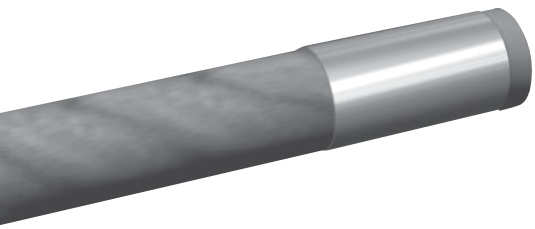
Capítulo 32

Listas, pilas y colas en C



Contenido

Introducción

En este capítulo se estudian estructuras de datos dinámicas. Al contrario que las estructuras de datos estáticas (*arrays* —listas, vectores y tablas— y *estructuras*) en las que el tamaño en memoria se establece durante la compilación y permanece inalterable durante la ejecución del programa, las estructuras de datos dinámicas crecen y se contraen a medida que se ejecuta el programa.

Una lista enlazada (ligada o encadenada, *linked list*) es una colección de elementos (denominados nodos) dispuestos uno a continuación de otro, cada uno de ellos conectado al siguiente elemento por un "enlace" o "referencia". En el capítulo se desarrollan algoritmos para insertar, buscar y borrar elementos en las listas enlazadas.

Una pila es una estructura de datos que almacena y recupera sus elementos atendiendo a un estricto orden. Las pilas se conocen también como estructuras LIFO (*Last-in*, *first-out*, último en entrar-primero en salir), todas las inserciones y retirada de elementos se realizan por un mismo extremo denominado *cima* de la pila. Las pilas se utilizan frecuentemente en programas y en la vida diaria.

Las colas se conocen como estructuras FIFO (*First-in*, *First-out*, primero en entrar-primero en salir), debido a la forma y orden de inserción y de extracción de elementos de la cola. Las colas tienen numerosas aplicaciones en el mundo de la computación: colas de mensajes, colas de tareas a realizar por una impresora, colas de prioridades.

Conceptos clave

- Asignación de memoria
- Eliminar un nodo en una lista enlazada
- Estructura FIFO
- Lista enlazada
- Memoria libre
- Puntero nulo
- Punteros
- Recorrido de una lista
- Referencia
- Variables puntero

Listas enlazadas

Una *lista enlazada* es una colección o secuencia de elementos dispuestos uno detrás de otro, en la que cada elemento se conecta al siguiente por un "enlace" o "puntero". La idea básica consiste en construir una lista cuyos elementos, llamados **nodos**, se componen de dos partes o *campos*: la primera contiene la información y es, por consiguiente, un valor de un tipo genérico (denominado *Dato*, *TipoElemento*, *Info*, etc.) y la segunda parte o *campo* es un puntero (denominado *enlace* o *sgte*) que apunta al siguiente elemento de la lista.

Figura 32.1 Lista enlazada (representación simple).

La representación gráfica más extendida es aquella que utiliza una caja (un rectángulo) con dos secciones en su interior. En la primera sección se escribe el elemento o valor del dato, y en la segunda, el enlace o puntero mediante una flecha que sale de la caja y apunta al nodo siguiente.

e_1, e_2, ... e_n son valores del tipo `TipoElemento`

Figura 32.2 Lista enlazada (representación gráfica típica).

Estructura de una lista
Una **lista enlazada** consta de un número de elementos y cada elemento tiene dos componentes (*campos*), un puntero al siguiente elemento de la lista y un valor, que puede ser de cualquier tipo.

Los enlaces se representan por flechas para facilitar la comprensión de la conexión entre dos nodos; ello indica que el enlace tiene la dirección en memoria del siguiente nodo. En la figura 32.2

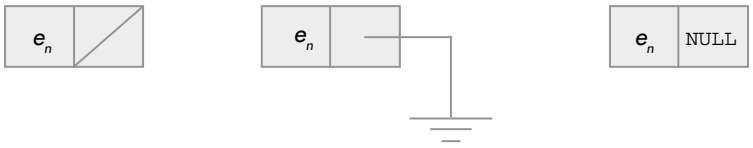


Figura 32.3 Diferentes representaciones gráficas del nodo último.

los nodos forman una secuencia desde el primer elemento (e_1) al último elemento (e_n). El primer nodo se enlaza al segundo nodo; éste se enlaza al tercero y así sucesivamente hasta llegar al último nodo. El último nodo se debe representar de forma diferente para significar que éste no se enlaza a ningún otro. La figura 32.3 muestra distintas representaciones gráficas que se utilizan para dibujar el campo enlace del último nodo.

Clasificación de las listas enlazadas

Las listas se pueden dividir en cuatro categorías:

- *Listas simplemente enlazadas*. Cada nodo (elemento) contiene un único enlace que conecta ese nodo al nodo siguiente o nodo sucesor. La lista es eficiente en recorridos directos ("adelante").
- *Listas doblemente enlazadas*. Cada nodo contiene dos enlaces, uno a su nodo predecesor y el otro a su nodo sucesor. La lista es eficiente tanto en recorrido directo ("adelante") como en recorrido inverso ("atrás").
- *Lista circular simplemente enlazada*. Una lista enlazada simplemente en la que el último elemento (cola) se enlaza al primer elemento (cabeza) de tal modo que la lista puede ser recorrida de modo circular ("en anillo").
- *Lista circular doblemente enlazada*. Una lista doblemente enlazada en la que el último elemento se enlaza al primer elemento y viceversa. Esta lista se puede recorrer de modo circular (en anillo) tanto en dirección directa ("adelante") como inversa ("atrás").

Por cada uno de estos cuatro tipos de estructuras de listas, se puede elegir una implementación basada en arreglos o una implementación basada en *punteros*. Estas implementaciones difieren en el modo en que asigna la memoria para los datos de los elementos, cómo se enlazan juntos los elementos y cómo se accede a dichos elementos. De forma más específica, según la forma de reservar memoria las implementaciones pueden hacerse con:

- Asignación fija, o estática, de memoria mediante arreglos.
- Asignación dinámica de memoria mediante punteros.

Dado que la asignación fija de memoria mediante arrays es más ineficiente, se utilizará en este capítulo la *asignación de memoria* mediante punteros, dejando como ejercicio al lector la implementación mediante arreglos.

Conceptos básicos sobre listas
Una lista enlazada consta de un conjunto de nodos. Un **nodo** consta de un campo dato y un puntero que apunta al "siguiente" elemento de la lista.

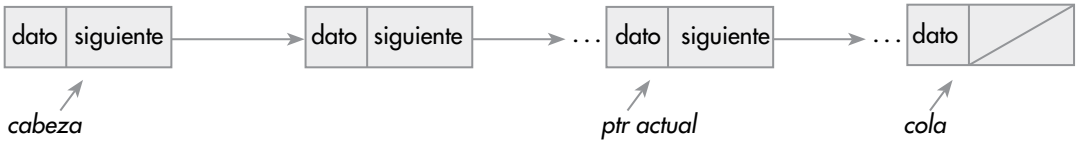


Figura 32.4 Representación gráfica de una lista enlazada.

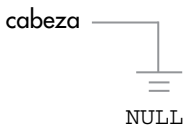


Figura 32.5 Representación de lista *vacía* con `cabeza = null`.

El primer nodo, **frente**, es el nodo apuntado por **cabeza**. La lista encadena nodos juntos desde el frente al final (**cola**) de la lista. El final se identifica como el nodo cuyo campo puntero tiene el valor `NULL = 0`. La lista se recorre desde el primero al último nodo; en cualquier punto del recorrido la posición actual se *referencia* por el puntero `ptrActual`. En el caso en que la lista no contiene ningún nodo (está vacía), el puntero `cabeza` es nulo.

Operaciones en listas enlazadas

Una lista enlazada requiere realizar la gestión de los elementos contenidos en ella. Estos controles se manifiestan en forma de las operaciones básicas, especificadas al definir el tipo abstracto lista, que tendrán las siguientes funciones:

- *Declarar los tipos nodo y puntero a nodo*.
- *Inicializar o crear*.
- *Insertar elementos en una lista*
- *Eliminar elementos de una lista*
- *Buscar elementos de una lista* (comprobar la existencia de elementos en una lista)
- *Recorrer una lista enlazada* (visitar cada nodo de la lista)
- *Comprobar si la lista está vacía*

Declaración de un nodo

Cada nodo de una lista enlazada combina dos partes: un tipo de dato (entero, real, doble, carácter o tipo predefinido) y un enlace (puntero) al siguiente nodo. En C se puede declarar un nuevo tipo de dato, correspondiente a un nodo, mediante la especificación de `struct` que agrupa las dos partes citadas.

```
struct Nodo
{
  int dato;
  struct Nodo* enlace;
};
```

```
typedef struct Nodo
{
   int dato;
   struct Nodo *enlace;
} NODO;
```

La declaración utiliza el tipo `struct` que permite agrupar campos de diferentes tipos, el campo dato y el campo enlace. Con `typedef` se puede declarar a la vez un nuevo identificador de `struct Nodo`; en el caso anterior se ha elegido `NODO`.

Dado que los tipos de datos que se puede incluir en una lista pueden ser de cualquier tipo (enteros, dobles, caracteres o incluso cadenas), con el objeto de que el tipo de dato de cada nodo se pueda cambiar con facilidad se suele utilizar la sentencia `typedef` para declarar el nombre de `elemento` como un sinónimo del tipo de dato de cada campo. A continuación se muestra la declaración:

```
typedef double elemento;

struct Nodo
{
  elemento dato;
  struct nodo *enlace;
};
```

Entonces, si se necesita cambiar el tipo de elemento en los nodos, sólo se tendrá que cambiar la sentencia de declaración de tipos que afecta a `elemento`. Siempre que una función necesite referirse al tipo de dato del nodo, puede utilizar el nombre `elemento`.

Ejemplo 32.1

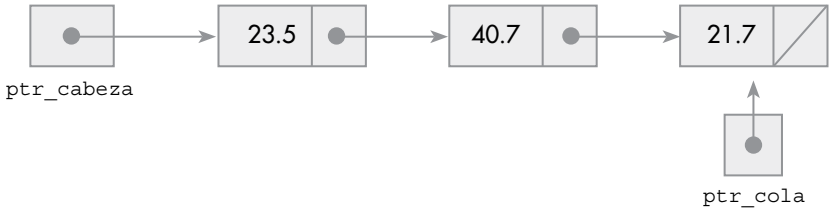
En este ejemplo se declara un tipo denominado `Punto`; representa un punto en el plano con su coordenada *x* y *y*. También se declara el tipo `Nodo` con el campo dato del tipo `Punto`. Por último, se define un puntero a `Nodo`.

```
#include <stdlib.h>

typedef struct punto
{
  float x, y;
} Punto;

typedef struct nodo
{
  PUNTO dato;
  struct nodo* enlace;
} Nodo;

Nodo* cabecera;
cabecera = NULL;
```



Declaración del nodo

```
typedef double elemento;
struct nodo
{
  elemento dato;
  struct nodo *enlace;
};
```

Definición de punteros

```
struct nodo *ptr_cabeza;

struct nodo *ptr_cola;
```

Figura 32.6 Declaraciones de tipos en lista enlazada.

