# ENUMERACIONES Y COLECCIONES EN C#

# COLECCIONES EN LAS BCL

## ¿Dónde Están Definidas?

- En las versiones 1.0 y 1.1 del Framework las podemos encontrar en:
  - System.Collections
  - System.Collections.Specialized

- A partir de la versión 2.0 usaremos las de …
  - System.Collections.Generic

- Todas incluyen al menos los miembros de ICollection<T>. En realidad la interfaz ICollection hereda de la interfaz IEnumerable<T> en que se basa la instrucción foreach para poder recorrerlas.
- Más adelante hablaremos de este interfaz.

# Colecciones En Las BCL Implemetadas De Forma VINCULADA O ENLAZADA

# COLECCIONES EN LAS BCL IMPLEMETADAS DE FORMA VINCULADA

## Características Y Tipos En Las BCL

- Implementación de forma vinculada
- Deponemos de la clase `LinkedList<T>` a partir de la v2.0
- No disponemos de los métodos del interfaz de IList y los accesos a través de indizador no tienen complejidad 1.
- Disponemos de la clase `LinkedListNode<T>` que representará un nodo de la lista y nos ayudará a interactuar con ella.
- Me permitirá hacer inserciones y borrados de complejidad 1.

## Resumen

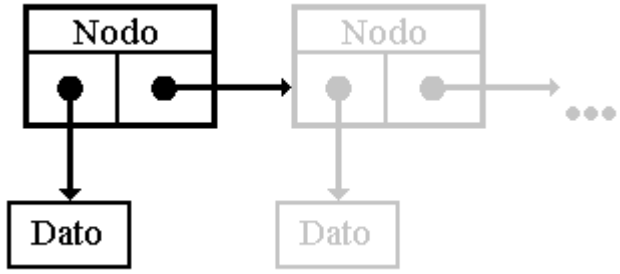
- Usaré `LinkedList<T>` cuando vaya a realizar muchas inserciones al principio y en mitad de la lista.
- Utilizaré `List<T>` cuando sólo inserte al final y además me interese mucho acceder rápidamente a una posición determinada.

# IMPLEMETACIÓN DE TADS VINCULADOS

## 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 un atributo con el dato, tendrá otro que referencie a otro Nodo.
- La definición más común de Nodo será la siguiente:

```
class Nodo<T>
{
   T dato;
   Nodo<T> siguiente;
}
```

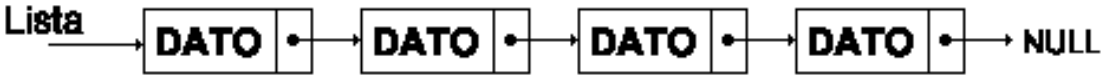



- Así podremos ir añadiendo o quitando nodos según el número de elementos variable que tenga nuestra colección en un instante determinado de la ejecución.

# LISTA ENLAZADA BÁSICA

- Es la forma más simple de estructura dinámica.
- La Lista, como mínimo, deberá guardar una referencia al primer NODO.

```
class ListaEnlazada<T>
{
   Nodo<T> primero;
   Nodo<T> ultimo;
   int longitud;
}
```




- Es muy importante no perder el valor a este primer nodo, sino perderemos toda referencia a la lista.
- Dispondremos de las siguientes operaciones básicas:
  - Añadir o insertar nodos.
  - Borrar nodos.
  - Moverse por los nodos.
  - Buscar o localizar nodos.

# LISTA ENLAZADA BÁSICA

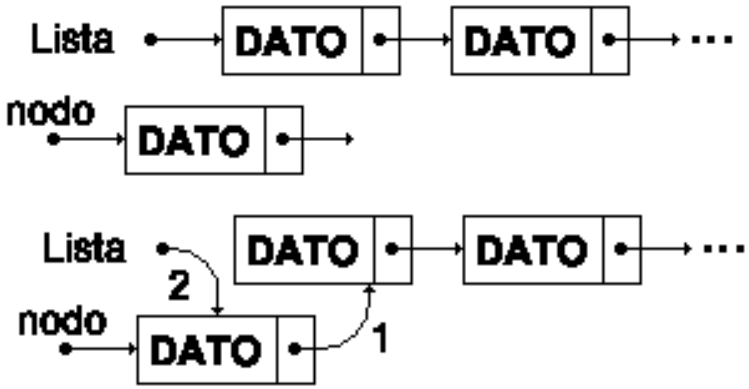
## Añadir O Insertar Nodos - I

- Se nos pueden dar 4 casos:
  - Caso 1: Insertamos en una lista vacía.

```
public void Añade(T dato)
{
   Nodo<T> nodo = new Nodo<T>(dato);
   primero = nodo;
   ultimo = nodo;
}
```


  - Caso 2: Insertamos al principio de una lista.

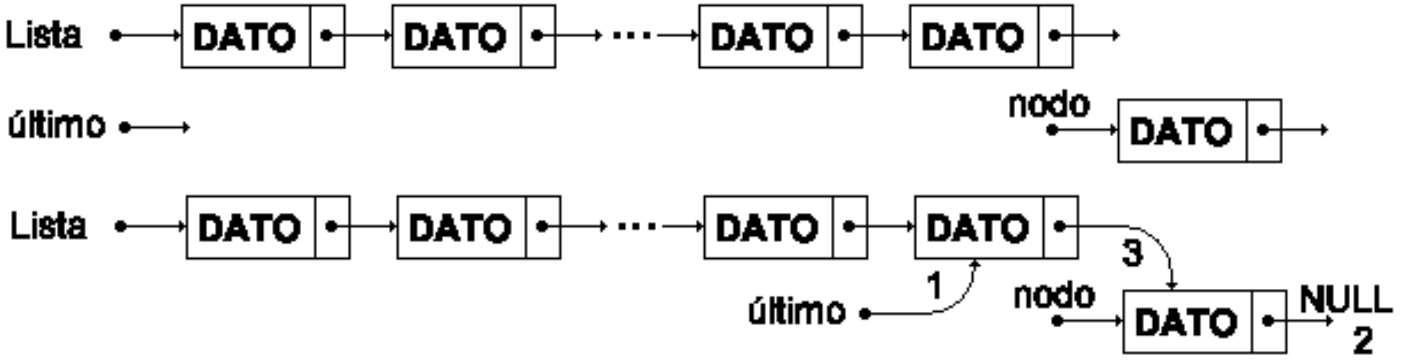
```
public void AñadeAlPrincipio(T data)
{
   Nodo<T> nodo = new Nodo<T>(dato);
   nodo.siguiente = primero; // 1
   primero = nodo;           // 2
}
```



# LISTA ENLAZADA BÁSICA

## Añadir O Insertar Nodos - II

- Caso 3: Insertamos al final de una lista.



