Ejercicios Colecciones

Descargar estos ejercicios

Índice

☑ Ejercicio 1

Partiendo de los pasos de diseño que hemos visto en los apuntes para implementar nuestra própia `ListaSimplementeEnlazada<T>` . Vamos ha 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 Vacia => 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 ...

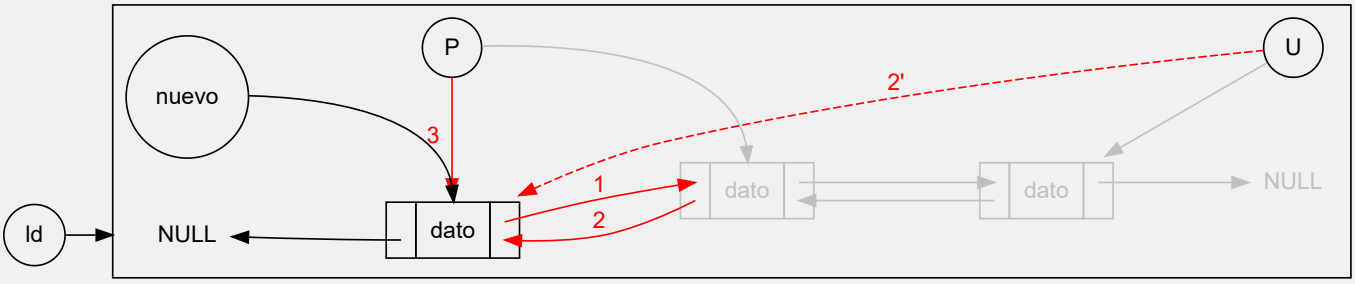
- **Paso 1**: El `Siguiente` del nodo `nuevo` apuntará al `Primero` de la lista.

