



# Tema 2

## Diseño e implementación de TADs lineales

---

Estructuras de datos  
Facultad de Informática - Universidad Complutense de Madrid

Transparencias de los Profs.  
Mercedes Gómez Albarrán y José Luis Sierra Rodríguez



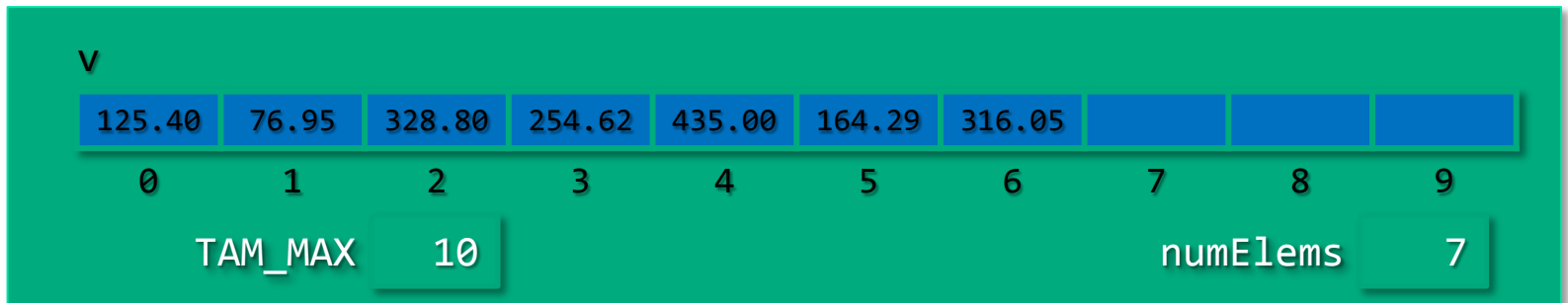
## Estructuras de datos lineales

---

- Estructura de datos = estrategia de almacenamiento en memoria de la información que se desea guardar
- Estructuras de datos lineales = estrategias para guardar en memoria colecciones lineales de datos (hay un primer elemento, un segundo, ..., un último elemento)
- Dos aproximaciones básicas:
  - Los elementos almacenados de forma consecutiva en memoria
    - Arrays (estáticos, dinámicos)
  - Los elementos dispersos en memoria manteniendo enlaces entre ellos
    - Listas enlazadas (simples, dobles, con nodo cabecera, circulares)
- A la hora de implementar el TAD la elección de la estructura de datos influirá en el coste de las operaciones del TAD

## Implementaciones de TADs basadas en arrays estáticos

- Las implementaciones de TADs que quieran hacer uso de esta estructura de datos normalmente manejarán:
  - Un array estático (componentes del tipo T del TAD)
  - La dimensión de ese array (número de elementos que puede almacenar como máximo): constante
  - Un contador con el tamaño “real” del array (número de elementos realmente almacenados en cada momento)



```
const int TAM_MAX = 10;  
T v[TAM_MAX];  
unsigned int numElems;
```



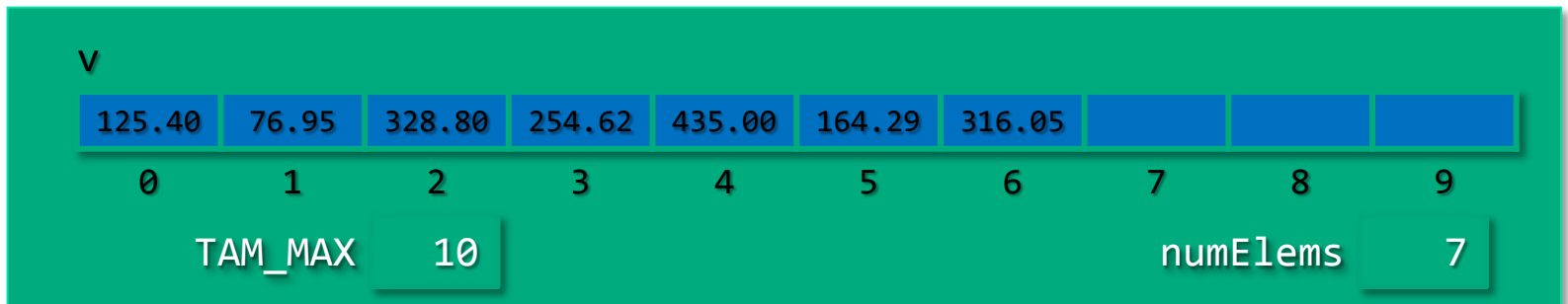
## Implementaciones de TADs basadas en arrays estáticos

---

- Características:
  - El array no puede cambiar de tamaño → limitación en el número de elementos que se podrán almacenar
  - Convenio: las posiciones ocupadas por los elementos se condensan al principio, el resto de componentes del array están “vacías”

## Implementaciones de TADs basadas en arrays estáticos

- Operaciones típicas para el manejo de la estructura de datos:  
¿Cómo se...
  - Crea una estructura vacía?
  - Añade un elemento  $e$  por el final de la estructura?
  - Inserta un elemento  $e$  en posición intermedia  $pos$  de la estructura?
  - Elimina un elemento por el final de la estructura?
  - Elimina un elemento en posición intermedia  $pos$  de la estructura?





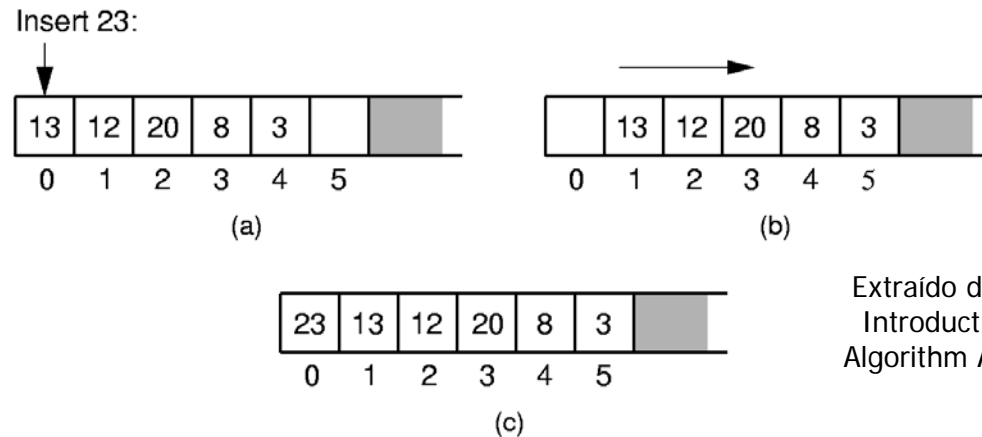
## Implementaciones de TADs basadas en arrays estáticos

---

- Operaciones típicas para el manejo de la estructura de datos:  
¿Cómo se...
  - Crea una estructura vacía?  
`numElems = 0;`
  - Añade un elemento  $e$  por el final de la estructura?
    - Operación parcial: sólo es posible si no está llena la estructura  
`v[numElems] = e;`  
`numElems++;`

# Implementaciones de TADs basadas en arrays estáticos

- Inserta un elemento  $e$  en posición intermedia  $pos$  de la estructura?  
Posición intermedia = hay elementos a la derecha



Extraído de "Coursenotes: A Practical Introduction to Data Structures and Algorithm Analysis", Clifford A. Shaffer

- Operación parcial: sólo es posible si no está llena la estructura
- Desplazar a la derecha desde la posición  $pos$  para hacer hueco, colocar  $e$  en la posición "hueco" generada y aumentar en 1 el contador

```
for(int i = numElems; i > pos ; i--)
```

```
    v[i] = v[i-1];
```

```
v[pos] = e;
```

```
numElems++;
```

## Implementaciones de TADs basadas en arrays estáticos

- Operaciones típicas para el manejo de la estructura de datos:  
¿Cómo se...
  - Elimina un elemento por el final de la estructura?
    - Operación parcial: sólo es posible si no está vacía la estructura  
**numElems--;**
  - Elimina un elemento en posición intermedia *pos* de la estructura?

Posición intermedia = hay elementos a la derecha

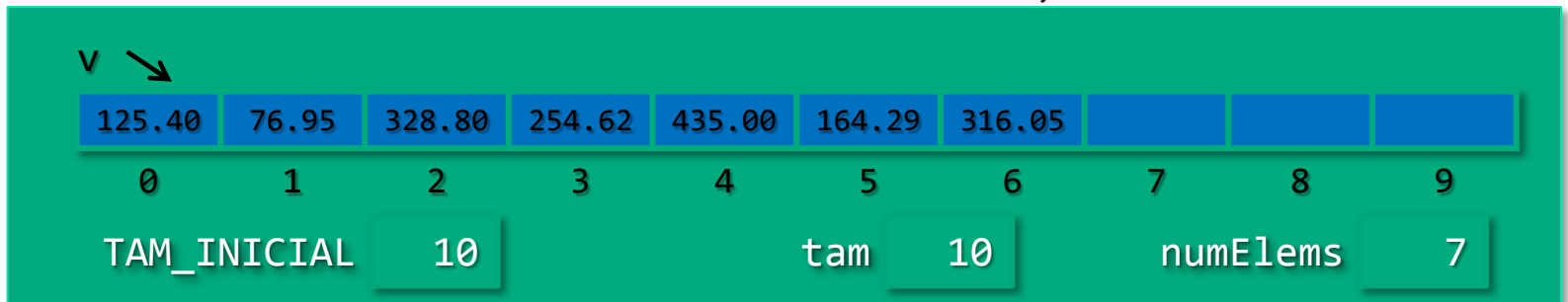
- Operación parcial: sólo es posible si no está vacía la estructura
- Desplazar (a la izquierda) los elementos a la derecha de *pos* para “machacar” dicha posición y disminuir en 1 el contador

```
for (int i = pos; i < numElems-1; i++)  
    v[i] = v[i+1];  
numElems--;
```



## Implementaciones de TADs basadas en arrays dinámicos

- Las implementaciones de TADs que quieran hacer uso de esta estructura de datos normalmente manejarán:
  - Puntero al array almacenado en memoria dinámica (puntero al tipo T de los elementos del TAD)
  - La dimensión inicial del array: constante
  - La dimensión del array (número de elementos que puede almacenar como máximo): variable
  - Un contador con el tamaño "real" del array (número de elementos realmente almacenados en cada momento)



```
const int TAM_INICIAL = 10;  
T *v;  
unsigned int tam;  
unsigned int numElems;
```



## Implementaciones de TADs basadas en arrays dinámicos

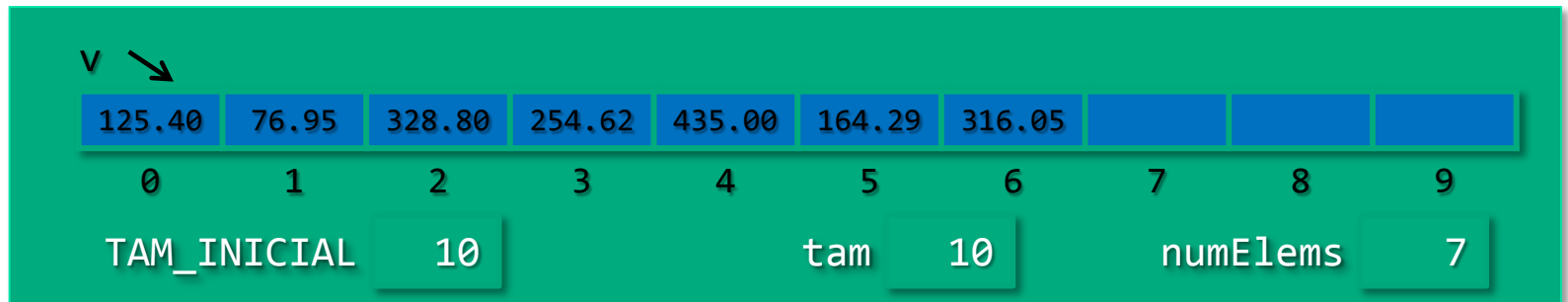
---

- Características:

- El array sí puede *cambiar de tamaño* → se sustituye el lleno por otro más amplio
  - `tam` inicialmente vale `TAM_INICIAL` pero puede aumentar de valor en ejecución y reservarse memoria acorde al nuevo tamaño
- Convenio: las posiciones ocupadas por los elementos se condensan al principio, el resto de componentes del array están “vacías”
- En el TAD:
  - Reserva de memoria en el constructor
  - Liberación de memoria en el destructor
  - Constructor de copia y operador de asignación para hacer copias profundas

## Implementaciones de TADs basadas en arrays dinámicos

- Operaciones típicas para el manejo de la estructura de datos:  
¿Cómo se...
  - Crea una estructura vacía?
  - Añade un elemento  $e$  por el final de la estructura?
  - Inserta un elemento  $e$  en posición intermedia  $pos$  de la estructura?
  - Elimina un elemento por el final de la estructura?
  - Elimina un elemento en posición intermedia  $pos$  de la estructura?
  - Destruye una estructura?



