Grado en Ingeniería Informática
Metodología de la Programación

UAL. Área de Lenguajes y Sistemas Informáticos. Mercedes Martínez Durbán

Tema 3. Listas

- Introducción
- Características comunes de las listas
- Implementación de listas con arrays
- Implementación de listas con nodos
- Variaciones de listas enlazadas

Introducción

Este capítulo y los siguientes se centrarán en la implementación de *estructuras de datos*.

Las listas, pilas, colas y colas de prioridad son estructuras de datos típicas que deben estudiarse en un curso de estructuras de datos. Estas estructuras están soportadas en la API de Java y su uso se presentará también.

Características comunes de las listas

Las características comunes de las listas se definen en la interface *Lista.*

Una lista es una estructura de datos muy común para *almacenar datos en orden secuencial.* Por ejemplo, una lista de estudiantes, una lista de habitaciones disponibles, de ciudades, de libros… Una lista realiza las siguientes operaciones:

- Recuperar un elemento de la lista.
- Insertar un nuevo elemento en la lista.
- Borrar un elemento de la lista.
- Averiguar cuántos elementos tiene la lista.
- Determinar cuándo un elemento está en la lista
- Comprobar cuándo una lista está vacía.

Hay dos maneras de implementar la lista.

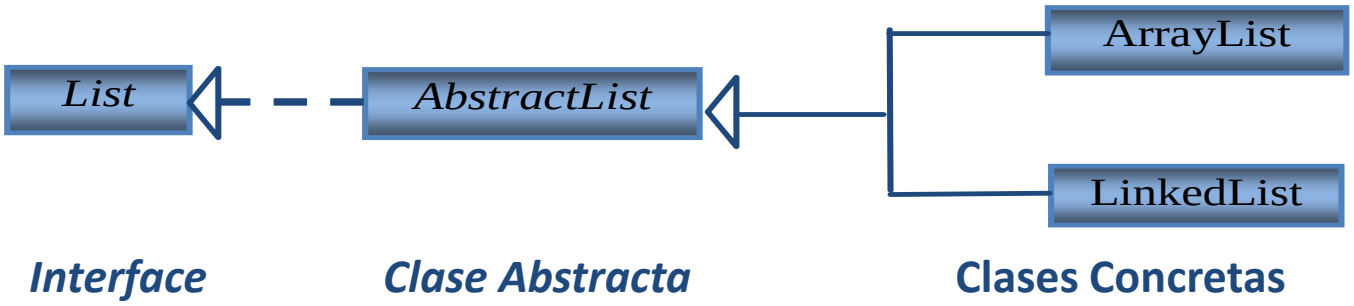
- Una, es utilizando ***arrays*** para almacenar los elementos. El array tiene un tamaño fijo. Si la capacidad del array se excede, necesitaríamos crear uno nuevo, más grande y copiar todos los elementos del array en el nuevo array.

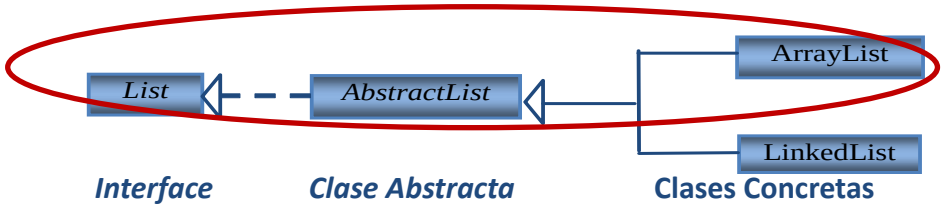
- Otra, es usar una ***estructura enlazada***. Una estructura enlazada consiste en ***nodos***. Cada nodo se crea dinámicamente para contener un elemento. Todos los nodos se enlazan para formar una lista.

Podemos definir dos clases para las listas: **ArrayList** y **LinkedList**. Las dos clases tienen las operaciones comunes de las listas pero las implementaciones son diferentes.

Diseño

Para el **diseño**, podemos pensar en utilizar una interface o una clase abstracta para las operaciones comunes. Una buena estrategia es combinar las virtudes de las interfaces y de las clases abstractas proporcionando ambas. La clase abstracta proporciona un esqueleto de implementación de la interface, lo cual minimiza el esfuerzo requerido para la implementación de la interface. Este diseño, es el llevado a cabo en la API. Dejamos el nombre **List** para la interface y **AbstractList** para la clase abstracta. Las relaciones entre **List, AbstracList, ArrayList** y **LinkedList** se muestran en la diapositiva que sigue.





«interface» java.lang.Iterable<E>

```
+iterator(): Iterator<E>
```

Devuelve un iterador para los elementos de la lista.

«interface» List<E>

```
+add(e: E): void
+add(index: int, e: E): void
+clear(): void
+contains(e: E): boolean
+get(index: int): E
+indexOf(e: E): int
+isEmpty(): boolean
+lastIndexOf(e: E): int
+remove(e: E): boolean
+size(): int
+remove(index: int): E
+set(index: int, e: E): E
```

Añade un elemento al final de la lista.
Inserta un nuevo elemento que está en la posición index en la lista.
Borra todos los elementos de la lista.
Devuelve verdadero si la lista contiene el elemento especificado e.
Devuelve el elemento de la lista que está en la posición index especificada.
Devuelve el índice del primer elemento de la lista que coincide con e.
Devuelve verdadero si la lista no contiene elementos.
Devuelve el índice del último elemento de la lista que coincide con e.
Elimina el elemento e de la lista.
Devuelve el número de elementos de la lista.
Elimina el elemento en la posición especificada por index y devuelve el elemento borrado.
Modifica el elemento de la posición especificada por index por e y devuelve el elemento sustituido.

AbstractList<E>

```
#size: int

#  AbstractList()
#  AbstractList(objects: E[])
+add(e: E): void
+isEmpty(): boolean
+size(): int
+remove(e: E): boolean
```

Tamaño de la lista.

Crea una lista por defecto
Crea una lista a partir de un array de objetos.
Implementa el método add.
Implementa el método isEmpty.
Implementa el método size.
Implementa el método remove.

```java
package org.mp.tema03;

public interface List<E> extends Iterable<E>{

  /** Añade un nuevo elemento al final de la lista */
  public void add(E e);

  /** Añade un nuevo elemento en la posición especificiada por index en la lista */
  public void add(int index, E e);

  /** Borra todos los elementos de la lista */
  public void clear();

  /** Devuelve true si la lista contiene el elemento e*/
  public boolean contains(E e);

  /** Devuelve el elemento de la lista que está en la posición especificada por index */
  public E get(int index);

  /** Devuelve el índice de la primera ocurrencia del elemento e en la lista.
   *  Devuelve -1 si no está. */
  public int indexOf(E e);

  /** Devuelve true si la lista no tiene elementos */
  public boolean isEmpty();

  /** Devuelve el índice de la última ocurrencia del elemento e en la lista.
   *  Devuelve -1 si no está. */
  public int lastIndexOf(E e);

  /** Borra la primera ocurrencia del elemento e en la lista.
   *  Desplaza la subsecuencia de elementos a la izquierda.
   *  Devuelve true si el elemento se ha borrado. */
  public boolean remove(E e);

  /** Borra el elemento de la posición especificada por index de la lista.
   *  Desplaza la subsecuencia de elementos a la izquierda.
   *  Devuelve el elemento que ha sido borrado de la lista. */
  public E remove(int index);

  /** Sustituye el elemento de la posición especificada por index en la lista
   *  por el elemento e y devuelve el elemento antiguo */
  public Object set(int index, E e);

  /** Devuelve el número de elementos de la lista */
  public int size();
}
```

INTERFACE LIST

```java
package org.mp.tema03;

public abstract class AbstractList<E> implements List<E> {

  protected int size = 0; // Tamaño de la lista

  /** Crea una lista por defecto */
  protected AbstractList() {
  }

  /** Crea una lista a partir de un array de objetos */
  protected AbstractList(E[] objects) {
    for (int i = 0; i < objects.length; i++)
      add(objects[i]);
  }

  /** Añade un nuevo elemento al final de la lista */
  public void add(E e) {
    add(size, e);
  }

  /** Devuelve true si la lista no contiene ningún elemento*/
  public boolean isEmpty() {
    return size == 0;
  }

  /** Devuelve el número de elementos de la lista */
  public int size() {
    return size;
  }

  /** Elimina la primera ocurrencia del elemento e de la lista.
   *  Desplaza la subsecuencia de elementos a la izquierda.
   *  Shift any subsequent elements to the left.
   *  Devuelve true si el elemento se eliminó. */
  public boolean remove(E e) {
    if (indexOf(e) >= 0) {
      remove(indexOf(e));
      return true;
    }
    else
      return false;
  }
}
```

Diseño

Es poco habitual el uso del modificador protected en las propiedades. Sin embargo , en este caso es una buena elección. Las subclases de AbstractList pueden acceder a la propiedad size, pero las que no sean subclases de AbstractList no pueden hacerlo. Como regla general, declararemos propiedades protected en clases abstractas.

Importante

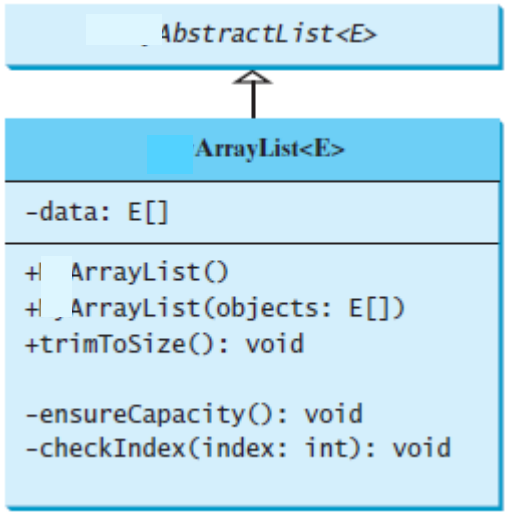
Faltaría la implementación del resto de los métodos de las interfaces List e Iterable.

