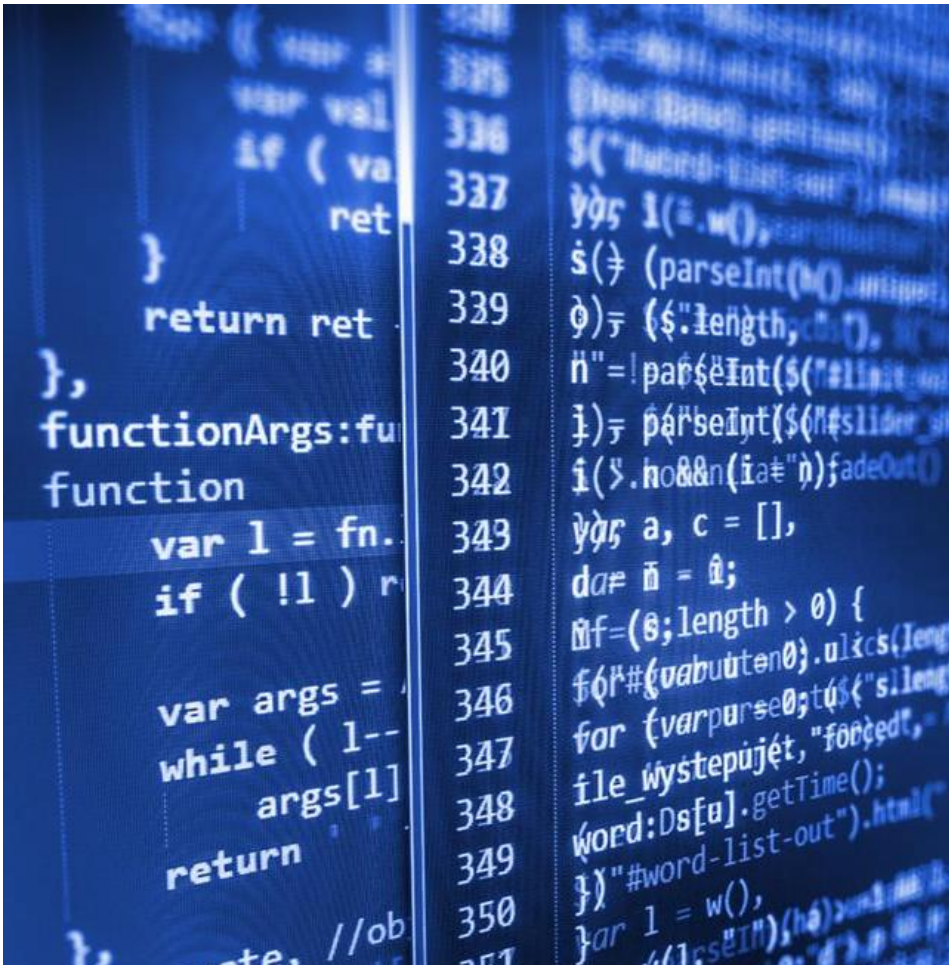


PRACTICA 1

EVALUACION DE EXPRESIONES INFIJAS

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.

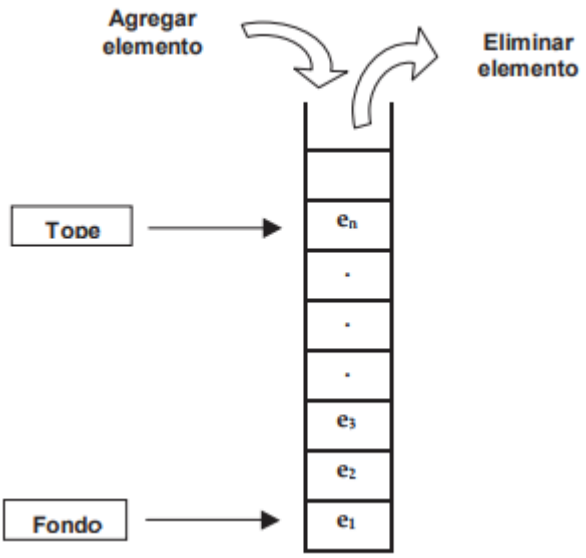
2. Especificación del tipo de datos abstracto.

3. Implementación. [1]

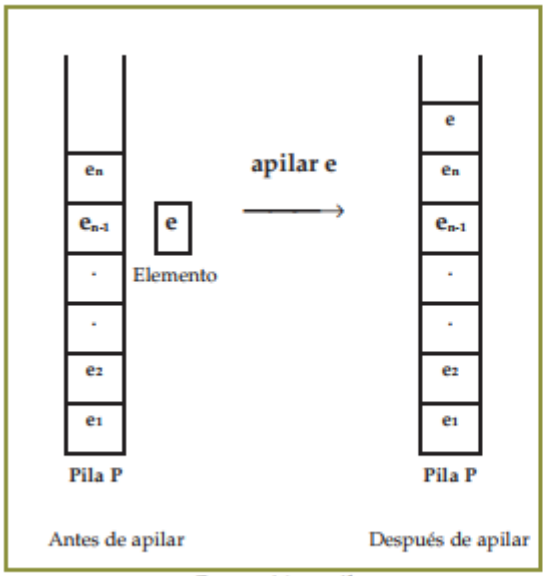
TAD pila.

