

Estructura de Datos y Algoritmos

ITBA 2024-Q2

Solución a otros problemas

Más allá de la problemática de los “índices”, existen otros problemas que requieren de estructuras de datos sencillas (que podrían implementarse con un arreglo o una lista).

Analicemos algunos de esos problemas.

Orden de llegada...

El concepto de búsqueda de un elemento nos llevó a la idea de precisar un índice para facilitar la búsqueda.

Pero si no precisamos “buscar” elementos? Es más, si ni siquiera precisamos compararlos entre sí?

Existe otro “orden” de elementos que tienen que ver con el momento en que se generan los datos o su orden de llegada.

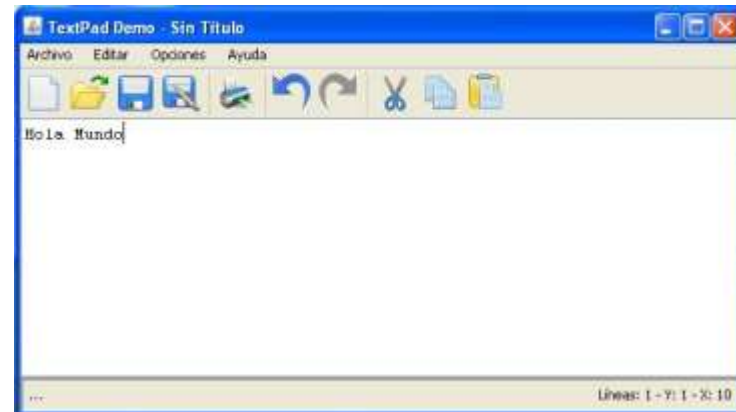
Problemas

Ejemplo 1: Editores

Los editores de texto permiten realizar operaciones: copy, paste, move, etc. y afortunadamente los avanzados permiten “deshacer” las últimas acciones realizadas. Permiten “arrepentirse” e inspeccionar la herramienta.

¿Qué estructura auxiliar nos permite implementar esta característica?

Rta: Pila (Stack o LIFO)



Problemas

Ejemplo 2: Runtime execution

Cuando en tiempo de ejecución se invoca a un método se utiliza el stack del runtime para almacenar: parámetros, variables locales y dirección de retorno. Esto también pasa en recursión.

Existen algoritmos que son ideales para programar en forma recursiva.



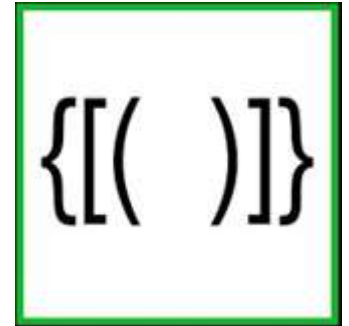
Si un lenguaje de programación no dispone de recursión, y el algoritmo es de naturaleza recursiva. ¿Qué podemos hacer?

Rta: usar un Stack para solucionar esa dificultad.

Ejemplo: Chequeos de compilación

Los lenguajes precisan de ciertos chequeos sintácticos. Tal es el caso de las expresiones matemáticas. En ellas, los operadores que utilizamos presentan diferente “precedencia” la cual puede cambiarse con tal de usar “paréntesis”.

Si bien los paréntesis pueden anidarse todo lo que querramos, deben aparearse correctamente.



¿Cómo pueden resolverse los chequeos sintácticos de ese tipo?

Rta: Con una PILA

Stack

Definición Stack/LIFO/Pila

Colección de datos ordenada por orden de llegada. La única forma de acceso es por medio de un elemento distinguido: TOPE que es el ultimo elemento que llegó.

Las operaciones que debe ofrecer son:

- **push**: agrega un elemento a la colección y se convierte en el nuevo tope.
- **pop**: quita un elemento de la colección y cambia el tope de la pila. Es una operación destructiva y solo puede usarse si la colección no está vacía.
- **isEmpty**: devuelve true/false segun la colección tenga o no elementos
- **peek**: devuelve el elemento tope pero sin removerlo. Es una operación de lectura y solo puede usarse si la colección no está vacía.

Stack: su implementación

¿Puede implementarse con un arreglo? ¿Con una lista lineal simplemente encadenada? ¿Tiene sentido usar una lista lineal doblemente encadenada?

