

Tema 10.2

Descargar estos apuntes en [pdf](#) o [html](#)

Índice

1. Colecciones tipo tabla en las BCL
 1. Colección List<T>
 1. Conversiones entre List<T> y otras colecciones
 2. El TAD Cola
 1. Queue<T> el TAD Cola en las BCL
 3. El TAD Pila
 1. Stack<T> el TAD Pila en las BCL
 4. Colección Dictionary<K, V>
 1. Profundizando en el uso de los diccionarios
 1. Usando tipos propios como claves

Colecciones tipo tabla en las BCL

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()
    {
        // Definiremos la lista del tipo Persona y aunque es posible,
        // no hará falta dimensionarla previamente.
        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());

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

        // 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 indicando 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);

// Puedo definir unos datos iniciales como en los arrays.
var textos = new List<string>() { "texto1", "texto2", "texto3"};
```

- **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 **Length** como en los arrays, sino la propiedad **Count**.

Conversiones entre List<T> y otras colecciones

Como ya vimos con las colecciones vinculadas, podemos crear una **List<T>** a partir de cualquier colección que implemente **IEnumerable<T>**, pero tendremos que tener cuidado si son tipos referencia mutables, porque ambas colecciones compartirán la referencia a esos objetos.

1. Convertir colección a **List<T>**

```
int[] v = { 8, 3 };
// Lista a partir de un array.
List<int> l1 = new List<int>(v);

LinkedList<int> l1 = new LinkedList<int>();
l1.AddLast(8);
l1.AddLast(3);
// Lista a partir de una lista enlazada.
List<int> l2 = new List<int>(l1);
```

2. Convertir **List<T>** a otra colección.

```
List<int> l = new List<int>();
l.Add(8);
l.Add(3);

// Al implementar List<T> IEnumerable<T> podremos pasarla como secuencia a cualquier colección
LinkedList<int> l1 = new LinkedList<int>(l);

// Para convertir a array las colecciones que internamente son tablas
// suelen implementar el método ToArray() de paso inmediato.
int[] v = l.ToArray();
```

Caso de estudio:

Supongamos que tenemos un **array de enteros muy grande** (en nuestra prueba vamos a usar **500000** enteros) y queremos hacer una función que reciba dicho array y me filtre los datos devolviéndome únicamente los enteros pares. Para ello, vamos a utilizar diferentes algoritmos o estrategias así como diferentes colecciones para realizar el filtrado y testear así las diferentes características que hemos descrito a lo largo de este tema 10.

Vamos a definir el primer lugar un método que me genere el array de números enteros a filtrar de forma aleatoria. Una posibilidad podría ser el siguiente código...


```
using System;
using System.Collections.Generic;
using System.Diagnostics;

class CasoDeEstudio
{
    static int[] ArrayNumeros()
    {
        const int LIMITE = 500000;
        int[] numeros = new int[LIMITE];
        Random seed = new Random();

        for (int i = 0; i < numeros.Length; i++)
            numeros[i] = seed.Next(1, LIMITE + 1);

        return numeros;
    }
}
```

En primer lugar vamos a utilizar únicamente arrays y para filtrar vamos a crear otro array que redimensionaremos en 1 como hemos hecho a lo largo del curso.

 **Nota:** Recordemos que la operación **Resize** sobre un array tiene un coste temporal alto y que los arrays no permiten operaciones de eliminación de elementos que me permitiría optar por una estrategia de obtener los pares eliminando los impares. Una posible implementación podría ser la siguiente....

```
static int[] ObtenerParesArraysYResize(in int[] numeros)
{
    int[] pares = null;

    for (int i = 0; i < numeros.Length; i++)
        if (numeros[i] % 2 == 0)
        {
            Array.Resize(ref pares, pares == null ? 1 : pares.Length + 1);
            pares[pares.Length - 1] = numeros[i];
        }

    return pares;
}
```

Ahora vamos a utilizar la misma estrategia pero usando internamente una colección de tipo **List<int>** en lugar de un **int[]** . Internamente es similar, pero recordemos que su implementación se ha optimizado para **añadir elementos al final**.

Una posible implementación podría ser la siguiente....


