

Tema 10.1

[Descargar estos apuntes](#)

Índice

1. Colecciones en la BCL
 1. Concepto de lista enlazada
 1. El TAD Nodo básico
 2. El TAD Lista Simple Enlazada
 1. Añadir o insertar nodos
 2. Borrar nodos
 3. Moverse por los nodos
 4. Buscar o localizar nodos
 2. Colección LinkedList<T>
 3. Colección List<T>
 4. Colección Dictionary<K, V>
 1. Ejemplo de uso de diccionarios

Colecciones en la 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 `ICloneable`, 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 qué ser 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 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** (FIFO) a través del tipo `Stack<T>`
 - las **Colas** (LIFO) 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.

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.

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

El TAD Nodo básico

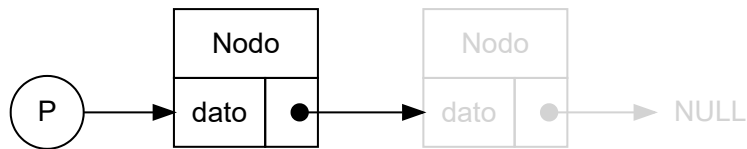
La forma que tenemos de **almacenar los datos** en las colecciones dinámicas en encapsulados en un TAD (Clase) denominada **Nodo**, que además de almacenar una propiedad con el dato, tendrá otra que referencie al siguiente Nodo.

La definición más común de Nodo será la siguiente:

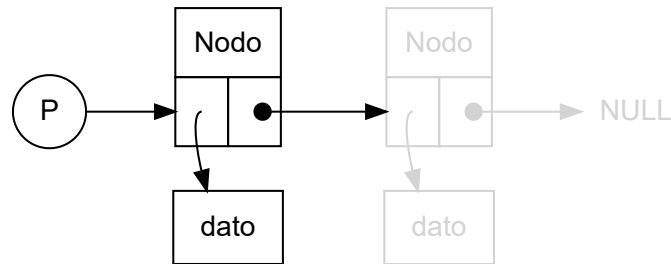
```
public class Nodo<T> : IDisposable
{
    public Nodo<T> Siguiente { get; set; }
    public T Dato { get; }

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

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



Representación conceptual si dato es un tipo valor



Representación conceptual si dato es un tipo referencia

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

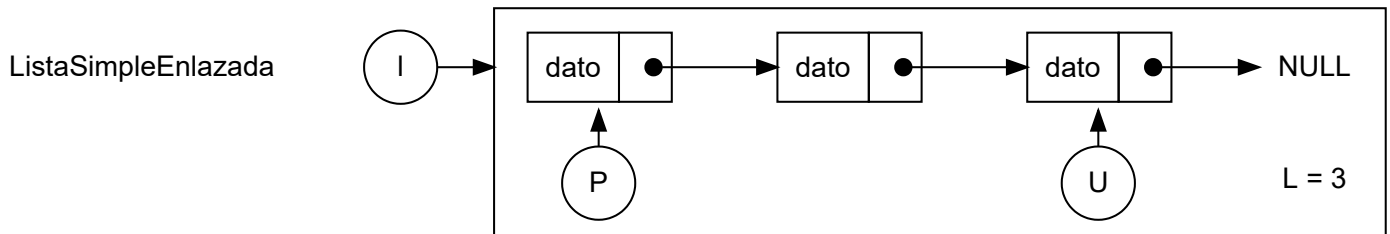
El tipo, como mínimo, deberá guardar una **referencia al primer Nodo**. Aunque lo normal es que esté formada por una referencia al primer nodo, una referencia al último y una propiedad con la longitud actual de la lista.

⚠ 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>
    : IDisposable, ICloneable, IComparable<Lista<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; }
}
```

📌 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:

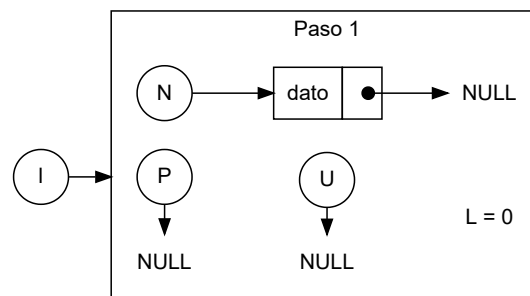
Añadir o insertar nodos

Se nos podrán dar **4 casos**:

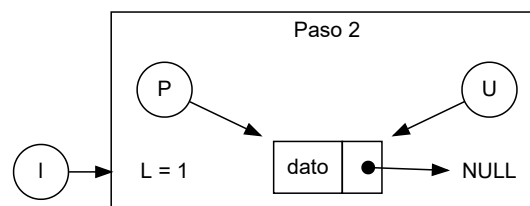
1. Insertamos en una lista vacía:

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.

En el primer paso es siempre crear el nodo que vamos a insertar en la lista.



En el segundo paso, como la lista está vacía **Primero** y **Ultimo** apuntarán al nuevo nodo e incrementaremos **Longitud** en 1.

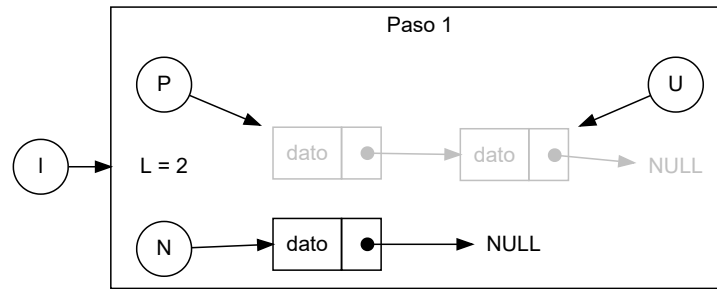


```

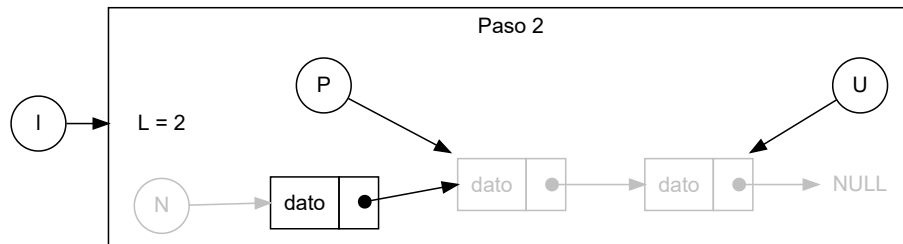
public void AñadeVacía(T dato)
{
    // Paso 1
    Nodo<T> n = new Nodo<T>(dato);
    // Paso 2
    Primero = n;
    Ultimo = n;
    Longitud++;
}
  
```

2. Insertar al principio de la lista:

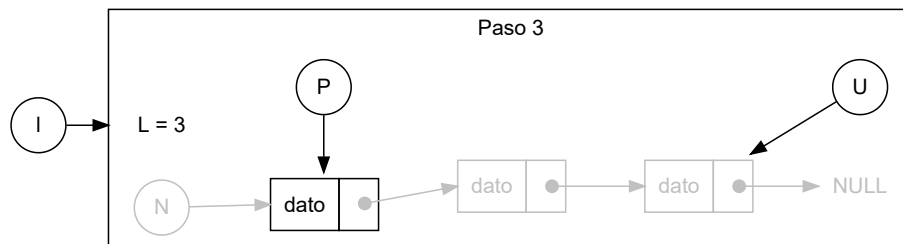
Como en todos los casos de inserción, el primer paso será crear el nodo a insertar y que contendrá el dato que nos llega.



El segundo paso es hacer que la propiedad **Siguiente** del nuevo nodo referencie al **primer** elemento de la lista.



