Systems Programming

Stacks and Queues

Departamento de Ingeniería Telemática

Contents

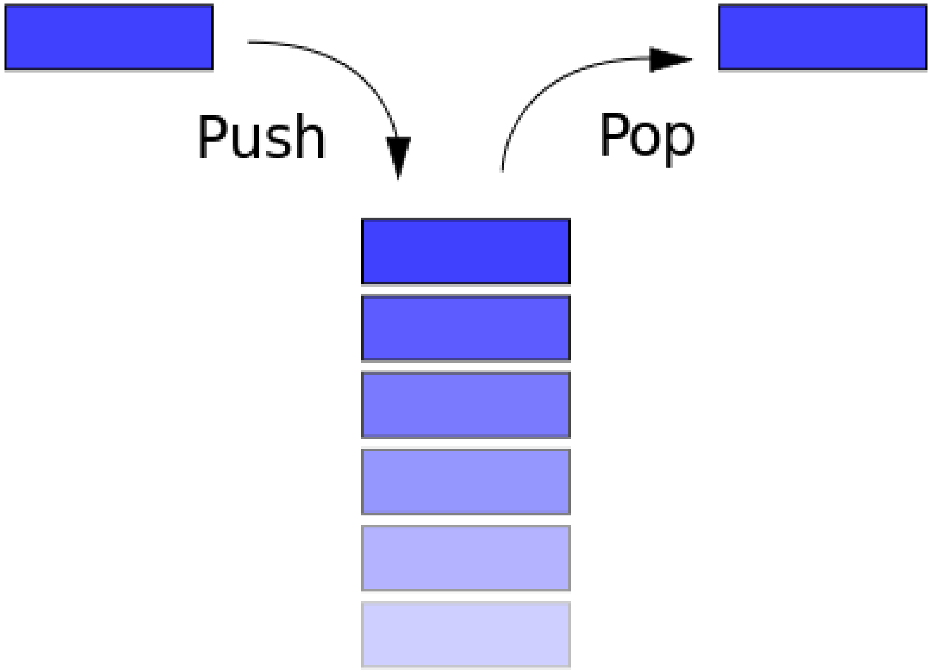
- Stacks
- Queues
- Deques – double-ended queues

Stacks

- Linear data structures
- Insertion and extraction into/from the (same) end
  - ✓ LIFO (*Last-In-First-Out*)

