

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

# EL TAD Lista

Manuel Montenegro Montes

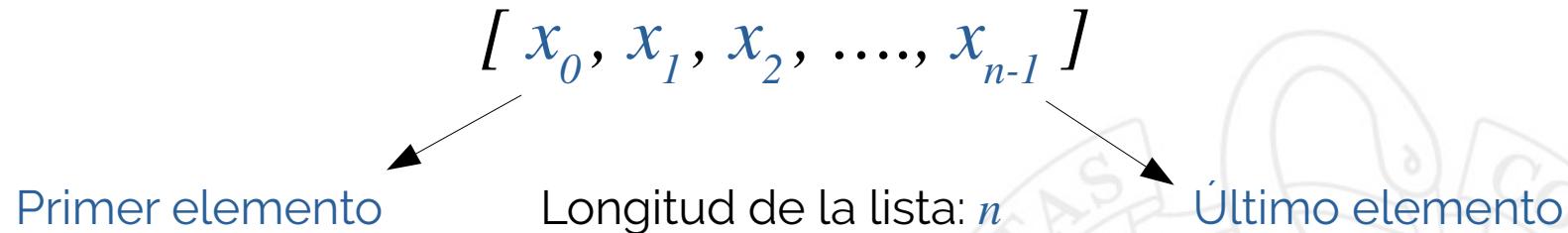
Departamento de Sistemas Informáticos y Computación  
Facultad de Informática – Universidad Complutense de Madrid

# El TAD Lista

- Una **lista** es una colección de elementos, con las siguientes características:
  - Tiene longitud finita.
  - Los elementos siguen un orden secuencial:
    - Existe un primer elemento y un último elemento.
    - Todos los elementos en la lista tienen un predecesor (excepto el primero) y un sucesor (excepto el último).
  - Se permiten elementos duplicados en la lista.

# Modelo del TAD Lista

- Las listas representan **secuencias** de elementos de la siguiente forma:



- Ejemplos:

$[3, 1, 6, 14, 5]$

$[1, 3, 5, 6, 14]$

$[25, 25, 7, 5, 7, 7]$

$[Marta, David, Gerardo]$

$[true]$

$[[1, 0, 0], [0, 1, 0], [0]]$

$[]$

**Lista vacía (longitud 0)**

# Operaciones sobre el TAD Lista

- **Constructoras:**
  - Crear una lista vacía (*create\_empty*).
- **Mutadoras:**
  - Añadir un elemento al principio de la lista (*push\_front*).
  - Añadir un elemento al final de la lista (*push\_back*).
  - Eliminar el elemento del principio de la lista (*pop\_front*).
  - Eliminar el elemento del final de la lista (*pop\_back*).
- **Observadoras:**
  - Obtener el tamaño de la lista (*size*).
  - Comprobar si la lista es vacía (*empty*).
  - Acceder al primer elemento de la lista (*front*).
  - Acceder al último elemento de la lista (*back*).
  - Acceder a un elemento que ocupa una posición determinada (*at*).

# Operaciones constructoras

{ *true* }

***create\_empty()*** → (*L*: *List*)

{ *L* = [ ] }



# Operaciones mutadoras

{  $L = [x_0, \dots, x_{n-1}]$  }

**push\_front**( $x$ : elem,  $L$ : List)

{  $L = [x, x_0, \dots, x_{n-1}]$  }

{  $L = [x_0, x_1, \dots, x_{n-1}], n \geq 1$  }

**pop\_front**( $L$ : List)

{  $L = [x_1, \dots, x_{n-1}]$  }

{  $L = [x_0, \dots, x_{n-1}]$  }

**push\_back**( $x$ : elem,  $L$ : List)

{  $L = [x_0, \dots, x_{n-1}, x]$  }

{  $L = [x_0, \dots, x_{n-2}, x_{n-1}], n \geq 1$  }

**pop\_back**( $L$ : List)

{  $L = [x_0, \dots, x_{n-2}]$  }

# Operaciones observadoras

$\{ L = [x_0, \dots, x_{n-1}] \}$

**size**( $L: List$ )  $\rightarrow$  ( $tamaño: int$ )  
 $\{ tamaño = n \}$

$\{ L = [x_0, \dots, x_{n-1}] \wedge n \geq 1 \}$

**front**( $L: List$ )  $\rightarrow$  ( $e: elem$ )  
 $\{ e = x_0 \}$

$\{ L = [x_0, \dots, x_{n-1}] \wedge 0 \leq i < n \}$

**at**( $i: int, L: List$ )  $\rightarrow$  ( $e: elem$ )  
 $\{ e = x_i \}$

$\{ L = [x_0, \dots, x_{n-1}] \}$

**empty**( $L: List$ )  $\rightarrow$  ( $b: bool$ )  
 $\{ b \Leftrightarrow n = 0 \}$

$\{ L = [x_0, \dots, x_{n-1}] \wedge n \geq 1 \}$

**back**( $L: List$ )  $\rightarrow$  ( $e: elem$ )  
 $\{ e = x_{n-1} \}$

# Operaciones adicionales

- Algunas implementaciones permiten las siguientes operaciones:
  - Mostrar una lista por pantalla.
  - Insertar/eliminar en una posición determinada.
  - Concatenar dos listas.
  - Invertir el orden de los elementos de una lista.
  - Recorrer una lista.

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

# Implementación del TAD Lista mediante arrays

Manuel Montenegro Montes

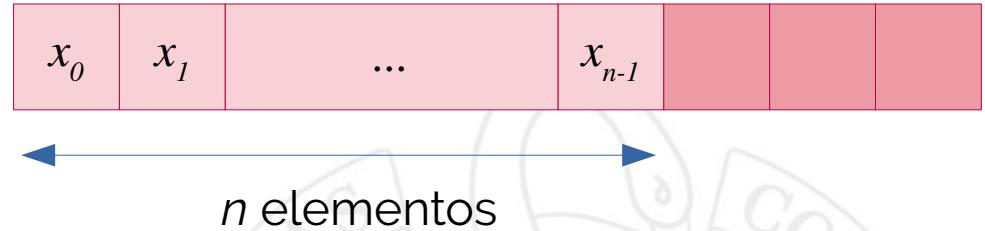
Departamento de Sistemas Informáticos y Computación  
Facultad de Informática – Universidad Complutense de Madrid

# Recordatorio: operaciones del TAD Lista

- **Constructoras:**
  - Crear una lista vacía: ***create\_empty()*** →  $L: List$
- **Mutadoras:**
  - Añadir un elemento al principio de la lista: ***push\_front(x: elem, L: List)***.
  - Añadir un elemento al final de la lista: ***push\_back(x: elem, L: List)***.
  - Eliminar el elemento del principio de la lista: ***pop\_front(L: List)***.
  - Eliminar el elemento del final de la lista: ***pop\_back(L: List)***.
- **Observadoras:**
  - Obtener el tamaño de la lista: ***size(L: List)*** →  $tam: int$ .
  - Comprobar si la lista es vacía ***empty(L: List)*** →  $b: bool$ .
  - Acceder al primer elemento de la lista ***front(L: List)*** →  $e: elem$ .
  - Acceder al último elemento de la lista ***back(L: List)*** →  $e: elem$ .
  - Acceder a un elemento que ocupa una posición determinada ***at(idx: int, L: List)*** →  $e: elem$ .

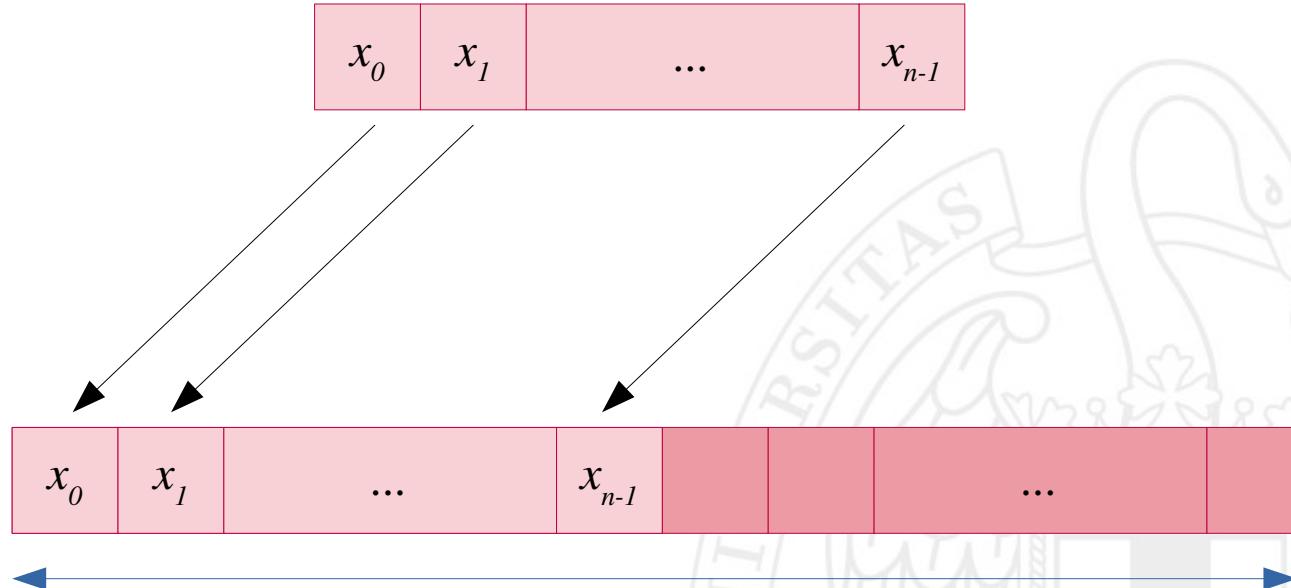
# Implementación mediante arrays

$[x_0, x_1, \dots, x_{n-1}]$



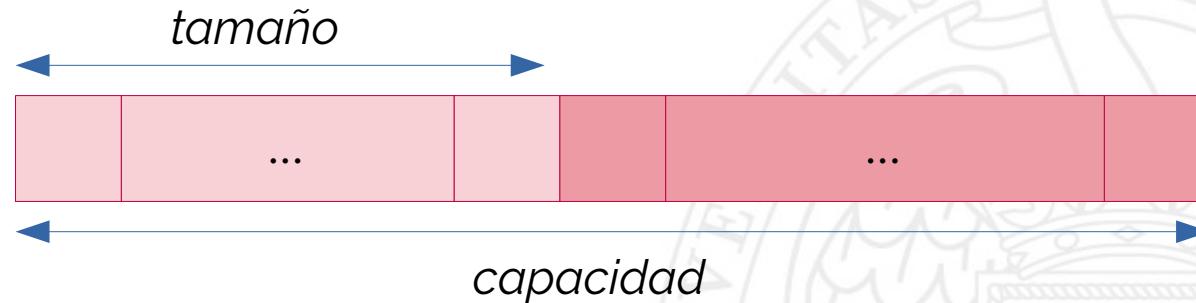
```
class ListArray {  
public:  
    ...  
private:  
    int num_elems;  
    std::string elems[MAX_CAPACITY];  
};
```

# ¿Y si el array se llena?



# Tamaño vs. capacidad

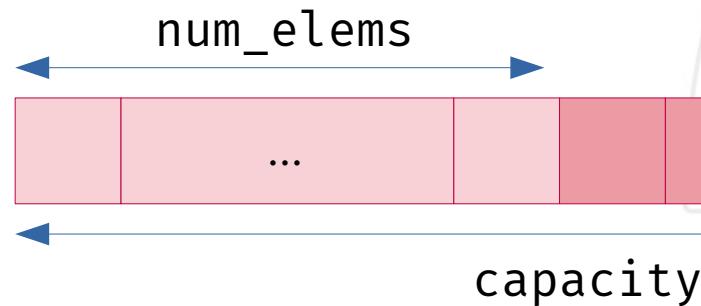
- **Capacidad de una lista:** Tamaño del array que contiene los elementos.
- **Tamaño de una lista:** Número de posiciones ocupadas por elementos.
  - Siempre se cumple  $\text{tamaño} \leq \text{capacidad}$ .



# Implementación con tamaño y capacidad

```
class ListArray {  
public:  
    ...  
private:  
    int num_elems;  
    int capacity;  
    std::string *elems;  
};
```

Array reservado dinámicamente



# Invariante y modelo de representación

```
class ListArray {  
public:  
    ...  
private:  
    int num_elems;  
    int capacity;  
    std::string *elems;  
};
```

- Invariante de la representación:

$$I(x) = 0 \leq x.\text{num\_elems} \leq x.\text{capacity}$$

- Función de abstracción:

$$f(x) = [x.\text{elems}[0], x.\text{elems}[1], \dots, x.\text{elems}[x.\text{num\_elems} - 1]]$$



# Creación y destrucción de una lista

```
const int DEFAULT_CAPACITY = 10;

class ListArray {

public:
    ListArray(int initial_capacity)
        : num_elems(0), capacity(initial_capacity), elems(new std::string[capacity]) { }

    ListArray()
        : ListArray(DEFAULT_CAPACITY) { }

    ~ListArray() { delete[] elems; }

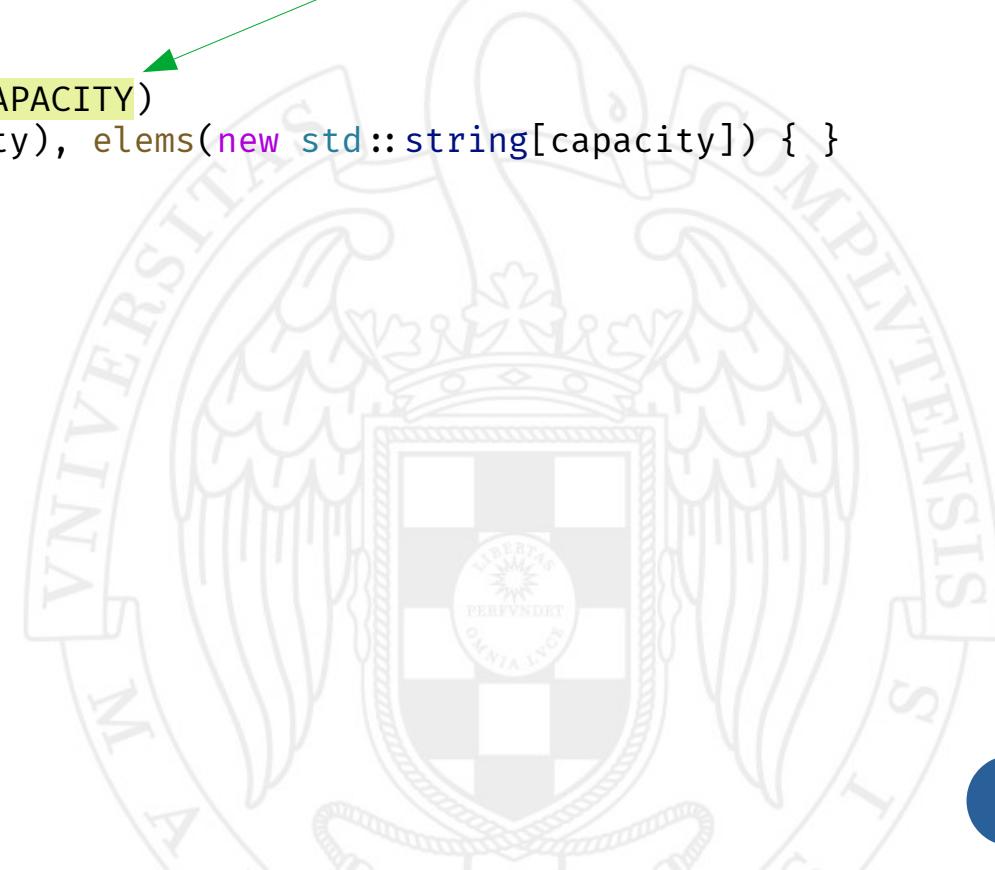
    ...

private:
    int num_elems;
    int capacity;
    std::string *elems;
};
```

# Creación y destrucción de una lista

```
const int DEFAULT_CAPACITY = 10;  
  
class ListArray {  
  
public:  
    ListArray(int initial_capacity = DEFAULT_CAPACITY)  
        : num_elems(0), capacity(initial_capacity), elems(new std::string[capacity]) {}  
  
    ListArray()  
        : ListArray(DEFAULT_CAPACITY) {}  
  
    ~ListArray() { delete[] elems; }  
  
    ...  
  
private:  
    int num_elems;  
    int capacity;  
    std::string *elems;  
};
```

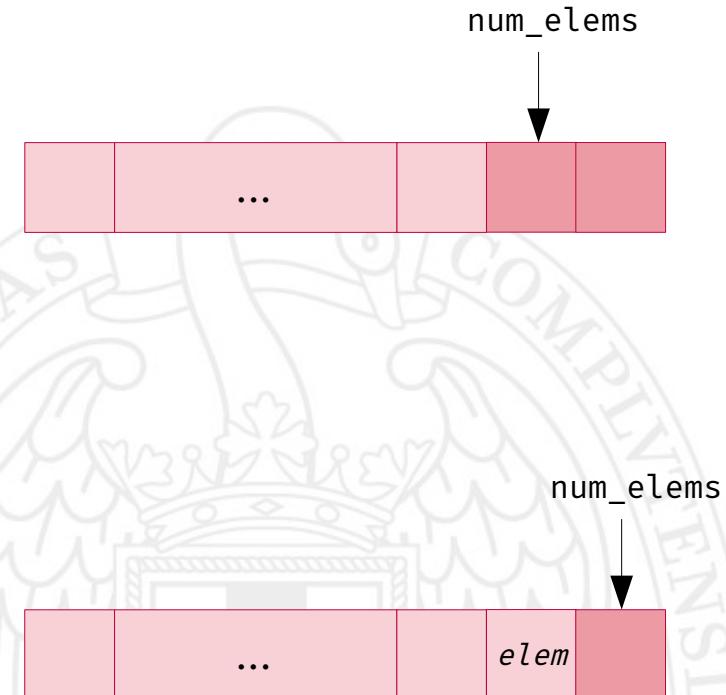
Valores por defecto para parámetros



# Añadir elementos al final de una lista

```
class ListArray {  
public:  
    void push_back(const std::string &elem);  
    ...  
private:  
    int num_elems;  
    int capacity;  
    std::string *elems;  
};
```

```
void ListArray::push_back(const std::string &elem) {  
    if (num_elems == capacity) {  
        resize_array(capacity * 2);  
    }  
  
    elems[num_elems] = elem;  
    num_elems++;  
}
```



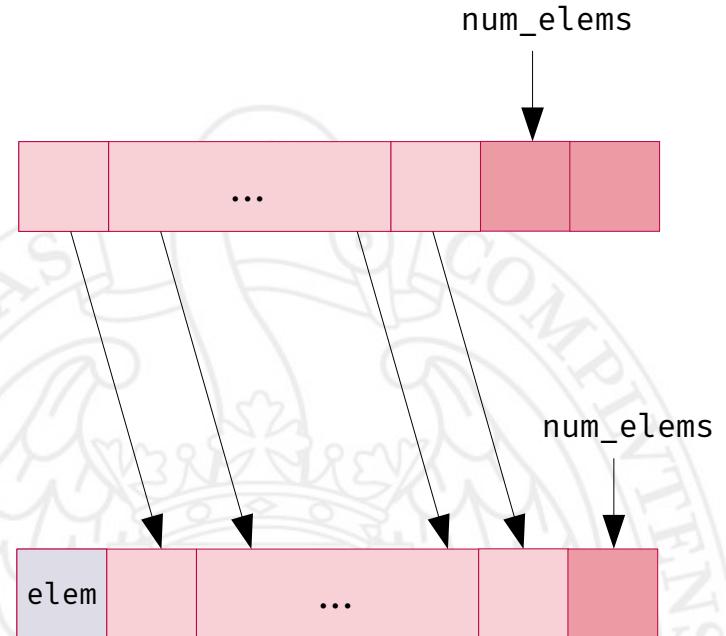
# Redimensionar el array

```
class ListArray {  
public:  
    ...  
private:  
    int num_elems;  
    int capacity;  
    std::string *elems;  
  
    void resize_array(int new_capacity);  
};  
  
void ListArray::resize_array(int new_capacity) {  
    std::string *new_elems = new std::string[new_capacity];  
    for (int i = 0; i < num_elems; i++) {  
        new_elems[i] = elems[i];  
    }  
  
    delete[] elems;  
    elems = new_elems;  
    capacity = new_capacity;  
}
```

# Añadir elementos al principio de una lista

```
class ListArray {  
public:  
    void push_front(const std::string &elem);  
    ...  
};
```

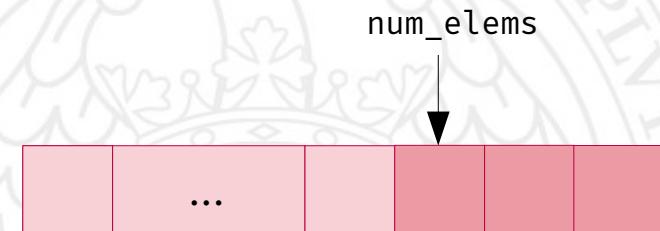
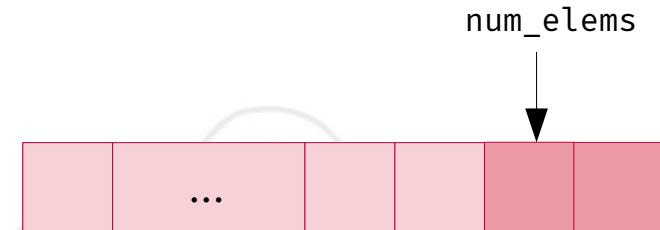
```
void ListArray::push_front(const std::string &elem) {  
    if (num_elems == capacity) {  
        resize_array(capacity * 2);  
    }  
  
    for (int i = num_elems - 1; i ≥ 0; i--) {  
        elems[i + 1] = elems[i];  
    }  
    elems[0] = elem;  
    num_elems++;  
}
```



# Eliminar elementos del final de una lista

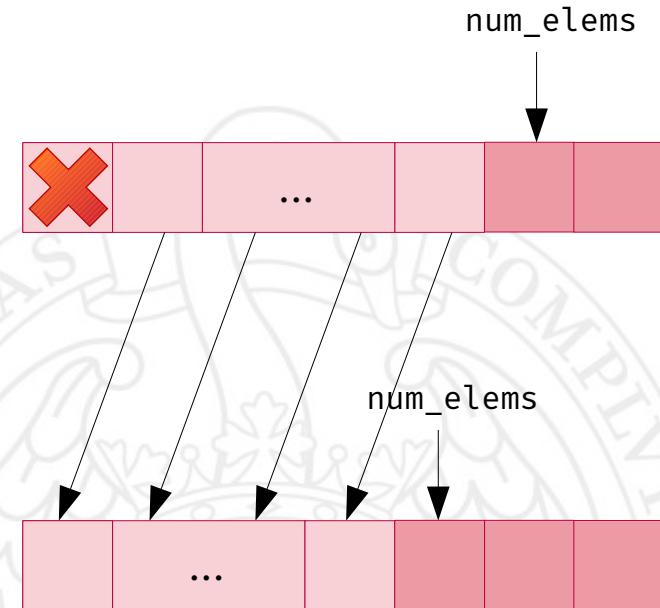
```
class ListArray {  
public:  
    void pop_back();  
    ...  
};
```

```
void ListArray::pop_back() {  
    assert (num_elems > 0);  
    num_elems--;  
}
```



# Eliminar elementos del principio de una lista

```
class ListArray {  
public:  
    void pop_front();  
    ...  
};  
  
void ListArray::pop_front() {  
    assert (num_elems > 0);  
  
    for (int i = 1; i < num_elems; i++) {  
        elems[i - 1] = elems[i];  
    }  
    num_elems--;  
}
```



# Funciones observadoras (1)

```
class ListArray {  
public:  
  
    int size() const {  
        return num_elems;  
    }  
  
    bool empty() const {  
        return num_elems == 0;  
    }  
  
    std::string front() const {  
        assert (num_elems > 0);  
        return elems[0];  
    }  
    ...  
};
```



# Funciones observadoras (2)

```
class ListArray {  
public:  
  
    std::string back() const {  
        assert (num_elems > 0);  
        return elems[num_elems - 1];  
    }  
  
    std::string at(int index) const {  
        assert (0 <= index && index < num_elems);  
        return elems[index];  
    }  
    ...  
};
```



# Mostrar el contenido de una lista

```
class ListArray {
public:
    void display() const;
    ...
};

void ListArray::display() const {
    std::cout << "[";
    if (num_elems > 0) {
        std::cout << elems[0];
        for (int i = 1; i < num_elems; i++) {
            std::cout << ", " << elems[i];
        }
    }
    std::cout << "]";
}
```



# Prueba de ejecución

```
int main() {
    ListArray l;
    l.push_back("David");
    l.push_back("Maria");
    l.push_back("Elvira");
    l.display(); std::cout << std::endl;

    std::cout << "Elemento 1: " << l.at(1) << std::endl;

    l.pop_front();
    l.display(); std::cout << std::endl;

    return 0;
}
```

[David, Maria, Elvira]

Maria

[Maria, Elvira]

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

# Modificación de listas mediante referencias

Manuel Montenegro Montes

Departamento de Sistemas Informáticos y Computación  
Facultad de Informática – Universidad Complutense de Madrid

# Comportamiento en memoria de at()

```
class ListArray {
public:

    std::string at(int index) const {
        assert (0 <= index && index < num_elems);
        return elems[index];
    }
    ...

private:
    int num_elems;
    int capacity;
    std::string *elems;
};
```



# Comportamiento en memoria de `at()`

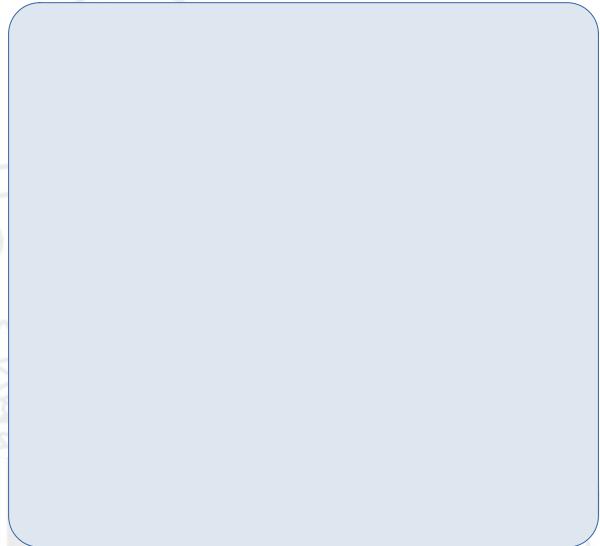
```
ListArray l;  
l.push_back("David");  
l.push_back("Maria");  
l.push_back("Elvira");  
  
std::string m = l.at(1);  
  
m = "Manuel"  
l.display();
```

[David, Maria, Elvira]

Pila



Heap



# Comportamiento en memoria de at()

```
class ListArray {
public:

    std::string & at(int index) const {
        assert (0 <= index && index < num_elems);
        return elems[index];
    }
    ...

private:
    int num_elems;
    int capacity;
    std::string *elems;
};
```

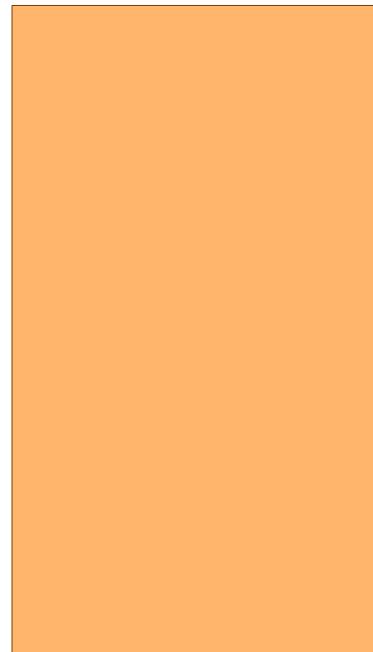


# ¿Y si `at()` devolviese una referencia?

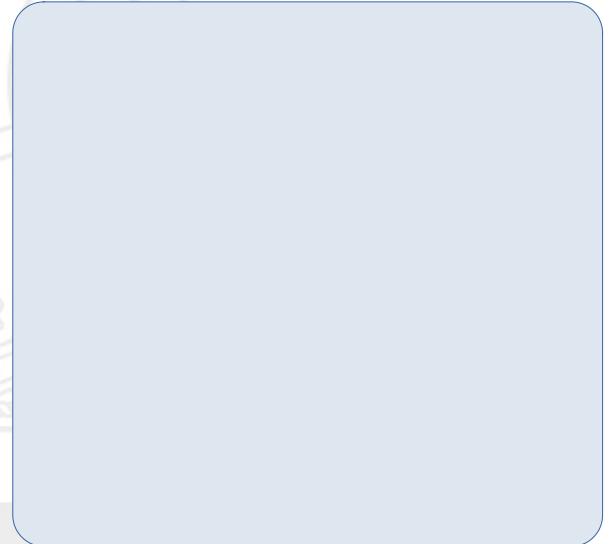
```
ListArray l;  
l.push_back("David");  
l.push_back("Maria");  
l.push_back("Elvira");  
  
std::string &m = l.at(1);  
m = "Manuel"  
  
l.display();
```

[David, Manuel, Elvira]

Pila



Heap



# ¿Y si `at()` devolviese una referencia?

```
ListArray l;  
l.push_back("David");  
l.push_back("Maria");  
l.push_back("Elvira");
```

`std::string &m = l.at(1);` ] equivale a ➔ `l.at(1) = "Manuel";`

```
l.display();
```

# Consecuencias

- Haciendo que `at()` devuelva una referencia al elemento del array permitimos la posibilidad de **actualizar** elementos de la lista, sin necesidad de necesitar un método específico para ello.
- Pero, a cambio, la función ha dejado de ser `const`. ☹
- Por ejemplo, la siguiente función dejaría de ser aceptada por el compilador:

```
int contar_caracteres(const ListArray &l) {  
    int suma = 0;  
    for (int i = 0; i < l.size(); i++) {  
        suma += l.at(i).length();  
    }  
    return suma;  
}
```

No puede llamarse a `at()`, porque `l` es una referencia constante.



# Solución: dos versiones para at()

```
class ListArray {
public:

    const std::string & at(int index) const {
        assert (0 <= index && index < num_elems);
        return elems[index];
    }

    std::string & at(int index) {
        assert (0 <= index && index < num_elems);
        return elems[index];
    }

    ...
};

};
```

Versión constante

Versión no constante

# Solución: dos versiones para at()

```
int contar_caracteres(const ListArray &l) {  
    int suma = 0;  
    for (int i = 0; i < l.size(); i++) {  
        suma += l.at(i).length();  
    }  
    return suma;  
}
```

Se llama a la versión  
constante de at()

```
ListArray l;  
...  
l.at(1) = "Manuel";
```

Se llama a la versión  
no constante de at()

# Referencias en front() y back()

```
const std::string & front() const {
    assert (num_elems > 0);
    return elems[0];
}

std::string & front() {
    assert (num_elems > 0);
    return elems[0];
}

const std::string & back() const {
    assert (num_elems > 0);
    return elems[num_elems - 1];
}

std::string & back() {
    assert (num_elems > 0);
    return elems[num_elems - 1];
}
```



# ¡Cuidado con las referencias!

```
int main() {
    ListArray l(3);
    l.push_back("Javier");
    l.push_back("Simona");
    l.push_back("Jerry");

    std::string &primero = l.front();

    l.push_back("David");

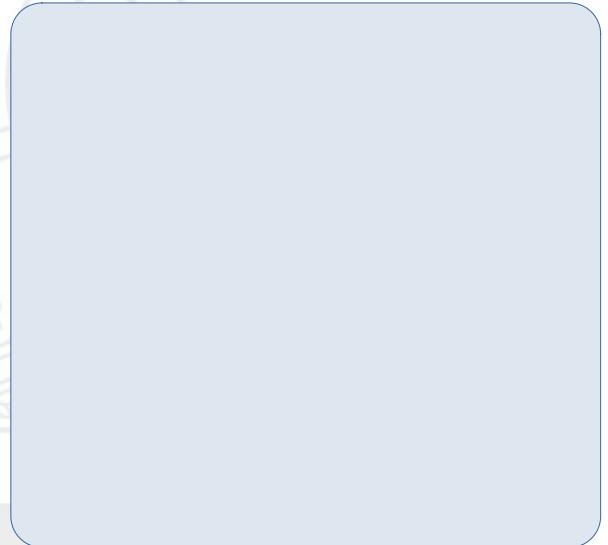
    primero = "Javier Francisco";

    return 0;
}
```

Pila



Heap



# ¡Cuidado con las referencias!

- Si se obtiene una referencia a un elemento de la lista, debe hacerse uso de esa referencia (para leer o modificar el valor apuntado por la referencia) **antes** de añadir o eliminar otros elementos de la lista.

```
l.front() = "Javier Francisco"; 
```

```
std::string &primero = l.front();   
...  
primero = "Javier Francisco";
```



ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

# Implementación del TAD Lista mediante listas enlazadas

Manuel Montenegro Montes

Departamento de Sistemas Informáticos y Computación  
Facultad de Informática – Universidad Complutense de Madrid

# Recordatorio: operaciones del TAD Lista

- **Constructoras:**
  - Crear una lista vacía: ***create\_empty()*** →  $L: List$
- **Mutadoras:**
  - Añadir un elemento al principio de la lista: ***push\_front(x: elem, L: List)***.
  - Añadir un elemento al final de la lista: ***push\_back(x: elem, L: List)***.
  - Eliminar el elemento del principio de la lista: ***pop\_front(L: List)***.
  - Eliminar el elemento del final de la lista: ***pop\_back(L: List)***.
- **Observadoras:**
  - Obtener el tamaño de la lista: ***size(L: List)*** →  $tam: int$ .
  - Comprobar si la lista es vacía ***empty(L: List)*** →  $b: bool$ .
  - Acceder al primer elemento de la lista ***front(L: List)*** →  $e: elem$ .
  - Acceder al último elemento de la lista ***back(L: List)*** →  $e: elem$ .
  - Acceder a un elemento que ocupa una posición determinada ***at(idx: int, L: List)*** →  $e: elem$ .

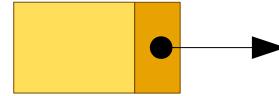
# ¿Qué es una lista enlazada?

- Secuencia de **nodos**, en la que cada nodo contiene:
  - Un campo con información arbitraria.
  - Un puntero al siguiente nodo de la secuencia.
- En este caso, decimos que son **listas enlazadas simples**.



# Definición de un nodo

```
struct Node {  
    std::string value;  
    Node *next;  
};
```



- Cuando un nodo no tiene sucesor, su campo next contiene el puntero nulo (`nullptr` en C++).

