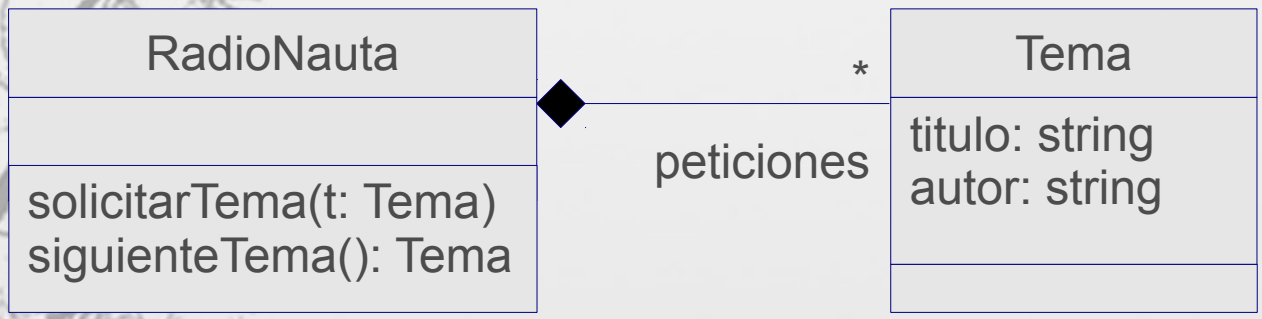
Lección 6:
Listas enlazadas

- Motivación
- Listas enlazadas
- Nodos enlazados
- Inserción
- Borrado
- Iteración
- Conclusiones

Motivación

- Una emisora de radio acepta peticiones de canciones por parte de los oyentes
- Las peticiones se guardan en una lista de donde se van pinchando canciones por orden de llegada
- Para mejorar la calidad de la selección musical no se pinchan dos canciones consecutivas del mismo interprete

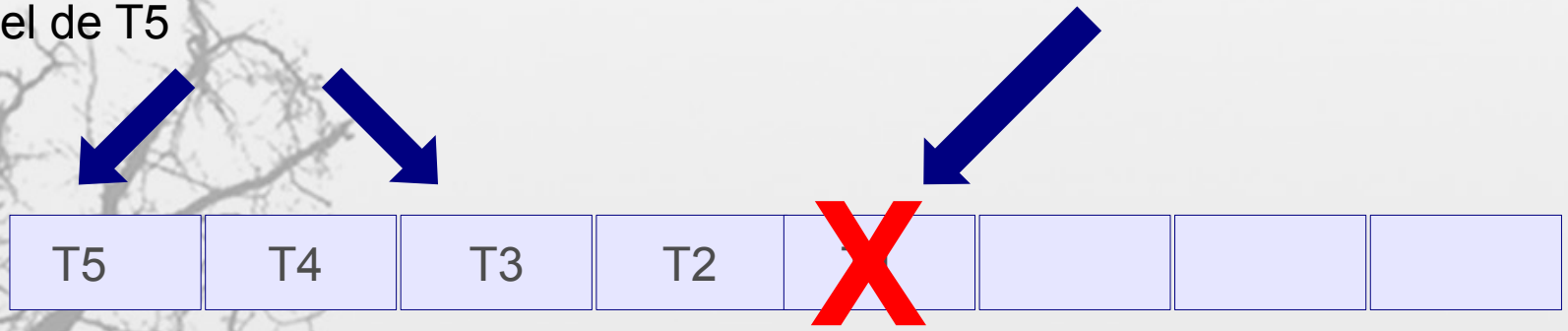


Motivación (cont.)

- Un vector no es una buena elección para la lista de peticiones
- Añadir cada nueva petición tienen un coste O(n)!

