UCAB UNIVERSIDAD CATÓLICA ANDRÉS BELLO

Algoritmos y Estructuras de Datos

Oscar Meza

omezahou@ucab.edu.ve

Quis 1 al final de clases: 1 hora

Comenzamos a las 9:50 am

REPASO: Volvamos a ver como modularizamos para utilizar una posible implementación del tipo lista enlazada
Primero el archivo header Lista_Enlazada.h

```
#include <iostream>
struct Nodo {
    int data; // informacion del nodo
    Nodo *proximo;// enlace a otro
                  //  próximo nodo
};
struct Lista_enlazada  {
    Nodo* apun_cabeza;
    int num_elem;  // mantenemos el
                // número de elementos
} ;
```

```
//necesito & para pasarla por referencia
Lista_Enlazada& crearlista_vacia();
int num_elem(const Lista_enlazada &lista);
void vaciar_lista(Lista_enlazada &lista);
void insertar_primero(Lista_enlazada &lista,
int n);
void eliminar_primero(Lista_enlazada  &lista);
void imprimir(const Lista_enlazada &lista);
```

Ahora el archivo cpp , Lista_Enlazada.cpp:

```
#include <iostream>
#include "Lista_enlazada.h“

Lista_enlazada crearlista_vacia(){

       return { nullptr, 0};
}

int num_elem(const Lista_enlazada &
lista){
    return lista.num_elem;
}
```

```
void vaciar_lista(Lista_enlazada &lista){

    for (Nodo *cursor = lista.apun_cabeza,
        *temp; cursor != nullptr;      ) {
        temp = cursor;
        cursor = cursor->proximo;
        delete temp;
    }
    lista.apun_cabeza = nullptr;
    lista.num_elem = 0;

}
```

```
void insertar_primero
      (Lista_enlazada &lista, int n){

    lista.apun_cabeza =
          new Nodo{n,lista.apun_cabeza};

     lista.num_elem++;
}
```

```
void eliminar_primero
        (Lista_enlazada  &lista){
   if (lista.apun_cabeza!=nullptr){
      Nodo* temp = lista.apun_cabeza;
      lista.apun_cabeza =
           (lista.apun_cabeza)->proximo;
      delete temp;

      lista.num_elem--;
    }

}
```

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

Programa de prueba, prueba.cpp (NOTE QUE NO NOS ENTERAMOS COMO SE IMPLEMENTÓ Lista_enlazada)

```
#include <iostream>
#include "Lista_enlazada.h"
int main(){
    Lista_enlazada lista ;
    crearlista_vacia(lista);
    int n = 4; // crear cuatro elementos
    for (int i=0 ; i< n ; ++i)
        insertar_primero(lista,i);
    // imprimir lista
    eliminar_primero(lista);
    imprimir(lista);
    vaciar_lista(lista);
 }
```

Hagamos juntos la función que inserta un elemento de último en el tipo Lista_enlazada.
El prototipo sería:

```
void insertar_cola(Lista_enlazada &lista, int x)
```

```
#include <iostream>
struct Nodo {
    int data; // informacion del nodo
    Nodo *proximo;// enlace a otro
                  //  próximo nodo
};
struct Lista_enlazada  {
    Nodo* apun_cabeza;
    int num_elem;  // mantenemos el
               // número de elementos
} ;
```

La función sería:

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

Ilustremos con dibujos el paso a paso del siguiente programa:

```
int main(){
  Lista_Enlazada lista = crearlista_vacia( );
  insertar_primero(lista, int 56);
  insertar_primero(lista, int 45);
  imprimir(lista);
}
```

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

Agreguemos a la Lista_enlazada dos funciones:
- una que devuelve verdad si y solo la lista es vacía:

```
bool es_vacia(const Lista_Enlazada &lista)
```

