Estructuras de Datos
Clase 5 – Aplicaciones de Pilas y Colas

Dr. Sergio A. Gómez
http://cs.uns.edu.ar/~sag

Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur
Bahía Blanca, Argentina

TDA Pila (Stack)

Pila: Colección de objetos actualizada usando una política LIFO (last-in first-out, el primero en entrar es el último en salir).

Operaciones:

- push(e): Inserta el elemento e en el tope de la pila
- pop(): Elimina el elemento del tope de la pila y lo entrega como resultado. Si se aplica a una pila vacía, produce una situación de error.
- isEmpty(): Retorna verdadero si la pila no contiene elementos y falso en caso contrario.
- top(): Retorna el elemento del tope de la pila. Si se aplica a una pila vacía, produce una situación de error.
- size(): Retorna un entero natural que indica cuántos elementos hay en la pila.

OJO: En la PC usar comentarios Javadoc.

```
public interface Stack<E> {
        // Inserta item en el tope de la pila.
        public void push( E item );

        // Retorna true si la pila está vacía y falso en caso contrario.
        public boolean isEmpty();

        // Elimina el elemento del tope de la pila y lo retorna.
        // Produce un error si la pila está vacía.
        public E pop() throws EmptyStackException;

        // Retorna el elemento del tope de la pila y lo retorna.
        // Produce un error si la pila está vacía.
        public E top() throws EmptyStackException;

        // Retorna la cantidad de elementos de la pila.
        public int size();
}
```

TDA Cola

Cola: Colección de objetos actualizada siguiendo una política FIFO (first-in first-out, el primero en entrar es el primero en salir)

Operaciones:

- enqueue(e): Pone el elemento e al final de la cola
- dequeue(): Elimina el elemento del frente de la cola y lo retorna. Si la cola está vacía se produce un error.
- front(): Retorna el elemento del frente de la cola. Si la cola está vacía se produce un error.
- isEmpty(): Retorna verdadero si la cola no tiene elementos y falso en caso contrario
- size(): Retorna la cantidad de elementos de la cola.

```
public interface Queue<E> {
        // Inserta el elemento e al final de la cola
        public void enqueue(E e);

        // Elimina el elemento del frente de la cola y lo retorna.
        // Si la cola está vacía se produce un error.
        public E dequeue() throws EmptyQueueException;

        // Retorna el elemento del frente de la cola.
        // Si la cola está vacía se produce un error.
        public E front() throws EmptyQueueException;

        // Retorna verdadero si la cola no tiene elementos
        // y falso en caso contrario
        public boolean isEmpty();

        // Retorna la cantidad de elementos de la cola.
        public int size();
}
```

Línea de comandos

- Se ingresan caracteres individuales que formarán el comando
- La tecla Backspace (Retroceso) borra el último carácter ingresado
- La tecla Escape borra todo el contenido de la línea de comandos
- La tecla Enter consolida el comando.

Línea de comando

```
P ← new Pila()
Leer carácter C de la consola
Mientras C <> Enter
        si C = Escape entonces
                mientras no P.isEmpty()
                        P.pop()
        sino
            si C = Retroceso
                si no P.isEmpty()
                        P.pop()
                sino
                        beep()
            sino
                P.push( c )
        Leer carácter C de la consola
S ← ""
Mientras no P.isEmpty() S ← P.pop() + S
Retornar S
```

<u>Tiempo de ejecución:</u>
Sea n=cantidad de caracteres del comando, entonces T(n) = O(n)

Análisis de la complejidad temporal

- De acuerdo a la estructura del código: si la cadena de entrada mide n caracteres, el while externo realiza n iteraciones y el while anidado otras n. Entonces, $T(n) = O(n^2)$.
- Sin embargo, viendo su semántica, supongamos que hay k escapes y $S = S_1 + Esc + S_2 + Esc + ... + S_k + Esc$ (donde "+" es el operador de concatenación de strings) con el largo de $S_1 = n_1$, largo de $S_2 = n_2$, …, largo de $S_k = n_k$. Vemos que $T(n) = (2n_1 + 1) + (2n_2 + 1) + … + (2n_k + 1) = O(2n)$. Luego $T(n) = O(n)$.

