UD4

COLECCIONES Y ESTRUCTURAS DE DATOS

4.1 Colecciones y listas

MP_0485
Programación

Introducción

Los arrays nos ofrecen una interesante forma de estructurar datos en la memoria, pero tienen el problema de que es necesario saber previamente la longitud o número de elementos que tendrá cada array. Para resolver este problema podemos utilizar otro recurso que nos ofrece Java; las clases que representan colecciones y las clases que representan mapas

Colecciones y mapas sirven para almacenar en la memoria un grupo o conjunto de elementos, al igual que los arrays, pero tienen la gran ventaja de crecer dinámicamente según las necesidades del programa en cada momento.

Java cuenta con muchísimas clases que sirven para representar colecciones de datos y mapas, clases que forman parte de una compleja jerarquía. En la siguiente imagen te mostramos la jerarquía que corresponde a las clases con las que vamos a trabajar.

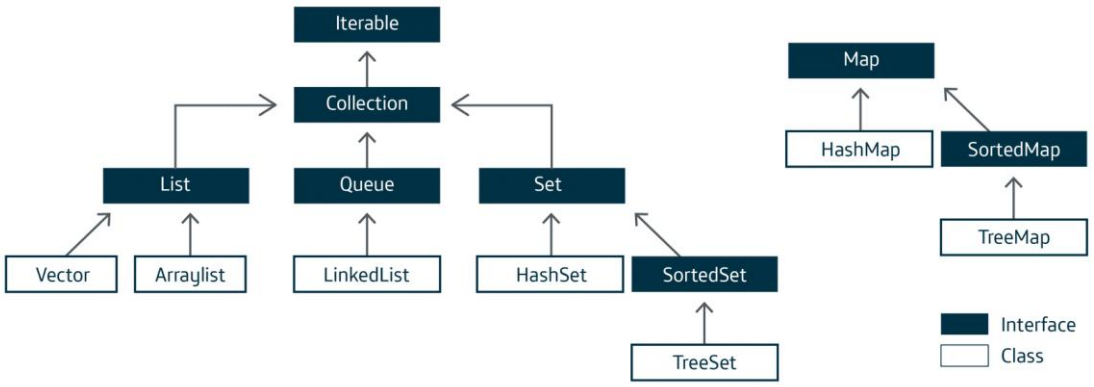

Figura 1: Estructura jerárquica de las interfaces Collection y Map.

En la imagen hemos simplificado el árbol jerárquico de las colecciones y mapas, eliminando algunas ramas y dejando lo más relevante. Lo importante es que comprendas que cuando utilizas una clase, esta no está aislada, sino que forma parte de una estructura jerárquica más compleja.

Ten en cuenta que en la imagen los rectángulos con fondo negro son interfaces y los rectángulos con fondo blanco son clases. Así puedes fácilmente deducir que la clase Vector implementa la interfaz List, e indirectamente, también las interfaces Collection e Iterable. También puedes comprobar que la clase LinkedList implementa las interfaces List y Queue, y también, de manera indirecta, las interfaces Collection e Iterable.

Observa además que las clases que implementan la interfaz Map no implementan las interfaces Collection e Iterable, es decir, están separadas y no forman parte de la misma estructura o familia jerárquica. Aunque esto es relativo, porque ya sabes que todas las clases heredan de la superclase Object, así que de alguna manera cualquier clase está unida a otra a través de la superclase Object.

Diferencia entre una colección y un mapa

Tanto colecciones como mapas representan un conjunto o colección de elementos, pero hay una importante diferencia entre ambos:

- Los mapas no implementan las interfaces Iterable y Collection, tal como puedes apreciar en la imagen. En el siguiente apartado aprenderás qué aportan estas interfaces a las clases que las implementan.
- Técnicamente llamamos colecciones a las clases que implementan la interfaz Collection. Aunque en la práctica también se llama colecciones a los mapas, porque representan una colección o conjunto de elementos. De ahí que ambos conceptos se confundan a menudo.
- La diferencia importante a nivel operativo es que, aunque tanto mapas como colecciones almacenan una colección o conjunto de objetos, los mapas asocian una clave a cada elemento, que servirá después como llave para acceder a dicho elemento.