  > 📌 **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(T dato)
```

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

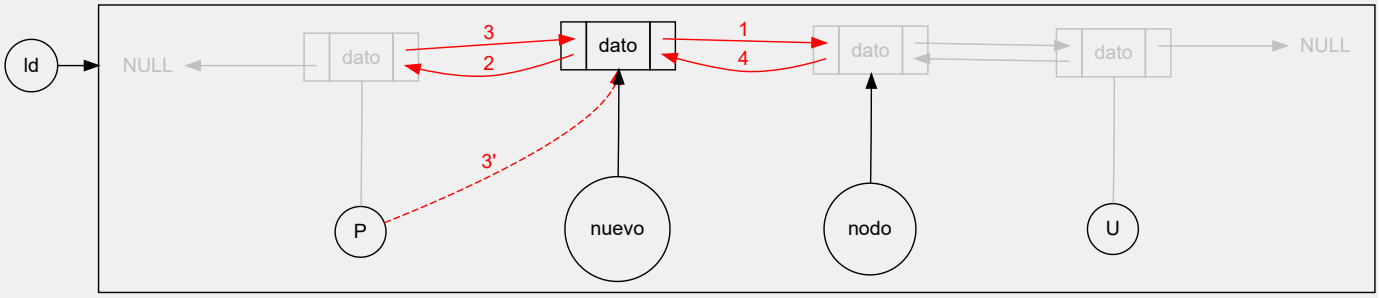
   - **Pasos 1 y 2**: Actualizaremos los enlaces del nuevo nodo. Primero la referencia al `Siguiente` del nodo `nuevo` que apuntará a `nodo` y segundo la referencia al `Anterior` del nodo `nuevo` que apuntará al `Anterior` del `nodo` independientemente de si es null o no.
   - **Paso 3 y 3'**: Si en nodo 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 vaciá 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, para ello deberás implementar los siguientes métodos y un programa para poder testear el funcionamiento de estos.

1. Implementa un método llamado `BorraEnteros` que reciba como parámetros una lista genérica de enteros (`List<int>` , que deberás inicializar previamente) y un número entero.
   Lo que hará será modificar la lista borrando los números que coincidan con el entero indicado.
2. Implementa una función llamada `Mezcla` que reciba como parámetro dos listas de enteros (ya ordenadas), y devuelva como resultado otra lista donde se unan las dos anteriores, pero con los números también en orden.
3. Implementa un método llamado `ImprimeInverso` que reciba como parámetros una lista de Personas y una posición (entero), e imprima por pantalla en orden inverso los nombres de la lista desde esa posición hasta el principio.
   Puedes utiliza la clase persona de otros ejercicios.
4. Implementa un procedimiento llamado `OrdenaCadenas` que reciba como parámetro una lista de strings la ordene alfabéticamente. Deberás usar algún algoritmo de ordenación y el método `CompareTo` para comparar las cadenas antes de realizar los intercambios.

✅ Ejercicio 3

Crea una clase `Automovil` con los datos básicos de los automóviles (marca, modelo, cilindrada, año de fabricación, etc) y los métodos necesarios para poder usarla con comodidad.

Crea una clase `Program` con una serie de métodos que nos permitan trabajar con una lista genérica de automóviles.

- Necesitaremos un Método `AñadeAutomovil` que a partir de una lista y un automóvil, añadirá este a la lista.
- `EliminaAutomovil` que eliminará el automóvil con el índice `i` que se haya pasado como argumento.
- Crea un método `AutomovilesPorAñoFabricacion`, que te permita encontrar en la lista los coches con una determinada fecha de fabricación y que retorne una nueva lista con esos datos.
- Otro método `AutomovilesPorAñoFabricacionYColor` que devuelva una sublista con los coches de la lista que sean de un determinado color y una fecha pasados ambos como parámetros.
- Método que permita mostrar el contenido de la lista.

✅ Ejercicio 4

Utilizando la clase genérica `Dictionary<K, V>` definida en `System.Collections.Generic`, implementa un sencillo programa de consola que pida nombres por teclado hasta que introduzcamos la cadena **"fin"**.

En ese momento mostraremos los nombres introducidos y cuantas veces se ha introducido cada uno. Para mostrar el resultado, deberás mostrar por un lado las claves y posteriormente el par clave valor. Para ello tendrás que recorre el diccionario 2 veces, una con un `foreach` para las claves y otra con un `foreach` para el diccionario obteniendo pares clave valor con la siguiente clase `KeyValuePair<K, V>`

💡 **Pista**:

Tienes que guardar los nombres **como clave** en el diccionario y el número de veces que se ha introducido **como valor**.

☑ Ejercicio 5

Crea una clase **`Polinomio`** que guarde los datos de un polinomio en un diccionario genérico ordenado.

$$9x^7 - 3x^3 - 7x + 5$$

Por ejemplo el polinomio se guardaría en una propiedad privada automática de forma equivalente a esta ...

```
private SortedDictionary<int, int> Monomios { get; set; }
// Guardando el polinomio en un diccionario de la siguiente forma ...
Monomios = new SortedDictionary<int, int>() { {0, 5}, {1, -7}, {3, -3}, {7, 9} };
```

Donde cada **monomio** (cX^e) se guardan en orden inverso, la "*clave*" es el **exponente e`** y el "*valor*" el **coeficiente c**.

El constructor de la clase polinomio lo recibirá **una cadena** de la siguiente forma: **`"9x7-3x3-7x+5"`** y su método string **`ToString()`** lo mostrará de forma análoga.

Define un método de clase estático y publico llamado **`Suma`** donde entren dos objetos polinomios en forma devuelva otro con el resultado de la suma de tal manera que si en el programa principal quiero hacer la siguiente suma ...

$$(9x^7 - 3x^3 - 7x + 5) + (4x^2 - 1) = 9x^7 - 3x^3 + 4x^2 - 7x + 4$$

Tendré que escribir el siguiente código ...