## Implementaciones de TADs basadas en arrays dinámicos

- Operaciones típicas para el manejo de la estructura de datos:  
¿Cómo se...

- Crea una estructura vacía?

```
v = new T[TAM_INICIAL]; tam = TAM_INICIAL; numElems = 0;
```

- Añade un elemento *e* por el final de la estructura?

```
v[numElems] = elem; // colocamos en 1er hueco
```

```
numElems++; // aumentamos el contador
```

```
if (numElems == tam) { // ampliamos el vector si lo hemos llenado
```

```
    T *viejo = v;
```

```
    tam *= 2; //(*)
```

```
    v = new T[tam];
```

```
    for (unsigned int i = 0; i < numElems; ++i)
```

```
        v[i] = viejo[i];
```

```
    delete []viejo;
```

```
}
```

(\*) Para que el coste amortizado se mantenga constante, duplicamos el tamaño del array

## Implementaciones de TADs basadas en arrays dinámicos

- Operaciones típicas para el manejo de la estructura de datos:  
¿Cómo se...

- Inserta un elemento  $e$  en posición intermedia  $pos$  de la estructura?

```
for(int i = numElems; i > pos ; i--) // hacemos hueco
    v[i] = v[i-1];
v[pos] = e; // colocamos en el hueco creado
numElems++; // aumentamos el contador
if (numElems == tam) { // ampliamos el vector si lo hemos llenado
    T *viejo = v;
    tam *= 2;
    v = new T[tam];
    for (unsigned int i = 0; i < numElems; ++i)
        v[i] = viejo[i];
    delete []viejo;
}
```



## Implementaciones de TADs basadas en arrays dinámicos

- Operaciones típicas para el manejo de la estructura de datos:  
¿Cómo se...
  - Elimina un elemento por el final de la estructura?
    - Operación parcial: sólo es posible si no está vacía la estructura  
`numElems--;`
  - Elimina un elemento en posición intermedia *pos* de la estructura?
    - Operación parcial: sólo es posible si no está vacía la estructura  
`for (int i = pos; i < numElems-1; i++)`  
`v[i] = v[i+1];`  
`numElems--;`
  - Destruye una estructura?  
`delete []v;`

## Implementaciones de TADs basadas en listas enlazadas simples

- Las listas enlazadas simples se basan en el uso de nodos (espacios de memoria dispersos e independientes) que se componen de una parte de datos (elemento) y una parte de enlace (puntero al siguiente nodo de la lista)
  - Listas unidireccionales: fácil acceso desde un nodo al nodo sucesor
- Las implementaciones de TADs que quieran hacer uso de esta estructura de datos, en el caso más básico, manejarán:
  - Puntero al nodo de la lista que contiene el primer elemento de la misma (de tipo T, el tipo de los elementos del TAD)



```
class Nodo {  
    T _elem;  
    Nodo *_sig;  
    //...  
};  
Nodo *ini;
```

# Implementaciones de TADs basadas en listas enlazadas simples

- Características:

- En el TAD:

- El constructor dejará el objeto en disposición de ser usado
- Liberación de memoria en el destructor
- Constructor de copia y operador de asignación para hacer copias profundas

- En este curso implementaremos la clase `Nodo` como una clase interna (privada) del TAD

```
class Nodo {  
    public:  
        Nodo() : _sig(NULL) {}  
        Nodo(const T &elem) : _elem(elem), _sig(NULL) {}  
        Nodo(const T &elem, Nodo *sig) :  
            _elem(elem), _sig(sig) {}  
  
        T _elem;  
        Nodo *_sig;  
};
```





## Implementaciones de TADs basadas en listas enlazadas simples

---

- Operaciones típicas para el manejo de la estructura de datos:  
¿Cómo se...
  - Crea una estructura vacía?
  - Añade un elemento  $e$  por el principio de la estructura?
  - Inserta un elemento  $e$  tras otro elemento de la estructura?
  - Elimina un elemento por el principio de la estructura?
  - Elimina un elemento que tiene predecesor?
  - Destruye una estructura?

## Implementaciones de TADs basadas en listas enlazadas simples

- Operaciones típicas para el manejo de la estructura de datos:  
¿Cómo se...
  - Crea una estructura vacía?  
`ini = NULL;`
  - Inserta...
    1. Obtener un nodo y almacenar el valor en la parte de datos
    2. Conectar el nuevo nodo con la lista
  - Añade un elemento  $e$  por el principio de la estructura?  
`Nodo *nuevo = new Nodo(e, ini);`  
`ini = nuevo;`
  - Inserta un elemento  $e$  tras otro elemento de la estructura (apuntado por  $p$ )?  
`Nodo *nuevo = new Nodo(e, p->_sig);`  
`p->_sig = nuevo;`

## Implementaciones de TADs basadas en listas enlazadas simples

- Operaciones típicas para el manejo de la estructura de datos:  
¿Cómo se...

- Elimina...

Operación parcial: sólo es posible si no está vacía la estructura

1. Desconectar el nodo a eliminar de la lista y reconstruir las conexiones del resto apropiadamente
2. Destruir el nodo devolviendo su memoria al heap

- Elimina un elemento por el principio de la estructura?

```
Nodo *aBorrar = ini;
```

```
ini = ini->_sig;
```

```
delete aBorrar;
```

- Elimina un elemento que tiene predecesor (apuntado por  $p$ )?

```
Nodo *aBorrar = p->_sig;
```

```
p->_sig = p->_sig->_sig;
```

```
delete aBorrar;
```



## Implementaciones de TADs basadas en listas enlazadas simples

---

- Operaciones típicas para el manejo de la estructura de datos:  
¿Cómo se...

- Destruye una estructura?

```
while (ini != NULL) {  
    Nodo *aBorrar = ini;  
    ini = ini->_sig;  
    delete aBorrar;  
}
```



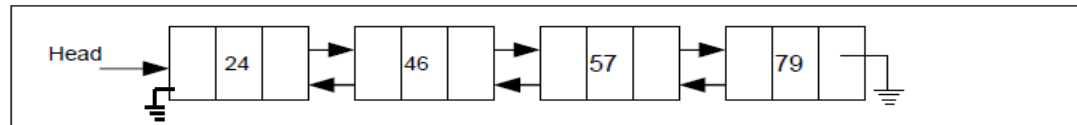
# Basadas en array vs Basadas en lista enlazada simple

---

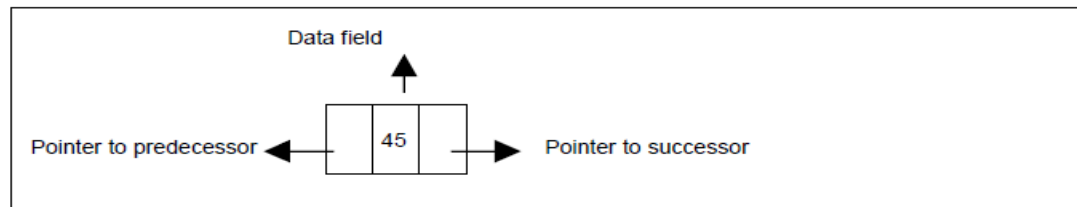
- Comparando aproximaciones
  - Implementación basada en array
    - Facilidad de uso
    - Tamaño fijo en el caso de los estáticos
    - Con arrays dinámicos el crecimiento es posible
      - Se gasta tiempo en copias y en liberaciones de memoria cada vez que se necesita crecer
      - Se puede llegar a tener mucho espacio en desuso
    - La localización del siguiente es implícita
    - Acceso directo a elemento i-ésimo
    - Inserciones y eliminaciones: requieren desplazamientos de elementos
  - Implementación basada en lista enlazada simple
    - Resuelve las dificultades del tamaño fijo y de los crecimientos prefijados
    - La localización del siguiente es explícita: hay que almacenar información explícitamente sobre dónde está el siguiente → más gasto de memoria
    - No hay acceso directo al elemento i-ésimo
    - Inserciones y eliminaciones: no requieren desplazamientos pero localizar el punto de inserción requiere atravesar la lista

# Implementaciones de TADs basadas en listas doblemente enlazadas

- Las listas doblemente enlazadas se basan en el uso de nodos (espacios de memoria independientes) que se componen de una parte de datos (elemento) y dos partes de enlace (puntero al siguiente nodo de la lista y puntero al anterior)
  - Listas bidireccionales: fácil acceso desde un nodo al nodo sucesor y al predecesor



Each node of a doubly linked list has generally three fields. Each node of doubly linked list must have at least two link fields.

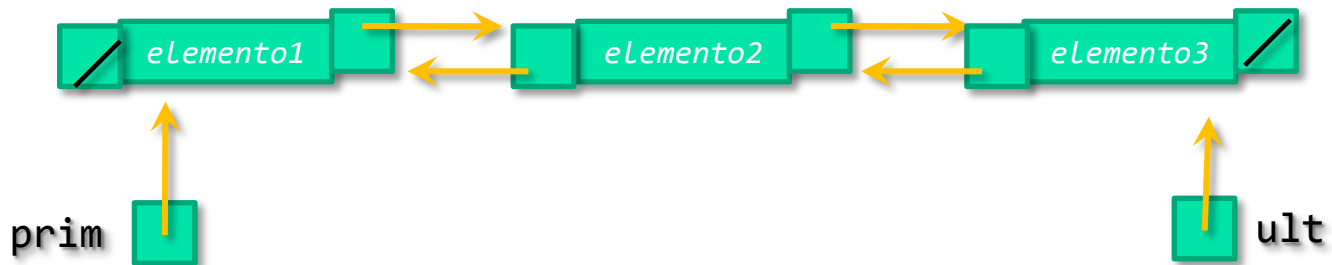


Extraído de Data Structures using C++ , V. Pathil

# Implementaciones de TADs basadas en listas doblemente enlazadas

- Las implementaciones de TADs que quieran hacer uso de esta estructura de datos, en el caso más básico, manejarán:
  - Puntero al nodo de la lista que contiene el primer elemento de la misma (de tipo T, el tipo de los elementos del TAD)
  - Puntero al nodo de la lista que contiene el último elemento de la misma (de tipo T, el tipo de los elementos del TAD)

facilitando así los desplazamientos hacia delante y hacia atrás



# Implementaciones de TADs basadas en listas doblemente enlazadas

- Características:
  - En el TAD:
    - El constructor dejará el objeto en disposición de ser usado
    - Liberación de memoria en el destructor
    - Constructor de copia y operador de asignación para hacer copias profundas
  - La clase `Nodo` nuevamente será una clase interna del TAD

```
class Nodo {  
public:  
    Nodo() : _sig(NULL), _ant(NULL) {}  
    Nodo(const T &elem) : _elem(elem), _sig(NULL), _ant(NULL) {}  
    Nodo(Nodo *ant, const T &elem, Nodo *sig) :  
        _elem(elem), _sig(sig), _ant(ant) {}  
  
    T _elem;  
    Nodo *_sig;  
    Nodo *_ant;  
};
```





## Implementaciones de TADs basadas en listas doblemente enlazadas

---

- Operaciones típicas para el manejo de la estructura de datos:  
¿Cómo se...
  - Crea una estructura vacía?
  - Inserta un elemento  $e$  entre nodos?
  - Inserta un elemento  $e$  por el principio de la estructura?
  - Inserta un elemento  $e$  por el final de la estructura?
  - Elimina un elemento entre nodos?
  - Elimina el primer elemento de la estructura?
  - Elimina el último elemento de la estructura?
  - Destruye una estructura?

