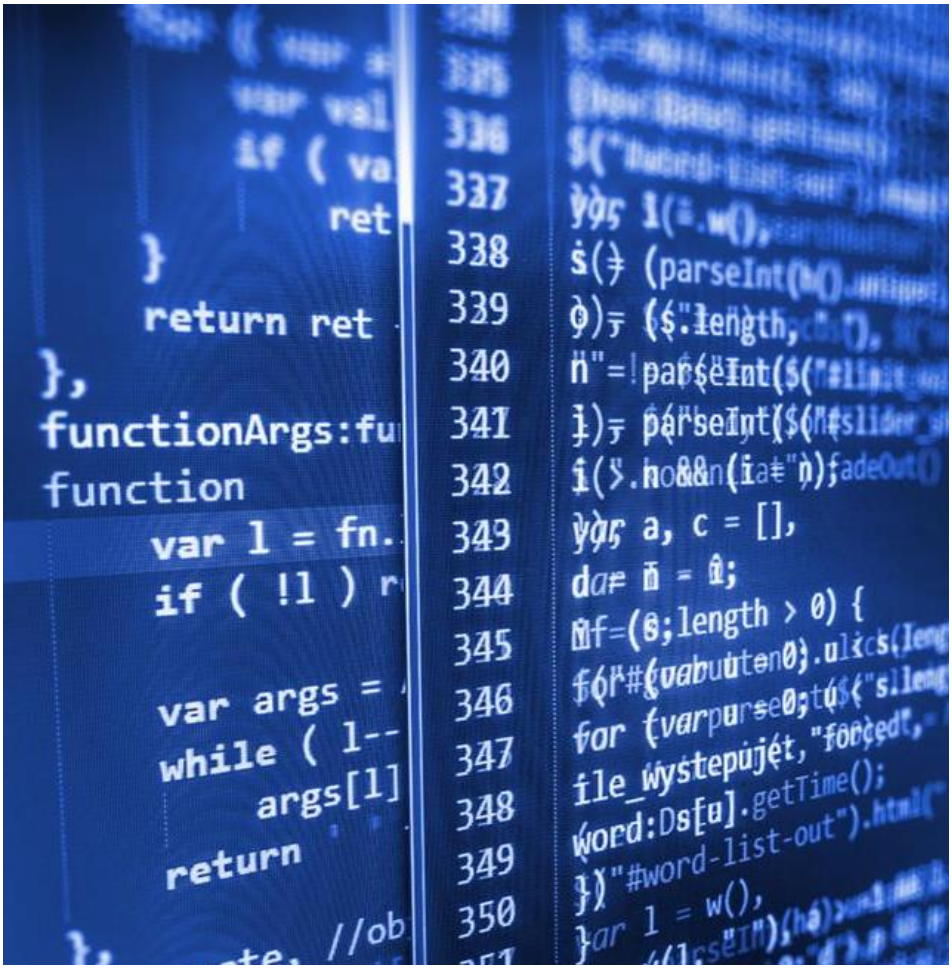


PRACTICA 1

EVALUACION DE EXPRESIONES INFIJAS

Equipo: DAG TEAM

INTITUTO POLITECNICO NACIONAL

ESCOM

ESCOM

INTEGRANTES

OLEDO ENRIQUEZ GILBERTO IRVING

ANALIS RAMIREZ DAMIAN

MENDIETA TORRES ALFONSO ULISES

GRUPO
1CM13

UNIDAD DE APRENDIZAJE

Estructuras de Datos

Practica 01: Evaluación de expresiones infijas.

Introducción.

Tipos de datos abstractos.

Una abstracción es la simplificación de un objeto o de un proceso de la realidad en la que sólo se consideran los aspectos más relevantes.

La abstracción se utiliza por los programadores para dar sencillez de expresión al algoritmo.

La abstracción tiene dos puntos de vista en programación:

1. Funcional.

2. De datos.

Tipos de abstracción.

La abstracción funcional: - Permite dotar a la aplicación de operaciones que no están definidas en el lenguaje en el que se está trabajando.

- Se corresponden con el mecanismo del subprograma (acción que se realiza y argumentos a través de los cuales toma información y devuelve resultados).

- Es irrelevante cómo realiza la acción y no importa su tiempo de ejecución.

Las abstracciones de datos (= Clase): - Permiten utilizar nuevos tipos de datos que se definirán especificando sus posibles valores y las operaciones que los manipulan.

- Cada operación constituye una abstracción funcional.

Definición Un tipo abstracto de datos (TAD) es un tipo definido por el usuario que:

- Tiene un conjunto de valores y un conjunto de operaciones.

- Cumple con los principios de abstracción, ocultación de la información y se puede manejar sin conocer la representación interna. Es decir, los TADs ponen a disposición del programador un conjunto de objetos junto con sus operaciones básicas que son independientes de la implementación elegida.

Metodología de la programación de un TAD.

Debido al proceso de abstracción, la programación de un TAD deberá realizarse siguiendo tres pasos fundamentales:

1. Análisis de datos y operaciones.

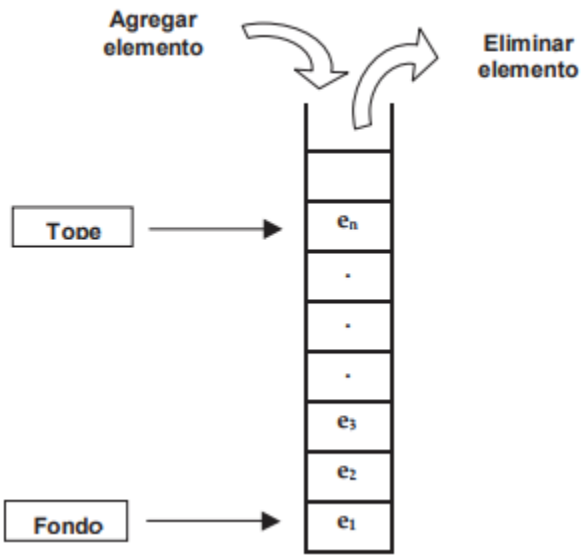
2. Especificación del tipo de datos abstracto.

3. Implementación. [1]

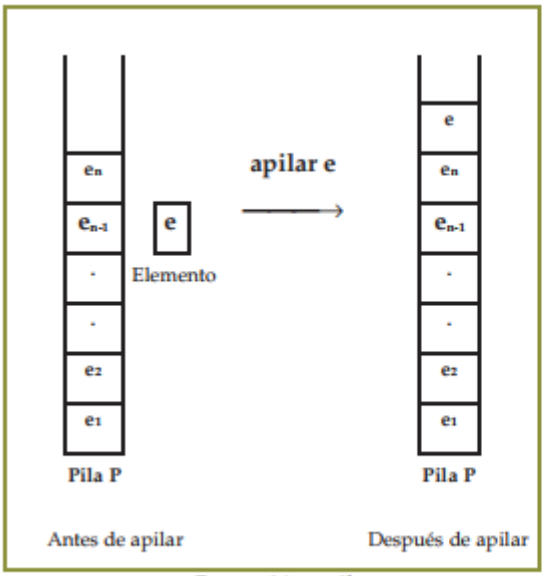
TAD pila.

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.

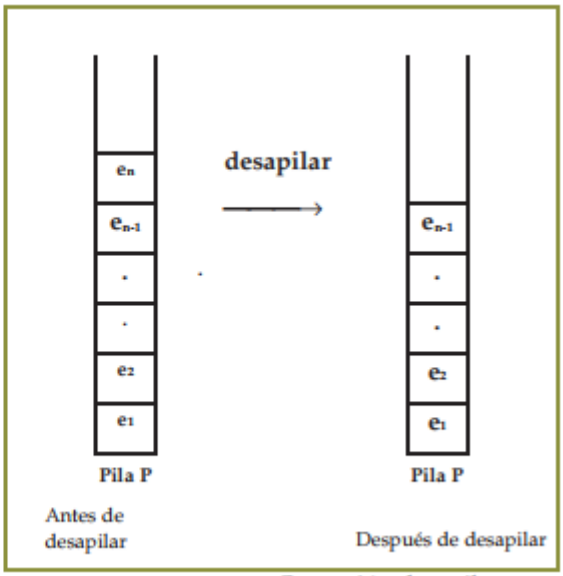
La representación gráfica de una pila es:



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, 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. [2]

Notación Prefija, Infija y Postfija.

Expresión aritmética: – Formada por operandos y operadores: A*B / (A+C)

– Operandos: variables que toman valores enteros o reales.

– Operadores:

Paréntesis	()	Nivel mayor de prioridad
Potencia	^	↓
Multiplicación/ División	* /	↓
Suma/Resta	+ -	Nivel menor de prioridad

– En caso de igualdad de prioridad.

Son evaluados de izquierda a derecha (se evalúa primero el que primero aparece)

⇒ 5*4/2 = (5*4)/2 = 10

Cuando aparecen varios operadores de potenciación juntos la expresión se evalúa de derecha a izquierda.

