CAPÍTULO 5

ESTRUCTURAS DE DATOS

En la práctica, la mayor parte de información útil no aparece aislada en forma de datos simples, sino que lo hace de forma organizada y estructurada. Los diccionarios, guías, enciclopedias, etc., son colecciones de datos que serían inútiles si no estuvieran organizadas de acuerdo con unas determinadas reglas. Además, tener estructurada la información supone ventajas adicionales, al facilitar el acceso y el manejo de los datos. Por ello parece razonable desarrollar la idea de la agrupación de datos, que tengan un cierto tipo de estructura y organización interna.

Como tendremos ocasión de ver, la selección de una estructura de datos frente a otra, a la hora de programar es una decisión importante, ya que ello influye decisivamente en el algoritmo que vaya a usarse para resolver un determinado problema. El objetivo de este capítulo no es sólo la descripción de las distintas estructuras, sino también la comparación de las mismas en términos de utilidad para la programación. De hecho, se trata de dar una idea, acerca de los pros y contras de cada una de ellas con el propósito final de justificar la ya citada ecuación de:

PROGRAMACION = ESTRUCTURAS DE DATOS + ALGORITMOS

5.1. EL CONCEPTO DE DATOS ESTRUCTURADOS.

Empecemos recordando que un dato de tipo simple, no esta compuesto de otras estructuras, que no sean los bits, y que por tanto su representación sobre el ordenador es directa, sin embargo existen unas operaciones propias de cada tipo, que en cierta manera los caracterizan. Una **estructura de datos** es, a grandes rasgos, una colección de datos (normalmente de tipo simple) que se caracterizan por su organización y las operaciones que se definen en ellos. Por tanto, una estructura de datos vendrá caracterizada tanto por unas ciertas relaciones entre los datos que la constituyen (p.e., el orden de las componentes de un vector de números reales), como por las operaciones posibles en ella. Esto supone que podamos expresar formalmente, mediante un conjunto de reglas, las relaciones y operaciones posibles (tales como insertar nuevos elementos o como eliminar los ya

existentes). Por el momento y a falta de otros, pensemos en un vector de números, como el mejor ejemplo de una estructura de datos.

Llamaremos **dato de tipo estructurado** a una entidad, con un solo identificador, constituida por datos de otro tipo, de acuerdo con las reglas que definen cada una de las **estructuras de datos.** Por ejemplo: una cadena esta formada por una sucesión de caracteres, una matriz por datos simples organizados en forma de filas y columnas y un archivo, está constituido por registros, éstos por campos, que se componen, a su vez, de datos de tipo simple. Por un abuso de lenguaje, se tiende a hacer sinónimos, el dato estructurado con su estructura correspondiente. Aunque ello evidentemente no es así, a un primer nivel, en este libro, asumiremos esta identificación.

Para muchos propósitos es conveniente tratar una estructura de datos como si fuera un objeto individual y afortunadamente, muchos lenguajes de programación permiten manipular estructuras completas como si se trataran de datos individuales, de forma que los datos estructurados y simples se consideran a menudo por el programador de la misma manera. Así a partir de ahora un dato puede ser tanto un entero como una matriz, por nombrar dos ejemplos.

Las estructuras de datos son necesarias tanto en la memoria principal como en la secundaria, de forma que en este capítulo nos centraremos en las correspondientes a la memoria principal, dejando para el capítulo siguiente las estructuras más adecuadas para el almacenamiento masivo de datos.

5.2. TIPOS DE DATOS ESTRUCTURADOS

Los datos de tipo simple tienen una representación conocida en términos de espacio de memoria. Sin embargo, cuando nos referimos a datos estructurados esta correspondencia puede no ser tan directa; por ello vamos a hacer una primera clasificación de los datos estructurados en: contiguos y enlazados. Las **estructuras contiguas** o físicas son aquellas que al representarse en el hardware del ordenador, lo hacen situando sus datos en áreas adyacentes de memoria; un dato en una estructura contigua se localiza directamente calculando su posición relativa al principio del área de memoria que contiene la estructura. Los datos se relacionan por su vecindad o por su posición relativa dentro de la estructura. Las **estructuras enlazadas** son estructuras cuyos datos no tienen por qué situarse de forma contigua en la memoria; en las estructuras enlazadas los datos se relacionan unos con otros mediante punteros (un tipo de dato que sirve para ‘apuntar’ hacia otro dato y por tanto para determinar cuál es el siguiente datos de la estructura). La localización de un dato no es inmediata sino que se produce a través de los punteros que relacionan unos datos con otros.

Los datos estructurados se pueden clasificar, también, según la variabilidad de su tamaño durante la ejecución del programa en: estáticos y dinámicos. Las **estructuras estáticas** son aquellas en las que el tamaño ocupado en memoria, se define con anterioridad a la ejecución del programa que los usa, de forma que su dimensión no puede modificarse durante la misma (p.e., una matriz) aunque no necesariamente se tenga que utilizar toda la memoria reservada al inicio (en todos los lenguajes de programación las estructuras estáticas se representan en memoria de forma contigua). Por el contrario, ciertas estructuras de datos pueden crecer o decrecer en tamaño, durante la ejecución, dependiendo de las necesidades de la aplicación, sin que el programador pueda o deba determinarlo previamente: son las llamadas **estructuras dinámicas**. Las estructuras dinámicas no tienen teóricamente limitaciones en su tamaño, salvo la única restricción de la memoria disponible en el computador.

Estas dos clasificaciones nos ayudarán a exponer los distintos tipos de datos estructurados, incidiendo en las ventajas e inconvenientes para su almacenamiento y tratamiento, en términos de la eficacia de una determinada aplicación ya sea de economía espacial (no emplear más memoria de la necesaria) o temporal (emplear el menor tiempo posible en las operaciones).

5.3. ESTRUCTURAS DE DATOS CONTIGUAS

Vamos a estudiar una serie de agrupaciones de datos que son utilizadas en todos los lenguajes de programación, y que tienen en común la ubicación de sus datos en zonas de memoria adyacentes.

5.3.1 Cadenas

La cadena es quizás la estructura más simple y se define como una secuencia de caracteres que se interpretan como un dato único. Su longitud puede ser fija o variable por lo que, además de saber que están constituidas por caracteres alfanuméricos, hemos de conocer su longitud. En una variable tipo cadena se puede almacenar una palabra, una frase, una matricula de coche, una temperatura, etc. La longitud de una cadena se puede determinar bien indicando al principio de la misma el número de caracteres que contiene, bien situando un carácter especial denominado **fin-de-cadena**. Los siguientes ejemplos muestran los dos métodos de representar la cadena “Capital 94” :

en el segundo caso el carácter elegido como fin-de-cadena ha sido el #. La cadena que no contiene ningún carácter se denomina cadena vacía y su longitud es 0, que no tiene que ser confundida por una cadena formada sólo por blancos (o espacios), cuya longitud es igual al número de blancos que contiene. De esta manera, una variable de tipo cadena de tamaño 10 puede guardar cadenas de 10 caracteres, pero también de menos si indicamos dónde terminan los caracteres de la cadena. Por ejemplo la cadena "Jaca 99":

Sobre datos de tipo cadena se pueden realizar las siguientes operaciones:

Asignación: Guardar una cadena en una variable tipo cadena. Como en toda asignación a una variable, la cadena que se guarda puede ser una constante, una variable tipo cadena o una expresión que produzca un dato tipo cadena. Por ejemplo:

nombre ← "Pepe"
nombre ← mi-nombre-de-pila

Concatenación: Formar una cadena a partir de dos ya existentes, yuxtaponiendo los caracteres de ambas. Si se denota por // al operador "concatenación", el resultado de:

"ab" // "cd" es "abcd"

Nótese que las constantes de tipo cadena se escriben entre comillas, para no confundirlos con nombres de variables u otros identificadores del programa.

Extracción de subcadena: Permite formar una cadena (subcadena) a partir de otra ya existente. La subcadena se forma tomando un tramo consecutivo de la cadena inicial. Si NOMBRE es una variable de tipo cadena que contiene "JUAN PEDRO ORTEGA" y denotamos por **(n:m)** la extracción de **m** caracteres tomados a partir del lugar **n**, entonces NOMBRE(6:5) es una subcadena que contiene "PEDRO".

Un caso particular de extracción que se utiliza a menudo es el de extraer un único caracter. Por ello se suele proporcionar un método directo: el nombre seguido por el lugar que ocupa dentro de la cadena. Así, en el ejemplo anterior, NOMBRE(6) = "P" = NOMBRE(6:1)

Obtención de longitud: La longitud de una cadena es un dato de tipo entero, cuyo valor es el número de caracteres que contiene ésta. En el primero de los dos métodos anteriores de representación de cadenas, la longitud se obtiene consultando el número de la primera casilla; en el segundo método la longitud es el número de orden que ocupa el caracter de fin-de-cadena, menos uno.

Comparación de cadenas: Consiste en comparar las cadenas carácter a carácter comenzando por el primero de la izquierda, igual que se consulta un diccionario. El orden de comparación viene dado por el código de E/S del ordenador (ASCII habitualmente). Así la expresión booleana: "Jose" < "Julio", se evaluara como verdadera. Nótese que en los códigos de E/S, las mayúsculas y las minúsculas son diferentes, dando lugar a resultados paradójicos en la comparación, así pues, si el código de E/S es ASCII, donde las mayúsculas tienen códigos inferiores a las minúsculas, se cumpliría que "Z" < "a".

5.3.2 Arrays[1]

Es un conjunto de datos del mismo tipo almacenados en la memoria del ordenador en posiciones adyacentes. Sus componentes individuales se llaman ***elementos*** y se distinguen entre ellos por el nombre del array seguido de uno o varios ***índices*** *o subíndices*. Estos elementos se pueden procesar, bien individualmente, determinando su posición dentro del array, bien como array completo. El número de elementos del array se específica cuando se crea éste, en la fase declarativa del programa, definiendo el **número de dimensiones** o número de índices del mismo y los límites máximo y mínimo que cada uno de ellos puede tomar, que llamaremos **rango**. Según sea este número, distinguiremos los siguientes tipos de arrays:

- unidimensionales (vectores)
- bidimensionales (matrices)
- multidimensionales

Por ello hablaremos de arrays de **dimensión 1, 2 ó n**, cuyo producto por el rango (o rangos) especifica el número de elementos que lo constituyen. Este dato lo utiliza el compilador para reservar el espacio necesario para almacenar en memoria todos

[1]Utilizaremos el término array, ya que su traducción castellana, "arreglo, colección, etc" es poco significativa y la imensa mayoría de veces se usa en el argot informático el anglicismo array. Sin embargo, sí usaremos vector y matriz para referirnos a determinados tipos de array.

sus elementos ocupando un área contigua. Cada elemento ocupa el mismo número de palabras, que será el que corresponda al tipo de éstos. No debemos olvidar que, al nivel físico, la memoria es lineal; por ello los elementos se colocan en la memoria linealmente según un orden prefijado de los índices.

En el ejemplo de la figura el número de dimensiones es 2, su rango (3;5) y el número total de elementos es 15.

