14 ÁRBOLES

En analogía con el mundo real una forma conveniente de organizar y estructurar la información es de manera jerárquica en donde cada nivel de la jerarquía representa a cualidades, propiedades, responsabilidades, funciones, atribuciones, etc., que son comunes a todos los niveles por debajo en la jerarquía. En términos matemáticos y computacionales se denomina **árbol** a una estructura que expresa la organización de la información de manera jerárquica. Ese es el caso por ejemplo cuando hablamos de jerarquía de tipos en el Capítulo 10 sobre herencia, del árbol genealógico de una familia (Figura 14-1), de la definición de la cadena de mando en una organización[1], o de la forma que organizamos la estructura de carpetas y archivos en nuestra computadora.

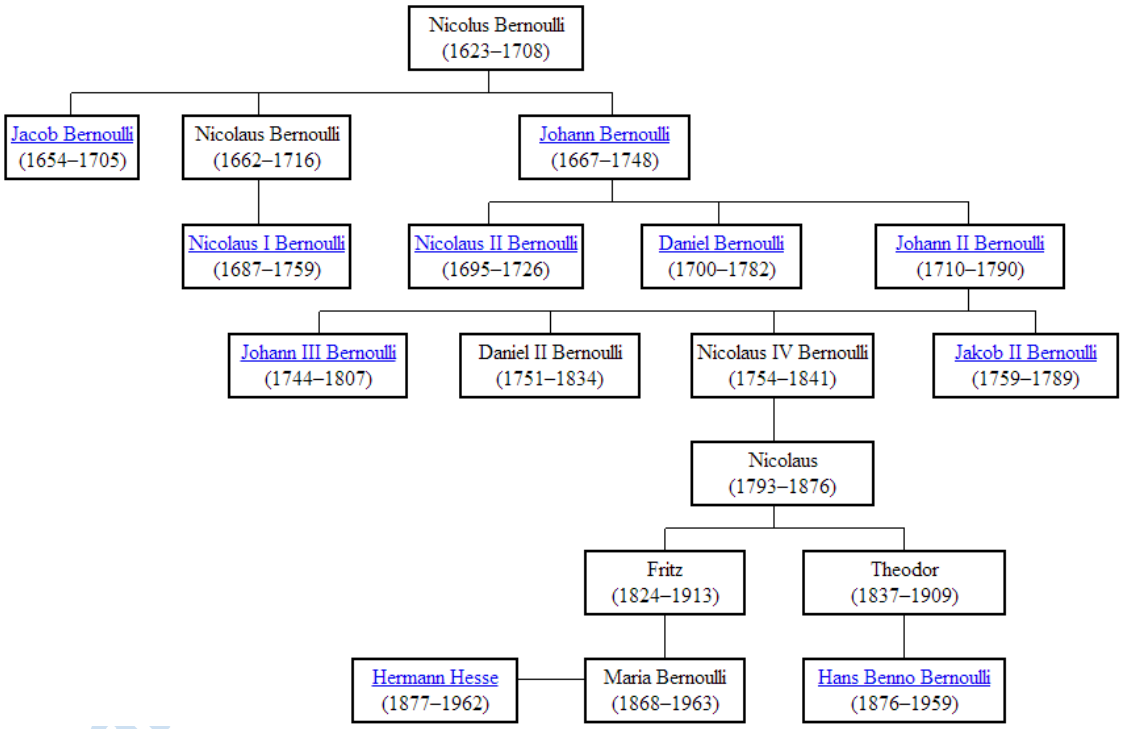

Figura 14-1 Árbol genealógico de la célebre familia Bernoulli, reconocida mundialmente por sus grandes aportes matemáticos durante varias generaciones.

Los árboles son una de las representaciones más importantes en Computación por su forma sencilla y la vez muy útil para estructurar los datos que a la vez son muy útiles para diseñar algoritmos eficientes y para formar parte de otras estructuras de datos más complejas. Existen varios tipos de árboles y varias propiedades que estos cumplen, pero estudiarlas todas se sale del alcance de este libro. A diferencia de conceptos más simples como los de pilas, colas y diccionarios (que se estudian en el Capítulo 13) los árboles

[1] Y de ahí tal vez nuestro rechazo a la palabra jerarquía.

tienen una gran variedad de conceptos y propiedades; que paradójicamente no tiene sentido a nivel de programación querer resumirla en una jerarquía universal de tipos. Por esta razón es que en las bibliotecas de .NET no pretenda encontrar siempre un tipo de árbol que encaje a la medida con las necesidades de su aplicación. En este capítulo se estudiarán algunos de los tipos de árboles más básicos y cómo puede ser su implementación computacional utilizando C#.

En el epígrafe siguiente se resumen las definiciones más básicas sobre árboles.

14.1 DEFINICIONES SOBRE ÁRBOLES

14.1.1 ÁRBOL CON RAÍZ

Un **árbol con raíz** es un conjunto finito de objetos llamados **nodos** (recuerde del Capítulo 13 de Estructuras de Datos que un nodo tiene información asociada y una posible referencia a otro(s) nodo), entre los cuales uno se distingue como **raíz**. Siguiendo las referencias a cuáles nodos se refiere un nodo, a partir de la raíz se encuentra definida una relación de paternidad que establece la estructura jerárquica. Se le llama **hoja** a todo nodo que no sea padre de ningún otro nodo (es decir que no se refiera a ningún otro nodo).

Por simplicidad, en lo adelante se utilizará siempre el término **árbol** para referirse a un **árbol con raíz**, aunque existen otras definiciones como la de **árbol libre** que se estudia en *Teoría de Grafos*.

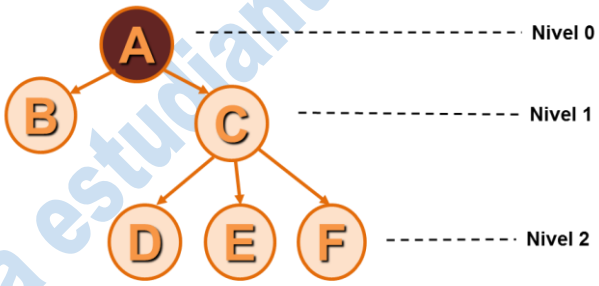

Figura 14-2 Ejemplo de árbol con raíz A

El árbol que se muestra en la Figura 14-2 puede representarse matemáticamente utilizando notación de conjuntos $T = \{\langle a, b\rangle, \langle a, c\rangle, \langle c, d\rangle, \langle c, e\rangle, \langle c, f\rangle\}$ donde queda explícitamente definida la relación de paternidad.

Se dice que la raíz de un árbol está en el **nivel 0** y los hijos de esta en el **nivel 1**, y así sucesivamente, se enumeran los distintos niveles del árbol hasta llegar a las hojas.

14.1.2 ALTURA Y PROFUNDIDAD DE UN ÁRBOL

Se define la **altura del árbol** como el máximo nivel en que se encuentra alguna hoja del árbol (la "distancia" de una hoja a la raíz). Se dice a su vez que un nodo está a **profundidad** d si se encuentra en el nivel d del árbol.

Se dice entonces, por ejemplo, que la altura del árbol de la Figura 14-2 es 2 y que el nodo C se encuentra a profundidad 1.

Pero la definición siguiente nos da una definición recursiva para árbol con raíz, que es equivalente a la definición anterior, y además es útil desde el punto de vista de programación.

14.1.3 DEFINICIÓN RECURSIVA DE ÁRBOL CON RAÍZ

Un **árbol con raíz** es un conjunto finito de objetos llamados **nodos** tal que:

1. Existe un nodo especial llamado **raíz**
2. El resto de los nodos se particionan en $k \geq 0$ conjuntos disjuntos $T_1, T_2, \ldots, T_k$, donde cada uno de estos conjuntos es un **sub-árbol** de la raíz. Si $k = 0$ el nodo es una **hoja**.

Siguiendo esta idea la Figura 14-3 muestra una interpretación gráfica del árbol de la Figura 14-2. Como se puede observar, un árbol sería un nodo raíz más un conjunto de sub-árboles, donde cada sub-árbol, es también a su vez un árbol que cumple esta definición.

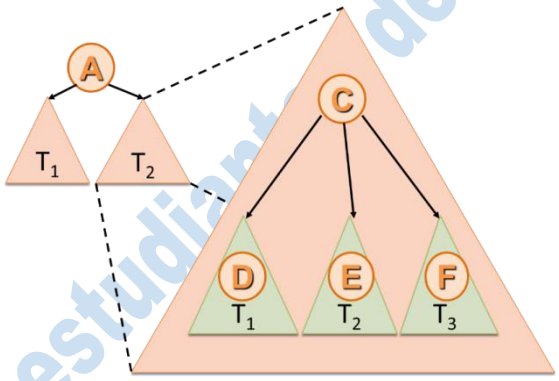

Figura 14-3 Interpretación recursiva de un árbol, donde cada sub-árbol se analiza como un árbol independiente

Bajo esta definición, un árbol tiene que tener al menos un nodo que hace de raíz. No obstante, en algunos problemas es conveniente la consideración del concepto de **árbol nulo** o **nodo nulo** (*dummy node*) como se verá en la definición siguiente.

