Tema 8. Listas

`http://aulavirtual.uji.es`

José M. Badía, Begoña Martínez, Antonio Morales y José M. Sanchiz

{badia, bmartine, morales, sanchiz}@icc.uji.es

Estructuras de datos y de la información

Universitat Jaume I

Índice

Bibliografía

- (Nyhoff'06), capítulos 6 y 11
- (Orallo'02), capítulo 21
- *Using the STL, the C++ Standard Template Library*, R. Robson, Springer, Segunda Edición, 1999, Capítulo 7.4

Objetivos

- Conocer el concepto, funcionamiento y aplicaciones del tipo *lista*.
- Conocer el TAD *lista* y sus operaciones asociadas.
- Saber implementar el tipo *lista* mediante vectores y mediante nodos enlazados.
- Saber usar el tipo *lista* para resolver problemas.
- Conocer variantes de la lista enlazada, como listas circulares, listas con nodo de cabecera, o listas doblemente enlazadas.
- Conocer variantes del TAD *lista*, como listas ordenadas, multilistas, o listas de listas.
- Conocer y saber usar el contenedor `list` de la STL.

1 Definición y aplicaciones

Las listas son algo común en la vida diaria: listas de alumnos, listas de clientes, listas de espera, listas de distribución de correo, etc.

Lista $\Rightarrow$ estructura de datos lineal, finita y ordenada, que permite la inserción y borrado de elementos en cualquier posición

Características

- Todos los elementos de la lista son del mismo tipo.
- Existe un orden en los elementos, ya que es una estructura lineal, pero los elementos no están ordenados por su valor sino por la posición en que se han insertado.
- Para cada elemento existe un anterior y un siguiente, excepto para el *primero*, que no tiene anterior, y para el *último*, que no tiene siguiente.
- Se puede acceder y eliminar cualquier elemento.
- Se pueden insertar elementos en cualquier posición.

1 Definición y aplicaciones (II)

- Operaciones de manipulación:
  - **Insertar**: inserta un elemento antes o después de otro elemento.
  - **Eliminar**: elimina un elemento de una posición.

Ejemplos de operaciones:

(X F G A H)

(X F G H)

(L X F H)

(L E X F H)

- Al igual que la pila y la cola, la lista es una estructura de datos dinámica.
- Las pilas y las colas son tipos especiales de listas con restricciones de acceso.

2 Tipo Abstracto de Datos *lista*

TAD: lista

Usa: logico, nat, tipobase, apuntador

Operaciones:

CrearLista: → lista

InsertarUltimo: lista x tipobase → lista

InsertarAntes: tipobase x lista x apuntador → lista

Buscar: lista x tipobase → apuntador

Dato: lista x apuntador → tipobase

Modificar: lista x apuntador x tipobase → lista

Borrar: lista x apuntador → lista

ListaVacia: lista → logico

Tamaño: lista → nat

Primero: lista → apuntador

Ultimo: lista → apuntador

Siguiente: lista x apuntador → apuntador

Anterior: lista x apuntador → apuntador

2 Tipo Abstracto de Datos *lista* (II)

Axiomas: ∀ L ∈ lista, ∀ e,f ∈ tipobase, ∀ p ∈ apuntador

1) InsertarAntes(e,CrearLista,p) = error
2) InsertarAntes(f,InsertarUltimo(L,e),p) =
   si p=Ultimo(InsertarUltimo(L,e)) entonces InsertarUltimo(InsertarUltimo(L,f),e)
   sino InsertarUltimo(InsertarAntes(f,L,p),e)
3) Dato(CrearLista,p) = error
4) Dato(InsertarUltimo(L,e),p) = si p=Ultimo(InsertarUltimo(L,e)) entonces e
   sino Dato(L,p)
5) Modificar(CrearLista,p,e) = error
6) Modificar(InsertarUltimo(L,e),p,f) = si p=Ultimo(InsertarUltimo(L,e))
   entonces InsertarUltimo(L,f) sino InsertarUltimo(Modificar(L,p,f),e)
7) Borrar(CrearLista,p) = error
8) Borrar(InsertarUltimo(L,e),p) = si p=Ultimo(InsertarUltimo(L,e)) entonces L
   sino InsertarUltimo(Borrar(L,p),e)
9) ListaVacia(CrearLista) = verdadero
10) ListaVacia(InsertarUltimo(L,e)) = falso
11) Tamaño(CrearLista) = cero
12) Tamaño(InsertarUltimo(L,e)) = succ(Tamaño(L))

3 Implementación estática

Se usa un vector para almacenar los elementos de la lista.

Primera idea:

Guardar los elementos en el vector en el mismo orden y sin espacios vacíos.

Ejemplo: $lista = (e_1, e_2, e_3, e_4)$

0	1	2	3	4	5	6	7
e_1	e_2	e_3	e_4				

Problemas:

Borrar e insertar serían operaciones muy costosas, pues implicarían movimiento de datos en el vector.

3 Implementación estática (II)

Segunda idea: no guardar los elementos en orden ni consecutivos.