```
static int[] ObtenerParesUsandoListYAdd(in int[] numeros)
{
    List<int> pares = new List<int>();

    for (int i = 0; i < numeros.Length; i++)
        if (numeros[i] % 2 == 0)
            pares.Add(numeros[i]);

    return pares.ToArray();
}
```

Caso de estudio continuación ...

En la siguiente implementación aprovechando que las listas implementan el método `RemoveAt(int pos)` para borrar en una posición determinada. Vamos a implementar el algoritmo borrando aquellos números impares que encontremos.

 **Nota:** Recuerda que el borrado es costoso en `List<T>`

```
static int[] ObtenerParesUsandoListYRemoveAt(in int[] numeros)
{
    // Convierto el array de entrada a una lista para las operaciones.
    List<int> lpares = new List<int>(numeros);

    var i = 0;
    // Mientras no hayamos llegado el final del array...
    while(i < lpares.Count)
    {
        if (lpares[i] % 2 != 0)
            // Si borramos no hacemos i++ porque al borrar el siguiente
            // elemento ocupará el lugar del que estamos borrando.
            lpares.RemoveAt(i);
        else
            i++;
    }

    // Vuelvo a transformar la lista en un array
    // que es lo que devuelve el método.
    return lpares.ToArray();
}
```

Ahora vamos a implementar un algoritmo análogo pero usando `LinkedList<T>` que es más eficiente para borrados.

```
static int[] ObtenerParesUsandoLinkedListYRemove(in int[] numeros)
{
    // Convierto el array de entrada a una lista enlazada.
    LinkedList<int> llpares = new LinkedList<int>(numeros);

    var it = llpares.First;
    // Mientras el iterador de nodos apunte a algún nodo.
    while(it != null)
    {
        // Me quedo con la referencia al siguiente nodo por
        // si al hacer el Remove del mismo se perdiese.
        var s = it.Next;

        if (it.Value % 2 != 0)
            llpares.Remove(it);

        it = s; // Tanto si borro como sino voy al siguiente.
    }

    // Paso de nuevo la lista enlazada a un array
    // que es lo que devuelve el método.
    int[] pares = new int[llpares.Count];
    llpares.CopyTo(pares, 0);
    return pares;
}
```

Voy a realizar un programa principal donde '*cronometre*' el tiempo de ejecución de cada uno de los métodos que he implementado, operando sobre el mismo array generado aleatoriamente y así poder comparar las diferentes estrategias.

Caso de estudio continuación ...

En la siguiente implementación aprovechando que las listas implementan el método

```
public static void Main()
{
    Stopwatch c = new Stopwatch(); // Instancio un cronometro.

    int[] numeros = ArrayNumeros(); // Genero el array de 500000 números.

    c.Start();
    Console.Write("Usando Arrays y Resize, t -> ");
    _ = ObtenerParesArraysYResize(numeros);
    c.Stop();
    Console.WriteLine($"{c.ElapsedMilliseconds} ms");

    c.Restart();
    Console.Write("Usando List y Add, t -> ");
    _ = ObtenerParesUsandoListYAdd(numeros);
    c.Stop();
    Console.WriteLine($"{c.ElapsedMilliseconds} ms");

    c.Restart();
    Console.Write("Usando List y RemoveAt, t -> ");
    _ = ObtenerParesUsandoListYRemoveAt(numeros);
    c.Stop();
    Console.WriteLine($"{c.ElapsedMilliseconds} ms");

    c.Restart();
    Console.Write("Usando LinkedList y Remove, t -> ");
    _ = ObtenerParesUsandoLinkedListYRemove(numeros);
    c.Stop();
    Console.WriteLine($"{c.ElapsedMilliseconds} ms");
}
```

Tras ejecutar el programa principal obtengo los siguientes tiempos ...

