Examen Extraordinario de Programación
Curso 2012-2013

TQuery

NOTA: Si usted está leyendo este documento sin haber extraído el compactado que descargó del sitio, ciérrelo ahora, extraiga todos los archivos en el escritorio, y siga trabajando desde ahí. Es un error común trabajar en la solución dentro del compactado, lo cual provoca que los cambios **no se guarden**. Si usted comete este error y entrega una solución vacía, no tendrá oportunidad de reclamar.

En este problema se debe implementar una estructura de datos que permita representar una colección de árboles *n-arios* con las operaciones que se listan en la clase a continuación. Para ello usted debe heredar de la siguiente clase abstracta implementando los métodos abstractos y un constructor con la misma signatura.

```
public abstract class TQuery<T> : IEnumerable<T>
{
  protected TQuery(params Tree<T>[] roots) { }
  //Su implementación debe incluir un constructor con esta signatura

  public abstract int Count { get; }

  public abstract TQuery<T> Find(Func<T, bool> predicate);

  public abstract IEnumerator<T> GetEnumerator();

  IEnumerator IEnumerable.GetEnumerator()
  {
    return GetEnumerator();
  }
}
```

Su implementación de `TQuery` debe representar una estructura de datos que contenga un conjunto de árboles (basta que cada árbol se considere representado por el nodo raíz, es decir no hay que hacer copias o clonaciones de los árboles).

La operación fundamental de un `TQuery`, es decir, la operación fundamental sobre los árboles de un objeto `TQuery` es el método `Find()`. Este método "navega" por los nodos de cada árbol del `TQuery` para determinar aquellos nodos que cumplen con el delegado `predicate`. Por este motivo se dice que un objeto `TQuery` representa una consulta sobre los árboles, ya que en cada momento almacena los árboles cuyos valores cumplen con la última consulta realizada mediante este método `Find()` (inicialmente tiene los árboles que se le hayan indicado en el constructor).

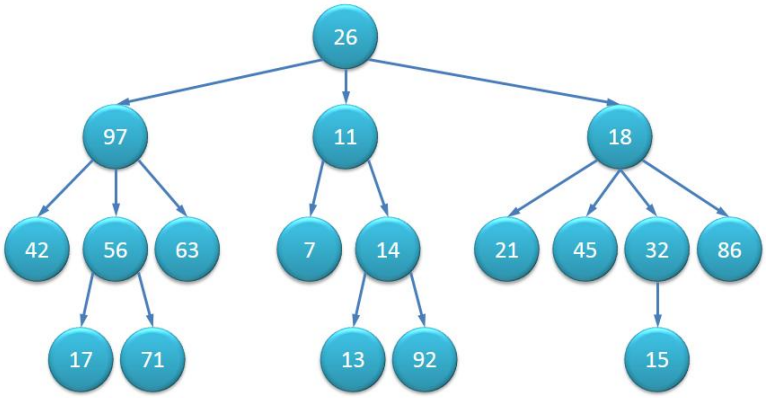
Usted debe trabajar trabajará con una definición `Tree<T>` cuyos métodos se muestran a continuación. La implementación de esta clase se le brinda en el mismo ensamblado que la clase abstracta `TQuery`, usted no necesita ver ni trabajar con el código fuente implementación de esta clase.

```
public class Tree<T>
{
  public Tree(T value, params Tree<T>[] children) { }

  public T Value { get; }

  public IEnumerable<Tree<T>> Children { get; }
}
```

NOTA Usted debe ser capaz de resolver el ejercicio sin necesidad de adicionar ninguna información a los árboles más allá de lo que permite esta clase. No puede realizar modificaciones a esta clase ni implementar su propia clase para representar árboles porque esta es la clase que se usará en el probador.

