

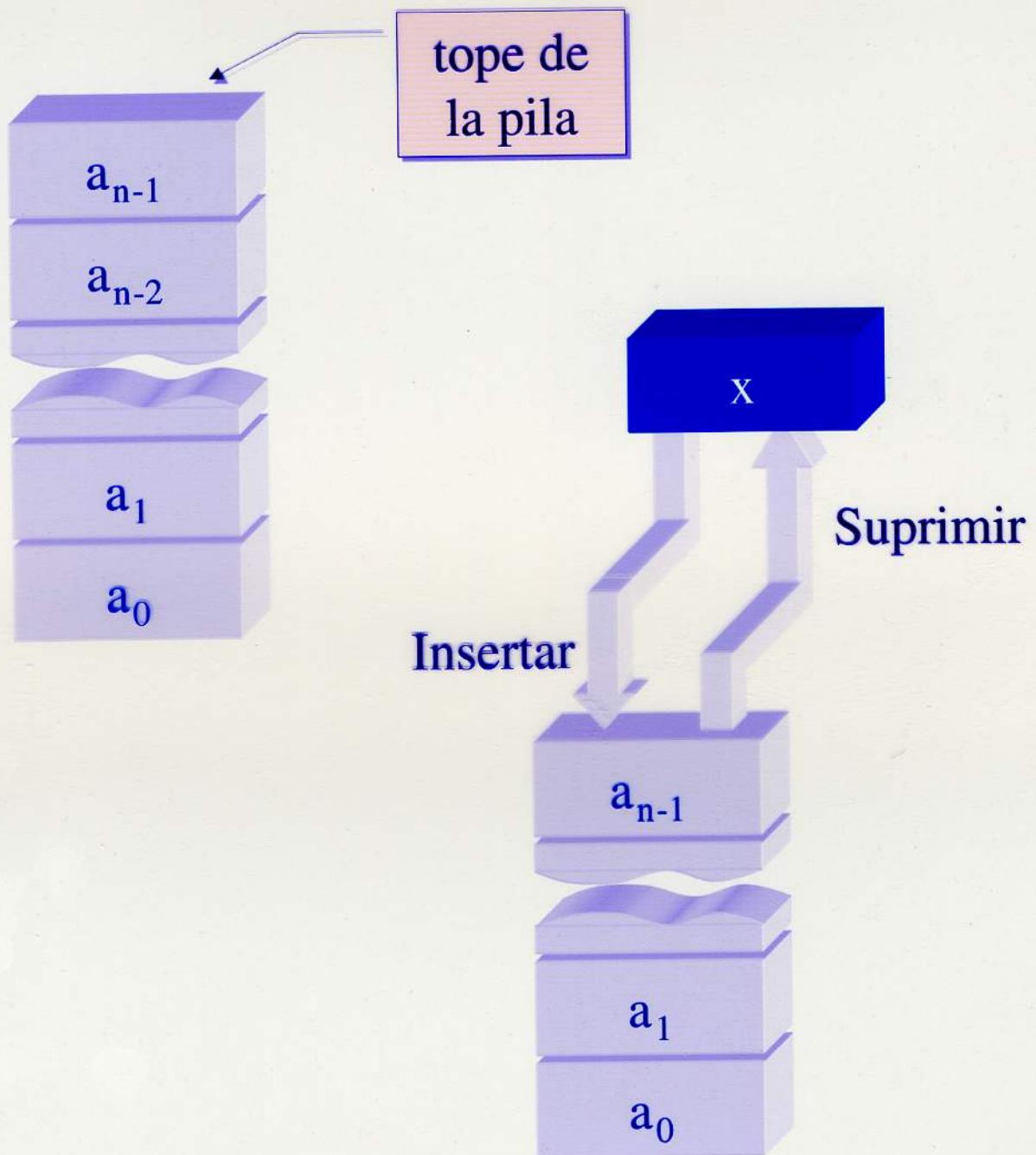
Pilas

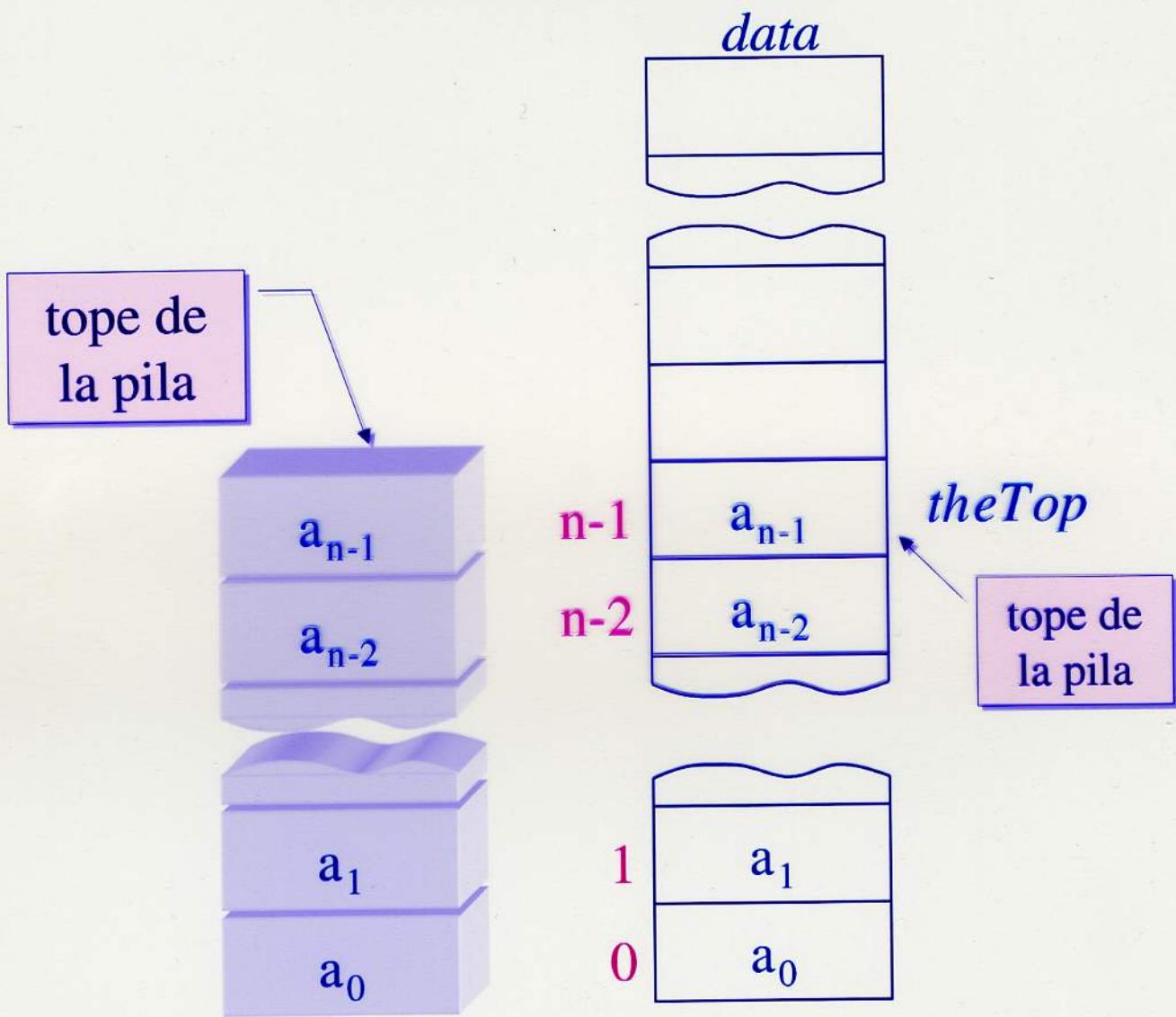
Las pilas son estructuras de datos lineales, denominadas así, porque consisten en una secuencia de elementos $a_0 \ a_1 \ \dots \ a_{n-1}$ dispuestos en una dimensión.

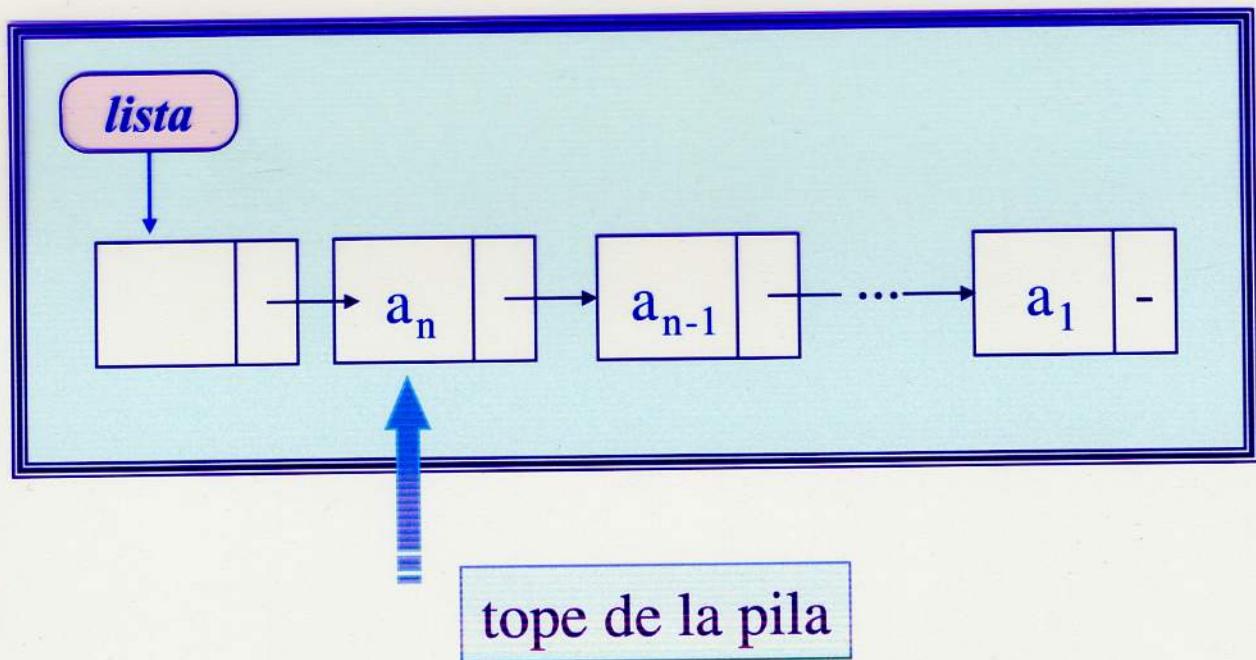
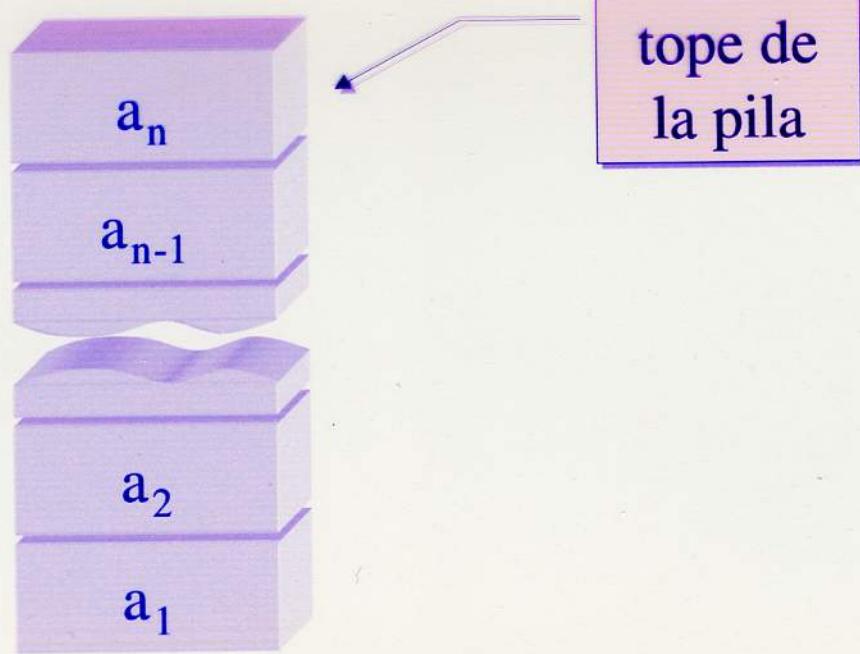
Dentro de las estructuras lineales, las pilas también se conocen como estructuras **LIFO** (*Last In, First Out*).

El nombre **LIFO** hace referencia al modo en que se accede a los elementos:

- ☞ *todas las inserciones y supresiones tienen lugar en un extremo denominado tope.*

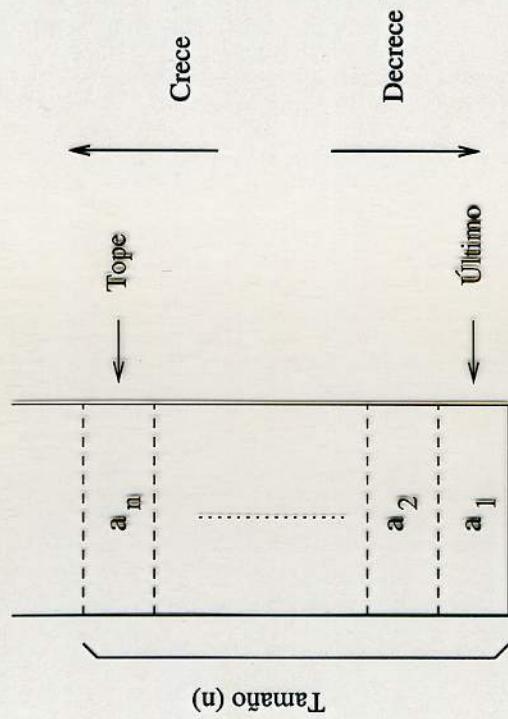






Pilas.

Una *Pila* es un tipo de dato que contiene una secuencia de valores, especialmente diseñado para realizar inserciones y borrados en uno de sus extremos. Por ello, se suele representar



Pila

También denominadas listas LIFO, ya que el último en entrar es el primero en salir. Así, los accesos a los elementos de la Pila se realizan por un extremo, denominado *tope*. Las operaciones básicas son:

- *Tope*. Devuelve el elemento del tope
- *Poner*. Añade un elemento "encima" del tope.
- *Quitar*. Elimina el elemento del tope.
- *Vacia*. Indica si la pila está vacía.