- Otra que devuelve verdad si y solo si un elemento x está en una lista:

```
bool esta(const Lista_enlazada &lista, int x)
```

```
bool esvacia(const Lista_enlazada &lista){
    if (lista.num_elem == 0) return true;
    else return false;
}

bool esta(Lista_enlazada &lista, int x){
              //necesito & porque modifico lista
    for (Nodo *cursor = lista.apun_cabeza;
            cursor != nullptr;   cursor = cursor->proximo   )
        if (cursor->data == x) return true;

    return false;
}
```

Ejercicio para la casa:
Implementemos el tipo Conjunto de enteros con Lista_Enlazada (es decir, usando las operaciones de Lista_Enlazada). Con header y cpp

typedef Lista_Enlazada Conjunto;

Las funciones que tendrá el tipo conjunto de enteros serán:
- Crear un conjunto vacío
- Insertar un elemento en un Conjunto
- Verificar si un elemento pertence a un conjunto
- Unir dos conjuntos y devolver el conjunto union.
- Imprimir un conjunto.

Hacer un programa de prueba donde se use el tipo Conjunto.

Ahora veremos los Tipos de Datos Lista, Pila y Cola

TIPOS DE DATOS ABSTRACTOS

- Un **Tipo Datos Abstractos** es una abstracción matemática dada por un conjunto de **valores** (objetos) junto a un **conjunto de operaciones** que se pueden realizar sobre estos. Término que abreviaremos como **TDAs**

- **Por ejemplo, los números enteros** con sus operaciones de suma, resta, división, etc., es un tipo Dato Abstracto que en los lenguajes de programación ya tienen una representación concreta de éstos

- La construcción de **un nuevo tipo de datos** requiere que **las características deseadas de éste sean inicialmente especificadas de manera clara**, para luego proceder a implementarlo y obtener un tipo concreto de dato.

- El nombre **TDAs se debe a que el comportamiento deseado de estos tipos debe ser especificado de manera abstracta, independientemente de las múltiples posibles implementaciones concretas** que puedan luego construirse.

- **Para especificar** (o representar) **Tipos de Datos Abstractos utilizamos modelos matemáticos conocidos**, como por ejemplo, conjuntos, secuencias, funciones y relaciones.

- **Los TDAs permiten reforzar los conceptos "encapsulamiento de datos"** (los datos junto a sus operaciones) **y "ocultamiento de datos"** (sabemos **qué hace y no cómo se hace**, se ocultan detalles de implementación.

El Tipo de Dato Abstracto **Lista**

El Tipo Dato Abstracto LISTA

- **Una lista es una secuencia** (modelo matemático que representa una lista) de objetos de un mismo tipo $<A_0, A_{1, \ldots} A_{n-1}>$. El número de elementos **(tamaño**) de la lista es **n.** Una **lista vacía** es una secuencia vacía, su tamaño es 0. La **posición** del elemento A_i es i.

- Las operaciones básicas de una lista las presentaremos utilizando la nomenclatura de C++. Los prototipos.

El Tipo Datos Abstractos LISTA

Valores: secuencias de elementos de un mismo tipo

Operaciones básicas (las operaciones de lista varían según los autores):

- int 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 esta(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

Más operaciones básicas del tipo 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 reemplazar(Lista , int idx, Tipo newVal); // reemplaza el objeto en la posición
- // idx por newVal
- void insertar_pos(Lista , Tipo x, int idx); // 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

Ejemplo:

Insertar de último un elemento **x** a la lista **lista**, **insertar_ultimo(lista,x):**

Si la lista es <1, 2, 2, 0, -1, 4>

Y queremos insertar 7 a la lista, obtendríamos:

<1, 2, 2, 0, -1, 4, 7>

Podríamos utilizar un vector (tipo concreto) para implementar el TDA lista

Implementar una lista de elementos por un vector

(veamos un vector como un arreglo y no utilicemos las operaciones que ya vienen definidas en la biblioteca estándar para vector, solo las básicas que mencione en mis exposiciones):

La implementaríamos con un vector, de esta forma:

```
typedef std::vector<int> Lista;
```

Y una variable tipo Lista se declararía:

```
Lista lista;  // crea una lista vacía
```