Tipos de colecciones

Observando de nuevo la fig.1, puedes apreciar que hay colecciones que implementan la interfaz List (Vector, ArrayList y LinkedList) y colecciones que implementan la interfaz Set (HashSet, TreeSet). Clasificaremos las colecciones según se trate de listas o conjuntos.

- **Listas:** objetos que pertenecen a una clase que implementa la interfaz List. Los elementos que se añaden a una lista estarán siempre situados según el orden en que se han añadido. Se admiten valores duplicados.
- **Tipo devuelto**: objetos que pertenecen a una clase que implementa la interfaz Set, inspirada en la teoría de conjuntos de matemáticas. Los elementos que se añaden no guardan ningún orden específico, ni se sitúan en el orden en que se van introduciendo. La característica más importante de los conjuntos es que **no admiten valores duplicados**.

Colecciones y mapas: estructura

Todas las clases e interfaces que se encuentran en este esquema pertenecen al paquete java.util, a excepción de la interfaz Iterable, que se encuentra en java.lang.

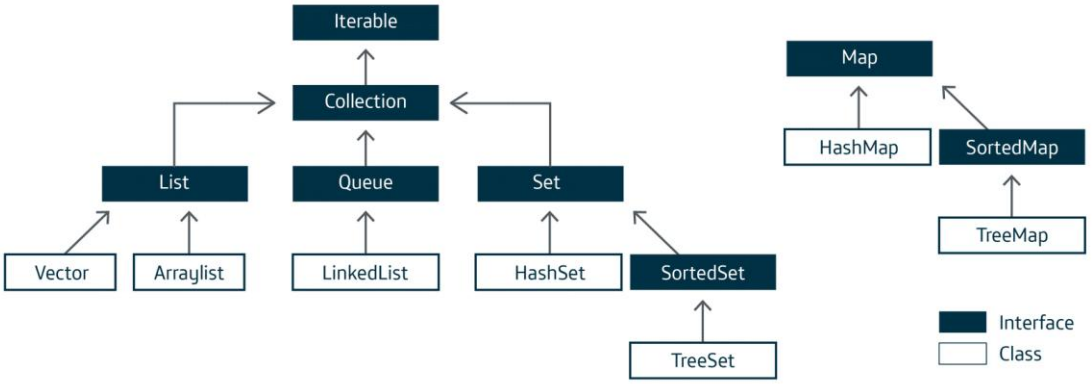

Figura 2: Estructura jerárquica numerada de las interfaces Collection y Map.

- **Iterable:** Las clases que implementan la interfaz Iterable tienen la característica de que sus elementos pueden recorrerse con una estructura de tipo foreach y además poseen el método iterator(). Trabajaremos con estos conceptos más adelante.

- **Map:** Los mapas asocian una clave a cada elemento, que servirá después como llave para acceder a dicho elemento.

- **Collection:** Las clases que implementan la interfaz Collection representan una colección dinámica de objetos. Internamente trabajan con arrays.

- **List:** Interfaz que implementa las colecciones de datos que se organizan en forma de lista, es decir, un elemento detrás de otro en el orden en que se van introduciendo.

- **Queue:** Las clases que implementan la interfaz Queue representan colecciones cuyos elementos se organizan en forma de cola. Una cola es una estructura de datos en memoria RAM donde el primer elemento que entra, es el primero en salir. También se denominan estructuras FIFO, abreviatura del inglés “First In, First Out”.

- **Set:** Conjuntos: los elementos que se añaden no se sitúan en el orden natural en que se van introduciendo. La característica más importante de los conjuntos es que no admiten valores duplicados.

- **HashMap:** Colección de elementos donde cada elemento tiene asociada una clave. Los elementos no se sitúan en el orden en que se van añadiendo.

- **SortedMap:** Interfaces cuyas clases que la implementan representan mapas cuyos elementos quedan ordenados según la clave.

- **TreeMap:** Colección de elementos donde cada elemento tiene asociada una clave. Los elementos se sitúan en orden de clave.

- **Vector:** Lista basada en un array tipo vector con una longitud inicial de 10 elementos, que irá creciendo en 10 elementos más cuando sea necesario, permitiendo que crezca dinámicamente. Sus métodos son sincronizados, es decir, es útil cuando trabajamos con programas multitarea utilizando distintos hilos de ejecución.

