Tecnología de la Programación
Estructuras Enlazadas Lineales

2020

Juan Antonio Sánchez Laguna
Grado en Ingeniería Informática
Facultad de Informática
Universidad de Murcia

TABLA DE CONTENIDOS

Estructuras de datos enlazadas lineales

Las estructuras de datos enlazadas lineales, también llamadas listas enlazadas, sirven para representar secuencias ordenadas (listas) de elementos de cualquier tipo. Se construyen conectando nodos que, a su vez, son estructuras de datos capaces de almacenar un elemento de la secuencia y, al menos, un puntero que permita acceder (enlace) al nodo que guarde el siguiente elemento de la secuencia.

Por ejemplo, la secuencia de números (3, 2, 5) se puede representar con una estructura enlazada como la de la Fig. 1. Cada nodo se representa gráficamente mediante una caja con dos partes: una contiene el elemento que guarda y la otra una flecha apuntando al nodo que guarda el siguiente elemento de la secuencia. El último nodo se distingue por tener un aspa en lugar de una flecha, indicando que no hay ningún otro nodo a continuación.

Fig. 1 Representación gráfica de una estructura enlazada lineal

El primer nodo de la estructura enlazada lineal es el más importante pues es el único a partir del cual se puede acceder al resto de nodos. El acceso se hace pasando de uno a otro gracias a los enlaces que los conectan. Por eso se dice que las estructuras enlazadas lineales tienen un modelo de acceso secuencial.

Para utilizar una estructura enlazada se necesita mantener en todo momento la dirección de su primer nodo. Este puntero, externo a la estructura enlazada, representa la secuencia completa y da acceso a todos sus elementos. De este modo, la secuencia vacía se puede representar dando el valor `NULL` a dicho puntero externo.

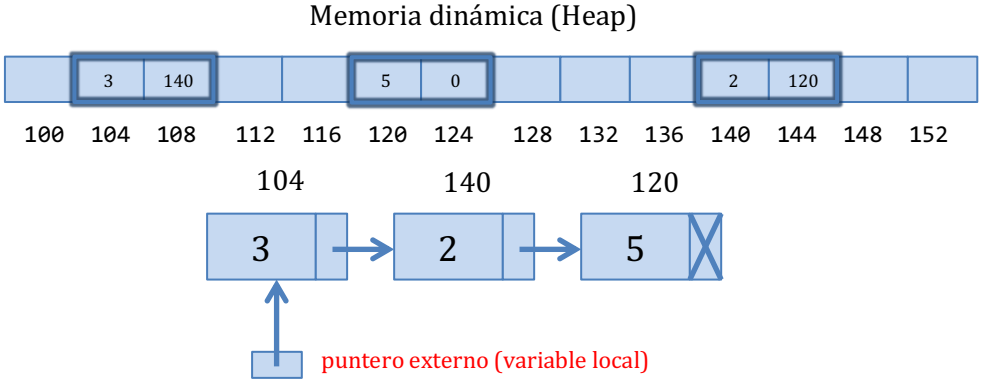

Fig. 2. Representación gráfica de la memoria dinámica usada por una estructura enlazada

Los nodos de una estructura enlazada no existen a priori, se van creando y conectando a ella a medida que se necesita añadir elementos a la secuencia. Esto requiere el uso de memoria dinámica, pero nos permite añadir, eliminar y reordenar elementos de la lista de forma eficiente. Basta con cambiar los enlaces que conectan los nodos. Este tipo de operaciones en un array puede requerir mover los datos de unas posiciones a otras. Además, como siempre se pueden añadir más nodos, el tamaño de la lista no está limitado como el de los arrays.

Del mismo modo, cuando se desea eliminar un elemento de la secuencia, es necesario desconectar el nodo que lo contiene de la estructura enlazada, y después, liberar la memoria asociada a dicho nodo. Pero esta operación es sencilla y rápida.

La estrategia del nodo cabecera

Como ya se ha visto, el puntero externo que mantiene la dirección del primer nodo de una estructura enlazada lineal representa toda la secuencia. Así pues, cada vez que se añade un elemento al principio de una secuencia, el valor de dicho puntero externo debe modificarse. Sin embargo, las inserciones a continuación de un elemento existente no producen el mismo efecto.

