Estructuras de Datos. TAD, Colas.

Profesores Estructuras de Datos, 2024.

Dpto. Lenguajes y Ciencias de la Computación.

University of Málaga

Beneficios para el usuario de TAD

- **Encapsulación**: Los TAD encapsulan los datos y proporcionan una *interfaz* a las operaciones.
- **Abstracción**: Nos permiten centrarnos en las operaciones sin considerar los detalles de implementación.
- **Reutilización**: los TAD se pueden implementar de múltiples maneras, lo que proporciona flexibilidad para elegir la mejor *implementación* para un problema determinado. Además, pueden reutilizarse en distintos proyectos o en varias partes del mismo programa. Esto reduce la duplicación de código y mejora la eficiencia del desarrollo.

Beneficios para el desarrollador de TAD

- **Mantenibilidad**: Permiten cambiar la implementación interna para mejorar el rendimiento o adaptarse a nuevas necesidades. Esto reduce el riesgo de errores cuando se hacen modificaciones.
- **Modularidad**: El código se estructura en módulos más pequeños y manejables. Esto mejora la reutilización y separación de preocupaciones, ya que cada TAD se ocupa de un aspecto específico del problema.
- **Estandarización**: Si se generan de forma consistente, facilita la colaboración entre desarrolladores, ya que se pueden utilizar convenciones y patrones reconocidos.
- **Análisis**: Facilitan la evaluación del rendimiento y la complejidad de los algoritmos asociados.

El TAD cola

- **Una Cola** es una colección que almacena elementos **en un orden** de primero en entrar, primero en salir (*FIFO*).
- **Operaciones**:
  - `enqueue` : Agrega un elemento al *final* de la cola.
  - `dequeue` : Elimina el *primer* elemento de la cola.
  - `first` : Devuelve el *primer* elemento de la cola sin eliminarlo.
  - `isEmpty` : Comprueba si la cola está vacía.
  - `size` : Devuelve el número de elementos en la cola.
  - `clear` : Elimina todos los elementos de la cola.

Queue of people

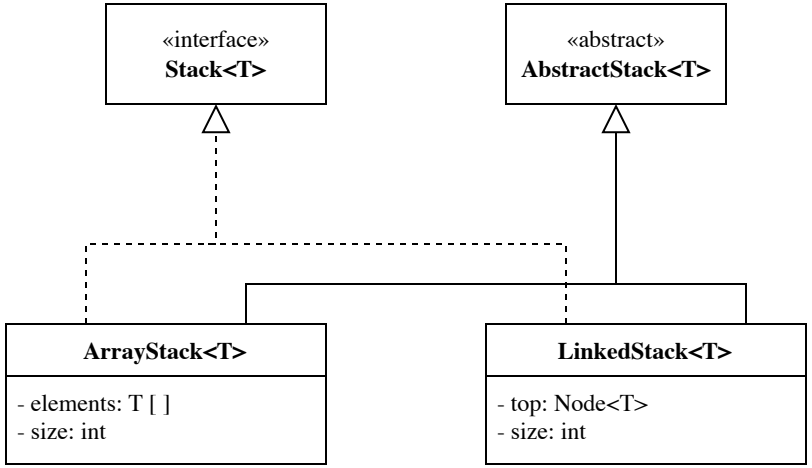
El TAD cola en Java: la interfaz

`Queue<T>` define una cola con elementos de tipo `T`.

```
package org.uma.ed.datastructure.queue;;

public interface Queue<T> {
  void enqueue(T element);
  T first();
  void dequeue();
  boolean isEmpty();
  int size();
  void clear();
}
```

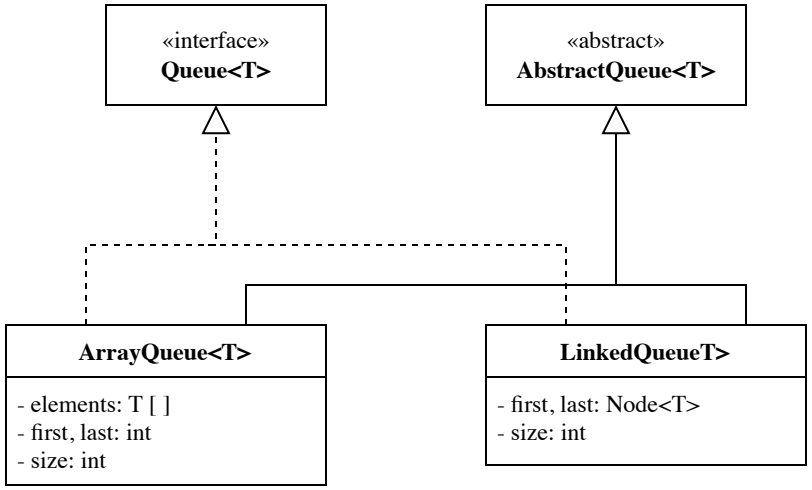
Recordando: implementaciones del TAD de pila

- Una pila se puede implementar utilizando diferentes *estructuras de datos*.
- Diferentes *clases* pueden implementar la interfaz `Stack<T>`:
  - `ArrayStack<T>`: utiliza un array para almacenar elementos.
  - `LinkedStack<T>`: utiliza una estructura vinculada para almacenar elementos.



Implementaciones del TAD cola:

- Una cola se puede implementar utilizando diferentes **estructuras de datos**.
- Diferentes **clases** pueden implementar la interfaz `Queue<T>`:
  - `ArrayQueue<T>`: utiliza una array para almacenar elementos.
  - `LinkedQueue<T>`: utiliza una estructura vinculada para almacenar elementos.
- La clase abstracta base `AbstractQueue<T>` proporciona implementación para los métodos `equals`, `hashCode` y `toString`.

