ESTRUCTURAS DE DATOS

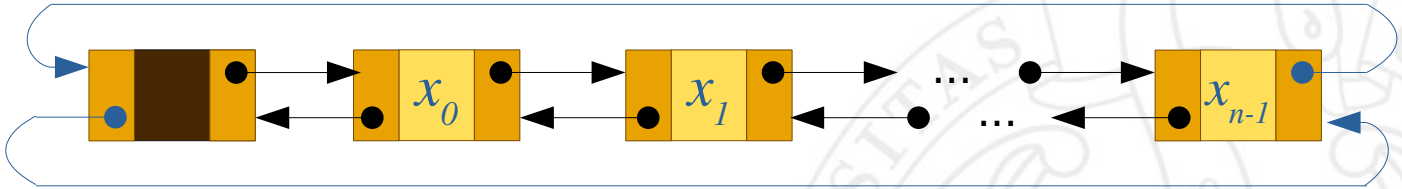
TIPOS ABSTRACTOS DE DATOS LINEALES

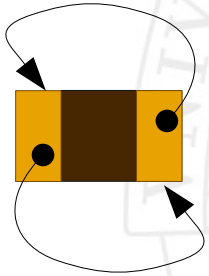
Listas enlazadas circulares

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;
}
```

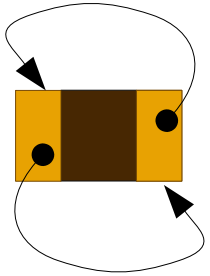
Eliminamos atributo `last`

