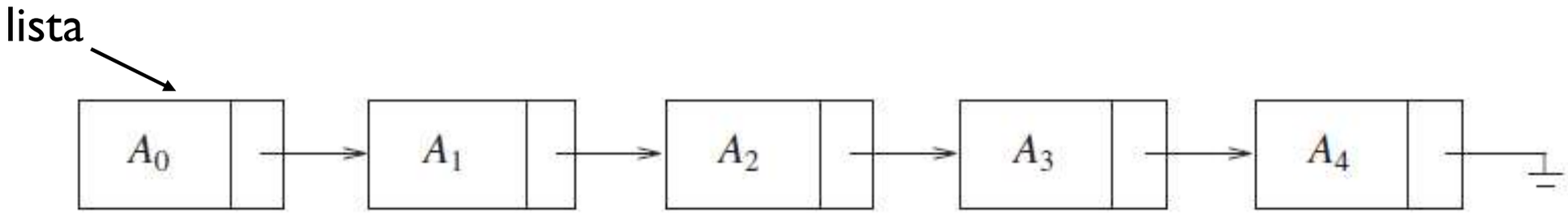
UCAB UNIVERSIDAD CATÓLICA ANDRÉS BELLO

Algoritmos y Estructuras de Datos

Oscar Meza

omezahou@ucab.edu.ve

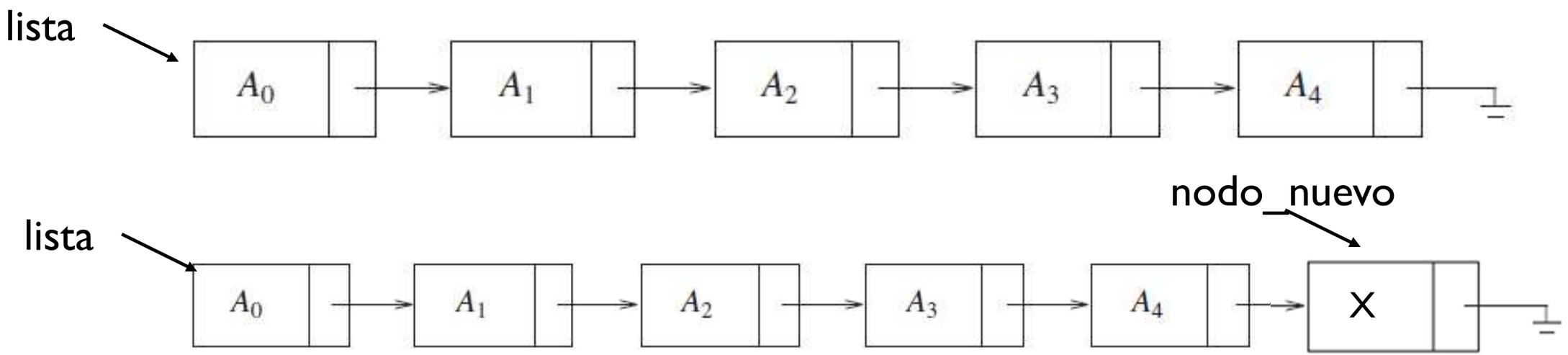
Otra implementación del TDA Lista es utilizando una estructura de **lista enlazada con apuntador a primero (Lista_enlazada) como ya la vimos:**



Cada elemento de la lista es un **nodo** que contiene el objeto (ej: int x) y un apuntador al siguiente nodo de la lista que llamamos "**próximo**". En memoria los nodos no están contiguos. El último nodo tiene **próximo** igual a **nullptr**.

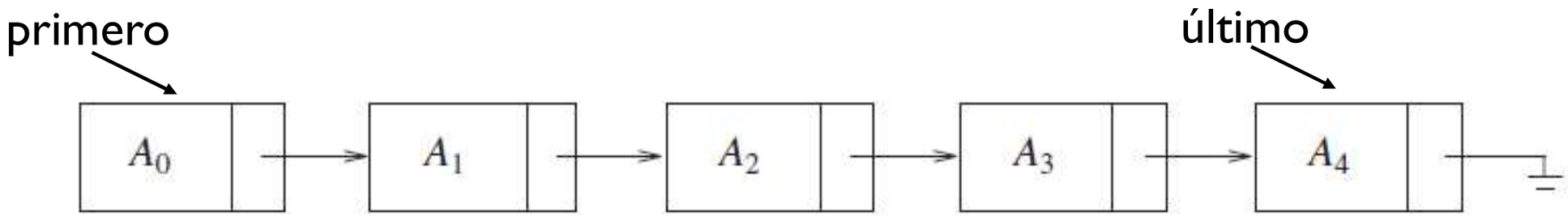
Ejemplo: Insertar X en la lista de último **(insertar_cola ()), el número de operaciones elementales sería proporcional a N, donde N es el tamaño de la lista**

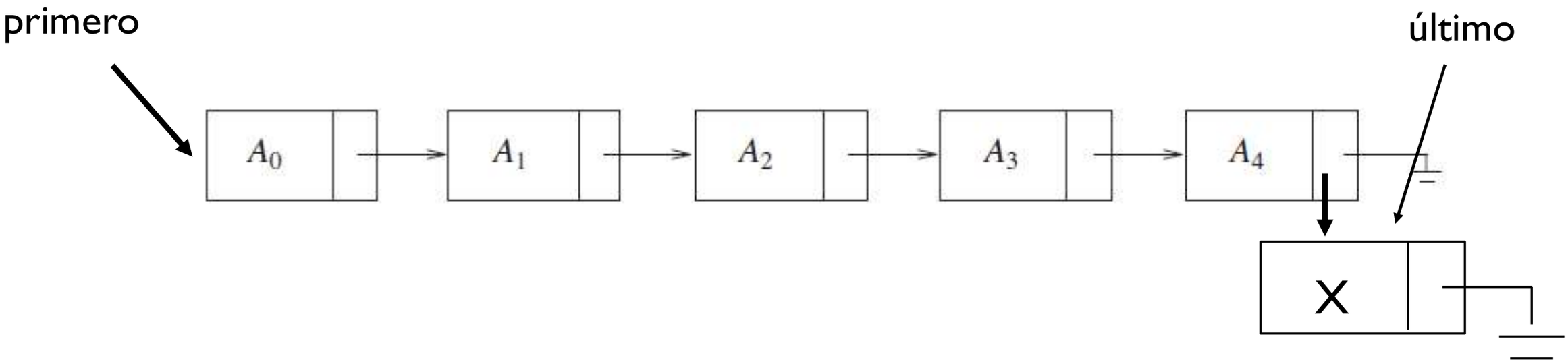
Se crea un nodo con el elemento X y se recorre la lista hasta el final (si lista es vacía se trata distinta a si no es vacía)



Note que la operación de **insertar de primero es mejor con lista que con vector**, pues solo hay que hacer un número constante de operaciones, mientras que en un vector había mover todos sus elementos.

Otra implementación de Lista es utilizando una estructura de **lista enlazada con apuntador a primero y último para facilitar las operaciones (por ejemplo insertar de último)**





insertar un elemento en la lista, sea de primero o de último, sería un número constante de operaciones elementales (no depende del tamaño de la lista) .

Sin embargo, eliminar el último sigue siendo costoso.

Ya vimos una implementación de una lista enlazada simple que llamamos Lista_enlazada. Así que una lista implementada como lista enlazada sería:

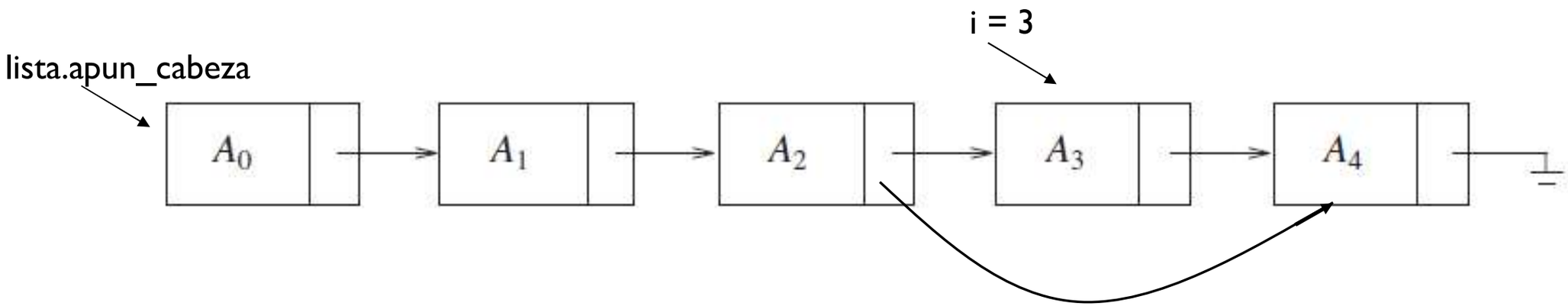
typedef Lista_enlazada Lista;

