamado TOPE. Es una estructura LIFO (**L**ast **I**nput **F**irst **O**utput) que son las iniciales de las palabras en inglés último en entrar primero en salir, debido a que los datos almacenados en ella se retiran en orden inverso al que fueron entrados.

Un ejemplo de pilas en computadores se presenta en el proceso de llamadas a subprogramas y sus retornos.

Supongamos que tenemos un programa principal y 3 subprogramas así:

Programa principal	Subprog. P1	Subprog. P2	Subprog. P3
---	---	---	---
---	---	---	---
---	---	---	---
---	P2	P3	---
P1	S ---	T ---	---
R ---	---	---	---
---	---	---	---
---	---	---	---
fin(prog. Ppal)	fin(P1)	fin(P2)	fin(P3)

Cuando se ejecuta el programa principal, se hace una llamada al subprograma P1, es decir, ocurre una interrupción a la ejecución del programa principal.

Antes de iniciar la ejecución de este subprograma, se guarda la dirección de la instrucción donde debe retornar a continuar la ejecución del programa principal cuando termine de ejecutar el subprograma. Llamemos R esta dirección.

Cuando ejecuta el subprograma P1 existe una llamada al subprograma P2, hay una nueva interrupción, pero antes de ejecutar el subprograma P2 se guarda la dirección de la instrucción donde debe retornar a continuar la ejecución del subprograma P1, cuando termine de ejecutar el subprograma P2. Llamemos S esta dirección.

Hasta el momento hay guardados dos direcciones de retorno: R, S

Cuando ejecuta el subprograma P2 hay llamada a un subprograma P3, lo cual implica una nueva interrupción y por ende guardar una dirección de retorno al subprograma P2, la cual llamamos T.

Tenemos entonces tres direcciones guardadas así: R, S, T

Al terminar la ejecución del subprograma P3, retorna a continuar ejecutando en la última dirección que guardó, es decir, extrae la dirección T y regresa a continuar ejecutando el subprograma P2 en dicha instrucción. Los datos guardados ya son:

R, S

Al terminar el subprograma P2, extrae la última dirección que tiene guardada y en este caso S, y retorna a esta dirección a continuar la ejecución del subprograma P1.

En este momento los datos guardados son: R

Al terminar la ejecución del subprograma P1 retornará a la dirección que tiene guardada, o sea a R.

Obsérvese que los datos fueron procesados en orden inverso al que fueron almacenados, es decir, último en entrar primero en salir. Es esta forma de procesamiento la que define una estructura PILA.

Definamos ahora formalmente la estructura pila:

```
ESTRUCTURA PILA
  Funciones:
    Crear()→ pila
    Apilar(elemento, pila) → pila
    Desapilar(pila) → pila
```

Tope(pila) → elemento
Esvacia(pila) → lógico

Axiomas:
Para todo P ∈ pilas, i ∈ elementos

Esvacia(Crear) ::= Verdad
Esvacia(Apilar(i, P)) ::= Falso
Desapilar(Crear) ::= Crear
Desapilar(Apilar(i, P)) ::= P
Tope(Crear) ::= error
Tope(Apilar(i, P)) ::= i

Crear, es la función constante que crea la pila vacía.

Apilar, es la función que permite incluir un elemento en una pila si tenemos una pila con 3 elementos R, S, T la estructura sería:

Apilar(T, Apilar(S, Apilar(R, Crear)))

De acuerdo a la forma general Apilar(i, P), el elemento i es T y

P = Apilar(S,Apilar(R,Crear))

Desapilar extrae el último elemento de la pila y deja una nueva pila la cual queda con un elemento menos.

Tope da información sobre el último elemento que se halla en la pila sin extraerlo.

Esvacia chequea que haya elementos en la pila.

8.2 REPRESENTACIÓN DE PILAS

Hemos definido la estructura pila en abstracto, veamos ahora cómo representar una pila en un computador.

Una pila dentro de un computador la podremos representar de dos formas: en un vector o como lista ligada.

Representación de pilas en un vector

La forma más simple es utilizar un arreglo de una dimensión y una variable, que llamaremos **Tope**, que indique la posición del arreglo en la cual se halla el último elemento de la pila.