- **ArrayList:** Lista basada en un array tipo vector con una longitud inicial de 10 elementos, que irá creciendo en 10 elementos más cuando sea necesario, permitiendo que crezca dinámicamente. A diferencia de la clase Vector, sus métodos no son sincronizados.

- **LinkedList:** Basado en una lista enlazada y con los elementos organizados en forma de cola. Mucho más eficiente que Vector y ArrayList cuando hay que hacer muchas inserciones y eliminaciones.

- **HashSet:** Colección de objetos que no guardan ningún tipo de orden. Los elementos no se sitúan en el mismo orden que se han ido introduciendo. Puesto que se trata de un conjunto, no admite duplicados, es decir, no admite ningún nuevo elemento que, aplicándole el método equals sobre cualquiera de los existentes, dé true.

- **SortedSet:** Las clases que implementan esta interfaz representan colecciones de objetos que pueden ordenarse en función del método compareTo de cada elemento con el resto de los elementos, y además no admiten duplicados, puesto que se trata de conjuntos.

- **TreeSet:** Colección de objetos que pueden ordenarse en función del método compareTo de cada elemento con el resto de los elementos y además, no admiten duplicados, es decir, no admite ningún nuevo elemento que, aplicándole el método equals sobre cualquiera de los existentes, dé true.

ArrayList y Vector

Las clases ArrayList y Vector son muy similares, ya que ambas basan su estructura de datos interna en un array que crece dinámicamente. Si no se especifica lo contrario, inicialmente crean un array de 10 elementos, que duplicará su tamaño cuando el número de elementos sobrepase su capacidad. Es posible indicar en el constructor el número de elementos iniciales. Además, el constructor de Vector también admite un segundo argumento para especificar el incremento.

La diferencia entre Vector y ArrayList es que un vector tiene propiedad de sincronización, algo útil para la concurrencia, es decir, cuando realizamos programas multitarea utilizando distintos hilos de ejecución, ya que garantiza mayor seguridad.

ArrayList no es sincronizado, por lo que resulta más eficiente si no estamos trabajando con hilos y métodos sincronizados. Los ejemplos que veremos a lo largo de este apartado estarán basados en la clase ArrayList, pero podrías aplicarlos igual con un objeto de la clase Vector.

ArrayList y Vector son clases genéricas, por lo que hay que especificar la clase a la que pertenecen los elementos que portará la lista. Veamos varios ejemplos:

```
ArrayList<String> textos = new ArrayList<String>();
```

Lista inicial de 10 elementos de tipo String. En el momento que se añada el elemento 11, se añadirán otros 10, quedando la longitud del array en 20 elementos.

```
ArrayList<String> textos = new ArrayList<String>(15);
```

Lista inicial de 15 elementos de tipo Triángulo. En el momento en que se añada el elemento 16, se duplicará el array.

> Igual que ocurría con los arrays, en las colecciones el primer elemento ocupa la posición 0

Métodos de la interfaz Collection

int size()

El método size devuelve el número de elementos que contiene la lista. En el ejemplo anterior estamos utilizando este método para controlar cuándo deber terminar el bucle for.

boolean add(Object element)

Añade el elemento pasado como argumento al final de la lista. Devuelve true si el elemento se ha añadido con éxito y devuelve false si, por alguna circunstancia, el elemento no se ha podido añadir.

boolean remove(Object obj)

Elimina de la colección el objeto pasado como argumento. Si el objeto no ha podido eliminarse por cualquier circunstancia, devuelve false, de lo contrario devuelve true.

void clear()

Elimina todos los elementos dejando la colección vacía.

boolean contains(Object o)

Devuelve true si la colección contiene el objeto especi cado como argumento, de lo contrario devuelve false.

boolean isEmpty()

Devuelve true si la colección no contiene ningún elemento, de lo contrario devuelve false.

