PUNTEROS

Francisco Javier Gil Chica

dfists, marzo 2010

Índice general

Prólogo v

1. Fundamentos **1**
1.1. Qué es un puntero 1
1.2. Valor inicial de un puntero 2
1.3. Punteros a punteros 4
1.4. Aritmética de punteros 5
1.5. Punteros a bloques 6
1.6. Punteros a estructuras y estructuras auto-referenciadas 6
1.7. Punteros a funciones 8
1.8. Declaraciones complejas 11

2. Punteros, cadenas y matrices **13**
2.1. Cadenas de caracteres 13
2.2. Matrices 16

3. Listas **19**
3.1. Introducción 19
3.2. Iniciar la lista 20
3.3. Insertar y eliminar nodos 20
3.4. Fusión de listas 25
3.5. Intercambio de elementos y ordenación de listas 26
3.6. Extracción de sub-listas 28

4. Árboles **31**
4.1. Introducción 31
4.2. Inserción de nodos 33
4.3. Encontrar y eliminar nodos 36

5. Punteros en otros contextos **41**
5.1. Concepto 41

5.2. FAT e i-nodos 41
5.2.1. FAT 42
5.2.2. i-nodos 43
5.3. Gestión de memoria 45
5.4. Conclusión 46

Prólogo

Estas notas están dirigidas a mis alumnos de la asignatura de Periféricos, de las titulaciones de Informática en la EPS de la Universidad de Alicante. Aunque dicha asignatura no trata ni de programación, ni de estructuras de datos ni de algorítmica, cada vez se ha hecho más evidente en los últimos años la necesidad de estas notas con vistas a la realización de las prácticas de la asignatura, que por tratar de cuestiones de bajo nivel requieren el manejo de puertos, punteros, operaciones de bits y algunas pocas cosas más que tienden a quedar relegadas debido a la evolución de la programación hacia lenguajes de más alto nivel. Aquí he pretendido cubrir las posibles lagunas por lo que respecta al uso de punteros en el contexto del lenguaje de programación C, al tiempo que realzar la importancia que tiene manejarlos con soltura en relación con aspectos fundamentales de la programación y estructuras de datos.

A medida que se desciende de nivel, acercándonos a la máquina, la computación se simplifica más y más. Al nivel más bajo, el del procesador, las cosas son tremendamente sencillas (lo que no impide que puedan complicarse si hay empeño en ello). Puesto que un programa es una secuencia de símbolos que una vez compilada queda reducida a una larga serie de instrucciones de procesador, podría pensarse en que nada de interés se encontrará allí. Pero ocurre que la diferencia entre unos niveles y otros no es cuantitativa, sino cualitativa. A bajo nivel, una secuencia de instrucciones de procesador es algo aburrido y monótono. A muy alto nivel, todas esas abstracciones modernas son un estorbo que aleja de las soluciones eficientes. Y hay un nivel intermedio, el que se encuentra entre ambos, donde ocurren cosas muy interesantes. Las dos más interesantes son estas: la recursión y los punteros. Estas notas tratan de los segundos.

Capítulo 1

Fundamentos

1.1. Qué es un puntero

Cuando declaramos una variable entera, por ejemplo mediante

```
int i;
```

esta variable, una vez compilado el programa, se encontrará, en tiempo de ejecución, en alguna parte de la memoria, y contendrá el valor que le hayamos asignado, por ejemplo mediante

```
i=10;
```

Ocurre igual con los caracteres y los números reales, con las estructuras y con las uniones: se encuentran en algún lugar de la memoria, y en ese lugar de la memoria se almacena un valor. En el caso de la variable `i`, llamemos `m` a la posición de memoria que ocupa. En la posición m de la memoria se encontrará entonces el valor `10`. Pues bien, un puntero es una variable como cualquier otra, y contiene un número entero. Pero ese número no representa unidades de algo (por ejemplo, con `i=10;` podríamos indicar un número de pares de zapatos, los botones de una caja, los escalones de una escalera...) sino la posición en memoria de otra variable.

En C, un puntero a entero se declara en la forma

```
int *p;
```

Un puntero a un número real

```
float *q;
```

Un puntero a carácter como

```
char *c;
```

y así sucesivamente. Cuando escribimos

```
p=&i;
```

el operador `&` devuelve la dirección de la variable `i`, que en nuestro caso es `m`. De manera que la variable `p` va a contener el valor `m`. Se dice que `p` «apunta» a `i`, ya que indica en qué lugar se encuentra `i`. A su vez, `p` se encuentra también en algún lugar. Puesto que `p` apunta a un lugar de memoria donde está almacenado el valor `10`, el «valor apuntado» por `p` es precisamente `10`. Este valor apuntado se escribe como

```
*p
```

1.2. Valor inicial de un puntero

Cuando declaramos una variable como en

```
int j;
```

no podemos suponer que `j` vaya a contener ningún valor particular. Sólo sabremos qué contenga cuando pongamos allí un valor concreto. Por ejemplo, después de hacer

```
j=20
```

sabemos que la variable `j` contiene el valor `20`, de modo que la expresión `j+4` vale `24`. De la misma manera, cuando declaramos un puntero como

```
int *p;
```

no podemos suponer que contiene ningún valor en concreto, es decir, que apunta a ninguna dirección en particular. Pues bien, existen tres formas distintas de asignar un valor a un puntero:

1. Llamando al gestor de memoria, que buscará un hueco libre y devolverá su dirección. Así, al escribir

   ```
   int *p=(int *)malloc(sizeof(int));
   ```

estamos asignando a `p` la dirección de un hueco cuyo tamaño es suficiente para contener un entero. El gestor de memoria habrá buscado un hueco libre del tamaño adecuado y la dirección de ese hueco queda asignada a `p`. En principio, no sabemos cual es esa dirección, de la misma forma que cuando declaramos un entero no sabemos dónde se localizará, lo cual no impide usarlo almacenando y recuperando en él valores.

2. Asignándole la dirección de una variable ya declarada. Por ejemplo

```
int j=10;
int *p=&j;
```

Ahora `p` contiene la dirección de `j` (apunta a `j`), de forma que `*p` vale `10`

3. Asignándole manualmente una dirección. Por ejemplo

```
char *p=0xb8000000;
```

hace que `p` apunte a una posición particular de la memoria de vídeo. En concreto, al primer carácter de la pantalla (esquina superior izquierda) en el modo texto a color de 25 filas por 80 columnas del adaptador VGA.

Puesto que un puntero declarado no tiene asignado ningún valor, si le es asignado ninguno (por error) antes de ser usado, cuando llegue ese momento no habrá forma de saber si la dirección apuntada es o no lícita. Por eso es buena práctica de programación asignar un valor a un puntero cuando es declarado, o bien asignarle el valor `NULL`, de forma que, en un momento posterior, pueda saberse si el puntero fue asignado o no:

```
char *p=(char *)NULL;
...
if (p==NULL){
...
}
```

1.3. Punteros a punteros

Los punteros apuntan a variables. Pero un puntero es una clase de variable, luego nada impide que un puntero apunte a otro. Así que podemos tener punteros que apuntan a punteros, punteros que apuntan a punteros que apuntan a punteros y así sucesivamente. Véase el siguiente programa:

```
#include <stdio.h>
main()
{
    int j=10;
    int *p=(int *)&j;
    int **q=(int **)&p;
    int ***r=(int ***)&q;
    printf("%d %d %d\n",*p,**q,***r);
    return;
}
```

`p` es un puntero que apunta a un entero; `q` es un puntero que apunta a `p`, y es por tanto un puntero a un puntero; `r` es un puntero que apunta a `q`; `*p`, `**q` y `***r` son tres formas distintas de referirnos al mismo valor `10` almacenado en `j`.

En este punto, dado un puntero `p` declarado como

```
int *p;
```

y apuntado ya algún lugar, por ejemplo

```
int j=10;
....
p=(int *)&j;
```

no ha de haber dificultad en comprender la diferencia que existe entre `p`, `&p` y `*p`. `p` es una variable, que contiene un número, que es una dirección. Así que

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

imprimirá el contenido de la variable `p` de la misma forma que

```
printf("%d",j);
```

imprime el contenido de la variable `j`. Puesto que `p` es una variable como otra cualquiera, `&p` indica su dirección. No la dirección del lugar al que apunta `p`, sino la dirección de la misma `p`. Finalmente, `*p` proporciona el contenido que se haya almacenado en la dirección a la que apunta `p`.