Insertamos una nueva canción (T6) por el principio o más adelante si su autor coincide con el de T5

Extraemos la siguiente canción a pinchar por el final (T1)



Listas enlazadas

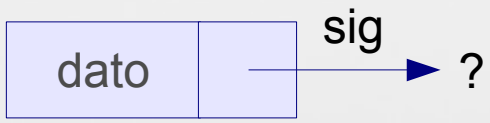
- Una **lista enlazada** es un contenedor secuencial que admite inserciones por el principio y el final en tiempo O(1)
- Borrados en O(1) al principio y O(n) al final
- Mucho más complejas que los vectores

	Inserción/ borrado principio	Inserción/ borrado final	Inserción/ borrado arbitrario	Lectura/ escritura posición arbitraria
listas	O(1)	O(1)/O(n)	O(n)	O(n)
vectores	O(n)	O(1)	O(n)	O(1)

Nodos enlazados

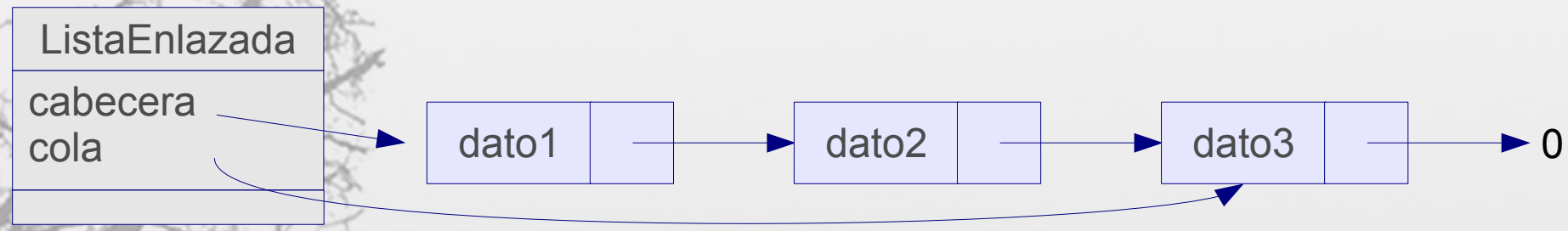
- Una lista enlazada está formada por nodos enlazados
- Cada nodo contiene un dato y un puntero al siguiente nodo en la secuencia

```
template<class T>
class Nodo {
public:
    T dato;
    Nodo *sig;

    Nodo(T &aDato, Nodo *aSig = 0):
        dato(aDato), sig(aSig) {}
};
```



Listas enlazadas

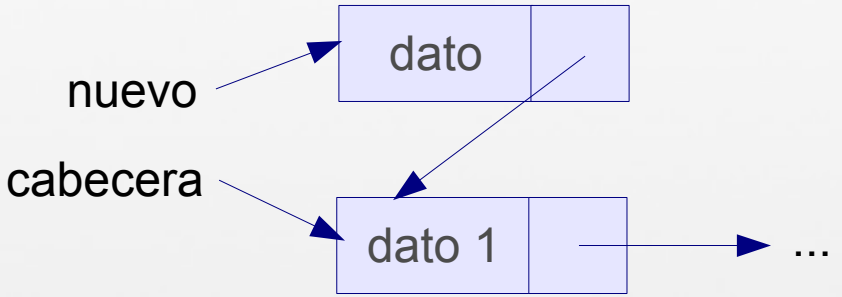
- Una lista enlazada es una secuencia de nodos enlazados
- El primer y último nodo deben ser apuntados por atributos de la clase
- El último nodo apunta a nulo



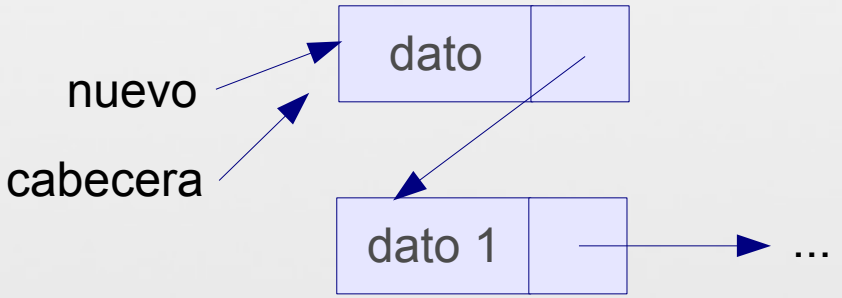
Inserción al principio

- La inserción varía ligeramente si la inserción es al principio, final u otra posición

```
Nodo<T> *nuevo;
nuevo  = new Nodo<T>(dato, cabecera);
```

