DEPARTAMENTO DE ELECTRÓNICA, TELEMÁTICA Y AUTOMÁTICA

PROGRAMACIÓN

Estructuras de Datos Dinámicas

Carmen Nieves Ojeda Guerra
Ernestina A. Martel Jordán
Carlos M. Ramírez Casañas
Miguel Angel Quintana Suárez

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

Impreso en la Universidad de Las Palmas de G.C.
Campus Universitario de Tafira. Las Palmas G.C. (35017)
Febrero de 1997

D.L. GC-1-1997
Fotocopiadora
Marca: Rank Xerox
Modelo: 5090
Número de serie: 1104236487

Carmen Nieves Ojeda Guerra
Ernestina A. Martel Jordán
Carlos M. Ramírez Casañas
Miguel Angel Quintana Suárez

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

PREFACIO

Estos tomos de apuntes se han editado para ayudar a los alumnos de la asignatura PROGRAMACIÓN del plan de estudios de Ingeniería Técnica de Telecomunicación, implantado en el curso académico 96-97 en la Universidad de Las Palmas de G.C.

La asignatura, para la cual estos tomos han sido editados, es troncal para la especialidad de Telemática y obligatoria para las especialidades de Sistemas de Telecomunicación, Sonido e Imagen y Sistemas Electrónicos. Su duración es cuatrimestral (2º cuatrimestre) y su carga docente es de 3 créditos (2 horas) de clases teóricas + 3 créditos (2 horas) de clases prácticas en el laboratorio, por semana. PROGRAMACIÓN es la asignatura de la carrera que se ocupa fundamentalmente del estudio de las *estructuras de datos dinámicas.*

Los autores agradecerán cualquier comentario y corrección de esta edición para mejorar ediciones posteriores.

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

Todas las marcas comerciales utilizadas en este tomo son marcas registradas

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

Índice General

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

Capítulo 1

ARCHIVOS

En este capítulo vamos a estudiar unas estructuras de datos que se caracterizan porque su tamaño es desconocido en tiempo de compilación del programa, al contrario de las que se han visto hasta ahora. La información que previamente se tiene de estas estructuras es la forma que van a tener, pero nunca la dimensión final que tendrán (es decir, no tienen *cardinalidad* finita). La primera estructura que veremos de este tipo son los *archivos* o *ficheros*. Los archivos físicamente se encuentran almacenados en un dispositivo de entrada/salida pudiendo el usuario acceder a la información que contienen, mediante instrucciones adecuadas.

Este capítulo se dividirá en dos grandes temas. El primero tratará sobre el *procesamiento de la información*, donde se estudiarán los archivos tipo *texto*. La información de este tipo de archivos está compuesta por ristras de caracteres y forman una parte importante en el cálculo y proceso de datos. El segundo tema de que consta este capítulo se denomina *Archivos con tipo* y en él se explicarán los archivos que tienen una subestructura interna. Por ejemplo, el documento de identidad de un individuo está formado por una serie de datos que se podrían almacenar en un archivo siguiendo una estructura determinada. A su vez, el archivo completo estará formado por varias estructuras de ese tipo.

- El tema *Procesamiento de textos* básicamente está formado por:
  - Introducción al procesamiento de textos
  - Operaciones básicas con cadenas de caracteres
  - Otras funciones con cadenas de caracteres
  - Archivos de texto
  - Problemas
- El tema *Archivos con tipo* principalmente consta de:
  - Introducción a los archivos con tipos
  - Archivos con tipo y operaciones sobre ellos
  - Problemas

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

PROCESAMIENTO DE TEXTOS

1.1 INTRODUCCIÓN AL PROCESAMIENTO DE TEXTOS

El procesamiento de cadenas de caracteres es una tarea muy importante dentro de la programación de computadores. Como se dijo en la introducción del capítulo, el programador escribe ristras de caracteres (algoritmos) donde le indica al computador el orden de ejecución de su programa. Por tanto, el manejo de estas cadenas es fundamental y es lo que se explicará a lo largo de este tema.

Las cadenas de caracteres están formadas por caracteres concatenados unos con otros. Así, tenemos caracteres individuales como 'c', 'o','l' o 'a' que por sí mismos sólo representan un símbolo (letra), pero que si los concatenamos, 'cola', tenemos un texto (palabra) con un posible significado. Sobre las ristras de caracteres se pueden realizar múltiples operaciones, como por ejemplo:

- Calcular su longitud: 'cola' tiene longitud 4
- Insertar un carácter u otra cadena: 'corola', donde se ha insertado la cadena 'ro'
- Concatenar un carácter u otra cadena: 'colabora', donde se ha concatenado la cadena 'bora'
- Borrar un trozo de la cadena: 'la', donde hemos borrado la subcadena 'co'
- Búsqueda de subcadena: 'col' es una subcadena de la cadena 'cola'

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

A continuación describiremos más detalladamente cada una de estas operaciones y otras posibles que se pueden realizar a partir de ellas. Es importante mencionar que las cadenas de caracteres pueden ser vistas como vectores de tamaño igual a su longitud. Por tanto, se puede acceder a los elementos individuales de la cadena (a los caracteres) al igual que se accedía a los elementos de un vector. Por ejemplo, en la posición 1 de la cadena cad1 : = 'Programación' tenemos el carácter 'P', es decir, cad1[1] es igual a 'P' y en la posición 12 tenemos 'n', cad1[12] es igual a 'n'. Hay que tener en cuenta que no se puede acceder a una posición que sea mayor que la longitud de la cadena.

1.2 OPERACIONES BÁSICAS CON CADENAS DE CARACTERES

1.2.1 CÁLCULO DE SU LONGITUD

La longitud de una cadena es el número de caracteres que la forman, es decir, que están incluidos entre las comillas. Algorítmicamente se puede obtener la longitud de una cadena mediante la función:

longitud(cad) devuelve entero o entero largo

donde *cad* debe ser del tipo cadena. La función devolverá un entero o un entero largo que indicará la longitud. Por ejemplo:

longitud('Estudiamos las cadenas') es igual a 22

cad := 'Esto es una prueba'
longitud(cad) es igual a 18

Dos cosas hay que tener en cuenta. La primera es que los espacios en blanco se toman como caracteres (porque lo son) y la segunda es que el parámetro de la función *longitud* puede ser una cadena entre comillas (primer ejemplo) o una variable del tipo cadena (segundo ejemplo).

1.2.2 COMPARACIÓN ENTRE CADENAS

Al comparar dos cadenas hay que tener en cuenta la posición que ocupa cada carácter que las componen dentro del código usado por el computador -ASCII o EBCDIC-. Las cadenas se compararán carácter a carácter, es decir, primero se comparan los caracteres que ocupan la primera posición dentro de las cadenas, luego los que ocupan la segunda posición y así sucesivamente. Si una cadena es menor que la otra, se compararán los caracteres situados desde la primera posición hasta los colocados en la posición igual a la longitud de la cadena más corta, y así se verá cual es la menor. Si dos cadenas son las

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

mismas pero una de ellas es más corta que la otra, entonces la más corta es menor que la más larga. Así, tenemos los siguientes ejemplos:

'SANTANA' > 'SANCHEZ' es una expresión correcta,

' SANTANA' > 'SANCHEZ' es una expresión incorrecta porque el espacio en blanco (primera cadena) ocupa la posición 32 y la letra 'S' (segunda cadena) ocupa la posición 83 dentro de la tabla ASCII,

'1239' > '123 ' es una expresión correcta porque no estamos hablando de los números 1239 y 123, sino de las cadenas de caracteres '1239' y '123 ',

'JULIA' < 'julia' es una expresión correcta porque la letra 'J' ocupa la posición 74 y la letra 'j' ocupa la posición 106 dentro de la tabla ASCII,

'1239' < '548' es una expresión correcta porque no estamos hablando de los números 1239 y 548, sino de las cadenas de caracteres '1239' y '548' donde el '1' ocupa la posición 49 y el '5' la posición 53 de la tabla ASCII.

La comparación de dos cadenas se utiliza como expresión de una estructura condicional (*si-entonces-si no*) o repetitiva (*mientras-hacer* o *repetir-hasta que*). A continuación se muestra un ejemplo de una función a la que se le pasan dos cadenas *cad1* y *cad2* y devuelve un valor entero tal que, si el valor es igual a 1, la cadena *cad1* es mayor que la cadena *cad2*, si el valor es igual a 0, ambas cadenas son iguales y si el valor es -1, la cadena *cad1* es menor que la cadena *cad2*.

```
Funcion comparar(cad1,cad2) devuelve entero
  {cadena[20] cad1,cad2 por valor}
inicio
  si cad1 > cad2 entonces
    comparar:=1
  si no
    si cad1 = cad2 entonces
      comparar:=0
    si no
      comparar:=-1
    fin si
  fin si
fin funcion
```

1.2.3 CONCATENACIÓN DE CADENAS

Concatenar dos cadenas significa unirlas, es decir, el último carácter de la primera cadena se une al primer carácter de la segunda cadena, siendo la longitud de la cadena final, la suma de las longitudes de las dos cadenas originales. Ejemplos de concatenación son:

cad1 := 'Esto es '

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

cad2 := 'una prueba'
cadena_final es igual a 'Esto es una prueba'

Al igual que se pueden concatenar dos cadenas se pueden concatenar varias cadenas. De esta forma, el primer carácter de la tercera cadena será el siguiente al último carácter de la segunda cadena y así sucesivamente.

Para realizar la concatenación de dos o más cadenas se utiliza la función:

concatenar(cad1,cad2[,cad3,cad4,...,cadN]) devuelve cadena

donde los parámetros colocados dentro de los corchetes son opcionales y *cad1*, *cad2*, *cad3*, ..., *cadN* pueden ser cadenas de caracteres entre comillados o variables del tipo cadena. La función devolverá el resultado de la concatenación de las distintas cadenas pasadas como parámetros.

Es importante mencionar que si la suma de las longitudes de las cadenas que se van a concatenar es mayor que la longitud de la cadena destino, sólo se almacenarán aquellos caracteres que quepan en esta última cadena. El resto de los caracteres (que no quepan) se ignorarán, es decir, se produce un truncamiento al almacenarlas en la cadena final. Un ejemplo sería,

cad1 := 'Esto es '
cad2 := 'una prueba'

Si *cadena_final* fuese del tipo cadena[10] el resultado final sería 'Esto es un'.

Para entender mejor la concatenación vamos a realizar una función que pida por teclado el nombre y apellidos de una persona y devuelva todos estos datos dentro de una cadena, si caben. Si no caben, la función devuelve la cadena vacía.

```
Funcion datos() devuelve cadena[30]
variables
   cadena[20] nombre,ape1,ape2
inicio
   leer(nombre)
   leer(ape1)
   leer(ape2)
   si (longitud(nombre)+longitud(ape1)+longitud(ape2)) ≤ 30 entonces
      datos:=concatenar(nombre,ape1,ape2)
   si no
      datos:=''
   fin si
fin funcion
```

Otra forma posible de realizar la concatenación es mediante el operador '+', es decir, si queremos concatenar la cadena *cad1* y la cadena *cad2* del primer ejemplo se puede hacer de la siguiente forma:

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

cadena_final := cad1 + cad2

obteniéndose el mismo resultado anterior.

1.2.4 SUBCADENAS

Una subcadena es una parte de una cadena. Por ejemplo 'se llama' es una subcadena de 'Esta asignatura se llama Programación'. Para trabajar con subcadenas vamos a utilizar la función:

copiar(cadena_origen,inicio,numero) devuelve cadena

donde *cadena_origen* es la cadena original en la que está la subcadena, *inicio* es el valor a partir del cual se va a extrar la subcadena y *numero* es el número de caracteres que se van a extraer. Si *inicio* es mayor que la longitud total de la cadena original, la función devolverá la cadena vacía. Por otro lado, si el parámetro *numero* indica un valor mayor que los caracteres que realmente existen en la cadena desde *inicio*, la subrutina devuelve la subcadena que va desde *inicio* al último elemento de la cadena original. A continuación tenemos varios ejemplos para aclarar el concepto explicado.

cadena_origen:='Esta asignatura se llama Programación'
trozo1:=**copiar**(cadena_origen,10,10)
trozo2:=**copiar**(cadena_origen,26,20)
trozo3:=**copiar**(cadena_origen,39,2)

siendo:

trozo1 es igual a 'natura se '
trozo2 es igual a 'Programación'
trozo3 es igual a ''

1.2.5 BÚSQUEDA DE SUBCADENAS

Otra operación importante a realizar sobre una cadena es la búsqueda dentro de ella de otra subcadena de longitud menor. Por ejemplo, la subcadena 'cola' está contenida en la cadena 'La cola está vacía'. Para buscar una subcadena dentro de una cadena fuente vamos a utilizar la función:

posicion(subcad,cadena_origen) devuelve entero

La función devolverá la primera ocurrencia de la subcadena *subcad* dentro de la cadena *cadena_origen*. Si la subcadena no existe, la función devuelve como resultado el valor 0.

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

cadena_origen:='Esta asignatura se llama Programación'
posicion1:=**posicion**('asignatura',cadena)
posicion2:=**posicion**('t',cadena)
posicion3:=**posicion**('llamada',cadena)

siendo:

posicion1 es igual a 6
posicion2 es igual a 3
posicion3 es igual a 0

1.3 OTRAS OPERACIONES CON CADENAS DE CARACTERES

Además de las operaciones vistas hasta el momento, existen otras como la inserción, la eliminación, la conversión (a mayúsculas o minúsculas, a valores numéricos ...) que pasamos a comentar.

1.3.1 INSERCIÓN

Esta operación realiza la inserción de un carácter o cadena en otra cadena. Por ejemplo:

cad1:='+ estructuras de datos := '
cadena_origen:= 'algoritmos programa'

Si insertamos *cad1* en *cadena_origen* en la posición 12 tenemos como resultado 'algoritmos + estructuras de datos := programa'. El procedimiento que realiza la inserción será:

insertar(cadena_insertar,cadena_origen,pos)

donde la *cadena_insertar* es la cadena que vamos a insertar en *cadena_origen* desde la posición indicada por el parámetro *pos*. El parámetro *cadena_origen* contendrá el resultado de la inserción. Si la posición especificada es mayor que la longitud de la *cadena_origen* no se insertará nada. Por otro lado, si después de la inserción la longitud de la cadena resultante fuese mayor que la longitud de la cadena *cadena_origen*, se truncará el resultado. Veamos los siguiente ejemplos:

insertar(cad1,cadena_origen,5)
el resultado será *cadena_origen* igual a 'algo+ estructuras de datos := ritmos programa'

insertar(cad1,cadena_origen,20)
el resultado será *cadena_origen* igual a 'algoritmos programa+ estructuras de datos := ',

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

si la variable *cadena_origen* tiene como longitud mayor o igual a 45

insertar(cad1,cadena_origen,20)
el resultado será *cadena_origen* igual a 'algoritmos programa+ estr', si la variable *cadena_origen* tiene como longitud máxima 25 caracteres

La operación de inserción se puede realizar también utilizando funciones vistas anteriormente. En el siguiente algoritmo vamos a realizar la inserción de una cadena llamada *cadena_inser* en otra llamada *cadena_final* a partir de una posición dada, utilizando las funciones: **longitud**, **copiar** y **concatenar**.

```
Procedimiento insertar_propio(cadena_inser,cadena_final,pos)
   {cadena[10] cadena_inser por valor}
   {cadena[30] cadena_final por referencia}
   {enteras pos por valor}
variables
   cadena[30] trozo1,trozo2
inicio
   trozo1:=copiar(cadena_final,1,pos-1)
   trozo2:=copiar(cadena_final,pos,longitud(cadena_final)-(pos-1))
   cadena_final:=concatenar(trozo1,cadena_inser,trozo2)
fin procedimiento
```

Un error que se comete en la inserción es suponer que siempre se puede insertar en una cadena. Ésto es cierto hasta cierto punto, ya que si consideramos la cadena como un vector e insertamos en elementos individuales de la misma, la inserción sólo se hará correctamente si la cadena no está vacía o si se quiere insertar en una posición menor a la última ocupada dentro de la cadena.

1.3.2 ELIMINACIÓN

Esta operación realiza la eliminación de un carácter o cadena de otra cadena. Por ejemplo si en la cadena 'comando' eliminamos la subcadena que empieza en la posición 3 y tiene como longitud 3 tenemos como resultado 'codo'. El procedimiento que realiza la eliminación será:

eliminar(cadena_origen,pos,numero)

donde se eliminarán tantos caracteres como indica el parámetro *numero* a partir de la posición que indica el parámetro *pos* en la cadena *cadena_origen*. El parámetro *cadena_origen* contendrá finalmente el resultado de la eliminación. Si la posición especificada es mayor que la longitud de la *cadena_origen* no se eliminará nada. Por otro lado, si el número de caracteres a borrar es mayor que el número de caracteres que hay desde la posición *pos* hasta el final de la cadena, sólo se borrarán los caracteres que se encuentran desde la posición *pos* hasta el final. Veamos los siguiente ejemplos:

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

cadena := 'Escuela Universitaria de Ingeniería'
eliminar(cadena,9,14)

el resultado será cadena igual a 'Escuela de Ingeniería'

cadena := 'Escuela Universitaria de Ingeniería'
eliminar(cadena,22,25)

el resultado será cadena igual a 'Escuela Universitaria'

cadena := 'Escuela Universitaria de Ingeniería'
eliminar(cadena,40,1)

el resultado será cadena igual a 'Escuela Universitaria de Ingeniería'

1.3.3 CONVERSIÓN A MAYÚSCULAS Y MINÚSCULAS

Mediante esta operación transformamos un carácter a su correspondiente en mayúsculas o minúsculas. Las funciones que realizan esta transformación serán:

Convertir_Mayusculas(car) devuelve caracter

Convertir_Minusculas(car) devuelve caracter

donde *car* es el carácter a convertir. El carácter convertido se devuelve como resultado de la función. A continuación vamos a ver un ejemplo que convierte una cadena a mayúsculas usando la función *Convertir_Mayusculas* y la función *longitud.*

```
Procedimiento cadena_mayusculas(cadena_origen)
   {cadena[30] cadena_origen por referencia}
variables
   enteras i
inicio
   desde i:=1 hasta longitud(cadena_origen) hacer
      cadena_origen[i]:=Convertir_Mayusculas(cadena_origen[i])
   fin desde
fin procedimiento
```

es decir, se está convirtiendo uno a uno los caracteres de la *cadena_origen* en mayúsculas y almacenándose en el mismo lugar que ocupaban inicialmente. Nótese que accedemos a los elementos individualmente como si fuera un vector de caracteres en vez de una cadena.

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

1.3.4 CONVERSIÓN DE VALORES NUMÉRICOS A CADENAS Y VICEVERSA

Mediante esta operación transformamos un número a una cadena y de una cadena a un número. Los procedimientos que realizan esta transformación serán:

Num_Str(valor,cadena)

Str_Num(cadena,valor,codigo)

Para el primer caso, se va a transformar el número *valor* (de tipo real o entero) en una cadena, almacenándose el resultado en el parámetro *cadena*. Para el segundo caso, se va a transformar la cadena *cadena* en un valor numérico *valor*. Si se puede hacer la transformación, en el parámetro *codigo* se almacenará un 0, pero si no se pudo transformar la cadena, en *codigo* se indicará la posición dentro de *cadena* donde se produjo un error (no se pudo convertir el carácter en un número). El tipo del parámetro *valor* puede ser real o entero, por tanto, la cadena se transformará en un valor numérico real o entero. Veamos los siguientes ejemplos:

cadena := '345'
Str_Num(cadena,valor,codigo)

donde *valor* almacenará el número entero 345 (si *valor* es entero) o el número real 345.0 (si *valor* es real) y *codigo* almacenará el valor 0.

cadena := '34a5'
Str_Num(cadena,valor,codigo)

donde *valor* toma un valor indefinido porque no se ha podido convertir la *cadena*. El parámetro *codigo* almacenará el valor 3 porque en ese carácter fue donde se produjo el error (el carácter 'a' no se puede convertir a un valor numérico). De igual forma,

valor := 101
Num_Str(valor,cadena)

donde *cadena* almacenará la cadena '101'.

A continuación veamos un algoritmo que sustituye dentro de una frase dada, todas aquellas secuencias de dos o más blancos por un único carácter blanco. El algoritmo devuelve en la cadena de entrada, el resultado de la sustitución y se utilizarán procedimientos y funciones estudiados hasta el momento.

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
Procedimiento sustituir(frase)
   {cadena[30] frase por referencia}
variables
   cadena[30] auxiliar
   enteras inic_blanco
inicio
   auxiliar:=''
   inic_blanco:=posicion(' ',frase)
   mientras inic_blanco<>0 hacer
      auxiliar:=concatenar(auxiliar,copiar(frase,1,inic_blanco))
      inic_blanco:=inic_blanco+1
      mientras (inic_blanco≤longitud(frase)) y (frase[inic_blanco]=' ') hacer
         inic_blanco:=inic_blanco+1
      fin mientras
      eliminar(frase,1,inic_blanco-1)
      inic_blanco:=posicion(' ',frase)
   fin mientras
   frase:=auxiliar
fin procedimiento
```

1.4 ARCHIVOS DE TEXTO

El procesamiento de textos visto hasta ahora es extremadamente útil cuando trabajamos con archivos de texto. Antes de comenzar a explicar los archivos de texto y su manejo, vamos a ver una nueva estructura de datos que denominaremos *secuencia.*

La secuencia es una estructura de datos que está formada por un conjunto de elementos del mismo tipo al que se le denomina *tipo básico* de la secuencia. El tipo básico de la secuencia puede ser cualquiera de los estudiados hasta el momento: enteros, reales, vectores, registros, etc.

$elemento_0$
$elemento_1$
$elemento_2$
$elemento_3$
$elemento_4$
$elemento_5$
.....
$elemento_{n-1}$

Visto de esta manera, podríamos pensar que la diferencia entre una secuencia y un vector no es muy grande. A fin de cuentas, ambos están formados por un número determinado de elementos del mismo tipo. Sin embargo, existe una diferencia fundamental entre ambos que los caracteriza: mientras el tamaño (*cardinalidad*) del vector se tiene que fijar previamente, antes de ejecutar el programa, el tamaño final de la secuencia no se conoce de

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

antemano. Es decir, mientras los primeros tienen cardinalidad finita, las segundas tienen cardinalidad infinita.

Una consecuencia inmediata de lo explicado en el párrafo anterior es que si el tamaño de la secuencia no se conoce previamente ¿dónde podemos almacenarla? y ¿qué tamaño debe tener ese espacio donde se va a almacenar? Para explicar la razón de estas dos preguntas veamos un ejemplo sobre un vector.

Cuando un vector es declarado dentro de nuestro programa, específicamente se indica su tamaño,

```
.
.
tipo
   tipo_vector=vector[30] de enteros
.
.
variables
   tipo_vector mi_vector
.
.
```

como vemos en el ejemplo, el vector *mi_vector* tiene dimensión 30 y almacenará valores enteros. Desde que ésto se especifica en el programa, el tamaño del vector no variará en toda la vida del programa (por esta razón, la dimensión de un vector tiene que ser un número fijo -constante-). Si no hubiera espacio disponible para almacenar los valores de nuestro vector, simplemente el programa no se podría ejecutar y si hubiese espacio suficiente para él, estamos seguros que el programa se ejecutará sin problemas (siempre que esté correctamente escrito). Sin embargo, pongamos ahora el caso de una secuencia,

```
.
.
tipo
   tipo_secuencia=secuencia de enteros
.
.
variables
   tipo_secuencia mi_secuencia
.
.
```

donde tenemos una variable *mi_secuencia* que almacena una serie de valores enteros. Como podemos ver, la información que tenemos en este momento del tamaño último que tendrá la secuencia es nula, por tanto, ¿qué espacio se le dedica?, es más, ¿cómo sabemos si la secuencia completa cabe en el almacenamiento que se le destina? Debido a ésto, el almacenamiento para la secuencia tiene que ser asignado en tiempo de ejecución de nuestro programa. Ésta, es una característica que comparten algunas estructuras de datos a las que se les llama *estructuras de datos dinámicas* y que veremos en los próximos capítulos de este tomo. Sin embargo, aquí tenemos un primer punto de discordia. Algunos autores

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

hablan de la estructura secuencia como una estructura de datos dinámica, mientras que otros defienden que es una estructura de datos estática especial porque tiene cardinalidad infinita (recordar que las estructuras estáticas tienen todas cardinalidad finita). Éstos últimos se basan en el hecho de que el manejo de la memoria es simple en el caso de una secuencia para no incluirlas dentro de las estructuras dinámicas propiamente dichas (léase listas, árboles, grafos). El manejo de estas estructuras se realiza a través de mecanismos que ofrece el sistema operativo del computador, ya que su almacenamiento se realiza en memoria secundaria (discos, cintas magnéticas, etc.).

En nuestro caso, cuando queramos utilizar la estructura secuencia utilizaremos la siguiente nomenclatura:

<nombre_de_variable> = **archivo de** <tipo_definido>

donde *nombre_de_variable* es cualquier nombre permitido y *tipo_definido* es cualquier tipo de los disponible en el lenguaje utilizado o bien un tipo definido por el usuario.

Una vez hecha esta introducción se puede definir un archivo como una unidad de información que tiene un nombre específico, donde cada una de las unidades que lo forman se caracterizan porque comparten un mismo tipo de datos. Como explicamos al inicio de esta sección vamos a analizar los archivos de texto que, como su nombre indica, son estructuras donde el tipo básico es un carácter (pertenecientes a un determinado código, por ejemplo el ASCII) y están formados por líneas de caracteres separadas por una marca de fin de línea (*EOLN -end of line-*). Los archivos de texto se pueden crear utilizando cualquier editor de texto (por ejemplo el *edit* del sistema operativo MS-DOS) y siempre finalizan con una marca del fin de archivo (*EOF -end of file-*).

Los archivos de texto se dice también que son *archivos secuenciales* porque el elemento i-ésimo del archivo va siempre antes que el elemento i+1-ésimo. Un ejemplo de archivo de texto es simplemente el programa que escribimos en un lenguaje de programación, para luego compilar y ejecutar.

Sobre los archivos de texto se pueden realizar las siguientes operaciones: declaración, asignación, creación, apertura, lectura, escritura, cierre y otras funciones que pasamos a describir.

1.4.1 DECLARACIÓN Y ASIGNACIÓN

La forma de declarar, en lenguaje algorítmico, una variable cuyo tipo es un archivo de texto es la siguiente:

variable
  archivo de texto nombre_variable

De esta forma, el programa conocerá que se va a utilizar un archivo de textos y que para referirnos a él usaremos la variable declarada. Sin embargo, y aunque se haya declarado la variable de este tipo, ésto no quiere decir que se haya asignado un archivo específico

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

dentro de la memoria secundaria. Para asignar un archivo físico a la variable (*archivo lógico*) hay que llamar al siguiente procedimiento:

asignar(nombre_variable,nombre_fichero)

donde *nombre_variable* es el nombre de la variable o archivo lógico y *nombre_fichero* es el nombre externo del fichero guardado en memoria secundaria. Hay que tener en cuenta, que cuando se hace la asignación no se comprueba si el archivo físico existe o no. En la figura 1.1 podemos ver dos asignaciones: en la primera asignación el archivo físico existe y en la segunda no existe.

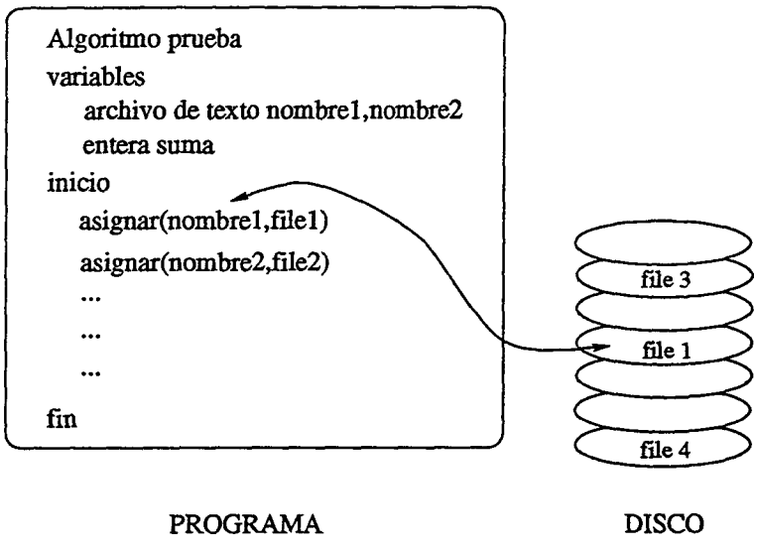


Figura 1.1: Asignación de variables a archivos en disco

1.4.2 CREACIÓN Y APERTURA

Una vez que se ha asignado un nombre de variable a un archivo físico ya se puede operar sobre él. Sin embargo, previamente a realizar cualquier tipo de operación permitida, el archivo físico tiene que existir. Para comprobar la existencia de un archivo en disco se utilizan las dos funciones diferentes:

abrir(nombre_variable) devuelve logica

unir(nombre_variable) devuelve logica

Las dos funciones se invocan cuando se quiere abrir un archivo de texto (que ha sido asignado a la variable *nombre_variable*) y devuelven un valor lógico, de forma que si ese valor es "verdadero", el archivo existía en el soporte físico y si el valor es "falso", el archivo no existía. La diferencia entre ellos es que, mientras el primero se llama cuando se quiere abrir un archivo para lectura, el segundo se llama cuando se quiere añadir al final del

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

archivo, es decir, escribir en él. En ambos casos, la llamada a estas subrutinas produce un posicionamiento del dispositivo lectura/escritura en el soporte físico del archivo, si existe.

En caso de no existir el archivo asignado, se utiliza un nuevo procedimiento que se encarga de *crearlo* y posteriormente de abrirlo:

```
crear(nombre_variable)
```

En el siguiente ejemplo vemos como se debe abrir un archivo de texto (para lectura) de forma que si éste no existe se creará, abriéndose posteriormente.