1.4. Aritmética de punteros

Considérese el siguiente programa:

```
#include <stdio.h>
main()
{
    int j;
    char *p=(char *)&j;
    int *q=(int *)&j;
    long *r=(long *)&j;
    printf("%p %p %p\n",p,q,r);
    p+=1;
    q+=1;
    r+=1;
    printf("%p %p %p\n",p,q,r);
    return;
}
```

Se declaran tres punteros. Los tres apuntan a la misma dirección de memoria, que es la de la variable `j`. A continuación incrementamos en una unidad cada puntero y volvemos a imprimirlos. En efecto, el puntero `p` queda incrementado en una unidad, pero los punteros `q` y `r` se han incrementado en cuatro unidades, no en una. El motivo es el siguiente: cuando un puntero se incrementa en una unidad, el resultado no apunta a la siguiente dirección de memoria, sino al siguiente objeto del tipo al que apunte el puntero. Si tenemos un puntero a carácter, como `p`, `p+1` apunta al carácter siguiente. En un sistema en que los caracteres son de ocho bits, el carácter siguiente a uno dado se encuentra en el byte siguiente. Pero en el caso de `q`, que apunta a un entero, `q+1` no apuntará a la dirección siguiente, sino al entero siguiente. En nuestro sistema, en que los enteros son de 4 bytes, `q+1` es una dirección que difiere en cuatro unidades de `q`. A esta forma particular de aritmética se la denomina «aritmética de punteros». Una aplicación obvia consiste en copiar secuencias de objetos de una posición a otra. Por ejemplo, si `p` y `q` son punteros a carácter, la forma de copiar `n` caracteres que se encuentran a partir de la posición a que apunta `p` en las posiciones sucesivas a partir de la dirección a la que apunta `q` es

```
for(j=0;j<n;++j)
    *(q+j)=*(p+j);
```

Si `n==1000`, se copiarán mil caracteres, que en nuestro sistema son 1000 bytes. Pero si `p` y `q` fuesen punteros a entero, la dos líneas anteriores copiarán 1000 enteros, que en nuestro sistema son 4000 bytes.

1.5. Punteros a bloques

Cuando un puntero se hace apuntar a un carácter, apunta a ese carácter. Pero cuando apunta a un entero, lo hace al primer byte de ese entero. La generalización es obvia: cuando un puntero apunta a un bloque de un número cualquiera de bytes, lo hace al primer byte de ese bloque. Por ejemplo, en:

```
char *p=(char *)malloc(1000*sizeof(char));
```

`p` se hace apuntar al primer byte de un bloque que contiene 1000 caracteres. Es por eso que en C a las cadenas de caracteres se accede mediante un puntero al primer carácter de la cadena. En C, las cadenas terminan con el carácter nulo, lo que significa que una cadena de `n` caracteres está ocupando en memoria `n+1` bytes. Una implementación que obtiene la longitud de una cadena y usa aritmética de punteros se da como ilustración:

```
int strlen(char *cadena)
{
    char *p=cadena;
    int n=0;
    while (*p!='\0'){
        ++p;
        ++n;
    }
    return(n);
}
```

1.6. Punteros a estructuras y estructuras autoreferenciadas

Una estructura se representa mediante un bloque de memoria que ha sido dividido en sub-bloques, de forma que cada sub-bloque tiene un tamaño y un tipo determinados, así como un nombre por el que podemos referirnos a él. Desde el punto de vista de la sintaxis, una estructura se declara como en el ejemplo siguiente:

```
struct libro{
    char *titulo;
    char *autor;
    int paginas;
    int precio;
};
```

De esta forma, es posible agrupar distintos tipos de datos (en este caso, dos enteros y dos punteros a carácter) relacionados lógicamente entre sí. La declaración anterior define un nuevo tipo de dato, de nombre `libro`, de tal forma que ahora pueden declararse variables de ese tipo:

```
struct libro biblioteca[1000];
```

Es posible también declarar variables del nuevo tipo antes de cerrar la declaración, como en

```
struct libro{
    char *titulo;
    char *autor;
    int paginas;
    int precio;
}un_libro,una_biblioteca[200];
```

en que se declara una variable del tipo `struct libro` y un array de variables del mismo. El acceso a los campos de la estructura se hace mediante el operador `'.'`. Por ejemplo:

```
un_libro.paginas=372;
un_libro.precio=24;
```

Pero la estructura anterior contiene también punteros, en este caso a carácter, que no requieren ningún tratamiento especial:

```
un_libro.titulo=(char *)malloc(50*sizeof(char));
strcpy(un_libro.titulo,"Aurelius Augustinus");
```

Esto abre la interesante posibilidad de que una estructura de un tipo contenga punteros a estructuras del mismo tipo, lo que permite crear listas. Consideremos por ejemplo:

```
struct libro{
    struct libro *siguiente;
    char *titulo;
    char *autor;
    int precio;
    int paginas;
}*lista;
```

Ahora `lista` es un puntero (al que no se ha asignado ningún valor, y por tanto no puede aún usarse) a una estructura que contiene, entre otras cosas, un puntero a otra estructura del mismo tipo. Así:

```
lista=(struct libro *)malloc(sizeof(struct libro));
```

reserva un bloque de memoria suficiente para contener un libro y apunta la variable `lista` a ese bloque. Para acceder a los campos de estructuras apuntadas se usa el operador `'->'`:

```
lista->precio=20;
lista->paginas=400;
lista->autor=(char *)malloc(50*sizeof(char));
```

Si hubiésemos declarado variables tipo `struct libro` estáticas, ¿cuántas de ellas habríamos declarado? Si pocas, pudieran no ser suficientes. Si muchas, pudieramos estar desperdiciando espacio. Pero al incluir en cada estructura un puntero a otra del mismo tipo, podemos construir una lista, reservando exactamente la cantidad de memoria que precisamos: ni más ni menos. Así, si después de haber almacenado el primer libro es preciso almacenar otro, creamos un nuevo nodo:

```
lista->siguiente=(struct libro *)malloc(sizeof(struct libro));
```

Dedicaremos más adelante un capítulo a la creación y manipulación de listas.

1.7. Punteros a funciones

Un puntero apunta a una dirección de memoria. Hasta ahora, hemos supuesto que en esa dirección está contenida una variable. Pero la memoria, aparte de datos, contiene código, y no hay diferencia cualitativa alguna entre aquellas áreas de memoria que contienen datos y las que contienen instrucciones. Se sigue que un puntero puede igualmente apuntar a código que a

datos. La cuestión es si, además de apuntar a código, sería posible a través del puntero ejecutar ese código. En C, esto se consigue declarando punteros a función.

Las dos utilidades principales de esta característica son:

1. Permite usar las funciones como parámetros de otras funciones. Un ejemplo clásico se encuentra en las funciones de ordenación. Una ordenación, de un vector por ejemplo, contiene dos mecanismos: una comparación entre elementos y un intercambio entre elementos. Pero la comparación puede hacerse según diversos criterios: comparación numérica, comparación alfabética o en caso de que se comparen datos complejos, como estructuras, según el valor de algún campo en concreto. Según esto, no sería posible escribir un algoritmo genérico de ordenación, ya que dependería del criterio de comparación. Pero si la función de ordenación recibe como parámetro un puntero a la función que efectúa la comparación, entonces el mismo algoritmo puede ejecutarse con distintos criterios.

   Otros ejemplos los podemos tomar del cálculo numérico. Una integración numérica depende obviamente de la función a integrar. Pero podemos escribir un algoritmo genérico de integración que reciba como parámetro un puntero a la función que ha de integrarse.

2. Permite aplicar la ejecución vectorizada. Para ilustrar esto, consideremos un intérprete de bytecode. El intérprete lee códigos y actúa en consecuencia. Cuando el bytecode leído es 1, se efectúa una acción; cuando es 2, otra, y así sucesivamente. Un intérprete para un pequeño lenguaje puede contener del orden de un centenar de códigos distintos, así que la estructura obvia:

   ```
   if (codigo==1){
       /* acciones */
   } else
   if (codigo==2){
       /* acciones */
   } else
   if (codigo==3){
       /* acciones */
   } ....
   ```

   puede contener unos centenares de líneas. El código no es ni elegante ni práctico, pues, por término medio, habrán de evaluarse la mitad de

> las condiciones cada vez. Sin embargo, imaginemos que disponemos de un vector de punteros a función. El valor de `codigo` puede usarse para indexar un elemento del vector y acceder al código correspondiente. El larguísimo `if ... else ... if ...` queda reducido a una sola línea de código, y el tiempo necesario para ejecutar la acción que corresponda es el mismo cualquiera que sea ésta.

Dicho esto, ¿cómo se declaran y usan punteros a funciones? Así:

```
int (*f)(int, int);
```

`f` es un puntero a una función que toma como parámetros dos enteros y devuelve otro. Como todo puntero, es preciso asignarle a `f` un valor, es decir, hacer que apunte a la dirección de una función. Por ejemplo

```
#include <stdio.h>

int (*f)(int, int);

int suma(int a, int b){
    return(a+b);
}
main()
{
    f=suma;
    printf("%d\n",(*f)(2,3));
    return;
}
```

y el resultado es la impresión del valor `5`. Como segundo ejemplo, vamos a declarar y usar un vector de dos punteros a función.

```
#include <stdio.h>

int (*f[2])(int, int);

int suma(int a, int b){
    return(a+b);
}
int prod(int a, int b){
    return(a*b);
}
```

```
main()
{
    f[0]=suma;
    f[1]=prod;
    printf("%d %d\n",(*f[0])(2,3),(*f[1])(2,4));
    return;
}
```

1.8. Declaraciones complejas

El hecho de que las declaraciones en C incluyan modificadores prefijos y postfijos simultáneamente puede conducir a confusión y error. Es por eso que daremos aquí las reglas precisas. Considérense las siguiente declaraciones:

1. `int a[10];`
2. `int **b;`
3. `int *c[10];`
4. `int (*d)[10];`
5. `int *e[10][20];`
6. `int (*f)[10][20];`

En el primer caso tenemos un vector de diez enteros; en el segundo, un puntero a un puntero a entero; en el tercero, un vector de diez punteros a entero; en el cuarto, un puntero a un vector de diez enteros; en el quinto, un vector de diez punteros a vector de veinte enteros; en el sexto, un puntero a un vector de 10×20 enteros. La comprensión de las declaraciones anteriores pasa por saber el orden en que se aplican los modificadores. Para ello se tendrán en cuenta las tres reglas siguientes:

1. La prioridad de un modificador es tanto mayor cuanto más cerca se encuentre del nombre al que modifica.
2. Los modificadores `()` y `[]` tienen mayor prioridad que `*`.
3. Los paréntesis agrupan partes de la declaración otorgándoles mayor prioridad.