CLASE ABSTRACTA ABSTRACTLIST

Implementación de una lista con arrays

Un ArrayList es la implementación de una lista utilizando arrays.

Un array es una estructura de datos de tamaño fijo. Cuando se crea un array, su tamaño no puede ser cambiado. A pesar de ello, podemos utilizar un array para implementar estructuras de datos dinámicas. El truco está en crear un nuevo array más grande para sustituir el array que tenemos que no nos permitiría aumentar en un nuevo elemento la lista.

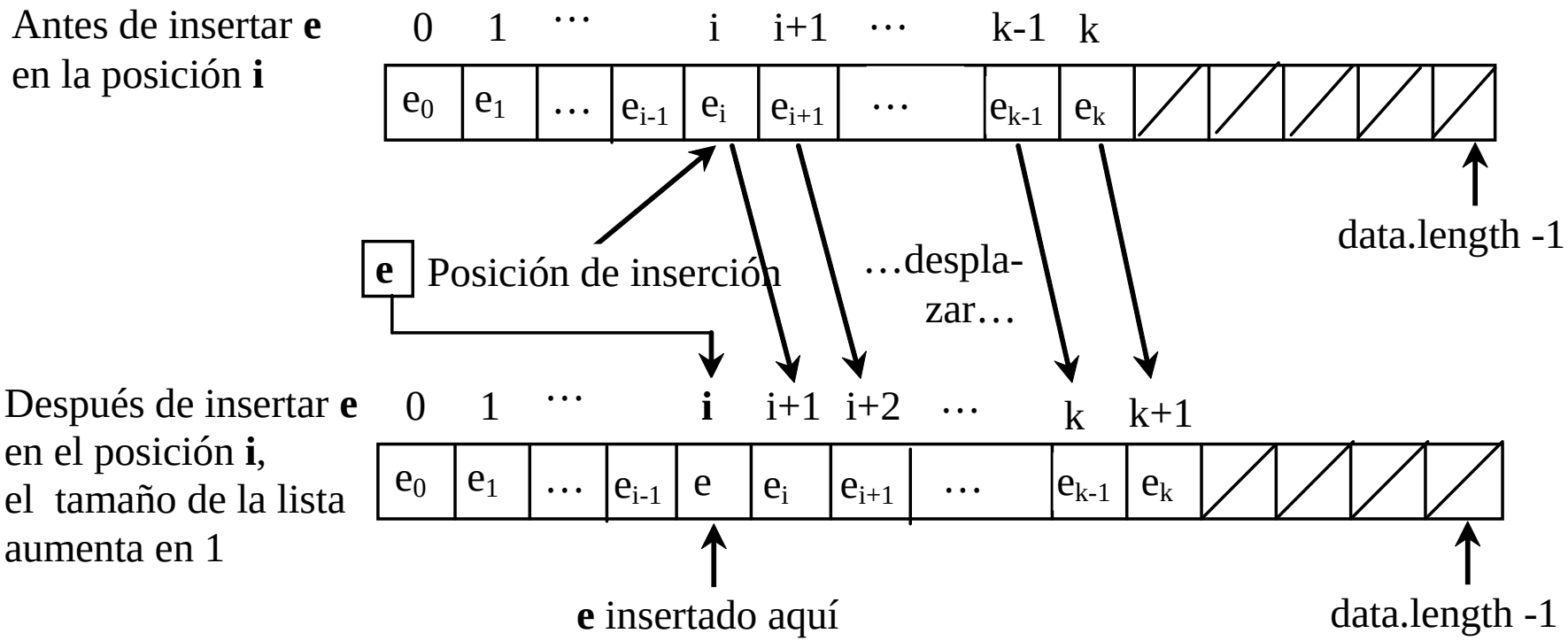


Crea un array list por defecto.
Crea un array list a partir de un array de objetos.
Ajusta la capacidad del array list al tamaño de la lista.
Dobla el tamaño del array si se necesita.
Lanza una excepción si el indice está fuera de los límites de la lista.

Inserción de un elemento en un array

Inicialmente, un array, **data** de tipo **E[]** se crea con un tamaño por defecto. Cuando queramos insertar un nuevo elemento en el array:

1. Tendremos que asegurarnos que hay suficiente espacio en el array.
2. De no ser así, creamos un nuevo array de tamaño doble del que tenemos y copiamos los elementos del array antiguo en el array nuevo.

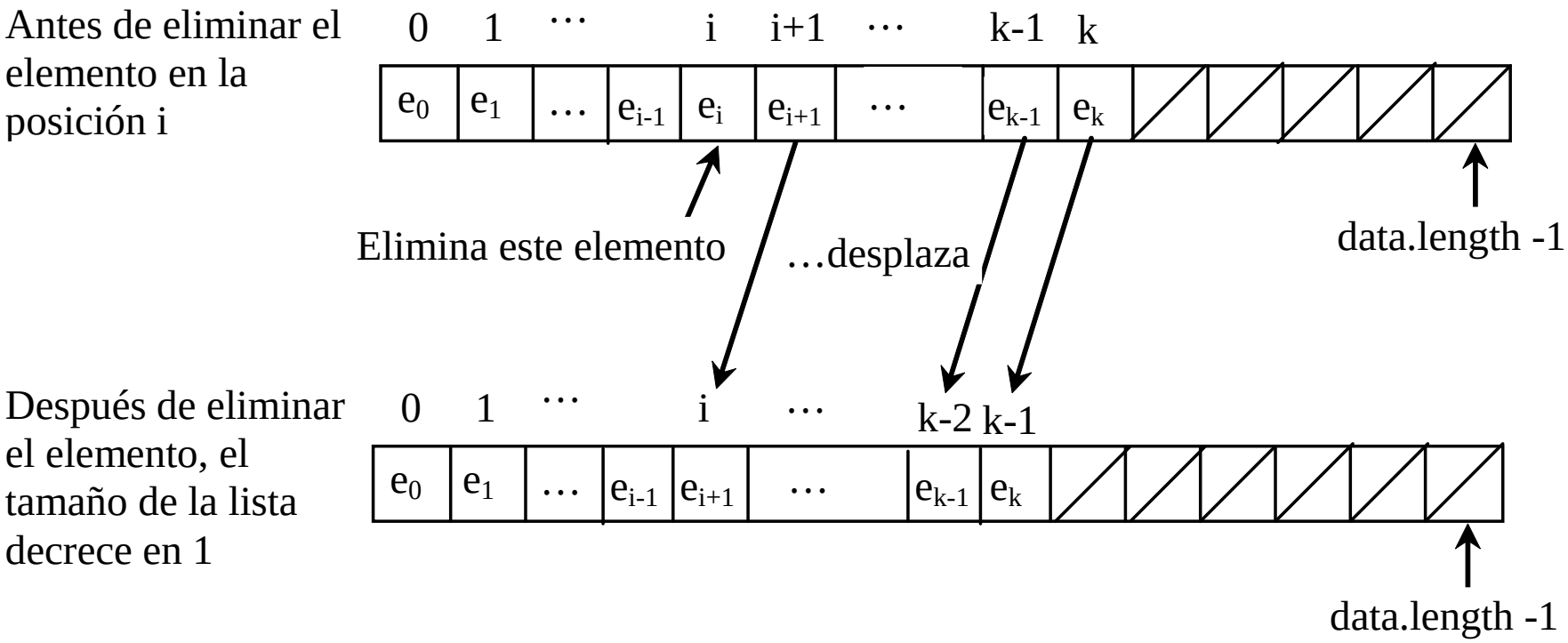


Método add

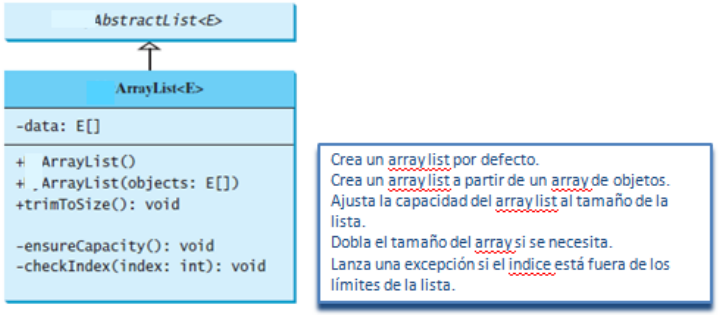
```
@Override/** Añade un nuevo elemento e en la posición especificada por index */
  public void add(int index, E e) {
    ensureCapacity(); // Verifica cuando el array está lleno. De ser así,
                      // crea un nuevo array con el doble tamaño más 1 y copia
                 // el array en el nuevo utilizando el método System.arraycopy y
                      // pone el nuevo array como el array a utilizar
   // Mueve los elementos a la derecha desde la posición especificada por index
    for (int i = size - 1; i >= index; i--)
      data[i + 1] = data[i];

    // Inserta un nuevo elemento en data[index]
    data[index] = e;

    // Incrementa el tamaño en 1
    size++;
  }
```

Eliminar un elemento de un array

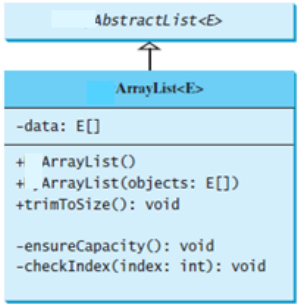
Para eliminar un elemento en una posición específica index, desplazamos todos los elementos después de la posición index a la izquierda una posición y decrementamos el tamaño de la lista en uno. Se puede ver en la figura.