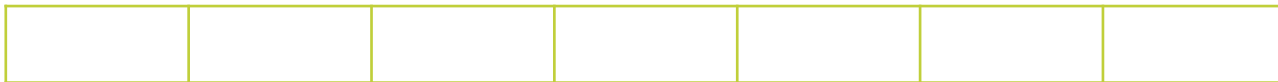
Stack

- Si se lo implementara con un arreglo. ¿Tenemos el problema de “movimiento de datos”?

Rta: la contigüedad está garantizada. Nunca quedarían huecos internos y no haría falta mover elementos para garantizar contigüidad ya que las operaciones solo se acceden por el TOPE.

Sin embargo, si se acaba el espacio sí debemos buscar espacio contiguo en otro lugar y copiar componentes.

Es decir, nos conviene hacer crecer/decrecer la estructura de a “chunks”.



Stack

- Si se lo implementara con una lista lineal simplemente encadenada. ¿Tenemos el problema de navegación?

Rta: No. Los elementos en el Stack solo se acceden por el tope. Es solo cuestión de “apuntar” el tope al elemento conveniente. Es decir, el primer elemento de la lista. Así, jamás tenemos que recorrer para push/pop.



Stack

Java viene equipada con una clase para el Stack.

<https://github.com/AdoptOpenJDK/openjdk-jdk14u/blob/master/src/java.base/share/classes/java/util/Stack.java>

Analizar el código y discutir:

- a) Cómo está implementada internamente
- b) Desde el punto de vista de OOP, ¿es correcto de dónde extiende comportamiento? ¿Por qué?

Stack

Este código, según Java, compilaría y ejecutaría. ¿Tiene sentido?

```
public static void main(String[] args) {  
  
    Stack<String> myStack = new Stack<>();  
  
    myStack.push("copy 1");  
    myStack.push("paste 1");  
    myStack.push("move 1");  
    myStack.push("move 2");  
    myStack.push("move 3");  
  
    myStack.add(3, "copy 2");  
  
    myStack.forEach(System.out::println);  
  
}
```

Stack

Realmente en Java se cometió un importante error de diseño.

Stack

La implementación correcta de la clase Stack es encapsulando una **LinkedList** (o arreglo). No especializando alguno de ellos...

```
public class Stack<T> {  
    private LinkedList<T> data = new LinkedList<>();  
  
    public void push(T v) { data.addFirst(v); }  
    public T peek() { return data.getFirst(); }  
    public T pop() { return data.removeFirst(); }  
    public boolean isEmpty() { return data.isEmpty(); }  
}
```

Faltaría mejorar el manejo de excepciones... (que no aparezcan excepciones de LinkedList).

Stack

Caso de Uso: Evaluador de expresiones

Una expresión es una combinación de operadores y operandos. En esta discusión vamos a considerar un subconjunto de operadores: solo binarios.

Las expresiones se pueden clasificar según la notación que utilizan:

- a) **prefija:** el operador se encuentra antes de los operandos sobre los que aplica
- b) **infija:** el operador se encuentra entre los operandos sobre los que aplica
- c) **postfija:** el operador se encuentra detrás de los operandos sobre los que aplica.

Ej: ¿Cómo es la notación que usamos en general en expresiones aritméticas?

Rta: Infija

$10 + 3 - 100$

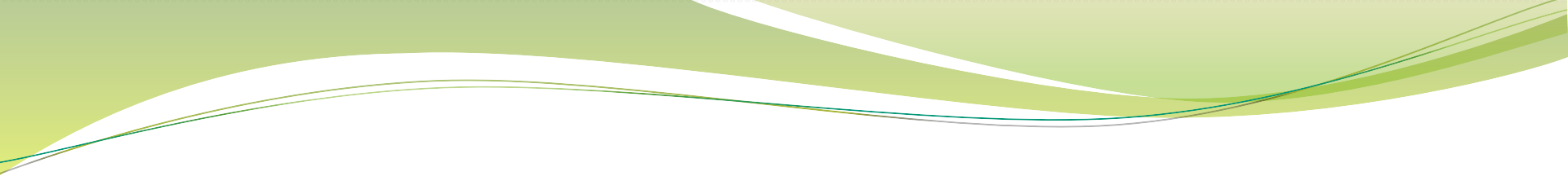
Dados 2 operandos A y B, el operador binario * puede tener las siguientes posibilidades:

Prefija	Infija	Postfija	Prefija Inversa	Infija Inversa	Postfija Inversa
* A B	A * B	A B *	* B A	B * A	B A *

El problema con la evaluación de una expresión infija es que existen ambigüedades cuando dos operadores tienen la misma precedencia. ¿ Por ejemplo?

