Capítulo 6

Estructuras de Datos Dinámicas

Índice General

6.1. Introducción. 157
6.2. Memoria Dinámica. Concepto de puntero 158
6.2.1. Trabajo con punteros en C. 160
6.2.2. El lenguaje C y los punteros. 163
6.3. Estructuras de Datos Dinámicas. 167
6.3.1. Gestión dinámica de la memoria. 168
6.3.2. Estructuras enlazadas. 170
6.3.3. Ejemplo: Definición de una lista simplemente enlazada. 175
6.4. Glosario. 182
6.5. Resumen y Consideraciones de Estilo. 183
6.6. Bibliografía. 184
6.7. Problemas Propuestos. 184

"Once upon a time, there were 3 pointers. The Papa pointer pointed to a big array. The Mama pointer pointed to a double precision floating-point. And the Baby pointer pointed to a little integer.

One day they all entered a function, and a little lost pointer named Goldilocks entered their house (also known as int main( )*). Goldilocks was hungry, so she called* strcpy( ) *to copy Papa's array into herself. Her calls to* malloc( )*, slowed the main program to a near standstill, but Goldie didn't care. She then went for Mama's double precision floating-point. But before she could get Baby's integer, the three pointers came back, and they were so angry at Goldie's thievery that they caused a run-time error and the whole house caved in."*

Folklore Popular. "Goldilocks and the three pointers".

6.1. Introducción.

El objetivo de este tema es estudiar ciertos tipos de datos, muy importantes y básicos en muchas aplicaciones, que normalmente no están presentes en los lenguajes de programación. Para definirlos y usarlos, ni siquiera se cuenta con una estructura en el lenguaje, tal y como ocurría en el caso de los vectores. Se trata de tipos que el usuario debe definir completamente, tanto en lo que se refiere

a cuál es la estructura de datos que lo soportan, como en lo que se refiere a las operaciones para manejarlos.

Además, dichos tipos se caracterizan porque las estructuras que los soportan son estructuras de datos dinámicas, es decir, estructuras de datos cuyo tamaño varía en tiempo de ejecución y a medida que se realizan operaciones sobre ellas.

Ejemplos de estos tipos de datos son los tipos *pila*, *cola*, *lista*, *árbol* y *grafo*. Son tipos definidos mediante un formalismo conocido como *Tipo Abstracto de Datos*, TAD, en el que se especifica qué operaciones soporta cada tipo y cuáles son los axiomas que permiten verificar si una implementación concreta es o no es correcta.

El estudio completo de qué es un TAD, qué ventajas aporta a la programación y cuáles son las diversas implementaciones que admite, es uno de los objetivos de la asignatura *"Estructuras de Datos y de la Información"*. Por ello, en este tema sólo se estudiará uno de estos tipos, el tipo de datos *lista*.

Pero, para poder implementar correctamente un tipo de datos que se sustenta en una estructura dinámica, es preciso saber realizar la *gestión dinámica de la memoria*, esto es, saber cómo reservar espacio en memoria a medida que la estructura lo requiera, cómo liberarlo a medida que la estructura deje de necesitarlo y cómo manejar la información que se almacenará en la estructura.

Para ello, es preciso familiarizarse con el concepto de *puntero*. Se trata de un concepto algo distinto de los que se han desarrollado hasta ahora; de hecho, supone descender en el nivel de abstracción en el que un lenguaje de programación sitúa al programador: es preciso entender bien cómo se almacena la información, cada uno de los objetos manejados en un programa, en la memoria del computador.

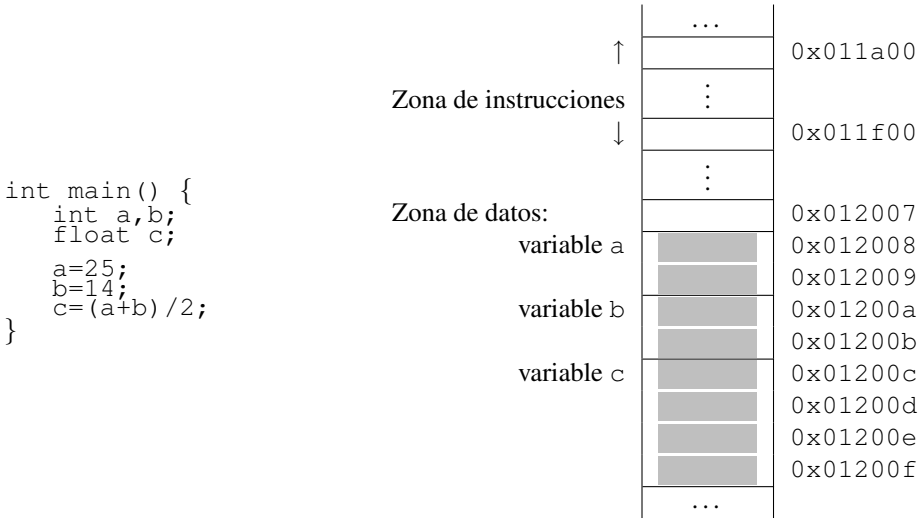
Los punteros permiten al programador trabajar directamente con la memoria. Ello dota de mucha potencia a los lenguajes de programación; por supuesto, a cambio del riesgo que supone dicho trabajo directo sobre la memoria y que puede redundar en la aparición de errores de ejecución "imprevistos".

No todos los lenguajes permiten el manejo de punteros: algunos debido a su antigüedad, algunos para evitar los problemas derivados de su mal uso. De algún modo, esto les resta potencia y flexibilidad.

Históricamente, el primer lenguaje que permitió una manipulación flexible y simple de los punteros fue el lenguaje Pascal. Pero si hay un lenguaje de programación que desarrolle completamente las posibilidades de uso de los punteros, ese es el lenguaje `C`: se puede decir que es un lenguaje orientado a los punteros (a veces, por desgracia, a costa de la legibilidad del código). Casi todos los programas desarrollados en este lenguaje utilizan punteros, bien sea explícita o implícitamente, tal y como se verá a lo largo del tema.

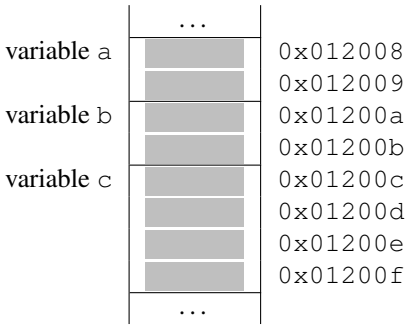
6.2. Memoria Dinámica. Concepto de puntero

Los programas residen en la memoria principal del computador, tanto las instrucciones que lo forman, como los datos con los que trabaja. Una representación gráfica de lo que ocurre cuando comienza la ejecución de un programa, podría ser la siguiente:



Cada palabra de la memoria posee una dirección que la identifica. Internamente, el gestor de la memoria sabrá qué direcciones están ocupadas y cuáles están disponibles para que el usuario pueda ejecutar sus programas. Un programa, una vez compilado y obtenido su código objeto, se almacenará en posiciones consecutivas de memoria, en lo que se denominará *zona de instrucciones* de dicho proceso. También se reserva memoria para almacenar los datos con los que el programa trabaja; a esa zona se le denominará *zona de datos*: cuando se define una variable, se le asigna, dependiendo del tipo, una o varias posiciones de memoria en las que siempre estará almacenado el valor que tenga dicha variable, a lo largo de la ejecución. Una vez finalizado el proceso completo del programa, toda esa memoria, la de datos y la de instrucciones, se libera y la puede utilizar otro programa.

La situación que refleja el ejemplo anterior esquematiza estas ideas[1]: hay una zona de instrucciones (de la dirección `0x011a00` a la dirección `0x011f00`) y una zona de datos en la que se almacenan en posiciones consecutivas los valores de las variables `a`, `b` y `c`. Son de distintos tipos y, en general, el espacio reservado para variables de distinto tipo no será el mismo; así, para `a`, que es una variable entera, se reservan las palabras de memoria `0x012008` y `0x012009`. Para `b`, también entera, otras dos palabras, `0x001200a` y `0x01200b`. La variable `c` es real y se reservan cuatro palabras: `0x01200c`, `0x01200d`, `0x01200e` y `0x01200f`.



La expresión *"la dirección de la variable* `a`*"* se refiere, siguiendo con el mismo ejemplo, a la posición de memoria `0x012008` que es la primera reservada para almacenar el valor de `a`. La dirección de `b` en el mismo ejemplo, sería `0x01200a` y la dirección de `c`, `0x01200c`.

[1] Y es completamente imaginaria, es decir, no se basa en una situación real de un sistema real.

Es muy importante distinguir claramente los conceptos *"la dirección de* `a` *"* y *"el valor de* `a` *"*. El primero se refiere a una dirección de memoria, el segundo al valor que se almacena en la memoria a partir de esa dirección. El primero no cambia durante la ejecución del programa, el segundo cambia cada vez que se realiza una asignación sobre `a`.

Definición 6.1 (Puntero) *Un puntero es un tipo de datos que permite almacenar una dirección de memoria.*

Los punteros permiten acceder directamente a una determinada posición de la memoria y trabajar con ella; normalmente, existen operadores asociados que le permitirán no sólo leer sino también modificar el contenido de la posición de memoria a la que apuntan. Por supuesto, ello implica un gran riesgo: la potencia de que dotan al lenguaje, se puede pagar accediendo a zonas de memoria que pongan en peligro el comportamiento del propio computador (por ejemplo, si se accede a zonas reservadas para datos o instrucciones del sistema operativo, o simplemente variando el contenido de una posición de memoria en la que haya una instrucción del propio programa que maneja al puntero).

6.2.1. Trabajo con punteros en C.

Operador de Dirección.

Los lenguajes que permiten la gestión de memoria dinámica y el uso de punteros, normalmente también permiten conocer la dirección de memoria de una variable.

En el lenguaje `C` esta operación se realiza utilizando el operador `&`, que se puede traducir al lenguaje natural como "la dirección de".

&<nombreVariable>;

Así, `&a` es la dirección de memoria que se ha asignado a la variable `a`. En el contexto del ejemplo anterior, la ejecución de la instrucción

```
printf("%p",&a);
```

mostraría por la salida estándar la dirección de la variable `a`, `0x012008`, y no su valor, `25`. Nótese la diferencia con la instrucción

```
printf("Valor de a: %d, almacenado en la dirección %p.\n",a,&a);
```

que mostraría por pantalla la cadena

```
Valor de a: 25, almacenado en la dirección 0x012008.
```

Declaración de punteros.

Como ocurre con cualquier otra variable de cualquier otro tipo, en `C` los punteros deben declararse. Esta declaración se hace asociando al puntero el tipo de datos al que hará referencia. La sintaxis es:

<tipo_base> *<nombrePuntero>;

donde `nombrePuntero` es el identificador de la variable y el operador `*` indica que se trata de un puntero.

