Capítulo 36

Listas, pilas y colas en Java

Contenido

- Tipo abstracto de datos (TAD) `lista`
- Clase `Lista`
- Lista ordenada
- Lista doblemente enlazada
- Lista circular
- Tipo abstracto de datos `pila`
- Pila dinámica implementada con `vector`
- Pila implementada con una lista enlazada
- Tipo abstracto de datos `cola`
- Cola implementada con una lista enlazada
- Resumen
- Ejercicios
- Problemas

Introducción

La estructura de datos que se estudia en este capítulo es la lista enlazada (ligada o encadenada, "linked list"), que es una colección de elementos (denominados nodos) dispuestos uno a continuación de otro, cada uno de ellos conectado al siguiente elemento por un "enlace" o "referencia". En el capítulo se desarrollan métodos para insertar, buscar y borrar elementos en las listas enlazadas. De igual modo se muestra el tipo abstracto de datos (TAD) que representa a las listas enlazadas.

Las pilas, como tipo abstracto de datos, son objeto de estudio en este capítulo; son estructuras de datos **LIFO** (*last-in*/*first-out*, último en entrar/primero en salir). Las pilas se utilizan en compiladores, sistemas operativos y programas de aplicaciones.

También en este capítulo se estudia el tipo abstracto cola. Las colas se conocen como estructuras **FIFO** (*first-in*/*first-out*, primero en entrar/primero en salir), debido al orden de inserción y de extrac-

ción de elementos de la cola. Las colas tienen numerosas aplicaciones en el mundo de la computación: colas de mensajes, colas de tareas a realizar por una impresora, colas de prioridades.

Conceptos clave

- Arreglo *circular*
- Estructura de datos LIFO
- Estructura FIFO
- Lista circular
- Lista doble
- Lista enlazada
- Nodos

Tipo abstracto de datos (TAD) `lista`

Una *lista enlazada* es una colección o secuencia de elementos dispuestos uno detrás de otro, en la que cada elemento se conecta al siguiente por un "enlace" o "referencia". La idea básica consiste en construir una lista cuyos elementos llamados *nodos* se componen de dos partes (*campos*): la primera parte contiene la información y es, por consiguiente, un valor de un tipo genérico (denominado `Dato`, `TipoElemento`, `Info`, etc.) y la segunda parte es una referencia (denominada *enlace*) que apunta, enlaza, al siguiente elemento de la lista.

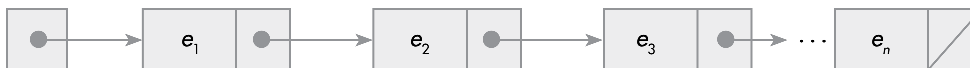


Figura 36.1 Lista enlazada (representación gráfica típica).

Los enlaces se representan por flechas para facilitar la comprensión de la conexión entre dos nodos; ello indica que el enlace tiene la dirección en memoria del siguiente nodo. Los enlaces también sitúan los nodos en una secuencia. El primer nodo se enlaza al segundo nodo, el segundo se enlaza al tercero y así sucesivamente hasta llegar al último. El nodo último ha de ser representado de forma diferente para significar que éste no se enlaza a ningún otro. Las listas se pueden dividir en cuatro categorías.

A recordar

Una **lista enlazada** consta de un número de elementos y cada elemento tiene dos componentes (*campos*), una referencia al siguiente elemento de la lista y un valor, que puede ser de cualquier tipo.

Clasificación de las listas enlazadas

Los diferentes tipos de listas dependen de la forma de enlazar los nodos, son:

- *Listas simplemente enlazadas*. Cada nodo (elemento) contiene un único enlace que conecta ese nodo al nodo siguiente o nodo sucesor. La lista es eficiente en recorridos directos ("adelante").
- *Listas doblemente enlazadas*. Cada nodo contiene dos enlaces, uno a su nodo predecesor y el otro a su nodo sucesor. La lista es eficiente tanto en recorrido directo ("adelante") como en recorrido inverso ("atrás").
- *Lista circular simplemente enlazada*. Una lista enlazada simplemente en la que el último elemento (cola) se enlaza al primer elemento (cabeza) de tal modo que la lista puede ser recorrida de modo circular ("en anillo").
- *Lista circular doblemente enlazada*. Una lista doblemente enlazada en la que el último elemento se enlaza al primer elemento y viceversa. Esta lista se puede recorrer de modo circular (en anillo) tanto en dirección directa ("adelante") como inversa ("atrás").

La implementación de cada uno de los cuatro tipos de estructuras de listas se puede desarrollar utilizando referencias.

El primer nodo, **frente**, de una lista es el nodo apuntado por **cabeza**. La lista encadena nodos juntos desde el frente hasta el final (**cola**) de la lista. El final se identifica como el nodo cuyo campo referencia tiene el valor `null`.

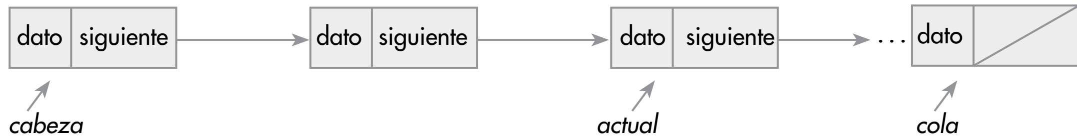


Figura 36.2 Representación gráfica de una lista enlazada.

TAD Lista

Una lista almacena información del mismo tipo, con la característica de que puede contener un número indeterminado de elementos, y que estos elementos mantienen un orden explícito. Este ordenamiento explícito se manifiesta en que, en sí mismo, cada elemento contiene la dirección del siguiente elemento. Una lista es una estructura de datos dinámica, el número de nodos puede variar rápidamente en un proceso, aumentando los nodos por inserciones, o bien, disminuyendo por eliminación de nodos.

Matemáticamente, una lista es una secuencia de cero o más elementos de un determinado tipo.

```
(a1, a2, a3, ... , an) siendo n >= 0,
                       si n = 0 la lista es vacía.
```

Los elementos de la lista tienen la propiedad de estar ordenados de forma lineal, según las posiciones que ocupan en la misma. Se dice que a_i precede a a_{i+1} para `i = 1 ... ,n-1`; y que a_i sucede a a_{i-1} para `i = 2 ... n`.

Para formalizar el *tipo de dato abstracto* `Lista` a partir de la noción matemática, se define un conjunto de operaciones básicas con objetos de tipo `Lista`. Las operaciones:

`∀ L ∈ Lista , ∀ x ∈ Dato, ∀ p ∈ puntero`

`Listavacia(L)`	Inicializa la lista `L` como lista vacía.
`Esvacia(L)`	Determina si la lista `L` está vacía.
`Insertar(L, x, p)`	Inserta en la lista `L` un nodo con el campo dato `x`, delante del nodo de dirección `p`.
`Localizar(L, x)`	Devuelve la posición/dirección donde está el campo de información `x`.
`Suprimir(L, x)`	Elimina de la lista el nodo que contiene el dato `x`.
`Anterior(L, p)`	Devuelve la posición/dirección del nodo anterior a `p`.
`Primero(L)`	Devuelve la posición/dirección del primer nodo de la lista `L`.
`Anula(L)`	Vacía la lista `L`.

Estas operaciones son las que pueden considerarse básicas para manejar listas. En realidad, la decisión de qué operaciones son las básicas depende de las características de la aplicación que se va a realizar con los datos de la lista.

Clase `lista`

Una lista enlazada se compone de una serie de nodos enlazados mediante referencias. En Java, se declara una clase para contener las dos partes del nodo: dato y enlace. Por ejemplo, para una lista enlazada de números enteros la clase `Nodo`:

```
class Nodo
{
  protected int dato;
  protected Nodo enlace;
  public Nodo(int t)
  {
    dato = t;
    enlace = null;
  }
}
```

Dado que los tipos de datos que se puede incluir en una lista pueden ser de cualquier tipo (enteros, dobles, caracteres o cualquier objeto), con el fin de que el tipo de dato de cada nodo se pueda cambiar con facilidad, a veces se define la clase `Elemento` como una generalización del tipo de dato de cada campo. En ese caso se utiliza una referencia a `Elemento` dentro del nodo, como se muestra a continuación:

```
class Elemento
{
  // ...
}

class Nodo
{
  protected Elemento dato;
    protected Nodo enlace;
}
```

Ejemplo 36.1

Se declara la clase `Punto`*, representa un punto en el plano con su coordenada* x *y* y*. También, la clase* `Nodo` *con el campo dato referencia a objetos de la clase* `Punto`*. Estas clases formarán parte del paquete* `ListaPuntos`*.*

```
package ListaPuntos;

public class Punto
{
  double x, y;

  public Punto(double x, double y)
  {
    this.x = x;
    this.y = y;
  }
  public Punto( ) // constructor por defecto
  {
    x = y = 0.0;
  }
}
```

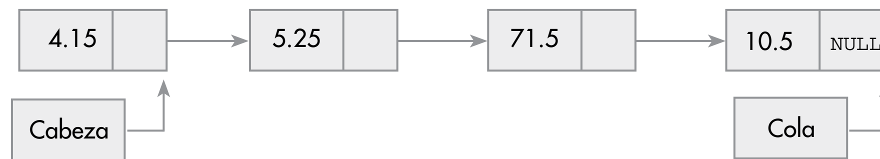
La clase `Nodo` que se escribe a continuación, tiene como campo `dato` una referencia a `Punto`, y el campo `enlace` una referencia a otro `nodo`. Se definen dos constructores, el primero inicializa `dato` a un objeto `Punto` y `enlace` a `null`. El segundo, inicializa `enlace` de tal forma que referencia a un `nodo`.

```
package ListaPuntos;

public class Nodo
{
  protected Punto dato;
  protected Nodo enlace;
  public Nodo(Punto p)
  {
    dato = p;
    enlace = null;
  }
  public Nodo(Punto p, Nodo n)
  {
    dato = p;
    enlace = n;
  }
}
```

Definición de la clase `lista`

El acceso a una lista se hace mediante una, o más, *referencias* a los nodos. Normalmente, se accede a partir del primer nodo de la lista, llamado **cabeza** o **cabecera** de la lista. En ocasiones, se mantiene también una referencia al último nodo de la lista enlazada, llamado **cola** de la lista.



