Sesión 23: Pilas y Colas

Programación 2

Ángel Herranz

2022-2023

Universidad Politécnica de Madrid

En capítulos anteriores

- 👍 Tema 1: Intro a POO
- 👍 Tema 2: Clases y Objetos y TADs y módulos
- 👍 Tema 3: Colecciones con *arrays*
- 👍 Tema 4: Cadenas simplemente enlazadas
- 👍 Tema 5: TAD Listas
- 👍 Tema 7: Polimorfismo
- 👍 Tema 8: Excepciones

En el capítulo de hoy

- 👍 Tema 6: Pilas y Colas
  - 👍 *Stack*
    - *Puntero* al primero
  - 👍 *Queue*
    - *Puntero* al primero
    - *Puntero* al primero y al último
    - Estructuras "circulares": cadena y *buffer* (*array*)

IList y LinkedList

Un TAD (**tipo abstracto de datos**) lo definen

- Un nombre, el nombre del tipo.
- Unas operaciones.
- La semántica de las operaciones.

```
interface IList
```

IList y LinkedList

Un TAD (**tipo abstracto de datos**) lo definen

- Un nombre, el nombre del tipo.
- Unas operaciones.
- La semántica de las operaciones.

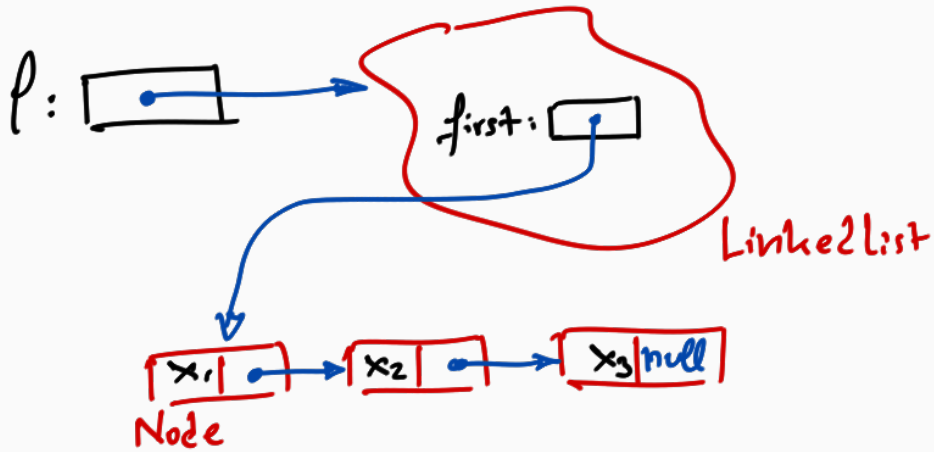
  interface IList

- El mismo TAD puede tener múltiples implementaciones concretas: **estructuras de datos**.