La clase ArrayList

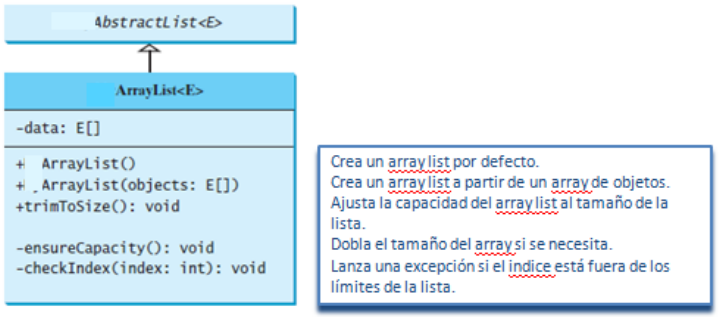



```
public class ArrayList<E> extends AbstractList<E> {

  private static final int CAPACIDAD_INICIAL = 16;
  private E[] data = (E[])new Object[CAPACIDAD_INICIAL]; //Crea un array

  /** Crea una lista por defecto */
  public ArrayList() {
  }

  /** Crea una lista a partir de un array de objetos */
  public ArrayList(E[] objects) {
    for (int i = 0; i < objects.length; i++)
      add(objects[i]); // Advertencia: no usar super(objects)!
  }

  @Override /** Añade un nuevo elemento en la posición especifica por index */
  public void add(int index, E e) {
          // HECHO
```

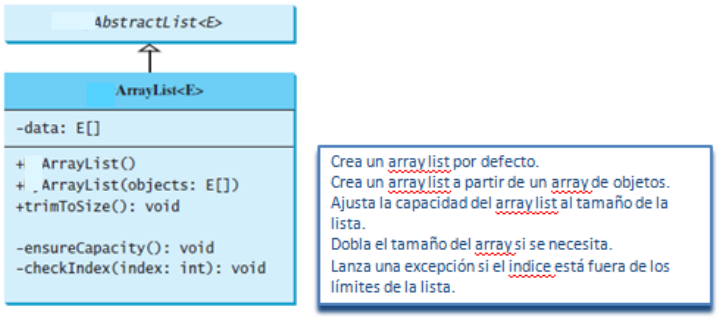


Crea un array list por defecto.
Crea un array list a partir de un array de objetos.
Ajusta la capacidad del array list al tamaño de la lista.
Dobla el tamaño del array si se necesita.
Lanza una excepción si el índice está fuera de los límites de la lista.

```
private void ensureCapacity() {
    // POR HACER
    // Verifica cuando el array está lleno. De ser así,
    // crea un nuevo array con el doble tamaño más 1 y copia
    // el array en el nuevo utilizando el método System.
// arraycopy y pone el nuevo array como el array a utilizar
}

@Override /** Elimina todos los elementos de la lista */
 public void clear() {
    // POR HACER
    // Crea un nuevo array de tamaño CAPACIDAD_INICIAL y resetea la variable size a 0
}

@Override /** Devuelve true si la lista contiene el elemento*/
 public boolean contains(E e) {
    // POR HACER
   // Comprueba si e está en el array comparando cada elemento del array con e usando para
   // ello equals
}

@Override /** Devuelve el elemento en la posición index especificada */
 public E get(int index) {
    checkIndex(index);
    return data[index];
  }
 private void checkIndex(int index) {
    if (index < 0 || index >= size)              // Comprueba si index está en el rango
      throw new IndexOutOfBoundsException        // Si no, el método lanza esa Excepción
        ("Indice: " + index + ", Tamaño: " + size);
  }
```




```
 @Override /** Devuelve el índice de la primera
   * ocurrencia del elemento en la lista.
   * Devuelve -1 si no está. */
  public int indexOf(E e) {
    // POR HACER
}

@Override /** Devuelve el índice de la última ocurrencia del elemento
  * en la lista. Devuelve -1 si no está. */
  public int lastIndexOf(E e) {
    // POR HACER
}

@Override /** Elimina el elemento en la posición especificada
    *  en la lista. Desplaza la subsecuencia de elementos a la izquierda.
    *  Devuelve el elemento eliminado. */
  public E remove(int index) {
    // POR HACER

   }
@Override /** Sustituye el elemento de la posición especificada
   *  en la lista por el elemento especificado. */
  public E set(int index, E e) {
     checkIndex(index);
     E antiguo = data[index];
     data[index] = e;
     return antiguo;
   }
```




```
@Override
 public String toString() {
    StringBuilder resultado = new StringBuilder("[");
    for (int i = 0; i < size; i++) {
      resultado.append(data[i]);
      if (i < size - 1) resultado.append(", ");
    }
    return resultado.toString() + "]";
  }

  /** Ajusta la capacidad del array al tamaño de la lista */
 public void trimToSize() {
    // POR HACER
    // si el tamaño del array es distinto de la capacidad
    // Crea un nuevo array del tamaño del que se tiene
    // Copia el array en el nuevo array utilizando System.arraycopy
    // Si tamaño == capacidad, no es necesario hacer nada
}
@Override /** Sobre-escribe el método iterator() definido en Iterable */
 public java.util.Iterator<E> iterator() {
    // POR HACER
    // Devuelve una instancia de java.util.Iterator, una instancia de ArrayListIterator

}
 private class ArrayListIterator implements java.util.Iterator<E>{
    // POR HACER
    // Esta clase implementa la interface Iterator
    // Implementa por tanto los métodos hasNext, next y remove
    // Usa current para indicar la posición del elemento que está siendo atravesado
}
```

Ejemplo de uso

```
1  package org.mp.tema03;
2  public class TestrrayList {
3    public static void main(String[] args) {
4          // Creaa una lista
5      List<String> lista = new ArrayList<String>();
6
7      System.out.println();
8          // Añade elementos a la lista
9      lista.add("España"); // Añade a la lista
10     System.out.println("(1) " + lista);
11
12     lista.add(0, "Canada"); // Añade al principio de la lista
13         System.out.println("(2) " + lista);
14
15     lista.add("Rusia"); // Añade al final de la lista
16     System.out.println("(3) " + lista);
17
18     lista.add("Francia"); // Añade al final de la lista
19     System.out.println("(4) " + lista);
20
21     lista.add(2, "alemania"); // Añade en la posición 2
22     System.out.println("(5) " + lista);
23
24     lista.add(5, "Noruega"); // Añade en la posición 5
25     System.out.println("(6) " + lista);
26
27     // Elimina elementos de la lista
28     lista.remove("Canada"); // Igual que lista.remove(0)
29     System.out.println("(7) " + lista);
30
31     lista.remove(2); // Elimina el elemento en la posición 2
32     System.out.println("(8) " + lista);
33
34     lista.remove(lista.size() - 1); // elimina el último elemento
35     System.out.print("(9) " + lista + "\n(10) ");
36
37     for (String s: lista)
38       System.out.print(s.toUpperCase() + " ");
39   }
40 }
```

Salida

```
(1) [España]
(2) [Canada, España]
(3) [Canada, España, Rusia]
(4) [Canada, España, Rusia, Francia]
(5) [Canada, España, alemania, Rusia, Francia]
(6) [Canada, España, alemania, Rusia, Francia, Noruega]
(7) [España, alemania, Rusia, Francia, Noruega]
(8) [España, alemania, Francia, Noruega]
(9) [España, alemania, Francia]
(10) ESPAÑA ALEMANIA FRANCIA
```

```
//otra forma
   Iterator<String>  iterador = lista.iterator();
   while (iterador.hasNext()){
      System.out.print((iterador.next()).toUpperCase()+" ");
   }
```

Usando iterator

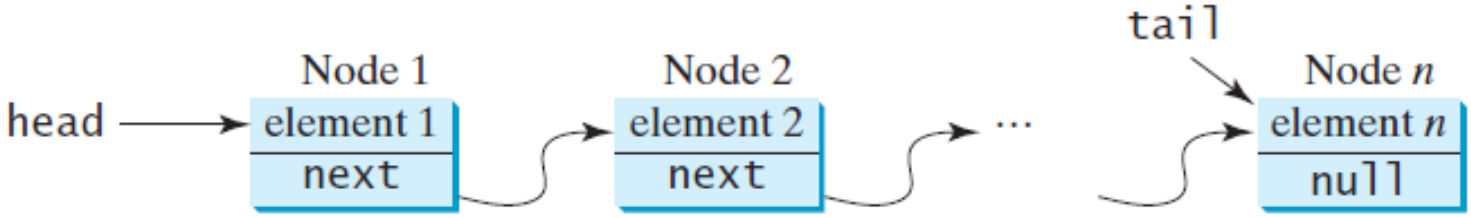
Implementación de una lista con nodos. Listas enlazadas

Una lista enlazada es la implementación usando una estructura enlazada.

Las implementaciones hechas en ArrayList de los métodos **get(int index)** y **set(int index, E e)** para acceder y modificar un elemento a partir de un índice y el método **add(E e)** para añadir un elemento al final de la lista, son **eficientes.** Sin embargo, los métodos **add(int index, E e)** y **remove(int index)** son **ineficientes** porque requieren el desplazamiento de un amplio número de elementos. Usaremos una estructura enlazada para implementar una lista para mejorar la eficiencia de los métodos anteriores.

Nodos

En una lista enlazada, cada elemento contiene un objeto llamado *nodo.* Cuando un nuevo elemento se añade a la lista, se debe crear un nodo que lo contenga. Cada nodo se enlaza con el siguiente vecino tal y como se muestra en la figura.



Lista enlazada consistente en un número cualquiera de nodos enlazados.

Un nodo se puede crear a partir de una clase definida como sigue:

```
public class Node<E> {
  E element;
  Node next;

  public Node(E o) {
    element = o;
  }
}
```

Se usa la variable **head** para referirnos al primer nodo de la lista y la variable **tail** para el último nodo. Si la lista está vacía, ambos **head** y **tail** son **null.** Veamos un ejemplo que crea una lista enlazada que contiene tres nodos. Cada nodo almacena un elemento string.