14.1.4 ÁRBOL POSICIONAL

Un árbol posicional es aquel en el cual se define un orden o posición estructural para cada hijo de un nodo. Esto se logra etiquetando los hijos con un número entero. Visualizado gráficamente el hijo más a la izquierda es el hijo 0 y las posiciones se consideran entonces de izquierda a derecha.

Se dice que el *i-ésimo* hijo es un **árbol nulo** o **nodo nulo**, si no existe un hijo para la posición *i-ésima*.

La diferencia entre árbol y árbol posicional es sutil, y para notar su diferencia es conveniente la utilización del concepto de **árbol nulo** o **nodo nulo**.

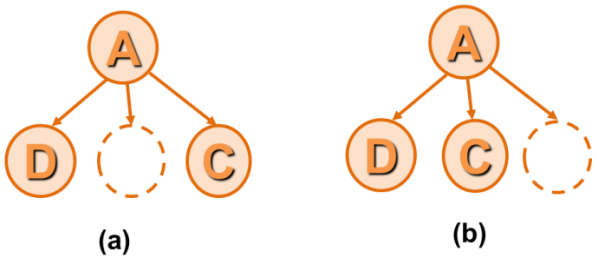

Figura 14-4 Representación del concepto de árbol posicional. El árbol (b) es diferente del árbol (a)

Como se muestra en la Figura 14-4, el árbol posicional respeta, estructuralmente, la posición de los hijos. Computacionalmente, el concepto de árbol posicional puede ser conveniente, y en este caso, se interpretaría un árbol con tres nodos hijos, donde el nodo "central" sería el valor `null` para el ejemplo (a).

14.1.5 GRADO DE UN NODO

Se define **grado** de un nodo como la cantidad de hijos que tiene. Un árbol posicional se dice es **árbol k-ario** si cada nodo no hoja tiene a lo sumo k hijos o que cada nodo tiene grado k si consideramos el concepto de nodo nulo.

14.1.6 ÁRBOL K-ARIO COMPLETO

Un **árbol k-ario** se dice **completo** si todas las hojas se encuentran a la misma profundidad y todos los nodos interiores tienen grado k. Se dice que un árbol k-ario es **semi-completo** cuando las hojas se encuentran a una profundidad d o $d-1$ y todos los nodos interiores tienen grado k.

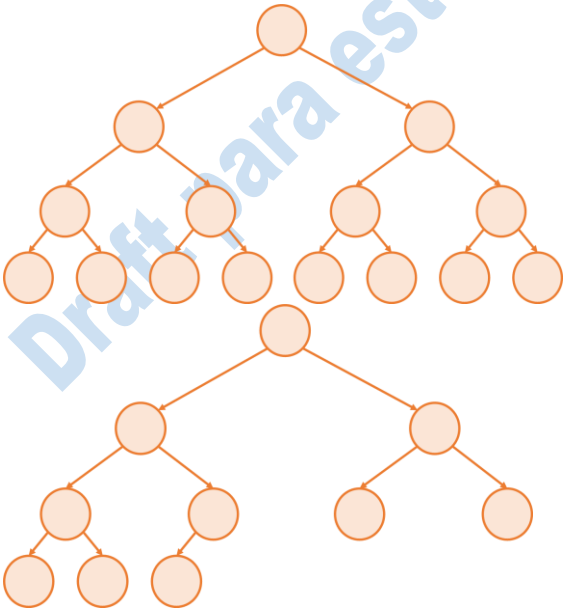

Figura 14-5 Árbol k-ario con k=2. A la izquierda un árbol 2-ario completo y a la derecha un árbol 2-ario semi-completo.

14.1.7 ÁRBOL BINARIO

Árbol **2-ario** en donde al hijo en la posición 0 se le llama **hijo izquierdo** y al hijo en la posición 1 se le llama **hijo derecho** y en donde éstos pueden ser `null` según sea conveniente.

Propiedades de un árbol binario completo:

- Un árbol binario completo de altura h tiene $2^h - 1$ nodos interiores.
- La altura de un árbol binario completo es $\log_2\left(\frac{n+1}{2}\right)$ donde n es la cantidad de nodos del árbol.
- La diferencia entre las hojas y los nodos interiores en un árbol binario completo es 1.

Para demostrar estas propiedades, empecemos por contar los nodos por cada nivel del árbol. En el primer nivel (nivel 0) está la raíz solamente, en el segundo nivel (nivel 1) existen 2 hijos de la raíz, en el tercero habrán 2^2 nodos y así sucesivamente, hasta llegar al nivel h en donde hay 2^h nodos. Por tanto:

$$\underbrace{1 + 2 + 2^2 + \cdots + 2^{h-1}}_{nodos\ interiores} + \underbrace{2^h}_{hojas} = n$$

$$nodos\ interiores = 1 + 2 + 2^2 + \cdots + 2^{h-1} = \sum_{i=0}^{h-1} 2^i = \frac{2^h - 1}{2 - 1} = 2^h - 1$$

$$n = 1 + 2 + 2^2 + \cdots + 2^h = \sum_{i=0}^{h} 2^i = \frac{2^{h+1} - 1}{2 - 1} = 2^{h+1} - 1$$

$$n + 1 = 2^{h+1} = 2 \cdot 2^h$$

$$\frac{n+1}{2} = 2^h$$

$$h = \log_2\left(\frac{n+1}{2}\right) \blacksquare$$

Esta relación logarítmica entre la altura del árbol y la cantidad de nodos es útil para hacer búsquedas en estos árboles como se verá más adelante.

14.2 IMPLEMENTACIÓN COMPUTACIONAL

Un programador tiene que tener en cuenta, antes de implementar una estructura de datos, cuáles son los objetivos de la misma: ahorrar memoria o ejecución más rápida. La cantidad de elementos (n) , por lo general nodos, que almacenará la estructura es un aspecto determinante, pero también será muy importante tener en cuenta si el árbol será una estructura que una vez construida no variará (porque se usa solo para consultar la información que representa) o si es una estructura que puede variar (y con qué frecuencia) durante la propia ejecución de la aplicación.

Existen dos vías fundamentales para implementar árboles:

1. Implementación por arrays
2. Implementación por clases siguiendo una definición recursiva

Ambas tienen ventajas y desventajas que se analizarán en este epígrafe.

14.2.1 REPRESENTACIÓN DE UN ÁRBOL K-ARIO UTILIZANDO UN ARRAY

Como el lector sabe (Capítulo 7), los arrays son estructuras fijas; una vez que se crean no se pueden extender, o tiene un sobrecosto porque implica pasar pasa por el proceso de crear un nuevo array con mayor capacidad y copiar hacia él los valores del array original (lo que hace el propio .NET en la implementación del tipo `List<T>`.lista del Capítulo 7).

La representación por array de un árbol, se basa en "empaquetar" el árbol en un solo array fijo, y la navegación por la jerarquía se realiza a través de fórmulas matemáticas para calcular la posición dentro del array que corresponde a determinado nodo del árbol.

La estructura del array para un árbol *k-ario* es la siguiente:

- En la posición 0 del array se almacena la raíz del árbol k-ario
- A partir de la posición 1 se almacenan de forma contigua los niveles del árbol.
- Siempre se reserva, para un árbol de altura h un array de tamaño $2^{h+1} - 1$ independientemente de si el árbol es completo o no.

Dado una posición i en el array, se utilizan las siguientes fórmulas para moverse dentro del árbol *k*-ario siendo $1 \leq m \leq k$ la posición (hijo) al cual se quiere navegar:

$$PADRE(i) = \left\lfloor \frac{i+1}{k} \right\rfloor - 1$$

$$HIJO_m(i) = k \cdot i + m$$

La Figura 14-6 muestra un ejemplo de árbol binario, representado con un array. Note para esta forma de implementación la conveniencia de manejar el concepto de nodo nulo, como el caso del nodo hijo izquierdo del nodo C, y se representa con valor `null` en el array.

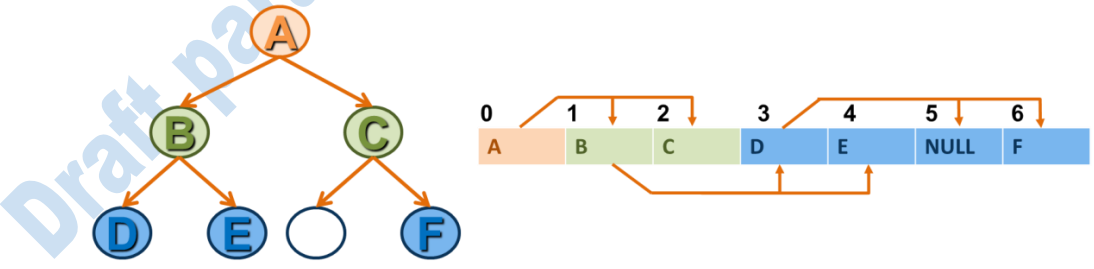

Figura 14-6 Representación por arrays de un árbol binario

Se navega entonces en los árboles manipulando numéricamente los índices del array. Por ejemplo, si se quiere, obtener el hijo derecho E del nodo B, como B se encuentra en el índice 1 del array y el árbol es binario ($k = 2$), se aplica la fórmula siguiente $HIJO_2(1) = 2 \cdot 1 + 2$ y se obtiene el índice 4 que efectivamente corresponde con E. Si se quiere

obtener el padre de B, se aplica $PADRE(1) = \left\lfloor \frac{1+1}{2} \right\rfloor - 1$ que devuelve el índice 0, que corresponde con el nodo A.

