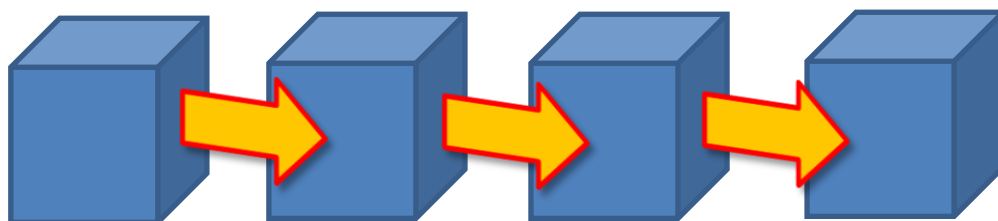




UCAM
UNIVERSIDAD
CATÓLICA DE MURCIA

2020

GRUPO  TUDIUM
FORMACIÓN



Tema 5

LISTAS

STUDIUM

www.grupostudium.com
informacion@grupostudium.com
954 539 952

Introducción

Cuando necesitamos **guardar información** podemos echar mano de las sencillas **variables** de tipos básicos, hasta complejos **arrays** cuyos datos viene definidos por **estructuras** más o menos complejas. Pero en todos los casos, siempre debemos saber, más o menos, de antemano, la **cantidad de elementos** con los que vamos a trabajar. Pero esto no siempre es posible. En numerosas ocasiones cuando desarrollamos una aplicación no sabes a priori la cantidad de datos que va a albergar nuestro programa, con lo cual, todas estas **estructuras estáticas de datos** no son suficientes.

Comenzamos a presentar una serie de **estructuras dinámicas de datos** que nos van a permitir ampliar o reducir la cantidad de memoria reservada para guardar información a medida que lo vamos necesitando; es decir, nuestro programa a medida que requiera espacio para guardar datos los demandará, y, a medida que no necesite dichos recursos, los irá liberando para que otros procesos puedan usar dichos recursos.

Estamos hablando de las **Listas**, las **Pilas**, las **Colas**, los **Árboles** y los **Grafos**. Todos ellos son estructuras dinámicas de datos. Se diferencia básicamente por la forma de trabajar con los elementos que contiene, la manera de distribuir la información a lo largo y ancho de toda la estructura, etcétera.

A pesar de las diferencias, veremos ahora una serie de conceptos en los que se basarán todo el desarrollo posterior de estas estructuras.

Asignación dinámica de memoria

Comenzamos con la **asignación dinámica de memoria**, o lo que es lo mismo, cómo podemos usar la memoria RAM a demanda, a medida que la vamos necesitando; e igualmente cuando ya no la necesitemos cómo liberarla.

Otra de las grandes ventajas de la utilización de **punteros** es la posibilidad de realizar una **asignación dinámica de memoria**. Esto significa que la reserva de memoria se realiza dinámicamente en tiempo de ejecución, no siendo necesario entonces tener que especificar en la declaración de variables la cantidad de memoria que se va a requerir. La reserva de memoria dinámica añade una gran flexibilidad a los programas porque permite al programador o programadora la posibilidad de reservar la cantidad de memoria exacta en el preciso instante en el que se necesite, sin tener que realizar una reserva por exceso en prevención a la que pueda llegar a necesitar.

Dado que los punteros se pueden aplicar a cualquier tipo de variable se puede entonces realizar una asignación de memoria dinámica para cualquier variable. La función **malloc()** es la que se utiliza para realizar una reserva de memoria y se encuentra en el archivo de cabecera `<stdlib.h>`.

Vemos ahora un ejemplo simple para reservar espacio de memoria para un entero:

```
main.c X
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      int *dato_simple;
7      dato_simple = (int *) malloc (1*sizeof(int));
8      printf("Dame un valor:");
9      scanf("%d", dato_simple);
10     printf("Escribiste un %d", *dato_simple);
11     return 0;
12 }
```

```
"C:\Users\Stodium\Desktop\Jorge\UCAM\Fundam...
Dame un valor:15
Escribiste un 15
Process returned 0 (0x0)   execution time : 48.150 s
Press any key to continue.
```

NOTAS:

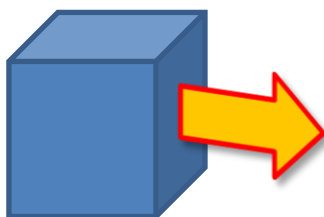
- En la **línea 6** se declara un puntero a entero.
- En la **línea 7** se asigna dinámicamente espacio a dicho puntero para guardar un dato posteriormente en **línea 9**. La función `malloc()` lleva un casting a puntero entero (`int *`) y como argumentos se le envía el tamaño que debe reservar; en este caso como se trata de un entero, lo indicamos como `1*sizeof(int)`.

Nodos

Los **nodos** son las **porciones de información** que se guardan en estas estructuras. Si tenemos una estructura llamada **Cliente** y necesitamos guardar veinte clientes en nuestro programa, tendremos que usar tantos nodos como clientes queramos guardar. Pero además de la información, los nodos disponen de un mecanismo para enlazarse unos con otros, de manera que dicha información no está dispuesta de cualquier forma, sino que sigue unas reglas bien definidas, según sea el tipo de estructura de la que estemos hablando, a saber, si **Lista**, si **Pila**, etcétera.

Un nodo puede guardar, por un lado, desde un simple número entero, hasta arrays de complejas estructuras (Struct). Y por otro, guardará uno o varios punteros hacia el resto de los nodos con los que se relaciona cada uno, como veremos más adelante.

La mínima representación de un **nodo** podría ser la siguiente, una parte que guarda la **información** y otra parte (un puntero) que guarda la **dirección al siguiente** elemento de la estructura:



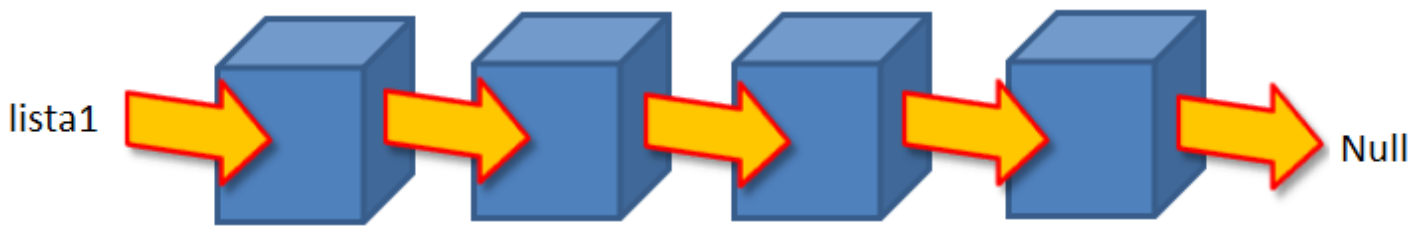
Luego, dependiendo de la estructura dinámica de la que tratemos, tendremos alguna que otra modificación; pero esta representación es la mínima.