Ya se ha indicado antes el peligro que supone asignar valores a los punteros de forma poco cuidadosa. El operador `&` proporciona una forma segura de asignar un valor a un puntero. Si se hace, por ejemplo, la siguiente declaración,

```
int a, *p;
```

se está definiendo una variable de tipo entero, `a`, y un puntero a entero, `p`. Un posible valor inicial para `p` es la dirección de la variable `a`, para lo que se puede utilizar el operador `&`: la secuencia

```
a=25;
p=&a;
```

almacena el valor `25` en `a` y la dirección de memoria asignada a `a` en `p`.

La dirección de una variable ya existente es un valor que siempre se puede asignar a un puntero, sin riesgo de que se le esté asignando un valor "peligroso". Existe un valor constante, el `0` (valor que se suele asignar mediante la macro del preprocesador `NULL`), que también es un valor válido para asignar a un puntero, si bien su uso suele estar asociado a circunstancias no habituales, ya que de alguna forma se entiende como un "puntero nulo", un puntero que no apunta a ninguna dirección de memoria.

Operador de Indirección.

El lenguaje `C` también permite acceder al contenido de la dirección de memoria almacenada en un puntero. Para ello se utiliza el operador `*`, que se puede traducir al lenguaje natural como "el contenido de lo que está siendo apuntado por el puntero".

***<nombrePuntero>;**

Así, siguiendo con el ejemplo anterior,

```
printf("%d", *p);
```

muestra por la salida estándar el valor `25`, ya que `p` apunta a la variable `a`, a la que se ha asignado el valor `25`.

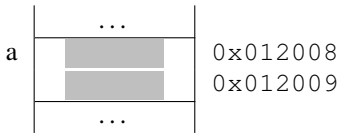
El operador `*` permite que se puedan cambiar los contenidos de una posición de memoria; así, la asignación

```
*p=7;
```

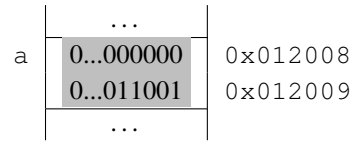
almacena un `7` en la posición de memoria apuntada por el puntero `p`. Es decir, si el valor de `p` es la dirección de memoria asignada a `a`, la asignación `*p=7` es equivalente a la asignación `a=7`: se ha modificado el valor de `a` sin utilizar la propia variable, utilizando un puntero a `a`.

En la figura, se pretende esquematizar el efecto de las tres asignaciones anteriores, si se ejecutan en secuencia:

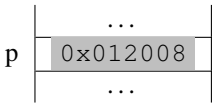
```
int main() {
    int a, *p;

    a=25;  /* (1) */
    p=&a;  /* (2) */
    *p=7;  /* (3) */
}
```



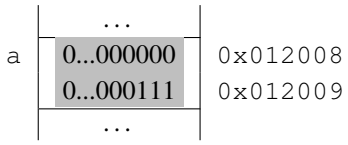
Situación inicial



Después de (1)



Después de (2)



Después de (3)

Expresiones de punteros.

Los punteros son variables, por lo que pueden formar parte de expresiones. Una de las operaciones más habituales de la aritmética de punteros es el incremento (sumar 1 al valor de un puntero).

Para entender bien esta operación, y cuál es su resultado, hay que saber que cuando se incrementa un puntero en una unidad, este incremento no aumenta necesariamente en 1 el valor de dicho puntero, sino que *el incremento será igual al número de palabras de memoria que ocupa el tipo base del puntero*. Los valores más típicos son: `1` si el puntero apunta a un valor de de tipo `char`, `2` ó `4` si es de tipo `int` (dependiendo de la máquina[2]), `4` si es de tipo `float`, `8` si es de tipo `double` ....

Es decir, dadas las siguientes definiciones de punteros:

```
int *pi;
char *pc;
float *pf;
double *pd;
```

y suponiendo que se les ha asignado las direcciones `0x1000`, `0x2000`, `0x3000` y `0x4000`, respectivamente, la ejecución de las siguientes operaciones

```
pi=pi+1;
pc=pc+1;
pf=pf+1;
pd=pd+1;
```

[2]Cuando el microprocesador es de 16 bits, el entero ocupa dos palabras; si es de 32 bits, ocupa cuatro.

almacena los valores `0x1004` (ó `0x1002`) en `pi`, `0x2001` en `pc`, `0x3004` en `pf` y `0x4008` en `pd`.

En general, si a un puntero se le suma un valor `n`, el puntero se incrementara en `n*tam`, donde `tam` es el tamaño en palabras de memoria del tipo de datos al que apunta el puntero.

6.2.2. El lenguaje C y los punteros.

Como se decía en la introducción, el lenguaje `C` hace un uso intensivo de los punteros. De hecho, de entre las utilidades asociadas al uso de punteros en este lenguaje, se van a destacar las siguientes,

- Manejo de vectores, incluidas las cadenas.
- Definir parámetros de E/S. Y, además, que el paso de parámetros –cuando los parámetros son estructuras, como vectores y tuplas– sea más eficiente.
- Manejo de estructuras enlazadas.

lo que lleva a una reflexión: ya se han utilizado punteros en la asignatura, sin que se hiciera de forma explícita, puesto que ya se han manejado vectores y ya se han definido parámetros de Entrada/Salida. En esta subsección se pretende ilustrar cuál ha sido el uso que ya se ha hecho de los punteros, y por ello, se centra en los dos primeros ítems.

Manejo de Vectores.

Cuando se define un vector en el lenguaje `C`, lo que ocurre es que, en realidad, se está definiendo un puntero a la zona de memoria que se reserva para almacenar los datos del vector, siendo esto algo totalmente transparente para quien está realizando el programa.

Una definición de vector como la siguiente

```
int v[100];
```

define un vector, `v`, para almacenar 100 elementos de tipo entero. Lo que hace el compilador es reservar una zona de memoria para dichos elementos y definir un puntero, `v`, que apunta a dicha zona de memoria.

Sabiendo esto, es posible acceder a los elementos del vector utilizando la aritmética de punteros; así, `v` apunta al primer elemento del vector, `v[0]`, `v+1` apunta al segundo (`v[1]`), ... y, en general, `v+i` apunta al elemento `v[i]`. Utilizando el operador `*`, se puede acceder al valor de dichos elementos, ya que se accede al contenido de la posición de memoria apuntada. El siguiente fragmento de código almacena el valor `0` en los 100 elementos del vector,

```
for(i=0;i<100;i=i+1)
  *(v+i)=0;
```

y es equivalente al siguiente:

```
  for(i=0;i<100;i=i+1)
    v[i]=0;
```

¿Quiero esto decir que punteros y vectores son la misma cosa en C? Rotundamente, no. El uso de punteros permite una forma de acceso alternativa a los elementos del vector, por medio de la dirección de memoria en la que están almacenados (igual que permite acceder de forma alternativa a cualquier otra variable). Este es quizás, uno de los puntos más difíciles de entender de C, y puede ser motivo de confusión.

Desde el punto de vista de la asignatura, el hecho de saber cómo implementa internamente C el manejo de los vectores, permite hacer especial hincapié en el tema de *respetar* el rango del vector, sobre el que tanto se insistió en el tema 5: no tiene sentido indexar el elemento −5 de un vector, ni el elemento 100 en un vector de tamaño 50... pero, técnicamente, es posible. ¿Con qué consecuencias? No hay que descartar ninguna: desde un bucle infinito (se accede a una dirección que pertenece a la zona de datos del programa y se está sobreescribiendo una variable cuyo valor es crítico en el control de un bucle) hasta un error en tiempo de ejecución.

Además, tal y como se verá a continuación, conocer el manejo interno de los vectores en este lenuguaje, permite justificar por qué en C los vectores siempre son considerados parámetros de E/S.

Parámetros de E/S.

En la asignatura ya se habían utilizado los operadores * y &. Tal y como se vio en el tema 4, para definir un parámetro formal de E/S en la cabecera de un procedimiento se utiliza el operador * delante del identificador; es decir, cuando el parámetro es de E/S, realmente se trabaja con un puntero a dicho parámetro.

Se puede tomar como ejemplo cualquier procedimiento de los ya diseñados, como el que permite calcular las raíces reales de una ecuación de segundo grado. Este procedimiento tiene tres parámetros de entrada, los coeficientes de la ecuación a,b,c que representan los datos a partir de los cuales se puede obtener la solución, que incluye dos parámetros de salida, x1 y x2, sobre los que se devuelven las raíces de la ecuación y un tercer parámetro de salida, haySol, que indicará si hay soluciones reales o no.

```
void ecSegGra(float a, float b, float c,
                         float *x1, float *x2, boole *haySol) {
  float d;

  d=b*b-4*a*c;

  if (d<0)
    *haySol=FALSO;
  else {
    *haySol=CIERTO;
    d=sqrt(d);
    *x1=(-b+d)/(2*a);
    *x2=(-b-d)/(2*a);
   }
}
```

Para utilizar el procedimiento, en la llamada se utiliza el operador & delante del parámetro;

es decir, en la llamada realmente se utiliza la dirección de memoria de los parámetros actuales, indicando así en dónde debe almacenarse el resultado que se produzca.

Siguiendo con el ejemplo anterior, en el fragmento de código siguiente,

```
float r1,r2;
boole f;

ecSegGra(1,2,-1,&r1,&r2,&f);

if (f)
  printf("Soluciones Reales: %f y %f \n",r1,r2);
else
  printf("No hay solución en el dominio Real\n");
```

se está realizando una llamada a `ecSegGra` para resolver la ecuación $x^2 + 2x - 1 = 0$; para ello, se pasan las direcciones de `r1` y `r2` para que el resultado del cálculo se almacene en ellas. El procedimiento trabaja con los valores almacenados en dichas direcciones; por lo tanto, al final del proceso las soluciones se almacenan en dichas posiciones de memoria. Algo similar ocurre con el tercer parámetro actual, se indica cuál es la dirección de `f`, de tipo `boole`, para que el procedimiento almacene en esa posición el valor `CIERTO` o `FALSO`, dependiendo de si hay o no hay solución.

Este mecanismo se conoce como *"paso de parámetros por referencia"* y se implementa haciendo que el procedimiento trabaje *siempre* utilizando el operador `*` con los parámetros formales de E/S. Es decir, trabaja con el contenido de la dirección de los parámetros actuales. Cualquier modificación que haga el procedimiento en los valores apuntados por los parámetros formales de E/S se traduce en un cambio en los valores de los parámetros actuales (igual que ocurre con la asignación `*p=7`: si `p` apunta a `a`, es equivalente a la asignación `a=7`.)

Paso de parámetros: vectores, tuplas y punteros.

Se ha visto que los vectores son considerados internamente por `C` como punteros a los elementos que forman el vector. De ahí, que siempre sean parámetros de E/S y que su paso a un procedimiento siempre se haga por referencia y, de hecho, el siguiente ejemplo, un procedimiento para calcular el máximo y el mínimo de un vector de enteros,

```
void maxMin(int v[], int N, int *max, int *min) {
  int i;

  *max=v[0];
  *min=v[0];

  for (i=0;i<N;i=i+1)
     if (v[i]>*max)
        *max=v[i];
     else
        if (v[i]<*min)
           *min=v[i];
}
```

podría tener la siguiente cabecera,

```
void maxMin(int *v, int N, int *max, int *min)
```

manteniéndose exactamente igual el resto del código.

