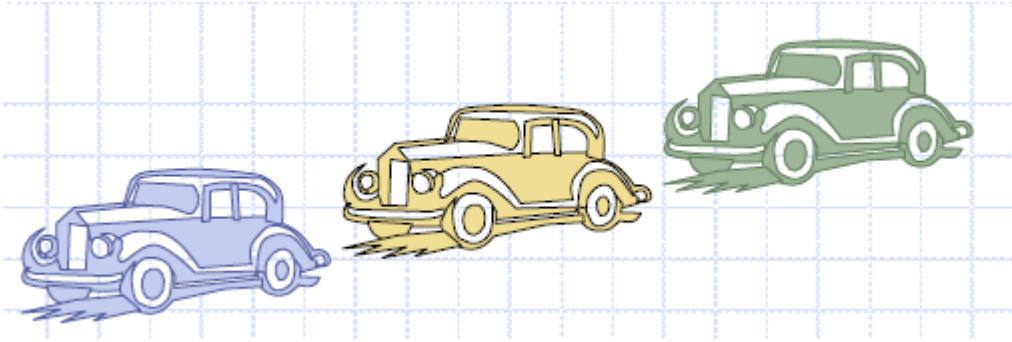
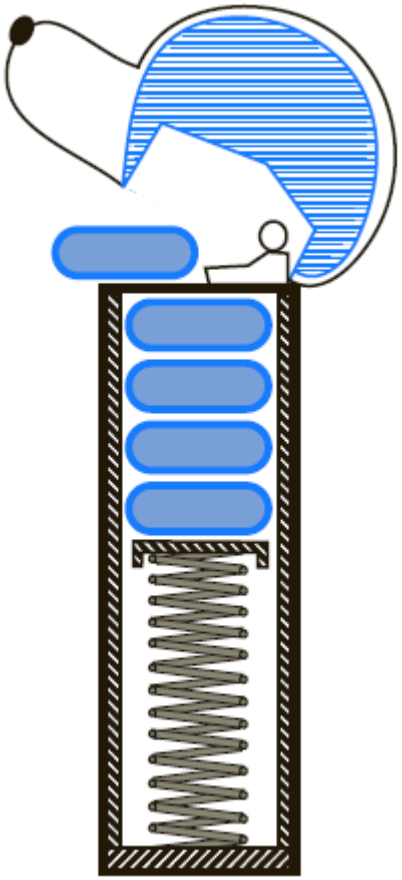




Pilas y Colas

AyP II

Lic. Renato Mazzanti

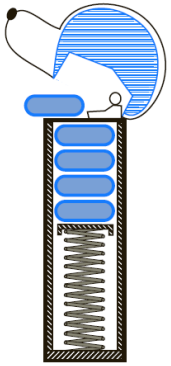
Tipos Abstractos de Datos (TAD/ADT)

- Es una abstracción de una estructura de datos.
- Especifica:
  - Los datos almacenados
  - Las operaciones sobre esos datos
  - Las condiciones de error asociadas con esos datos

- Ejemplo: modelando un sistema simple de gestión de stock (acciones)
  - Los datos almacenados son órdenes de compra / venta
  - Las operaciones soportadas son:
    - buy(stock, shares, price)
    - sell(stock, shares, price)
    - void cancel(order)
  - Error conditions:
    - buy/sell a nonexistent stock
    - Cancel a nonexistent order

El TAD Pila

- La pila almacena objetos arbitrarios
- Inserción y eliminación
- Sigue el esquema last-in first-out (LIFO)
- Pensemos en el dispenser con resorte
- La operaciones principales son:
  - push(object)
  - pop()



- Operaciones auxiliares:
- object top(): retorna el último elemento insertado sin removerlo
- integer size(): retorna el número de elementos almacenados
- boolean isEmpty(): indica si no hay elementos almacenados

Interfaz de Pila en Java

- Interfaz Java del TAD Pila
- Asume que si devuelve null top() and pop() la pila está vacía
- Diferente de la clase Java incorporada Java.util.Stack

```
public interface Stack<E> {
  int size();
  boolean isEmpty();
  E top();
  void push(E element);
  E pop();
}
```

Ejemplo

Method	Return Value	Stack Contents
push(5)	–	(5)
push(3)	–	(5, 3)
size( )	2	(5, 3)
pop( )	3	(5)
isEmpty( )	false	(5)
pop( )	5	( )
isEmpty( )	true	( )
pop( )	null	( )
push(7)	–	(7)
push(9)	–	(7, 9)
top( )	9	(7, 9)
push(4)	–	(7, 9, 4)
size( )	3	(7, 9, 4)
pop( )	4	(7, 9)
push(6)	–	(7, 9, 6)
push(8)	–	(7, 9, 6, 8)
pop( )	8	(7, 9, 6)

Excepciones vs retornar null

- Intentar la ejecución de una operación de un ADT puede causar a veces una condición de error
- Java soporta una abstracción general de errores, llamada excepción
- Una excepción se dice que es "lanzada" por una operación que no se puede ejecutar correctamente
- En nuestro TAD Stack, no usamos excepciones
- En su lugar, permitimos que las operaciones pop y top se realicen incluso si la pila está vacía
- Para una pila vacía, pop y top simplemente devuelven null

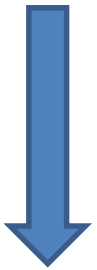
Aplicaciones de Pila

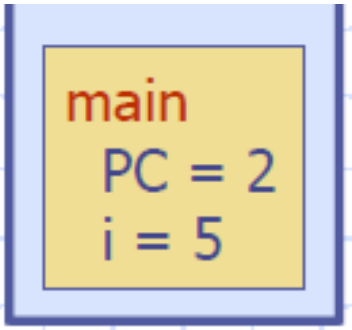
- Directas
  - Historial visitado por la página en un navegador Web
  - Deshacer secuencia en un editor de texto
  - Cadena de llamadas de método en la máquina virtual Java
- Indirectas
  - Estructura de datos auxiliares para algoritmos
  - Componente de otras estructuras de datos