Stacks

- Insert into one end: `push(x)`
- Extract from the same end: `pop()`

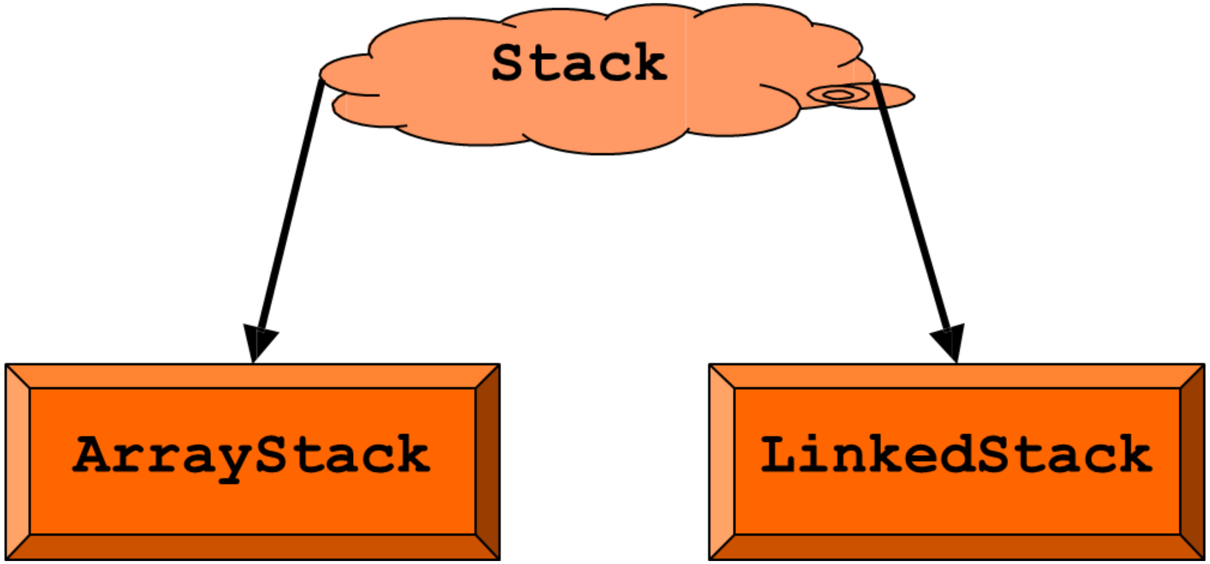


Interface for stacks

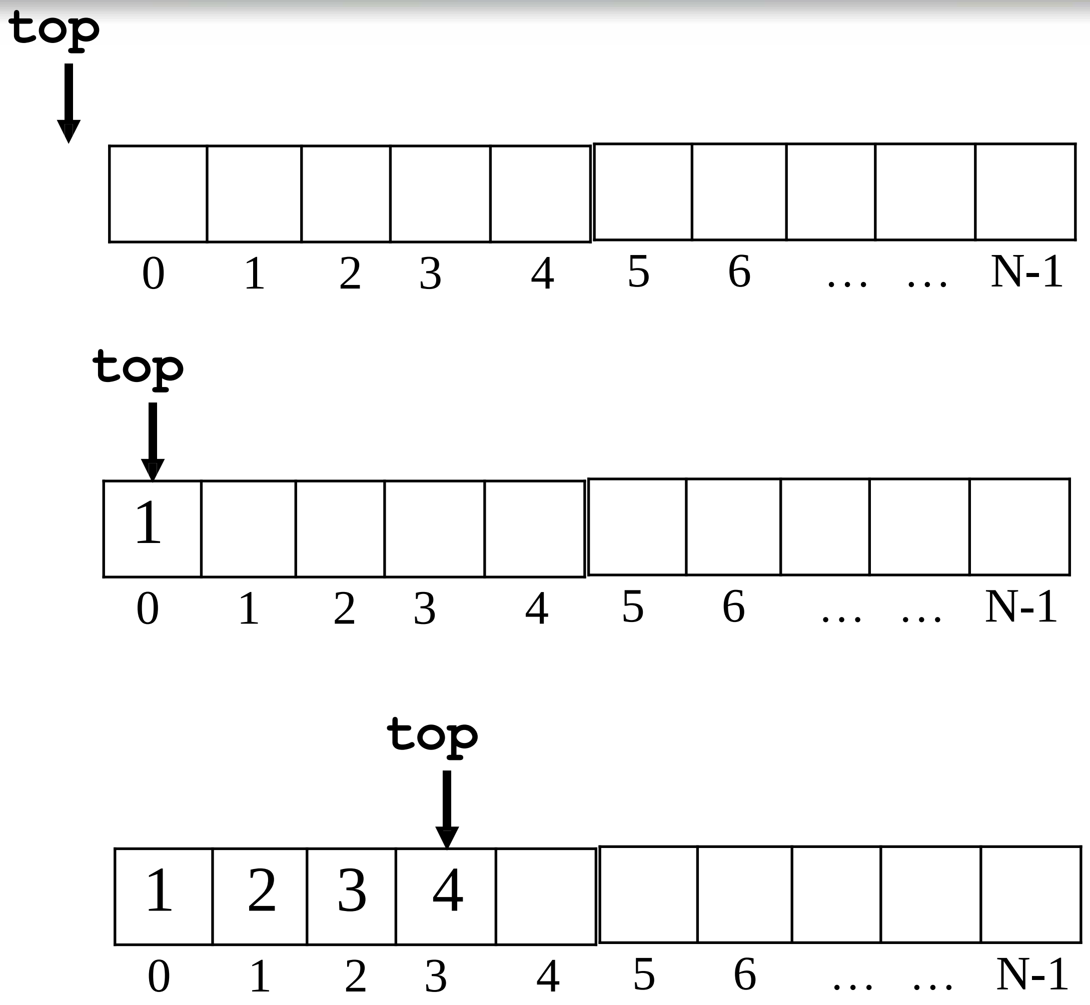
```
public interface Stack<E> {
    boolean isEmpty();
    int size();
    E top();
    void push(E info);
    E pop();
}
```

One interface, two implementations

- Array-based implementation:
  - ✓ `ArrayStack`
- Linked-list-based implementation:
  - ✓ `LinkedStack`