  class LinkedList y **class** ArrayList

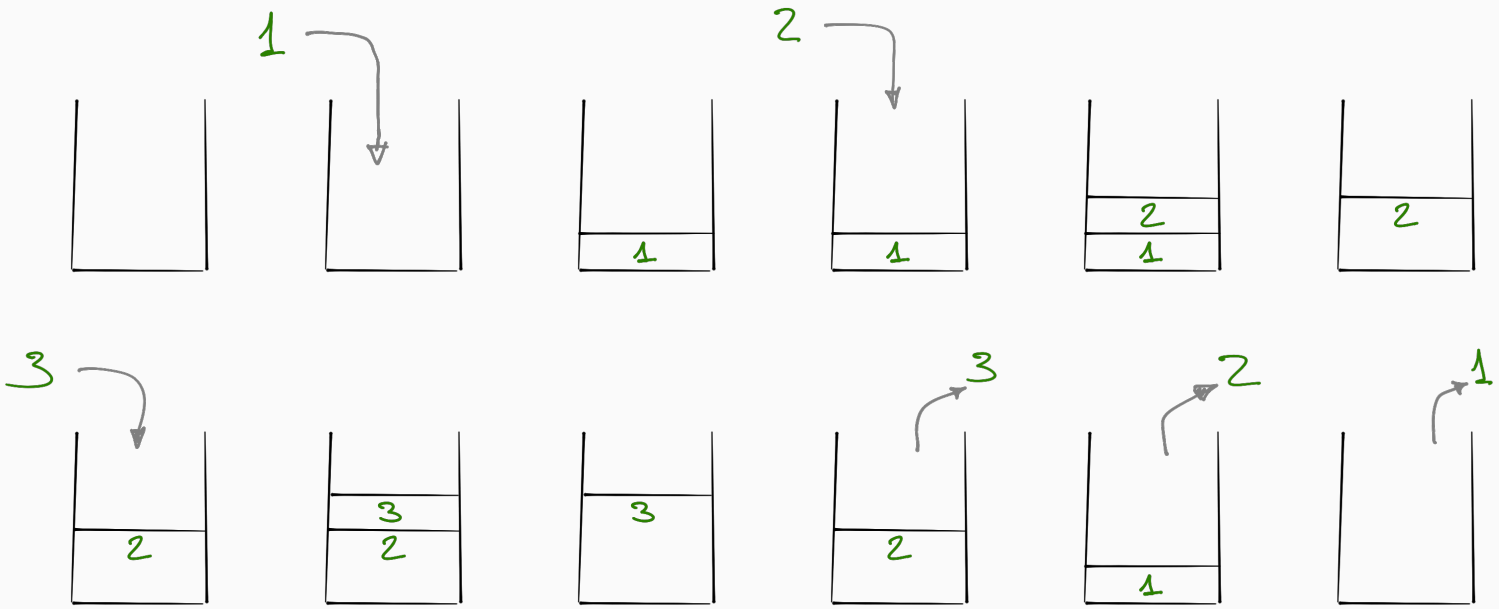
Cadena “simplemente” enlazada

```
class Node<T> {
  T data;
  Node<T> next;
}
```




```
public class LinkedList<T> implements IList<T> {
  private Node<T> first;
  ...
}
```

Stack



Imposible acceder a elementos internos: acceso *lineal*[1]

(vs. las listas que son *random access*)

[1] O *secuencial*.

TAD `IStack`

```
public interface IStack<T> {
  boolean isEmpty();
  T peek();
  void pop();
  void push(T e);
}
```

Estructura de datos “pilas acotadas”

- Implementar `ArrayStack<T>` con las técnicas usadas en la sesión 10: pilas acotadas

```
public class ArrayStack<T> implements IStack<T> {
  private Object data[] = new Object[MAX_DATA];
  private int top = -1;
  ...
}
```

  - Puedes aplicar técnicas de redimensionamiento

Estructura de datos cadena enlazada

- Implementar `LinkedStack<T>` usando una cadena simplemente enlazada

```
public class LinkedStack<T> implements IStack<T> {
  private Node<T> top;
  ...
}
```

- Trivial: todas las operaciones se ejecutan en tiempo constante[2]

[2] $O(1)$

LIFO: *Last in, first out*

- Las pilas tienen un comportamiento lineal

  No es posible acceder a
  un elemento intermedio

- Con organización LIFO

  El primer dato que se extrae
  es el último introducido

- Ubícuo: pila de ejecución, inversión, evaluación de expresiones, compiladores, etc.

FIFO: *First in, first out*

- También existe un comportamiento lineal

  No es posible acceder a
  un elemento intermedio

- Con organización FIFO

  El primer dato que se extrae
  es el primero que se introdujo

- Ubícuo: espera de procesos, atención de peticiones en servidores, etc.