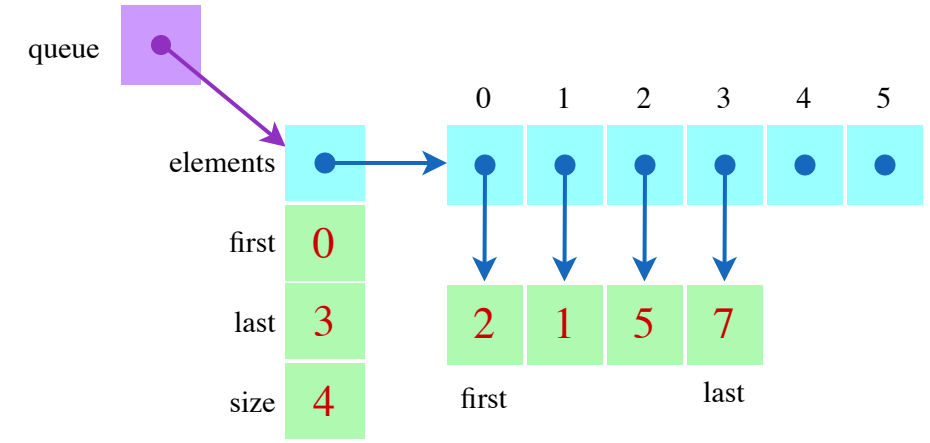


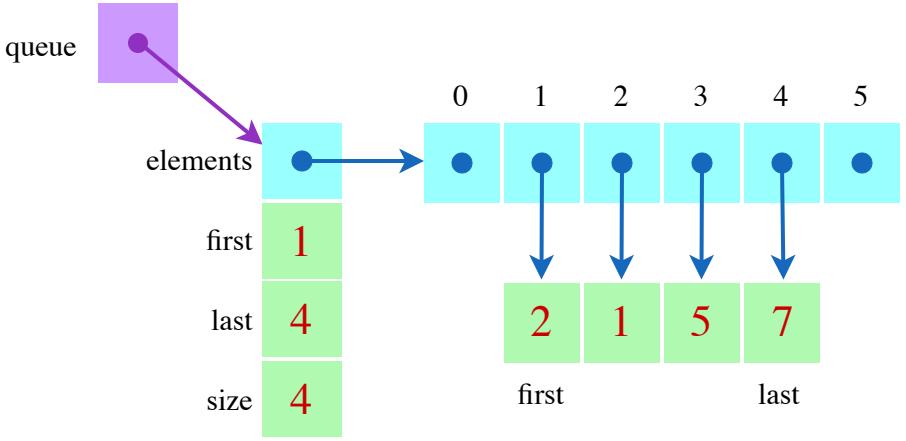
La clase `ArrayQueue`

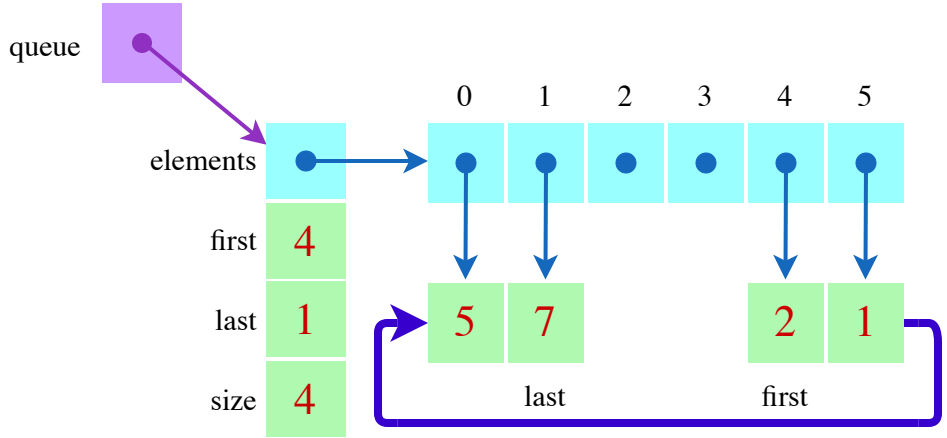
- `ArrayQueue<T>` implementa la interfaz `Queue<T>` utilizando una array para almacenar elementos.
- Inicialmente, la array tiene un tamaño fijo (**capacidad** de la cola), pero puede crecer dinámicamente cuando sea necesario.
- La clase mantiene dos índices enteros: `first` y `last`. Estos índices indican la posición del primer y último elemento de la cola en la array.
- **A medida que se agregan nuevos elementos a la cola, se almacenan en la array después del elemento `last` actual**.
- La palabra **after** indica la posición posterior en la array, volviendo al índice 0 si `last` está al final de la array.
- A medida que se extraen elementos de la cola, el índice `first` se incrementa para apuntar al siguiente elemento de la cola, volviendo al índice 0 si `first` está al final de la array.
- La clase también mantiene una variable entera `size` para realizar un seguimiento de la cantidad de elementos en la cola.

La clase `ArrayQueue` (II)

- Aquí mostramos tres posibles representaciones de la misma cola.
  Suponemos que la array tiene una capacidad de 6 elementos.
- Podemos ver que el primer elemento no tiene que estar en el índice 0 y que el índice del primer elemento puede ser mayor que el índice del último elemento.