El método STACK en la JVM

- La Máquina Virtual Java (JVM) realiza un seguimiento de la cadena de métodos activos con una pila.
- Cuando se llama a un método, la JVM empuja sobre la pila un frame.
  - Variables locales y valores de retorno
  - Contador de programa, seguimiento de la instrucción en ejecución
- Cuando un método termina, su frame se desapila de la pila y el control se pasa al método en el tope de la pila
- Permite la recursividad

```
main() {
  int i = 5;
  foo(i);
  }

foo(int j) {
  int k;
  k = j+1;
  bar(k);
  }

bar(int m) {
  …
  }
```

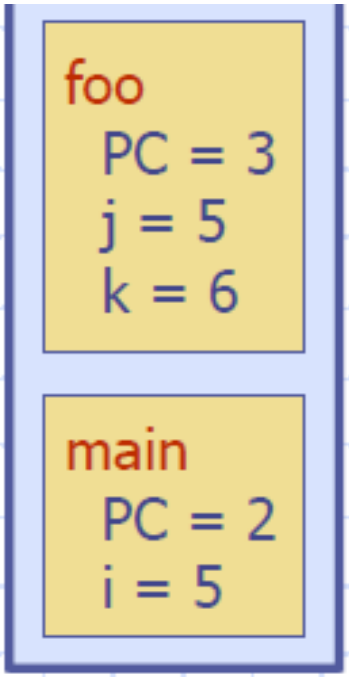




El método STACK en la JVM

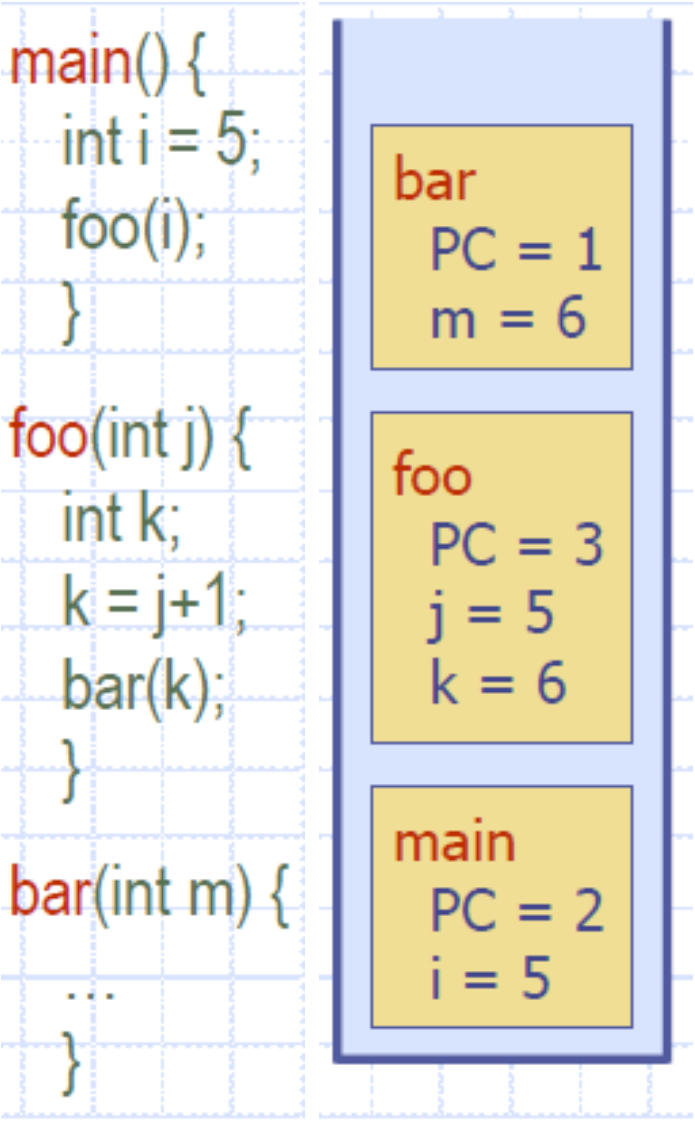
- La Máquina Virtual Java (JVM) realiza un seguimiento de la cadena de métodos activos con una pila.
- Cuando se llama a un método, la JVM empuja sobre la pila un frame.
  - Variables locales y valores de retorno
  - Contador de programa, seguimiento de la instrucción en ejecución
- Cuando un método termina, su frame se desapila de la pila y el control se pasa al método en el tope de la pila
- Permite la recursividad

```
main() {
  int i = 5;
  foo(i);
  }

foo(int j) {
  int k;
  k = j+1;
  bar(k);
  }

bar(int m) {
  …
  }
```



El método STACK en la JVM

- La Máquina Virtual Java (JVM) realiza un seguimiento de la cadena de métodos activos con una pila.
- Cuando se llama a un método, la JVM empuja sobre la pila un frame.
  - Variables locales y valores de retorno
  - Contador de programa, seguimiento de la instrucción en ejecución
- Cuando un método termina, su frame se desapila de la pila y el control se pasa al método en el tope de la pila
- Permite la recursividad



Pila basada en arreglo

- Una forma simple de implementación es usar un arreglo
- Agregamos elementos de izquierda a derecha
- Una variable mantiene el seguimiento del índice del elemento superior

```
Algorithm size()
    return t + 1

Algorithm pop()
    if isEmpty() then
        return null
    else
        t ← t − 1
        return S[t + 1]
```