```
Polinomio suma = Polinomio.Suma(new Polinomio("9x7-3x3-7x+5"), new Polinomio("4x2-1"));
Console.WriteLine(suma);
```

y su ejecución mostrará ... **`9x7-3x3+4x2-7x+4`**

Recuerda que internamente tendremos en cada objeto polinomio una colección equivalente a las siguientes...

```
// polinomio 1
new SortedDictionary<int, int>(){{ 0, 5 }, { 1, -7 }, { 3, -3 }, { 7, 9 }};
// polinomio 2
new SortedDictionary<int, int>(){{ 0, -1 }, { 2, 4 }};
// suma
new SortedDictionary<int, int>() {{ 0, 4 }, { 1, -7 }, { 2, 4 }, { 3, -3 }, { 7, 9 }};
```

💡 **Pista**:

Para inicializar el diccionario con el polinomio en el constructor que recibe una cadena con el polinomio, habrá diferentes propuestas. Pero lo más lógico sería usar expresiones regulares.

¿Cómo extraer lo monomios de una cadena con un polinomio de entrada usando una expresión regular?

Lo más lógico es hacer un **match** de los diferentes `monomios` con la forma cx^e en la cadena de entrada.

Una propuesta de expresión regular sería la siguiente ...

1. Definimos un **grupo** para el **coeficiente** que puede estar (cx^e) o no (x^e). Además, si hay coeficiente este puede ser un signo solo o tener valor numérico $-x^e$ (**c = -1**), $-3x^e$ (**c = -3**), $+x^e$ (**c = 1**), $+3x^e$ o $3x^e$ (**c = 3**).
   Una posibilidad sería `string grupoCoeficiente = @"(?<grupoCoeficiente>[+-]?\d*)?";` aunque admita la cadena vacía nos valdría como simplificación.
2. Definimos un **grupo** para el **incógnita** que puede estar (cx^e) o no (c)
   Una posibilidad sería `string grupoIncognita = @"(?<grupoIncognita>[xX])?";`
   Su interpretación sería: Si no hay incógnita `x` o `X` entonces el exponente es 0 y en caso contrário es mayor que cero. (No admitiremos negativos).
3. Definimos un **grupo** para el **exponente** que puede estar (cx^e) o no (c, cx)
   Una posibilidad sería `string grupoExponente = @"(?<grupoExponente>[1-9]|\d*)?";`
   Su interpretación sería: Si no hay exponente entonces este puede ser 0 o 1 dependiendo de si hay incógnita o no y si lo hay tomará el valor entero del grupo.
4. Por tanto el patrón para ir buscando los monomios en la cadena e ir inicializando el polinomio quedaría ...
   ```
   string patronMonomio = $"{grupoCoeficiente}{grupoIncognita}{grupoExponente}";
   ```

🚀 **Ampliación:**

Redefine el operador `+` y úsalo en lugar de la función suma.

Ejercicio 6

Crea una clase llamada **DatosContacto** con los siguientes campos:

- El DNI de la persona (string)
- El nombre completo de la persona (string)
- La dirección completa de la persona (string)
- Telefono (string)

En la clase del programa principal:

- Crea un diccionario llamado `agendaContactos` con una clave de tipo string y el valor de la clase `DatosContacto`.
- Añade un método estático `CreaContactos` que devuelva el diccionario rellenado por el usuario, hasta que se introduzca un DNI vacío.
- Añade el método estático `BorraContacto` que reciba como parámetros un DNI (como cadena) y el diccionario. Borrará del mismo, el dato cuya clave coincida con el DNI que se le pasa.
- Añade un método estático `AñadeContacto` que reciba como parámetro un dato de tipo DatosContacto, y el diccionario con la agenda de contactos, y añada a la tabla la persona indicada.
- Crea `MuestraAgenda` que reciba como parámetro el diccionario y muestre su contenido.
- Crea un programa principal que te permita probar todos los métodos.

Nota: Como clave para cada contacto utiliza su DNI. Deberás controlar si las claves están en el diccionario anteriores de acceder a el, para evitar posteriores excepciones.