⇒ 2^3^2 = 2^ (3^2) = 2^9 = 512

Notación Infija – Es la notación ya vista que sitúa el operador entre sus operandos.

– Ventaja: Es la forma natural de escribir expresiones aritméticas.

– Inconveniente: Muchas veces necesita de paréntesis para indicar el orden de evaluación:

⇒ A*B/(A+C) ≠ A*B/A+C

Notación Prefija o Polaca – En 1920 un matemático de origen polaco, Jan Lukasiewicz, desarrollo un sistema para especificar expresiones matemáticas sin paréntesis. – Esta notación se conoce como notación prefija o polaca (en honor a la nacionalidad de Lukasiewicz) y consiste en situar al operador ANTES que los operandos.

– Ejemplo: la expresión infija **⇒ A*B / (A+C)**

Se representaría en notación prefija como: **⇒ /*AB+AC**

Notación Postfija o Polaca Inversa – La notación postfija o polaca inversa es una variación de la notación prefija de forma que el operador se pone DESPUÉS de los operandos.

– Ejemplo: la expresión infija **⇒ A*B / (A+C)**

Se representaría en notación postfija como: **⇒ AB*AC+/**

– Ventajas: La notación postfija (como la prefija) no necesita paréntesis.

La notación postfija es más utilizada por los computadores ya que permite una forma muy sencilla y eficiente de evaluar expresiones aritméticas (con pilas). [3]

Planteamiento del Problema.

Programar en ANSI C, el algoritmo para validar paréntesis en una expresión aritmética. Emplee el algoritmo que se apoya de una pila para ello.

Programar en ANSI C, el algoritmo para pasar una expresión aritmética en posfijo a partir de la expresión infija con la ayuda de una pila.

Programar en ANSI C, el algoritmo para evaluar una expresión aritmética posfija mediante la ayuda de una pila.

Crear el programa final que realiza todas las actividades anteriores de manera agradable para el usuario.

Diseño y funcionamiento de la solución.

Validad paréntesis.

Como se sabe, los paréntesis se utilizan para agrupar elementos, o para modificar la prioridad en una evaluación. Y como es bien sabido, cuando se tienen muchos paréntesis, se pueden omitir algunos que abren o cierran al momento de generar una cadena que los utilice (como al programar en LISP por ejemplo).

El evaluar los paréntesis en una expresión arimética es usado habitualmente como ejemplo, para explicar el uso de una estructura de datos simple, llamado pila.

La pila solo se le pueden agregar elementos nuevos por un solo lado, llamado el tope y para trabajar con ella, ésta tiene dos operaciones básicas, push para poner un nuevo elemento en el tope de la pila. y pop para extraer el último elemento del tope de la pila.

Aunque teóricamente se pueden apilar elementos hasta el infinito, dado que no existe un límite para el tope, en la práctica estamos limitados por el método de implementación y lugar donde se realiza. Particularmente, en una computadora, el límite de la pila está marcado por la cantidad de memoria que se puede utilizar para la pila. Por ello existen tres elementos más que describen a una pila: un indicador de principio de pila, un indicador de tope actual y un indicador del límite máximo de la pila.

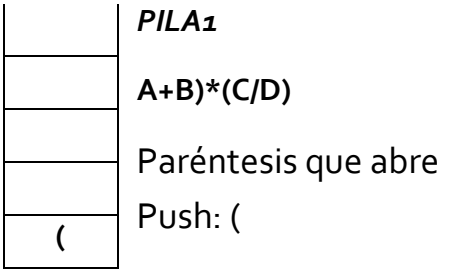
El indicador actual se incrementa si se hace un push, y se decrementa si se hace un pop. Y por supuesto, debe marcar un error cuando se esta en la base en donde el principio de pila es igual a indicador actual; lo que significa que la pila esta vaciá y se desea extraer un elemento (se marca underflow). De igual forma si el indicador actual es igual a límite máximo de la pila, que significa que estamos en el límite máximo, y se desea agregar un nuevo elemento se debe marcar un error (overflow).

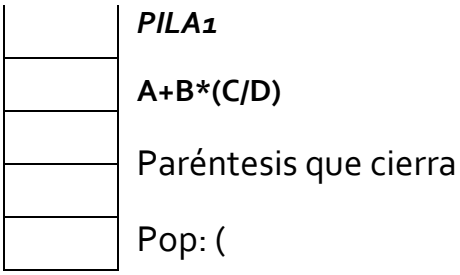
Para evaluar la concordancia de paréntesis en una expresión mediante una pila, lo que se hace es un push a la pila con un "(" cada vez que se encuentra en la expresión, y se hace un pop por cada ")" encontrado.

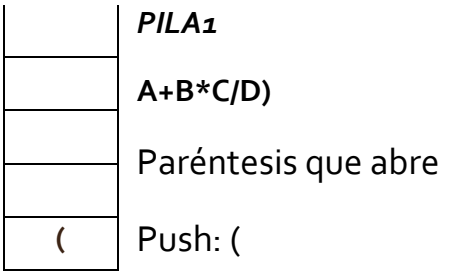
Después de recorrer toda la expresión que contiene paréntesis, si los paréntesis están correctos, la pila debe terminar vacía. En caso de que exista algún problema de concordancia de paréntesis y se notificara con una leyenda de “ERROR: Revisar sintaxis”

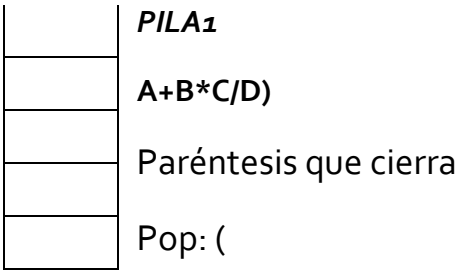
Ejemplo con paréntesis correctos:

(A+B)*(C/D)









¿La pila1 está *Vacía*? Si

Paréntesis Ok

Ejemplo con paréntesis incorrectos:

(((A+B)*(C/D)

	PILA1
	((A+B)*(C/D)
	Paréntesis que abre
(	Push: (

	PILA1
	(A+B)*(C/D)
	Paréntesis que abre
(	
(	Push: (

	PILA1
	A+B)*(C/D)
(	Paréntesis que abre
(	
(	Push: (

	PILA1
	A+B*C/D)
	Paréntesis que cierra
(	
(	Pop: (

	PILA1
	A+B*C/D)
(	Paréntesis que abre
(	
(	Push: (

	PILA1
	A+B*C/D
	Paréntesis que cierra
(	
(	Pop: (

¿La pila1 está *Vacía*? No
Paréntesis Incorrectos

Transformación de expresión infija a postfija

Se parte de una expresión en notación infija que tiene operandos, operadores y puede tener paréntesis. Los operandos vienen representados pos letras y los operadores son: **^ * / + -**

La transformación se realiza utilizando una pila en la cual se almacenan los operadores y los paréntesis izquierdos. La expresión aritmética se va leyendo desde el teclado de izquierda a derecha, carácter a carácter, los operandos pasan directamente a formar parte de la expresión en postfija la cual se guarda en un arreglo.

Los operadores se meten en la pila siempre que ésta esté vacía, o bien siempre que tengan mayor prioridad que el operador de la cima de la pila (o bien si es la máxima prioridad). Si la prioridad es menor o igual se saca el elemento cima de la pila y se vuelve a hacer la comparación con el nuevo elemento cima. Los paréntesis izquierdos siempre se meten en la pila; dentro de la pila se les considera de mínima prioridad para que todo operador que se encuentra dentro del paréntesis entre en la pila.

Cuando se lee un paréntesis derecho hay que sacar todos los operadores de la pila pasando a formar parte de la expresión postfija, hasta llegar a un paréntesis izquierdo, el cual se elimina ya

que los paréntesis no forman parte de la expresión postfija. El proceso termina cuando no hay más elementos de la expresión y la pila esté vacía.

Ejemplo: (A-B)^C+D

SalidaPostfijo

+
^
C
-
B
A

(

PILA1

A-B)^C+D

Paréntesis que abre (izquierdo)

Push: (

-
(

PILA1

AB)^C+D

Operador entra en la pila (hasta el momento es el de mayor precedencia)

Push: (

PILA1

AB^C+D

Paréntesis que cierra (derecho)

Vaciar pila.

^

PILA1

ABC+D

Operador entra en la pila (hasta el momento es el de mayor precedencia)

Push: ^

	PILA1
	ABC+D
	Sacar el ^ porque + es de menor precedencia
	Pop: ^

	PILA1
	ABCD
	Operador entra en la pila (hasta el momento es el de mayor precedencia)
+	Push: +

La expresión se terminó y se vacía la pila

	PILA1
	ABCD
	Pop: +

Tenemos la expresión en postfijo en ***SalidaPostfijo***

+
D
^
C
-
B
A

Imprimimos salidapostfijo de manera inversa

Evaluación de la expresión en notación postfija.

