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

VAMOS A VER OTRA FORMA DE IMPLEMENTAR EL TAD LISTA SIN UTILIZAR ARRAYS. LAS LISTAS

Recordatorio: operaciones del TAD Lista

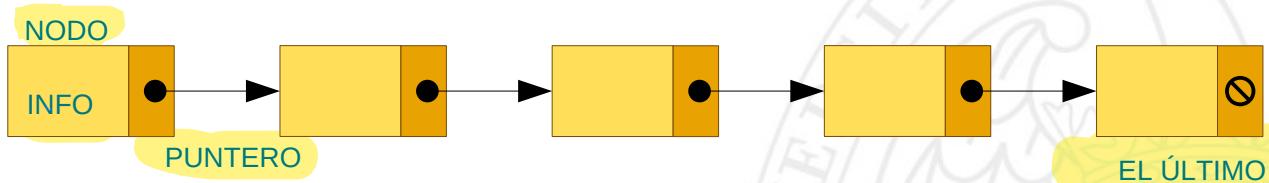
Esto aparece explicado en las diapositivas anteriores.

- **Constructoras:**
  - Crear una lista vacía: ***create_empty***() → *L: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*) → *tam: int.*
  - Comprobar si la lista es vacía ***empty***(*L: List*) → *b: bool.*
  - Acceder al primer elemento de la lista ***front***(*L: List*) → *e: elem.*
  - Acceder al último elemento de la lista ***back***(*L: List*) → *e: elem.*
  - Acceder a un elemento que ocupa una posición determinada ***at***(*idx: int, L: List*) → *e: 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**.

Depende de lo que nosotros queramos almacenar en la lista. Puede ser una cadena, puede ser un número...



EL ÚLTIMO NO TIENE NODO SUCESOR.

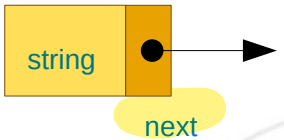
Se les llaman listas enlazadas simples porque cada nodo SÓLAMENTE TIENE UN PUNTERO QUE APUNTA AL SIGUIENTE NODO DE LA LISTA. VEREMOS EN OTRAS SEMANAS LISTAS DOBLES (CADA NODO TIENE DOS PUNTEROS:UNO AL SIGUIENTE Y OTRO AL ANTERIOR)

Definición de un nodo

Los nodos en C++ los definimos mediante REGISTROS O STRUCTS, con los dos campos de los que hablamos en la anterior diapositiva. El valor y el puntero al siguiente nodo.

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

información arbitraria

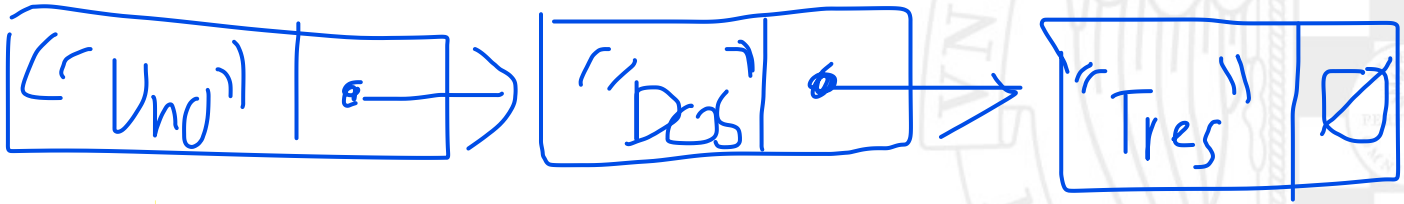


- Cuando un nodo no tiene sucesor, su campo `next` contiene el puntero nulo (`nullptr` en C++).