Respecto al manejo de punteros a tuplas, hay que comentar un par de cuestiones específicas del lenguaje.

Tal y como se vio en el tema 5, el operador "`.`" permite acceder a los diferentes campos que forman una tupla. Este operador también puede utilizarse cuando se referencia a la tupla por medio de un puntero. Por ejemplo, si se define el tipo `tPersona` por medio de la tupla

```
typedef struct {
  char nombre[80];
  char telefono[15];
  int edad;
} tPersona;
```

y, después, se declaran las siguientes variables

```
tPersona p1,p2;
tPersona *p;
```

se obtiene que `p1` y `p2` son variables del tipo `tPersona` y que `p` es un puntero que permite apuntar a tuplas del tipo `tPersona`.

La instrucción

```
p=&p1;
```

guarda en `p` la dirección donde está almacenada la tupla `p1`. Para acceder a los campos de `p1` utilizando el puntero se puede utilizar el operador "`.`" junto con el operador `*`. Así,

```
(*p).edad
```

accede al campo `edad` de la tupla `p1` (ya que `p` apunta a `p1`).

Esta notación puede resultar un tanto pesada. Tanto es así, que en `C` existe un operador específico, el operador `->`[3], que permite acceder a los campos de una tupla, si se accede a ella utilizando un puntero. Siguiendo con el ejemplo anterior, para acceder al campo `edad` de la tupla que está siendo apuntada por `p`, se puede utilizar la notación ya vista,

```
(*p).edad
```

o la notación

```
p->edad
```

[3] Escrito "−" y ">".

y ambas son totalmente equivalentes.

Al comentar el mecanismo de paso de parámetros de E/S, se hacía mención al mecanismo de *"paso de parámetros por referencia"*. Cuando se realiza el paso de parámetros de entrada, el mecanismo habitual es el *"paso de parámetros por valor"*: en la llamada a una función o procedimiento se realiza una copia del valor del parámetro de entrada e, internamente, se trabaja con él. De ese modo, cualquier operación que se haga en la función o procedimiento que pueda afectar al valor de dicho parámetro no tiene efecto en el entorno en el que se realiza la llamada.

Realizar una copia de toda la información de un tupla puede suponer, si el tamaño de la tupla es grande, una pérdida de eficiencia, sobre todo si se realizan muchas llamadas con esas características.

Para mejorar la eficiencia en el paso de parámetros, especialmente con tuplas, se pueden utilizar punteros, aunque se trate de un parámetro de entrada[4]. De esta forma en la llamada a la función o procedimiento, no se realizará una copia de toda la tupla, sino que se le pasará su dirección. Si se hace esto, es conveniente utilizar la palabra reservada `const` para indicarle al compilador que dicho parámetro no se debe modificar dentro de la función (ya que es un parámetro de entrada).

Como ejemplo, en el siguiente procedimiento, que muestra por la salida estándar los datos de una persona, se ha definido el parámetro `per` como un puntero al tipo `tPersona` (es decir, a una tupla), aunque se trata de un parámetro de entrada.

```
void escribirPersona(const tPersona *per) {
  printf("Nombre: %s\n",per->nombre);
  printf("Telefono: %s\n",per->telefono);
  printf("Edad: %d\n",per->edad);
}
```

En la llamada a este procedimiento, el correspondiente parámetro actual será una tupla del tipo `tPersona` afectada del operador `&`, igual que si el parámetro fuera de E/S. La siguiente instrucción mostraría los datos almacenados en `p1`:

```
  escribirPersona(&p1);
```

6.3. Estructuras de Datos Dinámicas.

Los contenidos del tema, hasta el momento, no parecen dotar al programador de ninguna herramienta especial; es cierto que el uso de punteros, permite acceder a posiciones de memoria y modificarlas, pero es algo que se había realizado hasta ahora utilizando el concepto de variable. El uso interno de los punteros por parte del lenguaje `C` puede ayudar a comprender mejor cómo funcionan ciertos mecanismos, pero esos mecanismos funcionan independientemente de como lo haga el lenguaje y el programador tampoco tiene necesidad de conocer exactamente como se desarrollan para utilizarlos. De hecho, parece incluso que haya una pequeña contradicción entre el nivel de abstracción en el que se discurría hasta ahora (y que se ha defendido como una ventaja para hacer más fácil la tarea de diseñar y programar algoritmos) y el hecho de descender hacia un nivel más cercano a la implementación.

Y, sin embargo, el objetivo del tema es, precisamente, aumentar algo más el nivel de abstracción.

[4]De hecho, es una práctica muy habitual y uno de los motivos de que exista el operador "->".

Es necesario conocer qué es un puntero porque será la herramienta básica para poder crear nuevos tipos de datos: los basados en estructuras de datos dinámicas.

Las estructuras dinámicas son estructuras que no tienen definido un tamaño a priori (como es el caso de los vectores y tuplas, por ejemplo) sino que su tamaño puede variar según las necesidades del programa en tiempo de ejecución.

El objetivo de esta sección es conocer qué son las estructuras dinámicas y la implementación de un tipo basado en una estructura lineal que se denomina *lista*. Antes, se deben presentar las herramientas que permiten desarrollar realmente el potencial de los punteros, puesto que permiten realizar la gestión dinámica de la memoria; es decir, permiten reservar en tiempo de ejecución el espacio para nuevos valores y permiten liberar en tiempo de ejecución el espacio ocupado por valores prescindibles.

6.3.1. Gestión dinámica de la memoria.

El gestor de memoria puede reservar zonas de memoria en tiempo de ejecución, lo que se conoce como gestión dinámica de la memoria.

Todos los lenguajes de programación que permiten manejo de punteros, tienen también funciones específicas para realizar dicha gestión. El lenguaje `C` dispone de las siguientes funciones, que permiten reservar y liberar bloques de memoria:

- `malloc(tamaño)`: Reserva el número de bytes indicados mediante `tamaño` y devuelve un puntero a la dirección de memoria a partir de la cual se ha hecho la reserva.
- `free(dirección)`: Libera la memoria apuntada por `dirección`, que previamente se había reservado.
- `calloc(elementos,tamaño)`: Reserva espacio suficiente para almacenar el número de datos indicado por `elementos`, sabiendo que ocupan el `tamaño` indicado y devuelve un puntero a la dirección de memoria a partir de la cual se ha hecho la reserva.
- `realloc(dirección,tamaño)`: Reserva el número de bytes indicados mediante `tamaño` en una dirección de memoria que ya había sido reservada, `dirección`. El nuevo tamaño puede ser mayor o menor que el anterior.

De estas funciones, en el presente capítulo, sólo se van a utilizar las dos primeras.

La función `malloc()`.

Tal y como se ha dicho, para reservar una zona de memoria se utiliza la función `malloc(tamaño)`. Sobre su uso hay que realizar algunas consideraciones.

Por ejemplo, al realizar la llamada,

```
malloc(80)
```

el resultado es un puntero de tipo `void`, es decir, sin tipo asociado. Por lo tanto, lo más habitual al utilizar esta función es realizar un *"cast"*, una conversión de tipo, que consistirá en poner entre paréntesis el tipo al que se quiere convertir el puntero.

Si se supone que se ha declarado `vd` como un puntero a `double`,

```
double *vd;
```

la siguiente instrucción reserva 80 palabras de memoria y guarda la dirección de la primera posición reservada en `vd`:

```
vd=(double *)malloc(80);
```

y, entonces, `vd` contiene la dirección del primer `double` (y `*(vd)` permite acceder a su valor), `vd+1` contiene la dirección del segundo (y su valor es accesible mediante `*(vd+1))` ... Con 80 bytes se pueden almacenar hasta 10 reales de tipo `double` ya que cada uno ocupa 8 bytes.

Pero no parece una función muy cómoda de usar si, para ello, es preciso calcular cuánto espacio requerirán en memoria los datos; más aún si se tiene en cuenta que de un sistema a otro, el número de bytes para almacenar un mismo tipo puede cambiar. Por eso, suele utilizarse en combinación con la función `sizeof(tipo)`, que indica cuántos bytes ocupa un valor del tipo indicado. Así, si se quiere reservar espacio para almacenar `73` valores de tipo entero, se puede hacer lo siguiente:

```
vi=(int *)malloc(73*sizeof(int));
```

que reserva espacio para almacenarlos, sin necesidad de saber si el tipo `int` ocupa 2 ó 4 ó el número de palabras de memoria que sea. En general, si se supone declarado un puntero,

```
tipo_base *ptro;
```

y se quiere reservar espacio para `n` elementos de `tipo_base`, se utilizará la expresión

<ptro>=(<tipo_base> *)malloc(<n>*sizeof(<tipo_base>));

Un riesgo a asumir cuando se trabaja con la función `malloc()` es que se solicite más memoria de la disponible. En ese caso, la función devolverá el valor `NULL`. Una secuencia típica de instrucciones para prever esta circunstancia puede ser la siguiente:

```
ptro=(tipo_base *)malloc(n*sizeof(tipo_base));
if (ptro == NULL) {
    /* Gestión del error */
}
else {
    /* Gestión de las posiciones reservadas */
}
```

La función `free()`.

Una orden extremadamente importante es `free(ptro)`, que libera la memoria que se había reservado para un elemento. Si no se libera, el gestor de memoria considerará que está reservada

cuando, realmente, ya no se está utilizando. Esto es especialmente preocupante si, después de un largo periodo de ejecución del programa, se han realizado muchas inserciones a medida que se ha ido necesitando gestionar nuevos datos, pero sin tener el cuidado de borrarlos cuando ya no son necesarios: puede ser que el ordenador se "quede sin memoria" (esto es, el gestor no dispone de posiciones libres) cuando en realidad en ese momento sólo se esté trabajando con pocos valores.

Cuando se necesita almacenar nuevos valores, la orden es

```
ptro=(tipo_base *)malloc(n*sizeof(tipo_base));
```

Al ejecutar la orden

free(<**ptro**>**);**

se libera la memoria que se había reservado a partir de la dirección apuntada por `ptro`.

6.3.2. Estructuras enlazadas.

Un tipo de datos es el conjunto de una estructura de datos y las operaciones básicas de manipulación. Los tipos de datos dinámicos se caracterizan porque sus operaciones básicas permiten *aumentar o disminuir el número de elementos* que se agrupan en su estructura de datos. Este comportamiento dinámico es una ventaja en gran número de aplicaciones pero, a cambio, el programador debe definir completamente la estructura de datos y las operaciones que lo definen, puesto que no es habitual que los lenguajes de programación los incorporen.

Los principales tipos dinámicos son:

- Pila: Basado en una estructura dinámica con un único punto de acceso, el *tope*, sus operaciones definen un comportamiento conocido como *Last In, First Out* o LIFO: al tener un único punto de acceso el último elemento incorporado a la estructura es el primero que se puede eliminar de ella. La idea gráfica sería una pila de bandejas en un autoservicio: se coge la que está más arriba y cuando se ha terminado de utilizar se deja en otro montón en la parte superior, de forma que será la primera en ser extraída.