Listas versión estática

Una **lista** consiste en un conjunto de elementos de algún tipo básico (int, float, char, ...) o estructurado (struct), que se relacionan entre sí indicándose quién va detrás de quién. Las listas que veremos tienen las siguientes características:

- En cada nodo tendremos un dato y una referencia al siguiente nodo.
- Existe una referencia que indica el Primer nodo que será la representación de la lista en sí y que al principio estará vacía referenciado a NULO.
- El último elemento de la lista referencia a NULO.
- El tamaño de la lista es indefinido pudiendo crecer o reducirse según se necesite.

Una posible representación de una lista llamada **lista1** podría ser la siguiente:



Lo que llamamos **lista1** sería realmente un **puntero** que contiene la dirección del primer elemento de la lista y el último elemento apunta a Null indicando así que se trata del elemento que está en dicha posición.

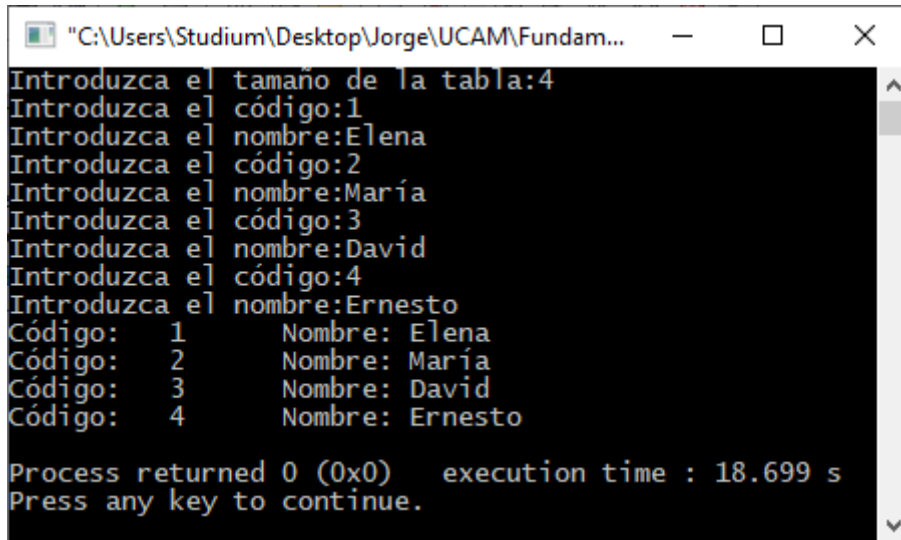
Los elementos en la lista tienen un **orden**: está el primero, el último, y cada uno conoce quien le sigue.

Una tabla o un array podría asimilarse a una lista. A esto llamaremos **lista versión estática**, pues no cumple la última condición de poder crecer o decrecer indefinidamente.

El siguiente ejemplo pregunta por teclado el tamaño de un array o tabla, tras lo cual la genera de forma dinámica. En su interior guardamos registros compuestos por un entero y una cadena:

```
main.c X
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  // Definición de tipoElemento
5  typedef struct
6  {
7      int codigo;
8      char nombre[10];
9  } tipoElemento;
10
11 // Declaración de funciones
12 tipoElemento* declararTabla(int);
13
```

```
14 // Programa principal
15 int main()
16 {
17     int i, tam;
18     tipoElemento *p;
19     printf("Introduzca el tama%co de la tabla:", 164);
20     scanf("%d", &tam);
21     p = declararTabla(tam);
22     if(p)
23     {
24         for(i = 0; i < tam; i++)
25         {
26             printf("Introduzca el c%codigo:", 162);
27             scanf("%d", &p->codigo);
28             fflush(stdin);
29             printf("Introduzca el nombre:");
30             gets(p->nombre);
31             fflush(stdin);
32             p++;
33         }
34         p = p - tam; // Volver al principio
35         for(i = 0; i < tam; i++)
36         {
37             printf("C%codigo: %3d \t Nombre: %s\n", 162, p->codigo, p->nombre);
38             p++;
39         }
40     }
41     return 0;
42 }
43
44 // Implementación de funciones
45 tipoElemento* declararTabla(int tam)
46 {
47     tipoElemento *p;
48     p = (tipoElemento*) calloc(tam, (sizeof(tipoElemento)));
49     if(!p)
50     {
51         printf("Error en asignación de memoria");
52     }
53     return(p);
54 }
```



```
"C:\Users\Stadium\Desktop\Jorge\UCAM\Fundam...
Introduzca el tamaño de la tabla:4
Introduzca el código:1
Introduzca el nombre:Elena
Introduzca el código:2
Introduzca el nombre:María
Introduzca el código:3
Introduzca el nombre:David
Introduzca el código:4
Introduzca el nombre:Ernesto
Código: 1      Nombre: Elena
Código: 2      Nombre: María
Código: 3      Nombre: David
Código: 4      Nombre: Ernesto

Process returned 0 (0x0)   execution time : 18.699 s
Press any key to continue.
```

NOTAS:

- **Líneas 5 a 9:** Definición de una estructura llamada **tipoElemento** compuesta por un entero y una cadena.
- **Línea 12:** Declaración de la función **declararTabla** que recibe un entero, el tamaño de un array, y devuelve un puntero al primer elemento de la memoria reservada para albergar la información oportuna.
- **Línea 21:** Llamada a la función para reservar espacio según el tamaño indicado por la cantidad de elementos (tam) y el tipo de dato (tipoElemento).
- **Líneas 24 a 33:** Bucle que pregunta un código y un nombre por cada elemento de la estructura creada. La instrucción **fflush(stdin)** limpia el buffer del teclado para que lo que se escribe para un elemento no afecte al siguiente, por ejemplo, un Enter.
- **Línea 34:** Colocamos el puntero al principio del todo restando a la posición actual (p) el número de elementos que contiene (tam).
- **Líneas 35 a 39:** Bucle que recorre toda la estructura para mostrar el contenido.
- **Línea 47:** Declaramos una variable puntero a tipoElemento.
- **Línea 48:** Reservamos espacio para la estructura deseada según el tamaño indicado por la cantidad de elementos (tam) y el tipo de dato (tipoElemento). La función **calloc()** se castea al tipo puntero adecuado y lleva dos parámetros, el primero es el número de elementos y el segundo es el tamaño de cada elemento; de esta forma calcula el espacio total necesario.