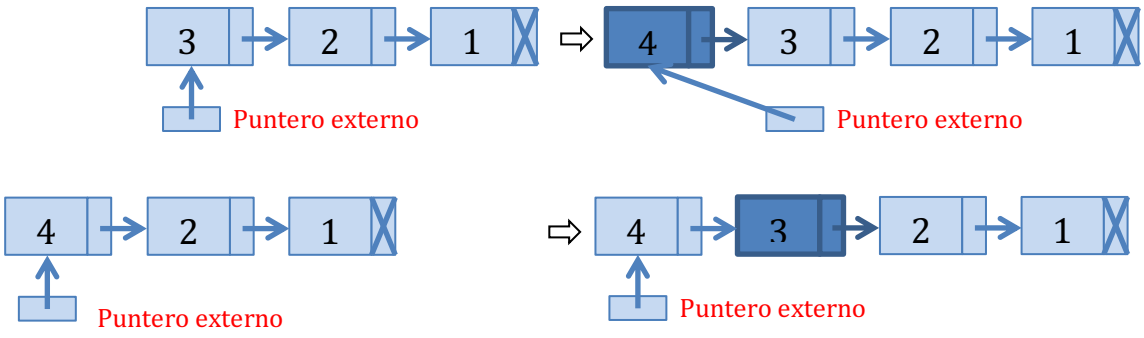

Fig. 3. Diferencia entre las inserciones por el principio y en posiciones intermedias

Algo similar ocurre si se suprime el primer elemento. El que estuviera segundo pasa a ser el primero. Esto motiva, de nuevo, el cambio del mencionado puntero externo. Sin embargo, la supresión de un elemento intermedio no conlleva esta modificación del puntero externo.

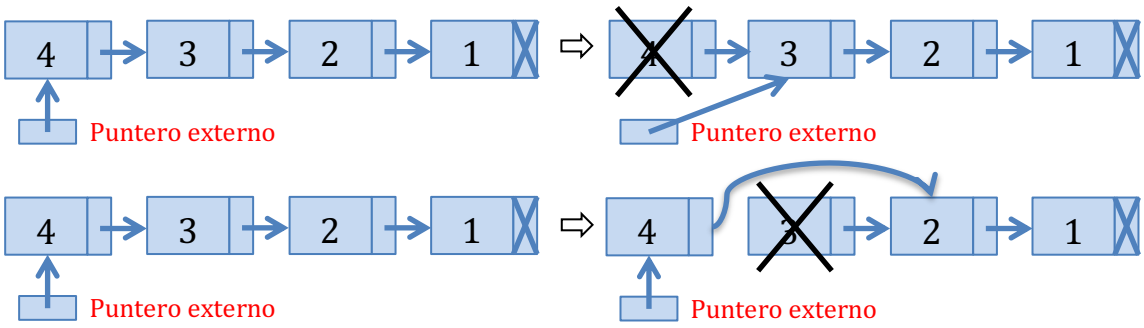

Fig. 4. Diferencia entre las supresiones por el principio y en posiciones intermedias

Usando la **estrategia del nodo cabecera** se evita hacer un tratamiento especial de las inserciones y supresiones por el principio, y se simplifica el código del resto de operaciones.

Esta estrategia consiste en representar la secuencia vacía mediante un nodo sin datos llamado cabecera. Además, el nodo cabecera debe permanecer siempre como primer nodo de la estructura enlazada. Esto permite que las inserciones y supresiones ya no tengan casos especiales, ni impliquen un cambio en el primer nodo de la estructura. Por lo tanto, cualquier puntero externo usado para gestionar la estructura enlazada apuntará siempre al mismo nodo, al nodo cabecera, independientemente de las inserciones y supresiones que se lleven a cabo.

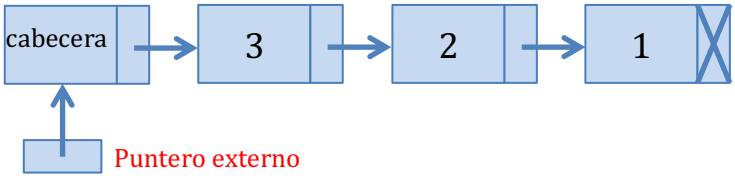

Fig. 5. Estructura enlazada con nodo cabecera

Implementación en C

Una estructura enlazada se define a partir de la definición de los nodos que la componen. Para definir un nodo en C se usa una estructura con dos campos. El primero, llamado **dato**, representará un elemento del tipo de datos de los elementos que vaya a guardar la estructura enlazada. El segundo, llamado **sig**, es un puntero a una estructura como la que se está definiendo y se encargará de almacenar la dirección en memoria del nodo que contenga el siguiente elemento de la secuencia.

```
struct Nodo {
  int dato;
  struct Nodo * sig;
};
typedef struct Nodo * NodoPtr;
```

Fig. 6. Definición de una estructura enlazada lineal de enteros en C

La estructura definida en la Fig. 6 permite crear estructuras enlazadas lineales de cualquier longitud, pues en ella no se especifica ningún valor que limite la cantidad de nodos que se pueden conectar uno a continuación de otro. También se define la palabra `NodoPtr` como sinónimo de puntero a `struct Nodo`. Esto permite simplificar la declaración de variables capaces de almacenar direcciones de nodos.