Métodos de la interfaz List

Object get(int pos)

El método get devuelve el elemento que ocupa la posición especificada como argumento. En el anterior ejemplo lo estamos utilizando para recuperar el objeto Triangulo que ocupa la posición 1 y después para recorrer cada uno de los objetos Triangulo por medio de un bucle for.

boolean add(int pos, Object element)
También es posible añadir un nuevo elemento en la posición deseada. Puedes probar a añadir el siguiente código al ejemplo:

```
trians.add(1, new Triangulo(7, 8, 9));
System.out.println("Recorriendo de nuevo todos los elementos");
for (int i=0; i<trians.size(); i++) {
	System.out.println(trians.get(i).verTipo());
}
```

El nuevo triángulo de lados 7, 8 y 9 es insertado en la posición 1 y los triángulos que antes ocupaban las posiciones 1 y 2 han sido desplazados una posición.

Object remove(int pos)
Elimina el elemento que ocupa la posición especificada en el argumento y retorna el elemento borrado. Puedes ponerlo en práctica añadiendo este código:

```
Triangulo t = trians.remove(2);
System.out.println("Se ha eliminado el triángulo: " + t.verTipo());
```

Se ha eliminado el elemento que ocupaba la posición indicada y los elementos que le seguían han tenido que desplazarse una posición hacia arriba.

> Todas las clases que implementan la interfaz Collection también implementan indirectamente la interfaz Iterable, lo que convierte a sus objetos en colecciones de elementos iterables, es decir, que se pueden recorrer secuencialmente

Recorrer las colecciones

Para recorrer o iterar las colecciones tenemos varias posibilidades, la primera opción es muy similar a como recorríamos los arrays.

```
for (int i=0; i<trians.size(); i++) {
	System.out.println(trians.get(i));
}
```

Sin embargo, esta forma no es la más cómoda en muchas ocasiones o incluso en algunos momentos es hasta menos eficiente.

Iterar con la estructura for each

La estructura for each (para cada) es un formato especial de la instrucción for que permite repetir un grupo de sentencias tantas veces como elementos tenga una colección o un array.
El formato es el siguiente:

```
for (Object obj : colección_o_array) {
	// Bloque de sentencias
}
```

Por ejemplo, para recorrer los elementos de una colección de objetos Triangulo:

```
ArrayList<Triangulo> trians = new ArrayList<Triangulo>();

trians.add(new Triangulo(1, 2, 3));
trians.add(new Triangulo(1, 1, 2));
trians.add(new Triangulo(1, 1, 1));

for (Triangulo t : trians) {
	System.out.println(t);
}
```

El bucle se ejecuta tantas veces como elementos tenga la colección trians. En cada iteración guardará el elemento correspondiente en la variable t.

Iterar con el método iterator()

El método iterator() retorna un objeto de la clase Iterator que provee de un mecanismo para recorrer secuencialmente los elementos de una colección a través de un cursor. Llamamos cursor a una marca que se va desplazando y que va posicionándose en el siguiente elemento a leer dentro de la colección.

Veamos un ejemplo:

```
ArrayList<Triangulo> trians = new ArrayList<Triangulo>();

trians.add(new Triangulo(1, 2, 3));
trians.add(new Triangulo(1, 1, 2));
trians.add(new Triangulo(1, 1, 1));

Iterator<Triangulo> itera = trians.iterator();
Triangulo cadaTriangulo;
while (itera.hasNext()) {
	cadaTriangulo = itera.next();
	System.out.println(cadaTriangulo.toString());
}
```

Primero, hemos tenido que obtener el objeto Iterator asociado a la colección trians.

```
Iterator<Triangulo> itera = trians.iterator();
```

Nuestro nuevo objeto Iterator se llama itera y estamos utilizándolo para acceder secuencialmente a los elementos de la colección trians, de manera similar a como se recorren los registros de un fichero secuencial. Para lograrlo utilizamos los siguiente métodos de la clase Iterator:

boolean hasNext()

Esta función devuelve true mientras sigan quedando elementos para iterar en la colección. En el ejemplo lo estamos utilizando para ejecutar el bucle mientras haya más elementos que recorrer.

Object next()

Devuelve el siguiente elemento de la colección.

```
CadaTriangulo = itera.next();
```

Esta sentencia devuelve el objeto Triangulo que corresponde a la posición donde se encuentra el cursor de lectura.

```
while (itera.hasNext()) {
	cadaTriangulo = itera.next();
	System.out.println(cadaTriangulo.verTipo());
}
```

La expresión `Itera.hastNext()` devuelve el valor true si el puntero no se encuentra al final de la colección, es decir, si todavía hay elementos que leer. La estructura `while` nos está permitiendo acceder secuencialmente a cada elemento mientras queden más elementos para leer.