Lista versión cursor

Si hacemos el tratamiento de un array con punteros, ya tendremos una lista igualmente. Muy similar al caso visto en el apartado anterior. Ni esa, ni esta es la mejor forma de trabajar con listas.

Listas dinámicas

Ahora veremos la mejor forma de tratamiento de listas en lenguaje C. nos centraremos en las siguientes operaciones sobre listas:

- **Crear** una lista vacía
- **Insertar** un elemento en la lista
 - Al principio
 - Al final
 - En una posición determinada
- **Recorrer** una lista
- **Eliminar** un elemento:
 - El primero
 - El último
 - Uno intermedio
 - Todos los elementos

Crear una lista vacía

Para construir una lista, lo primero que debemos hacer es definir la estructura del **nodo** mediante un **registro** (struct):

```
// Definición del nodo
typedef struct estructura
{
    int codigo;
    char nombre[10];
    struct estructura *siguiente;
}nodo;
```

El **registro** se llama **estructura**, pero el **tipo definido** con typedef se denomina **nodo**. Observad cómo en su interior aparece un puntero al siguiente registro **estructura**. Debemos hacerlo así debido a que, en ese preciso instante, **nodo** aún no ha sido definido.

Lo siguiente que haremos será declarar e implementar una función que permita crear un nodo nuevo a demanda del usuario:

Sería así su declaración:

```
nodo* nuevo();
```


Y así su implementación:

```
nodo* nuevo()  
{  
    nodo *p;  
    p = (nodo*) malloc(sizeof(nodo));  
    p->siguiente = NULL;  
    return(p);  
}
```

Observad cómo se reserva espacio con **malloc()** y cómo se asigna el valor nulo al puntero **siguiente**.

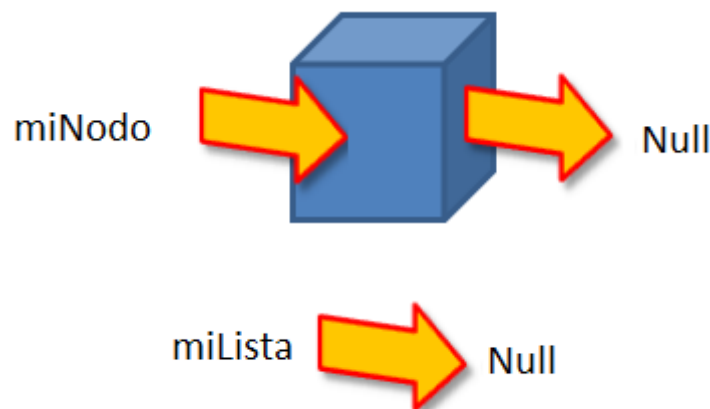
Para **crear** una **lista vacía**:

```
nodo* miLista;  
miLista = NULL;
```

Inserción de elementos

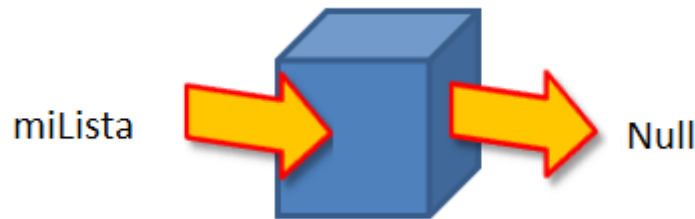
Para **insertar un elemento al principio** debemos contemplar dos situaciones: que la **lista esté vacía**, en cuyo caso no hay ni principio ni final, sería lo mismo, y por otro lado que la lista tenga al menos **un elemento**.

En el primer caso es sencillo. Tenemos **un nodo** llamado **miNodo** con datos y una **lista vacía** llamada **miLista**:

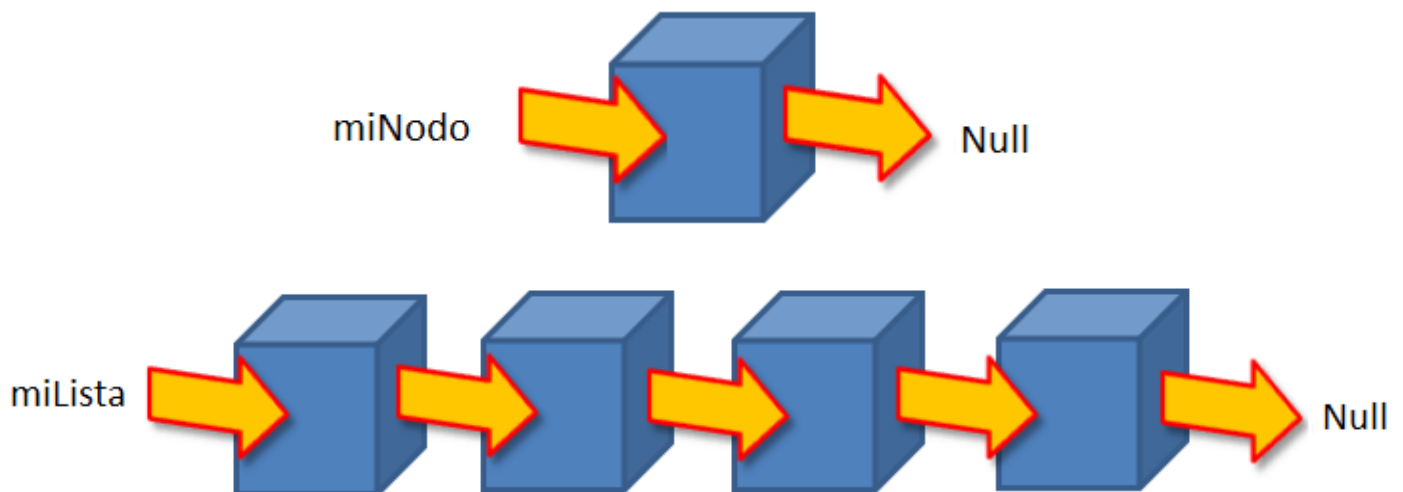


Tanto **miLista** como **miNodo** son **punteros**, con lo cual, lo único que debemos hacer, usando la aritmética de punteros, es que **miLista** apunte a **miNodo**:

```
nodo* miLista = nuevo();  
nodo* miNodo = nuevo();  
miNodo->codigo = 3;  
strcpy(miNodo->nombre, "Jacinto");  
miLista = miNodo;
```