Para crear nodos en memoria dinámica se usa la función `malloc` de la biblioteca estándar de C `<stdlib.h>` que devuelve la dirección de la zona de memoria reservada. La cantidad de memoria necesaria se calcula aplicando `sizeof` al tipo `struct Nodo`.

```
NodoPtr nuevo_nodo( int dato ) {
  NodoPtr nuevo = malloc( sizeof( struct Nodo ) );
  nuevo->dato = dato;
  nuevo->sig = NULL;
  return nuevo;
}
```

Fig. 7. Función para crear un nuevo nodo en memoria dinámica.

La función de la Fig. 7 crea un nuevo nodo en memoria dinámica listo para ser conectado a una estructura enlazada. Primero se reserva la memoria necesaria y se almacena la dirección de la misma en la variable `nuevo`. A continuación es necesario inicializar todos los campos de la nueva estructura, especialmente los de tipo puntero. El campo `sig` se inicializa a `NULL`, pues así es como se indica que este nodo aún no está conectado con ningún otro. Finalmente se devuelve la dirección del nuevo nodo creado en memoria dinámica.

Para crear una estructura enlazada que represente una secuencia vacía basta con crear un nodo cuyo dato nunca será usado. Para ello también se puede utilizar la función `nuevo_nodo`. Su dirección se almacena en una variable que será el puntero externo usado para gestionar la secuencia.

```
// Creación de estructura enlazada que representa una secuencia de enteros vacía
NodoPtr numeros = nuevo_nodo(0);
```

Fig. 8. Creación de una estructura enlazada con cabecera representando una secuencia vacía.

Inserción por el principio

La inserción por el principio es la forma más sencilla de añadir elementos a una estructura enlazada previamente creada. Esta operación se usa con tanta frecuencia que conviene encapsularla en una función como la siguiente.

```
void inserta_principio( NodoPtr cabecera, int dato ) {
  NodoPtr nuevo = nuevo_nodo( dato );
  nuevo->sig = cabecera->sig;
  cabecera->sig = nuevo;
}
```

Fig. 9. Código para añadir un nodo por el principio.

La función `inserta_principio` recibe a través del parámetro `cabecera` una estructura enlazada ya creada, es decir, la dirección de su nodo cabecera. En el código, primero se crea el nuevo nodo para guardar el dato recibido a través del parámetro `dato`. Después, el nodo que hasta este momento contenía el primer elemento, es decir, el siguiente de la cabecera, se coloca a continuación del nuevo nodo. Por último, el nuevo nodo se coloca a continuación de la cabecera, por lo que el elemento que contiene pasa a ser el primero de la secuencia.

Para comprobar el buen funcionamiento de este y de otros algoritmos que usen estructuras enlazadas se deben probar, al menos, dos casos: cuando la estructura enlazada representa una secuencia vacía y cuando tiene uno o más elementos.

En el primer caso, el campo `sig` de la cabecera valdrá `NULL`. Al copiarlo en el campo `sig` del nuevo nodo y colocar éste a continuación de la cabecera, el nuevo nodo se convertirá en el que contenga el primer y único elemento de la secuencia.

Si la estructura representa una secuencia con uno o más elementos, el campo `sig` de la cabecera apuntará al nodo que contenga el primer elemento. Tras copiar esa dirección en el campo `sig` del nuevo nodo y colocar éste a continuación de la cabecera, todos los elementos que haya se mantendrán en la secuencia pero ocupando el siguiente puesto al que ocupaban hasta ese momento

La Fig. 10 muestra un ejemplo de uso de la función anterior. Primero se declara una variable llamada `naturales` que representará una secuencia mediante una estructura enlazada. Después, se usa `nuevo_nodo` para crear la secuencia vacía y, a continuación, se invoca tres veces a `inserta_principio` para añadir elementos.

```
int main( ) {
  NodoPtr naturales = nuevo_nodo( 0 );
  inserta_principio( naturales, 1 );
  inserta_principio( naturales, 2 );
  inserta_principio( naturales, 3 );
}
```

Fig. 10. Ejemplo de creación de una secuencia de números.

La inserción por el principio es una operación muy eficiente pues, independientemente del número de elementos que tenga en la secuencia, añadir otro siempre tiene el mismo coste computacional: una reserva de memoria y dos cambios de punteros. Por otro lado, es importante entender que el orden en el que se guardan los elementos es el inverso al usado para insertarlos. Por lo tanto, el resultado del ejemplo anterior será la secuencia (3,2,1).

Inserción a continuación de un nodo conocido

Si se conoce la dirección de memoria del nodo que guarda un elemento de la secuencia, la inserción de un nuevo elemento como su sucesor se reduce a dos instrucciones. Sin embargo, el orden de las mismas es fundamental. La función de la Fig. 11 inserta un nuevo elemento (`dato`) como sucesor del elemento almacenado en el nodo cuya dirección se recibe a través del parámetro `nptr`.

```
void inserta_sucesor( NodoPtr nptr, int dato ) {
  NodoPtr nuevo = nuevo_nodo( dato );
  nuevo->sig = nptr->sig;
  nptr->sig = nuevo;
}
```

Fig. 11. Código para añadir un elemento a continuación de otro.