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.

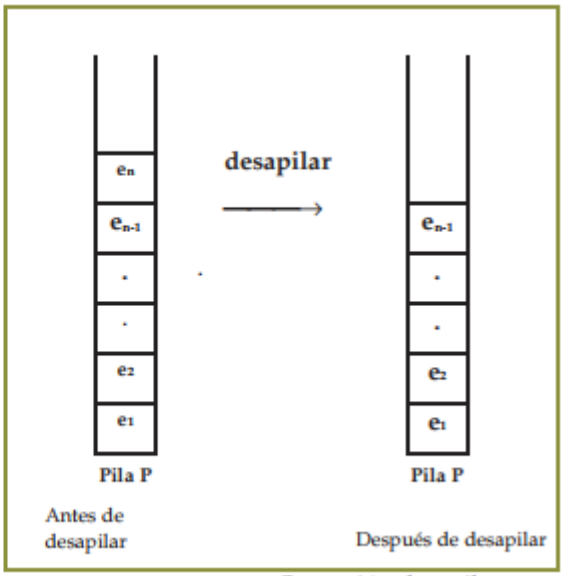
La representación gráfica de una pila es:



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.

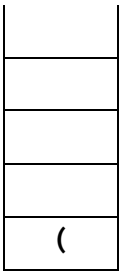
El indicador actual se incrementa si se hace un push, y se decrementa si se hace un pop. Y por supuesto, debe marcar un error cuando se esta en la base en donde el principio de pila es igual a indicador actual; lo que significa que la pila esta vaciá y se desea extraer un elemento (se marca underflow).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

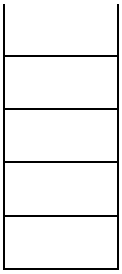
Para evaluar la concordancia de paréntesis en una expresión mediante una pila, lo que se hace es un push a la pila con un "(" cada vez que se encuentra en la expresión, y se hace un pop por cada ")" encontrado.

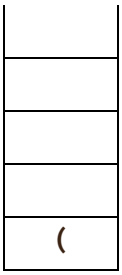
Después de recorrer toda la expresión que contiene paréntesis, si los paréntesis están correctos, la pila debe terminar vacía. En caso de que exista algún problema de concordancia de paréntesis y se notificara con una leyenda de “ERROR: Revisar sintaxis”

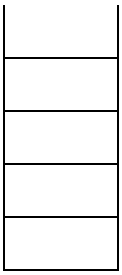
Ejemplo con paréntesis correctos:

(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***

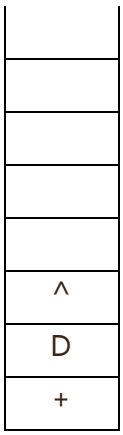
El operador es – vaciamos ***pila1*** y ponemos el tope de la izquierdo y el siguiente elemento de lado derecho efectuamos la operación y nuevamente guardamos el resultado en ***pila1.***

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

10 – 5 = 5

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

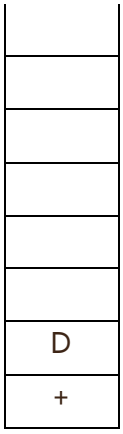
SalidaPostfijo que se muestra al usuario ***Pila Pila1***



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){

                                        elemento e;
                                        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 abre se realiza un pop del paréntesis en ***pila1***

```
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] == ')'){
                        Pop(&p1);
                        printf("\n%i Se detecto ')' por lo que se saco un '(' de la pila", i+1);
                        }
                        }
```

Al final la función `validarParentesis` devuelve un valor booleano y si la pila ***pila1*** esta 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 (expresion) este vacia, es decir, que todos sus elementos ya fueron analizados. .

```
    void pasarPostfijo(char const *cadena){

elemento e1;
                pila p1;
                int indice = 0;
                int i, j, n, operadorAbajo, operadorArriba, tamano;
                char salidaPostfijo[MAX];
                strcpy(salidaPostfijo, " ");
                setbuf(stdin, NULL);
                Initialize(&p1);
                tamano = tamanoReal(cadena);
                for(i = 0; i < strlen(cadena); i++){
                   if(cadena[i] >= 97 && cadena[i] <= 122){
                        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);//Saca operadores hasta que llegue
        al '('
        }
```

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:

```
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;
setbuf(stdin, NULL);
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){
                        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:

```
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.

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].