ArrayStack



top

0	1	2	3	4	5	6	…	…	N-1

Empty stack

top

0	1	2	3	4	5	6	…	…	N-1
1									

Stack with 1 element

top

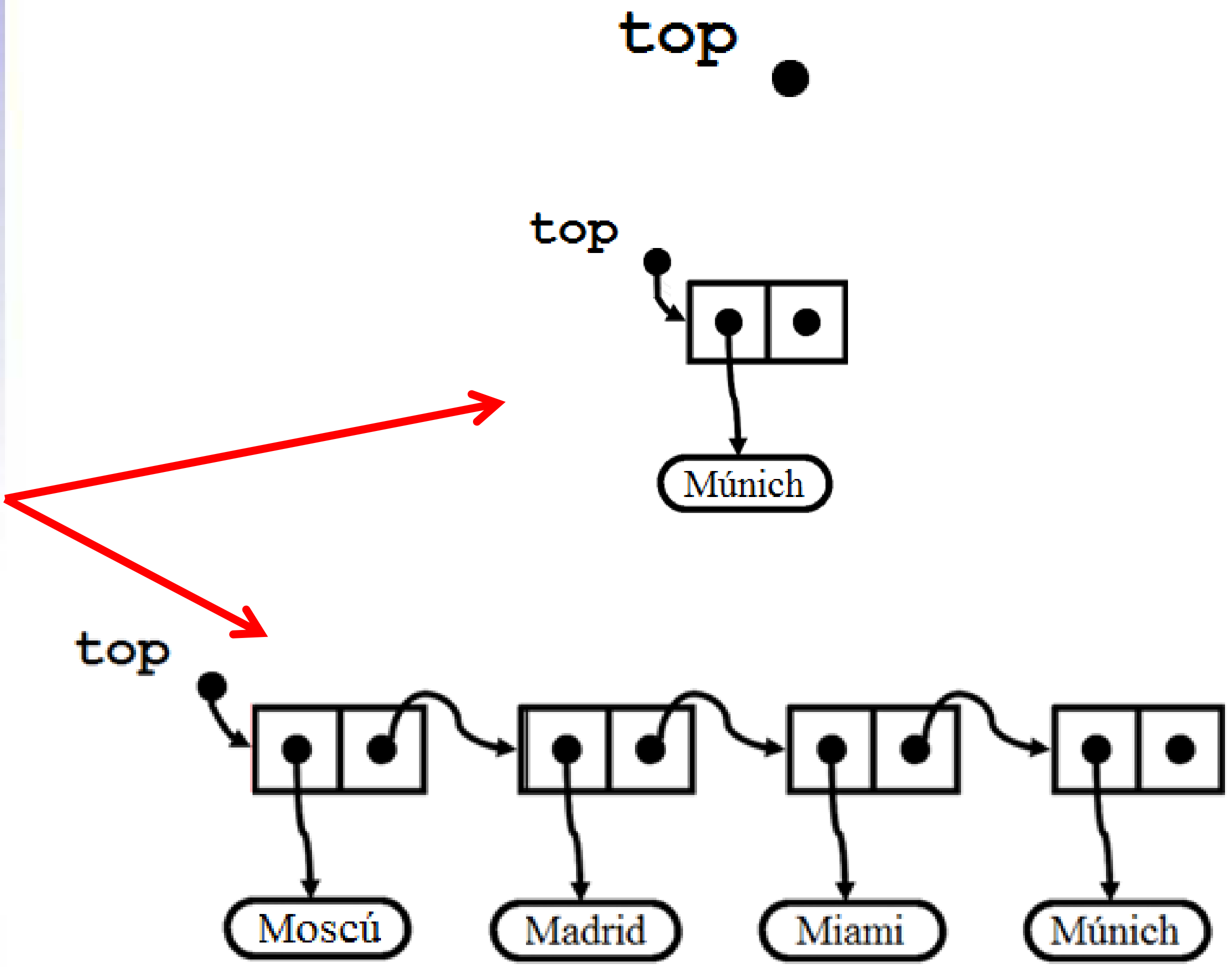
0	1	2	3	4	5	6	…	…	N-1
1	2	3	4						

Stack with 4 elements

Exercise 1

- Create the **ArrayStack** class, with three attributes: int **capacity**, the Object **data** array and int **top** with -1 as initial value.
- Create the class constructor, which takes just one parameter to initialise the **capacity** attribute and creates an array of such capacity.
- Implement the following methods:
  - ○ **boolean isEmpty()**
  - ○ **int size()**
  - ○ **void push(Object info)**
- *Homework: implement these methods:*
  - ○ ***Object pop()***
  - ○ ***Object top()***

LinkedStack



Empty stack

Stack with 1 element

Stack with 4 elements

End for insertion and extraction

Remembering the Node class

```
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) {…}

}
```

LinkedStack (I)

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

    public boolean isEmpty() {
        return (size == 0);
    }

    public int size() {
        return size;
    }

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

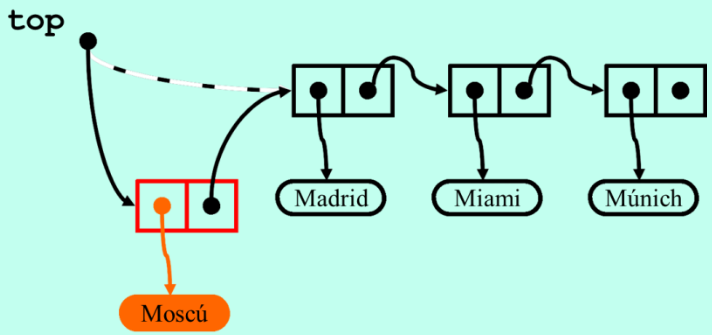
Attributes

Constructor

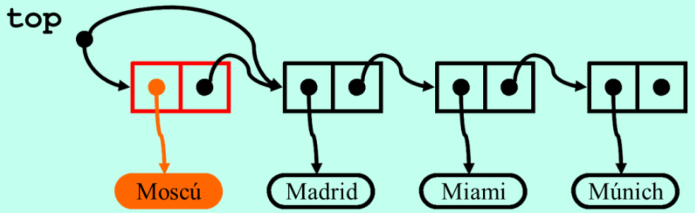
Stack interface methods to implement (I)