Tras crear el nuevo nodo, lo primero que se hace es colocar a continuación suya el que hasta ese momento era el siguiente del nodo apuntado por `nptr`. Después, se coloca el nuevo nodo a continuación del apuntado por `nptr`. De ejecutar las instrucciones en el orden inverso se perdería la dirección del sucesor original del apuntado por `nptr` y, por tanto, de todos los nodos que le siguieran.

Recorridos y búsquedas

Tanto para recorrer una estructura enlazada completamente como para buscar elementos en ella se puede usar un puntero externo auxiliar que, gracias a un bucle, vaya apuntando a los distintos nodos pasando de uno a otro con el campo `sig` de cada uno. Además, el uso de la estrategia del nodo cabecera hace posible plantear algoritmos con bucles sin casos especiales. En todos ellos el puntero externo auxiliar (`aux`) se inicializa apuntando a la cabecera y en cada iteración apuntará al nodo anterior al que realmente se desea estudiar que será `aux->sig`.

```
int longitud( NodoPtr cabecera ) {
  NodoPtr aux = cabecera;
  int contador = 0;
  while ( aux->sig != NULL ) {
    contador = contador + 1;
    aux = aux->sig; // Avance
  }
  return contador;
}
```

Fig. 12. Longitud de una secuencia representada por una estructura enlazada lineal con cabecera.

Un ejemplo sencillo de recorrido es el de la función mostrada en la Fig. 12 que cuenta los elementos de la secuencia representada por una estructura enlazada. La función comienza declarando el puntero externo `aux` que se usará para recorrer toda la estructura enlazada. Este se inicializa apuntando a la cabecera de la misma. También se inicializa a cero la variable que guardará el número de elementos de la secuencia. A continuación, el bucle principal se ejecuta mientras que el campo `sig` del nodo apuntado por `aux` sea distinto de `NULL`, es decir, mientras que haya algún nodo válido sin estudiar. Como `aux` está inicializado apuntando a la cabecera, la primera comprobación determina la existencia de, al menos, un elemento en la secuencia. Si lo hay, se ejecutará la primera iteración y si no la función terminará devolviendo 0 pues la estructura representa una secuencia vacía.

Si la secuencia representada no está vacía, en cada iteración se realizan dos acciones. Primero se incrementa la variable `contador` para reflejar que hay un elemento más: el guardado en el nodo siguiente al apuntado por `aux`. Después se avanza, es decir, se modifica `aux` para que apunte al siguiente nodo de la estructura enlazada. El bucle acabará cuando `aux` apunte al último nodo de la estructura y su campo `sig` valga `NULL`. Es decir, después de haber contado el último elemento.

Modificando ligeramente el cuerpo del bucle de la función `longitud` se pueden hacer otras. Por ejemplo, la función de la Fig. 13 determina cuántos elementos hay en la secuencia que sean mayores que cero. Como puede verse, para contar el número de elementos que cumplan cierta condición basta con añadir una sentencia `if` que condicione el incremento del contador.

```
int positivos( NodoPtr cabecera) {
  NodoPtr aux = cabecera;
  int contador = 0;
  while ( aux->sig != NULL ) {
    if ( aux->sig->dato > 0 ) contador = contador + 1;
    aux = aux->sig; // Avance
  }
  return contador;
}
```

Fig. 13. Función que cuenta los elementos positivos de una secuencia.

Como último ejemplo de recorrido, la Fig. 14 muestra una función que imprime todos los elementos de una secuencia dada. La estrategia de recorrido es la misma, pero merece la pena destacar que el tratamiento del elemento, que en este caso consiste únicamente en mostrar su valor, se debe realizar siempre antes del avance del puntero auxiliar. Del mismo modo es importante recalcar que el puntero auxiliar comienza apuntando a la cabecera por lo que el elemento tratado en cada iteración será el contenido en el siguiente nodo al apuntado por `aux`.

```
void muestra( NodoPtr cabecera) {
  NodoPtr aux = cabecera;
  while ( aux->sig != NULL ) {
    // Tratamiento
    printf("%d ", aux->sig->dato );
    // Avance
    aux = aux->sig;
  }
}
```

Fig. 14. Recorrido básico de una estructura de datos enlazada lineal con cabecera.

Pero, sin duda, el aspecto más importante de la estrategia usada en los recorridos es comprender que la variable auxiliar es la que cambia de valor durante el bucle para ir apuntando a los distintos nodos. Las conexiones que hay entre ellos no deben cambiar. Es decir, no se altera la estructura enlazada, sólo se explora.

Un error muy frecuente consiste en pretender hacer el avance con la sentencia de la Fig. 15. Esta sentencia **destruye** la estructura enlazada dejándola inservible.

```
aux->sig = aux->sig->sig; // No avanza, destruye la lista;
```

Fig. 15. Sentencia incorrecta si se usa para avanzar durante un recorrido.