# Implementaciones de TADs basadas en listas doblemente enlazadas

- Operaciones típicas para el manejo de la estructura de datos:  
¿Cómo se...

- Crea una estructura vacía?

```
prim = NULL;
```

```
ult = NULL;
```

- Inserta...

- Algoritmos similares a los de las listas enlazadas simples pero es necesario ajustar más enlaces

- Inserta un elemento *e* entre nodos?

```
Nodo *insertaElem(const T &e, Nodo *nodo1, Nodo *nodo2) {  
    Nodo *nuevo = new Nodo(nodo1, e, nodo2);  
    if (nodo1 != NULL) // hay un nodo anterior al nuevo  
        nodo1->_sig = nuevo;  
    if (nodo2 != NULL) // hay un nodo posterior al nuevo  
        nodo2->_ant = nuevo;  
    return nuevo;  
}
```

# Implementaciones de TADs basadas en listas doblemente enlazadas

- Operaciones típicas para el manejo de la estructura de datos:  
¿Cómo se...
  - Inserta un elemento  $e$  por el principio de la estructura?  
`prim = insertaElem(e, NULL, prim);`  
`if (ult == NULL) // no había elementos previamente`  
`ult = prim;`
  - Inserta un elemento  $e$  por el final de la estructura?  
`ult = insertaElem(e, ult, NULL);`  
`if (prim == NULL) // no había elementos previamente`  
`prim = ult;`

# Implementaciones de TADs basadas en listas doblemente enlazadas

- Operaciones típicas para el manejo de la estructura de datos:  
¿Cómo se...

- Elimina...

Operación parcial: sólo es posible si no está vacía la estructura

- Algoritmos similares a los de las listas enlazadas simples pero es necesario ajustar más enlaces
- Elimina un elemento entre nodos?

```
void borraElem(Nodo *n) {  
    Nodo *ant = n->_ant;  
    Nodo *sig = n->_sig;  
    if (ant != NULL) // hay nodo predecesor  
        ant->_sig = sig;  
    if (sig != NULL) // hay nodo sucesor  
        sig->_ant = ant;  
    delete n;  
}
```

# Implementaciones de TADs basadas en listas doblemente enlazadas

- Operaciones típicas para el manejo de la estructura de datos:  
¿Cómo se...

- Elimina el primer elemento de la estructura?

```
Nodo *aBorrar = prim;  
prim = prim->_sig;  
borraElem(aBorrar);  
if (prim == NULL) // sólo había un elemento  
    ult = NULL;
```

- Elimina el último elemento de la estructura?

```
Nodo *aBorrar = ult;  
ult = ult->_ant;  
borraElem(aBorrar);  
if (ult == NULL) // sólo había un elemento  
    prim = NULL;
```

# Implementaciones de TADs basadas en listas doblemente enlazadas

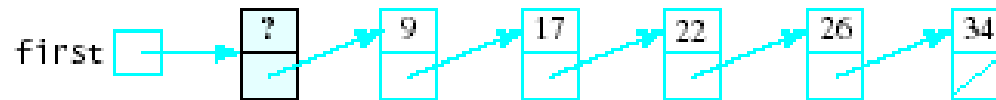
- Operaciones típicas para el manejo de la estructura de datos:  
¿Cómo se...

- Destruye una estructura?

```
while (prim != NULL) {  
    Nodo *aBorrar = prim;  
    prim = prim->_sig;  
    delete aBorrar;  
}
```

# Implementaciones de TADs basadas en listas enlazadas simples con nodo cabecera (o nodo fantasma)

- El primer nodo de una lista enlazada simple es diferente al resto: no tiene predecesor
  - Inconveniente: hay que considerar dos casos en las operaciones básicas de inserción y eliminación (primer nodo o no primer nodo)
- Solución: Lista enlazada simple con nodo cabecera
  - Lista enlazada que cuenta siempre con un nodo falso, el nodo cabecera o fantasma, al principio de la secuencia de nodos
  - La parte de datos del nodo cabecera no tiene dentro un elemento de la lista: se deja indefinido o, si interesa, puede almacenar alguna información/metadato

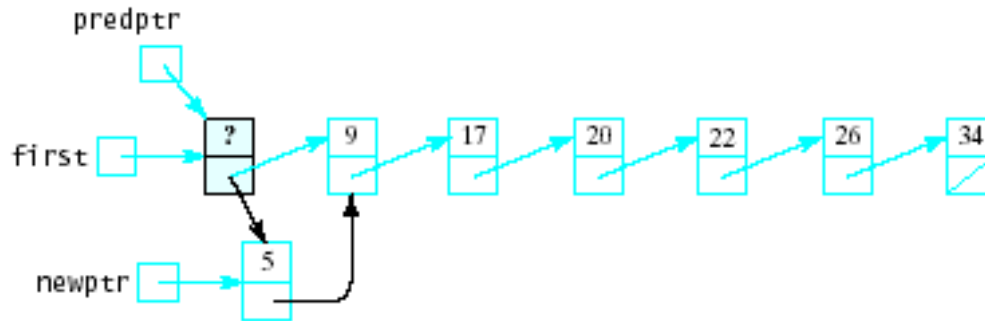


- Ahora toda lista tendrá nodo cabecera, incluso la vacía

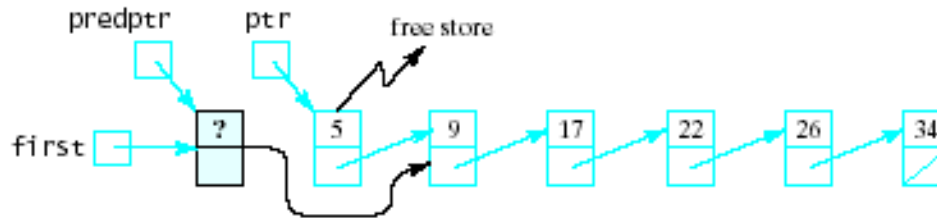


# Implementaciones de TADs basadas en listas enlazadas simples con nodo cabecera (o nodo fantasma)

- Insertar o eliminar al principio de la estructura ya no son casos especiales: el nodo cabecera sirve de predecesor



Figuras extraídas de  
Nyhoff, ADTs, Data  
Structures and Problem  
Solving with C++

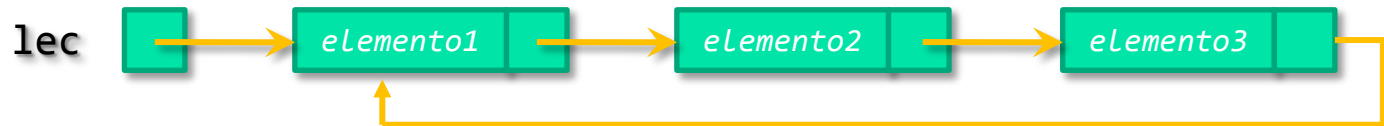


- Extensión: El uso de nodos cabecera en las listas doblemente enlazadas elimina también algunos casos especiales (el primer nodo, la lista vacía)



# Implementaciones de TADs basadas en listas enlazadas simples circulares

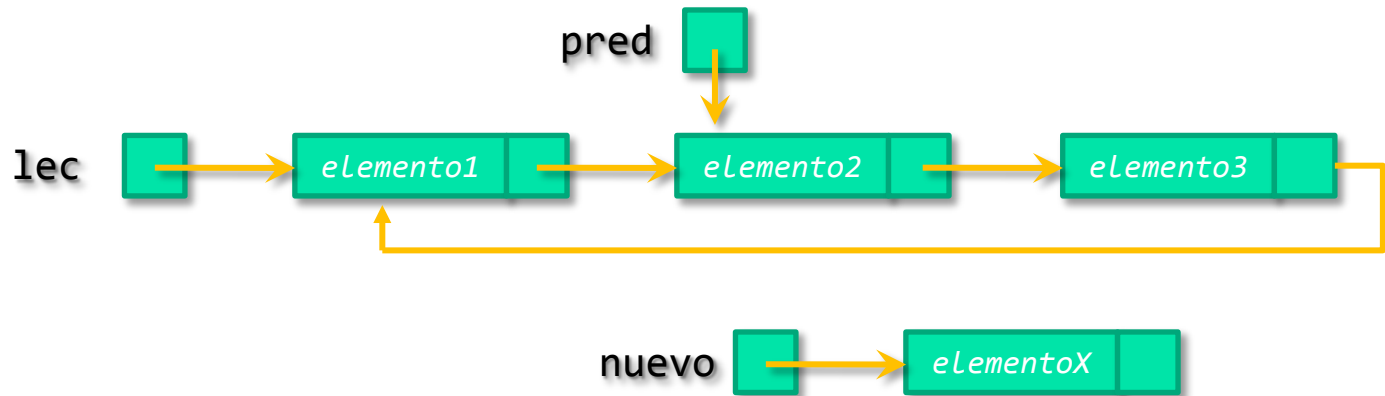
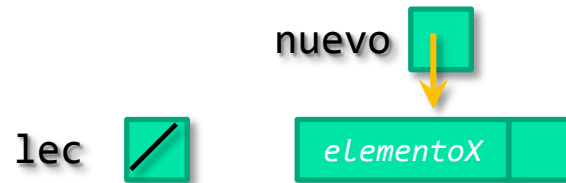
- Adaptación de la visión circular de arrays
- Lista enlazada circular: lo que sería el enlace del último nodo de una lista enlazada *lineal* estándar pasa a apuntar al primer nodo de esa lista enlazada *lineal*



- Todo nodo de una lista enlazada circular no vacía tiene un predecesor y un sucesor
  - La inserción y la eliminación no requieren el caso especial de nodos sin predecesor
  - La inserción requiere una consideración especial cuando se hace sobre una lista vacía
  - La eliminación requiere una consideración especial cuando la lista tiene un único elemento

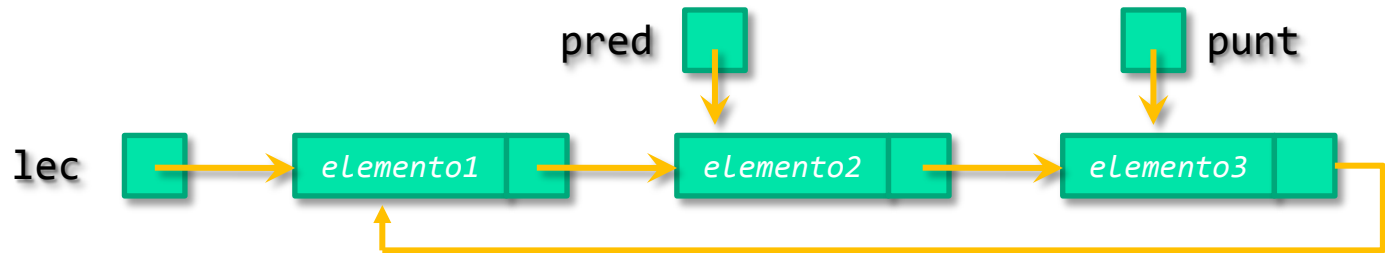
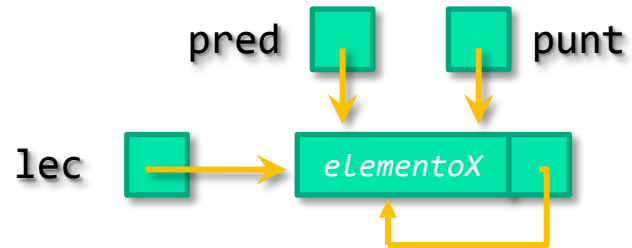
# Implementaciones de TADs basadas en listas enlazadas simples circulares

```
if (lec == NULL) {  
    nuevo->_sig = nuevo;  
    lec = nuevo;  
}  
else{  
    nuevo->_sig = pred->_sig;  
    pred->_sig = nuevo;  
}
```



