Data Structures. Abstract Data Types and Linear Data Structures

Luis LLopis, 2024.
Dpto. Lenguajes y Ciencias de la Computación.
University of Málaga

Tipos de datos abstractos (ADT): descripción general del concepto

¿Qué es un ADT?

- **Definición**: Un modelo para tipos de datos definidos por *comportamiento* e *interacción*, no por implementación.

Tipos de datos abstractos (ADT): descripción general del concepto

Aspectos clave de los ADT

- **Valores**: ¿Cuál es el rango de valores que pueden contener los datos?
- **Operaciones**: ¿Qué operaciones se pueden realizar sobre los datos?
- **Comportamiento**: ¿Cómo interactúan estas operaciones con los datos?

Tipos de datos abstractos (ADT): descripción general del concepto

ADT frente a estructura de datos

- **ADT**: Concepto teórico, centrado en el '*qué*'.
- **Estructura de datos**: Implementación práctica, centrándose en el '*cómo*'.

Beneficios de los tipos de datos abstractos (ADT)

- **Encapsulación**: Los ADT 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 ADT se pueden implementar de múltiples maneras, lo que proporciona flexibilidad para elegir la mejor *implementación* para un problema determinado.

El TAD Pila

- Una *pila* es una colección que almacena elementos en un orden de último en entrar, primero en salir (*LIFO*).
- **Operaciones**:
  - `push` : agrega un elemento a la parte superior de la pila.
  - `pop` : elimina el elemento superior de la pila.
  - `top` : Devuelve el elemento superior de la pila sin eliminarlo.

- **Operaciones**:
  - `isEmpty` : Comprueba si la pila está vacía.
  - `size` : Devuelve el número de elementos en la pila.
  - `clear` : elimina todos los elementos de la pila.

Stack of books

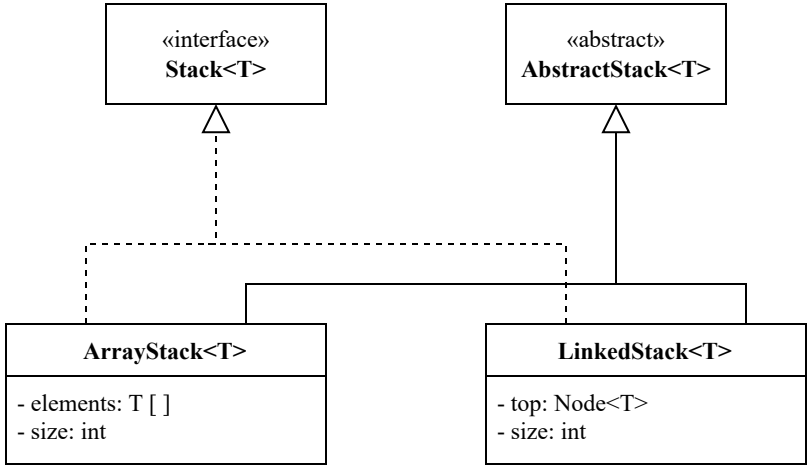
El TAD de pila en Java

- En Java, los ADT normalmente se especifican mediante *interfaces*.
- La interfaz `Stack<T>` define una pila con elementos de tipo `T`.

```java
package dataStructures.stack;

public interface Stack<T> {
  void push(T element);
  T top();
  void pop();
  boolean isEmpty();
  int size();
  void clear();
}
```

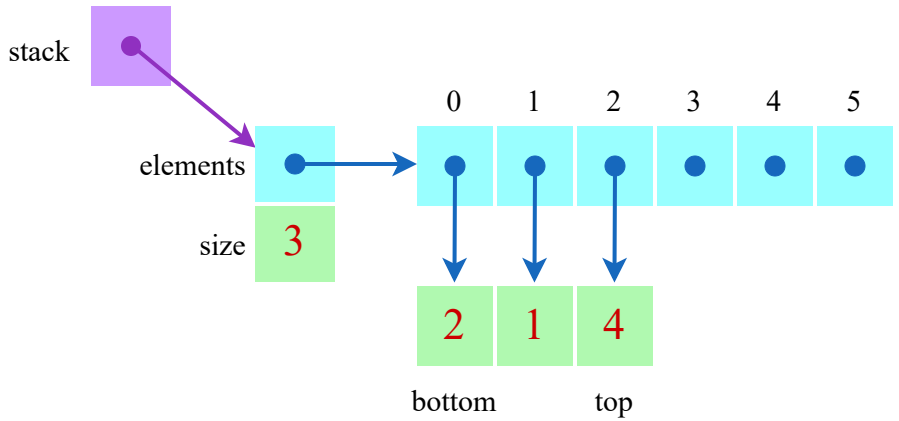
Implementaciones del TAD de pila

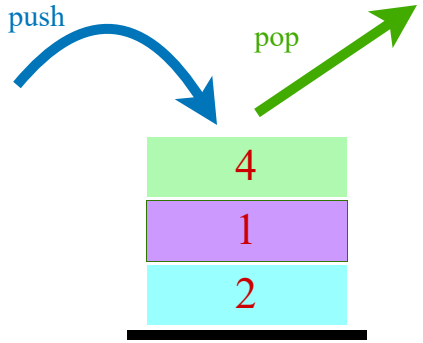
- Una pila se puede implementar utilizando diferentes *estructuras de datos*.
- Diferentes *clases* pueden implementar la interfaz `Stack<T>`:
  - `ArrayStack<T>`: utiliza un array para almacenar elementos.
  - `LinkedStack<T>`: utiliza una estructura enlazada para almacenar elementos.
- La clase abstracta base `AbstractStack<T>` proporciona implementación para los métodos `equals`, `hashCode` y `toString`.



La clase `ArrayStack`

- `ArrayStack<T>` implementa la interfaz `Stack<T>` utilizando una matriz para almacenar elementos.
- Inicialmente, la matriz tiene un tamaño fijo (*capacidad* de la pila), pero puede crecer dinámicamente cuando sea necesario.
- A medida que se introducen nuevos elementos en la pila, se almacenan en la matriz en orden *de izquierda a derecha*.
- La clase también mantiene una variable entera `size` para realizar un seguimiento de la cantidad de elementos en la pila.
- Esta variable `size` también se utiliza para realizar un seguimiento de la *primera posición libre* en la matriz.


stack
elements
size
3
0
1
2
3
4
5
2
1
4
bottom
top

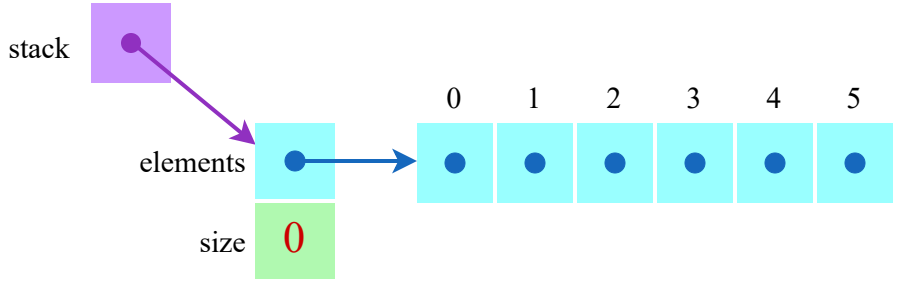

push
pop
4
1
2

Implementación de `ArrayStack`

```
package dataStructures.stack;

public class ArrayStack<T> extends AbstractStack<T> implements Stack<T> {
  private static final int DEFAULT_INITIAL_CAPACITY = 6;

  private T[] elements;
  private int size;

  public ArrayStack(int initialCapacity) {  // ArrayStack constructor
    if (initialCapacity <= 0) {
      throw new IllegalArgumentException("initial capacity must be greater than 0");
    }
    elements = (T[]) new Object[initialCapacity];
    size = 0;
  }

  public ArrayStack() {  // ArrayStack constructor
    this(DEFAULT_INITIAL_CAPACITY);
  }

  ...
}
```

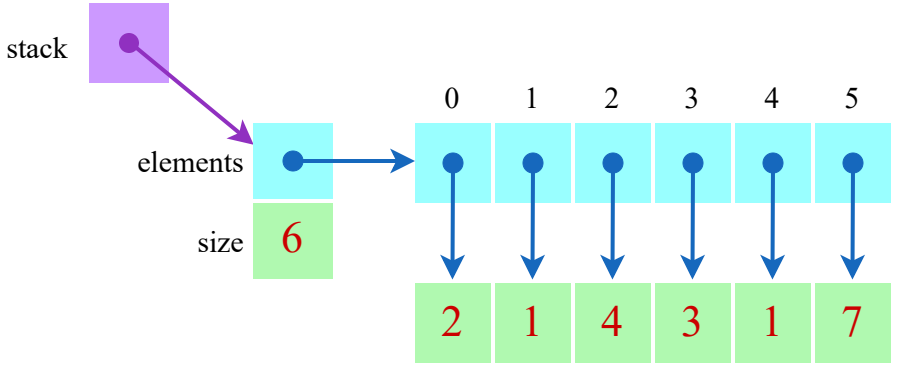
Inicialización de `ArrayStack`

- Suponiendo que la *capacidad inicial* es 6, se crea la matriz. Cuando se construye una matriz de tipos de referencia con `new`, la máquina virtual Java (JVM) inicializa automáticamente todos los elementos con `null`.

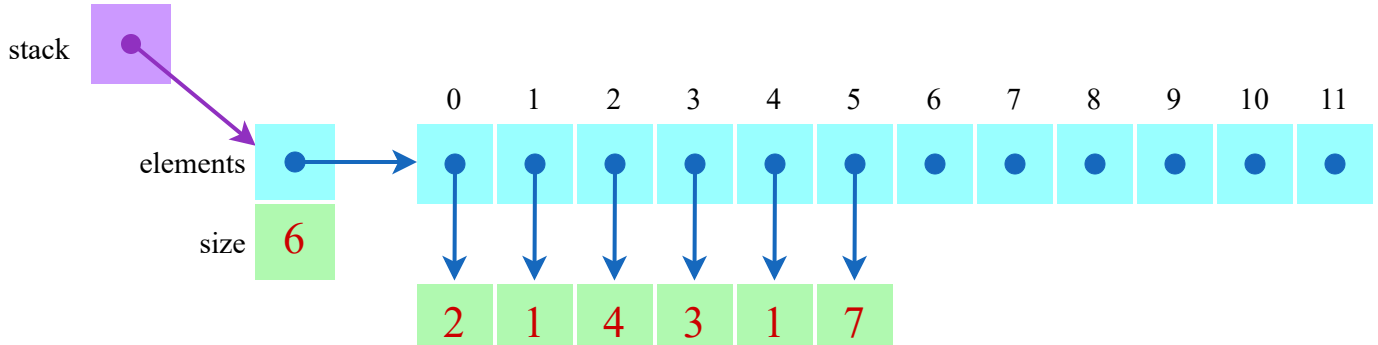


Garantizar la capacidad en `ArrayStack`

- 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.

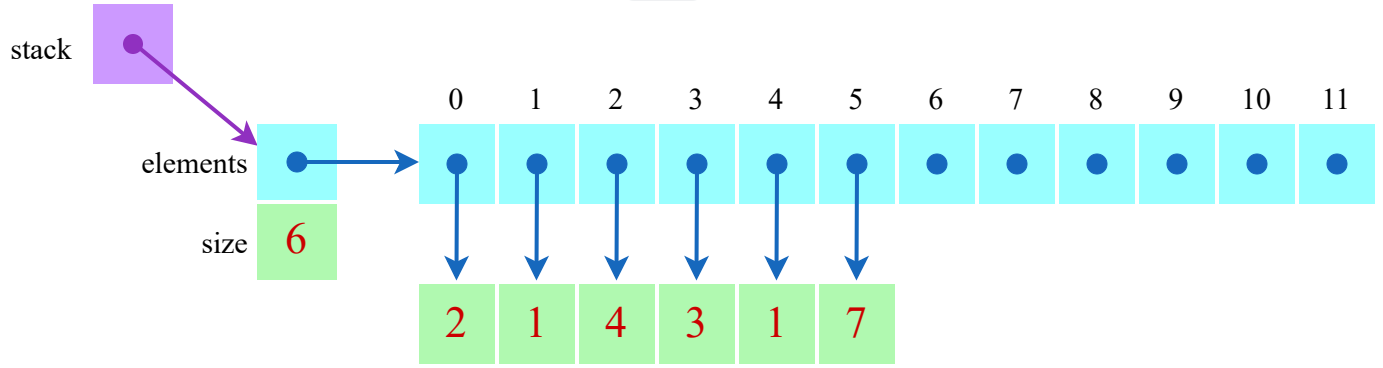


- El método `Arrays.copyOf(oldArray, newLength)` asigna una nueva matriz que conserva todos los elementos de `oldArray` y expande su capacidad a `newLength`.