- Junto a cada elemento guardar un índice al siguiente elemento.
- Otro índice, **`primero`**, indica la posición del primer elemento.
- Un valor especial (p.e. -1) indica fin de lista.

primero 3

0	1	2	3	4	5	6	7
e_3	e_2		e_1	e_4			
4	0		1	-1			

Ventajas:

- Borrar e insertar NO implican movimiento de datos en el vector.
- Puede haber varias listas en el vector.

3 Implementación estática (III)

Lista de vacíos:

Las posiciones vacías podrían ser otra lista.

Se obtienen en tiempo constante de la lista de vacíos.

primero 3

0	1	2	3	4	5	6	7
e_3	e_2		e_1	e_4			
4	0	5	1	-1	6	7	-1

vacíos 2

La implementación estática con índices es una primera versión de lista enlazada.

Inconveniente:

Siempre se está reservando toda la memoria ocupada por el vector.

Solución:

Una implementación con nodos enlazados: memoria dinámica.

3 Implementación estática (IV)

```
template <class T>
class list {
public:
  typedef int apuntador;
  list(int n=256);
  ~list();
  list(const list<T> &origen); // Constructor de copia
  list<T> & operator=(const list<T> &origen); // Operador de asignación

  bool push_back(const T &e); // Inserta al final. Error si llena
  apuntador insert(apuntador p, const T &e); // Inserta antes de p
  apuntador find(const T &e) const; // Busca e, devuelve su posición ó -1
  const T & data(apuntador p) const; // Devuelve el dato en p
  T & data(apuntador p); // Permite modificar el dato en p
  apuntador erase(apuntador p); // Borra el elemento de p
```

3 Implementación estática (V)

```
  bool empty() const;
  int size() const;
  apuntador begin() const;
  apuntador end() const; // apuntador después del último
  apuntador next(apuntador p) const;
  apuntador prev(apuntador p) const;
private:
  struct nodo {
    T info; apuntador sig;
  };
  apuntador primero, vacios;
  nodo *elementos; // vector dinámico de nodos
  int capacidad;
  apuntador NuevoNodo(); // Desengancha un nodo de vacíos. Si no hay, -1
  void LiberarNodo(apuntador p);
};
```

3 Implementación estática (VI)

```
template <class T>
list<T>::list(int n) {
  capacidad = n;
  elementos = new nodo[n];
  primero = -1;
  vacios = 0;
  for (apuntador i = 0; i < n-1; i++) elementos[i].sig = i+1;
  elementos[n-1].sig = -1;
}

template <class T>
list<T>::~list() {
  delete [] elementos;
}
```

3 Implementación estática (VII)

```
template <class T>
typename list<T>::apuntador
list<T>::NuevoNodo() {
  apuntador aux = vacios;
  if ( vacios != -1 )
    vacios = elementos[aux].sig;
  return aux;
}

template <class T>
void
list<T>::LiberarNodo (apuntador p) {
  elementos[p].sig = vacios;
  vacios = p;
}
```

3 Implementación estática (VIII)

```
template <class T>
typename list<T>::apuntador
list<T>::insert(apuntador p, const T &e) {
  // Devuelve la posición del nuevo elemento
  apuntador nuevo = NuevoNodo(), ant = primero;
  if ( nuevo != -1 ) {  // si no está llena
    elementos[nuevo].info = e;
    elementos[nuevo].sig = p;
    if ( p == primero) primero = nuevo;
    else {
      while (elementos[ant].sig != p) ant = elementos[ant].sig;
      elementos[ant].sig = nuevo;
  }
  return nuevo;
}
```

3 Implementación estática (IX)

```
template <class T>
typename list<T>::apuntador
list<T>::erase(apuntador p) {
  // Devuelve la posición del siguiente elemento
  apuntador ant = primero, res = elementos[p].sig;
  if (p == primero) primero = elementos[primero].sig;
  else  {
    while (elementos[ant].sig != p )
       ant = elementos[ant].sig;
    elementos[ant].sig = elementos[p].sig;
  }
  LiberarNodo(p);
  return res;
}
```

4 Lista enlazada

Se usan nodos ubicados en memoria de forma dinámica.

Ventaja:

- Sólo se reserva en cada momento la memoria que se necesita.

Inconveniente:

- Se deja la gestión de memoria en manos del Sistema Operativo.
  Puede ser ineficiente, pero se puede solucionar gestionando la memoria directamente.
  Se darán algunas ideas.

