

2015 Septiembre

**J. VIVIANA MOLANO**  
**LIBARDO PANTOJA Y.**  
**RICARDO ZAMBRANO**

Departamento de Sistemas

Universidad del Cauca

LIBRO

# **ESTRUCTURAS DE DATOS DINAMICAS**

Una manera fácil de aprender



Editorial Unicauca

Departamento de Sistemas, Universidad del Cauca

J. Viviana Mora., W Libardo Pantoja Y.

Estructuras de Datos

# Estructuras de Datos Dinámicas

*Una forma fácil de aprender*



Revista digital

**Matemática, Educación e Internet.** (<http://www.tec-digital.itcr.ac.cr/revistamatematica/>).

Copyright© Universidad del Cauca ([www.unicauca.edu.co](http://www.unicauca.edu.co)).  
Correo Electrónico: [editorial@unicauca.edu.co](mailto:editorial@unicauca.edu.co)  
Departamento de Sistemas  
Facultad de Ingeniería Electrónica y Telecomunicaciones  
Apdo. 159-7050, Popayán  
Teléfono (057)8209800  
Fax (058)8209900

Mora, Viviana.  
Estructuras de Datos Dinámicas  
W. Libardo Pantoja Yépez  
– Departamento de Sistemas, FIET, Universidad del Cauca. 2015.  
xxx p.  
ISBN 978-9977-66-227-5  
1. Estructuras. 2. de Datos 3. Dinámicas.

Licencia.

Revista digital

**Matemática, Educación e Internet.**

<http://www.tec-digital.itcr.ac.cr/revistamatematica/>.



Este libro se distribuye bajo la licencia Creative Commons: Atribución-NoComercial-SinDerivadas CC BY-NC-ND (la “Licencia”). Usted puede utilizar este archivo de conformidad con la Licencia. Usted puede obtener una copia de la Licencia en <http://creativecommons.org/licenses/by-nc-nd/3.0/>. En particular, esta licencia permite copiado y distribución gratuita, pero no permite venta ni modificaciones de este material.

Límite de responsabilidad y exención de garantía: El autor o los autores han hecho su mejor esfuerzo en la preparación de este material. Esta edición se proporciona “tal cual”. Se distribuye gratuitamente con la esperanza de que sea útil, pero sin ninguna garantía expresa o implícita respecto a la exactitud o completitud del contenido.

La Revista digital Matemáticas, Educación e Internet es una publicación electrónica. El material publicado en ella expresa la opinión de sus autores y no necesariamente la opinión de la revista ni la del Instituto Tecnológico de Costa Rica.



# ÍNDICE GENERAL

## PRÓLOGO

VII

## 1

### CAPÍTULO 1: CONTENIDO Y ESTILOS EN LATEX

1

1.1 Estructura General del Libro

1

1.2 Prueba de entornos

4

Tablas

6

Ejemplo código fuente Java

6

## 2

### PILAS

7

2.1 Definición de Pila

7

2.2 El TAD Pila

7

2.3 Implementación del TAD en Java

8

2.4 Problemas que se resuelven con Pilas

12

Evaluación de la correspondencia de delimitadores

12

Evaluación de expresiones aritméticas

13

2.5 Ejercicios Propuestos

15



# Prólogo

Este texto cubre aspectos básicos e intermedios sobre ipsúm dolor sit amet, consectetur adipiscing elit. Nam dignissim varius tempus. Cras eu malesuada ipsum. Pellentesque ut lorem velit. Mauris vehicula est orci, bibendum tincidunt enim mattis a. Interdum et malesuada fames ac ante ipsum primis in faucibus..

...

*Popayan, 2015.*

V. MORA, W. PANTOJA.





# 1

## Capítulo 1: Contenido y Estilos en LaTeX

Este capítulo sirve como modelo, es decir, para mostrar cómo utilizar latex. También muestra el posible contenido del libro. Un ejemplo de referencia bibliográfica es: [1].