Formatos para expresiones aritméticas

- Infija: El operador va al medio de los operandos

$$E \rightarrow N \mid E_1 + E_2 \mid E_1 - E_2 \mid E_1 * E_2 \mid E_1 / E_2$$

$$N \rightarrow 0 \mid 1 \mid \ldots \mid 9$$

- Prefija: El operador va delante de los operandos (notación polaca)

$$E \rightarrow N \mid + E_1 E_2 \mid - E_1 E_2 \mid * E_1 E_2 \mid / E_1 E_2$$

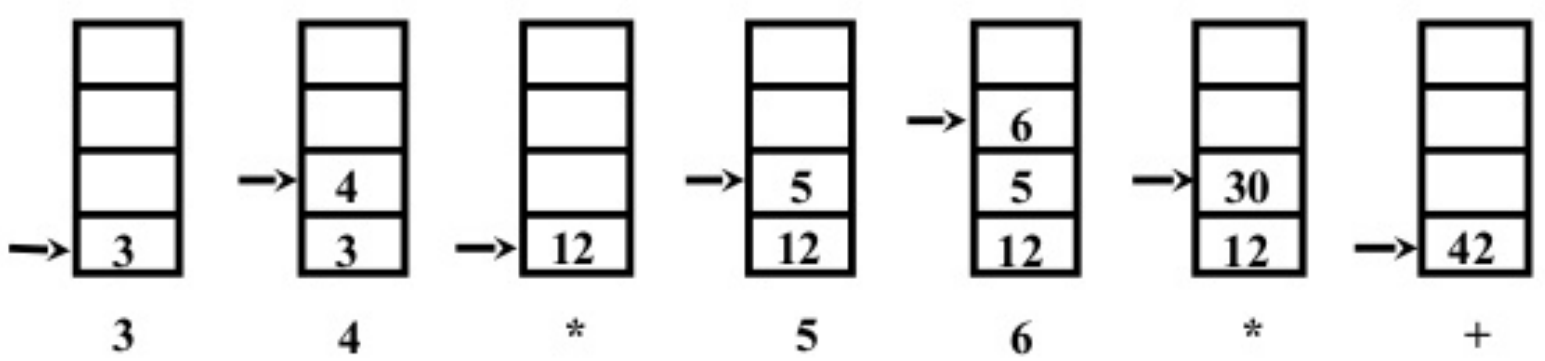
- Posfija: El operador va detrás de los operandos (notación polaca inversa)

$$E \rightarrow N \mid E_1 E_2 + \mid E_1 E_2 - \mid E_1 E_2 * \mid E_1 E_2 /$$

Infija	Prefija	Postfija
3	3	3
3+4	+ 3 4	3 4 +
7 * 5	* 7 5	7 5 *
(3+4) * 5	* + 3 4 5	3 4 + 5 *
((3+4)*5) – 9	- * + 3 4 5 9	3 4 + 5 * 9 -
(((3+4)*5) – 9) / 4	/ - * + 3 4 5 9 4	3 4 + 5 * 9 – 4 /
(3*4) + (5*6)	+ * 3 4 * 5 6	3 4 * 5 6 * +

Cómo evaluar expresiones en notación polaca inversa

$$(3 * 4) + (5 * 6) \quad \Rightarrow \quad 3\ 4\ *\ 5\ 6\ *\ +$$



Dada una expresión en notación postfija, se comienza con una pila vacía. Se consumen los símbolos de a uno por vez. Los operandos se apilan. Con cada operador se desapilan dos operandos, se aplica el operador y se vuelve a apilar el resultado. Si al finalizar la expresión, la pila tiene un elemento, ese elemento es el resultado de la expresión. Si la pila tiene más de un elemento, la expresión estaba incorrectamente formada (lo mismo si al tratar de aplicar un operador no hay por lo menos dos elementos en la pila).