4 Lista enlazada (II)

```
template <class T>
class list {
  class nodo;
public:
  typedef nodo * apuntador;
  list();
  void push_back(const T &e); // Resto de operaciones como la estática
private:
  class nodo {
  public:
    T info; nodo *sig;
    nodo(const T &valor = T() , nodo * siguiente = NULL):
          info(valor), sig(siguiente) { } // Constructor de nodo
  };
  nodo *primero;
};
```

4 Lista enlazada (III)

```
template <class T>
list<T>::~list() {  //destructor
   apuntador aux;
   while (!empty()) {
      aux = primero;
      primero = aux->sig;
      delete aux;
    }
}
```

4 Lista enlazada (IV)

```
template <class T>
typename list<T>::apuntador
list<T>::insert(apuntador p, const T &e) {
  // Devuelve la posición del nuevo elemento
  apuntador nuevo = new nodo(e, p), ant = primero;
  if ( p == primero) primero = nuevo;
  else {
    while (ant->sig != p)
      ant = ant->sig;
    ant->sig = nuevo;
  }
  return nuevo;
}
```

4 Lista enlazada (V)

```
template <class T>
typename list<T>::apuntador
list<T>::erase(apuntador p) {
  apuntador ant = primero, res = p->sig;
  if (p == primero) primero = primero->sig;
  else {
    while ( ant->sig != p)
      ant = ant->sig;
    ant->sig = p->sig;
  }
  delete p;
  return res;
}
```

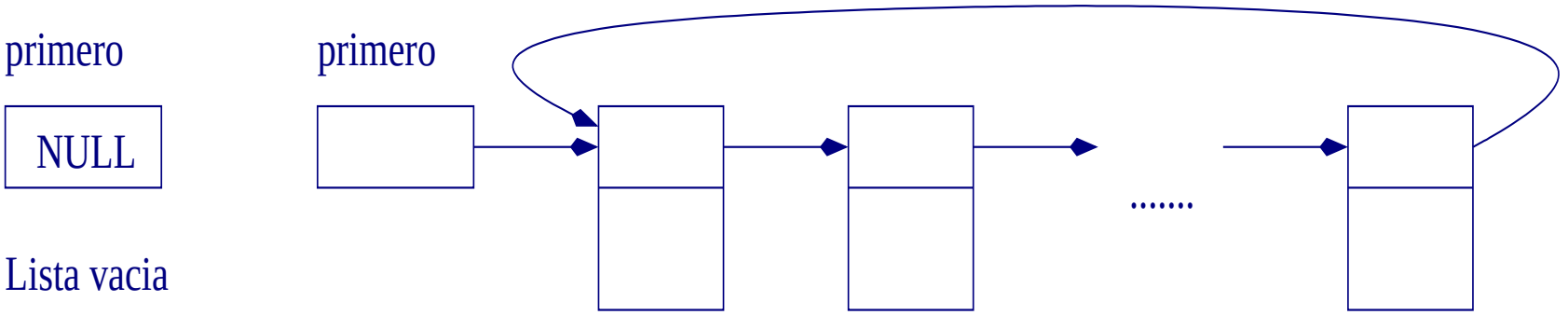
4 Lista enlazada (VI)

Ejemplo de uso:

```
list<int> l;
list<int>::apuntador p;
...
// código para recorrer una lista
for (p = l.begin(); p != l.end(); p = l.next(p))
  cout << l.data(p) << endl;
// código para vaciar una lista
while (!l.empty()) l.erase(l.begin());
// código para buscar un dato
p = l.find(x);
if (p != l.end()) //encontrado
  // borrarlo
  l.erase(p);
```

Lista Circular

El último nodo apunta al primero.



Ventajas:

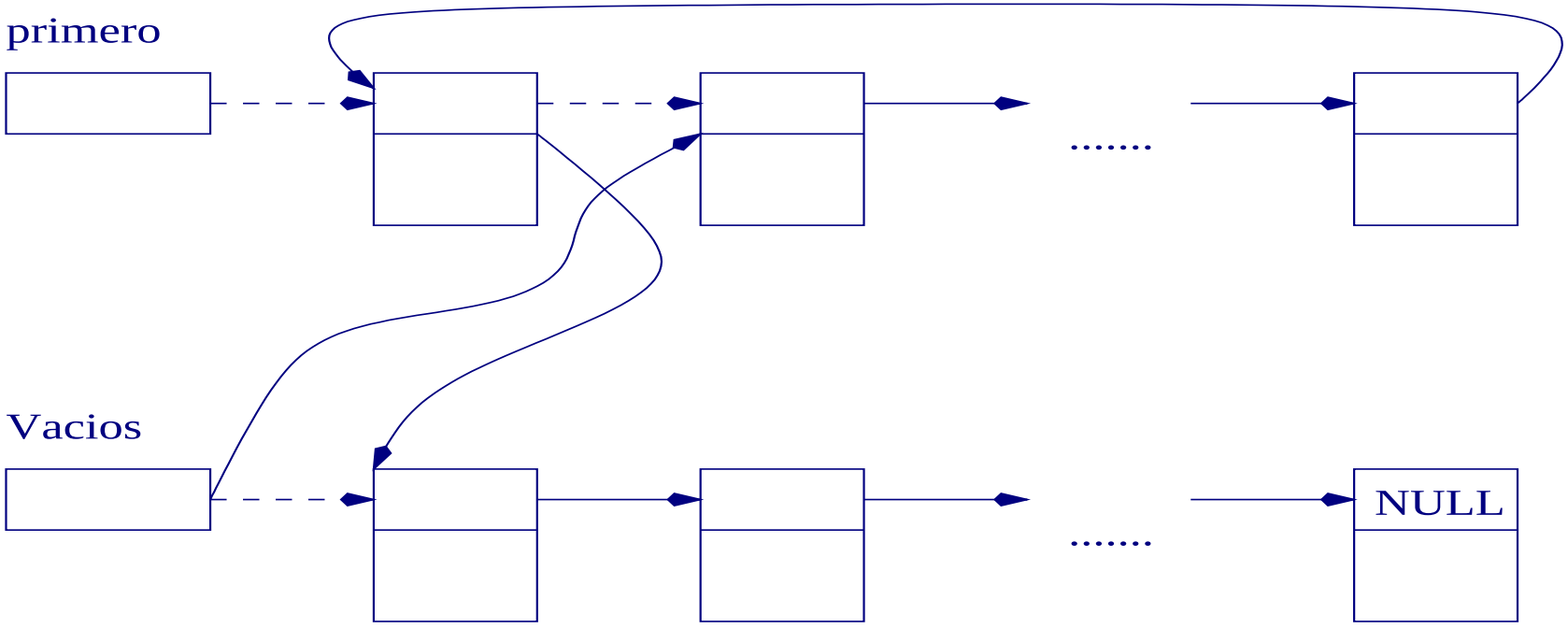
- Se puede recorrer la lista empezando en cualquier posición.
- **Principal ventaja**: Se puede borrar la lista completa en tiempo constante, independiente del número de nodos.