  En un sistema informático es una estructura muy importante: aunque sólo sea por el papel que juega la *pila de la CPU* en el mecanismo de las llamadas a las subrutinas.

- Cola: Está basado en una estructura dinámica con dos puntos de acceso, el *frente* y la *cola*, y sus operaciones definen un comportamiento conocido como *First In, First Out* o FIFO: del frente se elimina siempre el elemento más antiguo de la estructura de datos y los nuevos elementos se incorporan por la cola. La idea gráfica, como se puede esperar, sería la de una cola de personas esperando a ser atendidas en una ventanilla.

  En un sistema informático, las colas juegan un papel crucial en todas las tareas que permiten controlar el acceso a recursos compartidos en el sistema, como pueden ser la propia CPU (en un sistema multiusuario), o una impresora, por ejemplo.

- Lista: Se basa en una estructura dinámica que permite que todos los elementos que la componen sean accesibles, y que en cualquier punto puedan eliminarse o añadirse elementos. Sus operaciones y una posible implementación, serán objeto de la siguiente subsección.

- Árbol: Si los tipos anteriores se basan en una estructura *lineal* (cada elemento tiene un elemento anterior y un elemento posterior), la estructura de datos que sustenta este tipo se caracteriza, además de por ser dinámica, porque cada elemento tiene un único *antecesor* y ninguno, uno o varios *sucesores*. Sus operaciones permiten acceder a cualquier elementos, así como eliminarlo o insertar uno nuevo en cualquier punto. La idea gráfica sería un árbol genealógico, por ejemplo, en el que cada persona tuviera dos elementos posteriores, sus padres.

  En un sistema informático, los árboles son básicos en la gestión del sistema de ficheros y su división en directorios y subdirectorios.

- Grafo: También se basa en una estructura no lineal pero, además, no hay límite en cuanto al número de elementos antecesores y elementos posteriores: en la estructura de datos sobre la que se implementa el tipo cada elemento puede tener ninguno, uno o varios *antecesores* y ninguno, uno o varios *sucesores*. Sus operaciones permiten acceder sin ninguna restricción a cada elemento y se puede añadir o eliminar un elemento en cualquier punto de la estructura. La idea gráfica podría ser un mapa de carreteras: cada ciudad se identifica con un elemento y las carreteras entre ciudades representarían los enlaces entre elementos.

  Su importancia es vital a la hora de modelizar sistemas de redes de ordenadores, por ejemplo, a la hora de estudiar los caminos óptimos o cómo distribuir información entre todos los computadores de una red local o cómo establecer caminos alternativos si desaparece un servidor de la red.

En esta tema el único tipo de datos dinámico que se estudiará será el tipo *lista*. Para comprender cuáles son las operaciones y cuáles son las ventajas de su uso, se recurrirá a su comparación con el vector.

Ambos tipos se basan en estructuras de datos homogéneas y lineales; en ambos, puede accederse a cualquier elemento. Cuando se trabaja con vectores, el acceso es directo. Ello hace que manejar los datos almacenados y procesarlos sea simple, pero existen operaciones que muestran la poca flexibilidad que se ha de pagar por ese acceso simple (y muy rápido) a cada elemento del vector.

Por ejemplo, se quiere disponer de los datos de varias personas (nombre, teléfono, edad...) en un vector. El primer inconveniente es la necesidad de saber el número de elementos que se han de almacenar en el vector; como no siempre es factible, lo que se suele hacer es definir un tamaño máximo y, por lo tanto, desperdiciar memoria, ya que de todo el vector sólo se utiliza una parte. Pero, además, si se desea insertar un nuevo dato en una zona intermedia del vector, no queda más remedio que desplazar todos los elementos posteriores una posición a la derecha con respecto a la que ocupan. O, si hay que borrar algún dato, de nuevo hay que realizar un desplazamiento, ahora hacia la izquierda, de todos los elementos en posiciones posteriores si se desea que la información esté almacenada de forma compacta.

Estos inconvenientes no existen en una lista. Al ser una estructura dinámica, no es preciso definir a priori su tamaño; a medida que se quieran incorporar nuevos elementos o eliminar elementos ya existentes, simplemente se almacenan o eliminan, sin necesidad de desplazar los demás. Pero, al igual que ocurre con los vectores sus ventajas llevan implícitas un precio a pagar: una lista es una estructura de datos enlazada.

Resulta habitual que, en la implementación de los tipos dinámicos, se recurra a estructuras *enlazadas*, que se caracterizan porque almacenan, además de los datos, la información adicional necesaria para mantener unidos dichos datos o, como su nombre indica, mantenerlos enlazados.

El siguiente ejemplo ilustra la idea: la información (nombre y teléfonos) de unas personas se almacenan en una estructura enlazada, ya que junto con la información se almacena un puntero al siguiente elemento de la estructura.

```
1  struct telefonos {
2     char nombre[30];
3     char tel[15];
4     struct telefonos *sig;
5  };
6
7  int main() {
8     struct telefonos t1={"Pepe","666999666"};
9     struct telefonos t2={"Jose","999666999"};
10    struct telefonos t3={"Pepin","969696969"};
11    struct telefonos *primero, *aux;
12
13    primero=&t1;  /* t1 será el primer elemento de la estructura */
14    t1.sig=&t2;   /* t2 será el siguiente al primero, el segundo */
15    t2.sig=&t3;   /* t3 será el siguiente al segundo, el tercero */
16    t3.sig=NULL;  /* y no hay ningún elemento más */
17
18    /* Recorrido de la estructura para imprimir los datos */
19    aux=primero;
20    do {
21        printf("%-30s%-15s \n", aux->nombre, aux->tel);
22        aux=aux->sig;
23    } while (aux!=NULL);
24 }
```

En el ejemplo anterior hay dos cuestiones que recalcar. La primera, con respecto a cómo se realiza la definición de un nodo básico de información (líneas 1–5),

```
struct telefonos {
   char nombre[30];
   char tel[15];
   struct telefonos *sig;
};
```

mediante una tupla en la que uno de los campos es un puntero a la propia tupla. Eso no quiere decir que se "apunte a sí mismo". Esto se hace así, porque cada nodo tiene un puntero al siguiente (`struct telefonos *sig`) y el siguiente nodo es *exactamente del mismo tipo básico*.

Y, segundo, tal y como se accede a la información es *necesario conocer cuál es el puntero al primer nodo* para poder acceder al resto de los elementos; es decir, si se pierde la información de cuál es la dirección del primer nodo de la estructura es imposible acceder al resto, ya que el primero contiene la información sobre la dirección del segundo, el segundo apunta al tercero, etc. (líneas 19–23).

```
   aux=primero;
   do {
       printf("%-30s%-15s \n", aux->nombre, aux->tel);
       aux=aux->sig;
   } while (aux!=NULL);
```

En una estructura enlazada el acceso es secuencial: sabiendo cuál es el primer elemento es posible acceder a todos los demás en secuencia. No hay un acceso directo como en los vectores, pero a

cambio las operaciones de inserción o borrado son más sencillas y el tamaño de la estructura puede crecer y decrecer en tiempo de ejecución.

El ejemplo anterior no es todavía un buen ejemplo de una lista. Es una estructura enlazada pero definida en función de tres variables estáticas (`t1`, `t2` y `t3`). Una lista no tiene esa limitación y la gestión de su información es completamente dinámica.

Definición 6.2 (Estructura de Datos Lista) *Estructura de datos homogénea, lineal y totalmente dinámica en la que todos sus elementos son accesibles de forma secuencial.*

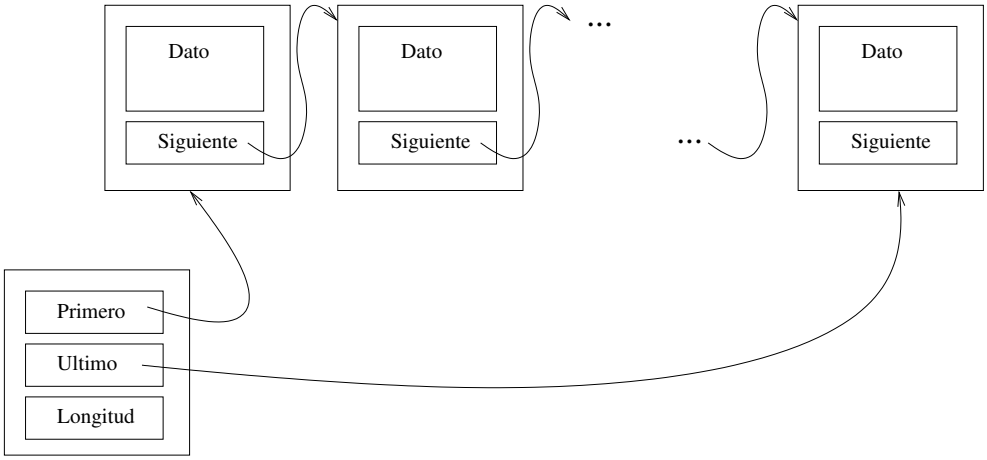


Figura 6.1: Representación gráfica de una estructura enlazada

La figura 6.1 muestra una representación gráfica de una lista simplemente enlazada. Cada elemento es una tupla que almacena un dato (que puede ser de un tipo básico, o predefinido: un vector, una tupla ... u otra lista) y un puntero al siguiente elemento de la lista. Es lo que se denominará el *nodo básico*.

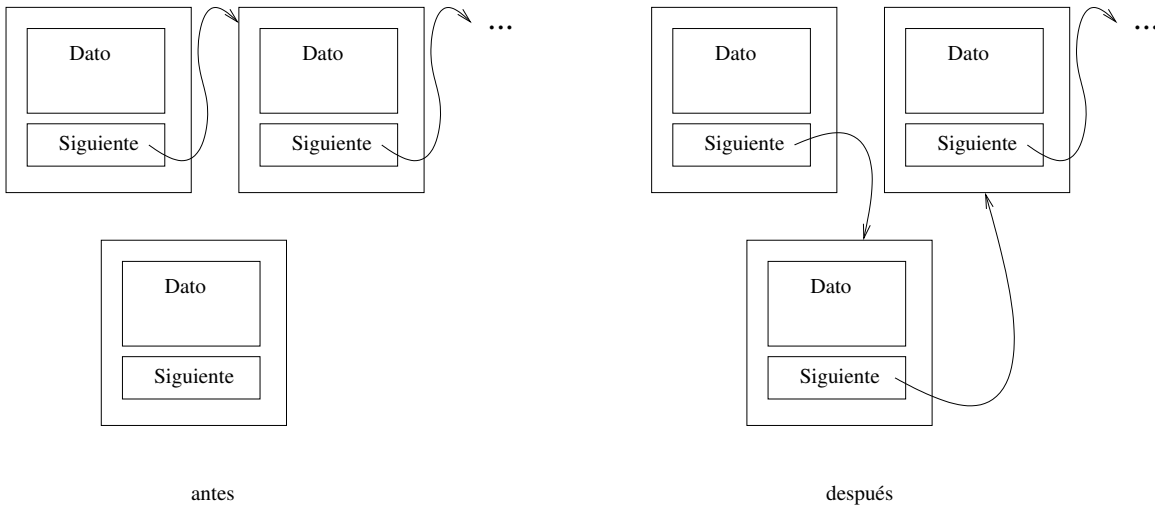


Figura 6.2: Inserción de un dato en una estructura enlazada

Una tupla adicional, a la que se denominará *estructura principal*, almacena la información necesaria para acceder a los elementos de la estructura enlazada. Como se ha dicho, es básico e imprescindible conocer cuál es la dirección del primer nodo. Pero, además, en el ejemplo de la figura, en esa tupla se almacenan la dirección del último elemento y la longitud de la lista, es decir, el número de elementos que la componen. Se ha elegido esta información adicional para facilitar algunas de las operaciones de manejo de la lista.

