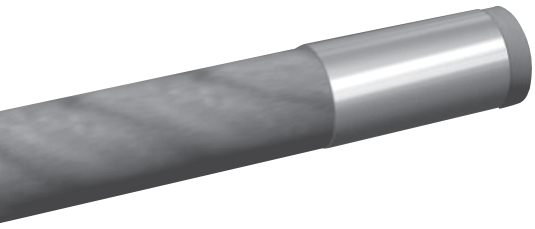
Capítulo 34

Listas, pilas y colas en C++



Contenido

Introducción

Este capítulo estudia tres tipos abstractos de datos fundamentales, las listas enlazadas, las pilas y las colas. Cada una de ellas se implementa en C++ utilizando clases. La estructura lista enlazada (ligada o encadenada, "*linked list*") 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 define una clase que agrupa la representación de una lista y las operaciones que se apli-

can: insertar, buscar, borrar elementos y recorrer la lista. De igual modo se muestra el tipo abstracto de datos (*TAD*) que representa a las listas enlazadas.

La *Pila* es una estructura de datos que almacena y recupera sus elementos atendiendo a un estricto orden; las pilas se conocen también como estructuras **LIFO** (*last-in/first-out*, último en entrar/primero en salir). Las pilas se utilizan en compiladores, sistemas operativos y programas de aplicaciones.

El tipo abstracto de datos *cola* almacena y recupera sus elementos atendiendo a un estricto orden. Las colas se conocen como estructuras **FIFO** (*first-in/first-out*, primero en entrar/primero en salir), debido a la forma y orden de inserción y de extracción 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

- Cola
- Enlace
- Estructura enlazada
- Lista
- Lista doblemente enlazada
- Nodo
- Pila
- `template`
- Tipo abstracto de datos

Tipo abstracto de datos *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. Cada elemento es un *nodo* de la lista.

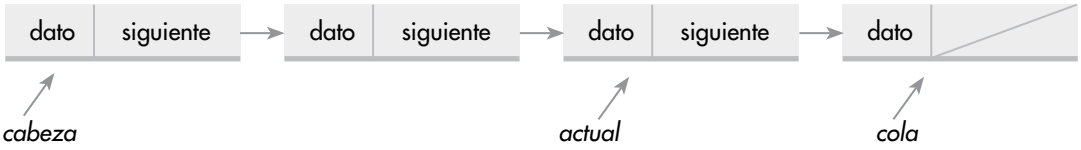


Figura 34.1 Representación gráfica de una lista enlazada.

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.

Las inserciones se pueden realizar por cualquier punto de la lista. Por la cabeza (inicio), por el final (cola), a partir o antes de un nodo determinado de la lista. Las eliminaciones también se pueden realizar en cualquier punto de la lista; además, se eliminan nodos dependiendo del campo de información o dato que se desea suprimir de la lista.

Especificación formal del *TAD lista*

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 que están 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 dato `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)`	Retorna la posición/dirección del primer nodo de la lista `L`.
`Anula(L)`	Vacía la lista `L`.

Estas operaciones son las 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. También dependerá del tipo de representación elegido para las listas. Así, para añadir nuevos nodos a una lista se implementan, además de `insertar( )`, versiones de ésta como:

`insertarPrimero(L,x)`	Inserta un nodo con el dato `x` como primer nodo de la lista `L`.
`insertarFinal(L,x)`	Inserta un nodo con el dato `x` como último nodo de la lista `L`.

Otra operación típica de toda *estructura enlazada* de datos es *recorrer*. Consiste en visitar cada uno de los datos o nodos de que consta. En las listas enlazadas, normalmente se realiza desde el nodo *cabeza* al último nodo o *cola* de la lista.

Operaciones de listas enlazadas, clase `lista`

La implementación del *TAD lista* requiere, en primer lugar, declarar la clase `Nodo` en la cual se *encierra* el dato (entero, real, doble, carácter, o referencias a objetos) y el *enlace*. Además, la clase *Lista* con las operaciones (*creación*, *inserción*,...) y el atributo *cabeza* de la lista.

Clase `Nodo`

Una lista enlazada se compone de una serie de nodos enlazados mediante punteros. La clase `Nodo` declara las dos partes en que se divide: *dato* y *enlace*. Además, el constructor y la interfaz básica; por ejemplo, para una lista enlazada de números enteros:

```
typedef int Dato;
  // archivo de cabecera Nodo.h
#ifndef _NODO_H
#define _NODO_H
class Nodo
{
protected:
  Dato dato;
  Nodo* enlace;
public:
  Nodo(Dato t)
  {
    dato = t;
    enlace = 0; // 0 es el puntero NULL en C++
  }
  Nodo(Dato p, Nodo* n) // crea el nodo y lo enlaza a n
  {
    dato = p;
    enlace = n;
  }
  Dato datoNodo( ) const
  {
    return dato;
  }
```

```
  Nodo* enlaceNodo( ) const
  {
    return enlace;
  }
  void ponerEnlace(Nodo* sgte)
  {
    enlace = sgte;
  }
};
#endif
```

Ejemplo 34.1

Se declara la clase `Punto` para representar un punto en el plano, con su coordenada *x* y *y*. También, se declara la clase `Nodo` con el campo dato de tipo `Punto`.

```
// archivo de cabecera punto.h

class Punto
{
    double x, y;
  public:
    Punto(double x = 0.0, double y = 0.0)
  {
    this.x = x;
    this.y = y;
  }
};
```

La declaración de la clase `Nodo` consiste, sencillamente, en asociar el nuevo tipo de dato; el resto no cambia.

```
typedef Punto Dato;
#include "Nodo.h"
```

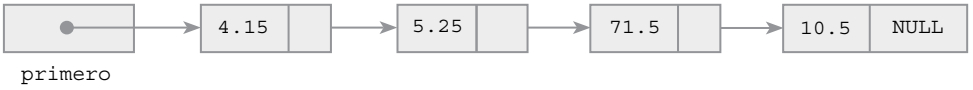


Figura 34.2 Acceso a una lista con apuntador cabeza denominado `primero`.

Clase `Lista`, construcción de una lista

La creación de una lista enlazada implica la definición de, al menos, las clases `Nodo` y `Lista`. La clase `Lista` contiene el puntero de acceso a la lista enlazada, de nombre `primero`, que apunta al nodo cabeza; también se puede declarar un puntero al nodo cola, como no se necesita para implementar las operaciones, no se ha declarado.

Las funciones miembro de la clase `Lista` implementan las operaciones de una lista enlazada: *inserción, búsqueda*... El constructor inicializa `primero` a `null`, (*lista vacía*). Además, `crearLista( )` construye iterativamente el primer elemento (`primero`) y los elementos sucesivos de una lista enlazada. La declaración de la clase es:

```
  // archivo Lista.h
class Lista
{
protected:
  Nodo* primero;
```

```
public:
  Lista( ) { primero = NULL;}
  void crearLista( );
  void insertarCabezaLista(Dato entrada);
  void insertarUltimo(Dato entrada);
  void insertarLista(Dato entrada);
  Nodo* buscarLista(Dato destino);
  void eliminar(Dato entrada)
  Nodo* ultimo( );
  void visualizar( );
};
```

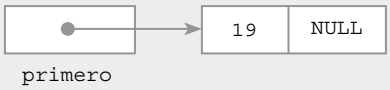
El ejemplo 34.2 declara una lista para un tipo particular de dato: `int`. Lo más interesante del ejemplo es la codificación, paso a paso, de la función `crearLista( )`.

Ejemplo 34.2

Crear una lista enlazada de elementos que almacenen datos de tipo entero.

Antes de escribir su código, se muestra el comportamiento de la función `crearLista( )` de la clase `Lista`. En primer lugar, se crea un nodo con un valor y su dirección se asigna a `primero`:

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



Ahora se desea añadir un nuevo elemento con el valor `61`, y situarlo en el primer lugar de la lista. Se utiliza el constructor de `Nodo` que enlaza con otro nodo ya creado:

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

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

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

A continuación se escribe la función `crearLista( )` que codifica las acciones descritas anteriormente. Los valores se leen del teclado; termina con el valor clave `-1`.

```
void Lista::crearLista( )
{
  int x;
  primero = NULL;
  cout << "Termina con -1" << endl;
  do {
    cin >> x;
    if (x != -1)
    {
      primero = new Nodo(x,primero);
    }
  }while (x != -1);
}
```

Inserción en una lista

Hay distintas formas de añadir un elemento, según la posición o punto de inserción. Éste puede ser:

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

Insertar en la cabeza de la lista

La posición más fácil y, a la vez, más eficiente donde insertar un nuevo elemento en una lista es por la cabeza. El proceso de inserción se resume en este algoritmo:

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

El código fuente de `insertarCabezaLista:`

```
void Lista::insertarCabezaLista(Dato entrada)
{
  Nodo* nuevo ;
  nuevo = new Nodo(entrada);
    nuevo -> ponerEnlace(primero); // enlaza nuevo con primero
  primero = nuevo; // mueve primero y apunta al nuevo nodo
}
```

Caso particular

`insertarCabezaLista` también actúa correctamente si se trata de añadir un primer nodo a una lista vacía. En este caso, `primero` apunta a `NULL` y termina apuntando al nuevo nodo de la lista enlazada.