### Advertencia.

Las siguientes plantillas usan la versión 2014 del paquete `tcolorbox` (entre otros paquetes recientes), por lo tanto *debe actualizar los paquetes de sus distribución TeX* o instalar manualmente este paquete (ver el capítulo 9 del libro, [http://www.tec-digital.itcr.ac.cr/revistamatematica/Libros/LATEX/LaTeX\\_2014.pdf](http://www.tec-digital.itcr.ac.cr/revistamatematica/Libros/LATEX/LaTeX_2014.pdf)). El paquete “psboxit” viene incluido en la carpeta.

## 1.1 Estructura General del Libro

### 1. ANÁLISIS DE ALGORITMOS

- a) Los algoritmos
- b) El análisis de algoritmos
- c) Función de complejidad
- d) Cómo calcular la función de complejidad de un algoritmo
- e) Orden de Magnitud (Notación O Grande)
- f) Complejidad de un Algoritmo Recursivo
- g) Ejercicios Propuestos

### 2. INTRODUCCION A LAS ESTRUCTURAS DE DATOS

- a) Conceptos básicos sobre estructuras de datos.
- b) Clasificación.
  - 1) Estructuras de Datos Estáticas.
  - 2) Estructuras de Datos Dinámicas.

### 3. TIPOS ABSTRACTOS DE DATOS - TAD

- a) Tipos de datos

- b) Tipos abstractos de datos
- c) Métodos para la Especificación de un TAD
- d) Tipos de operaciones
- e) Ejemplos de TADs
- f) Ejercicios Propuestos

#### 4. LISTAS DINÁMICAS

- a) Definición
- b) Usos de las listas
- c) El TAD Lista
- d) Implementación del TAD Lista orientado a objetos
- e) Implementación del TAD lista mediante arreglos
- f) Casos de estudio
- g) Ejercicios propuestos

#### 5. PILAS

- a) Definición
- b) Usos de las pilas
- c) El TAD Pila
- d) Implementación del TAD pila orientado a objetos
- e) Implementación del TAD pila mediante arreglos
- f) Casos de estudio
  - 1) Correspondencia de delimitadores
  - 2) Evaluación de expresiones aritméticas
  - 3) Convertir una expresión dada en notación infija a una notación postfija
  - 4) Evaluación de la Expresión en notación postfija
- g) Ejercicios propuestos

#### 6. COLAS

- a) Definición
- b) Usos de las colas
- c) El TAD Cola
- d) Implementación del TAD cola orientado a objetos
- e) Implementación del TAD cola mediante arreglos
- f) Casos de estudio
  - 1) Correspondencia de delimitadores
  - 2) Evaluación de expresiones aritméticas
  - 3) Convertir una expresión dada en notación infija a una notación postfija
  - 4) Evaluación de la Expresión en notación postfija
- g) Ejercicios propuestos

## 7. ESTRUCTURAS DE DATOS NO LINEALES. ARBOLES BINARIOS

- a) Introducción
- b) Definición de árbol
- c) Definición de árbol binario
- d) Árbol de expresiones
- e) Balance o equilibrio de un árbol binario
- f) Árbol binario completo
- g) TAD Árbol binario
- h) Implementación del TAD de un árbol binario
- i) Recorridos de un árbol
  - 1) Recorrido inorden
  - 2) Recorrido en preorden
  - 3) Recorrido en postorden
  - 4) Recorrido en anchura
- j) Árbol binario de búsqueda
  - 1) Operación de inserción
  - 2) Operación de búsqueda
  - 3) Operación de eliminación
- k) Árbol binario de búsqueda equilibrados AVL
  - 1) Eficiencia en la búsqueda de un árbol equilibrado
  - 2) Inserción en árboles AVL
  - 3) Borrado de un nodo en un árbol AVL
- l) Ejercicios propuestos

## 8. ESTRUCTURAS DE DATOS NO LINEALES. ARBOLES N-ARIOS

- a) Introducción
- b) Definiciones y conceptos básicos
- c) El TAD ArbolN
- d) Implementación del TAD ArbolN
- e) Ejercicios propuestos