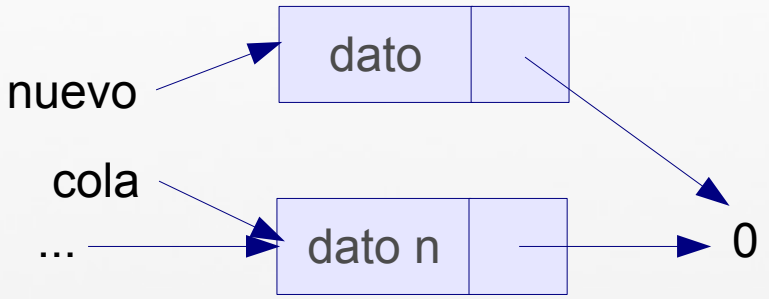



```
// Caso especial: si la lista
// estaba vacía poner la cola
// apuntando al nodo
if (cola == 0)
    cola = nuevo;

cabecera = nuevo;
```

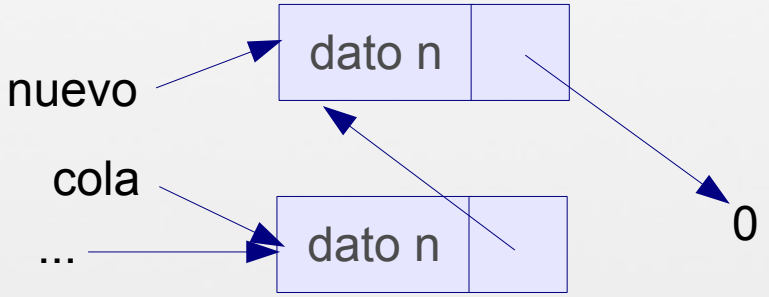


Inserción al final

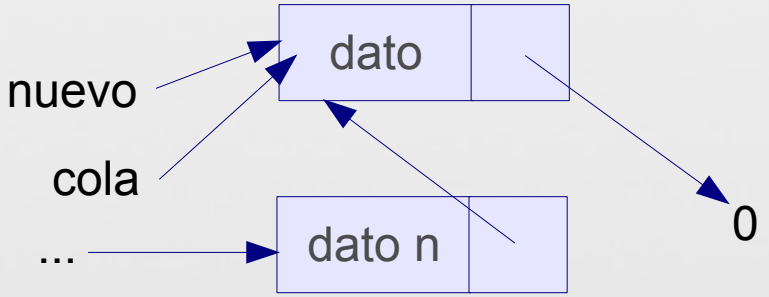
```
Nodo<T> *nuevo;
nuevo = new Nodo<T>(dato, 0);
```




```
if (cola != 0)
    cola->sig = nuevo;
```

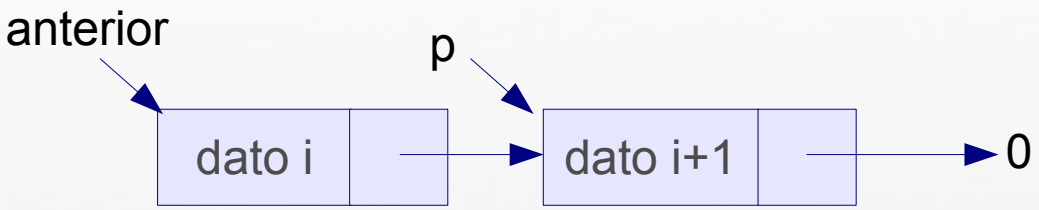



```
// Caso especial: si la lista
// estaba vacía, poner la
// cabecera apuntando al nodo
if (cabecera == 0)
    cabecera = nuevo;

cola = nuevo;
```

