Los últimos serán los primeros, y los primeros, los últimos

Libro de Mateo 19:30, 20:16

Los arrays (Capítulo 7) son la forma más primaria de estructurar datos. Siendo una abstracción del funcionamiento de la memoria de las computadoras, los arrays permiten acceder y modificar sus datos a través de índices numéricos que indican la posición secuencial del dato dentro de su estructura .

Los objetos (Capítulo 6), que son una evolución del concepto de record o artículo presente desde los primeros lenguajes de programación, permiten estructurar datos por composición. Los datos de la estructura se acceden y modifican a través de nombres, el concepto de posición del dato dentro de la estructura no tiene significación para su uso.

El desarrollo de la programación ha estado acompañado del desarrollo de distintas formas de estructurar los datos, tema que se conoce en la literatura como Estructuras de Datos. Una **estructura de datos** actúa como una suerte de "contenedor" de datos. Los datos del contenedor son de un (o más de uno) tipo de datos pero por lo general el funcionamiento del contenedor no depende del tipo de los datos que contiene. Las diferentes estructuras se caracterizan por la forma en que se pueden añadir, quitar, acceder, buscar y modificar los datos que contienen, ocultando al programador, usuario de los mismas, los detalles de su implementación.

Tal vez por la competencia y la vanidad de los propios desarrolladores, y por la dinámica misma de la evolución de la programación; los diferentes lenguajes y plataformas no han usado una terminología completamente similar y la funcionalidad de las estructuras pueden tener matices diferentes entre un lenguaje y otro. En este capítulo y el siguiente se estudian algunas de las estructuras de datos más conocidas y utilizadas, como colecciones, listas, pilas, colas, diccionarios, árboles. Los ejemplos se basan en la disponibilidad de éstas en el lenguaje C# y en las bibliotecas .NET; sin embargo, son estructuras de datos que están presentes en la mayoría de los lenguajes, aunque puedan diferir en la sintaxis de su uso y en detalles en cuanto a la semántica de su funcionalidad.

13.1 Colecciones

El tipo interface `ICollection<T>` que se vio en el Capítulo 11 es el concepto de colección de datos más cercano al concepto básico de conjunto en matemática. `ICollection` representa un tipo que permite almacenar, organizar y enumerar elementos de un mismo tipo (Listado 13-1).

```
public interface ICollection<T> : IEnumerable<T>, IEnumerable
{
  void Add(T item);
  bool Remove(T item);
  bool Contains(T item);
  int Count { get; }
  void Clear();
  void CopyTo(T[] array, int arrayIndex);
  bool IsReadOnly { get; }
}
```

Listado 13-1 Interface ICollection

Las operaciones fundamentales de una colección se pueden agrupar en dos categorías: aquellas que modifican la colección y aquellas que permiten consultar el estado de la colección o de un elemento particular.

En el primer grupo se encuentran los métodos `Add`, `Remove` y `Clear`, que como su nombre indica, sirven respectivamente para adicionar un elemento a la colección, eliminar un elemento particular, y eliminar todos los elementos de la colección. Es importante destacar que la colección no define cómo están almacenados los elementos, ni tampoco que estos tengan que cumplir con un orden particular. El método `Remove` devuelve `true` si existía tal elemento dentro de la colección y por tanto se quitó de la colección, y `false` en caso de que no existiera.

En el segundo grupo se encuentran el método `Contains` y la propiedad `Count`. El método `Contains` permite averiguar si un elemento está en la colección, mientras que la propiedad `Count` devuelve la cantidad de elementos almacenados. En este mismo grupo se encuentra el método `GetEnumerator`, heredado de la interface `IEnumerable<T>` (Capítulo 11) mediante la cual se pueden recorrer los elementos de una colección sin depender de que tengan un orden en la misma.

Las estructuras de datos que se verán en este capítulo utilizan parámetros genéricos para especificar el tipo de los elementos que se almacenarán en dicha estructura. Cuando se trabaja con cualquier estructura de datos, es necesario conocer cómo se comportan éstas cuando el tipo genérico `T` con el que se crea una instancia de la estructura es de tipo por valor o por referencia.

Cuando se utilizan tipos por valor hay que tener en cuenta que lo que se guarda en la estructura (por ejemplo con una operación `Add` o `Insert`) es una copia del dato a guardar. Por el contrario cuando se utilizan tipos por referencia lo que se guarda en la estructura es la propia referencia pasada como parámetro.

Debe tener en cuenta entonces que si en la estructura se guardan referencias, quitar un elemento de la estructura no significa eliminar éste, y de igual modo si se modifica el elemento dentro de la estructura se está modificando al objeto original que pudiera estar siendo a su vez compartido en otra estructura.

Las operaciones que necesiten comparar por igualdad (como puede suponer que hace cualquier implementación de `Contains`) debieran basarse en el método `Equals` para garantizar que pueden aplicarse a cualquiera que sea el tipo concreto de los elementos que se guarden en la estructura. Es más, la manera correcta es emplear el método

estático `object.Equals(x, y)`, que permite que ambos argumentos sean `null`, y no el método `Equals` de instancia (`x.Equals(y)`) que fallaría en el caso en que `x` sea `null`.

Por simplificación no siempre se ejemplificarán ni se implementarán los tipos homólogos de dichas estructuras usados en la biblioteca de .NET (ubicados en el espacio de nombres `System.Collections` y `System.Collections.Generic`).

13.2 LISTAS

El tipo de dato **lista**, como estructura de datos, es aquel en que sus elementos están dispuestos en un orden lineal asociado cada uno a una posición numérica en la lista. Esta estructura de datos fue presentada en el Capítulo 7, a partir de una implementación particular usando arrays. En dicho capítulo se presentó un esbozo de la implementación del tipo `List<T>` y a partir de éste una descripción funcional de sus operaciones. Para formalizar la semántica de las operaciones de este tipo, presentaremos a continuación una descripción basada en la interface `IList<T>`. Esta interface, presente en la biblioteca estándar de .NET, contiene las definiciones de métodos y propiedades que caracterizan a una lista (Listado 13-2).

```
public interface IList<T> : ICollection<T>
{
  void Insert(int index, T item);
  void RemoveAt(int index);
  int IndexOf(T item);
  T this[int index] { get; set; }
}
```

Listado 13-2 Definición de la interface **IList<T> en la biblioteca estándar de .NET.**

Nótese como `IList` deriva a su vez de `ICollection` que a su vez deriva de `IEnumerable`, de modo que esto permite dividir las operaciones que puede realizar una lista en tres grupos: las operaciones relacionadas con índices, las operaciones relacionadas con la colección no dependientes de índices, y las operaciones relacionadas con la lista vista en cuanto a poder recorrer sus elementos sin depender del índice. Esto cumple con un principio importante de programación conocido como **segregación de interfaces** (Capítulo 16).

13.2.1 OPERACIONES BASADAS EN ÍNDICES

La interface `IList<T>` define a una lista como una colección de elementos, donde existe además un orden definido para éstos. Para ello se asocia a cada elemento un índice, que depende del momento en que fue añadido a la colección, y puede variar durante el tiempo de vida de la lista a partir de la invocación de las operaciones que modifican su estado interno. En este sentido, una lista puede verse como un array con capacidad variable, y por este motivo fue presentada en el Capítulo 7 a partir de una implementación basada justamente en arrays. Por tanto, una manera conveniente de representar gráficamente una lista, consiste en una secuencia de elementos ordenados por el índice o posición en la lista (Figura 13-1).

Figura 13.1 Representación de una lista como una secuencia ordenada de elementos.

Esto **no significa** que los elementos se encuentran ordenados según el tipo de los elementos de la lista (como puede ser el orden numérico si la lista es de tipo `int`, o el orden lexicográfico, si la lista es de tipo `string`). Por lo tanto, **es incorrecto** pensar que el elemento al que se le asocia el índice i+1 sea "mayor" que el elemento al que se le asocia el índice i, bajo ninguna interpretación particular de "mayor". Simplemente debe entenderse como existe un "primer" elemento, un "segundo" elemento, y así sucesivamente, pero cualquier interpretación semántica asociada al índice dependerá de la funcionalidad de las operaciones de la lista. Esto quiere decir, entre otras cosas, que no es posible por defecto implementar una búsqueda binaria (ver Capítulo 7) sobre los elementos de una lista a menos que en un caso particular, según el tipo de los elementos, se pueda garantizar que la posición que ocupan en la lista coincide con alguna relación de orden definida por un criterio de comparación aplicado al tipo de los elementos.

El hecho de que en la definición de `IList<T>` se especifique un indizador que asocia un índice a los elementos no significa, como se verá más adelante con las listas enlazadas, que la implementación concreta del `IList` tenga que basarse en almacenar estos en un array (físicamente consecutivos en la memoria). La noción de orden establecida por la lista simplemente indica que existe un índice que define biunívocamente a un elemento de la lista. Es responsabilidad de la implementación concreta mantener esta relación entre los índices y los elementos.

13.2.1.1 INSERT

La operación `Insert` toma como argumentos el elemento a insertar `x` y una posición `i` en la lista e inserta el elemento `x` en la posición `i`, desplazando todos los elementos en las posiciones posteriores a `i` una posición "hacia la derecha" (de forma que los índices de los elementos posteriores a i aumentan en 1). El valor de `i` tiene que estar en los rangos de la lista, es decir ser mayor o igual que cero y menor que la cantidad de elementos que hay en la lista.

La **Error! Reference source not found.** ilustra cómo quedaría la lista del ejemplo después de la inserción del valor `10` en la posición `4` de la lista. Note que los elementos `67`, `8`, `31` y `4` fueron desplazados una posición hacia la derecha, y ahora el índice asociado a cada uno es respectivamente el sucesor del índice que tenían asociado anteriormente.

13.2.1.2 REMOVEAT

Figura 13.2 Inserción del elemento 10 en la posición 4

La operación `RemoveAt`, que es la operación recíproca a `Insert`, toma como argumento una posición `i` y quita de la estructura el elemento en dicha posición, desplazando una posición "hacia la izquierda" a todos los elementos que estaban en una posición posterior a `i` (todos los elementos con índice mayor que i disminuyen en 1). Al igual que en la operación `Insert`, el valor de `i` tiene que estar en los rangos de la lista, es decir ser mayor o igual que cero y menor que la cantidad de elementos que hay en la lista.

En la Figura 13.3 se muestra cómo quedaría la lista si se quita ahora el elemento que tiene asociado el índice 2.

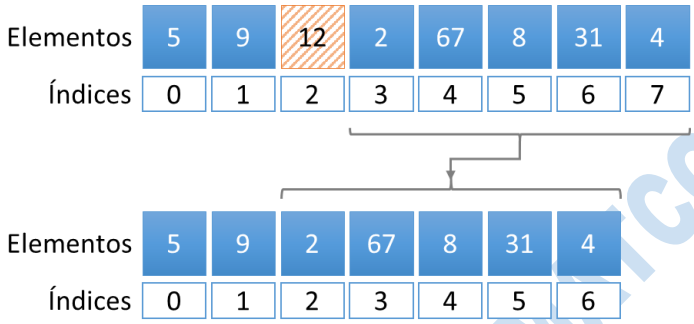


Figura 13.3 Resultado de eliminar el elemento asociado al índice 2.

Note cómo los elementos 2, 67, 8, 31 y 4 se corren hacia la izquierda, es decir, sus índices disminuyen en 1.

13.2.1.3 IndexOf

La operación `IndexOf` toma como argumento un elemento `x` y devuelve la posición asociada a ese elemento dentro de la lista. El convenio si el elemento no se encuentra en la lista es devolver `-1`. Si el elemento `x` se encuentra repetido, entonces se devuelve la posición de la primera ocurrencia de `x` en la lista, es decir la ocurrencia de menor índice. De esta forma, en la posición asociada al elemento `12` de la lista de la Figura 13.1 es la `2`, para el `4` es la `7` y para el 3 se devolvería `-1`.

13.2.1.4 Indizador

El indizador (operador `this`) permite interactuar con la lista con una sintaxis similar a la que se usa en los array (que es prácticamente la que utilizan la mayoría de los lenguajes de programación). Como se explica en el Capítulo 7, esta operación tiene doble funcionalidad: consultar el elemento asociado a un índice o cambiar su valor. De esta manera, incluso aunque la implementación concreta no almacena los elementos de la lista consecutivos en memoria, es posible ejecutar un código como el mostrado en el

```
IList<int> lista = new List<int>();
// Llenar la lista

for (int i = 0; i < lista.Count; i++)
{
  int x = lista[i];
  // Usar x
}
```

Listado 13-3. La eficiencia de esta operación dependerá de la implementación concreta de la lista.

```
IList<int> lista = new List<int>();
// Llenar la lista

for (int i = 0; i < lista.Count; i++)
{
  int x = lista[i];
  // Usar x
}
```

Listado 13-3 Recorriendo una lista usando el indizador

13.2.2 OPERACIONES RELACIONADAS CON LA COLECCIÓN

Las operaciones definidas en la interface `ICollection<T>` permiten ver a una lista como una colección de datos. Aunque en la definición de esta interface no se especifica una noción de orden para los elementos, veremos que debido a la forma en que se definen las listas, las operaciones de `ICollection<T>` serán interpretadas con una definición más restringida que si tendrá en consideración el orden entre los elementos.

13.2.2.1 COUNT

La propiedad `Count` devuelve la cantidad de elementos que hay en la lista. Esta propiedad cambia de valor cada vez que se ejecuta `Insert`, `RemoveAt`, `Add`, o un `Remove` (si éste realmente quitó al elemento porque estaba en la lista). Los valores de índices válidos van entonces de `0` a `Count-1`. Referirse a un índice fuera de este rango dará excepción.

13.2.2.2 ADD

La operación `Add` para el caso particular de una lista adiciona a un elemento a continuación del que ocupa el índice mayor. Teniendo en cuenta que el primer elemento está en la posición 0 y que la cantidad de elementos en la lista es el valor de la propiedad `Count`, entonces el nuevo elemento se ubica en la posición `Count` y el valor de `Count` quedará incrementado en 1. Por ejemplo, si se adiciona el elemento 25 a la lista

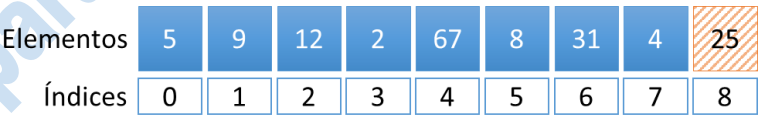


Figura 13.4 Resutado de adicionar una elemento a una lista.

representada en la Figura 13.1, el resultado sería la lista representada en la Figura 13.4.

13.2.2.3 REMOVE

La operación `Remove`, en el caso particular de una lista, quita el elemento de la lista (según la semántica del método `Equals` del tipo de los elementos) y "mueve" los elementos restantes de forma que todos los índices queden consecutivos. En caso de existir más de una ocurrencia del elemento se quita solamente la primera aparición, es

decir, aquel elemento con menor índice. La operación devuelve un valor `bool`, `true` si realmente había un elemento a quitar y `false` en caso contrario.

13.2.2.4 CLEAR

La operación `Clear` vacía la lista, es decir quita todos los elementos de la lista, quedando el valor de `Count` en 0.

Disponer de una operación `Clear` permite que todas las variables que en distintas partes del código estuviesen refiriéndose a la lista puedan quedar a la vez refiriéndose a la lista vacía haciendo simplemente `lista.Clear()` a través de cualquiera de ellas. Debe tenerse esto en cuenta si por el contrario este no es el efecto buscado. Si `lista1` y `lista2` se refiriesen a una misma lista y solo quisiéramos lograr el efecto de vaciar una de ellas entonces debe asignarse una nueva lista a dicha variable haciendo `lista1 = new List()`.

13.2.3 IMPLEMENTACIÓN DE LISTA BASADA EN ARRAY

Debido a la noción de orden que impone la definición de lista por el uso de los índices, la implementación más común y eficiente consiste en un usar internamente un array para almacenar los elementos. De esta forma el índice asociado a cada elemento queda definido explícitamente por el array, y la implementación del indizador, por ejemplo, se simplifica considerablemente. No es de extrañar entonces que la implementación nativa del tipo `List<T>` en la biblioteca estándar de .NET haya escogido esta representación.

Aunque de manera más informal, ya que no se manejó aún el concepto de interface, en el Capítulo 7 se presentó una implementación de lista empleando un array. Por tanto no volveremos a repetir aquí los detalles de implementación de la mayoría de los métodos. De todas formas es conveniente adicionar algunas acotaciones y detalles que no se vieron en el Capítulo 7.

Uno de los problemas fundamentales con los que se debe lidiar a la hora de implementar una lista basada en un array es el relativo al crecimiento de dicho array cuando se quiera añadir o insertar un elemento y el array ya esté lleno. En el Capítulo 7 se presenta una estrategia, que consiste en lograr el efecto de hacer crecer dicho array. El Listado 13-4

```
public void Insert(int index, T item)
{
  if (index < 0 || index > Count)
    throw new ArgumentOutOfRangeException("index");

  if (Count == items.Length)
    Grow();

  if (index < Count)
    Array.Copy(items, index, items, index + 1, Count - index);

  items[index] = item;
  Count++;
}
```

Listado 13-4 Una posible implementación del método Insert.