Esta representación es conveniente de usar cuando:

- La estructura del árbol no varía frecuentemente y éste no tiende a crecer (se sabe a priori la cantidad máxima de nodos a almacenar y se reserva en consecuencia el tamaño del array al crear éste).
- Se sabe que el árbol será completo o semi-completo (lo que implica que no habrá mucho espacio reservado ociosamente en el array)

La ventaja de esta representación es la forma compacta para almacenar un árbol si este es completo o semi-completo (lo que puede ser un útil ahorro de memoria si el árbol es muy grande). Note que un árbol degradado (aquél dónde cada nodo tiene un solo hijo, ver Figura 14-7) tendría memoria subutilizada.

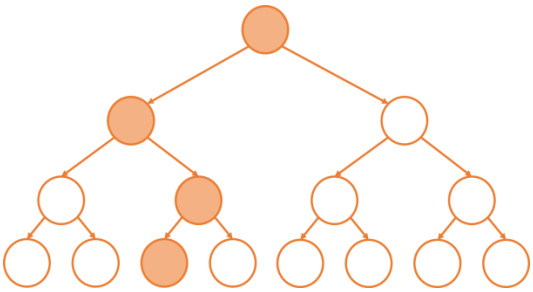

Figura 14-7 Ejemplo de Árbol degradado. La cantidad de nodos null es mucho mayor que la de nodos con valor.

Una posible implementación en C# se muestra en el Listado 14-1

```
public class ArrayTree<T>
{
  private readonly int _degree;
  private readonly T[] _array;
  public ArrayTree(int degree, int height)
  {
    _degree = degree;
    if (height<0 || degree<=2)
      throw new ArgumentException("La altura o el grado no es válido");

     _array = new T[(int)(Math.Pow(2, height + 1f) - 1f)];
  }

  public int GetParent(int i)
  {
    return (int) (Math.Floor((i + 1f)/_degree) - 1f);
  }

  public int GetChild(int parentPosition, int childPosition)
  {
    if (childPosition <= 0 || parentPosition < 0)
      throw new ArgumentException("La posición no es válida");
```

```
    return _degree*parentPosition + childPosition;
  }

  public T this[int index]
  {
    get { return _array[index]; } set { _array[index] = value; }
  }
  public int Size
  {
    get { return _array.Length;
  }
}
```

Listado 14-1 Implementación de árbol basada en arrays

Como se puede apreciar, el tipo `ArrayTree<T>` (Listado 14-1), que representa a un árbol k-ario, utiliza genericidad para expresar el tipo de valor que se quiere guardar en los nodos del árbol (es decir la lógica de la representación de los índices del árbol permite expresar la relación entre los nodos y el valor `T` de un nodo expresa la información propia de cada nodo.

En el constructor hay que indicar el grado *k* del árbol y la altura máxima para reservar reserva en consecuencia el espacio de memoria requerido. En este ejemplo hemos omitido algunas verificaciones en los valores de los parámetros para simplificar el espacio del listado con el código. Por ejemplo, en el método `GetParent` habría que verificar que *i* tenga un valor válido, si no este método debe devolver alguna excepción con índice fuera de rango para el array.

Los métodos `GetParent` y `GetChild` reciben una posición en el array y devuelven nuevas posiciones en dependencia si se navega al padre o hacia un hijo determinado. La forma de almacenar u obtener un valor en árbol es a través del *indizador de clase*, que recibe un índice y accede al array interno.

En el Listado 14-2 se muestra el código cliente de la clase `ArrayTree<T>` para construir el árbol correspondiente a la Figura 14-6.

```
// Creando un árbol binario de altura 2 (3 niveles)
var t = new ArrayTree<char>(2, 2);

// Estableciendo el valor de la raíz
t[0] = 'A';

// Obteniendo el hijo izquierdo de la raíz
var bNode = t.GetChild(0, 1);
t[bNode] = 'B';
// Actualizando los valores de los hijos del nodo B
t[t.GetChild(bNode, 1)] = 'D';
t[t.GetChild(bNode, 2)] = 'E';

// Obteniendo el hijo derecho de la raíz
var cNode = t.GetChild(0, 2);
t[cNode] = 'C';
// Actualizando los valores del hijo del nodo A
t[t.GetChild(cNode, 2)] = 'F';
```

Listado 14-2 Ejemplo de inicialización de un árbol utilizando la implementación por array

Esta implementación con arrays si bien eficiente para acceder a los elementos del árbol, descansa en el programador la responsabilidad de manejar adecuadamente los índices para acceder o modificar los valores de los nodos del árbol o alterar la relación entre los mismos.

En la sección siguiente se verá una implementación basada en referencias, es decir un nodo padre tiene las referencias sus nodos hijos basándose en una definición recursiva del tipo árbol.

No obstante, hay que resaltar que lo que se pierde en diseño y limpieza de código se gana en eficiencia y ahorro de memoria. Ésta dicotomía entre "programar con estilo" y "programar eficiente" es algo que un programador debe tener en cuenta. No hay un consejo absoluto, aunque a lo largo del libro hemos abogado por la sencillez, en la solución de problemas reales hay que intentar llegar a un punto en donde converjan ambos extremos, según sea necesario para el problema en concreto que se está resolviendo.

14.2.2 REPRESENTACIÓN DE UN ÁRBOL K-ARIO UTILIZANDO LA DEFINICIÓN RECURSIVA DE UNA CLASE

La idea consiste en definir una clase `Tree<T>` que modele un nodo y que a la vez se use a sí misma:

```
public class Tree<T>
{
    public T Value { get; set; }
    public Tree<T>[] Childrens { get; protected set; }
    ...
}
```

Básicamente, se tendrá una propiedad `Value` de tipo `T` genérico que almacene el valor que se desea guardar en el nodo del árbol y que cuente con un array `Tree<T>[]` que contenga la referencia a los posibles hijos del árbol. Recuerde que se está modelando un árbol k-ario por lo que cada nodo tendrá exactamente asociado un array con k posiciones, las cuales pueden están vacías (`null`) o tener alguna referencia a otro sub-árbol (que es también de tipo `Tree<T>`).

La clase `Tree` podría tener los métodos que se muestran en el Listado 14-3

```
public class Tree<T>
{
  public T Value { get; set; }
  public Tree<T>[] Childrens { get; protected set;
  public Tree(int degree, T value)
  {
    Childrens = new Tree<T>[degree];
    Value = value;
  }
  public bool IsLeaf
  {
```

```
    get { return Childrens.All(x => x == null); }
  }
  public virtual void Attach(int position, Tree<T> subtree)
  {
    Childrens[position] = subtree;
  }
  public virtual void Attach(params Tree<T>[] subtrees)
  {
    for (int i = 0; i < Math.Min(Childrens.Length, subtrees.Length); i++)
    {
        Childrens[i] = subtrees[i];
    }
  }
  public virtual void Remove(int position)
  {
    Attach(position, null);
  }
  ...
}
```

Listado 14-3 Implementación de algunos métodos de la clase Tree

Note que el constructor se inicializa el array `Childrens` con las k posiciones que define el grado del árbol y se almacena en `Value` el valor que se desea guardar.

Como se puede observar, la propiedad `IsLeaf` devuelve `true` si todos los hijos son `null` es decir si es una hoja. Eso se logra con el método extensor `All` que devuelve `true` si se cumple la condición `x==null` para todo valor del array `Childrens`.

El método `Attach` permite ir construyendo el árbol. Existen dos sobrecargas, una adjunta un sub-árbol dado en una posición específica, y la otra sobrecarga, utiliza un `params` y adjunta todo un array de subárboles, verificando que coincidan la cantidad de hijos con la cantidad de parámetros. Le queda propuesto al lector incluir estas verificaciones y lanzar excepción según convenga.

También se podría tener la siguiente sobrecarga del constructor

```
public Tree(T value; params Tree<T>[] subtrees)
```

para si en el momento de la construcción del árbol ya se conocen los que van a ser los hijos.

El método `Remove`, lo que hace es llamar al método `Attach`, pasando un `null` como sub-árbol, logrando el comportamiento de "remover una rama".

Ciertamente cambiar los hijos de un nodo (operación `Attach` y operación `Remove`) puede llevar una memoria ocupada innecesariamente que es la que puede quedar al "desconectar" el nodo de sus antiguos hijos. Es el recolectos de basura de .NET quien se encarga de reutlizar esta memoria cuando lo necesite.

Estos métodos se han especificado `virtual` para permitir, que futuras clases herederas los puedan redefinir según sea conveniente.