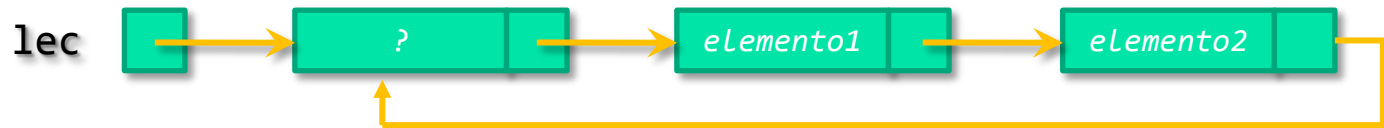
# Implementaciones de TADs basadas en listas enlazadas simples circulares

```
if (punt == pred)
    lec = NULL;
else
    pred->_sig = punt->_sig;
delete punt;
```

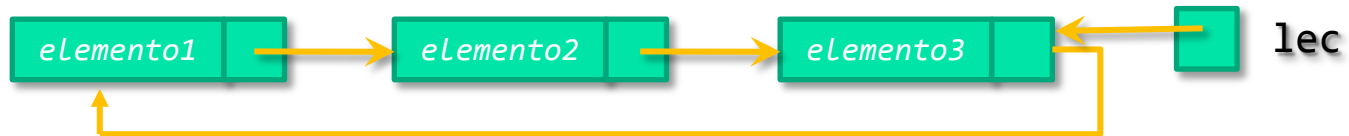


# Implementaciones de TADs basadas en listas enlazadas simples circulares

- Como en el caso de las listas enlazadas no circulares (lineales), los algoritmos de inserción y eliminación se pueden simplificar si la lista circular tiene un nodo cabecera



- En algunas implementaciones de TADs que quieran hacer uso de esta estructura de datos será preferible mantener el puntero a lo que sería el último nodo de su correspondiente lineal



- Acceso directo al "último" nodo y acceso casi directo al "primer" nodo
- Útil cuando se precisa tener acceso a los dos extremos (p.e., en implementaciones enlazadas de colas y dobles colas)

- Colección de valores de un mismo tipo, no necesariamente distintos, ordenados según van siendo añadidos y donde sólo es posible acceder al último elemento añadido
- El último elemento colocado en la pila será el primero en ser eliminado: LIFO (*last in, first out*)
- Operaciones
  - Crear una pila vacía: **pilaVacía**: --> Pila. Generadora
  - Apilar un nuevo elemento: **apila**: Pila, Elem --> Pila. Modificadora
  - Desapilar el último elemento: **desapila**: Pila --> Pila. Modificadora parcial.
  - Acceder al último elemento: **cima**: Pila --> Elem. Observadora parcial.
  - Averiguar si una pila tiene elementos: **esVacía**: Pila --> Bool. Observadora.



## El TAD Pila: implementación basada en array estático

---

1. Tipos representantes: Un array estático y un entero.
2. Función de abstracción: El array contiene los elementos de la pila y el contador entero indica el número de elementos que tiene la pila. Los elementos están condensados en las primeras posiciones del array y la cima de la pila está en la posición contador-1.
3. Invariante de la representación: Todos los elementos de la pila cumplen el invariante de la representación del tipo de sus elementos. El valor del contador  $\in [0, \text{dimensión del array}]$ .
4. Función de equivalencia: Dos pilas son iguales si el número de elementos almacenados en cada una coincide y si sus valores respectivos, uno a uno, también.

## El TAD Pila: implementación basada en array estático

array\_estatico

→ Pila.h

```
/// Excepciones generadas por algunos métodos
class EPilaLlena {};
class EPilaVacía {};

template <class T>
class Pila {
    /** Array de elementos. */
    T _v[TAM_MAX];
    /** N° de elementos almacenados en la pila. */
    unsigned int _numElems;
public:
    /** Número máximo de elementos. */
    static const int TAM_MAX = 100;
};
```

## El TAD Pila: implementación basada en array estático

```
/** Constructor; operación PilaVacía */
Pila() : _numElems(0) { }
/**
Apila un elemento. Operación modificadora parcial.
error: falla si la pila está llena
@param elem Elemento a apilar.
*/
void apila(const T &elem) {
    if (_numElems == TAM_MAX) throw EPilaLlena();
    _v[_numElems] = elem;
    _numElems++;
}
```





## El TAD Pila: implementación basada en array estático

---

```
/**
    Desapila un elemento. Operación modificadora parcial
    error: falla si la pila está vacía
 */
void desapila() {
    if (esVacía()) throw EPilaVacía();
    _numElems--;
}
```



## El TAD Pila: implementación basada en array estático

---

```
/**  
    Devuelve el elemento en la cima de la pila. Operación  
    observadora parcial.  
    error: falla si la pila está vacía  
    @return Elemento en la cima de la pila.  
    */  
const T &cima() const {  
    if (esVacia()) throw EPilaVacia();  
    return _v[_numElems-1];  
}
```



## El TAD Pila: implementación basada en array estático

---

```
/**  
    Indica si la pila está vacía.  
    @return true si la pila no tiene ningún elemento.  
*/  
bool esVacia() const {  
    return _numElems == 0;  
}
```

## El TAD Pila: implementación basada en array estático

```
/** Operador de comparación. */
bool operator==(const Pila<T> &rhs) const {
    bool iguales = true;
    if (_numElems != rhs._numElems)
        iguales = false;
    else{
        unsigned int i = 0;
        while (iguales && i < _numElems) {
            if (_v[i] != rhs._v[i]) iguales = false;
            i++;
        }
    }
    return iguales;
}
```

```
};
```



## El TAD Pila: implementación basada en array estático

---

Operación	Coste (*)
pilaVacía	$O(1)$
apila	$O(1)$
desapila	$O(1)$
cima	$O(1)$
esVacía	$O(1)$

(\*) Asumimos que el tipo de los elementos tiene operaciones de construcción, destrucción y copia  $O(1)$



## El TAD Pila: implementación basada en array dinámico

---

1. Tipos representantes: Un array dinámico (puntero a elemento de la pila) y dos enteros (contador y tamaño).
2. Función de abstracción: El array contiene los elementos de la pila y el contador indica el número de elementos que tiene la pila. Los elementos están condensados en las primeras posiciones del array y la cima de la pila está en la posición contador-1. El tamaño indica la dimensión del array. El array duplica su tamaño cuando se llena.
3. Invariante de la representación: Todos los elementos de la pila cumplen el invariante de la representación del tipo de sus elementos. El valor del contador  $\in [0, \text{dimensión del array}]$ .
4. Función de equivalencia: Dos pilas son iguales si el número de elementos almacenados en cada una coincide y si sus valores respectivos, uno a uno, también.

## El TAD Pila: implementación basada en array dinámico

vector\_dinamico

↳ Pila.h

```
/// Excepciones generadas por algunos métodos
```

```
class EPilaVacía {};
```

```
template <class T>
```

```
class Pila {
```

```
    /** Puntero al array que contiene los datos. */
```

```
    T *_v;
```

```
    /** Tamaño del vector _v. */
```

```
    unsigned int _tam;
```

```
    /** Nº de elementos almacenados en la pila. */
```

```
    unsigned int _numElem;
```



## El TAD Pila: implementación basada en array dinámico

---

```
/** Código para liberar la memoria dinámica de la pila */  
void libera() {  
    delete []_v;  
    _v = NULL;  
}  
  
/** Código para copiar sobre el receptor */  
void copia(const Pila<T> &other) {  
    _tam = other._numElems + TAM_INICIAL;  
    _numElems = other._numElems;  
    _v = new T[_tam];  
    for (unsigned int i = 0; i < _numElems; ++i)  
        _v[i] = other._v[i];  
}
```





## El TAD Pila: implementación basada en array dinámico

```
/** Código para hacer crecer el vector dinámico */  
void amplia() {  
    T *viejo = _v;  
    _tam *= 2;  
    _v = new T[_tam]; // (*)  
    for (unsigned int i = 0; i < _numElems; ++i)  
        _v[i] = viejo[i];  
    delete []viejo;  
}
```

(\*) Esta inocente instrucción hace que se llame al constructor del tipo T para crear \_tam elementos de ese tipo



## El TAD Pila: implementación basada en array dinámico

---

**public:**

```
/** Tamaño inicial del vector dinámico. */  
static const int TAM_INICIAL = 10;  
  
Pila() {  
    _v = new T[TAM_INICIAL]; // (*)  
    _tam = TAM_INICIAL;  
    _numElems = 0;  
}
```

(\*) La reserva de memoria podría llegar a fallar si no hay memoria disponible y se lanzaría una excepción `bad_alloc`. En nuestros ejemplos, por simplicidad, no nos vamos a preocupar de este tipo de incidencias.

## El TAD Pila: implementación basada en array dinámico

```
void apila(const T &elem) {
    _v[_numElems] = elem;
    _numElems++;
    if (_numElems == _tam) amplia();
}

void desapila() {
    if (esVacia()) throw EPilaVacia();
    --_numElems; // (*)
}
```

(\*) Obsérvese que nuestros vectores crecen, pero nunca decrecen. Desde el punto de vista del consumo de memoria una solución más óptima, al desapilar, debería reducir el tamaño del array cuando haya un número *considerable* de huecos libres. Para determinar qué es un número *considerable* de huecos libres se debería hacer un análisis del coste amortizado y se podría implementar un método de funcionalidad inversa a la de `amplia()` encargado de sustituir el array por otro más pequeño.



## El TAD Pila: implementación basada en array dinámico

---

```
const T &cima() const {  
    if (esVacia()) throw EPilaVacia();  
    return _v[_numElems - 1];  
}  
  
bool esVacia() const {  
    return _numElems == 0;  
}
```



## El TAD Pila: implementación basada en array dinámico

---

```
/** Destructor; elimina el vector. */
~Pila() {
    libera();
}

/** Constructor copia */
Pila(const Pila<T> &other) {
    copia(other);
}
```



## El TAD Pila: implementación basada en array dinámico

---

```
/** Operador de asignación */
Pila<T> &operator=(const Pila<T> &other) {
    if (this != &other) {// si no son la misma pila
        libera();
        copia(other);
    }
    return *this;
}
```

## El TAD Pila: implementación basada en array dinámico

```
/** Operador de comparación. */
bool operator==(const Pila<T> &rhs) const {
    bool iguales = true;
    if (_numElems != rhs._numElems)
        iguales = false;
    else{
        unsigned int i = 0;
        while (iguales && i < _numElems) {
            if (_v[i] != rhs._v[i])
                iguales = false;
            i++;
        }
    }
    return iguales;
}
```

## El TAD Pila: implementación basada en array dinámico

Operación	Coste (*)
pilaVacía	$O(1)$
apila	$O(1)$
desapila	$O(1)$
cima	$O(1)$
esVacía	$O(1)$

(\*) Asumimos que el tipo de los elementos tiene operaciones de construcción, destrucción y copia  $O(1)$





## El TAD Pila: implementación basada en lista enlazada simple

---

1. Tipos representantes: Un puntero a nodo que contiene elemento de la pila. Lista enlazada simple.
2. Función de abstracción: La lista enlazada contiene los elementos de la pila: el primer elemento de la lista es la cima; la base de la pila se guarda en el último elemento de la lista enlazada. El puntero valdrá NULL si la pila está vacía.
3. Invariante de la representación: Todos los elementos de la pila cumplen el invariante de la representación del tipo de sus elementos. La secuencia de nodos termina en NULL. Los nodos están correctamente ubicados.
4. Función de equivalencia: Dos pilas son iguales si el número de elementos almacenados en cada una coincide (sus listas enlazadas tienen el mismo  $n^\circ$  de elementos) y si sus valores respectivos, uno a uno, también.

