Programación II: Cuarta lección docencia *online*. Reserva dinámica de memoria. Listas enlazadas

Iván Álvarez Navia
Dpto. de Informática y Automática
Universidad de Salamanca
email: inavia@usal.es

19 de abril de 2020

Resumen

Este es el cuarto documento de los que publicaré en este periodo de suspensión de actividades lectivas. El objetivo de esta lección es presentar un nuevo mecanismo para relacionar los elementos que pertenecen una la colección. Hasta ahora, las tres lecciones estudiadas, el mecanismo que relaciona los elementos que pertenecen a la colección y que, por lo tanto, determina la forma de acceder a dichos elementos, es siempre el mismo: punteros más aritmética de punteros. En esta lección se presenta la lista simplemente enlazada, como el ejemplo más sencillo de lo que conocemos como *estructuras autoreferenciadas*.

1. Introducción

Seguimos trabajando con reserva dinámica de memoria. Comenzaremos con un pequeño recordatorio sobre lo ya explicado. A continuación, el objetivo de esta lección: estructuras autoreferenciadas, listas enlazadas.

Antes de iniciar el estudio sobre las funciones de reserva dinámica de memoria, tanto en la asignatura **Programación I**, como en los primeros temas de **Programación II**, en todos los ejercicios se han utilizado **variables estáticas**. Decimos que son estáticas en el sentido de que son variables declaradas en el código del programa, que se crean al comienzo del mismo con las características que se le han dado en tiempo de compilación, las cuales son inalterables. En este grupo se incluyen las variables de duración automática (variables locales y parámetros formales de las funciones) y las variables de duración estática (variables globales, y las locales precedidas del especificador de tipo de almacenamiento `static`). Las primeras, las automáticas, ocupan espacio en la pila del programa y la gestión de memoria es realizada de forma transparente por el mecanismo de invocación de funciones. Las segundas se almacenan en el segmento de datos, en la zona de variables globales (inicializadas y no inicializadas) y el espacio de memoria es reservado en el inicio de ejecución del programa y permanecen hasta la finalización de dicha ejecución.

A partir del Tema 3 de la asignatura **Programación II** y en las tres lecciones ya presentadas en este periodo de docencia *online*, estamos trabajando con las **variables dinámicas**. Estas variables, como ya se ha explicado, se diferencia de las estáticas por el momento en que se fija la cantidad de memoria a reservar. Mientras que en las estáticas es en el tiempo de compilación, en las dinámicas se fija en tiempo de ejecución, acomodando, por tanto, las características de las mismas a las necesidades concretas de esa ejecución del programa. La reserva de memoria se realiza en tiempo de ejecución en el *heap* o montículo del programa, y es responsabilidad del programador, tanto la reserva como la liberación. La asignación dinámica de memoria proporciona control directo sobre los requisitos de memoria del programa. El programa puede crear una variable dinámica en

cualquier momento de su ejecución, determinando exactamente en ese momento la cantidad de memoria necesaria. Igualmente el programa puede destruir una variable dinámica cuando ya no necesita usarla, dejando la memoria que ésta ocupada disponible de nuevo.

Hasta aquí lo estudiado. Hemos podido trabajar con vectores de tipos primitivos, con vectores de tipos de datos definidos por el programador, los registros. También hemos trabajado con vectores de punteros a registros y, por último, hemos estudiado como poder trabajar con variables dinámicas de tipo matriz bidimensional. En todos los casos estudiados el mecanismo para acceder a los valores es el uso de **punteros y la aritmética de punteros**. Precisamente este mecanismo de acceso es posible porque los elementos a los que se accede, o bien los punteros que se utilizan para acceder a dichos elementos, ocupan posiciones consecutivas de memoria y, por lo tanto, se puede utilizar la aritmética de punteros de forma adecuada para poder escribir las expresiones correctas para acceder a dichos elementos.

Ahora viene el problema. Es cierto que el tamaño se decide en tiempo de ejecución, lo que ya ha mejorado la eficiencia y la flexibilidad, pero una vez el tamaño es fijado, se realiza la reserva y, a continuación se utilizan estas variables dinámicas. La pregunta es ¿qué sucede si se desea cambiar el tamaño de la memoria reservada para una variable dinámica? Disponemos de un vector de 15 registros y, ahora, necesito añadir un nuevo registro. Pero, además, necesito insertar este registro en la posición sexta del vector de registros. Inmediatamente nos acordamos de una de las funciones de reserva dinámica de memoria estudiadas: `realloc()`. Pasos a dar:

1. Se invoca la función `realloc()` para incrementar en una celda más el vector reservado, para conseguir pasar de 15 a 16 celdas. Recordad que `realloc()` intentará reservar el espacio que está justo a continuación de la última celda del vector actual, pero si no hay espacio libre suficiente, lo buscará en otra zona de memoria. Si lo encuentra, se reserva y se copian los elementos del vector original a la nueva localización, liberando, a continuación, la memoria asignada antes de la invocación. Esto implica costes computacionales (tiempo) y, por lo tanto, penaliza la eficiencia.

2. Una vez se tienen reservadas la memoria para las 16 celdas, hay que abrir hueco para poder almacenar la información en la citada sexta posición. Para ello se copiarán, desde la posición 15 hasta la 6 una celda "hacia atrás" todos los registros. Así, el registro que estaba almacenado en la posición 6 aparecerá duplicado en la posición 7 y se podrá copiar la información en la citada celda 6 sin perder información. Más costes computacionales, menos eficiencia.

La eliminación de un elemento de este vector implica operaciones análogas, pero invirtiendo el orden de las operaciones a realizar. En cualquier caso, costes computacionales adicionales.

Se plantean, entonces, otros mecanismos para poder relacionar los elementos que pertenecen a la colección, con el objetivo de conseguir mayor flexibilidad a la hora de redimensionar, cambiar el tamaño, de las variables dinámicas. En vez de ocupar posiciones consecutivas de memoria, los elementos podrán estar dispersos por la memoria. Y, en vez de tener un vector de punteros apuntando a estos elementos, lo que sucederá es que cada elemento apuntara al siguiente, o a los siguientes elementos que forman parte de la colección. El objetivo: poder realizar una operación de inserción o eliminación de un elemento en la colección en cualquier posición sin necesidad de costes computacionales adicionales.

Surgen, así, diferentes tipos de estructuras, cada una con sus características, que serán muy útiles para resolver diferentes tipos de problemas. En general, a este tipo de estructuras las denominaremos **estructuras autoreferenciadas**, ya que cada elemento tiene referencia a uno o varios de los elementos que forman parte de la misma colección. En la Figura 1 se presentan algunos ejemplos de este tipo de estructuras.

Escoger una u otra dependerá del tipo de problema a resolver y de los algoritmos que se tengan que implementar y que puedan aprovechar las características propias de cada tipo. Por ejemplo, entre lista simplemente enlazada y lista doblemente enlazada la principal diferencia es que la primera únicamente permite un recorrido en un sentido, mientras que en la doblemente enlazada se puede

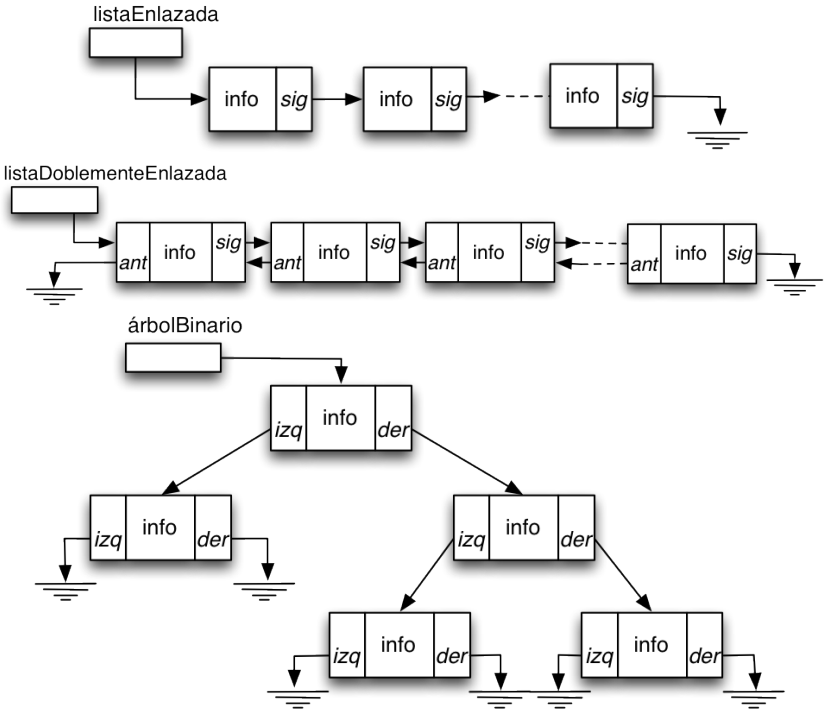

Figura 1: Agunos ejemplos de estructuras autoreferenciadas: lista simplemente enlazada, lista doblemente enlazada, árbol binario

hacer el recorrido en ambos sentidos. Otra variación son las listas circulares, cuando el último nodo "apunta" al primero, en en caso de las listas simplemente enlazadas, y último nodo "apunta" al primero y, a la vez, el primero al último, en las listas doblemente enlazadas.