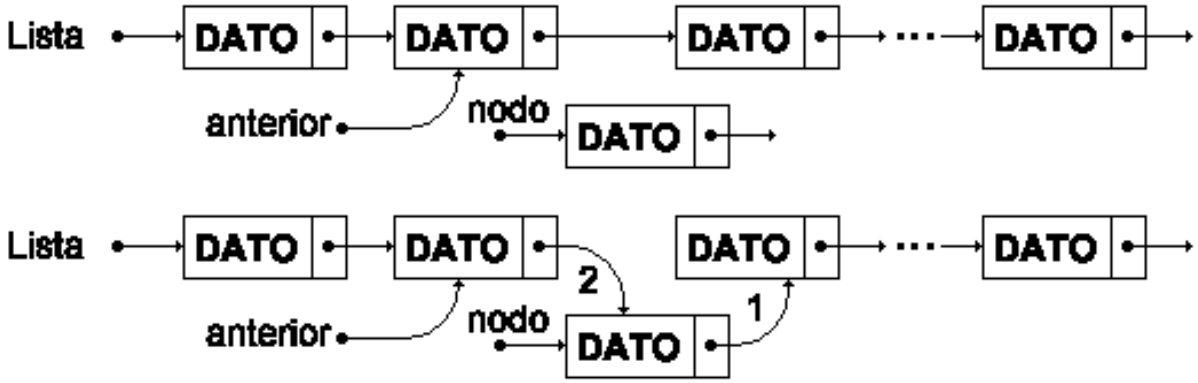

```
public void AñadeAlFinal(T data)
{
   Nodo<T> nodo = new Nodo<T>(dato);
   ultimo.siguiente = nodo;  // 2
   ultimo = nodo;            // 3
}
```

- Tanto si añadimos al final como al principio deberemos tener en cuenta que la lista esté vacía.

# LISTA ENLAZADA

## Añadir O Insertar Nodos - III

- Caso 4: Insertamos en mitad de una lista.



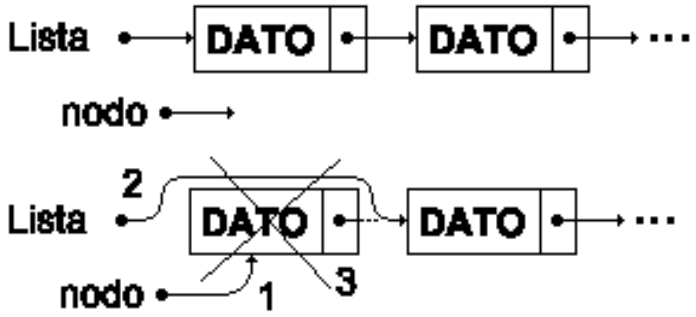

```
public void AñadeEnPosicion(T data, Nodo<T> anterior)
{
   Nodo<T> nodo = new Nodo<T>(dato);
   nodo.siguiente = anterior.siguiente;  // 1
   anterior.siguiente = nodo;            // 2
}
```

- Si añadimos en medio, la lista deberá tener al menos un elemento.

# LISTA ENLAZADA

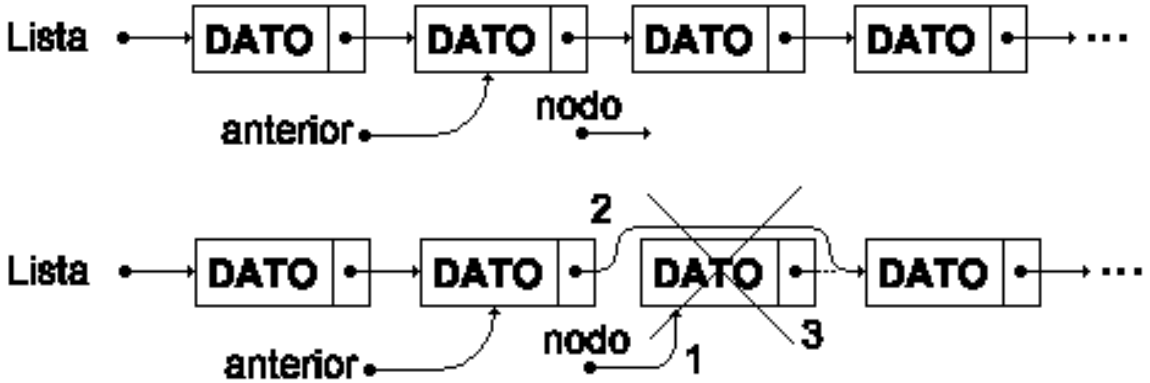
## Borrar Nodos

- Podremos tener 2 casos …
  - Caso 1: Borrar nodo del principio:




```
public void BorrarPrimero()
{
    Nodo<T> nodo = primero;
    primero = primero.siguiente;
    // Esto podría ir en una método
    // dispose de nodo.
    nodo.dato = null;
    nodo.siguiente = null;
}
```

  - Caso 2: Borrado resto de nodos.



