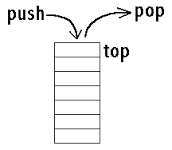
la pila. muestra una secuencia de operaciones Insertar y Quitar. El último elemento añadido a la pila es el primero que se quita de ella

La operación Insertar (push) sitúa un elemento dato en la cima de la pila, y Quitar (pop)

elimina o quita el elemento de la pila



La pila se puede implementar guardando los elementos en un array, en cuyo caso su dimensión o longitud es fija. También se puede utilizar un Vector para almacenar los elementos.

Otra forma de implementación consiste en construir una lista enlazada, de modo que cada elemento de la pila forma un nodo de la lista. La lista crece o decrece según se añaden o se extraen, respectivamente, elementos de la pila; ésta es una representación dinámica, y no existe limitación en su tamaño excepto la memoria del computador.

Una pila puede estar vacía (no tiene elementos) o llena (en la representación con un array —arreglo—, si se ha llegado al último elemento). Si un programa intenta sacar un elemento de una pila vacía, se producirá un error, una excepción, debido a que esa operación es imposible; esta situación se denomina desbordamiento negativo (underflow). Por el contrario, si un programa intenta poner un elemento en una pila llena, se produce un error, una excepción, de desbordamiento (overflow) o rebosamiento. Para evitar estas situaciones se diseñan métodos que comprueban si la pila está llena o vacía.

**TIPO DE DATO PILA IMPLEMENTADO CON ARRAYS**

Los elementos que forman la pila se guardan en arrays (arreglos), en el contenedor “Vector” o bien formando una lista enlazada. La implementación con un array (arreglo) es estática ya que el arrray es de tamaño fijo.

En otras palabras, podría sr útil en algunas ocasiones, pero la idea es ver, que podamos crear una pila sin necesidad de estar preocupado por el tamaño.

La clase PilaLineal, con esta representación, necesita un array y una variable numérica, cima, que apunte al último elemento colocado en la pila. Al utilizar un array es necesario tener en cuenta que el tamaño de la pila no puede exceder el número de elementos del array, y la condición pila llena será significativa para el diseño.

El método usual de introducir elementos en una pila es definir el fondo de la pila en la posición 0 del array y sin ningún elemento en su interior, es decir, definir una pila vacía; a continuación, se van introduciendo elementos en la pila (en el array), de modo que el primer elemento añadido se introduce en una pila vacía y en la posición 0, el segundo elemento en la posición 1, el siguiente en la posición 2 y así sucesivamente. Con estas operaciones, el índice que apunta a la cima de la pila se va incrementando en 1 cada vez que se añade un nuevo elemento. Los algoritmos de introducir, “insertar” (push) y “quitar”, sacar, (pop) datos de la pila son

1. Verificar si la pila no está llena.

2. Incrementar en 1 el puntero índice de la pila.

3. Almacenar elemento en la posición del puntero de la pila

**Quitar (pop)**

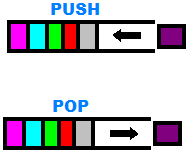
1. Verificar si la pila no está vacía.

2. Leer el elemento de la posición del puntero de la pila.

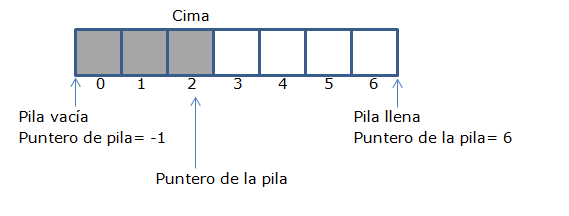
3. Decrementar en 1 el puntero de la pila

El rango de elementos que puede tener una pila varía de 0 a TAMPILA-1 en el supuesto de que el array se defina de tamaño TAMPILA elementos. ​

De modo que, en una pila llena, el puntero (índice del array) de la pila tiene el valor TAMPILA-1, y en una pila vacía el puntero tendrá el valor -1 (el valor 0, teóricamente, es el índice del primer elemento).​

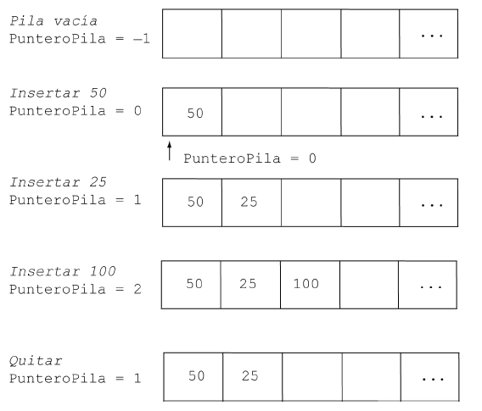


Una pila de 7 elementos se puede representar gráficamente así:



A continuación, se muestra la imagen de una pila según diferentes operaciones realizadas en

un posible programa



**Clase PilaLinea:**

La declaración de un tipo abstracto incluye la representación de los datos y la definición de las

operaciones. En el TAD Pila los datos pueden ser de cualquier tipo, y las operaciones, las citadas

anteriormente en el apartado 9.1.1.