Se almacena la expresión aritmética transformada a notación postfija en un arreglo *arregloa1*, en la que los operandos están representados por variables de una sola letra. Antes de evaluar la expresión se requiere dar valores numéricos a los operandos. Una vez que se tiene los valores de los operandos, la expresión es evaluada. El algoritmo de evaluación utiliza una pila *pila1* de operandos, en definitiva de números reales.

Al describir el algoritmo el arreglo que contiene la expresión. El número de elementos de que consta la expresión es n.

Examinar el arreglo desde el elemento 1 hasta n. Si el elemento es un operando, asignar su valor antes obtenido y meterlo en la pila.

Si el elemento es un operador, lo designamos por ejemplo con +

Sacar los dos elementos superiores de la pila, los denominamos con los identificadores x, y respectivamente.

Sacarlos y colocar el tope de la pila *pila* de lado izquierdo del operador y el siguiente elemento de lado derecho del operador.

Evaluar x + y; el resultado es z = x + y.

El resultado z, meterlo en la pila.

Repetir el paso de recorrer el arreglo *arreglo1* pero ahora en la posición 2 hasta n.

El resultado de la evaluación de la expresión está en el elemento cima de la pila.

Fin del algoritmo.

Tomamos como ejemplo la salida postfijo que teníamos anteriormente ya de informa inversa como se indicó.

SalidaPostfijo que se muestra al usuario ***Pila Pila1***

Suponiendo tener los valores A=10 B=5 C=2 D=1 el valor esperado después de evaluar la expresión es 26.

Sacamos A y guardamos su valor en ***pila1.***

SalidaPostfijo que se muestra al usuario ***Pila Pila1***

Sacamos B y guardamos su valor en ***pila1.***

SalidaPostfijo que se muestra al usuario ***Pila Pila1***

El operador es – vaciamos ***pila1*** y ponemos el tope de la izquierdo y el siguiente elemento de lado derecho efectuamos la operación y nuevamente guardamos el resultado en ***pila1.***

SalidaPostfijo que se muestra al usuario ***Pila Pila1***

10 – 5 = 5

Sacamos C y guardamos su valor en ***pila1.***

SalidaPostfijo que se muestra al usuario ***Pila Pila1***

El operador es ^ vaciamos ***pila1*** y ponemos el tope de la izquierdo y el siguiente elemento de lado derecho efectuamos la operación y nuevamente guardamos el resultado en ***pila1.***

5 ^ 2 = 25

SalidaPostfijo que se muestra al usuario ***Pila Pila1***

Sacamos D y guardamos su valor en ***pila1.***

SalidaPostfijo que se muestra al usuario ***Pila Pila1***

El operador es + vaciamos ***pila1*** y ponemos el tope de la izquierdo y el siguiente elemento de lado derecho efectuamos la operación y nuevamente guardamos el resultado en ***pila1.***

25 + 1 = 26

SalidaPostfijo que se muestra al usuario ***Pila Pila1***

El resultado es el esperado, el cual se encuentra en ***pila1.***

Implementación de la solución.

TAD pila

Todo comienza con la implementación de un TAD pila respetando teoría explicada en la introducción sobre los tipos de datos abstractos.

Por tanto tenemos un TAD pila con sus operaciones, las cuales están listas para ser utilizadas dentro del programa principal.

```
#include "TADPilaEst.h"

void Initialize(pila *s){
	s -> tope = -1;
	return;
}

void Push(pila *s, elemento e){
	(*s).tope++;
	(*s).arreglo[(*s).tope] = e;	return;
}

elemento Pop(pila *s){
	elemento e;
	e = (*s).arreglo[(*s).tope];
	(*s).tope--;
	return e;
}

boolean Empty(pila *s){

	return (s -> tope == -1)?TRUE:FALSE;
}

elemento Top(pila *s){
	e = (*s).arreglo[(*s).tope];
	return e;
}

int Size(pila *s){
	int size;
	size = s -> tope +1;
	return size;
}

void Destroy(pila *s){
	Initialize(s);
	return;
}
```

Validación de paréntesis.

Se parte de un arreglo que guarda la expresión que el usuario ingreso desde el teclado, se invoca a una función llamada `validarParentesis` a la cual se le pasa esta cadena e inicializa una pila

pila1, la cadena se analiza con un contador que va incrementando, se crea una condicional, si el arreglo en la posición [i] es paréntesis que abre se realiza un push del paréntesis en ***pila1*** y si es un paréntesis que cierra se realiza un pop del paréntesis en ***pila1***

Al final la función `validarParentesis` devuelve un valor booleano y si la pila ***pila1*** está vacía se procede a decir que los paréntesis son correctos y si no está vacía se procede a decir que los paréntesis no son correctos

```
return (Empty(&p1) == TRUE)?TRUE:FALSE;
```

Convertir expresión de infija a postfija

Se crea una función de `precedencia` para poder decidir si el operador puede entrar a la pila pila1 o no.

```
int precedencia(char c){
      int resultado;
      switch(c){
            case '+': resultado = 1; break;
            case '-': resultado = 1; break;
            case '*': resultado = 2; break;
            case '/': resultado = 2; break;
            case '^': resultado = 3; break;
            case '(': resultado = 0; break;
      }
      return resultado;
}
```

Después, se crea una función llamada `pasarPostfijo` que recibe la cadena de la expresión que ingreso el usuario e inicializa una pila llamada ***pila1*** e inicializa un arreglo llamado `salidaPostfijo`

Ahora se analiza la cadena que usuario ingreso desde el teclado y se recorre posición por posición verificando si es un paréntesis que abre o un operador de mayor precedencia y entonces entran directamente en ***pila1*** y si es un operando entra directamente en `salidaPostfijo` si el operador es menor precedencia entonces se procede a sacar los elementos que hay en pila y guárdalos en `salidaPostfijo` hasta que el operador tendrá una precedencia mayor que los elementos que están en pila1 o hasta que ***pila1*** este vacía, si el siguiente elemento siguiente en la cadena (expresión) es un paréntesis que cierra se procede a vaciar la pila pasando todos los elementos a `salidaPostfijo` excluyendo los paréntesis que abren en caso de haber, ya que estos no pasan a formar parte del postfijo y este proceso de condiciones se repite hasta que la cadena (expresión) este vacía, es decir, que todos sus elementos ya fueron analizados.

```
void pasarPostfijo(char const *cadena){
	elemento e1;
	pila p1;
	int indice = 0;
	int i, j, tamano = 0, operadorAbajo, operadorArriba;
	char salidaPostfijo[MAX];
	Initialize(&p1);
	for(i = 0; i < strlen(cadena); i++){
		if(cadena[i] != '(' && cadena[i] != ')') tamano++;
	}
	for(i = 0; i < strlen(cadena); i++){
		if(cadena[i] >= 65 && cadena[i] <= 90){
	salidaPostfijo[indice] = cadena[i];
			printf("\n%i Operando detectado, agregado a arreglo
salidaPostfijo[%d]", i+1, indice);
			indice++;
		}
		if(cadena[i] == '('){
			e1.c = cadena[i];
			Push(&p1, e1);
			printf("\n%d Analizando expresion, %c detectado", i+1, e1.c);
		}
		if(cadena[i] == ')' && Empty(&p1) == FALSE){
			j = 0;
			printf("\n%d ')' Detectado, sacando elementos de la pila hasta
encontrar '('", i+1);
			do{
				e1 = Pop(&p1);
				if(esOperador(e1.c) == TRUE){
					printf("\n%d.%d Operador %c detectado y desempilado,
pasado salidaPostfijo[%d]", i + 1, j, e1.c, indice);
					salidaPostfijo[indice] = e1.c;
					indice ++;
				}
				j++;
			}while(e1.c != '(' && Empty(&p1) == FALSE);
		}
```

Nota: Aquí se muestra una parte de las condicionales explicadas anteriormente, no se agrega todo el código porque es muy extenso pero en anexos se puede ver el código completo.

Evaluación de la expresión en notación postfija.

