La forma de almacenamiento de registros por lo general es simple y lineal. Algunas veces se inserta un espacio de relleno dentro del registro con el propósito de aumentar la eficiencia de ciertas operaciones aritméticas. Este espacio puede ser insertado explícitamente o mediante el uso de una instrucción del lenguaje de programación, como la cláusula de COBOL SYNCHRONIZED.

Los registros son importantes porque pueden preservar la estructura lógica natural de los elementos de información relacionados.

TERMINOLOGIA

Archivo
Bytes inactivos
Campo
Clave
Elemento básico
Elemento grupal
Estructura
Número de nivel
Registro
Relleno

REFERENCIAS SUGERIDAS

COBOL, PL/I, and Pascal textbooks and language manuals.

MARCH, S. T. "Techniques for structuring database records," *ACM Computing Surveys* 15(1): 45–79, March 1983.

EJERCICIOS DE REPASO

1. ¿Cuál es la diferencia entre un arreglo y un registro? ¿Puede un arreglo ser parte de un registro? ¿Puede un registro ser parte de un arreglo?
2. ¿Por qué podría un programador desglosar un campo en componentes parciales?
3. Dos programas accesan el mismo archivo. Un programa trata a *Num-telefónico* como un elemento básico, mientras que el otro programa considera a *Num-telefónico* como un elemento grupal. ¿Cuál es la razón de la discrepancia?
4. Basados en el contenido de este capítulo, ¿por qué cree que FORTRAN ha sido considerado como un lenguaje inapropiado para la administración de negocios?
5. ¿Cuáles son las ventajas y desventajas que deben tomarse en cuenta al usar rellenos y al usar la cláusula SYNCHRONIZED del COBOL?
6. Encuentra una estructura para los registros del archivo de estudiantes o empleados en su medio.
7. Estudie algunos ejemplos de declaración de registros de programas en COBOL o Pascal.
8. Escriba un programa para leer los datos del registro EMPLEADO usado como ejemplo en este capítulo.
9. Por lo general todos los registros en un archivo tienen la misma estructura. Sin embargo, algunas veces existen varias estructuras de registros en un solo archivo. ¿Cuáles son las ventajas y desventajas entre a) mezclar los registros de ESTUDIANTE y CLASE en un solo archivo, y b) poner cada clase de registro en su propio archivo? Dé una situación donde la opción a) sea más apropiada y una donde la opción b) sea más correcta.

capítulo cuatro

pilas

Una de las estructuras lineales de datos más comunes es la pila. Las operaciones que definen una estructura de datos de tipo pila se presentan para, después, dar paso a la declaración y manipulación de pilas. Casi todo este capítulo está dedicado a ejemplificar el uso de las pilas.

DEFINICIONES

Lista lineal

Una *lista lineal* es una estructura de datos formada por un conjunto de elementos ordenados; el número de elementos en la lista puede variar. Denotemos la lista lineal llamada A, formada por T elementos como:

$$A = [A_1, A_2, \ldots, A_T].$$

Si T = 0, entonces A se dice que es una lista vacía o nula. Se puede borrar un elemento o insertar en cualquier posición de la lista. Así la lista puede crecer o decrecer al transcurrir el tiempo.

Pila

Una *pila* es un caso especial de una lista lineal en el cual, la inserción y supresión son operaciones que sólo pueden ocurrir en un extremo de la pila, el cual se denomina como

tope de la pila. Denotemos a TOPE(P) como el elemento tope de la pila P. Para la pila P, donde

$$P = [P_1, P_2, \ldots, P_T]$$

el TOPE(P) es P_T.

Usemos Numel(P) para denotar el número de elementos en la pila P. Numel(P) es un atributo de la pila P, el cual tiene un valor entero. Para la pila P de arriba, Numel(P) es T.

El *T*-ésimo elemento de la pila es el elemento tope. Podemos representar gráficamente una pila de varias formas: el tope puede estar en la parte superior, inferior, horizontal derecha u horizontal izquierda, como se muestra en la figura 4-1.

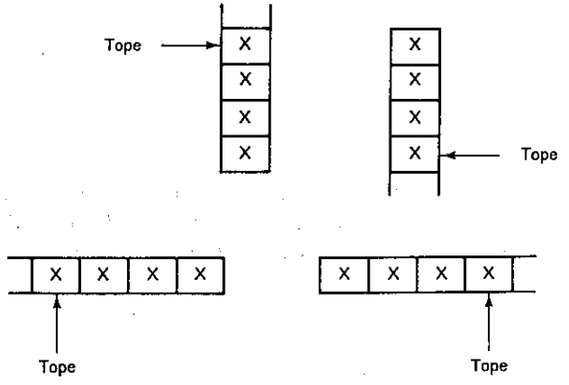


Figura 4-1. Representaciones gráficas de pilas.

La representación gráfica que se elija no importa en tanto que haya consistencia para cualquier pila en particular. Aquí generalmente usaremos la convención de tener el tope en dirección superior.

En cualquier caso, para una pila,

$$P = [P_1, P_2, \ldots, P_T],$$

diremos que el elemento P_J está arriba del elemento P_I si $J > I$. P_J está más accesible que los elementos anteriores, es decir, P_J podrá sacarse de la pila antes que cualquier elemento que esté abajo de él. P_J permanece en la pila menos tiempo que cualquier elemento anterior.

Cualquier gráfica de una pila sólo la representa en un instante de tiempo. Los elementos en el tope cambian en tanto que la pila crece o decrece, en cambio, la base permanece fija.

Ejemplos

Un ejemplo común de una pila es un apilamiento de bandejas en una cafetería. Las bandejas están sostenidas por un mecanismo de resorte, de tal forma que sólo las bandejas

en el tope son visibles y pueden utilizarse (véase la figura 4-2). Al agregar otra bandeja en la parte superior de la pila, se comprime el resorte; al retirar una bandeja, el resorte se extiende y la siguiente bandeja se encuentra accesible.

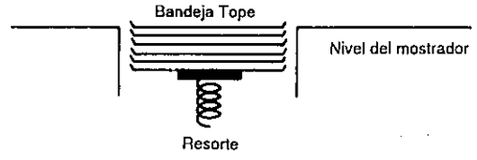


Figura 4-2 Ejemplo de una pila de bandejas.

