

Ejercicios Colecciones Enlazadas

[Descargar estos ejercicios](#)

Índice

1.  [Ejercicio 1](#)
2. [Ejercicio 2](#)

✓ Ejercicio 1

Partiendo de los pasos de diseño que hemos visto en los apuntes para implementar nuestra propia `ListaSimpleEnlazada<T>`. Vamos a hacer lo mismo implementando nuestra `ListaDoblementeEnlazada<T>` 'equivalente' a la colección `LinkedList<T>` de las BCL.

Para ello vamos a definir el tipo nodo asociado como ...

```
public class NodoListaDoblementeEnlazada<T> : IDisposable where T : IComparable<T>
```

donde los tipos almacenados deben implementar la interfaz `IComparable<T>`

Nuestra lista doblemente enlazada tendrá la siguiente definición

```
class ListaDoblementeEnlazada<T> : IDisposable, IEnumerable<T> where T : IComparable<T>
```

Definirá las siguientes propiedades...

```
public NodoListaDoblementeEnlazada<T> Primero { get; private set; }  
public NodoListaDoblementeEnlazada<T> Ultimo { get; private set; }  
public int Longitud { get; private set; }  
public bool Vacía => Longitud == 0;
```

los siguientes constructores ...

```
public ListaDoblementeEnlazada()  
public ListaDoblementeEnlazada(IEnumerable<T> secuencia)
```

y las siguientes operaciones 'equivalentes' a las del `LinkedList` ...

1. Implementación de `Dispose()` que llamará al `Dispose()` para cada uno de los nodos de la lista y pondrá a `null` `Primero` y `Ultimo`.

✦ **Nota:** Otra opción es ir llamando al método `Borra`, que implementaremos más adelante, para el primer nodo. Mientras la lista no esté vacía.

```
Dispose();  
public Clear() => Dispose();
```

2. Vamos a agregar nodos o datos al principio de la lista. Si te fijas en el diagrama de abajo, deberemos seguir una serie de pasos tal y como sucedía con la lista simple. Sin embargo ahora deberemos tener en cuenta que los nodos también apuntan al anterior.

```
public void AñadeAlPrincipio(NodoListaDoblementeEnlazada<T> nuevo)  
public void AñadeAlPrincipio(T dato)
```

Si describimos los pasos de añadir al principio tendremos ...

- **Paso 1:** El **siguiente** del nodo **nuevo** apuntará al **Primero** de la lista.

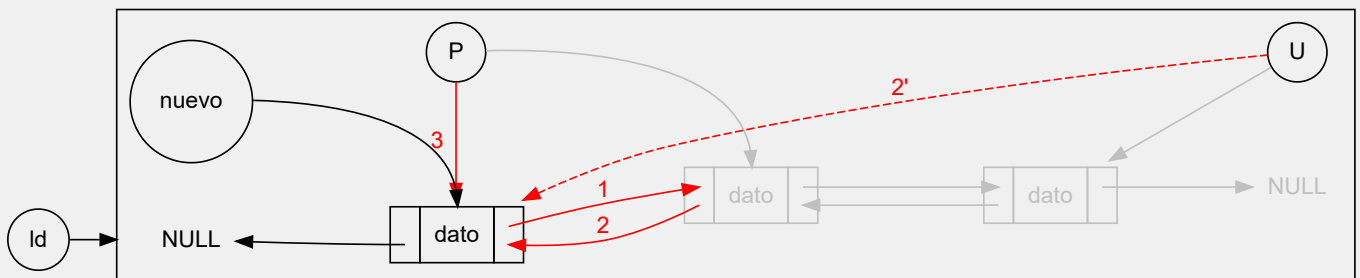
📌 **Nota:** No importa si la lista está vacía porque en ese caso **Primero** sería **null** y por tanto el **Siguiente** del nodo **nuevo** también apuntaría a **null**

- **Paso 2 y 2':** Si la lista **no está vacía** el **Anterior** del **Primero** apuntará al nodo **nuevo** y si está vacía esto ya no será necesario pero sí que **Ultimo** apuntará al nodo **nuevo**.