El método `Grow` (que también es usado por el método `Add`) simplemente crea un array interno más grande y copia todos los elementos hacia el nuevo array. Una desventaja de esta implementación es que todos los elementos de la posición `index` en adelante van ser copiados 2 veces, una vez por el método `Grow`, y luego de nuevo al desplazarlos producto de la inserción. Previendo esto, una implementación más eficiente de `Insert` debería hacer crecer el array y mover los elementos a la vez, de forma que solamente haga falta copiar cada elemento una vez. Esta estrategia se muestra en el Listado 13-5.

```
public void Insert(int index, T item)
{
  if (index < 0 || index > Count)
    throw new ArgumentOutOfRangeException("index");

  T[] origin = items;
  T[] destination = items;

  if (Count == items.Length)
  {
    destination = new T[2 * Count];
    Array.Copy(origin, destination, index);
  }

  destination[index] = item;

  if (index < Count)
    Array.Copy(origin, index, destination, index + 1, Count - index);

  items = destination;
  Count++;
}
```

Listado 13-5 Implementación del método Insert resolviendo el crecimiento y corrimiento de la lista a la misma vez.

Aunque este enfoque es ligeramente más eficiente, la implementación nativa del tipo `List<T>` en .NET no sigue esta estrategia. La ventaja o desventaja de esta decisión es discutible, ya que en la práctica es necesario siempre realizar un balance entre una implementación más eficiente o una más elegante. En particular en la implementación de estructuras de datos, que se usan mucho y para las que la eficiencia es un factor crítico, generalmente la elegancia puede quedar relegada a un segundo plano ya que queda oculta por la interfaz.

13.2.4 IMPLEMENTACIÓN DE LA INTERFACE IENUMERABLE DE LISTA

Tal y como se presenta en el Capítulo 11 existen dos alternativas para implementar el recorrido de un enumerable. La primera consiste en implementar explícitamente la interface `IEnumerator<T>`. La segunda consiste en hacer uso de la instrucción `yield`, dejando al compilador la tarea de generar toda la maquinaria de enumeración. Aunque es obvio que esta segunda opción es más simple, por motivos didácticos presentaremos a continuación ambas formas.

El proceso de iteración a lo largo de una lista debe cumplir dos características: los elementos son "producidos" por el iterador en orden creciente según los índices, y la iteración debe fallar si se produce una modificación a la estructura que está siendo recorrida.

La primera condición es fácil de garantizar en una implementación basada en arrays (como la que nos ocupa ahora), ya que solamente es necesario recorrer el array interno e ir devolviendo consecutivamente los elementos.

Para garantizar la segunda condición, es decir, la consistencia de la estructura durante la iteración, es conveniente introducir un concepto de "versión" en la implementación. Esta versión es simplemente un valor entero que se cambia cada vez que se realiza una operación de modificación de la estructura (`Add`, `Insert`, `Remove`, etc.). De esta forma, es muy fácil y eficiente saber si en un momento de la ejecución la lista ha sido modificada, comparando la versión actual con la versión que tenía la lista al comenzar la iteración.

En el Listado 13-6 se muestra un esbozo de una posible implementación de un tipo `ArrayedList<T>` basado en array (similar a la implementación que se vio en el Capítulo 7). Para implementar de forma explícita el mecanismo de iteración, se define un tipo interno `Enumerator` que contendrá toda la lógica de enumeración. Este tipo implementa la interfaz `IEnumerator<T>`, y es devuelto por los métodos `GetEnumerator()`.

```
public class ArrayedList<T> : IList<T>
{
  private T[] items;
  private int version;

  // ... resto de la clase

  IEnumerator<T> GetEnumerator()
  {
    return new Enumerator(this);
  }

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

  class Enumerator : IEnumerator<T>
  {
    private readonly ArrayedList<T> list;
    private int version;
    private T current;
    private int index = -1;

    public Enumerator(ArrayedList<T> list)
    {
      this.list = list;
      version = list.version;
    }

    //...
  }
```

```
}
```

Listado 13-6 Implementación parcial de la enumeración de una lista.

Nótese cómo en el constructor del enumerador se almacena una referencia a la lista que debe ser enumerada, así como el valor original de la variable `version`. El método `MoveNext`() comienza siempre comprobando que la lista no haya sido modificada, y luego procede a incrementar un índice que va a ser utilizado para acceder al array interno y colocar en `current` el valor correspondiente. Por su parte, la propiedad `Current` simplemente debe devolver el valor previamente almacenado, una vez realizadas todas las validaciones necesarias.

```
class Enumerator : IEnumerator<T>
{
  //...

  public bool MoveNext()
  {
    if (version != list.version)
      throw new InvalidOperationException();

    if (++index >= list.Count)
      return false;

    Current = list.items[index];
    return true;
  }

  public T Current
  {
    get
    {
      if (version != list.version || index < 0
        || index >= list.Count)
        throw new InvalidOperationException();

      return current;
    }
  }

  //...
}
```

Listado 13-7 Implementación de las operaciones MoveNext y Current.

A modo de comparación, en el Listado 13-8 se muestra una implementación alternativa empleando la instrucción `yield`. Compárese la implementación anterior con la simplicidad de esta variante, que expresa de forma mucho más concisa la semántica del recorrido y deja al compilador toda la complejidad asociada a la maquinaria `MoveNext-Current`.

```
IEnumerator<T> GetEnumerator()
{
  int originalVersion = this.version;

  for (int i = 0; i < Count; i++)
```

```
  {
    if (originalVersion != this.version)
      throw new InvalidOperationException();

    yield return items[i];
  }
}
```

Listado 13-8 Implementación de la iteración basada en la instrucción yield.

Esta idea de usar una variable entera `version` para llevar el control de que la lista no pueda modificarse mientras se está recorriendo puede considerarse muy simplista pero es de hecho la que usa por defecto .NET. Lograr que verdaderamente las colecciones sean seguras aún cuando varias hebras trabajen en paralelo con ellas (*thread-safe*) es realmente más complicado. Esto requeriría un costo que no tendrían por qué pagar las aplicaciones que no lo necesitan. La concurrencia y el paralelismo se estudian en el Capítulo 17.

13.3 ESTRUCTURAS ENLAZADAS

Usar arrays para la representación interna de los datos en una estructura de datos (como ha sido el caso de lista) es una estrategia de implementación bastante común. Una representación basada en array brinda la ventaja de un acceso muy eficiente a los elementos a través de los índices y permite una implementación eficiente de muchas operaciones. Sin embargo, puede resultar en costo de tiempo mayor para aquellas operaciones que cambian el orden posicional de los elementos (tales como `Insert` en una lista).

Una representación alternativa para implementar estructuras de datos consiste en usar **enlaces** (*links*) entre las partes de la estructura que llamaremos entonces **estructuras enlazadas**. En este tipo de implementación, en lugar de almacenar solamente los datos se almacenan también enlaces que lo conectan con otros datos o partes de la propia estructura.

En el resto de este capítulo se verán diversas estructuras de datos en cuya implementación se usan elementos enlazables, entre ellas la propia **lista enlazada**.

13.3.1 NODO ENLAZABLE

La definición fundamental en toda estructura enlazable consiste en que lo que almacenan no son solo los datos sino enlaces con otros datos. Esto es lo que se llama un **nodo enlazable** (*linked node*). Cada nodo representa un valor (del tipo de datos que propiamente se quiere guardar en la estructura) y un (o más de uno) enlace con otro (o con otros) nodos (Listado 13-9.

```
public class LinkedNode<T>
{
  public LinkedNode(T value, LinkedNode<T> next)
  {
    Value = value;
    Next = next;
  }

  public T Value { get; set; }
  public LinkedNode<T> Next { get; set; }
}
```

Listado 13-9. Definición de la clase LinkedNode<T>

Nótese que cada nodo almacena un valor (propiedad `Value`) y una referencia a otro nodo del mismo tipo (propiedad `Next`). Por este motivo se dice que este tipo de estructura es "recursiva", en el sentido de que en la propia definición del tipo se hace referencia al propio tipo que se está definiendo.

Empleando únicamente este tipo es posible representar una lista. Para ello en cada nodo se almacena un valor, y una referencia al nodo que contiene al valor siguiente, en el orden definido por la lista. Por ejemplo, para representar una lista con los elementos de la Figura 13.1 se puede emplear el código mostrado en el Listado 13-10.

```
LinkedNode<int> primero =
  new LinkedNode<int>(5,
      new LinkedNode<int>(9,
          new LinkedNode<int>(12,
              new LinkedNode<int>(2,
                  new LinkedNode<int>(67,
                      new LinkedNode<int>(8,
                          new LinkedNode<int>(31,
                              new LinkedNode<int>(4, null)))))));
```

Listado 13-10 Representando una secuencia de elementos a partir de nodos enlazables.

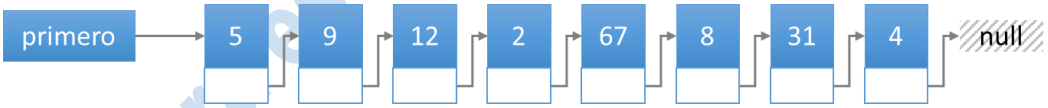


Figura 13.5 Representación gráfica de una secuencia de elementos enlazados.

Aunque este código es suficiente para representar la estructura de la lista, no es conveniente porque no encapsula el resto de la información que hace falta para darle a la lista toda la funcionalidad con la que se ha trabajado anteriormente, pasándole entones esa carga de trabajo al programador que quiere usar una lista. Por ello, es más conveniente definir un tipo `LinkedList<T>`, que emplee internamente esta representación y esconda todas las complejidades relacionadas con la implementación de las operaciones de lista.

13.3.2 IMPLEMENTACIÓN DE LISTAS CON ENLAZABLES

Para implementar las operaciones de `IList` es conveniente representar en la estructura dos referencias, que denominaremos `first` y `last`, al primer y al último nodo (respectivamente) de la lista. El nodo `first` servirá para todas las operaciones que

impliquen realizar un recorrido por los elementos de la lista, mientras que el nodo `last` servirá para acelerar la operación de adicionar un elemento al final de la lista. En el Listado 13-11 se muestra la definición parcial de esta clase (a la que deben adicionarse todos los métodos de la interface `IList<T>`), y en la Figura 13.6 una representación gráfica de una lista con varios elementos ya almacenados.

```
private class LinkedList<T> : IList<T>
{
  private LinkedNode<T> first;
  private LinkedNode<T> last;
  private int count;
  //...
}
```

Listado 13-11. Definición de la clase LinkedList<T>.

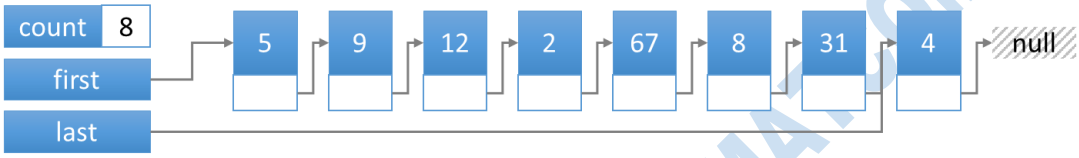


Figura 13.6 Representación gráfica de una lista enlazada.

La operación de adición (método `Add`) queda ahora muy simple y, a diferencia de la implementación basada en arrays, no necesita hacer crecer el array. El único caso particular que debe ser tenido en cuenta es cuando la lista se encuentra inicialmente vacía (Listado 13-15).

```
public void Add(T item)
{
  if (count == 0)
    first = last = new LinkedNode<T>(item, null);
  else
    last = last.Next = new LinkedNode<T>(item, null);
  count++;
}
```

Listado 13-12. Implementación de la operación Add para listas enlazadas.

Sin embargo, para el resto de las operaciones, que requieren de encontrar el nodo asociado a un índice particular, es necesario realizar un recorrido por la lista. Para ello será conveniente definir un método auxiliar `FindNode` (Listado 13-13). La búsqueda se realiza empleando un **cursor**, que no es más que una variable de tipo `LinkedNode<T>` cuyo valor se refiere al nodo actual con el que se va a trabajar (se suele decir "el nodo sobre el que está el cursor"). Este **cursor** se modifica en cada iteración hasta que se cumpla la condición de parada (en este caso, que se alcance el índice deseado).

```
private LinkedNode<T> FindNode(int index)
{
  LinkedNode<T> current = first;

  for (int i = 0; i < index; i++)
    current = current.Next;

  return current;
```

```
}
```

Listado 13-13 Búsqueda iterativa del nodo asociado a un índice particular

Empleando este método auxiliar es muy sencillo implementar el indizador (Listado 13-13 Búsqueda iterativa del nodo asociado a un índice particular

).

```
public T this[int index]
{
  get
  {
    if (index < 0 || index >= Count)
      throw new IndexOutOfRangeException();

    return FindNode(index).Value;
  }
  set
  {
    if (index < 0 || index >= Count)
      throw new InvalidOperationException();

    FindNode(index).Value = value;
  }
}
```

Listado 13-14 Implementación del indizador para una lista enlazada.

Es importante destacar que, aunque es posible emplear el indizador para acceder a un elemento de la lista mediante un índice esto es no se hace con acceso directo como en el caso de una lista implementada con array ya que habría que recorrer la lista a través de los enlaces hasta llegar a la posición indicada (lo que tendría un costo $O(n)$ y no constante como cuando se usa una implementación con array).

De manera similar se implementan las operaciones de modificación de la lista que dependen de índices. La operación `Insert`, por ejemplo, solamente tiene que lidiar con los casos particulares en que se inserte al inicio o al final, o que la lista esté vacía. En estos casos simplemente se crea el nodo (o nodos) correspondiente y se actualizan los campos first y/o last. En cualquier otro caso, solamente deben actualizarse las referencias del nuevo nodo a insertar, y de su antecesor (Listado 13-15).

```
public void Insert(int index, T item)
{
  if (index < 0 || index > Count)
    throw new IndexOutOfRangeException();

  if (Count == 0)
    first = last = new LinkedNode<T>(item, null);
  else if (index == 0)
    first = new LinkedNode<T>(item, first);
  else if (index == Count)
    last = last.Next = new LinkedNode<T>(item, null);
  else
  {
    LinkedNode<T> node = FindNode(index);
```

```
      node.Next = new LinkedNode<T>(item, node.Next);
    }

    count++;
  }
```

Listado 13-15. Implementación de la operación Insert para listas enlazadas.

Nótese que, debido a la naturaleza de los nodos enlazables, no es necesario "desplazar" los elementos una vez se produzca la inserción La Figura 13.7 muestra una representación gráfica de esta operación.

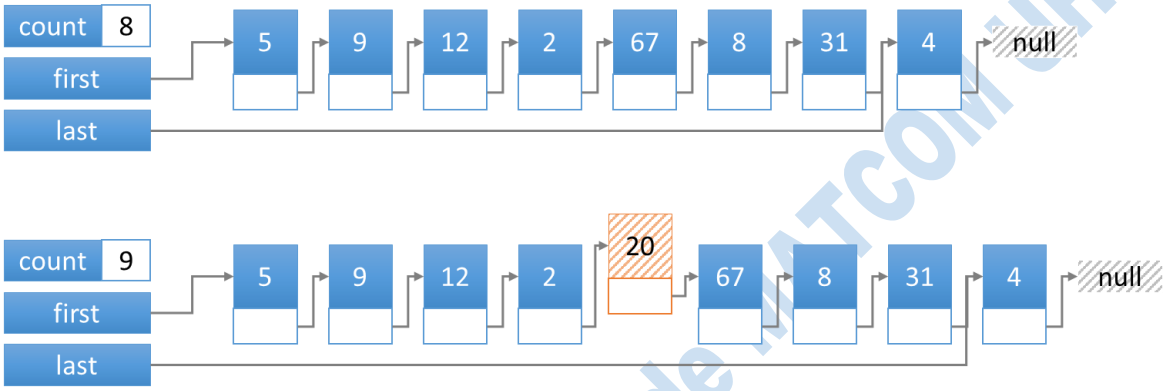


Figura 13.7. Inserción de un elemento en una lista enlazada.

De forma similar, la operación `RemoveAt` se implementa a partir de encontrar el nodo correspondiente al índice inmediatamente anterior al que se desea eliminar, y a partir de ahí actualizar las referencias necesarias. Como casos particulares, cuando se elimina el primer o el último elemento de la lista, es necesario actualizar los campos **first** y/o **last**. Otro caso particular es cuando la lista tiene un solo elemento y éste es eliminado. En el Listado 13-16 se muestra una implementación de este método, y en la Figura 13.8 una representación gráfica de esta operación para el caso general en que se elimina un elemento del medio de la lista.

```
public void RemoveAt(int index)
{
  if (index < 0 || index >= Count)
    throw new IndexOutOfRangeException();

  if (Count == 1)
    first = last = null;
  else if (index == 0)
    first = first.Next;
  else
  {
    var node = FindNode(index - 1);
    node.Next = node.Next.Next;

    if (index == Count - 1)
      last = node;
  }

  count--;
```

```
}
```

Listado 13-16 Eliminando un nodo a partir de un índice en una lista enlazada.