For-each vs iterator

¿Cuándo utilizo la estructura for-each y cuándo un objeto Iterator? La estructura for-each parece más sencilla cuando solo quiero recorrer los elementos, por ejemplo, para realizar un listado. Pero si lo que deseamos es eliminar de la colección los elementos que cumplan una determinada condición, entonces podemos encontrarnos con una sorpresa.

Observa detenidamente el siguiente ejemplo donde, dentro de una colección de nombres de personas, queremos eliminar a aquellas personas que se llaman Carmen.

```
import java.util.ArrayList;
public class Principal {
	public static void main(String[] args) {
		ArrayList<String> nombres = new ArrayList<String>();
		nombres.add("Carmen");
		nombres.add("Rosa");
		nombres.add("Carmen");
		nombres.add("Miguel");
		nombres.add("Carmen");

		for (String n : nombres) {
			System.out.println(n);
			if (n.equals("Carmen")) {
				nombres.remove(n);
			}
		}
	}
}
```

Parece lógico y sencillo. Sin embargo, este programa provoca error de ejecución **porque NO permite eliminar el elemento sobre el que se está iterando.**

```
Exception in thread "main" java.util.ConcurrentModi cationException
at java.util.ArrayList$Itr.checkForComodi cation(ArrayList.java:901)
at java.util.ArrayList$Itr.next(ArrayList.java:851)
at Principal.main(Principal.java:14)
```

Para solucionar este problema podemos usar el método Iterator.remove(), que elimina el objeto que acaba de iterarse. Esto vendría a resolver el problema del ejemplo anterior.

```
import java.util.ArrayList;
import java.util.Iterator;
public class Principal {
	public static void main(String[] args) {
		ArrayList<String> nombres = new ArrayList<String>();
		nombres.add("Carmen");
		nombres.add("Rosa");
		nombres.add("Carmen");
		nombres.add("Miguel");
		nombres.add("Carmen");

		Iterator<String> itera = nombres.iterator();
		String cadaNombre;
		while (itera.hasNext()) {
			cadaNombre = itera.next();
			System.out.println(cadaNombre);
			if (cadaNombre.equals("Carmen")) {
				itera.remove();
			}
		}

		System.out.println("Recorrido después de borrar las Carmene
		for (String n : nombres) {
			System.out.println(n);
}}}
```

LinkedList: listas enlazadas

En este apartado estudiaremos la clase LinkedList. Observando el diagrama, puedes comprobar que también deriva de List, Collection e Iterable. Todos los ejemplos que has realizado en el apartado anterior seguirían funcionando exactamente igual cambiando el tipo de referencia ArrayList por LinkedList, ya que comparten la mayoría de los métodos.

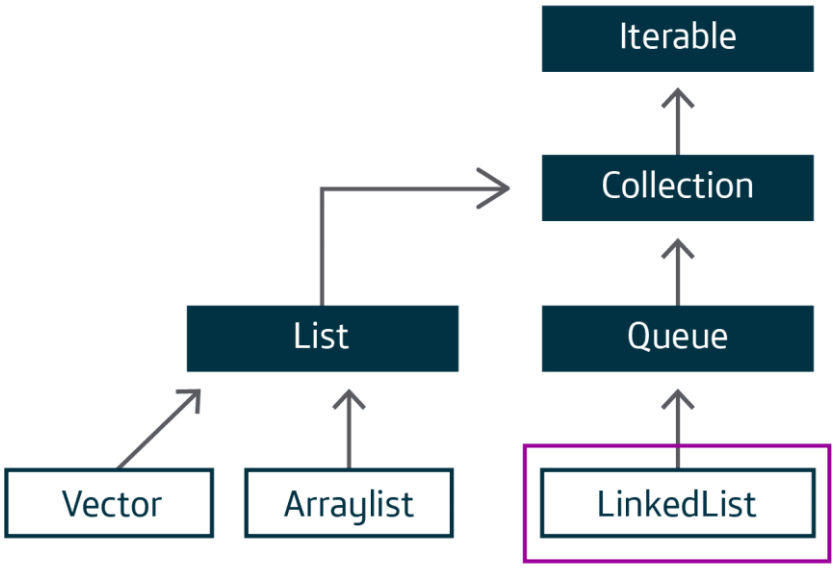

Figura 3: Estructura jerárquica de las interfaces Collection y List.