Paso 1. Declara **head** y **tail**.

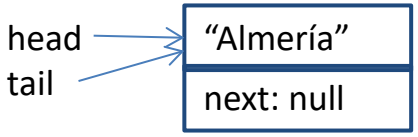
```
Node<String> head = null;
Node<String> tail = null;
```

La lista está vacía

Paso 2. Crea el primer nodo y lo añade a la lista tal y como se muestra. Después de insertar el primer nodo en la lista, **head** y **tail** apuntan a este nodo.

```
head = new Node<> (“Almería”);
tail = head;
```

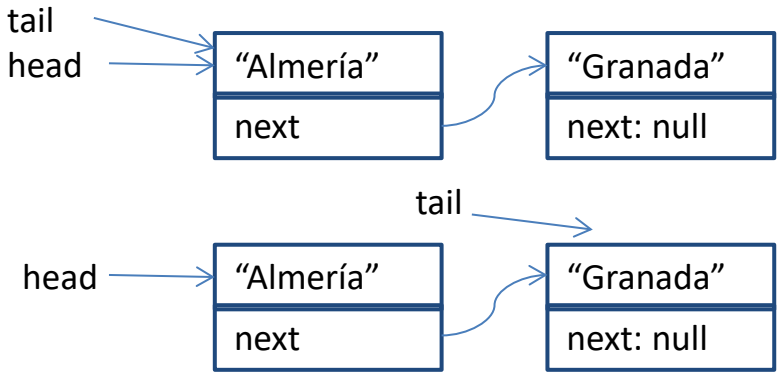
Después, se inserta el primer nodo



Añade el primer nodo a la lista. head y tail apuntan a ese nodo.

Paso 3. Crea un segundo nodo y lo añade a la lista tal y como se muestra. Para añadir el segundo nodo, se enlaza el primer nodo con el nuevo nodo. El nuevo nodo es ahora el nodo tail por lo que se debe mover tail para apuntar a este nuevo nodo tal y como se muestra.

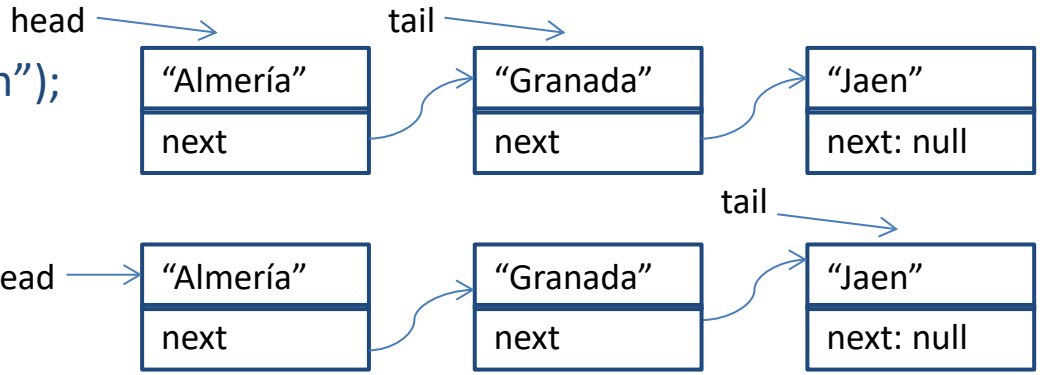
```
tail.next = new Node<> (“Granada”);
tail = tail.next;
```



Añade el segundo nodo a la lista. tail ahora apunta a ese nuevo nodo.

Paso 4. Crea el tercer nodo y lo añade a la lista tal y como se muestra. Para añadir el tercer nodo, se enlaza el último nodo con el nuevo nodo. El nuevo nodo es ahora el nodo tail por lo que se debe mover tail para apuntar a este nuevo nodo tal y como se muestra.

tail.next = new Node<> (“Jaen”);



tail = tail.next;

Añade el tercer nodo a la lista.

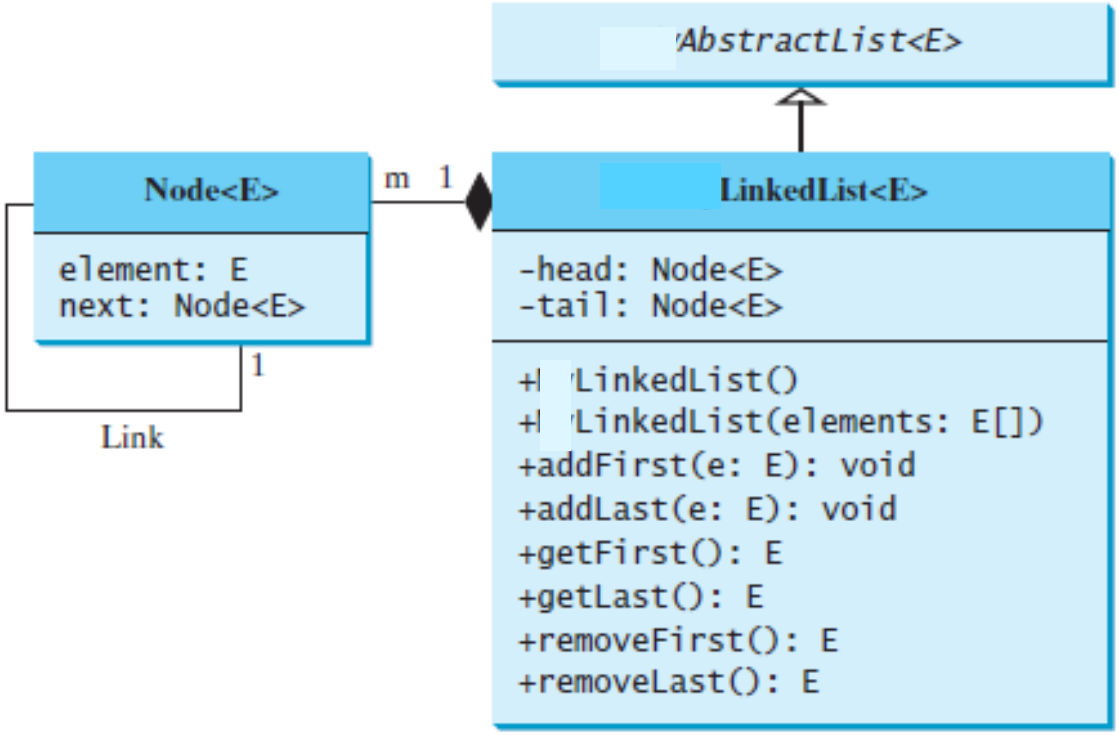
Cada nodo contiene el elemento y la propiedad llamada **next** que apunta al siguiente elemento. Si el nodo es el último en la lista, su propiedad **next** apuntará a **null**. Usaremos esta característica para detectar el último nodo. Por ejemplo, escribiremos el siguiente bucle para recorrer todos los nodos de una lista.

```
Node<E> current = head;
while (current != null) {
  System.out.println(current.element);
  current = current.next;
}
```

La variable **current** apunta inicialmente al primer nodo en la lista. En el bucle, el elemento del nodo current se recupera y el puntero **current** apunta al siguiente nodo. El bucle continua hasta que el nodo current es nulo.

La clase LinkedList

La clase **LinkedList** usa una estructura enlazada para implementar una lista dinámica. Hereda de **AbstractList**. Además, proporciona los métodos **addFirst, addLast, removeFirst, removeLast, getFirst** y **getLast**.



Crea una lista enlazada por defecto.
Crea una lista enlazada a partir de un array de elementos.
Añade un elemento a la cabeza de la lista.
Añade un elemento a la cola de la lista.
Devuelve el primer elemento de la lista.
Devuelve el último elemento de la lista.
Elimina el primer elemento de la lista.
Elimina el último elemento de la lista.

Ejemplo de uso

```
1  package org.mp.tema03;
2
3  public class TestLinkedList {
4      /** Main method */
5      public static void main(String[] args) {
6          // Crea una lista de string
7          LinkedList<String> lista = new LinkedList<String>();
8
9          System.out.println();
10         // Añade elementos a la lista
11         lista.add("España"); // Añade a la lista
12         System.out.println("(1) " + lista);
13
14         lista.add(0, "Canada"); // Añade al principio de la lista
15         System.out.println("(2) " + lista);
16
17         lista.add("Rusia"); // Añade al final de la lista
18         System.out.println("(3) " + lista);
19
20         lista.addLast("Francia"); // Añade al final de la lista
21         System.out.println("(4) " + lista);
22
23         lista.add(2, "Alemania"); // Añade en la posición 2 de la lista
24         System.out.println("(5) " + lista);
25
26         lista.add(5, "Noruega"); // Añade en la posición 5 de la lista
27         System.out.println("(6) " + lista);
28
29         lista.add(0, "Polonia"); // Igual que lista.addFirst("Polonia")
30         System.out.println("(7) " + lista);
31
32     // Elimina elementos de la lista
33     lista.remove(0); // Igual que lista.remove("Polonia")
34     System.out.println("(8) " + lista);
35
36     lista.remove(2); // Elimina el elemento en la posición 2
37     System.out.println("(9) " + lista);
38
39     lista.remove(lista.size() - 1); // Elimina el último elemento
40     System.out.print("(10) " + lista + "\n(11) ");
41
42     for (String s: lista)
43         System.out.print(s.toUpperCase() + " ");
44
45         lista.clear();
46         System.out.println("\nDespués de limpiar la lista, el tamaño es  "
47           + lista.size());
48     }
49 }
```