La función **Crear** sería simplemente definir un vector y una variable, que llamaremos **Tope**, la cual inicialmente tendrá el valor cero.

```
dimension pila(100)
tope = 0
```

La función **Esvacia** consiste simplemente en preguntar por el valor de **Tope**:

```
if tope = 0 then
    verdad
else
    falso
end(if)
```

La función **Tope**, que da información sobre el último elemento de la pila es:

```
if tope = 0 then
   error
else
    return(pila(tope))
end(if)
```

Como se puede observar, estas funciones son tan sencillas y tan cortas que no se justifica elaborar algoritmos independientes para ellas. En su defecto se codificarán estas instrucciones, en el proceso que se esté efectuando, cuando sea necesario.

Los algoritmos para apilar y desapilar se presentan en los siguientes subprogramas:

```
sub_programa apilar(pila, n, tope, dato)
   if  tope = n then
       pila_llena
   else
      tope = tope + 1
      pila(tope) = dato
    end(if)
fin(apilar)
```

n y dato son parámetros por valor y **tope** y **pila** parámetros por referencia.

pila es un vector con capacidad de **n** elementos.

tope es la variable que indica la posición del vector en la cual se halla el último elemento de la pila.

dato es la variable que lleva la información a guardar en la pila.

Ejemplo:

	1	2	3	4	5	6	7	8	9
Pila	A	B	C	D					

En nuestro ejemplo el vector se llama **Pila**, **n** es 9 y **tope** es 4.
El subprograma PILA_LLENA se invocará cuando **tope** sea 9.

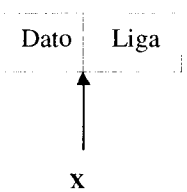
Dicho subprograma sacará un mensaje apropiado y detendrá el proceso.

```
sub_programa desapilar(pila,tope,dato)
   if  tope = 0  then
       pila_vacia
   else
       dato = pila(tope)
       tope = tope - 1
   end(if)
end(sub_programa)
```

pila, tope y **dato** son parámetros por variable.
dato es la variable que retorna la información sacada de la pila.

Representación de pilas como lista ligada

Siempre que se desee representar un objeto como lista ligada lo primero que se debe hacer es definir la configuración del registro.



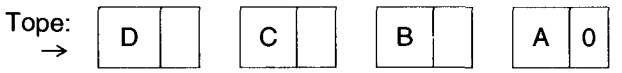
Dos campos: Dato y Liga

dato(x) = Se refiere al campo dato en el registro **x**.
liga(x) = Se refiere al campo liga del registro **x**, es el campo que apunta hacia otro registro.

Representemos como lista ligada la siguiente pila:

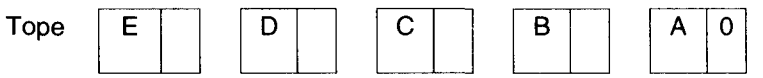
Pila

Dibujémosla de la forma como es común hacerlo con listas ligadas



Las operaciones son *Apilar y Desapilar.*

Apilar: Consiste en insertar un registro al principio de una lista ligada.




```
sub_programa apilar(tope,d)
      new(x)
      dato(x) = d
      liga(x) = tope
      tope = x
fin(apilar)
```

Se observa que es más fácil y más eficiente que manejando la pila en vectores, ya que en listas ligadas no hay necesidad de controlar pila llena antes de apilar.

Desapilar: consiste en borrar el primer registro de una lista ligada. Para desapilar es necesario saber si la pila está o no vacía. Si está vacía no se podrá desapilar.

La pila está vacía cuando la lista sea vacía, es decir, cuando **tope = 0**

```
sub_programa desapilar(tope, d)
    if  tope = 0 then
        pila_vacía
    else
        d = dato(tope)
        x = tope
        tope = liga(tope)
        devolver_registro(x)
fin(desapilar)
```

8.3 APLICACIÓN DE PILAS: MANEJO DE EXPRESIONES

Cuando los pioneros de la ciencia de los computadores concibieron la idea de lenguajes de programación de alto nivel, se enfrentaron a muchas dificultades técnicas. Una de las mayores fue cómo generar instrucciones en lenguaje de máquina que evaluaran correctamente una expresión aritmética.

Una asignación completa tal como: **X = A / B^C + D * E - A * C**

Puede tener varios significados; más aún si estuviera definida únicamente, es decir, utilizando paréntesis para definir el orden de ejecución de las operaciones, aún se ve dificultoso generar una secuencia correcta y razonable de instrucciones para evaluar dicha expresión. Afortunadamente la solución de que se dispone en la actualidad es simple y elegante. Más aún, es tan simple que este aspecto, en la elaboración de compiladores, es de los que requiere menor esfuerzo.