1. Datos de la pila (TipoDato es cualquier tipo de dato primitivo o tipo clase).

2. crearPila inicializa una pila. Es como crear una pila sin elementos, por tanto, vacía.

3. Verificar que la pila no está llena antes de insertar o poner (“push”) un elemento en la

pila; verificar que una pila no está vacía antes de quitar o sacar (“pop”) un elemento de

la pila. Si estas precondiciones no se cumplen, se debe visualizar un mensaje de error y el

programa debe terminar.

4. pilaVacia devuelve verdadero si la pila está vacía y falso en caso contrario.

5. pilaLlena devuelve verdadero si la pila está llena y falso en caso contrario. Estas dos

últimas operaciones se utilizan para verificar las precondiciones de insertar y quitar.

6. limpiarPila vacía la pila, dejándola sin elementos y disponible para otras tareas.

7. cimaPila devuelve el valor situado en la cima de la pila, pero no se decrementa su

puntero, ya que la pila queda intacta.

Definición:

package tipoPila;  
  
public class PilaLineal {  
 private static final int TAMPILA = 49;  
 private int cima;  
 private Object [] listaPila;  
 // operaciones que añaden o extraen elementos  
 public void insertar(Object elemento)  
 public Object quitar()throws Exception  
 public Object cimaPila()throws Exception  
 // resto de operaciones  
}

Escribir un programa que cree una pila de enteros y realice operaciones de añadir datos a la pila, quitar...​

Se supone implementada la pila con el tipo primitivo int. ​

El programa crea una pila de números enteros, inserta en la pila elementos leídos del teclado (hasta leer la clave -1) y a continuación extrae los elementos de la pila hasta que se vacía. ​

El bloque de sentencias se encierra en un bloque try para tratar errores de desbordamiento de la pila.​

package tipoPila;  
  
public class PilaLineal {  
 private static final int TAMPILA = 49;  
 private int cima;  
 private int [] listaPila;  
  
 public PilaLineal() {  
 cima = -1; // condición de pila vacía  
 listaPila = new int[TAMPILA];  
 }  
  
 // operaciones que modifican la pila  
 public void insertar(int elemento){}  
  
 public int quitar(){}  
  
 public void limpiarPila(){}  
  
 // operación de acceso a la pila  
 public int cimaPila(){}  
  
 // verificación estado de la pila  
 public boolean pilaVacia(){}  
  
 public boolean pilaLlena(){}  
}

package tipoPila;  
  
import java.io.\*;  
  
public class EjemploPila {  
  
 public static void main(String[] args) {  
 // TODO Auto-generated method stub  
 PilaLineal pila;  
 int x;  
 final int CLAVE = -1;  
 BufferedReader entrada = new BufferedReader(new InputStreamReader(System.in));  
 System.out.println("Teclea los elemento (termina con -1).");  
 try {  
 pila = new PilaLineal(); // crea pila vacía  
 do {  
 x = Integer.parseInt(entrada.readLine());  
 pila.insertar(x);  
 } while (x != CLAVE);  
 System.out.println("Elementos de la Pila: ");  
 while (!pila.pilaVacia()) {  
 x = pila.quitar();  
 System.out.print(x + " ");  
 }  
 } catch (Exception er) {  
 System.err.println("Excepcion: " + er);  
 }  
 }  
  
}

Declarar la clase Pila con elementos de tipo Object. ​

Insertar y extraer de la pila datos de tipo entero.​

{

private static final int TAMPILA = 49;

private int cima;

private Object [] listaPila;

// operaciones que añaden o extraen elementos

public void insertar(Object elemento)

public Object quitar()throws Exception

public Object cimaPila()throws Exception

// resto de operaciones

}

El siguiente segmento de código inserta los datos 11, 50 y 22:

PilaLineal pila = new PilaLineal();

pila.insertar(new Integer(11));

pila.insertar(new Integer(50));

pila.insertar(new Integer(22));

Para extraer de la pila y asignar el dato a una variable:

Integer dato;

dato = (Integer) pila.quitar();

**Implementación de las operaciones sobre pilas**

Los métodos de la clase Pila se implementan fácilmente, teniendo en cuenta la característica

principal de esta estructura: inserciones y borrados se realizan por el mismo extremo, la cima

de la pila.