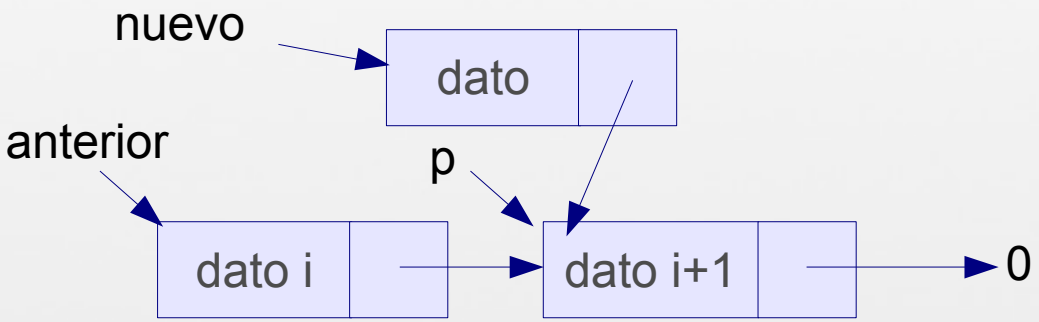


Inserción en medio

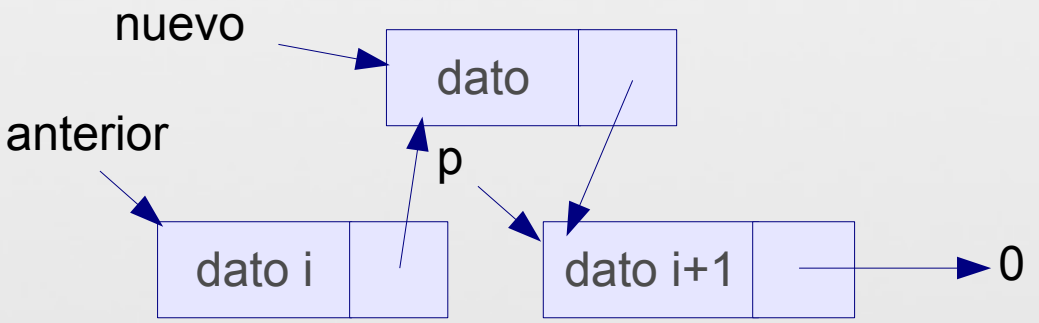
```
Nodo<T> *anterior = 0;

if (cabecera != cola) {
    anterior = cabecera;
    while (anterior->sig != p)
        anterior = anterior->sig
}
```




```
Nodo<T> *nuevo;
nuevo  = new Nodo<T>(dato, p);
```



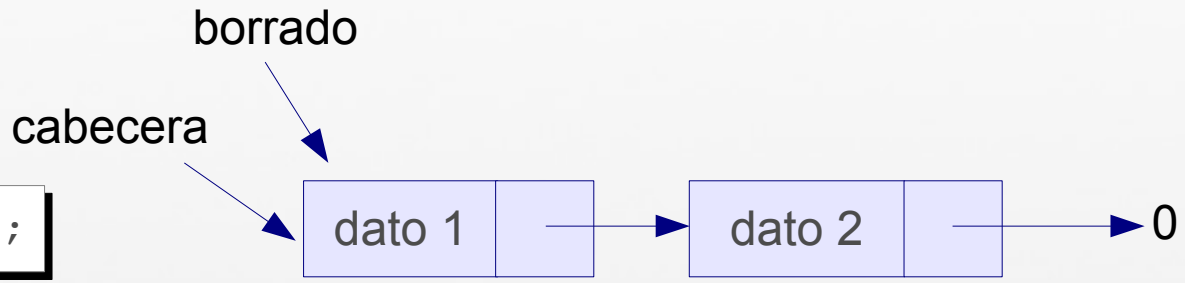

```
anterior->sig = nuevo;

if (cabecera == 0)
    cabecera = cola = nuevo;
```