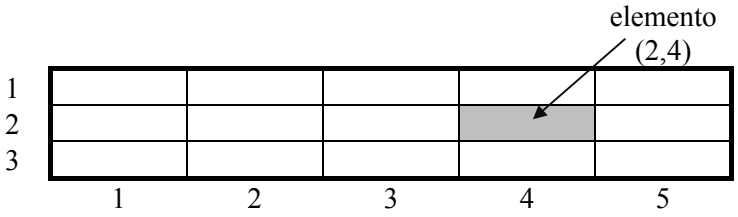


Fig. 5.1. *Representación de un array bidimensional*.

5.3.2.1 Vectores

Por motivos de simplicidad y mayor frecuencia de uso, a la hora de revisar las operaciones con arrays nos centraremos en los vectores, que además presentan la ventaja de ser estructuras ordenadas (sólo existe orden total cuando tenemos estructuras de una dimensión o lineales). Las notaciones algorítmicas que utilizaremos son:

nombre_vector = **vector** *[inf . . sup]* **de** *tipo*

nombre_vector	nombre válido del *vector*
inf .. sup	límites inferior y superior del rango (valor entero que puede tomar el índice)
tipo	tipo de los elementos del vector {entero, real, carácter}

La declaración de un vector supone una mera reserva de espacio, pudiéndose asumir que, hasta que asignemos valores por cualquier mecanismo a sus distintos elementos, estamos ante una estructura vacía.

NUMEROS = vector [1..10] **de** *real*

significa que NUMEROS es un vector, que podrá contener como elementos, al menos 10 números de tipo real, cuyos índices varían desde el 1 hasta el 10.

FORTRAN	BASIC

REAL X(10)	DIM X(10) AS SINGLE
PASCAL x: array[1..10] of real	**C** `float x[10]`

Es importante señalar que podemos implementar arrays cuyos elementos sean a su vez cadenas o elementos de otro tipo, así podemos pensar en situaciones como la siguiente, durante la fase declarativa

```
tipo: palabra = cadena[16]
COCHES = vector [1..9] de palabra
```

lo que nos permitirá manipular en un solo vector hasta 9 cadenas conteniendo como máximo 16 caracteres cada una. Con lo cual COCHES puede contener una información tal como:

1	Alfa Romeo
2	Fiat
3	Ford
4	Lancia
5	Renault
6	Seat
7	
8	
9	

Ya hemos dicho que las operaciones sobre arrays se pueden realizar con elementos individuales o sobre la estructura completa mediante las correspondientes instrucciones y estructuras de control del lenguaje. Las operaciones sobre elementos del vector son:

Asignación: Tiene el mismo significado que la asignación de un valor a una variable no dimensionada, ya que un vector con su índice representa la misma entidad que una variable no dimensionada.

A[20] ← 5 asigna el valor 5 al elemento 20 del vector A
A[17] ← B asigna el valor de la variable B al elemento 17 del vector A

Acceso secuencial o recorrido del vector: Consiste en acceder a los elementos de un vector para someterlos a un determinado proceso, tal como introducir datos (escribir) en él, visualizar su contenido (leer), etc.. A la operación de efectuar una acción general sobre todos los elementos de un vector se la denomina **recorrido** y para ello utilizaremos estructuras repetitivas, cuyas variables de control se utilizan

como subíndices del vector, de forma que el incremento del contador del bucle producirá el tratamiento sucesivo de los elementos del vector.

Ejemplo 1:

Escribir un algoritmo para recorrer secuencialmente un vector H de 10 elementos (haciendo la lectura y escritura de cada elemento) primero con un bucle desde y luego con un bucle mientras.

El pseudocódigo correspondiente será el siguiente:

```
desde i ← 1 hasta 10 hacer
        leer (H[i])
        escribir (H[i])
fin_desde
```

```
i ← 1
mientras i <= 10 hacer
        leer (H[i])
        escribir (H[i])
        i ← i+1
fin-mientras
```

Ejemplo 2:

Supongamos que queremos procesar los primeros elementos de un vector PUNTOS, previamente declarado, realizando las siguientes operaciones desde 1 hasta LIMITE (donde este valor debe necesariamente ser menor que el límite superior del rango): a) lectura del array; b) cálculo de la suma de los valores del array; c) cálculo de la media de los valores. Escribir el algoritmo correspondiente.

```
inicio
   escribir 'número de datos'
   leer numero
   suma← 0
   escribir 'datos del array'
   desde i=1 hasta numero hacer
      leer PUNTOS[i]
      suma← suma + PUNTOS[i]
   fin_desde
   media ← suma/numero
   escribir 'la media es', media
fin
```

BÚSQUEDA EN UN VECTOR: Esta operación, relacionada con la recuperación de información, consiste en encontrar un determinado valor dentro del vector, obteniendo su posición en el mismo en caso que éste exista o declarar la búsqueda

como fallida en caso de no encontrarlo. La experiencia a la hora de buscar un dato entre una colección de ellos nos dice que este proceso varía sensiblemente según que éstos estén o no ordenados; por esta razón presentaremos dos métodos básicos, según que el vector esté desordenado u ordenado.

Búsqueda secuencial: Consiste en recorrer el vector, con sus datos no necesariamente ordenados, del principio hacia el final. Si se encuentra el valor buscado se da por finalizada la búsqueda; en caso contrario, tras haber recorrido todo el vector se indica que el elemento en cuestión no se encuentra almacenado en el vector. Supongamos que deseamos localizar un determinado valor, que, obviamente, supondremos es del mismo tipo que los elementos del vector de rango n. El algoritmo es el siguiente:

```
algoritmo búsqueda_secuencial
A: vector donde se busca (contiene n elementos)
t: valor buscado
inicio
    encontrado ← Falso
    i ← 1
    mientras (i <= n) y (encontrado = Falso)  hacer
        si t = A[i]
            entonces
                escribir 'Se encontró el elemento buscado en la posición', i
                encontrado ← Verdadero
            si_no
                i ← i + 1
        fin_si
    fin_mientras
    si i = n+1   {también puede utilizarse si encontrado = Falso}
        entonces
            escribir 'No se encuentra el elemento'
        si_no
            escribir 'El elemento se encuentra en la posición', i
    fin_si
fin
```

Una manera más eficaz de realizar una búsqueda secuencial consiste en modificar el algoritmo utilizando un elemento centinela. Este elemento se agrega al vector al final del mismo. El valor del elemento centinela es el del argumento. El propósito de este elemento centinela, A[n+1], es significar que la búsqueda siempre tendrá éxito. El elemento A[n+1] sirve como centinela y se le asigna el valor de *t* antes de iniciar la búsqueda. En cada paso se evita la comparación de *i* con n y por

consiguiente este algoritmo será preferible al método anterior. Si el índice alcanzase el valor n+1, supondrá que el argumento no pertenece al vector original y en consecuencia la búsqueda no tiene éxito.

```
algoritmo búsqueda_centinela
A: vector donde se busca (contiene n elementos); t: valor buscado
inicio
   encontrado ← Falso
   i ← 1
   A[n+1] ← t
   mientras A(i) <> t hacer
      i ← i + 1
   fin_mientras
   si i = n+1
      entonces
         escribir 'No se encuentra el elemento'
   si_no
      entonces
         escribir 'El elemento se encuentra en la posición', i
   fin_si
fin
```

Notemos que, a pesar de esta mejora, el número de comparaciones que hemos de efectuar es del orden de la magnitud del vector y si ésta es muy grande el tiempo necesario para la búsqueda puede ser alto. Por ello, puede valer la pena tener ordenado el vector pues, como veremos ahora, las búsquedas sobre vectores ordenados son sensiblemente más rápidas.

Búsqueda binaria: Se aplica a vectores cuyos datos han sido previamente ordenados y es un ejemplo del uso del 'divide y vencerás' para localizar el valor deseado. Más adelante, trataremos los algoritmos (y su coste) para ordenar un vector; sin embargo ahora supondremos que esta ordenación ya ha tenido lugar. El algoritmo de búsqueda binaria se basa en los siguientes pasos:

1) Examinar el elemento central del vector; si éste es el elemento buscado, entonces la búsqueda ha terminado

2) En caso contrario, se determina si el elemento buscado está en la primera o en la segunda mitad del vector (de aquí el nombre de binario) y a continuación se repite este proceso, utilizando el elemento central del subvector correspondiente.

Ejemplo 3:

Considerar el siguiente vector ordenado de nueve elementos enteros, donde queremos buscar el valor 2983

1	2	3	4	5	6	7	8	9
2473	2545	2834	2892	2898	2983	3005	3446	3685
				central				

Para buscar el elemento 2983, se examina el número central 2898, situado en la quinta posición, que resulta ser distinto. Al ser 2983 mayor que 2898, se desprecia la primera mitad del vector, quedándonos con la segunda:

6	7	8	9
2983	3005	3446	3685
	central		

Se examina ahora el número central 3005, situado en la posición 7, que resulta ser distinto. Al ser 2983 menor que 3005, nos quedamos con la primera mitad del vector:

6
2983
central

Finalmente encontramos que el valor buscado coincide con el central. Nótese que si el valor buscado hubiera sido, por ejemplo, el 2900, la búsqueda habría finalizado con fracaso al no quedar mitades donde buscar.

Vamos a dar el algoritmo de búsqueda binaria para encontrar un elemento K, en un vector de elementos X(1), X(2),..., X(n), previamente clasificados en orden ascendente (ordenado en orden creciente si los datos son numéricos, o alfabéticamente si son caracteres). El proceso de búsqueda debe terminar normalmente conociendo si la búsqueda *ha tenido éxito* (se ha encontrado el elemento) o bien *no ha tenido éxito* (no se ha encontrado el elemento), y normalmente se deberá devolver la posición del elemento buscado dentro del vector. Las variables enteras BAJO, CENTRAL, ALTO, indican los límites inferior, central y superior del intervalo de búsqueda, en cada subvector que sucesivamente se esta considerando, durante la búsqueda binaria.

```
algoritmo búsqueda_binaria
X: vector de N elementos
K: valor buscado
inicio
   BAJO ← 1
```

```
    ALTO ← N
    CENTRAL ← ent((BAJO + ALTO) / 2)
    mientras BAJO < ALTO y X[CENTRAL] <> K hacer
        si K < X[CENTRAL]
            entonces
                ALTO ← CENTRAL - 1
            si_no
                BAJO ← CENTRAL + 1
        fin_si
        CENTRAL ← ent ((BAJO + ALTO) / 2)
    fin_mientras
    si K = X(CENTRAL)
        entonces escribir 'valor encontrado en', CENTRAL
        si_no escribir 'valor no encontrado'
    fin_si
fin
```

La función ent, entenderemos que obtiene un entero redondeándolo por defecto.

