

Lección 7:

Listas doblemente enlazadas.

Listas circulares y matrices dispersas

- Motivación
- Listas doblemente enlazadas
- Inserciones y borrados
- Iteración en listas doblemente enlazadas
- Listas circulares
- Matrices dispersas

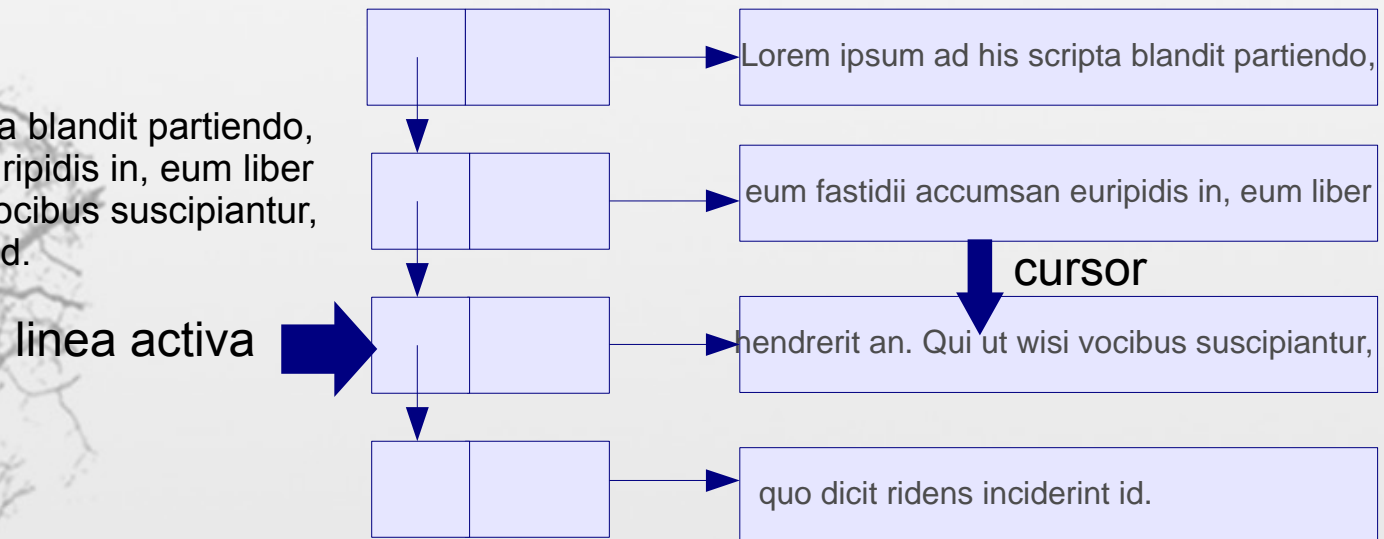


Motivación



- Estamos implementando un editor de textos sencillo
- Un documento está formado por un conjunto de líneas, cada una de ellas formadas por un conjunto de caracteres
- Usaremos una lista para las líneas y un vector dinámico para guardar el contenido de cada línea

Lorem ipsum ad his scripta blandit partiendo,
eum fastidii accumsan euripidis in, eum liber
hendrerit an. Qui ut wisi vocibus suscipiantur,
quo dicit ridens inciderint id.



Motivación

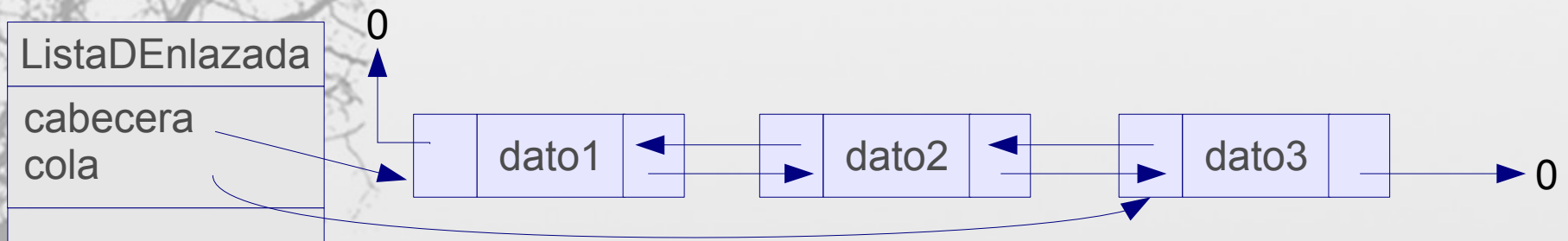


- Esta estructura permite añadir líneas en cualquier posición y modificar una línea añadiendo, borrando o modificando caracteres
- Sin embargo conforme se edita el texto, la estructura de datos facilita el pasar de una línea a la siguiente, pero no a la anterior
- La lista enlazada no es suficientemente flexible