Al igual que para recorrer una estructura enlazada completamente, para buscar elementos en ella se debe tener en cuenta el caso de la secuencia vacía. La Fig. 16 muestra un primer ejemplo de búsqueda consistente en una función que determina si un elemento está incluido en una secuencia dada.

```
int existe( NodoPtr cabecera, int buscado ) {
  NodoPtr aux = cabecera;
  while ( aux->sig != NULL && aux->sig->dato != buscado ) {
    // Avance
    aux = aux->sig;
  }
  if ( aux->sig != NULL ) return 1;
  else return 0;
}
```

Fig. 16. Función que implementa una búsqueda secuencial en la estructura enlazada.

Aquí también se inicializa el puntero externo `aux` apuntando a la cabecera y el bucle se encarga de ir modificando su valor para recorrer todos los nodos. Sin embargo, además de al alcanzar el final, el bucle también debe acabar si, antes de alcanzarlo, se encuentra el elemento buscado. Para ello, la condición del bucle primero comprueba que exista algún nodo válido a continuación del apuntado por `aux` y, además, si el elemento allí almacenado es distinto del buscado. Es importante hacer las comprobaciones en este orden pues, en caso contrario, se podría intentar acceder al campo `dato` de un nodo no válido cuando `aux` estuviera apuntando al último nodo de la estructura y, por tanto, su campo `sig` valiera `NULL`.

Al terminar el bucle, `aux` siempre apunta a algún nodo de la estructura enlazada. Si el elemento buscado está incluido en la secuencia, `aux` quedará apuntando al nodo anterior al que lo contenga. En caso contrario, `aux` quedará apuntando al último nodo de la estructura, que puede ser el nodo cabecera cuando la estructura enlazada represente la secuencia vacía.

Si se numeran los elementos de una secuencia de longitud **n** empezando por el 1 y acabando por **n**, la función de la Fig. 17 devuelve el puesto que ocupa un elemento dado cuando éste se encuentra en la secuencia pasada como parámetro. En caso contrario devuelve -1.

```
int puesto( NodoPtr cabecera, int buscado ) {
  NodoPtr aux = cabecera;
  int puesto = 1;
  while ( aux->sig != NULL && aux->sig->dato != buscado ) {
    puesto = puesto + 1;
    aux = aux->sig;
  }
  if ( aux->sig != NULL ) return puesto;
  else return -1;
}
```

Fig. 17. Función que devuelve el índice de un elemento en una secuencia.

El código es muy similar al anterior con el añadido de la variable `puesto` que actúa como contador para calcular el puesto. Esta se inicializa a 1 y en cada iteración aumenta en una unidad. Al terminar el bucle su valor sólo es utilizado si se llegó a encontrar el elemento buscado, es decir, si el bucle terminó antes de que `aux` apuntara al último nodo porque `aux->sig->dato` coincidía con el valor buscado.

Supresión de elementos

Para eliminar un elemento se debe modificar el campo `sig` del nodo anterior al que lo contenga. En concreto se debe hacer que apunte al nodo que contenga el elemento siguiente al que se pretende borrar. Esto desconecta de la estructura enlazada el nodo que se desea borrar. Por lo tanto, primero se debe guardar su dirección para poder liberar la memoria que ocupa una vez que esté desconectado.

El primer elemento de la secuencia representada por una estructura enlazada es el más sencillo de eliminar. No hay que buscarlo porque siempre se encuentra a continuación de la cabecera. Evidentemente, sólo se borra si hay algo que borrar.

```
void suprime_primero( NodoPtr cabecera ) {
  NodoPtr borrar = cabecera->sig;
  if ( borrar != NULL ) {
    cabecera->sig = borrar->sig;
    free( borrar );
  }
}
```

Fig. 18. Supresión del primer elemento.

En la función de la Fig. 18, primero se guarda la dirección del nodo a borrar, después, si la secuencia representada no estaba vacía, se desconecta dicho nodo de la estructura enlazada colocando a continuación de la cabecera el nodo siguiente del que se va a borrar. Por último se libera la memoria dinámica usada por el nodo a borrar usando la función `free` declarada en el fichero de cabecera `<stdlib.h>`.

Una vez eliminado el primer elemento, el que hasta ese momento fuera el segundo pasará a ser el primero. Teniendo en cuenta este hecho resulta sencillo plantear un algoritmo que elimine todos los elementos de la estructura enlazada. La función de la Fig. 19 libera toda la memoria asociada a una estructura enlazada borrando todos sus elementos y después la cabecera.

```
void libera( NodoPtr cabecera ) {
  while ( cabecera->sig != NULL ) {
    NodoPtr borrar = cabecera->sig;
    cabecera->sig = borrar->sig;
    free( borrar );
  }
  free( cabecera );
}
```

Fig. 19. Liberación de la memoria usada por una estructura enlazada.

