Estructuras de Datos

Paralelo 1 – PAO I 2020

Tarea TDA Pila

Introducción

Como revisamos en la clase anterior, el TDA pila es una colección homogénea de objetos genéricos que casi se comporta como una lista, pero con una restricción: los elementos añadidos a una pila –o removidos de ésta– siempre se añaden (o se remueven) por un mismo extremo. Ese extremo se conoce como el **tope** de la pila, el mismo que cambia a medida que nuevas inserciones o remociones de elementos tienen lugar.

La pila tiene dos operaciones principales: el método `push(E e)` inserta un elemento a la vez (el mismo que se convierte en el tope de la pila) y el método `pop()` remueve el tope actual de la pila y lo retorna. Otros métodos permiten averiguar si la pila está vacía (`isEmpty()`) y conocer su tamaño (`size()`). Finalmente, el método `peek()` permite conocer qué elemento está en el tope de la pila, pero **sin removerlo** de la misma.

Como estructura de dato, la pila permite lograr un comportamiento *last-in, first-out* (LIFO). Esta filosofía de "*el primero en entrar es el último en salir*" (o "*el último en entrar es el primero en salir*") es ideal en la solución de problemas donde se requiere *recordar* o llevar un *historial de operaciones*. Dado que lo último en ser insertado es siempre lo primero en salir, el tope de la pila siempre contiene el elemento más *reciente* (lo último que se recordó).

Ejemplo: En una pila de correspondencia la carta más reciente se ubica en el tope. Cuando se modifica la pila ejecutando el método `pop()` varias veces, se logra procesar las cartas en orden inverso al que éstas llegaron.

Tope: carta más reciente

En esta práctica, usted hará uso de las clases de Java del paquete java.util:

Stack (https://docs.oracle.com/javase/7/docs/api/java/util/Stack.html) ,
ArrayDeque (https://docs.oracle.com/javase/7/docs/api/java/util/ArrayDeque.html), o
LinkedList (https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html)

que implementan, respectivamente, el TDA pila en versión estática (Stack y ArrayDeque) y dinámica (LinkedList). Estas clases son parametrizables por tipo. El ejemplo mostrado a continuación inicializa dos pilas estáticas de números enteros y una pila dinámica de cadenas de caracteres:

```
Stack<Integer> stack1 = new Stack<>();
Deque <Integer> stack2 = new ArrayDeque <>();
Deque <String> stack3 = new LinkedList <>();
```

Para los propósitos de este curso, estas tres clases son equivalentes. Sin embargo, es importante que usted conozca las diferencias entre ellas que discutimos la clase pasada.

Tarea

Esta tarea consiste de dos ejercicios:

1. Conversión de expresiones infijas a posfijas

Escriba un método estático que, dada una cadena de caracteres de números y operadores aritméticos en notación infija, retorne la expresión en notación posfija.
Ejemplo: Si el método recibe la expresión `21 + 13 * 16 – 18`, debe retornar `21 13 16 * + 18 –`

2. Evaluación de expresiones posfijas

Escriba un método estático que, dada una cadena de caracteres que contenga números y operadores aritméticos en formato posfijo, retorne el valor de la expresión.

Su método debe evaluar, por ejemplo, la cadena `6 12 3 + - 31 18 27 / + * 2 ^ 3 +`
Ejemplo: Si su método recibe la expresión `13 16 +`, debe retornar un el número 29.

Tenga en cuenta que los operandos de la cadena pueden ser números de más de un dígito. Por esta razón, en el ejemplo anterior, la expresión separa a sus componentes con un espacio en blanco. En la práctica, usted puede separar estos componentes en un arreglo a través del método `split` de la clase `String`. A continuación, se muestra un ejemplo:

```
String expression = "15 34 + 28 *";
String[] parts = expression.split(" ");
// al iterar el arreglo parts, se accede a cada elemento de la expresión
for (String part : parts) {
      // procesar cada parte
}
```

3. Probando sus métodos

Pruebe sus soluciones de los puntos 1 y 2 en un programa principal, dentro de una clase llamada `Expressions`.

4. Conversión Infija a Posfija con paréntesis (OPCIONAL)

El algoritmo del punto 1 de esta práctica es válido solo para convertir expresiones infijas que no contienen paréntesis. Para resolver esta limitación, podemos implementar un algoritmo siguiendo las siguientes reglas:

Los paréntesis izquierdos (:
    Siempre van a ser añadidos a la pila, pase lo que pase
Los paréntesis derechos):
    Significa que un ambiente de paréntesis ha terminado,
    Por tanto, todos los operadores de la pila se sacan hasta encontrar un (

Implemente un método estático adicional que permita convertir expresiones infijas que contienen paréntesis a su equivalente postfijo. Su método debe convertir, por ejemplo, la expresión `((A-(B+C))*D^(E+F)` a `ABC+-D*EF+^`.