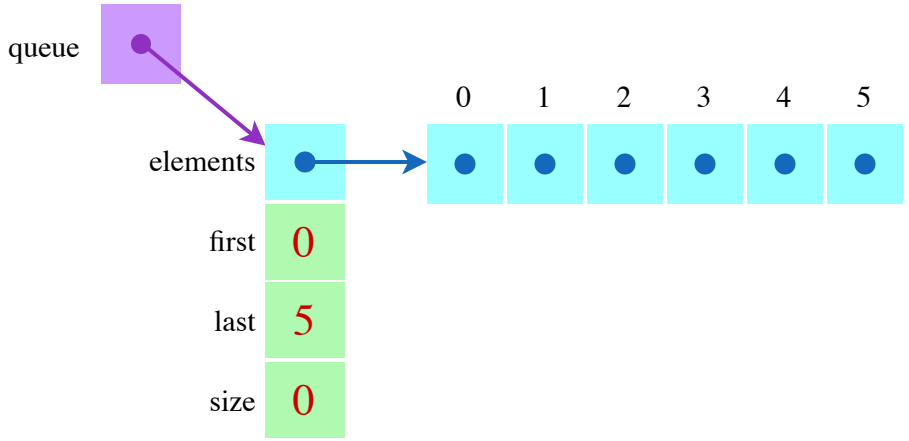


Implementación de `ArrayQueue`

```
package org.uma.ed.datastructure.queue;

public class ArrayQueue<T> extends AbstractQueue<T> implements Queue<T> {
  private static final int DEFAULT_INITIAL_CAPACITY = 16;

  private T[] elements;
  private int first, last;
  private int size;

  public ArrayQueue(int initialCapacity) {  // ArrayQueue constructor
    if (initialCapacity <= 0) {
      throw new IllegalArgumentException("initial capacity must be greater than 0");
    }
    elements = (T[]) new Object[initialCapacity];
    size = 0;
    first = 0;
    last = initialCapacity - 1; // so that first increment makes it 0
  }

  public ArrayQueue() {  // ArrayQueue constructor
    this(DEFAULT_INITIAL_CAPACITY);
  }
  ...
}
```

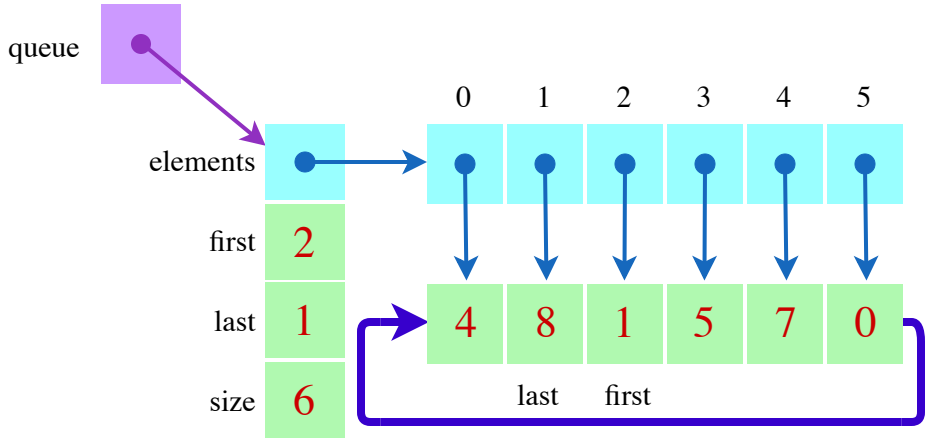
Inicialización de `ArrayQueue`

- Cuando la cola está vacía, `first` es 0 y `last` es el último índice de la array.
- La primera operación `enqueue` que se realiza comenzará incrementando `last` (haciéndolo 0) de modo que el primer elemento se almacenará en el índice 0.

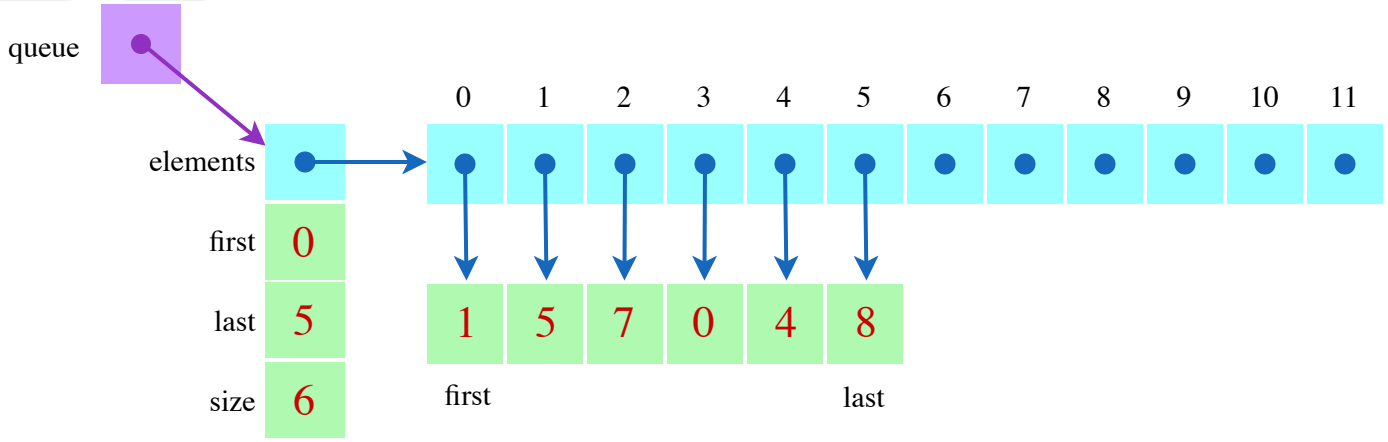


Garantizar la capacidad en `ArrayQueue`

- Al alcanzar su capacidad máxima, el conjunto necesita ser ampliado para albergar nuevos elementos; por lo tanto, se construye un nuevo conjunto con *el doble* de capacidad para asegurar espacio para elementos adicionales.



- Para transferir elementos sin problemas a la nueva array, inicie la secuencia de copia en el "primer" índice de la array anterior, continúe hasta el "último" índice y colóquelos secuencialmente al comienzo de la array recién creada.
- `first` y `last` se actualizan para reflejar los índices de la nueva array.