Pilas

Esquema de pila

Una posible clase *Pila* para almacenar datos de tipo *char* puede tener la siguiente sintaxis.

```
#ifndef __PILA_H__
#define __PILA_H__

class Pila{
    ...
public:
    Pila();
    Pila(const Pila& p);
    ~Pila();
    Pila& operator= (const Pila& p);
};

bool vacia() const;
void poner (char c);
void quitar();
char tope() const;
};
```

Uso de una pila

Con esta sintaxis y la semántica comentada

```
#include <iostream>
#include <pila.h>
using namespace std;

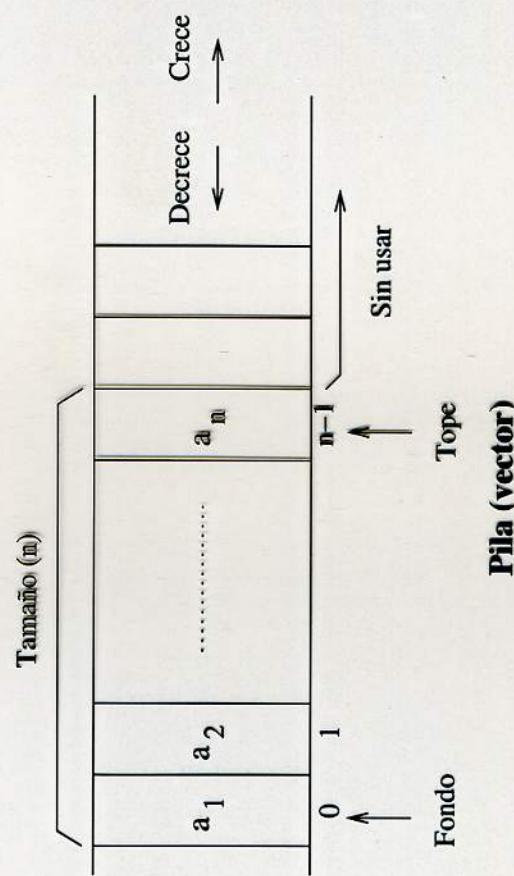
int main()
{
    Pila p,q;
    char dato;

    cout << "Escriba una frase" << endl;
    while ((dato==cin.get())!='\n')
        p.poner(dato);

    cout << "Los escribimos al revés" << endl;
    while (!p.vacia()) {
        cout << p.tope();
        q.poner(p.tope());
        p.quitar();
    }
    cout << endl << "Los originales eran" << endl;
    while (!q.vacia()) {
        cout << q.tope();
        q.quitar();
    }
    cout << endl;
    return 0;
}
```

Pilas (vectores) (1/3).

Almacenamos la secuencia de valores en un vector,



- El *fondo* de la pila se encuentra en la *posición cero*.
- El *número de elementos* varía con el *crecimiento y decrecimiento*. Tenemos que guardar *un entero*.
- Si *insertamos* elementos, el vector se *puede agotar* (*capacidad limitada*). Para *resolverlo*, podemos *usar memoria dinámica*.

Pilas (vectores) (2/3)

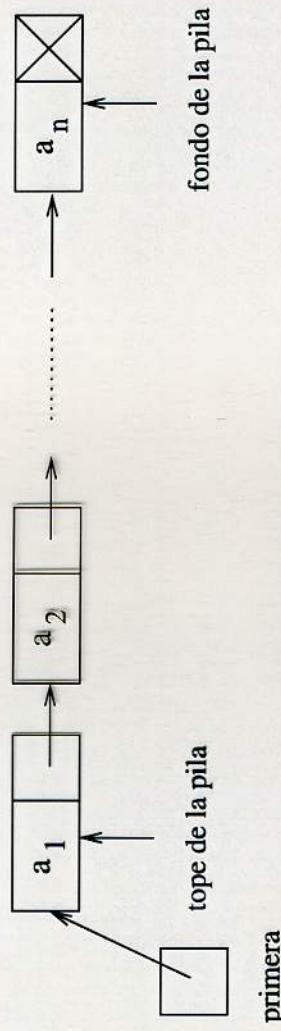
Implementación de pila	Comentarios
<p>Una posible clase Pila implementada</p> <pre> typedef char Tbase; class Pila{ Tbase datos[500]; int nelem; public: Pila(); Pila(const Pila& p); ~Pila(); Pila& operator= (const Pila& p); bool vacia() const { return nelem==0;} void poner (Tbase c) { assert(nelem<500); datos[nelem]= c; nelem++; } void quitar() { assert (nelem>0); nelem--; } Tbase tope() const { assert (nelem>0); return datos[nelem-1]; } };</pre>	<p>Algunos comentarios:</p> <ul style="list-style-type: none"> • No se han implementado las primeras funciones. • Nótese que se ha incluido un tipo <i>Tbase</i> para hacer más fácil la selección de un tipo base almacenado. • Se puede mejorar. Sin embargo, nos centraremos en implementaciones en <i>memoria dinámica</i>, para permitir el crecimiento y decrecimiento de la pila sin limitaciones. <p>Las principales ventajas y desventajas son:</p> <ul style="list-style-type: none"> • Ventajas: implementación muy sencilla que hace que el compilador busque de forma automática la memoria necesaria. Es más sencilla de programar e incluso más eficiente. • Desventajas: se desperdicia memoria y, pero aún, se puede llenar.

Pilas (vectores) (3/3)

<i>Pila.h</i>	<i>Pila.cpp</i>
<pre>#ifndef __PILA_H__ #define __PILA_H__ typedef char Tbase; class Pila{ Tbase *datos; int reservados; int nelem; public: Pila(); Pila(const Pila& p); ~Pila(); Pila& operator= (const Pila& p); bool vacia() const { return nelem==0;} void poner (Tbase c) void quitar(); Tbase top() const; }; #endif</pre>	<p>Algunas funciones no se implementan. Véanse por ejemplo las de Vector_Disperso.</p> <pre>Pila::Pila() { ... } Pila::Pila(const Pila& p) { ... } Pila::~Pila() { ... } Pila& Pila::operator= (const Pila& p) { ... } void Pila::resize(int n){ ... } void Pila::poner(TBase c) { if (nelem==reservados) resize(2*reservados); datos[nelem]= c; nelem++; } void Pila::quitar() { assert (nelem>0); nelem--; if (nelem<reservados/4) resize(reservados/2); } TBase Pila::top() const { assert (nelem>0); return datos[nelem-1]; }</pre>

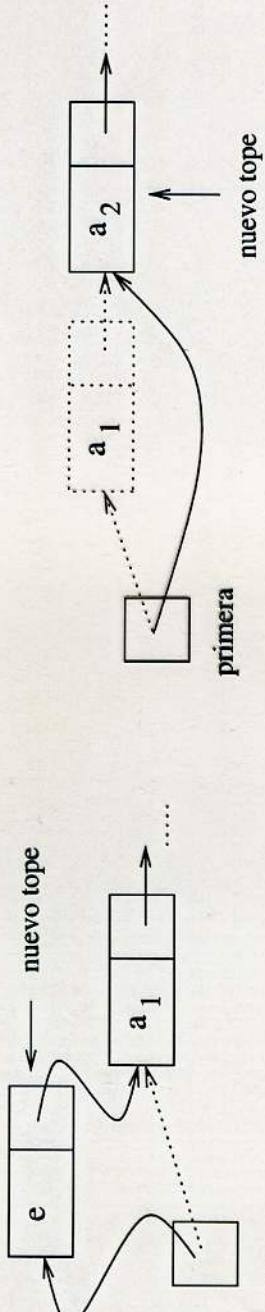
Pilas (celdas enlazadas) (1/3).

Almacenamos la secuencia de valores **en celdas enlazadas**,



Pila (celdas enlazadas)

- Una pila vacía contiene un *puntero (primera) nulo*.
- El *tope* de la pila se encuentra en la primera celda. Así es más eficiente.
- Si *insertamos* un elemento, se añade una nueva celda *al principio* y si lo *borramos*, eliminamos la *primera* celda.



Poner un nuevo elemento

Ostrar un elemento

Pilas (celdas enlazadas) (2/3).

Pila.h	Pila.cpp
<pre>#ifndef __PILA_H__ #define __PILA_H__ typedef char Tbase; struct CeldaPila{ Tbase elemento; CeldaPila *sig; }; class Pila{ CeldaPila *primera; public: Pila(); Pila(const Pila& p); ~Pila(); Pila& operator= (const Pila& p); bool vacia() const; void poner (TBase c) void quitar(); Tbase tope() const; }; #endif</pre>	<pre>Pila::Pila(): primera(0) { } Pila::Pila(const Pila& p) { if (p.primera==0) primera= 0; else { primera= new CeldaPila; primera->elemento= p.primera->elemento; CeldaPila *src=p.primera,*dest=primera; while (src->sig!=0) { dest->sig= new CeldaPila; src= src->sig; dest= dest->sig; dest->elemento= src->elemento } dest->sig=0; } } Pila::~Pila() { CeldaPila *aux; while (primera!=0) { aux= primera; primera= primera->sig; delete aux; } }</pre>

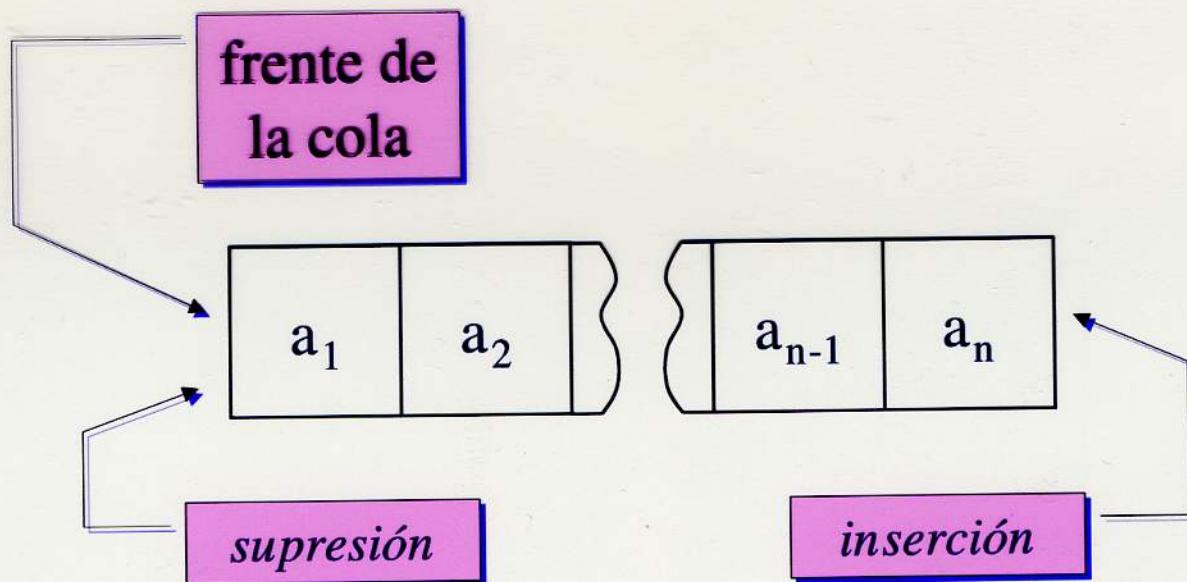
Pilas (celdas enlazadas) (3/3).

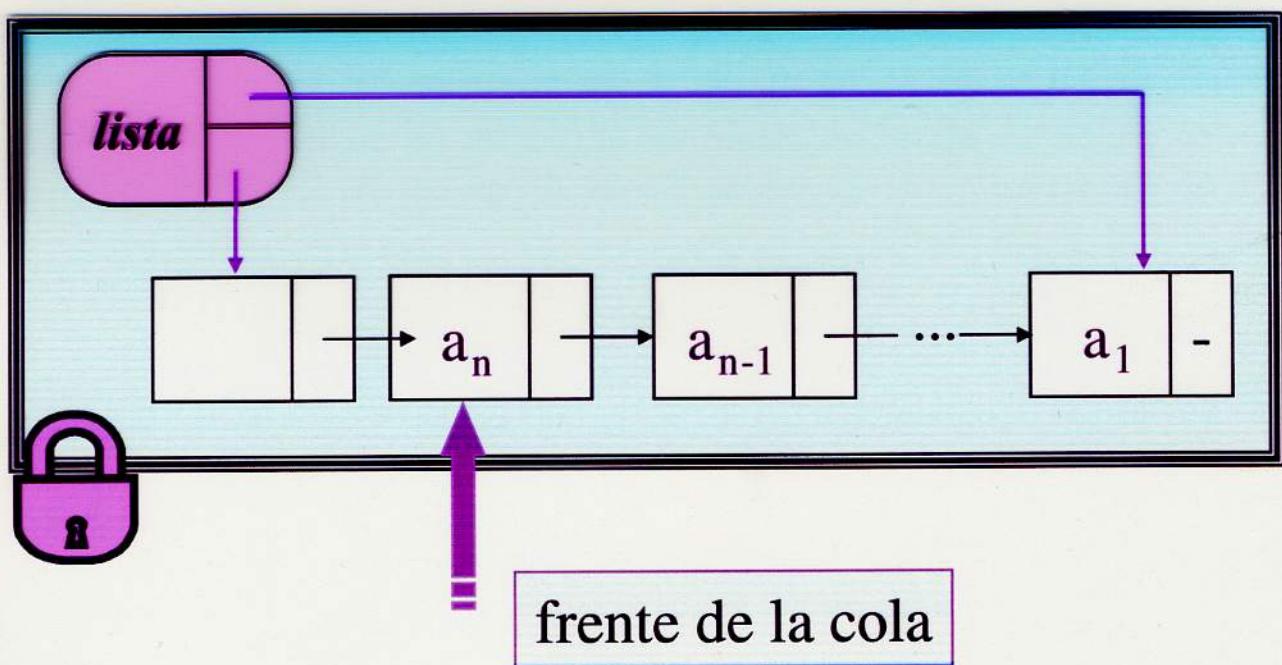
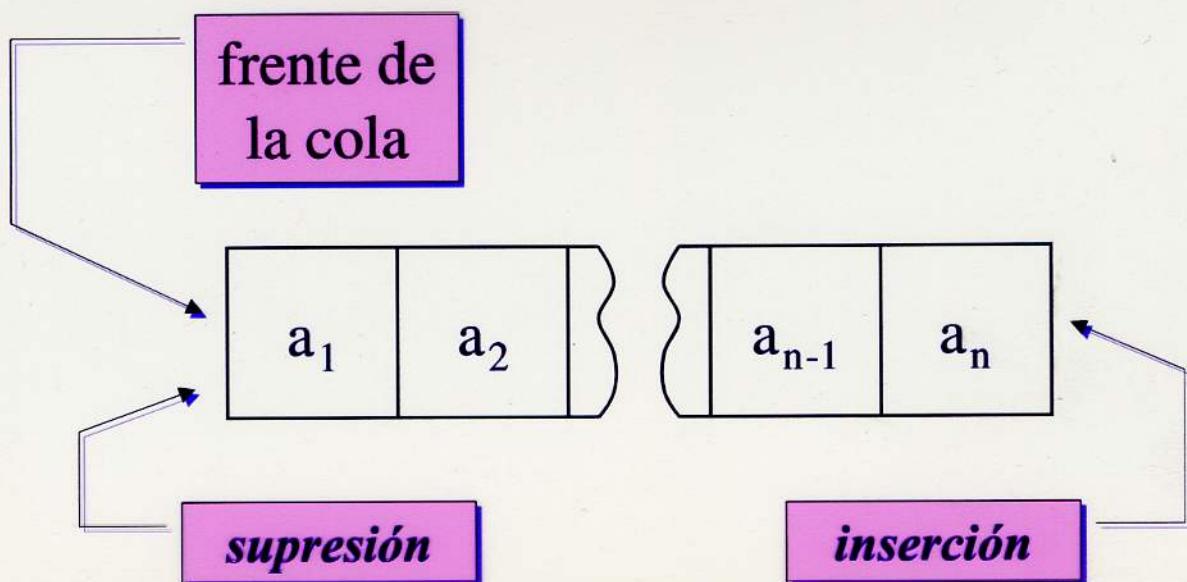
<i>Pila.h</i>	<i>Pila.cpp</i>
<pre>Pila& Pila::operator= (const Pila& p) { Pila paux(p); CeldaPila *aux; aux= this→primera; this→primera= paux→primera; paux→primera= aux; return *this; }</pre>	<pre>TBase Pila::tope() const { assert (primera!=0); return primera→elemento; } bool Pila::vacia() const { return primera==0; } void Pila::poner(TBase c) { CeldaPila *aux= new CeldaPila; aux→elemento= c; aux→sig= primera; primera= aux; } void Pila::quitar() { assert (primera!=0); CeldaPila *aux= primera; primera= primera→sig; delete aux; }</pre>