TAD `IQueue`

```
public interface IQueue<T> {
  boolean isEmpty();
  void add(T valor);
  T peek();
  void poll();
}
```

Tests: LIFO vs FIFO

```
IStack<Integer> s = ...;
for (int i = 0; i < N; i++) {
  s.add(i);
  assert s.peek().equals(i);
}
for (int i = 0; i < N; i++) {
  assert s.peek().equals(N-i-1);
  s.pop();
}
```

Tests: LIFO vs FIFO

```
IStack<Integer> s = ...;
for (int i = 0; i < N; i++) {
  s.add(i);
  assert s.peek().equals(i);
}
for (int i = 0; i < N; i++) {
  assert s.peek().equals(N-i-1);
  s.pop();
}
```

```
IQueue<Integer> q = ...;
for (int i = 0; i < N; i++) {
  q.add(i);
  assert q.peek().equals(0);
}
for (int i = 0; i < N; i++) {
  assert q.peek().equals(i);
  q.pop();
}
```

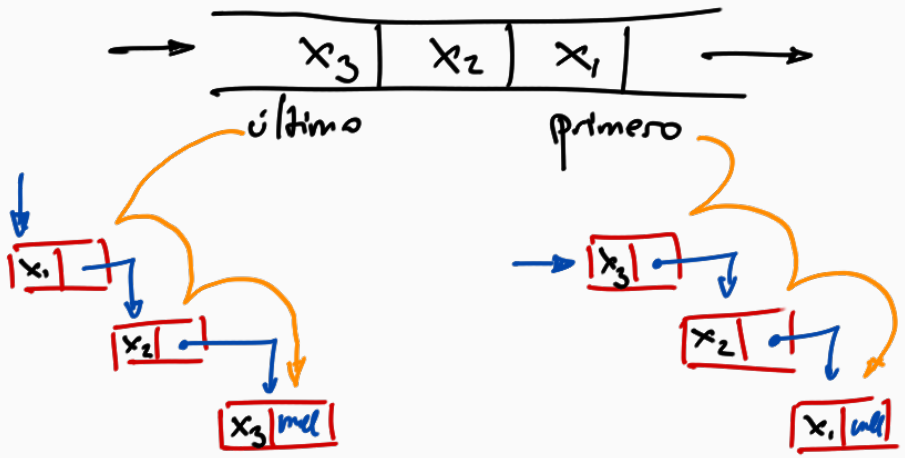
Estructura de datos cadena enlazada

- Implementar `LinkedQueue<T>` usando una cadena simplemente enlazada

```
public class LinkedQueue<T> implements IQueue<T> {
  private Node<T> first;
  ...
}
```

¿$O(1)$?

