ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

Listas enlazadas circulares

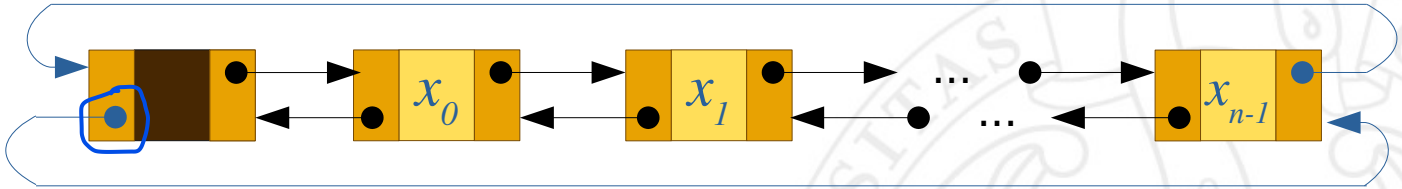
Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid

Listas doblemente enlazadas circulares

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

Último nodo de la secuencia apunta al primero y viceversa

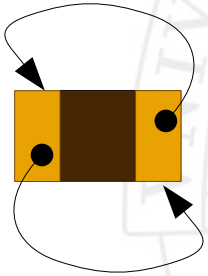
Ya no van a ser nulos



Vemos que no hay ninguno que tenga puntero a nullptr

Esto tiene la misma utilidad que, por ejemplo, añadir un nodo fantasma: No es nada más ni nada menos que simplificar algunos de nuestros métodos.

Si la lista es vacía, los punteros next y prev se apuntan a si mismo.



Podemos iterar sobre la lista tantas veces como nosotros queramos

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) {
   ...  cuando terminabamos de iterar sobre nodos
   current = current→next;
}
```

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

aqui acaba cuando lleguemos al nodo fantasma

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;
};
```

Nos ha durado bien poco este chorro lambe-bicho

Esto lo tenemos que mantener para mejorar el coste de la función size() de las listas de nodos.

Eliminar `*last`

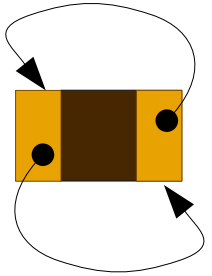
Creación de una lista

inicializamos el número de elementos a 0

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

head apunta a ese nodo

tanto next como prev apuntan a este nodo



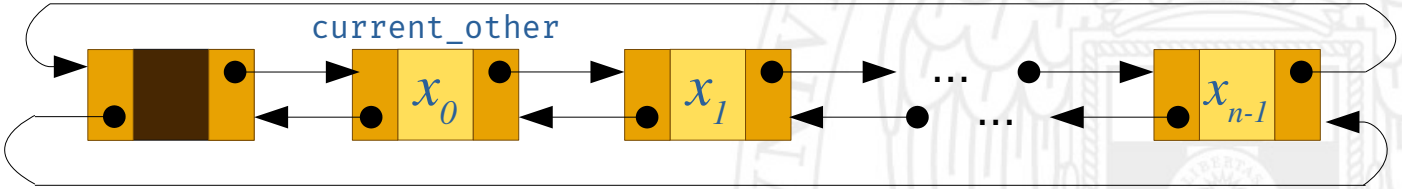
Copia de una cadena de nodos

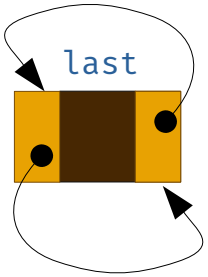
```
ListLinkedDouble(const ListLinkedDouble &other): ListLinkedDouble() {
  Node *current_other = other.head→next;
  Node *last = head;

  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;
}
```

Necesitamos una variable local last,no quiere decir que utilicemos el last anterior.

