

# Estructuras Lineales

## Pilas y Colas

Verónica E. Arriola-Rios

Facultad de Ciencias, UNAM

2 de octubre de 2024



# TDA: Pila, Cola y Bicola

- 1 TDA: Pila, Cola y Bicola
- 2 Notaciones infija, prefija y postfija
- 3 Implementaciones
- 4 Bibliografía

# Temas

## 1 TDA: Pila, Cola y Bicola

- Pilas
- Colas
- Bicolas

## Pilas (Stacks)

- Son estructuras de datos en las que el último elemento almacenado es el primero en ser devuelto.
  - **LIFO**: “Last in, first out”. «Último en entrar, primero en salir»
  - Operaciones:
    - empty: Pila → boolean
    - pop: Pila → Elemento // Devuelve y elimina el primer elemento
    - peek: Pila → Elemento // Devuelve el primer elemento
    - push: Pila, Elemento → void // Agrega Elemento a la cabeza de la pila



## Aplicaciones

- Revisar paréntesis bien balanceados:

((((())((()())))))

- Invertir secuencias de elementos:

frase = papalote

inversa = etolapap

- Llevar el registro de las llamadas a funciones en una máquina virtual: la **pila de ejecución**.

# Paréntesis balanceados

```
1 public class ParéntesisBalanceados
2 {
3     public static bool Verifica(string expresión)
4     {
5         Stack<char> s = new Stack<char>();
6         foreach(char c in expresión.ToCharArray())
7         {
8             if (c == '(') s.Push(c);
9             else if (c == ')')
10             {
11                 char dato;
12                 if (!s.TryPop(out dato)) return false; // ')' de más
13             }
14             else throw new ArgumentException("Sólo parentesis");
15         }
16         if (!(s.Count == 0)) return false; // No se cerraron todos los paréntesis.
17         return true;
18     }
19 }
```

```
1 public class ClasePrueba
2 {
3     private static void Prueba(string expr, bool rEsperado)
4     {
5         Console.WriteLine(expr + ": " + rEsperado + " -> " +
6             ParéntesisBalanceados.Verifica(expr));
7     }
8
9     public static void Main()
10    {
11        Prueba("", true);
12        Prueba("((())())()", true);
13        Prueba("(()((", false);
14        Prueba("()()()()", true);
15        Prueba("(((())())()", false);
16    }
17 }
```

```
1 $ dotnet run
2 : true->true
3 (((()())()): true->true
4 ((()(: false->false
5 ()()()(): true->true
6 (((()())()): false->false
```

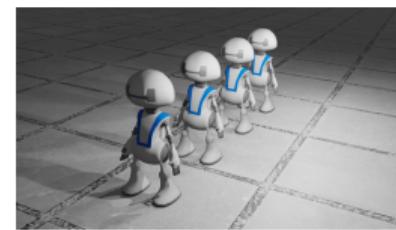
# Temas

## 1 TDA: Pila, Cola y Bicola

- Pilas
- Colas
- Bicolas

# Colas (Queues)

- Son estructuras de datos en las que los elementos se insertan en un extremo (el posterior) y se suprimen en el otro (el anterior o frente) Vargas Villazón, Lozano Moreno y Levine Gutiérrez 1998.
- **FIFO:** “First in, first out”. «Primero en entrar, primero en salir»
- Operaciones:
  - empty: Cola → boolean
  - queue: Cola, Elemento → void // Agrega Elemento al final de la cola
  - dequeue: Cola → Elemento // Devuelve y elimina el primer elemento
  - peek: Cola → Elemento // Devuelve el primer elemento
- Nota: Java usa comportamientos ligeramente modificados.



# Aplicaciones

En general se utilizan para mantener elementos a ser atendidos en el orden en que llegaron:

- Cola de impresiones.
- Cola de caracteres (o palabras) a procesar en un texto.

Una versión más avanzada es la *cola de prioridades*, donde los elementos se mantienen en el orden en que llegan dentro de niveles de prioridad:

- Cola de atención en un hospital.
- ~~Cola para inscripciones a grupos en la Facultad de Ciencias.~~ Bueno, de hecho no, porque dentro de cada prioridad se hace un sorteo, no aplica “el dedo más rápido del oeste”.