```
Procedimiento activar(nombre_variable,nombre_archivo)
   {archivo de texto nombre_variable por referencia}
   {cadena[20] nombre_archivo por valor}
inicio
   asignar(nombre_variable,nombre_archivo)
   si no abrir(nombre_variable) entonces
      crear(nombre_variable)
   fin si
fin procedimiento
```

En el algoritmo vemos que después de asignar el nombre de archivo físico al nombre de archivo lógico se llama a la función *abrir*. Esta función, si no puede abrir el archivo (porque no existe) devolverá "falso" de forma que se entrará dentro de la estructura condicional y se pasará a crear el archivo con el procedimiento *crear*. En caso de que el archivo ya existiera, la función *abrir* devolverá "verdadero" y, por tanto, no creará de nuevo el archivo. Hay que tener en cuenta que siempre que se abre un archivo mediante el procedimiento de creación se borrará el contenido que tenía el archivo hasta ese momento.

En este punto debemos destacar que al ser los archivos de texto exclusivamente secuenciales sólo se puede escribir al final del mismo. Debido a ésto es por lo que existen dos formas de abrir un archivo de texto. Si el archivo de texto se abre para escritura, **sólo** se podrá escribir en él y si se abre para lectura **sólo** se podrá leer de él. Esta idea se puede entender si observamos la figura 1.2.

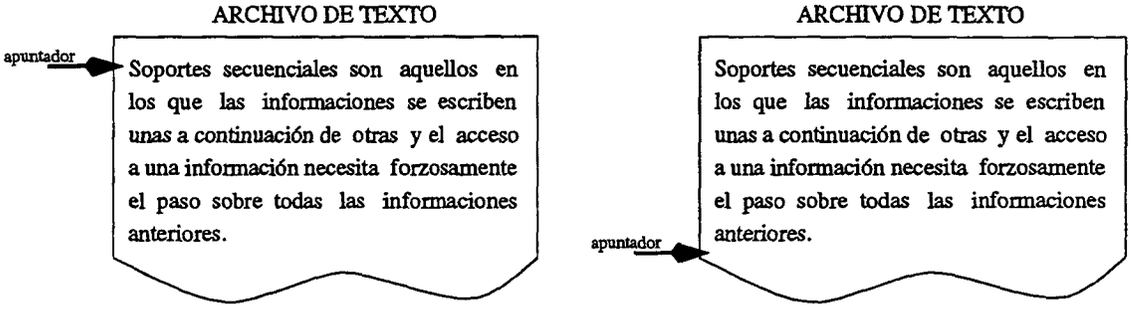


Figura 1.2: a) Después de *abrir*. b) Después de *unir*

En dicha figura, tenemos un apuntador que direcciona una "posición" dentro del archivo de texto. Después de una operación de apertura de archivo (operación *abrir*), el

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

apuntador señalará a la primera posición. Ésto es así ya que al abrir el archivo para lectura es lógico que empecemos a leerlo desde la primera posición. Sin embargo, cuando se abre el archivo para escritura (operación *unir*), el apuntador señalará la posición siguiente a la última del archivo, ya que lo correcto es que se añada al final de lo que ya está escrito. Por tanto, el apuntador de la figura apuntará siempre a la posición dentro del archivo con la que se va a trabajar.

1.4.3 LECTURA Y ESCRITURA

Las operaciones que se pueden realizar sobre los archivos de texto son básicamente la lectura y escritura de uno o más caracteres. Los procedimientos que se encargan de leer y escribir en un archivo de texto son:

leer(nombre_fichero,n1[,n2,n3,...,nN])

escribir(nombre_fichero,n1[,n2,n3,...,nN])

donde *n1*, *n2*,...,*nN* son nombres de variables de tipo carácter, entero, real o cadena y *nombre_fichero* es la variable del tipo archivo de texto. Si se invoca al procedimiento *leer*, se leerá del archivo de texto asignado a la variable *nombre_fichero* tantos elementos como indiquen las variables *n1*, *n2*,...,*nN* y que sean del mismo tipo. Si se invoca al procedimiento *escribir*, se escribirán en el archivo de texto los datos que almacenan las variables *n1*, *n2*,...,*nN*. Hay que tener en cuenta que ésto no está en contradicción con lo dicho anteriormente de que el archivo de texto almacena sólo caracteres. Realmente los datos que están guardados en él son almacenados como caracteres aunque inicialmente fueran de otro tipo. A continuación mostraremos mediante gráficas (figuras 1.3, 1.4, 1.5 y 1.6) distintas situaciones que se pueden dar a la hora de leer o escribir en un archivo de texto donde los datos a almacenar y leer son cadenas de caracteres.

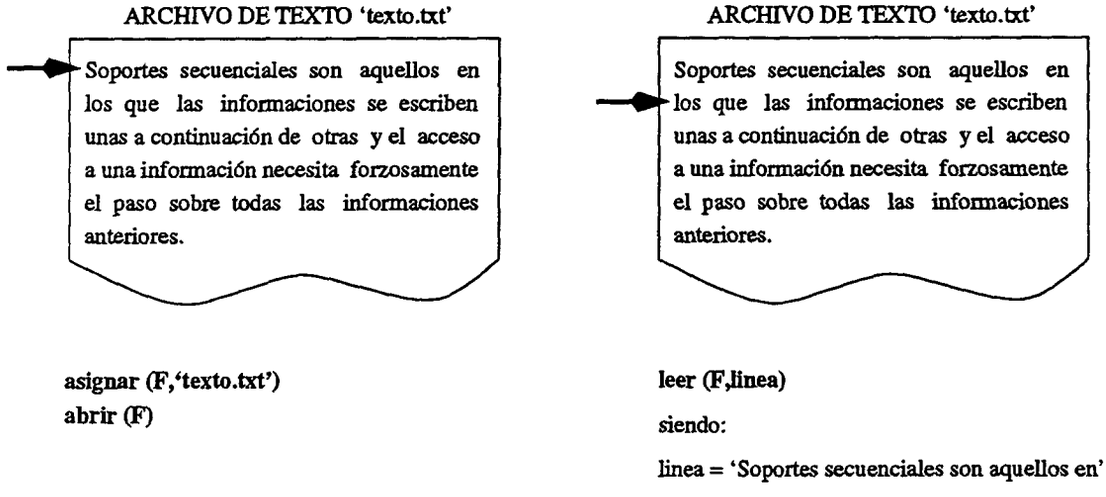


Figura 1.3: Lectura de un archivo de texto

Cuando se abre el archivo de texto, el apuntador se colocará en la primera línea del archivo. Después de una operación de lectura, el apuntador "saltará" a la siguiente línea,

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

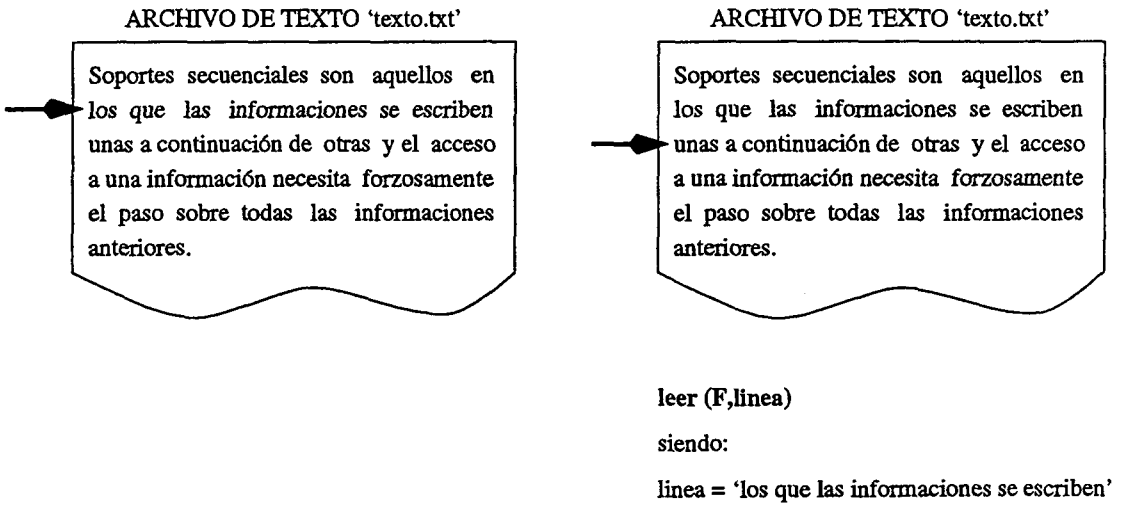


Figura 1.4: Lectura de un archivo de texto

almacenándose en el parámetro del procedimiento *leer* la cadena leída (en nuestro ejemplo -figuras 1.3 y 1.4- dicho parámetro es *linea*).

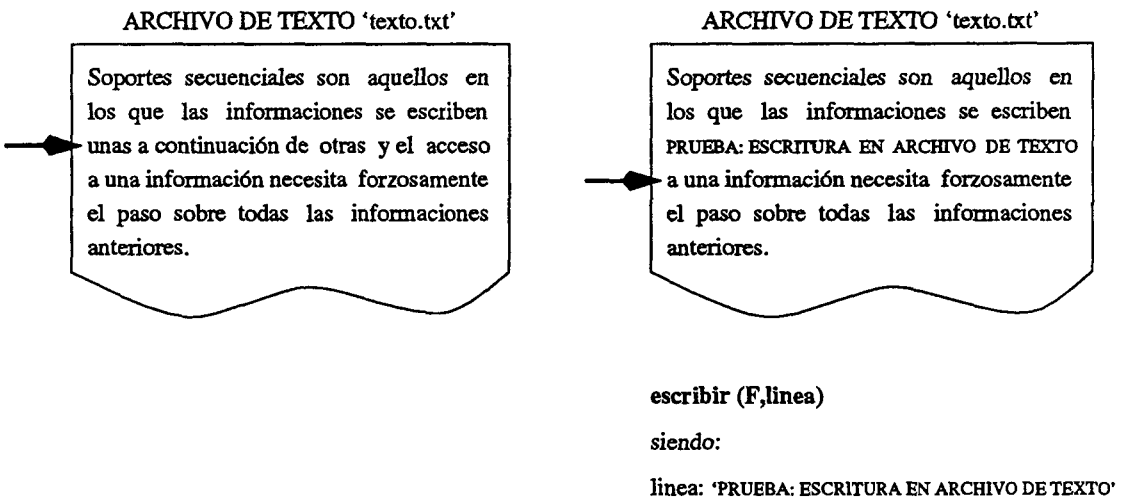


Figura 1.5: Escritura de un archivo de texto

De la misma forma, cuando se quiere realizar una operación de escritura en archivo de texto (figura 1.5), se escribirá en la posición señalada por el apuntador y una vez realizada la escritura, el apuntador "saltará" a la siguiente línea del archivo. Hay que recalcar que si se escribe en una posición intermedia de un archivo se eliminará la información que había previamente almacenada. La información **no** se desplaza dejando hueco para la nueva línea de caracteres escrita. Por otro lado, si se añade al final del archivo (figura 1.6), el apuntador pasará a señalar la última posición y a continuación se realiza la operación de escritura.

Dos preguntas interesantes y que nos surgen en este momento serían las siguientes: ¿qué pasa si la dimensión de la cadena *linea* fuese menor que la línea del archivo de texto que se va a leer? y, por otro lado, ¿cómo hacemos para separar una línea de otra? La respuesta a la primera pregunta es que el apuntador, en este caso, no señalará la siguiente línea del archivo de texto sino al carácter siguiente al último que fue leído (figura 1.7). Para responder a la segunda pregunta hay que tener en cuenta que para separar las distintas líneas de texto hay que incluir el carácter de fin de línea (*<CR> o retorno de carro*).

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

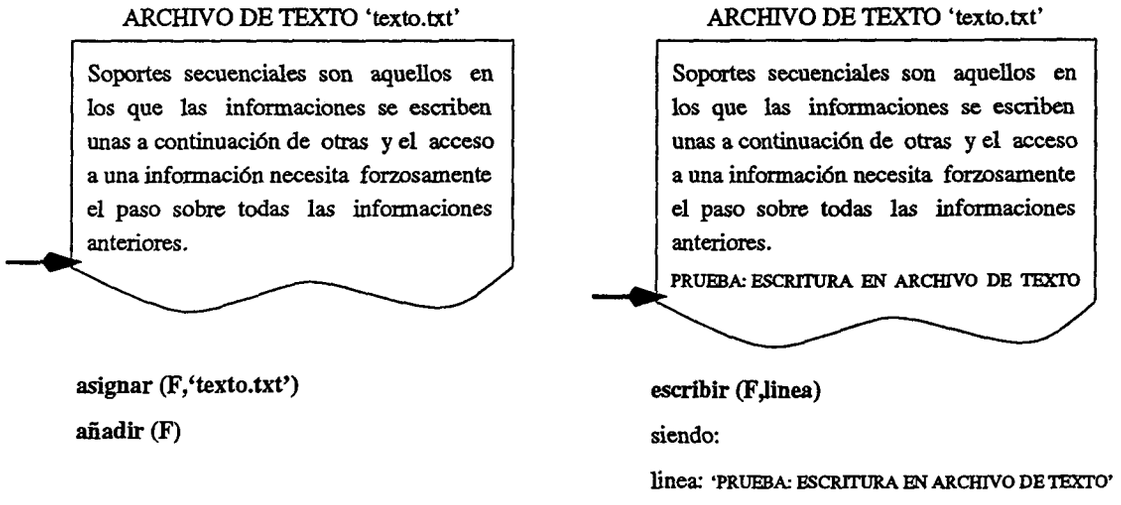


Figura 1.6: Escritura de un archivo de texto

En algunos lenguajes de programación esta situación se especifica de distintas maneras (por ejemplo en PASCAL se hará utilizando para escribir la orden *writeln(...)* en vez de *write(...)*), sin embargo, en lenguaje algorítmico vamos a utilizar el símbolo ←↩, quedando por tanto el procedimiento de escritura como sigue:

escribir(nombre_fichero,n1[,n2,n3,...,nN],←↩)

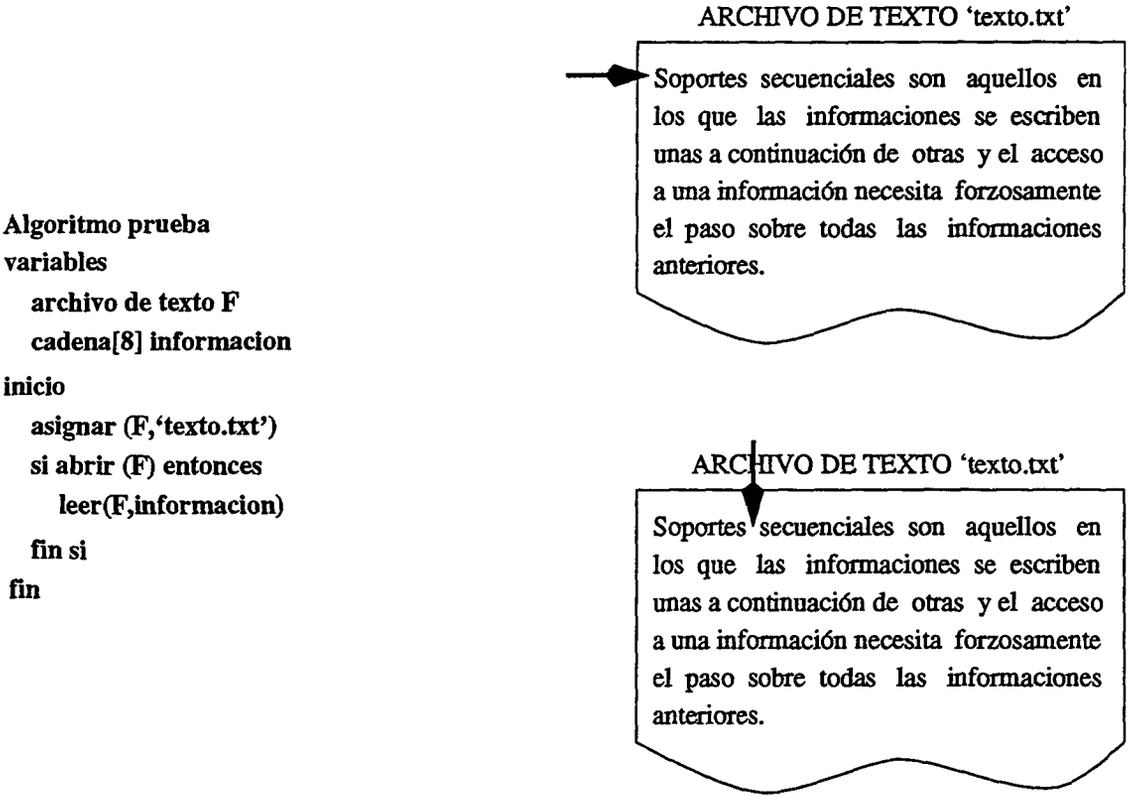


Figura 1.7: Lectura de un archivo de texto

Esto último hay que tenerlo en cuenta también cuando vayamos a leer del archivo (tendremos que leer también el elemento retorno de carro, que será un carácter).

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

1.4.4 CIERRE

Siempre que se realice la apertura de un archivo de texto se debe realizar también su cierre. El procedimiento que se encarga de cerrar un archivo de texto es:

cerrar(nombre_fichero)

Una vez que se cierra un archivo no se puede realizar ninguna operación sobre él. Para que ésto sea posible, habrá que volver a abrir el archivo, realizar las operaciones que se quieran y cerrarlo de nuevo. Se debe tomar como norma que aquella subrutina encargada de abrir el archivo (llamar a *abrir(...)*) sea también la encargada de cerrarlo (llamar a *cerrar(...)*).

1.4.5 OTRAS FUNCIONES

Además de las operaciones explicadas anteriormente existen otras funciones que nos pueden dar a conocer el estado del archivo de texto devolviendo la posición del apuntador dentro del archivo. Estas dos funciones son *eoln -end of line-* y *eof - end of file-* que algorítmicamente se expresan:

eoln(nombre_fichero) devuelve logico

eof(nombre_fichero) devuelve logico

donde la primera función indicará con un resultado "verdadero" que el apuntador está al final de una línea de texto dentro del archivo y con un resultado "falso" el caso contrario. La segunda función devolverá "verdadero" si se ha llegado al final del archivo de texto y "falso" en caso contrario.

1.5 PROBLEMAS

1. Supongamos un archivo de texto 'informe.txt' que almacena los siguientes datos:

34378965	Ortega	Luis	8.3	22
90876547	Lorenzo	Clara	9.1	23
92938717	Pérez	José	4.2	24
.....				
.....				
45261739	López	Bernardo	9.0	21

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

donde cada columna corresponde con: DNI, apellido, nombre, nota y edad. Se pide, indicar la nota media de la clase, el DNI, nombre y apellido de la persona con mayor nota y el DNI, nombre y apellido de la persona con menor edad.

```
Algoritmo datos
constantes
   fichero='informe.txt'
   MAYOR_EDAD=150
   MENOR_NOTA=-1
tipos
   personal=registro
      cadena[8] dni
      cadena[20] apel,nombre
   fin registro

Procedimiento cambiar(reg,cad1,cad2,cad3)
   {personal reg por referencia}
   {cadena[8] cad1 por valor}
   {cadena[20] cad2,cad3 por valor}
inicio
   reg.dni:=cad1
   reg.apel:=cad2
   reg.nombre:=cad3
fin procedimiento

variables
   personal mayor_nota,menor_edad
   enteras elementos,edad,edadm
   real m_nota,nota,notam
   cadena[8] dni
   cadena[20] apel,nombre
   archivo de texto F
   logica codigo
   caracter rc
inicio
   asignar(F,fichero)
   elementos:=0
   m_nota:=0.0
   edadm:=MAYOR_EDAD
   notam:=MENOR_NOTA
   codigo:=abrir(F)
   mientras no eof(F) hacer
      leer(F,dni,apel,nombre,nota,edad,rc)
      si edad<edadm entonces
         cambiar(menor_edad,dni,apel,nombre)
         edadm:=edad
      fin si
      si nota>notam entonces
         cambiar(mayor_nota,dni,apel,nombre)
```

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
            notam:=nota
        fin si
        m_nota:=m_nota+nota
        elementos:=elementos+1
    fin mientras
    m_nota:=m_nota/elementos
    escribir('El alumno con nota máxima es:')
    escribir(mayor_nota.dni,mayor_nota.apel,mayor_nota.nombre,notam)
    escribir('El alumno con menor edad es:')
    escribir(menor_edad.dni,menor_edad.apel,menor_edad.nombre,edadm)
    escribir('La media de notas de los alumnos es:')
    escribir(m_nota)
    cerrar(F)
fin
```

Al analizar el algoritmo tenemos que darnos cuenta que debido a que el archivo existía y tenía datos inicialmente, no es necesario comprobar el valor de la variable lógica *codigo*, cuyo valor es "verdadero" (sin embargo, hay que utilizar dicha variable porque *abrir* es una función y éstas son siempre asignadas a variables de su mismo tipo). Por otro lado, y sin tener en cuenta si el archivo existe, dejamos al lector la respuesta a la siguiente pregunta: ¿por qué no se utiliza la estructura repetitiva **repetir** en vez de la estructura **mientras**?

2. Implementar un algoritmo que dado un archivo 'entrada.txt' genere otro archivo 'salida.txt'. Para generar el segundo archivo hay que recorrer el primer archivo buscando toda ocurrencia de una palabra dada (*antigua_palabra*) sustituyéndola por una nueva palabra (*nueva_palabra*). Las frases transformadas se irán escribiendo en el archivo 'salida.txt'. Suponemos que las líneas son como máximo de 80 caracteres incluyendo el carácter de retorno de carro.

```
Algoritmo transformar
constantes
    in='entrada.txt'
    out='salida.txt'

Procedimiento buscar(frase,antigua,nueva)
    {cadena[80] frase por referencia}
    {cadena[10] antigua,nueva por valor}
variables
    enteras p,tamano
inicio
    tamano:=longitud(antigua)
    p:=posicion(antigua,frase)
    mientras p<>0 hacer
        eliminar(frase,p,tamano)
        insertar(nueva,frase,p)
        p:=posicion(antigua,frase)
    fin mientras
fin procedimiento
```

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
variables
   cadena[80] frase
   cadena[10] antigua_palabra,nueva_palabra
   archivo de texto F1,F2
inicio
   asignar(F1,in)
   asignar(F2,out)
   leer(antigua_palabra,nueva_palabra)
   si abrir(F1) entonces
      crear(F2)
      mientras no eof(F1) hacer
         leer(F1,frase)
         buscar(frase,antigua_palabra,nueva_palabra)
         escribir(F2,frase)
      fin mientras
      cerrar(F1)
      cerrar(F2)
   si no
      escribir('El archivo ',in,'no existe')
   fin si
fin
```

En este ejemplo se puede observar que al leer y escribir no tenemos en cuenta el carácter de retorno de carro. Ésto se debe a que como la variable *frase* es de 80 caracteres, al leer una línea del archivo se lee también el retorno de carro y al escribir la línea en el archivo de salida, el retorno de carro se escribe también ya que está almacenado en dicha variable.

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

ARCHIVOS CON TIPO

1.1 INTRODUCCIÓN A LOS ARCHIVOS CON TIPO

Los archivos con tipo, archivos de acceso aleatorio, archivos binarios o directos se utilizan cuando la información que se quiere almacenar tiene una estructura interna o básica fija. En la figura 1.8 podemos ver un esquema del significado de estructura básica y de archivo con tipo. En este esquema se parte de una estructura básica formada por cinco campos (que pueden ser de cualquier tipo definido en capítulos anteriores) que se usará como pieza fundamental en la construcción del archivo (en el ejemplo hay seis piezas de la estructura básica). En la figura 1.9 tenemos un ejemplo real de este tipo de archivos donde la estructura básica es DATO_INDIVIDUAL y el archivo se llama FICHERO_DNI.

1.2 ARCHIVOS CON TIPO

Como se ha explicado en la introducción, un archivo binario es un conjunto de informaciones que están estructuradas en bloques que se denominan *registros*. Así, los seis registros representados en la figura 1.8 forman un archivo. A su vez, un registro se puede definir como una colección de *campos* donde cada campo es de un tipo determinado y guarda una cierta información.

El acceso a los registros de un archivo puede hacerse de forma directa, es decir, podemos acceder a la información que se encuentra en el registro tres del archivo siempre y cuando el archivo tenga mayor o igual a tres registros. Debido a ésto es por lo que

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

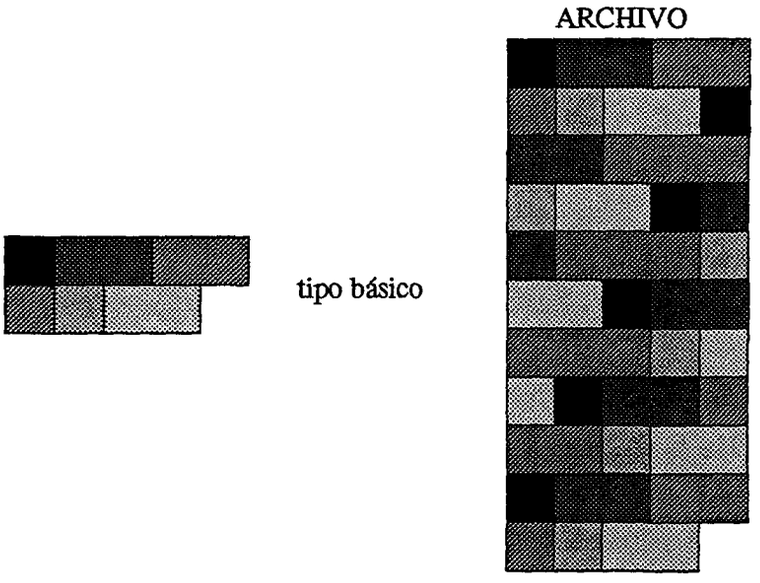


Figura 1.8: Estructura de un archivo binario

a los archivos con tipo o binarios también se les llama archivos de acceso aleatorio o directos. El acceso directo a un registro determinado se puede hacer porque todos los registros tienen el mismo tamaño, por tanto, ésta es una condición indispensable para que cualquier archivo con tipo pueda ser manipulado de forma correcta. Para entender este concepto supongamos que al ejemplo de la figura 1.8 alguien le ha añadido un registro de tamaño diferente a la estructura básica (ver figura 1.10). En la figura 1.11 vemos los registros cuarto, quinto y sexto del archivo modificado. Ante esta situación podemos entender que si quisieramos acceder al registro sexto tendremos problemas para recuperar la información de forma correcta (recuperaríamos parte de dos registros).

En los archivos binarios el concepto del apuntador visto en el tema anterior toma una especial relevancia. El apuntador siempre señala al siguiente registro a tratar dentro del archivo, por tanto, la posición del apuntador es un dato que siempre se debe conocer.

Sobre los archivos de acceso aleatorio se pueden realizar las siguientes operaciones: declaración, asignación, creación, apertura, lectura, escritura, cierre y otras funciones que se describirán a continuación.

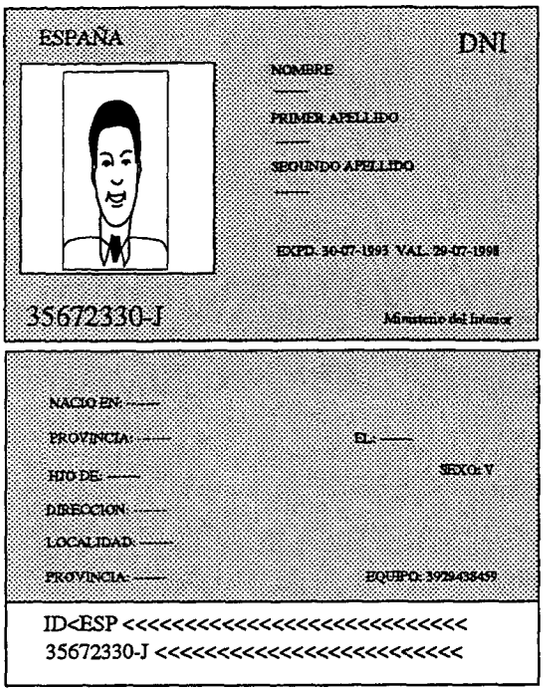
1.2.1 DECLARACIÓN Y ASIGNACIÓN

La forma de declarar en lenguaje algorítmico una variable cuyo tipo es un archivo con tipo es la siguiente:

variables
  archivo de nombre_tipo nombre_variable

donde *nombre_tipo* es un tipo especificado previamente. Al igual que en los archivos tipo texto, al hacer esta declaración el programa conocerá que se va a utilizar un archivo binario y que para referirnos a él usaremos la variable declarada *nombre_variable*, sin embargo

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006




```
tipo
    registro DATO_INDIVIDUAL
        cadena[9] DNI
        cadena[20] Nombre,Apellido1,Apellido2
        cadena[20] lugar_nacimiento, provincia_nacimiento
        cadena[8] dia_de_nacimiento
        cadena[30] hijo_de
        logica sexo
        cadena[30] direccion,localidad,provincia
    fin registro

variables
    fichero de DATO_INDIVIDUAL FICHERO_DNI
```

Figura 1.9: Estructura del archivo DNI

habrá que asignar un archivo específico dentro de la memoria secundaria. Para asignar un archivo físico a la variable (*archivo lógico*) hay que llamar al siguiente procedimiento:

asignar(nombre_variable,nombre_fichero)

de la misma forma que se hacia en el tema anterior.

1.2.2 CREACIÓN Y APERTURA

Es exactamente igual al caso de los archivos de texto. Se usará la función:

abrir(nombre_variable) devuelve logica

que se invoca cuando se quiere abrir un archivo tipeado (que ha sido asignado a la variable *nombre_variable*) y devuelve un valor lógico, de forma que si ese valor es "verdadero", el archivo existía en el soporte físico y si el valor es "falso", el archivo no existía. La llamada a esta función produce un posicionamiento del dispositivo lectura/escritura en el soporte físico del archivo, si existe.

En caso de no existir el archivo asignado, se utiliza de nuevo un procedimiento que se encarga de *crearlo* y posteriormente de abrirlo:

crear(nombre_variable)

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

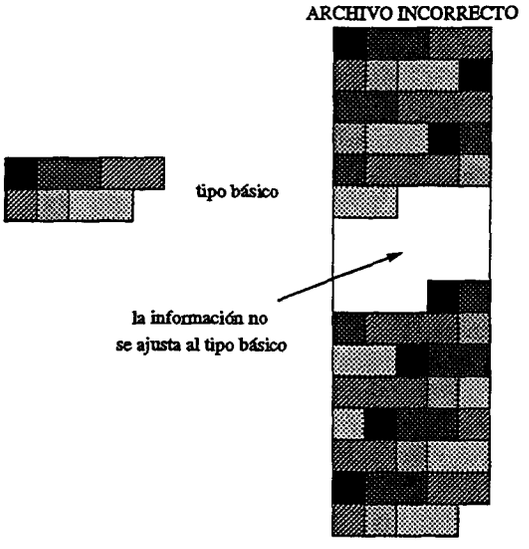


Figura 1.10: Archivo modificado con registros de otro tipo al básico

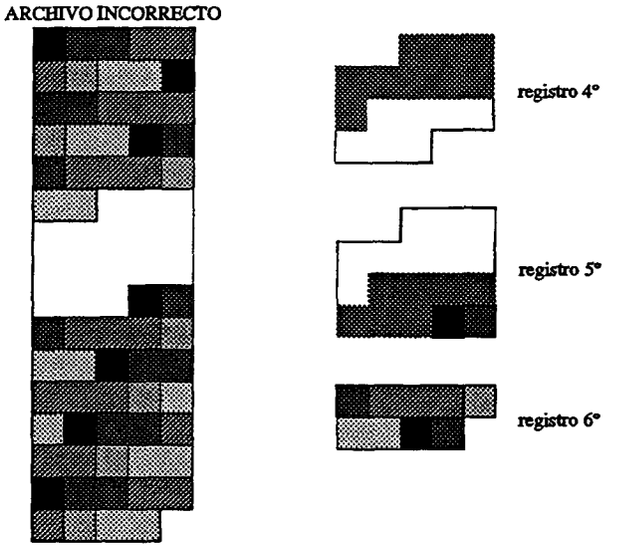


Figura 1.11: Acceso a los registros cuarto, quinto y sexto del archivo modificado

Igual que se comentó en el tema anterior, hay que tener cuidado a la hora de abrir un archivo utilizando el procedimiento *crear* ya que si existía se perderá la información que almacenaba. Tener en cuenta que en el caso de los archivos tipeados no existe la función *unir* que sí podíamos utilizar cuando trabajabamos con los archivos de texto. La no existencia de esta función no supone ningún problema ya que los archivos binarios permiten posicionar el apuntador en cualquier registro del archivo y, por tanto, si queremos situarnos al final del mismo simplemente haremos que el apuntador señale al último registro existente. En un apartado posterior explicaremos con más detalle este tema.

1.2.3 LECTURA Y ESCRITURA

Las operaciones más importantes que se pueden realizar sobre los archivos binarios son la lectura y escritura de un registro. Los procedimientos que se encargan de leer y escribir en un archivo tipeado son:

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

leer(nombre_fichero,nombre_variable)

escribir(nombre_fichero,nombre_variable)

donde *nombre_variable* es el nombre de una variable del mismo tipo que el tipo de la estructura básica del archivo y *nombre_fichero* es la variable del tipo archivo binario. Si se invoca al procedimiento *leer*, se leerá del archivo binario asignado a la variable *nombre_fichero* un registro. Si se invoca al procedimiento *escribir*, se escribirá en el archivo binario el contenido de la variable *nombre_variable*. Después de una lectura o escritura a un archivo tipeado, el apuntador señalará al siguiente registro dentro del archivo. Un error que normalmente se comete se produce al querer alterar un cierto registro del archivo. Para realizar esta operación, primero habrá que leer el archivo en cuestión, a continuación alterar el campo o los campos correspondientes dentro del registro leído y por último escribir el registro modificado dentro del archivo. En la figura 1.12 podemos ver representado esta situación utilizando un archivo formado por registros de tres campos del tipo cadena[25]. Supongamos que queremos leer el primer registro y cambiar el nombre por 'Luis José'. Inicialmente el apuntador marca al principio del archivo, pero una vez leído el primer registro, el apuntador marcará al segundo registro del archivo (figura 1.12(a)). A continuación se modifica el contenido del registro *dato* y por último se escribe en el archivo (figura 1.12(b)). Sin embargo, como la escritura en un archivo se hace siempre en la posición que indica el apuntador, en nuestro ejemplo el segundo registro, la información que había anteriormente en esa posición se pierde. Para evitar este problema, se vuelve a posicionar el apuntador de forma que señale al registro que se quiere sobreescribir, para lo cual se utiliza un procedimiento que veremos posteriormente.

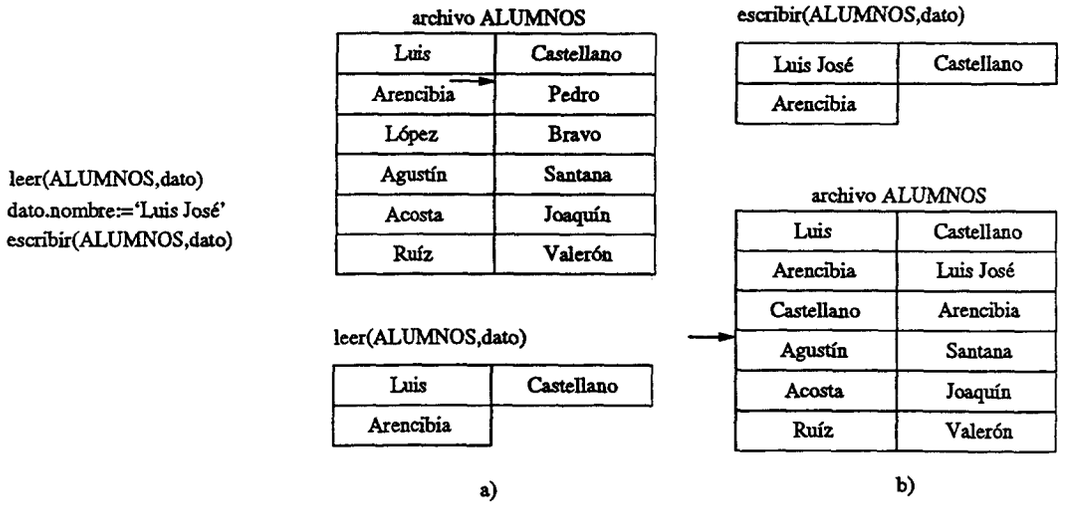


Figura 1.12: Ejemplo de modificarción de un archivo binario

Es importante tener en cuenta que la variable empleada a la hora de leer o escribir en un archivo directo tiene que ser del mismo tipo que el tipo de la estructura básica. Para entender este concepto supongamos que tenemos el archivo de la figura 1.13 con la información que se indica. Si quisieramos leer el archivo ALUMNOS utilizando como variable la siguiente:

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
tipo
   nuevo_registro=registro
      cadena[25]nombre,apellido1
   fin registro

variables
   nuevo_registro dato
```

tendríamos que la primera lectura sobre el archivo asignará a la variable *dato* los valores *dato.nombre* igual a 'Luis' y *dato.apellido1* igual a 'Castellano', ya que inicialmente el apuntador estará al comienzo del archivo. Al terminar la primera lectura, el apuntador señalará al campo que almacena la cadena *'Arencibia'* puesto que hemos leído una variable que es un registro de dos campos del tipo cadena de dimensión 25 (*nombre* y *apellido1*). Una segunda lectura sobre el archivo incurrirá en un error porque el valor de la variable *dato* será ahora *dato.nombre* igual a ' Arencibia' y *dato.apellido1* igual a 'Pedro'. Este error se irá arrastrando mientras se siga leyendo el archivo utilizando esta variable que es de un tipo diferente al tipo de la estructura básica del archivo. De igual forma, si quisieramos escribir en el archivo utilizando una variable de tipo diferente, seguiríamos cayendo en el mismo error. Por tanto, hay que tomar como norma inviolable que siempre que se quiera leer o escribir en archivos binarios hay que utilizar variables del mismo tipo que el tipo de la estructura básica del archivo.

archivo ALUMNOS

Luis	Castellano
Arencibia	Pedro
López	Bravo
Agustín	Santana
Acosta	Joaquín
Ruíz	Valerón

```
tipo
   registro DATOS_PERSONALES
      cadena[25] nombre,apellido1,apellido2
   fin registro

archivo de tipo DATOS_PERSONALES ALUMNOS
```

Figura 1.13: Ejemplo de archivo binario

1.2.4 CIERRE

Siempre que se realice la apertura de un archivo binario se debe realizar también su cierre. El procedimiento que se encarga de cerrar este tipo de archivos es igual al visto en el tema anterior, es decir:

```
cerrar(nombre_fichero)
```

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

Al igual que se explicó anteriormente, si se quiere volver a realizar alguna operación sobre el archivo habrá que abrirlo de nuevo. Se aconseja que la llamada a las operaciones de apertura y cierre las realice la misma subrutina.

1.2.5 OTRAS FUNCIONES

Además de las operaciones explicadas en los apartados anteriores, existen otras funciones que nos ayudan a trabajar con archivos binarios. Una de estas funciones es:

```
eof(nombre_fichero) devuelve logico
```

que nos indica si el apuntador está situado al final del archivo ("verdadero") o si no lo está ("falso"). Los archivos binarios al no estar formados por líneas no pueden utilizar una función que indique la posición del apuntador dentro de la línea, es decir, no pueden usar la función *eoln* que si utilizaban los archivos de texto.

1.2.5.1 ACCESO ALEATORIO A UN ARCHIVO

Como se ha explicado hasta ahora, los archivos binarios están formados por registros de igual tamaño, por lo que se puede acceder a cualquiera de ellos de forma directa. Para realizar esta operación se utiliza el procedimiento:

```
colocar(nombre_fichero,numero_registro)
```

donde *nombre_registro* es la variable asignada al archivo y *numero_registro* es el registro al cual se quiere apuntar. El primer registro del archivo será el número 0. Es importante mencionar que el procedimiento *colocar* sólo se puede usar con archivos binarios ya que sólo éstos son de acceso aleatorio.

En el ejemplo que vimos anteriormente figura 1.12, el error que se cometía cuando se escribía el registro modificado se puede solucionar colocando de nuevo el apuntador sobre el primer registro, como indica la figura 1.14.

1.2.5.2 TAMAÑO DE UN ARCHIVO BINARIO

En el ejemplo que vimos en el subapartado anterior se quería modificar el primer registro del archivo, sin embargo, supongamos que queremos modificar el último registro del archivo después de que a éste se le han añadido múltiples registros. Evidentemente, para saber cuál es el último registro tendríamos que saber qué dimensión o tamaño tiene el archivo para poder situar el apuntador correctamente. Una función en lenguaje algorítmico que nos indica este dato es:

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

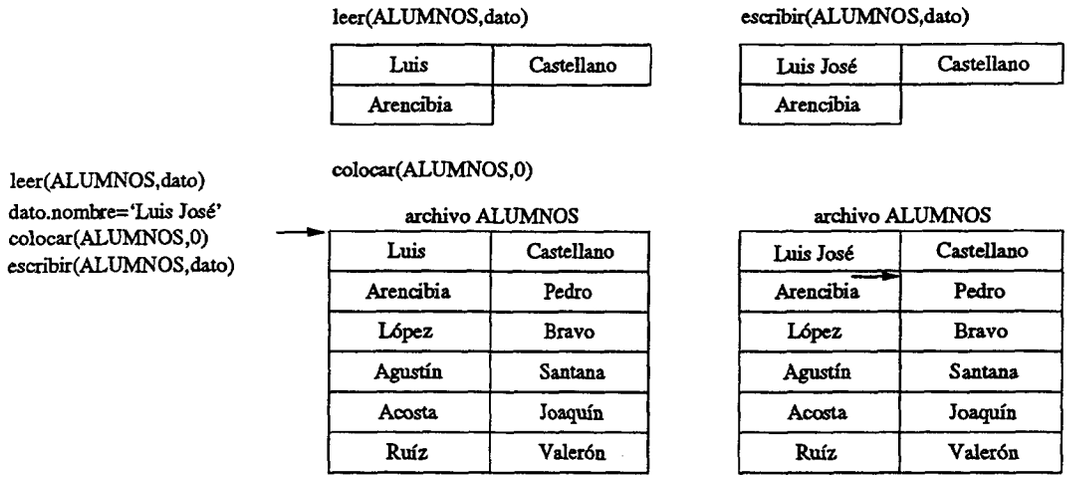


Figura 1.14: Ejemplo de modificación de un archivo binario

longitud_archivo(nombre_archivo) devuelve entero o entero largo

que nos devuelve un valor que nos indica la dimensión del archivo.

Si el primer registro de nuestro archivo es el 0 y queremos situar el apuntador en el último registro, la posición de ese registro será la *longitud_archivo(nombre_archivo)*-1.

1.2.5.3 POSICIONAMIENTO DEL APUNTADOR

Otra función interesante y que resulta de mucha ayuda es la que nos indica la posición del apuntador dentro del archivo. Muchas veces empezamos a trabajar con un archivo binario realizando operaciones de lectura, escritura y posicionamiento del apuntador, de forma que después de cierto tiempo no sabemos dónde está éste situado. Para resolver este problema se utiliza la función algorítmica siguiente:

puntero(nombre_archivo) devuelve entero o entero largo

donde la función devuelve la posición a dónde señala el apuntador dentro del archivo.

Recordar que la primera posición dentro del archivo es la posición 0 y la última la *longitud_archivo(nombre_archivo)*-1.