## 9. ARBOL1-2-3: UN ÁRBOL TRIARIO ORDENADO

- a) Introducción
- b) Definiciones
- c) El TAD ARBOL1-2-3
- d) Implementación del TAD ARBOL1-2-3
- e) Ejercicios propuestos

## 10. ARBOL2-3: UN ÁRBOL TRIARIO ORDENADO

- a) Introducción

- b)* Definiciones
- c)* Un árbol B
- d)* El TAD ARBOL2-3
- e)* Implementación del TAD ARBOL2-3
- f)* Ejercicios propuestos

#### 11. TRIE: CONJUNTO DE PALABRAS

- a)* Introducción
- b)* Definiciones
- c)* El TAD TRIE
- d)* Implementación del TAD TRIE
- e)* Ejercicios propuestos

#### 12. CUADTREE: REPRESENTACIÓN DE IMÁGENES

- a)* Introducción
- b)* Definiciones
- c)* El TAD CUADTREE
- d)* Implementación del TAD CUADTREE
- e)* Ejercicios propuestos

#### 13. ESTRUCTURA DINÁMICAS NO LINEALES: GRAFOS

- a)* Introducción
- b)* Definiciones
- c)* El TAD Grafo
- d)* Representación de los Grafos
  - 1) Matriz de adyacencia
  - 2) Implementación de la Matriz de Adyacencia
  - 3) Listas de adyacencia
  - 4) Implementación de la lista de Adyacencia
- e)* Recorridos de un Grafo
  - 1) Recorrido en anchura
  - 2) Recorrido en profundidad
- f)* Conexiones en un grafo
  - 1) Componentes conexas de un grafo
  - 2) Matriz de caminos, cierre transitivo
  - 3) Matriz de caminos y cierre transitivo
- g)* Matriz de caminos: Algoritmo de Warshall
- h)* Algoritmo de costos mínimos: Dijkstra
- i)* Algoritmo de Floyd
- j)* Ejercicios propuestos

## 1.2 Prueba de entornos

**Definición 1.1 (Igualdad)**

$$a = b$$

Según la definición 1.1, la igualdad...

**Teorema 1.1**

$$a = b$$

**Ejemplo 1.1**

$$a = b$$

**Lema 1.1**

$$a = b$$

**Corolario 1.1**

$$a = b$$

**Una caja de comentario**

$$a = b$$

**1.2.1 Tablas**

Iteración		
	$x_i$	$y_i = f(x_i)$
A	$x_0 = 0$	0
B	$x_1 = 0,75$	-0,0409838
C	$x_2 = 1,5$	1,31799

**1.2.2 Ejemplo código fuente Java**

```

1 package com.unicauca.ejemplo;
2 public class Hello {
3     //Comentario
4     /*Comentario*/
5     /**Comentario*/
6     public static void main(String[] args) {
7         System.out.println("Hola mundo");
8     }
9 }

```

## 2.1 Definición de Pila

Una pila (stack en inglés) es una lista ordinal o estructura de datos en la que el modo de acceso a sus elementos es de tipo LIFO (del inglés Last In First Out, último en entrar, primero en salir) que permite almacenar y recuperar datos. Se aplica en multitud de ocasiones en informática debido a su simplicidad y ordenación implícita en la propia estructura.

La Figura 2.3 muestra la representación gráfica de una pila con sus operaciones fundamentales de apilar (push en inglés) y desapilar o retirar (pop en inglés.).

La pila es muy útil en situaciones cuando los datos deben almacenarse y luego recuperarse en orden inverso.