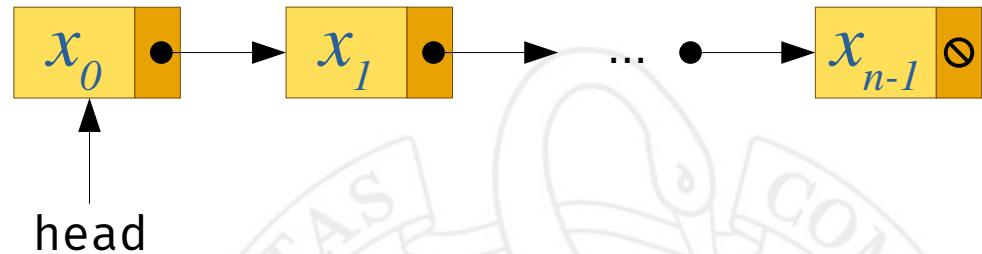


# Ejemplo

```
struct Node {  
    std::string value;  
    Node *next;  
};  
  
Node *tres = new Node { "Tres", nullptr };  
Node *dos = new Node { "Dos", tres };  
Node *uno = new Node { "Uno", dos };
```

# El TAD Lista mediante listas enlazadas

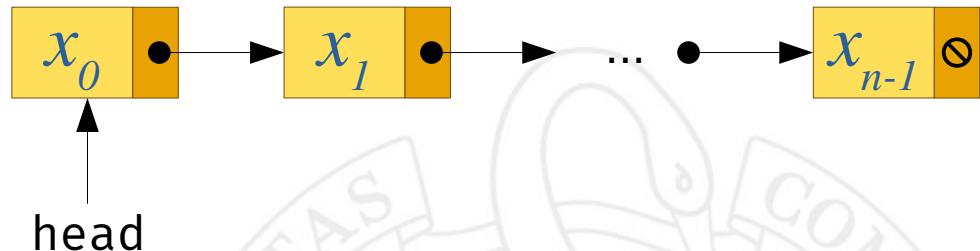
[  $x_0, x_1, \dots, x_{n-1}$  ]



```
class ListLinkedSingle {  
public:  
    ...  
private:  
    struct Node { ... };  
    Node *head;  
};
```

# El TAD Lista mediante listas enlazadas

[  $x_0, x_1, \dots, x_{n-1}$  ]



- Invariante de representación:  
 $I(x) = \text{true}$
- Función de abstracción:  
 $f(x) = [ x.\text{head} \rightarrow \text{value}, x.\text{head} \rightarrow \text{next} \rightarrow \text{value}, x.\text{head} \rightarrow \text{next} \rightarrow \text{next} \rightarrow \text{value}, \dots ]$

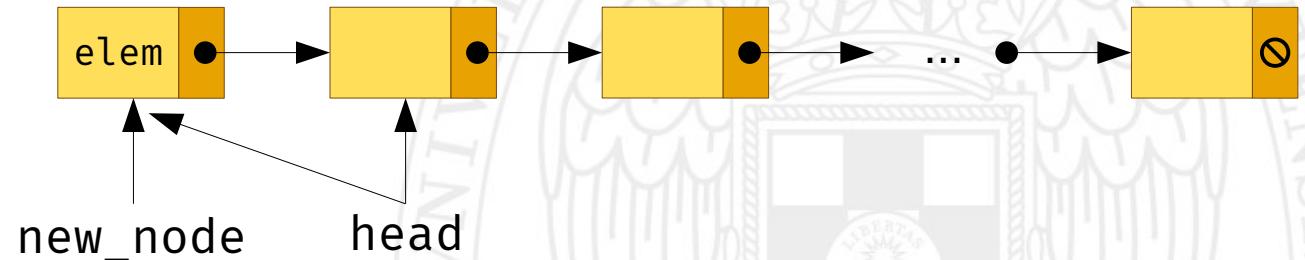
# Inicializar lista

```
class ListLinkedSingle {  
public:  
    ListLinkedSingle(): head(nullptr) { }  
    ...  
  
private:  
    struct Node { ... };  
    Node *head;  
};
```



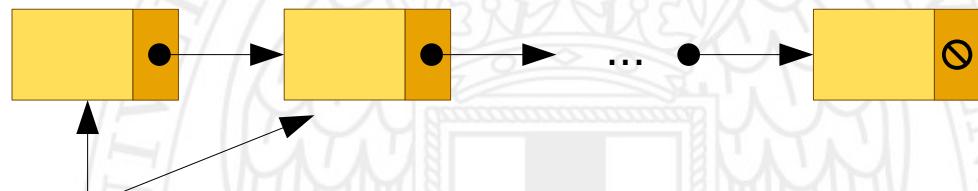
# Añadir un elemento al principio de la lista

```
class ListLinkedSingle {  
public:  
    ...  
    void push_front(const std::string &elem) {  
        Node *new_node = new Node { elem, head };  
        head = new_node;  
    }  
  
private:  
    struct Node { ... };  
    Node *head;  
};
```



# Eliminar un elemento del principio de la lista

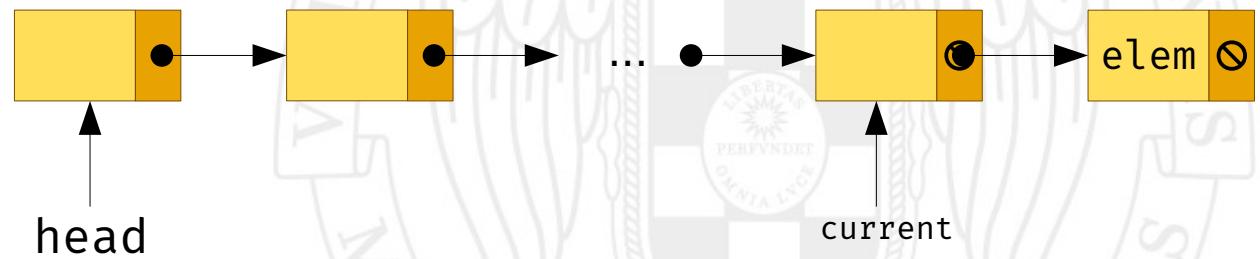
```
class ListLinkedSingle {  
public:  
    ...  
    void pop_front() {  
        assert (head != nullptr);  
        Node *old_head = head;  
        head = head->next;  
        delete old_head;  
    }  
  
private:  
    struct Node { ... };  
    Node *head;  
};
```



head

# Añadir un elemento al final de la lista

```
class ListLinkedSingle {  
public:  
    ...  
    void push_back(const std::string &elem);  
    ...  
};  
  
void ListLinkedSingle::push_back(const std::string &elem) {  
    Node *new_node = new Node { elem, nullptr };  
    if (head == nullptr) {  
        head = new_node;  
    } else {  
        Node *current = head;  
        while (current->next != nullptr) {  
            current = current->next;  
        }  
        current->next = new_node;  
    }  
}
```



# Refactorizando: obtener el último nodo

```
ListLinkedSingle::Node * ListLinkedSingle::last_node() const {
    assert (head != nullptr);
    Node *current = head;
    while (current->next != nullptr) {
        current = current->next;
    }
    return current;
}

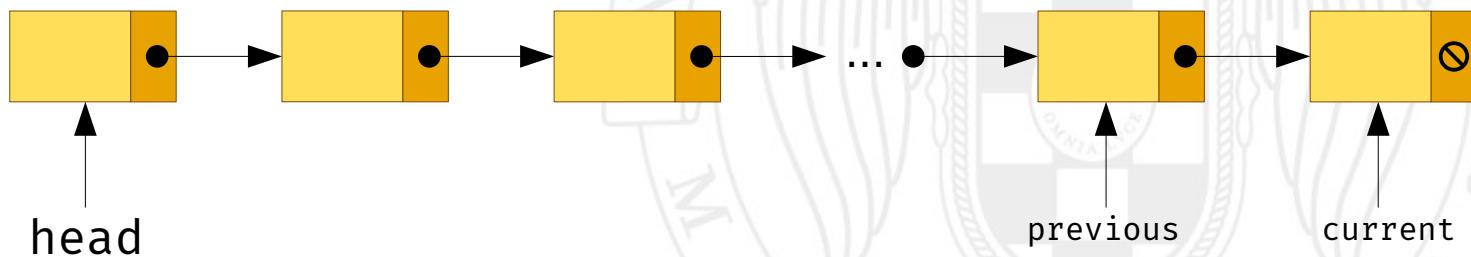
void ListLinkedSingle::push_back(const std::string &elem) {
    Node *new_node = new Node { elem, nullptr };
    if (head == nullptr) {
        head = new_node;
    } else {
        last_node()->next = new_node;
    }
}
```

# Eliminar un elemento del final de la lista

```
void ListLinkedSingle::pop_back() {
    assert (head != nullptr);
    if (head->next == nullptr) {
        delete head;
        head = nullptr;
    } else {
        Node *previous = head;
        Node *current = head->next;

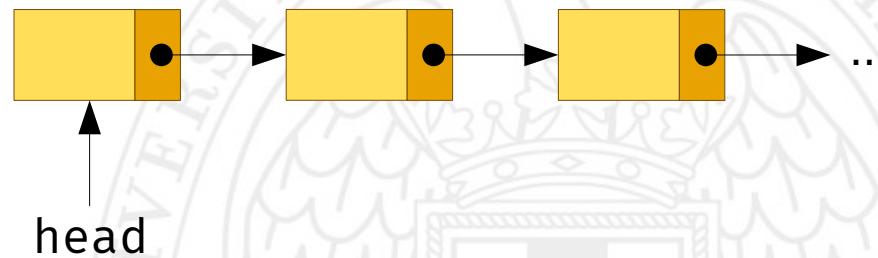
        while (current->next != nullptr) {
            previous = current;
            current = current->next;
        }

        delete current;
        previous->next = nullptr;
    }
}
```



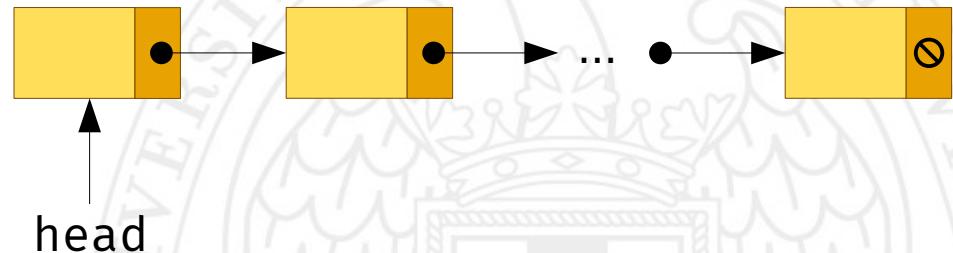
# Acceder al primer elemento de la lista

```
class ListLinkedSingle {  
public:  
    ...  
    const std::string & front() const {  
        assert (head != nullptr);  
        return head->value;  
    }  
  
    std::string & front() { ... }  
  
private:  
    struct Node { ... };  
    Node *head;  
};
```



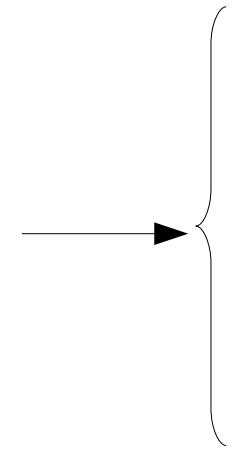
# Acceder al último elemento de la lista

```
class ListLinkedSingle {  
public:  
    ...  
    const std::string & back() const {  
        return last_node()→value;  
    }  
  
    std::string & back() { ... }  
  
private:  
    struct Node { ... };  
    Node *head;  
  
    Node *last_node() const;  
};
```



# Acceder al elemento $n$ -ésimo de la lista

```
class ListLinkedSingle {  
public:  
    ...  
    const std::string & at(int index) const {  
        Node *result_node = nth_node(index);  
        assert (result_node != nullptr);  
        return result_node->value;  
    }  
  
    std::string & at(int index) { ... }  
  
private:  
    struct Node { ... };  
    Node *head;  
  
    Node *last_node() const;  
    Node *nth_node(int n) const;  
};
```



```
Node * ListLinkedSingle::nth_node(int n) const {  
    assert (0 <= n);  
    int current_index = 0;  
    Node *current = head;  
  
    while (current_index < n && current != nullptr) {  
        current_index++;  
        current = current->next;  
    }  
  
    return current;  
}
```

# Obtener el tamaño de una lista

```
class ListLinkedSingle {
public:
    int size() const;

    bool empty() const {
        return head == nullptr;
    };
    ...
};

int ListLinkedSingle::size() const {
    int num_nodes = 0;

    Node *current = head;
    while (current != nullptr) {
        num_nodes++;
        current = current->next;
    }

    return num_nodes;
}
```



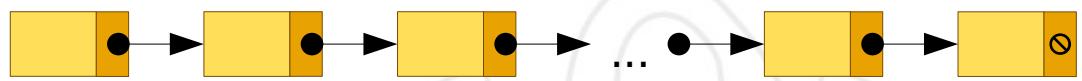
# Mostrar una lista por pantalla

```
void ListLinkedSingle::display(std::ostream &out) const {
    std::cout << "[";
    if (head != nullptr) {
        out << head->value;
        Node *current = head->next;
        while (current != nullptr) {
            out << ", " << current->value;
            current = current->next;
        }
    }
    out << "]";
}
```



# Destrucción de una lista

```
class ListLinkedSingle {  
public:  
    ...  
    ~ListLinkedSingle() {  
        delete_list(head);  
    }  
  
private:  
    ...  
    void delete_list(Node *start_node);  
}  
  
void ListLinkedSingle::delete_list(Node *start_node) {  
    if (start_node != nullptr) {  
        delete_list(start_node->next);  
        delete start_node;  
    }  
}
```



ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

# Constructores de copia en el TAD Lista

Manuel Montenegro Montes

Departamento de Sistemas Informáticos y Computación  
Facultad de Informática – Universidad Complutense de Madrid

# Implementaciones del TAD Lista

- Mediante vectores.
- Mediante listas enlazadas simples.



# Implementación mediante vectores

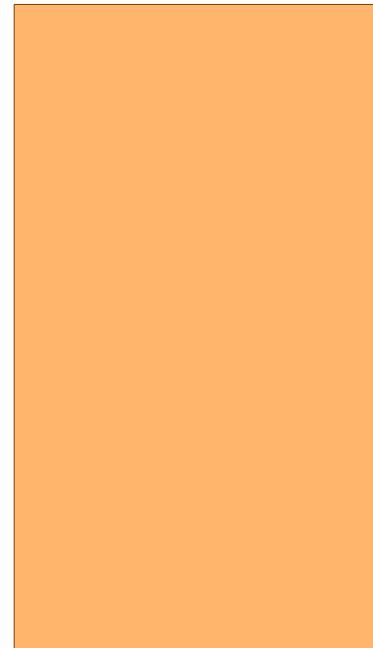
# Implementación mediante vectores

- El constructor de copia por defecto para la clase `ListArray` no nos sirve.

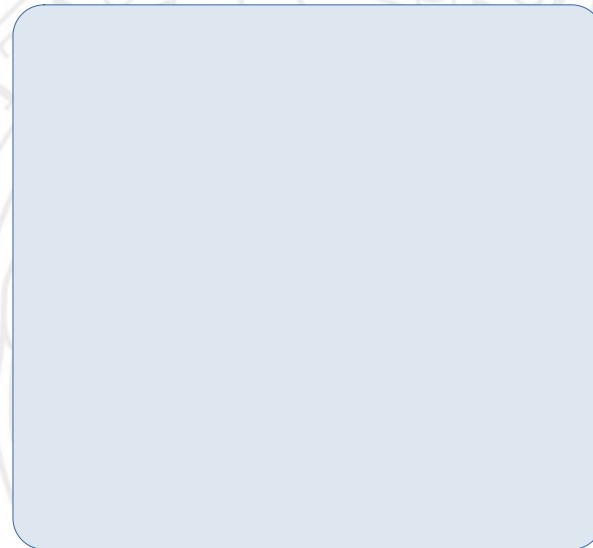
```
ListArray l1;  
l1.push_back("David");  
l1.push_back("Maria");  
l1.push_back("Eugenio");
```

```
ListArray l2 = l1;  
l2.front() = "Pepe";
```

Pila

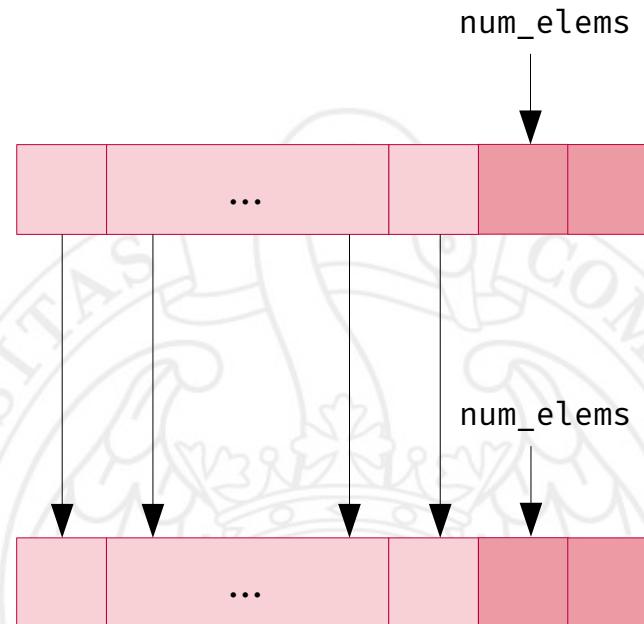


Heap



# Constructor de copia para ListArray

```
class ListArray {  
public:  
    ...  
    ListArray(const ListArray &other);  
    ...  
};  
  
ListArray::ListArray(const ListArray &other)  
: num_elems(other.num_elems),  
 capacity(other.capacity),  
 elems(new std::string[other.capacity])  
{  
    for (int i = 0; i < num_elems; i++) {  
        elems[i] = other.elems[i];  
    }  
}
```

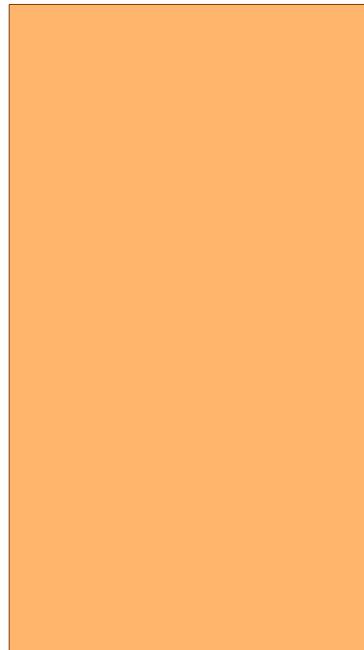


# Ejemplo

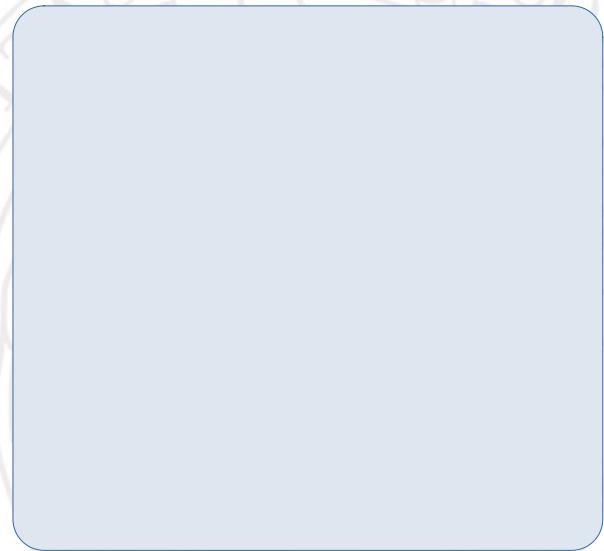
- Con el constructor de copia, l1 y l2 pueden ser modificadas de manera independiente.

```
ListArray l1;  
l1.push_back("David");  
l1.push_back("Maria");  
l1.push_back("Eugenio");  
  
ListArray l2 = l1;  
l2.front() = "Pepe";
```

Pila



Heap



# Implementación mediante listas enlazadas

# Implementación mediante listas enlazadas

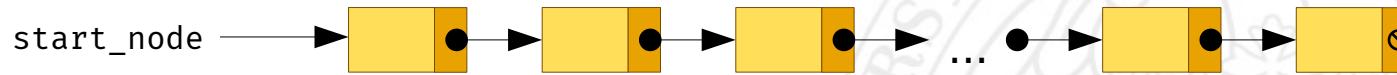
- Creamos una función auxiliar (`copy_nodes`) para producir una copia de una lista enlazada de nodos.
- El constructor de copia inicializa la cabeza creando una copia de la lista enlazada original.

```
class ListLinkedSingle {  
public:  
    ListLinkedSingle(const ListLinkedSingle &other)  
        : head(copy_nodes(other.head)) { }  
  
private:  
    Node *head;  
  
...  
    Node *copy_nodes(Node *start_node) const;  
};
```



# Implementación recursiva de copy\_nodes

```
ListLinkedSingle::Node * ListLinkedSingle::copy_nodes(Node *start_node) const {
    if (start_node != nullptr) {
        Node *result = new Node { start_node->value, copy_nodes(start_node->next) };
        return result;
    } else {
        return nullptr;
    }
}
```



ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

# Nodos fantasma

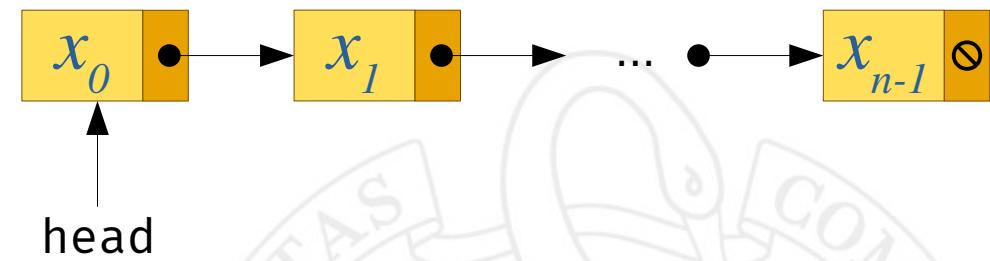
Manuel Montenegro Montes

Departamento de Sistemas Informáticos y Computación  
Facultad de Informática – Universidad Complutense de Madrid

# Recordatorio

$[x_0, x_1, \dots, x_{n-1}]$

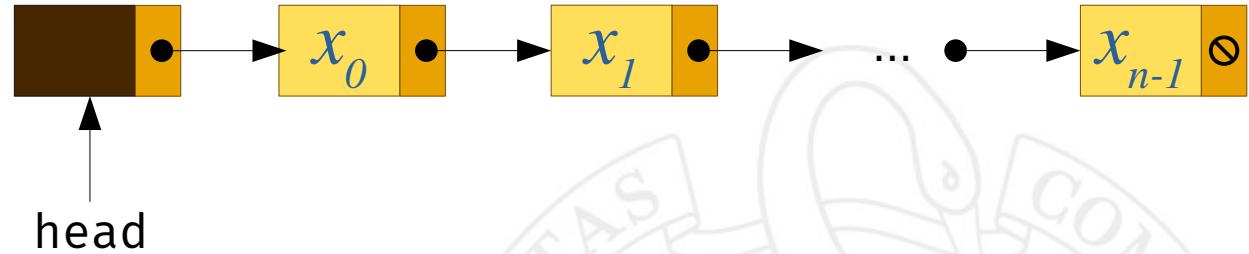
$[]$



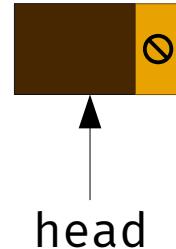
`head = nullptr`

# Introduciendo un nodo fantasma

$[x_0, x_1, \dots, x_{n-1}]$



$[]$



# Nodo fantasma

- Es un nodo que se sitúa siempre al principio de la lista enlazada de nodos.
- La información que contiene (esto es, su atributo `value`) es irrelevante.
- El atributo `head` de la lista apunta siempre a este nodo fantasma.  
    ⇒ **head nunca va a tomar el valor `nullptr`.**

**Consecuencia:** simplificación de la implementación de algunos métodos.

# Interfaz del TAD Lista

```
class ListLinkedSingle {  
public:  
    ListLinkedSingle();  
    ListLinkedSingle(const ListLinkedSingle &other);  
    ~ListLinkedSingle();  
  
    void push_front(const std::string &elem);  
    void push_back(const std::string &elem);  
    void pop_front();  
    void pop_back();  
    int size() const;  
    bool empty() const;  
    const std::string & front() const;  
    std::string & front();  
    const std::string & back() const;  
    std::string & back();  
    const std::string & at(int index) const;  
    std::string & at(int index);  
    void display() const;  
private:  
    ...  
};
```

```
ListLinkedSingle() {  
    head = new Node;  
    head->next = nullptr;  
}
```



# Cambios en la implementación

```
class ListLinkedSingle {  
public:  
    ListLinkedSingle();  
    ListLinkedSingle(const ListLinkedSingle &other);  
    ~ListLinkedSingle();  
  
    void push_front(const std::string &elem);  
    void push_back(const std::string &elem);  
    void pop_front();  
    void pop_back();  
    int size() const;  
    bool empty() const;  
    const std::string & front() const;  
    std::string & front();  
    const std::string & back() const;  
    std::string & back();  
    const std::string & at(int index) const;  
    std::string & at(int index);  
    void display() const;  
private:  
    ...  
};
```

- El constructor de copia y el destructor no cambian con la incorporación de nodos fantasma.
- Tampoco varían los métodos privados asociados:

`delete_list()`  
`copy_nodes()`

# Cambios en la implementación

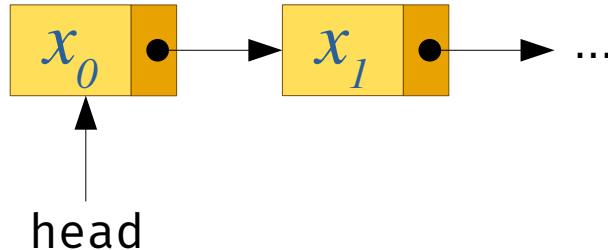
```
class ListLinkedSingle {  
public:  
    ListLinkedSingle();  
    ListLinkedSingle(const ListLinkedSingle &other);  
    ~ListLinkedSingle();  
  
    void push_front(const std::string &elem);  
    void push_back(const std::string &elem);  
    void pop_front();  
    void pop_back();  
    int size() const;  
    bool empty() const;  
    const std::string & front() const;  
    std::string & front();  
    const std::string & back() const;  
    std::string & back();  
    const std::string & at(int index) const;  
    std::string & at(int index);  
    void display() const;  
private:  
    ...  
};
```

- La mayoría de operaciones requieren cambios triviales.
- Por ejemplo:  
`assert(head != nullptr)`  
se transforma en  
`assert(head->next != nullptr)`
- Las operaciones de iteración comienzan en `head->next` en lugar de en `head`.

# Ejemplo: método front()

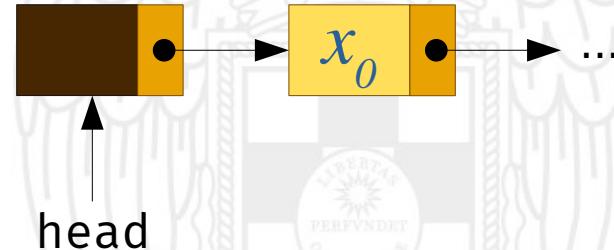
Antes

```
std::string & front() {  
    assert (head != nullptr);  
    return head->value;  
}
```



Después

```
std::string & front() {  
    assert (head->next != nullptr);  
    return head->next->value;  
}
```



# Ejemplo: método nth\_node()

## Antes

```
Node *nth_node(int n) const {
    assert (0 ≤ n);
    int current_index = 0;
    Node *current = head;

    while (current_index < n
           && current ≠ nullptr) {
        current_index++;
        current = current→next;
    }
    return current;
}
```

## Después

```
Node *nth_node(int n) const {
    assert (0 ≤ n);
    int current_index = 0;
    Node *current = head→next;

    while (current_index < n
           && current ≠ nullptr) {
        current_index++;
        current = current→next;
    }
    return current;
}
```

# Cambios en la implementación

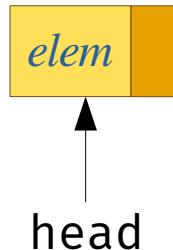
```
class ListLinkedSingle {  
public:  
    ListLinkedSingle();  
    ListLinkedSingle(const ListLinkedSingle &other);  
    ~ListLinkedSingle();  
  
    void push_front(const std::string &elem);  
    void push_back(const std::string &elem);  
    void pop_front();  
    void pop_back();  
    int size() const;  
    bool empty() const;  
    const std::string & front() const;  
    std::string & front();  
    const std::string & back() const;  
    std::string & back();  
    const std::string & at(int index) const;  
    std::string & at(int index);  
    void display() const;  
private:  
    ...  
};
```

- La implementación de las operaciones `push_back()` y `pop_back()` se simplifican, ya que no tienen que comprobar si la lista es vacía o no.

# Cambios en push\_back()

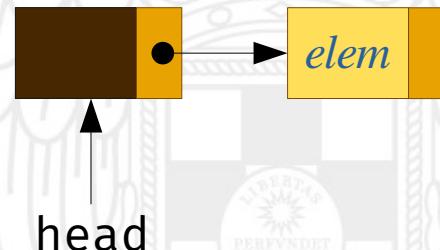
## Antes

```
void push_back(const std::string &elem) {  
    Node *new_node = new Node { elem, nullptr };  
    if (head == nullptr) {  
        head = new_node;  
    } else {  
        last_node()→next = new_node;  
    }  
}
```



## Después

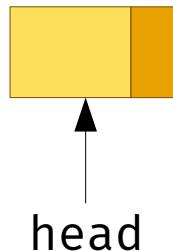
```
void push_back(const std::string &elem) {  
    Node *new_node = new Node { elem, nullptr };  
    last_node()→next = new_node;  
}
```



# Cambios en pop\_back()

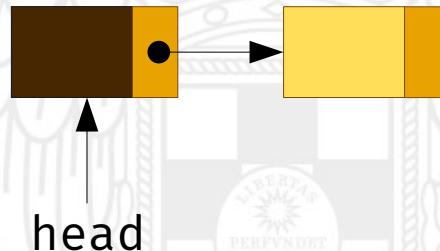
## Antes

```
void pop_back() {
    assert (head != nullptr);
    if (head->next = nullptr) {
        delete head;
        head = nullptr;
    } else {
        // borrar último nodo
    }
}
```



## Después

```
void pop_back() {
    assert (head->next != nullptr);
    // borrar último nodo
}
```



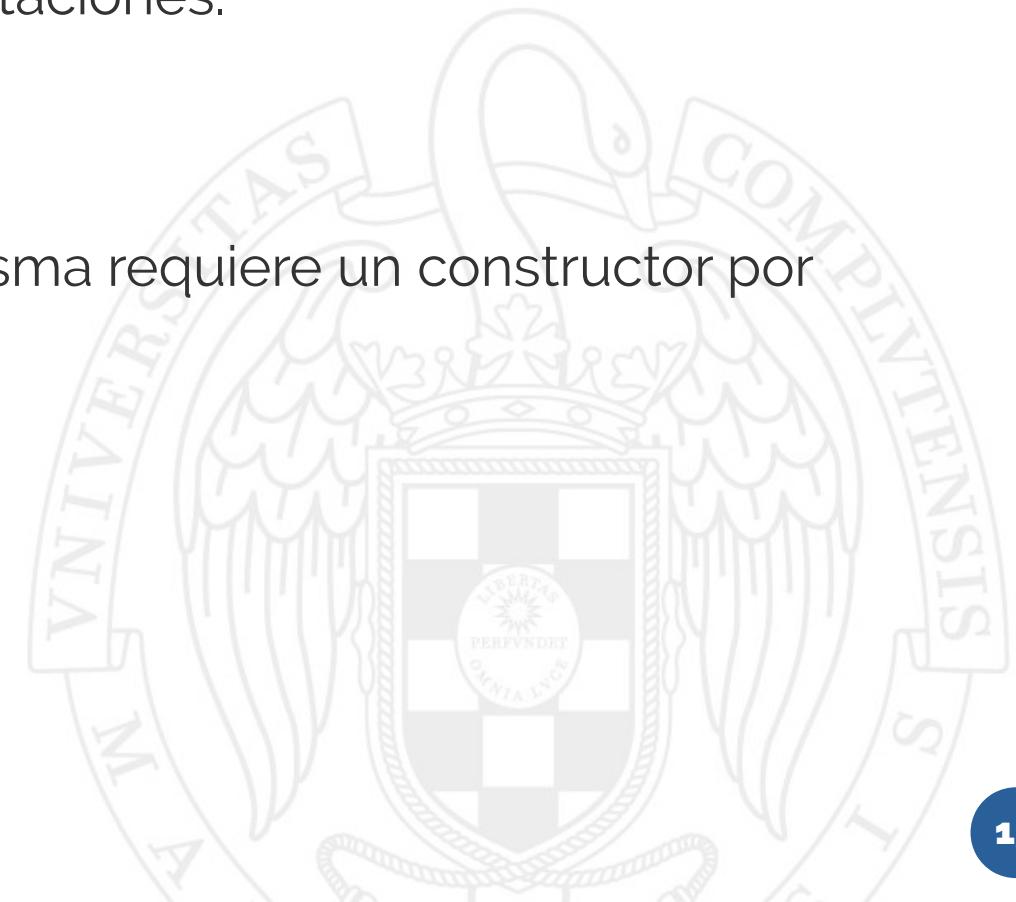
# Conclusiones

## Ventajas

- Simplificación en las implementaciones.

## Desventajas

- Un nodo extra en memoria.
- La inicialización del nodo fantasma requiere un constructor por defecto.



ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

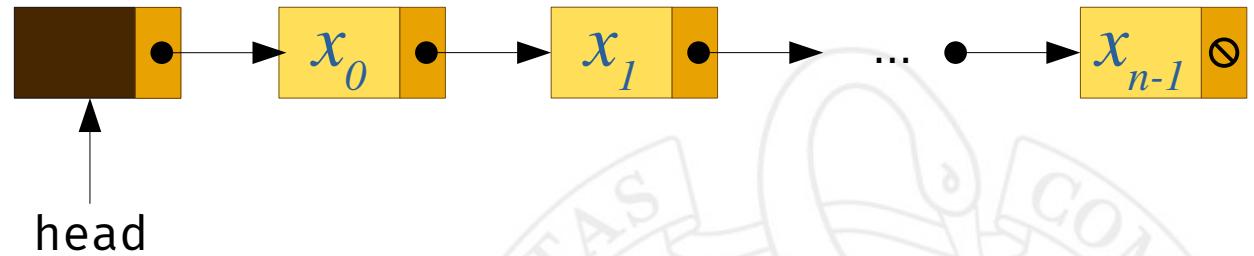
# Listas doblemente enlazadas (1)

Manuel Montenegro Montes

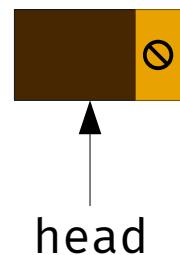
Departamento de Sistemas Informáticos y Computación  
Facultad de Informática – Universidad Complutense de Madrid

# Recordatorio: listas enlazadas simples

$[x_0, x_1, \dots, x_{n-1}]$

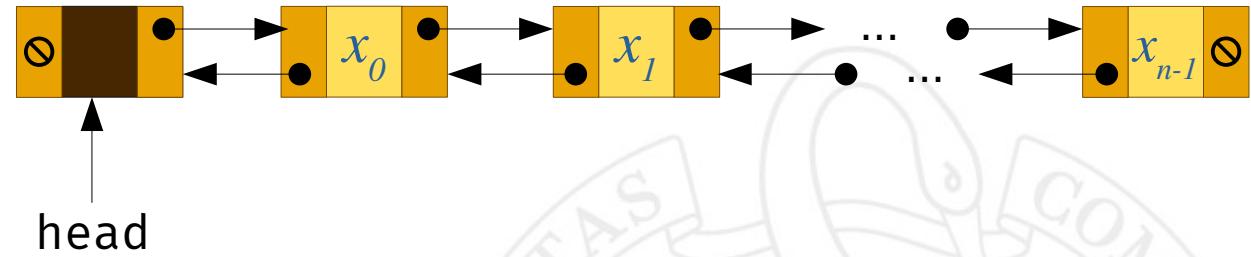


$[]$

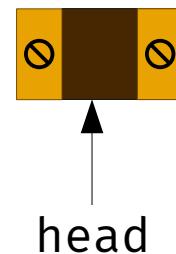


# Listas doblemente enlazadas

$[ x_0, x_1, \dots, x_{n-1} ]$



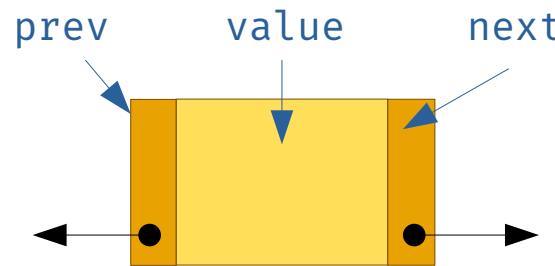
$[]$



# Listas enlazadas dobles

```
struct Node {  
    std::string value;  
    Node *next;  
    Node *prev;  
};
```

- Cada nodo tiene dos punteros:
  - next**: Nodo siguiente en la lista enlazada.
  - prev**: Nodo anterior en la lista enlazada.