Por ejemplo, para interpretar el tipo de `int *e[10][20]` vemos que los modificadores más cercanos son `*` y `[10]`, pero, por la segunda regla, `[10]` tiene mayor prioridad, de manera que `e` es un vector de diez punteros (ya que tras el `[10]` es `*`, por cercanía, quien tiene prioridad) a vectores de 20 enteros. En el caso de la declaración `int (*f)[10][20]` los paréntesis otorgan la máxima prioridad a `*`, de manera que `f` es un puntero, a un vector de 10×20 enteros. Aplicamos las mismas reglas para interpretar las declaraciones siguientes:

1. `char *g();`
2. `char (*h)();`
3. `char *i()[10];`
4. `char *j[10]();`

En el primer caso, tenemos una función (los paréntesis tienen mayor prioridad) que devuelve un puntero a carácter; en el segundo caso es el `*` quien tiene mayor prioridad, al estar asociado mediante paréntesis al nombre que se declara, de forma que tenemos un puntero, a una función que devuelve un carácter; en el tercer caso tenemos una función que devuelve un puntero a un vector de 10 caracteres; finalmente, tenemos un vector de 10 punteros a función que devuelve un carácter. Con esto es suficiente, y reconozcamos que son poco probables (y en todo caso el programador preferiría evitarlas) declaraciones como

```
char (*(*x())[])();
```

donde `x` es una función que devuelve un puntero a un vector de punteros a función que devuelve `char`, o como

```
char (*(*z[10])())[5];
```

donde `z` es un vector de 10 punteros a función que devuelve puntero a vector de cinco elementos de tipo `char`.

Capítulo 2

Punteros, cadenas y matrices

2.1. Cadenas de caracteres

En C, una cadena de caracteres es una sucesión de bytes que acaba con un carácter especial, indicando el fin de la cadena. Esta forma de representar una cadena tiene la ventaja de que no limita su tamaño, y dos inconvenientes: que la cadena no puede contener el carácter especial que indica el final y que para obtener su longitud es preciso recorrerla en su totalidad, hasta alcanzar el carácter especial que indica el fin. Por ejemplo, una implementación de `strlen()` es:

```
int strlen(char *p)
{
    int n=0;
    while (*(p+n)!='\0'){
        ++n;
    }
    return(n);
}
```

En esta implementación es evidente que a la cadena se accede a través de un puntero que apunta al primer carácter. Pero no deben confundirse ambas cosas, ya que un puntero a un carácter no es una cadena, sino la forma en que accedemos a ella. De hecho, podríamos acceder a la cadena no a partir del primer carácter, sino a partir de cualquier otro:

```
#include <stdio.h>
main()
{
```

```
    char cadena[]="Sancte Michele Arcangele";
    char *p=cadena;
    printf("%s\n%s\n%s\n",p,p+7,p+15);
}
```

Por otra parte, la siguiente implementación muestra que pueden restarse punteros:

```
int strlen(char *cadena)
{
    char *p=cadena;
    while (*p!='\0') ++p;
    return(p-cadena);
}
```

Como segundo ejemplo, consideremos una implementación de `strcmp()`. Esta función toma dos punteros a cadena y recorre ambas. Si alcanza el final y no encuentra diferencia entre ambas, devuelve un 0, y 1 en caso contrario:

```
int strcmp(char *a, char *b)
{
    char *p=a,*q=b;
    while (*p!='\0'){
        if (*p!=*q) return(1);
        ++p;
        ++q;
    }
    return(0);
}
```

Como ilustración de estos conceptos, escribiremos una función capaz de dividir una cadena en las palabras que la constituyen. Las palabras se encuentran separadas de otras palabras por espacios en blanco. Dispondremos de un vector de punteros a carácter, y haremos apuntar cada uno de los elementos del vector al principio de cada palabra. Esto no sería suficiente, pues de esta forma cada puntero apuntaría a una cadena que comienza al principio de la palabra correspondiente pero termina siempre al final de la cadena original. Entonces, será preciso insertar caracteres especiales de fin de cadena al final de cada palabra. Suponemos que una cadena contiene a lo sumo `MAX` palabras.

```
#include <stdio.h>
#define MAX 20
char *p[MAX];
void divide(char *cadena)
{
    int j;
    char *r=cadena;
    for(j=0;j<MAX;++j) p[j]=NULL;
    j=0;
    while (1){
        while (*r==' ') ++r;
        if (*r=='\0') return;
        p[j]=r;
        ++j;
        while (*r!=' '){
            if (*r=='\0') return;
            ++r;
        }
        *r='\0';
        ++r;
    }
}
main()
{
    int j=0;
    char cadena[]="defende nos in proelio";
    divide(cadena);
    while (p[j]!=NULL){
        printf("\n%2d  /%s/",j,p[j]);
        ++j;
    }
    printf("\n");
    return;
}
```

El funcionamiento de `divide()` consta esencialmente de un bucle principal que se ejecuta en tanto en cuanto no se alcance el final de la cadena. Dentro de él, están contenidos dos bucles secundarios. El primero salta espacios en blanco hasta encontrar uno distinto, que será el de comienzo de una palabra. Entonces, se hace apuntar allí uno de los punteros del vector de punteros a carácter. El segundo, avanza a lo largo de una palabra hasta que

encuentra un espacio en blanco, que es sustituido por un carácter de fin de cadena. El lector no tendrá dificultad en mejorar este sencillo procedimiento en tres aspectos. Primero, en manejar como separadores entre palabras los tabuladores, además de los espacios en blanco. Segundo, en tener en cuenta que la cadena puede terminar con un carácter nueva línea, en lugar de con el carácter de fin de cadena. Tercero, que las cadenas pueden contener subcadenas literales (generalmente delimitadas por comillas, simples o dobles) que contengan más de una palabra, pero que han de ser tratadas como una sola.

2.2. Matrices

Trataremos sobre matrices de tamaño arbitrario, ya que los vectores son casos particulares de matrices de una sola fila o una sola columna. Entonces, no existe diferencia cualitativa entre una cadena de caracteres y un vector de números de una fila. Al igual que accedemos a una cadena mediante un puntero a su primer carácter, accedemos a un vector de números mediante un puntero al primer número. Al contrario que con las cadenas, sin embargo, los vectores de números no terminan en un valor especial. O bien la dimensión del vector se encuentra en algún otro lugar o bien el primer elemento del vector contiene el número total de elementos, o bien la dimensión del vector es conocida por el programador.

```
#include <stdio.h>
main()
{
    int v[6]={1,2,3,4,5,6};
    int *w=v;
    int j;
    for(j=0;j<6;++j){
        printf("\n%d %d\n",v[j],*(w+j));
    }
}
```

Más interesantes son las matrices. Por ejemplo, el vector del programa anterior puede considerarse como un vector de una fila y seis columnas, pero también como una matriz de dos filas y tres columnas. En ese caso, las filas se almacenan en memoria una tras otra. Véase:

```
#include <stdio.h>
main()
{
```

```
    int v[2][3]={1,2,3,4,5,6};
    int *w=&(v[0][0]);
    int j;
    for(j=0;j<6;++j){
        printf("\n%d %d\n",v[j/3][j%3],*(w+j));
    }
}
```

Ahora hacemos apuntar `w` al primer elemento de la matriz. El bucle nos demuestra que los elementos se encuentran en memoria por filas, y una fila a continuación de la anterior. Sin embargo, ya no podemos referirnos al elemento `j` mediante `v[j]` sino que, siendo `v` una matriz, es preciso especificar fila y columna. En el ejemplo anterior, de la posición lineal respecto al principio de la matriz, tal como se almacena en memoria, obtenemos la fila y columna correspondientes. A la inversa, dada una fila y columna es posible obtener el desplazamiento lineal:

```
#include <stdio.h>
main()
{
    int v[2][3]={1,2,3,4,5,6};
    int *w=&(v[0][0]);
    int j,k;
    for(j=0;j<2;++j){
        for(k=0;k<3;++k){
            printf("\n%d %d\n",v[j][k],*(w+3*j+k));
        }
    }
}
```

Capítulo 3

Listas

3.1. Introducción

Las listas son las estructuras de datos más versátiles que existen. Una lista puede representar, obviamente, a una lista. Pero también a una pila, un árbol, un vector, una matriz. Esta versatilidad es la que inspiró al lenguaje LISP. No decimos claro está que la implementación mediante listas de cualquier estructura de datos sea la forma más adecuada. Por ejemplo, representar mediante listas una matriz es ineficiente. Lo que queremos es poner de manifiesto esta versatilidad.