Definición nodo

```
 class Nodo
{
  double dato;
  Nodo enlace;
  public Nodo( ){;}
}
```

Definición de referencias

```
Nodo cabeza;

Nodo cola;
```

Figura 36.3 Declaraciones de tipos en lista enlazada.

A recordar

La construcción y manipulación de una lista enlazada requiere el acceso a los nodos de la lista a través de una o más referencias a nodos. Normalmente, se incluye una referencia al primer nodo (*cabeza*) y además, en algunas aplicaciones, una referencia al último nodo (*cola*).

Nota de programación

La referencia cabeza (y cola) de una lista enlazada, normalmente se inicializa a `null`, indica lista vacía (no tiene nodos), cuando se inicia la construcción de una lista. Cualquier método que se escriba para implementar listas enlazadas debe poder manejar una referencia de cabeza (y de cola) `null`.

Error

Uno de los errores típicos en el tratamiento de referencias consiste en escribir la expresión `p.miembro` cuando el valor de la referencia `p` es `null`.

La clase **Lista** define el atributo `cabeza` o `primero` para acceder a los elementos de la lista. Normalmente, no es necesario definir el atributo referencia `cola`. El constructor de `lista` inicializa `primero` a `null` (*lista vacía*).

Los métodos de la clase `lista` implementan las operaciones de una lista enlazada: *inserción*, *búsqueda*... Además, el método `crearLista( )` construye iterativamente el primer elemento (`primero`) y los elementos sucesivos de una lista enlazada.

A continuación se declara la clase `lista` para representar una lista enlazada de números enteros. La declaración se realiza paso a paso con el fin de describir detalladamente las operaciones. En primer lugar, la declaración del nodo de la lista con el dato de la lista, dos constructores básicos y métodos de acceso:

```
package ListaEnteros;

public class Nodo
{
  protected int dato;
  protected Nodo enlace;
  public Nodo(int x)
  {
    dato = x;
    enlace = null;
```

```
    }
    public Nodo(int x, Nodo n)
    {
      dato = x;
      enlace = n;
    }
    public int getDato( )
    {
      return dato;
    }
    public Nodo getEnlace( )
    {
      return enlace;
    }
    public void setEnlace(Nodo enlace)
    {
      this.enlace = enlace;
    }
  }
```

A continuación la clase `lista`:

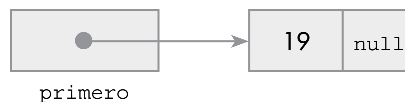
```
package ListaEnteros;

public class Lista
{
  private Nodo primero;
  public Lista( )
  {
    primero = null;
  }
  //...
```

La referencia `primero` (a veces se denomina `cabeza`) se ha inicializado en el constructor a un valor *nulo*, es decir, a *lista vacía*.

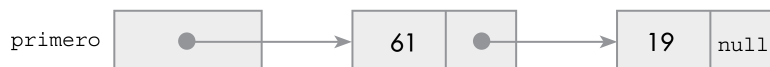
El método `crearLista( )`, en primer lugar crea un nodo con un valor y su referencia se asigna a `primero`:

```
primero = new Nodo(19);
```



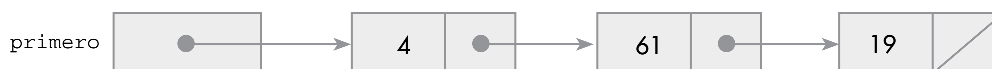
La operación de crear un nodo se puede realizar en un método al que se pasa el valor del campo `dato` y del campo `enlace`. Si ahora se desea añadir un nuevo elemento con el `61`, y situarlo en el primer lugar de la lista:

```
primero = new Nodo(61,primero);
```



Por último, para obtener una lista compuesta de `4`, `61`, `19` se habrá de ejecutar

```
primero = new Nodo(4,primero);
```



El método `crearLista( )` construye una lista y devuelve una referencia al objeto lista creado (`this`).

```
private int leerEntero( ) {;}

public Lista crearLista( )
{
  int x;
```

```
    primero = null;
    do {
      x = leerEntero( );
      if (x != -1)
      {
        primero = new Nodo(x,primero);
      }
    } while (x != -1);
    return this;
  }
```

Inserción de un elemento en una lista

El nuevo elemento que se desea incorporar a una lista se puede insertar de distintas formas, según la posición de inserción. Ésta puede ser:

- En la cabeza (elemento primero) de la lista.
- En el final de la lista (elemento último).
- Antes de un elemento especificado, o bien,
- Después de un elemento especificado.

Insertar un nuevo elemento en la cabeza de la lista

La posición más eficiente para insertar un nuevo elemento es la *cabeza*, es decir, por el primer nodo de la lista. El proceso de inserción se resume en este algoritmo:

1. Crear un nodo e inicializar el campo `dato` al nuevo elemento. La referencia del nodo creado se asigna a `nuevo`, variable local del método.
2. Hacer que el campo `enlace` del nuevo nodo apunte a la *cabeza* (`primero`) de la lista original.
3. Hacer que `primero` apunte al nodo que se ha creado.

El código fuente del método `insertarCabezaLista`:

```
public Lista insertarCabezaLista(Elemento entrada)
{
  Nodo nuevo ;
  nuevo = new Nodo(entrada);
  nuevo.enlace = primero;          // enlaza nuevo nodo al frente de la lista
  primero = nuevo;                 // mueve primero y apunta al nuevo nodo
  return this;                     // devuelve referencia del objeto Lista
}
```

Insertar entre dos nodos de la lista

Por ejemplo, en la lista de la figura 36.4 insertar el elemento 75 entre los nodos con los datos 25 y 40.

El algoritmo para la operación de insertar entre dos nodos (`n1, n2`) requiere las siguientes etapas:

1. Crear un nodo con el nuevo elemento y el campo `enlace` a `null`. La referencia al nodo se asigna a `nuevo`.
2. Hacer que el campo `enlace` del nuevo nodo apunte al nodo `n2`, ya que el nodo creado se ubicará justo antes de `n2` (en el ejemplo de la figura 36.4, el nodo `40`).
3. La variable referencia `anterior` tiene la dirección del nodo `n1` (en el ejemplo de la figura 36.4, el nodo `25`), entonces hacer que `anterior.enlace` apunte al nodo creado.

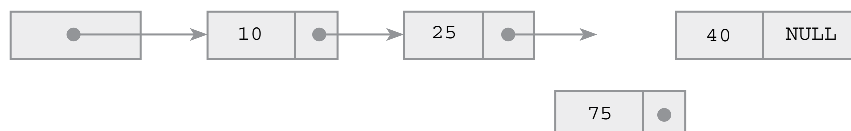
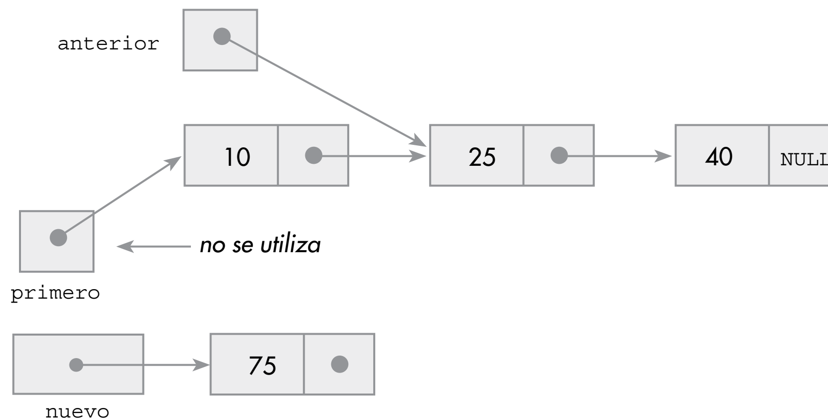


Figura 36.4 Inserción entre dos nodos.

Gráficamente las etapas del algoritmo y el código que implementa la operación.

Etapa 1

Se crea un nodo con el dato `75`. La variable `anterior` apunta al nodo `n1`, en la figura `25`.

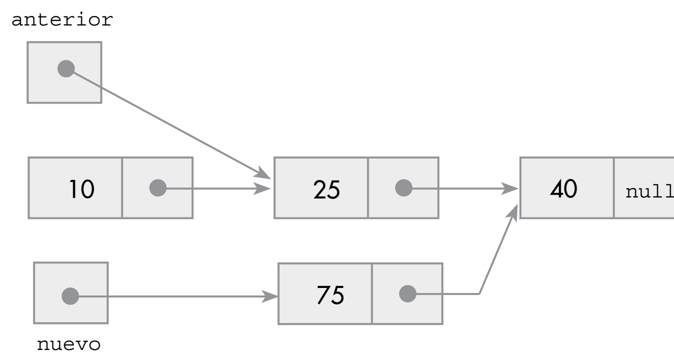


Código Java

```
nuevo = new Nodo(entrada);
```

Etapa 2

El campo enlace del nodo creado que apunte a `n2`, en la figura `40`. La dirección de `n2` se consigue con `anterior.enlace`:

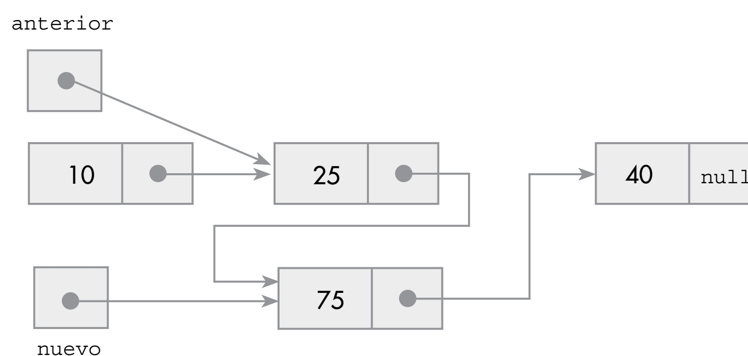


Código Java

```
nuevo.enlace = anterior.enlace
```

Etapa 3

Por último, el campo enlace del nodo `n1` (`anterior`) que apunte al nodo creado:



La operación es un método de la clase `lista`:

```
public Lista insertarLista(Nodo anterior, Elemento entrada)
{
  Nodo nuevo;

  nuevo = new Nodo(entrada);
  nuevo.enlace = anterior.enlace;
  anterior.enlace = nuevo;
  return this;
}
```

Antes de llamar al método `insertarLista( )`, es necesario buscar la dirección del nodo `n1`, esto es, del nodo a partir del cual se enlazará el nodo que se va a crear.