# Implementación: ListLinkedDouble

```
class ListLinkedDouble {
public:
    ListLinkedDouble();
    ListLinkedDouble(const ListLinkedDouble &other);
    ~ListLinkedDouble();

    void push_front(const std::string &elem);
    void push_back(const std::string &elem);
    void pop_front();
    void pop_back();
    int size() const;
    bool empty() const;
    const std::string & front() const;
    std::string & front();
    const std::string & back() const;
    std::string & back();
    const std::string & at(int index) const;
    std::string & at(int index);
    void display() const;
private:
    ...
    Node *head;
};
```



# Constructores y destructor

```
ListLinkedDouble() {  
    head = new Node;  
    head->next = nullptr;  
    head->prev = nullptr;  
}
```

```
~ListLinkedDouble() {  
    delete_list(head);  
}
```

```
ListLinkedDouble(const ListLinkedDouble &other)  
: head(copy_nodes(other.head)) { }
```

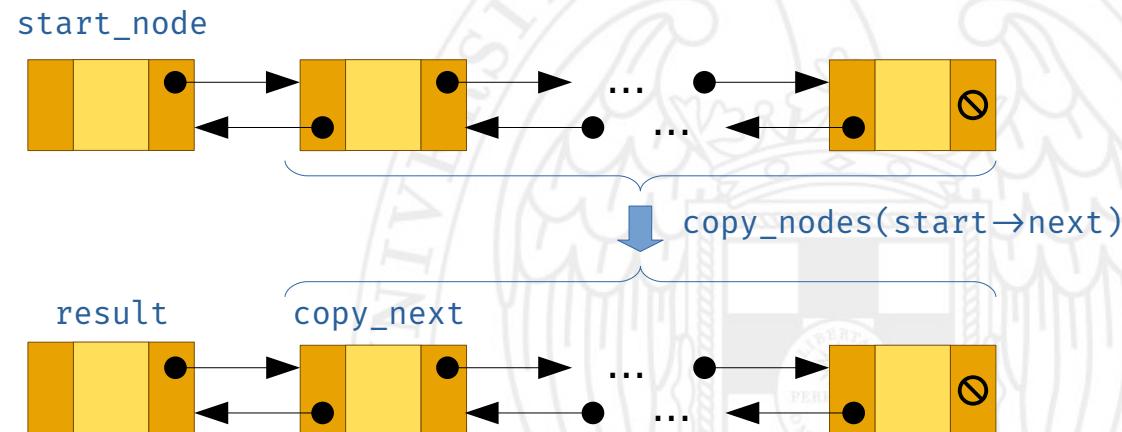


head



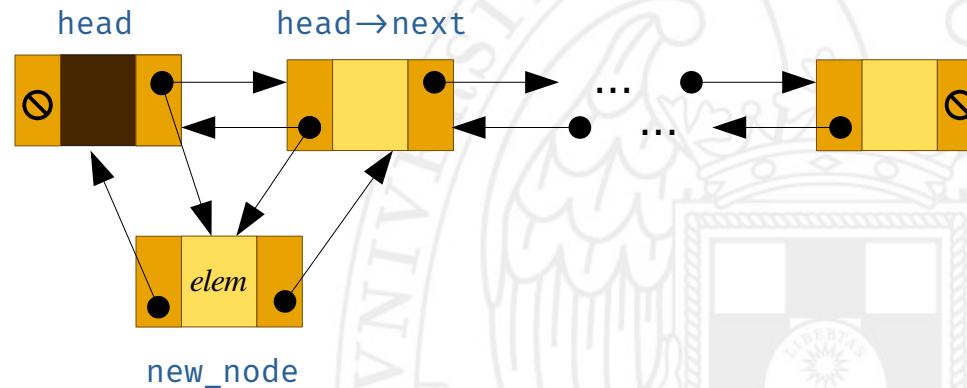
# Copia de una cadena de nodos

```
Node * ListLinkedDouble::copy_nodes(Node *start_node) const {
    if (start_node != nullptr) {
        Node *copy_next = copy_nodes(start_node->next);
        Node *result = new Node { start_node->value, copy_next, nullptr };
        if (copy_next != nullptr) {
            copy_next->prev = result;
        }
        return result;
    } else {
        return nullptr;
    }
}
```



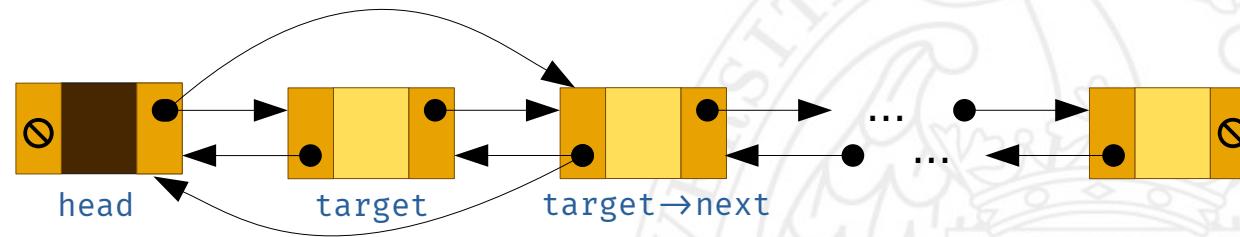
# Insertar al principio de la cadena

```
void push_front(const std::string &elem) {  
    Node *new_node = new Node { elem, head->next, head };  
    if (head->next != nullptr) {  
        head->next->prev = new_node;  
    }  
    head->next = new_node;  
}
```



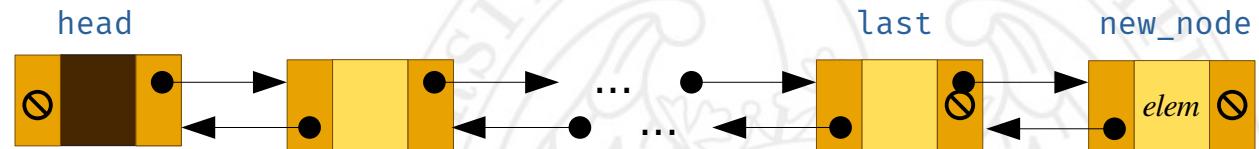
# Eliminar al principio de la cadena

```
void pop_front() {
    assert (head->next != nullptr);
    Node *target = head->next;
    head->next = target->next;
    if (target->next != nullptr) {
        target->next->prev = head;
    }
    delete target;
}
```



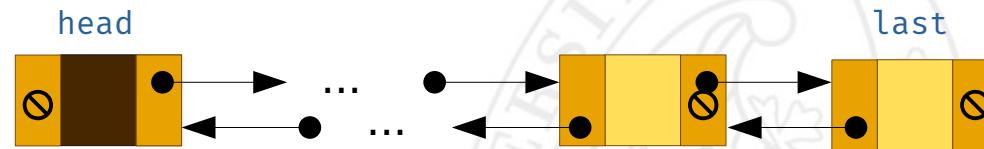
# Insertar al final de la cadena

```
void push_back(const std::string &elem) {
    Node *last = last_node();
    Node *new_node = new Node { elem, nullptr, last };
    last->next = new_node;
}
```



# Eliminar al final de la cadena

```
void pop_back() {
    assert (head->next != nullptr);
    Node *last = last_node();
    last->prev->next = nullptr;
    delete last;
}
```



# ¿Mejoras en el coste?

Operación	Listas enlazadas simples	Listas doblemente enlazadas
Creación	$O(1)$	$O(1)$
Copia	$O(n)$	$O(n)$
push_back	$O(n)$	$O(n)$
push_front	$O(1)$	$O(1)$
pop_back	$O(n)$	$O(n)$
pop_front	$O(1)$	$O(1)$
back	$O(n)$	$O(n)$
front	$O(1)$	$O(1)$
display	$O(n)$	$O(n)$
at(index)	$O(index)$	$O(index)$
size	$O(n)$	$O(n)$
empty	$O(1)$	$O(1)$

$n$  = número de elementos de la lista de entrada

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

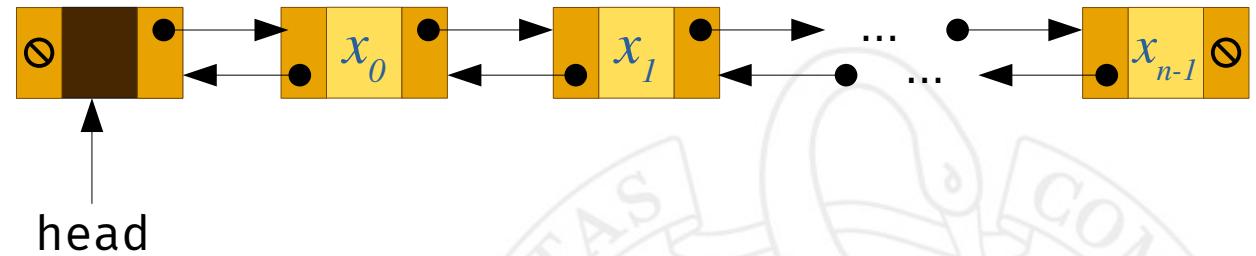
# Listas doblemente enlazadas (2)

Manuel Montenegro Montes

Departamento de Sistemas Informáticos y Computación  
Facultad de Informática – Universidad Complutense de Madrid

# Listas doblemente enlazadas

$[ x_0, x_1, \dots, x_{n-1} ]$



head

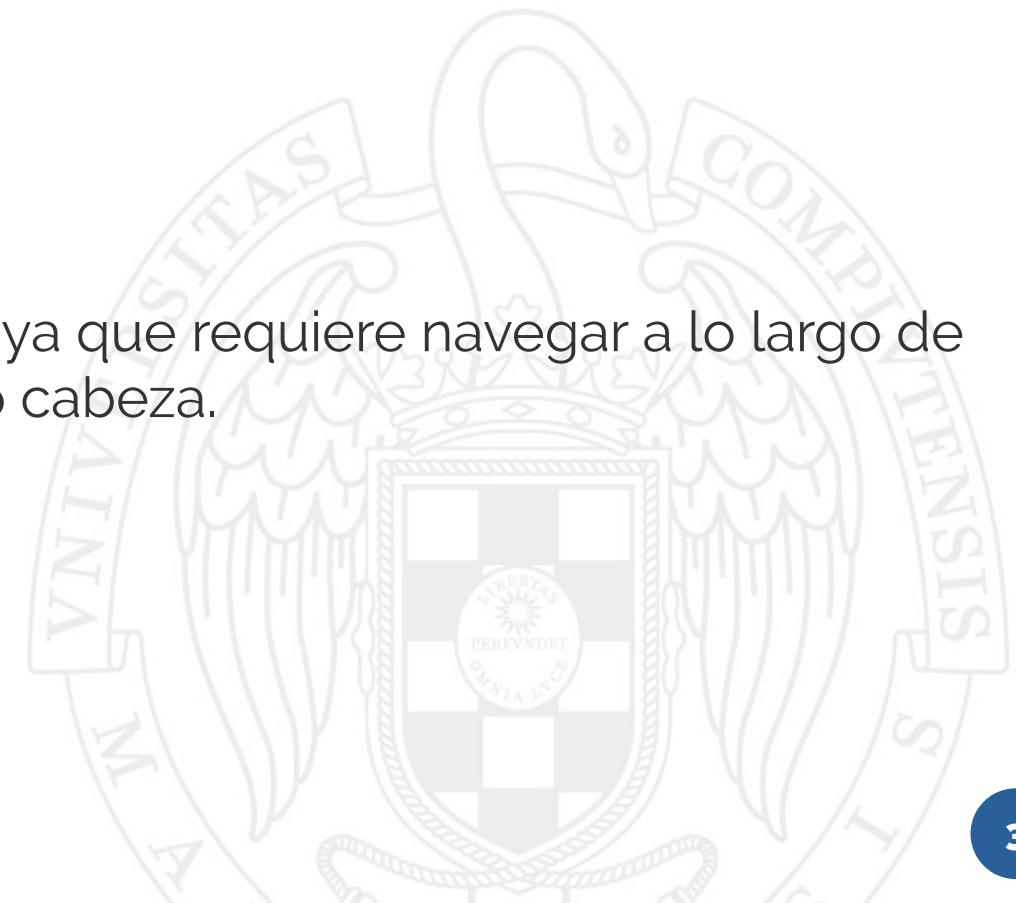
$[]$



head

# Mejorando algunas operaciones

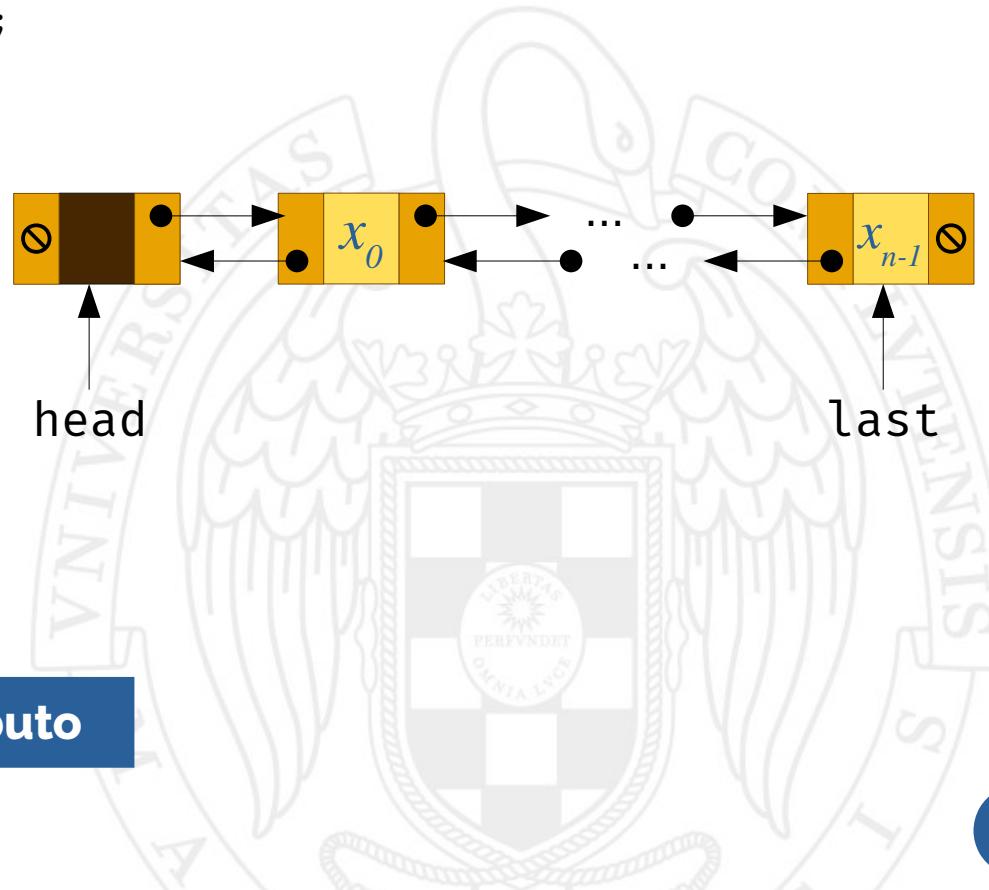
- Las siguientes operaciones requieren situarnos en el último nodo de la lista:
  - `push_back()`
  - `pop_back()`
  - `back()`
- Esto hace que tengan coste lineal, ya que requiere navegar a lo largo de toda la cadena, partiendo del nodo cabeza.
- ¿Podemos mejorar esto?



# Añadiendo un nuevo atributo

```
class ListLinkedDouble {  
public:  
    ListLinkedDouble();  
    ListLinkedDouble(const ListLinkedDouble &other);  
    ~ListLinkedDouble();  
  
    void push_front(const std::string &elem);  
    void push_back(const std::string &elem);  
    void pop_front();  
    void pop_back();  
    int size() const;  
    bool empty() const;  
    const std::string & front() const;  
    std::string & front();  
    const std::string & back() const;  
    std::string & back();  
    const std::string & at(int index) const;  
    std::string & at(int index);  
    void display() const;  
private:  
    ...  
    Node *head, *last;
```

Nuevo atributo

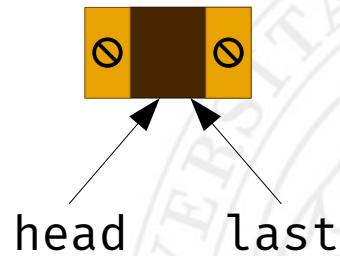


# Añadiendo un nuevo atributo

- **Ventajas:**
  - La operación privada `last_node()` pasa a tener coste constante, ya que se limita a devolver el atributo `last`.
  - De hecho, podemos eliminar la función `last_node()`.
- **Desventajas:**
  - Tenemos que actualizar el atributo `last` cada vez que añadamos un nodo.

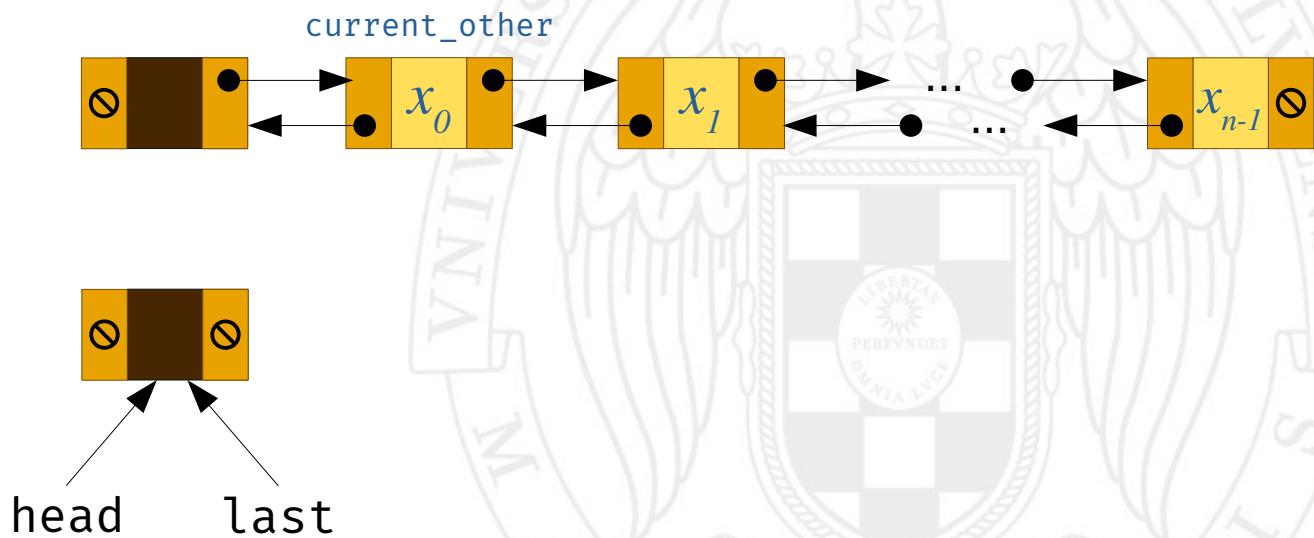
# Creación de una cadena de nodos

```
ListLinkedDouble() {  
    head = new Node;  
    head→next = nullptr;  
    head→prev = nullptr;  
    last = head;  
}
```



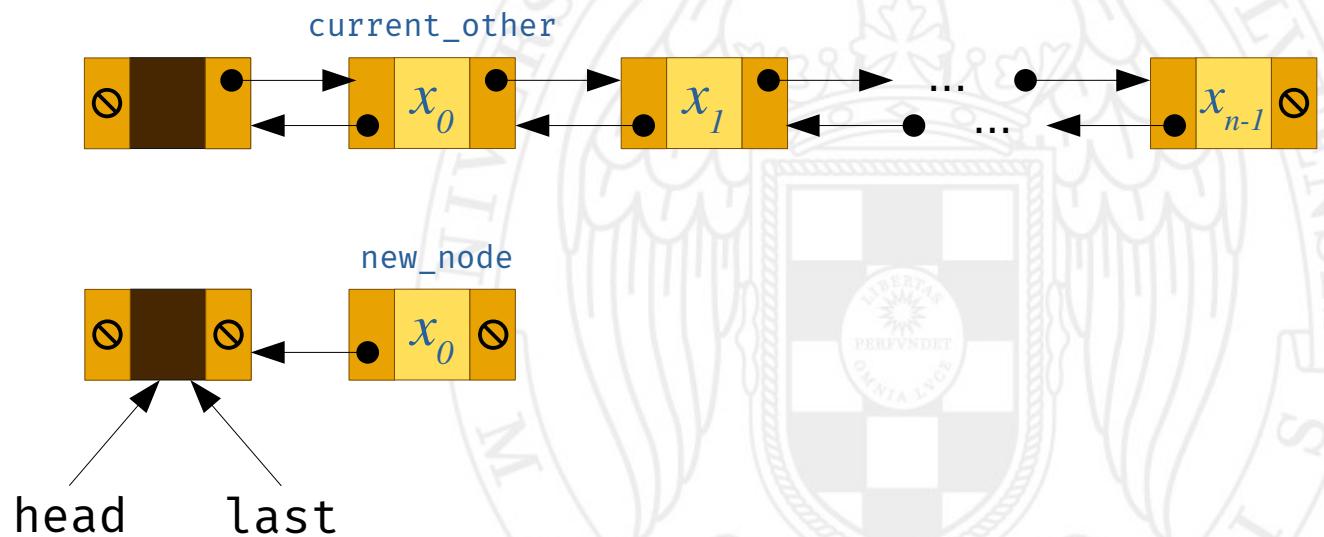
# Copia de una cadena de nodos

```
ListLinkedDouble(const ListLinkedDouble &other): ListLinkedDouble() {  
    Node *current_other = other.head->next;  
  
    while (current_other != nullptr) {  
        Node *new_node = new Node { current_other->value, nullptr, last };  
        last->next = new_node;  
  
        last = new_node;  
        current_other = current_other->next;  
    }  
}
```



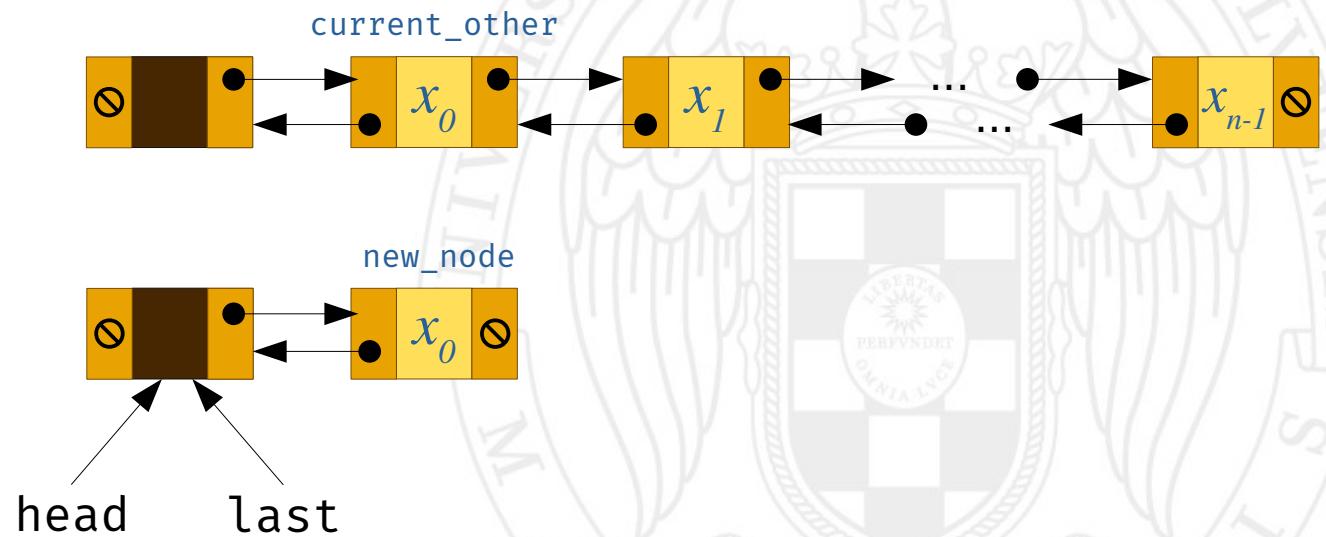
# Copia de una cadena de nodos

```
ListLinkedDouble(const ListLinkedDouble &other): ListLinkedDouble() {  
    Node *current_other = other.head->next;  
  
    while (current_other != nullptr) {  
        Node *new_node = new Node { current_other->value, nullptr, last };  
        last->next = new_node;  
        last = new_node;  
        current_other = current_other->next;  
    }  
}
```



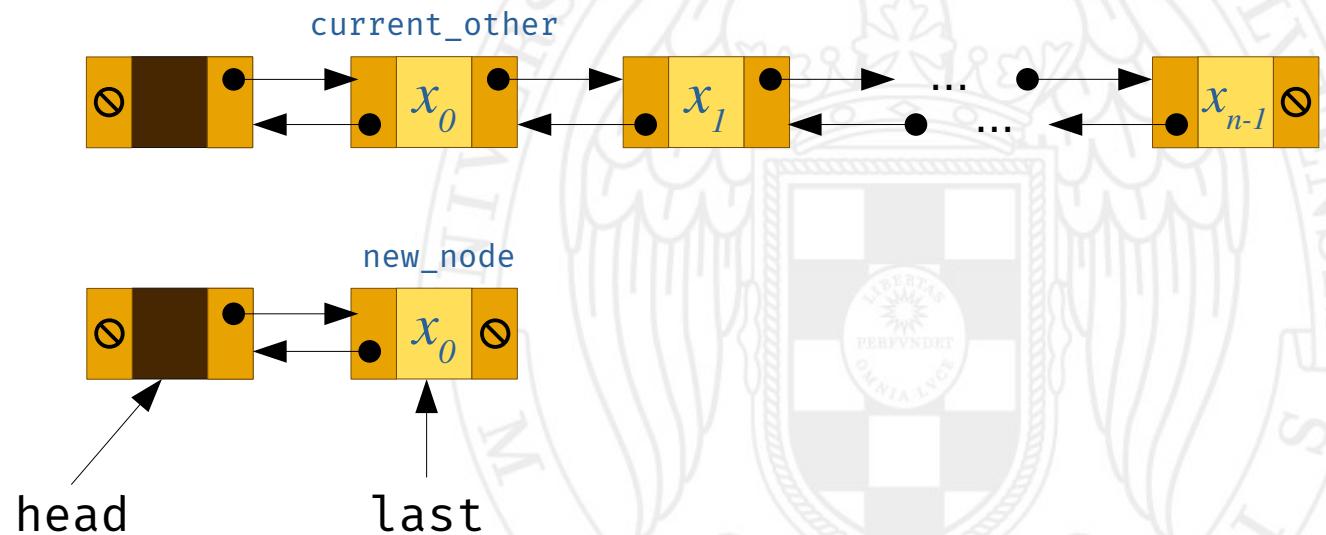
# Copia de una cadena de nodos

```
ListLinkedDouble(const ListLinkedDouble &other): ListLinkedDouble() {  
    Node *current_other = other.head->next;  
  
    while (current_other != nullptr) {  
        Node *new_node = new Node { current_other->value, nullptr, last };  
        last->next = new_node;  
        last = new_node;  
        current_other = current_other->next;  
    }  
}
```



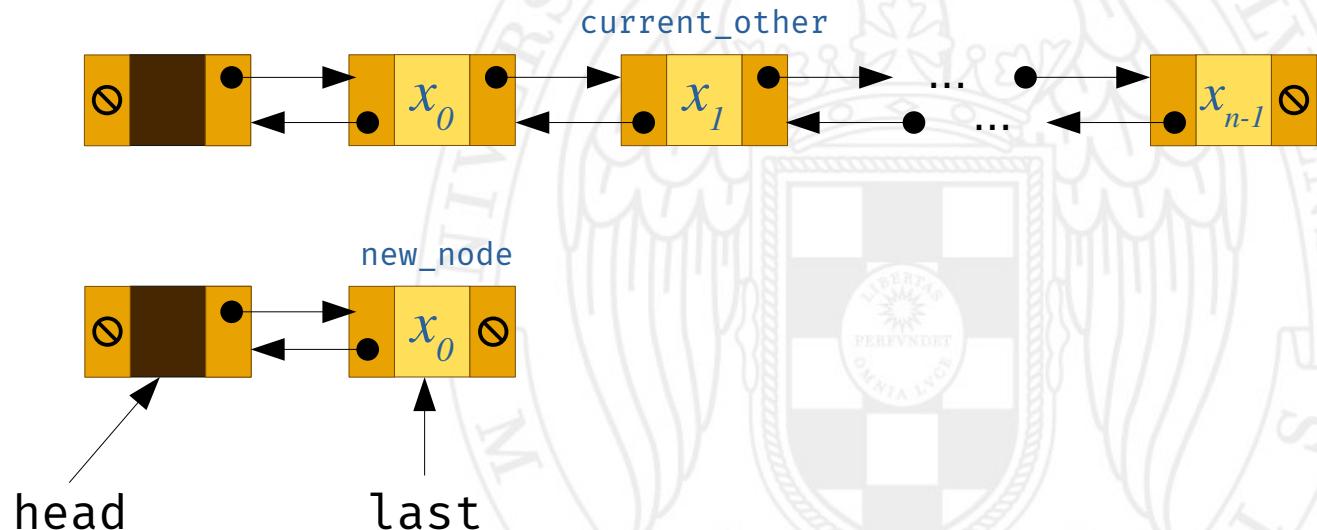
# Copia de una cadena de nodos

```
ListLinkedDouble(const ListLinkedDouble &other): ListLinkedDouble() {  
    Node *current_other = other.head->next;  
  
    while (current_other != nullptr) {  
        Node *new_node = new Node { current_other->value, nullptr, last };  
        last->next = new_node;  
        last = new_node;  
        current_other = current_other->next;  
    }  
}
```



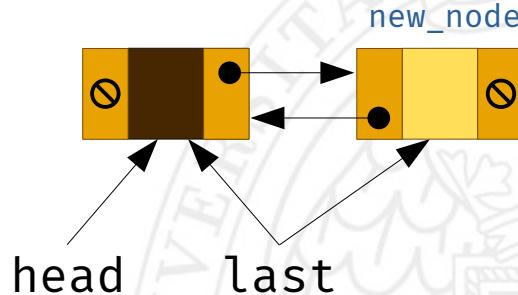
# Copia de una cadena de nodos

```
ListLinkedDouble(const ListLinkedDouble &other): ListLinkedDouble() {  
    Node *current_other = other.head->next;  
  
    while (current_other != nullptr) {  
        Node *new_node = new Node { current_other->value, nullptr, last };  
        last->next = new_node;  
        last = new_node;  
        current_other = current_other->next;  
    }  
}
```



# Añadir al principio de la lista

```
void push_front(const std::string &elem) {  
    Node *new_node = new Node { elem, head->next, head };  
    if (head->next != nullptr) {  
        head->next->prev = new_node;  
    }  
    head->next = new_node;  
    if (last == head) {  
        last = new_node;  
    }  
}
```



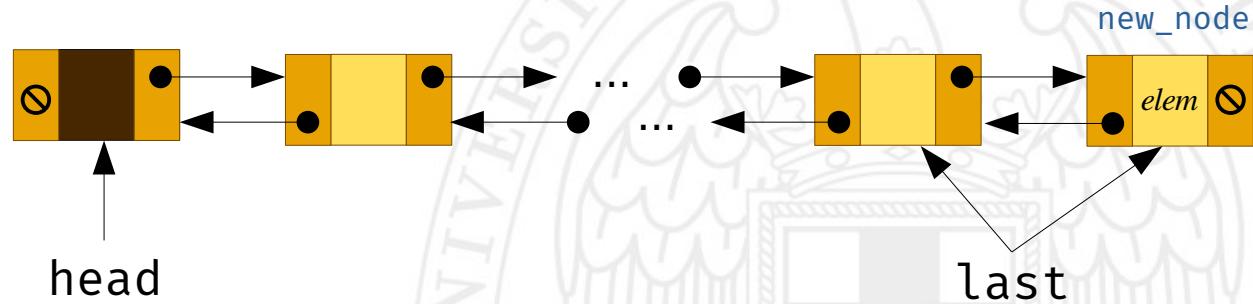
# Eliminar al principio de la lista

```
void pop_front() {
    assert (head->next != nullptr);
    Node *target = head->next;
    head->next = target->next;
    if (target->next != nullptr) {
        target->next->prev = head;
    }
    if (last == target) {
        last = head;
    }
    delete target;
}
```



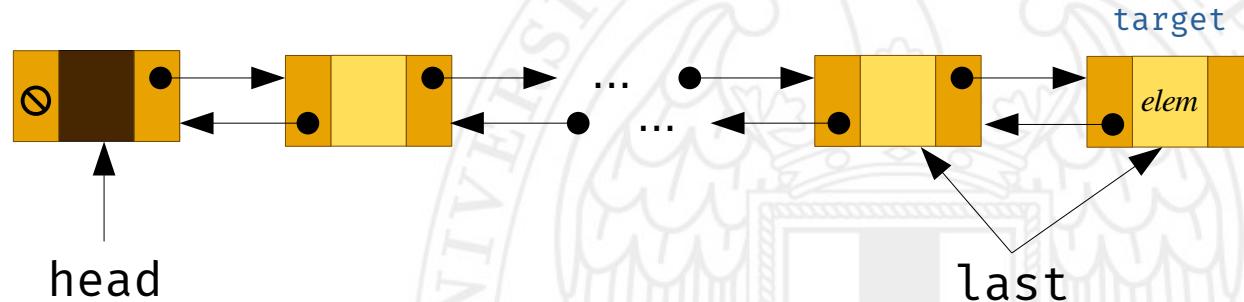
# Añadir al final de la lista

```
void push_back(const std::string &elem) {  
    Node *new_node = new Node { elem, nullptr, last };  
    last->next = new_node;  
    last = new_node;  
}
```



# Eliminar del final de la lista

```
void pop_back() {
    assert (head->next != nullptr);
    Node *target = last;
    target->prev->next = nullptr;
    last = target->prev;
    delete target;
}
```



# ¿Mejoras en el coste?

Operación	Listas enlazadas simples	Listas doblemente enlazadas
Creación	$O(1)$	$O(1)$
Copia	$O(n)$	$O(n)$
push_back	$O(n)$	$O(1)$
push_front	$O(1)$	$O(1)$
pop_back	$O(n)$	$O(1)$
pop_front	$O(1)$	$O(1)$
back	$O(n)$	$O(1)$
front	$O(1)$	$O(1)$
display	$O(n)$	$O(n)$
at(index)	$O(index)$	$O(index)$
size	$O(n)$	$O(n)$
empty	$O(1)$	$O(1)$

$n$  = número de elementos de la lista de entrada

# ¿Podemos mejorar size( )?

- Sí. Para ello añadimos un nuevo atributo num\_elems a la clase que mantenga el número de elementos en la lista.
- La función size( ) devuelve el valor de este atributo.
- Actualizamos este elemento al añadir/quitar elementos de la lista.

```
class ListLinkedDouble {  
public:  
    ...  
    int size() const { return num_elems; }  
private:  
    ...  
    Node *head, *last;  
    int num_elems;  
};
```

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

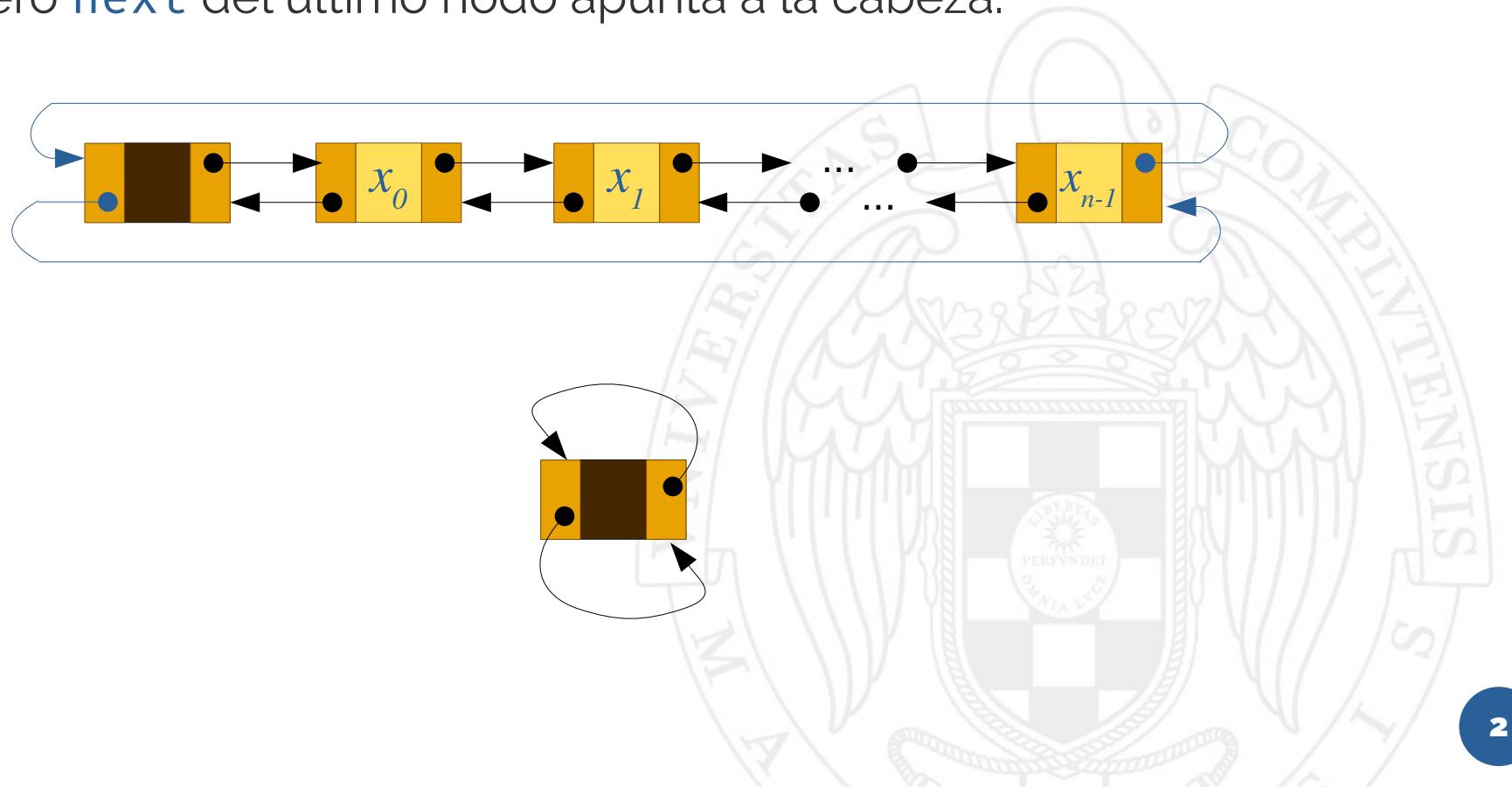
# Listas enlazadas circulares

Manuel Montenegro Montes

Departamento de Sistemas Informáticos y Computación  
Facultad de Informática – Universidad Complutense de Madrid

# Listas doblemente enlazadas circulares

- El puntero `prev` de la cabeza apunta al último nodo.
- El puntero `next` del último nodo apunta a la cabeza.