Hagamos juntos el método **eliminar el elemento en la posición i** de la lista (usando la implementación Lista_enlazada):

El prototipo de la función sería:

```
bool eliminar(Lista lista, int  i)      // si i no está entre 0 y
                                        // num_elem – 1  devuelve false
```

¿Cuál sería el cuerpo del método? Para eliminar un elemento en la posición **i** necesitamos la referencia al elemento anterior a él y el primer elemento (**i=0**) necesita un tratamiento diferente:

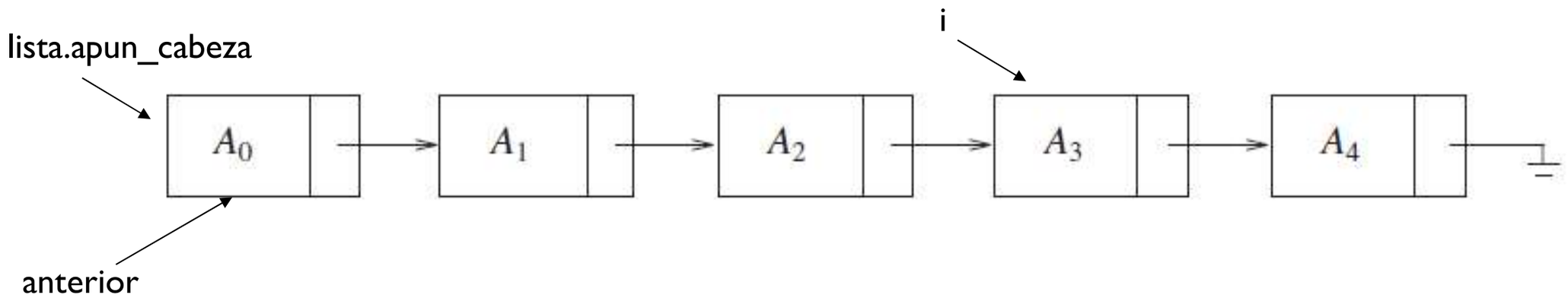

i = 3
lista.apun_cabeza
A_0
A_1
A_2
A_3
A_4

Si **i = 0**

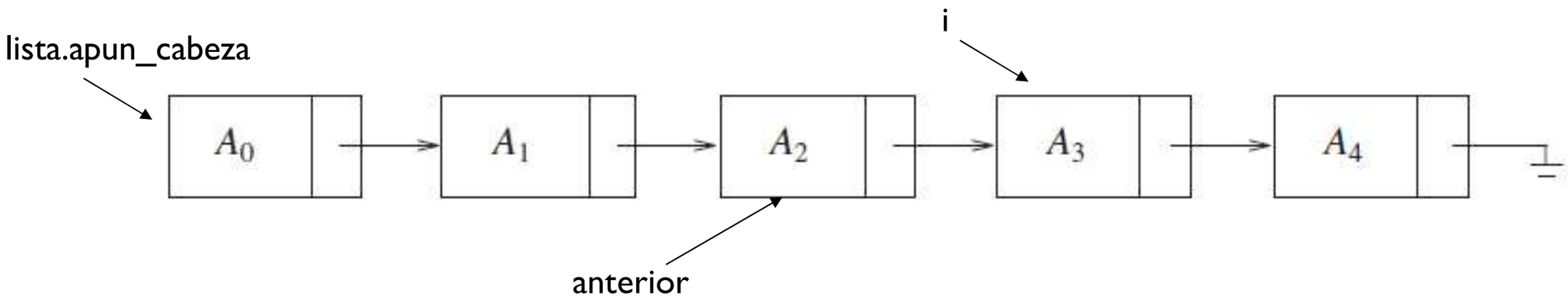
Solo necesitamos asignar a **lista.apun_cabeza** el próximo de **lista.apun_cabeza:**

```
Nodo* temp = lista.apunt_cabeza;
lista.apun_cabeza = lista.apunt_cabeza - > proximo;
lista.num_elem--;
del temp;
```

Para i > 0 el tratamiento es diferente, vamos recorriendo la lista, manteniendo un apuntador llamado **anterior, hasta que el próximo de anterior sea el elemento en la posición i**



Proceso iterativo hasta que **anterior** llegue a posición i-1:



Luego al próximo del nodo **anterior** le asignamos el próximo del nodo **en posición i y liberamos el nodo (delete)**

Si **i > 0** recorremos la lista hasta llegar al elemento en la posición anterior a **i**, a la vez vamos guardando la referencia al anterior a medida que recorremos la lista:

```
// anterior apunta inicialmente al nodo en posición 0
Nodo*  anterior = lista.apun_cabeza;
int j = 0;
```

Qué seguiría?.... ¿alguien quiere hacerlo?

UCAB UNIVERSIDAD CATÓLICA ANDRÉS BELLO

Si **i > 0** recorremos la lista hasta llegar al elemento en la posición i, a la vez vamos guardando la referencia al anterior a medida que recorremos la lista:

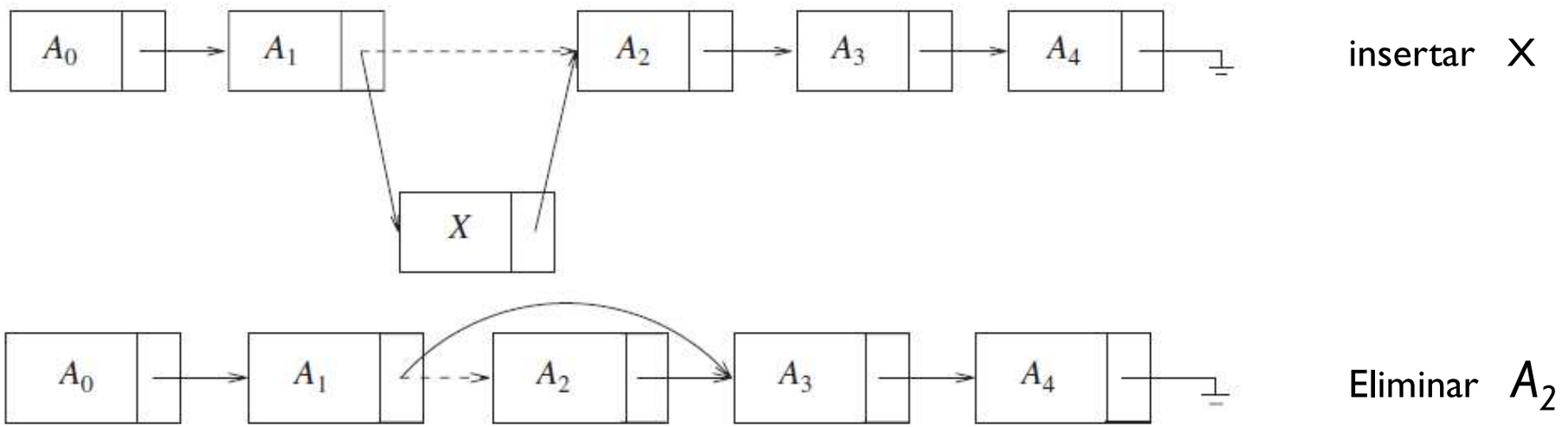
```
Nodo*  anterior = lista.apun_cabeza;
int j = 0;  // indica la posición del nodo
      // anterior al que hay que eliminar
while (  j <  i -1) { // si todavía no estamos en la pos anterior a i
         j++;
        anterior = anterior->proximo;   // anterior apunta a elem  en posición j
}
Nodo* temp = anterior->proximo;
anterior->proximo = temp->proximo;
delete temp;       lista.num_elem--;
return true;
```

