



Programación de sistemas

Pilas y Colas

Departamento de Ingeniería Telemática





Contenidos

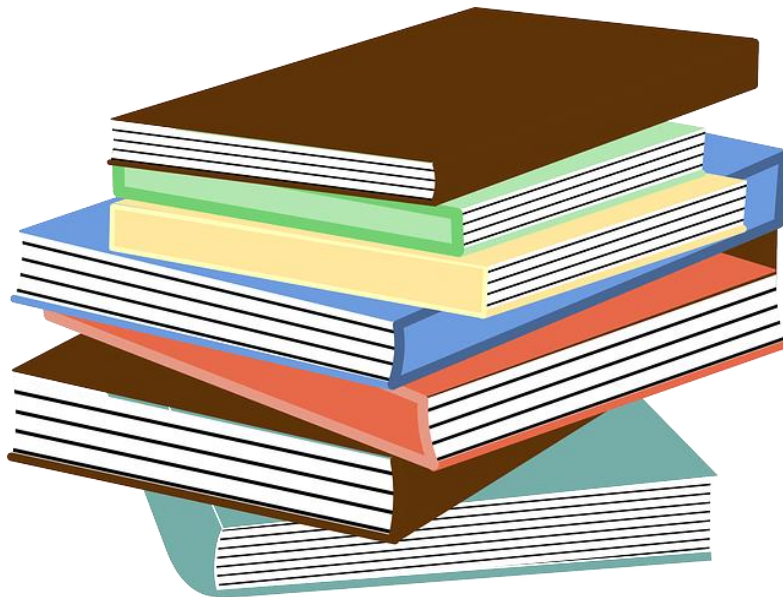
- ❖ Pilas (*stacks*)
- ❖ Colas (*queues*)
- ❖ Colas dobles (*deques – double-ended queues*)





Pilas

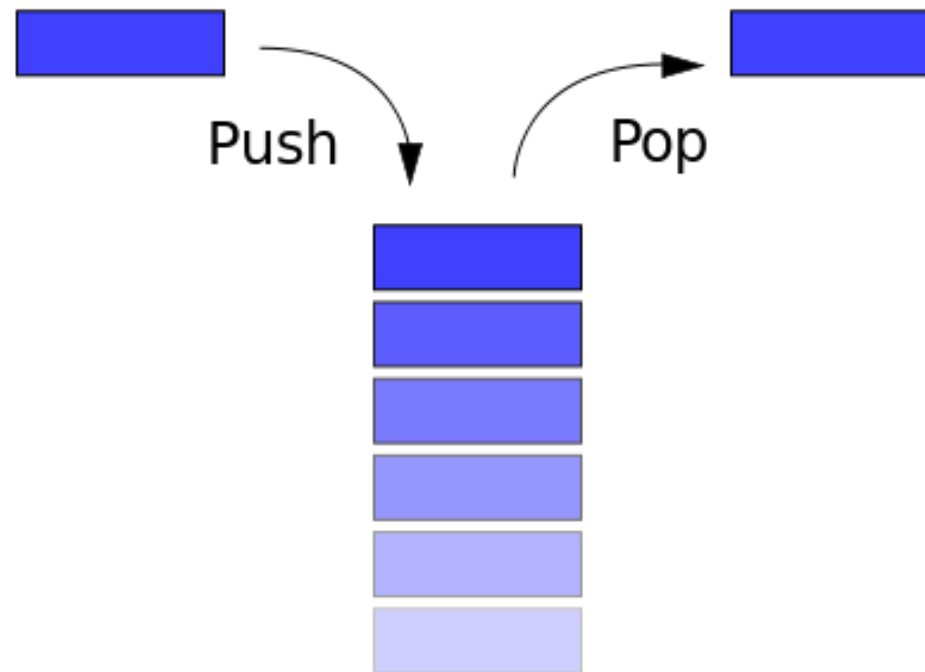
- Estructura de datos lineal
- Inserción y extracción por un único extremo
 - ✓ LIFO (*Last-In-First-Out*)