1.3 PROBLEMAS

1. Se dispone de un archivo de inventario "stock.tex" cuyos registros tienen los siguientes campos: nombre (cadena de 10 caracteres), código (entero), cantidad (entero), precio (entero), fabricante (cadena de 10 caracteres). Escribir un algoritmo que busque un determinado artículo por el número de código. Si el artículo existe, visualizar

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

nombre y código; en caso contrario hay que indicar con un mensaje lo sucedido.

```
Algoritmo inventario
constantes
   fichero='stock.tex'
tipos
   articulo=registro
      cadena[10] nombre,fabricante
      enteras codigo,cantidad,precio
   fin registro

Funcion encontrar(F,codigo,nombre) devuelve logico
   {archivo de articulo F por referencia}
   {enteras codigo por valor}
   {cadena[10] nombre por referencia}
variables
   articulo elemento
    logica salir
inicio
   salir:=falso
   mientras (no salir) y (no eof(F)) hacer
      leer(F,elemento)
      si elemento.codigo=codigo entonces
         nombre:=elemento.nombre
         salir:=verdadero
      fin si
   fin mientras
   encontrar:=salir
fin funcion

variables
   enteras codigo
   cadena[10] nombre
   archivo de articulo F
inicio
   asignar(F,fichero)
   leer(codigo)
   si abrir(F) entonces
      si encontrar(F,codigo,nombre) entonces
         escribir(' El producto es:',codigo,nombre)
      si no
         escribir(' El producto no existe')
      fin si
      cerrar(F)
   si no
      escribir(' Error al abrir archivo')
   fin si
fin
```

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

2. Se tiene un archivo binario llamado "informe.tex" en el que cada registro contiene los siguientes campos: nombre (cadena de 10 caracteres), apellidos (cadena de 20 caracteres), fecha (cadena con el siguiente formato dd-mm-aa), aviso (logico). En este archivo se guardan los abonados a un video club de forma que en el campo fecha se almacena la última fecha que cada individuo pagó la cuota (los abonados tienen únicamente facturas correspondientes al año en curso). Escribir una subrutina que busque todos los individuos que no han pagado un mes dado (entero introducido por teclado) y ponga su campo aviso a "verdadero".

```
Algoritmo control
constantes
   fichero='informe.tex'
tipos
   datos=registro
      cadena[10] nombre
      cadena[20] apellidos
      cadena[8] fecha
      logico aviso
   fin registro

Funcion comparar(elemento,mes) devuelve logico
   {datos elemento por valor}
   {enteras mes por valor}
variables
   enteras pos,num_mes,codigo
   cadena[2] cad_mes
inicio
   comparar:=falso
   pos:=posicion('-',elemento.fecha)
   si pos<>0 entonces
      cad_mes:=copiar(elemento.fecha,pos,2)
      Str_Num(cad_mes,num_mes,codigo)
      si (codigo=0) y (num_mes<mes) entonces
         comparar:=verdadero
      fin si
   fin si
fin funcion

Procedimiento marcar(F,mes)
   {archivo de articulo F por referencia}
   {enteras mes por valor}
variables
   datos elemento
inicio
   mientras no eof(F) hacer
      leer(F,elemento)
```

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
        si comparar(elemento,mes) entonces
          elemento.aviso:=verdadero
          colocar(F,puntero(F)-1)
          escribir(F,elemento)
        fin si
    fin mientras
fin procedimiento

variables
    enteras mes
    archivo de articulo F
inicio
    asignar(F,fichero)
    leer(mes)
    si abrir(F) entonces
        marcar(F,mes)
        cerrar(F)
    fin si
fin
```

1.4 BIBLIOGRAFÍA

[JOYANES90] L. JOYANES, *Problemas de metodología de la programación.* McGraw-Hill, 1990.

[JOYANES96] L. JOYANES, *Fundamentos de programación. Algoritmos y estructuras de datos.* Segunda edición. McGraw-Hill, 1996.

[WIRTH87] N. WIRTH, *Algoritmos y estructuras de datos.* Prentice Hall, 1987.

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

Capítulo 2

ESTRUCTURAS DINÁMICAS LINEALES

2.1 INTRODUCCIÓN

En este capítulo se introducen las estructuras dinámicas lineales como una alternativa a los tipos de datos estáticos. Si recordamos de capítulos anteriores, las variables con estructura de datos estáticas, una vez declaradas podían alterar su contenido, pero no su tamaño o cardinalidad. Por ejemplo, pensemos en una variable de tipo vector de enteros de dimensión 10. Una vez declarada esta variable, en ella se podrán almacenar elementos enteros que pueden variar a lo largo del desarrollo del programa, sin embargo, la forma de dicha variable no puede ser alterada (sólo habrá espacio para 10 elementos enteros). Sin embargo, muchas veces se necesitan estructuras de información o datos más complicadas que las estudiadas hasta el momento. Estas estructuras se caracterizan porque no sólo pueden variar su contenido durante la ejecución del programa sino que además pueden variar su tamaño. Debido a ésto, las variables que son de este tipo se denominan *estructuras de datos dinámicas.* En este capítulo veremos las estructuras dinámicas más simples y en capítulos posteriores estudiaremos estructuras más complejas pero que siguen la misma filosofía que las que analizaremos aquí.

2.2 OPERACIONES BÁSICAS CON PUNTEROS

Al igual que los archivos vistos en el capítulo anterior, las estructuras dinámicas estarán formadas por un número indeterminado de componentes estáticos (tipo básico), es decir, el número de elementos del tipo básico que existirá a lo largo del desarrollo del programa no se sabrá antes de la ejecución del mismo, por tanto, las estructuras dinámicas se dice que tienen *cardinalidad* infinita. Como consecuencia de lo explicado, es imposible asignar un espacio definido y fijo de memoria para almacenar las variables dinámicas declaradas, surgiendo así el concepto de *asignación dinámica de memoria.* De esta forma, cada vez que se necesite un nuevo componente de la estructura, se deberá asignar de forma dinámica,

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

es decir, el sistema operativo se encargará de asignar un hueco no ocupado de memoria para ese elemento. Una primera impresión que se puede sacar de ésto es que las distintas componentes de una estructura dinámica no tienen porque ocupar posiciones consecutivas de memoria debido a que el sistema operativo puede asignar huecos situados en posiciones arbitrarias.

Una pregunta que el lector puede hacerse es ¿cómo saber qué huecos de memoria pertenecen a una misma estructura que se ha declarado previamente? Si observamos la figura 2.1 vemos que no existe un patrón que nos diga qué componentes pertenecen a qué estructura, por tanto, cuando el usuario necesite acceder a un determinado componente no podrá hacerlo porque no sabrá donde está (en la figura hemos escrito la estructura a la que pertenece sólo para aclarar el concepto).

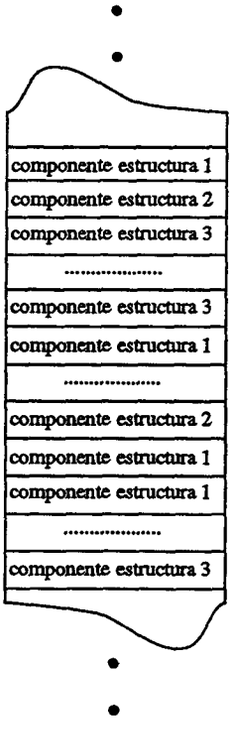


Figura 2.1: Asignación dinámica de memoria

La solución a este problema se consigue enlazando los distintos componentes de la estructura dinámica de manera que desde unos se pueda acceder a los otros (figura 2.2), por tanto, cada elemento de la estructura debe almacenar la dirección de una posición de memoria a la cual puede acceder. Ésta es la definición de *puntero.* Un puntero es una variable especial que almacena una dirección de memoria que corresponde a otra variable, es decir, un puntero "apunta" a otra variable dentro de la memoria. En la figura 2.3, la variable situada en la posición 1000 de memoria apunta a la variable situada en la posición 1004, luego la primera variable es un puntero. Al definir una variable de tipo puntero se debe indicar el tipo de elementos que se va a almacenar en las posiciones apuntadas. Ésto se debe, como ya sabemos, a que los diferentes tipos de datos requieren diferentes cantidades de memoria para guardar sus valores. Si una variable va a almacenar un puntero, debe ser declarada de una forma especial:

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

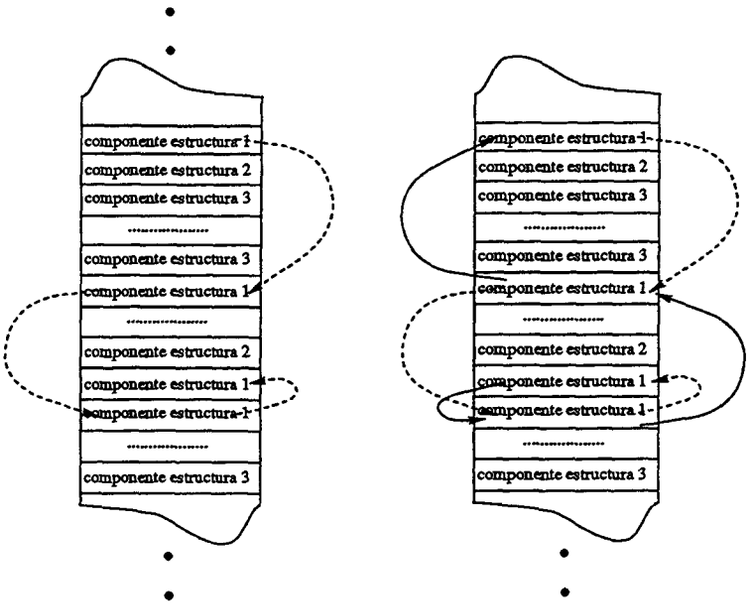


Figura 2.2: a) Enlace simple entre los componentes de la estructura. b) Enlace doble entre los componentes de la estructura

```
tipos
   tipo_puntero=puntero a tipo_datos
variables
   tipo_puntero var1
```

donde *var1* es una variable de tipo *tipo_puntero* que apunta hacia otra variable cuyo tipo es *tipo_dato* (cualquier tipo de dato estático). La frase *puntero a* significa "apunta hacia". La figura 2.4 nos muestra un ejemplo de punteros donde cada elemento de la estructura dinámica almacena un registro y apunta hacia el siguinte elemento de la estructura. A partir de ahora llamaremos *nodo* a cada elemento de la estructura dinámica y emplearemos la forma de representación que se ha utilizado en la figura 2.4 (tener en cuenta que cada nodo ocupa un espacio de la memoria del sistema y ese espacio tiene un tamaño que depende del tipo de los datos que almacena).

Llegados a este punto hay una serie de preguntas que pueden surgir: ¿cómo se crea una variable dinámica?, ¿cómo se puede acceder al primer elemento de la estructura? y ¿ a qué nodo apunta el último elemento? Para contestar a la primera pregunta vamos a utilizar un nuevo procedimiento que se encarga de asignar un hueco de memoria y hacer que una variable puntero apunte a él. Este procedimiento tiene el siguiente formato:

```
asignar_memoria(p)
```

donde p es un puntero que apunta hacia una posición de memoria asignada. Evidentemente, el tamaño de la memoria no es infinito y por tanto puede ocurrir que, en un momento dado, no haya suficiente espacio de memoria libre, produciéndose un error en tiempo de ejecución del programa. Muchas veces estos errores se pueden evitar si una vez que se termina de utilizar una estructura, los nodos que la formaban son "devueltos" a la memoria

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

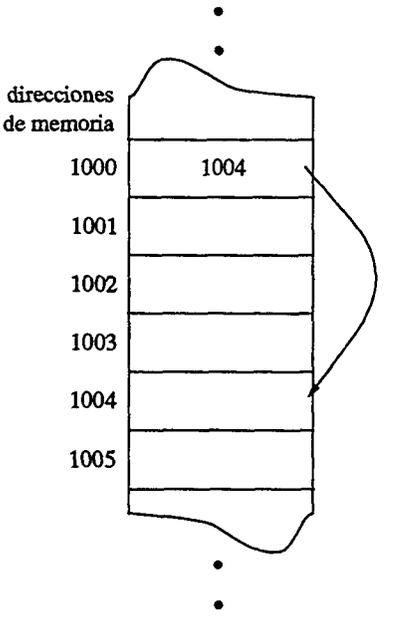


Figura 2.3: Variable puntero

(volviendo a quedar como espacio libre para su posterior utilización). Por tanto, es fundamental la existencia de otro procedimiento que libera la posición de memoria ocupada por una variable dinámica:

liberar_memoria(p)

Como norma podemos decir que por cada llamada al procedimiento *asignar_memoria* debe haber, en algún punto de nuestro programa, una llamada al procedimiento *liberar_memoria*. A continuación presentamos una posible subrutina que crea la estructura dinámica de la figura 2.4:

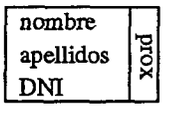
```
tipos
   tipo_puntero1=puntero a datos
   registro datos
      cadena[15] nombre, apellidos
      enteras DNI
      tipo_puntero1 prox
   fin registro

Procedimiento Crea_estructura()
variables
   tipo_puntero1 persona1,persona2
   enteras i
inicio
   asignar_memoria(persona1)
   leer(datos.nombre(persona1))
   leer(datos.apellidos(persona1))
   leer(datos.DNI(persona1))
```

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
  desde i:=1 hasta 3 hacer
    asignar_memoria(persona2)
    leer(datos.nombre(persona2))
    leer(datos.apellidos(persona2))
    leer(datos.DNI(persona2))
    datos.prox(persona1):=persona2
    persona1:=persona2
  fin desde
fin procedimiento
```

En la figura 2.5 vemos una pequeña traza de las tres iteraciones del algoritmo. Tener en cuenta que antes de usar cualquier variable hay que inicializarla con algún valor. En este caso, cuando se lee por teclado el nombre, apellidos y DNI de una persona se almacena esta información en un nodo de la estructura previamente asignado. Además de la inicialización, en el algoritmo podemos ver dos ejemplos de asignación de variables: *datos.prox(persona1):=persona2* y *puntero1:=persona2* donde todos son punteros del tipo *tipo_puntero1*. Si observamos la última asignación, habrá más de un puntero apuntando a una misma posición de memoria (ver figura 2.5).

```
tipos
  tipo_puntero1=puntero a datos
  registro datos
    cadena de caracteres[15] nombre,apellidos
    entero DNI
    tipo_puntero1 prox
  fin registro

variables
  tipo_puntero1 persona
```



variable puntero

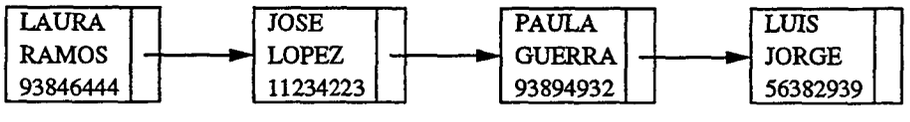


Figura 2.4: Ejemplo de punteros

Aunque ya sabemos asignar y liberar memoria todavía falta la respuesta a las dos preguntas formuladas anteriormente. Para contestarlas vamos a fijarnos en el algoritmo y en la figura 2.5. Inicialmente la variable que almacena el primer nodo de la estructura es la variable *persona1*, sin embargo, el contenido de esa variable se pierde al hacer una posterior asignación. Por tanto, si queremos conservar la dirección en memoria del primer nodo hay que utilizar una variable adicional que llamaremos *cabeza* y que siempre apuntará al primer elemento de la estructura. De esta forma, al algoritmo se le debe añadir una nueva instrucción, antes del bucle *desde*, que guarda este valor (*cabeza:=persona1*). Tener en cuenta que si el valor de la cabeza se pierde no se podrá acceder posteriormente al resto de la estructura (necesitamos saber dónde está el primer elemento para poder acceder al resto).

La respuesta a la tercera pregunta la vamos a dar introduciendo el concepto de la

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

constante *nulo*. La constante nulo se utiliza para dar un valor a una variable puntero que apunta a ninguna posición. Si hacemos la siguiente asignación: *p:=nulo*, es ilegal hacer una referencia al valor almacenado en *p* ya que *p* no apunta a ninguna parte. Éste es un error que se comete con bastante frecuencia y que se evita si antes de querer leer la información que almacena *p*, preguntamos si esta variable apunta a nulo. Si nos fijamos nuevamente en la figura 2.5 vemos que el último puntero no apunta a otro nodo, por tanto, se hace que apunte a nulo. En el algoritmo se debe añadir una nueva instrucción, después de la estructura *desde*, que haga lo explicado (*datos.prox(persona2):=nulo*). En la figura 2.6 vemos como queda la estructura representando nulo con el símbolo /.

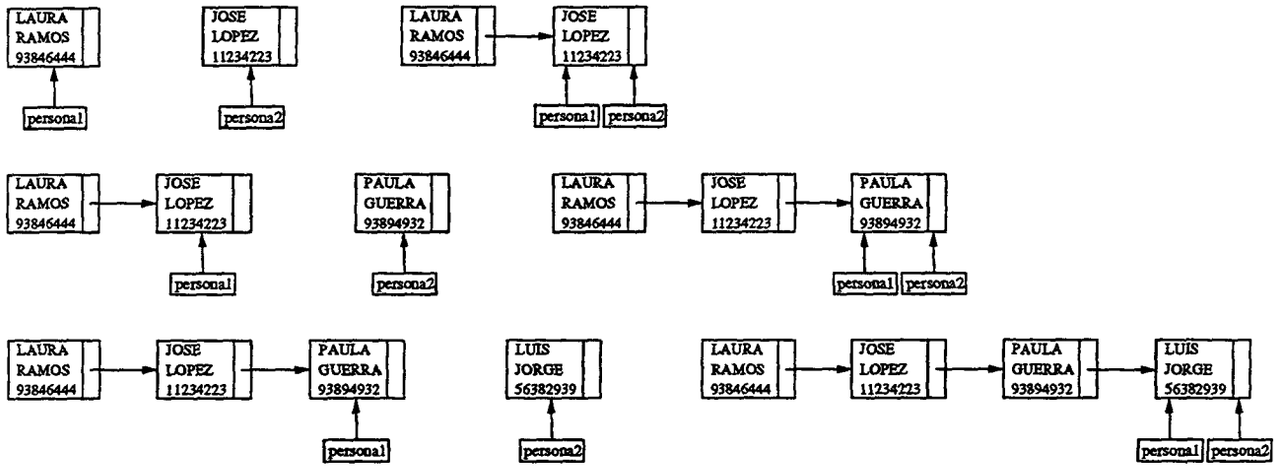


Figura 2.5: Traza de asignación de punteros

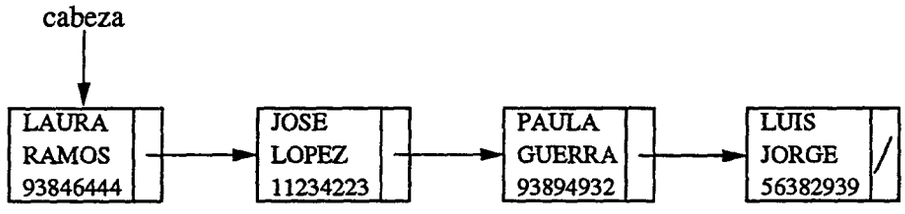


Figura 2.6: Estructura con el puntero *cabeza* y la constante *nulo*

Una vez vistos los operadores básicos con punteros vamos a ver una serie de estructuras dinámicas lineales como son: las listas simplemente enlazadas, las listas circulares, las listas doblemente enlazada, las pilas y las colas.

2.3 LISTAS SIMPLEMENTE ENLAZADAS

Las listas enlazadas son una secuencia de elementos del mismo tipo (nodos) unidos unos a otros mediante un enlace o puntero. Cada nodo de la lista tiene un predecesor y un sucesor, excepto el primer elemento de la lista que no tiene predecesor y el último elemento que no tiene sucesor. En la figura 2.6 podemos ver una lista simplemente enlazada.

Como se explicó anteriormente, cada nodo de la lista está formado por un campo de información (que puede ser cualquier estructura estática) y por un campo puntero, que lo enlaza con su sucesor.

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

2.3.1 REPRESENTACIÓN DE UNA LISTA ENLAZADA EN MEMORIA

Una de las partes en las que se divide la memoria de los computadores y que introducimos aquí debido a la relación que tiene con los punteros, es la *zona de almacenamiento dinámico (heap)*. El almacenamiento dinámico guarda las variables dinámicas declaradas en nuestro programa. Estas variables pueden crearse y eliminarse del *heap* por lo que éste puede crecer o decrecer. El tamaño máximo que puede tener el almacenamiento dinámico depende del tamaño de la RAM del computador. El *heap* lo gestionan subrutinas que se encargan de asignar memoria (procedimiento *asignar_memoria*) y de liberar memoria (procedimiento *liberar_memoria*). Como ya se explicó, puede suceder que en algún momento en la ejecución de nuestro programa, el *heap* no tenga memoria suficiente para asignar cuando se le pida. En este caso, se producirá un error por *fallo de memoria*. Para evitar este problema (que el programa "aborte" en tiempo de ejecución) se pueden utilizar la función:

memoria_disponible() devuelve entero largo

que dice cuánta memoria hay libre en el almacenamiento dinámico. Esta función debe ser usada antes de solicitar espacio de memoria para alguna variable dinámica.

2.3.2 CREACIÓN E INICIALIZACIÓN DE LISTAS ENLAZADAS

Antes de empezar a manipular una lista enlazada hay que crear una variable dinámica con estructura de lista. Cada nodo de la lista contiene como mínimo un campo valor y un campo puntero, por tanto y como ya se vio previamente, la forma más simple de representarlo es mediante un registro. En lenguaje algorítmico, podemos crear una lista enlazada de la siguiente manera:

```
tipos
   tipo_valor=...
   ptr=puntero a datos
   registro datos
      tipo_valor info
      ptr prox
   fin registro

variables
   ptr cabeza
```

donde *cabeza* debe ser inicializada a *nulo* ya que la lista inicialmente esta vacía.

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

2.3.3 RECORRIDO DE UNA LISTA ENLAZADA

Recorrer una lista enlazada consiste en desplazarse desde el primer nodo hasta el último para manipular su información (presentarla en pantalla, escribirla en un archivo, aplicarle alguna transformación ...). La siguiente subrutina se encarga de recorrer una lista enlazada para presentar sus datos por pantalla:

```
tipos
   ptr=puntero a datos
   registro datos
      cadena[15] nombre,apellidos
      enteras DNI
      ptr prox
   fin registro
variables
   ptr cabeza

Procedimiento Recorre_estructura(cabeza)
   {ptr cabeza por valor}
variables
   ptr nodo
inicio
   nodo:=cabeza
   mientras nodo<>nulo hacer
      escribir(datos.nombre(nodos),datos.apellidos(nodo),datos.DNI(nodo))
      nodo:=datos.prox(nodo)
   fin mientras
fin procedimiento
```

donde el puntero *cabeza* ha sido inicializado previamente, bien con el valor nulo o con la dirección de un espacio de memoria que contiene datos.

2.3.4 BÚSQUEDA DE UN ELEMENTO EN UNA LISTA ENLAZADA

Para buscar un elemento en una lista enlazada hay que recorrerla hasta el final o hasta que el elemento buscado se encuentre. Una subrutina que realiza la búsqueda de un DNI en la lista enlazada del apartado anterior viene a continuación:

```
Funcion Busca_elemento(cabeza,DNI) devuelve logico
   {ptr cabeza por valor}
   {enteras DNI por valor}
variables
   ptr nodo
inicio
   nodo:=cabeza
```

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
   mientras (nodo<>nulo) y (datos.DNI(nodo)<>DNI) hacer
      nodo:=datos.prox(nodo)
   fin mientras
   si nodo<>nulo entonces
      Busca_elemento:=verdadero
   si no
      Busca_elemento:=falso
   fin si
fin funcion
```

Una pregunta interesante y que dejamos al lector es la siguiente: ¿por qué no se puede cambiar el orden de las dos expresiones en la estructura repetitiva **mientras** (datos.DNI(nodo) <> DNI) y (nodo<>nulo)?

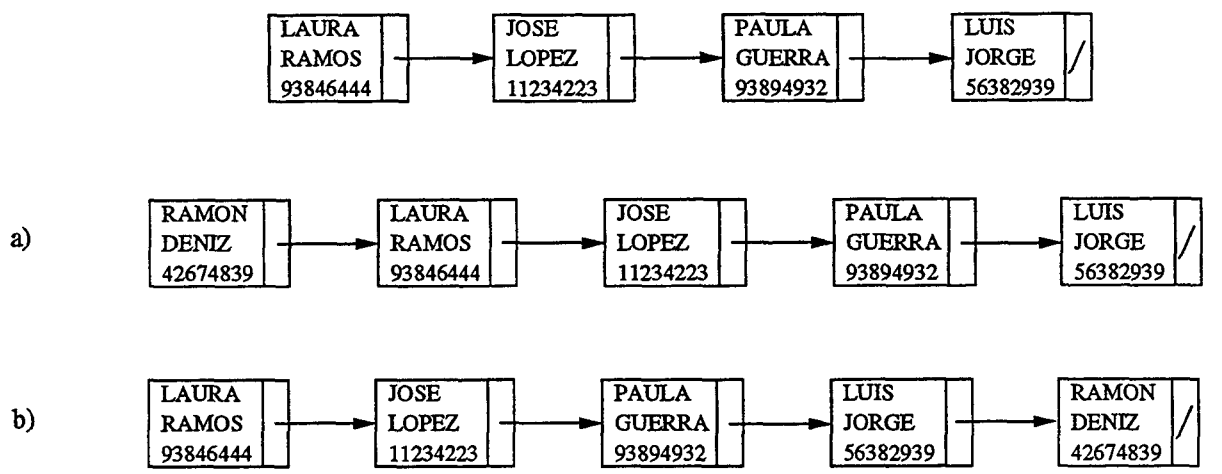


Figura 2.7: Inserción en una lista enlazada. a) Por el frente. b) Por la cola

2.3.5 INSERCIÓN DE UN ELEMENTO EN UNA LISTA ENLAZADA

Para insertar en una lista enlazada hay distintas formas de hacerlo: inserción por delante (por el frente), inserción por el medio e inserción por detrás (por la cola). En la figura 2.7 vemos la inserción por el frente y por la cola. A continuación mostraremos las subrutinas que realizan las inserciones en esas posiciones, considerándose en cualquiera de los dos casos que hay espacio suficiente en el almacenamiento dinámico.

```
Procedimiento Insercion_delante(cabeza,nombre,apellidos,DNI)
   {ptr cabeza por referencia}
   {cadena[15] nombre,apellidos por valor}
   {enteras DNI por valor}
variables
   ptr nodo
inicio
   asignar_memoria(nodo)
   datos.nombre(nodo):=nombre
   datos.apellidos(nodo):=apellidos
```

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
   datos.DNI(nodo):=DNI
   datos.prox(nodo):=cabeza
   cabeza:=nodo
fin procedimiento

Procedimiento Insercion_detras(cabeza,nombre,apellidos,DNI)
   {ptr cabeza por referencia}
   {cadena[15] nombre,apellidos por valor}
   {enteras DNI por valor}
variables
   ptr nodo,q
inicio
   asignar_memoria(nodo)
   datos.nombre(nodo):=nombre
   datos.apellidos(nodo):=apellidos
   datos.DNI(nodo):=DNI
   q:=cabeza
   mientras datos.prox(q)<>nulo hacer
      q:=datos.prox(q)
   fin mientras
   datos.prox(q):=nodo
   datos.prox(nodo):=nulo
fin procedimiento
```

2.3.6 ELIMINACIÓN DE UN ELEMENTO DE UNA LISTA ENLAZADA

Al borrar un elemento de una lista enlazada hay que enlazar correctamente el puntero del sucesor para no perder la información en el resto de la lista. En la figura 2.8 presentamos el caso en el que al borrar un nodo concreto, la lista queda dividida en dos porciones inconexas. A continuación mostramos una subrutina que borra un nodo de la lista del apartado anterior.

```
Procedimiento Borrar_nodo(cabeza,nodo)
   {ptr cabeza por referencia}
   {ptr nodo por valor}
variables
   ptr q
inicio
   si nodo=cabeza entonces
      cabeza:=datos.prox(nodo)
   si no
      q:=cabeza
      mientras (datos.prox(q)<>nulo) y (datos.prox(q)<>nodo) hacer
         q:=datos.prox(q)
      fin mientras
```

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
      si datos.prox(q)=nodo entonces
         datos.prox(q):=datos.prox(nodo)
      fin si
   fin si
   liberar_memoria(nodo)
fin procedimiento
```

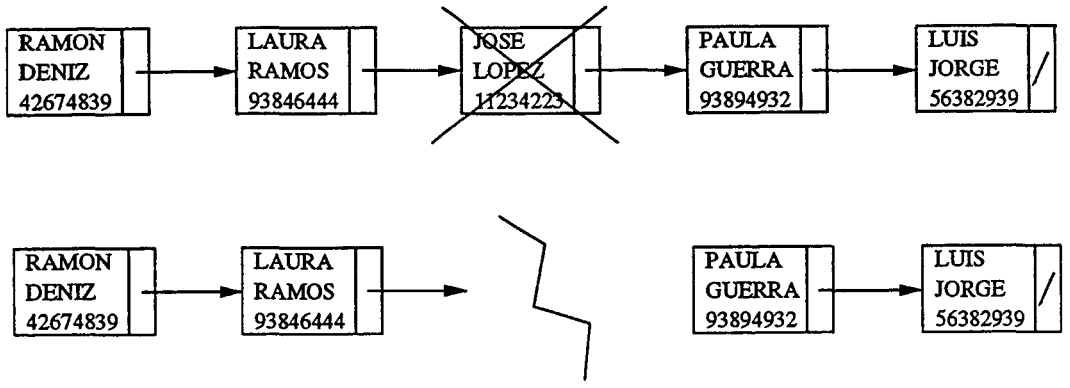


Figura 2.8: Eliminación de un nodo en una lista enlazada

2.4 LISTAS CIRCULARES

Las listas circulares son listas enlazadas que se caracterizan porque el último nodo tiene como sucesor al primero (obviamente, el primero tiene como predecesor al último). En la figura 2.9 hay un ejemplo de este tipo de listas. Fijarse que en una lista circular se pierde el concepto de qué nodo es el primero y cuál el último, sin embargo, como se explicó en el apartado anterior, es necesario tener un puntero para poder acceder a la lista, por tanto, en este tipo de estructuras dinámicas también existe un puntero *cabeza*.

2.4.1 INSERCIÓN EN UNA LISTA CIRCULAR

A continuación se presenta un algoritmo que inserta un nodo *nuevo* en una lista circular después de otro nodo llamado *anterior*. La función devuelve verdadero si se pudo insertar y falso en caso contrario.

```
Funcion Insertar_circular(nuevo,anterior) devuelve logico
   {ptr nuevo,anterior por referencia}
inicio
   si (anterior<>nulo) y (anterior<>nuevo) entonces
      datos.prox(nuevo):=datos.prox(anterior)
      datos.prox(anterior):=nuevo
      Insertar_circular:=verdadero
   si no
      Insertar_circular:=falso
   fin si
fin funcion
```

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

Suponemos que hay espacio suficiente en el almacenamiento dinámico, por lo que no se comprueba.

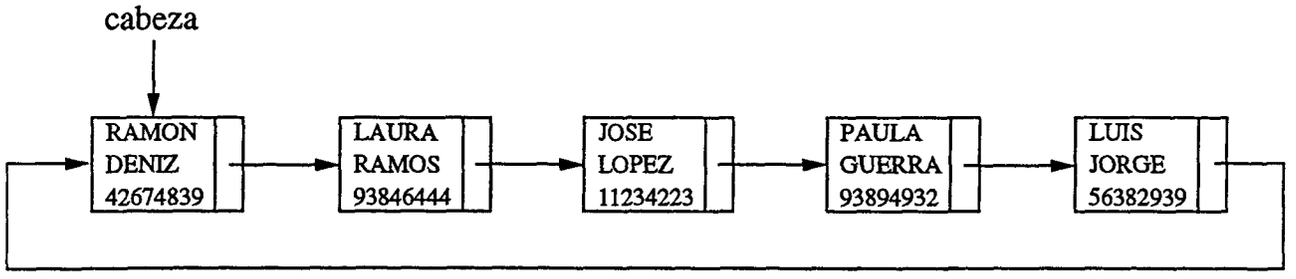


Figura 2.9: Lista enlazada circularmente

2.4.2 ELIMINACIÓN DE UN NODO DE UNA LISTA CIRCULAR

A continuación se presenta un algoritmo que elimina un nodo *antiguo* de una lista circular. El nodo a eliminar es el sucesor de un nodo llamado *anterior*. La función devuelve verdadero si se pudo eliminar y falso en caso contrario.

```
Funcion Eliminar_circular(cabeza,antiguo,anterior) devuelve logico
   {ptr cabeza,antiguo,anterior por referencia}
inicio
   si (anterior<>nulo) y (anterior<>antiguo) entonces
      si cabeza=antiguo entonces
         cabeza:=datos.prox(cabeza)
      fin si
      datos.prox(anterior):=datos.prox(antiguo)
      liberar_memoria(antiguo)
      Eliminar_circular:=verdadero
   si no
      Eliminar_circular:=falso
   fin si
fin funcion
```

2.5 LISTAS DOBLEMENTE ENLAZADAS

Una lista doblemente enlazada es aquella en la que cada nodo tiene acceso directo (enlace) con su predecesor y su sucesor. Las únicas excepciones son: el primer nodo de la lista, que sólo tiene acceso a su sucesor, y el último nodo de la estructura, que tiene acceso únicamente a su predecesor. Teniendo en cuenta que el primer nodo de la lista y el último tienen un puntero que no apuntan a otro nodo, se les da el valor constante *nulo*. En la figura 2.10 se muestra un ejemplo de este tipo de listas.

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

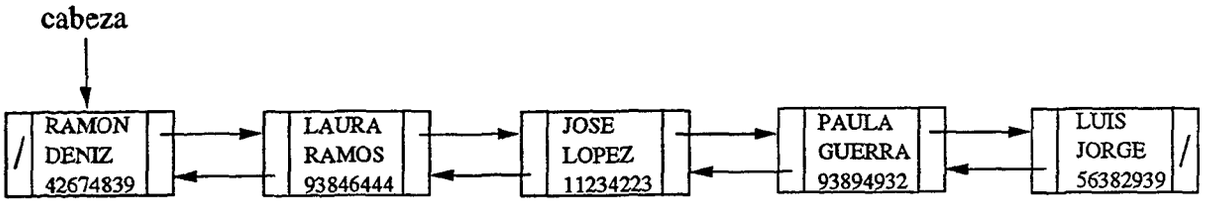


Figura 2.10: Lista doblemente enlazada

2.5.1 CREACIÓN DE UNA LISTA DOBLEMENTE ENLAZADA

Cada nodo de una lista doblemente enlazada contiene como mínimo un campo valor y dos campos puntero, por tanto, la forma más simple de representarlo es mediante un registro. En lenguaje algorítmico, podemos crear una lista doblemente enlazada de la siguente forma:

```
tipos
   tipo_valor=...
   ptr=puntero a datos
   registro datos
      tipo_valor info
      ptr prox
      ptr ant
   fin registro

variables
   ptr cabeza
```

donde *cabeza* debe ser inicializada a *nulo* ya que la lista inicialmente esta vacía.

2.5.2 INSERCIÓN EN UNA LISTA DOBLEMENTE ENLAZADA

En las líneas siguientes mostramos una subrutina que inserta un nodo *nuevo* en una lista doblemente enlazada después de un nodo llamado *anterior*. La función devuelve verdadero si se pudo insertar y falso en caso contrario.

```
Funcion Insertar_doble1(nuevo,anterior) devuelve logico
   {ptr nuevo,anterior por referencia}
inicio
   si (anterior<>nulo) y (anterior<>nuevo) entonces
      datos.prox(nuevo):=datos.prox(anterior)
      si datos.prox(anterior)<>nulo entonces
         datos.ant(datos.prox(anterior)):=nuevo
      fin si
      datos.ant(nuevo):=anterior
```

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
      datos.prox(anterior):=nuevo
      Insertar_doble1:=verdadero
   si no
      Insertar_doble1:=falso
   fin si
fin funcion
```

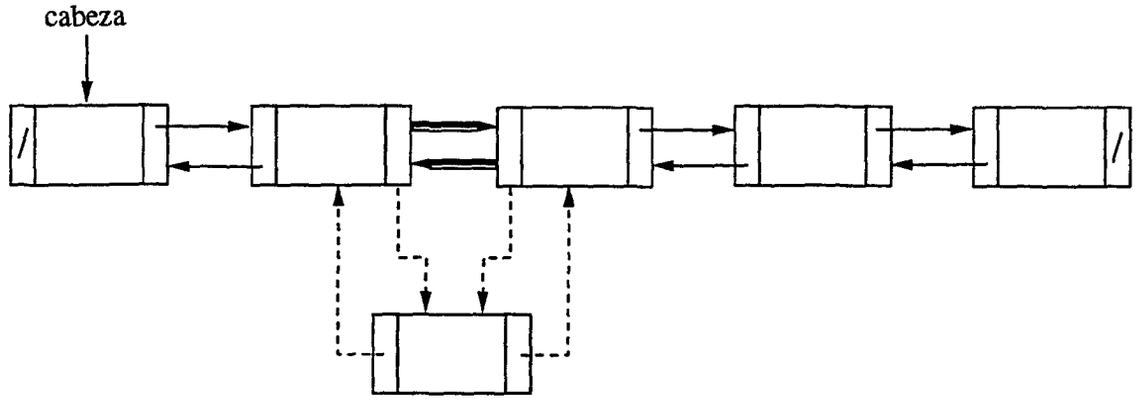


Figura 2.11: Inserción en una lista doblemente enlazada

En esta subrutina aparece una doble indirección: *datos.ant(datos.prox(anterior))* que se lee como el puntero *ant* del elemento sucesor del nodo *anterior*. El siguiente algoritmo realiza también una inserción en una lista doblemente enlazada pero antes de un nodo llamado *siguiente*. La función devuelve verdadero si se pudo insertar y falso en caso contrario.

```
Funcion Insertar_doble2(cabeza,nuevo,siguiente) devuelve logico
   {ptr cabeza,nuevo,siguiente por referencia}
inicio
   si (siguiente<>nulo) y (siguiente<>nuevo) entonces
      datos.ant(nuevo):=datos.ant(siguiente)
      si datos.ant(siguiente)<>nulo entonces
         datos.prox(datos.ant(siguiente):=nuevo
      si no
         cabeza:=nuevo
      fin si
      datos.prox(nuevo):=siguiente
      datos.ant(siguiente):=nuevo
      Insertar_doble2:=verdadero
   si no
      Insertar_doble2:=falso
   fin si
fin funcion
```

En esta subrutina hay que pasar el puntero *cabeza* como parámetro de entrada/salida porque el puntero *siguiente* puede ser igual a la cabeza. En este caso, cuando se inserte el nuevo nodo, éste pasará a ser la cabeza de la lista. Suponemos que hay espacio suficiente en el almacenamiento dinámico, por lo que no se comprueba.