## El TAD Pila: implementación basada en lista enlazada simple

```
/// Excepciones generadas por algunos métodos
```

```
class EPilaVacía {};
```

```
template <class T>
```

```
class Pila {
```

```
/**
```

Clase nodo que almacena internamente el elemento (de tipo T),  
y un puntero al nodo siguiente, que podría ser NULL si el nodo es  
el último de la lista enlazada.

```
*/
```

```
class Nodo {
```

```
public:
```

```
    Nodo() : _sig(NULL) {}
```

```
    Nodo(const T &elem) : _elem(elem), _sig(NULL) {}
```

```
    Nodo(const T &elem, Nodo *sig) :
```

```
        _elem(elem), _sig(sig) {}
```

```
    T _elem;
```

```
    Nodo *_sig;
```

```
};
```

```
nodos_enlazados
```

```
↳ Pila.h
```

## El TAD Pila: implementación basada en lista enlazada simple

```
/** Puntero al primer elemento */
    Nodo *_cima;

/**
Elimina todos los nodos de la lista enlazada que soporta la
pila.
*/
void libera() {
    while (_cima != NULL) {
        Nodo *aBorrar = _cima;
        _cima = _cima->sig;
        delete aBorrar;
    }
}
```

## El TAD Pila: implementación basada en lista enlazada simple

```
/** Genera una copia de other en la pila receptora
 */
void copia(const Pila &other) {
    if (other.esVacia())
        _cima = NULL;
    else {
        Nodo *puntaOrigen = other._cima; // para desplazarse por other
        Nodo *ultimo; // para tener acceso al último nodo copiado
        _cima = new Nodo(puntaOrigen->_elem);
        ultimo = _cima;
        while (puntaOrigen->_sig != NULL) {
            puntaOrigen = puntaOrigen->_sig;
            ultimo->_sig = new Nodo(puntaOrigen->_elem);
            ultimo = ultimo->_sig;
        }
    }
}
```



## El TAD Pila: implementación basada en lista enlazada simple

---

**public:**

```
Pila() : _cima(NULL) {}
```

```
void apila(const T &elem) {  
    _cima = new Nodo(elem, _cima);  
}
```

```
void desapila() {  
    if (esVacia())  
        throw EPilaVacia();  
    Nodo *aBorrar = _cima;  
    _cima = _cima->_sig;  
    delete aBorrar;  
}
```



## El TAD Pila: implementación basada en lista enlazada simple

---

```
const T &cima() const {  
    if (esVacia())  
        throw EPilaVacia();  
    return _cima->_elem;  
}
```

```
bool esVacia() const {  
    return _cima == NULL;  
}
```

```
~Pila() {  
    libera();  
    _cima = NULL;  
}
```

## El TAD Pila: implementación basada en lista enlazada simple

```
Pila(const Pila<T> &other) {  
    copia(other);  
}
```

```
Pila<T> &operator=(const Pila<T> &other) {  
    if (this != &other) { // no se intenta copiar una pila sobre sí misma  
        libera();  
        copia(other);  
    }  
    return *this;  
}
```

## El TAD Pila: implementación basada en lista enlazada simple

```
bool operator==(const Pila<T> &rhs) const {
    Nodo *cima1 = _cima;
    Nodo *cima2 = rhs._cima;
    while ((cima1 != NULL) && (cima2 != NULL) &&
           (cima1->_elem == cima2->_elem)) {
        cima1 = cima1->_sig;
        cima2 = cima2->_sig;
    }
    return (cima1 == NULL) && (cima2 == NULL);
}
```



## El TAD Pila: implementación basada en lista enlazada simple

Operación	Coste (*)
pilaVacía	$O(1)$
apila	$O(1)$
desapila	$O(1)$
cima	$O(1)$
esVacía	$O(1)$

(\*) Asumimos que el tipo de los elementos tiene operaciones de construcción, destrucción y copia  $O(1)$



## El TAD Cola

---

- Colección de valores de un mismo tipo, no necesariamente distintos, que se añaden por un extremo (el final) y se consultan y eliminan por el otro (el principio)
- El primer elemento colocado en la cola será el primero en ser eliminado: FIFO (*first in, first out*)
- Operaciones
  - Crear una cola vacía: **colaVacía**: --> Cola. Generadora
  - Añadir un elemento: **pon**: Cola, Elem --> Cola. Modificadora
  - Eliminar un elemento: **quita**: Cola--> Cola. Modificadora parcial.
  - Acceder al primer elemento: **primero**: Cola--> Elem. Observadora parcial.
  - Averiguar si una cola tiene elementos: **esVacía**: Cola--> Bool. Observadora.

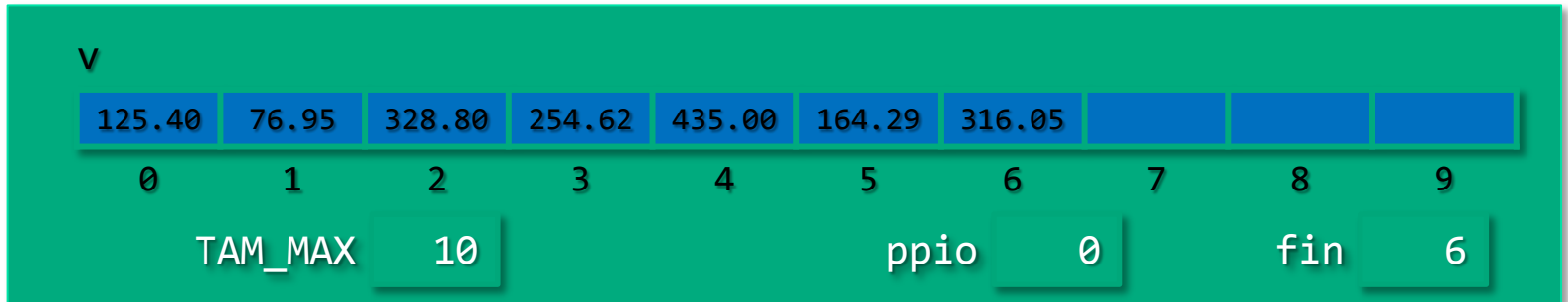


## El TAD Cola: implementación *naive* basada en array estático

---

1. Tipos representantes: Un array estático y dos enteros (principio y fin).
2. Función de abstracción: El array contiene los elementos de la cola. principio y fin contienen los índices al primer y último elemento de la cola, respectivamente (principio = 0 y fin = -1 → cola vacía)
3. Invariante de la representación: Todos los elementos de la cola cumplen el invariante de la representación del tipo de sus elementos. principio  $\in [0, \text{dimensión del array}-1]$ . fin  $\in [-1, \text{dimensión del array}-1]$
4. Función de equivalencia: Dos colas son iguales si el número de elementos almacenados en cada una coincide y si sus valores respectivos, uno a uno, también.

## El TAD Cola: implementación *naive* basada en array estático



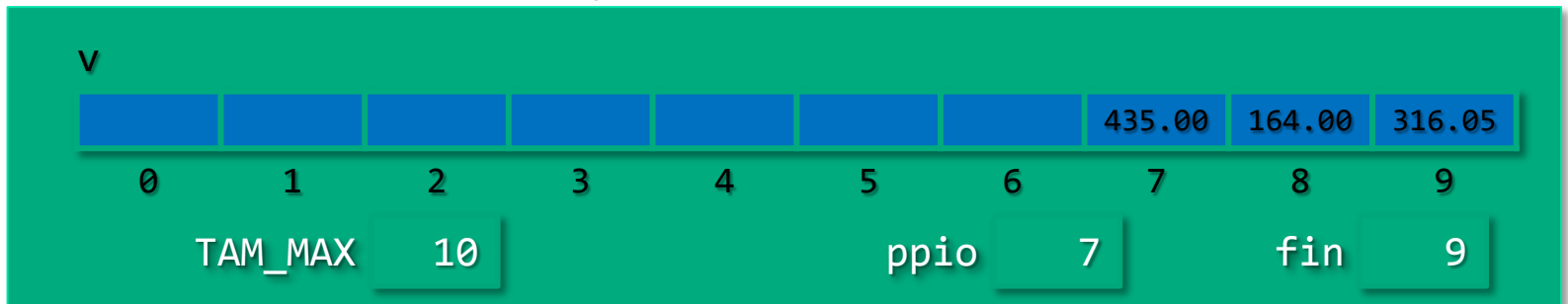
Supón el siguiente comportamiento:

- **colaVacía**:  $ppio = 0$  y  $fin = -1$
- **pon**: incrementar  $fin$  y colocar el elemento en  $v[fin]$  (si no está llena)
- **quita**: incrementar  $ppio$  (si no está vacía)
- **primero**:  $v[ppio]$  (si no está vacía)
- **esVacía**:  $fin < ppio$

¿qué inconveniente(s) tiene esta estrategia?

## El TAD Cola: implementación *naive* basada en array estático

Tras una serie de inserciones y eliminaciones la cola podría estar así...



¡La cola parece llena... pero no lo está!

Soluciones:

- quita: Tras eliminar el elemento desplazamos los demás una posición a la izquierda
- quita: Tras eliminar el elemento desplazamos los demás a la izquierda si la cola parece llena ( $\text{fin} = \text{TAM\_MAX} - 1$ )
- pon: Antes de insertar el nuevo elemento desplazamos los existentes a la izquierda si la cola parece llena pero no lo está ( $\text{fin} = \text{TAM\_MAX} - 1$  y  $\text{ppio} > 0$ )

¡El desplazamiento penaliza el coste de la implementación!

Alguna operación resulta  $O(n)$



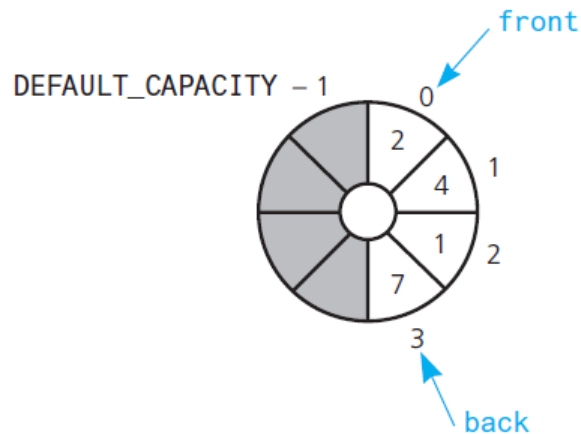
## El TAD Cola: implementación *naive* basada en array dinámico

---

1. Tipos representantes: Un array dinámico (puntero a elemento de la cola) y tres enteros (principio, fin y tamaño).
2. Función de abstracción: El array contiene los elementos de la cola. Principio y fin contienen los índices al primer y último elemento de la cola, respectivamente (principio = 0 y fin = -1  $\rightarrow$  cola vacía). El tamaño indica la dimensión del array. El array duplica su tamaño cuando se llena.
3. Invariante de la representación: Todos los elementos de la cola cumplen el invariante de la representación del tipo de sus elementos. principio  $\in [0, \text{tamaño del array}-1]$ . fin  $\in [-1, \text{tamaño del array}-1]$
4. Función de equivalencia: Dos colas son iguales si el número de elementos almacenados en cada una coincide y si sus valores respectivos, uno a uno, también.

¡Los desplazamientos siguen penalizando el coste de quita y dominan el coste de la implementación!