# LISTA ENLAZADA BÁSICA

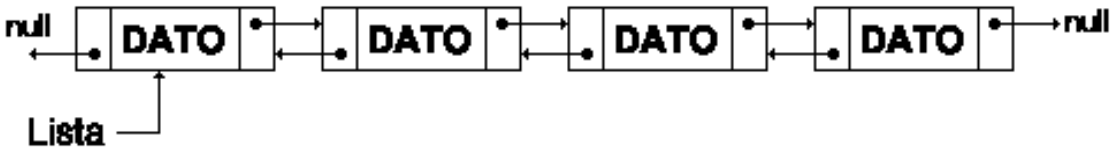
## Buscar Y Moverse Por Los Nodos

- Deberemos empezar siempre desde el principio de la listas.
- Deberemos recorrerla de forma secuencial hasta que el puntero al siguiente sea NULL.
- Aparece pues la figura del iterador que será una referencia auxiliar a un nodo, que me servirá para indicar una posición en la lista.
- Si queremos que nuestra colección de ejemplo se pueda recorrer con un foreach tendrá que implementar el interfaz IEnumerable o podremos recorrerla con un for y un iterador de la siguiente manera…

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

// Añadimos los nodos que consideremos oportunos.
for (Nodo<int> it = l.Primero; it != null; it = it.Siguinete)
{
 …
}
```

# LISTA DOBLEMENTE ENLAZADA (CASO MÁS GENERAL)

- La tendremos implementada en las BCL a través del tipo LinkedList<T>
  y las referencias a sus nodos serán objetos de la clase LinkedListNode<T>
- Cada nodo referenciará al siguiente y al anterior en la lista.
  - Para el primer nodo anterior = null
  - Para el último nodo siguiente = null

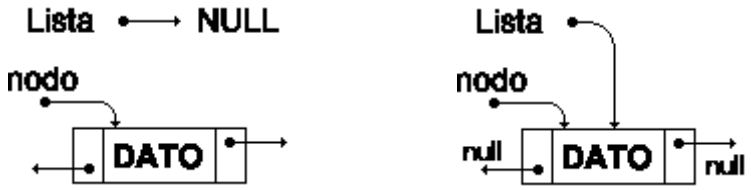



- Dispondremos de las mismas operaciones básicas que en las listas abiertas, pero con las siguientes mejoras.
  - Podremos movernos fácilmente al nodo anterior.
  - Podremos recorrer fácilmente la lista a la inversa.

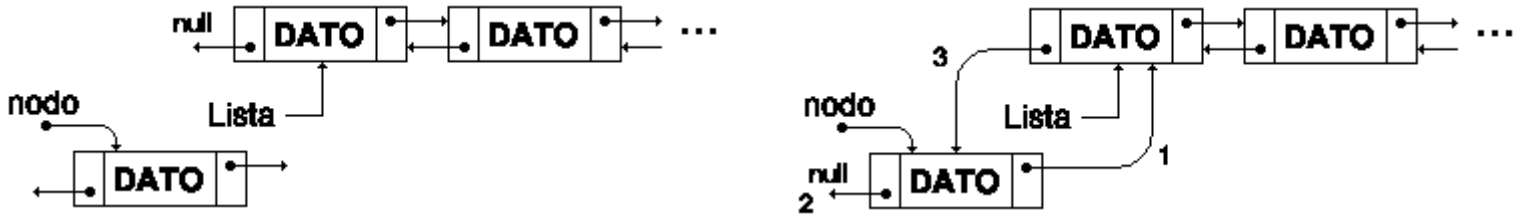
# LISTA DOBLEMENTE ENLAZADA (CASO MÁS GENERAL)

## Añadir O Insertar Nodos – I

- Se nos pueden dar 4 casos:
  - Caso 1: Insertamos en una lista vacía.




  - Caso 2: Insertamos al principio de una lista.

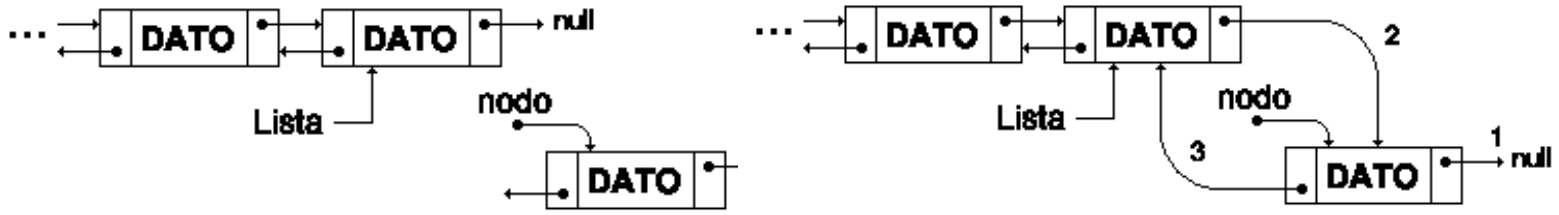



```
LinkedList<int> numeros = new LinkedList<int>();
numeros.AddFirst(2);
```

# LISTA DOBLEMENTE ENLAZADA (CASO MÁS GENERAL)

## Añadir O Insertar Nodos – II

- Caso 3: Insertamos al final de una lista.

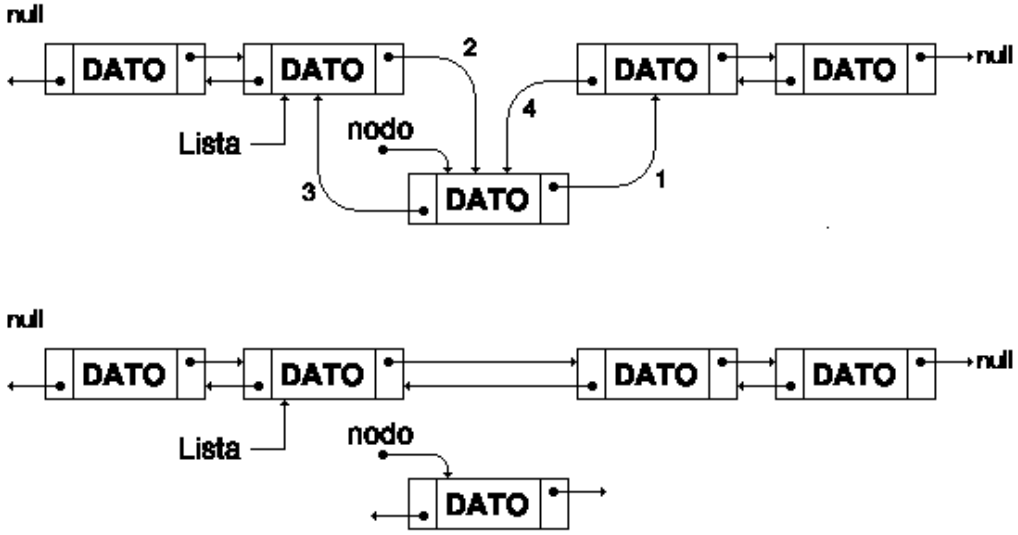



