Capítulo 5 Listas doblemente enlazadas:

5.1 Definición

Una lista doblemente enlazada es una lista lineal en la que cada nodo tiene dos enlaces, uno al nodo siguiente, y otro al anterior.

Las listas doblemente enlazadas no necesitan un nodo especial para acceder a ellas, pueden recorrerse en ambos sentidos a partir de cualquier nodo, esto es porque a partir de cualquier nodo, siempre es posible alcanzar cualquier nodo de la lista, hasta que se llega a uno de los extremos.

El nodo típico es el mismo que para construir las listas que hemos visto, salvo que tienen otro puntero al nodo anterior:

```
struct nodo {
   int dato;
   struct nodo *siguiente;
   struct nodo *anterior;
};
```

5.2 Declaraciones de tipos para manejar listas doblemente enlazadas en C:

Para C, y basándonos en la declaración de nodo que hemos visto más arriba, trabajaremos con los siguientes tipos:

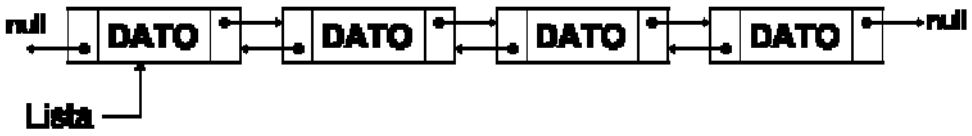
```
typedef struct _nodo {
   int dato;
   struct _nodo *siguiente;
   struct _nodo *anterior;
} tipoNodo;

typedef tipoNodo *pNodo;
typedef tipoNodo *Lista;
```

tipoNodo es el tipo para declarar nodos, evidentemente.

pNodo es el tipo para declarar punteros a un nodo.

Lista es el tipo para declarar listas abiertas doblemente enlazadas. También es posible, y potencialmente útil, crear listas doblemente enlazadas y circulares.



El movimiento a través de listas doblemente enlazadas es más sencillo, y como veremos las operaciones de búsqueda, inserción y borrado, también tienen más ventajas.

5.3 Operaciones básicas con listas doblemente enlazadas:

De nuevo tenemos el mismo repertorio de operaciones sobre este tipo listas:

- Añadir o insertar elementos.
- Buscar o localizar elementos.
- Borrar elementos.
- Moverse a través de la lista, siguiente y anterior.

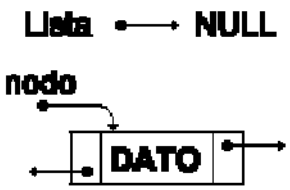
5.4 Añadir un elemento:

Nos encontramos ahora ante un tipo de estructura algo diferente de las que hemos estado viendo, así que entraremos en más detalles.

Vamos a intentar ver todos los casos posibles de inserción de elementos en listas doblemente enlazadas.

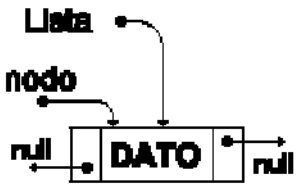
Añadir elemento en una lista doblemente enlazada vacía:

Partiremos de que ya tenemos el nodo a insertar y, por supuesto un puntero que apunte a él, además el puntero que define la lista, que valdrá NULL:



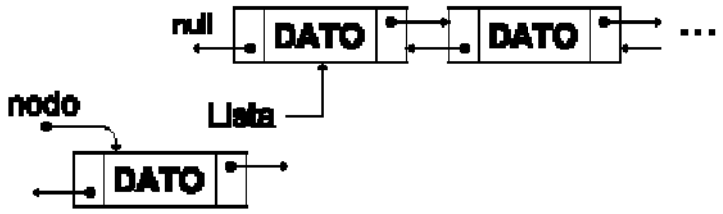
El proceso es muy simple, bastará con que:

1. lista apunta a nodo.
2. lista->siguiente y lista->anterior apunten a null.