Apuntador (puntero) de cabecera y cola

A una lista enlazada se accede a través de uno o más *punteros* a los nodos. El acceso más frecuente a una lista enlazada es a través del primer nodo de la lista que se llama **cabeza** o **cabecera** de la lista, o simplemente **L**. En ocasiones, se mantiene también un puntero al último nodo de una lista enlazada, es el apuntador (puntero) **cola**.

Cada puntero a un nodo debe ser declarado como una variable puntero. Por ejemplo, si se mantiene una lista enlazada con un puntero de cabecera y otro de cola, se deben declarar dos *variables puntero*:

```
struct nodo *ptr _cabeza;
struct nodo *ptr_cola;
```

El tipo `struct nodo` a veces se simplifica utilizando la declaración `typedef`. Así se puede escribir:

```
typedef struct nodo Nodo;
typedef struct nodo* ptrNodo;
ptrNodo ptr_cabeza;
ptrNodo ptr_cola;
```

Acceso a una lista
La construcción y manipulación de una lista enlazada requiere el acceso a los nodos de la lista a través de uno o más punteros a nodos. Normalmente, un programa incluye un puntero al primer nodo (*cabeza*) y un puntero al último nodo (*cola*).

El apuntador nulo

La figura 32.7 muestra una lista con un *apuntador cabeza* y un *apuntador nulo* al final de la lista sobre el que se ha escrito la palabra `NULL`. La palabra `NULL` representa el **apuntador nulo**, que es una constante especial de C. Se puede utilizar el apuntador nulo para cualquier valor de apuntador que no apunte a ningún sitio. El apuntador nulo se utiliza, normalmente, en dos situaciones:

- Usar el apuntador nulo en el campo enlace o siguiente del nodo final de una lista enlazada.
- Cuando una lista enlazada no tiene ningún nodo, se utiliza el apuntador `NULL` como apuntador de cabeza y de cola. Tal lista se denomina **lista vacía**.

En un programa, el apuntador nulo se puede escribir como `NULL`, que es una constante de la biblioteca estándar `stdlib.h`.[1] El apuntador nulo se puede asignar a una variable apuntador con una sentencia de asignación ordinaria. Por ejemplo:

```
ptrNodo *ptr_cabeza;
ptr_cabeza = NULL ;          /* incluir archivo stdlib.h */
```

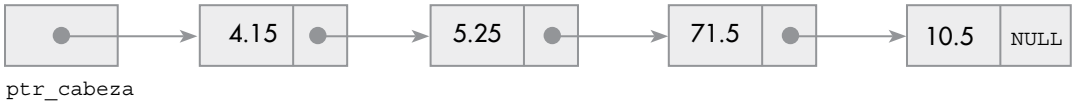


Figura 32.7 Apuntador `NULL`.

A recordar

El apuntador de cabeza y de cola en una lista enlazada puede ser `NULL`, lo que indicará que la lista está vacía (no tiene nodos). Éste suele ser un método usual para construir una lista. Cualquier función que se escribe para manipular listas enlazadas debe poder manejar un apuntador de cabeza y un apuntador de cola nulos.

Construcción de una lista

Una primera operación que se realiza con una lista es asignar un valor inicial, que es el de *lista vacía*. Sencillamente consiste en asignar `nulo` al puntero de acceso a la lista:

```
Nodo* cabeza;
cabeza = NULL;
```

Para añadir elementos a la lista se utiliza la operación `insertar( )`, con sus diferentes versiones. También, para la creación de una lista enlazada, se utiliza el siguiente algoritmo:

Paso 1. Declarar el tipo de dato y el puntero de cabeza o primero.
Paso 2. Asignar memoria para un elemento del tipo definido anteriormente utilizando alguna de las funciones de asignación de memoria (`malloc ( )`, `calloc ( )`, `realloc ( ) )` y un *cast* para la conversión de `void*` al tipo puntero a nodo.

[1] A veces algunos programadores escriben el puntero nulo como 0, pero pensamos que es un estilo más claro escribirlo como NULL.

Paso 3. Crear iterativamente el primer elemento (`cabeza`) y los elementos sucesivos de la lista enlazada simplemente.
Paso 4. Repetir hasta que no haya más entrada para el elemento.

Ejemplo 32.2

Crear una lista enlazada de elementos que almacenen datos de tipo entero.

Un elemento de la lista se puede definir con la ayuda de la estructura siguiente:

```
typedef elemento int;
struct nodo
{
  elemento dato;
  struct nodo * siguiente;
};
typedef struct  nodo Nodo;
```

En la estructura `nodo` hay dos miembros, `dato` y `siguiente`. Este último es un puntero al siguiente nodo, y `dato` contiene el valor del elemento de la lista. También se declara un nuevo tipo: `Nodo` que es sinónimo de `struct nodo`.

El siguiente paso para construir la lista es declarar la variable `primero` que apuntará al primer elemento de la lista:

```
Nodo *primero = NULL
```

A continuación, se crea un elemento de la lista, para lo cual hay que reservar memoria, tanta como tamaño tenga cada nodo, y asignar la dirección de la memoria reservada al puntero `primero`:

```
primero = (Nodo*)malloc (sizeof (Nodo) );
```

Con el operador `sizeof` se obtiene el tamaño de cada nodo de la lista, la función `malloc ( )` devuelve un puntero genérico (`void*`), por lo que se convierte a `Nodo*`. Ahora se puede asignar un valor al campo dato:

```
primero->dato = 11;
primero->siguiente = NULL;
```

La operación de crear un nodo se puede hacer en una función a la que se pasa el valor del campo `dato` y del campo `siguiente`. La función devuelve un puntero al nodo creado:

```
Nodo* crearNodo (elemento x, Nodo* enlace)
{
  Nodo *p;
  p = (Nodo*)malloc (sizeof (Nodo) );
  p->dato = x;
  p->siguiente = enlace;
  return p;
}
```

La llamada a la función `crearNodo ( )` para crear el primer nodo de la lista:

```
primero = crearNodo (11, NULL);
```

Si ahora se desea añadir un nuevo elemento con un valor 6, y situarlo en el primer lugar de la lista, se escribe simplemente:

```
primero = crearNodo (6,primero);
```

Por último, para obtener una lista compuesta de 4, 6, 11, se habría de ejecutar

```
primero = crearNodo (4,primero);
```

Inserción de un elemento en una lista

El algoritmo empleado para añadir o insertar un elemento en una lista enlazada varía dependiendo de la posición en que se desea insertar el elemento. La posición de inserción puede ser:

- En la cabeza (elemento primero) de la lista.
- En el final de la lista (elemento último).
- Antes de un elemento especificado, o bien
- Después de un elemento especificado.

Insertar un nuevo elemento en la cabeza de una lista

El proceso de inserción se puede resumir en este algoritmo:

1. Asignar un nuevo nodo apuntado por `nuevo` que es una variable puntero local que apunta al nuevo nodo que se va a insertar en la lista.
2. Situar el nuevo elemento en el campo `dato` del nuevo nodo.
3. Hacer que el campo enlace `siguiente` del nuevo nodo apunte a la cabeza (primer nodo) de la lista original.
4. Hacer que `cabeza` (puntero cabeza) apunte al nuevo nodo que se ha creado.

Ejemplo 32.3

Una lista enlazada contiene tres elementos: 10, 25 y 40. Insertar un nuevo elemento, 4, en cabeza de la lista

Pasos 1 y 2

`nuevo`

`cabeza`

Código C

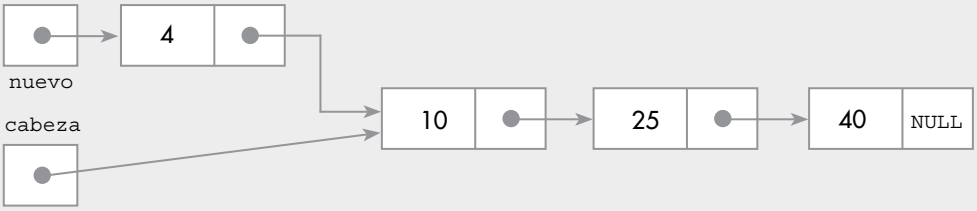
```
Nodo* nuevo;
nuevo = (Nodo*)malloc (sizeof (Nodo) ); /* asigna un nuevo nodo */
nuevo -> dato = entrada;
```

Paso 3

El campo enlace (`siguiente`) del nuevo nodo apunta a la cabeza actual de la lista.

Código C

```
nuevo -> siguiente = cabeza;
```

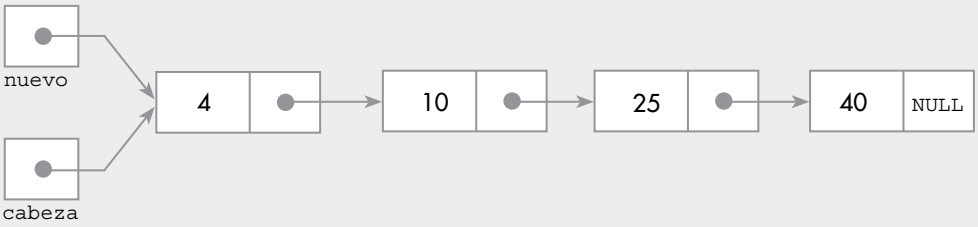


Paso 4

Se cambia el puntero de `cabeza` para apuntar al nuevo nodo creado; es decir, el puntero de `cabeza` apunta al mismo sitio que apunte `nuevo`.

Código C

```
cabeza = nuevo;
```



En este momento, la función de insertar un elemento en la lista termina su ejecución, la variable local `nuevo` desaparece y sólo permanece el puntero de cabeza `cabeza` que apunta a la nueva lista enlazada.

El código fuente de la función `inserPrimero ( )`:

```
void inserPrimero (Nodo** cabeza, Item entrada)
{
  Nodo *nuevo ;
  nuevo = (Nodo*)malloc (sizeof (Nodo) );
  nuevo -> dato = entrada;                    /* pone elemento en nuevo    */
  nuevo -> siguiente = *cabeza;               /* enlaza con el primer nodo */
  *cabeza = nuevo;
}
```

Ejercicio 32.1

Se va a formar una lista enlazada de números aleatorios. El programa que realiza esta tarea inserta los nuevos nodos por la cabeza de la lista. Una vez creada la lista, se recorren los nodos para mostrar los números pares.

La función `inserPrimero ( )` añade un nodo a la lista, siempre como nodo cabeza. El primer argumento es un puntero a puntero porque tiene que modificar la variable `cabeza`, que es a su vez un puntero a `Nodo`. La función `crearNodo ( )` reserva memoria para un nodo, asigna el campo dato y devuelve la dirección del nodo creado.