Pilas

- Insertar por un extremo: `push (x)`
- Extraer por el mismo extremo: `pop ()`





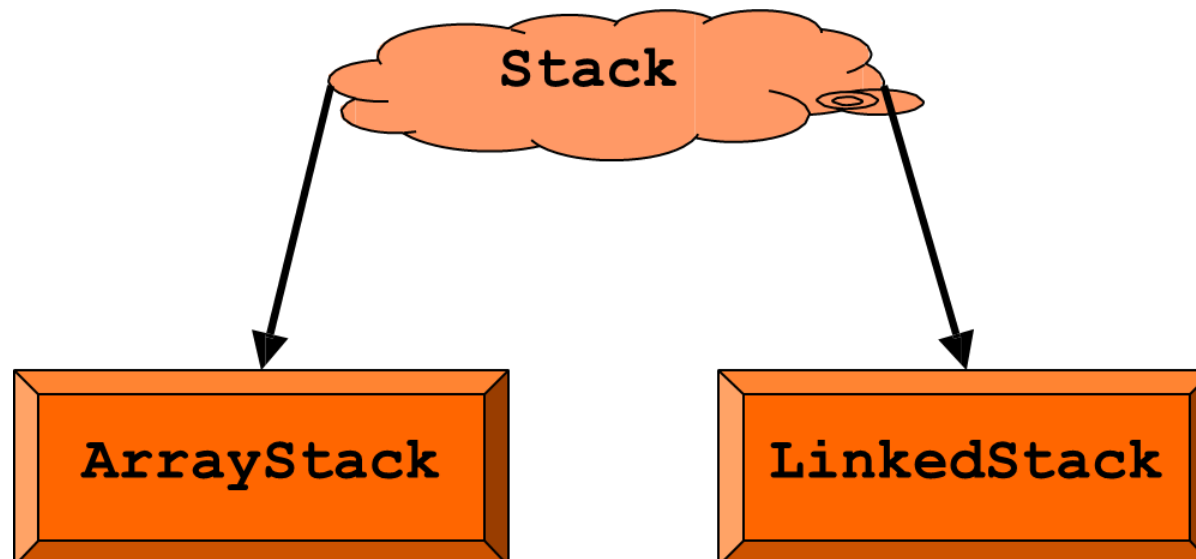
Interfaz para pilas

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



Una interfaz dos implementaciones

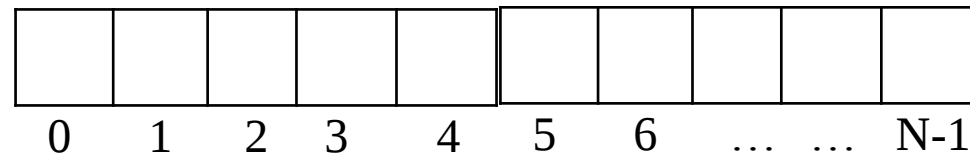
- Implementación basada en arrays:
 - ✓ `ArrayStack`
- Implementación basada en listas enlazadas:
 - ✓ `LinkedStack`





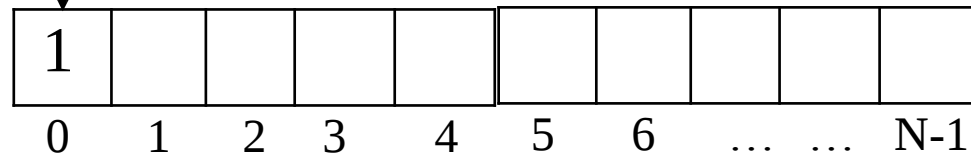
ArrayStack

top
↓



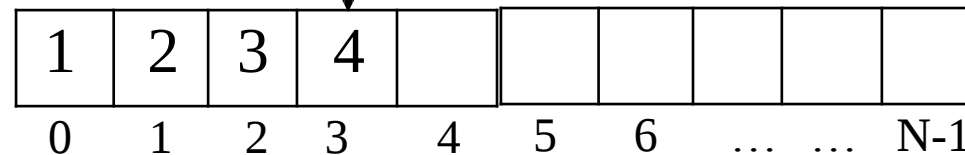
Pila vacía

top
↓



Pila con 1 elemento

top
↓



Pila con 4 elementos





Ejercicio 1

- Crea la clase **ArrayStack**, la cual tendrá tres atributos: el entero **capacity**, el array de Object **data**, y el entero **top** con valor inicial -1.
- Crea el constructor de la clase, el cual recibe un único parámetro que inicializa el atributo **capacity**, y crea un array de datos con esa capacidad.
- Implementa los métodos:
 - **boolean isEmpty()**
 - **int size()**
 - **void push(Object info)**
- *Para casa, implementa los métodos*
 - **Object pop()**
 - **Object top()**

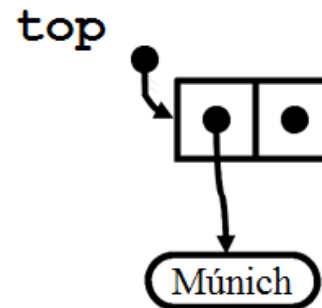




LinkedStack

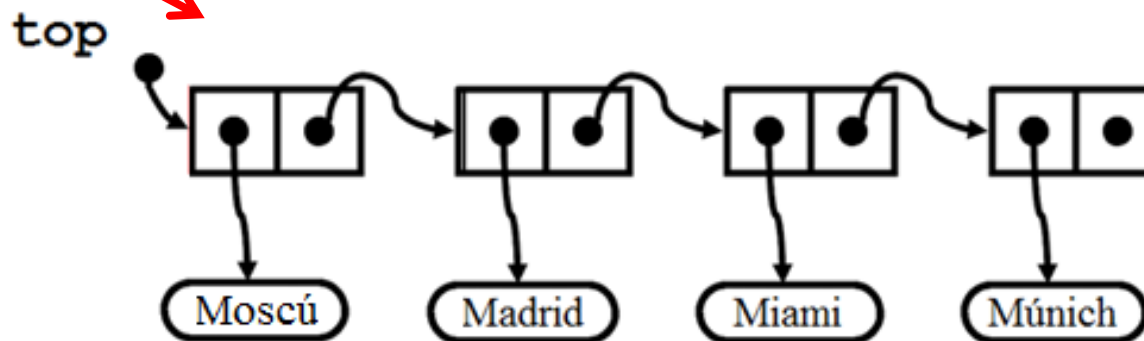
top •

Pila vacía



Pila con 1 elemento

Extremo de
inserción y
extracción



Pila con 4 elementos





Recordando la clase Node

```
public class Node<E> {  
    private E info;  
    private Node<E> next;  
  
    public Node()    {...}  
    public Node(E info)    {...}  
    public Node(E info, Node<E> next)    {...}  
  
    public Node<E> getNext() {...}  
    public void setNext(Node<E> next) {...}  
    public E getInfo() {...}  
    public void setInfo(E info) {...}  
}
```





LinkedList (I)

```
public class LinkedList<E> implements Stack<E> {  
    private Node<E> top;  
    private int size;  
    public LinkedList() {  
        top = null;  
        size = 0;  
    }
```

Atributos

Constructor

```
    public boolean isEmpty() {  
        return (top == null);  
    }
```

```
    public int size() {  
        return size;  
    }
```

```
    public E top() {  
        if(isEmpty()){  
            return null;  
        }  
        return top.getInfo();  
    }
```

