Tema 10.1

Descargar estos apuntes en pdf o html

Índice

- Colecciones en las BCL
- Colecciones con elementos enlazados en las BCL
  - Concepto de lista enlazada
    - El TAD Nodo básico
    - El TAD Lista Simple Enlazada
      - Añadir o insertar nodos
      - Moverse y buscar en una lista
      - Borrar nodos
  - El patrón iterador
    - Uso de el patrón iterador
    - Implementando el patrón iterador en la lista simplemente enlazada
  - Transformando y copiando colecciones
  - Lista vinculada LinkedList<T> definida en las BCL
    - Nodo doblemente enlazado LinkedListNode<T>
    - Operaciones implementadas en LinkedListNode<T>

Colecciones en las BCL

Una **colección** es un tipo de dato cuyos objetos almacenan otros objetos. Un ejemplo típico son las tablas, aunque en la BCL se incluyen muchas otras clases de colecciones que iremos viendo a lo largo de este tema. En las versiones recientes de C# las podemos encontrar en **System.Collections.Generic**.

Aunque las colecciones predefinidas incluidas en la BCL disponen de miembros propios con los que manipularlas, todas incluyen al menos los miembros de `ICollection<T>`. En realidad la interfaz `ICollection<T>` hereda de:

1. La interfaz `IEnumerable<T>` que permite que sean recorridas con la instrucción `foreach` usando el **patrón iterador**.
2. La interfaz `IClonable`, formada por un único método `object Clone()` que devuelve una copia del objeto sobre el que se aplica.

Algunas de las colecciones más utilizadas son las siguientes:

1. Las **listas** implementadas a través del tipo `List<T>` que serán equivalentes a los arrays. Esto es, permitirá accesos y modificaciones con complejidad `O(1)` a través del operador `[]` y un índice entero. Sin embargo, añadir y borrar elementos puede ser más costoso. Además, a diferencia de los Arrays tradicionales, las operaciones de añadir y borrar estarán encapsuladas al usuario para realizarse con la mayor eficiencia posible.
2. Las **tablas de dispersión** o '*Hash Tables*', implementadas a través del tipo `Dictionary<K, V>`. Son estructuras de datos que **asocian una clave al valor a guardar**.
   La diferencia con las Listas es que la clave de acceso y modificación a través del operador `[]` no tiene por que se solo un índice entero. Sino cualquier otro tipo de objeto que implemente el interfaz **IEquatable<T>**. Además, estas operaciones de acceso y modificación en ocasiones puntuales pueden tener una complejidad ligeramente superior a `O(1)`. Por esta razón la la clave es un entero siempre será algo más eficiente usar un tipo `List<T>`.
3. Otros tipos usados para resolver problemas o algoritmos con un conjunto de operaciones o métodos específicos como ...
   - Las **Pilas** (LIFO) acrónimo en ingles de *'**Último** en Entrar, **Primero** en Salir'*. Implementado a través del tipo **Stack<T>**
   - las **Colas** (FIFO) acrónimo en ingles de *'**Primero** en Entrar, **Primero** en Salir'*. Implementado a través del tipo **Queue<T>**
4. Las listas enlazadas implementadas a través del tipo **LinkedList<T>** al contrario que `List<T>` realizará inserciones y borrados con complejidad `O(1)` sin embargo será menos eficiente en accesos y modificaciones.
5. Colecciones ordenadas como ...
   - **SortedSet<T>** optimizado para guardar datos de forma ordenada y **sin duplicados** de forma eficiente.
     > **Nota:** Esta notación de `Set` o '*conjunto*' también se utiliza en otros lenguajes para denotar colecciones sin duplicados.
   - **SortedDictionary<TKey,TValue>** Serían diccionarios ordenados que optimizan las operaciones de inserción y eliminación.
   - **SortedList<TKey,TValue>** Serían diccionarios ordenados que optimizan memoria y operaciones de acceso y modificación.

> ✋ **Importante:** Como vemos hay muchas colecciones parecidas, donde la única diferencia es la '*eficiencia*' de un cierto tipo de operaciones sobre otras. Por eso, deberemos escoger cuidadosamente el tipo de colección dependiendo de las casuísticas que se nos puedan dar en nuestro programa.

Colecciones con elementos enlazados en las BCL

Concepto de lista enlazada

Vamos a ver el concepto de **lista enlazada** a través de su implementación más sencilla que son las **listas simplemente enlazadas**, para ahondar posteriormente en las **listas doblemente enlazadas** que son las que implementan la mayoría de lenguajes y veremos la razón de ello.

Para ello vamos a tratar una serie de estructuras que definiremos a continuación.

El TAD Nodo básico

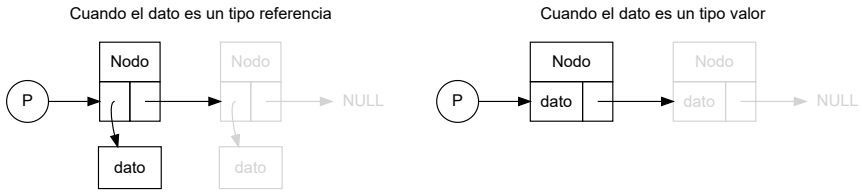
La forma que tenemos de **almacenar los datos** en las colecciones dinámicas en encapsulados en un TAD (Clase) denominada **Nodo**, que además de almacenar una propiedad con el dato, tendrá otra que referencie al siguiente Nodo.

La definición más común de Nodo será la siguiente:

```
public class Nodo<T> : IDisposable
{
    public Nodo<T>? Siguiente { get; set; }
    public T Dato { get; }

    public Nodo(T dato)
    {
        Siguiente = null;
        Dato = dato;
    }

    public void Dispose()
    {
        Siguiente = null;
        if (Dato is IDisposable d)
            d.Dispose();
    }
}
```



Representacines conceptual de un nodo en memoria

Así podremos ir añadiendo o quitando nodos según el número de elementos variable que tenga nuestra colección en un instante determinado de la ejecución.