Usted puede usar esta clase para crear árboles para sus pruebas, por ejemplo, el siguiente código representa el árbol mostrado en la figura:

```
Tree<int> root = new Tree<int>(26,
  new Tree<int>(97,
    new Tree<int>(42),
    new Tree<int>(56,
          new Tree<int>(17),
          new Tree<int>(71)),
    new Tree<int>(63)),
  new Tree<int>(11,
    new Tree<int>(7),
    new Tree<int>(14,
      new Tree<int>(13),
      new Tree<int>(92))),
  new Tree<int>(18,
    new Tree<int>(21),
    new Tree<int>(45),
    new Tree<int>(32,
       new Tree<int>(15)),
    new Tree<int>(86)));
```



Un objeto `TQuery` se construye inicialmente pasándole al constructor varios parámetros árboles. Si se quiere construir un `TQuery` solo con el árbol anterior bastaría hacer

```
TQuery<T> query = new MyQuery<T>(root);
```

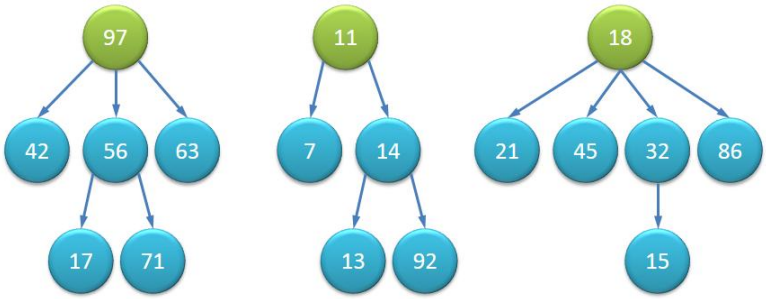
La propiedad `Count` devuelve la cantidad de árboles que se encuentran referenciados en el `TQuery`. Así para el ejemplo de arriba `query.Count` es `1`. Es decir, inicialmente su valor será igual a la cantidad de árboles pasados en el constructor; pero según se vayan haciendo aplicaciones del método `Find()`, las instancias de `TQuery` devueltas tendrán un valor igual al total de subárboles de todos estos árboles cuyo nodo raíz cumple con el criterio de búsqueda deseado.

Note que un `TQuery` es un `IEnumerable<T>` y por tanto su implementación de `TQuery` debe implementar también el método `abstract IEnumerator<T> GetEnumerator()` lo que debe dar un iterador con **los valores de los nodos raíces de los árboles** que se encuentran actualmente almacenados en la consulta, **en el mismo orden** en que fueron descubiertos por el método `Find()`. Note que antes de aplicar ningún `Find()` en el `TQuery` solo están los árboles pasados en el constructor, por lo que la enumeración solo debe devolver los valores asociados a la raíz de estos árboles.

Por ejemplo, el siguiente código construye un `TQuery<int>` asociado a tres árboles. La estructura resultante se representa en la figura. En color verde se distinguen las raíces de los árboles, que son los valores devueltos al iterar por la instancia de `TQuery` construida.

```
Tree<int> tree1 = new Tree<int>(97,
    new Tree<int>(42),
    new Tree<int>(56,
          new Tree<int>(17),
          new Tree<int>(71)),
    new Tree<int>(63));

Tree<int> tree1 = new Tree<int>(11,
    new Tree<int>(7),
    new Tree<int>(14,
      new Tree<int>(13),
      new Tree<int>(92)));

Tree<int> tree1 = new Tree<int>(18,
```

```
    new Tree<int>(21),
    new Tree<int>(45),
    new Tree<int>(32,
       new Tree<int>(15)),
    new Tree<int>(86)));

TQuery<int> query = new MyQuery<int>(tree1, tree2, tree3);
```



Recorrer este `TQuery` como `IEnumerable` debe devolver los valores `97`, `11` y `18`, que corresponden a los valores de las raíces de los árboles almacenados en el `TQuery` (resaltados en verde).

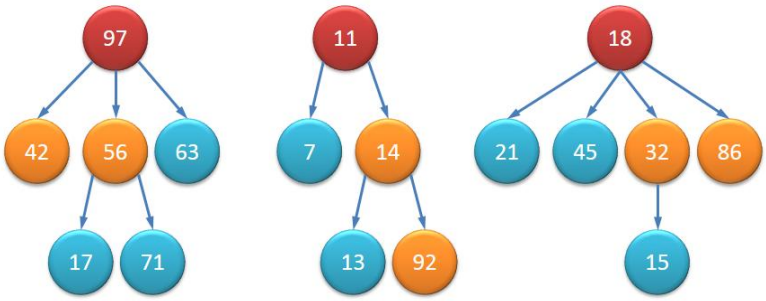
El método `Find()` devuelve **una nueva instancia** de la clase `TQuery<T>` que contiene todos los árboles y subárboles **descendientes** de los árboles almacenados en la instancia de `TQuery` a la que se le está haciendo el `Find()` y cuyo correspondiente nodo raíz cumple con el delegado `predicate` (es decir devuelve `true` al aplicarle el delegado).