El árbol de la Figura 14-5 se pudiera construir entonces con el código del

```
// Creando el sub-árbol con raíz 'B'
var bNode = new Tree<char>(2, 'B');
//...
bNode.Attach(new Tree<char>(2, 'D'), new Tree<char>(2, 'E'));
//...

// Creando el sub-árbol con raíz 'C'
var cNode = new Tree<char>(2, 'C');
cNode.Attach(null, new Tree<char>(2, 'F'));

//...
//Creando el árbol con raíz 'A' y adjuntando 'B' y 'C' como hijos
var root = new Tree<char>(2, 'A');
root.Attach(bNode, cNode);
```

Listado 14-4 Creando un árbol

Si en el momento de la construcción del árbol ya conocemos cómo serán todas las componentes del mismo entonces se puede usar la sobrecarga del constructor para hacerlo directamente en forma anidada en la propia construcción (Listado 14-5)

```
var a = new Tree<char>('A',
            new Tree('B',
              new Tree('D',null),
              new Tree('E',null)
            )
            new Tree('C',
              null,
              new Tree('F',null)
            )
        );
```

Listado 14-5 Construcción directa del árbol de la Figura 14-5

Como se puede apreciar, este código es más legible, en comparación con la implementación utilizando un array y se pueden apreciar claramente la relación de paternidad, limitando así los errores de programación.

En la definición recursiva de árbol se gana en estilo, sencillez y claridad del código y en que por tanto es menos propensa a cometer errores de programación; en la definición basada en arrays se gana en tener menos particionamiento y en más eficiencia de acceso a la información. Ésta dicotomía entre "programar con estilo" y "programar eficiente" es algo que un programador debe tener en cuenta. Aunque por razones didácticas a lo largo del libro hemos abogado por la sencillez, no hay un consejo absoluto ni una receta única. En la solución de problemas reales, sobre todo en el uso de árboles por la gran diversidad de escenarios en los que se pueden aplicar, hay que intentar llegar a un punto en donde se concilien ambos extremos, según

14.2.3 IMPLEMENTACIÓN DE ÁRBOL BINARIO HEREDANDO DE ÁRBOL

Se puede utilizar la herencia para definir un árbol binario basado en el tipo `Tree<T>` tal que se reutilice la funcionalidad ya implementada como se muestra en el Listado 14-6.

```
public class BinaryTree<T> : Tree<T>
{
  public BinaryTree<T> Left
```

```
  {
    get { return Childrens[0] as BinaryTree<T>; }
    set { Childrens[0] = value; }
  }
  public BinaryTree<T> Right
  {
    get { return Children[1] as BinaryTree<T>; }
    set { Childrens[1] = value; }
  }

  public BinaryTree(T value)
      : base(value, null, null)
  {

  }
  public BinaryTree(T value, BinaryTree<T> left, BinaryTree<T> right)
      : base(value, left, right)
  {

  }

    ...
}
```

Listado 14-6 Implementación de un árbol binario utilizando herencia

La clase heredera ha añadido las propiedades `Left` y `Right` para acceder directamente de modo simple a la posición 0 y a la posición 1 (respectivamente) del array `Childrens`. Note que hay que realizar una la coerción `Childrens[0] as BinaryTree<T>` y `Childrens[1] as TreeBinaryTree<T>` (porque recuerde que aunque sabemos que su valor es de tipo `BinaryTree<T>`, `Childrens` está declarado estáticamente de tipo `Tree<T>`).

En cuanto a los métodos de modificación del árbol, hay que tener en cuenta que ahora se quieren insertar sub-árboles de tipo `BinaryTree<T>` y no de tipo `Tree<T>`. Por tanto, habrá que redefinir los también los métodos `Attach` para verificar que el sub-árbol que se pasa es de tipo `BinaryTree<T>`. Es por ello que en la implementación del Listado 14-7 se verifica con el operador `is`, si un objeto es del tipo especificado.

```
  public override void Attach(int position, Tree<T> subtree)
  {
    if (position < 0 || position > 1)
      throw new ArgumentException("La posición tiene que ser 0 o 1");
    if (!(subtree is BinaryTree<T>))
      throw new ArgumentException("El árbol tiene que ser Binario ");
    base.Attach(position, subtree);
  }

  public override void Attach(params Tree<T>[] subtrees)
  {
    if (subtrees.Any(subtree => !(subtree is BinaryTree<T>)) || subtrees.Length
> 1)
      throw new ArgumentException("El árbol tiene que ser Binario");
    base.Attach(subtrees);
  }
```

Listado 14-7 Reimplementación del método Attach para BinaryTree

En la sobrecarga en que se pasa un array de sub-árboles como parámetro, para verificar que todos sus elementos sean de tipo `TreeBinaryTree<T>` de una forma sencilla y legible, se ha utilizado la notación funcional (Capítulo 12) con el método extensor `Any` que devuelve `true` si existe al menos un elemento que cumpla la condición de que `!(subTree is TreeBinaryTree<T>)`, es decir, si existe algún elemento que no sea de tipo `TreeBinaryTree<T>` .

Realmente esta definición de `TreeBinaryTree<T>` se ha construido extendiendo el tipo `Tree<T>` para ilustrar el funcionamiento de la herencia, el polimorfismo y la reutilización. Sin embargo, en la práctica, el árbol binario puede definirse de una forma extremadamente simple, como es el caso del tipo `SimpleBinaryTree<T>` del Listado 14-8.

```
public class SimpleBinaryTree<T>
{
    public SimpleBinaryTree<T> Left { get; set; }
    public SimpleBinaryTree<T> Right { get; set; }
    public bool IsLeaf { get { return Left == null &&
                                      Right == null; } }
    public T Value { get; set; }
    public SimpleBinaryTree(T value)
    {
        Value = value;
    }
}
```

Listado 14-8 Implementación simple de árbol binario

El tipo `SimpleBinaryTree<T>` solamente cuenta con las propiedades `Left` y `Right`, para definir al hijo izquierdo y al hijo derecho respectivamente, una propiedad `IsLeaf` para saber si el nodo es una hoja y una propiedad `Value` para almacenar el valor del nodo. Desde el punto de vista práctico, esta es una definición minimalista y eficiente de árbol binario. La Figura 14-8 nos ilustra la forma interna de representación en memoria del árbol de la Figura 14-6 utilizando el tipo `SimpleBinaryTree<T>`

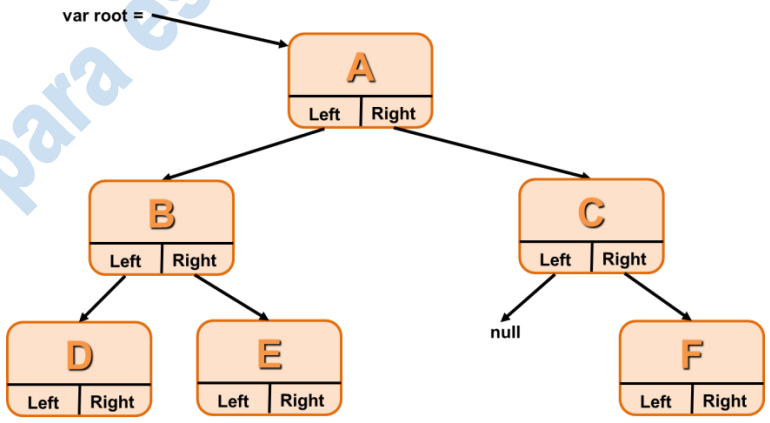

Figura 14-8 Representación de un árbol binario (Las flechas indican referencias y los cuadrados objetos)

La representación de un árbol k-ario utilizando clases, es una vía mucho más elegante y flexible que la representación utilizando un array, pero gasta al menos 8 bytes (tamaño en memoria de una referencia en una arquitectura de 64 bits) más por cada nodo del árbol (referencia del objeto), tamaño que puede parecer una nimiedad, pero cuando no disponemos de mucha memoria (tal es el caso de las aplicaciones que ejecutan en dispositivos embebidos, cajas registradoras, equipos médicos, routers, etc) y el árbol es grande, pues sí que puede acabar con la memoria RAM. En el caso de una PC de las actuales, donde se cuenta con abundante RAM, entonces esto no representa un problema.

En el caso de la representación utilizando un array, la memoria es contigua y se reserva una vez al crear el objeto árbol. Pero esto impide hacer crecer el árbol o hace costoso el hacerlo, siendo $O(n)$. Por otro lado en el caso de la representación por clase, cada vez que se crea un objeto, se reserva memoria y se asocia una referencia. De modo que a diferencia de la implementación vía array, los objetos del árbol podrán estar dispersos en la memoria de la computadora. La memoria ocupada por aquellos que queden "desconectados" (producto de alguna eliminación o poda que se le haya hecho al árbol) será recuperada automáticamente por el recolector de basura de la plataforma .Net cuando sea necesario.

14.3 ÁRBOL BINARIO DE BÚSQUEDA

Un **árbol binario de búsqueda** (ABB), es un árbol binario que cuenta en cada nodo con valor que se denomina **llave**, que tiene que ser de un tipo `IComparable` (basado en el cual se realizarán las búsquedas) también puede tener un valor opcional que puede ser de cualquier tipo `T` como en el caso de la definición genérica `BinaryTree` (claro en ejemplos sencillos el propio valor informacional del tipo `T` del nodo pudiera servir como llave).