Una vez definida la estructura de datos, debe pasarse a la definición del tipo de datos lista; esto es, debe pasarse a la definición de las operaciones que permiten manipular la información almacenada en la estructura. Puesto que se trata de una estructura dinámica, las principales serán las que permiten la inserción de nuevos datos y la de borrado.

La inserción de un elemento consistirá en reservar memoria para este nuevo elemento, determinar en qué posición se desea insertarlo y reasignar los punteros de los elementos implicados para que forme parte de la estructura enlazada, tal y como se muestra en la figura 6.2.

En cuanto al borrado, primero habrá que determinar cuál es el elemento a borrar y que posición ocupa en la estructura, reasignar los valores de los punteros de los elementos adyacentes y liberar la memoria que ocupaba el dato tal y como se muestra en la figura 6.3.

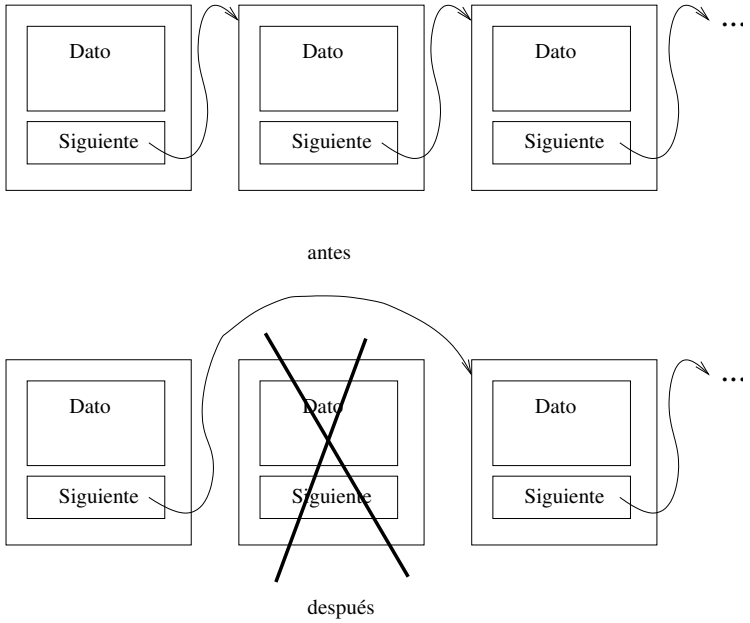


Figura 6.3: Borrado de un dato en una estructura enlazada

Pero el tipo de datos lista viene definido por un *Tipo Abstracto de Datos*, TAD, que ofrece muchas más operaciones. Su definición es la siguiente:

Definición 6.3 (TAD lista) *Tipo Abstracto de Datos cuyo comportamiento queda definido por el siguiente conjunto de operaciones, clasificadas en tres categorías:*

1. *Constructoras:* `crearLista()` *y* `almacenar()`.
2. *Consultoras:* `listaVacia()`, `primero()`, `ultimo()`, `longitud()`, `siguiente()`, `anterior()`, `dato()` *y* `buscar()`.
3. *Modificadoras:* `insertarAntes()`, `insertarDespues()` *y* `borrar()`.

(Además, el TAD incluye una serie de axiomas para garantizar la corrección de la implementación de estas operaciones).

A continuación y como ejemplo, se realiza la implementación completa en C del TAD lista entera (es decir, el tipo base de la lista, el del dato que se almacena, es el entero). Para ello, primero hay que definir una estructura enlazada, tanto el nodo básico, como la estructura principal. Después se definirán las operaciones del TAD, como funciones o procedimientos.

6.3.3. Ejemplo: Definición de una lista simplemente enlazada.

Definición del nodo básico.

El nodo básico está formado por el *"Dato"* (así aparece indicado en las figuras 6.1 y 6.2) y un *puntero a un nodo básico*[5]. Por este motivo, estas estructuras reciben el nombre de estructuras autoreferenciadas. En C las estructuras autoreferenciadas se definen del siguiente modo:

Definición_Nodo_Básico

```
1  /* Definición de la tupla básica */
2  struct nodo {
3    int dato;
4    struct nodo *sig;
5  };
6
7  /* Definición del tipo nodo básico */
8  typedef struct nodo tNodo;
9
10 /* Definición de un tipo puntero al nodo básico */
11 typedef tNodo *tDirNodo;
```

que define el tipo `tNodo` como el tipo del nodo básico[6] y el tipo `tDirNodo` como un puntero a un nodo básico.

El ejemplo que se va a desarrollar es el de la lista de enteros, pero cualquier tipo podría ser el tipo base de la lista. Si, por ejemplo, se desea que sea una tupla con información relativa a una persona, se definiría en primer lugar una tupla conteniendo los datos que van a almacenar en la lista, y esa sería la definición del "Dato":

[5] Esto ya se hizo en el primer ejemplo de estructura enlazada y no quiere decir que una tupla tenga un puntero que apunta a ella misma, sino que tiene un puntero que apunta a una tupla de su mismo tipo de datos.

[6] La definición del tipo `tNodo` ha de hacerse en dos pasos por cuestiones de sintaxis del lenguaje C: la definición de tipo lleva implícito un orden de declaraciones que haría imposible la autoreferencia.

```
typedef struct {
  char nombre[80];
  char telefono[15];
  int edad;
} tPersona;
```

A continuación se definiría el nodo básico con un campo dato de tipo `tPersona` y un campo puntero (la línea 3 en la definición del nodo sería `tPersona dato`).

Definición de la estructura principal.

Como ya se ha comentado la información de la dirección del primer elemento es imprescindible; además, por cuestiones de comodidad se añade la información relativa a la dirección del último nodo y la de cuántos elementos hay en la estructura:

Definición_Estructura

```
1 typedef struct {
2   int longitud;
3   tDirNodo primero, ultimo;
4 } tLista;
```

De esta forma, quedan definidas las estructuras para almacenar y acceder a los los datos que se agrupen en una lista enlazada. Sólo falta definir las operaciones (funciones y procedimientos) que se pueden utilizar con estas estructuras de datos.

Definición de las operaciones Creadoras.

En esta categoría están las operaciones imprescindibles para poder crear una lista y comenzar a trabajar con ella. En el TAD lista hay dos:

1. `crearLista`: Crea una lista vacía. Esta operación es imprescindible para comenzar a trabajar con la lista. Nótese que con los tipos básicos o estructurados, incorporados al lenguaje, basta con declarar variables de un tipo para "crearlas". En el tipo lista, deben crearse explícitamente.

crearLista()

```
1  tLista crearLista(){
2  /* Pre: no hay */
3
4    tLista l;
5
6    l.longitud=0;
7    l.primero=NULL;
8    l.ultimo=NULL;
9
10   return l;
11
12 /* Post: devuelve una lista l vacia, l=VACIA */
13 }
```

Puesto que la lista inicialmente estará vacía, su longitud es `0` y los punteros al primer y último elementos son nulos. No necesita argumentos y devuelve una lista que está vacía.

2. `almacenar`: Dados una lista y un dato, añade el dato a la lista como último elemento.

almacenar()

```
1  void almacenar(tLista *l, int d) {
2  /* Pre: l=L1 es una lista ya creada y d es un entero */
3
4    tDirNodo aux;
5
6    aux=(tDirNodo)malloc(sizeof(tNodo));
7
8    aux->dato=d;
9    aux->sig=NULL;
10
11   if (l->longitud == 0)
12     l->primero=aux;
13   else
14     l->ultimo->sig=aux;
15
16   l->ultimo=aux;
17   l->longitud=l->longitud+1;
18
19 /* Post: l=L2 es la lista L1 con el valor d almacenado */
20 /* como último elemento de la lista y long(L2)=long(L1)+1 */
21 }
```

La lista será un parámetro de E/S, puesto que va a variar en el proceso (por lo menos, el campo `ultimo` y el campo `longitud`). El entero es un parámetro de entrada.

En la línea 6, se ve cómo se hace la reserva de espacio para un nuevo elemento. La dirección del espacio reservado se devuelve sobre `aux`; una vez que se conoce su valor, ya se puede almacenar la información asociada a ese nodo (líneas 8 y 9), el dato y el puntero al siguiente, que se sabe que será `NULL` puesto que el nuevo dato se incorporará al final de la lista.

El condicional de la línea 11 es necesario, puesto que si la lista inicialmente está vacía el nuevo elemento es el único de la estructura; por lo tanto, no sólo será el último sino que también será el primero. Si no fuera así, la asignación de la línea 14 lo enlaza en la estructura previa.

Por último, en las líneas 16 y 17 se actualizan convenientemente los campos `ultimo` y `longitud` de la lista.

Definición de las operaciones Consultoras.

En esta categoría entran una serie de operaciones que normalmente facilitan el trabajo con la lista. Simplemente, consultan un dato determinado y devuelven su valor.

1. `listaVacia`: Dada una lista indica si está o no está vacía.

listaVacia()

```
1  boole listaVacia(tLista l) {
2  /* Pre: l es una lista previamente creada */
3
```

```
4   return (l.longitud == 0);
5
6   /* Post: devuelve CIERTO si la lista está vacía, l=VACIA */
7   /* FALSO en caso contrario, l!=VACIA */
8   }
```

La lista es un parámetro de entrada; puesto que la implementación incluye el campo `longitud` se puede saber si la lista está o no está vacía consultando si su valor es igual o distinto de `0`.