Se crea una función llamada `solucionAlgebra` que recibe la cadena donde se almacenó la expresión ingresada por el usuario y define lo siguiente:

```
float solucionAlgebra(char const *cadena, int n){
```

```
      int i;
      elemento e1;
      pila p1;
      float val[27], valMarcado[27] =
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1};
```

val[27] está relacionado con que cada letra del alfabeto va a ocupar una posición en ese arreglo, por ejemplo, la 'a' ocuparía val[0] y la 'z' val[26] respectivamente, de modo que cada vez que se repita una letra simplemente se direccionará al valor que guarda esa letra en el vector de valores. Asimismo, valMarcado sirve para saber si una posición del arreglo val (del 0 al 26) ya cuenta con el valor que va a tener esa letra cada vez que aparezca, es decir, nos ayuda a saber si el valor de una variable (letra) ya fue preguntado y almacenado en el vector val. Cuando cierta letra ha aparecido por primera vez, por defecto el valor en valMarcado es 1, una vez que se le asigna un valor para cada vez que aparezca esa letra en la expresión, y que este valor se guarda en val[posición de la letra en el abecedario - 1] se modifica el valor de valMarcado[posición de la letra en el abecedario - 1] y pasa a ser 0, lo cual hara que cuando se lea el 0 se sepa que ya fue preguntado el valor de esa letra y ya está almacenado en val[] para su consulta. Para hacerlo más cómodo, se consideró lo siguiente: el índice val[indice] se determina: indice = cadena[i] - 'a', es decir, el código ASCII del carácter contenido en cadena[i] - el valor del código ASCII de a, de modo que el resultado vaya de 0 a 26 y no de 97 a 122 como sería originalmente. Por ejemplo, el código ASCII de 'a' es 97, por lo que la posición a la que estaría relacionada en val sería 0 por ser la primera letra del alfabeto, lo cual se obtiene de

a esta relacionada con val[cadena[i] - 'a'] si cadena[i] = 'a' -> a esta relacionada con val['a' - 'a'] = val[0] es decir, una sola vez se pide el valor de a si esta aparece en la expresión, y este valor se guarda en val[0], si a vuelve a aparecer más adelante en la expresión simplemente se consulta val[0]

```
float resultado, a, b;
      Initialize(&p1);
      for(i = 0; i < n; i++){
```

Y crea las condicionales para poder saber si es un operando y así pedir un valor para esa letra.

```
if(esOperador(cadena[i]) == FALSE && cadena[i] >= 65 && cadena[i] <= 90){
      if(valMarcado[cadena[i] - 'A'] == 1 ){
                        printf("\nIntroduzca un valor para '%c': ", cadena[i]);
                        scanf("%f",&val[cadena[i] - 'A']);
                        e1.n = val[cadena[i] - 'A'];
      Push(&p1,e1);
                        printf("\nOperando %f empilado correctamente", val[cadena[i]
- 'A']);
```

```
                        valMarcado[cadena[i] - 'A'] = 0;
                }
                else{
                        if(valMarcado[cadena[i] - 'A'] == 0 ){
                                e1.n = val[cadena[i] - 'A'];
                                Push(&p1, e1);
                                printf("\nOperando %f empilado correctamente",
val[cadena[i] - 'A']);
                        }

                }
        }
```

Ahora, para cada operación se creó una función llamas `Suma` `Resta` `Multiplicacion` `Division` `Potencia` las cuales son llamadas por cada uno de los casos que marca el siguiente switch:

```
if(esOperador(cadena[i]) == TRUE && Empty(&p1) == FALSE){
      switch(cadena[i]){
                        case '+':
                                e1 = Pop(&p1);
                                b = e1.n;
                                e1 = Pop(&p1);
                                a = e1.n;
                                e1.n = Suma(a,b);
                                printf("\n Efectuando suma de %f + %f y empilando", a,
b);
                                break;
                        case '-':
                                e1 = Pop(&p1);
                                b = e1.n;
                                e1 = Pop(&p1);
                                a = e1.n;
                                e1.n = Resta(a,b);
                                printf("\n Efectuando resta: %f - %f y empilando", a,
b);
                                break;
```

Las funciones `Suma` `Resta` son las siguientes:

```
float Suma(float a, float b){
      float resultado;
      resultado = a + b;
      return resultado;
}

float Resta(float a, float b){
```

```
    float resultado;
    resultado = a - b;
    return resultado;
}
```

Nota: Aquí se muestra una parte de las funciones mencionadas anteriormente (`Suma Resta`), no se agrega todo el código porque es muy extenso pero en anexos se puede ver el código completo.

Finalmente con ayuda de una función llamada `otroProceso` se le pregunta al usuario si desea evaluar otra expresión.

```
boolean otroProceso(){
    char sn, respuesta[20];
    printf("\n\n\t Desea introducir otra expresion (s|S, n|N)?: ");
    setbuf(stdin, NULL);
    scanf("%c",&sn);
    printf("\n Usted selecciono %c \n", sn);
    return (sn == 's' || sn == 'S')?TRUE:FALSE;
```

Funcionamiento

Pruebao1:

Expresión: (A+B)*(C/D)

Valores: A =5; B=10; C=4; D=2

```
                IPN
                ESCOM
Programa que valida y resuelve expresiones algebraicas por medio del TAD pila
Introduzca una expresion algebraica considerando:
 + = Suma                - = Resta
 / = Division            * = Multiplicacion
 ^ = Potencia
Puede introducir parentesis, los cuales seran validados
Como ejemplo, puede introducir una expresion como (A+B)^A-C
NOTA: procure no dejar espacios

 A continuacion introduzca su expresion: (A+B)*(C/D)
```

```
 Su expresion (A+B)*(C/D) fue guardada correctamente.
Se procedera a validar parentesis

1 Parentesis '(' introducido a la pila
5 Se detecto ')' por lo que se saco un '(' de la pila
7 Parentesis '(' introducido a la pila
11 Se detecto ')' por lo que se saco un '(' de la pila
Parentesis correctos, se procedera a realizar conversion a postfijo
```

```
1 Analizando expresion, ( detectado
2 Operando detectado, agregado a arreglo salidaPostfijo[0]
3 Analizando expresion, + detectado y empilado
4 Operando detectado, agregado a arreglo salidaPostfijo[1]
5 ')' Detectado, sacando elementos de la pila hasta encontrar '('
5.0 Operador + detectado y desempilado, pasado salidaPostfijo[2]
6 Analizando expresion, * detectado y empilado
7 Analizando expresion, ( detectado
8 Operando detectado, agregado a arreglo salidaPostfijo[3]
9 Analizando expresion, / detectado y empilado
10 Operando detectado, agregado a arreglo salidaPostfijo[4]
11 ')' Detectado, sacando elementos de la pila hasta encontrar '('
11.0 Operador / detectado y desempilado, pasado salidaPostfijo[5]
El tamano de la pila de operadores es: 1
12.0 Se saco el operador * de la pila y se agrego a salidaPostfijo[6]
La pila esta vacia, a continuacion se imprimira la expresion en Postfijo:
Expresion en postfijo = AB+CD/*
```

```
Introduzca un valor para 'A': 5

Operando 5.000000 empilado correctamente
Introduzca un valor para 'B': 10

Operando 10.000000 empilado correctamente
 Efectuando suma de 5.000000 + 10.000000 y empilando
Introduzca un valor para 'C': 4

Operando 4.000000 empilado correctamente
Introduzca un valor para 'D': 2

Operando 2.000000 empilado correctamente
 Efectuando division: 4.000000 / 2.000000 y empilando
 Efectuando multiplicacion: 15.000000 * 2.000000 y empilando



         El resultado de la expresion es: 30.000000

         Desea introducir otra expresion (s|S, n|N)?: S
```

Pruebao2:

Expresión: (A*(B*C))^(D-A)
Valores: A =5; B=1; C=4; D=7

```
                IPN
                ESCOM
Programa que valida y resuelve expresiones algebraicas por medio del TAD pila
Introduzca una expresion algebraica considerando:
 + = Suma                  - = Resta
 / = Division              * = Multiplicacion
 ^ = Potencia
Puede introducir parentesis, los cuales seran validados
Como ejemplo, puede introducir una expresion como (A+B)^A-C
NOTA: procure no dejar espacios

 A continuacion introduzca su expresion: (A*(B*C))^(D-A)
```

```
 Su expresion (A*(B*C))^(D-A) fue guardada correctamente.
Se procedera a validar parentesis

1 Parentesis '(' introducido a la pila
4 Parentesis '(' introducido a la pila
8 Se detecto ')' por lo que se saco un '(' de la pila
9 Se detecto ')' por lo que se saco un '(' de la pila
11 Parentesis '(' introducido a la pila
15 Se detecto ')' por lo que se saco un '(' de la pila
Parentesis correctos, se procedera a realizar conversion a postfijo
```

```
1 Analizando expresion, ( detectado
2 Operando detectado, agregado a arreglo salidaPostfijo[0]
3 Analizando expresion, * detectado y empilado
4 Analizando expresion, ( detectado
5 Operando detectado, agregado a arreglo salidaPostfijo[1]
6 Analizando expresion, * detectado y empilado
7 Operando detectado, agregado a arreglo salidaPostfijo[2]
8 ')' Detectado, sacando elementos de la pila hasta encontrar '('
8.0 Operador * detectado y desempilado, pasado salidaPostfijo[3]
9 ')' Detectado, sacando elementos de la pila hasta encontrar '('
9.0 Operador * detectado y desempilado, pasado salidaPostfijo[4]
10 Analizando expresion, ^ detectado y empilado
11 Analizando expresion, ( detectado
12 Operando detectado, agregado a arreglo salidaPostfijo[5]
13 Analizando expresion, - detectado y empilado
14 Operando detectado, agregado a arreglo salidaPostfijo[6]
15 ')' Detectado, sacando elementos de la pila hasta encontrar '('
15.0 Operador - detectado y desempilado, pasado salidaPostfijo[7]
El tamano de la pila de operadores es: 1
16.0 Se saco el operador ^ de la pila y se agrego a salidaPostfijo[8]
La pila esta vacia, a continuacion se imprimira la expresion en Postfijo:
Expresion en postfijo = ABC**DA-^
```

```
Introduzca un valor para 'A': 5

Operando 5.000000 empilado correctamente
Introduzca un valor para 'B': 1

Operando 1.000000 empilado correctamente
Introduzca un valor para 'C': 4

Operando 4.000000 empilado correctamente
 Efectuando multiplicacion: 1.000000 * 4.000000 y empilando
 Efectuando multiplicacion: 5.000000 * 4.000000 y empilando
Introduzca un valor para 'D': 7

Operando 7.000000 empilado correctamente
Operando 5.000000 empilado correctamente
 Efectuando resta: 7.000000 - 5.000000 y empilando
 Efectuando potencia: 20.000000 ^ 2.000000 y empilando



         El resultado de la expresion es: 400.000000

         Desea introducir otra expresion (s|S, n|N)?:
```

