***TP - Unidad 03-B***

***Implementaciones y Casos de Uso: Pila y Cola***

***Ejercicio 1***

* 1. Analizar las ventajas y desventajas de implementar una Pila con arreglos y con lista lineales simplemente encadenadas. Explicar dónde conviene colocar el elemento distinguido y por qué. Calcular en cada caso la complejidad espacial y temporal de cada operación.
  2. Analizar cómo es el diseño e implementación de Pila que se encuentra en Java 8 desde el punto de vista de OOP.
  3. Analizar las ventajas y desventajas de implementar una Cola con arreglos y con lista lineales simplemente encadenadas. Explicar dónde conviene colocar el elemento distinguido y por qué. Calcular en cada caso la complejidad espacial y temporal de cada operación.
  4. Analizar cómo es el diseño e implementación de Cola que se encuentra en Java 8 desde el punto de vista de OOP.

***Ejercicio 2***

Dadas las siguientes expresiones en notación postfija, formada por los operadores binarios: +, -, \*, / se pide mostrar paso a paso cómo queda el input y la pila asociada luego de consumir cada token y hasta que se obtiene el resultado numérico final.

2.1) **3 10 + 2 - 5 4 \* -**

2.2) 2 -0.1 + 10 2 \* /

2.3) -9 -1 - 10 2 \* / 1 5 - 2 -3 / / \*

***Ejercicio 3***

Para poder implementar el evaluador en Java que toma un input en notación postfija y devuelve el resultado numérico final, se debe usar alguna biblioteca que separe en tokens.

3.1) Usando la clase Scanner de Java se pueden leer tokens. El siguiente código utiliza 2 scanners. Uno para leer input de un usuario que finaliza con el carácter \n y otro que analiza tokens del string leído.

**public static void main(String[] args) {**

System.**out.print("Introduzca la expresión en notación postfija: ");**

Scanner inputScanner = **new Scanner(System.in).useDelimiter("\\n");**

Scanner **lineScanner** = **new Scanner(line).useDelimiter("\\s+");**

**while(lineScanner.hasNext())**

System.**out.println(lineScanner.next());**

}

Basado en esa idea, modificarlo para que valide que los únicos tokens que acepta son esto cuatro: **"**¡!**", "**,**", "**;**", "**##**".** Usar la función **matches()** la cual permite evaluar expresiones regulares.

Chequear correctitud. Por ejemplo, el input "!¡" es erróneo. También lo es "¡ !".

3.2) Modificar el código anterior para que acepte un token más: **"**¿?**".** Explicar por qué hay que escapar ciertos símbolos.

3.3) Con las ideas anteriores, implementar la clase **Evaluator**. La misma debe leer de la entrada estándar una expresión en notación postfija conteniendo constantes numéricas en notación punto flotante (el menos pegado al número) y los operadores +, -, \*, /.

La misma debe devolver el valor resultante.

Caso de Uso:

**Double rta = new Evaluator().evaluate();**

Si se ingresa 2 -0.1 + 10 2 \* / debe devolver 0.095

Si se ingresa -9 -1 - 10 2 \* / 1 5 - 2 -3 / / \* debe devolver -2.4

3.4) generar testeos de unidad para el evaluador. No omitir el chequeo de Excepciones en el caso en que la expresión sea incorrecta con el cartel lo más preciso posible. Ej: si se ingresa 3 - deberá indicar que faltan operandos. Ej: si se ingresa -3.2 4 5 - deberá indicar que faltan operadores. Ej: si se ingresa -3 4.5 & deberá indicar que el operador es inválido.

Tip: para chequear input de la entrada estándar, deberán redireccionarla.

Por ejemplo:

@Test

**public** **void** test1() {

// inyecto entrada estandard

String input= "2 -0.1 + 10 2 \* /";

InputStream inputstream= **new** ByteArrayInputStream(input.getBytes());

System.*setIn*(inputstream);

Double rta = **new** Evaluator().evaluate();

*assertEquals*(0.095, rta);

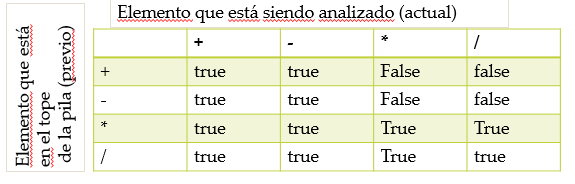
System.*setIn*(System.***in***);

}

***Ejercicio 4***

Para no solicitar al usuario que ingrese una expresión en notación postfija (a la cual probablemente no esté acostumbrado) se puede proceder de la siguiente forma: aceptar como entrada una expresión en notación infija, aplicar un parser de precedencia de operadores para realizar la transformación esperada y finalmente invocar al evaluador.