# Temas

## 1 TDA: Pila, Cola y Bicola

- Pilas
- Colas
- Bicolas

# Bicolas (Queues)

- Semejante a la cola, pero con acceso al primer y último elemento.

## Notaciones infija, prefija y postfija

- 1 TDA: Pila, Cola y Bicola
- 2 Notaciones infija, prefija y postfija
- 3 Implementaciones
- 4 Bibliografía

# Temas

## 2 Notaciones infija, prefija y postfija

- Infija
- Prefija
- Postfija (o Sufija)
- Evaluación
- Infija a postfija

# Notación infija

- La *notación infija* es la que utilizamos normalmente desde la primaria.

$$2 + 9 = 11$$

$$5 \times 8 = 40$$

$$7 - 9 = -2$$

- Los **operando**s se ubican a ambos lados de los **operador**e{s}.
- Requiere de la definición de reglas de **precedencia** y el uso de paréntesis para alterar el orden de operación.

Primaria

$$2 + 9 \times 3 = 11 \times 3 = 33$$

Secundaria

$$2 + 9 \times 3 = 2 + 27 = 29$$

$$(2 + 9) \times 3 = 11 \times 3 = 33$$

# Temas

## 2 Notaciones infija, prefija y postfija

- Infija
- **Prefija**
- Postfija (o Sufija)
- Evaluación
- Infija a postfija

# Prefija

- En la *notación prefija* el **operador** va **antes** que los **operandos**.

$$+ \ 2 \times 9 \ 3 = + \ 2 \ 27 = 29$$

$$\times + \ 2 \ 9 \ 3 = \times 11 \ 3 = 33$$

- Se utiliza en lenguajes como LISP.
- No requiere el uso de precedencia ni paréntesis, por lo que es más fácil implementar un algoritmo para evaluar las operaciones.

# Temas

## 2 Notaciones infija, prefija y postfija

- Infija
- Prefija
- Postfija (o Sufija)
- Evaluación
- Infija a postfija

# Postfija

- La *notación postfija* es semejante a la prefija, pero los operadores van **después** de los operandos.

$$2 \ 9 \ 3 \ \times \ + = 2 \ 27 \ + = 29$$

$$2 \ 9 \ + \ 3 \ \times = 11 \ 3 \ \times = 33$$

- Tiene las mismas ventajas que la prefija.

# Temas

## 2 Notaciones infija, prefija y postfija

- Infija
- Prefija
- Postfija (o Sufija)
- Evaluación
- Infija a postfija

# Evaluar notación prefija

---

## Algorithm Algoritmo para evaluar una expresión en notación prefija

---

```
1: function EVALÚAPREFIJA(expresión)
2:   pila  $\leftarrow$  []
3:   for each símbolo in expresión leyendo de derecha a izquierda do
4:     if símbolo es un número then
5:       pila.push(símbolo)
6:     else if símbolo es un operador then
7:       operando1  $\leftarrow$  pila.pop()
8:       operando2  $\leftarrow$  pila.pop()
9:       resultado  $\leftarrow$  operando1símbolooperando2
10:      pila.push(resultado)
11:   return pila.pop()
```

---

# Evaluar notación postfija

---

## Algorithm Algoritmo para evaluar una expresión en notación postfija

---

```
1: function EVALÚAPOSTFIJA(expresión)
2:   pila ← []
3:   for each símbolo in expresión leyendo de izquierda a derecha do
4:     if símbolo es un número then
5:       pila.push(símbolo)
6:     else if símbolo es un operador then
7:       operando2 ← pila.pop()
8:       operando1 ← pila.pop()
9:       resultado ← operando1símbolooperando2
10:      pila.push(resultado)
11:   return pila.pop()
```

---

# Temas

## 2 Notaciones infija, prefija y postfija

- Infija
- Prefija
- Postfija (o Sufija)
- Evaluación
- Infija a postfija

# Pasar de notación infija a postfija

## Algorithm Algoritmo para pasar de notación infija a postfija

```
1: function INFJAAPREFIJA(expresión)
2:   pila ← []
3:   cola ← []
4:   for each símbolo in expresión leyendo de izquierda a derecha do
5:     if símbolo es un número then cola.queue(símbolo)
6:     else if símbolo = ')' then
7:       while pila.peek() ≠ ')' do cola.queue(pila.pop())
8:         pila.pop()
9:     else if símbolo = '(' then
10:      pila.push(símbolo)
11:    else
12:      prec ← precedencia(símbolo)
13:      while pila ≠ ∅, símbolo ≠ ')' y precedencia(pila.peek()) ≥ prec do
14:        cola.queue(pila.pop())
15:      pila.push(símbolo)
16:    while !pila.empty() do
17:      cola.queue(pila.pop())
18:    return cola
```