Pruebao3:

Expresión: (A((B*C)

```
                IPN
                ESCOM
Programa que valida y resuelve expresiones algebraicas por medio del TAD pila
Introduzca una expresion algebraica considerando:
 + = Suma                - = Resta
 / = Division            * = Multiplicacion
 ^ = Potencia
Puede introducir parentesis, los cuales seran validados
Como ejemplo, puede introducir una expresion como (A+B)^A-C
NOTA: procure no dejar espacios

 A continuacion introduzca su expresion: (A((B*C)

 Su expresion (A((B*C) fue guardada correctamente.
Se procedera a validar parentesis

1 Parentesis '(' introducido a la pila
3 Parentesis '(' introducido a la pila
4 Parentesis '(' introducido a la pila
8 Se detecto ')' por lo que se saco un '(' de la pila
 Parentesis incorrectos, revise la sintaxis
```

Errores detectados

Al terminar el programa y probarlo con la pila en la implementación estática funciono correctamente, el único problema que surgió fue al probarlo con la pila en la implementación dinámica el programa dejaba de funcionar, al revisar el código, nos dimos cuenta que en el momento de analizar la cadena para poder convertir a postfijo no existía ninguna validad comprobar que la pila no estuviera vacía antes de realizar un pop, por lo cual se producía un subdesbordamiento de pila. El problema fue detectado y corregido y finalmente el programa funciono correctamente con la implementación estática y con la implementación dinámica.

Después de esta modificación el programa funcionaba de la forma correcta ya que daba los valores esperados pero no podemos asegurar que con todas las expresiones funcione de forma correcta ya que existen expresiones más "complejas" lo cual podría hacer que el programa realice de forma errónea alguna operación pero hasta el momento no se ha encontrado ningún error de este tipo.

Posibles Mejoras

Hasta el momento no hemos encontrado una manera más eficiente en nuestra implementación, seguiremos buscando alternativas. La posible mejora quizá mas evidente seria que en las funciones que definimos como suma, resta, multiplicación, división y potencia para realizar las operaciones podríamos ahorrarnos la memoria de una variable y solo retornar el valor requerido, por ejemplo

return (a+b); en lugar de **c= (a+b);**

***return c*;**

Anexos

Aquí se muestran los códigos completos con la documentación requerida.

Implementacion Estatica

Nota: La implementación contiene los mismos códigos, lo que distingue a una de la otra es que en el código principal se incluye la librería TADPilaDin.h

TADpilaEst.c

```
/*****************************************************************
* IMPLEMENTACIÓN DE LA LIBRERÍA TAD PILA ESTÁTICA                *
* AUTORES:                                                       *
* - Alanís Ramírez Damián                                        *
* - Mendieta Torres Alfonso Ulises                               *
* - Oledo Enriquez Gilberto Irving                               *
*                                                                *
* DESCRIPCIÓN: TAD pila o stack.                                 *
* Estructura de datos en la cual se cumple:                      *
* Los elementos se añaden y remueven por un solo extremo (siguiendo el  *
* criterio LIFO - Last In First Out).                            *
* Este extremo se llama "tope" de la pila.                       *
*                                                                *
* OBSERVACIONES: Hablamos de una estructura de datos estática cuando se  *
* le asigna una cantidad fija de memoria antes de la ejecución del pro-  *
* grama.                                                         *
*                                                                *
* COMPILACIÓN PARA GENERAR EL CÓDIGO OBJETO: gcc -c TADPilaEst.c *

*****************************************************************/


//LIBRERIAS
#include "TADPilaEst.h"

//DEFINICIÓN DE FUNCIONES

/*
void Initialize(pila *s);
Descripción: Inicializar pila (Iniciar una pila para su uso)
Recibe: pila *s (Referencia a la pila "s" a operar)
Devuelve:
Observaciones: El usuario ha creado una pila y s tiene la referencia a ella,
si esto no ha pasado se producirá un error.
*/
void Initialize(pila *s){
     s -> tope = -1;       //(*s).tope = -1
     return;
```

```
}

/*
void Push(pila *s, elemento e);
Descripción: Esta función empila un elemento (lo introduce a la pila)
Recibe: pila *s (Referencia a la pila "s" a operar), elemento e (Elemento a
introducir en la pila)
Devuelve:
Observaciones: El usuario ha creado una pila y s tiene la referencia a ella, s ya
ha sido inicializada.
Ademas no se valida el indice del arreglo (tope) si esta fuera del arreglo es
decir hay desbordamiento
y se producirá un error.
*/
void Push(pila *s, elemento e){
	(*s).tope++;
	(*s).arreglo[(*s).tope] = e; //(*s).tope es el índice, es el tope, en el
cual se introduce el elemento
	return;
}

/*
elemento Pop(pila *s);
Descripción: Desempila (extrae un elemento de la pila)
Recibe: pila *s (Referencia a la pila "s" a operar)
Devuelve: elemento (Elemento e extraido de la pila)
Observaciones: El usuario ha creado una pila y s tiene la referencia a ella, s ya
ha sido inicializada.
Ademas no se valida si la pila esta vacia (tope == -1) antes de desempilar (causa
error desempilar si la pila es vacía)
*/
elemento Pop(pila *s){
	elemento e;
	e = (*s).arreglo[(*s).tope];	//guarda en e el valor contenido en
arreglo[tope] de la pila s
	(*s).tope--;				//reduce en 1 el tope debido a que ya
se extrajo un elemento
	return e;
}

/*
boolean Empty(pila *s);
Descripción: ¿Es vacia? (Preguntar si la pila esta vacia)
Recibe: pila *s (Referencia a la pila "s" a operar)
Devuelve: boolean (TRUE o FALSE según sea el caso)
Observaciones: El usuario ha creado una pila y s tiene la referencia a ella, s ya
ha sido inicializada.
*/
boolean Empty(pila *s){
```

```
	/*
	boolean b;
	if(s-> tope == -1) b = TRUE;
	else b = FALSE;
	return b;
	*/
	return (s -> tope == -1)?TRUE:FALSE;//esta expresión es equivalente a todo
el bloque anterior
}

/*
elemento Top(pila *s);
Descripción: Tope (Obtener el "elemento" del tope de la pila si extraerlo de la
pila)
Recibe: pila *s (Referencia a la pila "s" a operar)
Devuelve: elemento (Elemento del tope de la pila)
Observaciones: El usuario ha creado una pila y s tiene la referencia a ella, s ya
ha sido inicializada.
Ademas no se valida si la pila esta vacia antes de consultar al elemento del tope
(causa error si esta esta vacía).
*/
elemento Top(pila *s){                          //se diferencia de Pop en que
este no lo saca de la pila, solo lo muestra
	elemento e;
	e = (*s).arreglo[(*s).tope];
	return e;
}

/*
elemento Size(pila *s);
Descripción: Función que devuelve el tamaño de la pila (Obtener el número de
elementos en la pila)
Recibe: pila *s (Referencia a la pila "s" a operar)
Devuelve: int (Tamaño de la pila -1->Vacia, 1->1 elemento, 2->2 elementos, ...)
Observaciones: El usuario ha creado una pila y s tiene la referencia a ella, s ya
ha sido inicializada.
*/
int Size(pila *s){
	int size;
	size = s -> tope +1;                    //debido a que tope inicialmente es -1
	return size;
}

/*
void Destroy(pila *s);
Descripción: Elimina pila (Borra a todos los elementos en a la pila de memoria)
Recibe: pila *s (Referencia a la pila "s" a operar)
Devuelve:
Observaciones: El usuario ha creado una pila y s tiene la referencia a ella.
```

```
*/
void Destroy(pila *s){
      Initialize(s);
      return;
}
```