En esta lección nos centraremos en las más sencillas: listas simplemente enlazadas. Comenzaremos por lo más básico, definiciones y reserva dinámica de memoria, para pasar, a continuación a realizar operaciones básicas: insertar en diferentes posiciones, recorrer, eliminar en diferentes posiciones.

2. Primeros pasos: conceptos básicos y reserva de memoria

El primer concepto que se introduce es el de **nodo**. Un nodo de una lista enlazada no es más que un registro que agrupa dos tipos de información:

- La información propiamente dicha, el elemento de la colección que se desea almacenar. Puede ser un tipo de dato primitivo o también una colección de campos, en cuyo caso se suelen agrupar, a su vez, en un registro.
- El enlace o referencia al siguiente elemento, al siguiente nodo, dentro de la colección. Se trata de un campo que será de tipo puntero a nodo, al mismo nodo en el que está definido.

Una vez establecido el concepto de nodo, podemos dar una definición más formal de lista enlazada:

> Secuencia de cero o más nodos, donde el orden (o posición) del nodo dentro de la secuencia viene determinado por el valor de un campo del mismo

Algunos aspectos a considerar:

- No es necesario saber de antemano la dimensión de la lista (no es necesario que definamos una dimensión máxima). Se trata de una estructura de datos dinámica de longitud desconocida:

se expande y se contrae según necesidades durante la ejecución del programa.

- Los nodos se crean (se reserva la memoria) y se destruyen (se libera la memoria) durante la ejecución del programa, y según se va necesitando.
- Los nodos **NO** van *físicamente* uno detrás del otro, **NO** ocupan posiciones consecutivas de memoria, como en una matriz, sino que van *lógicamente* uno detrás del otro. Un nodo apunta al siguiente.
- Se trata de una **estructura de acceso secuencial**. La única forma de acceder a un nodo es a través del inmediato anterior en la *secuencia lógica*. Este aspecto es determinante ya que implica que para poder acceder a un nodo concreto **hay que recorrer, de forma secuencial, todos los anteriores**, partiendo del primero hasta alcanzar el nodo deseado.

Pasamos a concretar la implementación. Lo primero que vamos a determinar es cómo se referencia una lista enlazada. Si cuando nos referimos a un vector o a una matriz, el nombre del vector/matriz es la dirección de memoria de la primera celda, cuando se trata con listas enlazadas no podemos hablar del "nombre" de la lista. Se utiliza una variable puntero, de tipo nodo, que almacenará la dirección de memoria del primer nodo de la lista enlazada. Este puntero suele recibir el nombre de **raíz** o **cabecera** de la lista enlazada.

El segundo concepto fundamental es el **nodo**. Un nodo es un registro que contiene una serie de campos:

- Un campo, o varios de igual o diferente tipo, que almacenan los datos o información

  Si son varios campos de información se suelen encapsular en una estructura registro que suele recibir el nombre de `info`, `datos`, `data`, ...

- Un campo de tipo puntero al mismo registro nodo (estructura anidada) y que almacena:
  - La dirección de memoria del nodo siguiente de la lista enlazada
  - El valor `NULL` si se trata del último nodo de la lista enlazada

  Este campo suele tener el nombre de `sig`, `siguiente`, `next`, ...

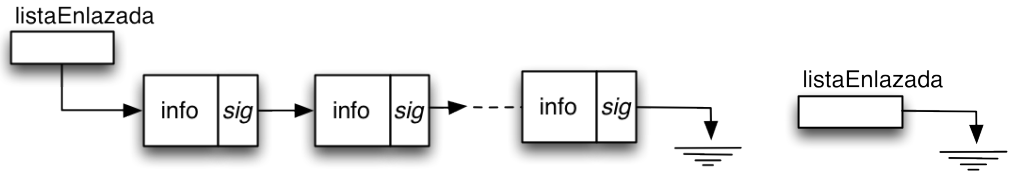

Figura 2: Lista simplemente enlazada: una con varios nodos y otra vacía

La Figura 2 representa la esencia de una lista simplemente enlazada. A la izquierda de la imagen, con varios nodos, y a la derecha una lista vacía. En este ejemplo el puntero **raíz** que referencia la lista enlazada es la variable, de tipo puntero a nodo, con el nombre `listaEnlazada`. El símbolo de "toma a tierra" del último nodo de la imagen de la izquierda, o del puntero `listaEnlazada`, en la imagen de la derecha, significa puntero `NULL`. Esta es la forma de representar que no hay enlace. En el caso del último nodo, la forma de finalizar la lista enlazada, no hay más nodos a continuación. En el caso de tener el puntero raíz, cabecera, `listaEnlazada` en nuestro ejemplo, a `NULL`, significa que está vacía, no hay nodos.

Con esta idea básica de lista nodo y lista enlazada, pasamos a concretar en lenguaje C el registro que representa el nodo:

```
typedef struct tipoNodo {
  tipoInfo info;
  struct tipoNodo * sig;
} tipoNodo;
```

donde el `tipoInfo` puede ser:

1. un tipo básico, por ejemplo, un entero

   ```
   typedef int tipoInfo;
   ```

2. o bien otro tipo estructurado, por ejemplo:

   ```
   typedef struct empleados {
     char apellidos[30];
     char nombre[15];
     int  matricula;
     char bufferRelleno[50];
   } tipoEmpleado;

   typedef tipoEmpleado tipoInfo;
   ```

esta aproximación facilita:

- la creación de nodos y la recuperación de la información almacenada: recordad que una de las operaciones perfectamente correctas en C, cuando se utilizan registros, es la asignación, en este caso de `tipoInfo`, y
- la reutilización de código como unidades, ya que se puede cambiar el tipo de información que se almacena en los nodos simplemente cambiando la definición del `typedef ... tipoInfo;`, manteniendo todo el código C correspondiente al manejo de los nodos.

Vamos a comenzar a ver las operaciones básicas con nodos y listas enlazadas. El siguiente paso será diseñar e implementar una biblioteca de funciones que permitan gestionar listas enlazadas. Recordad, disponer de una biblioteca de funciones que implementen las operaciones básicas de una lista enlazada, permite la reutilización de ese código. Pero eso será, insisto la segunda parte de esta lección. La primera es comprender bien los pasos básicos a realizar para resolver cada operación relevante:

- Creación de nodo.
- Crear una lista enlazada.
- Recorrer una lista enlazada.
- Insertar en una posición: inicio, final e intermedia.
- Eliminar un nodo y liberación de la memoria reservada.

3. Operaciones básicas

Comenzamos con las operaciones básicas que se pueden realizar con nodos y listas enlazadas, siempre por lo más sencillo e iremos avanzando a operaciones más complejas.

3.1. Declaraciones e iniciación de variables

Lo primero es declarar los tipos de datos que vamos a necesitar para poder definir variables. Comenzaremos con un ejemplo muy sencillo, donde el campo `tipoInfo` del nodo va a ser un número entero. Un cambio a otro tipo de dato, que puede ser todo lo complejo que se desee mediante el uso de registros, se puede realizar de forma sencilla sin más que cambiar el correspondiente `typedef`.