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 };
```

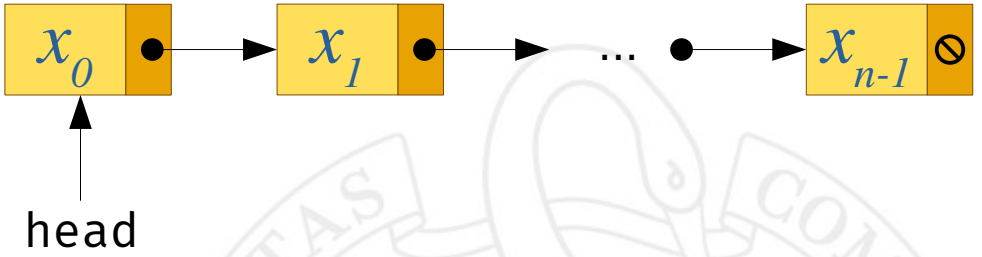
3 nodos



El TAD Lista mediante listas enlazadas

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

Último nodo sin sucesor



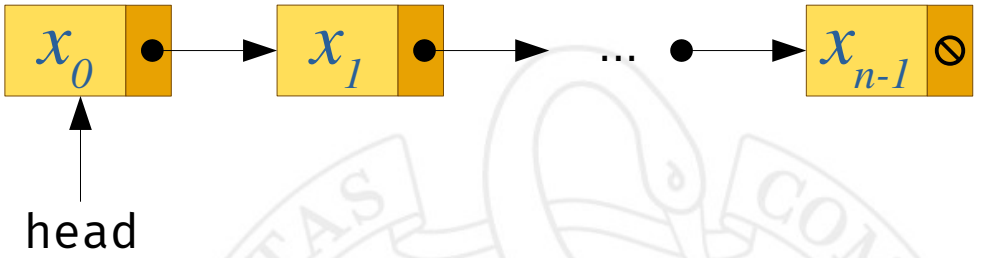

```
class ListLinkedSingle {
public:
    ...
private:
    struct Node { … };
    Node *head;
};
```

Dentro de la zona privada de la clase. Sólo manipulamos nodos dentro de esta clase.

Puntero al primer nodo de la lista. Si lo que tuviéramos fuera una lista enlazada VACÍA head apuntaría a nullptr

El TAD Lista mediante listas enlazadas

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



- Invariante de representación:

  $I(x) = true$ Puede tener nullptr o un nodo

- Función de abstracción:

  $f(x) = [\ x.head \rightarrow value,\ x.head \rightarrow next \rightarrow value,\ x.head \rightarrow next \rightarrow next \rightarrow value,\ \ldots]$

Inicializar lista

```
class ListLinkedSingle {
public:
  ListLinkedSingle(): head(nullptr) { }
  …

private:
  struct Node { … };
  Node *head;
};
```

inicializamos a nullptr

Luego la inicialización tiene coste CONSTANTE.

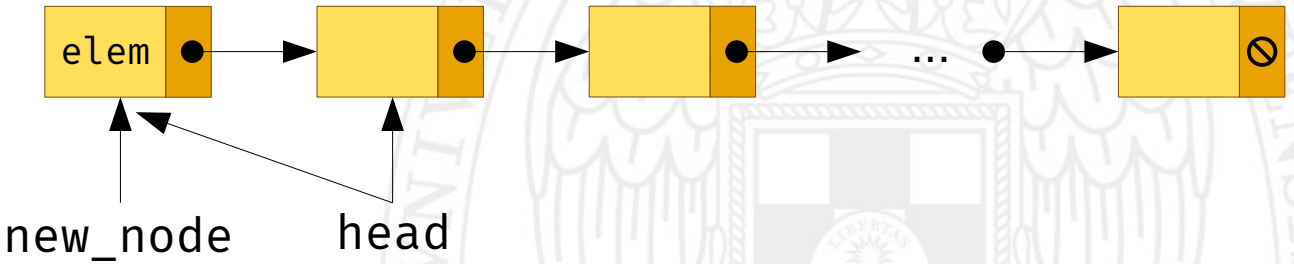
Añadir un elemento al principio de la lista

```
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;
};
```

Este nuevo nodo es el primero de la lista

head apunta al nuevo nodo

TODO ESTO TIENE COSTE CONSTANTE



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;
};
```