  Para ello es necesario mantener una lista de nodos vacíos.

5 Variantes de la lista enlazada (II)

Código para eliminar una lista circular completa:

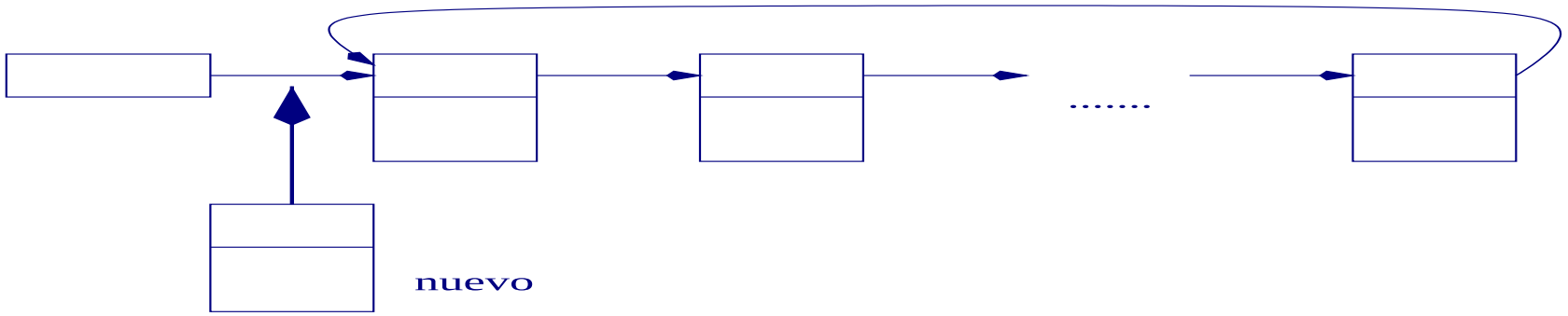
```
apuntador aux = vacios;
vacios = primero->sig;
primero->sig = aux;
primero = NULL;
```



5 Variantes de la lista enlazada (III)

Problema:

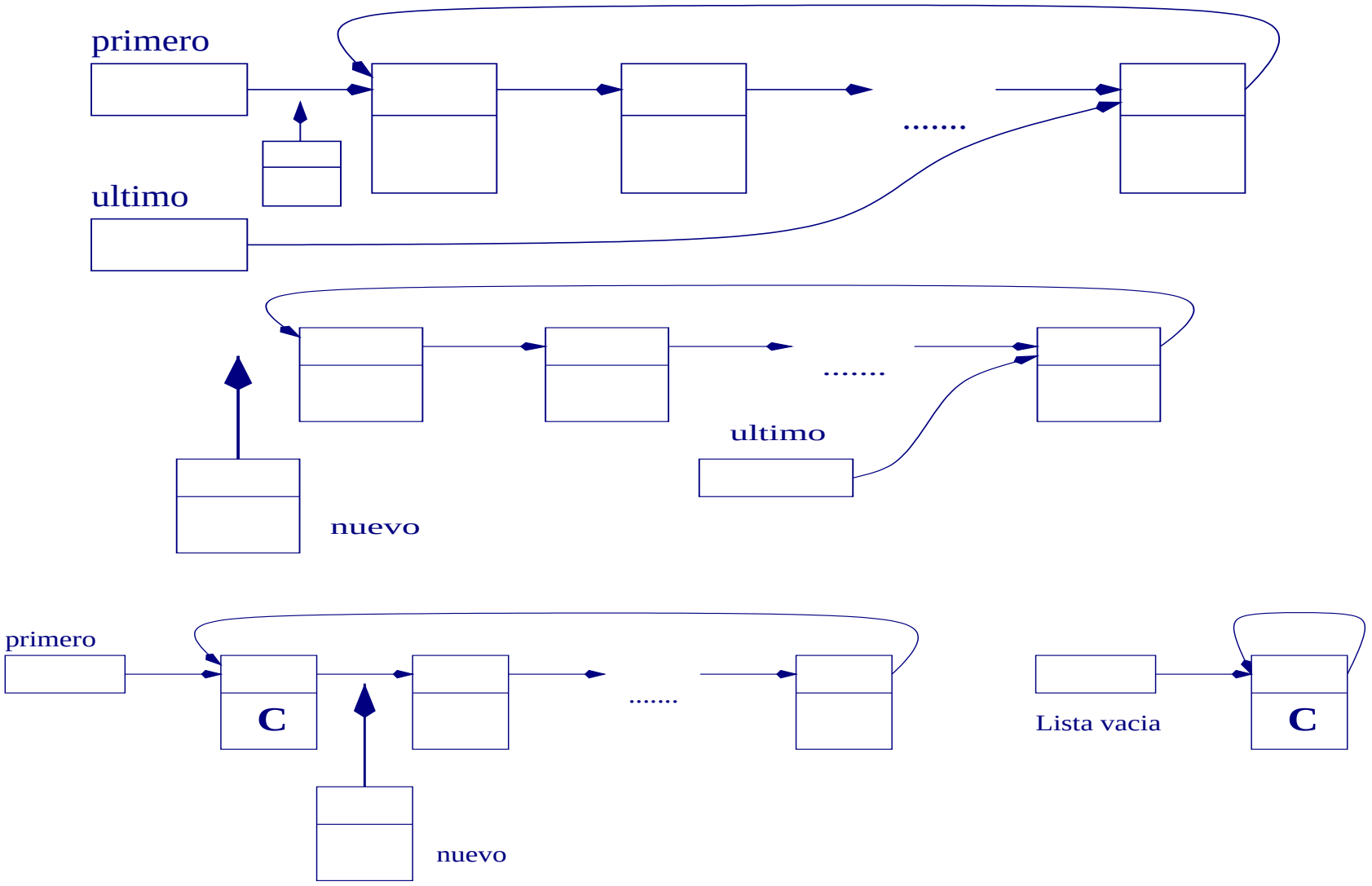
Insertar el un elemento al principio es costoso, hay que modificar el último nodo.



Soluciones:

- Mantener en la clase lista un puntero al primero y uno al último.
- Tener en la clase lista un solo puntero al último.
- Considerar que el primer nodo no contiene ningún elemento $\Rightarrow$ **nodo de cabecera**.

5 Variantes de la lista enlazada (IV)



5 Variantes de la lista enlazada (V)

Listas con nodo de Cabecera

El primer nodo no contiene ningún elemento. Puede contener información como: número de elementos, fecha de creación, etc.

Ventaja: el nodo de cabecera evita el caso especial de insertar y borrar al principio.

Puede haber nodo de cabecera en cualquier tipo de lista:

- simple o doblemente enlazadas,
- circulares o no circulares,
- multilistas.