La función `main ( )` primero inicializa el puntero `cabeza` a `NULL`, que es la forma de indicar *lista vacía*. En un bucle se generan números aleatorios; cada número generado se inserta como nuevo nodo en la lista, invocando a la función `inserPrimero ( )`; el bucle termina cuando se genera el 0. Una vez terminado el bucle de creación de la lista, otro bucle realiza tantas iteraciones como nodos tiene la lista, y en cada pasada escribe el campo dato si es par.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MX 3131
typedef int Item;
typedef struct Elemento
{
  Item dato;
  struct Elemento* siguiente;
} Nodo;

void inserPrimero (Nodo** cabeza, Item entrada);
Nodo* crearNodo (Item x);

void main ( )
{
  Item d;
  Nodo *cabeza, *ptr;
  int k;

  cabeza = NULL;                  /* lista vacía */
  randomize ( );
    /* El bucle termina cuando se genera el número aleatorio 0 */
  for (d = random (MX); d; )
  {
    inserPrimero (&cabeza, d);
    d = random (MX);
  }                               /* recorre la lista para escribir los pares */
  for (k = 0, ptr = cabeza; ptr; )
  {
    if (ptr -> dato%2 == 0)
    {
      printf ("%d ", ptr -> dato);
      k++;
      printf ("%c", (k%12 ? ' ' : '\n') );   /* 12 datos por línea */
    }
    ptr = ptr -> siguiente;
  }
  printf ("\n\n");
}

void inserPrimero (Nodo** cabeza, Item entrada)
{
  Nodo *nuevo ;
  nuevo = crearNodo (entrada);
  nuevo -> siguiente = *cabeza;
```

```
  *cabeza = nuevo;
}

Nodo* crearNodo (Item x)
{
  Nodo *a ;
  a = (Nodo*)malloc (sizeof (Nodo) ); /* asigna nuevo nodo */
  a -> dato = x;
  a -> siguiente = NULL;
  return a;
}
```

Inserción de un nodo al final de la lista

La inserción al final de la lista es menos eficiente debido a que, normalmente, no se tiene un puntero al último elemento de la lista y entonces se ha de seguir la traza desde la cabeza de la lista hasta el último nodo de la lista y a continuación realizar la inserción. Cuando `ultimo` es puntero que apunta al último nodo de la lista, las sentencias siguientes insertan un nodo al final de la lista:

```
ultimo -> siguiente = crearNodo (x);
ultimo  -> siguiente -> siguiente = NULL;
ultimo = ultimo -> siguiente;
```

La primera sentencia crea un nuevo nodo, llamada a `crearNodo ( )`, y se asigna al campo `siguiente` del último nodo de la lista (antes de la inserción) de modo que el nuevo nodo ahora es el último. La segunda sentencia establece el campo `siguiente` del nuevo último nodo a `NULL`. Y la última sentencia pone la variable `ultimo` al nuevo último nodo de la lista.

La función `inserFinal ( )` tiene como entrada el puntero `cabeza`, recorre la lista hasta situarse al final y realiza la inserción. Se tiene en cuenta la circunstancia de que la lista esté vacía, en cuyo caso inserta el nuevo nodo como nodo primero y único.

```
void inserFinal (Nodo** cabeza, Item entrada)
{
  Nodo *ultimo;
  ultimo = *cabeza;

  if (ultimo == NULL)       /* lista vacía */
  {
    *cabeza = crearNodo (entrada);
  }
  else
  {
    for (; ultimo -> siguiente; )   /* termina con ultimo
                                       referenciando al nodo final */
      ultimo = ultimo -> siguiente;
    ultimo -> siguiente = crearNodo (entrada);
  }
}
```

Inserción de un nodo entre dos nodos de la lista

La inserción de un nuevo nodo no siempre se realiza al principio (en cabeza) de la lista o al final; se puede insertar entre dos nodos cualesquiera de la lista. Por ejemplo, se tiene una lista enlazada con los nodos 10, 25, 40 en la que se quiere insertar un nuevo elemento 75 entre el elemento 25 y el elemento 40. La figura 32.8 representa la lista y el nuevo nodo a insertar.

El algoritmo de la nueva operación insertar requiere las siguientes etapas:

1. Asignar el nuevo nodo, con el campo *dato*, apuntado por el puntero `nuevo`.
2. Hacer que el campo enlace `siguiente` del nuevo nodo apunte al nodo que va después de la posición que se desea para el nuevo nodo (o bien a `NULL` si no hay ningún nodo después de la nueva posición).

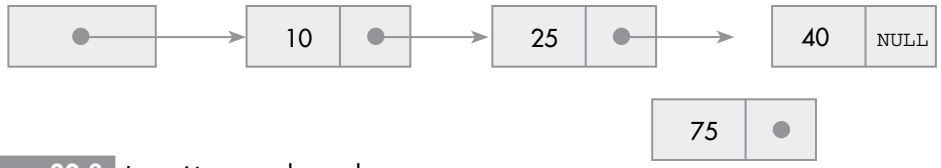
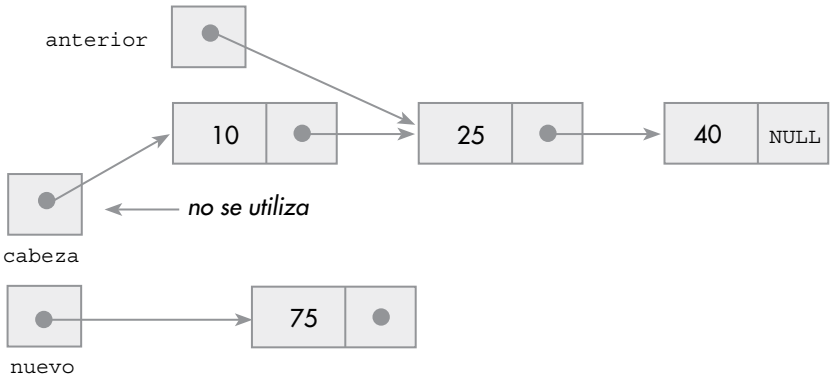


Figura 32.8 Inserción entre dos nodos.

3. En la variable puntero `anterior` tener la dirección del nodo que está antes de la posición deseada para el nuevo nodo. Hacer que `anterior -> siguiente` apunte al nuevo nodo que se acaba de crear.

Etapa 1

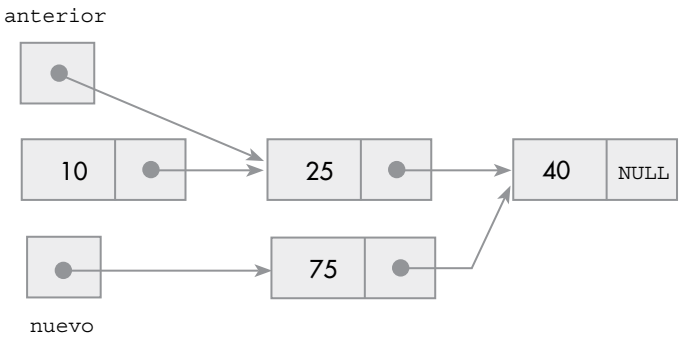
Se crea un nuevo nodo que contiene a 75



Código C

```
nuevo = crearNodo (entrada);
```

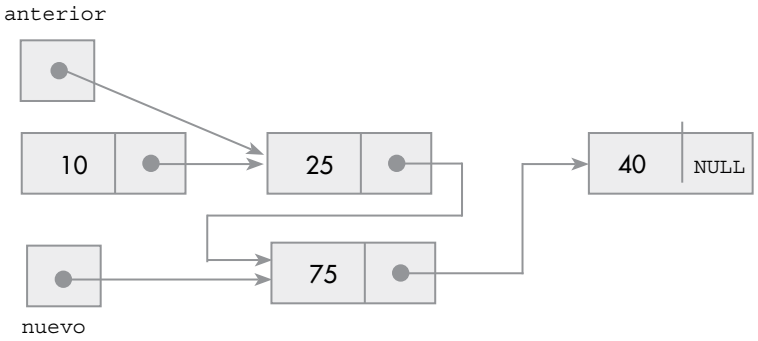
Etapa 2



Código C

```
nuevo -> siguiente = anterior -> siguiente
```

Etapa 3



Después de ejecutar todas las sentencias de las sucesivas etapas, la nueva lista comenzaría en el nodo 10, seguiría 25, 75 y por último 40.

Código C

```
void insertar (Nodo* anterior, Item entrada)
{
  Nodo *nuevo;

  nuevo = crearNodo (entrada);
  nuevo -> siguiente = anterior -> siguiente;
  anterior -> siguiente = nuevo;
}
```

Para llamar a la función `insertar( )`, previamente se ha de buscar la dirección del nodo anterior y asegurarse de que la lista no esté vacía ni que se quiera insertar como primer nodo.

Una versión de la función tiene como argumentos la dirección de inicio de la lista, `cabeza`, el campo dato a partir del cual se inserta y el dato del nuevo nodo. El algoritmo de esta versión de la operación insertar requiere las siguientes etapas:

1. Asignar el nuevo nodo, con el campo *dato*, apuntado por el puntero `nuevo`.
2. Buscar la dirección del nodo, `despues`, que contiene el campo dato a partir del cual se ha de insertar.
3. Hacer que el campo enlace `siguiente` del nuevo nodo apunte al nodo que va a continuación de la posición que se desea para el nuevo nodo (o bien a `NULL` si no hay ningún nodo).
4. Hacer que `despues -> siguiente` apunte al nuevo nodo que se acaba de crear.

La primera comprobación, previa a las etapas a seguir, es que la lista esté vacía. Si es así, se inserta como primer nodo. También se tiene que tener precaución en la búsqueda que se realiza en la etapa 2; ésta puede ser negativa, no se encuentra el dato; entonces no se inserta el nuevo nodo.

Código C

```
void insertar (Nodo** cabeza, Item testigo, Item entrada)
{
  Nodo *nuevo, *despues;

  nuevo = crearNodo (entrada);
  if (*cabeza == NULL)
    *cabeza = nuevo;
  else
  {
    int esta = 0;
    despues = *cabeza;  /* etapa de búsqueda */
    while ( (despues != NULL) && !esta)
    {
      if (despues -> dato != testigo)
        despues = despues -> siguiente;
      else
        esta = 1;
    }

    if (esta)
    {
      nuevo -> siguiente = despues -> siguiente;
      despues -> siguiente = nuevo;
    }
  }
}
```

Búsqueda de un elemento de una lista

El algoritmo que sirva para localizar un elemento en una lista enlazada puede devolver un puntero a ese elemento, o bien, un valor lógico que indique su existencia.

La función `localizar ( )` utiliza una variable puntero denominada `indice` que va recorriendo la lista nodo a nodo. Mediante un bucle, `indice` apunta a los nodos de la lista de modo que si se encuentra el nodo buscado devuelve un puntero al nodo con la sentencia de retorno (`return`); en el caso de no encontrarse el nodo buscado, la función debe devolver `NULL` (`return NULL`). El nodo que se busca es el que tiene un campo dato que coincide con una clave.

Código C

```
Nodo* localizar (Nodo* cabeza, Item destino)
/* cabeza: puntero de cabeza de una lista enlazada.
   destino: dato que se busca en la lista.
*/
{
  Nodo *indice;

  for (indice = cabeza; indice != NULL; indice = indice -> siguiente)
    if (destino == indice -> dato)
      return indice;
  return NULL;
}
```

A tener en cuenta

La operación de búsqueda puede tener diferentes enfoques: búsqueda de la posición del primer nodo que contiene un *dato*, búsqueda de la posición de un nodo según el número de orden, determinación si existe o no un nodo con un campo dato determinado.

Ejemplo 32.4

En este ejemplo se escribe una función para encontrar la dirección de un nodo dado el orden que ocupa en una lista enlazada.

El nodo o elemento se especifica por su número de orden en la lista; para ello se considera posición 1 la correspondiente al nodo de cabeza; posición 2 la correspondiente al siguiente nodo, y así sucesivamente.

El algoritmo de búsqueda del elemento comienza con el recorrido de la lista mediante un puntero `indice` que comienza apuntando al nodo cabeza de la lista. Un bucle mueve el `indice` hacia adelante el número correcto de sitios (lugares). A cada iteración del bucle se mueve el puntero `indice` un nodo hacia adelante. El bucle termina cuando se alcanza la posición deseada e `indice` apunta al nodo correcto. El bucle también puede terminar si `indice` apunta a `NULL`, lo que indicará que la posición solicitada era más grande que el número de nodos de la lista.