2. `primero`: Dada una lista, devuelve un puntero a la dirección del primer elemento de la lista.

primero()

```
1   tDirNodo primero(tLista l) {
2   /* Pre: l es una lista previamente creada */
3
4     return l.primero;
5
6   /* Post: devuelve el puntero al primero */
7   }
```

3. `ultimo`: Dada una lista, devuelve un puntero a la dirección del último elemento de la lista.

ultimo()

```
1   tDirNodo ultimo(tLista l) {
2   /* Pre: l es una lista previamente creada */
3
4     return l.ultimo;
5
6   /* Post: devuelve el puntero al ultimo */
7   }
```

4. `longitud`: Dada una lista, devuelve su longitud.

longitud()

```
1   int longitud(tLista l) {
2   /* Pre: l es una lista previamente creada */
3
4     return l.longitud;
5
6   /* Post: devuelve un entero que indica cuántos */
7   /* elementos hay en la lista */
8   }
```

5. `siguiente`: Dada una lista y un puntero a un elemento dado, devuelve un puntero a la dirección del elemento siguiente al dado.

siguiente()

```
1   tDirNodo siguiente(tLista l, tDirNodo dn) {
2   /* Pre: l es una lista ya creada y l!=VACIA y */
3   /* dn es un puntero válido a un elemento de la lista */
4
5     return dn->sig;
6
7   /* Post: devuelve el puntero al nodo siguiente */
8   /* al apuntado por dn, NULL si dn=ultimo(l) */
9   }
```

6. `anterior`: Dada una lista y un puntero a un elemento dado, devuelve un puntero a la dirección del elemento anterior al dado.

anterior()

```
1  tDirNodo anterior(tLista l, tDirNodo dn) {
2  /* Pre: l es una lista ya creada y l!=VACIA y */
3  /* dn es un puntero válido a un elemento de la lista */
4
5    tDirNodo aux;
6
7    if (dn==primero(l))
8      aux=NULL;
9    else {
10     aux=primero(l);
11     while (siguiente(l,aux)!=dn)
12       aux=siguiente(l,aux);
13   }
14   return aux;
15
16 /* Post: devuelve el puntero al nodo anterior */
17 /* al apuntado por dn y NULL si dn=primero(l) */
18 }
```

El condicional de la línea 7 es necesario puesto que si el puntero corresponde al primero de la lista, el valor a devolver debe ser NULL, ya que no hay ningún otro nodo antes (¿qué ocurre en la operación `siguiente()` en el caso de que `dn` coincida con el puntero al último?).

Si no es así, en la línea 10 comienza un recorrido por la lista: a partir del primer elemento, mientras el siguiente no coincida con `dn` (línea 11), se avanza al siguiente nodo (línea 12).

Nótese que tanto en ejemplos anteriores como en la definición de esta operación, el esquema de recorrido sigue el esquema genérico que se estudió en el tema 5.

7. `dato`: Dada una lista y un puntero a un elemento dado, devuelve el dato almacenado en dicho elemento.

dato()

```
1  int dato(tLista l, tDirNodo dn) {
2  /* Pre: l es una lista ya creada y l!=VACIA y */
3  /* dn es un puntero válido a un elemento de la lista */
4
5    return dn->dato;
6
7  /* Post: devuelve el valor de tipo base almacenado */
8  /* en el nodo apuntado por dn */
9  }
```

8. `buscar`: Dada una lista y un valor del tipo base de la lista, devuelve un puntero al nodo en el que se encuentra dicho valor.

buscar()

```
1  tDirNodo buscar(tLista l, int d) {
2  /* Pre: l es una lista ya creada y l!=VACIA y */
3  /* d es un valor del tipo base de la lista */
4
5    tDirNodo aux;
6
```

```
7    aux=primero(l);
8    while ((dato(l,aux)!=d) && (aux!=ultimo(l))) {
9      aux=siguiente(l,aux);
10   }
11
12   if (dato(l,aux)!=d)
13     aux=NULL;
14
15   return aux;
16
17 /* Post: devuelve el puntero al nodo en el que se ha */
18 /* encontrado a d y NULL si d no está en l */
19 }
```

La operación está basada también en el esquema general de búsqueda presentado en el tema 5: en la línea 7 se da un valor inicial a la variable auxiliar que permite recorrer la estructura. El condicional de la línea 8 está gobernado por dos condiciones, de forma que se avanza en la estructura hasta que la búsqueda tiene éxito o se llega al último nodo (¿qué pasaría si se modificara la segunda condición y se utilizara el predicado `aux!=NULL`?).

En la línea 12, el condicional permite sobreescribir con NULL el valor de `aux` si la búsqueda no ha tenido éxito (se ha llegado al último elemento y tampoco coincide con el valor de `d`). En caso contrario, se devuelve el puntero del nodo en el que se ha encontrado el valor `d`.

Definición de las operaciones Modificadoras.

Como su propio nombre indica, en esta categoría entran aquellas operaciones que permiten modificar a la estructura de datos, bien añadiendo, bien eliminando elementos en ella.

1. `insertarAntes`: Dada una lista, un dato y un puntero a un elemento de la lista, añade el dato nuevo justo antes de dicho elemento.

insertarAntes()

```
1  void insertarAntes(tLista *l, tDirNodo dn, int d) {
2  /* Pre: l=L1 es una lista ya creada y l!=VACIA y
3  /* dn es un puntero válido a un elemento de la lista y */
4  /* d es un valor del tipo base de la lista */
5
6    tDirNodo aux, aux2;
7
8    aux=(tDirNodo)malloc(sizeof(tNodo));
9
10   aux->dato=d;
11   aux->sig=dn;
12
13   if (dn==primero(*l))
14     l->primero=aux;
15   else {
16     aux2=anterior(*l,dn);
17     aux2->sig=aux;
18  }
19
20   l->longitud=l->longitud+1;
```

```
21
22  /* Post: l=L2 es la lista L1 con el valor d almacenado */
23  /* antes del nodo apuntado por dn y long(L2)=long(L1)+1 */
24  }
```

En la línea 8 se reserva memoria para el nuevo dato y se asigna la dirección reservada a `aux`. A continuación, líneas 10 y 11, se asignan los valores correspondientes a los campos `dato` y `sig`; este se conoce puesto que si se inserta antes de `dn`, el siguiente nodo seguro que es el apuntado por `dn`.

A continuación, el condicional de la línea 13 permite distinguir el caso especial de que se pretenda insertar antes del primero; en ese caso, el nuevo elemento será el primero de la lista (línea 14). Si no, la línea 14 asigna a `aux2` el valor al anterior a `dn` y el nuevo elemento es enlazado como siguiente. Por último, en la línea 20 se incrementa la longitud de la lista.

2. `insertarDespues`: Dada una lista, un dato y un puntero a un elemento de la lista, añade el dato nuevo justo después de dicho elemento.

insertarDespues()

```
1   void insertarDespues(tLista *l, tDirNodo dn, int d) {
2   /* Pre: l=L1 es una lista ya creada y l!=VACIA y
3   /* dn es un puntero válido a un elemento de la lista y */
4   /* d es un valor del tipo base de la lista */
5
6     tDirNodo aux;
7
8     aux=(tDirNodo)malloc(sizeof(tNodo));
9
10    aux->dato=d;
11    aux->sig=dn->sig;
12    dn->sig=aux;
13
14    if (dn==ultimo(*l))
15      l->ultimo=aux;
16
17    l->longitud=l->longitud+1;
18
19  /* Post: l=L2 es la lista L1 con el valor d almacenado */
20  /* después del nodo apuntado por dn y long(L2)=long(L1)+1 */
21  }
```

Como en la operación `insertarAntes()`, lo primero que se hace es reservar memoria para el nuevo dato (línea 8) y, una vez que esa dirección se ha asignado sobre `aux`, se asignan los valores correspondientes a los campos `dato` (línea 10) y `sig` (línea 11); a este campo se le asigna el valor del siguiente a `dn`, puesto que la inserción es después de él. Además, línea 12, se sabe que el nuevo nodo será el siguiente a `dn`.

En el condicional de la línea 14 se actualiza el campo `ultimo` de la lista, si se hace la inserción después del último. Y en la línea 17 se incrementa el campo `longitud` de la lista.

3. `borrar`: Dada una lista y un puntero a un elemento de la lista, elimina de la lista el nodo apuntado por dicho puntero.

borrar()

```
1  void borrar(tLista *l, tDirNodo dn) {
2  /* Pre: l=L1 es una lista ya creada y l!=VACIA y
3  /* dn es un puntero válido a un elemento de la lista */
4
5    tDirNodo aux;
6
7    if (dn==primero(*l)) {
8      l->primero=dn->sig;
9      if (dn==ultimo(*l))
10       l->ultimo=NULL;
11   }
12   else {
13     aux=anterior(*l,dn);
14     aux->sig=dn->sig;
15     if (dn==ultimo(*l))
16       l->ultimo=aux;
17   }
18
19   free(dn);
20   l->longitud=l->longitud-1;
21
22 /* Post: l=L2 es la lista L1 una vez eliminado el */
23 /* nodo apuntado por dn y long(L2)=long(L1)-1 */
24 }
```

Para implementar la operación `borrar()`, se han de tener en cuenta los distintos casos que se pueden dar: borrar un nodo en el interior de la lista, borrar el que es primero, el que es último o borrar el único nodo de una lista de longitud `1`. Por ello, la operación se estructura mediante condicionales que representen todos los casos.

El de la línea 7, representa el caso en el que haya que borrar el primer elemento; se debe actualizar el campo `primero` de la lista, ya que será el segundo el que pase a ocupar esa posición. Se comprueba también en la línea 9 que, además, no sea también el último, caso que se dará cuando sólo haya un elemento en la lista; si lo es, también hay que actualizar el campo `ultimo` teniendo en cuenta que la lista quedará vacía.

A partir de la línea 12 se trata el caso en el que se pretende borrar un elemento que no es el primero; entonces, se utiliza un puntero auxiliar, `aux`, para apuntar al elemento anterior al que se desea borrar y hacer que su campo `sig` se actualice convenientemente (línea 14). El condicional de la línea 15 tiene por objeto modificar el campo `ultimo` si lo que se pretende es borrar precisamente este elemento.

De esta forma, el elemento apuntado por `dn` ya no forma parte de la lista, ya que se han perdido los enlaces a él, pero aún queda algo muy importante que hacer: en la línea 19, el uso de `free(dn)` permite liberar la memoria que hasta ahora ocupaba el elemento en memoria. Además, en la línea 20, se actualiza el campo `longitud` de la lista.

6.4. Glosario.

apuntador Este término suele aparecer en algunas traducciones sudamericanas como la traducción de la palabra inglesa *pointer* y que aquí se ha traducido como *puntero*.

gestión dinámica de la memoria La realizada cuando en tiempo de ejecución, el programador puede reservar o liberar memoria, a medida que el número de datos a manipular así lo haga conveniente.

gestión estática de la memoria La realizada cuando sólo se manipula el número de datos especificado en la declaración de variables del programa.

gestor de memoria Parte del sistema informático (habitualmente del sistema operativo) que gestiona y conoce en cada momento la situación del mapa de memoria, es decir, qué direcciones están asignadas y cuáles están libres.

palabra de memoria En este tema, se ha utilizado este término como *unidad mínima direccionable* de memoria y, por lo tanto, equivalente a *byte*.

paso de parámetros por referencia Mecanismo de comunicación entre programas y procedimientos que permite que se pueda manipular el valor de un parámetro de E/S, ya que la llamada se realiza haciendo referencia a la posición de memoria del parámetro y el procedimiento puede así acceder directamente a su valor. En el caso del lenguaje `C`, supone realizar la llamada con el operador `&` y trabajar, dentro del procedimiento, con el operador `*`.

paso de parámetros por valor Mecanismo de comunicación entre programas, funciones y procedimientos en la que los valores de los parámetros de entrada son copiados, y se trabaja internamente sobre la copia. Por lo tanto, dichos valores no cambian tras la ejecución de la función o procedimiento.

tipo abstracto de datos, TAD Definición teórica de las operaciones que determinan el comportamiento de un tipo de datos. Incluye la definición formal de las operaciones y un conjunto de axiomas que dichas operaciones deben verificar (y que pueden permitir comprobar si una implementación determinada es o no es correcta).