# El TAD Cola: implementación basada en array *circular*



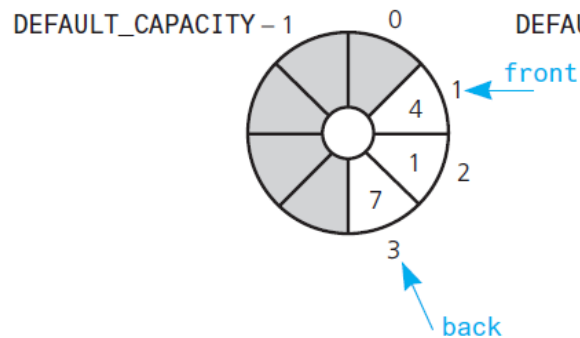
## ■ pon

```
back = (back + 1) % DEFAULT_CAPACITY;  
items[back] = newEntry;
```

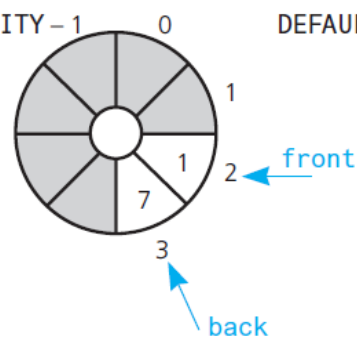
## ■ quita

```
front = (front + 1) % DEFAULT_CAPACITY;
```

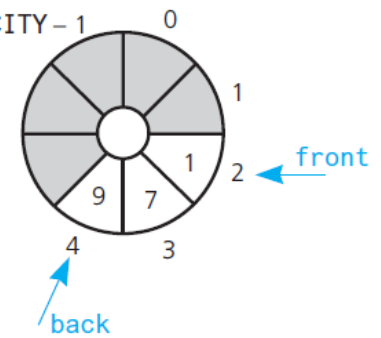
dequeue();



dequeue();

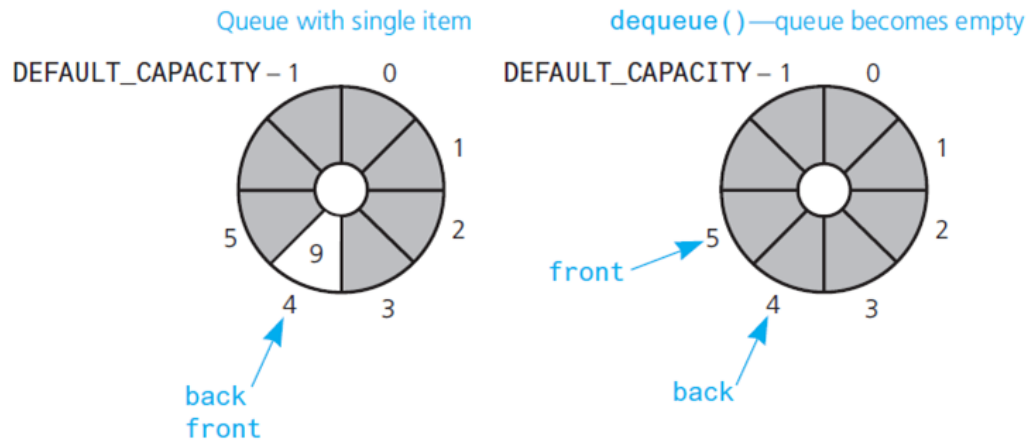


enqueue(9);

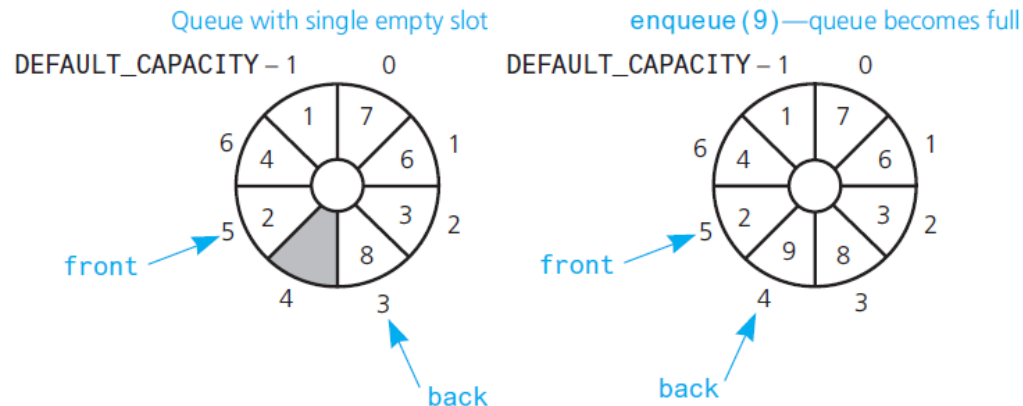


# El TAD Cola: implementación basada en array *circular*

(a) front passes back when the queue becomes empty



(b) back catches up to front when the queue becomes full



- ¿Cómo distinguir entre una cola vacía y una llena?
  - $\text{front} > \text{back}$  no es suficiente
  - Solución: llevar la cuenta del número de elementos en la cola





## El TAD Cola: implementación basada en array estático *circular*

---

1. Tipos representantes: Un array estático y tres enteros (front, back y count)
2. Función de abstracción: El array contiene los elementos de la cola. front y back contienen los índices al primer y último elemento de la cola, respectivamente. El n° de elementos de la cola se guarda en count
3. Invariante de la representación: Todos los elementos de la cola cumplen el invariante de la representación del tipo de sus elementos.  $\text{front} \in [0, \text{dimensión del array}-1]$ .  $\text{fin} \in [0, \text{dimensión del array}-1]$ .  $\text{count} \in [0, \text{dimensión del array}]$ .
4. Función de equivalencia: Dos colas son iguales si el número de elementos almacenados en cada una coincide y si sus valores respectivos, uno a uno, también

# El TAD Cola: implementación basada en array estático *circular*

Array\_estatico\_circular

Cola.h

Operación	Coste (*)
colaVacia	$O(1)$
pon	$O(1)$
quita	$O(1)$
primero	$O(1)$
esVacia	$O(1)$

(\*) Asumimos que el tipo de los elementos tiene operaciones de construcción, destrucción y copia  $O(1)$

## El TAD Cola: implementación *naive* basada en lista enlazada simple

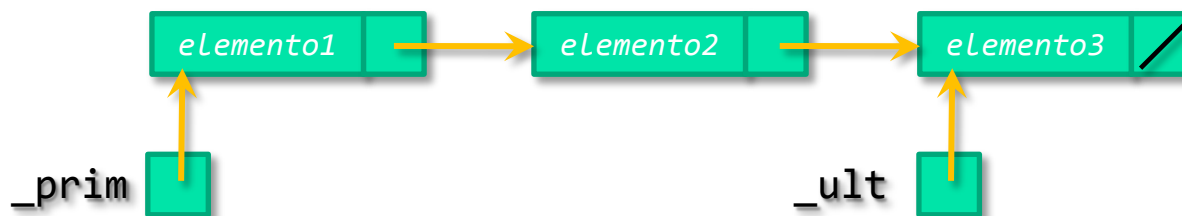
1. Tipos representantes: Un puntero a nodo que contiene elemento de la cola. Lista enlazada simple.
2. Función de abstracción: La lista enlazada contiene los elementos de la cola: el primer nodo de la lista contiene el primer elemento de la cola; el último nodo de la lista contiene el último elemento de la cola. El puntero valdrá NULL si la cola está vacía.
3. Invariante de la representación: Todos los elementos de la cola cumplen el invariante de la representación del tipo de sus elementos. La secuencia de nodos termina en NULL. Los nodos están correctamente ubicados.
4. Función de equivalencia: Dos colas son iguales si el número de elementos almacenados en cada una coincide (sus listas enlazadas tienen el mismo n° de elementos) y si sus valores respectivos, uno a uno, también.



¡La operación `pon` es ineficiente: requiere recorrer la lista hasta llegar al último nodo ( $O(n)$ )!

## El TAD Cola: implementación basada en lista enlazada simple

1. Tipos representantes: Dos punteros, `_prim` y `_ult`, a nodos que contienen elementos de la cola. Lista enlazada simple.
2. Función de abstracción: La lista enlazada contiene los elementos de la cola: el primer nodo de la lista, apuntado por `_prim`, contiene el primer elemento de la cola; el último nodo de la lista, apuntado por `_ult`, contiene el último elemento de la cola. `_prim` y `_ult` valen NULL si la cola está vacía.
3. Invariante de la representación: Todos los elementos de la cola cumplen el invariante de la representación del tipo de sus elementos. La secuencia de nodos termina en NULL. Los nodos están correctamente ubicados.
4. Función de equivalencia: Dos colas son iguales si el número de elementos almacenados en cada una coincide (sus listas enlazadas tienen el mismo n° de elementos) y si sus valores respectivos, uno a uno, también.



# El TAD Cola: implementación basada en lista enlazada simple

`nodos_enlazados`

↳ `Cola.h`

Operación	Coste (*)
colaVacia	$O(1)$
pon	$O(1)$
quita	$O(1)$
primero	$O(1)$
esVacia	$O(1)$

(\*) Asumimos que el tipo de los elementos tiene operaciones de construcción, destrucción y copia  $O(1)$



## El TAD Cola Doble (*Double-ended queue, deque*)

---

- Colección de valores de un mismo tipo, no necesariamente distintos, que se añaden, eliminan y consultan por ambos extremos
- Generalización de pila y cola
- Operaciones
  - Crear una doble cola vacía: **dcolaVacía**: --> DCola. Generadora
  - Añadir un elemento al final: **pon\_final**: DCola, Elem --> DCola. Modificadora
  - Añadir un elemento al principio: **pon\_ppio**: DCola, Elem --> DCola. Modificadora
  - Eliminar un elemento por el final: **quita\_final**: DCola--> DCola. Modificadora parcial.
  - Eliminar un elemento por el principio: **quita\_ppio**: DCola--> DCola. Modificadora parcial.
  - Acceder al primer elemento: **primero**: DCola--> Elem. Observadora parcial.
  - Acceder al último elemento: **ultimo**: DCola--> Elem. Observadora parcial.
  - Averiguar si una doble cola tiene elementos: **esVacía**: DCola--> Bool. Observadora.

# El TAD Doble Cola: implementación basada en lista doblemente enlazada

1. Tipos representantes: Dos punteros, `_prim` y `_ult`, a nodos que contienen elementos de la doble cola. Lista doble enlazada.
2. Función de abstracción: La lista doblemente enlazada contiene los elementos de la doble cola: el primer nodo de la lista, apuntado por `_prim`, contiene el primer elemento; el último nodo de la lista, apuntado por `_ult`, contiene el último elemento. `_prim` y `_ult` valen `NULL` si la doble cola está vacía.
3. Invariante de la representación: Todos los elementos de la doble cola cumplen el invariante de la representación del tipo de sus elementos. La secuencia de nodos termina en `NULL`. Los nodos están correctamente ubicados y sus enlaces al nodo anterior y siguiente correctos (si vamos al nodo anterior a `n` y luego pasamos a su siguiente debemos volver a `n`; si vamos al nodo posterior a `n` y luego pasamos a su anterior debemos volver a `n`)
4. Función de equivalencia: Dos dobles colas son iguales si el número de elementos almacenados en cada una coincide (sus listas enlazadas tienen el mismo nº de elementos) y si sus valores respectivos, uno a uno, también.