Veamos la función completa……

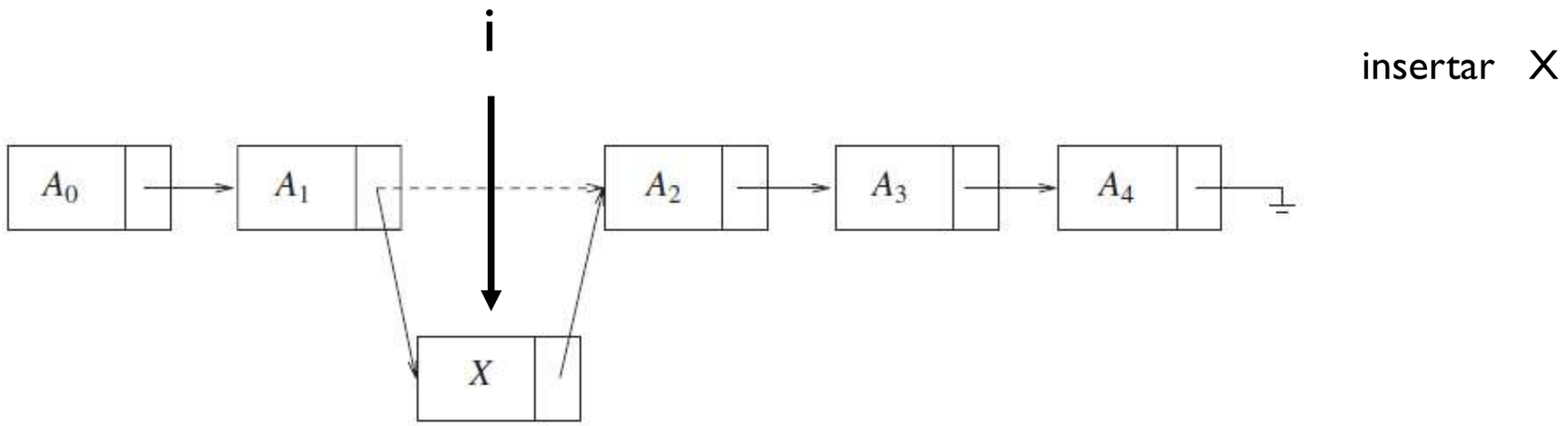
```
bool eliminar(Lista_enlazada& lista, int  i) {
// Elimina el elemento en la posición i de la lista
// Devuelve falso si y solo si  i no está entre 0 y
// num_elem-1,

  if ((i<0) || (i>=lista.num_elem)) return false;
  else
      if (i==0) {
         Nodo* temp = lista.apun_cabeza;
         lista.apun_cabeza = lista.apun_cabeza ->
                                          proximo;
         delete temp;
         lista.num_elem--;
         return true;
      }
else {
     Nodo*  anterior = lista.apun_cabeza;
     int j = 0;  // indica la posición del nodo
           // anterior al que hay que eliminar
     while (  j  < ( i -1)  ) {
        anterior = anterior->proximo;
       j++;
     }
     Nodo* temp = anterior->proximo;
     anterior->proximo = temp->proximo;
     delete temp;
     lista.num_elem--;
     return true;
   }
}
```

Si insertamos o eliminamos un elemento en la posición i, tenemos que esta operación en el peor caso haría un número de **operaciones elementales proporcional al número de elementos de la lista**, por ejemplo **i** puede ser la última posición de la lista. Porque primero hay que buscar el elemento en la posición **i**. Aunque insertar el elemento no involucra desplazar otros elementos como en arreglos.

Ejemplo: insertar X en la posición 2 y eliminar el de posición 2



Ejercicio1: Hacer una función que inserte un elemento en la posición i (i puede ser igual al número de elementos de la lista, con lo cual estaríamos insertándolo de último)



Ejercicio 2: implementar el tipo abstracto lista con el tipo concreto Lista_simple (todas las operaciones).

```
struct Nodo {
   int data;    // informacion del nodo
   Nodo *proximo;  // enlace a otro  nodo
};

typedef *Nodo Lista_simple;
```

La siguiente función tiene un error de compilación no evidente:

```
void imprimir(const Nodo * & lista){
  for (Nodo *cursor = lista;
          cursor != nullptr;
       cursor = cursor->proximo)
       std::cout << cursor->data << '
';
  std::cout << '\n';
}
```

lista es un apuntador a una constante Tipo Nodo

Y en el for: **Nodo *cursor = lista**

NO se puede asignar un apuntador a una constante a un apuntador (cursor) que NO apunte a una constante, en principio cursor podría modificar a lo que apunta

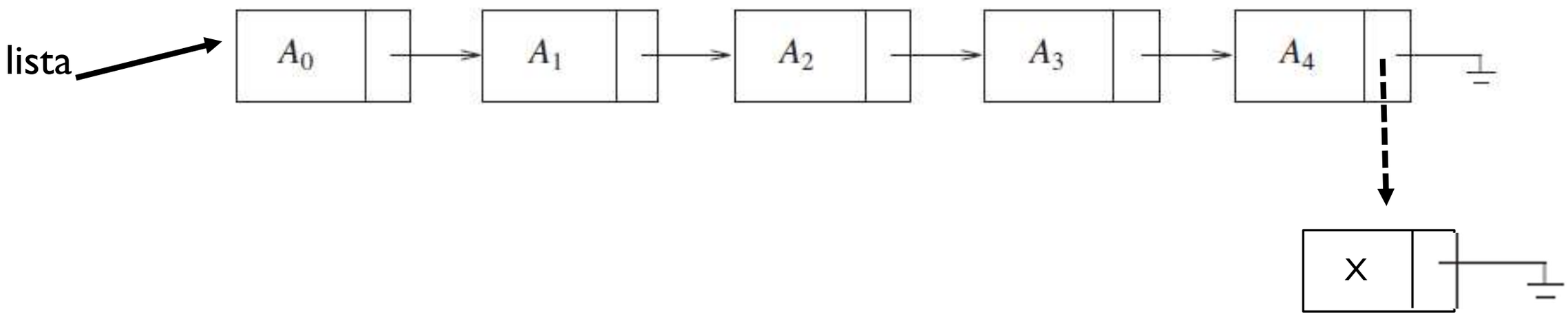
DA ERROR DE COMPILACIÓN !!

Sería correcto:

```
void imprimir(Nodo* const & lista){
  for (Nodo *cursor = lista;
          cursor != nullptr;
       cursor = cursor->proximo)
       std::cout << cursor->data << '
';
  std::cout << '\n';
}
```

Ahora **lista** es la constante, pero el valor de una constante se puede asignar a una variable que no es constante como **cursor**

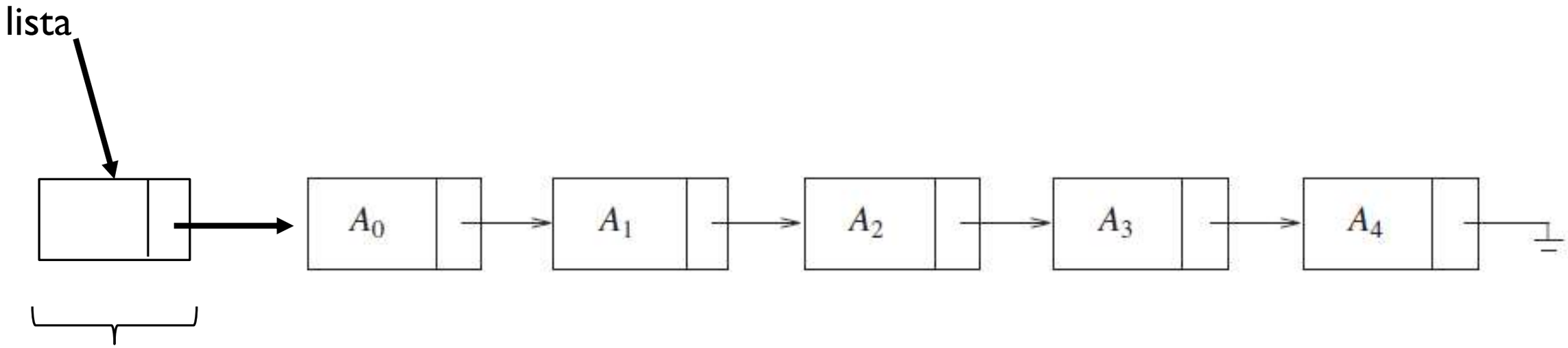
Recordemos que para insertar un elemento de último en una lista simple con apuntador solo al primer nodo, había que tratar el caso lista vacía distinto al caso lista no vacía:



El algoritmo era:

```
void insertar_cola(Nodo* &lista, int n){
    if (lista == nullptr)
        lista = new Nodo{n,nullptr} ;
    else {    // lo inserta de ultimo
        Nodo *cursor = lista;
        while (cursor->proximo != nullptr)
                cursor = cursor->proximo;
        cursor->proximo = new Nodo{n,nullptr} ;
    }
}
```

Podemos hacer esta operación más sencilla (sin distinguir lista vacía) si creamos un nodo "Centinela" al comienzo de la lista:
Un nodo Centinela es un objeto "ficticio" que nos ayuda a simplificar el tratamiento de las condiciones especiales "boundary conditions" (ejemplo: lista vacía)



Con nodo Centinela el algoritmo de insertar de último quedaría:

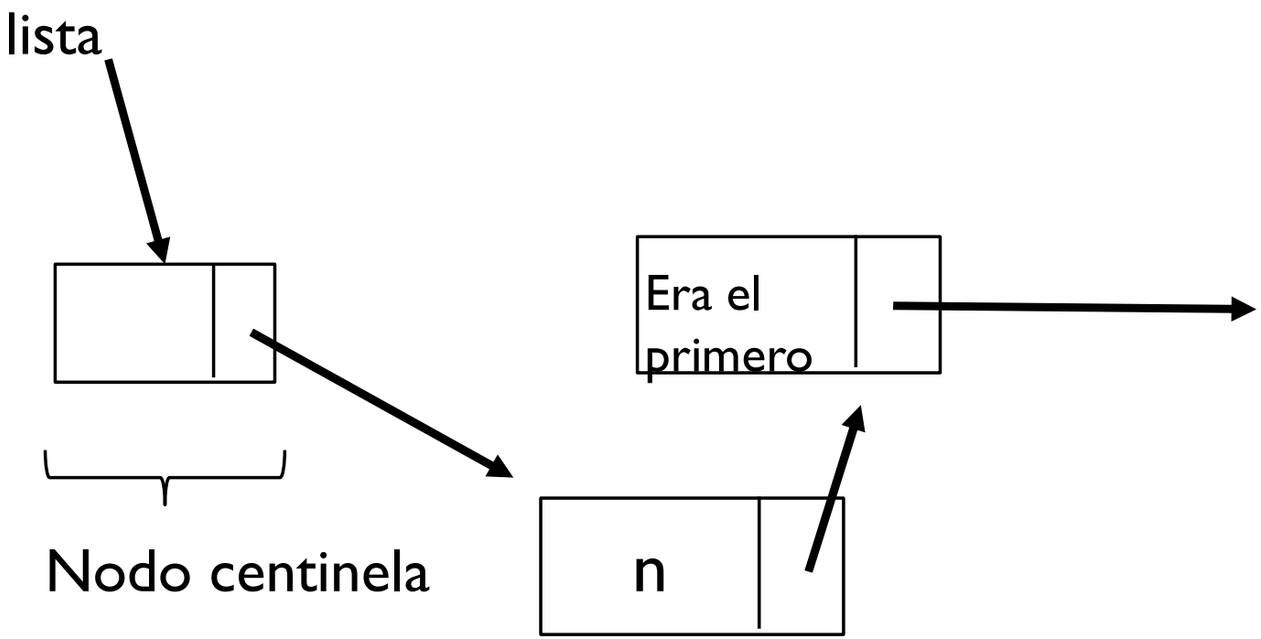
```
void insertar_cola(Nodo* &lista, int n){

        Nodo *cursor = lista;
        while (cursor->proximo != nullptr)
              cursor = cursor->proximo;
        cursor->proximo = new Nodo{n,nullptr} ;
}
```

No se trató aparte el caso lista vacía!!

Ejecutemos el algoritmo anterior paso a paso y con dibujos, insertando los elementos 1, 2, 3 a partir de una lista vacía con centinela

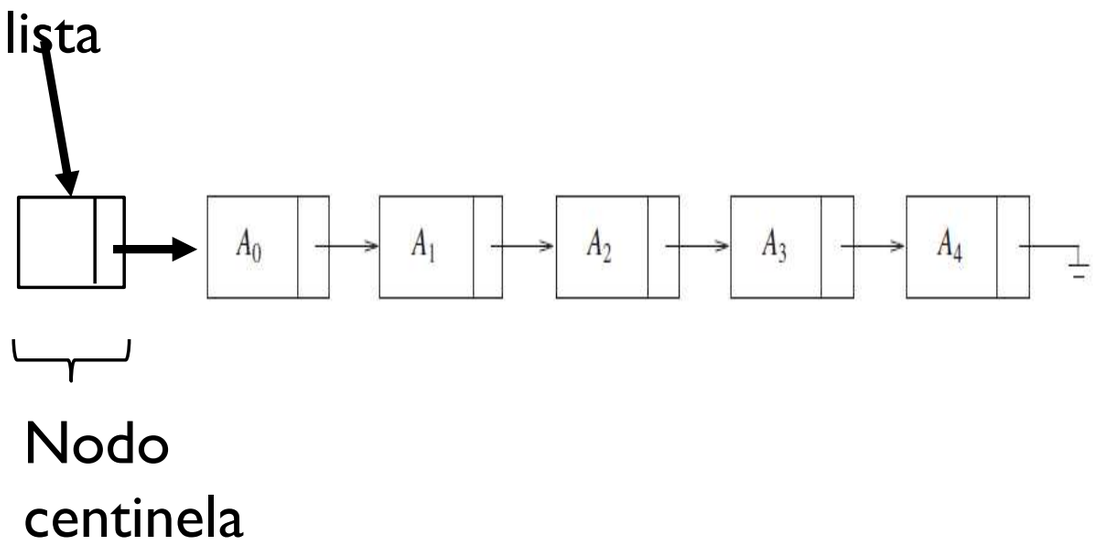
```
void insertar_cola(Nodo* &lista, int n){

        Nodo *cursor = lista;
        while (cursor->proximo != nullptr)
                cursor = cursor->proximo;
        cursor->proximo = new Nodo{n,nullptr} ;
}
```

Insertar de primero



Incluso el algoritmo de eliminar un elemento en la posición i se simplifica:

```
bool eliminar(Nodo*  &lista, int  i) {
        if ((i<0)&&(i>=num_elem(lista)) return false;
        Nodo*  anterior = lista; // comenzamos con el centinela
        int j = -1;  // indica la posición del nodo anterior al que hay que eliminar
              // el centinela tiene "posición" igual a -1
        while (  j  < ( i -1)  ) {
           anterior = anterior->proximo;
          j++;
        }
        Nodo* temp = anterior->proximo;
        anterior->proximo = temp->proximo;
        delete temp;
        return true;
}
```



Ejecutemos el algoritmo anterior paso a paso y con dibujos con una lista con centinela que contiene los elementos 1, 2, 3 y se quiere eliminar el que está en la posición 1 (es decir el elemento 2)

```
bool eliminar(Nodo*  &lista, int  i) {
   Nodo*  anterior = lista.proximo;
   int j = 0
   while (  j  < ( i -1)  ) {
     anterior = anterior->proximo;
     j++;
   }
   Nodo* temp = anterior->proximo;
   anterior->proximo = temp->proximo;
   delete temp;
}
```

El tipo **Lista Ordenada**, es una secuencia de elementos en orden creciente, si i < j el elemento en la posición i es menor o igual al elemento en la posición j.

Podemos implementar el tipo Lista Ordenada, sea con vectores o con listas enlazadas.

Lo importante es que en cualquier operación que hagamos sobre la lista, esta debe permanecer ordenada.

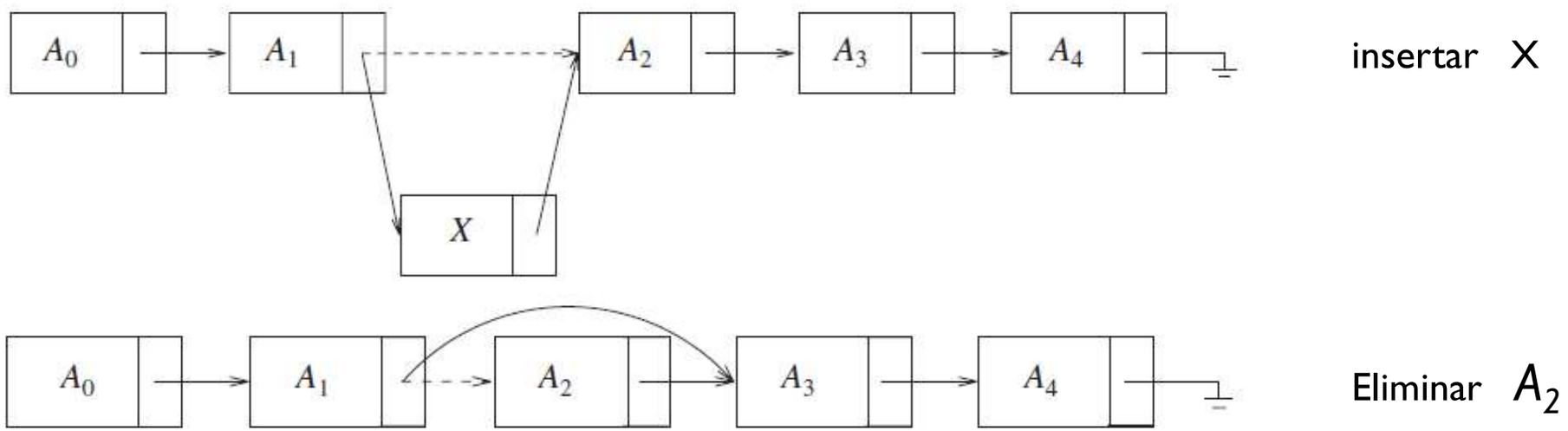
La operación "**insertar un elemento" adquiere un significado particular** pues lo debemos insertar en una posición donde la lista permanezca ordenada.