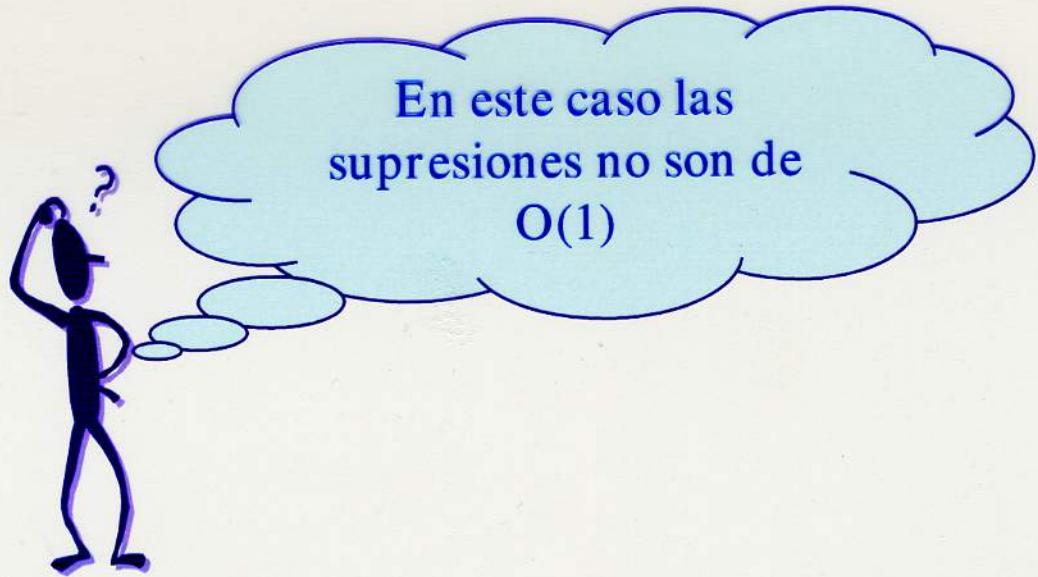
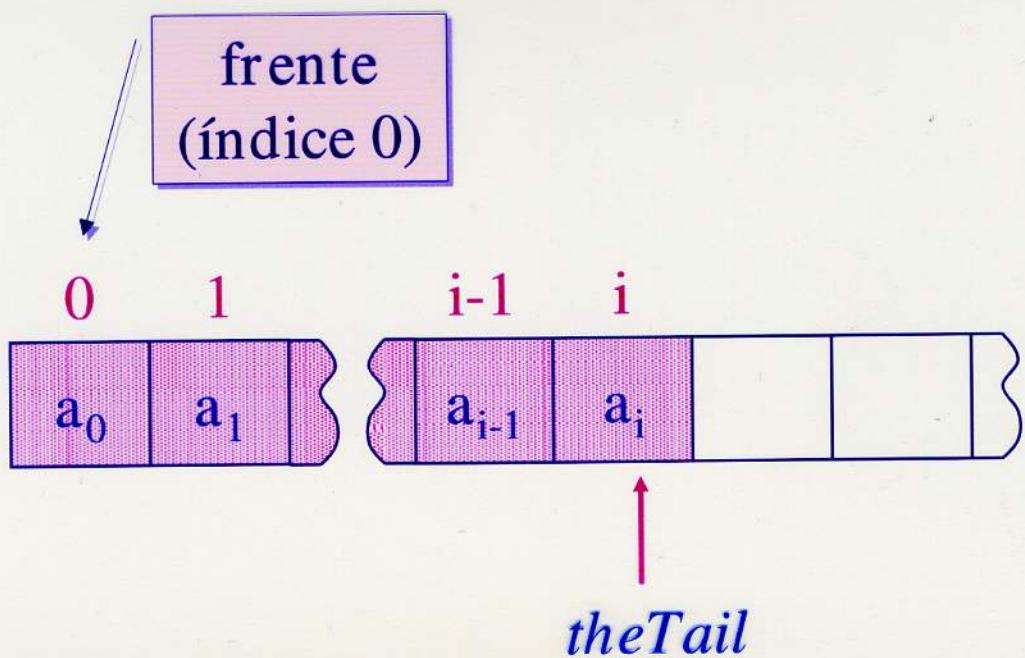
Colas

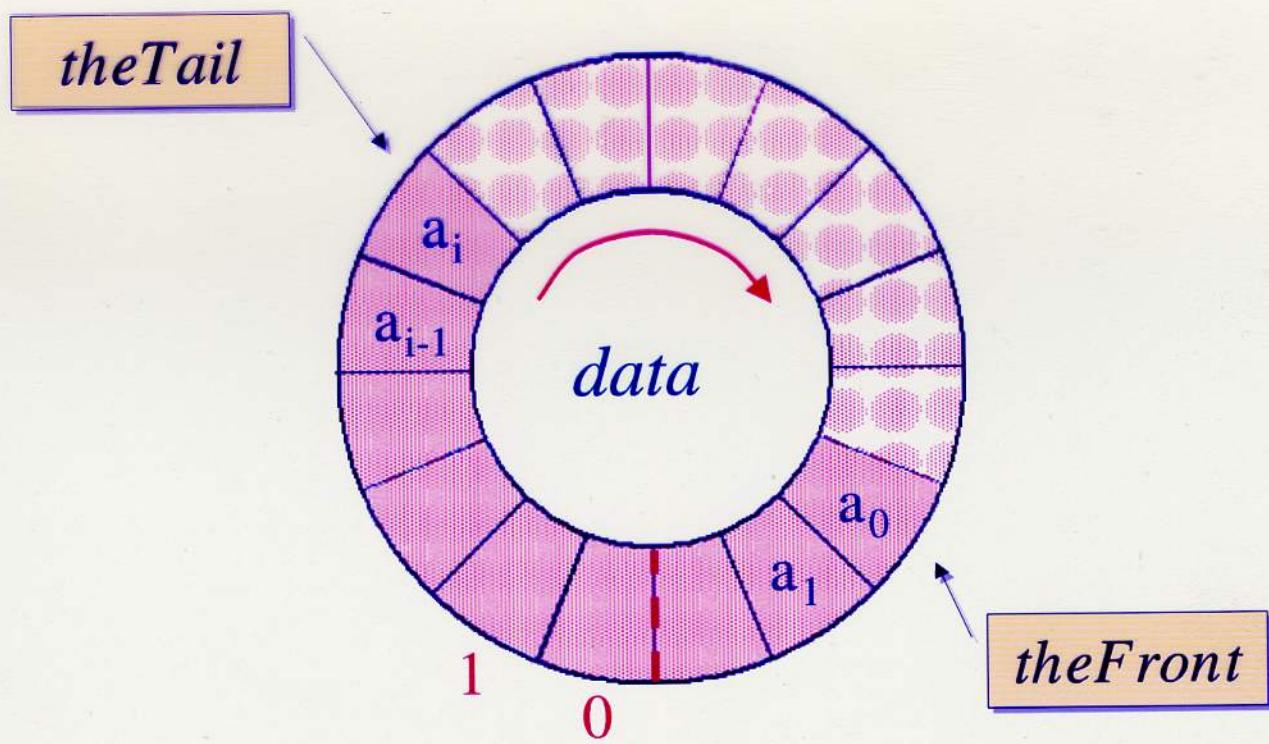
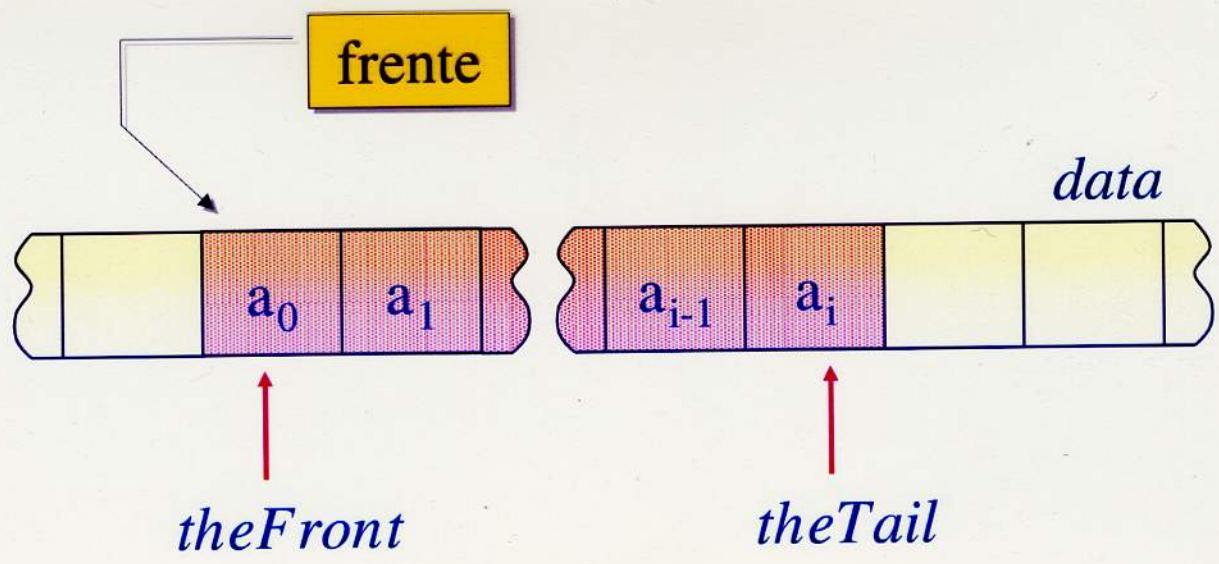
Una cola es una estructura de datos lineal en la que los elementos se suprimen e insertan por extremos opuestos.

También se las denomina estructuras FIFO (*First In First Out*).



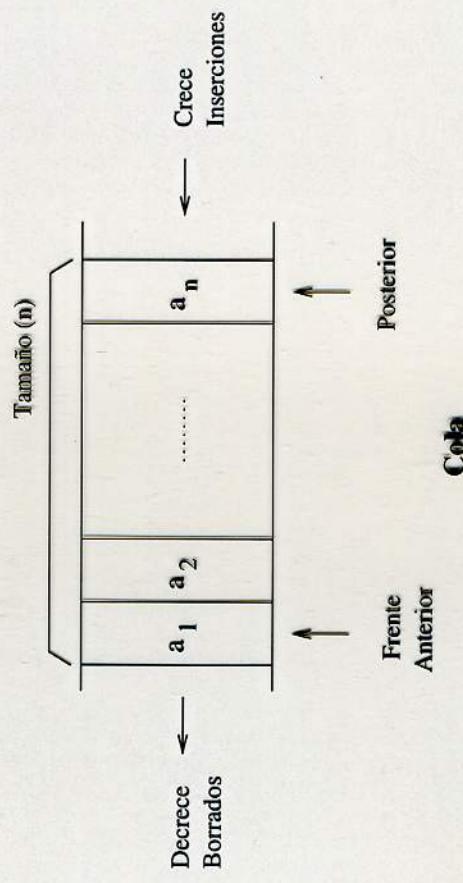






Colas

Una *Cola* es un tipo de dato que *contiene una secuencia de valores, especialmente diseñado para realizar inserciones en uno de los extremos, mientras los borrados y accesos se realizan en el otro. Por ello, se suele representar*



También denominadas listas FIFO, ya que el primero en entrar es el primero en salir. Los accesos a los elementos de la Cola se *realizan* por un extremo, denominado *frente*. Las operaciones básicas son:

- *Frente*. Devuelve el elemento del frente
- *Poner*. Añade un elemento al final de la cola.
- *Quitar*. Elimina el elemento del frente.
- *Vacia*. Indica si la cola está vacía.

Colas

Esquema de cola

```
#ifndef __COLA_H__
#define __COLA_H__

class Cola{
    ...
public:
    Cola();
    Cola(const Cola& p);
    ~Cola();
    Cola& operator= (const Cola& p);
    bool vacia() const;
    void poner (char c);
    void quitar();
    char frente() const;
};

#endif
```

Una posible clase *Cola* para almacenar datos de tipo *char* puede tener la siguiente sintaxis.

Uso de una cola

```
#include <iostream>
#include <pila.h>
#include <cola.h>
using namespace std;

int main()
{
    Pila p;
    Cola q;
    char dato;

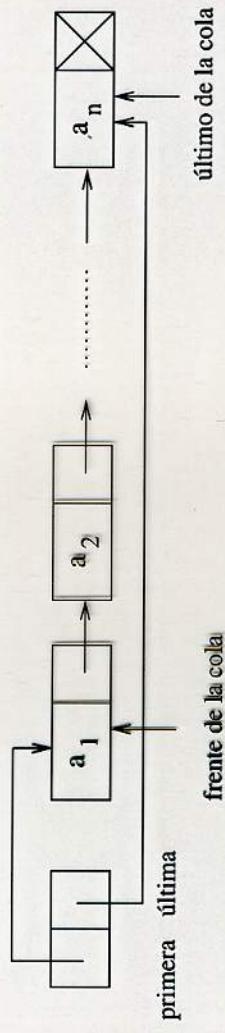
    cout << "Escriba una frase" << endl;
    while ((dato==cin.get())!= '\n')
        if (dato! = ' ')
            p.poner(dato);
            q.poner(dato);

    bool palindromo=true;
    while (!p.vacia() && palindromo) {
        if (p.tope()!=q.frente())
            palindromo= false;
        p.quitar();
        q.quitar();
    }

    if (palindromo)
        cout << "La frase es un palíndromo" << endl;
    else cout << "La frase no es un palíndromo" << endl;
    return 0;
}
```

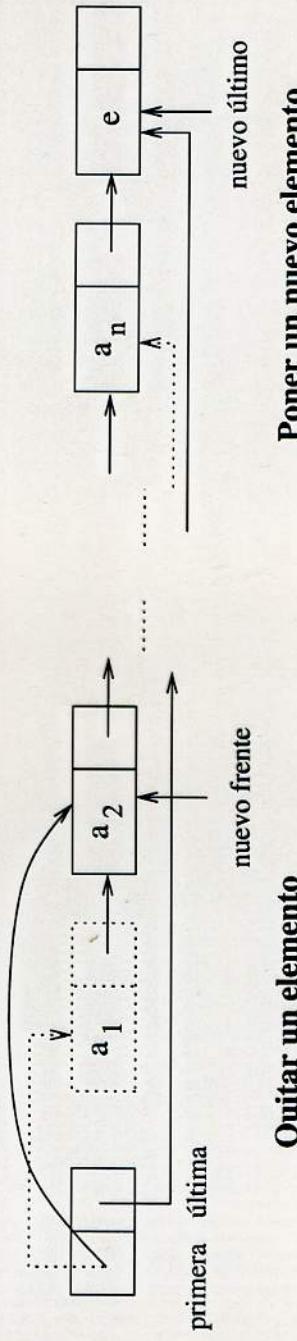
Colas (celdas enlazadas) (1/3).

Almacenamos la secuencia de valores *en celdas enlazadas*,



Colas (celdas enlazadas)

- Una cola vacía contiene un dos punteros *nulos*.
- El *frente* de la cola se encuentra en la primera celda. Así es más eficiente.
- Si *insertamos* un elemento, se añade una nueva celda *al final* y si lo *borrarmos*, eliminamos *la primera* celda.



Poner un nuevo elemento

Colas (celdas enlazadas) (2/3).