Ahora el segundo caso, en el que insertamos un nodo (**miNodo**) al principio de una lista (**miLista**) que ya tiene nodos insertados:



Debemos hacer dos pasos:

1. Que el puntero **siguiente** de **miNodo** apunte a **miLista**
2. Que **miLista** apunte ahora a **miNodo**

Paso 1

```
nodo* miNodo = nuevo();  
miNodo->codigo = 3;  
strcpy(miNodo->nombre, "Jacinto");  
miNodo->siguiente = miLista;  
miLista = miNodo;
```

Paso 2

Ya tendríamos nuestra lista (**miLista**) con un nodo más al principio (**miNodo**).

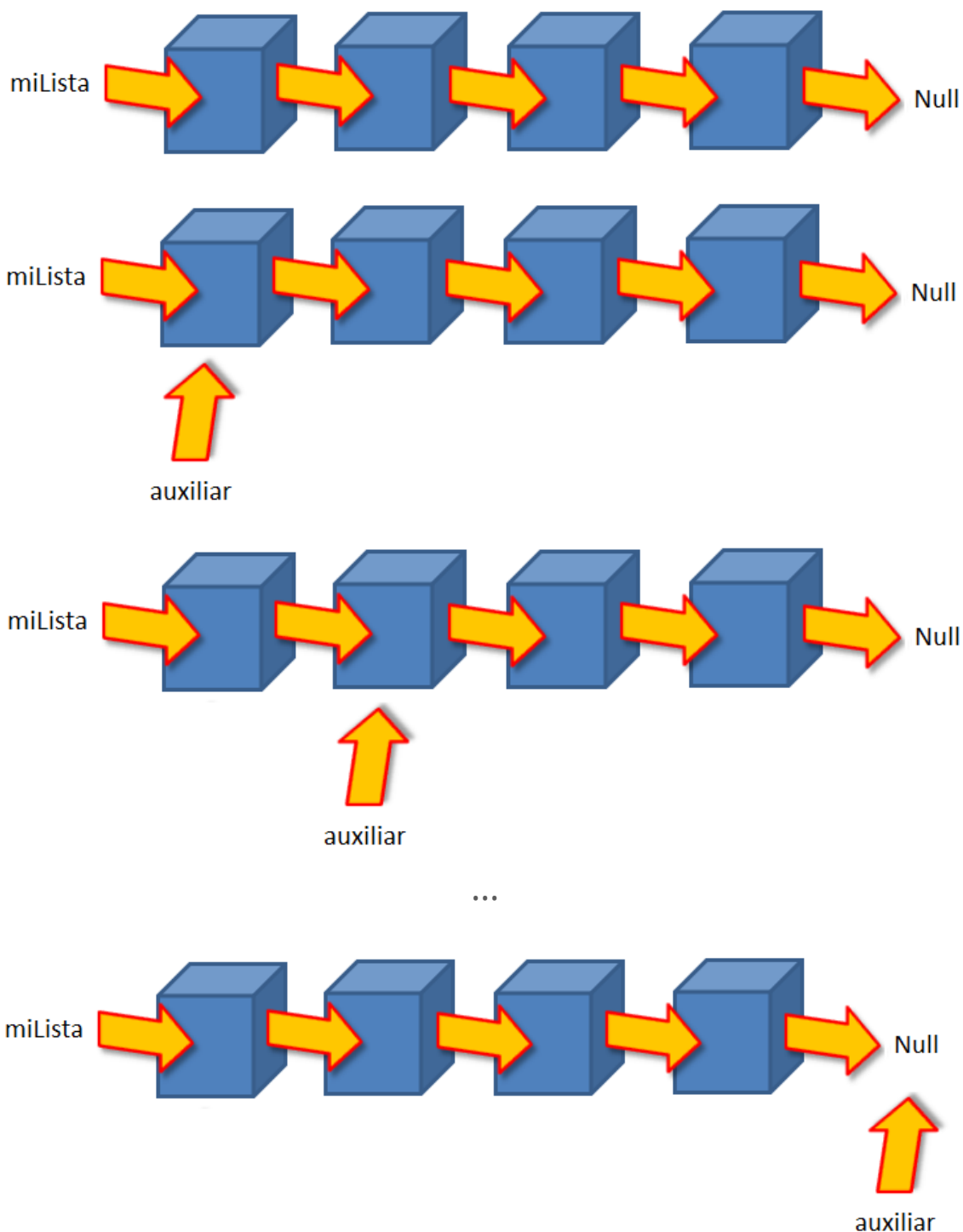
Recorrer los elementos de una lista

Continuemos con otras operaciones antes de hacer un ejemplo completo. Ahora veremos la forma de **recorrer** una lista desde el primer elemento al último.

Consistirá en pasar por todos los elementos de una lista hasta el final o bien hasta un elemento en concreto. Para ello, necesitaremos un puntero auxiliar que recorrerá la lista. El proceso es así de sencillo:

1. Creamos el puntero **auxiliar** de tipo nodo.

2. Hacemos que ese puntero **auxiliar** apunte al inicio de la lista, es decir a **miLista**.
3. Mientras no se cumpla alguna condición, como por ejemplo que el puntero **auxiliar** llegue al final, procesamos el nodo apuntado por **auxiliar** y lo avanzamos para que apunte al siguiente.