TADPilaEst.h

```
/*****************************************************************
* LIBRERIA: TAD Pila Estática                                    *
* AUTORES:                                                       *
* - Alanís Ramírez Damián                                        *
* - Mendieta Torres Alfonso Ulises                               *
* - Oledo Enriquez Gilberto Irving                               *
*                                                                *
* DESCRIPCIÓN: TAD pila o stack.                                 *
* Estructura de datos en la cual se cumple:                      *
* Los elementos se añaden y remueven por un solo extremo (siguiendo el  *
* criterio LIFO - Last In First Out).                            *
* Este extremo se llama "tope" de la pila.                       *
*                                                                *
* OBSERVACIONES: Hablamos de una estructura de datos estática cuando se  *
* le asigna una cantidad fija de memoria antes de la ejecución del pro-  *
* grama.                                                         *

*****************************************************************/

//DEFINICIONES DE CONSTANTES
#define MAX_ELEMENT 1000
#define TRUE    1
#define FALSE   0

//DEFINICIONES DE TIPOS DE DATOS

//Definición de un boolean con un char
typedef unsigned char boolean;
//Definición de un elemento con un estructura Elemento
typedef struct elemento{              //Define una estructura Elemento
     //Variables de la estructura elemento, el usuario puede crear y/o
modificarlas
     char c;
     float n;                              //Permitirá introducir caracteres como
'(' o '+'
     //***
     //***
     //***
}elemento;                                 //Define un tipo de dato llamado
elemento
//Definición de una pila por medio de una estructura Pila que contiene puntero a
Elemento
typedef struct pila{
     elemento arreglo[MAX_ELEMENT];   //La pila es un arreglo estático de
MAX_ELEMENT
     int tope;                             //El tope es un entero (índice del
arreglo de elementos)
```

```
}pila;

//DECLARACIÓN DE FUNCIONES

/* Nota:
   La variable tipo pila se pasa por referencia debido a que si no se hace así se perdería todo cambio
   sobre esa variable pues se pasa por valor, y es una variable local de la función. Pasando por refe-
   rencia se logra que la variable pila conserve sus modificaciones en cualquier otra función o segmen-
   to del programa.

   Siempre que se trabaje con estructuras de datos se pasan los tipos de datos por REFERENCIA.
*/

void Initialize(pila *s);              //Inicializa la pila para su uso
void Push(pila *s, elemento e);        //Empila (introduce un elemento en la cima de la pila)
elemento Pop(pila *s);                      //Desempila (saca el elemento de la pila)
boolean Empty(pila *s);                     //Vacia (pregunta si la pila esta vacia y retorna TRUE o FALSE)
elemento Top(pila *s);                      //Devuelve el elemento que se encuentra en el tope sin extraerlo
int Size(pila *s);                          //Tamaño de la pila, obtiene el número de elementos de la pila
void Destroy(pila *s);                      //Elimina pila (borra todos los elementos y a la pila de memoria)
```