Inserción al final de la lista

La inserción al final de la lista es menos eficiente debido a que, normalmente, no se tiene un puntero al último nodo y entonces se ha de seguir la traza desde la cabeza de la lista hasta el último nodo y, a continuación, realizar la inserción. Una vez que la variable `ultimo` apunta al final de la lista, el enlace con el nuevo nodo es sencillo:

```
ultimo -> ponerEnlace(new Nodo(entrada));
```

El campo `enlace` del último nodo queda apuntando al nodo creado y así se enlaza, como nodo final, a la lista.

A continuación, se codifica la función `insertarUltimo( )` que implementa esta forma de insertar. También la función que recorre la lista y devuelve el puntero al último nodo.

```
void Lista::insertarUltimo(Dato entrada)
{
  Nodo* ultimo = this -> ultimo( );
  ultimo -> ponerEnlace(new Nodo(entrada));
}

Nodo* Lista::ultimo( )

{
  Nodo* p = primero;
  if (p == NULL ) throw "Error, lista vacía";
  while (p -> enlaceNodo( )!= NULL) p = p -> enlaceNodo( );
  return p;
}
```

Insertar entre dos nodos de la lista

La inserción de un nodo no siempre se realiza al principio o al final de la lista; puede hacerse entre dos nodos cualesquiera. Por ejemplo, en la lista de la figura 34.3 se quiere insertar el elemento 35 entre los nodos con los datos 20 y 40.

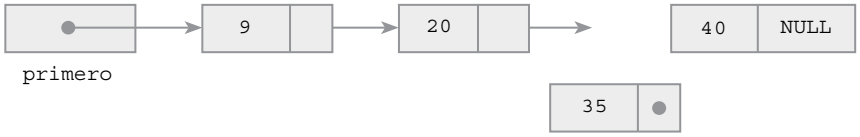


Figura 34.3 Inserción entre dos nodos.

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

1. Crear un nodo con el elemento y el campo `enlace` a `NULL`.
2. Poner campo `enlace` del nuevo nodo apuntando a `n2`, ya que el nodo creado se ubicará justo antes de `n2`.
3. Si el puntero `anterior` tiene la dirección del nodo `n1`, entonces poner su campo `enlace` apuntando al nodo creado.

La función `insertarLista( )`, miembro de la clase `Lista`, implementa la operación:

```
void Lista::insertarLista(Nodo* anterior, Dato entrada)
{
  Nodo* nuevo;
  nuevo = new Nodo(entrada);
  nuevo -> ponerEnlace(anterior -> enlaceNodo( ));
  anterior -> ponerEnlace(nuevo);
}
```

Antes de llamar a `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.

Otra versión de la función tiene como argumentos el dato a partir del cual se realiza el enlace, y el dato del nuevo nodo: `insertarLista(Dato testigo, Dato entrada)`. El algoritmo de esta versión, primero busca el nodo con el dato *testigo* a partir del cual se inserta, y a continuación realiza los mismos enlaces que en la anterior función.

Buscar un elemento y recorrer una lista enlazada

La *búsqueda* de un elemento en una lista enlazada recorre la lista hasta encontrar el nodo con el elemento. La función de búsqueda devuelve el puntero al nodo (en caso negativo, devuelva `NULL`). Otro planteamiento consiste en devolver `true` si encuentra el nodo y `false` si no está en la lista.

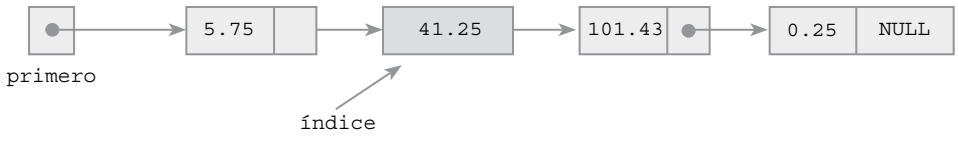


Figura 34.4 Búsqueda en una lista.

La función `buscarLista` de la clase `Lista`, utiliza el puntero `indice` para recorrer la lista, nodo a nodo. Primero, se inicializa `indice` al nodo *cabeza* (`primero`), a continuación se compara el dato del nodo apuntado 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 terminado de recorrer la lista y entonces `indice` toma el valor `NULL`.

```
Nodo* Lista:: buscarLista(Dato destino)
{
  Nodo* indice;

  for (indice = primero; indice!= NULL; indice=indice->enlaceNodo( ))
    if (destino == indice -> datoNodo( ))
      return indice;
  return NULL;
}
```

Ejemplo 34.3

Se escribe una función alternativa a la búsqueda del nodo que contiene un campo dato. Ahora también se devuelve un puntero a nodo, pero con el criterio de que ocupa una posición, pasada como argumento, en una lista enlazada.

La función es un miembro *público* de la clase `Lista`, por consiguiente tiene acceso a sus miembros. Como se busca por posición en la lista, se considera posición 1 la del nodo cabeza (`primero`); posición 2 al siguiente nodo, y así sucesivamente.

El algoritmo de búsqueda comienza inicializando `indice` al nodo cabeza de la lista. En cada iteración del bucle se mueve `indice` un nodo hacia adelante. El bucle termina cuando se alcanza la posición deseada, o bien cuando se ha recorrido toda la lista, es decir, si `indice` apunta a `NULL`.

```
Nodo* Lista::buscarPosicion(int posicion)
{
  Nodo* indice;
  int i;

  if (0 > posicion) // posición ha de ser mayor que 0
    return NULL;
  indice = primero;
  for (i = 1 ;(i < posicion) && (indice != NULL); i++)
    indice = indice -> enlaceNodo( );
  return indice;
}
```

Recorrer una estructura enlazada implica visitar cada uno de sus nodos. Para una lista, el recorrido se hace desde el primer nodo hasta el último. El fin del bucle diseñado para recorrer la lista ocurre cuando el campo enlace es `NULL`. La codificación que se escribe a continuación, escribe el dato de cada nodo de la lista.

```
void Lista::visualizar( )
{
  Nodo* n;
  int k = 0;
  n = primero;
  while (n != NULL)
  {
    char c;
    k++; c = (k%15 != 0 ? ' ' : '\n');
    cout << n -> datoNodo( ) << c;
    n = n -> enlaceNodo( );
  }
}
```

Borrado de un nodo

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 que se enfoca para eliminar un nodo que contiene un dato, 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.

La función miembro, de la clase `Lista`, `eliminar( )` implementa el algoritmo; recibe el dato del nodo que se quiere borrar, desarrolla su propio bucle de búsqueda con el fin de disponer de la dirección del nodo anterior.

```
void Lista::eliminar(Dato entrada)
{
  Nodo *actual, *anterior;
  bool encontrado;

  actual = primero;
  anterior = NULL;
  encontrado = false;
   // búsqueda del nodo y del anterior
  while ((actual != NULL) && !encontrado)
  {
    encontrado = (actual -> datoNodo( ) == entrada);
    if (!encontrado)
    {
       anterior = actual;
       actual = actual -> enlaceNodo( );
    }
  }
   // enlace del nodo anterior con el siguiente
  if (actual != NULL)
  {
      // distingue entre cabecera y resto de la lista
    if (actual == primero)
    {
      primero = actual -> enlaceNodo( );
    }
    else
    {
      anterior -> ponerEnlace(actual -> enlaceNodo( ));
    }
    delete actual;
  }
}
```

Lista doblemente enlazada

Una *lista doblemente enlazada* se caracteriza porque se puede avanzar hacia adelante, o bien hacia atrás. Cada nodo de una lista doble tiene tres campos, el dato y dos punteros; uno apunta al siguiente nodo de la lista y el otro al nodo anterior. La figura 34.5 muestra una lista doblemente enlazada y un nodo.

