


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()*** $\rightarrow L: \text{List}$
- **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*) $\rightarrow \text{tam: int}$.
 - Comprobar si la lista es vacía ***empty***(*L: List*) $\rightarrow b: \text{bool}$.
 - Acceder al primer elemento de la lista ***front***(*L: List*) $\rightarrow e: \text{elem}$.
 - Acceder al último elemento de la lista ***back***(*L: List*) $\rightarrow e: \text{elem}$.
 - Acceder a un elemento que ocupa una posición determinada ***at***(*idx: int, L: List*) $\rightarrow e: \text{elem}$.

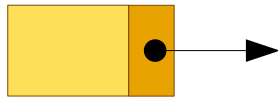
¿Qué es una lista enlazada?

- 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

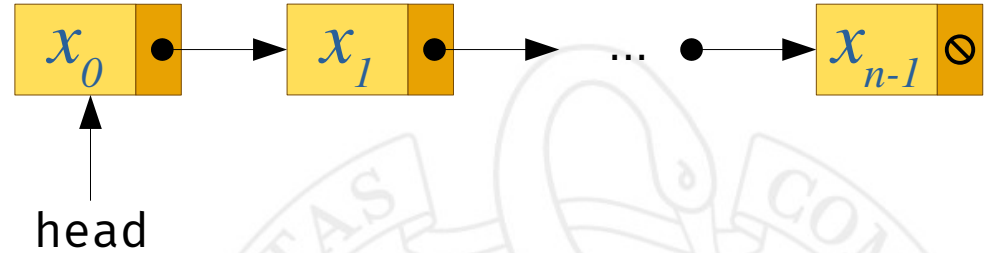
```
struct Node {  
    std::string value;  
    Node *next;  
};
```

```
Node *tres = new Node { "Tres", nullptr };  
Node *dos  = new Node { "Dos", tres };  
Node *uno  = new Node { "Uno", dos };
```



El TAD Lista mediante listas enlazadas

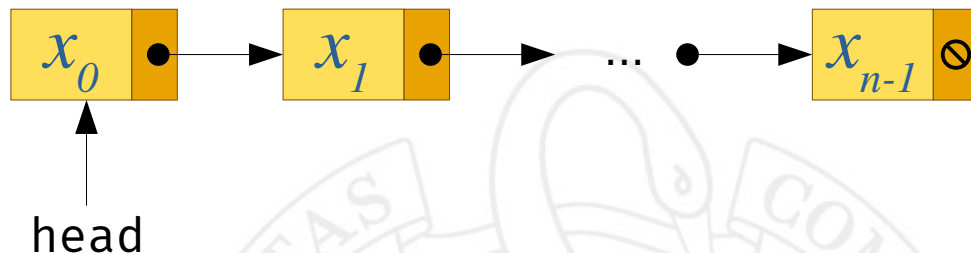
$[x_0, x_1, \dots, x_{n-1}]$



```
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:

$$f(x) = [x.\text{head} \rightarrow \text{value}, x.\text{head} \rightarrow \text{next} \rightarrow \text{value}, x.\text{head} \rightarrow \text{next} \rightarrow \text{next} \rightarrow \text{value}, \dots]$$

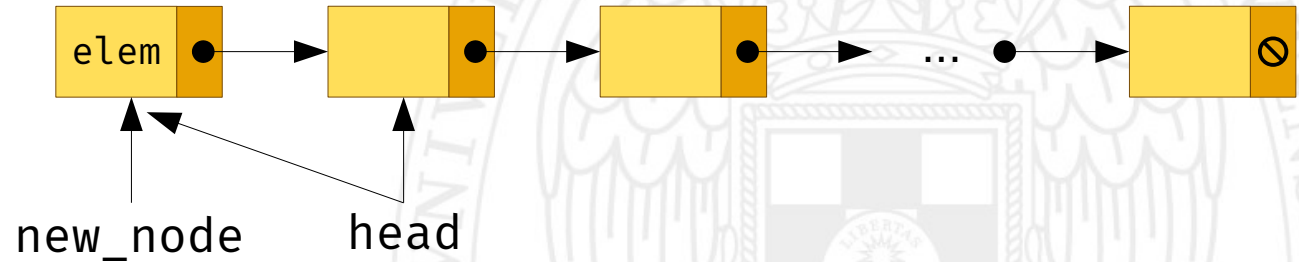
Inicializar lista

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



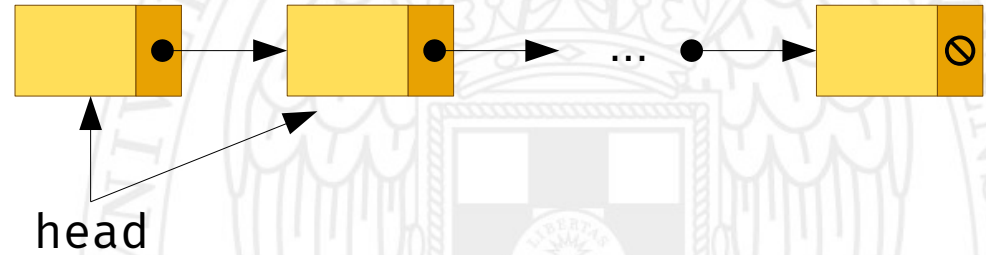
Añadir un elemento al principio de la lista

