ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

# Listas doblemente enlazadas (2)

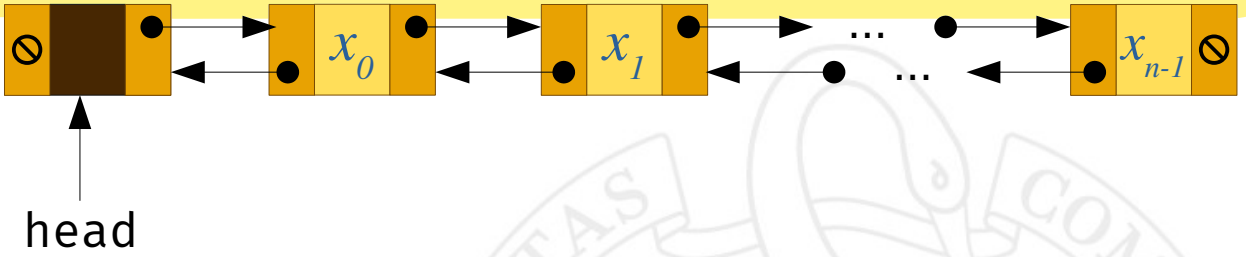
Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid

# Listas doblemente enlazadas

SEGUIMOS CON LAS LISTAS DOBLEMENTE ENLAZADAS

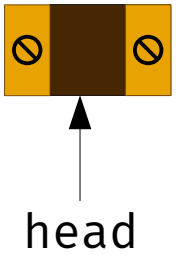
Vamos a buscar mejoras en el coste asintótico. Hasta ahora hemos visto modificaciones con la introducción del nodo fantasma PERO no mejoras del coste.

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




Añadiremos un puntero al último nodo de la lista de forma que operaciones de back no tendrán coste lineal si no que lo tendrán constante.

$[\ ]$



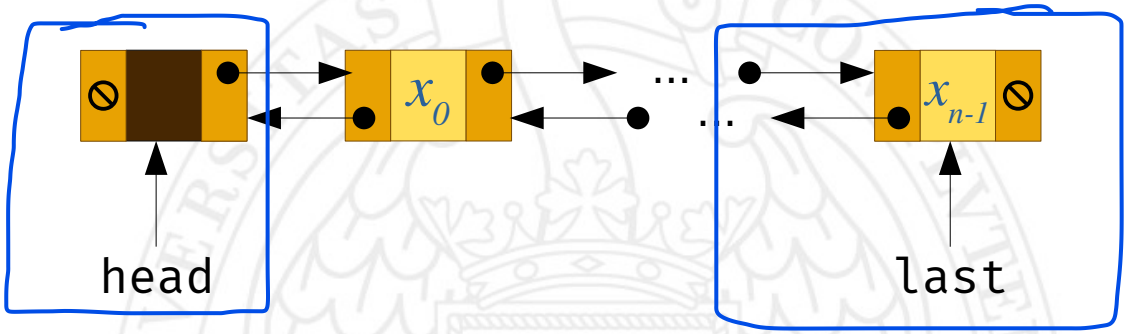
# Mejorando algunas operaciones

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

Queremos que sean de coste constante.

# 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**

Puntero al último nodo dentro de la lista enlazada. De esta forma si queremos acceder al último nodo de la lista, accederemos al atributo last.

# 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.

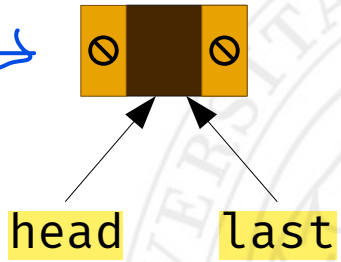
Ya que tener un método que me devuelva un atributo privado es una tontería.

# Creación de una cadena de nodos

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

CONSTRUCTOR.

La cabeza y last empiezan apuntando ambos al nodo fantasma



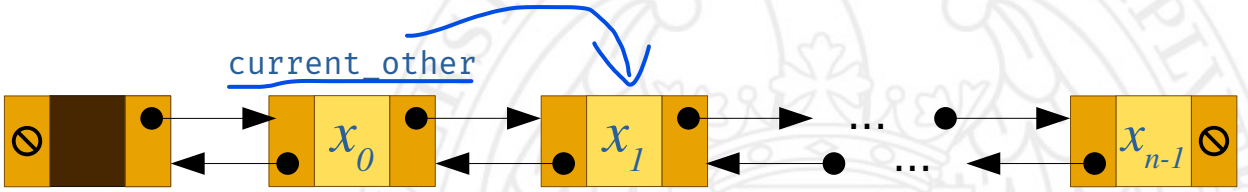
# Copia de una cadena de nodos

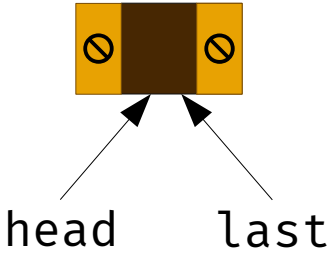
Constructor de copia.

```
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;
  }
}
```