```
Nodo* buscarPosicion (Nodo *cabeza, size_t posicion)
/* El programa que llame a esta función ha de incluir
   biblioteca stdlib.h (para implementar tipo size_t)
*/

{
  Nodo *indice;
  size_t i;

  if (posicion < 1)           /* posición ha de ser mayor que 1 */
    return NULL;

  indice = cabeza;
  for (i = 1 ; (i < posición) && (indice != NULL) ; i++)
    indice = indice -> siguiente;

  return indice;
}
```

Borrado de un nodo en una lista

La operación de *eliminar un nodo de una lista enlazada* implica enlazar el nodo anterior con el nodo siguiente al que se desea eliminar y liberar la memoria que ocupa.

El algoritmo para eliminar un nodo que contiene un dato se puede expresar en estos pasos o etapas:

1. *Búsqueda del nodo* que contiene el dato. Se ha de tener la dirección del nodo a eliminar y la dirección del anterior.
2. El puntero `siguiente` del nodo anterior ha de apuntar al `siguiente` del nodo a eliminar.
3. En caso de que el nodo a eliminar sea el primero, `cabeza`, se modifica `cabeza` para que tenga la dirección del nodo `siguiente`.
4. Por último, se libera la memoria ocupada por el nodo.

A continuación se escribe una función que recibe la dirección de inicio, `cabeza`, de la lista y el dato del nodo que se quiere borrar.

Código C

```
void eliminar (Nodo** cabeza, Item entrada)
{
  Nodo* actual, *anterior;
  int encontrado = 0;

  actual = *cabeza; anterior = NULL;
  /* búsqueda del nodo y del anterior */
  while ( (actual!=NULL) && (!encontrado) )
  {
    encontrado = (actual -> dato == entrada);
    if (!encontrado)
    {
      anterior = actual;
      actual = actual -> siguiente;
    }
  }
  /* enlace de nodo anterior con siguiente */
  if (actual != NULL)
  {
     /* distingue entre que el nodo sea el cabecera o del
        resto de la lista */
    if (actual == *cabeza)
      *cabeza = actual -> siguiente;
    else
      anterior -> siguiente = actual -> siguiente;

    free (actual);
  }
}
```

Ejercicio 32.2

Se crea una lista enlazada de números enteros ordenada. La lista va estar organizada de tal forma que el nodo cabecera tenga el menor elemento, y así en orden creciente los demás nodos. Una vez creada la lista, se puede recorrer para escribir los datos por pantalla; también se ha de poder eliminar un nodo con un dato determinado.

Inicialmente la lista se crea con el primer valor. El segundo elemento se ha de insertar antes del primero o después, dependiendo de que sea menor o mayor. Así en general, para insertar un nuevo elemento, primero se busca la posición de inserción en la lista actual, que en todo momento está ordenada, del nodo a partir del cual se ha de enlazar el nuevo nodo para que la lista siga ordenada. Las

diversas formas de realizar la inserción de un nodo en una lista estudiadas en apartados anteriores no se ajustan exactamente a la funcionalidad requerida; por ello se escribe la función `insertaOrden( )` para añadir los nuevos elementos en el orden requerido. Los datos con los que se va a formar la lista se generan aleatoriamente.

La función `recorrer( )` avanza por cada uno de los nodos de la lista con la finalidad de escribir el campo dato. Para eliminar nodos, se invoca a la función `eliminar( )` escrita antes con una pequeña modificación, escribir un literal en el caso de que el nodo no esté en la lista.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MX 101
typedef int Item;
typedef struct Elemento
{
  Item dato;
  struct Elemento* siguiente;
} Nodo;

void insertaOrden (Nodo** cabeza, Item entrada);
Nodo* crearNodo (Item x);
void recorrer (Nodo* cabeza);
void eliminar (Nodo** cabeza, Item entrada);

void main ( )
{
  Item d;
  Nodo* cabeza;

  cabeza = NULL;                        /* lista vacía */
  randomize ( );
  /* bucle termina cuando se genera el número aleatorio 0 */
  for (d = random (MX); d; )
  {
    insertaOrden (&cabeza,d);
    d = random (MX);
  }

  recorrer (cabeza);

  printf ("\n Elemento a eliminar: ");
  scanf ("%d", &d);
  eliminar (&cabeza,d);
  recorrer (cabeza);
}

void insertaOrden (Nodo** cabeza, Item entrada)
{
  Nodo *nuevo;

  nuevo = crearNodo (entrada);

  if (*cabeza == NULL)
    *cabeza = nuevo;
  else if (entrada < (*cabeza) -> dato)              /* primer nodo */
  {
    nuevo -> siguiente = *cabeza;
    *cabeza = nuevo;
  }
  else         /* búsqueda del nodo anterior a partir del que se debe insertar */
  {
    Nodo* anterior, *p;
    anterior = p = *cabeza;
```

```
    while ( (p -> siguiente != NULL) && (entrada > p -> dato) )
    {
      anterior = p;
      p = p -> siguiente;
    }

    if (entrada > p -> dato)                /* inserta por el final */
      anterior = p;

        /* Se procede al enlace del nuevo nodo */
    nuevo -> siguiente = anterior -> siguiente;
    anterior -> siguiente = nuevo;
  }
}

Nodo* crearNodo (Item x)
{
  Nodo *a ;
  a = (Nodo*)malloc (sizeof(Nodo) );
  a -> dato = x;
  a -> siguiente = NULL;
  return a;
}

void recorrer (Nodo* cabeza)
{
  int k;
  printf ("\n\t\t Lista Ordenada \n");
  for (k = 0; cabeza; cabeza = cabeza -> siguiente)
  {
    printf ("%d ",cabeza -> dato);
    k++;
    printf ("%c",(k%15 ?' ':'\n') );
  }
  printf ("\n\n");
}

void eliminar (Nodo** cabeza, Item entrada)
{
  Nodo* actual, *anterior;
  int encontrado = 0;

  actual = *cabeza; anterior = NULL;
  while ( (actual!=NULL) && (!encontrado) )
  {
    encontrado = (actual -> dato == entrada);
    if (!encontrado)
    {
      anterior = actual;
      actual = actual -> siguiente;
    }
  }
  if (actual != NULL)
  {
    if (actual == *cabeza)
      *cabeza = actual -> siguiente;
    else
      anterior -> siguiente = actual -> siguiente;
    free (actual);
  }
  else
    puts ("Nodo no eliminado, elemento no esta en la lista ");
}
```

Concepto de pila

Una **pila** (*stack*) es una colección ordenada de elementos a los que sólo se puede acceder por un único lugar o extremo de la pila. Los elementos de la pila se añaden o quitan (borran) de la misma sólo por la parte superior (**cima**) de la pila. Éste es el caso de una pila de platos, una pila de libros, la organización de la *memoria libre*, etcétera.

A recordar

Una pila es una estructura de datos de entradas ordenadas tales que sólo se pueden introducir y eliminar por un extremo, llamado **cima**.

Cuando se dice que la pila está ordenada, lo que se quiere decir es que hay un elemento al que se puede acceder primero (el que está encima de la pila), otro elemento al que se puede acceder en segundo lugar (justo el elemento que está debajo de la cima), un tercero, etc. No se requiere que las entradas se puedan comparar utilizando el operador "menor que" (<) y pueden ser de cualquier tipo.

Las entradas de la pila deben ser eliminadas en el orden inverso al que se situaron en la misma. Por ejemplo, se puede crear una pila de libros, situando primero un diccionario, encima de él una enciclopedia y encima de ambos una novela de modo que la pila tendrá la novela en la parte superior.

Figura 32.9 Pila de libros.

Cuando se quitan los libros de la pila, primero debe quitarse la novela, luego la enciclopedia y por último el diccionario.

Debido a su propiedad específica "último en entrar, primero en salir" se conoce a las pilas como estructura de datos **LIFO** (*last-in, first-out*)

Las operaciones usuales en la pila son *insertar* y *quitar*. La operación **insertar** o **poner** (`push`) añade un elemento en la cima de la pila y la operación **quitar** (`pop`) elimina o saca un elemento de la pila. La figura 32.10 muestra una secuencia de operaciones *insertar* y *quitar*.

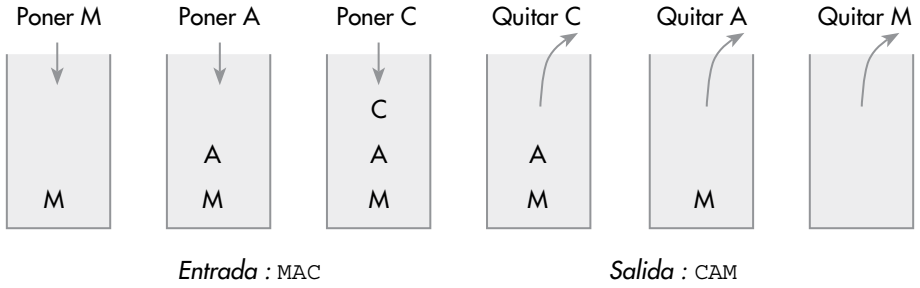


Figura 32.10 Poner y quitar elementos de la pila.

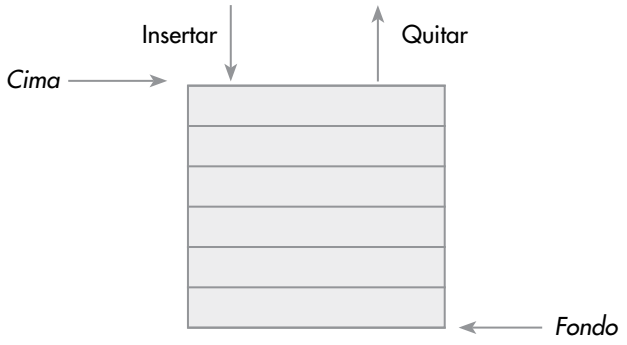


Figura 32.11 Operaciones básicas de una pila.

Una pila puede estar *vacía* (no tiene elementos) o *llena* (en el caso de tener tamaño fijo, si no caben más elementos en la pila). Si un programa intenta sacar un elemento de una pila vacía, se producirá un error, debido a que esa operación es imposible; esta situación se denomina **desbordamiento negativo** (*underflow*). Por el contrario, si un programa intenta poner un elemento en una pila llena se produce un error llamado **desbordamiento** (*overflow*) o *rebosamiento*. Para evitar estas situaciones se diseñan funciones que comprueban si la pila está llena o vacía.