Implementar una lista de elementos por un vector

La función que determina si una lista está vacía sería:

```
bool esVacia(const Lista &lista){
    return lista.empty(); // se usa la función empty() de vector
}
```

Implementar una lista de elementos por un vector:

La operación de insertar un elemento al final de la lista sería sencilla y poco costosa pues agrega un elemento al final del vector (incluso un vector vacío se crea con una capacidad inicial no vacía como vimos para arreglos dinámicos) :

```
void insertar_ultimo(Lista& lista, int x){
   lista.push_back(x);
}
```

La operación de **insertar un elemento al comienzo de la lista** es mas complicada. Utilizaremos el tipo vector como si fuese un arreglo dinámico. Si vamos a insertar un elemento de primero tenemos el método **resize( )** que cambia el tamaño del vector:

```
vector<int> vec = {1, 2, 3, 4, 5};
// Vector resized a 8
vec.resize(8);  // aumenta el tamaño a 8 y
                // contendrá {1, 2, 3, 4, 5, 0, 0,0}
```

Implementar una lista de elementos por un vector:

Y la operación **de insertar al comienzo sería** (es costosa pues tiene que mover todos los elementos del vector original una posición):

```
void insertar_primero(Lista &lista, int x){
  lista.resize(lista.size()+1);  // aumentamos el tamaño del vector en 1
  for (int i= lista.size()-2; i>=0; i--){
      lista[i+1]= lista[i];
  }
  lista[0]=x;
}
```

Si **insertamos un elemento en la posición 0** de un vector (que es un arreglo), si la lista tiene **N** elementos, ¿esta operación cuantas operaciones elementales haría?. Pregunta…

N=7 número de elementos (tamaño original del vector)

0	1	2	3	4	5	6	7
4	3	-11	4	-5	2	2	-2

insertar 6 en la posición 0

N=8

0	1	2	3	4	5	6	7	8
6	4	3	-11	4	-5	2	2	-2

Eliminar un elemento siempre sería del orden de N (tamaño del vector) **operaciones en el peor caso**, por ejemplo si eliminamos el primero hay que desplazar todos los demás elementos una posición a la izquierda y hacer un **resize** de uno menos del vector.

Obtener un elemento en una posición dada (obtener(lista, int idx)) es una operación elemental por ser un arreglo se **devuelve lista[idx]**

Hagamos juntos la operación de eliminación de la primera ocurrencia, comenzando desde la posición 0, del elemento X (si existe, si no existe no hacer nada) con la implementación de lista como un vector:

typedef vector<int> Lista;

Prototipo:

```
void eli_pri_ocu(Lista &lista, int x)
```

Ejemplo <1,2,2,3,2> y 2 resultado <1,2,3,2>

<1,2,2,3,2,5 > y 5 resultado <1,2,2,3,2>

```
void eli_pri_ocu(Lista &lista, int x){

   for (int i = 0; i < lista.size() ; i++)
       // encontrar la posición del elemento
       if (lista[i] == x) {
          for (int j = i; j<lista.size( )-1 ; j++)
             lista[j] = lista[j+1]; // si quiero
          //preservar el orden de los elementos
          // si no, basta con intercambiar el elemento
          // en posición i con el último en posición
          // lista.size()-1
          lista.resize(lista.size()-1);
          return;
       }
}
```

Ejercicio para la casa:

- Hacer todas las operaciones del tipo Lista implementado por un vector

- Eliminar todos los elementos iguales a x

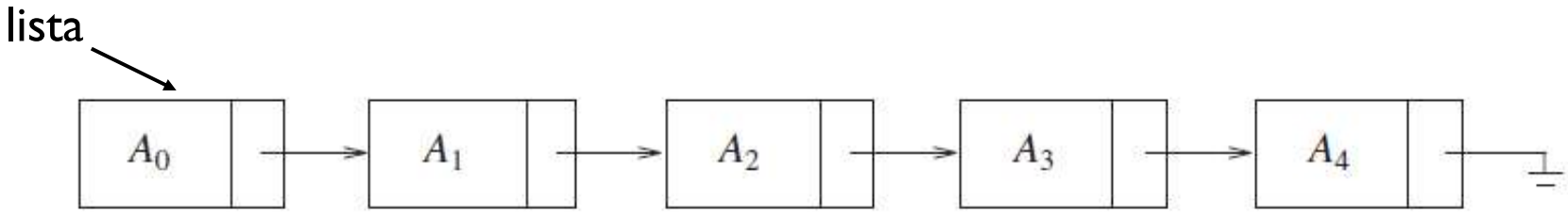
También hemos podido hacer una implementación que comience con un vector de muchos elementos para no tener que hacer resize( ) cada vez que insertemos o eliminemos y tener una variable "**int num_elem**" con el número de elementos de la lista en el vector:

```
struct Lista {
    vector<int> v (100);
    int num_elem = 0;
}
```

Solo en caso de que se trate de insertar un elemento y num_elem es 100 se haría un resize( ) del vector

Queda como ejercicio implementar el tipo Lista con esta estructura de datos

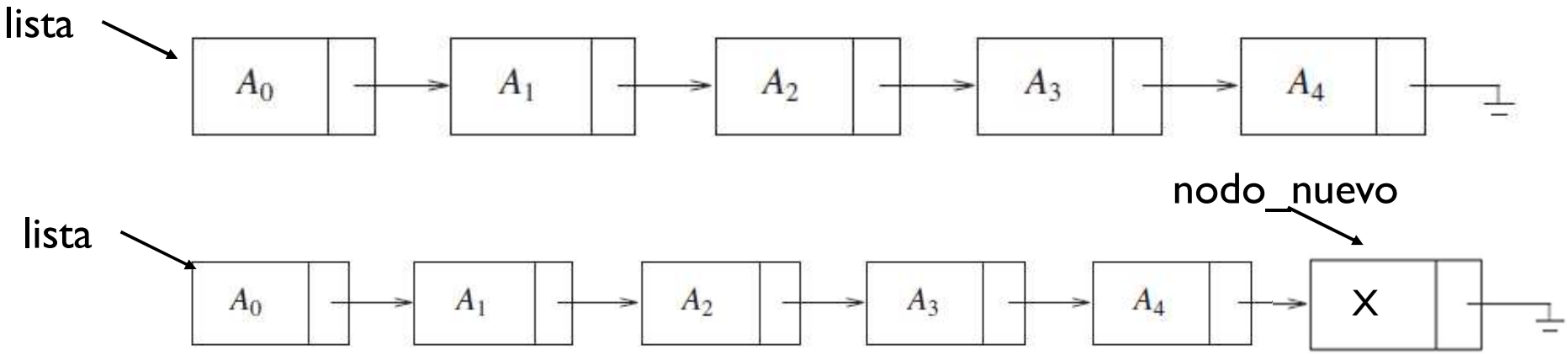
Otra implementación de Lista es utilizando una estructura de **lista enlazada con apuntador a primero (Lista_enlazada) como ya la vimos:**



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**.

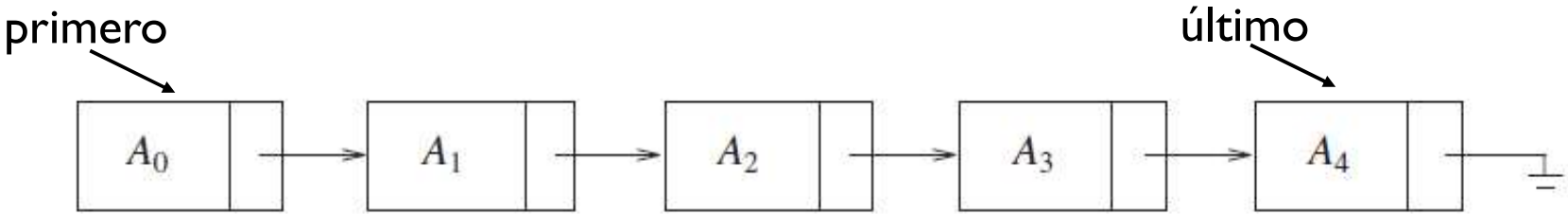
Ejemplo: Insertar X en la lista de último **(insertar_cola ()), el número de operaciones elementales sería proporcional a N, donde N es el tamaño de la lista**