Como se puede ver, el bucle se ejecuta mientras exista algún nodo a continuación de la cabecera, es decir, mientras la secuencia representada no esté vacía. En cada iteración se borra el primer elemento de la secuencia y, al terminar el bucle, se elimina el nodo cabecera. Sin embargo, aunque la función libere toda la memoria de la estructura, el puntero externo que represente la secuencia en el código desde donde se invoque a `libera`, seguirá apuntando a la zona de memoria liberada. Por eso es una buena práctica asignarle `NULL` inmediatamente después.

```
...
libera( números );
números = NULL;
...
```

Fig. 20. Liberación de la memoria usada por una estructura enlazada.

Inserción por el final

Dada una estructura enlazada ya creada, para poder insertar un nuevo elemento al final de la secuencia que representa, primero se debe localizar su último nodo.

```
void inserta_final( NodoPtr cabecera, int dato ) {
  NodoPtr nuevo = nuevo_nodo( dato );
  NodoPtr aux = cabecera;
  while ( aux->sig != NULL ) {
    aux = aux->sig; // Avanzar
  }
  aux->sig = nuevo;
}
```

Fig. 21. Inserción por el final.

La función `inserta_final` mostrada en la Fig. 21 primero crea un nuevo nodo para guardar el dato recibido como segundo parámetro. A continuación, usando un puntero externo auxiliar, localiza la dirección del último nodo de la estructura. Para ello recorre la estructura enlazada desde el principio en busca de un nodo cuyo campo `sig` valga `NULL`. Al terminar el bucle, el nuevo nodo se coloca a continuación del apuntado por `aux`. Si la secuencia estuviera vacía no habría ningún nodo válido a continuación de la cabecera por lo que `aux` quedaría apuntando a ésta, y la inserción sería equivalente a una inserción por el principio.

Inserción en un puesto concreto

A diferencia de los arrays, las estructuras enlazadas no permiten un acceso indexado a sus elementos. Esto no impide que se pueda usar la posición de los elementos en la secuencia para referirse a ellos. Pero, evidentemente, la inserción de un nuevo elemento en un puesto concreto de la secuencia implica una búsqueda previa. Además, se debe tener en cuenta que los únicos puestos válidos son los que van desde el primero (el 1) hasta uno más del último, cuando se desea añadir un elemento al final de la secuencia, aumentando con ello su longitud en una unidad.

```
void inserta_en_puesto( NodoPtr cabecera, int dato, int puesto ) {
  NodoPtr nuevo = nuevo_nodo( dato );
  NodoPtr aux = cabecera;
  int i = 1;
  while ( i < puesto ) {
    i = i + 1;
    aux = aux->sig; // Avanzar
  }
  nuevo->sig = aux->sig;
  aux->sig = nuevo;
}
```

Fig. 22. Inserción en un puesto concreto.

En la función de la Fig. 22 se asume que el puesto pasado como parámetro es correcto. Eso hace innecesario comprobar si existe un siguiente nodo o no en la condición del bucle. Cuando el bucle termina, `aux` queda apuntando justo al nodo anterior al que debe ocupar el nuevo elemento. Este se colocará a continuación de `aux`, pero primero se copia el valor de `aux->sig` en el campo `sig` del nuevo nodo. De este modo, si ya había un elemento ocupando dicho puesto ahora pasará a estar a continuación del nuevo elemento insertado.

Búsqueda y eliminación

También es habitual combinar las búsquedas con las eliminaciones, puesto que para borrar un elemento concreto de la secuencia primero se debe localizar la posición del nodo anterior al que lo contiene.

```
void suprime_primera_aparicion( NodoPtr cabecera, int dato ) {
  NodoPtr aux = cabecera;
  while ( aux->sig != NULL && aux->sig->dato != dato ) {
    aux = aux->sig; // Avanzar
  }
  if ( aux->sig != NULL ) {
    NodoPtr borrar = aux->sig;
    aux->sig = borrar->sig;
    free( borrar );
  }
}
```

Fig. 23. Supresión de la primera aparición de un elemento.

La función de la Fig. 23 busca y elimina la primera aparición de un elemento en una secuencia. La primera parte del código consiste en un bucle para localizar la dirección del nodo anterior al primero que contenga el elemento buscado. En caso de no existir el bucle acaba dejando aux apuntando al último nodo de la estructura, por tanto, su campo `sig` valdrá `NULL`. Así pues, en la segunda parte del código, si al terminar el bucle el campo `sig` del nodo apuntado por aux no vale `NULL` es que se ha encontrado el nodo. En ese caso se procede a eliminarlo desconectándolo de la estructura y, posteriormente, liberando la memoria dinámica que ocupara.