El funcionamiento del algoritmo de búsqueda binaria, en un vector de enteros, se ilustra en la Figura 5.2 para dos búsquedas: los enteros 8 y 11 que finalizan respectivamente *con éxito* (localizado el elemento) y *sin éxito* (no encontrado el elemento)

Aunque dejamos para un capítulo posterior el análisis de la complejidad algorítmica, con el único objeto de ilustrar la diferencia entre los dos tipos de búsqueda, digamos que para un vector de 1000 elementos, la búsqueda secuencial supone efectuar un promedio de unas 500 comparaciones, mientras que la binaria, resuelve el problema con sólo 10 en el peor de los casos. Otra cuestión, es el esfuerzo que nos suponga ordenar el vector, pero esto lo veremos más adelante.

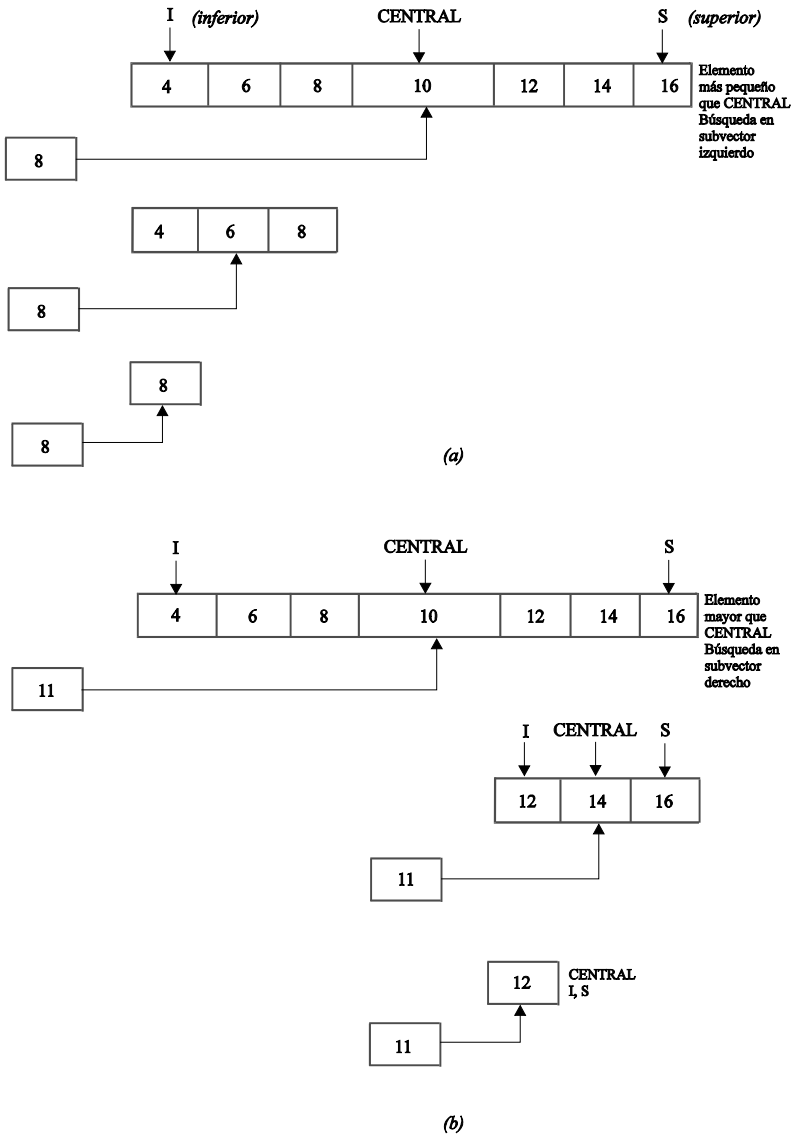


Fig. 5.2. *Ejemplo de búsqueda binaria: (a), con éxito; (b), sin éxito*.

Insertar datos en un vector

La operación *insertar* consiste en colocar un nuevo elemento, en una determinada posición del vector; ello supone no perder la información, que pudiera hallarse anteriormente, en la posición que va a ocupar, el valor a insertar. Es condición necesaria para que esta operación pueda tener lugar, la comprobación

de espacio de memoria suficiente en el vector, para el nuevo valor; dicho de otro modo, que el vector no tenga todos sus elemento ocupados por datos. Cada vez que insertamos un dato en una determinada posición hemos de correr todos los elementos posteriores del vector, hacia abajo, poniendo especial cuidado en que no se pierda ninguno de estos datos.

Ejemplo 4:

Consideremos el ya visto vector COCHES de 9 elementos que contiene 7 marcas de automóviles, en orden alfabético, en el que se desea insertar dos nuevas marcas OPEL y CITRÖEN manteniendo el orden alfabético del vector. Como *Opel* está comprendido entre *Lancia* y *Renault*, se deberán desplazar hacia abajo los elementos 5 y 6, que pasarán a ocupar la posición relativa 6 y 7. Posteriormente debe realizarse la misma operación con *Citröen* que ocupará la posición 2. El algoritmo que realiza esta operación para un vector de n elementos es el siguiente, suponiendo que hay espacio suficiente en el vector, y que conocemos la posición, que designaremos por P, que debe ocupar el elemento a insertar. Por ejemplo, la primera inserción, supone que *Opel* debe ocupar la posición P = 5.

```
1.    i ← n  {Indicar el número de posiciones del vector que ya están ocupadas}
2.    mientras i> = P hacer
            {mover el elemento actual i-ésimo hacia abajo, a la posición i+1}
            COCHES[i+1] ← COCHES[i]
            {decrementar contador}
             i← i-1
      fin_mientras
3.    {insertar el elemento en la posición P}
            COCHES[P] ←  "Opel"
4.    {actualizar el contador de elementos del vector}
5.    n←n + 1 {el vector acaba con un elemento más ocupado por datos}
6.    fin
```

Los contenidos sucesivos del vector COCHES son los siguientes:

1	Alfa Romeo
2	Fiat
3	Ford
4	Lancia
5	Renault

1	Alfa Romeo
2	Fiat
3	Ford
4	Lancia
5	Opel

1	Alfa Romeo
2	Citröen
3	Fiat
4	Ford
5	Lancia

6	Seat		6	Renault		6	Opel	
7			7	Seat		7	Renault	
8			8			8	Seat	
9			9			9		

Eliminar datos de un vector

La operación de *borrar* es distinta, según el elemento a eliminar se encuentre al final del vector (no presenta ningún problema) o se borre un elemento del interior de mismo vector. En este último la eliminación provocara el movimiento hacia arriba de los elementos posteriores a él para reorganizar el vector. El algoritmo de borrado del dato almacenado en el elemento j-ésimo del vector COCHES es el siguiente:

```
algoritmo borrado {borrar el elemento j-ésimo}
inicio
   desde i = j hasta N - 1
      {llevar elemento i + 1 hacia arriba}
      COCHES[i] ← COCHES[i+1]
   fin-desde
   {actualizar contador de elementos}
   {ahora COCHES tendrá ocupado por datos un elementos menos, n - 1}
   N ← N - 1
fin
```

5.3.2.2 Matrices

Llamaremos **matriz o tabla** a un array bidimensional, esto es, un conjunto de elementos del mismo tipo en el que sus componentes vienen definidos por dos subíndices, el primero referido a la fila y el segundo a la columna. Además de las matrices, que utilizamos en álgebra lineal, un ejemplo típico de esta estructura es el tablero de ajedrez, que se representa por un array T[8,8]. Se puede representar la posición o casilla (recuadro) de cada tablero del siguiente modo:

T[i,j] = 0 si no existe ninguna pieza en la fila i-esima y columna j-esima
T[i,j] = 1 si la casilla contiene un peón blanco
= 2 para un caballo blanco
= 3 para alfil blanco

= 4 para una torre blanca
= 5 para una reina blanca
= 6 para un rey blanco

con los correspondientes números negativos para las piezas negras.

Puesto que la memoria del ordenador no está organizada en forma rectangular sino en forma de una enorme fila para almacenar una matriz hemos de recurrir a guardar las filas una a continuación de otra. Así, si en una matriz con un número de columnas C, que es el número de elementos que tiene cada fila, para localizar el elemento de fila i y columna j, tendríamos que movernos, desde la posición de inicio de la matriz, hasta la posición: C*(i-1) + j. Afortunadamente el propio software del sistema se encarga de convertir los términos en forma de filas y columnas, en localizaciones concretas dentro de la memoria, de forma que podemos pensar conceptualmente en forma de tabla, aunque se almacenen en forma de fila, dentro de la máquina[2].

5.3.2.3 Arrays multidimensionales

Dependiendo del tipo de lenguaje, pueden existir arrays de tres o más dimensiones (por ejemplo FORTRAN 77 admite hasta siete dimensiones). Para el caso de tres dimensiones, la estructura puede visualizarse como un cubo, y para mayor número de dimensiones ésta visualización no es posible.

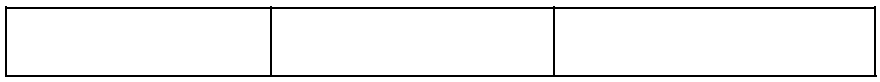
El tratamiento de estos arrays es similar al de las matrices, cada conjunto de índices individualiza un elemento de la estructura, que se almacena en memoria de forma secuencial.

5.3.3 Registros

Hasta ahora nos hemos referido a estructuras formadas por datos simples del mismo tipo; sin embargo, es interesante poder manejar una especie de arrays heterogéneos en los que sus elementos puedan ser de tipos diferentes. Llamaremos **registro** a una estructura de datos, formada por yuxtaposición de elementos que contienen información relativa a un mismo ente. A los elementos que componen el registro los llamamos campos, cada uno de los cuales es de un determinado tipo, simple o estructurado. Los campos dentro del registro aparecen en un orden determinado y se identifican por un nombre. Para definir el registro es necesario

[2]La forma de almacenar ls matrices expuesta se conoce como “orden principal de la fila”, pero no es la única. Así el FORTRAN utiliza el “orden principal de la columna” consistente en almacenar los elementos de una columna consecutivamente y pasar a la columna siguiente.

especificar el nombre y tipo de cada campo. Por ejemplo consideremos un registro, referido a Empleado, que está constituido por tres campos: Nombre (cadena), Edad (entero) y Porcentaje de impuestos (real).



Nombre Edad Porcentaje de impuestos

Las operaciones básicas que se ejecutan con los registros son: asignación del registro completo a una variable de tipo registro, (definida con sus mismos campos) y selección de un campo, que se realiza especificando el nombre del campo. Puesto que esta estructura, esta especialmente ligada a las transferencias con los periféricos de almacenamiento, volveremos sobre ella cuando nos refiramos a los archivos.

5.4. ESTRUCTURAS DINÁMICAS Y PUNTEROS

Hasta este momento hemos venido trabajando con variables, dimensionadas o no, que son direcciones simbólicas de posiciones de memoria, de forma que existe una relación bien determinada entre nombres de variables y posiciones de memoria durante toda la ejecución del programa. Aunque el contenido de una posición de memoria asociada con una variable puede cambiar durante la ejecución, es decir el valor asignado a la variable puede variar, las variables por si mismas no pueden crecer ni disminuir, durante la ejecución. Sin embargo, en muchas ocasiones es conveniente poder disponer de un método por el cual, podamos adquirir posiciones de memoria adicionales, a medida que las vayamos necesitando durante la ejecución y al contrario liberarlas cuando no se necesiten.