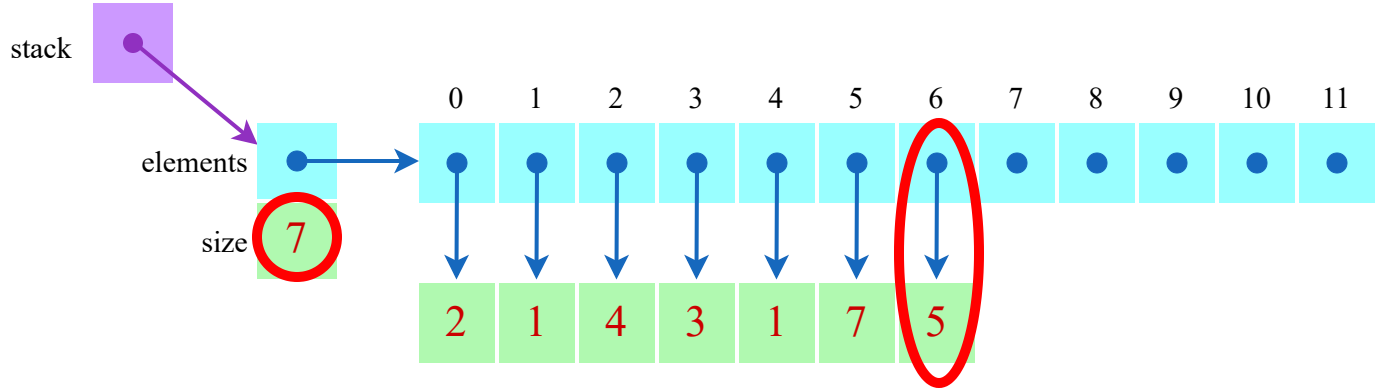


Insertar un elemento en `ArrayStack`

- Para insertar un elemento en la pila, se realizan los siguientes pasos:
  - **Asegurar la capacidad**: Verifique que la matriz tenga suficiente capacidad para alojar el nuevo elemento. En caso contrario, duplica el tamaño de la matriz.
  - **Almacenar elemento**: coloca el nuevo elemento en el índice especificado por el `tamaño` actual.
  - **Incrementar tamaño**: aumenta el `tamaño` en uno para reflejar el nuevo recuento de elementos en la pila.
- Partiendo de esta configuración vamos a hacer `push` 5 en la pila:

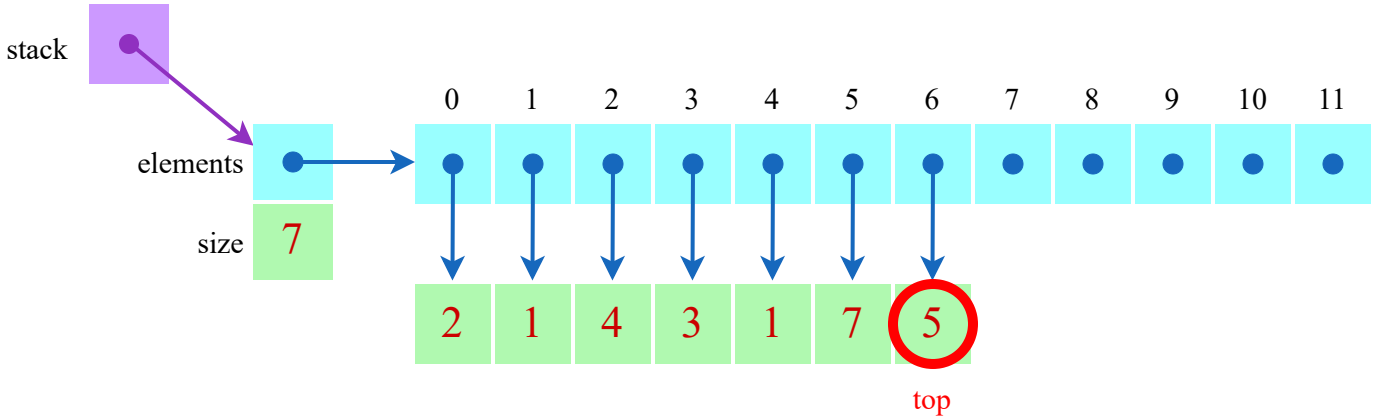


- Después de insertar 5:



Accediendo al elemento superior en `ArrayStack`

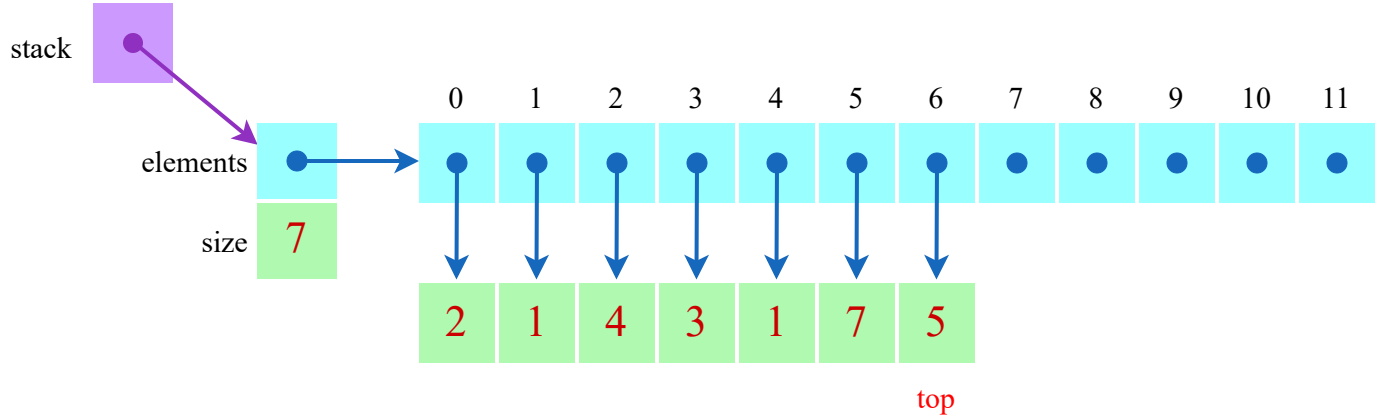
- Si la pila está vacía, se debe lanzar una `EmptyStackException`.
- Si la pila no está vacía:
  - El elemento superior es el que se ha introducido más recientemente en el pila.
  - Este elemento se almacena en la posición `size - 1` y debe ser devuelto.
- Aquí hay una representación visual del acceso al elemento superior de la pila:



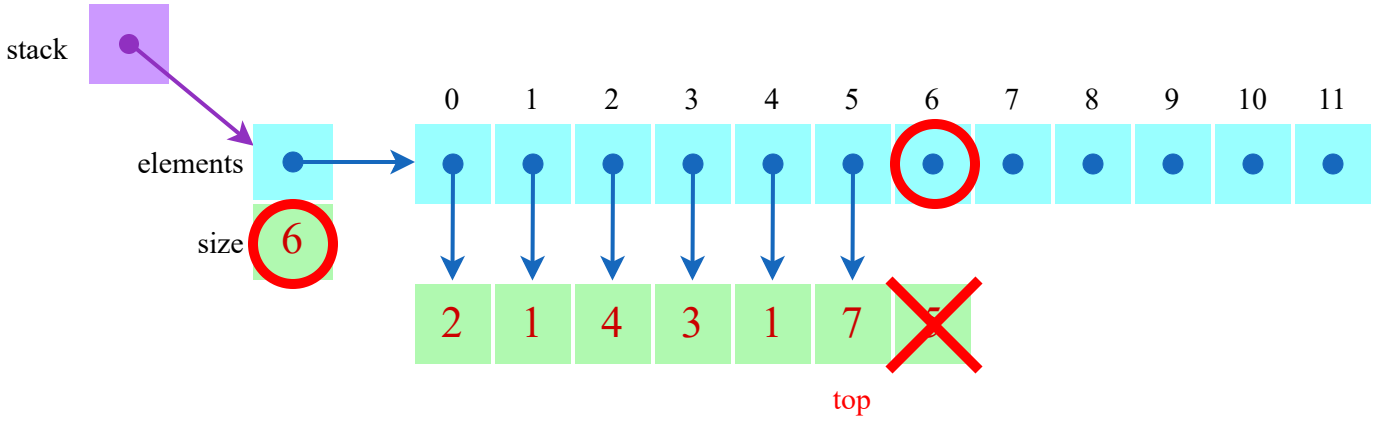
Eliminando el elemento superior en `ArrayStack`

Para implementar correctamente la operación `pop`, podemos seguir estos pasos:

- **Verificar si la pila está vacía**: si la pila está vacía, lanza una `EmptyStackException`.
- **Quitar elemento superior**:
- **Disminuir `tamaño`**: Reduce el `tamaño` en 1 para eliminar el elemento superior de la pila.
- **Borrar referencia**: Establece el elemento en el valor ahora decrementado
  El índice `size` se convierte en `null`. Esto eliminará el elemento superior y permitirá que el *recolector de basura* recupere la memoria utilizada por el elemento eliminado.
- Partiendo de esta configuración, vamos a hacer `pop` en el elemento superior de la pila:



- Después de extraer:



Métodos de fábrica para `ArrayStack`

- Los métodos de fábrica ofrecen una forma conveniente de crear instancias de objetos `ArrayStack<T>` sin invocar directamente constructores.
  Estos métodos incluyen:
  - `empty()`: construye una *pila vacía* con una capacidad inicial *predeterminada*, adecuada para cuando se desconoce el número esperado de elementos.
  - `withCapacity(int initialCapacity)`: construye una *pila vacía* con una capacidad inicial especificada, optimizando la asignación de memoria para un número conocido de elementos.
  - `of(T... elementos)`: construye una pila *previamente rellena* con
    los elementos proporcionados, lo que permite una configuración de pila rápida y sencilla.

Métodos de fábrica para `ArrayStack`

- `copyOf(Stack<T> stack)`: construye una nueva pila que es un *duplicado* de la `pila` dada, preservando el orden de los elementos.
- `from(Iterable<T> iterable)`: construye una nueva pila que contiene todos los elementos del *iterable* especificado, manteniendo su orden de iteración.

```
Stack<Integer> stack1 = ArrayStack.empty(); // Crea una pila vacía con capacidad inicial predeterminada
Stack<Integer> stack2 = ArrayStack.of(1, 2, 3); // Crea una pila que contiene los elementos 1, 2 y 3
Stack<Integer> stack3 = ArrayStack.copyOf(stack2); // Crea una copia de stack2 y coloca el elemento 4 sobre ella
pila3.push(4);
// Crea una pila a partir de una lista de elementos y calcula su suma
Pila<Entero> pila4 = ArrayStack.from(LinkedList.of(5, 6, 7));
int suma = 0;
while (!stack4.isEmpty()) {
  suma += pila4.top();
  pila4.pop();
}
```

Matrices. Complejidad computacional en Java

- Para evaluar la complejidad computacional de las estructuras de datos que utilizan matrices, como la clase `ArrayStack`, debemos considerar el coste de las operaciones de las matrices:
    - **Acceso a un elemento por índice**: O(1)
    - **Establecer un elemento por índice**: O(1)
    - **Asignación de una matriz**: O(n)
    - **Copiar una matriz**: O(n)

Complejidad computacional de las operaciones de `ArrayStack`

Operation	Cost
`empty`	O(1) †
`push`	O(1), O(n) §
`pop`	O(1)
`top`	O(1)
`isEmpty`	O(1)
`size`	O(1)
`clear`	O(n) *

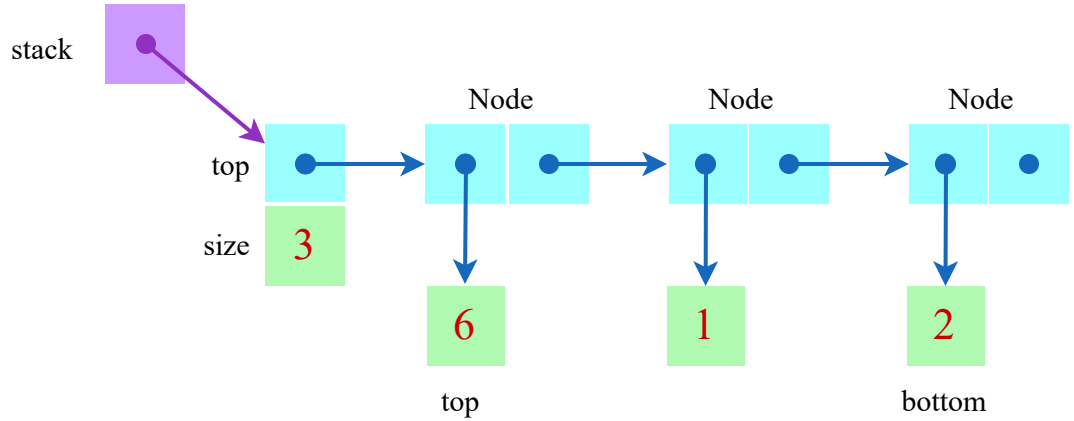
† In `empty` the size of the created array is a constant.

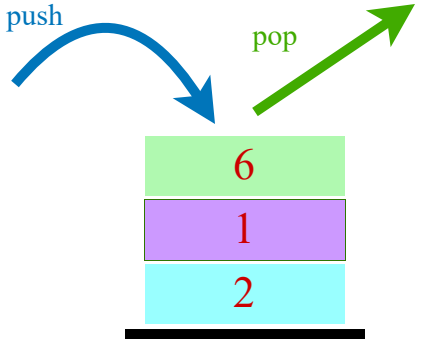
§ `push` will need to copy n elements when the array has to be enlarged.

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

La clase `LinkedStack`

- `LinkedStack<T>` implementa la interfaz `Stack<T>` utilizando una estructura enlazada de nodos.
- A medida que se introducen nuevos elementos en la pila, se insertan nuevos nodos al **principio** de la estructura vinculada.
- La clase mantiene una referencia `top` al primer nodo en la estructura vinculada que corresponde a la parte superior de la pila.
- Cada nodo contiene un elemento y una referencia (`next)` al nodo que contiene el elemento debajo de él en la pila.
- El último nodo corresponde a la *parte inferior* de la pila y tiene su referencia `siguiente` establecida en `null`.
- La clase también mantiene una variable entera `size` para realizar un seguimiento de la cantidad de elementos en la pila.



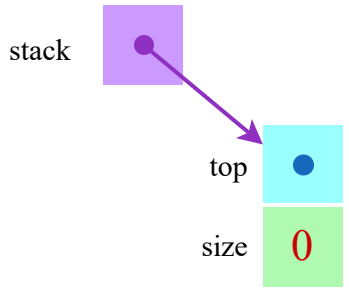


Implementación de `LinkedStack`

```
package dataStructures.stack;

public class LinkedStack<T> extends AbstractStack<T> implements Stack<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> top;
  private int size;

  public LinkedStack() {  // LinkedStack constructor
    top = null;
    size = 0;
  }

}
```

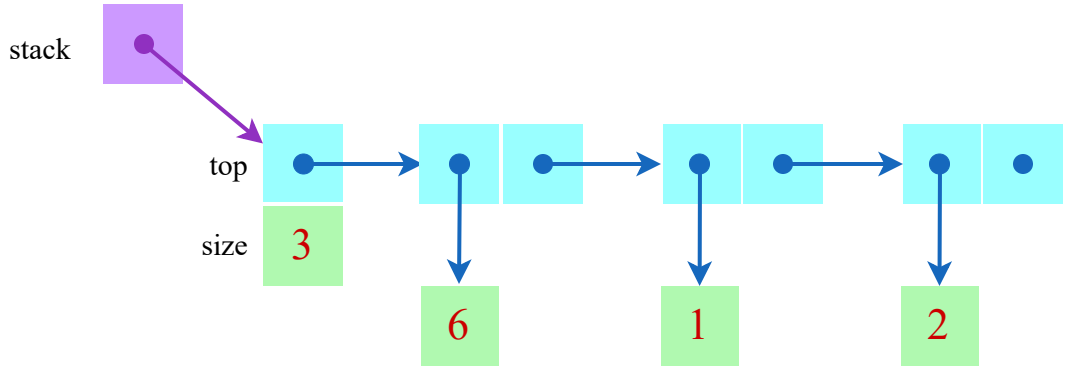
Inicialización de `LinkedStack`:

- Cuando la pila está vacía, `top` es `null` y `size` es 0.

