COLECCIONES IMPLEMENTADAS COMO LISTAS ENLAZADAS

BLOQUE 1: Implementación del TAD Lista Doblemente Enlazada própio

Ejercicio 1

Crea una librería denominada **MisColecciones.dll** y agrega la siguiente definición de colección genérica vinculada...

Vamos a crear el tipo abstracto de datos (TAD) **lista doblemente enlazada**.

- **Lista**: Clase que va a encapsular mi lista.
- **Nodo**: Que se corresponderá con un nodo de la lista.

Nuestra clase lista será genérica y utilizarán un parámetro tipo T para el dato que almacenará, con la restricción de que los tipos pasados tienen que implementar el interfaz **IComparable<T>, ICloneable**.

Además mi clase Lista deberá contener información para hacer más eficiente el uso de la lista, como será una referencia al primer y último nodo de la misma, así como el valor del número de nodos que iremos actualizando.

Al ser una lista doblemente enlazada la clase nodo Nodo deberá tener las referencias al nodo siguiente y al anterior.

Al definir la lista genérica también deberemos declarar la clase Lista, con la misma restricción de interfaz que hemos hecho en la clase nodo.

Además, la clase Lista heredará del interfaz **ICloneable**, **IComparable<T>**, **IEnumerable<T>** (Este último me permitirá recorrerlas con un iterador).

Además, tanto mi clase Nodo, como la clase Lista heredarán del interfaz **IDisposable** que me obligará a implementar un método de borrado de datos Dispose(), esto es, dejar los objetos que consideremos oportunos sin ninguna referencia para que el colector de basura los libere posteriormente.

Como mínimo la clase Lista deberá de tener, los siguientes métodos:

- InsertaAlFinal de la lista.
- InsertaAlPrincipio de la lista.
- InsertaAntes de un nodo un dato.
- InsertaDespues de un nodo un dato.
- EstaOrdenada devuelve true si la lista esta ordenada (utilizará el método CompareTo de la restricción del tipo).
- InsertaOrdenado que comprueba la lista con EstaOrdenada, y si esta ordenada lo inserta.
- Borra borra un nodo que le pasemos como parámetro
- BorraAlFinal borra el último nodo, llamando a Borra
- BorraAlPrincipo borra el primer nodo, llamando a Borra
- Busca que devuelve true si encuentra el nodo con un dato pasado como parámetro y además devuelve el nodo encontrado.

- ToString que devuelve una cadena con el contenido de la lista.
- Un indizador con solo el get, que te devuelva el nodo correspondiente a un índice.

Ejercicio 2

Deberás crear un programa principal que te permita probar la lista a través de la clase Fecha (puedes reutilizar el código de ejercicios anteriores), que estará en un fichero llamado Fecha.cs.

Con este programa podremos gestionar fechas que estarán contenidas en la lista.

Hay que tener en cuenta que la clase Fecha heredará de los interfaces ya definidos en C# como son: **IComparable<T>** e **IClonable** la implementación de estos interfaces será la que utilicemos dentro de la lista para insertar un dato ordenado, buscar un dato y Clonar una lista.

BLOQUE 2: Ejercicios pilas, colas y LinkedList<T> (Lista enlazada de las BCL)

Ejercicio 1

Implementa el TAD **PilaVinculada<T>** como **envoltorio** (wrapper) del tipo de las BCL **LinkedList<T>.**

El tipo debe implementar los métodos:

- PilaVinculada<T>()
- PilaVinculada<T>(IEnumerable coleccion)
- void Apila(T dato)
- T Desapila()
- T Peek()
- bool Vacía { get; }
- int Elementos { get; }
- Implemente el interfaz IEnumerable<T>

Ejercicio 2

Implementa el TAD **ColaVinculada<T>** como **envoltorio** (wrapper) del tipo de las BCL **LinkedList<T>**.

El tipo debe implementar los métodos:

- ColaVinculada<T>()
- ColaVinculada<T>(IEnumerable coleccion)
- void Encola(T dato)
- T Desencola()
- T Peek()
- bool Vacía { get; }
- int Elementos { get; }
- Implemente el interfaz IEnumerable<T>

Ejercicio 3

Para realizar el ejercicio usa el tipo **ColaVinculada** que has creado en el ejercicio 2.

Implementa un método llamado **SumaCola** que reciba una cola de **doubles** como parámetro (ya creada previamente), y vaya extrayendo uno a uno los elementos del extremo correspondiente, y sumándolos. Al final deberá devolver la suma de todos los elementos de la cola, pero sin modificar la cola original.

Implementa un método llamado **InvierteCola** que reciba como parámetro una cola de **doubles** que sumaste anteriormente y la modifique invirtiendo el orden de los elementos. Para realizar el proceso de invertir la cola usa el tipo **Stack<T>**(Pila Genérica) implementado ya en las BCL.

Ejercicio 4

Para realizar el ejercicio usa el tipo **PilaVinculada** que has creado en el ejercicio 1.

Implementa un procedimiento llamado **CambiaPila** que reciba como parámetros una pila de **cadenas** y las **cadenas** *origen* y *reemplazo*.

Lo que deberá hacer es cambiar en la pila todas las veces que se encuentre la cadena *origen* por la cadena *reemplazo*. Para realizar el proceso de sustitución usa el tipo **Queue<T>**(Cola Genérica) implementado ya en las BCL.