Otro ejemplo de una pila es un sistema de rieles, usado para desviar carros de tren de una posición a otra (Figura 4-3).

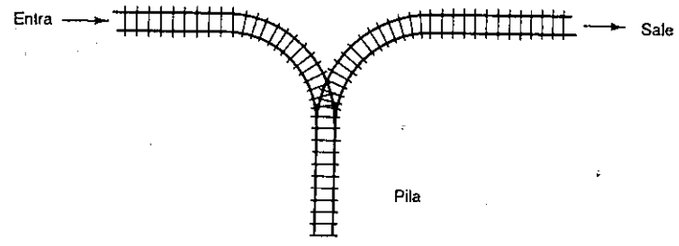


Figura 4-3 Rieles que forman una pila.

El último carro de tren que entra en la pila es el primero que puede salir de ella.

Operaciones sobre pilas

Existen cuatro operaciones básicas que son válidas para el tipo de datos pila:

1. Crear (pila)
2. Está-vacía (pila)
3. Meter (elemento, pila)
4. Sacar (pila).

El operador Crear(P) regresa una pila vacía con el nombre P. Por definición

Numel(Crear(P)) es 0
y Tope(Crear(P)) es nulo.

El operador determina cuando una pila está vacía o no. El operando es la pila; el resultado es booleano. Está-vacía(P) es verdadero si la pila P está vacía (es decir, si

Numel(P) = 0) y falso en caso contrario. Hay que notar que Está-vacía(Crear(P)) es verdadero.

A la operación de añadir un elemento a la pila se le llama meter a la pila. Meter(E,P), agrega un elemento E en el tope de la pila P, aumentando de tamaño la pila. Note que

Tope(Meter(E,P)) es E.

La operación Meter (E,P), también aumenta el Numel(P). El resultado de meter un elemento en cualquier pila no puede ser una pila vacía:

Está-vacía (Meter(E,P)) es falso.

A la operación de remover un elemento de la pila se le llama sacar de la pila. Sacar(P) remueve un elemento del tope de la pila P' disminuyendo el tamaño de la pila. Si el elemento extraído debe conservarse se deberá tomar alguna acción antes de la operación de sacar. Note que la operación Sacar(P) disminuye a Numel(P), y que sería un error intentar sacar un elemento de una pila vacía:

Sacar(Crear(P)) da una condición de error.

En ausencia de tal error, la operación Sacar(P) decrementa a Numel(P) y cambia el Tope(P).

El operador sacar deshace el resultado de la operación meter:

Sacar(Meter(E,P)) produce P.

Usted puede encontrar útil derivar otras tautologías de la combinación de las operaciones elementales sobre pilas.

Ejemplo

Comencemos con una pila vacía y consideremos los efectos de una secuencia de operaciones de Meter y Sacar. Representaremos gráficamente la pila vacía con la figura 4-4*a*).

P Numel(P) = 0, Tope(P) indefinido. Figura 4-4a)

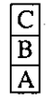
Primero se mete el elemento A, y se obtiene P = [A] (Figura 4-4*b*):

P Numel(P) = 1, Tope(P) = A, Figura 4-4b)

Después entra el elemento B, obteniéndose P = [A,B] (Figura 4-4*c*):

P Numel(P) = 2, Tope(P) = B, Figura 4-4c)

Ahora introduzcamos al elemento C, así P = [A,B,C] (Figura 4-4*d*):

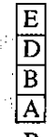


P Numel(P) = 3, Tope(P) = C. Figura 4-4d)

Si deseamos sacar un elemento en la pila, tenemos que P = [A,B] (Figura 4-4*e*):

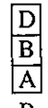
P Numel(P) = 2, Tope(P) = B, Figura 4-4e)

Después, quizás metamos dos elementos más, digamos D y E, obteniendo P = [A,B,D,E] (Figura 4-4*f*);



P Numel(P) = 4, Tope(P) = E. Figura 4-4f)

Y si quisiéramos sacar un elemento de la pila, obtendríamos P = [A,B,D] (Figura 4-4*g*);



P Numel(P) = 3, Tope(P) = D, Figura 4-4g)

Y así sucesivamente. Se dice que las pilas operan en la forma *último-en-entrar-primero-en-salir* (en inglés: last-in-first-out o LIFO). Es decir, los elementos se remueven en orden inverso al orden en que fueron insertados en la pila.

PILAS EN COBOL Y PASCAL

Pilas alojadas en arreglos

Aunque las pilas son estructuras muy usadas, la mayoría de los lenguajes de programación no tienen predefinida la estructura de datos tipo *pila*. Para resolver este problema,

el programador deberá usar las características existentes del lenguaje para simular las operaciones sobre una pila. Existen varias formas de representar pilas, quizás la forma más simple para representar pilas es alojarlas en arreglos. En esta sección introducimos este método para manipular pilas en COBOL y Pascal.

Una pila se puede alojar en un arreglo, pero es muy importante distinguir que una pila y un arreglo son dos estructuras de datos con diferentes reglas. Primero, recuerde que no hay restricciones para insertar o borrar elementos en alguna parte del arreglo. Lo cual hace que el programador sea el responsable de poner en vigor las reglas LIFO para una pila que se aloja en un arreglo. Segundo, al alojar una pila en un arreglo, inmediatamente se restringe a la pila para contener elementos homogéneos. En ningún momento el concepto de pila previene que su contenido no pueda ser de diferentes tipos de elementos. Otra restricción artificial impuesta, es que el programador debe establecer el límite superior para los subíndices del arreglo, a diferencia de la estructura de pila que no restringe al número máximo de elementos que puede contener (de bandejas en una cafetería, o fichas de póker por ejemplo). Sin embargo, una pila variable alojada en un arreglo queda restringida al espacio asignado para ese arreglo. De hecho, la pila crece o decrece con el tiempo, pero un arreglo tiene un tamaño constante.

Declaración de pilas

Consideremos la declaración de una variable de tipo pila llamada P. Asumamos que cada elemento de P será un entero y que P tendrá un máximo de 100 elementos. Además de declarar el arreglo que alojará a P, tenemos que declarar una variable APUNTADOR-TOPE, cuyo valor será el subíndice del elemento tope de la pila. Llamemos a la combinación de arreglo y tope como ESTRUCTURA-PILA. Con esta representación, Numel(P) = APUNTADOR-TOPE. La función Está-vacía(P) es verdadera cuando APUNTADOR-TOPE = 0, y falsa cuando APUNTADOR-TOPE > 0.