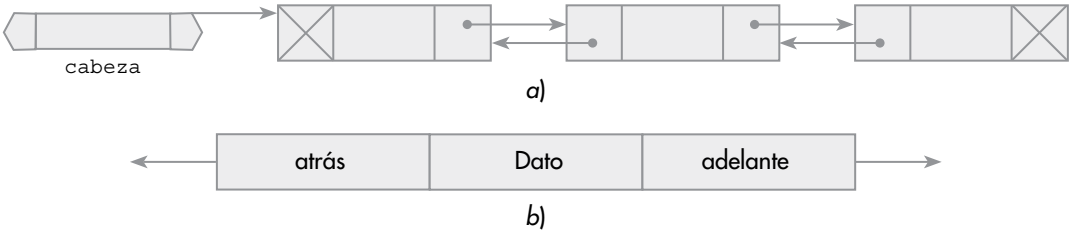


Figura 34.5 Lista doblemente enlazada. *a*) Lista con tres nodos; *b*) nodo.

Las operaciones de una *lista doble* son similares a las de una *Lista: insertar, eliminar, buscar, recorrer...* Tanto para insertar, como para borrar un nodo, se deben realizar ajustes de los dos punteros del nodo. La figura 34.6 muestra los movimientos de punteros para insertar un nodo; como se observa, intervienen cuatro enlaces.

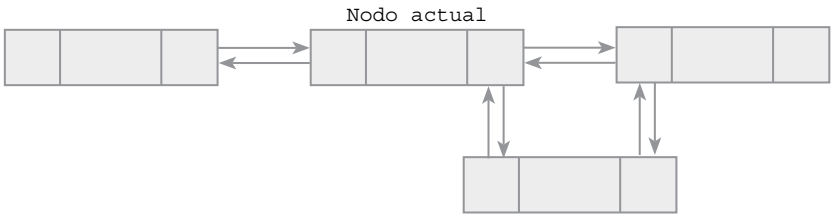


Figura 34.6 Inserción de un nodo en una lista doblemente enlazada.

La operación de eliminar un nodo de la lista doble necesita enlazar, mutuamente, el nodo anterior y el nodo siguiente del que se borra, como se observa en la figura 34.7.

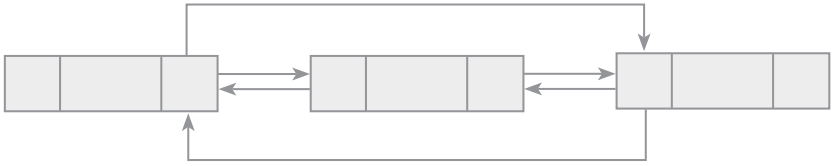


Figura 34.7 Eliminación de un nodo en una lista doblemente enlazada.

Nodo de una lista doblemente enlazada

La clase `NodoDoble` agrupa los componentes del nodo de una lista doble y las operaciones de la *interfaz*.

```
  // archivo de cabecera NodoDoble.h
class NodoDoble
{
protected:
  Dato dato;
  NodoDoble* adelante;
  NodoDoble* atras;
public:
  NodoDoble(Dato t)
  {
    dato = t;
    adelante = atras = NULL;
  }
  Dato datoNodo( ) const { return dato; }
  NodoDoble* adelanteNodo( ) const { return adelante; }
  NodoDoble* atrasNodo( ) const { return atras; }
  void ponerAdelante(NodoDoble* a) { adelante = a; }
  void ponerAtras(NodoDoble* a) { atras = a; }
};
```

Clase `ListaDoble`

La clase `ListaDoble` encapsula las operaciones básicas de las listas doblemente enlazadas. La clase dispone del puntero `cabeza` para acceder a la lista, apunta al primer nodo. El constructor de la clase inicializa la lista vacía. La declaración de la clase es:

```
  // archivo ListaDoble.h
class ListaDoble
{
  protected:
    Nodo* cabeza;
  public:
    ListaDoble( ) { cabeza = NULL;}
    void insertarCabezaLista(Dato entrada);
    void insertarUltimo(Dato entrada);
    void insertarLista(Dato entrada);
    void insertaDespues(NodoDoble* anterior, Dato entrada)
    NodoDoble* buscarLista(Dato destino);
    void eliminar(Dato entrada);
    Nodo* ultimo( );
    void visualizar( );
};
```

Insertar un nodo en una lista doblemente enlazada

Se pueden añadir nodos a la lista de distintas formas, según la posición donde se inserte. La posición de inserción puede ser:

- En la *cabeza* de la lista.
- Al final de la lista.
- Antes de un elemento especificado.
- Después de un elemento especificado.

Se describe e implementa la operación que inserta un nodo después de otro. Las otras formas de insertar siguen los mismos pasos que los descritos en una lista enlazada, teniendo en cuenta el doble enlace de los nodos.

Insertar después de un nodo

El algoritmo de la operación que inserta un nodo después de otro, `n`, requiere las siguientes etapas:

1. Crear un nodo, `nuevo`, con el elemento.
2. Poner el enlace `adelante` del nodo creado apuntando al nodo siguiente de `n`. 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`.

```
void ListaDoble::insertaDespues(NodoDoble* anterior, Dato entrada)
{
  NodoDoble* nuevo;
  nuevo = new NodoDoble(entrada);
  nuevo -> ponerAdelante(anterior -> adelanteNodo( ));
  if (anterior -> adelanteNodo( ) != NULL)
    anterior -> adelanteNodo( ) -> ponerAtras(nuevo);
  anterior-> ponerAdelante(nuevo);
  nuevo -> ponerAtras(anterior);
}
```

Eliminar un nodo de una lista doblemente enlazada

Quitar un nodo de una lista doble supone ajustar los enlaces de dos nodos, el nodo *anterior* con el nodo *siguiente* al que se desea eliminar. El puntero `adelante` del nodo anterior debe apuntar al nodo *siguiente*, y el puntero `atras` del nodo *siguiente* debe apuntar al nodo *anterior*.

El algoritmo es similar al del borrado para una lista simple, más simple ya que ahora la dirección del nodo *anterior* se encuentra en el campo `atras` del nodo a borrar. Los pasos a seguir:

1. Búsqueda del nodo que contiene el dato.
2. El puntero `adelante` del nodo anterior tiene que apuntar al puntero `adelante` del nodo a eliminar (si no es el nodo *cabecera*).

3. El puntero `atras` del nodo siguiente a borrar tiene que apuntar a donde apunta el puntero `atras` del nodo a eliminar (si no es el último nodo).
4. Si el nodo que se elimina es el primero se modifica `cabeza` para que tenga la dirección del nodo `siguiente`.
5. La memoria ocupada por el nodo es liberada.

La implementación del algoritmo es:

```
void ListaDoble::eliminar(Dato entrada)
{
  NodoDoble* actual;
  bool encontrado = false;
  actual = cabeza;
  // Bucle de búsqueda
  while ((actual != NULL) && (!encontrado))
  {
    encontrado = (actual -> datoNodo( ) == entrada);
    if (!encontrado)
      actual = actual -> adelanteNodo( );
  }
   // Enlace de nodo anterior con el siguiente
  if (actual != NULL)
  {
     //distingue entre nodo cabecera o resto de la lista
    if (actual == cabeza)
    {
      cabeza = actual -> adelanteNodo( );
      if (actual -> adelanteNodo( ) != NULL)
          actual -> adelanteNodo( ) -> ponerAtras(NULL);
    }
    else if (actual -> adelanteNodo( ) != NULL) // No es el último
    {
      actual->atrasNodo( ) -> ponerAdelante(actual->adelanteNodo( ));
      actual->adelanteNodo( ) -> ponerAtras(actual->atrasNodo( ));
    }
    else // último nodo
      actual->atrasNodo( ) ->ponerAdelante(NULL);
    delete actual;
  }
}
```

Listas enlazadas genéricas

La definición de una lista está muy ligada al tipo de datos de sus elementos; así se han puesto ejemplos en los que el tipo es `int`, otros en los que el tipo es `double`. C++ dispone del mecanismo `template` para declarar clases y funciones con independencia del tipo de dato de al menos un elemento. Entonces, el tipo de los elementos de una lista genérica será *un tipo genérico T*, que será conocido en el momento de crear la lista. La declaración es:

```
template <class T> class ListaGenerica
template <class T> class NodoGenerico

ListaGenerica<double> lista1; // lista de número reales
ListaGenerica<string> lista2; // lista de cadenas
```