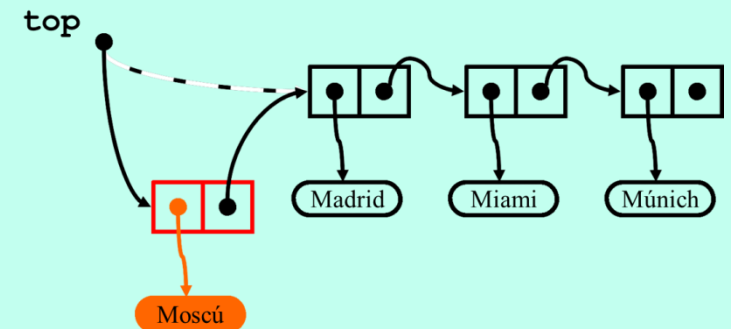
**Métodos de la interfaz
Stack a implementar (I)**



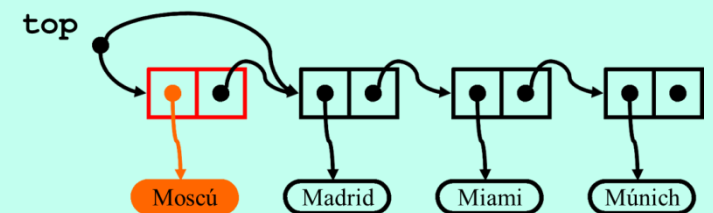
LinkedList (II)

```
...  
public void push(E info){  
    Node<E> n = new Node<E>(info, top);  
    top = n;  
    size++;  
}
```

```
public E pop() {  
    E info;  
    if(isEmpty()){  
        return null;  
    } else{  
        info = top.getInfo();  
        top = top.getNext();  
        size--;  
        return info;  
    }  
}
```



**Métodos de la interfaz
Stack a implementar (II)**





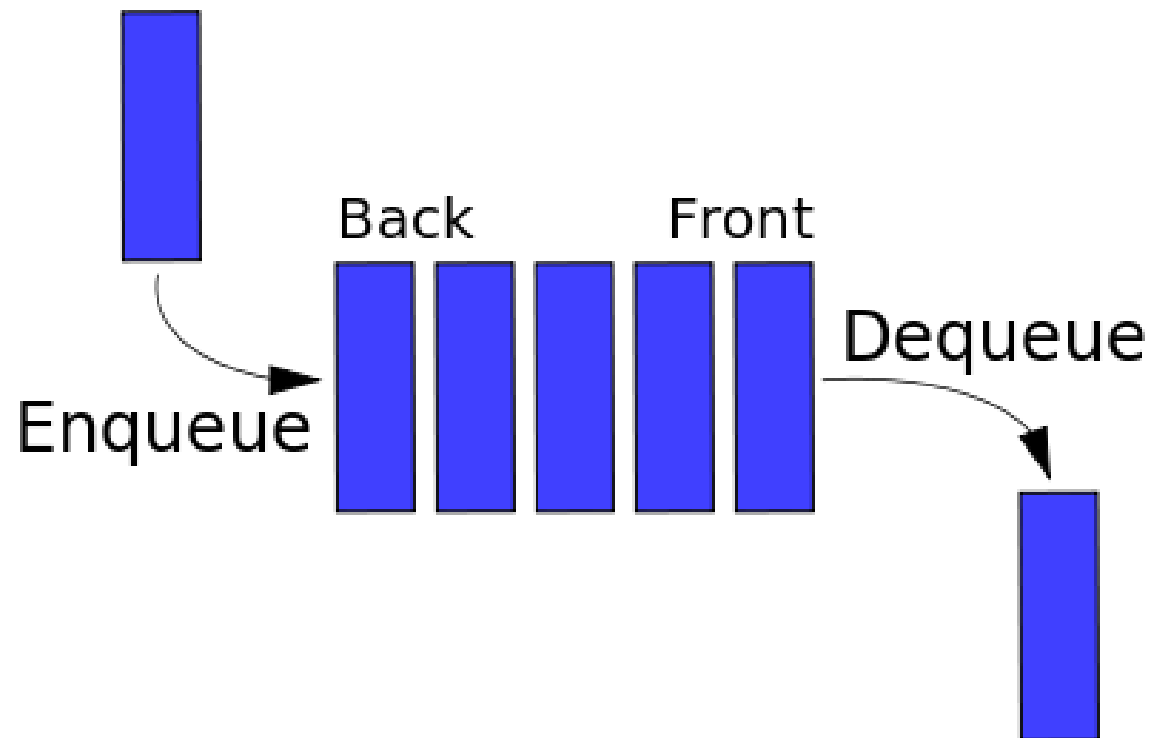
Colas

- Estructura de datos lineal
- Inserción por un extremo y extracción por el extremo opuesto
 - ✓ FIFO (*First-In-First-Out*)



Colas

- Insertar por un extremo: enqueue (x)
- Extraer por el extremo opuesto: dequeue ()