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

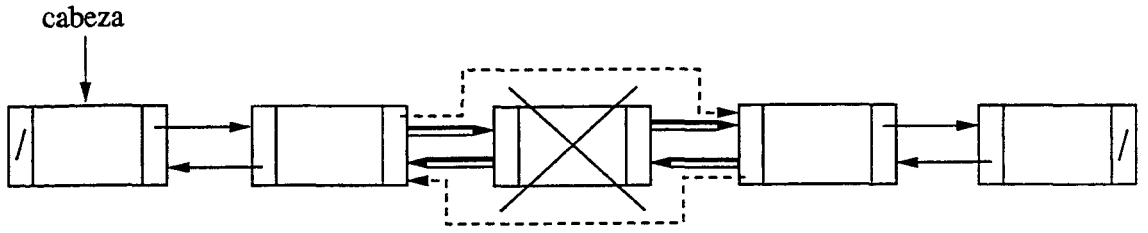


Figura 2.12: Eliminación en una lista doblemente enlazada

2.5.3 ELIMINACIÓN DE UNA LISTA DOBLEMENTE ENLAZADA

A continuación se presenta un algoritmo que elimina un nodo *antiguo* de una lista doblemente enlazada. El nodo a eliminar es el sucesor de un nodo llamado *anterior*. La función devuelve verdadero si se pudo eliminar y falso en caso contrario.

```
Funcion Eliminar_doble1(antiguo,anterior) devuelve logico
   {ptr antiguo,anterior por referencia}
inicio
   si (anterior<>nulo) y (anterior<>antiguo) entonces
      datos.prox(anterior):=datos.prox(antiguo)
      si datos.prox(antiguo)<>nulo entonces
         datos.ant(datos.prox(antiguo)):=anterior
      fin si
      liberar_memoria(antiguo)
      Eliminar_doble1:=verdadero
   si no
      Eliminar_doble1:=falso
   fin si
fin funcion
```

El siguiente algoritmo realiza también la eliminación de un nodo en una lista doblemente enlazada pero situado antes de un nodo llamado *siguiente*. La función devuelve verdadero si se pudo insertar y falso en caso contrario.

```
Funcion Eliminar_doble2(cabeza,antiguo,siguiente) devuelve logico
   {ptr cabeza,antiguo,siguiente por referencia}
inicio
   si (siguiente<>nulo) y (siguiente<>antiguo) entonces
      datos.ant(siguiente):=datos.ant(antiguo)
      si datos.ant(antiguo)<>nulo entonces
         datos.prox(datos.ant(antiguo)):=siguiente
      si no
         cabeza:=siguiente
      fin si
      liberar_memoria(antiguo)
      Eliminar_doble2:=verdadero
```

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
    si no
      Eliminar_doble2:=falso
    fin si
fin funcion
```

En esta subrutina hay que pasar el puntero *cabeza* como parámetro de entrada/salida porque el puntero *siguiente* puede ser igual a la cabeza. En este caso, cuando se elimine el nodo *antiguo*, el nodo *siguiente* pasará a ser la cabeza de la lista.

Dentro del grupo de listas doblemente enlazadas podemos encontrar las listas circulares doblemente enlazadas. En la figura 2.13 podemos encontrar un ejemplo de este tipo de estructuras (al final de este capítulo hay un problema dedicado a estas listas). En las listas circulares doblemente enlazadas, todos los nodos tienen un sucesor y un predecesor y al igual que en las listas circulares simplemente enlazadas debe existir un puntero (*cabeza*) para acceder a los nodos de la lista.

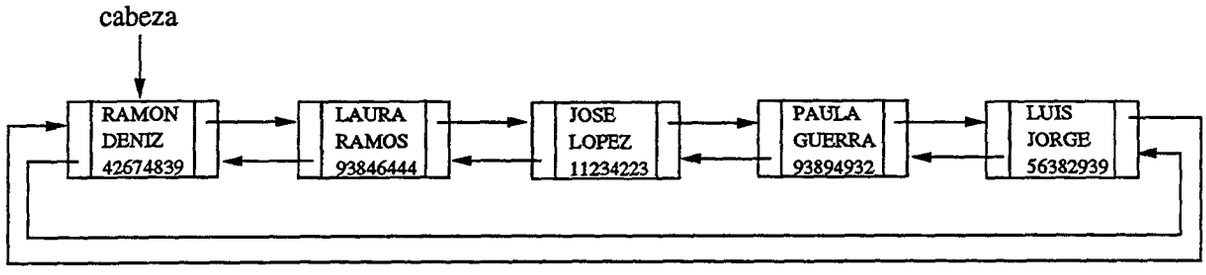


Figura 2.13: Lista doblemente enlazada circularmente

2.6 PILAS

Las listas enlazadas que hemos visto hasta ahora permitían la inserción y eliminación de nodos en cualquier punto de la misma (ver ejemplos de apartados anteriores), sin embargo, existen listas enlazadas especiales llamadas *pilas* que se caracterizan porque la inserción y la extracción de elementos se realizan por un único punto llamado *cima* de la pila. El concepto de pila en un computador es similar al concepto de pila en el mundo real (en la figura 2.14 podemos ver ejemplos de pilas en el mundo real). Como se observa en la figura, los elementos de la pila real sólo pueden añadirse por encima y extraerse por este mismo punto. Debido a que el último elemento que entra en la pila tiene que ser el siguiente (o primero) en salir, a las pilas también se les llama estructuras *LIFO* (*last in-first out* o *último en entrar-primero en salir*). En la figura 2.15 se muestra el ejemplo que hemos usado para las listas enlazadas en los apartados anteriores, utilizando estructura de pila.

Las operaciones básicas que se pueden realizar sobre las pilas son: creación, determinar si la pila está vacía, introducir un elemento en la pila y extraer un elemento de la pila. Al igual que con las listas enlazadas, el tamaño que puede tener una pila sólo está limitado por la memoria del computador, por tanto, antes de introducir elementos en ella hay que ver si existen huecos libres en el almacenamiento dinámico.

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

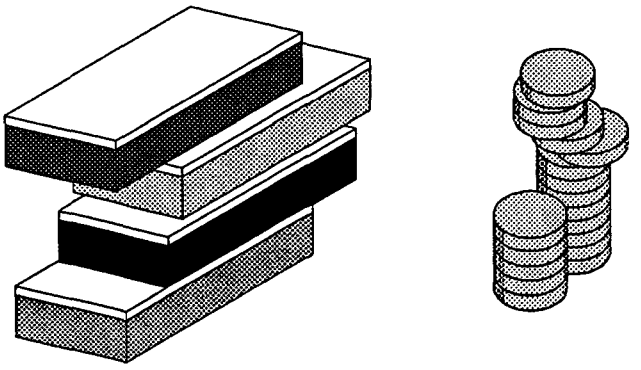


Figura 2.14: Ejemplos de pilas en la vida real

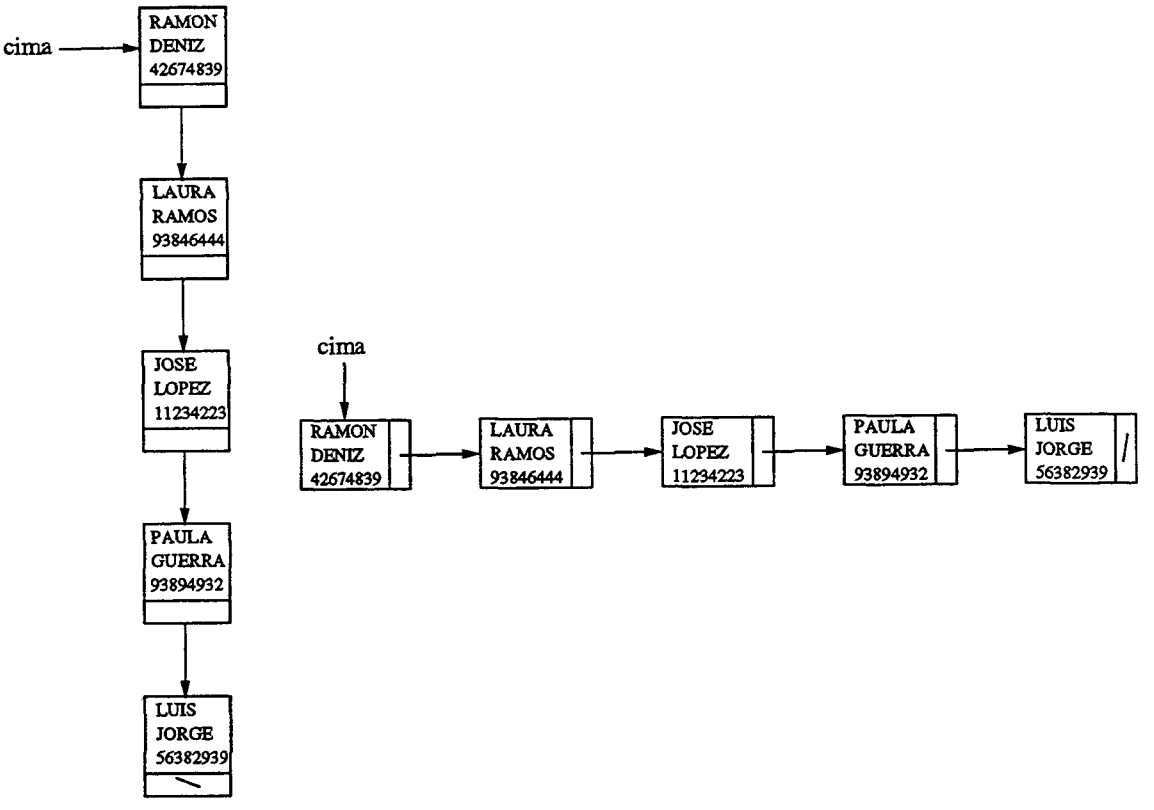


Figura 2.15: Ejemplo de pila

2.6.1 CREACIÓN DE UNA PILA

Antes de trabajar sobre una pila (insertar y extraer elementos) hay que crear una variable dinámica que la soporte. En lenguaje algorítmico, ésto se puede hacer de la siguiente forma:

```
tipos
   tipo_valor=...
   ptr=puntero a datos
   registro datos
      tipo_valor info
      ptr ant
   fin registro
```

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
variables
   ptr cima
```

donde *cima* debe ser inicializada a *nulo* ya que la pila inicialmente esta vacía. Fijarse que la forma de declarar una variable pila es exactamente igual a la forma de declarar una lista simplemente enlazada (realmente una pila es una lista simplemente enlazada especial).

2.6.2 PILA VACÍA

Para comprobar si la pila está vacía basta con comprobar si el valor de la cima es nulo. Si ésto ocurre, en la pila no se ha introducido ningún dato. Antes de sacar elementos de la pila hay que comprobar que hayan elementos, es decir, que la pila no esté vacía. A continuación se muestra la subrutina *Pila_vacia.*

```
Funcion Pila_vacia(cima) devuelve logico
   {ptr cima por valor}
inicio
   si cima=nulo entonces
      Pila_vacia:=verdadero
   si no
      Pila_vacia:=falso
   fin si
fin funcion
```

2.6.3 INSERCIÓN EN UNA PILA

La inserción en una pila debe hacerse siempre por la cima. La siguiente subrutina muestra la inserción de un nuevo dato en la pila:

```
Procedimiento Insertar_pila(cima,dato)
   {ptr cima por referencia}
   {tipo_valor dato por valor}
variables
   ptr nodo
inicio
   asignar_memoria(nodo)
   datos.info(nodo):=dato
   datos.ant(nodo):=cima
   cima:=nodo
fin procedimiento
```

Suponemos que hay espacio suficiente en el almacenamiento dinámico para insertar en la pila, por lo que no se comprueba.

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

2.6.4 EXTRACCIÓN EN UNA PILA

La extracción en una pila debe hacerse siempre por la cima. Es muy importante tener en cuenta que antes de extraer algún elemento de la pila hay que comprobar que la pila tenga elementos. La siguiente subrutina muestra la extracción de un dato de la pila.

```
Procedimiento Eliminar_pila(cima,dato)
   {ptr cima por referencia}
   {tipo_valor dato por referencia}
variables
   ptr nodo
inicio
   nodo:=cima
   dato:=datos.info(nodo)
   cima:=datos.ant(nodo)
   liberar_memoria(nodo)
fin procedimiento
```

2.7 COLAS

Al igual que las pilas, las colas son listas enlazadas especiales que se caracterizan porque los elementos se añaden por un extremo llamado *final* de la cola y se extraen por el extremo contrario llamado *frente* o *cabeza* de la cola. Las colas se asemejan a las colas de la vida real (en la figura 2.16 podemos ver una cola en el mundo real). Por su particular funcionamiento, las colas se llaman también estructuras *FIFO* (*first in-first out* o lo que es lo mismo, *primero en entrar-primero en salir*). En la figura 2.17 se muestra una cola utilizando el mismo ejemplo del apartado anterior.

Las operaciones básicas que se pueden realizar sobre las colas son: creación, determinar si la cola está vacía, introducir un elemento en la cola y extraer un elemento de la cola. Al igual que con las listas enlazadas, el tamaño que puede tener una cola sólo está limitado por la memoria del computador, por tanto antes de introducir elementos en ella hay que ver si existen huecos libres en el almacenamiento dinámico.

2.7.1 CREACIÓN DE UNA COLA

Antes de trabajar sobre una cola (insertar y extraer elementos) hay que crear una variable dinámica que la soporte. En lenguaje algorítmico, ésto se puede hacer de la siguiente forma:

```
tipos
   tipo_valor=...
   ptr=puntero a datos
```

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
registro datos
   tipo_valor info
   ptr prox
fin registro

variables
   ptr frente,final
```

donde *frente* y *final* deben ser inicializadas a *nulo* ya que la cola inicialmente esta vacía. Fijarse que la forma de declarar una variable cola es exactamente igual a la forma de declarar una lista simplemente enlazada salvo que existirá un puntero para el primer elemento y otro para el último elemento.

Figura 2.16: Ejemplo de cola en la vida real

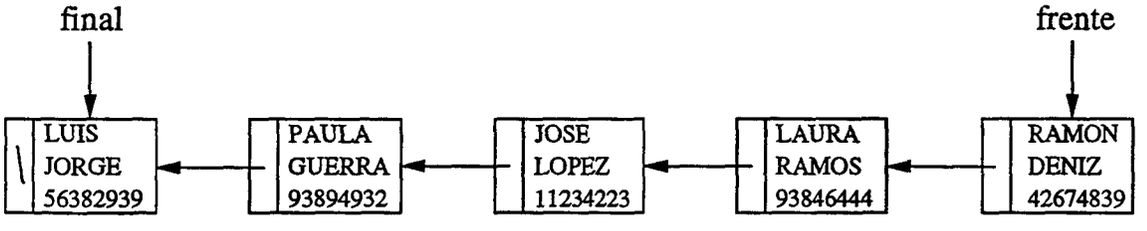


Figura 2.17: Ejemplo de cola

2.7.2 COLA VACÍA

Para comprobar si la cola está vacía basta con comprobar si el valor del frente es nulo. Si ésto ocurre, en la cola no se ha introducido ningún dato. Antes de sacar elementos de la cola hay que comprobar que hayan elementos, es decir, que la cola no esté vacía. A continuación se muestra la subrutina *Cola_vacia.*

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
Funcion Cola_vacia(frente) devuelve logico
   {ptr frente por valor}
inicio
   si frente=nulo entonces
      Cola_vacia:=verdadero
   si no
      Cola_vacia:=falso
   fin si
fin funcion
```

2.7.3 INSERCIÓN EN UNA COLA

La inserción en una cola debe hacerse siempre por el final. La siguiente subrutina muestra la inserción de un nuevo dato en la cola, suponiendo que hay espacio suficiente en memoria.

```
Procedimiento Insertar_cola(frente,final,dato)
   {ptr frente,final por referencia}
   {tipo_valor dato por valor}
variables
   ptr nodo
inicio
   asignar_memoria(nodo)
   datos.info(nodo):=dato
   datos.prox(nodo):=nulo
   si final<>nulo entonces
      datos.prox(final):=nodo
   fin si
   final:=nodo
   si frente=nulo entonces
      frente:=final
   fin si
fin procedimiento
```

2.7.4 EXTRACCIÓN DE UNA COLA

La extracción en una cola debe hacerse siempre por el frente. Es muy importante tener en cuenta que antes de extraer algún elemento de la cola hay que comprobar que la cola tenga elementos. La siguiente subrutina muestra la extracción de un dato de la cola.

```
Procedimiento Eliminar_cola(frente,final,dato)
   {ptr frente,final por referencia}
   {tipo_valor dato por referencia}
variables
   ptr nodo
inicio
   dato:=datos.info(frente)
```

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
   nodo:=frente
   frente:=datos.prox(frente)
   si frente=nulo entonces
      final:=frente
   fin si
   liberar_memoria(nodo)
fin procedimiento
```

2.8 RECURSIVIDAD

Una subrutina se dice que es recursiva si se puede expresar en función de sí misma, es decir, en el cuerpo de la subrutina aparece alguna mención suya. La recursividad es un mecanismo utilizado cuando hay que repetir unos ciertos pasos (instrucciones), siendo el número de repetición variable. Por ejemplo, el factorial de un número natural se puede definir como una subrutina (función) recursiva. Si recordamos, el factorial de un número natural se puede expresar como:

$$\begin{array}{ll} n! = 1 & (\text{si } n = 0) \\ n \times (n-1) \times (n-2) \times ... \times 1 & (\text{si } n > 0) \end{array}$$

Cuando $n > 0$, hay que hacer un número determinado de multiplicaciones, donde el número de veces que hay que hacer el mismo proceso depende de n. La expresión anterior también se puede expresar de la siguiente forma:

$$n! = n \times (n-1) \times (n-2) \times ... \times 1 = n \times (n-1)!$$

Ahora la definición de factorial decimos que es recursiva ya que hay un factorial dentro del cálculo de un factorial, es decir, para calcular el factorial de n hay que calcular el factorial de $n-1$, para calcular el factorial de $n-1$ hay que calcular el factorial de $n-2$, y así sucesivamente.

Para diseñar algoritmos recursivos hay que seguir una serie de pasos. En primer lugar hay que identificar el proceso repetitivo que se va a realizar (en el ejemplo anterior serán las multiplicaciones) y en segundo lugar hay que considerar que una función recursiva tiene que finalizar su ejecución alguna vez. Para que ésto ocurra se deben cumplir las dos condiciones siguientes:

- Debe haber un valor en los parámetros de la subrutina que se considere caso elemental. Ésto significa que, cuando los parámetros tomen estos valores, la subrutina finalizará su ejecución devolviendo el control al punto siguiente a donde fue llamada (en el ejemplo del factorial, la condición elemental será que el parámetro n tuviera valor 0, puesto que, en ese caso el factorial se sabe que vale 1. Tener en cuenta que si $n > 0$, el factorial de n se calcula en base al factorial de $n-1, n-2, n-3...$ porque éste no se conoce, pero si $n = 0$, el valor del factorial sí es conocido, por tanto, no hay que calcular ningún otro factorial).

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

- Las llamadas recursivas deben afectar a parámetros con valores, en cada llamada, "más pequeños" que los iniciales, es decir, el valor de los parámetros debe disminuir en cada llamada recursiva, de esta forma, cada llamada se va acercando más al caso elemental (para el ejemplo del factorial en cada paso el valor del factorial a calcular se va haciendo más pequeño. En la primera llamada será n, en la siguiente $n-1$, luego $n-2$ y así hasta 0).

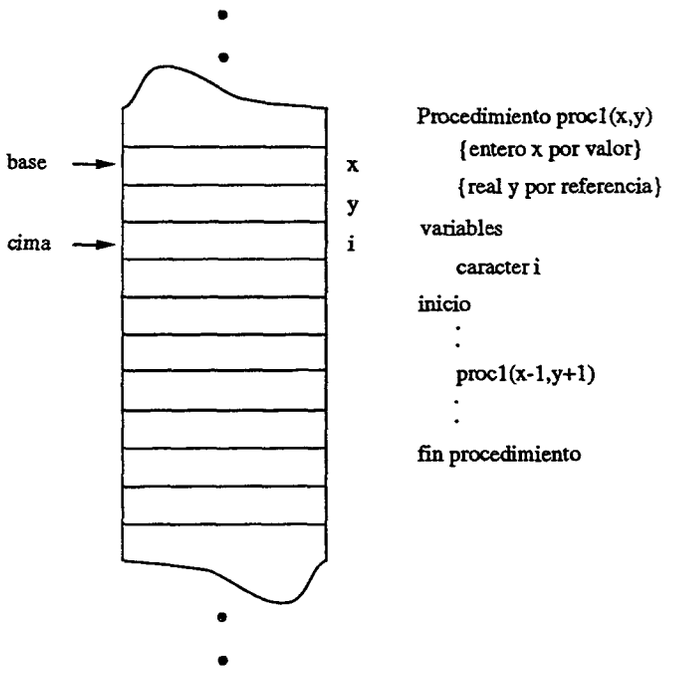


Figura 2.18: Parámetros y variables locales de una subrutina en memoria

A continuación se muestra la función factorial que calcula el factorial de un número natural.

```
Funcion factorial(n) devuelve entero largo
  {enteras n por valor}
inicio
  si n=0 entonces
    factorial:=1
  si no
    factorial:=n*factorial(n-1)
  fin si
fin funcion
```

Si observamos en el algoritmo, el uso en la cláusula *si no* de *factorial* tiene dos significados diferentes. Por un lado, se utiliza como nombre de la función recibiendo un valor (lado izquierdo de la asignación) que es la forma en la que estamos acostumbrados a verlo y por otro, se utiliza como llamada recursiva a la función *factorial* con el parámetro $n-1$ (lado derecho de la asignación).

Después de analizar las condiciones que deben cumplir las subrutinas recursivas y visto un sencillo ejemplo, vamos a exponer los puntos fundamentales que se deben seguir a la hora de diseñar un algoritmo recursivo:

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

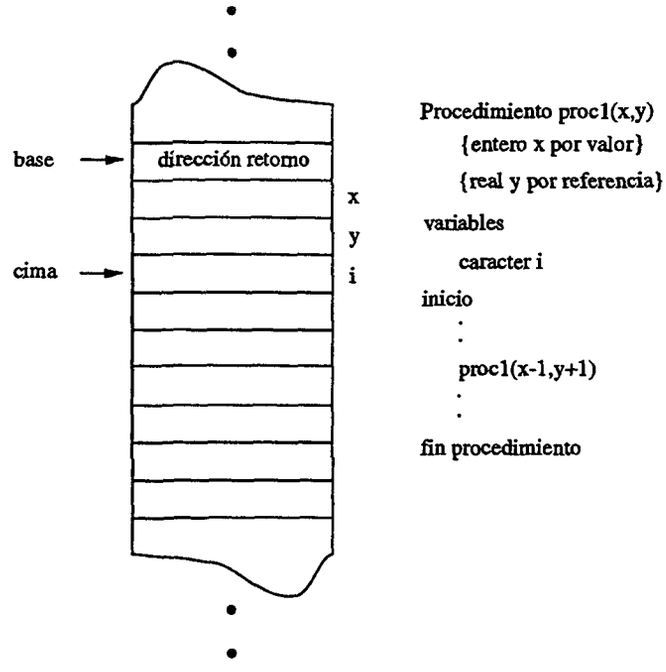


Figura 2.19: Parámetros, variables locales y dirección de retorno de una subrutina en memoria

- Obtener una definición exacta del problema a resolver
- Determinar el tamaño del problema (depende de los valores iniciales de los parámetros del problema)
- Determinar el caso elemental y resolverlo. Éste es el único punto del algoritmo que no se resuelve recursivamente
- Determinar y resolver el caso general como descomposición en partes más pequeñas del mismo problema

2.8.1 IMPLEMENTACIÓN DE LA RECURSIVIDAD MEDIANTE PILAS

Si recordamos, las subrutinas (procedimientos o funciones) estaban formadas por un cuerpo principal (proceso), por unas variables locales (declaradas dentro de la subrutina) y por unos parámetros que le sirven para comunicarse con el exterior. Cada parámetro formal y cada variable local tiene asignada una posición de memoria relativo a una posición (dirección *base*), donde almacenan el valor que tienen en un momento dado (en la figura 2.18 podemos ver un ejemplo de subprograma y asignación en memoria de sus parámetros y variables). Como se puede observar en la figura, tanto los parámetros como las variables ocupan posiciones contiguas en memoria a partir de una dirección determinada. Para acceder al valor de cualquiera de ellos, basta con acceder hacia atrás desde la posición *cima*. Como vemos, la estructura es típicamente la de una pila. De esta forma, el parámetro x estaría en la posición $cima - 2$, el parámetro y estaría en $cima - 1$ y la variable i en la posición *cima*. Sin embargo, además de estos valores, de una subrutina hay que almacenar

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

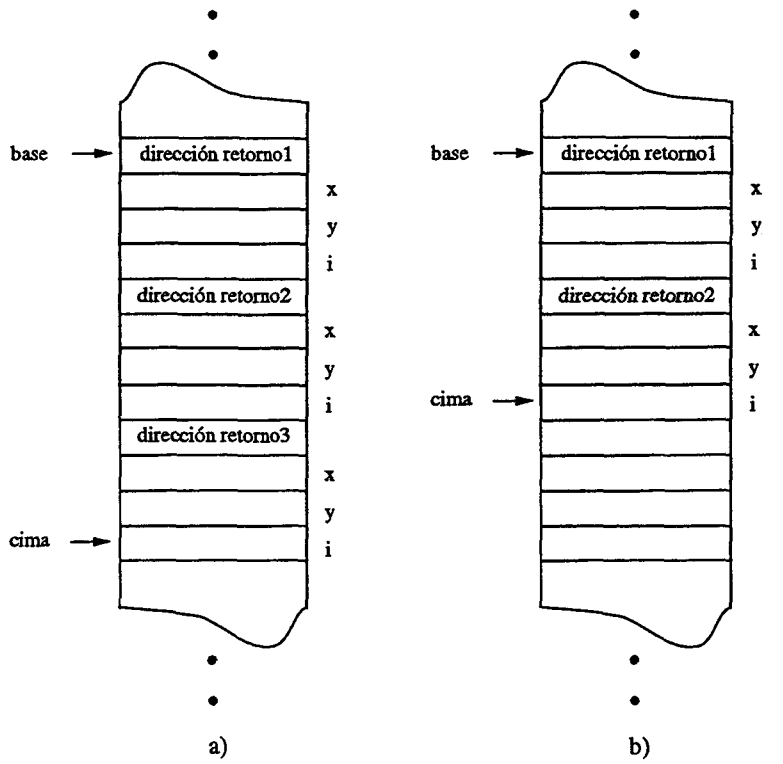


Figura 2.20: a) Llamadas recursivas de un procedimiento. b) Retorno de una llamada recursiva

algo más. Si recordamos, cuando una subrutina finaliza su ejecución se debe retornar a la instrucción que sigue a la llamada dentro del programa *llamador* (figura 2.21). Para saber cuál es el punto al que hay que retornar, se tiene que almacenar su dirección en algún lugar de la memoria que no puede alterarse (figura 2.19). Como vemos en la figura, no sólo se asigna espacio de memoria a los parámetros y variables locales de una subrutina, sino que además se reserva espacio para guardar la dirección de retorno (en la posición $cima - 3$).

Cada vez que se llama a un procedimiento o función, los parámetros ocupan posiciones de memoria, y *cima* se pone apuntando a la dirección del último parámetro o variable local. De esta forma, si se vuelve a hacer una llamada a la misma subrutina, el espacio requerido para almacenar sus parámetros y variables locales, estará ubicado a continuación de la posición de *cima* (en la figura 2.20(a) se muestran tres llamadas al procedimiento

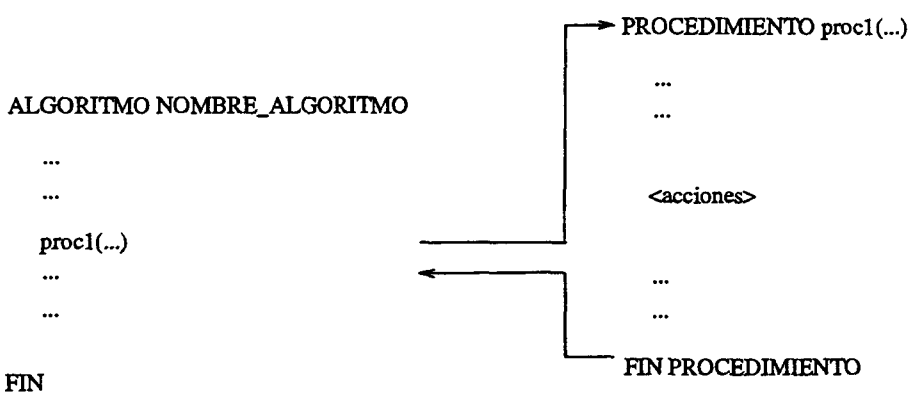


Figura 2.21: Llamada y retorno de un procedimiento

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

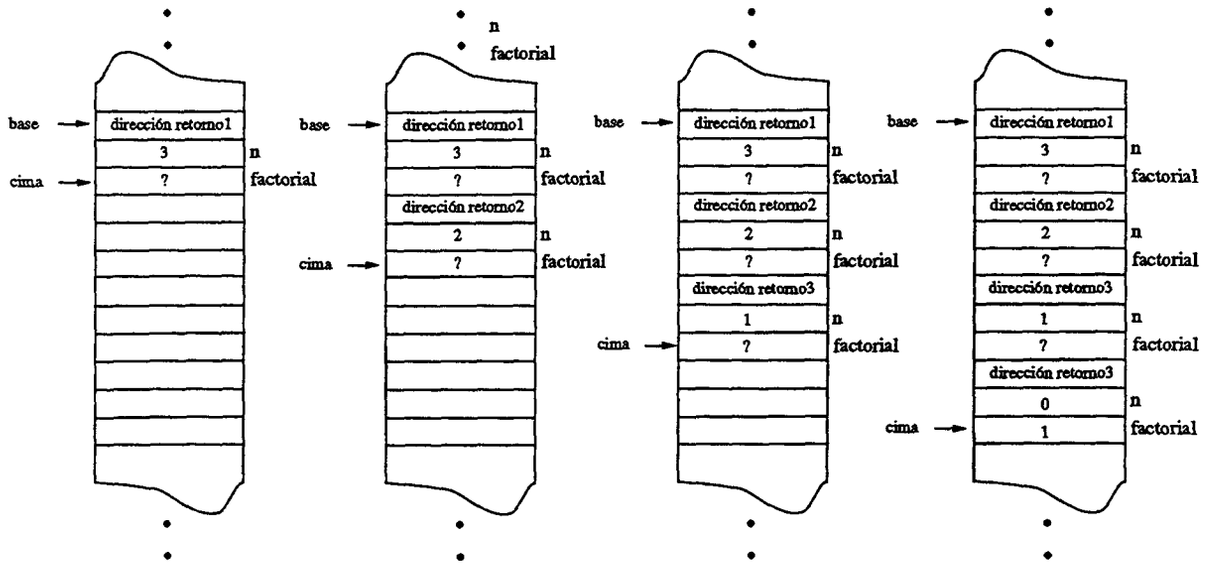


Figura 2.22: Llamadas recursivas de la función *factorial*

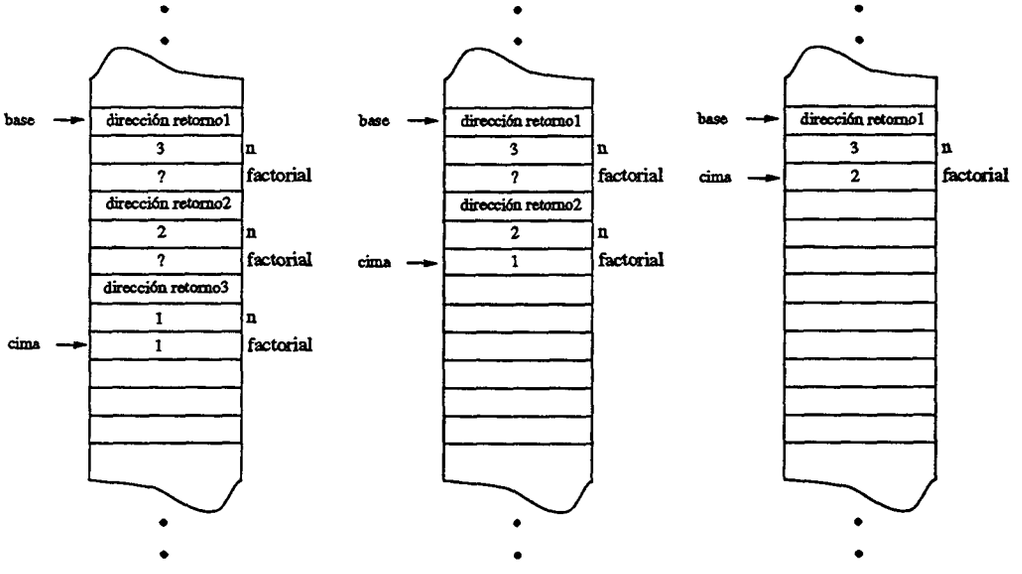


Figura 2.23: Retorno de llamadas recursivas de la función *factorial*

anterior). Una vez que finaliza la ejecución del procedimiento o función, las posiciones ocupadas por sus parámetros, variables y dirección de retorno, son devueltas para ser utilizadas posteriormente (en la figura 2.20(b) se muestra el contenido de la memoria después del retorno de la tercera llamada). Como se ve, el funcionamiento de la recursividad corresponde con el funcionamiento de una pila. Las variables asociadas a la subrutina se guardan en la pila al empezar a ejecutar la subrutina y son extraídas de la pila cuando el procedimiento o función terminan su ejecución.

En las figuras 2.22 y 2.23 se muestra el comportamiento de la pila en la ejecución del factorial para $n = 3$. La primera llamada a la función *factorial* devuelve a la subrutina que la llamó, el valor 6 ($n \times factorial$ -ver figura 2.23). En los próximos capítulos se estudiará más a fondo el tema de la recursividad aplicado a estructuras de datos más complejas.

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

2.9 PROBLEMAS

1. Dada una lista enlazada (de tipo *ptr1*), cuyo campo de información contiene el nombre, apellidos (cadenas de caracteres de tamaño 15) y DNI (entero) de una persona, ordenada de menor a mayor por el subcampo apellidos se pide:

   a) Realizar una subrutina que dado un nombre, apellidos y DNI, inserte ordenadamente un nuevo nodo.

```
Procedimiento Inser_orden(cabeza,n,a,DNI)
   {ptr1 cabeza por referencia}
   {cadena[15] n,a por valor}
   {enteras DNI por valor}
variables
   ptr1 p,q
inicio
   asignar_memoria(p)
   datos.nom(p):=n
   datos.apell(p):=a
   datos.DNI(p):=DNI
   si (cabeza=nulo) o (datos.apell(cabeza)>a) entonces
      datos.prox(p):=cabeza
      cabeza:=p
   si no
      q:=datos.prox(cabeza)
      mientras (datos.prox(q)<>nulo) y (datos.apell(datos.prox(q))<a) hacer
         q:=datos.prox(q)
      fin mientras
      datos.prox(p):=datos.prox(q)
      datos.prox(q):=p
   fin si
fin procedimiento
```

   b) Realizar una subrutina que dado un nombre, apellidos y DNI, borre de la lista el nodo que lo contenga si éste existe.

```
Procedimiento Elim_orden(cabeza,n,a,DNI)
   {ptr1 cabeza por referencia}
   {cadena[15] n,a por valor}
   {enteras DNI por valor}
variables
   ptr1 p,q
   logica encontrado
inicio
   q:=cabeza
   p:=cabeza
   encontrado:=falso
```

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
   mientras (q<>nulo) y (no encontrado) hacer
      si (datos.nom(q)=n) y (datos.apell(q)=a) y (datos.DNI(q)=DNI) entonces
         encontrado:=verdadero
      si no
         p:=q
         q:=datos.prox(q)
      fin si
   fin mientras
   si encontrado entonces
      si q=cabeza entonces
         cabeza:=datos.prox(cabeza)
      si no
         datos.prox(p):=datos.prox(q)
      fin si
      liberar_memoria(q)
   fin si
fin procedimiento
```

2. Dada una lista circular doblemente enlazada (de tipo *ptr2*), cuyo campo de información contiene el DNI y el grupo de prácticas de un alumno (carácter), ordenada de menor a mayor por el subcampo DNI, a partir del nodo cabeza. Se pide realizar una subrutina que dado un DNI y un grupo de prácticas, inserte ordenadamente un nuevo nodo.

```
Procedimiento Inser_orden_cir(cabeza,grupo,DNI)
   {ptr2 cabeza por referencia}
   {caracter grupo por valor}
   {enteras DNI por valor}
variables
   ptr2 p,q
inicio
   asignar_memoria(p)
   datos.grupo(p):=grupo
   datos.DNI(p):=DNI
   si cabeza=nulo entonces
      datos.prox(p):=cabeza
      datos.ant(p):=cabeza
      cabeza:=p
   si no
      si datos.DNI(cabeza)>DNI entonces
         q:=cabeza
         cabeza:=p
      si no
         q:=datos.prox(cabeza)
         mientras (q<>cabeza) y (datos.DNI(q)<datos.DNI(p)) hacer
            q:=datos.prox(q)
         fin mientras
      fin si
```

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
      datos.prox(datos.ant(q)):=p
      datos.ant(p):=datos.ant(q)
      datos.ant(q):=p
      datos.prox(p):=q
   fin si
fin procedimiento
```

3. Realizar una subrutina que recorra una pila (de tipo *ptr3*), cuyo campo de información está formado por el nombre y apellidos de una persona, y lo presente por pantalla. Hacer los mismo con una cola (de tipo *ptr4*), pero eliminando los nodos.

```
Procedimiento Mostrar_pila(cima)
   {ptr3 cima por valor}
variables
   ptr3 p
inicio
   p:=cima
   mientras p<>nulo hacer
      escribir(datos.nombre(p),datos.apell(p))
      p:=datos.ant(p)
   fin mientras
fin procedimiento

Procedimiento Mostrar_Borrar_cola(frente,final)
   {ptr4 frente,final por referencia}
variables
   ptr4 p
inicio
   mientras frente<>nulo hacer
      escribir(datos.nombre(frente),datos.apell(frente))
      p:=frente
      frente:=datos.prox(frente)
      liberar_memoria(p)
   fin mientras
   final:=frente
fin procedimiento
```

4.-Escribir una subrutina que lea una cadena (de 255 dígitos como máximo) y meta cada carácter en una pila (de tipo *ptr5*) y lo añada a continuación en una cola (de tipo *ptr6*).

Cuando se encuentre el final de la cadena, utilice las operaciones básicas de colas y pilas para determinar si la cadena es un palíndromo (es decir, se lee igual de derecha a izquierda que de izquierda a derecha).

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
Funcion Palindromo(cima,frente,final) devuelve logico
   {ptr5 cima por referencia}
   {ptr6 frente,final por referencia}
variables
   ptr5 p
   ptr6 q
   enteras i
   cadena[255] cad
inicio
   desde i:=1 hasta longitud(cad) hacer
      asignar_memoria(p)
      asignar_memoria(q)
      datos.dato(p):=cad[i]
      datos.ant(p):=cima
      cima:=p
      datos.dato(q):=cad[i]
      datos.prox(q):=nulo
      si final<>nulo entonces
         datos.prox(final):=q
         final:=q
      si no
         final:=q
         frente:=final
      fi si
   fin desde
   si longitud(cad)>0 entonces
      p:=cima
      q:=frente
      mientras (p<>nulo) y (q<>nulo) y (datos.dato(p)=datos.dato(q)) hacer
         p:=datos.ant(p)
         q:=datos.prox(q)
      fin mientras
      si (p=nulo) y (q=nulo) entonces
         Palindromo:=verdadero
      si no
         Palindromo:=falso
      fin si
   fin si
fin funcion
```

BIBLIOGRAFÍA

[DALE89] N. DALE, S.C. LILLY, *Pascal y estructuras de datos.* McGraw Hill, 1989.

[GALVE93] J. GALVE, J.C. GONZÁLEZ, A. SÁNCHEZ, J.A. VELÁZQUEZ, *Algorítmi-*

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

ca: Diseño y análisis de algoritmos funcionales e imperativos. Serie paradigma. Ra-ma, 1993.