Las variables y estructuras de datos que reúnen estas condiciones se llaman **dinámicas**[3] y se representan con la ayuda de un nuevo tipo de dato, llamado **puntero**, que se define como un dato que indica la posición de memoria ocupada por otro dato. Puede concebirse como una flecha, que "apunta", al dato en cuestión (Ver Figura 5.3).

[3]El concepto de estructura dinámica se refiere a la utilización de punteros que permiten que la estructura tenga las propiedades expuestas. Sin embargo, como veremos más adelante, éste es un concepto ligado a la implementación que se haga de la estructura y no de la estructura en sí. Así hay estructuras que pueden implementarse estática o dinámicamente (p.e. colas o pilas).

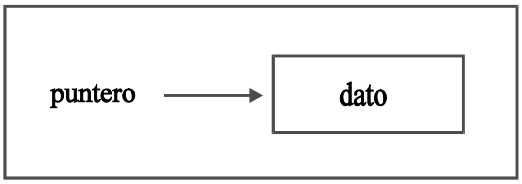


Fig. 5.3 *Representación de un puntero*

Los punteros proporcionan los enlaces de unión entre los elementos, permitiendo que, durante la ejecución del programa, las estructuras dinámicas puedan cambiar sus tamaños. En las estructuras dinámicas estos elementos, llamados **nodos,** son normalmente registros de al menos dos campos, donde por lo menos uno de ellos, es un puntero (es decir contiene información que permite localizar al siguiente - siguientes- nodo de la estructura). Ver Figura 5.4.

Fig. 5.4. *Representación de un nodo*

La utilización de punteros permite que sea relativamente fácil añadir indeterminadamente nuevos datos, insertar nuevos nodos en otros ya existentes y en general modificar estas estructuras. Dependiendo de las relaciones entre los nodos de la estructura hablaremos de estructuras lineales y no lineales: si partiendo del nodo inicial es posible dirigirse sucesivamente a todos los nodos visitando cada uno una única vez diremos que es una **estructura lineal**; en caso de no ser posible el recorrido en estas condiciones se habla de **estructura no lineal**.

5.5. ESTRUCTURAS LINEALES

Una **lista** en su sentido amplio, es un conjunto de datos del mismo tipo (simple o estructurado), cada elemento de la cual tiene un único predecesor (excepto el primero) y un único sucesor (excepto el último) y cuyo numero de elementos es variable. Se distinguen dos tipos de listas: contiguas y lineales. La lista **contigua** es una estructura intermedia entre las estáticas y las dinámicas, ya que sus datos se almacenan en la memoria del ordenador en posiciones sucesivas y se procesan como vectores. Con esta disposición secuencial, el acceso a cualquier elemento de la lista y la adición por los extremos de nuevos elementos es fácil, siempre que haya espacio para ello. Para que una lista contigua pueda variar de tamaño y por tanto, dar la impresión de que se comporta como una estructura dinámica, hay que definir un vector dimensionado con tamaño suficiente para que pueda contener todos los posibles elementos de la lista. Como se observará, una lista contigua es un vector que tiene posiciones libres por delante y detrás y el

índice del mismo hace de puntero. Es un caso relativamente banal de estructura dinámica, al cual no prestaremos mayor atención.

5.5.1 LISTAS ENLAZADAS

Estas listas están formadas por un conjunto de nodos, en los que cada elemento contiene un puntero con la posición o dirección del siguiente elemento de la lista, es decir su enlace. Se dice entonces, que los elementos de una lista están **enlazados** por medio de los campos enlaces. Cada nodo está constituido por dos partes: **información** o **campos** de datos (uno o varios) y un **puntero** (con la dirección del nodo siguiente). Al campo o campos de datos del nodo lo designaremos como **INFORMACIÓN** del nodo y al puntero por **SIGUIENTE** del nodo. Con esta organización de los datos, es evidente que no es necesario que los elementos de la lista estén almacenados en posiciones físicas adyacentes para estar relacionados entre sí, ya que el puntero indica unívocamente la posición del dato siguiente en la lista.

Una forma alternativa de ver las listas enlazadas, es considerarlas como una estructura que contienen un dato dado como la **cabeza**, y el resto como la **cola**. La notación usual es la siguiente:**(A | B)**, donde la lista tiene el elemento **A** en la cabeza y el elemento **B** como cola. Recursivamente, **(A| (B |(C D)))** es la lista con el elemento A en la cabeza, mientras que la cola está forma por una lista con el elemento B en la cabeza y una lista que contiene los elementos C y D en la cola, etc.

Nótese que para tener definida una lista enlazada, además de la estructura de cada uno de sus nodos, necesitamos una variable externa a la propia lista, con un puntero que marque la posición de la cabeza (inicio, primero) de la lista. Esta variable es quien normalmente da nombre a la lista, pues nos dice donde localizarla, sea cual sea su tamaño. Para detectar el último elemento de la lista se emplea un puntero nulo, que por convenio, se suele representar de diversas formas: por un enlace con la palabra reservada **nil (NULO)**, por una barra inclinada (/) (Ver Figura 5.5) o por un signo especial, tomado de la toma de tierra en electricidad. Una lista enlazada sin ningún elemento se llama lista vacía y las distinguiremos asignando a su puntero de cabecera el valor nil.

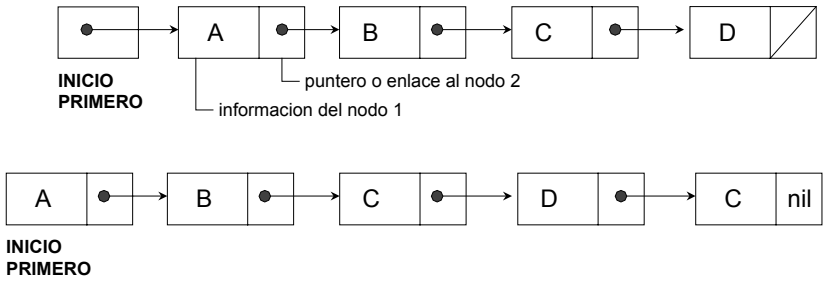
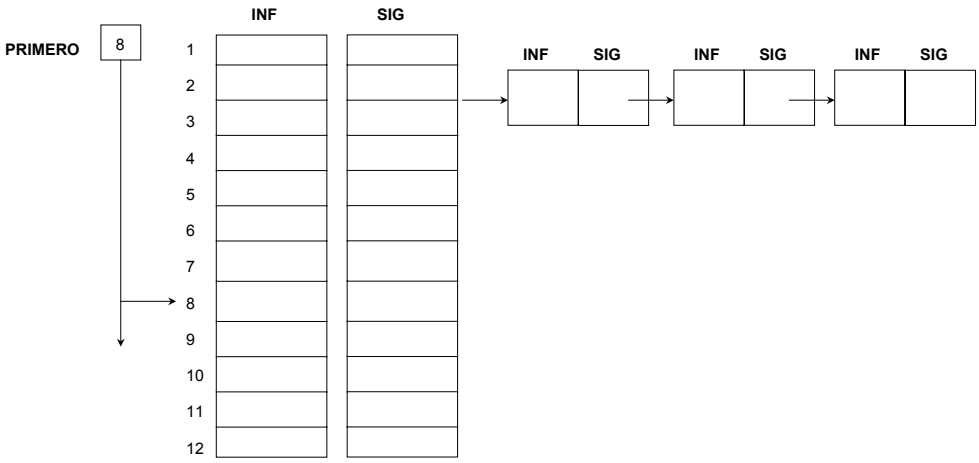


Fig. 5.5. *Ejemplos de listas enlazadas*

5.5.1.1 Creación de una lista

La implementación de una lista enlazada, depende del lenguaje de programación, ya que no todos soportan el puntero como tipo de datos; afortunadamente los más modernos como C, Pascal, etc., si lo hacen, por lo que en lo que sigue vamos a asumir el uso de punteros para su manejo. La alternativa, al uso de punteros, es recurrir a vectores paralelos en los que se almacenan los datos correspondientes a INFORMACION y SIGUIENTE, con una variable que apunte al índice que contiene la cabeza de la lista, de forma que nos podemos imaginar el siguiente esquema de equivalencia, entre ambas estructuras físicas:



Una vez definida la estructura de sus nodos, cosa que dejamos aparte, pues dependerá de cada lenguaje de programación, crear una lista consiste en llenar un primer nodo con la información correspondiente a un elemento y cuyo campo de enlace sea NULO. Además hay que definir la variable con el puntero externo que debe contener la dirección de este nodo inicial. Fijarse que ello supone que el lenguaje de programación que utilicemos debe tener una función que nos de la posición de memoria en la que se ha almacenado este nodo. A partir de este nodo

inicial la lista empieza a modificarse, creciendo y disminuyendo, insertando y borrando nodos.

5.5.1.2 Procesamiento de listas enlazadas

El objetivo de lo que sigue, no es tanto la descripción de las operaciones de procesamiento del ejemplo más significativo de estructura dinámica, como la comparación de éstas con las estáticas, en términos de utilidad para la programación. Se trata por tanto solamente de dar una idea, acerca de los pros y contras de utilizar vectores o listas enlazadas como estructuras lineales.

En esta línea, vamos ahora a ver como efectuamos, con listas enlazadas, las mismas cuatro operaciones (recorrido, búsqueda, inserción y eliminación) que ya hemos llevado a cabo utilizando vectores, como estructura de datos.
Vamos a utilizar las notaciones algorítmicas siguientes:

PRIMERO	es un puntero externo al primer nodo de una lista enlazada;
P	es un puntero a un nodo cualquiera de la lista;
NODO(P)	el nodo apuntado por P.
INFO(P)	campo de datos del nodo apuntado por P;
SIG(P)	campo puntero del nodo apuntado por P (apunta al nodo siguiente).

Recorrido de una lista

Recorrer un vector fue fácil con una estructura repetitiva, al poder utilizar su índice para poder ir del principio al final de la estructura. Sin embargo, en el caso de las listas, al carecer de índice, puede pensarse que no es tan fácil acceder directa o aleatoriamente a sus nodos, ya que para ello es necesario acceder al primer nodo mediante el puntero externo, al segundo, después de acceder al primero, etc. Afortunadamente, esto se resuelve, usando una variable puntero, auxiliar X, que apunte en cada momento al nodo procesado, de forma que. con solo hacer la asignación: X ← SIG(X), esto es guardar en X el campo puntero del nodo (Ver Figura 5.6) se tiene el efecto de avanzar hacia el siguiente nodo de la lista.