- **Paso 3:** El **Primero** será ahora el nodo **nuevo**.

📌 **Nota:** Esta operación no podemos hacerla antes o no podríamos hacer los pasos 1 y 2.
¡Piensaló!

- **Paso 4:** Qué no se nos olvide incrementar la propiedad **Longitud** ya que estamos insertando.



3. Vamos a agregar nodos o datos al final de la lista.

Dibújate un diagrama como el anterior, piensa en el orden de los pasos y piensa también si funcionaría con la lista vacía.

```
public void AñadeAlFinal(NodoListaDoblementeEnlazada<T> nuevo)
public void AñadeAlFinal(T dato)
```

4. Vamos a agregar un nuevo nodo o dato antes de otro nodo. Para ello, pasaremos el **nodo** donde vamos a insertar antes (que debe de existir y por lo tanto la lista no puede estar vacía) y el nodo **nuevo** que vamos a insertar inicializado con el dato.

```
public void AñadeAntesDe(
    NodoListaDoblementeEnlazada<T> nodo, NodoListaDoblementeEnlazada<T> nuevo)
public void AñadeAntesDe(
    NodoListaDoblementeEnlazada<T> nodo, T dato)
```

Si describimos los pasos de añadir antes de **nodo** ...

- **Pasos 1 y 2:** Actualizaremos los enlaces del nuevo nodo. Primero la referencia al **Siguiente** del nodo **nuevo** que apuntará a **nodo** y segundo la referencia al **Anterior** del nodo **nuevo** que apuntará al **Anterior** del **nodo** independientemente de si es null o no.

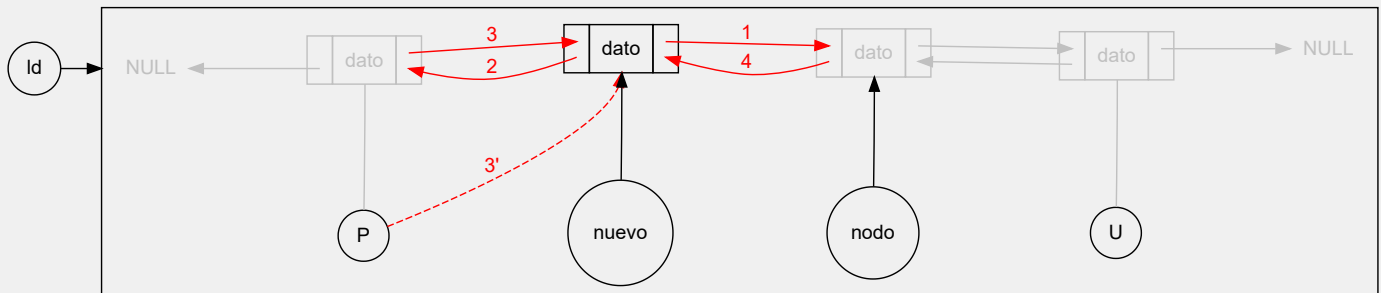
- **Paso 3 y 3':** Si en nodo donde insertamos antes **no es el primero** la referencia a **Siguiente** del nodo **Anterior** a **nodo**, apuntará ahora a **nuevo** y si fuera el primero esto ya no será necesario pero sí que **Primero** apuntará al nodo **nuevo**.

📌 **Nota:** Podemos saber si **nodo** es el primero cuando referencie al mismo nodo que **Primero** o si su referencia a **Anterior** es **null**.

- **Paso 4:** La referencia a **Anterior** de **nodo** apuntará ahora a **nuevo**.

🔗 **Nota:** Fíjate en el diagrama y piensa que el orden que hemos seguido es importante para no perder referencias y completar la operación.

- **Paso 5:** Qué no se nos olvide incrementar la propiedad **Longitud** ya que estamos insertando.