Sobre la llave del ABB, se define una relación de orden total, de modo que, dado un nodo cualquiera, sin perder generalidad, todas las llaves de los nodos del sub-árbol izquierdo de un nodo, serán menores o iguales que la llave del nodo, y todas las llaves de los nodos del sub-árbol derecho serán mayores o iguales.

Los ABB sirven para representar conjuntos y diccionarios, y sobre ellos se pueden definir operaciones de Búsqueda, Mínimo, Máximo, Predecesor, Sucesor, Inserción y Eliminación.

14.3.1 DEFINICIÓN 1 ÁRBOL BINARIO DE BÚSQUEDA (ABB)

Un **árbol binario de búsqueda** es un **árbol binario** donde todo nodo cuenta con una **llave** y se cumple que: Para todo nodo x, si y es un nodo del sub-árbol izquierdo de x, entonces $y.key \leq x.key$ y si z es un nodo del sub-árbol derecho de x, entonces $x.key \leq z.key$.

Usualmente, en un ABB, los valores de las llaves son únicos, ya que estos árboles se suelen usar para representar conjuntos (o diccionarios), pero es posible tener un ABB con llaves repetidas. No obstante, para simplificar la implementación de los algoritmos que se verán en lo adelante para ABB, se considerará que las llaves no se repiten.

Computacionalmente, un nodo de un ABB, puede representarse como se muestra en el Listado 14-9.

Listado 14-9 Implementación de un nodo de un árbol binario de búsqueda (ABB)

```
public class BinarySearchTree<TKey, TValue> where TKey : IComparable
{
    public BinarySearchTree<TKey, TValue> Left { get; protected set; }
    public BinarySearchTree<TKey, TValue> Right { get; protected set; }
    public bool IsLeaf { get { return Left == null && Right == null; } }

    public TKey Key { get; protected set; }
    public TValue Value { get; set; }

    public BinarySearchTree(TKey key, TValue value = default(TValue))
    {
        Key = key;
        Value = value;
    }
}
```

Como se puede observar, `BinarySearchTree<TKey,TValue>` ahora contiene dos parámetros genéricos, para poder almacenar un valor adicional asociado a la llave de modo que pueda servir como una implementación para la interface `IDictionary<TKey, TValue>` que se ve en el Capítulo 13.

Note que el parámetro `TKey` tiene que ser `IComparable`, para entonces poder satisfacer la Definición 1 de ABB. Esto se expresa con la notación `where TKey:IComparable` (**genericidad restringida**).

Las propiedades `Key`, `Left` y `Right` tienen la forma de acceso `protected set` con toda intención, ya que se quiere que una vez que se asigne un valor a la llave en el código del constructor, el consumidor no pueda cambiar directamente este valor, o sea que un valor de llave solo se pueda especificar mediante los métodos `Insert` y `Delete` durante el proceso de construcción del árbol.

14.3.2 VALORES POR DEFECTO DE UN TIPO GENÉRICO

En el constructor, el valor del parámetro `value` puede ser opcional (para el caso en que solo se usen las llaves en el ABB) lo que se ha indicado con `TValue value = default(TValue)` dentro del código del constructor.

El compilador sustituirá este **valor predeterminado** según sea el tipo concreto que se use para `TValue` que es `null` si es un tipo por referencia, 0 para los numéricos y `false` para `bool`.

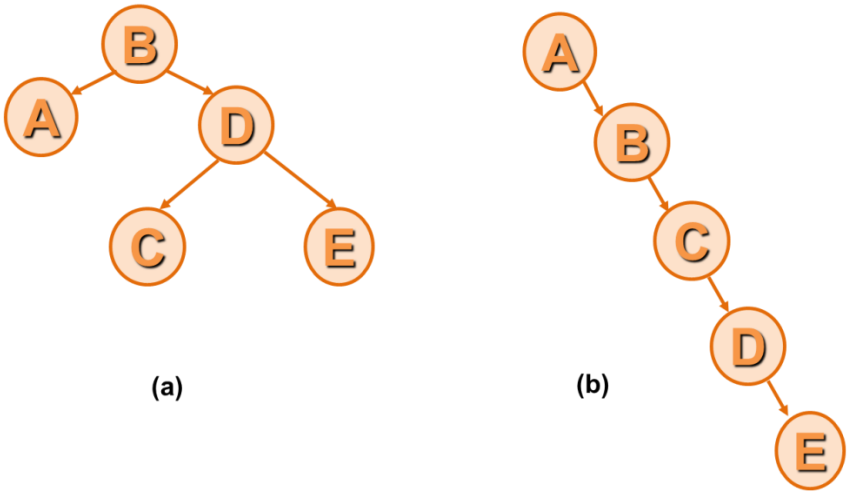

Figura 14-9 Ejemplos de árboles binarios de búsqueda (solo se han mostrado los valores de las llaves). El árbol (b) es un ABB degradado

En la Figura 14-9 se muestran dos ejemplos de ABB, en el que las llaves son de tipo `string`. Observe que a pesar de tener estructuras diferentes, mantienen el mismo orden de las llaves. En esta figura solo se ha mostrado el valor de las llaves, que es lo que se tiene en cuenta a la hora de ordenar el árbol.

Note que según la Definición 1, el árbol de la Figura 14-9 (b) es ordenado pero está completamente degradado, lo que no es eficiente para el algoritmo de búsqueda que se verá más abajo.

La definición de `BinarySearchTree` ha mantenido la diferenciación entre la llave de un nodo y el valor que puede estar asociado al nodo. En situaciones en que el ABB se usa simplemente paras representar conjuntos de valores de un tipo que sea a su vez `IComparable`, se pudiera prescindir entonces de la propiedad `Value` y basarnos solo en una definición de la forma `class BinarySearchTree<TKey> where TKey : IComparable.`

14.3.3 BÚSQUEDA Y CONSULTAS

La operación más básica que se puede realizar sobre un árbol binario de búsqueda es determinar si una llave se encuentra en el ABB y devolver el valor asociado a dicha llave (se está usando el árbol como una suerte de diccionario). O simplemente devolver un valor `bool` cuando el ABB se está usando para implementar un conjunto en el que el tipo de la llave es el propio tipo de los elementos del conjunto.

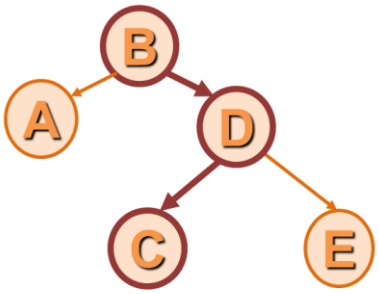

Figura 14-10 Ejemplo de búsqueda del elemento C en el ABB. Se parte de la raíz. Como B<C entonces se analiza la rama derecha. Luego se compara C<D y se selecciona la rama izquierda. Al llegar al nodo C, se devuelve el valor asociado en caso de que exista. Si se llega a una hoja y no se contiene la llave esperada o a una rama null, entonces el elemento no se encuentra en el árbol.

El algoritmo de búsqueda (Listado 14-10) es parecido al algoritmo de búsqueda binaria en un array ordenado. Partiendo del nodo raíz se compara el elemento que se está buscando con la llave del nodo en curso. Si la llave es mayor que la del nodo en curso entonces se continúa buscando en el sub-árbol derecho, si es menor se continúa la búsqueda en el árbol izquierdo y si es igual entonces es que ya la hemos encontrado. De manera recursiva, se sigue el proceso hasta que se llegue a un caso base: un nodo hoja, o un nodo `null`, en donde se tiene que notificar que no se encontró el valor.

Listado 14-10 Implementación de la búsqueda en el árbol binario de búsqueda (ABB)

```
public bool Search (TKey key, out TValue value)
{
  value = default(TValue);
  if (key.Equals(this.Key))
  {
    value = this.Value;
    return true;
  }
  if (key.CompareTo(this.Key) < 0)
    return Left != null && Left.Search(key, out value);
  return Right != null && Right.Search(key, out value);
}
public bool Search(TKey key)
{
  TValue dummyValue;
  return Search(key, out dummyValue);
}
```

El método `Search` tiene dos sobrecargas, una que permite obtener el valor asociado al nodo como un parámetro de salida (`out`) en el método, y otra sobrecarga que solamente devuelve si se encontró o no la llave en el árbol. Como los parámetros de salida tienen que inicializarse antes de que el método pueda retornar, pues éste se inicializa con `default(TValue)` que es el valor predeterminado que tendrá el valor asociado al nodo. Lo primero que se realiza es comparar que la llave que se pasa por parámetro sea igual que la del nodo, y en tal caso, se para el algoritmo, se retorna `true` y se asigna el valor almacenado en el nodo. En caso de que sea diferente, entonces se determina qué rama debe analizarse, siguiendo el criterio de ordenación establecido (los menores por la izquierda y los mayores por la derecha). Note como antes de bajar por alguna rama,

realizando una llamada recursiva al método `Search` del nodo derecho o izquierdo, se analiza si ésta es `null`, y en caso de serlo, entonces se retorna `false`. Aquí el operador `&&` funciona como corto circuito en donde, si el primer operando es `false`, entonces nunca se evalúa el segundo, por lo que nunca se realiza la llamada recursiva correspondiente. El algoritmo `Search` funciona porque, primeramente respeta la Definición 1 y además siempre para, ya que se llega a la llave que se está buscando en el árbol, retornando `true`, o simplemente se llega a un hijo `null` y se retorna `false`.