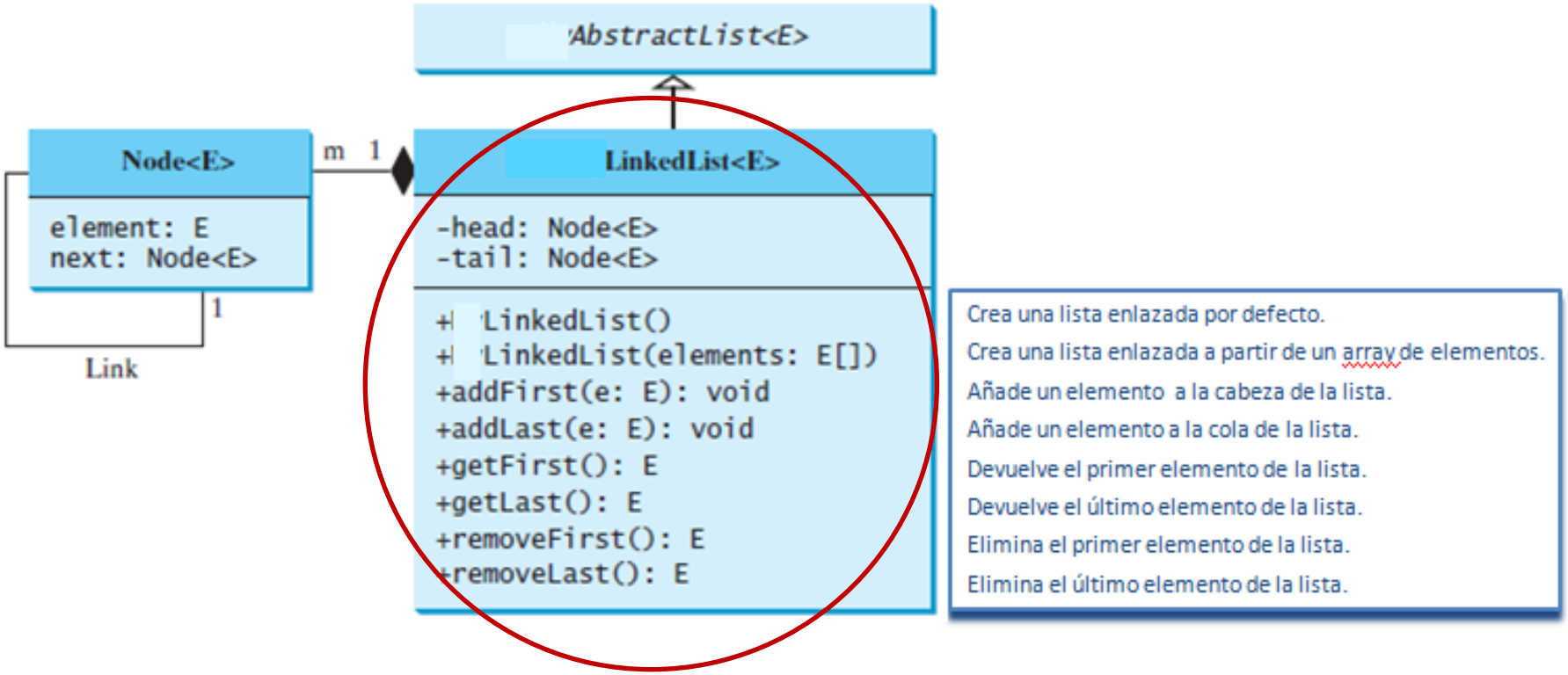
Salida

```
(1) [España]
(2) [Canada, España]
(3) [Canada, España, Rusia]
(4) [Canada, España, Rusia, Francia]
(5) [Canada, España, Alemania, Rusia, Francia]
(6) [Canada, España, Alemania, Rusia, Francia, Noruega]
(7) [Polonia, Canada, España, Alemania, Rusia, Francia, Noruega]
(8) [Canada, España, Alemania, Rusia, Francia, Noruega]
(9) [Canada, España, Rusia, Francia, Noruega]
(10) [Canada, España, Rusia, Francia]
(11) CANADA ESPAÑA RUSIA FRANCIA
Después de limpiar la lista, el tamaño es  0
```

Implementación de una lista con nodos

Ahora fijamos la atención en la implementación de la clase **LinkedList**. Discutiremos los métodos **addFirst, addLast, add(index, e), removeFirst, removeLast** y **remove(index)**.

Dejaremos el resto de los métodos para implementar en la sesión del grupo de trabajo.

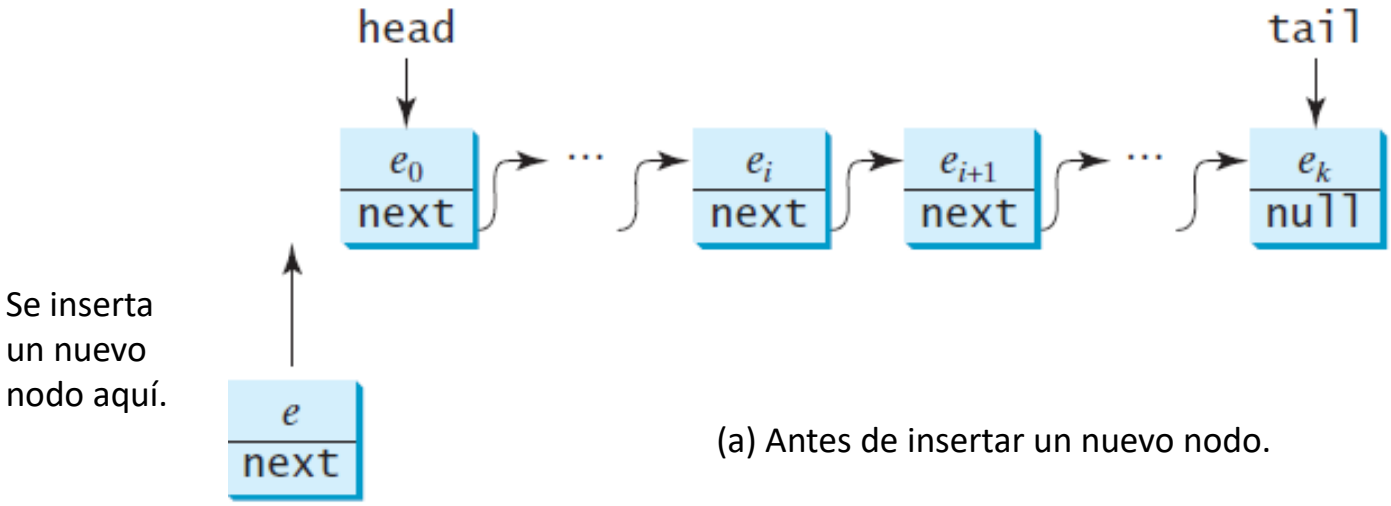


Implementación de addFirst(e)

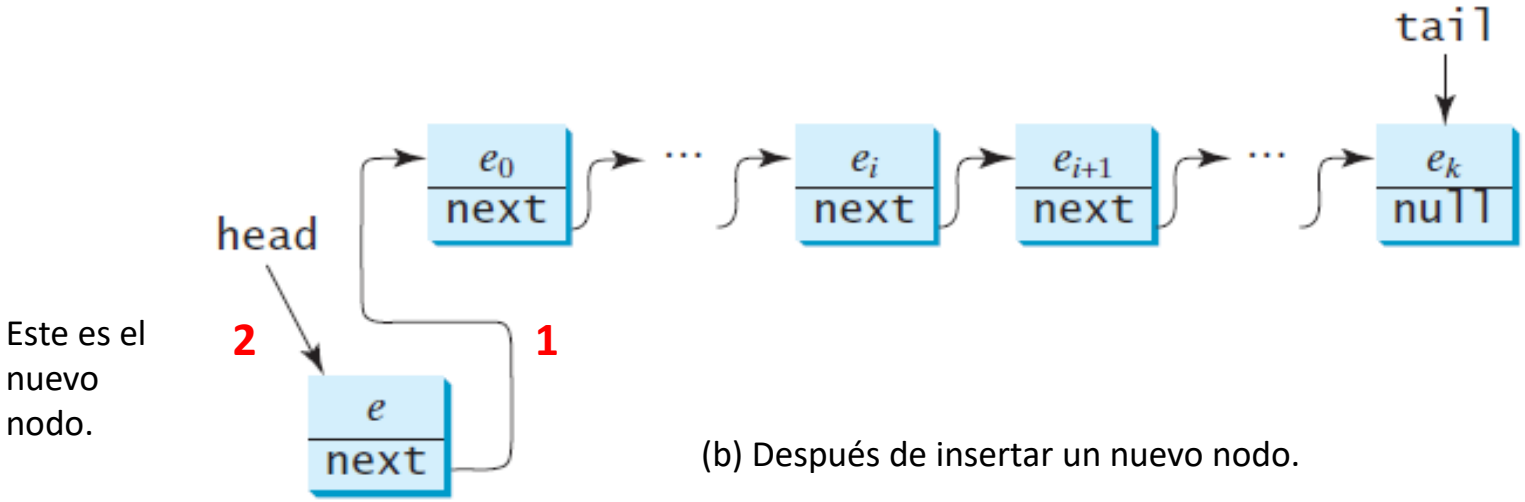
Este método crea un nodo que contiene el elemento **e**. El nuevo nodo pasa a ser el primer nodo de la lista. Podemos implementarlo como sigue:

```
public void addFirst(E e) {

  Node<E> nuevoNodo = new Node<E>(e); // Crea un nodo
  nuevoNodo.next = head; //enlaza el nuevoNodo con la cabeza
  head = nuevoNodo;   // la cabeza apunta al nuevoNodo
  size++;   //incrementa el tamaño de la lista

  if (tail == null)  //si la lista está vacia
    tail = head;     // el nuevoNodo es el único en la lista
}
```

Añade un nuevo elemento al principio de la lista



(a) Antes de insertar un nuevo nodo.