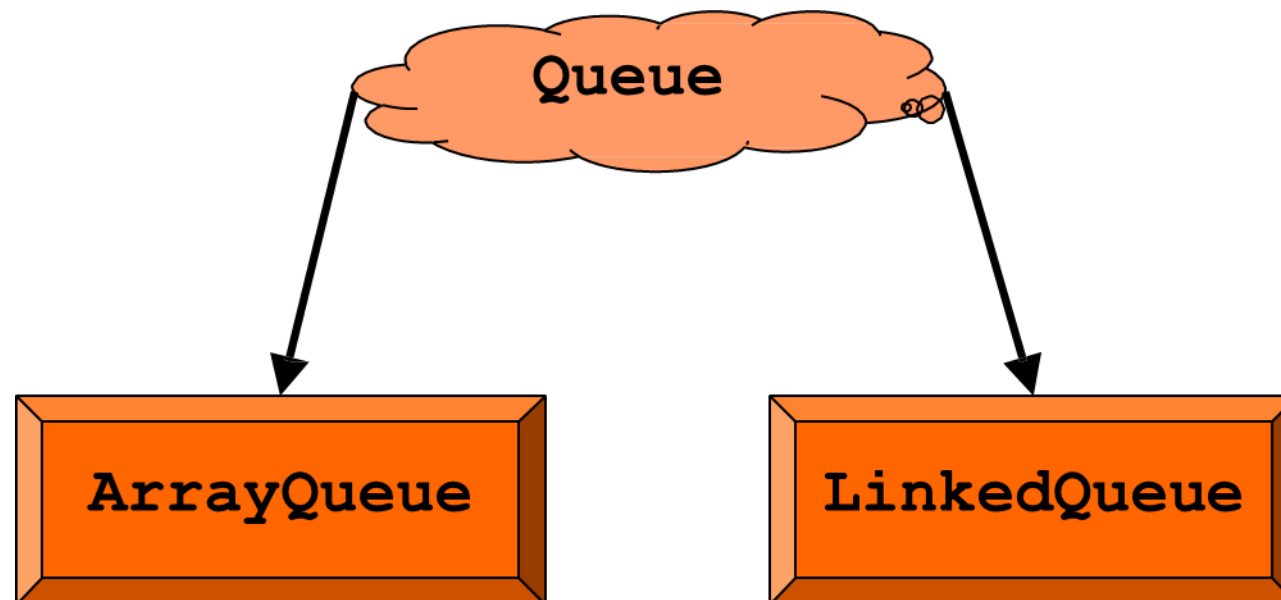
Interfaz para colas

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



Una interfaz dos implementaciones

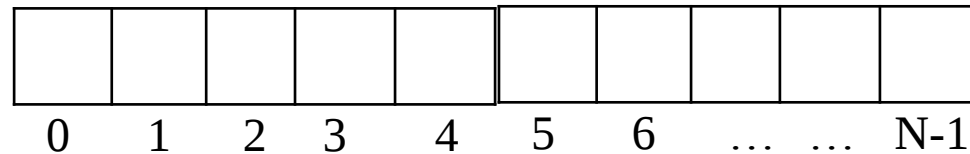
- Implementación basada en arrays:
 - ✓ `ArrayQueue`
- Implementación basada en listas enlazadas:
 - ✓ `LinkedList`





