Grado en Ingeniería Información

Estructura de Datos y Algoritmos

Sesión 7

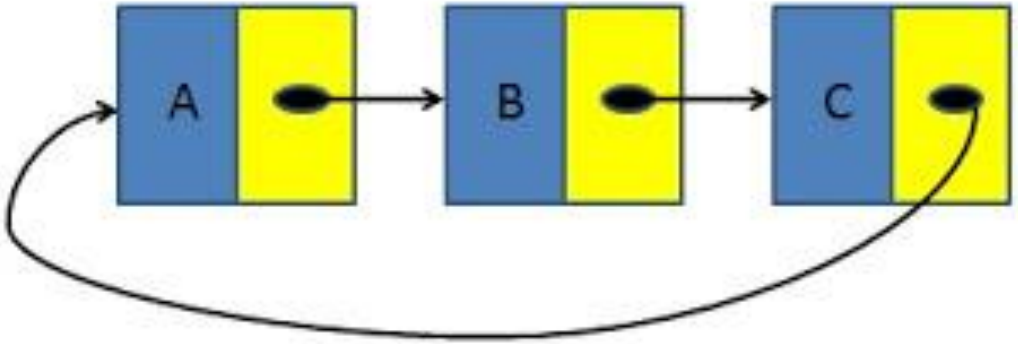
Curso 2022-2023

Marta N. Gómez

UNIVERSIDAD NEBRIJA

T3. Tipos Abstractos de Datos (TAD)

- Concepto.
- Tipos de datos lineales:
  - Pilas
  - Colas
  - **Listas**

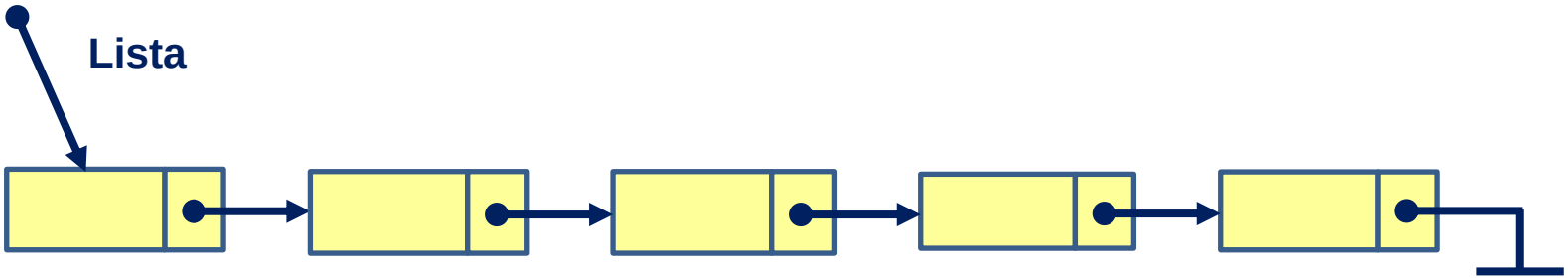


UNIVERSIDAD NEBRIJA

ÍNDICE

Una **lista** es una **colección ordenada** (**orden relativo**) de **elementos homogéneos** en la que **NO HAY restricciones para acceder** a los mismos, pudiendo **Añadir, Modificar o Eliminar** elementos en **cualquier posición** de la misma.

Es una estructura de datos **Lineal.**



Operaciones Básicas de Lista

empty: **Determina si la lista está vacía o no.**

Precondición: Ninguna.

Postcondición: Decide si la lista tiene elementos o no.

Por tanto, la lista no se modifica.

front: **Devuelve el primer elemento de la lista.**

Precondición: La lista no puede estar vacía.

Postcondición: Obtiene el elemento que ocupa la primera posición.

Por tanto, la lista no se modifica.

push_front: **Inserta un elemento al principio de la lista.**

Precondición: Ninguna.

Postcondición: Añade un elemento más como primero de la lista .

Por tanto, la lista se modifica.

push_back: **Inserta un elemento al final de la lista.**

Precondición: Ninguna.

Postcondición: Añade un elemento más como último de la lista .

Por tanto, la lista se modifica.

pop_front: Elimina el elemento del principio de la lista.

Precondición: La lista no puede estar vacía.

Postcondición: Elimina el primer elemento de la lista .

Por tanto, la lista se modifica.

pop_back: Elimina el último elemento de la Lista.

Precondición: La lista no puede estar vacía.

Postcondición: Elimina el elemento del final de la lista.

Por tanto, la lista se modifica.

erase: Elimina el elemento de una posición de la Lista.

Precondición: La lista no puede estar vacía y la posición debe ser menor o igual al tamaño de la lista.

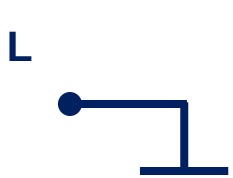
Postcondición: Elimina el elemento de la posición indicada.

Por tanto, la lista se modifica.