(b) Después de insertar un nuevo nodo.

Implementación de addLast(E, e)

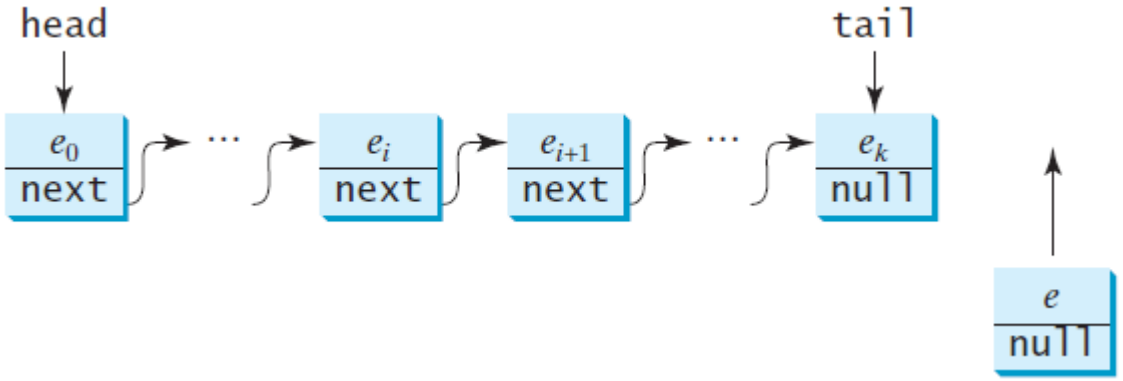
Este método crea un nodo que contiene el elemento **e** y añade el nodo al final de la lista. Podemos implementarlo como sigue:

```
public void addLast(E e) {
  Node<E> nuevoNodo = new Node<>(e); //Crea un nuevo nodo para e
  if (tail == null) {
    head = tail = nuevoNodo; // El único nodo en la lista
  }
  else {
    tail.next = nuevoNodo;//enlaza el nuevo nodo con el último nodo
    tail = tail.next; //la cola apunta ahora al último nodo
  }
  size++;  // incrementa el tamaño
}
```

Dos casos
- Si la lista está vacía, la cabeza y la cola apuntan al nuevo nodo.
- En otro caso, enlaza el nodo con el último nodo de la lista. La cola deberá apuntar ahora a ese nuevo nodo.

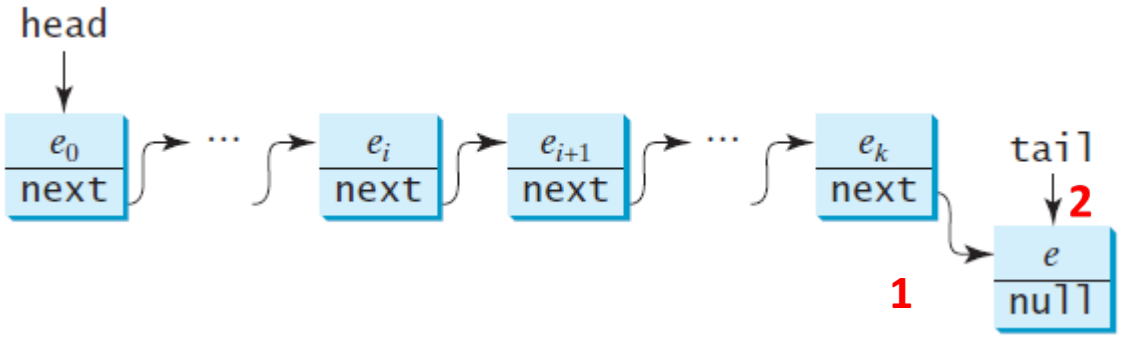
En cualquier caso, se incrementa el tamaño.

Añade un nuevo elemento al final de la lista



Se inserta un nuevo nodo aquí.

(a) Antes de insertar un nuevo nodo.



Este es el nuevo nodo.

(b) Después de insertar un nuevo nodo.

Implementación de add(index, e)

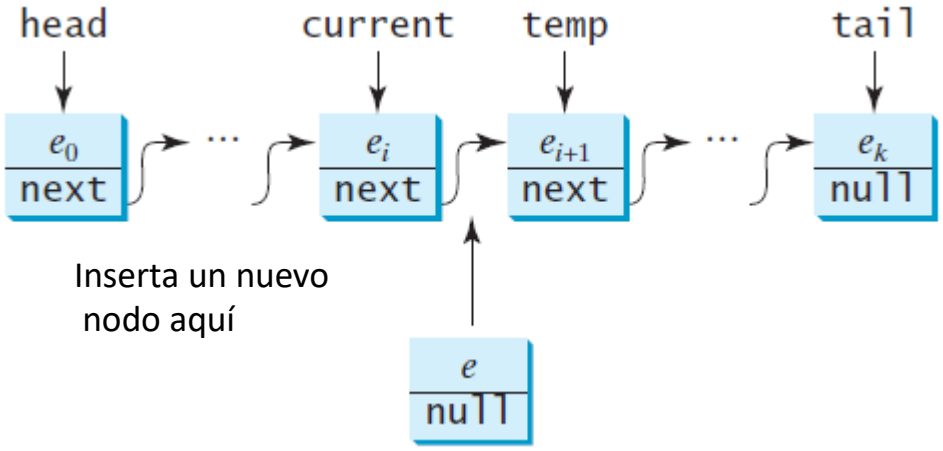
Este método inserta un elemento **e** en la lista en la posición index. Podemos implementarlo como sigue:

```
public void add(int index, E e) {
  if (index <= 0) addFirst(e); // Inserta al principio
  else if (index >= size) addLast(e); // Inserta al final
  else {// Inserta en medio
     Node<E> current = head;
     for (int i = 1; i < index; i++){
       current = current.next;          //Situo current
     }
     Node<E> temp = current.next;       //Situo temp
     current.next = new Node<E>(e);     //inserto el nuevo nodo
     (current.next).next = temp;
     size++;  // incrementa el tamaño
  }
}
```

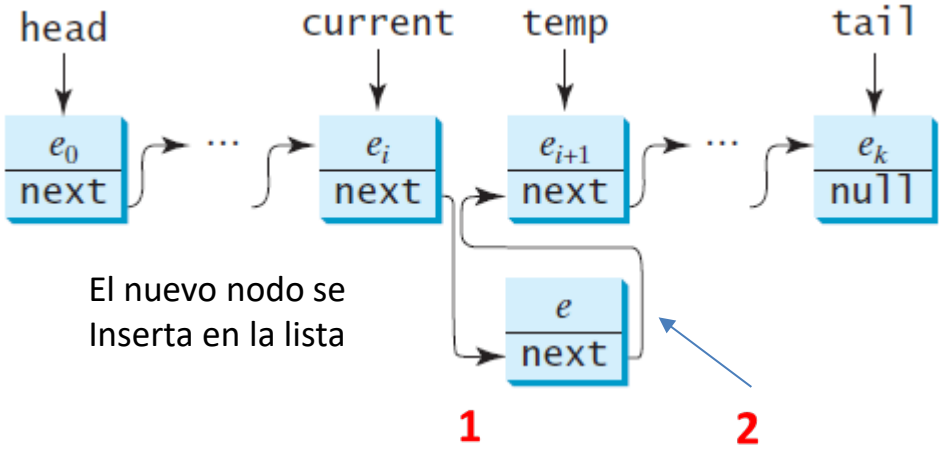
Cuando se inserta un elemento en una lista, se contemplan tres casos:

1. Si el índice, **index**, es 0, invocamos a **addFirst(e)** para insertar un elemento al principio de la lista.
2. Si el índice es mayor o igual que el tamaño, **size**, invocamos **addLast(e)** para insertar un elemento al final de la lista.
3. En otro caso, creamos un nodo para almacenar el nuevo elemento y localizamos donde insertarlo. Se puede ver en la figura, el nuevo nodo debe insertarse entre los nodos **current** y **temp**. El método asigna el nuevo nodo a **current.next** y asigna **temp** a **next** del nuevo nodo. El tamaño se incrementa en 1.

Inserta un nuevo elemento en medio de la lista



(a) Antes de insertar un nuevo nodo.



(b) Después de insertar un nuevo nodo.

Implementación de removeFirst( )

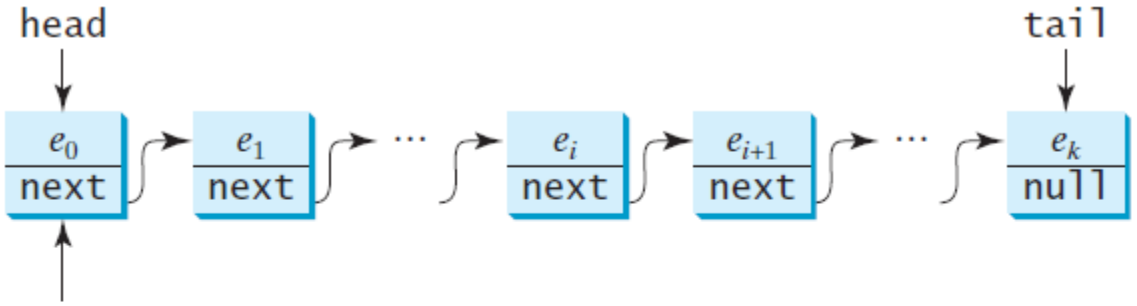
Este método elimina el primer elemento de la lista. Podemos implementarlo como sigue:

```
public E removeFirst( ) {

  if (size == 0) return null; // Nada que borrar
   else {
     Node<E> temp = head; // conserva el primer nodo temporalmente
     head = head.next; // mueve head para apuntar al siguiente nodo
     size--; // reduce en 1 el tamaño
     if (head == null) tail = null; // la lista se pone vacia
      return  temp.element; // devuelve el elemento borrado
   }
}
```