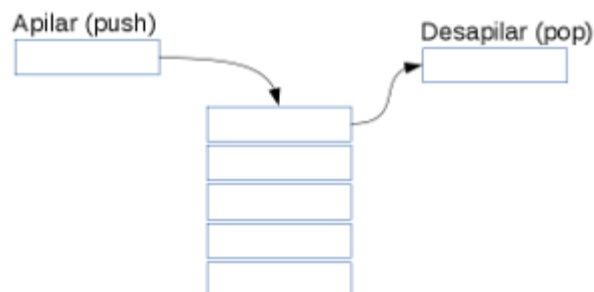
### Definición 2.1 Pila

Una pila (stack en inglés) es una lista ordinal o estructura de datos en la que el modo de acceso a sus elementos es de tipo LIFO (del inglés Last In First Out, último en entrar, primero en salir) que permite almacenar y recuperar datos. FALTA REFERENCIA

## 2.2 El TAD Pila

A continuación se especifica el TAD de la Pila con sus operaciones fundamentales. Las operaciones apilar y desapilar son las más importantes. En seguida la especificación de cada operación del TAD al estilo C.

**TAD Pila [ T ]**



**Figura 2.1.** Representación de una Pila



```

{ invariante: TRUE }
Constructoras:
    crearPila:
Modificadoras:
    apilar: Pila T
    desapilar: Pila
Analizadoras:
    cima: Pila
    esVacia: Pila
Destructoras:
    destruirPila: Pila

Pila crearPila( void )
/* Crea una pila vacia */
{ post: crearPila = }

void apilar(Pila pil, T elem)
/* Coloca sobre el tope de la pila el elemento elem */
{ post: pil = e1, e2, .. elem}

void desapilar(Pila pil)\
/* Elimina el elemento que se encuentra en el tope de la pila */
{ pre: pil =e1, e2, ..en, n > 0 }
{ post: pil =e1, e2, .., en-1 }

T cima(Pila pil )
/* Retorna el elemento que se encuentra en el tope de la pila */
{ pre: n > 0 }
{ post: cima = en }

int esVacia( Pila pil )
/* Informa si la pila esta vacia */
{ post: esVacia = ( pil = ) }

void destruirPila( Pila pil )
/* Destruye la pila retornando toda la memoria ocupada */
{post: pil ha sido destruida }

```

## 2.3 Implementación del TAD en Java

A continuación se muestra una implementación en Java del TAD Pila. Se ha implementado una pila dinámica utilizando nodos enlazados. Cada nodo almacena un valor y contiene una referencia al siguiente nodo. Supongamos una pila que almacene datos enteros, a la cual se le han aplicado las siguientes operaciones: `apilar(10)`, `apilar(35)`, `apilar(12)`, `apilar(7)`. La Figura 2.2 representa cómo quedaría la pila después de apilar en orden los datos 10, 35, 12 y 7. Se puede apreciar que cada nodo almacena un dato y a la vez existe una referencia que almacena la dirección del siguiente nodo. Además, existe una referencia llamada cabeza que apunta al