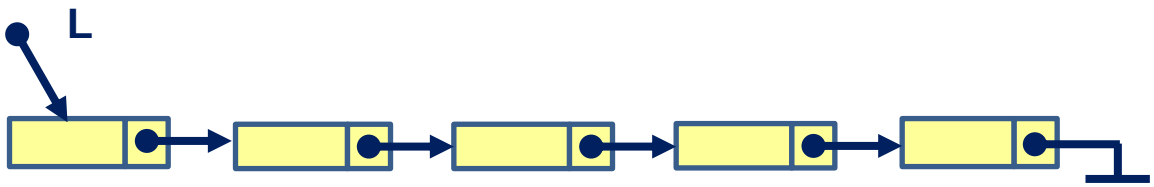
```cpp
#include <iostream>
#include <memory>

using namespace std;

//---------------Clase CDato
class CDato {
    private:
        int n;
    public:
        CDato():n(0){};

        int getN() const;
        void setN(int newN);
};

//---------------Clase Nodo
class Nodo {
    private:
        CDato dato;
        shared_ptr<Nodo> next = nullptr;
    public:
        Nodo(const CDato& d) : dato{d} {};

        const CDato &getDato() const;
        void setDato(const CDato &newDato);
        const shared_ptr<Nodo> &getNext() const;
        void setNext(const shared_ptr<Nodo> &newNext);
};
```

```
//---------------Clase Lista
class Lista {
    private:
        shared_ptr<Nodo> first;
    public:
        Lista():first(nullptr){};

        bool empty() const;
        const CDato &front() const;

        void push_back(const CDato&);
        void push_front(const CDato&);

        void pop_back();
        void pop_front();
        void erase(int pos);


        const shared_ptr<Nodo> &getFirst() const;
        void setFirst(const shared_ptr<Nodo> &newFirst);
};
```

Operación empty



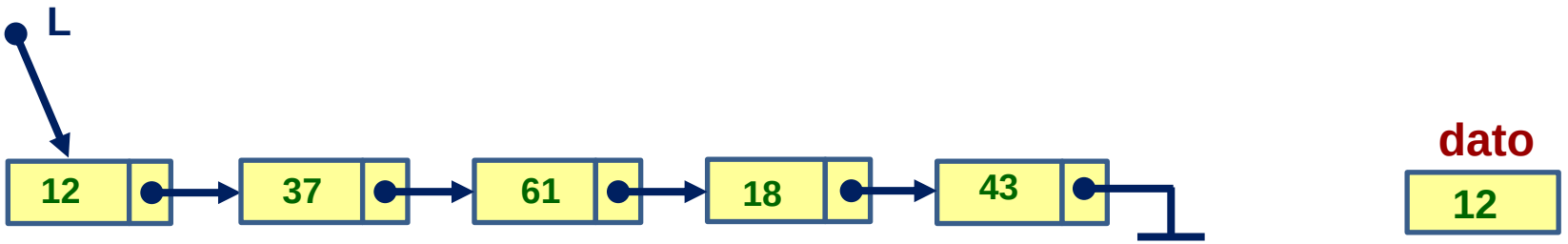
TRUE



FALSE

```
bool Lista::empty() const {
    return (first == nullptr);
}
```

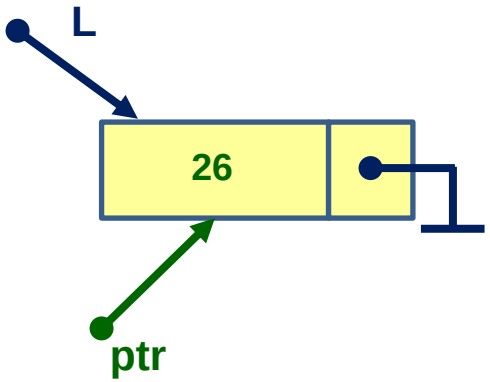
Operación front




```
// Devuelve el dato del primer elemento de la lista
const CDato & Lista::front() const {
    return first->getDato();
}
```

