

# **Filas (Queue)**

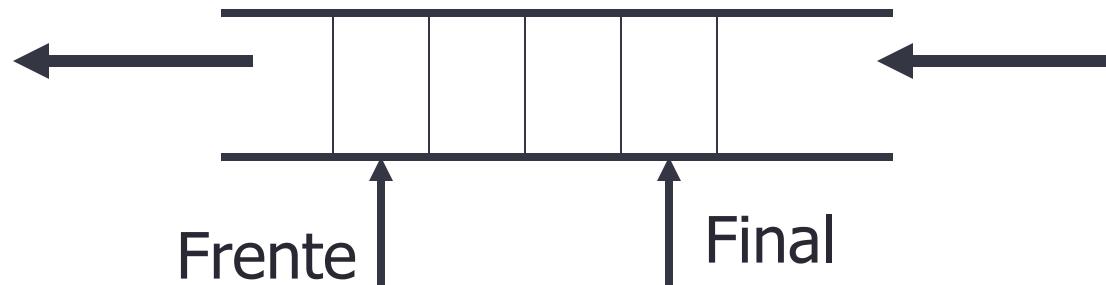
# **Listas Dblemente Encadenadas**

## **Dequeue**

Ing. Armandina Leal Flores

# Filas

- Estructura de datos **lineal**.
- El orden de entrada sigue la filosofía:  
**FIFO** (First Input First Output).



# Uso de una Fila

- ▶ En cualquier aplicación que requiera controlar el orden de entrada.
- ▶ Ejemplos:
  - Simulaciones
  - Buffer de entrada de datos
  - Fila de impresión
  - etc.

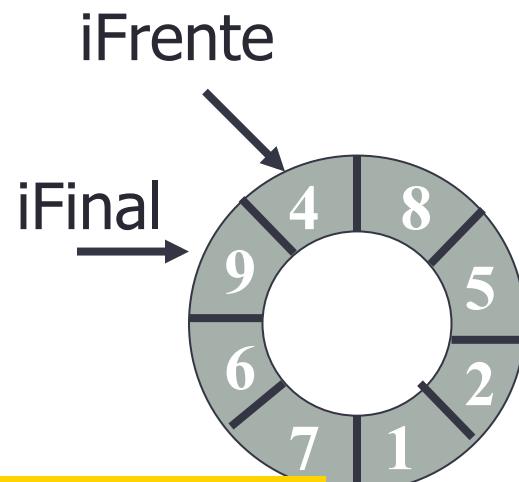
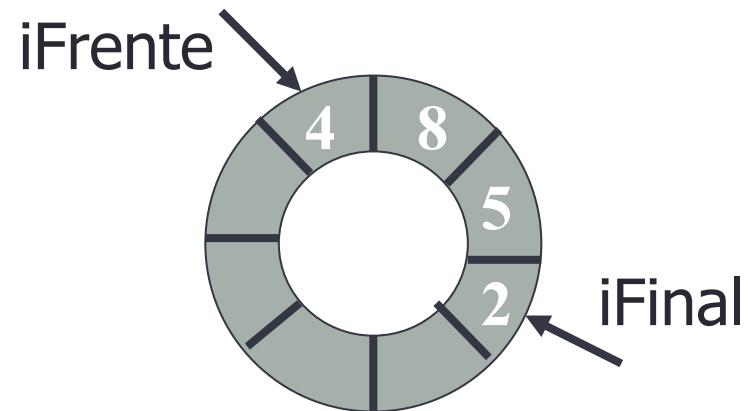
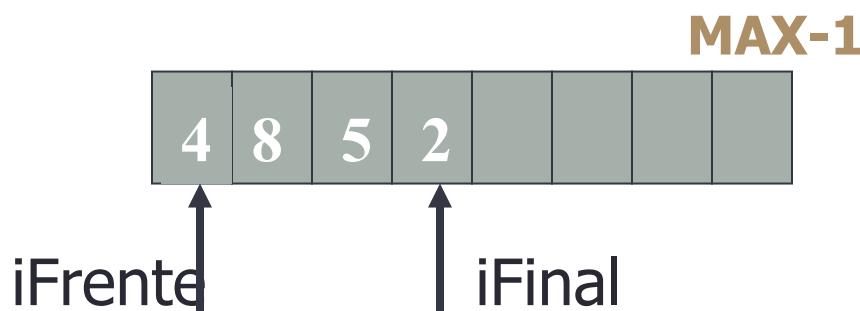
# ADT Fila

- ▶ **Elementos:** El tipo de elemento depende de la aplicación.
- ▶ **Estructura:** Lineal.