Las operaciones insertar() y quitar() añaden y eliminan, respectivamente, un elemento

de la pila; la operación cimaPila permite a un cliente recuperar los datos de la cima de la pila

sin quitar realmente el elemento de la misma

**método Insertar():**

publicó void insertar(TipoDeDato elemento) throws Exception

{

if (pilaLlena())

{

throw new Exception("Desbordamiento pila");

}

//incrementar puntero cima y copia elemento

cima++;

listaPila[cima] = elemento;

}

**Operación Quitar:**

public TipoDeDato quitar() throws Exception

{

TipoDeDato aux;

if (pilaVacia())

{

throw new Exception ("Pila vacía, no se puede extraer.");

}

// guarda elemento de la cima

aux = listaPila[cima];

// decrementar cima y devolver elemento

cima--;

return aux;

}

**Operación cimaPila** devuelve el elemento que se encuentra en la cima de la pila. ​

No se modifica la pila, únicamente se accede al elemento​

public TipoDato cimaPila() throws Exception

{

if (pilaVacia())

{

throw new Exception ("Pila vacía, no hay elementos.");

}

return listaPila[cima];

}

**Operaciones de verificación del estado de la pila​:**

El método **pilaVacia** comprueba (verifica) si la cima de la pila es -1.​

En ese caso, la pila está vacía y devuelve verdadero; en caso contrario, devuelve falso.​

public boolean pilaVacia()

{

return cima == -1;

}

El método **pilaLlena** comprueba (verifica) si la cima es TAMPILA - 1. ​

En ese caso, la pila está llena y devuelve verdadero; en caso contrario, devuelve falso.​

public boolean pilaLlena()

{

return cima == TAMPILA - 1;

}

La operación limpiarPila pone la cima de la pila a su valor inicial.​

public void limpiarPila()

{

cima = -1;

}

Escribir un programa que utilice una Pila para comprobar si una determinada frase/palabra (cadena de caracteres) es un palíndromo. ​

Nota: una palabra o frase es un palíndromo cuando la lectura directa e indirecta de la misma tiene igual valor: alila, es un palíndromo; cara (arac) no es un palíndromo.​

package palindromo;  
  
public class PilaLineal {  
 private static final int TAMPILA = 79;  
 private int cima;  
 private Object[] listaPila;  
  
 public PilaLineal() {  
 cima = -1;  
 listaPila = new Object[TAMPILA];  
 }  
  
 public void insertar(Object elemento) throws Exception {  
 if (pilaLlena()) {  
 throw new Exception("Desbordamiento pila");  
 }  
 cima++;  
 listaPila[cima] = elemento;  
 }  
  
 public Object quitar() throws Exception {  
 Object aux;  
 if (pilaVacia()) {  
 throw new Exception("Pila vacía, no se puede extraer.");  
 }  
 aux = listaPila[cima];  
 cima--;  
 return aux;  
 }  
  
 public Object cimaPila() throws Exception {  
 if (pilaVacia()) {  
 throw new Exception("Pila vacía, no se puede extraer.");  
 }  
 return listaPila[cima];  
 }  
  
 public boolean pilaVacia() {  
 return cima == -1;  
 }  
  
 public boolean pilaLlena() {  
 return cima == TAMPILA - 1;  
 }  
  
 public void limpiarPila() {  
 cima = -1;  
 }  
}

Main

package palindromo;  
  
import java.io.\*;  
  
public class Palindromo {  
  
 public static void main(String[] args) {  
 // TODO Auto-generated method stub  
 PilaLineal pilaChar;  
 //char ch;  
 boolean esPal;  
 String pal;  
 BufferedReader entrada = new BufferedReader(new InputStreamReader(System.in));  
 try {  
 pilaChar = new PilaLineal(); // crea pila vacía  
 System.out.print("Teclea la palabra" + " a verificar si es palíndromo: ");  
 pal = entrada.readLine();  
 // se crea la pila con los caracteres de la palabra  
 for (int i = 0; i < pal.length();) {  
 Character c;  
 c = pal.charAt(i++);  
 if (c != ' ')  
 pilaChar.insertar(c);  
 }  
 // se comprueba si es palíndromo  
 esPal = true;  
 for (int j = 0; esPal && !pilaChar.pilaVacia();) {  
 Character c;  
 c = (Character) pilaChar.quitar();  
 while (pal.charAt(j) == ' ')  
 j++;  
 esPal = pal.charAt(j++) == c.charValue();  
 }  
 pilaChar.limpiarPila();  
 if (esPal)  
 System.out.println("La palabra " + pal + " es un palíndromo \n");  
 else  
 System.out.println("La palabra " + pal + " no es un palíndromo \n");  
 } catch (Exception er) {  
 System.err.println("Excepcion: " + er);  
 }  
 }  
  
}

**PILA DINÁMICA IMPLEMENTADA CON UN VECTOR​:**

La clase Vector es un contenedor de objetos que puede ser manejado para que crezca y decrezca dinámicamente. ​

Los elementos de Vector son de tipo Object. ​

Dispone de métodos para asignar un elemento en una posición (insertElementAt), para añadir un elemento a continuación del último (addElement), para obtener el elemento que se encuentra en una posición determinada (elementAt), para eliminar un elemento (removeElementAt)... ​