Operación push_ front

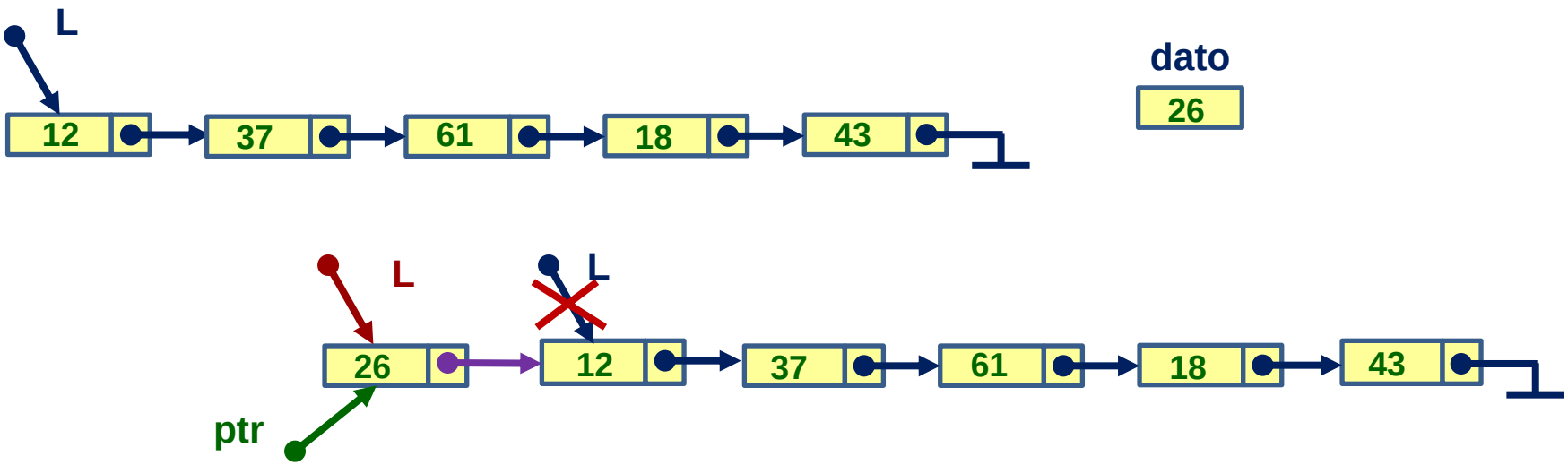
Caso 1º




```
// Inserta un elemento al principio de la lista
void Lista::push_front(const CDato& dato) {
    shared_ptr<Nodo> ptr = make_shared<Nodo>(Nodo{dato});
    if (empty()) {
        first = ptr;  // Solo hay un elemento en la lista
    }
    else {
        ptr->setNext(first);
        first = ptr;
    }
}
```

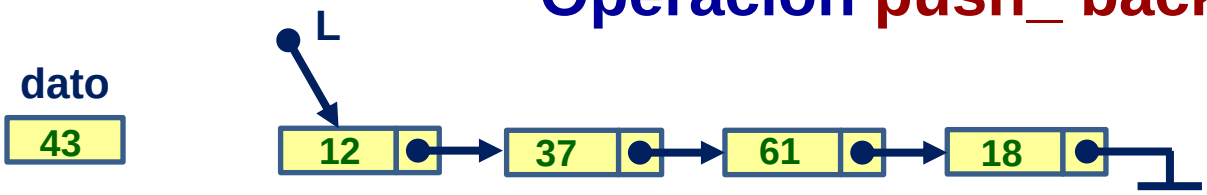
Operación push_ front

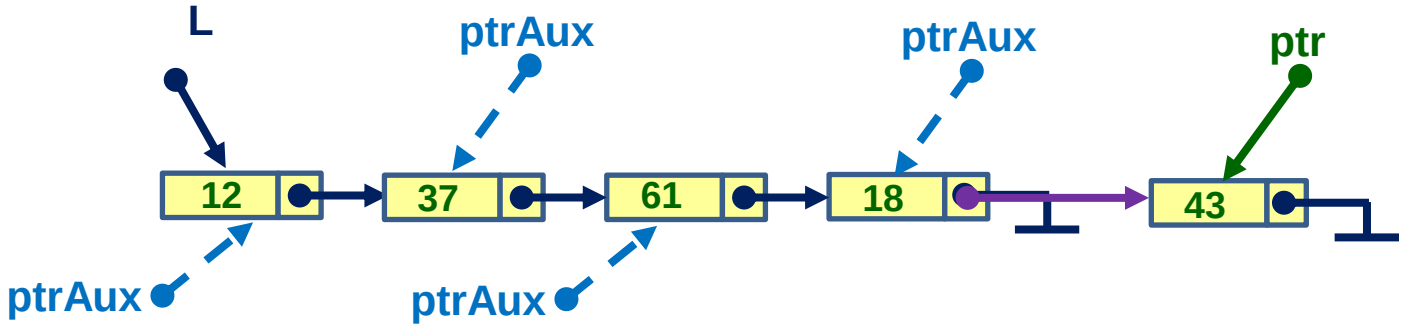
Caso 2º




```
// Inserta un elemento al principio de la lista
void Lista::push_front(const CDato& dato) {
    shared_ptr<Nodo> ptr = make_shared<Nodo>(Nodo{dato});
    if (empty()) {
        first = ptr;  // Solo hay un elemento en la lista
    }
    else {
        ptr->setNext(first);
        first = ptr;
    }
}
```