▷ Pila de operadores  
▷ Cola con la expresión final



# Implementaciones

- 1 TDA: Pila, Cola y Bicola
- 2 Notaciones infija, prefija y postfija
- 3 Implementaciones
- 4 Bibliografía

# Temas

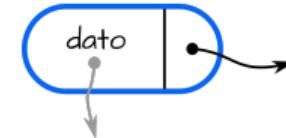
## 3 Implementaciones

- Implementación con referencias
- Implementación en arreglos

# Nodo

- La estructura básica para implementar estructuras usando referencias son objetos de tipo **Nodo**.
- Los nodos son objetos que contienen, esencialmente, dos atributos:
  - Una referencia al **dato** almacenado en la colección.
  - Una referencia al siguiente **nodo**.
- Son un tipo de *estructura recursiva* por contener una referencia a otro objeto del mismo tipo.

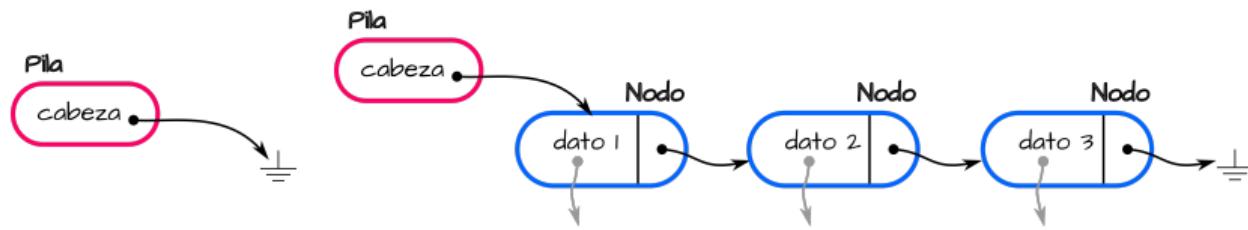
<b>Nodo&lt;E&gt;</b>
-dato: E
-siguiente: Nodo<E>
+Nodo(dato:E)
+Nodo(dato:E,siguiente:Nodo<E>)
+getDato(): E
+setDato(dato:E): void
+getSiguiente(): Nodo<E>
+setSiguiente(sig:Nodo<E>): void



# Pila

- La estructura *Pila* mantiene una referencia al nodo con el elemento a la *cabeza* de la pila.
- Si la pila está vacía esta referencia es null.
- Se insertan elementos (*push*) a la pila poniéndolos a la cabeza.
- Se remueven de la cabeza y se devuelve el contenido del nodo, siempre y cuando aún haya elementos en la pila.

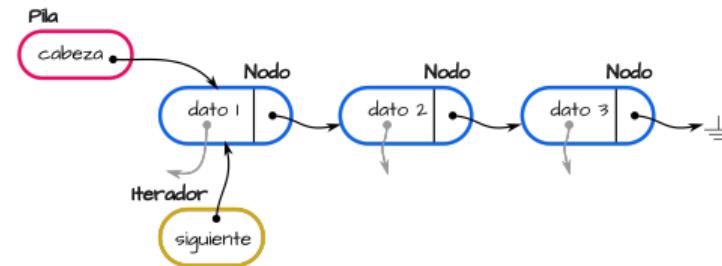
Pila<E>	
-	cabeza: Nodo<E>
+	Pila()
+	empty(): boolean
+	push(e:E): void
+	pop(): E
+	peek(): E



# Iterador

Para una pila con referencias

- La forma más sencilla de implementar un iterador es creando una *clase interna* que implemente la interfaz `Iterator<E>`.
- La clase interna tendrá acceso a los atributos de la clase que la contiene, en el caso de la pila, puede leer el atributo **cabeza**.
- El iterador es un objeto que contiene la referencia al siguiente nodo que devolverá, mientras haya un nodo siguiente.
- Esta referencia se inicializa con la dirección del nodo cabeza.