5 Variantes de la lista enlazada (VI)

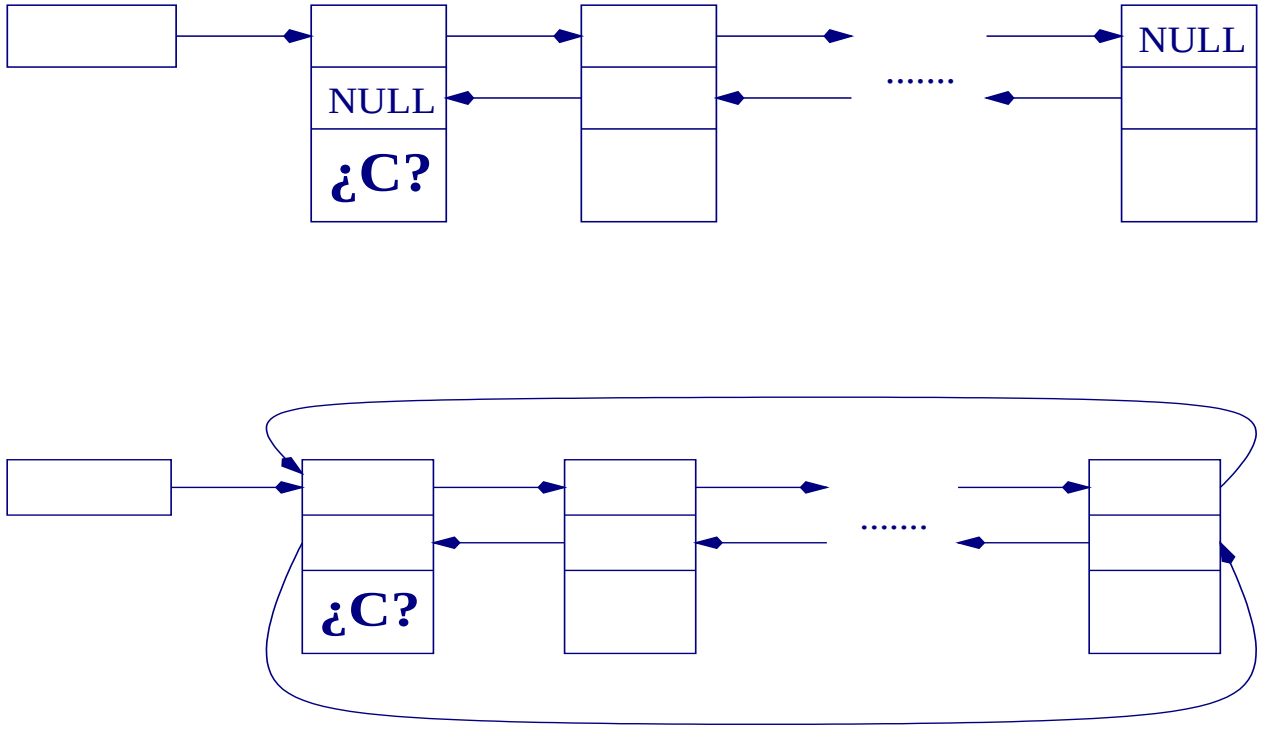
Lista doblemente enlazada

- Dada la posición de un elemento, se puede acceder en tiempo constante al anterior y al siguiente.
- Se puede borrar un elemento, insertar antes o después en tiempo constante.
- La implementación enlazada se realiza con dos punteros en la clase **nodo**:

```
class nodo {
public:
  T info;
  apuntador ant, sig;
  nodo(const T &valor=T(),apuntador anterior=NULL,apuntador siguiente=NULL):
       info(valor), ant(anterior), sig(siguiente) {}
};
```

5 Variantes de la lista enlazada (VII)

Puede ser circular o no circular, con o sin nodo de cabecera.



5 Variantes de la lista enlazada (VIII)

Operaciones:

Insertar antes	`nuevo->sig = p;`
	`nuevo->ant = p->ant;`
	`p->ant->sig = nuevo;`
	`p->ant = nuevo;`

Insertar después	`nuevo->sig = p->sig;`
	`nuevo->ant = p;`
	`p->sig->ant = nuevo;`
	`p->sig = nuevo;`

Borrar	`p->ant->sig = p->sig;`
	`p->sig->ant = p->ant;`
	`delete(p);`

Lista ordenada

- Los elementos están ordenados por su valor.
- Existe una función para comparar elementos. En C++ pueden ser los operadores $<$, $<=$, $>$, $>=$, definidos para el tipo base.
- Las operaciones de manejo de la lista permiten insertar un elemento, manteniendo el orden.
- No tiene sentido modificar un elemento, pues podría quedar desordenado. Para modificar hay que borrar e insertar el nuevo valor.

TAD Lista ordenada

TAD: listaord

Usa: logico, nat, tipobase, apuntador

Operaciones:

CrearLista: → listaord

InsertarOrd: listaord x tipobase → listaord

Siguiente: listaord x apuntador → apuntador

Primero: listaord → apuntador

Ultimo: listaord → apuntador

Buscar: listaord x tipobase → apuntador

ListaVacia: listaord → logico

Borrar: listaord x apuntador → listaord

Tamaño: listaord → nat

Dato: listaord x apuntador → tipobase

6 Variantes del TAD *lista* (III)

Axiomas: ∀ L ∈ listaord, ∀ e ∈ tipobase, ∀ p ∈ apuntador

1) ListaVacia(CrearLista) = verdadero
2) ListaVacia(InsertarOrd(L,e)) = falso
3) Borrar(CrearLista,p) = error
4) Borrar(InsertarOrd(L,e),p)=si Dato(InsertarOrd(L,e),p)=e entonces L
   sino InsertarOrd(Borrar(L,p),e)
5) Tamaño(CrearLista) = cero
6) Tamaño(InsertarOrd(L,e)) = succ(Tamaño(L))
7) Dato(CrearLista,p) = error
8) Dato(InsertarOrd(L,e),p)=si p=Buscar(InsertarOrd(L,e), e) entonces e
   sino Dato(L,p)

6 Variantes del TAD *lista* (IV)

- La lista ordenada se puede implementar como cualquier tipo de lista vista hasta ahora: simple o doblemente enlazada, circular o no circular, con o sin nodo de cabecera, con un vector o con nodos enlazados.
- Para insertar en una lista simplemente enlazada no circular sin cabecera, ordenada de menor a mayor, habría que realizar los siguientes pasos:
  - Buscar el primer elemento mayor que el nuevo a insertar.
  - No perder la posición del nodo anterior.
  - Insertar antes del primer elemento mayor.
  - Considerar casos especiales, como la inserción al principio.