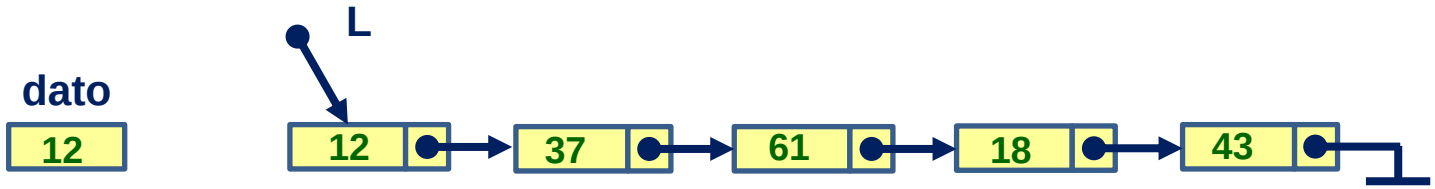
Operación push_ back

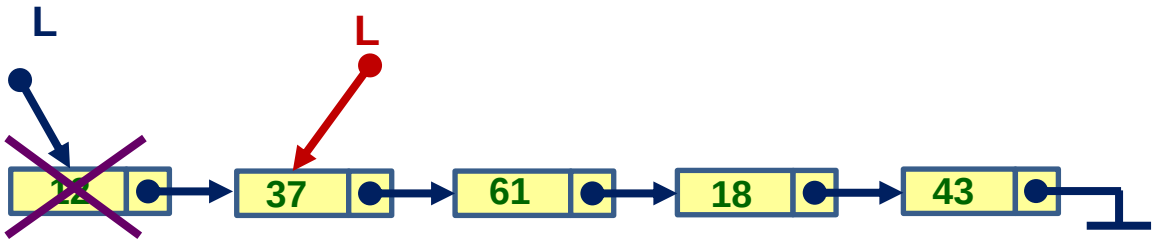





```
// Inserta un nuevo elemento al final de la lista
void Lista::push_back(const CDato& dato) {
    shared_ptr<Nodo> ptr = make_shared<Nodo>(Nodo{dato});

    if (empty()) {
        first = ptr;  //Solo hay un elemento en la lista
    }
    else {
        shared_ptr<Nodo> ptrAux = first;
        while (ptrAux->getNext() != nullptr) {
            ptrAux = ptrAux->getNext();
        }

        ptr->setNext(ptr);
    }
}
```

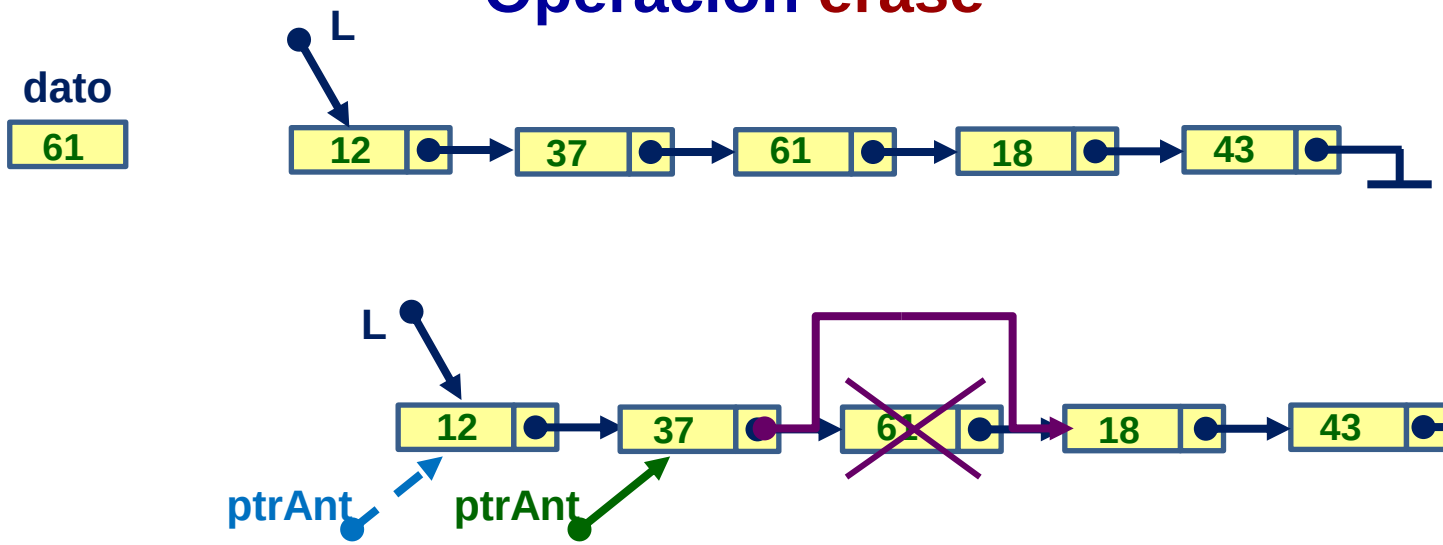
Operación pop_ front





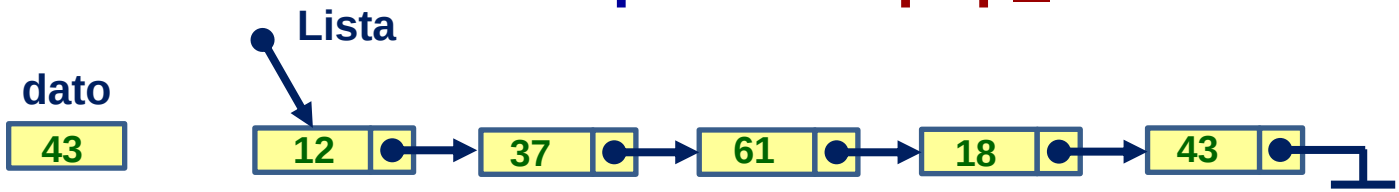

```
// Elimina el primer elemento de la lista
void Lista::pop_front() {
    first = first->getNext();
}
```