Operaciones:

crearFila  
meterFila  
sacarFila  
filaVacía  
filaLlena  
observaFila

# Implementación en Arreglos

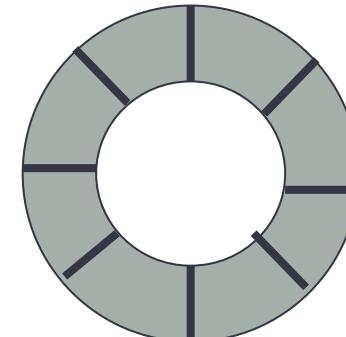


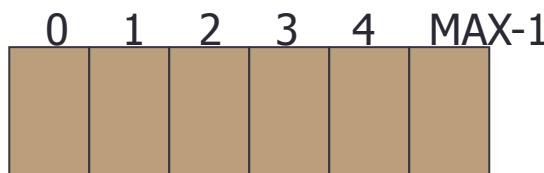
**Fila Llena**

**Fila Vacía**

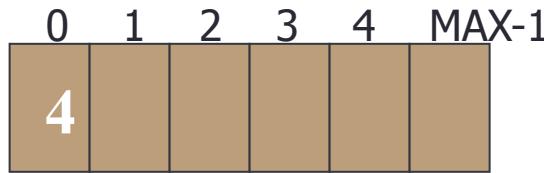
**Fila Vacía**

**iFrente = iFinal = -1**

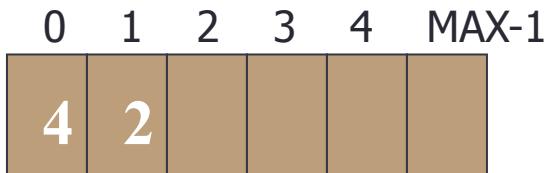




iFrente = iFinal



iFrente = iFinal



iFrente ↑      iFinal ↑



iFrente ↑      iFinal ↑

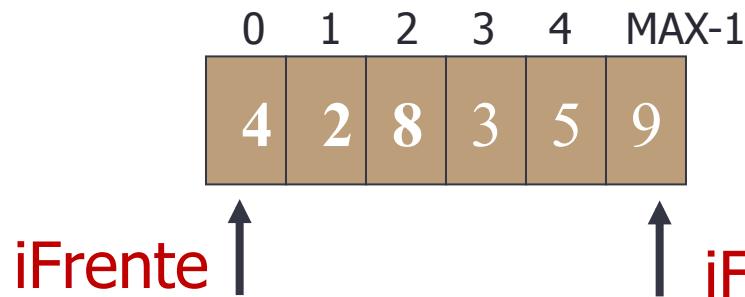
Fila

Fila vacía

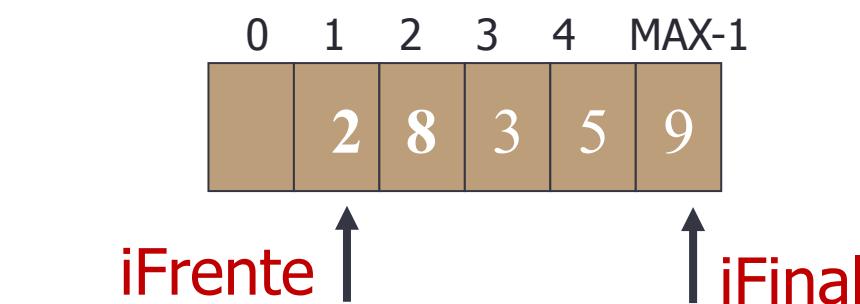
Meter a la fila el 4

Meter a la fila el 2

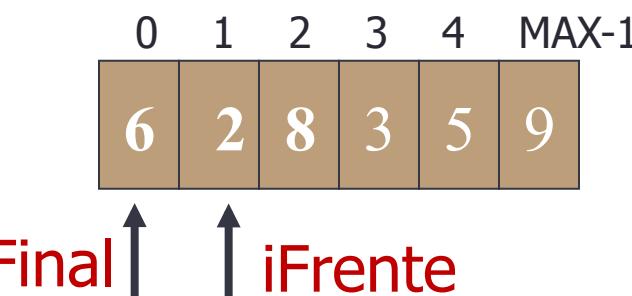
Meter a la fila el 8



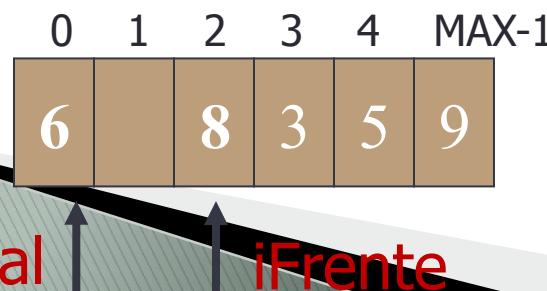
Meter a la fila el 9



Sacar de la fila



Meter a la fila el 6



Sacar de la fila

# Fila en Arreglos

```
const int MAX = 50;  