validarAlgebraPila.c

```c
/*************************************************************************
* PROGRAMA: validarAlgebraPila.c                                         *
* AUTORES:                                                               *
* - Alanís Ramírez Damián                                                *
* - Mendieta Torres Alfonso Ulises                                       *
* - Oledo Enriquez Gilberto Irving                                       *
* VERSIÓN: 1.8                                                           *
*                                                                        *
* DESCRIPCIÓN: Programa que pide una cadena de caracteres (expresión al-*
* gebraica) y procede a analizar los paréntesis, realizar la conversión *
* a postfijo y da la correspondiente solución. Esto por medio del uso de*
* la libreria TADPila(Est|Din).h creada en clase.                        *
*                                                                        *
* Compilación: cd (ruta_archivos)                                        *
* gcc -o validarAlgebraPila validarAlgebraPila.c TADPila(Est|Din).c      *
*************************************************************************/

#include <stdio.h>
#include <math.h>
#include <string.h>
#include <ctype.h>
#include "TADPilaEst.h"

//DEFINICIONES
#define MAX 100

//DECLARACIÓN DE FUNCIONES
boolean otroProceso();
boolean validarParentesis(char const *cadena);
int precedencia(char c);
boolean esOperador(char c);
void pasarPostfijo(char const *cadena);
float solucionAlgebra(char const *cadena, int n);
float Multiplicacion(float a, float b);
float Suma(float a, float b);
float Resta(float a, float b);
float Division(float a, float b);
float Potencia(float a, float b);

//MAIN
int main(){
      //DECLARACIÓN DE VARIABLES DEL MAIN
      elemento e1;
      pila p1;
      char expresion[MAX]; //Arreglo que contendrá a la expresión algebráica
      int i;                        //Variable para un contador
      while(1){
```

```
            Initialize(&p1);//Inicializa la pila p1
            //Impresión de las instrucciones y precondiciones del programa
            printf("Programa que valida y resuelve expresiones algebraicas por
medio del TAD pila");
            printf("\nIntroduzca una expresion algebraica considerando: ");
            printf("\n + = Suma \t\t - = Resta \n / = Division \t\t * =
Multiplicacion \n ^ = Potencia\n");
            printf("Puede introducir parentesis, los cuales seran validados\n");
            printf("Como ejemplo, puede introducir una expresion como (a+b)^a-
c\n");
            printf("NOTA: procure no dejar espacios");
            printf("\n\n A continuacion introduzca su expresion: ");
            scanf("%s",expresion);
            if(validarParentesis(expresion) == TRUE){ //Llama a la función
validarParentesis, si son correctos continua
                  printf("\nParentesis correctos, se procedera a realizar
conversion a postfijo\n");//la conversión
                  Destroy(&p1); //Destruye la pila p1 para ser empleada nuevamente
en la función pasarPostfijo(expresion)
                  pasarPostfijo(expresion); //Manda a llamar a pasarPostfijo,
pasándole expresion como argumento tipo char const *cadena
            }
            else{ //Si los parentesis no son correctos imprime una alerta y lleva
a la parte del ciclo en la que se llama a otroProceso()
                  printf("\n Parentesis incorrectos, revise la sintaxis \n");
            }
            if(otroProceso() == TRUE){ //Si la función devuelve TRUE se destruye
la pila y se limpia el buffer
                  Destroy(&p1);
                  getchar();
                  setbuf(stdin, NULL);
            }
            else break; //Si no se desea otro proceso (la función devuelve FALSE)
se sale del while(1) y main retorna 0
      }
      getchar();
      return 0;
}

//DEFINICIÓN DE FUNCIONES

/*
 boolean otroProceso()
 Descripción: imprime letrero y solicita una respuesta al usuario sobre si
 quiere introducir otra expresión.
 Recibe: -
 Devuelve: TRUE si el usuario teclea 's' ó 'S' y FALSE si teclea 'n' o 'N'
 Observaciones: el usuario ha introducido unicamente letras del siguiente
conjunto
```

```
 {'s','S','n','N'}
*/
boolean otroProceso(){
      char sn, respuesta[20];
      printf("\n\n\t Desea introducir otra expresion (s|S, n|N)?: ");
      setbuf(stdin, NULL);
      scanf("%c",&sn);
      printf("\n Usted selecciono %c \n", sn);
      return (sn == 's' || sn == 'S')?TRUE:FALSE;
}

/*
boolean validarParentesis(char const *cadena)
Descripción: valida los paréntesis de una expresión algebráica
Recibe: una cadena de caracteres (char const *cadena)
Devuelve: TRUE si los paréntesis son correctos o FALSE si son incorrectos
Observaciones: el usuario ha introducido la expresión y esta se ha guardado en
una cadena que
se pasa por referencia
*/
boolean validarParentesis(char const *cadena){
      int i;
      elemento e1;
      pila p1;
      Initialize(&p1);
      printf("\n Su expresion %s fue guardada correctamente. \nSe procedera a
validar parentesis\n", cadena);
      for(i = 0; i < strlen(cadena); i++){
            if(cadena[i] == '('){
                  e1.c = cadena[i];
                  Push(&p1, e1);
                  printf("\n%i Parentesis '(' introducido a la pila", i+1);
            }
            if(cadena[i] == ')'){
                  if(Empty(&p1) == FALSE){
                        Pop(&p1);
                        printf("\n%i Se detecto ')' por lo que se saco un '(' de la
pila", i+1);
                  }
                  else{
                        printf("\n\t%i ERROR, hay mas parentesis que cierran que los
que abren \n", i+1);
                        return FALSE;
                  }
            }
      }
      return (Empty(&p1) == TRUE)?TRUE:FALSE;
}
```

```
/*
 int precedencia(char c)
 Descripción: analiza el número que le correspondería a un operador en la
jerarquía
 de operadores.
 Recibe: un caracter (char c)
 Devuelve: un entero del 1 al 3 según sea la jerarquía del operador para la
conversión
 a postfijo
*/
int precedencia(char c){
      int resultado;
      switch(c){
            case '+': resultado = 1; break;
            case '-': resultado = 1; break;
            case '*': resultado = 2; break;
            case '/': resultado = 2; break;
            case '^': resultado = 3; break;
            case '(': resultado = 0; break;
      }
      return resultado;
}

/*
 boolean esOperador(char c)
 Descripción: determina si un caracter es o no un operando.
 Recibe: un caracter (char c)
 Devuelve: TRUE si es un operador o FALSE si no lo es
*/
boolean esOperador(char c){
      return (c == '+' || c == '-' || c == '*' || c == '/' || c ==
'^')?TRUE:FALSE;
}

/*
 float Suma(float a, float b)
 Descripción: suma dos flotantes y devuelve el resultado
 Recibe: dos números flotantes (float a, float b)
 Devuelve: el resultado de sumar a y b, como dato de tipo flotante
*/
float Suma(float a, float b){
      float resultado;
      resultado = a + b;
      return resultado;
}

/*
 float Resta(float a, float b)
 Descripción: resta un flotante a otro y devuelve el resultado
```

```
 Recibe: dos números flotantes (float a, float b)
 Devuelve: el resultado de la resta a - b, como dato de tipo flotante
*/
float Resta(float a, float b){
      float resultado;
      resultado = a - b;
      return resultado;
}

/*
 float Multiplicacion(float a, float b)
 Descripción: multiplica dos flotantes y devuelve el resultado
 Recibe: dos números flotantes (float a, float b)
 Devuelve: el resultado de multiplicar a y b, como dato de tipo flotante
*/
float Multiplicacion(float a, float b){
      float resultado;
      resultado = a * b;
      return resultado;
}

/*
 float Division(float a, float b)
 Descripción: divide un flotante entre otro
 Recibe: dos números flotantes (float a, float b)
 Devuelve: el resultado de la división a/b, como dato de tipo flotante
*/
float Division(float a, float b){
      float resultado;
      resultado = a / b;
      return resultado;
}

/*
 float Potencia(float a, float b)
 Descripción: eleva un flotante a un exponente flotante
 Recibe: dos números flotantes (float a, float b)
 Devuelve: el resultado de la operación a^b, como dato de tipo flotante
*/
float Potencia(float a, float b){
      double resultado;
      resultado = pow(a,b); //Función pow(a,b) eleva a a la b y requiere doubles
para operar
      return (float)resultado; //Se realiza el casting a flotante para su
incorporación a la pila
}

/*
 float solucionAlgebra(char const *cadena, int n)
```

```
 Descripción: función que obtiene el resultado de la expresión algebráica
 mediante el uso de la conversión a postfijo (que se le pasa como argumento)
 y que va pidiendo valores de las variables con el fin de dar la solución
 final de forma numérica
 Recibe: una cadena (salidaPostfijo) y un entero n que es el tamaño de esa cadena
salidaPostfijo
 sin contar la basura que habia quedado almacenada en las posiciones libres del
arreglo.
 Devuelve: el resultado de la expresión algebráica como tipo flotante
*/
float solucionAlgebra(char const *cadena, int n){
      int i;
      elemento e1;
      pila p1;
      float val[27], valMarcado[27] =
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1};
      /*
            val[27] esta relacionado con que cada letra del alfabeto va a ocupar
una posición en ese arreglo,
            por ejemplo, la 'a' ocuparía val[0] y la 'z' val[26] respectivamente,
de modo que cada vez que se
            repita una letra simplemente se direccionará al valor que guarda esa
letra en el vector de valores.
            Asimismo, valMarcado sirve para saber si una posición del arreglo val
(del 0 al 26) ya cuenta con
            el valor que va a tener esa letra cada vez que aparezca, es decir, nos
ayuda a saber si el valor
            de una variable (letra) ya fue preguntado y almacenado en el vector
val.
            Cuando cierta letra ha aparecido por primera vez, por defecto el valor
en valMarcado es 1, una vez
            que se le asigna un valor para cada vez que aparezca esa letra en la
expresión, y que este valor se
            guarda en val[posición de la letra en el abecedario - 1] se modifica
el valor de valMarcado[posición
            de la letra en el abecedario - 1] y pasa a ser 0, lo cual hara que
cuando se lea el 0 se sepa que ya
            fue preguntado el valor de esa letra y ya esta almacenado en val[]
para su consulta.
            Para hacerlo más cómodo, se consideró lo siguiente:
            el índice val[indice] se determina:
                  indice = cadena[i] - 'a', es decir, el código ASCII del caracter
contenido en cadena[i] - el valor
                  del código ASCII de a, de modo que el resultado vaya de 0 a 26 y
no de 97 a 122 como sería original-
                  mente.
                  Por ejemplo, el código ASCII de 'a' es 97, por lo que la posición
a la que estaría relacionada en val
```

```
                sería 0 por ser la primera letra del alfabeto, lo cual se obtiene
de

                a esta relacionada con val[cadena[i] - 'a']
                si cadena[i] = 'a' -> a esta relacionada con val['a' - 'a'] =
val[0]
                es decir, una sola vez se pide el valor de a sie sta aparece en
la expresión, y este valor se guarda en
                val[0], si a vuelve a aparecer más adelante en la expresión
simplemente se consulta val[0]
      */
      float resultado, a, b;
      Initialize(&p1);
      for(i = 0; i < n; i++){
            if(esOperador(cadena[i]) == FALSE && cadena[i] >= 97 && cadena[i] <=
122){ //Este procedimiento solo se aplica a las letras y no a los operadores
                if(valMarcado[cadena[i] - 'a'] == 1 ){ //Para determinar si el
valor ya ha sido indicado
                      printf("\nIntroduzca un valor para '%c': ", cadena[i]); //Se
pide por única vez el valor númerico de esa variable (letra)
                      scanf("%f",&val[cadena[i] - 'a']);
                      e1.n = val[cadena[i] - 'a']; //Se copia el valor a un tipo
de dato elemento para poder pasarlo a la Pila
                      Push(&p1,e1);   //Se apila el operando
                      printf("\nOperando %f empilado correctamente", val[cadena[i]
- 'a']);
                      valMarcado[cadena[i] - 'a'] = 0; //Se modifica el valor de
esa posición del arreglo para saber que ya se visitó
                }
                else{ //Ya no pide el valor, simplemente lo empila porque ya se
había dado
                      if(valMarcado[cadena[i] - 'a'] == 0 ){
                            e1.n = val[cadena[i] - 'a'];
                            Push(&p1, e1);
                            printf("\nOperando %f empilado correctamente",
val[cadena[i] - 'a']);
                      }

                }
            }
            if(esOperador(cadena[i]) == TRUE && Empty(&p1) == FALSE){ //Si lo que
encuentra en el recorrido de la cadena es un operador
                switch(cadena[i]){
                      case '+': //Si encuentra un más
                            e1 = Pop(&p1); //Hace un Pop
                            b = e1.n; // Guarda ese valor en una variable b
                            e1 = Pop(&p1);
                            a = e1.n;
```

```
                        e1.n = Suma(a,b); //Llama a la función pasándole a y b
como argumentos
                        printf("\n Efectuando suma de %f + %f y empilando", a,
b);
                        break;
                  case '-':
                        e1 = Pop(&p1);
                        b = e1.n;
                        e1 = Pop(&p1);
                        a = e1.n;
                        e1.n = Resta(a,b);
                        printf("\n Efectuando resta: %f - %f y empilando", a,
b);
                        break;
                  case '*':
                        e1 = Pop(&p1);
                        b = e1.n;
                        e1 = Pop(&p1);
                        a = e1.n;
                        e1.n = Multiplicacion(a,b);
                        printf("\n Efectuando multiplicacion: %f * %f y
empilando", a, b);
                        break;
                  case '/':
                        e1 = Pop(&p1);
                        b = e1.n;
                        e1 = Pop(&p1);
                        a = e1.n;
                        e1.n = Division(a,b);
                        printf("\n Efectuando division: %f / %f y empilando",
a, b);
                        break;
                  case '^':
                        e1 = Pop(&p1);
                        b = e1.n;
                        e1 = Pop(&p1);
                        a = e1.n;
                        e1.n = Potencia(a,b);
                        printf("\n Efectuando potencia: %f ^ %f y empilando",
a, b);
                        break;
            }
            Push(&p1, e1); //Apila el resultado de la operación que haya
realizado
         }
     } // Va haciendo ese ciclo y realiza todas las operaciones hasta que recorre
toda la cadena,
     e1 = Pop(&p1); //dejando un solo elemento en la pila, el cual es el
resultado, así que se le hace Pop
```

```
	resultado = e1.n;
	return resultado; //Devuelve el resultado como tipo flotante de la expresión algebráica
}

/*
 void pasarPostfijo(char const *cadena)
 Descripción: función que realiza la conversión de la expresión en infijo a postfijo (la expresión
 en infijo se le pasa como argumento) y muestra la conversión en pantalla. Finalmente llama a
 solucionAlgebra(...) y una vez que esta finaliza devuelve el control al main.
 Recibe: una cadena (expresión en infijo).
 Devuelve:
*/
void pasarPostfijo(char const *cadena){
	elemento e1;
	pila p1;
	int indice = 0; //Declara la variable indice para ser empleada en salidaPostfijo[indice] y la inicializa en 0
	int i, j, tamano = 0, operadorAbajo, operadorArriba; //operadorAbajo y operadorArriba servirán para analizar jerarquía
	char salidaPostfijo[MAX]; //salidaPostfijo contendrá la expresión en postfijo
	Initialize(&p1);
	for(i = 0; i < strlen(cadena); i++){
		if(cadena[i] != '(' && cadena[i] != ')') tamano++;
	}
	for(i = 0; i < strlen(cadena); i++){
		if(cadena[i] >= 97 && cadena[i] <= 122){ //Si es una letra minúscula de la 'a' a la 'z' (es decir, es un operando)
			salidaPostfijo[indice] = cadena[i]; //Directamente agrega el operando a salidaPostfijo
			printf("\n%i Operando detectado, agregado a arreglo salidaPostfijo[%d]", i+1, indice);
			indice++;
		}
		if(cadena[i] == '('){ //Cada paréntesis que abre hace un Push
			e1.c = cadena[i];
			Push(&p1, e1);
			printf("\n%d Analizando expresion, %c detectado", i+1, e1.c);
		}
		if(cadena[i] == ')' && Empty(&p1) == FALSE){
			j = 0;
			printf("\n%d ')' Detectado, sacando elementos de la pila hasta encontrar '('", i+1);
			do{
				e1 = Pop(&p1); //Si se detecta paréntesis de cierre se sacán todos los operadores de la Pila
```

```
                    if(esOperador(e1.c) == TRUE){
                        printf("\n%d.%d Operador %c detectado y desempilado,
pasado salidaPostfijo[%d]", i + 1, j, e1.c, indice);
                        salidaPostfijo[indice] = e1.c; //Los operadores que
van saliendo se van almacenando en salidaPostfijo
                        indice ++;
                    }
                    j++;
                }while(e1.c != '(' && Empty(&p1) == FALSE);//Saca operadores
hasta que llegue al '('
            }
            if(cadena[i] == '+' || cadena [i] == '-' || cadena[i] == '/' ||
cadena[i] == '*'){
                operadorArriba = precedencia(cadena[i]);//El operador que esta
por entrar a la pila procede del escaneo de cadena
                if(Empty(&p1) == FALSE){
                    e1 = Top(&p1);
                    operadorAbajo = precedencia(e1.c);          //El operador
que esta abajo del que se quiere introducir en la pila se obtiene con Top
                }
                else operadorAbajo = 0;
                if(operadorArriba <= operadorAbajo && Empty(&p1) == FALSE){ //Si
el operador que se busca introducir es de menor jerarquía
                    e1 = Pop(&p1); //hace un Pop explicando que un operador de
menor prioridad no puede estar arriba de uno de mayor
                    printf("\n Un operador '%c' no puede estar abajo de un
operador '%c', pasando '%c' a salidaPostfijo[%d]", e1.c, cadena[i], e1.c,
indice);
                    salidaPostfijo[indice] = e1.c; //Guarda el operador que se
sacó de la Pila en salidaPostfijo
                    indice++;
                    e1.c = cadena[i]; //Con el elemento conflictivo afuera será
posible introducir el operador a la Pila
                    Push(&p1, e1);
                    printf("\n El operador '%c' ya pudo ser introducido a la
pila", e1.c);
                }
                else{ //Si se cumple la jerarquía empila el operador sin problema
alguno
                    e1.c = cadena[i];
                    Push(&p1, e1);
                    printf("\n%d Analizando expresion, %c detectado y empilado",
i+1, e1.c);
                }
            }
            if(cadena[i] == '^'){ //El '^' al ser el de mayor prioridad puede
hacer una excepción a la comparación
                e1.c = cadena[i]; //y puede empilarse directamente
                Push(&p1, e1);
```

```
                printf("\n%d Analizando expresion, %c detectado y empilado", i+1,
e1.c);
            }
      }
      j = i; //Se guarda el valor que tenía i al final del for en j simplemente
para indicar subpasos (Ej 1.1, 1.2)
      printf("\nEl tamano de la pila de operadores es: %d", Size(&p1));
      for(i = 0; i <= Size(&p1); i++){ //Este ciclo for se hará para sacar
cualquier operador que todavía quedara en la pila
            if(Empty(&p1) == FALSE){
                  e1 = Pop(&p1);//y procede a guardarlos en salidaPostfijo
                  salidaPostfijo[indice] = e1.c;
            }
            printf("\n%d.%d Se saco el operador %c de la pila y se agrego a
salidaPostfijo[%d]", j + 1, i, e1.c, indice);
            indice++;
      }
      if(Empty(&p1) == TRUE){ //Cuando se haya vaciado la pila se imprime la
expresión en postfijo
            printf("\nLa pila esta vacia, a continuacion se imprimira la expresion
en Postfijo: ");
            printf("\nExpresion en postfijo = ");
            for(i = 0; i < tamano; i++){
                  printf("%c", salidaPostfijo[i]);
            }
            Destroy(&p1); //Se destruye la pila p1
            setbuf(stdin, NULL);
            printf("\n\n\n\n\t El resultado de la expresion es: %lf
",solucionAlgebra(salidaPostfijo, tamano)); //Se llama a la
            //función solucionAlgebra(char const *cadena, int n) y se le pasan la
salidaPostfijo y el tamaño de la cadena
            //original como argumentos, al finalizar retornará el resultado final,
mismo que se imprimirá
      }
      else printf("\n ERROR Desconocido");
      return;
}
```