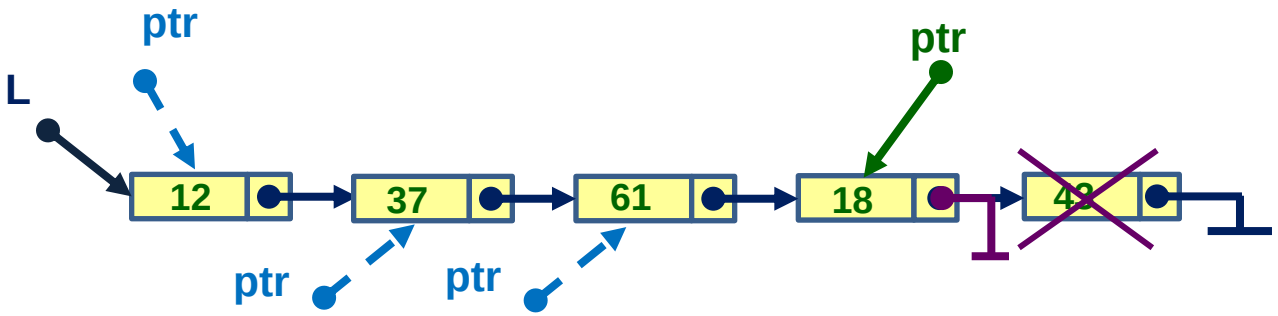
Operación erase




```
// Elimina un elemento de una determinada posiciÃ³n de a lista
void Lista::erase(int pos) {
    if (pos == 1) {
        first = first->getNext();
    }
    else {
        shared_ptr<Nodo> ptrAnt = first;
        int indice = 1;
        while (indice != pos - 1) {
            ptrAnt = ptrAnt->getNext();
            indice++;
        }
        ptrAnt->setNext(ptrAnt->getNext()->getNext());
    }
}
```

Operación pop_ back






```
// Elimina el último elemento de la lista
void Lista::pop_back() {
    if (first->getNext() == nullptr) {
        first = nullptr;  // la lista solo tiene un elemento
    }
    else {
        shared_ptr<Nodo> ptr = first;
        while (ptr->getNext()->getNext() != nullptr) {
            ptr = ptr->getNext();
        }
        ptr->setNext(nullptr);
    }
}
```

EJERCICIOS DE LISTAS

Escribir un programa en C++11, realizando las funciones necesarias, que permita generar y gestionar una **lista** de números enteros utilizando sólo las operaciones básicas de pila.

El programa deberá de mostrar un menú al usuario que le permita hacer los siguientes procesos:

1. **Introducir números enteros en la Lista:**

   Esta opción, ***introducirElem***, consistirá en solicitar un número entero al usuario y añadirlo al principio o al final de la Lista de números, según indique el usuario. Opcionalmente, se puede permitir añadir más valores si el usuario así lo indica.

2. **Mostrar el contenido de la Lista:**

   Esta opción, ***mostrarElemLista*****,** consistirá en mostrar por pantalla los números guardados en la Lista.

   Si la Lista está vacía se deberá de mostrar un mensaje para indicárselo al usuario.

3. **Elemento mayor de la Lista:**

   Esta opción, ***buscarMayorElemLista***, consistirá en mostrar por pantalla un mensaje con el MAYOR de los números almacenados en la Lista. Para ello, realizar una función miembro de la clase Lista que determine el valor de elemento mayor de la Lista.

   Si la Lista está vacía se deberá de mostrar un mensaje para indicárselo al usuario.

4. **Buscar un número en la Lista:**

   Esta opción, ***buscarElemLista***, consistirá en comprobar si un determinado valor solicitado al usuario está guardado en la Listaa.

   Si la Lista está vacía se deberá de mostrar un mensaje para indicárselo al usuario.