Listas doblemente enlazadas

- Una **lista doblemente enlazada** es similar a las listas que ya conocemos, con la diferencia de cada nodo tiene además un puntero al nodo anterior
- Este detalle garantiza $O(1)$ en la inserción y borrado en cualquier posición, además de permitir iteración bidireccional



Nodos doblemente enlazados

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

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

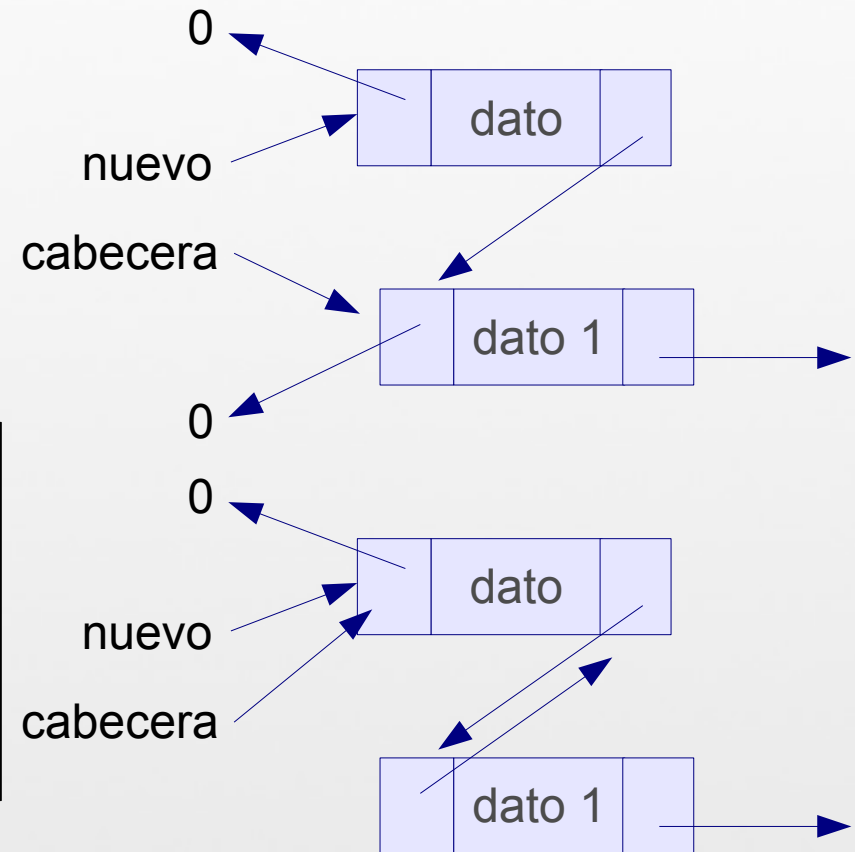


Inserción al principio