```
Usando Arrays y Resize, t -> 21402 ms
Usando List y Add, t -> 11 ms
Usando List y RemoveAt, t -> 5830 ms
Usando LinkedList y Remove, t -> 42 ms
```

Si nos fijamos en los resultados, podemos destacar que:

1. Usar el `List<T>` con `Add` es **2000 veces más rápido** que hacer un `Resize` para añadir cada número par.
2. **Borrar** una posición en un `LinkedList<T>` ha sido **138 veces más rápido** que hacerlo en un `List<T>` .
3. La mejor estrategia para filtrar ha sido añadir en un `List<T>` .

Prueba a ejecutar el programa con diferentes tamaños del array de entrada y ejecuciones aleatorias y reflexiona sobre las implementaciones propuestas.

El TAD Cola

Los elementos se añaden por el final y se suprimen por el principio denominado **frente de la cola**. Por esta razón, también se les conoce como estructuras **FIFO**, acrónimo en ingles de **'Primero en Entrar, Primero en Salir'**.

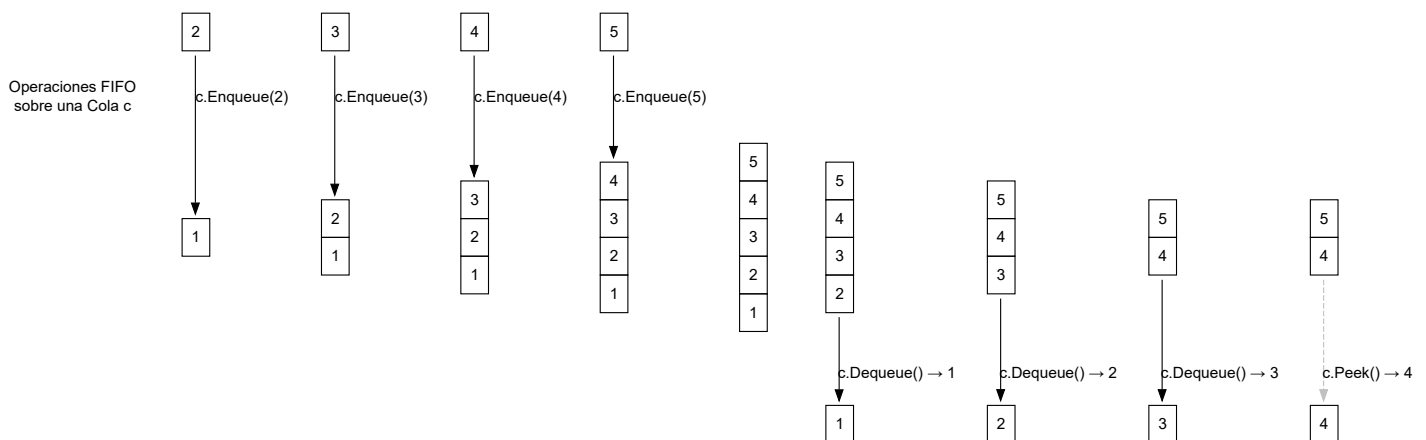
Podremos implementar el TAD Cola de forma vinculada o con una tabla. Pero su funcionalidad será la misma, porque realmente lo que implementa son ciertas operaciones básicas que veremos a continuación.

Queue<T> el TAD Cola en las BCL

En las BCL está **implementado como tabla** y no de forma vinculada.

Las **operaciones básicas** son:

- **Encolar** (`c.Enqueue(T dato)`).
- **Desencolar** (`c.Dequeue()` → `T`).
- **Borrar** toda la cola (`c.Clear()`)
- **Consultar el frente** de la cola sin desencolar (`c.Peek()` → `T`).
- Ver si está **vacía** (`c.Count == 0`).



El siguiente código equivaldría a las operaciones descritas en la ilustración.