¿Exactamente igual? Parece que funciona igual, pero, en realidad, una colección LinkedList es diferente a una colección ArrayList. Los elementos están organizados en memoria RAM de una manera muy distinta. Descubriremos las ventajas de uno sobre otro.

Puedes poner en práctica estos dos ejemplos que ya utilizamos anteriormente, pero ahora con objetos LinkedList.

```
import java.util.LinkedList;
public class Principal {
	public static void main(String[] args) {
		LinkedList<String> nombres = new LinkedList<String>();
		nombres.add("Carmen");
```

```
            nombres.add("Rosa");
            nombres.add("Carmen");
            nombres.add("Miguel");
            nombres.add("Carmen");

            for (String n : nombres) {
                  System.out.println(n);
            }
      }
}
```

Hay dos características muy importantes que hacen de LinkedList un tipo de colección muy especial:

- Un objeto de tipo LinkedList es una lista enlazada.
- Un objeto de tipo LinkedList es una cola, característica que está obligada a cumplir al implementar la interfaz Queue.

Las listas enlazadas son estructuras de datos organizadas como conjuntos de registros en memoria RAM, donde cada registro está situado a partir de una determinada dirección de memoria y enlazado con el siguiente.

Cada registro contendrá, además de los datos necesarios (en el ejemplo de la imagen: nombre, teléfono y dirección), la dirección de memoria del siguiente registro, excepto en el último registro, que contendrá null.

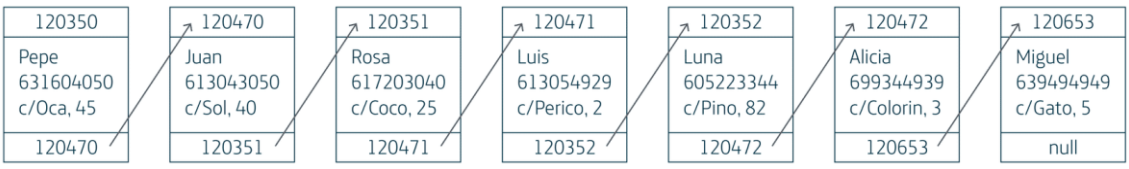

Figura 4: Direcciones de memoria y estructura de una lista enlazada.

Existen listas enlazadas y listas doblemente enlazadas, donde cada elemento contiene la dirección de memoria, no solo del siguiente elemento, sino también del anterior. Los objetos **LinkedList representan listas doblemente enlazadas**.