Veamos con código cómo se haría esto:

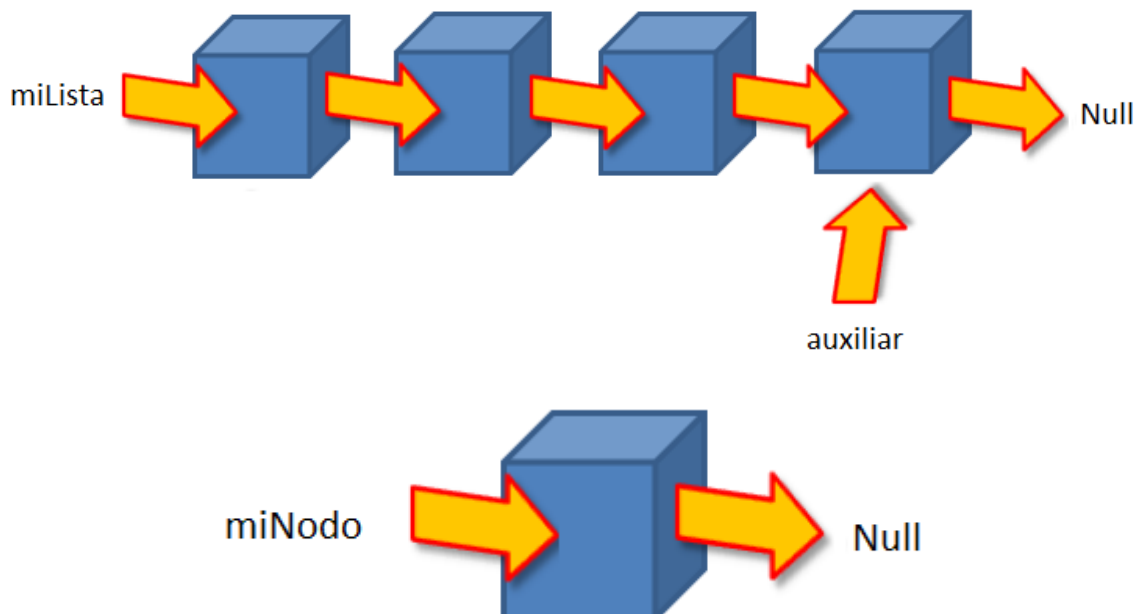
```
nodo* auxiliar = nuevo();  
auxiliar = miLista;  
  
while(auxiliar!=NULL)  
{  
    // Procesar el nodo apuntado por auxiliar  
    printf("Código: %3d \t Nombre: %s\n", 162, auxiliar->codigo, auxiliar->nombre);  
    // Avanzamos el puntero auxiliar  
    auxiliar = auxiliar->siguiente;  
}
```

Paso 1

Paso 2

Paso 3

Si hacemos una pequeña modificación a este bucle, podemos hacer la **inserción de un elemento al final**. Lo que hacemos es simplemente avanzar el puntero **auxiliar** hasta quedarnos en el último elemento de la lista, y entonces, hacer la inserción:



```
nodo* auxiliar = nuevo();  
auxiliar = miLista;  
  
while((auxiliar->siguiente)!=NULL)  
{  
    // Avanzamos el puntero auxiliar  
    auxiliar = auxiliar->siguiente;  
}  
(auxiliar->siguiente) = miNodo;
```

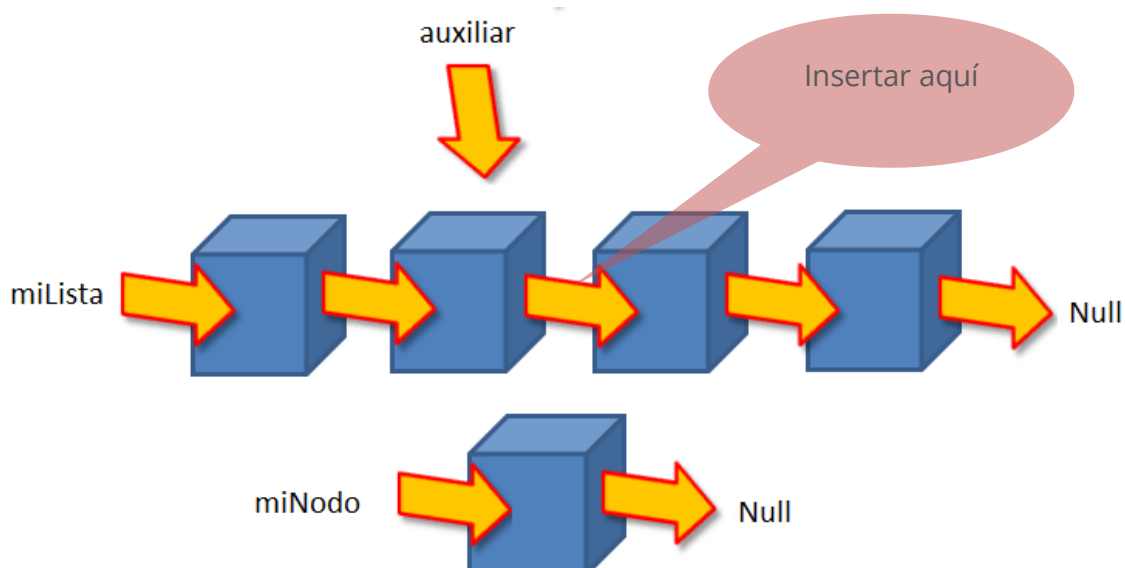
Lo que hacemos es cambiar el valor de **auxiliar->siguiente** que ahora vale NULL, por la dirección donde se encuentra el nuevo nodo (**miNodo**). Cuidado porque se ha cambiado también la condición del bucle.

Para insertar un elemento en una **posición intermedia**, simplemente debemos usar el bucle visto en el **recorrido de una lista**, cambiando la condición del bucle y haciendo la inserción dentro del propio bucle. Supongamos que buscamos por el código, de manera, que una vez encontrado, insertaremos el nuevo elemento detrás de dicho elemento.

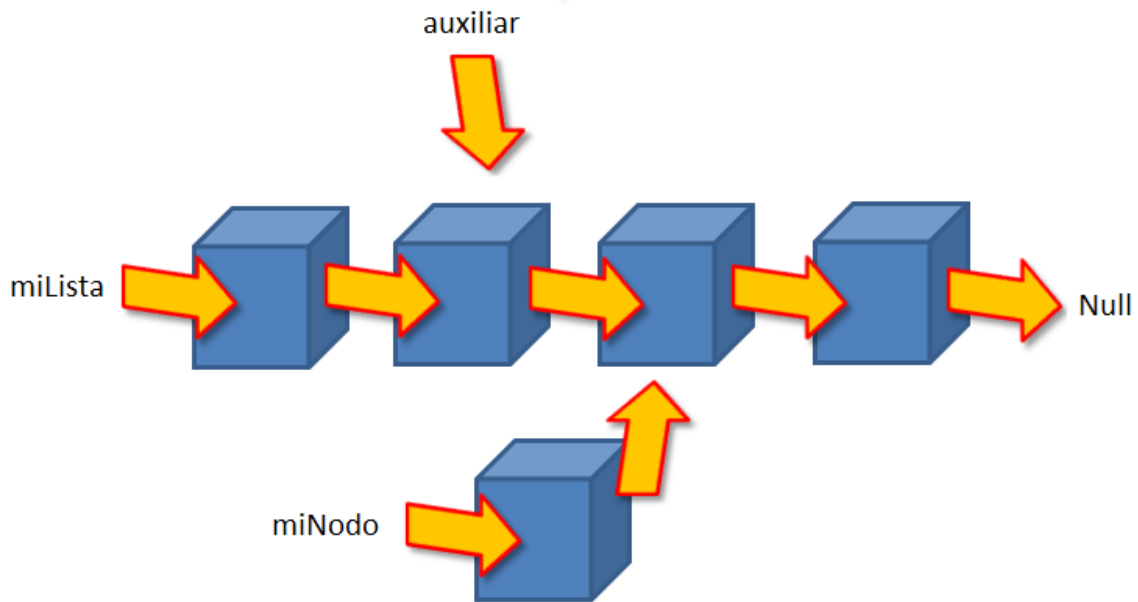
Las operaciones que tenemos que hacer son las siguientes, una vez encontrado el "sitio" a insertar:

1. Suponiendo que **auxiliar** apunta al elemento tras el que insertar, debemos hacer que el puntero al siguiente del **nuevo elemento**, apunte a donde apunta ahora **auxiliar->siguiente**.
2. Y ahora, hacemos que **auxiliar->siguiente** apunte al nuevo elemento.

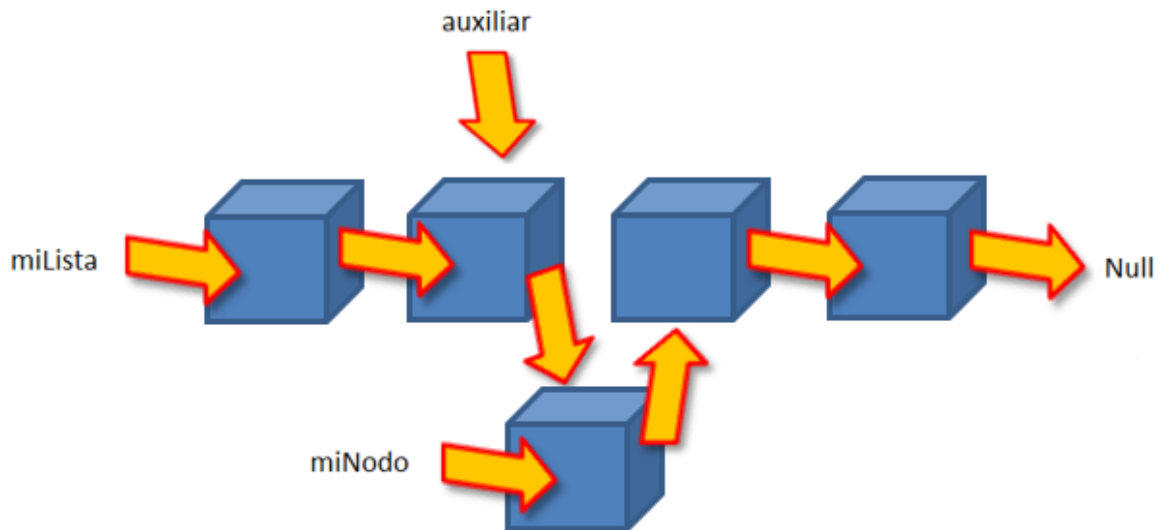
Situación inicial:



Tras el primer paso tenemos ahora:



Y por último tenemos la **lista** con el elemento insertado y todos los punteros apuntando al nodo correspondiente:



Eliminar elementos de una lista

Como se ha comentado con anterioridad, podemos eliminar el **primer** elemento de la lista, el **último**, uno **intermedio** o vaciar **por completo** la lista.

Esta última opción es la más sencilla de implementar. Si tenemos una lista llamada **miLista** con un número determinado de elementos, para **vaciarla por completo**, simplemente debemos hacer que el puntero de inicio, en lugar de apuntar al primer elemento, apunte a **nulo**. Así perdemos la referencia a los elementos de la lista y podemos comenzar a meter otros elementos en dicha lista que ahora estará vacía.

Sería interesante que **liberemos el espacio** ocupado de todos y cada uno de los nodos que componen la lista inicial para conseguir así una optimización de la aplicación que está usando la lista en cuestión. Sería un simple recorrido de la lista liberando todos y cada uno de los nodos por los que pasamos.

Gráficamente podemos ver este vaciado por completo de la siguiente forma partiendo de esta lista:



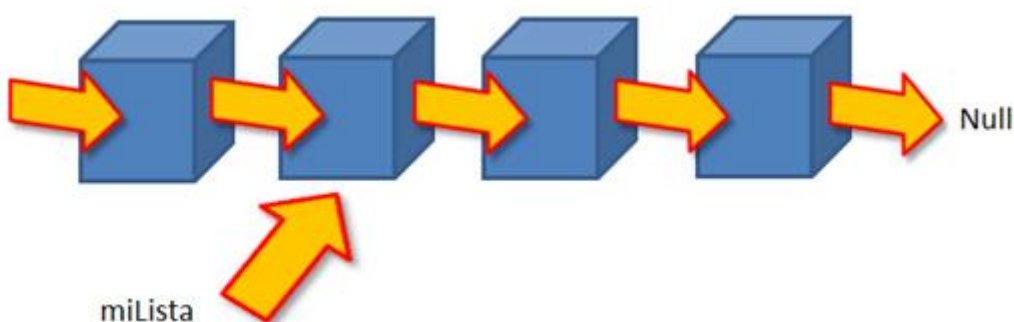
Al vaciar, simplemente debemos quedarnos con esto:



Si queremos borrar el **primer elemento** debemos hacer que el puntero **miLista**, apunte al segundo elemento de la lista, o lo que es lo mismo, que contenga el valor del puntero Siguiente del primer nodo. Si partimos de esta lista:

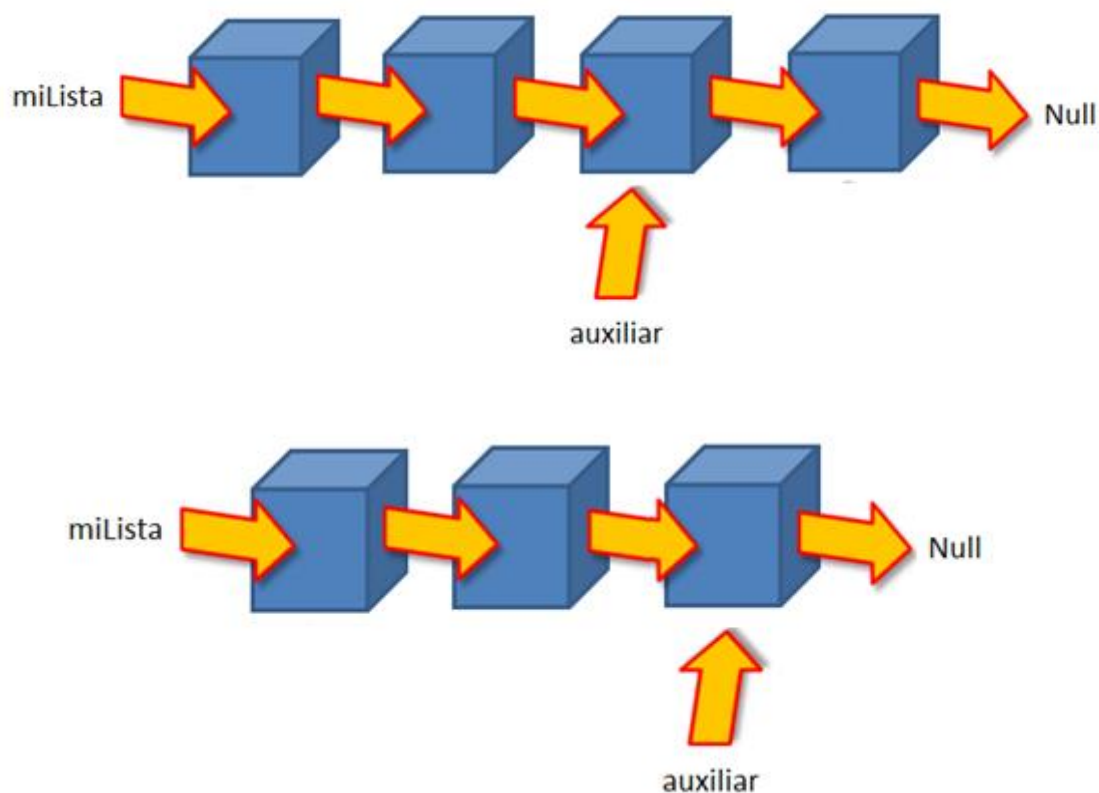


Para eliminar el primero haremos lo siguiente:

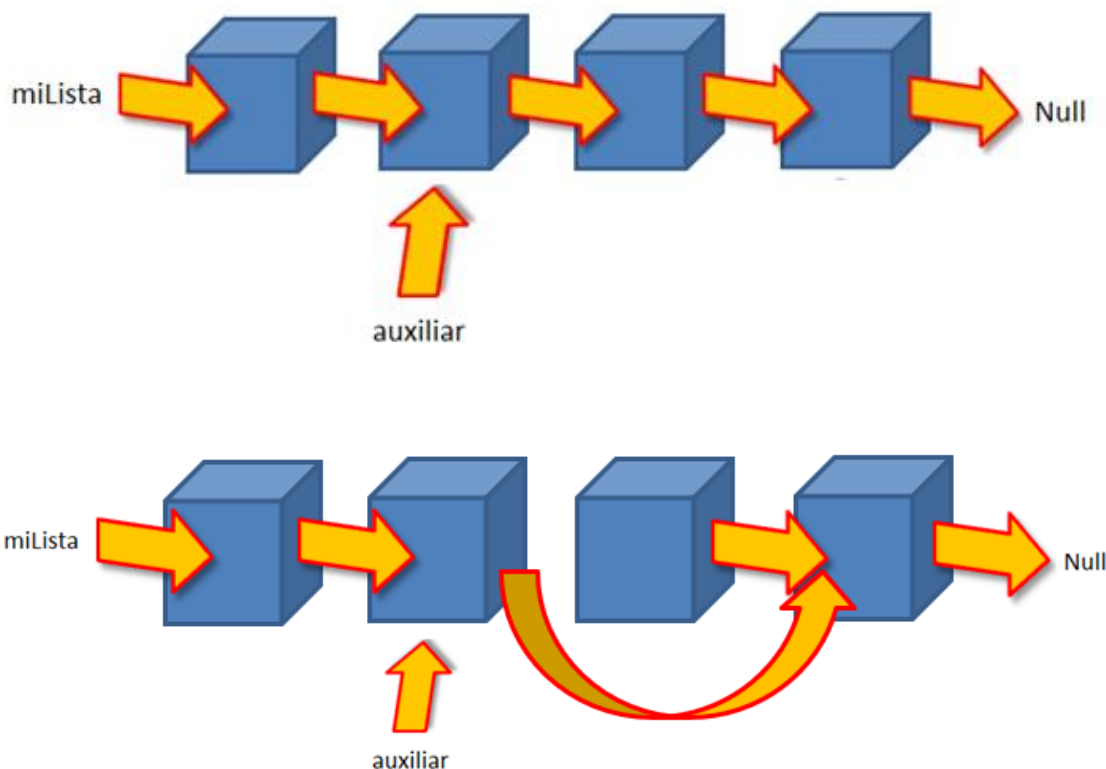


Ahora **miLista** apunta al elemento al que apuntaba antes el primer elemento, es decir, al segundo, que ahora se convierte en el primero por estar apuntado directamente por **miLista**. No olvidar liberar el espacio ocupado por este primer elemento eliminado.

Si el elemento a eliminar es el **último** debemos primero colocar un puntero **auxiliar** en el penúltimo elemento de la lista y hacer que este apunte a Nulo, en lugar del elemento al que actualmente apunta. Luego, simplemente debemos liberar el espacio de este nodo que se ha quedado fuera de la lista. Gráficamente sería algo similar a estos dos pasos:



Por último, si queremos borrar **un elemento en una posición concreta**, lo que debemos hacer es recorrer la lista con un puntero **auxiliar** que se quede justo ANTES del elemento a eliminar. Y ahora debemos hacer que se “salte” de la lista el elemento en cuestión. Jugaremos con los punteros: el elemento al que apunta Auxiliar debe apuntar al elemento al que apunta ahora el elemento a borrar y así excluimos ese elemento. En las siguientes imágenes vemos gráficamente el proceso:



Lista simple con cabecera

Hasta ahora, hemos contemplado siempre la opción de una lista con un **nodo inicial** igual al resto de nodos que componen dicha lista. Estas listas, inicialmente, se componen de un puntero a nulo representando la **lista vacía**. Luego, a medida que vamos introduciendo elementos, ese puntero apuntará al primer elemento siempre. Pero podemos hacer que, inicialmente, la lista contenga un primer nodo cuyo contenido difiere del resto pues lo que tiene es información sobre la propia lista, como, por ejemplo, número de elementos, dirección del primer o último nodo, y cualquier otra información relevante que se crea oportuna guardar en este nodo de cabecera. Estas serían las **listas simples con cabeceras**.