# Consecuencias

- No hay punteros nulos en la cadena.
- No es necesario un atributo `last` en la clase `ListLinkedDouble` que apunte al último nodo.
  - En su lugar: `head→prev`.
- Se simplifican algunas operaciones.
- ¡Cuidado al iterar sobre los nodos!

```
current = head→next;  
while (current ≠ nullptr) { !  
    ...  
    current = current→next;  
}
```



```
current = head→next;  
while (current ≠ head) { ✓  
    ...  
    current = current→next;  
}
```

# Eliminamos atributo last

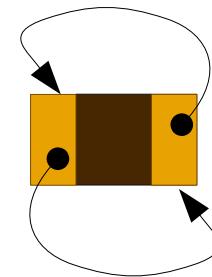
```
class ListLinkedDouble {  
public:  
    ListLinkedDouble();  
    ListLinkedDouble(const ListLinkedDouble &other);  
    ~ListLinkedDouble();  
  
    void push_front(const std::string &elem);  
    void push_back(const std::string &elem);  
    void pop_front();  
    void pop_back();  
    int size() const;  
    bool empty() const;  
    const std::string & front() const;  
    std::string & front();  
    const std::string & back() const;  
    std::string & back();  
    const std::string & at(int index) const;  
    std::string & at(int index);  
    void display() const;  
private:  
    ...  
    Node *head, *last;  
    int num_elems;  
};
```

← Eliminar ~~\*last~~



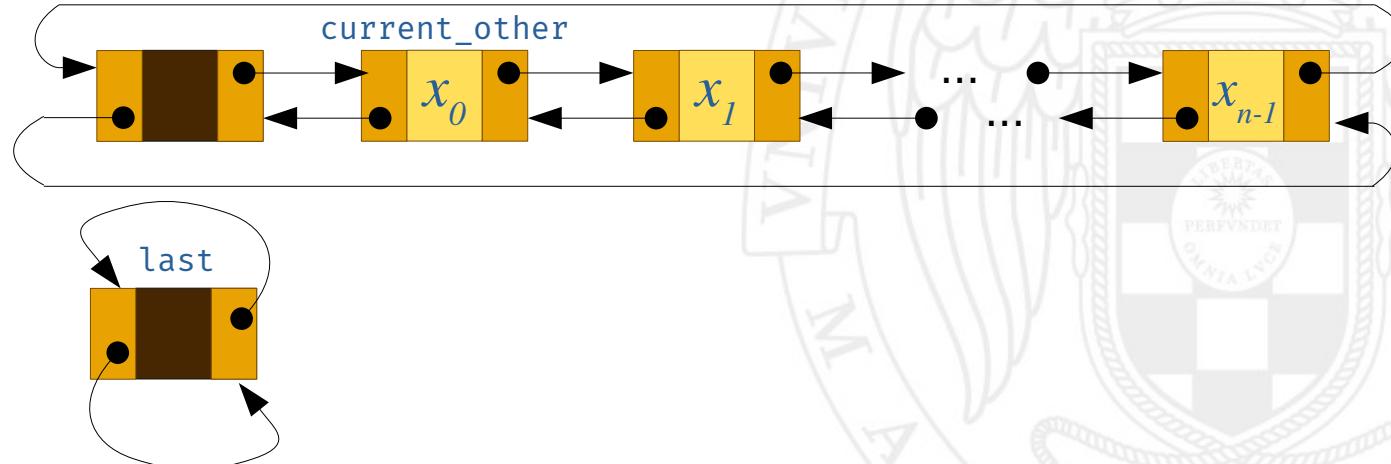
# Creación de una lista

```
ListLinkedDouble(): num_elems(0) {  
    head = new Node;  
    head→next = head;  
    head→prev = head;  
}
```



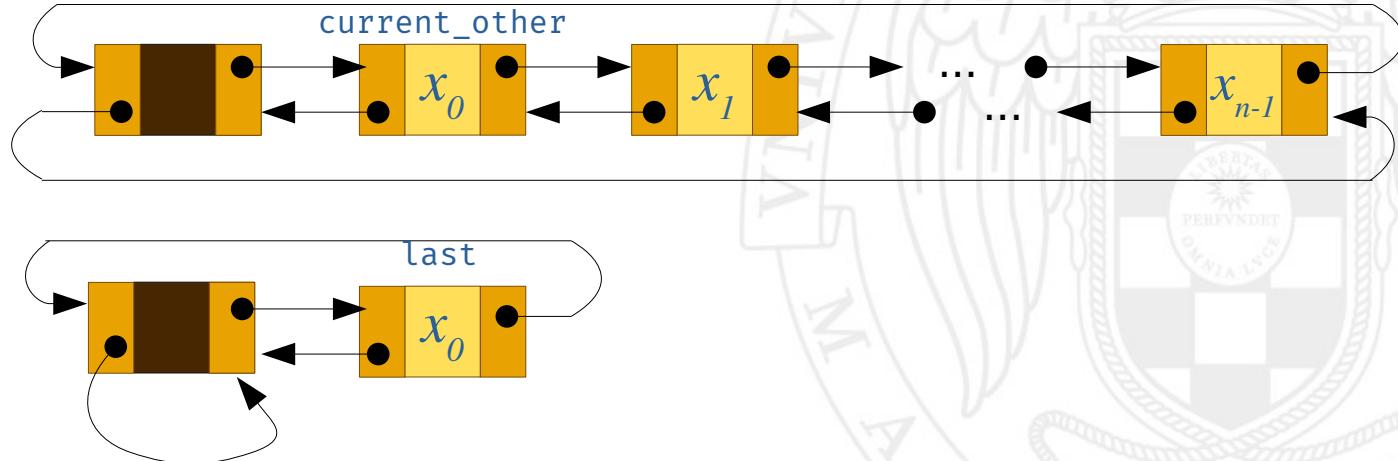
# Copia de una cadena de nodos

```
ListLinkedDouble(const ListLinkedDouble &other): ListLinkedDouble() {  
    Node *current_other = other.head->next;  
    Node *last = head;  
  
    while (current_other != other.head) {  
        Node *new_node = new Node { current_other->value, head, last };  
        last->next = new_node;  
        last = new_node;  
        current_other = current_other->next;  
    }  
    head->prev = last;  
    num_elems = other.num_elems;  
}
```



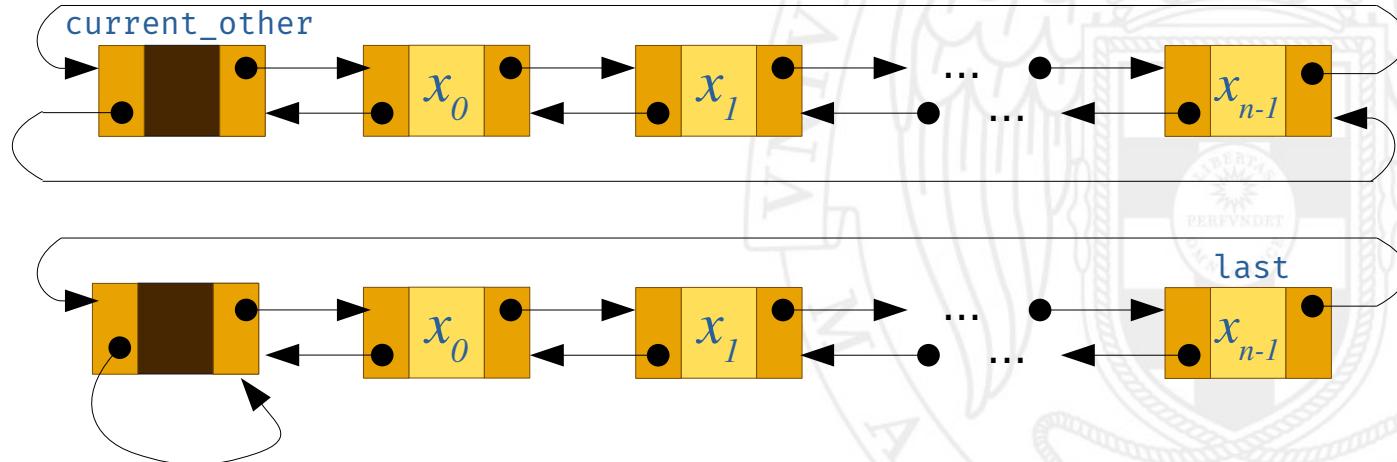
# Copia de una cadena de nodos

```
ListLinkedDouble(const ListLinkedDouble &other): ListLinkedDouble() {  
    Node *current_other = other.head->next;  
    Node *last = head;  
  
    while (current_other != other.head) {  
        Node *new_node = new Node { current_other->value, head, last };  
        last->next = new_node;  
        last = new_node;  
        current_other = current_other->next;  
    }  
    head->prev = last;  
    num_elems = other.num_elems;  
}
```



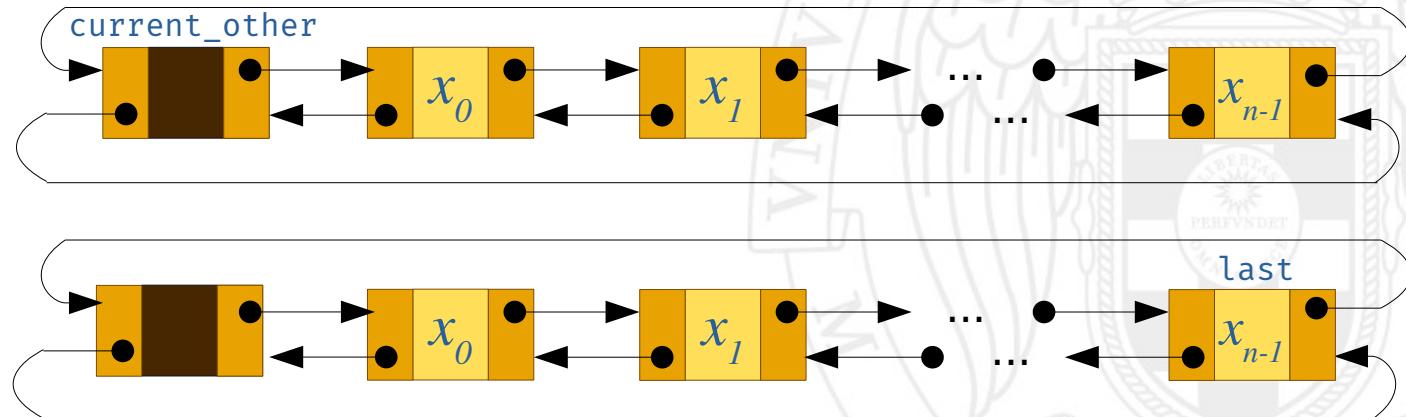
# Copia de una cadena de nodos

```
ListLinkedDouble(const ListLinkedDouble &other): ListLinkedDouble() {  
    Node *current_other = other.head->next;  
    Node *last = head;  
  
    while (current_other != other.head) {  
        Node *new_node = new Node { current_other->value, head, last };  
        last->next = new_node;  
        last = new_node;  
        current_other = current_other->next;  
    }  
    head->prev = last;  
    num_elems = other.num_elems;  
}
```



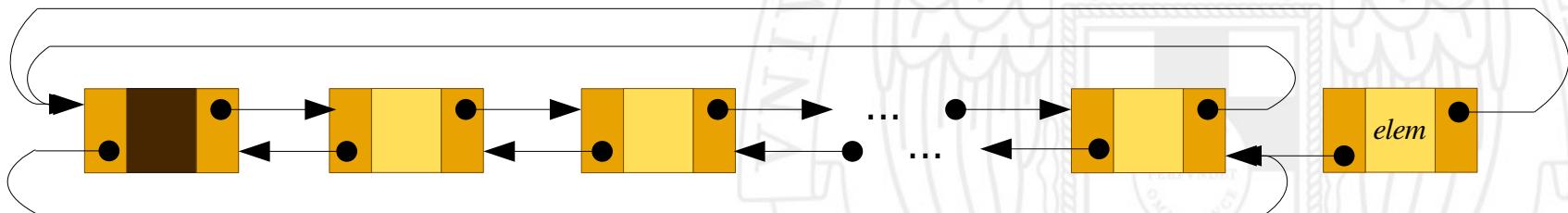
# Copia de una cadena de nodos

```
ListLinkedDouble(const ListLinkedDouble &other): ListLinkedDouble() {  
    Node *current_other = other.head->next;  
    Node *last = head;  
  
    while (current_other != other.head) {  
        Node *new_node = new Node { current_other->value, head, last };  
        last->next = new_node;  
        last = new_node;  
        current_other = current_other->next;  
    }  
    head->prev = last;  
    num_elems = other.num_elems;  
}
```



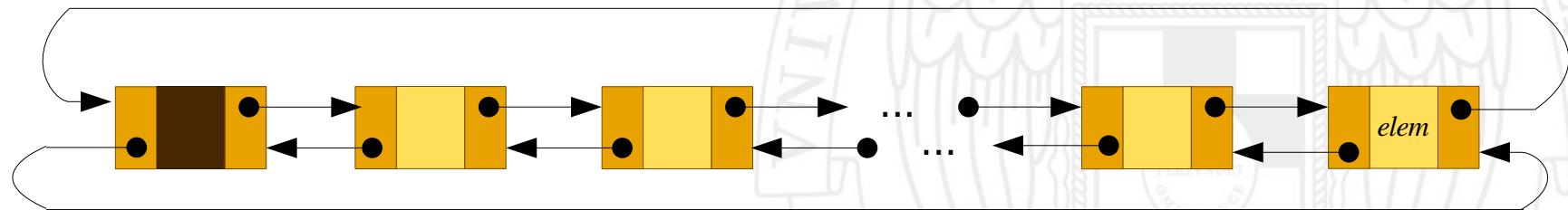
# Añadir elementos

```
void push_front(const std::string &elem) {  
    Node *new_node = new Node { elem, head->next, head };  
    head->next->prev = new_node;  
    head->next = new_node;  
    num_elems++;  
}  
  
void push_back(const std::string &elem) {  
    Node *new_node = new Node { elem, head, head->prev };  
    head->prev->next = new_node;  
    head->prev = new_node;  
    num_elems++;  
}
```



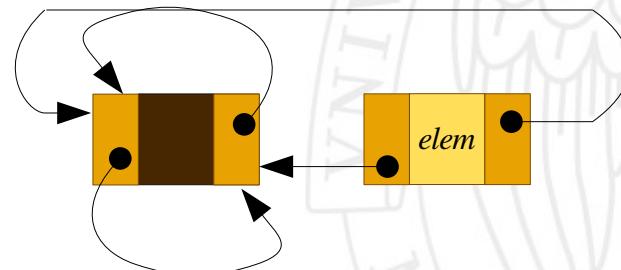
# Añadir elementos

```
void push_front(const std::string &elem) {  
    Node *new_node = new Node { elem, head->next, head };  
    head->next->prev = new_node;  
    head->next = new_node;  
    num_elems++;  
}  
  
void push_back(const std::string &elem) {  
    Node *new_node = new Node { elem, head, head->prev };  
    head->prev->next = new_node;  
    head->prev = new_node;  
    num_elems++;  
}
```



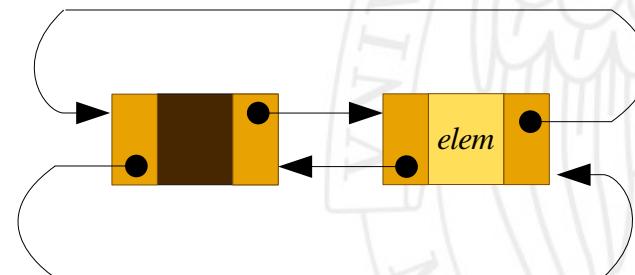
# Añadir elementos

```
void push_front(const std::string &elem) {  
    Node *new_node = new Node { elem, head->next, head };  
    head->next->prev = new_node;  
    head->next = new_node;  
    num_elems++;  
}  
  
void push_back(const std::string &elem) {  
    Node *new_node = new Node { elem, head, head->prev };  
    head->prev->next = new_node;  
    head->prev = new_node;  
    num_elems++;  
}
```



# Añadir elementos

```
void push_front(const std::string &elem) {  
    Node *new_node = new Node { elem, head->next, head };  
    head->next->prev = new_node;  
    head->next = new_node;  
    num_elems++;  
}  
  
void push_back(const std::string &elem) {  
    Node *new_node = new Node { elem, head, head->prev };  
    head->prev->next = new_node;  
    head->prev = new_node;  
    num_elems++;  
}
```



# Eliminar elementos

```
void pop_front() {
    assert (num_elems > 0);
    Node *target = head->next;
    head->next = target->next;
    target->next->prev = head;
    delete target;
    num_elems--;
}

void pop_back() {
    assert (num_elems > 0);
    Node *target = head->prev;
    target->prev->next = head;
    head->prev = target->prev;
    delete target;
    num_elems--;
}
```



# Coste de las operaciones

Operación	Coste en tiempo
Creación	$O(1)$
Copia	$O(n)$
push_back	$O(1)$
push_front	$O(1)$
pop_back	$O(1)$
pop_front	$O(1)$
back	$O(1)$
front	$O(1)$
display	$O(n)$
at(index)	$O(index)$
size	$O(1)$
empty	$O(1)$

$n$  = número de elementos de la lista de entrada

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

# Sobrecargando operadores en el TAD Lista

Manuel Montenegro Montes

Departamento de Sistemas Informáticos y Computación  
Facultad de Informática – Universidad Complutense de Madrid

# Sobrecarga del operador <<

# Generalizando el método `display()`

```
class ListArray {
public:
    void display() const;
    ...
};

void ListArray::display() const {
    std::cout << "[";
    if (num_elems > 0) {
        std::cout << elems[0];
        for (int i = 1; i < num_elems; i++) {
            std::cout << ", " << elems[i];
        }
    }
    std::cout << "]";
}
```



# Generalizando el método `display()`

```
class ListArray {  
public:  
    void display(std::ostream &out) const;  
    ...  
};  
  
void ListArray::display(std::ostream &out) const {  
    out << "[";  
    if (num_elems > 0) {  
        out << elems[0];  
        for (int i = 1; i < num_elems; i++) {  
            out << ", " << elems[i];  
        }  
    }  
    out << "]";  
}
```



# Sobrecargando el operador <<

```
class ListArray {  
public:  
    void display(std::ostream &out) const;  
    ...  
};  
  
std::ostream & operator<<(std::ostream &out, const ListArray &l) {  
    l.display(out);  
    return out;  
}
```



# Ejemplo

```
ListArray l1;  
l1.push_back("David");  
l1.push_back("Maria");  
l1.push_back("Elvira");  
  
ListArray l2 = l1;  
l2.at(1) = "Manuel";  
  
std::cout << l1 << " " << l2 << std::endl;
```

[David, Maria, Elvira] [David, Manuel, Elvira]

# Sobrecarga del operador [ ]

# Acceso a elementos de una lista

- El método `at(i)` nos permitía acceder a la posición  $i$ -ésima de una lista.

```
std::cout << l.at(1);
l.at(2) = "Francisco";
```

- Resultaría más intuitiva una notación similar a la de los arrays.

```
std::cout << l[1];
l[2] = "Francisco";
```

- Es posible habilitar esta notación sobrecargando el operador `[ ]`.
  - Para ello hay que definir un método llamado `operator[]` en la clase lista.
  - La expresión `l[i]` equivale a `l.operator[](i)`

# Sobrecarga del operador []

```
class ListArray {  
public:  
    ...  
    const std::string & at(int index) const {  
        assert (0 <= index && index < num_elems);  
        return elems[index];  
    }  
  
    std::string & at(int index) {  
        assert (0 <= index && index < num_elems);  
        return elems[index];  
    }  
  
    ...  
};
```



# Sobrecarga del operador []

```
class ListArray {  
public:  
    ...  
    const std::string & operator[](int index) const {  
        assert (0 <= index && index < num_elems);  
        return elems[index];  
    }  
  
    std::string & operator[](int index) {  
        assert (0 <= index && index < num_elems);  
        return elems[index];  
    }  
    ...  
};
```



# Ejemplo

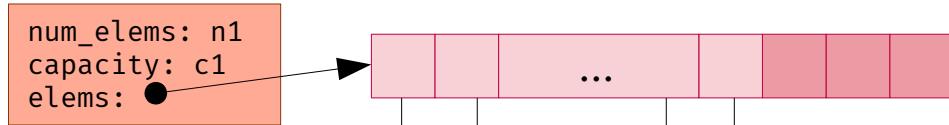
```
ListArray l;  
l.push_back("David");  
l.push_back("Maria");  
l.push_back("Elvira");  
l[2] = "Enriqueta";  
std::cout << l << std::endl;
```

[David, Maria, Enriqueta]

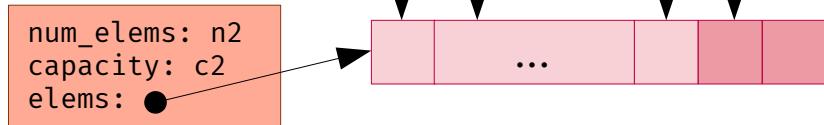
# Sobrecarga del operador de asignación

# Listas mediante arrays

`l1`



`l2`

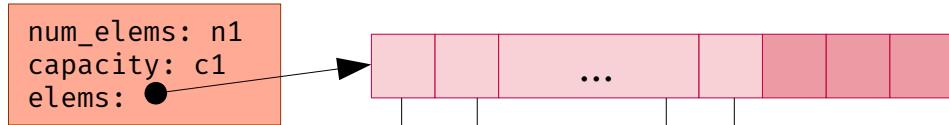


Al hacer la asignación `l2 = l1`.

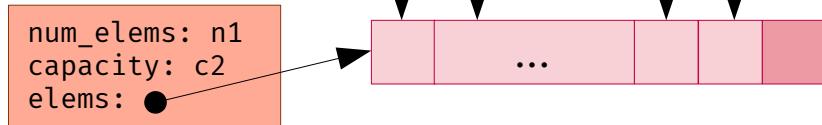
- 1) Si los elementos de `l1` caben en `l2`:
  - 1) Copiarlos de `l1` a `l2`.
  - 2) Copiar el atributo `num_elems`.

# Listas mediante arrays

`l1`



`l2`

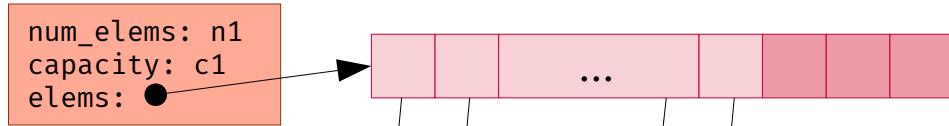


Al hacer la asignación `l2 = l1`.

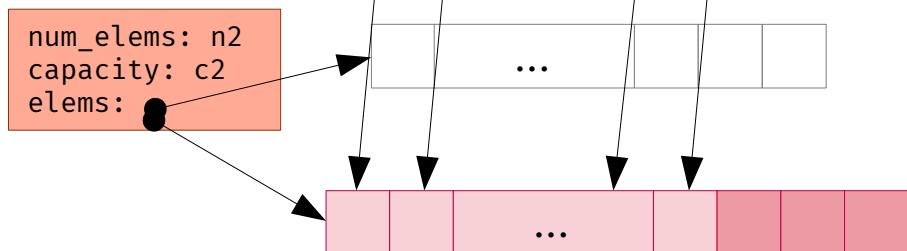
- 1) Si los elementos de `l1` caben en `l2`:
  - 1) Copiarlos de `l1` a `l2`.
  - 2) Copiar el atributo `num_elems`.

# Listas mediante arrays

`l1`



`l2`

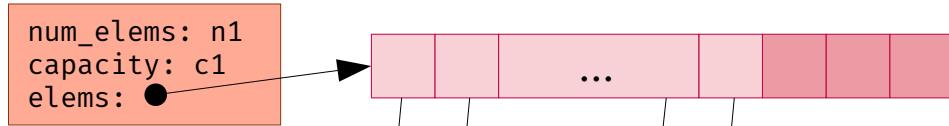


Al hacer la asignación `l2 = l1`.

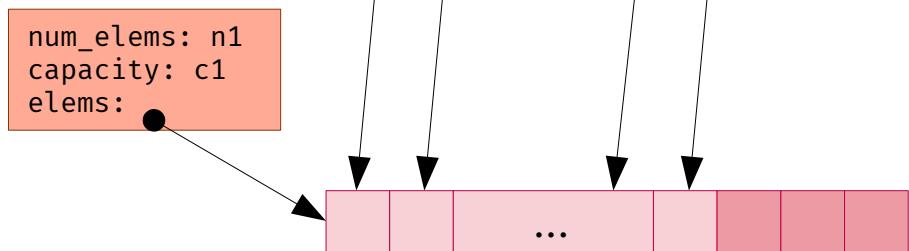
- 1) Si los elementos de `l1` caben en `l2`:
  - 1) Copiarlos de `l1` a `l2`.
  - 2) Copiar el atributo `num_elems`.
- 2) En caso contrario:
  - 1) Desechar `l2.elems` y reemplazarlo por otro array con la misma capacidad que el de `l1`.
  - 2) Copiar el atributo `capacity`.
  - 3) Copiar los elementos del array `elems` de `l1` a `l2`.
  - 4) Copiar el atributo `num_elems`.

# Listas mediante arrays

`l1`



`l2`



Al hacer la asignación `l2 = l1`.

- 1) Si los elementos de `l1` caben en `l2`:
  - 1) Copiarlos de `l1` a `l2`.
  - 2) Copiar el atributo `num_elems`.
- 2) En caso contrario:
  - 1) Desechar `l2.elems` y reemplazarlo por otro array con la misma capacidad que el de `l2`.
  - 2) Copiar el atributo `capacity`.
  - 3) Copiar los elementos del array `elems` de `l1` a `l2`.
  - 4) Copiar el atributo `num_elems`.

# Listas mediante arrays

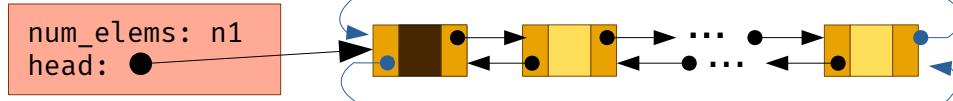
```
ListArray & operator=(const ListArray &other) {  
  
    if (this != &other) {  
        if (capacity < other.num_elems) {  
            delete[] elems;  
            elems = new std::string[other.capacity];  
            capacity = other.capacity;  
        }  
        num_elems = other.num_elems;  
        for (int i = 0; i < num_elems; i++) {  
            elems[i] = other.elems[i];  
        }  
    }  
    return *this;  
}
```

Al hacer la asignación `l2 = l1`.

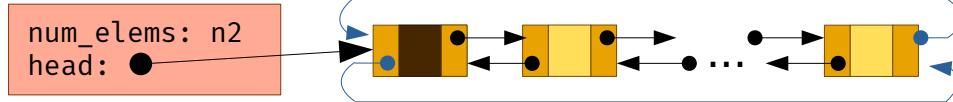
- 1) Si los elementos de `l1` caben en `l2`:
  - 1) Copiarlos de `l1` a `l2`.
  - 2) Copiar el atributo `num_elems`.
- 2) En caso contrario:
  - 1) Desechar `l2.elems` y reemplazarlo por otro array con la misma capacidad que el de `l2`.
  - 2) Copiar el atributo `capacity`.
  - 3) Copiar los elementos del array `elems` de `l1` a `l2`.
  - 4) Copiar el atributo `num_elems`.

# Listas doblemente enlazadas circulares

`l1`



`l2`

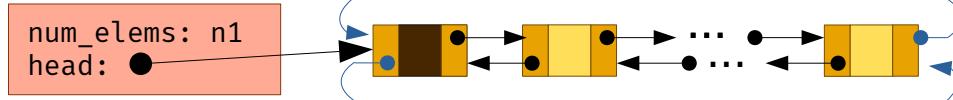


Al hacer la asignación `l2 = l1`.

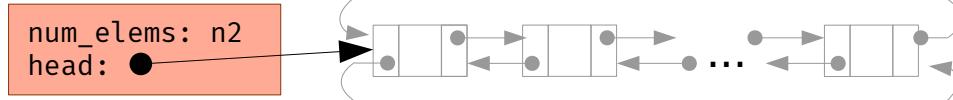
- 1) Borrar la cadena de nodos de `l2`.
- 2) Crear nodo fantasma en `l2` y hacer que `l2.head` apunte a él.
- 3) Hacer copias de los nodos de `l1` y encadenarlos en `l2` (similar al constructor de copia).
- 4) Copiar atributo `num_elems` de `l1` a `l2`.

# Listas doblemente enlazadas circulares

`l1`



`l2`

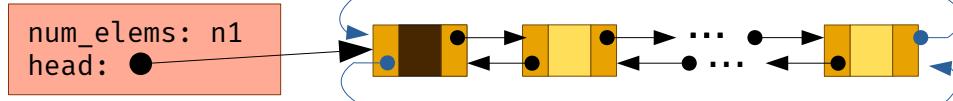


Al hacer la asignación `l2 = l1`.

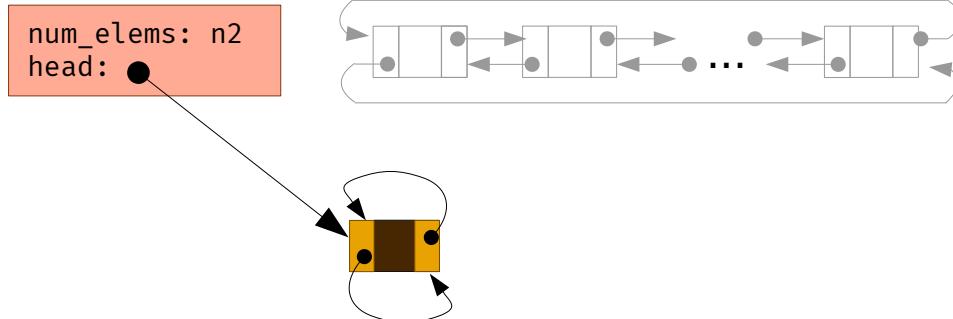
- 1) Borrar la cadena de nodos de `l2`.
- 2) Crear nodo fantasma en `l2` y hacer que `l2.head` apunte a él.
- 3) Hacer copias de los nodos de `l1` y encadenarlos en `l2` (similar al constructor de copia).
- 4) Copiar atributo `num_elems` de `l1` a `l2`.

# Listas doblemente enlazadas circulares

`l1`



`l2`



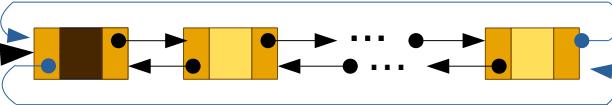
Al hacer la asignación `l2 = l1`.

- 1) Borrar la cadena de nodos de `l2`.
- 2) Crear nodo fantasma en `l2` y hacer que `l2.head` apunte a él.
- 3) Hacer copias de los nodos de `l1` y encadenarlos en `l2` (similar al constructor de copia).
- 4) Copiar atributo `num_elems` de `l1` a `l2`.

# Listas doblemente enlazadas circulares

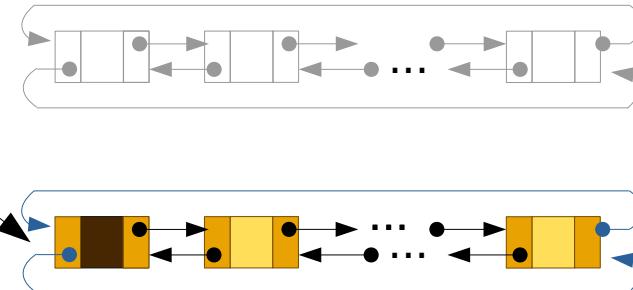
`l1`

`num_elems: n1`  
`head:` ●



`l2`

`num_elems: n2`  
`head:` ●



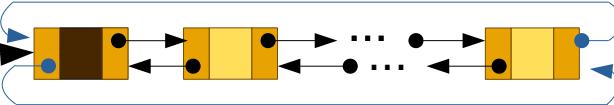
Al hacer la asignación `l2 = l1`.

- 1) Borrar la cadena de nodos de `l2`.
- 2) Crear nodo fantasma en `l2` y hacer que `l2.head` apunte a él.
- 3) Hacer copias de los nodos de `l1` y encadenarlos en `l2` (similar al constructor de copia).
- 4) Copiar atributo `num_elems` de `l1` a `l2`.

# Listas doblemente enlazadas circulares

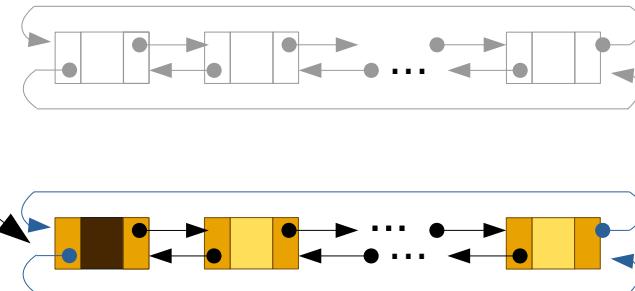
`l1`

`num_elems: n1`  
`head:` ●



`l2`

`num_elems: n1`  
`head:` ●



Al hacer la asignación `l2 = l1`.

- 1) Borrar la cadena de nodos de `l2`.
- 2) Crear nodo fantasma en `l2` y hacer que `l2.head` apunte a él.
- 3) Hacer copias de los nodos de `l1` y encadenarlos en `l2` (similar al constructor de copia).
- 4) Copiar atributo `num_elems` de `l1` a `l2`.

# Listas doblemente enlazadas circulares

```
ListLinkedDouble &  
operator=(const ListLinkedDouble &other) {  
  
    if (this != &other) {  
        delete_nodes();  
        head = new Node;  
        head->next = head->prev = head;  
        copy_nodes_from(other);  
        num_elems = other.num_elems;  
    }  
    return *this;  
}
```

Al hacer la asignación `l2 = l1`.

- 1) Borrar la cadena de nodos de `l2`.
- 2) Crear nodo fantasma en `l2` y hacer que `l2.head` apunte a él.
- 3) Hacer copias de los nodos de `l1` y encadenarlos en `l2` (similar al constructor de copia).
- 4) Copiar atributo `num_elems` de `l1` a `l2`.

ESTRUCTURAS DE DATOS

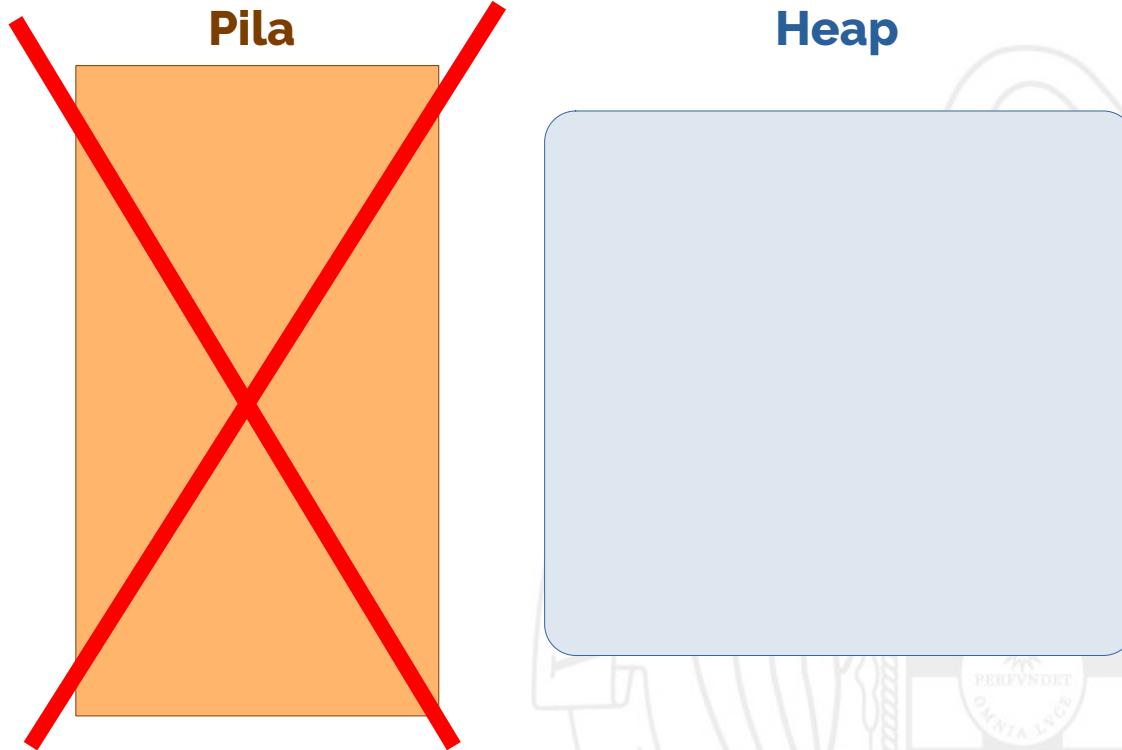
TIPOS ABSTRACTOS DE DATOS LINEALES

# El TAD Pila

Manuel Montenegro Montes

Departamento de Sistemas Informáticos y Computación  
Facultad de Informática – Universidad Complutense de Madrid

# ¿Qué es una pila?

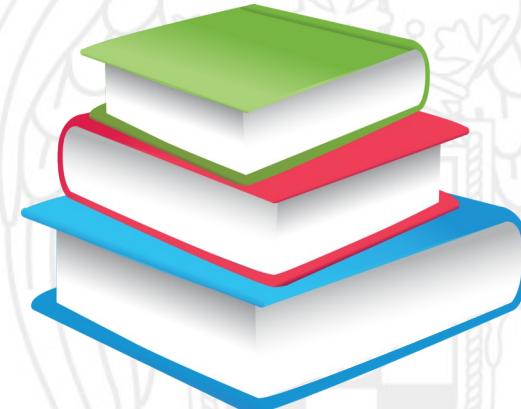


# ¿Qué es una pila?

- Es una colección de elementos que permite:
  - Insertar elementos.
  - Obtener o borrar el último elemento insertado que no haya sido borrado previamente.