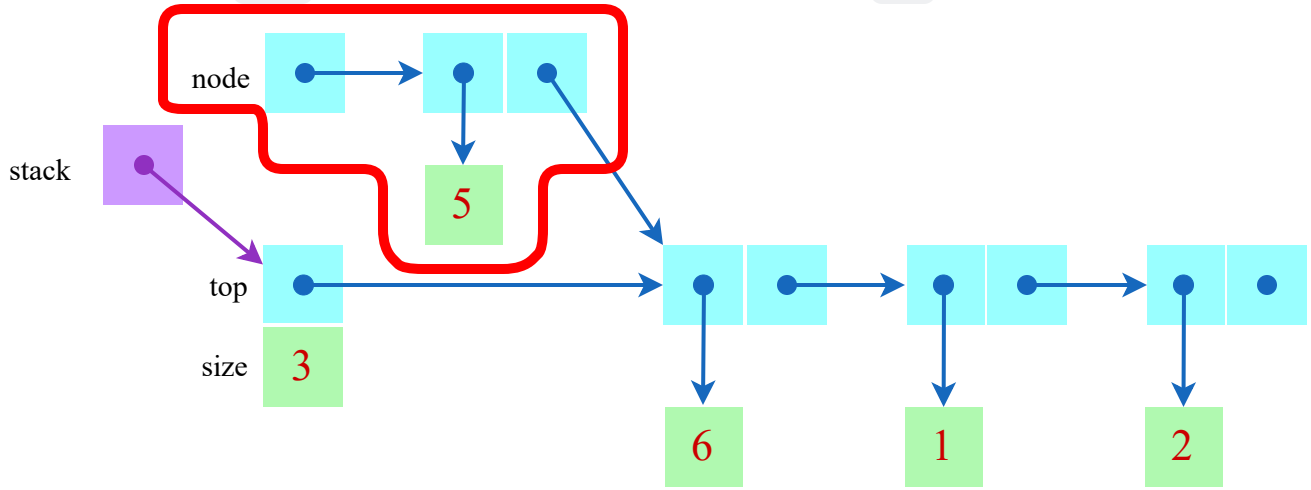


Insertar un elemento en la sección `LinkedStack`

- Partiendo de esta configuración, vamos a `push` el elemento 5:

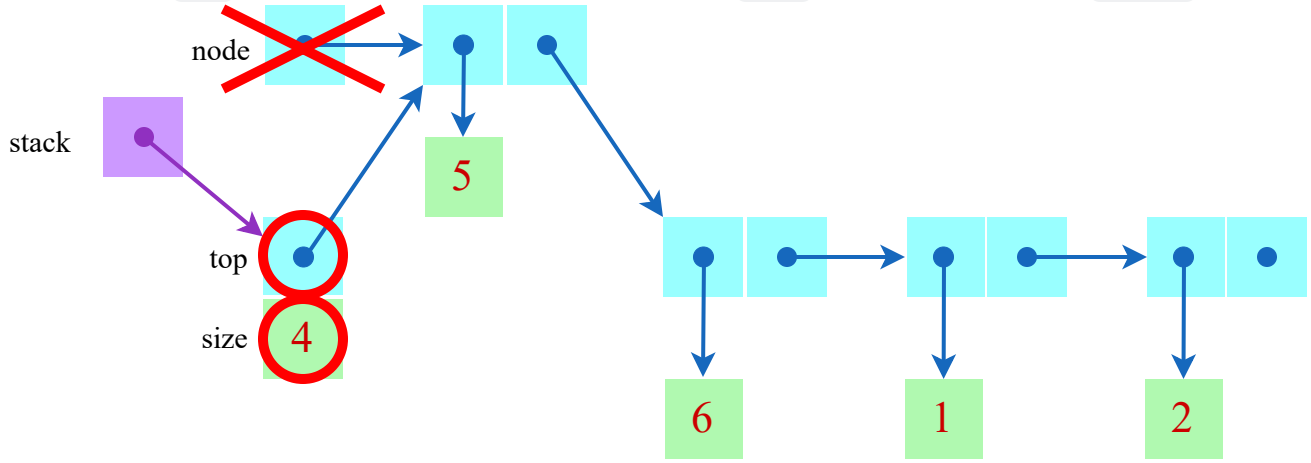


- Se crea un nuevo `nodo` con el elemento (5) y una referencia al `top` actual:



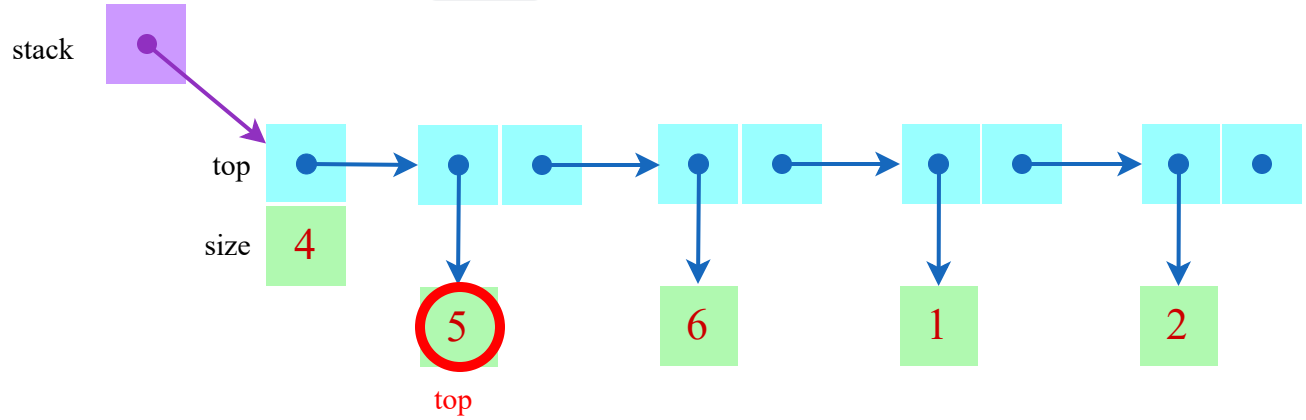
Insertar un elemento en la sección `LinkedStack`

- La referencia `top` se actualiza para apuntar al nuevo `nodo` y se incrementa el `tamaño`:



Acceso al elemento superior en `LinkedStack`:

- El elemento superior es el último elemento insertado en la pila.
- Si la pila está vacía, se debe lanzar una `EmptyStackException`.
- Si la pila no está vacía, el elemento superior es el que se almacena en el nodo referenciado por `top`.

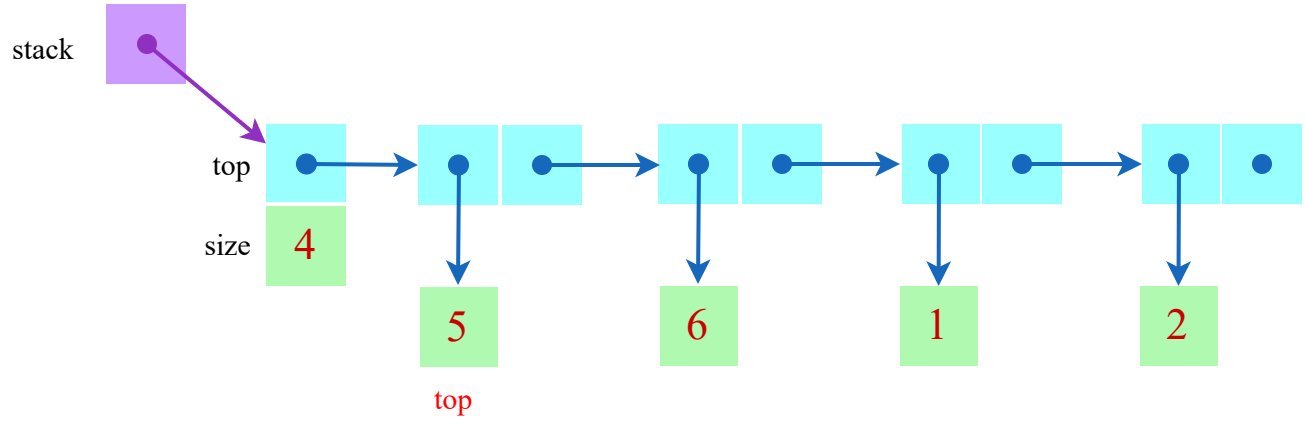


Cómo eliminar el elemento superior en `LinkedStack`

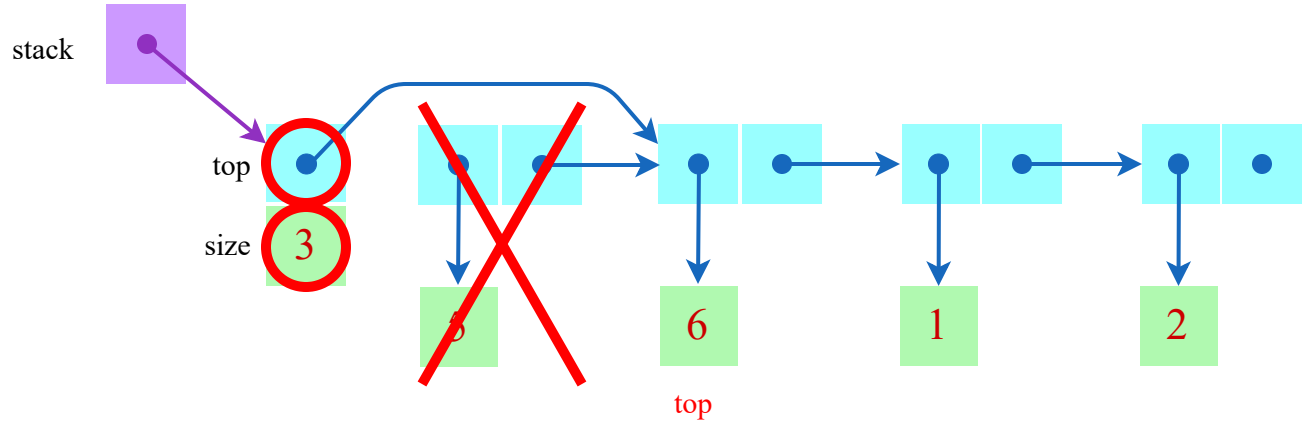
- `pop` elimina el elemento `top` de la pila.
- Si la pila está vacía, se debe lanzar una `EmptyStackException`.
- Si la pila no está vacía:
  - El elemento superior está en el nodo referenciado por `top`.
  - La referencia `top` debe actualizarse para apuntar al siguiente nodo.
  - El *recolector de basura* recuperará la memoria utilizada por el nodo eliminado
  - `size` debe reducirse para reflejar el nuevo recuento de elementos en la pila.

Cómo eliminar el elemento superior en `LinkedStack`

- Partiendo de esta configuración, vamos a hacer "pop" del elemento superior de la pila:



- Después de hacer estallar el elemento superior:



Métodos de fábrica para `LinkedStack`

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

Métodos de fábrica para `LinkedStack`

```
Stack<Integer> stack1 = LinkedStack.empty();

Stack<Integer> stack2 = LinkedStack.of(1, 2, 3);

Stack<Integer> stack3 = LinkedStack.copyOf(stack2);
stack3.push(4);

Stack<Integer> stack4 = LinkedStack.from(LinkedList.of(5, 6, 7));
int sum = 0;
while (!stack4.isEmpty()) {
  sum += stack4.top();
  stack4.pop();
}
```

Complejidad computacional de las operaciones de «LinkedStack»

Operation	Cost
`empty`	O(1)
`push`	O(1)
`pop`	O(1)
`top`	O(1)
`isEmpty`	O(1)
`size`	O(1)

Comparación experimental entre `ArrayStack` y `LinkedStack`

- Medimos el tiempo de ejecución al realizar 10 millones de operaciones aleatorias (`push` o `pop`) en una pila inicialmente vacía.
- Usando una CPU Intel i7 860 y JDK 22:
  - `ArrayStack` fue aproximadamente 1,50 veces más rápido que `LinkedStack`.

Pilas. Aplicaciones

- **Llamadas a funciones**: se utilizan para almacenar información sobre llamadas a funciones (parámetros, resultados, direcciones de retorno, etc.).
- **Evaluación de expresiones**: se utiliza para evaluar expresiones aritméticas (algoritmo de dos pilas de Dijkstra)
- **Búsqueda en profundidad**: se utiliza para almacenar los nodos que se visitarán en un algoritmo de búsqueda en profundidad.
- **Mecanismo de deshacer**: Se utiliza para almacenar el historial de operaciones para permitir deshacerlas.
- **Verificación de sintaxis**: se utiliza para comprobar la corrección de paréntesis, corchetes y llaves en un programa.
- **Expresar recursión**: se utiliza para simular recursión cuando el lenguaje de programación no la admite.