Búsqueda en listas enlazadas

La operación *búsqueda* de un elemento en una lista enlazada recorre la lista hasta encontrar el nodo con el elemento. El algoritmo, una vez encontrado el nodo, devuelve la referencia a ese nodo (en caso negativo, devuelve `null`). Otro planteamiento es que el método devuelva `true` si encuentra el nodo con el elemento, y `false` si no está en la lista.

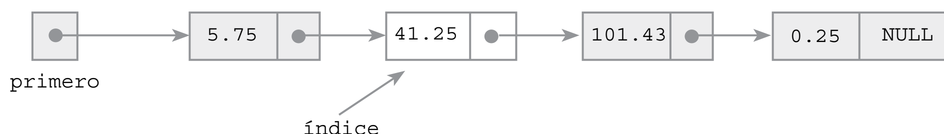


Figura 36.5 Búsqueda en una lista.

El método `buscarLista`, de la clase `Lista`, utiliza la referencia `indice` para recorrer la lista, nodo a nodo. El bucle de búsqueda inicializa `indice` al nodo `primero`, compara el nodo referenciado por `indice` con el elemento buscado. Si coincide, la búsqueda termina; en caso contrario, `indice` avanza al siguiente nodo. La búsqueda termina cuando se encuentra el nodo, o bien cuando se ha recorrido la lista y entonces `indice` toma el valor `null`. La comparación entre el dato buscado y el dato del nodo, que realiza el método `buscarLista( )`, utiliza el operador `==` por claridad; realmente sólo se utiliza dicho operador si los datos son de tipo simple (`int`, `double`, ...). Normalmente, los datos de los nodos son objetos y entonces se utiliza el método `equals( )` que compara dos objetos.

Código Java

```
public Nodo buscarLista(Elemento destino)
{
  Nodo indice;

  for (indice = primero; indice!= null; indice = indice.enlace)
    if (destino == indice.dato) // (destino.equals(indice.dato))
      return indice;
  return null;
}
```

Borrado de un nodo de una lista

La operación de eliminar un nodo de una lista enlazada supone enlazar el nodo anterior con el nodo siguiente al que se desea eliminar y liberar la memoria que ocupa. El algoritmo sigue estos pasos:

1. Búsqueda del nodo que contiene el dato. Se ha de obtener la dirección del nodo a eliminar y la dirección del anterior.
2. El `enlace` del nodo anterior que apunte al nodo siguiente del que se elimina.
3. Si el nodo a eliminar es el *cabeza* de la lista (`primero`), se modifica `primero` para que tenga la dirección del siguiente nodo.
4. Por último, la memoria ocupada por el nodo se libera. Es el propio sistema el que libera el nodo, al dejar de estar referenciado.

A continuación se escribe el método `eliminar( )`, miembro de la clase `Lista`, recibe el dato del nodo que se quiere borrar. Construye su bucle de búsqueda con el fin de disponer de la dirección del nodo anterior.

Código Java

```
public void eliminar (Elemento entrada)
{
  Nodo actual, anterior;
  boolean encontrado;

  actual = primero;
  anterior = null;
  encontrado = false;
   // búsqueda del nodo y del anterior
  while ((actual!=null) && (!encontrado))
  {
    encontrado = (actual.dato == entrada);
    //con objetos: actual.dato.equals(entrada)
    if (!encontrado)
    {
      anterior = actual;
      actual = actual.enlace;
    }
  }
   // Enlace del nodo anterior con el siguiente
  if (actual != null)
  {
    // Distingue entre que el nodo sea el cabecera,
    // o del resto de la lista
    if (actual == primero)
    {
      primero = actual.enlace;
    }
    else
    {
      anterior.enlace = actual.enlace;
    }
    actual = null; // no es necesario al ser una variable local
  }
}
```

Lista ordenada

Los elementos de una lista tienen la propiedad de estar ordenados de forma lineal, según las posiciones que ocupan en la misma. Ahora bien, también es posible mantener una lista enlazada ordenada según el dato asociado a cada nodo. La figura 36.6 muestra una lista enlazada de números reales, ordenada de forma creciente.

La forma de insertar un elemento en una lista ordenada siempre realiza la operación de tal forma que la lista resultante mantiene la propiedad de ordenación. Para esto, en primer lugar determina la posición de inserción, y a continuación ajusta los enlaces.

Por ejemplo, para insertar el dato `104` en la lista de la figura 36.7 es necesario recorrer la lista hasta el nodo con `110.0`, que es el inmediatamente mayor. El *pointer* `indice` se queda con la dirección del nodo anterior, a partir del cual se enlaza el nuevo nodo.

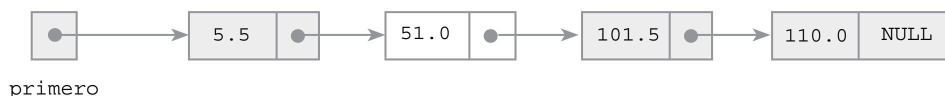


Figura 36.6 Lista ordenada.

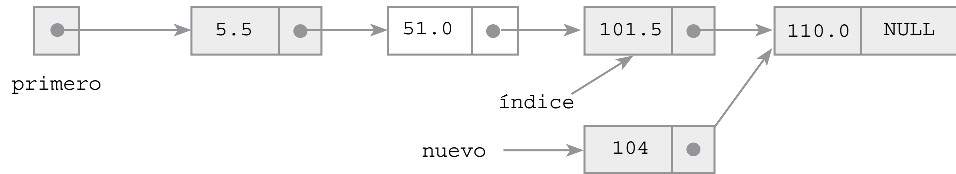


Figura 36.7 Inserción en una lista ordenada.

El método `insertaOrden( )` crea una lista ordenada; el punto de partida es una lista vacía, a la que se añaden nuevos elementos, de tal forma que en todo momento el orden de los elementos es creciente. La inserción del primer nodo de la lista consiste, sencillamente, en crear el nodo y asignar su referencia a la cabeza de la lista. El segundo elemento se ha de insertar antes o después del primero, dependiendo de que sea menor o mayor. En general, para insertar un nuevo elemento a la lista ordenada, primero se busca la posición de inserción en la lista actual, es decir, el nodo a partir del cual se ha de enlazar el nuevo nodo para que la lista mantenga la ordenación.

Los datos de una lista ordenada han de ser de tipo ordinal (tipo al que se pueda aplicar los operadores ==, <, >); o bien objetos de clases que tengan definidos métodos de comparación (`equals( )`, `compareTo( )`, ...). A continuación se escribe el código Java que implementa el método para una lista de enteros.

```
public ListaOrdenada insertaOrden(int entrada)
{
  Nodo nuevo ;
  nuevo = new Nodo(entrada);

  if (primero == null) // lista vacía
    primero = nuevo;
  else if (entrada < primero.getDato( ))
  {
    nuevo. setEnlace(primero);
    primero = nuevo;
  }
  else /* búsqueda del nodo anterior a partir del que
          se debe insertar */
  {
    Nodo anterior, p;
    anterior = p = primero;
    while ((p.getEnlace( ) != null) && (entrada > p.getDato( )))
    {
      anterior = p;
      p = p.getEnlace( );
    }
    if (entrada > p.getDato( )) //se inserta después del último nodo
      anterior = p;
    // Se procede al enlace del nuevo nodo
    nuevo.setEnlace(anterior.getEnlace( ));
    anterior.setEnlace(nuevo);
  }
  return this;
}
```

Clase `ListaOrdenada`

Se declara la clase `ListaEnlazada` como extensión (derivada) de la clase `lista`. Por consiguiente, hereda las propiedades de `lista`. Los métodos `eliminar( )` y `buscarLista( )` se deben redefinir para que la búsqueda del elemento aproveche el hecho de que éstos están ordenados.

La declaración de la clase:

```
public class ListaOrdenada extends Lista
{
  public ListaOrdenada( )
```

```
  {
    super( );
  }
  public ListaOrdenada insertaOrden(int entrada) // ya definido

  // métodos a codificar:

  public void eliminar (int entrada){ ; }
  public Nodo buscarLista(int destino){ ; }
}
```

Lista doblemente enlazada

Existen numerosas aplicaciones en las que es conveniente poder acceder a los elementos o nodos de una lista en cualquier orden, tanto hacia adelante como hacia atrás. La estructura **lista doblemente enlazada** hace posible el acceso bidireccional a los elementos. Cada elemento de la lista dispone de dos *pointers* (referencias), además del valor almacenado. Una referencia apunta al siguiente elemento de la lista y la otra referencia apunta al elemento anterior.