Declaración de la clase `ListaGenérica`

Las operaciones del tipo *lista genérica* son las especificadas anteriormente en la clase `lista`. La clase que representa un nodo también es una clase genérica; se declara a continuación:

```
    // archivo NodoGenerico.h

template <class T> class NodoGenerico
{
```

```
  protected:
    T dato;
    NodoGenerico <T>* enlace;
  public:
    NodoGenerico (T t)
    {
      dato = t;
      enlace = 0;
    }
    NodoGenerico (T p, NodoGenerico<T>* n)
    {
      dato = p;
      enlace = n;
    }
    T datoNodo( ) const
    {
      return dato;
    }
    NodoGenerico<T>* enlaceNodo( ) const
    {
      return enlace;
    }
    void ponerEnlace(NodoGenerico<T>* sgte)
    {
      enlace = sgte;
    }
};
```

La declaración de la `ListaGenerica` es:

```
  // archivo ListaGenerica.h

template <class T> class ListaGenerica
{
  protected:
    NodoGenerico<T>* primero;

  public:
    ListaGenerica( ){ primero = NULL;}
    NodoGenerico<T>* leerPrimero( ) const { return primero;}
    void insertarCabezaLista(T entrada);
    void insertarUltimo(T entrada);
    void insertarLista(NodoGenerico<T>* ant, T entrada);
    NodoGenerico<T>* ultimo( );
    void eliminar(T entrada);
    NodoGenerico<T>* buscarLista(T destino);
};
```

La implementación de las funciones miembro, también son funciones genéricas, se realiza en el mismo archivo que declara la clase, en `ListaGenerica.h`. Esta implementación no difiere de la realizada con una lista, salvo que hay que especificar la genericidad. Se escriben algunas de las funciones.

inserción entre dos nodos de la lista

```
template <class T> void
ListaGenerica<T>::insertarLista(NodoGenerico<T>* ant, T entrada)
{

  NodoGenerico<T>* nuevo = new NodoGenerico<T>(entrada);
  nuevo -> ponerEnlace(ant -> enlaceNodo( ));
  ant -> ponerEnlace(nuevo);
}
```

borrado del primer nodo encontrado con el dato *entrada*

```
template <class T>
void ListaGenerica<T>:: eliminar(T entrada)
{
```

```
  NodoGenerico<T> *actual, *anterior;
  bool encontrado;
  actual = primero;
  anterior = NULL;
  encontrado = false;
    // búsqueda del nodo y del anterior
  while ((actual != NULL) && !encontrado)
  {
    encontrado = (actual -> datoNodo( ) == entrada);
    if (!encontrado)
    {
      anterior = actual;
      actual = actual -> enlaceNodo( );
    }
  }
   // enlace del nodo anterior con el siguiente
  if (actual != NULL)
  {
    // Distingue entre cabecera y resto de la lista
    if (acual == primero)
    {
      primero = actual -> enlaceNodo( );
    }
    else
      anterior -> ponerEnlace(actual -> enlaceNodo( ));
  delete actual;
  }
}
```

Iterador de `ListaGenerica`

Un objeto *iterador* se diseña para recorrer los elementos de un contenedor. Un iterador de una lista enlazada accede a cada uno de sus nodos, hasta alcanzar el último elemento. El constructor del objeto iterador inicializa el puntero *actual* al primer elemento de la estructura; la función `siguiente( )` devuelve el elemento *actual* y hace que éste quede apuntando al siguiente elemento. Si no hay siguiente, devuelve `NULL`.

La clase `ListaIterador` implementa el iterador de una lista enlazada genérica, y en consecuencia también será clase genérica.

```
template <class T> class ListaIterador
{
private:
  NodoGenerico<T> *prm, *actual;
public:
  ListaIterador(const ListaGenerica<T>& list)
  {
    prm = actual = list.leerPrimero( );
  }
  NodoGenerico<T> *siguiente( )
  {
    NodoGenerico<T> * s;
    if (actual != NULL)
    {
      s = actual;
      actual = actual -> enlaceNodo( );
    }
    return s;
  }
  void iniciaIterador( ) // pone en actual la cabeza de la lista
  {
    actual = prm;
  }
}
```

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. Los elementos de la pila se añaden o quitan sólo por su parte superior, la **cima** de la pila. Las pilas son estructura de datos **LIFO** (*last-in, first-out/último en entrar, primero en salir*).

La operación `insertar` (*push*) sitúa un elemento en la cima de la pila y `quitar` (*pop*) elimina o extrae el elemento cima de la pila.

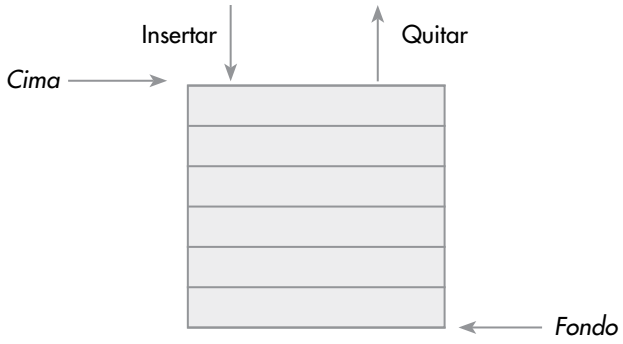


Figura 34.8 Operaciones básicas de una pila.

Especificaciones del tipo `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	Quitar 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

La pila se puede implementar guardando los elementos en un array, y por consiguiente su dimensión es fija, la que tiene el array. 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.

Clase `Pila` con arreglo (*array*)

Guardar los elementos de una pila tiene el problema del crecimiento, el máximo de elementos está prefijado por la longitud del arreglo. La operación *Pila llena* se implementa cuando se utiliza un arreglo para almacenar los elementos; cuando se utiliza una estructura dinámica no tiene sentido ya que crece indefinidamente.

La declaración de la clase se realiza en el archivo `PilaLineal.h`:

```
typedef tipo TipoDeDato;        // tipo de los elementos de la pila
const int TAMPILA = 49;

class PilaLineal
{
```

```
private:
  int cima;
  TipoDeDato listaPila[TAMPILA];
public:
  PilaLineal( )
  {
    cima = -1;    // condición de pila vacía
  }
   // operaciones que modifican la pila
  void insertar(TipoDeDato elemento);
  TipoDeDato quitar( );
  void limpiarPila( );
   // operación de acceso a la pila
  TipoDeDato cimaPila( );
   // verificación estado de la pila
  bool pilaVacia( );
  bool pilaLlena( );
};
```

Las funciones son sencillas de implementar, teniendo en cuenta la característica principal de esta estructura: *inserciones y borrados se realizan por el mismo extremo, la cima de la pila.*

La operación `insertar` un elemento en la pila incrementa el apuntador `cima` y asigna el nuevo elemento a `listaPila`. Cualquier intento de añadir un elemento en una pila llena genera una excepción debido al "Desbordamiento de la pila".

```
void PilaLineal::insertar(TipoDeDato elemento)
{
  if (pilaLlena( ))
  {
    throw "Desbordamiento pila";
  }
  //incrementar puntero cima y copia elemento
  cima++;
  listaPila[cima] = elemento;
}
```

La operación `quitar` elimina un elemento de la pila. Copia, en primer lugar, el valor de la cima de la pila en una variable local, `aux`, y, a continuación, decrementa el índice `cima` de la pila. Devuelve el elemento eliminado por la operación. Si se intenta eliminar o borrar un elemento en una pila vacía se produce error, se lanza una excepción.

```
TipoDeDato PilaLineal::quitar( )
{
  TipoDeDato aux;
  if (pilaVacia( ))
  {
    throw "Pila vacía, no se puede extraer";
  }
   // guarda elemento de la cima
  aux = listaPila[cima];
   // decrementar cima y devolver elemento
  cima--;
  return aux;
}
```

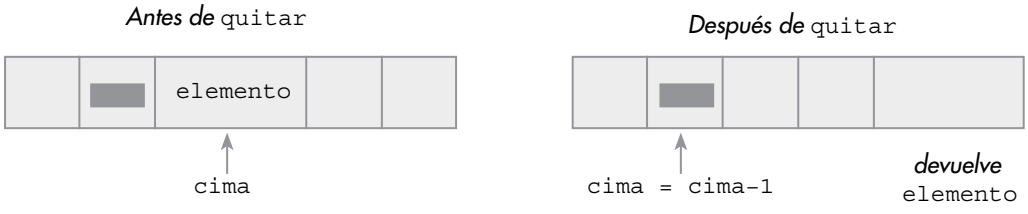


Figura 34.9 Quitar el elemento `cima`.