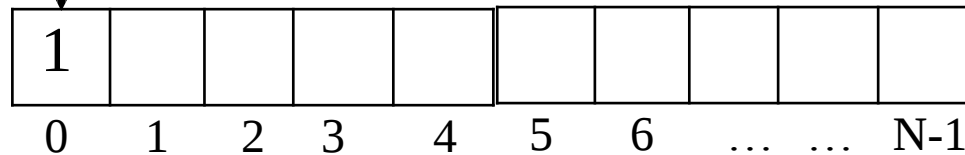
ArrayQueue

top tail



Cola vacía

top tail

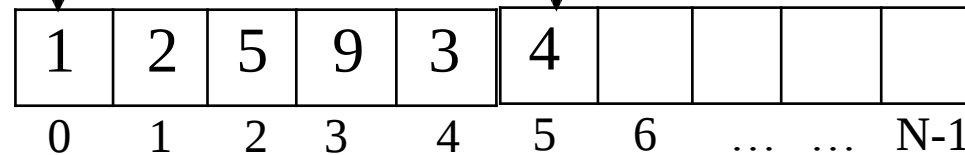


Insertamos 1 elemento

top



tail

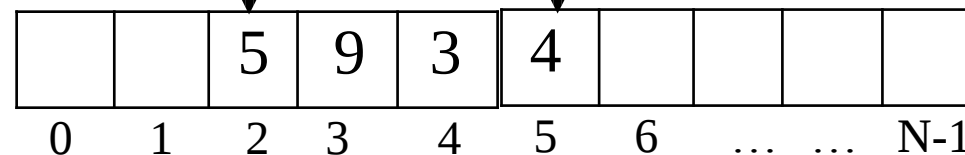


Insertamos 5
elementos más

top



tail



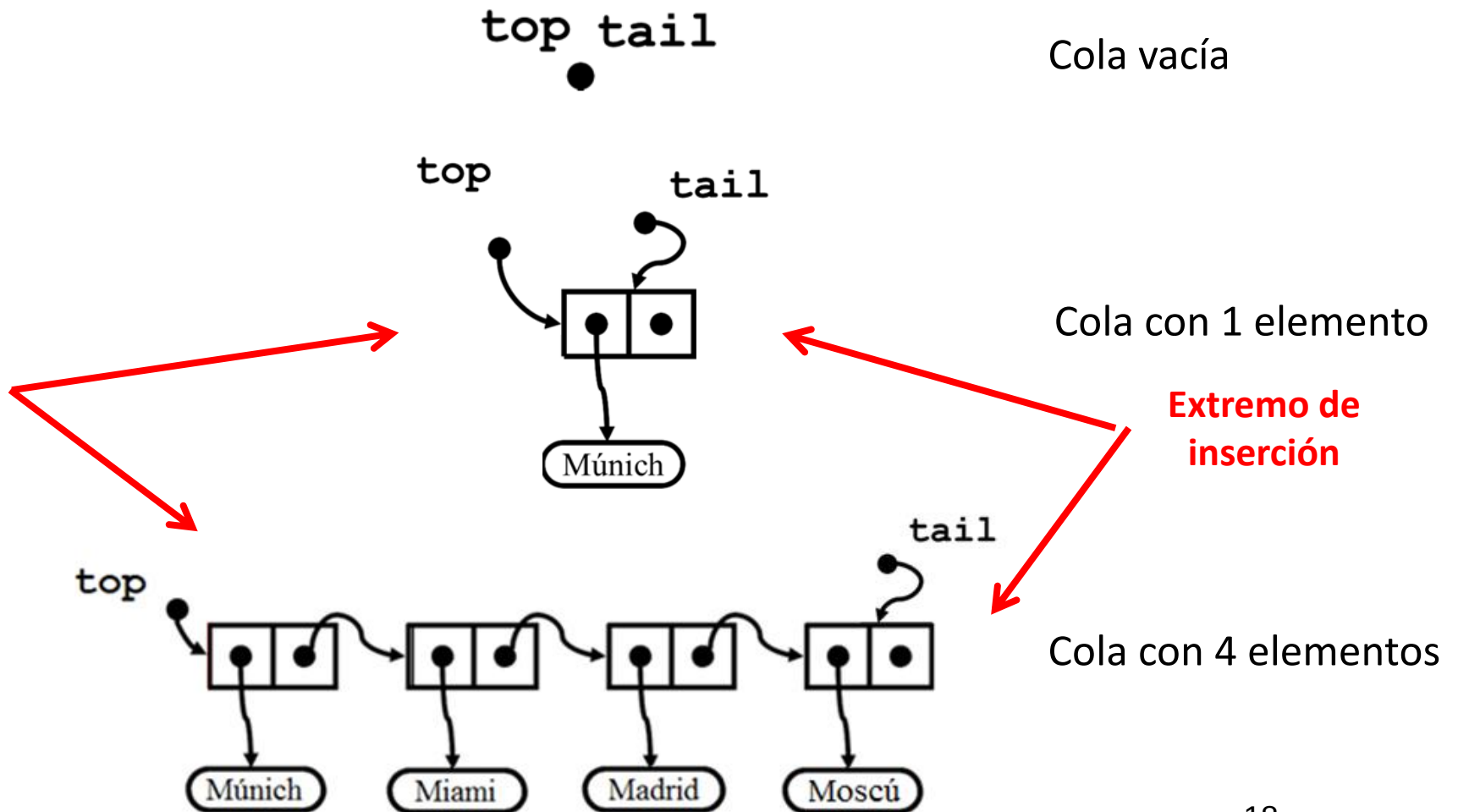
Extraemos dos
elementos



LinkedQueue

Extremo de
extracción

Extremo de
inserción





LinkedList (I)

```
public class LinkedList<E> implements Queue<E> {  
    private Node<E> top = null;  
    private Node<E> tail = null;  
    private int size = 0;  
    public LinkedList(){  
        top = null;  
        tail = null;  
        size = 0;  
    }  
  
    public boolean isEmpty() {  
        return (top == null);  
    }  
  
    public int size() {  
        return size;  
    }  
  
    public E front() {  
        if (isEmpty()){  
            return null;  
        } else {  
            return top.getInfo();  
        }  
    }  
}
```

Atributos

Constructor

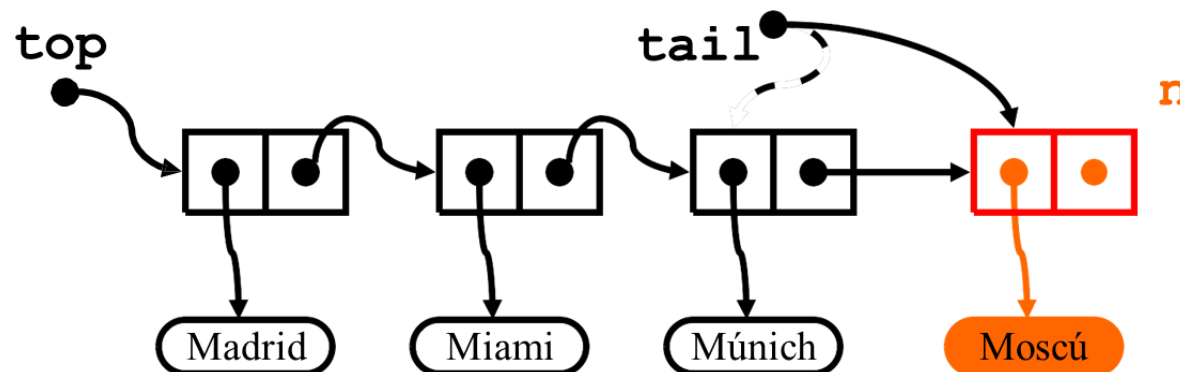
Métodos de la interfaz
Queue a implementar (I)



LinkedList (II)

```
...  
public void enqueue (E info){  
    Node<E> n = new Node<E>(info, null);  
    if (isEmpty()){  
        top = n;  
    } else{  
        tail.setNext(n);  
    }  
    tail = n;  
    size++;  
}
```

Métodos de la interfaz
Queue a implementar (II)



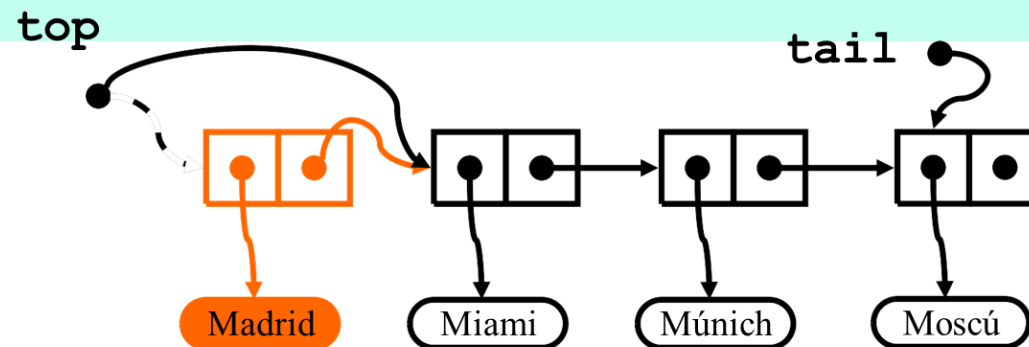


LinkedList (III)

...