⚠ **Aviso:** En este punto se nos podría ocurrir la simplificación de que la operación **AñadeAlPrincipio** equivale a **AñadeAntesDe** el **Primero** con lo cual podremos reutilizar este último código para añadir el principio. Sin embargo esto último no funcionaría cuando la lista esté vacía pues **Primero** sería **null** y estaríamos añadiendo antes de **null**. Es por esa razón que **LinkedList<T>** también las considera operaciones diferentes.

5. Vamos a agregar nodos o datos al después de otro nodo.

Dibújate un diagrama similar al punto 4 y piensa en el orden de los pasos de forma equivalente. Ten en cuenta que si insertamos después del último la propiedad **Ultimo** deberá actualizarse.

```
public void AñadeDespuesDe(
    NodoListaDoblementeEnlazada<T> nodo,
    NodoListaDoblementeEnlazada<T> nuevo)
public void AñadeDespuesDe(T dato)
```

6. Borrar un nodo dado de la lista. Para realizar este método, te puedes basar el que se propone para la lista simplemente enlazada en los apuntes. Pero ten en cuenta que:

1. La lista se puede quedar vacía.
2. Puedo estar borrando el primero por lo que no hay anterior y deberé actualizar la propiedad **Primero**.
3. Puedo estar borrando el último por lo que no hay posterior y deberé actualizar la propiedad **Ultimo**.
4. Deberé la referencia a **Siguiente** del nodo anterior al que borro **si lo hay**, así como la referencia a **Anterior** del nodo posterior al que borro **si lo hay**.

🔗 **Nota:** Es importante que te dibujes un diagrama para saber actualizar las referencias.

```
public void Borra(NodoListaDoblementeEnlazada<T> nodo)
```

7. Buscar un dato en la lista equivalente al método **Find** de **LinkedList<T>**. Si encuentra el dato me devolverá el nodo que lo contiene y **null** si no lo encuentra.

```
public NodoListaDoblementeEnlazada<T> Busca(T dato)
```

8. Vamos a invalidar `ToStroing()` para que devuelve una cadena con los elementos de la lista entre corchetes y esos mismos elementos en orden inverso.

En ambos casos debes recorrer la lista para componer la cadena.

```
var ld = new ListaDoblementeEnlazada<int>(new int[]{2, 3, 4});  
Console.Write(ld); // Mostrará [1][2][3] - [3][2][1]
```

9. Por último, realiza el siguiente programa principal con el que podremos testear que nuestra lista funciona correctamente.

```
public static void Main()  
{  
    ListaDoblementeEnlazada<int> ld = new ListaDoblementeEnlazada<int>();  
    ld.AñadeAlPrincipio(4);  
    ld.AñadeAlPrincipio(3);  
    Console.WriteLine(ld);  
    ld.Clear();  
    ld.AñadeAlFinal(6);  
    ld.AñadeAlFinal(9);  
    ld.AñadeAlPrincipio(3);  
    Console.WriteLine(ld);  
    NodoListaDoblementeEnlazada<int> nodo = ld.Busca(6);  
    ld.AñadeAntesDe(nodo, 5);  
    ld.AñadeAntesDe(ld.PrimerO, 1);  
    ld.AñadeDespuesDe(nodo, 7);  
    ld.AñadeDespuesDe(ld.Ultimo, 12);  
    Console.WriteLine(ld);  
    ld.Borra(nodo);  
    ld.Borra(ld.PrimerO);  
    ld.Borra(ld.Ultimo);  
    Console.WriteLine(ld);  
}
```

Deberías obtener la siguiente salida ...

```
[3][4] - [4][3]  
[3][6][9] - [9][6][3]  
[1][3][5][6][7][9][12] - [12][9][7][6][5][3][1]  
[3][5][7][9] - [9][7][5][3]
```

Ejercicio 2

En este ejercicio se va a practicar el uso de la lista genérica de la BCL **LinkedList**, para ello deberás implementar la clase **CaracterASCII** que permitirá la gestión de un caracter en los tres modos de expresión que se suelen mostrar en las tablas ASCII (decimal, octal, hexadecimal y carácter).

```
DEC OCT HEX CAR
20  24  14  ¨
```

Para ello deberemos crear una propiedad:

- **Caracter** de tipo **char** privada de modificación y pública de acceso, con control de valor de entrada de forma que solo se podrán asignar valores entre 0 y 255. Lanzando una excepción interna a la clase **CharacterException** , con el mensaje del error ocurrido si no se cumple el rango.
- **CaracterDecimal** de tipo **string** de solo lectura, que devolverá la cadena en decimal de **Caracter** .
- **CaracterOctal** de tipo **string** de solo lectura, que devolverá la cadena en octal de **Caracter** .
- **CaracterHexadecimal** de tipo **string** de solo lectura, que devolverá la cadena en hexadecimal de **Caracter** .