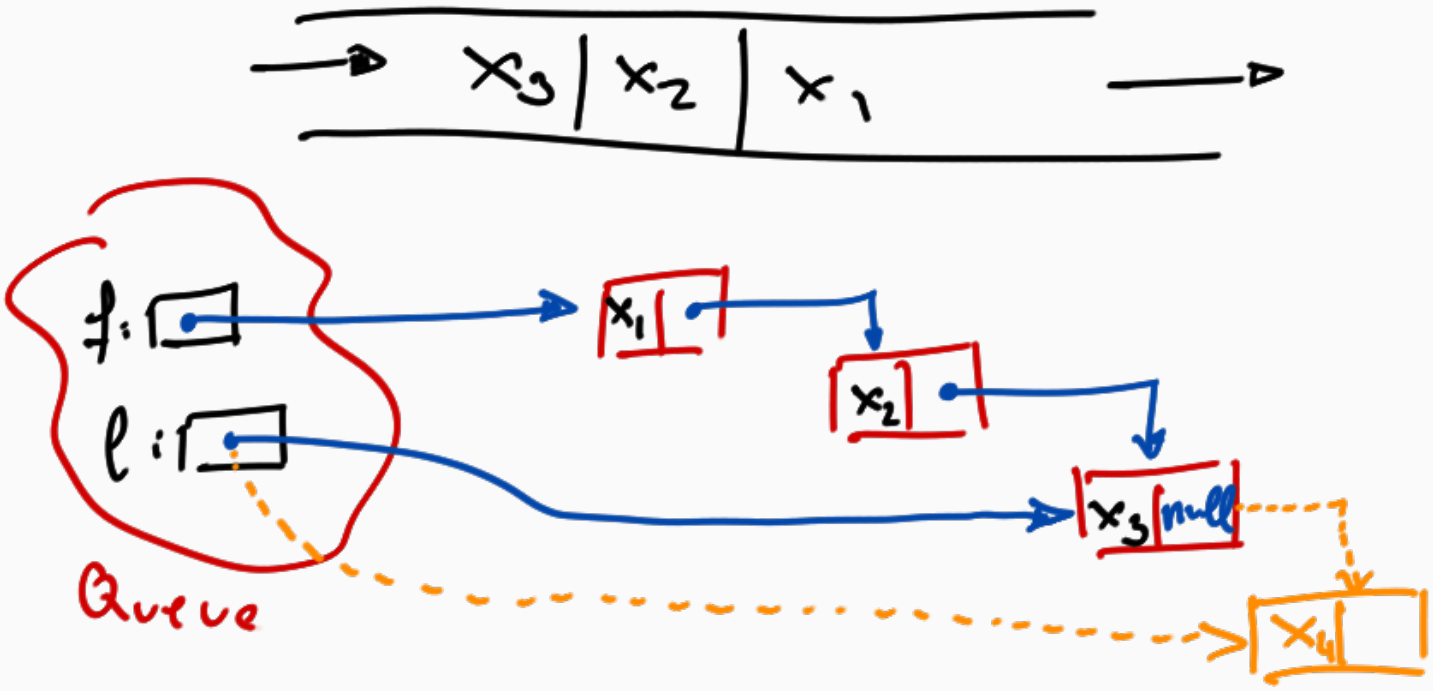
- Con una cadena simplemente enlazada no es posible implementar todos los métodos de colas en tiempo constante



¿Qué se te ocurre?

$O(1)$

- Si el problema es acceder al último (o al primero) una opción es tener dos *punteros*

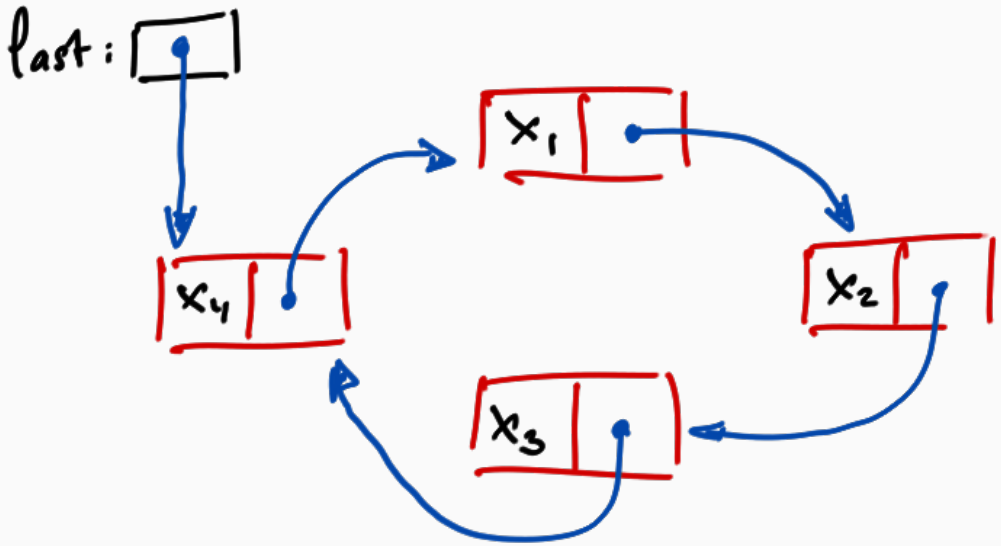


Estructura de datos cadena *first-last*

- Modificar `LinkedQueue<T>` usando una cadena enlazada con punteros al primero y al último

```
public class LinkedQueue<T> implements IQueue<T> {
  private Node<T> first;
  private Node<T> last;
  ...
}
```