COMPROBAMOS QUE LA CABEZA APUNTA ALGÚN NODO

GUARDAMOS LA CABEZA DE LA LISTA

COMO QUEREMOS ELIMINAR EL PRIMER ELEMENTO, LA CABEZA VA A SER EL SEGUNDO, ACTUALIZAMOS HEAD, PARA QUE APUNTE HEAD_NEXT

LIBERAMOS EL NODO CORRESPONDIENTE AL PRIMER ELEMENTO

TODO ESTO TIENE COSTE CONSTANTE

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;
  }
}
```

COSTE LINEAL RESPECTO AL NÚMERO DE ELEMENTOS DE LA LISTA.

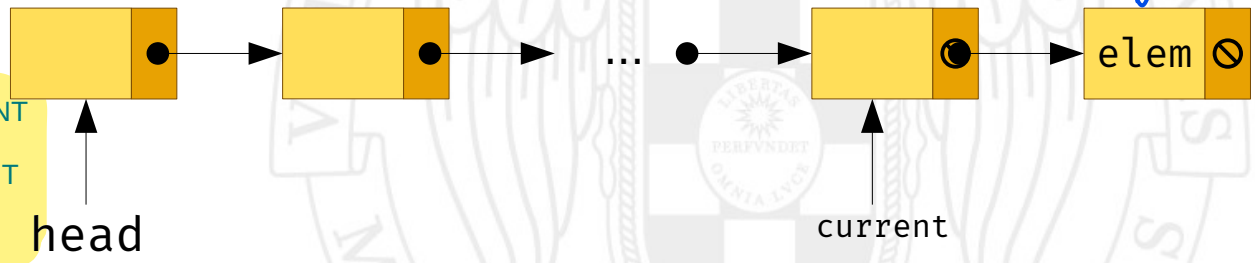
CREAMOS EL NUEVO NODO, EL DEL FINAL

HASTA ENTONCES NO TENÍAMOS ELEMENTOS EN LA LISTA.
POR TANTO NUEVA CABEZA ES ESE NUEVO NODO

APUNTA AL PRIMER NODO

Y AVANZAMOS CURRENT HASTA QUE LLEGUE AL NODO DEL FINAL

CUANDO SALIMOS DEL BUCLE, CURRENT APUNTA AL ÚLTIMO NODO DE LA LISTA, SOLO ACTUALIZAMOS NEX DE CURRENT AL NODO CREADO AL PRINCIPIO.



Refactorizando: obtener el último nodo

Esto es para simplificar el push_back de la diapositiva anterior

Me devuelve el último nodo de una lista

```
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;
  }
}
```

esto hace lo mismo que hemos hecho antes en el bucle de coste lineal pero lo hacemos en una función aparte

Eliminar un elemento del final de la lista

Más complicado aun

```
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;
  }
}
```

Suponemos que la lista no es vacía

Si la lista solo tiene un elemento

nos tenemos que cargar ese elemento

la cabeza apunta a algo null

si hay + de 1 elemento

apunta a la cabeza

apunta al nodo siguiente a la cabeza.

hasta que su siguiente sea null (sea el último)

avanzamos las dos simultaneamente.

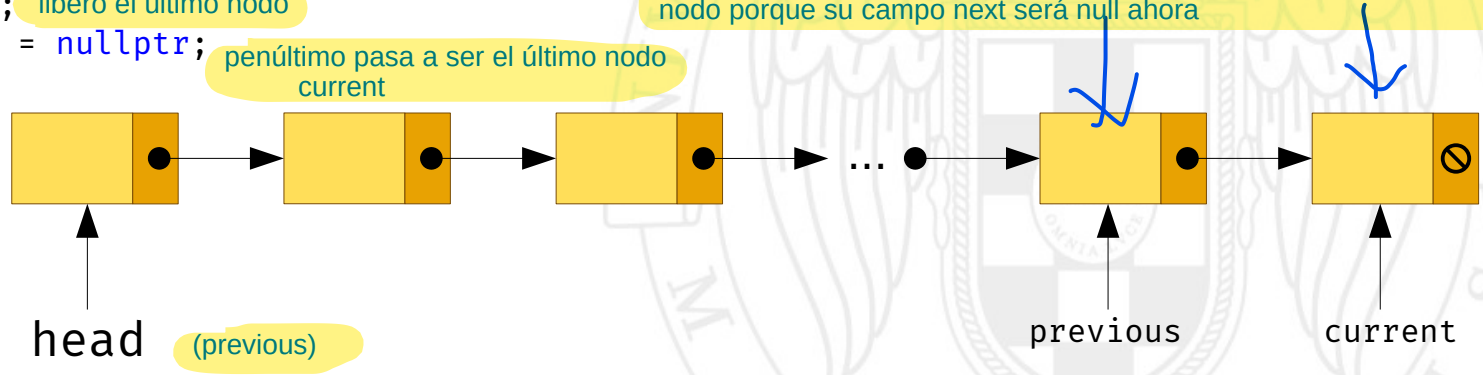
libero el último nodo

penúltimo pasa a ser el último nodo

current

coste lineal respecto al número de elementos de la lista

tenemos que acceder también a este penúltimo Es el que queremos borrar nodo porque su campo next será null ahora

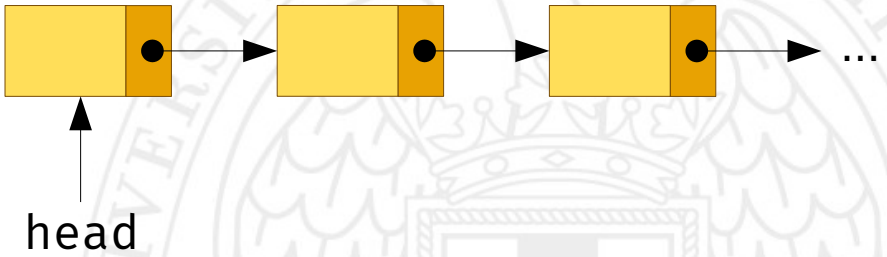


Acceder al primer elemento de la lista