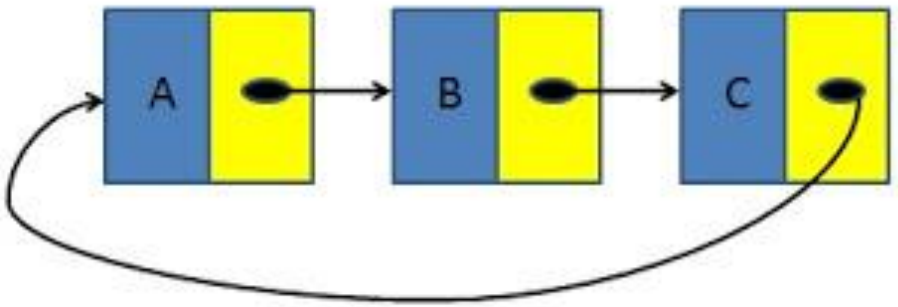
5. **Número de veces que está un número en la Lista:**

   Esta opción, ***calcularVecesElemLista***, consistirá en determinar las veces que un determinado valor solicitado al usuario está en la Lista.

   Si la Lista está vacía se deberá de mostrar un mensaje para indicárselo al usuario.

T3. Tipos Abstractos de Datos (TAD)

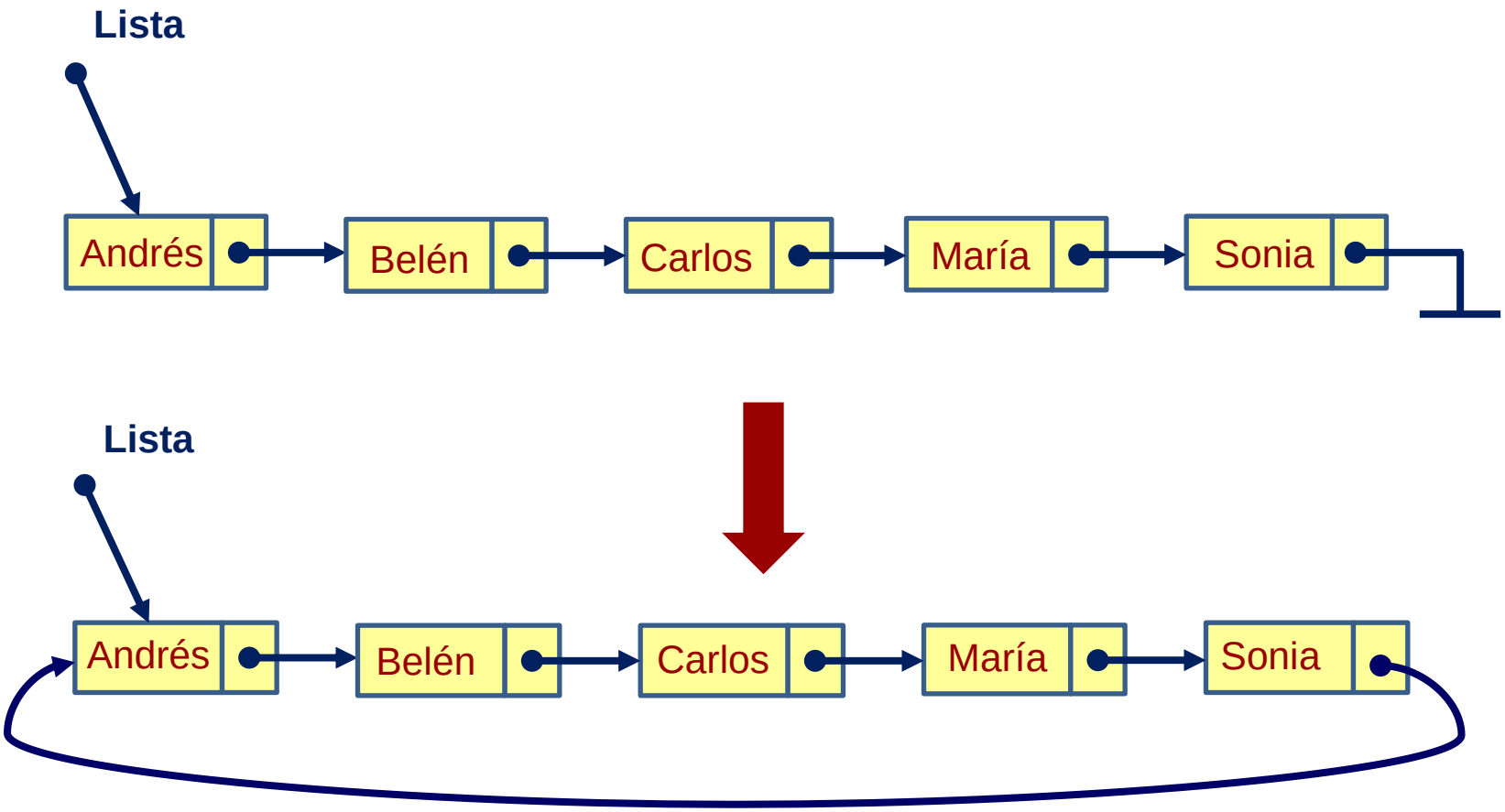
- Concepto.
- Tipos de datos lineales:
  - Pilas
  - Colas
  - **Listas Circulares**