```
LinkedList<int> numeros = new LinkedList<int>();

numeros.AddFirst(2);
numeros.AddLast(5);

// Además de insertar un darto directamente. Podemos insertar un nodo
// que contenga el dato.

LinkedListNode<int> nodoAInsertar = new LinkedListNode<int>(8);
numeros.AddLast(nodoAInsertar);
```

# LISTA DOBLEMENTE ENLAZADA (CASO MÁS GENERAL)

## Añadir O Insertar Nodos - II

- Caso 4: Insertamos en mitad de una lista.



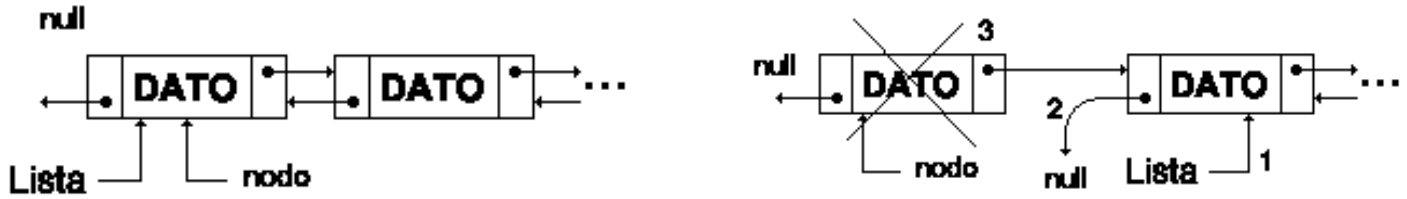

- El ejemplo de arriba representa la inserción después (before) del iterador lista, pero de forma análoga se puede hacer la inserción antes del iterador.

```
LinkedList<int> numeros = new LinkedList<int>();
numeros.AddFirst(2);
numeros.AddLast(5);
numeros.AddLast(8);
// Podemos insertar directamente el dato un nodo con el dato.
LinkedListNode<int> posDel5 = numeros.Find(5);
numeros.AddBefore(posDel5, 3);
numeros.AddAfter(posDel5, new LinkedListNode<int>(7));
```

# LISTA DOBLEMENTE ENLAZADA (CASO MÁS GENERAL)

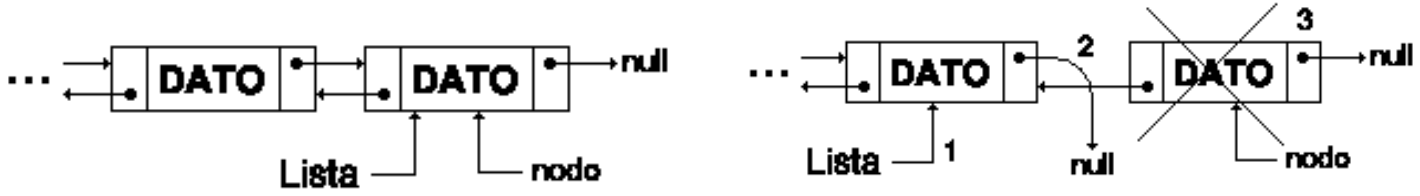
## Borrar Nodos - I

- Se nos pueden dar 3 casos:
  - Caso 1: Borrado nodos del principio de la lista:




```
numeros.RemoveFirst();
```

  - Caso 2: Borrado nodos del final de la lista:

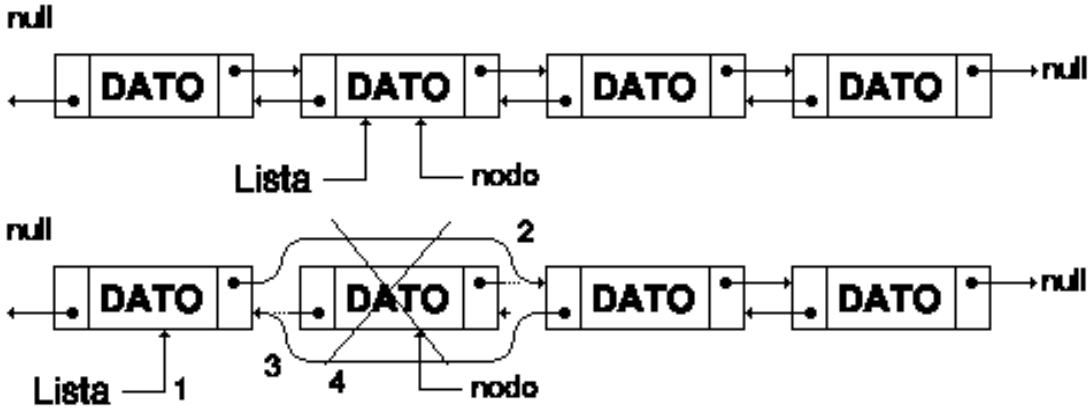



```
numeros.RemoveLast();
```

# LISTA DOBLEMENTE ENLAZADA (CASO MÁS GENERAL)

## Borrar Nodos - II

- Caso 3: Borrado nodos del intermedios de la lista