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

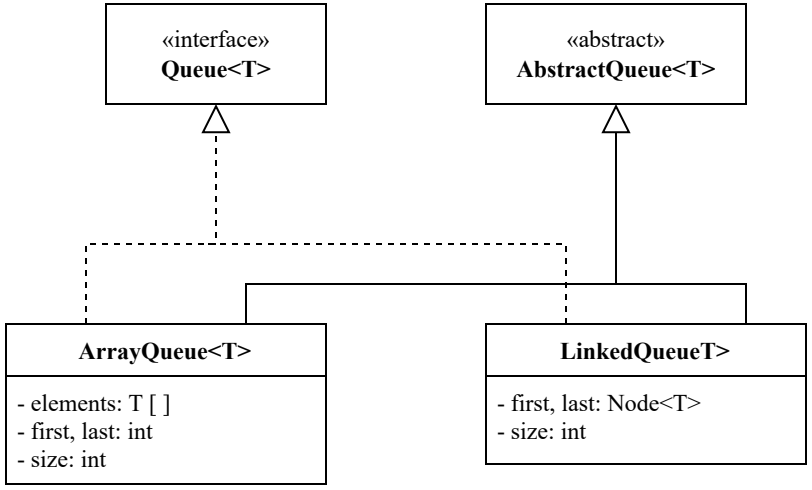
El TAD cola en Java: la interfaz

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

```
package dataStructures.queue;

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

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 matriz 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`.

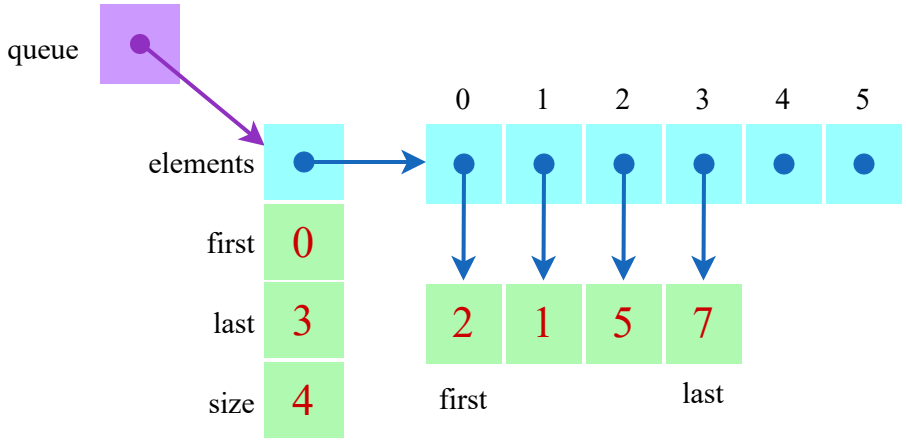


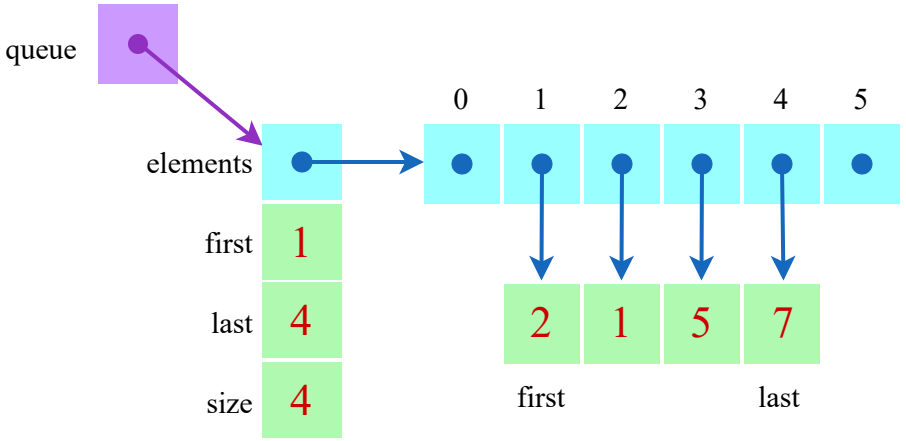
La clase `ArrayQueue`

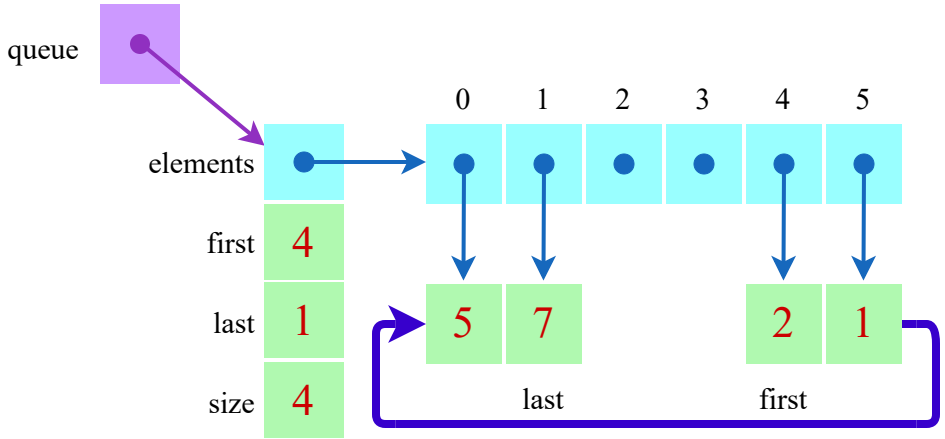
- `ArrayQueue<T>` implementa la interfaz `Queue<T>` utilizando una matriz para almacenar elementos.
- Inicialmente, la matriz tiene un tamaño fijo (**capacidad** de la cola), pero puede crecer dinámicamente cuando sea necesario.
- La clase mantiene dos índices enteros: `first` y `last`. Estos índices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





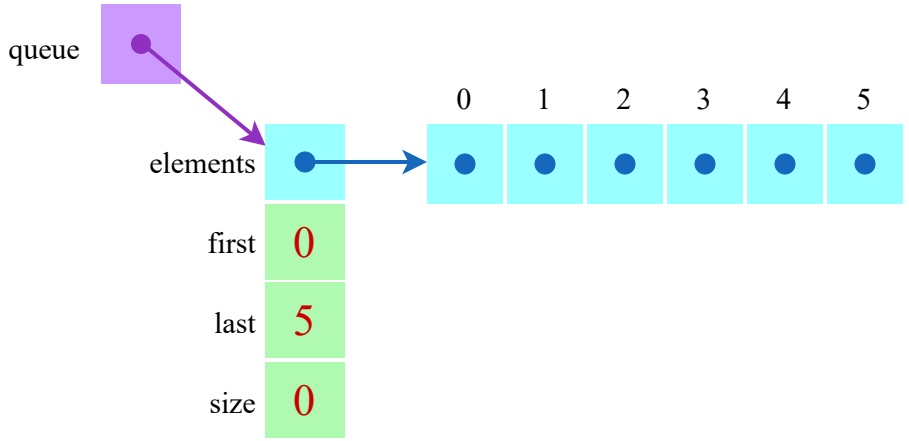


Implementación de `ArrayQueue`

```
package dataStructures.queue;

public class ArrayQueue<T> extends AbstractQueue<T> implements Queue<T> {
  private static final int DEFAULT_INITIAL_CAPACITY = 6;

  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);
  }
  ...
}
```

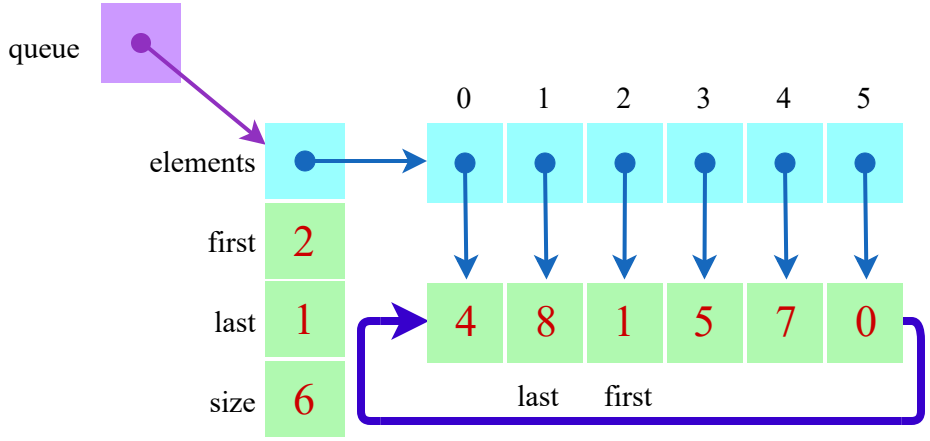
Inicialización de `ArrayQueue`

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

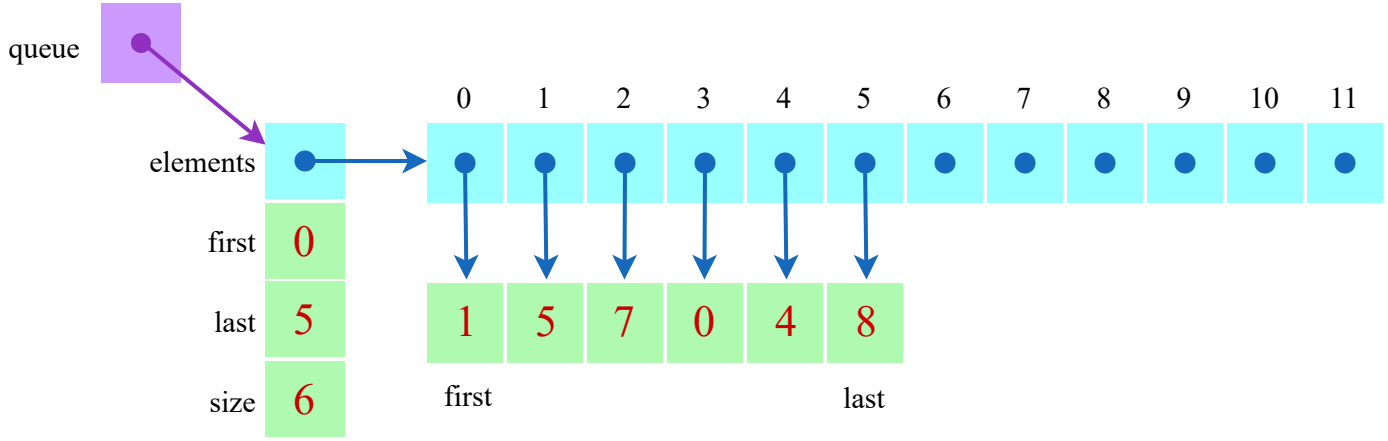


Garantizar la capacidad en `ArrayQueue`

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

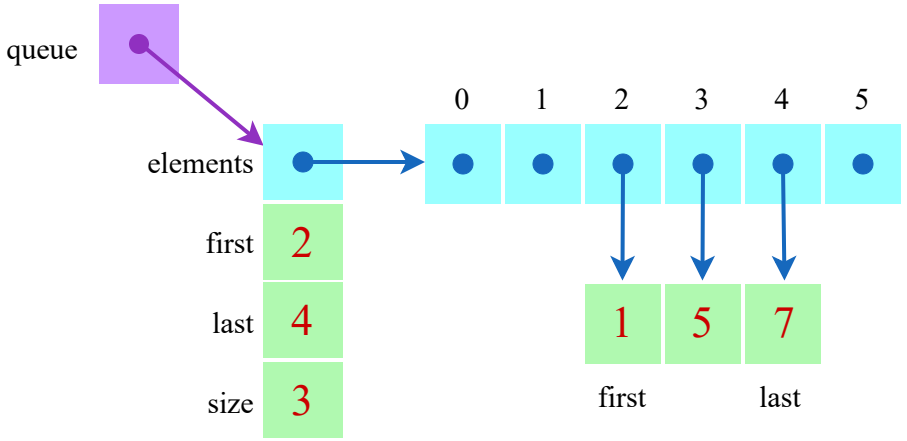


Poner un elemento al final de `ArrayQueue`

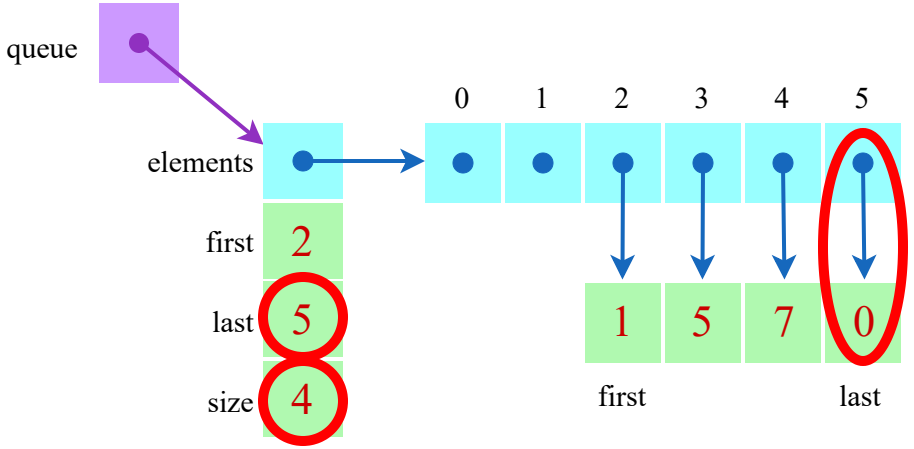
- `enqueue` agrega un elemento *después* del último elemento en la cola:
  - Se debe asegurar la capacidad del array para alojar el nuevo elemento.
  - `last` se incrementa para apuntar a la siguiente posición disponible (retrocediendo al índice 0 si `last` está al final de la matriz)
  - La matriz almacena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

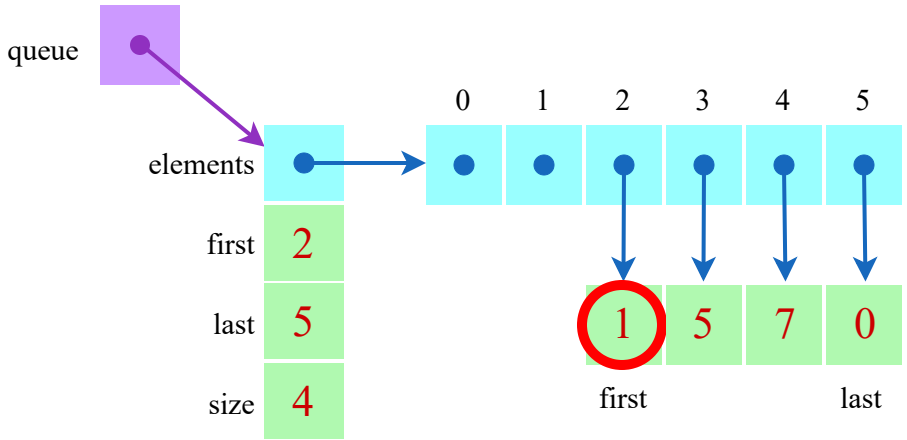


- Después de poner en cola 0:



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.

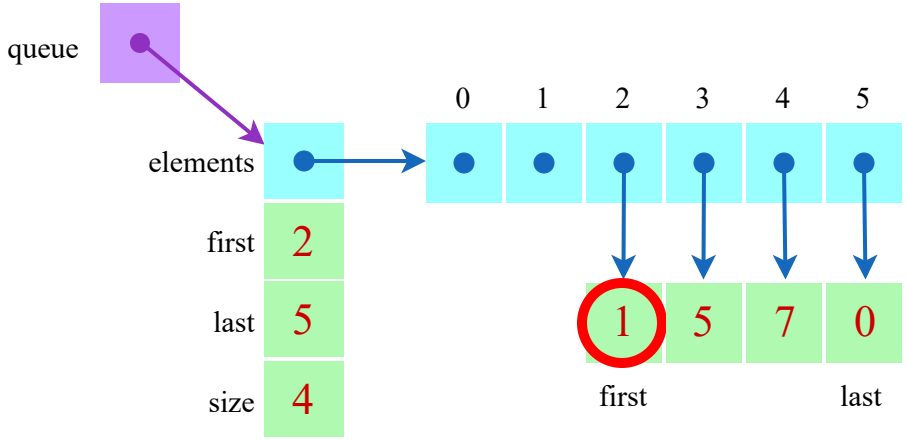


Quitar de la cola el primer elemento en `ArrayQueue`

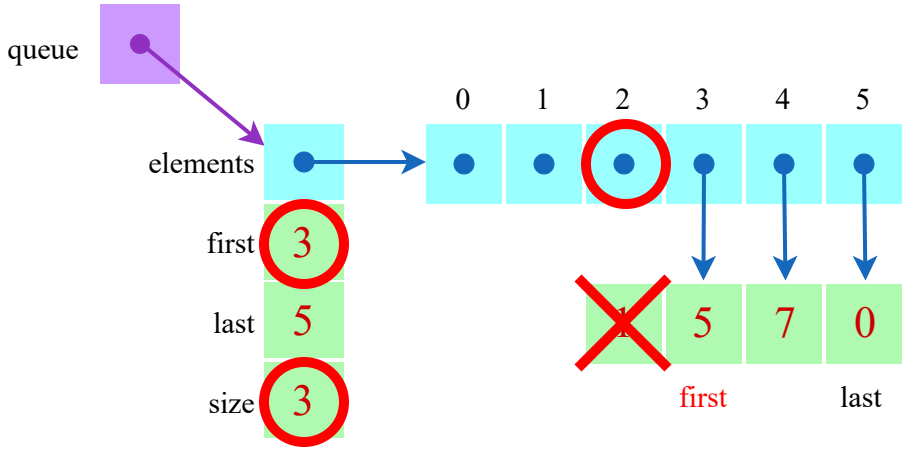
- Si la cola está vacía, se debe lanzar una `EmptyQueueException`.
- Si la cola no está vacía, el primer elemento se elimina mediante:
  - Establecer el elemento en la posición `first` como `null`. Esto elimina el primer elemento y permitirá que el *recolector de basura* recupere la memoria utilizada 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 matriz)
  - 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) *

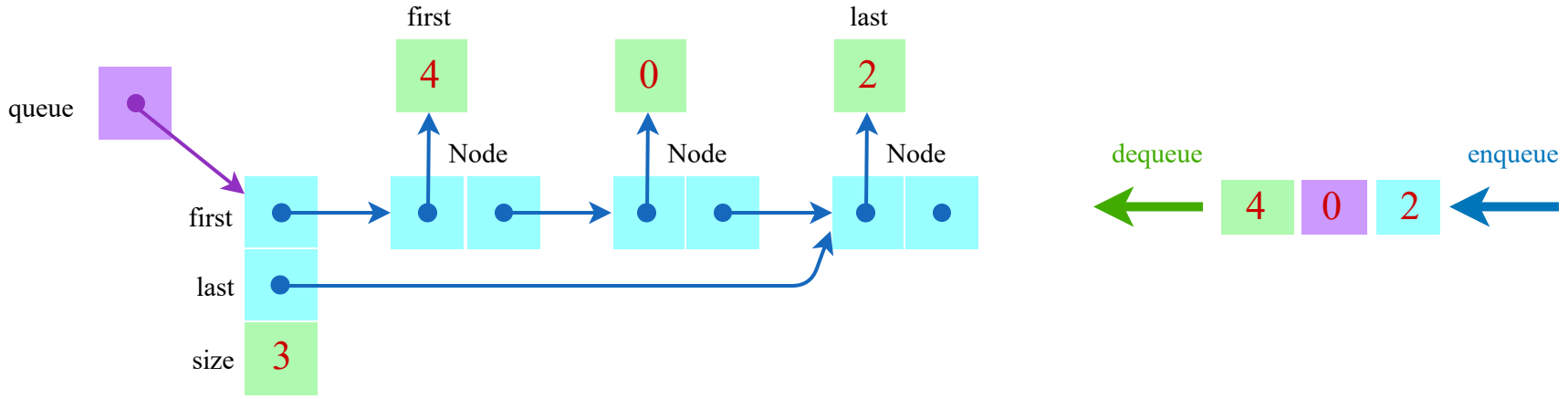
† In `empty` the size of the created array is a constant.

§ `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.

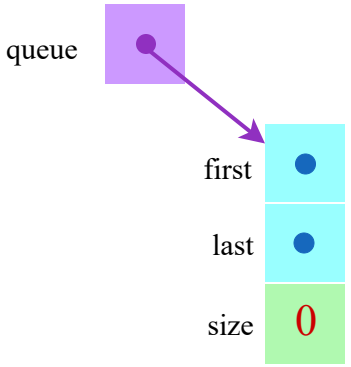


Implementación de `LinkedQueue`

