

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

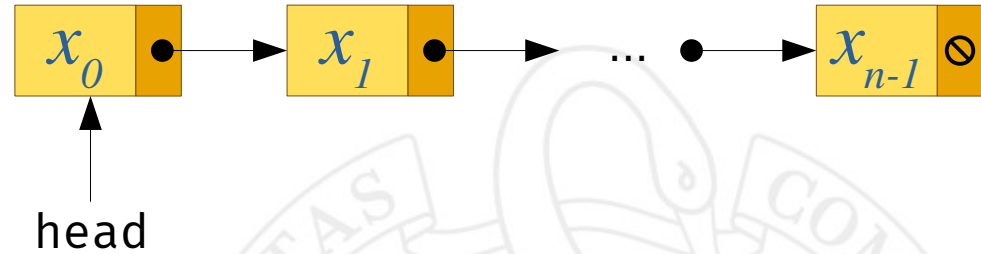
Nodos fantasma

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

Recordatorio

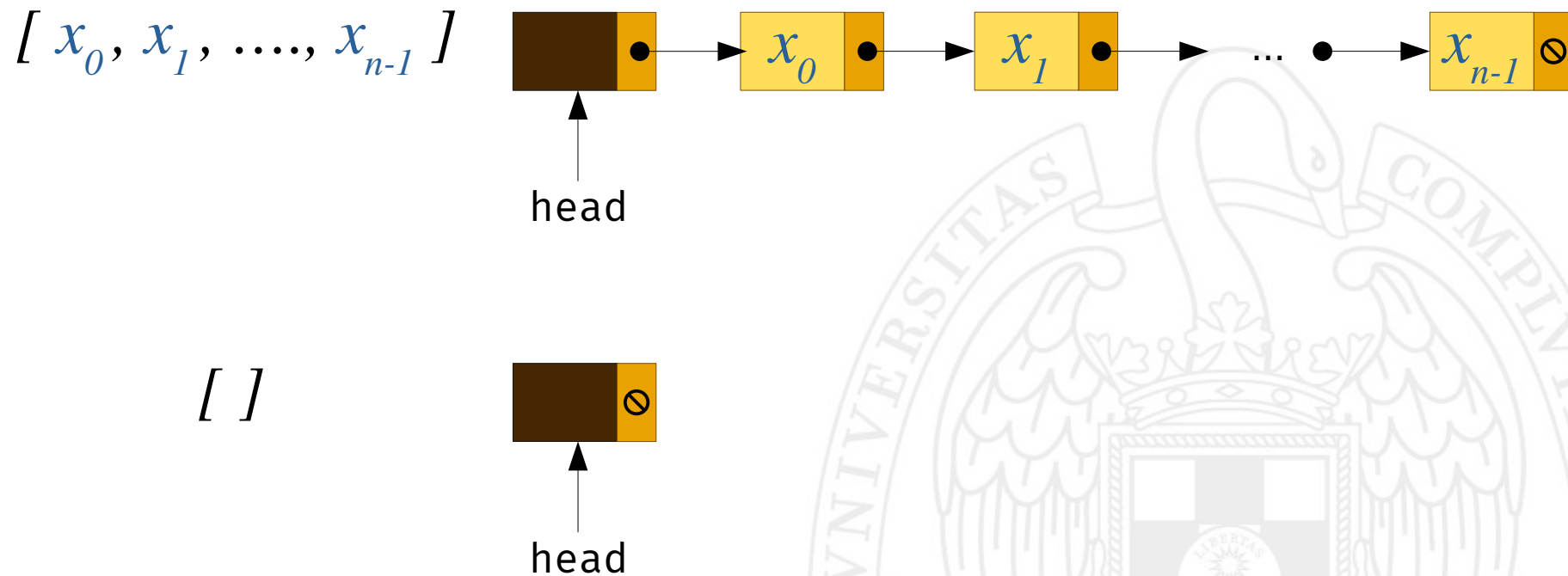
$[x_0, x_1, \dots, x_{n-1}]$

$[]$



head = nullptr

Introduciendo un nodo fantasma




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();

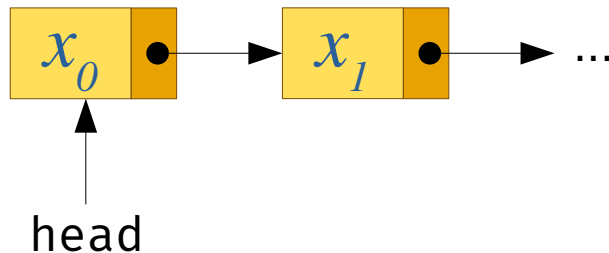
    void push_front(const std::string &elem);
    void push_back(const std::string &elem);
    void pop_front();
    void pop_back();
    int size() const;
    bool empty() const;
    const std::string & front() const;
    std::string & front();
    const std::string & back() const;
    std::string & back();
    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()

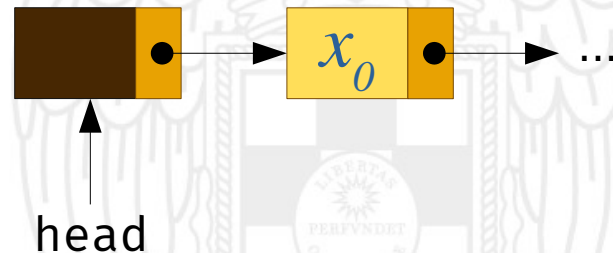
Antes

```
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();

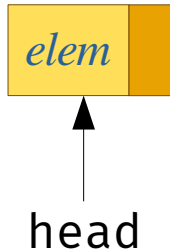
    void push_front(const std::string &elem);
    void push_back(const std::string &elem);
    void pop_front();
    void pop_back();
    int size() const;
    bool empty() const;
    const std::string & front() const;
    std::string & front();
    const std::string & back() const;
    std::string & back();
    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()

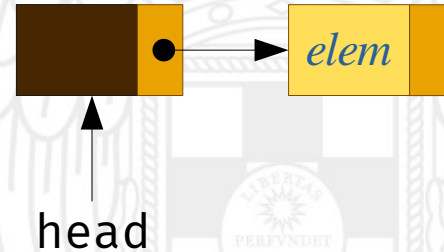
Antes

```
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

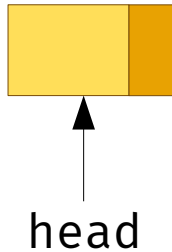
```
void push_back(const std::string &elem) {  
    Node *new_node = new Node { elem, nullptr };  
    last_node()→next = new_node;  
}
```



Cambios en pop_back()

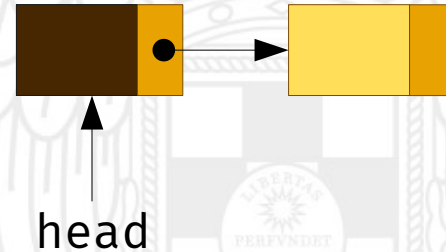
Antes

```
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.