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



```
public void AñadeAlPrincipio(T data)
{
    // 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.

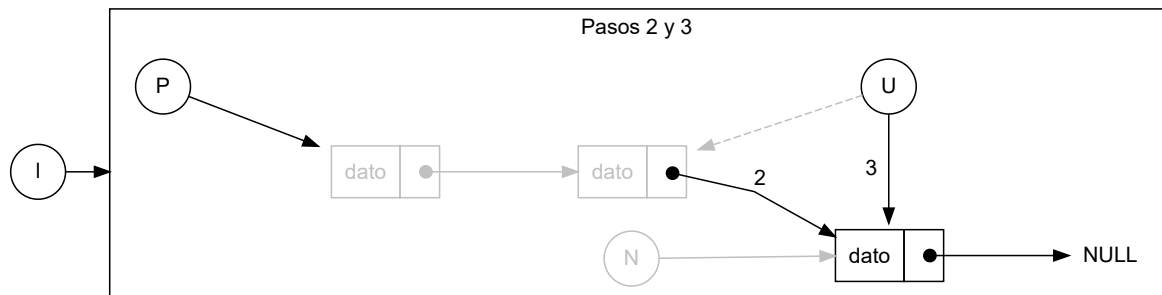
```

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

```

3. Insertar al final de la lista:

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



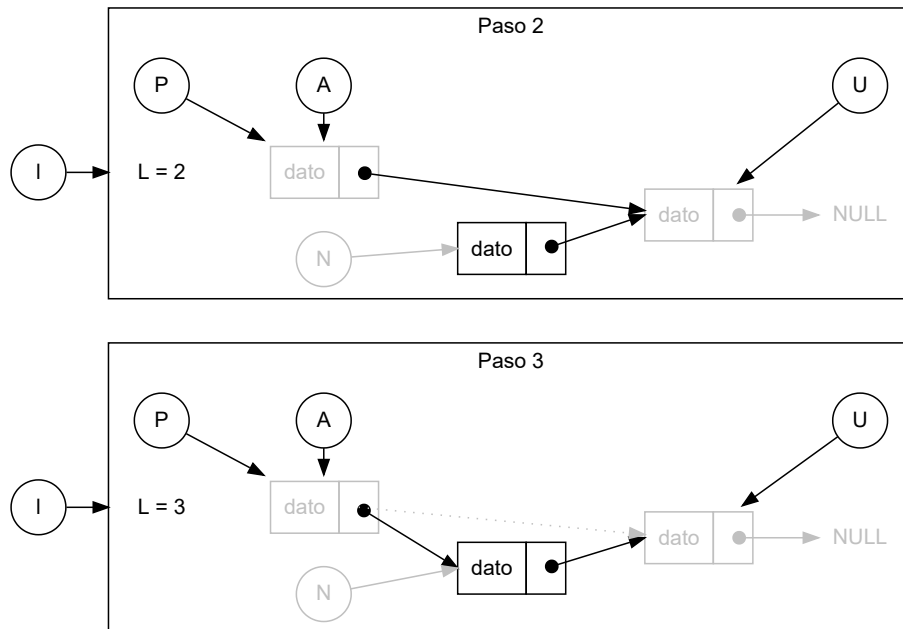
```

