TEMA 10. ESTRUCTURAS DE DATOS DINÁMICAS Y COLECCIONES EN LA BCL

1. Introducción a las estructuras dinámicas

Los TAD's más básicos usados en programación son las colecciones dinámicas. Hasta ahora hemos visto colecciones con un tamaño fijo que se establece en el momento de la definición como pueden ser los arrays y las matrices. Pero se hace imprescindible el tener colecciones que puedan modificar el número de elementos durante la ejecución, en casos en que no se puede establecer un número a priori. También tendremos topologías especiales de colecciones que me permitirán formas específicas de recorrido de sus elementos. Las más comunes son: Listas, Colas, Pilas y Árboles (Estos últimos no los vamos a estudiar).

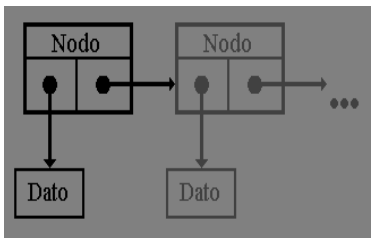
2. Topologías más comunes

Como hemos comentado anteriormente, tenemos una serie de topologías que permiten adaptarse a las necesidades específicas del problema, para conseguir un uso más eficiente de la colección. Usemos la topología que usemos, siempre se necesitará el uso del TAD Nodo.

El TAD Nodo → La forma que tenemos de almacenar los datos en las colecciones dinámicas es encapsularlos 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;
  Public Nodo(T dato)
  {
    Dato=dato;
    Siguiente=null;
  }
  Public Nodo(T dato, Nodo<T> nodo)
  {
    Dato=dato;
    Siguiente=nodo;
  }
 }
```



2.1. Listas lineales (listas enlazadas o abiertas)

En una lista lineal cada elemento referencia al siguiente y el último a NULL. Esto permite crear cada elemento individualmente, por lo que no es necesario disponer de un número de bytes consecutivos en memoria igual al tamaño total de la lista.

Las operaciones fundamentales que podemos realizar con listas son:

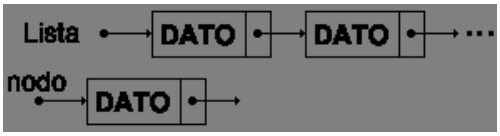
- Insertar en una lista vacía
- Insertar elementos (principio, final, mitad)
- Borrar un elemento
- Recorrer la lista
- Borrar todos los elementos
- Buscar un elemento

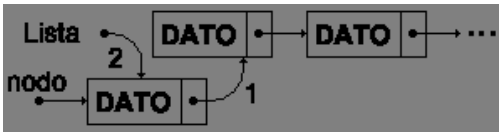
- Insertar en una lista vacía.

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

- Insertamos al principio de una lista.

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





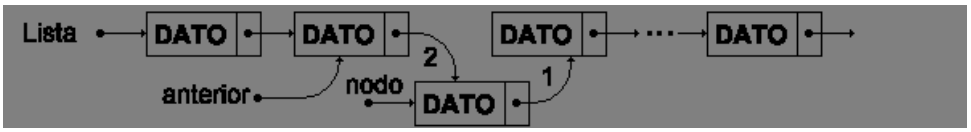
- Insertamos al final de una lista.