```
LinkedListNode<int> posDel5 = numeros.Find(5);
numeros.Remove(pos5);
// pos5Del ya no se debería usar porque es una referencia a un nodo
// que ya no está en a lista. Aunque esta misma referencia a nodo
// podríamos insertarla en otra posición de la misma solo si la hemos
// borrado con …
numeros.AddFirst(pos5Del);
posDel5 = null;
// También puedo borrar directamente pasando el valor.
numeros.Remove(8);
```

# LISTA DOBLEMENTE ENLAZADA (CASO MÁS GENERAL)

## Recorrido

- Al implementar el interfaz IEnumerable . Podremos recorrer los datos con un foreach.

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

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

```
for (LinkedListNode<int> it = numeros.First; it!=null; it = it.Next)
    Console.WriteLine(it.Value);
```

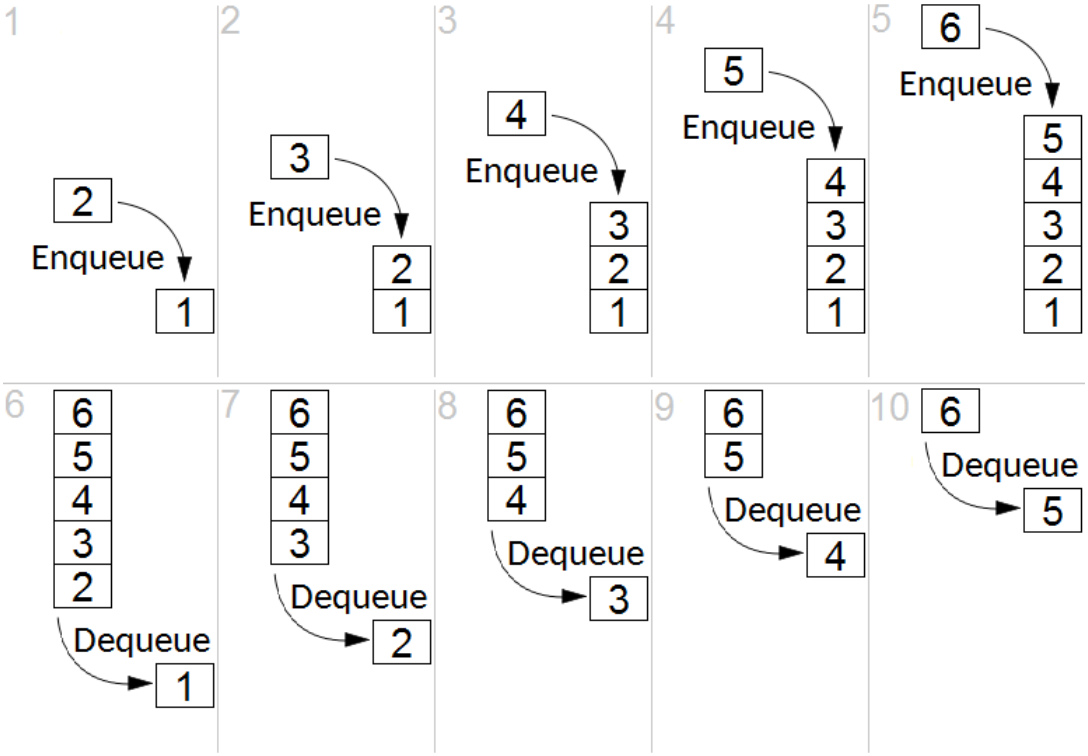
## Recorrido En Orden Inverso.

```
for (LinkedListNode<int> it = numeros.Last; it!=null; it = it.Previous)
    Console.WriteLine(it.Value);
```

# EL TAD COLA (QUEUE)

## Definición

- Los elementos se añaden por el final y se suprimen por el principio denominado frente de la cola.
- También se les conoce como estructuras FIFO, acrónimo en ingles de Primero en Entrar, Primero en Salir.
- Podremos implementar el TAD Cola de forma vinculada o con una tabla. Pero su funcionalidad será la misma.



# EL TAD COLA (QUEUE)

## Implementación En Las BCL

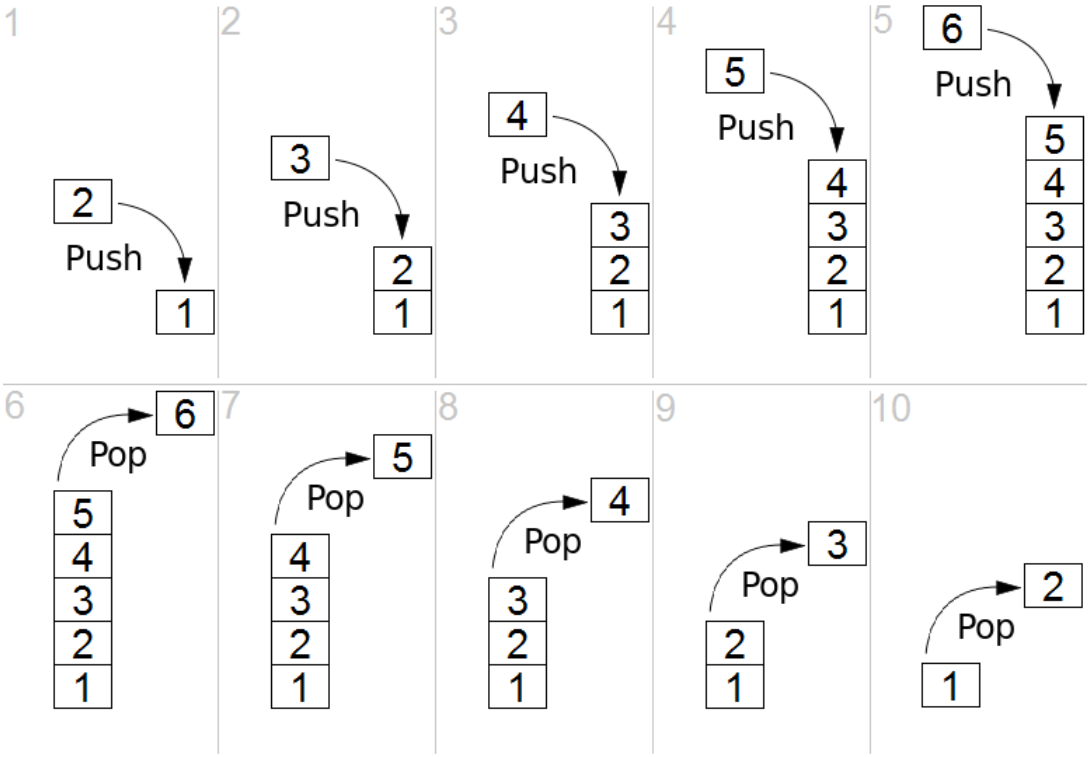
- En las BCL está implementado como tabla y no de forma vinculada.
- Las operaciones básicas son:
- Encolar (Enqueue).
- Desencolar (Dequeue).
- Coger siguiente sin desencolar (Peek).
- Clear (Borrar toda la cola).
- Vacía (Count == 0).