Como el algoritmo empieza sobre la raíz del árbol de **n** nodos y siempre baja un nivel, entonces el costo del mismo es **O(h)** donde **h** será la altura del árbol. Como el árbol puede estar degradado (Figura 14-9 b), entonces, para el caso peor, se tendrá un costo $O(n)$ pero si el árbol está adecuadamente balanceado el costo será $log_2 n$

14.3.3.1 MÍNIMO Y MÁXIMO

Como se tiene definida una relación de orden total sobre las llaves del árbol binario de búsqueda y la cantidad de nodos es finita entonces hay un valor mínimo y un valor máximo.

Utilizando la Definición 1, se puede deducir que le menor elemento estará siempre en la parte izquierda, y el máximo estará en la parte derecha. Por tanto, podemos desarrollar un algoritmo, esta vez iterativo, que navegue por el nodo más a la izquierda (derecha) mientras que éste no sea `null` para encontrar el valor mínimo y máximo que almacena el árbol binario de búsqueda. En este caso, en vez de devolver la llave o el valor, se devolverá el nodo completo, ya que estos métodos servirán de base para otros métodos que se verán más adelante.

```
public BinarySearchTree<TKey,TValue>
Min()
{
    var current = this;
    while (current.Left != null)
        current = current.Left;
    return current;
}
```

```
public BinarySearchTree<TKey,
TValue> Max()
{
    var current = this;
    while (current.Right != null)
        current = current.Right;
    return current;
}
```

Listado 14-11 Implementación de los métodos Min y Max de un árbol binario de búsqueda (ABB)

Para analizar el funcionamiento de `Min` y `Max` (Listado 14-11), se puede seguir un procedimiento similar al del algoritmo `Search`. Primeramente, se sabe que el mínimo (máximo) estará siempre en el extremo izquierdo (derecho). Demostrar esto es muy sencillo realizando una reducción al absurdo, suponga que el mínimo no se encuentra en el nodo más a la izquierda, entonces pueden suceder dos cosas: el nodo tiene un hijo izquierdo o es hijo derecho de su padre. En cualquiera de los casos, se muestra que la suposición es contradictoria, porque existe otro menor, por tanto, se cumple la hipótesis. El algoritmo finaliza, porque en cada iteración se baja un nivel y se llega a un `null`, terminando el ciclo devolviendo el nodo almacenado en la variable `current`. Como en el caso peor hay que recorrer la altura del árbol y un ABB puede quedar degradado, entonces estos algoritmos son de costo $O(n)$.

14.3.3.2 PREDECESOR Y SUCESOR

Al estar ordenadas las llaves del ABB, se puede, definir para cualquiera de ellas, su sucesor y su predecesor.

Dado un nodo con llave x cualquiera en un ABB, se define el **predecesor de x** como el nodo con la mayor llave y, posible tal que $y < x$. De forma análoga se define el **sucesor de x**, como el nodo con menor **llave** z tal que $x < z$. De manera trivial, el predecesor del mínimo y el sucesor del máximo será vacío (`null`).

Como en el ABB, los nodos se organizan estructuralmente según su llave, es decir los menores por la izquierda y los mayores por la derecha, pues dado un nodo x cualquiera, se puede saber, siguiendo la estructura del árbol y sin realizar comparaciones con la llave, cuál es el predecesor y el sucesor de dicho nodo.

En la Figura 14-11 se muestra la relación entre las llaves del árbol y un nodo x cualquiera, en dependencia de la posición estructural del mismo. Si x es un hijo izquierdo de algún nodo, entonces todos los nodos alcanzables a través del padre son mayores que x. Las llaves del sub-árbol izquierdo son menores y las del derecho son mayores. En caso de que el nodo x sea un hijo derecho, entonces las llaves de los nodos alcanzables desde x a través del padre, serán menores que x.

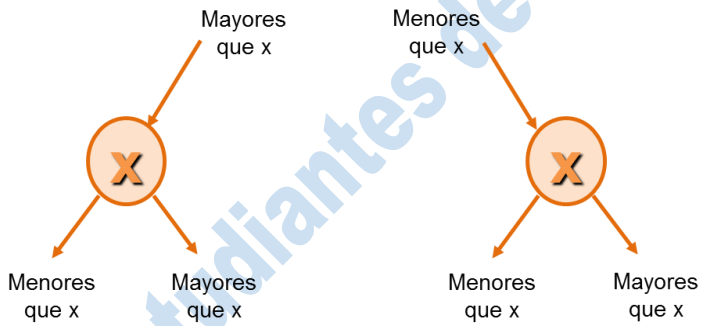

Figura 14-11 Relación de las llaves con respecto a un nodo x en dependencia de la posición estructural en el ABB

Teniendo esto en mente, el algoritmo para hallar el sucesor de un nodo con llave x hay que dividirlo en dos casos separados para analizarlo adecuadamente: cuando el nodo tiene un hijo derecho y cuando no lo tiene. Si el nodo tiene un hijo derecho, entonces se halla el mínimo del hijo derecho (algoritmo `Min` visto anteriormente) y éste será el sucesor. Note que en este caso, se asegura que el mínimo del hijo derecho sea el sucesor, porque la otra posible llave mayor que x, se encuentra en el padre, pero por la estructura del árbol, este valor será mayor que el encontrado ya que está en la rama izquierda del padre que por definición son valores menores. En caso de que el nodo con llave x, no tenga hijo derecho, el sucesor será el menor ancestro tal que su hijo izquierdo también sea ancestro del nodo con llave x (o el propio nodo). En la Figura 14-12 se muestra un ejemplo no trivial de sucesor.

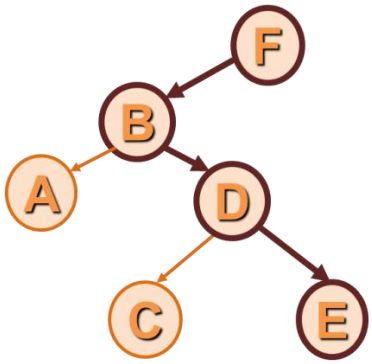

Figura 14-12 El sucesor de E es el nodo raíz F, que es el menor ancestro tal que su hijo izquierdo también es ancestro del nodo E

La complejidad de este algoritmo radica en que, cuando el nodo que se está analizando no cuenta con hijo derecho, es necesario recorrer el árbol hacia arriba a partir de un nodo y analizar la posición estructural de los ancestros. Pero el tipo `BinarySearchTree<TKey, TValue>` previamente diseñado, solo cuenta con propiedades (`Left` y `Right`) para referenciar a los hijos, por lo que si se quiere que un ABB tenga esta funcionalidad será necesario, añadir una propiedad adicional `Parent` en la implementación del árbol:

```
public BinarySearchTree<TKey, TValue> Parent { get; protected set; }
```

Con la propiedad `Parent`, se puede entonces navegar por el árbol en cualquier sentido; a partir de un nodo, ir hacia los hijos o ir hacia el padre.

Claro esta propiedad `Parent` tiene un costo en memoria debido a que se tiene esta referencia por cada nodo. Gráficamente entonces, las conexiones del árbol se representarían, en vez de con una flecha hacia abajo, pues con dos flechas o con una en dos sentidos. Si suponemos que en una arquitectura de 64 bits, una referencia son 8 Bytes, entonces tendremos para cada nodo un gasto de 24 Bytes solo en referencias (una al padre y una a cada uno de sus dos hijos).

El algoritmo `Sucesor` quedaría muy simple ahora (Listado 14-12).

```
public BinarySearchTree<TKey, TValue> Successor()
{
  if (Right != null)
    return Right.Min();
  var parent = Parent;
  var current = this;
  while (current != null && current == parent.Right)
  {
    current = parent;
    parent = current.Parent;
  }
  return parent;
}
```

Listado 14-12 Implementación del sucesor para un nodo de un árbol binario de búsqueda (ABB)

En el ciclo se recorren los ancestros para encontrar el menor tal que su hijo derecho también sea ancestro del nodo de partida.

Este algoritmo siempre finaliza, ya que recorre el árbol en un sentido (hacia arriba hasta que llega a la raíz o hacia abajo hasta que llega a una hoja utilizando el método `Min` que ya se sabe que es correcto). Como navega siempre por una rama, aumentando o disminuyendo el nivel en 1, entonces el costo para el caso peor será $O(h)$ y como el ABB puede estar degradado el costo podría ser $O(n)$. Proponga una implementación para el método `Predecesor` y analice su funcionamiento.

14.3.4 INSERCIÓN

La inserción en el árbol binario es la operación que permite ir construyendo el árbol no de manera explícita directamente en el momento de la construcción sino haciendo inserciones sucesivas de llaves (y opcionalmente sus valores asociados).

El algoritmo de inserción garantizará que una vez finalizado, el ABB siga cumpliendo la Definición 1, es decir, que las llaves queden organizadas ordenadamente según la estructura del árbol.