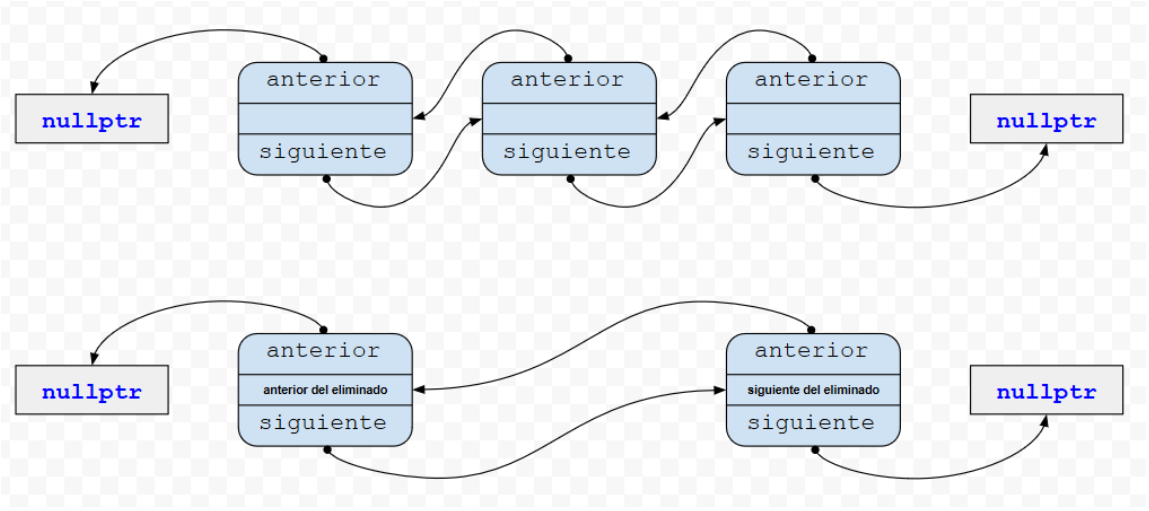

Figura 5: Direcciones de memoria y estructura de una lista doblemente enlazada. Además, se muestra un ejemplo de eliminación de un elemento.

¿Cuándo usar LinkedList y cuándo ArrayList?

El hecho de que los objetos LinkedList sean listas enlazadas resulta, en ocasiones, una ventaja y en otras, una desventaja.

Cuando insertamos un elemento en una posición determinada dentro de un ArrayList, el resto de los elementos deben desplazarse. Recuerda que un ArrayList está basado en un array interno cuyos elementos están situados en posiciones contiguas de memoria. Si el array es muy grande el tiempo de proceso puede ser grande. Si se añade un elemento al final no hay ningún problema.

Con LinkedList insertar un elemento no conlleva nada de tiempo, porque los elementos no ocupan posiciones contiguas de memoria, sino que pueden estar ubicados en cualquier parte, con tal de que apunten al anterior elemento y al siguiente elemento.

Es más eficiente usar LinkedList cuando tenemos una lista grande en la que hay que **realizar muchas inserciones en cualquier posición**.

Cuando eliminamos un elemento de posiciones intermedias en un ArrayList, todos los elementos que siguen deben recolocarse para no dejar huecos. Si el array es muy grande el tiempo de

proceso puede ser un problema. Con un LinkedList esto no ocurre, si se elimina un elemento intermedio basta con hacer que el anterior apunte al siguiente.

Es más eficiente usar LinkedList cuando tenemos una lista grande, donde hay que **realizar muchas eliminaciones en cualquier posición**.

Cuando un objeto ArrayList necesita crecer porque se ha llegado al nal del array, el sistema debe crear un nuevo array más grande y copiar todos los elementos del array antiguo. Esto no ocurre con un objeto LinkedList, ya que no está basado en arrays. Si vamos a utilizar un ArrayList y sabemos de antemano que crecerá mucho, puede ser interesante crearlo de antemano de un tamaño grande.

Los **objetos LinkedList pueden crecer** todo lo que sea necesario **sin peligrar el rendimiento**.

Si necesitamos acceder aleatoriamente a cualquier elemento, con ArrayList se accede directamente al elemento buscado sin tener que recorrer todos los elementos (acceso aleatorio).

```
buscado = lista.get(3);
```

Esta sentencia recupera el tercer elemento de un ArrayList sin necesidad de recorrer los elementos anteriores. La misma sentencia aplicada a un LinkedList también funciona, pero internamente tiene que recorrer todos los elementos hasta llegar al que ocupa la posición 3.

El acceso a un LinkedList solo se realiza secuencialmente, mientras que **el acceso a un ArrayList puede ser aleatorio**.

Un **ArrayList solo es mejor** cuando tenemos una lista grande y debemos **realizar muchas búsquedas aleatorias**, pero **no vamos a realizar inserciones o borrados**.

LinkedList es mejor cuando tenemos una lista grande donde debemos **realizar muchas inserciones y borrados** o **accesos secuenciales**.

LinkedList es también una cola. Una estructura de datos en memoria RAM donde el primer elemento que entró, es el primero en salir. También se denominan estructuras FIFO, abreviatura del inglés “First In, First Out”.

Referencias

Apuntes elaborados a partir de la siguiente documentación:

[1] Apuntes Fernando Barber y Ricardo Ferris. Universidad de Valencia.

[2] Apuntes Programación de José Antonio Díaz-Alejo. IES Camp de Morvedre.

[3] Apuntes de Programación de Carlos Cacho y Raquel Torres. Ceedcv.

[4] Apuntes de Programación Edix Digital Workers.