Y las listas se construyen mediante punteros. Los punteros hacen de enlace entre un elemento de la lista y el siguiente y/o el anterior. A cada elemento de la lista le llamaremos «nodo». Así que cada nodo ha de contener cierta cantidad de información y al menos un puntero que apunte al nodo siguiente de la lista. Conviene con frecuencia tener acceso no sólo al nodo siguiente, sino también al anterior, y por eso incluiremos en cada nodo dos punteros: al nodo siguiente y al anterior.

Qué información se guarde en cada nodo, depende del problema que se quiera resolver. A título de ejemplo, en nuestra discusión guardaremos una cadena en cada nodo, de manera que declaramos a éste mediante una estructura de la forma:

```
struct nodo{
    struct nodo *ant;
    struct nodo *sig;
    char *cadena;
};
```

En general, no guardaremos el contenido del nodo en el mismo nodo, sino que éste contendrá un puntero a dicho contenido. El tipo del puntero

dependerá de qué se quiera guardar en el nodo. Esto tiene dos ventajas. Una «estética» y es que el nodo queda constituido por tres punteros, solamente. Otra «práctica», y es que cuando deseemos intercambiar dos elementos de la lista (p. ej. en el curso de una ordenación) no será preciso reasignar los ocho punteros involucrados (los dos enlace de cada uno de los dos punteros, mas los enlaces de los elementos anterior y posterior a cada uno) , sino que bastará intercambiar los puntero al contenido.

Accedemos a la lista a través del primero de sus nodos, de manera que una lista no es más que una variable del tipo puntero a `struct nodo`:

```
struct nodo *lista;
```

3.2. Iniciar la lista

Al declarar la varibles `struct nodo *lista`, no podemos asegurar si `lista` apunta a algún sitio en concreto: es algo que está indeterminado. Por tanto, la primera operación tras haber declarado un puntero a un nodo es declararlo como vacío, asignándole el valor `NULL`

```
struct nodo *lista=NULL;
```

Así, las operaciones subsiguientes de inserción y eliminación de elementos habrán de comprobar, en primer lugar, si la lista contiene algún nodo, o bien si se encuentra vacía.

3.3. Insertar y eliminar nodos

Consideremos en primer lugar la inserción de un nodo en la lista. La posición de inserción vendrá marcada por un puntero que apunta a un nodo que llamaremos el nodo actual, y podrá hacerse delante o detrás del nodo actual. Hay algunos casos particulares. En primer lugar, puede que el nodo que se desea insertar sea el primero de la lista. En ese caso, el puntero que enlaza con la lista será `NULL`: reservaremos espacio con `malloc()` para un nuevo nodo, apuntaremos allí el puntero y estableceremos como `NULL` los punteros del nuevo nodo a los nodos siguiente y anterior (que no existen todavía). Puede suceder, en segundo lugar, que el nodo a continuación del cual queremos realizar la inserción sea el último, en cuyo caso será `x->sig==NULL`. En este caso: creamos un nuevo nodo, lo enlazamos hacia atrás con el nodo actual y establecemos a `NULL` su enlace hacia adelante. Finalmente, puede suceder que el nodo actual tenga uno anterior y otro siguiente. En ese caso, nos servimos

de una variable temporal que hacemos apuntar a un nodo de nueva creación, y ajustamos los enlaces, como muestra la Figura 1. El código de inserción de un nodo a continuación del actual es el siguiente:

```
struct nodo *L_inserta_despues(struct nodo *x)
{
    struct nodo *y;
    if (x==NULL){
        x=(struct nodo *)malloc(sizeof(struct nodo));
        x->ant=NULL;
        x->sig=NULL;
    } else
    if (x->sig==NULL){
        x->sig=(struct nodo *)malloc(sizeof(struct nodo));
        x->sig->sig=NULL;
        x->sig->ant=x;
        x=x->sig;
    } else
    {
        y=(struct nodo *)malloc(sizeof(struct nodo));
        y->ant=x;
        y->sig=x->sig;
        x->sig->ant=y;
        x->sig=y;
        x=x->sig;
    }
    return(x);
}
```

En cualquier caso, la función de inserción de vuelve un puntero al nodo recién insertado. Apunta por tanto en cada momento al último elemento de la lista. Esto no es un inconveniente, sino una ventaja, ya que en general los elementos serán añadidos a partir del último nodo, que ya estará apuntado y no será preciso localizar. El otro extremo de la lista tampoco se perderá, ya que el nodo por donde comenzó a construirse la lista se identifica fácilmente por la condición de que `x->ant==NULL`. De la misma forma escribimos una función que inserta un nodo en la posición anterior al actual:

```
struct nodo *L_inserta_antes(struct nodo *x)
{
    struct nodo *y;
```

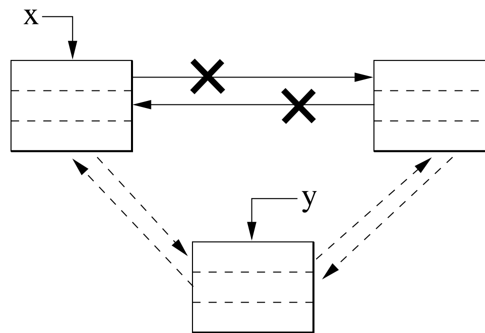

Figura 1

```
    if (x==NULL){
        x=(struct nodo *)malloc(sizeof(struct nodo));
        x->ant=NULL;
        x->sig=NULL;
    } else
    if (x->ant==NULL){
        x->ant=(struct nodo *)malloc(sizeof(struct nodo));
        x->ant->sig=x;
        x->ant->ant=NULL;
        x=x->ant;
    } else
    {
        y=(struct nodo *)malloc(sizeof(struct nodo));
        y->ant=x->ant;
        y->sig=x;
        x->ant->sig=y;
        x->ant=y;
        x=x->ant;
    }
    return(x);
}
```

Localizar los extremos de la lista a partir de un nodo dado es trivial:

```
struct nodo *L_inicio(struct nodo *x)
{
    while (x->ant!=NULL)
        x=x->ant;
```

```
    return(x);
}

struct nodo *L_fin(struct nodo *x)
{
    while (x->sig!=NULL)
        x=x->sig;
    return(x);
}
```

Pero es hora ya de hacer algo útil. En muchas ocasiones, las listas se construyen a partir de la información contenida en archivos. Para ilustrar estas ideas, escribimos una función que carga un archivo de texto en una lista, colocando una línea en cada nodo.

```
struct nodo *L_carga_archivo(struct nodo *x, char *archivo)
{
    FILE *f;
    char buffer[200];
    if ((f=fopen(archivo,"r"))==NULL) return(x);
    while (fgets(buffer,200,f)!=NULL){
        x=L_inserta_despues(x);
        x->cadena=(char *)malloc(1+strlen(buffer)*sizeof(char));
        strcpy(x->cadena,buffer);
    }
    fclose(f);
    return(L_inicio(x));
}
```

Una vez insertado un nuevo nodo, se reserva memoria para acomodar la cadena, que ha sido leída mediante `fgets()` en un buffer para el que hemos estimado suficientes 200 caracteres. En realidad, se reserva un byte más de la longitud de la cadena, puesto que ésta ha de llevar el carácter especial de fin de cadena al final. Y he aquí un pequeño programa que carga un archivo en una lista y luego imprime cada elemento:

```
#include <stdio.h>
main()
{
    struct nodo *z=NULL;
    z=L_carga_archivo(z,"listas.c");
```

```
    while (z->sig!=NULL){
        printf("%s",z->cadena);
        z=z->sig;
    }
    printf("%s",z->cadena);
    return;
}
```

Para eliminar un nodo de la lista hemos de decidir qué hacer con el puntero que apunta al nodo actual. Por analogía con el comportamiento de los editores de texto, y con el ejemplo mostrado anteriormente sobre un archivo de texto, vamos a hacer la siguiente implementación: si el nodo que se desea eliminar es el primero de la lista, se elimina y se devuelve un puntero al nodo siguiente, que pasa a ser el primero. Si el nodo a eliminar es el último, se elimina y se devuelve un puntero al nodo anterior. Si el nodo a eliminar tiene otros antes y después, se devolverá un puntero al nodo siguiente al eliminado. Finalmente, si la lista consistía en un sólo nodo, se devolverá `NULL`.

Es muy importante eliminar el contenido del nodo antes que el nodo mismo. Si no fuese así, el contenido apuntado desde el nodo quedaría inaccesible y la memoria ocupada por él no podría liberarse (al menos en tanto se ejecuta el programa), con gran contento de los detractores del uso de punteros. En nuestro caso, en que cada nodo contiene un puntero a una cadena, haríamos `free(x->cadena)`. En el caso más general, encapsularíamos esta operación en una funcion que tomase como argumento un puntero a nodo.

```
struct nodo *L_elimina_nodo(struct nodo *x)
{
    struct nodo *y;

    if ((x->ant==NULL)&&(x->sig==NULL)){
        free(x->cadena);
        free(x);
        x=NULL;
    } else
    if ((x->ant!=NULL)&&(x->sig!=NULL)){
        y=x->sig;
        x->ant->sig=x->sig;
        x->sig->ant=x->ant;
        free(x->cadena);
        free(x);
        x=y;
```

```
    } else
    if (x->ant==NULL){
        x=x->sig;
        free(x->ant->cadena);
        free(x->ant);
        x->ant=NULL;
    } else
    if (x->sig==NULL){
        x=x->ant;
        free(x->sig->cadena);
        free(x->sig);
        x->sig=NULL;
    }
    return(x);
}
```

Eliminamos la lista eliminando uno a uno los nodos hasta que no quede ninguno:

```
void L_elimina_lista(struct nodo *x)
{
    x=L_inicio(x);
    while (x!=NULL)
        x=L_elimina_nodo(x);
    return;
}
```

3.4. Fusión de listas

Fusionamos dos listas conectando la cola (último elemento) de una con la cabeza (primer elemento) de otra, tal y como se ve en la Figura 2.