Pila basada en arreglo

- Almacena elementos mientras no está lleno el arreglo
- Push lanza la excepción FullStackException
  - Limitación por el arreglo
  - No intrínseca de la pila

```
Algorithm push(o)
  if t = S.length − 1 then
    throw IllegalStateException
  else
    t ← t + 1
    S[t] ← o
```

Performance y limitaciones

- Performance
  - Sea n el número de elementos de la pila
  - El espacio usados es $O(n)$
  - Cada operación corre en $O(1)$
- Limitaciones
  - El tamaño máximo de la pila se define apriori y no se puede cambiar
  - Intentar empujar un nuevo elemento con la pila llena causa un excepción específica de la implementación

Pila basada en arreglo en Java

```
public class ArrayStack<E>
    implements Stack<E> {

  // holds the stack elements
  private E[ ] S;

  // index to top element
  private int top = -1;

  // constructor
  public ArrayStack(int capacity) {
      S = (E[ ]) new Object[capacity];
  }
```

```
  public E pop() {
    if isEmpty()
          return null;
      E temp = S[top];
      // facilitate garbage collection:
      S[top] = null;
      top = top – 1;
      return temp;
    }

…  (other methods of Stack interface)
```

Nota: ver Pag. 234 Goodrich

Ejemplo de uso en Java

```
public class Tester {
   // … other methods
   public intReverse(Integer a[]) {
       Stack<Integer> s;
                        s = new
ArrayStack<Integer>();
       … (code to reverse array a) …
    }
```

```
   public floatReverse(Float f[]) {
       Stack<Float> s;
       s = new ArrayStack<Float>();
       … (code to reverse array f) …
    }
```

LinkedStack

Patrón de diseño Adaptador

Stack Method	*Singly Linked List Method*
size( )	list.size( )
isEmpty( )	list.isEmpty( )
push(e)	list.addFirst(e)
pop( )	list.removeFirst( )
top( )	list.first( )

```
1  public class LinkedStack<E> implements Stack<E> {
2    private SinglyLinkedList<E> list = new SinglyLinkedList<>();     // an empty list
3    public LinkedStack( ) { }                  // new stack relies on the initially empty list
4    public int size( ) { return list.size( ); }
5    public boolean isEmpty( ) { return list.isEmpty( ); }
6    public void push(E element) { list.addFirst(element); }
7    public E top( ) { return list.first( ); }
8    public E pop( ) { return list.removeFirst( ); }
9  }
```

Balanceo de paréntesis

- Cada “(”, “{”, o “[” debe esta apareado con su coincidente “)”, “}”, o “[”
  - correcto: ( )(( )){([( )])}
  - correcto: ((( )(( )){([( )])}
  - incorrecto:)(( )){([( )])}
  - incorrecto: ({[ ])}
  - incorrecto: (

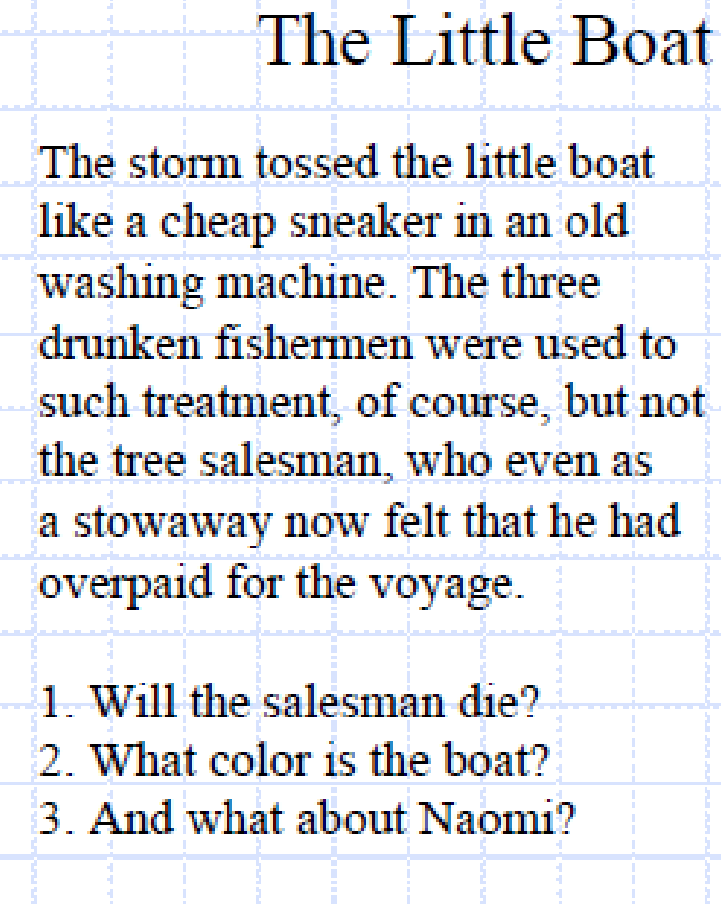
Balanceo de paréntesis en Java

```
public static boolean isMatched(String expression) {
  final String opening = "({["; // opening delimiters
  final String closing = ")}]"; // respective closing delimiters
  Stack<Character> buffer = new LinkedStack<>( );
  for (char c : expression.toCharArray( )) {
    if (opening.indexOf(c) != -1) // this is a left delimiter
      buffer.push(c);
    else if (closing.indexOf(c) != -1) { // this is a right delimiter
      if (buffer.isEmpty( )) // nothing to match with
        return false;
      if (closing.indexOf(c) != opening.indexOf(buffer.pop( )))
        return false; // mismatched delimiter
      }
    }
  return buffer.isEmpty( ); // were all opening delimiters matched?
}
```

HTML tag matching

El código HTML correcto, cada <name> debe estar apareado con su </name>

```
<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine. The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even as
a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>
```