¿Cómo sería la función de insertar x en una lista ordenada simple con centinela?
Discutámoslo con ejemplos....
A partir de una lista vacía vamos insertar los elementos 3, 4, 6, 2

Veamos la operación de **insertar un elemento x en una lista ordenada** implementada con una lista enlazada simple con centinela:

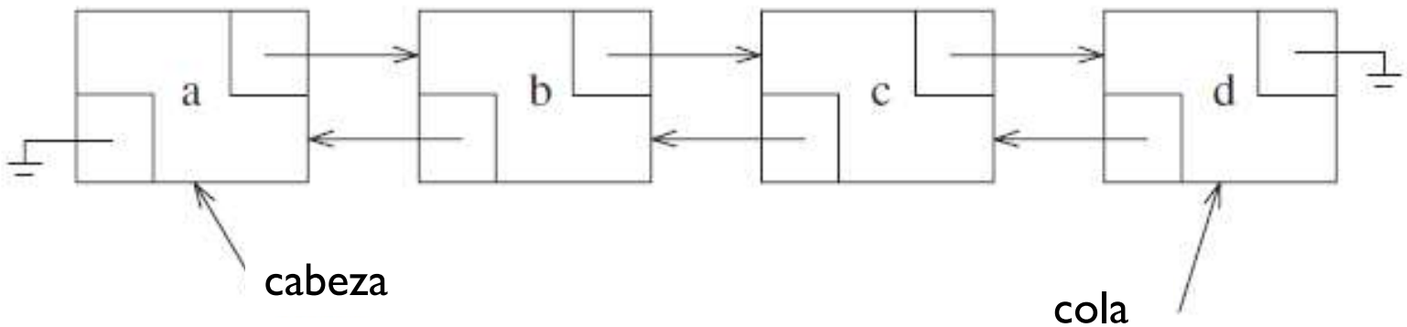
```
void insertar(Nodo* &lista, int x){
  Nodo *cursor = lista;
  while ((cursor->proximo != nullptr)&&
         ((cursor->proximo)->data <= x))
         cursor = cursor->proximo;
  cursor->proximo = new Nodo{x, cursor->proximo} ;
}
```

Si insertamos o eliminamos un elemento en la posición i, tenemos que esta operación en el peor caso haría un número de operaciones elementales proporcional al largo de la lista, por ejemplo si **i** es la última posición de la lista.
Porque primero hay que buscar el elemento en la posición **i**.
Aunque insertar el elemento no involucra desplazar otros elementos como en arreglos.
Ejemplo: insertar X en la posición 2 y eliminar el de posición 2

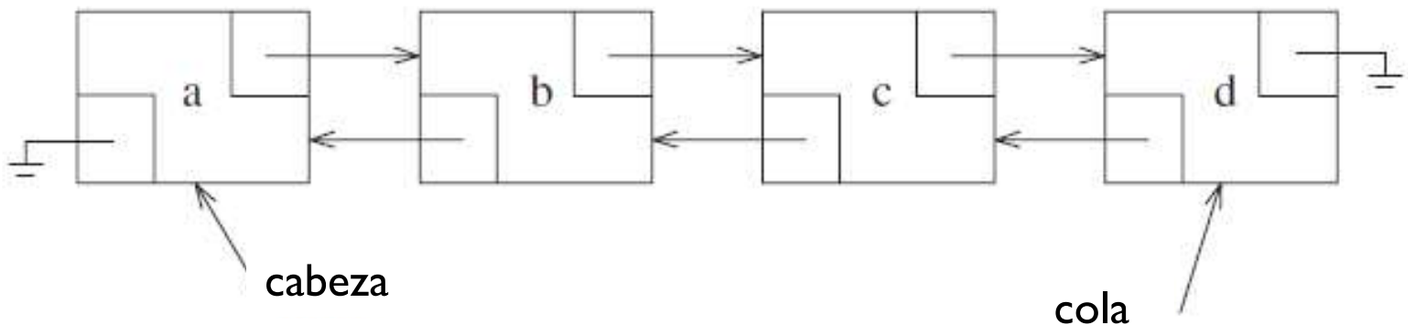


Podemos hacer más sencilla aún la operación **de inserción y eliminación de elementos** en una posición dada. No necesitamos guardar el anterior. Basta con el apuntador del elemento en posición dada

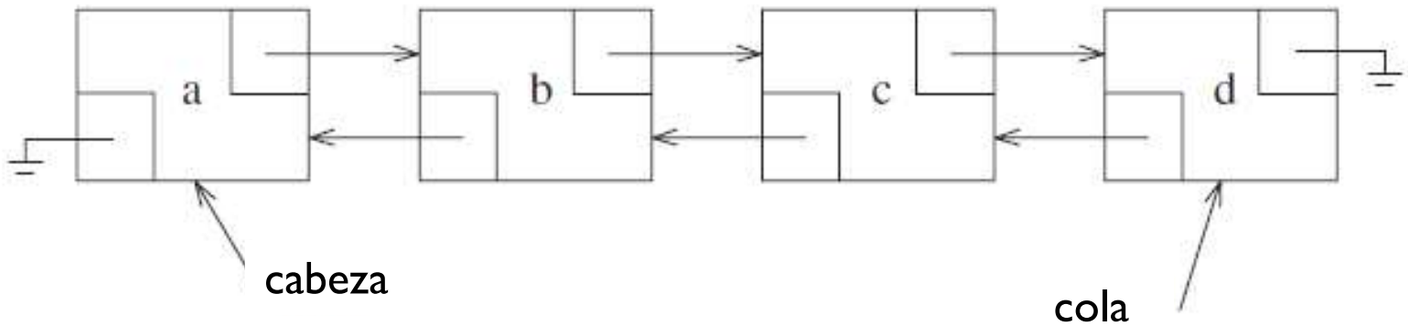
Esto se puede resolver si cada nodo posee un **apuntador al nodo siguiente** y **apuntador al anterior**. Y obtenemos **una lista doblemente enlazada**. Y con un apuntador al primero (cabeza) y al último de la lista (cola):

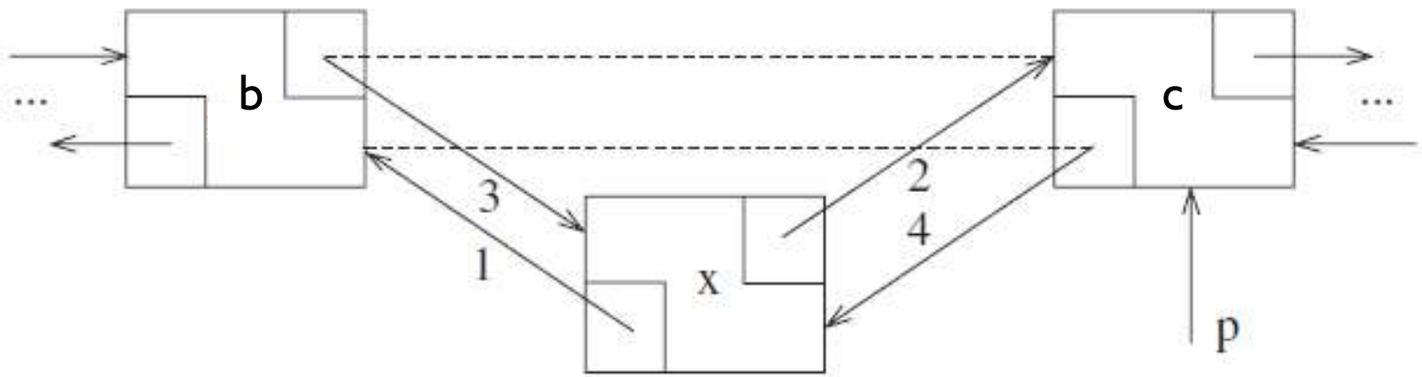


Sin embargo hay que distinguir cuando insertamos (o eliminamos) de primero o de último porque hay que ajustar el apuntador cabeza y/o cola