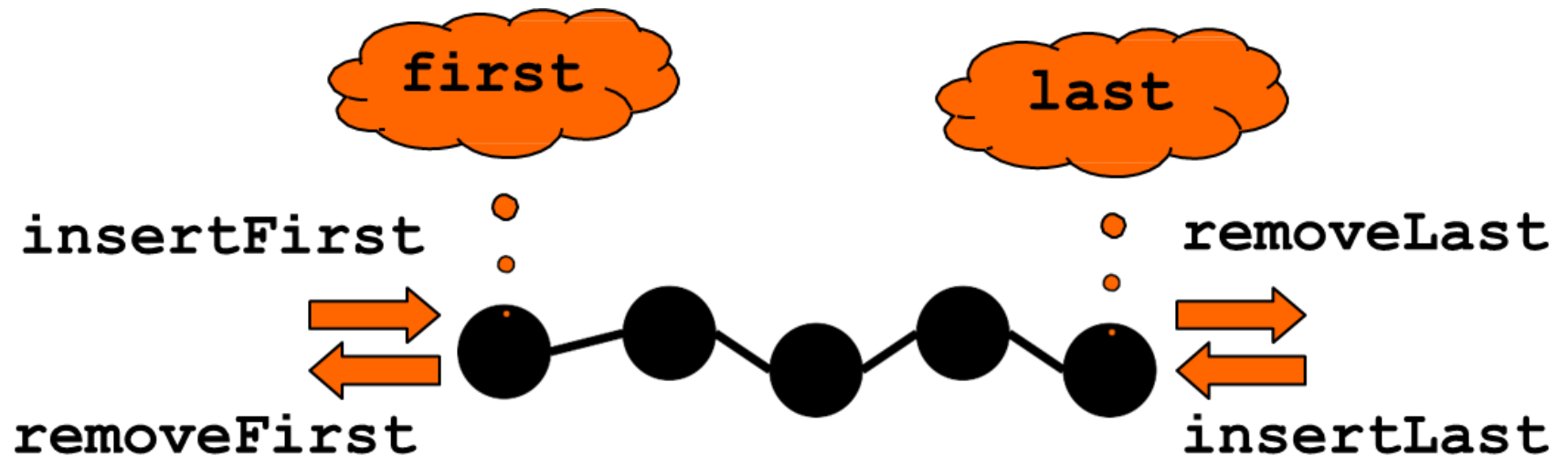
```
public E dequeue(){  
    E info;  
    if (isEmpty()){  
        info = null;  
    } else{  
        info = top.getInfo();  
        top = top.getNext();  
        size--;  
        if (isEmpty()){  
            tail = null;  
        }  
    }  
    return info;  
}
```

Métodos de la interfaz
Queue a implementar (III)



Colas dobles (*deques*)

- Estructura de datos lineal
 - *Deque (double-ended queue)*
- Inserción y extracción por cualquiera de los extremos





Interfaz para colas dobles

```
public interface Deque<E> {  
    public boolean isEmpty();  
    public int size();  
    public E first();  
    public E last();  
    public void insertFirst(E info);  
    public void insertLast(E info);  
    public E removeFirst();  
    public E removeLast();  
}
```





Interfaz para colas dobles

Stack	Deque
<code>size()</code>	<code>size()</code>
<code>isEmpty()</code>	<code>isEmpty()</code>
<code>top()</code>	<code>last()</code>
<code>push(x)</code>	<code>insertLast(x)</code>
<code>pop()</code>	<code>removeLast()</code>

Queue	Deque
<code>size()</code>	<code>size()</code>
<code>isEmpty()</code>	<code>isEmpty()</code>
<code>front()</code>	<code>first()</code>
<code>enqueue(x)</code>	<code>insertLast(x)</code>
<code>dequeue()</code>	<code>removeFirst()</code>