```
Queue<int> cola = new Queue<int>();

for (int i = 1; i <= 6; i++)
    cola.Enqueue(i);

while (cola.Count > 0)
    Console.WriteLine(cola.Dequeue());
```

# EL TAD PILA (STACK)

## Definición

- Los elementos se añaden y extraen por el mismo extremo que denominaremos cabeza de la pila.
- También se les conoce como estructuras LIFO, acrónimo en ingles de Último en Entrar, Primero en Salir.
- Podremos implementar el TAD Pila de forma vinculada o con una tabla. Pero su funcionalidad será la misma.



# EL TAD PILA (STACK)

## Implementación En Las BCL

- En las BCL está implementado como tabla y no de forma vinculada.
- Las operaciones básicas son:
- Apilar (Push).
- Desapilar (Pop).
- Coger siguiente sin desapilar (Peek).
- Clear (Borrar toda la pila).
- Vacía (Count == 0).

```
Stack<int> pila = new Stack<int>();

for (int i = 1; i <= 6; i++)
    pila.Push(i);

while (pila.Count > 0)
    Console.WriteLine(pila.Pop());
```

# Recorer Colecciones
# Patrón Iterador

# RECORER O ITERAR EN SECUENCIAS Y COLECCIONES

## Patrón IEnumerable, IEnumerable<T>

- Tanto la interfaz genérica IEnumerable<T> como su antecesor IEnumerable, ofrecen un mecanismo para la iteración sobre los elementos de una secuencia, generalmente con la vista puesta en aplicar a esa secuencia el patrón de programación foreach.

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

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

- 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.

# RECORER O ITERAR EN SECUENCIAS Y COLECCIONES

## Patrón IEnumerable, IEnumerable<T>

- Estas interfaces obligan a implementar un único método GetEnumerator(), que devuelve un objeto enumerador. Este objeto contendrá un iterador que me permitirá recorrer datos secuencialmente.
- Por tanto definiremos una clase anidada privada, que implemente el interfaz de este objeto enumerador IEnumerator<T>.

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

```
// System.Collections
public interface Ienumerator {
    object Current { get; }
    void Reset();
    bool MoveNext();
}
// System.Collections.Generic
public interface IEnumerator<T> : IDisposable, Ienumerator {
    T Current { get; }
}
```

- Como sucedía con IEnumerable<T>, la interfaz IEnumerator<T> se apoya en su contrapartida no genérica.

# RECORER O ITERAR EN SECUENCIAS Y COLECCIONES

## Patrón IEnumerable, IEnumerable<T>

En conjunto, IEnumerator<T>debe implementar los siguientes miembros:

- La propiedad Current, que devuelve el elemento actual de la secuencia.
- El método Reset(), que restablece la enumeración a su valor inicial.
- El método MoveNext(), que desplaza el enumerador al siguiente elemento de la secuencia.
  **En la primera llamada se situará al principio de la secuencia.**
  **Devuelve true si se puede avanzar y false si no se puede.**
- El método Dispose(), restablece la enumeración a su valor inicial y libera cualquier recurso no administrado asociado al enumerador.

El código foreach del ejemplo, internamente hará las llamadas de la derecha.

```
int[] v = { 2, 3, 4, 5 };