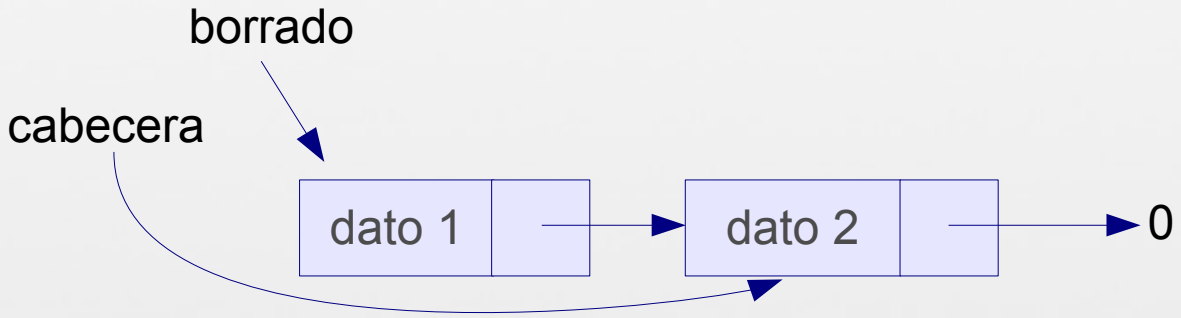
Borrar el primer nodo

- Tres casos también para el borrado

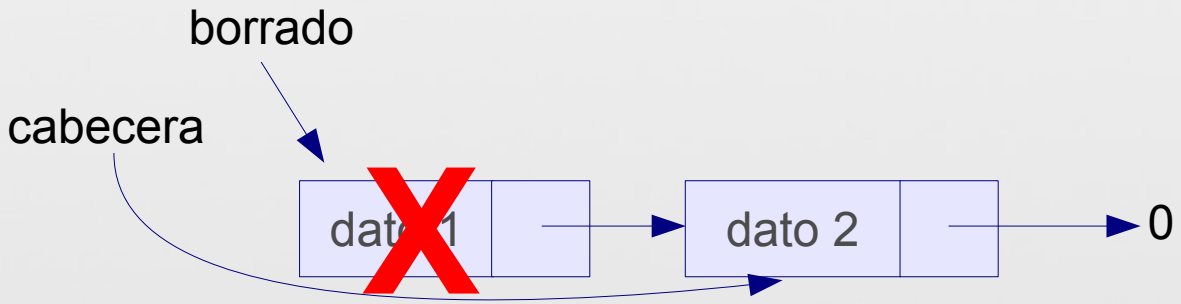
```
Nodo<T> *borrado = cabecera;
```




```
cabecera = cabecera->sig;
```

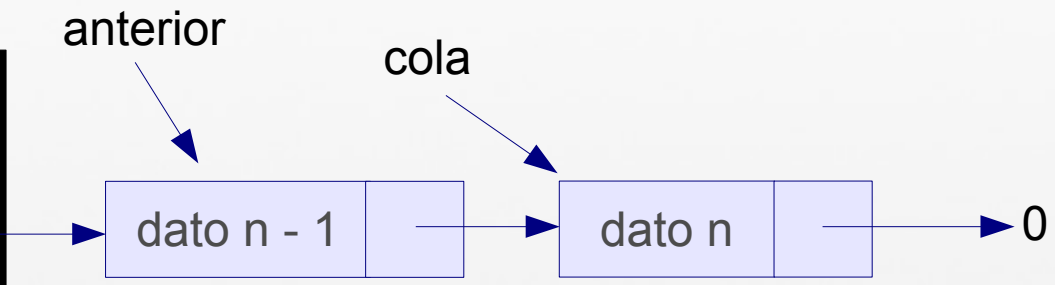



```
delete borrado;

// Caso especial al
// borrar el último nodo
if (cabecera == 0)
     cola = 0;
```