Rta $10 - 2 - 3$ ¿Cómo se evalúa?

Algo así debe resolverse con asociatividad. Pero se complica más aún cuando aparecen paréntesis que cambian las prioridades.



En las notaciones prefija y postfija el uso de paréntesis es innecesario debido a que el orden de los operadores determina el orden real de las operaciones en la evaluación de expresiones.

Algoritmo para evaluar una expresión que ya esté en notación postfija:

- Cada operador en una expresión postfija se refiere a los operandos previos en la misma.
- Cuando aparece un operando hay que postergarlo porque no se puede hacer nada con él hasta que no llegue el operador, y como la notación es postfija el operador va a llegar después. Por lo tanto cada vez que se encuentre un operando la acción a tomar es **“pusharlo” en una pila**
- Cuando aparezca un operador en la expresión implica que llegó el momento de aplicárselo a los operandos que lo preceden, por lo tanto se deben **“poppear” los dos elementos más recientes de la pila**, aplicarles el operador y volver a dejar el resultado en la pila porque dicho valor puede ser operando para otra subexpresión (al resultado habrá que aplicársele el próximo operador que aparezca).
- Cuando se termine de analizar la expresión de entrada el resultado de su evaluación es el único valor que quedó en la **pila**.

- ¿Es un buen diseño elegir una pila para implementar este algoritmo?

Rta

Obviamente sí, porque como se observó la única forma de acceso a la estructura de datos fue a través de su tope.

Jamás se necesitó navegar por dentro de la estructura en busca de otras componentes. Siempre se respetó el orden de llegada de los elementos a la estructura. Nada mejor que una pila para esto.

- *Ejemplo:*

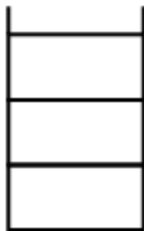
Supongamos que tenemos la expresión postfija

3 10 + 2 - 5 4 * -

(que corresponde a la infija: $(3 + 10) - 2 - 5 * 4$)

El algoritmo funciona así:

Pila



Entrada a analizar

3 10 + 2 - 5 4 * -
↑

Accion a tomar

Pushear 3

Pila



Entrada a analizar

3 10 + 2 ^ 5 4 * -
↑

Accion a tomar

Pushear 10

10
3

3 10 + 2 - 5 4 * -
 ↑

Pop 10
 Pop 3
Push 3 + 10

13

3 10 + 2 - 5 4 * -
 ↑

Push 2

2
13

3 10 + 2 - 5 4 * -
 ↑

Pop 2
 Pop 13
Push 13 - 2

11

3 10 + 2 - 5 4 * -
 ↑

Push 5

5
11

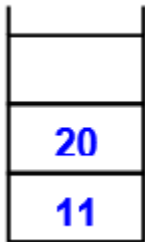
3 10 + 2 - 5 4 * -
 ↑

Push 4

4
5
11

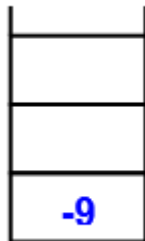
3 10 + 2 - 5 4 * -
 ↑

Pop 4
 Pop 5
Push 5 * 4



3 10 + 2 - 5 4 * -
 ↑

Pop 20
 Pop 11
 Pushear 11 - 20



Fin de la entrada

Pop -9



Pila vacía, y el resultado de evaluar la expresión es -9

TP 3B- Ejer 2 (2.2 y 2.3)

Como funciona el algoritmo para evaluar en postfija?

Seguimiento gráfico en papel (snapshot instante a instante de qué hace el algoritmo)

2.2) Mostrar paso a paso como se analiza el input, se utiliza la pila y se evalúa la siguiente expresión postfija:

2 -0.1 + 10 2 * /

Rta dibujito en pizarrón y valor devuelto 0.095

Ejercicio

¿A qué expresión infija correspondía?