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

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



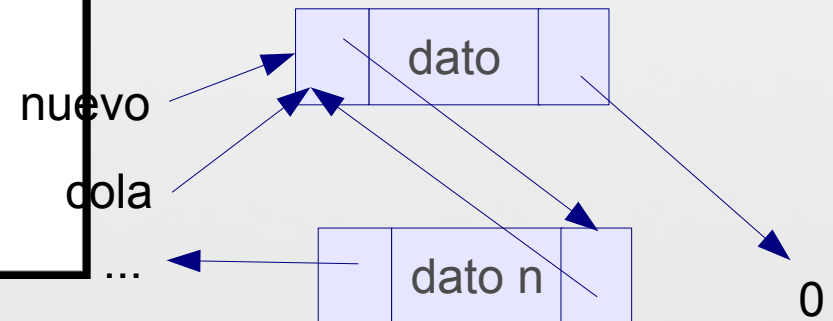
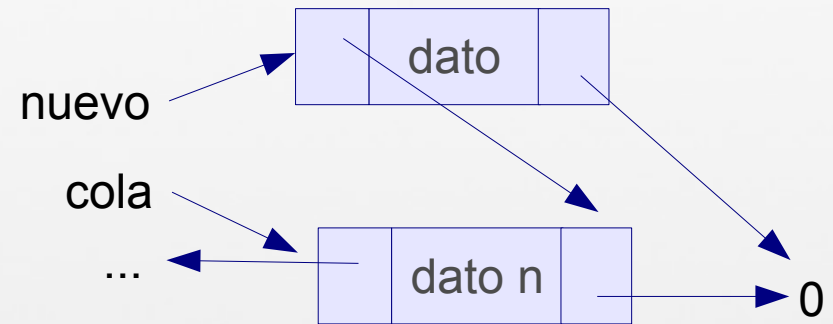


Inserción al final

- Es simétrico a la inserción al principio

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

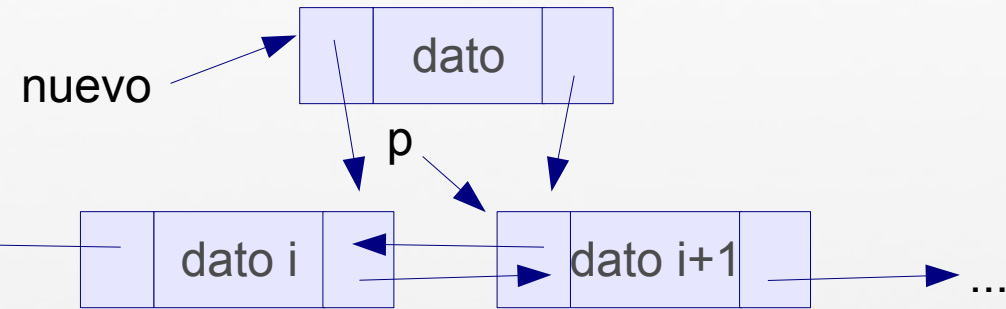
```
// Caso especial: si la lista estaba vacía,  
// poner la cola apuntando al nodo  
if (cabecera == 0)  
    cabecera = nuevo;  
  
if (cola != 0)  
    cola->sig = nuevo  
cola = nuevo
```