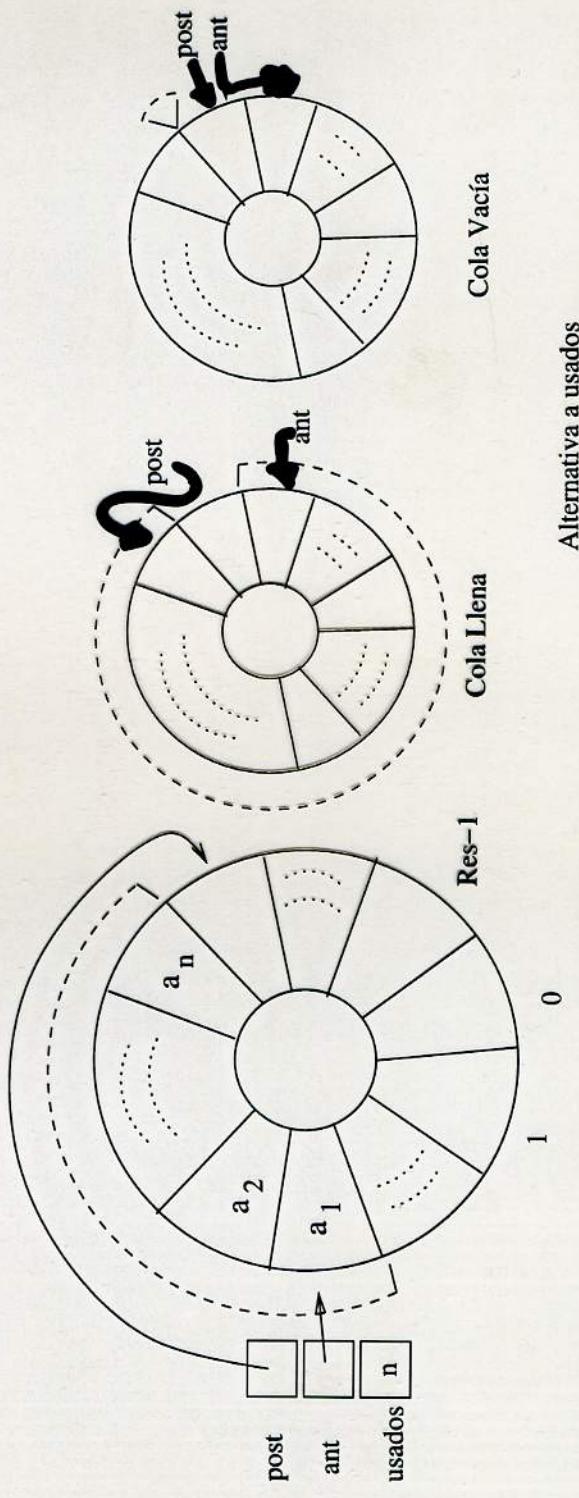
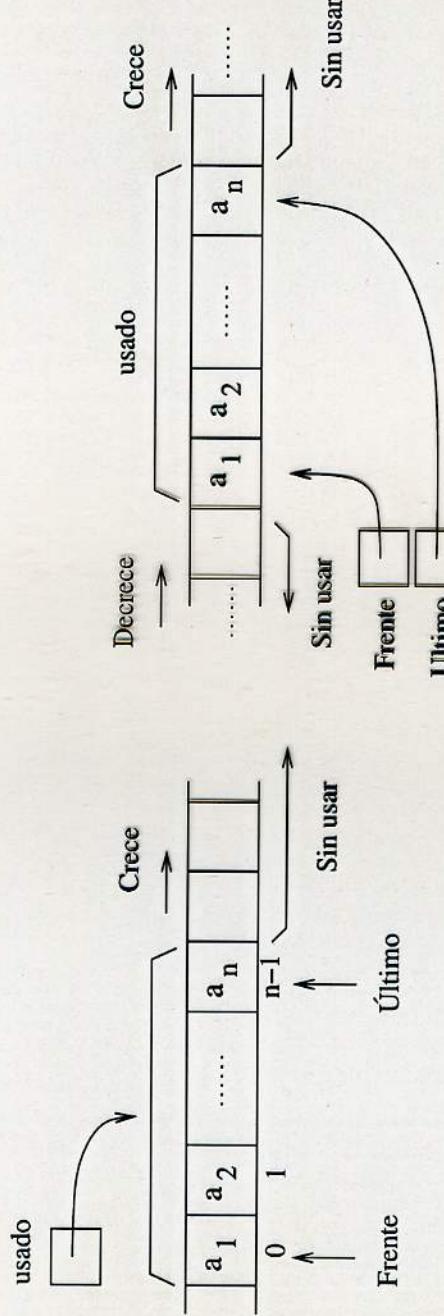
<i>Cola.h</i>	<i>Cola.cpp</i>
<pre>#ifndef __COLA_H__ #define __COLA_H__ typedef char Tbase; struct CeldaCola{ Tbase elemento; CeldaCola *sig; }; class Cola{ CeldaCola *primera; CeldaCola *ultima; public: Cola(); Cola(const Cola& p); ~Cola(); Cola& operator= (const Cola& p); bool vacia() const;{ return primera==0;} void poner (TBase c); void quitar(); Tbase frente() const; }; #endif</pre>	<pre>Cola::Cola(): primera(0), ultima(0) { } Cola::Cola(const Cola& c) { if (c.primera==0) primera= ultima= 0; else { primera= new CeldaCola; primera->elemento= c.primera->elemento; CeldaCola *src=c.primera; ultima=primera; while (src->sig!=0) { ultima->sig= new CeldaCola; src= src->sig; ultima= ultima->sig; ultima->elemento= src->elemento } ultima->sig=0; } } #endif</pre>

Colas (celdas enlazadas) (3/3).

Cola.cpp	Cola.cpp
<pre> Cola::~Cola() { CeldaCola *aux; while (primera! =0) { aux= primera; primera= primera→sig; delete aux; } } Cola& Cola::operator= (const Cola& c) { Cola caux(c); CeldaCola *aux; aux= this→primera; this→primera= caux→primera; caux→primera= aux; aux= this→ultima; this→ultima= caux→ultima; caux→ultima= aux; return *this; } </pre>	<pre> void Cola::poner(TBase c) { CeldaCola *aux= new CeldaCola; aux→elemento= c; aux→sig= 0; if (primera==0) primera=ultima=aux; else { ultima→sig= aux;; ultima= aux; } } void Cola::quitar() { assert (primera!=0); CeldaCola *aux= primera; primera= primera→sig; delete aux; if (primera==0) ultima=0; } TBase Cola::frente() const { assert (primera!=0); return primera→elemento; } </pre>

Colas (vectores) (1/3).

Almacenamos la secuencia de valores en un vector,



Colas (vectores) (2/3).

Cola.h	Cola.cpp
<pre>#ifndef __Cola_H__ #define __Cola_H__ typedef char Tbase; class Cola{ Tbase *datos; int reservados; int num_elem; int anterior, posterior; void resize(int n); public: Cola(); Cola(const Cola& p); ~Cola(); Cola& operator= (const Cola& p); bool vacia() const { return num_elem==0; } void poner(TBase c) void quitar(); Tbase frente() const; }; #endif</pre>	<pre>void Cola::resize(int n) { assert(n>=num_elem && n>0); Tbase *aux= new Tbase[n]; for (int i=0;i<num_elem;++) aux[i]= datos[(anterior+i)%reservados]; anterior=0; posterior= anterior+ num_elem; delete[] datos; datos= aux; reservados=n; } Cola::~Cola() { delete[] datos; } Cola::Cola() { datos= new Tbase[1]; reservados= 1; // siempre >= 1 anterior= posterior= num_elem= 0; }</pre>

Colas (vectores) (3/3).

Cola.cpp	Cola.cpp
<pre> Cola::Cola(const Cola& c) { reservados= c.reservados; datos= new Tbase[c.reservados]; for (int i=anterior;i!=posterior;i=(i+1)%reservados) datos[i]= c.datos[i]; anterior=c.anterior; posterior=c.posterior; num_elem=c.num_elem; } Cola& Cola::operator= (const Cola& c) { if (this! = &c) { delete[] datos; reservados=c.reservados; datos= new Tbase[c.reservados]; for (int i=anterior;i!=posterior;i=(i+1)%reservados) datos[i]= c.datos[i]; anterior=c.anterior; posterior=c.posterior; num_elem=c.num_elem; } return *this; } </pre>	<pre> void Cola::poner(TBase c) { if (num_elem==reservados) datos[anterior]=(2*reservados); datos[posterior]= c; posterior= (posterior+1)%reservados; num_elem++; } void Cola::quitar() { assert (num_elem!=0); anterior=(anterior+1)%reservados; num_elem--; if (num_elem<reservados/4) datos[anterior]=(reservados/2); } TBase Cola::frente() const { assert (num_elem!=0); return datos[anterior]; } </pre>

Listas

Las listas son las estructuras de datos lineales más generales posibles. Los elementos son accesibles y se pueden insertar y suprimir en cualquiera de los extremos de la estructura o en una posición intermedia de la misma.

Matemáticamente, una lista es una secuencia de cero o más elementos de un mismo tipo:

$$a_1 \ a_2 \ a_3 \dots a_n$$

donde $n \geq 0$.

El número de elementos n se denomina longitud de la lista.

Se dice que a_1 es el primer elemento de la lista y a_n el último.

Si la lista no tiene elementos ($n=0$) se dice que es vacía.

Listas (cont.)

Una propiedad importante de las listas es que los elementos siguen un orden lineal de acuerdo a su **posición** en la lista:

$$a_1 \ a_2 \ a_3 \dots a_n$$

Se dice que:

a_i está en la posición i para $i=1, 2, \dots, n$

a_i precede a a_{i+1} para $i=1, 2, \dots, n-1$

a_i sigue a a_{i-1} para $i=2, 3, \dots, n$

Es necesario identificar de alguna manera una posición no válida o fuera de la lista.

Operaciones con listas

Lista vacía: ()

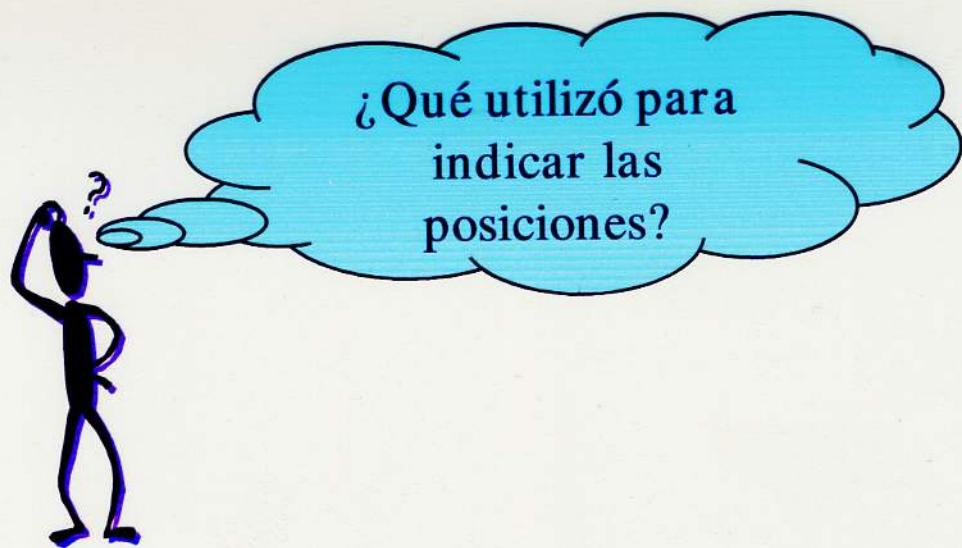
Lista con 5 elementos: (a1, a2, a3, a4, a5)

Insertar x en la posición 2:

(a1, x, a2, a3, a4, a5)

Borrar elemento de la posición 4:

(a1, x, a2, a4, a5)



Mecanismos para indicar posiciones

- ☞ Usar un índice i ($1 \leq i \leq n$) que indique el número de orden de los elementos.
 - No precisa conocer la representación interna.
 - A priori esta solución parece que podría dar lugar a que las operaciones de inserción y borrado fueran de $O(n)$, pero este problema se puede obviar fácilmente en la implementación.
- ☞ Usar tipos de datos indicando donde se almacena el elemento i -ésimo de la lista.
 - Precisa conocer la representación interna, que va en contra del principio de ocultación
- ☞ Adoptar la idea de la primera opción pero utilizando un iterador.

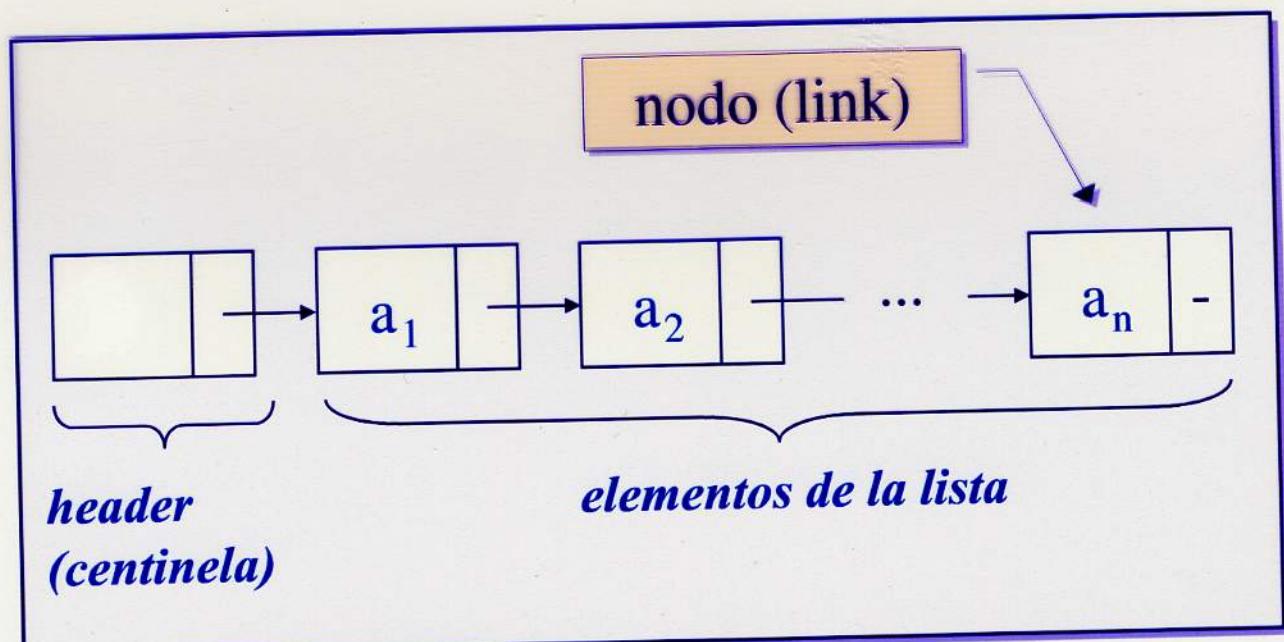
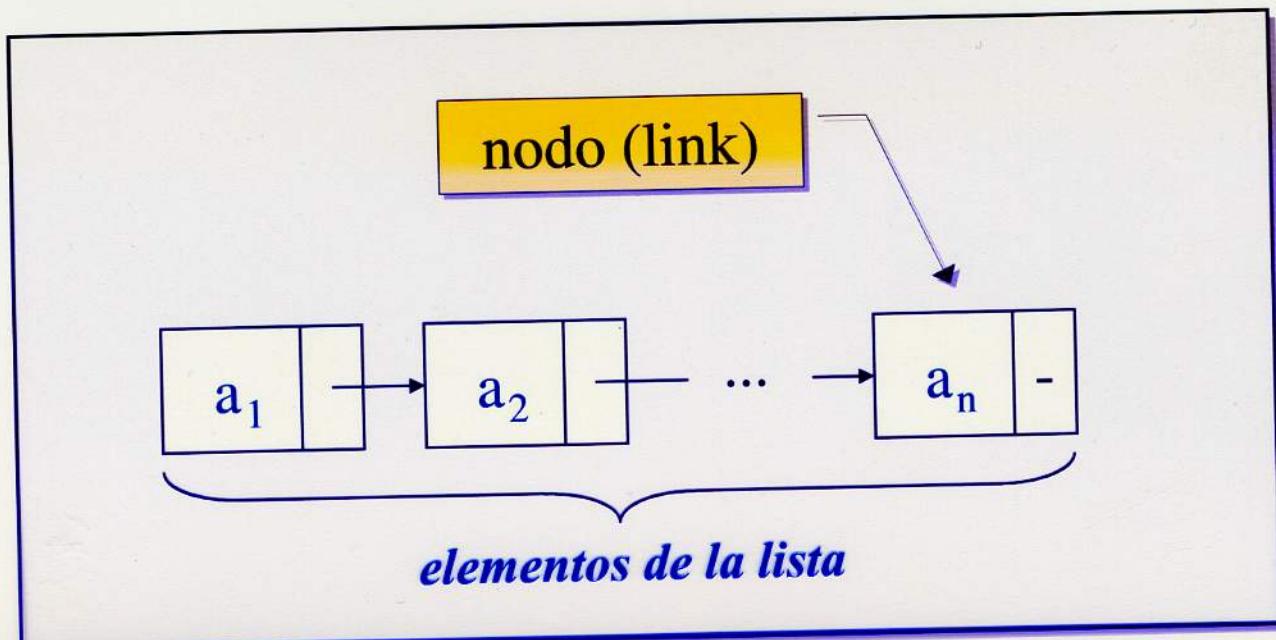
Estructura de un nodo simple