```
package dataStructures.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`

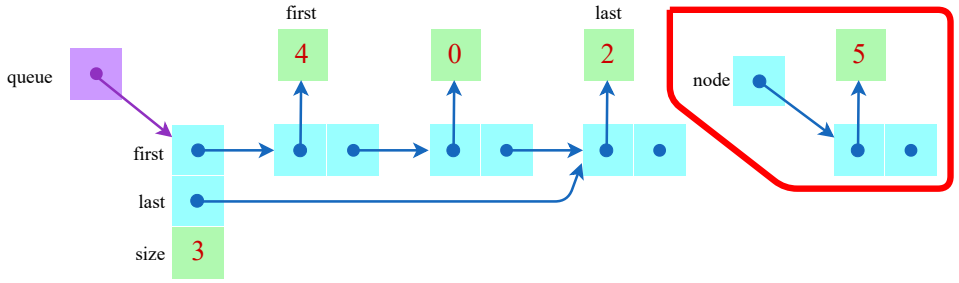
- Cuando la cola está vacía:
  - `first` y `last` son `null`.
  - `size` es 0



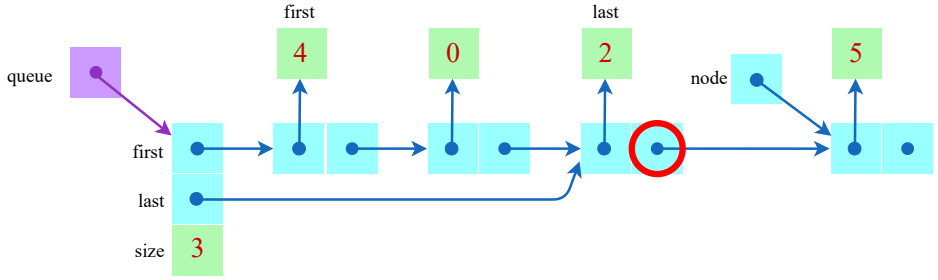
Poner en cola un elemento al final en `LinkedQueue`

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

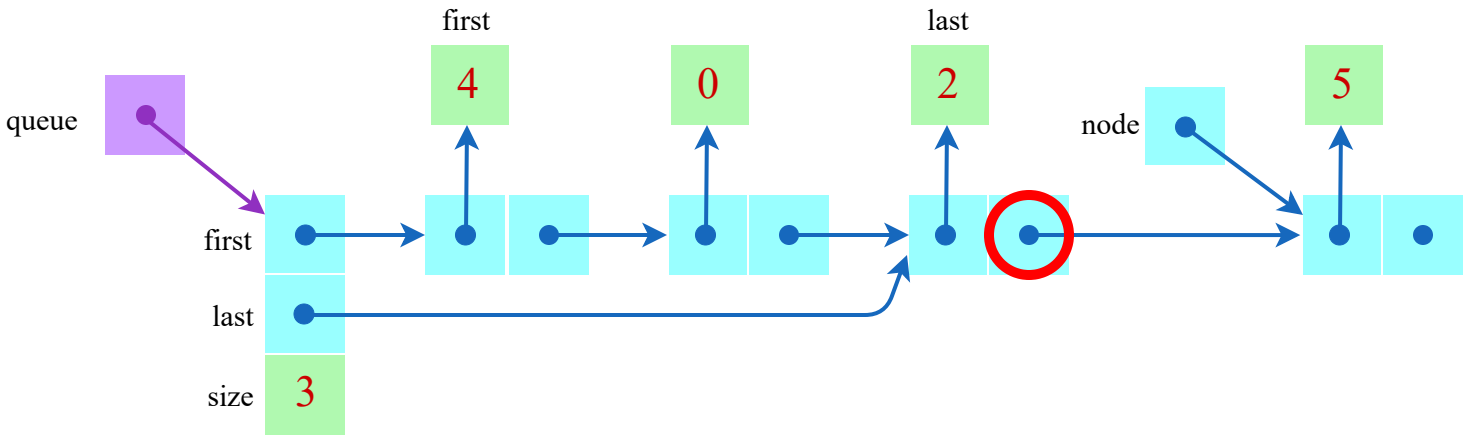
- Partiendo de esta configuración, vamos a poner en cola el elemento 5:
  ![center height:145px] (images/linked-queue-02.drawio.svg)
- Un nuevo `nodo` con el nuevo elemento (5) y se crea una referencia `null`:



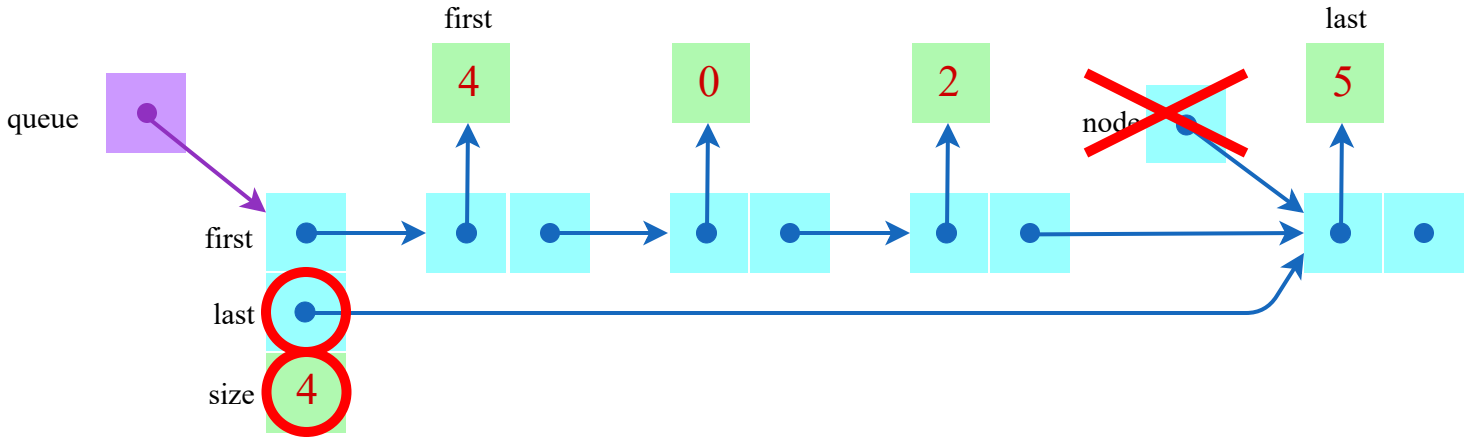
- El componente `siguiente` del nodo al que hace referencia `último` se actualiza para apuntar al nuevo `nodo`:



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



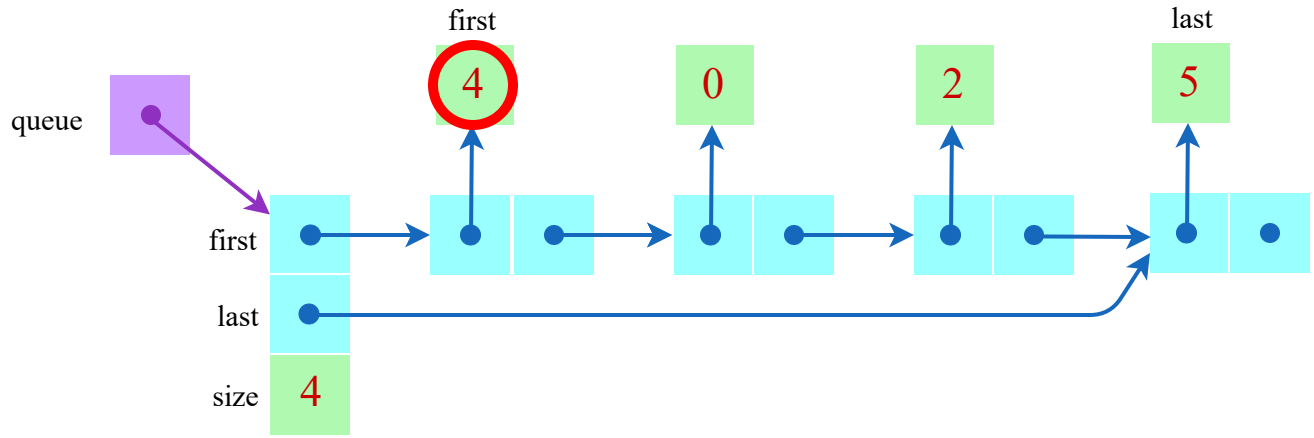
- `last` se actualiza para apuntar al nuevo `nodo` y `size` es incrementado:



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

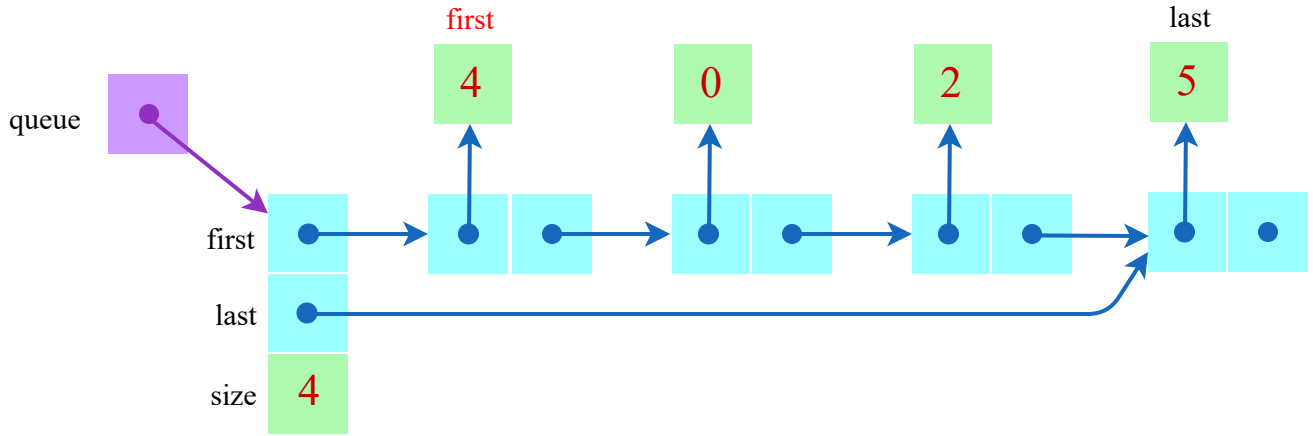
Accediendo al primer elemento en `LinkedQueue`

- 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`.



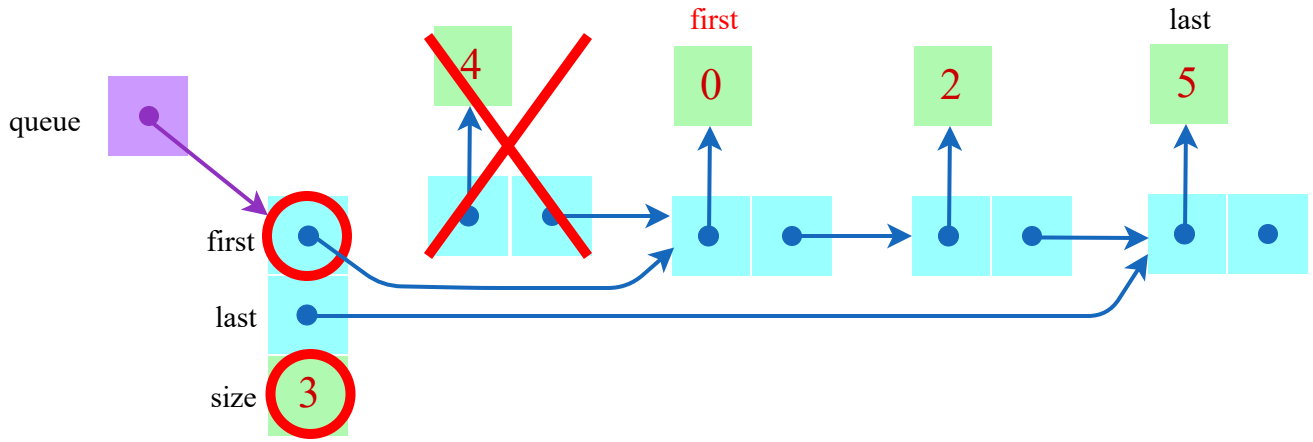
Quitar de la cola el primer elemento en `LinkedQueue`

- 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.