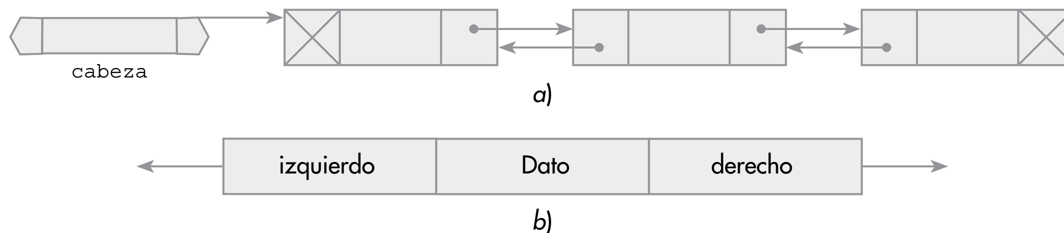


Figura 36.8 Lista doblemente enlazada. *a*) Lista con tres nodos; *b*) nodo.

Las operaciones de una *Lista doble* son similares a las de una *Lista: insertar*, *eliminar*, *buscar*, *recorrer*... Las operaciones que modifican la lista, *insertar* y *eliminar*, deben realizar ajustes de los dos *pointers* de cada nodo. La figura 36.9 muestra los ajustes de los enlaces que se deben realizar al insertar un nodo a la derecha del nodo *actual*.

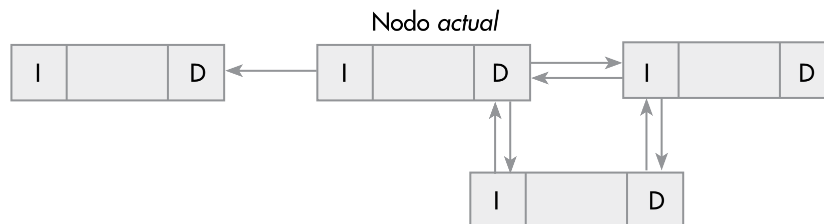


Figura 36.9 Inserción de un nodo en una lista doblemente enlazada.

Para eliminar un nodo de la lista doble se necesita enlazar el nodo anterior con el nodo siguiente del que se borra, como se observa en la figura 36.10.

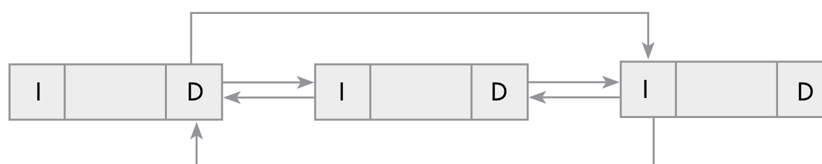


Figura 36.10 Eliminación de un nodo en una lista doblemente enlazada.

Clase *Lista Doble*

Un nodo de una lista doblemente enlazada tiene dos *pointer* (referencias) para enlazar con nodo izquierdo y derecho, además la parte correspondiente al campo dato. La declaración de la clase *Nodo*:

```
package listaDobleEnlace;

public class Nodo
{
  protected TipoDato dato;
  protected Nodo adelante;
  protected Nodo atras;
  public Nodo(int entrada)
  {
    dato = entrada;
    adelante = atras = null;
  }
    // ...
}
```

La clase `ListaDoble` implementa la estructura *lista doblemente enlazada*. La clase dispone de la variable `cabeza` que referencia al primer nodo de la lista, y la variable `cola` para referenciar al último nodo.

```
package listaDobleEnlace;

public class ListaDoble
{
  protected Nodo cabeza;
  protected Nodo cola;
    // métodos de la clase (implementación)
  public ListaDoble( ){ cabeza = cola = null;}
  public ListaDoble insertarCabezaLista(TipoDato entrada){;}
  public ListaDoble insertaDespues(Nodo anterior, TipoDato entrada)
  public void eliminar (TipoDato entrada)
  public void visualizar( )
  public void buscarLista(TipoDato destino)
}
```

Insertar un elemento en una lista doblemente enlazada

Se puede añadir un nuevo nodo a la lista de distintas formas, según la posición donde se inserte el nodo. La posición de inserción puede ser:

- En la *cabeza* (elemento primero) de la lista.
- Al final de la lista (elemento último).
- Antes de un elemento especificado.
- Después de un elemento especificado.

El algoritmo de inserción depende de la posición donde se inserte. Los pasos que se siguen para insertar después de un nodo `n` son:

1. Crear un nodo con el nuevo elemento y asignar su referencia a la variable `nuevo`.
2. Hacer que el enlace `adelante` del nuevo nodo apunte al nodo siguiente de `n` (o bien a `null` si `n` es el último nodo). El enlace `atras` del nodo siguiente a `n` (si `n` no es el último nodo) tiene que apuntar a `nuevo`.
3. Hacer que el enlace `adelante` del nodo `n` apunte al nuevo nodo. A su vez, el enlace `atras` del nuevo nodo debe apuntar a `n`.

El método de la clase `ListaDoble`, `insertaDespues( )`, implementa el algoritmo. El primer argumento, `anterior`, representa el nodo `n` a partir del cual se enlaza el nuevo nodo. El segundo argumento, `entrada`, es el dato que se añade a la lista.

Código Java

```
public ListaDoble insertaDespues(Nodo anterior, TipoDato entrada)
{
  Nodo nuevo;

  nuevo = new Nodo(entrada);
  nuevo.adelante = anterior.adelante;
  if (anterior.adelante !=null)
    anterior.adelante.atras = nuevo;
  anterior.adelante = nuevo;
  nuevo.atras = anterior;
  if (anterior == cola) cola = nuevo; // es el último elemento
  return this;
}
```

Eliminar un elemento de una lista doblemente enlazada

Quitar un nodo de una lista doble supone realizar el enlace de dos nodos, el nodo *anterior* con el nodo *siguiente* al que se desea eliminar. La referencia `adelante` del nodo anterior debe apuntar al nodo *siguiente*, y la referencia `atras` del nodo *siguiente* debe apuntar al nodo *anterior*.

El algoritmo es similar al del borrado para una lista simple. Ahora, la dirección del nodo *anterior* se encuentra en la referencia `atras` del nodo a borrar. Los pasos a seguir son:

1. Búsqueda del nodo que contiene el dato.
2. La referencia `adelante` del nodo anterior tiene que apuntar a la referencia `adelante` del nodo a eliminar (si no es el nodo *cabecera*).
3. La referencia `atras` del nodo siguiente a borrar tiene que apuntar a la referencia `atras` del nodo a eliminar (si no es el último nodo).
4. Si el nodo que se elimina es el primero, `cabeza`, se modifica `cabeza` para que tenga la dirección del nodo `siguiente`.
5. La memoria ocupada por el nodo es liberada automáticamente.

El método `eliminar`, de la clase `ListaDoble`, implementa el algoritmo:

```
public void eliminar (TipoDato entrada)
{
  Nodo actual;
  boolean encontrado = false;

  actual = cabeza;
   // Bucle de búsqueda
  while ((actual != null) && (!encontrado))
  {
    /* La comparación se podrá realizar con el método equals( )...,
       depende del tipo que tenga entrada */
    encontrado = (actual.dato == entrada);
    if (!encontrado)actual = actual.adelante;
  }
   // Enlace de nodo anterior con el siguiente
  if (actual != null)
  {
    //distingue entre nodo cabecera o resto de la lista
    if (actual == cabeza)
    {
      cabeza = actual.adelante;
      if (actual.adelante != null)
        actual.adelante.atras = null;
    }
    else if (actual.adelante != null) // No es el último nodo
    {
      actual.atras.adelante = actual.adelante;
      actual.adelante.atras = actual.atras;
    }
    else // último nodo
      actual.atras.adelante = null;
```

```
      if (actual == cola) cola = actual.atras;
      actual = null;
    }
  }
```

Lista circular

En las listas lineales simples o dobles siempre hay un primer nodo (cabeza) y un último nodo (cola). Una *lista circular*, por propia naturaleza, no tiene ni principio ni fin. Sin embargo, resulta útil establecer un nodo a partir del cual se acceda a la lista y así poder acceder a sus nodos.

La figura 36.11 muestra una lista circular con enlace simple; podría considerarse que es una lista lineal cuyo último nodo apunte al primero.

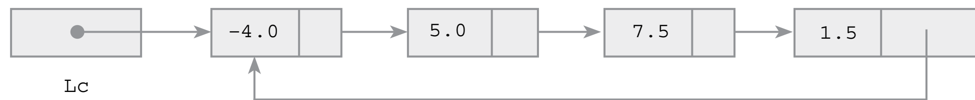


Figura 36.11 Lista circular.

Las operaciones que se realizan sobre una lista circular son similares a las operaciones sobre listas lineales, teniendo en cuenta que no hay `primero` ni `último` nodo, aunque sí un nodo de acceso a la lista. Estas operaciones permiten construir el *TAD lista circular* y su funcionalidad es la siguiente:

- Inicialización o creación.
- Inserción de elementos en una lista circular.
- Eliminación de elementos de una lista circular.
- Búsqueda de elementos de una lista circular.
- Recorrido de cada uno de los nodos de una lista circular.
- Verificación de lista vacía.

La construcción de una lista circular se puede hacer con enlace simple o enlace doble. La implementación que se desarrolla, en este apartado, enlaza dos nodos con un enlace simple.

Se declara la clase `Nodo`, con el campo dato y enlace; y la clase `ListaCircular` con el pointer de acceso a la lista, junto a los métodos que implementan las operaciones. Los elementos de la lista pueden ser de cualquier tipo, se puede abstraer el tipo de éstos en otra clase, por ejemplo `Elemento`; con el fin de simplificar se supone un tipo conocido.

El constructor de la clase `Nodo` varía respecto al de las listas no circulares; el campo referencia `enlace`, en vez de quedar a `null`, se inicializa para que apunte al mismo nodo, de tal forma que queda como lista circular de un solo nodo.