Entonces, es posible mejorar la implementación del tipo abstracto Pila utilizando como depósito de los elementos un objeto Vector**.​**

La posición del último elemento añadido a la pila se mantiene con la variable cima.​

Inicialmente, cima = -1, que es la condición de pila vacía.​

Insertar un nuevo elemento supone aumentar cima y asignar el elemento a la posición cima del Vector. ​

La operación se realiza llamando al método addElement(), que añade un elemento a partir del último y, si es necesario, aumenta la capacidad. ​

No es necesario implementar el método pilaLlena() ya que la capacidad del Vector crece dinámicamente.​

**El método quitar()** devuelve el elemento cima de la pila y lo elimina. ​

Al utilizar un Vector, llamando a removeElementAt(cima) se elimina el elemento cima, y a continuación se decrementa cima.​

package pilaVector;  
  
import java.util.Vector;  
  
public class PilaVector {  
 private static final int INICIAL = 19;  
 private int cima;  
 private Vector listaPila;  
  
 public PilaVector() {  
 cima = -1;  
 listaPila = new Vector(INICIAL);  
 }  
  
 public void insertar(Object elemento) throws Exception {  
 cima++;  
 listaPila.addElement(elemento);  
 }  
  
 public Object quitar() throws Exception {  
 Object aux;  
 if (pilaVacia()) {  
 throw new Exception("Pila vacía, no se puede extraer.");  
 }  
 aux = listaPila.elementAt(cima);  
 listaPila.removeElementAt(cima);  
 cima--;  
 return aux;  
 }  
  
 public Object cimaPila() throws Exception {  
 if (pilaVacia()) {  
 throw new Exception("Pila vacía, no se puede extraer.");  
 }  
 return listaPila.elementAt(cima);  
 }  
  
 public boolean pilaVacia() {  
 return cima == -1;  
 }  
  
 public void limpiarPila() throws Exception {  
 while (!pilaVacia())  
 quitar();  
 }  
}

**Nota de programación​**

**Para utilizar una pila de elementos de tipo primitivo (int, char, long, float, double...) es necesario, para insertar, crear un objeto de la correspondiente clase envolvente (Integer, Character, Long, Float, Double...) y pasar dicho objeto como argumento del método insertar().​**

Llenar una pila con números leídos del teclado. ​

A continuación, vaciar la pila de tal forma que se muestren los valores positivos.​

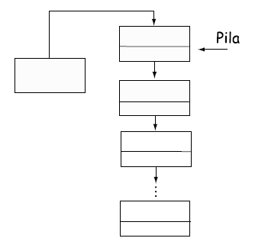
package pilaVector;  
  
import java.io.\*;  
  
public class EjemploPilaDinamica {  
  
 public static void main(String[] args) {  
 // TODO Auto-generated method stub  
 PilaVector pila = new PilaVector();  
 int x;  
 BufferedReader entrada = new BufferedReader(new InputStreamReader(System.in));  
 System.out.println("Teclea número de elementos: ");  
 try {  
 x = Integer.parseInt(entrada.readLine());  
 for (int j = 1; j <= x; j++) {  
 Double d;  
 d = Double.valueOf(entrada.readLine());  
 pila.insertar(d);  
 }  
 // vaciado de la pila  
 System.out.println("Elementos de la Pila: ");  
 while (!pila.pilaVacia()) {  
 Double d;  
 d = (Double) pila.quitar();  
 if (d.doubleValue() > 0.0)  
 System.out.print(d + " ");  
 }  
 } catch (Exception er) {  
 System.err.println("Excepcion: " + er);  
 }  
 }  
  
}

**Implementación de las operaciones del TAD Pila​**

**con listas enlazadas​:**

La realización dinámica de una pila utilizando una lista enlazada almacena cada elemento de la pila como un nodo de la lista, con la particularidad típica del TAD Pila: se inserta un elemento por el mismo extremo por el que se extrae, es decir, por la cima de la pila.

Esta realización tiene la ventaja de que el tamaño se ajusta exactamente al número de elementos de la pila. Sin embargo, para cada elemento es necesaria más memoria ya que hay que guardar el campo de enlace entre nodos consecutivos. La Figura muestra la imagen de una pila implementada con una lista enlazada.



La estructura que tiene la pila implementada con una lista enlazada es muy similar a la expuesta

en listas enlazadas. Los elementos de la pila son los nodos de la lista, con un campo para guardar

el elemento y otro de enlace. Las operaciones del tipo pila implementada con listas son, naturalmente, las mismas que si la pila se implementa con arrays, salvo la operación que controla si

la pila está llena, pilaLlena, que ahora no tiene significado ya que las listas enlazadas crecen

indefinidamente, con el único límite de la memoria.

La clase Nodo Pila representa un nodo de la lista enlazada. Tiene dos atributos: elemento

guarda el elemento de la pila y siguiente contiene la dirección del siguiente nodo de la lista.