En COBOL:

```
01      ESTRUCTURA-PILA.
        02      P                       OCCURS 100 TIMES PICTURE 9(5).
        02      APUNTADOR-TOPE          PICTURE 9(3).
```

En Pascal:

```
type estructura-pila =
        record          pila: array [1. .100] of integer;
                        apuntador-pila: integer
        end;
var     p: estructura-pila;
```

Si se utilizan varias pilas en un mismo programa, cada una necesitará su propio apuntador de tope. (Nota: En COBOL la palabra TOP es palabra reservada.)

Operaciones sobre pilas

Un compilador no detecta las violaciones a las reglas de operación LIFO para una pila alojada en un arreglo, por lo tanto, el programador debe tener cuidado de no indexar cualquier elemento en P a excepción del elemento indicado por APUNTADOR-TOPE.

Las operaciones de meter y sacar elementos de P se pueden programar como sigue: se puede usar EON para indicar el elemento que deberá meterse dentro de P y EOFF para indicar el elemento que deberá sacarse de P (Note que almacenaremos a EOFF); se usará la variable NUMEL-MAX para indicar el número máximo de elementos que el arreglo P puede contener; aquí NUMEL-MAX = 100. La aplicación determina las acciones a tomar cuando ocurre una condición de desborde (cuando se intenta meter elementos en una pila llena), o de subdesborde (cuando se intenta sacar elementos de una pila vacía).

En COBOL los párrafos son:

```
METER.
        IF APUNTADOR-PILA < NUMEL-MAX
        THEN COMPUTE APUNTADOR-TOPE = APUNTADOR-TOPE + 1
                MOVE EON TO P (APUNTADOR-TOPE)
        ELSE condición-de-desborde

SACAR.

        IF APUNTADOR-TOPE > 0
        THEN MOVE P (APUNTADOR-TOPE) TO EOFF
                COMPUTE APUNTADOR-TOPE = APUNTADOR-TOPE - 1
        ELSE condición-de-subdesborde.
```

En Pascal los procedimientos son:

```
procedure meter (eon : integer);
begin if (p. apuntador-tope < numel-max)
    then begin p.apuntador-tope : = p.apuntador-tope + 1;
                p.pila [p.apuntador-tope] : = eon
        end
    else CONDICION-DESBORDE
end;
procedure sacar (eoff : integer);
begin if (p.apuntador-tope > 0)
    then begin eoff: = p.pila [p.apuntador-tope];
                p.apuntador-tope : = p.apuntador-tope - 1
        end
    else CONDICION-SUBDESBORDE
    end;
```

Se supone que P es una variable global conocida.

Es importante tener precaución sobre la integridad de la pila, la cual se debe mantener por el programador, cuando la pila se aloja en un arreglo. Es responsabilidad del programador asegurar que el apuntador de tope sólo sea usado como punto de referencia de la pila.

Hay otras formas de representar pilas, aun cuando no se tengan incorporadas las estructuras de datos de tipo pila. Analizaremos estas representaciones más adelante, en el capítulo 6, que trata sobre listas ligadas.

EJEMPLOS DE APLICACIONES DE PILAS

Las pilas se utilizan mucho en la resolución de una gran variedad de problemas. Se usan en compiladores, en sistemas operativos y en diversos programas de aplicación. En esta sección ejemplificaremos sólo algunos usos de las pilas; aunque existen muchos otros. Veremos tres ejemplos y utilizaremos uno para ilustrarlo con nuestros lenguajes de programación (COBOL, Pascal).

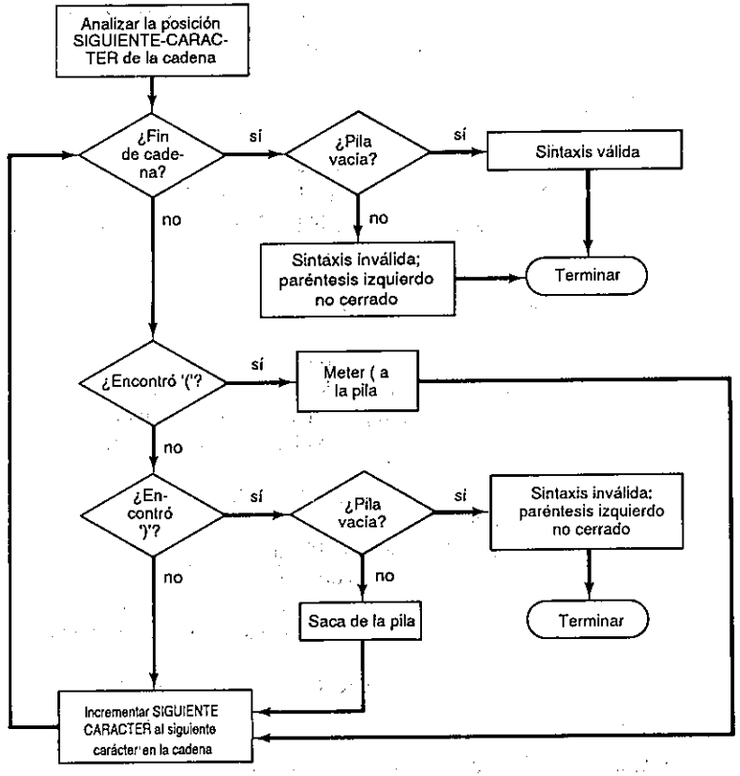


Figura 4-5 Lógica para apareamiento de paréntesis.

Ejemplo: correspondencia de paréntesis

Una de las tareas del compilador es la de verificar que el programador ha especificado de forma correcta las reglas gramaticales (sintaxis) del lenguaje de programación. Considere el problema de verificación de sintaxis que consiste en asegurar que cada paréntesis izquierdo tiene su paréntesis derecho correspondiente. Una pila se puede usar para facilitar el procedimiento de apareamiento.