Foto: John Leffmann (CC BY 3.0)



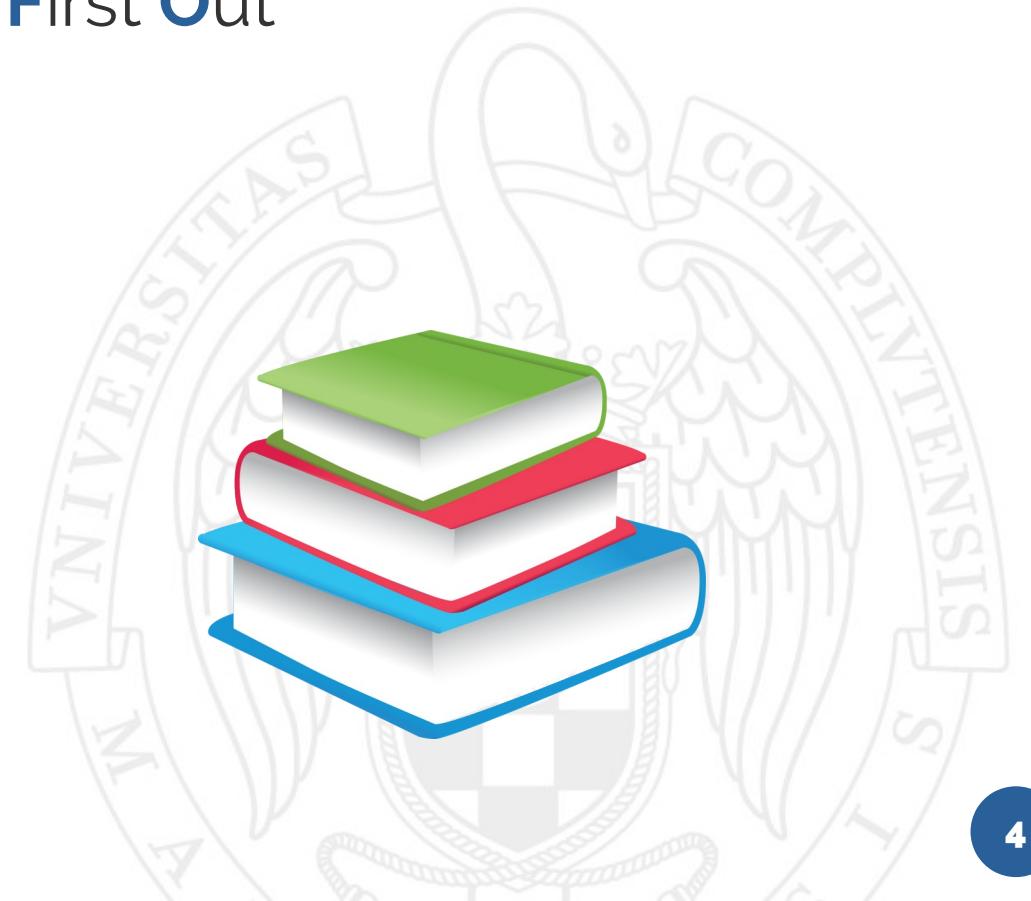
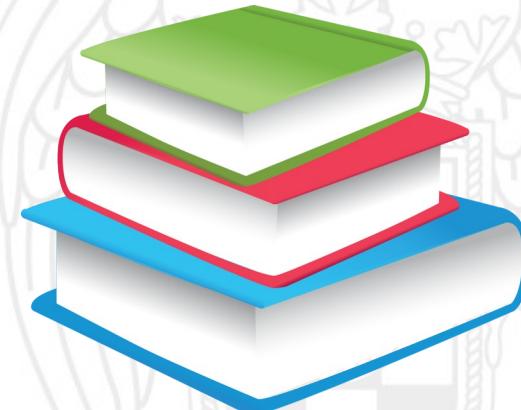
# ¿Qué es una pila?

- Las pilas reciben el nombre de estructuras de acceso **LIFO**

**Last In, First Out**

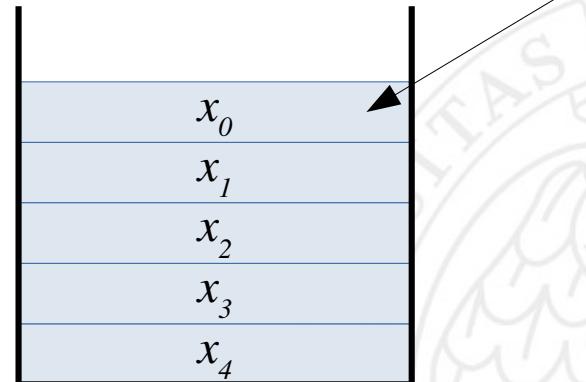


Foto: John Leffmann (CC BY 3.0)



# Modelo de pilas

- Conceptualmente representamos las pilas de esta forma:



**Cima** de la pila:  
último elemento  
insertado.

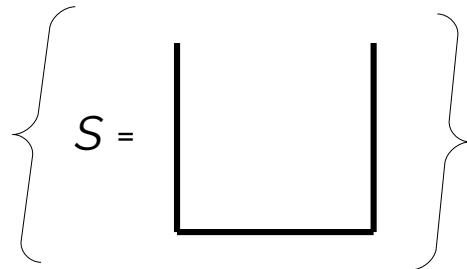
# Operaciones sobre pilas

- **Constructoras:**
  - Crear una pila vacía (***create\_empty***).
- **Mutadoras:**
  - Añadir elemento en la cima de la pila (***push***).
  - Eliminar elemento en la cima de la pila (***pop***).
- **Observadoras:**
  - Obtener el elemento en la cima de la pila (***top***).
  - Saber si una pila está vacía (***empty***).

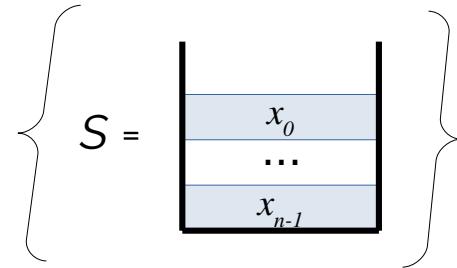
# Operación *create\_empty*

{ *true* }

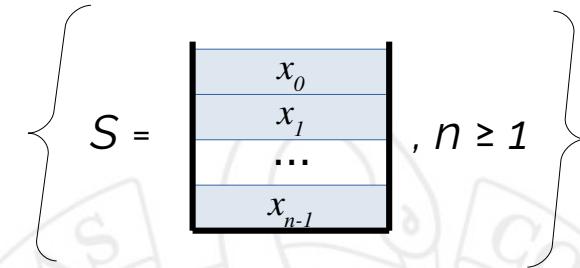
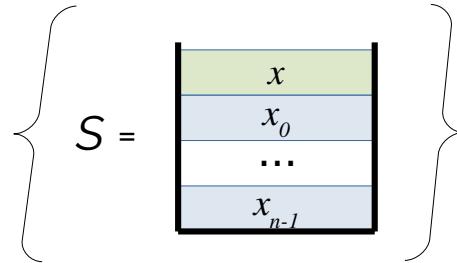
***create\_empty()*** → (*S*: Stack)



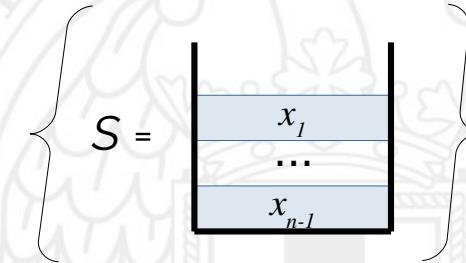
# Operaciones *push* y *pop*



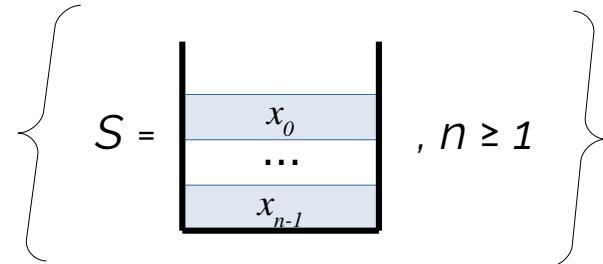
***push*( $S$ : Stack,  $x$ : elem)**



***pop*( $S$ : Stack)**

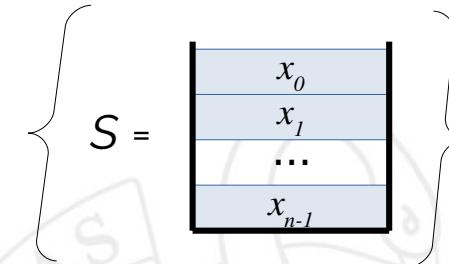


# Operaciones *top* y *empty*



***top***( $S$ : Stack)  $\rightarrow$  ( $x$ : elem)

$$\{ x = x_0 \}$$



***empty***( $S$ : Stack)  $\rightarrow$  ( $b$ : bool)

$$\{ b \Leftrightarrow n = 0 \}$$

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

# Implementando el TAD Pila

Manuel Montenegro Montes

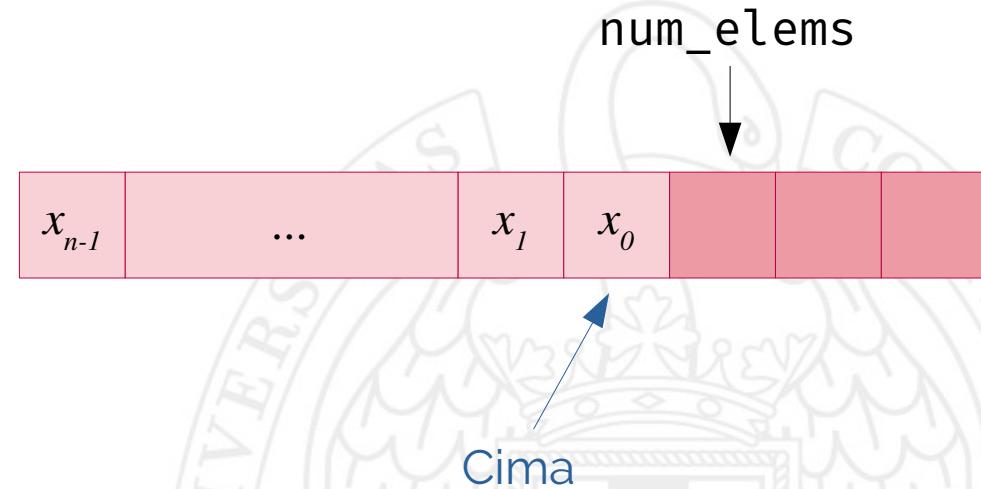
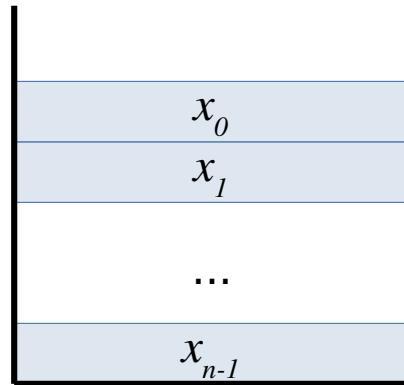
Departamento de Sistemas Informáticos y Computación  
Facultad de Informática – Universidad Complutense de Madrid

# Operaciones sobre pilas

- **Constructoras:**
  - Crear una pila vacía (***create\_empty***).
- **Mutadoras:**
  - Añadir elemento en la cima de la pila (***push***).
  - Eliminar elemento en la cima de la pila (***pop***).
- **Observadoras:**
  - Obtener el elemento en la cima de la pila (***top***).
  - Saber si una pila está vacía (***empty***).

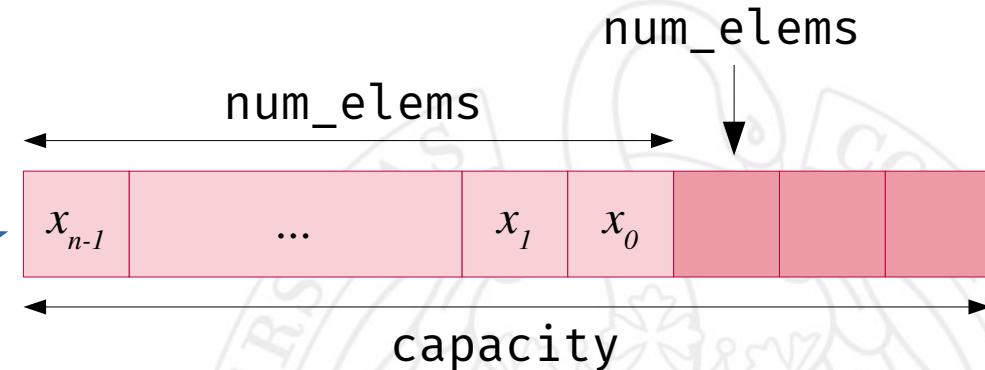
# Implementación mediante vectores

# Implementación mediante vectores



# Implementación mediante vectores

```
template<typename T>
class StackArray {
public:
    ...
private:
    int num_elems;
    int capacity;
    T *elems;
};
```



# Interfaz pública de StackArray

```
template<typename T>
class StackArray {
public:
    StackArray(int initial_capacity = DEFAULT_CAPACITY);
    StackArray(const StackArray &other);
    ~StackArray();

    StackArray & operator=(const StackArray<T> &other);

    void push(const T &elem);
    void pop();
    const T & top() const;
    T & top();
    bool empty() const;

private:
    ...
};
```



# Interfaz pública de StackArray

```
template<typename T>
class StackArray {
public:
    StackArray(int initial_capacity = DEFAULT_CAPACITY);
    StackArray(const StackArray &other);
    ~StackArray();

    StackArray & operator=(const StackArray<T> &other);

    void push(const T &elem);
    void pop();
    const T & top() const;
    T & top();
    bool empty() const;

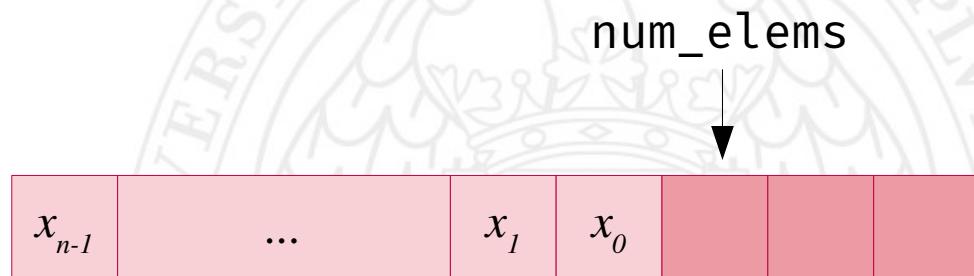
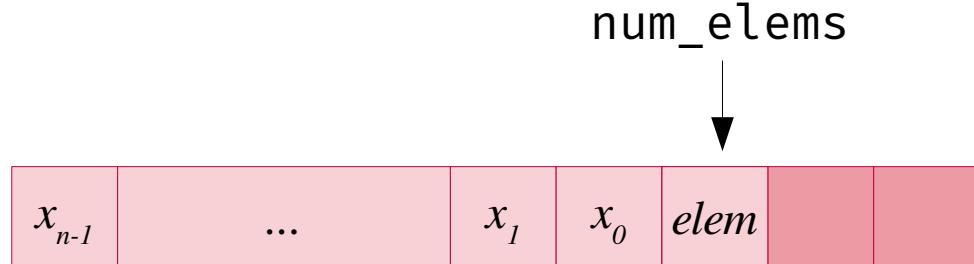
private:
    ...
};
```



# Métodos push() y pop()

```
void push(const T &elem) {  
    if (num_elems == capacity) {  
        resize_array(capacity * 2);  
    }  
    elems[num_elems] = elem;  
    num_elems++;  
}
```

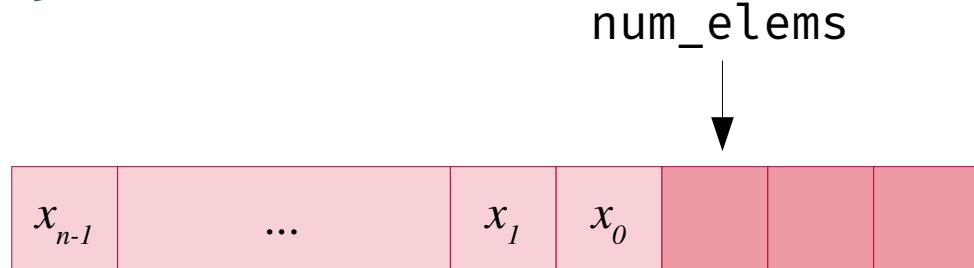
```
void pop() {  
    assert(num_elems > 0);  
    num_elems--;  
}
```



# Métodos `top()` y `empty()`

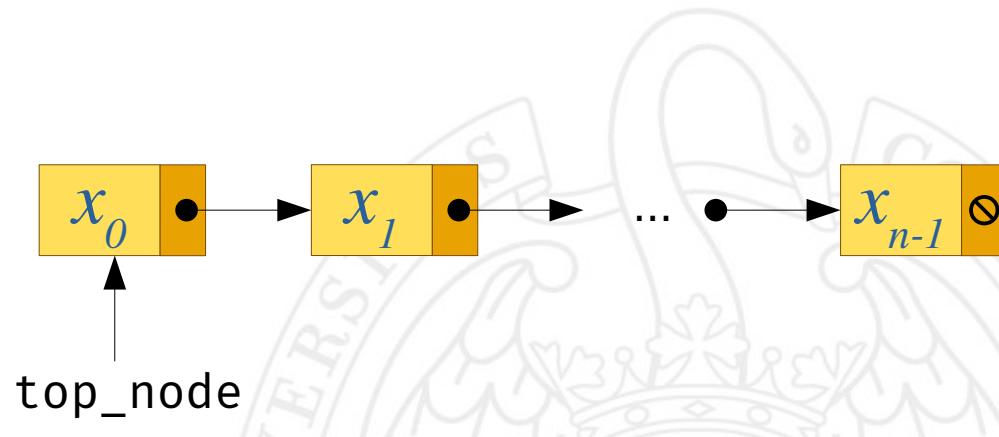
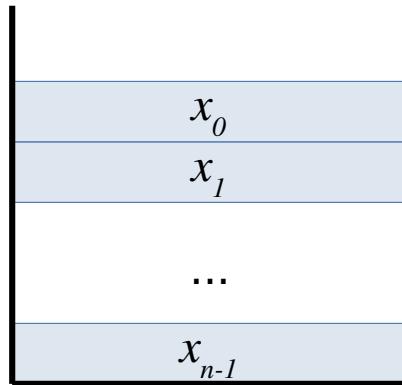
```
const T & top() const {  
    assert(num_elems > 0);  
    return elems[num_elems - 1];  
}
```

```
bool empty() const {  
    return num_elems == 0;  
}
```



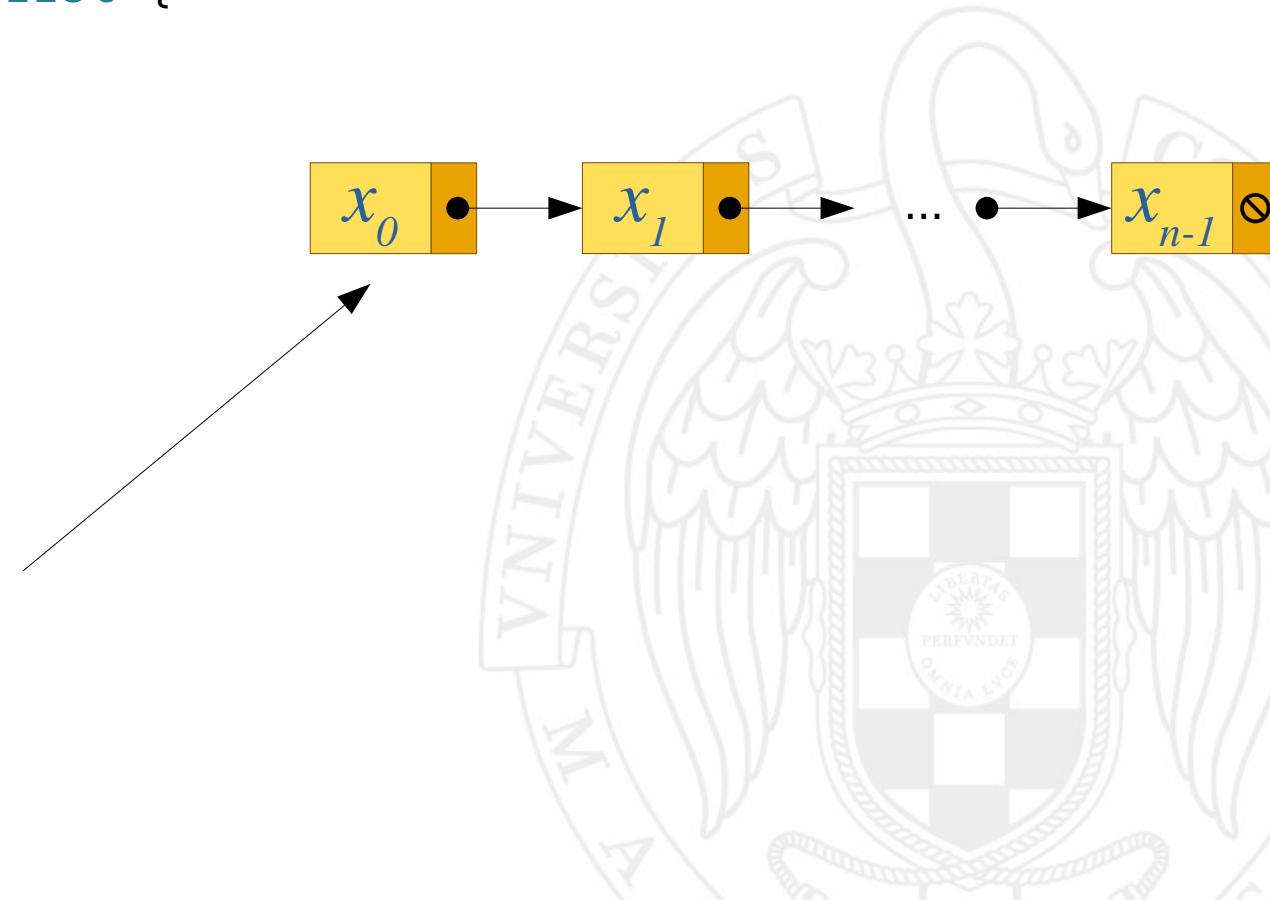
# Implementación mediante listas enlazadas simples

# Implementación mediante listas enlazadas



# Implementación mediante listas enlazadas

```
template<typename T>
class StackLinkedList {
public:
    ...
private:
    struct Node {
        T value;
        Node *next;
    };
    Node *top_node;
};
```



# Interfaz pública de StackLinkedList

```
template<typename T>
class StackLinkedList {
public:
    StackLinkedList();
    StackLinkedList(const StackLinkedList &other);
    ~StackLinkedList();

    StackLinkedList & operator=(const StackLinkedList<T> &other);

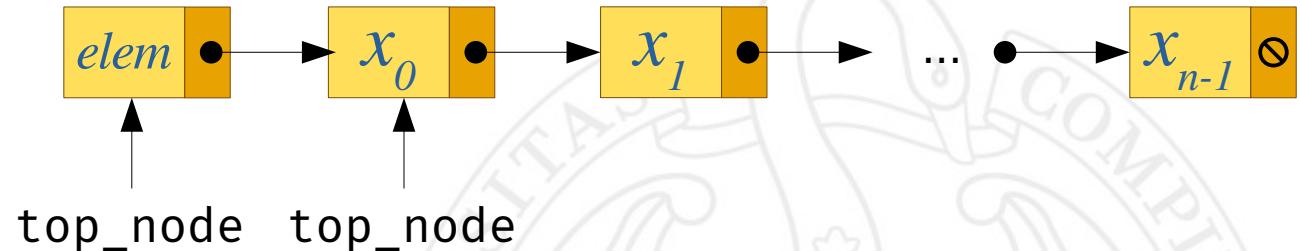
    void push(const T &elem);
    void pop();
    const T & top() const;
    T & top();
    bool empty() const;

private:
    ...
};
```

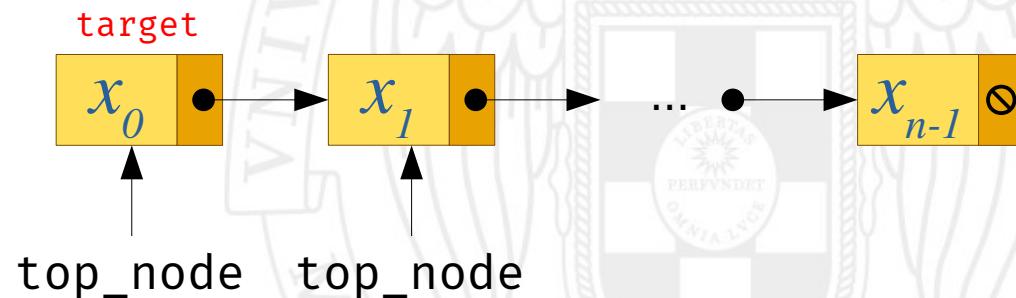


# Operaciones push() y pop()

```
void push(const T &elem) {  
    top_node = new Node{ elem, top_node };  
}
```



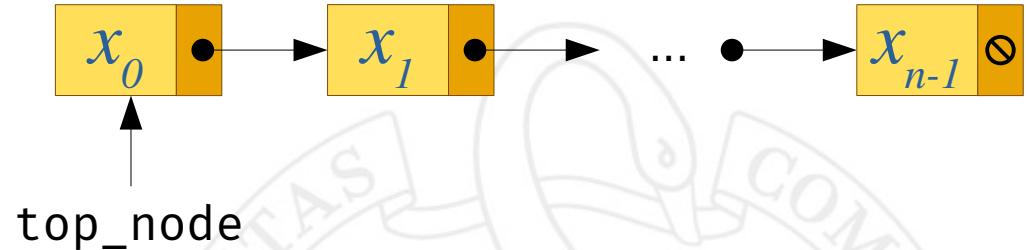
```
void pop() {  
    assert (top_node != nullptr);  
    Node *target = top_node;  
    top_node = top_node->next;  
    delete target;  
}
```



# Operaciones top() y empty()

```
const T & top() const {
    assert (top_node != nullptr);
    return top_node->value;
}

bool empty() const {
    return (top_node == nullptr);
}
```



# Coste de las operaciones

Operación	Vectores	Listas enlazadas
push	$O(n) / O(1)$	$O(1)$
pop	$O(1)$	$O(1)$
top	$O(1)$	$O(1)$
empty	$O(1)$	$O(1)$

$n$  = número de elementos en la pila

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

# Aplicaciones de pilas

Manuel Montenegro Montes

Departamento de Sistemas Informáticos y Computación  
Facultad de Informática – Universidad Complutense de Madrid

# Expresiones en forma postfija

# Expresiones en forma postfija

- Expresión en forma infija:

$$(3 * (5 + 2)) - 6$$

- La misma expresión en forma postfija:

$$(3 (5 2 +) *) 6 -$$

$$3 5 2 + * 6 -$$

- **Objetivo:** evaluar expresión en forma postfija.

- Por ejemplo,  $3 5 2 + * 6 -$  se evalúa al valor 15.

# Expresiones en forma postfija

- Las expresiones en forma postfija no contienen ambigüedades, no necesitan paréntesis o reglas de precedencia entre operadores, al contrario que las expresiones en forma infija.

2 + 4 \* 5

# Otros ejemplos

- 2 3 + 6 + 1 +
- 3 1 - 6 5 \* +



# Evaluando las expresiones mediante una pila

- Comenzamos con una pila vacía.
- Recorremos de izquierda a derecha los caracteres de la expresión.
- Si el carácter actual es un número:
  - Insertar el número en la pila.
- Si el carácter actual es un operador:
  - Desapilar los dos operandos y realizar la operación con ellos.
  - Apilar el resultado.
- Al finalizar el recorrido, el elemento que quede en la pila es el resultado de evaluar la expresión.

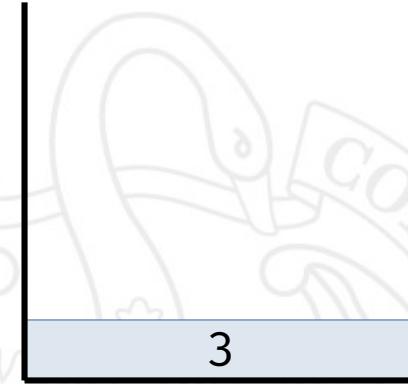
# Ejemplo de ejecución

3 5 2 + \* 6 -  
↑



# Ejemplo de ejecución

3 5 2 + \* 6 -  
↑



# Ejemplo de ejecución

3 5 2 + \* 6 -  
↑

5
3

# Ejemplo de ejecución

3 5 2 + \* 6 -



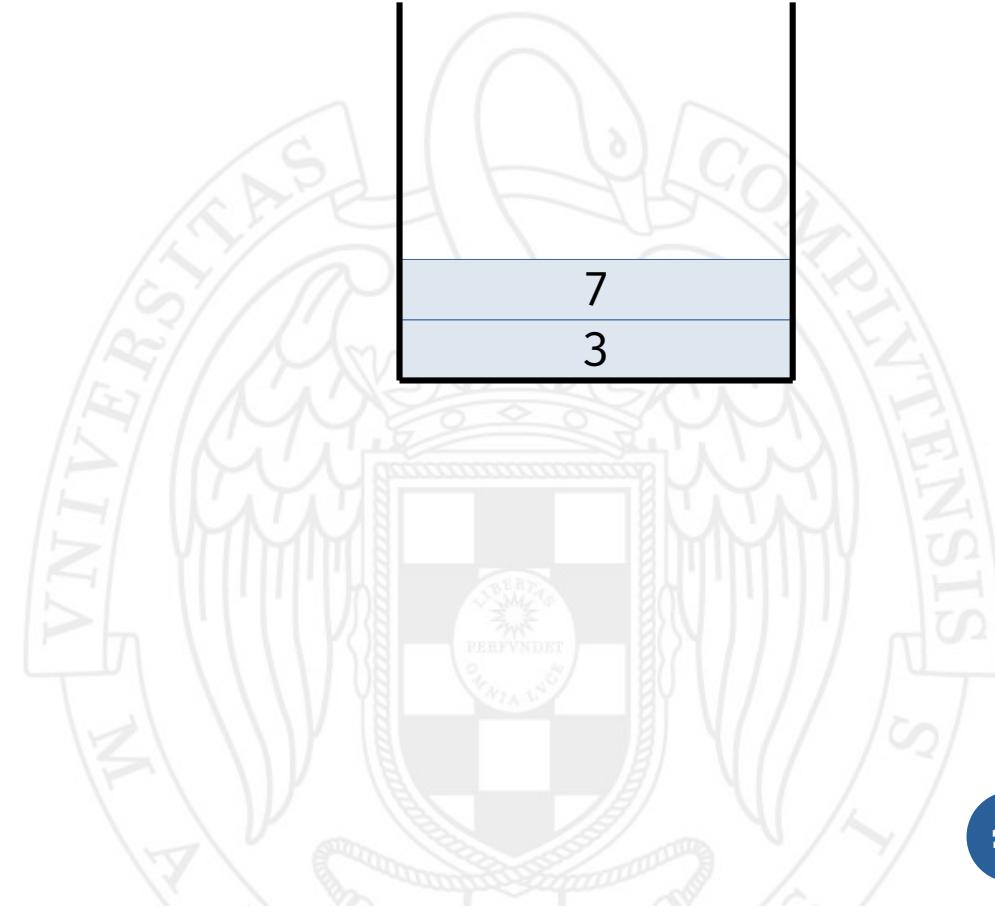
2
5
3

# Ejemplo de ejecución

3 5 2 + \* 6 -



7
3



# Ejemplo de ejecución

3 5 2 + \* 6 -  
      ↑

21

# Ejemplo de ejecución

3 5 2 + \* 6 -

6
21

# Ejemplo de ejecución

3 5 2 + \* 6 -

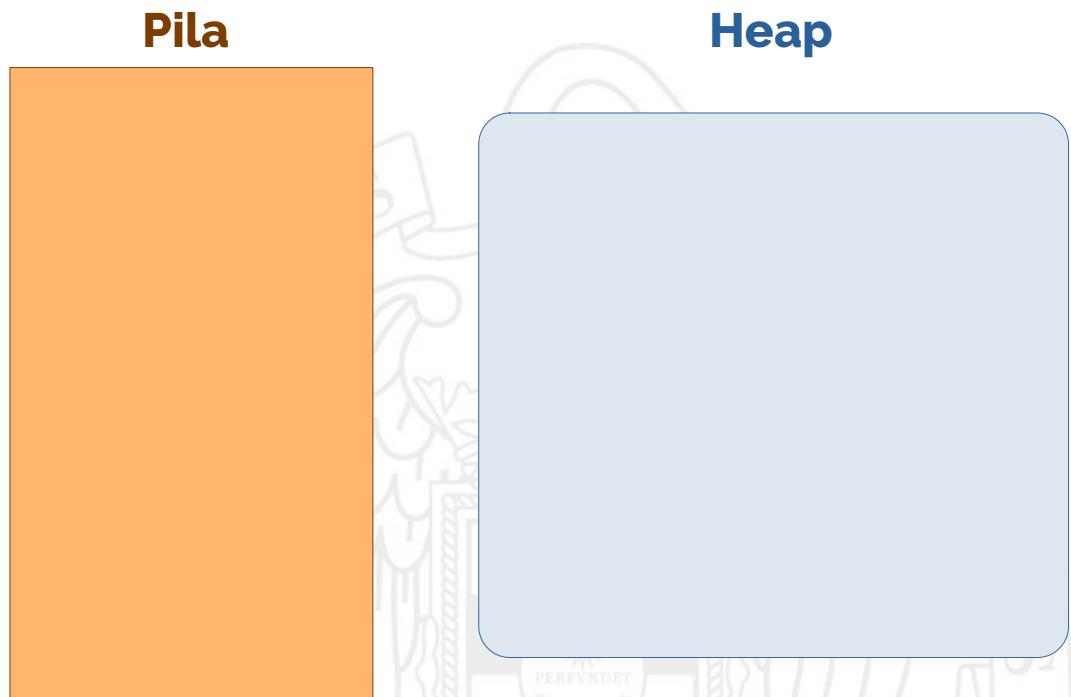


15

# Pila de llamadas a funciones

# Ejemplo de ejecución

- La región de memoria en la que se almacenan las variables locales y parámetros de un programa funciona como una pila.