[LIPSCHUTZ87] S. LIPSCHUTZ, *Estructura de datos.* McGraw Hill, 1987.

[WIRTH87] N. WIRTH, *Algoritmos y estructuras de datos.* Prentice Hall, 1987.

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

Capítulo 3

ÁRBOLES

3.1 INTRODUCCIÓN

Hasta ahora hemos estudiado varios tipos de estructuras de datos lineales: cadenas, vectores, listas, pilas y colas. En este capítulo se define una nueva estructura de datos no lineal llamada *árbol*. Esta estructura se usa principalmente para representar datos con una relación jerárquica entre sus elementos, como por ejemplo, árboles genealógicos (ver figura 3.1), jerarquía de una empresa, índice de una asignaura, etc..

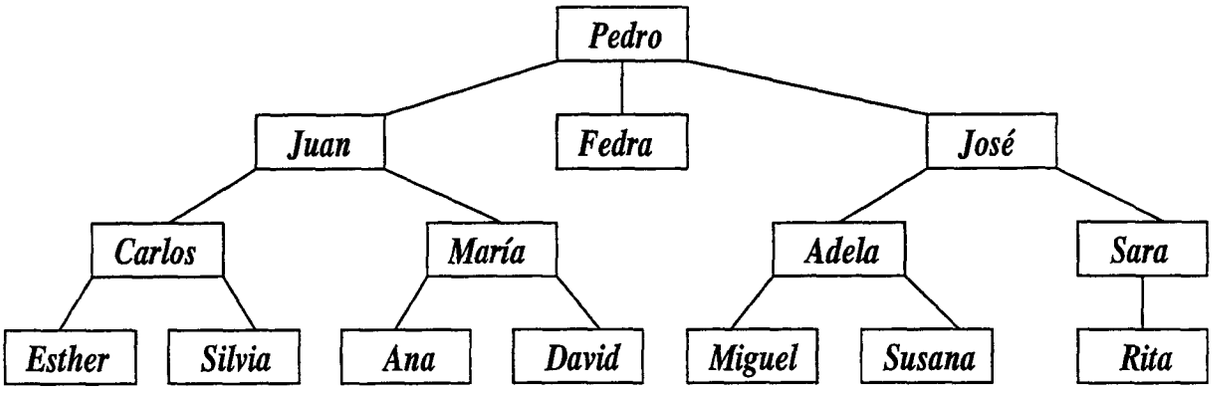


Figura 3.1: Árbol genealógico

En la figura 3.1, Pedro es el padre de Juan, Fedra y José, los cuales son hermanos. Juan, a su vez tiene dos hijos, Carlos y María, que son nietos de Pedro. En definitiva, el árbol de la figura 3.1 muestra los descendientes de Pedro.

3.2 CONCEPTOS

Definimos formalmente un *árbol* como un conjunto finito T de uno o más elementos llamados *nodos* tales que:

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

- Hay un nodo especial llamado *raíz* del árbol
- Los nodos restantes del árbol T se agrupan en $m>=0$ conjuntos disjuntos $T_1...T_m$, los cuales a su vez constituyen un árbol. Los árboles $T_1...T_m$ se llaman *subárboles* de la raíz.

La definición de árbol es recursiva, es decir, se ha definido un árbol en función de árboles. Los árboles con un sólo nodo constan únicamente de la raíz y aquellos árboles con n nodos (n>1) se definen en función de árboles con menos de n nodos. La recursividad es una característica inherente a la estructura de árbol.

De la definición de árbol se deduce que cada nodo de un árbol es la raíz de algún subárbol contenido en la totalidad del mismo. El número de subárboles de un nodo se llama *grado* de ese nodo. Un nodo de grado cero se llama *nodo terminal* o a veces *nodo hoja.* Cada nodo de un árbol tiene asignado un número de *nivel.* A la raíz del árbol se le asigna el número de nivel 0, y al resto de los nodos se le asigna un número de nivel que es superior en una unidad al número de nivel de su padre. Aquellos nodos con el mismo número de nivel se dice que pertenecen a la misma *generación.*

La figura 3.2 muestra un árbol con siete nodos. La raíz es el nodo A, tiene dos subárboles {B} y {C,D,E,F,G} y por tanto, su grado es 2. El nodo A es de nivel 0 por ser el nodo raíz. El subárbol {C,D,E,F,G} tiene el nodo C como raíz. El nodo C es de nivel 1 con respecto a la totalidad del árbol y tiene tres subárboles {D}, {E} y {F, G}; por consiguiente, C es de grado 3. Los nodos terminales de la figura son B, D, E y G; F es el único nodo de grado 1; G es el único nodo de nivel 3.

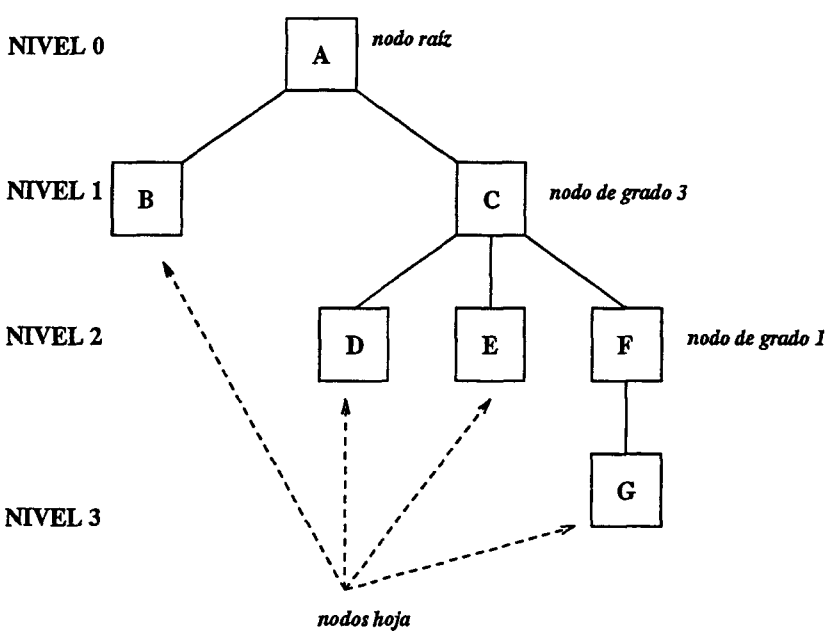


Figura 3.2: Árbol

Si se borra la raíz y las ramas que le conectan al nivel uno, se obtiene un conjunto de árboles separados que se denomina *bosque.*

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

3.3 ÁRBOLES BINARIOS

Un *árbol binario* T se define como un conjunto finito de nodos, de forma que:

- T es vacío (en cuyo caso se llama *árbol nulo o vacío*) o
- T contiene un nodo llamado raíz de T, formando un par, como máximo, de árboles binarios disjuntos T_1 y T_2 los nodos restantes

Si T contiene una raíz R, los dos árboles T_1 y T_2 se llaman, respectivamente, *subárboles izquierdo y derecho* de la raíz R. Si T_1 no es vacío, entonces su raíz se llama *sucesor izquierdo* de R y de igual forma, si T_2 no es vacío, su raíz se llama *sucesor derecho* de R.

La figura 3.3 representa a un árbol binario con 11 nodos, representados por las letras de la A a la L, excluyendo la I. La raíz del árbol es el nodo de A. Una línea hacia abajo y a la izquierda de un nodo dado representa el sucesor izquierdo de ese nodo, y una línea hacia abajo y a la derecha de un nodo determinado representa el sucesor derecho de ese nodo. En la figura 3.3 observamos que:

- B es un sucesor izquierdo de A, mientras que C es un sucesor derecho de A.
- El subárbol izquierdo de la raíz A está formado por los nodos con B,D,E y F, y el subárbol derecho de A está formado por los nodos con C,G,H,J,K y L.

En definitiva, cualquier nodo en un árbol binario tiene 0, 1 ó 2 sucesores. Los nodos A,B,C y H tienen dos sucesores, los nodos E y J sólo tienen un sucesor, y los nodos D,F,G,L y K no tienen sucesores, por lo que son nodos hoja o terminales.

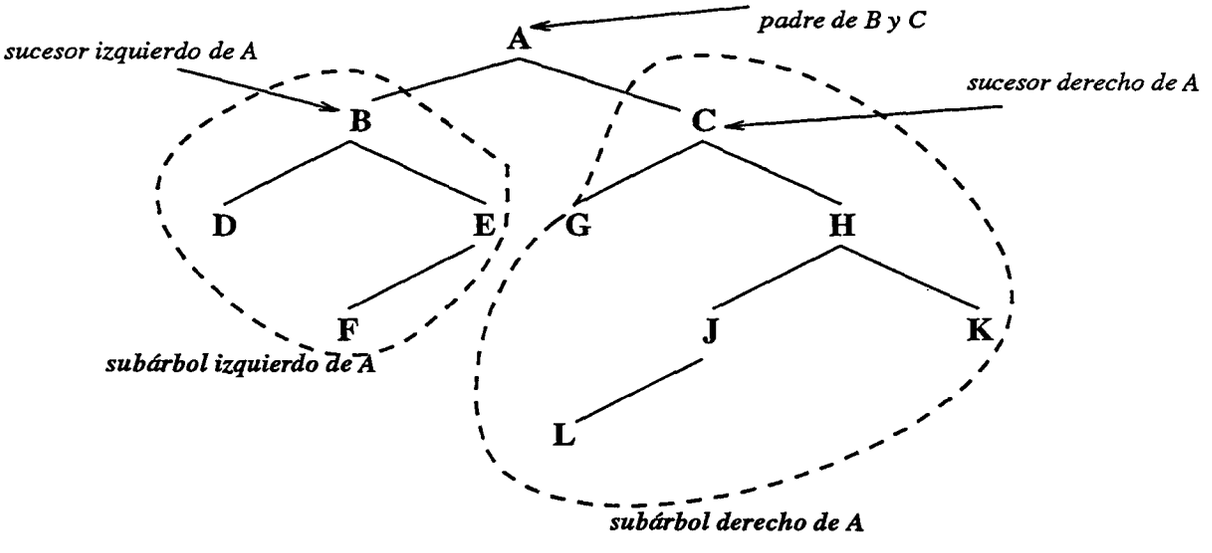


Figura 3.3: Árbol binario

Como se dijo anteriormente, la definición de árbol binario T es recursiva puesto que se define en función de los subárboles binarios T_1 y T_2. Esto significa que cada nodo N

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

de T contiene un subárbol izquierdo y uno derecho. Si N es nodo hoja, ambos subárboles están vacíos.

Dos árboles binarios T y T' son *similares* si tienen la misma forma. Dos árboles son *copia* si son similares y sus nodos tienen los mismos contenidos. En la figura 3.4 los árboles a), c) y d) son similares. Los árboles a) y c) son copias, ya que además tienen los mismos datos en los correspondientes nodos. El árbol b) no es similar ni es una copia del d).

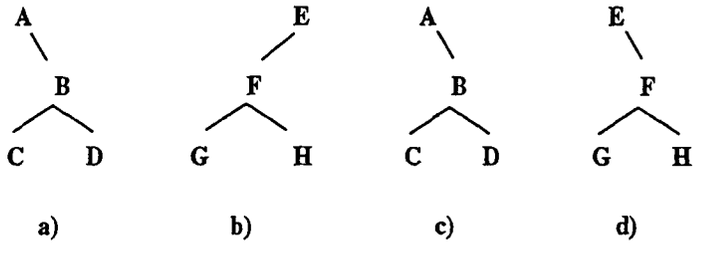


Figura 3.4: Árboles similares (a), c) y d)) y árboles copia (a) y c))

Frecuentemente se usa una terminología de relaciones familiares para describir las relaciones entre los nodos de un árbol T. Si N es un nodo de T con un sucesor izquierdo S_1 y un sucesor derecho S_2, a N se le llamará *padre* de S_1 y S_2. Análogamente, S_1 se llama *hijo izquierdo* de N y S_2 el *hijo derecho* de N. Es más, S_1 y S_2 se dice que son *hermanos*. Cada nodo N de un árbol binario T, excepto la raíz, tiene un único padre, llamado *predecesor* de N. Se dice que un nodo L es *descendiente* de otro nodo N (y a su vez N se dice que es *antecesor* de L) si existe una sucesión de hijos desde N hasta L, siendo L *descendiente izquierdo* o *derecho* de N dependiendo de si pertenece al subárbol izquierdo o al derecho de N.

La línea dibujada entre un nodo N del árbol T y su sucesor se llama *arista* y una secuencia de aristas consecutivas se llama *camino*. Un camino que termina en un nodo hoja se llama *rama*.

La *profundidad* (o *altura*) de un árbol T es el número máximo de nodos de una rama de T. Equivale al mayor número de nivel de T incrementado en 1. Por ejemplo, el árbol T de la figura 3.3 tiene una profundidad de 5.

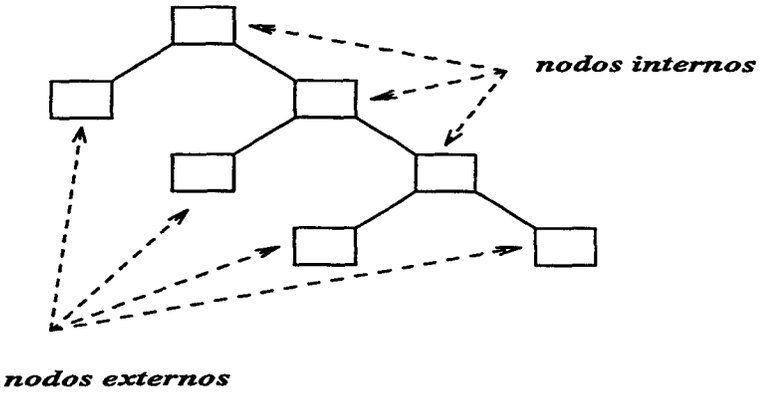


Figura 3.5: Árbol binario extendido o ampliado

Un *árbol binario extendido o árboles-2* es un árbol binario en el cual cada nodo es de grado 0 ó 2 (ver figura 3.5). Los nodos con 2 hijos se llaman nodos internos y los nodos

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

con 0 hijos se llaman nodos externos.

A partir de un árbol binario T se puede obtener un árbol-2 reemplazando cada subárbol vacío por un nuevo nodo, el cual representaremos en la figura 3.6 por un cuadrado. El árbol obtenido es un árbol-2, en el cual los nodos del árbol binario original T son los nodos internos del árbol-2, y los nuevos nodos son los nodos externos del árbol extendido.

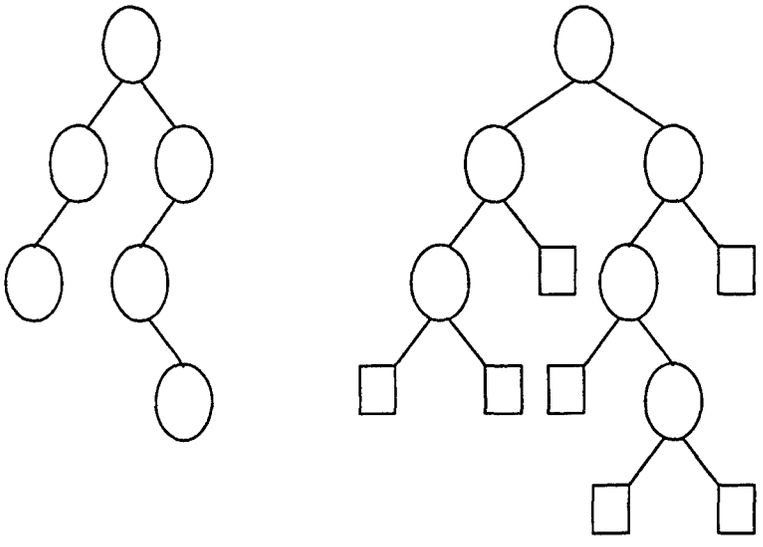


Figura 3.6: Conversión de árbol binario T en árbol-2

Un *árbol binario completo* es aquel en el cual todos los nodos de grado cero o uno están en los dos últimos niveles, *l-1* y *l*, de forma que las hojas del último nivel, *l*, ocupen las posiciones más a la izquierda de dicho nivel. En la figura 3.7 se muestra un árbol binario completo T_{26} de 26 nodos. Los nodos del árbol T_{26} han sido etiquetados con los enteros 1,2,...,26, de izquierda a derecha y de generación en generación.

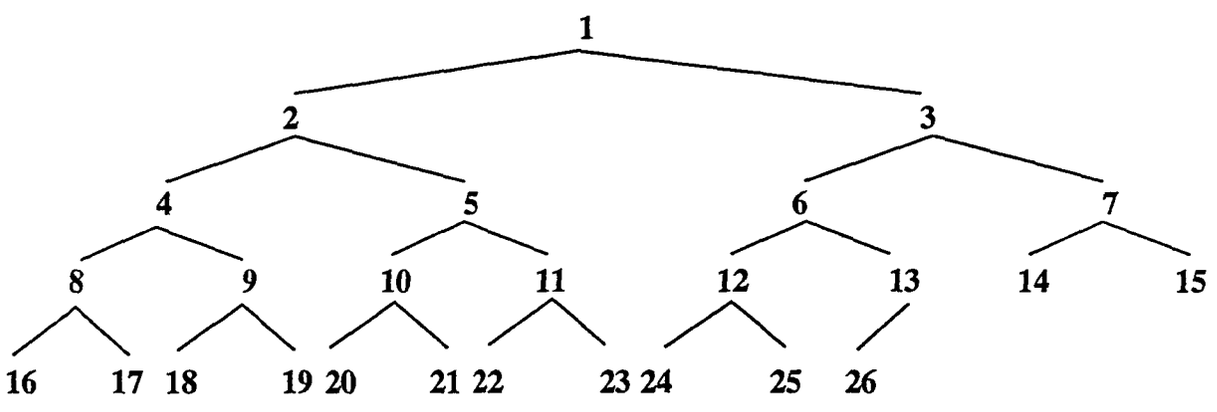


Figura 3.7: Árbol binario completo

Un nodo está *descompensado por la izquierda* si el camino más largo en su subárbol izquierdo es una unidad mayor que el camino más largo de su subárbol derecho. Un nodo está *equilibrado* cuando caminos máximos son iguales. Un nodo está *descompensado por la derecha* cuando el camino más largo asociado al subárbol derecho es una unidad mayor que el camino más largo asociado al subárbol izquierdo del nodo en cuestión. Si cada nodo de un árbol binario está en uno de esos tres estados se dice que el árbol está *equilibrado* (ver figura 3.8).

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

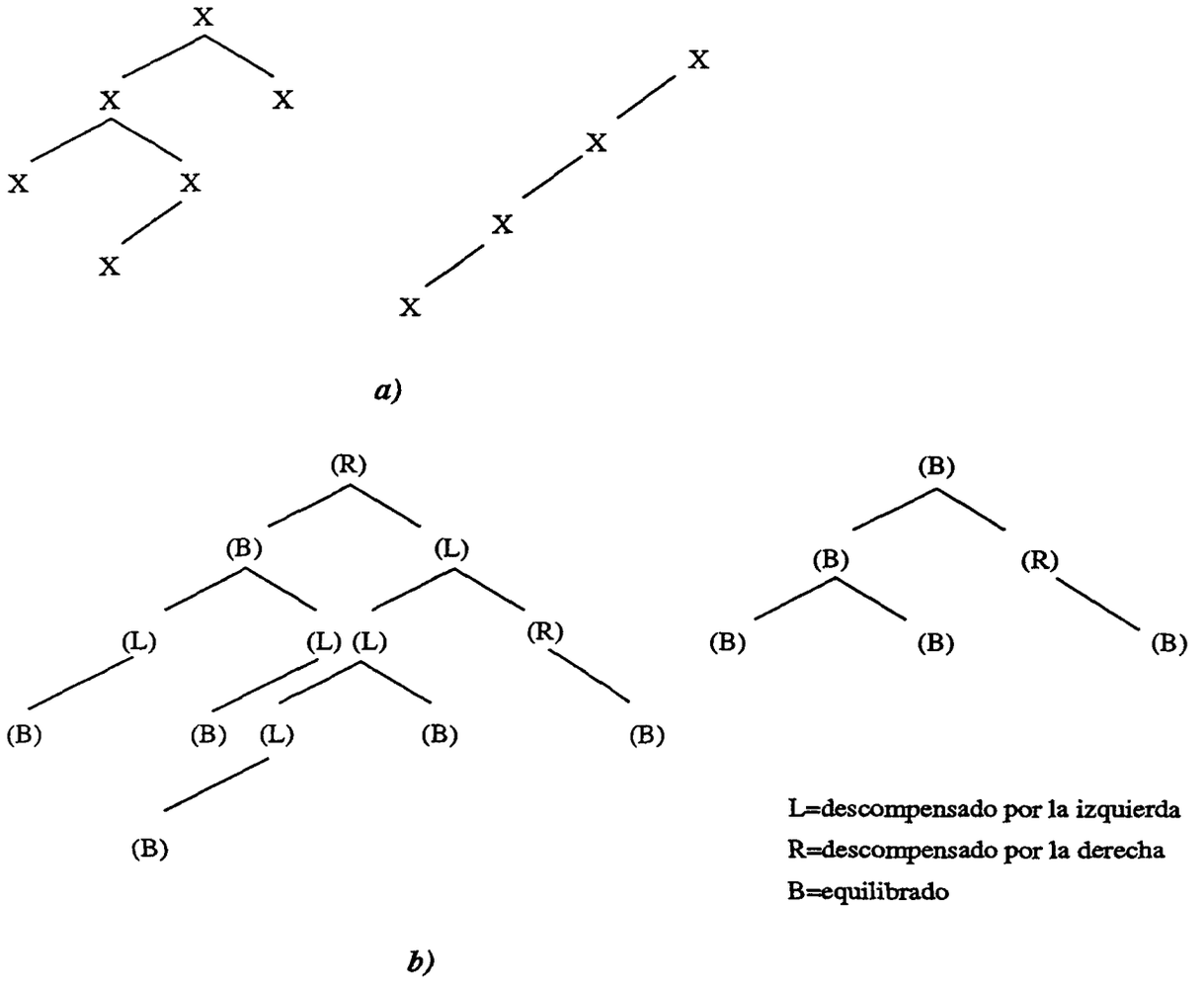


Figura 3.8: a)Árboles no equilibrados. b) Árboles equilibrados

3.4 REPRESENTACIÓN ENLAZADA DE ÁRBOLES BINARIOS EN MEMORIA

Un árbol binario está formado por nodos que pueden tener como máximo dos descendientes. Por tanto, la representación enlazada de tales árboles implica nodos de la forma:

LPTR	INFO	RPTR

donde LPTR y RPTR representan las direcciones de los subárboles izquierdo y derecho, respectivamente, de un nodo raíz determinado. Los subárboles vacíos se representan por un valor nulo en los campos LPTR y RPTR del nodo. El campo INFO del nodo contiene los datos del nodo.

En definitiva, cada nodo N de un árbol binario tendrá en memoria una posición K, de forma que:

1. LPTR(K) *contiene la localización del hijo izquierdo de* N

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

2. INFO(K) contiene los datos del nodo N
3. RPTR(K) contiene la localización del hijo derecho de N

La figura 3.9 muestra una representación enlazada de un árbol binario. La variable T es una variable puntero que indica la dirección de la raíz. Si algún subárbol está vacío, el correspondiente puntero contendrá el valor nulo (en la representación, el valor nulo se representa por una barra /); si el árbol está vacío, entonces la variable puntero T tendrá el valor nulo.

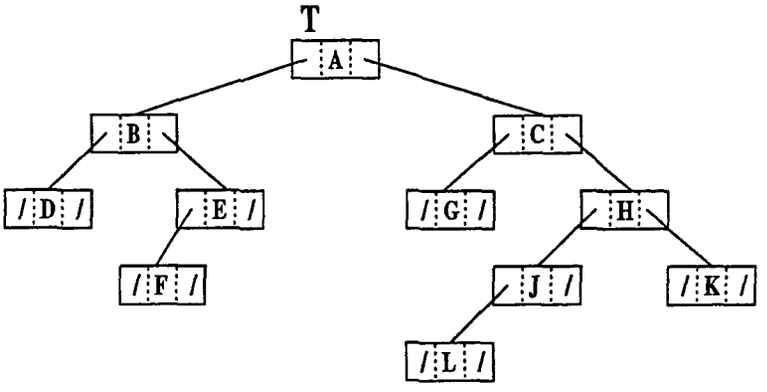


Figura 3.9: Representación enlazada de árbol binario

La figura 3.10 muestra como aparecería en memoria una representación enlazada del árbol de la figura 3.9.

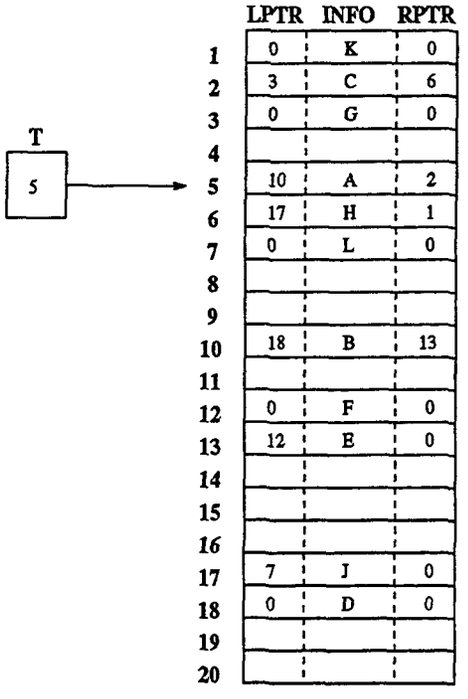


Figura 3.10: Representación enlazada de un árbol binario en memoria

Los números del 1 al 20 representan direcciones de memoria. El puntero T contiene la dirección de la raíz. Los campos LPTR y RPTR contienen en cada nodo las direcciones de memoria de los hijos izquierdo y derecho, respectivamente. Aquellos punteros con valor nulo aparecen con un cero en la figura.

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

3.5 RECORRIDOS DE UN ÁRBOL

3.5.1 ESQUEMAS DE RECORRIDO DE UN ÁRBOL BINARIO

Hay varios algoritmos para manejo de las estructuras en árbol, y una idea que aparece en esos algoritmos es la noción de *recorrido* de un árbol. Éste es un método para examinar los nodos del árbol sistemáticamente de forma que cada nodo sea visitado exactamente una vez. Un recorrido completo de un árbol nos da una distribución lineal de los nodos, haciendo que los algoritmos sean más fáciles a la hora de proporcionar el nodo siguiente en la secuencia.

Existen tres métodos estándar para recorrer un árbol binario:

1. Preorden
2. Inorden o simétrico
3. Postorden

En estos métodos, cuando el árbol no está vacío, el recorrido se efectúa en tres pasos:

- Recorrido en **Preorden**:
  1. Visitar la raíz
  2. Recorrer el subárbol izquierdo
  3. Recorrer el subárbol derecho
- Recorrido en **Inorden**:
  1. Recorrer el subárbol izquierdo
  2. Visitar la raíz
  3. Recorrer el subárbol derecho
- Recorrido en **Postorden**:
  1. Recorrer el subárbol izquierdo
  2. Recorrer el subárbol derecho
  3. Visitar la raíz

En cada uno de estos recorridos se llevan a cabo los mismos pasos. La diferencia entre ellos es el momento en el que se procesa la raíz. En el recorrido en preorden, la raíz se procesa antes de recorrer los subárboles (*pre*). En el recorrido del árbol en inorden, la

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

raíz se procesa entre los recorridos de los subárboles (*in*). Por último, en el recorrido en postorden la raíz se procesa después de recorrer los subárboles (*post*).

Los tres algoritmos se denominan a veces, respectivamente, recorrido nodo-izquierda-derecha (del inglés, NLR), recorrido izquierda-nodo-derecha (del inglés, LNR) y recorrido izquierda-derecha-nodo (del inglés, LRN).

Los algoritmos de recorrido de árboles binarios se definen de forma recursiva, ya que cada algoritmo implica el recorrido de los subárboles en un orden dado. Estos algoritmos también pueden implementarse de forma iterativa mediante el uso de pilas.

Veamos un ejemplo de cada tipo de recorrido en el árbol binario de la figura 3.11.

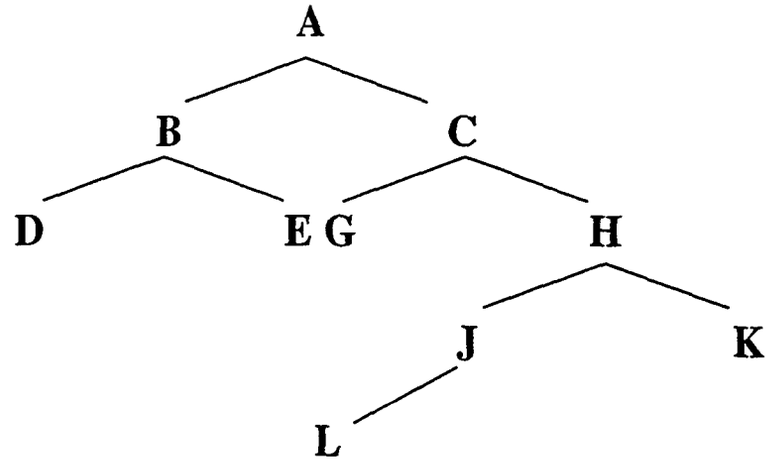


Figura 3.11: Árbol binario

3.5.1.1 RECORRIDO EN PREORDEN

El recorrido en preorden (ver figura 3.12) del árbol binario de la figura 3.11 es el siguiente: A,B,D,E,C,G,H,J,L,K.

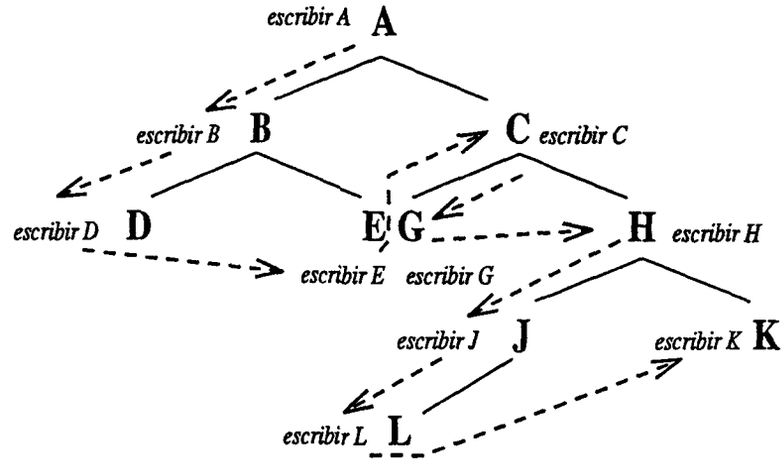


Figura 3.12: Recorrido en preorden de un árbol binario

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

3.5.1.2 RECORRIDO EN INORDEN

El recorrido en inorden (ver figura 3.13) del árbol binario de la figura 3.11 es el siguiente: D,B,E,A,G,C,L,J,K,H,K.

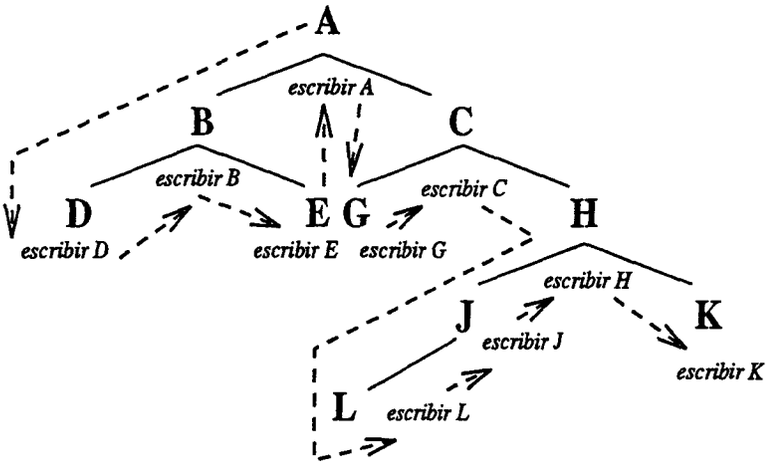


Figura 3.13: Recorrido inorden de un árbol binario

3.5.1.3 RECORRIDO EN POSTORDEN

El recorrido en postorden (ver figura 3.14) del árbol binario de la figura 3.11 es el siguiente: D,E,B,G,L,J,K,H,C,A.

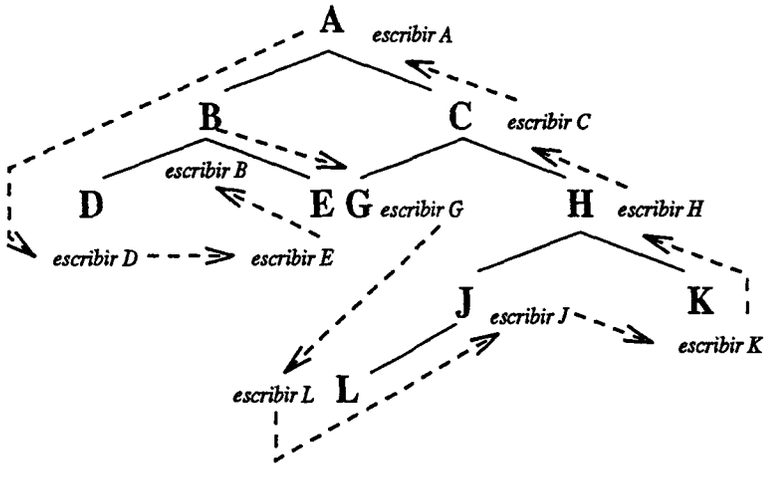


Figura 3.14: Recorrido en postorden de un árbol binario

3.5.2 MÉTODOS RECURSIVOS PARA LOS RECORRIDOS DE UN ÁRBOL BINARIO

A continuación se presentan los algoritmos de recorrido de un árbol binario de forma recursiva. A cada uno de los procedimientos se le pasa una variable puntero que representa

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

la dirección de la raíz del subárbol que se va a recorrer. La declaración de la estructura de datos a usar en los tres procedimientos es la siguiente:

```
tipo
   TipoPuntero=puntero a TipoNodo
   registro TipoNodo
      caracter INFO
      TipoPuntero LPTR,RPTR
   fin registro
```

3.5.2.1 RECORRIDO EN PREORDEN

El algoritmo recursivo para el recorrido en preorden de un árbol binario es:

```
Procedimiento RPreorden(T)
      {TipoPuntero T por valor}
inicio
   si T<>nulo entonces
      escribir(INFO(T))
      RPreorden(LPTR(T))
      RPreorden(RPTR(T))
   fin si
fin procedimiento
```

3.5.2.2 RECORRIDO EN INORDEN

El algoritmo recursivo para el recorrido en inorden de un árbol binario es:

```
Procedimiento RInorden(T)
      {TipoPuntero T por valor}
inicio
   si T<>nulo entonces
      RInorden(LPTR(T))
      escribir(INFO(T))
      RInorden(RPTR(T))
   fin si
fin procedimiento
```

3.5.2.3 RECORRIDO EN POSTORDEN

El algoritmo recursivo para el recorrido en postorden de un árbol binario es:

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
Procedimiento RPostorden(T)
      {TipoPuntero T por valor}
inicio
   si T<>nulo entonces
      RPostorden(LPTR(T))
      RPostorden(RPTR(T))
      escribir(INFO(T))
   fin si
fin procedimiento
```

3.6 REPRESENTACIÓN SECUENCIAL DE ÁRBOLES BINARIOS

Hay varias formas de representar un árbol en memoria, además de la representación enlazada. La elección de la representación apropiada depende fundamentalmente de qué clase de operaciones queramos realizar en los árboles. Otra forma de representar un árbol en memoria es mediante el uso de técnicas de memoria secuencial. Estas técnicas son adecuadas cuando deseemos una representación compacta de la estructura árbol que no cambia mucho (inserciones, extracciones...).

3.6.1 REPRESENTACIÓN CONTIGUA DE ÁRBOLES BINARIOS COMPLETOS

En un árbol binario, cada nodo puede tener como mucho 2 hijos. De acuerdo con ésto se puede probar que el nivel k de un árbol puede tener como mucho 2^k-1 nodos. Si el árbol es completo, todos sus niveles, excepto probablemente el último, tienen el máximo número de nodos posibles y todos los nodos del último nivel están situados lo más a la izquierda. Así, sólo existe un único árbol completo T_n con exactamente n nodos.

En un árbol binario completo las posiciones de los hijos izquierdo y derecho de un nodo i son, $2\times i$ y $2\times i+1$ respectivamente. De la misma forma, la posición del padre del nodo i es la parte entera de $i/2$. Por ejemplo, la figura 3.15 presenta un árbol binario completo y su correspondiente representación secuencial: el árbol tiene 3 niveles y por tanto, 7 nodos. La posición del hijo izquierdo de A en el vector INFO es 2 (2×1, siendo 1 la posición de A) y la de su hijo izquierdo es 3 ($2\times1+1$).

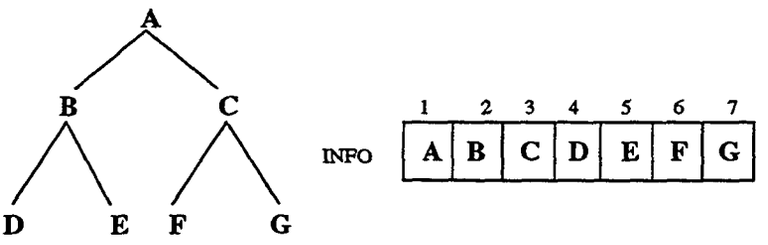


Figura 3.15: Representación secuencial de árbol binario completo

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

Esta representación secuencial de árboles binarios completos presenta el inconveniente de que si el árbol tiene menos de 2^n-1 nodos (siendo n el número de niveles del árbol) (caso de figura 3.7) se desperdicia memoria.

3.6.2 ENUMERACIONES SECUENCIALES

3.6.2.1 REPRESENTACIÓN EN PREORDEN SECUENCIAL

Otra posibilidad de representación secuencial es almacenar los nodos secuencialmente siguiendo el recorrido del árbol en preorden (ver figura 3.16), con los campos INFO, RPTR y TAG. El campo TAG está a 1 si el nodo en cuestión es un nodo hoja. El campo RPTR apunta al hijo derecho del nodo en cuestión.

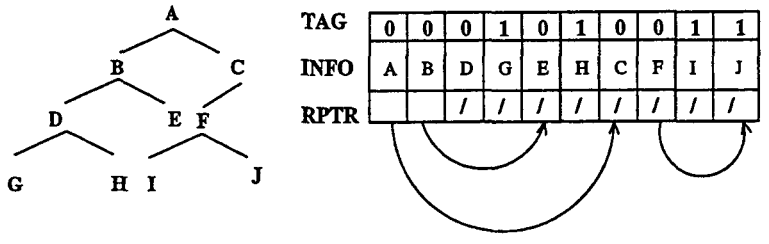


Figura 3.16: Representación en preorden secuencial

En esta representación se desperdicia memoria ya que muchos de los encadenamientos derechos son nulos. Se puede mejorar esta representación mediante el campo puntero RANGO, como aparece en la figura 3.17. El puntero RANGO de cada nodo apunta a su sucesor derecho, en caso de tenerlo. Si el nodo en cuestión carece de sucesor derecho o es sucesor derecho de forma directa de otro nodo (el RPTR de ese nodo apunta al nodo en cuestión) apuntará al sucesor derecho de su padre, si éste no tiene, al de su abuelo y así sucesivamente. En la representación secuencial resultante no se necesita el campo TAG. Veamos el ejemplo de la figura 3.17 en la que el sucesor derecho del nodo A es el nodo C (en este caso el sucesor derecho es directo), con lo cual el campo RANGO de A apunta a C; el sucesor derecho del nodo B es el E (también directo), apuntando su RANGO a E; en el caso del nodo D no tiene sucesor derecho por lo que su RANGO apunta al sucesor derecho de su padre, B, es decir, al nodo E; el nodo G tampoco tiene sucesor derecho ni tampoco lo tiene su padre por lo que su RANGO apuntará al sucesor derecho de su abuelo, B, o sea, el nodo E. Para los nodos J y C al no existir sucesores derechos posibles sus RANGO apuntarán a nulo.

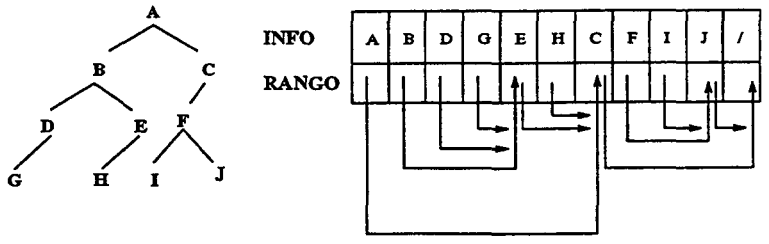
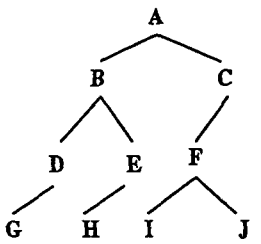


Figura 3.17: Representación en preorden secuencial mejorada

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

3.6.2.2 REPRESENTACIÓN EN POSTORDEN CON GRADOS

En este método se utilizan dos vectores paralelos, uno para representar el recorrido en postorden de un árbol y el otro para indicar el grado de cada nodo (ver figura 3.18).



INFO	G	D	H	E	B	I	J	F	C	A
GRADOS	0	1	0	1	2	0	0	2	1	2

Figura 3.18: Representación en postorden con grados

3.7 OPERACIONES BÁSICAS SOBRE ÁRBOLES

3.7.1 ÁRBOLES BINARIOS ORDENADOS

Anteriormente se han estudiado las técnicas de búsqueda secuencial y binaria en vectores. La mejor de ambas era la binaria siempre que los elementos del vector estuvieran ordenados. El hecho de que el vector esté ordenado presenta inconvenientes para otras operaciones tales como la inserción y extracción de elementos. La inserción de un elemento en un conjunto ordenado puede dar lugar al movimiento de muchos otros para así mantener el orden. Una operación de extracción puede provocar el mismo problema.

Una forma de llevar a cabo operaciones de inserción y extracción más fácilmente puede hacerse mediante el uso de listas enlazadas. Sin embargo, en este tipo de estructura, es costosa la búsqueda de elementos puesto que se debe usar una búsqueda secuencial.

Los inconvenientes que presentaban los vectores y listas enlazadas con respecto a las operaciones de búsqueda, inserción y extracción pueden aliviarse mediante un árbol binario de búsqueda.

El *árbol binario de búsqueda* o *árbol binario ordenado* permite llevar a cabo la búsqueda binaria en un árbol. Se dice que un árbol binario es un árbol binario de búsqueda (ver figura 3.19) si cada nodo N de T cumple la siguiente propiedad: el valor de N es mayor que cualquier valor del subárbol izquierdo de N y es menor que cualquier valor del subárbol derecho de N (esta propiedad garantiza que el recorrido inorden de un árbol binario de búsqueda dará una lista ordenada de los elementos del árbol).

3.7.2 BÚSQUEDA EN ÁRBOLES BINARIOS ORDENADOS

El algoritmo de búsqueda en un árbol binario de búsqueda T encuentra la posición de un elemento, *Item*, en dicho árbol T. El método empleado es el siguiente:

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

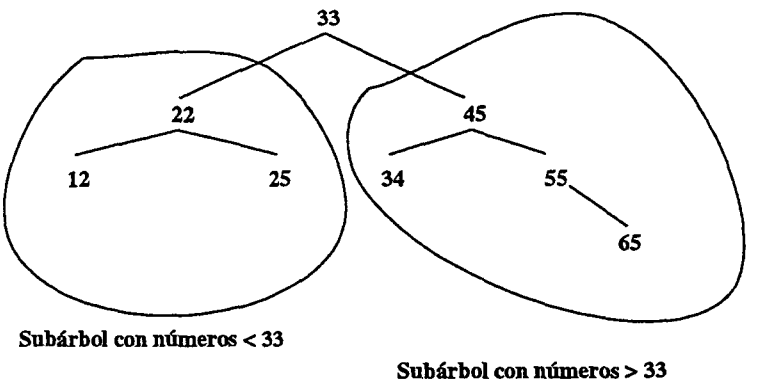


Figura 3.19: Árbol binario de búsqueda

- Se compara *Item* con el nodo raíz del árbol. Se pueden presentar dos casos:
  - Si *Item*<N, se procede con el hijo izquierdo de N
  - Si *Item*>N, se procede con el hijo derecho de N
- Se repite el paso anterior hasta que se cumpla una de las siguientes condiciones:
  - Se encuentra un nodo N tal que *Item*=INFO(N). En este caso la búsqueda ha tenido éxito.
  - Se encuentra un subárbol vacío, lo cual indica que la búsqueda ha sido infructuosa. En tal caso, se podría proceder a insertar el elemento.

En definitiva, se desciende desde la raíz del árbol hasta que se encuentre el elemento o bien hasta que se inserte.

Considérese un árbol binario de búsqueda de la figura 3.20. Si se desea buscar el elemento 22, los pasos a seguir serían los siguientes:

- Se compara *Item*=22 con la raíz del árbol, 40. Como 22<40, se procede con el hijo izquierdo de 40 que es 16.
- Se compara *Item*=22 con 16. Como 22>16, se pocede con el hijo derecho de 20 que es 25.
- Se compara *Item*=22 con 25. Como 22<25, se procede con el hijo izquierdo de 25 que es 20.
- Se compara *Item*=22 con 20. Como 22>20, se procede con el hijo derecho de 20 que es 22.

Para simplicar la inserción y borrado resulta conveniente añadir al principio de un árbol binario un nodo especial llamado *cabeza*, el cual es el padre de la raíz del árbol, siendo la raíz del árbol el hijo izquierdo del nodo cabeza. Cuando se usa este nodo extra, la variable puntero *Cabeza* apuntará al nodo cabeza, y el hijo izquierdo del nodo cabeza será la raíz del árbol binario (ver figura 3.21). Si un árbol binario está vacío, el hijo

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

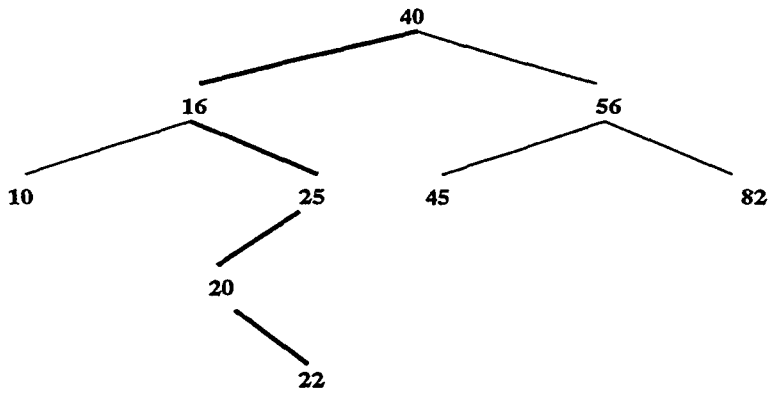


Figura 3.20: Búsqueda del elemento 22 en árbol binario de búsqueda

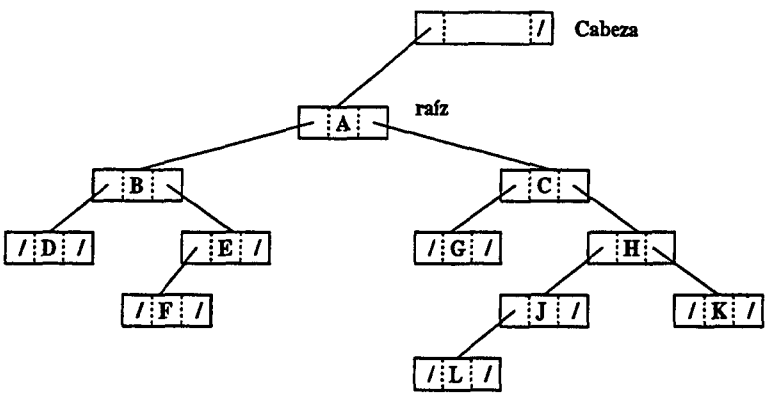


Figura 3.21: Árbol binario con nodo cabeza

izquierdo del nodo cabeza estará a nulo (*LPTR(Cabeza)=nulo*). Debe tenerse en cuenta que el *RPTR(Cabeza)* siempre es nulo.

A continuación se presenta el algoritmo de búsqueda: Un árbol binario de búsqueda, cuya raíz viene dada por *Raiz*, está en memoria y se da un *Item* de información a buscar en el árbol. Este procedimiento devuelve la posición de *Item* en el árbol, *Pos*, así como la posición del padre de *Item*, *Pad*. Existen tres casos especiales:

1. *Pos=nulo* y *Pad=nulo*, lo que indica que el árbol está vacío.
2. *Pos<>nulo* y *Pad=nulo*, lo que indica que *Item* es la raíz de T.
3. *Pos=nulo* y *Pad<>nulo*, lo que indica que *Item* no está en el árbol y que puede ser añadido al árbol como hijo del nodo N de la posición *Pad*.

```
tipo
  TipoPuntero=puntero a TipoNodo
  registro TipoNodo
    caracter INFO
    TipoPuntero LPTR,RPTR
  fin registro
```

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
Procedimiento BusquedaBinaria(Raiz,Item,Pos,Pad)
      {TipoPuntero Raiz,Pos,Pad por referencia}
      {caracter Item por valor}
variables
   TipoPuntero P
   logicas encontrado
inicio
   si Raiz=nulo entonces
      {Árbol vacío}
      Pos:=nulo
      Pad:=nulo
   si no
      si Item=INFO(Raiz) entonces
         {Item está en la raíz}
         Pos:=Raíz
         Pad:=nulo
      si no
         Pad:=Raiz
         si Item<INFO(Raiz) entonces
            P:=LPTR(Raiz)
         si no
            P:=RPTR(Raiz)
         fin si
         encontrado:=falso
         mientras (P<>nulo) y (encontrado=falso) hacer
            si Item=INFO(P) entonces
               Pos:=P
               encontrado:=verdadero
            si no
               Pad:=P
               si Item<INFO(P) entonces
                  P:=LPTR(P)
               si no
                  P:=RPTR(P)
               fin si
            fin si
         fin mientras
         si encontrado=falso entonces
            Pos:=nulo
         fin si
      fin si
   fin si
fin procedimiento
```

Si se realiza la búsqueda de un elemento en un árbol binario de búsqueda T, se observa que el número de comparaciones está limitado por la profundidad del árbol. Ello es debido a que descendemos por un único camino del árbol. Así, el tiempo de búsqueda en el árbol es proporcional a la profundidad del árbol, en el peor de los casos.

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

3.7.3 INSERCIÓN EN ÁRBOLES BINARIOS ORDENADOS

Para insertar un nodo en un árbol binario de búsqueda se ha de determinar en qué lugar del árbol se ha de insertar el elemento para que el árbol permanezca ordenado. Para obtener la posición en la que se ha de insertar el elemento se hará uso del procedimiento de búsqueda binaria del apartado anterior.

Los casos que se pueden presentar a la hora de insertar un elemento *Item* en un árbol binario de búsqueda son los siguientes:

- El elemento en cuestión ya está en el árbol. En tal caso, el algoritmo de búsqueda binaria nos devuelve una dirección distinta de nulo y no se procede a la inserción ya que daría lugar a elementos duplicados.
- El elemento a insertar no se encuentra en el árbol, por tanto, la dirección devuelta por la función de búsqueda binaria es nula. En este caso, se crea el nodo y se arreglan los enlaces.

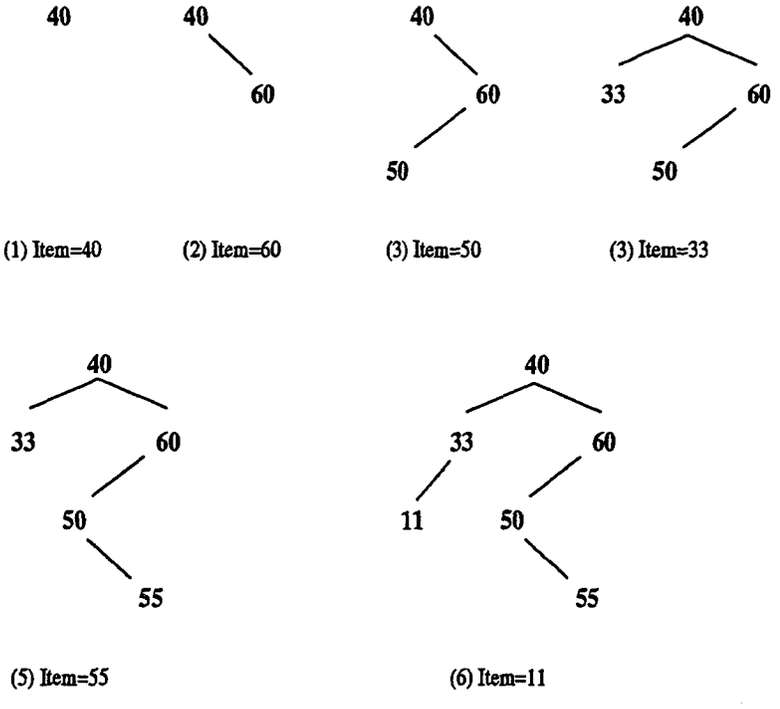


Figura 3.22: Inserción en árbol binario de búsqueda vacío

Supongamos que los siguientes números se insertan en orden en un árbol binario de búsqueda vacío:

40,60,50,33,55,11

La figura 3.22 muestra las seis inserciones. Si los seis números se hubieran dado en un orden diferente, entonces el árbol podría haber sido diferente y además podría haber tenido una profundidad diferente.

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

A continuación se presenta un procedimiento que dado un árbol binario con un nodo raíz, *Raiz*, y la información que se desea añadir, *Item*, inserta el nodo como descendiente izquierdo o derecho del nodo correspondiente, siempre y cuando no se encuentre *Item* en el árbol. El procedimiento devuelve la dirección del nodo insertado, *Pos*, si es posible la inserción y nulo en caso contrario.

```
tipo
   TipoPuntero=puntero a TipoNodo
   registro TipoNodo
      caracter INFO
      TipoPuntero LPTR,RPTR
   fin registro

Procedimiento Insercion (Item,Pos,Raiz)
      {caracter Item por valor}
      {TipoPuntero Pos,Raiz por referencia}
variables
   TipoPuntero Pad,Nuevo
inicio
   BusquedaBinaria(Raiz,Item,Pos,Pad)
   si Pos<>nulo entonces
      escribir('Nodo duplicado')
   si no
      asignar_memoria(Nuevo)
      INFO(Nuevo):=Item
      LPTR(Nuevo):=nulo
      RPTR(Nuevo):=nulo
      Pos:=Nuevo
      si Pad=nulo entonces
         Raiz:=Nuevo
      si no
         si Item<INFO(Pos) entonces
            LPTR(Pad):=Nuevo
         si no
            RPTR(Pad):=Nuevo
         fin si
      fin si
   fin si
fin procedimiento
```

3.7.4 ELIMINACIÓN EN ÁRBOLES BINARIOS ORDENADOS

Tenemos un árbol binario de búsqueda, T, en memoria y deseamos eliminar un elemento dado, *Item*.

El algoritmo de eliminación hace uso del algoritmo de búsqueda de la posición del nodo N que contiene a *Item*, así como la posición del padre de N, *P(N)*. La forma en que

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

N es eliminado del árbol depende principalmente del número de hijos de N. Existen tres casos:

1. N no tiene hijos. Entonces N se elimina de T simplemente reemplazando la posición de N en $P(N)$ por el puntero nulo.
2. N tiene exactamente un hijo. Entonces N se elimina de T simplemente reemplazando la posición de N en $P(N)$ por la posición del hijo único de N.
3. N tiene dos hijos. Sea $S(N)$ el sucesor inorden de N. Entonces N es eliminado de T primero eliminando $S(N)$ de T (mediante los casos 1 ó 2) y luego reemplazando N en T por el nodo $S(N)$.

En cualquiera de los tres casos, el espacio de memoria del nodo eliminado, N, se debe liberar de memoria.

Consideremos el árbol binario T de la figura 3.23. Veamos un ejemplo de cada uno de los casos anteriores:

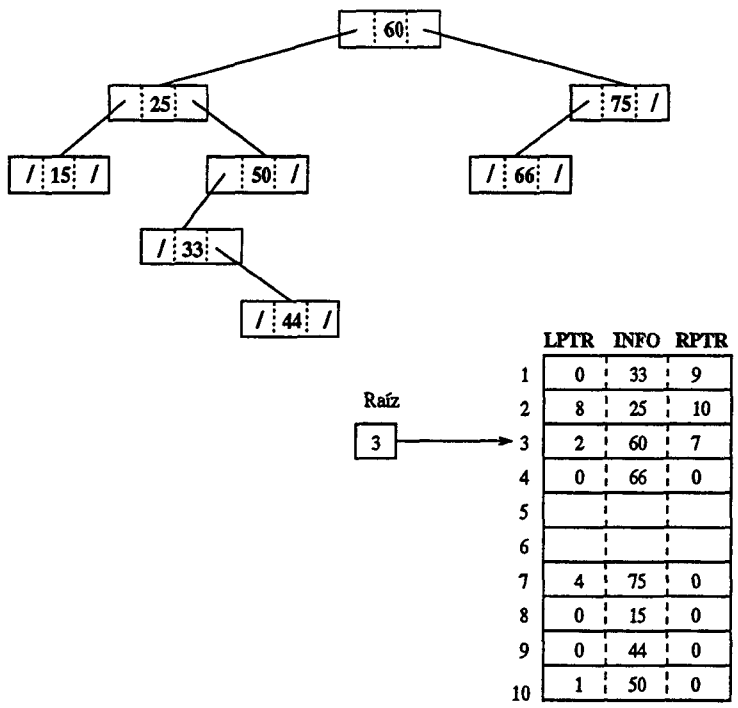
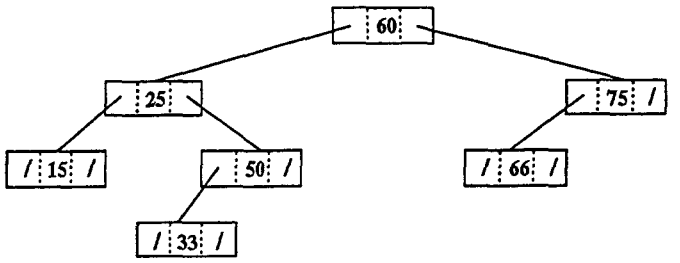


Figura 3.23: Árbol binario de búsqueda y su representación enlazada en memoria

1. Supongamos que eliminamos el nodo 44 del árbol T. La figura 3.24 muestra el árbol tras eliminar el 44, así como la representación enlazada correspondiente. El borrado se lleva a cabo simplemente asignando nulo al RPTR del nodo padre, 33. A continuación se libera memoria.

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006



Raíz

3

	LPTR	INFO	RPTR
1	0	33	0
2	8	25	10
3	2	60	7
4	0	66	0
5			
6			
7	4	75	0
8	0	15	0
9			
10	1	50	0

Representación enlazada en memoria

Figura 3.24: Árbol binario después de eliminar el nodo 44

2. Supongamos que se elimina el nodo 75, el cual sólo tiene un hijo, del árbol de la figura 3.23. La eliminación se lleva a cabo cambiando el puntero derecho del nodo padre, 60, que inicialmente apunta al 75, de forma que apunte ahora al nodo 66, el hijo único de 75 (ver figura 3.25). Al igual que en el caso anterior, se libera el espacio de memoria.

3. Supongamos que del árbol de la figura 3.23 eliminamos el nodo 25. El nodo 25 tiene dos hijos. Observamos, además que el nodo 33 es el sucesor inorden del nodo 25. La figura 3.26 muestra el árbol binario y la representación enlazada resultante tras borrar el 25. El borrado se lleva a cabo borrando primero el 33 del árbol (33 es el sucesor inorden del nodo a borrar) y luego reemplazando el nodo 25 por el nodo 33. El reemplazo del nodo 25 por el 33 se realiza en memoria únicamente cambiando punteros, no moviendo los contenidos del nodo de una posición a otra. Finalmente, el LPTR del padre de 33, 50 apunta al hijo derecho de 33, por éste distinto de nulo.

El algoritmo de eliminación se basará en dos procedimientos CasoA y CasoB, los cuales se presentan a continuación. El procedimiento CasoA se refiere a los casos 1 y 2, en los que el nodo a eliminar no tiene hijos o bien tiene un solo hijo (derecho o izquierdo); el procedimiento CasoB se refiere al caso 3, en el que el nodo a eliminar tiene dos hijos. En este último procedimiento el sucesor inorden del nodo a eliminar se puede encontrar moviéndose al hijo derecho del nodo a eliminar y luego moviéndose repetidamente hacia la izquierda hasta que se encuentre un nodo sin hijo izquierdo. Existen muchos subcasos que reflejan los hechos de que N sea hijo izquierdo, derecho o la raíz.

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

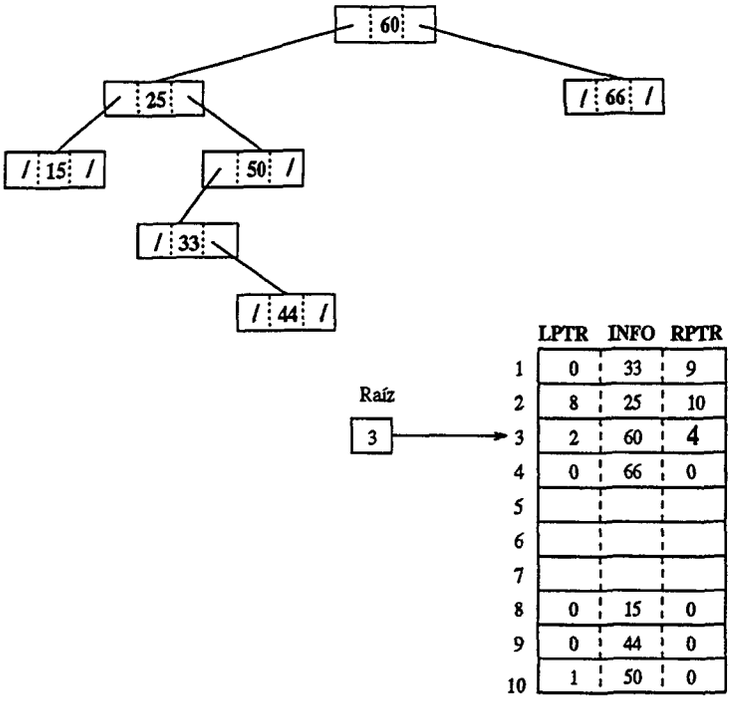
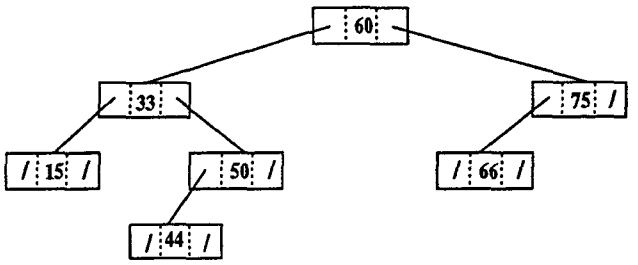


Figura 3.25: Árbol binario después de eliminar el nodo 75

```
tipo
   TipoPuntero=puntero a TipoNodo
   registro TipoNodo
      caracter INFO
      TipoPuntero LPTR,RPTR
   fin registro

Procedimiento CasoA (Pos,Raiz,Pad)
{Este procedimiento elimina un nodo sin hijo o con un solo hijo de la posición Pos. El puntero
Pad da la posición del padre del nodo a eliminar, o si Pad=nulo es que el nodo a eliminar es
el nodo raíz. El puntero Hijo da la posición del único hijo del nodo a eliminar, o si Hijo=nulo
es que el nodo a eliminar no tiene hijos}
      {Pos,Raiz,Pad por referencia}
variables
   TipoPuntero Hijo
inicio
   {Inicializar Hijo}
   si (LPTR(Pos)=nulo) y (RPTR(Pos)=nulo) entonces
      Hijo:=nulo
   si no
      si LPTR(Pos)<>nulo entonces
         Hijo:=LPTR(Pos)
      si no
         Hijo:=RPTR(Pos)
      fin si
   fin si
```

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006



Raíz: 3

	LPTR	INFO	RPTR
1	8	33	9
2			
3	1	60	7
4	0	66	0
5			
6			
7	4	75	0
8	0	15	0
9	0	44	0
10	9	50	0

Representación enlazada en memoria

Figura 3.26: Árbol binario después de eliminar el nodo 25

```
   si Pad<>nulo entonces
      si Pos=LPTR(Pad) entonces
         LPTR(Pad):=Hijo
      si no
         RPTR(Pad):=Hijo
      fin si
   si no
      Raiz:=Hijo
   fin si
fin procedimiento
```

Procedimiento CasoB (Pos,Raiz,Pad)

{Este procedimiento eliminará un nodo con dos hijos de la posición *Pos*. El puntero *Pad* da la posición del padre del nodo a eliminar. Si *Pad*=nulo es que el nodo a eliminar es el nodo raíz. El puntero *Suc* da la poscición del sucesor inorden del nodo a eliminar, y *PadSuc* da la posición del padre del sucesor inorden}

{TipoPuntero Pos,Raiz,Pad por referencia}

```
variables
      TipoPuntero P,Salva,Suc,PadSuc
inicio
   {Búsqueda de Pad y PadSuc}
   P:=RPTR(Pos)
   Salva:=Pos
```

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
   mientras LPTR(P)<>nulo hacer
      Salva:=P
      P:=LPTR(P)
   fin mientras
   Suc:=P
   PadSuc:=Salva
   {Eliminar al sucesor inorden mediante el procedimiento CasoA}
   CasoA(Suc,PadSuc)
   {Reemplazar el nodo eliminado por su sucesor inorden}
   si Pad<>nulo entonces
      si Pos=LPTR(Pad) entonces
         LPTR(Pad):=Suc
      si no
         RPTR(Pad):=Suc
      fin si
   si no
      Raiz:=Suc
   fin si
   LPTR(Suc):=LPTR(Pos)
   RPTR(Suc):=RPTR(Pos)
fin procedimiento
```

A continuación se presenta un procedimiento que dada la raíz de un árbol binario en memoria, *Raiz* y un elemento a borrar, *Item*, el procedimiento busca dicho elemento en el árbol (mediante el procedimiento de búsqueda binaria visto anteriormente). Si encuentra el elemento, *Item*, lo borra haciendo uso de los procedimientos de borrado anteriores y, posteriormente liberando memoria.

```
Procedimiento Eliminar (Raiz,Item)
   {TipoPuntero Raiz por referencia}
   {caracter Item por valor}
variables
   TipoPuntero Pos,Pad
inicio
   {Encontrar la posición de Item en el árbol}
   BusquedaBinaria(Raiz,Item,Pos,Pad)
   {Se mira si Item está en el árbol}
   si Pos=nulo entonces
      escribir ('El elemento no está en el árbol')
   si no
      si (RPTR(Pos)<>nulo) y (LPTR(Pos)<>nulo) entonces
         CasoB(Pos,Raiz,Pad)
      si no
         CasoA(Pos,Raiz,Pad)
      fin si
      liberar_memoria(Pos)
   fin si
fin procedimiento
```

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

3.8 PROBLEMAS

1. Realizar una función recursiva que permita realizar la copia de un árbol.

```
tipo
  TipoPuntero=puntero a TipoNodo
  registro TipoNodo
    caracter INFO
    TipoPuntero LPTR,RPTR
  fin registro
Funcion Copia (T) devuelve TipoPuntero
    {TipoPuntero T por referencia}
variables
  TipoPuntero Nuevo,Disp
Inicio
  si T=nulo entonces
    Copia:=nulo
  si no
    asignar_memoria(Nuevo)
    INFO(Nuevo):=Info(T)
    LPTR(Nuevo):=Copia(LPTR(T))
    RPTR(Nuevo):=Copia(RPTR(T))
    Copia:=Nuevo
  fin si
fin funcion
```

2. Dada la dirección del nodo cabeza, *Cabeza*, de un árbol binario, se pide calcular los niveles de cada uno de los nodos, devolviendo la dirección y nivel de forma que posteriormente se pueda dar la dirección de un nodo y se devuelva su nivel.

   En el método empleado se crearán dos vectores paralelos: *Nodos* y *Niveles*, los cuales contendrán la dirección y nivel, respectivamente, de cada uno de los nodos del árbol binario.

```
constantes
  Max=50 {máximo tamaño de vectores}
tipo
  TipoPuntero=puntero a TipoNodo
  registro TipoNodo
    caracter INFO
    TipoPuntero LPTR,RPTR
  fin registro
  Vec_P=vector [1..Max] de TipoPuntero
  Vec_Niv=vector [1..Max] de enteros
```

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
Procedimiento Nodo_Nivel(P,Nivel,indice,Nodos,Niveles)
      {TipoPuntero P por referencia}
      {enteras Nivel por valor}
      {enteras indice por valor}
      {Vec_P Nodos por refrencia}
      {Vec_Niv Niveles por referencia}
inicio
   si P<>nulo entonces
      indice:=indice+1
      Nodos[indice]:=P
      Niveles[indice]:=Nivel
      Nodo_Nivel(LPTR(P),Nivel+1,indice,Nodos,Niveles)
      Nodo_Nivel(RPTR(P),Nivel+1,indice,Nodos,Niveles)
   fin si
fin procedimiento
```

En el programa principal se realizarán las siguientes inicializaciones:

```
indice:=1
Nivel:=1
P:=LPTR(Head)
```

siendo *Cabeza*, la cabeza del árbol cuyo LPTR apunta al nodo raíz del árbol, P.

3. Partiendo de la imagen especular (ver figura 3.27) de un árbol binario ordenado se pide desarrollar un algoritmo recursivo que lo reordene de tal forma que en el árbol obtenido se verifique que para cada tripleta (padre-hijo izquierdo-hijo derecho) el campo de información del padre es mayor que el de cualquiera de sus hijos. La dirección de la raíz de la imagen especular viene dada por el puntero T.

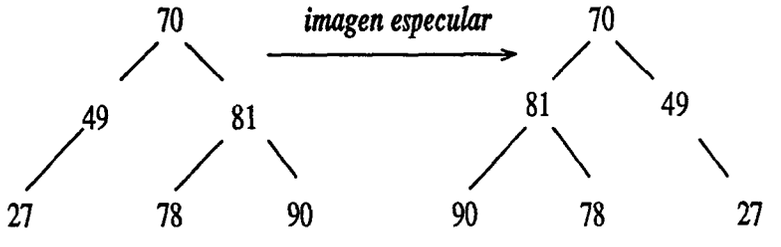


Figura 3.27: Ejemplo de imagen especular de un árbol binario

Método a emplear:

(a) Almacenamos en un vector, *Inorden*, el recorrido inorden de la imagen especular del árbol binario.

(b) Se recorre el vector y se van intercambiando los campos INFO del árbol binario con los elementos del vector.

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
Algoritmo Imagen_especular
constantes
   M=50 {máximo tamaño de la pila}
tipo
   TipoPuntero=puntero a TipoNodo
   registro TipoNodo
      enteras INFO
      TipoPuntero LPTR,RPTR
   fin registro
   Vec_enteros=vector [1..M] de enteros

Procedimiento Recorrer(T,Inorden,indice)
      {Vec_enteros Inorden por referencia}
      {TipoPuntero T por referencia}
      {enteras indice por valor}
inicio
   si T<>nulo entonces
      Recorrer(LPTR(T),Inorden,indice)
      Inorden[indice]:=INFO(T)
      indice:=indice+1
      Recorrer(RPTR(T),Inorden,indice)
   fin si
fin procedimiento

Procedimiento Intercambio(T,Inorden,indice)
      {TipoPuntero T por referencia}
      {Vec_enteros Inorden por referencia}
      {enteras indice por referencia}
variables
   enteras aux
inicio
   si T<>nulo entonces
      aux:=Inorden[indice]
      Inorden[indice]:=INFO(T)
      INFO(T):=aux
      indice:=indice+1
      Intercambio(LPTR(T),Inorden,indice)
      Intercambio(RPTR(T),Inorden,indice)
   fin si
fin procedimiento

variables
   enteras indice
   TipoPuntero T
   Vec_enteros Inorden
inicio
   si T<>nulo entonces
      indice:=1
```

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
        Recorrer(T,Inorden,indice)
        indice:=1
        Intercambio(T,Inorden,indice)
    fin si
fin
```

3.9 BIBLIOGRAFÍA

[DALE89] NELL DALE, SUSAN C. LILLY, *Pascal y estructuras de datos.* McGraw Hill, 1989.

[KNUTH86] DONALD E. KNUTH, *El arte de programar ordenadores. Volumen I. Algoritmos fundamentales.* Reverté, 1986.

[LIPSHUTZ87] SEYMOUR LIPSHUTZ, *Estructuras de datos.* McGraw Hill, 1987.

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

Capítulo 4

GRAFOS

4.1 INTRODUCCIÓN

En este capítulo se profundiza en la estructura de datos llamada *grafo*. Aunque la terminología usada en la teoría de grafos no es del todo uniforme, trataremos de presentar los conceptos y terminología básicas de dicha teoría.

4.1.1 CONCEPTO DE GRAFO

Informalmente, un grafo es un conjunto de puntos y un conjunto de flechas, cada flecha uniendo dos puntos. Los puntos se llaman *vértices* o *nodos* del grafo, y las flechas se llaman *arcos* del grafo.

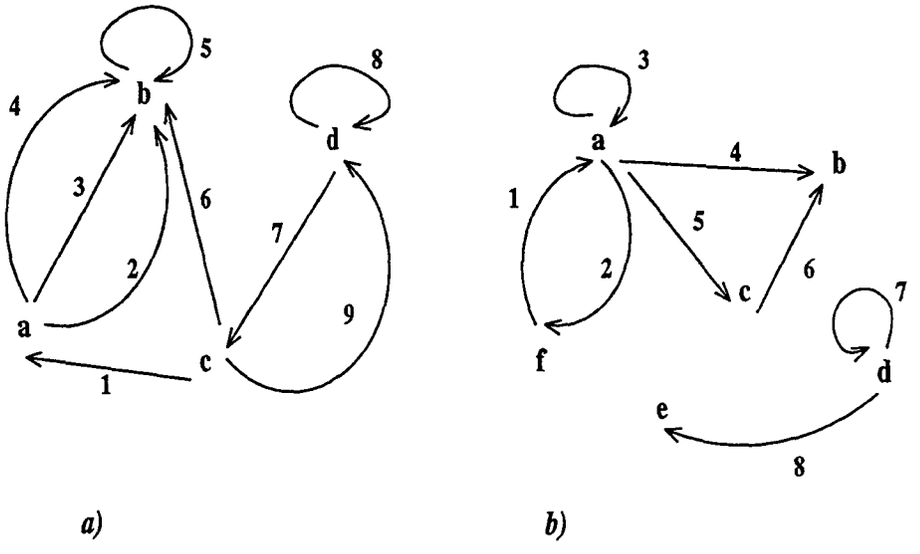


Figura 4.1: a) 3-Grafo y b) 1-grafo

Formalmente, un *grafo G(V,E)* es una estructura formada por:

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

- un conjunto de *vértices* $V=\{v_1,v_2,...,v_n\}$ y
- un conjunto de *arcos* $E=\{e_1,e_2,...e_m\}$. Cada arco puede aparecer más de una vez en el conjunto E. Un grafo en el que un arco no puede aparecer más de una vez en dicho conjunto se llama *1-grafo*. Generalizando, un grafo en el que un arco no puede aparecer más de p veces se llama *p-grafo*.

El número de vértices de un grafo se llama *orden* del grafo. Por ejemplo, en el grafo de la figura 4.1 a):

$$V=\{a,b,c,d\},\ E=\{1,2,3,4,5,6,7,8,9\}$$

donde, por ejemplo, decimos que el orden del grafo es 4 porque V tiene 4 elementos.