# Complejidad de las operaciones

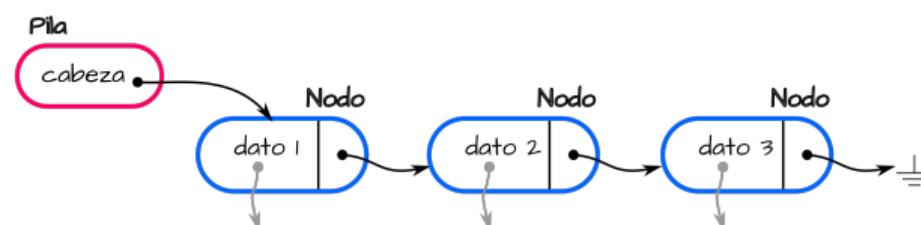
## Pila con referencias

**empty**  $\Theta(1)$

**push**  $\Theta(1)$

**pop**  $\Theta(1)$

**peek**  $\Theta(1)$



# Clase Nodo en C# I

```
1 public class Nodo<E>
2 {
3     private E _dato = default(E)!;
4     private Nodo<E>? _siguiente;
5
6     public E Dato
7     {
8         get { return _dato; }
9         set { this._dato = value; }
10    }
11
12    public Nodo<E>? Siguiente
13    {
14        get { return _siguiente; }
15        set { this._siguiente = value; }
16    }
17
18    public Nodo(E dato)
```

# Clase Nodo en C# II

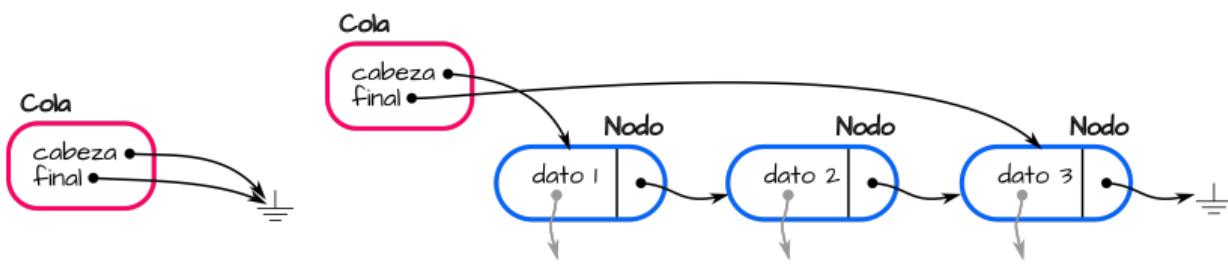
```
19         {
20             this.Dato = dato;
21         }
22
23     public Nodo(E dato, Nodo<E> siguiente)
24     {
25         this.Dato = dato;
26         this.Siguiente = siguiente;
27     }
28
29 }
```

# Cola

## Cola con referencias

- La estructura *Cola* mantiene una referencia al nodo con el elemento a la *cabeza* de la cola y otra al nodo **final**.
- Si la cola está vacía ambas referencias son null.
- Se insertan elementos (*add* o *queue*) a la cola poniéndolos al final.
- Se atienden removiéndolos de la cabeza y se devolviendo el contenido del nodo, siempre y cuando aún haya elementos en la cola.

Cola<E>
-cabeza: Nodo<E>
-final: Nodo<E>
+empty(): boolean
+queue(e:E): void
+dequeue(): E
+peek(): E