El recorrido por los nodos descendientes debe hacerse en **pre-orden**. El nuevo `TQuery` resultante deberá almacenar **todos** aquellos descendientes que cumplan con la condición `predicate`, **en el mismo orden** en que son descubiertos por el recorrido en **pre-orden**, exceptuando **los nodos raíces** de los árboles almacenados en la consulta original. En las figuras correspondientes a los ejemplos siguientes, estos nodos se resaltan en rojo, para denotar que son excluidos de la aplicación del método `Find()`.

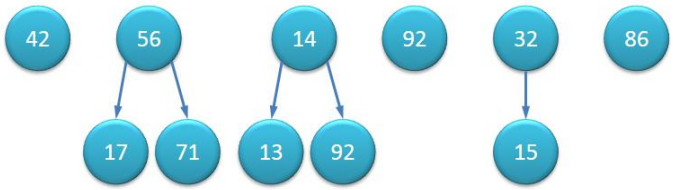
Por ejemplo, supongamos como delegado pasado como parámetro del método `Find()` un método `bool IsEven()` que determina si un número es par. Empleando este método como delegado al `TQuery` creado en el ejemplo anterior:

```
TQuery<int> queryEven = query.Find(IsEven);
```

En este punto se buscan en **pre-orden** los nodos que tienen como valor un entero par y que son descendientes de los árboles originales (se han resaltado en naranja en la figura), **exceptuando las raíces originales** (resaltadas en rojo), como muestra la figura:



La nueva instancia de `TQuery` devuelta contiene los sub-árboles cuyas raíces son los nodos encontrados:



Observe el nodo con valor `92` aparece dos veces en la consulta, una vez como sub-árbol del nodo con valor `14` porque ese nodo `14` satisfizo la consulta, y otra porque él mismo como nodo `92` también satisface la consulta (en este caso la de ser un valor par). Note que el árbol raíz con valor `18` no se tiene en cuenta aunque es par, por ser raíz en la consulta original. Note además que los árboles aparecen en el orden en son descubiertos por un recorrido en **pre-orden**, por ejemplo, el árbol con valor `92` aparece justo después del árbol con valor `14`, ya que es un descendiente.

NOTA: Aunque por razones de ilustración en las figuras los resultados parecen como figuras independientes, en su implementación del método `Find()` usted no tiene que formar físicamente nuevos árboles para dar el resultado, es decir en ninguna parte de este código tiene por qué hacer `new Tree(…)`

Implementación

Usted debe haber descargado, junto a este documento, una solución de *Visual Studio* con dos proyectos. El primero de ellos se denomina `Examen.TQuery` y consiste en una biblioteca de clases donde se brinda una clase heredera de `TQuery<T>` para que usted implemente todos los métodos necesarios sobre esta plantilla. **Es imprescindible que usted use esta clase para que el probador funcione de forma correcta.** Usted puede, por supuesto, adicionar a esta clase todos los campos, métodos, y propiedades que considere necesarios para su implementación de la solución. El constructor implementado en esta clase **no hace nada**, es también su responsabilidad cambiar su implementación según le convenga **PERO NO CAMBIE NUNCA EL NOMBRE NI LA SIGNATURA DEL RESTO DE LOS MÉTODOS Y PROPIEDADES**.

El código de esta plantilla que usted debe realmente rellenar cambiando la implementación de sus métodos es:

```
public class MyQuery<T> : TQuery<T>
{
  public MyQuery(params Tree<T>[] roots) : base(roots) { }
```

```
  public override int Count
  {
    get { throw new NotImplementedException(); }
  }

  public override TQuery<T> Find(Func<T, bool> predicate)
  {
    throw new NotImplementedException();
  }

  public override IEnumerator<T> GetEnumerator()
  {
    throw new NotImplementedException();
  }
}
```

Usted no tiene por qué lanzar ninguna excepción, pues se garantiza que todos los argumentos que se le pasarán en la prueba serán válidos. En particular se garantiza que:

- El parámetro `roots` del constructor y el parámetro `predicate` del método `Find()` **nunca** serán `null`.
- Ningún nodo del árbol pasado en el constructor tendrá valor `null`, ni lanzará **ninguna excepción** al enumerar los hijos u obtener el valor correspondiente al nodo.
- El delegado `predicate`, al ser ejecutado con un valor de tipo `T`, **nunca lanzará excepción**, incluso cuando este valor `T` sea `null`.
- Todos los nodos de los árboles pasados en el constructor tendrán valores **diferentes**. Sin embargo, cuando se aplique el método `Find()` es posible que aparezcan **nodos repetidos**, como en el ejemplo anterior.

