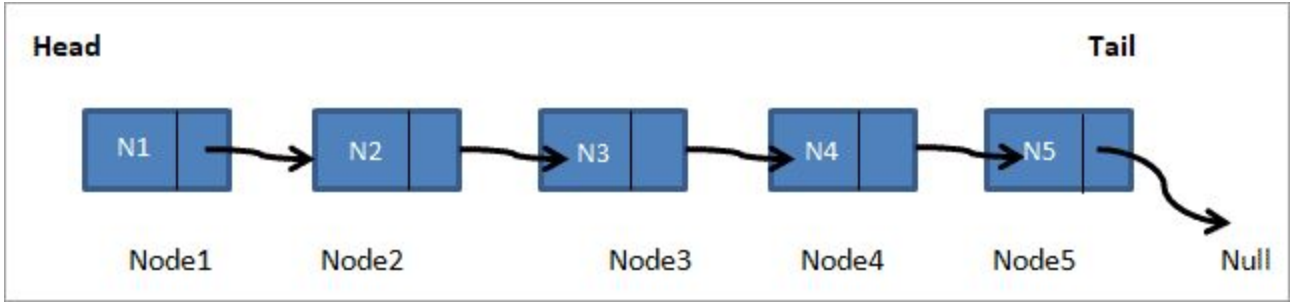
Listas simplemente enlazadas



Lección 6:

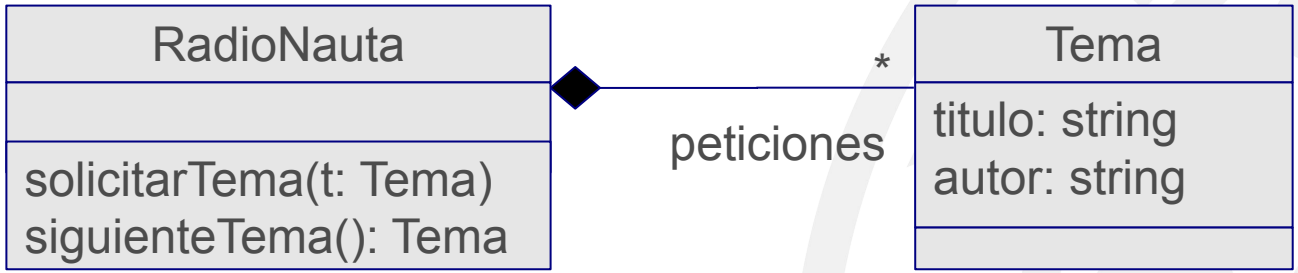
- Listas simplemente enlazadas
- Iteración sobre listas

Parte I:
Listas simplemente enlazadas

- Motivación
- Definición de listas y nodos
- Inserción
- Borrado

Motivación

- Una emisora de radio acepta peticiones de canciones por parte de los oyentes.
- Las peticiones se guardan en una lista desde donde se van pinchando canciones por orden de llegada.
- Para mejorar la calidad de la selección musical no se pincha la misma canción si se pinchó recientemente.

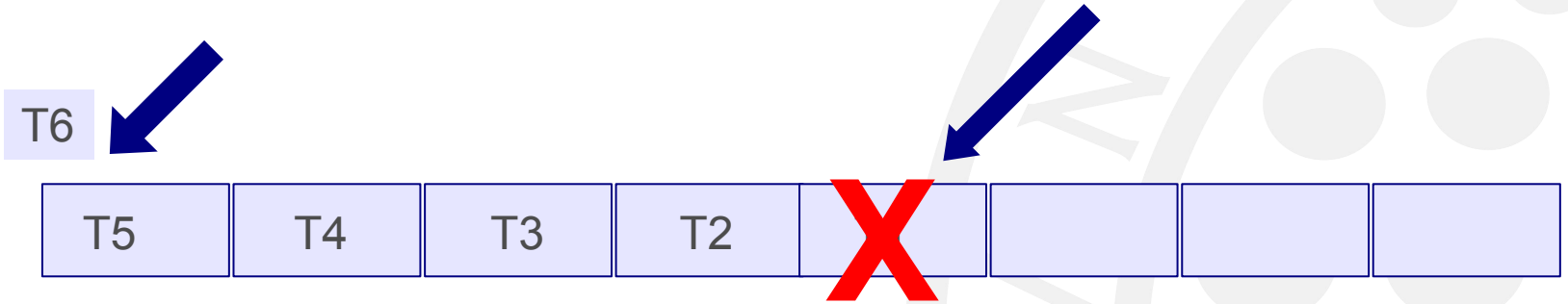


Motivación

- Un vector no es una buena elección para la lista de peticiones
- Si se quiere eliminar una canción porque ya ha sido escuchada el coste es O(n)