La implementación es inmediata:

```
struct nodo *L_fusiona(struct nodo *x, struct nodo *y)
{
    x=L_fin(x);
    y=L_inicio(y);
    x->sig=y;
    y->ant=x;
    return(L_inicio(x));
}
```

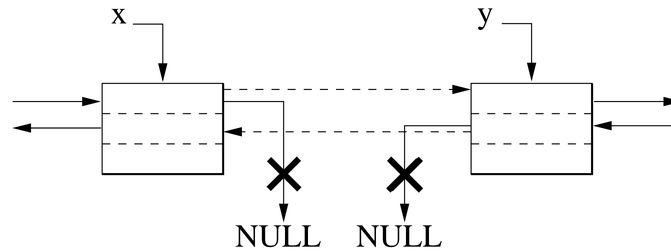

Figura 2

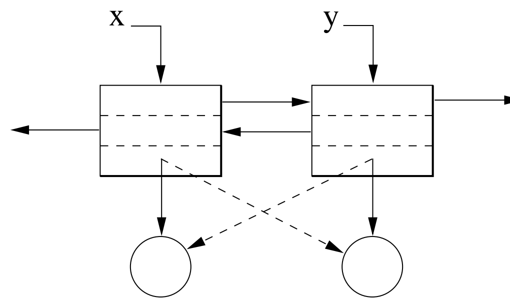

Figura 3

3.5. Intercambio de elementos y ordenación de listas

La ordenación de una lista descansa en la operación elemental de intercambiar dos elementos. Pero, al contrario de lo que sucede con la ordenación de arreglos, donde es preciso mover físicamente los datos, en nuestro caso no es necesario, ya que cada nodo no contiene dato alguno, sino un puntero a dato. Por tanto, intercambiar los contenidos de dos nodos se resuelve intercambiando los punteros a los datos, y no es preciso mover estos, como muestra la Figura 3.

Por tanto, el intercambio se realiza fácilmente, y con independencia de qué cantidad o tipo de datos pertenezcan a cada nodo:

```
void L_intercambia(struct nodo *x, struct nodo *y)
{
    char *z;
    z=x->cadena;
    x->cadena=y->cadena;
    y->cadena=z;
```

```
    return;
}
```

Una vez que sabemos cómo intercambiar dos nodos (los datos apuntados por ellos), es fácil implementar algún algoritmo de ordenación. Como estas páginas tratan de punteros, no de algorítmica, implementaremos el algoritmo obvio de ordenación por el método de la *burbuja*:

```
void L_ordena(struct nodo *x)
{
    struct nodo *a,*b,*i;
    a=L_inicio(x);
    b=L_fin(x);
    while (a!=b){
        i=a->sig;
        while (i!=NULL){
            if (strcmp(i->cadena,a->cadena)<0)
                L_intercambia(a,i);
            i=i->sig;
        }
        a=a->sig;
    }
    return(L_inicio(x));
}
```

El algoritmo de la burbuja es el más sencillo, y se comprende de inmediato. También es el más lento, por lo cual suele ser descartado en favor de otros más rápidos. Ahora bien, el mundo de la alta velocidad tiene sus peligros, pues se suele olvidar que la velocidad se compra con espacio (y a la inversa, el espacio se paga con tiempo). Cuando el número de elementos a ordenar es pequeño (hasta unos pocos miles de elementos), la diferencia entre usar quicksort y *burbuja* es inapreciable. El primero es más interesante cuanto mayor es el número de elementos a ordenar. Pero si este número es muy alto, quicksort puede fallar miserablemente. En nuestro sistema, ordena en unos 20 ms un vector de 15357 elementos. Pero 15358 hacen que el mismo programa quede detenido. Suponemos que se agotó el espacio en la pila por exceso de llamadas recursivas, y el sistema operativo no puede o no sabe resolver la situación. Pero *burbuja* sigue funcionando perfectamente, si bien consume aproximadamente un segundo y medio en ordenar los 15358 elementos. N. Wirth [1] ofrece una versión iterativa de quicksort. Remitimos allí al lector

[1] NIKLAUS WIRTH, *Estructuras de datos + algoritmos = programas*

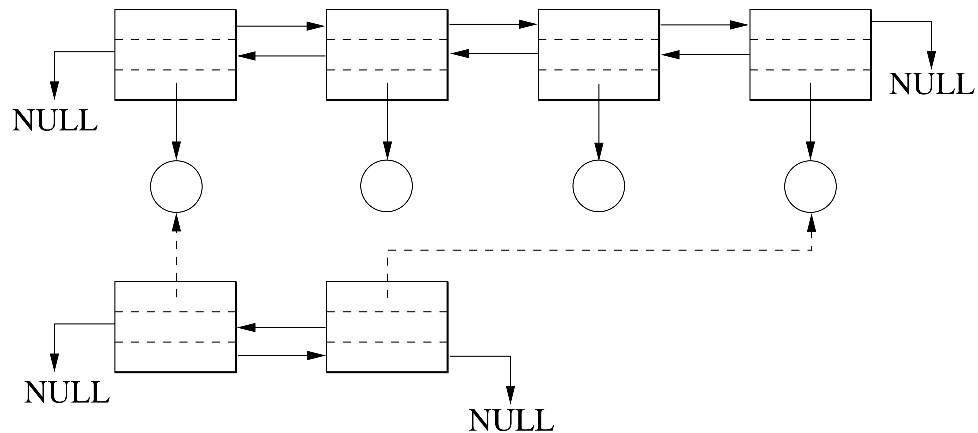

Figura 4

interesado. Y de todas formas, existen otros algoritmos, como la ordenación *Shell*, no tan rápidos como quicksort, pero más seguros y mucho más rápidos que el de la burbuja.

3.6. Extracción de sub-listas

Hay dos formas en que puede construirse una sub-lista a partir de una lista dada. La primera es destructiva, y consiste simplemente en eliminar los nodos que no cumplan la condición deseada. Para ello ya disponemos de la funcion que elimina un nodo dado. La segunda es no-destructiva, y procede extrayendo los elementos de la lista que satisfacen la condición deseada y agregándolos a otra lista, la sub-lista que deseamos obtener. Aquí trataremos de este segundo caso. A su vez, éste tiene dos variantes: en la primera, cada nodo de la sub-lista apunta a un bloque nuevo, creado para copiar en él los datos apuntados por el nodo correspondiente en la lista principal; en la segunda, los nuevos nodos apuntan a los datos ya existentes, apuntados por los nodos de la lista principal, tal y como muestra la Figura 4.

El problema de esta segunda opción es que cuando sea preciso liberar las listas, habrá que seguir un procedimiento *ad hoc* para evitar querer liberar dos veces los datos apuntados en común por la lista y la sublista. En el siguiente ejemplo, a partir de la lista `x` se construye la sub-lista `s`, de tal

forma que esta última está constituida por las cadenas de longitud inferior a 20 apuntadas por la lista principal.

```
struct nodo *L_sublista(struct nodo *x, struct nodo *s)
{
    /*
    apunta mediante la sublista a los nodos que
    contienen cadenas de longitud inferior a 20
    */
    struct nodo *y;
    y=L_inicio(x);
    while (y!=NULL){
        if (strlen(y->cadena)<20){
           s=L_inserta_despues(s);
           s->cadena=(char *)malloc(strlen(y->cadena)+1);
           strcpy(s->cadena,y->cadena);
        }
        y=y->sig;
    }
    return(L_inicio(s));
}
```

Capítulo 4

Árboles

4.1. Introducción

Un árbol es una estructura jerárquica formada por nodos de los que parten *ramas* que conducen a otros nodos. De cada nodo pueden partir dos o más ramas. Aquí nos limitaremos al caso en que de un nodo parten sólo dos ramas. Se habla entonces de árboles binarios. El concepto de árbol, ilustrado en la Figura 6, nos parece lo suficientemente claro como para no dar una descripción formal, que de todas formas puede encontrarse en los textos al uso. Adviértase, Figura 5, que la estructura del nodo de una lista es la misma que la del nodo de un árbol, si bien las representamos, y las usamos, de distinta forma. Por eso también a los punteros que enlazan unos nodos con otros no los llamamos *anterior* y *siguiente*, o de cualquier forma similar, sino *izquierda* y *derecha*, o similar. Hemos indicado los punteros `NULL` rellenando en negro el rectángulo izquierdo o derecho (o ambos), por claridad en el dibujo. Al nodo etiquetado con la letra "R" le llamamos *raíz*. Como cada nodo puede considerarse raíz para la estructura que *cuelga* de él, se sigue que los algoritmos que tratan con árboles son naturalmente recursivos. Dado que la estructura de los nodos en árboles y listas es la misma, podríamos preguntarnos el motivo por el que, en el capítulo anterior, las listas no han sido tratadas recursivamente. Simplemente, no era necesario, pero las listas se prestan igualmente bien al tratamiento recursivo. Por otro lado, en la implementación de listas del capítulo anterior no es preciso mantener un puntero al primer elemento de la lista, puesto que ésta se puede recorrer en un sentido u otro. Bastan un par de funciones para desplazarnos desde un nodo cualquiera al primero o al último. Sin embargo, en la implementación de este capítulo un nodo no tiene enlace con el nodo del que *cuelga*, así que será preciso mantener siempre un puntero al nodo raíz.