Rta $(2 + -0.1) / (10 * 2)$

2.3) Mostrar paso a paso como se analiza el input, se utiliza la pila y se evalúa la siguiente expresión postfija:

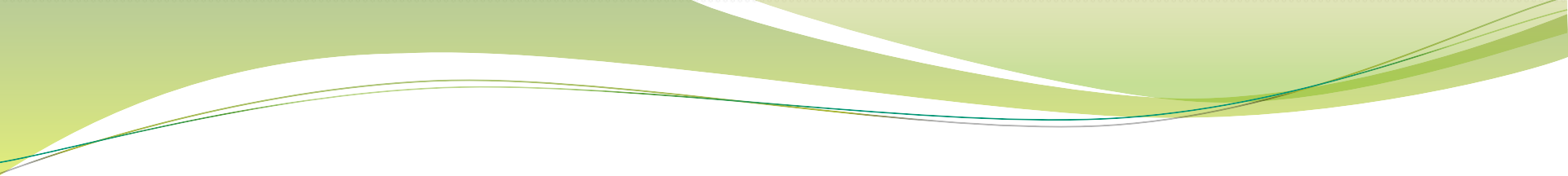
-9 -1 - 10 2 * / 1 5 - 2 -3 / / *

Rta dibujito en pizarrón y valor devuelto -2.4

Ejercicio

¿A qué expresión infija correspondía?

Rta $((-9 - -1) / (10 * 2)) * ((1 - 5) / (2 / -3))$



Acá tenemos 2 subproblemas:

- a) Cómo parsear un string de entrada para separarlo en tokens válidos (dónde terminan los números y los operadores)
- b) La evaluación en sí de la expresión postfija. Eso incluye el manejo de errores.

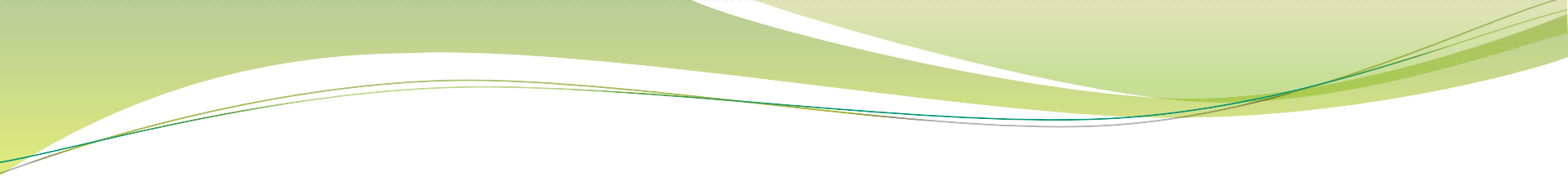
Si nuestros operadores admitidos son $+$ $-$ $*$ $/$

Proponer cuáles serían expresiones inválidas y explicar qué tipo de excepción debería lanzarse en cada caso.

Rta: $3 - 4$ es inválida porque el $-$ espera un operando previo

Rta: $0.2 \ 3 \ ?$ Es inválida porque el $?$ No es operador válido

Rta: $4 \ 3 \ 1 - 4$ Es inválida porque faltan operadores



Empecemos analizando qué clase Java me simplifica el análisis de tokens...

La clase **Scanner** permite leer de estandar input/archive/string información, indicarle cuáles son los separadores y navegar por sus tokens (iterador).

Probar el siguiente código que toma una línea de la **entrada estándar** que termina con \n. A ese primer scanner lo llamamos **inputScanner**.

Luego usa otro scanner sobre la línea previamente leída y la tokeniza separando en espacios (blancos, tabulador, etc). Ese segundo scanner lo llamamos **lineScanner**

```
public static void main(String[] args) {  
    // primer scanner: separador enter  
    Scanner inputScanner = new Scanner(System.in).useDelimiter("\\n");  
    System.out.print("Introduzca la expresión en notación postfija: ");  
  
    String line = inputScanner.next; // si usan nextLine() no poner \\r  
  
    // segundo scanner: separador espacios sobre el anterior  
    Scanner lineScanner = new Scanner(line).useDelimiter("\\s+");  
    while(lineScanner.hasNext())  
        System.out.println(lineScanner.next());  
}
```

Podemos chequearlo con una expresión regular. En este caso, la enumeración de opciones válidas se escriben separadas por un pipe.

```
String token = lineScanner.next();  
System.out.println(token);  
if (token.matches("¡!|,|;|##") )  
    System.out.println("OK:“: token );  
else  
    System.out.println("invalid: " + token);
```

Agregar este chequeo al código anterior y ver que valida correctamente

Ejercicio

Modificar el código anterior para que acepte un token más: ¿?

Chequearlo con expresiones válidas e inválidas. ¿Funciona correctamente?

Rta:

Así funciona mal `if (token.matches("¿?|!|,|;|##"))` porque el símbolo “?” es metasímbolo y significa opcionalidad

La forma correcta es escaparlo:

```
if (token.matches("¿\\?|!|,|;|##"))
```