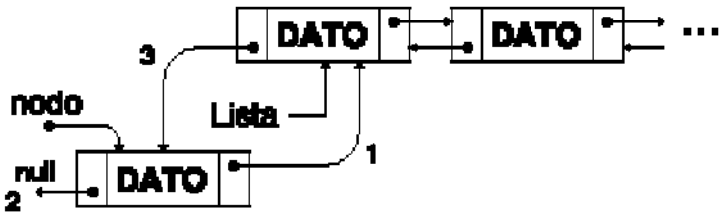
Insertar un elemento en la primera posición de la lista:

Partimos de una lista no vacía. Para simplificar, consideraremos que lista apunta al primer elemento de la lista doblemente enlazada:



El proceso es el siguiente:

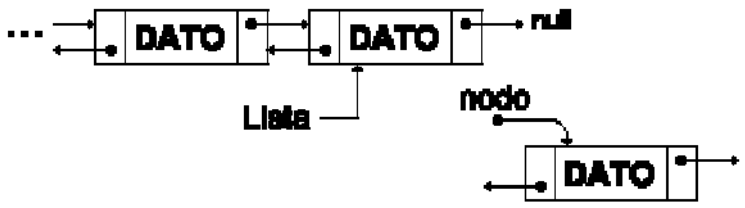
1. nodo->siguiente debe apuntar a Lista.
2. nodo->anterior apuntará a Lista->anterior.
3. Lista->anterior debe apuntar a nodo.



Recuerda que Lista no tiene por qué apuntar a ningún miembro concreto de una lista doblemente enlazada, cualquier miembro es igualmente válido como referencia.

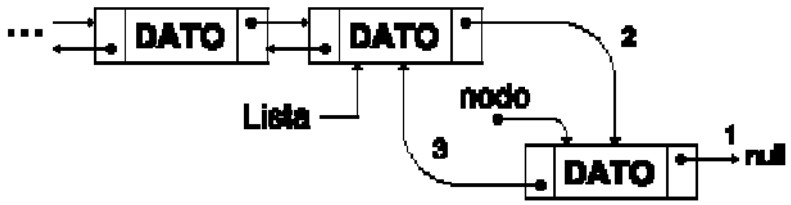
Insertar un elemento en la última posición de la lista:

Igual que en el caso anterior, partiremos de una lista no vacía, y de nuevo para simplificar, que Lista está apuntando al último elemento de la lista:



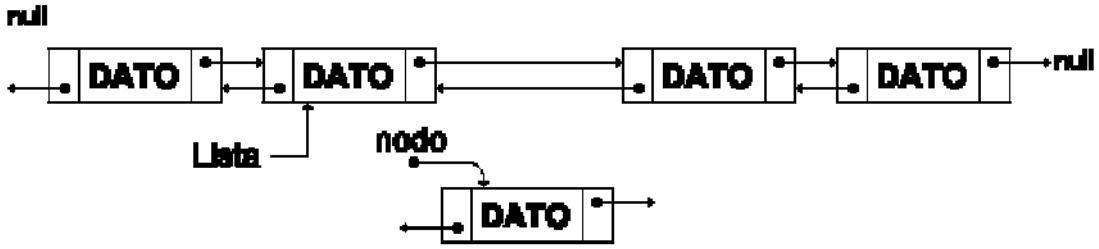
El proceso es el siguiente:

1. nodo->siguiente debe apuntar a Lista->siguiente (NULL).
2. Lista->siguiente debe apuntar a nodo.
3. nodo->anterior apuntará a Lista.