# El TAD Doble Cola: implementación basada en lista doblemente enlazada

`nodos_doblemente_enlazados`

↳ `DCola.h`

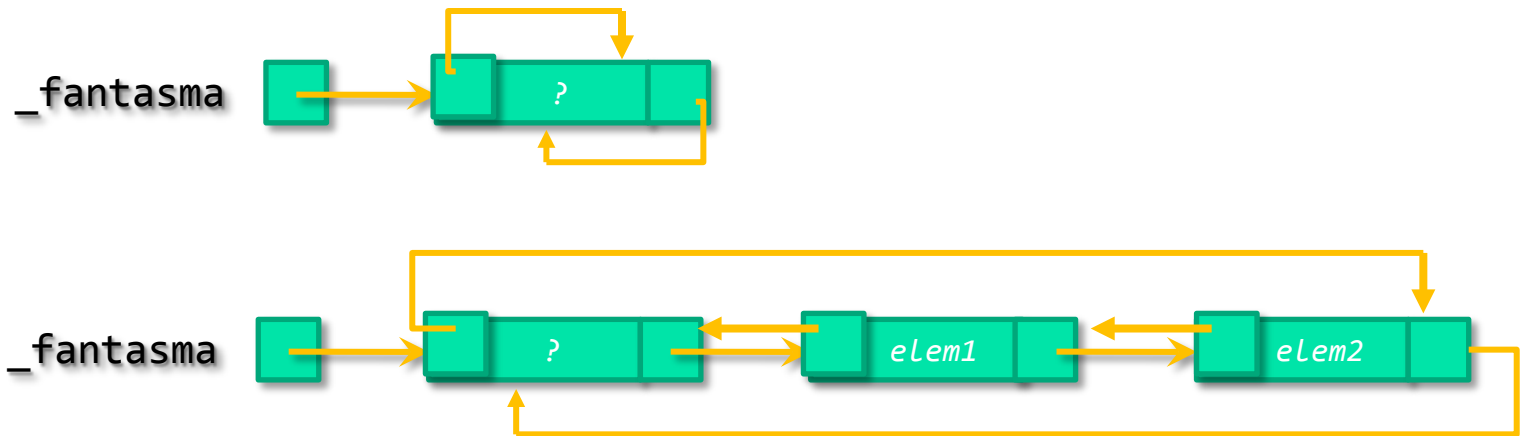
Operación	Coste (*)
<code>dcolaVacia</code>	$O(1)$
<code>pon_ppio</code>	$O(1)$
<code>pon_final</code>	$O(1)$
<code>quita_ppio</code>	$O(1)$
<code>quita_final</code>	$O(1)$
<code>primero</code>	$O(1)$
<code>ultimo</code>	$O(1)$
<code>esVacia</code>	$O(1)$

(\*) Asumimos que el tipo de los elementos tiene operaciones de construcción, destrucción y copia  $O(1)$



# El TAD Doble Cola: implementación basada en lista doblemente enlazada circular con nodo cabecera

1. Tipos representantes: Un puntero, `_fantasma`, al nodo cabecera de la lista doblemente enlazada circular.
2. Función de abstracción: Los nodos contienen los elementos de la lista, salvo el nodo cabecera. La doble cola vacía está representada por un nodo cabecera cuyos punteros `_ant` y `_sig` apuntan a él mismo. En dobles colas no vacías el siguiente al nodo cabecera contiene el primer elemento y el anterior contiene el último elemento.





## El TAD Doble Cola: implementación basada en lista doblemente enlazada circular con nodo cabecera

---

3. Invariante de la representación: Todos los elementos de la doble cola cumplen el invariante de la representación del tipo de sus elementos. Los nodos están correctamente ubicados y sus enlaces al nodo anterior y siguiente correctos (si vamos al nodo anterior a  $n$  y luego pasamos a su siguiente debemos volver a  $n$ ; si vamos al nodo posterior a  $n$  y luego pasamos a su anterior debemos volver a  $n$ ).  
El conjunto de nodos alcanzables desde el nodo cabecera en un sentido y en otro deben ser los mismos. Al ser circular, el nodo cabecera debe aparecer entre los alcanzables desde él.
4. Función de equivalencia: Dos dobles colas son iguales si el número de elementos almacenados en cada una coincide (sus listas enlazadas tienen el mismo  $n^o$  de elementos) y si sus valores respectivos, uno a uno, también.

# El TAD Doble Cola: implementación basada en lista doblemente enlazada circular con nodo cabecera

`nodos_doblemente_enlazados_circular_nodo_cabecera`

→ DCola.h

Operación	Coste (*)
dcolaVacia	O(1)
pon_ppio	O(1)
pon_final	O(1)
quita_ppio	O(1)
quita_final	O(1)
primero	O(1)
ultimo	O(1)
esVacia	O(1)

(\*) Asumimos que el tipo de los elementos tiene operaciones de construcción, destrucción y copia O(1)



## El TAD Lista

---

- Colección de valores de un mismo tipo, no necesariamente distintos, que permite añadir, eliminar y consultar por ambos extremos así como acceder a cualquier punto intermedio
- Operaciones
  - Crear una lista vacía: **listaVacía**: --> Lista. Generadora
  - Añadir un elemento al final: **pon\_final**: Lista, Elem --> Lista. Modificadora
  - Añadir un elemento al principio: **pon\_ppio**: Lista, Elem --> Lista. Modificadora
  - Eliminar un elemento por el final: **quita\_final**: Lista -> Lista. Modificadora parcial.
  - Eliminar un elemento por el principio: **quita\_ppio**: Lista -> Lista. Modificadora parcial.
  - Acceder al primer elemento: **primero**: Lista -> Elem. Observadora parcial.
  - Acceder al último elemento: : **ultimo**: Lista -> Elem. Observadora parcial.
  - Acceder al elemento i-ésimo: **elem**: Lista, pos -> Elem. Observadora parcial.
  - Averiguar si la lista tiene elementos: **esVacía**: Lista-> Bool. Observadora.
  - Obtener el número de elementos de la lista: **longitud**: Lista -> Int. Observadora.



## El TAD Lista: implementación basada en lista doblemente enlazada

1. Tipos representantes: Dos punteros, `_prim` y `_ult`, a nodos que contienen elementos de la lista, y un contador entero.
2. Función de abstracción: La lista doblemente enlazada contiene los elementos de la lista: el primer nodo de la lista enlazada, apuntado por `_prim`, contiene el primer elemento; el último nodo de la lista enlazada, apuntado por `_ult`, contiene el último elemento. `_prim` y `_ult` valen `NULL` si la lista está vacía. El contador representa el número de elementos de la lista\*.
3. Invariante de la representación: Todos los elementos de la lista cumplen el invariante de la representación del tipo de sus elementos. La secuencia de nodos termina en `NULL`. Los nodos están correctamente ubicados y sus enlaces al nodo anterior y siguiente correctos (si vamos al nodo anterior a  $n$  y luego pasamos a su siguiente debemos volver a  $n$ ; si vamos al nodo posterior a  $n$  y luego pasamos a su anterior debemos volver a  $n$ )
4. Función de equivalencia: Dos listas son iguales si el número de elementos almacenados en cada una coincide (sus listas enlazadas tienen el mismo  $n^\circ$  de elementos) y si sus valores respectivos, uno a uno, también.

(\*) Como veremos el contador es de utilidad a la hora de comprobar la validez del valor pasado a la operación elem.

# El TAD Lista: implementación basada en lista doblemente enlazada

`nodos_doblemente_enlazados`

↳ `Lista.h`

Operación	Coste (*)
<code>listaVacia</code>	$O(1)$
<code>pon_ppio</code>	$O(1)$
<code>pon_final</code>	$O(1)$
<code>quita_ppio</code>	$O(1)$
<code>quita_final</code>	$O(1)$
<code>primero</code>	$O(1)$
<code>ultimo</code>	$O(1)$
<code>elem</code>	$O(n)$
<code>esVacia</code>	$O(1)$
<code>longitud</code>	$O(1)$

(\*) Asumimos que el tipo de los elementos tiene operaciones de construcción, destrucción y copia  $O(1)$

## Recorridos de listas

- Un cliente del TAD Lista, ¿cómo muestra una lista por pantalla?

```
for (unsigned int i = 0; i < lista.longitud(); i++)  
    std::cout << lista.elem(i) << std::endl;
```

$O(n^2)$

- Los recorridos, como éste, son una operación muy común a realizar sobre una lista y podríamos estar tentados de incluir la operación como parte del TAD
  - Ventaja: implementación más eficiente

```
void print(){  
    Nodo *ptr;  
    ptr = _prim;  
    while (ptr != NULL) {  
        std::cout << ptr->_elem << std::endl;  
        ptr = ptr->_sig;  
    }  
}
```

## Recorridos de listas

- ¿Qué pasa si otro cliente quiere mostrar la lista de otra forma?  

```
std::cout << "[ ";  
for (unsigned int i = 0; i < lista.longitud(); i++)  
    std::cout << lista.elem(i) << " ";  
std::cout << "]";
```

$O(n^2)$
- ¿Tiene sentido incorporar otra versión de la operación en el TAD?

```
void print2(){  
    Nodo *ptr;  
    std::cout << "[ ";  
    ptr = _prim;  
    while (ptr != NULL) {  
        std::cout << ptr->_elem << " ";  
        ptr = ptr->_sig;  
    }  
    std::cout << "]\n";  
}
```





## Recorridos de listas

---

- La operación anterior hace un recorrido para visualizar los elementos pero... ¿qué pasa si queremos hacer otra cosa con las entradas de la lista al tiempo que nos movemos por ella?
  - Dirigirlas no a la salida estándar sino a un archivo
  - Modificar todas las entradas de una lista (p.e., sumar 1 a todos los elementos de una lista de enteros; pasar a mayúsculas todos los elementos de una lista de cadenas de caracteres, ....)
  - Localizar todos los valores que cumplen una cierta propiedad
  - ...
- Las variantes son infinitas... no parece lógico añadir otra operación al TAD cada vez que pensemos en otra forma de usar un recorrido por la lista... ni somos capaces de anticipar las necesidades de cada cliente
- La solución es proporcionar a los clientes del TAD una forma de moverse por la lista y acceder, modificar, ...

## Recorridos de listas: los iteradores como solución

- Una página de un libro puede verse como una lista de líneas...  
¿cómo cuentas el número de líneas de una página de un libro?



- Un iterador es una abstracción de la noción de puntero
  - Es un objeto que permite recorrer o atravesar una colección de datos empezando por el principio (la primera entrada)
  - En ese recorrido cada dato es considerado una vez
  - El progreso del recorrido se controla solicitando repetidamente al iterador la referencia a la siguiente entrada de la colección
  - A su paso el iterador puede acceder al dato o modificar la colección (añadiendo, eliminando o cambiando entradas)



## Recorridos de listas: iterador básico (de lectura)

---

- Un iterador de lectura es un objeto de una clase que:
  - Representa un punto intermedio en el recorrido de una colección de datos (en nuestro caso, una lista)
  - Tiene un método `elem()` que devuelve el elemento por el que va el recorrido (y tendrá el tipo base utilizado en la colección). Falla si el recorrido ha finalizado
  - Tiene un método `next()` que hace que el iterador pase al siguiente elemento del recorrido.
  - Tiene implementada la operación de comparación, de forma que se puede saber si dos iteradores son iguales.
    - Dos iteradores son iguales si: representan el mismo punto en el recorrido de una lista concreta o los dos representan el final del recorrido.



## Recorridos de listas: iterador básico (de lectura)

---

- Se define una clase interna `ConstIterator` en el TAD `Lista`, que tiene como atributo un puntero al nodo actual del recorrido
- El TAD `Lista` se extiende con dos operaciones
  - `cbegin()`: devuelve un iterador inicializado al primer elemento del recorrido
  - `cend()`: devuelve un iterador apuntando fuera del recorrido (su operación `elem()` falla)

## Recorridos de listas: iterador básico (de lectura)

```
/// Excepciones generadas por algunos métodos
class EListaVacía {};
class EAccesoInvalido {};

template <class T>
class Lista {
private:
    class Nodo {...}
    // resto de elementos privados
public:
```



## Recorridos de listas: iterador básico (de lectura)

---

```
class ConstIterator {
public:
    void next() {
        if (_act == NULL) throw EAccesoInvalido();
        _act = _act->_sig;
    }
    const T &elem() const {
        if (_act == NULL) throw EAccesoInvalido();
        return _act->_elem;
    }
    bool operator==(const ConstIterator &other) const {
        return _act == other._act;
    }
    bool operator!=(const ConstIterator &other) const {
        return !(this->operator==(other));
    }
}
```

## Recorridos de listas: iterador básico (de lectura)

**protected:**

```
friend class Lista; // Para crear y manipular objetos del tipo iterador
ConstIterator() : _act(NULL) {}
ConstIterator(Nodo *act) : _act(act) {}
// Puntero al nodo actual del recorrido
Nodo *_act;
}; // Fin de clase ConstIterator

ConstIterator cbegin() const {
    return ConstIterator(_prim);
}
ConstIterator cend() const {
    return ConstIterator();
}
// resto de elementos públicos de clase Lista
};
```

## Recorridos de listas: uso del iterador básico (de lectura)

- Ahora, un cliente del TAD Lista que quiera mostrar la lista por pantalla haría

```
Lista<int>::ConstIterator it = lista.cbegin();  
    // iterador que apunta al principio de la lista  
while (it != lista.cend()){  
    std::cout << it.elem() << endl;  
    it.next();  
}
```

$O(n)$

- Y otro haría

```
Lista<int>::ConstIterator it = lista.cbegin();  
std::cout << "[ ";  
while (it != lista.cend()){  
    std::cout << it.elem() << " ";  
    it.next();  
}  
std::cout << "]";
```





## Recorridos de listas: iterador de escritura (para modificar elementos)

---

- Un iterador de escritura es un objeto de una clase que:
  - Representa un punto intermedio en el recorrido de una colección de datos (en nuestro caso, una lista)
  - Tiene un método `elem()` que devuelve el elemento por el que va el recorrido (y tendrá el tipo base utilizado en la colección). Falla si el recorrido ha finalizado
  - Tiene un método `next()` que hace que el iterador pase al siguiente elemento del recorrido.
  - Tiene un método `set(item)` que cambia por `item` el valor del elemento de la lista por el que está pasando
  - Tiene implementada la operación de comparación, de forma que se puede saber si dos iteradores son iguales.
    - Dos iteradores son iguales si: representan el mismo punto en el recorrido de una lista concreta o los dos representan el final del recorrido.