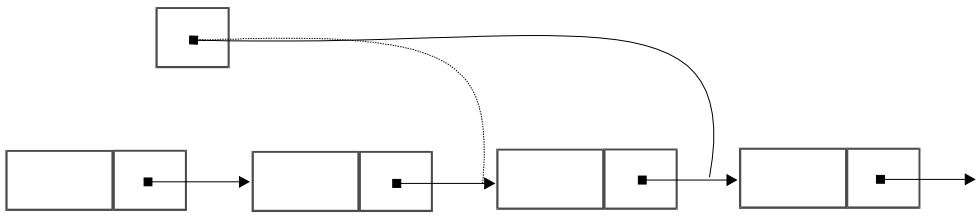


Fig. 5.6 *Asignación X ← SIG(X)*

Basándonos en lo anterior , el recorrido de una lista, es posible utilizando el puntero temporal P. Así el siguiente algoritmo, nos permite leer la lista completa.

```
1.  inicio
2.      P ← PRIMERO
3.      mientras P <> nil hacer
4.          escribir INFO(P)
5.          P ← SIG(P)
6.      fin_mientras
7.  fin
```

El puntero P contiene el valor del primer elemento de la lista. El bucle recorre toda la lista hasta que encuentra un nodo cuyo siguiente sea NULO, en cuyo caso se efectuaría: (P ← NULO), que corresponde al ultimo de la lista. La línea 4 escribe el valor de cada elemento y la línea 5 avanza el puntero P, dándole el valor del campo SIG del nodo actual.

Ejemplo 5:

Escribir un algoritmo que recorra una lista enlazada para calcular el número de elementos que la componen (que en principio es indeterminado):

```
inicio
   N ← 0                        {contador de elementos}
   P ← PRIMERO
   mientras P <> nil hacer
      N ← N + 1
      P ← SIG(P)
   fin_mientras
fin
```

Búsqueda en una lista

En el caso de trabajar con un vector, la búsqueda variaba sensiblemente según que éste estuviera o no ordenado. En el caso de una lista enlazada, la búsqueda debe hacerse mediante un recorrido de la misma, elemento a elemento, hasta o bien encontrar el elemento deseado o bien detectar el final de la lista, aunque en el caso de que la lista este ordenada podemos dar por terminada la búsqueda sin éxito, cuando en este recorrido, nos encontremos con un nodo, cuya información es posterior al valor buscado. Veamos estos dos casos, con listas desordenadas y ordenadas.

Dada una lista enlazada cualquiera cuyo primer nodo está apuntado por PRIMERO, el siguiente procedimiento busca un elemento *t* obteniendo un puntero POS que lo apunta (si no se encuentra el elemento POS debe ser NULO).

```
procedimiento BusquedaDesordenada(PRIMERO, t, REF POS)
{Elemento buscado: t}
{puntero al lugar que ocupa: POS}
inicio
  P ← PRIMERO
  POS ← NULO
  mientras P <> NULO hacer
     si t= INFO [P]
        entonces
        POS ← P
        P ← NULO
        sino
        P ← SIG [P]
     fin-si
  fin-mientras
  si POS = NULO entonces escribir  ´busqueda sin exito¨ {esto es opcional}
fin
```

En caso de que la lista está ordenada en orden ascendente, el algoritmo de búsqueda en las condiciones arriba indicadas es el siguiente.

```
procedimiento BusquedaOrdenada (PRIMERO, t,REF POS)
inicio              {se supone la lista ordenada}
  P ← PRIMERO
  POS ← NULO
  mientras P <> nulo hacer
     si INFO[P] < t
        entonces
        P ← SIG[P]
        si_no si t = INFO [P]
                 entonces
                 POS ← P
                 P ← NULO
              fin-si
     fin-si
  fin-mientras
  si  POS = NULO entonces escribir  ´búsqueda sin exito¨
fin
```

Como podemos observar, la búsqueda en listas sin ordenar resulta muy parecida a la que se da en el caso de vectores. Sin embargo para listas enlazadas ordenadas, al no poder conocer el tamaño que tiene la estructura en cada momento, no podemos explotar todas las posibilidades que nos proporcionaban los vectores ordenados con la búsqueda binaria.

Inserción de un elemento

Insertar, al igual que borrar, consiste básicamente en modificar los punteros de la lista. Vamos a hacer el siguiente planteamiento general: Sea una lista enlazada en la que se desea insertar un nuevo nodo, cosa que puede hacerse bien al principio de la lista, bien a continuación de un nodo específico. La inserción al principio de la lista es directa (Ver Figura 5.7)
En cualquier inserción necesitamos disponer de un nodo vacío, donde depositar la información que queremos añadir, previamente al propio proceso de inserción. Llamaremos DISPO a una lista de nodos disponibles, para guardar el nodo antes de ser insertado en la lista.

Veamos como se efectúa la inserción de un nodo, con la información ELEMENTO, al principio de la lista.

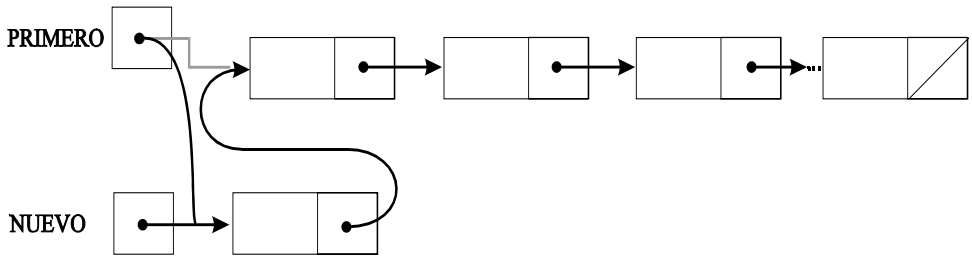


Fig. 5.7. *Inserción en el primer nodo*

algoritmo inserción
NUEVO ← DISPO {obtiene un nuevo nodo si es posible; si no, da NULO}
si NUEVO = NULO
{comprobación de desbordamiento}
  entonces
    escribir "desbordamiento"
  sino
    INFO(NUEVO) ← ELEMENTO
     {Colocamos en el campo INFORMACIÓN del nuevo nodo los datos}
    SIG(NUEVO) ← PRIMERO

```
        {El puntero del nuevo nodo apunta a la cabeza de la lista original}
      PRIMERO ← NUEVO
        {El nuevo nodo es ahora la nueva cabeza de la lista}
fin_si
fin
```

La inserción de un nuevo nodo (cuya información llamaremos NOMBRE) a continuación de un nodo dado apuntado por P, exige la utilización de un puntero auxiliar Q, aparte de los ya utilizados P y NUEVO. Veamos su algoritmo (Ver Figura 5.8) asumiendo que habrá disponibilidad de nodo (no hay desbordamiento) con la misma nomenclatura que en la Figura 5.7.:

1. NUEVO ← DISPO
2. INFO(NUEVO) ← NOMBRE
3. Q ← SIG(P)
4. SIG(P) ← NUEVO
5. SIG(NUEVO) ← Q

Después de los pasos 1 y 2, tendremos la estructura mostrada en la Figura 5.8 (a).

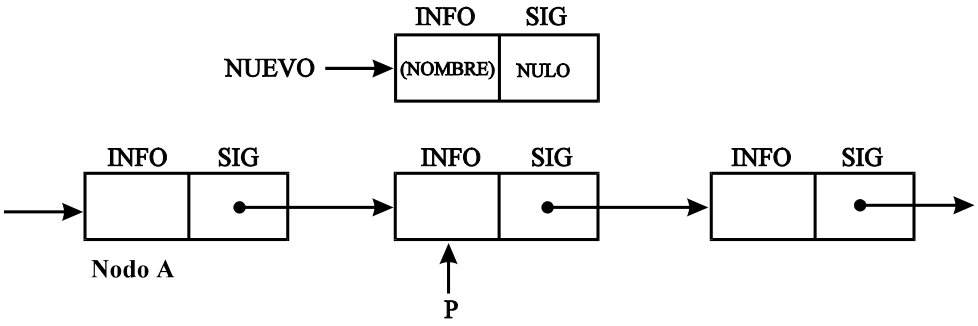


Fig. 5.8 (a)

Tras los pasos 3 y 4, se tiene la estructura mostrada en la Figura 5.8 (b)

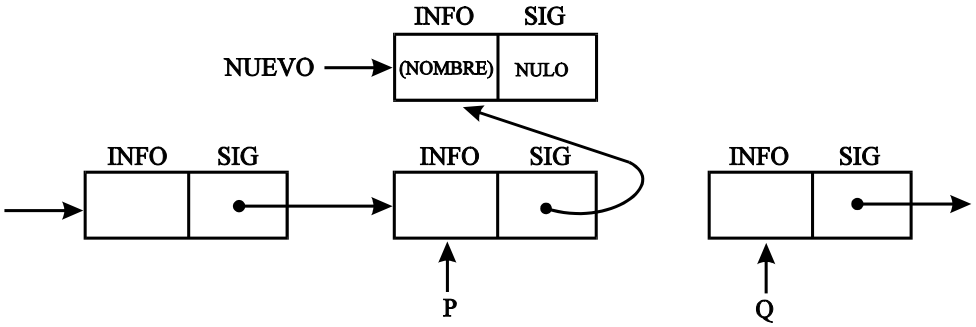


Fig. 5.8 (b)

Finalmente tras la ejecución de la última instrucción, la situación será la mostrada en la Figura 5.8 (c)

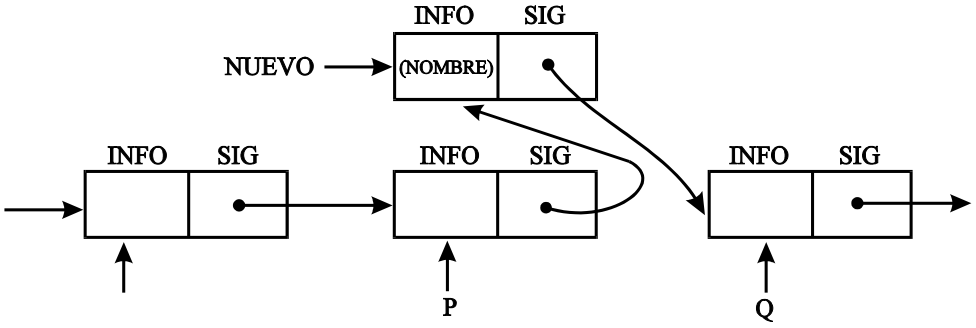


Fig. 5.8 (c)

Nótese que el puntero Q puede evitarse ya que los pasos 3, 4 y 5 son reemplazables por:

SIG (NUEVO) ← SIG (P)
SIG (P) ← NUEVO

Sin embargo, recurrir a Q nos será útil durante el posible proceso de búsqueda del nodo a partir del cual insertar.