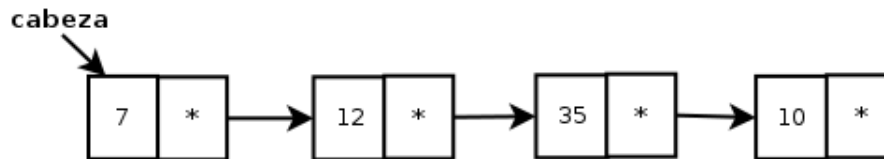


Figura 2.2. Representación de la Pila con Nodos Enlazados

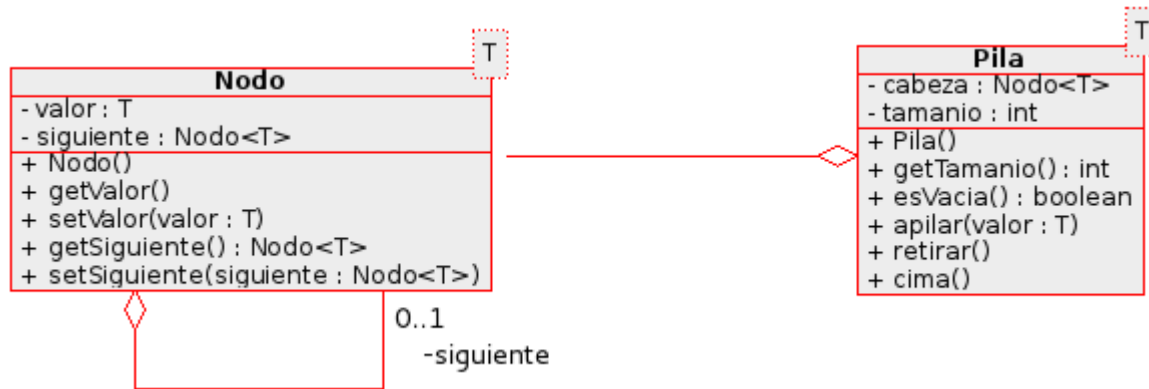


Figura 2.3. Diagrama de clase de la implementación del TAD Pila

último elemento que entró (en este caso al dato 7).

La Figura 2.3 muestra el diagrama de clases. Se puede apreciar básicamente dos clases: *Nodo* y *Pila*. La clase *Nodo* representa cada uno de los nodos enlazados que almacenan los objetos que se apilan. Tiene dos atributos, *valor* que representa el valor que guarda el nodo, en este caso es una referencia a un objetivo de tipo *T* (siendo *T* un tipo genérico). El atributo *siguiente*, representa la referencia al siguiente nodo. Los demás son únicamente, constructor y getters y setters de cada atributo. A continuación el código fuente de la clase *Nodo*.

```

1 package co.unicauca.pilas;
2 public class Nodo<T> {
3     //Atributo valor de tipo T. Almacena la referencia al objeto que se guarda
4     //en el nodo
5     private T valor;
6     //Referencia al siguiente nodo enlazado
7     Nodo<T> siguiente;
8     //Constructor por defecto
9     public Nodo() {
10         valor = null;
11         siguiente = null;
12     }
13     //Devuelve el valor
14     public T getValor() {
15         return valor;
16     }
17     //Modifica el valor
18     public void setValor(T valor) {
19         this.valor = valor;
20     }
21 }

```

```

20 //Devuelve el atributo siguiente
21 public Nodo<T> getSiguiente() {
22     return siguiente;
23 }
24 //Modifica el atributo siguiente
25 public void setSiguiente(Nodo<T> siguiente) {
26     this.siguiente = siguiente;
27 }
28 }

```

La clase *Pila* representa la pila como tal con sus operaciones principales de *apilar* y *retirar*. A continuación el código fuente de la clase *Pila*.

```

1 package co.unicauca.pilas;
2 public class Pila<T> {
3     //Atributo cabeza, que apunta al tope la pila
4     private Nodo<T> cabeza;
5     //Almacena el total de elemento de la pila
6     private int tamaño;
7     //Constructor por defecto
8     public Pila() {
9         cabeza = null;
10        tamaño = 0;
11    }
12    //Devuelve el total de elementos de la pila
13    public int getTamaño() {
14        return tamaño;
15    }
16    //Verifica si la pila esta vacia
17    public boolean esVacia() {
18        return (cabeza == null)
19    }
20    //Apila un elemento nuevo
21    public void apilar(T valor) {
22        //Crear un nuevo Nodo
23        Nodo<T> nuevo = new Nodo<T>();
24        //Fijaer el valor dentro del nodo
25        nuevo.setValor(valor);
26        if (esVacia()) {
27            //Cabeza apunta al nodo nuevo
28            cabeza = nuevo;
29        } else {
30            //Se enlaza el campo siguiente de nuevo con la cabeza
31            nuevo.setSiguiente(cabeza);
32            //La nueva cabeza de la pila pasa a ser nuevo
33            cabeza = nuevo;
34        }
35        //Incrementa el tamaño porque hay un nuevo elemento en la pila
36        tamaño++;