El TAD Lista Simple Enlazada

El tipo, como mínimo, deberá guardar una **referencia al primer Nodo**. Aunque lo normal es que esté formada por una referencia al primer nodo, una referencia al último y una propiedad con la longitud actual de la lista.

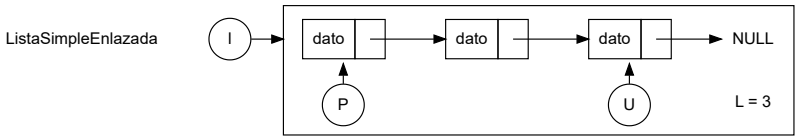
> ⚠️ **Aviso:** Es muy importante no perder la referencia a este primer nodo, sino perderemos toda referencia a la lista.

Una posible definición del tipo asociado al TAD podría ser la siguiente ....

```
class ListaSimpleEnlazada<T> where T : IComparable<T>
{
    public Nodo<T>? Primero { get; private set; }
    public Nodo<T>? Ultimo { get; private set; }
    public int Longitud { get; private set; }

    public ListaSimpleEnlazada()
    {
        Primero = Ultimo = null;
        Longitud = 0;
    }
}
```

> 📌 **Nota:** a la hora de representarla en memoria vamos a suponer que los nodos almacenan **tipos valor** para simplificar su representación.



Representación conceptual de la lista simplemente enlazada

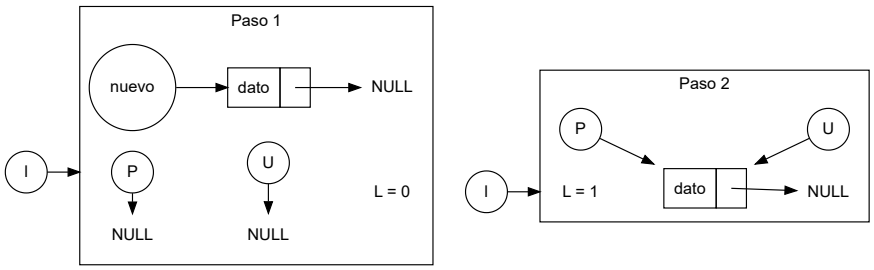
Dispondremos de las siguientes operaciones básicas:

Añadir o insertar nodos

Caso 1: Insertamos en una lista vacía

Este caso posiblemente se puede simplificar e incluirse en los `AñadeAlPrincipio` y `AñadeAlFinal` que trataremos después, sin embargo, lo hemos abordado por separado porque debemos tenerlo presente a la hora de añadir.

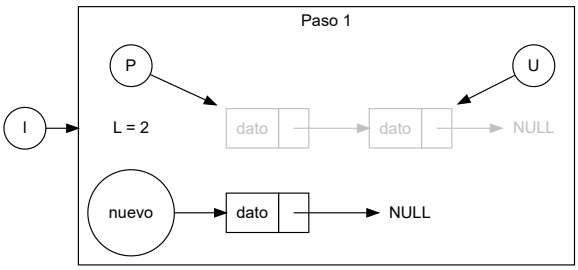
- **Paso 1:** Será siempre crear el nodo que vamos a insertar en la lista.
- **Paso 2:** Puesto que lista está vacía `Primero` y `Ultimo` apuntarán al nuevo nodo e incrementaremos `Longitud` en 1.



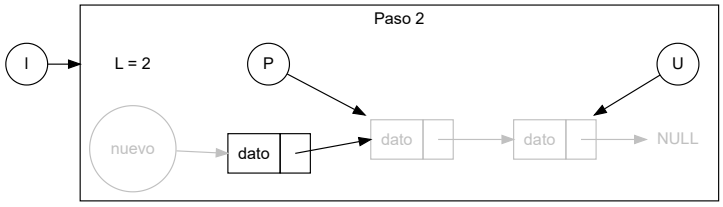

```
public void AñadeVacía(T dato)
{
    // Paso 1
    Nodo<T> nuevo = new Nodo<T>(dato);
    // Paso 2
    Primero = nuevo;
    Ultimo = nuevo;
    Longitud++;
}
```

Caso 2: Insertar al principio de la lista

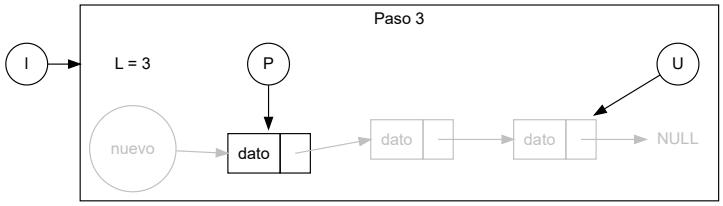
- **Paso 1**: Como en todos los casos de inserción, el primer paso será crear el nodo a insertar y que contendrá el dato que nos llega.



- **Paso 2**: Haremos que la propiedad `Siguente` del nuevo nodo referencie al **primer** elemento de la lista.



- **Paso 3**: Por último, esto es, en tercer lugar actualizaremos la propiedad `Primero` al nuevo nodo que acabamos de insertar.