```
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

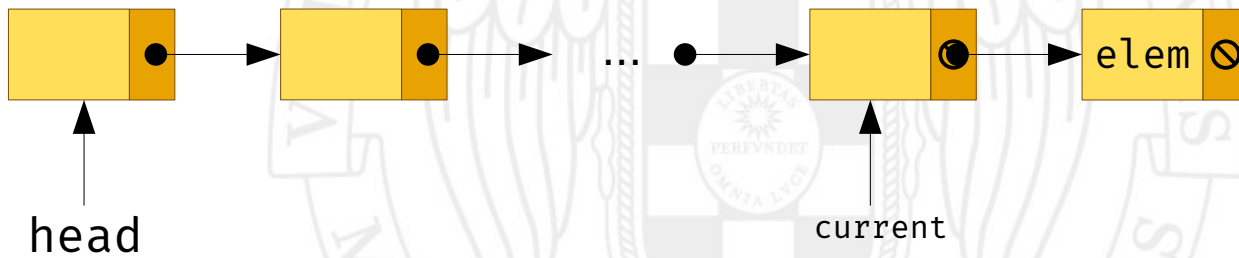
```
class ListLinkedSingle {  
public:  
    ...  
    void pop_front() {  
        assert (head ≠ nullptr);  
        Node *old_head = head;  
        head = head→next;  
        delete old_head;  
    }  
  
private:  
    struct Node { ... };  
    Node *head;  
};
```



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;  
    }  
}
```



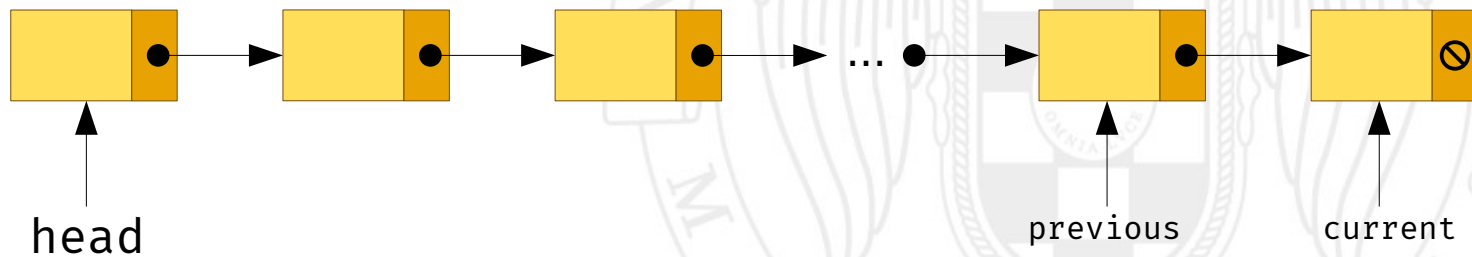
Refactorizando: obtener el último nodo

```
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;
    }
}
```

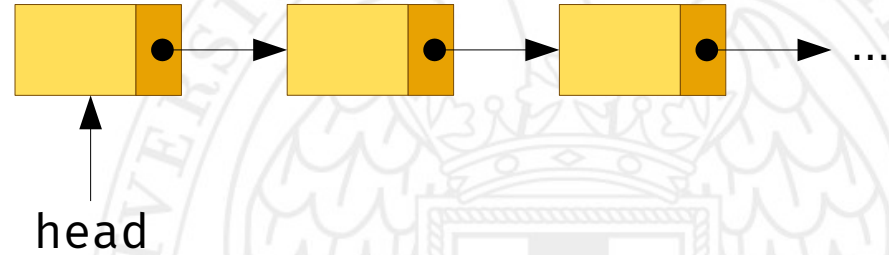
Eliminar un elemento del final de la lista

```
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;  
};
```



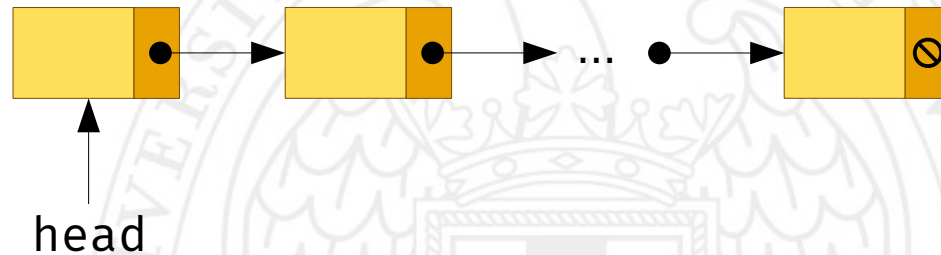
Acceder al último elemento de la lista

```
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;  
    }  
}
```

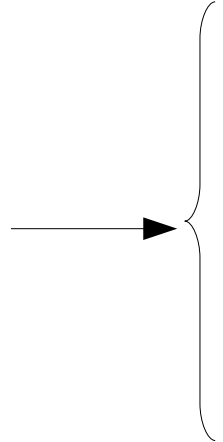
```
std::string & at(int index) { ... }
```

```
private:
```

```
    struct Node { ... };  
    Node *head;
```

```
    Node *last_node() const;
```

```
    Node *nth_node(int n) const;  
};
```



A large curly brace on the right side of the code blocks groups the implementation of the `nth_node` method. An arrow points from the `Node *nth_node(int n) const;` declaration in the private section to the implementation block.

```
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 ListLinkedListSingle::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;  
    }  
}
```