```

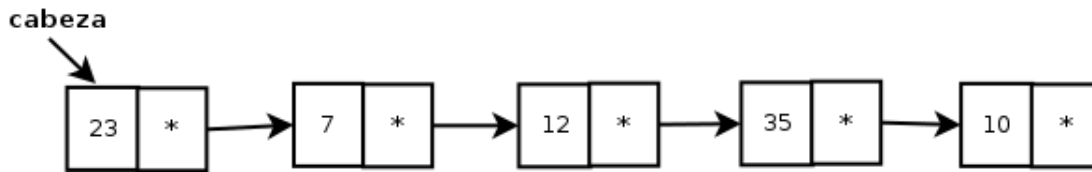


Figura 2.4. Representación de la Pila después de apilar el elemento 23

```

37     }
38     //Elimina un elemento de la pila
39     public void retirar() {
40         if (!esVacia()) {
41             cabeza = cabeza.getSiguiente();
42             tamano--;
43         }
44     }
45     //Devuelve el elemento almacenado en el tope de la pila
46     public T cima() {
47         if (!esVacia())
48             return cabeza.getValor();
49         else
50             return null;
51     }
52 }

```

En la línea 21 se puede apreciar el método **apilar**. Este método recibe como parámetro el **valor** que quiere apilar. Lo primero que se hace es crear un *nuevo* nodo y fijar su valor (líneas 23 y 24). Una vez creado el nodo, se pregunta si la pila es vacía (línea 26). En caso verdadero la referencia *cabeza* apunta al nodo nuevo (línea 28). De lo contrario se enlaza el campo *siguiente* de nuevo con la *cabeza* (línea 31), y finalmente la *cabeza* apunta a *nuevo* (línea 33) para indicar cual fue el último nodo que entró a la pila. Finalmente, la variable *tamano* se incrementa (indicando que la pila tiene una elemento mas). Por ejemplo si a la pila de la Figura 2.2 le aplicáramos la operación *apilar(23)*, nos daría como resultado la pila de la Figura 2.4.

En la línea 39 se aprecia el método **retirar** (o desapilar). Lo primero que hace es preguntar si la pila no esta vacía (línea 40), en caso afirmativo, la *cabeza* se desplaza al siguiente nodo (línea 41). Finalmente se decrementa la variable *tamano* (indicando que hay un elemento menos en la pila).

A continuación el código de un Cliente que instancia la Pila que hemos creado. En este caso se almacenan objetos de tipo entero, se apilan algunos números, se imprimen los valores del tope de pila y se desapilan sus elementos.

```

1 package co.unicauca.pilas;
2 public class ClienteMain {
3     public static void main(String[] args) {
4         //Crear una nueva pila de enteros
5         Pila<Integer> pila2 = new Pila<Integer>();
6         //Se apilan algunos datos enteros
7         pila2.apilar(2);
8         pila2.apilar(5);

```

```

9      pila2.apilar(7);
10     System.out.println("El tope de la pila es: " + pila2.cima());
11     //Se desapila
12     pila2.retirar();
13     System.out.println("El tope de la pila es: " + pila2.cima());
14     //Se desapila
15     pila2.retirar();
16     System.out.println("El tope de la pila es: " + pila2.cima());
17     //Se desapila, como la pila esta vacia devuelve null
18     pila2.retirar();
19     System.out.println("El tope de la pila es: " + pila2.cima());
20     //Probar con otra pila, donde se almacenen objetos
21 }
22 }

```

La salida por consola de este programa sería la siguiente.

```

El tope de la pila es: 7
El tope de la pila es: 5
El tope de la pila es: 2
El tope de la pila es: null

```

## 2.4 Problemas que se resuelven con Pilas

Las pilas son muy útiles en situaciones cuando los datos deben almacenarse y luego recuperarse en orden inverso. A continuación se ilustran algunos ejemplos.