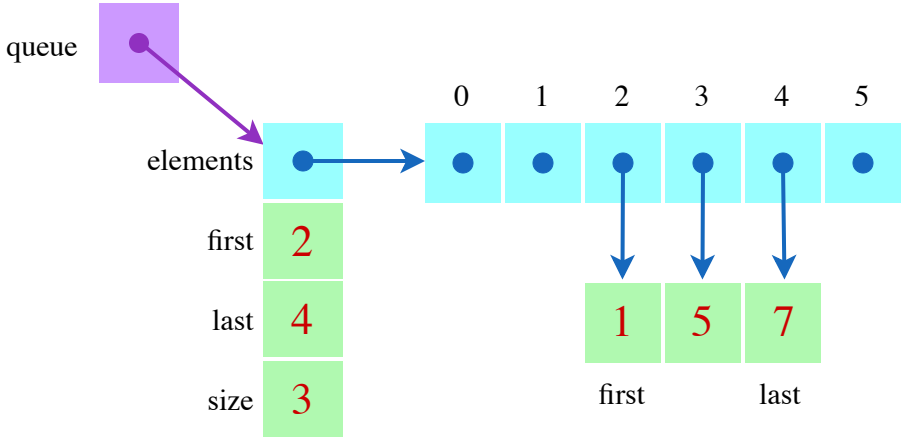


Poner un elemento al final de `ArrayQueue`

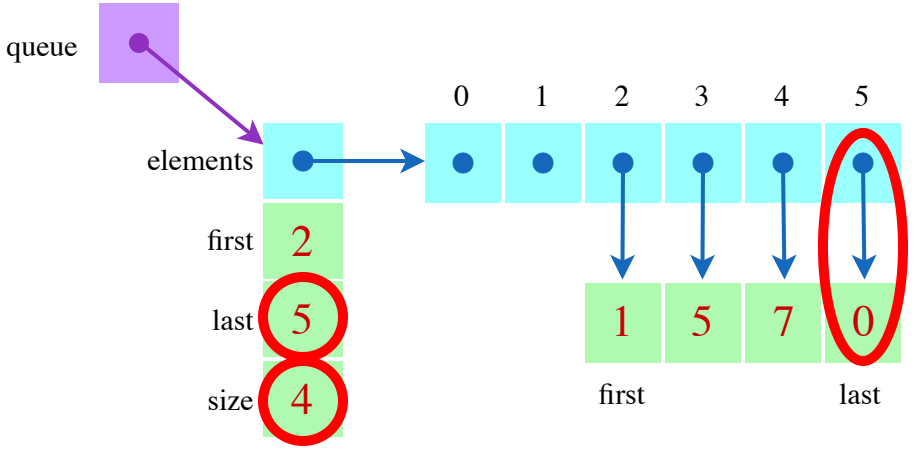
- `enqueue` agrega un elemento *después* del último elemento en la cola:
  - Se debe asegurar la capacidad del array para alojar el nuevo elemento.
  - `last` se incrementa para apuntar a la siguiente posición disponible (retrocediendo al índice 0 si `last` está al final de la array)
  - La array almacena el nuevo elemento en el índice designado por 'last'.
  - `size` se incrementa para realizar un seguimiento de la cantidad de elementos en la cola.

Poner un elemento al final de `ArrayQueue`

- A partir de esta configuración, vamos a poner en cola el elemento 0 al final de la cola:

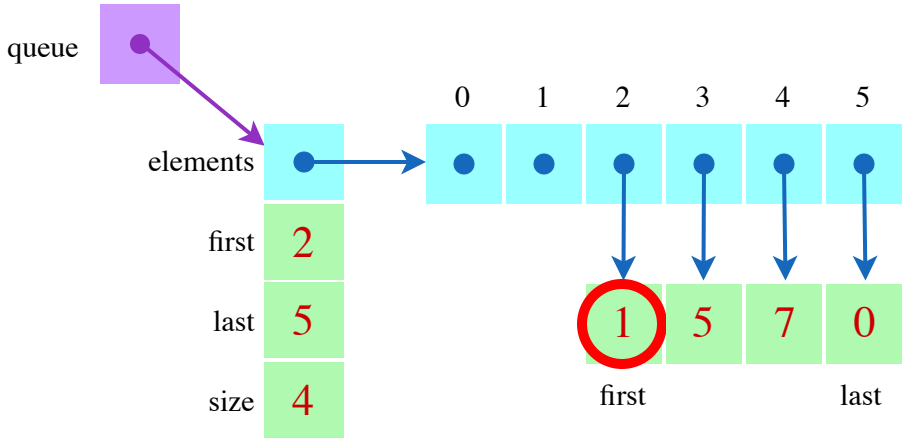


- Después de poner en cola 0:



Accediendo al primer elemento en `ArrayQueue`

- Si la cola está vacía, se debe lanzar una `EmptyQueueException`.
- Si la cola no está vacía:
  - El primer elemento es el almacenado en la posición `first`.
  - Este elemento debe ser devuelto.