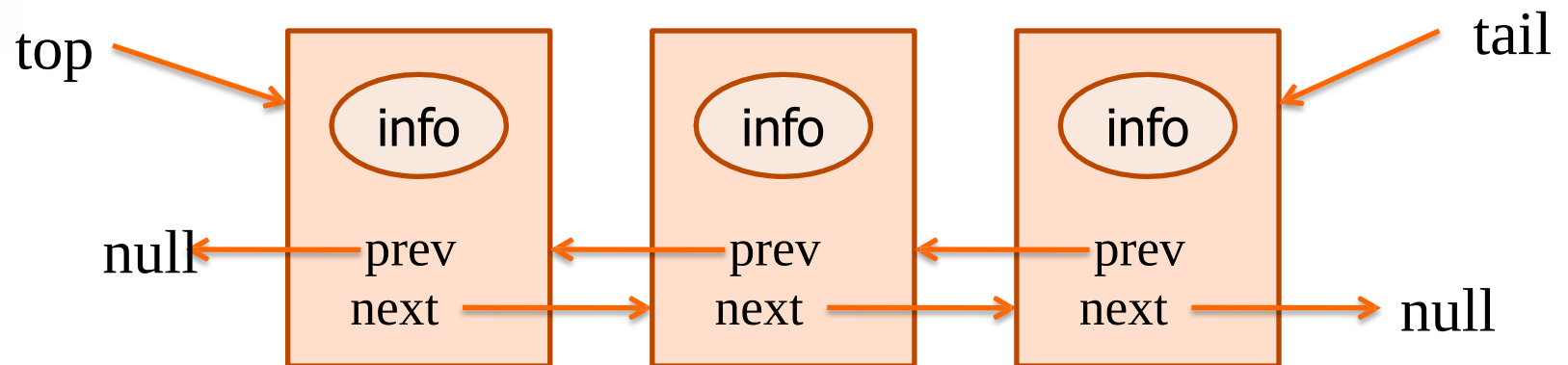
Implementación de colas dobles

- Las listas enlazadas no son apropiadas porque **removeLast** necesita recorrer la lista completa para obtener la referencia del penúltimo
- Solución: **listas doblemente enlazadas**



Listas doblemente enlazadas

- Listas enlazadas en que cada nodo, además de almacenar el dato y la referencia del siguiente nodo, almacena también la referencia del nodo anterior
 - Permiten recorrer la lista en ambos sentidos
 - Reducen el coste de extracción del último nodo





La classe DLNode

```
public class DLNode<E> {  
    private E info;  
    private DLNode<E> prev;  
    private DLNode<E> next;  
  
    public DLNode() {...}  
    public DLNode(E info) {...}  
    public DLNode(E info, DLNode<E> prev, DLNode<E> next) {...}  
  
    public DLNode<E> getNext() {...}  
    public void setNext(DLNode<E> next) {...}  
    public DLNode<E> getPrev() {...}  
    public void setPrev(DLNode<E> prev) {...}  
    public E getInfo() {...}  
    public void setInfo(E info) {...}  
}
```



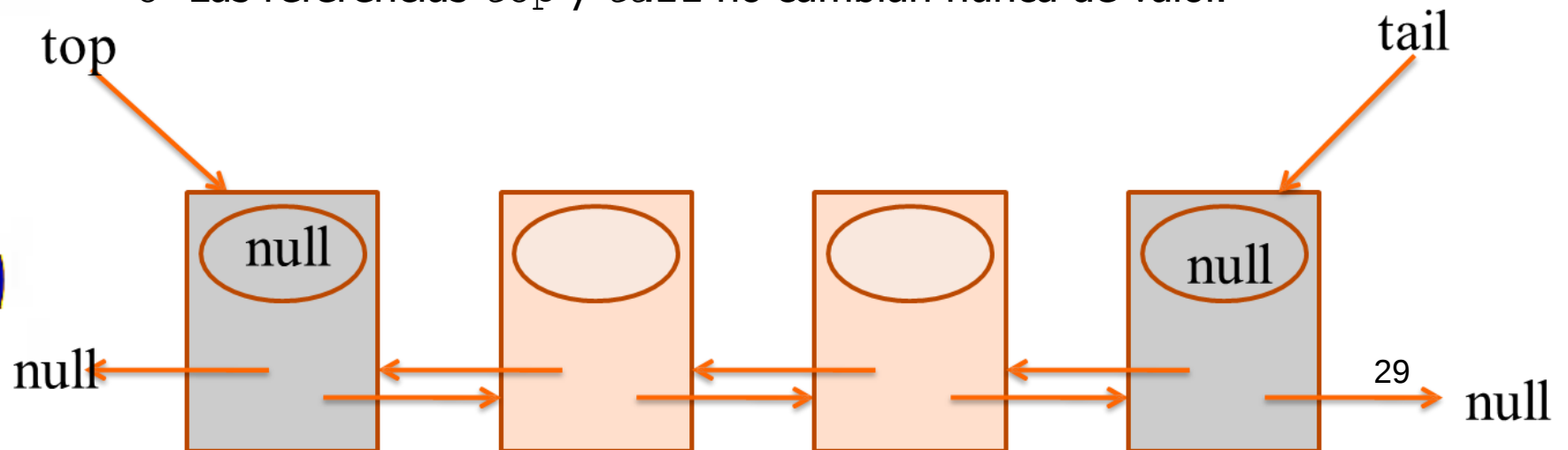
Ejercicio 2

- Completa el código de la clase **DLNode**. Incluye tres constructores: uno que no recibe información para inicializar ningún atributo; otro que permite inicializar el atributo **info**, y otro que permite inicializar los tres atributos



Listas doblemente enlazadas

- La implementación de colas dobles basada en listas enlazadas se complica debido a la necesidad de comprobar siempre que existen los nodos anterior y posterior
- **Simplificación:** Crear dos nodos especiales, sin datos, de tal forma que uno esté siempre al principio y el otro siempre al final:
 - Una lista vacía sólo contiene estos dos nodos.
 - Está garantizado en cualquier operación de inserción o extracción que siempre existen el nodo anterior y siguiente.
 - Las referencias `top` y `tail` no cambian nunca de valor.





La clase cola doble (DLDeque) con listas doblemente enlazadas

```
public class DLDeque<E> implements Deque<E>{  
    private DLNode<E> top;  
    private DLNode<E> tail;  
    private int size;
```

Atributos

```
    public DLDeque() {  
        top = new DLNode<E>();  
        tail = new DLNode<E>();  
        tail.setPrev(top);  
        top.setNext(tail);  
        size = 0;  
    }
```

Constructor

...