class FilaArreglos  
{ public:  
    filaArreglos( );  
    void meterFila(int iValor);  
    void sacarFila();  
    int observaFila();  
    bool filaVacia( );  
    bool filaLlena( );  
private:  
    int iArrDatos[MAX];  
    int iFrente, iFinal;  
};
```

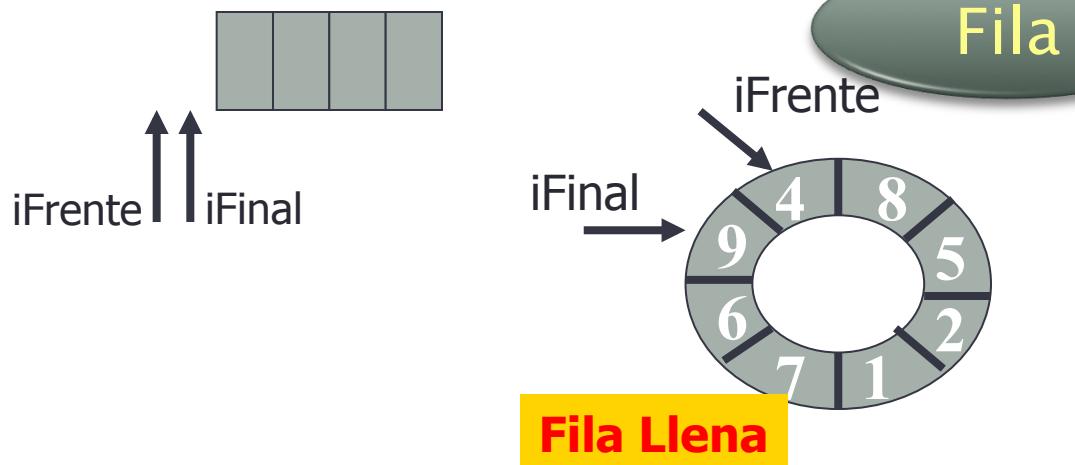
MAX es la cantidad máxima de elementos

Lugar donde se guardan los elementos

iFrente indica donde está el primer elemento que entró a la fila

iFinal contiene la ubicación del último elemento que entró a la fila.

```
filaArreglos () {  
    iFrente = -1;  
    iFinal = -1;  
}
```

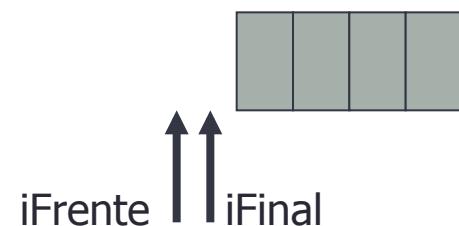


**(iFinal+1==iFrente) || (iFrente==0 && iFinal==MAX-1);**

```
bool filaLlena()  
{ return (iFinal + 1) % MAX == iFrente; }
```

También puede ser  
*iFinal == -1*

```
bool filaVacia()  
{ return iFrente == -1; }
```