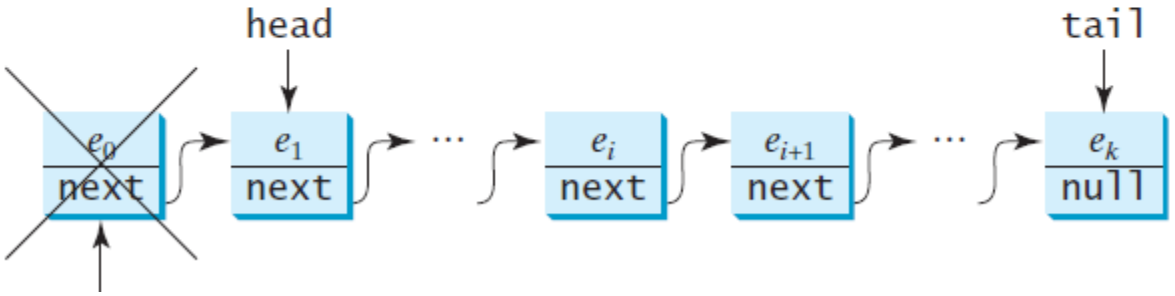
Elimina el primer nodo de la lista

Consideramos dos casos:

1. Si la lista está vacía, no hay nada que borrar, entonces devolvemos **null**.
2. En otro caso, eliminamos el primer nodo de la lista, apuntando **head** al segundo nodo. La figura que sigue muestra el antes y el después del borrado. El tamaño se reduce en 1 después de eliminar. Si la lista se pone vacía, después de eliminar el elemento, **tail** deberá ser **null**.



(a) Antes de que el nodo sea eliminado.



(b) Después de que el nodo sea eliminado.

Implementación de removeLast( )

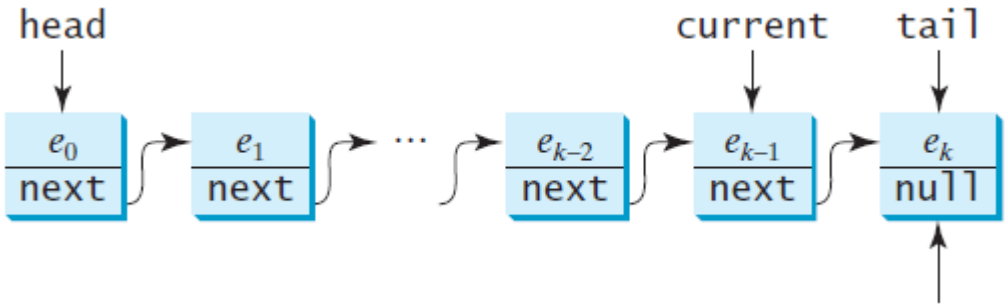
Este método elimina el último elemento de la lista. Podemos implementarlo como sigue:

```
public E removeLast( ) {
  if (size == 0) return null; // Nada que eliminar
  else if(size == 1){ // solo un elemento en la lista
       Node<E> temp = head;
       head = tail = null;  // la lista la hacemos vacía
       size = 0;
       return temp.element;
   }
 else {
       Node<E> current = head;
       for (int i = 0 ; i < size – 2;i++)
              current = current.next;
       Node<E> temp = tail;
       tail = current;
       tail.next = null;
       size--;
       return temp.element;
}
```

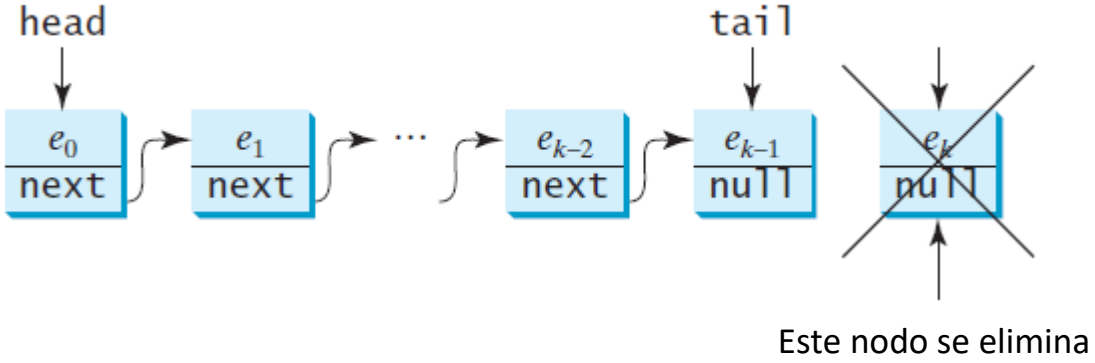
Consideraremos tres casos:

1. Si la lista está, devuelve **null**.
2. Si la lista contiene solo un nodo, el nodo se destruye; **head** y **tail** lo hacemos **null**. El tamaño lo ponemos a **0** después de eliminar y el valor del elemento eliminado se devuelve.
3. En otro caso, el último nodo se destruye y **tail** y el puntero se vuelve a posicionar apuntando al segundo nodo de la lista. La siguiente figura muestra el último nodo antes y después de ser eliminado. El tamaño se reduce en 1 después de eliminar y el valor del elemento del nodo borrado, se devuelve.

Elimina el último nodo de la lista



(a) Antes de eliminar el nodo.



(b) Después de eliminar el nodo.

Implementación de remove(index)

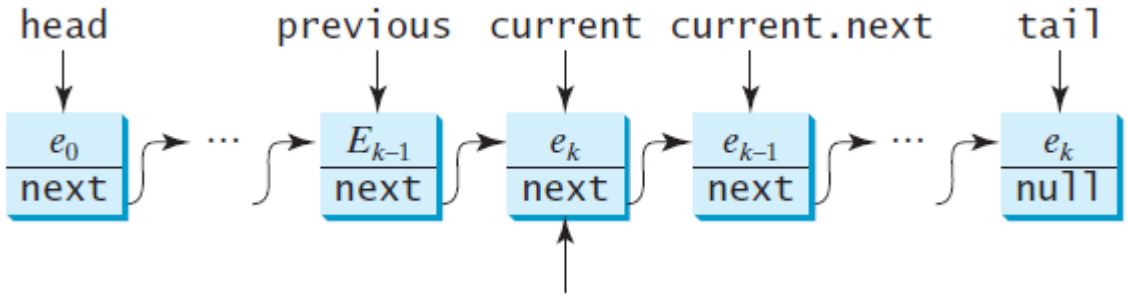
Este método encuentra el nodo en la posición especifica por index y lo elimina. Podemos implementarlo como sigue:

```
public E remove( int index) {
  if (index < 0 || index >= size)return null;// Fuera de rango
  else if(index == 0) return removeFirst();// elimina el primero
  else if(index == size - 1)return removeLast();//elimina el último
  else {
       Node<E> previous = head;
       for(int i = 1; i < index;i++){
          previous = previous.next;
       }
       Node<E> current = previous.next;
       previous.next = current.next;
       size--;
       return current.element;
}
```

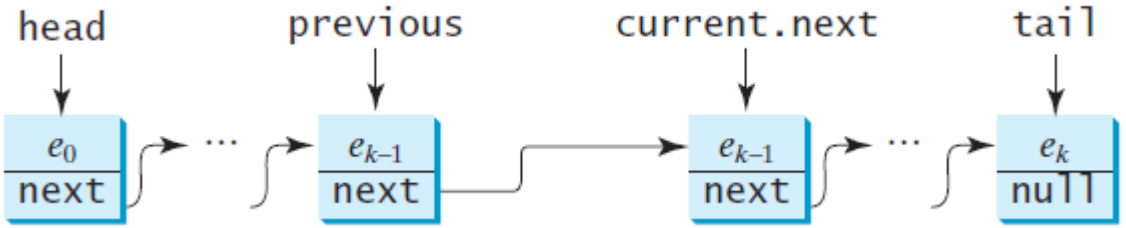
Consideraremos cuatro casos:

1. Si **index** está fuera del rango de la lista (por ejemplo, **index** < 0 || **index** >= **size**) devuelve **null**.
2. Si **index** es **0**, invoca **removeFirst( )** para eliminar el primer nodo.
3. Si **index** es **size – 1**, invoca **removeLast( )** para eliminar el último nodo.
4. En otro caso, localiza el nodo en la posición especificada por **index**. Deja **current** apuntando a ese nodo y **previous** apuntando al nodo anterior a ese nodo. Se puede ver en la figura siguiente (a). Asigna **current.next** a **previous.next** para eliminar el nodo **current** tal y como se muestra en la figura, (b)

Elimina un nodo de la lista

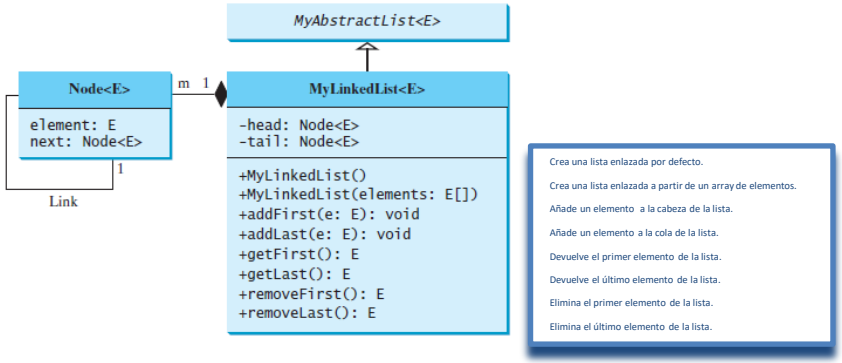


(a) Antes de eliminar el nodo.