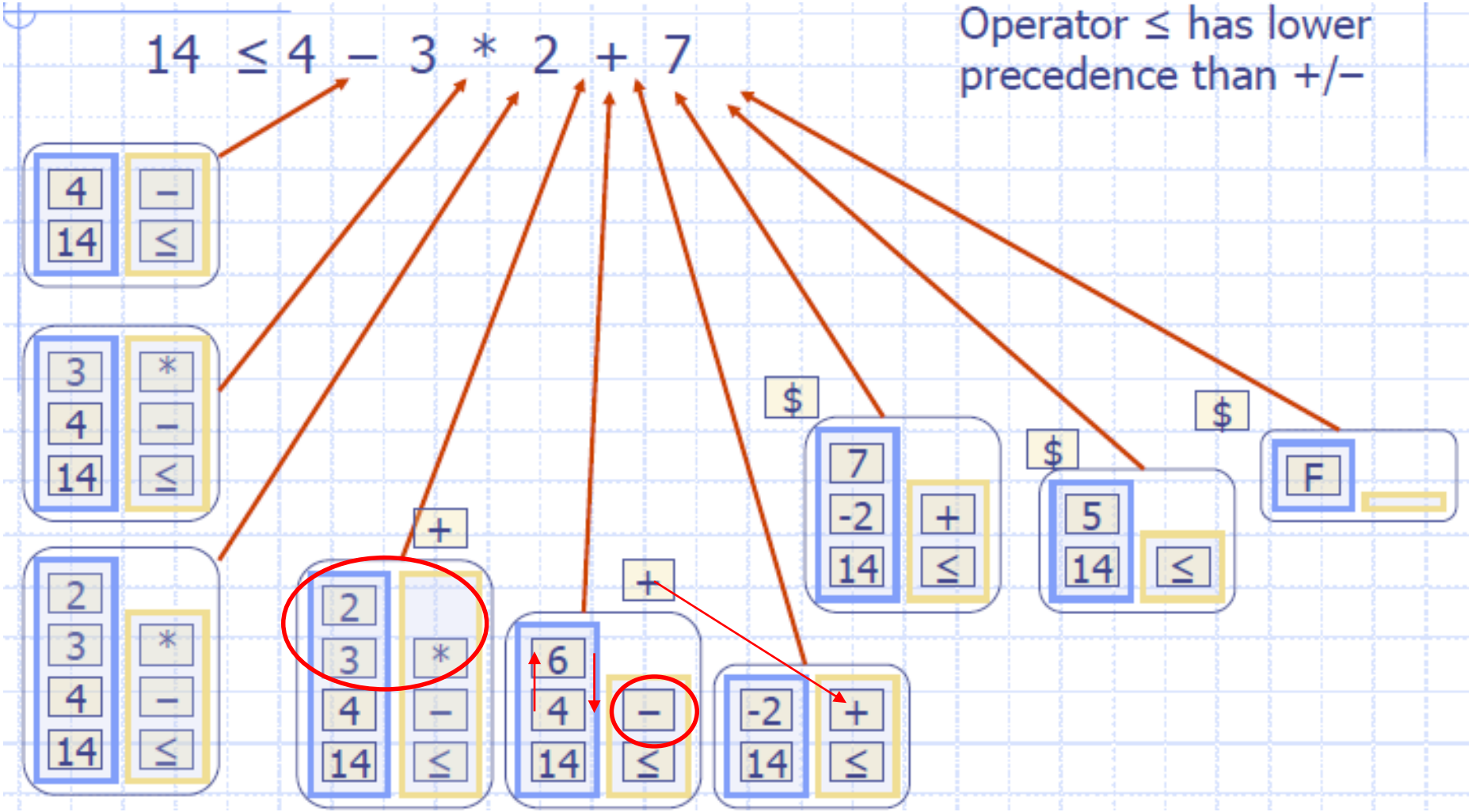
HTML Tag Matching (Java)

```
public static boolean isHTMLMatched(String html) {
  Stack<String> buffer =  new LinkedStack<>( );
  int j = html.indexOf('<'); // find first '<' character (if any)
  while (j != −1) {
    int k = html.indexOf('>', j+1); // find next '>' character
    if (k == −1)
      return false; // invalid tag
    String tag = html.substring(j+1, k); // strip away < >
    if (!tag.startsWith("/")) // this is an opening tag
      buffer.push(tag);
    else { // this is a closing tag
      if (buffer.isEmpty( ))
        return false; // no tag to match
      if (!tag.substring(1).equals(buffer.pop( )))
        return false; // mismatched tag
    }
    j = html.indexOf('<', k+1); // find next '<' character (if any)
  }
  return buffer.isEmpty( ); // were all opening tags matched?
}
```

Evaluando expresiones aritméticas

- 14 – 3 * 2 + 7 = (14 – (3 * 2)) + 7
- Precedencias de operadores
  - * tiene precedencia sobre +/–
- Asociatividad
- Operadores del mismo grupo de precedencia se evaluan de **izquierda a derecha**
- Ejemplo: “***+ y –”*** *se resuelve* **(x – y) + z** en lugar de x – (y + z)
- Idea: empujar cada operando u operados a la pila correspondiente, ***push***, pero ejecuta la operación con más alta o igual precedencia con ***pop***.