El constructor pone el dato en elemento e inicializa siguiente a null. El tipo de dato de

elemento se corresponde con el tipo de los elementos de la pila para que no dependa de un tipo

concreto; para que sea más genérico, se utiliza el tipo Object y, de esa forma, puede almacenar

cualquier tipo de referencia

package TipoPila;

public clase NodoPila

{

Object elemento;

NodoPila siguiente;

NodoPila(Object x)

{

elemento = x;

siguiente = null;

}

}

**Implementación de las operaciones del TAD Pila​**

**con listas enlazadas​**

No necesitan recorren los nodos de la lista, no dependen del número de nodos, la eficiencia de cada operación es constante, O(1).​

**Verificación del estado de la pila.**

public boolean pilaVacia()

{

return cima == null;

}

**Poner un elemento en la pila. Crea un nuevo nodo con el elemento que se pone en la pila y se**

**enlaza por la cima.**

public void insertar(Object elemento)

{

NodoPila nuevo;

nuevo = new NodoPila(elemento);

nuevo.siguiente = cima;

cima = nuevo;

}

**Eliminación del elemento cima. Retorna el elemento cima y lo quita de la pila.**

**public Object quitar() throws Exception**

{

if (pilaVacia())

throw new Exception ("Pila vacía, no se puede extraer.");

Object aux = cima.elemento;

cima = cima.siguiente;

return aux;

}

**Obtención del elemento cabeza o cima de la pila, sin modificar la pila:**

public Object cimaPila() throws Exception

{

if (pilaVacia())

throw new Exception ("Pila vacía, no se puede leer cima.");

return cima.elemento;

}

**Vaciado de la pila. Libera todos los nodos de que consta la pila. Recorre los n nodos de la lista**

**enlazada; por consiguiente, es una operación lineal, O(n).**

public void limpiarPila()

{

NodoPila t;

while(!pilaVacia())

{

t = cima;

cima = cima.siguiente;

t.siguiente = null;

}

}

**EVALUACIÓN DE EXPRESIONES ARITMÉTICAS CON PILAS​:**

Una expresión aritmética está formada por operandos y operadores. ​

La expresión x\*y - (a+b) consta de los operadores \*, -, + y de los operandos x, y, a, b. ​

Los operandos pueden ser valores constantes, variables o, incluso, otra expresión. ​

Los operadores son los símbolos conocidos de las operaciones matemáticas. ​

La evaluación de una expresión aritmética da lugar a un valor numérico, se realiza sustituyendo los operandos que son variables por valores concretos y ejecutando las operaciones aritméticas representadas por los operadores. ​

Si los operandos de la expresión anterior toman los valores x = 5, y = 2, a = 3, b = 4, el resultado de la evaluación es:​

5\*2 – (3+4) = 5\*2 – 7 = 10 – 7 = 3​

La forma habitual de escribir expresiones matemáticas es aquella en la que el operador está entre sus dos operandos.​

La expresión anterior está escrita de esa forma, y recibe el nombre de notación infija. ​

Esta forma de escribir las expresiones exige, en algunas ocasiones, el uso de paréntesis para encerrar subexpresiones con mayor prioridad.​

Los operadores, como es sabido, tienen distintos niveles de precedencia o prioridad a la hora de su evaluación. ​

A continuación, se recuerdan estos niveles de prioridad en orden de mayor a menor:​

Paréntesis: ()​

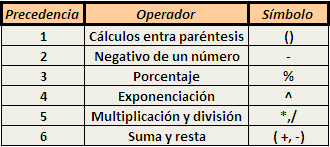
Potencia: ^​

Multiplicación/división: \*, /​

Suma/Resta: +, -

Normalmente, en una expresión hay operadores con la misma prioridad. ​

A igualdad de precedencia, los operadores se evalúan de izquierda a derecha (asociatividad), excepto la potencia, que es de derecha a izquierda.​



**Notación prefija y notación postfija de una expresión aritmética​:**

La forma habitual de escribir operaciones aritméticas (notación infija) sitúa el operador entre sus dos operandos. ​

Esta forma de notación obliga, en muchas ocasiones, a utilizar paréntesis para indicar el orden de evaluación. ​

Las expresiones siguientes:​

r = a\*b/(a+c) g = a\*b/a+c​

son distintas al no poner paréntesis en la expresión g. ​

**​**Igual ocurre con estas otras:​

r = (a-b)^c+d g = a-b^c+d​

Existen otras formas de escribir expresiones aritméticas, que se diferencian por la ubicación del operador respecto de los operandos. ​

La notación en la cual el operador se coloca delante de los dos operandos, se conoce como notación prefija y se conoce como notación polaca (en honor del matemático polaco que la propuso).​