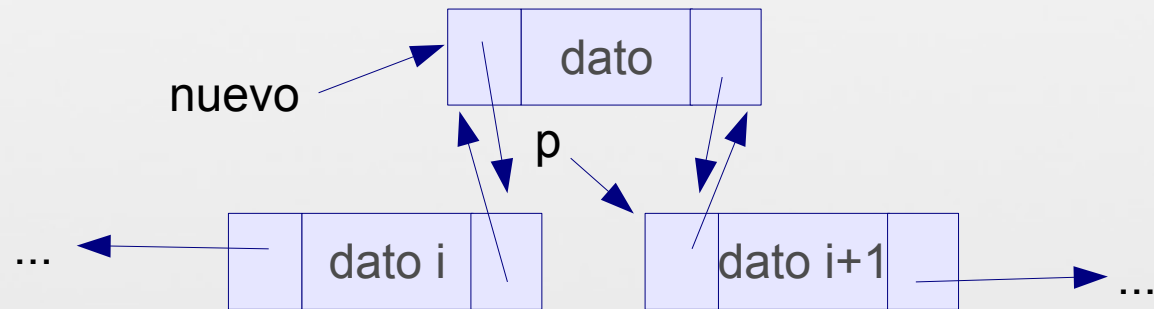


Inserción en medio

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



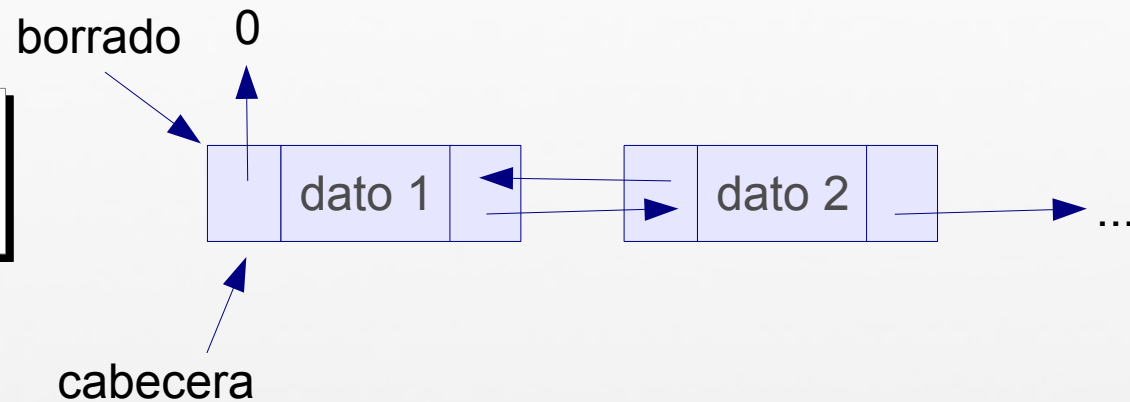
```
p->ant->sig = nuevo;  
p->ant = nuevo;
```



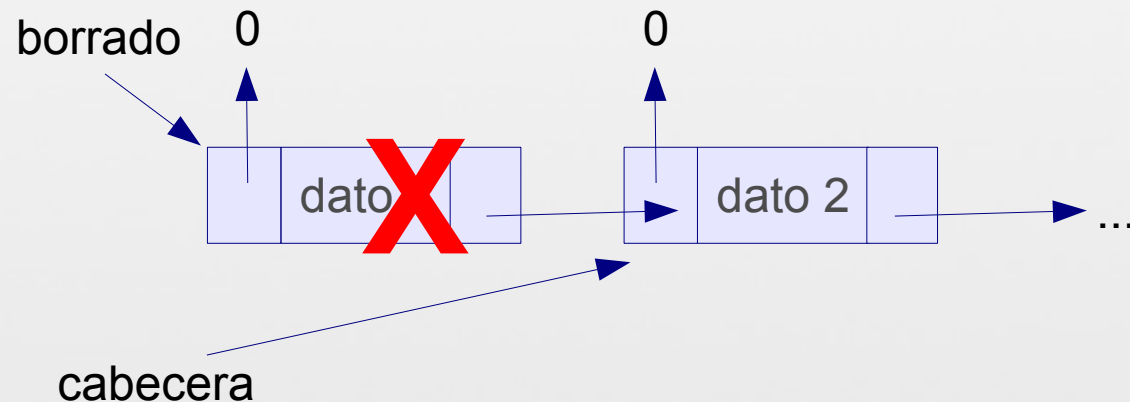


Borrar el primer nodo

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



```
cabecera = cabecera->sig  
delete borrado  
  
if (cabecera != 0)  
    cabecera->ant = 0  
else  
    cola = 0
```



Borrados al final y en medio



- El borrado al final es simétrico al borrado del primer nodo
- El borrado en medio tampoco plantea especiales dificultades
- Ejercicio propuesto: resolver en papel el borrado en medio y al final



Iteración

- Los iteradores en una lista enlazada soportan las dos direcciones

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

    bool hayAnterior() { return nodo != 0; }
    bool haySiguiente() { return nodo != 0; }
    void anterior() { nodo = nodo->ant; }
    void siguiente() { nodo = nodo->sig; }