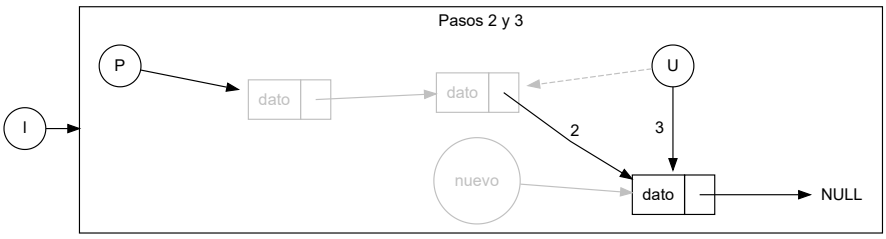

```
public void AñadeAlPrincipio(T dato)
{
    // Paso 1
    Nodo<T> n = new Nodo<T>(dato);
    // Paso 2
    n.Siguiente = Primero;
    // Paso 3
    Primero = n;
    Longitud++;
}
```

📌 **Nota:** Fíjate que si añadimos la línea 6, este método también funcionará si queremos añadir al principio de un lista vacía. Lo que nos podría '*ahorrar*' la implementación de la propiedad anterior.

```
    public void AñadeAlPrincipio(T dato)
    {
        Nodo<T> nuevo = new Nodo<T>(dato);
        nuevo.Siguiente = Primero;
        Primero = nuevo;
6       if (Longitud == 0) Ultimo = nuevo;
        Longitud++;
    }
```

Caso 3: Insertar al final de la lista

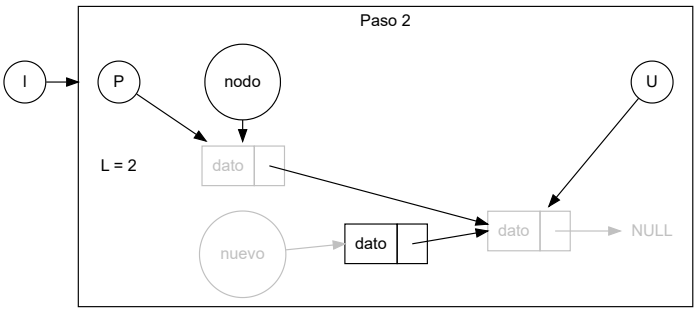
Será idéntico al caso anterior salvo que en el **paso 2** haremos que el `Siguiente` de el `Ultimo` nodo, referencie al **nuevo nodo** que acabamos de crear y posteriormente, en el **paso 3**, actualizaremos el `Ultimo` haciendo que referencie al nuevo nodo e incrementando la `Longitud` de la lista.

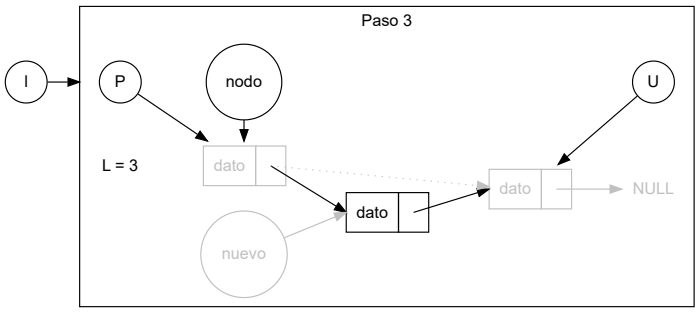



```
public void AñadeAlFinal(T dato)
{
    Nodo<T> nuevo = new Nodo<T>(dato); // Paso 1
    // Paso 2
    if (Longitud == 0)   // Como antes también podemos controlar si la lista
        Primero = nuevo; // está vacía pues no podremos acceder a Ultimo.Siguiente
    else
        Ultimo!.Siguiente = nuevo; // Último no puede ser null por L > 0
    // Paso 3
    Ultimo = nuevo;
    Longitud++;
}
```

Caso 4: Insertar en una posición determinada de la lista

En este caso al método método de inserción, además del dato a insertar, deberemos indicarle el nodo (posición) en la lista, después de el cual vamos a insertar el nuevo nodo. En la siguiente representación del proceso, vamos a representar con `A` la referencia l nodo anterior al que queremos insertar.





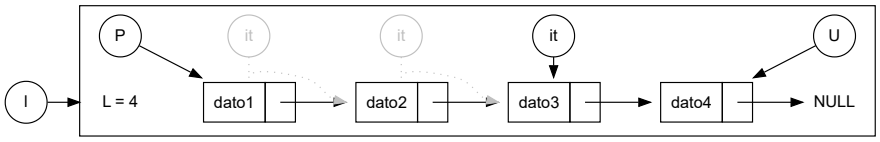
> 📌 **Nota:** Fíjate que si la posición que indicáramos para insertar fura la última, es como si hiciéramos un `AñadeAlFinal` . Si tenemos en cuenta esto y que la lista pueda estar vacía el método quedará.

```
    public void AñadeDespuesDe(Nodo<T> nodo, T dato)
    {
        // Precondiciones
4       if (Longitud == 0)
5           throw new ListaException("La lista no puede estar vacía.");
        Nodo<T> nuevo = new Nodo<T>(dato); // Paso 1
        // Paso 2
        nuevo.Siguiente = nodo.Siguiente;
        // Paso 3
        nodo.Siguiente = nuevo;
        // Si el nodo donde añadimos después es el último
        // el último pasará a ser eñ nuevo nodo.
        if (Ultimo == nodo) Ultimo = nuevo;
        Longitud++;
    }
```

Moverse y buscar en una lista

Los indizadores no es la mejor opción para recorrer listas. Para ello, dispondremos del **patrón iterador que trataremos más adelante en el tema** y nos permitirá recorrerlas con la instrucción **foreach**. Sin embargo, vamos a ver unos conceptos básicos que nos van a permitir recorrer el TAD en cualquier lenguaje.

1. Puesto que solo disponemos de un enlace simple entre nodos, deberemos empezar ha hacerlo desde el primer nodo.
2. Aparece pues la figura del **iterador** que tratará una referencia auxiliar a un nodo y me servirá para indicar una posición en la misma.
3. El recorrido será de forma secuencial hasta que el **iterador** sea `null`.