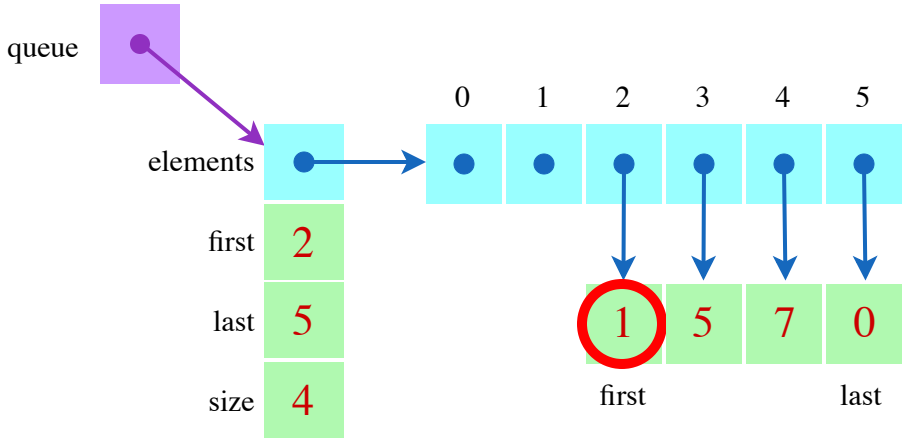


Quitar de la cola el primer elemento en `ArrayQueue`

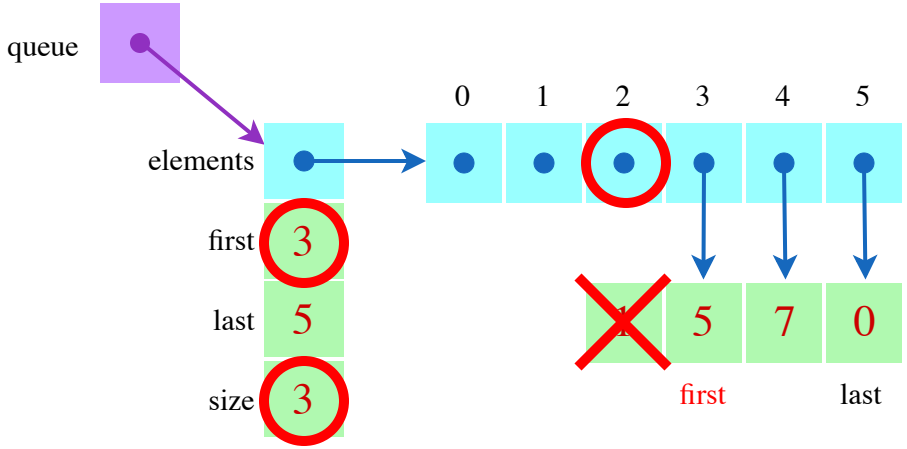
- Si la cola está vacía, se debe lanzar una `EmptyQueueException`.
- Si la cola no está vacía, el primer elemento se elimina mediante:
  - Establecer el elemento en la posición `first` como `null`. Esto elimina el primer elemento y permitirá que el *recolector de basura* recupere la memoria utilizada por el elemento eliminado.
  - `first` se incrementa para apuntar al siguiente elemento en la cola (regresando al índice 0 si `first` está al final de la array)
  - Disminuimos el tamaño.

Quitar de la cola el primer elemento en `ArrayQueue`

- Partiendo de esta configuración, vamos a 'desencolar' el primer elemento:



- Después de sacar de la cola:



Métodos de fábrica para `ArrayQueue`

- Se pueden utilizar métodos de fábrica para crear instancias de `ArrayQueue<T>`.
- `empty()`: construye una *cola vacía* con una capacidad inicial *predeterminada*, adecuada para cuando se desconoce el número esperado de elementos.
- `withCapacity(int initialCapacity)`: construye un **cola vacía** con una capacidad inicial especificada, optimizando la asignación de memoria para un número conocido de elementos.
- `of(T... elementos)`: construye una cola *previamente rellenada* con los elementos proporcionados, lo que permite una configuración de cola rápida y sencilla.
- `copyOf(Queue<T> queue)`: construye una nueva cola que es una **duplicado** de la `cola` dada, preservando el orden de los elementos.
- `from(Iterable<T> iterable)`: construye una nueva cola
  que contiene todos los elementos del *iterable* especificado, manteniendo su orden de iteración.

```
Queue<Integer> queue1 = ArrayQueue.empty();

Queue<Integer> queue2 = ArrayQueue.of(1, 2, 3);

Queue<Integer> queue3 = ArrayQueue.copyOf(queue2);
queue3.push(4);

Queue<Integer> queue4 = ArrayQueue.from(LinkedList.of(5, 6, 7));
int sum = 0;
while (!queue4.isEmpty()) {
  sum += queue4.first();
  queue4.dequeue();
}
```

Complejidad computacional de las operaciones de `ArrayQueue`

Operation	Cost
`empty`	O(1) †
`enqueue`	O(1), O(n) §
`dequeue`	O(1)
`first`	O(1)
`isEmpty`	O(1)
`size`	O(1)
`clear`	O(n) *

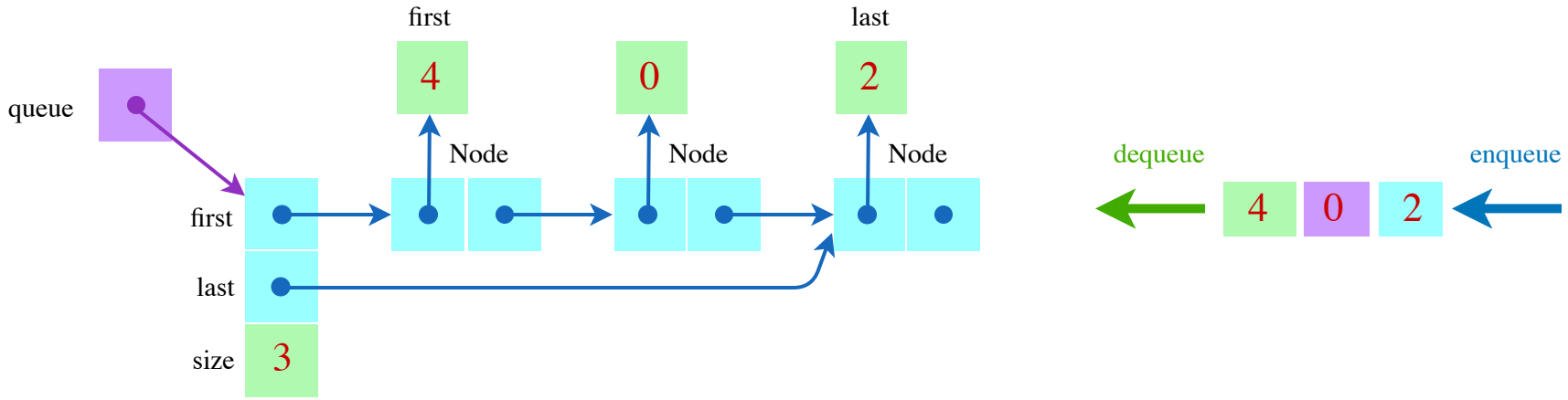
† In `empty` the size of the created array is a constant.

§ `enqueue` will need to copy n elements when the array has to be enlarged.

* `clear` will need to set n references to `null`.