Una expresión está compuesta por operandos, operadores y delimitadores. La expresión anterior tiene 5 operandos : A, B, C, D y E.

Aunque en el ejemplo, las variables son de una letra, los operandos pueden ser cualquier nombre de variable o constantes en algún lenguaje de programación.

En cualquier lenguaje de programación los valores de las variables deben ser consistentes con las operaciones que se ejecutan sobre ellas. Estas operaciones son descritas por los operadores. En la mayoría de los lenguajes de programación hay diferentes operadores, los cuales se aplican a los diferentes tipos de datos que las variables pueden almacenar.

Primero están los operadores aritméticos básicos: Suma, resta, multiplicación, división y potenciación (+, -, *, /, ^). Otros operadores aritméticos son el más y el menos unitario.

Luego están los operadores relacionales: <, <=, =, >, >=, <>, los cuales generalmente se aplican sobre operadores aritméticos, aunque también se aplican a hileras de caracteres. ('GATO' es menor que 'PERRO' debido a su precedencia en orden alfabético). El resultado de una expresión que contiene operadores relacionales es o verdadero o falso.

Existen también los operadores lógicos, los cuales se aplican sobre expresiones o variables lógicas.

El primer problema para evaluar una expresión es decidir el orden en que se ejecutan las operaciones. Esto implica definir ese orden. Por ejemplo, si A = 4, B = 2, C = 2, D = 3 y E = 3 el valor asignado a X en la expresión dada podría ser:

```
X = 4/(2^2) + (3*3) - (4*2)
X = (4/4) + 9 - 8
X = 2
```

Sin embargo, la verdadera intención del programador podría haber sido:

X = (4/2)^(2 + 3) * (3-4) * 2
X = (4/2)^(5) * (-1) *2
X = (2^5) * (-2)
X = 32 * (-2)
X = - 64

Obviamente, él pudo haber especificado este último orden de evaluación, utilizando paréntesis:

X = ((((A / B)^(C + D))*(E - A))*C)

Para definir el orden de evaluación se ha asignado a cada operador una prioridad. Entonces, los operadores con mayor prioridad se ejecutan primero, los operadores con igual prioridad se ejecutan en el orden en que aparezcan de izquierda a derecha. Una muestra de prioridades de operadores es:

Operador	Prioridad
^	6
*, /	5
+, -	4
<, <=, =, >, >=, <>	3
NOT	2
AND	1
OR	0

Cuando se tiene una expresión con dos operaciones de la misma prioridad adyacentes, es necesario reglamentar cual se ejecuta primero. En álgebra se considera A^B^C como A^(B^C) y esta es la norma que rige para la potenciación cuando no hay paréntesis; la potenciación se ejecuta de derecha a izquierda. Sin embargo, expresiones como A*B/C ó A+B-C se evalúan de izquierda a derecha.

Recuerde que este orden se puede alterar utilizando paréntesis, en cuyo caso la evaluación se efectúa desde los paréntesis más internos hacia los más externos.

Ahora que hemos definido prioridades sabemos cómo se evalúa correctamente la expresión dada.

Ahora, ¿cómo hace un compilador para evaluar una expresión dada?
La respuesta se obtiene reescribiendo la expresión en una forma llamada notación **posfijo**.

La forma tradicional como escribimos las expresiones se denomina **infijo**, porque los operadores van precedidos y seguidos por operandos.

Realmente existen 3 formas de escribir expresiones. Si queremos indicar que hay que sumar **a** con **b** lo podremos hacer así:

- Notación **INFIJO**: Operando operador operando **a+b**.
- Notación **POSFIJO**: Operando operando operador **ab+**.
- Notación **PREFIJO**: Operador operando operando **+ab**.

Cuando se tiene una expresión POSFIJO, el operador actúa sobre los dos operandos que le preceden, sin importar la prioridad de ejecución de los operadores y sin tener que utilizar paréntesis.

Cuando se tiene una expresión PREFIJO, el operador actúa sobre los dos operandos que le suceden, sin importar la prioridad de ejecución de los operadores y sin tener que utilizar paréntesis.

Consideremos la expresión que hemos estado trabajando: Su forma **POSFIJO** es: **ABC^/DE*+AC*-**

Hagamos un seguimiento a la forma de evaluación.