Ejemplo


14 ≤ 4 – 3 * 2 + 7
Operator ≤ has lower precedence than +/–
4
14
–
≤
3
4
14
*
–
≤
2
3
4
14
*
–
≤
+
2
3
4
14
*
–
≤
+
6
4
14
–
≤
$
7
-2
14
+
≤
+
-2
14
+
≤
$
5
14
≤
$
F

Algoritmo para evaluar expresiones aritméticas

Two stacks:

- opStk holds operators
- valStk holds values
- Use $ as special "end of input" token with lowest precedence

Algorithm doOp()

```
x ← valStk.pop();
y ← valStk.pop();
op ← opStk.pop();
valStk.push( y op x )
```

Algorithm repeatOps(refOp):

```
while ( valStk.size() > 1 ∧
        prec(refOp) ≤
        prec(opStk.top())
    doOp()
```

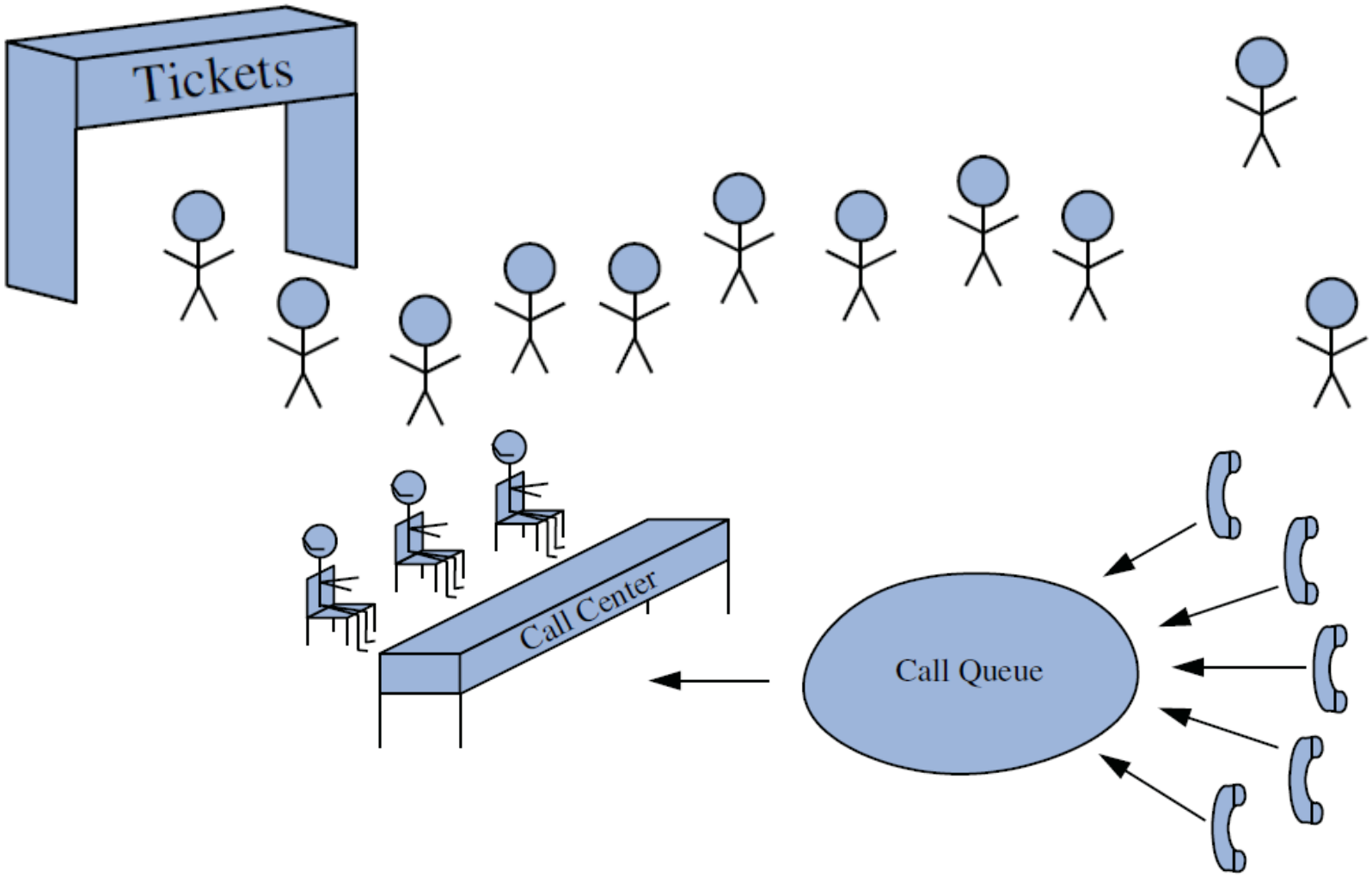
Algorithm EvalExp()

Input: a stream of tokens representing an arithmetic expression (with numbers)

Output: the value of the expression

```
while there's another token z
    if isNumber(z) then
        valStk.push(z)
    else
        repeatOps(z);
        opStk.push(z)
repeatOps($);
return valStk.top()
```

El TAD Cola



El TAD cola

- El TAD cola almacena objetos arbitrarios
- La inserción y eliminación sigue el esquema first-in first-out (FIFO)
- La inserción se hace por atrás de la cola y se remueve por el frente de la cola
- Principales operaciones:
  - enqueue(object): inserta un elemento al final de la cola
  - object dequeue(): remueve y retorna el elemento del frente de la cola

El TAD Cola

- Operaciones auxiliares:
  - object first(): retorna el elemento del frente sin removerlo
  - integer size(): retorna el número de elementos almacenados
  - boolean isEmpty(): Indica si no hay elementos almacenados
- Casos límite:
  - intentar ejecutar un dequeue o first sobre una cola vacía retorna null