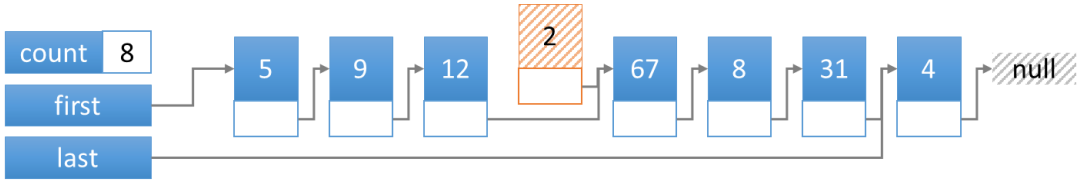


Figura 13.8 Eliminación de un elemento de una lista enlazada.

La operación `IndexOf` se implementa de forma similar a `FindNode`, pero buscando el nodo según el valor que almacena (Listado 13-17).

```
public int IndexOf(T item)
{
  LinkedNode<T> current = first;
  int i = 0;

  while (current != null)
  {
    if (Equals(current.Value, item))
      return i;

    i++;
    current = current.Next;
  }

  return -1;
}
```

Listado 13-17 Implementación de IndexOf en una lista enlazada.

A partir de esta operación es sencillo implementar el método `Contains`. El método `Remove`, por su parte, requiere de una implementación un poco más cuidadosa, pues una vez que se encuentra el nodo que se desea quitar de la lista, es necesario actualizar la referencia a éste que tenía el nodo anterior. Por tanto, se deben tener dos cursores, uno para el nodo que se desea encontrar, y otro para el nodo anterior. Se propone al lector como ejercicio implementar estos dos métodos, teniendo especial cuidado con los casos particulares en los que el elemento a quitar es el primero o el último nodo de la lista.

Otra operación interesante para comparar con la implementación basada en array es la enumeración. A continuación se verán tanto la versión explícita implementando la interface `IEnumerator<T>` como la versión reducida empleando la instrucción `yield`.

En la enumeración de una lista empleando enlazables se vuelve a usar el concepto de cursor. Debido a que la maquinaria del iterador requiere que se almacene el estado actual de la iteración, es muy útil mantener una referencia al nodo que cuyo valor debe ser devuelto al consultar la propiedad `Current`. Este nodo también permite continuar la iteración mediante la referencia al siguiente nodo. Al igual que en la implementación basada en arrays, se adiciona un campo `version` que será incrementado en cada operación que modifica la lista (`Add`, `Insert`, `RemoveAt`, `Remove`, etc.). En el Listado 13-18 se muestra la implementación de los métodos `MoveNext` y `Current` para un tipo

`LinkedEnumerator` interno a la clase `LinkedList<T>`. El resto de la implementación es muy similar a su homólogo basado en array. La diferencia fundamental es la utilización de una variable `started` para desambiguar entre el caso en que `node` es `null` porque aún no ha comenzado la iteración y el caso en que la iteración ya terminó.

```
class LinkedEnumerator : IEnumerator<T>
{
  private LinkedList<T> list;
  private LinkedNode<T> node;
  private bool started = false;
  private int version;

  public bool MoveNext()
  {
    if (version != list.version)
      throw new InvalidOperationException();

    if (!started)
      node = list.first;
    else
      node = node.Next;

    started = true;
    return node != null;
  }

  public T Current
  {
    get
    {
      if (version != list.version || node == null)
        throw new InvalidOperationException();

      return node.Value;
    }
  }

  //...
}
```

Listado 13-18 Iterador para una lista enlazada.

El Listado 13-19 muestra la misma implementación pero empleando la instrucción `yield`. Nótese que en este caso se ha empleado un ciclo `for` aun cuando la variable de control del ciclo no es un entero. Esto se debe a que existe una noción clara de inicialización, condición de parada e incremento, por lo que un ciclo `for` expresa mejor la semántica de la iteración que el código equivalente empleando un ciclo `while`.

```
public IEnumerator<T> GetEnumerator()
{
  int originalVersion = version;

  for (LinkedNode<T> current = first; current != null;
        current = current.Next)
  {
```

```
      if (originalVersion != version)
        throw new InvalidOperationException();

      yield return current.Value;
    }
  }
```

Listado 13-19 Implementación de un iterador para lista enlazada empleando la instrucción yield.

13.3.3 LISTA DOBLEMENTE ENLAZABLE

La implementación anterior usa enlaces simples para navegar de un elemento al siguiente. Debido a esta disposición de los enlaces, la lista solo se puede recorrer en una dirección, desde el primer elemento hasta el último. Si se quisiera recorrer en el sentido contrario habría que hacerlo a través de los índices, lo que sería en este caso muy ineficiente).

Si fuese conveniente poder recorrer una lista en ambas direcciones, así como determinar eficientemente dado un valor cuál es el nodo anterior al nodo que tiene dicho valor, sería útil que un nodo también tuviera un segundo enlace (Listado 13-20).

Esto permitiría, por ejemplo, simplificar la implementación de un método como `Remove`. Se le propone como ejercicio modificar todos los métodos vistos hasta el momento en `LinkedList` para implementar una lista doblemente enlazada (`BiLinkedList`).

```
public class BiLinkedNode<T>
{
  public LinkedNode(T value, BiLinkedNode<T> previous, BiLinkedNode<T> next)
  {
    Value = value;
    Previous = previous;
    Next = next;
  }

  public T Value { get; set; }
  public BiLinkedNode<T> Previous { get; set; }
  public BiLinkedNode<T> Next { get; set; }
}
```

Listado 13-20 Definición de nodo con dos enlaces

Implementaciones de estructuras de datos como los árboles (Capítulo 14) pueden usar nodos con más de un enlace.

13.3.4 ¿ARRAYS O NODOS ENLAZABLES?

Una implementación de una estructura de datos (como ha sido la de lista en este caso) mediante arrays requiere manejar el tamaño del array que almacenará los elementos y hacerlo crecer o decrecer según sea necesario o conveniente. Si en la aplicación se va a trabajar con una lista que por lo general se crea de una vez, sin hacer adiciones frecuentes, pero a la que se accede continuamente a sus elementos, puede ser mejor usar una implementación basada en arrays. Pero si es una lista a la que con frecuencia se le añaden y quitan elementos, los cuales no son accedidos tan frecuentemente por su

posición (índice), entonces puede que venga mejor usar una implementación basada en enlaces.

Una ventaja de las listas enlazadas es la eficiencia con la que los objetos pueden ser agregados a la colección, dado que solo se necesita crear un nuevo nodo y actualizar algunas referencias. Sin embargo, la operación de acceder al elemento en una posición implica recorrer la lista hasta llegar a la posición.

Ciertas operaciones son más costosas en una implementación que en otra. Por ejemplo, las operaciones de insertar y eliminar requieren una cantidad constante de pasos para una lista enlazada (solo las relativas a la de alcanzar el elemento). Por otra parte en el caso de una lista implementada con arrays requiere un tiempo proporcional a la cantidad de elementos que haya después de la posición ya que habría que hacer corrimiento de los elementos dentro del array (además de eventualmente crecer si el array está lleno).

La implementación con array puede subutilizar espacio si el tamaño del array es muy grande en comparación con la cantidad de elementos que realmente serán almacenados en la lista. Por otro lado la implementación con enlaces solo ocupa espacio según la cantidad de elementos almacenados, aunque por cada valor almacenado en la lista se ocupa el espacio adicional para la referencia o enlace con el siguiente (y con el anterior si es doblemente enlazada).

Estas consideraciones entre implementación con arrays e implementación con enlaces son también válidas para las diferentes estructuras de datos que estudian en el resto de este capítulo y por consiguiente no volveremos a insistir en este tema.

Tanto los arrays como las listas son estructuras de datos de propósito general y podría decirse que basándose en ambos pudiera implementarse cualquier aplicación. Sin embargo, muchas aplicaciones requieren de un comportamiento más específico de una estructura de datos. En lo que sigue de este capítulo se verán algunas de las más relevantes como las pilas, colas y diccionarios.

13.4 PILAS

Una **pila** (*stack*) es un tipo de estructura de datos en la que toda inserción y eliminación se realizan por un mismo lugar que se denomina **tope** de la pila. Este tipo de estructura es conocida como estructura **Last In First Out** (LIFO) es decir, **el último elemento que se ha guardado en la pila es el primero en extraerse**.

Hay muchas agrupaciones que en la práctica real funcionan de esta manera. Por ejemplo la forma en que se apilan libros sobre una mesa cuando son devueltos a un bibliotecario que luego los distribuirá en sus estantes, la forma en que se apilan los platos en un fregadero, la forma en que se introducen y salen los cartuchos en el cargador de un fusil. En estos casos los objetos se ponen y se extraen por un mismo lugar.

Como se ilustró en el epígrafe de cómo se ejecuta internamente la recursividad (Capítulo 8) las pilas son utilizadas en el trabajo interno de los sistemas operativos y compiladores. Cuando un método `M` invoca a un método `F`, al comenzar la ejecución del método llamado `F` se ubican en el tope de una pila los datos locales que usa ese método `F` junto con la ubicación dentro del código de `M` hacia donde debe regresarse al terminar `F`, de

modo que cuando `F` termina su ejecución y se quitan de la pila los datos de `F`, quedarán en el tope los datos del método `M` que fue quién llamó a `F`. Gracias a esta forma de apilamiento es que es posible implementar un método recursivo `R`.

Es por ello frecuente la excepción conocida como "stack overflow" (desbordamiento de la pila) generalmente provocada cuando por un error de programación un método recursivo cae en un ciclo infinito sin alcanzar un caso base.

Si a usted aún no le ha ocurrido esta excepción es que todavía ha programado lo suficiente.

Un tipo interface que caracterizaría el funcionamiento de una pila pudiera ser el del Listado 13-21 Interface IStack

. Note que una pila no es, de forma general, una colección, es decir no se ha indicado que implementa el tipo `ICollection` ya que la semántica de sus operaciones no considera que se puede quitar un elemento cualquiera arbitrario, sino solamente el elemento ubicado en el tope. Por este motivo nuestra **interface** `IStack<T>` implementa directamente `IEnumerable<T>`.

```
public interface IStack<T> : IEnumerable<T>
{
  int Count { get; }
  void Push(T item);
  T Pop();
  T Peek();
}
```

Listado 13-21 Interface IStack

A continuación se presentan las operaciones fundamentales de esta estructura de datos.

13.4.1 PUSH

La operación de **empilar** (generalmente denominada en inglés como `Push` de empujar) toma como argumento un elemento `x` y lo pone en el "tope" de la pila. Puede represéntaselo visualmente como que el elemento que se encontraba en el tope antes de la operación queda "debajo" de `x`. La Figura 13.9 ilustra el resultado de colocar el elemento 5 en una pila.

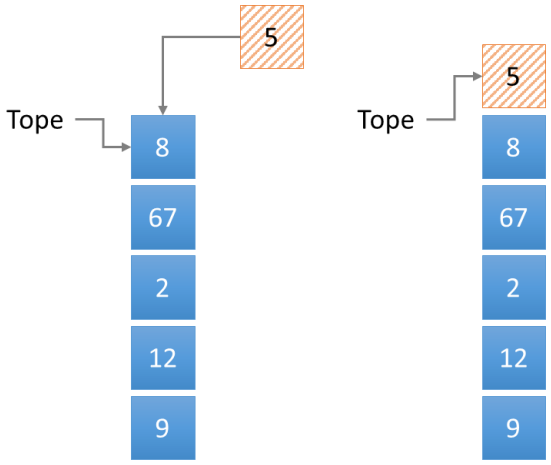


Figura 13.9. Inserción (push) de un elemento en una pila.

13.4.2 POP

La operación quitar de la pila o **desempilar** (en inglés `Pop`[1]) devuelve como resultado el elemento que se encuentra en el tope de la pila y a su vez lo quita de la pila. Si la pila está vacía el intento de aplicar esta operación provocará una excepción. La Figura 13.10 muestra un ejemplo de esta operación.

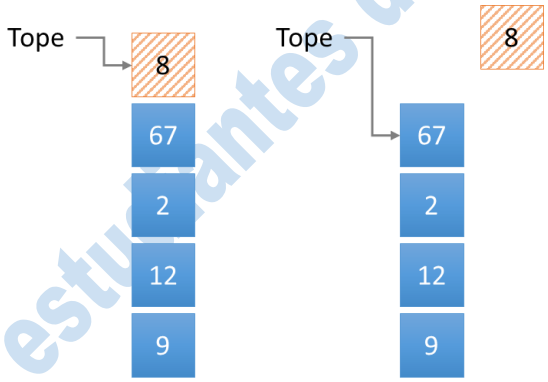


Figura 13.10. Eliminación (pop) del tope de la pila.

13.4.3 PEEK

La operación **tope** (`Peek`) devuelve el elemento que se encuentra en el tope de la pila sin quitarlo de la misma. Al igual que la operación `Pop`, debe lanzar una excepción si se pretende aplicar en el caso que la pila esté vacía.

[1] Por imitación del sonido que hace una burbuja al llegar a la superficie y estallar.

En una estructura de datos bien diseñada no pueden ocurrir excepciones en el uso de la estructura que no puedan ser prevenidas por operaciones que ofrezca la propia estructura.

Por ejemplo la excepción que pueden dar `Peek` y `Pop`, si se intentan aplicar a una pila `p` vacía puede evitarse si antes se pregunta por `if (p.Count!=0)`.

Esto suelen denominarse precondiciones (expresiones lógicas que deben evaluar true) que deben cumplirse para que un método pueda aplicarse.

La documentación de los tipos de datos en la biblioteca de .NET está plagada de este tipo de información expresada en términos de comentarios llamados contratos. Hay lenguajes de programación en los que este tipo de recursos está integrado sintácticamente en el lenguaje. C# por ahora no tiene incorporado este tipo de recursos aunque se han desarrollado algunas extensiones para lograr ese efecto (como Code Contracts).

En el espacio de nombres `System.Collections.Generic` está la clase `Stack<T>`, que brinda una implementación de esta esta estructura de datos pila. A continuación se presentan algunos ejemplos de utilización de las pilas usando este tipo `Stack`

13.4.4 EJEMPLO DE USO DE PILA. PARÉNTESIS BALANCEADOS

Muchos de los problemas interesantes a resolver con pilas involucran el análisis de estructuras anidadas. Un ejemplo típico es comprobar si una cadena de texto que contiene paréntesis abiertos `'('` y cerrados `')'` está balanceada, es decir, cada paréntesis que abre tiene su pareja que lo cierra, hay la misma cantidad de abiertos que cerrados y nunca aparece un cerrado antes que su correspondiente abierto.

Por ejemplo la cadena `a * (b + c / f(x)) - m(x, h(y))` tiene los paréntesis balanceados, pero la cadena `b + (c / f)(x` no los tiene balanceados. Una solución sencilla para este problema se muestra en el Listado 13-22. Dicha solución se basa en mantener un valor entero de balance, que indica la cantidad de paréntesis abiertos que aún no han sido cerrados. Se recorre la cadena y balance se incrementa en 1 cada vez que se encuentra un paréntesis abierto y se decrementa cuando se encuentra un cerrado. Si balance llega a ser negativo, eso significa que la cadena no está balanceada porque han aparecido más paréntesis cerrados que abiertos. Al llegar al final de la cadena si balance es cero quiere decir que los paréntesis están balanceados, y en caso contrario no lo están.

```
static bool Balanceados(string s)
{
  int balance = 0;

  foreach (char c in s)
  {
    if (c == '(')
      balance ++;
    else if (c == ')')
      balance--;

    if (balance < 0)
      return false;
  }
```

```
  return balance == 0;
}
```

Listado 13-22 Comprobando si una cadena de paréntesis está balanceada.

Aunque esta solución es muy sencilla no se puede generalizar si queremos saber si los paréntesis están balanceados cuando en una cadena puede haber más de una pareja de paréntesis de diferentes tipos. Por ejemplo '('y ')', '['y ']', '{' y '}'.

En este caso, cuando aparece un paréntesis que cierra el último paréntesis encontrado debe haber sido su correspondiente abierto.

El listado Listado 13-23 muestra una solución usando una pila. La cadena se recorre de izquierda a derecha, si nos encontramos un paréntesis abierto este se empila, si nos encontramos un paréntesis cerrado entonces en el tope de la pila debe estar su correspondiente abierto y en tal caso éste se quita de la pila. Si lo que está en el tope de la pila no es el correspondiente abierto entonces los paréntesis no están balanceados. Cualquier carácter que no sea paréntesis será ignorado. Luego de haber recorrido toda la cadena si la pila está vacía es que sí están balanceados, de lo contrario no están balanceados porque habrán aparecido más abiertos que sus respectivos cerrados.

```
public bool Balanceados(string s)
{
  var stack = new Stack<char>();

  foreach (var c in s)
  {
    if (c == '(' || c == '[' || c == '{')
      stack.Push(c);

    if (c == ')')
    {
      if (stack.Count == 0 || stack.Peek() != '(')
        return false;

      stack.Pop();
    }
    else if (c == ']')
    {
      if (stack.Count == 0 || stack.Peek() != '[')
        return false;

      stack.Pop();
    }
    else if (c == '}')
    {
      if (stack.Count == 0 || stack.Peek() != '{')
        return false;

      stack.Pop();
    }
  }

  return stack.Count == 0;
```

```
}
```

Listado 13-23 Verificando si una cadena está balanceada mediante el empleo de una pila.