El código correspondiente a las declaraciones sería:

```
1 typedef int tipoInfo;
2
3 typedef struct tipoNodo{
4   tipoInfo info;
5   struct tipoNodo *sig;
6 }tipoNodo;
```

Se puede apreciar que los dos campos, dentro del registro `tipoNodo`, son los ya indicados: el campo `info`, con la información a almacenar, y el campo `sig`, el puntero, de tipo `tipoNodo`, que almacenará la dirección del siguiente nodo.

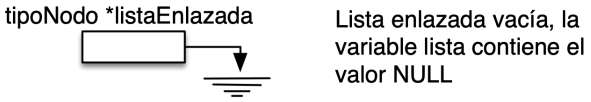

Figura 3: Lista simplemente enlazada: iniciar variables

La declaración de variables y su iniciación es la habitual cuando se manejan punteros:

```
1 tipoNodo *nuevo, *listaEnlazada;
2 tipoInfo valor;
3
4 listaEnlazada = nuevo = NULL;
```

como siempre, las variables puntero se inician a `NULL` para evitar errores de acceso posteriores (Figura 3). En el caso de la variable `listaEnlazada` será el puntero **raíz** de la lista enlazada que vamos a crear.

3.2. Creación de un nodo

En esta subsección estudiamos los pasos a dar para crear, reservar memoria dinámica, un nodo de la lista enlazada. Los pasos son sencillos:

- Se va a reservar memoria dinámica para un nodo.
- Se debe asegurar que la dirección de memoria devuelta por la invocación a la función de reserva dinámica de memoria es distinto de `NULL`, es decir, que ha tenido éxito. De esta manera se evitan errores de acceso que provocarán un error de ejecución.
- Se debe cargar la información en el campo `info`, si existe o está disponible en ese instante.
- Se debe poner a `NULL` el campo `sig` del nodo creado para asegurarnos, nuevamente, errores de acceso posteriores.

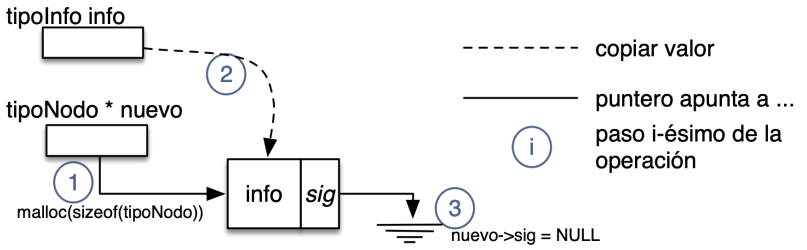

Figura 4: Lista simplemente enlazada: reserva de memoria, creación de un nodo

El código en lenguaje C sigue estos pasos, tal y como se pueden ver en la Figura 4 sería:

```
nuevo = malloc(sizeof(tipoNodo));
if (nuevo != NULL){
  nuevo->info = valorInfo;
  nuevo->sig = NULL;
}
```

En este ejemplo, se carga en el campo `info` del nodo la información de la variable `valorInfo`, dando por supuesto que está disponible. Si no está disponible se deja sin dar valor. Sin embargo, el campo `sig` si es conveniente ponerlo a `NULL` y así evitamos que un olvido posterior nos dé problemas.

Este código solo crea el nodo, reserva la memoria, comprueba si tiene éxito y, si tiene éxito, carga, al menos, el campo `sig`. Pero esto no crea la lista enlazada, solo reserva, crea, el nodo. Ahora vamos convertir este nodo en un nodo de una lista enlazada. Para ello hay que utilizar la variable que hemos escogido como **raíz** de la lista enlazada, en nuestro ejemplo, la variable puntero `listaEnlazada`.

```
nuevo = malloc(sizeof(tipoNodo));
if (nuevo != NULL){
  nuevo->info = valorInfo;
  nuevo->sig = NULL;
  if (listaEnlazada == NULL) {
    listaEnlazada = nuevo;
  }
}
```

en este código hemos añadido dos líneas: la primera comprueba que la variable `listaEnlazada` esté a `NULL`, lo que significa que la lista enlazada está vacía y estamos insertando el primer nodo de la lista enlazada, que es la asignación de la segunda línea añadida. Si no se cumple esa condición, `listaEnlazada == NULL`, no se puede realizar la inserción de una manera tan sencilla, se tiene que realizar alguna operación más. Siguiente subsección.

3.3. Creación de una lista enlazada insertando por el final

En esta subsección vamos a crear una lista enlazada, con nueve nodos, añadiendo el nuevo nodo por el final de la lista en construcción.

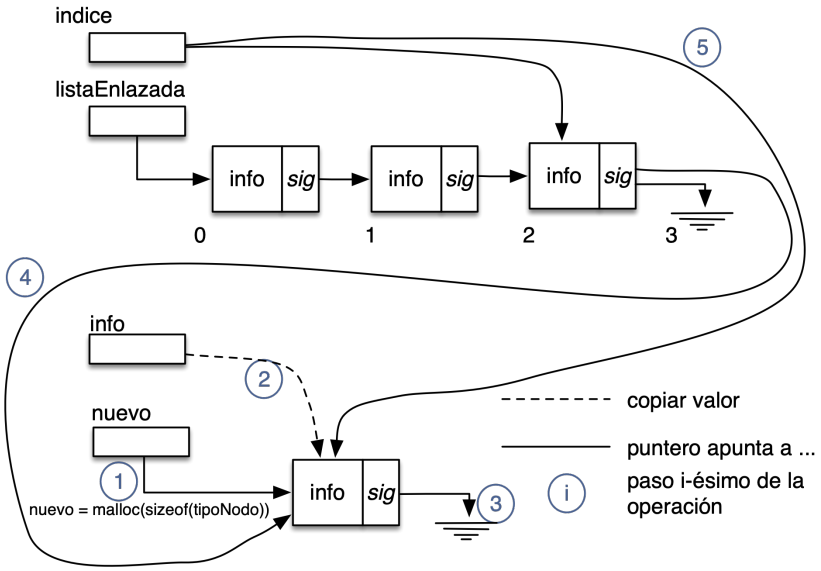

Figura 5: Lista simplemente enlazada: creación añadiendo por el final

Para evitar recorridos innecesarios, y que todavía no hemos aprendido a hacer, vamos a utilizar una variable auxiliar, de tipo puntero `tipoNodo` que apuntará siempre al último nodo añadido. De esta manera, tenemos acceso directo, a través de dicha variable auxiliar, al nodo cuyo campo `sig` hay que cambiar para poder añadir el nodo nuevo. Los pasos están numerados en la Figura 5. Es importante resaltar que la variable `indice` está apuntando siempre al último nodo añadido a la lista enlazada, es la operación 5. Obsérvese como, antes de añadir el nodo, cuando el último es el número 2 (están numerados comenzando por 0), `indice` está apuntando a ese nodo 2. Una vez se ha creado el nuevo nodo, se ha añadido, es la opearción 4, se cambia el campo `sig` del nodo 2 de `NULL` a `nuevo`, se hace que la variable `indice`, que apuntaba al nodo 2, apunte ahora al nodo que va a ocupar la posición 3, y esa operación es la numerada con el 5.

El código fuente que implementa estas operaciones se encierra en un bucle `for` que es el encargado de repetir la misma operación 9 veces.

```
1  listaEnlazada = indice = NULL;
2  for(i = 0; i < 9; i++){
3    nuevo = malloc(sizeof(tipoNodo));
4    if (nuevo != NULL){
5      nuevo->info = 2*i + 1;
6      nuevo->sig = NULL;
7      if (listaEnlazada == NULL)
8        listaEnlazada = nuevo;
9      else
10       indice->sig = nuevo;
11     indice = nuevo;
12   }
13 }
```

Por simplificar, en campo `info` se carga con el valor de la variable contadora del bucle. El `if` de la línea 6 es la que se encarga de controlar si estamos insertando el primer nodo de la lista enlazada (en ese caso `listaEnalazada` es igual `NULL`), o si ya hay nodos en la lista enlazada (en ese caso `listaEnlazada` es distinta de `NULL`). Si estamos en el primer caso, pues el nuevo nodo, `nuevo`, va a ser el primer y único nodo de la lista enlazada, línea 7. Si estamos en el segundo caso, el nuevo nodo se añade por el final, haciendo uso del puntero `indice`, que apunta a dicho último nodo, línea 9. En ambos casos, en la línea 10 se hace que la variable `indice` apunte a este nuevo nodo que ha

pasado a ser el último de la lista enlazada, dejando así todo preparado para la siguiente iteración del bucle.