Insertar el tema 6 al inicio es una operación O(n)

- Eliminar la última canción una vez que se ha pinchado es O(1)
- Pero eliminar una canción en medio es O(n)



Listas enlazadas

- Una **lista simplemente 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
- Son mucho más complejas de trabajar que los vectores
- No necesita bloques contiguos de memoria

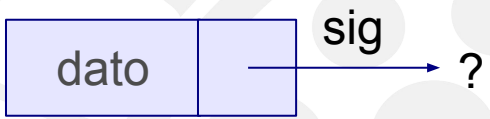
	Inserción/ borrado principio	Inserción/ borrado final	Inserción/borrado posición arbitraria	Lectura/ escritura posición arbitraria
listas	O(1)	O(1)/O(n)	O(n)	O(n)
vectores	O(n)	O(1)	O(n)	O(1)

Nodos de listas enlazadas

- Una lista enlazada está formada por nodos enlazados
- Cada nodo contiene un dato y un puntero al siguiente nodo en la secuencia
- Se crea un constructor que por defecto hace que el puntero al siguiente apunte a nulo (**nullptr == 0**)

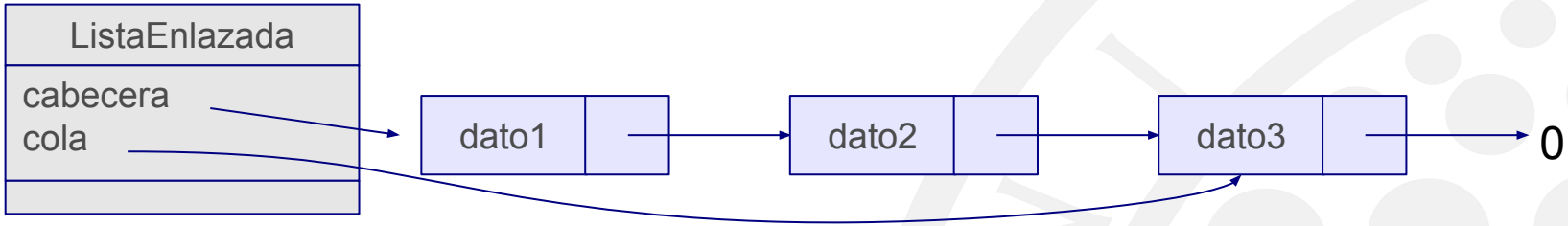
```
template<class T>
class Nodo {
public:
     T dato;
     Nodo *sig;

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



Definición de 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



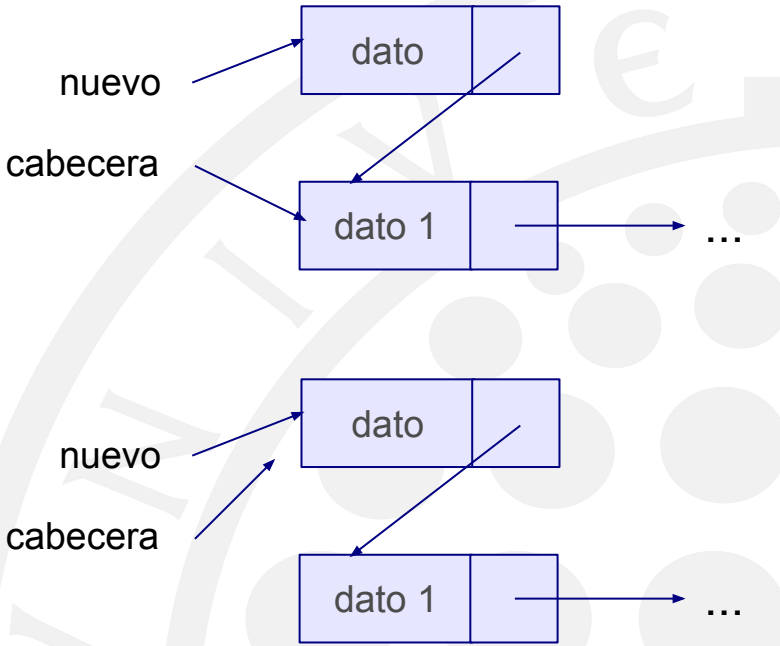
Inserción al principio

- La inserción varía ligeramente si 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;
```