13.4.5 PILA Y RECURSIVIDAD. INVERTIR UNA SECUENCIA

Otro problema donde es conveniente emplear una pila es para invertir una secuencia de valores. Los valores se van guardando en la pila según se van obteniendo de la secuencia (por ejemplo leyéndolos de `Console`) de modo que va quedando en el tope el último leído. Una vez que se termina la secuencia entonces los valores se van sacando de la pila (en este caso para escribirlos a `Console`) de modo que se reproduce la secuencia pero en orden inverso (Listado 13-24

```
static void Invierte()
{
  Stack<string> stack = new Stack<string>();
  string s;

  while (!string.IsNullOrEmpty(s = Console.ReadLine()))
    stack.Push(s);

  while (stack.Count > 0)
    Console.WriteLine(stack.Pop());
}
```

Listado 13-24. Método que invierte una secuencia de elementos utilizando una pila.

Este problema también puede ser resuelto muy fácilmente, y con una notación más abreviada, usando recursividad (Listado 13-25)

```
static void Invierte()
{
    string s = Console.ReadLine();

    if (!string.IsNullOrEmpty(s))
    {
        Invierte();
        Console.WriteLine(s);
    }
}
```

Listado 13-25 Implementación recursiva del método para invertir una secuencia de elementos.

Esta dualidad de soluciones, empleando recursividad y empleando pilas, no es casual. Como se explica en el epígrafe **Cómo ejecuta internamente la recursividad** del Capítulo 8, la implementación de la recursividad utiliza internamente una pila para guardar, y recuperar al regresar, la información que está utilizando cuando se hace una llamada recursiva. Esto se ilustrará mejor en el ejemplo a continuación.

13.4.6 PILA Y RECURSIVIDAD. LAS TORRES DE HANOI

Retomemos el problema de las Torres de Hanoi (Capítulo 8 epígrafe 8.2) El Listado 13-26 muestra la solución recursiva.

```
static void Hanoi(int n, string  orig, string dest, string aux) {
  if (n == 1)
```

```
    Console.WriteLine("Mover de " + orig + " a " + dest);
  else
  {
    Hanoi(n - 1, orig, aux, dest);
    Hanoi(1, orig, dest, aux);
    Hanoi(n - 1, aux, dest, orig);
  }
}
```

Listado 13-26 Implementación recursiva del problema de las Torres de Hanoi.

Para diseñar una solución no recursiva para este problema, es conveniente analizar qué información necesita ser manejada entre una llamada recursiva y otra. La recursividad empila en cada llamada el valor de los parámetros `n`, `orig`, `dest` y `aux` en lo que fue denominado **registro de activación**. Esto lo modelaremos con el tipo `RA` (Listado 13-27).

```
public struct RA
{
    public string Origen { get; set; }
    public string Destino { get; set; }
    public string Auxiliar { get; set; }
    public int N { get; set; }
}
```

Listado 13-27 Estructura para representar el equivalente al registro de información en una llamada recursiva de las Torres de Hanoi.

El método recursivo del Listado 13-26 lo convertimos en el Listado 13-28.

Se comienza empilando un objeto de tipo `RA` con los valores de los parámetros del método. Luego se comienza un ciclo mientras la pila no esté vacía (`s.Count > 0`).

En cada iteración del ciclo se saca de la pila el valor de tipo `RA` que está en el tope. Si el valor de la propiedad `N` es `1`, quiere decir que se debe hacer un movimiento de un disco (en tal caso se escribe la información correspondiente) y se continúa el ciclo. Si el valor de `N` es diferente de cero, entonces se empilan 3 objetos de tipo `RA` correspondientes a las llamadas

```
Hanoi(n - 1, orig, aux, dest);
Hanoi(1, orig, dest, aux);
Hanoi(n - 1, aux, dest, orig);
```

del código recursivo pero se empilan en orden inverso para que quede en el tope de la pila la primera llamada en terminar su secuencia recursiva.

Cuando la pila quede vacía se habrán terminado todos los movimientos.

```
static void HanoiConPila(int n, string orig, string dest, string aux)
{
  Stack<RA> s = new Stack<RA>();
  s.Push(new RA { N = n, Origen = orig, Destino = dest, Auxiliar = aux});

  while (s.Count > 0)
  {
      RA ra = s.Pop();
      if (ra.N == 1)
```

```
            Console.WriteLine("Mueve de {0} a {1}", ra.Origen, ra.Destino);
        else
        {
            s.Push(new RA {N = ra.N - 1, Origen = ra.Auxiliar,
                           Destino = ra.Destino, Auxiliar = ra.Origen});
            s.Push(new RA {N = 1, Origen = ra.Origen,
                           Destino = ra.Destino, Auxiliar = ra.Auxiliar});
            s.Push(new RA {N = ra.N - 1, Origen = ra.Origen,
                           Destino = ra.Auxiliar, Auxiliar = ra.Destino});
        }
    }
}
```

Listado 13-28 Implementación de la solución al problema de las Torres de Hanoi simulando la recursión con una pila.

Si al llegar aquí no ha entendido esta solución, vuélvalo a mirar y ejecútela manualmente con paciencia. Realmente esta implementación es mucho más complicada de entender que la recursividad. Precisamente, en esto radica la magia de la recursividad, en que oculta en su implementación toda esta maquinaria de trabajo con la pila. Posiblemente si no hubiésemos pensado en la solución recursiva no hubiésemos encontrado esta solución "iterativa".

Algo similar ocurre en la implementación interna del funcionamiento de la instrucción `yield` con los `IEnumerables`. El compilador genera un código que usa una pila para mantener el estado y llevar el control de la iteración.

En general, cuando se diseña un lenguaje de programación (tal es el caso de C#) se intenta expresar en términos sintácticos simples los patrones que ocultan procesos repetitivos y tediosos de escribir y por tanto nos previenen de cometer errores. Mejores abstracciones permite a su vez atacar problemas más complejos, que aunque fueran solubles teóricamente con lenguajes más elementales, la complejidad que eso implicaría haría más improductiva la tarea.

13.4.7 IMPLEMENTACIÓN DE PILA BASADA EN ARRAYS

Al igual que en el caso de las listas, una representación basada en arrays permite organizar convenientemente los elementos de una pila. La implementación del Listado 13-29 mantiene internamente un índice que indica la posición en el array inmediatamente después del tope de la pila, es decir, la posición donde se debiera colocar un nuevo elemento si se hace un `Push`.

Como la pila crece y decrece por el mismo extremo, solamente hace falta mantener este índice, que sirve a la vez para obtener el elemento si se hace `Peek` o `Pop`. La misma propiedad `Count` nos sirve de índice para acceder al tope de la pila y para darnos la cantidad de elementos que hay en la pila.

Inicialmente se comienza con un array de determinado tamaño que puede hacerse crecer cuando se llene (con una estrategia similar a la que se hizo al implementar listas basadas en array).

```
public class Stack<T> : IStack<T>
{
  private T[] elements;

  public int Count { get; private set; }

  public Stack()
  {
    elements = new T[4];
  }

  public void Push(T item)
  {
    if (Count == elements.Length)
      Grow();

    elements[Count++] = item;
  }

  private void Grow()
  {
    //...
  }

  // resto de la implementación
}
```

Listado 13-29. Definición de la clase Stack<T>.

Vea la implementación de los métodos `Peek` y `Pop` en el Listado 13-30. En ambos casos el elemento que se devuelve está en la posición `Count - 1`. La única diferencia entre ambos es que el método `Pop` debe dejar decrementado el valor de `Count` para dar el efecto de haber quitado el elemento de la pila.

Se deja como ejercicio al lector implementar el enumerador, teniendo en cuenta adicionar el manejo de versión en los métodos que lo requieran.

```
public T Pop()
{
  if (Count == 0)
    throw new InvalidOperationException();

  return elements[--Count];
}

public T Peek()
{
  if (Count == 0)
    throw new InvalidOperationException();

  return elements[Count - 1];
}
```

Listado 13-30. Implementación de los métodos Pop y Peek basados en array.

13.4.8 IMPLEMENTACIÓN DE PILA CON NODOS ENLAZABLES

Al igual para las listas, se puede hacer una implementación de pila usando nodos enlazables.

En este caso hacer `Push`, es decir poner un elemento en el tope de la pila, implicaría ponerlo a la cabeza de la secuencia enlazada y actualizar la propiedad `Next` del nodo que ahora se pone en el tope con el valor que antes estaba como tope (Listado 13-31).

Usar enlazables en la implementación de una pila brinda la ventaja de que no es necesario hacer crecer un array. Por otro lado, al no haber operaciones relacionadas con índices, no se carga con la desventaja de tener que recorrer todos los nodos.

```
public class LinkedStack<T> : IStack<T>
{
  private LinkedNode<T> top;
  public int Count { get; private set; }

  public void Push(T item)
  {
    top = new LinkedNode<T>(item, top);
    Count++;
  }

  public T Pop()
  {
    if (Count == 0)
      throw new InvalidOperationException();

    T result = top.Value;
    Count--;
    top = top.Next;
    return result;
  }

  public T Peek()
  {
    if (Count == 0)
      throw new InvalidOperationException();

    return top.Value;
  }

  //... resto de la implementación
}
```

Listado 13-31. Implementación de la clase Stack<T> con nodos enlazados.

Aunque por objetivos didácticos se ha usado en este epígrafe una hipotético tipo interface `IStack`, tal tipo no existe realmente en la biblioteca de .NET. Tal parece que los implementadores de .NET no están interesados en que los programadores definan sus implementaciones de un tal `IStack`, sino que usen la clase `Stack` que se proporciona y cuya implementación muy eficiente está basada en arrays. Incluso por esa misma razón debe ser que ningún miembro de la clase `Stack` es `virtual` para que el programador no se vea tentado a suplantar el comportamiento de `Stack` heredando y redefiniendo sus métodos.

En la práctica el uso mayoritario de pilas está subsumido en el código de más bajo nivel que generan los compiladores y otras herramientas a nivel de sistema. Una buena parte de las situaciones en que podrían usarse pilas puede resolverse con la recursividad. En ocasiones en que se esté buscando mucha eficiencia, tal vez valga la pena que se monte su propia implementación de pila orientada al tipo de datos que va a almacenar.

13.5 COLAS

Una **cola** (*queue*) es un tipo de estructura de datos en que los elementos se almacenan por un "extremo" y se extraen por el otro extremo siguiendo un orden que se conoce como **el primero en entrar es el primero en salir** (conocido por las siglas FIFO del inglés First In First Out).

Las operaciones que se pueden realizar sobre una cola son análogas a las de una pila, con la diferencia de que las inserciones se realizan por el extremo contrario a por donde se realizan las extracciones (Figura 13.11). El nombre de la operación de consulta es `Peek`, similar al que se usa en pila. Pero los nombres de las operaciones que modifican la cola son `Enqueue` y `Dequeue`, diferentes a `Push` y `Pop` de pila para dar bien la idea del significado de las mismas y marcar la diferencia con las pilas.

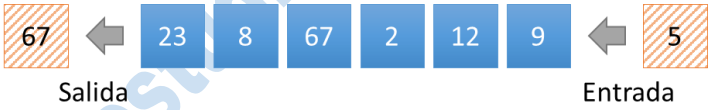


Figura 13.11 Representación gráfica de una cola.

Inspirada en el concepto de cola del mundo real, una cola sirve de mecanismo de organización y sincronización entre **productores** y **consumidores** (ver Capítulo 17) cuando el ritmo de producción y de consumo no son los mismos. El consumidor se pone en una cola en espera por un servicio del productor si no hay un productor disponible en el momento en que solicita el servicio (por ejemplo, la cola de *check-in* en un aeropuerto). De forma análoga, el productor puede poner sus productos en una cola a la espera de un consumidor (por ejemplo, los correos electrónicos escritos por los usuarios se ponen en cola para que el servidor de email los envíe).

Las colas como estructuras de datos son por tanto muy utilizadas en las aplicaciones computacionales donde quiera que sea necesario coordinar dos partes de la aplicación, una que produce datos y otra que los consume.

Al igual que para el caso de las pilas, en la biblioteca estándar de .NET no existe una interface para representar una cola de forma abstracta. El Listado 13-32

```
public interface IQueue<T> : IEnumerable<T>
```

```
{
  int Count { get; }
  void Enqueue(T item);
  T Dequeue();
  T Peek();
  //...
}
```

Listado 13-32 Definición de la interface IQueue.

13.5.1 ENQUEUE

La operación `Enqueue` es similar a la operación `Push` para las pilas. Toma como argumento un elemento `x` y lo coloca en al final de la cola. La Figura 13.12 muestra un ejemplo de la esta operación en donde se inserta el elemento `5` en el extremo posterior de la cola.

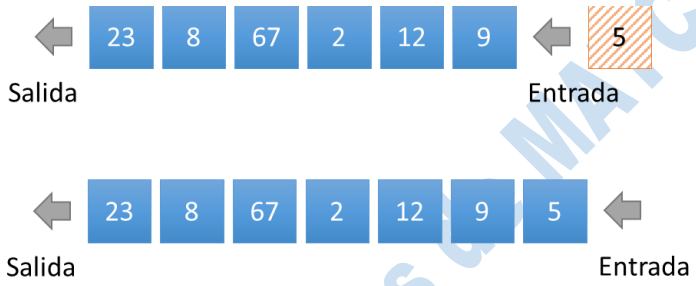


Figura 13.12. Inserción del elemento 5 en la cola.

13.5.2 DEQUEUE

La operación `Dequeue` extrae el primer elemento de la cola devolviéndolo al que realiza la operación. Esta operación exige que la cola no esté vacía al momento de realizar la operación. Sería la homóloga a la operación `Pop` en el caso de pila. La Figura 13.13 muestra la eliminación del frente de la cola.

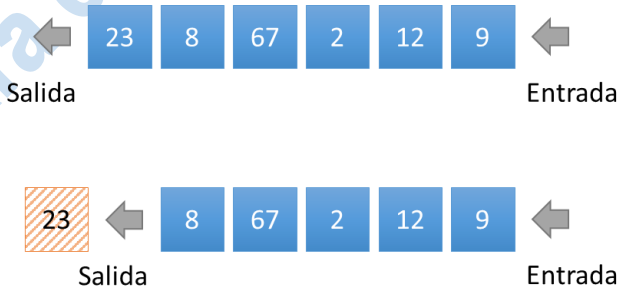


Figura 13.13. Eliminación del elemento al inicio de la cola.

13.5.3 PEEK

La operación `Peek`, de forma similar a su homólogo para las pilas, devuelve al elemento que se encuentra al inicio de la cola, pero sin extraerlo. Al igual que `Enqueue` esta operación requiere que la cola no se encuentre vacía.

13.5.4 EJEMPLO DE USO DE UNA COLA. EL CONJUNTO DE WIRTH

Ejemplifiquemos la utilización de una cola usando la clase `Queue<T>` del espacio de nombres `System.Collection.Generic`.

Nos interesa definir un iterador que produzca los elementos del conjunto de Wirth[2]. El conjunto de Wirth es un conjunto de números enteros que se define de forma recurrente, a partir de los siguientes axiomas.

1. El número 1 pertenece al conjunto.
2. Si el número x pertenece al conjunto, entonces también pertenecen $2x + 1$ y $3x + 1$.
3. Todo número que pertenece al conjunto tiene que cumplir con una de las dos condiciones anteriores.

A partir de esta definición es fácil realizar una implementación recursiva que permita determinar si un número pertenece o no a este conjunto (Listado 13-33).

```
static bool PerteneceWirth(long x)
{
  if (x < 1)
    return false;
  if (x == 1)
    return true;
  if ((x - 1) % 2 == 0 && PerteneceWirth((x - 1) / 2))
    return true;
  if ((x - 1) % 3 == 0 && PerteneceWirth((x - 1) / 3))
    return true;
  return false;
}
```

Listado 13-33 Determinando si un número pertenece al conjunto de Wirth.

Aunque esta implementación es efectiva para determinar si un número particular está en el conjunto deseado, es ineficiente si lo que se desea es definir un enumerador que produzca los números que pertenecen a dicho conjunto (Listado 13-34). Esto se debe a que el porciento de números que pertenecen a este conjunto disminuye considerablemente a medida que aumenta la cantidad a buscar; por ejemplo, del primer millón de enteros solo pertenecen al conjunto 43310, o sea, solo un 4.3%.

```
static IEnumerable<int> WirthSetRecursivo()
{
  for (int i=1; i<=int.MaxValue; i++)
    if (PerteneceWirth(i)) yield return i;
}
```

Listado 13-34 Producción con recursividad del conjunto de Wirth

El problema es que mientras mayor sea la cantidad de elementos del conjunto de Wirth que deseamos encontrar mayor es la cantidad de llamadas recursivas que se han realizado en vano al método `PerteneceWirth`. Este fenómeno es muy similar al que se analiza en el Capítulo 7 cuando se deseaba generar todos los números primos en un intervalo. En aquella ocasión hubo que acudir a la Criba de Eratóstenes, lo que

[2] Niklaus Wirth es el autor del conocido lenguaje de programación Pascal

representaba un cambio de paradigma en lugar de ir determinando cuáles eran primos se cambió por un método que iba "construyendo" aquellos que eran primos.