```
public void AñadeAlFinal(T data)
{
  Nodo<T> nodo = new Nodo<T>(dato,null);
  Nodo<T> ultimo=UltimoNodo(lista);//return último nodo lista
  ultimo.siguiente = nodo;  // 2
}
```

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

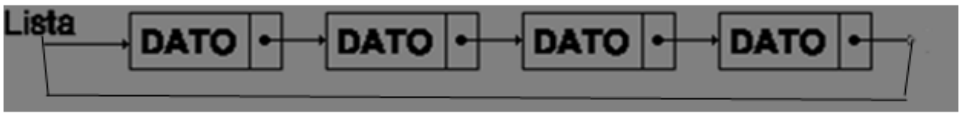
- Insertamos en mitad de una lista, pasando el nodo anterior

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



2.2. Listas circulares

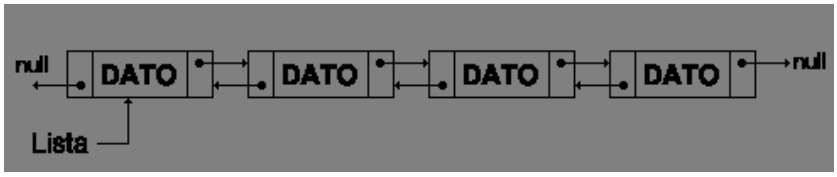
Una lista circular es una lista lineal en la que el último elemento enlaza a su vez con el primero. De este modo se puede acceder a cualquier miembro de la lista desde cualquier punto dado.



Las operaciones con este tipo de listas son más sencillas.

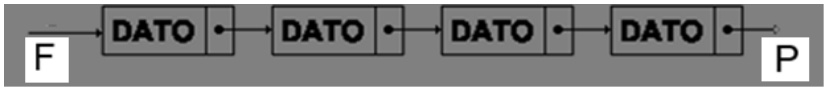
2.3. Listas doblemente enlazadas

Son listas lineales en las que cada elemento tiene dos enlaces. Uno al elemento siguiente y otro al anterior. Esto permite avanzar en la lista en cualquier dirección.



2.4. Colas

Las colas son listas lineales en las que todas las inserciones se hacen por un extremo y todas las supresiones se hacen por el otro. Se llaman listas FIFO (First in firs out- primero en entrar primero en salir).
En este caso es importante crear dos referencias para controlar el principio y el final de la cola.

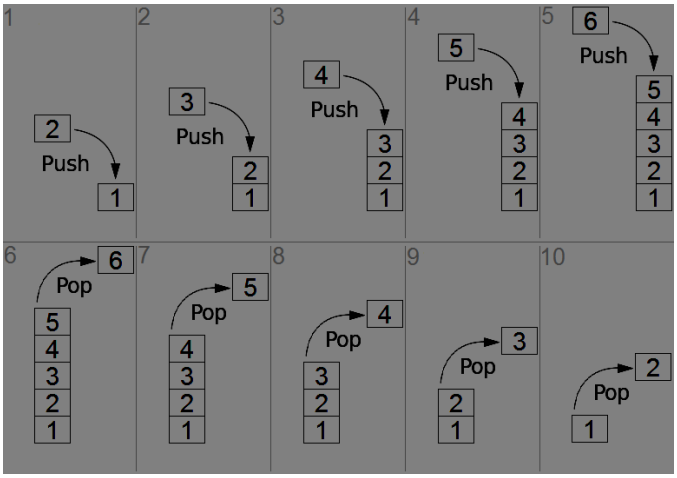


Da igual como implementemos el TAD ya sea a través de listas o de un array, su funcionalidad mínima debe, para cualquier implementación, ser la siguiente: Supongamos Tipo el tipo de los elementos que guarda la cola.

- Cola cola = new Cola(); -> Crea una cola vacía, necesitaremos una referencia al primer y último nodo para hacer la extracciones e inserciones.
- cola.Finalize() o cola.Dispose(); -> ”Destruye” una instancia de nuestra cola, incluidos sus nodos y los datos que contenga.
- bool cola.Vacia(); -> Devolverá cierto si la cola no contiene ningún nodo.
- void cola.Encolar(Tipo elemento); -> Inserta el elemento del tipo al final de la cola y actualiza la referencia al último nodo.
- Tipo cola.Desencolar(); -> Extrae un elemento del tipo del frente de la cola, “borra” el nodo que lo contenía y actualiza la referencia al primer nodo.
- Tipo Peek() - > Coger siguiente sin desencolar

2.5. Pilas

Una pila es una lista lineal en la que todas las inserciones y supresiones se hacen en un extremo de la lista. Estas lista reciben el nombre de LIFO (Last in firs out – último en entrar primero en salir).
Podremos implementar el TAD Pila de forma vinculada o con una tabla. Pero su funcionalidad será la misma.



Su funcionalidad mínima debe, para cualquier implementación, ser la siguiente: Supongamos Tipo el tipo de los elementos que guarda la pila.

- Pila pila = new Pila() -> Crea una pila vacía, necesitaremos una referencia a la cabeza de la pila para hacer la extracciones e inserciones.
- pila.Finalize() o pila.Dispose() -> ”Destruye” una instancia de nuestra pila, incluidos sus nodos y los datos que contenga.
- bool pila.Vacia() -> Devolverá cierto si la pila no contiene ningún nodo.
- void pila.Apilar(Tipo elemento) -> Inserta el elemento del tipo en la cabeza de la pila y actualiza la referencia a la cabeza.
- Tipo pila.Desapilar() -> Extrae un elemento del tipo de la cabeza de la pila, “borrando” el nodo que lo contenía y actualizando la referencia a la cabeza.
- Tipo Peek() - > Coger siguiente sin desapilar

Ejemplo completo de una Pila de tipo Object:

```
class Nodo<T> : IDisposable
{
   public Nodo<T> Siguiente;
   public T Dato { get; private set; }
   public Nodo() { }
   public Nodo(T dato, Nodo<T> siguiente)
   {
     if (dato == null) throw new InvalidOperationException
                                                ("El dato no puede ser nulo");
     Dato = dato;
     Siguiente = siguiente;
   }
   public Nodo(Nodo<T> nodo)
   {
     if (nodo == null) throw new InvalidOperationException
                                      ("El nodo no puede ser nulo");
     Dato = nodo.Dato;
     Siguiente = nodo.Siguiente;
   }
   public void Dispose()
   {
      Dato = default(T);
      Siguiente = null;
    }
}
```

```
   class Pila<T> : IDisposable
   {
       Nodo<T> tope;
       short numElementos;

       public Pila() { tope = null; numElementos = 0; }
       public Pila(Nodo<T> t, short ne)
       {
           tope = t;
           numElementos = ne;
       }
            public void Dispose()
       {
```

```
            while(!PilaVacia) Desapilar();
        }
        public bool PilaVacia
        {
            get { return tope == null; }
        }
        public void Apilar(T dato)
        {
            Nodo<T> nodo;
            if (this.numElementos == 0) nodo = new Nodo<T>(dato, null);
            else nodo = new Nodo<T>(dato, tope);
            tope = nodo;
            this.numElementos++;
        }
        public T Desapilar()
        {
            if (PilaVacia) return default(T);
            Nodo<T> aux = tope;
            T dato = tope.Dato;
            tope = tope.Siguiente;
            numElementos--;
            aux.Dispose();
            return dato;
       }        public void Mostrar()
        {
            if (PilaVacia) Console.WriteLine("Pila vacia");
            else
            {
                Nodo<T> aux = tope;
                while (aux != null)
                {
                    Console.Write(aux.Dato + "->"); aux = aux.Siguiente;
                }
                Console.WriteLine("Null");
            }
        }
    }
```

```
class Program
{
   static void Main(string[] args)
   {
       Pila<String> p = new Pila<String>();
       ConsoleKeyInfo tecla;
       do
       {
          Console.WriteLine("\n\t\t [+]PARA APILAR    [M]PARA MOSTRAR
                                                     [-]PARA DESAPILAR");
          tecla = Console.ReadKey();
          if (tecla.Key == ConsoleKey.Add)
          { Console.WriteLine("Introduce dato a apilar");
             p.Apilar(Console.ReadLine()); }
          else if (tecla.Key == ConsoleKey.Subtract)
                                              Console.Write(p.Desapilar());
          else if (tecla.Key == ConsoleKey.M)
          { Console.WriteLine("\n\n La pila es:"); p.Mostrar(); }
        } while (tecla.Key != ConsoleKey.Escape);
```

```
    }
}
```

3. Recorrer Colecciones, patrón Iterador. La interfaz IEnumerable<T>

Tanto la interfaz genérica **IEnumerable<T>** del espacio de nombres **System.Collections.Generic** como la interfaz **IEnumerable** de **System.Collections.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.Generic
public interface IEnumerable<T> : IEnumerable
{
    IEnumerator<T> GetEnumerator();
}
// System.Collections
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

Estas interfaces incluyen un único método **GetEnumerator()**, que devuelve un **enumerador** . Un objeto cuyo objetivo es generar elementos secuencialmente, para hacer posible el recorrido de colecciones genéricas desde código no genérico, **IEnumerable<T>** hereda de su homóloga no genérica, **IEnumerable**, y por tanto debe implementar también una versión no genérica de **GetEnumerator()**, para la que generalmente sirve el mismo código de la versión genérica. Al necesitar devolver un elemento del tipo **IEnumerator** deberemos implementar también la interfaz **IEnumerator**.

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

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