Cada vez que se efectúe una operación almacenamos su resultado en una posición temporal.

Procesando de izquierda a derecha, el primer operador que se encuentra es la potenciación, la cual se aplica sobre los dos operandos que le proceden que son B y C. A continuación damos una tabla del orden de ejecución de las operaciones y la forma como va quedando la expresión posfijo.

OPERACION		POSFIJO
R1 = B^C	→	A R1 / DE * + AC * -
R2 = A/R1	→	R2 DE * + AC * -
R3 = D * E	→	R2 R3 + AC * -
R4 = R2 + R3	→	R4 AC * -
R5 = A*C	→	R4 R5 -
R6 = R4 - R5	→	R6

Y R6 contiene el resultado.

Como se puede ver los paréntesis ya no son necesarios y la prioridad de los operadores ya no es relevante. La expresión se evalúa haciendo una búsqueda de izquierda a derecha, apilando operandos y aplicando los operadores sobre los dos últimos operandos que se hallen en la pila y colocando el resultado en la pila.

Este proceso de evaluación es mucho más simple que intentar una evaluación directa sobre la notación infijo.

Escribamos ahora un algoritmo para evaluar una expresión en notación POSFIJO.

```
sub_programa evapos(e)
    dimension pila(100)
    tope = 0
    x = siguiente_token(e)
    while  x <> ' '  do
        if x in operadores then
            opn2 = pila(tope)
            opn1 = pila(tope-1)
            casos de  x
                :*:  res = opn1 * opn2
                :/:  res = opn1 / opn2
                :+: res = opn1 + opn2
                :-:  res = opn1 - opn2
            fin(casos)
            tope = tope - 1
            pila(tope) = res
        else
            tope = tope + 1
            pila(tope) = x
        end(if)
        x = siguiente_token(e)
    end(while)
    res = pila(tope)
fin(evapos)
```

El anterior algoritmo utiliza una función llamada SIGUIENTE_TOKEN.

Cuando se tiene una expresión los operandos y operadores no necesariamente son de un carácter. En general, cualquier elemento de una expresión, operando u operador, recibe el nombre de TOKEN, la función SIGUIENTE_TOKEN, se encarga de extraer de la expresión, el elemento a procesar, sin importar el número de caracteres que tenga.

Veamos ahora cómo convertir una expresión en notación INFIJO a notación POSFIJO. Los pasos a seguir son:

- Parentizar completamente la expresión infijo.
- Mover los operadores a reemplazar su correspondiente paréntesis derecho.
- Borrar todos los paréntesis izquierdos.

Como ejemplo consideremos la expresión: A / B^C + D * E - A * C

Parentizarla completamente de acuerdo al orden de ejecución de los operadores. A cada operador le corresponderá un paréntesis izquierdo y un paréntesis derecho.

(((A / (B^C)) + (D * E))- (A * C))

Los operadores se moverán a reemplazar su correspondiente paréntesis derecho.

Al hacer los movimientos la expresión queda: (((A (BC^ / (DE * + (AC * -

Reescribir la expresión suprimiendo los paréntesis izquierdos. La expresión quedará:

ABC^ /DE * + AC * -

El problema con este método es que requiere varios pasos: parentizar la expresión, mover operadores y suprimir paréntesis izquierdos.

Si observamos bien ambas expresiones (en infijo y en posfijo) vemos que los operandos conservan el mismo orden. Por tanto si recorremos la expresión infijo de izquierda a derecha, podemos comenzar a armar la expresión posfijo, escribiendo los operandos a medida que se encuentren y el problema se reduce a un manejo de operadores, es decir, en qué momento se deben incluír en la expresión posfijo. La solución es almacenarlos en una pila y determinar el momento preciso en el cual se deben pasar a la expresión posfijo.

Por ejemplo, si tenemos la expresión A + B * C y deseamos llegar a su forma posfijo ABC*+ la secuencia de apilar y desapilar sería:

SIGUIENTE-TOKEN	PILA	EXPRESION POSFIJO
A	Vacía	A
+	+	A
B	+	AB

En este punto el SIGUIENTE_TOKEN es el operador * y debemos determinar si se coloca en la pila o si desapilamos el operador +. Debido a que el * es de mayor prioridad que el + lo apilamos:

*	+ *	AB
C	+ *	ABC

Hemos terminado con la expresión infijo y lo que hay que hacer es sacar los operadores que tenemos en la pila y llevarlos a la expresión obteniendo ABC*+.