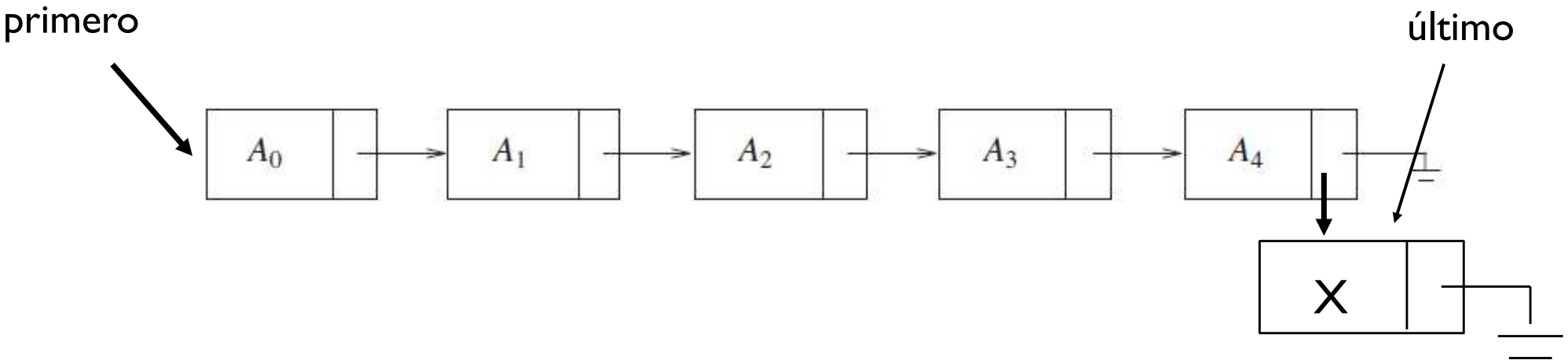
Se crea un nodo con el elemento X y se recorre la lista hasta el final (si lista es vacía se trata distinta a si no es vacía)



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:

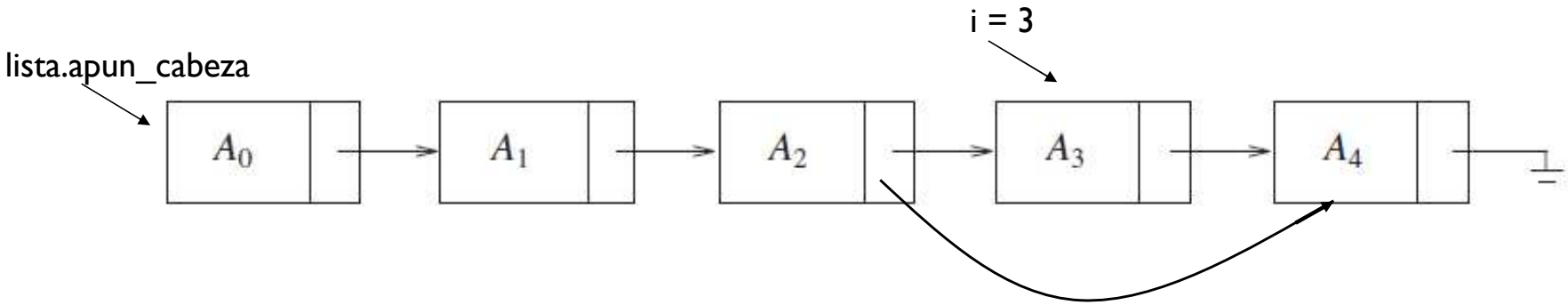
typedef Lista_enlazada Lista;

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:

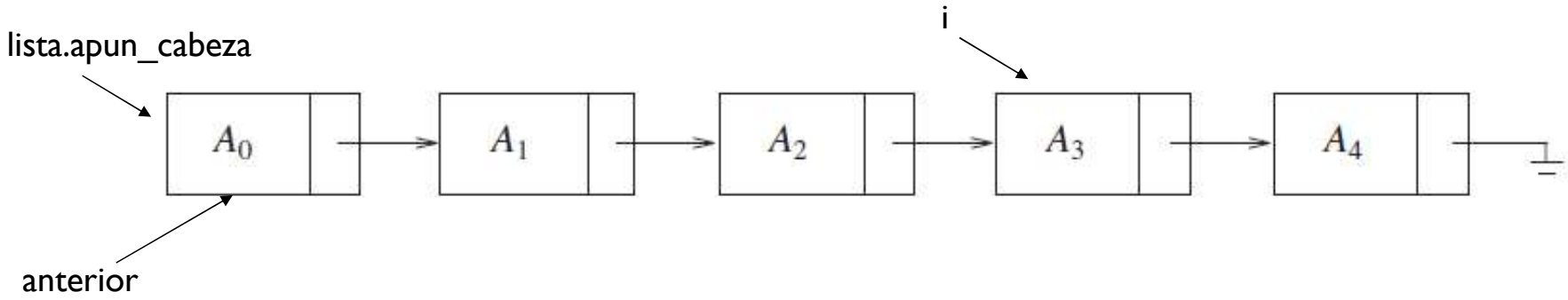

i = 3
lista.apun_cabeza
A_0
A_1
A_2
A_3
A_4

Si **i = 0**

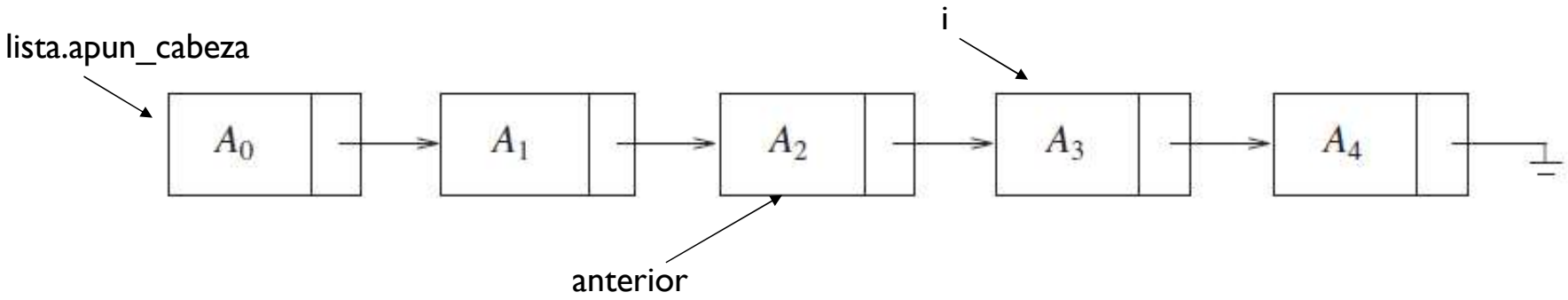
Solo necesitamos asignar a **lista.apun_cabeza** el próximo de **lista.apun_cabeza:**

```
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:

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

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

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:

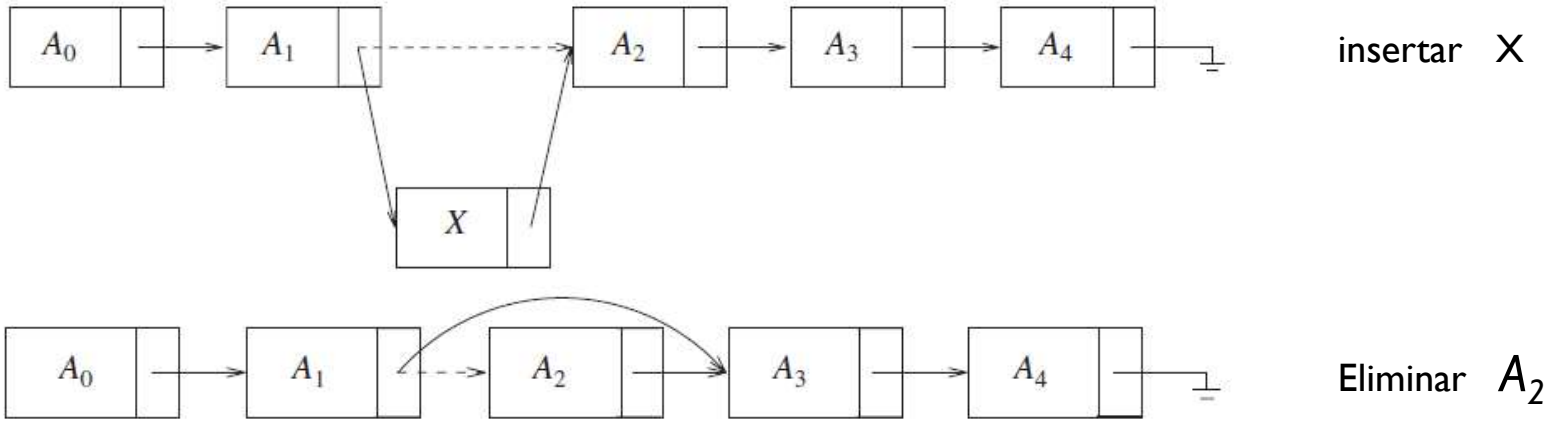
```
Nodo*  anterior = lista.apun_cabeza;
int j = 0;  // indica la posición del nodo
      // anterior al que hay que eliminar
while (  j <  i -1) { // si todavía no estamos en la pos anterior a i
         j++;
        anterior = anterior->proximo;   // anterior apunta a elem  en posición j
}
Nodo* temp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