Un arco que empieza y termina en el mismo vértice se llama *bucle* (ver arco 5 en figura 4.1 a)). Para un arco, el vértice del que sale el arco se llama *extremo inicial*, y al que llega se llama *extremo final*. El arco 3 en la figura 4.1 a) tiene como extremo inicial al vértice a y como extremo final al vértice b. Un vértice y se llama *sucesor* de un vértice x si existe un arco con x como extremo inicial y con y como extremo final. Por ejemplo, en la figura 4.1 a), el vértice b es sucesor del vértice a, puesto que existe un arco, el 3, de a a b. De forma similar, un vértice y se llama *predecesor* de un vértice x si hay un arco de y a x. En el ejemplo anterior, a es predecesor de b en el arco 3. El conjunto de *vecinos* de un vértice x es la suma de todos los sucesores y predecesores de x. En la figura 4.1 a) los vecinos del vértice a son los vértices b y c. Si un vértice no tiene vecinos se llama *vértice aislado*. En las figuras 4.1 a) y b) no existe ningún vértice aislado.

En un grafo $G=(V,E)$, donde cada arco entre dos vértices, x e y, es una línea continua sin dirección (es decir, no es una flecha), se llama *grafo no dirigido* (el grafo considerado hasta ahora, se llama *grafo dirigido* o *digrafo*). En un grafo no dirigido, las líneas que unen los vértices se llaman *aristas*. Por ejemplo, en la figura 4.2 tenemos un ejemplo de grafo no dirigido.

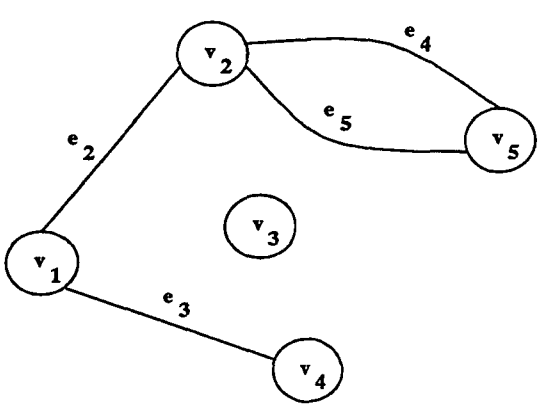
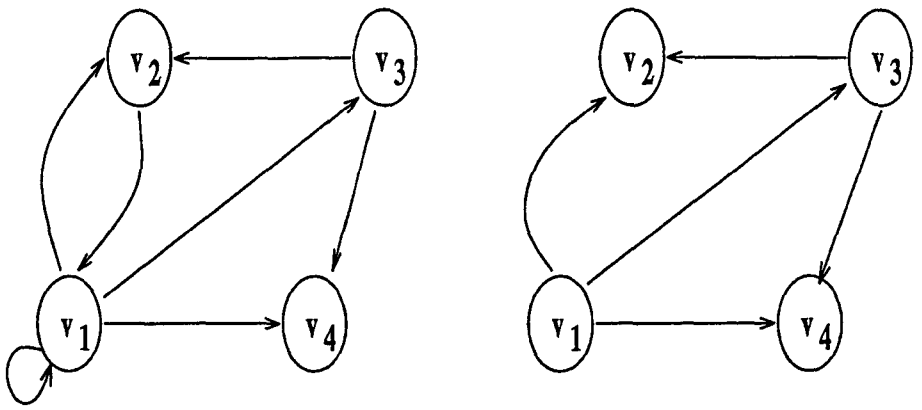


Figura 4.2: Grafo no dirigido

Un grafo dirigido se dice que es *simple* (figura 4.3) si no tiene bucles ni puede haber arcos con un mismo vértice en los dos extremos (*arcos paralelos*). El concepto se puede aplicar también a grafos no dirigidos.

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006



a) *b)*

Figura 4.3: a) Grafo dirigido y b) grafo dirigido simple

Un *grafo* dirigido se llama *etiquetado* si sus arcos tienen pesos. Un arco se dice que tiene asignado un peso si cada arco posee un valor numérico no negativo. La definición también es válida para grafos no dirigidos.

4.1.2 DEFINICIONES BÁSICAS

A continuación comentaremos algunas definiciones propias de la teoría de grafos:

Arcos adyacentes, aristas adyacentes. Dos arcos (o aristas) se llaman adyacentes si son distintos/as entre sí y al menos tienen un extremo en común. Por ejemplo, para el grafo de la figura 4.2, e_2 y e_3 son aristas adyacentes.

Vértices adyacentes. Dos vértices son adyacentes si son distintos y existe un arco que va de uno a otro. En la figura 4.3, v_2 y v_3 son vértices adyacentes.

Un grafo, G, es *completo*, si cada vértice de G es adyacente a todos los demás vértices de G. Un grafo completo de n vértices tendrá $n \times (n-1)/2$ arcos.

Arco incidente a un vértice. Si un vértice x es el extremo inicial de un arco u, el cual no es un bucle, se dice que el arco u es incidente hacia el exterior al vértice x. Se dice que un arco u es incidente hacia el interior a un vértice y si x es el extremo final del arco u.

El *grado* de un vértice v, *grad(v)*, es el número de arcos con v como extremo; cada bucle se cuenta dos veces (ver figura 4.4). Aquellos grafos en los que cada vértice tiene el mismo grado se llaman *grafos regulares* . En un grafo no dirigido como cada arista incide en dos vértices distintos (o dos veces en un mismo vértice) se verifica la siguiente relación:

$$\sum_{i=1}^{n} grad(v_i) = 2 \times q$$

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

siendo q el número de aristas y n el número de vértices.

En un grafo dirigido, el *grado de salida* de un vértice u de G, es el número de arcos que empiezan en u. Similarmente, el *grado de entrada* de u es el número de arcos que terminan en u. Un vértice u se llama *fuente* si tiene un grado de salida positivo y un grado de entrada nulo. De igual forma, u se llama *sumidero* si tiene un grado de salida nulo y un grado de entrada positivo. Para un grafo dirigido se verifica la relación:

$$\sum_{i=1}^{n} grad_{in}(v_i) = \sum_{i=1}^{n} grad_{out}(v_i) = q$$

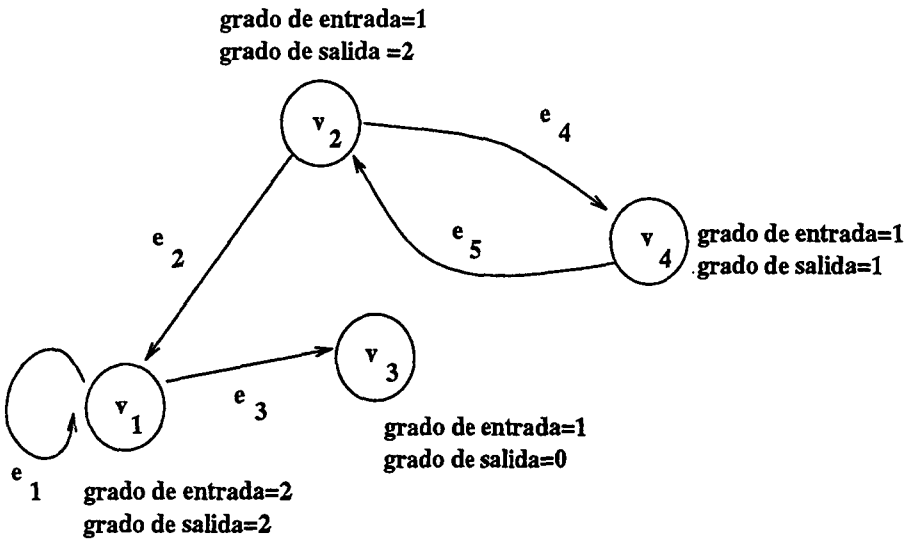


Figura 4.4: Grafo dirigido

Grafo simétrico (ver figura 4.5 a)). Un grafo se dice que es simétrico cuando para todo arco u que va del vértice x al vértice y, existe otro arco v que va del vértice y al x.

Grafo antisimétrico (ver figura 4.5 b)). Un grafo es antisimétrico cuando para cada arco u que va del vértice x al y no existe un arco v que va del vértice y al x.

Un *grafo orientado* es un grafo dirigido en el que no existen arcos simétricos.

Un grafo G'(V',E') se llama *subgrafo* de un grafo G(V,E) si V'$\subseteq$V y E'$\subseteq$E.

4.1.3 REPRESENTACIÓN DE GRAFOS: MATRICES Y LISTAS

Existen dos formas estándar de representar un grafo en memoria:

- Una representación secuencial mediante una matriz de adyacencia
- Una representación enlazada mediante listas

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

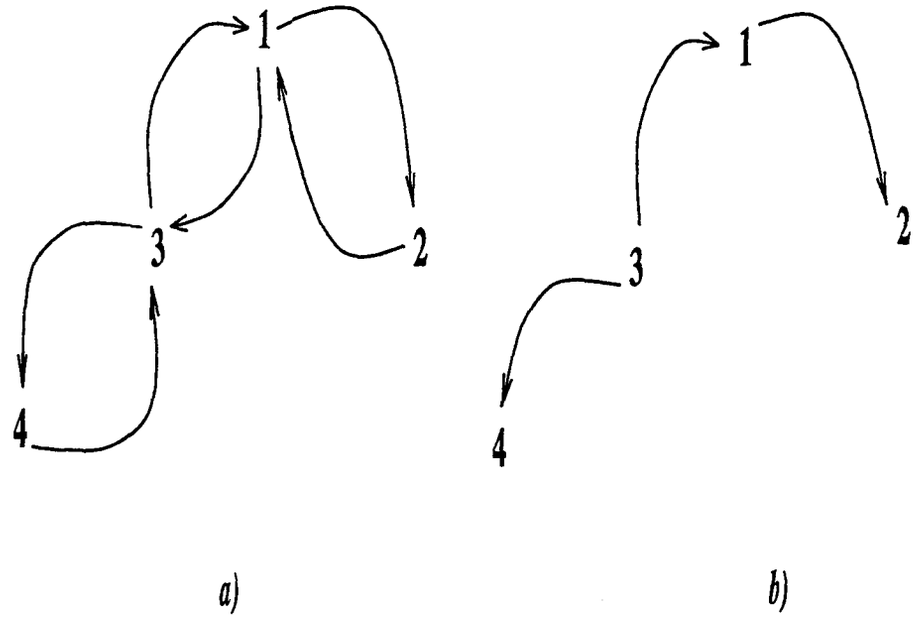


Figura 4.5: a) Grafo simétrico, b) antisimétrico

4.1.3.1 MATRIZ DE ADYACENCIA

Un grafo de n vértices lo podemos representar en memoria mediante una matriz cuadrada, *A(a[i,j])*, de dimensión N×N, denominada *matriz de adyacencia*. En dicha matriz los arcos se podrán representar de la siguiente forma:

a[i,j]=1 si existe arco del vértice i al vértice j
a[i,j]=0 si no existe arco del vértice i al j

Los bucles del grafo, siguiendo la anterior notación, aparecerán en la diagonal principal de la matriz. La matriz de adyacencia de un grafo depende lógicamente de la ordenación de los nodos del grafo; es decir, ante diferentes ordenaciones de nodos se pueden obtener matrices de adyacencia diferentes.

Para el caso de grafos no dirigidos, la matriz de adyacencia será una matriz simétrica, debido a que cada arista se corresponde con dos arcos dirigidos. Por ejemplo, la arista del vértice i al j vendrá representada en la matriz de adyacencia por los elementos *a[i,j]* y *a[j,i]*.

Para los grafos etiquetados, la matriz de adyacencia se denomina *matriz de adyacencia etiquetada*. En ella, el elemento *a[i,j]* contendrá la etiqueta o peso asociado al arco que va del vértice i al vértice j.

El poder representar los grafos mediante matrices nos permite determinar de forma directa si existe un arco de un vértice a otro del grafo. Sin embargo, la representación matricial necesita mucha memoria para aquellos casos en que el número de nodos es muy grande o bien cuando hay pocos arcos.

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

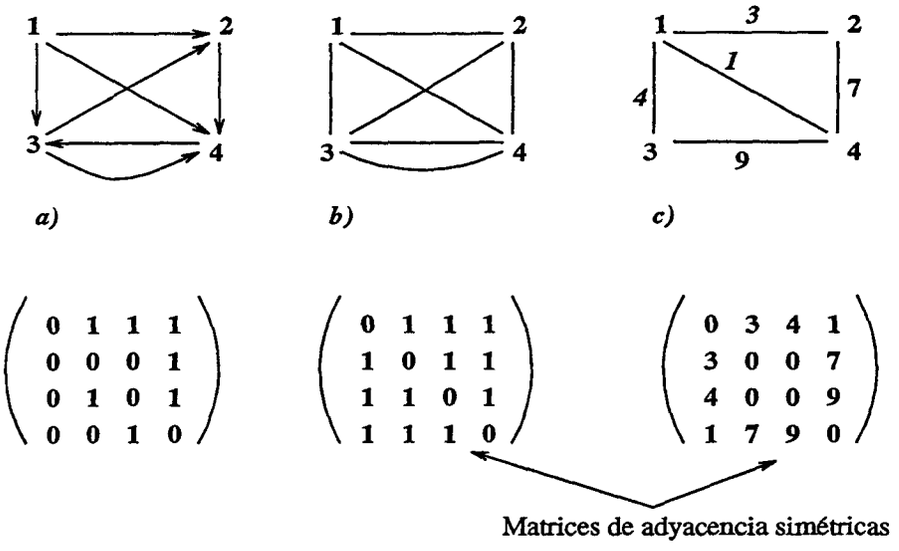


Figura 4.6: Matrices de adyacencia: a) Grafo dirigido, b) Grafo no dirigido y c) Grafo dirigido etiquetado

4.1.3.2 LISTAS DE ADYACENCIA

La representación secuencial de un grafo mediante una matriz de adyacencia presenta las siguientes desventajas:

- En primer lugar, las operaciones de inserción y borrado de nodos son difíciles de realizar. En tales operaciones el tamaño de la matriz de adyacencia cambiaría y los nodos tendrían que reordenarse.
- Además si el grafo tiene muchos vértices y pocos arcos, la matriz de adyacencia desperdicia memoria, ya que la mayoría de los elementos serían cero.

Para evitar los inconvenientes a que da lugar la representación secuencial de un grafo, se puede optar por la representación enlazada mediante listas (ver figura 4.7). En la figura 4.7 se muestra una lista de vértices y una lista de arcos:

- Cada elemento de la lista de vértices será un registro con los siguientes campos:

VERTICE	SIG	ADY

Donde *VERTICE* representa el nombre de cada vértice o nodo del grafo, *SIG* será un puntero al siguiente vértice de la lista, *ADY* será un puntero al primer elemento de la lista de arcos del vértice. Los vértices forman una lista enlazada por lo que existirá una variable puntero *PRINCIPIO* apuntando al comienzo de la lista. Muchas veces, dependiendo de la aplicación, los vértices pueden estar organizados mediante un vector ordenado o mediante un árbol binario de búsqueda en lugar de una lista enlazada.

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

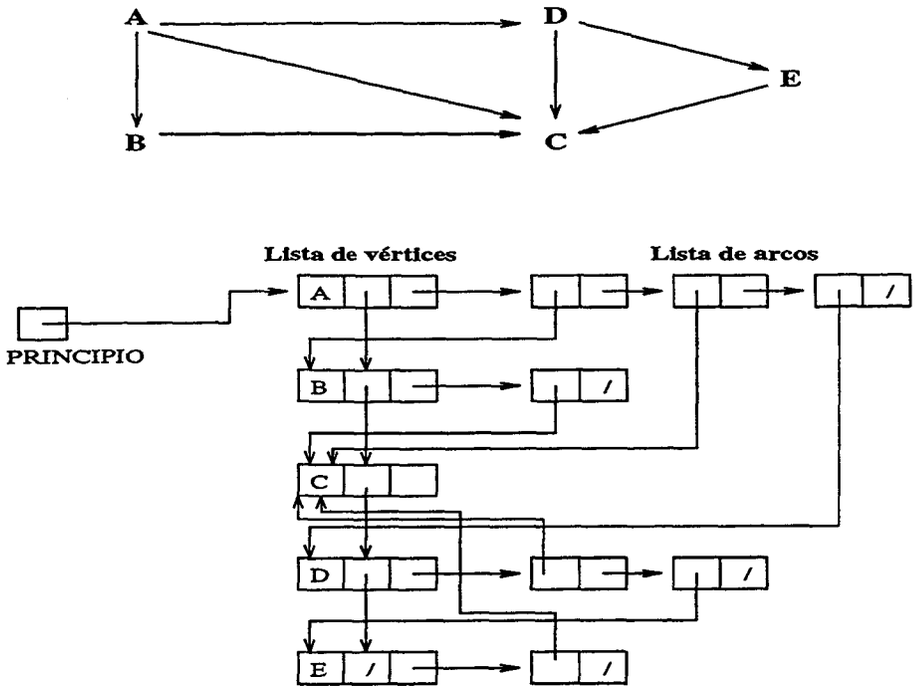


Figura 4.7: Representación enlazada de un grafo mediante listas

- Cada elemento de la lista de arcos o lista de adyacencia corresponde a un arco del grafo y será un registro con los siguientes campos:

DEST	ENL

El campo *DEST* indicará la posición del extremo final del arco en la lista de vértices. El campo *ENL* enlaza los arcos con el mismo vértice inicial, o sea, los vértices que tienen la misma lista de adyacencia.

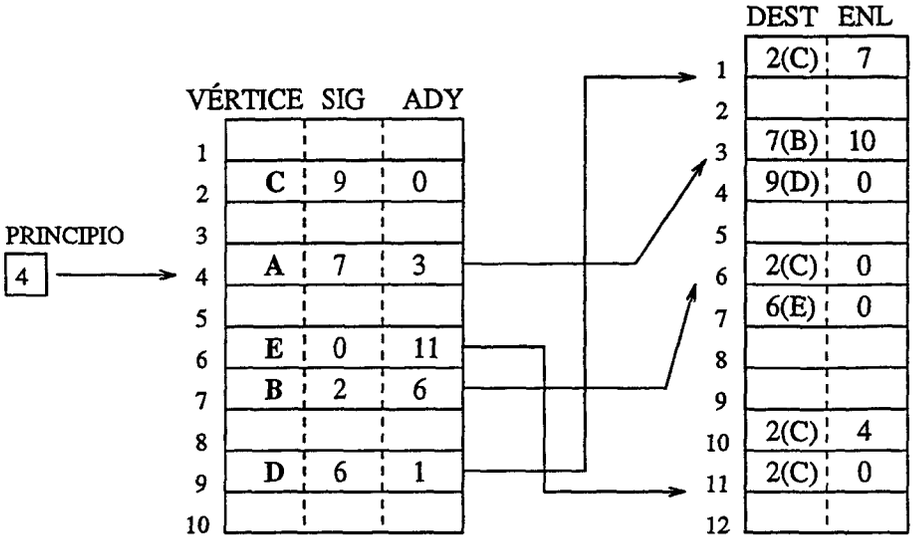


Figura 4.8: Representación enlazada de un grafo en memoria

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

4.2 GRAFOS DIRIGIDOS

4.2.1 CAMINOS

Un *camino* es una secuencia de arcos en los que el extremo final de cada arco es el extremo inicial del siguiente arco.

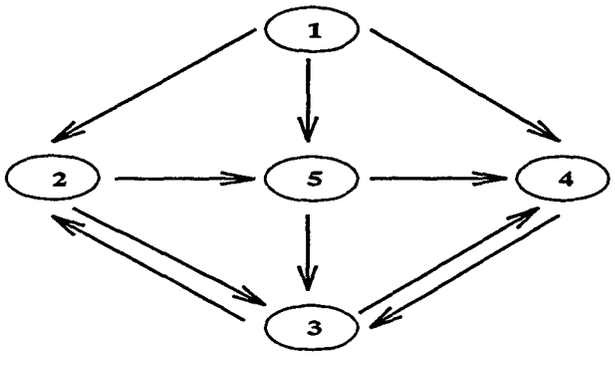
Un *camino simple* es el que no utiliza dos veces el mismo arco. Un camino no simple se denomina *camino compuesto*.

Un *camino elemental* es el que no utiliza dos veces el mismo vértice. El caso contrario se denomina *camino no elemental*.

4.2.1.1 BÚSQUEDA DE CAMINOS EN UN GRAFO

La búsqueda de caminos elementales en un grafo podría asemejarse a la búsqueda de la salida en un laberinto. En un laberinto, partimos de un origen y deseamos llegar a un destino. Muchas veces seguiremos caminos sin salida en los cuales tendremos que retroceder hasta un punto en que existan varios caminos alternativos. Una vez en dicho punto, tomaremos aquellos caminos en los que no hayamos comprobado si existe o no, salida.

La búsqueda de caminos elementales en un grafo intenta encontrar todos los caminos elementales posibles entre un extremo inicial y final. A diferencia de la búsqueda en un laberinto, una vez que se haya alcanzado el vértice final, se retrocede al vértice inmediatamente anterior al vértice final en el camino encontrado, para desde él, intentar encontrar otro camino elemental que nos lleve al vértice final. Para comprender el método a emplear vamos a buscar en el grafo de la figura 4.9 todos los caminos elementales entre el vértice 1 y el 4.



$$\text{Grafo}= \begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Figura 4.9: Grafo y su matriz de adyacencia

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

Como dijimos anteriormente, camino es una secuencia de arcos en el que el extremo final de cada arco es el extremo inicial del siguiente, por tanto, si partimos del vértice 1, veremos que existe un arco al vértice 2, de éste al vértice 3 y de éste al vértice final 4. De esta forma, el primer camino encontrado es 1-2-3-4, el cual almacenamos en la primera columna de una matriz que llamaremos *CAMINOS*.

$$CAMINOS = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}$$

A partir de este camino se intenta buscar otro, partiendo del vértice anterior al vértice final, el vértice 3. Para ello se copia la primera columna de la matriz *CAMINOS*, excepto el último elemento (el 4), en la segunda columna de dicha matriz.

$$CAMINOS = \begin{pmatrix} 1 & 1 \\ 2 & 2 \\ 3 & 3 \\ 4 & \end{pmatrix}$$

Se intenta a partir del vértice 3 buscar otro arco que nos lleve al destino. Al no haberlo se retrocede y se vuelve a realizar la búsqueda a partir del vértice anterior, el vértice 2.

$$CAMINOS = \begin{pmatrix} 1 & 1 \\ 2 & 2 \\ 3 & 0 \\ 4 & \end{pmatrix}$$

Para evitar que de nuevo el siguiente arco encontrado nos lleve al vértice 3, la búsqueda del siguiente arco (con extremo inicial en el vértice 2) comienza tomando como extremo final al vértice 4 (vértice siguiente al 3, del cual habíamos retrocedido). Como del vértice 2 no existe arco al vértice 4, se coge el vértice siguiente a éste último (vértice 5) como extremo final del arco. Del vértice 2 al 5 si existe arco, por tanto, ése es el siguiente arco y de esa forma obtendremos el camino elemental 1-2-5-3-4 que almacenamos en la matriz *CAMINOS*.

$$CAMINOS = \begin{pmatrix} 1 & 1 \\ 2 & 2 \\ 3 & 5 \\ 4 & 3 \\ & 4 \end{pmatrix}$$

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

Partiendo del último camino encontrado se intenta buscar otro, del mismo modo que actuamos anteriormente: Se copian todos los vértices del camino anterior, excepto el vértice final, en la siguiente columna de la matriz *CAMINOS*

$$CAMINOS = \begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 5 & 5 \\ 4 & 3 & 3 \\ & 4 & \end{pmatrix}$$

Como desde el vértice 3 no existe ningún camino diferente que nos lleve al vértice final, se retrocede en el último camino encontrado, buscándose un camino que partiendo del vértice 5 nos lleve al vértice final. Para evitar repetir caminos en la matriz *CAMINOS* la búsqueda del siguiente arco tomará como extremo final el 4 (vértice siguiente al 3, del cual habíamos retrocedido).

$$CAMINOS = \begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 5 & 5 \\ 4 & 3 & 0 \\ & 4 & \end{pmatrix}$$

Esta vez existe arco del vértice 5 al vértice final 4, por lo que hemos encontrado un tercer camino elemental.

$$CAMINOS = \begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 5 & 5 \\ 4 & 3 & 4 \\ & 4 & \end{pmatrix}$$

El proceso se repite hasta llegar a la siguiente matriz *CAMINOS*, la cual contiene todos los caminos posibles entre el vértice 1 y el 4:

$$CAMINOS = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 4 & 5 & 5 \\ 3 & 5 & 5 & & 3 & 4 \\ 4 & 3 & 4 & & 4 & \\ & 4 & & & & \end{pmatrix}$$

Para llevar a cabo el algoritmo tendremos en cuenta los siguientes pasos:

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

- Necesitaremos un vector que nos indique la longitud de cada camino, *LongC*. Cada camino se irá almacenando de la forma mostrada anteriormente en una matriz que llamaremos *Caminos*. Inicialmente, se almacena el vértice de partida en la matriz de caminos, por lo que la longitud del primer camino será 1 y ello se indicará en el vector *LongC*. El resto del vector *LongC* y de la matriz *Caminos* estarán a 0.

- Se ha de determinar el arco por el que se reemprende la búsqueda de caminos. Este arco, llamésmole *arco*, será uno para la primera búsqueda a partir de un vértice determinado. En caso de estar retrocediendo, *arco*, será el vértice desde el que se retrocede incrementado en uno. Con *arco* se evita que al retroceder en un camino para obtener el siguiente volvamos a obtener el mismo camino.

- Cada vez que se encuentre un arco, el extremo final de ese arco se convierte en extremo inicial del nuevo arco a buscar. La búsqueda del nuevo arco se llevará a cabo tomando como vértice final el indicado por *arco* para evitar repeticiones.

- Para cada arco encontrado se ha de comprobar que su vértice final no está ya en el camino, puesto que estamos buscando caminos elementales, en los cuales no se pueden repetir los vértices.

- Cuando el arco encontrado tenga como extremo final el vértice final del camino, se llevará a cabo la copia del camino a la siguiente columna de la matriz *Caminos*, exceptuando el último vértice de dicho camino.

A continuación se presenta un algoritmo que encuentra como máximo los M caminos elementales de un grafo con *numvert* vértices. La matriz de adyacencia del grafo se encuentra en *Grafo*. Supondremos que las longitudes de los caminos elementales nunca será superior a N.

```
Algoritmo Caminos_elementales
constantes
   M=10 {indica máximo número de caminos posibles}
   N=5 {indica longitud máxima para un camino}
   numvert=5 {número de vértices del grafo}
tipo
   vector_enteros=vector [1..M] de enteros
   matriz_caminos=matriz [1..N,1..M] de enteros
   matriz_adyacencia=matriz [1..numvert,1..numvert] de enteros

Funcion vert_en_camino(vertice,Caminos,Longcam,num) devuelve logico
      {enteras vertice,Longcam,num por valor}
      {matriz_caminos Caminos por referencia}
variables
   enteras k
   logicas encontrado
inicio
{Esta función devuelve verdadero si vertice ya está en el camino indicado por num y de longitud Longcam}
   k:=1
```

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
   encontrado:=falso
   mientras (k<=Longcam) y (no encontrado) hacer
      si vertice=Caminos[k,num] entonces
         encontrado:=verdadero
      fin si
      k:=k+1
   fin mientras
   vert_en_camino:=encontrado
fin funcion


Procedimiento Copia_Columna(Caminos,num,LongC)
      {matriz_caminos Caminos por referencia}
      {enteras num por valor}
      {vector_enteros LongC por referencia}
variables
   enteras k
inicio
{Este procedimiento copia el último camino encontrado en la siguiente columna de la matriz
Caminos}
   LongC[num]:=LongC[num-1]-1
   desde k:=1 hasta LongC[num] hacer
      Caminos[k,num]:=Caminos[k,num-1]
   fin desde
fin procedimiento


Procedimiento Comprobar_vfinal(v,vfinal,num,Caminos,LongC,vert)
      {enteras v,vfinal por valor}
      {enteras num,vert por referencia}
      {matriz_caminos Caminos por referencia}
      {vector_enteros LongC por referencia}
inicio
{Procedimiento que comprueba si v es el vértice final, vfinal. En tal caso se incrementa el
número de caminos, num, y se copia el camino encontrado en la siguiente columna de la
matriz Caminos. Si v no es vértice final lo toma como extremo inicial del nuevo arco del
camino}
   si v=vfinal entonces
      num:=num+1
      Copia_Columna(Caminos,num,LongC)
   si no
      vert:=v
   fin si
fin procedimiento
```

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
Procedimiento Buscar_Caminos(Grafo,Caminos,LongC,vinicial,vfinal,numcaminos)
      {matriz_adyacencia Grafo por referencia}
      {matriz_caminos Caminos por referencia}
      {vector_enteros LongC por referencia}
      {enteras vinicial,vfinal,numcaminos por valor}
variables
   enteras vertice,arco,j
inicio
   mientras LongC[numcaminos]>0 hacer
      vertice:=Caminos[LongC[numcaminos],numcaminos]
      arco:=Caminos[LongC[numcaminos]+1,numcaminos]+1
      Caminos[LongC[numcaminos]+1,numcaminos]:=0
      si arco<=numvert entonces
         desde j:=arco hasta numvert hacer
            si Grafo[vertice,j]<>0 entonces
               si no vert_en_camino(j,Caminos,LongC[numcaminos],numcaminos) entonces
                  LongC[numcaminos]:=LongC[numcaminos]+1
                  Caminos[LongC[numcaminos],numcaminos]:=j
                  Comprobar_vfinal(j,vfinal,numcaminos,Caminos,LongC,vertice)
               fin si
            fin si
         fin desde
      fin si
      LongC[numcaminos]:=LongC[numcaminos]-1
      si LongC[numcaminos=0 entonces
         numcaminos:=numcaminos+1
      fin si
   fin mientras
fin procedimiento

variables
   matriz_adyacencia Grafo
   matriz_caminos Caminos
   vector_enteros LongC
   enteras vinicial,vfinal,numcaminos
inicio {Programa principal}
   Leer_matriz_adyacencia(Grafo) {Lectura de matriz de adyacencia del grafo}
   Inicializar_Longc(LongC) {Incialización de LongC a 0}
   Inicializar_Caminos(Caminos) {Inicializar Caminos a 0}
   leer(vinicial)
   leer(vfinal)
   numcaminos:=1
   LongC[numcaminos]:=1
   Caminos[LongC[numcaminos],numcaminos]:=vinicial
   Buscar_caminos(Grafo,Caminos,LongC,vinicial,vfinal,numcaminos)
   Escribir_Caminos(Caminos) {Escritura de Caminos}
fin
```

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

4.2.1.2 CAMINO MÍNIMO DESDE UN VÉRTICE AL RESTO DE LOS VÉRTICES

A continuación se plantea un problema bastante común en la teoría de grafos. Consideremos un grafo etiquetado en el que las etiquetas de cada uno de sus arcos representa el coste de ir de un extremo a otro. Trataremos de encontrar el coste mínimo de ir de una fuente o vértice dado a cada uno de los vértices del grafo. El algoritmo empleado se denomina algoritmo de *Dijkstra*.

El algoritmo obtendrá un conjunto de vértices que tienen distancia mínima a una fuente. Inicialmente ese conjunto sólo contiene al vértice fuente. En cada paso se añade el vértice con distancia más corta al conjunto. Para cada vértice añadido a ese conjunto se calcula la distancia mínima de ese vértice a los restantes del grafo que no formen parte del conjunto. Las distancias mínimas se mantienen en un vector de distancias, en el que cada posición está asociada a un vértice del grafo. Al final, este vector nos indicará el coste mínimo de la fuente inicial a cada uno de los vértices del grafo.

Supongamos que los vértices del grafo son ciudades y el coste de los arcos representa el precio de viajar de una ciudad a otra. Para el grafo etiquetado de la figura 4.10 tendremos la matriz de adyacencia etiquetada *coste*. Cada elemento de dicha matriz representa el coste de viajar de una ciudad a otra (las ciudades son los vértices del grafo, cuya numeración se indica en la figura 4.10). Cuando no exista vuelo directo entre un par de ciudades del grafo se supone el coste de viajar de una a otra a infinito (arcos infinitos en matriz *coste*).

$$coste = \begin{pmatrix} \infty & 20 & \infty & 60 & 200 \\ \infty & \infty & 100 & \infty & 40 \\ \infty & \infty & \infty & \infty & 20 \\ \infty & \infty & 40 & \infty & 120 \\ \infty & \infty & \infty & \infty & \infty \end{pmatrix}$$

Tomamos como fuente de partida la ciudad de Las Palmas G.C. y mediante el algoritmo de *Dijkstra* calculamos cual es el coste mínimo de viajar desde Las Palmas G.C. a cualquiera de las otras ciudades del grafo.

Los pasos a seguir son los siguientes:

- Inicialmente tendremos un vector *distancia* (construido a partir del grafo de la figura 4.10) que contiene los precios desde la fuente (Las Palmas G.C., en nuestro caso) al resto de las ciudades del grafo. Para aquellas ciudades a las que no lleguen vuelos directos desde la fuente, la distancia inicial se supone a infinito. Para nuestro ejemplo el vector *distancia* será:

(2) Madrid	(3) Nueva York	(4) Roma	(5) Londres
20	∞	60	200

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

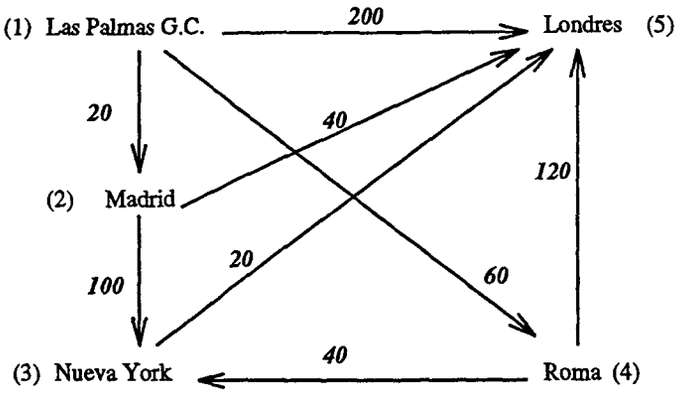


Figura 4.10: Grafo de ciudades y costos de vuelos

- La siguiente fuente a añadir al conjunto de fuentes será aquella que tenga distancia más corta. En nuestro ejemplo, Madrid es la ciudad de *distancia* más cercana a Las Palmas G.C., con distancia=20. Por tanto Madrid se añade al conjunto de fuentes: {Las Palmas G.C., Madrid}. A continuación se modificarán las distancias de aquellas ciudades que no pertenezcan al conjunto de fuentes de la siguiente forma:

Nueva York	min[distancia (Las Palmas G.C.-Nueva York), distancia(Las Palmas G.C.-Madrid)+coste de Madrid-Nueva York]= min[∞,20+100]=120
Roma	min[distancia (Las Palmas G.C.-Roma), distancia(Las Palmas G.C.-Madrid)+coste de Madrid-Roma]= min[60,20+∞]=60
Londres	min[distancia (Las Palmas G.C.-Londres), distancia(Las Palmas G.C.-Madrid)+coste de Madrid-Londres]= min[200,20+40]=60

El nuevo vector *distancia* será:

(2) Madrid	(3) Roma	(4) Nueva York	(5) Londres
20	60	120	60

- A continuación se determina, entre las ciudades que no pertenecen al conjunto, aquella de distancia más corta. Esa ciudad, Roma en nuestro ejemplo, se añade al conjunto: {Las Palmas G.C., Madrid, Roma}. Al igual que en el caso anterior se comprobará si se ha de modificar el vector de distancias:

Nueva York	min[distancia (Las Palmas-Nueva York), distancia(Las Palmas-Roma)+coste de Roma-Nueva York]= min[120,60+40]=100
Londres	min [distancia (Las Palmas-Londres), distancia(Las Palmas-Roma)+coste de Roma-Londres]= min[60,60+120]=60

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

Para este caso el vector *distancia* quedará como:

(2) Madrid	(3) Roma	(4) Nueva York	(5) Londres
20	60	100	60

- La siguiente distancia más corta, de entre las ciudades que no pertenecen al conjunto, es la correspondiente a Londres, la cual se añade al conjunto:{Las Palmas G.C., Madrid, Roma,Londres}. Se analiza de nuevo el vector de distancias:

Nueva York	min[distancia (Las Palmas G.C.-Nueva York), distancia(Las Palmas G.C.-Londres)+coste de Londres-Nueva York]= min[100,60+∞]=100

En este caso el vector *distancia* no se altera:

(2) Madrid	(3) Roma	(4) Nueva York	(5) Londres
20	60	100	60

- La siguiente ciudad a tratar es Nueva York, la cual se añade al conjunto: {Las Palmas G.C.,Madrid,Roma,Londres,Nueva York}. El vector distancias no se altera ya que todas las ciudades pertenecen al conjunto.

El vector de distancias resultante es:

(2) Madrid	(3) Roma	(4) Nueva York	(5) Londres
20	60	100	60

lo cual nos indica que el coste mínimo desde Las Palmas G.C. a Madrid es 20 (si nos fijamos en el grafo de la figura 4.10 es un vuelo directo), desde Las Palmas G.C. a Roma el coste mínimo es de 60 (vuelo directo), de Las Palmas G.C. a Nueva York es de 80 (Las Palmas G.C.-Madrid-Londres) y de Las Palmas a Londres es de 60 (Las Palmas G.C.-Madrid-Londres).