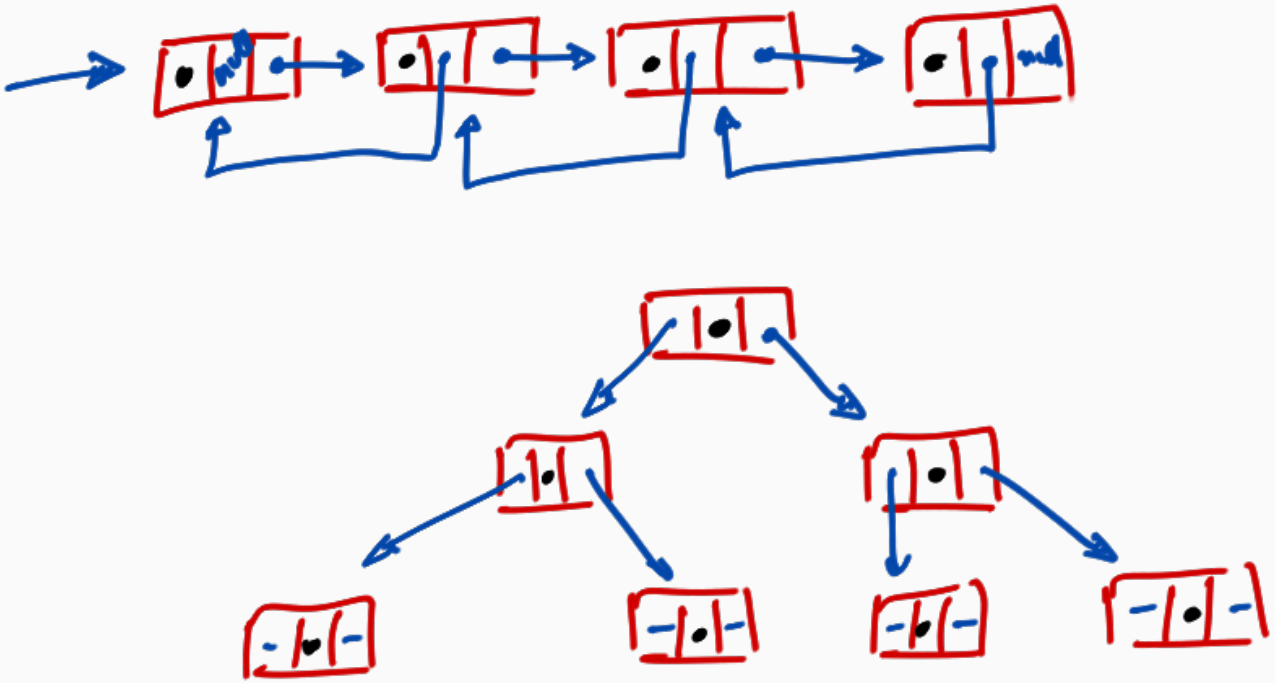
Ejercicio mental: cadena “circular”



¡El primero es `last.next`!

Las cadenas son muy importantes

- Permiten crear estructuras de datos potentes
- Recorridos en dos sentidos, árboles, grafos, etc.