Se dice que un algoritmo modifica una estructura de datos dinámicamente, cuando luego de creada dicha estructura es mutable. El algoritmo puede cambiar el estado de la estructura de datos, pero una vez finalizado, debe mantener las invariantes que define dicha estructura. Es decir, llamadas independientes y progresivas al algoritmo, van modificando la estructura de datos manteniendo sus invariantes.

El algoritmo consiste en, dado una llave a insertar, localizar, respetando el orden del árbol, un nodo nulo en donde guardar la llave a insertar. El árbol resultante sería de tamaño $n + 1$, es decir, crecería en 1.

Para localizar el nodo adecuado, primero se selecciona la raíz y se compara la llave a insertar con la llave de la raíz y se decide, en tal caso, si moverse al hijo izquierdo o derecho en dependencia de si el valor a insertar es menor o mayor que el de la raíz. Y así sucesivamente se analizan todos los nodos hasta que se encuentre un hijo nulo (que es donde se colocará el nuevo nodo). Como se está considerando que el árbol no tiene llaves repetidas, si se detecta que la llave existe se lanza la excepción `"La llave existe"`. Note que cuando se inserte el nuevo nodo hay entonces que actualizar la propiedad `Parent` como se muestra en Listado 14-13.

```
public void Insert(TKey key, TValue value = default (TValue))
{
  if (key.Equals(this.Key)
    throw new ArgumentException("La llave existe");
  if (key.CompareTo(this.Key) < 0)
  {
    if (Left == null)
       Left = new BinarySearchTree<TKey, TValue>(key, value)
                 { Parent = this };
    else
      Left.Insert(key, value);
  }
  else
  {
    if (Right == null)
      Right = new BinarySearchTree<TKey, TValue>(key, value)
```

```
                    {Parent = this};
            else
              Right.Insert(key, value);
          }
        }
```

Listado 14-13 Método de inserción en un árbol binario de búsqueda (ABB)

El comportamiento del método `Insert` es muy similar al método `Search`, con la salvedad de que cuando se encuentra un hijo en `null`, ahí mismo se crea un nuevo nodo, pasando como parámetros del constructor, la llave y el valor, y se engancha al árbol, actualizando la propiedad `Parent`.

Es importante el orden en que se inserten las llaves en un ABB porque se puede crear un ABB degradado. Tal es el caso de insertar las llaves en orden ascendente o descendente. Hay algoritmos para "balancear" un árbol degradado o para insertar tratando de mantener el "balanceo" pero esto se sale del alcance de este libro.

El algoritmo de inserción finaliza, ya que en el caso peor se llega a una hoja, y ahí es donde se inserta el nuevo nodo como hijo derecho o izquierdo. Esto sería $O(h)$ y como puede llegar a ser degradado el árbol, pues se obtiene $O(n)$

14.3.5 ELIMINACIÓN

Eliminar (quitar) una llave de un ABB (lo que significa quitar también el valor asociado), en dependencia de cómo se implemente, puede alterar drásticamente la estructura del árbol. Por ejemplo, si se quiere eliminar exactamente la llave que se encuentra en la raíz de un árbol, una variante burda sería eliminar el nodo raíz, y luego, al quedar dos árboles diferentes desconectados, habría que mezclarlos en uno solo. Este proceso de mezcla no es trivial y puede ser costoso.

Sin embargo existe una variante que, haciendo un “intercambio” de llaves y valores en los nodos, posibilita que las eliminaciones de nodos sólo ocurran en las hojas. Esto resultaría en un árbol de n-1 nodos en donde el resto de la estructura del árbol original se mantiene íntegra.

Éste algoritmo de eliminación se divide en tres casos:

I. Si la llave que se quiere eliminar, se encuentra en un nodo hoja, entonces se elimina el nodo.
II. Si la llave que se quiere eliminar, se encuentra en un nodo con un solo hijo, entonces se reconecta el padre con el hijo del nodo que corresponde a dicha llave, quedando éste eliminado.
III. Si la llave que se quiere eliminar, se encuentra en un nodo con dos hijos, entonces se localiza el Máximo del hijo izquierdo (el mayor de los menores) o de forma análoga, el Mínimo del hijo derecho (el menor de los mayores) y se intercambian los valores con el nodo actual, entonces se elimina este nodo para caer recursivamente en uno de estos tres casos hasta terminar en un caso I o II.

En la Figura 14-13 se muestra un ejemplo de eliminación en un ABB, donde se presenta el tercer caso, y luego de realizar el intercambio con el menor de los mayores (`Right.Min()`) se vuelve a caer en un caso II que se resuelve directamente. Es importante notar, que cuando se intercambia con el menor de los mayores, el valor intercambiado sigue manteniendo la invariante del ABB (el valor C sigue siendo mayor que A y menor que E y D). Y es por eso que se escoge el menor de los mayores (`Right.Min()`) o el mayor de los menores (`Left.Max()`) para realizar el intercambio, porque esta llave no afecta la invariante del ABB (los menores para la izquierda y los mayores para la derecha).

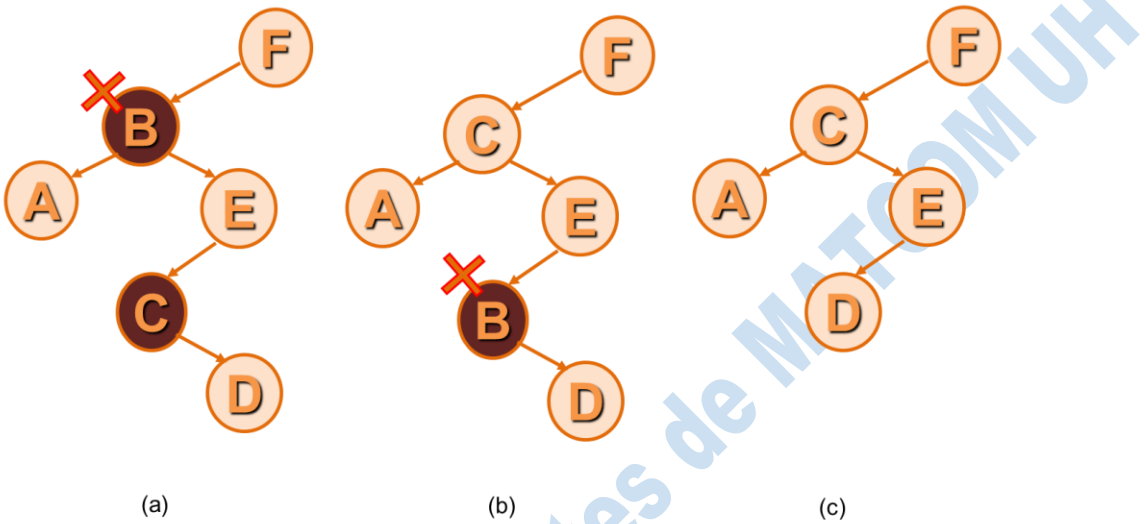

Figura 14-13 Eliminación de la llave B, del ABB con raíz F: (a) por III se llega a B y se intercambia con el menor de los mayores y se obtiene el árbol (b) Por II se elimina el nodo B pasando D a ser el hijo de E obteniéndose el árbol (c)

El algoritmo de eliminación quedaría de la siguiente forma:

```
public void Delete(TKey key)
{
  var node = InternalSearch(key);
  if (node == null)
    throw new Exception("La llave no existe");

  // CASO I
  if (node.IsLeaf)
  {
    if (node.Parent.Left==node)
       node.Parent.Left = null;
    else
      node.Parent.Right = null;
    return;
  }

  // CASO II-a
  if (node.Left==null)
  {
    if (node.Parent.Left == node)
       node.Parent.Left = node.Right;
     else
```

```
            node.Parent.Right = node.Right;
          return;
        }

        // CASO II-b
        if (node.Right == null)
        {
          if (node.Parent.Left == node)
            node.Parent.Left = node.Left;
          else
            node.Parent.Right = node.Left;
          return;
        }

        // CASO III
        var minOfMax = node.Right.Min();
        Key = minOfMax.Key;
        Value = minOfMax.Value;
        minOfMax.Delete(minOfMax.Key);
    }
```

Para analizar el código del algoritmo, éste se ha separado en los tres casos posibles. Inicialmente se utiliza una variación del método `Search`, llamado `InternalSearch` que devuelve, en vez del valor, una referencia al nodo completo (se deja al lector), de forma tal que posteriormente se puedan realizar operaciones sobre dicho nodo. En caso de que este método devuelva `null`, entonces se lanza excepción porque la llave que se desea eliminar no existe en el árbol. Para verificar el caso uno se utiliza la propiedad `IsLeaf` que devuelve `true` si el nodo es hoja. En caso verdadero, se "elimina" el nodo, pero para ello, hay que desconectarlo del padre, por lo que hay que verificar si el nodo a eliminar es hijo derecho o izquierdo de su padre. Cuando se determina, entonces se asigna a `null` para desconectar el nodo del árbol. El caso II es muy similar, pero se divide en IIa y IIb que son casos simétricos en dependencia de si el hijo `null` se encuentra a la derecha o a la izquierda. Sin perder generalidad, se reconecta el hijo con su abuelo, eliminando así al padre intermediario. El caso III resulta el más simple: se busca el menor de los mayores (que siempre existe porque en caso peor es el mismo nodo derecho), se reemplazan los valores del nodo actual y luego se elimina recursivamente el menor de los mayores. En la práctica, no es necesario intercambiar los valores como se muestra la Figura 14-13, ya que la llamada recursiva a `Delete` solo analiza el sub-árbol que tiene como raíz el menor de los mayores.