Ejemplo

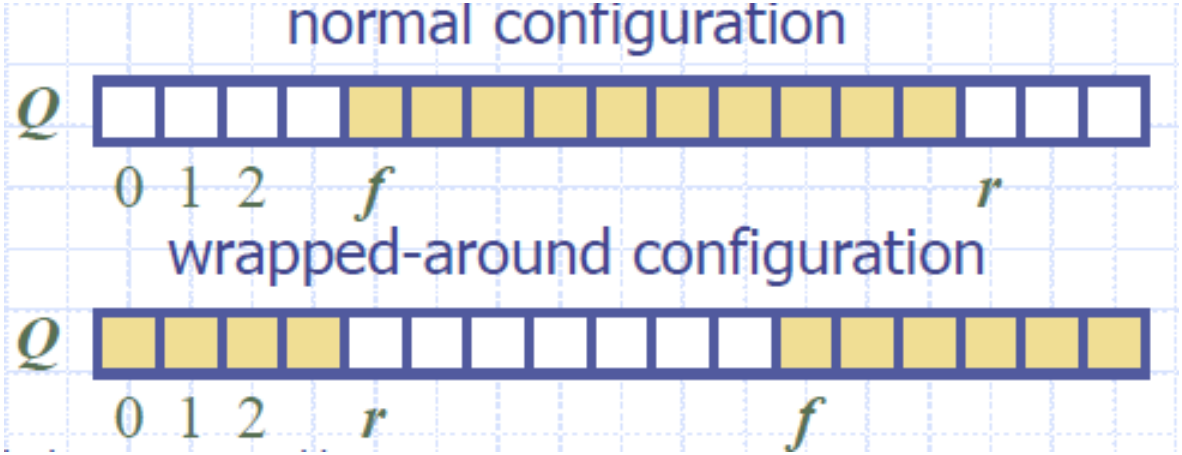
Operation		*Output*	*Q*
enqueue(5)	–	(5)	
enqueue(3)	–	(5, 3)	
dequeue()	*5*	(3)	
enqueue(7)	–	(3, 7)	
dequeue()	*3*	(7)	
first()	*7*	(7)	
dequeue()	*7*	()	
dequeue()	*null*	()	
isEmpty()	*true*	()	
enqueue(9)	–	(9)	
enqueue(7)	–	(9, 7)	
size()	*2*	(9, 7)	
enqueue(3)	–	(9, 7, 3)	
enqueue(5)	–	(9, 7, 3, 5)	
dequeue()	*9*	(7, 3, 5)	

Aplicaciones de Cola

- Directas
  - Listas de espera, burocracia
  - Acceso a recursos compartidos (ej., impresora)
  - Multiprogramación
- Aplicaciones indirectas
  - Estructura de datos auxiliar para algorítmos
  - Componente de otras estructuras de datos

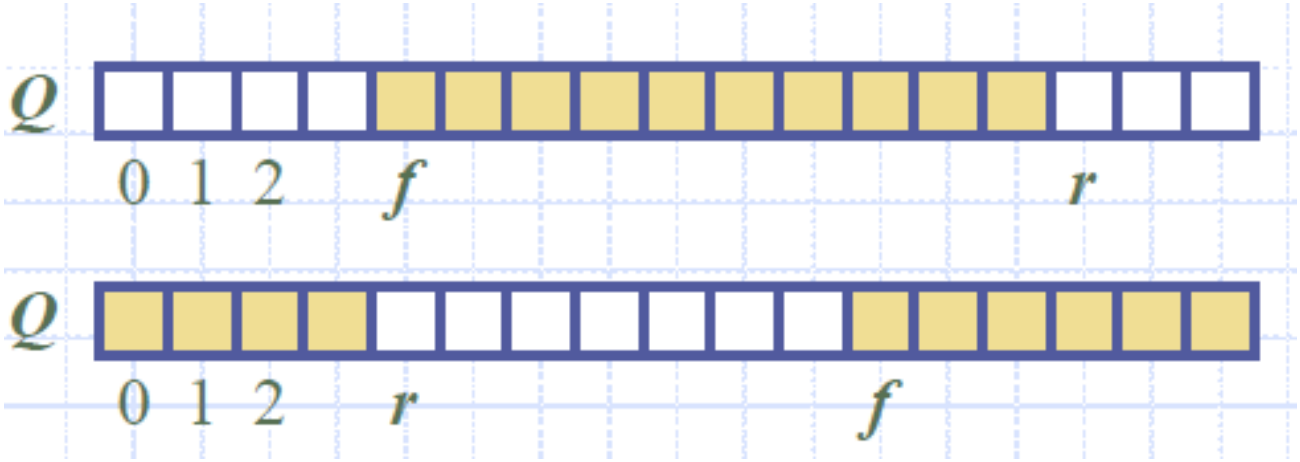
Cola basada en arreglo

- Usa un arreglo de tamaño ***N*** en forma circular
- Dos variables hacen el seguimientos del frente y el tamaño ***f*** es el índice del frente y ***sz*** número de elementos almacenados
- Cuando la cola tiene menos que ***N*** elementos, la ubicación del arreglo ***r = (f + sz) mod N*** es el primer lugar vacío luego del final de la cola.



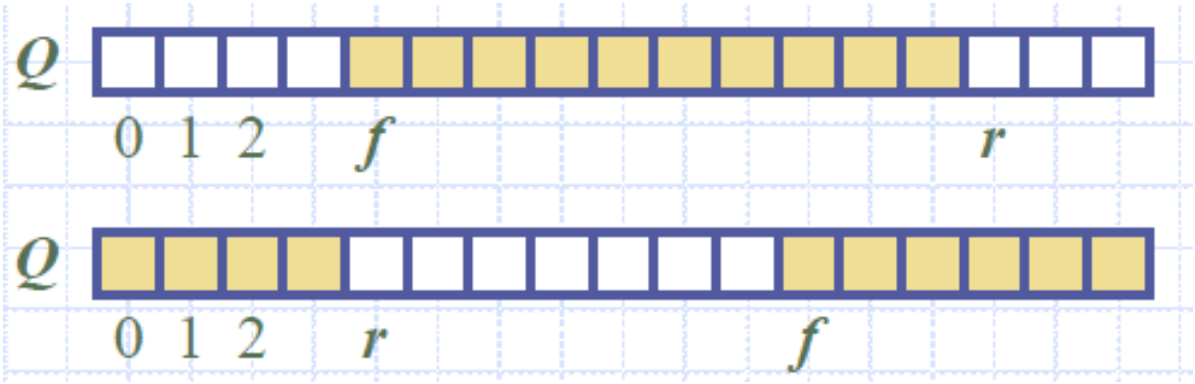
Operaciones de Cola

```
Algorithm size()
  return sz

Algorithm isEmpty()
  return (sz == 0)
```