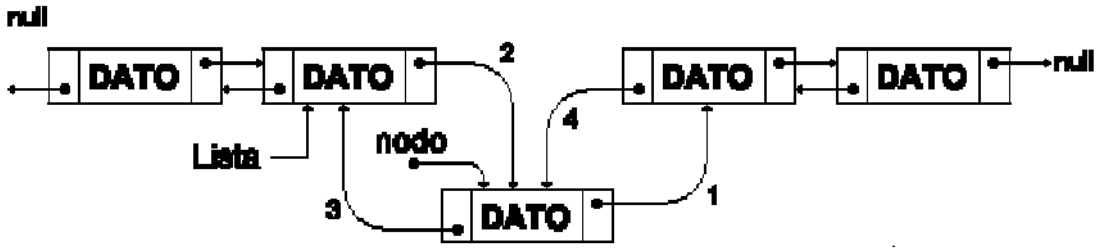
Insertar un elemento a continuación de un nodo cualquiera de una lista:

Bien, este caso es más genérico, ahora partimos de una lista no vacía, e insertaremos un nodo a continuación de uno nodo cualquiera que no sea el último de la lista:



El proceso sigue siendo muy sencillo:

1. Hacemos que nodo->siguiente apunte a lista->siguiente.
2. Hacemos que Lista->siguiente apunte a nodo.
3. Hacemos que nodo->anterior apunte a lista.
4. Hacemos que nodo->siguiente->anterior apunte a nodo.



Lo que hemos hecho es trabajar como si tuviéramos dos listas enlazadas, los dos primeros pasos equivalen a lo que hacíamos para insertar elementos en una lista abierta corriente.

Los dos siguientes pasos hacen lo mismo con la lista que enlaza los nodos en sentido contrario.

El paso 4 es el más oscuro, quizás requiera alguna explicación.

Supongamos que disponemos de un puntero auxiliar, "p" y que antes de empezar a insertar nodo, hacemos que apunte al nodo que quedará a continuación de nodo después de insertarlo, es decir p = Lista->siguiente.

Ahora empezamos el proceso de inserción, ejecutamos los pasos 1, 2 y 3. El cuarto sería sólo hacer que p->anterior apunte a nodo. Pero nodo->siguiente ya apunta a p, así que en realidad no necesitamos el puntero auxiliar, bastará con hacer que nodo->siguiente->anterior apunte a nodo.

Añadir elemento en una lista doblemente enlazada, caso general:

Para generalizar todos los casos anteriores, sólo necesitamos añadir una operación:

1. Si lista está vacía hacemos que Lista apunte a nodo. Y nodo->anterior y nodo->siguiente a NULL.
2. Si lista no está vacía, hacemos que nodo->siguiente apunte a Lista->siguiente.
3. Después que Lista->siguiente apunte a nodo.
4. Hacemos que nodo->anterior apunte a Lista.
5. Si nodo->siguiente no es NULL, entonces hacemos que nodo->siguiente->anterior apunte a nodo.

El paso 1 es equivalente a insertar un nodo en una lista vacía.

Los pasos 2 y 3 equivalen a la inserción en una lista enlazada corriente.

Los pasos 4, 5 equivalen a insertar en una lista que recorre los nodos en sentido contrario.

Existen más casos, las listas doblemente enlazadas son mucho más versátiles, pero todos los casos pueden reducirse a uno de los que hemos explicado aquí.

5.5 Buscar o localizar un elemento de una lista doblemente enlazada:

En muchos aspectos, una lista doblemente enlazada se comporta como dos listas abiertas que comparten los datos. En ese sentido, todo lo dicho en el capítulo sobre la localización de elementos en listas abiertas se puede aplicar a listas doblemente enlazadas.

Pero además tenemos la ventaja de que podemos avanzar y retroceder desde cualquier nodo, sin necesidad de volver a uno de los extremos de la lista.

Por supuesto, se pueden hacer listas doblemente enlazadas no ordenadas, existen cientos de problemas que pueden requerir de este tipo de estructuras. Pero parece que la aplicación más sencilla de listas doblemente enlazadas es hacer arrays dinámicos ordenados, donde buscar un elemento concreto a partir de cualquier otro es más sencillo que en una lista abierta corriente.

Pero de todos modos veamos algún ejemplo sencillo.