siguiente a la cabeza

En vez de versión recursiva, lo hace iterativo

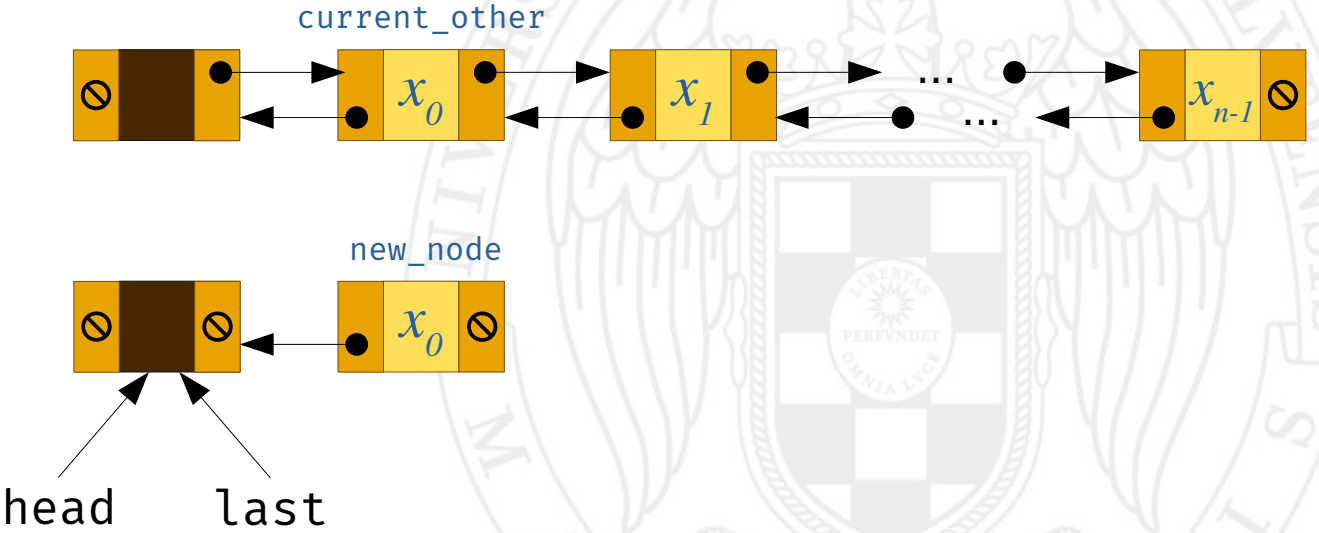


# Copia de una cadena de nodos

```
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;
  }
}
```

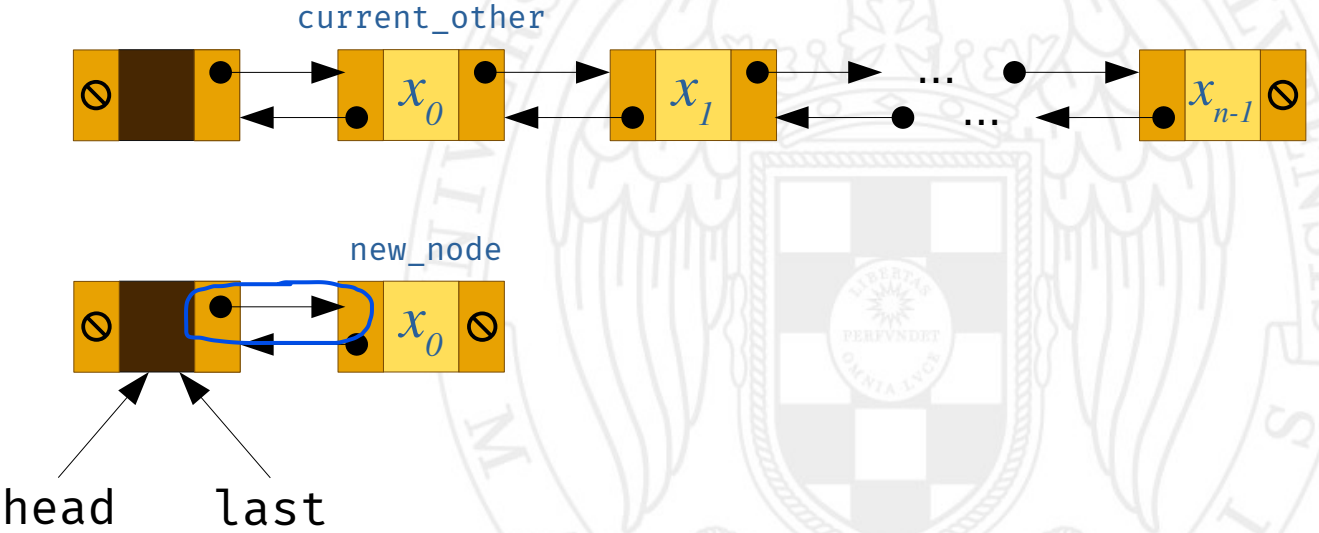
sucesor predecesor