public void AñadeAlFinal(T data)
{
    // Paso 1
    Nodo<T> n = new Nodo<T>(dato);
    // Paso 2
    6 if (Longitud == 0) // Como antes también podemos controlar si la lista
    7     Primero = n; // está vacía pues no podremos acceder a Ultimo.Siguiente
    else
        Ultimo.Siguiente = n;
    // Paso 3
    Ultimo = n;
    Longitud++;
}

```

4. Insertar en una posición determinada de la lista:

En este caso al 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 al nodo anterior al que queremos insertar.



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

📌 **Nota:** En las listas simples solo podremos insertar después del nodo indicado. Tras ver el proceso de inserción reflexiona por qué.

Borrar nodos

Moverse por los nodos

Buscar o localizar nodos

Colección LinkedList<T>

- Implementan el TAD (Tipo Abstracto de Datos) de programación clásico denominado: '*lista doblemente enlazada*'.
- Mejora las listas simplemente enlazadas ya que me permitirá recorrer de forma más simple los nodos en las mismas, así como hacer recorridos inversos de elementos.
- No dispondrá de los métodos del interfaz de **IList** y **los accesos a través de indizador no tienen complejidad $O(1)$** .
- Este TAD irá asociado al que define la clase **LinkedListNode** que representará un nodo de la lista y nos ayudará a interactuar con ella.
- Como comentamos en la introducción, es el tipo adecuado si voy a necesitar hacer muchas inserciones y borrados pues tienen un coste **$O(1)$** .

Colección List<T>

Las listas son una especie de arrays dinámicos que encapsulan las operaciones añadir o quitar elementos en cualquier momento y de cualquier parte del array. Para trabajar con ellas se utiliza el TAD

List<T> de C#.

Por ejemplo, si creamos una lista de tipo **Persona** podremos acceder a datos de forma idéntica a los Arrays ...

```
class Persona
{
    public string Nombre { get; }
    public int Edad { get; private set; }
    public Persona(string nombre, int edad)
    {
        Nombre = nombre;
        Edad = edad;
    }
    public override string ToString() => $"{Nombre} {Edad} años";
}
```

```

class Ejemplo
{
    static void Main()
    {
5        // Definiremos la lista del tipo Persona y aunque es posible,
        // no hará falta dimensionarla previamente.
7        var personas = new List<Persona>();

        Console.Write("Introduce las persona a leer: ");
        int numeroPersonas = int.Parse(Console.ReadLine());

        for (int i = 0; i < numeroPersonas; i++)
        {
            Console.Write($"Nombre {i+1}: ");
            string nombre = Console.ReadLine();
            Console.Write($"Edad {i+1}: ");
            int edad = int.Parse(Console.ReadLine());

19        // La operación de añadir se simplifica y no habrá que hacer un Resize
20        personas.Add(new Persona(nombre, edad));
        }

23        // Puedo recorrer la colección como en los arrays tradicionales.
        for (int i = 0; i < personas.Count; i++)
            Console.WriteLine(personas[i]);
        foreach (var p in personas)
            Console.WriteLine(p);
    }
}

```

Principales **operaciones** que se pueden realizar en una lista:

- **Inicializar** la Lista, para poder usarla.

Es lo primero que debemos hacer es instanciar la lista para luego ir añadiendo elementos. Esto se hace definiendo una variable de tipo `List<T>` y usando el operador `new` para inicializarla.

Supongamos una lista de cadenas, la inicialización se haría de la siguiente manera...


```

// Se instanciarán indicándo el tipo que va a contener List<T> lista = new List<T>();
List<string> textos = new List<string>();

// Puedo darles una capacidad inicial como a los arrays si la se.
var textos = new List<string>(5000);

```

- **Añadir** datos una vez tenemos la lista creada, podemos añadir datos de dos formas:
 1. Usando el método de instancia **Add**, añadiremos un elemento al final de ella.
`textos.Add("cadena añadida")`
 2. Usando el método de instancia **Insert**, añadiremos un elemento en la posición que indiquemos.
`textos.Insert(1, "otra cadena")`


 **Importante:** A la hora de gestionar las listas, hay que tener en cuenta que las posiciones empiezan a numerarse por el cero. Además, si se intenta sobrepasar o acceder a una posición mayor al tamaño de la lista se producirá una excepción similar a lo que ocurre con los arrays.

- **Modificar** un dato. Como en los arrays, si queremos cambiar directamente el valor de uno de los datos de la lista, basta con que accedamos a su posición y modifiquemos o le asignemos otro valor.