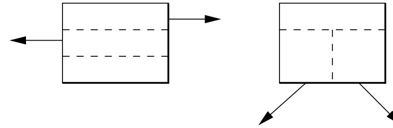

Figura 5

Si en el capítulo dedicado a listas ilustramos su uso con nodos que apuntan a cadenas, en éste usaremos nodos que contienen números enteros. Como el tamaño de un entero y el de un puntero es el mismo, no hay ventaja entre apuntar al dato desde el nodo y guardar el dato en el mismo nodo, así que declaramos

```
struct nodo{
    struct arbol *izq;
    struct arbol *der;
    int valor;
};
```

y un árbol comienza siendo, simplemente

```
struct nodo *arbol=NULL:
```

Al contrario de lo que sucede con las listas, en que normalmente los nodos se añaden secuencialmente (aunque luego puedan ejecutarse operaciones de reordenación), en los árboles necesariamente se ha de elegir dónde colocar cada nuevo elemento. Por su naturaleza de estructura jerárquica, cuando es preciso un árbol se tiene ya un criterio por el que se insertarán nuevos nodos. Por ejemplo, dado el nodo raíz, que contiene un número, se puede decidir colocar los números menores a ese en la rama izquierda, y los mayores a la derecha. Si todos los niveles se encontrasen llenos, con punteros vacíos sólo en el último nivel, el primer nivel contendría dos nodos, el segundo cuatro, el tercero ocho, y el n, 2^n nodos. De la misma forma, para alcanzar un nodo del primer nivel sólo tendríamos que decidir, a partir del raíz, si viajar a izquierda o derecha. Para alcanzar un nodo del segundo nivel sólo son precisos dos saltos, tres para el tercer nivel y así sucesivamente.

Pero puesto que el árbol es una estructura que se construye dinámicamente, nunca podemos asegurar que se encontrará equilibrado (con el mismo número de nodos a izquierda y derecha de cualquier nodo dado). Al contrario,

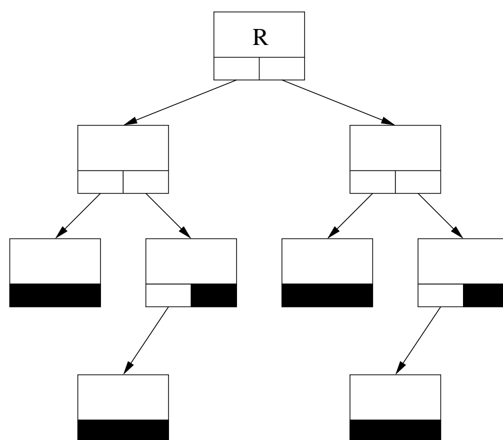

Figura 6

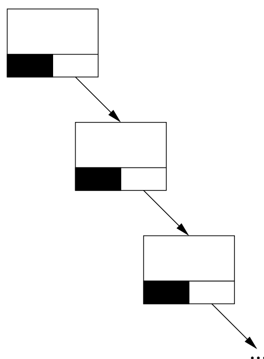

Figura 7

hay casos en que puede encontrarse totalmente desequilibrado, habiendo degenerado en una lista, como se muestra en la Figura 7. Así que su utilidad depende en gran medida de que los nodos se encuentren lo más simétricamente distribuidos.

4.2. Inserción de nodos

La naturaleza recursiva del árbol permite expresar de forma sencilla, recursiva, la operación de inserción. Si el árbol está vacío, simplemente se reserva un nodo, se apuntan `izq` y `der` a `NULL` y se coloca el valor en el nodo. Si el nodo no está vacío, se decide si el nuevo valor se colocará a izquierda o a derecha. Si el lado correspondiente se encuentra a `NULL`, se crea un nuevo nodo y se coloca allí el valor, si no está vacío, se llama recursivamente a la

función de inserción pasándole como argumento *la dirección del puntero* al nodo del siguiente nivel. El texto en cursiva es importante. Si la función de inserción tomase como argumento un puntero a nodo, cuando se produjese una llamada recursiva no se usaría el puntero a nodo que se quiere modificar, sino una copia del mismo. El original quedaría inalterado. Por eso la función de inserción tomará como argumento la dirección del puntero a nodo: para modificar el puntero propiamente, y no una copia que se pasa por valor.

```
void insertar(struct nodo **a, int n)
{
    if (*a==NULL)
    {
        *a=(struct nodo *)malloc(sizeof(struct nodo));
        (*a)->valor=n;
        (*a)->izq=NULL;
        (*a)->der=NULL;
    }
    else
    {
        if (n<(*a)->valor)
            insertar(&((*a)->izq),n);
        else
            insertar(&((*a)->der),n);
    }
    return;
}
```

Usaremos un vector de siete elementos para construir un árbol, insertando cada número en un nodo. Si seguimos el criterio de que a partir de un nodo el siguiente número se coloca a izquierda o derecha según sea menor o mayor que el valor contenido en el nodo, la secuencia `8,6,10,4,7,9,12` generará el árbol de la Figura 8. En `main()` no hemos olvidado dar al puntero al nodo raíz el valor inicial `NULL`.

```
main()
{
    struct nodo *a=NULL;
    int v[7]={8,6,10,4,7,9,12};
    int j;
    for(j=0;j<7;++j)
        A_inserta(&a,v[j]);
```

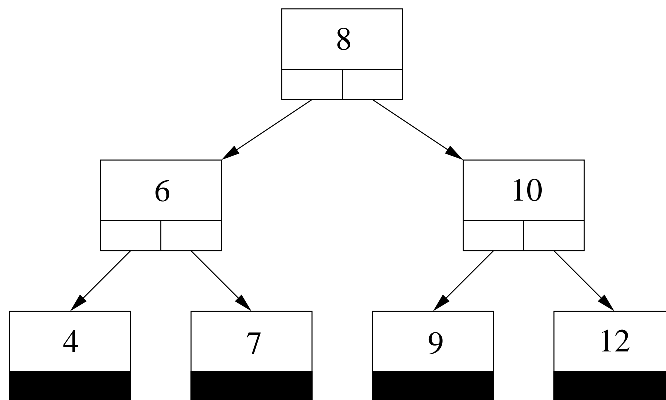

Figura 8

```
    return;
}
```

Se presenta ahora el problema de recorrer el árbol e imprimir (o cualquier otra operación) los valores contenidos en los nodos. Aprovechamos nuevamente la estructura recursiva del árbol y escribimos la siguiente función:

```
void A_print_in(struct nodo **a)
{
    if ((*a)->izq!=NULL){
        A_print_in( &((*a)->izq));
    }
    printf("%d\n",(*a)->valor);
    if ((*a)->der!=NULL){
        A_print_in( &((*a)->der));
    }
}
```

Si volvemos a la Figura 8 y hacemos una traza de la `A_print_in`(), vemos que el resultado es la impresión en orden creciente de los valores contenidos en el árbol. A este tipo de recorrido en profundidad del árbol se le llama *in-orden*. Pero podemos elegir imprimir en primer lugar el valor del nodo y luego ocuparnos de las ramas, y en ese caso hablamos de un recorrido en *pre-orden*; o bien ocuparnos de las ramas en primer lugar, y finalmente del nodo. Tenemos entonces un recorrido en *post-orden*

```
void A_print_pre(struct nodo **a)
{
    printf("%d\n",(*a)->valor);
    if ((*a)->izq!=NULL){
        A_print_pre( &((*a)->izq));
    }
    if ((*a)->der!=NULL){
        A_print_pre( &((*a)->der));
    }
}

void A_print_post(struct nodo **a)
{
    if ((*a)->izq!=NULL){
        A_print_post( &((*a)->izq));
    }
    if ((*a)->der!=NULL){
        A_print_post( &((*a)->der));
    }
    printf("%d\n",(*a)->valor);
}
```

4.3. Encontrar y eliminar nodos

Puesto que un nodo se inserta siguiendo un criterio, basta seguir ese mismo criterio para averiguar su posición dentro del árbol. Y como un nodo se identifica por el valor que contiene, la función de búsqueda tomará como argumento dicho valor, devolviendo un puntero a `NULL` si dicho valor no se encuentra en el árbol y al nodo correspondiente en caso contrario:

```
struct nodo *A_buscar_rec(struct nodo **a, int n)
{
    if ((*a)==NULL)
        return(NULL);
    else
    if ((*a)->valor==n)
        return(*a);
```

```
    else
    {
        if (n < (*a)->valor)
            A_buscar_rec(&((*a)->izq),n);
        else
            A_buscar_rec(&((*a)->der),n);
    }
}
```