### 2.4.1 Evaluación de la correspondencia de delimitadores

Un problema que se puede solucionar utilizando una pila, es la correspondencia de delimitadores en un programa. Esto es un ejemplo importante debido a que la correspondencia de delimitadores es parte de cualquier compilador: ningún programa se considera correcto si los delimitadores no tienen su pareja. Por ejemplo, la instrucción:

```
while (m<(n[8]+o)) { p = 7; /* comentarios */ 6=6;}
```

Se puede apreciar que está bien construida pues todos los paréntesis izquierdos corresponden con sus paréntesis derechos, así como las llaves y comentarios.

El algoritmo que resuelve la correspondencia adecuada de delimitadores evalúa la expresión de izquierda a derecha y sigue los siguientes pasos:

1. Obtener el carácter de la expresión y repetir pasos 2 al 3 para cada carácter.
2. Si es un operador de apertura ( ( , [ , /\* se lo apila.
3. Si es un operador de cierre: ) , ] , \*/:
  - a) Comparar que corresponda con el operador del tope de la pila.
  - b) Si no corresponde, termina el programa y la expresión es incorrecta.
  - c) Si hay correspondencia, se elimina elemento del tope de la pila y volver al paso 1.

4. Si al terminar de evaluar la expresión quedan elementos en la pila, la expresión es incorrecta.
5. Si al terminar de evaluar la expresión la pila queda vacía, la expresión es correcta.
6. Fin de algoritmo.

### 2.4.2 Evaluación de expresiones aritméticas

Las pilas se utilizan para evaluar expresiones aritméticas. Por ejemplo:

$$\frac{(100 + 23) * 231}{(31 - 14)^2} - 34$$

Para evaluar este tipo de expresiones se deben pasar a expresiones en *notación postfija* y luego aplicar un algoritmo para evaluar la expresión en notación postfija. Para entender este tipo de notaciones se sugiere ver el siguiente cuadro de definiciones.

#### Definición 2.2 Notaciones

La expresión **A+B** se dice que esta en *notación infija*, y su nombre se debe a que el operador + está entre los operandos A y B.

Dada la expresión **AB+** se dice que esta en *notación postfija* y su nombre se debe a que el operador + esta después de los operandos A y B.

Dada la expresión **AB** se dice que esta en *notación prefija*, y su nombre se debe a que el operador + está antes que los operandos A y B.

La ventaja de usar expresiones en notación postfija es que no son necesarios los paréntesis para indicar orden de operación, ya que éste queda establecido por la ubicación de los operadores con respecto a los operandos, Por ejemplo,

- Expresión infija:  $(X + Z) * W / T ^ Y - V$
- Expresión postfija:  $X Z + W * T Y ^ / V -$

Para convertir una expresión dada en notación infija a una notación postfija, deberá establecerse previamente ciertas condiciones:

- Solamente se manejarán los siguientes operadores (Están dados ordenadamente de mayor a menor según su prioridad de ejecución):

Operador	Prioridad dentro de la pila	Prioridad fuera de la pila
$^$ : potencia	3	3
$*$ / : Multiplicación y división	2	2
$+$ - : Suma y resta	1	1
( : Paréntesis izquierdo	0	4

- Los operadores de más alta prioridad se ejecutan primero. Si hubiera en una expresión dos o más operadores de igual prioridad, entonces se procesan de izquierda a derecha. Las subexpresiones parentizadas tendrán más prioridad que cualquier operador.
- Obsérvese que no se trata el paréntesis derecho ya que éste provoca sacar operadores de la pila hasta el paréntesis izquierdo.

### Algoritmo para Convertir una expresión infija a postfija

El algoritmo para convertir una expresión aritmética de notación infija a notación postfija debe tener en cuenta las siguientes consideraciones:

- Se parte de una expresión en notación infija que tiene operandos, operadores y puede tener paréntesis. Los operandos vienen representados por letras y los operadores son:  $\wedge$ ,  $*$ ,  $/$ ,  $+$ ,  $-$
- 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, caracter a caracter, 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 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 encuentre 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.