Evaluación de expresiones

Solución:

- Entrada: Expresión aritmética infija válida con operadores +, -, *, / (binarios con * y / con mayor precedencia que + y -) y sin paréntesis
- Paso 1: Traducir expresión infija en postfija
- Paso 2: Evaluar expresión postfija

Paso 1: Paso de notación infija a postfija

- Entrada: *ExpInf* es una cola que contiene una expresión válida en notación infija, sin parentizar y con operadores aditivos (+, -) y multiplicativos (*,/).
- Salida: *ExpPost* es una cola que al finalizar queda con la expresión en notación postfija
- Se usa una pila auxiliar de operadores según el esquema que se detalla en la próxima diapositiva.
- Luego de retirar el último elemento de la cola *ExpInf*, se desapila todo lo que resta y se lo ingresa en la cola *ExpPost*.

Paso de notación infija a postfija

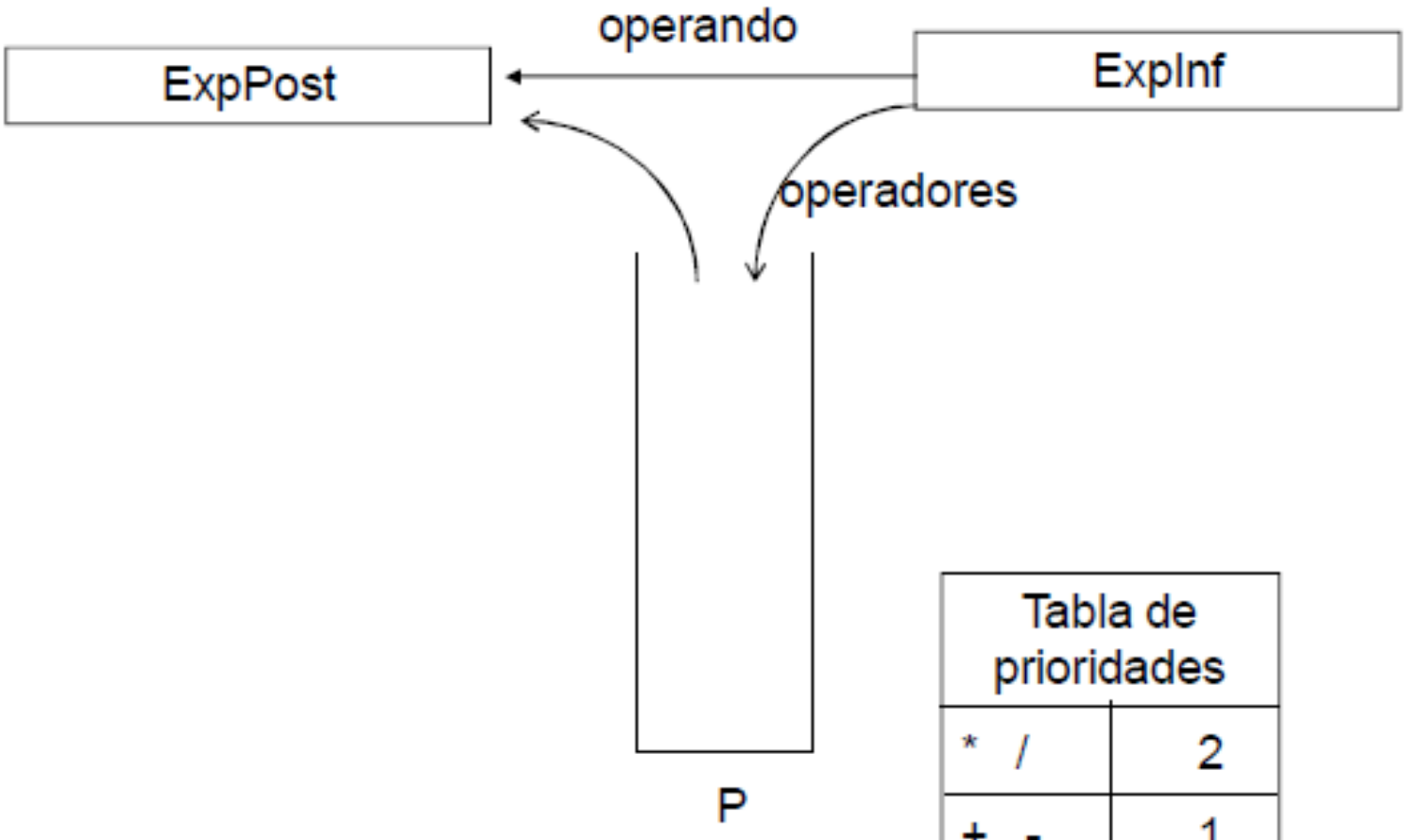


Tabla de prioridades	
* /	2
+ -	1

Paso de notación infija a postfija

```
ExpPost ← new Cola()
P ← new Pila()
Repetir
    átomo ← ExpInf.dequeue()
    si átomo es operando entonces
            ExpPost.enqueue( átomo )
    sino            // átomo es un operador
            Mientras (no P.isEmpty()) Y
                (prioridad(átomo) <= prioridad(P.top()) )
                    ExpPost.enqueue( P.pop() )
            P.push( átomo )
Hasta ExpInf.isEmpty()

Mientras no P.isEmpty()
        ExpPost.enqueue( P.pop() )
```

Paso 2: Evaluación de expresión válida en notación postfija

```
P ← new Pila()
Repetir
   átomo ← E.dequeue()
   si átomo es operando entonces
         P.push( átomo )
   sino   // átomo es un operador
         operador ← átomo