Para mejorar la solución de determinar los elementos del conjunto de Wirth vamos escribir una solución que los va **produciendo** en la medida en que se van **consumiendo**.

La dificultad de generar los números de Wirth de forma lineal radica en que existen dos formas distintas de obtener un número, ya sea multiplicando x por 2 o por 3 y sumarle 1. Por cada nuevo número obtenido se pueden entonces generar otros dos números que pertenecen al conjunto, de forma que la cantidad de números crece. Incluso, algunos números pueden ser generados por las dos vías y no deben estar repetidos en el conjunto. Por ejemplo, el número 31 es obtenido por la vía del $2x+1$ en la forma (1, 3, 7, 15, 31) y por la vía del $3x+1$ en la forma (1, 3, 10, 31). Por estos motivos no es posible diseñar un algoritmo lineal que produzca en orden creciente un número de Wirth cada vez, porque de cada número consumido se pueden producir dos.

Una solución a este problema consiste en emplear dos colas (Listado 13-35), una para los números que surgen como resultado de multiplicar por 2, y otra para los números que surgen como resultado de multiplicar por 3.

Cada vez que se quiera "consumir" un nuevo número, hay dos fuentes (colas) de donde escoger, de modo que se escoge el menor de los que están al frente en cada una de las colas y se quita de la cola. Si los que están al frente de ambas colas son iguales entonces se quita de ambas colas. Antes de retornar el valor `x` producido se pone el correspondiente `x*2+1` en la `cola2` y el `3*x+1` en la `cola3`.

```
static IEnumerable<long> WirthSetViaCola(long n)
{
  Queue<long> cola2 = new Queue<long>();
  Queue<long> cola3 = new Queue<long>();
  cola2.Enqueue(1); cola3.Enqueue(1);
  for (long k = 0; k < n; i++)
  {
    long x;
    if (cola2.Peek() < cola3.Peek())
      x = cola2.Dequeue();
    else if (cola3.Peek() < cola2.Peek())
      x = cola3.Dequeue();
    else //son iguales
    {
      x = cola2.Dequeue();
      cola3.Dequeue();
    }
    cola2.Enqueue(2 * x + 1);
    cola3.Enqueue(3 * x + 1);
    yield return x;
  }
}
```

Listado 13-35 Iterador para generar los números de Wirth empleando dos colas.

Esta solución es mucho más eficiente que la recursiva ya que empleando las colas solamente se generan y guardan los números que pertenecen al conjunto de Wirth (aunque algunos de ellos duplicados). Ejecutados en un procesador i5 con 4 GB de

RAM, la solución por la vía recursiva para producir los primeros 5 millones de elementos del conjunto de Wirth demoró 82,067 milisegundos mientras que la solución que usa las colas demoró solo 419 milisegundos. Invitamos al lector a probarlo en su hardware.

En el mismo hardware producir por la vía de colas los primeros 100 millones de elementos del conjunto de Wirth demoró unos 8405 milisegundos. Sin embargo, el intento de producir los primeros 200 millones provocó excepción por falta de memoria.

Esto se debe a que las colas han ido creciendo innecesariamente. Note que por cada elemento producido se ha colocado un valor en cada cola. Esto quiere decir que cuando se hayan producido *n* valores, quedarán distribuidos entre las dos colas otros *n* valores. Al producir los primeros 100 millones han quedado en la cola2 42,021,933 valores y en la cola3 quedaron 57,965,866 donde la suma de ambos es 99,987,799, que son aproximandamente los 100 millones si tenemos en cuenta los valores iguales que se obtienen por las dos vías y que son extraídos de ambas colas.

Este patrón de **productor consumidor**, donde una parte de la aplicación produce datos y los pone en una cola para que sean consumidos y otra parte de la aplicación los va quitando de la cola y procesando, es muy útil en la solución de problemas donde la velocidad a la que se generan los resultados no es igual a la velocidad a la que se consumen. Como en la práctica el hardware concreto hace que las colas no puedan crecer infinitamente, entonces hay que tener cuidado de que el productor no genere a una velocidad mucho mayor que la del consumidor porque se corre el riesgo de agotar la memoria.

13.5.5 IMPLEMENTACIÓN DE COLA CON NODOS ENLAZABLES

Una forma sencilla de implementar una cola es empleando el concepto de nodo enlazable. Si las operaciones de `Enqueue` y `Dequeue` se ven como casos particulares de `Insert` y `RemoveAt`, entonces es fácil notar la similitud con una lista enlazada en la que solamente se inserta al final y se quita por el principio. En la implementación de lista con enlazables presentada al inicio de este capítulo ambas operaciones pueden realizarse en tiempo constante.

Al igual que una lista enlazada, una cola enlazada debe mantener dos nodos que referencian respectivamente al primer (`first`) y al último (`last`) elemento. La operación de `Enqueue` es exactamente igual que la operación de inserción en listas en el caso particular en que se inserta en la última posición (Listado 13-36).

```
public class LinkedQueue<T>: IQueue<T>
{
  private LinkedNode<T> first;
  private LinkedNode<T> last;
  public int Count { get; private set; }

  public void Enqueue(T item)
  {
    if (Count == 0)
      first = last = new LinkedNode<T>(item, null);
    else
      last = last.Next = new LinkedNode<T>(item, null);
    Count += 1;
  }

  //... resto de la implementación
}
```

Listado 13-36. Implementación de cola con nodos enlazados.

Las operación `Peek` es equivalente a la de la lista enlazada y la operación `Dequeue` sería similar a hacer `RemoveAt(0)` en la lista enlazada (Listado 13-37).

```
public T Dequeue()
{
  if (Count == 0)
    throw new InvalidOperationException();
  T result = first.Value;

  if (Count == 1)
    first = last = null;
  else
    first = first.Next;

  Count--;
  return result;
}

public T Peek()
{
  if (Count == 0)
    throw new InvalidOperationException();

  return first.Value;
}
```

Listado 13-37 Implementación de los métodos Enqueue y Peek basados en nodos enlazables.

Se propone al lector implementar el enumerador para una cola basada en enlazables (recuerde el manejo de versión en cada operación que modifique el contenido de la estructura para evitar que se haga una modificación mientras se está recorriendo).

13.5.6 IMPLEMENTACIÓN DE COLA USANDO ARRAYS

Al igual que en el caso de las pilas, se puede hacer una implementación de cola basada en arrays. Una forma fácil de implementar una cola es usar internamente una lista

implementada por array (como es el caso del tipo `List` de la biblioteca de .NET) como se muestra en el Listado 13-38 .

Note que el método `Enqueue` lo que hace es invocar al método `Add` de lista interna, el método `Peek` retorna el primer elemento de la lista interna (`lista[0]`) y la propiedad `Count` sería la misma que la de la lista.

El método `Dequeue` se ha implementado como `lista.RemoveAt(0),` es decir quitar el primer elemento de la lista. Sin embargo, esta implementación es extremadamente ineficiente, porque el método `RemoveAt` en lista implementada con array provoca un desplazamiento de todos los elementos del array de la lista interna.

```
class Queue<T>: IQueue<T>
{
  List<T> lista = new List<T>();
  public int Count
  {
    get { return lista.Count; }
  }
  public void Enqueue(T x)
  {
    lista.Add(x);
  }
  public T Peek()
  {
    return lista[0];
  }
  public T Dequeue()
  {
    T primero = lista[0];
    lista.RemoveAt(0);
    return primero;
  }
}
```

Listado 13-38 Implementación de cola usando List

Se puede lograr una implementación de cola usando arrays obviando la ineficiencia anterior (Listado 13-39). Para ello se mantienen dos índices `first` y `last` que indican respectivamente la posición dentro del array del primer elemento de de la cola, y la posición donde se debe colocar el próximo `Enqueue`. En cada una de las operaciones se

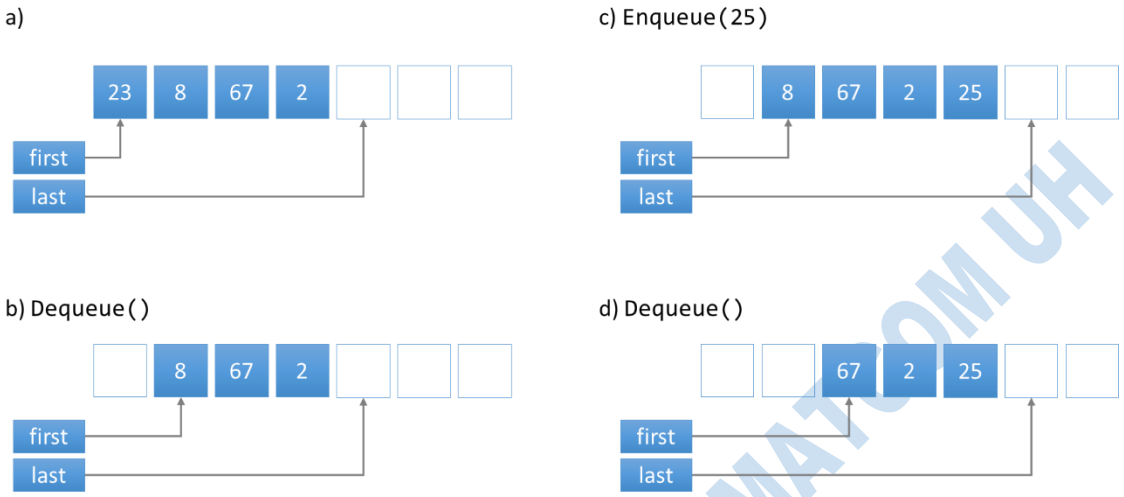


Figura 13.14 Ejecución sucesiva de Enqueue y Dequeue

mueve el índice correspondiente, por lo que no hay que hacer corrimientos y la implementación tiene un costo temporal constante (Error! Reference source not found.). Una vez el array se ha llenado, se emplea una estrategia similar a la usada anteriormente en la implementación de lista, para "hacer crecer" el espacio disponible.

A diferencia de la implementación del Listado 13-38, que usa internamente una lista, y que hace por tanto ineficiente la operación `Dequeue` al basarla en el desplazamiento provocado por el `RemoveAt(0)`, esta implementación mediante la variable `first` lleva directamente el control de cuál es la posición del array en la que está el elemento al frente de la cola y por tanto la extracción del elemento al frente no provoca ningún desplazamiento. Sin embargo, tiene la deficiencia de que se puede llegar al final del array haciendo `Enqueue` mientras quedan espacios libres al comienzo del array producto de previos `Dequeue` (Figura 13.15). En este caso esta implementación desencadena un crecimiento del array (método `Grow`) cuando realmente pudiera haber espacio disponible en el array.

Una implementación inteligente del método `Grow` (cuya implementación se deja propuesta al lector) pudiera aliviar esta dificultad, por ejemplo intentando primero "desplazar" todos los elementos hacia el inicio del array y solo al comprobar que aún no queda espacio realizar el verdadero crecimiento. Sin embargo, existe una alternativa mucho más eficiente, que aprovecha al máximo el espacio disponible en el array que se verá en el epígrafe siguiente.

```
public class ArrayedQueue<T> : IQueue<T>
{
  private T[] elements = new T[4];
  private int first;
  private int last;

  public int Count { get { return last - first; } }

  public void Enqueue(T item)
  {
    if (last == elements.Length)
      Grow();
    elements[last++] = item;
  }

  private void Grow()
  {
    //...
  }

  public T Dequeue()
  {
    if (Count == 0)
      throw new InvalidOperationException();

    return elements[first++];
  }

  public T Peek()
  {
    if (Count == 0)
      throw new InvalidOperationException();

    return elements[first];
  }
}
```

Listado 13-39 Implementación de cola usando arrays.

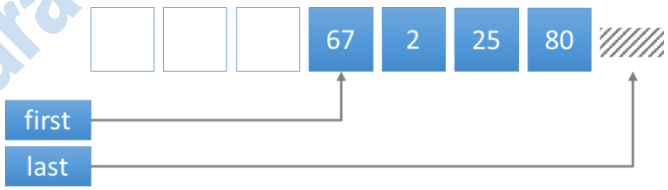


Figura 13.15. Espacios libres subutilizados en la implementación de cola con array.

13.5.7 COLA CIRCULAR

La implementación conocida como **cola circular** pretende aprovechar los espacios libres que van quedando al frente sin necesidad de desplazar los elementos (del mismo modo que los pacientes que esperan en un consultorio médico no se cambian de asiento cada vez que un paciente nuevo entra a la consulta).

La idea de la solución es tratar el array interno `elements` como si fuera "circular" (Figura 13.16). Esto significa que si se ha alcanzado la última posición del array y se quiere añadir un elemento a la cola se puede continuar poniendo el elemento por el comienzo del array si es que ya éste ha sido liberado (de ahí la analogía de lo circular). La cola tiene que llevar el control de la posición por la que está el primero y el último y saber "dar la vuelta" cuando se alcanza el final del array. Teniendo esto en cuenta las variables `first` y `last` indicarán la primera y última posición de la cola dentro del array, note entonces que no se tiene que cumplir necesariamente que `first <= last` tal y como se muestra en la Figura 13.16.

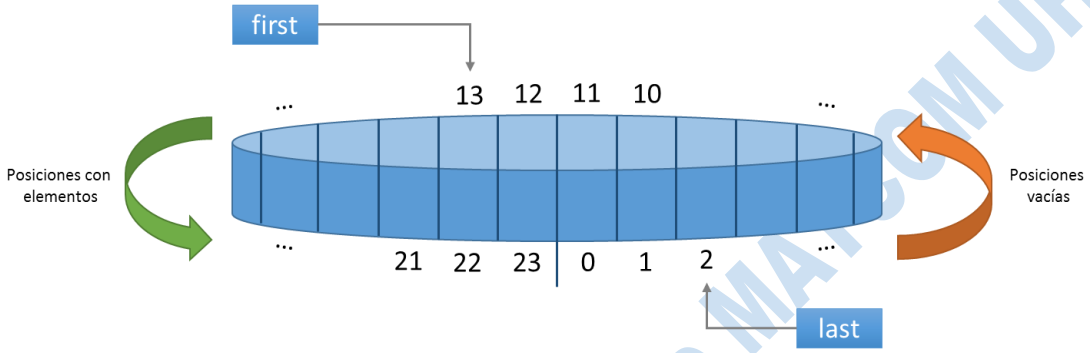


Figura 13.16. Ejemplo de una cola con array circular de longitud N=24.

La complejidad fundamental en esta implementación radica en cómo actualizar los índices para que den la vuelta de manera transparente. Para "dar la vuelta" se empleará el operador de **resto de la división** (%), de forma que cada índice, una vez que pase la longitud del array, vuelva a ser acotado dentro de los índices válidos del array. Teniendo esto en cuenta, la implementación de los métodos fundamentales es prácticamente idéntica, solo cambiando la forma en que se actualiza cada índice (Listado 13-40).

```
public class CircularQueue<T> : IQueue<T>
{
  private T[] elements = new T[4];
  private int first;
  private int last;

  public int Count { get; private set; }

  public void Enqueue(T item)
  {
    if (Count == elements.Length)
      Grow();

    elements[last] = item;
    last = (last + 1) % elements.Length;
    Count++;
  }

  private void Grow()
  {
    //...
  }
```

```
  public T Dequeue()
  {
    if (Count == 0)
      throw new InvalidOperationException();

    T result = elements[first];
    first = (first + 1) % elements.Length;
    Count--;
    return result;
  }

  public T Peek()
  {
    if (Count == 0)
      throw new InvalidOperationException();

    return elements[first];
  }
}
```

Listado 13-40. Implementación de una cola circular.

La implementación del método `Grow` (Listado 13-41) lleva algunas consideraciones adicionales, pues ahora la parte inicial de la cola puede estar al final del array y la parte final al inicio. Después de crear el array mayor (en este caso se duplica el tamaño del array) para determinar correctamente los elementos a copiar se utiliza también la lógica de la circularidad usando la operación de resto `%`. Se propone al lector mejorar esta implementación usando el método `Array.Copy`.

```
private void Grow()
{
  T[] temp = new T[elements.Length * 2];

  for (int i = 0; i < Count; i++)
    temp[i] = elements[(first + i)%elements.Length];

  elements = temp;
}
```

Listado 13-41 Implementación del método Grow para una cola circular.

13.6 CONJUNTOS Y FUNCIONES HASH

Los miembros `Add`, `Contains`, `Remove` y `Count` de la interface `ICollection` (Listado 13-1) nos dan la funcionalidad básica de conjuntos.