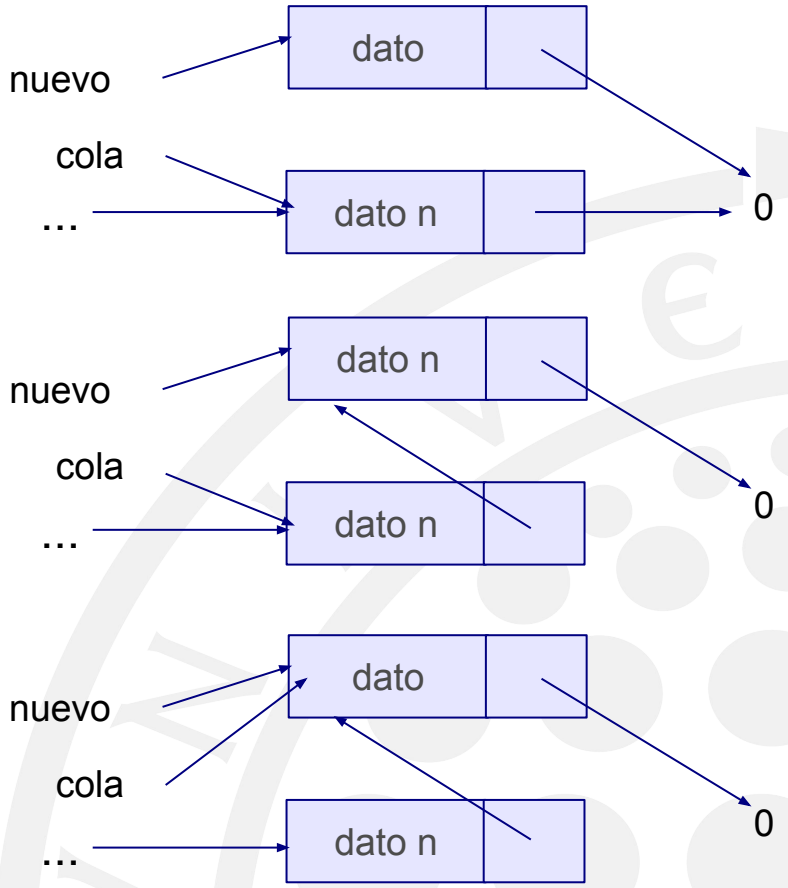
Inserción al final

```
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 (delante de *p*)

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

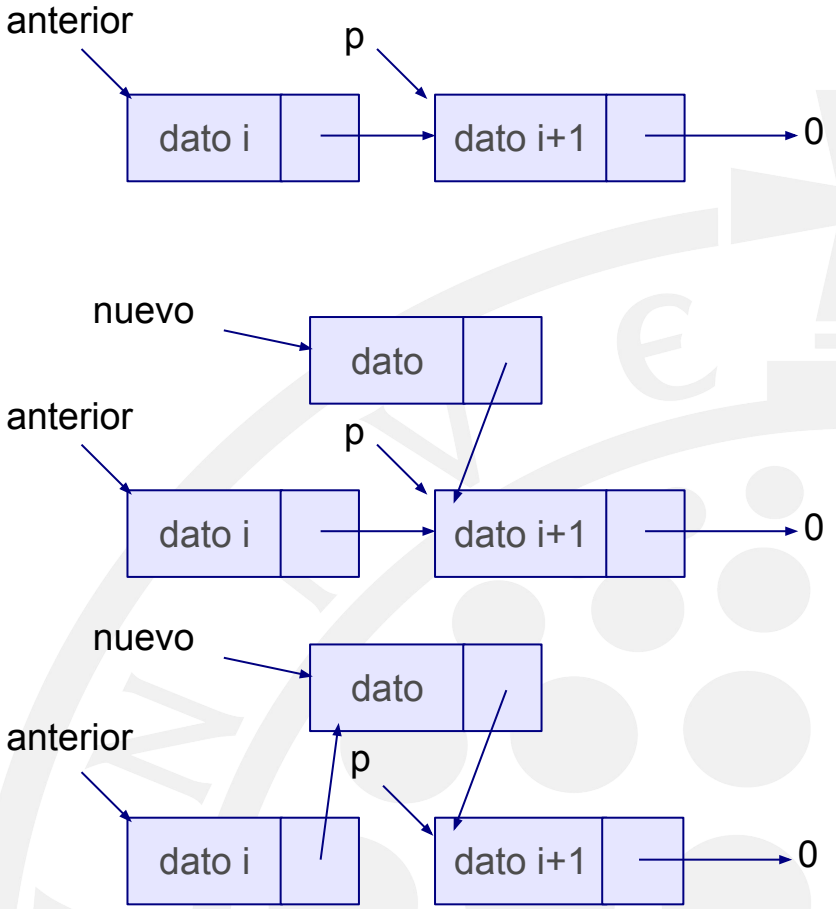
Insertar por el inicio