LinkedStack (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;
        }
    }
}
```



Stack interface methods to implement (II)

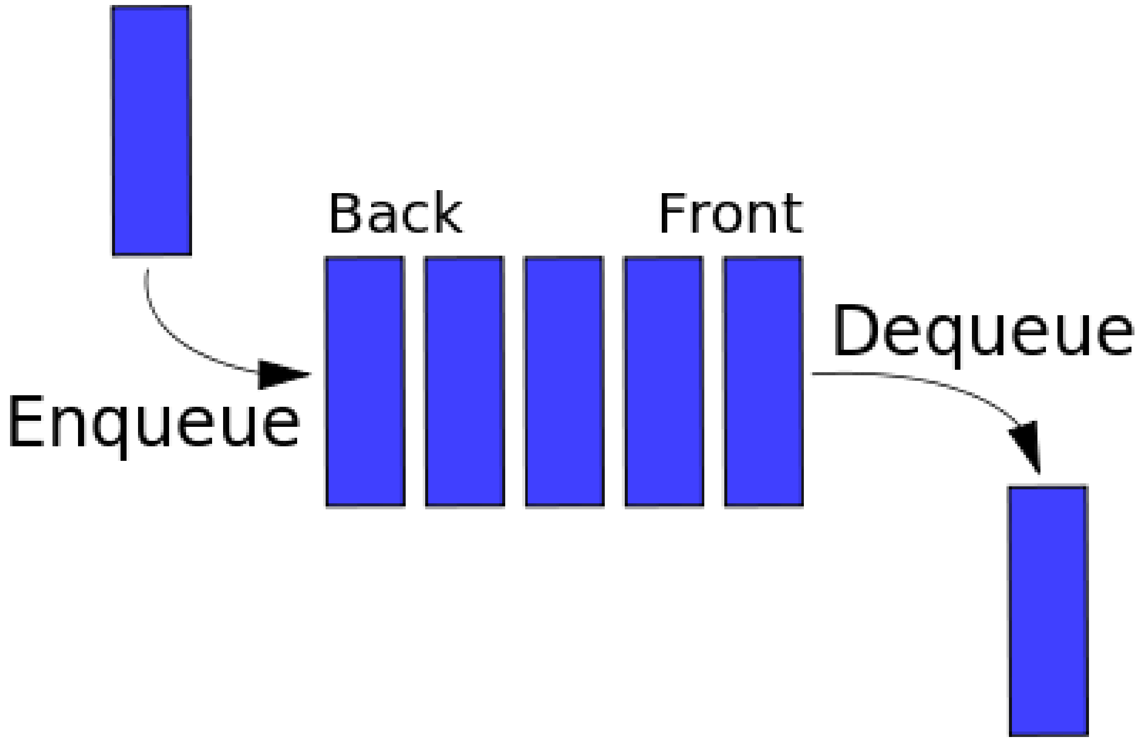


Queues

- Linear data structures
- Insertion into one end and extraction from the opposite end
  - ✓ FIFO (*First-In-First-Out*)

Queues

- Insert into one end: `enqueue(x)`
- Extract from the opposite end: `dequeue()`

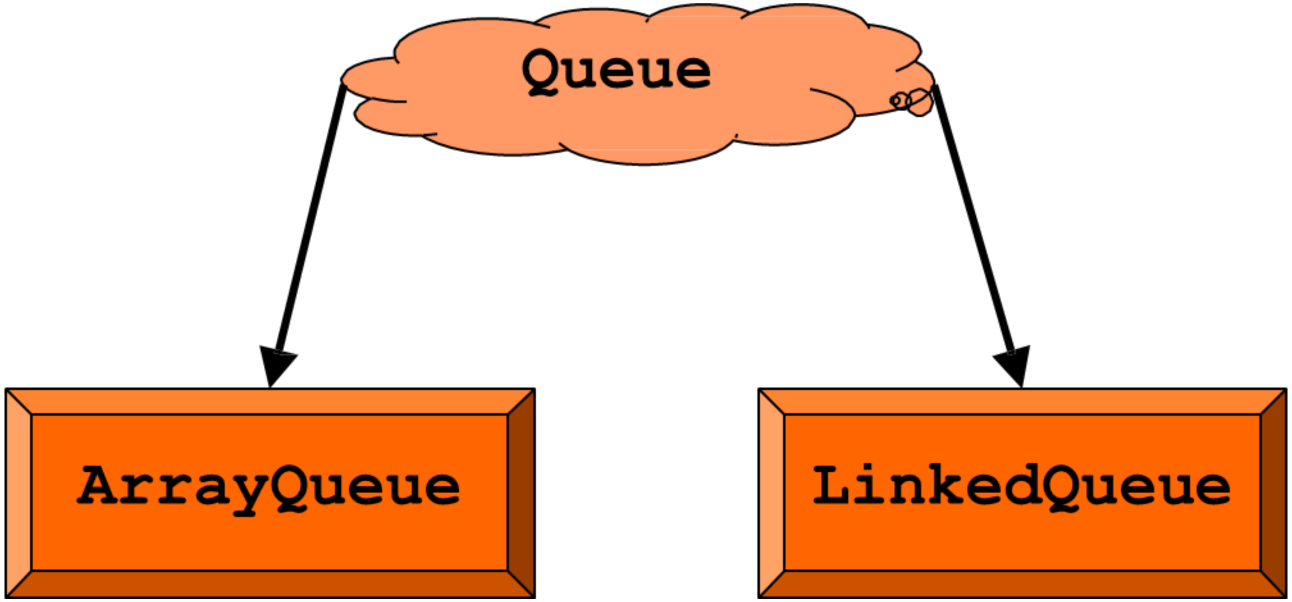


Interface for queues

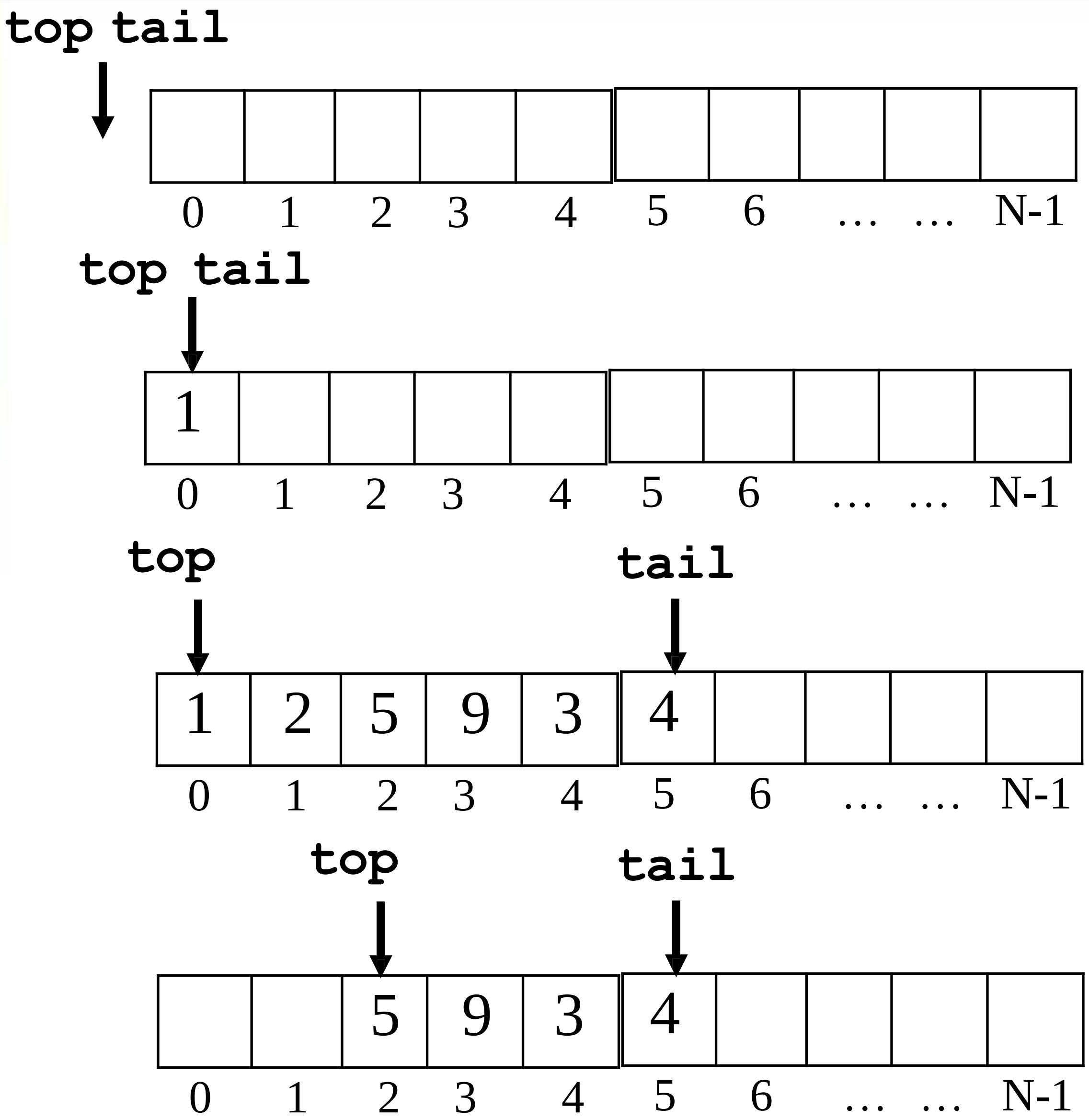
```
public interface Queue<E> {
    boolean isEmpty();
    int size();
    E front();
    void enqueue (E info);
    E dequeue();
}
```

One interface, two implementations

- Array-based implementation:
  - ✓ `ArrayQueue`
- Linked-list-based implementation:
  - ✓ `LinkedQueue`



ArrayQueue



0	1	2	3	4	5	6	…	…	N-1

Empty queue

0	1	2	3	4	5	6	…	…	N-1
1									

Insertion of 1 element

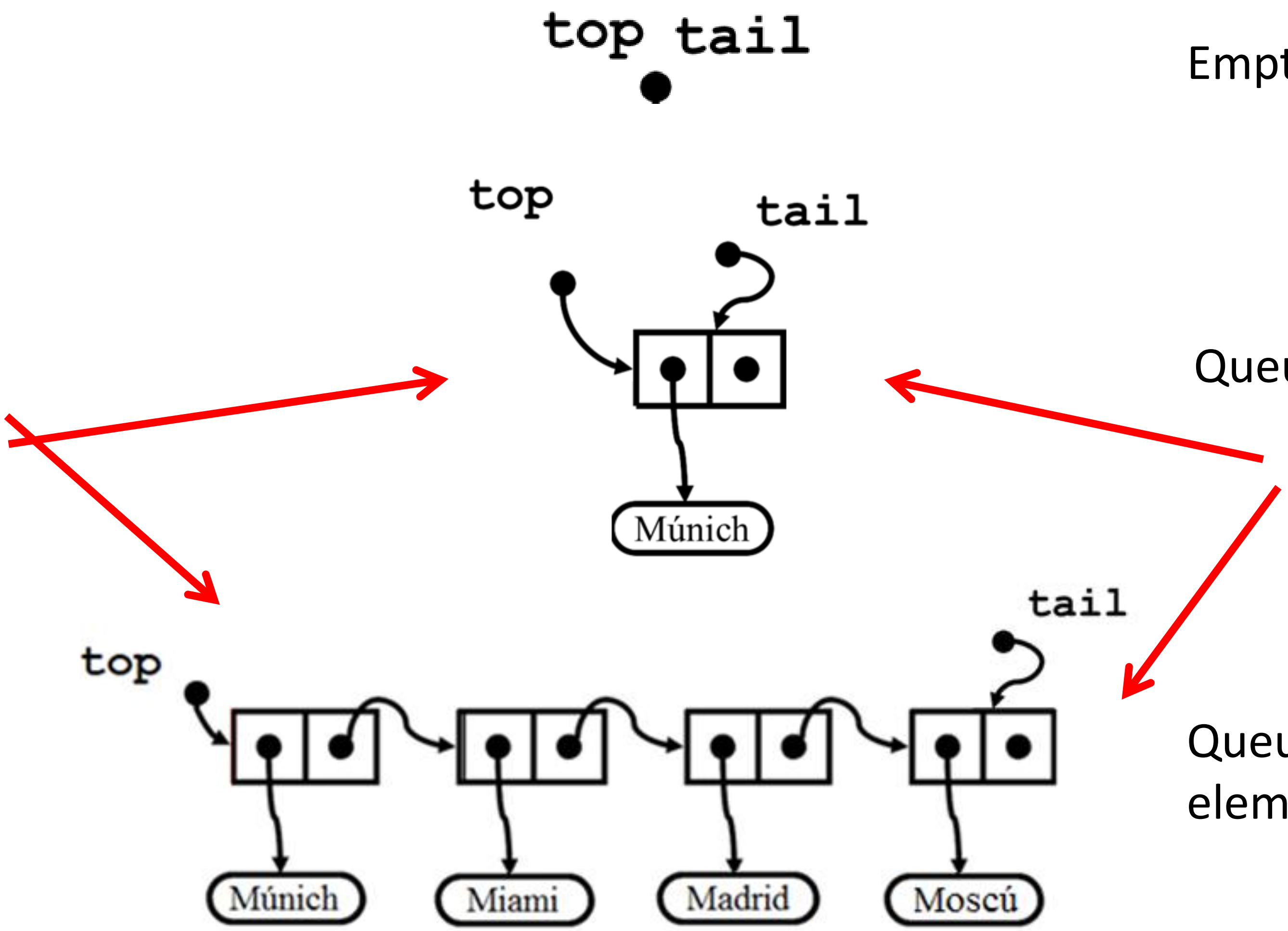
0	1	2	3	4	5	6	…	…	N-1
1	2	5	9	3	4				

Insertion of 5 extra elements

0	1	2	3	4	5	6	…	…	N-1
		5	9	3	4				

Extraction of 2 elements

LinkedQueue



Empty queue

Queue with 1element

Extraction end

Insertion end

Queue with 4 elements

LinkedQueue (I)

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

    public boolean isEmpty() {
        return (size == 0);
    }

    public int size() {
        return size;
    }

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

Attributes

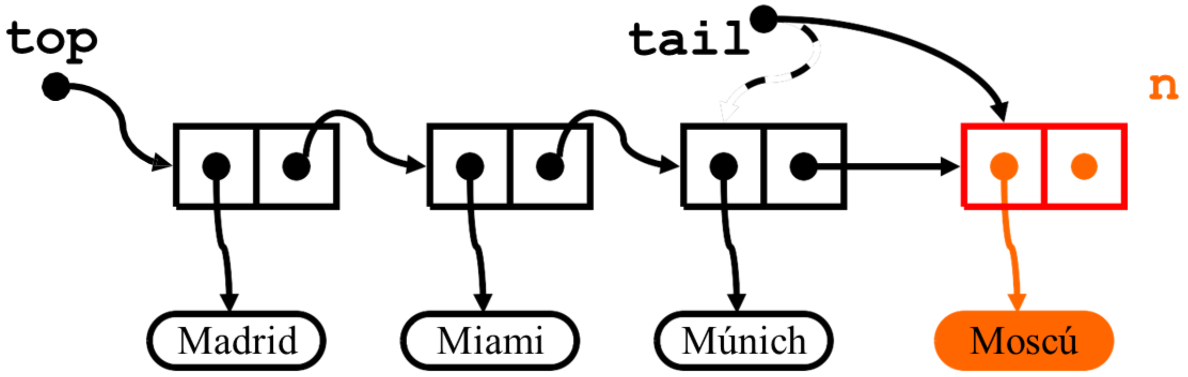
Constructor

Queue interface methods to implement(I)

LinkedQueue (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++;
    }
…
```