6.5. Resumen y Consideraciones de Estilo.

En este tema se presentan dos nuevos conceptos que requieren esfuerzos de sentido contrario. Primero, exige descender en el nivel de abstracción que supone el manejo de variables y tomar consciencia de que una variable no es más que una forma abstracta de trabajar con un valor almacenado en la memoria. El concepto de puntero permite trabajar directamente con estos valores; además, permite conocer la forma en que un lenguaje de programación, en este caso el lenguaje `C`, utiliza la herramienta en distintos casos (manejo de vectores y paso de parámetros por referencia).

Pero, una vez conocido el mecanismo de la gestión dinámica de la memoria, el mismo tema exige al programador ascender en el nivel de abstracción, que asuma el concepto de Tipo Abstracto de Datos y que sea capaz de definir el mecanismo capaz de enriquecer el lenguaje con tipos de datos que no están implementados. De acuerdo al ejemplo desarrollado, la lista simplemente enlazada, ello supone por un lado, definir estructuras para poder almacenar los datos que se suponen enlazados en una lista (lo cual no es obvio: la estructura no está presente en el lenguaje, pero se debe manejar. La abstracción realizada, manejando una tupla para representar un nodo y un puntero al primer elemento como elemento imprescindible para "simbolizar" la lista es, posiblemente, una de las más fuertes que se deban asumir en esta asignatura). Por otro, además, hay que definir e implementar sus operaciones de manejo. De esta forma, se incorpora un nuevo tipo de datos al lenguaje.

De entre los objetivos expuestos en el primer tema, el trabajo desarrollado en este puede servir para ilustrar especialmente lo que se planteaba en el último de ellos, *Implementar la solución*: se advertía allí de las dificultades que puede suponer para el programador el hecho de tener que depender

de la sintaxis de un lenguaje, a la hora de implementar la solución diseñada. La definición de nuevos tipos, además de reforzar el concepto de *enriquecimiento del lenguaje* mediante la definición de los módulos adecuados, debe servir para ilustrar las posibilidades de que se disponen para superar las limitaciones de un lenguaje y aprovechar sus recursos en el diseño de una solución: si a través del análisis de un problema se llega a la conclusión de que lo más adecuado es resolverlo utilizando una lista, es preciso explorar cuáles son las herramientas de que se dispone para poder realizar ese trabajo. Es posible que el análisis lleve a plantear el uso de un lenguaje que disponga de esa estructura (si ello es factible); pero también es necesario aprender a explotar las posibilidades de un lenguaje de programación que no implemente el tipo, para así poder enriquecerlo con él.

A lo largo del tema se ha utilizado una de la notaciones estándar más habituales en la definición de tipos y operaciones; así, para todos los tipos definidos se ha utilizado la notación 'tNuevoTipo', es decir, comenzar el identificador por 't' y, para cualquier otra palabra que intervenga, comenzar con mayúscula (tPersona, tLista, ...). En las operaciones se ha utilizado la misma norma: se comienza con letra minúscula y, cualquier otra palabra que intervenga en el identificador, comienza por mayúscula (crearLista, listaVacia, insertarAntes,...).

Por supuesto, al crear un nuevo tipo cobra especial relieve el uso correcto de precondiciones y postcondiciones, no sólo para denotar qué se debe cumplir antes y después de haberse ejecutado la operación, sino también como herramienta para documentar a otro usuario sobre lo que debe esperar de las operaciones que se ponen a su disposición.

6.6. Bibliografía.

1. "The C Programming Language". Brian W. Kernigham & Dennis M. Ritchie. Prentice-Hall Inc. 1988.

6.7. Problemas Propuestos.

1. Indicar y justificar qué valores acabarán saliendo por pantalla al ejecutarse el siguiente fragmento de código:

```
int i, j, *k;

k=&i;
j=7;
*k=j;
i=i+*k;
printf("%d %d", i, j);
```

2. Dadas las siguientes cabeceras y el fragmento de código:

```
int   buscar (float x[], float y);
float elevar (float *v,  int n);
void  contar (float *w, int *m);

int main() {
```

```
  float z[100];
  float t;
  float *p;
  int a;
  ...
  p = &t;
  a = buscar(...z, ...p);
  z[a] = elevar(...t, ...a);
  contar(...z, ...a);
  ...
}
```

¿En cuáles de los parámetros reales de las llamadas a `buscar`, `elevar` y `contar` pondrías `&` ? ¿En cuáles pondrías `*` ? ¿Y en cuáles nada? Justifica la respuesta.

3. El siguiente programa:

```
int main() {
   long *ptrDato;

   *ptrDato=10;
   printf(" %p\n %ld\n", ptrDato, *ptrDato);
}
```

¿provocará errores de compilación? ¿provocará errores de ejecución?. Justificar cada una de las respuestas.

4. Encontrar todos los errores del siguiente programa, y justificar la respuesta.

```
int main() {
    int *A;
    int B[4][8];
    int i,j;

    /* se lee A, un vector dinamico */
    for (j=0; j<10; j++) {
        printf("Componente %d? ", j);
        scanf(" %d", &A[j]);
    }
    /* se imprime B */
    for (j=1; j<=4; j++)
       for(j=1; j<=8; j++)
          printf("B[ %d, %d]= %d\n",i,j,B[i,j]);
}
```

5. Indicar y justificar qué valores acabarán saliendo por pantalla al ejecutarse el siguiente fragmento de código:

```
....
void primero(int *x, int *y) {
 int temp;
```

```
 temp=*x;
 *x=*y;
 *y=temp;
}

void segundo(int *i, int *j) {
 *i=*i+1;
 *j=*j+1;
 primero(i,j);
}

int main() {
 int a=20, b=15;
 primero(&a, &b);
 segundo(&b, &a);
 printf("%d %d\n", a, b);
}
```

6. Dado el siguiente fragmento de programa:

```
float a[10] = {100,1,2,3,4,5,6,7,8,9};
float *b;
float *c;

b=a;
c=b;
for (i=0; i<10; i++) {
  a[i] = *(c+i) + *b;
  b++;
}
```

Indica, para cada una de las siguientes afirmaciones, si es cierta o falsa, y por qué.

(A) Las variables b y c apuntan a la misma direccion de memoria durante toda la ejecución del programa.

(B) El valor de `a[3]` se modifica y termina con valor `6`.

(C) El valor de `a[9]` se modifica y termina con valor `109`.

(D) Si sustituimos `a[i]` por `*a`, la secuencia de instrucciones es equivalente a la original.

7. Indicar y justificar qué valores acabarán saliendo por pantalla al ejecutarse el siguiente fragmento de código:

```
....
void primero(int x, int *y) {
 int temp;
 temp=x;
 x=*y;
 *y=temp;
}
void segundo(int *i, int *j) {
 *i=*j+*j;
 primero(*i,j);
}
```

```
int main() {
 int a=20, b=15;
 primero(a, &b);
 segundo(&a, &b);
 printf("%d %d\n", a, b);
}
```

8. Dadas las siguientes definiciones de variables:

   ```
   int x;
   int *p1;
   int **p2;
   ```

   ¿Cuál de las siguientes secuencias permite que `x` tome el valor de 4 de forma correcta?

   (A) `p1 = &p2; *p2 = &x; *p1 = 4;`

   (B) `p2 = &x; *p2 =4;`

   (C) `p2 = p1; p1=&x; *p2 = 4;`

   (D) `p2 = &p1; p1 = &x; **p2 = 4;`

9. **(0.5 p.)** Dado el siguiente fragmento de programa:

   ```
   float a = 0.0001;
   float *b;
   float *c;

   b = &a;
   c = b;
   a = *c + *b;
   ```

   Indica, para cada una de las siguientes afirmaciones, si es cierta o falsa, y por qué.

   (A) Las variables `b` y `c` se almacenan en la misma dirección de memoria.

   (B) La sentencia `*c = 4;` modificaría el contenido de la variable `a`.

   (C) `a` tomará el valor 0.0002.

   (D) `c` almacena la dirección de la variable `b`.

   (E) La sentencia `a = *c + *b;` es incorrecta porque se asigna un puntero a la variable `a`, que es un número real.

10. Se ha realizado la siguiente declaración de tipos y de estructuras de datos en `C`:

    ```
    typedef struct {
        float media;
        float notas[3];
    } calif;

    typedef struct {
        char nom[51];
        calif asigs[9];
    } alum;

    alum alumnos[120];
    ```

Para acceder a la segunda nota de la primera asignatura del cuarto alumno, se puede usar una sola de las siguientes opciones:

a) `alumnos[3][0][1]`

b) `alumnos[3]->asigs[0]->notas[1]`

c) `alumnos[3].asigs[0].notas[1]`

d) `alumnos[3]->asigs[0].notas[1]`

Se pide:

a) Indicar cuál es la opción correcta, justificando la respuesta.

b) Indicar al menos un fallo en cada una de las otras opciones.

11. Una lista enlazada almacena valores enteros. Hay que escribir una función o procedimiento que devuelva dos listas: en una se incluyen los elementos positivos y en la otra los negativos.

    Se pueden usar todas las funciones estudiadas en teoría.

12. Dada una lista enlazada, tal y como se ha definido en teoría, y un puntero válido a un elemento de la lista, escribir una función o procedimiento que implemente la operación `borraAntPost()`, que consiste en borrar el elemento anterior y el posterior (si ello es posible) del apuntado por el puntero.

13. Dada una lista enlazada como la estudiada en teoría, escribir un procedimiento al que se le pasarán como parámetros la lista y un valor de tipo base y cuyo efecto debe ser que dicho valor se borra completamente de la lista (se eliminan todas sus ocurrencias); si no está en la lista, la lista queda como estaba.

    Se pueden usar todas las funciones estudiadas en teoría.

14. Escribir en una función o procedimiento (justificar la elección) que, dada una lista de `n` números reales, añada al final de la lista `n` nuevos nodos con los mismos valores de la lista original repetidos y en el mismo orden. Por ejemplo, si la lista contiene 4 nodos con los valores

    `[3.5, 6.7, 2.2, 3.5],`

    la lista resultante deberá contener 8 nodos con los valores

    `[3.5, 6.7, 2.2, 3.5, 3.5, 6.7, 2.2, 3.5].`

    Se pueden usar todas las funciones estudiadas en teoría.