El algoritmo es simple. Rastreamos la cadena de elementos de izquierda a derecha. Cada vez que encontramos un paréntesis izquierdo lo metemos en la pila. Cada vez que encontramos un paréntesis derecho revisamos el contenido de la pila. Si está vacía entonces habremos encontrado un paréntesis derecho que no cierra un paréntesis izquierdo, y tendremos un error. Si la pila no está vacía habremos encontrado el par, y sólo lo sacaremos de la misma. Si la pila no está vacía al terminar la cadena, entonces hay un paréntesis izquierdo sin cerrar. En cualquier punto del rastreo, el número de elementos en la pila es la profundidad de anidamiento de los paréntesis, que varía de acuerdo al momento en el que se halle la búsqueda. Este algoritmo se muestra en el diagrama de flujo de la figura 4-5.

Programamos esta rutina en COBOL. La cadena a ser rastreada se almacena, carácter por carácter, en un arreglo llamado CADENA. La pila se aloja en un arreglo llamado PILA. Supongamos que el número máximo de caracteres en la cadena es de 80 y que la cadena termina con un punto y coma (;). Las estructuras se definen por:

```
01  ESTRUCTURA-PILA.
    02  PILA                    OCCURS 80 TIMES
                                 PICTURE X.
    02  APUNTADOR-TOPE          PICTURE 99 VALUE CERO.
01  CADENA.
    02  CARACTER                OCCURS 80 TIMES
                                 PICTURE X.
01  SIGUIENTE-CARACTER          PICTURE 99.
```

Estas estructuras se manejan por el siguiente código:

```
PERFORM EXAMINA-SIG-CARACTER
        VARYING SIGUIENTE-CARACTER FROM 1 BY 1
        UNTIL SIGUIENTE-CARACTER > 80
            OR CARACTER(SIGUIENTE-CARACTER) = ";".
IF APUNTADOR-TOPE = 0
    sintaxis válida
ELSE sintaxis-inválida-paréntesis-izquierdo-no-cerrado.
```

donde

```
EXAMINA-SIG-CARACTER.
    IF CARACTER(SIGUIENTE-CARACTER) = "(" PERFORM METE
```

```
        ELSE IF CARACTER(SIGUIENTE-CARACTER) = "("
            PERFORM SACA.
    METE.
        COMPUTE APUNTADOR-TOPE = APUNTADOR-TOPE + 1.
        MOVE CARACTER(SIGUIENTE-CARACTER) TO PILA
        (APUNTADOR-TOPE).
    SACA.
        IF APUNTADOR-TOPE = 0
            Sintaxis-inválida-paréntesis-izquierdo-no-cerrado
        ELSE COMPUTE APUNTADOR-TOPE = APUNTADOR-TOPE - 1.
```

En realidad, para el propósito de este problema, en el cual no vemos los elementos que se sacan de la pila, es suficiente con registrar el número de elementos que contiene la pila en un momento dado. Por tanto, en este ejemplo la pila puede ser virtual.

Un problema más interesante de verificación sintáctica es la construcción de un algoritmo para aparear no sólo paréntesis, sino también llaves y corchetes izquierdos con derechos, en una misma cadena. Por ejemplo, dada la siguiente cadena como entrada, el algoritmo debe ser capaz de determinar si el apareamiento de corchetes es correcto o incorrecto.

$$(\{A + B*C/([D + E - (F + G)*(A + B)])*(F - G + B)\}/(D - E - (A + B))).$$

El desarrollo y la codificación del algoritmo se deja como ejercicio. ¿Cuántas pilas son necesarias?

En este problema, como en muchos otros que utilizan pilas, la pila contiene un registro de obligaciones pospuestas, las cuales necesitan ser atendidas en orden LIFO.

Ejemplo: recursión

Las pilas se usan comúnmente para indicar en qué punto se encuentra un programa cuando contiene procedimientos que se llaman a sí mismos. Estos procedimientos se conocen como procedimientos *recursivos.*

Por ejemplo, considere el problema de convertirse en millonario. Para tener un millón de dólares dentro de 25 años, ¿cuánto dinero necesita depositar hoy en el banco, con interés compuesto del 8% anual? Una forma de programar este problema es a través del uso de procedimientos recursivos. Veamos el problema de la siguiente manera: supongamos que han pasado 25 años y ahora tiene un millón de dólares. ¿Cuánto dinero tuvo en el banco en el año 24? en esa época usted tuvo una cantidad que, al aplicarle el 8% al año siguiente, se convirtió en un millón de dólares. La cantidad es un millón dividido entre 1.08. ¿Cuánto necesitó tener en el año 23?, y así sucesivamente.

El siguiente programa en Pascal usa un procedimiento recursivo llamado compuesto para calcular el dinero que necesita depositar el primer año para obtener en 25 años un millón.


```
        program retiro(output);
        const objetivo = 1000000; tasa = 0.08;
        var ahora-tengo:real; año:integer;
1       procedure compuesto (var año: integer); {determina el monto del ahorro
          1 año antes}
2       begin if (año > 0)then
3               begin ahora-tengo: = ahora-tengo/(1.0 + tasa);
4                       año : = año - 1;
5                       compuesto (año) end;
6       end; {compuesto}
7       begin año := 25; {bloque programa principal}
8               ahora-tengo := objetivo;
9               compuesto (año);
10              write (ahora-tengo); writeln;end;
11      end.
```

El valor de la variable año cambia cada vez que el procedimiento denominado compuesto es llamado, comenzando con 25 y terminando la ejecución del programa cuando es igual a cero. El sistema utiliza una pila para registrar dónde se encuentra el programa de retiro, en un momento dado, durante la ejecución. Cada vez que el procedimiento compuesto se invoca, la dirección de llamada se mete en la pila. Cada vez que el compuesto termina, se extrae la dirección de la pila y se regresa el control a la dirección que se encontraba en el tope. La primera llamada a compuesto se da en la instrucción 9, y la pila de control está como se muestra en la figura 4-6*a*.

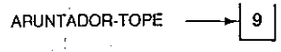


Figura 4-6a)

La segunda invocación a compuesto proviene de la instrucción 5, cuando año tiene el valor de 24. La pila de control está como se muestra en la figura 4-6*b*.

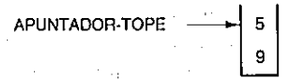


Figura 4-6b)

La tercera llamada a compuesto proviene de la instrucción 5, cuando año tiene el valor de 23. La pila de control, entonces, se encuentra como se muestra en la figura 4-6*c*.

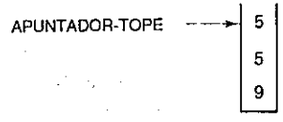


Figura 4-6c)

Después de 25 llamadas al procedimiento compuesto, la pila está como se muestra en la figura 4-6*d*.