Una vez ya tenemos nodos y una lista con nueve nodos, vamos a la siguiente operación: recorridos.

3.4. Recorrido de una lista enlazada

Esta operación puede parecer sencilla y, de hecho, lo es. Además es fundamental porque será, normalmente, la forma de identificar la posición donde insertar o eliminar un nodo. El elemento clave, se trata de una estructura de **acceso secuencial**. La única forma de acceder a un nodo es a través del campo `sig` del nodo anterior. Esto implica que no existe el acceso directo, el que se utiliza en los vectores y matrices utilizando los índices. Para poder acceder a un nodo se deben recorrer, secuencialmente, todos los anteriores partiendo del nodo raíz.

En el ejemplo que estamos construyendo vamos a recorrer la lista enlazada mostrando por pantalla el contenido del campo `info` de cada nodo.

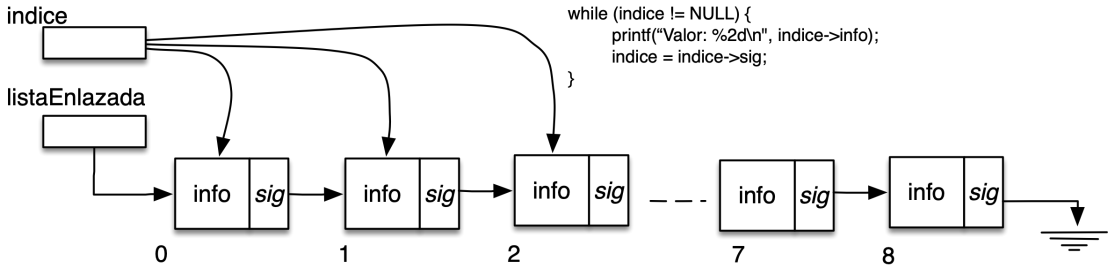

Figura 6: Lista simplemente enlazada: recorrer una lista enlazada

En la Figura 6 se observa la totalidad del código. Primero con la variable que se va a realizar el recorrido, `indice` en nuestro ejemplo, se debe apuntar al primer nodo de la lista enlazada, referenciado por la variable `listaEnlazada` y, a continuación, se debe entrar en un bucle, con condición de entrada, en nuestro caso un `while`, que evalúa el puntero `indice`, de manera que se garantice que, antes de entrar a ejecutar el código del cuerpo del bucle, dicha variable es distinta de `NULL`. Es la forma de garantizar que estamos apuntando a un nodo existente. Cuando `indice` apunta al último nodo, al realizar el avance dicha variable `indice` pasará a almacenar `NULL`, lo que implica el final de recorrido ya que evaluación de la condición de entrada del bucle, resulta falsa, con lo que termina el bucle y, por lo tanto, el recorrido.

Cada flecha que sale de la variable `indice` y apunta a un nodo es un avance. Este avance se produce al ejecutar la línea `indice = indice->sig;`, y significa que en la variable `indice` se almacena la dirección del nodo siguiente, y dicha dirección está almacenada en el campo `sig` del nodo referenciado por la citada `indice`. El código completo sería:

```
1 printf("\nPrimer recorrido:\n");
2 indice = listaEnlazada;
3 i = 0;
4 while(indice != NULL){
5   printf("Posición: %2d -> elemento: %2d\n", i++, indice->info);
6   indice = indice->sig;
7 }
```

La variable `i` únicamente sirve para poder mostrar un ordinal que indica la posición del elemento mostrado en la secuencia. Lo importante es la iniciación de `indice`, en la línea 2, y el avance, en la línea 6.

Una vez ya sabemos crear y recorrer, hay que aprender a liberar la memoria, toda. Así podremos reescribir mejor nuestro código de crear una lista enlazada de nueve nodos, gestionando la situación en la que se produce un error en la reserva de memoria de un nodo.

3.5. Liberación de todos los nodos de una lista enlazada

Esta tarea es muy sencilla, porque es análoga a la vista en la subsección 3.4. Se trata de recorrer la lista y, en cada avance, hacer las operaciones en el orden correcto para poder liberar la memoria sin perder el acceso al nodo cuando se produce el avance. Para ello se utilizará un puntero auxiliar, `aBorrar`, que guarda la referencia al nodo a liberar. Si no se hace de esta manera se corre el riesgo de acceder al campo `sig` de un nodo ya liberado y eso sería un acceso de memoria ilegal (aunque no produzca un error de ejecución o, al menos, no siempre).

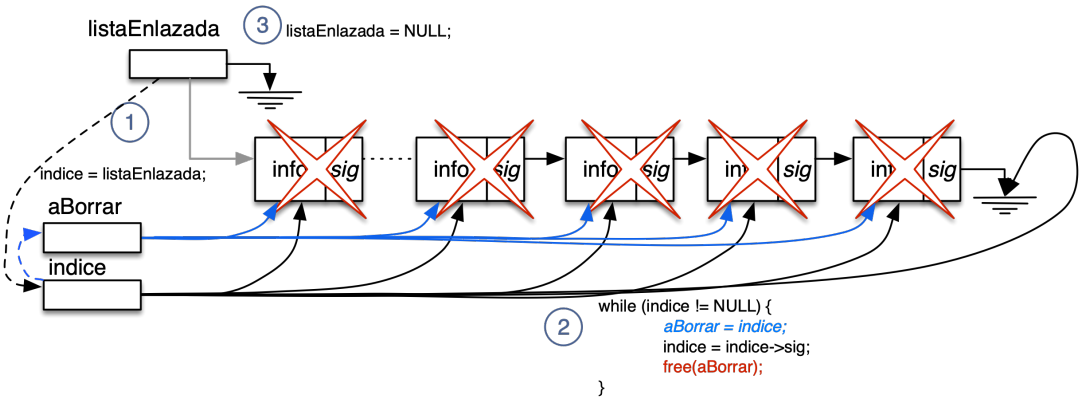

Figura 7: Lista simplemente enlazada: eliminar todos los nodos

El la Figura 7 se puede apreciar las operaciones básicas. La variable `indice` apunta, inicialmente, al nodo raíz. A partir de ahí, bucle de recorrido. En cada iteración del bucle, y **SIEMPRE** antes del avance, se guarda en la variable auxiliar `aBorrar` (líneas azules) la dirección del nodo a liberar, referenciado por `indice`, y entonces ya puede avanzar (líneas en negro). Una vez avanza `indice` se procede a liberar el nodo referenciado por la variable auxiliar `aBorrar`.

```
1 indice = listaEnlazada;
2 aBorrar = NULL;
3 while (indice != NULL){
4   aBorrar = indice;
5   indice = indice->sig;
6   free(aBorrar);
7 }
8 listaEnlazada = NULL;
```

El último paso es, obviamente, poner la variable `listaEnlazada` a `NULL` para indicar que la lista enlazada está vacía (paso 3 en la Figura 7).

Como el resto de operaciones, esta tarea se puede realizar de diferentes maneras. A continuación se escribe el código que realiza la misma tarea, libera todos los nodos, pero de una manera diferente. Lo que se hace es guardar en la variable auxiliar, `aBorrar`, la referencia del nodo a liberar y, a continuación, se avanza con la misma variable puntero `listaEnlazada`. El efecto es ir liberando siempre el nodo raiz:

```
1  while (listaEnlazada != NULL){
2    aBorrar = listaEnlazada;
3    listaEnlazada = listaEnlazada->sig;
4    free(aBorrar);
5  }
```

La gran ventaja es que `listaEnlazada` ya queda a `NULL` una vez ha finalizado el bucle, es justo la condición de finalización del mismo.

Se recomienda al estudiante que realice un dibujo de esta alternativa y que comprenda el orden correcto de las operaciones.