Operaciones en una pila

Las operaciones que sirven para definir una pila y poder manipular su contenido son las siguientes:

CrearPila	Inicia
Insertar (push)	Pone un dato en la pila
Quitar (pop)	Retira (saca) un dato de la pila
Pila vacía	Comprueba si la pila no tiene elementos
Pila llena	Comprueba si la pila está llena de elementos
Limpiar pila	Quita todos sus elementos y deja la pila vacía
Cima	Obtiene el elemento cima de la pila
Tamaño de la pila	Número de elementos máximo que puede contener la pila

El tipo pila implementado con arreglos (*arrays*)

Una pila se puede implementar mediante arreglos (*arrays*) o mediante listas enlazadas. Una implementación estática se realiza utilizando un array de tamaño fijo y una implementación dinámica mediante una lista enlazada.

El tipo pila implementado con arreglos incluye una lista (arreglo) y un índice a la cima de la pila; además una constante con el máximo número de elementos. Al utilizar un arreglo para contener los elementos de la pila es necesario tener en cuenta que el tamaño de la pila no puede exceder el número de elementos del arreglo, y la condición *pila llena* será significativa para el diseño.

El método usual de introducir elementos en una pila es definir el *fondo* de la pila en la posición −1 y sin ningún elemento en su interior, es decir, definir una *pila vacía*; a continuación, se van introduciendo elementos en la pila (el arreglo) de modo que el primer elemento añadido se introduce en una pila vacía y en la posición 0, el segundo elemento en la posición 1, el siguiente en la posición 2 y así sucesivamente. Con estas operaciones el índice que apunta a la cima de la pila se va incrementando en 1 cada vez que se añade un nuevo elemento. Los algoritmos de poner "insertar" (*push*) y quitar "sacar" (*pop*) datos de la pila:

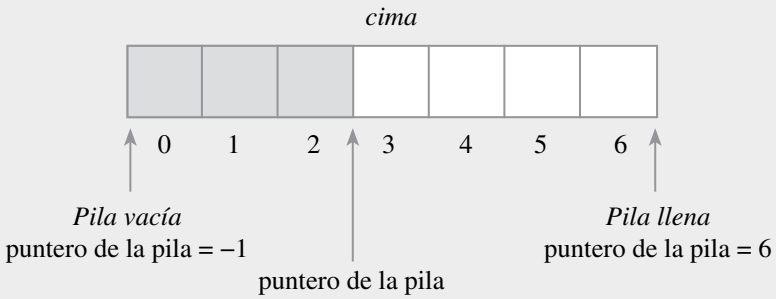
```
Insertar (push)
1. Verificar si la pila no está llena.
2. Incrementar en 1 el puntero índice de la pila.
3. Almacenar elemento en la posición del puntero de la pila.
```

```
Quitar (pop)
1. Si la pila no está vacía.
2. Leer el elemento de la posición del puntero de la pila.
3. Decrementar en 1 el puntero de la pila.
```

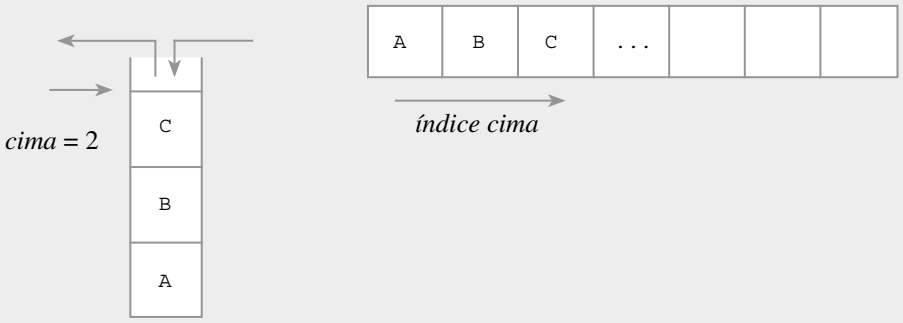
En el caso de que el arreglo que define la pila tenga `TAMPILA` elementos, las posiciones del arreglo, es decir el índice o puntero de la pila, estarán comprendidas en el rango `0` a `TAMPILA-1` elementos, de modo que *en una pila llena* el puntero de la pila apunta a `TAMPILA-1` y *en una pila vacía* el puntero de la pila apunta a −1, ya que 0 será el índice del primer elemento.

Ejemplo 32.5

Una pila de 7 elementos se puede representar gráficamente así:



Si se almacenan los datos `A`, `B`, `C`,... en la pila se puede representar gráficamente de alguna de estas formas



Declaración del tipo pila

Para que la declaración sea lo más abstracta posible, en un archivo *.h* se declara el tipo de los datos que contiene la pila y los prototipos de las funciones que representan las operaciones del *tipo abstracto pila*:

```
/* archivo pilaarray.h */

#include <stdio.h>
#include <stdlib.h>

typedef int TipoDato; /* Tipo de los elementos de la pila */

#define TAMPILA 100
typedef struct
{
  TipoDato listaPila[TAMPILA];
  int cima;
} Pila;

  /* Operaciones sobre la pila */

void crearPila (Pila* pila);
void insertar (Pila* pila, TipoDato elemento);
TipoDato quitar (Pila* pila);
void limpiarPila (Pila* pila);

  /* Operación de acceso a pila */

TipoDato cima (Pila pila);

  /* Verificación estado de la pila */

int pilaVacia (Pila pila);
int pilaLlena (Pila pila);
```

El tipo de los datos que contiene la pila deben declararse como `TipoDato`. Así, si se quiere una pila de números enteros:

```
typedef int TipoDato;
```

Si la pila fuera de números complejos:

```
typedef struct
{
  float x, y;
} complejo

typedef complejo TipoDato;
```

Implementación de las operaciones sobre pilas

Las operaciones del tipo *Pila* definidas en la especificación se implementan en el archivo `pilaarray.c`, para después formar un proyecto con otros módulos y la función principal. La codificación de las funciones exige incluir el archivo donde se encuentra la representación de los elementos y los prototipos.

Archivo pilaarray.c

```
typedef int TipoDato;       /* tipo de los elementos de la pila */
#include "pilaarray.h"
```

`crearPila` inicializa una pila sin elementos, vacía, pone el índice `cima` a –1.

```
void crearPila (Pila* pila)
{
  pila -> cima = -1;
}
```

La operación `insertar` un elemento incrementa el puntero de la pila (`cima`) en 1 y asigna el nuevo elemento a la lista de la pila. Cualquier intento de añadir un elemento en una pila llena produce un mensaje de error "Desbordamiento pila".

```
/* poner un elemento en la pila */

void insertar (Pila* pila, TipoDato elemento)
{
  /* si la pila está llena, termina el programa */
  if (pilaLlena (*pila) )
  {
    puts ("Desbordamiento pila");
    exit (1);
  }
  pila -> cima++;
  pila -> listaPila[pila->cima] = elemento;
}
```

La operación `quitar` elimina un elemento de la pila; copia el primer valor de la cima de la pila en una variable auxiliar, `tem`, y a continuación decrementa el puntero de la pila en 1. El objetivo de la operación es retirar el elemento `cima`; no obstante, la función devuelve la variable `tem` para que pueda ser utilizado. Al eliminar un elemento en una pila vacía se debe producir un mensaje de error y terminar.

Figura 32.12 Operación de extraer un elemento de la pila.

```
/* Quitar un elemento de la pila */

TipoDato quitar (Pila* pila)
{
  TipoDato tem;
    /* si la pila está vacía, termina el programa */
  if (pilaVacia (*pila) )
  {
    puts (" Se intenta sacar un elemento en pila vacía");
    exit (1);
  }

  tem = pila -> listaPila[pila->cima];
  pila -> cima--;
  return tem;
}
```

La operación `cima` devuelve el elemento que se encuentra en la cima de la pila, no se modifica la pila.

```
TipoDato cima (Pila pila)
{
  if (pilaVacia (pila) )
  {
    puts (" Error de ejecución, pila vacía");
    exit (1);
  }
  return pila.listaPila[pila.cima];
}
```

La función `pilaVacia` comprueba si la pila no tiene elementos; para que eso ocurra, la cima de la pila debe ser −1.

```
int pilaVacia (Pila pila)
{
  return pila.cima == -1;
}
```

La función `pilaLlena` comprueba si la pila está llena, según el tamaño del arreglo donde se guardan sus elementos.

```
int pilaLlena (Pila pila)
{
  return pila.cima == TAMPILA - 1;
}
```

Por último, la operación `limpiarPila` vacía íntegramente la pila, para lo que pone `cima` al valor que se corresponde con pila vacía (−1)

```
void limpiarPila (Pila* pila)
{
  pila -> cima = -1;
}
```

El tipo pila implementado como una lista enlazada

La realización dinámica de una pila se hace almacenando los elementos como nodos de una lista enlazada, con la particularidad de que siempre que se quiera insertar o poner (empujar) un elemento se hace por el mismo extremo que se extrae.

> ***A tener en cuenta***
> Una pila realizada con una lista enlazada crece y decrece dinámicamente. En tiempo de ejecución se reserva memoria según se ponen elementos en la pila y se libera memoria según se extraen elementos de la pila.

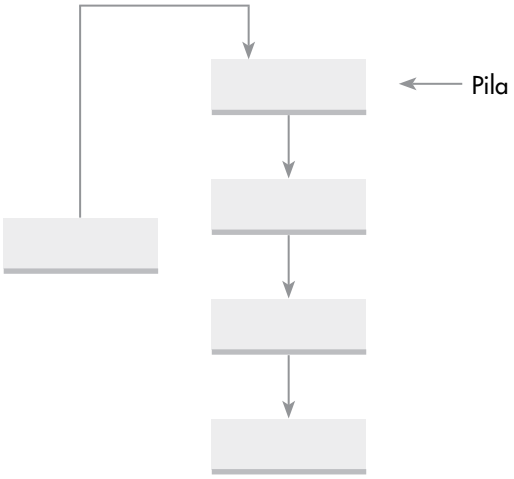


Figura 32.13 Representación de una pila con una lista enlazada.

Tipos de datos para una pila con listas enlazadas

Los elementos de la pila van a ser los nodos de la lista, con un campo para guardar el elemento y otro de enlace; la base para todo ello es la utilización de variables puntero.

Las operaciones del tipo pila implementada con listas son las mismas que si la pila se implementa con arrays, salvo la operación que controla si la pila está llena, *pilaLlena*, que con listas enlazadas no se produce al poder crecer indefinidamente, con el único límite de la memoria. También se añade la operación que elimina el elemento cabeza, *suprimir*, que se diferencia de *quitar* en que no devuelve el elemento extraído.