Recorriendo la lista simple con un iterador

Por ejemplo, si quisiéramos recorrer un objeto del tipo `ListaSimpleEnlazada` que estamos definiendo. Podríamos usar un esquema similar a este...

```
ListaSimpleEnlazada<int> l = new ListaSimpleEnlazada<int>();

l.AñadeAlFinal(9);
l.AñadeAlFinal(3);
l.AñadeAlFinal(6);

Nodo<int>? it = l.Primero;
while(it != null)
{
    Console.WriteLine(it.Dato);
    it = it.Siguiente;
}

// O el equivalente con for más simple....
for (Nodo<int>? it = l.Primero; it != null; it = it.Siguiente)
    Console.WriteLine(it.Dato);
```

Podemos aprovechar este recorrido para invalidar el `ToString()` en nuestro tipo lista con la siguiente propuesta ...

```
public override string ToString()
{
    // Tendremos en cuenta que la lista puede estar vacía.
    StringBuilder salida = new StringBuilder(Longitud > 0 ? "P" : "P -> NULL");

    for (Nodo<T>? it = Primero; it != null; it = it.Siguiente)
        salida.Append($" -> [{it.Dato}]");

    salida.Append($" <- U");

    return salida.ToString();
}
```

Borrar nodos

Vamos a implementarlo a través de una sola operación que recibirá una referencia a la referencia al nodo que queremos borrar, por ejemplo `public void Borra(Nodo<T> nodo)`.

La idea, es que el nodo que se recibe como referencia se desvincule de la misma y posteriormente nosotros tras la llamada a la misma decidiremos que hacer con él.

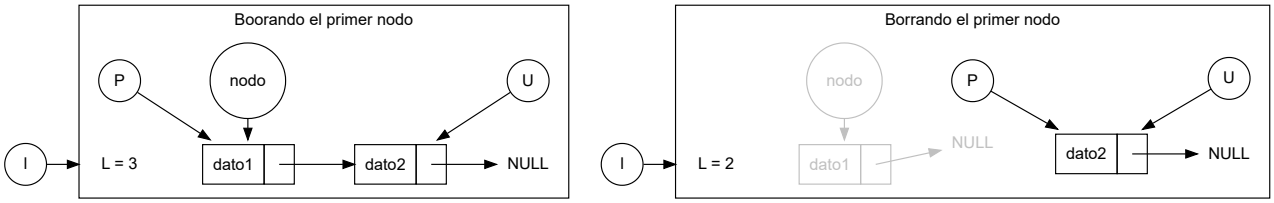
Como al añadir se podrán dar varios casos:

Caso 1: La lista tiene un único nodo

Puede suceder que al borrar el nodo, la lista se queda vacía y por tanto `Primero` y `Ultimo` deban establecerse a `null`.

Caso 2: Más de un nodo y borramos el primero

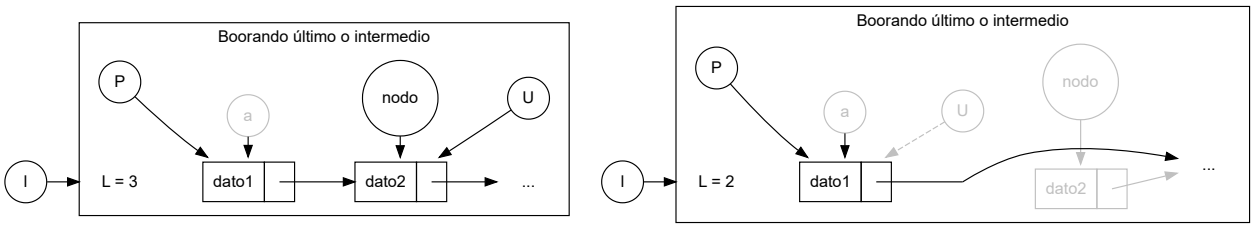
Si borramos el **primer nodo** de la lista, tendremos que actualizar la propiedad `Primero`, decrementar la propiedad `Longitud` y hacer que el `Siguiente` del nodo borrado referencie a `null`.



Caso 3: Más de un nodo y borramos el último o un nodo intermedio

> ✋ **Importante:** Si te fijas en la ilustración de abajo deberemos acceder al **nodo anterior** `a` al que queremos borrar. Sin embargo, esto no lo podemos hacer directamente, ya que no podemos acceder de forma simple al nodo anterior al que estamos borrando porque no tenemos un enlace directo. Deberemos pues, recorrer desde el principio de la lista, hasta llegar al nodo anterior junto al que queremos borrar y esto en el peor de los casos, hará que el borrado un coste de `O(n)` y esta no es la idea del uso de listas. Por lo que normalmente las listas implementadas en los lenguajes está doblemente enlazada como comentaremos más adelante.

Además, una vez tenemos una referencia al nodo anterior `a` haremos que su `Siguiente` sea igual al `Siguiente` del `nodo` que queremos borrar independientemente de si es `null` por ser el último o no. Y en el caso de que estemos borrando el último actualizaremos la propiedad `Ultimo` haciendo que referencie al nodo anterior `a`.



Como en el caso anterior, deberemos decrementar la propiedad `Longitud` y hacer que el `Siguiente` del nodo borrado referencie a `null`.

```
public void Borra(Nodo<T> nodo)
{
    // Precondición: La lista tenga algún nodo para borrar.
    if (Longitud == 0)
        throw new ListaException("Para borrar en una lista tiene que tener algún elemento.");
    // Caso 1: La lista tiene un único nodo.
    if (Longitud == 1)
    {
        // Aunque no sería necesario en este nivel de implementación
        // podemos comprobar que el nodo pertenezca a la lista.
        if (nodo != Primero)
            throw new ListaException("El nodo debe ser de la lista.");
        Primero = Ultimo = null;
    }
    // Caso 2: El nodo a borrar es el primero.
    else if (Primero == nodo)
    {
        Primero = nodo.Siguiente;
    }
    // Caso 3: El nodo a borrar es intermedio o el último.
    else
    {
        // Buscamos el nodo anterior al que queremos borrar, recorriendo la lista.
        // piensa esta operación haciéndote un pequeño diagrama de una lista
        // con varios nodos.
        Nodo<T>? a = null;
        for (Nodo<T>? n = Primero; n != null && n != nodo; n = n.Siguiente) a = n;
        if (a == null)
            throw new Exception("El nodo a borrar no pertenece a la lista.");
        a.Siguiente = nodo.Siguiente;
        if (Ultimo == nodo)
            Ultimo = a;
    }

    // Aislamos el nodo borrado del resto de la lista desligándolo del siguiente.
    nodo.Siguiente = null;
    // Nota: Sería más correcto hacer nodo.Dispose(); pero ...
    // dejaría el nodo inutilizable si lo quisiéramos
    // insertar en otra lista.

    // Decrementamos la longitud de la lista.
    Longitud--;
}
```