Usted no necesita validar ninguna de estas circunstancias.

Tenga en cuenta de que el resultado de un `Find()` sea **vacío** es algo **totalmente válido**. Para el objeto `TQuery` resultante la propiedad `Count` devolverá `0`, y el intento de recorrer el `TQuery` como `IEnumerable` no devolverá ningún valor, pero **no lanzará ninguna excepción**. Simplemente es un `IEnumerable` **vacío**. La aplicación del método `Find()` sobre un `TQuery` vacío dará también por supuesto como resultado **una consulta vacía**.

El segundo proyecto que usted verá en la solución es una aplicación de consola que muestra el ejemplo incluido en este documento. Aunque este ejemplo prueba algunos de los casos extremos contemplados en el ejercicio, como usted debe estar acostumbrado, **no es suficiente** para garantizar que su implementación está correcta. Asegúrese de adicionar todas las pruebas que considere necesarias.

Ejemplos

A continuación se muestra un ejemplo, que también está incluido en la aplicación de consola incluida en la solución.

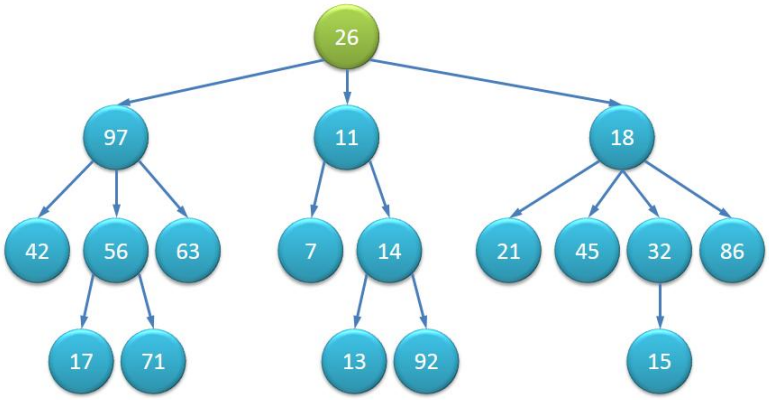
En este ejemplo se usa el siguiente árbol, mostrado anteriormente en el documento:

```
Tree<int> root = new Tree<int>(26,
  new Tree<int>(97,
    new Tree<int>(42),
    new Tree<int>(56,
          new Tree<int>(17),
          new Tree<int>(71)),
    new Tree<int>(63)),
  new Tree<int>(11,
    new Tree<int>(7),
    new Tree<int>(14,
      new Tree<int>(13),
      new Tree<int>(92))),
  new Tree<int>(18,
    new Tree<int>(21),
    new Tree<int>(45),
    new Tree<int>(32,
       new Tree<int>(15)),
    new Tree<int>(86)));
```

Se construye una consulta inicial formada **en este caso** por solo este árbol:

```
TQuery<int> query = new MyQuery<int>(root);
```

En la figura se representa el resultado de construir esta consulta, que inicialmente solamente contiene a la raíz del árbol pasado en el constructor (resaltada en verde).



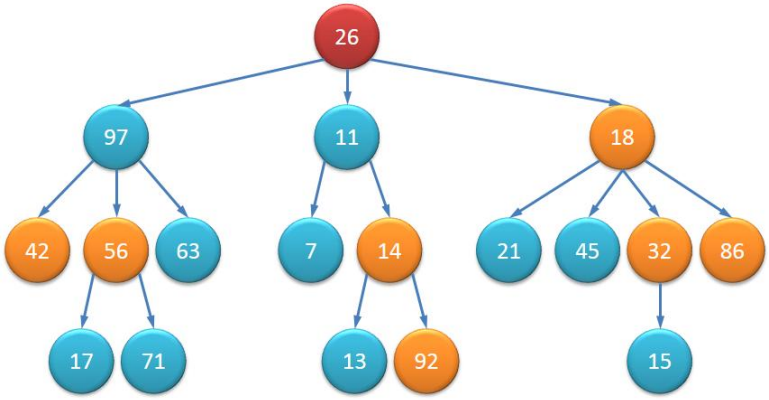
Inicialmente la propiedad `Count` debe devolver el valor `1`.

```
Console.WriteLine("Count es {0}", miQuery.Count);
// Imprime 1
```