```
public static void Main()
{
    // Creamos una cola con un elemento inicial a partir de otra colección.
    Queue<int> c = new Queue<int>(new int[] {1});

    Console.WriteLine($"Cola -> {string.Join(", ", c)}");
    for (int i = 2; i <= 5; i++)
    {
        c.Enqueue(i);
        Console.WriteLine($"Encolando el {i}");
        Console.WriteLine($"Cola -> {string.Join(", ", c)}");
    }
    while (c.Count > 2)
    {
        Console.WriteLine($"Desencolado el {c.Dequeue()}");
        Console.WriteLine($"Cola -> {string.Join(", ", c)}");
    }
    Console.WriteLine($"Consultando el frente de la cola {c.Peek()}");
    Console.WriteLine($"Cola -> {string.Join(", ", c)}");
}
```

Aunque a simple vista son una concreción de las listas, puede ser interesante tener solo un subconjunto de operaciones más limitado, enfocado a ciertos problemas y que nos pueda proporcionar más *'robustez'* de cara a errores, legibilidad del código e incluso rendimiento en la ejecución del mismo.

Normalmente las colas se usan para gestionar datos, eventos o procesos asíncronos por orden de llegada. El ejemplo más paradigmático sería la **cola de impresión**, que irá encolando los '*trabajos*' de impresión y desencolándolos por orden conforme quede libre un recurso de impresión.

Caso de estudio:

Hemos visto que internamente el tipo `Queue<T>` está implementado con una **tabla o 'Array'**. Pero, también podría estarlo como una **lista vinculada**. Para ello, vamos a definir nosotros la colección `LinkedList<T>` y vamos usar internamente una `LinkedList<T>` linked list para hacerlo y generará excepciones del tipo `LinkedListException` si no se puede realizar la operación.

Nuestra va ofrecernos solo las operaciones típicas de una cola como son:

- `LinkedList()`
- `LinkedList(IEnumerable secuencia)`
- `void Enqueue(T data)`
- `T Dequeue()`
- `T Peek()`
- `void Clear()`
- `bool Empty { get; }`
- `int Count { get; }`
- Además implementará el interfaz `IEnumerable`

Se nos podría ocurrir establecer una relación de herencia del tipo `class LinkedList<T> : LinkedList<T>`. Pero esto sería incorrecto pues nuestra cola además de sus operaciones específicas, también nos ofrecería las de la lista vinculada. Realmente lo que sucede es que una cola **no es una** lista vinculada y por tanto en este caso la mejor opción es definir una **composición**.

Nuestra definición pues **'envolverá'** encapsulando el tipo `LinkedList<T>` en el que se va a apoyar ofreciendo solo las operaciones de cola que gestionaremos a través del tipo envuelto. A este tipo de diseño se le conoce como **Wrapper**.

¿Se te ocurre cómo hacerlo?. Si no es así a continuación puedes implementar para su estudio la siguiente propuesta comentada.

```
// Definición del tipo que concretará la excepciones de la cola vinculada.
class LinkedListException : Exception
{
    public LinkedListException(string message) : base(message)
    {
    }
}

// Definimos nuestro tipo parametrizado con el comportamiento de
// una secuencia iterable como otras colecciones.
class LinkedList<T> : IEnumerable<T>
{
    // Tipo al que estamos 'envolviendo' y sobre el que implementaremos las
    // operaciones de cola. Debe ser 'transparente' para el usuario de nuestra definición.
    private LinkedList<T> Lista { get; }

    // Definimos los 2 constructores que llamarán a los constructores del tipo 'envuelto'.
    public LinkedList()
    {
        Lista = new LinkedList<T>();
    }
    public LinkedList(IEnumerable<T> secuencia)
    {
        Lista = new LinkedList<T>(secuencia);
    }

    // Algunas de las propiedades y operaciones sobre el tipo envuelto son triviales.
    public bool Empty => Lista.Count == 0;
    public int Count => Lista.Count;
    public void Clear() => Lista.Clear();
    public override string ToString() => string.Join(", ", Lista);
}
```

Caso de estudio continuación

```
// Encolar añade al final de la lista.
public void Enqueue(T data) => Lista.AddLast(data);