Normalmente dispondremos de un método/operación que vamos a denominar `Clear()` que eliminará todos los nodos de la lista dejándola vacía. Es importante desenlazar todos los nodos para que el *'recolector de basura'* los elimine cuanto antes de la memoria.

Una posible implementación podría ser la siguiente ...

```
public void Clear()
{
    while (Longitud > 0) // Mientras queden nodos en la lista
    {
        // Me quedo con la referencia al primero porque se actualizará tras su borrado
        // y ya no podríamos hacer el Dispose().
        Nodo<T> n = Primero!; // Se que no es null por L > 0
        // Borro el primero y lo aíslo.
        Borra(n);
        // Llamamos al dispose del nodo para que si el dato que contiene es IDisposable
        // se libere correctamente.
        n.Dispose();
    }
}
```

El patrón iterador

Es un **patrón de diseño orientado a objetos** ideado para recorrer cualquier colección de datos. Además, en el lenguaje C# cobra especial importancia para aplicar ciertos esquemas de **programación funcional** que trataremos en temas posteriores.

Este patrón se implementa en C# a través de la definición de la interfaz genérica `IEnumerable<T>` así como su antecesora `IEnumerable`. Ambas ofrecen un mecanismo para la iteración sobre los elementos de una **secuencia**, generalmente con la vista puesta en aplicar a esa secuencia la instrucción de programación `foreach` para recorrerla.

Dicho interfaz deberá ser implementada por la clase o tipo que implemente la secuencia o contenga una serie de datos que nos interese recorrer de forma secuencial.

La definición de `IEnumerable<T>` e `IEnumerable` es la siguiente:

```
// Definido dentro de System.Collections
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}

// Definido dentro de System.Collections.Generic
public interface IEnumerable<T> : IEnumerable
{
    IEnumerator<T> GetEnumerator();
}
```

> 📌 **Nota:** Fíjate que la interfaz se apoya en su contrapartida no genérica, para tener compatibilidad con la misma y por tanto hacia versiones anteriores de las BCL.

Como vemos, estas interfaces obligan a implementar un único método `GetEnumerator()`, que devuelve un objeto '*enumerador*' que implementa a su vez el interfaz o abstracción `IEnumerator<T>`. Dicho interfaz, nos asegura una serie de operaciones sobre un **iterador** que contendrá de forma encapsulada el objeto y el cual me permitirá recorrer datos de forma secuencial.

Por su parte, la interfaz `IEnumerator<T>` está definida de la siguiente forma:

> 📌 **Nota:** Como sucedía con `IEnumerable<T>`, la interfaz `IEnumerator<T>` se apoya en su contrapartida no genérica, para compatibilidad hacia atrás para redefinir de forma parametrizada la propiedad de solo lectura `Current`.

```
// Definido dentro de System.Collections
public interface IEnumerator
{
    object Current { get; }
    void Reset();
    bool MoveNext();
}

// Definido dentro de System.Collections.Generic
public interface IEnumerator<T> : IDisposable, IEnumerator
{
    T Current { get; }
}
```

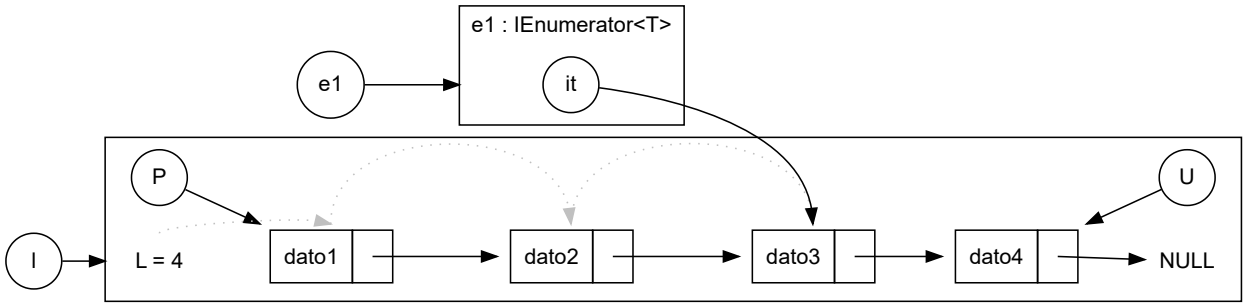
Como vemos en la definición, las clases que implementen este interfaz deben implementar los siguientes miembros:

- La propiedad `Current`, que devuelve el elemento actual apuntado por el iterador interno en la secuencia.
- El método `Reset()`, que establecerá el iterador a un estado inicial **justo antes del primer elemento**.
  > ✋ **Importante:** Este método, será llamado desde el constructor del objeto y el estado indicará en siguiente `MoveNext()` que debo ir al primer elemento.
- El método `MoveNext()`, que desplaza el enumerador al siguiente elemento de la secuencia devolviendo `true` si he podido hacerlo o `false` si no he podido avanzar porque ya estaba al final de la misma.

> ✋ **Importante:** Como hemos comentado antes, en la primera llamada tras el `Reset()` se situará al principio de la secuencia.

- El método `Dispose()`, restablece el iterador a su valor inicial y libera cualquier recurso no administrado asociado al mismo.