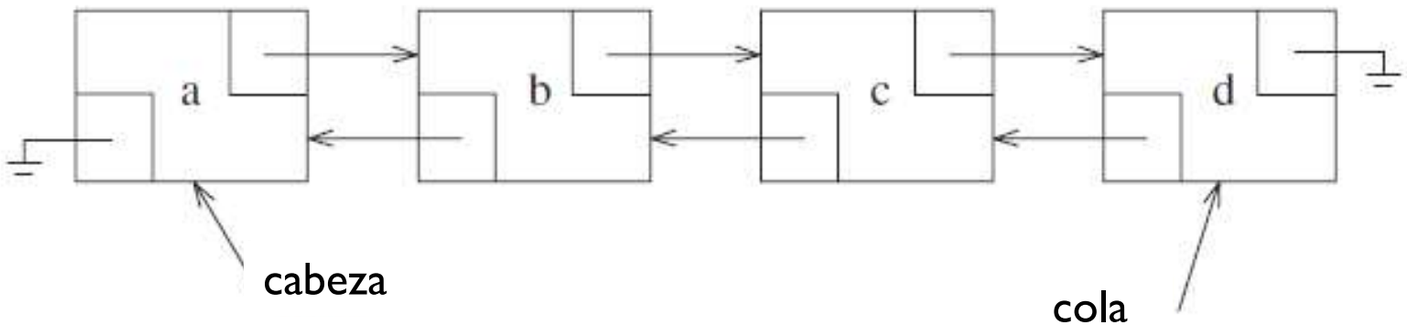


Veamos un ejemplo si insertamos en la posición 2:

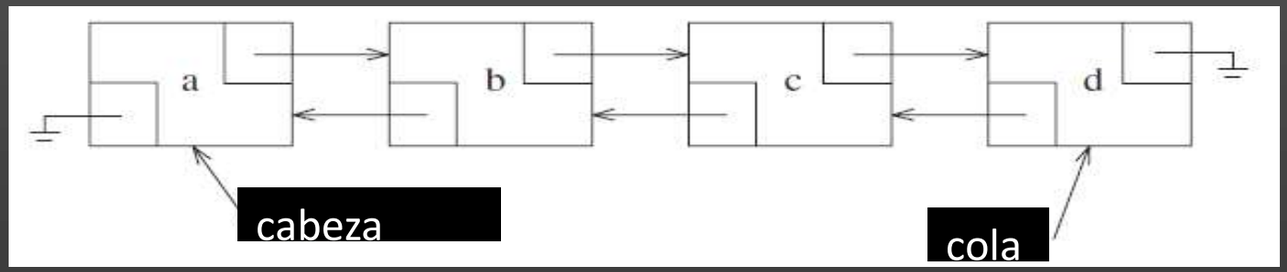




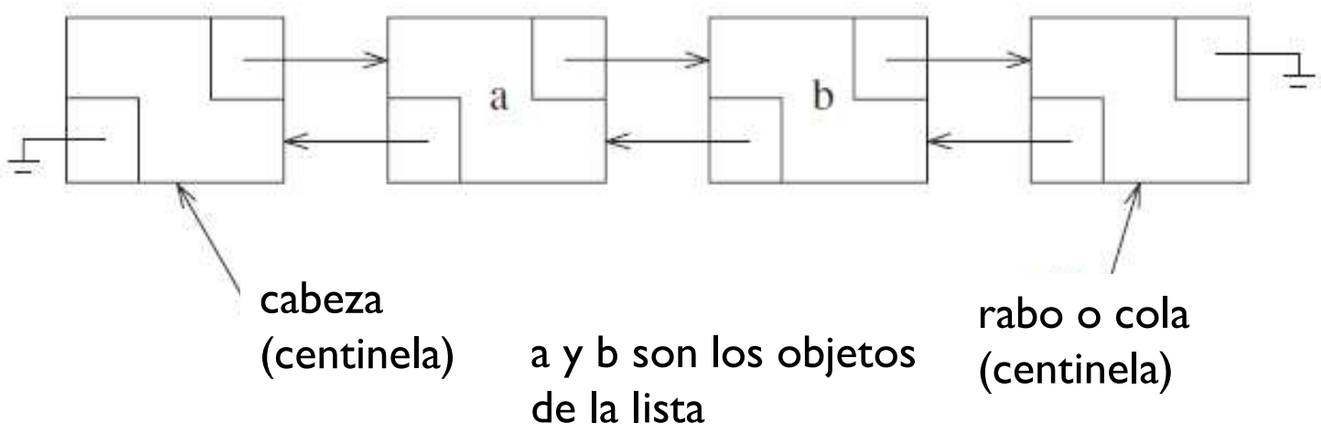
¿Cuáles son los pasos si insertamos en la posición 0?



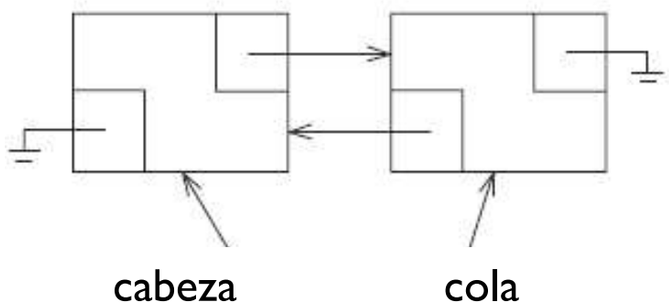
Hagámoslo juntos...


a
b
c
d
cabeza
cola

También **podemos eliminar los casos especiales (boundary conditions) para insertar o eliminar en una posición dada, s**i utilizamos unos nodos **centinela** al comienzo y final de la lista.
Facilitamos la implementación de estas operaciones (**estos nodos no contienen elementos**):



Lista vacía doblemente enlazada con centinelas:



Una estructura de datos para definir una lista doblemente enlazada con centinelas podría ser:

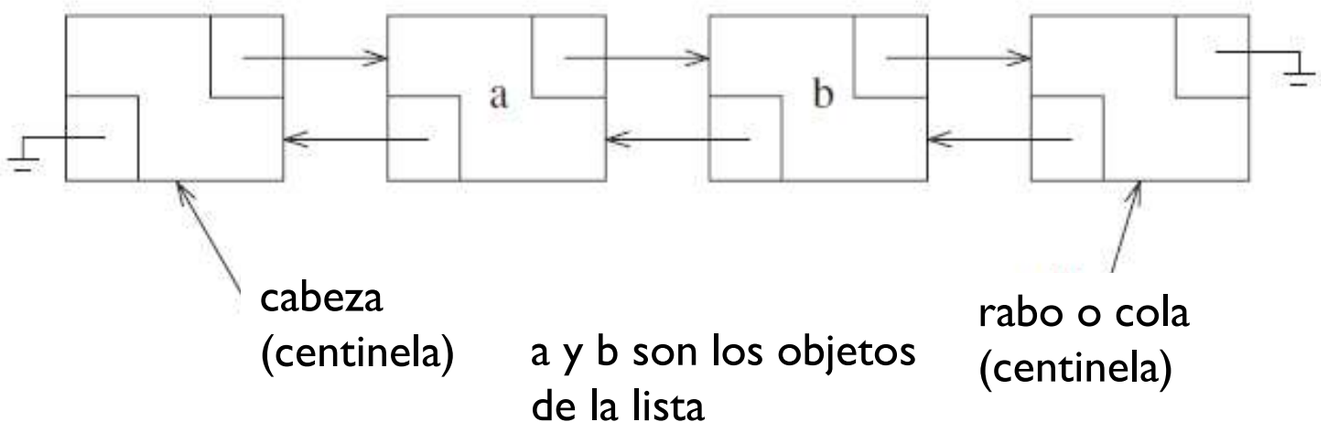
```
struct Nodo {
    int data;
    Nodo *anterior;
    Nodo *proximo;
};
```

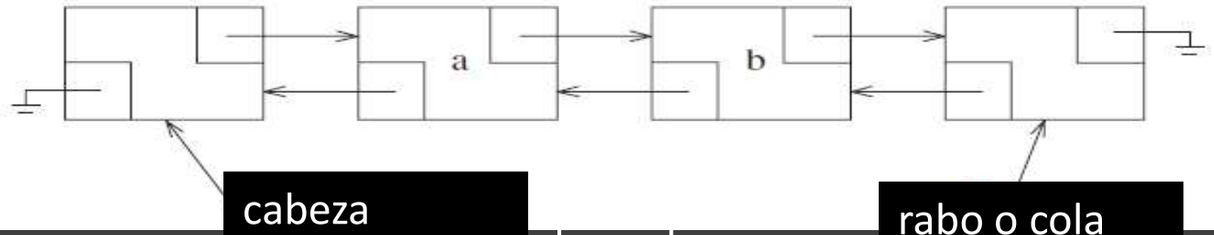
```
struct Lista_dob_enl  {
  Nodo* cabeza = nullptr;
  Nodo* cola = nullptr;
  int num_elem=0;  // número de elementos
} ;
```