Apenas hay ventaja entre escribir la función de forma recursiva y hacerlo de forma iterativa:

```
struct nodo *A_buscar_it(struct nodo **a, int n)
{
    struct nodo *b=(*a);
    while (b!=NULL){
        if (b->valor==n)
            return(b);
        else
        if (n< b->valor)
            b=b->izq;
        else
            b=b->der;
    }
    return(b);
}
```

Es más: la versión recursiva, tal y como la hemos escrito, no funciona. El código parece correcto, pero ilustra un error común; a saber: el valor devuelto por una función recursiva no es usado por la función que la invoca, excepto en la primera instancia. Como en general una función recursiva se llama a sí misma varias veces, el valor devuelto por la segunda, tercera y sucesivas instancias no lo es a la función invocadora, sino a la instancia anterior, que en nuestro caso no hace nada con él. Por ese motivo, el valor devuelto será, siempre, el de la primera instancia. La solución consiste en declarar una variable pública que pueda ser modificada por una instancia cualquiera de la función recursiva, que se declara como `void`:

```
struct nodo *busca=NULL;
void A_buscar_rec2(struct nodo **a, int n)
{
```

```
    if ((*a)==NULL)
        busca=NULL;
        return;
    else
    if ((*a)->valor==n)
        busca=(*a);
        return;
    else
    {
        if (n < (*a)->valor)
            A_buscar_rec2(&((*a)->izq),n);
        else
            A_buscar_rec2(&((*a)->der),n);
    }
}
```

Es útil también una función que, dado un nodo, devuelva un puntero al nodo padre, ya que éste es único y será necesario tenerlo en cuenta para eliminar nodos. La implementación es directa, tanto en forma recursiva (con la salvedad comentada antes) como en forma iterativa. Como carece de sentido buscar el padre de un nodo que no existe, supondremos que el nodo pertenece al árbol, devolviendo `NULL` si se trata del nodo raíz o un puntero al nodo padre, en otro caso.

```
struct nodo *busca_padre;
void A_buscar_padre_rec(struct nodo **a, struct nodo *x)
{
    int n=x->valor;
    if (*a==x){
        busca_padre=NULL;
        return;
    } else
    if (n < (*a)->valor){
        if ((*a)->izq==x){
            busca_padre=(*a);
            return;
        } else
            A_buscar_padre_rec(&((*a)->izq),x);
    } else
    {
        if ((*a)->der==x){
```

```
            busca_padre=(*a);
            return;
        } else
            A_buscar_padre_rec(&((*a)->der),x);
    }
}
```

El borrado de un nodo puede ser sencillo o difícil, según donde esté el nodo. El caso más simple se da cuando un nodo no tiene descendientes, es decir, sus punteros `izq` y `der` se encuentran a `NULL`. En ese caso, basta eliminarlo y poner a `NULL` el puntero que apuntaba a él. Es más complejo cuando el nodo es interior al árbol, y tiene tanto padre como hijos. Renunciamos a discutir ese caso, ya que éstas notas no son sobre algorítmica, sino sobre punteros. No obstante, existe una solución muy sencilla, aunque poco eficiente, que consiste en recorrer el árbol en *pre-orden* y reconstruirlo en otro lugar, omitiendo el nodo que se desea eliminar. Una vez hecho, se procede al borrado del árbol original. Esta operación es fácil si se recorre el árbol en *post-orden*:

```
void A_borrar_arbol(struct nodo **a)
{
    /* borrar en post-orden */
    if ((*a)->izq!=NULL){
        A_borrar_arbol( &((*a)->izq));
    }
    if ((*a)->der!=NULL){
        A_borrar_arbol( &((*a)->der));
    }
    free(*a);
}
```

Capítulo 5

Punteros en otros contextos

5.1. Concepto

Hasta ahora hemos tratado con punteros como un tipo de dato en C. Ahora bien, este tipo de dato es la implementación del concepto de variable que contiene la dirección de un objeto en una secuencia. Si atendemos únicamente a esta definición, un puntero se puede reducir a un número entero que indica una posición. Y este concepto lo podemos encontrar en algunos lugares, aparte de la memoria.

5.2. FAT e i-nodos

Un disco se puede ver como una secuencia de bloques de igual tamaño. Cada bloque tiene usualmente un tamaño en bytes que es una potencia de 2. Cada bloque es la unidad mínima de lectura/escritura. El problema consiste en dar una estructura lógica a una tal colección de bloques de forma que sea apta para guardar archivos, acceder a ellos, borrarlos y modificarlos. De entre las numerosas técnicas posibles, expondremos dos de ellas: los sistemas FAT y los sistemas de i-nodos. Para motivar ambos sistemas, consideremos la situación en que los archivos se escriben en forma de secuencia continua de bytes, pudiendo ocupar un número cualquiera de bloques. La Figura 9 ilustra el caso donde se ha escrito un archivo A y después de él un archivo B. Si ahora se desea modificar el primero, incrementando su tamaño, sería necesario trasladarlo entero tras el segundo, lo cual es una operación lenta e ineficiente. Mejor sería poder colocar el añadido A' en otro lugar, de forma que el archivo consta ahora de dos fragmentos físicamente disjuntos pero lógicamente contiguos.

Si ahora borramos B y lo sustituimos por otro archivo de mayor tamaño,

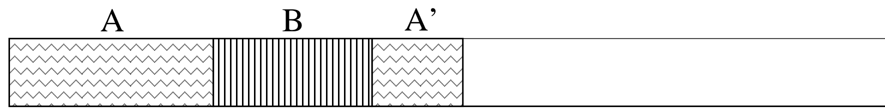

Figura 9

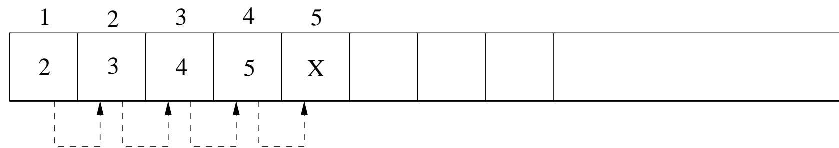

Figura 10

quisieramos poder aprovechar ese hueco, colocando ahí parte de C, y el resto a continuación de A'. En definitiva, deseamos un sistema que permita dividir un archivo en un número de fragmentos físicamente disjuntos, y poder asignar un hueco cualquiera libre a un archivo cualquiera. Obsérvese que decimos *permita*. Los archivos podrán, a pesar de todo, si las circunstancias lo permiten, seguir escribiéndose como una única sucesión física de bloques.

5.2.1. FAT

Si N es el número de bloques que se desean administrar, la solución FAT consiste en mantener un vector de N punteros. Consideremos el momento en que aún no se ha guardado información alguna. Todos los bloques se encuentran libres, y eso lo indicaremos escribiendo un número especial en cada elemento del vector. Ahora, imaginemos que escribimos un archivo que va a ocupar los bloques números { `1,2,3,4,5`}. Lo indicaremos formando una lista enlazada de punteros en la FAT, escribiendo un número `2` en la entrada `1`, un `3` en la entrada `2`, un `4` en la entrada `3`, un `5` en la entrada `4` y finalmente un número especial en la entrada `5`, que en la Figura nombramos como `X`, indicando así que ésta es la última. La lista se representa en la Figura 10.

Pero más claramente se aprecia el mecanismo cuando un archivo no ocupa bloques consecutivos. Imaginemos que han quedado libres los bloques {`7,9,22,111,204`}, y que van a ser usados para alojar un nuevo archivo. En ese caso, en la entrada `7` de la FAT escribiremos un `9`, en la entrada `9` un `22` y así sucesivamente, como muestra la Figura 11.

Así, las entradas de la tabla FAT cumplen dos funciones simultáneamente: indican si el bloque correspondiente está libre u ocupado y dado un bloque ocupado, sea el número `x`, la entrada `x` de la FAT indica cual es el bloque

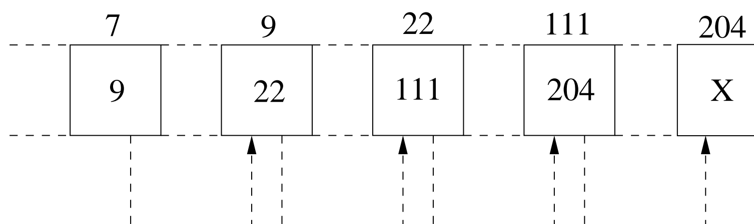

Figura 11

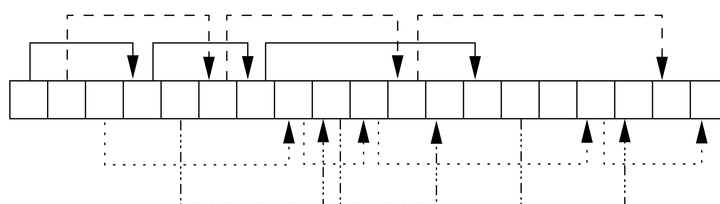

Figura 12

siguiente del archivo, o si es el último bloque. Queda la cuestión de cómo saber dónde comienza la lista de bloques asignada a un archivo. Para eso existe el directorio raíz. El directorio raíz consiste en una serie de bloques reservados donde se escriben los nombres de los archivos y el primer bloque que ocupa cada uno (entre otras informaciones que no vienen al caso, como tamaño, fecha, etc). Ahora bien, el directorio raíz tiene un tamaño limitado, luego ahí no pueden estar los nombres de todos los archivos de un volumen (disco). En efecto, no están ahí. Algunas de las entradas del directorio raíz no son archivos como los otros, sino un tipo especial de archivo llamado *subdirectorio*. Un subdirectorio es un archivo que contiene entradas con nombres de archivos u otros subdirectorios y, entre otras cosas, el primer bloque de cada archivo. A diferencia del directorio raíz, un subdirectorio, como cualquier otro archivo, puede crecer ocupando nuevos bloques, de manera que, aunque el directorio raíz pueda contener sólo un número limitado de entradas, también puede contener subdirectorios que a su vez pueden contener un número ilimitado (limitado por el tamaño del disco) de entradas.