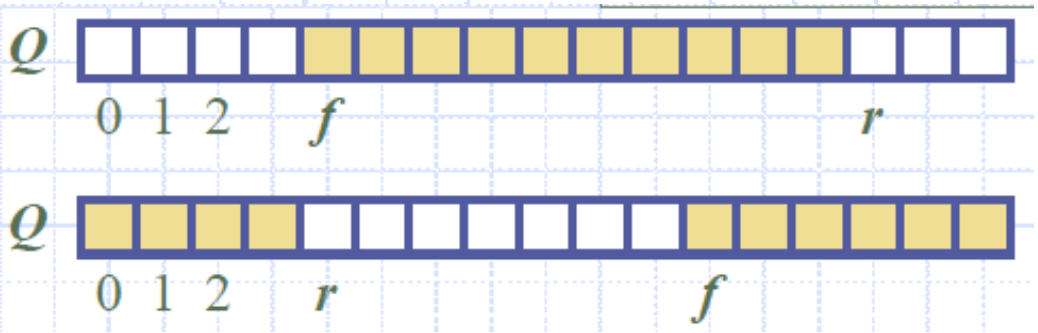
Operaciones de Cola

```
Algorithm enqueue(o)
  if size() = N − 1 then
    throw IllegalStateException
  else
    r ← (f + sz) mod N
    Q[r] ← o
    sz ← (sz + 1)
```



Operaciones de Cola

```
Algorithm dequeue()
  if isEmpty() then
    return null
  else
    o ← Q[f]
    f ← (f + 1) mod N
    sz ← (sz − 1)
    return o
```



Interfaz de Cola en Java

```
public interface Queue<E> {
    int size();
    boolean isEmpty();
    E first();
    void enqueue(E e);
    E dequeue();
}
```

Implementación basada en arreglo

```
1  /** Implementation of the queue ADT using a fixed-length array. */
2  public class ArrayQueue<E> implements Queue<E> {
3    // instance variables
4    private E[ ] data;                          // generic array used for storage
5    private int f = 0;                          // index of the front element
6    private int sz = 0;                         // current number of elements
7
8    // constructors
9    public ArrayQueue( ) {this(CAPACITY);} // constructs queue with default capacity
10   public ArrayQueue(int capacity) {          // constructs queue with given capacity
11     data = (E[ ]) new Object[capacity];      // safe cast; compiler may give warning
12   }
13
14   // methods
15   /** Returns the number of elements in the queue. */
16   public int size( ) { return sz; }
17
18   /** Tests whether the queue is empty. */
19   public boolean isEmpty( ) { return (sz == 0); }
20
```

Implementación basada en arreglo

```
21    /** Inserts an element at the rear of the queue. */
22    public void enqueue(E e) throws IllegalStateException {
23      if (sz == data.length) throw new IllegalStateException("Queue is full");
24      int avail = (f + sz) % data.length;       // use modular arithmetic
25      data[avail] = e;
26      sz++;
27    }
28
29    /** Returns, but does not remove, the first element of the queue (null if empty). */
30    public E first( ) {
31      if (isEmpty( )) return null;
32      return data[f];
33    }
34
35    /** Removes and returns the first element of the queue (null if empty). */
36    public E dequeue( ) {
37      if (isEmpty( )) return null;
38      E answer = data[f];
39      data[f] = null;                           // dereference to help garbage collection
40      f = (f + 1) % data.length;
41      sz--;
42      return answer;
43    }
```

Análisis del algorítmo cola implementada con arreglo

- El tamaño de memoria es O(n)
- El tiempo de ejecución

Method	Running Time
size	$O(1)$
isEmpty	$O(1)$
first	$O(1)$
enqueue	$O(1)$
dequeue	$O(1)$

Comparación java.util.Queue

- Nuestros métodos y los de java.util.Queue

Our Queue ADT	**Interface java.util.Queue**	
	throws exceptions	returns special value
enqueue(e)	add(e)	offer(e)
dequeue( )	remove( )	poll( )
first( )	element( )	peek( )
size( )	size( )	
isEmpty( )	isEmpty( )	

Implementación de Cola con lista enlazada simple

- Usando el patrón de diseño adaptador

```
1  /** Realization of a FIFO queue as an adaptation of a SinglyLinkedList. */
2  public class LinkedQueue<E> implements Queue<E> {
3    private SinglyLinkedList<E> list = new SinglyLinkedList<>();    // an empty list
4    public LinkedQueue() { }                    // new queue relies on the initially empty list
5    public int size() { return list.size(); }
6    public boolean isEmpty() { return list.isEmpty(); }
7    public void enqueue(E element) { list.addLast(element); }
8    public E first() { return list.first(); }
9    public E dequeue() { return list.removeFirst(); }
10 }
```

Análisis de la eficiencia de la Cola *SinglyLinkedList*

- *SinglyLinkedList*, al examinarlo queda claro que cada método de esa clase se ejecuta en O (1) el peor de los casos.
- Por lo tanto, cada método de nuestra adaptación de *LinkedQueue* también se ejecuta en O (1) en el peor de los casos.
- También evitamos la necesidad de especificar un tamaño máximo para la cola, como se hizo en la implementación de la cola basada en arreglos.
- Debido a que cada nodo almacena una referencia al siguiente, además de la referencia del elemento que contiene, una lista enlazada usa más espacio por elemento que una implementada con arreglo.
- Aunque todos los métodos se ejecutan en tiempo constante para ambas implementaciones, parece claro que las operaciones que involucran listas enlazadas tienen una gran cantidad de operaciones primitivas para resolver las referencias.
- En la práctica, esto hace que el método de lista enlazada sea más costoso que el método basado en matrices.