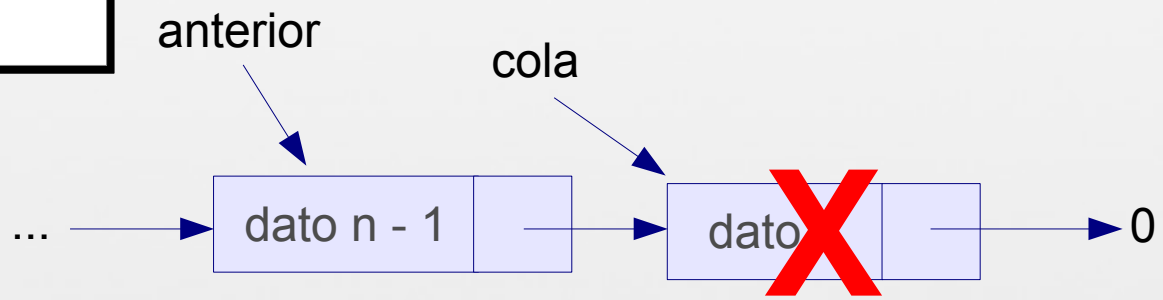


Borrar el último nodo

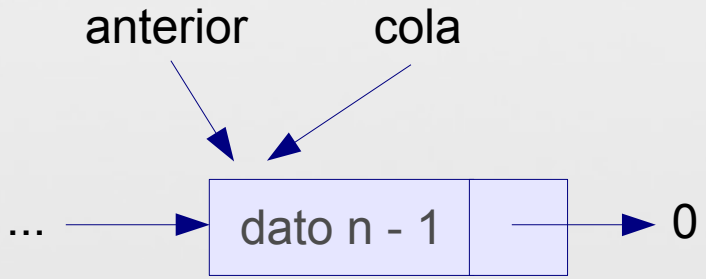
```
Nodo<T> *anterior = 0;

if (cabecera != cola) {
    anterior = cabecera;
    while (anterior->sig != cola)
        anterior = anterior->sig
}
```




```
delete cola;
```

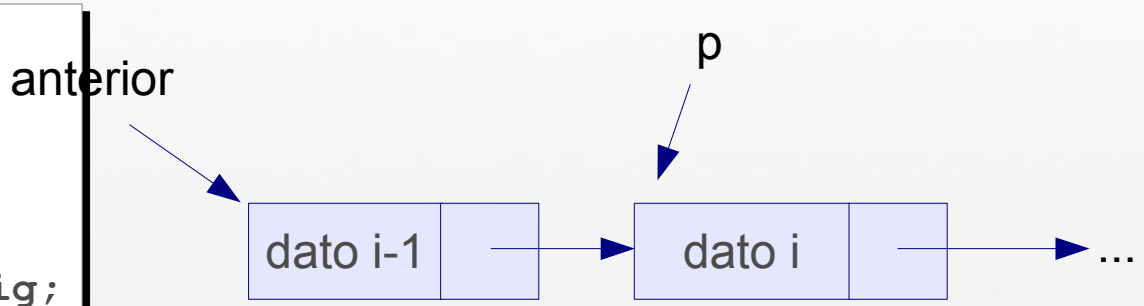



```
cola = anterior;
if (anterior != 0)
    anterior->sig = 0;
else
    cabecera = 0;
```

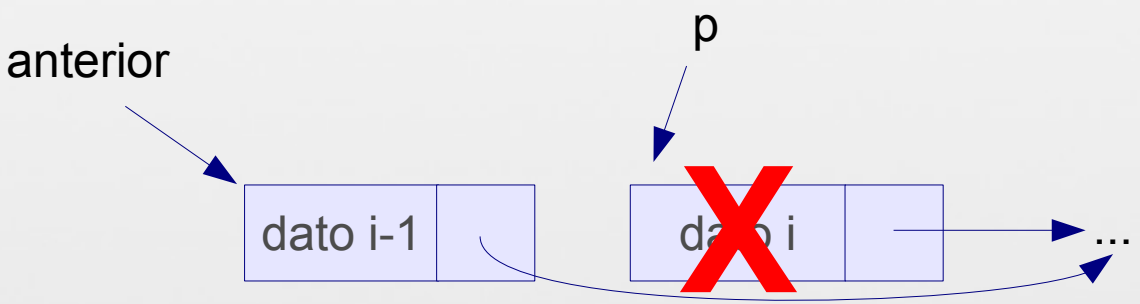


Borrar un nodo interior

```
Nodo<T> *anterior = 0;

if (cabecera != cola) {
    anterior = cabecera;
    while (anterior->sig != p)
        anterior = anterior->sig;
}
```




```
anterior->sig = p->sig;
delete p
```