Dadas las expresiones: a\*b/(a+c); a\*b/a+c; (a-b)^c+d . ​

Escribir las expresiones equivalentes en notación prefija.​

Se escribe, paso a paso, la transformación de cada expresión aritmética en la expresión equivalente e​

a\*b/(a+c) (infija) → a\*b/+ac → \*ab/+ac → /\*ab+ac (polaca)​

a\*b/a+c (infija) → \*ab/a+c → /\*aba+c → +/\*abac (polaca)​

(a-b)^c+d (infija) → -ab^c+d → ^-abc+d → +^-abcd (polaca)​

En el Ejemplo, se observa que no son necesarios los paréntesis al escribir la expresión en notación polaca. ​

Ésta es su propiedad fundamental: el orden de ejecución de las operaciones está determinado por las posiciones de los operadores y los operandos en la expresión.​

**Notación postfija​**

Hay más formas de escribir las expresiones. ​

La tercera notación más utilizada se denomina postfija o polaca inversa y coloca el operador a continuación de sus dos operandos.​

Ejemplo

Dadas las expresiones: a\*b/(a+c); a\*b/a+c; (a-b)^c+d, escribir las expresiones equivalentes en notación postfija.​

Paso a paso, se transforma cada subexpresión en notación postfija.​

a\*b/(a+c) (infija) → a\*b/ac+ → ab\*/ac+ → ab\*ac+/ (polaca inversa)​

a\*b/a+c (infija) → ab\*/a+c → ab\*a/+c → ab\*a/c+ (polaca inversa)​

(a-b)^c+d (infija) → ab-^c+d → ab-c^+d → ab-c^d+ (polaca inversa)​

**Evaluación de una expresión aritmética​:**

La evaluación de una expresión aritmética escrita de manera habitual, en notación infija, se realiza en dos pasos principales:​

1. Transformar la expresión de notación infija a postfija.​

2. Evaluar la expresión en notación postfija.​

En los algoritmos que se aplican, en cada uno de los pasos es básico el uso del TAD Pila. ​

La estructura pila asegura que el último en entrar es el primero en salir y, de esa forma, el algoritmo de transformación a postfija sitúa los operadores después de sus operandos con la prioridad o precedencia que les corresponde. ​

Una vez que se tiene la expresión en notación postfija, se utiliza otra pila, de elementos numéricos, para guardar los valores de los operandos y de las operaciones parciales con el fin de obtener el valor numérico de la expresión.​

**Transformación de una expresión infija a postfija​:**

Se parte de una expresión en notación infija que tiene operandos, operadores y puede tener paréntesis. ​

Los operandos vienen representados por letras, mientras los operadores son éstos:​

^ (potenciación), \*, /, +, - .​

La transformación se realiza utilizando una pila en la que se almacenan los operadores y los paréntesis izquierdos. ​

La expresión aritmética se lee del teclado y se procesa carácter a carácter.​

Los operandos pasan directamente a formar parte de la expresión en postfija, la cual se guarda en un array. ​

Un operador se mete en la pila si se cumple que:​

La pila esta vacía.​

• El operador tiene mayor prioridad que el operador cima de la pila.​

• El operador tiene igual prioridad que el operador cima de la pila y se trata de la máxima prioridad.​

Si la prioridad es menor o igual, se saca el elemento cima de la pila, se pone la expresión en postfija y se vuelve a hacer la comparación con el nuevo elemento cima.​

El paréntesis izquierdo siempre se mete en la pila; ya en 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 y ponerlos en 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 algoritmo termina cuando no hay más ítems de la expresión origen y la pila está vacía.​

Por ejemplo, la expresión a\*(b+c-(d/e^f)-g)-h está escrita en notación infija, la expresión equivalente en postfija se va a ir formando paso a paso:​

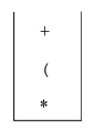
a Operando a pasa a la expresión postfija; operador \* a la pila.​

ab Operador ( pasa a la pila; operando b a la expresión.​

abc Operador + pasa a la pila; operando c a la expresión.​

En este punto, el estado de la pila es:​

El siguiente caracter de la expresión, -, tiene igual prioridad que el operador de la cima (+) y da lugar a:​

abc+​

abc+d El operador ( se mete en la pila; el operando d a la expresión.​

abc+de El operador / pasa a la pila; el operando e a la expresión.​

abc+def El operador ^ pasa a la pila; el operando f a la expresión.​

El siguiente ítem, ) (paréntesis derecho), produce que se vacíe la pila hasta un '('. La pila, en este momento, dispone de estos operadores:​

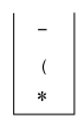


abc+def^/ El algoritmo saca operadores de la pila hasta un '(' y da lugar a la pila:​

abc+def^/- El operador – pasa a la pila y, a su vez, se extrae –; el siguiente carácter, el operando g, pasa a la expresión.​

abc+def^/-g El siguiente carácter es ')', por lo que son extraídos de la pila los operadores hasta un (, la pila queda :​

abc+def^/-g- El siguiente carácter es el operador -, hace que se saque de la pila el operador \* y se meta en la pila el operador –.​