Para recorrer una lista procederemos de un modo parecido al que usábamos con las listas abiertas, ahora no necesitamos un puntero auxiliar, pero tenemos que tener en cuenta que Lista no tiene por qué estar en uno de los extremos:

1. Retrocedemos hasta el comienzo de la lista, asignamos a lista el valor de lista->anterior mientras lista->anterior no sea NULL.
2. Abriremos un bucle que al menos debe tener una condición, que el índice no sea NULL.
3. Dentro del bucle asignaremos a lista el valor del nodo siguiente al actual.

Por ejemplo, para mostrar todos los valores de los nodos de una lista, podemos usar el siguiente bucle en C:

```
typedef struct _nodo {
  int dato;
  struct _nodo *siguiente;
  struct _nodo *anterior;
  } tipoNodo;

  typedef tipoNodo *pNodo;
  typedef tipoNodo *Lista;
  ...
  pNodo = indice;
  ...
  indice = Lista;
  while(indice->anterior) indice = indice->anterior;
  while(indice) {
     printf("%d\n", indice->dato);
     indice = indice->siguiente;
  }
  ...
```

Es importante que no perdamos el nodo Lista, si por error le asignáramos un valor de un puntero a un nodo que no esté en la lista, no podríamos acceder de nuevo a ella.

Es por eso que tendremos especial cuidado en no asignar el valor NULL a Lista.

5.6 Eliminar un elemento de una lista doblemente enlazada:

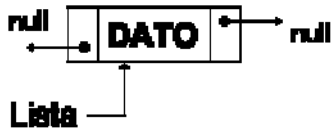
Analizaremos tres casos diferentes:

1. Eliminar el único nodo de una lista doblemente enlazada.
2. Eliminar el primer nodo.
3. Eliminar el último nodo.
4. Eliminar un nodo intermedio.

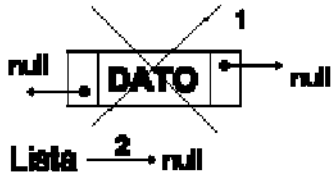
Para los casos que lo permitan consideraremos dos casos: que el nodo a eliminar es el actualmente apuntado por Lista o que no.

Eliminar el único nodo en una lista doblemente enlazada:

En este caso, ese nodo será el apuntado por Lista.

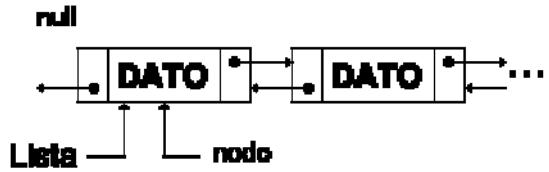


1. Eliminamos el nodo.
2. Hacemos que Lista apunte a NULL.

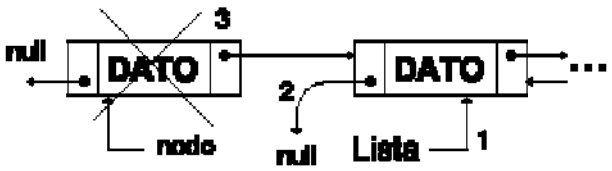


Eliminar el primer nodo de una lista doblemente enlazada:

Tenemos los dos casos posibles, que el nodo a borrar esté apuntado por Lista o que no. Si lo está, simplemente hacemos que Lista sea Lista->siguiente.



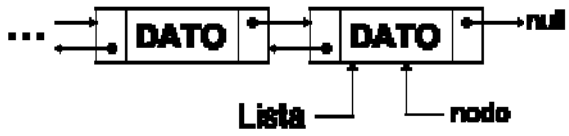
1. Si nodo apunta a Lista, hacemos que Lista apunte a Lista->siguiente.
2. Hacemos que nodo->siguiente->anterior apunte a NULL
3. Borramos el nodo apuntado por nodo.