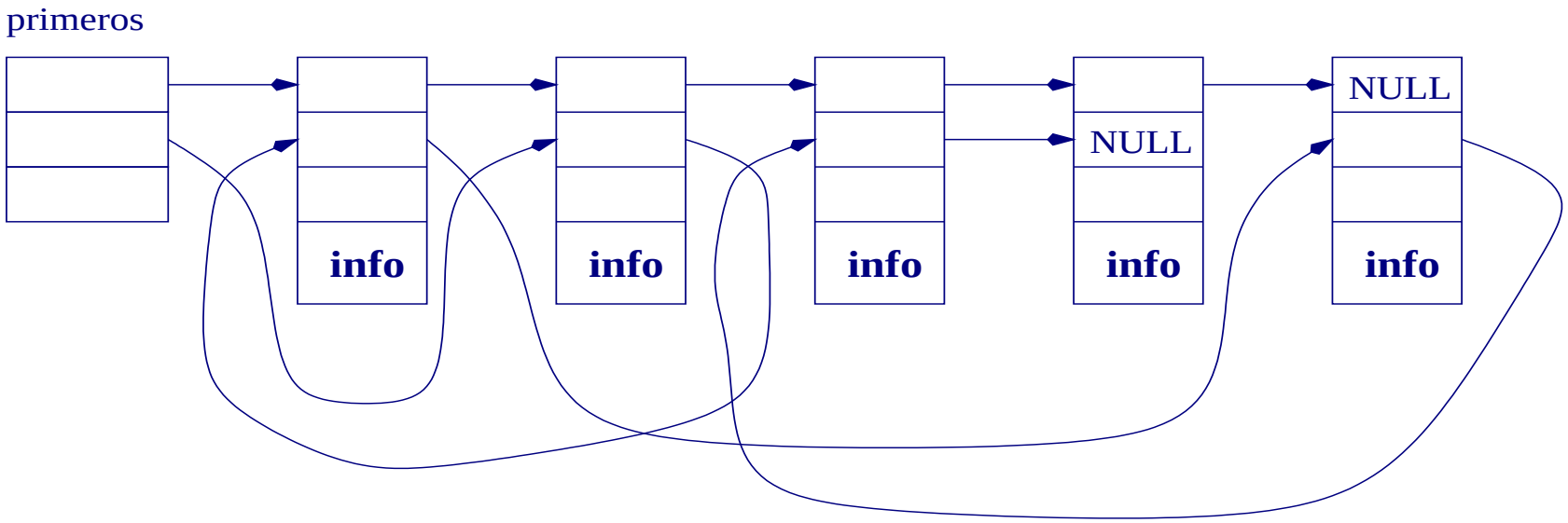
6 Variantes del TAD *lista* (V)

Multilistas

- Una multilista es una lista ordenada por varios criterios a la vez.
- Debe haber un puntero al elemento siguiente por cada uno de los criterios de ordenación. Si la lista es doblemente enlazada, también un puntero al anterior.
- Se puede considerar que tenemos tantas listas ordenadas como criterios, pero la información sólo se guarda una vez.
- Cada una de estas listas puede ser simple o doble, circular o no circular, con o sin cabecera. Normalmente todas tienen la misma organización.
- Al insertar hay que actualizar los punteros de cada criterio. Esto implica localizar la posición a insertar en cada una de las listas.
- Borrar también implica actualizar los punteros de cada criterio de ordenación.

6 Variantes del TAD *lista* (VI)

Ejemplo: 3-multilista:



Los punteros de primero y de siguiente se definirían como vectores de punteros.