Ejemplo 34.4

Escribir un programa que cree una pila de enteros. Se realizan operaciones de añadir datos a la pila, quitar ... .

Se supone implementada la clase pila con el tipo primitivo `int`. El programa crea una pila de números enteros, inserta en la pila elementos leídos del teclado (hasta leer la clave `-1`), a continuación extrae los elementos de la pila hasta que se vacía. En pantalla deben escribirse los números leídos en orden inverso por la naturaleza de la pila. El bloque de sentencias se encierra en un bloque `try` para tratar errores de desbordamiento de la pila.

```
#include <iostream>
using namespace std;
typedef int TipoDeDato;
#include "pilalineal.h"

int main( )
{
  PilaLineal pila;  // crea pila vacía
  TipoDeDato x;
  const TipoDeDato CLAVE = -1;

  cout << "Teclea elemento de la pila(termina con -1)" << endl;
  try {
    do {
      cin >> x;
      pila.insertar(x);
    }while (x != CLAVE);

    cout << "Elementos de la Pila: " ;
    while (!pila.pilaVacia( ))
    {
      x = pila.quitar( );
      cout << x << " ";
    }
  }
  catch (const char * error)
  {
    cout << "Excepcion: " << error;
  }
  return 0;
}
```

Pila genérica con listas enlazadas

Utilizar una lista enlazada para implementar una pila hace que ésta pueda crecer o decrecer dinámicamente. Como las operaciones de *insertar* y *extraer* en el *TAD Pila* se realizan por el mismo extremo (cima de la pila), las acciones correspondientes con la lista se realizarán siempre por el mismo extremo de la lista.

> ***Nota***
> Una pila realizada con una lista enlazada crece y decrece dinámicamente. En tiempo de ejecución, se reserva memoria según se ponen elementos en la pila, y se libera memoria según se extraen elementos de la pila.

Clase `PilaGenerica` y `NodoPila`

Los elementos de la pila son los nodos de la lista, con un campo para guardar el elemento y otro de enlace. Las operaciones a implementar son, naturalmente, las mismas que si la pila se implementara

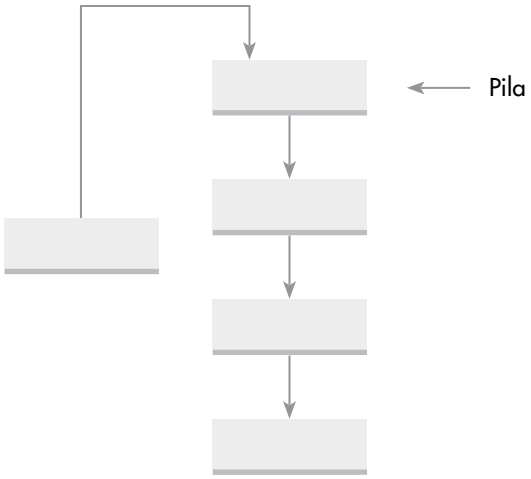


Figura 34.10 Representación de una pila con una lista enlazada.

con arreglos, salvo la operación que controla si la pila está llena, `pilaLlena`, que ahora no tiene significado ya que las listas enlazadas crecen indefinidamente, con el único límite de la memoria.

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 genérico, se diseña una pila genérica utilizando las *plantillas* (`template`) de C++. La clase `NodoPila` representa un nodo de la lista enlazada y tiene dos atributos: `elemento`, guarda el elemento de la pila y `siguiente`, contiene la dirección del siguiente nodo de la lista. En esta implementación, `NodoPila` es una clase anidada de `PilaGenerica`.

```
// archivo PilaGenerica.h

template <class T>
class PilaGenerica
{
  class NodoPila
  {
    public:
      NodoPila* siguiente;
      T elemento;
      NodoPila(T x)
      {
        elemento = x;
        siguiente = NULL;
      }
    };
    NodoPila* cima;

public:
  PilaGenerica ( )
  {
    cima = NULL;
  }
  void insertar(T elemento);
  T quitar( );
  T cimaPila( );
  bool pilaVacia( );
  void limpiarPila( );
  ~PilaGenerica( )
  {
    limpiarPila( );
  }
};
```

Implementación de las operaciones

El constructor de `PilaGenerica` inicializa a ésta como pila vacía (`cima = NULL`), realmente, a la condición de *lista vacía*. Las operaciones `insertar`, `quitar` y `cimaPila` acceden a la lista directamente con el puntero `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)`.

A continuación se escribe la codificación de las operaciones que modifican el estado de la pila.

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.

```
template <class T>
void PilaGenerica<T>::insertar(T elemento)
{
  NodoPila* nuevo;
  nuevo = new NodoPila(elemento);
  nuevo -> siguiente = cima;
  cima = nuevo;
}
```

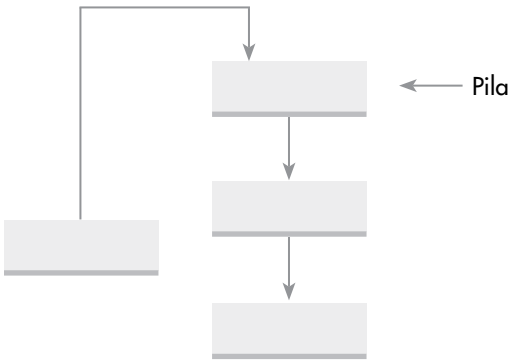


Figura 34.11 Apilar un elemento.

Eliminación del elemento cima

Retorna el elemento cima y lo quita de la pila, disminuye el tamaño de la pila.

```
template <class T>
T PilaGenerica<T>::quitar( )
{
  if (pilaVacia( ))
    throw "Pila vacía, no se puede extraer.";
  T aux = cima -> elemento;
  Nodofila *a = cima -> siguiente;
  cima = cima -> siguiente;
  delete a;
  return aux;
}
```

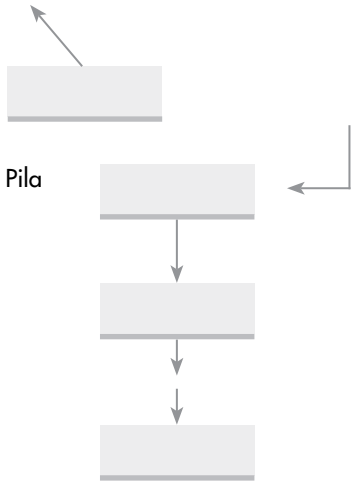


Figura 34.12 Quita la cima de la pila.

Vaciado de la pila

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

```
template <class T>
void PilaGenerica<T>::limpiarPila( )
{
  NodoPila* n;
  while(!pilaVacia( ))
  {
    n = cima;
    cima = cima -> siguiente;
    delete n;
  }
}
```

Tipo abstacto de datos `cola`

Una **cola** es una estructura de datos lineal, a la que se accede por uno de los dos extremos de la lista. Los elementos se insertan por el extremo *final*, y se suprimen por el otro extremo, llamado *frente*. Las aplicaciones utilizan una cola para almacenar elementos en su orden de aparición o concurrencia.

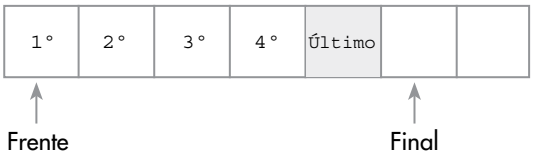


Figura 34.13 Una cola.

Los elementos se quitan de la cola en el mismo orden en que se almacenan y, por consiguiente, 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*).

> ***Definición***
> Una cola es una estructura de datos cuyos elementos mantienen un 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**.

Especificaciones del tipo abstracto de datos `cola`

Las operaciones que definen la estructura de una cola son las siguientes:

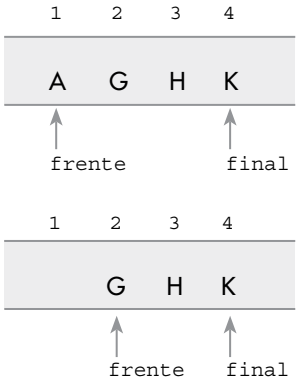
Tipo de dato	Elemento que se almacena en la cola
Operaciones	
CrearCola	Inicia la cola como vacía
Insertar	Añade un elemento por el final de la cola
Quitar	Retira (extrae) el elemento frente
Cola vacía	Comprueba si la cola no tiene elementos
Cola llena	Comprueba si la cola está llena de elementos
Frente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

La forma que los lenguajes tienen para representar el *TAD cola* depende de dónde se almacenen los elementos, en un arreglo, o en una estructura dinámica como puede ser una lista enlazada. La utilización de arreglo tiene el problema de que la cola no puede crecer indefinidamente, está limitada por el tamaño del arreglo. Como contrapartida, el acceso a los extremos es muy eficiente. Utilizar una estructura dinámica permite que el número de nodos se ajuste al de elementos de la cola.