## Recorridos de listas: iterador de escritura (para modificar elementos)

---

- Se define una clase interna `Iterator` en el TAD `Lista`, que tiene como atributo un puntero al nodo actual del recorrido
- El TAD `Lista` se extiende con dos operaciones
  - `begin()`: devuelve un iterador inicializado al primer elemento del recorrido
  - `end()`: devuelve un iterador apuntando fuera del recorrido (su operación `elem()` falla)

## Recorridos de listas: iterador de escritura (para modificar elementos)

```
/// Excepciones generadas por algunos métodos
```

```
class EListaVacía {};
```

```
class EAccesoInvalido {};
```

```
template <class T>
```

```
class Lista {
```

```
private:
```

```
    class Nodo {...}
```

```
    // resto de elementos privados
```

```
public:
```

## Recorridos de listas: iterador de escritura (para modificar elementos)

```
class Iterator {
public:
    void next() {
        if (_act == NULL) throw EAccesoInvalido();
        _act = _act->_sig;
    }
    const T &elem() const {
        if (_act == NULL) throw EAccesoInvalido();
        return _act->_elem;
    }
    void set(const T &elem) {
        if (_act == NULL) throw EAccesoInvalido();
        _act->_elem = elem;
    }
    bool operator==(const Iterator &other) const {
        return _act == other._act;
    }
    bool operator!=(const Iterator &other) const {
        return !(this->operator==(other));
    }
}
```

## Recorridos de listas: iterador de escritura (para modificar elementos)

**protected:**

```
friend class Lista;
```

```
Iterator() : _act(NULL) {}
```

```
Iterator(Nodo *act) : _act(act) {}
```

```
// Puntero al nodo actual del recorrido
```

```
Nodo *_act;
```

```
}; // Fin de clase Iterator
```

```
Iterator begin() {  
    return Iterator(_prim);  
}
```

```
Iterator end() const {  
    return Iterator();  
}
```

```
// resto de elementos públicos de clase Lista
```

```
};
```



## Recorridos de listas: uso del iterador de escritura (para modificar elementos)

---

- Un cliente del TAD Lista (de enteros) que quiera multiplicar por 2 todos los elementos haría

```
Lista<int>::Iterator itEscritura = lista.begin();
while (itEscritura != lista.end()){
    itEscritura.set(itEscritura.elem()*2);
    itEscritura.next();
}
```



## Usando iteradores para insertar elementos en una lista

---

- Extensión del TAD Lista con una nueva operación
  - Insertar elemento: **insertar**: Lista, Elem, Iterador --> Lista. Generadora
  - El iterador (de escritura) marca el punto de inserción
  - El elemento se añade *a la izquierda* del punto de inserción:
    - Si insertamos cuando el iterador está colocado al principio de la lista (`begin()`), el nuevo elemento pasará a ser el primero y el iterador pasará a apuntar al segundo
    - Si el iterador está colocado al final (en `end()`), el elemento insertado será el nuevo último elemento de la lista y el iterador seguirá apuntado a `end()`

## Usando iteradores para insertar elementos en una lista

`nodos_doblemente_enlazados`

↳ `Lista.h`

```
void insertar(const T &elem, const Iterator &it) {
    if (_prim == it._act) // Caso especial: añadir al principio
        pon_ppio(elem);
    else if (it._act == NULL) // Caso especial: añadir al final
        pon_final(elem);
    else { // Caso general: inserción en punto medio
        insertaElem(elem, it._act->_ant, it._act);
        _numElems++;
    }
}
```





## Usando iteradores para eliminar elementos en una lista

---

- Extensión del TAD Lista con una nueva operación
  - Eliminar elemento: **eliminar**: Lista, Iterador --> Lista, Iterador.  
Modificadora parcial
  - El iterador (de escritura) marca el punto de eliminación
  - El elemento a eliminar deja de existir y el iterador recibido deja de ser válido.
    - Para poder seguir recorriendo la lista la operación devuelve un nuevo iterador que deberá utilizarse a partir de ese momento para continuar el recorrido (“apuntará” al siguiente en el recorrido)

## Usando iteradores para eliminar elementos en una lista

`nodos_doblemente_enlazados`

```
Iterator eliminar(const Iterator &it) {  
    if (it._act == NULL) throw EAccesoInvalido();  
    if (it._act == _prim) { // Caso especial: primer elemento  
        quita_ppio();  
        return Iterator(_prim); // iterador al nuevo primer elemento  
    }  
    else if (it._act == _ult) { // Caso especial: último elemento  
        quita_final();  
        return Iterator(); // iterador apuntando a NULL  
    }  
    else { // Caso general: el elemento a borrar es interno a la lista  
        Nodo *sig = it._act->_sig;  
        borraElem(it._act);  
        _numElems--;  
        return Iterator(sig);  
    }  
}
```

↳ Lista.h

## El TAD Lista: implementación basada en lista doblemente enlazada

Operación	Coste (*)
listaVacia	$O(1)$
pon_ppio	$O(1)$
pon_final	$O(1)$
quita_ppio	$O(1)$
quita_final	$O(1)$
primero	$O(1)$
ultimo	$O(1)$
elem	$O(n)$
esVacia	$O(1)$
longitud	$O(1)$
insertar	$O(1)$
eliminar	$O(1)$

(\*) Asumimos que el tipo de los elementos tiene operaciones de construcción, destrucción y copia  $O(1)$



## Iteradores y algoritmos genéricos

- Los iteradores permiten abstraer el TAD que se recorre y se pueden desarrollar algoritmos genéricos que funcionen sobre distintas colecciones
- Por ejemplo: algoritmo genérico para sumar los elementos dentro de un intervalo de una colección de datos enteros

```
template <class TIterador>
int sumaTodos(TIterador it, TIterador fin) {
    int resul = 0;
    while (it != fin) {
        resul = resul + it.elem();
        it.next();
    }
    return resul;
}
```

Los iteradores deben tener los métodos elem y next

```
cout << sumaTodos(lista.cbegin(), lista.cend());
```



## Peligro de los iteradores

---

- Los iteradores permiten recorridos eficientes pero su uso conlleva un riesgo debido a la existencia de efectos laterales, ya que un iterador abre la puerta a acceder a los elementos de la lista desde fuera de la propia lista
- Cambios que ocurran en la lista pueden afectar al resultado de las operaciones del propio iterador

```
Lista<int> l;  
l.pon_ppio(3);  
Lista<int>::Iterator iter = l.begin();  
l.quita_ppio(); // Quitamos el primer elemento y el iterador  
                queda invalidado  
cout << iter.elem() << endl; // Accedemos a él ¡¡CRASH!!
```



## En el mundo real...

---

- La STL de C++ es una colección de componentes reutilizables implementados como clases y funciones genéricas (plantillas), optimizados para dotarlos de buena eficiencia
- En la STL se distinguen tres categorías básicas de componentes reutilizables:
  - Contenedores: distintos tipos genéricos de colecciones de datos con comportamientos particulares (vectores, listas, conjuntos, etc.)
  - Algoritmos: operaciones genéricas para manipular contenedores
  - Iteradores: índices *sofisticados* para colecciones que permiten distintas formas de acceso a los elementos de los contenedores



## En el mundo real...

---

- Contenedores de la STL: de primera clase y adaptadores de contenedores
  - Contenedores de primera clase
    - Contenedores de secuencia (o secuenciales): almacenan los elementos en un orden específico, preservando las posiciones relativas en las que los elementos se insertan y eliminan

La clase **vector** implementa secuencias con acceso directo a los elementos e inserciones/eliminaciones rápidas por el final.

La clase **deque** implementa secuencias con acceso directo a los elementos e inserciones/eliminaciones rápidas por el principio y final.

La clase **list** implementa listas con inserciones/eliminaciones rápidas en cualquier posición.
    - Contenedores asociativos: eficientes búsquedas; los elementos se guardan ordenados según el criterio de comparación del contenedor (por defecto, <)

La clase **set** implementa conjuntos de elementos con búsquedas rápidas y sin posibilidad de elementos duplicados.

La clase **multiset** implementa conjuntos de elementos con búsquedas rápidas y con posibilidad de elementos duplicados.

La clase **map** implementa conjuntos de asociaciones (pares) con búsquedas rápidas por clave y sin posibilidad de elementos duplicados.

La clase **multimap** implementa conjuntos de asociaciones (pares) con búsquedas rápidas por clave y con posibilidad de elementos duplicados.



## En el mundo real...

---

- Adaptadores de contenedores: contenedores de propósito específico creados a partir de los de primera clase (**deque** es el más usado) y que proporcionan una interfaz restringida al contenedor base (en particular no proporcionan iteradores)  
La clase **stack** implementa pilas (estructura LIFO: *último en entrar, primero en salir*)  
La clase **queue** implementa colas (estructura FIFO: *primero en entrar, primero en salir*)  
La clase **priority\_queue** implementa colas de prioridad





## En el mundo real...

---

- Los contenedores definen varios tipos para los iteradores
  - **iterator** que permite lecturas y escrituras
  - **const\_iterator** es para iteradores de sólo lectura
  - **reverse\_iterator** es para iteradores de lectura/escritura que se desplazan *al revés*, de forma que con al avanzar se pasa al elemento anterior (iteradores inversos que recorren la colección en sentido contrario)
  - **const\_reverse\_iterator** es para iteradores inversos de sólo lectura



## En el mundo real...

---

- Los algoritmos de la STL son operaciones genéricas de tratamiento de contenedores (ordenaciones, búsquedas, inserciones, eliminaciones, etc.)
- Implementados a través de plantillas de función
- Se apoyan en un amplio uso de iteradores
- Aplicables sobre arrays
- La inmensa mayoría están definidos en el archivo de cabecera `<algorithm>`



## Bibliografía del tema

---

- Apuntes ISBN 978-84-697-0852-1
- Data abstraction & Problem Solving with C++: Walls and Mirrors, 6<sup>th</sup> edition / Frank Carrano & Timothy Henry / Pearson, 2013
- ADTs, Data Structures, and Problem Solving with C++, 2<sup>nd</sup> edition / Larry Nyhoff / Pearson – Prentice Hall, 2005
- Estructuras de datos y métodos algorítmicos: Ejercicios resueltos / Martí Oliet, Ortega Mallén y Verdejo López / Pearson – Prentice Hall, 2010






## Acerca de Creative Commons

---

### Licencia CC ([Creative Commons](http://creativecommons.org/))

Este tipo de licencias ofrecen algunos derechos a terceras personas bajo ciertas condiciones.

Este documento tiene establecidas las siguientes:

-  Reconocimiento (*Attribution*):  
En cualquier explotación de la obra autorizada por la licencia hará falta reconocer la autoría.
  
-  No comercial (*Non commercial*):  
La explotación de la obra queda limitada a usos no comerciales.
  
-  Compartir igual (*Share alike*):  
La explotación autorizada incluye la creación de obras derivadas siempre que mantengan la misma licencia al ser divulgadas.

En <http://es.creativecommons.org/> y <http://creativecommons.org/> puedes saber más de Creative Commons.