# Complejidad de las operaciones

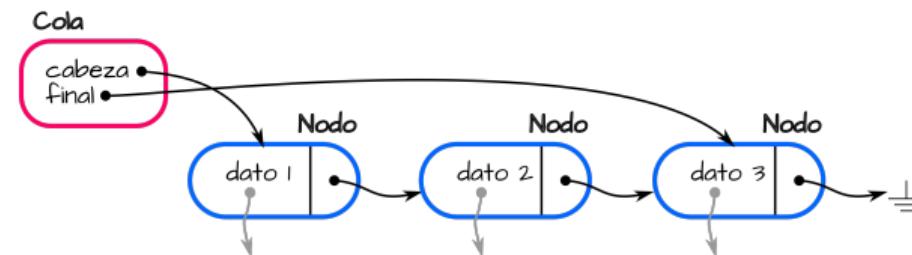
## Cola con referencias

empty  $\Theta(1)$

queue  $\Theta(1)$

dequeue  $\Theta(1)$

peek  $\Theta(1)$



# Temas

## 3 Implementaciones

- Implementación con referencias
- Implementación en arreglos

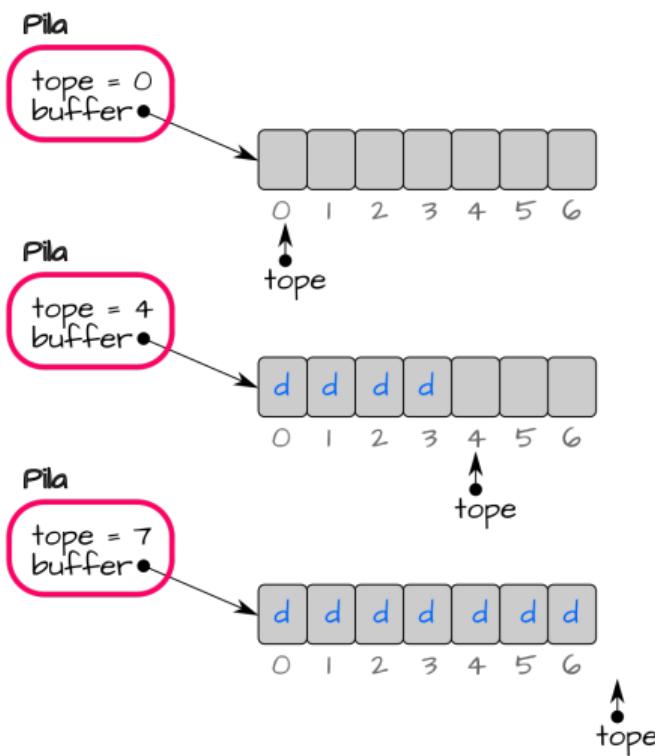
# Implementación en arreglos

- Las **interfaces** públicas son, en principio, idénticas a las de las implementaciones con referencias, pues se trata del mismo TDA.<sup>[1]</sup>
- Se utiliza un arreglo como **buffer** para contener a los elementos.
- Si el arreglo se llena y se desea agregar más elementos es necesario:
  - 1 Crear un nuevo arreglo.
  - 2 Copiar el contenido al arreglo nuevo.
  - 3 Agregar el elemento nuevo.

---

<sup>[1]</sup> Debido a la imposibilidad de crear arreglos a partir de tipos genéricos, en Java los constructores no pueden ser idénticos.