```
textos[1] = "Hola"
```

- **Eliminar** un dato de la lista, en este caso también tenemos varias alternativas:
 1. Usando el método de instancia **Remove**, eliminamos el elemento que coincida con el que se pasa como parámetro.
`textos.Remove("Hola");`
Elimina el elemento `"Hola"` de la lista y si hubiera varios, eliminaría el **primero** que se encuentre.
 2. Usando el método de instancia **RemoveAt**, elimina el elemento de la posición que indiquemos:

```
textos.RemoveAt(0);
```

 **Importante:** Debemos tener en cuenta que estas operaciones cambiarán el tamaño de la lista y por tanto deberemos llevar cuidado si eliminamos elementos mientras la recorremos o si nos hemos guardado la posición de algún dato ya que cambiará tras la eliminación de un elemento anterior.

- **Recorrer** los elementos de la lista a través de un bucle `for` o `foreach` como vimos en el ejemplo anterior.

 **Nota:** La longitud de la lista ya no es la propiedad `Lenght` como en los arrays, sino la propiedad `Count`.

Colección Dictionary<K, V>

Las tablas hash o **Diccionarios**, son otro tipo de colecciones, pero que tienen un comportamiento particular.


Hasta ahora, todos los elementos de una colección se acceden a través de un índice numérico. Si tenemos una lista, el primer elemento ocupa la posición 0, el siguiente la 1, etc. Si queremos acceder al cuarto elemento de una lista llamada `miLista`, tenemos que poner `miLista[3]`, y **si no sabemos la posición** debemos usar un bucle.

Para evitar el bucle y realizar un acceso directo podemos usar tablas hash. En este tipo de colecciones, cada dato que agregamos a ella no tiene asociado un índice numérico, sino un objeto **clave** que lo identifica.

De esta manera, si conocemos la **clave** del dato, podemos acceder directamente a sus datos sin tener que recorrer toda la lista.

Por ejemplo, podemos asociar el dni de cada persona con sus datos completos, teniendo al final una tabla como esta:

Clave	Valor
11224441K	Nombre = "Pepe" Edad = 30
11335499M	Nombre = "María" Edad = 22
12345678O	Nombre = "Juan" Edad = 33

 **Importante:** Si quiero consultar los datos de María, buscaré por su clave que es su dni. Fíjate que la clave puede ser cualquier tipo de dato. En este caso es un string, pero podrían ser enteros u otro tipo cualquiera, siempre que nos aseguremos que **no haya dos claves repetidas**.

Si nos fijamos, el funcionamiento es similar a un diccionario real. Si quiero consultar el significado de una palabra y sé cuál es esa palabra, voy a la página donde está y la consulto, sin tener que ir palabra a palabra comprobando si es esa la que busco.


Como ya hemos comentado, las tablas hash en C# se manejan con el TAD **Dictionary<K, V>** y sus operaciones básicas son...

- **Inicializar** podemos crear un diccionario en C# inicializándolo de la siguiente forma...

```
Dictionary<TClave, TValor> tabla = new Dictionary<TClave, TValor>()
```

Donde realmente los elemento de mi colección serán objetos del tipo

`KeyValuePair<TClave, TValor>` que guardará una clave y su valor. No obstante, muy raramente los vamos a trabajar a través de él.

 **Importante:** Como hemos comentado en la introducción del tema, el tipo usado como clave debe implementar el interfáz **IEquatable<T>** sin embargo no debemos preocuparnos por los tipos básicos y los definidos en las BCL porque ya están implementados en todos ellos.

- **Añadir** un dato al diccionario creado, se puede realizar de varias formas, una de ellas es usar el método **Add**, indicando la clave que queremos asociar a cada elemento y el elemento en sí.

 La operación **Add** generará una excepción si añadimos una clave ya existente.

Si por ejemplo estamos haciendo una tabla de elementos de tipo **Persona**, la clave puede ser el **dni** de la persona en sí, y el elemento a guardar el resto de datos podríamos hacer...