6 Variantes del TAD *lista* (VII)

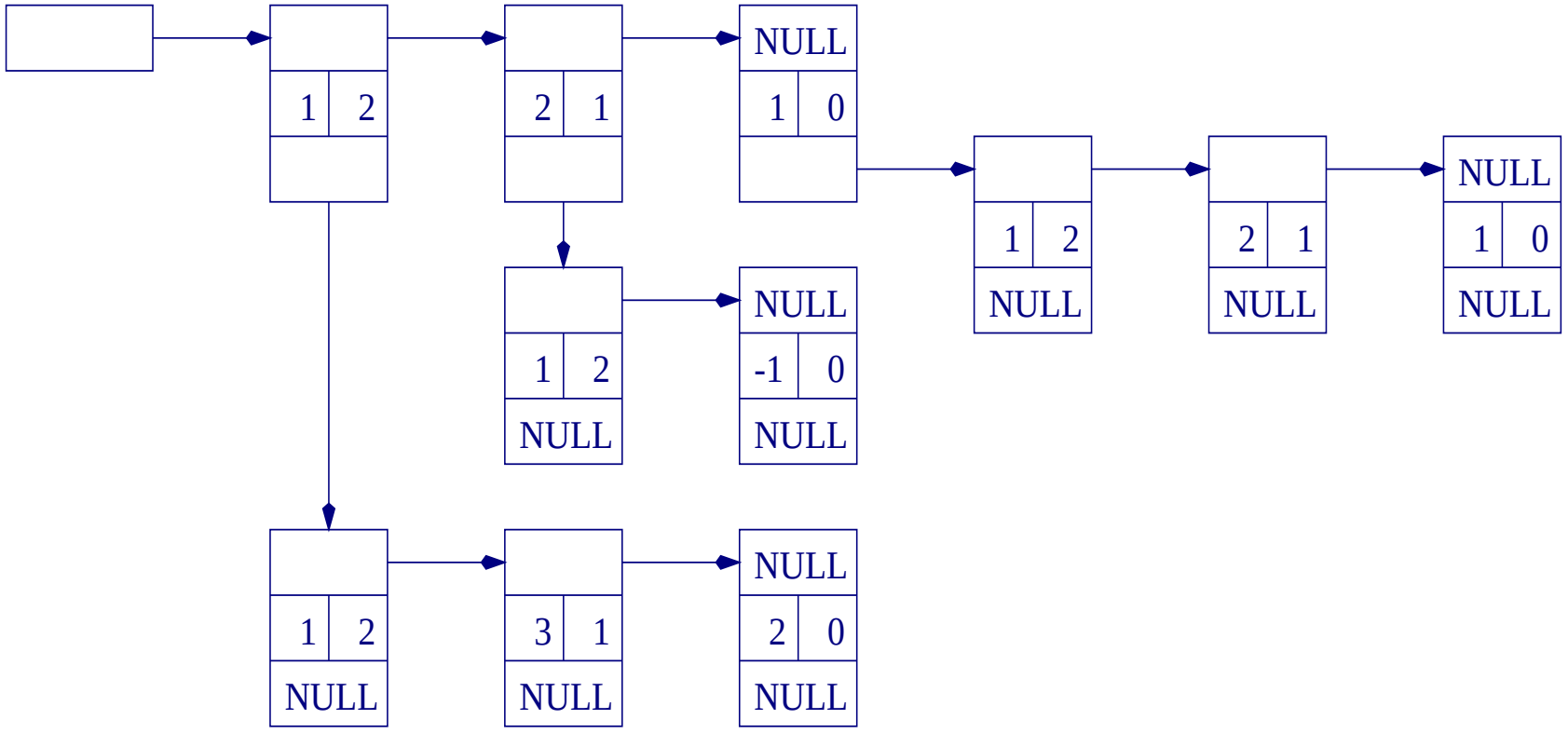
Lista de listas

- Sus elementos son a su vez listas.
- Caben todas las variantes: simple o doble, circular o no, con o sin cabecera.
- Los elementos de la lista de listas pueden ser a su vez listas de listas.
- los elementos de la lista pueden contener información, además de punteros a las listas.

6 Variantes del TAD *lista* (VIII)

Aplicación: representar polinomios de varias variables,

$$x^2(y^2+3y+2)+2x(y^2-1)+(y^2+2y+1)$$



7 El contenedor *list* de la STL

- **Uso**:

```
#include<list>
list<tipobase> l;
```

- **Iteradores**: se usan para recorrer una lista. Contienen una posición de la lista. Los iteradores de la clase `list` son de tipo bidireccional.
  - **Operaciones**:
    - ++, posición siguiente. −− posición anterior.
    - *, acceso al dato guardado en esa posición de la lista.
    - =, asignación.
    - ==, !=, comparación entre iteradores.
  - **Operaciones no permitidas**:
    - +, -, +=, -=, sumas y restas.
    - [], no se puede acceder directamente al elemento i de la lista.

7 El contenedor *list* de la STL (II)

- **Operaciones básicas**:

list<T> l;	constructor
list<T> l(n);	lista para contener n elementos, vacía
list<T> l(n,initval);	lista con n elementos inicializados a initval
~list()	destructor
l.empty()	
l.size()	
l.push_back(valor)	
l.push_front(valor)	
l.pop_back()	
l.pop_front()	
l.front()	
l.back()	

7 El contenedor *list* de la STL (III)

- **Operaciones básicas (cont.)**:

l.insert(pos,valor)	devuelve la posición al elemento insertado
l.erase(pos)	
l.erase(pos1,pos2)	borra elementos entre pos1 y pos2
l.remove(valor)	borra todos los elementos valor
l.begin()	iterador apuntando a la primera posición
l.end()	iterador apuntando una posición más allá de la última

- **La operación `find` no forma parte de las clases**:

```
iterator find(iterator first, iterator last, const T &value);
```

7 El contenedor *list* de la STL (IV)

- **Ejemplo de uso**:

Librería STL:

```
list<int> l;
list<int>::iterator i;
i = l.begin();
while (i != l.end()) {
  cout << *i << endl;
  ++i;
}
```

Nuestra implementación:

```
list<int> l;
list<int>::apuntador i;
i = l.begin();
while (i != l.end()) {
  cout << l.data(i) << endl;
  i = l.next(i);
}
```

7 El contenedor *list* de la STL (V)

- **Detalles de implementación**:
  - Es una lista de nodos enlazados, doblemente enlazada, circular, con nodo de cabecera.
  - Los campos privados de la clase incluyen un puntero al primer nodo y un contador del número de elementos.
  - La clase incluye su propio manejo de memoria dinámica.
    - Para evitar la ineficiencia de llamar al Sistema Operativo (`new` y `delete`), se lleva una lista de nodos vacíos.
    - Cuando hace falta memoria, se ubica de golpe un gran bloque y se enlaza en la lista de vacíos.
    - Sólo hay una lista de vacíos para todas las variables `list` declaradas.