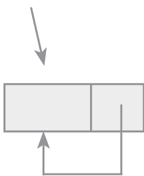


Figura 36.12 Creación de un nodo de lista circular.

```
package listaCircular;

public class Nodo
{
  Elemento dato;
  Nodo enlace;
  public Nodo (Elemento entrada)
  {
    dato = entrada;
    enlace = this; // se apunta a sí mismo
  }
}
```

> ***A tener en cuenta***
> El pointer de acceso a una lista circular, `lc`, normalmente apunta al último nodo añadido a la estructura. Esta convención puede cambiar ya que en una estructura circular no hay *primero* ni *último*.

Clase *Lista Circular*

La clase `ListaCircular` declara la variable `lc` a partir de la cual se puede acceder a cualquier nodo de la lista. El constructor de la clase inicializa la *lista vacía*.

```
package listaCircular;

public class ListaCircular
{
  private Nodo lc;

  public ListaCircular( )
  {
    lc = null;
  }
  public ListaCircular insertar(Elemento entrada)
  {
    Nodo nuevo;

    nuevo = new Nodo(entrada);
    if (lc != null) // lista circular no vacía
    {
      nuevo.enlace = lc.enlace;
      lc.enlace = nuevo;
    }
    lc = nuevo;
    return this;
}
```

El algoritmo empleado por el método `insertar`, realiza la inserción en el nodo anterior al de acceso a la lista `lc` y considera que `lc` tiene la dirección del último nodo insertado.

Eliminar un elemento de una lista circular

Para eliminar un nodo hay que enlazar el nodo anterior con el nodo siguiente al que se desea eliminar y que el sistema libere la memoria que ocupa. El algoritmo es:

1. Búsqueda del nodo que contiene el dato.
2. Enlace del nodo anterior con el siguiente.
3. En caso de que el nodo a eliminar sea por el que se accede a la lista, `lc`, se modifica `lc` para que tenga la dirección del nodo anterior.
4. Por último, el sistema libera la memoria ocupada por el nodo al anular la referencia.

La implementación del método debe tener en cuenta que la lista circular tendrá un solo nodo, ya que al eliminarlo la lista se queda vacía (`lc = null`). La condición de lista con un nodo se corresponde con la forma de inicializar un nodo: `lc = lc.enlace`.

El método recorre la lista buscando el nodo con el dato a eliminar, utiliza un *pointer* al nodo *anterior* para que cuando se encuentre el nodo se enlace con el *siguiente*. Se accede al dato con la sentencia `actual.enlace.dato`, con el fin de que si coincide con el dato a eliminar, se disponga en `actual` el nodo anterior.

Código Java

```
public void eliminar(Elemento entrada)
{
  Nodo actual;
  boolean encontrado = false;

  actual = lc;
  while ((actual.enlace != lc) && (!encontrado))
  {
    encontrado = (actual.enlace.dato == entrada);
    if (!encontrado)
    {
      actual = actual.enlace;
    }
  }
```

```
      encontrado = (actual.enlace.dato == entrada);
        // Enlace de nodo anterior con el siguiente
      if (encontrado)
      {
        Nodo p;
        p = actual.enlace;      // Nodo a eliminar
        if (lc == lc.enlace)    // Lista con un solo nodo
          lc = null;
        else
        {
          if (p == lc)
          {
            lc = actual;        // Se borra el elemento referenciado por lc,
                                // el nuevo acceso a la lista es el anterior
          }
          actual.enlace = p.enlace;
        }
        p = null;
      }
    }
```

Recorrer una lista circular

Una operación común a todas las estructuras enlazadas es recorrer o visitar todos los nodos de la estructura. En una lista circular el recorrido puede empezar en cualquier nodo e iterativamente ir procesando cada nodo hasta alcanzar el nodo de partida. El método `recorrer`, miembro de la clase `ListaCircular`, inicia el recorrido en el nodo siguiente al de acceso a la lista, `lc`, y termina cuando alcanza el nodo `lc`.

```
public void recorrer( )
{
  Nodo p;
  if (lc != null)
  {
    p = lc.enlace; // siguiente nodo al de acceso
    do {
      System.out.println("\t" + p.dato);
      p = p.enlace;
    } while(p != lc.enlace);
  }
  else
    System.out.println("\t Lista Circular vacía.");
}
```

Tipo abstracto de datos `pila`

Una **pila** (*stack*) es una colección ordenada de elementos a los que sólo se puede acceder por un único lugar o extremo de la pila. Los elementos de la pila se añaden o quitan (borran) de la misma sólo por su parte superior (**cima**) de la pila.

Las entradas de la pila deben ser eliminadas en el orden inverso al que se situaron en la misma. Por ejemplo, se puede crear una pila de libros, situando primero un diccionario, encima de él una enciclopedia y encima de ambos una novela de modo que la pila tendrá la novela en la parte superior.

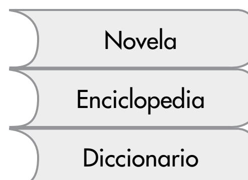


Figura 36.13 Pila de libros.

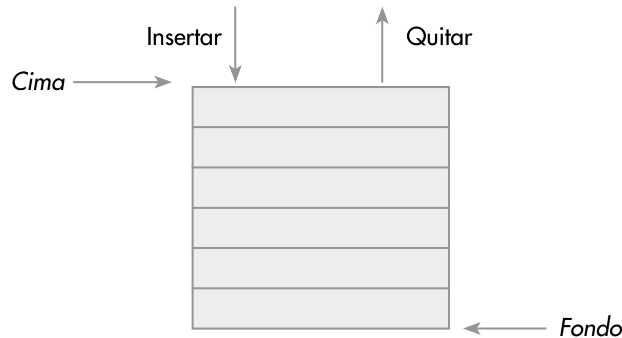


Figura 36.14 Operaciones básicas de una pila.

Debido a su propiedad específica "*último en entrar, primero en salir*" se conoce a las pilas como *estructura de datos* **LIFO** (*last-in/first-out*).

La pila se puede implementar guardando los elementos en un arreglo en cuyo caso su dimensión o longitud es fija. También se puede utilizar un `Vector` para almacenar los elementos. Otra forma de implementación consiste en construir una lista enlazada, cada elemento de la pila forma un nodo de la lista; la lista crece o decrece según se añaden o se extraen, respectivamente, elementos de la pila; ésta es una representación dinámica y no existe limitación en su tamaño excepto la memoria de la computadora.

Una pila puede estar *vacía* (no tiene elementos) o *llena* (en la representación con un arreglo, si se ha llegado al último elemento). Si un programa intenta sacar un elemento de una pila vacía, se producirá un error, una *excepción*, debido a que esa operación es imposible; esta situación se denomina **desbordamiento negativo** (*underflow*). Por el contrario, si un programa intenta poner un elemento en una pila *llena* se produce un error, o *excepción*, de **desbordamiento** (*overflow*) o *rebosamiento*.

Especificaciones de una pila

Las operaciones que sirven para definir una pila y poder manipular su contenido son las siguientes.

Tipo de dato — Elemento que se almacena en la pila

Operaciones

`CrearPila`	Inicia
Insertar (push)	Pone un dato en la pila
Quitar (pop)	Retira (saca) un dato de la pila
Pila vacía	Comprueba si la pila no tiene elementos
Pila llena	Comprueba si la pila está llena de elementos
Limpiar pila	Quita todos sus elementos y deja la pila vacía
`CimaPila`	Obtiene el elemento cima de la pila
Tamaño de la pila	Número de elementos máximo que puede contener la pila

El ejemplo 36.2 muestra cómo declarar una pila con un arreglo lineal. El tipo de los elementos de la pila es `Object`, lo que permite que pueda contener cualquier tipo de objeto.

Ejemplo 36.2

Declarar la clase `Pila` *con elementos de tipo* `Object`*. Insertar y extraer de la pila datos de tipo entero.*

La declaración de la clase:

```
package TipoPila;
```

```
public class PilaLineal
{
  private static final int TAMPILA = 49;
  private int cima;
  private Object [ ]listaPila;
    // operaciones que añaden o extraen elementos
  public void insertar(Object elemento)
  public Object quitar( )throws Exception
  public Object cimaPila( )throws Exception
    // resto de operaciones
}
```

El siguiente segmento de código inserta los datos `11`, `50` y `22`:

```
PilaLineal pila = new PilaLineal( );
pila.insertar(new Integer(11));
pila.insertar(new Integer(50));
pila.insertar(new Integer(22));
```

Para extraer de la pila y asignar el dato a una variable:

```
Integer dato;
dato = (Integer) pila.quitar( );
```

Pila dinámica implementada con `Vector`

La clase `Vector` es un contenedor de objetos que puede ser manejado para que crezca y decrezca dinámicamente. Los elementos de `Vector` son de tipo `Object`. Dispone de métodos para asignar un elemento en una posición (`insertElementAt`), para añadir un elemento a continuación del último (`addElement`), para obtener el elemento que se encuentra en una posición determinada (`elementAt`), para eliminar un elemento (`removeElementAt`) ...

La posición del último elemento añadido a la pila se mantiene con la variable `cima`. Inicialmente `cima == -1`, que es la condición de *pila vacía*.

Insertar un nuevo elemento supone aumentar `cima` y asignar el elemento a la posición `cima` del `Vector`. La operación se realiza llamando al método `addElement`. No resulta necesario implementar el método `pilaLlena` ya que la capacidad del `Vector` crece dinámicamente.

El método `quitar( )` devuelve el elemento cima de la pila y lo elimina. Al utilizar un `Vector`, llamando a `removeElementAt(cima)` se elimina el elemento `cima`; a continuación se decrementa `cima`.