Ejercicio 3

- Implementa los siguientes métodos de la clase

DLDeque:

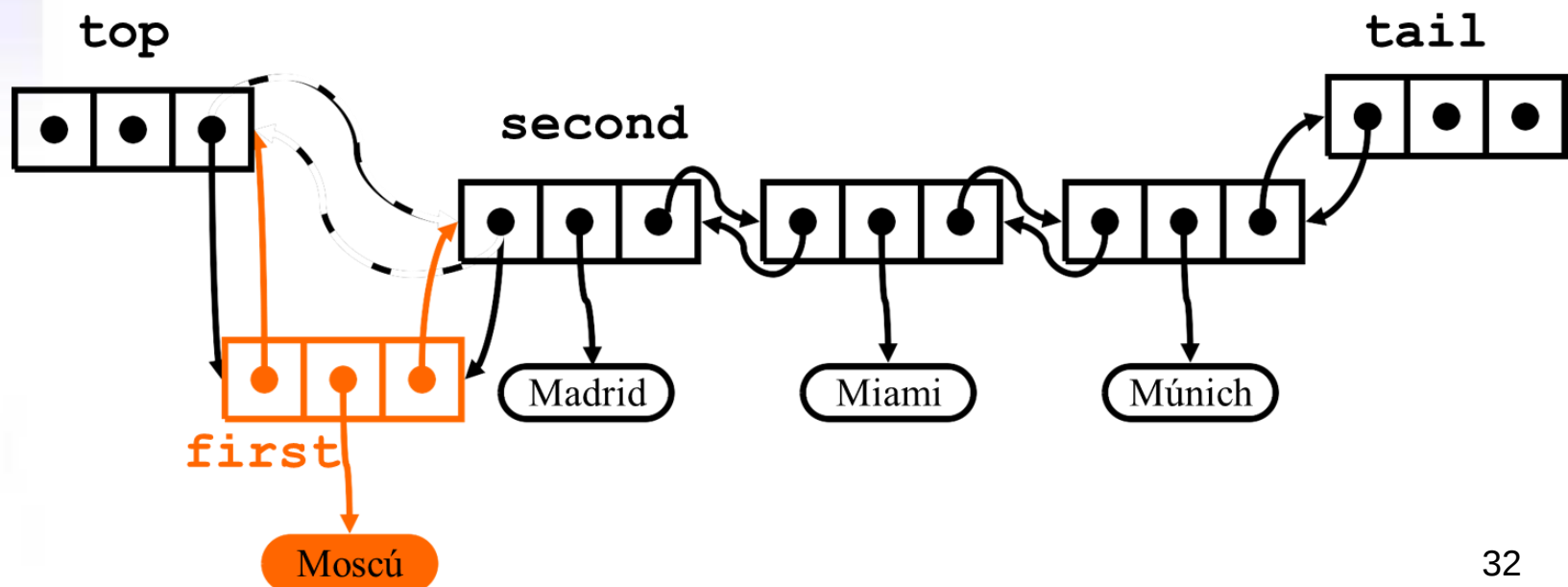
- `boolean isEmpty()`
- `int size()`
- `E first()`
- `E last()`





La clase cola doble (DLDeque) con listas doblemente enlazadas: Inserción por el principio

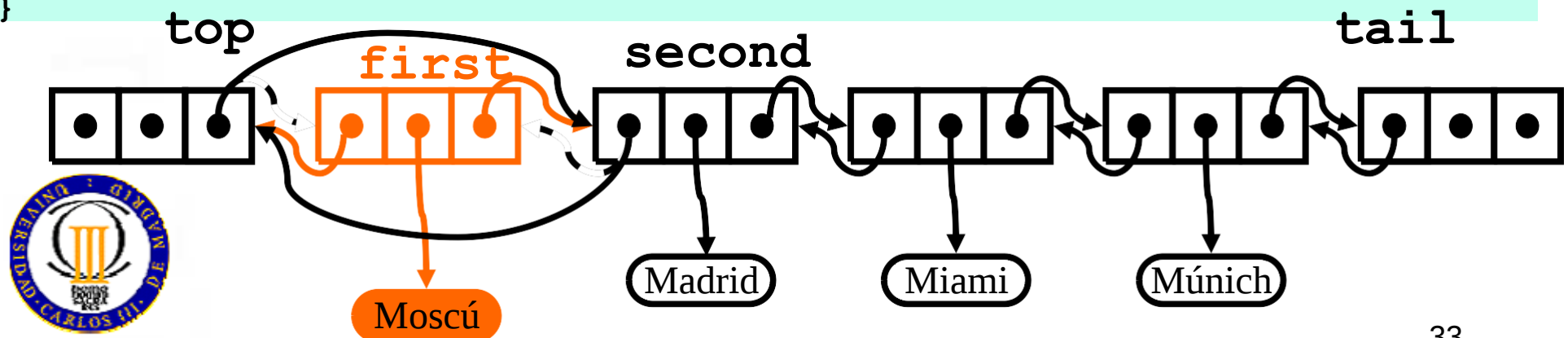
```
public void insertFirst(E info) {  
    DLNode<E> second = top.getNext();  
    DLNode<E> first = new DLNode<E>(info, top, second);  
    second.setPrev(first);  
    top.setNext(first);  
    size++;  
}
```





La clase cola doble (DLDeque) con listas doblemente enlazadas: Extracción por el principio

```
public E removeFirst() {  
    E info;  
    if (top.getNext() == tail){  
        info = null;  
    } else {  
        DLNode<E> first = top.getNext();  
        info = first.getInfo();  
        DLNode<E> second = first.getNext();  
        top.setNext(second);  
        second.setPrev(top);  
        size--;  
    }  
    return info;  
}
```





Ejercicio 4

- Implementa los siguientes métodos de la clase **DLDeque**:
 - `void insertLast (E info)`
- *Para casa,*
 - `E removeLast()`