```
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`

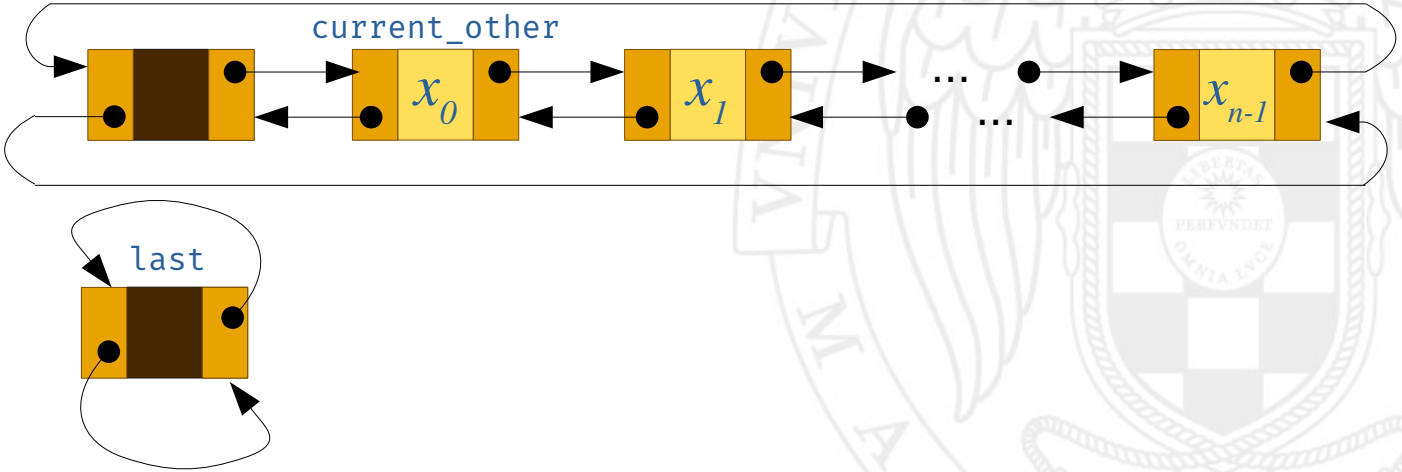
Creación de una lista

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



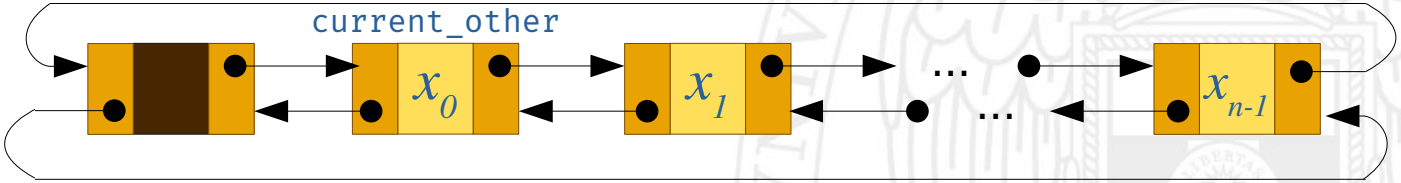
Copia de una cadena de nodos

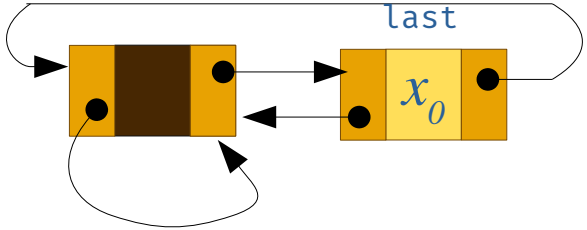
```
ListLinkedDouble(const ListLinkedDouble &other): ListLinkedDouble() {
  Node *current_other = other.head→next;
  Node *last = head;

  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

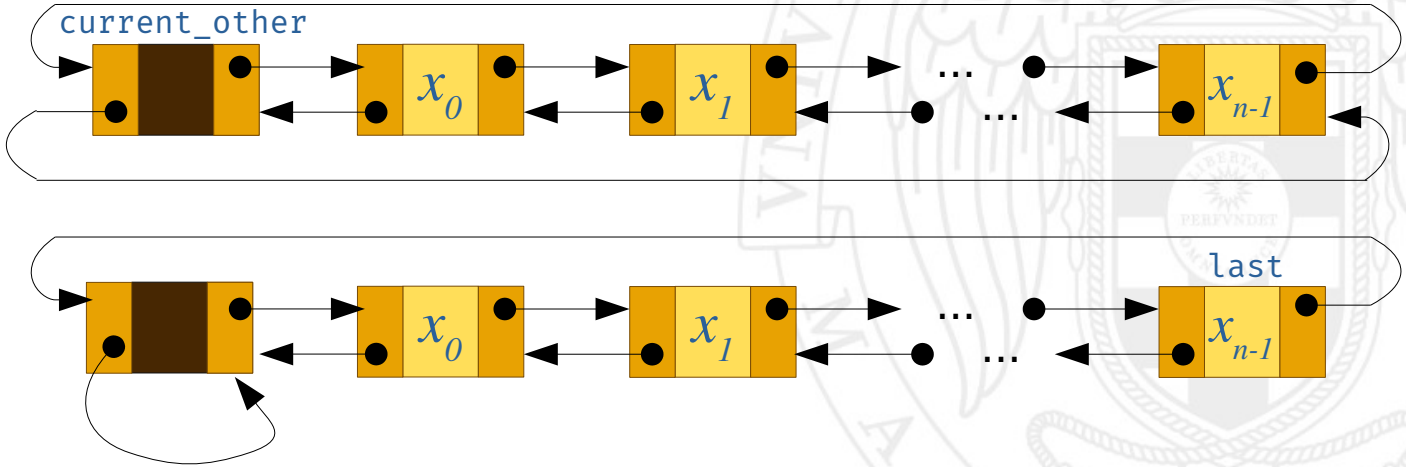
```
ListLinkedDouble(const ListLinkedDouble &other): ListLinkedDouble() {
  Node *current_other = other.head→next;
  Node *last = head;

  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;
}
```





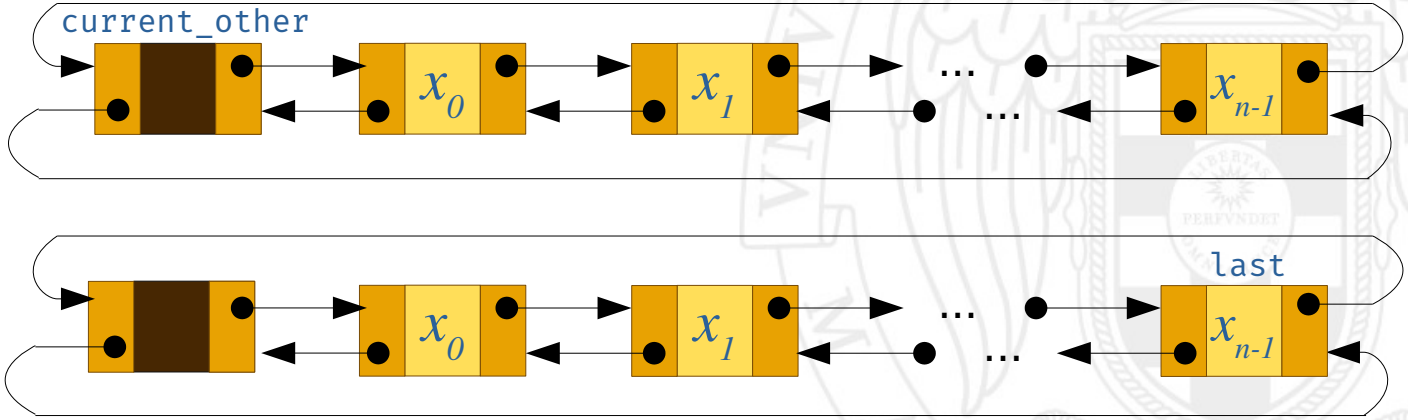
Copia de una cadena de nodos