Eliminación de un elemento de una lista enlazada

Esta operación consiste en hacer que el nodo anterior, al nodo que quiere eliminarse, se enlace con el posterior a éste, con lo cual el nodo que nos interesa quedará fuera de la lista. Consideremos la lista enlazada de la Figura 5.9(a). El algoritmo que sigue elimina de la lista enlazada el elemento siguiente al apuntado por P, utilizando un puntero auxiliar Q:

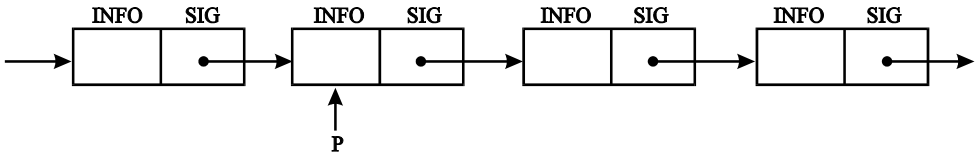


Fig. 5.9 (a) *Lista donde se desea eliminar el elemento apuntado por SIG(P)*

1. Q ← SIG(P)
2. SIG(P) ← SIG(Q)
3. LIBERARNODO(Q)

Tras cada uno de estos tres pasos, la lista sufre los siguientes pasos:

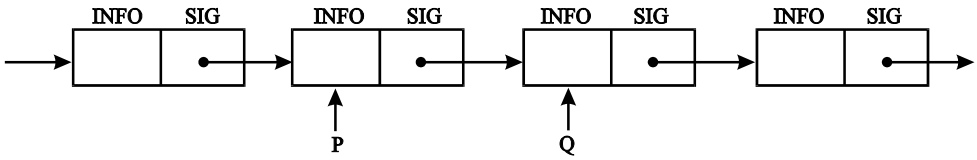


Fig. 5.9 (b) *Situación tras el paso 1*

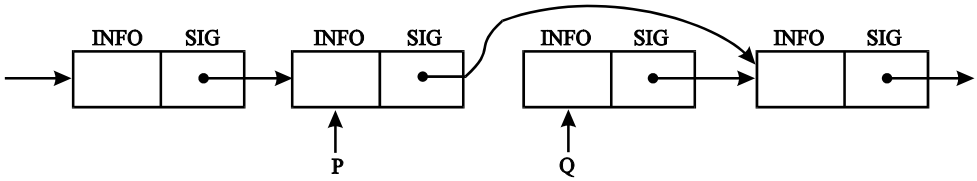


Fig. 5.9 (c) *Situación tras el paso 2*

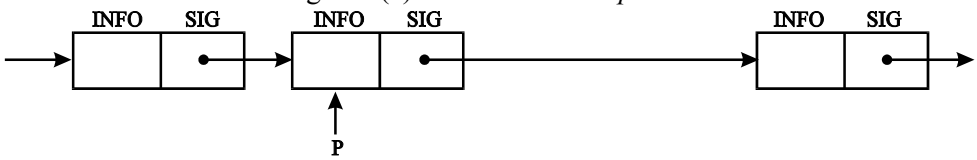


Fig. 5.9 (d) *Situación tras el paso 3*

Notemos que a su vez, el proceso de borrado dará lugar a la existencia de nodos libres o disponibles, al liberar el nodo que se elimina; este espacio de memoria podrá ser utilizado al invocar DISPO.

5.5.2 VECTORES vs LISTAS ENLAZADAS

El uso de estructuras lineales en programación, es constante, por lo que la decisión entre la utilización de vectores o listas enlazadas, en el caso de que el lenguaje de programación que utilicemos las soporte, es una constante disyuntiva. No podemos dar ninguna regla al respecto, ya que será cada aplicación la que

aconsejará una u otra opción. Solamente podemos hacer algunas consideraciones a la luz de lo que hemos aprendido hasta ahora, en este capítulo.

Comparando los procesos de inserción y borrado en listas enlazadas con los correspondientes a los que vimos en vectores, resulta bastante evidente que el número de operaciones necesarias es mucho menor en las listas. Por tanto, para aquellas aplicaciones en las que la inserción y el borrado juegan un papel significativo, hay que considerar seriamente la utilización de listas enlazadas, al objeto de mejorar la eficiencia y rapidez del procedimiento. Esta situación, como vimos, no se repite si consideramos el proceso de búsqueda, en cuyo caso los algoritmos para vectores ordenados son mucho más eficaces. Evidentemente, si necesitamos hacer frecuentemente estas tres operaciones, la solución acabará siendo un compromiso, entre las dos opciones. Más adelante propondremos una nueva estructura, no lineal, que será razonablemente aceptable, cuando las tres operaciones búsqueda, inserción y borrado sean igualmente frecuentes.

5.5.3 PILAS

Para introducir esta estructura, recordemos la forma en que se apilan los platos en los restaurantes: una pila de platos se soporta sobre un muelle, cuando se retira un plato, los demás suben. Vamos a trasladar esta idea a la informática. Una **pila (stack)** es una estructura lineal a cuyos datos sólo se puede acceder por un solo extremo, denominado **tope** o **cima (top)**. En esta estructura sólo se pueden efectuar dos operaciones: añadir y eliminar un elemento, acciones que se conocen por **meter (push),** y **sacar (pop)**. Si se meten varios elementos en la pila y después se sacan de ésta, el último elemento en entrar será el primero en salir. Por esta razón se dice que la pila es una estructura en la que el último en entrar es el primero en salir, en inglés, **LIFO (last in first out)**.

Cuando se almacena una pila en la memoria de un ordenador, los elementos realmente no se mueven arriba y abajo, a medida que se meten o sacan de la pila. Simplemente es la posición del tope de la pila la que varía. Un puntero, denominado, **puntero de pila**, indica la posición del tope o, lo que es lo mismo, el primer elemento disponible en la cima. Otro puntero se emplea para determinar la base de la pila que mantiene el mismo valor mientras existe la pila. La Figura 5.10 muestra el uso del puntero de la pila y la base de ésta. Si se realiza la secuencia de operaciones: sacar, sacar y meter 5.9, el estado resultante de la pila aparece en la Figura 5.11. Para representar una pila vacía, el puntero de la pila tiene el mismo valor que la base de la pila.

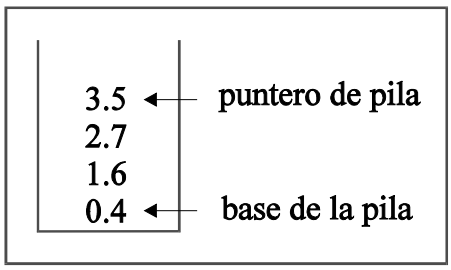


Fig. 5.10 *Pila*

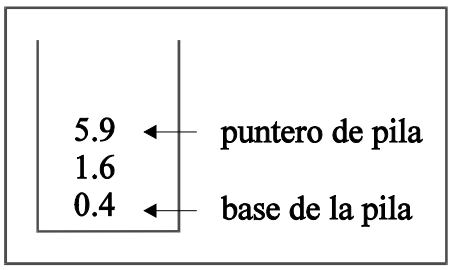


Fig. 5.11. *Estado de la pila de la Fig. 5.10 tras las operaciones: sacar, sacar, meter 5.9*

La pila es una de las estructuras más importantes en computación, se usa para calcular expresiones, para pasar de un lenguaje de ordenador a otro y para transferir el control de una parte de un programa a otra. Como ejemplo de uso de la pila, considérese un programa que llama a la función raíz cuadrada de un número. El subprograma recibe el argumento cuya raíz ha de calcularse y retorna la raíz ya calculada. Este argumento se saca de la pila para utilizarlo y una vez calculada la raíz, cuando la función termina, mete el resultado en la pila. El último dato entrado, argumento, es el primer dato sacado e igual ocurre con el resultado de la función que es el primer dato utilizado, cuando el control retorna al programa.

5.5.3.1 Procesamiento de una pila

Para trabajar con pilas, hay que contar con procedimientos para meter y sacar elementos y para comprobar si la pila está vacía (puede utilizarse una variable o función booleana VACIA, de modo que cuando su valor sea *verdadero* la pila está vacía, y *falso* en caso contrario).

Las pilas pueden implementarse utilizando memoria estática (por medio de vectores) o dinámica (utilizando punteros). Vamos a estudiar la implementación de una pila utilizando un vector S, de tamaño LONGMAX, dejando al lector la implementación dinámico de una pila como caso particular de lista enlazada.

Utilizaremos las siguientes notaciones:

p = CIMA	puntero de la pila
LONGMAX	longitud máxima de la pila
S(i)	elemento i-ésimo de la pila S
X	elemento a añadir/quitar de la pila
VACIA	función booleana “pila vacía”

PUSH	subprograma para añadir o meter elementos
POP	subprograma para eliminar o sacar elementos

Veamos cuál es el algoritmo de los procedimientos PUSH, POP y de la función booleana VACIA

Meter el dato x en la pila (push):

```
inicio
   si p = LONGMAX
      entonces
         escribir 'pila llena'
      sino
         p ← p + 1
         S(p) ← x
   fin_si
fin
```

Sacar un dato de la pila (pop) y ponerlo en x:

```
inicio
{este subprograma pone el valor de x en la cima de la pila}
si p = 0
  entonces
     escribir 'pila vacía'
  sino
     x ← S(p)
     p ← p - 1
  fin_si
fin
```

Función Pila vacía (*vacía*)

```
inicio
   si p = 0
      entonces
      VACIA ← cierto
      sino
      VACIA ← falso
   fin_si
fin
```

5.5.4 COLAS

A pesar de los orígenes no europeos de muchas de las ideas asociadas con los ordenadores, esa importante institución británica, la **cola**, ha encontrado su lugar en las ciencias de la computación. Todo el mundo sabe como funciona una cola, los recién llegados se sitúan al final, mientras que la desaparición se hace por el principio, sin que esté permitido "colarse". Vamos a definir cola, como una estructura lineal, en la que los datos entran por la parte de atrás y salen por la de delante. Una cola es una estructura en la que el primer dato en entrar es el primer dato en salir, es decir, una estructura **FIFO (first in, first out).**

Hay varias formas de implementar una cola en la memoria de un ordenador, bien con vectores, bien en listas enlazadas. En cualquier caso se necesitan dos variables que representan a los punteros FRENTE (f = front) y FINAL (r = rear). El estado de COLA VACIA se manifiesta cuando f y r son ambas nulos en la implementación dinámica o cuando coinciden en el caso estático.

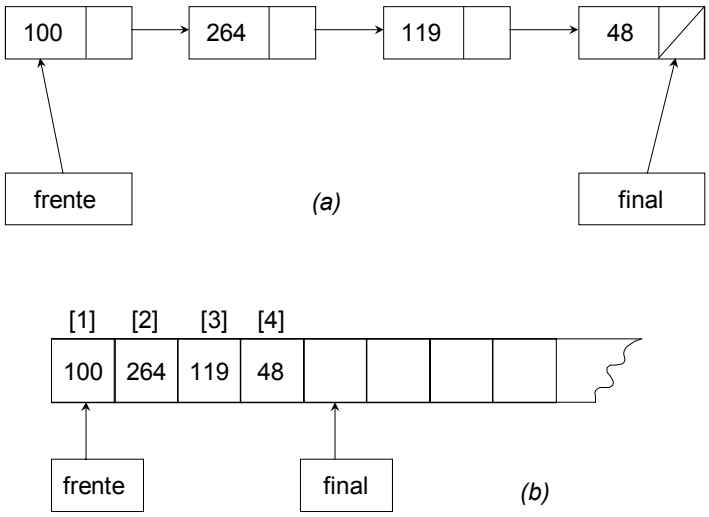


Fig.5.12 *Ejemplos de Implementación de una cola: a) con una lista enlazada. b) con un vector*

Las colas se utilizan algo menos que las pilas, por lo que no insistiremos en las operaciones que facilitan su procesamiento. Sin embargo digamos que facilitan la interconexión y el almacenamiento de datos en tránsito tanto en redes de ordenadores, como entre un procesador y un periférico (así por ejemplo, en los trabajos para imprimir, decimos que están en cola de impresión, por orden de llegada).