<i>información</i>	<i>dirección</i>
valor del elemento	La dirección de otro nodo <i>(nodo siguiente)</i>

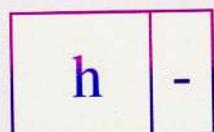


*Un solo
enlace*

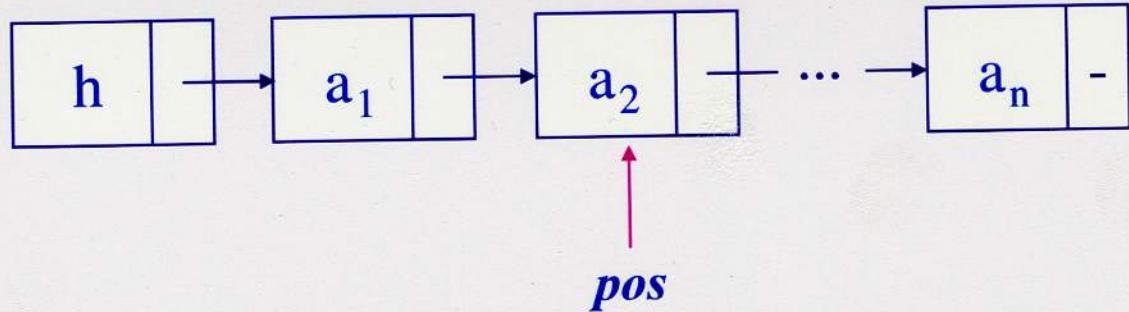
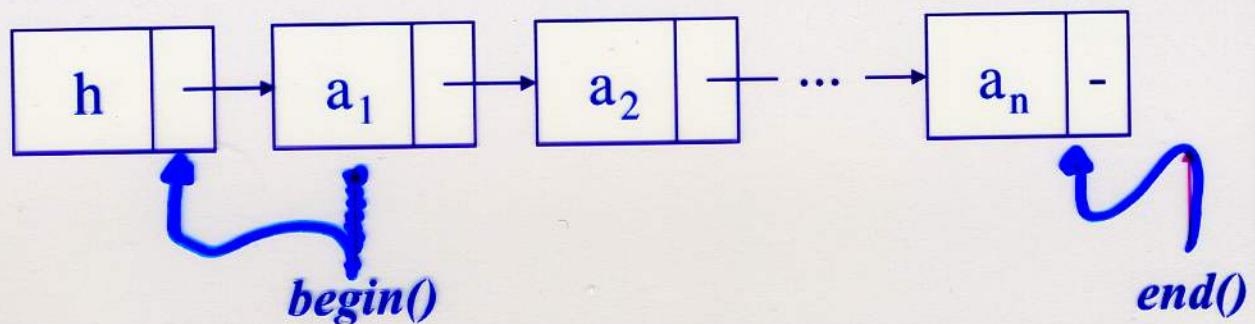
Listas enlazadas (un solo enlace)



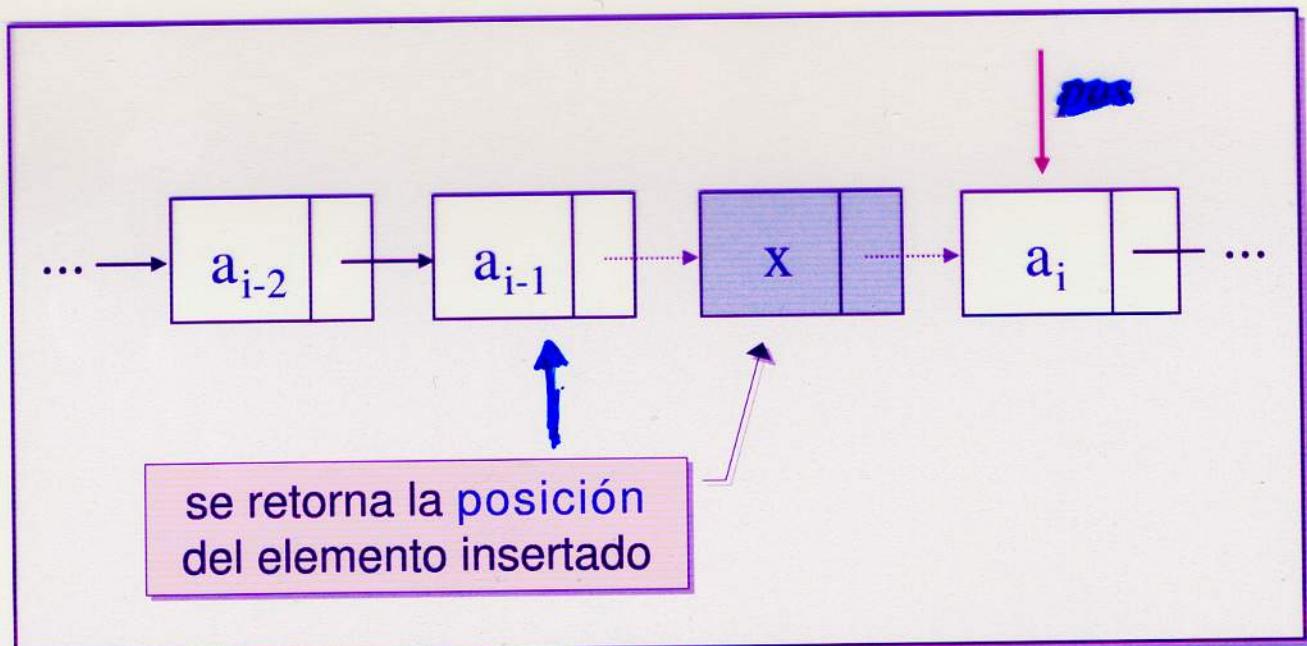
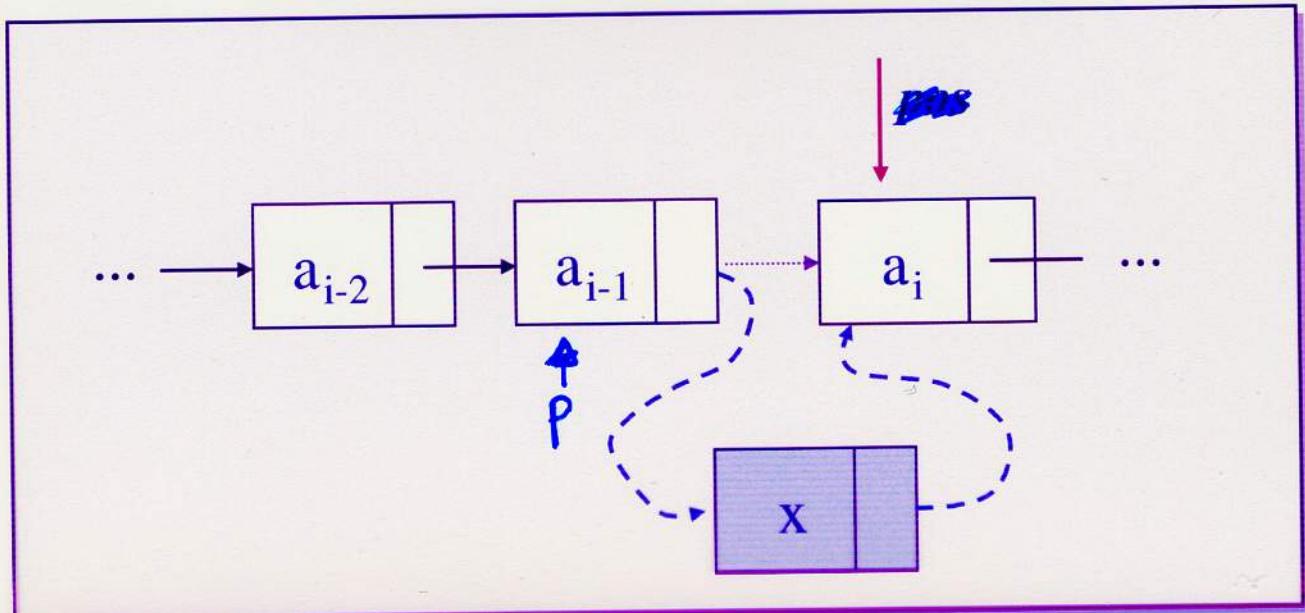
Listas enlazadas (cont.)



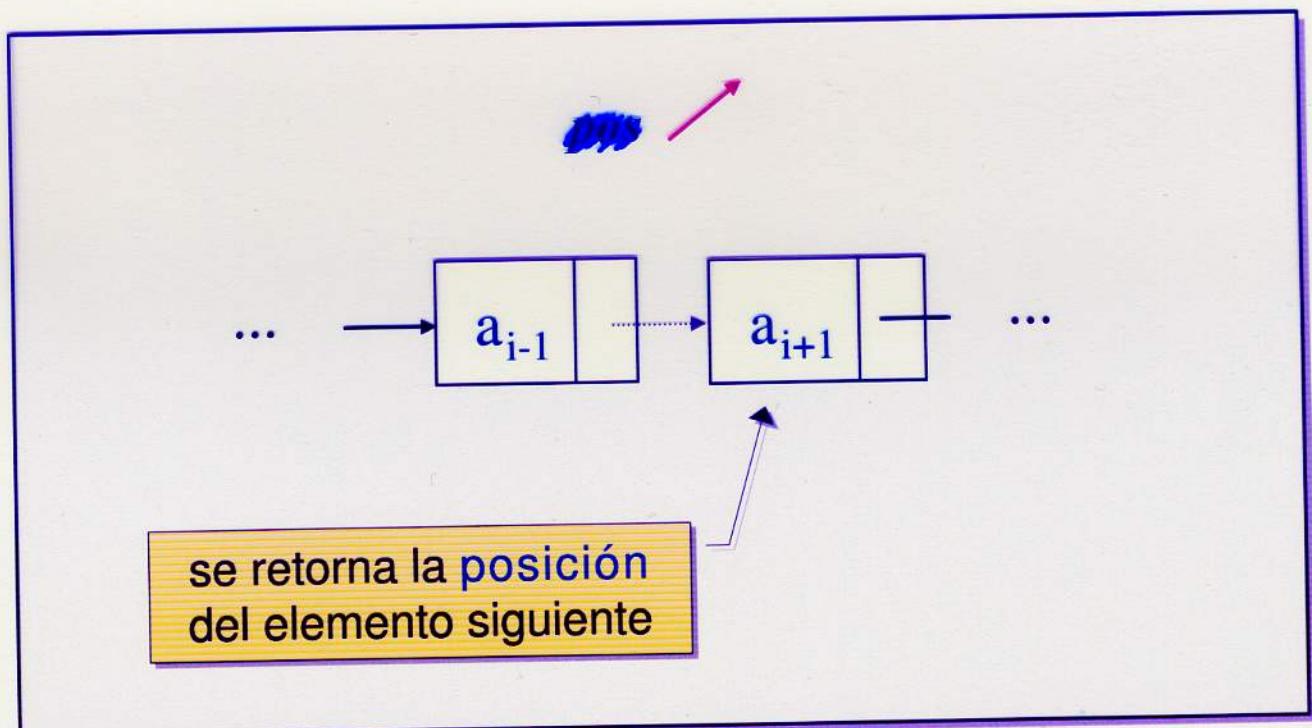
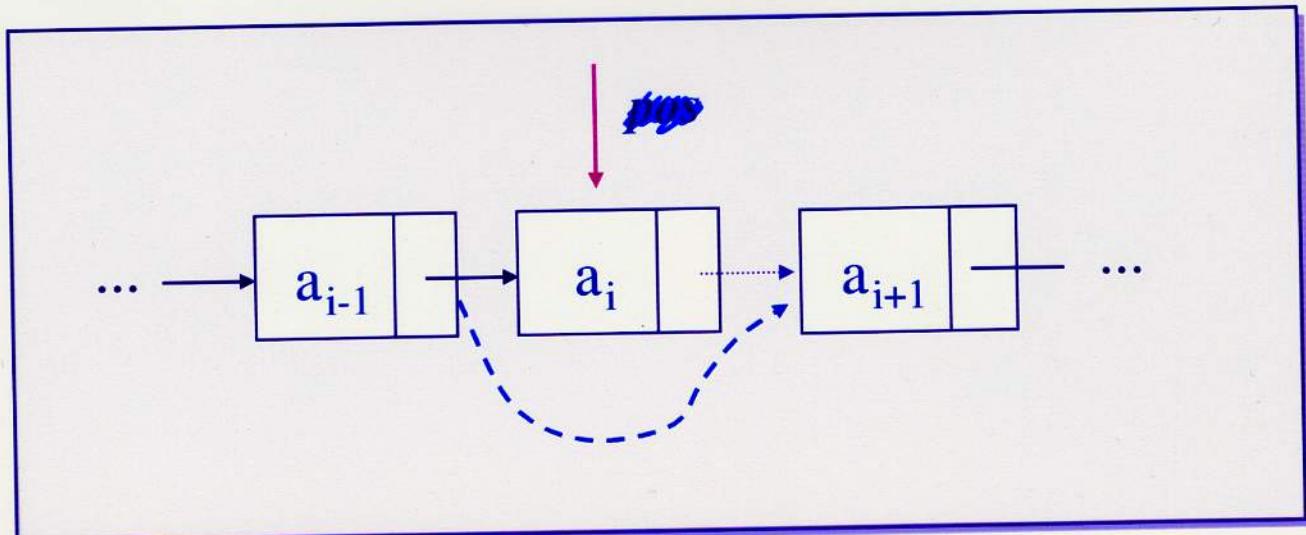
Lista vacía