# 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;
  }
}
```

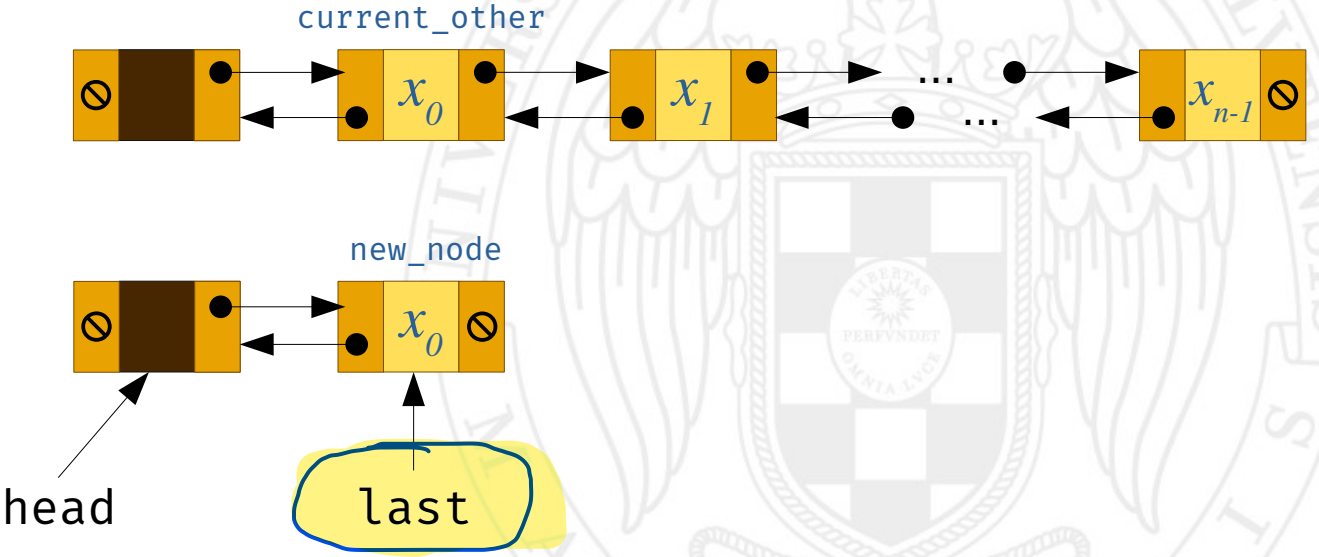
hacemos que el nodo last apunte al nuevo nodo de la lista.



# 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;
  }
}
```

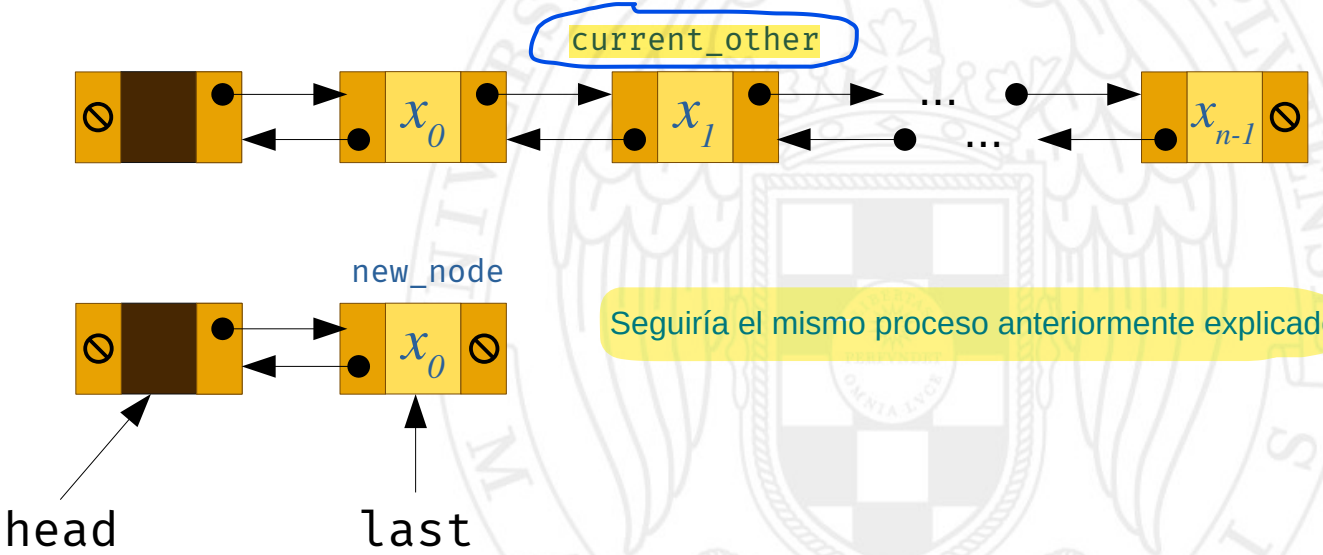
Last apunta al nuevo nodo.