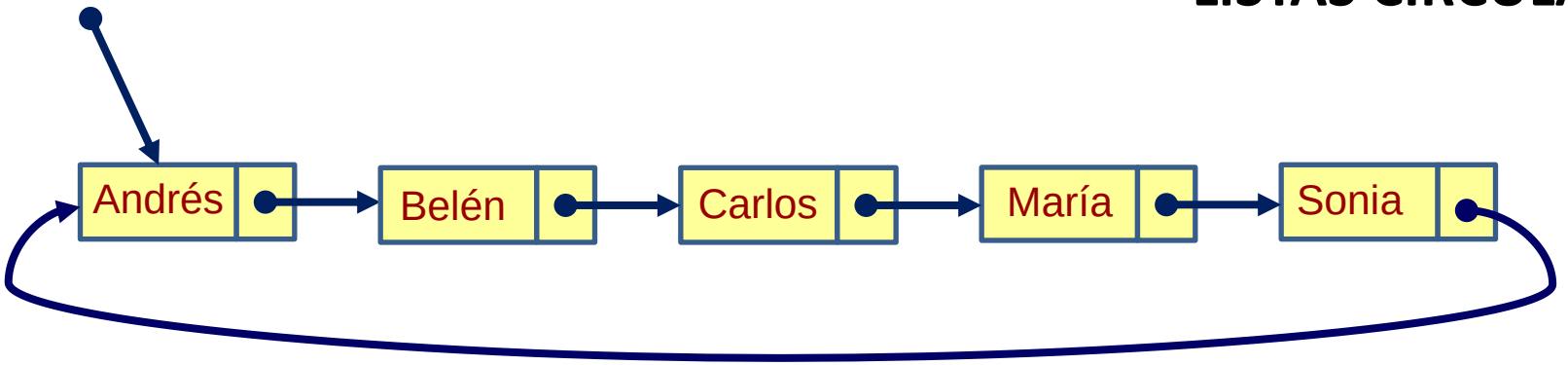
UNIVERSIDAD NEBRIJA

ÍNDICE

Hay que guardar cuál es el **primero** o **último** elemento para evitar un **bucle infinito** cuando se recorre.