## Fila

```
iFinal++;
if (iFinal > MAX-1) iFinal = 0;
```

```
void meterFila (int iDato)
```

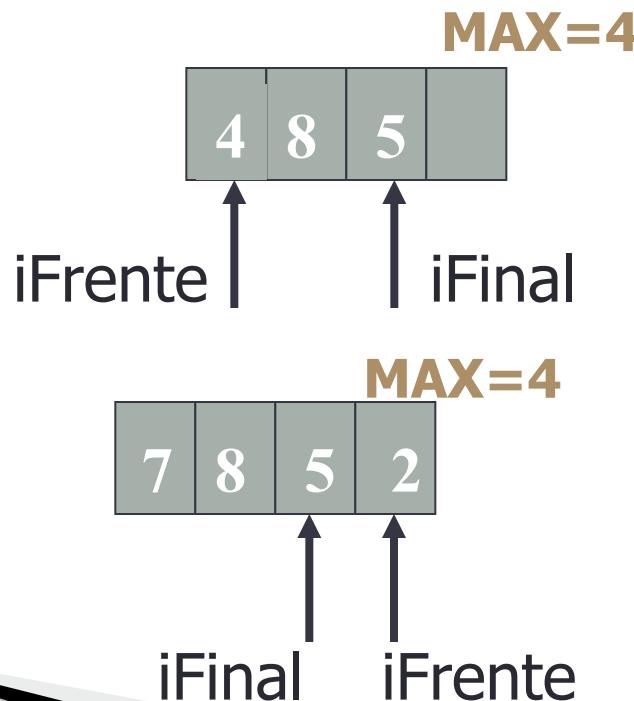
```
{
```

```
    iFinal = (iFinal + 1 ) % MAX;
    iArrDatos[iFinal] = iDato;
    if (iFrente == -1) iFrente++;
}
```



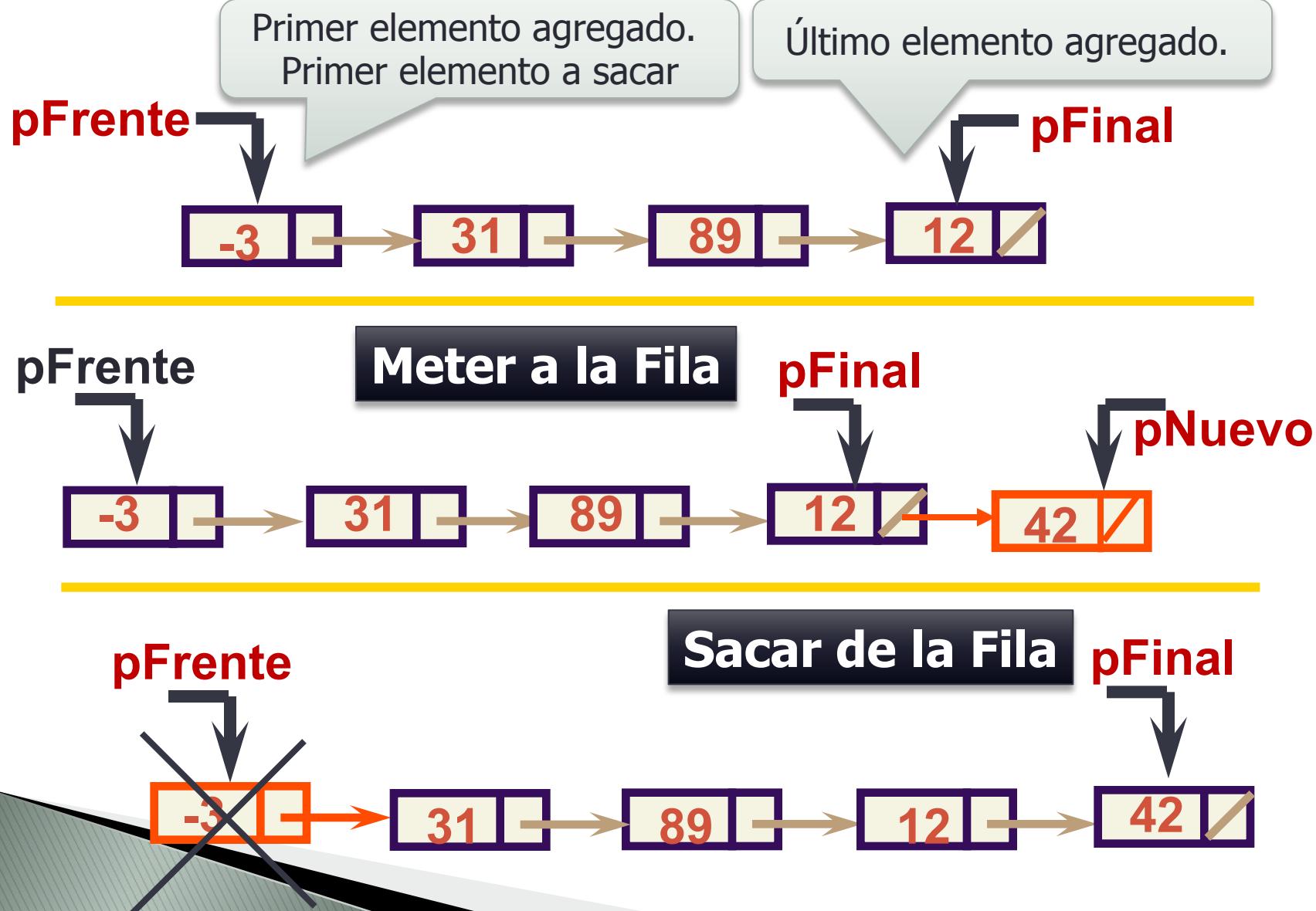
```
void sacarFila(){  
    if (iFrente == iFinal)  
        iFrente = iFinal = -1;  
    else iFrente = (iFrente + 1) % MAX;  
}
```

```
iFrente++;  
if (iFrente > MAX-1) iFrente = 0;
```