El TAD Lista

- Una **lista** es una colección que almacena elementos en un orden lineal.
- Cada elemento tiene una **posición** (o **índice**) en la lista (0 para el primer elemento, 1 para el segundo, etc.).
- **Operaciones**:
  - `append`: Inserta un elemento al final de la lista.
  - `prepend`: Inserta un elemento al principio de la lista.
  - `insert` (en una posición dada): inserta un elemento en una posición dada en la lista.
  - `delete` (en una posición determinada): elimina el elemento en una posición dada en la lista.
  - `get` (en una posición dada): Devuelve (sin eliminar) el elemento en una posición determinada en la lista.
  - `set` (en una posición dada): reemplaza el elemento en una posición dada posición en la lista.
  - `isEmpty`: Comprueba si la lista está vacía.
  - `size`: Devuelve el número de elementos en la lista.
  - `clear`: elimina todos los elementos de la lista.
  - `contains`: comprueba si la lista contiene un elemento determinado.
  - `iterator`: Devuelve un `Iterador` para recorrer los elementos de la lista.

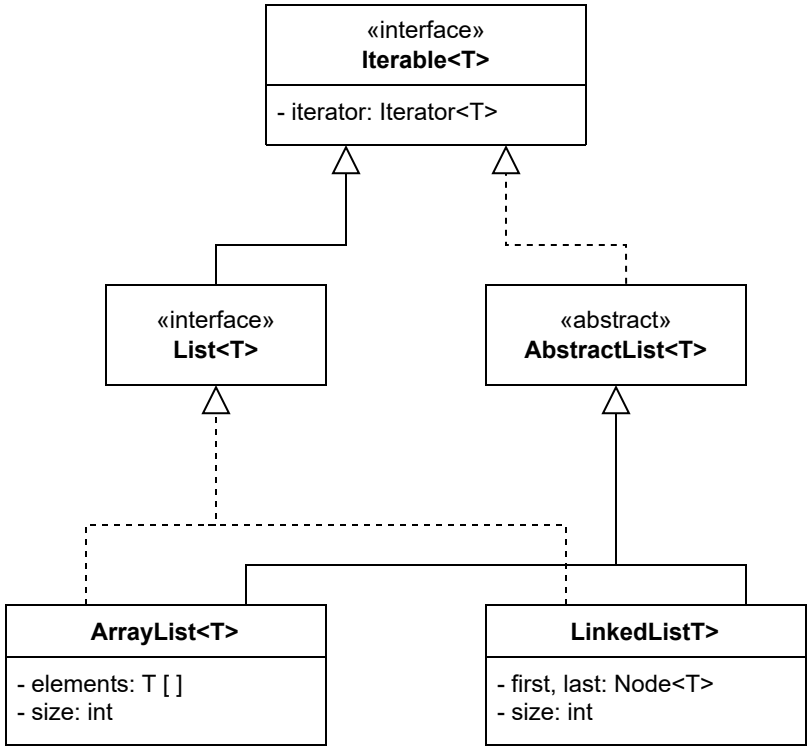
El ADT de lista en Java

- La interfaz `List<T>` define una lista con elementos de tipo `T`.

```
package dataStructures.list;

public interface List<T> extends Iterable<T> {
  void append(T element);
  void prepend(T element);
  void insert(int index, T element);
  void delete(int index);
  T get(int index);
  void set(int index, T element);
  boolean isEmpty();
  int size();
  void clear();
  boolean contains(T element);
}
```

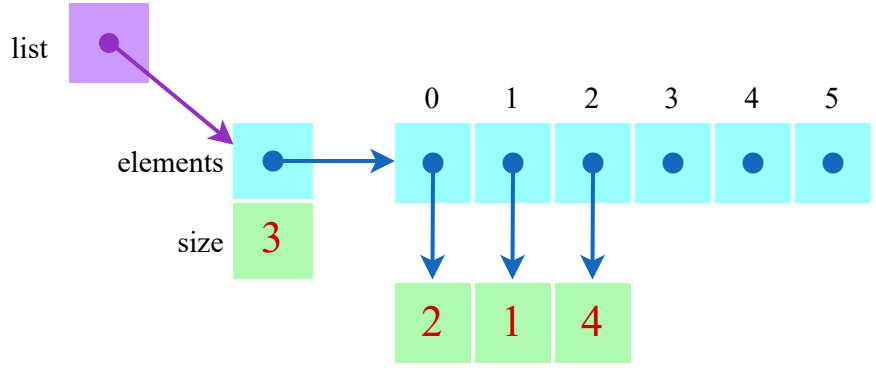
Lista de implementaciones de ADT

- Se puede implementar una lista utilizando diferentes *estructuras de datos*.
- Diferentes *clases* pueden implementar la interfaz `List<T>`:
  - `ArrayList<T>`: utiliza una matriz para almacenar elementos.
  - `LinkedList<T>`: utiliza una estructura vinculada para almacenar elementos.
- La clase abstracta base `AbstractList<T>` proporciona implementación para los métodos `equals`, `hashCode` y `toString`.



La clase `ArrayList`

- `ArrayList<T>` implementa la interfaz `List<T>` utilizando una matriz para almacenar elementos.
- Inicialmente, la matriz tiene un tamaño fijo (la *capacidad* de la lista), pero puede crecer dinámicamente cuando sea necesario.
- La posición de cada elemento en la lista corresponde directamente a su índice en la matriz, y el elemento en el índice de la lista `i` se ubica en el índice de la matriz `i`.
- Insertar nuevos elementos en la lista provoca un **desplazamiento** hacia la derecha de los elementos subsiguientes de la matriz, creando los elementos necesarios espacio.
- Por el contrario, cuando se elimina un elemento, los elementos subsiguientes se **desplazan** hacia la izquierda, eliminando cualquier espacio libre.
- La clase también mantiene una variable entera `size` para realizar un seguimiento de la cantidad de elementos en la lista.

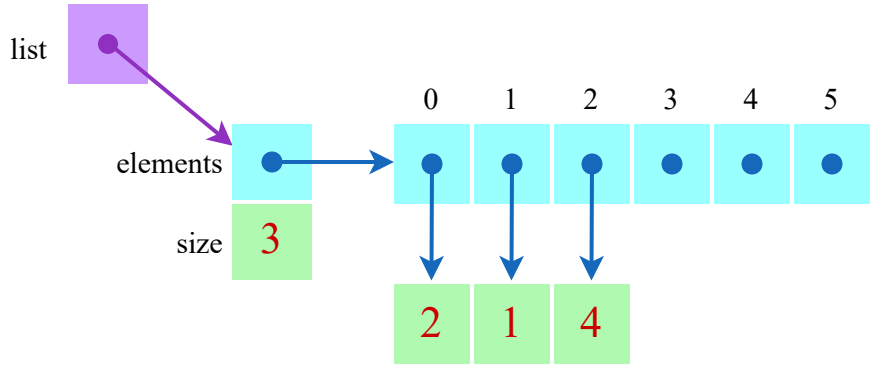


Implementación de `ArrayList`

```
package dataStructures.list;

public class ArrayList<T> extends AbstractList<T> implements List<T> {
  private static final int DEFAULT_INITIAL_CAPACITY = 6;

  private T[] elements;
  private int size;

  public ArrayList(int initialCapacity) {  // ArrayList constructor
    if (initialCapacity <= 0) {
      throw new IllegalArgumentException("initial capacity must be greater than 0");
    }
    elements = (T[]) new Object[initialCapacity];
    size = 0;
  }

  public ArrayList() {  // ArrayList constructor
    this(DEFAULT_INITIAL_CAPACITY);
  }

  ...
}
```

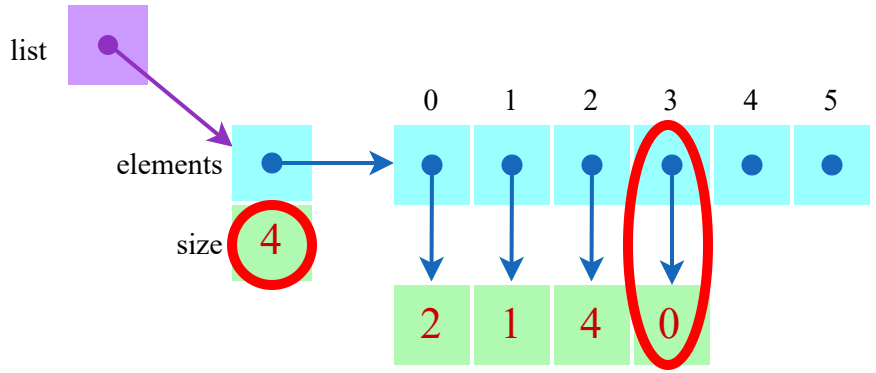
Agregar un elemento al final en `ArrayList`

- `append` agrega un elemento después del último elemento de la lista
- Se debe asegurar la capacidad del array para alojar el nuevo elemento.
- La matriz almacena el nuevo elemento en el índice designado por `size`.
- `size` se incrementa para realizar un seguimiento de la cantidad de elementos en la lista.
- A partir de esta configuración, vamos a 'añadir' el elemento 0 a la lista:



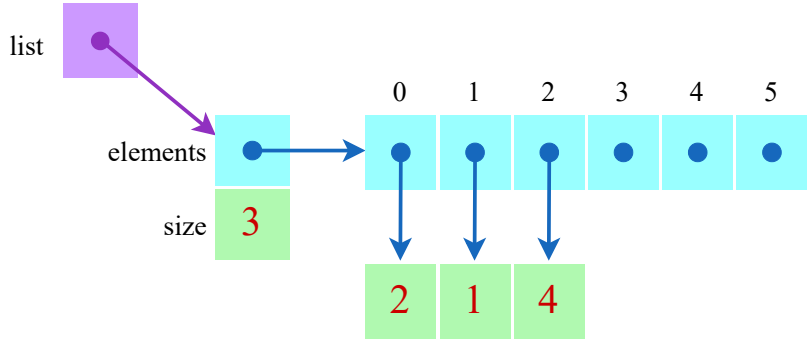
Agregar un elemento al final en `ArrayList` (II)

- Después de agregar 0:

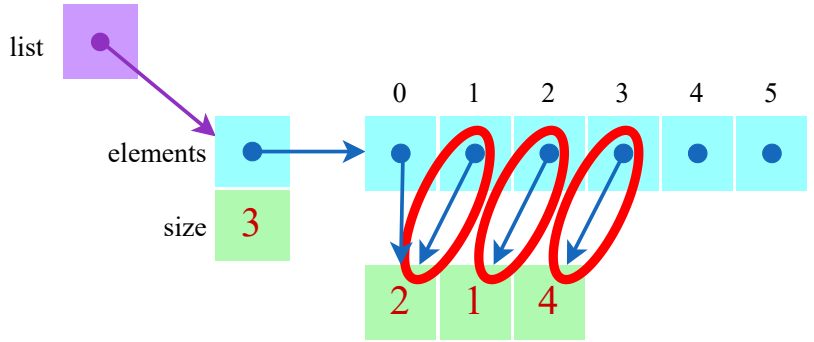


Insertar un elemento al principio en `ArrayList`

- A partir de esta configuración, vamos a anteponer el elemento 7 al inicio:

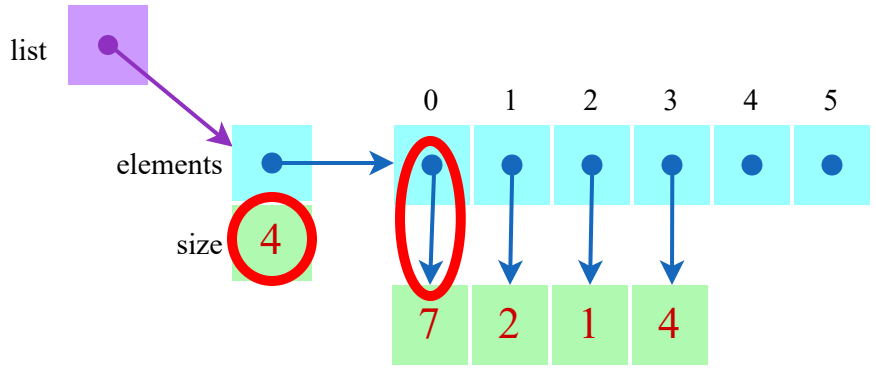


- Las referencias en las posiciones 2, 1 y 0 se desplazan hacia la derecha para hacer espacio para el nuevo elemento:



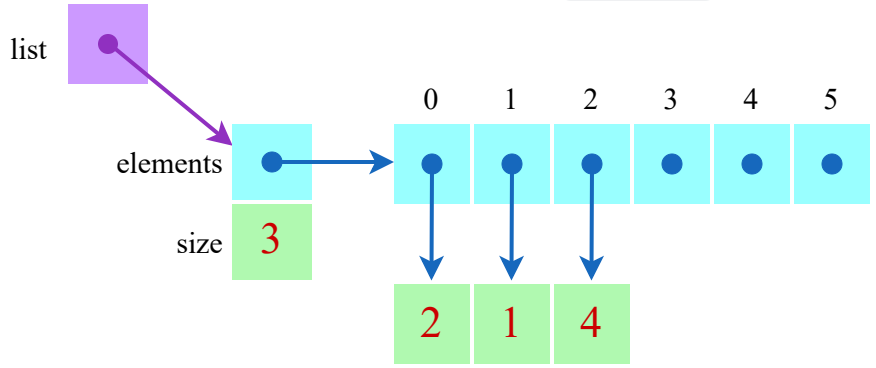
Insertar un elemento al principio en `ArrayList`

- El nuevo elemento (7) se almacena en la posición 0 y se incrementa el tamaño:

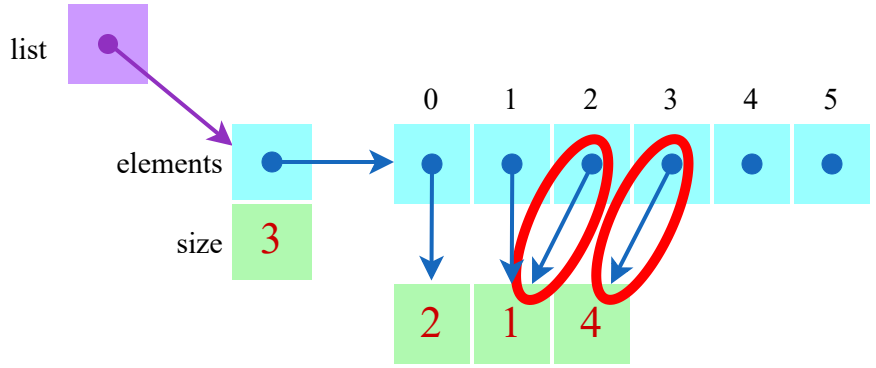


Insertar un elemento en una posición arbitraria en `ArrayList`

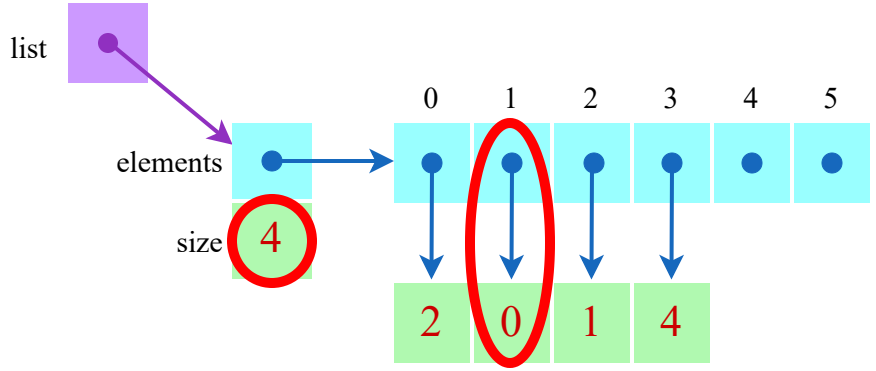
- A partir de esta configuración, vamos a `insertar` el elemento 0 en la posición 1:



- Las referencias en las posiciones 2 y 1 se desplazan *hacia la derecha* para hacer espacio para el nuevo elemento:

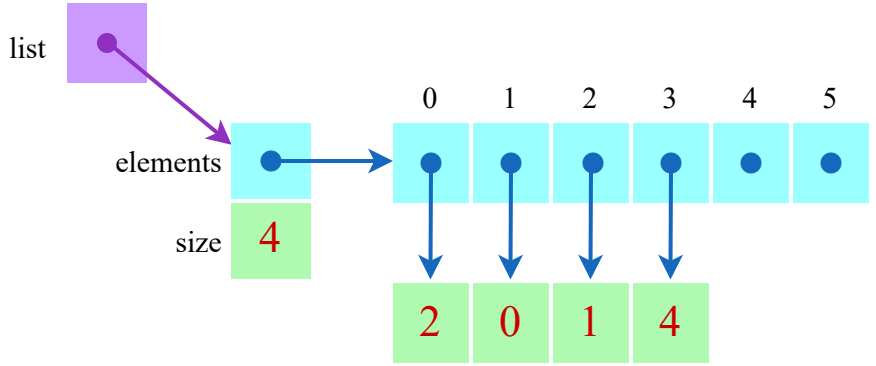


- El nuevo elemento (0) se almacena en la posición 1 y se incrementa el tamaño:

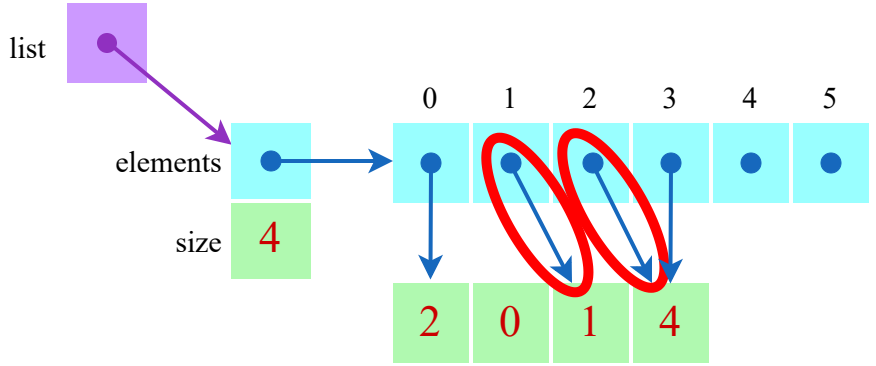


Eliminar un elemento en una posición arbitraria en `ArrayList`

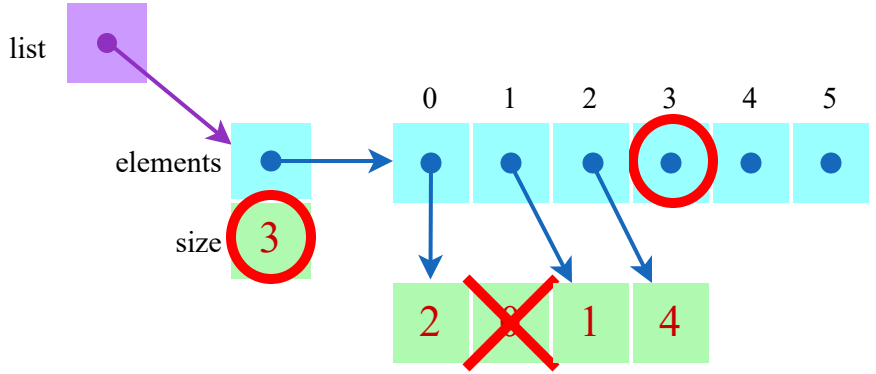
- A partir de esta configuración, vamos a `eliminar` el elemento en la posición 1:



- Las referencias en las posiciones 2 y 3 se desplazan *hacia la izquierda* para llenar el espacio dejado por el elemento eliminado:



- `size` se reduce y se establece la referencia en el índice `size` a `null`. El *recolector de basura* recuperará la memoria utilizada por el elemento eliminado:



Métodos de fábrica para `ArrayList`

Los métodos de fábrica ofrecen una forma conveniente de crear instancias de objetos `ArrayList<T>` sin invocar directamente constructores. Estos métodos incluyen:

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

```
List<Integer> list1 = ArrayList.empty(); // Create an empty list with default initial capacity

List<Integer> list2 = ArrayList.of(1, 2, 3); // Create a list containing the elements 1, 2 and 3

List<Integer> list3 = ArrayList.copyOf(list2); // Create a copy of list2 and append the element 4
list3.append(4);

// Create a list from a queue of elements and calculate their sum
List<Integer> list4 = ArrayList.from(ArrayQueue.of(5, 6, 7));
int sum = 0;
for (Integer element : list4) {
  sum += element;
}
```

La interfaz `Iterator` en Java

- La interfaz `Iterator<T>` proporciona una manera de *recorrer* los elementos de una colección.

```
package java.util;

public interface Iterator<T> {
  boolean hasNext();
  T next();
}
```

- Un `Iterador` es un objeto *con estado* que realiza un seguimiento de su posición actual en la colección.
- Tiene dos métodos:
  - `hasNext()`: Devuelve `true` si hay más elementos para atravesar.
  - `next()`: devuelve el siguiente elemento de la colección y avanza en el iterador, garantizando que la llamada posterior produzca un elemento diferente.
- Utilizando un 'Iterador':

```
List<Integer> list = ArrayList.of(1, 2, 3);

Iterator<Integer> it = list.iterator();
while (it.hasNext()) {  // this loop iterates over all elements of the list
  Integer element = it.next();
  System.out.println(element);
}
```

La interfaz «Iterable» en Java

- La interfaz `Iterable<T>` proporciona una manera de obtener un `Iterador` para recorrer los elementos de una colección.

```
package java.lang;

public interface Iterable<T> {
  Iterator<T> iterator();
}
```

- `iterator()`: Devuelve un `Iterador` que se puede utilizar para recorrer los elementos de la colección.
- Un objeto `Iterable` se puede utilizar en un bucle `for` mejorado (bucle *foreach*) para iterar sobre sus elementos:

```
List<Integer> list = ArrayList.of(1, 2, 3);
for (Integer element : list) { // this loop iterates over all elements of the list
  System.out.println(element);
}
```

Mejorar `ArrayList` con iterabilidad

- Para dotar a la clase `ArrayList` de iterabilidad, debe implementar la interfaz `Iterable` y proporcionar un método `iterator`. Este método es responsable de producir una instancia `Iterator` para recorrer los elementos de la lista.
- Se crea una *clase interna* llamada `ArrayListIterator` para implementar la interfaz `Iterator`, proporcionando la funcionalidad necesaria.
- Al invocar el método `iterator` se crea una instancia y se devuelve un nuevo objeto `ArrayListIterator`.
- El `ArrayListIterator` está diseñado para implementar el recorrido de los elementos de la lista:
  - Determina si la iteración ya ha cubierto todo elementos de la lista.
  - Identifica el índice del próximo elemento a recorrer.
  - Esto se logra manteniendo una variable `int` `current` que almacena el índice del próximo elemento que se devolverá.

Mejora de `ArrayList` con iterabilidad. Código

```
import java.util.Iterator;

public class ArrayList<T> extends AbstractList<T> implements List<T> {  // List extends Iterable
  private T[] elements;
  private int size;

  ...

  public Iterator<T> iterator() {  // implements Iterable interface method
    return new ArrayListIterator();  // return a new instance of ArrayListIterator
  }

  // Inner class to implement the Iterator interface
  private final class ArrayListIterator implements Iterator<T> {
    int current;  // index of the next element to be returned

    public ArrayListIterator() {  // ArrayListIterator constructor
      current = 0;  // start at the first element
    }

    public boolean hasNext() {
      return current < size;  // there are more elements if current is less than size
    }

    public T next() {
      if (!hasNext()) {
        throw new NoSuchElementException();  // all elements have been traversed
      }
      T element = elements[current];  // get the next element
      current++;  // advance iterator's state for subsequent invocations
      return element;  // return the element
    }
  }
}
```

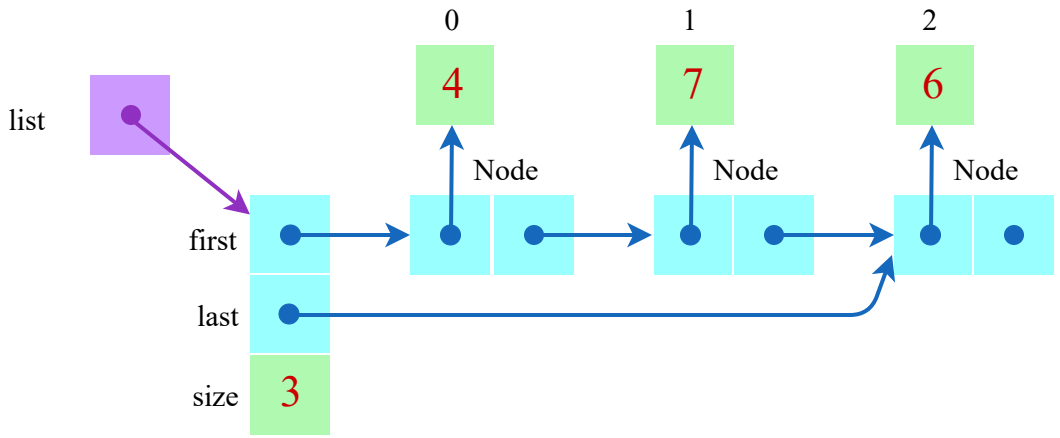
Complejidad computacional de las operaciones de `ArrayList`

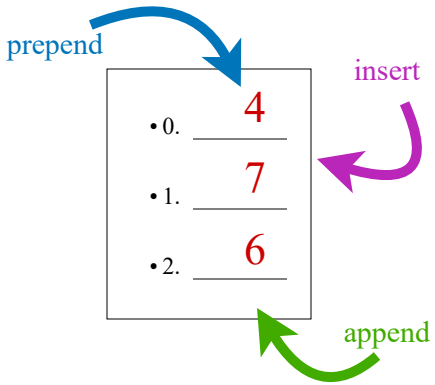
Operation	Cost
`empty`	O(1) †
`append`	O(1), O(n) §
`prepend`	O(n)
`insert`	O(n)
`delete`	O(n)
`get`	O(n)
`set`	O(n)
`isEmpty`	O(1)
`size`	O(1)
`clear`	O(n) *
`contains`	O(n)

† 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 `LinkedList`

- `LinkedList<T>` implementa la interfaz `List<T>` utilizando una estructura vinculada de nodos.
- La posición de cada elemento en la lista corresponde directamente a su posición en la estructura vinculada, y el elemento en el índice de lista `i` se ubica en el nodo en la posición `i`.
- La clase mantiene una referencia `first` al primer nodo en la estructura vinculada que corresponde al primer elemento de la lista (el que está en el índice 0).
- La clase también mantiene una referencia `last` al último nodo en la estructura vinculada que corresponde al último elemento de la lista (el que está en el índice `size - 1`).
- Cada nodo contiene un elemento y una referencia (`next`) al nodo que contiene el elemento después de él en la lista, excepto el último nodo que tiene su referencia `next` establecida en `null`.
- La clase también mantiene una variable entera `size` para realizar un seguimiento de la cantidad de elementos en la lista.



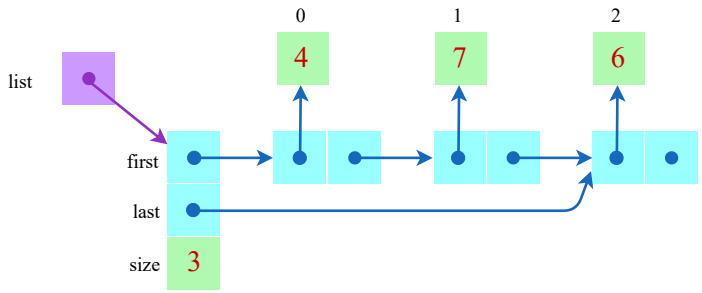


Implementación de `LinkedList`

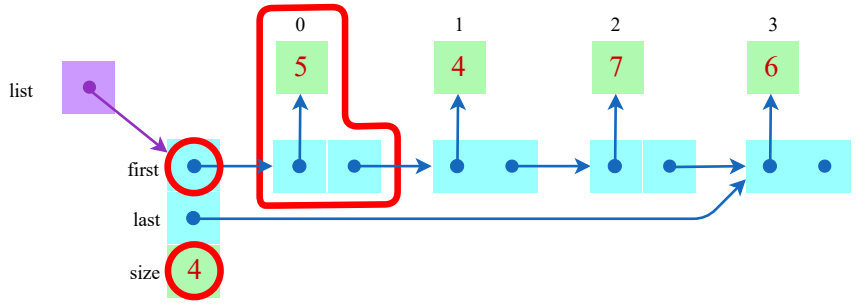
```
public class LinkedList<T> extends AbstractList<T> implements List<T> {
  private static final class Node<E> {  // Node inner class
    E element;
    Node<E> next;

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

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

  public LinkedList() {  // LinkedList constructor
    first = null;
    last = null;
    size = 0;
  }
  ...
```

Insertar un elemento al principio en `LinkedList`

- Partiendo de esta configuración, vamos a anteponer el elemento 5:



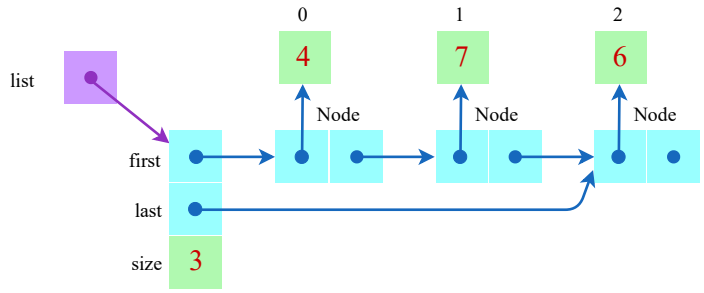
- Después de anteponer 5:



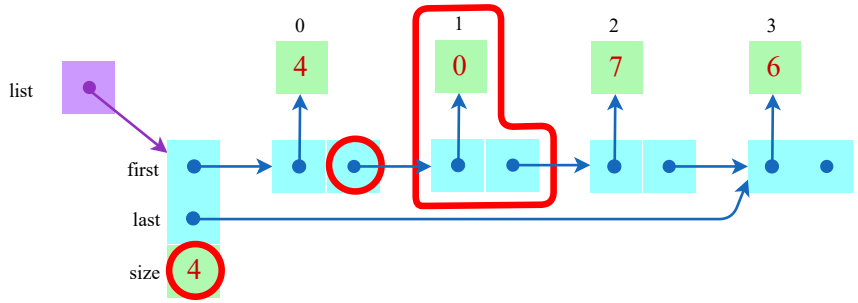
- Tener en cuenta que, si la lista estaba vacía, `last` también debe actualizarse para apuntar al nuevo `nodo`.