Lista simple sin cabecera

Estas listas son las que hemos presentado en apartados anteriores: se componen de un puntero al primer elemento, si existe, si no apunta a nulo, y siguientes nodos, pero todos iguales con la misma estructura de información.

Lista circular

Hemos visto las listas como una secuencia lineal de información, con un principio (Puntero apuntando al primer elemento o nulo si está vacía la lista) y un final (Nodo apuntando a Nulo).

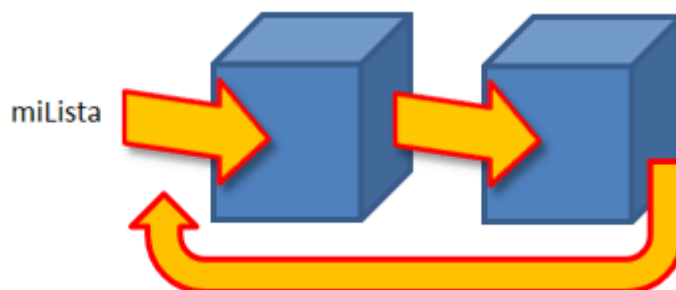
Presentamos ahora las **Listas Circulares**, en las que no hay un **inicio** ni un **final**. Partiendo de la lista vacía, a la que habría que darle un nombre identificativo, tendríamos:



Al meter un primer elemento en la misma tendríamos que el puntero al siguiente sería a él mismo:



Si metemos un segundo nodo de información, el resultado final sería como sigue:



Ahora, el puntero con el nombre de la lista no apunta al primero como antes. Ahora puede apuntar a cualquier elemento de la propia lista sin otro significado que tener localizada a la misma. Antes, este puntero con el nombre de la lista apuntaba siempre al primer elemento. Aquí ya no hay primero, ni segundo, ..., ni último.

Lista doblemente enlazada

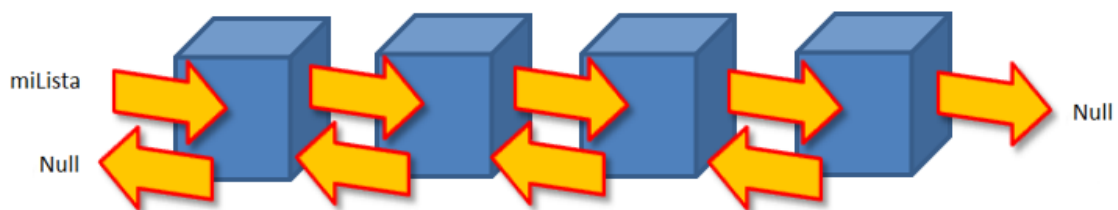
Hasta ahora, cuando queríamos recorrer una lista, debíamos comenzar por un nodo concreto, habitualmente el primero, e ir “saltando” de nodo en nodo hasta posicionarnos en el elemento deseado. Pero ese avance solamente se podía hacer en una dirección; no podíamos volver atrás, y si lo teníamos que hacer, se tenía que “resetear” el puntero **auxiliar** volviendo al principio y comenzando de nuevo el recorrido.

Con las **listas doblemente enlazadas**, podemos movernos en ambas direcciones, atrás y adelante. Tan solo debemos incluir otro puntero en cada nodo que apunte al nodo anterior, es decir, que contenga la dirección de memoria donde se encuentra en nodo anterior.

Dicha estructura podría ser algo similar a esto:

```
// Definición del nodo
typedef struct estructura
{
    int codigo;
    char nombre[10];
    struct estructura *anterior;
    struct estructura *siguiente;
} nodo;
```

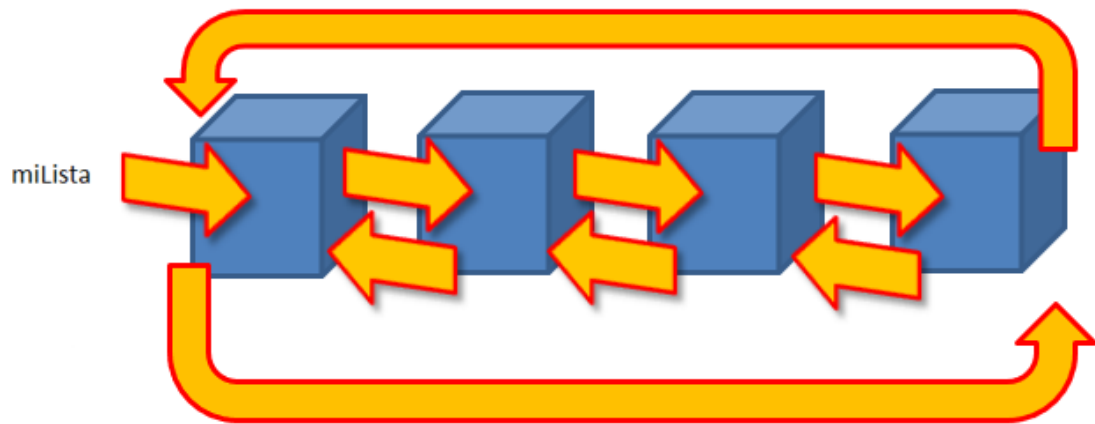
Ahora, tanto el anterior al primero, como el siguiente al último, apuntarán a Nulo. El resto tiene el mismo comportamiento. Podemos ver las listas doblemente enlazadas de la siguiente manera:



Obviamente, se dispone de las mismas operaciones que con las otras listas, pero su implementación se complica un poco, pero solo un poco, pues hay que trabajar con un puntero más.

Lista circular doblemente enlazada

Finalmente, podemos combinar estos dos últimos tipos de listas en las **listas circulares doblemente enlazadas**, en las que eliminamos los conceptos de primero y último y enlazamos todos y cada uno de los nodos con otros de la lista.



Referencias

N/A

17/08/2020