Conclusiones

Oledo Enriquez Gilberto Irving

Finalmente puedo concluir que esta práctica es una muy buena práctica para comenzar ya que incluye muchos temas vistos en la unidad de aprendizaje previa a estructuras de datos y por lo tanto ayuda mucho a reforzar estos conocimientos y entenderlos para poder aplicarlos no de forma mecánica sino poder modificarlos y solucionar problemas, siempre y cuando entendamos el funcionamientos de cada uno de los temas aprendidos; esto me lleva a que poder decir que estamos aprendiendo a poder visualizar los problemas desde otro punto de vista en el cual como ya mencione no debemos mecanizar los conocimientos sino ver más allá y poder atacar los problemas de una forma más eficiente.

Aprendí lo relacionado con el TAD Pila y considero que fue de gran ayuda el realizar las dos implementaciones (estática y dinámica) ya que podemos ver de una forma más clara cuales son las ventajas que presenta una sobre la otra y como dije, tener otro método para la solución de problemas.

Finalmente también comprendí que las estructuras de datos son muy importantes ya que no solo es obtener y almacenar información o datos sino también poder organizarlos para usarlos posteriormente de una mejor manera.

Alanís Ramírez Damián

Con esta práctica reforcé mis conocimientos de programación estructurada, a la vez que aprendía a implementar nuevas estrategias para atacar problemas por medio de la programación. Aprendí a implementar el tipo de dato abstracto (TAD) conocido como pila, el cual cuenta con sus propiedades y operaciones específicas, mismas que nos ayudan a simplificar procesos que de otra forma costaría trabajo poder incluso expresar computacionalmente.

Aprendí tanto a estar del lado implementador, pudiendo definir por medio de un lenguaje de programación las operaciones elementales de una pila, como a ponerme del lado usuario, en este último caso, limitándome a emplear las operaciones propias del TAD pila, especificadas previamente. Y esto me ayuda a entender lo versátil que puede ser la programación, y lo importante que son las estructuras de datos para escribir código más eficiente y que sea más fácil de comprender.

Mendieta Torres Alfonso Ulises

Con esta práctica mejore mis conocimientos de punteros, comprendí el funcionamiento de un apuntador a una referencia, el operador flecha, entender el comportamiento de la pila y la manera de evaluar y convertir a posfijo es fácil, sin embargo pensar en el código no lo es tanto.

Bibliografía

[1] M. S. Montero, Universidad de Valladolid Departamento de informática Campus de Segovia, [En línea]. Available: https://www.infor.uva.es/~mserrano/EDI/cap2.pdf. [Último acceso: 30 Agosto 2017].

[2] I. Y. Villalobos, Unicauca, [En línea]. Available: http://artemisa.unicauca.edu.co/~nediaz/EDDI/cap07.pdf. [Último acceso: 30 Agosto 2017].

[3] A. D. Roldan, LIDIA Laboratorio de Investigación y desarrollo en Inteligencia Artificial, [En línea]. Available: http://quegrande.org/apuntes/EI/1/EDI/teoria/06-07/tad_-_pila_-_expresiones_aritmeticas.pdf. [Último acceso: 01 Septiembre 2017].