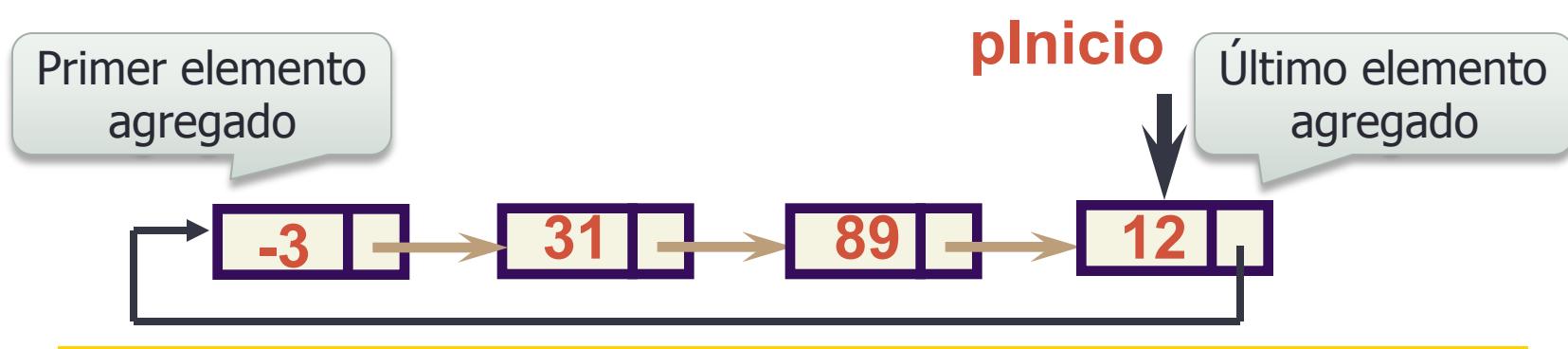


```
int observaFila(){  
    return iArrDatos[iFrente];  
}
```

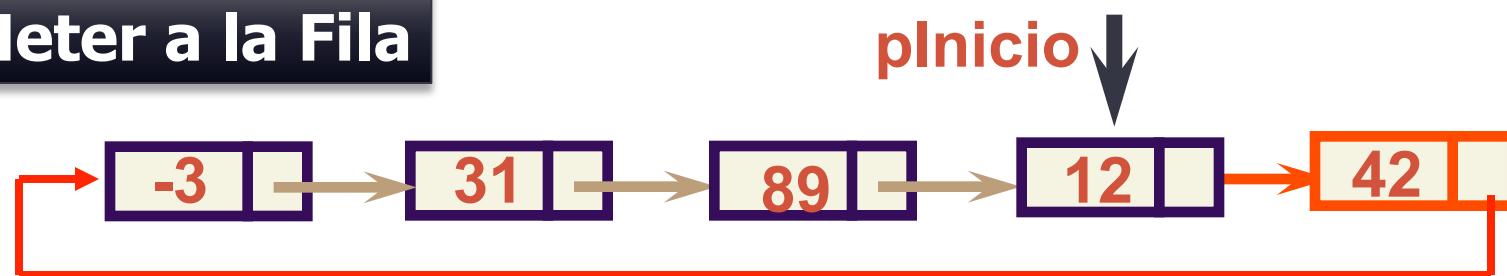
# Implementación en una Lista Encadenada Lineal