El tipo `List` implementa la interface `ICollection`, de modo que puede usarse como una implementación si nos interesan las funcionalidades anteriores. El tipo `List` implementa además las operaciones relacionadas con el acceso a los elementos a través de índices. Podemos presuponer que la implementación del método `Contains` en `List` es como se muestra en el Listado 13-42. Es decir, una búsqueda secuencial recorriendo el array.

```
public bool Contains(T x)
{
  for (int i = 0; i < Count; i++)
```

```
    if (Object.Equals(x, items[i]))
      return true;

  return false;
}
```

Listado 13-42 Implementación de Contains en una lista basada en arrays

Muchas aplicaciones que trabajan con conjuntos no están interesadas en el manejo explícito de índices sino en averiguar de manera eficiente si un determinado valor está en el conjunto (método `Contains`). Una implementación eficiente de `Contains` podría basarse en suponer que los elementos en el array interno del conjunto están ordenados y entonces hacer una búsqueda binaria (Ver Capítulos 6 y 7).

Sin embargo, este enfoque tiene varias limitaciones. En primer lugar el tipo de los elementos del conjunto (y por tanto del array) debe tener definida una función de orden (interface `IComparable<T>` o el conjunto debe tener un `IComparer<T>`).

Para mantener ordenados los elementos en el array habría que modificar las operaciones `Add` y `Remove`, que son las que hacen cambios en el array.

Todo dependería entonces de hacer una valoración de cuáles operaciones se realizarán con más frecuencia en la aplicación que va a trabajar con los conjuntos, si `Add` y `Remove` o si `Contains`. Pero esto necesariamente no hay que conocerlo por anticipado si se quiere dar una implementación lo más genérica posible. En el epígrafe siguiente se verá un enfoque diferente que no se basa en que los elementos del conjunto sean ordenables.

13.6.1 IMPLEMENTACIÓN EFICIENTE DE CONJUNTO USANDO UNA FUNCIÓN HASH

En este epígrafe se adoptará un enfoque diferente, que nos permitirá almacenar y buscar eficientemente cualquier tipo de dato, sin necesidad de una función de ordenación ni costosas operaciones de adición y eliminación.

Para ilustrar inicialmente la idea supongamos que se desea tener una implementación eficiente para representar conjuntos de valores enteros que están comprendidos en un intervalo (Min, Max). En este caso una solución muy eficiente consiste en emplear un array de tipo `bool` (Listado 13-43**Error! Reference source not found.Error! Reference source not found.**).

```
public class IntegerSet : ICollection<int>
{
  private readonly int min;
  private readonly int max;
  private readonly bool[] used;

  public int Count { get; private set; }

  public IntegerSet(int min, int max)
  {
    this.min = min;
    this.max = max;
    used = new bool[max - min + 1];
  }
```

```
  public void Add(int item)
  {
    if (item < min || item > max)
      throw new ArgumentOutOfRangeException();

    used[max - item] = true;
    Count += 1;
  }

  public void Remove(int item)
  {
    if (item < min || item > max)
      throw new ArgumentOutOfRangeException();

    used[max - item] = false;
    Count -= 1;
    return true;
  }

  public bool Contains(int item)
  {
    if (item < min || item > max)
      throw new ArgumentOutOfRangeException();
    return used[max - item];
  }
}
```

Listado 13-43 Implementación de un conjunto eficiente de enteros.

El enfoque consiste en tratar de llevar la solución anterior, que es válida para conjuntos de enteros, a conjuntos de cualquier tipo. Para ello la primera cuestión a resolver consiste en cómo convertir un tipo de dato cualquiera a un número entero acotado en un intervalo. Este es el rol que va a jugar lo que se denomina **función hash**[3].

13.6.1.1 FUNCIONES DE HASH

Una función **hash** es una función que recibe como entrada todos los posibles valores cierto tipo `T` y retorna un valor numérico entero que se denomina **código hash** (*hash code*) que se encuentra en un determinado rango.

Un ejemplo de función hash es el método mostrado en el Listado 13-44, que presenta una función de hash para el tipo `string`.

```
public static int Hash(string s)
{
    var v = 0;

    for (int i = 0; i < s.Length; i++)
        v += s[i];

    return v;
}
```

[3] Función de hash pudiera traducirse como función de desmenuzamiento, pero no se ha consensuado en español una traducción de hash para usar en este enfoque, por lo que se mantendrá el término original en inglés.

Listado 13-44 Implementación de una función de hash simple para el tipo string.

Esta función de hash cumple que a cada posible valor del dominio (`string`) le asigna un valor entero acotado (en este caso un entero positivo con valor entre `0` e `int.Maxvalue`). Sin embargo, esta función no cumple con alguna otras propiedades que son necesarias desde un punto de vista computacional.

A continuación las propiedades que debe cumplir una función hash:

- *Estabilidad*: Se dice que una función de hash es estable si produce la misma salida para la misma entrada. Evidentemente la función de hash anterior es estable, pues solo depende del valor tipo `string` de entrada. Sin embargo, en algunos dominios se emplean funciones de hash aleatorias, que no solo dependen de la entrada sino que además incluyen un factor de aleatoriedad.
- *Resistente a las colisiones*: Una colisión ocurre cuando dos valores diferentes del dominio tienen el mismo código hash. Es evidente que la función `Hash` del Listado 13-44 no cumple con esto, cualquier par de cadenas que tengan los mismos caracteres (`"hola"` y `"ohla"`) tendrán el mismo código hash. Una función de hash es más resistente a las colisiones en la medida en que lo anterior ocurra con menor frecuencia. Formalmente, una función es resistente a colisiones cuando no existe una manera fácil de encontrar, dada una entrada particular, otra entrada que colisione con ella. Esta es posiblemente la propiedad fundamental por la que se mide la calidad de una función de hash, ya que mientras menos colisiones puedan ocurrir se pueden lograr implementaciones más eficientes de las estructuras de datos que usen dicha función hash.
- *Uniformidad*: Se dice que una función de hash es uniforme si todos los valores del intervalo numérico de la imagen son igualmente probables de ocurrir. Cuando esta propiedad no se cumple, porque hay algunos valores de la imagen que son mucho más probables que otros, entonces aumenta la cantidad de colisiones. Esta propiedad tiene importancia para aplicaciones de seguridad, donde las funciones de hash se emplean para cifrar contraseñas, ya que una baja uniformidad permite descubrir más fácilmente la contraseña original a partir del código hash.
- *Eficiente*: El costo de generar un código hash debe ser adecuado para las necesidades de la aplicación. Por ejemplo, para usarla en la implementación de un conjunto (como se verá en el epígrafe 13.6.1.2) es deseable que el costo sea proporcional al tamaño de la entrada (en este caso del valor que se va a guardar en el conjunto), y que no dependa de cuántos valores ya se han almacenado en el conjunto.

13.6.1.2 UNA MEJOR IMPLEMENTACIÓN DE UNA FUNCIÓN DE HASH

El problema que tiene la función hash del Listado 13-44 anterior es que la posición en la que están los caracteres dentro de la cadena no participa en el valor que está calculando, de modo que cualquier permutación de los mismos caracteres dará el mismo resultado. En este sentido, la función no es resistente a colisiones, pues es muy fácil, dada una cadena particular, encontrar otras cadenas que tengan el mismo código hash. El Listado

13-45 nos da una implementación que sí toma en cuenta la posición de los caracteres en la cadena[4].

```
public static int FoldingHash(string s)
{
  s += new string('0', 4 - s.Length % 4);

  var v = 0;
  for (int i = 0; i < s.Length; i += 4)
  {
    var byte1 = (int)s[i + 0] << 24;
    var byte2 = (int)s[i + 1] << 16;
    var byte3 = (int)s[i + 2] << 8;
    var byte4 = (int)s[i + 3];
    v += byte1 | byte2 | byte3 | byte4;
  }
  return v;
}
```

Listado 13-45. Una función hash que considera la posición de cada carácter.

Note que esta función primero se asegura de que la cadena pasada como parámetro tenga una longitud múltiplo de `4`. Esto se logra con la primera línea del método concatenando los caracteres `'0'` necesarios al final.

Básicamente la función calcula el código hash tomando los valores ASCII de los caracteres de cuatro en cuatro. Debido a que los valores ASCII son valores entre `0` y `255` (`1` byte) entonces con cuatro caracteres se puede conformar un entero de `32` bits (`4` bytes). Los valores de cuatro caracteres consecutivos pueden ser concatenados al aplicar el operador `|` (or binario) luego de calcular la posición correcta de los bytes de cada carácter en los `32` bits mediante el operador `<<` (corrimiento a la izquierda) para formar el número de `32` bits. La Figura 13.17 muestra un ejemplo de cómo se hace esto.

Figura 13.17. Ejemplo de la aplicación de la función de hash mejorada.

Note que esta función sigue siendo *estable* y *eficiente*. La mejora con respecto a la anterior consiste en que al considerar la posición de cada carácter para calcular el valor final se gana en uniformidad y en resistencia a colisiones, pues no será frecuente que dos cadenas diferentes, aunque muy parecidas, den el mismo código hash. Pero no es muy seguro, aunque es un poco más complejo de romper, no es imposible explorar las distintas cadenas que pueden producir un mismo código hash.

[4] Estamos considerando cadenas con el alfabeto latino

13.6.1.3 FUNCIONES DE HASH EN .NET

Las funciones de hash aparecen a menudo en la implementación de estructuras de datos, algoritmos de criptografía, y otros escenarios. Tal es la utilidad de disponer de una función de este tipo en la biblioteca estándar de .NET, por lo que para ello la clase `object` tiene un método `GetHashCode` declarado como `virtual`. De esta forma, todo objeto en .NET tendrá predeterminadamente un método de hash y podrá dar su propia implementación personalizada si la por defecto no le conviene.

Aunque la implementación por defecto de `GetHashCode` es suficiente en muchos escenarios, en ocasiones es conveniente dar una implementación. En particular, cuando se define un nuevo tipo, es común redefinir el método `Equals` para brindar un criterio de igualdad adecuado a la semántica del tipo implementado. En este caso el compilador emite una advertencia, indicando que se debe redefinir también el método `GetHashCode`. Esto se debe a que, debido a la propiedad de estabilidad, se espera que si `x.GetHashCode() == y.GetHashCode()`, entonces tiene que cumplirse que `x.Equals(y) == true`. El cumplimiento de esta propiedad será fundamental para garantizar que funcionen correctamente las estructuras de datos basadas en funciones de hash.

Idealmente, debería también cumplirse el recíproco, es decir, que para tipos distintos se obtengan códigos hash distintos. Sin embargo, en muchas ocasiones esto no es posible. En particular, cuando el dominio es mucho más grande que la imagen, necesariamente habrán colisiones. Por ejemplo, es imposible definir una función de hash perfecta para el tipo `string`, ya que la cantidad de `string` distintos es MUCHO mayor (potencialmente infinita) que la cantidad de valores que caben un entero de 32 bits.

La implementación concreta del método `GetHashCode` provista predeterminadamente en .NET devuelve un valor que depende la posición en la memoria (en el heap) designada para el objeto cuando fue creado en caso de que sea un tipo por referencia. Para los tipos por valor (`struct`) lo que hace es calcular un hash que tiene en cuenta el valor de cada bit de la memoria que representa al tipo. Por este motivo, el hash por defecto para tipos por referencia generalmente es útil, excepto cuando convenga hacer una implementación particular de `Equals`.

Sin embargo, el hash por defecto de los tipos por valor no es tan útil, en especial porque para los tipos mutables este código hash cambia si cambia el valor de alguna de las variables de instancia, lo que lo hace inútil de usar en estructuras de datos que se apoyen en un valor hash. De modo que tenga en cuenta estas consideraciones al usar objetos de un tipo por valor, especialmente si son mutables.

Los tipos básicos (`int`, `double`, `string`, etc.) todos tienen una implementación particular optimizada de su `GetHashCode` por lo que son seguros y muy eficientes de usar en la práctica.

13.6.1.4 IMPLEMENTACIÓN DE CONJUNTOS USANDO TABLAS HASH

A partir del concepto de **función hash** es posible diseñar una estructura de datos para representar conjuntos con un costo prácticamente constante para averiguar la pertenencia de un valor a un conjunto (generalmente expresada por un método `Contains`). La idea es inspirarnos en la implementación de `IntegerSet` que se vio anteriormente; para ello, se transforma un elemento del dominio `T` en un valor entero,

empleando una función de hash (la definida por el método `GetHashCode` de la raíz `object` y que por tanto está disponible en todos los tipos de .NET).

El entero devuelto por `x.GetHashCode()` serviría entonces de índice de acceso directo a una tabla de elementos de tipo `T` (array) donde se guardaría el valor de `x` si `x` está en el conjunto o estaría vacía (`null`) si `x` no está.

Sin embargo, `GetHashCode` nos da un entero de 32 bits, por lo que no es viable usar directamente ese entero como índice de un array, ya que haría falta un array de 16GB de RAM! para representar cada conjunto. Por lo tanto hay que usar un array de un tamaño viable, lo que significa que la función hash debe darnos un entero acorde al tamaño del array destinado como **tabla hash**. Una solución simple sería almacenar el elemento en la posición `x.GetHashCode() % items.Length` (siendo `items` el array interno que se use para dicha tabla), de forma que se pueda llevar cualquier valor de hash a un valor de índice válido (Figura 13.18). Es precisamente esta implementación la que emplea la clase `HashSet<T>` de la biblioteca de .NET.

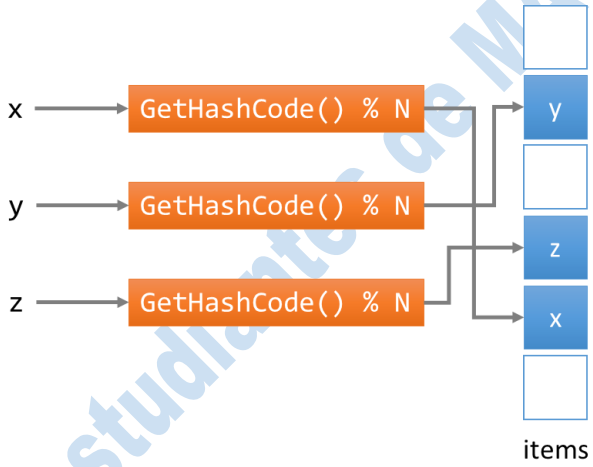


Figura 13.18 Ubicación de elementos de un tipo arbitrario en una tabla hash

Pero queda entonces un problema a resolver. ¿Qué hacer si dos valores distintos `x` e `y` colisionan en una misma entrada a la tabla? Es evidente que esto puede ocurrir, basta con que el dominio dado por el tipo `T` tenga más valores que el tamaño destinado al array-tabla. Por ejemplo si el dominio es el tipo `string`, ni aun dedicando una array de longitud `int.MaxValue` (lo que ya de por sí no es viable) bastaría para tener una entrada diferente para cada valor `string`; por lo que es imposible evitar que pueda haber colisiones. En los dos epígrafes siguientes se verán dos enfoques para solucionar este problema con lo que se denomina tabla hash **abierta** y tabla hash **cerrada**.

13.6.2 IMPLEMENTACIÓN DE UNA TABLA HASH ABIERTA

Una estrategia para solucionar el problema de las colisiones consiste en una estructura que se denomina **tabla hash abierta**. Esta estructura se basa en que en cada entrada de la tabla (elemento del array) lo que se almacena es una referencia al comienzo de una lista

enlazada de nodos en la que se almacenan todos los valores cuyo código hash colisiona en ese índice (Figura 13.19).

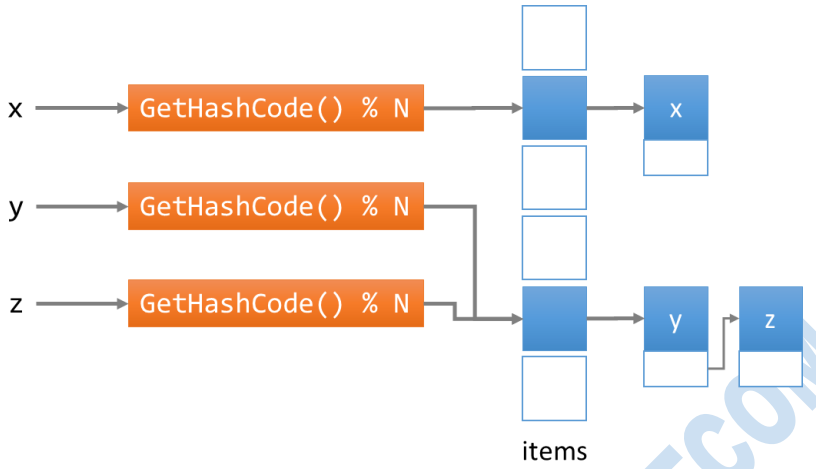


Figura 13.19 Representación de los elementos en una tabla hash abierta.

Para añadir, buscar o quitar un elemento de un conjunto, primero se computa el índice (usando la función de hash) de acceso a la tabla y luego se recorre secuencialmente la lista enlazada asociada a ese índice; buscando mediante el método `Equals` cuál es realmente el nodo que almacena el elemento a buscar (o a quitar), o para en caso de la operación de añadir (`Add`) para comprobar si el valor ya se encuentra en la tabla hash y en caso negativo añadirlo. El Listado 13-46 nos muestra una posible implementación que hemos denominado `OpenHashSet`.

```
public class OpenHashSet<T> : ISet<T>
{
  private LinkedNode<T>[] items = new LinkedNode<T>[16];
  private const int MaxSize = 5;

  public bool Add(T item)
  {
    int index = item.GetHashCode() % items.Length;
    int count = 0;
    for (LinkedNode<T> node = items[index]; node != null; node = node.Next)
    {
      count++;

      if (Equals(node.Value, item))
        return false;
    }
    items[index] = new LinkedNode<T>(item, items[index]);
    if (count >= MaxSize)
      Rehash();
    Count += 1;
    return true;
  }

  public bool Contains(T item)
  {
    int index = item.GetHashCode() % items.Length;
```

```
    for (LinkedNode<T> node = items[index]; node != null; node = node.Next)
      if (Equals(node.Value, item))
        return true;
    return false;
  }

  //...
  }
```

Listado 13-46 Implementación de los métodos Add y Contains de una tabla hash abierta.