Colas implementadas con *arrays*

La implementación estática de una cola se realiza declarando un array para almacenar los elementos, y dos marcadores o apuntadores para mantener las posiciones `frente` y `final` de la cola. Cuando un elemento se añade a la cola, se verifica si el marcador `final` apunta a una posición válida, entonces se asigna el elemento en esa posición y se incrementa el marcador `final` en 1. Cuando un elemento se elimina de la cola, se hace una prueba para ver si la cola está vacía y, si no es así, se recupera el elemento de la posición apuntada por el marcador `frente` y éste se incrementa en 1.

El avance lineal de `frente` y `final` tiene un grave problema, deja *huecos* por la *izquierda del array*. Llega a ocurrir que `final` alcanza el índice más alto del arreglo, no pudiéndose añadir nuevos elementos y que, sin embargo existan posiciones libres a la izquierda de `frente`.



posición de `frente` y `final` después de extraer.

Figura 34.14 Una cola representada en un *array*.

La clase `ColaLineal` declara un arreglo (`listaCola`) de tamaño `MAXTAMQ`. Las variables `frente` y `final` son los apuntadores a cabecera y cola, respectivamente. El constructor de la clase inicializa la estructura, de tal forma que se parte de una cola vacía.

Las operaciones básicas del tipo abstracto de datos cola: `insertar`, `quitar`, `colaVacia`, `colaLlena`, y `frente` se implementan en la clase. `insertar` toma un elemento y lo añade por el final. `quitar` suprime y devuelve el elemento *cabeza* de la cola. La operación `frente` devuelve el elemento que está en la primera posición (*frente*) de la cola, sin eliminar el elemento. `colaVacia` comprueba si la cola tiene elementos; `colaLlena` comprueba si se pueden añadir nuevos elementos.

```
// archivo ColaLineal.h

typedef tipo TipoDeDato;   // tipo ha de ser conocido
const int MAXTAMQ = 39;

class ColaLineal
{
protected:
  int frente;
  int final;
  TipoDeDato listaCola[MAXTAMQ];
public:
  ColaLineal( )
  {
```

```
    frente = 0;
    final = -1;
  }
   // operaciones de modificación de la cola
  void insertar(const TipoDeDato& elemento)
  {
    if (!colaLlena( ))
    {
      listaCola[++final] = elemento;
    }
    else
      throw "Overflow en la cola";
  }
  TipoDeDato quitar( )
  {
    if (!colaVacia( ))
    {
      return listaCola[frente++];
    }
    else
      throw "Cola vacia ";
  }
  void borrarCola( )
  {
    frente = 0;
    final = -1;
  }
   // acceso a la cola
  TipoDeDato frenteCola( )
  {
    if (!colaVacia( ))
    {
      return listaCola[frente];
    }
    else
      throw "Cola vacia ";
  }
   // métodos de verificación del estado de la cola
  bool colaVacia( )
  {
    return frente > final;
  }
  bool colaLlena( )
  {
    return final == MAXTAMQ-1;
  }
};
```

A recordar

La realización de una cola con un arreglo lineal es notablemente ineficiente; se puede alcanzar la condición de cola llena habiendo elementos del arreglo sin ocupar.

Cola con un arreglo (*array*) circular

La forma más eficiente de almacenar una cola en un arreglo es modelar 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.

El apuntador `frente` siempre contiene la posición del primer elemento de la cola y avanza en el sentido de las agujas del reloj; `final` 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`:

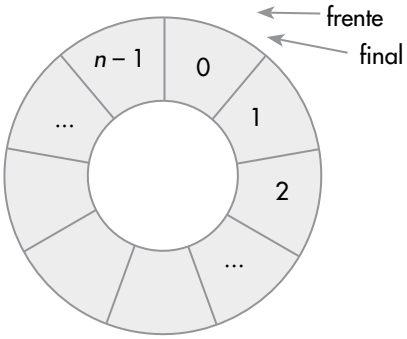


Figura 34.15 Cola con un elemento en un arreglo circular.

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

La implementación de la gestión de colas con un *array circular* han de incluir las operaciones básicas del *TAD cola*, es decir, las siguientes tareas básicas:

- Creación de una cola vacía, de tal forma que `final` apunte a una posición inmediatamente anterior a `frente`:

  ```
  frente = 0; final = MAXTAMQ-1.
  ```

- Comprobar si una cola está vacía:

  ```
  frente == siguiente(final)
  ```

- Comprobar si una cola está llena. Para diferenciar la condición *cola llena* de *cola vacía* se sacrifica una posición del arreglo: entonces la capacidad real de la cola será `MAXTAMQ-1`. La condición de cola llena:

  ```
  frente == siguiente(siguiente(final))
  ```

- Poner un elemento en la cola: si la cola no está llena, fijar `final` a la siguiente posición: `final = (final + 1) % MAXTAMQ` y asignar el elemento.
- Retirar un elemento de la cola: si la cola no está vacía, quitarlo de la posición `frente` y establecer `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 un arreglo circular

La clase `ColaCircular` implementa la representación de una cola en un arreglo circular. Para aprovechar la potencia de la orientación a objetos, y de C++, la clase deriva de `ColaLineal`. Es necesario *redefinir* las funciones de la *interfaz*. Además, se escribe la función auxiliar `siguiente( )` para obtener la siguiente posición de una dada, aplicando la *teoría de los restos*.

```
// archivo ColaCircular.h

#include "ColaLineal.h"
class ColaCircular : public ColaLineal
{

protected:
  int siguiente(int r)
  {
    return (r+1) % MAXTAMQ;
  }

  public:
    ColaCircular( )
```

```
    {
      frente = 0;
      final = MAXTAMQ-1;
    }
      // operaciones de modificación de la cola
    void insertar(const TipoDeDato& elemento);
    TipoDeDato quitar( );
    void borrarCola( );
     // acceso a la cola
    TipoDeDato frenteCola( );
     // métodos de verificación del estado de la cola
    bool colaVacia( );
    bool colaLlena( );
};
```

Implementación:

```
void ColaCircular :: insertar(const TipoDeDato& elemento)
{
  if (!colaLlena( ))
  {
    final = siguiente(final);
    listaCola[final] = elemento;
  }
  else
    throw "Overflow en la cola";
}

TipoDeDato ColaCircular :: quitar( )
{
  if (!colaVacia( ))
  {
    TipoDeDato tm = listaCola[frente];
    frente = siguiente(frente);
    return tm;
  }
  else
    throw "Cola vacia ";
}

void ColaCircular :: borrarCola( )
{
  frente = 0;
  final = MAXTAMQ-1;
}

TipoDeDato ColaCircular :: frenteCola( )
{
  if (!colaVacia( ))
  {
    return listaCola[frente];
  }
  else
    throw "Cola vacia ";
}

bool ColaCircular :: colaVacia( )
{
  return frente == siguiente(final);
}

bool ColaCircular :: colaLlena( )
{
  return frente == siguiente(siguiente(final));
}
```

Ejemplo 34.5

Se utiliza una Cola y una Pila para encontrar un número capicúa leído del dispositivo estándar de entrada.

El número se lee del teclado en forma de cadena de dígitos. La cadena se procesa carácter a carácter, es decir dígito a dígito (un dígito es un carácter del '0' al '9'). Cada dígito se pone en la cola y a la vez en la pila. Una vez que se terminan de leer los dígitos y de ponerlos en la cola y en la pila, comienza la comprobación: se extraen consecutivamente elementos de la cola y de la pila, y se comparan por igualdad; de producirse alguna no coincidencia entre dígitos es que el número no es capicúa y entonces se vacían las estructuras. El número es capicúa si el proceso de comprobación termina habiendo coincidido todos los dígitos en orden inverso, lo cual equivale a que la pila y la cola terminen vacías.

¿Por qué utilizar una pila y una cola? Sencillamente para asegurar que se procesan los dígitos en orden inverso; en la pila el *último en entrar es el primero en salir*, en la cola el *primero en entrar es el primero en salir*.