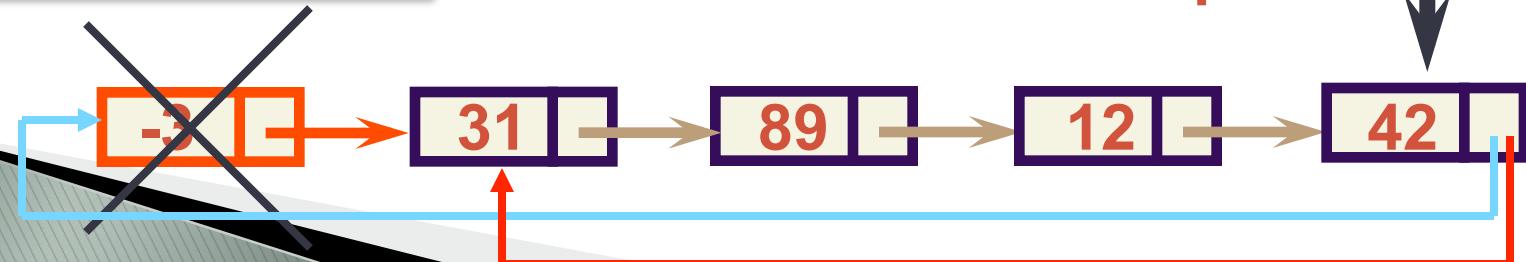
# Implementación en una Lista Encadenada Circular



## Meter a la Fila

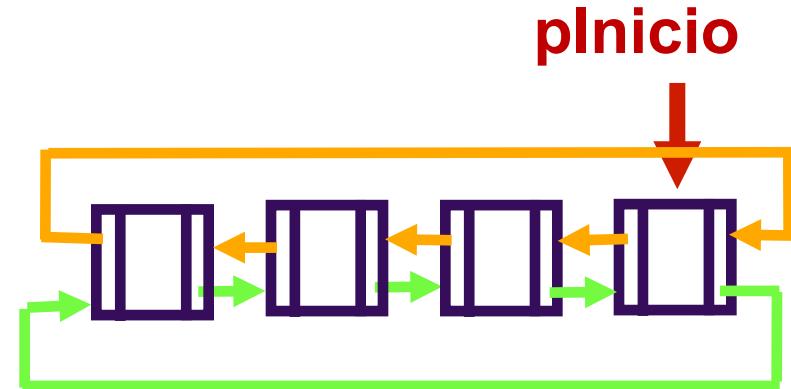
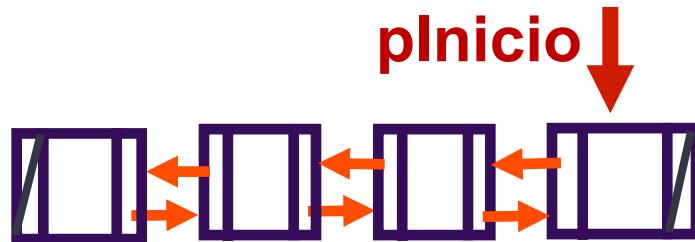


## Sacar de la Fila



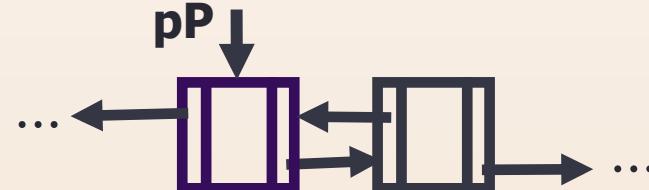
# Listas Dblemente Encadenadas

- ▶ Una lista doblemente encadenada es una lista lineal o circular en la que **cada nodo tiene dos enlaces**, uno al nodo siguiente, y otro al nodo anterior.
- ▶ Pueden recorrerse en ambos sentidos a partir de cualquier nodo.