```
if (p==cabecera)
   cabecera = nuevo;
```

Caso general: posición anterior a p

```
Nodo<T> *anterior = 0;
if (cabecera != cola) {
     anterior = cabecera;
     while (anterior->sig != p)
          anterior = anterior->sig
}
```

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



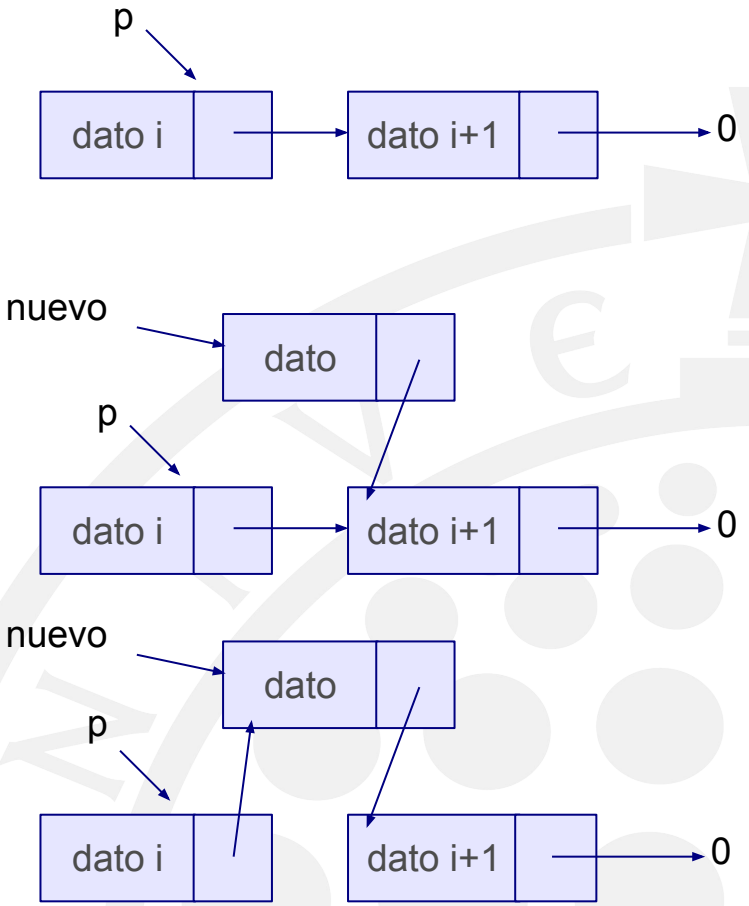
Inserción en medio (detrás de *p*)

Caso general: posición a continuación de p

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