```
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;
}
```



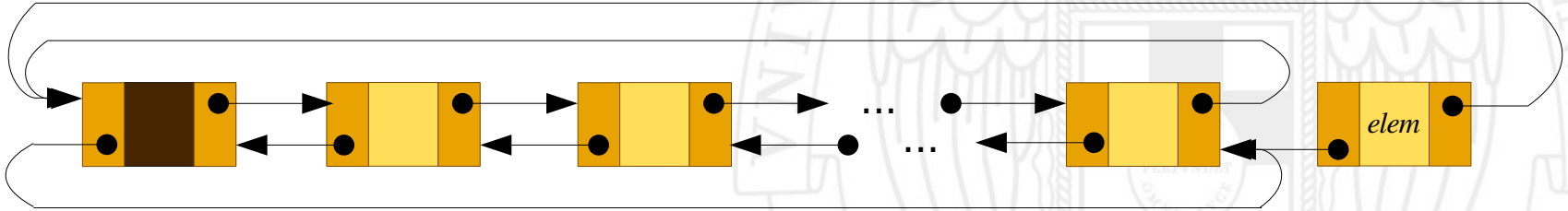
Copia de una cadena de nodos

```
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;
}
```



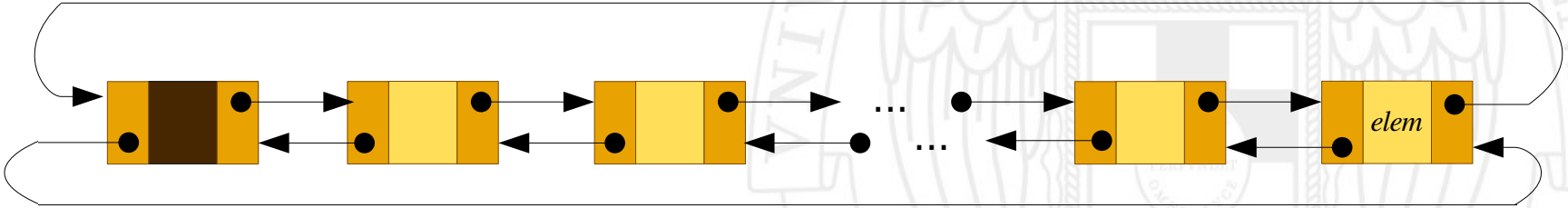
Añadir elementos

```
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++;
}
```



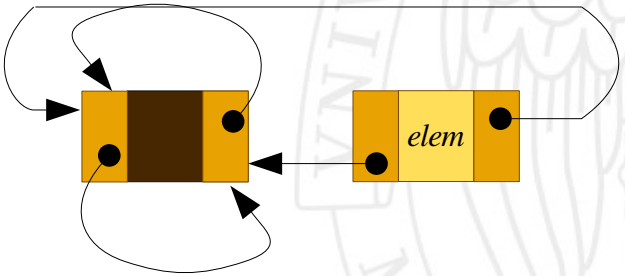
Añadir elementos

```
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++;
}
```



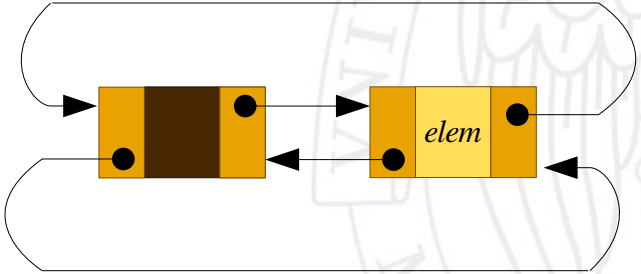
Añadir elementos

```
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*