Biblioteca TADs lineales

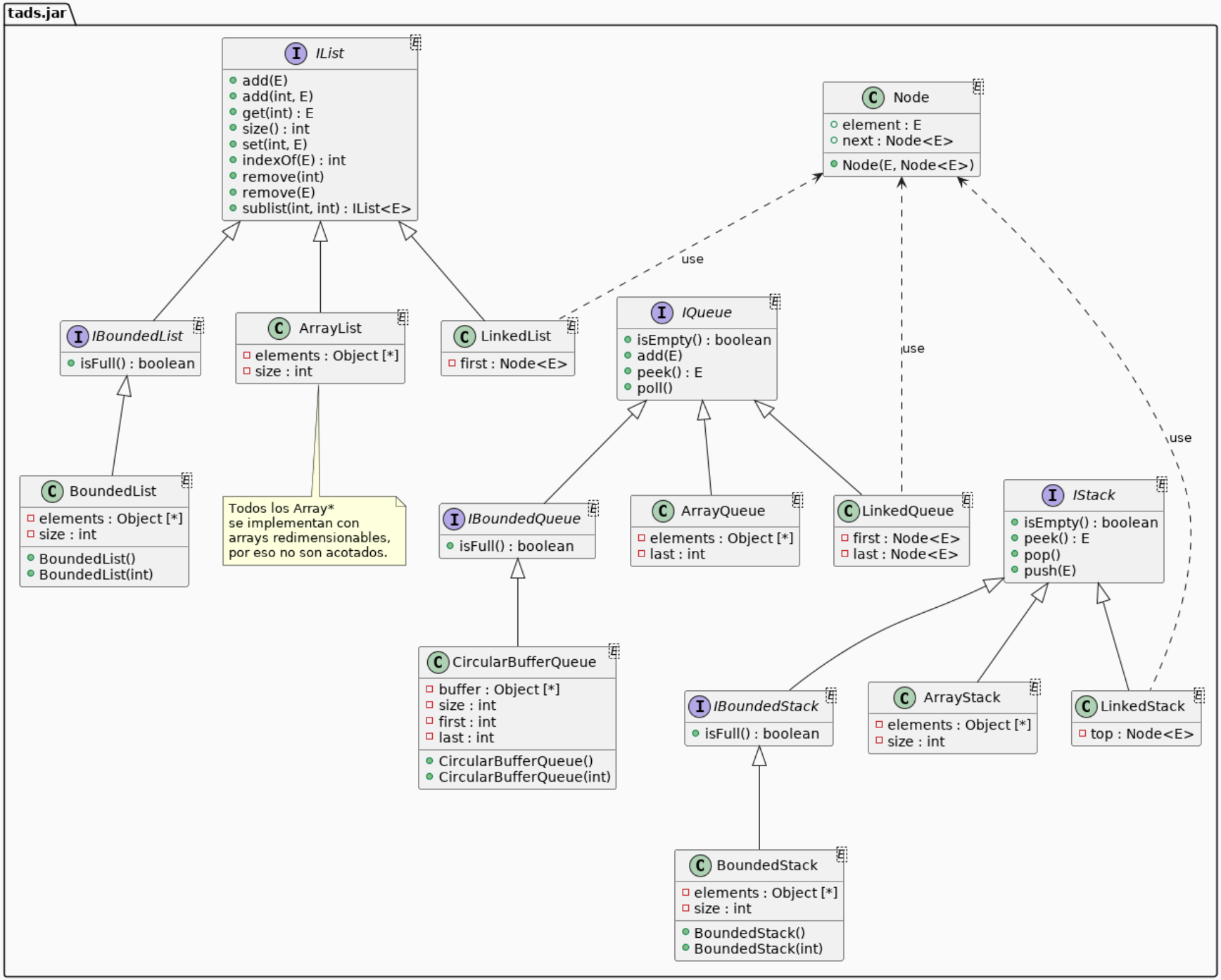
- ¿Dónde está la biblioteca?
- ¿Qué hay dentro del `jar`?
- ¿Y la documentación?
- ¿Cómo puedo verla?