```
if (cola==p)
    cola = nuevo;
```

Esta operación es O(1) si el iterador ya está colocado



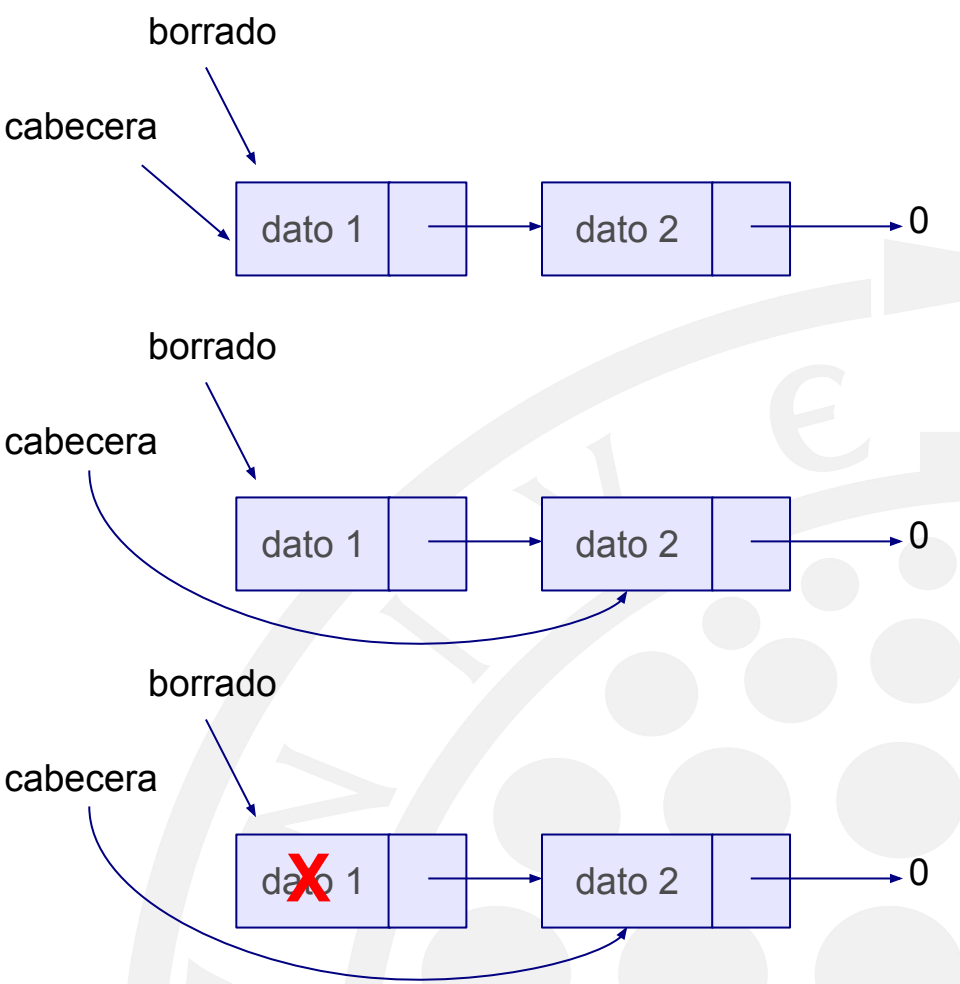
Borrar el primer nodo

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

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

```
delete borrado;

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



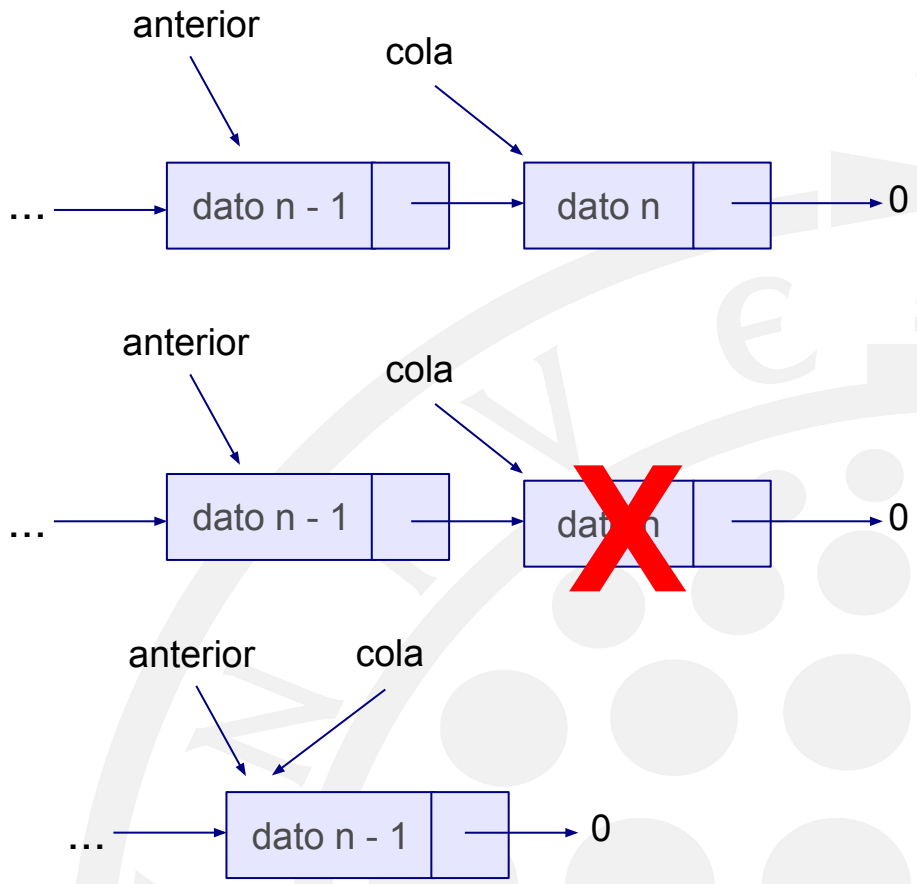
Borrar el último nodo

```
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;
```