# 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;
  }
}
```

Avanzamos current other para copiar el siguiente nodo de la lista.

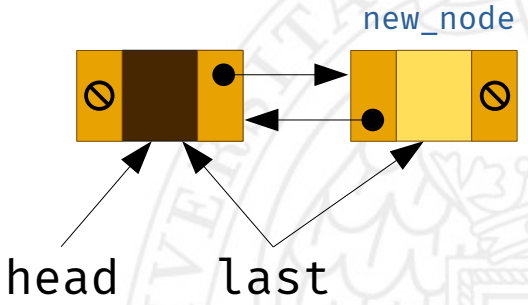



Seguiría el mismo proceso anteriormente explicado

# 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;
  }
}
```

Si la lista comienza teniendo únicamente al NODO FANTASMA entonces haremos que last apunte al nuevo nodo añadido



# 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;
}
```

Si resulta que la lista solo tenía dos nodos: Uno normal y el nodo fantasma, last apuntaría al nodo que queremos eliminar. Por tanto tendríamos que hacer que last apunte al mismo nod que head, al nodo fantasma

# 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;
}
```

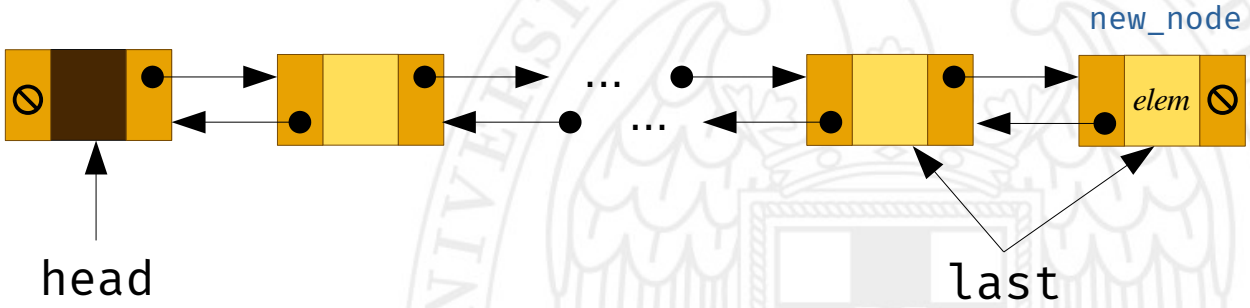
siguiente

anterior es el antiguo last.

El elemento que añadimos al final es el nuevo último

el siguiente del antiguo last es el nuevo en vez de null

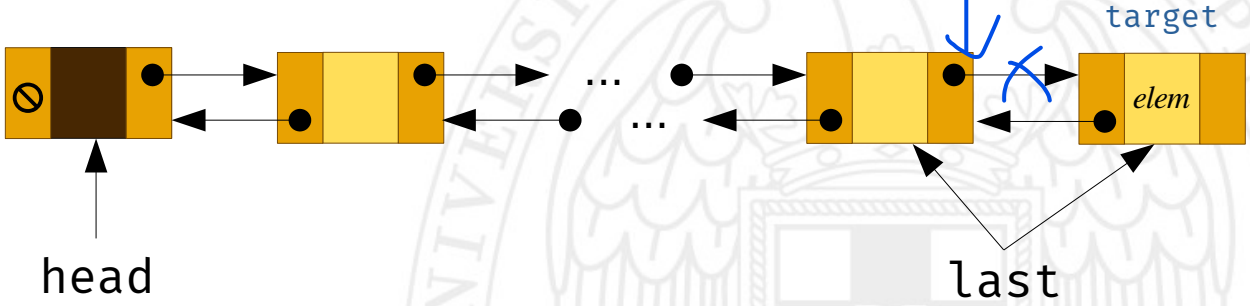
Last pasa a ser el nuevo nodo último



# 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;
}
```

Lo eliminamos




last ahora apunta a target->prev

# ¿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.

En las funciones de push y pop debemos de actualizar el número de nodos de la lista.

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

Push back y pop back, push front y pop front ocurre lo mismo.

Este atributo last no lo utilizaremos mucho en teoría en las siguientes diapositivas.