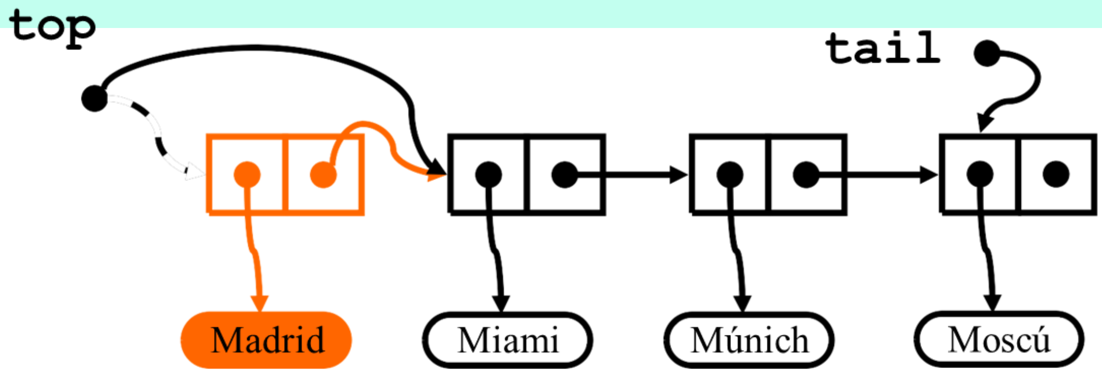
Queue interface methods
to implement (II)