> 🔔 **Resumen:**
>
> Si nos fijamos en la ilustración de abajo:
>
> 1. La lista al implementar `IEnumerable<T>` contendrá un método **GetEnumerator()** que me devolverá un objeto que implementa `IEnumerable<T>`, por ejemplo `e1`, y que contendrá un iterador a los elemento de la lista, en nuestro caso los nodos, que a través de las operaciones `Reset()`, `MoveNext()` y la propiedad `Current` me permitirá ir recorriendo la colección de forma secuencial y acceder a su contenido.




```
IEnumerator<T> e1 = l.GetEnumerator();
e1.MoveNext();
e1.MoveNext();
e1.MoveNext();
e1.Current; \\ Se evaluará a dato3
```

Uso de el patrón iterador

Básicamente recorrer secuencias con la instrucción `foreach`, **pasar como parámetro o guardar** cualquier colección como la **generalización** secuencia recorrible `IEnumerable<T>`.

```
    int[] v = { 2, 3, 4, 5 };
2   // Cuando hacemos un foreach como este...
    foreach (int n in v) Console.Write($"{n} ");
4   // C# internamente realmente está ejecutando lo siguiente...
    IEnumerator e = v.GetEnumerator(); // El constructor hace el Reset();
    while (e.MoveNext()) Console.Write($"{e.Current} ");
```

> ✋ **Importrnte:** Pero ... ¿Por qué crear un objeto aparte para iterar y no dejar que las colecciones implementen directamente `IEnumerator<T>` ?

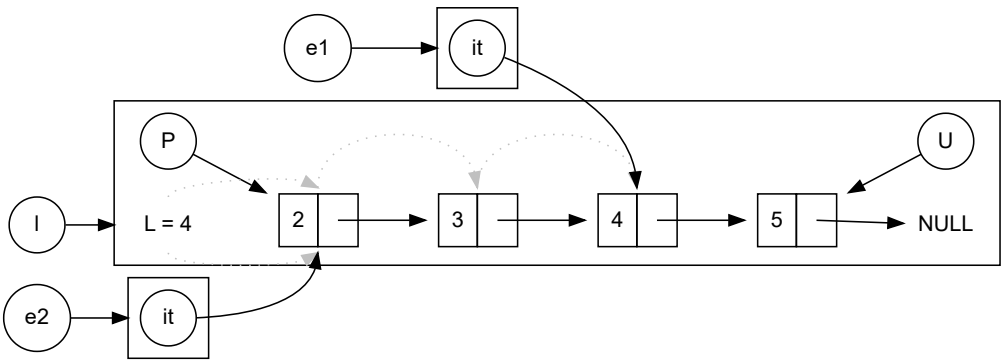
La respuesta tiene que ver con la necesidad de permitir la ejecución de **iteraciones anidadas sobre una misma secuencia**. Si la secuencia implementara directamente la interfaz enumeradora, solo se dispondría de un único '*estado de iteración*' en cada momento y sería imposible implementar bucles anidados `foreach` sobre una misma secuencia.

```
// Supongamos que nuestra lista simplemente enlazada implementa ya IEnumerable<T>
ListaSimpleEnlazada<int> l = new ListaSimpleEnlazada<int>();
l.AñadeAlFinal(2); l.AñadeAlFinal(3); l.AñadeAlFinal(4); l.AñadeAlFinal(5);

// La mejor opción es obtener el iterador en el ámbito donde lo voy a usar.
IEnumerator<int> e1 = l.GetEnumerator();
while (e1.MoveNext())
{
    IEnumerator<int> e2 = l.GetEnumerator();
    while (e2.MoveNext())
        Console.Write($"it1 = {e1.Current},  it2 = {e2.Current}");
}

// Es esta otra opción obtengo los dos iteradores al principio pero hago un
// reset al acabar la vuelta del segundo para empezar por el principio en
// futuras vueltas.
IEnumerator<int> e1 = l.GetEnumerator();
IEnumerator<int> e2 = l.GetEnumerator();
while (e1.MoveNext())
{
    while (e2.MoveNext())
        Console.Write($"it1 = {e1.Current},  it2 = {e2.Current}");
    e.Reset();
}
```

Si nos fijamos en la ilustración de abajo, el usar este patrón nos permite además mantener varios objetos enumeradores que me permiten iterar en la lista al mismo tiempo.



Implementando el patrón iterador en la lista simplemente enlazada

1. Lo primero será hacer que nuestra lista nos devuelva objetos que implementen `IEnumerator<T>` como en el ejemplo de arriba y para ello debe implementar `IEnumerable<T>` como vemos en la línea 2.

```
    class ListaSimpleEnlazada<T> :
2                       IEnumerable<T>
                        where T : IComparable<T>
    {
        ...

7       // Creo un objeto enumerador que defino a continuación en el punto 2
8       public IEnumerator<T> GetEnumerator() => new Enumerador(Primero);

        // Llamo al método de arriba que me hará la sustitución a la generalización.
        IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
    }
```

2. Deberemos definir una **clase anidada privada** dentro de `ListaSimpleEnlazada<T>` que cree los `IEnumerator<T>` que me permitirán iterar sobre la lista. Ala cual podemos llamar por ejemplo `Enumerador`.

```
    class ListaSimpleEnlazada<T> :
                        IEnumerable<T>
                        where T : IComparable<T>
    {
        ...

7       private class Enumerador : IEnumerator<T>, IDisposable
        {
            // Referencia al nodo actual sobre el que estamos iterando.
            // como se muestra en la ilustración de arriba.
            private Nodo<T>? It { get; set; }

            // Posición del primer elemento donde movernos después del reset.
            private Nodo<T>? Primero { get; set; }

            public Enumerador(Nodo<T>? primero)
            {
                Primero = primero;
19              Reset(); // Debe hacer el reset en el constructor.
            }
21          // El reset debe situar al iterador a un valor 'anterior' al Primero.
            public void Reset() => It = null;

            public T Current => It!.Dato;

            object IEnumerator.Current => Current;

28          public bool MoveNext()
            {
                // Antes de iterar confirmo si voy a poder hacerlo.
                bool puedoIterar = It == null && Primero != null || It != null && It.Siguiente != null;
                // Itero si puedo
                if (puedoIterar) It = It == null ? Primero : It.Siguiente;
34              return puedoIterar; // Retorno true si he podido iterar.
            }
            public void Dispose()
            {
                Primero = null;
                It = null;
            }
        }
    }
```