previo al añadido es el anterior

sucesor del último añadido es la cabeza

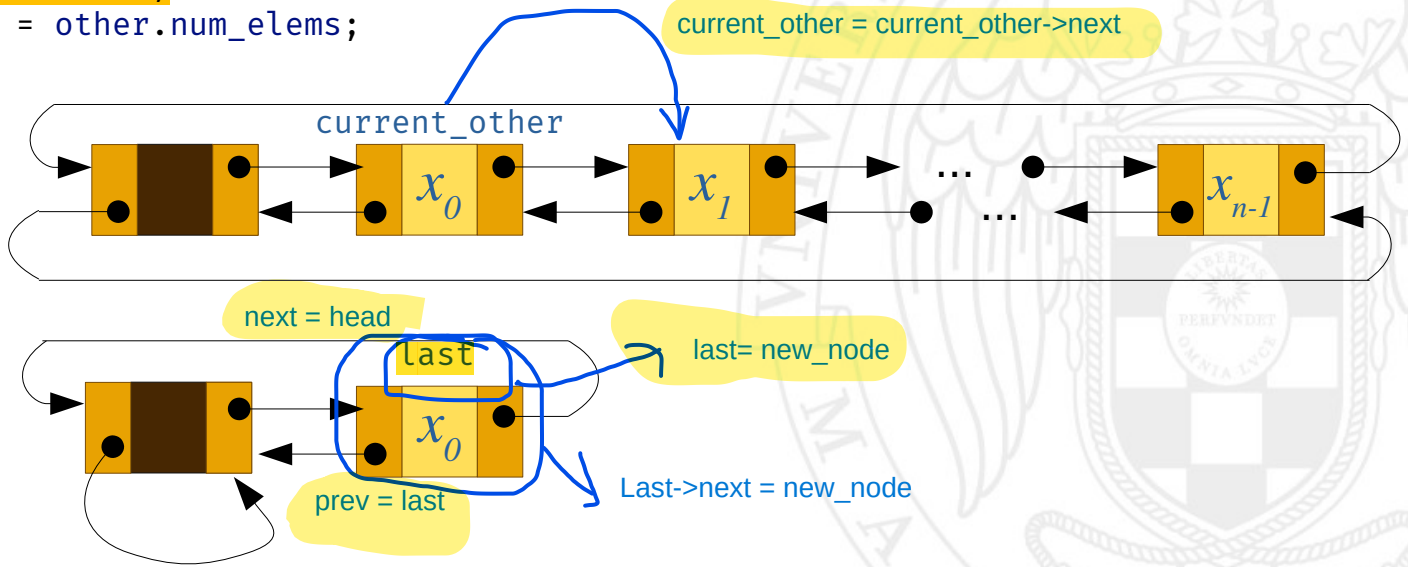




Node*last = head;