// Desencolar extrae el elemento del principio de la lista.
public T Dequeue()
{
    if (Empty)
        new LinkedListException("No se puede desencolar en un cola vacía.");
    LinkedListNode<T> nodo = Lista.First;
    Lista.RemoveFirst();
    return nodo.Value;
}

public T Peek()
{
    if (Empty)
        new LinkedListException("No se puede consultar el frente de una cola vacía.");
    return Lista.First.Value;
}

// Para implementar IEnumerable<T> nos basamos en que ya está implementado en
// la colección subyacente y realmente devolvemos el objeto iterador que devuelva la misma.
public IEnumerator<T> GetEnumerator() => Lista.GetEnumerator();
IEnumerator IEnumerable.GetEnumerator() => return GetEnumerator();
}
```

Vamos a usar nuestra definición cola para implementar un simple programa que simule '*El juego de la bomba*' en el cual una bomba programada para explotar en un tiempo aleatorio. Es pasada entre los jugadores de forma rotativa del tal manera que cada jugador la retendrá un tiempo 'aleatorio' que el estime oportuno para que le explote a un compañero y no le vuelva a llegar.

```
class Program
{
    static void Main()
    {
        // Cola de jugadores.
        LinkedList<string> jugadores = new LinkedList<string>(
            new string[] {
                "Pepe", "María", "Juan", "Sara"
            }
        );
        Random seed = new Random();
        int segundosHastaExplosion = seed.Next(30, 60);
        bool explosion = false;
        do
        {
            // Desencolamos al jugador que recibirá la bomba.
            string jugador = jugadores.Dequeue();
            int espera = seed.Next(5, 10);
            Console.WriteLine($"{jugador} esperando para pasar la bomba.");
            Thread.Sleep(espera * 1000); // Espera aleatoria.
            segundosHastaExplosion -= espera;
            explosion = segundosHastaExplosion <= 0; // Comprobamos si ha explotado.
            if (!explosion)
                // Si no ha explotado encolamos de nuevo al jugador para que vuelva a recibir la bomba.
                jugadores.Enqueue(jugador);
            else
                Console.WriteLine($"La bomba le ha explotado a {jugador}.");
        }
        while(!explosion);
        jugadores.Clear();
    }
}
```

El TAD Pila

Los elementos se añaden y extraen **por el mismo extremo** que denominaremos **cabeza de la pila**. Por esta razón, también se les conoce como estructuras **LIFO**, acrónimo en inglés de *'Último en Entrar, Primero en Salir'*.

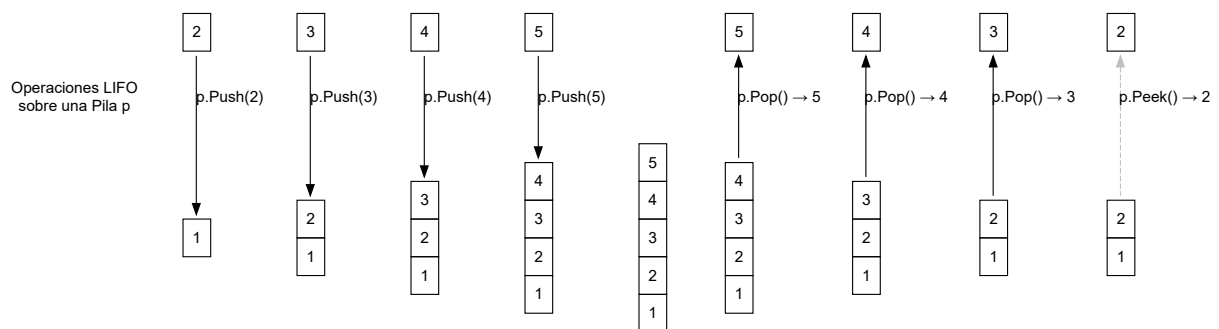
Al igual que sucedía con las colas, podremos implementar el TAD Pila de forma vinculada o con una tabla. Pero su funcionalidad será la misma, porque realmente lo que implementa son ciertas operaciones básicas que veremos a continuación.

Satck<T> el TAD Pila en las BCL

En las BCL está **implementado como tabla** y no de forma vinculada.

Las **operaciones básicas** son:

- **Apilar** (`p.Push(T dato)`).
- **Desapilar** (`p.Pop() → T`).
- **Borrar** toda la pila (`p.Clear()`).
- **Consultar la cabeza** de la cola sin desapilar (`p.Peek() → T`).
- Ver si está **vacía** (`p.Count == 0`).



El siguiente código equivaldría a las operaciones descritas en la ilustración.