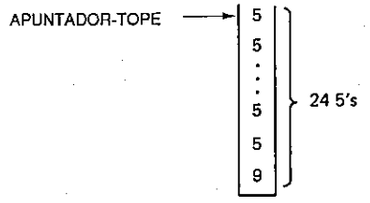


Figura 4-6d)

Hubo 24 llamadas a la instrucción 5, y año finalmente tiene el valor de 0. Ahora la pila entra en acción a medida que las llamadas se desencadenan. Cada invocación a compuesto necesita ser regresada en una secuencia (LIFO), hasta que se alcanza la llamada original. El regreso de la última llamada a compuesto resulta en una extracción de la pila, tal y como se muestra en la figura 4-6*e*.

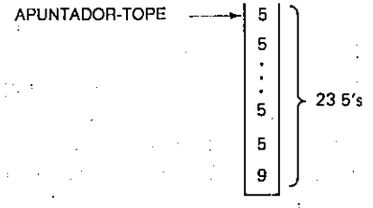


Figura 4-6c)

Además, todas las direcciones se extraen de la pila de control hasta regresar a la instrucción 9. La siguiente instrucción escribe el valor determinado de la variable ahora-tengo, el cual es la cantidad que necesita depositar.

En este ejemplo, el compilador de Pascal estableció la necesidad de la pila y proporcionó las instrucciones apropiadas para las operaciones de meter y sacar de la pila. En el ejemplo anterior, el programador en COBOL cargaba con toda la responsabilidad del manejo de la pila.

Algunos compiladores no son capaces de manipular llamadas a procedimientos recursivos porque no tienen los mecanismos de pila necesarios. Antes de programar y esperar que la recursividad funcione (especialmente en COBOL o FORTRAN) consulte sus manuales de lenguaje. Desafortunadamente, los procedimientos recursivos en estos lenguajes por lo común compilan correctamente en apariencia, pero fallan durante la ejecución.

Ejemplo: Notación postfija

Una tercera aplicación de pilas es la compilación de expresiones aritméticas en lenguajes de programación de alto nivel. Los compiladores necesitan ser capaces de traducir de la forma usual de presentación de instrucciones aritméticas (llamada *notación infija*) a una forma más fácil de usar para la generación del código objeto. Para operadores binarios, como la suma, resta, multiplicación, división y exponenciación, los operadores en notación infija aparecen entre dos operadores, es decir, $A + B$, $E \uparrow F$. Las pilas pueden usarse para transformar esta notación a *notación postfija*, en la cual los dos operandos

aparecen antes del operador, es decir, $AB +$, $EF \uparrow$. Esta forma es más fácil de manipular por el compilador, como veremos en seguida.

Considere el siguiente algoritmo, el cual convierte expresiones de notación infija a postfija. En una pila se retendrán los operadores infijos hasta que sus operandos hayan

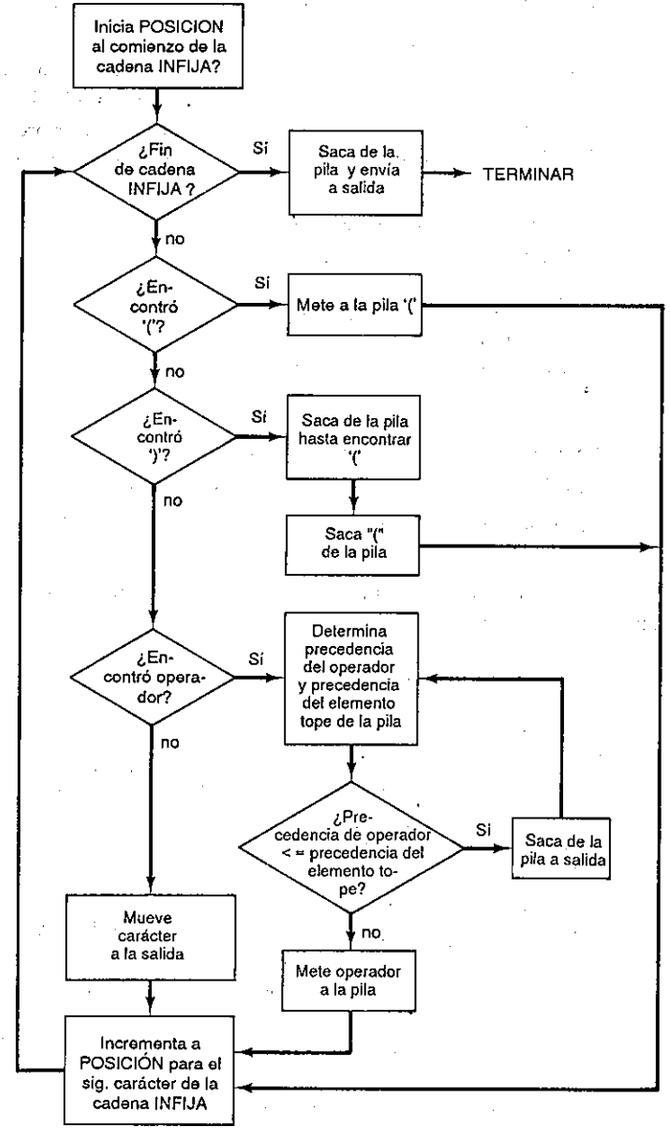


Figura 4-7 Lógica para convertir expresiones de notación infija a postfija.

sido examinados. La expresión se examina de izquierda a derecha. Existen cuatro reglas básicas.

1. Si el símbolo es un "("
   éste se mete a la pila de operadores.
2. Si el símbolo es un ")"
   se saca de la pila todo lo que exista hasta llegar al primer "(". Los operadores van a la salida, a medida que salen de la pila. El "(" se saca pero no va a la salida.
3. Si el símbolo es un operador,
   entonces si el operador en el tope de la pila es de la misma o de mayor precedencia, dicho operador se saca y va a la salida, continuando de esta manera hasta que el primer paréntesis izquierdo o un operador de menor precedencia se encuentre en la pila. Cuando esta situación ocurre, entonces, el operador en turno se mete a la pila.
4. Si el símbolo es un operando,
   éste se envía directamente a la salida.

El símbolo de terminación, aquí es el punto y coma, saca todos los símbolos de la pila. El algoritmo se muestra en la figura 4-7.