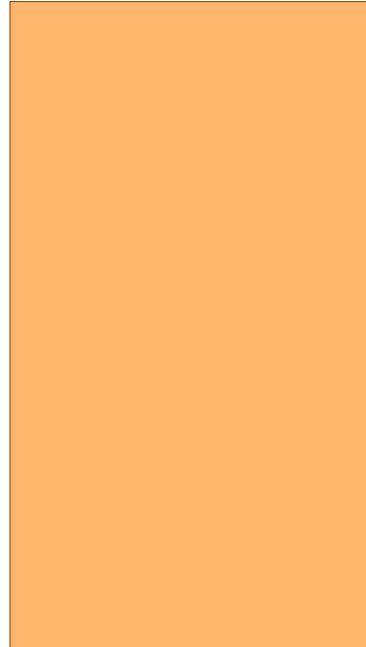
# Ejemplo de ejecución

```
void f(int x) {  
    int y = 40;  
}
```

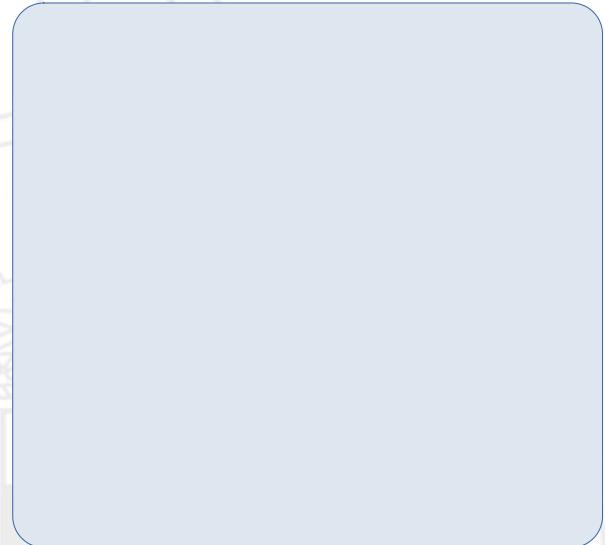
```
void g() {  
    int x = 10;  
    int z = 20;  
    f(30);  
    x = 50;  
}
```

```
int main() {  
    int y = 4;  
    g();  
}
```

Pila



Heap



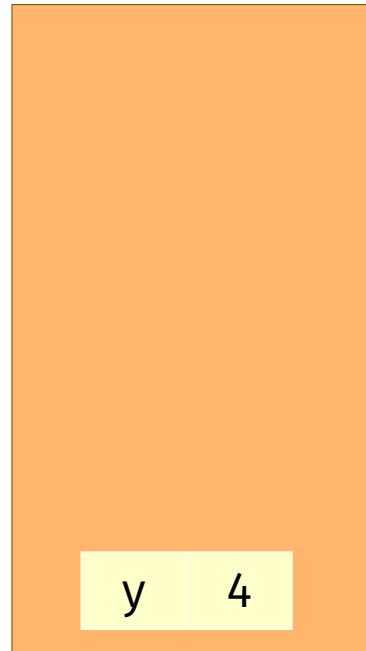
# Ejemplo de ejecución

```
void f(int x) {  
    int y = 40;  
}
```

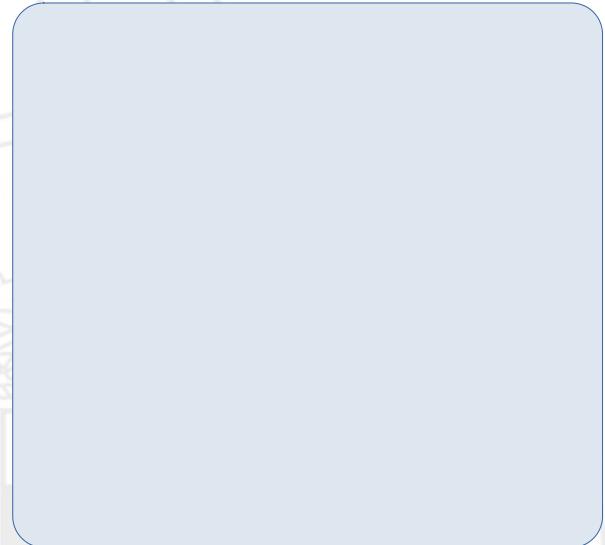
```
void g() {  
    int x = 10;  
    int z = 20;  
    f(30);  
    x = 50;  
}
```

```
int main() {  
    int y = 4;  
    g();  
}
```

Pila



Heap



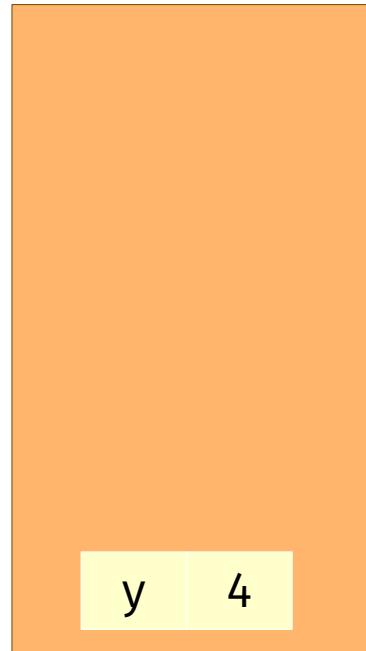
# Ejemplo de ejecución

```
void f(int x) {  
    int y = 40;  
}
```

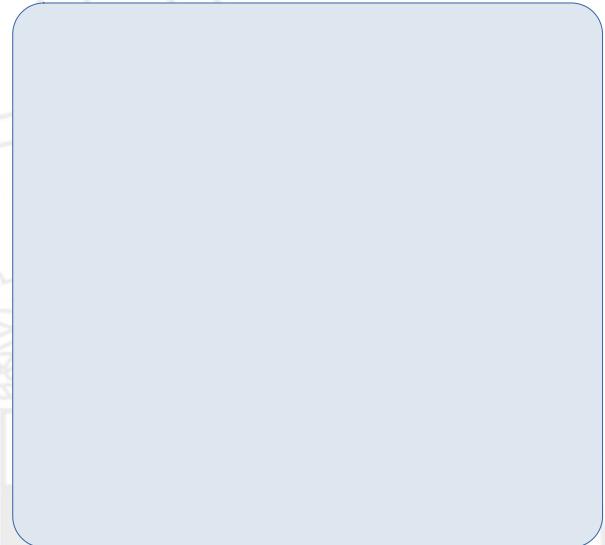
```
void g() {  
    int x = 10;  
    int z = 20;  
    f(30);  
    x = 50;  
}
```

```
int main() {  
    int y = 4;  
    g();  
}
```

Pila



Heap

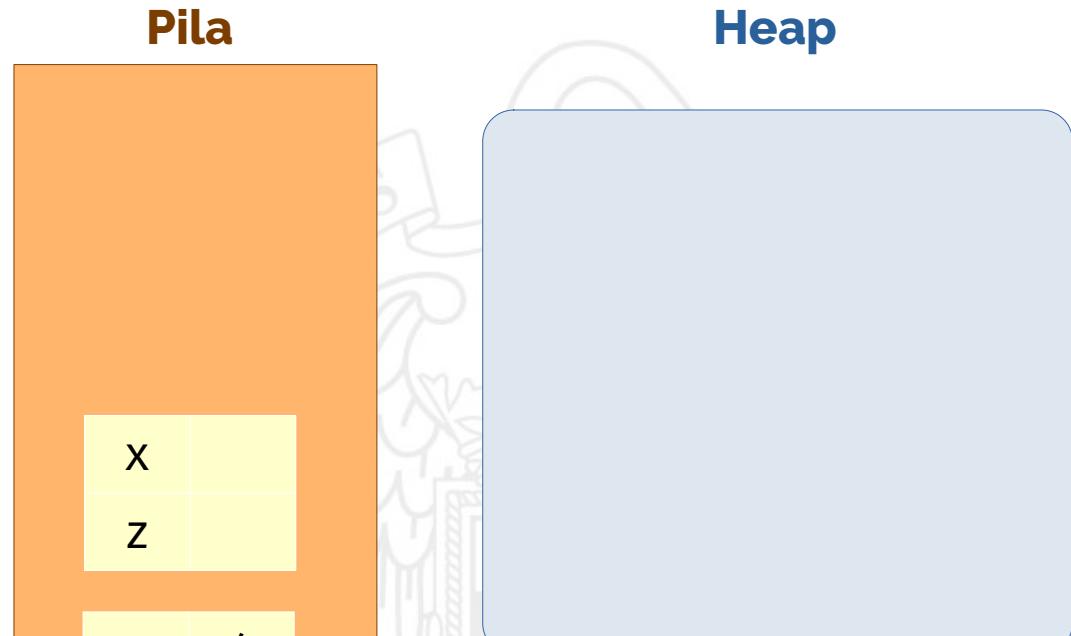


# Ejemplo de ejecución

```
void f(int x) {  
    int y = 40;  
}
```

```
void g() {  
    int x = 10;  
    int z = 20;  
    f(30);  
    x = 50;  
}
```

```
int main() {  
    int y = 4;  
    g();  
}
```



# Ejemplo de ejecución

```
void f(int x) {  
    int y = 40;  
}
```

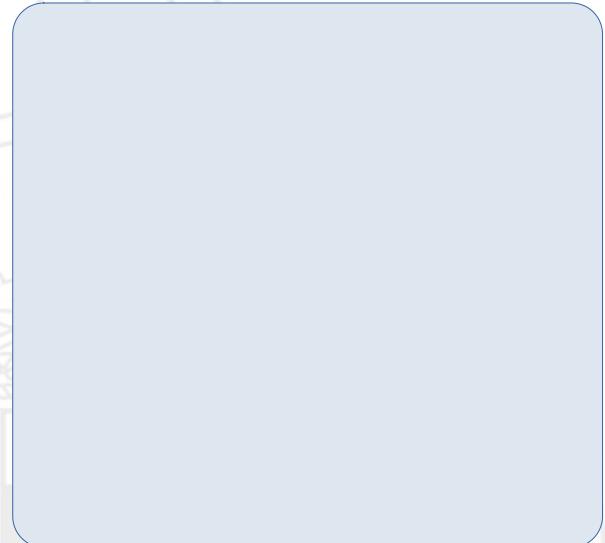
```
void g() {  
    int x = 10;  
    int z = 20;  
    f(30);  
    x = 50;  
}
```

```
int main() {  
    int y = 4;  
    g();  
}
```

Pila

x	10
z	
y	4

Heap



# Ejemplo de ejecución

```
void f(int x) {  
    int y = 40;  
}
```

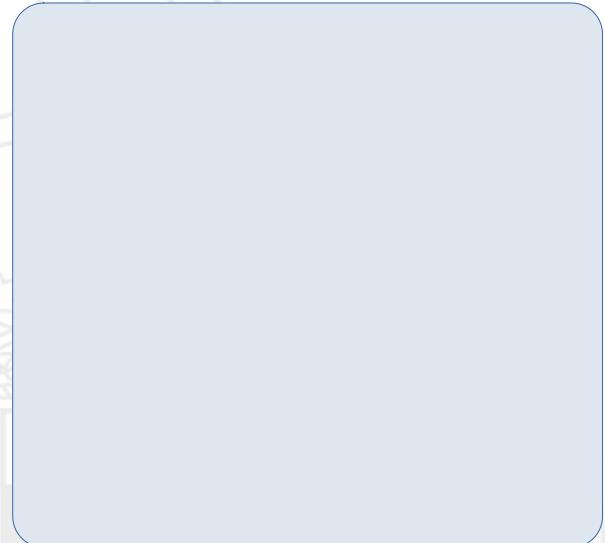
```
void g() {  
    int x = 10;  
    int z = 20;  
    f(30);  
    x = 50;  
}
```

```
int main() {  
    int y = 4;  
    g();  
}
```

Pila

x	10
z	20
y	4

Heap



# Ejemplo de ejecución

```
void f(int x) {  
    int y = 40;  
}
```

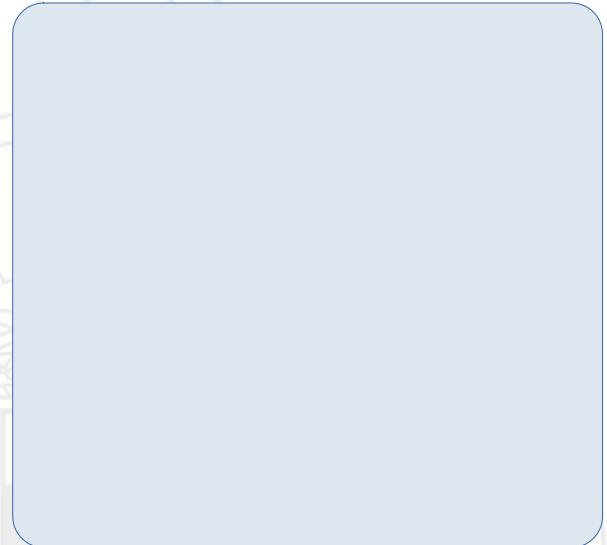
```
void g() {  
    int x = 10;  
    int z = 20;  
    f(30);  
    x = 50;  
}
```

```
int main() {  
    int y = 4;  
    g();  
}
```

Pila

x	10
z	20
y	4

Heap



# Ejemplo de ejecución

```
void f(int x) {  
    int y = 40;  
}
```

```
void g() {  
    int x = 10;  
    int z = 20;  
    f(30);  
    x = 50;  
}
```

```
int main() {  
    int y = 4;  
    g();  
}
```

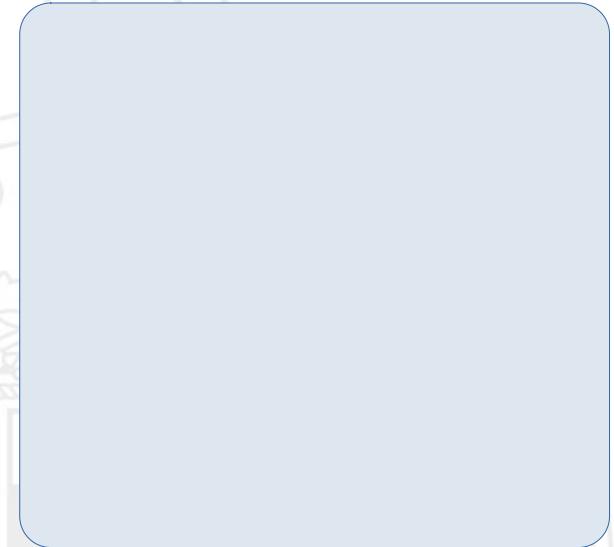
Pila

x	30
y	

x	10
z	20

y	4
---	---

Heap



# Ejemplo de ejecución

```
void f(int x) {  
    int y = 40;  
}
```

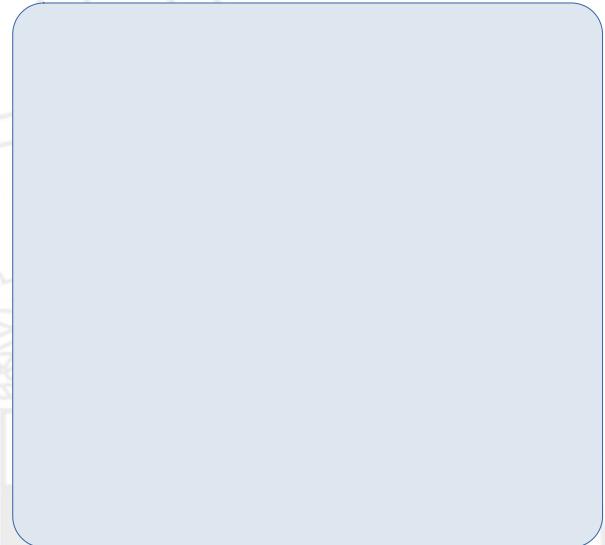
```
void g() {  
    int x = 10;  
    int z = 20;  
    f(30);  
    x = 50;  
}
```

```
int main() {  
    int y = 4;  
    g();  
}
```

Pila

x	30
y	40
x	10
z	20
y	4

Heap



# Ejemplo de ejecución

```
void f(int x) {  
    int y = 40;  
}
```

```
void g() {  
    int x = 10;  
    int z = 20;  
    f(30);  
    x = 50;  
}
```

```
int main() {  
    int y = 4;  
    g();  
}
```

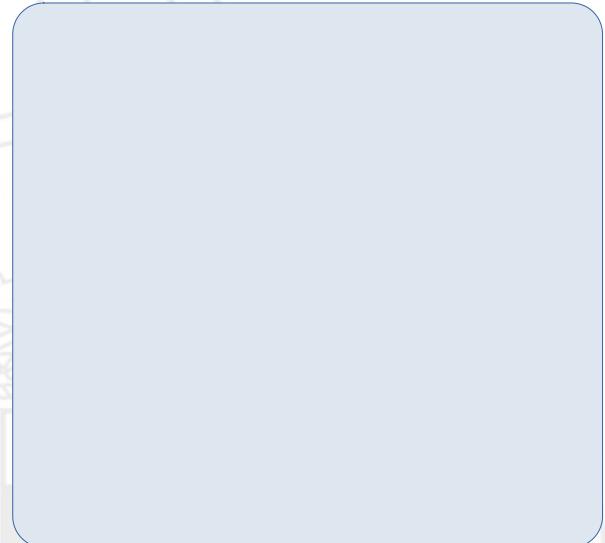
Pila

x	30
y	40

x	10
z	20

y	4
---	---

Heap



# Ejemplo de ejecución

```
void f(int x) {  
    int y = 40;  
}
```

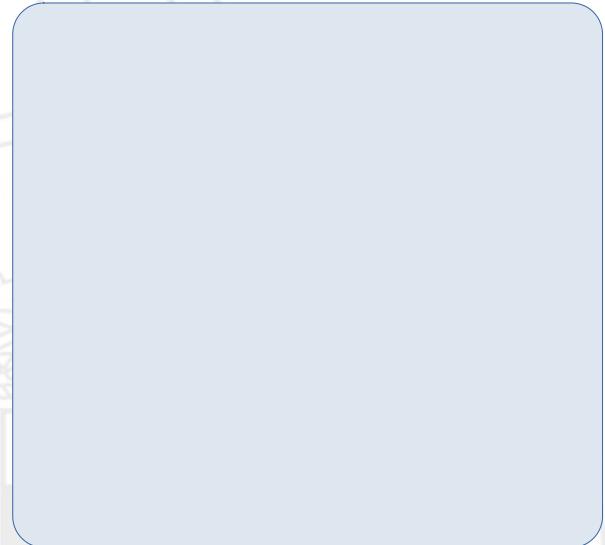
```
void g() {  
    int x = 10;  
    int z = 20;  
    f(30);  
    x = 50;  
}
```

```
int main() {  
    int y = 4;  
    g();  
}
```

Pila

x	50
z	20
y	4

Heap



# Ejemplo de ejecución

```
void f(int x) {  
    int y = 40;  
}
```

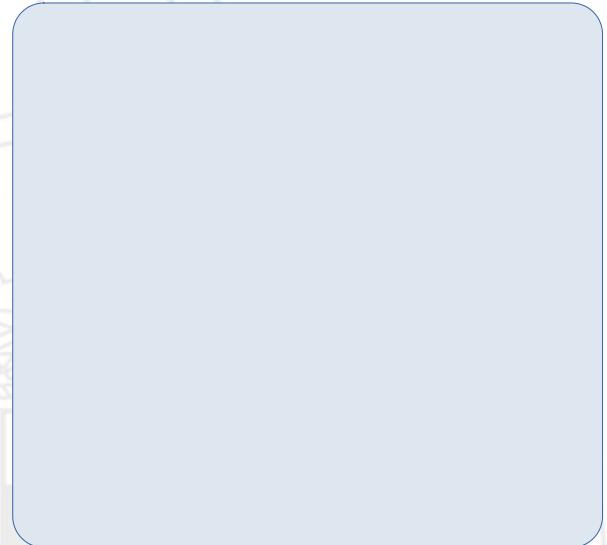
```
void g() {  
    int x = 10;  
    int z = 20;  
    f(30);  
    x = 50;  
}
```

```
int main() {  
    int y = 4;  
    g();  
}
```

Pila

x	50
z	20
y	4

Heap



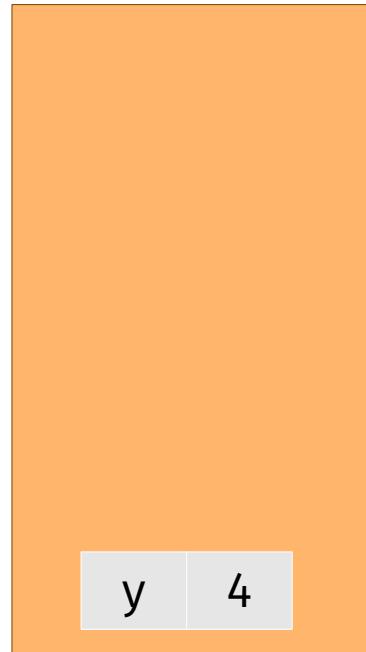
# Ejemplo de ejecución

```
void f(int x) {  
    int y = 40;  
}
```

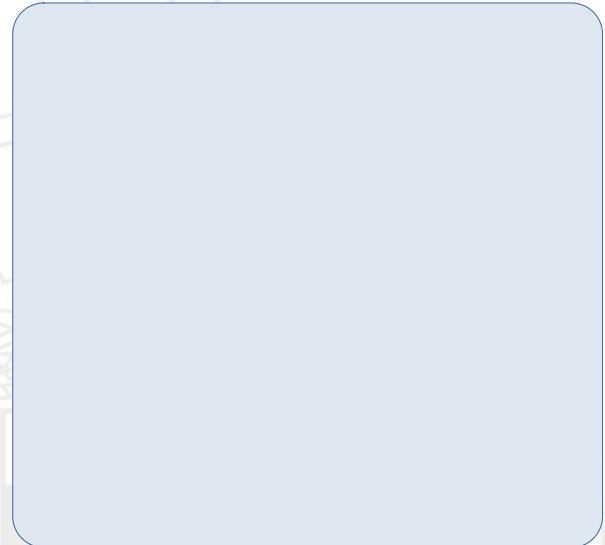
```
void g() {  
    int x = 10;  
    int z = 20;  
    f(30);  
    x = 50;  
}
```

```
int main() {  
    int y = 4;  
    g();  
}
```

Pila



Heap



ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

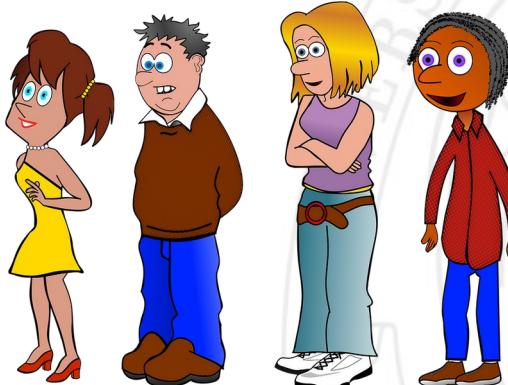
# EL TAD Cola

Manuel Montenegro Montes

Departamento de Sistemas Informáticos y Computación  
Facultad de Informática – Universidad Complutense de Madrid

# ¿Qué es una cola?

- Es una colección de elementos que permite:
  - Insertar elementos.
  - Borrar elementos en el orden en el que han sido insertados.
  - Obtener el primer elemento insertado no borrado.



# ¿Qué es una cola?

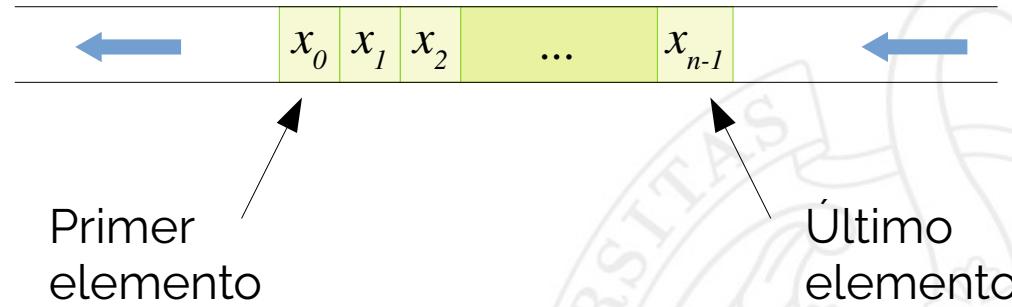
- Las colas siguen una disciplina de acceso **FIFO**

**First In, First Out**



# Modelo de colas

- Conceptualmente representamos las colas de esta forma:



# Operaciones sobre colas

- **Constructoras:**
  - Crear una cola vacía (***create\_empty***).
- **Mutadoras:**
  - Añadir un elemento al final de la cola (***enqueue***)
  - Eliminar el primer elemento de la cola (***dequeue***).
- **Observadoras:**
  - Obtener el primer elemento de la cola (***front***)
  - Saber si una cola está vacía (***empty***).

# Operaciones sobre colas

- **Constructoras:**
  - Crear una cola vacía (*create\_empty*).
- **Mutadoras:**
  - Añadir un elemento al final de la cola (*enqueue push*)
  - Eliminar el primer elemento de la cola (*dequeue pop*).
- **Observadoras:**
  - Obtener el primer elemento de la cola (*front*)
  - Saber si una cola está vacía (*empty*).

# Operación *create\_empty*

{ true }

***create\_empty()*** → (Q: Queue)

{ Q = \_\_\_\_\_ }



# Operaciones *push* (*enqueue*) y *pop* (*dequeue*)

$$\{ Q = \underline{\underline{x_0 \ x_1 \ \dots \ x_{n-1}}} \}$$

***push*(Q: Queue, x: elem)**

$$\{ Q = \underline{\underline{x_0 \ x_1 \ \dots \ x_{n-1} \ x}} \}$$

$$\{ Q = \underline{\underline{x_0 \ x_1 \ \dots \ x_{n-1}}} \quad n \geq 1 \}$$

***pop*(Q: Queue)**

$$\{ Q = \underline{\underline{\phantom{x_0 \ x_1 \ \dots \ x_{n-1}}}} \}$$

# Operaciones *front* y *size*

$$\left\{ Q = \overbrace{\quad | x_0 | x_1 | \dots | x_{n-1} | \quad}^{n \geq 1} \right\}$$

**front**(Q: Queue)  $\rightarrow$  (x: elem)

$$\{ x = x_0 \}$$

$$\left\{ Q = \overbrace{\quad | x_0 | x_1 | \dots | x_{n-1} | \quad}^{n > 0} \right\}$$

**empty**(Q: Queue)  $\rightarrow$  (x: elem)

$$\{ b \Leftrightarrow n = 0 \}$$

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

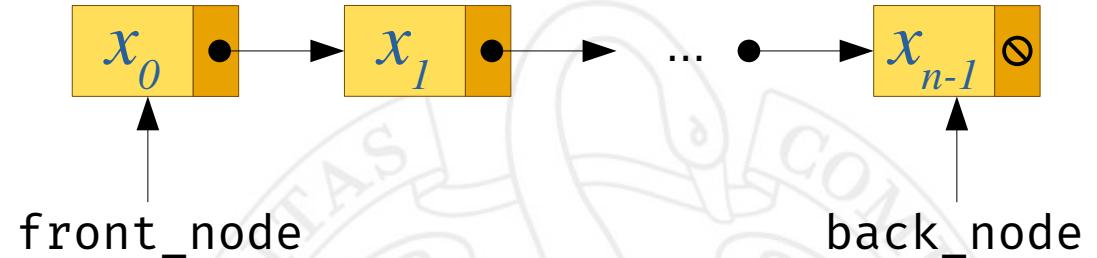
# Implementando el TAD Cola

Manuel Montenegro Montes

Departamento de Sistemas Informáticos y Computación  
Facultad de Informática – Universidad Complutense de Madrid

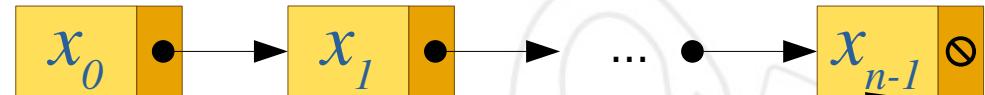
# Implementación mediante listas enlazadas

# Implementación mediante listas enlazadas



# Clase QueueLinkedList

```
template<typename T>
class QueueLinkedList {  
  
...  
  
private:  
    struct Node {  
        T value;  
        Node *next;  
    };  
  
    Node *front_node;  
    Node *back_node;  
};
```



- Si la cola está vacía:

$\text{front\_node} = \text{back\_node} = \text{nullptr}$

# Interfaz pública de QueueLinkedList

```
template<typename T>
class QueueLinkedList {

    QueueLinkedList();
    QueueLinkedList(const QueueLinkedList &other);
    ~QueueLinkedList();

    QueueLinkedList & operator=(const QueueLinkedList &other);

    void push(const T &elem);
    void pop();
    T & front();
    const T & front() const;
    bool empty() const;

    ...
};

}
```



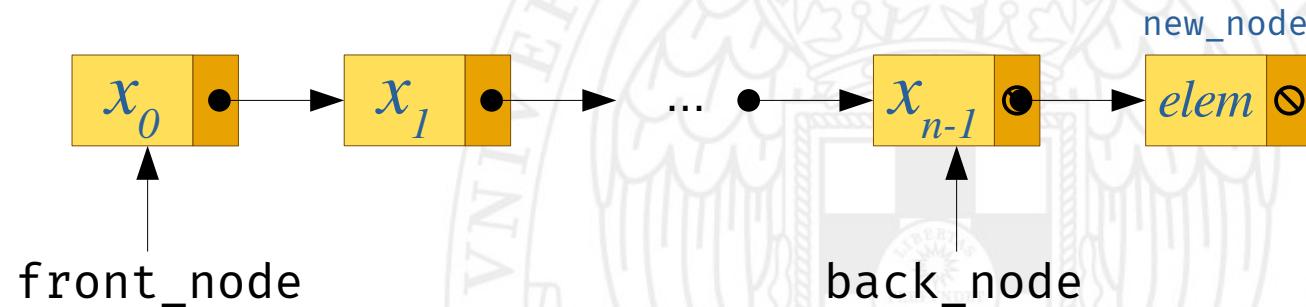
# Interfaz pública de QueueLinkedList

```
template<typename T>
class QueueLinkedList {  
  
    QueueLinkedList();
    QueueLinkedList(const QueueLinkedList &other);
    ~QueueLinkedList();  
  
    QueueLinkedList & operator=(const QueueLinkedList &other);  
  
    void push(const T &elem);
    void pop();
    T & front();
    const T & front() const;
    bool empty() const;  
  
    ...  
};
```



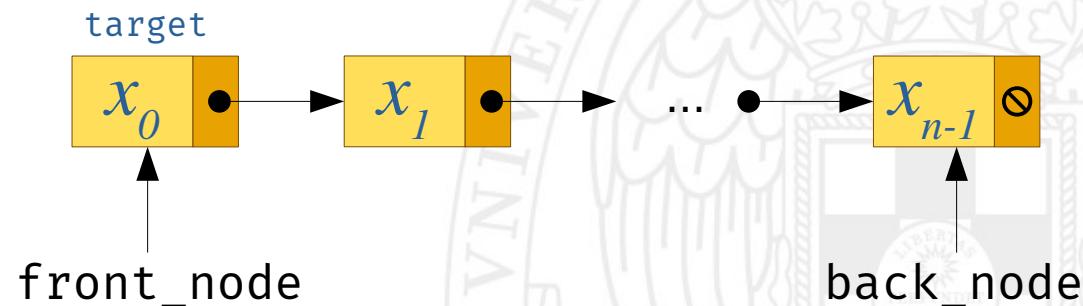
# Método push()

```
void push(const T &elem) {  
    Node *new_node = new Node { elem, nullptr };  
    if (back_node == nullptr) {  
        back_node = new_node;  
        front_node = new_node;  
    } else {  
        back_node->next = new_node;  
        back_node = new_node;  
    }  
}
```



# Método pop()

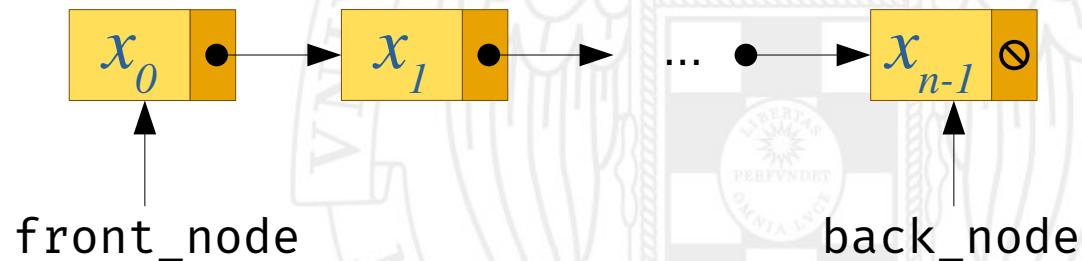
```
void pop() {
    assert (front_node != nullptr);
    Node *target = front_node;
    front_node = front_node->next;
    if (back_node == target) {
        back_node = nullptr;
    }
    delete target;
}
```



# Métodos front() y empty()

```
const T & front() const {
    assert (front_node != nullptr);
    return front_node->value;
}

bool empty() const {
    return (front_node == nullptr);
}
```



# Implementación mediante vectores circulares

# Implementación mediante vectores circulares



# Idea: vectores circulares

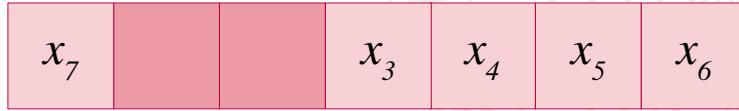
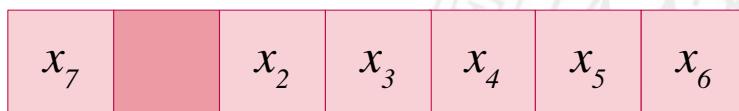
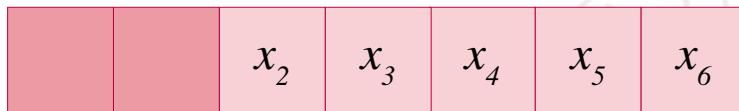
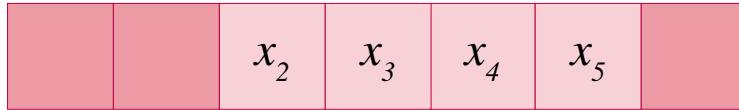
$x_0$	$x_1$	$x_2$	$x_3$	$x_4$		
-------	-------	-------	-------	-------	--	--

$x_0$	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	
-------	-------	-------	-------	-------	-------	--

	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	
--	-------	-------	-------	-------	-------	--

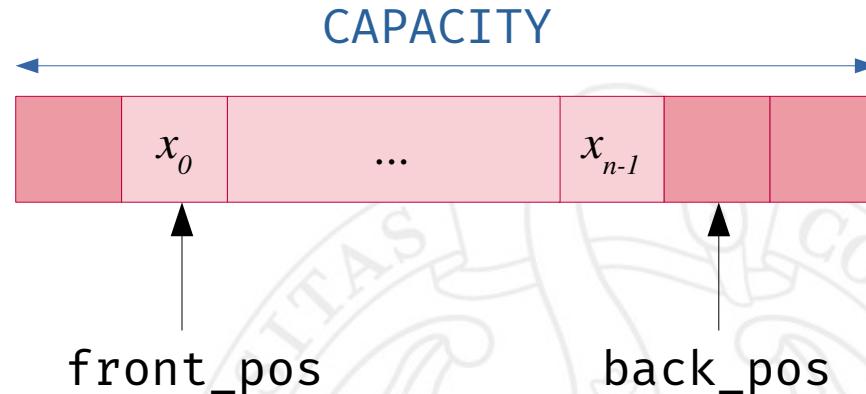
		$x_2$	$x_3$	$x_4$	$x_5$	
--	--	-------	-------	-------	-------	--

# Idea: vectores circulares



# Clase QueueArray

```
class QueueArray {  
public:  
    ...  
  
private:  
    T *elems;  
    int front_pos, back_pos;  
};
```



- Si la cola está vacía:  
 $\text{front\_pos} == \text{back\_pos}$

# Constructor

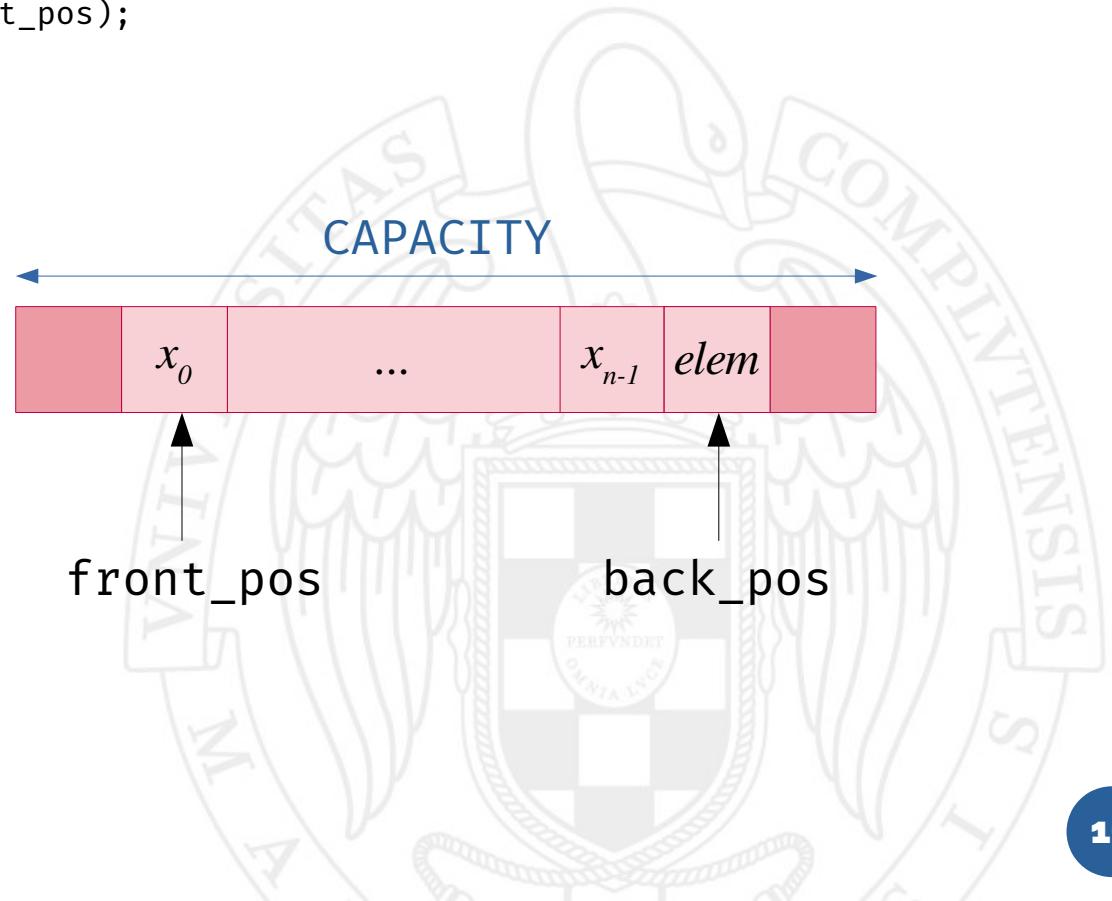
```
QueueArray() {  
    elems = new T[CAPACITY];  
    front_pos = 0;  
    back_pos = 0;  
}
```



front\_pos  
back\_pos

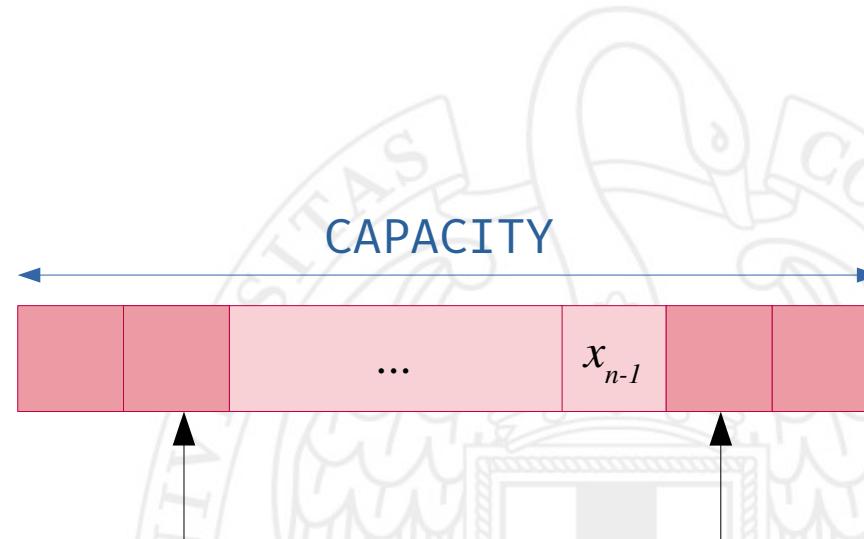
# Método push()

```
void push(const T &elem) {  
    // Cabe el elemento en la cola?  
    assert ((back_pos + 1) % CAPACITY != front_pos);  
    elems[back_pos] = elem;  
    back_pos = (back_pos + 1) % CAPACITY;  
}
```



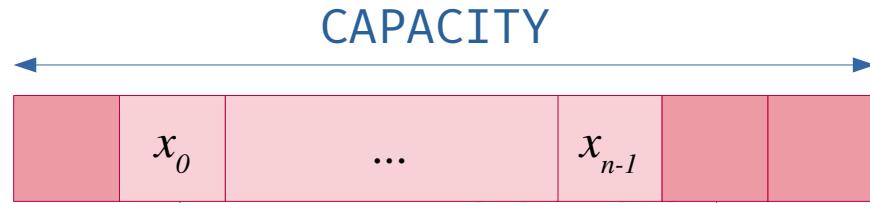
# Método pop()

```
void pop() {  
    assert (front_pos != back_pos);  
    front_pos = (front_pos + 1) % CAPACITY;  
}
```

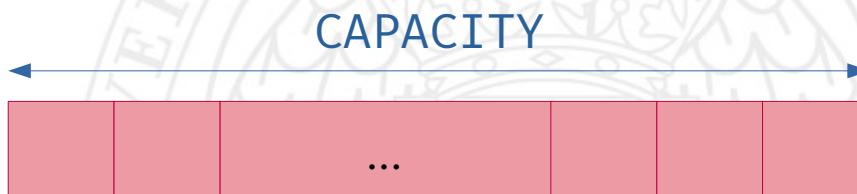


# Métodos front() y empty()

```
const T& front() const {
    assert(front_pos != back_pos);
    return elems[front_pos];
}
```



```
bool empty() const {
    return front_pos == back_pos;
}
```



front\_pos  
back\_pos

# Coste de las operaciones

Operación	Listas enlazadas	Vectores circulares
push	$O(1)$	$O(1)$
pop	$O(1)$	$O(1)$
front	$O(1)$	$O(1)$
empty	$O(1)$	$O(1)$

$n$  = número de elementos en la cola

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

# EL TAD de colas dobles (*deque*)

Manuel Montenegro Montes

Departamento de Sistemas Informáticos y Computación  
Facultad de Informática – Universidad Complutense de Madrid

# ¿Qué es una cola doble?

- Es una estructura de datos que combina la funcionalidad de la pila con la funcionalidad de la cola. En particular:
  - Insertar/eliminar/acceder a elementos al final de la cola doble.
  - Insertar/eliminar/acceder a elementos al principio de la cola doble.