A continuación presentamos el algoritmo de *Dijkstra* para un grafo de *numvert* vértices cuya matriz de adyacencia viene dada por *Grafo*.

```
Algoritmo Dijkstra
constantes
   {número de vértices del grafo}
   numvert=5
tipo
   matriz_enteros=matriz [1..numvert,1..numvert] de enteros
   vector_enteros=vector [1..numvert-1] de enteros
   conjunto_enteros=conjunto de enteros
```

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
Procedimiento calcular_minimo_distancia(D,F,min_dist,vert)
      {vector_enteros D por valor}
      {conjunto_enteros F por valor}
      {enteras min_dist,vert por referencia}
variables
   enteras i
inicio
   desde i:=1 hasta numvert hacer
      si (i no pertenece F) y (D[i]<min_dist) entonces
            min_dist:=D[i]
            vert:=i
      fin si
   fin desde
fin procedimiento

Funcion minimo(a,b) devuelve entero
      {enteras a,b por valor}
inicio
   {Función que calcula el mínimo entre dos enteros a y b}
   si a<b entonces
      minimo:=a
   si no
      minimo:=b
   fin si
fin funcion

Procedimiento DIJKSTRA (Grafo,Fuentes,distancia)
      {matriz_enteros Grafo por referencia}
      {conjunto_enteros Fuentes por referencia}
      {vector_enteros distancia por referencia}
variables
   enteras i,j,vert,min_dist
inicio
   desde j:=1 hasta numvert-1 hacer
      min_dist:=∞
      calcular_minimo_distancia(distancia,Fuentes,min_dist,vert)
      Fuentes:=Fuentes+[vert]
      desde i:=1 hasta numvert hacer
         si i no pertenece Fuentes entonces
            distancia[i]:=minimo (distancia[i],distancia[vert]+Grafo[vert,i])
         fin si
      fin desde
   fin desde
fin procedimiento
```

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
variables
   vector_enteros distancia
   matriz_enteros Grafo
   conjunto_enteros Fuentes
   enteras i,j
inicio
   {Lectura de la matriz de adyacencia del grafo:}
   Leer_matriz_adyacencia(Grafo)
   {Lectura de la fuente de partida:}
   leer(j)
   Fuentes:=[j]
   desde i:=1 hasta numvert hacer
      distancia[i]:=Grafo[j,i]
   fin desde
   DIJKSTRA(Grafo,Fuentes,distancia)
   desde i:=1 hasta numvert hacer
      si i<>j entonces
         escribir(' Distancia mínima del vértice ',j,' al ',i,' :',distancia[i])
      fin si
   fin desde
fin
```

4.2.1.3 CAMINO MÍNIMO ENTRE CUALQUIER PAREJA DE VÉRTICES

Consideremos un grafo orientado etiquetado en el que sus vértices son ciudades y las etiquetas de sus arcos representan el coste de viajar de una ciudad a otra. Partiendo de dicho grafo sería deseable poder determinar una tabla con los caminos más baratos para viajar entre cualquier par de ciudades. Este problema podría resolverse aplicando el algoritmo de *Dijkstra* tantas veces como vértices tuviese el grafo. El algoritmo usaría como fuente un vértice diferente del grafo cada vez. Sin embargo, existe una forma más simple de obtener el camino mínimo entre cada par de vértices de un grafo llamada algoritmo de *Floyd*.

El algoritmo de *Floyd* parte de la matriz de adyacencia del grafo y obtiene una matriz en la que cada elemento *a[i,j]* representa el coste mínimo para ir del vértice i al vértice j. Inicialmente se establece que:

a[i,j]=coste[i,j]	si existe arco de i a j, con $i<>j$
a[i,j]=∞	si no existe arco de i a j
a[i,i]=0	los bucles del grafo se ponen a cero

Sobre la matriz A se llevan a cabo n iteraciones, siendo n el número de vértices del grafo. En cada iteración, k, cada elemento *a[i,j]* quedará con el mínimo coste posible para ir del vértice i al j pasando por vértices con numeración menor que k. Para conseguir esto, si en cada iteración k se cumple que:

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

$$a[i,k]+a[k,j]<a[i,j]$$

significará que el camino para ir del vértice i al vértice j tendrá menor coste si se pasa por el vértice k que si el camino es directo de i a j. Por tanto, si la expresión anterior es verdadera el elemento $a[i,j]$ de la matriz se sustituirá por $a[i,k]+a[k,j]$.

Consideremos el grafo de la figura 4.11 y calculemos el camino más barato para viajar entre cada par de ciudades del grafo. En cada iteración sólo se presentarán aquellos casos que den lugar a actualizaciones de la matriz de adyacencia del grafo.

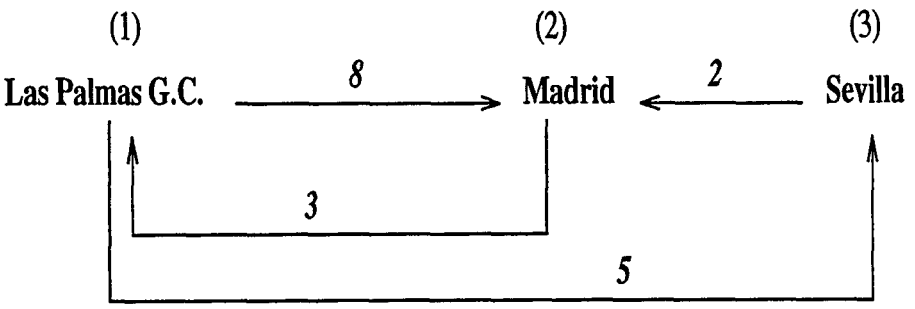


Figura 4.11: Grafo de ciudades y costos de vuelos

Inicialmente la matriz del grafo, A será la siguiente:

$$A = \begin{pmatrix} 0 & 8 & 5 \\ 3 & 0 & \infty \\ \infty & 2 & 0 \end{pmatrix}$$

Para la primera iteración, k=1 (lo que significa pasar del vértice i al j a través del vértice (1) Las Palmas G.C.), comprobamos que para ir de Madrid (vértice 2) a Sevilla (vértice 3) el camino más barato es Madrid(2)-Las Palmas G.C.(1)-Sevilla(3) (con coste 8), con lo cual la matriz queda de la forma:

$$A = \begin{pmatrix} 0 & 8 & 5 \\ 3 & 0 & 8 \\ \infty & 2 & 0 \end{pmatrix}$$

Para la segunda iteración, k=2 (pasar del vértice i al j pasando por vértice (2) Madrid) resulta más económico ir de Sevilla a Las Palmas G.C. pasando por Madrid. La matriz A actualizada será:

$$A = \begin{pmatrix} 0 & 8 & 5 \\ 3 & 0 & 8 \\ 5 & 2 & 0 \end{pmatrix}$$

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

En la tercera iteración observamos que resulta más barato ir de Las Palmas G. C. (1) a Madrid (2) pasando por Sevilla, actualizando en consecuencia la matriz A tendremos:

$$A = \begin{pmatrix} 0 & 7 & 5 \\ 3 & 0 & 8 \\ 5 & 2 & 0 \end{pmatrix}$$

En el algoritmo de *Floyd* podríamos tener una matriz, *Vertices*, cuyos elementos fuesen de la forma:

Vertices[i,j]=0 si existe camino mínimo directo de i a j
Vertices[i,j]=k si el camino mínimo de i a j pasa por el vértices k

Para nuestro ejemplo, la matriz *Vertices* resultante de aplicar el algoritmo de *Floyd* sería:

$$Vertices = \begin{pmatrix} 0 & 3 & 0 \\ 0 & 0 & 1 \\ 2 & 0 & 0 \end{pmatrix}$$

en la cual el elemento *Vertice[1,2]*=3 indica que para ir de Las Palmas G.C.(1) a Madrid (2) el camino más barato es a través de Sevilla (3); el camino de coste mínimo entre el vértice Madrid (2) y Sevilla (3) es pasando por Las Palmas G.C. (1) (*vertice[1,3]*=1); para ir de Sevilla (3) a Las Palmas G.C.(1), lo más barato es pasar por Madrid (2).

El algoritmo de *Floyd* para un grafo de *numvert* vértices con matriz de adyacencia *Costes*, se presenta a continuación:

```
constantes
   numvert=3 {número de vértices del grafo}
tipo
   matriz_enteros=matriz [1..numvert,1..numvert] de enteros

Procedimiento Inicializar_matrices(Costes,A,Vertices)
      {matriz_enteros Costes,A,Vertices por referencia}
variables
   enteras i,j
inicio
   desde i:=1 hasta numvert hacer
      desde j:=1 hasta numvert hacer
         A[i,j]:=Costes[i,j]
         Vertices[i,j]:=0
      fin desde
```

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
      A[i,i]:=0
   fin desde
fin procedimiento

Procedimiento FLOYD(Costes,A,Vertices)
      {matriz_enteros Costes,A,Vertices por referencia}
variables
   enteras i,j,k
inicio
   Inicializar_matrices(Costes,A,Vertices)
   desde k:=1 hasta numvert hacer
      desde i:=1 hasta numvert hacer
         desde j:=1 hasta numvert hacer
            si A[i,k]+A[k,j]<A[i,j] entonces
               A[i,j]:=A[i,k]+A[k,j]
               Vertices[i,j]:=k
            fin si
         fin desde
      fin desde
   fin desde
fin prcedimiento
```

4.2.2 RECORRIDOS DE GRAFOS DIRIGIDOS

Al igual que en los árboles existían diversos recorridos (preorden, inorden, postorden) en los grafos también se necesita una forma sistemática para recorrer los vértices.

4.2.2.1 RECORRIDO EN PROFUNDIDAD

El recorrido en profundidad de un grafo es una generalización del recorrido en preorden de un árbol. Este recorrido es la base en torno a la cual pueden construirse otros muchos algoritmos de tratamiento de grafos. Para recorrer un grafo en profundidad se considera inicialmente que todos los vértices del grafo no han sido visitados. Se parte de un vértice inicial y se recorren de forma recursiva todos los vértices adyacentes a ese vértice por los cuales no se haya pasado aún. A medida que se va pasando por los vértices, éstos se marcan como *visitados*. Si cuando el recorrido recursivo termina quedan vértices sin visitar, se toma el siguiente vértice *no visitado* y se repite el proceso hasta que todos los vértices del grafo hayan sido marcados a *visitados*.

Por ejemplo, vamos a realizar el recorrido en profundidad del grafo de la figura 4.12:

Si se parte del vértice 1, el algoritmo marca 1 como *visitado*. A continuación se selecciona el primer vértice adyacente no visitado al vértice 1, el vértice 2. Éste se marca como *visitado* y se busca su vértice adyacente no visitado, el vértice 3. El vértice 3 se marca como *visitado*. Dicho vértice es adyacente al vértice 1 pero éste ya ha sido visitado. En

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

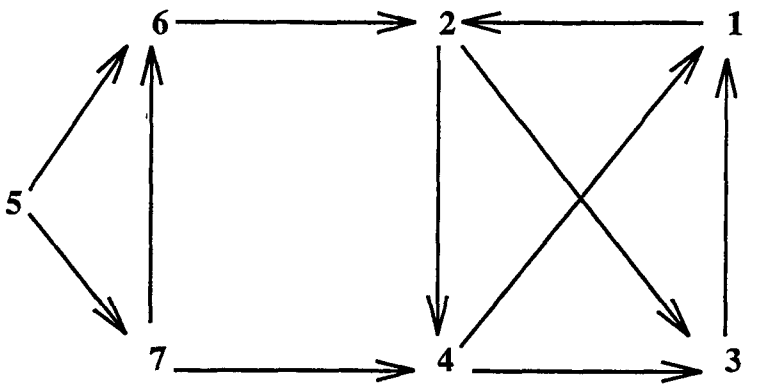


Figura 4.12: Grafo a recorrer en profundidad

este punto termina el recorrido recursivo que partió del 1. Por tanto, volvemos al vértice 2, el cual es adyacente al vértice 4, no visitado aún. Marcamos el vértice 4 como *visitado*. El vértice 4 es adyacente al 1 y al 3 que ya han sido visitados. Retrocedemos al vértice 1 pero éste no tiene más vértices adyacentes no visitados. El algoritmo selecciona el vértice 5 (vértice siguiente en numeración) que no ha sido visitado. Este vértice es adyacente al vértice 6 no visitado, el cual se marca como *visitado*. El vértice 6 es adyacente al 2 pero éste ya ha sido visitado. Por tanto, se retrocede al vértice 5 que tiene otro vértice adyacente no visitado, el 7. Este vértice tiene al 4 como adyacente pero éste ya ha sido visitado. En este punto termina el recorrido en profundidad ya que todos los vértices han sido visitados.

El algoritmo recursivo de recorrido en profundidad de un grafo de N vértices y natriz de adyacencia *Grafo*, se presenta a continuación. En el algoritmo la marca *visitado* equivale a 1 y la de *no visitado* a 0 en el vector *Visitado*.

```
AlgoritmoRecorrido_grafo_profundidad
constantes
   N=3 {número de vértices del grafo}
tipo
   matriz_enteros=matriz [1..N,1..N] de enteros
   vector_enteros=vector [1..N] de enteros

Procedimiento Recorrido_en_profundidad(vertice,Grafo,Visitado)
      {enteras vertice por valor}
      {matriz_enteros Grafo por referencia}
      {vector_enteros Visitado por referencia}
variables
   enteras k
inicio
   Visitado[vertice]:=1
   desde k:=1 hasta N hacer
      si (vertice<>k) y (Grafo[vertice,k]<>0) y (Visitado[k]=0) entonces
         Recorrido_en_profundidad(k,Grafo,Visitado)
      fin si
   fin desde
fin procedimiento
```

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
variables
   enteras i
   vector_enteros Visitado
   matriz_enteros Grafo
inicio
   Leer_matriz_adyacencia(Grafo) {Lectura matriz de adyacencia}
   desde i:=1 hasta N hacer
      Visitado[i]:=0
   fin desde
   desde i:=1 hasta N hacer
      si Visitado[i]=0 entonces
         Recorrido_en_profundidad(i,Grafo,Visitado)
      fin si
   fin desde
fin
```

4.2.2.2 TEST DE NO EXISTENCIA DE CIRCUITOS

Un *circuito* es un camino finito en el que el vértice final coincide con el vértice inicial.

Un circuito se dice que es *elemental* si no utiliza dos veces el mismo vértice, salvo el vértice final.

Se define *arco de retorno* en un recorrido en profundidad a aquel arco que va de un vértice a uno de sus predecesores o a él mismo.

Si tenemos un grafo dirigido en el que se desea saber si existen o no circuitos utilizaremos el recorrido en profundidad comprobando en cada momento que no existan arcos de retorno. Un arco es de retorno si mientras se lleva a cabo el recorrido en profundidad, el vértice sucesor de uno dado ya ha sido visitado.

En la figura 4.13 cuando el grafo se recorre en profundidad observamos que el vértice E tiene como sucesor al D, el cual ya ha sido visitado, puesto que es sucesor del B. De esta forma se detecta el circuito C-E-D-C en dicho grafo.

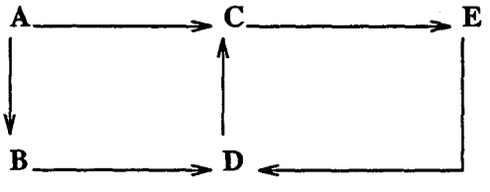


Figura 4.13: Grafo orientado con circuito

El algoritmo que nos permite determinar la existencia de circuitos en un grafo de N vértices y matriz de adyacencia *Grafo* es el siguiente (inicialmente el vector *Visitado* está a cero):

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
Algoritmo Test_existencia_circuitos
constantes
   N=5 {número de vértices del grafo}
tipo
   matriz_enteros=matriz [1..N,1..N] de enteros
   vector_enteros=vector [1..N] de enteros

Procedimiento Recorre_en_Profundidad(Grafo,Visitado,i,enc)
      {matriz_enteros Grafo por referencia}
      {vector_enteros Visitado por referencia}
      {logica enc por referencia}
      {enteras i por valor}
variables
   enteras j
inicio
   j:=1
   mientras (j<=N) y (no enc) hacer
      si (i<>j) y (Grafo[i,j]<>0) entonces
         si Visitado[j]=0 entonces
            Visitado[j]:=1
            Recorre_en_Profundidad(Grafo,Visitado,j,enc)
         si no
            enc:=verdadero
         fin si
      fin si
      j:=j+1
   fin mientras
fin procedimiento

Funcion Buscar_Circuitos(Grafo,Visitado) devuelve logico
      {matriz_enteros Grafo por referencia}
      {vector_enteros Visitado por referencia}
variables
   logica circ_enc
   enteras i
inicio
   circ_enc:=false
   i:=1
   mientras (i<=N) y (no circ_enc) hacer
      si Visitado[i]=0 entonces
         Visitado[i]:=1
         Recorre_en_profundidad(Grafo,Visitado,i,circ_enc)
      fin si
      i:=i+1
   fin mientras
   Buscar_Circuitos:=circ_enc
fin funcion
```

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
variables
   matriz_enteros Grafo
   vector_enteros Visitado
   enteras i,j
inicio
   desde i:=1 hasta N hacer
      Visitado[i]:=0
      desde j:=1 hasta N hacer
         leer(Grafo[i,j])
      fin desde
   fin desde
   si Buscar_Circuitos(Grafo,Visitado) entonces
      escribir('Grafo con circuitos')
   si no
      escribir('Grafo sin circuitos')
   fin si
fin
```

4.2.3 FUERTE CONEXIDAD

4.2.3.1 GRAFO FUERTEMENTE CONEXO

Un grafo se dice que es *fuertemente conexo* si para todo par de vértices del grafo existe al menos un camino entre ellos, de longitud uno o superior.

En la figura 4.14 a) es fuertemente conexo, ya que existe camino entre los vértices 1-2, 1-3 (a través del 2), 1-4 (pasando por 2-3), 2-3, 2-4 (pasando por 3), 3-4. El grafo 4.14 b) no es fuertemente conexo porque no existe conexión entre el vértice 1 y 4.

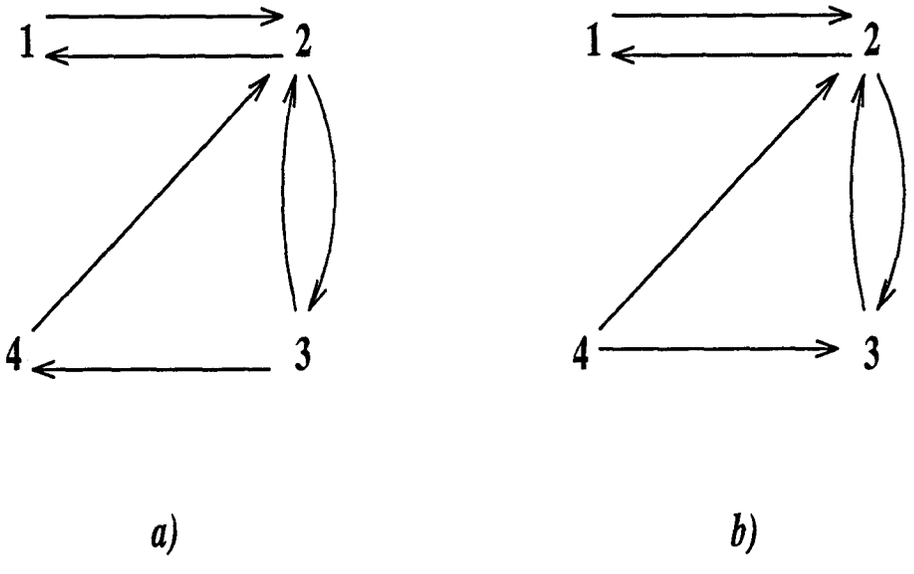


Figura 4.14: a) Grafo fuertemente conexo y b) grafo no fuertemente conexo

Para determinar si un grafo es o no fuertemente conexo inicialmente, almacenamos

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

la primera fila y columna de la matriz de adyacencia, en dos vectores. Ambos vectores se modificarán, poniendo sus componentes a uno, a medida que se vayan encontrando caminos de longitud uno o superior entre los vértices del grafo. Si después de haber recorrido todos los vértices del grafo, existe alguna componente de uno de esos vectores que esté a cero, el grafo no será fuertemente conexo. En caso contrario existe camino de longitud uno o superior entre cada uno de los vértices del grafo y por tanto, el grafo será fuertemente conexo.

Veamos el método empleado con un ejemplo. La matriz de adyacencia del grafo de la figura 4.14 a), *Grafo* es:

$$Grafo = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Inicialmente se almacenan la primera fila y columna de la matriz en dos vectores, *fila* y *columna* respectivamente:

$$fila = \begin{pmatrix} 0 & 1 & 0 & 0 \end{pmatrix}$$

$$columna = \begin{pmatrix} 0 & 1 & 0 & 0 \end{pmatrix}$$

Se repiten los siguientes pasos hasta que no sea posible actualizar ninguna de las componentes de los vectores *fila* o *columna*:

- *Tratamiento del vector fila*: *fila[2]*<>0 quiere decir que existe un arco cuyo extremo inicial es el 2, el cual estará en la fila 2 de la matriz de adyacencia del grafo. Analizamos los elementos en la segunda fila de *Grafo* para determinar los posibles extremos finales del arco. El elemento *Grafo[2,1]* es distinto de cero, luego existe un arco del vértice 2 al vértice 1. Éste es un posible extremo final del arco, por lo que ponemos *fila[1]* a 1 para indicarlo. Procedemos de la misma forma para el elemento *Grafo[2,3]*. El vector *fila* resultante será:

$$fila = \begin{pmatrix} 1 & 1 & 1 & 0 \end{pmatrix}$$

- *Tratamiento del vector columna*: el segundo elemento del vector *columna* es distinto de cero, es decir, existe un arco en el grafo con extremo final en el vértice 2, el cual estará en columna 2 de *Grafo*. En este caso, se trata de encontrar los posibles extremos iniciales de ese arco analizando la segunda columna de *Grafo*. Observamos que *Grafo[1,2]*, *Grafo[3,2]* y *Grafo[4,2]* son distintos de cero, luego los posibles extremos iniciales son los vértices 1, 3 y 4 y se ponen a 1 esas posiciones en el vector

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

columna para indicarlo. El vector *columna* resultante será:

$$columna = \begin{pmatrix} 1 & 1 & 1 & 1 \end{pmatrix}$$

- Volvemos al *tramiento del vector fila* y aunque el primer y el segundo elemento del vector son distintos de cero no dan lugar a ninguna actualización en *fila*. Sin embargo, *fila[3]*=1 y en este caso si se produce actualización. Ello es debido a que existe un arco con extremo inicial 3 y final 4, (*Grafo[3,4]*=1) en este caso se pone a 1 el cuarto elemento de *fila*. El vector *fila* quedará como:

$$fila = \begin{pmatrix} 1 & 1 & 1 & 1 \end{pmatrix}$$

- Como todos los elemntos del vector *fila* y *columna* están a 1 ya no se pueden llevar a cabo más actualizaciones, por tanto, el grafo en cuestión es fuertemente conexo. Con los dos vectores a 1 se refleja que entre cada par de vértices del grafo existe un camino de longitud uno o superior, por lo que se cumple que el grafo es fuertemente conexo.

A continuación se presenta una función que devuelve verdadero si un grafo de N vértices y matriz de adyacencia *Grafo* es fuertemente conexo:

```
constantes
   N=4 {número de vértices del grafo}
tipo
   matriz_enteros=matriz [1..N,1..N] de enteros
   vector_enteros=vector [1..N] de enteros

Procedimiento Tratamiento_fila(Grafo,fila,cambio,j)
      {matriz_enteros Grafo por referencia}
      {vector_enteros fila por referencia}
      {logica cambio por referencia}
      {entera j por valor}
variables
   enteras k
inicio
   si fila[j]<>0 entonces
      desde k:=1 hasta N hacer
         si (Grafo[j,k]=1) y (fila[k]=0) entoces
            fila[k]:=1
            si k<j entonces
               cambio:=verdadero
            fin si
         fin si
      fin desde
   fin si
fin procedimiento
```

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
ProcedimientoTratamiento_columna(Grafo,columna,cambio,j)
      {matriz_enteros Grafo por referencia}
      {vector_enteros columna por referencia}
      {logica cambio por referencia}
      {entera j por valor}
variables
   enteras k
inicio
   si columna[j]<>0 entonces
      desde k:=1 hasta N hacer
         si (Grafo[k,j]=1) y (columna[k]=0) entonces
            columna[k]:=1
            si k<j entonces
               cambio:=verdadero
            fin si
         fin si
      fin desde
   fin si
fin procedimiento

Procedimiento Inicializar_fila_columna(fila,columna,Grafo)
      {vector_enteros fila,columna por referencia}
      {matriz_enteros Grafo por referencia}
variables
   enteras i
inicio
   desde i:=1 hasta N hacer
      fila[i]:=Grafo[1,i]
      columna[i]:=Grafo[i,1]
   fin desde
fin procedimiento

Funcion Fuertemente_Conexo (Grafo) devuelve logico
      {matriz_enteros Grafo por referencia}
variables
   logica cambio
   vector_enteros fila, columna
   enteras j,k
inicio
   si N=1 entonces
      Fuertemente_Conexo:=verdadero
   si no
      Inicializar_fila_columna(fila,columna,Grafo)
      repetir
         cambio:=falso
         desde j:=1 hasta N hacer
            Tratamiento_fila(Grafo,fila,cambio,j)
            Tratamiento_columna(Grafo,columna,cambio,j)
         fin desde
      hasta que no cambio
```

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
      Fuertemente_Conexo:=verdadero
      desde k:=1 hasta N hacer
        si (fila[k]=0) o (columna[k]=0) entonces
          Fuertemente_Conexo:=falso
        fin si
      fin desde
    fin si
fin funcion
```

4.3 GRAFOS NO DIRIGIDOS

Un grafo no dirigido es aquel en el que las líneas que unen cada par de vértices no tienen dirección. En los grafos no dirigidos hablaremos de aristas en lugar de arcos.

Al igual que en la representación de grafos dirigidos las estructuras de datos utilizadas para representar los grafos no dirigidos serán la matriz de adyacencia y las listas enlazadas.

4.3.1 CONCEPTOS

A continuación se presentan algunos conceptos relacionados con grafos no dirigidos:

Cadena. Una cadena es una secuencia de aristas en la que cada arista está unida por un extremo a la arista anterior y por el otro a la arista siguiente si existe. Una cadena equivale a un camino en un grafo dirigido.

Cadena simple. Cadena que no utiliza una arista más de una vez.

Cadena elemental. Cadena que no utiliza un vértice más de una vez.

Ciclo. Cadena que parte de un vértice y llega a él mismo. Un ciclo equivale a un circuito en un grafo dirigido.

Grafo Conexo. Grafo tal que para cada par de vértices existe al menos una cadena entre ellos.

4.3.2 RECORRIDOS DE UN GRAFO NO DIRIGIDO

4.3.2.1 RECORRIDO EN PROFUNDIDAD

El recorrido en profundidad de un grafo dirigido puede aplicarse a un grafo no dirigido con una pequeña modificación ya que en los grafos no dirigidos una arista del vértice i al j viene dada por dos elementos en la matriz de adyacencia: $a[i,j]$ y $a[j,i]$. Por tanto,

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

el algoritmo para llevar a cabo el recorrido en profundidad de un grafo no dirigido de N vértices y matriz de adyacencia *Grafo* será de la forma:

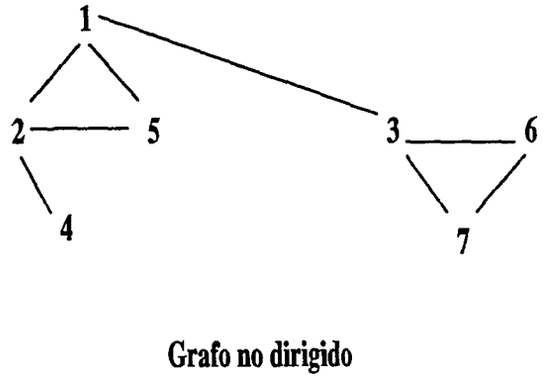
```
Algoritmo Recorrido_profundidad_nodirigido
constantes
   N=5 {número de vértices del grafo}
tipo
   matriz_enteros=matriz [1..N,1..N] de enteros
   vector_enteros=vector [1..N] de enteros

Procedimiento Recorrido_profundidad_nodirigido(vertice,Grafo,Visitado)
      {entera vertice por valor}
      {matriz_enteros Grafo por referencia}
      {vector_enteros Visitado por referencia}
variables
   enteras k
inicio
   Visitado[vertice]:=1
   desde k:=1 hasta N hacer
      si (Grafo[vertice,k]=1 o Grafo[k,vertice]=1) y (vertice<>k) y (Visitado[k]=0)
      entonces
         Recorrido_profundidad_nodirigido(k,Grafo,Visitado)
      fin si
   fin desde
fin procedimiento

variables
   enteras i
   vector_enteros Visitado
   matriz_enteros Grafo
inicio
   Leer_matriz_adyacencia(Grafo)
   desde i:=1 hasta N hacer
      Visitado[i]:=0
   fin desde
   desde i:=1 hasta N hacer
      si Visitado[i]=0 entonces
         Recorrido_profundidad_nodirigido(i,Grafo,Visitado)
      fin si
   fin desde
fin
```

4.3.2.2 RECORRIDO EN AMPLITUD

Otra forma de recorrer un grafo es mediante el recorrido en amplitud. En este recorrido para cada vértice se visitan todos sus vértices adyacentes, marcando éstos como *visitados*.

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

En la figura 4.15 se muestra un grafo no dirigido y las aristas por las que pasa el recorrido en amplitud desde el vértice 1. Partiendo del vértice 1 y siguiendo el orden de numeración, el primer vértice adyacente al 1 es el vértice 2. Por tanto, la primera arista recorrida en amplitud es la (1,2). El siguiente vértice en numeración adyacente al 1 es el vértice 3 y la siguiente arista del recorrido será (1,3). Después del 3 la siguiente arista es la (1,5). En este punto ya no hay más vértices adyacentes al vértice 1 y se continúa con los vértices no visitados adyacentes al vértice 2. La única arista recorrida desde el vértice 2 será la (2,4). Notar que aunque el vértice 2 es adyacente al 1, éste ya ha sido visitado por lo que no se visita de nuevo. Aplicando sucesivamente el recorrido obtendremos el conjunto de aristas de la figura 4.15, las cuales representan el recorrido en amplitud del grafo no dirigido de dicha figura.



Recorrido en amplitud

(1,2)
(1,3)
(1,5)
(2,4)
(3,6)
(3,7)

Figura 4.15: Recorrido en amplitud de grafo no dirigido

El procedimiento que lleva a cabo el recorrido en amplitud de un grafo no dirigido hace uso de un vector *Visitado* en el que se van marcando aquellos vértices que van siendo visitados. A medida que los vértices se van visitando se insertan en una cola, *Col*. Cuando se han visitado todos los vértices adyacentes a uno dado, se extrae el siguiente vértice de la cola y se determinan todos los vértices adyacentes al nuevo vértice. Así sucesivamente, hasta que la cola esté vacía. El procedimiento de recorrido en amplitud de un grafo no dirigido hace uso de los siguientes procedimientos y funciones:

Insertar_en_Cola(v,Col)	Procedimiento que inserta en la cola, *Col*, un vértice, *v*
v=Extraer_de_Cola(Col)	Función que extrae un vértice,*v*, del frente de la cola, *Col*
Insertar_arista(vi,vj,Aristas,numarista)	Procedimiento que inserta la arista *numarista*, formada por los vértices *vi* y *vj* en la matriz *Aristas*
bool=Vacia(Col)	Función que devuelve verdadero si la cola *Col* está vacía

A continuación se presenta el procedimiento que lleva a cabo el recorrido en amplitud de un grafo no dirigido, *Grafo* con N vértices.

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
constantes
   N=5 {número de vértices del grafo}
   M=10 {máximo número de aristas del recorrido}
tipo
   matriz_vertices=matriz [1..N,1..N] de enteros
   matriz_aristas=matriz [1..M,1..2] de enteros
   vector_enteros=vector [1..N] de enteros

Procedimiento Recorrer_en_amplitud(Grafo,Visitado,vinicial,Aristas)
      {matriz_vertices Grafo por referencia }
      {vector_enteros Visitado por referencia}
      {entera vinicial por valor}
      {matriz_aristas Aristas por referencia}
variables
   enteras i,j,numarista
inicio
   numarista:=0
   Visitado[vinicial]:=1
   Insertar_en_Cola(vinicial,Col)
   mientras no Vacia(Col) hacer
      i:=Extrae_de_Cola(Col)
      desde j:=1 hasta N hacer
         si (Grafo[i,j]=1 o Grafo[j,i]=1) y (i<>j) y (Visitado[j]=0) entonces
            Visitado[j]:=1
            Insertar_en_cola(j,Col)
            numarista:=numarista+1
            Insertar_arista(i,j,Aristas,numarista)
         fin si
      fin desde
   fin mientras
fin procedimiento
```

4.4 PROBLEMAS

1. En teoría de grafos se define el *semigrado exterior* de un vértice como el número de arcos incidentes hacia el exterior que tiene ese vértice. De la misma forma, se define el *semigrado interior* de un vértice como el número de arcos incidentes hacia el interior que tiene ese vértice. Diseñar un procedimiento que a partir de la matriz de adyacencia del grafo calcule el semigrado exterior e interior de un grafo.

   El método consistirá en sumar los elementos de la fila, *G[i,col]*, correspondiente al vértice i para obtener el semigrado exterior a ese vértice y sumar los elementos de la columna correspondiente al vértice, *G[fil, i]*, para obtener el semigrado interior del vértice i.

   ```
   constantes
      numvert=5 {número de vértices del grafo}
   ```

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
tipo
   grafo=vector [1..numvert,1..numvert] de enteros

Procedimiento Semigrados (G,vertice,semigrext,semigrin)
        {grafo G por referencia}
        {enteras vertice por valor}
        {enteras semigrext,semigrin por referencia}

variables
   enteras valorbucle,k
inicio
   {Guarda el valor del bucle porque lo va a poner a cero}
   valorbucle:=G[vertice,vertice]
   G[vertice,vertice]:=0
   semigrin:=0
   semigrext:=0
   desde k:=1 hasta numvert hacer
      semigrin:=semigrin+G[k,vertice]
      semigrext:=semigrext+G[vertice,k]
   fin desde
   {Restablece el valor del bucle}
   G[vertice,vertice]:=valorbucle
fin procedimiento
```

2. A partir de la matriz de adyacencia de un grafo, diseñar un procedimiento que calcule los vértices adyacentes a uno dado. Se dice que dos vértices, i y j, son adyacentes si *G[i,j]* o *G[j,i]* son distintos de 0, siendo G la matriz de adyacencia del grafo. Para determinar los vértices adyacentes a uno dado, i, tendremos que recorrer la fila i guardando los índices de los elementos distintos de cero y recorrer la columna i guardando los índices de los elementos distintos de cero.

```
constantes
   numvert=5 {número de vértices del grafo}
   M=6 {máximo número de vértices adyacentes}
tipo
   grafo=vector [1..numvert,1..numvert] de enteros
   vector_vertady=vector [1..M] de enteros

Procedimiento VerticesAdyacentes(G,Adyac,vertice,numvertady)
        {grafo G por referencia }
        {vector_vertady Adyac por referencia}
        {enteras vertice por valor}
        {enteras numvertady por referencia}
variables
   enteras valorbucle,i
inicio
   valorbucle:=G[vertice,vertice]
```

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
    {Ponemos a cero el valor del bucle correspondiente a vertice}
    G[vertice,vertice]:=0
    numvertady:=0
    desde i:=1 hasta numvert hacer
      si (G[vertice,i]<>0) o (G[i,vertice]<>0) entonces
        numvertady:=numvertady+1
        Adyac[numvertady]:=i
      fin si
    fin desde
fin procedimiento
```

3. Un grafo se dice que es *fuertemente conexo mínimo* si al eliminar uno cualquiera de sus arcos pierde su fuerte conexidad. Diseñar una función que devuelva verdadero si el grafo es fuertemente conexo mínimo y falso en caso contrario.

   El método a emplear consiste en eliminar uno a uno los arcos del grafo y comprobar si el grafo sigue siendo fuertemente conexo. Desde el momento en que al eliminar uno de los arcos el grafo deje de ser fuertemente conexo, el grafo examinado será fuertemente conexo mínimo.

```
constantes
  N=5 {número de vértices}
tipos
  matriz_enteros=matriz [1..N,1..N] de enteros
  vector_enteros=vector [1..N] de enteros

Funcion Fuert_Conexo_Minimo(Grafo) devuelve logico
    {matriz_enteros Grafo por referencia}
variables
  logica salir
  enteras i,j,temp
inicio
  salir:=falso
  i:=1; j:=1;
  mientras (i<= N) y (no salir) hacer
    mientras (j<= N) y (no salir) hacer
      si Grafo[i,j]>0 entonces
        {guardamos el arco para luego restaurarlo}
        temp:=Grafo[i,j]
        Grafo[i,j]:=0
        salir:=Fuertemente_Conexo(Grafo,numvert)
        si salir entonces
          Fuert_Conexo_Minimo:=falso
        fin si
        Grafo[i,j]:=temp
      fin si
```

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

```
        j:=j+1
      fin mientras
      i:=i+1
    fin mientras
    Fuert_Conexo_Minimo:=verdadero
fin funcion
```

4. Diseñar un procedimiento que determine el número de aristas de un grafo no dirigido y devuelva los vértices que componen cada arista en una matriz *Arista*.

```
constantes
   N=5 {número de vértices del grafo}
   M=10 {máximo número de aristas}
tipo
   matriz_vertices=matriz [1..N,1..N] de enteros
   matriz_aristas=matriz [1..M,1..2] de enteros
   vector_enteros=vector [1..N] de enteros

Procedimiento Cuenta_aristas (Grafo,Arista,num_arist)
      {matriz_vertices Grafo por referencia}
      {matriz_aristas Arista por referencia}
      {enteras num_arist por referencia}
variables
   enteras i,j,exist_arist
inicio
   exist_arist:=0
   desde i:=1 hasta N hacer
      desde j:=1 hasta N hacer
         {La función entera nos devulve la parte entera de la operación}
         exist_arist:=entera((Grafo[i,j]+Grafo[j,i])+1)/2)
         si exit_arist<>0 entonces
            num_arist:=num_arist+1
            Arista[mun_arist,1]:=i
            Arista[num_arist,2]:=j
         fin si
      fin desde
   fin desde
fin procedimiento
```

4.5 BIBLIOGRAFÍA

Apuntes de Informática. Escuela Universitaria de Informática. ULPGC. 1987.

[BERGE89] CLAUDE BERGE, *Graphs*. North-Holland Mathematical Library,1989.

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006

[EVEN79] SHIMON EVEN, *Graph Algorithms*. Computer Science Press, 1979.

[LIPSHUTZ87] SEYMOUR LIPSHUTZ, *Estructura de datos*, McGraw Hill, 1987.

[PEREZ-95] JUAN FCO. PÉREZ CASTELLANO, *Teoría de grafos aplicada*, Departamento de Electrónica y Telecomunicación. Escuela Técnica Superior de Ingenieros de Telecomunicación. ULPGC, 1995.

© Del documento, los autores. Digitalización realizada por ULPGC. Biblioteca Universitaria, 2006