Inserción en listas enlazadas

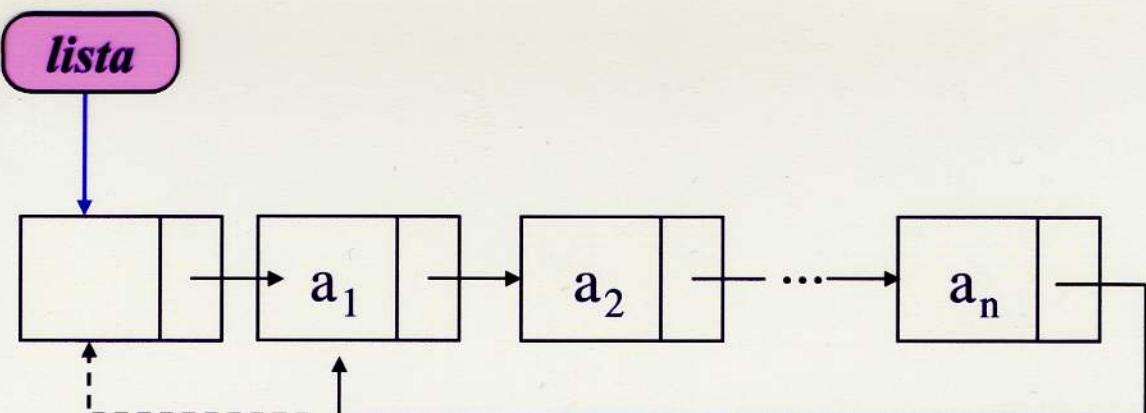
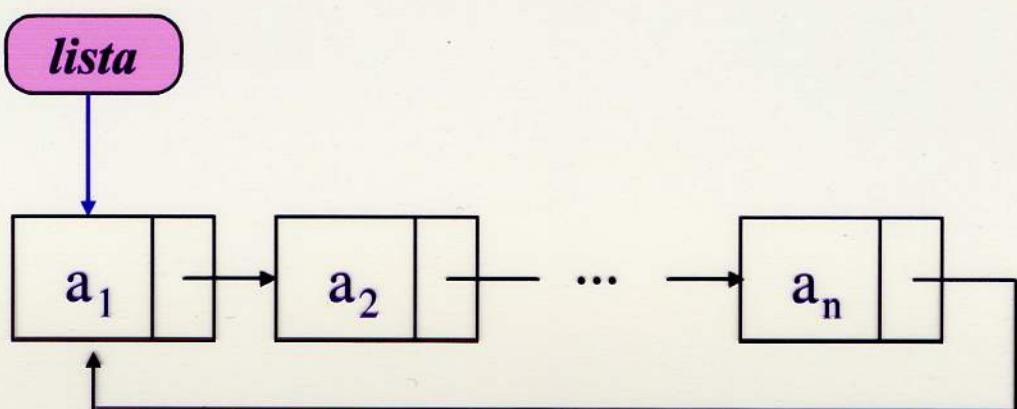


Borrado en listas enlazadas



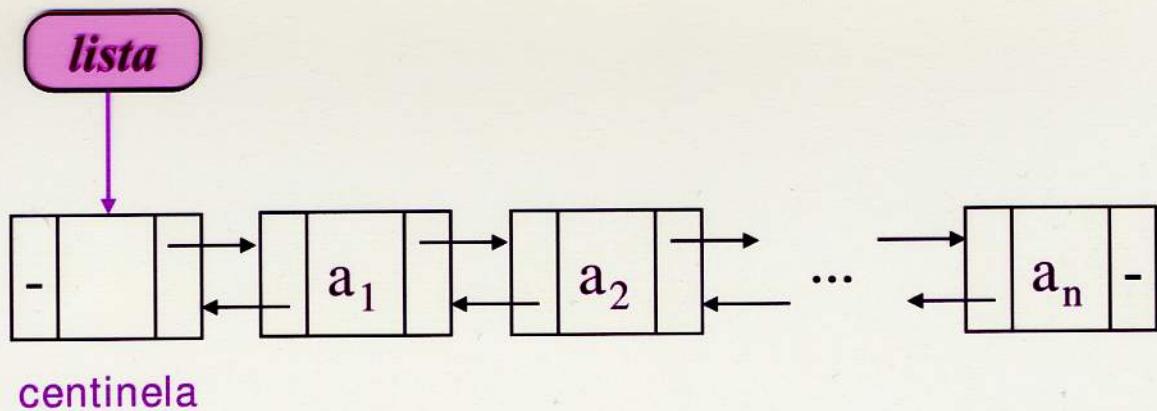
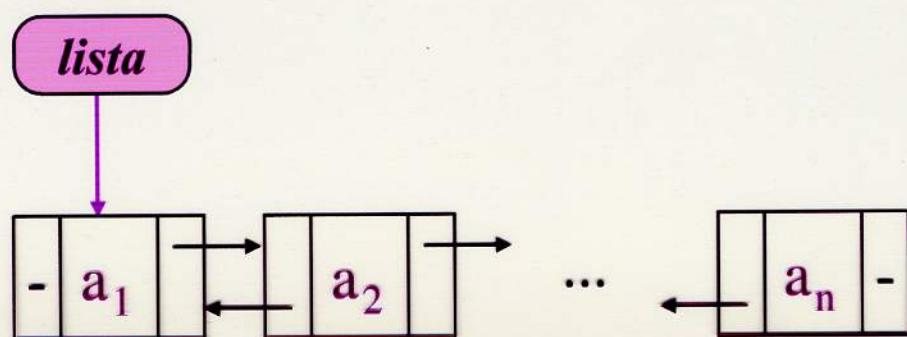
Otras variantes de listas

Listas circulares



Otras variantes de listas (cont.)

Listas doblemente enlazadas



Listas.

Una *Lista* es un tipo de dato que contiene una secuencia de elementos, especialmente diseñado para realizar *inserciones*, *borrados* y *accesos en cualquier parte*.

Una *lista* la podemos representar

$$< a_1, a_2, \dots, a_n >$$

Las operaciones básicas son:

- *Set.* Modifica un elemento de una posición.
- *Get.* Devuelve un elemento de una posición.
- *Borrar.* Eliminamos el elemento de una posición.
- *Insertar.* Insertamos un elemento en una posición.
- *Num_elements.* Devuelve el número de elementos en la lista.

Consideremos la función *Insertar*. Para cubrir todas las posibilidades, una lista con n elementos contendrá $n+1$ posiciones. Desde la primera, hasta la *siguiente a la última*, que denominaremos *posición fin de la lista*.

Listas (primera aproximación).

Implementación de Lista	Comentarios
<pre>Una posible clase Lista implementada #ifndef __LISTA_H__ #define __LISTA_H__</pre> <p>typedef char Tbase;</p> <pre>class Lista{ ... public: Lista(); Lista(const Lista& l); ~Lista(); Lista& operator= (const Lista& l);</pre>	<p>Para <i>evaluar el interface</i> en esta propuesta, pensemos en varias implementaciones y la eficiencia derivada:</p> <ul style="list-style-type: none"> ■ <i>Vectores</i>. Esta <i>implementación</i> parece <i>sencilla</i>, ya que las posiciones enteras que se pasan a las funciones se traducen directamente en posiciones en un vector. Las <i>funciones de inserción y borrado</i> serían bastante <i>ineficientes (orden lineal)</i>. ■ <i>Celdas enlazadas</i>. Esta <i>implementación</i> parece más eficiente, ya que los <i>borrados e inserciones</i> se pueden realizar <i>sin desplazar elementos</i>. Sin embargo, las funciones <i>set, get, ...</i> son de <i>orden lineal</i>. El problema es que <i>un entero es una mala solución para representar una posición en una lista de celdas</i>. <p>Por tanto, la <i>posición</i> en una lista <i>debería variar</i> dependiendo de la <i>implementación</i>. Para eliminar esta dependencia, podemos <i>crear una abstracción</i> de lo que es una posición, encapsulando <i>esa variabilidad</i> en una clase. La solución puede ser</p> <ul style="list-style-type: none"> ■ Crear una clase <i>Posicion</i>. Un objeto representa una posición en una determinada lista. <ul style="list-style-type: none"> ● Para el caso de un vector, se implementa como <i>un entero</i>, ● Para el caso de <i>celdas enlazadas</i>, se puede representar como <i>un puntero</i>.

Listas (Clases Posicion y Lista).

<i>Lista.h</i>	<i>Lista.h</i>	
<pre>#ifndef __LISTA_H__ #define __LISTA_H__ typedef char Tbase; class Posicion { ... public: Posicion(); Posicion(const Posicion& p); ~Posicion(); Posicion& operator= (const Posicion& p); Posicion& operator++(); Posicion& operator--(); bool operator==(const Posicion& p); bool operator!=(const Posicion& p); }; #endif</pre>	<pre>// Una implementación class Lista{ ... public: Lista(); Lista(const Lista& l); ~Lista(); Lista& operator= (const Lista& l); void set (Posicion p, Tbase e); Tbase get (Posicion p) const; Posicion insertar(Posicion p, Tbase e); Posicion borrar(Posicion p); Posicion begin() const; Posicion end() const; }; #endif</pre>	<ul style="list-style-type: none"> ■ <i>Insertar</i> y <i>borrar</i> se modifican. ■ <i>Num_elementos</i> no es fundamental. ■ Es necesario conocer donde comienza y acaba una lista. Las funciones <ul style="list-style-type: none"> • <i>Begin</i>. Devuelve la posición del <i>primer elemento</i>. • <i>End</i>. Devuelve la posición del elemento <i>detrás del último</i> (donde se añadiría un elemento). ■ En una <i>lista vacía</i>, la posición <i>begin</i> coincide con <i>end</i>.

Listas (uso) (1/2).

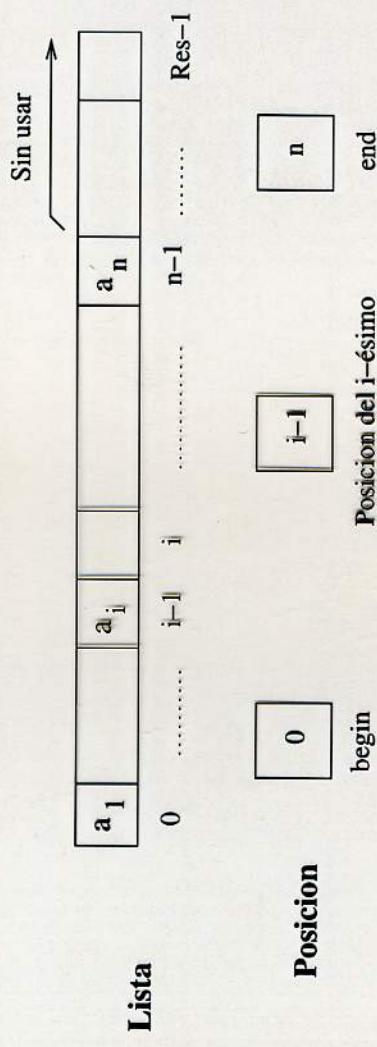
Uso de lista	Uso de lista	Uso de lista
<pre> bool vacia(const Lista& l) { return l.begin()==l.end(); } int numero_elementos(const Lista& l) { int n=0; for (Posicion p=l.begin(); p!=l.end(); ++p) n++; return n; } void todo_minuscula(Lista &l) { for (Posicion p=l.begin(); p!=l.end(); ++p) l.set(p, tolower(l.get(p))); } void escribir(const Lista& l) { for (Posicion p=l.begin(); p!=l.end(); ++p) cout << l.get(p); cout << endl; } void escribir_minuscula (Lista l) { todo_minuscula(l); escribir(l); } </pre>	<pre> void borrar_caracter (Lista& l, char c) { Posicion p=l.begin(); while (p!=l.end()) if (l.get(p)==c) p=l.borrar(p); else ++p; } Lista al_reves(const Lista& l) { Lista aux; for (Posicion p=l.begin(); p!=l.end(); ++p) aux.insertar(aux.begin(),l.get(p)); return aux; } Posicion localizar(const Lista& l, char c) { for (Posicion p=l.begin(); p!=l.end(); ++p) if (l.get(p)==c) return p; return l.end(); } </pre>	

Listas (uso) (2/2).

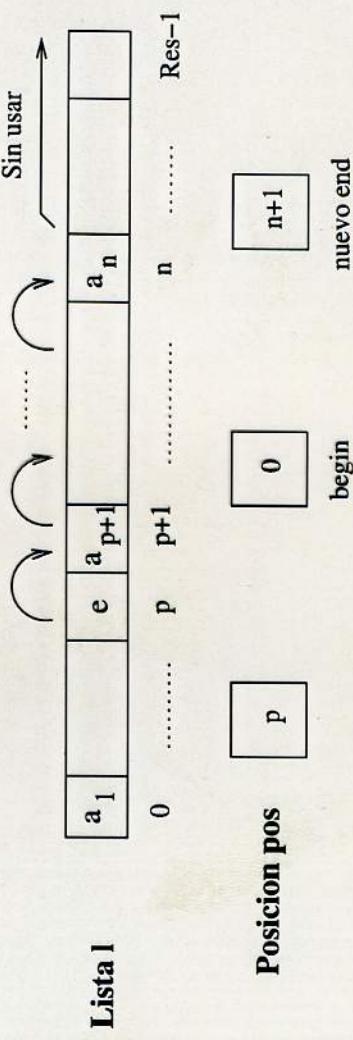
Uso de lista	Uso de lista
<pre> int numero_elementos(const Lista& l); void borrar_caracter(Lista& l, char c); void todo_minuscula(Lista &l); bool palindromo (const Lista& l) { Lista laux(l); int n= numero_elementos(l); if (n<2) return true; borrar_caracter(laux, ' '); todo_minuscula(laux); Posicion p1,p2; p1=laux.begin(); p2=laux.end();--p2; for (int i=0;i<n/2;i++) { if (laux.get(p1)! =laux.get(p2)) return false; ++p1;--p2; } return true; } </pre>	<pre> int main() { char dato; Lista l; cout << endl << "Escriba una frase" << endl; while ((dato==cin.get())!= '\n') l.insertar(l.end(),dato); cout << endl << "La frase introducida es:" << endl; escribir (l); cout << endl << "La frase en minúscula:" << endl; escribir_minuscula (l); if (localizar(l, ',')==l.end()) cout << endl << "La frase no tiene espacios." << endl; else { cout << endl << "La frase sin espacios:" << endl; Lista aux(l); borrar_caracter(aux, ' '); escribir (aux); } cout << endl << "La frase al revés:" << endl; escribir (al_reves(l)); if (palindromo(l)) cout << endl << "Es un palíndromo" << endl; else cout << endl << "No es un palíndromo" << endl; } return 0; } </pre> <p>Note que <i>no sería válido</i> mover las posiciones “mientras” la primera sea “menor” que la segunda.</p>

Listas (vectores) (1/3).

Almacenamos la secuencia de valores, la *lista*, en un vector, y controlamos cada posición con un entero



- La posición *begin* corresponde al entero *zero*.
- La posición *end* corresponde al entero detrás del último (*n*).
- Si insertamos un elemento, se desplazan los elementos a la derecha, y si lo borramos, se desplazan a la izquierda.



Insertar un elemento en pos de la lista 1

Listas (vectores) (2/3).