Y al crear una lista vacía utilizamos la función:

```
Lista_dob_enl crear_lista_vacia(){
  Lista_dob_enl lista;
  lista.cabeza = new Nodo{0,nullptr, nullptr};
  lista.cola = new Nodo{0,lista.cabeza, nullptr};
  (lista.cabeza)->proximo = lista.cola;
  lista.num_elem =0;
  return lista;
}
```

Hagamos juntos insertar x en posición num_elem (de útimo). Sea al comienzo, al final o en el medio resulta en el mismo código, **incluso si la lista está vacía…..**



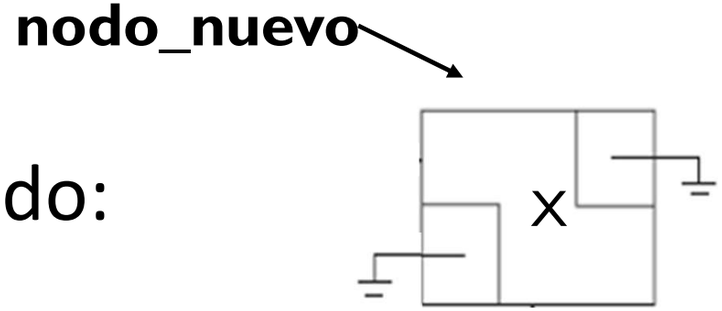


cabeza (centinela)

a y b son los objetos de la lista

rabo o cola (centinela)

Sea **lista** es un objeto **tipo** lista doblemente enlazada con centinelas (**Lista_dob_enl**).

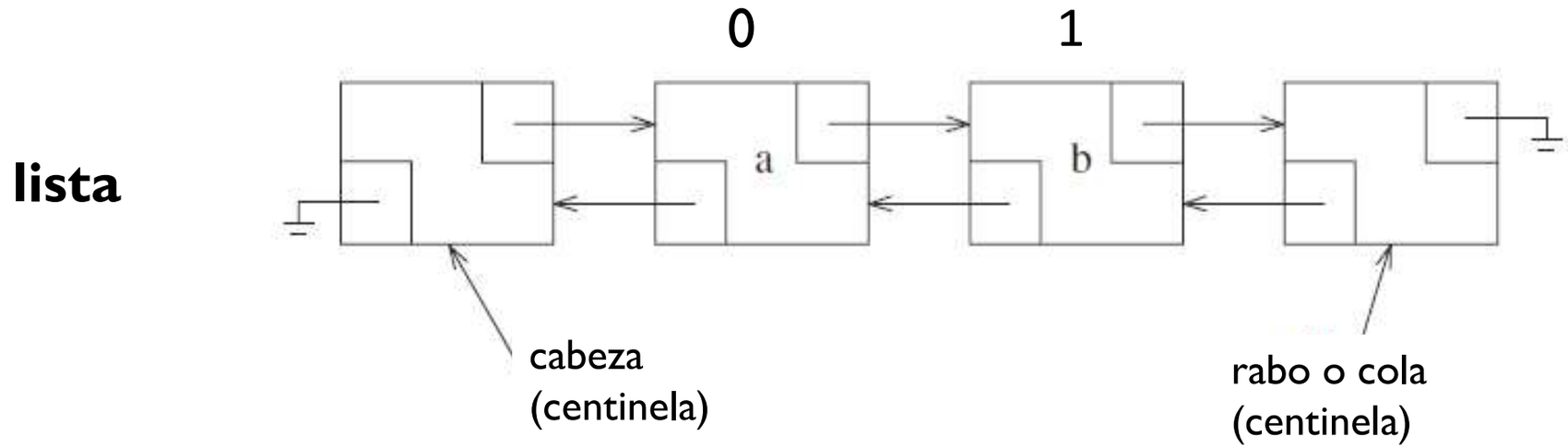


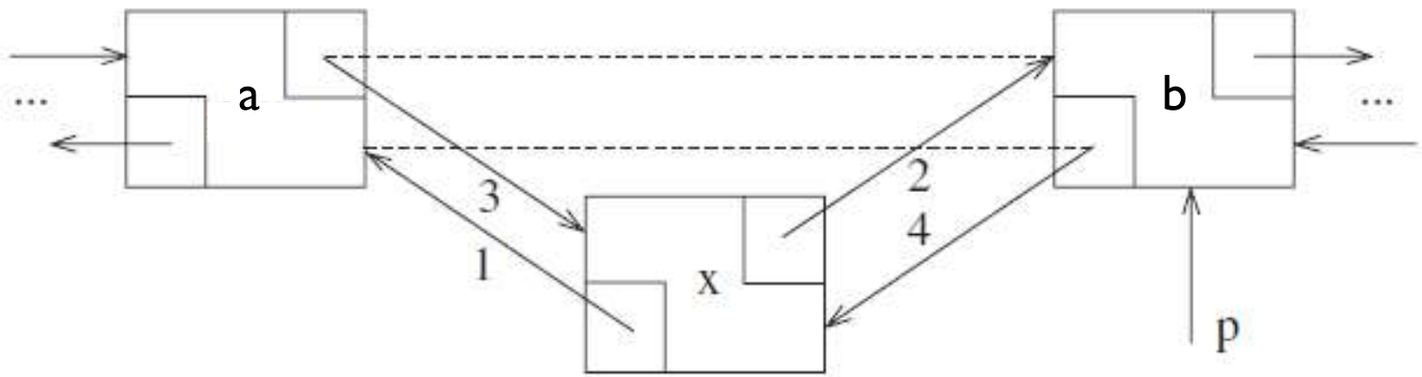
Queremos insertar en la posición **i** al nodo:

insertar(lista, x, i), suponemos 0<= i <= num_elem (Note que i puede valer num_elem que significa insertarlo de último)

Buscamos el apuntador **p** al nodo en la posición **i** en **lista** y a partir de **p** hacemos la inserción en la lista del **nodo_nuevo** antes que **p** (para que quede en la posición **i**)

Por ejemplo: insertar un nodo en la posición 1 de la lista siguiente:





Veamos el programa para insertar un nuevo elemento **x** en la posición **i.**

El siguiente algoritmo encuentra la referencia p del elemento en la posición i. Lo llamaremos **obtenerRef(lista,int i)** y devuelve p

- Inicializamos **p** = lista.cabeza.siguiente
- j=0 (me indicará la posición del elemento en el recorrido)
- Mientras (j < i) {
- j++; **p** = **p**.siguiente; }
- Devolver **p**.

```
Nodo * ObtenerRef(Lista_dob_enl  lista, int i){
   // i debe estar entre 0 y num_elem de la lista

   *Nodo p = lista.cabeza -> siguiente;
   int j=0; //me indicará la posición del elemento en el recorrido )
   while ( j < i )  {
         j++; p = p.siguiente; //p apunta al elemento en la posición j
   }
   return p;  // note que si la lista está vacía devuelve apuntador al centinela cola
}
```

Y el método insertar un elemento x en la posición i:

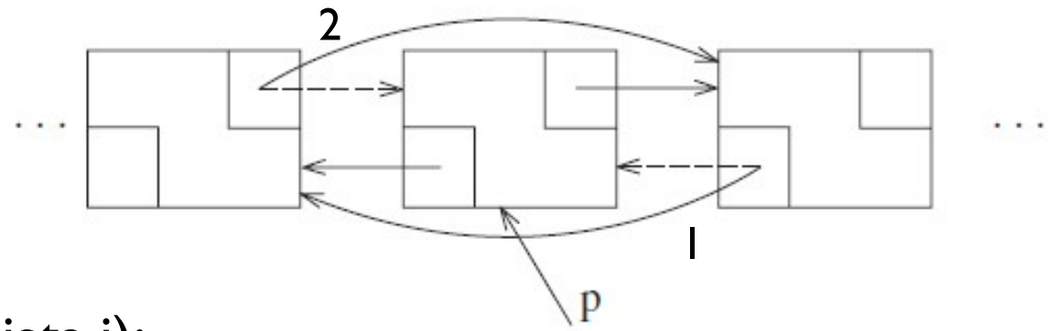
```
// insertar un nodo conteniendo el objeto x en la posición
// i de la lista, suponemos 0<=i<=num_elem
void insertar( Lista_dob_enl lista, int x, int i )

{  if ((i<0)&&(i> lista.num_elem) return;
   Nodo* p  = obtenerRef (lista, int i) ;
   Nodo* nuevo_nodo = new Nodo{ x, p.anterior, p };
   (nuevo_nodo.anterior)  ->  siguiente = nuevo_nodo;
   p.anterior = nuevo_nodo;
   num_elem ++;  // se suma 1 al número de elementos de la lista
}
```