Para una expresión infija que no acepta paréntesis, sus operandos son números en notación punto flotante y los operadores son los 4 explicados anteriormente, el parser precisa de la siguiente tabla de precedencia, la cual está diseñada por la precedencia de los operadores y en el caso de ambigüedad, por la asociatividad.



Dadas las siguientes expresiones en notación infija se pide mostrar paso a paso cómo queda el input y la pila asociada luego de consumir cada token hasta que finaliza el input y se genera el output que usaría el método evaluate().

4.1) 2.3 - 1.5 \* 4 - 3

4.2) 3 + 10 \* 2 / 1

***Ejercicio 5***

5.1) Analizar por lo menos dos formas diferentes de representar en Java la tabla con la precedencia.

5.2) Agregar el método **private String infijaToPostfija()** en la clase Evaluator para implementar el parser de precedencia que utiliza la tabla antes diseñada.

La biblioteca interactiva será utilizada así:

Evaluator e = **new Evaluator();**

System.**out.println(e.evaluate());**

System.**out.println(e.evaluate());**

System.**out.println(e.evaluate());**

En este caso, se solicita 3 veces que se ingrese una expresión en notación infija y se devolvuelve el valor de la evaluación, el cual se imprime.

Internamente**, evaluate()** invoca al método **infijaToPostfija()** lee input y devuelve un String con la transformación. Luego evaluate() construye un nuevo escáner sobre el string recibido y este se encarga de evaluar la expresión (lo realizado previamente).

**public** **class** Evaluator {

**public** **double** evaluate()

{

System.***out***.print("Introduzca la expresión en notación infija: ");

Scanner postfijaScanner= new Scanner(infijaToPostfija() );

**if** (! postfijaScanner.hasNext())

**throw** **new** RuntimeException("no hay nada para evaluar");

// bla bla bla, es decir básicamente lo del ejercicio 3, pero sin

// pedir el cartel de ingreso de datos

// retorna el valor double de la expresión infija original

return valor;

}

**private** String infijaToPostfija()

{

Scanner s = **new** Scanner(System.***in***).useDelimiter("\\n");

String ss= s.nextLine();

Scanner scannerLine = **new** Scanner(ss).useDelimiter("\\s+");

// bla bla bla. Lee tokens de infija y transforma a postfija

// retorna un string en notación postfija

return …;

}

}

Ejemplo:

2 - 3 \* -3 devuelve 11

2 / 4 / 2 devuelve 0.25

5.3) Chequear correctitud.

Tip: para inyectar un input desde testeo de unidad, podemos hacerlo asi:

@Test

**void** test() {

// inyecto en la estandard input

String input = "15 + 3";

InputStream inputStream = **new** ByteArrayInputStream(input.getBytes());

System.*setIn*(inputStream);

Evaluator myEval = **new** Evaluator();

**double** rta= myEval.evaluate();

*assertEquals*( 18, rta);

}

Tip: los método private pueden ser chequeados con Reflection

Si tenemos una clase con 2 métodos private:

**public** **class** Sorpresa {

**private** **double** f()

{

**return** 35;

}

**private** **double** f(**double** param)

{

**return** param;

}

}

En los testeos de unidad, podemos hacer el chequeo de los métodos private, así:

**class** Testing {

@Test

**void** test1() **throws** NoSuchMethodException, SecurityException,

IllegalAccessException, IllegalArgumentException,

InvocationTargetException {

Sorpresa sorpresaInstance = **new** Sorpresa();

Method myMethod

= Sorpresa.**class**.getDeclaredMethod( "f", **double**.**class**);

myMethod.setAccessible(**true**);

**double** result

= (Double) myMethod.invoke(sorpresaInstance, 34.5);

*assertEquals*(34.5, result);

}

@Test

**void** test2() **throws** NoSuchMethodException, SecurityException,

IllegalAccessException, IllegalArgumentException,

InvocationTargetException {

Sorpresa sopresaInstance = **new** Sorpresa();

Method myMethod

= Sorpresa.**class**.getDeclaredMethod( "f");

myMethod.setAccessible(**true**);

**double** result

= (Double) myMethod.invoke(sopresaInstance);

*assertEquals*(35, result);

}

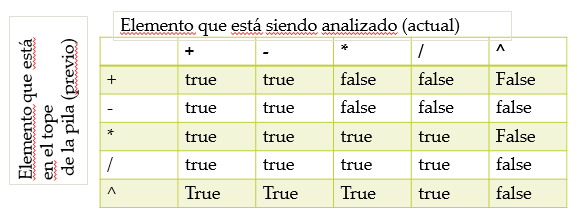
}

**Ejercicio 6**

Incorporar el operador exponenciación sabiendo que tiene mayor precedencia que todos los demás, pero es asociativo a derecha.