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

La interfaz de la clase ListaDEnlazada



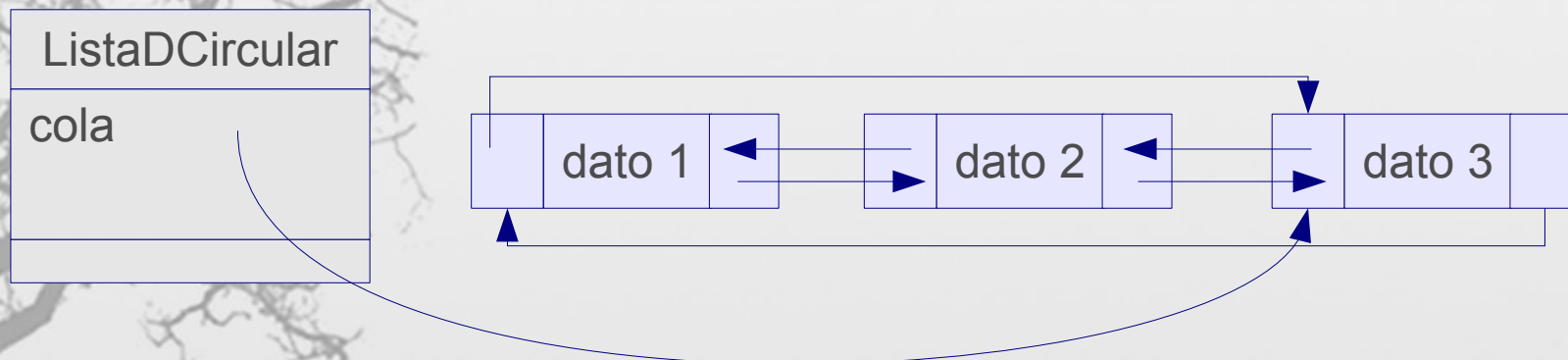
```
template<class T>
class ListaDEnlazada {
    Nodo<T> *cabecera, *cola;
public:
    ListaDEnlazada() : cabecera(0), cola(0) {}
    ~ListaDEnlazada();
    ListaDEnlazada(const ListaDEnlazada &l);
    ListaDEnlazada &operator=(ListaDEnlazada &l);

    Iterador<T> iteradorInicio() { return Iterador<T>(cabecera); }
    Iterador<T> iteradorFinal() { return Iterador<T>(cola); }
    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() { return cabecera->dato; }
    T &final() { return cola->dato; }
};
```



Listas circulares

- Es una lista simple o doblemente enlazada donde todos los nodos tienen puntero al siguiente
- Basta con un puntero al último nodo. El primer nodo se accede como el siguiente al último
- Útil para representar:
 - Listas de procesos que comparten el acceso a un recurso por turno
 - Elementos circulares por naturaleza, como la lista de vértices de un polígono





Matrices dispersas


- Una matriz dispersa es una matriz de un tamaño grande formada básicamente por valores nulos
- Aparecen en numerosos problemas de ciencia e ingeniería

```
00000300000000002000000001000000000
01000000000000000000000000000000000
00000000000000000000000000000000000
00000000000000000000000000000000000
00000300000000000000000000000000000
002000090000000000000000000000000200
000000000000000000000000000000600000
00000000000000000000000000000000000
00100000000005000000000000000000000
00000000000000000000000000000000000
00000000000000000000000000000000000
0000000000000000000000000000000200
```

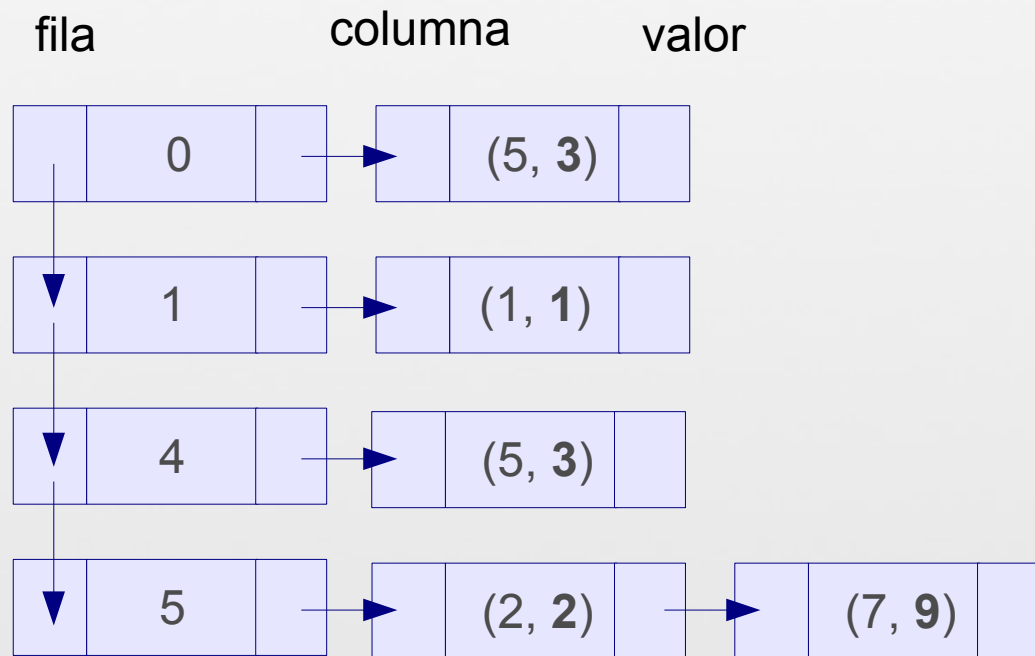
Matrices dispersas como listas de listas