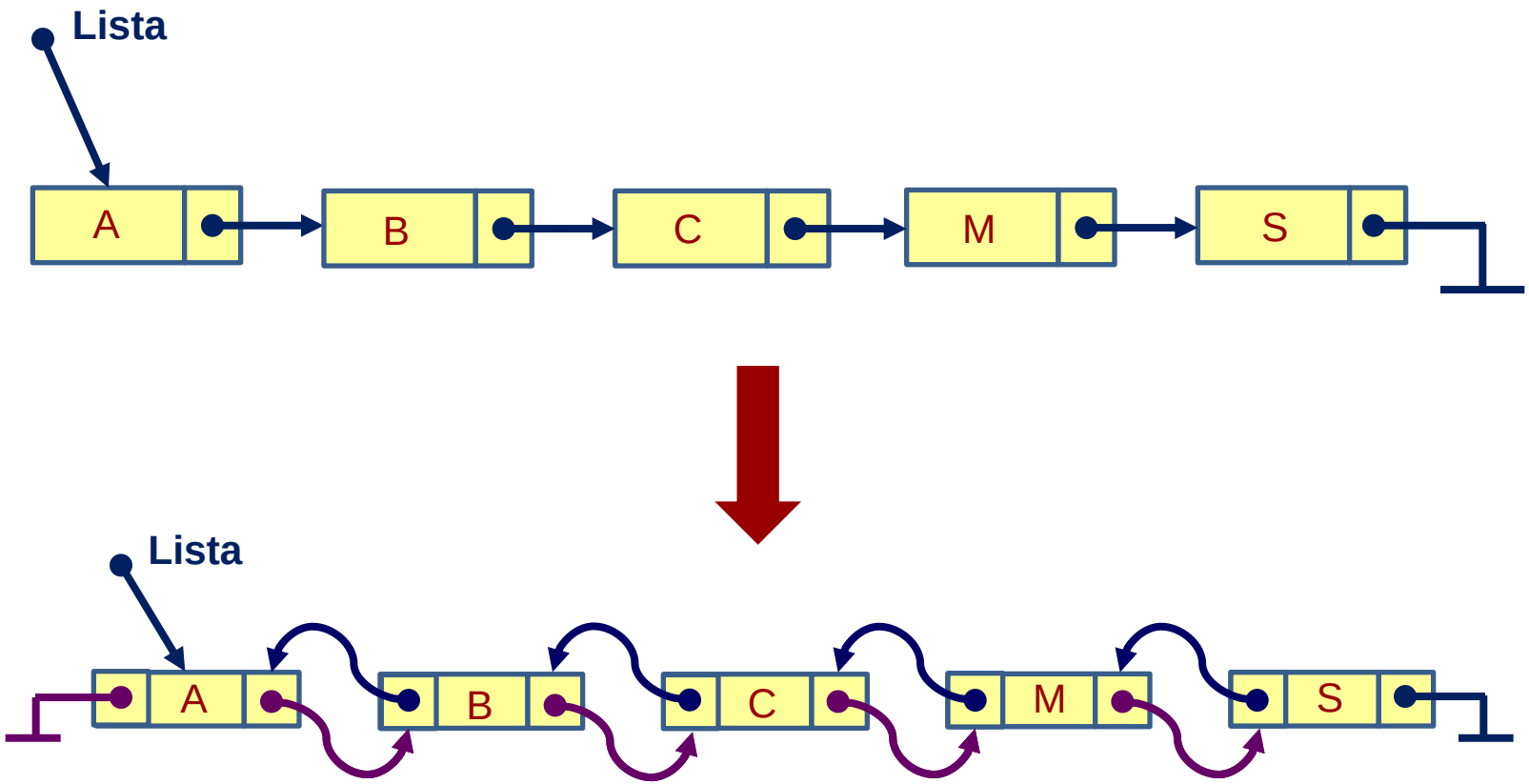


LISTAS CIRCULARES

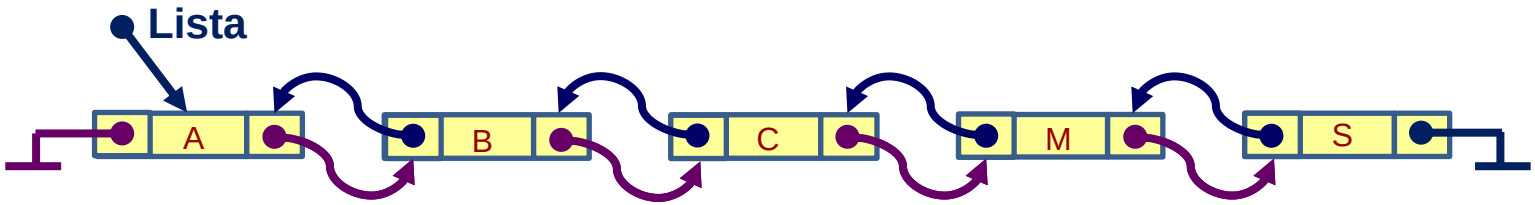



```
// Comprueba si se ha llegado al final de la lista
bool ListaC::serFinalDeLista (shared_ptr<Nodo> const &ptr)
{
    if (empty())
        return true;
    else
        return (first==ptr);
}
```

LISTAS DOBLEMENTE ENLAZADAS



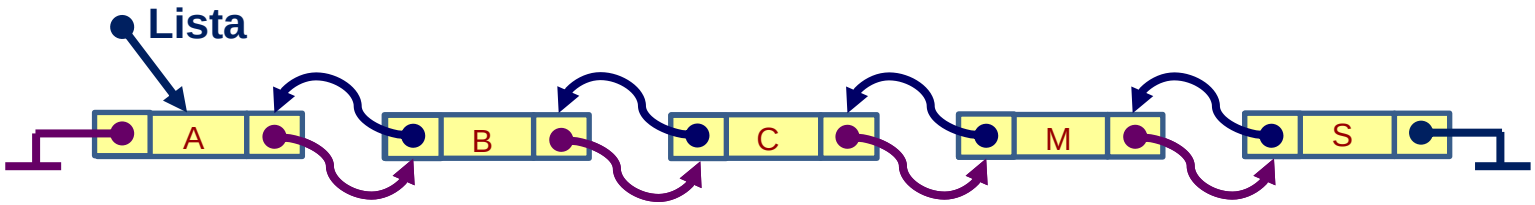
LISTAS DOBLEMENTE ENLAZADAS




```
//---------------Clase CDato
class CDato {
    private:
        int n;
    public:
        CDato():n(0){};

        int getN() const;
        void setN(int newN);
};
```

```
//---------------Clase Nodo
class Nodo {
    private:
        CDato dato;
        shared_ptr<Nodo> next = nullptr;
        shared_ptr<Nodo> prev = nullptr;
    public:
        Nodo(const CDato& d):dato{d} {};

        const CDato &getDato() const;
        void setDato(const CDato &newDato);

        const shared_ptr<Nodo> &getNext() const;
        void setNext(const shared_ptr<Nodo> &newNext);

        const shared_ptr<Nodo> &getPrev() const;
        void setPrev(const shared_ptr<Nodo> &newPrev);
};
```

LISTAS DOBLEMENTE ENLAZADAS




```
//---------------Clase Lista Doble
class ListaD {
    private:
        shared_ptr<Nodo> first;
    public:
        Lista():first(nullptr){};

        bool empty() const;

        void push_back(const CDato&);
        void push_front(const CDato&);
        const CDato &front() const;
        const CDato &back() const;
        void pop_back();
        void pop_front();
        void erase(int pos);

        const shared_ptr<Nodo> &getFirst() const;
        void setFirst(const shared_ptr<Nodo> &newFirst);
};
```