```
public static void Main()
{
    // Creamos una pila con un elemento inicial a partir de otra colección.
    Stack<int> p = new Stack<int>(new int[] {1});

    Console.WriteLine($"Pila -> {string.Join(", ", p)}");
    for (int i = 2; i <= 5; i++)
    {
        p.Push(i);
        Console.WriteLine($"Apilando el {i}");
        Console.WriteLine($"Pila -> {string.Join(", ", p)}");
    }

    while (p.Count > 2)
    {
        Console.WriteLine($"Desapilando el {p.Pop()}");
        Console.WriteLine($"Pila -> {string.Join(", ", p)}");
    }

    Console.WriteLine($"Consultando la cabeza de la pila {p.Peek()}");
    Console.WriteLine($"Pila -> {string.Join(", ", p)}");
}
```

Caso de estudio:

Un típico ejemplo de uso de pilas es comprobar si una **expresión está balanceada** o no. Por ejemplo, $((3 + 4) * (2 - 5 / (2 * 4)))$ está balanceada porque se abren tantos paréntesis como se cierran de forma correcta.

Para ello, vamos a definir un método estático denominado `bool VerificaParentesis(in string expresion)` que me diga si la cadena con la expresión de entrada lo está o no.

El algoritmo consistirá en recorrer la cadena y cada vez que encontremos el caracter `'('` **apilarlo** y cuando encontremos un `')'` **desapilar** de la cadena. Si desapilamos y no hay nada en la pila, significará que no hay ningún paréntesis de apertura para el de cierre que acabamos de encontrar. Además, si tras recorrer toda la expresión quedan elementos en la pila significará que no todos los paréntesis de apertura se han logrado cerrar.

¿Serías capaz de escribir una propuesta de solución tal usando `Stack<T>` para realizar el algoritmo?

Si no se te ocurre ninguna, aquí tienes una...

```
class Program
{
    static bool VerificaParentesis(in string expresion)
    {
        var p = new Stack<char>();
        bool balanceados = true;

        for (int i = 0; i < expresion.Length && balanceados; i++)
        {
            if (expresion[i] == '(')
                p.Push('(');
            else if (expresion[i] == ')')
                balanceados = !p.Empty && p.Pop() == '(';
        }

        balanceados = balanceados && p.Empty;
        return balanceados;
    }

    static void Main()
    {
        Console.WriteLine(VerificaParentesis("((3 + 4) * (2 - 5 / (2 * 4)))"));
        Console.WriteLine(VerificaParentesis("(3 + 4) * (2 - 5 / (2 * 4)))"));
        Console.WriteLine(VerificaParentesis("((3 + 4) * (2 - 5 / (2 * 4)))"));
    }
}
```

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
13898743Y	Nombre = "Sara" Edad = 27

 **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.

- **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.
        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 que implementa **IEnumerable<T>** y que por ende puedo transformar a algún tipo de colección de las que conocemos ya sea un array o una lista como hemos visto a lo largo del tema.

```
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...

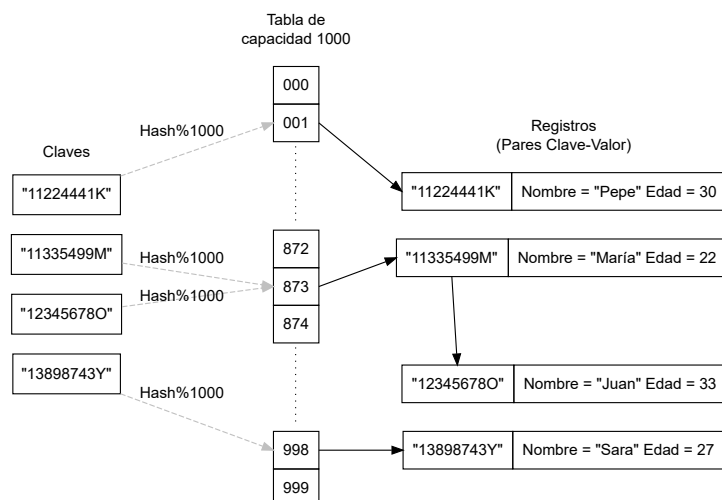
```
foreach (KeyValuePair<string, Persona> par in personas)
    Console.WriteLine($"{par.Key}: {par.Value}");
```