- Guardar todos los valores en una matriz bidimensional es ineficiente
- Una mejor implementación guarda sólo los valores no nulos en una lista de listas



```
00000300000
01000000000
00000000000
00000000000
00000300000
00200009000
00000000000
```



Interfaz de la clase MatrizDispersa



```
class ColMatrizDispersa {
public:
    int col;
    float val;

    ColMatrizDispersa(inta aCol, float aVal) :
        col(aCol), val(aVal) {}
};

class FilaMatrizDispersa {
    ListaEnlazada<ColMatrizDispersa> columnas;

    Iterador<ColMatrizDispersa> buscar(int columna);

public:
    int fila;

    FilaMatrizDispersa(int aFila): fila(aFila), columnas() {}

    float valor(int columna);
    void cambiarValor(int columna, float valor);
};
```


Interfaz de la clase MatrizDispersa



```
class MatrizDispersa {
    int maxFilas, maxColumnas;
    ListaEnlazada<FilaMatrizDispersa> filas;

    Iterador<FilaMatrizDispersa> buscar(int fila);

public:
    MatrizDispersa(int filas, int columnas):
        maxFilas(filas), maxColumnas(columnas) {}

    float valor(int fila, int columna);
    void cambiarValor(int fila, int columna, float valor);

    int getNumFilas() { return maxFilas; }
    int getNumColumnas() { return maxColumnas; }
};
```

Implementación de la Matriz Dispersa

```
Iterador FilaMatrizDispersa::buscar(int columna) {  
    Iterador<ColMatrizDispersa> i = columnas.iterador();  
    while (i.haySiguiente() && i.dato().col < columna) {  
        i.siguiente();  
    }  
    return i;  
}
```

```
float FilaMatrizDispersa::valor(int columna) {  
    Iterador<ColMatrizDispersa> i = buscar(columna);  
  
    if (i.haySiguiente() && i.dato().col == columna) {  
        return i.dato().val;  
    }  
  
    return 0;  
}
```

```
void FilaMatrizDispersa::cambiarValor(int columna, float valor) {  
    Iterador<ColMatrizDispersa> i = buscar(columna);  
  
    if (i.haySiguiente() && i.dato().col == columna) {  
        i.dato().val = valor;  
    }  
    else {  
        columnas.insertar(i, ColMatrizDispersa(columna, valor));  
    }  
}
```

Implementación de la Matriz Dispersa

```
Iterador<FilaMatrizDispersa> MatrizDispersa::buscar(int fila) {
    Iterador<FilaMatrizDispersa> i = filas.iterador();
    while (i.haySiguiete() && i.dato().fila < fila) {
        i.siguiete();
    }

    return i;
}

float MatrizDispersa::valor(int fila, int columna) {
    Iterador<FilaMatrizDispersa> i = buscar(fila);

    if (i.haySiguiete() && i.dato().fila == fila) {
        return i.dato().valor(columna);
    }

    return 0;
}

void MatrizDispersa::cambiarValor(int fila, int columna, float
valor) {
    Iterador<FilaMatrizDispersa> i = buscar(fila);

    if (!i.haySiguiete() || i.dato().fila != fila) {
        filas.insertar(i, FilaMatrizDispersa(fila));
    }

    i.dato().cambiarValor(columna, valor);
}
```

Conclusiones

(listas doblemente enlazadas)



- Las listas doblemente enlazadas son una estructura de datos flexible y eficiente para aplicaciones que requieren inserciones y borrados arbitrarios
- Complejas comparadas con un vector dinámico
- Consumo de memoria alto al requerir dos punteros por dato almacenado



Conclusiones

(listas circulares y mat. dispersas)

- Las listas circulares son listas especializadas útiles para ciertas aplicaciones
- Las matrices dispersas pueden ser implementadas con un consumo de memoria mínimo mediante listas de listas que únicamente guardan los datos relevantes