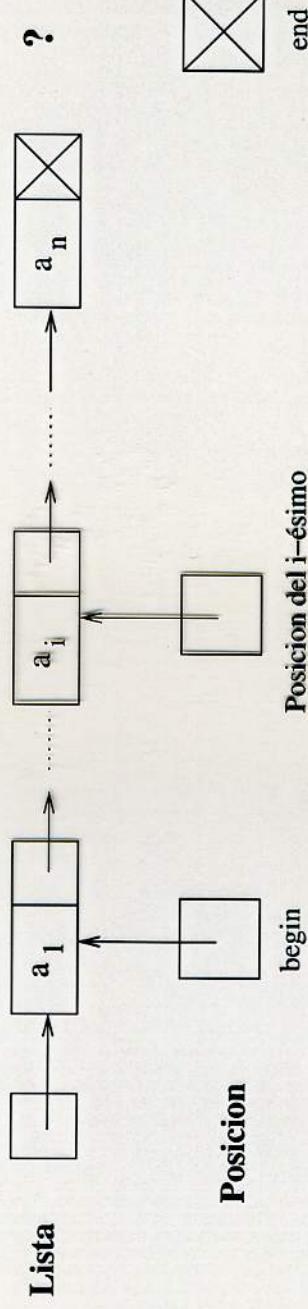
<i>Lista.h</i>	<i>Lista.h</i>
<pre>#ifndef __LISTA_H__ #define __LISTA_H__ typedef char Tbase; class Lista; class Posicion { int i; public: Posicion(): i(0) {} //Posicion(const Posicion& p); //~Posicion(); //Posicion& operator= (const Posicion& p); Posicion& operator++() { ++i; return *this; } Posicion& operator--() { --i; return *this; } bool operator==(const Posicion& p) { return i==p.i; } bool operator!= (const Posicion& p) { return i!=p.i; } friend class Lista; };</pre>	<pre>class Lista{ Tbase *datos; int nelementos; int reservados; void resize(int n); public: Lista(): nelementos(0),reservados(1) { datos= new Tbase[1]; } Lista(const Lista& l); ~Lista() { delete[] datos; } Lista& operator=(const Lista& l); void set (Posicion p, Tbase e) { assert(p.i>=0 && p.i<nelementos); datos[p.i]= e; } Tbase get (Posicion p) const { assert(p.i>=0 && p.i<nelementos); return datos[p.i]; } Posicion insertar(Posicion p, Tbase e); Posicion borrar(Posicion p); Posicion begin() const { Posicion p; p.i=0; return p; } Posicion end() const { Posicion p; p.i=nelementos; return p; } }; #endif</pre>

Listas (vectores) (3/3).

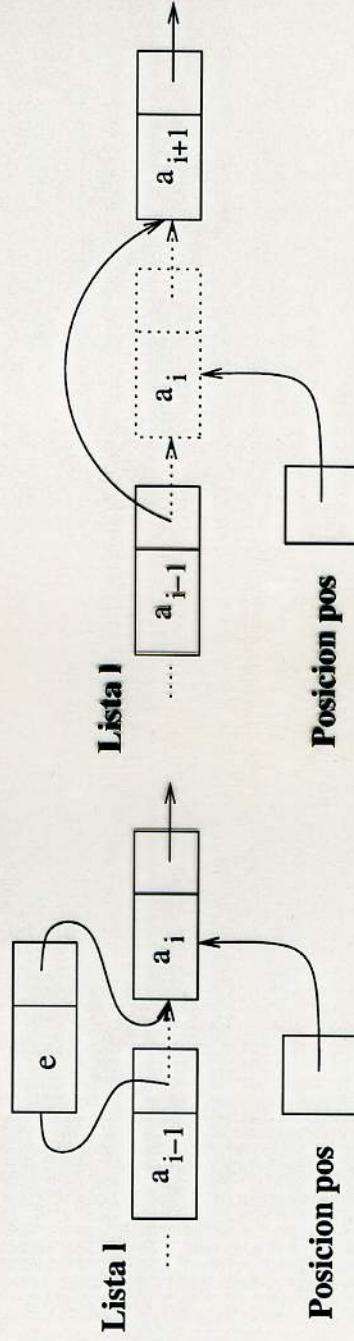
Lista.cpp	Lista.cpp
<pre> void Lista::resize(int n) { assert(n>=nelementos && n>0); Tbase *aux= new Tbase[n]; for (int i=0;i<nelementos;++) aux[i]= datos[i]; delete[] datos; datos= aux; reservados= n; } Lista::Lista(const Lista& l){ datos= new Tbase[l.reservados]; nelementos=l.nelementos; reservados=l.reservados; for (int i=0;i<nelementos;i++) datos[i]=l.datos[i]; } Lista& Lista::operator= (const Lista& l){ Lista aux(); Tbase *paux; paux=datos; datos=l.datos; l.datos=paux; iaux=nelementos; nelementos=l.nelementos; l.nelementos=iaux; iaux=reservados; reservados=l.reservados; l.reservados=iaux; return *this; } </pre>	<pre> <i>Posición Lista::insertar(Posición p, Tbase e) {</i> if (nelementos==reservados) resize(2*reservados); for (int j=nelementos;j>p;j--) datos[j]= datos[j-1]; datos[p]= e; nelementos++; return p; } <i>// posición del insertado</i> <i>Posición Lista::borrar(Posición p){</i> assert (p!=end()); for (int j=p;j<nelementos-1;j++) datos[j]=datos[j+1]; nelementos--; if (nelementos<reservados/4) resize(reservados/2); return p; } <i>// posición del siguiente</i> <i>int</i> iaux; iaux=nelementos; nelementos=l.nelementos; l.nelementos=iaux; iaux=reservados; reservados=l.reservados; l.reservados=iaux; return *this; } </pre>

Listas (celdas enlazadas).

Almacenamos la secuencia de valores en *celdas enlazadas*, y controlamos cada posición con un *puntero a la celda*



- Una *lista* es un *puntero a la primera celda (nulo si es vacía)*.
- Una *posición* son *dos punteros*. El *segundo (no mostrado)* es *necesario para implementar operator -*.
- *Inserciones y borrados en la primera posición son casos especiales.*

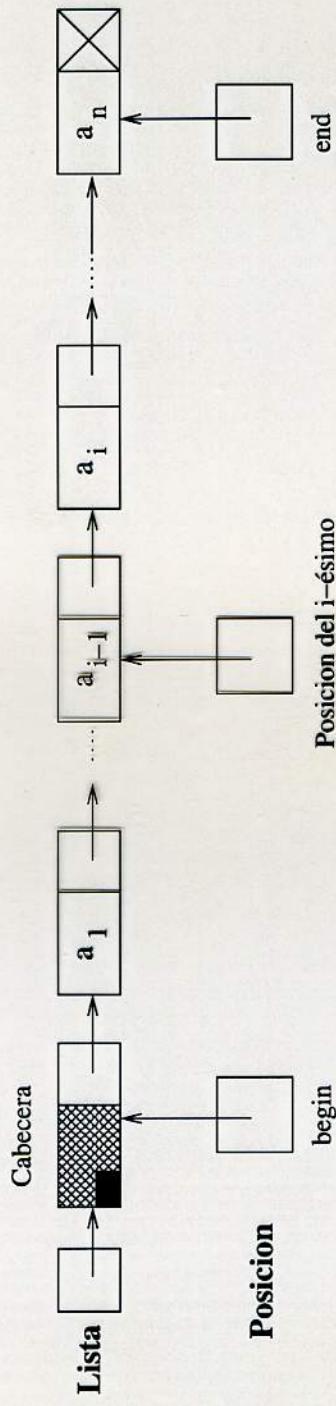


Insertar un elemento en pos de la lista l
(No en la primera posición)

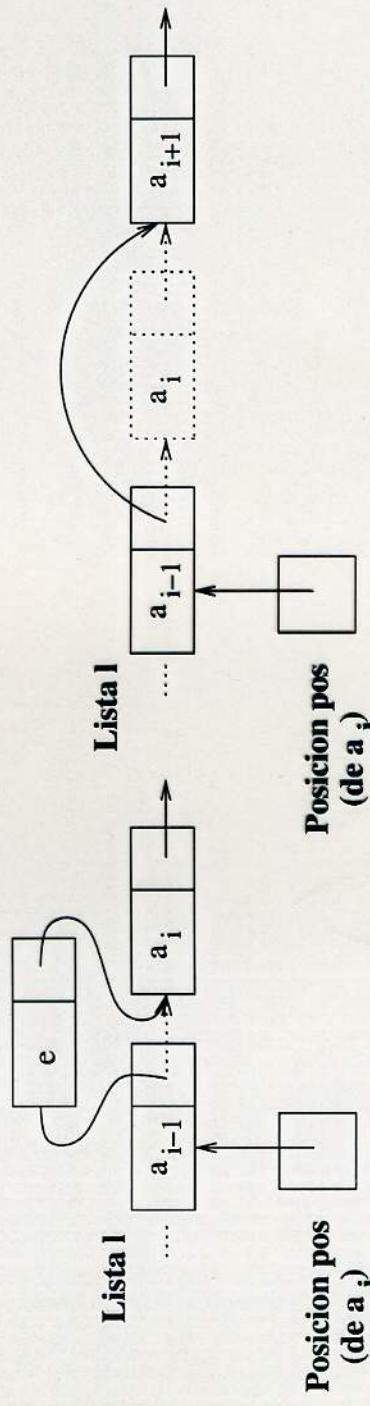
Borrar un elemento en pos de la lista l
(No en la primera posición)

Listas (celdas enlazadas con cabecera) (1/3).

Almacenamos la secuencia de valores en **celdas enlazadas**, y controlamos cada posición con un puntero a la celda anterior



- Una *lista* es un *puntero* a la *cabecera* (Si es vacía tiene *una celda*).
- Una *posición* son *dos punteros*. El *segundo* (no mostrado) es *necesario* para implementar *operator=*.
- *Inserciones* y *borrados* en la *primera posición* *NO* son *casos especiales*.



Insertar un elemento en pos de la lista 1

Borrar un elemento en pos de la lista 1

Listas (celdas enlazadas con cabecera) (2/3).

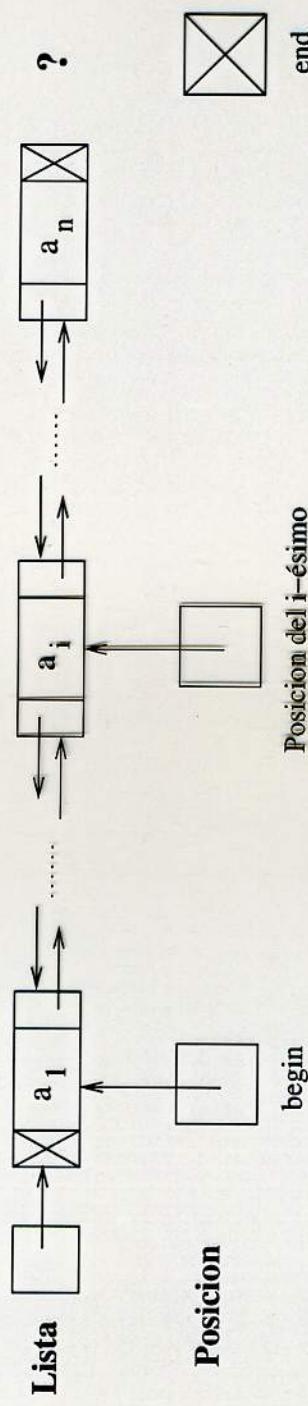
Lista.h	Lista.h
<pre>#ifndef __LISTA_H__ #define __LISTA_H__ typedef char Tbase; struct CeldaLista{ Tbase elemento; CeldaLista *siguiente; }; class Lista;</pre>	<pre>class Lista{ CeldaLista *cab; CeldaLista *ultima; public: Lista() { ultima= cab= new CeldaLista; cab->siguiente= 0; } Lista(const Lista&); ~Lista(); Lista& operator= (const Lista&); void set (Posicion p, Tbase e) { p.puntero->siguiente->elemento= e; } Tbase get (Posicion p) const { return p.puntero->siguiente->elemento; } Posicion insertar(Posicion p, Tbase e); Posicion borrar(Posicion p); Posicion begin() const { Posicion p; p.puntero= p.primer= cab; return p; } Posicion end() const { Posicion p; p.puntero= ultima; p.primer= cab; return p; } }; Position& operator--(); bool operator==(const Posicion& p) { return puntero==p.puntero; } bool operator!= (const Posicion& p) { return puntero!=p.puntero; } friend class Lista;</pre>

Listas (celdas enlazadas con cabecera) (3/3).

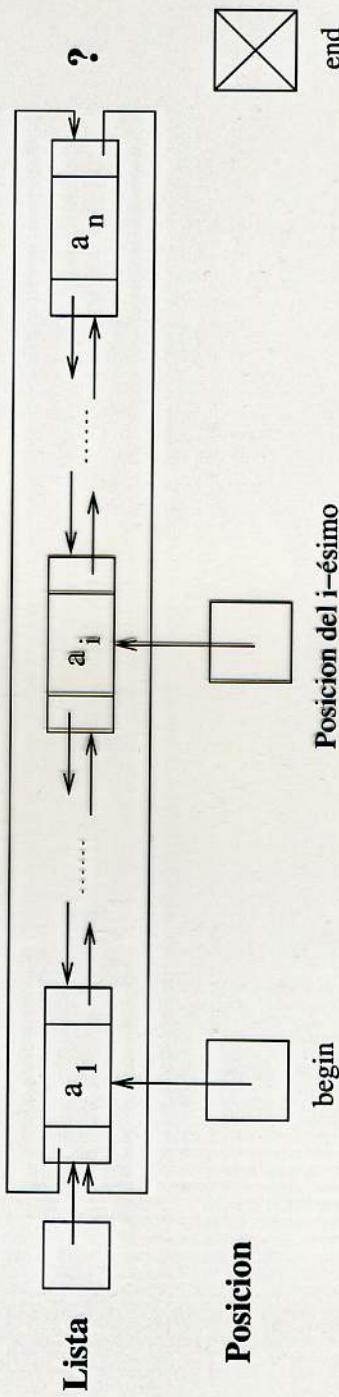
Lista.cpp	Lista.cpp
<pre> Posicion& Posicion::operator--() { assert(puntero!=primera); CeldaLista *aux=primera; while(aux->siguiente!=puntero) aux=aux->siguiente; puntero= aux; return *this; } List::Lista(const Lista& l) { cab= new CeldaLista; CeldaCola *src=c.primera; ultima=cab; while (src->siguiente!=0) { ultima->siguiente= new CeldaLista; src= src->siguiente; ultima= ultima->siguiente; ultima->elemento= src->elemento } ultima->siguiente=0; } </pre>	<pre> Lista& Lista::operator=(const Lista& l){ Lista aux(); CeldaLista *p; p= this->cab; this->cab= aux->cab; aux->cab= p; p= this->ultima; this->ultima= aux->ultima; aux->ultima= p; return *this; } Posicion Lista::insertar(Posicion p, Tbase e) { CeldaLista *q= new CeldaLista; q->siguiente=p.puntero->siguiente; p.puntero->siguiente=q; q->elemento=e; if (p.puntero==ultima) ultima=q; return p; } // Al elemento insertado Posicion Lista::borrar(Posicion p){ assert (p!=end()); CeldaLista *q= p.puntero->siguiente; p.puntero->siguiente=q->siguiente; if (q==ultima) ultima=p.puntero; delete q; return p; } // Al elemento siguiente List::~Lista() { CeldaLista *aux; while (cab!=0) { aux=cab; cab=cab->siguiente; delete aux; } } </pre>

Listas (celdas doblemente enlazadas).