Biblioteca TADs lineales

- ¿Dónde está la biblioteca? En moodle: `tads.jar`
- ¿Qué hay dentro del `jar`? `*.class` y `*.java`
- ¿Y la documentación? Moodle: `tads-doc.zip`
- ¿Cómo puedo verla? Descomprimir y abrir `index.html` con un navegador Web

 Profundiza en la (java)documentación

 Profundiza en las implementaciones

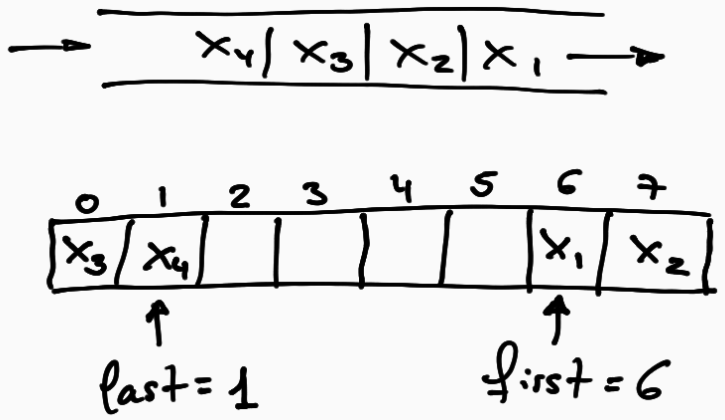

tads.jar
IList
add(E)
add(int, E)
get(int) : E
size() : int
set(int, E)
indexOf(E) : int
remove(int)
remove(E)
sublist(int, int) : IList<E>
Node
element : E
next : Node<E>
Node(E, Node<E>)
use
use
use
IBoundedList
isFull() : boolean
ArrayList
elements : Object [*]
size : int
LinkedList
first : Node<E>
IQueue
isEmpty() : boolean
add(E)
peek() : E
poll()
BoundedList
elements : Object [*]
size : int
BoundedList()
BoundedList(int)
Todos los Array*
se implementan con
arrays redimensionables,
por eso no son acotados.
IBoundedQueue
isFull() : boolean
ArrayQueue
elements : Object [*]
last : int
LinkedQueue
first : Node<E>
last : Node<E>
IStack
isEmpty() : boolean
peek() : E
pop()
push(E)
CircularBufferQueue
buffer : Object [*]
size : int
first : int
last : int
CircularBufferQueue()
CircularBufferQueue(int)
IBoundedStack
isFull() : boolean
ArrayStack
elements : Object [*]
size : int
LinkedStack
top : Node<E>
BoundedStack
elements : Object [*]
size : int
BoundedStack()
BoundedStack(int)

Programar `CircularBufferQueue<T>`

```
public class CircularBufferQueue<T>
  implements IBoundedQueue<T> {

  private Object[] buffer;
  private int first;
  private int last;

  ...
}
```

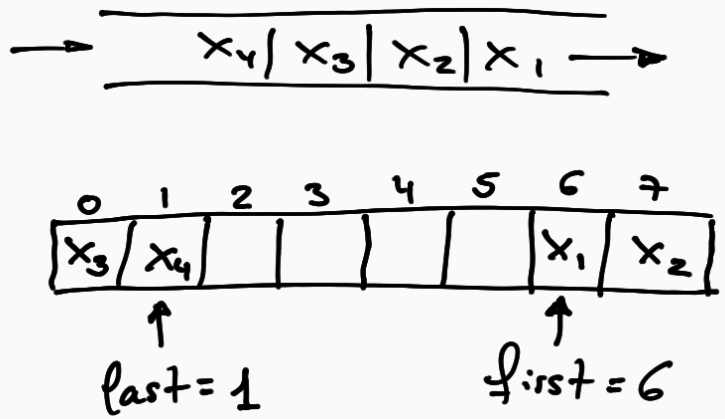


Programar `CircularBufferQueue<T>`

```
public class CircularBufferQueue<T>
  implements IBoundedQueue<T> {

  private Object[] buffer;
  private int first;
  private int last;

  ...
}
```



Peek: `return buffer[first];`

Poll: `first = (first + 1) % 8;`

Add: `last = (last + 1) % 8;`

`buffer[last] = elem;`