Transformando y copiando colecciones

Una vez ya conocemos el **patrón iterador** y que `IEnumetable<T>` es la abstracción de una secuencia recorrible secuencialmente que toda colección debería implementar. Vamos a ver cómo hacer copias de una colección y lo que es más importante. Veamos cómo transformar una secuencia en otra.

Lo más común en las BCL ers que las colecciones dispongan de un constructor que reciban una secuencia de datos a partir de la cual rellenar la colección. Para nuestra lista simplemente enlazada de ejemplo podría ser el siguiente ...

```
    class ListaSimpleEnlazada<T> where T : IComparable<T>
    {
        // ...

        public ListaSimpleEnlazada(IEnumerable<T> secuencia)
        {
            Primero = null;
            Ultimo = null;
            Longitud = 0;
10          // Añado a la lista los elementos de la secuencia de entrada.
            foreach (T dato in secuencia)
                AñadeAlFinal(dato);
        }
    }
```

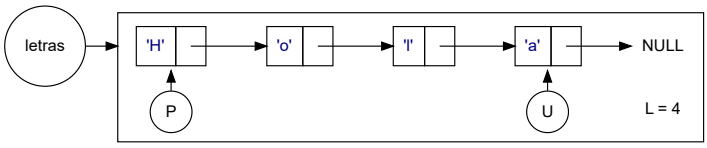
Ahora podríamos crear una lista simplemente enlazada de enteros a partir de un array con el nuevo constructor, ya que sabemos que cualquier colección en las BCL se puede transformar en una secuencia `IEnumerable<T>` ...

```
var l = new ListaSimpleEnlazada<int>(new []{2, 3, 4, 5});
```

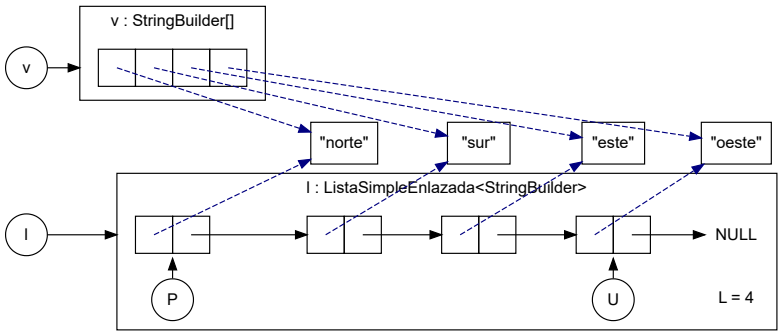
Pero también lo podríamos hacer con una cadena que es una secuencia de letras ....

```
var letras = new ListaSimpleEnlazada<char>("Hola");
```

Como sucedía con los arrays, si el contenido de la secuencia es un tipo referencia. Deberemos llevar cuidado con este tipo de constructores ya que ambas colecciones compartirán las referencias a los objetos. Por tanto si el objeto es mutable, como ocurre en el ejemplo de abajo, al modificarlo en una colección lo estaremos modificando en la otra.




```
var v = new StringBuilder[] {"norte", "sur", "este", "oeste"};
var l = new ListaSimpleEnlazada<StringBuilder>(v);
```



Lista vinculada LinkedList<T> definida en las BCL

Implementan el TAD (Tipo Abstracto de Datos) de programación clásico denominado: '**lista doblemente enlazada**'.

Mejorará las listas simplemente enlazadas que hemos visto para entender los conceptos por dos razones:

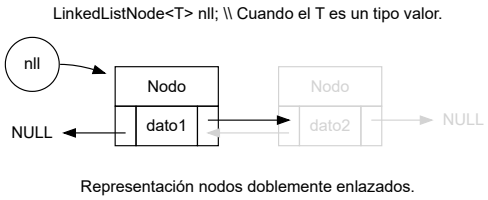
1. Me permiten recorrer la lista en ambas direcciones.
2. Los borrados de un nodo tienen coste `O(1)` respecto a los de la lista simplemente enlazada que hemos visto que tenías `O(n)`.
3. Podré insertar y borrar antes o después de un nodo.

Este tipo está definido de forma análoga en ostros lenguajes como Java o Kotlin con el mismo nombre `LinkedList<E>` y cómo comentamos en la introducción, es el tipo adecuado si voy a necesitar hacer muchas **inserciones** y **borrados** pues tienen un coste `O(1)`.

> 📌 **Nota:** No dispondrá de los métodos del interfaz de `IList` y **los accesos a través de indizador tienen complejidad** `O(n)`.

Nodo doblemente enlazado LinkedListNode<T>

La clase `LinkedListNode<T>` representará un **nodo de la lista** y al igual que en la definición de nuestro TAD, nos ayudará a interactuar con ella.

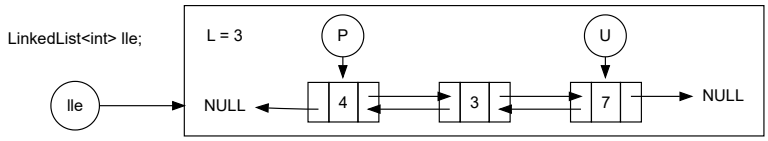


Representación nodos doblemente enlazados.

y por tanto la representación conceptual de un objeto de tipo LinkedList de enteros del código siguiente

```
LinkedList<int> lle = new LinkedList<int>(new int[]{4, 3, 7});
```

podría ser...