foreach (int n in v)
    Console.Write($"{n} ");
```

```
int[] v = { 2, 3, 4, 5 };
IEnumerator it = v.GetEnumerator();
// El constructor hace el Reset();
while (it.MoveNext())
    Console.Write($"{it.Current} ");
```

# RECORER O ITERAR EN SECUENCIAS Y COLECCIONES

## Patrón IEnumerable, IEnumerable<T>

¿Por qué esta separación en dos niveles, en la que básicamente IEnumerable<T> es de un nivel más alto, mientras que IEnumerator<T> se encarga del "trabajo sucio"?

¿Por qué 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 "estado de iteración" en cada momento y sería imposible implementar bucles anidados sobre una misma secuencia.
- La clase que implemente IEnumerator<T> deberá encargarse de mantener el estado del iterador para garantizar que los métodos de la interfaz funcionen correctamente.

# Colecciones En C# Implementadas Como TABLAS

# COLECCIONES EN C# IMPLEMENTADAS COMO TABLAS

## Tipos Más Usados

- Tenemos las listas que implementan el interfaz IList
  - Entre ellas se encuentran los Arrays tradicionales, sólo que tendremos restricciones de añadir, remover, etc…
  - ArrayList aparece en la v1.0 y implementa IList completo permitiendo Añadir y Borrar e incluyendo métodos como BinarySearch y Sort que a diferencia de sus hermanos en los arrays no son estáticos.
  - List<T> es su equivalente genérico a ArrayList y aparece a partir de la v2.0, la utilizaremos en su lugar por ser más robusta en la restricción de tipos.
- Otros tipos:
  - Las Pilas Stack o Stack<T> y las Colas Queue o Queue<T>. También se implementan como tablas sólo que en las colas podemos establecer un factor de crecimiento al añadir.

# COLECCIONES EN C# IMPLEMENTADAS COMO TABLAS

## Tabla Hash O Dispersión

- Una tabla hash o mapa hash es una estructura de datos que asocia llaves o claves con valores. La operación principal que soporta de manera eficiente es la búsqueda.

- Resumiendo podemos decir que lo que utiliza para indexar un elemento en la colección no es un entero, sino una cadena o cualquier otro objeto.

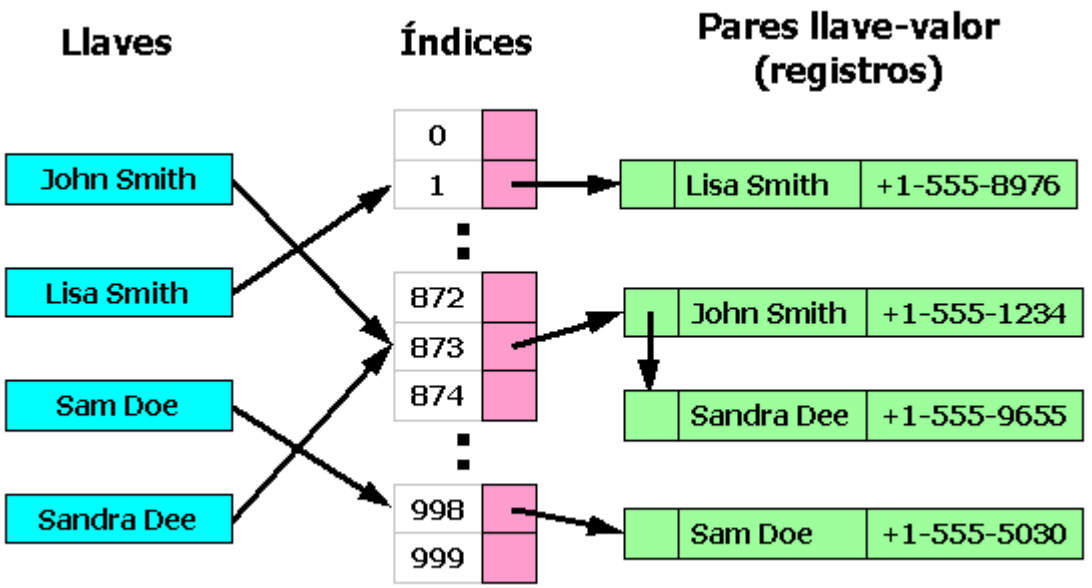
- Las tablas hash se suelen implementar sobre arrays de una dimensión, aunque se pueden hacer implementaciones multi-dimensionales basadas en varias claves.

Veamos el proceso…

# COLECCIONES EN C# IMPLEMENTADAS COMO TABLAS

## Esquema De Almacenamiento De Una Tabla Hash

1. Supongamos que la Llave (índice o key) es un nombre y el Valor (value) guardado es un teléfono.
2. Definimos una tabla de tamaño fijo ej. 1000
3. Para meter un teléfono en la clave nombre. Calculamos su hash, del que a su vez calculamos el módulo.
   C# usará el método sobreescribible de la clase Object int GetHashCode().
4. Nos vamos a esa posición del array y añadimos el par clave valor a una lista de claves con el mismo módulo.




Un proceso idéntico realizaremos para acceder.

# COLECCIONES EN C# IMPLEMENTADAS COMO TABLAS

## ¿Qué Implementación De Tabla De Dispersión Tendremos En Las BCL?

- En la v1.0 disponemos de HashTable, en desuso.

- A partir de la v2.0 Disponemos de…
  - Disctionary<TClave, Tvalor> tabla has parametrizada.
  - SortedDisctionary<TClave, TValor>

- Cada elemento del diccionario serán objetos de la clase:

KeyValuePair<TClave, TValor>

- Podré usar un indizador con la clave.

# COLECCIONES EN C# IMPLEMENTADAS COMO TABLAS

## Métodos Principales

- Para seguir nuestros ejemplos vamos a suponer la implementación de un tipo referencia fecha propio que hemos ido usando en otros ejercicios. Por ejemplo:

```
class Fecha : IComparable<Fecha>, ICloneable
{
    public int Dia { get; private set; }
    public int Mes { get; private set; }
    public int Año { get; private set; }

    public Fecha(int dia, int mes, int año)  {
        Dia = dia; Mes = mes; Año = año;
        try {
            DateTime f = (DateTime)this;
        }
        catch (ArgumentOutOfRangeException e) {
            throw new ArgumentOutOfRangeException("Fecha Incorrecta",e);
        }
    }

    public static explicit operator DateTime(Fecha f) {
        return new DateTime(f.Año, f.Mes, f.Dia);
    }
```

```
    public override string ToString() {
        return $"{Dia:D2}/{Mes:D2}/{Año:D4}";
    }
    public int CompareTo(Fecha fecha) {
        return ((DateTime)this).CompareTo((DateTime)fecha);
    }
    public Object Clone() {
        return new Fecha(Dia, Mes, Año);
    }
}
```

## Métodos Principales – Definición Y Nuevos Elementos

- Vamos a crear un diccionario donde la clave sea la fecha de un hecho importante y el valor un string con su descripción.