Para la operación de quitar se sigue una estrategia similar (Listado 13-47) para encontrar el valor a quitar manteniendo la referencia al nodo anterior (variable `previous`).

```
public bool Remove(T item)
{
  int index = item.GetHashCode() % items.Length;
  LinkedNode<T> node = null;
  LinkedNode<T> previous = null;
  for (node = items[index]; node != null; previous = node, node = node.Next)
  {
    if (Equals(node.Value, item)) break;
  }
  if (node == null)
    return false;
  if (previous == null)
    items[index] = node.Next;
  else
    previous.Next = node.Next;
  Count -= 1;
  return true;
}
```

Listado 13-47 Eliminación en una tabla hash abierta.

Para que esta implementación sea eficiente, es conveniente el tamaño de cada lista de colisiones no sobrepase un determinado umbral. Si esto ocurre entonces se hace crecer el array interno. Realizar esta operación puede implicar que ahora al valor de hash le corresponda un índice distinto en la tabla (debido a la operación de `%`), por lo que habría que volver a insertar todos los valores (aplicando el método `Add`).

Esta operación, que se denomina `Rehash`, puede emplear diversas estrategias para hacer crecer el array. Una estrategia sencilla es duplicar el tamaño del array (Listado 13-48), aunque existen estrategias más elaboradas que serán discutidas más adelante.

```
private void Rehash()
{
  LinkedNode<T>[] temp = items;
  items = new LinkedNode<T>[items.Length * 2];

  for (int i = 0; i < temp.Length; i++)
  {
    LinkedNode<T> node = temp[i];

    while (node != null)
    {
      Add(node.Value);
```

```
            node = node.Next;
        }
    }
}
```

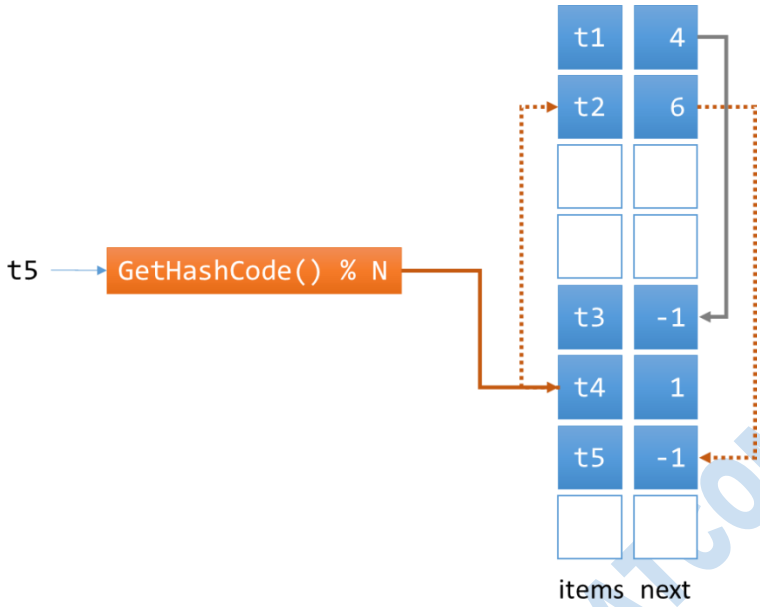
Listado 13-48 Operación de Rehash para hacer crecer el array en una tabla hash abierta.

Este enfoque de hash abierto es sencillo y efectivo para muchas aplicaciones. Sin embargo, puede tener la desventaja de que si una mala elección de la función de hash o del tamaño del array interno puede provocar que no haya una distribución uniforme de los valores en la tabla; por ejemplo que se alcance la cantidad máxima de colisiones permitidas para una entrada (índice) aún cuando la mayoría del array está vacío. Por otro lado si se incrementa el número máximo de colisiones tolerado para una entrada puede que se esté aprovechando mejor la memoria pero se está incrementando el costo computacional al dar la posibilidad de tener que recorrer listas de enlaces más largas.

13.6.3 IMPLEMENTACIÓN DE UNA TABLA HASH CERRADA

Otro enfoque para la solución de conflictos en una tabla hash es el que se conoce como **tabla hash cerrada**. En esta implementación todos los valores se guardarán en el propio array, pero como hay colisiones no necesariamente en la posición que indique su código hash. Cuando un elemento colisiona con otro que ya está en ese índice, se busca una posición libre en el propio array y se deja una referencia en el índice anterior a este nuevo índice, de modo de poder encontrarlo cuando en el futuro se vuelva a buscar ese valor.

De cierta forma es como si la lista enlazada empleada en la implementación abierta ahora estuviera distribuida por todo el array, en vez de linealmente organizada en cada índice. En la **Error! Reference source not found.** se muestra un ejemplo gráfico. Al buscar el valor `t5`, la función de hash indica la entrada 5, pero el elemento que realmente se encuentra allí es `t4`, cuyo código hash colisiona con el código hash de `t5`. La posición correspondiente en el segundo array (`next`), que en la figura tiene valor 1, indica en qué posición está el siguiente valor que cuyo hash colisiona con el actual (`t6` en la posición 1). Siguiendo esta secuencia de referencias se llega finalmente a `t5`.



Para implementar esto vamos a definir un tipo `HashEntry` para almacenar tanto el valor como la referencia al siguiente de la misma colisión (Listado 13-49).

```
public class HashEntry<T>
{
  public T Item;
  public int Next;
}
```

Listado 13-49 Tipo HashEntry que representa un elemento de la tabla hash cerrada.

La operación de adición inicialmente intenta ubicar al elemento en la posición que indica su código hash. En caso de estar ocupada esta posición, se recorren las referencias para determinar si el elemento deseado ya se encuentra en el array. Si finalmente se determina que el elemento no está en el array, entonces se mueve el elemento que ocupaba su posición a una nueva posición vacía arbitraria y se actualizan las referencias correspondientes (Listado 13-50). Conceptualmente esta estrategia es muy similar a la empleada en la tabla hash abierta, solo que los otrora nodos enlazados ahora son objetos de tipo `HashEntry` que se almacenan en el mismo array.

```
public bool Add(T item)
{
  int index = item.GetHashCode() % items.Length;

  for (var node = items[index]; node != null; node = items[node.Next])
    if (Equals(node.Item, item))
      return false;

  int location = FindEmptyIndex();
  items[location] = items[index];
  items[index] = new HashEntry<T> { Item = item, Next = location };
  if (Count >= items.Length * 0.9)
    Rehash();
  return true;
}
```

Listado 13-50 Implementación del método Add en una tabla hash cerrada

La operación de eliminación también es similar a su homóloga en la tabla hash abierta. Sin embargo, en este caso es conveniente manejar directamente los índices asociados a los nodos en vez de los nodos propiamente dichos, de forma que todas las referencias puedan ser actualizadas cómodamente (Listado 13-51).

```
public bool Remove(T item)
{
  int index = item.GetHashCode() % items.Length;
  int previous = -1;
  int i;

  for (i = index; items[i] != null; previous = i, i = items[i].Next)
    if (Equals(items[i].Item, item))
      break;

  if (items[i] == null) return false;
  if (previous < 0)
  {
    items[index] = items[items[i].Next];
    items[items[i].Next] = null;
  }
  else
  {
    items[previous].Next = items[i].Next;
  }
  items[i] = null;
  Count -= 1;
  return true;
}
```

Listado 13-51 Eliminación en una tabla hash cerrada

Para implementar el método `FindEmptyIndex` empleado en la operación `Add`, una variante simple consiste en iterar por todo el array buscando la primera posición con valor `null`.

Implementaciones más avanzadas emplean una lista adicional para ir almacenando las posiciones que se van vaciando en cada llamada a Remove, de forma que a la hora de insertar un nuevo elemento exista una alta probabilidad de encontrar rápidamente una posición vacía, pero esta optimización se sale del alcance de este libro.

En una implementación de tabla hash cerrada generalmente se efectúa la operación de Rehash una vez que la cantidad de elementos es mayor que un cierto porciento. Para lograr una distribución más uniforme de las colisiones es conveniente que la longitud del array sea un número primo, y esto debe tenerse en cuenta a la hora de hacer crecer el array. Para ello es útil almacenar previamente un array (inicializado estáticamente) con los números primos que indiquen el tamaño de los crecimientos que deben hacerse del array.

La implementación de la clase `HashSet<T>` presente en la biblioteca estándar de .NET emplea una representación basada en una tabla hash cerrada. En esta implementación existen optimizaciones adicionales, tales como las descritas anteriormente, para garantizar un rendimiento óptimo.

13.7 DICCIONARIOS

Como todas las de este capítulo, el **diccionario** es también una estructura de datos muy útil en la Ciencia de la Computación. Aparece en todas las situaciones donde se manejan dos tipos de información: una información (**llave**) por la que se conoce, se maneja y nos resulta más fácil buscar y otra información asociada a la anterior (**valor**) que es el significado. Como en una agenda cuando se busca por un nombre para obtener un número de teléfono.

El diccionario sirve para representar una colección de pares de valores, uno por el que se busca y otro que da un significado; por ejemplo como un diccionario de traducción de un idioma a otro que representa una colección de pares de cadenas (digamos que el primer elemento del par es una palabra en español y el segundo elemento del par es una cadena con todas sus posibles traducciones al inglés).

Formalmente, un tipo diccionario representa una función entre un dominio (cuyos valores son denominados **llaves**, y generalmente representado con un tipo genérico denotado por `TKey`) y una imagen o valor asociado a la llave (denominado **valor**, y representado generalmente con el tipo genérico denotado por `TValue`).

El tipo interface que en .NET caracteriza a un diccionario se muestra en el Listado 13-52. Note que esta definición emplea dos parámetros genéricos distintos, de forma que se puede tener un diccionario de `string` a `int`, o de cualquier otro par de tipos. Esto no excluye por supuesto la posibilidad de tener un diccionario donde llave y valor sean del mismo tipo (por ejemplo de `string` a `string`).

```
public interface IDictionary<TKey, TValue> :
  ICollection<KeyValuePair<TKey,TValue>>,IEnumerable<KeyValuePair<TKey, TValue>>
{
  bool ContainsKey(TKey key);
  void Add(TKey key, TValue value);
  bool Remove(TKey key);
  bool TryGetValue(TKey key, out TValue value);
  TValue this[TKey key] { get; set; }
  ICollection<TKey> Keys { get; }
  ICollection<TValue> Values { get; }
}
```

Listado 13-52 Definición del tipo Diccionario en .NET.

El tipo `KeyValuePair<TKey, TValue>` sirve para representar la pareja llave-valor en un solo objeto que pueda ser almacenado, manipulado y pasado como parámetro de forma conveniente (Listado 13-53). Note que por eficiencia se define como `struct` y no como `class` para tratarlo por valor.

```
public struct KeyValuePair<TKey, TValue>
{
    public TKey Key { get; }
    public TValue Value { get; }
}
```

Listado 13-53 Tipo KeyValuePair.

Como la inteface `IDictionary` implementa a su vez `ICollection<KeyValuePair<TKey, TValue>>`, además de los métodos definidos en la interface `IDictionary` hay que implementar los métodos asociados a `ICollection`. Sin embargo, ahora estos métodos heredados de `ICollection` tienen parámetros de tipo `KeyValuePair<TKey, Value>`. Por tanto, para adicionar un nuevo elemento, empleando el método `Add` heredado de `ICollection`, es necesario escribir `dic.Add(new KeyValuePair<string,string>("hola", "hello")`, lo cual evidentemente es bastante más engorroso. Por este motivo, en `IDictionary` se añade una variante método `Add` que recibe directamente un `TKey` y un `TValue`.

Otro caso similar ocurre con el método `Remove`, al que empleando la definición de `ICollection`, es necesario pasarle una instancia `KeyValuePair` con la llave **y** el valor a eliminar. En la práctica, usualmente se conoce la llave que se desea eliminar, pero no es fácil saber el valor asociado a ésta sin consultar el diccionario (sino para qué quería el diccionario en primer lugar). Además, es necesario definir qué sucede si la llave está, pero no está asociada al valor indicado. Por todos estos motivos, se adiciona un método `Remove` que solo recibe la llave.

La operación más novedosa en un diccionario es el indizador, que sirve a la vez para obtener el valor asociado una llave (en caso de existir), para cambiarlo o para crear una nueva pareja de llave-valor en caso de no existir.

Además, es posible obtener directamente una colección que contiene todas las llaves (propiedad `Keys`), y otra que contiene todos los valores (propiedad `Values`). Ambas colecciones son de solo lectura, es decir, no permiten adicionar ni eliminar directamente un elemento, pues de lo contrario el diccionario quedaría en estado inconsistente).

13.7.1 EJEMPLO DE USO DE UN DICCIONARIO

Los diccionarios se usan en aplicaciones que gestionan agendas, calendarios (de uso tan común en muchas aplicaciones). El propio compilador que usted usa, cuando está aprendiendo programación con este libro, emplea un diccionario para asociar los nombres que usted usa en su código con un significado (una variable, un método, un tipo, etc).

A modo de ejemplo, se presenta a continuación la solución de un problema típico empleando la clase `Dictionary<TKey,TValue>` presente en la biblioteca estándar de .NET. Se tiene acceso una secuencia de información de un cierto tipo `T` (un `IEnumerable<T>` para mayor abstracción) y se quiere organizar agrupadamente esa información para accederla de modo más simple y efectivo según las propiedades de un valor de tipo `T`.

Por ejemplo tenemos una secuencia de datos de estudiantes, representados por el tipo `Estudiante` (Listado 13-54), y queremos agruparlos, y accederlos con facilidad, según el grupo de clase al que pertenecen.

Para ello se quiere implementar un método `Agrupar` que recibe una secuencia de instancias de tipo `Estudiante` (Listado 13-54) y debe devolver una secuencia de grupos de estudiantes, de modo que en cada una de las secuencias más internas todas las instancias tienen el mismo valor en la propiedad `Grupo`.

```
public class Estudiante
```

```
{
  public string Nombre { get; private set; }
  public string Grupo { get; private set; }
  public int Edad { get; private set; }

  //...
}
```

Listado 13-54 Clase Estudiante.

En ests caso se empleará un diccionario cuyas llaves son los nombres de grupos, y el valor asociado a cada llave será una colección (lista) con todos los estudiantes que pertenezcan a dicho grupo. La definición de grupo la de el Listado 13-55.

```
public class Grupo
{
  public string Nombre { get; set; }
  public List<Estudiante> Estudiantes { get; set; }
}
```

Listado 13-55 Definición del tipo Grupo

El código del Listado 13-56 recorre la secuencia de estudiantes y los va guardando en un diccionario donde la llave es el nombre del grupo y el valor es una lista con los estudiantes que tienen ese grupo como valor de la propiedad `Grupo` de cada estudiantes.

```
static List<Grupo> Agrupar(IEnumerable<Estudiante> estudiantes)
{
  Dictionary<string, List<Estudiante>> dicc =
    new Dictionary<string, List<Estudiante>>();
  foreach (Estudiante e in estudiantes)
  {
    if (!dicc.ContainsKey(e.Grupo))
      dicc.Add(e.Grupo, new List<Estudiante>());

    dicc[e.Grupo].Add(e);
  }
  List<Grupo> grupos = new List<Grupo>();
  foreach (KeyValuePair<string, List<Estudiante>> par in dicc)
    grupos.Add(new Grupo { Grupo = par.Key, Estudiantes = par.Value });

  return grupos;
}
```

Listado 13-56 Implementación del método Agrupar usando diccionarios.

13.7.2 DICCIONARIOS Y PILAS. EVALUANDO EXPRESIONES ARITMÉTICAS

En este epígrafe veremos un ejemplo que integra de forma interesante el empleo de pilas y diccionarios.

Consideremos el problema de evaluar expresiones aritméticas en notación postfija. A diferencia de la notación que usualmente se emplea en matemática, y que es la que por lo general asumen la mayoría de los lenguajes de programación, en que los operadores

aparecen entre los operandos, en la notación postfija primero aparecen los operandos y continuación los operadores.

Así por ejemplo una expresión como (x+1) * (y-2) se escribiría en notación postfija en la forma x 1 + y 2 - *. Una forma sencilla de evaluar una expresión en notación postfija es recorrer la secuencia de izquierda a derecha y cada vez que se encuentre un operador se aplica dicha operación a los operandos (u operando si el operador es unario) anteriores sustituyendo a éstos por el resultado de la operación.