Una vez se sabe liberar, se puede modificar el código de creación, controlando el fallo en la reserva de un nuevo nodo y, en caso de que falle, entonces se liberen todos los nodos creados hasta ese instante:

```
1  listaEnlazada = indice = NULL;
2  for(i = 0; i < 9; i++){
3    nuevo = malloc(sizeof(tipoNodo));
4    if (nuevo != NULL){
5      nuevo->info = 2*i + 1;
6      nuevo->sig = NULL;
7      if (listaEnlazada == NULL)
8        listaEnlazada = nuevo;
9      else
10       indice->sig = nuevo;
11     indice = nuevo;
12   }
13   else {
14     aBorrar = listaEnlazada;
15     while (listaEnlazada != NULL){
16       aBorrar = listaEnlazada;
17       listaEnlazada = listaEnlazada->sig;
18       free(aBorrar);
19     }
20     break;  //Ha fallado la reserva, debe finalizar el for
21   }
22 }
```

Cuando termina de ejecutarse este código, o bien tenemos una lista enlazada con nueve nodos, referenciada por `listaEnlazada`, o bien tenemos una lista vacía, `listaEnlazada = NULL`, en el caso de que se haya producido un error en alguna de las reservas. El bloque del `else`, líneas de 13 a 21, es la parte de liberación de todos los nodos que han sido creados hasta ese instante en el que ha fallado la reserva de memoria, `if` de la línea 4. El `break;` de la línea 20 hace que finalice el bucle `for`.

A partir de aquí, vamos a ver algunos ejemplos de algunas operaciones, también típicas, pero que se propone al estudiante que comience a trabajar por su parte, como ejercicio individual. La idea, primero, dibujar, insisto, **DIBUJAR**, la lista enlazada, dibujar un esbozo de la operación a realizar, las variables involucradas, los enlaces, las flechas, ... Cuando ya se tiene una idea clara de las variables involucradas, las operaciones a realizar y el orden correcto en que se deben realizar

dichas operaciones, entonces ya se puede pasar a escribir código en C.

3.6. Añadir un nodo nuevo al principio de una lista enlazada

En este caso se trata de insertar un nodo en una lista enlazada de manera que, dicho nodo, pase a ser la nueva raíz de la lista enlazada. Las operaciones básicas se pueden ver en la Figura 8.

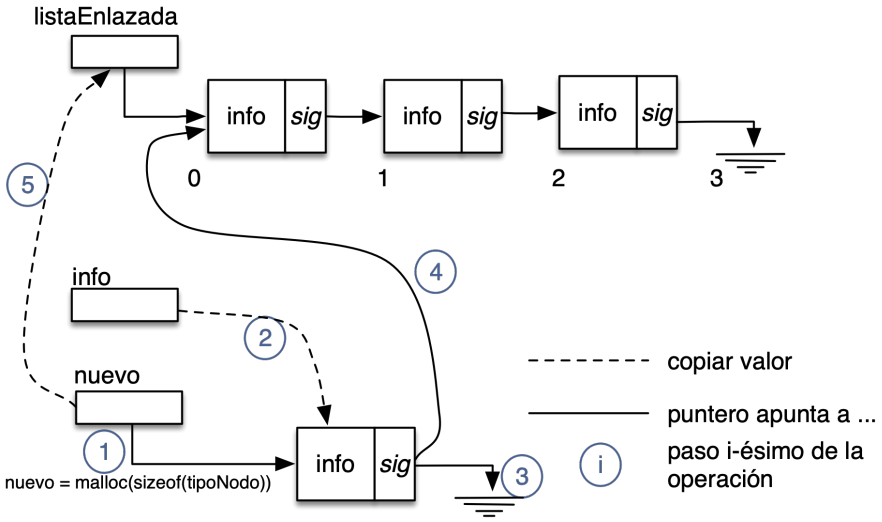

Figura 8: Lista simplemente enlazada: insertar nuevo nodo por la raíz

Los pasos a dar comienzan por la creación del nodo, cargar el valor a almacenar en el campo `info`, y enlazar. El enlace es sencillo. Primero se engancha en nodo recién creado con la raíz actual, eso es la operación 4, se almacena en el campo `sig` del nodo referenciado por el puntero `nuevo`, la dirección del nodo raíz que, como ya sabemos, está almacenada en el puntero `listaEnlazada`. Ahora hay que convertir el nuevo nodo en el nodo raíz. Eso significa que hay que almacenar en la variable `listaEnlazada` la dirección del nuevo nodo, `nuevo`, operación 5.

En código C:

```
1 nuevo = malloc(sizeof(tipoNodo));
2 if (nuevo != NULL){
3   nuevo->info = 1000;
4   nuevo->sig = listaEnlazada;
5   listaEnlazada = nuevo;
6 }
```

Es importante resaltar dos detalles:

- En la Figura 8 se ha mantenido el campo `sig` del nuevo nodo a `NULL` en la creación del mismo, porque puede ser implementado por una función aparte, como haremos más adelante. En el código C presentado, sin embargo, no aparece esa asignación. Se asigna directamente ya la dirección de la raíz: `nuevo->sig = lista;`.
- En este tipo de operaciones hay que comprobar siempre qué puede ocurrir en diferentes situaciones. En esta operación en concreto, insertar por la raíz, solo se ve afectada la variable `listaEnlazada`. La pregunta que debe contestar el programador es ¿qué pasa si `listaEnlazada` es `NULL`?. Se deja al estudiante que razone si el código presentado funciona correctamente o no.

3.7. Añadir un nodo nuevo al final de una lista enlazada

Es otra operación particular de inserción: añadir el nuevo nodo al final de la lista enlazada y que pase a ser en nuevo último nodo de la lista enlazada.

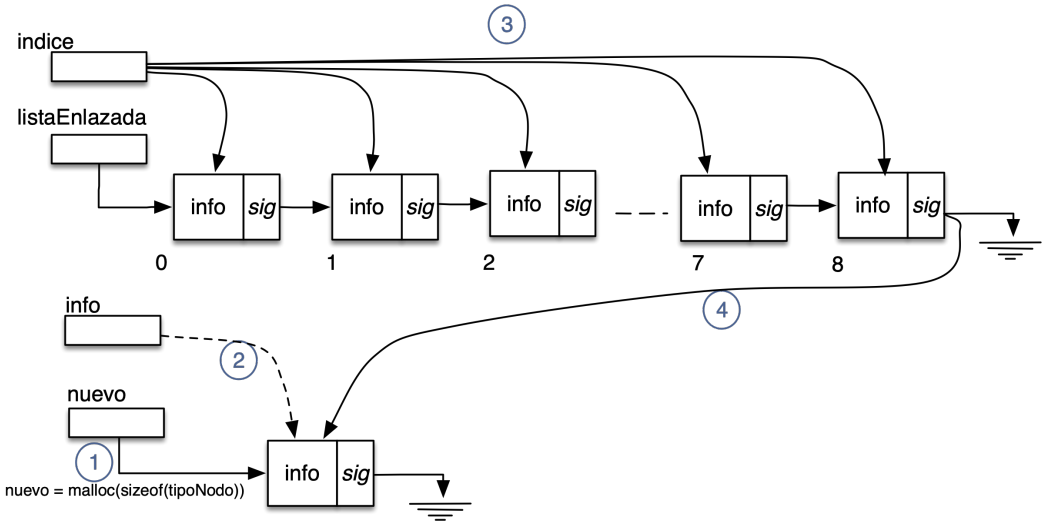

Figura 9: Lista simplemente enlazada: añadir un nuevo nodo al final de la lista enlazada

La operación es relativamente sencilla: se debe localizar el último nodo, para ello se utiliza un recorrido secuencial, análogo al presentado en la subsección 3.4, pero no exactamente igual, porque lo que buscamos es que la variable `indice` quede apuntando al último nodo, por eso en el código veremos que en la condición del bucle `while` del recorrido se utiliza la comparación `indice->sig != NULL`. Esta es la operación 3.

Una vez localizado el último nodo, se queda referenciado por la variable `indice`, entonces hay que hacer que este nodo apunte al nuevo, se cambia el campo `sig` del nodo referenciado por `indice`, que está a `NULL`, por la dirección del nuevo nodo, `nuevo`.

También es importante controlar que la lista no esté vacía, porque si lo está, no hay recorrido, el nuevo nodo pasa a ser el único nodo de la lista enlazada.