         operando2 ← P.pop()
         operando1 ← P.pop()
         resultado ← operar( operador, operando1, operando2 )
         P.push( resultado )
Hasta E.isEmpty()

Resultado ← P.pop()
Retornar resultado
```

Nota: operando2 sale primero en caso en que el operador no sea conmutativo (como – y /)

Validación de código HTML

- Página web: Archivo de texto con extensión HTML conteniendo texto con marcas (*tagged text*) y texto sin marcas (*untagged text*).
- Un tal archivo es interpretado por un navegador web y el texto sin marcas es formateado de acuerdo al texto con marcas para su presentación en la pantalla del navegador.
- Marcas:
  - De apertura: <html>, <head>, , …
  - De cierre: </html>, </head>, , …
  - Simplificación del problema:
    - No consideraremos marcas con atributos:

    e.g
    - No consideraremos marcas únicas:

    e.g.

Mini curso de HTML

- Página HTML = <html>Encabezado Cuerpo</html>
- Encabezado = <head> …. </head>
- Cuerpo = <body> … </body>
- Contenido del encabezado: <title>…</title> (otras marcas incluyen detalle del idioma, palabras clave para los buscadores como Google que no consideraremos)

Mini curso de HTML

- Contenido del cuerpo:
  - Encabezados: <h1>…</h1>, <h2>…</h2>, <h3> … </h3>
  - Formato de texto:
    - Cursiva (italics): texto en itálica
    - Negrita (bold): texto en negrita
    - Subrayado (underlined): <u>texto subrayado</u>
  - Párrafos: …..
  - Listas con viñetas (unordered list):

```
<ul>
        <li>Primer item</li>    (li = list item)
        <li>Segundo item</li>
</ul>
```

Mini curso de HTML

- Contenido del cuerpo:

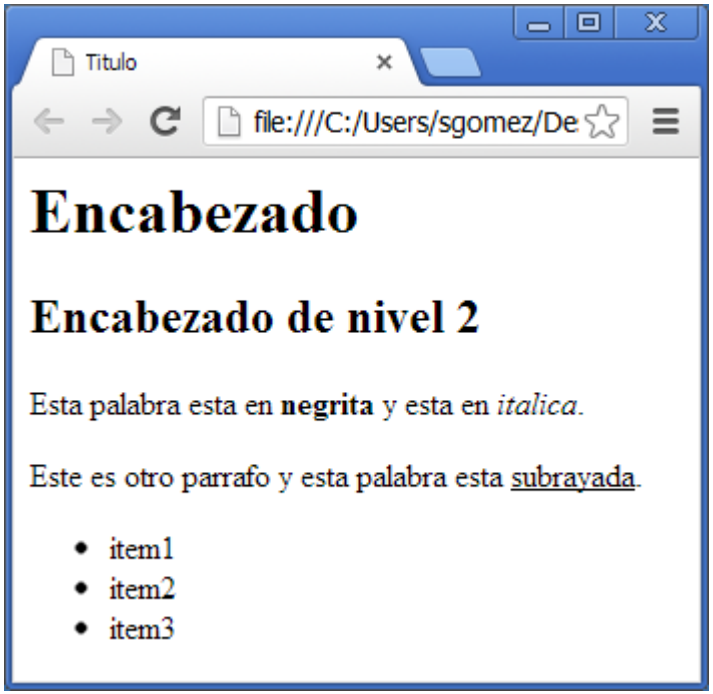
  Listas con enumeraciones (ordered list):

```
<ol>
        <li>Primer item</li>
        <li>Segundo item</li>
</ol>
```

  Tablas: (table = tabla, tr = table row, td = table datum)

```
<table>
        <tr>  <td>a</td>   <td>b</td>   </tr>
        <tr>  <td>c</td>   <td>d</td>   </tr>
</table>
```

```
<html>
<head><title>Titulo</title></head>
<body>
    <h1>Encabezado</h1>
    <h2>Encabezado de nivel 2</h2>
    <p>Esta palabra esta en <b>negrita</b>
    y esta en <i>italica</i>.</p>
    <p>Este es otro parrafo
    y esta palabra esta <u>subrayada</u>.</p>
    <ul>
        <li>item1</li>
        <li>item2</li>
        <li>item3</li>
    </ul>
</body>
</html>
```



```
<html>
<head><title>Titulo</title></head>
<body>
    <h1>Encabezado</h1>
    <h2>Encabezado de nivel 2</h2>
    <p>Esta palabra esta en <b>negrita</b>
    y esta en <i>italica</i>.</p>
    <p>Este es otro parrafo
    y esta palabra esta <u>subrayada</u>.</p>
    <ul>
        <li>item1</li>
        <li>item2</li>
        <li>item3</li>
    </ul>
</body>
</html>
```

Cola de tokens: “<html>”, “<head>”, “<title>”, Titulo, “</title>”, “</head>”, “<body>”, “<h1>”, “Encabezado”, “</h1>”, “<h2>”, “Encabezado”, “de”, “nivel”, “2”, “</h2>”, …

Algoritmo Validar Archivo HTML

Entrada: Archivo HTML H
Salida: true si el archivo es válido y false en caso contrario
Pila_tags ← new Pila<String>()
Cola_Tokens ← obtener cola de tokens a partir de archivo HTML H
Es_Valido ← true
Mientras NOT Cola_Tokens.isEmpty() AND Es_Valido
    Token ← Cola_Tokens.dequeue()
    si Token es una marca de apertura entonces
        Pila_Tags.push(Token)
    sino
        si Token es una marca de cierre entonces
            Si NOT Pila_tags.isEmpty() ANDToken coincide con Pila_tags.top() entonces
                Pila_tags.pop()
            sino
                Es_valido ← false

Si no Pila_tags.isEmpty() entonces Es_valido ← false
Retornar Es_valido

Invertir el contenido de una pila

```
public class PilaEnlazada<E> implements Stack<E> {
        protected Nodo<E> tope;
        protected int size;
        …
}
```

Tres opciones de implementación:

(1) Escribir una clase *InvertidorDePila* que reciba una pila y retorne la misma pila pero con su contenido invertido.