Como se puede observar, la cola puede considerarse en la implementación dinámica como un caso particular de lista enlazada. Se deja al lector la escritura de los algoritmos de inserción y borrado de un elemento y la comprobación de cola vacía, tanto en el caso estático como en el dinámico.

5.6. ESTRUCTURAS NO LINEALES (ÁRBOLES)

Existen en Informática varias estructuras no lineales, en las que un elemento puede estar relacionado con más de uno sea por delante o detrás de él (por ejemplo, los grafos); no obstante, dentro del carácter introductorio de este curso, nosotros nos restringiremos a la más sencilla de ellas, la de árbol.

A todos nos son familiares expresiones como "árbol genealógico" o "recorrer un árbol". En este sentido, un **árbol** es una estructura que implica una jerarquía, en la que cada elemento está unido a otros por debajo de él. Comparada con las estructuras lineales anteriores, el árbol tiene la particularidad de que cada elemento puede tener más de un "siguiente", aunque un solo antecedente o padre.

Definiremos **árbol**, de forma recursiva, como un conjunto finito de uno o más nodos**,** de tal manera, que exista un nodo especial denominado **raíz** y los nodos restantes están divididos en conjuntos denominados **subárboles**, que también responden a la estructura de un árbol. Por extensión a la idea de árbol genealógico se habla de nodos padres y nodos hijo y un nodo, en la parte inferior del que no cuelgue ningún subárbol (no tiene ningún hijo) se denomina **nodo terminal** u **hoja** (véase Figura 5.13).

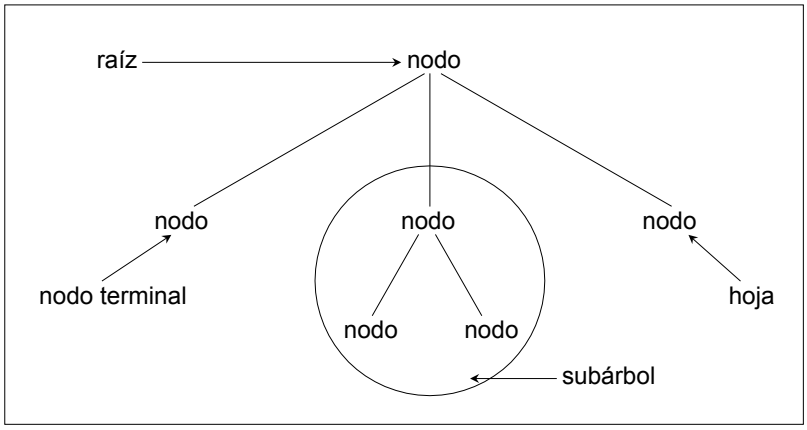


Fig. 5.13. *Conceptos relacionados con los árboles*

5.6.1 ÁRBOLES BINARIOS

Hay un tipo especial de árbol muy usado en computación, denominado **árbol binario**, en el que de cada nodo pueden colgar, a lo más, dos subárboles. Estos se denominan **subárbol derecho** y **subárbol izquierdo**, y también son árboles binarios. La Figura 5.14 representa un árbol binario.

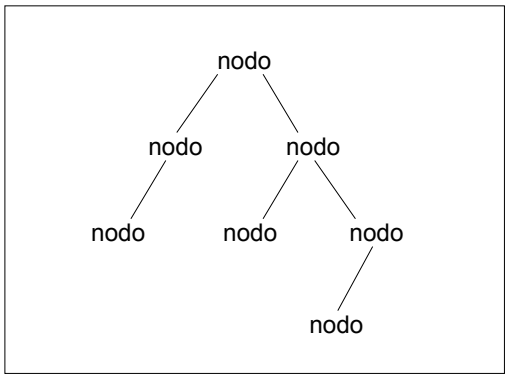


Fig. 5.14. *Un árbol binario*

La forma usual de representar los árboles supone el uso de punteros, aunque también se puede hacer con vectores. En un árbol binario cada nodo está constituido por una parte de datos (INFORMACION) y dos punteros que indican la posición de sus hijos. Uno o ambos de los punteros pueden tener un valor nulo si del nodo no cuelgan subárboles. Cada nodo de un árbol será un registro que contiene al menos tres campos:

- un campo INFORMACION de datos de un cierto tipo.
- un puntero al nodo del subárbol izquierdo (que puede ser nulo).
- un puntero al nodo del subárbol derecho (que puede ser nulo).

La Figura 5.15 ilustra el uso de los punteros para construir el mismo árbol representado en la Figura 5.14. Nótese cómo los punteros de los nodos terminales son punteros nulos.

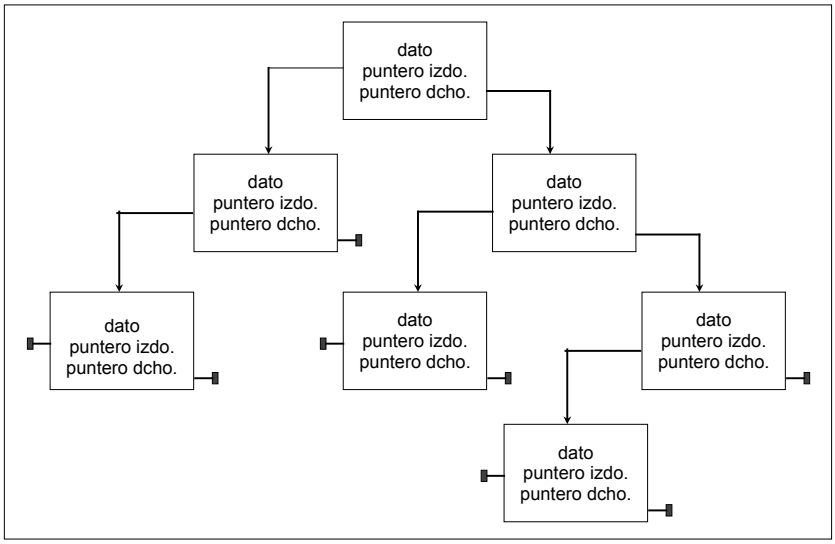


Fig. 5.15. *Punteros empleados para construir un árbol binario*

5.6.1.1 Recorrido de un árbol binario

Recorrer un árbol supone acceder a sus elementos de forma sistemática lo que supone llevar a cabo tres actividades:

1. Visitar el nodo raíz
2. Recorrer el subárbol izquierdo
3. Recorrer el subárbol derecho

Estas tres acciones repartidas en diferentes órdenes proporcionan diferentes recorridos del árbol, llamados: **pre-orden, post-orden, in-orden**. Su nombre refleja el momento en que se visita el nodo raíz. En el "in-orden" el raíz está en el medio del recorrido, en el "pre-orden", el raíz está el primero y en el "post-orden", el raíz está el último:

Recorrido pre-orden

1. Visitar el raíz
2. Recorrer el subárbol izquierdo en pre-orden
3. Recorrer el subárbol derecho en pre-orden

Recorrido in-orden

1. Recorrer el subárbol izquierdo en in-orden
2. Visitar el raíz
3. Recorrer el subárbol derecho en in-orden

Recorrido post-orden

1. Recorrer el subárbol izquierdo en post-orden
2. Recorrer el subárbol derecho en post-orden
3. Visitar el raíz

Obsérvese que todas estas definiciones de recorrido tienen naturaleza recursiva.

Ejemplo 6:

Determinar cual sería el resultado de los tres posibles recorridos para cada uno de los tres árboles mostrados en la Figura 5.16

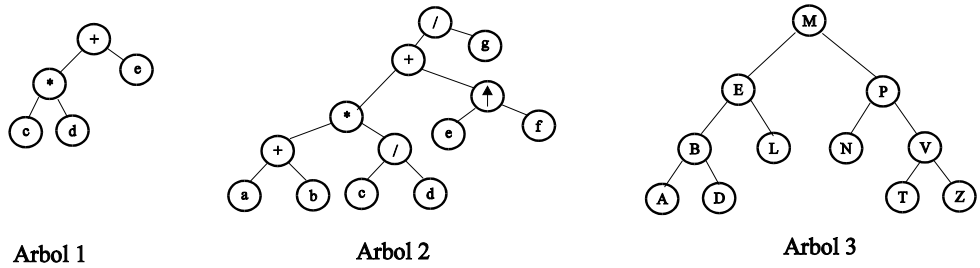


Fig. 5.16 *Recorrido de árboles binarios*

Árbol 1	Pre-orden	+ * cde
	In-orden	c * d + e
	Post-orden	cd * e +
Árbol 2	Pre-orden	/ + * + ab / cd ↑ efg
	In-orden	a + b * c/d + e ↑ f/g
	Post-orden	ab + cd/*ef ↑ + g
Árbol 3	Pre-orden	mebadlpnvtz
	In-orden	abdelmnptvz
	Post-orden	adblentzvpm

La característica esencial de los árboles es que cada uno de sus nodos puede estar conectado a subárboles, que a su vez tiene estructura arbórea. En otras palabras, siempre que se está en un árbol, la estructura inferior tiene carácter de árbol. En

este sentido un árbol es una estructura de datos **recursiva**, que puede manipularse mediante programas recursivos. Esta propiedad de los árboles es la que los hace más interesantes desde un punto de vista informático y por ello se utilizan ampliamente, como por ejemplo: los módulos de muchos programas se enlazan como si de árboles se tratara, la estructura que emplean muchos sistemas operativos para manejar los ficheros son árboles, algunos ordenadores se refieren a su memoria como si ésta estuviera fragmentada en forma de árbol; asimismo acabamos de ver cómo los árboles se usan para representar operaciones aritméticas.

5.6.2 ÁRBOL BINARIO DE BÚSQUEDA

Ya hemos hecho notar que una de las aplicaciones que se dan más frecuentemente en informática, es la de manejar una colección de datos sobre los cuales se efectúan de forma constante operaciones de búsqueda inserción y borrado (pensemos por ejemplo en el trabajo habitual de un servicio de reserva en una agencia de viajes). A lo largo de este capítulo hemos visto que cada una de estas tres operaciones elementales se resuelven de forma distinta, según la estructura elegida para organizar esta colección de datos sea un vector o una lista enlazada. Sabemos que la inserción y el borrado se mejoran sensiblemente, si elegimos una lista en vez de un vector, por el contrario, para la búsqueda parece claramente preferible trabajar con un vector ordenado. Para superar esta situación, vamos a describir una variante del árbol binario con la que podemos localizar, insertar y borrar con mayor eficacia. Ello nos dará una nueva posibilidad a la hora de programar, al poder seleccionar nuevas estructuras, que nos permitan utilizar nuevos y mejores algoritmos.