Acceso a los datos en una lista

- Acceder al dato n-esimo en un vector requiere tiempo O(1)
- En cambio en una lista es ineficiente!: O(n)

```
template<class T>
T ListaEnlazada<T>::leer(int n) {
    Nodo<T> *nodo = cabecera;
    while (n-- > 0 && nodo != 0) {
        nodo = nodo->sig
    }

    if (nodo != 0)
        return nodo->dato;
    return 0;
}
```

Iteración

- Hay que evitar los accesos aleatorios y sustituirlos por recorridos secuenciales mediante **iteradores**
- Un iterador es una clase que apunta a un nodo como un puntero, pero su manejo es de más alto nivel
- Un iterador tiene operaciones para avanzar al siguiente nodo, obtener o modificar el dato del nodo apuntado
- El iterador es construido y devuelto por la clase *ListaEnlazada*, apuntando al primer nodo
- Las operaciones de inserción y borrado en posiciones interiores usan ahora iteradores para indicar la posición

La clase Iterador

```
template<class T>
class Iterador {
    Nodo<T> *nodo;
    friend class ListaEnlazada;
public:
    Iterador(Nodo<T> *aNodo) : nodo(aNodo) {}

    bool fin() { return nodo == 0; }
    void siguiente() {
        nodo = nodo->sig;
    }

    T &dato() { return nodo->dato; }
};
```

Uso de iteradores

Imprimir los datos de una lista de enteros

```
Iterador<int> i = lista.iterador();
while (!i.fin()) {
    cout << i.dato() << endl;
    i.siguiente();
}
```

Escribir 0 en todas las posiciones de la lista

```
Iterador<int> i = lista.iterador();
while (!i.fin()) {
    i.dato() = 0;
    i.siguiente();
}
```

La interfaz de la clase ListaEnlazada

```
template<class T>
class ListaEnlazada {
    Nodo<T> *cabecera, *cola;
public:
    ListaEnlazada() : cabecera(0), cola(0) {}
    ~ListaEnlazada();
    ListaEnlazada(const ListaEnlazada &l);
    ListaEnlazada &operator=(ListaEnlazada &l);

    Iterador<T> iterador() { return Iterador<T>(cabecera); }
    void insertarInicio(T &dato);
    void insertarFinal(T &dato);
    void insertar(Iterador<T> &i, T &dato);
    void borrarInicio();
    void borrarFinal();
    void borrar(Iterador<T> &i);
    T &inicio();
    T &final();
};
```

Solución al ejemplo

En el ejemplo de la radio usaríamos una lista enlazada para las peticiones

```
Tema RadioNauta::siguienteTema() {
    Tema t = peticiones.final();
    peticiones.borrarFinal();
    return t;
}

void RadioNauta::solicitarTema(Tema &t) {
    Iterador<Tema> i = peticiones.iterador();

    while (i.dato().autor() == t.autor()) {
        if (!i.fin())
            i.siguiente();
        if (!i.fin())
            i.siguiente();
    }
    peticiones.insertar(i, t);
}
```

Conclusiones

- Mejoran los vectores al permitir inserción y borrado por el principio en tiempo O(1)
- Las inserciones en posiciones intermedias requieren tiempo O(n), aunque con una constante asociada mucho menor que un vector
- La inserción por el final es O(1) pero el borrado es O(n) (peor que en un vector)
- No son una buena elección si se requieren accesos arbitrarios mediante índices enteros
- Sólo permiten iteración desde el principio al final