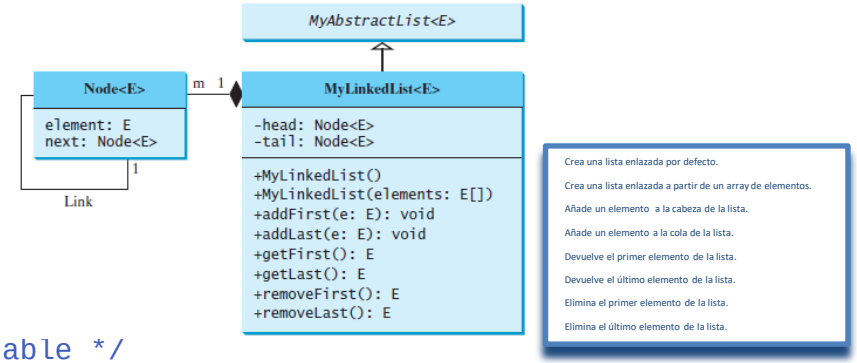
(b) Después de eliminar el nodo.

La clase LinkedList




```
import java.util.Iterator

public class LinkedList<E> extends AbstractList<E>{

  private Nodo<E> head, tail;

  /** Crea una lista por defecto */
  public LinkedList() {
  }

  /** Crea una lista a partir de un array de objetos */
  public LinkedList(E[] objects) {
      super(objects);
  }

  @Override /** Añade un nuevo elemento en la posición especifica por index */
  public void add(int index, E e) {
          // HECHO
   }

.....................................
```

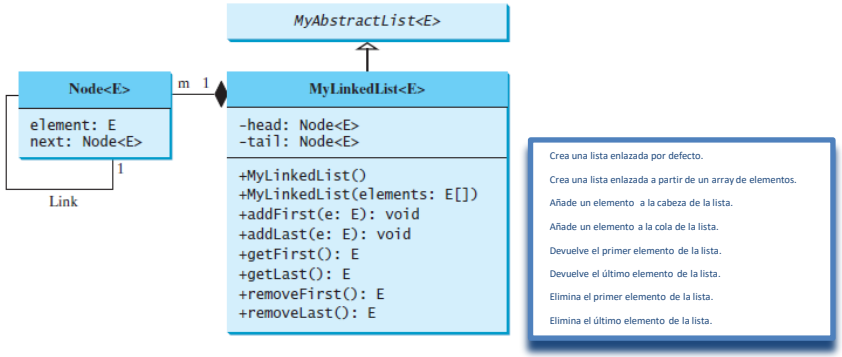
La clase LinkedList



.....

```
@Override
/** Sobre-escribe el método iterator() definido en Iterable */
public Iterator<E> iterator() {
    return null;
// Devuelve una instancia de LinkedListIterator
}

/** Esta clase implementa la interface Iterator*/
private class LinkedListIterator implements Iterator<E> {

    @Override
     public boolean hasNext() {
       return true;
    }

    @Override
     public E next() {
      return null;
     }

    @Override
    public void remove() {

    }
}
```

La clase LinkedList




```
  // Esta clase solo se usa en LinkedList, por eso es private.
  // Esta clase no necesita acceder a ningún miembro de instancia de LinkedList,
  // por lo que se define estático.

  private static class Node<E> {
    // Propiedades

     public Node(E element) {

    }
  }

}
```

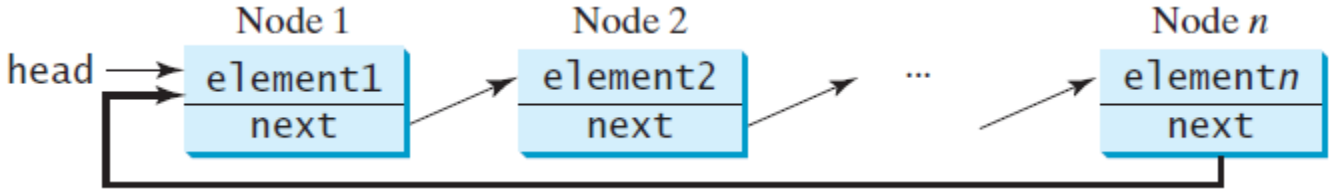
Complejidades de los métodos en ArrayList y LinkedList

Métodos	*ArrayList*	*LinkedList*
`add(e: E)`	$O(1)$	$O(1)$
`add(index: int, e: E)`	$O(n)$	$O(n)$
`clear()`	$O(1)$	$O(1)$
`contains(e: E)`	$O(n)$	$O(n)$
`get(index: int)`	$O(1)$	$O(n)$
`indexOf(e: E)`	$O(n)$	$O(n)$
`isEmpty()`	$O(1)$	$O(1)$
`lastIndexOf(e: E)`	$O(n)$	$O(n)$
`remove(e: E)`	$O(n)$	$O(n)$
`size()`	$O(1)$	$O(1)$
`remove(index: int)`	$O(n)$	$O(n)$
`set(index: int, e: E)`	*O(1)*	$O(n)$
`addFirst(e: E)`	$O(n)$	$O(1)$
`removeFirst()`	$O(n)$	$O(1)$

Variaciones de Listas Enlazadas

Las listas enlazadas introducidas en las anteriores secciones se conocen como *listas enlazadas simples.* Contienen un puntero al primer nodo y cada nodo contiene un puntero al siguiente nodo secuencialmente. Algunas variaciones de la lista enlazada son útiles en ciertas aplicaciones.

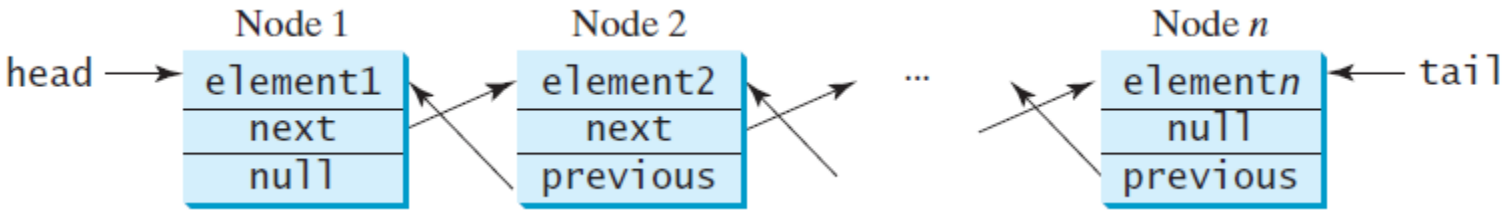
Una lista *circular* es similar a una lista *enlazada simple* excepto en que el puntero del último nodo apunta al primer nodo. Se muestra a continuación.



(a) Lista enlazada circular

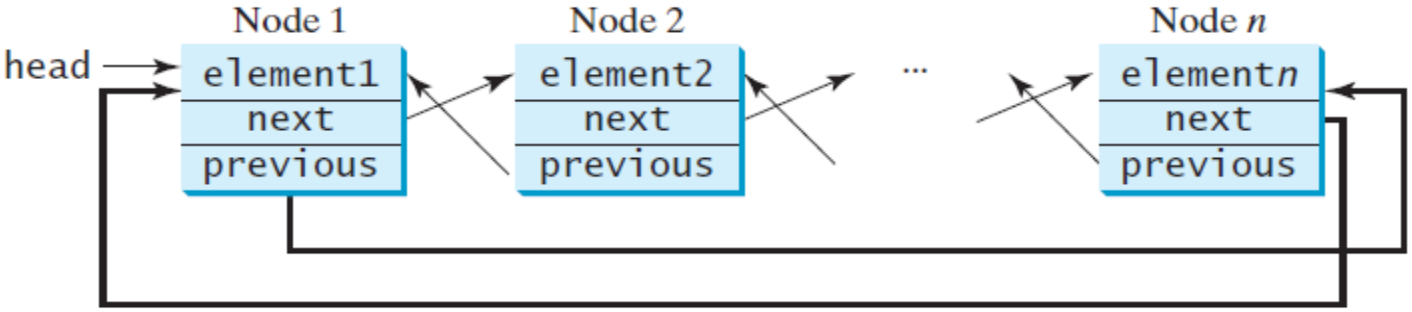
Se puede observar que en las listas enlazadas circulares no se necesita **tail**. **head** apunta al nodo actual (current) en la lista. La inserción y eliminación se llevará a cabo en el nodo actual de la lista. Una buena aplicación de las listas enlazadas circulares está en el sistema operativo que sirve a varios usuarios compartiendo el tiempo. El sistema coge a un usuario de la lista circular y le otorga una pequeña cantidad de tiempo de CPU, luego pasa al siguiente usuario de la lista.

Una *lista doblemente enlazada* contiene nodos con dos punteros. Un puntero al siguiente nodo y otro al nodo previo, anterior. Estos dos nodos se llaman *puntero adelante* y puntero *hacía atrás.* Estas listas doblemente enlazadas se pueden recorrer adelante y hacía atrás. La clase **java.util.LinkedList** se implementa usando una lista doblemente enlazada y supone recorrer adelante y hacía atrás usando **ListIterator**.



(b) Lista doblemente enlazada

Una *lista doblemente enlazada circular* es una lista doblemente enlazada excepto que el puntero adelante del último nodo apunta al primer nodo y el puntero hacía atrás del primer nodo apunta al último nodo.



(c) Lista doblemente enlazada circular

Resumen

- Hemos aprendido como implementar listas con arrays y estructuras enlazadas.

- Para definir una estructura de datos es esencial definir una clase. La clase para la estructura de datos debería usar campos de datos para almacenar datos y proporcionar métodos para dar soporte a las operaciones tales como inserción y eliminación.

- Crear una estructura de datos es crear una instancia de la clase. Podemos aplicar los métodos de instancia para manipular la estructura de datos tales como insertar un elemento en la estructura o eliminar un elemento de la estructura.

¡MUCHAS GRACIAS!

UNIVERSIDAD DE ALMERÍA