```
{
    object Current { get; }
    void Reset();
    bool MoveNext();
}
```

Nuevamente, la interfaz se apoya en su contrapartida no genérica. En conjunto**, IEnumerator<T>** debe implementar los siguientes miembros:

- La **propiedad Current**, que devuelve el elemento actual de la secuencia (a dos niveles).
- El método **Reset()**, que restablece la enumeración a su valor inicial.
- El método **MoveNext()**, que desplaza el enumerador al siguiente elemento de la secuencia. Devuelve false cuando se llega al final de la secuencia.
- El método **Dispose()**,restablece la enumeración a su valor inicial y libera cualquier recurso no administrado asociado al enumerador.

La clase que implemente **IEnumerator<T>** deberá encargarse de mantener el estado necesario para garantizar que los métodos de la interfaz funcionen correctamente.

¿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, como por ejemplo los que se encuentran en la implementación típica de la ordenación mediante el algoritmo de la burbuja. En vez de eso, las secuencias implementan IEnumerable<T>, cuyo método GetEnumerator() debe producir un nuevo objeto de enumeración cada vez que es llamado.

4. 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 1.0 y 1.1 del Framework las podemos encontrar en:

- System.Collections
- System.Collections.Specialized

A partir de la versión 2.0 se usan las de:

- 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**. En realidad la interfaz **ICollection** hereda de la interfaz **IEnumerable** en que se basa la instrucción **foreach**. Y también implementan 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 siguiente:

- 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 e 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. Otro tipo sería SortedList que es una lista ordenada, pero no la veremos en este tema
https://msdn.microsoft.com/es-es/library/ms132319(VS.80).aspx

- También se dispone a partir de la v2.0 de la colección **LinkedList<T>,** que no implementa la interfaz IList y se basa en el concepto de nodo, visto anteriormente.

- 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.

Tabla Hash o Dispersión y los ***Dictionary<T>.*** 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.

4.1. Colección List<T>

Las listas son una especie de arrays dinámicos donde en cualquier momento podemos añadir o quitar elementos de cualquier parte del array. Para trabajar con ellas se utiliza el TAD List<T> de C#.

```
List<T> lista=new List<T>();
```

Al crearnos una lista de tipo Persona podremos acceder a sus miembros

```
struct Persona
{
   public string Nombre;
   public int Edad;
 }
 class Program
 {
     static void Main(string[] args)
     {
         List<Persona> personas = new List<Persona>();
         Persona p;
         for (int i = 0; i < 5; i++)
         {
             p.Nombre = Console.ReadLine();
             p.Edad = int.Parse(Console.ReadLine());
             personas.Add(p);
         }
         for (int i = 0; i <personas.Count ; i++)
              Console.WriteLine(personas[i].Nombre);
     }
}
```

1. Inicializar Lista -> Lo primero que debemos hacer es crear la lista (vacía), para luego ir añadiendo elementos. Esto se hace definiendo una variable de tipo ArrayList, y usando el operador new para inicializarla. De la siguiente manera:

List<string> lista =new List<string>();

2. Añadir datos -> Una vez tenemos la lista creada, podemos añadir datos de dos formas:

- Usando el método **Add** de la propia lista, añadiremos un elemento al final de ella:

lista.Add("hola");

- Usando el método **Insert** de la propia lista, añadiremos un elemento en la posición que indiquemos:

lista.Insert(2, "otra cadena");

A la hora de gestionar las listas, hay que tener en cuenta que las posiciones empiezan a numerarse por el cero.

3. Eliminar datos -> Para borrar un dato de la lista, tenemos también varias alternativas:

- Usando el método Remove de la propia lista, eliminamos el elemento que coincida con el que pongamos entre paréntesis:

lista.Remove("hola");

Eliminamos el elemento que "hola" de la lista. Si hubiera varios, elimina el primero que se encuentre.

- Usando el método RemoveAt de la propia lista, eliminamos el elemento de la posición que indiquemos:

lista.RemoveAt(2);

4. Modificar datos -> 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 0otro valor. Por ejemplo:

Lista[1]=”Hola”;

5. Recorrer lista -> Para recorrer los elementos de una lista, usaremos un bucle for o forech. La longitud de la lista la tendremos Count.