# Operaciones de una cola doble

- **Constructoras:**
  - Crear una cola doble vacía (*create\_empty*).
- **Mutadoras:**
  - Añadir un elemento al principio (*push\_front*).
  - Añadir un elemento al final (*push\_back*).
  - Eliminar un elemento del principio (*pop\_front*).
  - Eliminar un elemento del final (*pop\_back*).
- **Observadoras:**
  - Comprobar si una cola doble es vacía (*empty*).
  - Acceder al primer elemento de la cola doble (*front*).
  - Acceder al último elemento de la cola doble (*back*).



# TAD Colas dobles vs TAD Lista

## Colas dobles

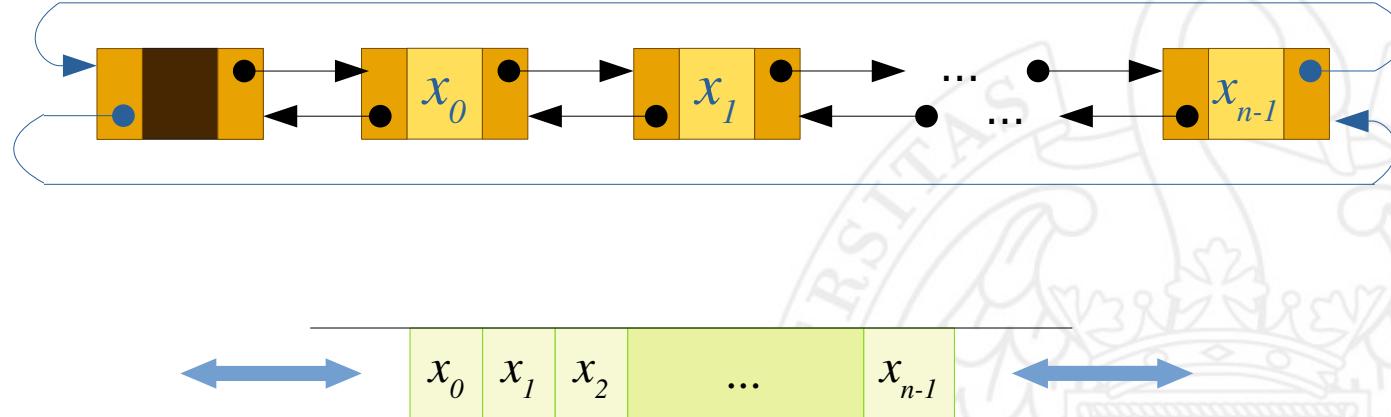
- Constructoras:
  - *create\_empty*
- Mutadoras:
  - *push\_front*
  - *push\_back*
  - *pop\_front*
  - *pop\_back*
- Observadoras:
  - *empty*
  - *front*
  - *back*

## Listas

- Constructoras:
  - *create\_empty*
- Mutadoras:
  - *push\_front*
  - *push\_back*
  - *pop\_front*
  - *pop\_back*
- Observadoras:
  - *empty*
  - *front*
  - *back*
  - *size*
  - *at*

# Implementación de colas dobles

- Puede utilizarse una lista circular con nodo fantasma:



# Coste de las operaciones

Operación	Coste en tiempo
create_empty	$O(1)$
push_back	$O(1)$
push_front	$O(1)$
pop_back	$O(1)$
pop_front	$O(1)$
front	$O(1)$
back	$O(1)$
empty	$O(1)$

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

# Introducción a los iteradores

Manuel Montenegro Montes

Departamento de Sistemas Informáticos y Computación  
Facultad de Informática – Universidad Complutense de Madrid

# Motivación

# Problema

- Tenemos una lista de enteros, y queremos calcular la suma de todos los elementos de la lista.

```
int suma_elems(const ListLinkedDouble<int> &l) {  
    int suma = 0;  
    for (int i = 0; i < l.size(); i++) {  
        suma += l[i];  
    }  
    return suma;  
}
```

¿Qué coste tiene esta función?

# Possible solución 1

- Utilizar otra implementación de listas, de modo que la operación `at()` tenga coste constante.

```
int suma_elems(const ListArray<int> &l) {  
    int suma = 0;  
    for (int i = 0; i < l.size(); i++) {  
        suma += l[i];  
    }  
    return suma;  
}
```

¿Y si necesito una lista enlazada?

# Possible solución 2

- Hacer copia de la lista de entrada, e ir eliminando los elementos de la copia uno a uno.

```
int suma_elems(const ListLinkedDouble<int> &l) {  
    ListLinkedDouble<int> copia = l;  
    int suma = 0;  
    while (!copia.empty()) {  
        suma += copia.front();  
        copia.pop_front();  
    }  
    return suma;  
}
```

¿Cuál es el coste en espacio de esta solución?

# Possible solución 3

- Integrar la operación dentro de la clase `ListLinkedDouble`.

```
template <typename T>
class ListLinkedDouble {
    ...
    int suma_elems() const {
        int suma = 0;
        Node *current = head->next;
        while (current != head) {
            suma += current->value;
            current = current->next;
        }
        return suma;
    }
};
```

¿Y si la lista no es de enteros?

¿Tengo que prever de antemano todas las cosas que puedo hacer en el recorrido de una lista?

# La solución que presentamos

- Proporcionar una abstracción al programador/a para que pueda navegar por los elementos de una lista, de modo independiente de la implementación.
- La navegación se realiza de manera secuencial.
- Esta abstracción recibe el nombre de **iterador**.



# Iteradores

# Iteradores

- Un iterador es un cursor que se mueve por los elementos de la lista de manera secuencial.
- Es posible realizar operaciones de acceso y modificación en la posición actual de un iterador.

```
[ 1, 4, 15, 7, 10, 23 ]
```

# Iteradores

- En el momento de su creación, un iterador está ligado a una lista.
- Representamos los iteradores de la siguiente forma:

$$[ \ x_0, |x_1|, x_2, \dots, x_{n-1} ]$$

iterador

Elemento apuntado por  
el iterador.

# Operaciones sobre un iterador

- Obtener el elemento apuntado por el iterador (**elem**)
- Avanzar el iterador a la siguiente posición de la lista (**advance**)
- Igualdad entre dos iteradores (==)
  - Dos iteradores son iguales si recorren la misma lista y apuntan a la misma posición dentro de esta.

$$\{ [x_0, \dots, \underset{\text{it}}{|} x_i, \dots, x_{n-1}], 0 \leq i < n \}$$

**elem**(*it*: Iterator) → (*x*: Elem)

$$\{ x = x_i \}$$
$$\{ [x_0, \dots, \underset{\text{it}}{|} x_i, \dots, x_{n-1}], 0 \leq i < n \}$$

**advance**(*it*: Iterator)

$$\{ [x_0, \dots, x_i, \underset{\text{it}}{|} x_{i+1}, \dots, x_{n-1}] \}$$

# Creación de iteradores

- Añadimos dos operaciones al TAD Lista:
  - Obtener un iterador al principio de la lista (**begin**)
  - Obtener un iterador al final de la lista (**end**)

$\{ l = [x_0, \dots, x_i, \dots, x_{n-1}] \}$

**begin**(*l: List*) → (*it: Iterator*)

$\{ [ \underset{\text{it}}{|} x_0, \dots, x_i, \dots, x_{n-1} ] \}$

$\{ l = [x_0, \dots, x_i, \dots, x_{n-1}] \}$

**end**(*l: List*) → (*it: Iterator*)

$\{ [ x_0, \dots, x_i, \dots, \underset{\text{it}}{|} x_{n-1} ] \}$

# Creación de iteradores

- Añadimos dos operaciones al TAD Lista:
  - Obtener un iterador al principio de la lista (**begin**)
  - Obtener un iterador al final de la lista (**end**)



Las operaciones **elem()** y **advance()** no están definidas para el iterador devuelto por **end()**

# Creación de iteradores

- Añadimos dos operaciones al TAD Lista:
  - Obtener un iterador al principio de la lista (**begin**)
  - Obtener un iterador al final de la lista (**end**)

$$\{ [x_0, \dots, |x_i, \dots, x_{n-1}], \textcolor{red}{0 \leq i < n} \}$$

**elem**(*it: Iterator*) → (*x: Elem*)

$$\{ x = x_i \}$$
$$\{ [x_0, \dots, |x_i, \dots, x_{n-1}], \textcolor{red}{0 \leq i < n} \}$$

**advance**(*it: Iterator*)

$$\{ [x_0, \dots, x_i, |x_{i+1}, \dots, x_{n-1}] \}$$

# Operaciones adicionales sobre listas

- Insertar un elemento en la posición apuntada por un iterador (**insert**)
- Eliminar el elemento apuntado por el iterador (**erase**)

$$\{ l = [ x_0, \dots, |x_i, \dots, x_{n-1}| ] \}$$

**insert**(*l*: List, *it*: Iterator, *e*: Elem) → *it'*: Iterator

$$\{ l = [ x_0, \dots, |e, x_i, \dots, x_{n-1}| ] \}$$
$$\{ l = [ x_0, \dots, |x_i, x_{i+1}, \dots, x_{n-1}| ], i < n \}$$

**erase**(*l*: List, *it*: Iterator) → *it'*: Iterator

$$\{ l = [ x_0, \dots, |x_{i+1}, \dots, x_{n-1}| ] \}$$

# Ejemplo: suma de enteros

```
int suma_elems(ListLinkedDouble<int> &l) {
    int suma = 0;
    ListLinkedDouble<int>::iterator it = l.begin();
    while (it != l.end()) {
        suma += it.elem();
        it.advance();
    }
    return suma;
}
```



# Ejemplo: suma de enteros

```
int suma_elems_iterator(ListLinkedDouble<int> &l) {  
    int suma = 0;  
    for (ListLinkedDouble<int>::iterator it = l.begin(); it != l.end(); it.advance()) {  
        suma += it.elem();  
    }  
    return suma;  
}
```



ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

# Iteradores y listas enlazadas

Manuel Montenegro Montes

Departamento de Sistemas Informáticos y Computación  
Facultad de Informática – Universidad Complutense de Madrid

# Operaciones a implementar

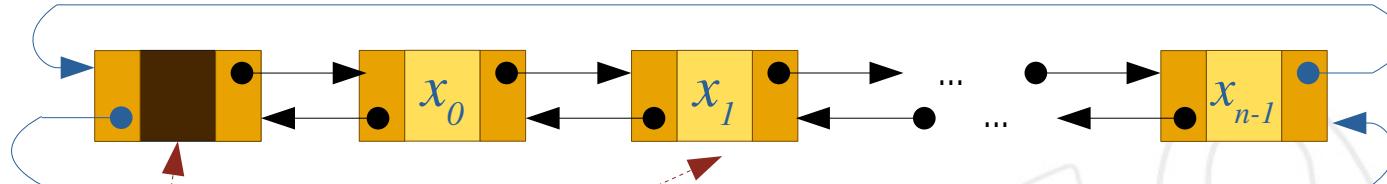
## Para iteradores

- Obtener el elemento apuntado por el iterador (**elem**)
- Avanzar el iterador a la siguiente posición de la lista (**advance**)
- Igualdad entre dos iteradores (==)
  - Dos iteradores son iguales si recorren la misma lista y apuntan a la misma posición dentro de esta.

## Para listas

- Obtener un iterador al principio de la lista (**begin**)
- Obtener un iterador al final de la lista (**end**)

# Iteradores en listas doblemente enlazadas



**Iterador**

head:	•
current:	•

- Un iterador contiene dos atributos:
  - El nodo del elemento apuntado por el iterador (**current**).
  - El nodo fantasma de la lista a la que el iterador pertenece (**head**).

# La clase ListLinkedDouble<T> :: iterator

```
template <typename T>
class ListLinkedDouble {
public:
    ...
    class iterator {
public:
    iterator(Node *head, Node *current): head(head), current(current) { }
    ...
private:
    Node *head;
    Node *current;
};
...
};
```

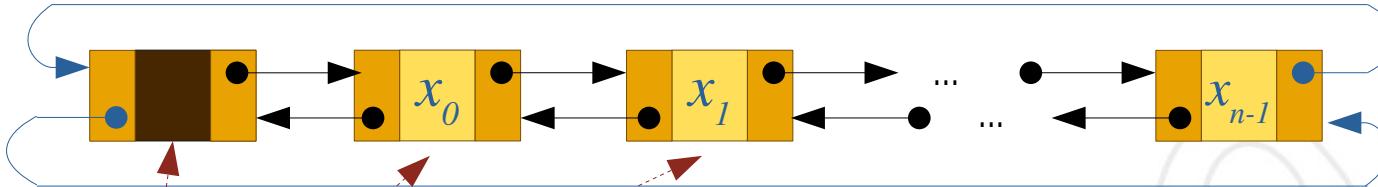
# La clase ListLinkedDouble<T> :: iterator

```
template <typename T>
class ListLinkedDouble {
public:
...
class iterator {
public:
...
void advance();
T & elem();
bool operator==(const iterator &other) const;
bool operator!=(const iterator &other) const;
...
};

};

...;
```

# Implementación de advance()

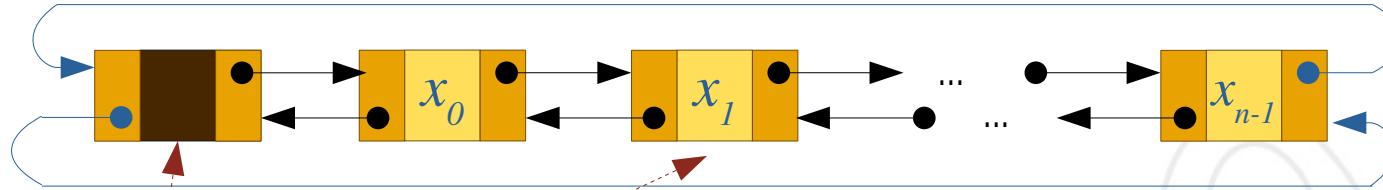


Iterador

head:	
current:	

```
class iterator {  
public:  
    void advance() {  
        assert (current != head);  
        current = current->next;  
    }  
    ...  
};
```

# Implementación de elem()



## Iterador

head:	
current:	

```
class iterator {
public:
    T & elem() {
        assert (current != head);
        return current->value;
    }
    ...
};
```

# Implementación de los operadores == y !=

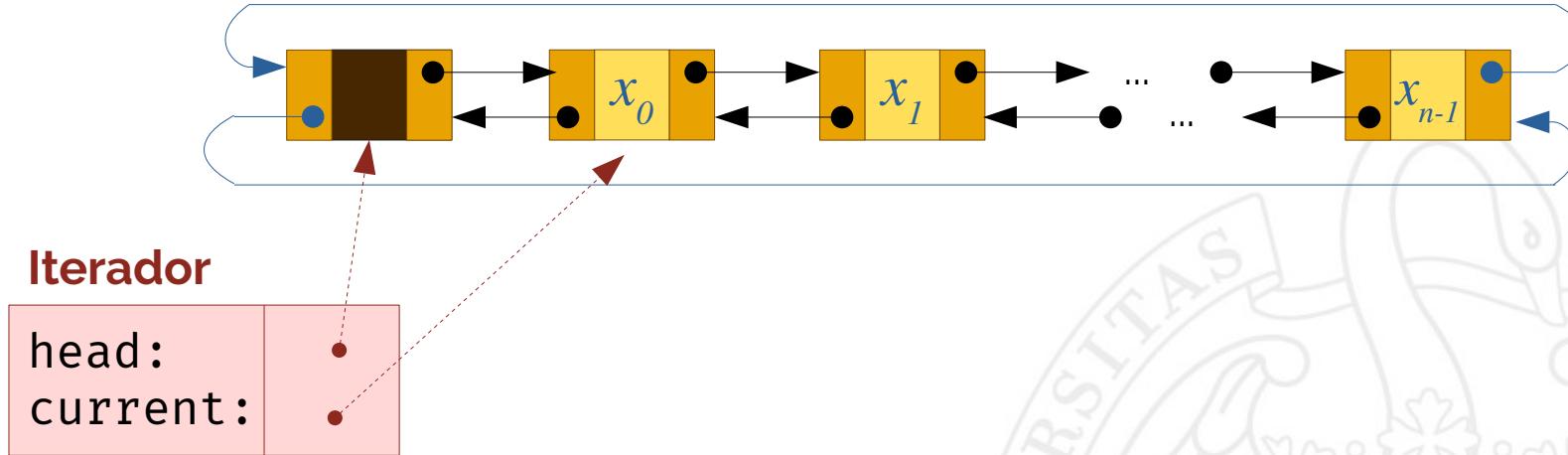
```
class iterator {  
public:  
    bool operator==(const iterator &other) const {  
        return (head == other.head) &&  
               (current == other.current);  
    }  
  
    bool operator!=(const iterator &other) const {  
        return !(*this == other);  
    }  
};
```

# Creación de iteradores

```
template <typename T>
class ListLinkedDouble {
public:
    class iterator { ... }
    ...
    iterator begin();
    iterator end();
    ...
};
```

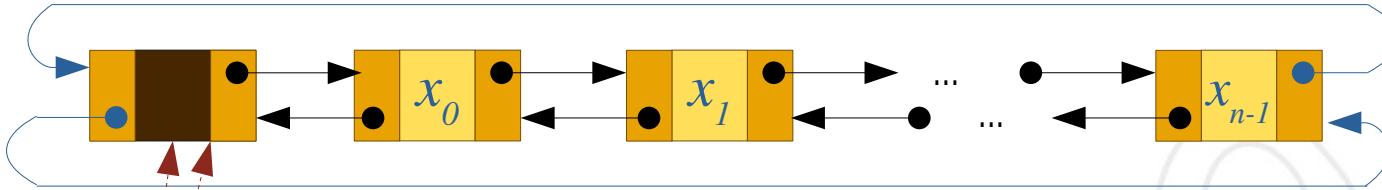
Nuevos métodos

# Implementación de begin()



```
template <typename T>
class ListLinkedDouble {
    ...
    iterator begin() {
        return iterator(head, head->next);
    }
};
```

# Implementación de end()



Iterador

head:	•
current:	•

```
template <typename T>
class ListLinkedDouble {
    ...
    iterator end() {
        return iterator(head, head);
    }
};
```

# Ejemplo

- Dada una lista de nombres de personas, imprimir aquellas que empiecen por un carácter pasado como parámetro:

```
void imprime_empieza_por(ListLinkedDouble<std::string> &l, char c) {  
    for (ListLinkedDouble<std::string>::iterator it = l.begin();  
        it != l.end();  
        it.advance()) {  
  
        if (it.elem()[0] == c) {  
            std::cout << it.elem() << std::endl;  
        }  
    }  
}
```

# Ejemplo

- Modificar todos los elementos de una lista de enteros, multiplicándolos por uno pasado como parámetro.

```
void multiplicar_por(ListLinkedDouble<int> &l, int num) {  
    for (ListLinkedDouble<int>::iterator it = l.begin();  
        it != l.end();  
        it.advance()) {  
        it.elem() = it.elem() * num;  
    }  
}  
  
class iterator {  
public:  
    ...  
    void advance();  
    T & elem();  
    bool operator==(const iterator &other) const;  
    bool operator!=(const iterator &other) const;  
    ...  
};
```

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

# Inserción y borrado con iteradores

Manuel Montenegro Montes

Departamento de Sistemas Informáticos y Computación  
Facultad de Informática – Universidad Complutense de Madrid

# Operaciones a implementar

## Para listas

- Insertar un elemento en la posición apuntada por un iterador (**insert**)
- Eliminar el elemento apuntado por el iterador (**erase**)

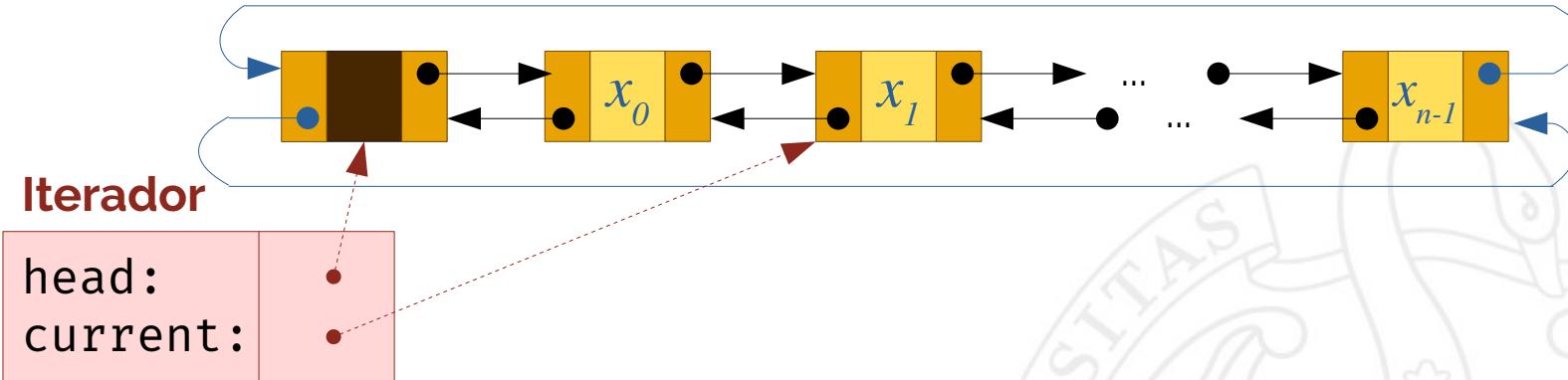


# Interfaz de ListLinkedDouble<T>

```
template <typename T>
class ListLinkedDouble {
public:
    ...
    iterator insert(iterator it, const T &elem);
    iterator erase(iterator it);
    ...
};
```

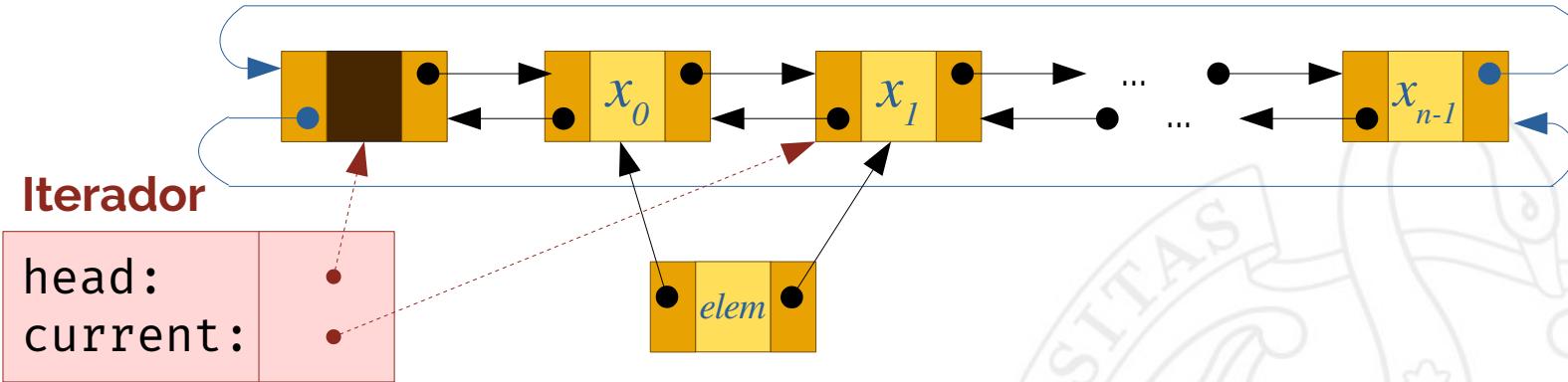


# Insertar un elemento



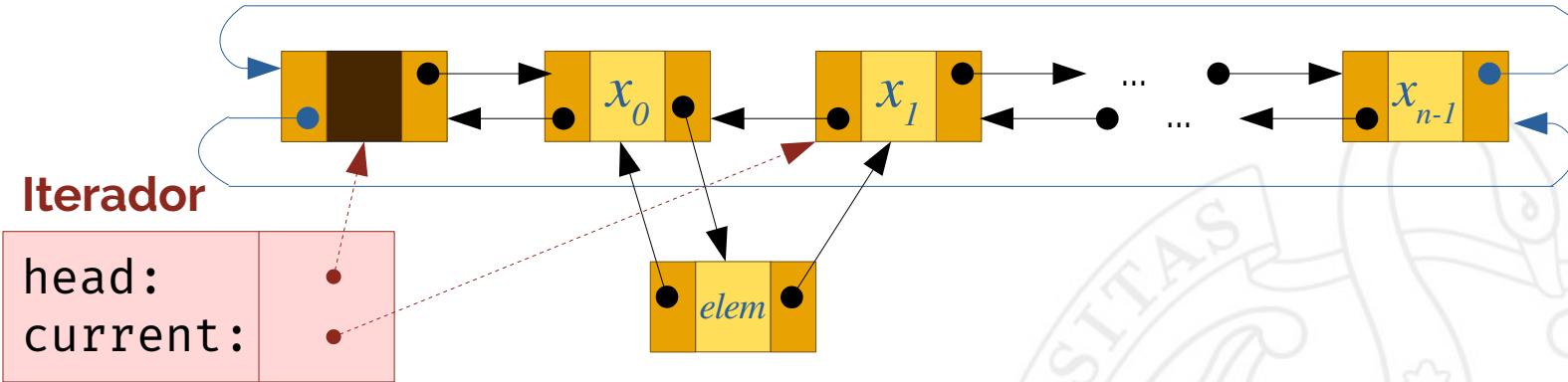
```
iterator insert(iterator it, const T &elem) {
    assert(it.head == head);
    Node *new_node = new Node { elem, it.current, it.current->prev };
    it.current->prev->next = new_node;
    it.current->prev = new_node;
    num_elems++;
    return iterator(head, new_node);
}
```

# Insertar un elemento



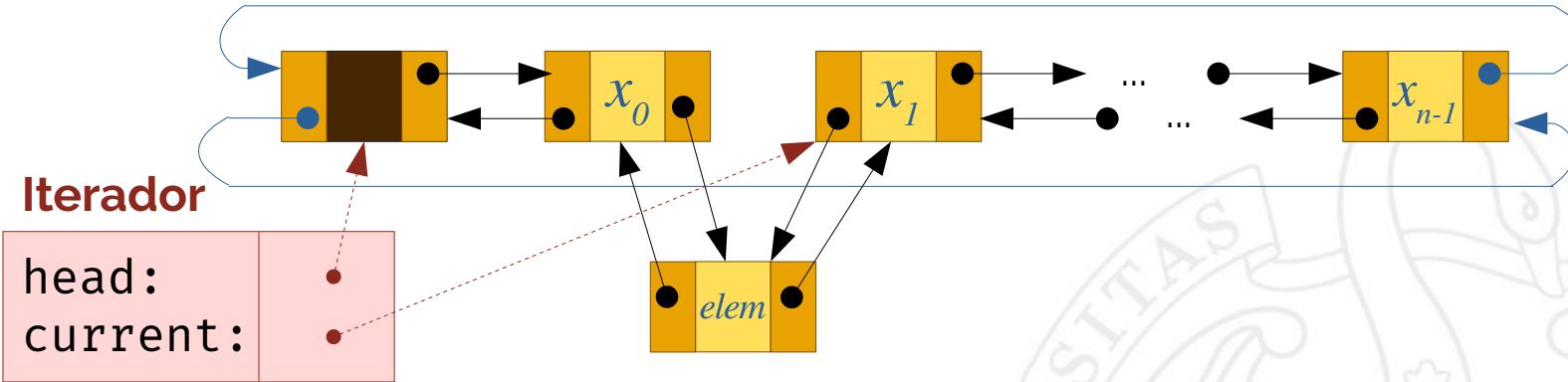
```
iterator insert(iterator it, const T &elem) {
    assert(it.head == head);
    Node *new_node = new Node { elem, it.current, it.current->prev };
    it.current->prev->next = new_node;
    it.current->prev = new_node;
    num_elems++;
    return iterator(head, new_node);
}
```

# Insertar un elemento



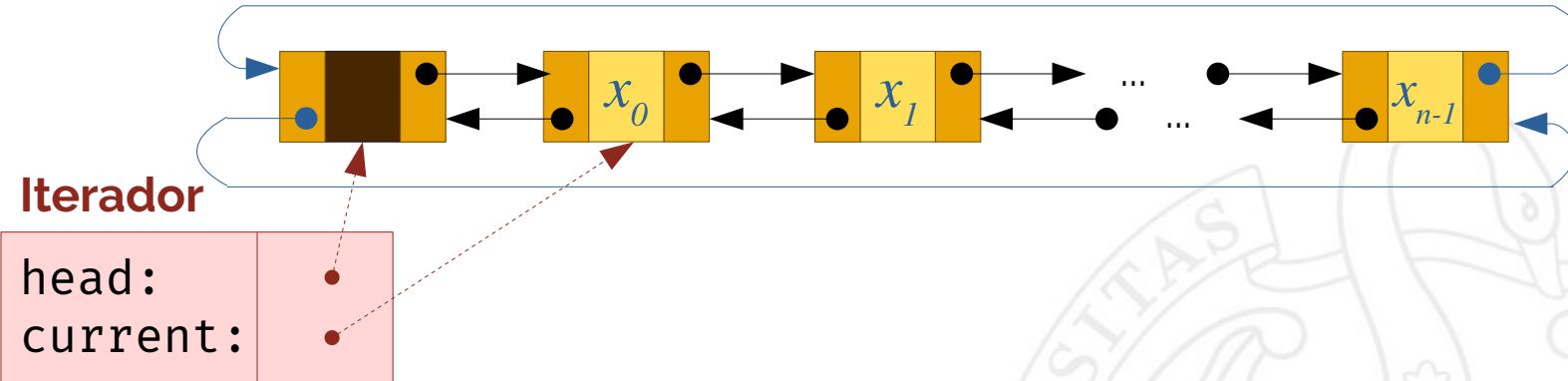
```
iterator insert(iterator it, const T &elem) {
    assert(it.head == head);
    Node *new_node = new Node { elem, it.current, it.current->prev };
    it.current->prev->next = new_node;
    it.current->prev = new_node;
    num_elems++;
    return iterator(head, new_node);
}
```

# Insertar un elemento



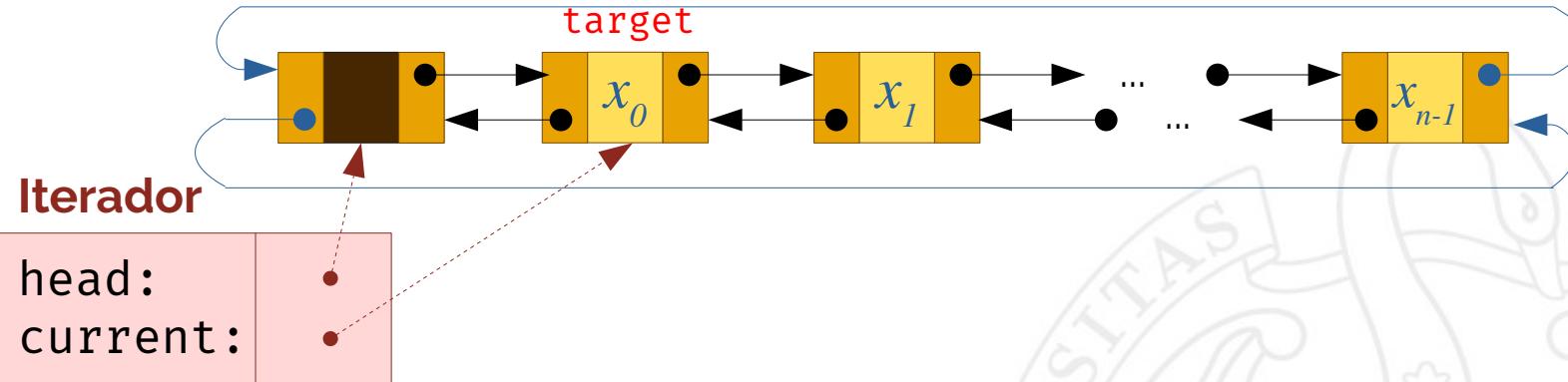
```
iterator insert(iterator it, const T &elem) {
    assert(it.head == head);
    Node *new_node = new Node { elem, it.current, it.current->prev };
    it.current->prev->next = new_node;
    it.current->prev = new_node;
    num_elems++;
    return iterator(head, new_node);
}
```

# Borrar un elemento



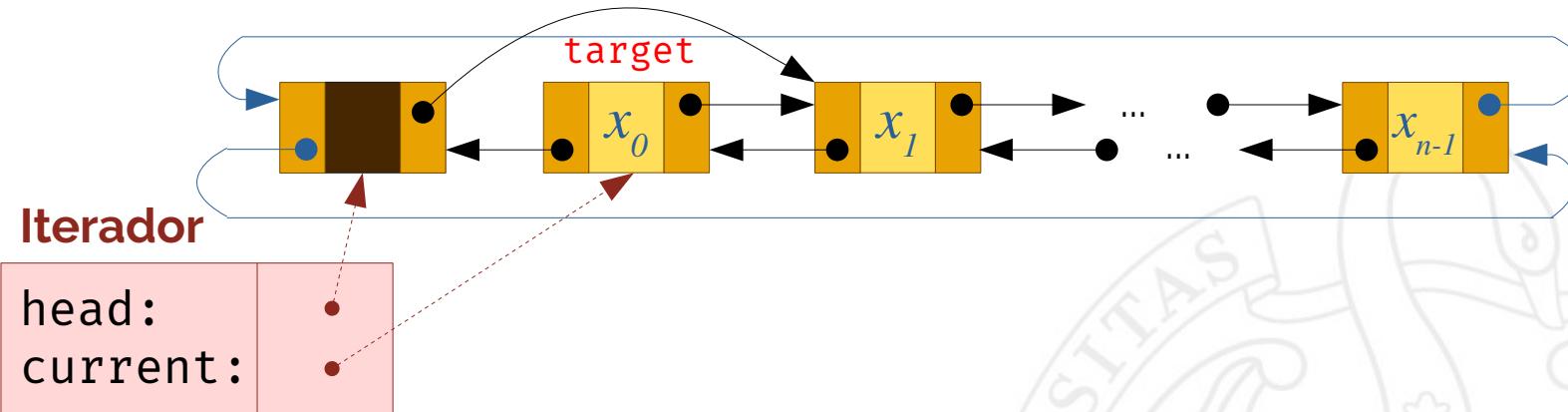
```
iterator erase(iterator it) {
    assert(it.head == head && it.current != head);
    Node *target = it.current;
    target->prev->next = it.current->next;
    target->next->prev = it.current->prev;
    iterator result(head, target->next);
    delete target;
    num_elems--;
    return result;
}
```

# Borrar un elemento



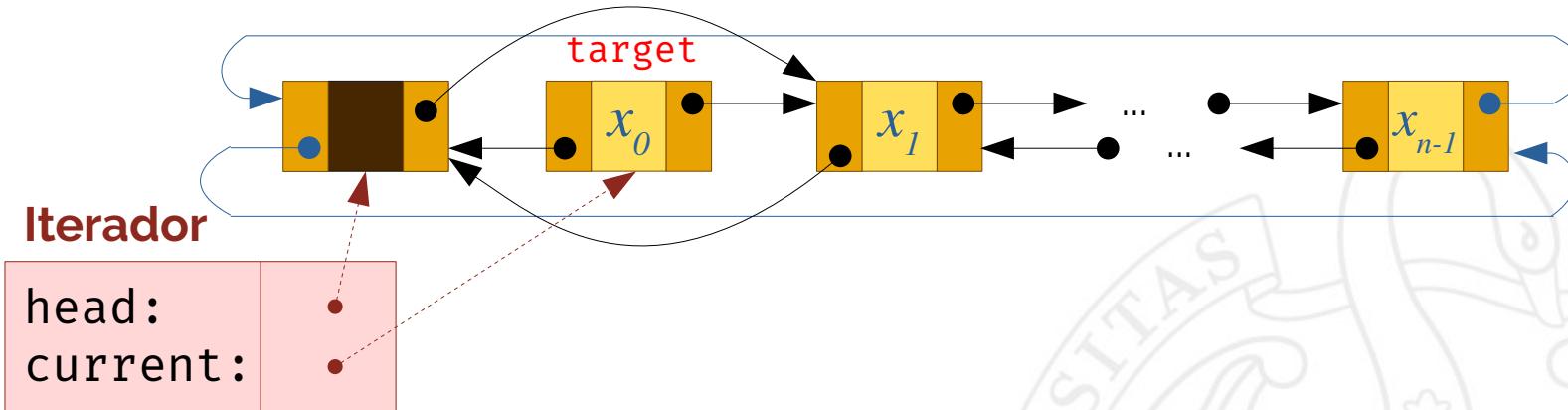
```
iterator erase(iterator it) {
    assert(it.head == head && it.current != head);
    Node *target = it.current;
    target->prev->next = it.current->next;
    target->next->prev = it.current->prev;
    iterator result(head, target->next);
    delete target;
    num_elems--;
    return result;
}
```