En código C:

```
nuevo = malloc(sizeof(tipoNodo));
if (nuevo != NULL){
  nuevo->info = 25;
  nuevo->sig = NULL;
  if (listaEnlazada == NULL){
    listaEnlazada = nuevo;
  }
  else {
    indice = listaEnlazada;
    while(indice->sig !=NULL){
      indice = indice->sig;
    }
    indice->sig = nuevo;
  }
}
```

Se comprueba si la lista enlazada está vacía, y si lo está pues `listaEnlazada = nuevo;`. Si no está

vacía se inicia el recorrido, iniciando la variable puntero que se utilizará realizar dicho recorrido, `indice`. Se debe llegar con esta variable hasta el último nodo. La característica del último nodo es que su campo `sig` es `NULL`. Por esta razón tenemos esa condición en el bucle `while`. Una vez se ha finalizado ese bucle, `indice` apunta al último nodo. Solo queda hacer que dicho nodo apunte al nuevo, el campo `sig` del último nodo, que es `NULL`, como ya hemos dicho, ahora almacenará la dirección del nuevo nodo, `nuevo`.

3.8. Insertar un nodo nuevo en una posición arbitraria de una lista enlazada

Esta tarea ya es más complicada. Todo depende del criterio que se utilice para decidir la posición. Para facilitar un poco las cosas, vamos a plantear un ejercicio muy típico: crear una lista de números enteros ordenados de forma ascendente, es decir, que $a_{i-1} \leq a_i \ \forall \ i = 1, \ldots, n-1$, siendo los a_i los valores almacenados en el campo `info` de los nodos, y estando las posiciones de esos nodos en la secuencia dentro del rango de 0 a $n-1$.

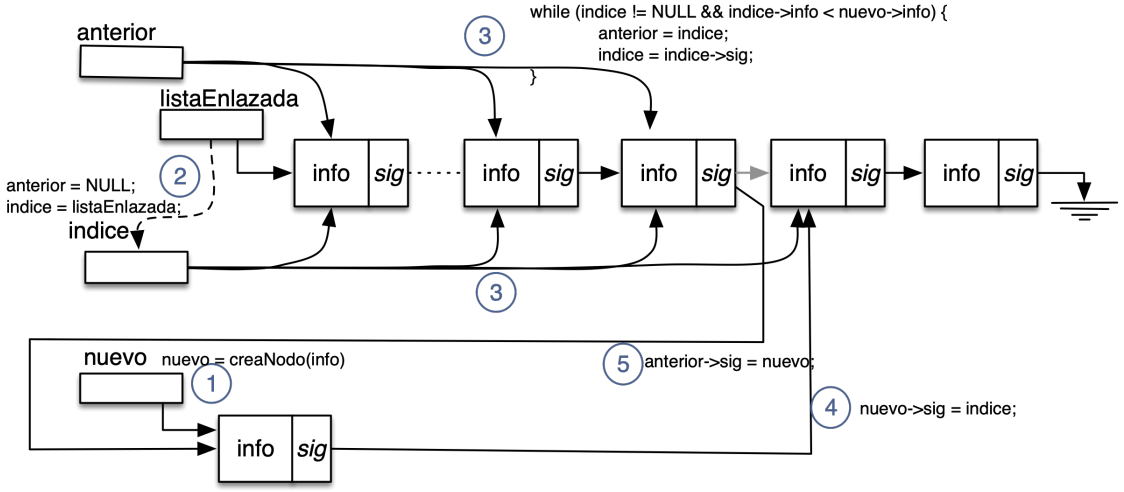

Figura 10: Lista simplemente enlazada: insertar nuevo nodo en posición ordenada

Lo primero que se hace es localizar la posición que le corresponde al nuevo valor. Para ello está la operación 3, un recorrido secuencial que se realiza mientras que sea cierto que `indice->info < nuevo->info`. Pero, además, y mediante un operador `&&` nos aseguramos que el puntero `indice` es distinto de `NULL`. Hay que pensar que el valor a insertar puede ser mayor que el almacenado en el último nodo, entonces, si no se realiza esta comprobación, recuérdese, **en cortocircuito**, se podría dar la situación `indice = NULL`, con el consiguiente error de ejecución (acceso puntero `NULL`) en la expresión `indice->info`. El segundo problema es que cuando se para el bucle, con esta condición, supone que `indice` apunta al valor que es MAYOR que el valor a insertar, es decir, hay que insertar el nuevo nodo ANTES del nodo apuntado por `indice`. Por esa razón se utilizan dos punteros: el ya explicado `indice` y otro, auxiliar, que siempre apunta al nodo anterior, `anterior`. Así se puede acceder al campo `sig` del nodo anterior, e insertar `nuevo` justo a continuación: operaciones 4 y 5. Aquí puede ser importante el orden. En este caso concreto, con dos variables puntero auxiliares, `indice` y `anterior`, no. La operación 4 engancha el nodo nuevo con la lista a partir de `indice` y la operación 5 engancha los nodos anteriores, hasta `anterior`, con nuevo. Así se mantiene intacta la secuencia, con el nuevo nodo insertado en la posición que le corresponde.

Una versión del algoritmo presentada en la Figura 10, y digo versión, porque hay variaciones relacionadas con comprobaciones de situaciones que se pueden presentar, como que el valor a insertar deba estar en la primera posición, o al final:

```
1  nuevo = malloc(sizeof(tipoNodo));
2  if (nuevo != NULL){
3    nuevo->info = random()%100;
4    nuevo->sig = NULL;
5    if (listaEnlazada == NULL || listaEnlazada->info > nuevo->info){
6      //lista vacía o el valor a insertar debe ser el primero
7      nuevo->sig = listaEnlazada;
8      listaEnlazada = nuevo;
9    }
10   else{
11     anterior = NULL;
12     indice = listaEnlazada;
13     while(indice != NULL && indice->info < nuevo->info){
14       anterior = indice;
15       indice = indice->sig;
16     }
17     nuevo->sig = indice;
18     anterior->sig = nuevo;
19   }
20 }
```

En la versión presentada: se garantiza que si la lista está vacía, el valor pasa a constituir el único nodo de la lista enlazada. En esa misma comprobación también se realiza la inserción si el valor a insertar es menor que el almacenado en el primer nodo (línea 5). Para el resto de casos: recorrido (líneas 13 a 16) con dos variables puntero. Si el bucle finaliza es, o bien porque se ha llegado al final de la lista enlazada, es decir, el valor a insertar es mayor que el almacenado en el último nodo, o bien se localiza la posición donde insertar, un nodo de la lista enlazada tiene un valor que es mayor que el valor a insertar. Ese nodo quedará apuntado por el puntero `indice`. Entonces, hay que insertar antes. Para ello hay que cambiar el campo `sig` del nodo anterior, apuntado por la variable `anterior`. En ambos casos el código es el mismo (líneas 17 y 18). Se deja al estudiante que reflexione y comprenda que ese código funciona correctamente en los dos casos explicados.

En caso de fallar la reserva dinámica de memoria para el nodo nuevo a insertar, comprobación en la línea 2, no se hace nada (obsérvese que el `if` de la línea 2 no tiene `else`). Aquí es donde el programador puede plantear otras opciones diferentes, como liberar todos los nodos insertados hasta ese instante. Las acciones a realizar en ese caso dependerán del contexto donde se ejecuta este código.

3.9. Eliminar el primer nodo de una lista enlazada

En este caso se trata de eliminar el primer nodo de una lista enlazada, el nodo raíz. Las operaciones básicas se pueden ver en la Figura 11.

En esencia, siempre que se tiene que eliminar un nodo de una lista enlazada, la operación básica a realizar es "puentear" el nodo a eliminar. Se trata de conseguir que la secuencia de nodos continúe intacta pero saltando el nodo que se debe eliminar y liberar. En el caso de eliminar el nodo raíz de una lista enlazada implica conseguir que el nuevo nodo raíz será el que, antes de realizar la operación, era el segundo nodo de la lista enlazada. Para ello se debe conseguir que la variable puntero que referencia dicho nodo raíz, `listaEnlazada`, pase a apuntar a ese segundo nodo de la lista enlazada. En la Figura 11 se observan dos operaciones a realizar. Se comienza referenciando, con una variable auxiliar, el puntero `aBorrar`, al nodo raíz. A continuación, se "puentea" ese nodo raíz, es la operación 2. En este momento, el que era nodo raíz ha desaparecido de la lista enlazada. Pero todavía queda un paso más, liberar la memoria, paso 3.