Profundizando en el uso de los diccionarios

Para entender alguna de las propiedades y métodos que implementa la clase **Dictionary<K, V>** deberemos entender como funcionan los mismos internamente. Para ello, supongamos la tabla de pares clave-valor que hemos puesto antes de ejemplo. Donde la clave era un **string** con el DNI y el valor un objeto de tipo **Persona**.

Una posible **aproximación** a como se organiza la información internamente podría ser el siguiente....

- 🚩 **Nota:** Para simplificar no hemos representado las referencias a los objetos **string** y **persona**



Si nos fijamos, internamente el diccionario guarda una tabla de **n** elementos de capacidad y esto es importante para que funcionen de manera eficiente. De hecho si a priori conocemos cuantos elementos va a tener el diccionario podremos dimensionarla en el constructor. Por ejemplo, si

queremos que tenga una capacidad de **1000** como en nuestro ejemplo podríamos construir el diccionario de la siguiente manera ...

```
var personas = new Dictionary<string, Persona>(1000);
```

Internamente calcula el Hash de la **clave**. En C# el Hash es un valor numérico entero que se obtiene a través del método **GetHashCode()** que cualquier objeto implemente por estar definido en la clase **Object** como virtual y que por tanto podremos invalidar en nuestras definiciones de tipos.

Si consultamos como funciona la **función de Hash** por ejemplo en la [Wikipedia](#) podemos deducir que, para un determinado estado de un objeto, esta me devuelve un valor numérico '*único*'.

Pero... ¿Para que nos sirve calcular el Hash de la clave?. Si nos fijamos en el diagrama, lo que hace el diccionario es calcular el módulo del Hash entre la capacidad de la tabla **Hash % 1000** esto nos asegurará obtener un resto entre **0 y 999** que son índices válidos para la tabla definida. Por tanto, podemos inferir que el Hash me permitirá crear una **correspondencia entre los objetos usados como clave** (**string** en nuestro caso) y un **índice** dónde guardar **el valor asociado** en la tabla, tal y como se muestra en la ilustración.

Pero... ¿Si la tabla tiene poca capacidad entonces, las posibilidades de que el resto de dividir el Hash de dos claves por el tamaño me de el mismo índice es muy alta?. Efectivamente, por eso si nos fijamos en la ilustración hay dos DNI que al dar el mismo índice se deben guardar en la misma posición, por ejemplo a través de una lista enlazada. Una vez vamos a esa posición de la tabla deberemos buscar la clave en la lista. Es por esa razón, que cada registro de la lista guarda pares de clave-valor y no únicamente el valor. Además, para saber buscar la clave en la lista deberemos saber si dos claves son iguales y por ello **los objetos que usemos como clave deben implementar** la interfaz **IEquatable<T>**.

👉 **Importante:** No debemos preocuparnos por **los tipos básicos y los definidos en las BCL**, puesto que **IEquatable<T>** está implementado en todos ellos.

🔊 Resumen:

Podemos decir que **para que un tipo pueda hacer de clave** debe invalidar **GetHashCode()** y debe implementar **IEquatable<T>**. Afortunadamente, no debemos preocuparnos porque **los tipos básicos y los definidos en las BCL** cumplen estas condiciones.

💡 **Tips:** Del funcionamiento descrito podemos entender los siguiente métodos de optimización y rendimiento de los diccionarios ...