La pila es una instancia de la clase `PilaGenerica` y la cola de la clase `ColaCircular`. El código principal:

```
#include <iostream>
using namespace std;
#include "PilaGenerica.h"
typedef char TipoDeDato;
#include "ColaCircular.h"

bool capicua;
char numero[81];
PilaGenerica<char> pila;
ColaCircular q;

capicua = false;
while (!capicua)
{
  do {
    cout << "\nTeclea el número: ";
    cin.getline(numero,80);
  }while (!valido(numero)); // todos los caracteres dígitos
    // pone en la cola y en la pila cada dígito
  for (int i = 0; i < strlen(numero); i++)
  {
    char c;
    c = numero[i];
    q.insertar(c);
    pila.insertar(c);
  }
   // se retira de la cola y la pila para comparar
  do {
    char d;
    d = q.quitar( );
    capicua = d == pila.quitar( ); //compara por igualdad
  } while (capicua && !q.colaVacia( ));

  if (capicua)
    cout << numero << " es capicúa " << endl;
  else
  {
    cout << numero << " no es capicúa, ";
    cout << " intente con otro. ";
     // se vacía la cola y la pila
    q.borrarCola( );
    pila.limpiarPila( );
  }
}
```

Cola genérica con una lista enlazada

La implementación del *TAD cola* con independencia del tipo de dato de los elementos se consigue utilizando las *plantillas* (`template`) de C++. Para que la cola pueda crecer o decrecer, se utiliza la estructura dinámica lista enlazada. Los extremos de la cola son los punteros de acceso a la lista: `frente` y `final`. El puntero `frente` apunta al primer elemento de la lista; el puntero `final`, al último elemento de la lista.

Clase `ColaGenerica`

La implementación de la *Cola Genérica* se realiza con dos clases: clase `ColaGenerica` y clase `NodoCola` que será una clase anidada. 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`.

La declaración de la clase y la implementación se encuentran en el archivo `ColaGenerica.h`:

```
// archivo ColaGenerica.h
template <class T>
class ColaGenerica
{
protected:
  class NodoCola
  {
  public:
    NodoCola* siguiente;
    T elemento;
    NodoCola (T x)
    {
      elemento = x;
      siguiente = NULL;
    }
  };
  NodoCola* frente;
  NodoCola* final;
public:
  ColaGenerica( )
  {
    frente = final = NULL;
  }
  void insertar(T elemento);
  T quitar( );
  void borrarCola( );
  T frenteCola( ) const;
  bool colaVacia( ) const;
  ~ColaGenerica( )
  {
    borrarCola ( );
  }
};
```

Las funciones acceden directamente a la lista, son también funciones genéricas. A continuación se escribe la codificación de las funciones que modifican la cola.

Añadir un elemento a la cola

```
template <class T>
void ColaGenerica<T> :: insertar(T elemento)
{
  NodoCola* nuevo;
  nuevo = new NodoCola (elemento);
  if (colaVacia( ))
  {
    frente = nuevo;
  }
```

```
  else
  {
    final -> siguiente = nuevo;
  }
  final = nuevo;
}
```

Sacar elemento frente de la cola

```
template <class T>
T ColaGenerica<T> :: quitar( )
{
  if (colaVacia( ))
    throw "Cola vacía, no se puede extraer.";
  T aux = frente -> elemento;
  NodoCola* a = frente;
  frente = frente -> siguiente;
  delete a;
  return aux;
}
```

Vaciado de la cola

Elimina todos los elementos de la cola. Recorre la lista desde frente a final; es una operación de complejidad lineal, `O(n)`.

```
template <class T>
void ColaGenerica<T> :: borrarCola( )
{
  for (;frente != NULL;)
  {
    NodoCola* a;
    a = frente;
    frente = frente -> siguiente;
    delete a;
  }
  final = NULL;
}
```

Bicolas: colas de doble entrada

Una *bicola* o *cola de doble entrada* es un conjunto *ordenado* de elementos al que se pueden *añadir* o *quitar* elementos desde cualquier extremo del mismo. El acceso a la *bicola* está permitido desde cualquier extremo, por lo que se considera que es una *cola bidireccional*. La estructura *bicola* es una extensión del *TAD cola*.

Los dos extremos de una bicola se identifican con los apuntadores `frente` y `final` (mismos nombres que en una cola). Las operaciones básicas que definen una bicola son una ampliación de las operaciones de una cola:

CrearBicola : inicializa una bicola sin elementos.
BicolaVacia : devuelve `true` si la bicola no tiene elementos.
PonerFrente : añade un elemento por el extremo frente.
PonerFinal : añade un elemento por el extremo final.
QuitarFrente : devuelve el elemento `frente` y lo retira de la bicola.
QuitarFinal : devuelve el elemento `final` y lo retira de la bicola.
Frente : devuelve el elemento `frente` de la bicola.
Final : devuelve el elemento `final` de la bicola.

Al tipo de datos bicola se pueden poner restricciones respecto a la entrada o a la salida de elementos. Una *bicola con restricción de entrada* es aquella que sólo permite inserciones por uno de los dos extremos, pero que permite retirar elementos por los dos extremos. Una *bicola con restricción de sa-*

lida es aquella que permite inserciones por los dos extremos, pero sólo permite retirar elementos por un extremo.

La representación de una bicola puede ser con un arreglo, con un *arreglo circular*, o bien con *listas enlazadas*. Siempre se debe disponer de *dos marcadores* o variables índice (*apuntadores*) que se correspondan con los extremos, *frente* y *final*, de la estructura.

`BicolaGenerica` con listas enlazadas

La implementación del *TAD bicola* con una lista enlazada se caracteriza por ajustarse al número de elementos; es una implementación dinámica, *crece* o *decrece* según lo requiera la ejecución del programa que utiliza la bicola. Como los elementos de una bicola, y en general de cualquier *estructura contenedora*, pueden ser de cualquier tipo, se declara la clase genérica *bicola*. Además, la clase va a heredar de `ColaGenerica` ya que es una extensión de ésta.

```
template <class T> class BicolaGenerica : public ColaGenerica<T>
```

De esta forma, `BicolaGenerica` dispone de todas las funciones y variables de la clase `ColaGenerica`. Entonces, sólo es necesario codificar las operaciones de `Bicola` que no están implementadas en `ColaGenerica`. La declaración de la clase:

```
// archivo BicolaGenerica.h
#include "ColaGenerica.h"

template <class T>
class BicolaGenerica : public ColaGenerica<T>
{
public:
  void ponerFinal(T elemento);
  void ponerFrente(T elemento);
  T quitarFrente( );
  T quitarFinal( );
  T frenteBicola( )const;
  T finalBicola( )const;
  bool bicolaVacia( )const;
  void borrarBicola( );
  int numElemsBicola( )const;// cuenta los elementos de la bicola
};
```

Implementación de las operaciones de `BicolaGenerica`

Las funciones: `ponerFinal( )`, `quitarFrente( )`, `bicolaVacia( )`, `frenteBicola( )` son idénticas a las funciones de la clase `ColaGenerica` `insertar( )`, `quitar( )`, `colaVacia( )` y `frenteCola( )` respectivamente; se han heredado por el mecanismo de derivación de clases; su implementación consiste en una simple llamada a la correspondiente función heredada. La implementación de alguna de estas funciones:

Añadir un elemento a la bicola

Añadir por el extremo `final` de la bicola.

```
template <class T>
void BicolaGenerica<T> :: ponerFinal(T elemento)
{
  insertar(elemento); // heredado de ColaGenerica
}
```

Añadir por el extremo `frente` de la bicola.

```
template <class T>
void BicolaGenerica<T> :: ponerFrente(T elemento)
{
  NodoCola* nuevo;
  nuevo = new NodoCola(elemento);
```

```
  if (bicolaVacia( ))
  {
    final = nuevo;
  }
  nuevo -> siguiente = frente;
  frente = nuevo;
}
```

Sacar un elemento de la bicola

Devuelve el elemento `frente` y lo quita de `Bicola`, disminuye su tamaño.

```
template <class T>
T BicolaGenerica<T> :: quitarFrente( )
{
  return quitar( ); // método heredado de ColaLista
}
```

Devuelve el elemento `final` y lo quita de la bicola, disminuye su tamaño. Es necesario recorrer la lista para situarse en el nodo anterior a `final`, y después enlazar.

```
template <class T>
T BicolaGenerica<T> :: quitarFinal( )
{
  T aux;
  if (! bicolaVacia( ))
  {
    if (frente == final)   // Bicola dispone de un solo nodo
    {
      aux = quitar( );
    }
    else
    {
      NodoCola* a = frente;
      while (a -> siguiente != final)
        a = a -> siguiente;
      aux = final -> elemento;
      final = a;
      delete (a -> siguiente);
    }
  }
  else
    throw "Eliminar de una bicola vacía";
  return aux;
}
```