La implementación es:

```
package TipoPila;
import java.util.Vector;
public class PilaVector
{
  private static final int INICIAL = 19;
  private int cima;
  private Vector listaPila;
  public PilaVector( )
  {
    cima = -1;
    listaPila = new Vector(INICIAL);
  }
  public void insertar(Object elemento)throws Exception
  {
    cima++;
    listaPila.addElement(elemento);
  }

  public Object quitar( ) throws Exception
  {
    Object aux;
```

```
      if (pilaVacia( ))
      {
        throw new Exception ("Pila vacía, no se puede extraer.");
      }
      aux = listaPila.elementAt(cima);
      listaPila.removeElementAt(cima);
      cima--;
      return aux;
    }
    public Object cimaPila( ) throws Exception
    {
      if (pilaVacia( ))
      {
        throw new Exception ("Pila vacía, no se puede extraer.");
      }
      return listaPila.elementAt(cima);
    }
    public boolean pilaVacia( )
    {
      return cima == -1;
    }
    public void limpiarPila( )throws Exception
    {
      while (! pilaVacia( ))
        quitar( );
    }
  }
```

Nota de programación
Para utilizar una pila de elementos de tipo primitivo (`int`, `char`, `long`, `float`, `double` ...) es necesario, para insertar, crear un objeto de la correspondiente clase *envolvente* (`Integer`, `Character`, `Long`, `Float`, `Double`...) y pasar dicho objeto como argumento del método `insertar( )`.

Pila implementada con una lista enlazada

Guardar los elementos de la pila en una lista enlazada permite que la pila crezca o decrezca de manera indefinida. El tamaño se ajusta exactamente a su número de elementos.

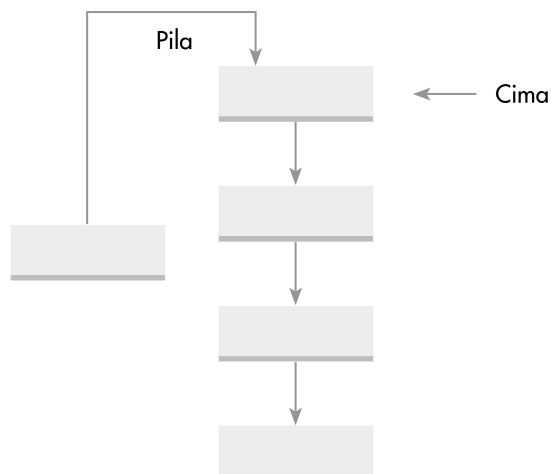


Figura 36. 15 Representación de una pila con una lista enlazada.

Clase `Pila` y `NodoPila`

Los elementos de la pila son los nodos de la lista, con un campo para guardar el elemento y otro de enlace. La clase `NodoPila` representa un nodo de la lista enlazada. Tiene dos atributos: `elemento` guarda el elemento de la pila y `siguiente` contiene la dirección del siguiente nodo de la lista. El constructor pone el dato en `elemento` e inicializa `siguiente` a `null`. El tipo de dato de elemento se corresponde con el tipo de los elementos de la pila, para que no dependa de un tipo concreto; para que sea más genérico se utiliza el tipo `Object` y de esa forma puede almacenar cualquier tipo de referencia.

```
package TipoPila;

public class NodoPila
{
  Object elemento;
  NodoPila siguiente;
  NodoPila(Object x)
  {
    elemento = x;
    siguiente = null;
  }
}
```

La clase `PilaLista` implementa las operaciones del *TAD pila*. Además, dispone del atributo `cima` que es la dirección del primer nodo de la lista. El constructor inicializa la pila vacía (`cima == null`), realmente, a la condición de *lista vacía*.

```
package TipoPila;

public class PilaLista
{
  private NodoPila cima;

  public PilaLista( )
  {
    cima = null;
  }
    // operaciones
}
```

Implementación de las operaciones

Las operaciones `insertar`, `quitar`, `cima` acceden a la lista directamente con la referencia `cima` (apunta al último nodo apilado). Entonces, como no necesitan recorrer los nodos de la lista, no dependen del número de nodos, la eficiencia de cada operación es constante, `O(1)`.

La clase `PilaLista` forma parte del mismo paquete que `NodoLista`, por ello tiene acceso a todos sus miembros.

Verificación del estado de la pila:

```
public boolean pilaVacia( )
{
  return cima == null;
}
```

Poner un elemento en la pila. Crea un nuevo nodo con el elemento que se pone en la pila y se enlaza por la cima.

```
public void insertar(Object elemento)
{
  NodoPila nuevo;
  nuevo = new NodoPila(elemento);
  nuevo.siguiente = cima;
  cima = nuevo;
}
```

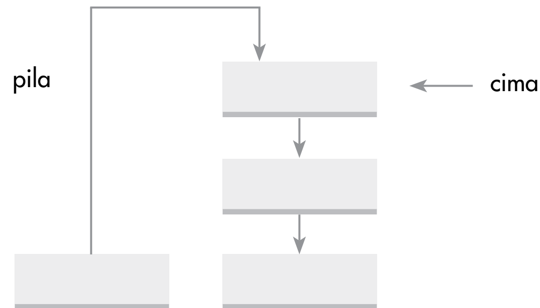


Figura 36.16 Apilar un elemento.

Eliminación del elemento cima. Retorna el elemento cima y lo quita de la pila.

```
public Object quitar( ) throws Exception
{
  if (pilaVacia( ))
    throw new Exception ("Pila vacía, no se puede extraer.");

  Object aux = cima.elemento;
  cima = cima.siguiente;
  return aux;
}
```

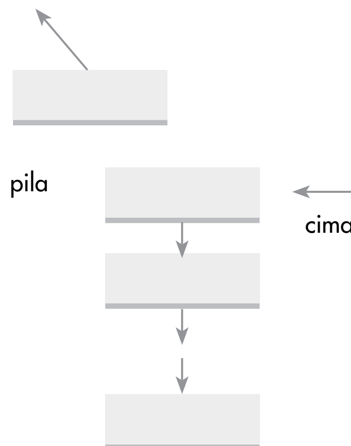


Figura 36.17 Quita la cima de la pila.

Obtención del elemento cabeza o cima de la pila, sin modificar la pila:

```
public Object cimaPila( ) throws Exception
{
  if (pilaVacia( ))
    throw new Exception ("Pila vacía, no se puede leer cima.");

  return cima.elemento;
}
```

Vaciado de la pila. Libera todos los nodos de que consta la pila. Recorre los `n` nodos de la lista enlazada, entonces es una operación lineal, `O(n)`.

```
public void limpiarPila( )
{
  NodoPila t;
  while(!pilaVacia( ))
  {
    t = cima;
    cima = cima.siguiente;
    t.siguiente = null;
  }
}
```

Tipo abstracto de datos `Cola`

Una **cola** es una estructura de datos que almacena elementos en una lista y permite acceder a los datos por uno de los dos extremos de la lista (figura 36.18). Un elemento se inserta en la cola (parte *final*) de la lista y se suprime o elimina por el frente (parte inicial, *frente*) de la lista. Las aplicaciones utilizan una cola para almacenar elementos en su orden de aparición o concurrencia.

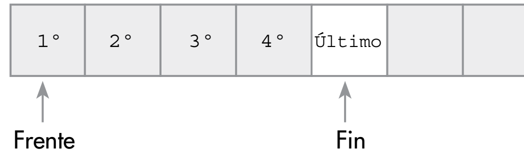


Figura 36.18 Una cola.

A recordar

Una cola es una estructura de datos cuyos elementos mantienen un cierto orden, tal que sólo se pueden añadir elementos por un extremo, **final** de la cola, y eliminar o extraer por el otro extremo, llamado **frente**.

Los elementos se eliminan (se quitan) de la cola en el mismo orden en que se almacena, por ello una cola es una estructura de tipo **FIFO** (*first-in/first-out*, *primero en entrar-primero en salir* o bien *primero en llegar-primero en ser servido*).

Especificaciones del tipo abstracto de datos Cola

Las operaciones que sirven para definir una cola y poder manipular su contenido son las siguientes:

Tipo de dato	Elemento que se almacena en la cola
Operaciones	
`CrearCola`	Inicia la cola como vacía
Insertar	Añade un elemento por el final de la cola
Quitar	Retira (extrae) el elemento frente de la cola
Cola vacía	Comprobar si la cola no tiene elementos
Cola llena	Comprobar si la cola está llena de elementos
Frente	Obtiene el elemento frente o primero de la cola
Tamaño de la cola	Número de elementos máximo que puede contener la cola

Las colas se implementan utilizando una estructura estática (arreglos), o una estructura dinámica (listas enlazadas, `Vector`...). Utilizar un arreglo tiene el problema del avance lineal de `frente` y `fin`; este avance deja *huecos* por la *izquierda del arreglo*. Llega a ocurrir que `fin` alcanza el índice más alto del arreglo, sin poder añadir nuevos elementos y sin embargo haber posiciones libres a la izquierda de `frente`.

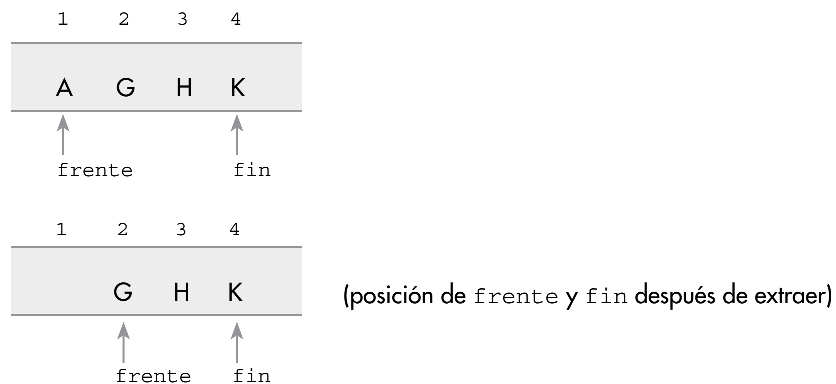


Figura 36.19 Una cola representada en un arreglo.