Veamos la función completa……

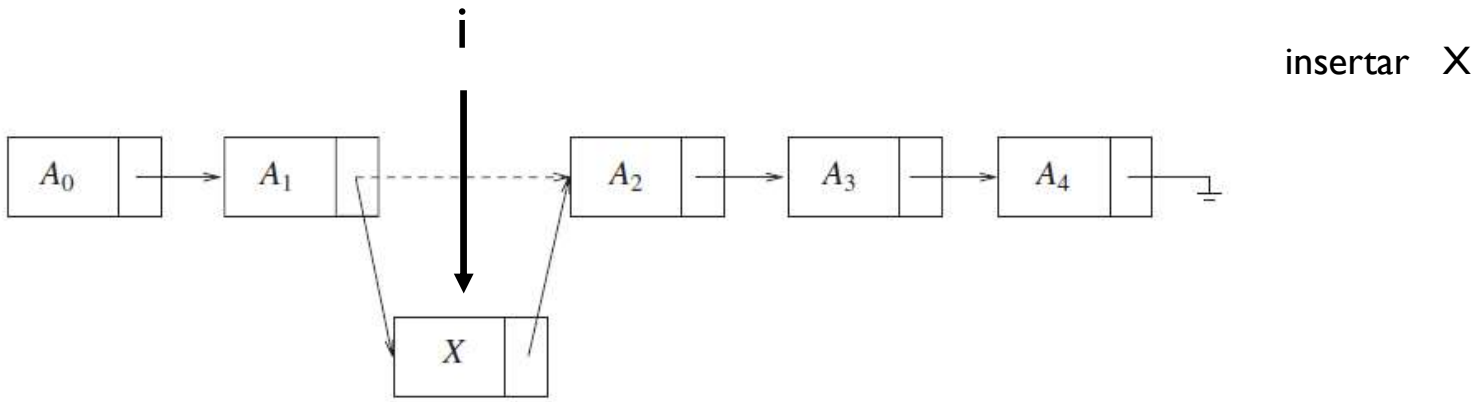
```
bool eliminar(Lista_enlazada& lista, int  i) {
// Elimina el elemento en la posición i de la lista
// Devuelve falso si y solo si  i no está entre 0 y
// num_elem-1,

  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**

PREGUNTAS???