**Algoritmo de paso de notación *infija*a *postfija***​:

* Los pasos por seguir para transformar una expresión algebraica de notación infija a postfija son:​
* 1. Obtener caracteres de la expresión y repetir los pasos 2 al 4 para cada carácter.​
* 2. Si es un operando, pasarlo a la expresión *postfija*.​
* 3. Si es operador:​
* 3.1. Si la pila está vacía, meterlo en la pila. Repetir a partir de 1.​
* 3.2. Si la pila no está vacía:​
* 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 de 1.​
* Si la prioridad del operador es menor o igual que la prioridad del operador de la cima de la pila, sacar el operador cima de la pila y pasarlo a la expresión postfija, volver a 3.
* ​4. Si es paréntesis derecho:​
* 4.1. Sacar el operador cima y pasarlo a la expresión postfija.​
* 4.2. Si el nuevo operador cima es paréntesis izquierdo, suprimir el elemento cima.​
* 4.3. Si la cima no es paréntesis izquierdo, volver a 4.1.​
* 4.4. Volver a partir de 1.​
* 5. Si quedan elementos en la pila, pasarlos a la expresión postfija.​
* 6. Fin del algoritmo.​
* Una vez que se tiene la expresión en notación postfija, se evalúa la expresión. ​
* En primer lugar, hay que dar valores numéricos a los operandos de la expresión. ​
* De nuevo, el algoritmo de evaluación utiliza una pila, en esta ocasión de operandos, es decir, de números reales.​
* Al describir el algoritmo, expresión es la cadena con la expresión *postfija*. ​
* El número de elementos es la longitud, n, de la cadena. ​
* Los pasos por seguir son los siguientes:​
* Examinar expresión elemento a elemento: repetir los pasos 2 y 3 para cada elemento.​
* 2. Si el elemento es un operando, meterlo en la pila.​
* 3. Si el elemento es un operador, se simboliza con &, entonces:​
* **•**Sacar los dos elementos superiores de la pila, que se denominarán b y a respectivamente.​
* **•**Evaluar a & b, el resultado es z = a & b.​
* **•**El resultado z, meterlo en la pila. 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 del algoritmo.​
* package evaluarExpresion;  
    
  import java.io.\*;  
    
  public class EjemploEvaluaExpresion {  
     
   public static void main(String[] a) {  
   double[] v = new double[26];  
   double resultado;  
   BufferedReader entrada = new BufferedReader(new InputStreamReader(System.in));  
   String expresion;  
   try {  
   System.out.print("\nExpresión aritmética: ");  
   expresion = entrada.readLine();  
     
   // Conversión de infija a postfija  
   expresion = ExprePostfija.postFija(expresion);  
   System.out.println("\nExpresión en postfija: " + expresion);  
     
   // Evaluación de la expresión  
   EvalPostfija.valorOprdos(expresion, v); // valor de operandos  
   resultado = EvalPostfija.evalua(expresion, v);  
   System.out.println("Resultado = " + resultado);  
   } catch (Exception e) {  
   System.out.println("\nError en el proceso ... " + e);  
   e.getStackTrace();  
   }  
   }  
  }

package evaluarExpresion;  
  
import java.io.\*;  
  
public class EvalPostfija {  
  
 public static double evalua(String postFija, double v[]) throws Exception {  
 PilaLista pila;  
 double valor = -1.0;  
 double a, b;  
 int i;  
  
 pila = new PilaLista();  
  
 for (i = 0; i < postFija.length(); i++) {  
 char op;  
 op = postFija.charAt(i);  
 if (ExprePostfija.operando(op)) { // es un operando  
 int indice;  
 indice = op - 'A';  
 pila.insertar(v[indice]);  
 } else { // es un operador  
 Double opCima;  
 opCima = (Double) pila.quitar();  
 b = opCima.doubleValue();  
 opCima = (Double) pila.quitar();  
 a = opCima.doubleValue();  
  
 switch (op) {  
 case '^':  
 valor = Math.pow(a, b);  
 break;  
 case '\*':  
 valor = a \* b;  
 break;  
 case '/':  
 if (b != 0.0)  
 valor = a / b;  
 else  
 throw new Exception("División por cero.");  
 break;  
 case '+':  
 valor = a + b;  
 break;  
 case '-':  
 valor = a - b;  
 }  
 pila.insertar(valor);  
 }  
 }  
 return ((Double) pila.quitar()).doubleValue();  
 }  
  
 public static void valorOprdos(String postFija, double v[]) throws Exception {  
 // asignan valores num�ricos a los operandos  
 int i;  
 BufferedReader entrada = new BufferedReader(new InputStreamReader(System.in));  
  
 for (i = 0; i < postFija.length(); i++) {  
 char op;  
  