```
class ListLinkedSingle {
public:
  ...
  const std::string & front() const {
    assert (head ≠ nullptr);
    return head→value;
  }

  std::string & front() { … }

private:
  struct Node { … };
  Node *head;
};
```

por referencia

accedemos al valor que esta dentro de la cabeza. Ya que la cabeza siempre apunta al primer elemento de la lista



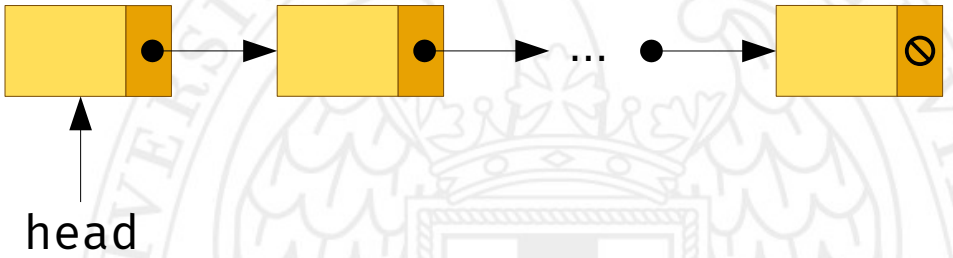
coste constante.

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;
};
```

Método privado que me devuelve el último nodo. Solo tenemos que acceder a ese nodo devuelto.

COSTE LINEAL



Acceder al elemento *n*-ésimo de la lista

Posición arbitraria 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;
};
```

posición index

devuelve puntero al nodo

en el caso de que el nodo no exista

coste lineal con respecto al indice

```
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;
}
```

recorrido de izquierda a derecha comenzando por la cabeza

contador del indice del nodo

Es parecido al at de las listas con arrays que hemos visto anteriormente.

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;
}
```

recorrer la lista de principio a fin y contar el número de nodos que nos encontramos

Mostrar una lista por pantalla

```
void ListLinkedSingle::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 << "]";
}
```

no tendriamos nodos

imprimo el valor de ese nodo

y hago que current apunte al siguiente de la cabeza

recorremos el resto de nodos

imprimir el corchete de cierrte

Destrucción de una lista

```
class ListLinkedSingle {
public:
   ...
  ~ListLinkedSingle() {
    delete_list(head);
  }

private:
   ...
  void delete_list(Node *start_node);
}
```

libera todos los nodos

elimina este nodo y todos los que le siguen de manera recursiva

```
void ListLinkedSingle::delete_list(Node *start_node) {
  if (start_node ≠ nullptr) {
    delete_list(start_node→next);
    delete start_node;
  }
}
```

si el puntero es null no hacemos nada

si es distinto de null borramos el siguiente.

se libera el nodo actual sobre el cual se ha hecho la llamada