(2) Escribir una subclase de *PilaEnlazada* llamada *PilaEnlazadaInvertible* que extienda la clase con una operación llamada invertir()

a) "invertir()" se puede programar en términos de las operaciones de pila, o,

b) "invertir()" se puede programar con acceso a la estructura de datos subyacente

Invertir pila: Solución (1)

```
public class Principal {
   public static void main( String [] args )
   {
      try {
            PilaEnlazada<Character> p = new PilaEnlazada<Character>();
            p.push( 'a' );   p.push( 'b' );     p.push( 'c' );

            InvertidorDePila inv = new InvertidorDePila();
            inv.invertirPila( p );

            while (!p.isEmpty() )
                  System.out.println( "Elemento: " + p.pop() );

      } catch( EmptyStackException e ) {
            System.out.println( "Error: "  + e.getMessage() );
      }
   }
}
```

Invertir Pila: Solución (1) (cont.)

```
public class InvertidorDePila {
   // Pasa el contenido de pila p1 a pila p2.
   private <E> void pasar( Stack<E> p1, Stack<E> p2 )   {
        try {
             while ( !p1.isEmpty() )     p2.push(p1.pop());
        } catch(EmptyStackException e) { e.printStackTrace(); }
   }

  // Invierte el contenido de la pila unaPila.
   public <E> void invertirPila(Stack<E> unaPila) {
             Stack<E> otraPila1, otraPila2;
             otraPila1 = new PilaEnlazada<E>();
             otraPila2 = new PilaEnlazada<E>();
             pasar( unaPila, otraPila1 );
             pasar( otraPila1, otraPila2 );
             pasar( otraPila2, unaPila );
    }
}
```

Invertir Pila: Solución (2.a)

Otra solución consiste en extender la funcionalidad del TDA con la operación *invertir().*
Esto lo modelamos extendiendo la interfaz Stack con una nueva operación.
Tal Nuevo TDA lo denominamos *StackInvertible*.

```
public interface StackInvertible<E> extends Stack<E>
{
        // Invierte el contenido del receptor.
        public void invertir();
}
```

Implementaremos una clase *PilaEnlazadaInvertible* que implementa la interfaz *StackInvertible* pero reutilizando la implementación que ya poseemos de *Stack* (i.e. *PilaEnlazada*):

```
public class PilaEnlazadaInvertible<E> extends PilaEnlazada<E>
                                        implements StackInvertible<E> {
private void pasar( Stack<E> p1, Stack<E> p2 ) {
        // Pasa el contenido de p1 a p2
        try {
            while( !p1.isEmpty() ) p2.push(p1.pop());
        } catch( EmptyStackException e ) { e.printStackTrace(); }
}

public void invertir() {    // Invierte el contenido del receptor del mensaje
        Stack<E> otraPila1, otraPila2;
        otraPila1 = new PilaEnlazada<E>();
        otraPila2 = new PilaEnlazada<E>();
        pasar( this, otraPila1 );
        pasar( otraPila1, otraPila2 );
        pasar( otraPila2, this );
}
}
```

Invertir Pila: Solución 2.b

En esta aproximación, implementamos la operación invertir() de PilaEnlazadaInvertible usando acceso al estado interno de la Pila modificando directamente la estructura de nodos enlazados:

```
public void invertir(){
        if( head != null ) {
                Nodo<E> p = head.getSiguiente();
                head.setSiguiente( null );
                while( p != null ) {
                        Nodo<E> q = p.getSiguiente();
                        p.setSiguiente( head );
                        head = p;
                        p = q;
                }
        }
}
```

NOTA: Es la solución más eficiente en utilización de memoria pero es por lejos la menos legible.

Pilas nativas en Java

- Java brinda la clase java.util.Stack<E> (la cual es subclase de java.util.Vector<E>)

Constructor Summary
`Stack()` Creates an empty Stack.

Method Summary	
`boolean`	`empty()` Tests if this stack is empty.
`E`	`peek()` Looks at the object at the top of this stack without removing it from the stack.
`E`	`pop()` Removes the object at the top of this stack and returns that object as the value of this function.
`E`	`push(E item)` Pushes an item onto the top of this stack.
`int`	`search(Object o)` Returns the 1-based position where an object is on this stack.

```
import java.util.Stack;

public class TestStackJava {
        public static void main( String [] args ) {
                Stack<Integer> s = new Stack<Integer>();

                s.push( 1 );
                s.push( 2 );
                s.push( 3 );
                s.push( 4 );
                if (!s.empty() ) System.out.println( "tope: " + s.peek() );
                while (!s.empty() )
                        System.out.println( "elem: " + s.pop() );
        }
}
```