💡 Puedes usar el método **ToString** de la clase **Convert** para convertir directamente a las distintas bases.

Un **Constructor** que permita construir un **CaracterASCII** a partir de un char.

Un **Constructor** que permita construir un **CaracterASCII** a partir de un short.

Anular el método **ToString** para que muestre el caracter en sus 4 posibles valores, como podemos ver en:

```
DEC OCT HEX CAR
20  24  14  ¨
```

Anular **Equals** para que permita comparar el objeto correctamente.

En la clase Program añadiremos los siguientes métodos y Main:

```
static void Main()
{
    LinkedList<CaracterASCII> listaAscii = new LinkedList<CaracterASCII>();
    GeneraElementos(listaAscii, 240); //METODO QUE DEBERAS CREAR PARA AÑADIR ELEMENTOS EN LA LISTA
    MuestraLista(listaAscii);
    Console.ReadLine();
}
private static void modificarDimensionesPantalla()
{
    Console.BufferWidth=150;
    Console.WindowWidth=150;
    Console.BufferHeight=40;
    Console.WindowHeight=40;;
}
```

```

static string Cabecera() => "DEC OCT HEX CAR  ";
static void Muestralista(LinkedList<CaracterAsCII> lista)
{
    modificarDimensionesPantalla();
    const short NUMERO_COLUMNAS = 8;
    for (int i = 0; i < NUMERO_COLUMNAS; i++) Console.Write(Cabecera());
    Console.WriteLine("\n");
    int columnas = 1;
    foreach (var caracter in lista)
    {
        Console.Write($"{caracter}  ");
        if (columnas % NUMERO_COLUMNAS == 0)
        {
            columnas = 0;
            Console.WriteLine();
        }
        columnas++;
    }
}


```

El método que deberás generar para conseguir la tabla Ascii con la cantidad de elementos que le pasemos como parámetro es **GeneraElementos(lista, numeroElementos)** . Este método solo podrá usar una LinkedList para conseguir los objetivos, que serán los siguientes:

- Generar un caracter AscII aleatorio (se recomienda que su rango esté entre **16 y 255** para evitar caracteres especiales que deformarán la salida).
- Crear con ese carácter una instancia CaracteAsCII.
- Comprobar si la instancia no está ya incluida en la lista (como hemos anulado equal, se realizará la comparación correctamente).
- En caso de no estar incluida, añadirla a la lista de forma ordenada. Por ejemplo, la salida con una llamada al método GeneraElementos con 16 elementos, los devolvería ordenados aunque no están todos.

DEC	OCT	HEX	CAR	DEC	OCT	HEX	CAR	DEC	OCT	HEX	CAR	DEC	OCT	HEX	CAR
37	45	25	%	87	127	57	W	90	132	5a	Z	96	140	60	`
111	157	6f	o	113	161	71	q	116	164	74	t	127	177	7f	△
134	206	86	?	151	227	97	?	158	236	9e	?	170	252	aa	ä
194	302	c2	Â	219	333	db	Û	228	344	e4	ä	243	363	f3	ó

Si se generan los 240 (255-15) elementos del código pasado en la main, se mostrará la lista completa.

 **Importante:** Recuerda que la LinkedList funciona con nodos, por lo que tendrás que recorrerlos usando la propiedad **Next** , y añadiendo antes o después del nodo correspondiente con los métodos de LinkedList apropiados. Además se tendrá que añadir el primer elemento con **AddFirst** para iniciar la lista.