Cola con un arreglo circular

La forma más eficiente de almacenar una cola en un array es modelar a éste de tal forma que se una el extremo final con el extremo cabeza. Tal arreglo se denomina *arreglo circular* y permite que la totalidad de sus posiciones se utilicen para almacenar elementos de la cola sin necesidad de desplazar elementos. La figura 36.20 muestra un *arreglo circular* de *n* elementos.

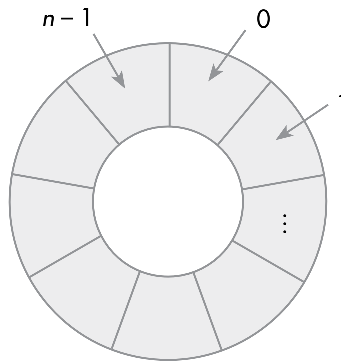


Figura 36.20 Un arreglo circular.

El arreglo se almacena de modo natural en la memoria, como un bloque lineal de *n* elementos. Se necesitan dos marcadores (apuntadores) *frente* y *fin* para indicar, respectivamente, la posición del elemento cabeza y del último elemento puesto en la cola.

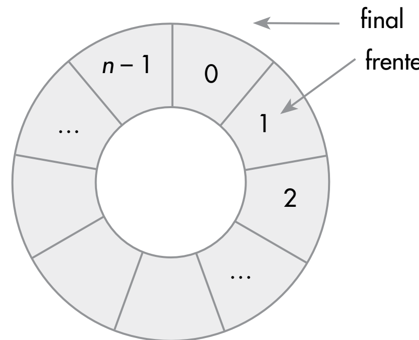


Figura 36.21 Una cola vacía.

El `frente` siempre contiene la posición del primer elemento de la cola y avanza en el sentido de las agujas del reloj; `fin` contiene la posición donde se puso el último elemento, también avanza en el sentido del reloj (circularmente a la derecha). La implementación del movimiento circular se realiza según la *teoría de los restos*, de tal forma que se generen índices de `0` a `MAXTAMQ-1`:

Mover `fin` adelante `= (fin + 1) % MAXTAMQ`
Mover `frente` adelante `= (frente + 1) % MAXTAMQ`

Los algoritmos que formalizan la gestión de colas en un arreglo circular han de incluir las operaciones básicas del *TAD cola*; en concreto, las siguientes tareas básicas:

- Creación de una cola vacía, de tal forma que `fin` apunte a una posición inmediatamente anterior a `frente`:
  `frente = 0; fin = MAXTAMQ-1.`
- Comprobar si una cola está vacía:
  `frente == siguiente(fin)`
- Comprobar si una cola está llena. Para diferenciar la condición entre *cola llena* y *cola vacía* se sacrifica una posición del arreglo, de tal forma que la capacidad de la cola va a ser `MAXTAMQ-1`. La condición de cola llena:
  `frente == siguiente(siguiente(fin))`

- Poner un elemento a la cola, si la cola no está llena, avanzar `fin` a la siguiente posición: `fin = (fin + 1) % MAXTAMQ` y asignar el elemento.
- Retirar un elemento de la cola, si la cola no está vacía, quitarlo de la posición `frente` y avanzar `frente` a la siguiente posición: `(frente + 1) % MAXTAMQ`.
- Obtener el elemento primero de la cola, si la cola no está vacía, sin suprimirlo de la cola.

Clase `cola` con *array circular*

La clase declara los apuntadores `frente`, `fin` y el arreglo `listaCola[ ]`. El método `siguiente( )` obtiene la posición *siguiente* de una dada, aplicando la *teoría de los restos*.

A continuación se codifica los métodos que implementan las operaciones del *TAD cola*. Ahora el tipo de los elementos es `Object`, de tal forma que se pueda guardar cualquier tipo de elementos.

```
package TipoCola;

public class ColaCircular
{
  private static fin int MAXTAMQ = 99;
  protected int frente;
  protected int fin;
  protected Object [ ]listaCola ;

  private int siguiente(int r)
  {
    return (r+1) % MAXTAMQ;
  }
  public ColaCircular( )
  {
    frente = 0;
    fin = MAXTAMQ-1;
    listaCola = new Object [MAXTAMQ];
  }
    // operaciones de modificación de la cola
  public void insertar(Object elemento) throws Exception
  {
    if (!colaLlena( ))
    {
      fin = siguiente(fin);
      listaCola[fin] = elemento;
    }
    else
      throw new Exception("Overflow en la cola");
  }
  public Object quitar( ) throws Exception
  {
    if (!colaVacia( ))
    {
      Object tm = listaCola[frente];
      frente = siguiente(frente);
      return tm;
    }
    else
      throw new Exception("Cola vacia ");
  }
  public void borrarCola( )
  {
    frente = 0;
    fin = MAXTAMQ-1;
  }
    // acceso a la cola
  public Object frenteCola( ) throws Exception
  {
    if (!colaVacia( ))
    {
```

```
        return listaCola[frente];
      }
      else
        throw new Exception("Cola vacia ");
    }
      // métodos de verificación del estado de la cola
    public boolean colaVacia( )
    {
      return frente == siguiente(fin);
    }

    public boolean colaLlena( )
    {
      return frente == siguiente(siguiente(fin));
    }
  }
```

Cola implementada con una lista enlazada

La implementación del *TAD Cola* con una lista enlazada permite ajustar el tamaño exactamente al número de elementos de la cola; la lista enlazada crece y decrece según las necesidades, según se incorporen elementos o se retiren. Utiliza dos apuntadores (referencias) para acceder a la lista, `frente` y `fin`, que son los extremos por donde salen y por donde se ponen, respectivamente, los elementos de la cola.

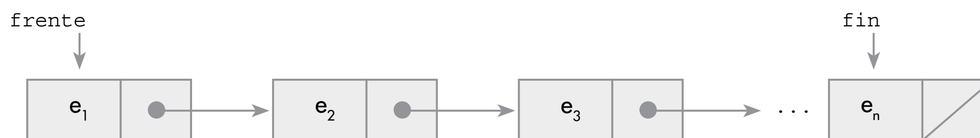


Figura 36.22 Cola con lista enlazada.

La referencia `frente` apunta al primer elemento de la lista y por tanto de la cola (el primero en ser retirado). La referencia `fin` apunta al último elemento de la lista y también de la cola.

Declaración de `nodo` y `cola`

Se utilizan las clases `Nodo` y `ColaLista`. El `Nodo` representa al elemento y al enlace con el siguiente nodo; al crear un `Nodo` se asigna el elemento y el enlace se pone `null`. Con el objetivo de generalizar, el elemento se declara de tipo `Object`.

La clase `ColaLista` define las variables (atributos) de acceso: `frente` y `fin`, y las operaciones básicas del *TAD cola*. El constructor de `ColaLista` inicializa `frente` y `fin` a `null`, es decir, a la condición *cola vacía*.

```
package TipoCola;
 // declaración de Nodo (sólo visible en este paquete)
class Nodo
{
  Object elemento;
  Nodo siguiente;
  public Nodo(Object x)
  {
    elemento = x;
    siguiente = null;
  }
}
```

```
  // declaración de la clase ColaLista
public class ColaLista
{
  protected Nodo frente;
  protected Nodo fin;
      // constructor: crea cola vacía
  public ColaLista( )
  {
    frente = fin = null;
  }
    // insertar: pone elemento por el final
  public void insertar(Object elemento)
  {
    Nodo a;
    a = new Nodo(elemento);
    if (colaVacia( ))
    {
      frente = a;
    }
    else
    {
      fin.siguiente = a;
    }
    fin = a;
  }
   // quitar: sale el elemento frente
  public Object quitar( )throws Exception
  {
    Object aux;
    if (!colaVacia( ))
    {
      aux = frente.elemento;
      frente = frente.siguiente;
    }
    else
      throw new Exception("Eliminar de una cola vacía");
    return aux;
  }
    // libera todos los nodos de la cola
  public void borrarCola( )
  {
    Nodo T;
    for (; frente != null;)
    {
      T = frente;
      frente = frente.siguiente; t.siguiente = null;
    }
    System.gc( );
  }
   // acceso al primero de la cola
  public Object frenteCola( ) throws Exception
  {
    if (colaVacia( ))
      throw new Exception("Error: cola vacía");
    return (frente.elemento);
  }
   // verificación del estado de la cola
  public boolean colaVacia( )
  {
    return (frente == null);
  }
}
```