En el archivo `pila.h` se declaran los tipos de datos y los prototipos de las funciones que representan las operaciones.

```
/* archivo pila.h */

#include <stdlib.h>

typedef struct nodo
{
  TipoDato elemento;
  struct nodo* siguiente;
} Nodo;

  /* prototipo de las operaciones */

void crearPila (Nodo** pila);
void insertar (Nodo** pila, TipoDato elemento);
void suprimir (Nodo** pila);
TipoDato quitar (Nodo** pila);
void limpiarPila (Nodo** pila);
TipoDato cima (Nodo* pila);
int pilaVacia (Nodo* pila);
```

Antes de incluir el archivo `pila.h` debe declararse el `TipoDato` según el tipo de los elementos que va a almacenar la pila. Así, si se quiere una pila cuyos elementos sean el par <*palabra, número de línea*>:

```
typedef struct
{
  char palabra[81];
  int numLinea;
} elemento
typedef elemento TipoDato;
```

Implementación de las operaciones de pilas con listas enlazadas

Las operaciones de la pila cuya declaración se encuentran en el archivo `pila.h`, se implementan en el archivo `pila.c` para después formar un proyecto con otros módulos y la función principal.

La codificación que a continuación se escribe es para una pila de números reales; al estar la pila representada por una lista enlazada, esta codificación es muy similar a la de funciones de listas enlazadas.

Archivo pila.c

```
typedef double TipoDato;    /* tipo de los elementos de la pila */
#include "pila.h"
```

Creación de una pila sin elementos, vacía:

```
void crearPila (Nodo** pila)
{
  *pila = NULL;
}
```

Verificación del estado de la pila:

```
int pilaVacia (Nodo* pila)
{
  return pila == NULL;
}
```

Poner un elemento en la pila. Se crea un nodo con el elemento que se pone en la pila y se enlaza por la cima de la pila.

```
void insertar (Nodo** pila, TipoDato elemento)
{
  Nodo* nuevo;
  nuevo = (Nodo*)malloc (sizeof (Nodo) );   /* crea un nuevo nodo */
  nuevo -> elemento = elemento;
  nuevo -> siguiente = *pila;
  (*pila) = nuevo;
}
```

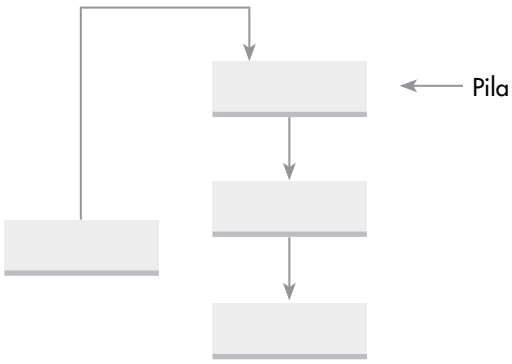


Figura 32.14 Operación de añadir un elemento a la pila.

Eliminación del elemento cima. Con esta operación es eliminado y liberado el elemento cabeza, modificando la pila.

```
void suprimir (Nodo** pila)
{
  if (!pilaVacia (*pila) )
  {
    Nodo* f;
    f = *pila;
    (*pila) = (*pila) -> siguiente;
    free (f);
  }
}
```

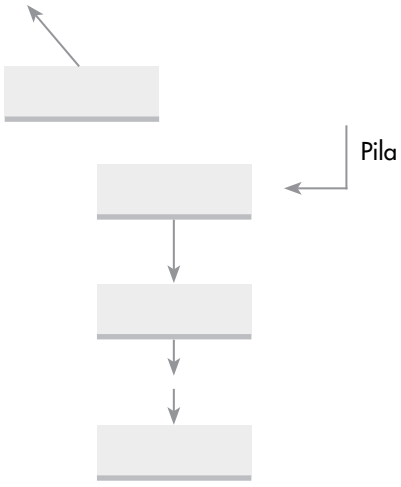


Figura 32.15 Operación de suprimir la cima de la pila.

Obtención del elemento cabeza o cima de la pila, sin modificar la pila:

```
TipoDato cima (Nodo* pila)
{
  if (pilaVacia (pila) )
  {
    puts (" Error de ejecución, pila vacía");
    exit (1);
  }
  return pila -> elemento;
}
```

Extracción y obtención del elemento cima. Esta operación no sólo elimina y libera el elemento cima sino que además devuelve dicho elemento. Se considera un error invocar esta operación estando la pila vacía.

```
TipoDato quitar (Nodo** pila)
{
  TipoDato tem;
  Nodo* q;
  if (pilaVacia (*pila) )
  {
    puts (" Se intenta sacar un elemento en pila vacía");
    exit (1);
  }
  tem = (*pila) -> elemento;
  q = *pila;
  (*pila) = (*pila) -> siguiente;
  free (q);
  return tem;
}
```

Vaciado de la pila. Libera todos los nodos de que consta la pila. Se basa en la operación suprimir.

```
void limpiarPila (Nodo** pila)
{
  while (!pilaVacia (*pila) )
  {
    suprimir (pila);
  }
}
```

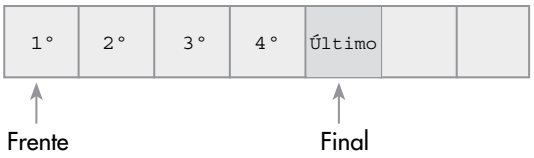


Figura 32.16 Una cola.

Concepto de cola

A recordar

Una cola es una estructura de datos cuyos elementos mantienen un cierto orden, tal que sólo se pueden añadir elementos por un extremo, **final** de la cola, y eliminar o extraer por el otro extremo, llamado **frente**.

Una **cola** es una estructura de datos que almacena elementos en una lista y permite acceder a los datos por uno de los dos extremos de la lista (figura 32.16). Un elemento se inserta en la cola (parte final) de la lista y se suprime o elimina por la frente (parte inicial, frente) de la lista. Las aplicaciones utilizan una cola para almacenar elementos en su orden de aparición o concurrencia.

Los elementos se eliminan de la cola en el mismo orden en que se almacenan; por esa razón una cola es una estructura de tipo **FIFO** (*first-in/first-out*, *primero en entrar/primero en salir* o bien *primero en llegar/primero en ser servido*). El servicio de atención a clientes en un almacén es un ejemplo típico de cola.

Las operaciones típicas usuales que se utilizan en las colas son las siguientes:

Crearcola	Inicia la cola como vacía
Insertar	Añade un dato por el final de la cola
Quitar	Retira (extrae) el elemento frente de la cola
Cola vacía	Comprobar si la cola no tiene elementos
Cola llena	Comprobar si la cola está llena de elementos
Frente	Obtiene el elemento frente o primero de la cola
Tamaño de la cola	Número de elementos máximo que puede contener la cola

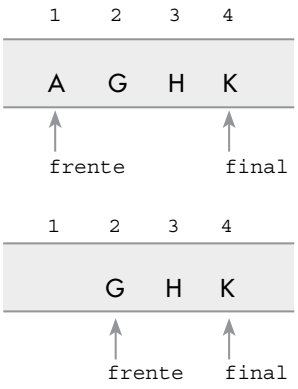
La forma que los lenguajes tienen para representar el tipo `cola` depende de dónde se almacenen los elementos, en un arreglo o en una lista dinámica. La utilización de arreglos tiene el problema de que la cola no puede crecer indefinidamente, está limitada por el tamaño del arreglo, como contrapartida, el acceso a los extremos es muy eficiente. Utilizar una lista dinámica permite que el número de nodos se ajuste al de elementos de la cola; por el contrario, cada nodo necesita memoria extra para el enlace y también está el límite de memoria de la pila de la computadora.

Tipo cola implementado con arreglos (*arrays*)

Se utiliza un arreglo para almacenar los elementos de la cola, y dos marcadores o apuntadores para mantener las posiciones `frente` y `final` de la cola; es decir, un marcador apuntando a la posición de la cabeza de la cola y el otro al primer espacio vacío que sigue al final de la cola. Cuando un elemento se añade a la cola, se verifica si el marcador `final` apunta a una posición válida, entonces se añade el elemento a la cola y se incrementa el marcador `final` en 1. Cuando un elemento se elimina de la cola, se hace una prueba para ver si la cola está vacía y, si no es así, se recupera el elemento de la posición apuntada por el marcador (puntero) de cabeza y éste se incrementa en 1.

La operación de añadir un elemento a la cola comienza a partir de la posición `final` 0; cada que se añade un nuevo elemento se incrementa `final` en 1. La retirada de un elemento se hace por el `frente`; cada que sale un elemento avanza `frente` una posición. En la figura 32.17 se puede observar cómo avanza `frente` al retirar un elemento.

El avance lineal de `frente` y `final` tiene un grave problema, deja *huecos* por la *izquierda del array*. Puede ocurrir que `final` alcance el índice más alto del arreglo, sin que puedan insertar nuevos elementos y sin embargo haber posiciones libres a la izquierda de `frente`.



posición de `frente` y `final` después de extraer.

Figura 32.17 Una cola representada en un arreglo (*array*).

Una alternativa a esto es mantener fijo el `frente` de la cola al comienzo del arreglo. Esto supone mover todos los elementos de la cola una posición cada vez que queramos retirar un elemento. Estos problemas quedan resueltos considerando el arreglo como *circular*.

El tipo cola implementado con arreglos (*arrays*) circulares

La forma más eficiente de almacenar una cola en un arreglo es considerar que el arreglo tiene unido el extremo final de la cola con su extremo cabeza. Tal arreglo se denomina *arreglo circular* y permite utilizar el arreglo completo para guardar los elementos de la cola, sin dejar posiciones libres a las que no se puede acceder; el arreglo sigue siendo una estructura lineal; es a nivel lógico como el arreglo se considera circular. Un arreglo circular con *n* elementos se visualiza en la figura 32.18.

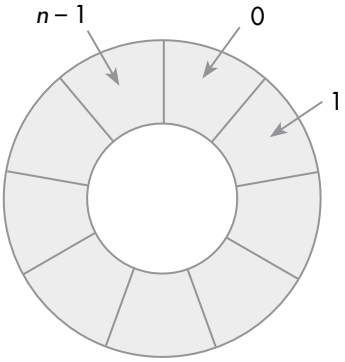
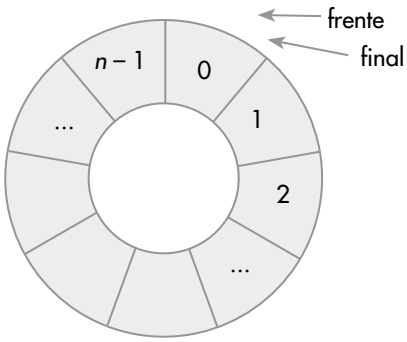


Figura 32.18 Un arreglo (*array*) circular.

El arreglo se almacena de modo natural en la memoria tal como un bloque lineal de *n* elementos. Se necesitan dos marcadores (apuntadores) *frente* y *final* para indicar la posición del elemento cabeza y la posición del final, donde se almacenó el último elemento añadido.

La variable `frente` es siempre la posición del elemento primero de la cola y avanza en el sentido de las agujas del reloj. La variable `final` es la posición de la última inserción; una nueva inserción supone mover `final` circularmente a la derecha y asignar el nuevo elemento.

La simulación del movimiento circular de los índices se realiza utilizando la *teoría de los restos* de tal forma que se generen índices de `0` a `MAXTAMQ-1`:



Mover `final` adelante	=	`(final + 1) % MAXTAMQ`
Mover `frente` adelante	=	`(frente + 1) % MAXTAMQ`

Figura 32.19 Una cola vacía.