5.2.2. i-nodos

Si imaginamos un disco como una sucesión de bloques ocupados por archivos (decenas de miles en los sistemas actuales), entonces la imagen de la FAT es la de decenas de miles de listas enlazadas, ocupando el mismo espacio y ofreciendo una imagen tan poco satisfactoria como la de la Figura 12.

Lo que es peor, para seguirle la pista a un archivo es preciso leer toda la

FAT, es decir, leer las entradas correspondientes a todos los archivos que no son el de nuestro interés. La solución obvia sería colocar juntos todos los punteros correspondientes a un archivo dado. Así, la entrada del directorio raíz de un archivo contendría el nombre del mismo, tamaño, fecha... y el número del bloque donde están contenidos los números de los bloques que ocupa el archivo. A este bloque que contiene los números de bloques ocupados por un archivo se le llama *i-nodo*. Puede suceder que un i-nodo no pueda contener todos los números de bloque ocupados por un archivo, si éste es muy grande. Por ejemplo, con bloques de 1K y enteros de 32 bits, un bloque *i-nodo* puede contener 256 entradas correspondientes a otros tantos bloques. Esto limitaría el tamaño de los archivos a 256K. Hay dos soluciones: la primera consiste en tomar bloques de mayor tamaño. Por ejemplo, con bloques de 8K el tamaño máximo de archivo es de 16MB. Pero, si el tamaño de bloque es muy grande, puede desperdiciarse una cantidad inaceptable de espacio en disco pues, por término medio, queda vacía la mitad del último bloque de cada archivo. La otra solución consiste en reservar algunas entradas del *i-nodo* que contengan no números de bloque pertenecientes al archivo, sino número de *i-nodo* donde hay más números de bloque pertenecientes al archivo. Y si este es muy grande, podemos tener incluso entradas que apunten a bloques que contienen números que apuntan a otros *i-nodos* que finalmente apuntan a los bloques del archivo. Esta situación se ilustra en la Figura 13, donde la penúltima entrada de un *i-nodo* apunta a otro *i-nodo*, y la última apunta a un bloque cuyas entradas a su vez apuntan a *i-nodos*. Con este esquema, y bloques de 1K, cada uno de los cuales contendría 256 entradas, el *i-nodo* contendría 254 entradas directas (números de bloque pertenecientes al archivo); más una entrada apuntando a un bloque que contendría otras 256 entradas que contendrían los números de otros tantos bloques, mas una entrada que apuntaría a un bloque con 256 entradas apuntando a 256 bloques cada uno de los cuales apuntaría a 256 bloques de archivo. En total entonces el archivo podría tener hasta $254 + 256 + 256^2$ bloques, lo que hacen un total de aproximadamente 66MB. Este tamaño puede doblarse si se dobla el tamaño del bloque, o se puede, en lugar de reservar las dos últimas entradas del *i-nodo*, resevar las tres últimas, de modo que la última sirva para indirección triple (contiene punteros a nodos, que contienen punteros a nodos, que contienen punteros a nodos que contienen números de bloque del archivo).

Un problema de este sistema es que no es posible saber qué nodos se encuentran ocupados y qué nodos libres. Entonces, será preciso reservar unos cuantos bloques para implementar sobre ellos un mapa de bits. Si el disco contiene n bloques de datos, será preciso guardar en algún lugar una secuencia de n bits. El bit x de esa secuencia a 1(0) indica que el bloque x de datos se encuentra ocupado(libre).

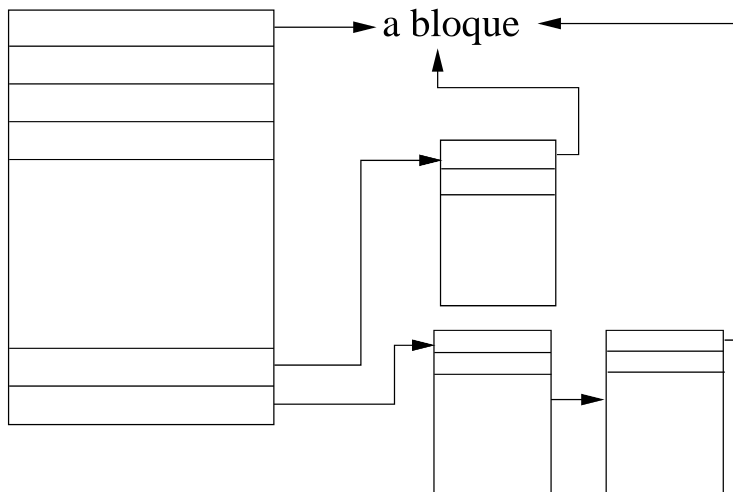

Figura 14

5.3. Gestión de memoria

Aunque la gestión de memoria cuenta en los procesadores modernos con un importante apoyo de *hardware*, no estará de más una discusión sencilla de cómo puede implementarse un gestor de memoria. Cuando hacemos una llamada a `malloc` y solicitamos una cierta cantidad de memoria, el gestor ha de localizar un bloque del tamaño adecuado y devolver su dirección. Cuando llamamos a `free` el gestor ha de liberar de alguna forma esa memoria antes ocupada. Una forma de hacerlo es manteniendo dos listas: una de bloques ocupados y otra de bloques libres. Cuando se quiere reservar un bloque, buscamos en la lista de libres uno de tamaño igual o mayor a la cantidad solicitada. Si se encuentra un bloque de tamaño justo el que se ha pedido se pasa a la lista de ocupados y se devuelve la dirección. Si se encuentra un bloque de tamaño mayor, se divide en dos partes: una que queda libre y otra, del tamaño solicitado, que se pasa a la lista de bloques ocupados. Los textos de sistemas operativos discuten en profundidad estas cuestiones, y en particular si, dada una solicitud, es mejor satisfacerla con el bloque más pequeño que pueda ser o con el más grande, o con el primero que se encuentre capaz de satisfacer la solicitud. No entraremos aquí en esa discusión. Liberar un bloque por otra parte no es más que pasarlo a la lista de bloques libres.

Así que, en esencia, el problema que subyace es el de la implementación de listas sobre secuencias de bytes. En el caso sencillo de una lista simple enlazada es suficiente una pequeña cabecera de un entero por cada elemento de la lista, que indique el comienzo del elemento siguiente. De aquí se sigue el tamaño del elemento actual (supuesta conocida su propia posición). La

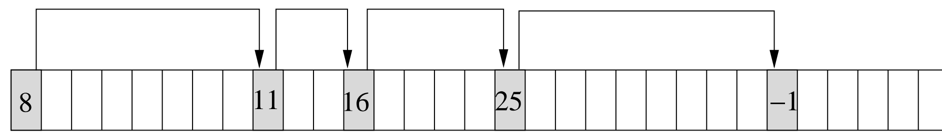

Figura 15

8	6	10
6	4	7
10	9	12
4	−	−
7	−	−
9	−	−
12	−	−

Figura 16

situación se ilustra en la Figura 15, donde cada cabecera, sombreada, contiene la posición del elemento siguiente, supuesto que la posición del primer elemento es la 0.

La propia representación gráfica sugiere ya que las operaciones de inserción y borrado de nuevos elementos es bastante directa. También las estructuras de tipo árbol pueden implementarse fácilmente mediante *punteros manuales*, es decir, mediante índices enteros que conceptualmente son punteros pero que desde el punto de vista de la implementación no son más que números enteros que indican posiciones. Por ejemplo, el árbol que aparece en la página 39 puede implementarse mediante una matriz como ilustra la Figura 16.

5.4. Conclusión

El dominio de los punteros es una habilidad de programación que tiende a quedar relegada debido a la preeminencia de los lenguajes de muy alto nivel y lenguajes de script. Pero estructuras de datos básicas como listas y árboles están basadas en punteros, sin contar con que esos mismos lenguajes de alto nivel están implementados haciendo un uso intensivo de los punteros. Y si vamos algo más abajo, al puntero no como tipo de dato en C sino como entero que indica una posición, los encontramos en otras partes de cualquier sistema.

Los inconvenientes clásicos que se oponen a su uso, como pérdidas de

memoria y violaciones de segmento, se solucionan teniendo presentes dos sencillas recetas: a) todo `malloc()` ha de tener su correspondiente `free()` y b) nunca usar un puntero sin saber a qué lugar apunta.

Estas notas no han pretendido cubrir ni la programación en lenguaje C ni la algorítmica ni las estructuras de datos; sin embargo, espero que aquellos a quienes van dirigidas puedan encontrarlas útiles, ya en relación la asignatura de Periféricos, aunque tangencialmente, ya en relación con cualquier otra.

Explicit libellus.