```
Dictionary<Fecha, string> d = new Dictionary<Fecha, string>() {
    { new Fecha(1, 6, 2018), "Moción de censura a Rajoy" },
    { new Fecha(7, 6, 2018),  "Gobierna Pedro Sánchez" }
};

d.Add(new Fecha(14, 4, 2019), "Última temporada Juego de Tronos");
Fecha f = new Fecha(4, 2, 2019);
d[f] = "Se cumplen 15 años de la creación de Facebook";
```

# COLECCIONES EN C# IMPLEMENTADAS COMO TABLAS

## Métodos Principales – Comprobar Si Una Clave Está Introducida

```
Fecha f = new Fecha(4, 2, 2019);
d[f] = "Se cumplen 15 años de la creación de Facebook";
Console.WriteLine(d.ContainsKey(f)); // Mostrará True
```

## Métodos Principales – Comprobar Si Una Clave Está Introducida

```
d.Remove(f); // Eliminará el par clave valor si existe sino ERROR.
```

## Métodos Principales – Obtener Colecciones Claves Y Valores

```
IEnumerable<Fecha> fechas1 = d.Keys;
List<Fecha> fechas2 = new List<Fecha>(d.Keys);
string[] descripciones = new List<string>(d.Values).ToArray();
```

# COLECCIONES EN C# IMPLEMENTADAS COMO TABLAS

## Saber El N-esimo Valor Guardado

```
int N = 0;

IEnumerator<KeyValuePair<Fecha, string>> it = d.GetEnumerator();
for (int i = 0; i <= N && it.MoveNext(); i++) ;
Console.WriteLine(it.Current.Value);

// Si tenemos System.Linq esté código sería equivalente a…
Console.WriteLine(d.ElementAt(N).Value);
```

## Recorrer Pares Clave - Valor

```
foreach(KeyValuePair<Fecha, string> par in d)
    Console.WriteLine($"{par.Key}: {par.Value}");

foreach(Fecha fecha in d.Keys)
    Console.WriteLine($"{fecha}: {d[fecha]}");
```

# COLECCIONES EN C# IMPLEMENTADAS COMO TABLAS

## Ejemplo Uso De Tabla De Dispersión O Hash

Vamos a realizar un programa que realice un pequeño examen sobre las capitales de la UE. Para ello, el programa preguntará 5 capitales. Puntuando con 2 puntos cada pregunta acertada.

```
static void Main()
{
    // Definimos el diccionario con los países y sus capitales.
    Dictionary<string, string> capitalesPorPais =
                                    new Dictionary<string, string>(){
        {"España", "Madrid"},
        {"Portugal", "Lisboa"}, {"Francia", "Paris"},
        {"Luxemburgo", "Luxemburgo"}, {"Irlanda", "Dublin"}
    }
    // Aunque hemos definido por extensión.
    // Podemos añadir elemetos a posteriori de dos formas.
    capitalesPorPais.Add("Belgica", "Bruselas");
    capitalesPorPais["Alemania"] = "Berlin";
    // Obtenemos una lista de claves indizable por un entero.
    List<string> paises = new List<string>(capitalesPorPais.Keys);
    // Definimos un lista donde almacenaré los países ya preguntados
    // para no repetirnos
    List<string> paisesPreguntados = new List<string>();
```

```
    const int NUMERO_PREGUNTAS = 5;
    Random semilla = new Random();
    uint puntos = 0;
    for (int i = 0; i < NUMERO_PREGUNTAS; i++) {
        string paisPreguntado;
        do {
            paisPreguntado = paises[semilla.Next(0, paises.Count)];
        } while (paisesPreguntados.Contains(paisPreguntado) == true);
        paisesPreguntados.Add(paisPreguntado);

        Console.Write($"¿Cual es la capital de {paisPreguntado}? > ");
        string capitalRespondida = Console.ReadLine().ToUpper();

        string mensaje;
        if (capitalRespondida ==
            capitalesPorPais[paisPreguntado].ToUpper()) {
            puntos += 2;
            mensaje = $"Correcto !!";
        }
        else {
            mensaje = "Incorrecto !!\n" +
            $"La respuesta es {capitalesPorPais[paisPreguntado]}.";
        }
        mensaje += $"\nLlevas {puntos} puntos.\n";
        Console.WriteLine(mensaje);
    }
    Console.WriteLine($"Tu nota final es {puntos}.");
}
```

# COLECCIONES EN C# IMPLEMENTADAS COMO TABLAS

## Comparación De Claves Para Acceso Y Comprobación De Existencia - I

- Aunque los tipos valor y las cadenas con el mismo contenido son tipos iguales. Esto último no está tan claro para tipos iguales.
- De esta forma para nuestro ejemplo sucederá que si …

```
Fecha f1 = new Fecha(4, 2, 2019);
d[f1] = "Se cumplen 15 años de la creación de Facebook";
Fecha f2 = new Fecha(4, 2, 2019);

Console.WriteLine(d[f1]); // Mostrará la descripción asociada.
Console.WriteLine(d[f2]); // generará una excepción.
```

## ¿Cómo Solucionarlo ?

Tendrmos 2 posiblididades:

- Mediante la definición de un tipo que implemente IEqualityComparer<TKey> y le pasemos una instancia al crear el diccionario.
- Que el tipo de la clave (Tkey) implemente IEquotable<TKey> y invalide el método de Object int GetHashCode()

# COLECCIONES EN C# IMPLEMENTADAS COMO TABLAS

## Uso De IEqualityComparer<TKey>

Para nuestro ejemplo haremos.

```
// En la clase Fecha
class Fecha : IComparable<Fecha>, Icloneable
{
    public class ComparaIguadad : IequalityComparer<Fecha>
    {
        public bool Equals(Fecha x, Fecha y) {
            return x.CompareTo(y) == 0;
        }

        public int GetHashCode(Fecha obj) {
            return int.Parse($"{obj.Año}{obj.Mes:D2}{obj.Dia:D2}");
        }
    }

    …
}

// Al Definir el Diccionario…
Dictionary<Fecha, string> d =
    new Dictionary<Fecha, string>(new Fecha.ComparaIguadad()) { … };
```

# COLECCIONES EN C# IMPLEMENTADAS COMO TABLAS

## Uso De IEquotable<TKey>

Para nuestro ejemplo dejaremos la creación del diccionario igual.

```
Dictionary<Fecha, string> d = new Dictionary<Fecha, string>() { … };
```

Pero en la clase Fecha que es el tipo de la clave haremos…

```
class Fecha : IComparable<Fecha>, ICloneable, IEquatable<Fecha>
{
    public bool Equals(Fecha other) {
        return CompareTo(other) == 0;
    }

    public override int GetHashCode() {
        return int.Parse($"{Año}{Mes:D2}{Dia:D2}");
    }
     …
}
```