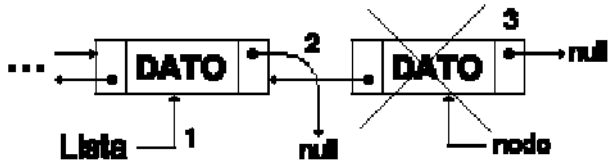
El paso 2 depara el nodo a borrar del resto de la lista, independientemente del nodo al que apunte Lista.

Eliminar el último nodo de una lista doblemente enlazada:

De nuevo tenemos los dos casos posibles, que el nodo a borrar esté apuntado por Lista o que no. Si lo está, simplemente hacemos que Lista sea Lista->anterior.



1. Si nodo apunta a Lista, hacemos que Lista apunte a Lista->anterior.
2. Hacemos que nodo->anterior->siguiente apunte a NULL
3. Borramos el nodo apuntado por nodo.

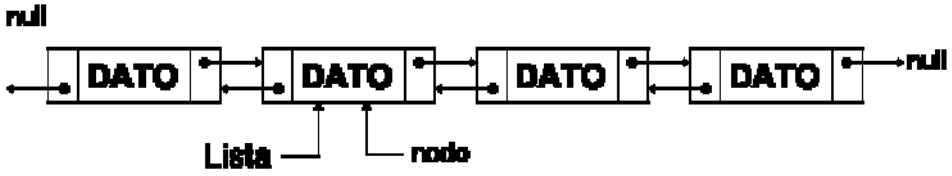


El paso 2 depara el nodo a borrar del resto de la lista, independientemente del nodo al que apunte Lista.

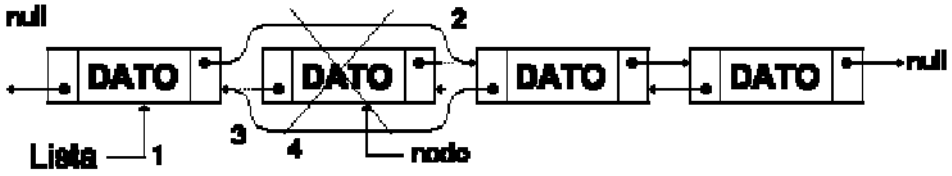
Eliminar un nodo intermedio de una lista doblemente enlazada:

De nuevo tenemos los dos casos posibles, que el nodo a borrar esté apuntado por Lista o que no. Si lo está, simplemente hacemos que Lista sea Lista->anterior o Lista->siguiente

Se trata de un caso más general de los dos casos anteriores..



1. Si nodo apunta a Lista, hacemos que Lista apunte a Lista->anterior (o Lista->siguiente).
2. Hacemos que nodo->anterior->siguiente apunte a nodo->siguiente.
3. Hacemos que nodo->siguiente->anterior apunte a nodo->anterior.
4. Borramos el nodo apuntado por nodo.



Eliminar un nodo de una lista doblemente enlazada, caso general:

De nuevo tenemos los dos casos posibles, que el nodo a borrar esté apuntado por Lista o que no. Si lo está, simplemente hacemos que Lista sea Lista->anterior, si no es NULL o Lista->siguiente en caso contrario.

1. Si nodo apunta a Lista,
    - Si Lista->anterior no es NULL hacemos que Lista apunte a Lista->anterior.
    - Si Lista->siguiente no es NULL hacemos que Lista apunte a Lista->siguiente.
    - Si ambos son NULL, hacemos que Lista sea NULL.
2. Si nodo->anterior no es NULL, hacemos que nodo->anterior->siguiente apunte a nodo->siguiente.
3. Si nodo->siguiente no es NULL, hacemos que nodo->siguiente->anterior apunte a nodo->anterior.

Borramos el nodo apuntado por nodo.

5.7 Ejemplo de lista doblemente enlazada en C:

Como en el caso de los ejemplos anteriores, construiremos una lista doblemente enlazada para almacenar números enteros. Para aprovechar mejor las posibilidades de estas listas, haremos que la lista esté ordenada. Haremos pruebas insertando varios valores, buscándolos y eliminándolos alternativamente para comprobar el resultado.