# Pila



# Complejidad de las operaciones

## Pila en arreglo

**empty**  $\mathcal{O}(1)$

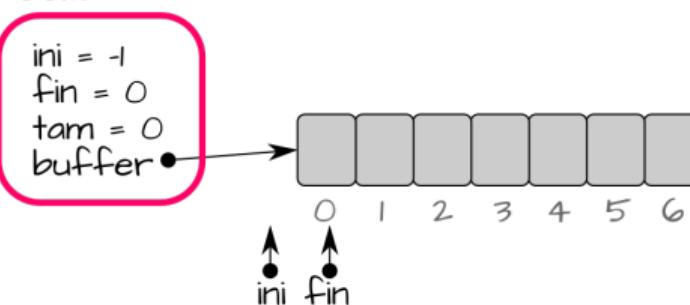
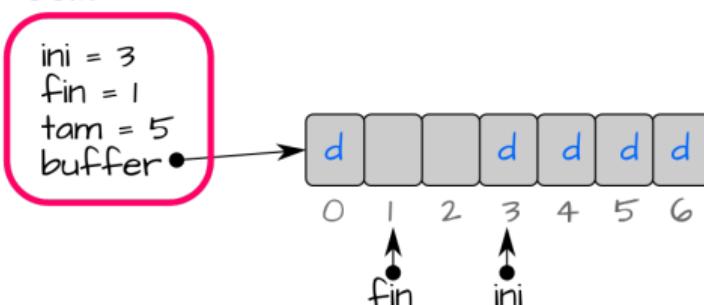
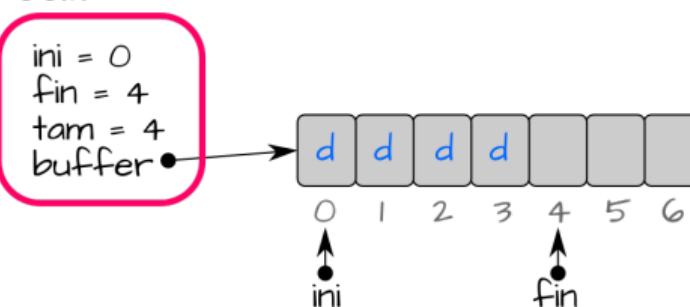
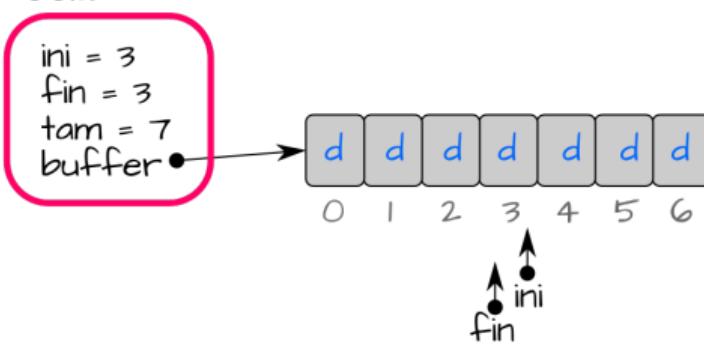
**push**  $\mathcal{O}(1)$ , si hay que modificar el tamaño del arreglo:  $\mathcal{O}(n)$

**pop**  $\mathcal{O}(1)$ , si se elige encoger de nuevo el arreglo cuando está muy vacío:  $\mathcal{O}(n)$ .

OJO: el tamaño para agrandar y enconger deben seguir reglas distintas, o se puede caer en casos donde agregar y remover pocos elementos produzcan altas complejidades por estar ocurriendo en la frontera.

**peek**  $\mathcal{O}(1)$

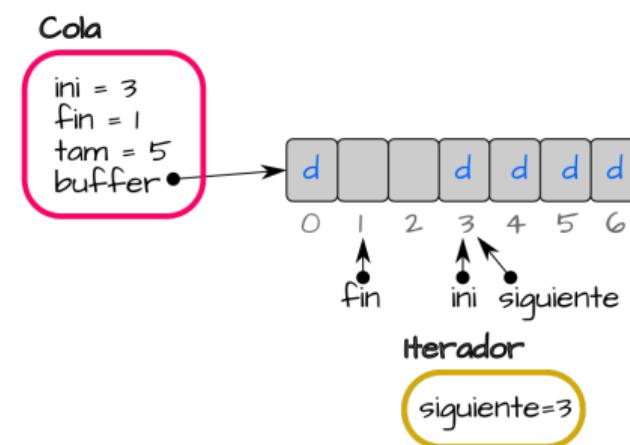
# Cola

**Cola****Cola****Cola****Cola**

# Iterador

Para una cola en arreglo

- El iterador se implementa como una clase interna.
- En su atributo contiene el valor del índice del arreglo en el que se encuentra el siguiente elemento.



# Complejidad de las operaciones

## Cola en arreglo

**empty**  $\mathcal{O}(1)$

**queue**  $\mathcal{O}(1)$ , si hay que modificar el tamaño del arreglo:  $\mathcal{O}(n)$

**dequeue**  $\mathcal{O}(1)$ , si se elige encoger de nuevo el arreglo cuando está muy vacío:  $\mathcal{O}(n)$ .  
Misma nota que para la pila.

**peek**  $\mathcal{O}(1)$

# Bibliografía

- 1 TDA: Pila, Cola y Bicola
- 2 Notaciones infija, prefija y postfija
- 3 Implementaciones
- 4 Bibliografía

# Bibliografía I

-  Preiss, Bruno (1999). *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*. John Wiley & Sons.
-  Vargas Villazón, América, Jorge Lozano Moreno y Guillermo Levine Gutiérrez, eds. (1998). *Estructuras de datos y Algoritmos*. John Wiley & Sons, 438 pp.

# Licencia

Creative Commons  
Atribución-No Comercial-Compartir Igual