Para evaluar la expresión postfija 3 2 + 6 4 – *, que correspondería a la expresión (3 + 2)* (6 – 4) se aplican los pasos a continuación:

- **3 2 +** 6 4 – * Se efectúa la operación + entre 3 y 2.
- 5 **6 4 -** * Se efectúa la operación - entre 6 y 4.
- **5 2 *** Se efectúa la operación * entre 5 y 2.
- 10 Se terminó la secuencia, el resultado final es 10

Observe cómo en esta notación no son necesarios los paréntesis para forzar el orden en que se quieren aplicar las operaciones. Con un solo recorrido por la expresión es posible calcular el valor de la misma, sin necesidad de analizar el operador más externo o considerar reglas de precedencia entre los operadores. Para implementar computacionalmente este algoritmo es conveniente emplear una pila, que servirá para ir poniendo los operandos y los resultados de las operaciones que a su vez se convierten en operandos.

El Listado 13-57 muestra una posible implementación. Para simplificar se considera por ahora como entrada al método un array de valores `string`. Cada cadena o es un número (que consideramos puede ser de tipo `double`) o es el símbolo de una operación de suma, resta, multiplicación o división. La cadena se recorre de izquierda a derecha. Cada vez que se encuentra una cadena que corresponda a un número se guarda el valor correspondiente en la pila. Cuando aparece un operador, simplemente se extraen de la pila los dos últimos números, se hace la operación correspondiente sobre ellos, y se empila el resultado.

Al terminar de recorrer toda la secuencia debe haber quedado un único valor en la pila que será el resultado de evaluar la expresión.

Por simplicidad se ha asumido que la secuencia original ya viene separada en los elementos que la componen (en este caso números y operadores). En realidad en un lenguaje de programación la entrada original es una gran cadena de texto con el código de todo el programa. Este proceso inicial (denominado tokenizacion o análisis léxico) consiste en dividir la cadena en tokens.

```
static double Evaluar(string[] expresion)
{
  Stack<double> s = new Stack<double>();

  foreach (var token in expresion)
  {
    double d;
    if (double.TryParse(token, out d))
    {
      s.Push(d);
    }
    else
    {
      double opndo2 = s.Pop();
      double opndo1 = s.Pop();

      switch (token)
      {
        case "+":
          s.Push(opndo1 + opndo2);
          break;
        case "-":
          s.Push(opndo1 - opndo2);
          break;
        case "*":
          s.Push(opndo1 * opndo2);
          break;
        case "/":
          s.Push(opndo1 / opndo2);
          break;
      }
    }
  }

  if (s.Count != 1)
    throw new ArgumentException();

  return s.Pop();
}
```

Listado 13-57 Evaluación de una expresión en notación postfija

Precisamente parte del trabajo que hace un compilador es convertir las expresiones presentes en el texto de entrada (posiblemente en notación infija) a una secuencia de elementos en notación postfija que luego en ejecución será evaluada de forma muy eficiente y en un recorrido lineal usando una pila como se muestra en el Listado 13-57.

13.7.2.1 INCLUYENDO VARIABLES EN LAS EXPRESIONES

Para dar algo más de realismo al ejemplo anterior vamos a considerar que en la notación postfija puede haber nombres de variables.

En este caso la notación postfija correspondiente al código

```
c = (a+1) * (b-2)
```

podría ser algo como

```
a 1 + b 2 - * = c
```

lo cual evidentemente es mucho menos legible que la notación infija si lo tuviéramos que escribir manualmente; pero más fácil de evaluar usando una pila si es un compilador el que convierte de notación infija a notación postfija.

Si el valor de `a` es 4 y el valor de `b` es 3, entonces el resultado de la evaluación de esta secuencia sería asignar 8 a la variable `c`

Note que ahora en la cadena postfija a evaluar una expresión no solo aparecen números y operaciones sino también nombres de variables que pueden tener un valor asignado sobre el cual se realizarán las operaciones.

En esta notación se ha considerado que cuando aparece una operación de asignación "=" el valor a asignar está a la izquierda (en el tope de la pila cuando se esté evaluando) y la variable donde se asignará debe ser el token que sigue al operador.

Para representar la "memoria" de estas variables se utilizará precisamente un diccionario (de nombre `variables` en el código del Listado 13-58) para a partir de un valor `string` llave (el nombre de la variable) poder tener acceso al valor de dicha variable (el valor asociado a dicha llave).

El código del Listado 13-58 nos expresa ahora la implementación del método `EvaluarAsignación` para evaluar una secuencia de cadenas que corresponden a los tokens de una instrucción de asignación. Se sobreentiende que un nombre de variable que aparezca en la secuencia debe ser un nombre de variable a la que anteriormente se le haya asignado un valor por eso al no estar en el diccionario se lanza la excepción por estar usando una variable a la que no se le ha asignado ningún valor.

```
static Dictionary<string, double> variables = new Dictionary<string, double>();

static void EvaluarAsignacion(string[] tokens)
{
  Stack<double> s = new Stack<double>();

  for (int k=0; k<tokens.Length; k++)
  {
    string token = tokens[k];
    double d;
    if (double.TryParse(tokens[k], out d))
    {
      //es un numero
      s.Push(d);
    }
    else if (char.IsLetter(token[0]))
    {
      //es un nombre de variable
      if (!variables.Keys.Contains(tokens[k]))
        throw new Exception("Variable sin valor");
      else //la cadena es un nombre de variable
        s.Push(variables[token]);
    }
    else if (token == "=")
```

```
    {
      //es el operador de asignacion
      //a continuación debe venir la variable
      variables[tokens[k++]] = s.Pop();
    }
    else
    { //es operador aritmético
      double opndo2 = s.Pop();
      double opndo1 = s.Pop();

      switch (token)
      {
        case "+":
          s.Push(opndo1 + opndo2);
          break;
        case "-":
          s.Push(opndo1 - opndo2);
          break;
        case "*":
          s.Push(opndo1 * opndo2);
          break;
        case "/":
          s.Push(opndo1 / opndo2);
          break;
      }
    }
  }
  if (s.Count != 1)
    //Si la cadena postfija estuvo bien generada esto no debe ocurrir
    throw new ArgumentException();
}
```

Listado 13-58 Evaluando expresiones aritméticas con variables

Este código presupone que la secuencia "postfija" a evaluar es correcta, es decir que los errores sintácticos fueron detectados en una fase previa cuando se convirtió el texto de entrada a notación postfija.

Por ello para identificar a un posible nombre de variable basta con verificar que su primera letra sea un carácter alfabético.

Complique el ejemplo incluyendo variables cuyo valor pueda ser un array. Modifique el evaluador para tener en cuenta este nuevo tipo de operando.

Recrear aquí un ejemplo totalmente realista sería prácticamente construir un compilador y su intérprete (ejecutor) del código que este genere.

Incluso el compilador de C# detecta la ocurrencia en el código de variables no inicializadas si en lugar en que se intente usar el valor de una variable hay algún camino posible para llegar ahí sin que dicha variable aparezca en la parte izquierda de una asignación.

En un lenguaje como C# los diccionarios se utilizan en tiempo de compilación. El compilador "traduce" los nombres de variables a las posiciones (relativas) que en memoria estás ocuparán en ejecución, de modo que para acceder a dichos valores no haya que buscar en un diccionario.

En lenguajes dinámicos como Python se trabaja con un código más interpretativo y la estructura de memoria de sus variables en ejecución se basa en el uso de diccionarios.

Lo importante es que el lector haya podido apreciar, aunque sea de forma simplificada, la importancia que el uso de pilas y diccionarios tienen en el proceso de análisis y ejecución de un lenguaje.

13.7.3 IMPLEMENTACIÓN DE DICCIONARIOS UTILIZANDO LISTAS

Una implementación sencilla de la estructura diccionario consiste en una lista de pares llave-valor. Las operaciones de inserción, búsqueda y eliminación se delegan en las operaciones correspondientes de la lista interna (Listado 13-59).

```
public class ListDictionary<TKey, TValue> : IDictionary<TKey, TValue>
{
  private List<KeyValuePair<TKey, TValue>> elements;

  public ListDictionary()
  {
    elements = new List<KeyValuePair<TKey, TValue>>();
  }

  public void Add(TKey key, TValue value)
  {
    if (ContainsKey(key))
      throw new InvalidOperationException();

    elements.Add(new KeyValuePair<TKey, TValue>(key, value));
  }

  public bool ContainsKey(TKey key)
  {
    foreach (var kvp in elements)
      if (Equals(kvp.Key, key))
        return true;

    return false;
  }

  public bool Remove(TKey key)
  {
    for (int i = 0; i < elements.Count; i++)
      if (elements[i].Key.Equals(key))
      {
        elements.RemoveAt(i);
        return true;
      }

    return false;
  }
  //...
}
```

Listado 13-59. Implementación de diccionario utilizando listas.

La implementación del indizador es ligeramente más complicada, pues tiene que lidiar con los casos en los que hay que sustituir el valor asociado a una llave existente, y los casos en los que tiene que adicionar una nueva llave (Listado 13-60).

```
public TValue this[TKey key]
{
  get
  {
    foreach (var kvp in elements)
      if (Equals(kvp.Key, key))
        return kvp.Value;

    throw new KeyNotFoundException();
  }
```

```
    set
    {
      for (int i = 0; i < elements.Count; i++)
        if (Equals(elements[i].Key, key))
        {
          elements[i] = new KeyValuePair<TKey, TValue>(key, value);
          return;
        }

      elements.Add(new KeyValuePair<TKey, TValue>(key, value));
    }
  }
```

Listado 13-60 Implementación del indizador en un diccionario.

Obviamente el lector se habrá dado cuenta de que esta implementación tiene la debilidad de que para determinar el valor asociado a una llave se recorre secuencialmente toda la lista de pares *llave-valor* hasta el final o hasta dar con la llave[5]. Aún con esa implementación es útil trabajar con diccionarios por su expresividad como estructura de datos y por la abstracción que nos representa. Pero el mayor atractivo del concepto de diccionario es brindar una implementación en la que sea eficiente encontrar la llave y de este modo acceder rápidamente al valor asociado a ésta. Tenga en cuenta que en la mayoría de las aplicaciones el par *llave-valor* se guarda una vez pero luego se pregunta por él muchas veces (por ejemplo un variable en el código de un programa se declara en un único lugar y luego el compilador tiene que verificar si según la declaración está siendo utilizada correctamente en muchas partes del código).

13.7.4 IMPLEMENTACIÓN DE DICCIONARIO CON TABLA HASH

Existen muchas variantes de implementación eficientes para diccionarios. Algunas de las más usadas son basadas en árboles (los árboles se estudian en Capítulo 14. Otra variante de implementación, que de hecho es la utilizada en la biblioteca estándar de .NET es emplear una tabla hash internamente para almacenar los pares *llave-valor*.

La implementación es muy parecida a la implementación de tabla hash vista en el epígrafe 4513.6.2.

A continuación se verá una implementación empleando una tabla hash abierta. En cada elemento del array interno se almacenará una instancia de tipo `KeyValuePair<TKey,TValue>`. La búsqueda y la gestión de las posibles colisiones se harán con el valor de la llave aunque en el array se guarda el par completo. El método `Add` queda como se muestra en el Listado 13-61).

```
public class OpenHashDict<TKey, TValue> : IDictionary<TKey, TValue>
{
  public int Count { get; private set; }

  private LinkedNode<KeyValuePair<TKey, TValue>>[] items =
    new LinkedNode<KeyValuePair<TKey, TValue>

  private const int MaxSize = 5;
```

[5] Imagine que para buscar la traducción de una palabra tenga que recorrer secuencialmente todas las palabras del diccionario hasta encontrar la que busca.

```
  public void Add(TKey key, TValue value)
  {
    int index = key.GetHashCode()%items.Length;
    int count = 0;

    for (var node = items[index]; node != null; node = node.Next)
    {
      count++;
      if (Equals(node.Value.Key, key))
        throw new InvalidOperationException();
    }

    items[index] = new LinkedNode<KeyValuePair<TKey, TValue>>(
      new KeyValuePair<TKey, TValue>(key, value), items[index]);
    if (count >= MaxSize)
      Rehash();
    Count += 1;
  }

  //...
}
```

Listado 13-61 Implementación del método Add en un diccionario basado en hash abierto.

Por su parte, el método `Remove` es muy similar a su homólogo en la implementación de la tabla hash abierta, solamente modificado para manejar la llave (Listado 13-62).

```
public bool Remove(TKey key)
{
  int index = key.GetHashCode() % items.Length;

  LinkedNode<KeyValuePair<TKey,TValue>> node = null;
  LinkedNode<KeyValuePair<TKey,TValue>> previous = null;

  for (node = items[index]; node != null; previous = node, node = node.Next)
    if (Equals(node.Value.Key, key))
      break;

  if (node == null)
    return false;

  if (previous == null)
    items[index] = node.Next;
  else
    previous.Next = node.Next;

  Count -= 1;
  return true;
}
```

Listado 13-62 Implementación del método Remove para diccionario basado en hash abierto

Para la implementación de los métodos de búsqueda (`ContainsKey`, `TryGeyValue`) y el indizador, será conveniente implementar un método auxiliar `FindNode` (Listado 13-63)

que permita encontrar el nodo asociado a una llave determinada. Este método es muy similar al método `Contains` en la implementación de tabla hash abierta, excepto que devuelve `null` en caso de no encontrar el nodo correspondiente.

```
private LinkedNode<KeyValuePair<TKey, TValue>> FindNode(TKey key)
{
  int index = key.GetHashCode() % items.Length;

  for (var node = items[index]; node != null; node = node.Next)
    if (Equals(node.Value.Key, key))
      return node;

  return null;
}
```

Listado 13-63 Método auxiliar para buscar el nodo asociado a una llave.

Empleando este método auxiliar, las implementaciones de los métodos restantes son considerablemente más sencillas (Listado 13-64).

```
public bool ContainsKey(TKey key)
{
  return FindNode(key) != null;
}

public bool TryGetValue(TKey key, out TValue value)
{
  value = default(TValue);
  var node = FindNode(key);

  if (node == null)
    return false;

  value = node.Value.Value;
  return true;
}

public TValue this[TKey key]
{
  get
  {
    var node = FindNode(key);
    if (node == null)
      throw new KeyNotFoundException();
    return node.Value.Value;
  }
  set
  {
    var node = FindNode(key);
    if (node == null)
      Add(key, value);
    else
      node.Value = new KeyValuePair<TKey, TValue>(key, value);
  }
}
```

Listado 13-64 Implementación de los métodos de búsqueda para diccionario basado en hash abierto.

Por su parte, la implementación de un diccionario basado en una tabla hash cerrada es muy similar a la implementación para conjuntos que se vio en el epígrafe 13.6.3. Al igual que en el ejemplo anterior, es necesario modificar ligeramente los métodos de adición y eliminación para manejar directamente los conceptos de llave y valor. La implementación se le deja al lector a modo de entrenamiento.

La clase `Dictionary<TKey, TValue>` presente en la biblioteca estándar de .NET está implementada basada en una tabla hash cerrada. Incluso la implementación de diccionario puede ser personalizada si el programador aporta su propio método para calcular el código hash code. Profundizar en los detalles de esta implementación se sale del alcance de este libro pero es importante que el lector lo pueda hacer por su cuenta.

13.8 APOSTILLA

Las estructuras de datos son un tema tan tratado como la programación misma. Los arrays, records (en su versión moderna objetos), pilas, listas, colas, colecciones, diccionarios y árboles están presente en la solución de la generalidad de los problemas. Los cursos de estructuras de datos forman parte de cualquier currículum de computación y no tiene sentido hablar de programación sin hablar de estructuras de datos.

Aunque todos los conceptos abordados aquí son comunes a la mayoría de los lenguajes y plataformas de desarrollo, no se presentan por igual, con la misma sintaxis y funcionalidad en cada uno de estos. De modo que no se alarme si la forma en que se utilizan las estructuras de datos en las distintas aplicaciones no siempre sea automáticamente portable de un lenguaje a otro. Aunque los nombres de tipos, métodos y propiedades que aquí se han utilizado en los ejemplos de código son los de C# y la biblioteca de .NET, el lector de seguro encontrará sus análogos en otros lenguajes y esperamos pueda adaptarse a ellos sin dificultad cuando le sea necesario.

Con la evolución de los lenguajes de programación la utilidad de muchas de estas estructuras de datos han quedado subsumidas en la propia implementación y en el funcionamiento interno de recursos más abstractos de los propios lenguajes, como los records de Pascal y C manifestados en el concepto de variables de instancia y propiedades de un objeto, como las pilas en la recursividad (Capítulo 8), las colas en los recursos de sincronización (Capítulo 17) y los diccionarios en los recursos para el tipado dinámico (Capítulo 15). Es por ello que tal vez no siempre el lector tenga la necesidad directa de utilizar alguna de estas estructuras en sus aplicaciones; pero de todos modos es muy importante que comprenda su funcionamiento.

Hay infinidad de libros y documentación sobre este tema. Por supuesto que este capítulo y este libro no pretenden haberlo contenido todo, solo se han introducido las nociones más básicas para fundamentar y completar su capacidad de programación y permitirle explorar por su cuenta la gran diversidad de bibliotecas con implementaciones concretas que hay en el ecosistema del software.