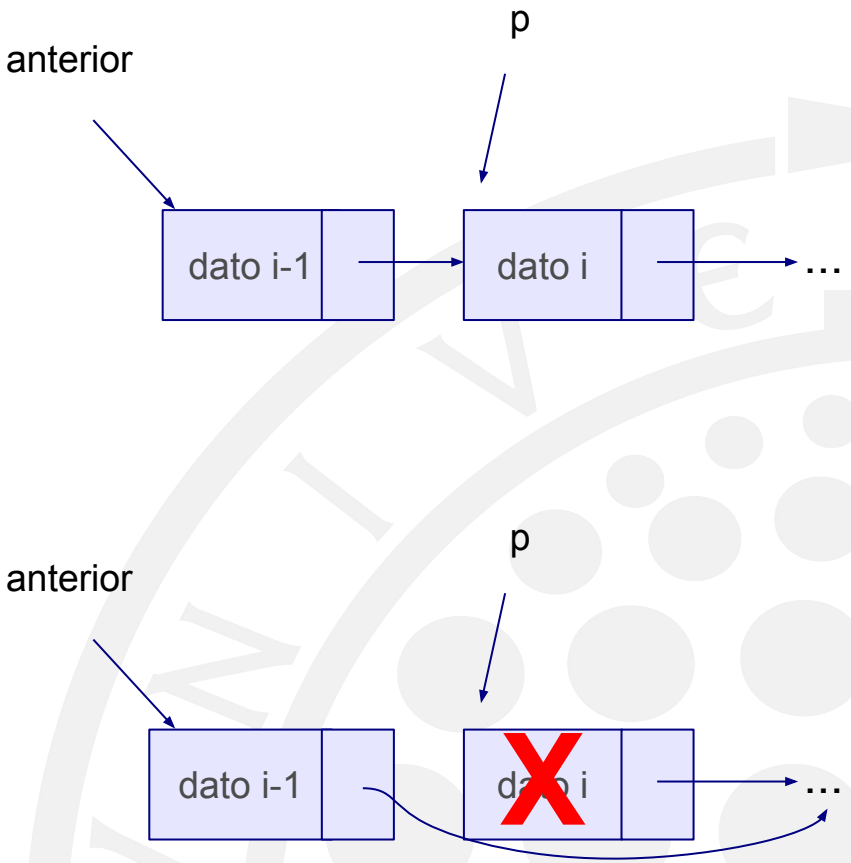


Borrar un nodo interior

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

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

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



Parte II:
Iteración sobre la lista

- Acceso a los datos de la lista
- La clase iteración
- La interfaz de la clase Lista
- Conclusiones

Acceso a los datos de la lista

- Acceder al dato n-ésimo en un vector requiere tiempo **O(1)**
- En cambio en una lista es muy ineficiente, **O(n)** , y no suele ser una operación válida

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

     if (nodo == 0)
          throw std::out_of_range("[listaEnlazada::leer] posición no válida");
    return nodo->dato
}
```

Iteración

- Hay que evitar los accesos aleatorios y sustituirlos por recorridos secuenciales mediante iteradores
- Un iterador es una clase que permite realizar un recorrido sobre una colección de elementos
- Funciona como una clase que encapsula a un puntero
- Un iterador tiene operaciones para avanzar al siguiente nodo, obtener o modificar el dato del nodo apuntado.
- De otro modo ese dato no es accesible desde fuera de la clase
- El iterador es construido y devuelto por la clase ListaEnlazada<T>, 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; }
     T &operator*() { 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();
}
```

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

Conclusiones

- Las listas simplemente enlazadas 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 si se tiene ya colocado el iterador en la posición es O(1)
- 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
- Parte de estos inconvenientes se resuelven con las listas doblemente enlazadas (Tema 7)

De ahora en adelante…

- Las listas enlazadas son más eficientes en las operaciones que aprovechan la segmentación de memoria
  - inserciones por el medio o por el principio, pero recordad que antes se debe colocar el iterador en el lugar correcto.
- Sin embargo, tienen problemas para borrar por el final y la iteración en sólo en una dirección
- Parte de esta problemática se arregla con las listas doblemente enlazadas

Lección 7: Listas doblemente enlazadas