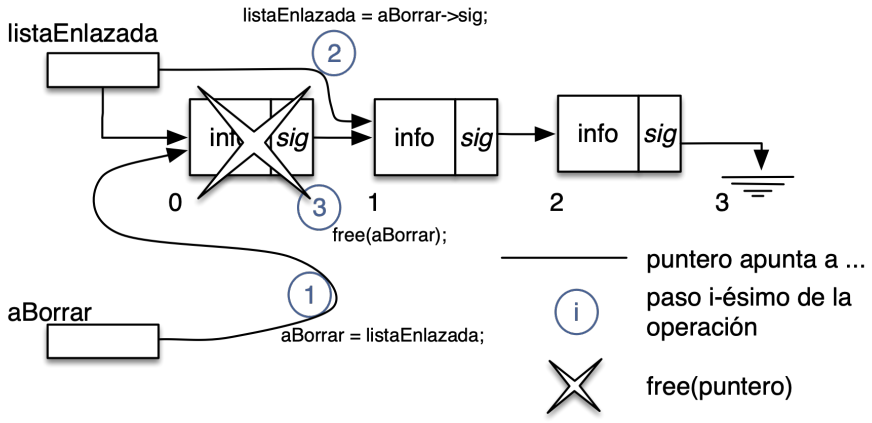

Figura 11: Lista simplemente enlazada: eliminar el nodo raíz de una lista enlazada

El código C sería:

```
1 aBorrar = listaEnlazada;
2 listaEnlazada = aBorrar->sig;
3 free(aBorrar);
```

En este código no se han tomado precauciones sobre si la lista está vacía o no, algo que sería recomendable. El código resultante es muy sencillo y refleja, de forma fiel, lo que se ha explicado en la Figura 11 por lo que se deja al estudiante su comprensión final. Por supuesto, lo único relevante es el orden en que se realizan las operaciones: ¿qué pasaría si invertimos el orden y ejecutamos primero la línea 3 y luego la línea 2?¿funcionaría? ¿daría error de ejecución?, si no da error de ejecución ¿es, entonces, correcto? Insisto, se deja al estudiante que conteste a estas preguntas.

3.10. Eliminar el último nodo de una lista enlazada

Es otra operación particular de eliminación: eliminar el último nodo de la lista enlazada y que el nodo anterior al eliminado pase a ser el nuevo último nodo de la lista enlazada.

La operación es relativamente sencilla: se debe localizar el último nodo, para ello se utiliza un recorrido secuencial, análogo al presentado en las subsecciones 3.4 y 3.7. Pero, de nuevo, con alguna variación, ya que lo que se necesita en este caso es que un puntero quede apuntando al último nodo, hay que liberarlo, y otro al nodo anterior, el penúltimo, para poder actualizar correctamente su campo `sig` para que pase a ser el nuevo último nodo de la lista enlazada.

Si nos fijamos en la Figura 12, las operaciones 1 y 2 son las involucradas en el recorrido secuencial de la lista para localizar los dos nodos: el último, que quedará referenciado por el puntero `aBorrar`, y el penúltimo, que quedará referenciado por el puntero `anterior`. La siguiente operación, la 3 es convertir el nodo anterior en el nuevo último nodo de la lista enlazada. Lo único que hay que hacer es poner a `NULL` su campo sig. Hecho esto, ya se puede liberar la memoria reservada para el nodo referenciado por `aBorrar`, el nodo que se desea eliminar.

También es importante controlar que la lista no esté vacía, porque si lo está, no hay recorrido, ni eliminación. Nuevamente, sería conveniente realizar las comprobaciones adecuadas para evitar errores de acceso a memoria (nodos inexistentes).

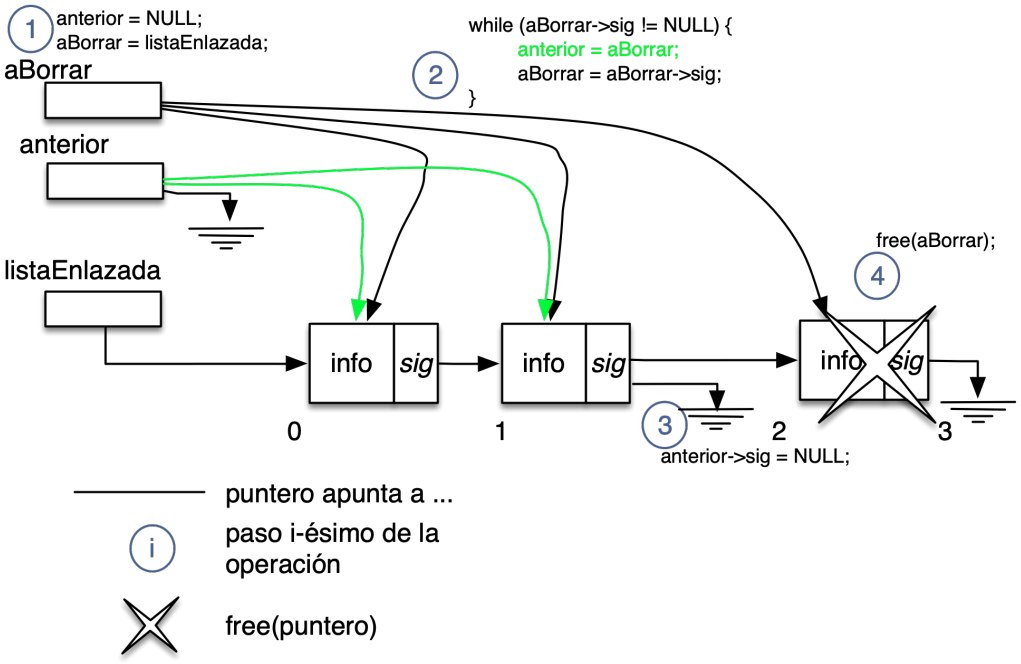

Figura 12: Lista simplemente enlazada: eliminar el último nodo de la lista enlazada

En código C:

```
1  if (listaEnlazada != NULL){
2    aBorrar = listaEnlazada;
3    anterior = NULL;
4    while(aBorrar->sig !=NULL){
5      anterior = aBorrar;
6      aBorrar = aBorrar->sig;
7    }
8    if (anterior == NULL){
9      listaEnlazada = NULL;
10   }
11   else {
12     anterior->sig = NULL;
13   }
14   free(aBorrar);
15 }
```

A diferencia de otros ejemplos, en este código se han tenido tres posibles situaciones:

- La lista está vacía o tiene nodos. Línea 1 del código. Si está vacía, obviamente, no se hace nada.
- La lista tiene un único nodo. En este caso el bucle de la línea 4 no itera, ya en la primera evaluación de la condición el resultado es falso (un único nodo, su campo `sig` está a `NULL`), el puntero `anterior` sigue con el valor de iniciación (línea 3), así que lo único que hay que hacer es eliminar ese único nodo, quedando la lista vacía (línea 9).
- La lista tiene dos o más nodos. El bucle itera, las variables `anterior` y `aBorrar` van avanzando por la lista enlazada, apuntando siempre a dos nodos consecutivos, el primero al nodo anterior al que apunta el segundo. El bucle termina cuando `aBorrar` apunta al último nodo (su campo `sig` es `NULL`). Entonces se debe convertir el nodo anterior, referenciado por `anterior`, en el

último nodo de la lista enlazada (línea 12).

Tanto en el segundo caso, lista enlazada con un único nodo, como en el tercer caso, lista con dos o más nodos, se debe liberar la memoria reservada por el nodo referenciado por `aBorrar` (línea 14).

3.11. Eliminar un nodo situado en una posición arbitraria de una lista enlazada

Análoga a la operación descrita en la subsección 3.8, pero para eliminar, en vez de para insertar. Nuevamente todo depende del criterio que se utilice para decidir la posición. Para facilitar un poco las cosas, vamos a plantear un ejercicio muy típico: localizar el primer nodo que contiene un valor concreto (variable `valor` en la Figura 13 y, si se encuentra, eliminarlo.