15. Dada una lista enlazada, tal y como se ha definido en la teoría, y dos punteros a dos nodos de dicha lista, se pretende escribir un procedimiento en C que permita eliminar los nodos que hay entre ambos, incluyendo los propios nodos apuntados. El procedimiento diseñado es el siguiente:

```
void borrarTrozo (tDirNodo *l, tDirNodo TP1, tDirNodo TP2) {
/* Pre: TP1 y TP2 son punteros válidos a nodos de l */
/* y el nodo apuntado por TP1 está una o varias posiciones */
/* antes que el nodo apuntado por TP2*/

  tDirNodo aux;
```

```
  if (TP1 !=  *l) {
  /* TP1 no es el primero de la lista */
       aux = anterior(*l, TP1);
       /* aux es el nodo anterior a TP1 */

       aux->sig = TP2->sig;
       /* el siguiente a aux es ahora el siguiente a TP2 */
  }
  else
  /* TP1 es el primero: el primero sera el siguiente a TP2 */
       *l = TP2->sig;

  free(TP1);
  free(TP2);

/* Post: en la lista l se han eliminado los nodos desde el */
/* apuntado por TP1 hasta el apuntado por TP2, */
/* incluidos TP1 y TP2 */
}
```

a) Este procedimiento no está bien diseñado, ya que se ha cometido un error lógico (por favor, pasad de ir a la "busca y captura" de puntos y comas fuera de sitio). Indicar cuál es y en qué consiste.

b) Reescribir el código, de forma que sea correcto.

16. Una lista enlazada almacena valores enteros de forma que están ordenados por orden creciente. Hay que escribir una función o procedimiento que dada una lista y un valor entero inserte dicho valor **en su sitio**, de forma que la **lista siga estando ordenada**.

    Se pueden usar todas las funciones estudiadas en teoría.

17. Definir una lista enlazada que almacene valores reales. Escribir un procedimiento al que se le pasarán como parámetros la lista y un puntero a un elemento de la lista y cuyo efecto debe ser que el elemento apuntado pase a ser el primero de la lista.

    Se pueden usar todas las funciones estudiadas en teoría, y además, asumir que están adaptadas de acuerdo a la nueva definición.

18. Definir una lista enlazada que almacene valores de tipo `char` y adaptar las operaciones de la lista a esta nueva definición. Además, hay que escribir una función o procedimiento que dadas dos listas devuelva una nueva lista que contiene los carácteres repetidos en ambas listas en la misma posición. Por ejemplo, con las listas

    ```
    lista1:  HOLA_COMO_ESTAS
    lista2:  HORACIO_ESTAMOS_AQUI
             ^^ ^  ^       ^
    ```

    debería devolver la lista

    ```
    listaNueva:  HOAOS
    ```

19. Dada una lista, se desea implementar la operación `invertir()` cuyo efecto será que los nodos de la lista original estarán en orden inverso tras ejecutar la operación (el primero será el último, el segundo el penúltimo, etc.).

    a) Realizar una función o procedimiento que dada una lista devuelva otra lista con los nodos de la original en orden inverso.

b) Realizar una función o procedimiento que dada una lista devuelva **la misma lista** invertida. En esta implementación no se pueden utilizar listas auxiliares (es decir, la única lista que se puede manejar es la lista original).

20. Dada la siguiente definición de tipos

```
typedef struct {
  char nombre[80];
  float importe;
} tCliente;
```

a) Definir las estructuras de datos necesarias para poder trabajar con una lista de tipo base `tCliente`.

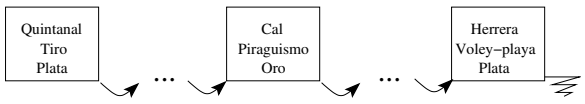
b) Implementar una operación nueva para esa lista, que consistirá en lo siguiente: esa lista es utilizada por una empresa de servicios para gestionar los recibos impagados de sus clientes. Cada cliente puede tener varios recibos impagados; por lo tanto, puede aparecer varias veces en la lista. La nueva operación debe transformar una lista en la que cada cliente puede aparecer varias veces, en otra en la que cada cliente aparecerá sólo una vez y en la que `importe` indica el total acumulado de sus recibos.

Para implementar esta operación se pueden utilizar todas las que se han visto en teoría pero, si hay que modificar alguna, hay que indicar por qué y cómo.

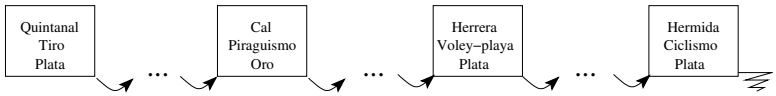
21. Se quiere crear una lista con los medallistas olímpicos españoles. En cada nodo se almacena el **nombre del medallista**, el **deporte** y el **metal de la medalla**.

Cada vez que se gana una medalla, se incorpora a la lista de modo que queden **agrupadas para cada deportista**: primero se mira si el deportista ya tiene una medalla; si ya la tiene se inserta la nueva detrás de la encontrada; si no, se almacena la nueva medalla al final.

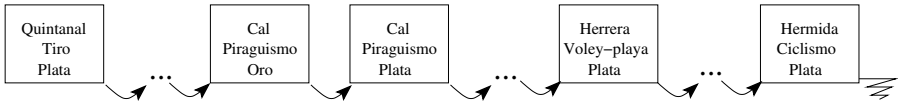
Por ejemplo: dada la lista en la siguiente situación,



el ciclista Hermida obtiene su primera medalla, de plata, de modo que al introducirla en la lista, la lista resultante quedaría así:



Si a continuación, David Cal gana su segunda medalla de plata en piragüismo, la lista entonces quedaría así:



Se pide la redefinición del tipo lista (estructuras de datos y operaciones) según el enunciado y, a continuación, escribir una función o procedimiento que dadas una nueva medalla y una lista ya creada como la anterior, incorpore la medalla a la lista.

22. Dada la siguiente definición de tipos

```
typedef struct {
  int exponente;
  float coeficiente;
} tPolinomio;
```

a) Definir las estructuras de datos necesarias para poder trabajar con una lista de tipo base `tPolinomio`.

b) Implementar una operación nueva para esa lista, que consistirá en lo siguiente: esa lista es utilizada para almacenar los coeficientes no nulos, y sus respectivos exponentes, de un polinomio; en ese polinomio no hay exponentes repetidos y además están ordenados, porque para almacenar se utiliza, precisamente, una operación que consiste en: dados un dato del tipo base (que, por supuesto, almacena un coeficiente y un exponente), se deben añadir ese nuevo dato en el lugar que le corresponde en la lista por orden creciente según el valor del exponente.

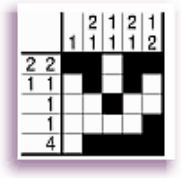
Si el exponente a introducir ya existe en la lista, la operación debe sumar los coeficientes (el existente y el nuevo) y comprobar que el resultado no sea cero: si no lo es, el nuevo coeficiente es la suma, si lo es, hay que eliminar el nodo.

Para implementar esta operación se pueden utilizar todas las que se han visto en teoría pero, si hay que modificar alguna, hay que indicar por qué y cómo.

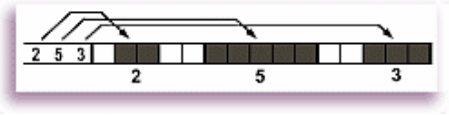
23. Se está celebrando un concurso-oposición que consta de dos exámenes. El primero ya se ha realizado y en una lista enlazada, en cada nodo, se han almacenado el nombre de cada aspirante y la nota que ha obtenido en el primer ejercicio.

    Hay que escribir una función o procedimiento que dada una lista como la anterior y una nota de corte, elimine de esa lista a los aspirantes que superan dicha nota de corte y los añada a una nueva lista en la que figurarán todos los que pasan al segundo ejercicio.

24. Los puzzles japoneses en Blanco y Negro consisten en una rejilla vacía con números a la izquierda de cada línea y en la parte superior de cada columna. Las reglas para resolverlos son las siguientes:

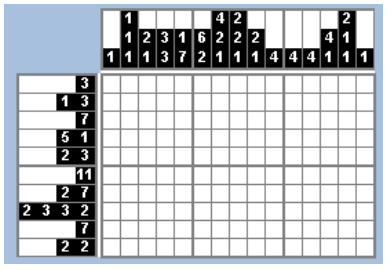
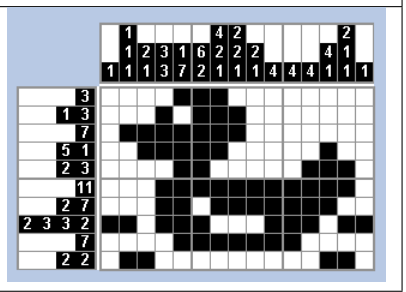


Cada número define la longitud de un bloque de cuadros negros consecutivos, entre cada bloque debe ir al menos un blanco y los bloques siguen la misma secuencia que los números.

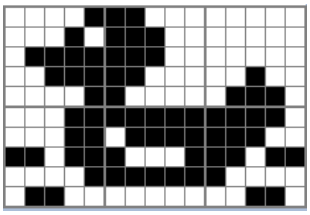
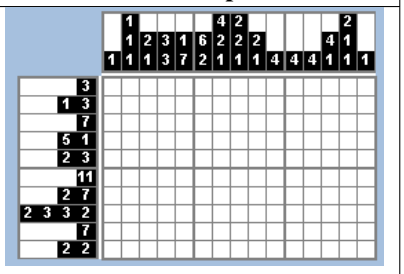


Los números 2, 5, 3 en una fila (por ejemplo) indican que primero va un bloque de 2 cuadros negros, empezando en algún lugar de la izquierda, seguido por un bloque de 5 y otro de 3 y que entre cada bloque debe ir, como mínimo, un cuadro en blanco.

Por ejemplo:

Puzzle propuesto	**Solución**
	

Lo que os proponemos es que ayudéis a los diseñadores de puzzles japoneses: a partir de una matriz de tamaño `N×M` de tipo base `boole` (con el convenio cierto, blanco y falso, negro) y que representa un dibujo como el anterior, hay que conseguir la información de los bloques asociados a cada fila y columna. Es decir,

Nos dan esto	**Resultado esperado**
	

En concreto, os pedimos:

a) Escribe una función o procedimiento (justifica la elección) que, dada la matriz de tamaño `N×M` de tipo base `boole` con el dibujo y un identificador de **columna**, devuelva sobre una estructura de datos **que debes elegir y justificar**, la información sobre los bloques de dicha columna.

b) Lo mismo para el caso de una fila: escoge la estructura de datos adecuada y diseña la función o procedimiento para "rellenarla", partiendo de la matriz y del identificador de fila. Justifica las respuestas.

c) ¿Qué estructura de datos utilizarías para almacenar TODA la información de todas las columnas y todas las filas? ¿Por qué?

d) De acuerdo a lo que hayas contestado en los apartados anteriores, escribe una función o procedimiento que dada una matriz de tamaño `N×M` de tipo base `boole` y que representa un dibujo como el anterior, devuelva TODA la información sobre los bloques de columnas y filas.