En el caso I y II se puede apreciar que el algoritmo es correcto (ya que por construcción, elimina un nodo manteniendo las invariantes del ABB) y finaliza (ya que se elimina una hoja o se reconecta un abuelo con un nieto). Estos serían los casos bases que tienen un costo constante $O(1)$. El caso general, consiste en obtener el menor de los mayores (que ya se vio anteriormente que era $O(h)$), reemplazar los valores del nodo actual (también $O(1)$) y eliminar recursivamente dicho nodo. Hay que notar que el sub-árbol en que se realiza la eliminación recursiva va a tener una altura estrictamente menor que el árbol del nodo actual, donde en su caso peor será h-1 si la altura del árbol actual es h. Por tanto, el caso peor será cuando el árbol esté degradado hacia la derecha, y se quiera eliminar la raíz. En este caso, se realizarán exactamente h-1 operaciones recursivas de reemplazo

más una de eliminación, por lo que en total se realizarán $O(h)$ operaciones, por lo que el costo para el caso peor será de $O(n)$

14.3.6 RECORRIDOS

Un árbol binario se puede recorrer de diferentes formas. Un recorrido significa visitar todos los nodos del árbol al menos una vez. Existen tres recorridos clásicos: **Preorden**, **Entreorden** (también conocido como recorrido en simétrico) y **Postorden**.

Estos recorridos clásicos tienen diversas utilidades y aplicaciones en dependencia del problema computacional que se quiera resolver. Por ejemplo, un árbol binario se puede utilizar para representar y facilitar la evaluación de una expresión aritmética como la que se muestra en la Figura 14-14.

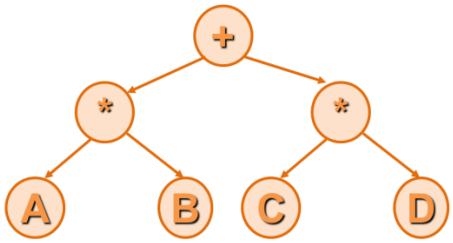

Figura 14-14 Árbol binario que representa una expresión matemática: a*b+c*d

Cada recorrido empieza en la raíz y sigue cierto patrón:

- **Preorden**: se visita el nodo actual y luego se visitan los sub-árboles.
- **Entreorden**: se visita el sub-árbol izquierdo, se visita el nodo actual y luego el sub-árbol derecho.
- **Postorden**: se visitan primeramente los sub-árboles y luego se visita el nodo actual.

Por ejemplo, si se visita el árbol de la Figura 14-14 utilizando el recorrido **entreorden**, se obtendría la expresión matemática $a * b + c * d$

Pero si se quisiera "evaluar" la expresión representada en el árbol de la Figura 14-14, como la multiplicación tiene precedencia sobre la suma, para calcular la suma (nodo raíz), tienen que haberse calculado previamente las expresiones de cada sub-árbol, que a su vez son multiplicaciones que dependen de dos operandos. Por tanto, habrá que garantizar tener calculado el valor de los operandos (hijo) antes de aplicar una operación (nodo), por lo que el recorrido conveniente es el de **postorden**.

En el caso de un ABB, al realizar un recorrido **entreorden** se garantiza que los nodos se visiten en orden según la relación de orden total definida sobre las llaves del ABB. Por tanto, se puede obtener una lista de valores ordenados. Esto es muy fácil de demostrar, ya que, por la definición 14.3.1, para un nodo cualquiera, las llaves del sub-árbol izquierdo son menores que la llave del nodo actual, y el recorrido entreorden, entonces visitará esas llaves que son menores, luego la llave del nodo, y posteriormente las llaves del sub-árbol derecho que son mayores por definición, obteniendo una lista ordenada.

Estos recorridos (preorden, entreorden y postorden) son $O(n)$ siempre y cuando recorran cada nodo del árbol a lo sumo una vez.

Si se quisieran imprimir las llaves de un ABB en los diferentes recorridos, una forma de hacerlo sería la que se muestra en el Listado 14-15.

```
public void PreOrder()
{
  Console.WriteLine(Key);
  if (Left != null)
    Left.PreOrder();
  if (Left != null)
    Left.PreOrder();
}
```

```
public void InOrder()
{
 if (Left!=null)
   Left.InOrder();
 Console.WriteLine(Key);
 if (Left != null)
   Left.InOrder();
}
```

```
public void PostOrder()
{
  if (Left != null)
    Left.PreOrder();
  if (Left != null)
    Left.PreOrder();
  Console.WriteLine(Key);
}
```

Listado 14-14 Diferentes recorridos sobre un árbol binario de búsqueda (ABB)

Note que en este ejemplo lo que se está haciendo en el recorrido es ir imprimiendo el valor de la llave, pero la idea es que este patrón de recorrido puede aplicarse a otras tareas como la de sumar un árbol de expresión. De acuerdo con lo estudiado en el capítulo 12 de Programación Funcional los métodos de recorrido podrían tener asociados un funcional a aplicar a cada elemento (valor de la llave o valor del nodo) del recorrido.

14.3.6.1 ITERADORES PARA HACER LOS RECORRIDOS

Si quisiéramos iterar por un árbol binario de búsqueda, siguiendo un recorrido en simétrico (entreorden) utilizando la interface `IEnumerable<TKey>`, se podría implementar dicha interface, y codificar el método `GetEnumerator()` como se muestra en Listado 14-15.

```
public IEnumerator<TKey> InOrder()
{
  foreach (var key in Left)
  {
    yield return key;
  }
  yield return Key;
  foreach (var key in Right)
  {
    yield return key;
  }
}
```

Listado 14-15 Recorrido entreorden en un árbol binario de búsqueda (ABB)

Note que el `foreach` implícitamente aplica a su vez el método `InOrder()` del nodo `Left` (o `Right`).

Aunque sencillo, debe observarse que este código no es lo más eficiente ya que es $O(n^2)$, sin contar que se gasta memoria adicional para crear un iterador por cada nodo del árbol. Imagine un árbol binario de altura h, en donde se itera a partir de la raíz. Devolver el primer elemento, implica crear h iteradores (desde la raíz hasta la hoja más a la izquierda) y devolver el primer elemento pasando por los h iteradores. Al pedir el segundo elemento, se destruye el iterador de la hoja más a la izquierda y se vuelve a bajar desde el padre hasta el nivel h - 1, y luego para el tercer elemento, se vuelve a bajar al nivel h, ahora en la hoja derecha, y así sucesivamente, para todos los valores del nodo. Esto significa, que para cada nodo se hacen $O(d)$ operaciones que corresponde al

recorrido desde la raíz hasta el nodo a profundidad d (donde d puede ser h en el caso de las hojas más alejadas). Por tanto, el costo general sería $O(nh)$. En el caso de un árbol de búsqueda degradado, se realizarían $\frac{n\cdot(n+1)}{2}$ operaciones siendo $O(n^2)$. En caso de que el ABB sea un árbol binario completo, entonces se obtendría un $n * log_2 n$ ya que la altura del árbol binario completo, es $log_2 n$.

Esto demuestra que un código bonito y simple, no necesariamente es el más eficiente. Es responsabilidad del programador saber reconocer cuán eficiente es el código que escribe y tratar de optimizarlo cuando sea necesario.

Para realizar un iterador entreorden de un ABB con una eficiencia de O(n) existe una técnica que consiste en ensartar las referencias nulas con el sucesor o el predecesor en orden.

Para ensartar un árbol binario de búsqueda, una vez construido, se tienen que cumplir las siguientes reglas:

Si la propiedad `Left` de un nodo es `null`, se ensarta con el sucesor del nodo:

```
this.Left = this.Predecessor()
```

Si la propiedad `Right` de un nodo es `null`, se ensarta con el sucesor del nodo:

```
this.Right = this.Successor()
```

Al terminar, solamente quedarán dos referencias nulas, el `Left` del mínimo, y el `Right` del máximo. De esta forma, se puede realizar el recorrido entreorden de forma lineal. Le queda propuesto al lector implementar esta idea.

14.4 APOSTILLA

Si consideramos que las pilas son inherentes al procesamiento recursivo, y que su utilización está subsumida en los lenguajes de programación que permiten la recursividad, puede decirse entonces que los árboles son las estructuras más utilizadas en la Ciencia de la Computación. Como ya se ha dicho la gran diversidad de interpretaciones y de formas de usarlos y aplicarlos hace impracticable intentar resumir toda su variedad en una biblioteca de tipos. Para el caso de .NET invitamos al lector a inspeccionar las bibliotecas existentes.

Una exposición más profunda y detallada del tema de árboles se sale del alcance de este libro, el lector puede buscarla en la abundante literatura existente sobre estructuras de datos. En este capítulo se han presentado los conceptos y operaciones más básicas que le pueden servir de punto de partida a un estudio más amplio y para que el programador pueda hacer las adaptaciones y adecuaciones necesarias según necesite en las aplicaciones que desarrolle.