Insertar un elemento en una posición arbitraria en `LinkedList`

- Casos sencillos:
  - Insertar en la posición `0` es lo mismo que anteponer.
  - Insertar en la posición `tamaño` es lo mismo que añadir.
- Insertar dentro de la lista:
  - Partiendo de esta configuración, vamos a `insertar` el elemento 0 en la posición 1:

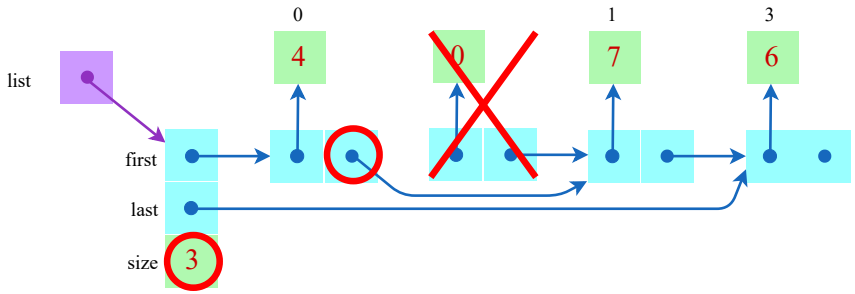


  - Después de insertar 0 en la posición 1:



Eliminar un elemento en una posición arbitraria en `LinkedList`

- Casos sencillos:
  - Eliminar en la posición `0` implica actualizar `first`. Si el la lista se vuelve vacía, `last` también se establece en `null`.
  - Eliminar en la posición `size - 1` implica actualizar `last`. Si la lista se vuelve vacía, `first` también se establece en `null`.
- Eliminar dentro de la lista en la posición `i`:
  - La referencia `siguiente` del nodo en la posición `i - 1` es actualizado con la referencia 'siguiente' del nodo que se está eliminando.
  - Se reduce el tamaño.
  - Lista después de eliminar el elemento que estaba en la posición 1:



Métodos de fábrica para `LinkedList`

Los métodos de fábrica ofrecen una forma conveniente de crear instancias de objetos `LinkedList<T>` sin invocar directamente constructores. Estos métodos incluyen:

- `empty()`: construye una *lista vacía* con una capacidad inicial *predeterminada*, adecuada para cuando se desconoce el número esperado de elementos.
- `of(T... elementos)`: construye una lista *previamente rellenada* con los elementos proporcionados, lo que permite una configuración de lista rápida y sencilla.
- `copyOf(List<T> list)`: construye una nueva lista que es una *duplicado* de la `lista` dada, preservando el orden de los elementos.
- `from(Iterable<T> iterable)`: construye una nueva lista que contiene todos los elementos del *iterable* especificado, manteniendo su orden de iteración.

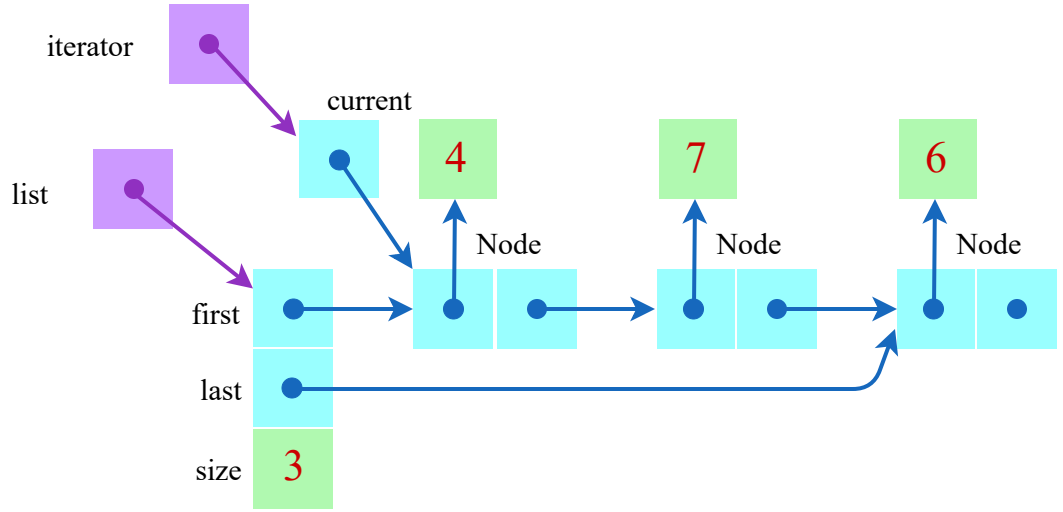
```
List<Integer> list1 = LinkedList.empty(); // Create an empty list with default initial capacity

List<Integer> list2 = LinkedList.of(1, 2, 3); // Create a list containing the elements 1, 2 and 3

List<Integer> list3 = LinkedList.copyOf(list2); // Create a copy of list2 and append the element 4
list3.append(4);

// Create a list from a queue of elements and calculate their sum
List<Integer> list4 = LinkedList.from(ArrayQueue.of(5, 6, 7));
int sum = 0;
for (Integer element : list4) {
  sum += element;
}
```

Mejorar `LinkedList` con iterabilidad

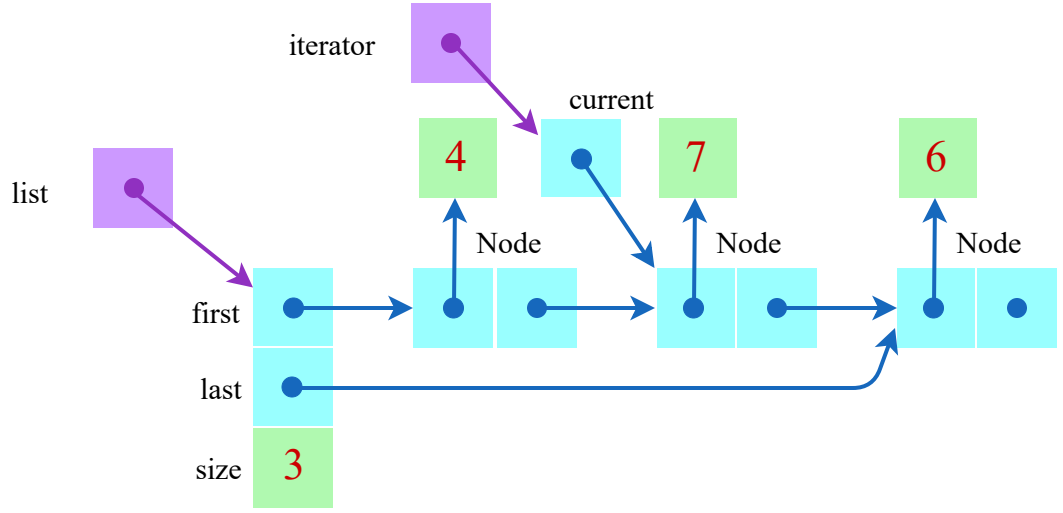
- Para dotar a la clase `LinkedList` de iterabilidad, debe implementar la interfaz `Iterable` y proporcionar un método `iterator`. Este método es responsable de producir una instancia `Iterator` para recorrer los elementos de la lista.
- Se crea una *clase interna* llamada `LinkedListIterator` para implementar la interfaz `Iterator`, proporcionando la funcionalidad necesaria.
- Al invocar el método `iterator` se crea una instancia y se devuelve un nuevo objeto `LinkedListIterator`.
- El `LinkedListIterator` está diseñado con conciencia de estado para implementar el recorrido de los elementos de la lista:
- Determina si la iteración ya ha cubierto todo
  elementos de la lista.
- Identifica el nodo del próximo elemento a recorrer.
- Esto se logra manteniendo una referencia al nodo que Será devuelto a continuación.

Mejora de `LinkedList` con iterabilidad. Ejemplo

- `LinkedListIterator` después de la inicialización:

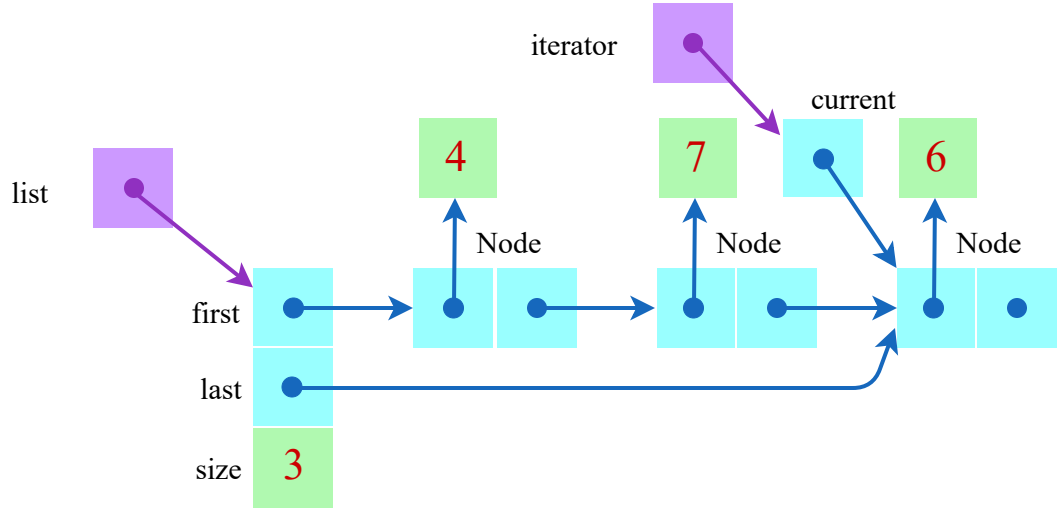


- Después de devolver el primer elemento:

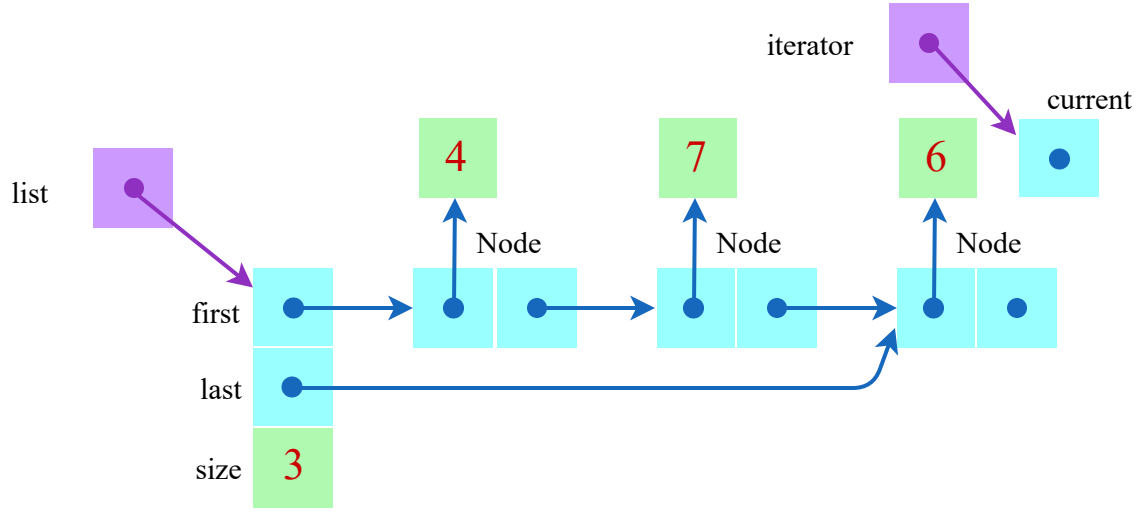


Mejora de `LinkedList` con iterabilidad. Ejemplo

- Después de devolver el segundo elemento:



- Después de devolver el tercer elemento:



Mejorar `LinkedList` con iterabilidad. Código

```
import java.util.Iterator;

public class LinkedList<T> extends AbstractList<T> implements List<T> {  // List extends Iterable
  ...
  private Node<T> first, last;
  private int size;
  ...

  public Iterator<T> iterator() {  // implements Iterable interface method
    return new ArrayListIterator();  // return a new instance of ArrayListIterator
  }

  // Inner class to implement the Iterator interface
  private final class LinkedListIterator implements Iterator<T> {
    Node<T> current;  // reference to the node with the next element to be returned

    public LinkedListIterator() {  // ArrayListIterator constructor
      current = ???;  // start at the first element
    }

    public boolean hasNext() {
      return ???;  // check if there are more elements
    }

    public T next() {
      if (!hasNext()) {
        throw new NoSuchElementException();  // all elements have been traversed
      }
      T element = ???;  // get the next element
      ???;  // advance iterator's state for subsequent invocations
      return element;  // return the element
    }
  }
}
```

Complejidad computacional de las operaciones de `LinkedList`

Operation	Cost
`empty`	O(1)
`append`	O(1)
`prepend`	O(1)
`insert`	O(n)
`delete`	O(n)
`get`	O(n)
`set`	O(n)
`isEmpty`	O(1)
`size`	O(1)
`clear`	O(1)
`contains`	O(n)

Comparación experimental entre «ArrayList» y «LinkedList»

- Medimos el tiempo de ejecución al realizar 100000 operaciones aleatorias (`insertar`, `eliminar` o `obtener`) en una lista inicialmente vacía.
- Usando una CPU Intel i7 860 y JDK 22:
  - `ArrayList` fue aproximadamente 4,30 veces más rápido que `LinkedList`.