 op = postFija.charAt(i);  
  
 if (ExprePostfija.operando(op)) { // es un operando  
 int indice;  
 Double d;  
  
 indice = op - 'A';  
 System.out.print(op + " = ");  
 d = Double.valueOf(entrada.readLine());  
 v[indice] = d.doubleValue();  
 }  
 }  
 }  
}

package evaluarExpresion;  
  
public class ExprePostfija {  
   
 public static String postFija(String expOrg) throws Exception {  
 PilaLista pila;  
 String e;  
 Character at;  
 char[] expsion;  
 char ch;  
 boolean desapila;  
 int n = 0; // contador de expresión en postfija  
  
 if (!valido(expOrg))  
 throw new Exception("Caracter no válido en una expresión");  
   
 e = expOrg.toUpperCase(); // operandos en mayúsculas  
 expsion = new char[e.length()];  
 pila = new PilaLista();  
  
 for (int i = 0; i < e.length(); i++) {  
 ch = e.charAt(i);  
 // análisis del elemento  
 if (operando(ch)) {  
 expsion[n++] = ch;  
 } else if (ch != ')') { // es un operador o un '('  
 desapila = true;  
 while (desapila) {  
 char opeCima = ' ';  
 if (!pila.pilaVacia()) {  
 at = (Character) pila.cimaPila();  
 opeCima = at.charValue();  
 }  
 if (pila.pilaVacia() || (prdadFuera(ch) > prdadDentro(opeCima))) {  
 pila.insertar(ch);  
 desapila = false;  
 } else if (prdadFuera(ch) <= prdadDentro(opeCima)) {  
 at = (Character) pila.quitar();  
 expsion[n++] = at.charValue();  
 }  
 }  
 } else { // es un ')'   
 at = (Character) pila.quitar();  
 do {  
 expsion[n++] = at.charValue();  
 at = (Character) pila.quitar();  
 } while (at.charValue() != '(');  
 }  
 }  
 /\*  
 \* se vuelca los operadores que quedan en la pila y se pasan a la expresión.  
 \*/  
 while (!pila.pilaVacia()) {  
 at = (Character) pila.quitar();  
 expsion[n++] = at.charValue();  
 }  
  
 return new String(expsion, 0, n); // expresión en postfija  
 }  
  
 private static int prdadDentro(char op) {  
 int pdp = -1;  
 switch (op) {  
 case '^':  
 pdp = 3;  
 break;  
 case '\*':  
 case '/':  
 pdp = 2;  
 break;  
 case '+':  
 case '-':  
 pdp = 1;  
 break;  
 case '(':  
 pdp = 0;  
 }  
 return pdp;  
 }  
  
 // prioridad del operador en la expresión infija  
 private static int prdadFuera(char op) {  
 int pfp = -1;  
 switch (op) {  
 case '^':  
 pfp = 4;  
 break;  
 case '\*':  
 case '/':  
 pfp = 2;  
 break;  
 case '+':  
 case '-':  
 pfp = 1;  
 break;  
 case '(':  
 pfp = 5;  
 }  
 return pfp;  
 }  
  
 private static boolean valido(String expresion) {  
 boolean sw = true;  
 for (int i = 0; (i < expresion.length()) && sw; i++) {  
 char c;  
 c = expresion.charAt(i);  
 sw = sw && ( (c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z')  
 || (c == '^' || c == '/' || c == '\*' || c == '+'   
 || c == '-' || c == '(' || c == ')') );  
 }  
 return sw;  
 }  
  
 public static boolean operando(char c) {  
 return (c >= 'A' && c <= 'Z');  
 }  
  
}

package evaluarExpresion;  
  
public class NodoPila {  
 Object elemento;  
 NodoPila siguiente;  
  
 NodoPila(Object x) {  
 elemento = x;  
 siguiente = null;  
 }  
}

package evaluarExpresion;  
  
public class PilaLista {  
 private NodoPila cima;  
  
 public PilaLista() {  
 cima = null;  
 }  
  
 public boolean pilaVacia() {  
 return cima == null;  
 }  
  
 public void insertar(Object elemento) {  
 NodoPila nuevo;  
 nuevo = new NodoPila(elemento);  
 nuevo.siguiente = cima;  
 cima = nuevo;  
 }  
  
 public Object quitar() throws Exception {  
 if (pilaVacia())  
 throw new Exception("Pila vacía, no se puede extraer.");  
 Object aux = cima.elemento;  
 cima = cima.siguiente;  
 return aux;  
 }  
  
 public Object cimaPila() throws Exception {  
 if (pilaVacia())  
 throw new Exception("Pila vacía, no se puede leer cima.");  
 return cima.elemento;  
 }  
  
 public void limpiarPila() {  
 NodoPila t;  
 while (!pilaVacia()) {  
 t = cima;  
 cima = cima.siguiente;  
 t.siguiente = null;  
 }  
 }  
}