Número de elementos de la bicola

Recorre la estructura, de `frente` a `final`, para contar el número de elementos de que consta.

```
template <class T>
int BicolaGenerica<T> :: numElemsBicola( ) const
{
  int n = 0;
  NodoCola* a = frente;
  if (!bicolaVacia( ))
  {
    n = 1;
    while (a != final)
    {
      n++;
      a = a -> siguiente;
    }
  }
  return n;
}
```

Resumen

- Una **lista enlazada** es una estructura de datos dinámica, que se crea vacía y crece o decrece en tiempo de ejecución. Los componentes de una lista están ordenados lógicamente por sus campos de enlace en vez de estar ordenados físicamente como están en un arreglo.
- El final de la lista se señala mediante una constante o puntero especial llamado `NULL`.
- La principal ventaja de una lista enlazada sobre un arreglo radica en el tamaño dinámico de la lista, ajustándose al número de elementos.
- La desventaja de la lista frente a un arreglo está en el acceso a los elementos: para un arreglo el acceso es directo, a partir del índice; para la lista el acceso a un elemento se realiza mediante el campo enlace entre nodos.
- 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.
- Una **lista doblemente enlazada** es aquella en la que cada nodo tiene una referencia a su sucesor y otro 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 genérica** tiene como tipo de dato el *tipo genérico T*, es decir, el tipo concreto se especificará en el momento de crear el objeto lista. La construcción de una *lista genérica* se realiza con las plantillas (`template`), mediante dos clases genéricas: clase `NodoGenerico` y `ListaGenerica`.
- En C++ cada tipo abstracto de datos se implementa con una clase. En el capítulo se ha implementado la clase `lista`, la clase `ListaDoble` y la clase `ListaGenerica`.
- 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`.
- El tipo abstracto `pila` se implementa mediante una clase en la que se agrupan las operaciones y el acceso a los datos de la pila.
- Una cola es una lista lineal en la que los datos se insertan por un extremo (final) y se retiran 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: `crearCola`, `colaVacia`, `colaLlena`, `insertar`, `frente` y `quitar`.
- La implementación del *TAD cola*, en C++, se realiza con *arrays*, o bien con listas enlazadas. La implementación con un *array* lineal es muy ineficiente; se ha de considerar el *array* como una estructura circular.
- La realización de una cola con listas enlazadas permite que el tamaño de la estructura se ajuste al número de elementos.
- Las *bicolas* son *colas dobles* en el sentido de que las operaciones básicas *insertar* y *retirar* elementos se realizan por los dos extremos. Una bicola es, realmente, una extensión de una cola. La implementación natural del *TAD bicola* es con una clase derivada de la clase *cola*.

Ejercicios

34.1. Escribir una función, en la clase `Lista`, que devuelva cierto si la lista está vacía.

34.2. Añadir a la clase `ListaDoble` una función que devuelva el número de nodos de una lista doble.

34.3. ¿Cuál es la salida de este segmento de código?, teniendo en cuenta que es una pila de tipo `int`:

```
Pila p;
int x = 4, y;
```

```
p.insertar(x);
cout << "\n " << cimaPila( );
y = p.quitar( );
p.insertar(32);
p.insertar(p.quitar( ));
do {
  cout << "\n " << p.quitar( );
}while (!p.pilaVacia( ));
```

34.4. A la clase `Lista` añadir la función `eliminaraPosicion( )` que retire el nodo que ocupa la posición `i`, siendo 0 la posición del nodo cabecera.

34.5. Escribir un función que tenga como argumento el puntero al primer nodo de una lista enlazada, y cree una lista doblemente enlazada con los mismos campos `dato` pero en orden inverso.

34.6. Supóngase que se tiene la clase `Cola` que implementa las operaciones del *TAD cola*. Escribir una función para crear un *clon* (una copia) de una cola determinada. Las operaciones que se han de utilizar serán únicamente las del *TAD cola*.

34.7. Escribir una función para crear una lista doblemente enlazada de palabras introducidas por teclado. El acceso a la lista debe ser el nodo que está en la posición intermedia.

34.8. Se tiene una pila de enteros positivos. Con las operaciones básicas de pilas y colas escribir un fragmento de código para poner todos los elementos de la pila que son par en la cola.

34.9. Con las operaciones implementadas en la clase `PilaGenerica` escribir la función `mostrarPila( )` que muestre en pantalla los elementos de una pila de cadenas.

34.10. Suponer una lista doblemente enlazada de cadenas ordenada alfabéticamente. Escribir una función para añadir una nueva palabra, en el orden que le corresponda, a la lista.

34.11. Dada la lista del ejercicio 34.10 escribir una función que elimine una palabra dada.

34.12. Considere una *bicola* de caracteres, representada en un *array* circular. El *array* consta de 9 posiciones. Los extremos actuales y los elementos de la bicola:

```
frente = 5 final = 7        bicola: A, C, E
```

Escribir los extremos y los elementos de la bicola según se realizan estas operaciones:

- Añadir los elementos `F` y `K` por el `final` de la bicola.
- Añadir los elementos `R`, `W` y `V` por el `frente` de la bicola.
- Añadir el elemento `M` por el `final` de la bicola.
- Eliminar dos caracteres por el `frente`.
- Añadir los elementos `K` y `L` por el `final` de la bicola.
- Añadir el elemento `S` por el `frente` de la bicola.

Problemas

34.1. Escribir un programa que realice las siguientes tareas:

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

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

34.3. Un polinomio se puede representar como una lista enlazada. El primer nodo representa el primer término del polinomio, el segundo nodo, el segundo término del polinomio y así sucesivamente. Cada nodo tiene como campo `dato` el coeficiente del término y su exponente. Por ejemplo, $3x^4 - 4x^2 + 11$ se representa:

Desarrollar la clase `Polinomio` con los atributos grado del polinomio, y una lista para representar los términos del polinomio. Dotar a la clase de funciones para dar entrada a un polinomio, visualizar el polinomio, evaluarlo para un valor numérico de la variable *x*.

34.4. Con la clase del ejercicio 34.3, escribir un programa que dé entrada a polinomios en *x*. A continuación, obtener valores del polinomio para valores de `x = 0.0, 0.5, 1.0, 1.5, ..., 5.0`.

34.5. Para determinar frases que son palíndromo, se ha de seguir la siguiente estrategia: considerar cada línea una frase; añadir cada carácter de la frase a una pila y, a la vez, a una lista enlazada doble por el final de la lista; extraer carácter a carácter, simultáneamente de la pila y de la lista doble por el extremo *cabeza*; su comparación determina si es palíndromo o no. Escribir un programa que lea líneas y determine si son palíndromo.

34.6. Según la representación de un polinomio propuesta en el problema 34.3, añadir las siguientes operaciones:

- Obtener la suma de dos polinomios.
- Obtener el polinomio derivado.

34.7. Un pequeño supermercado dispone en la salida de tres cajas de pago. En el local hay 25 carritos de compra. Escribir un programa que simule el funcionamiento, siguiendo las siguientes reglas:

- Si cuando llega un cliente no hay ningún carrito disponible, espera a que lo haya.
- Ningún cliente se impacienta y abandona el supermercado sin pasar por alguna de las colas de las cajas.
- Cuando un cliente finaliza su compra, se coloca en la cola de la caja donde hay menos gente, y no se cambia de cola.
- En el momento en que un cliente paga en la caja, el carro de la compra que tiene queda disponible.
- Representar la lista de carritos de la compra y las cajas de salida mediante colas.

34.8. Un conjunto es una secuencia de elementos todos del mismo sin duplicados. Representar el tipo conjunto con una clase, de tal forma que los elementos del conjunto se almacenen en un objeto `Lista`. Las operaciones a desarrollar:

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

34.9. Con la representación propuesta en el problema 34.8, añadir las operaciones básicas de conjuntos:

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

34.10. Un vector disperso es aquel que tiene muchos elementos que son cero. Escribir una clase que represente un vector disperso con listas enlazadas. Los nodos son los elementos del vector distintos de cero. Cada nodo contendrá el valor del elemento y su índice (posición del vector). Las operaciones a considerar: sumar dos vectores de igual dimensión y hallar el producto escalar.