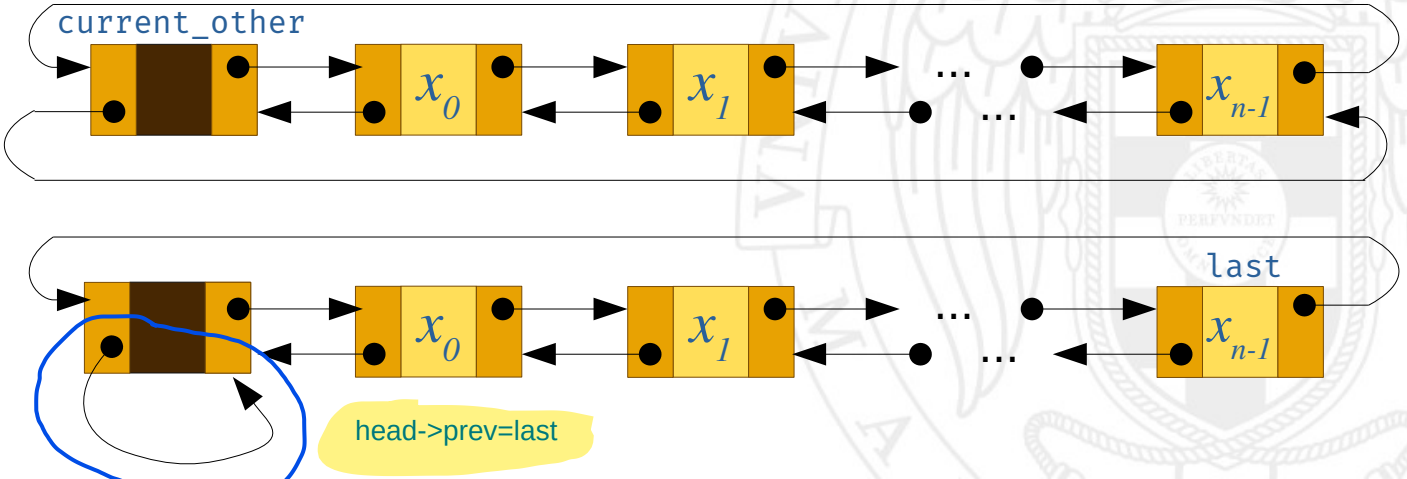
Copia de una cadena de nodos

```
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

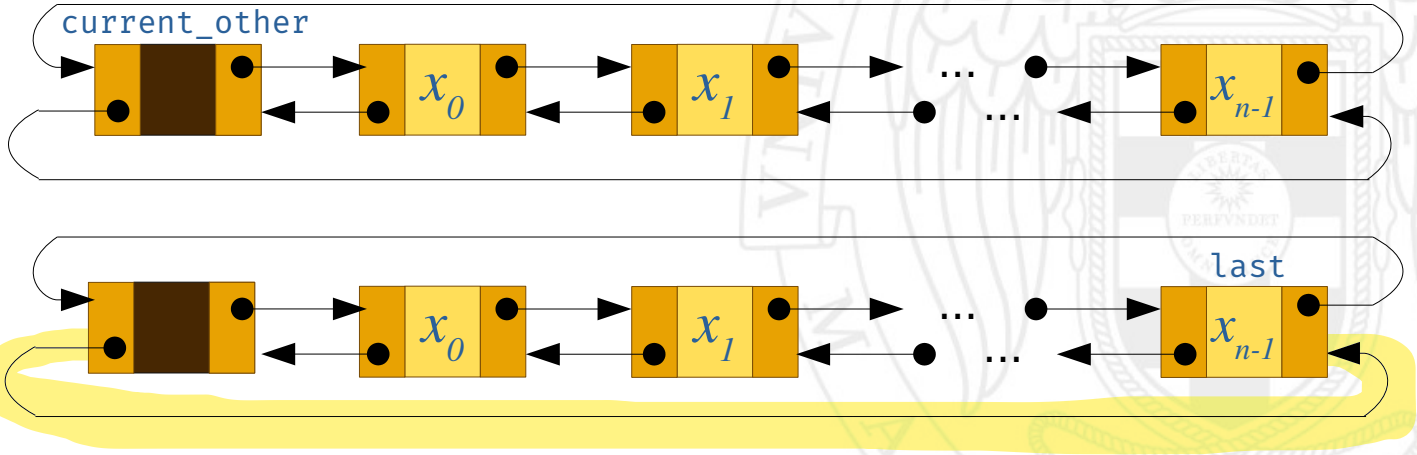
```
ListLinkedDouble(const ListLinkedDouble &other): ListLinkedDouble() {
  Node *current_other = other.head→next;
  Node *last = head;

  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;
}
```

El número de elementos no cambia porque no modificamos la lista, la copiamos.



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++;
}
```

Al principio de la lista.