Consideremos primero el algoritmo con sólo tres niveles de precedencia en operadores:

Exponenciación (↑), el más alto nivel.

Multiplicación (*), división (/), de medio nivel.

Suma (+), resta (-), el nivel más bajo.

La figura 4-8 muestra la pila de operadores y la salida a medida que se examina cada carácter de la expresión aritmética infije:

$((A + B) * C/D + E \uparrow F)/G;$

Punto temporal	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Carácter	(	(	A	+	B	)	*	C	/	D	+	E	↑	F	)	/	G	;
Pila de operadores →	(	(((	(((	+ ((	+ ((	(	* ((	* ((	/ ((	/ ((	+ ((	+ ((	↑ + ((	↑ + ((		/		
Salida			A		B	+		C	*	D	/	E		F	↑ +		G	/

Figura 4-8. Forma de examinar el contenido de la pila.

El resultado en notación postfija es:

$AB + C * D/EF \uparrow + G/.$

Note que todos los paréntesis se han removido de la expresión. Los paréntesis eran necesarios en la notación infija para indicar la precedencia deseada de operadores. En notación postfija, la precedencia de operadores se indica de acuerdo al orden de los operadores. La cadena postfija se puede rastrear de izquierda a derecha, estableciendo expresiones intermedias que involucren sólo operaciones binarias, como sigue:

Paso 1. $T_1 = AB +$	la cadena se convierte en: $T_1\ C*D/EF\uparrow\ +G/$
Paso 2. $T_2 = T_1\ C*$	la cadena se convierte en: $T_2 D/EF\uparrow\ +G/$
Paso 3. $T_3 = T_2 D/$	la cadena se convierte: $T_3 EF\ \uparrow\ +G/$
Paso 4. $T_4 = EF\uparrow$	la cadena se convierte en: $T_3\ T_4\ +\ G/$
Paso 5. $T_5 = T_3\ T_4\ +$	la cadena se convierte en: $T_5\ G/$
Paso 6. $T_6 = T_5\ G/$	la cadena se convierte en: T_6

Como podrá observar, ningún operador se rastrea hasta después de que sus dos operandos hayan sido rastreados. Si está familiarizado con un lenguaje ensamblador, puede resultar obvio el tipo de código que el compilador deberá generar para implantar los seis pasos anteriores.

Este algoritmo se puede programar en COBOL como sigue: Primero las variables se declaran en la DIVISION DE DATOS.

```
01 PILA-OPERADORES.
   02 PILA OCCURS 100 TIMES                PICTURE X.
   02 APUNTADOR-TOPE                       PICTURE 9(3).
   02 NUMERO-DE-ELEMENTOS                  PICTURE 9(3)
                                                VALUE 100.
01 CADENA-DE-ENTRADA.
   02 CADENA-INFIJA OCCURS 100 TIMES       PICTURE X.
01 MANEJO
   02 POSICION                             PICTURE 9(3).
   02 SIG-CARACTER                         PICTURE X.
   02 ELEMENTO-SACADO                      PICTURE X.
   02 OP-PRECEDENCIA                       PICTURE 9.
   02 PRECEDENCIA-TOPE                     PICTURE 9.
01 REGLA-DE-PRECEDENCIA.
   02 CARACTER-DE-VERIFICACION             PICTURE X.
      88 NIVEL-0                           VALUE '+', '-'.
      88 NIVEL-1                           VALUE '*', '/'.
      88 NIVEL-2                           VALUE '↑'.
      88 NIVEL-3                           VALUE '('.
   02 NIVEL-DE-PRECEDENCIA                 PICTURE 9.
```

En la *DIVISION DE PROCEDIMIENTOS* el conductor es como sigue:

```
COMPUTE APUNTADOR-TOPE = 0.
PERFORM EXAMINA-SIG-CARACTER
        VARYING POSICION FROM 1 BY 1
        UNTIL POSICION > NUMERO-MAX-DE-ELEMENTOS OR
          CADENA-INFIJA (POSICION)
          = ';'.
PERFORM SACA-DE-LA-PILA-A-SALIDA
        UNTIL APUNTADOR-TOPE = 0.
```

y

```
EXAMINA-SIG-CARACTER.
        MOVE CADENA-INFIJA (POSICION) TO SIG-CARACTER.
        IF SIG-CARACTER = '('
                PERFORM METE-A-LA-PILA.
        ELSE IF SIG-CARACTER = ')'
                PERFORM SACA-DE-LA-PILA-A-SALIDA
                        UNTIL PILA (APUNTADOR-TOPE) = '('
                PERFORM SACA-DE-LA-PILA
              ELSE IF SIG-CARACTER = '↑' OR '*' OR '/' OR '+' OR '-'
                     PERFORM ENCUENTRA-OP-PRECEDENCIA
                     PERFORM ENCUENTRA-PRECEDENCIA-TOPE
                     PERFORM PILA-VACIA-A-MENOR-PRECEDENCIA
                        UNTIL OP-PRECEDENCIA >
                        PRECEDENCIA-TOPE
                     PERFORM METE-A-LA-PILA
                   ELSE DISPLAY SIG-CARACTER.
METE-A-LA-PILA.
        COMPUTE APUNTADOR-TOPE = APUNTADOR-TOPE + 1.
        IF APUNTADOR-TOPE> NUMERO-MAX-DE-ELEMENTOS
                condición-de-error
        ELSE MOVE SIG-CARACTER TO PILA ( APUNTADOR-TOPE).
SACA-DE-LA-PILA.
        IF APUNTADOR-TOPE = 0
                condición-de-error
        ELSE MOVE PILA (APUNTADOR-TOPE) TO ELEMENTO
                -SACADO COMPUTE APUNTADOR-TOPE =
                  APUNTADOR-TOPE - 1.
SACA-DE-LA-PILA-A-SALIDA.
        PERFORM SACA-DE-LA-PILA.
        DISPLAY ELEMENTO-SACADO.
ENCUENTRA-OP-PRECEDENCIA.
        MOVE SIG-CARACTER TO CARACTER-DE-VERIFICACION.
        PERFORM ENCUENTRA-PRECEDENCIA.
        MOVE NIVEL DE-PRECEDENCIA TO OP-PRECEDENCIA.
ENCUENTRA PRECEDENCIA-TOPE.
        IF APUNTADOR-TOPE > 0
                MOVE PILA (APUNTADOR-TOPE) TO CARACTER-
                  DE-VERIFICACION
                PERFORM ENCUENTRA-PRECEDENCIA
                MOVE NIVEL-DE-PRECEDENCIA TO PRECEDENCIA-
                  TOPE
          ELSE MOVE 0 TO PRECEDENCIA-TOPE
PILA-VACIA-A-MENOR-PRECEDENCIA.
        PERFORM SACA-DE-LA-PILA-A-SALIDA.
        PERFORM ENCUENTRA-PRECEDENCIA-TOPE.
ENCUENTRA-PRECEDENCIA.
        IF NIVEL-0
                COMPUTE NIVEL-DE-PRECEDENCIA = 0
        ELSE IF NIVEL 1
                  COMPUTE NIVEL-DE-PRECEDENCIA = 1
                ELSE IF NIVEL 2
                    COMPUTE NIVEL-DE-PRECEDENCIA = 2
                  ELSE IF NIVEL 3
                            COMPUTE NIVEL-DE-PRECEDENCIA = 3
                    ELSE condición-de-error.
```