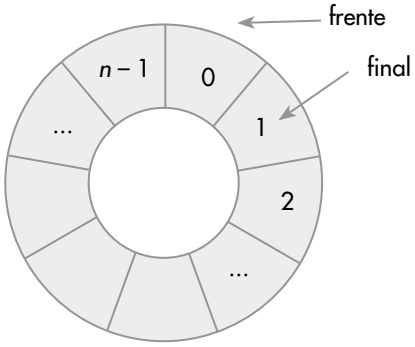


Figura 32.20 Una cola que contiene un elemento

Los algoritmos que formalizan la gestión de colas en un arreglo circular han de incluir las operaciones básicas del tipo *cola*, en concreto, al menos en las siguientes tareas:

- Creación de una cola vacía, de tal forma que `final` apunte a una posición inmediatamente anterior a `frente`: `frente = 0; final = MAXTAMQ-1`.
- Comprobar si una cola está vacía:

  es `frente = siguiente (final) ?`

- Comprobar si una cola está llena. Para distinguir entre la condición de *cola llena* y *cola vacía* se sacrifica una posición del arreglo, de tal forma que la capacidad de la cola va a ser `MAXTAMQ-1`. La condición de cola llena:

  es `frente = siguiente (siguiente (final) ) ?`

- Añadir un elemento a la cola: si la cola no está llena, establecer `final` a la siguiente posición y añadir el elemento en esa posición:

  ```
  final = (final + 1) % MAXTAMQ
  ```

- Eliminación de un elemento de una cola: si la cola no está vacía, suprimirlo de la posición `frente` y establecer `frente` a la siguiente posición:

  ```
  (frente + 1) % MAXTAMQ
  ```

- Obtener el elemento primero de la cola, si la cola no está vacía, sin suprimirlo de la cola.

Codificación del tipo cola con un array (arreglo) circular

A continuación se escribe el código de cada una de las funciones del tipo cola que implementan las operaciones sobre colas.

Siguiente circular

```
int siguiente (int n)
{
  return (n + 1) % MAXTAMQ;
}
```

Crear Cola

Inicializa una *cola vacía.*

```
void crearCola (Cola* cola)
{
  cola -> frente = 0;
  cola -> final = MAXTAMQ-1;
}
```

`Insertar`

El índice `final` apunta al último elemento insertado; éste avanza circularmente.

```
void insertar (Cola* cola, TipoDato entrada)
{
  if (colaLlena (*cola) )
  {
    puts (" desbordamiento cola");
    exit (1);
  }
     /* avance circular al siguiente del final */
  cola -> final = siguiente (cola -> final);
  cola -> listaCola[cola -> final] = entrada;
}
```

`Quitar`

El índice `frente` avanza *circularmente.*

```
TipoDato quitar (Cola* cola)
{
  TipoDato tmp;
  if (colaVacia (*cola) )
  {
    puts (" Extracción en cola vacía ");
    exit (1);
  }
  tmp = cola -> listaCola[cola->frente];
    /* avanza circularmente frente */
  cola -> frente = siguiente (cola -> frente);
  return tmp;
}
```

`Frente`

Obtiene el elemento del frente o primero de la cola, sin modificar la cola.

```
TipoDato frente (Cola cola)
{
  if (colaVacia (cola) )
  {
    puts (" Se requiere frente de una cola vacía ");
    exit (1);
  }
  return cola.listaCola[cola.frente];
}
```

Cola Vacía

```
int colaVacia (Cola cola)
{
  return cola.frente == siguiente (cola.final);
}
```

Cola Llena

Prueba si la cola no puede contener más elementos. En el array queda una posición no ocupada, y así se distingue de la condición de cola vacía.

```
int colaLlena (Cola cola)
{
  return cola.frente == siguiente (siguiente (cola.final) );
}
```

Ejemplo 32.6

Encontrar un número capicúa leído del dispositivo estándar de entrada.

Para encontrar el capicúa buscado se utiliza una cola y una pila. Los dígitos se leen carácter a carácter (un dígito es un carácter del "0" al "9"), almacenándose en una cola y a la vez en una pila. Una vez leído el número se extraen consecutivamente elementos de la cola y de la pila y se comparan por igualdad. De producirse alguna no coincidencia es que el número no es capicúa y en ese caso se vacían las estructuras para solicitar a continuación, otra entrada. El número es capicúa si el proceso de verificación termina al coincidir todos los dígitos en orden inverso; por consiguiente, con la pila y la cola vacía.

¿Por qué utilizar una pila y una cola? Sencillamente por el orden inverso en procesar los elementos; en la pila el *último en entrar es el primero en salir*, en la cola el *primero en entrar es el primero en salir*.

Para que no haya colisión con los nombres de las operaciones, en el tipo pila se añade la letra "*p*" a *insertar* y *quitar*: `insertarp ( ), quitarp ( )`.

```
typedef char TipoDato;
#include "colacircular.h"
#include "pila.h"
#include <stdio.h>

void main ( )
{
  char d;
  Cola cola;
  Pila pila
  int capicua = 0;

  crearCola (&q);
  crearPila (&pila);
  while (!capicua)
  {
    printf (" Número a investigar: ");
    while ( (d = getchar ( ) )!= '\n')
    {
      if ( d < '0' || d > '9')
      {
        puts (" \n Error en el número introducido ");
      }
      else
      {
        insertar (&cola, d);
        insertarp (&pila, d);
      }
```

```
    }
    capicua = 0;
    do {
      capicua = quitar (&cola) == quitarp (&pila);
    } while (capicua && !colaVacia (cola) );
    if (capicua)
      puts (" \n El numero introducido es capicúa ");
    else
    {
      puts (" \n El numero no es capicúa, intente con otro " );
      crearCola (&cola);      /* se inicializan las estructuras */
      crearPila (&pila);
    }
  }
}
```

El tipo cola implementado con una lista enlazada

La realización de una cola con arreglos exige reservar memoria contigua para el máximo de elementos previstos. En muchas ocasiones esto da lugar a que se desaproveche memoria; también puede ocurrir lo contrario, que se llene la cola y/o se pueda seguir con la ejecución del programa. La alternativa a esto está en utilizar memoria que se ajusta en todo momento al número de elementos de la cola, utilizar memoria dinámica mediante una lista enlazada, aún a pesar de tener el inconveniente de la memoria extra utilizada para realizar los encadenamientos entre nodos.

Esta implementación utiliza memoria dinámica, mediante una lista enlazada, que se ajusta en todo momento al número de elementos de la cola. Son necesarios dos punteros para acceder a la lista, `frente` y `final`, que, respectivamente, son los extremos por donde salen los elementos y por donde se insertan.

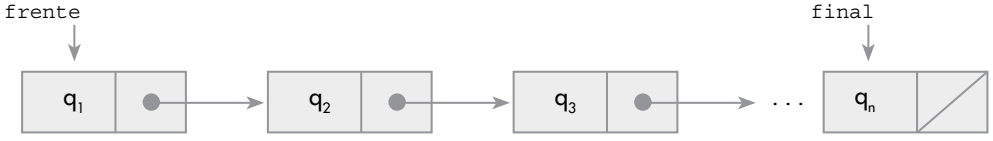


Figura 32.21 Cola con lista enlazada.

La variable puntero `frente` referencia al primer elemento de la cola, el primero en ser retirado de la cola. La otra variable puntero, `final`, referencia al último elemento en ser añadido, que será el último en ser retirado.

Nota de programación
Una estructura dinámica puede crecer y decrecer según las necesidades, según el número de nodos (el límite está en la memoria libre de la computadora). En una estructura dinámica no tiene sentido la operación que verifica un posible *desbordamiento*, esta prueba sí es necesaria en las estructuras de tamaño fijo.

Declaración de tipos de datos y operaciones

La representación de una cola con listas enlazadas maneja dos tipos de datos: el tipo nodo de la lista, y otro tipo para agrupar a las variables `frente` y `final`; este tipo lo denominamos `Cola`.

El archivo `coladinamica.h` contiene la declaración de estos tipos y los prototipos de las funciones que representan a las operaciones básicas de una cola.

```
/* archivo coladinamica.h */

#include <stdlib.h>
struct nodo
{
  TipoDato elemento;
  struct nodo* siguiente;
};
typedef struct nodo Nodo;
typedef struct
{
  Nodo* frente;
  Nodo* final;
}Cola;

/* prototipos de las operaciones */

void crearCola (Cola* cola);
void insertar (Cola* cola,TipoDato entrada);
TipoDato quitar (Cola* cola);
  void borrarCola (Cola* cola); /* libera todos los nodos */
/* acceso a la cola */
TipoDato frente (Cola cola);
/* métodos de verificación del estado de la cola */
int colaVacia (Cola cola);
```

Codificación de las funciones de una cola con listas

La codificación se encuentra en el archivo fuente `coladinamica.c`. En primer lugar hay que declarar el tipo concreto de los elementos de la cola e incluir el archivo `coladinamica.h`.

A continuación se escribe la codificación completa, el tipo de los datos de los elementos de la cola se supone cadena de caracteres, representado por `char*`.

```
/* archivo coladinamica.c */

typedef char* TipoDato;
#include "coladinamica.h"

void crearCola (Cola* cola)
{
  cola -> frente = cola -> final = NULL;
}

Nodo* crearNodo (TipoDato elemento)
{
  Nodo* t;
  t = (Nodo*)malloc (sizeof (Nodo) );
  t -> elemento = elemento;
  t -> siguiente = NULL;
  return t;
}

int colaVacia (Cola cola)
{
  return (cola.frente == NULL);
}

void insertar (Cola* cola, TipoDato entrada)
{
  Nodo* a;
  a = crearNodo (entrada);
  if (colaVacia (*cola) )
  {
    cola -> frente = a;
  }
```

```
  else
  {
    cola -> final -> siguiente = a;
  }
  cola -> final = a;
}

TipoDato quitar (Cola* cola)
{
  TipoDato tmp;

  if (!colaVacia (*cola) )
  {
    Nodo* a;
    a = cola -> frente;
    tmp = cola -> frente -> elemento;
    cola -> frente = cola -> frente -> siguiente;
    free (a);
  }
  else
  {
    puts ("Error cometido al eliminar de una cola vacía");
    exit (-1);
  }
  return tmp;
}

TipoDato frente (Cola cola)
{
  if (colaVacia (cola) )
  {
    puts ("Error: cola vacía ");
    exit (-1);
  }
  return (cola.frente -> elemento);
}

void borrarCola (Cola* cola)
{
  /* Elimina y libera todos los nodos de la cola */
  for (; cola->frente != NULL;)
  {
    Nodo* n;
    n = cola -> frente;
    cola -> frente = cola -> frente -> siguiente;
    free (n);
  }
}
```

Resumen

- Una **lista enlazada** es una estructura de datos dinámica en la que sus componentes están ordenados lógicamente por sus campos de enlace en vez de ordenados físicamente como están en un arreglo. El final de la lista se señala mediante una constante o puntero especial llamado NULL. La gran ventaja de una lista enlazada sobre un array es que la lista enlazada puede crecer y decrecer en tamaño, ajustándose al número de elementos.
- Una **lista simplemente enlazada** contiene sólo un enlace a un sucesor único, a menos que sea el último, en cuyo caso no se enlaza con ningún otro nodo.
- Cuando se inserta un elemento en una lista enlazada, se deben considerar cuatro casos: añadir a una lista vacía, añadir al principio de la lista, añadir en el interior y añadir al final de la lista.