COLECCIONES IMPLEMENTADAS COMO TABLAS

BLOQUE 3: Listas Genéricas en la BCL

Ejercicio 1 A implementar con listas genéricas

1. Implementa un método llamado **BorraEnteros** que reciba como parámetros una lista de enteros (que deberás crear previamente) y un número entero.
   Lo que hará será modificar la lista borrando los números que coincidan con el entero indicado.
2. Implementa una función llamada **Mezcla** que reciba como parámetro dos listas de enteros (ya ordenadas), y devuelva como resultado otra lista donde se unan las dos anteriores, pero con los números también en orden.
3. Implementa un método llamado **ImprimeInverso** que reciba como parámetros una lista de Personas y una posición (entero), e imprima por pantalla en orden inverso los nombres de las Personas de la lista desde esa posición hasta el principio. Utiliza una de las clases persona de otros ejercicios, o constrúyela nueva.
4. Implementa una función llamada **SumaRecursiva** que reciba como parámetro un lista de enteros y la sume recursivamente, devolviendo el resultado final.
5. Implementa un procedimiento llamado **OrdenaCadenas** que reciba como parámetro una lista de strings y la modifique ordenándola alfabéticamente.

Ejercicio 2

- Crea una clase Automóvil, con los datos básicos de los automóviles (marca, modelo, cilindrada, año de fabricación, etc) y los métodos necesarios para poder usarla posteriormente con comodidad.
- Crea una clase program con una serie de métodos que nos permitan trabajar con una lista genérica de automóviles.
- Necesitaremos un Método AñadirAutomovil que a partir de una lista y un automóvil, añadirá este a la lista.
- EliminarAutomovil que eliminará el automóvil con el índice X que se haya pasado como argumento.
- Crea un método que te permita encontrar en la lista, los coches con fecha de fabricación 2020, y el último coche de la lista que sea de color rojo y con fecha 2020.

BLOQUE 4: Ejercicios con tablas Hash o Diccionarios

Ejercicio 1

Utilizando la clase genérica **Dictionary<Tkey, Tvalue>** definida en en System.Collections.Generic, implementa un sencillo programa de consola que pida nombres por teclado hasta que introduzcamos la cadena "fin".

En ese momento mostraremos los nombres introducidos y cuantas veces se ha introducido cada uno. Puedes guardar los nombres como clave en el diccionario y el número de veces que se ha introducido como valor.

Recorre el diccionario 2 veces. Una con un foreach para las claves y otra con un foreach para el diccionario obteniendo pares clave valor con la siguiente clase **KeyValuePair<Tkey, Tvalue>**

Ejercicio 2

Crea una clase Polinomio que guarde los datos de un polinomio con un diccionario genérico orenado.

Por ejemplo el polinomio $9x^7 - 3x^3 - 7x + 5$ se guardaría ...

SortedDictionary<int, int> p = new SortedDictionary<int, int>()

{ {0, 5}, {1, -7}, {3, -3}, {7, 9} };

Donde los monomios se guardan en orden inverso y la “llave” es el exponente y el “valor” el coeficiente.

El constructor de la clase polinomio los recibirá como una cadena 9x7 - 3x3 - 7x + 5 de la siguiente forma y su método string ToString() lo mostrará de forma idéntica.

Redefine el operador suma binaria para polinomios para que, por ejemplo, si el usuario ha introducido estos 2 polinimios …

$$\begin{array}{r} 9x^7 - 3x^3 - 7x + 5 \\ 4x^2 - 1 \\ \hline 9x^7 - 3x^3 + 4x^2 - 7x + 4 \end{array}$$

```
SortedDictionary<int, int> polinomio1 =
new SortedDictionary<int, int>(){{ 0, 5 }, { 1, -7 }, { 3, -3 }, { 7, 9 }};

SortedDictionary<int, int> polinomio2 =
new SortedDictionary<int, int>(){{ 0, -1 }, { 2, 4 }};

SortedDictionary<int, int> polinomioSuma = polinomio1 + polinomio2;
con polinomioSuma = {{ 0, 4 }, { 1, -7 }, { 2, 4 }, { 3, -3 }, { 7, 9 }};
```

Ejercicio 3 Ampliación

Crea una clase llamada **DatosContacto** con los siguientes campos:

- El DNI de la persona (string)
- El nombre completo de la persona (string)
- La dirección completa de la persona (string)
- Telefono (string)

En la clase del programa principal:

- Crea un diccionario llamdo **AgendaContactos** con una clave de tipo string y el valor de la clase **DatosContacto**.
- Añade un método estático **CreaContactos** que devuelva el diccionario rellenado por el usuario, hasta que se introduzca un DNI vacio.
  Nota: Como clave para cada contacto utiliza su DNI.
- Añade el método estático **BorraContacto** que reciba como parámetros un DNI (como cadena) y el diccionario. Borrará deel mismo, el dato cuya clave coincida con el DNI que se le pasa.
- Añade un método estático **AñadeContacto** que reciba como parámetro un dato de tipo **DatosContacto**, y el diccionario con la agenda de contactos, y añada a la tabla la persona indicada.
- Crea **MuestraAgenda** que reciba como parámetro el diccionario y muestre su contenido.
- Crea un programa principal que te permita probar todos los métodos.