Resumen

- Una **lista enlazada** es una estructura de datos dinámica cuyos componentes están ordenados lógicamente por sus campos de enlace en vez de ordenados físicamente como están en un arreglo. El final de la lista se señala mediante una constante o referencia especial llamada `null`.
- Una **lista simplemente enlazada** contiene sólo un enlace a un sucesor único, a menos que sea el último, en cuyo caso no se enlaza con ningún otro nodo.
- Cuando se inserta un elemento en una lista enlazada, se deben considerar cuatro casos: añadir a una lista vacía, añadir al principio de la lista, añadir en el interior y añadir al final de la lista.
- Para borrar un elemento, primero hay que buscar el nodo que lo contiene y considerar dos casos: borrar el primer nodo y borrar cualquier otro nodo de la lista.
- El recorrido de una lista enlazada significa pasar por cada nodo (visitar) y procesarlo. El proceso de cada nodo puede consistir en escribir su contenido, modificar el campo dato, etcétera.
- Una **lista doblemente enlazada** es aquella en la que cada nodo tiene una referencia a su sucesor y otra a su predecesor. Las listas doblemente enlazadas se pueden recorrer en ambos sentidos. Las operaciones básicas son inserción, borrado y recorrer la lista; similares a las listas simples.
- Una **lista enlazada circularmente** por propia naturaleza no tiene primero ni último nodo. Las listas circulares pueden ser de enlace simple o doble.
- Una *pila* es una estructura de datos tipo **LIFO** (*last-in/first-out*, último en entrar/primero en salir) en la que los datos (todos del mismo tipo) se añaden y eliminan por el mismo extremo, denominado *cima* de la pila.
- Se definen las siguientes operaciones básicas sobre pilas: `crear`, `insertar`, `cima`, `quitar`, `pilaVacia`, `pilaLlena` y `limpiarPila`.
- `crear` inicializa la pila como pila vacía. Esta operación la implementa el constructor.
- `insertar` añade un elemento en la cima de la pila. Debe haber espacio en la pila.
- `cima` devuelve el elemento que está en la cima, sin extraerlo.
- `quitar` extrae de la pila el elemento cima de la pila.
- `pilaVacia` determina si el estado de la pila es vacía; en su caso devuelve el valor lógico *true*.
- `pilaLlena` determina si existe espacio en la pila para añadir un nuevo elemento. De no haber espacio devuelve `true`. Esta operación se aplica en la representación de la pila mediante arreglo.
- `limpiarPila`, el espacio asignado a la pila se libera, queda disponible.
- El TAD `Pila` se puede implementar con arreglos y con listas enlazadas.
- La realización de una pila con listas enlazadas ajusta el tamaño de la estructura al número de elementos de la pila, es una estructura dinámica.
- Una cola es una lista lineal en la que los datos se insertan por un extremo (*final*) y se extraen por el otro (*frente*). Es una *estructura* **FIFO** (*first-in/first-out*, primero en entrar/primero en salir).
- Las operaciones básicas que se aplican sobre colas son: `crearCola`, `colaVacia`, `colaLlena`, `insertar`, `frente`, `retirar`.
- `crearCola` inicializa a una cola sin elementos. Es la primera operación a realizar con una cola.
- `colaVacia` determina si una cola tiene o no elementos. Devuelve `true` si no tiene elementos.
- `colaLlena` determina si no se pueden almacenar más elementos en una cola. Se aplica esta operación cuando se utiliza un arreglo para guardar los elementos de la cola.
- `insertar` añade un nuevo elemento a la cola, siempre por el extremo final.
- `frente` devuelve el elemento que está en el extremo frente de la cola, sin extraerlo.
- `retirar` extrae el elemento frente de la cola.
- El TAD cola se puede implementar con arreglos y con listas enlazadas. La implementación con un arreglo lineal es muy ineficiente; se ha de considerar el arreglo como una estructura circular y aplicar la teoría de los restos para avanzar el frente y el final de la cola.
- La realización de una cola con listas enlazadas permite que el tamaño de la estructura se ajuste al número de elementos. La cola puede crecer indefinidamente, con el único tope de la memoria libre.

- Numerosos modelos de sistemas del mundo real son de tipo cola: cola de impresión en un servidor de impresoras, programas de simulación, colas de prioridades en organización de viajes. Una cola es la estructura típica que se suele utilizar como almacenamiento de datos, cuando se envían datos desde un componente rápido de una computadora a un componente lento (por ejemplo, una impresora).

Ejercicios

36.1. Escribir un método, en la clase `Lista`, que devuelva cierto si la lista está vacía.

36.2. Añadir a la clase `ListaDoble` un método que devuelva el número de nodos de una lista doble.

36.3. A la clase que representa una cola implementada con un arreglo circular y dos variables `frente` y `fin`, se le añade una variable más que guarda el número de elementos de la cola. Escribir de nuevo los métodos de manejo de colas considerando este campo contador.

36.4. A la clase `Lista` añadir el método `eliminaraPosicion( )` que elimine el nodo que ocupa la posición `i`, siendo el nodo cabecera el que ocupa la posición `0`.

36.5. ¿Cuál es la salida de este segmento de código, teniendo en cuenta que el tipo de dato de la pila es `int`:

```
Pila p = new Pila( );
int x = 4, y;
p.insertar(x);
System.out.println("\n " + p.cimaPila( ));
y = p.quitar( );
p.insertar(32);
p.insertar(p.quitar( ));
do {
  System.out.println("\n " + p.quitar( ));
}while (!p.pilaVacia( ));
```

36.6. Se tiene una lista simplemente enlazada de números reales. Escribir un método para obtener una lista doble ordenada respecto al campo dato, con los valores reales de la lista simple.

36.7. Se tiene una lista enlazada a la que se accede por el primer nodo. Escribir un método que imprima los nodos de la lista en orden inverso, desde el último nodo al primero; como estructura auxiliar utilizar una pila y sus operaciones.

36.8. Supóngase que se tienen ya codificados los métodos que implementan las operaciones del *TAD Cola*. Escribir un método para crear una copia de una cola determinada. Las operaciones que se han de utilizar serán únicamente las del *TAD cola*.

36.9. Dada una lista circular de números enteros, escribir un método que devuelva el mayor entero de la lista.

36.10. Se tiene una lista de simple enlace, el campo dato es un objeto alumno con las variables: *nombre, edad, sexo*. Escribir un método para transformar la lista de tal forma que si el primer nodo es de un alumno de sexo masculino, el siguiente sea de sexo femenino, así alternativamente, siempre que sea posible, masculino y femenino.

36.11. Una lista circular de cadenas está ordenada alfabéticamente. El pointer de acceso a la lista tiene la dirección del nodo alfabéticamente mayor. Escribir un método para añadir una nueva palabra, en el orden que le corresponda, a la lista.

36.12. Dada la lista del ejercicio 36.11 escribir un método que elimine una palabra dada.

Problemas

36.1. Escribir un programa que realice las siguientes tareas:

- Crear una lista enlazada de números enteros positivos al azar, la inserción se realiza por el último nodo.
- Recorrer la lista para mostrar los elementos por pantalla.
- Eliminar todos los nodos que superen un valor dado.

36.2. Se tiene un archivo de texto de palabras separadas por un blanco o el carácter de fin de línea. Escribir un programa para formar una lista enlazada con las palabras del archivo. Una vez formada la lista se pueden añadir nuevas palabras o borrar alguna de ellas. Al finalizar el programa escribir las palabras de la lista en el archivo.

36.3. Un polinomio se puede representar como una lista enlazada. El primer nodo de la lista representa el primer término del polinomio; el segundo nodo al segundo término del polinomio y así sucesivamente. Cada nodo tiene como campo dato el coeficiente del término y el exponente.

Por ejemplo, el polinomio $3x^4 - 4x^2 + 11$ se representa

Escribir un programa que permita dar entrada a polinomios en *x*, representándolos con una lista enlazada simple. A continuación, obtener una tabla de valores del polinomio para valores de $x = 0.0, 0.5, 1.0, 1.5, \ldots, 5.0$.

36.4. Teniendo en cuenta la representación de un polinomio propuesta en el problema 36.3, hacer los cambios necesarios para que la lista enlazada sea circular. La referencia de acceso debe tener la dirección del último término del polinomio, el cual apuntará al primer término.

36.5. Según la representación de un polinomio propuesta en el problema 36.4, escribir un programa para realizar las siguientes operaciones:

- Obtener la lista circular suma de dos polinomios.
- Obtener el polinomio derivada.
- Obtener una lista circular que sea el producto de dos polinomios.

36.6. Se tiene un archivo de texto del cual se quieren determinar las frases que son palíndromo, para lo cual se ha de seguir la siguiente estrategia:

- Considerar cada línea del texto una frase.
- Añadir cada carácter de la frase a una pila y a la vez a una cola.
- Extraer carácter a carácter, y simultáneamente de la pila y de la cola. Su comparación determina si es palíndromo o no.
- Escribir un programa que lea cada línea del archivo y determine si es palíndromo.

36.7. Un conjunto es una secuencia de elementos todos del mismo tipo sin duplicidades. Escribir un programa para representar un conjunto de enteros mediante una lista enlazada. El programa debe contemplar las operaciones:

- Cardinal del conjunto.
- Pertenencia de un elemento al conjunto.
- Añadir un elemento al conjunto.
- Escribir en pantalla los elementos del conjunto.

36.8. Con la representación propuesta en el problema 36.7, añadir las operaciones básicas de conjuntos:

- Unión de dos conjuntos.
- Intersección de dos conjuntos.
- Diferencia de dos conjuntos.
- Inclusión de un conjunto en otro.

36.9. Escribir un método para determinar si una secuencia de caracteres de entrada es de la forma:

```
X & Y
```

siendo `X` una cadena de caracteres y `Y` la cadena inversa. El carácter `&` es el separador.

36.10. Utilizar una lista doblemente enlazada para controlar una lista de pasajeros de una línea aérea. El programa principal debe ser controlado por menús y debe permitir al usuario visualizar los datos de un pasajero determinado, insertar un nodo (siempre por el final), eliminar un pasajero de la lista. A la lista se accede por dos variables, una referencia al primer nodo y la otra al último nodo.

36.11. Escribir un programa que haciendo uso de una pila, procese cada uno de los caracteres de una expresión que viene dada en una línea. La finalidad es verificar el equilibrio de paréntesis, llaves y corchetes. Por ejemplo, la siguiente expresión tiene un número de paréntesis equilibrado:

```
((a+b)*5) - 7
```

A esta otra expresión le falta un corchete: `2*[(a+b)/2.5 + x - 7*y`

36.12. Un vector disperso es aquel que tiene muchos elementos que son cero. Escribir un programa que permita representar mediante listas enlazadas un vector disperso. Los nodos de la lista son los elementos de la lista distintos de cero; en cada nodo se representa el valor del elemento y el índice (posición del vector). El programa ha de realizar las operaciones: sumar dos vectores de igual dimensión y hallar el producto escalar.