- Para borrar un elemento, primero hay que buscar el nodo que lo contiene y considerar dos casos: borrar el primer nodo y borrar cualquier otro de la lista.
- El *recorrido de una lista* enlazada significa pasar por cada nodo (visitar) y procesarlo. El proceso puede ser escribir su contenido o modificar el campo de datos.
- Una *pila* es una estructura de datos tipo LIFO (*last-in/first-out*, último en entrar primero en salir) en la que los datos (todos del mismo tipo) se añaden y eliminan por el mismo extremo, denominado *cima* de la pila.
- Se definen las siguientes operaciones básicas sobre pilas: `crearPila`, `insertar`, `cima`, `suprimir`, `quitar`, `pilaVacia`, `pilaLlena` y `liberarPila`.
- `crearPila` inicializa la pila como pila vacía. Ésta es la primera operación a ejecutar.
- `insertar` añade un elemento en la cima de la pila. Debe haber espacio en la pila.
- `cima` devuelve el elemento que está en la cima, sin extraerlo.
- `suprimir` extrae de la pila el elemento cima.
- `quitar` extrae de la pila el elemento cima de la pila que es el resultado de la operación.
- `pilaVacia` determina si el estado de la pila es vacía, en su caso devuelve el valor que representa a *true*.
- `pilaLlena` determina si existe espacio en la pila para añadir un nuevo elemento. De no haber espacio devuelve false(0). Esta operación se aplica en la representación de la pila mediante un arreglo.
- `liberarPila` el espacio asignado a la pila se libera, queda disponible.
- Una cola es una lista lineal en la que los datos se insertan por un extremo (*final*) y se extraen por el otro extremo (*frente*). Es una *estructura FIFO* (*first-in/first-out*, primero en entrar primero en salir).
- Las operaciones básicas que se aplican sobre colas: `crearCola`, `colaVacia`, `colaLlena`, `insertar`, `frente`, `retirar`.
- `crearCola` inicializa una cola sin elementos. Es la primera operación a realizar con una cola.
- `colaVacia` determina si una cola tiene o no tiene elementos. Devuelve 1 (`true`) si no tiene elementos.
- `colaLlena` determina si no se pueden almacenar más elementos en una cola. Se aplica esta operación cuando se utiliza un arreglo para guardar los elementos de la cola.
- `insertar` añade un nuevo elemento a la cola, siempre por el extremo final.
- `frente` devuelve el elemento que está en el extremo frente de la cola, sin extraerlo.
- `retirar` extrae el elemento frente de la cola.
- El tipo *Cola* se puede implementar con arrays y con listas enlazadas. La implementación con un arreglo lineal es muy ineficiente; se ha de considerar el arreglo como una estructura circular y aplicar la teoría de los restos para avanzar el frente y el final de la cola.

Ejercicios

32.1. Considerar una cola de nombres representada por un arreglo circular con 6 posiciones, el campo frente con el valor: `frente = 2`. Y los elementos de la cola: `Mar`, `Sella`, `Centurión`. Escribir los elementos de la cola y los campos `frente` y `final` según se realizan estas operaciones:

- Añadir `Gloria` y `Generosa` a la cola.
- Eliminar de la cola.
- Añadir `Positivo`.
- Añadir `Horche` a la cola.
- Eliminar todos los elementos de la cola.

32.2. Escribir una función entera que devuelva el número de nodos de una lista enlazada.

32.3. Escribir una función que elimine el nodo que ocupa la posición i, siendo el nodo cabecera el que ocupa la posición 0.

32.4. ¿Cuál es la salida de este segmento de código, teniendo en cuenta que el tipo de dato de la pila es `int`?

```
Nodo* pila:
int x=4, y;
crearPila (&pila);
insertar (&pila, x);
printf ("\n %d ", cima (pila) );
y = quitar (&pila);
insertar (&pila, 32);
insertar (&pila, quitar (&pila) );
do {
  printf ("\n %d ", quitar (&pila) );
}while (!pilaVacia (pila) );
```

32.5. Utilizando una pila de caracteres, transformar la siguiente expresión a su equivalente expresión en postfija.

```
(x-y)/(z+w) - (z+y)^x
```

32.6. Escribir una función entera que tenga como argumento una lista enlazada de números enteros y devuelva el dato del nodo con mayor valor.

32.7. Se tiene una lista de simple enlace, el campo dato es un registro (estructura) con los campos de un alumno: nombre, edad, sexo. Escribir una función para transformar la lista de tal forma que si el primer nodo es de un alumno de sexo masculino el siguiente sea de sexo femenino.

32.8. Supóngase que se tienen ya codificadas las declaraciones para definir el tipo cola. Escribir una función para crear una copia de una cola determinada. La función tendrá dos argumentos, el primero es la cola origen y el segundo la cola que va a ser la copia. Las operaciones que se han de utilizar serán únicamente las definidas para el tipo cola.

32.9. Escribir la función `mostrarPila ( )` para escribir los elementos de una pila de cadenas de caracteres, utilizando sólo las operaciones básicas y una pila auxiliar.

32.10. Se ha estudiado en el capítulo la realización de una cola mediante un arreglo circular; una variación a esta representación es aquella en la que, además de tener una variable con la posición del elemento frente, dispone de otra variable con la longitud de la cola (número de elementos), `longCola`, y el arreglo considerado circular. Dibujar una cola vacía; añadir a la cola 6 elementos; extraer de la cola tres elementos; añadir elementos hasta que haya *overflow*. En todas las representaciones escribir los valores de `frente` y `longCola`.

32.11. Escribir la implementación de las operaciones del tipo cola para la realización del ejercicio 32.10.

32.12. Se tiene una lista enlazada a la que se accede por el puntero `lista` que referencia al primer nodo. Escribir una función que imprima los nodos de la lista en orden inverso, desde el último nodo al primero; como estructura auxiliar utilizar una pila y sus operaciones.

Problemas

32.1. Escribir un programa o funciones individuales que realicen las siguientes tareas:

- Crear una lista enlazada de números enteros positivos al azar; la inserción se realiza por el último nodo.
- Recorrer la lista para mostrar los elementos por pantalla.
- Eliminar todos los nodos que superen un valor dado.

32.2. Se tiene un archivo de texto de palabras separadas por un blanco o el carácter de fin de línea. Escribir un programa para formar una lista enlazada con las palabras del archivo. Una vez formada la lista se pueden añadir nuevas palabras o borrar alguna de ellas. Al finalizar el programa escribir las palabras de la lista en el archivo.

32.3. Un polinomio se puede representar como una lista enlazada. El primer nodo de la lista representa el primer término del polinomio; el segundo nodo al segundo término del polinomio, y así sucesivamente. Cada nodo tiene como campo dato el coeficiente del término y el exponente. Por ejemplo, el polinomio $3x^4 - 4x^2 + 11$ se representa

Escribir un programa que permita dar entrada a polinomios en x, representándolos con una lista enlazada simple. A continuación obtener una tabla de valores del polinomio para valores de $x = 0.0, 0.5, 1.0, 1.5, \ldots, 5.0$

32.4. Con un archivo de texto se quieren realizar las siguientes acciones: se formará una lista de colas, de tal forma que en cada nodo de la lista esté la dirección de una cola que tiene todas las palabras del archivo que empiezan por una misma letra. Visualizar las palabras del archivo, empezando por la cola que contiene las palabras que comienzan por *a*, a continuación las de la letra *b* y así sucesivamente.

32.5. Según la representación de un polinomio propuesta en el problema 32.3, escribir un programa para realizar las siguientes operaciones:

- Obtener la lista suma de dos polinomios.
- Obtener el polinomio derivado.
- Obtener una lista que sea el producto de dos polinomios.

32.6. En un archivo F están almacenados números enteros arbitrariamente grandes. La disposición es tal que hay un número entero por cada línea de F. Escribir un programa que muestre por pantalla la suma de todos los números enteros. Al resolver el problema habrá que tener en cuenta que al ser enteros grandes no pueden almacenarse en variables numéricas.

Utilizar dos pilas para guardar los dos primeros números enteros, almacenándose dígito a dígito. Al extraer los elementos de la pila salen en orden inverso y por tanto de menos peso a mayor peso, se suman dígito con dígito y el resultado se guarda en una cola, también dígito a dígito. A partir de este primer paso, se obtiene el siguiente número del archivo, se guarda en una pila y a continuación se suma dígito a dígito con el número que se encuentra en la cola; el resultado se guarda en otra cola. El proceso se repite, el nuevo número del archivo se mete en la pila, que se suma con el número actual de la cola.

32.7. Un conjunto es una secuencia de elementos todos del mismo sin duplicidades. Escribir un programa para representar un conjunto de enteros mediante una lista enlazada. El programa debe contemplar las operaciones:

- Cardinal del conjunto.
- Pertenencia de un elemento al conjunto.
- Añadir un elemento al conjunto.
- Escribir en pantalla los elementos del conjunto.

32.8. Con la representación propuesta en el problema 32.7, añadir las operaciones básicas de conjuntos:

- Unión de dos conjuntos.
- Intersección de dos conjuntos.
- Diferencia de dos conjuntos.
- Inclusión de un conjunto en otro.

32.9. Escribir un programa que haciendo uso de una *pila de caracteres*, procese cada uno de los caracteres de una expresión que viene dada en una línea. La finalidad es verificar el equilibrio de paréntesis, llaves y corchetes.

Por ejemplo, la siguiente expresión tiene un número de paréntesis equilibrado:

```
( (a + b) *5) - 7
```

A esta otra expresión le falta un corchete:

```
2* [ (a + b) / 2.5 + x - 7 *y
```

32.10. Escribir un programa en el que dados dos archivos F1, F2 formados por palabras separadas por un blanco o fin de línea, se creen dos conjuntos con las palabras de F1 y F2 respectivamente. Después encontrar las palabras comunes y mostrarlas por pantalla. Para resolver el problema, utilizar la representación y operaciones sobre conjuntos definidas en los problemas 32.7 y 32.8.

32.11. Escribir un programa en el que se manejen un total de $n = 5$ pilas: P_1, P_2, P_3, P_4 y P_5. La entrada de datos será pares de enteros `(i,j)` tal que `1`$\leq$ `abs(i)` $\leq$ `n`. De tal forma que el criterio de selección de pila:

- Si *i* es positivo, debe insertarse el elemento *j* en la pila P_i.
- Si *i* es negativo, debe eliminarse el elemento *j* de la pila P_i.
- Si *i* es cero, fin del proceso de entrada.

Los datos de entrada se introducen por teclado. Cuando termina el proceso, el programa debe escribir el contenido de las *n* pilas en pantalla.

32.12. Un vector disperso es aquel que tiene muchos elementos que son cero. Escribir un programa que permita representar mediante listas enlazadas un vector disperso. Los nodos de la lista son los elementos de la lista distintos de cero; en cada nodo se representa el valor del elemento y el índice (posición del vector). El programa ha de realizar las operaciones: sumar dos vectores de igual dimensión y hallar el producto escalar.