La clase `LinkedQueue`

- `LinkedQueue<T>` implementa la interfaz `Queue<T>` utilizando una estructura vinculada de nodos.
- A medida que se colocan nuevos elementos al final de la cola, se insertan nuevos nodos al final de la estructura vinculada.
- La clase mantiene una referencia `first` al primer nodo en la estructura vinculada que corresponde al primer elemento en la cola.
- La clase también mantiene una referencia `last` al último nodo en la estructura vinculada que corresponde al último elemento en la cola.
- Cada nodo contiene un elemento y una referencia (`next`) al nodo que contiene el elemento después de él en la cola.
- El último nodo tiene su referencia `siguiente` establecida en `nulo` .
- La clase también mantiene una variable entera `size` para realizar un seguimiento de la cantidad de elementos en la cola.

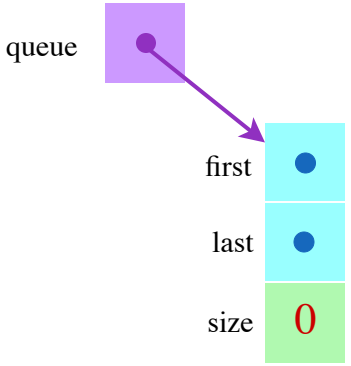


Implementación de `LinkedQueue`

```
package org.uma.ed.datastructure.queue;

public class LinkedQueue<T> extends AbstractQueue<T> implements Queue<T> {
  private static final class Node<E> {  // Node inner class
    E element;
    Node<E> next;

    Node(E element, Node<E> next) {  // Node constructor
      this.element = element;
      this.next = next;
    }
  }

  private Node<T> first, last;
  private int size;

  public LinkedQueue() {  // LinkedQueue constructor
    first = null;
    last = null;
    size = 0;
  }

  ...
}
```

Inicialización de `LinkedQueue`

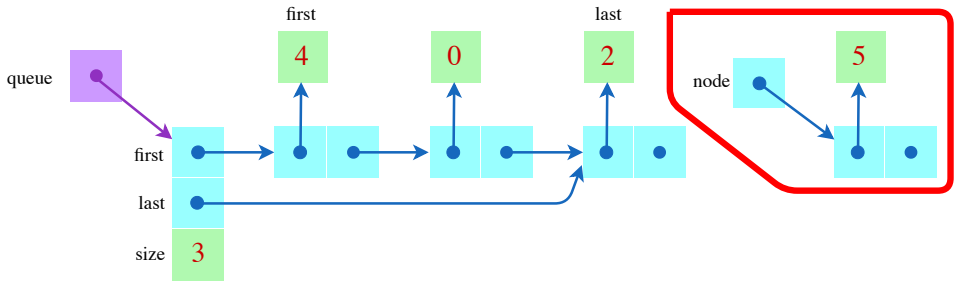
- Cuando la cola está vacía:
  - `first` y `last` son `null`.
  - `size` es 0



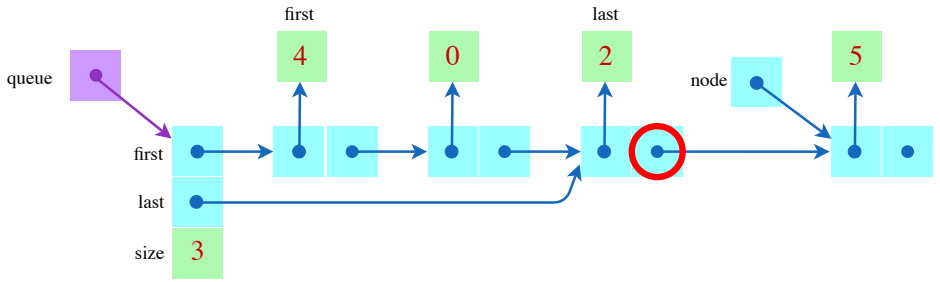
Poner en cola un elemento al final en `LinkedQueue`

`enqueue` agrega un elemento después del último elemento en la cola.

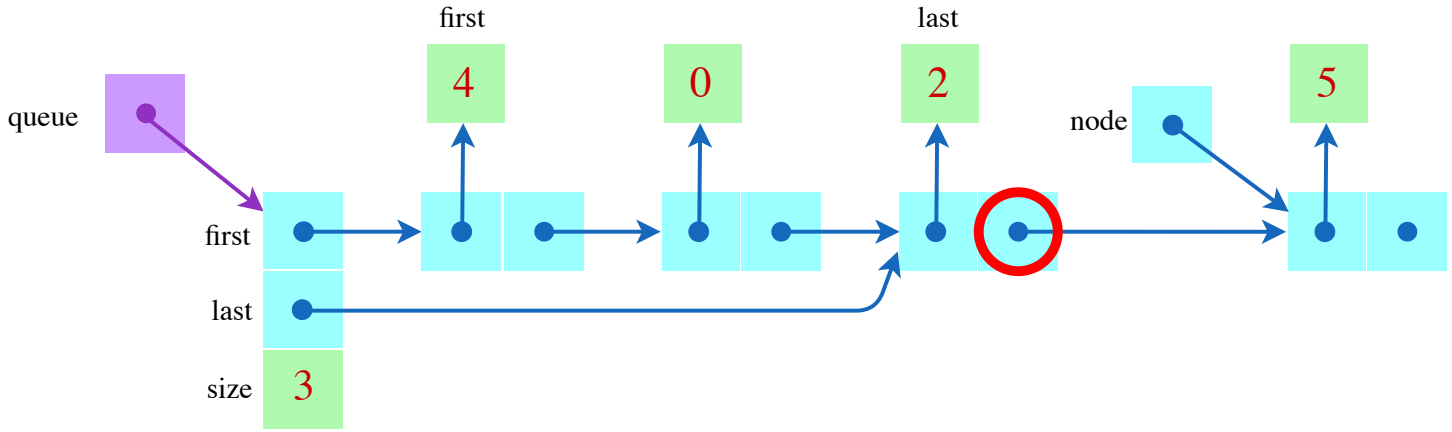
- Partiendo de esta configuración, vamos a poner en cola el elemento 5:
  ![center height:145px] (images/linked-queue-02.drawio.svg)
- Un nuevo `nodo` con el nuevo elemento (5) y se crea una referencia `null`:



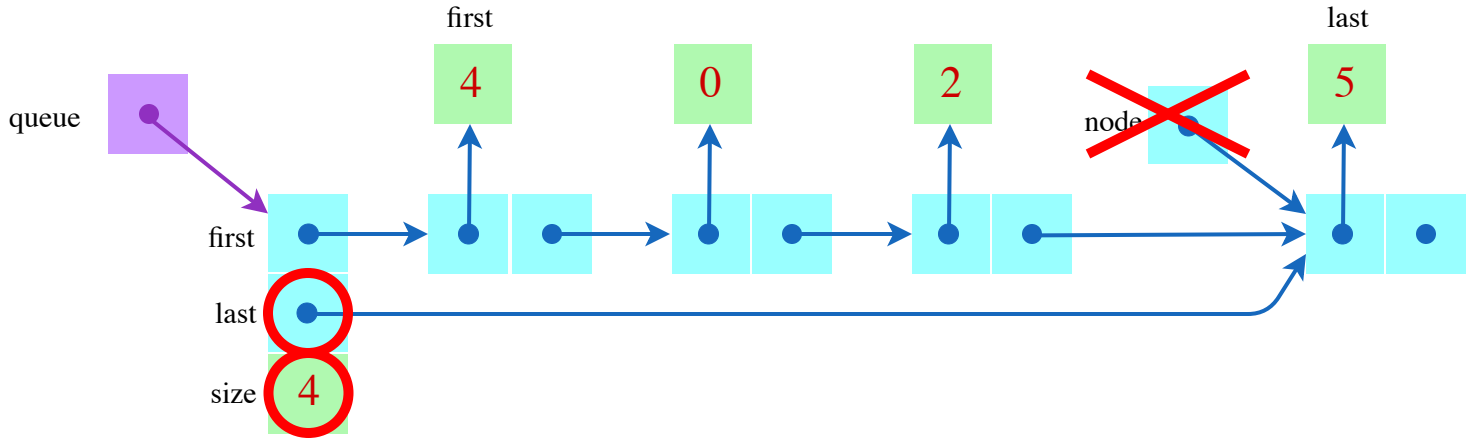
- El componente `siguiente` del nodo al que hace referencia `último` se actualiza para apuntar al nuevo `nodo`:



Poner en cola un elemento al final en `LinkedQueue` (II)



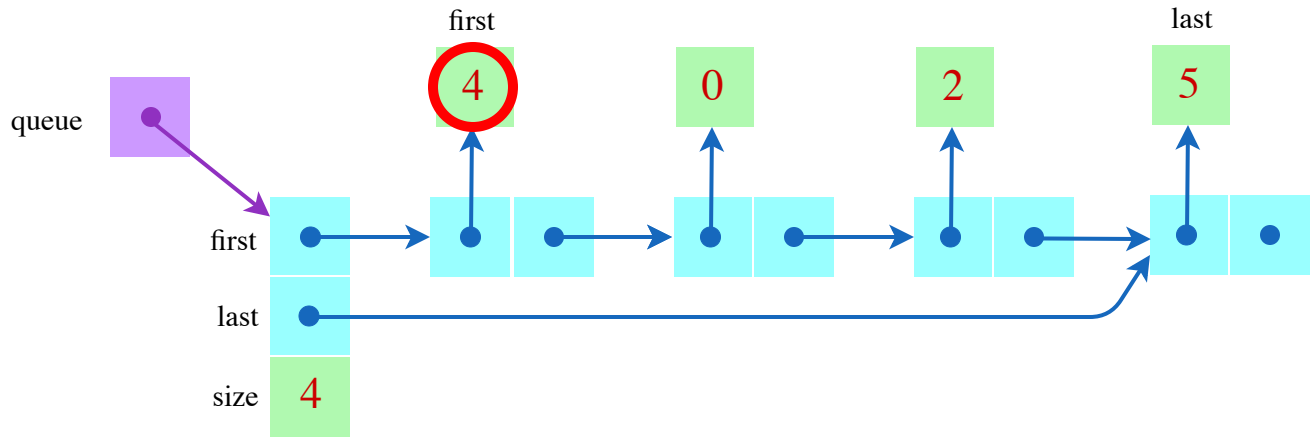
- `last` se actualiza para apuntar al nuevo `nodo` y `size` es incrementado:



- Si la cola estaba vacía, también se debe actualizar `first` a apunta al nuevo 'nodo'.

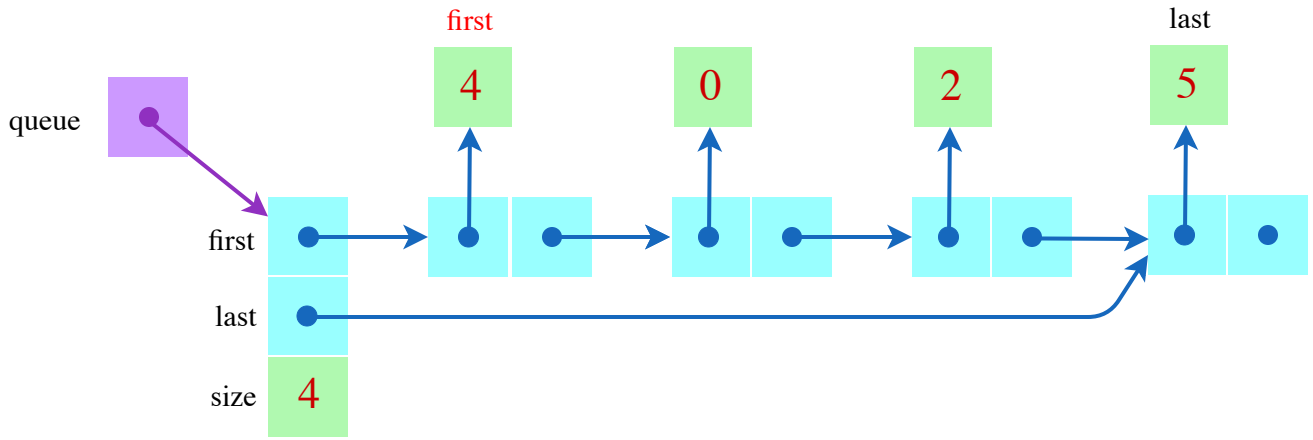
Accediendo al primer elemento en `LinkedQueue`