Supresión múltiple

Un caso más complejo es el que se da cuando lo que se desea es eliminar todas las apariciones de un elemento. La función de la Fig. 24 presenta la solución. En este caso, lo más destacable es que no se hace una búsqueda sino un recorrido completo pues, a priori, no se sabe cuántos elementos puede haber iguales al buscado, ni dónde estarán, por lo que se necesita estudiar todos los elementos de la secuencia.

```
void suprime_todos( NodoPtr cabecera, int dato ) {
  NodoPtr aux = cabecera;
  while ( aux->sig != NULL ) {
    if ( aux->sig->dato == dato ) {
      NodoPtr borrar = aux->sig;
      aux->sig = borrar->sig;
      free( borrar );
    } else {
      aux = aux->sig; // Avanzar
    }
  }
}
```

Fig. 24. Supresión de todas las apariciones de un elemento.

En cada iteración se comprueba si el elemento del nodo siguiente al apuntado por aux coincide con el buscado. En caso afirmativo, se elimina dicho nodo, y en caso contrario, se modifica aux para que apunte al siguiente nodo de la estructura enlazada. Es importante comprender que cuando se elimina un nodo, el que queda

a continuación de aux es justo el que toca estudiar en la siguiente iteración. Por eso, ni se necesita **ni se debe** avanzar explícitamente tras la eliminación.

Comparación entre estructuras enlazadas y Arrays

Un array también es una estructura de datos capaz de representar una secuencia de elementos. Pero, en principio, su tamaño máximo queda fijado en el código fuente al declarar el array, y no hay ninguna forma eficiente de cambiarlo en tiempo de ejecución. Por su parte, las estructuras enlazadas cambian de tamaño de forma dinámica según se necesita almacenar más o menos elementos.

Además, cada vez que se necesita añadir un elemento en una posición intermedia de una secuencia representada mediante un array, es necesario mover todos los elementos que quedarán a continuación del nuevo en la secuencia dejando así hueco para él. Algo similar ocurre cuando se suprime un elemento intermedio, en este caso para cerrar el hueco producido. Sin embargo, tras localizar un elemento en una estructura enlazada, insertar otro a continuación suya o suprimirlo es una operación sencilla y rápida que no depende del número de nodos de la estructura.

Pero los arrays también tienen ventajas frente a las estructuras enlazadas. Permiten acceso directo indexado a sus elementos, y el gasto de memoria por elemento es justo el necesario para almacenarlo. Las estructuras enlazadas siguen un modelo de acceso secuencial, lo que implica hacer un recorrido para localizar el nodo al que se quiera acceder. El uso de memoria también es mayor pues, por cada elemento se necesita almacenar un puntero al siguiente nodo.

En general no se puede decir que una forma de representar secuencias sea mejor que la otra. Depende del uso que se vaya a hacer en el programa. Claramente, las estructuras enlazadas se adaptan al tamaño necesario en cada momento y permiten inserciones y supresiones eficientes en posiciones intermedias. Pero los arrays soportan acceso indexado y usan la memoria mínima imprescindible.

Estructuras enlazadas lineales de doble enlace

Si cada nodo cuenta con un único puntero se dice que la estructura enlazada es de simple enlace, pero también se pueden definir nodos con dos punteros, uno hacia el nodo que contenga el siguiente elemento de la secuencia y otro hacia el nodo que contenga el elemento anterior. En este caso se dice que la estructura enlazada lineal es de doble enlace.

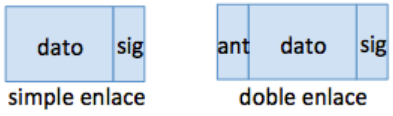

Fig. 25 Nodos usados para construir estructuras enlazadas lineales de simple enlace y de doble enlace

Evidentemente, todas las operaciones con estructuras doblemente enlazadas serán algo más complejas, pues hay más punteros que actualizar. Además, el uso de memoria por nodo aumenta. A cambio, usando este tipo de nodos es posible recorrer la secuencia hacia atrás más eficientemente que con las estructuras de simple enlace.

Aplicaciones de las estructuras enlazadas lineales

En general, en cualquier aplicación en la que se requiera trabajar con un número indeterminado y cambiante de forma dinámica de elementos es útil utilizar una estructura enlazada lineal. Además, son la base para construir otras estructuras de datos importantes entre las que se encuentran las Pilas y las Colas.

¿Por qué usar la estrategia del nodo cabecera?

Cuando no se usa la estrategia del nodo cabecera, la variable con el puntero externo que representa la secuencia apunta directamente al nodo que contiene el primer elemento. Y si la estructura enlazada representa una secuencia vacía, el valor de dicho puntero será `NULL`. La ejecución de las funciones que permiten añadir o suprimir elementos puede implicar un cambio de valor en dicha variable, haciendo que apunte a un nodo diferente del que antes apuntara. Sin embargo, el paso de parámetros en C se realiza por valor, es decir, las funciones no pueden modificar las variables que se usan como argumentos de la invocación.