Una forma alternativa para la representación de una expresión aritmética es la forma *prefija* en donde los operadores preceden a sus operandos, por ejemplo, *+AB*, *↑ EF*. Un ejercicio al final del capítulo pregunta qué cambios necesita hacer al algoritmo anterior para obtener cadenas en notación prefija en lugar de postfija.

El ejercicio también sugiere que considere una mejora para que el algoritmo tome en cuenta el examen de los operadores relacionales (<, < =, =, > =, >), los operadores booleanos (and, or, not) y los operadores aritméticos unitarios (-, +). Para auxiliarlo en su esfuerzo, anotamos a continuación la precedencia usual de los operadores

Nivel 6: Operador unitario -, unitario +, *not*

Nivel 5: Exponenciación ↑

Nivel 4: Multiplicación, división, *, /

Nivel 3: Suma, resta +, -

Nivel 2: Operadores relacionales <, < =, =, > =. >

Nivel 1: Operador booleano *and*

Nivel 0: Operador booleano *or*

Note que también hemos asignado una prioridad alta al "(" en la rutina de COBOL anterior, lo cual facilita la manipulación de la pila.

Más adelante, en el capítulo 8, volveremos a discutir las notaciones prefija, infija, y postfija, cuando estudiemos los árboles binarios.

FORMAS DE ALMACENAMIENTO

Ya discutimos el uso de arreglos para alojar pilas. También vimos que un arreglo unidimensional es, por lo general, almacenado de manera físicamente secuencial. Se reserva un conjunto contiguo de direcciones para los elementos del arreglo. La forma más sencilla para almacenar pilas sigue el mismo principio. A una pila se le asigna una localidad base, la cual se mantiene fija. Después la pila puede crecer sobre las localidades contiguas.

Espacio compartido

Hasta aquí consideramos la reservación de espacio para una sola pila. Suponga que ahora tenemos dos pilas que necesitan coexistir en memoria. Si la pila P1 tiene un máximo de *M* elementos y la pila P2 tiene un máximo de *N* elementos, se les puede asignar memoria como se muestra en la figura 4-9.

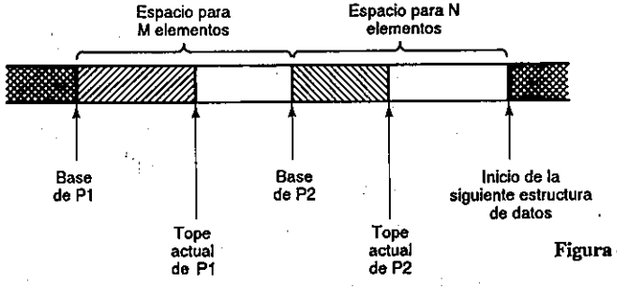


Figura 4-9 Asignación de espacio para dos pilas.

Esto no es particularmente eficiente, en especial si las pilas P1 y P2 nunca están al mismo tiempo llenas en su totalidad.

Considere el caso, en el cual sabemos que el número total de elementos en P1 y P2 *combinados*, nunca excederán de *N* elementos:

$$\text{Numel(Pl)} + \text{Numel(P2)} \leq N.$$

Por ejemplo, suponga que hay 500 personas resolviendo una prueba de aptitud en programación. En la actualidad algunos son empleados profesionales en computación; y el resto no lo son. Antes de terminar la prueba, no conocemos cuantas personas hay en cada categoría. Sin embargo, queremos registrar la secuencia en la cual las personas terminan la prueba, usando una pila para cada categoría. Un enfoque es reservar espacio para que cada pila crezca hasta su máximo número de *N* elementos, como ya fue planteado. En cualquier momento dado, habrá espacio reservado para al menos *N* elementos. Una forma alternativa sería disponer de espacio sólo para un máximo combinado de *N*, poniendo a las dos pilas una frente a la otra y que éstas fueran creciendo hasta encontrarse, tal y como se muestra en la figura 4-10.

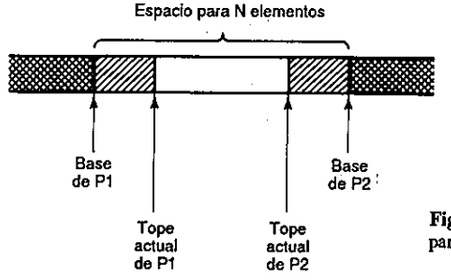


Figura 4-10. Forma alterna para asignar espacio para dos pilas, las cuales crecen una frente a otra.

P1 y P2 nunca se traslaparán puesto que ninguna alcanzará la condición de sobre-flujo. Unidas pueden alojar *N* elementos. La pila P1 crece a la derecha y la pila P2 crece a la izquierda.

Considere ahora el problema de reservación de memoria para tres o más pilas, tales que 1) la condición de desborde ocurra sólo cuando el tamaño total de todas las pilas, juntas, excedan el espacio disponible, y 2) cada pila tenga una localidad base fija. De hecho, no es posible que se cumplan ambas condiciones al mismo tiempo para tres o más pilas. Cuando sea necesario reservar memoria para varias pilas, una o ambas condiciones deberán ser flexibles. O bien, se tendrá que reservar más espacio o las bases de las pilas tendrán que cambiarse a otra dirección. Una técnica eficiente para reservar espacio cuando hay más de dos pilas es la técnica llamada de Garwick, la cual periódicamente reasigna espacio para las pilas. Durante el tiempo de reasignación, el crecimiento de las pilas se mide y el espacio se prorratea de tal manera que cada pila recibe una parte proporcional a su crecimiento desde la última reasignación. En esta técnica las bases de las pilas se van moviendo a otras direcciones; aquellas pilas que tienden a crecer obtienen más espacio.