LinkedQueue (III)

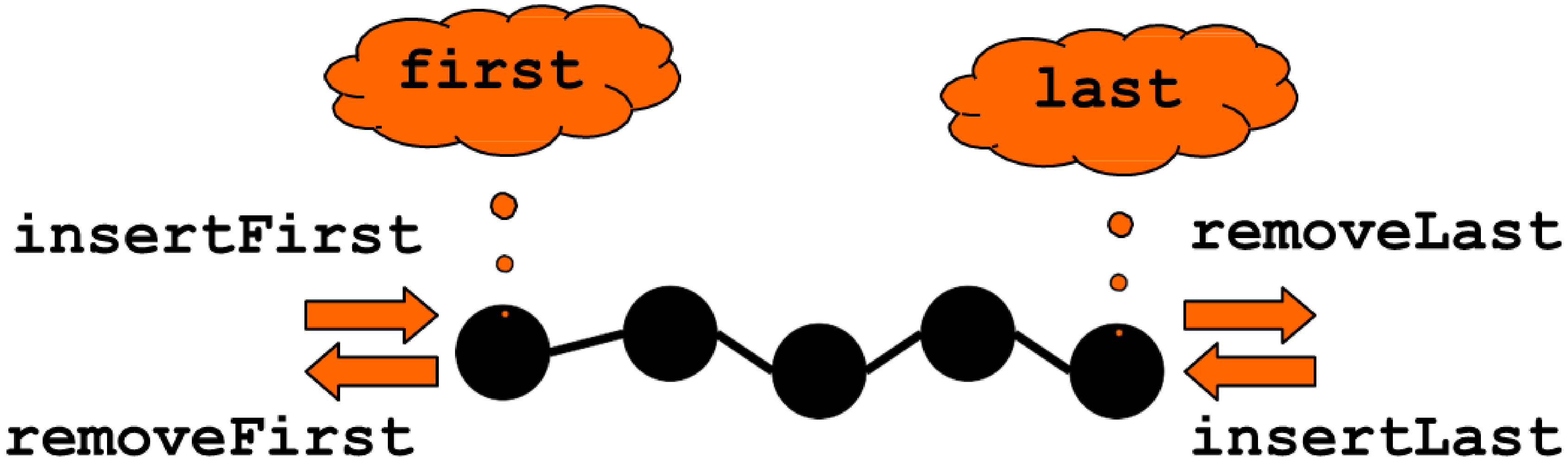
```
…
    public E dequeue(){
        E info;
        if (isEmpty()){
            return null;
        }
        info = top.getInfo();
        top = top.getNext();
        if (isEmpty()){
            tail = null;
        }
        size--;
        return info;
    }
}
```