Este problema se puede solucionar utilizando la estrategia de la simulación de paso de parámetros por referencia en C. Esta estrategia consiste en pasar la dirección de memoria de la variable cuyo valor se quiera usar o modificar dentro de la función. La Fig. 26 muestra cómo se puede escribir la función `inserta_principio` para que actúe de este modo.

```
void inserta_principio( NodoPtr * primero, int dato ) {
  NodoPtr nuevo = nuevo_nodo( dato );
  nuevo->sig = *primero;
  *primero = nuevo;
}


int main( ) {
  NodoPtr naturales = NULL;
  inserta_principio( &naturales, 1 );
  inserta_principio( &naturales, 2 );
  inserta_principio( &naturales, 3 );
}
```

Fig. 26. Inserción por el principio en una estructura enlazada lineal sin cabecera.

Como se puede ver, el parámetro `primero` es un puntero a un `NodoPtr`, es decir, un puntero a un puntero a una estructura `struct Nodo`. Esto obliga a usar como argumento de la llamada la dirección de la variable que contiene la dirección del primer nodo de la estructura. El operador `&` aplicado a la variable `naturales` devuelve su dirección, que es lo que finalmente se usa como argumento en la llamada a `inserta_principio`. En el código de la función, tras crear el nuevo nodo, se coloca a continuación suya el nodo que hasta ahora era el primero, es decir, el apuntado por la variable `naturales`. Esto se hace aplicando el operador * (desreferencia) al parámetro `primero` para obtener el valor de lo apuntado por él. Finalmente, se vuelve a usar el operador * sobre `primero` para modifica el valor de la variable `naturales` haciendo que apunte al nuevo nodo.

Esta estrategia permite reducir la cantidad de memoria usada ya que no se necesita un nodo cabecera extra, pero también hace que el código sea algo más

complejo. Además, en la mayoría de operaciones se necesita código que trate de forma especial el caso de la secuencia vacía.

Existen varias alternativas diferentes que permiten evitar que la secuencia vacía se trate de forma especial. Entre ellas está la del nodo cabecera que, además, evita tener que usar la simulación de paso de parámetros por referencia, lo que hace que sea una estrategia útil y sencilla de implementar. Por eso se ha elegido aplicar dicha estrategia en este texto.

Posibles optimizaciones

El paso de parámetros en C se hace siempre por valor. De este modo, en una función como la de la Fig. 12, el parámetro `cabecera` es en realidad una variable local, cuyo valor será el mismo que el usado al invocar la función. Por ejemplo, supongamos que en la función `main` existe una variable llamada `naturales`, que apunte a la cabecera de una estructura enlazada, y que se invoca la función `longitud` usando como argumento dicha variable, el parámetro `cabecera` recibirá una copia de la dirección del nodo cabecera. Por lo tanto, aunque en el código de la función se modifique el valor de la variable `cabecera`, la variable naturales seguirá apuntando al mismo nodo que al principio. Como muestra la Fig. 27, esto permite eliminar la necesidad de la variable auxiliar que se usaba en la función ya que se puede usar el propio parámetro para recorrer toda la estructura.

```
int longitud( NodoPtr cabecera) {
  int contador = 0;
  while ( cabecera->sig != NULL ) {
    contador = contador + 1;
    cabecera = cabecera->sig; // Avance
  }
  return contador;
}
```

Fig. 27. Ejemplo de uso de un parámetro de la función como puntero auxiliar en un recorrido.

Además de reducir el número de variables locales, también se puede optimizar el código eliminando accesos indirectos a memoria. Los accesos indirectos aparecen cuando se usa el operador `->` y lo que implican es un doble acceso a memoria. El primero, para obtener el valor de la variable y el segundo para usarlo al recuperar o modificar el valor apuntado por la misma. Por lo tanto, reduciendo su número se puede conseguir que el código generado sea más rápido.

En casos como el de la función `muestra` en los que se recorre una estructura enlazada, pero no se realizan inserciones ni supresiones, es posible reducir el número de accesos indirectos inicializando el puntero auxiliar al primer nodo que potencialmente puede tener datos, es decir, el siguiente a la cabecera. De este modo, tanto en la comprobación del bucle, como en los posibles accesos a los elementos que se hagan dentro de él, deja de ser necesario acceder al campo `sig`. En este caso, en cada iteración el elemento apuntado por el puntero auxiliar sí que será el que interesa estudiar. Así pues, la Fig. 28 incluye el código de la función `muestra` con el número de accesos indirectos reducido.

```
void muestra( NodoPtr cabecera ) {
  NodoPtr aux = cabecera->sig; // Saltamos cabecera
  while ( aux != NULL ) {
    // Tratamiento
    printf(" %d", aux->dato );
    // Avance
    aux = aux->sig;
  }
}
```

Fig. 28. Ejemplo de reducción de accesos indirectos en recorridos de estructuras enlazadas.