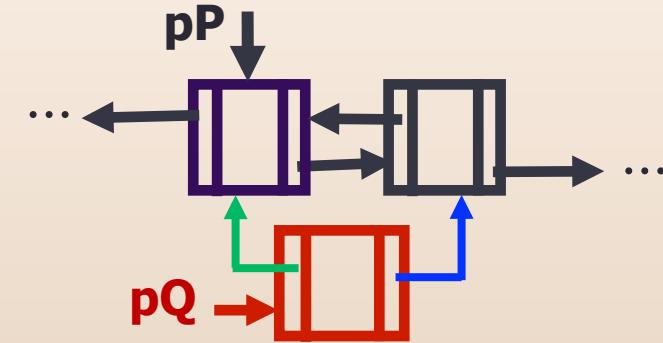


# Ejercicio...

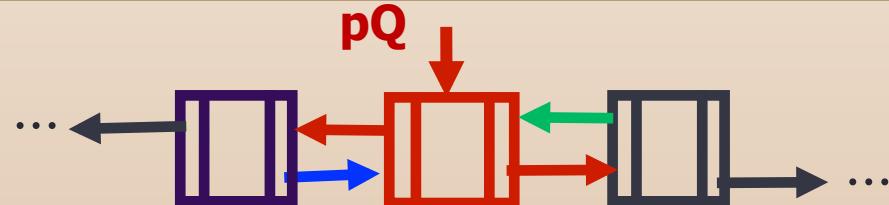
## Agregar nodo después del nodo P



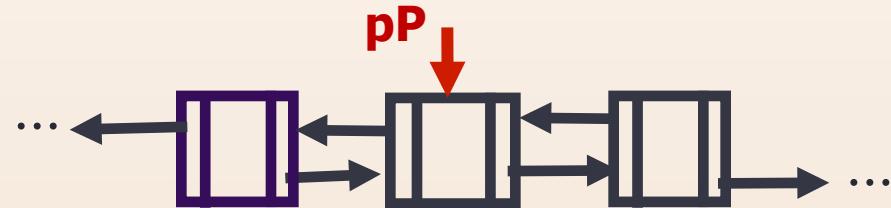
```
NodoDoble *pQ;
pQ = new NodoDoble(
    iDato,
    pP->pSig,
    pP);
```



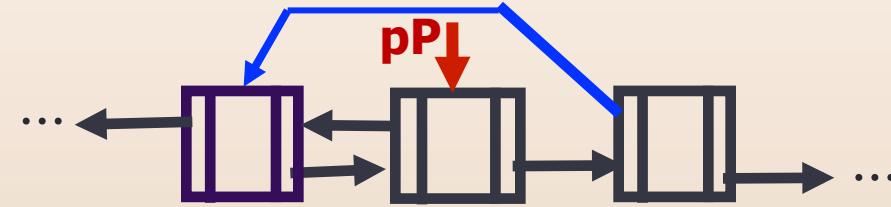
```
pQ->pSig->pAnt= pQ;
pQ->pAnt->pSig = pQ;
```



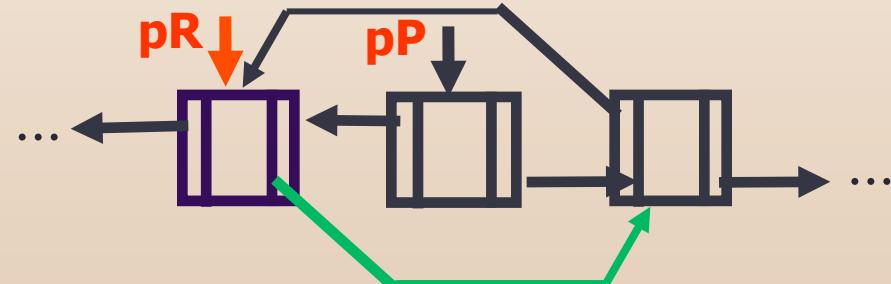
# Ejercicio: Borrar el nodo P



$pP \rightarrow pSig \rightarrow pAnt = pP \rightarrow pAnt;$



$pP \rightarrow pAnt \rightarrow pSig = pP \rightarrow pSig;$   
delete pP;