Note que el código de insertar(lista,x,i) es independiente de si la lista es vacía, o insertamos a x de primero o de último o en otra posición. Para esto sirven los centinelas

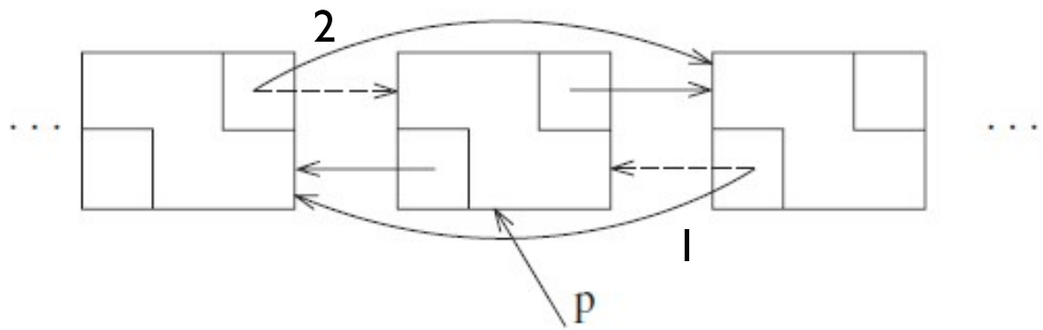
Veamos juntos con dibujos paso a paso cómo insertar un nuevo elemento **56** en la posición **1** de una lista doblemente enlazada con centinelas **Utilizando el algoritmo anterior**

La lista es <34,45,667>

Eliminar un elemento en la posición i (que suponemos válida), al igual que en **insertar** no tenemos que distinguir si es el primero o el último como casos especiales.
Buscamos el apuntador p que apunta al elemento en la posición i con el método **obtenerRef(lista, i)**



void eliminar(lista,i):
- **p** = obtenerRef(lista, i)
- Eliminar nodo referenciado por **p** (**ver la siguiente lámina**)




```
void eliminar(Lista_dob_enl   lista,  int i)
{
   Nodo*  p  = obtenerRef (int i) ; //p apuntador a objeto en posición i
   (p.siguiente) -> anterior = p.anterior;    // paso (1)
   (p.anterior) -> siguiente = p.siguiente;    // paso (2)
   delete p;
   num_elemt--;
 }
```

EJERCICIO:
Terminar de implementar en C++ la lista doblemente enlazada con centinelas, con algunas de la operaciones del tipo Lista, como:

- num_elem(Lista); // devuelve el número de elementos de la lista
- bool esVacia(Lista); // devuelve verdad si y sólo si la lista está vacía
- void vaciar(Lista); // convierte en vacía a una lista
- bool contiene(Lista, Tipo x); // devuelve verdad sii x está en la lista
- bool insertar_primero(Lista,Tipo x); // inserta x al comienzo de la lista
- bool eliminar(Lista,Tipo x); // elimina primera ocurrencia de x en la lista

- bool insertar_ultimo(Lista, Tipo x); // inserta x al final de la lista
- Tipo obtener(Lista, int idx); // devuelve el elemento en la posición idx de la lista
- Tipo asignar(Lista , int idx, Tipo newVal); // reemplaza el objeto en la posición
- // idx por newVal
- void insertar_pos(Lista , int idx, Tipo x); // inserta el objeto x en la posición idx
  // (mueve una posición los de la derecha, idx puede ser
  // una posición mas del ultimo y asi lo inserta de último)
- void eliminar_pos(Lista , int idx); // elimina el objeto en la posición idx

El siguiente algoritmo inserta a una lista existente vacía los N primeros números naturales de mayor a menor

<N-1, N-2, … , 0>

```
void hacerLista( Lista lst, int N ){

   for( int i = 0; i < N; i++ )
         insertar_primero(lst, int i);   //  insertar i de primero en la lista (insertar)
}
```

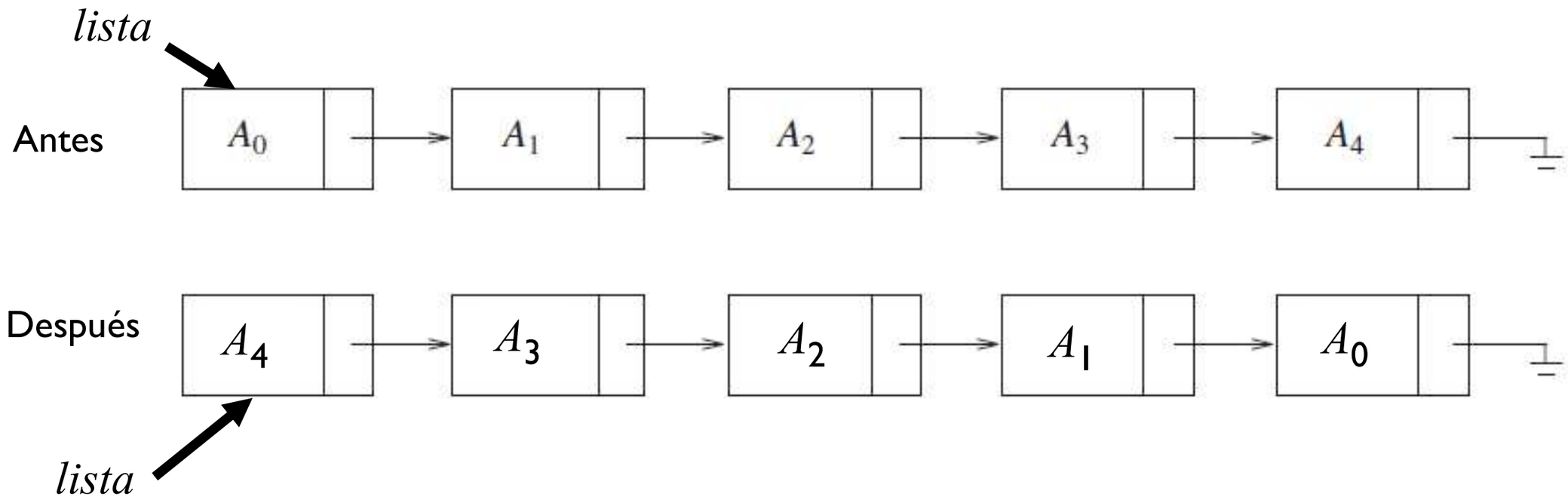
¿Que estructura de datos sería mas conveniete para reducir el número de operaciones elementales: vector o lista enlazada?

EJERCICIO para la casa Calcular el orden del número de operaciones del siguiente programa si implementamos la lista con un vector o si la implementamos con una lista doblemente enlazada con centinelas.

El siguiente algoritmo calcula la suma de los elementos de una lista de enteros:

```
int sum( Lista  lst )
{
   int total = 0; int N = num_elem(lst);
   for( int i = 0; i < N; i++ )
      total += obtener(lst, i );
   return total;
}
```

Hagamos juntos una función iterativa para invertir una lista utilizando una lista simple con apuntador al primero



Pensemos juntos como sería el programa en C++

Con un ejemplo: <1,2,3>

Iterativo:

```
void invertir_iter(Nodo* &lista){
   if (lista==nullptr) return;
   Nodo* anterior;
   Nodo* actual;
   Nodo* tmp;
   anterior = nullptr; actual = lista;
   while (actual != nullptr ){
      tmp=actual;
      actual = actual->proximo;
      tmp->proximo = anterior;
      anterior = tmp;
   }
   lista=anterior;
}
```

¿ Y uno recursivo que invierta la lista ?
Por supuesto, sin crear nuevos nodos

Recursivo: necesitamos de una función que agregue un nodo al final de una lista

```
void insertar_cola(Nodo* &lista, Nodo*
nodo){
   // nodo debe tener proximo en null
   if (lista == nullptr)
      lista = nodo ;
   else {   // lo inserta de ultimo
      Nodo *cursor = lista;
      while (cursor->proximo != nullptr)
            cursor = cursor->proximo;
      cursor->proximo = nodo ;
   }
}
```

```
void invertir(Nodo* &lista){
   if (lista==nullptr) return;
   Nodo* tmp = lista;
   Nodo* proximo = lista->proximo;
   tmp->proximo = nullptr;
   invertir (proximo);
   insertar_cola(proximo,tmp);
   lista = proximo;
}
```

Corrámoslo con la lista <1,2,3>

PREGUNTAS???