```
for (int i=0; i< lista.Count; i++) Console.WriteLine(lista[i]);
```

4.2. Colección Linkedlist

La diferencia entre la lista y LinkedList radica en su implementación. Mientras que la Lista es una colección basada en arrays (ArrayList), LinkedList<T> es una colección basada en el concepto de nodo-puntero (LinkedListNode) doblemente enlazada. Esto significa que cada elemento está enlazado con el anterior y el siguiente. Y los datos de cada elemento de la lista pueden estar ubicados en diferentes lugares de la memoria RAM. Aunque por este motivo también habrá más memoria utilizada para LinkedList<T> que para List<T>.

Se maneja a través de la clase LinkedList<T>, que contiene LinkedListNode <T> con sus múltiples instancias para los diferentes enlaces.

La diferencia clave es en materia de rendimiento. Por ejemplo, si se está implementando el "Insertar o Eliminar" LinkedList supera a la lista en velocidad de proceso debido a que la Lista puede necesitar ampliar el tamaño de la matriz subyacente. En cambio, el acceso es más rápido en el TAD List.

A nivel de uso de la API, los dos son más o menos lo mismo, ya que ambas implementan el mismo conjunto de interfaces ICollection, IEnumerable, etc.

Funcionalidad:

- Con los métodos **addFirst** y **addLast** se añadirán los nodos al principio o al final de la lista enlazada, respectivamente.
- **Find** devolverá el nodo correspondiente al dato que se le pase.

- **AddBefore** añadirá un nodo antes de la instancia del nodo que se pase como parámetro, mientras que **AddAfter** lo añadirá después.
- Otros métodos comunes y que su nombre indican lo que hacen son: **Remove, RemoveFirst, RemoveLast ...**

Se puede enumerar, foreach, o acceder a los elementos a partir de las propiedades First, Last, Next y Previous.

```
LinkedList<double> lista = new LinkedList<double>();
lista.AddLast(3.141);
lista.AddLast(3.1415);
lista.AddLast(3.14159);
foreach(double x in lista) Console.WriteLine(x);
Console.Write(lista.First.Next.Next.Value);//es el tercer elemento
lista.Clear();
```

4.3. Colección Stack<T>

Para trabajar con pilas se usa el TAD Stack en C#.

1. Inicializar Pila -> Lo primero que debemos hacer es crear la pila (vacía), para luego ir añadiendo elementos. Esto se hace definiendo una variable de tipo Stack, y usando el operador new para inicializarla (de forma similar a lo que hacíamos con List).

Stack<T> pila =new Stack<T>();

2. Añadir datos -> Una vez tenemos la pila creada, sólo podemos añadir datos en su tope, y para eso usamos el método Push, pasándole entre paréntesis el elemento que queremos añadir:

Pila.Push("hola");
Pila.Push("adios");

Esto nos dejaría dos elementos en la pila. Debajo quedaría "hola", y en el tope estaría "adios".

3. Eliminar datos -> Para borrar un dato de la pila, sólo podemos eliminar el que esté en el tope. Se usa el método Pop de la pila. Este método, además de borrar el elemento del tope de la pila, nos lo devuelve, con lo que podemos sacarlo por pantalla si queremos:

Console.WriteLine(pila.Pop);

4. Recorrer pilas -> Recorrer pilas no tiene sentido ya que sólo podemos insertar y borrar en el tope. Por tanto, el único elemento accesible en una pila es el tope. Para borrarlo usamos el método Pop() visto antes. Pero si queremos consultar su valor sin borrarlo, se usa el método Peek().

Console.WriteLine("El último elemento es {0}", pila.Peek);

También tenemos disponible el dato **Count**, para saber cuántos elementos hay en la pila.

```
class Program
{
  static void Main(string[] args)
  {
     Stack <Persona> p  = new
Stack<Persona>();
     Persona persona;
     for (int i = 0; i < 5; i++)
     {
        persona.Nombre=("nombre" + i);
        persona.Edad=10+i;
        p.Push(persona);
      }
      p.Pop();
      Console.WriteLine(p.Count);
      Console.WriteLine(p.Peek().Nombre);
  }
}
```

4.4. Colección Queue<T>

Para trabajar con colas se usa el TAD Queue en C#.

1. Inicializar Cola -> Lo primero que debemos hacer es crear la cola (vacía), para luego ir añadiendo elementos. Esto se hace definiendo una variable de tipo Queue, y usando el operador new para inicializarla (de forma similar a lo que hacíamos con List).

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

2. Añadir datos -> Una vez tenemos la cola creada, sólo podemos añadir datos en su parte trasera, y para eso usamos el método Enqueue, pasándole entre paréntesis el elemento que queremos añadir:

```
cola.Enqueue("hola");
cola.Enqueue("adiós");
```

Esto nos dejaría dos elementos en la cola. Al principio quedaría "hola", y al final estaría
"adiós".

3. Eliminar datos -> Para borrar un dato de la cola, sólo podemos eliminar el que esté en el principio. Se usa el método Dequeue de la cola. Este método, además de borrar el elemento del principio de la cola, nos lo devuelve, con lo que podemos sacarlo por pantalla si queremos:

```
cola.Dequeue();
```

Al igual que pasaba con las pilas, recorrer colas no tiene sentido, ya que sólo podemos acceder a la última posición para insertar, y obtener el primer elemento para borrarlo. El único elemento accesible en una cola es el primero. Para borrarlo usamos el método *Dequeue()* visto antes. Pero si queremos consultar su valor sin borrarlo, se usa el método **Peek()**, como en las pilas.

```
Console.WriteLine("El primer elemento es {0}", cola.Peek);
```

Como en el resto de las colecciones también disponemos de **Count.**

4.5. Colección Hashtable y Dictionary

Las tablas hash o diccionarios son otro tipo de colecciones con un comportamiento particular. Hasta ahora, todos los elementos de una colección tienen una posición numérica. 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 ll amada *miLista*, tenemos que poner *miLista[3]*, y si no sabemos la posición debemos echar mano de un bucle.

Para evitar el bucle y realizar un acceso directo podemos usar tablas hash, en estas, cada dato que pongamos en ella no tiene asociada una posición numérica, sino una **clave** que la identifique. Así, si sabemos qué clave tiene cada elemento, 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:

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

Si quiero consultar los datos de María, buscaré por su clave, que es su dni. También notar 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.

El funcionamiento es similar a un **diccionario**. 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 pal abra comprobando si es esa la que busco.

Las tablas hash en C# se manejan con el TAD **Hashtable,** no genérico y con el TAD **Dictionary** genérico**,** más usado.

1. Inicialización -> Para inicializar una diccionario, se define una variable de tipo Dictionary, usando el operador new.

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

2. Añadir datos-> Una vez tenemos la tabla creada, podemos añadir datos de varias formas, pero lo más normal es usar el método **Add**, indicando la clave que queremos asociar a cada elemento, y el elemento en sí. 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:

```
static void Main(string[] args)
{
  Dictionary<String, Persona> p = new Dictionary<string, Persona>();
  Persona persona;
  for (int i = 0; i < 5; i++) {
     persona.Nombre = ("nombre" + i);
     persona.Edad = 10 + i;
     Console.WriteLine("Introduce un DNI para{0}" , persona.Nombre);
     p.Add(Console.ReadLine(), persona);}
     Console.WriteLine("Introduce un DNI:");
     string  dni= Console.ReadLine();
     Console.WriteLine(p[dni].Edad);
     p.Remove(dni);
     foreach (string x in p.Keys) Console.Write(p[x].Edad);
 }
```

3. Eliminar datos -> Para borrar un dato de la lista, usamos el método Remove, pasándole entre paréntesis la clave del dato que queremos eliminar:

tabla.Remove("11223314L");

4. Modificar datos -> Si queremos cambiar el valor que tenemos almacenado en una clave (por ejemplo, cambiar la edad de "*11223314L*"), primero debemos sacar el valor a una variable (de Persona, en este caso), cambiarlo y volverlo a asignar:

Persona p = tabla["11223314L"];

5. Recorrer tablas hash ->Normalmente no recorreremos una tabla hash, porque sabremos exactamente la clave del elemento que queremos consultar. Pero por si en algún momento nos interesara recorrer y sacar cada elemento de la tabla hash para cambiarlo, o mostrarlo. Podemos usar el campo **Keys** dentro de cada tabla hash. Por ejemplo, este bucle saca las edades de todas las personas:

```
foreach (string x in p.Keys) Console.Write(p[x].Edad);
```

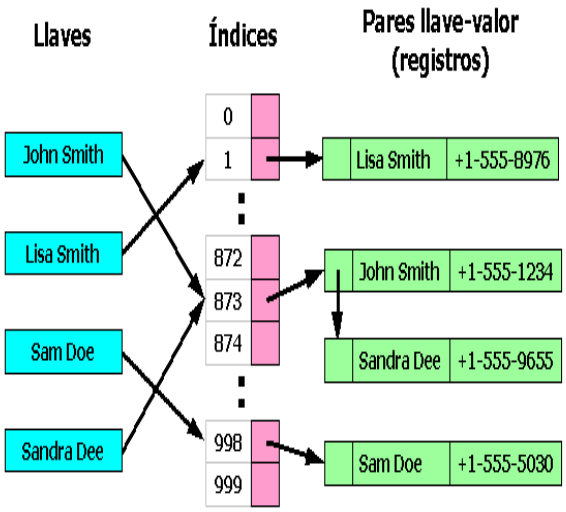
Si las sacamos por pantalla, es posible que **el orden no sea el mismo** que cuando las introdujimos, porque las tablas hash tienen **un mecanismo de ordenación diferente.**

Almacenamiento en Tablas Hash

Como hemos visto, lo que utilizan para indexar un elemento en la colección no es un entero, sino una cadena o cualquier otro objeto. Su almacenamiento se realiza sobre arrays de una dimensión, aunque se pueden hacer implementaciones multidimensionales basadas en varias claves.

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 con el tamaño del array.
C# usará el método sobreecribible 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.



Cada elemento del diccionario será un objeto de la clase, **KeyValuePair<TClave, TValor>.** Por lo que también podremos recorrer el diccionario de la siguiente manera:

```
 foreach (KeyValuePair<String, Persona>  x in p) Console.Write(x.Key+"La edad
de la persona asociada a este Dni es:" +x.Value.Edad);
```

Ejemplo completo de Dictionary:

Programa que realiza un examen sobre las capitales de la UE. Para ello, 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.
 capitalesPorPais.Add("Belgica", "Bruselas");
 capitalesPorPais.Add("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();
   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}.");
}
```

La BCL también nos proporciona el TAD ***SortedDictionary<TKey,TValue>,*** *que también re*presenta una colección de pares clave y valor, el uso el similar al Dictionary, pero es diferente su implementación ya que en este caso se ordenan por claves en su almacenamiento (internamente utiliza un árbol binario para ordenar). En este caso es importante que el Tipo TKey implemente **IComparable** correctamente para que las claves se puedan clasificar.

En cambio, como Dictionary almacena los datos usando hast, sus claves deben deribar de IEquatable<T> implementando el método Equals(), además se deberá anular GetHashCode () de manera apropiada.

Vamos a ver un ejemplo, suponiendo que en la clase Persona de los objetos, se ha sustituido el DNI de tipo string por la siguiente clase:

```
    class Dni:IEquatable<Dni>
    {
        long numero;
        char letra;
        public Dni(string numero)
        {
            if (numero.Length == 9)
            {
                this.numero = long.Parse(numero.Substring(0,8));
                this.letra = char.ToUpper(numero[8]);
```

```
            }
            else throw new FormatException("Dni invalido");
        }

        public bool Equals(Dni other)
        {
            return this.GetHashCode() == other.GetHashCode();
        }
        public override int GetHashCode()
        {
            return (numero + (long)letra).GetHashCode();
        }
    }
```

```
    class Persona
    {
        Dni dni;
        public string Nombre;
        public int Edad;
        public Persona() { }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Dictionary<Dni, Persona> p = new Dictionary<Dni, Persona>();
            Persona persona=new Persona();
            for (int i = 0; i < 5; i++)
            {
                persona.Nombre = ("nombre" + i);
                persona.Edad = 10 + i;
                Console.WriteLine("Introduce un DNI para{0}", persona.Nombre);
                p.Add(new Dni(Console.ReadLine()), persona);
            }
            Console.WriteLine("Introduce un DNI a buscar:");
            Dni dni = new Dni(Console.ReadLine());
            if(p.ContainsKey(dni)) Console.WriteLine(p[dni].Edad);
        }
    }
}
```

Si no implementáramos el IEqutable, no funcionaría el indizador ni tampoco el ContainsKey o cualquier otro método que use comparaciones.