# Borrar un elemento



```
iterator erase(iterator it) {
    assert(it.head == head && it.current != head);
    Node *target = it.current;
    target->prev->next = it.current->next;
    target->next->prev = it.current->prev;
    iterator result(head, target->next);
    delete target;
    num_elems--;
    return result;
}
```

# Borrar un elemento

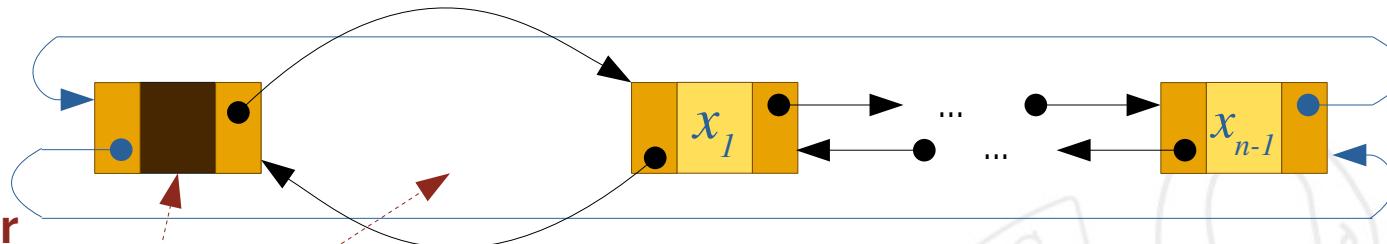


```
iterator erase(iterator it) {
    assert(it.head == head && it.current != head);
    Node *target = it.current;
    target->prev->next = it.current->next;
    target->next->prev = it.current->prev;
    iterator result(head, target->next);
    delete target;
    num_elems--;
    return result;
}
```

# Borrar un elemento

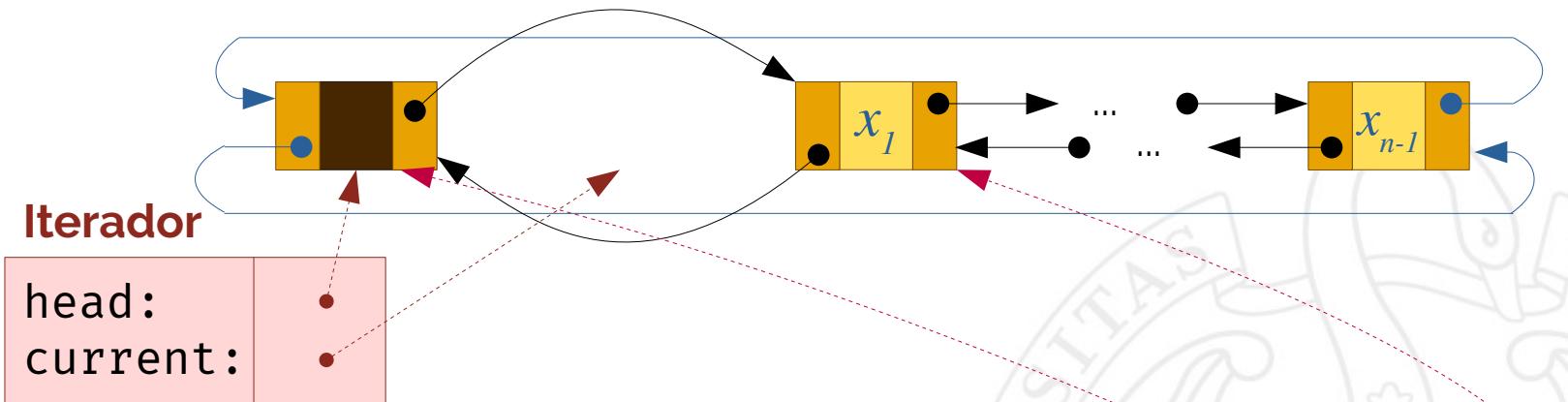
Iterador

head:  
current:



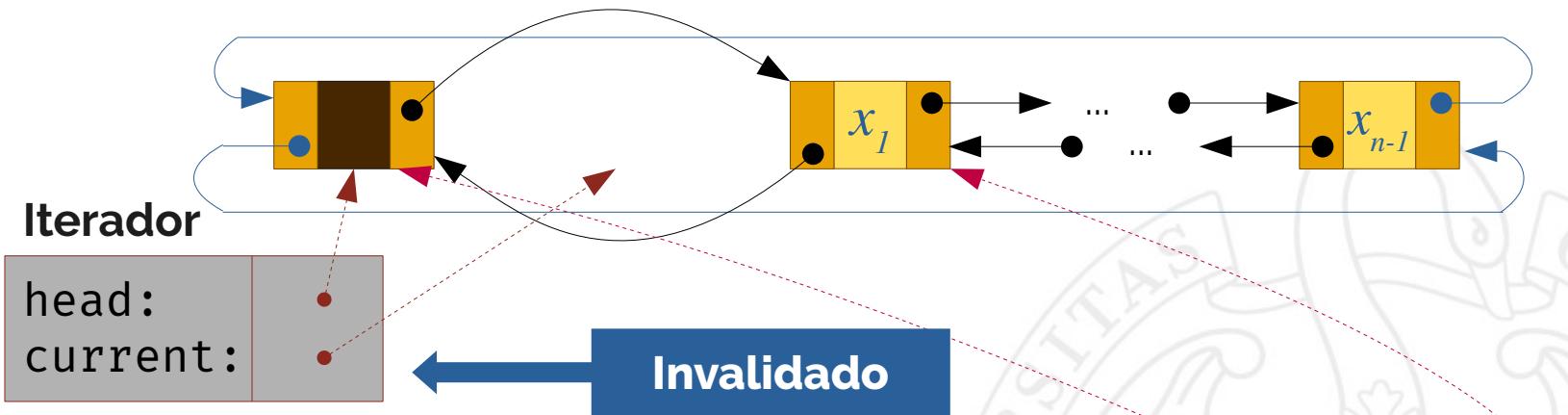
```
iterator erase(iterator it) {
    assert(it.head == head && it.current != head);
    Node *target = it.current;
    target->prev->next = it.current->next;
    target->next->prev = it.current->prev;
    iterator result(head, target->next);
    delete target;
    num_elems--;
    return result;
}
```

# Borrar un elemento

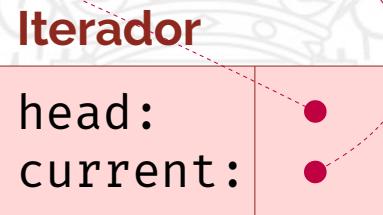


```
iterator erase(iterator it) {
    assert(it.head == head && it.current != head);
    Node *target = it.current;
    target->prev->next = it.current->next;
    target->next->prev = it.current->prev;
    iterator result(head, target->next);
    delete target;
    num_elems--;
    return result;
}
```

# Borrar un elemento



```
iterator erase(iterator it) {
    assert(it.head == head && it.current != head);
    Node *target = it.current;
    target->prev->next = it.current->next;
    target->next->prev = it.current->prev;
    iterator result(head, target->next);
    delete target;
    num_elems--;
    return result;
}
```



ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

# Iteradores constantes

Manuel Montenegro Montes

Departamento de Sistemas Informáticos y Computación  
Facultad de Informática – Universidad Complutense de Madrid

# Ejemplo

- Volvemos al ejemplo de la suma de una lista de enteros:

```
int suma_elems(ListLinkedDouble<int> &l) {  
    int suma = 0;  
    for (ListLinkedDouble<int>::iterator it = l.begin();  
         it != l.end();  
         it.advance()) {  
  
        suma += it.elem();  
    }  
    return suma;  
}
```

# ¿Podemos pasar l por referencia constante?

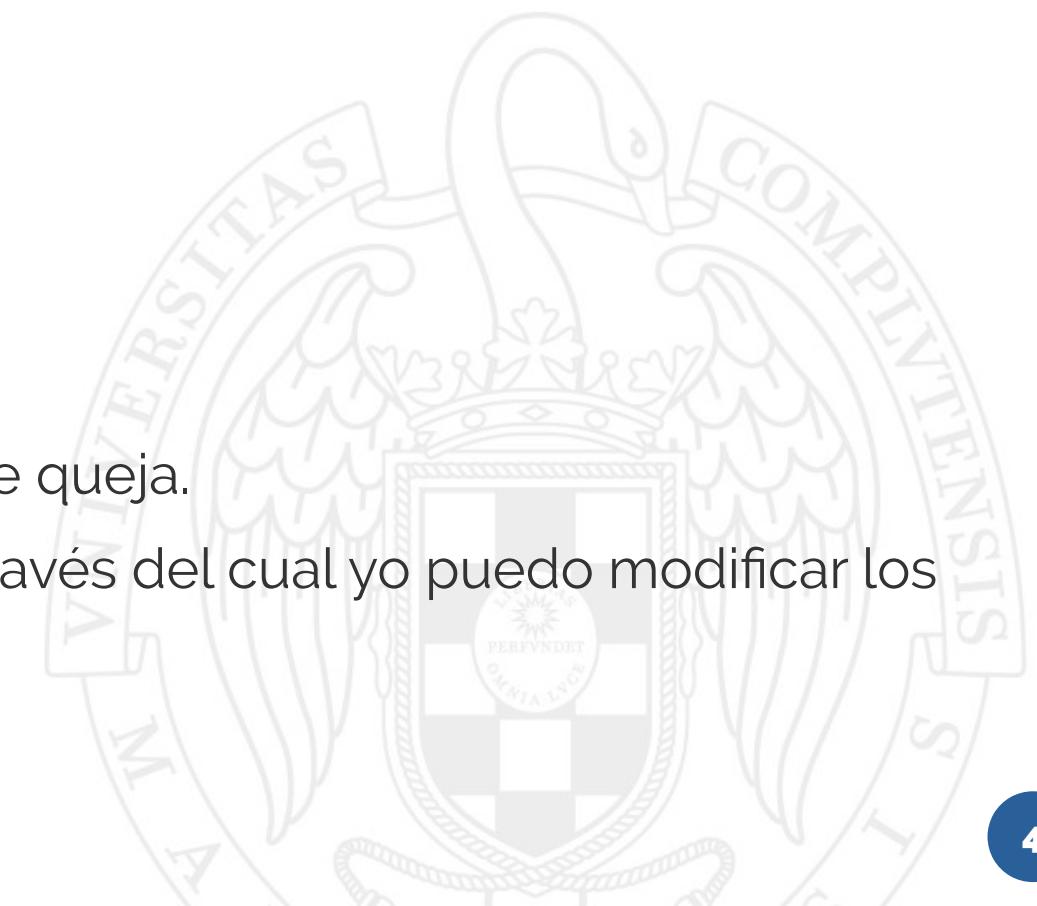
- No, porque begin() y end() no son métodos constantes.

```
int suma_elems(const ListLinkedDouble<int> &l) {  
    int suma = 0;  
    for (ListLinkedDouble<int>::iterator it = l.begin();  
         it != l.end();  
         it.advance()) {  
  
        suma += it.elem();  
    }  
    return suma;  
}
```

# ¿Y si begin() y end() fueran constantes?

```
template <typename T>
class ListLinkedDouble {
public:
    ...
    iterator begin() const;
    iterator end() const;
};
```

- Técnicamente, el compilador no se queja.
- Pero devuelven un **iterator**, a través del cual yo puedo modificar los elementos de la lista.



# ¿Por qué iterator puede modificar los elementos de la lista?

```
class iterator {  
public:  
    void advance();  
    T & elem();  
    bool operator==(const iterator &other) const;  
    bool operator!=(const iterator &other) const;  
    ...  
};
```

- Porque elem( ) devuelve una referencia al elemento apuntado por el iterador.
- A partir de esa referencia puedo cambiar el valor de ese elemento.

# ¿Y si elem() devolviera una referencia constante?

```
class iterator {  
public:  
    void advance();  
    const T & elem();  
    bool operator==(const iterator &other) const;  
    bool operator!=(const iterator &other) const;  
    ...  
};
```

# ¿Y si elem() devolviera una referencia constante?

- Ya no podría tener programas como este:

```
void multiplicar_por(ListLinkedDouble<int> &l, int num) {  
    for (ListLinkedDouble<int>::iterator it = l.begin();  
        it != l.end();  
        it.advance()) {  
        it.elem() = it.elem() * num;  
    }  
}
```

# Moraleja

- A veces quiero que los iteradores me devuelvan referencias no constantes, porque quiero utilizar un iterador para alterar los elementos de la lista.

```
void multiplicar_por(ListLinkedDouble<int> &l, int num) {  
    for (ListLinkedDouble<int>::iterator it = l.begin();  
        it != l.end();  
        it.advance()) {  
        it.elem() = it.elem() * num;  
    }  
}
```

# Moraleja

- A veces quiero que los iteradores me devuelvan referencias no constantes, porque quiero utilizar un iterador para alterar los elementos de la lista.
- Otras veces quiero iterar sobre una lista sin modificar sus elementos.

```
int suma_elems(const ListLinkedDouble<int> &l) {
    int suma = 0;
    for (ListLinkedDouble<int>::iterator it = l.begin();
         it != l.end();
         it.advance()) {
        suma += it.elem();
    }
    return suma;
}
```

# Moraleja

- A veces quiero que los iteradores me devuelvan referencias no constantes, porque quiero utilizar un iterador para alterar los elementos de la lista.
- Otras veces quiero iterar sobre una lista sin modificar sus elementos.
- Tengo que distinguir dos tipos de iteradores:
  - **Iteradores no constantes** (`iterator`)
  - **Iteradores constantes** (`const_iterator`)

# Iteradores constantes y no constantes

```
class iterator {  
public:  
    void advance();  
    T & elem();  
    bool operator==(const iterator &other) const;  
    bool operator!=(const iterator &other) const;  
    ...  
};  
  
class const_iterator {  
public:  
    void advance();  
    const T & elem();  
    bool operator==(const iterator &other) const;  
    bool operator!=(const iterator &other) const;  
    ...  
};
```

Las implementaciones de ambas clases son exactamente iguales

# Iteradores constantes y no constantes

```
template <typename T>
class ListLinkedDouble {
public:
    ...
    iterator begin();
    iterator end();

    const_iterator cbegin() const;
    const_iterator cend() const;
};

};
```

Funciones no constantes.  
Devuelven iteradores que  
me permiten modificar la lista.

Funciones constantes.  
Garantizan que no puedo  
modificar la lista con el  
iterador que devuelvan.

# Consecuencias

- Podemos utilizar iteradores para modificar elementos de la lista.
- Para ello utilizamos la clase `iterator`, como siempre.

```
void multiplicar_por(ListLinkedDouble<int> &l, int num) {  
    for (ListLinkedDouble<int>::iterator it = l.begin();  
        it != l.end();  
        it.advance()) {  
        it.elem() = it.elem() * num;  
    }  
}
```

# Consecuencias

- Podemos utilizar iteradores para recorrer la lista sin modificarla, y así poder declarar el objeto correspondiente como constante.
- Para ello utilizamos la clase `const_iterator`, y los métodos `cbegin()` y `cend()`.

```
int suma_elems(const ListLinkedDouble<int> &l) {
    int suma = 0;
    for (ListLinkedDouble<int>::const_iterator it = l.cbegin();
        it != l.cend();
        it.advance()) {

        suma += it.elem();
    }
    return suma;
}
```

# Cuánta duplicación, ¿no?

```
class iterator {  
public:  
    void advance();  
    T & elem();  
    bool operator==(const iterator &other) const;  
    bool operator!=(const iterator &other) const;  
    ...  
};  
  
class const_iterator {  
public:  
    void advance();  
    const T & elem();  
    bool operator==(const iterator &other) const;  
    bool operator!=(const iterator &other) const;  
    ...  
};
```

Sólo difieren en  
el tipo de retorno  
de elem( )!

# Cuánta duplicación, ¿no?

- Podemos utilizar las plantillas de C++:

```
template <typename U>
class gen_iterator {
public:
    void advance();
    U & elem();
    bool operator==(const gen_iterator &other) const;
    bool operator!=(const gen_iterator &other) const;
    ...
};
```

En iteradores no constantes:  $U = T$

En iteradores constantes:  $U = \text{const } T$

# Cuánta duplicación, ¿no?

- Podemos utilizar las plantillas de C++:

```
template <typename U>
class gen_iterator {
public:
    void advance();
    U & elem();
    bool operator==(const gen_iterator &other) const;
    bool operator!=(const gen_iterator &other) const;
    ...
};

using iterator = gen_iterator<T>;
using const_iterator = gen_iterator<const T>;
```

# En resumen, tenemos

```
template <typename T>
class ListLinkedDouble {
public:
    ...

    template <typename U>
    class gen_iterator { ... }

    using iterator = gen_iterator<T>;
    using const_iterator = gen_iterator<const T>;

    iterator begin();
    iterator end();

    const_iterator cbegin() const;
    const_iterator cend() const;

};
```



ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

# Adaptando la sintaxis de los iteradores

Manuel Montenegro Montes

Departamento de Sistemas Informáticos y Computación  
Facultad de Informática – Universidad Complutense de Madrid

# Recordatorio: suma de elementos

```
int suma_elems(const ListLinkedDouble<int> &l) {
    int suma = 0;
    for (ListLinkedDouble<int>::const_iterator it = l.cbegin();
        it != l.cend();
        it.advance()) {
        suma += it.elem();
    }
    return suma;
}
```



# Cambio de sintaxis

```
int suma_elems(const ListLinkedDouble<int> &l) {
    int suma = 0;
    for (ListLinkedDouble<int>::const_iterator it = l.cbegin();
        it != l.cend();
        ++it) {
        suma += *it;
    }
    return suma;
}
```

# Sobrecarga del operador \*

# Sobrecarga del operador \*

- El operador \* tiene dos significados en C++:
  - Multiplicación (binario). Ejemplo: `5 * x`
  - Indirección (unario). Ejemplo: `*y`

Queremos sobrecargar  
este

# Sobrecarga del operador \*

Antes

```
template <typename U>
class gen_iterator {
public:
    ...
    U & elem() const {
        assert (current != head);
        return current->value;
    }
};
```

Ahora

```
template <typename U>
class gen_iterator {
public:
    ...
    U & operator*() const {
        assert (current != head);
        return current->value;
    }
};
```

# Sobrecarga del operador ++

# Sobrecarga del operador ++

- El operador de incremento ++ también tiene dos significados en C++:
  - Preincremento, cuando se sitúa a la izquierda de lo que se quiere incrementar. Ejemplo: `++x`
  - Postincremento, cuando se sitúa a la derecha de lo que se quiere incrementar. Ejemplo: `x++`

# Preincremento vs postincremento

- Aplicados a tipos numéricos, ambos incrementan en una unidad el valor de la variable a la que se aplican.
- Pero:
  - El operador de preincremento devuelve el valor de la variable **tras** haberla incrementado.
  - El operador de postincremento devuelve el valor de la variable **antes de** haberla incrementado.

```
int x = 2;  
int z = ++x;  
// x vale 3, z vale 3
```

```
int x = 2;  
int z = x++;  
// x vale 3, z vale 2
```

# ¿Tanto nos interesa esto?

- En general no, porque casi siempre utilizamos las expresiones de incremento de manera aislada, sin asignar el valor resultante:

```
while (x < 10) {  
    // ...  
    x++;  
}
```

```
for (int i = 0; i < 10; i++) {  
    // ...  
}
```

- PERO... tenemos que conocer esta distinción a la hora de sobrecargar los operadores, para saber qué tenemos que devolver:
  - `it++` devuelve el iterador antes de haberlo avanzado.
  - `++it` devuelve el iterador tras haberlo avanzado.

# Sobrecarga del operador ++

Antes

```
template <typename U>
class gen_iterator {
public:
...
void advance() const {
    assert (current != head);
    current = current->next;
}
};
```

Ahora

```
gen_iterator & operator++() {
    assert (current != head);
    current = current->next;
    return *this;
}

gen_iterator operator++(int) {
    assert (current != head);
    gen_iterator antes = *this;
    current = current->next;
    return antes;
}
```

# Otro ejemplo

- Multiplicar todos los elementos de una lista por dos:

```
void mult_por_dos(ListLinkedDouble<int> &l) {  
    for (ListLinkedDouble<int>::iterator it = l.begin();  
        it != l.end();  
        ++it) {  
        *it = *it * 2;  
    }  
}
```

# El especificador de tipo auto (C++11)

# El especificador de tipo auto

- Cuando declaramos e inicializamos una variable, podemos indicar que su tipo es **auto**.
- En este caso, C++ infiere el tipo de la variable a partir del valor con el que se inicializa.

```
auto suma = 0;
```

↓  
equivale a

```
int suma = 0;
```

```
auto nombre = "";
```

↓  
equivale a

```
const char *nombre = "";
```

# Ejemplo

- Antes de utilizar auto:

```
int suma_elems(const ListLinkedDouble<int> &l) {
    int suma = 0;
    for (ListLinkedDouble<int>::const_iterator it = l.cbegin();
        it != l.cend();
        ++it) {

        suma += *it;
    }
    return suma;
}
```

# Ejemplo

- Después de utilizar auto:

```
int suma_elems(const ListLinkedDouble<int> &l) {  
    int suma = 0;  
    for (auto it = l.cbegin(); it != l.cend(); ++it) {  
        suma += *it;  
    }  
    return suma;  
}
```



# Bucles for basados en rango (C++11)

# Bucles for basados en rango

- C++11 introduce una sintaxis nueva en los bucles for:

```
for (tipo variable : expresion) {  
    cuerpo  
}
```

equivale (casi) a:

```
for (auto it = expresion.begin(); it != expresion.end(); ++it) {  
    tipo variable = *it;  
    cuerpo  
}
```

<https://en.cppreference.com/w/cpp/language/range-for>

# Bucles for basados en rango

```
int suma_elems(const ListLinkedDouble<int> &l) {  
    int suma = 0;  
    for (int x : l) {  
        suma += x;  
    }  
    return suma;  
}
```



# Bucles for basados en rango

```
int suma_elems(const ListLinkedDouble<int> &l) {  
    int suma = 0;  
    for (int x : l) {  
        suma += x;  
    }  
    return suma;  
}
```

```
int suma_elems(const ListLinkedDouble<int> &l) {  
    int suma = 0;  
    for (auto it = l.begin(); it != l.end(); ++it) {  
        int x = *it;  
        suma += x;  
    }  
    return suma;  
}
```

# Bucles for basados en rango

- Multiplicar todos los elementos de una lista por dos:

```
void mult_por_dos(ListLinkedDouble<int> &l) {  
    for (int &x : l) {  
        x *= 2;  
    }  
}
```



ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

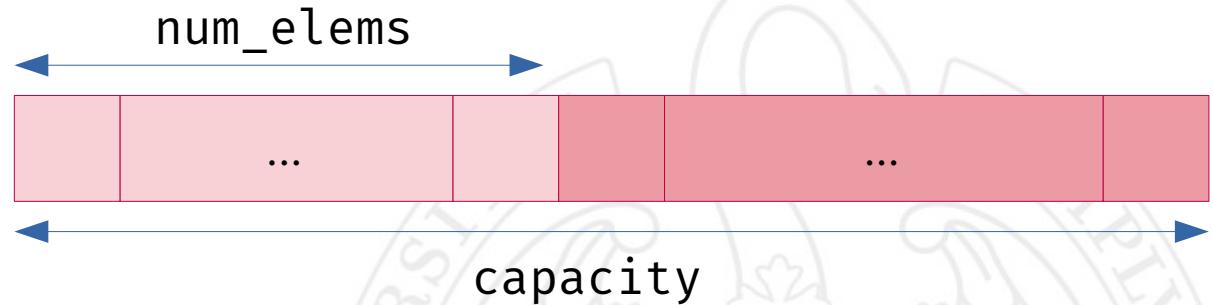
# Iteradores en ListArray

Manuel Montenegro Montes

Departamento de Sistemas Informáticos y Computación  
Facultad de Informática – Universidad Complutense de Madrid

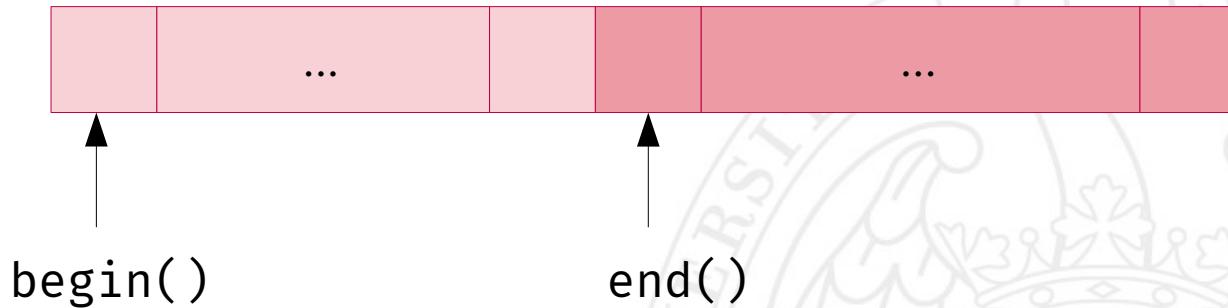
# Recordatorio: ListArray

```
template<typename T>
class ListArray {
public:
...
private:
    int num_elems;
    int capacity;
    T *elems;
};
```



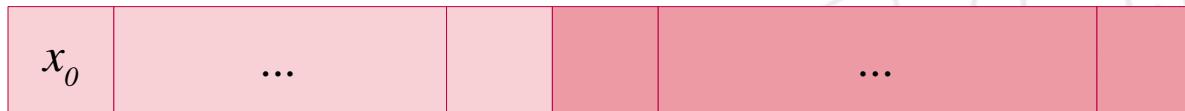
# Iteradores en ListArray

- Los iteradores van a ser **punteros** a los elementos del array.



# Iteradores en ListArray

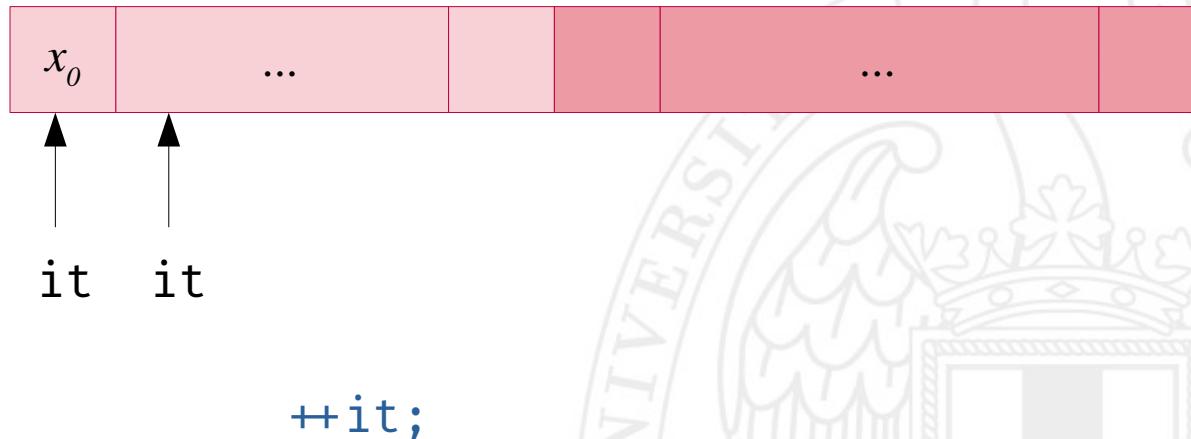
- Los iteradores van a ser **punteros** a los elementos del array.



$*it = x_0$

# Iteradores en ListArray

- Los iteradores van a ser **punteros** a los elementos del array.



# Definición de iteradores

```
template<typename T>
class ListArray {
public:
    ...
    using iterator = T *;
    using const_iterator = const T *;

private:
    int num_elems;
    int capacity;
    T *elems;
};
```



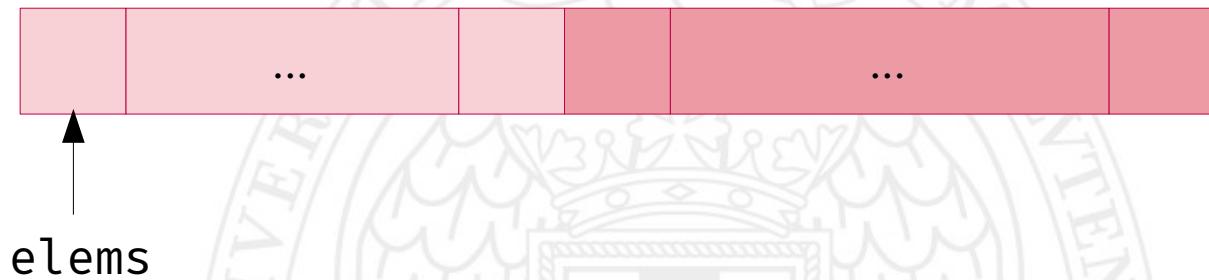
# Método begin()

```
template<typename T>
class ListArray {
public:
    ...
    using iterator = T *;
    using const_iterator = const T *;

    iterator begin() {
        return elems;
    }

    const_iterator begin() const {
        return elems;
    }

private:
    int num_elems;
    int capacity;
    T *elems;
};
```



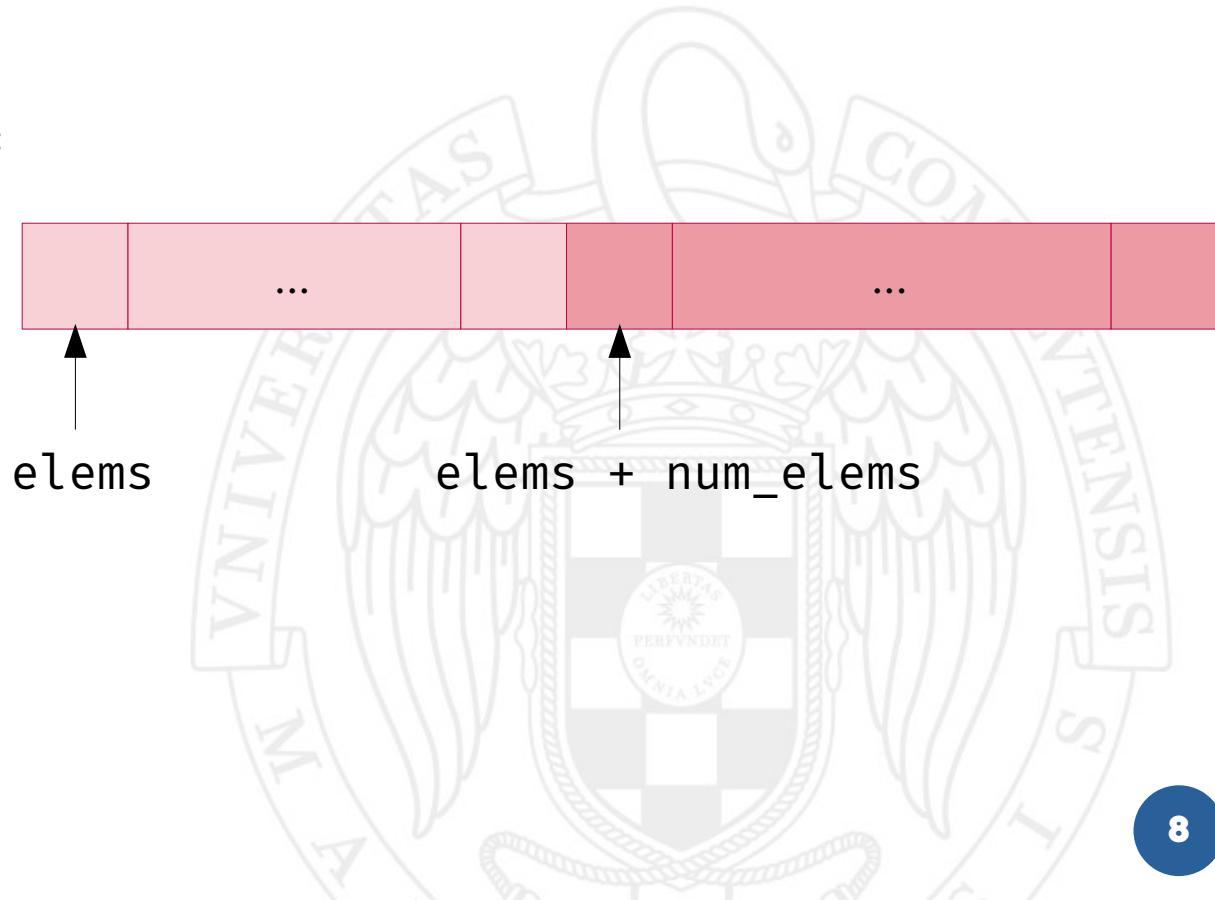
# Método end()

```
template<typename T>
class ListArray {
public:
    ...
    using iterator = T *;
    using const_iterator = const T *;

    iterator end() {
        return elems + num_elems;
    }

    const_iterator end() const {
        return elems + num_elems;
    }

private:
    int num_elems;
    int capacity;
    T *elems;
};
```



# Ejemplos

```
void mult_por_dos(ListArray<int> &l) {
    for (auto it = l.begin(); it != l.end(); ++it) {
        *it *= 2;
    }
}

int suma_elems(const ListArray<int> &l) {
    int suma = 0;
    for (auto it = l.begin(); it != l.end(); ++it) {
        suma += *it;
    }
    return suma;
}
```

# Ejemplos

```
void mult_por_dos(ListArray<int> &l) {
    for (int &x : l) {
        x *= 2;
    }
}

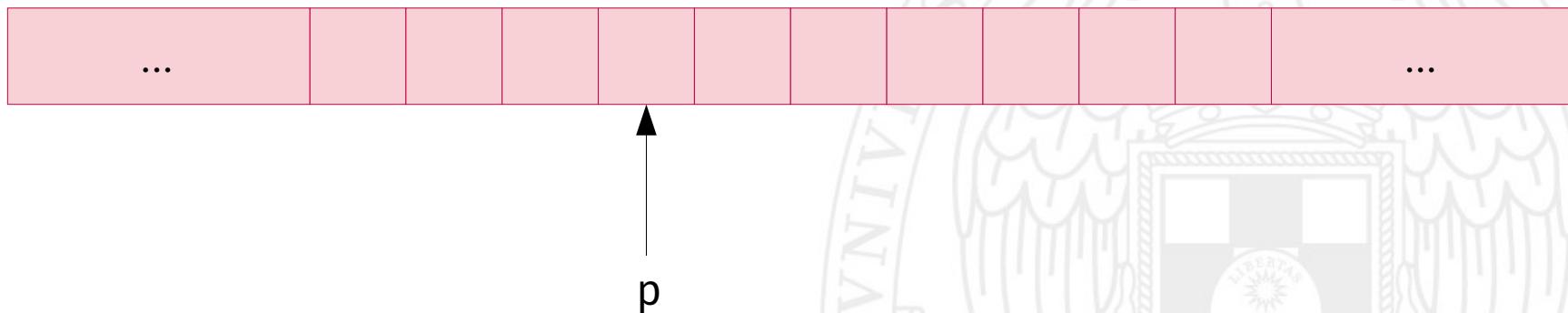
int suma_elems(const ListArray<int> &l) {
    int suma = 0;
    for (int x : l) {
        suma += x;
    }
    return suma;
}
```



# Moraleja

- En C++, los iteradores son generalizaciones de punteros.

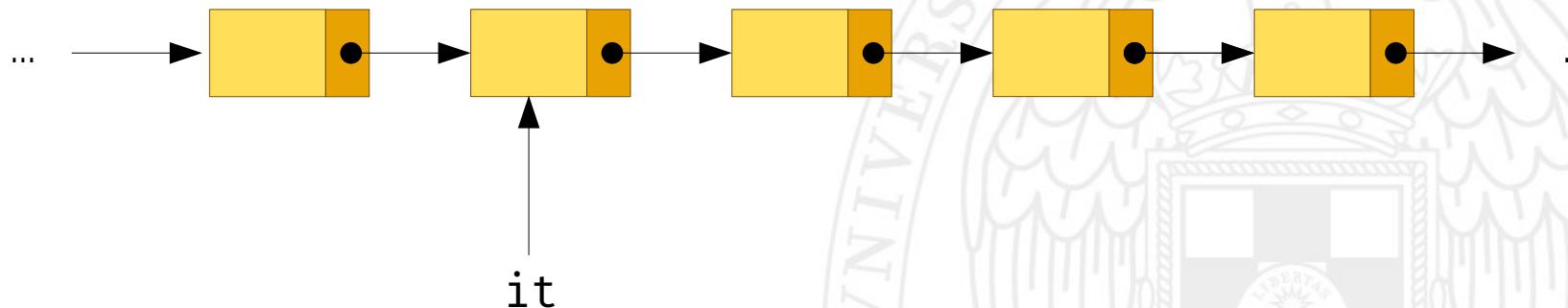
```
char *p;
```



# Moraleja

- En C++, los iteradores son generalizaciones de punteros.

```
ListLinked<int>::iterator it;
```

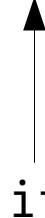


# Moraleja

- En C++, los iteradores son generalizaciones de punteros.

```
string::iterator it;
```

“Hola, mundo”



it



# Moraleja

- En C++, los iteradores son generalizaciones de punteros.

```
std::string cadena = "Hola, mundo";
for (auto it = cadena.begin(); it != cadena.end(); ++it) {
    std::cout << *it << std::endl;
}
```