Consideremos otro ejemplo: A*(B+C)*D, su forma posfijo es ABC+*D*, veamos la secuencia de operaciones para obtener esta expresión:

SIGUIENTE-TOKEN	PILA	EXPRESION POSFIJO
A	Vacía	A
*	*	A
(	* (	A
B	* (	AB
+	* (+	AB
C	* (+	ABC

En este momento el siguiente token es), la acción a tomar es desapilar todos los operadores hasta encontrar el paréntesis izquierdo correspondiente y luego sacarlo de la pila sin incluirlo en la expresión posfijo:

)	*	ABC +

El siguiente token es un asterisco, y el operador que está en la pila es también un asterisco; como son operadores de igual prioridad entonces sacamos el operador de la pila y lo llevamos a la expresión posfijo y apilamos el operador que se acabó de leer:

*	*	ABC + *
D	*	ABC + * D

Terminamos la expresión infijo, entonces sacamos los operadores que hay en la pila y obtenemos ABC + * D * que es la expresión posfija buscada.

En general, cuando se lea un operador siempre debe ser llevado a la pila; sin embargo, antes de hacer esto, debemos sacar de la pila todos los operadores cuya prioridad sea mayor o igual que la prioridad del operador que va a entrar.

Cuando se encuentre un paréntesis izquierdo siempre debe ser llevado a la pila sin sacar operadores de ella.

Cuando se encuentre un paréntesis derecho, se deben sacar todos los operadores que haya en la pila hacia la expresión posfijo, hasta hallar su correspondiente paréntesis izquierdo, el cual se borra de la pila sin llevarlo a la expresión posfijo.

Consideremos el caso de la potenciación, la cual se ejecuta de derecha a izquierda. Si tenemos la expresión A^B^C su forma posfijo es ABC^^

Al aplicar el proceso establecido tenemos:

SIGUIENTE-TOKEN	PILA	EXPRESION POSFIJO
A	Vacía	A
^	^	A
B	^	A B

El siguiente token es otro operador de potenciación y como la prioridad del operador que está en la pila es igual a la del operador que va a entrar, la acción es extraer el operador de la pila e incluir el que viene, por tanto obtenemos:

^	^	A B ^
C	^	A B ^ C

Se terminó la expresión infijo, luego la expresión posfijo resultante será:

AB^C^

la cual es diferente a la expresión posfija correcta que es ABC^^

Lo anterior nos lleva a adoptar una condición diferente para los operadores de operaciones que se ejecutan de derecha a izquierda. La solución a este problema es asignar prioridad distinta de ejecución a estos operadores para cuando están dentro de la pila y para cuando están fuera de la pila. Como el operador dentro de la pila no puede salir, le asignamos a estos operadores una prioridad mayor fuera de la pila que dentro de la pila.

Todas estas condiciones anteriores nos llevan a construir una tabla de prioridad de los operadores dentro y fuera de la pila, así:

OPERADOR	PRIORIDAD DENTRO DE LA PILA	PRIORIDAD FUERA DE LA PILA
^	3	4
*	2	2
/	2	2
+	1	1
-	1	1
(	0	4

El subprograma para convertir una expresión INFIJO a POSFIJO es:

```
sub_programa intopos(e)
      dimension pila(100)
      tope = 0
      x = siguiente_token(e)
      while x <> ' ' do
          casos de x
             :x in operadores:
                  while tope > 0 and pdp(pila(tope)) >= pfp(x) do
                        write (pila(tope))
                        tope = tope - 1
                  end(while)
                  tope = tope + 1
                  pila(tope) = x
             :x = ')':
                  while pila(tope) <> '(' do
                        write(pila(tope))
                        tope = tope - 1
                  end(while)
                  tope = tope - 1
             else
                  write(x)
          fin(casos)
          x = siguiente-token(e)
      end(while)
      while tope > 0 do
          write(pila(tope))
          tope = tope - 1
      end(while)
fin(intopos)
```

EJERCICIOS PROPUESTOS

1. Elabore un algoritmo para traducir una expresión de infijo a prefijo.
2. Elabore un algoritmo para traducir una expresión de posfijo a prefijo.
3. Elabore un algortimo para traducir una expresión de posfijo a infijo.