Con las anteriores consideraciones el algoritmo de conversión de expresiones infijas a postfijas es el siguiente:

1. Obtener caracteres de la expresión y repetir pasos 2 al 5 para cada caracter.
2. Si es un operando pasarlo a la expresión postfija
3. Si es un operador:
  - a) Si la pila está vacía, meterlo en la pila. Repetir a partir del paso 1.
  - b) Si la pila no está vacía:
    - 1) Si la prioridad del operador leído es mayor que la prioridad del operador cima de la pila, meterlo en la pila y repetir a partir del paso 1.
    - 2) Si la prioridad del operador es menor o igual que la prioridad del operador de la cima, sacar operador cima de la pila y pasarlo a la expresión postfija. Volver al paso 3.
4. Si es paréntesis derecho:
  - a) Sacar operador cima de la pila y pasarlo a la expresión postfija.
  - b) Si nueva cima es paréntesis izquierdo, suprimir elemento cima.

- c) Si cima no es paréntesis izquierdo volver a paso 4.1
- d) Volver a partir del paso 1.
- 5. Si es paréntesis izquierdo pasarlo a la pila.
- 6. Si quedan elementos en la pila, pasarlos a la expresión postfija
- 7. Fin de algoritmo

Como se puede apreciar el algoritmo anterior utiliza como mecanismo central el uso de pilas.

#### **Evaluación de la Expresión en notación postfija**

Una vez convertida la expresión infija a notación postfija se puede aplicar un algoritmo (el cual también maneja una pila) para evaluar dicha expresión y calcular su valor. Para ello, se almacena la expresión aritmética transformada a notación postfija en un vector, en la que los operandos están representados por variables de una sola letra. Antes de evaluar la expresión 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 de operandos de números reales. El algoritmo se describe a continuación.

1. Examinar el vector desde el elemento 1 hasta el N, repetir los pasos 2 y 3 para cada elemento del vector.
2. Si el elemento es un operando meterlo en la pila.
3. Si el elemento es un operador, lo designamos por ejemplo con +:
  - a) Sacar los dos elementos superiores de la pila, los denominamos con los identificadores x,y respectivamente.
  - b) Evaluar  $x+y$ ; el resultado es  $z = x + y$ .
  - c) El resultado z, meterlo en la pila.
  - d) Repetir a partir del paso 1.
4. El resultado de la evaluación de la expresión está en el elemento cima de la pila.
5. Fin de algoritmo.

## **2.5 Ejercicios Propuestos**

A continuación se plantean los siguientes ejercicios los cuales utilizan pilas como estructura central.

1. Utilizando el algoritmo de conversión de infijas a postfijas, pasar a notación postfija las siguientes expresiones:
  - a)  $(X + Y * Q) / R^T$  *Respuesta:*  $XYQ * + RT ^ /$
  - b)  $X ^ Y + Z - A * D - R$  *Respuesta:*  $XY ^ Z + AD * - R -$
  - c)  $A + (B - D) / Y - (A - C) / (Y - X) + Z$  *Respuesta:*  $ABD - Y / + AC - YX - / - Z +$
2. De las anteriores expresiones postfijas, dar valores numéricos a las variables y aplicar el algoritmo que permite evaluar una expresión postfija.



3. Buscar en Internet un algoritmo que permita pasar expresiones de notación infija a notación prefija. Entender el algoritmo mediante ejemplos.
4. Proponga una implementación en java (o cualquier lenguaje) del algoritmo que evalúa la correspondencia de limitadores.
5. Proponga una implementación en java (o cualquier lenguaje) del algoritmo que pasa expresiones de notación infija a postija.
6. Proponga una implementación en java (o cualquier lenguaje) del algoritmo que evalúa expresiones en notación postfija.

## Bibliografía

- [1] G. Kappel, B. Pröll, S. Reich, and W. Retschitzegger, *Web Engineering: The Discipline of Systematic Development of Web Applications*. Wiley, 2006.