el previo del siguiente nodo sea el nuevo nodo

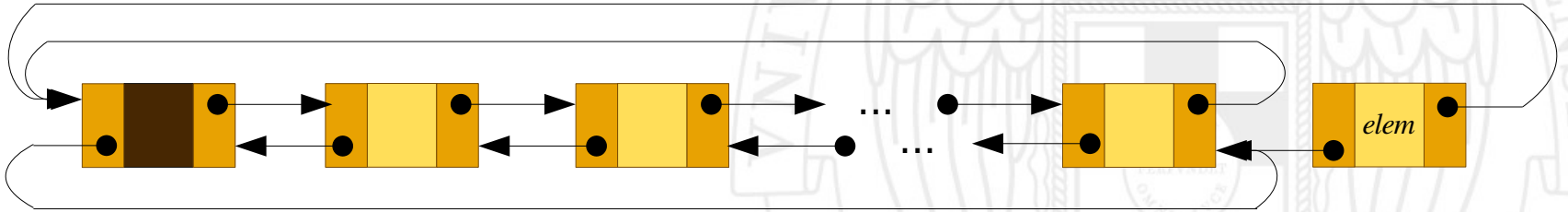
el siguiente del nodo fantasma sea el nuevo nodo

No nos olvidamos de actualizar el número de elementos.

añadir elemento al final de la lista.

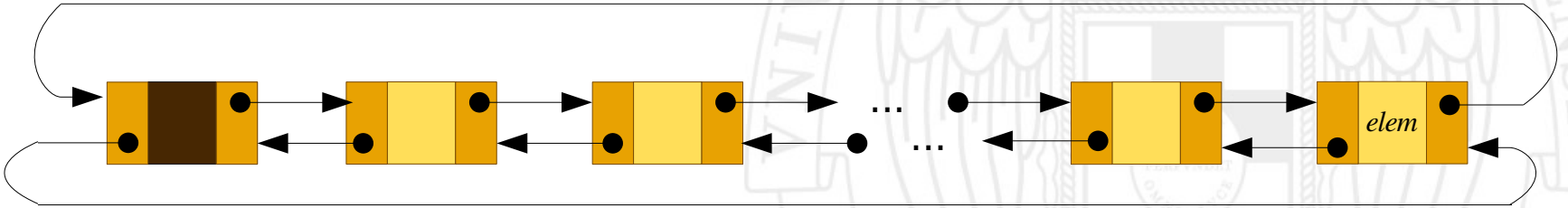
next

prev (es el prev de la cabeza.)



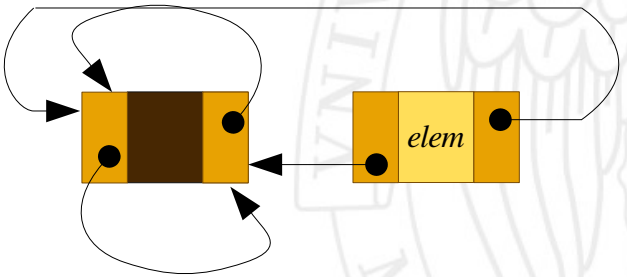
Añadir elementos

```
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++;
}
```



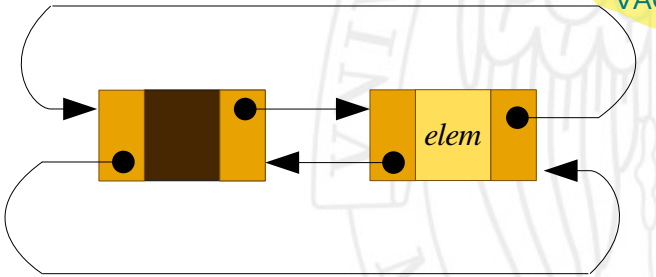
Añadir elementos

```
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++;
}
```



NO HACE FALTA DISTINGUIR EL CASO DE LA LISTA VACÍA. TRATAMIENTO MÁS UNIFORME.

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--;
}
```

Al principio

Apunta al siguiente del nodo fantasma.

Nodo fantasma apunta al siguiente de su siguiente.

Ese mismo ahora su previo apunta al fantasma.

Al final

Coste de las operaciones

Hemos ganado simplicidad en el código

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*