Algoritmo de inserción:

1. El primer paso es crear un nodo para el dato que vamos a insertar.
2. Si Lista está vacía, o el valor del primer elemento de la lista es mayor que el del nuevo, insertaremos el nuevo nodo en la primera posición de la lista.
3. En caso contrario, buscaremos el lugar adecuado para la inserción, tenemos un puntero "anterior". Lo inicializamos con el valor de Lista, y avanzaremos mientras anterior->siguiente no sea NULL y el dato que contiene anterior->siguiente sea menor o igual que el dato que queremos insertar.
4. Ahora ya tenemos anterior señalando al nodo adecuado, así que insertamos el nuevo nodo a continuación de él.

```
void Insertar(Lista *lista, int v)
{
   pNodo nuevo, actual;

   /* Crear un nodo nuevo */
   nuevo = (pNodo)malloc(sizeof(tipoNodo));
   nuevo->valor = v;
```

```
   /* Colocamos actual en la primera posición de la lista */
   actual = *lista;
   if(actual) while(actual->anterior) actual = actual->anterior;

   /* Si la lista está vacía o el primer miembro es mayor que el
nuevo */
   if(!actual || actual->valor > v) {
      /* Añadimos la lista a continuación del nuevo nodo */
      nuevo->siguiente = actual;
      nuevo->anterior = NULL;
      if(actual) actual->anterior = nuevo;
      if(!*lista) *lista = nuevo;
   }
   else {
      /* Avanzamos hasta el último elemento o hasta que el
         siguiente tenga un valor mayor que v */
      while(actual->siguiente &&actual->siguiente->valor <= v)
         actual = actual->siguiente;
      /* Insertamos el nuevo nodo después del nodo anterior */
      nuevo->siguiente = actual->siguiente;
      actual->siguiente = nuevo;
      nuevo->anterior = actual;
      if(nuevo->siguiente) nuevo->siguiente->anterior = nuevo;
   }
}
```

Algoritmo de la función "Borrar":

1. Localizamos el nodo de valor v
2. ¿Existe?
   - **SI**:
     - ¿Es el nodo apuntado por lista?
       - **SI**: Hacer que lista apunte a otro sitio.
     - ¿Es el primer nodo de la lista?
       - **NO**: nodo->anterior->siguiente = nodo->siguiente
     - ¿Es el último nodo de la lista?
       - **NO**: nodo->siguiente->anterior = nodo->anterior
     - Borrar nodo

```
void Borrar(Lista *lista, int v)
{
   pNodo nodo;

   /* Buscar el nodo de valor v */
   nodo = *lista;
   while(nodo && nodo->valor <v) nodo = nodo->siguiente;
   while(nodo && nodo->valor > v) nodo = nodo->anterior;

   /* El valor v no está en la lista */
   if(!nodo || nodo->valor != v) return;

   /* Borrar el nodo */
   /* Si lista apunta al nodo que queremos borrar, apuntar a otro
*/
   if(nodo == *lista)
     if(nodo->anterior) *lista = nodo->anterior;
     else *lista = nodo->siguiente;
```

```
   if(nodo->anterior) /* no es el primer elemento */
      nodo->anterior->siguiente = nodo->siguiente;
   if(nodo->siguiente) /* no es el último nodo */
      nodo->siguiente->anterior = nodo->anterior;
   free(nodo);
}
```

Código del ejemplo completo:

Tan sólo nos queda escribir una pequeña prueba para verificar el funcionamiento:

```
#include <stdlib.h>
#include <stdio.h>

#define ASCENDENTE 1
#define DESCENDENTE 0

typedef struct _nodo {
   int valor;
   struct _nodo *siguiente;
   struct _nodo *anterior;
} tipoNodo;

typedef tipoNodo *pNodo;
typedef tipoNodo *Lista;

/* Funciones con listas: */
void Insertar(Lista *l, int v);
void Borrar(Lista *l, int v);

void BorrarLista(Lista *);
void MostrarLista(Lista l, int orden);

int main()
{
   Lista lista = NULL;
   pNodo p;

   Insertar(&lista, 20);
   Insertar(&lista, 10);
   Insertar(&lista, 40);
   Insertar(&lista, 30);

   MostrarLista(lista, ASCENDENTE);
   MostrarLista(lista, DESCENDENTE);

   Borrar(&lista, 10);
   Borrar(&lista, 15);
   Borrar(&lista, 45);
   Borrar(&lista, 30);

   MostrarLista(lista, ASCENDENTE);
   MostrarLista(lista, DESCENDENTE);

   BorrarLista(&lista);

   system("PAUSE");
   return 0;
}
```

```
void Insertar(Lista *lista, int v)
{
   pNodo nuevo, actual;

   /* Crear un nodo nuevo */
   nuevo = (pNodo)malloc(sizeof(tipoNodo));
   nuevo->valor = v;

   /* Colocamos actual en la primera posición de la lista */
   actual = *lista;
   if(actual) while(actual->anterior) actual = actual->anterior;
   /* Si la lista está vacía o el primer miembro es mayor que el
      nuevo */
   if(!actual || actual->valor > v) {
      /* Añadimos la lista a continuación del nuevo nodo */
      nuevo->siguiente = actual;
      nuevo->anterior = NULL;
      if(actual) actual->anterior = nuevo;
      if(!*lista) *lista = nuevo;
   }
   else {
      /* Avanzamos hasta el último elemento o hasta que el
         siguiente tenga un valor mayor que v */
      while(actual->siguiente &&actual->siguiente->valor <= v)
         actual = actual->siguiente;
      /* Insertamos el nuevo nodo después del nodo anterior */
      nuevo->siguiente = actual->siguiente;
      actual->siguiente = nuevo;
      nuevo->anterior = actual;
      if(nuevo->siguiente) nuevo->siguiente->anterior = nuevo;
   }
}

void Borrar(Lista *lista, int v)
{
   pNodo nodo;

   /* Buscar el nodo de valor v */
   nodo = *lista;
   while(nodo && nodo->valor < v) nodo = nodo->siguiente;
   while(nodo && nodo->valor > v) nodo = nodo->anterior;

   /* El valor v no está en la lista */
   if(!nodo || nodo->valor != v) return;

   /* Borrar el nodo */
   /* Si lista apunta al nodo que queremos borrar, apuntarlo
      a otro */
   if(nodo == *lista)
     if(nodo->anterior) *lista = nodo->anterior;
     else *lista = nodo->siguiente;

   if(nodo->anterior) /* no es el primer elemento */
      nodo->anterior->siguiente = nodo->siguiente;
   if(nodo->siguiente) /* no es el último nodo */
      nodo->siguiente->anterior = nodo->anterior;
   free(nodo);
}

void BorrarLista(Lista *lista)
```

```
{
   pNodo nodo, actual;

   actual = *lista;
   while(actual->anterior) actual = actual->anterior;

   while(actual) {
      nodo = actual;
      actual = actual->siguiente;
      free(nodo);
   }
   *lista = NULL;
}

void MostrarLista(Lista lista, int orden)
{
   pNodo nodo = lista;

   if(!lista) printf("Lista vacía");

   nodo = lista;
   if(orden == ASCENDENTE) {
      while(nodo->anterior) nodo = nodo->anterior;
      printf("Orden ascendente: ");
      while(nodo) {
         printf("%d -> ", nodo->valor);
         nodo = nodo->siguiente;
      }
   }
   else {
      while(nodo->siguiente) nodo = nodo->siguiente;
      printf("Orden descendente: ");
      while(nodo) {
         printf("%d -> ", nodo->valor);
         nodo = nodo->anterior;
      }
   }

   printf("\n");
}
```