1. `int EnsureCapacity(int capacity)`

EnsureCapacity me permitirá definir el tamaño de la tabla interna del diccionario si tengo muchas entradas evitando así que se repitan índices en las claves.

2. `void TrimExcess()`

TrimExcess si he realizado un **Clear()** o voy a tener pocos elementos en el diccionario me permitirá reducir el tamaño de la tabla adecuándolo al número de entradas.

”

The possible solutions to a given problem emerge as the leaves of a tree, each node representing a point of deliberation and decision.

- Niklaus Wirth.

”

Usando tipos propios como claves

Imaginemos que queremos definir un diccionario donde la **clave** ahora será la clase **Persona** que hemos utilizado a lo largo del tema y el valor será una lista de cadenas con los nombres de las mascotas de esa persona.

La definición básica del tipo **Persona** para usarlo como clave sería la siguiente...

```
1 // Debe implementar IEquatable<Persona>
  class Persona : IEquatable<Persona>
  {
      public string Nombre { get; }
      public int Edad { get; private set; }
      public Persona(string nombre, int edad)
      {
          Nombre = nombre;
          Edad = edad;
      }
11 // Es interesante que redefinamos el ToString para que represente el estado del
12 // objeto mediante una cadena.
      public override string ToString() => $"{Nombre} {Edad} años";

15 // Invalidamos GetHashCode() y un buen 'truco' puede ser usar el Hash de la
// cadena que define al objeto como Hash del mismo.
17 // Recordemos que String ya invalida GetHashCode() internamente.
      public override int GetHashCode() => ToString().GetHashCode();

20 // Implementamos el interfaz, que nos obliga a implementar Equals y podemos
21 // comparar fácilmente dos objetos, viendo si tienen el mismo Hash o no.
      public bool Equals([AllowNull] Persona other) => GetHashCode() == other.GetHashCode();
  }
```

Analicemos la siguiente propuesta código de ejemplo comentado donde usamos la clase **Persona** que hemos definido como clave...

```
public static void Main()
{
    // Definimos el diccionario donde la clave es una persona y
    // el valor una lista de mascotas.
    Dictionary<Persona, List<string>> mascotasXPersona = new Dictionary<Persona, List<string>>();

    // Creamos un objeto persona pepe y para ese objeto
    // añadimos una lista vacía de mascotas.
    Persona pepe = new Persona("Pepe", 30);
    mascotasXPersona.Add(pepe, new List<string>());

    // Usamos la misma referencia al objeto pepe para acceder
    // a su lista de mascotas y añadir dos nombres.
    mascotasXPersona[pepe].Add("Snowball");
    mascotasXPersona[pepe].Add("Velvet");

    // Creamos un objeto persona de nombre María del que no nos guardamos la
    // referencia y añadimos una lista inicializada en la definición con dos mascotas.
    mascotasXPersona.Add(new Persona("María", 22), new List<string>(){ "Simba", "Bella" });
    // Añadimos una tercera mascota a María, pero volvemos a instanciar otro objeto
    // Persona para María porque no nos guardamos la referencia como con pepe.
    // No debería ser problema porque ambos deberían generar el mismo Hash y además
    // sabemos comparar objetos persona con Equal.
24 mascotasXPersona[new Persona("María", 22)].Add("Lucy");

    // Mostramos la lista de mascotas por persona.
    foreach (Persona p in mascotasXPersona.Keys)
        Console.WriteLine($"{p}: {string.Join(" ", mascotasXPersona[p])}");
}
```

Prueba a ejecutar este código y comprueba que funciona correctamente. Prueba ahora a eliminar la invalidación de **GetHashCode()** de **Persona** y ver que sucede. Deberías obtener un error en la **línea 24** puesto que estamos intentado usar como clave de acceso un objeto **Persona** de nombre **María** y edad **22** años que *'supuestamente'* no está definida en el diccionario porque su Hash es diferente al del objeto con que se añadió.

Caso de estudio:

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.

Veamos una posible solución usando diccionarios y otras colecciones vistas en el tema...

```
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;

        // Buscamos un país que ún no hayamos preguntado.
        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}.");
}
```