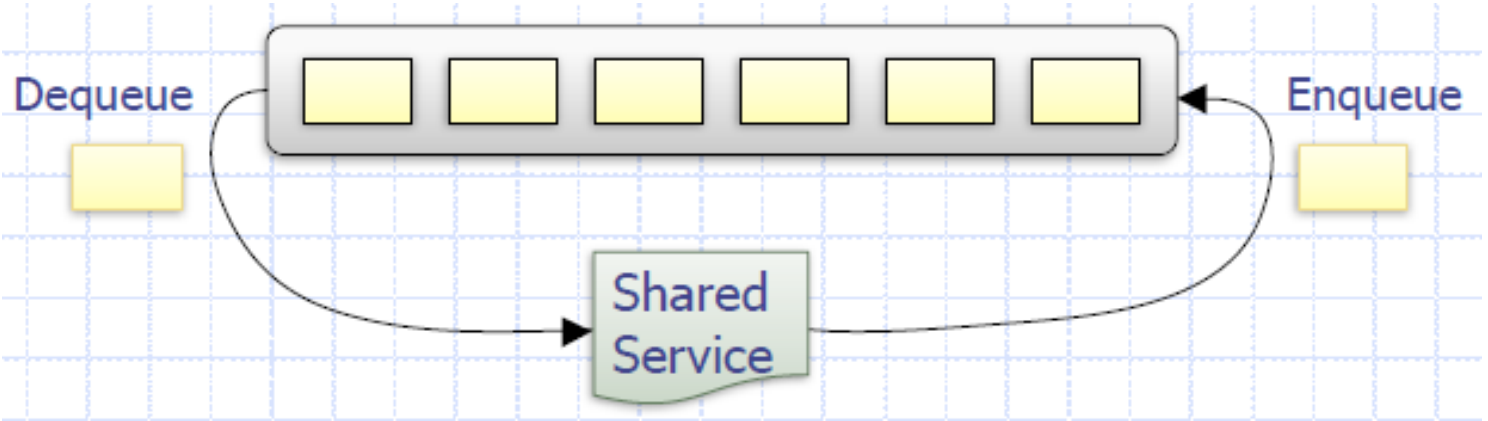
Cola Circular

- Una cola circular es una excelente abstracción para aplicaciones en las que los elementos se organizan cíclicamente, como para multijugadores, juegos por turnos o programación por turnos de procesos informáticos.

```
1 public interface CircularQueue<E> extends Queue<E> { 2 /**
3 * Rotates the front element of the queue to the back of the queue.
4 * This does nothing if the queue is empty.
5 */
6 void rotate( );
7 }
```

Application: Round Robin Schedulers

- Podemos implementar un ***round robin scheduler*** usando una cola Q ejecutando repetidamente los pasos:
- 1. e = Q.dequeue()
- 2. Service element e
- 3. Q.enqueue(e)



Problema de Josephus

- La gente está parada en un círculo esperando ser ejecutada.
- El conteo comienza en un punto específico del círculo y continúa alrededor del círculo en una dirección específica.
- Después de que se omite un número específico de personas, se ejecuta la siguiente persona.
- El procedimiento se repite con el resto de personas, comenzando por la siguiente, yendo en la misma dirección y saltando el mismo número de personas, hasta que solo queda una persona que se salva de la ejecución.

- ***Dado el número de personas, el punto de partida, la dirección y el número que se debe omitir, es cuestión de elegir la posición en el círculo inicial para evitar la ejecución.*** https://en.wikipedia.org/wiki/Josephus_problem#:~:text=The%20problem%20is%20named%20after,committing%20suicide%20by%20drawing%20lots.

Colas de doble final
Double-Ended Queues

- Esta estructura se llama cola de dos extremos, o deque, que generalmente se pronuncia "deck“ para evitar confusiones con el método de dequeue del TAD de cola normal, que se pronuncia como la abreviatura "D.Q.“
- El tipo de datos abstracto deque es más general que los TADs de pila y cola.
- Ejemplo:
  - La cola de un restaurante
  - la primera persona puede ser eliminada de la cola cuando no hay una mesa adecuada disponible
  - También puede ser que un cliente al final de la cola se impaciente y abandone el restaurante.

Colas de doble final
Double-Ended Queues

Method	Running Time
size, isEmpty	$O(1)$
first, last	$O(1)$
addFirst, addLast	$O(1)$
removeFirst, removeLast	$O(1)$

Our Deque ADT	Interface java.util.Deque	
	throws exceptions	returns special value
first( )	getFirst( )	peekFirst( )
last( )	getLast( )	peekLast( )
addFirst(e)	addFirst(e)	offerFirst(e)
addLast(e)	addLast(e)	offerLast(e)
removeFirst( )	removeFirst( )	pollFirst( )
removeLast( )	removeLast( )	pollLast( )
size( )	size( )	
isEmpty( )	isEmpty( )	

Bibliografía

- Data Structures and Algorithms in Java™. Sixth Edition. Michael T. Goodrich, Department of Computer Science University of California. Roberto Tamassia, Department of Computer Science Brown University. Michael H. Goldwasser, Department of Mathematics and Computer Science Saint Louis University. Wiley. 2014.