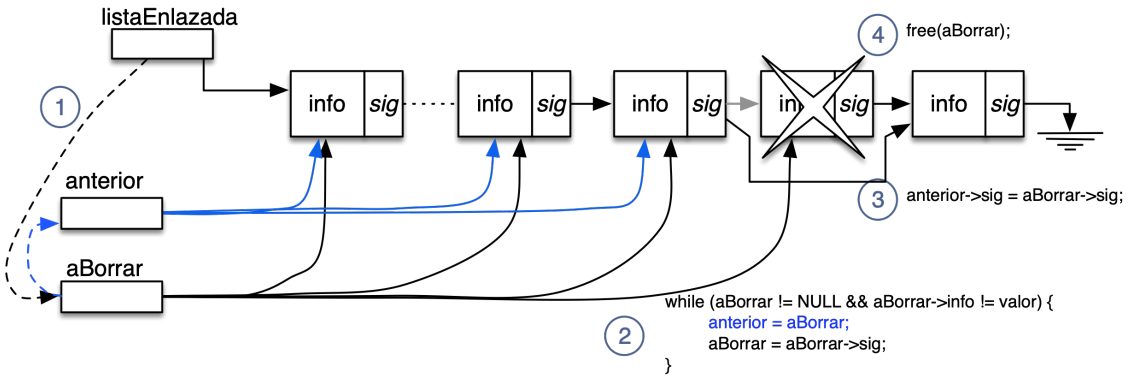

Figura 13: Lista simplemente enlazada: eliminar un nodo de una posición arbitraria, campo `info` es igual a `valor`

Lo primero que se hace es localizar la posición del nodo cuyo campo `info` es igual `valor` deseado. Esta tarea se resuelve en dos operaciones, la 1 y la 2. La primera es la iniciación de las variables que se utilizan en el recorrido, `anterior = NULL;` y `aBorrar = listaEnlazada`. La idea es que la segunda, `aBorrar` quede apuntando al nodo cuyo campo `info` es igual a `valor` y, por lo tanto, el nodo que hay que eliminar. La variable `anterior` apuntará al nodo anterior al apuntado por `aBorrar`. El bucle finaliza por una de estas dos razones: `aBorrar` es `NULL`, lo que significa que se ha recorrido toda la lista y no se ha encontrado un nodo que cumpla la condición especificada y, por lo tanto, no hay nodo que eliminar. Esta condición es importante porque, por un lado evita que nos salgamos de la lista enlazada y, además, al ser evaluada en cortocircuito, se evita un acceso ilegal en segunda condición del `&&`, `aBorrar->info != valor`.

Finalizado el bucle, se comprueba la razón de finalización, y si `aBorrar` sí apunta a un nodo, entonces hay que hacer la habitual operación, "puentear" el nodo apuntado por `aBorrar` y liberarlo. Eso se hace en las operaciones 3, "puentear" a través de `anterior`, y liberar en la operación 4.

Una versión del algoritmo presentada en la Figura 13, y digo versión, porque hay variaciones relacionadas con comprobaciones de situaciones que se pueden presentar, como que el valor a eliminar está en el nodo raíz, o al final:

```
1  if (listaEnlazada != NULL){
2    anterior = NULL;
3    aBorrar = listaEnlazada;
4    while(aBorrar != NULL && aBorrar->info != valor){
5      anterior = aBorrar;
6      aBorrar = aBorrar->sig;
7    }
8    if (aBorrar != NULL){
9    // Se ha encontrado un nodo que contiene el valor buscado
10     if (anterior == NULL){
11     //el valor buscado está en el nodo raíz, debe cambiar listaEnlazada
12       listaEnlazada = aBorrar->sig;
13     }
14     else{
15     //el valor buscado está en posición cualquiera no raíz, se puentea
16       anterior->sig = aBorrar->sig;
17     }
18     free(aBorrar);
19   }
20 }
```

En la versión presentada: se garantiza que la lista no está vacía. Si la lista no está vacía, se realiza el recorrido para localizar el nodo que contiene el valor buscado. Para realizar dicho recorrido se inician dos variables, líneas 2 y 3, de manera que ambas apuntarán a dos nodos consecutivos durante el recorrido que se realiza en las líneas 4 a 7. Este recorrido termina por dos razones:

- El valor buscado no se encuentra en la lista enlazada, en este caso `aBorrar` acaba en `NULL`. Se ha recorrido la lista enlazada por completo y, al no localizar ningún nodo que contiene el valor, no se hace nada.
- El valor buscado si está en la lista enlazada. En este caso `aBorrar` apunta al nodo que contiene el valor buscado (la condición del bucle `aBorrar->info != valor` es falsa). En este caso se debe comprobar de qué nodo se trata, si la raíz o cualquier otro nodo de la lista enlazada (`if` en la línea 10). Si es la raíz, se debe hacer que el segundo nodo pase a ser la nueva raíz, si existe (línea 12), y si no es la raíz, es cualquier otro nodo de la lista enlazada, se "puentea", como ya hemos hecho en anteriores ocasiones, haciendo uso del puntero `anterior`. En ambos casos, y a continuación, se debe liberar la memoria reservada para el nodo localizado, y es lo que se hace en la línea 18.

Se deja al estudiante, de nuevo, que reflexione sobre posibles situaciones que se pueden presentar como: ¿qué sucede si el nodo a eliminar es el único existente en la lista enlazada? ¿Qué sucede cuando el nodo a eliminar es último de la lista enlazada? Tal y como está escrito el código ¿funciona bien en esas situaciones? Siempre que se escribe código que gestiona listas enlazadas hay que hacerse esas preguntas y contemplar que están cubiertas todas las situaciones que se pueden presentar.

4. Para terminar

Una vez se tienen las ideas claras sobre los conceptos presentados: nodo, lista enlazada, recorrido secuencial, y se han comprendido las operaciones presentadas, el estudiante se debe plantear nuevos retos que consisten, básicamente, en nuevas operaciones que manejan colecciones de datos representadas mediante una lista enlazada. Por ejemplo, se puede plantear escribir el código correspondiente a otras operaciones como:

- Contar el número de nodos que tiene una lista enlazada.
- Contar el número de nodos de una lista enlazada que cumple una determinada condición, por ejemplo, que contiene un determinado valor.

- Insertar un nodo en una posición *i-ésima* de la lista enlazada. Para ello, se considera que el nodo raíz ocupa la posición 0, el segundo nodo ocupa la posición 1, y así sucesivamente hasta el último que ocupa la posición $n-1$. Importante, se debe garantizar que el valor ordinal de la posición es una posición correcta, es decir, dentro del rango $0,\ldots,n$, ambas inclusive. La posición n es válida porque significa añadir al final.

- Recorrer una lista enlazada eliminando todos los nodos que contienen un determinado valor, no solo la primera aparición, como se hace en la subsección 3.11, todos los nodos que contienen ese valor.

- Eliminar el nodo que se encuentra en la posición *i-ésima* de la lista enlazada. Para ello, se considera que el nodo raíz ocupa la posición 0, el segundo nodo ocupa la posición 1, y así sucesivamente hasta el último que ocupa la posición $n-1$.

Y muchas más operaciones que se pueden plantear. Para ello, el estudiante debe descargarse de la página de la asignatura en Studium el PDF con el título **Listas enlazadas** disponible en el **Tema 3**. A partir de la página 14 de dicho PDF hay ejercicios, tanto resueltos como propuestos, que el estudiante debe resolver.

Si hay un tema en la asignatura en la que es, todavía más cierto y más importante, la afirmación que se hace en la ficha de la asignatura, *a programar solo se aprende programando*, es justo en este tema y en esta lección sobre listas enlazadas. Es muy conveniente que se hagan muchos ejercicios, que se cometan muchos errores, que se utilice el depurador para localizar los errores de ejecución, ejecutando, paso a paso, un programa para localizar donde estamos perdiendo un puntero. Así, se recomienda que se resuelvan e implemente **todos** los ejercicios propuestos en el citado PDF **Listas enlazadas**, independientemente de que se solicite entregas en Studium.