Almacenamos en *celdas doblemente enlazadas*, y controlamos la *posición* con un *puntero a la celda*



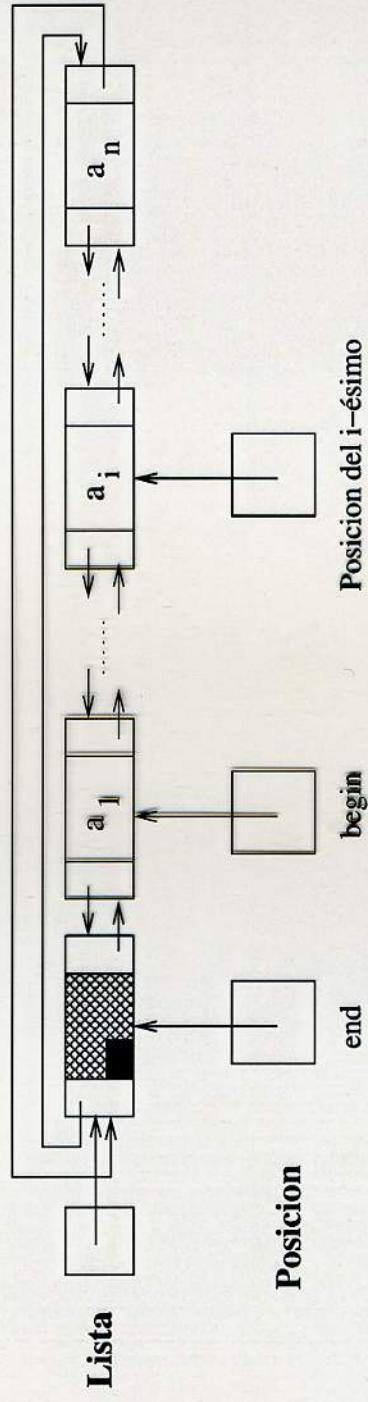
- Una *lista* es un *puntero* a la *primera celda* (*nulo* si es *vacía*).
- La posición *end* es *problemática*. Una *posición* son *dos punteros* para implementar *operator -*.
- La función *operator -* es eficiente, excepto para *end*, aunque se puede solucionar con la *circularidad*.



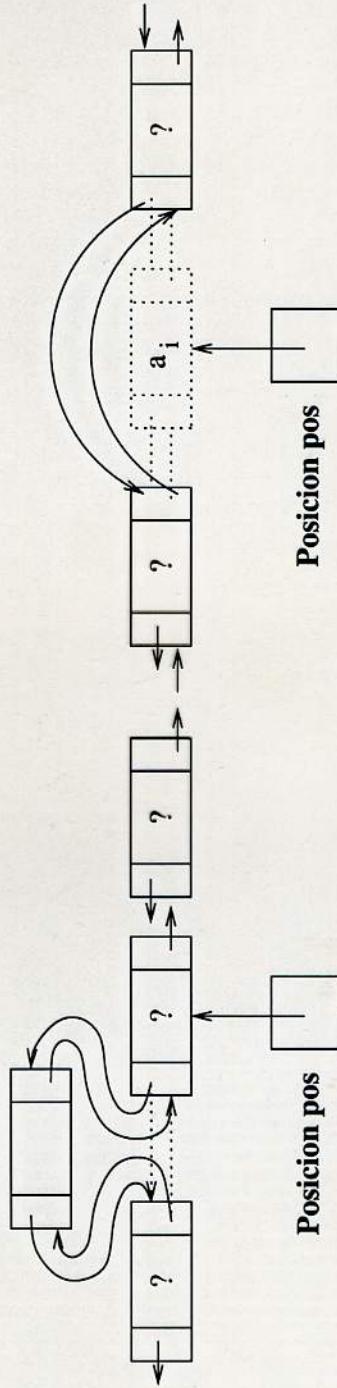
- Aunque la *representación* sigue requiriendo *dos punteros*.

Listas (celdas doblemente enlazadas con cabecera circulares).

Almacenamos la secuencia de valores en celdas doblemente enlazadas, y controlamos cada posición con un puntero a la celda



- Una lista es un puntero a la cabecera.
- Una posición es un único puntero a la celda.
- Las inserciones y borrados son independientes de la posición.



Insertar un elemento en pos de la lista I

Borrar un elemento en pos de la lista I

Listas (celdas doblemente enlazadas con cabecera circulares) (1/2).

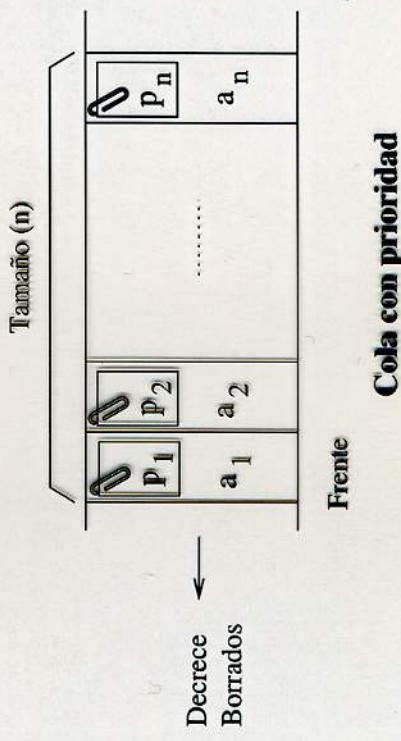
<pre>list.h</pre> <pre>typedef char Tbase; struct CeldaLista{ Tbase elemento; CeldaLista *anterior; CeldaLista *siguiente; }; class Lista; class Posicion { CeldaLista *puntero; public: Posicion(): puntero(0) {} //Posicion(const Posicion& p); //~Posicion(); //Posicion& operator=(const Posicion& p); Posicion& operator++() puntero=puntero->siguiente; return *this; } Posicion& operator--() puntero= puntero->anterior; return *this; } bool operator==(const Posicion& p) { return puntero==p.puntero; } bool operator!=(const Posicion& p) { return puntero!=p.puntero; } friend class Lista; };</pre>	<pre>list.h</pre> <pre>class Lista{ CeldaLista *cab; public: Lista(); Lista(const Lista& l); ~Lista(); Lista& operator=(const Lista& l); void set(Posicion p, Tbase e) { p.puntero->elemento=e; } Tbase get(Posicion p) const { return p.puntero->elemento; } Posicion insertar(Posicion p, Tbase e); Posicion borrar(Posicion p); Posicion begin() const { Posicion p; p.puntero=cab->siguiente; return p; } Posicion end() const { Posicion p; p.puntero=cab; return p; } };</pre>
---	---

Listas (celdas doblemente enlazadas con cabecera circulares) (2/2).

lista.cpp	lista.cpp
<pre> Lista::Lista() { cab = new CeldaLista; cab->siguiente= cab; cab->anterior=cab; } Lista::Lista(const Lista& l) { cab= new CeldaLista; cab->siguiente= cab; cab->anterior=cab; CeldaLista *p= l.cab->siguiente; while (p!=l.cab) { CeldaLista *q; q= new CeldaLista; q->elemento=p->elemento; q->anterior= cab->anterior; cab->anterior->siguiente= q; cab->anterior= q; q->siguiente =cab; p= p->siguiente; } } </pre>	<pre> Lista& Lista::operator= (const Lista& l){ Lista aux(l); CeldaLista *p; p= this->cab; this->cab= aux.cab; aux.cab=p; return *this; } Posicion Lista::insertar(Posicion p, Tbase e) { CeldaLista *q= new CeldaLista; q->anterior= p.puntero->anterior; q->siguiente=p.puntero; p.puntero->anterior= q; q->anterior->siguiente= q; q->elemento= e; p.puntero=q; return p; } Posicion Lista::borrar(Posicion p){ assert (p!=end()); CeldaLista *q= p.puntero; q->anterior->siguiente= q->siguiente; q->siguiente->anterior= q->anterior; p.puntero=q->siguiente; delete q; return p; } </pre>

Colas con prioridad

Una Cola es un tipo de dato que contiene una secuencia de valores, especialmente diseñado para realizar borrados y accesos en uno de los extremos, mientras la inserción se realiza en cualquier lugar, de acuerdo a un valor de prioridad. Se pueden representar



Los accesos y los borrados de elementos de la Cola con Prioridad se realizan por un extremo, denominado **frente**. El funcionamiento es muy parecido a las colas, excepto que podemos considerar que los elementos no mantienen el orden de inserción, sino el indicado por un valor de prioridad. Las operaciones básicas son:

- **Frente.** Devuelve el elemento del frente.
- **Prioridad_Frente.** Devuelve la prioridad asociada al elemento del frente.
- **Poner.** Añade un elemento con una prioridad asociada.
- **Quitar.** Elimina el elemento del frente.
- **Vacia.** Indica si la cola está vacía.

Colas con prioridad

Esquema de cola

Una posible clase `ColaPri` para almacenar datos de tipo `string` con una prioridad indicada por un valor entero puede tener la siguiente sintaxis.

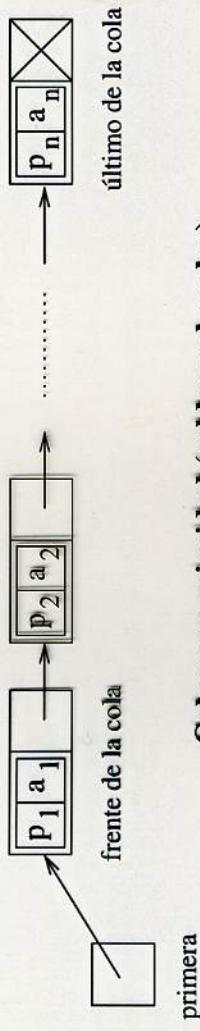
```
#ifndef __COLAPRI_H__
#define __COLAPRI_H__  
  
class ColaPri{  
    ...  
public:  
    ColaPri();  
    ColaPri(const ColaPri& p);  
    ~ColaPri();  
    ColaPri& operator=(const ColaPri& p);  
  
    bool vacia() const;  
    void poner(int pri, string c)  
    void quitar();  
    string frente() const;  
    int prioridad_frente() const;  
};  
endif
```

Uso de una cola con prioridad

```
#include <iostream>
#include <string>
#include <colapri.h>
using namespace std;  
  
int main()  
{  
    ColaPri q;  
    int nota;  
    string dni;  
  
    ...  
    // La implementación que desemos  
    cout << "Escriba una nota" << endl;  
    cin >> nota;  
    while (0<=nota && nota<=10) {  
        cout << "Escriba un dni" << endl;  
        cin >> dni;  
        q.poner(nota,dni);  
        cout << "Escriba una nota" << endl;  
        cin >> nota;  
    }  
    cout << "Los elementos en el orden de las notas son:" << endl;  
    while (!q.vacia()) {  
        cout << "Nota: " << q.prioridad_frente()  
            << " DNI: " << q.frente() << endl;  
        q.quitar();  
    }  
    return 0;  
}
```

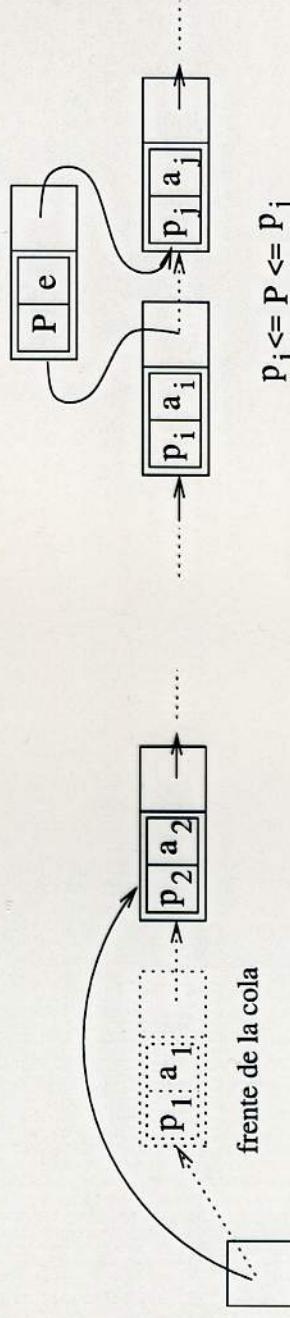
Colas con prioridad(celdas enlazadas) (1/3).

Almacenamos la secuencia de parejas en celdas enlazadas,



Colas con prioridad (celdas enlazadas)

- Una cola vacía contiene un puntero nulo.
- El **frente** de la cola se encuentra en la **primera** celda.
- Si **borrarmos** el frente, eliminamos **la primera** celda.
- Si **insertamos**, tenemos que **buscar su posición según la prioridad**



$p_i \leq P \leq P_j$

Poner un nuevo elemento
(posición intermedia)

Quitar un elemento
primera

Colas con prioridad (celdas enlazadas) (2/3).

<i>ColaPri.h</i>	<i>ColaPri.cpp</i>
<pre>#ifndef __COLAPRI_H__ #define __COLAPRI_H__ typedef int Tprio; typedef char Tbase; struct Pareja { Tprio prioridad; Tbase elemento; }; struct CeldaColaPri{ Pareja dato; CeldaCola *sig; }; class ColaPri{ CeldaColaPri *primera; public: ColaPri(); ColaPri(const ColaPri& p); ~ColaPri(); ColaPri& operator=(const ColaPri& p); bool vacia() const { return primera==0; } void poner(Tprio pri, TBase c) void quitar(); Tbase frente() const; Tprio prioridad_frente() const; }; #endif</pre>	<pre>ColaPri::ColaPri(): primera(0) {} ColaPri::ColaPri(const ColaPri& c) { if (c.primera==0) primera= 0; else { primera= new CeldaColaPri; primera->dato= c.primera->dato; CeldaColaPri *src=c.primera; CeldaColaPri *dst=primera; while (src->sig!=0) { dst->sig= new CeldaColaPri; src= src->sig; dst= dst->sig; dst->dato= src->dato } dst->sig=0; } } ColaPri::~ColaPri() { CeldaColaPri *aux; while (primera!=0) { aux= primera; primera= primera->sig; delete aux; } }</pre>

Colas con prioridad (celdas enlazadas) (3/3).

ColaPri.cpp	ColaPri.cpp
<pre>ColaPri& ColaPri::operator= (const ColaPri& c) { ColaPri caux(c); CeldaColaPri *aux; aux= this→primera; this→primera= caux→primera; caux→primera= aux; return *this; } TBase ColaPri::frente() const { assert (primera!=0); return primera→dato.elemento; } Tpri ColaPri::prioridad_frente() const { assert (primera!=0); return primera→dato.prioridad; } void ColaPri::quitar() { assert (primera!=0); CeldaColaPri *aux= primera; primera= primera→sig; delete aux; }</pre>	<pre>void ColaPri::poner(Tpri pri, TBase c) { CeldaColaPri *aux= new CeldaColaPri; aux→dato.elemento= c; aux→dato.prioridad= pri; aux→sig= 0; if (primera==0) primera=aux; else if (pri<primera→dato.prioridad) { aux→sig= primera; primera= aux; } else { CeldaColaPri *p= primera; while (p→sig!=0) { if (p→sig→dato.prioridad>pri) { aux→sig= p→sig; p→sig= aux; } return ; } else p= p→sig; } p→sig= aux; }</pre> <p style="color: blue; font-style: italic;">// se inserta al final</p>

Listas (Comparaciones).

	<i>Vector</i>	<i>Celdas</i>	<i>Celdas cab.</i>	<i>Celdas dob.</i>	<i>Celdas dob. cab.</i>
Rep. Lista	TBase *	Celda *	Celda *	CeldaDoble *	CeldaDoble *
Rep. Posicion	TBase * ó int	Celda * Lista *	Celda * Lista *	CeldaDoble * Lista * (end)	CeldaDoble *
Efic. Insertar	O(n)	O(n)	O(1)	O(1)	O(1)
Efic. Borrar	O(n)	O(n)	O(1)	O(1)	O(1)

La eficiencia de *Set*, *Get* siempre es, en peor caso, $O(1)$