Queue interface methods
to implement(III)



Double-ended queues (*deques*)

- Linear data structures
  - *Deque (double-ended queue)*
- Insertion and extraction from any end



Interface for deques

```java
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();
}
```

Interface for deques

Stack	Deque
`size()`	`size()`
`isEmpty()`	`isEmpty()`
`top()`	`last()`
`push(x)`	`insertLast(x)`
`pop()`	`removeLast()`

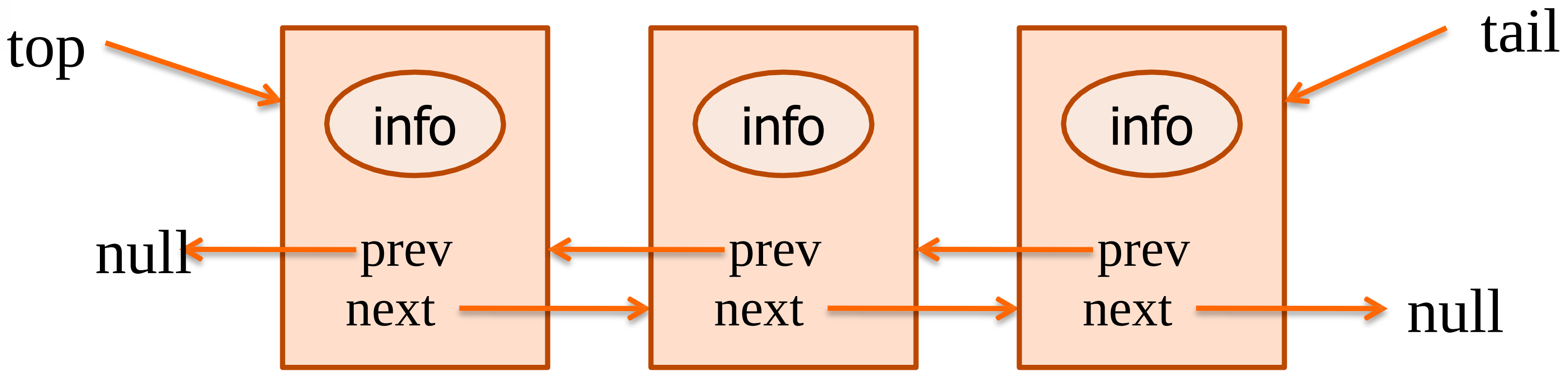
Queue	Deque
`size()`	`size()`
`isEmpty()`	`isEmpty()`
`front()`	`first()`
`enqueue(x)`	`insertLast(x)`
`dequeue()`	`removeFirst()`

Implementation of deques

- (regular) linked lists are not the best idea because `removeLast` needs to traverse the list from the beginning to find the reference to the next-to-last element
- Solution: **doubly-linked lists**

Doubly-linked lists

- Linked lists where each node, in addition to the **information** and the reference to the **next node** in the list, also stores a reference to the **previous node**
  - The list can be traversed in both directions
  - The cost to extract the last node is reduced



DLNode class

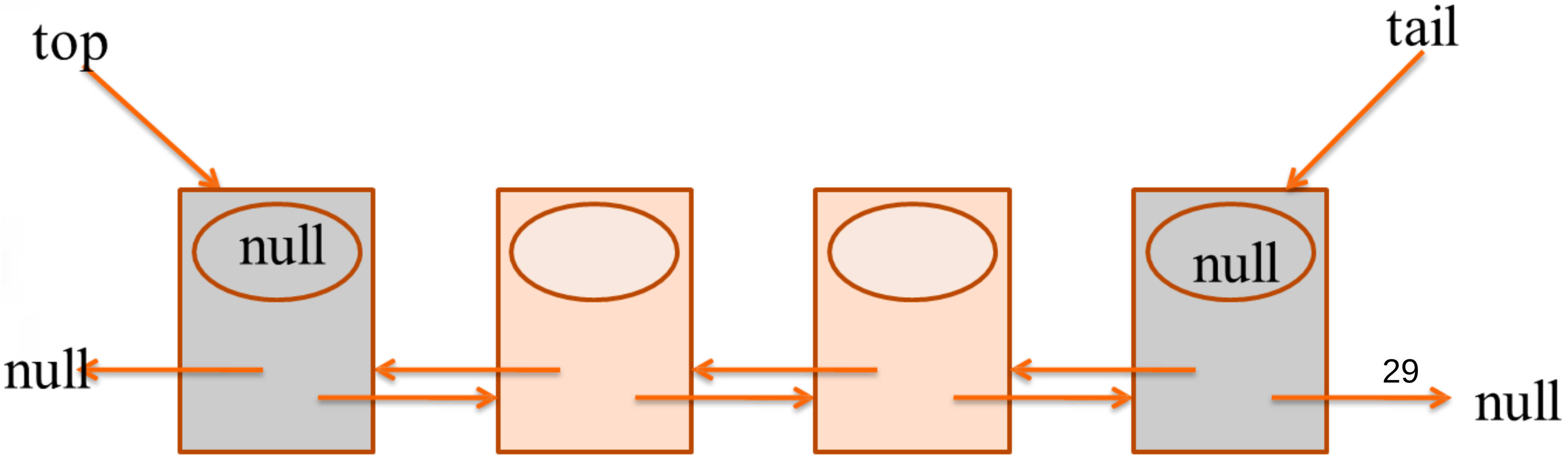
```
public class DLNode<E> {
    private E info;
    private DLNode<E> prev;
    private DLNode<E> next;

    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){…}
}
```

Exercise 2

- Complete the code for the **DLNode** class. Add three constructors: one with no parameters, a second one that allows to initialise the **info** attribute, and another constructor to initialise all attributes.