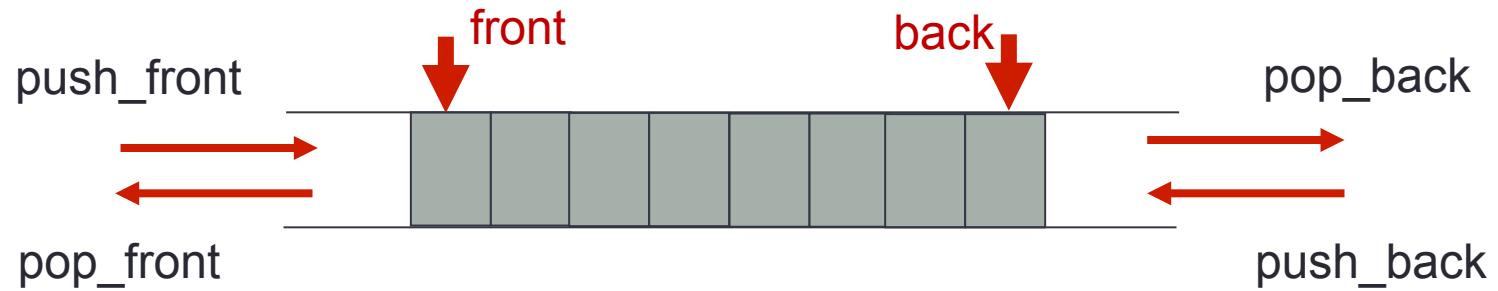


# Deque

- ▶ Estructura de datos lineal en la que los elementos pueden ser agregados o borrados por cualquiera de los extremos de la estructura.
- ▶ Esta estructura se puede implementar en una Lista Dblemente Encadenada Lineal.



# Operaciones



# Implementación en Lista Dblemente Encadenada Lineal

```
class Nodo
{ public:
    Nodo *sig, *ant;
    int info;
    Nodo(void) : sig(0), ant(0) { }
    Nodo(int Dato):
        info(Dato), sig(0), ant(0) { }
    Nodo(int Dato,
          Nodo* siguiente,
          Nodo* anterior) :
        info(Dato),
        sig(siguiente),
        ant(anterior) { }
};
```

```
class deque
{ public:
    deque( ) { }
    deque (deque const& D) { }
    ~deque( ) { }
    void push_front(int Valor) { }
    void push_back(int Valor) { }
    void pop_front () { }
    void pop_back () { }
    bool empty ( ) { }
    int top_front() { }
    int top_back() { }
    deque operator= (const deque& D) { }
private:
    Nodo *inicio, *final;
};
```

# Constructor, Destructor, Copy Constructor

```
deque( ):
    inicio(0), final(0) { }
```

```
~deque( )
{
    while ( !empty() )
        pop_back();
}
```

**deque (const deque& D)**

**{**

**inicio = final = NULL;**

**Nodo \*P;**

**P = D.inicio;**

**while ( P != NULL )**

**{ push\_back( P -> info );**

**P = P -> sig;**

**}**

**}**

¿Qué sucede  
si quitamos  
esta  
instrucción?

¿Qué pasa si en  
lugar de  
**P != NULL**  
escribimos !  
**D.empty()** ?

# push, pop, empty

```
void push_back (int Valor)
{ Nodo *P;
  P = new Nodo (Valor);
  if (final != NULL)
  { P -> ant = final;
    final -> sig = P;
    final = P;
  }
  else inicio = final = P;
}
```

```
void push_front (int Valor)
{ Nodo *P;
  P = new Nodo (Valor);
  if (inicio != NULL)
  { P -> sig = inicio;
    inicio -> ant = P;
    inicio = P;
  }
  else inicio = final = P;
}
```

```
void pop_back ()
{ Nodo *P;
  P = final;
  final = final -> ant;
  if (final == NULL) inicio = NULL;
  delete P;
}
```

```
void pop_front ()
{ Nodo *P;
  P = inicio;
  inicio = inicio -> sig;
  if (inicio == NULL) final = NULL;
  delete P;
}
```

```
int top_back ()
{ return final -> info;
}
```

```
int top_front ()
{ return inicio -> info;
}
```

```
bool empty ()
{ return (inicio == NULL);
}
```

# Operador de Asignación

**deque operator= (const deque& D)**

{

**if (this != &D )**  
{

Evita que se borren los valores cuando se asigna a sí mismo: A = A;

**while ( !empty() )**  
**pop\_front();**

Vacia la deque. Importante si la deque tiene elementos.

Ej. deque X (A); X = P;

**if (D.inicio != NULL )**  
{

Hace una copia de la deque D sólo si tiene elementos.

**Nodo \*P = D.inicio;**

**while (P != NULL)**

{ **push\_back( P -> info );**

**P = P -> sig;**

}

}

**return \*this;**

Regresa la deque. Importante para que funcione correctamente las asignaciones múltiples: A = B = C;

}