Llamaremos **árbol binario de búsqueda** a un árbol binario construido de acuerdo con el procedimiento siguiente:

1. El primer elemento se utiliza para crear el nodo raíz.
2. Los valores del árbol deben ser tales que pueda existir un orden (entero, real, lógico o carácter e incluso definido por el usuario si se tiene un orden total con él).
3. En cualquier nodo, todos los valores del subárbol izquierdo del nodo son menores o iguales que el valor del nodo. De modo similar todos los valores del subárbol derecho deben ser mayores que los valores del nodo.

Para este tipo de árbol, es sencillo probar que su recorrido “in-orden” obtiene los valores debidamente ordenados, lo que nos será de gran utilidad. Así, por ejemplo, en la Figura 5.17 se muestra un árbol binario de búsqueda.

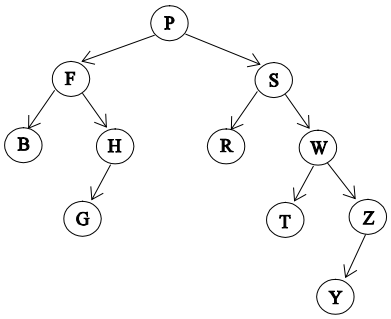


Fig. 5.17 *Árbol binario de búsqueda*

El recorrido *in-orden* del árbol de la figura 5.17 es: B F G H P R S T W Y Z. Ello nos permitirá almacenar y procesar un conjunto ordenado, con bastante facilidad, sin que tengamos que proceder a largas operaciones de readaptación. El paso de un conjunto cualquiera a un árbol binario de búsqueda es afortunadamente fácil, como muestra el ejemplo siguiente:

Ejemplo 7:

Supongamos que se dispone de un vector que contiene los siguiente caracteres:

D F E B A C G

para expresarlo mediante un árbol binario de búsqueda, vamos a seguir el algoritmo:

1. Nodo raíz del árbol: D.

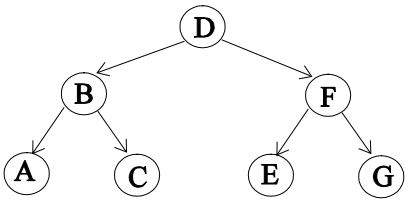
2. El siguiente elemento se convierte en el descendente derecho, dado que F alfabéticamente es mayor que D.

3. A continuación, se compara E con el raíz. Dado que E es mayor que D, pasará a ser un hijo de F y como E < F será el hijo izquierdo.

4. El siguiente elemento B se compara con el raíz D y como B<D y es el primer elemento que cumple esta condición, B será el hijo izquierdo de D.

5. Se repiten los pasos hasta el último elemento.

El árbol binario de búsqueda resultante sería:



Nótese que en caso de que el vector estuviese previamente ordenado (ascendente o descendentemente) el árbol de búsqueda que obtendríamos sería en realidad una lista.

Una vez definida esta nueva estructura tratemos de constatar su utilidad, cuando nos enfrentamos a esta hipotética aplicación, en la que las tres operaciones de búsqueda, inserción y borrado son muy frecuentes. Por motivos de espacio, no vamos a bajar hasta la especificación completa de los algoritmos correspondientes a cada una de estas tres operaciones, sólo presentaremos un esbozo del diseño correspondiente.

Búsqueda de un elemento

La búsqueda en un árbol binario ordenado es dicotómica, ya que a cada examen de un nodo, se elimina aquel de los subárboles que no contiene el valor buscado (valores todos inferiores o todos superiores).

El algoritmo de búsqueda del elemento -llamado *clave* - se realiza comparándolo con la raíz del árbol. Si no es el mismo, se pasa el subárbol izquierdo o derecho según sea el resultado de la comparación y se repite la búsqueda en ese subárbol, de forma recursiva.

La terminación del procedimiento se producirá cuando:

- se encuentra la clave
- no se encuentra la clave; y se llega a encontrar un subárbol vacío.

Así por ejemplo, si buscamos “W” en el árbol de la figura 5.17, se visitarán los nodos “P”, “S”, “W”. Si, en el mismo árbol, buscamos “X”, se visitarán los nodos “P”, “S”, “W”, “Z”, “Y” y se alcanza un subárbol vacío bajo la “Y” sin encontrar la clave.

Insertar un elemento

Para insertar un elemento en un árbol hay que comprobar en primer lugar que áquel no se encuentre ya en el árbol, dado que en este caso no precisa ser insertado (de hecho ésta es una comprobación que también deberíamos hacer trabajando con vectores o listas enlazadas, lo que supone que cada inserción se acompaña, en cierta manera, de un proceso de búsqueda). Si el elemento no existe, la inserción se realiza en un nodo en el que al menos uno de los dos punteros IZQ o DER tenga valor **nil**, con lo cual el nuevo elemento de inserta como una nueva hoja del árbol, sea cual sea su valor (Ver Figura 5.18).

La inserción no varía mucho del propio proceso de búsqueda, pues realmente vamos a insertar el nodo en la posición que ésta ocuparía, si ya se encontrara en el árbol. Para ello, se desciende en el árbol a partir del nodo raíz, dirigiéndose de izquierda a derecha de un nodo, según que el valor a insertar sea inferior o superior al valor del campo INFO de este nodo. Cuando se alcanza un nodo del árbol en que no se puede continuar, el nuevo elemento se engancha a la izquierda o derecha de este nodo, en función de que su valor sea inferior o superior al del nodo alcanzado.

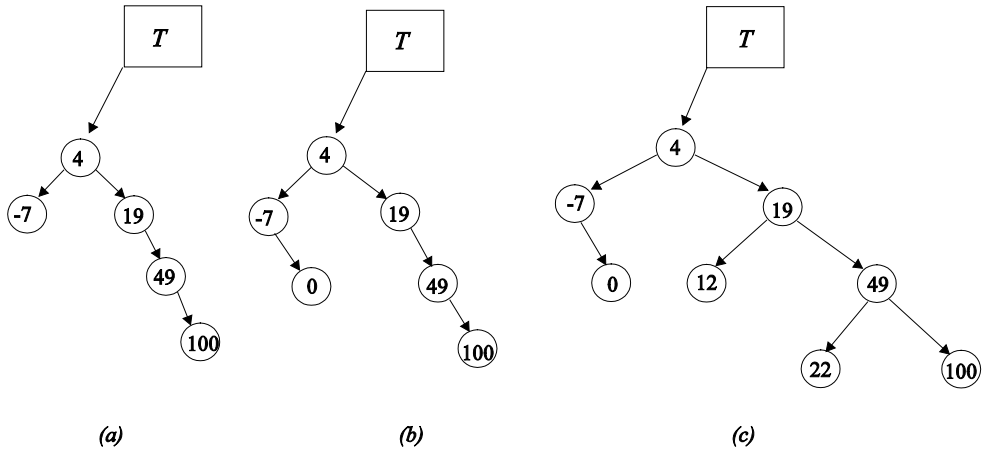


Fig. 5.18 *Ejemplo de Inserciones en un árbol de búsqueda binaria: (a), insertar 100; (b), insertar (o); (c), insertar 22 y 12.*

Eliminación de un elemento

La eliminación de un elemento debe hacerse conservando el orden de los elementos del árbol. Se consideran diferentes casos según la posición del elemento o nodo en el árbol a eliminar:

- si el elemento es una hoja, se suprime simplemente;

- si el elemento no tiene más que un descendiente, se sustituye entonces por ese descendiente (ver Figura 5.19);
- si el elemento tiene dos descendientes, la situación es un poco más complicada ya que la simple sustitución de un nodo por uno de sus hijos conllevaría la pérdida de estructura de árbol binario. En este caso el nodo a eliminar se debe sustituir por un descendiente más a la derecha o a la izquierda, de modo que siga conservando la ordenación (ver Figura 5.20).

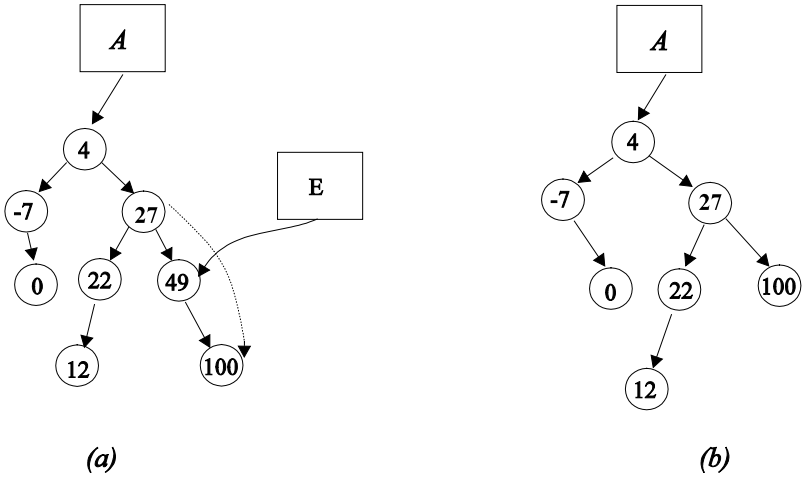


Fig. 5.19 *Ejemplo de Eliminación de un nodo (49) con un solo subárbol a) antes de la eliminación b) después de la eliminación*

Para poder realizar estas acciones, será preciso conocer la siguiente información del nodo a eliminar:

- Su posición en el árbol;
- La dirección de su padre;
- La dirección relativa a su ascendencia, es decir si el nodo a eliminar es un hijo derecho o izquierdo.

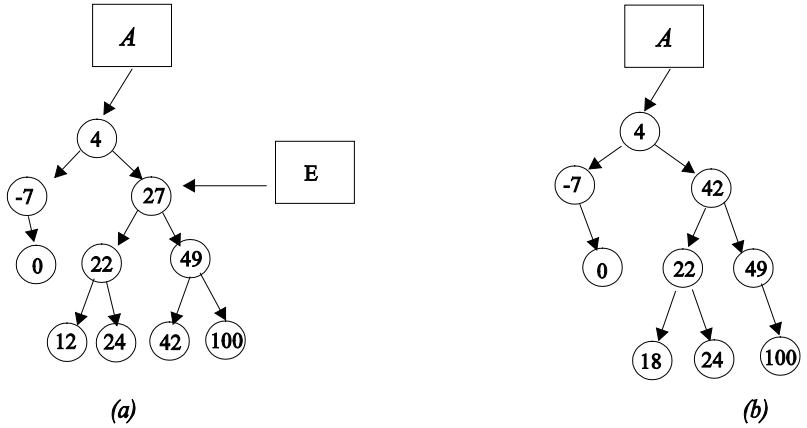


Fig. 5.20:*Ejemplo de eliminación de un nodo (27) con dos subárboles no nulos. Se ha sustituido por el elemento 42 que es un nodo sucesor siguiente en orden ascendente*.

Los algoritmos correspondientes no son difíciles aunque su detalle cae fuera de la óptica de este capítulo y se dejan como ejercicio para el lector.