```
static Dictionary<string, Persona> LeeDatos()
{
    Dictionary<string, Persona> personas = new Dictionary<string, Persona>();
    Console.WriteLine("Introduce las persona a leer: ");
    int numeroPersonas = int.Parse(Console.ReadLine());

    for (int i = 0; i < numeroPersonas; i++)
    {
        Console.WriteLine($"DNI {i + 1}: ");
        string dni = Console.ReadLine();
        Console.WriteLine($"Nombre {i + 1}: ");
        string nombre = Console.ReadLine();
        Console.WriteLine($"Edad {i + 1}: ");
        int edad = int.Parse(Console.ReadLine());

        // Añadir con el método add.
        17 personas.Add(dni, new Persona(nombre, edad));
    }
    return personas;
}
```

- **Eliminar** un dato de la lista, para ello usamos el método **Remove** con la clave del valor que queremos eliminar como argumento. Si no existe la clave obtendremos una excepción.

```
personas.Remove("11223314L")
```

- **Modificar/Añadir** el valor de un dato almacenado en el diccionario. También se accede indizando la clave y se asigna el objeto.

```
personas["11224441K"] = new Persona("Pepe", 31)
```

👉 **Importante:** Funciona igual que el **Add** solo que si la clave existe modificará su valor asociado sin generar una excepción.

💀 Además, como en el caso del borrado, si intentamos acceder a un valor del Dictionary del que no existe la clave el sistema lanzará una excepción. Por lo que es buena práctica utilizar el método **ContainsKey(clave)** , para comprobar si existe la clave antes de acceder al valor a través de ella.


- **Para obtener claves y valores por separado** dispondremos de las propiedades `Keys` y `Values` que me devolverán una secuencia iterable que puedo transformar a algún tipo de colección de las que conocemos ya sea un array una lista de la siguiente manera.

```
List<string> listaDNIs = new List<string>(personas.Keys);  
Persona[] arrayPersonas = new List<Persona>(personas.Values).ToArray();
```

- **Recorrer** diccionarios no es lo común, ya que su acceso es mediante clave. Aunque se puede realizar el acceso a todos los elementos usando un `foreach`.

Por ejemplo, este bucle saca las edades de todas las personas:

```
foreach (string dni in personas.Keys)  
    Console.WriteLine(personas[dni].Edad);
```

 **Nota:** Es posible que el orden de salida no sea el mismo que cuando se introdujeron los datos, ya que las tablas hash tienen un mecanismo de ordenación diferente.

Realmente si tenemos en cuenta que nuestro diccionario realmente es una secuencia de valores del tipo `KeyValuePair<string, Persona>`. Podríamos recorrer sus valores también de la siguiente forma...

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

Ejemplo de uso de diccionarios

Vamos su uso a través de un simple **programa de ejemplo** 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"}, // Par clave país, valor capital.
        {"Portugal", "Lisboa"},
        {"Francia", "Paris"},
        {"Luxemburgo", "Luxemburgo"},
        {"Irlanda", "Dublin"}
    };
    // Aunque hemos definido por extensión. Podemos añadir elemetos a posteriori.
    capitalesPorPais.Add("Belgica", "Bruselas");
    capitalesPorPais["Alemania"] = "Berlin";
    // Obtenemos una lista de claves indizable por un entero.
    List<string> paises = new List<string>(capitalesPorPais.Keys);
    // 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();
        bool respuestaCorrecta = capitalRespondida == capitalesPorPais[paisPreguntado].ToUpper();
        if (respuestaCorrecta) puntos += 2;
        string mensaje = (respuestaCorrecta
            ? "Correcto !!"
            : $"Incorrecto !!\nLa respuesta es {capitalesPorPais[paisPreguntado]}.
            + $" \nLlevas {puntos} puntos.\n");
        Console.WriteLine(mensaje);
    }
    Console.WriteLine($"Tu nota final es {puntos}.");
}
```