- Si la cola está vacía, se debe lanzar una `EmptyQueueException`.
- Si la cola no está vacía:
  - El primer elemento de la cola es el que ha estado en la cola por mas tiempo
  - El primer elemento es el almacenado en el nodo referenciado por `primero`.



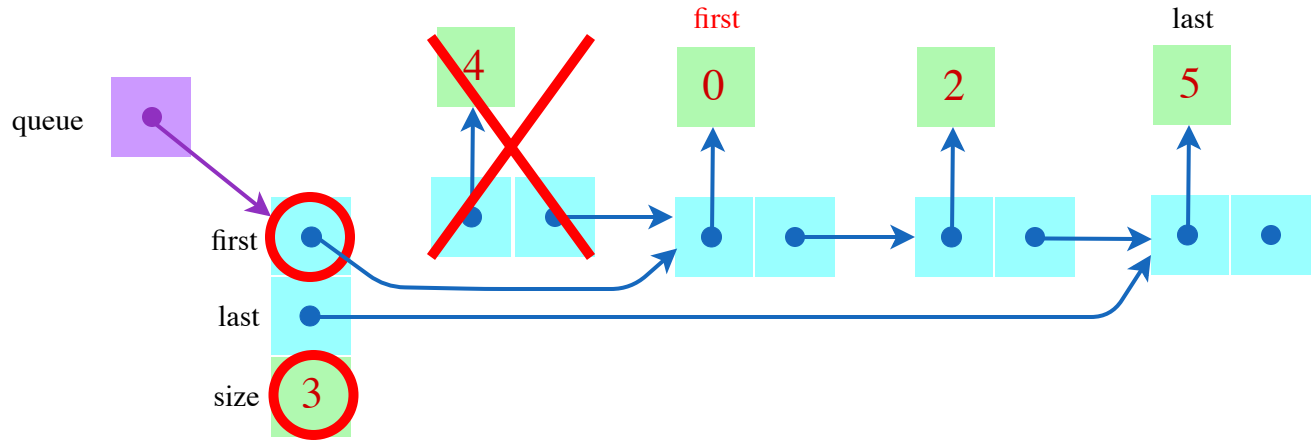
Quitar de la cola el primer elemento en `LinkedQueue`

- Si la cola está vacía, se debe lanzar una `EmptyQueueException`.
- Si la cola no está vacía, el primer elemento se elimina mediante:
  - Actualizar la referencia `primera` para apuntar al siguiente nodo.
  - El *recolector de basura* recuperará la memoria utilizada por el nodo eliminado.
  - Disminuyendo el tamaño.
- Si la cola se vacía, `last` también debe establecerse en `null`.
- Partiendo de esta configuración, vamos a 'desencolar' el primer elemento:



Quitar de la cola el primer elemento en `LinkedQueue` (II)

- Después de sacar de la cola:



Métodos de fábrica para `LinkedQueue`

- Se pueden utilizar métodos de fábrica para crear instancias de `LinkedQueue<T>`.
- `empty()`: construye una *cola vacía*.
- `of(T... elementos)`: construye una cola *previamente rellenada* con los elementos proporcionados, lo que permite una configuración de cola rápida y sencilla.
- `copyOf(Queue<T> queue)`: construye una nueva cola que es un *duplicado* de la `cola` dada, preservando el orden de los elementos.
- `from(Iterable<T> iterable)`: construye una nueva cola
  que contiene todos los elementos del *iterable* especificado, manteniendo su orden de iteración.

```
Queue<Integer> queue1 = LinkedQueue.empty();

Queue<Integer> queue2 = LinkedQueue.of(1, 2, 3);

Queue<Integer> queue3 = LinkedQueue.copyOf(queue2);
queue3.push(4);

Queue<Integer> queue4 = LinkedQueue.from(LinkedList.of(5, 6, 7));
int sum = 0;
while (!queue4.isEmpty()) {
  sum += queue4.first();
  queue4.dequeue();
}
```

Complejidad computacional de las operaciones de `LinkedQueue`

Operation	Cost
`empty`	O(1)
`enqueue`	O(1)
`dequeue`	O(1)
`first`	O(1)
`isEmpty`	O(1)
`size`	O(1)

Comparación experimental entre `ArrayQueue` y `LinkedQueue`

- Medimos el tiempo de ejecución al realizar 10 millones de operaciones aleatorias (`poner en cola` o `quitar de cola`) en una cola inicialmente vacía.
- Usando una CPU Intel i7 860 y JDK 22:
  - `ArrayQueue` fue aproximadamente 1,70 veces más rápido que `LinkedQueue`.

Colas. Aplicaciones

- **Programación de tareas**: se utiliza para almacenar tareas que se ejecutarán por orden de llegada.
- **Búsqueda en amplitud**: se utiliza para almacenar los nodos que se visitarán en un algoritmo de búsqueda en amplitud.
- **Almacenamiento en búfer**: se utiliza para almacenar datos antes de que se procesen.
- **Impresión**: Se utiliza para almacenar documentos que se imprimirán en el orden en que se enviaron a la impresora.
- **Paso de mensajes**: se utiliza para almacenar mensajes que se procesarán en el orden en que se recibieron.
- **Simulación**: Se utiliza para simular filas de espera en un sistema.