Doubly-linked lists

- The implementation of deques based on linked lists needs to check in each operation that both the previous and the next node exist
- **Simplification:** Create two special nodes (dummy nodes), with no data, one at the beginning and another at the end of the list:
  - An empty list only contains these two nodes.
  - In each insertion or extraction operation, both the previous and the next node always exist, without needing to check.
  - `top` and `tail` references never change.



Double queue class (DLDeque) with doubly-linked lists

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

Attributes

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

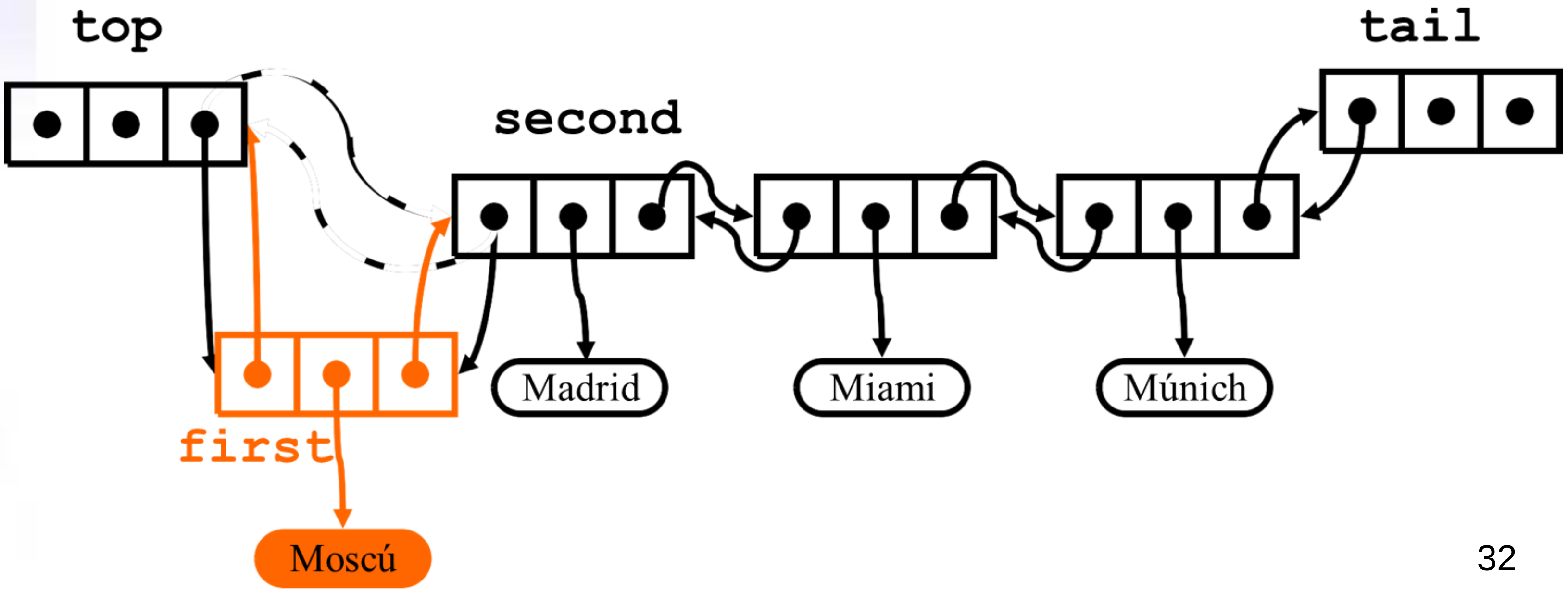
Constructor

Exercise 3

- Implement the following methods in the **DLDeque** class:
  - `boolean isEmpty()`
  - `int size()`
  - `E first()`
  - `E last()`

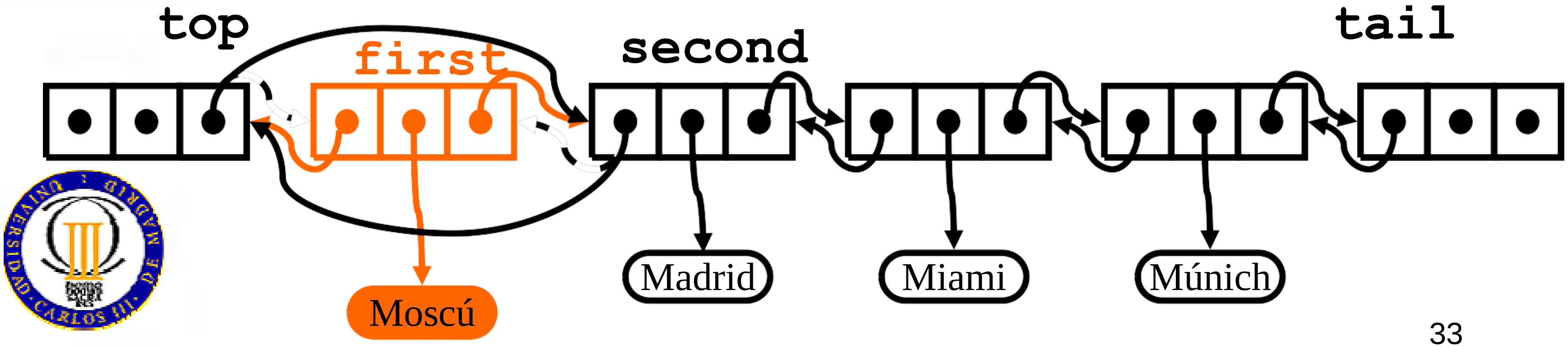
Double queue class (DLDeque) with doubly-linked lists: Insertion at the beginning

```
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++;
}
```



Double queue class (DLDeque) with doubly-linked lists: Extraction from the beginning

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



Exercise 4

- Implement the following methods in the **DLDeque** class:
  - ○ **`void insertLast (E info)`**
- *Homework:*
  - ○ ***`E removeLast()`***