En el capítulo 6 estudiaremos otras formas para reservar espacio para pilas en memorias no contiguas, utilizando listas ligadas. Por lo general no es posible predecir el tamaño máximo que puede llegar a obtener una pila; por lo cual los esquemas de almacenamiento dinámico (como la técnica de Garwick) y los esquemas de almacenamiento no contiguo, son los que se usan con más frecuencia.

RESUMEN

Una *pila* es un caso especial de una lista lineal en la cual las operaciones de inserción (Meter) y supresión (Sacar) son estrictamente efectuadas sobre un extremo de la pila, el cual es llamado tope de la pila. Las pilas se pueden alojar en arreglos. Este enfoque impone algunas restricciones en el uso de las pilas, pero se utiliza con frecuencia. La integridad de una pila debe mantenerse cuando ésta es alojada en arreglos. Es responsabilidad del programador asegurar que la inserción y supresión se efectúen sólo en el elemento tope de la pila. En el capítulo se introdujo la declaración y manipulación de pilas en COBOL y Pascal.

Los ejemplos ilustraron que las pilas pueden emplearse en la solución de problemas que necesitan de una estructura de datos tipo último-que-entra-primero-que-sale. Un

ejemplo de llamadas recursivas para determinar la inversión de capital mostró la necesidad de regresar a una dirección de llamada. Para ello se usó una pila para conservar el registro de estas direcciones en orden de las llamadas. El ejemplo de notación postfija utilizó un patrón LIFO, para identificar cuales operandos pertenecían a algún operador particular, así como un patrón de anidamiento de paréntesis.

El capítulo terminó con una discusión sobre asignación de memoria para pilas múltiples.

TERMINOLOGIA

Lista lineal
Meter
Notación infija
Notación postfija
Notación prefija
Pila
Procedimiento recursivo
Sacar
Sintaxis
Tope
Ultimo-en-entrar-primero-en-salir (LIFO)

REFERENCIAS SUGERIDAS

GARWICK, J. "Data storage in compilers," *BIT*, 4: 137–140, 1964.

KNUTH, D. E. *The Art of Computer Programming, Vol. I, Fundamental Algorithms*. Reading, Mass.: Addison-Wesley Publishing Co., 1973, pp. 234–251.

KORSH, J. F. and G. LAISON. "A multiple-stack manipulation procedure," *Comm. ACM* 26(11): 921–923, Nov. 1983.

STANDISH, T. A. "Stacks and queues," Chapter 2 in *Data Structure Techniques*. Reading, Mass.: Addison-Wesley Publishing Co., 1980, pp. 28–41.

YEH, D. Y. and T. MUNAKATA. "Dynamic allocation and local reallocation procedures for multiple stacks," *Comm. ACM* 29(2): 134–141, Feb. 1986.

EJERCICIOS DE REPASO

1. Dibuje una pila e indique la dirección de entrada y salida de los elementos.
2. Llene las líneas en blanco:
   a) Si aplica una operación de sacar en una pila vacía, provocará un(a):_____.
   b) Si aplica correctamente la operación Meter el elemento de i a una pila, Tope =_____.
   c) El tope de una pila vacía es igual a: _____.
   d) Si primero saca y después Mete el elemento i, Tope = _____.
   e) ¿Cuántos elementos hay en una pila de nueva creación? _____.
3. ¿Cuál es el resultado de las siguientes aseveraciones?
   a) Está-vacía (Crear(P)), tiene el valor de: _____.

   b) Está-vacía (Meter(i, P)), tiene el valor de:_____.
   c) Sacar (Crear(P)), tiene el valor de _____.
4. ¿Qué resultado se obtiene al aplicar Tope (Meter(i,P))?: _____.
5. ¿Cuál es el resultado que se obtiene si aplica Sacar (Meter(i, P))?_____.
6. ¿Cómo puede implantar una pila en lenguaje FORTRAN?
7. Escriba un algoritmo que agregue elementos a una pila.
8. Escriba un algoritmo que extraiga los elementos de una pila.
9. Escriba un algoritmo que cree y llene una pila.
10. Consigne tres aplicaciones de pilas. Para cada una describa las razones de por qué las pilas serían preferibles a los arreglos.
11. El algoritmo METER verifica la condición de desborde y el algoritmo SACAR verifica la condición de subdesborde. ¿Qué significado tienen las condiciones de desborde y subdesborde y por qué es aconsejable verificarlas? ¿Cuál podría ser la acción adecuada a tomar bajo cada condición?
12. ¿Cuál es la forma postfija para cada una de las siguientes expresiones?:
    a) $A * B - (C + D) - (E - F) + F/H \uparrow I$
    b) $((B * C) + C/D \uparrow F) + G$
    c) $A \uparrow B * C - D + E / F / (G + H)$
13. Escriba un algoritmo para sacar un elemento de una pila y asignar el valor sacado a una variable *x*.
14. Escriba un programa que use una pila para verificar la existencia de paréntesis izquierdo y derecho, corchetes izquierdo y derecho, y llaves izquierda y derecha en una cadena de caracteres.
15. Escriba un algoritmo que use una pila para convertir una expresión aritmética de notación infija a notación prefija.
16. Escriba un programa para implantar su algoritmo de conversión infija a prefija.
17. Escriba un programa en lenguaje Pascal para convertir expresiones aritméticas de notación infija a postfija.
18. Escriba un programa para convertir una expresión aritmética de notación infija a postfija, donde la expresión pueda contener los siguientes operadores: <, <=, =, >=, >, *and, or, not,* *T*, - unario, + unario, ↑, *, /, +, -.
19. La siguiente PILA contiene los elementos:

    3
    4
    1
    8
    5
    6

    Escriba un algoritmo para crear una nueva pila que contenga a los elementos de la PILA de arriba, en orden ascendente. Minimice el número de pilas intermedias usadas.