<https://en.wikipedia.org/wiki/Order_of_operations>

Así, la tabla queda:



Ejemplo:

2 - 3 ^ -3 devuelve 1.9629

2 ^ 4 ^ 2 devuelve 65536 ( y no 256)

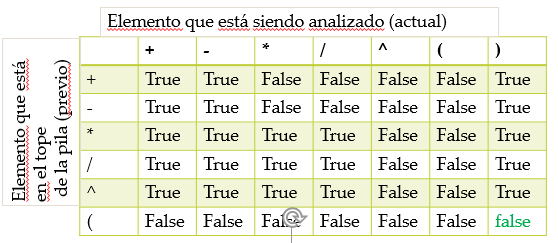
3 + 10 \* 2 / 1 (toPostfija() da 3 10 2 \* 1 / + ) y evalua a 23

13 ^ 2 - 5 \* 7 (toPostfija() da 13 2 ^ 5 7 \* - ) y evalúa a 134

5 ^ 2 ^ 3 - 1 (toPostfija() da 5 2 3 ^ ^ 1 - ) y evalua a 390624

**Ejercicio 7**

Para incorporar paréntesis en las expresiones, conviene tratarlas como un operador más. Obviamente los paréntesis no se colocarán en la salida, porque no existen en notación postfija. Sin embargo, servirán para postergar acciones. La tabla queda así:



Dadas las siguientes expresiones en notación infija con paréntesis se pide mostrar paso a paso cómo queda el input y la pila asociada luego de consumir cada token hasta que finaliza el input y se genera el output en postfija

* 1. **( 3 + 10 ) ^ 2 - 5 \* 7**
  2. 3 \* ( ( 5 - 10.2 ) / 0.5 ) - 2
  3. 3 \* ( ( 5 - 10.2 / 0.5 ) -
  4. 3 \* ( 5 - 10.2 ) / 0.5 ) - 2

**Ejercicio 8**

Agregar a la clase Java Evaluate el manejo de paréntesis en las expresiones infijas, asegurándose que no se generan en las expresiones postfijas auxiliares. Realizar chequeos para verificar correctitud.

En estos casos, se obtiene:

* 1. **( 3 + 10 ) ^ 2 - 5 \* 7 rta: 134**
  2. 3 \* ( ( 5 - 10.2 ) / 0.5 ) - 2 rta: -33.19999
  3. 3 \* ( ( 5 - 10.2 / 0.5 ) - rta: error falta )
  4. 3 \* ( 5 - 10.2 ) / 0.5 ) - 2 rta: error falta (

**Ejercicio 9**

Incorporar la evaluacación de expresiones que contengan variables. Los valores de las mismas serán proporcionadas en el constructor y estarán disponibles para todas las evaluacaiones.

Caso de uso:

EvaluatorConParentesisVariables x = **new** EvaluatorConParentesisVariables(

**new** HashMap<String, Double>()

{ { put("v1", 12.3);

put("v3", 5.0);

} } );

System.***out***.println(x.evaluate());

System.***out***.println(x.evaluate());

System.***out***.println(x.evaluate());

Y las expresiones pueden contener v1, v3, números y operadores (incluido paréntesis).

**Ejercicio 10**

Implementar un bounded queue por medio de un arreglo estático (no usar las clases que vienen en Java ArrayList o Vector). En la creación de proporciona la máxima cantidad de elementos que podrá contener. No realoca dinámicamente, es decir, si se encola más allá del tamaño reservado lanza excepción. Debe “aprovechar” al máximo el espacio reservado en la creación.

Caso de Uso

BoundedQueue<Integer> myQueue = **new BoundedQueue<>(10);**

myQueue.enqueue(10);

myQueue.enqueue(20);

myQueue.enqueue(30);

myQueue.enqueue(40);

System.***out.println(myQueue.dequeue() );***

System.***out.println(myQueue.dequeue() );***

myQueue.enqueue(50);

myQueue.enqueue(60);

myQueue.enqueue(70);

System.***out.println("\nquedaron 5 elementos");***

myQueue.dump();