Operaciones implementadas en LinkedListNode<T>

Básicamente tendremos las mismas que en la lista simple que hemos implementado, más alguna adicional consecuencia del doble enlace.

1. **Constructores**

```
// Crear la lista vacía.
LinkedList<int> numeros = new LinkedList<int>();
// Crear la lista a partir de una secuencia de entrada.
LinkedList<int> numeros = new LinkedList<int>(new []{2, 3, 4, 5});
```

2. **Añadir o insertar datos al principio y al final**

```
    numeros.AddFirst(2);    // Inserta el 2 al principio
    numeros.AddLast(5);     // Inserta un 5 al final

    // Además de insertar un tipo de dato directamente.
    // Podemos insertar un nodo que contenga el dato.
    LinkedListNode<int> nodoAInsertar = new LinkedListNode<int>(8);
    numeros.AddLast(nodoAInsertar);
```

3. **Añadir o insertar datos en medio de la lista**

   A diferencia de las listas simplemente enlazadas donde solo podíamos insertar después de un determinado nodo de referencia. En `LinkedList<T>` podremos hacerlo también antes. Supongamos la siguiente lista de números...

```
LinkedList<int> numeros = new LinkedList<int>();
numeros.AddLast(2);
numeros.AddLast(5);
numeros.AddLast(8);
```

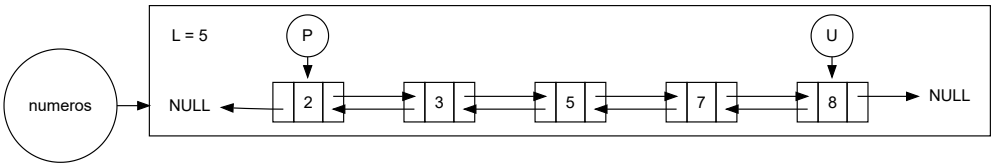
   Podremos iterar hasta donde queremos insertar o buscar directamente el dato donde queremos insertar con el método `LinkedListNode<T> linkedList.Find(T dato)` que me devolverá una referencia al nodo que contiene el dato o `null` si no lo encuentra. Por ejemplo si quisiéramos **insertar un 3 antes del 5** haríamos...

```
LinkedListNode<int> posDel5 = numeros.Find(5);
if (posDel5 != null)
    numeros.AddBefore(posDel5, 3); // Inserta antes del nodo posDel5
```

   También podremos **insertar después** de un determinado nodo y al igual que en el caso de inserción al principio y al final, **también podremos pasar el nodo a insertar en lugar del dato**. Por ejemplo...

```
LinkedListNode<int> posDel5 = numeros.Find(5);
if (posDel5 != null)
    numeros.AddAfter(posDel5, new LinkedListNode<int>(7)); // Inserta después del nodo posDel5
```

   Tras la operaciones anteriores la lista debería tener los siguiente datos...



4. **Propiedades básicas**

   Como en el tad que definimos `LinkedListNode<T>` nos ofrecerá una serie de propiedades para saber la longitud de la lista y acceder directamente al primer y último nodo. Por ejemplo para la lista anterior tendremos ...

```
Console.WriteLine(numeros.Count);       // Mostrará el número de nodos = 5

Console.WriteLine(numeros.First.Value); // Mostrará 2
Console.WriteLine(numeros.Last.Value);  // Mostrará 8
```

5. **Borrado de datos**

   Puedo borrar nodos de forma inmediata con `RemoveFirst()` y `RemoveLast()` pero si quiero concretar el nodo a borrar deberé indicárselo al método de borrado `void Remove(LinkedListNode<T> node)`.
   Si nos fijamos, una vez lo borramos `node` aún seguirá apuntando a ún nodo que no pertenece ya a la lista y que su siguiente y anterior debería ser null. La clase deja en manos del programador la decisión de que hacer con el mismo, como por ejemplo insertarlo en otra posición de la misma lista. Veamos un ejemplo con varias opciones de borrado.

```
int dato = 5;
LinkedListNode<int> posDel5 = numeros.Find(dato);

// Lo borramos y lo volvemos a insertar al principio de la lista.
numeros.Remove(posDel5);
numeros.AddFirst(posDel5);

// Lo borramos y ya no lo vamos a usar.
numeros.Remove(posDel5);
posDel5 = null;

// El caso anterior equivaldría a borrarlo directamente
// pasando el valor de la lista que queremos borrar.
bool borrado = numeros.Remove(dato);
```

6. **Recorriendo un `LinkedList<T>`**

   Al implementar el interfaz `IEnumerable<T>`. Podremos recorrer los datos con un `foreach` de la siguiente manera.

```
foreach (int numero in numeros)
    Console.WriteLine(numero);

// Creando una instáncia de un iterador directamente.
IEnumerator<int> e = numeros.GetEnumerator();
while(e.MoveNext())
    Console.WriteLine(e.Current);
```

   También podremos recorrerla en ambos sentidos **con un nodo a modo de iterador**.

```
for (var it = numeros.First; it!=null; it = it.Next)
    Console.WriteLine(it.Value);

// Recorrido inverso.
for (var it = numeros.Last; it!=null; it = it.Previous)
    Console.WriteLine(it.Value);
```

7. **Transformando la lista en un array**

   Ya hemos visto que el constructor me permite crear una lista a partir de una array. Pero, ¿Cómo transformamos de nuevo la lista en un array si hemps terminado de hacer inserciones y borrados?

   Dispondremos del método `void CopyTo(T[] array, int index)`.

```
int[] numeros = {2, 5, 8};
LinkedList<int> llnumeros = new LinkedList<int>(numeros);

// Realizamos las inserciones y borrados en llnumeros ...

numeros = new int[llnumeros.Count];
llnumeros.CopyTo(numeros, 0);
```