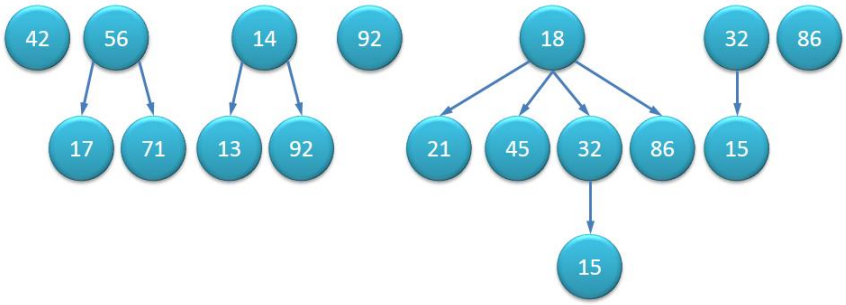
Se aplica el método `Find()` con un predicado que comprueba si los valores son pares:

```
queryEven= query.Find(IsEven);
```

La siguiente figura ilustra los subárboles que cumplen con este predicado (resaltados en color naranja). Note que el nodo raíz no se tiene en cuenta, a pesar de cumplir la condición (resaltado en rojo).



Por lo tanto el `TQuery` resultante debe estar formado por los árboles:

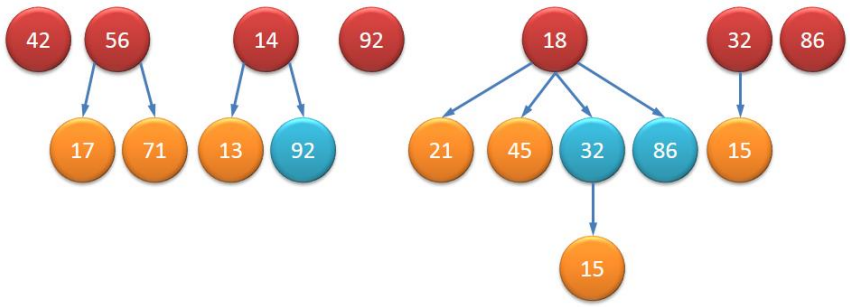


Note que el árbol con valor `32` (con sus descendientes) aparece dos veces, porque a su vez era descendiente del árbol con valor `18` que también cumple con ser par. Igualmente sucede con el árbol con valor `92`, y el árbol con valor `86`.

Si a este `queryEven` se le aplica ahora el método `Find()` con el delegado `bool IsOdd()` que determina si un número es impar):

```
queryOdd = queryEven.Find(IsOdd);
```

Cumplirían con el mismo los nodos coloreados en naranja en la figura a continuación (en color naranja).



Por tanto el resultado `queryOdd` es el objeto representado en la figura:

Observe que en este caso son siete árboles que todos son hojas. Note que el árbol con valor `15` aparece dos veces, porque había dos subárboles con valor `15` en el `queryEven` al que se le aplicó el `Find()`.

Si ahora a este `queryOdd` se le hace:

```
queryZero = queryOdd.Find((int j) => j == 0);
```

El resultado sería un `TQuery` **vacío** ya que ningún nodo en `queryOdd` tiene valor `0`.

MUY IMPORTANTE: Aunque en los ejemplos se ha utilizado el tipo `int` como el tipo del valor de los nodos, recuerde que su solución debe ser genérica y no debe depender del tipo concreto del parámetro genérico. De modo que las pruebas de la evaluación se podrán realizar con árboles en los que el tipo de los nodos no sea `int`. Digamos por ejemplo que el tipo puede ser `Fecha` y un predicado que se le pase a `Find()` ser la función `((Fecha f) => f.Mes == 12)`. Asegúrese de hacer pruebas con algún tipo diferente de `int`, para probar que su solución es genérica.